Abstract

Nowadays equipment with Embedded systems is very common, it is normal to buy a fridge with heat sensor, light sensor to remind you when the door is not closed, also heaters, air conditioner and even cars either. Cars have very complicated computer system it is based on many MCUs each has unique and important task, tasks in automotive are usually critical and should not be delayed.

So, what would we do if there is a bug in one of the MCUs code? or even if the publish company has an update to keep its system useful and meet customer requirements?

we all know that we can reprogram any MCU by hard wiring to JTAG programmer or by connecting the MCU to pc that has Bootloader to this MCU, but do we really need our customers to waste their time and money to keep their devices updated?! time and money can be saved if hard wiring is replaced by another wireless way, this way should be able to flash the code on the car's MCU without interrupting the main program which is already running. so we need to send firmware over the air.

FOTA is abbreviation to firmware over the air, it means that the software will be sent to the target MCU in the car through wireless method and then be flashed and run on it in a secure way. FOTA is particularly useful when it comes to IOT systems, especially those with large numbers of connected devices that require frequent updates, **FOTA technology** allows manufactures to provide efficient and timely firmware updates for handsets, which increases customer satisfaction and reduces technical support requirements, it also allows manufactures to repair bugs in sold devices by remotely installation of new software.

In this project FOTA technology will be implemented from scratch, to do so we have to take in consideration some challenges, for example memory in target MCU should be able to keep the running project, the new installed one and also the bootloader, memory management and remapping should be done as the memory in embedded MCUs are partially limited so we have to use it efficiently, we also should make sure that the running program is not interrupted while flashing the new one to apply FOTA technology. another challenge is the design itself as we are working with many MCUs which are target MCU (car MCU) and telematics unit which receives the updated hex file from the server and send it to the target MCU, so we must program each of the MCUs separately then integrate all work.

Figure 1

Table of Contents

Table of Figures

Chapter 1 : 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 2

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.

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.

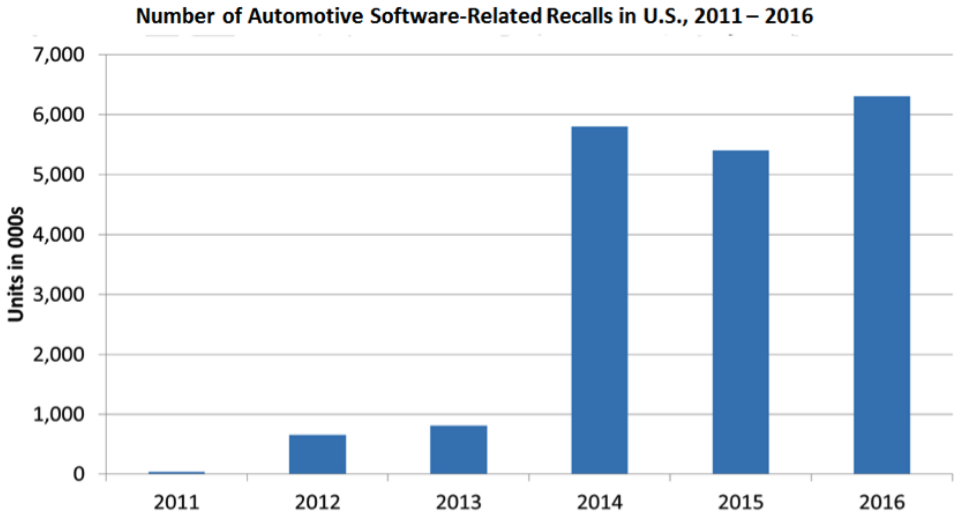


Figure 3

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.

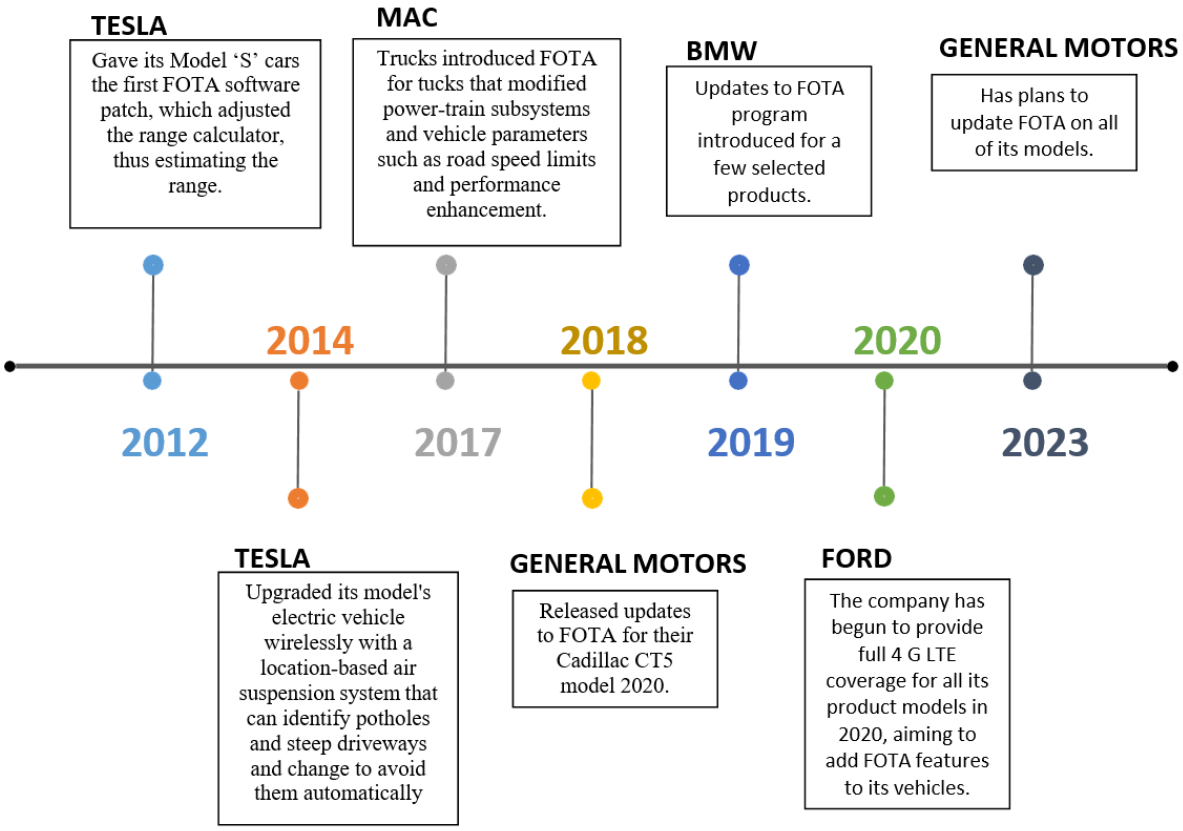


Figure 4

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.

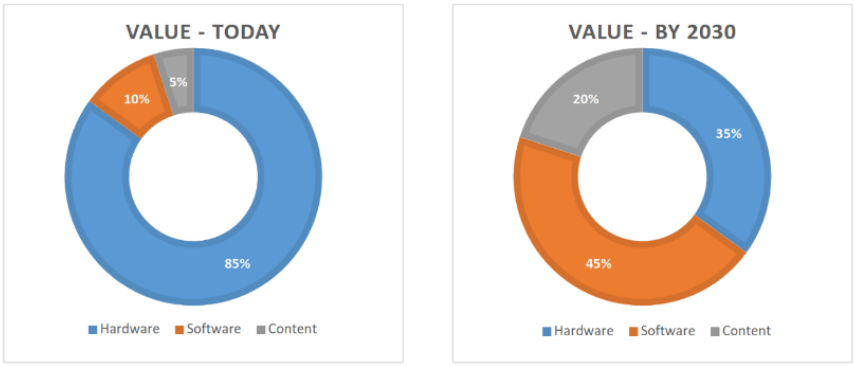


Figure 5

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.

Tesla Motors has started to deliver FOTA capability for its electric cars, which has shaken the automotive industry. Other global automakers, such as BMW, Mercedes-Benz and General Motors, then started investing in providing FOTA capabilities for their connected vehicles. FOTA updates are supposed to help OEMs save a large sum they spend as a result of car recalls. It also gives car owners the ease of updating all their software without having to visit the location of the dealer.

Robert Bosch GmbH, NXP Semiconductors N.V, Verizon Communications, Inc., Continental AG, Infineon Technologies AG, Qualcomm Incorporated, Intel Corporation, HARMAN International, Airbiquity Inc, Aptiv, HERE Technologies, BlackBerry QNX Software Systems Limited, Garmin Ltd. and Intellias Ltd are some of the major players on the market.

The global FOTA automotive updates market is characterized by the presence of numerous national, regional, and local suppliers. The market is highly competitive, competing with all players to achieve full market share.

According to the report of Market Research Future (MRFR), as of 2018, HARMAN International, Aptiv, QNX Computer Systems Limited, Intel Corporation and Airbiquity Inc. are some of the major players on the global FOTA automotive update market. By growth, mergers & acquisitions, and by providing a broad product range, these businesses continue to maintain their strong global grip.

Another main announcement in the market that Qualcomm Technologies, Inc., a subsidiary of Qualcomm Incorporated, previously confirmed that its 5G test networks would be extended to include new end-to - end over-the-air (FOTA) systems for both (mmWave) and sub-6 GHz bands.

While Verizon Introduces Hum, an all-in-one connected car solution that provides users with vehicle position, diagnostics, on-road assistance, travel history and more. It's the first 4G LTE connected car solution to have Google Assistant built-in.

With a CAGR of 18.1% from 2021 to 2030, the global automobile over-the-air (OTA) market, which was valued at $2.59 billion in 2020, is anticipated to grow to $13.71 billion by 2030.

The COVID-19 outbreak harshly impacted the automotive sector on a global level, which in turn resulted in a substantial drop in vehicle sales and the insufficiency of raw materials. Many industry players in the automotive sector have witnessed issues such as a halt of production activities and mandated plant closures by the government. However, the pandemic resulted in an indirect effect on the automotive over-the-air (OTA) market.

At the mid-pandemic industry leaders are to formulate different ways to integrate critical information such as contact information, a list of nearest hospitals/medical centers, and quick reference information for primary symptoms with official local governmental guidance along with enhanced user experience. Adoption of such methods is foreseen to reinforce the demand for automotive over the air (OTA) in near future.

The need for automotive over-the-air updates is anticipated to increase due to factors like the rise in connected vehicle adoption, the rise in demand for electric vehicles, government regulations regarding vehicle safety and cyber security, and government initiatives for implementing connected car technology. However, high over-the-air update costs and a lack of infrastructure in developing nations may limit growth potential.

Figure 6

Based on type, the automotive over-the-air updates market had a greater revenue share for software in 2020. The OTA software update technology is a more effective approach for OEMs to update the software and repair issues than the traditional method. Easy connectivity between embedded systems and connected devices, which offers vital information like the location of fuel and charging stations and the status of the battery and charging, in addition to facilitating remote diagnosis of all types of connected vehicles, is the main factor causing this category to dominate the market.

In terms of market share for automobile over-the-air upgrades in 2020, North America dominated. The region's growing use of connected automobiles is largely responsible for this leadership position. With mobile devices becoming increasingly compatible with these vehicles, there is a huge demand for networking, navigation, communication, and entertainment services. Additionally, regional OEMs and

technology providers continually invest sizable sums in the development of sturdy technologies, which is drawing in buyers for vehicles. For instance, Tesla Inc. offers a module for troubleshooting as well as routinely downloading OTA software upgrades that improve and add to the current capabilities over Wi-Fi. Major automakers are being encouraged to turn their focus to linked vehicles and either develop new variants or include cutting-edge connection capabilities into the existing models as a result of the growing acceptance of internet of things (IoT) technology. This is a response to people's rising desire for cutting-edge navigation, telematics, and infotainment systems. For proper operation, these systems depend on the software that is placed on the embedded hardware of the vehicle. Thus, key development in the market for automotive over-the-air updates is the increasing adoption of smart, connected technologies.

Figure 7

Because of the speed at which the automotive industry is going digital, it follows that to stay up with the most recent developments in connectivity technology, vehicles must regularly receive software upgrades, including cybersecurity patches. Installing these updates wirelessly or OTA is faster and more cost-effective than repeatedly calling the sold vehicles back to a garage. Additionally, OTA updates support IoT environments by improving functionality, simplifying the introduction of new features, and securing devices from hacking, data breaches, and other forms of cyber-attacks. These additional advantages of OTA updates serve as a key industry driver.

The global automotive OTA updates market is anticipated to be driven by the tight standards relating to vehicle and passenger safety. The primary priority for automotive manufacturers worldwide has always been vehicle safety. It has become crucial for manufacturers to adapt their products to the current vehicle standards as a result of the implementation of greater safety laws and monitoring standards. Additionally, automakers are concentrating quickly on upgrading their vehicles to ensure compliance with the updated norms and regulations due to the rising sophistication of automotive technology and the increasing degree of programmability.

Chapter 2 : Project Overview

2.1 Introduction

The Firmware Over-The-Air (FOTA) project utilizing STM Electronic Control Units (ECUs), ESP32, and a comprehensive web server integrated with a mobile application represents a cutting-edge approach to remote firmware updates for embedded systems. This project is designed to streamline the process of updating the firmware of various devices without requiring physical access, thereby enhancing operational efficiency and reducing maintenance costs. The STM ECUs, known for their robust performance and reliability in automotive and industrial applications, serve as the primary control units for the embedded systems. These ECUs manage critical functions and require periodic firmware updates to ensure optimal performance and security. By leveraging the ESP32, a powerful and versatile microcontroller with built-in Wi-Fi and Bluetooth capabilities, the project enables seamless communication between the ECUs and the web server. The ESP32 acts as a bridge, receiving firmware updates from the server and securely transmitting them to the ECUs. The web server, which hosts the firmware files, plays a pivotal role in the FOTA process. It not only stores the latest firmware versions but also manages the distribution and version control, ensuring that each ECU receives the appropriate update. Additionally, the web server's interface allows administrators to monitor the update process, track the status of each ECU, and troubleshoot any issues that may arise. Complementing this system is a mobile application, which provides a user-friendly interface for remote management. Through the app, users can initiate firmware updates, receive notifications about the update status, and access diagnostic information in real-time. This mobile app enhances the user experience by providing convenience and flexibility, enabling updates to be performed from virtually anywhere. The integration of STM ECUs, ESP32, and the web server with a mobile app in this FOTA project not only simplifies the firmware update process but also ensures that the embedded systems remain secure and up-to-date with the latest features and improvements. This project exemplifies the synergy between hardware and software, highlighting the potential of IoT solutions in modern embedded system management.

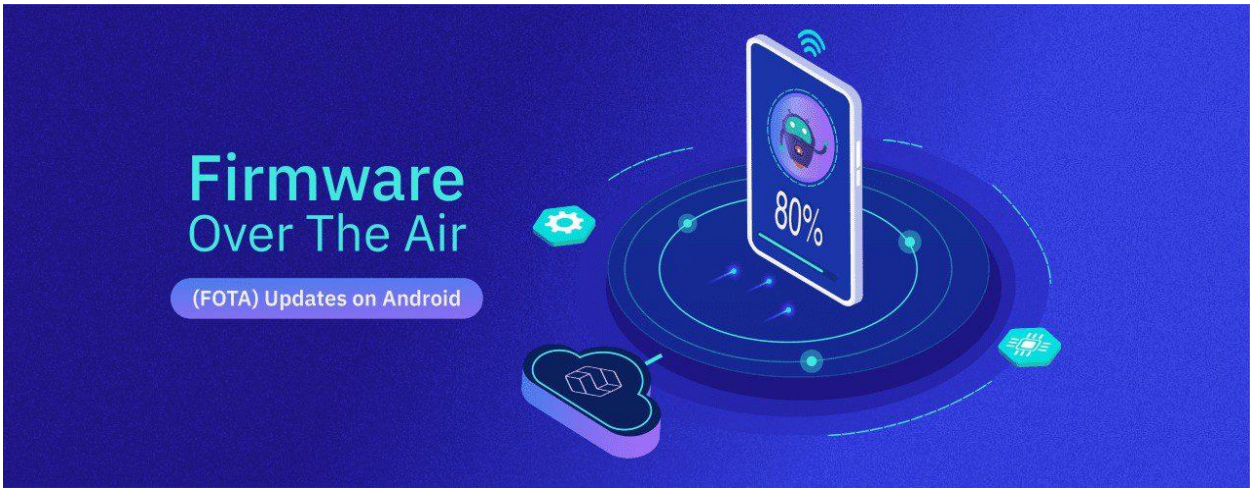


Figure 8

2.2 System Design

2.2.1 Block Diagram

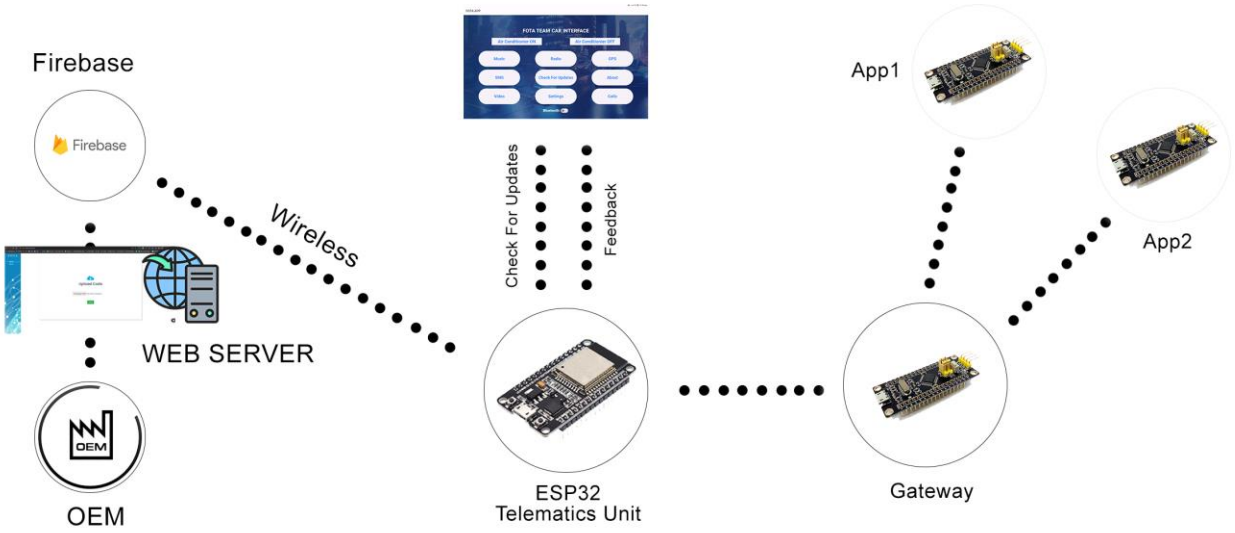


Figure 9

The central component of the system is the ESP32 microcontroller, embedded in the car's electronic control unit (ECU). This microcontroller is responsible for managing the firmware update process. It interacts with several other components, including a web server, Firebase cloud storage, and a mobile application.

- The process begins with the web server, which acts as the interface for uploading new firmware versions. Developers or authorized personnel can access this server to upload the latest firmware files. Once uploaded, these files are stored in Firebase, a robust and scalable cloud storage solution. Firebase not only stores the firmware files but also manages metadata such as version numbers and update timestamps, ensuring that the ESP32 always has access to the latest firmware information.
- The ESP32 periodically checks Firebase for any available updates. This check involves querying the metadata to determine if a new firmware version is available. If an update is found, the ESP32 notifies the user via a mobile application. The mobile app, designed with a user-friendly interface, allows the user to decide whether to proceed with the update immediately or delay it for later. The user can interact with the app by pressing a button labeled "Update" or "Later," sending a command back to the ESP32.
- Upon receiving the user's command to update, the ESP32 initiates the download of the new firmware from Firebase. It ensures the integrity and completeness of the download before beginning the update process. The ESP32 uses a secure and reliable method to write the new firmware into its memory, minimizing the risk of corruption or failure.

Additionally, it includes a rollback mechanism to revert to the previous firmware version if the update process encounters any issues.

This block diagram not only illustrates the flow of data and control between components but also highlights the modular and secure nature of the FOTA update system. Each component is designed to perform its specific function efficiently, ensuring that the entire system operates smoothly and reliably. The integration of cloud storage, microcontroller, and mobile application demonstrates the convergence of modern technologies to enhance the functionality and user experience of automotive firmware updates.

2.2.2 Abstract View

- **Web Server**

The web server plays a critical role in the FOTA update system by acting as the interface through which new firmware versions are uploaded and managed. Authorized developers or personnel can access the web server to upload the latest firmware files. Once a file is uploaded, the web server ensures it is correctly stored and indexed in Firebase. The server also handles user authentication and authorization, ensuring that only legitimate users can upload or modify firmware files. This centralized approach simplifies the management of firmware updates, providing a secure and efficient method for distributing new versions to all connected devices. The web server's integration with Firebase enables seamless updates and robust data handling capabilities, essential for maintaining the integrity and reliability of the FOTA system.

- **Firebase Cloud Storage**

Firebase Cloud Storage serves as the backbone for storing and managing firmware files in the FOTA update system. Once the firmware is uploaded via the web server, Firebase securely stores the files and manages the associated metadata, such as version numbers, file sizes, and timestamps. This metadata is crucial for the ESP32 microcontroller to determine whether a new firmware update is available. Firebase's robust and scalable infrastructure ensures that the firmware files are always accessible to the ESP32, regardless of the number of devices or geographical distribution. Additionally, Firebase's security rules and authentication mechanisms protect the firmware files from unauthorized access, ensuring that only legitimate updates are applied to the connected devices.

- **ESP32 Microcontroller**

The ESP32 microcontroller is the heart of the FOTA update system, embedded within the car's electronic control unit (ECU). Its primary function is to manage the firmware update process, ensuring that the car's software remains up-to-date with the latest features and security patches. The ESP32 periodically checks Firebase for any available firmware updates by querying the stored metadata. If a new update is found, the ESP32 downloads the firmware and verifies its integrity before proceeding with the installation. This process includes a secure and reliable

method for writing the new firmware into memory, along with a rollback mechanism to revert to the previous version in case of any issues. The ESP32's ability to handle over-the-air updates minimizes the need for physical interventions, enhancing the car's maintenance efficiency and user convenience.

- ## Mobile Application

The mobile application provides a user-friendly interface for car owners to manage firmware updates. When the ESP32 detects a new firmware version in Firebase, it notifies the user through the mobile app. The app presents the user with options to either update the firmware immediately or delay the update for a later time. This interaction is facilitated through simple buttons labeled "Update" and "Later." The mobile app communicates the user's decision back to the ESP32, which then takes the appropriate action. This interface ensures that users are always in control of the update process, allowing them to schedule updates at their convenience and minimizing disruption to their driving experience. The app's integration with Firebase and the ESP32 ensures seamless and efficient communication, enhancing the overall user experience and reliability of the FOTA system.

- ## Gateway

The Gateway STM (System-on-a-Chip) serves as a critical intermediary within the FOTA update system, facilitating communication between various components and ensuring the seamless integration of the ESP32 microcontroller, mobile application, and additional subsystems within the car. It acts as a central hub that routes data and commands appropriately, maintaining the overall integrity and efficiency of the system. The Gateway STM is responsible for managing the data flow between the ESP32 and other car subsystems, such as the autoparking and air conditioning modules. It ensures that firmware updates are correctly received and implemented across all relevant systems, providing a unified approach to managing the car's software environment. By leveraging the capabilities of the Gateway STM, the FOTA system achieves a higher level of coordination and reliability, essential for maintaining the optimal performance of the car's various functions.

- ## AutoParking Application

The AutoParking application, managed by its dedicated STM module, is an advanced feature within the car's electronic control unit, enhancing the vehicle's autonomy and convenience. This application uses sensors and algorithms to assist in parking the vehicle automatically, without requiring significant input from the driver. The FOTA update system ensures that the AutoParking application is always equipped with the latest software, improving its performance, accuracy, and safety features over time. By regularly updating the firmware, the system can introduce new capabilities, fix potential bugs, and enhance the overall user experience. The AutoParking STM works closely with the Gateway STM to receive updates and implement them seamlessly, ensuring that the autoparking functionality remains reliable and efficient.

- **Air Conditioning with Timer (Manual and Timer Mode)**

The introduction of the timer feature was facilitated through a FOTA update. This process involved creating a new version of the firmware with the timer functionality integrated. The updated firmware was then packaged and prepared for over-the-air delivery. Implementing a secure and reliable FOTA delivery mechanism was crucial to ensure that the firmware update could be transmitted and applied without any issues. The microcontroller was programmed to receive the update, verify its integrity, and apply the new firmware, all without user intervention. This capability highlights the significant advantages of FOTA technology, allowing for remote updates and enhancements to be made to embedded systems without the need for physical access. Extensive testing was conducted to validate the updated firmware, ensuring that the timer feature worked as intended and that the update process was smooth and error-free.

2.2.3 Hardware Components

The FOTA (Firmware Over-The-Air) update system for cars relies on a set of robust hardware components to ensure efficient and reliable operation. The key hardware components include the ESP32 microcontroller and three STM32F401 Blackpill microcontrollers, each serving specific functions within the system.

- **ESP32 Microcontroller:** The ESP32 is a powerful and versatile microcontroller that serves as the central hub for managing the firmware update process. Equipped with built-in Wi-Fi and Bluetooth capabilities, the ESP32 can connect to the internet to check for firmware updates stored in Firebase Cloud Storage. It periodically downloads new firmware versions, verifies their integrity, and initiates the update process. The ESP32's ability to handle over-the-air updates reduces the need for physical interventions, enhancing the efficiency and convenience of maintaining the car's software.

Figure 10

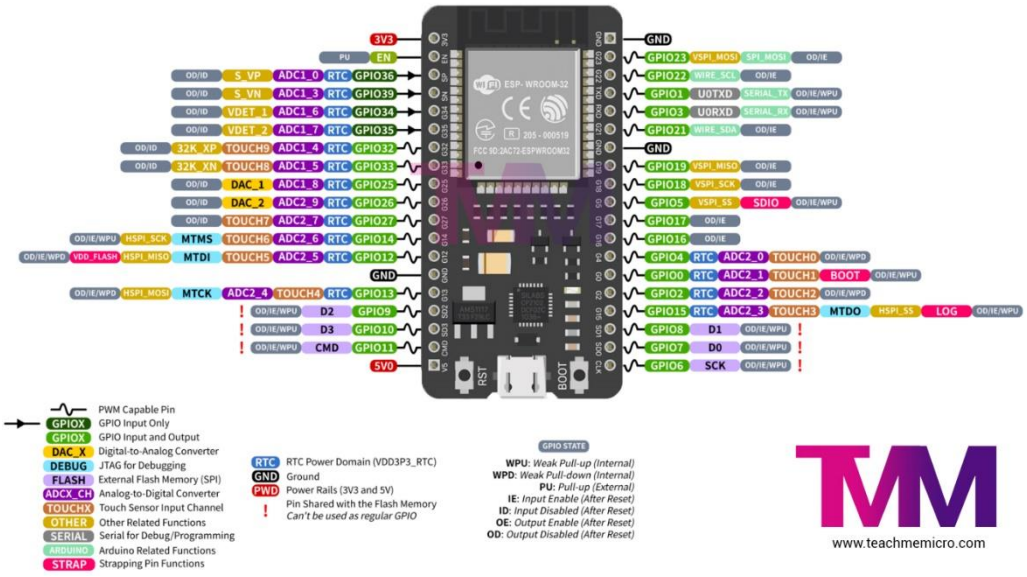


Figure 11

ESP32 Specs:

- Single or Dual-Core 32-bit Microprocessor with clock frequency up to 240 MHz.
- 520 KB of SRAM, 448 KB of ROM and 16 KB of RTC SRAM.
- Supports 802.11 b/g/n Wi-Fi connectivity with speeds up to 150 Mbps.
- Support for both Classic Bluetooth v4.2 and BLE specifications.
- 34 Programmable GPIOs.
- Up to 18 channels of 12-bit SAR ADC and 2 channels of 8-bit DAC
- Serial Connectivity include 4 x SPI, 2 x I2C, 2 x I2S, 3 x UART.
- Ethernet MAC for physical LAN Communication (requires external PHY).
- 1 Host controller for SD/SDIO/MMC and 1 Slave controller for SDIO/SPI.
- Motor PWM and up to 16-channels of LED PWM.
- Secure Boot and Flash Encryption.
- Cryptographic Hardware Acceleration for AES, Hash (SHA-2), RSA, ECC and RNG.

- **STM32F401 Black Pill Microcontrollers:** Three STM32F401 Black pill microcontrollers are used in the system, each dedicated to a specific function:

    - **Gateway STM32F401**: The Gateway STM32F401 acts as the intermediary between the ESP32 and the other subsystems within the car. It facilitates the communication and data flow between the ESP32, the autoparking application, and the air conditioning application. By managing this communication, the Gateway STM32F401 ensures that firmware updates are correctly distributed and implemented across all relevant systems.

Figure 12

    - **AutoParking STM32F401**: The AutoParking STM32F401 is dedicated to managing the car's autoparking functionality. This microcontroller processes sensor data and executes algorithms to assist in parking the vehicle automatically. By receiving firmware updates through the FOTA system, the AutoParking STM32F401 can continuously improve its performance, accuracy, and safety features.
    - **Air Conditioning STM32F401**: The Air Conditioning STM32F401 controls the vehicle's air conditioning system, providing both manual and automatic modes of operation. In manual mode, users can set their desired temperature and airflow preferences, while in automatic mode, the system adjusts settings dynamically based on cabin temperature sensors. Firmware updates delivered through the FOTA system enhance the air conditioning system's capabilities and efficiency, ensuring a comfortable cabin environment.

These hardware components work in harmony to create a seamless and reliable FOTA update system, ensuring that the car's software is always up-to-date with the latest features and security enhancements.

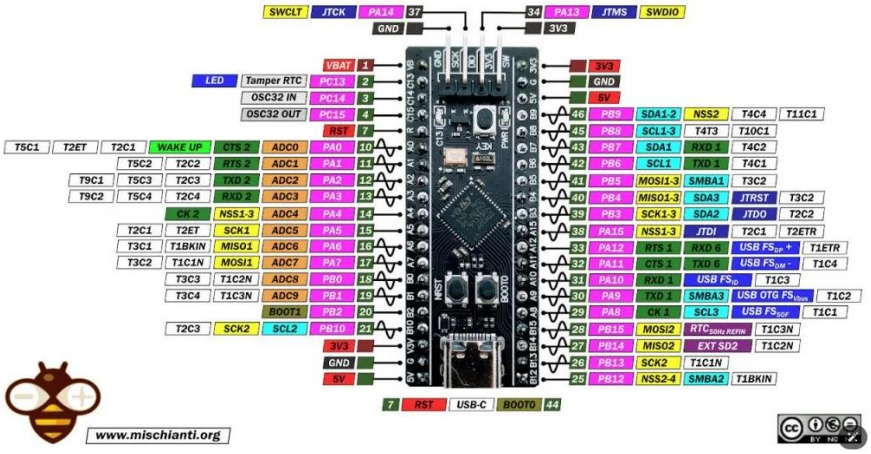
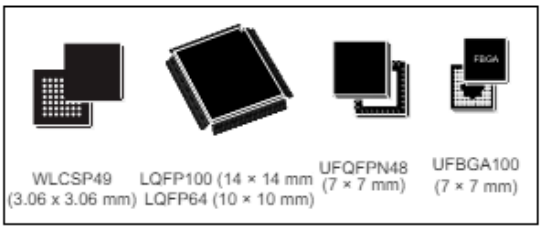


Figure 13

STM32F401 Specs:

Features

- Core: ARM® 32-bit Cortex®-M4 CPU with FPU, Adaptive real-time accelerator (ART Accelerator™) allowing 0-wait state execution from Flash memory, frequency up to 84 MHz, memory protection unit, 105 DMIPS/1.25 DMIPS/MHz (Dhrystone 2.1), and DSP instructions
- Memories
  - up to 512 Kbytes of Flash memory
  - up to 96 Kbytes of SRAM
- Clock, reset and supply management
  - 1.7 V to 3.6 V application supply and I/Os
  - POR, PDR, PVD and BOR
  - 4-to-26 MHz crystal oscillator
  - Internal 16 MHz factory-trimmed RC
  - 32 kHz oscillator for RTC with calibration
  - Internal 32 kHz RC with calibration
- Power consumption
  - Run: 146 µA/MHz (peripheral off)
  - Stop (Flash in Stop mode, fast wakeup time): 42 µA Typ @ 25C; 65 µA max @25 °C
  - Stop (Flash in Deep power down mode, fast wakeup time): down to 10 µA @ 25 °C; 30 µA max @25 °C
  - Standby: 2.4 µA @25 °C / 1.7 V without RTC; 12 µA @85 °C @1.7 V
  - V_{BAT} supply for RTC: 1 µA @25 °C
- 1×12-bit, 2.4 MSPS A/D converter: up to 16 channels
- General-purpose DMA: 16-stream DMA controllers with FIFOs and burst support
- Up to 11 timers: up to six 16-bit, two 32-bit timers up to 84 MHz, each with up to four IC/OC/PWM or pulse counter and quadrature (incremental) encoder input, two watchdog timers (independent and window) and a SysTick timer



- Debug mode
  - Serial wire debug (SWD) & JTAG interfaces
  - Cortex®-M4 Embedded Trace Macrocell™
- Up to 81 I/O ports with interrupt capability
  - Up to 78 fast I/Os up to 42 MHz
  - All I/O ports are 5 V-tolerant
- Up to 12 communication interfaces
  - Up to 3 x I^2C interfaces (SMBus/PMBus)
  - Up to 3 USARTs (2 x 10.5 Mbit/s, 1 x 5.25 Mbit/s), ISO 7816 interface, LIN, IrDA, modem control)
  - Up to 4 SPIs (up to 42Mbit/s at f_{CPU} = 84 MHz), SPI2 and SPI3 with muxed full-duplex I^2S to achieve audio class accuracy via internal audio PLL or external clock
  - SDIO interface
  - Advanced connectivity: USB 2.0 full-speed device/host/OTG controller with on-chip PHY
- CRC calculation unit
- 96-bit unique ID
- RTC: subsecond accuracy, hardware calendar
- All packages (WLCSP49, LQFP64/100, UFQFPN48, UFBGA100) are ECOPACK®2

Table 1. Device summary

Reference	Part number
STM32F401xD	STM32F401CD, STM32F401RD, STM32F401VD
STM32F401xE	STM32F401CE, STM32F401RE, STM32F401VE

- ## Auto parking Application System:

The AutoParking system is a sophisticated feature designed to enhance the convenience and safety of parking a vehicle. It leverages a combination of advanced hardware components, including ultrasonic sensors, DC motors, and servo motors, all orchestrated by the STM32F401 microcontroller. The STM32F401 acts as the brain of the autoparking system, processing sensor data and executing precise control commands to navigate the vehicle into a parking spot.

Ultrasonic sensors are strategically placed around the vehicle to detect obstacles and measure distances to surrounding objects. These sensors provide real-time feedback to the STM32F401 microcontroller, enabling it to make informed decisions about the car's position and trajectory. The high accuracy and reliability of ultrasonic sensors are crucial for safely maneuvering the vehicle in tight spaces and avoiding collisions.

Figure 14

DC motors are used to control the movement of the car's wheels, allowing it to move forward, backward, and perform complex maneuvers required for parking. The STM32F401 microcontroller regulates the speed and direction of the DC motors based on input from the ultrasonic sensors. This precise control ensures smooth and accurate movements during the parking process.

Figure 15

A **servo motor** is typically employed to control the steering mechanism of the vehicle. The servo motor receives commands from the STM32F401 microcontroller to adjust the angle of the front wheels, enabling the car to make precise turns. The high torque and positional accuracy of the servo motor are essential for executing the tight steering adjustments needed to navigate into a parking space.

Figure 16

Together, these hardware components create an integrated autoparking system that automates the parking process, reducing the need for driver intervention and minimizing the risk of parking-related accidents. By continuously receiving and implementing firmware updates via the FOTA system, the autoparking system can improve its algorithms and performance, ensuring it remains reliable and effective in various parking scenarios.

- ## Air Conditioner System:

The Air Conditioning system in this project is designed to enhance the comfort and convenience of the vehicle's occupants by providing both manual and automatic modes of operation. The system is controlled by an STM32F401 microcontroller and is simulated using an LED to indicate the status of the air conditioning (AC).

In **Manual mode**, the system allows users to control the Air Conditioning directly through a mobile application on a tablet. When the user clicks the "Air Conditioning On" button in the app, a command is sent to the STM32F401 microcontroller, which then turns on the LED, simulating the activation of the AC. Conversely, clicking the "Air Conditioning Off" button sends a command to turn off the LED, indicating the AC is off. This straightforward control mechanism ensures that users can easily manage their comfort settings as needed.

The **FOTA (Firmware Over-The-Air) update system** plays a crucial role in enhancing the functionality of the air conditioning system by enabling timer operation based on environmental conditions. Once the system receives a firmware update, it can operate in timer mode using data from serial that make it be adjusted to run for a specific time only like half an hour or one complete hour

By continuously updating the firmware via the FOTA system, the air conditioning system can adapt to changing requirements and incorporate new features, ensuring it remains efficient and user-friendly. The integration of both manual and timer modes provides a versatile solution for maintaining an optimal cabin environment.

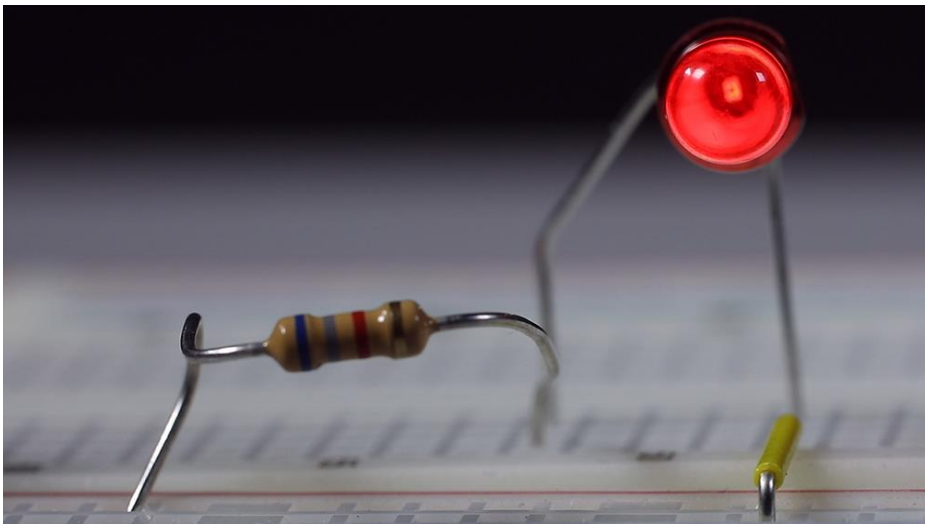


Figure 17

2.2.4 Software Components

- The FOTA (Firmware Over-The-Air) update system relies on two critical software components: the web server and the mobile application. Together, these components form a robust system for managing and facilitating firmware updates across vehicles. The web server acts as the central hub, hosting the latest firmware versions and managing distribution to vehicles based on their unique identifiers and update schedules. It provides a secure interface for administrators to upload new firmware, schedule deployments, and monitor update progress. On the other hand, the mobile application serves as the interface for vehicle owners or operators, allowing them to initiate updates, receive notifications, and monitor the status of ongoing updates. This tandem approach ensures that vehicles' electronic systems are always equipped with the latest enhancements and features, improving functionality, security, and user experience seamlessly.

- **WebServer:**

The web server serves as the central platform for managing and distributing firmware updates. Built using robust and scalable web technologies, the server provides an interface for developers and authorized personnel to upload new firmware versions. Once uploaded, the firmware files are securely stored and indexed in Firebase Cloud Storage. The web server handles user authentication and authorization to ensure that only legitimate users can upload or modify firmware files, maintaining the security and integrity of the update process. By managing the metadata associated with each firmware version, such as version numbers and update timestamps, the web server ensures that the ESP32 microcontroller can accurately determine the availability of new updates. This centralized management approach simplifies the deployment of firmware updates, making the process efficient and reliable.

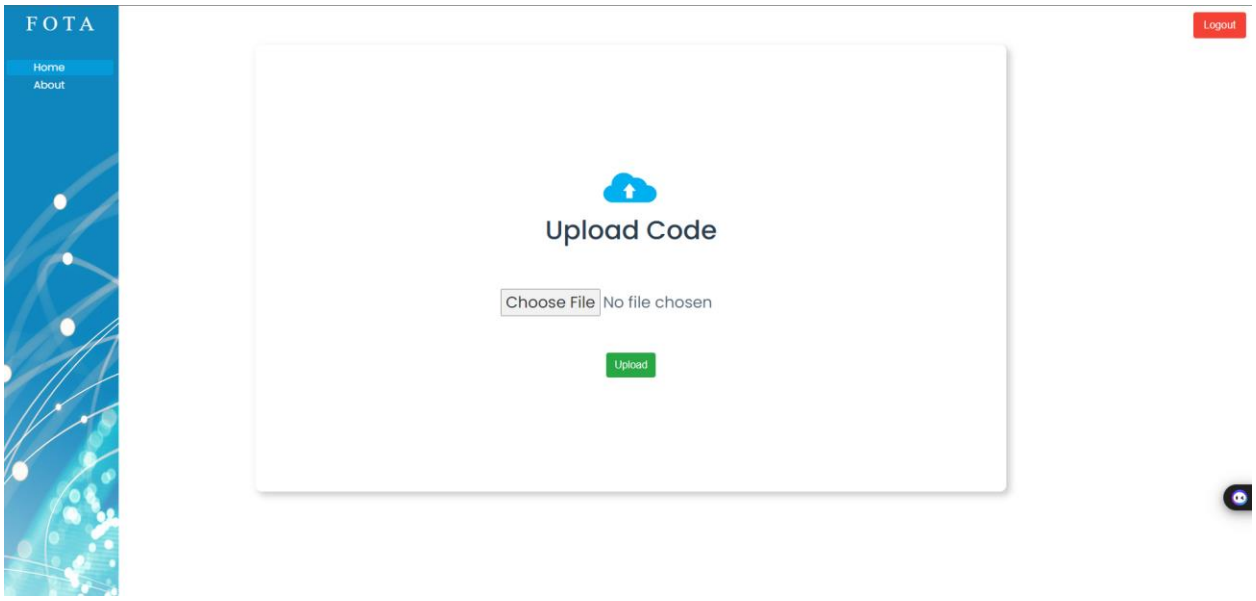


Figure 18

- **GUI (Mobile APP):**

The mobile application provides a user-friendly interface for car owners to interact with the FOTA system. Available on tablets or smartphones, the app notifies users when a new firmware update is available. Users can then choose to proceed with the update immediately or delay it for later, providing flexibility and convenience. In addition to managing firmware updates, the mobile app allows users to control the air conditioning system manually. By clicking the "Air Conditioning On" or "Off" buttons, users can directly send commands to the STM32F401 microcontroller to turn the system on or off. This intuitive control ensures that users can easily manage their comfort settings. Furthermore, the mobile app facilitates automatic updates, ensuring that the vehicle's systems operate optimally and efficiently. The app's seamless integration with Firebase and the ESP32 microcontroller ensures that firmware updates are applied smoothly, enhancing the overall reliability and functionality of the FOTA system.

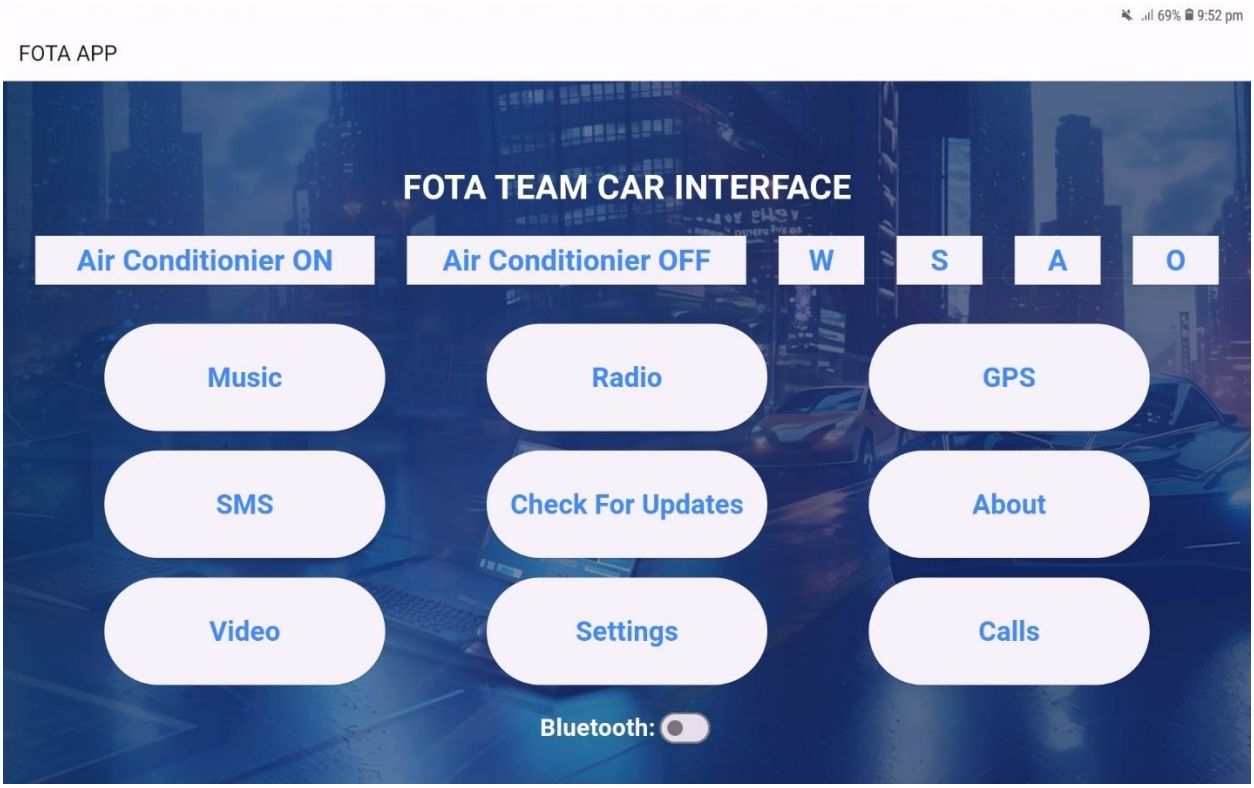


Figure 19

Together, these software components create a cohesive ecosystem that supports the seamless distribution and management of firmware updates, ensuring that the vehicle's electronic systems are always equipped with the latest advancements.

2.3 Software Design

2.3.1 Software Design Process

The software design process for the FOTA (Firmware Over-The-Air) update system is a structured and iterative approach that ensures the development of reliable, efficient, and user-friendly software components. This process encompasses several key stages, each aimed at addressing specific aspects of the system's functionality and integration.

Requirements Gathering and Analysis: The process begins with gathering detailed requirements from stakeholders, including developers, automotive engineers, and end-users. This phase involves understanding the needs for firmware updates, user interface design, security considerations, and system performance. By thoroughly analyzing these requirements, the development team establishes clear objectives and constraints for the project.

System Architecture Design: Based on the requirements, the team designs a comprehensive system architecture. This architecture outlines the interactions between the web server, mobile application, Firebase Cloud Storage, and the hardware components (ESP32 and STM32F401 microcontrollers). The architecture ensures that all components communicate effectively and that the system can handle firmware updates seamlessly. It also includes considerations for scalability, security, and fault tolerance.

Component Design: In this stage, detailed designs for each software component are created. For the web server, this includes defining API endpoints, database schemas, and security protocols. The mobile application design focuses on user interface (UI) and user experience (UX), ensuring that the app is intuitive and easy to navigate. Detailed flowcharts and pseudocode may be used to visualize the logic and interactions within each component.

Implementation: With the designs in place, the development team begins coding the software components. The web server is implemented using a backend framework such as Node.js or Django, while the mobile app is developed using a platform like Flutter, React Native, or native languages (Java/Kotlin for Android, Swift for iOS). During this phase, the team adheres to best practices in coding standards, version control, and documentation to maintain code quality and facilitate collaboration.

Testing and Validation: Comprehensive testing is conducted to ensure the software components function as intended. This includes unit testing individual functions, integration testing to verify interactions between components, and system testing to validate the entire FOTA update process. User acceptance testing (UAT) is also performed to ensure the software meets end-user expectations. Any identified issues are addressed through iterative debugging and refinement.

Deployment and Maintenance: After successful testing, the software is deployed to production environments. Continuous monitoring and maintenance are performed to ensure the system operates reliably. The team also remains responsive to user feedback and evolving requirements,

implementing updates and improvements as needed. This ongoing maintenance ensures that the FOTA system remains effective and up-to-date with the latest technological advancements.

By following this structured software design process, the development team ensures that the FOTA update system is robust, secure, and user-friendly, effectively meeting the needs of both developers and end-users.

Figure 20

2.3.2 Design Considerations

Designing the FOTA (Firmware Over-The-Air) update system for cars involves careful consideration of several key factors to ensure functionality, reliability, and user satisfaction. These design considerations encompass both technical and user-oriented aspects, shaping the overall architecture and implementation of the system.

Security: Security is paramount in the design of the FOTA system to protect against unauthorized access and potential vulnerabilities. Measures such as encryption of firmware files during transmission, secure authentication protocols for user access to the web server, and strict validation of firmware integrity before installation on the ESP32 microcontroller are essential. Implementing robust security practices ensures that the system can withstand potential cyber threats and safeguard sensitive vehicle data.

Scalability: As automotive technology advances, the FOTA system must accommodate future scalability requirements. This includes designing the system architecture to handle a growing number of vehicles and users seamlessly. Scalability considerations extend to Firebase Cloud Storage, ensuring it can manage increasing volumes of firmware files and metadata efficiently. By planning for scalability from the outset, the system can adapt to evolving demands without compromising performance.

Reliability: The reliability of the FOTA system is critical to maintain the continuous operation of vehicle functions. Designing redundant mechanisms for firmware distribution and installation, implementing error handling and recovery strategies in the ESP32 microcontroller firmware update process, and conducting thorough testing under various conditions are essential. Ensuring high availability of the web server and Firebase Cloud Storage further enhances system reliability, minimizing downtime and disruptions to vehicle operations.

User Experience: User experience (UX) is a key consideration in the design of both the mobile application and the overall FOTA system. The mobile app should provide an intuitive interface for users to receive update notifications, manage firmware installations, and control vehicle functions such as air conditioning. Designing clear and responsive UI elements, optimizing app performance for different device platforms, and incorporating user feedback into iterative improvements contribute to a positive and engaging user experience.

Compatibility and Integration: The FOTA system must integrate seamlessly with existing automotive systems and technologies. Compatibility with different vehicle models, adherence to industry standards for firmware updates, and interoperability with third-party applications are considerations. Ensuring that the ESP32 microcontroller firmware aligns with hardware specifications and capabilities, and supporting backward compatibility with older vehicle models, enhances the system's versatility and usability across diverse automotive environments.

Regulatory Compliance: Compliance with regulatory standards and industry guidelines is essential to ensure the safety, legality, and ethical use of the FOTA system. Adhering to data privacy regulations, automotive safety standards (such as ISO 26262), and industry best practices for software development and cybersecurity helps mitigate risks and maintain trust among stakeholders. Regular audits and compliance checks ensure ongoing adherence to regulatory requirements throughout the system lifecycle.

By carefully addressing these design considerations, the FOTA update system can deliver secure, scalable, reliable, and user-friendly firmware updates for vehicles, enhancing both operational efficiency and user satisfaction in the rapidly evolving automotive industry.

2.3.3 Design Approaches

Designing an effective FOTA (Firmware Over-The-Air) update system for automotive applications requires careful consideration of various design approaches to ensure scalability, reliability, security, and user experience. Here are key design approaches to consider:

1. Modular Architecture: Adopting a modular architecture allows the FOTA system to be divided into independent, manageable components. Each component, such as the web server, mobile application interface, firmware management module, and communication protocols with vehicle electronics, can be developed, tested, and updated separately. This approach facilitates scalability and flexibility, enabling easy integration of new features and components as the system evolves.

2. Over-The-Air Update Mechanism: Central to the FOTA system is its ability to deliver firmware updates wirelessly to vehicles. Designing a robust over-the-air (OTA) update mechanism involves implementing secure protocols for firmware distribution, validation, and installation. Techniques such as differential updates (only sending the changes between firmware versions) and rollback mechanisms (reverting to a previous firmware version in case of issues) enhance efficiency and reliability.

3. Secure Communication Protocols: Ensuring secure communication between the web server, mobile application, and vehicle electronics is critical. Implementing encrypted communication channels using protocols like HTTPS, MQTT (Message Queuing Telemetry Transport), or secure WebSocket connections protects sensitive data from unauthorized access and tampering. Authentication mechanisms, such as OAuth or token-based authentication, verify the identity of users and devices accessing the system.

4. Real-Time Monitoring and Diagnostics: Incorporating real-time monitoring and diagnostic capabilities into the FOTA system enables proactive management of firmware updates and system health. Monitoring tools can track update progress, detect anomalies during installation, and provide alerts for potential issues. Diagnostic features allow for remote troubleshooting and debugging, minimizing downtime and enhancing reliability.

5. User-Centric Design: Designing the web interface and mobile application with a focus on user experience (UX) is essential for user acceptance and satisfaction. Intuitive interfaces, clear status notifications about update availability and progress, and straightforward controls for scheduling or deferring updates contribute to a positive user experience. Accessibility considerations ensure that the system is usable by individuals with varying technical skills.

6. Compliance with Standards and Regulations: Adhering to industry standards, regulations, and best practices is crucial for ensuring the safety, reliability, and legal compliance of the FOTA system. Standards such as ISO 26262 (automotive functional safety), GDPR (General Data Protection Regulation), and cybersecurity guidelines specific to automotive applications help mitigate risks and build trust with stakeholders.

7. Continuous Improvement and Feedback Loop: Implementing a continuous improvement process allows the FOTA system to evolve over time. Gathering user feedback, conducting post-update surveys, and analyzing system performance metrics provide insights for refining features, enhancing security measures, and optimizing the update process. Iterative updates and enhancements based on real-world usage ensure that the FOTA system remains responsive to user needs and technological advancements.

By adopting these design approaches, developers can create a robust and efficient FOTA update system that enhances the functionality, reliability, and security of automotive firmware updates, ultimately improving vehicle performance and user satisfaction.

Two types of Design Models:

- **Top-Down Design Approach:**

1. **Comprehensive System View**: Starting with the top-down approach provides a holistic view of the entire FOTA system architecture and functionality. This approach ensures that all essential components and interactions are considered from the outset, aligning the system design with overarching goals and user requirements.
2. **Early Identification of Requirements**: By defining functional requirements early in the design process, stakeholders can prioritize key functionalities such as OTA update scheduling, security protocols, and user interface design. This clarity helps in avoiding scope creep and ensures that development efforts are focused on delivering essential features.
3. **Robust Security Planning**: Security considerations are addressed proactively in the top-down approach. By defining encryption standards, authentication mechanisms, and access control policies upfront, the system can mitigate potential security risks effectively. This approach aligns with regulatory compliance requirements and builds trust among users regarding data protection.
4. **Scalability and Performance Optimization**: Planning for scalability from the beginning allows the system to accommodate future growth in the number of vehicles and firmware updates seamlessly. Designing for performance optimization, such as load balancing and database indexing, ensures that the FOTA system operates efficiently under varying workload conditions.
5. **User-Centric Design**: The top-down approach emphasizes user experience (UX) design early in the process. By defining intuitive user interfaces and functionalities that meet user expectations, developers can create a user-friendly system that enhances usability and acceptance among vehicle owners and service personnel.

- **Bottom-Up Design Approach:**

1. **Incremental Development and Testing**: The bottom-up approach enables developers to focus on implementing and testing specific software components independently. This iterative process allows for early detection and resolution of technical challenges and integration issues, reducing overall development time and improving system reliability.

2. **Focused Component Optimization**: Developers can optimize individual components, such as firmware distribution algorithms or communication protocols, to perform efficiently and reliably. Fine-tuning these components ensures that they meet performance expectations and seamlessly integrate into the larger system framework.
3. **Flexible Integration**: After thorough testing and validation of individual components, integration into the overall FOTA system architecture becomes smoother and less prone to unexpected complications. This approach fosters flexibility in adapting to evolving requirements and technological advancements during the development lifecycle.
4. **Responsive Deployment and Maintenance**: Deploying validated components in stages allows for early deployment of critical functionalities while ongoing development continues. This iterative deployment approach enables continuous monitoring, maintenance, and updates to address emerging issues and incorporate user feedback effectively.
5. **Empirical Performance Validation**: Through performance testing and optimization at the component level, developers gain empirical insights into system behavior and resource utilization. This data-driven approach facilitates informed decisions regarding system architecture enhancements and scalability improvements.

2.3.4 Software Design Types

Software design encompasses various types or methodologies that guide the process of conceptualizing, planning, and implementing software systems. Each design type offers different approaches to organizing and structuring software components. Here are some common software design types:

1. **Structural Design**: Focuses on how the software components are organized and interconnected to form the overall structure of the system. This includes defining the relationships between modules, classes, and subsystems. Structural design ensures that the software is modular, maintainable, and scalable.
2. **Object-Oriented Design (OOD)**: Emphasizes the use of objects and classes to represent entities and their interactions within the system. OOD promotes encapsulation, inheritance, and polymorphism, enabling developers to model real-world concepts more effectively and facilitate code reuse.
3. **Functional Design**: Centers on defining the functions or operations that the software system must perform. Functional design specifies the behavior and logic of the system without focusing on the internal structure of components. It often involves decomposing requirements into discrete functions and defining how these functions interact.
4. **Data-Oriented Design**: Focuses on organizing and managing data within the software system. It includes designing data structures, databases, and data flows to ensure efficient

storage, retrieval, and manipulation of data. Data-oriented design is crucial for systems that handle large volumes of data or require complex data processing.

5. **Architectural Design**: Concerns the high-level structure of the software system, defining its components, interactions, and dependencies. Architectural design decisions, such as choosing between monolithic, client-server, microservices, or layered architectures, impact scalability, performance, and system integration.
6. **Component-Based Design**: Involves building software systems from reusable components or modules. Components encapsulate specific functionalities and can be assembled to create larger systems. Component-based design promotes modularity, flexibility, and code reusability, facilitating easier maintenance and updates.
7. **Service-Oriented Design (SOA)**: Focuses on designing software components as independent services that communicate via standardized protocols (e.g., REST, SOAP). SOA promotes loose coupling between services, allowing them to be developed, deployed, and scaled independently. It supports interoperability and flexibility in distributed systems.
8. **User Interface (UI) Design**: Deals with designing the visual and interactive aspects of the software's user interface. UI design aims to create intuitive, aesthetically pleasing interfaces that enhance usability and user experience. It involves layout design, navigation flows, information architecture, and usability testing.
9. **Database Design**: Specifically focuses on designing the structure and organization of databases that store and manage application data. Database design includes defining tables, relationships, indexes, and constraints to ensure efficient data storage, retrieval, and integrity.
10. **Iterative and Incremental Design**: Emphasizes an iterative approach to software development, where designs evolve through multiple cycles of prototyping, testing, and refinement. Incremental design involves delivering software in small, manageable increments or iterations, allowing for continuous improvement based on feedback and changing requirements.

Each software design type offers unique advantages and is chosen based on the specific requirements, complexity, and constraints of the software project. Effective software design combines elements from different types to create well-structured, maintainable, and scalable systems that meet user needs and business objectives.

Chapter 3 : Web Server

3.1 Introduction

Designing and implementing a web server using Node.js that integrates seamlessly with Firebase for file uploads represents a sophisticated solution poised at the intersection of modern web technologies and cloud-based database management. This robust architecture enables efficient file management and storage, catering specifically to the dynamic demands of contemporary software development practices. **Node.js**, renowned for its event-driven, non-blocking I/O model, empowers the web server with scalability and high performance. Leveraging its asynchronous capabilities, Node.js efficiently handles concurrent file uploads to Firebase, ensuring responsive and uninterrupted service for users. This server-side JavaScript runtime environment not only streamlines development but also enhances the server's ability to manage multiple tasks simultaneously, making it ideal for real-time applications like file handling. **Firebase**, Google's scalable real-time NoSQL cloud database, serves as the backbone of the web server's file storage system. Its seamless integration with Node.js allows for secure and reliable storage of uploaded files, facilitated through Firebase Storage. This cloud-based solution offers robust security features and automatic scaling, ensuring that the web server can handle varying workloads efficiently without compromising data integrity or performance. The combination of Node.js and Firebase represents a contemporary approach to web server development, where agility, scalability, and reliability converge to meet the demands of modern applications. By utilizing Node.js's lightweight and efficient runtime environment alongside Firebase's powerful database capabilities, developers can create a flexible and responsive platform for managing and storing files securely in the cloud.

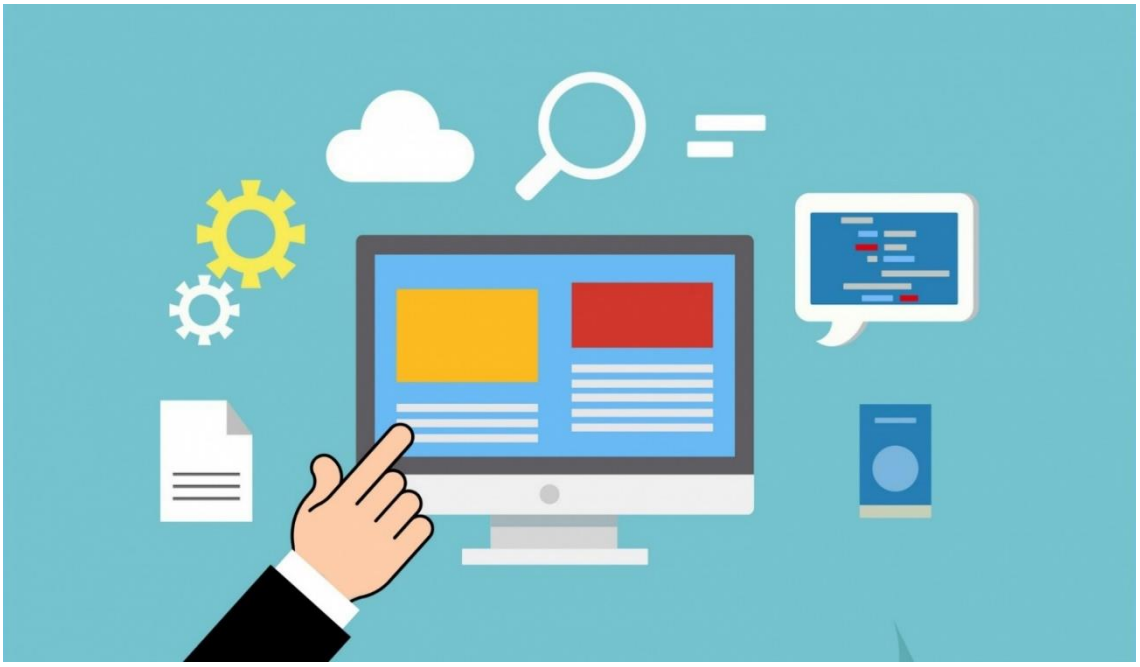


Figure 21

3.2 Webserver Design

Designing a robust web server involves integrating a backend powered by Node.js with Firebase as the database, creating a versatile and efficient foundation for modern web applications. This setup leverages the strengths of both technologies to handle data management, server-side logic, and real-time communication effectively.

- ### Node.js Backend

Node.js serves as the core backend technology, providing several key advantages:

- **Asynchronous and Non-blocking**: Node.js uses an event-driven, non-blocking I/O model, making it highly efficient for handling concurrent requests and I/O operations. This capability ensures that the server remains responsive even under heavy loads, enhancing scalability.
- **JavaScript Everywhere**: Node.js allows developers to use JavaScript for both frontend and backend development, promoting code reuse, consistency, and faster development cycles. This full-stack JavaScript approach simplifies the learning curve and facilitates seamless integration between frontend and backend components.
- **Rich Ecosystem**: Node.js benefits from a vast ecosystem of npm (Node Package Manager) modules and libraries, offering solutions for various functionalities such as web frameworks (e.g., Express.js), database drivers, authentication middleware, and utility modules. This extensive ecosystem accelerates development by providing pre-built components and tools.

- ### Firebase Database

Firebase serves as the backend-as-a-service (BaaS) database, offering real-time data synchronization and cloud storage capabilities:

- **Real-time Database**: Firebase Realtime Database provides synchronization across clients in milliseconds, enabling seamless data updates and real-time collaboration in applications. This feature is particularly beneficial for chat applications, collaborative tools, and live data streaming.
- **NoSQL Database**: Firebase uses a NoSQL data model, which is flexible and scalable for handling unstructured or semi-structured data. This flexibility allows developers to adapt data structures as application requirements evolve without rigid schema constraints.
- **Authentication and Security**: Firebase integrates authentication services, including email/password authentication, social login providers (e.g., Google, Facebook), and custom authentication mechanisms. It also offers built-in security rules to control access to data based on user roles and permissions, ensuring data integrity and user privacy.

- ## Vue.js Frontend

Vue.js is employed as the frontend framework, offering a dynamic and interactive user experience:

- **Reactive Data Binding**: Vue.js features a reactive data-binding system that automatically updates the user interface when the underlying data changes. This is particularly powerful when combined with Firebase's real-time data updates, ensuring the frontend stays in sync with the backend effortlessly.
- **Component-Based Architecture**: Vue.js's component-based architecture allows developers to build modular, reusable components that encapsulate their own logic and styling. This modularity simplifies the integration with Node.js backend APIs and Firebase, as each component can independently manage its own data fetching and state management.
- **Seamless Communication**: Vue.js components can make API calls to the Node.js server to request data, which is then fetched from Firebase. This data flow is streamlined by Vue.js's efficient state management, ensuring a responsive and dynamic user experience.

- ## Integration and Functionality

By utilizing Node.js as the backend runtime, integrating it with Firebase as the cloud database, and employing Vue.js for the frontend, developers can build scalable, real-time web applications that deliver responsive performance and seamless data synchronization. This architecture empowers developers to focus on application logic and user experience, leveraging the strengths of JavaScript and cloud computing to create dynamic and data-driven applications efficiently. The synergy between Node.js, Firebase, and Vue.js provides a comprehensive solution for modern web development, offering the tools needed to build robust, scalable, and interactive applications.

Figure 22

3.3 Webserver Overview

- **Login Page**

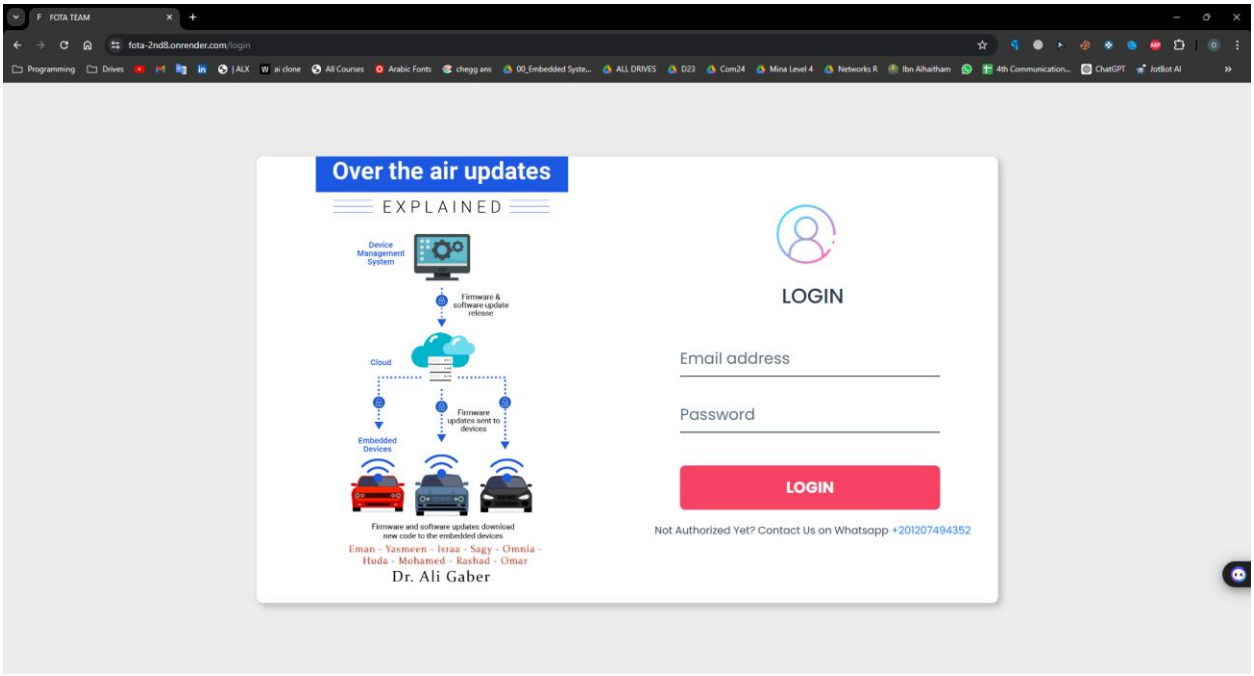


Figure 23

The login page is a critical component of the web application, providing a secure and user-friendly interface for users to authenticate themselves. Designed with simplicity and efficiency in mind, the login page features a clean layout with fields for entering a username or email and a password. A prominent login button is available to submit the credentials for authentication. To enhance user experience, the page includes visual cues and feedback, such as error messages for incorrect credentials or prompts for forgotten passwords. Additionally, it may offer options for users to register a new account or reset their password. The design prioritizes security and usability, ensuring that users can easily and safely access their accounts while maintaining a smooth and intuitive user experience.

- **Main Upload Page**

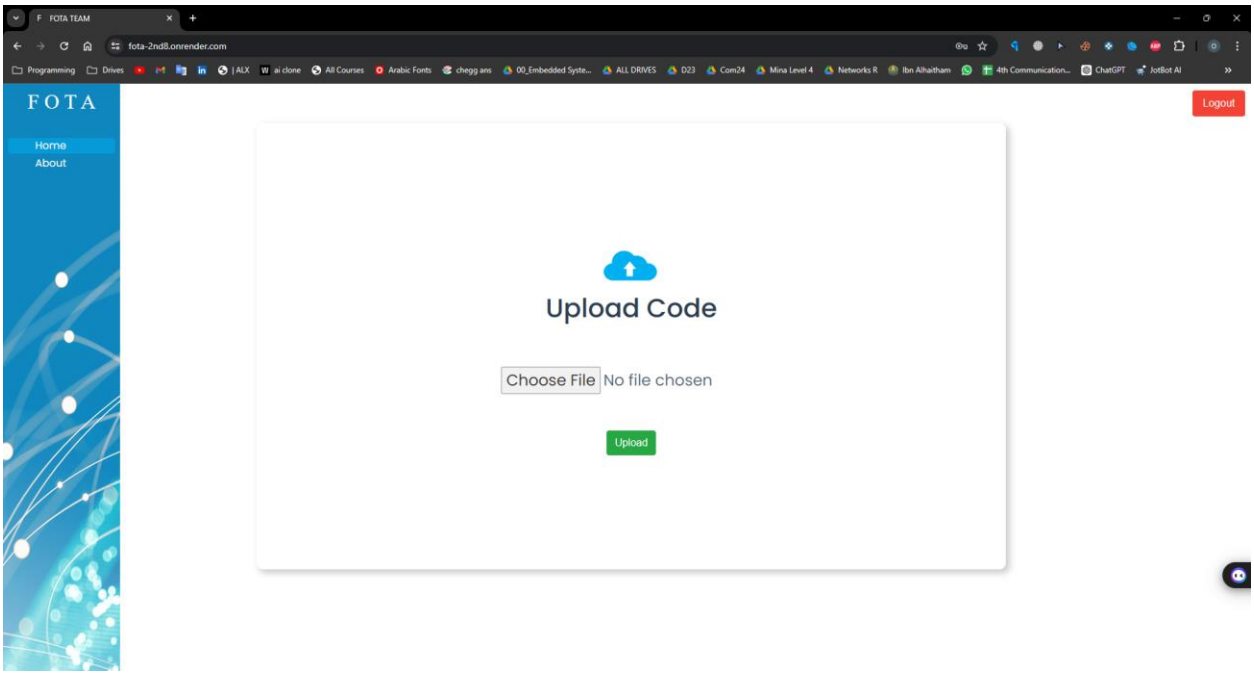


Figure 24

The main upload page serves as the primary interface for administrators and developers to upload firmware or code updates. This page is designed with simplicity and user-friendliness in mind, ensuring that the process of selecting and uploading files is straightforward and efficient. The upload page includes a clean, intuitive form where users can browse and select the firmware file from their local system. Upon selection, users can initiate the upload process with a single click of the upload button. To enhance user experience, the page features real-time feedback mechanisms such as progress bars that visually indicate the status of the upload, and notifications that inform users of the successful completion or failure of the upload process. The design emphasizes clarity and ease of use, providing a seamless and efficient workflow for managing firmware updates.

- **About Page**

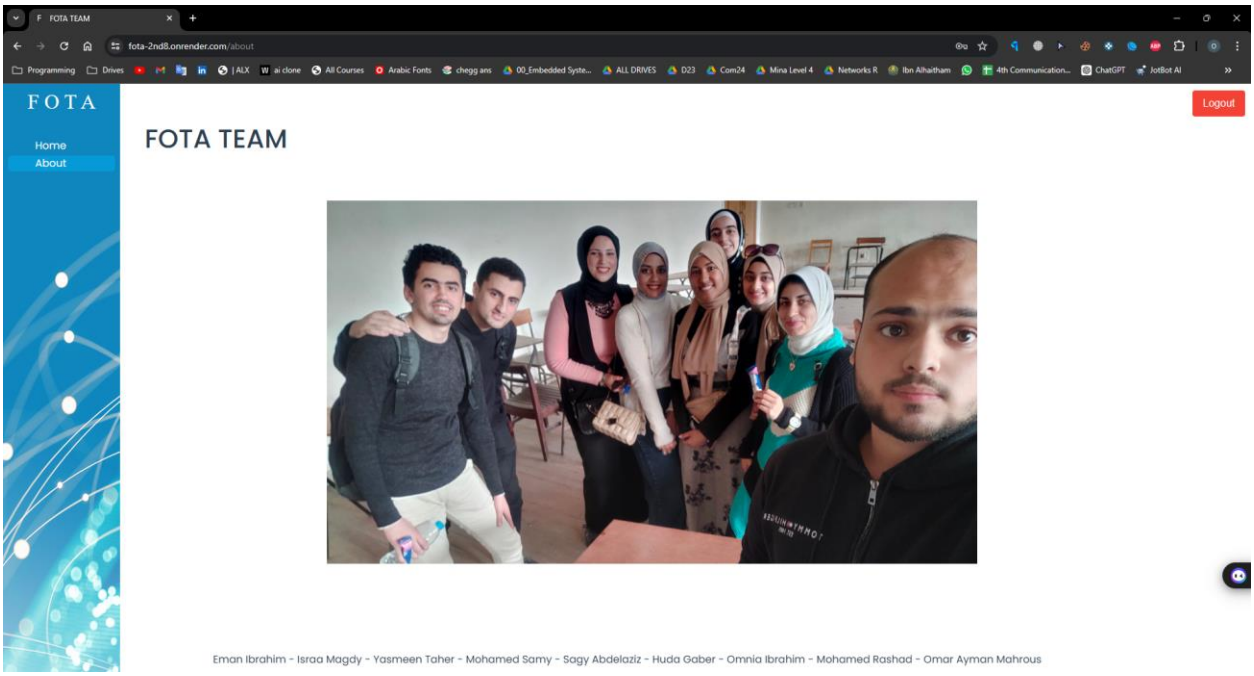


Figure 25

The About Page provides an overview of the team behind the web application, highlighting the individuals who have contributed to its development. Each team member is featured with their name and a professional photograph, creating a personal and engaging experience for users. The page is designed to be visually appealing and informative, giving users a glimpse into the people who made the project possible. By showcasing the team's names and images, the About Page fosters a sense of connection and transparency, allowing users to appreciate the collaborative effort and diverse expertise that went into creating the application.

- **Firebase Authentication**

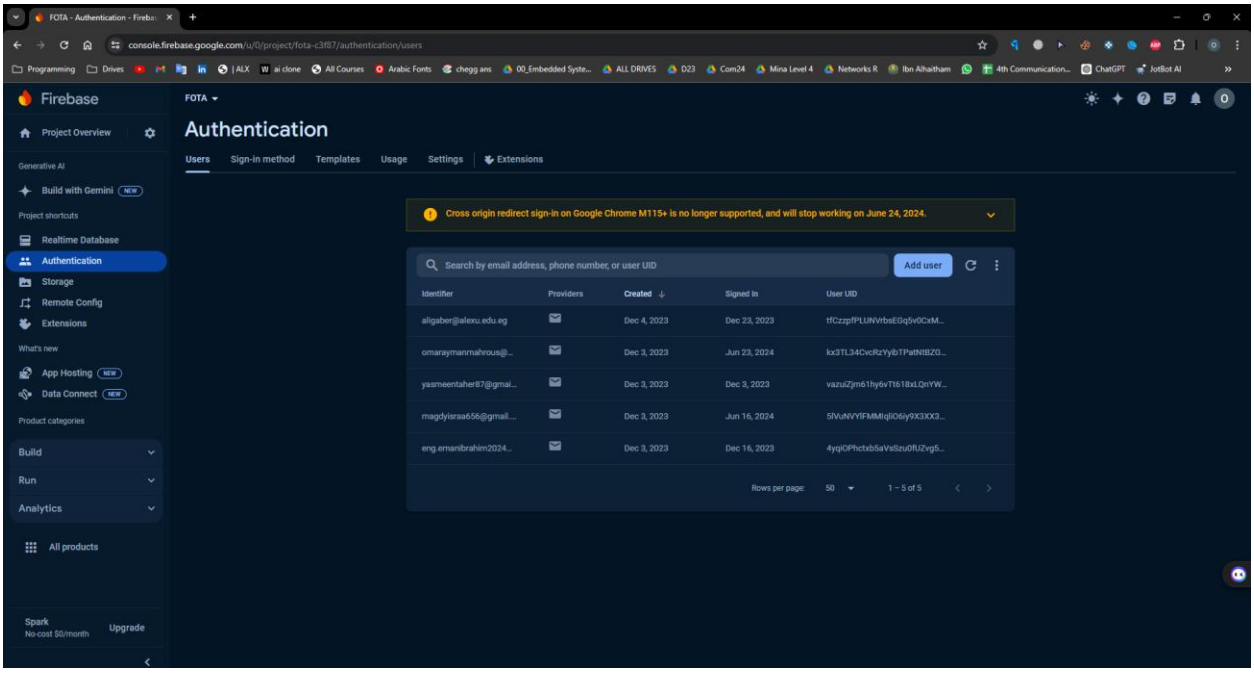


Figure 26

The Firebase Authentication Page is designed to provide a secure and seamless login experience for users. It features a straightforward interface where users can enter their credentials to access the application. The page includes input fields for username or email and password, along with a prominent login button to submit the credentials. To enhance user experience, the page also offers options for users to register a new account or reset their password if needed. By leveraging Firebase Authentication, this page ensures robust security measures, such as email/password authentication and secure token management, to protect user data. The design prioritizes ease of use and security, making the authentication process quick and reliable for users.

- **Firebase Storage**

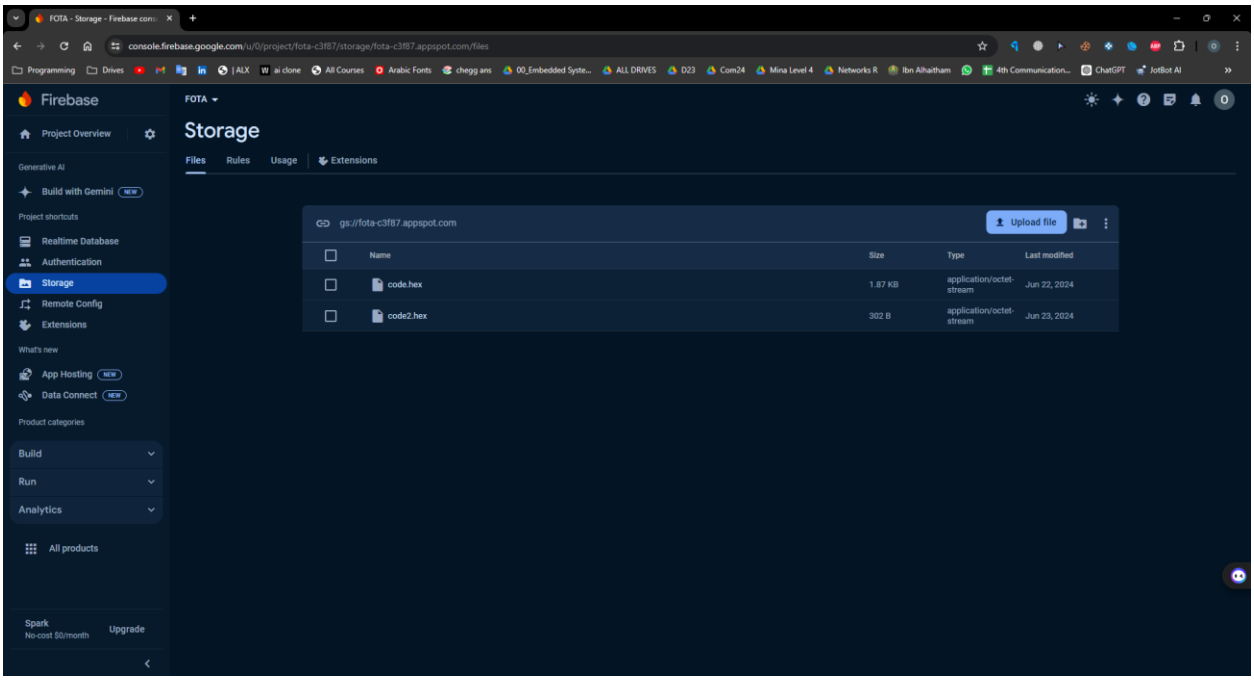


Figure 27

The Firebase Storage Page is integral to managing and storing firmware updates for the application. This page facilitates the upload and management of firmware files, such as code.hex and code2.hex, which correspond to two distinct applications. The interface is user-friendly, allowing administrators to easily upload these files to Firebase Storage. Each file is securely stored in the cloud, ensuring that they are readily accessible for deployment to the ESP32 devices. The page provides visual feedback during the upload process, such as progress indicators and confirmation messages, to ensure that users are informed about the status of their uploads. By leveraging Firebase Storage, the application benefits from scalable and reliable cloud storage, ensuring that firmware files are safely stored and can be efficiently distributed to connected devices. This setup not only streamlines the update process but also enhances the overall reliability and performance of the system.

3.4 Main Functions

We Have Main Files:

- Login.vue
- Home.vue
- About.vue
- Server.js
- Firebase - index.js
- router - index.js
- store - index.js

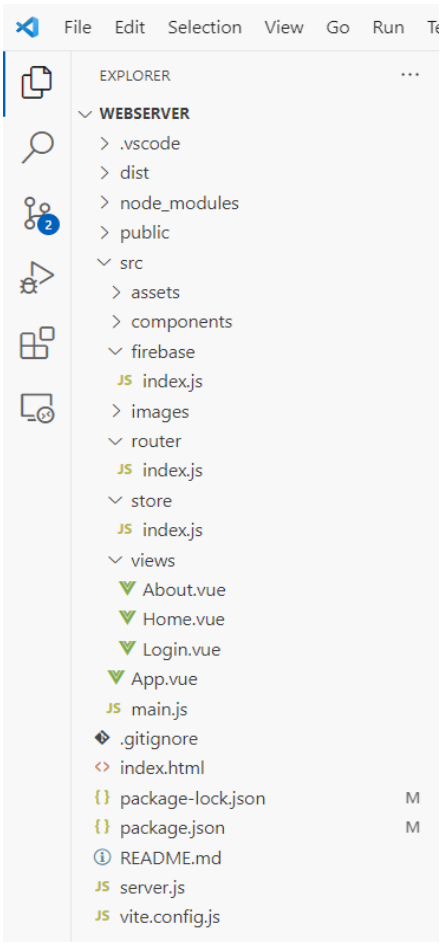


Figure 28

- **Login.vue**

Login file component defines a structured login page within its `<template>`, leveraging Bootstrap classes for layout and Vue directives (`v-model`, `@submit.prevent`) for interactivity. It consists of two main columns: one displaying a decorative image and the other hosting a login form. The form includes email and password input fields bound to reactive variables (`login_form.email`, `login_form.password`) managed by Vue's Composition API (`ref`). Functions `login()` and `register()` dispatch user credentials to Vuex store actions (`'login'`, `'register'`) upon form submission, facilitating state management. CSS styles define form aesthetics, emphasizing minimalism with borderless inputs and gradient backgrounds for registration prompts. Media queries ensure responsiveness, adjusting layout and typography for smaller screens. Overall, the component integrates Vue.js functionalities with Vuex for state handling and CSS for visual presentation, aiming for a user-friendly and adaptable login interface.

Main Points in the Code:

- **Template:** Defines layout using Bootstrap grid and Vue directives for data binding and event handling.
- **Script:** Uses Vue Composition API (`setup()`) to manage form data (`login_form`, `register_form`) and interact with Vuex store (`useStore()`).
- **Style:** Provides extensive styling for form elements, including input fields, buttons, and responsive design using media queries.

Code:

```
<template>
    <main class="login">
        <div class="wrapper">
            <div class="container main">
                <div class="row">
                    <div class="col-md-6 side-image">
                        <img src="images/white.png" alt="">
                        <div class="text"></div></div>
                    <div class="col-md-6 right">
                        <form class="login" @submit.prevent="login">
                            <img src="../images/log.png">
                            <h2>Login</h2>
                            <input type="email" placeholder="Email address"
v-model="login_form.email">
                            <input type="password" placeholder="Password" v-
model="login_form.password">
                            <input type="submit" value="Login">
                            <p class="contact-info">
                                Not Authorized Yet? Contact Us on Whatsapp <a
href="https://wa.me/201207494352" target="_blank">+201207494352</a>
                            </p>
                        </form></div></div></div></div></main></template>
<script>
import { ref } from 'vue'
import { useStore } from 'vuex'

export default {
    setup () {
        const login_form = ref({});
        const store = useStore();

        const login = () => {
            store.dispatch('login', login_form.value);
        }

        return {
            login_form,
            login
        }
    }}</script>
```

- ## Home.vue

Home Vue.js component manages a dynamic web application interface featuring components like `Sidebar` and `Navbar`, importing styles and functionalities from external resources and Firebase services. It integrates Vuex for state management (`$store.dispatch('logout')` triggers logout functionality), displaying a responsive layout with Bootstrap (`col-md-8`, `col-md-12`) and custom CSS (`veto`, `veto2`, `veto3`, `veto4`, `veto6`) for positioning and styling elements. The `upload` method uses Firebase Storage API (`ref`, `uploadBytes`) to facilitate file uploads, capturing file input changes (`@change="handleFileUpload"`) and triggering upload actions (`@click="upload"`). The template includes components for displaying uploaded images (`<img>`), handling file inputs (`<input type="file">`), and managing content layout (`<div class="container">`, `<div class="row center-row">`). Overall, the component demonstrates effective integration of Vue.js reactivity, Firebase services for data handling, and CSS for responsive design.

Main Points in the Code:

- **Vue Component Structure:** Utilizes Vue.js to structure dynamic components (`Sidebar`, `Navbar`) and manage state (`$store.dispatch('logout')`).
- **Firebase Integration:** Imports and utilizes Firebase Storage (`ref`, `uploadBytes`) for file handling and storage operations.
- **CSS Styling:** Applies custom and Bootstrap-based styles (`veto`, `center-row`, `veto4`) for layout and visual presentation.
- **File Upload Functionality:** Implements `upload` method to handle file uploads triggered by user interaction (`@change`, `@click`).
- **Responsive Design:** Utilizes Bootstrap (`col-md-8`, `col-md-12`) and custom CSS (`veto`, `veto6`) for responsive layout and styling adjustments.

Code:

```
<script>
import Sidebar from '@/components/sidebar/Sidebar'
import Navbar from '../components/Navbar.vue'
import { sidebarWidth } from '@/components/sidebar/state'
import { storage } from "../firebase/index"
import { ref, uploadBytes } from "firebase/storage"

export default {
  name: 'App',
  components: { Sidebar, Navbar } ,
  setup() {
    return { sidebarWidth }
  },
  methods:{
    upload: function(){
      const file = this.$refs.myfile.files[0];
      const storageRef = ref(storage, file.name);

uploadBytes(storageRef, file).then((snapshot) => {
  console.log("Uploaded successfully");
}).catch((error) => {
  console.error("Error uploading:", error);
});
    }
  }
}
</script>
<template>

  <button class="veto" @click="$store.dispatch('logout')">Logout</button>

<div class="col-md-8">
   <Sidebar class="veto6" />
  <div :style="{ 'margin-left': sidebarWidth }">
    <router-view />

  </div>
</div>


<div class="container">
  <div class="row center-row">
```

```
    <div class="col-md-12 center-content">

      <div id="app" class="d-flex flex-column align-items-center justify-content-
center">
         <img src="../images/up.png" alt="Your Image" class= "veto4" style="max-
width: 200px;">
        <h1>Upload Code</h1>
        <div class="mb-3">
          <!-- <label for="fileInput" class="form-label">Upload File</label> -->
          <input type="file" class="exampleFormControlInput1" id="fileInput"
ref="myfile" @change="handleFileUpload">
        </div>
        <button class="btn btn-success" @click="upload">Upload</button>
      </div>
    </div>
  </div>
</div>
```

- **About.vue**

About Vue.js component integrates a structured web application interface with dynamic components such as `Sidebar` and utilizes Vuex for state management. The setup function initializes variables `collapsed` and `toggleSidebar` from the imported `sidebar/state.js`, enabling control over the sidebar's visibility and state. The template section includes a logout button triggering Vuex actions (`$store.dispatch('logout')`), a responsive layout using Bootstrap (`col-md-8`, `col-md-6`), and custom CSS (`veto`, `veto8`) for styling. It incorporates Vue Router for routing (`<router-view />`) and displays static content like team names and images (`<img>`), emphasizing centralized content alignment (`text-center`) and fixed positioning (`fixed-bottom`) for team information.

Main Points in the Code:

- **Vue Component Structure:** Utilizes Vue.js for component-based architecture with `Sidebar` integration and Vuex state management.
- **State Management:** Imports and utilizes Vuex variables (`collapsed`, `toggleSidebar`) from `sidebar/state.js` to manage sidebar behavior.
- **Template Structure:** Includes a logout button triggering Vuex actions, responsive layout with Bootstrap classes (`col-md-8`, `col-md-6`), and centralized content alignment (`text-center`, `fixed-bottom`).

Code:

```
<script>
import Sidebar from '@/components/sidebar/Sidebar'
import { sidebarWidth } from '@/components/sidebar/state'
import { collapsed, toggleSidebar } from '@/components/sidebar/state.js'
export default {
  components: { Sidebar },
  setup() {
    return { collapsed, toggleSidebar }
  }
}
</script>
<template>
<button class="veto" @click="$store.dispatch('logout')">Logout</button>
<div class="col-md-8">
  <Sidebar class="veto6" />
  <div :style="{ 'margin-left': sidebarWidth }">
    <router-view />
  </div>
  <div class="col-md-6">
    <div class="text-center">
    <h1 class="text-center">
      FOTA TEAM
    </h1>
    </div>
    <img class="veto2" src="../images/ss.jpg" alt="">

      <p class="fixed-bottom">
      Eman Ibrahim - Israa Magdy - Yasmeen Taher - Mohamed Samy - Sagy Abdelaziz
- Huda Gaber - Omnia Ibrahim - Mohamed Rashad - Omar Ayman Mahrous
    </p>


  </div>
</div>
```

- ## Server.js

Server script sets up an Express server to host a Vue.js or similar frontend application's production build (`dist` directory). It begins by importing necessary modules like `express`, `serve-static` for static file serving, and `path` for directory operations. The `app.use('/', serveStatic(path.join(__dirname, '/dist')))` middleware configures Express to serve static files from the `dist` directory, which typically contains compiled and bundled frontend assets. The `app.get(/.*/, ...)` route ensures that any route not explicitly defined (`/`) will serve the `index.html` file from `dist`, essential for SPA client-side routing to function correctly. The server listens on a port defined by the environment variable `process.env.PORT` or defaults to port `8080`. Finally, a console log confirms the server's startup and the port it's listening on.

Main Points in the Code:

- **Express Setup:** Initializes an Express server (`app`) to handle HTTP requests.
- **Static File Serving:** Uses `serve-static` middleware to serve static files (`HTML`, `CSS`, `JS`) from the `dist` directory.
- **Fallback Route (`*`):** Routes any unmatched URLs (`/.*/`) to serve `index.html`, facilitating SPA routing.
- **Port Configuration:** Determines the server's port dynamically using `process.env.PORT` or defaults to `8080`.
- **Server Start:** Logs a message confirming the server's initialization and the port it's listening on, ensuring operational status and facilitating debugging during deployment.

Code:

```
const express = require('express')
const serveStatic = require('serve-static')
const path = require('path')

const app = express()

//here we are configuring dist to serve app files
app.use('/', serveStatic(path.join(__dirname, '/dist')))

// this * route is to serve project on different page routes except root `/`
app.get(/.*/, function (req, res) {
    res.sendFile(path.join(__dirname, '/dist/index.html'))
})
const port = process.env.PORT || 8080
app.listen(port)
console.log(`app is listening on port: ${port}`)
```

- ## Firebase – index.js

Firebase JavaScript code initializes Firebase services for authentication and storage within a web application. It begins by importing functions from the Firebase SDK (`initializeApp`, `getAuth`, `getStorage`) necessary for configuring Firebase services. The `firebaseConfig` object holds credentials and configuration details required to connect the application securely to Firebase, including API keys, project identifiers, and authentication domains. The `initializeApp` function initializes Firebase with these configurations, creating a connection to Firebase services. Subsequently, `getAuth(app)` initializes Firebase Authentication, providing methods for user authentication and management. `getStorage(app)` initializes Firebase Storage, enabling the application to interact with cloud storage for storing and retrieving files. Finally, the `auth` and `storage` objects are exported, making them available for use throughout the application.

Main Points in the Code:

- **Firebase Configuration:** Sets up Firebase with `firebaseConfig` containing API keys, project IDs, and related credentials.
- **Firebase Initialization (`initializeApp`):** Initializes Firebase with the provided configuration, establishing a connection to Firebase services.
- **Authentication Setup (`getAuth`):** Initializes Firebase Authentication (`auth`) with the initialized Firebase app (`app`), allowing user authentication functionalities.
- **Storage Setup (`getStorage`):** Initializes Firebase Storage (`storage`) with the initialized Firebase app (`app`), enabling file storage operations in Firebase cloud storage.
- **Exports:** Makes `auth` and `storage` objects available for use elsewhere in the application, facilitating integration of Firebase authentication and storage functionalities into the frontend or backend logic as needed.

Code:

```
import { initializeApp } from "firebase/app"
import { getAuth } from "firebase/auth"
import { getStorage } from "firebase/storage"
const firebaseConfig = {
    apiKey: "----------------------------------------",
    authDomain: "fota-c3f87.firebaseapp.com",
    projectId: "fota-c3f87",
    storageBucket: "fota-c3f87.appspot.com",
    messagingSenderId: "1092972955792",
    appId: "1:1092972955792:web:45a05dac28446aa35f7015",
    measurementId: "G-5G78NWJ1RK"}
const app = initializeApp(firebaseConfig)
const auth = getAuth(app)
const storage = getStorage(app);
export { storage }
```

- ## Router – index.js

This Vue.js router configuration manages navigation within a web application, integrating authentication checks using Firebase. The code imports necessary functions from Vue Router (`createRouter`, `createWebHistory`) to define routes such as Home, About, and Login, each specifying components to render and optional authentication requirements (`requiresAuth` metadata). The `createRouter` function initializes the router instance with route definitions and uses `createWebHistory` for navigation history management based on the base URL provided in the environment variables (`process.env.BASE_URL`). The `router.beforeEach` function is a global navigation guard that executes before every route change. It checks conditions: if the destination is the login page and the user is already authenticated (`auth.currentUser` exists), it redirects to the Home page (`/`). If a route requires authentication (`requiresAuth` metadata) but no user is authenticated, it redirects to the Login page (`/login`). Otherwise, it allows navigation to proceed. This setup ensures proper access control throughout the application based on user authentication status retrieved from Firebase.

Main Points in the Code:

- **Vue Router Setup:** Uses `createRouter` and `createWebHistory` from Vue Router to create a router instance with defined routes.
- **Route Definitions:** Defines routes for Home, About, and Login components, specifying paths, component associations, and optional `requiresAuth` metadata.
- **Navigation Guard (`router.beforeEach`):** Implements a global navigation guard that runs before each route change.
- **Authentication Checks:** Utilizes Firebase's `auth.currentUser` to verify user authentication status.
- **Routing Logic:** Redirects to appropriate routes based on authentication status: to Home if already logged in on the Login page, to Login if authentication is required but not authenticated, or allows navigation to proceed otherwise.

Code:

```
import { createRouter, createWebHistory } from 'vue-router'
import Home from '../views/Home.vue'
import Login from '../views/Login.vue'
import { auth } from '../firebase'

const routes = [
  {
    path: '/',
    name: 'Home',
    component: Home,
    meta: {
      requiresAuth: true
```

```
    }
  },
  {
    path: '/about',
    name: 'About',
    component: () => import('../views/About.vue'),
    meta: {
      requiresAuth: true
    }
  },
  {
    path: '/login',
    name: 'Login',
    component: Login
  },
]

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes
})

router.beforeEach((to, from, next) => {
  if (to.path === '/login' && auth.currentUser) {
    next('/')
    return;
  }

  if (to.matched.some(record => record.meta.requiresAuth) && !auth.currentUser) {
    next('/login')
    return;
  }

  next();
})

export default router
```

- ## Store – index.js

This Vuex store configuration manages state and actions for user authentication using Firebase in a Vue.js application. It imports necessary modules like `createStore` from Vuex, `router` from the application's routing setup, and Firebase's `auth` module along with authentication functions (`createUserWithEmailAndPassword`, `signInWithEmailAndPassword`, `signOut`). The store's state initializes with `user` set to `null`, mutations (`SET_USER` and `CLEAR_USER`) manage state updates for the authenticated user, and actions (`login`, `register`, `logout`, `fetchUser`) orchestrate user authentication flows asynchronously. The `login` action attempts to sign in a user with provided credentials, handling potential errors like user-not-found or wrong-password, committing the user to state upon successful login and redirecting to the home route (`/`). The `register` action similarly handles user registration errors and commits the authenticated user upon successful registration. The `logout` action signs out the current user, clears user state, and redirects to the login page. The `fetchUser` action listens for authentication state changes, updating the user state accordingly and redirecting to the home route if authenticated and navigating from the login page. This setup ensures robust user authentication management throughout the application.

Main Points in the Code:

- **State Management:** Initializes `user` state to `null` and provides mutations (`SET_USER` and `CLEAR_USER`) to update the user state.
- **Actions:**
  - `login`: Authenticates users with provided credentials, handles errors, commits user data, and redirects to the home route upon success.
  - `register`: Registers new users, handles registration errors, updates user state upon success, and redirects to the home route.
  - `logout`: Signs out the current user, clears user state, and redirects to the login page.
  - `fetchUser`: Listens for authentication state changes, updates user state, and manages route navigation based on authentication status.
- **Integration with Router and Firebase:** Utilizes Vue Router for navigation and Firebase authentication methods (`signInWithEmailAndPassword`, `createUserWithEmailAndPassword`, `signOut`) for user authentication operations.

Code:

```
import { createStore } from 'vuex'
import router from '../router'
import { auth } from '../firebase'
import {
  createUserWithEmailAndPassword,
  signInWithEmailAndPassword,
  signOut
} from 'firebase/auth'

export default createStore({
  state: {
    user: null
  },
  mutations: {
    SET_USER (state, user) {
      state.user = user
    },
    CLEAR_USER (state) {
      state.user = null
    }
  },
  actions: {
    async login ({ commit }, details) {
      const { email, password } = details
      try {
        await signInWithEmailAndPassword(auth, email, password)
      } catch (error) {
        switch(error.code) {
          case 'auth/user-not-found':
            alert("User not found")
            break
          case 'auth/wrong-password':
            alert("Wrong password")
            break
          default:
            alert("Something went wrong")
        }
        return
      }
      commit('SET_USER', auth.currentUser)
      router.push('/') },
    async register ({ commit}, details) {
```

```
    const { email, password } = details

    try {
      await createUserWithEmailAndPassword(auth, email, password)
    } catch (error) {
      switch(error.code) {
        case 'auth/email-already-in-use':
          alert("Email already in use")
          break
        case 'auth/invalid-email':
          alert("Invalid email")
          break
        case 'auth/operation-not-allowed':
          alert("Operation not allowed")
          break
        case 'auth/weak-password':
          alert("Weak password")
          break
        default:
          alert("Something went wrong")
      }
      return
    }
    commit('SET_USER', auth.currentUser)
    router.push('/')
  },
  async logout ({ commit }) {
    await signOut(auth)
    commit('CLEAR_USER')
    router.push('/login')
  },
  fetchUser ({ commit }) {
    auth.onAuthStateChanged(async user => {
      if (user === null) {
        commit('CLEAR_USER')
      } else {
        commit('SET_USER', user)

        if (router.isReady() && router.currentRoute.value.path === '/login') {
          router.push('/')
        }
      }}) }}})
```

3.5 Hosting

Render.com is a robust cloud platform renowned for its simplicity and scalability in hosting modern applications and websites. Offering a straightforward approach to deployment, Render.com supports a variety of technologies including Node.js, Python, Docker, and static sites, making it versatile for different development needs. The platform stands out with its intuitive interface and seamless integration with version control systems like GitHub, allowing developers to deploy applications directly from their repositories with ease. Render.com's deployment pipelines automate the build and deploy process, ensuring applications are always up-to-date without manual intervention. One of the key advantages of Render.com is its focus on performance and reliability. Applications hosted on Render benefit from built-in horizontal scaling, automatic HTTPS, global CDN, and managed databases, all of which contribute to fast load times and high availability. Developers can monitor application performance and resource usage through detailed dashboards, enabling proactive management and optimization. Render.com also offers a generous free tier suitable for personal projects and small applications, providing ample resources without upfront costs. This makes it accessible for developers to experiment, test, and deploy projects without financial barriers. For larger-scale applications, Render.com's transparent pricing model ensures cost efficiency with pay-as-you-go pricing based on actual usage

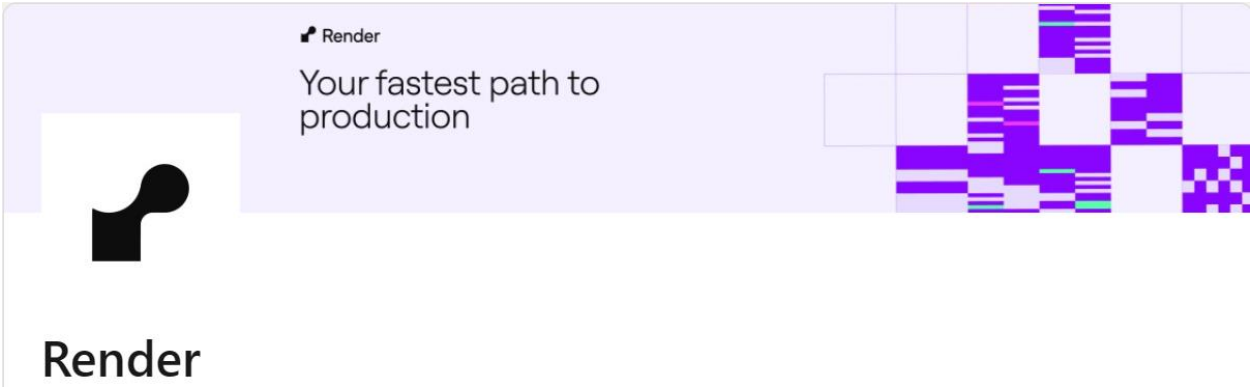


Figure 29

3.6 Try it Yourself : FOTA WEBSITE

https://fota-2nd8.onrender.com/

For Authentication use:

Email: **sampleuser@gmail.com**

Password : **sampleuser**

Chapter 4: Telematics Unit

Figure 30

4.1 Introduction

Telematics control unit (TCU) is the embedded onboard system that controls wireless tracking, diagnostics and communication to and from the vehicle

In the automotive industry, a telematics control unit (TCU) is an embedded device onboard a car that wirelessly links the vehicle to cloud storage. The Telematics Control Unit collects telematics data from the car, such as location, speed, engine data, connection quality, and so on, by connecting with various subsystems in the vehicle via data and control buses. It may also offer in-vehicle networking via Wi-Fi and Bluetooth, as well as the call capability in certain areas.

A Telematics Control Unit is made up of the following components:

- A satellite navigation system
- A microcontroller
- A mobile networking unit
- An external unit for cellular communication
- A unit that processes electrical signals
- A memory card to store valuable information
- Battery module

4.2 Why use a Telematics Control Unit?

- Increasing the vehicle's connectivity by introducing additional modems and processors to facilitate communication from the car to the cloud, infrastructure, or other vehicles.
- Antenna power optimization through improved current sensing, diagnostics, and noise reduction.
- Data transmission is reliable, efficient, and speedier, allowing connectivity with other systems in the car.
- Telematics Control Unit generates a digital blueprint of every component of a vehicle's operation, enabling fleet management in evaluating where enhancements in accident prevention and driver safety regulations can be applied.
- Telematics Control Unit can help with maintenance by analyzing hours-of-use information and planning maintenance inspections, as well as taking note of guarantee recovery, engine hour tracking, and service records tracking.
- Ensuring a high degree of engine health, such as battery voltage, transmission fluid, oxygen sensor issues, and more, may assist fleet managers in cutting costs and keeping cars in good condition and safe.

Telematics is poised for exponential growth as new applications are developed to make use of modern GPS units and the pervasive use of mobile devices. More fleets are realizing the value of fleet activity detection in order to save costs, increase efficiency, improve accountability, and assure total compliance with government regulations.

Telematics will become an indispensable component of all modern fleet operations as owners want to go beyond the bare essentials.

4.3 Main role of telematics unit in FOTA:

The Telematics unit is the bridge connecting between the server
and the whole system allowing sending and receiving multiple data through it.
We use the Telematics unit to communicate mainly with two other units, the
server and the gateway (STM32).

1. **Connecting to OEM server: Get and send data**
2. **Connect with user interface : to advertise that there is an updating file**
3. **Download the updating file in its memory**
4. **Control the updating operation**
5. **Send diagnostics to server to retransmit file.**

We used ESP32 module as it has a WI-FI module which is easy to be established using a few lines of code through Arduino IDE which gives us access to multiple libraries and functions in the ESP32.

4.4 ESP32

Figure 31

The ESP32 is a popular and versatile microcontroller; it is widely used on the Internet of Things (IoT) and embedded systems projects due to its powerful features and built-in connectivity options.

The Espressif Systems developed the ESP32. Also, the ESP32 comprises dual-mode Bluetooth and Wi-Fi connectivity. The ESP32 is technically a chip. However, the development boards and modules in this chip are what we refer to as ESP32. Furthermore, the ESP32 is highly ranked as the best chip or module for IoT developers and hobbyists.

Furthermore, ESP32 comprises various types of proprietary software. This board comprises sophisticated features like different power modes, dynamic power scaling, and fine-grained clock gating. Also, the ESP32 is a chip that offers a lot of powerful features. When it comes to IoT, this chip offers a lot of benefits.

While the reputation of this chip is merited in IoT applications, this low-cost device is also ideal for use in various production systems. Furthermore, its resources and capabilities have grown over the years. The ESp32 is a bare chip. However, it could also refer to ESP32 development boards. Some ESP32 boards feature Gerber and BOM files. These Gerber and BOM files help in designing reference circuit.

4.4.1 ESP with Firebase & OEM server:

Figure 32

OEM webserver is linked with firebase and makes it easy to upload the file without a lot steps.

One of Firebase's key features is Firebase Cloud Storage, which lets you store binary data or files which can be retrieved over the Internet using a simple HTTP request.

A manual way to upload a new binary to Firebase is to visit the Firebase console, click on "Storage" in the left nav bar, and then click on the "Upload file" button, but we upload the file from OAM web server.

By using ESP it can download the file from Firebase the ESP can interact with the database from anywhere in the world as long as it is connected to the internet

4.4.2 ESP as an access point

Figure 33

ESP32 can act as a Wi-Fi station, as an access point, or both.
The connection of GUI to ESP through internet needed to get a static IP, so configure the ESP32 as an access point , that creating its own Wi-Fi network and nearby Wi-Fi devices (stations) can connect to it (likes smartphones or computers) with static IP, then incorporate the static IP settings into GUI of client
Also, note that, unlike the phone's connection sharing, the ESP32 is not connected to the Internet. So it can use the Access Point mode to create a private Wi-Fi local area network wholly isolated from the Internet.

Here are some uses of the Access Point mode:

- Connecting to the ESP32 in Access Point mode temporarily to enter the credentials of its Wi-Fi router and allow the ESP32 to connect to our classic Wi-Fi network. Most connected objects use this principle to connect to the home Wi-Fi.
- Have a separate network from your home network and not connected to the Internet.
- To easily communicate between several ESP32 by Wi-Fi.
- To have a local server on an isolated Wi-Fi network.

4.4.3 ESP as a crucial intermediary

The ESP serves as a crucial intermediary, ensuring that the user commands from the GUI are accurately translated and communicated to control the ECUs, Enabling the proper functioning of various car systems.
When an HTTP request is sent to an Electronic Stability Program (ESP) module in a car, the ESP acts as a bridge between the Graphical User Interface (GUI) and the main Electronic Control Units (ECUs).

The process works as follows:

- HTTP **Request from GUI**: The GUI sends an HTTP request containing a specific flag or command to the ESP module. This flag represents a certain action or request from the user.
- **ESP Module**: Upon receiving the HTTP request, the ESP module interprets the flag from the GUI. The ESP is programmed to understand the different types of flags it might receive.
- **Translation of Flags**: The ESP translates the received flag into another flags that is recognized by the ECUs. This translation is based on predefined functions implemented in ESP
- **Communication with ECU**: The ESP then sends the translated flag to the appropriate ECU. This allows the ECU to understand and execute the command.
- **Execution by ECU**: The ECU receives the translated flag and performs the corresponding action. This could involve controlling various aspects of the car, such as adjusting the suspension, modifying engine parameters, or altering the braking system

4.4.4 UART communication in ESP

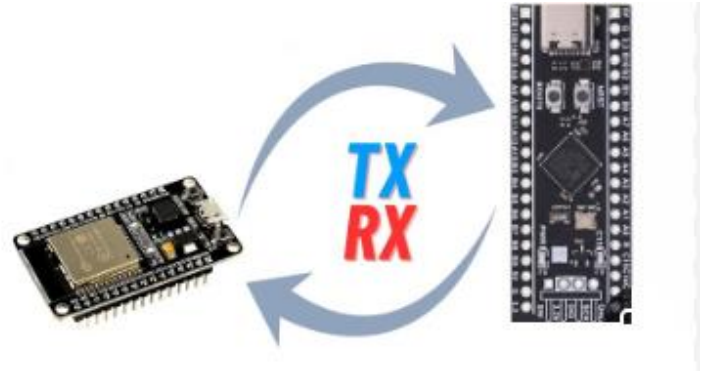


Figure 34

ESP is used to send data to Gateway through UART communication.

UART is a common serial protocol for a lot of devices. For example, when uploading a binary to your ESP you have probably used UART to access the chip. UART (or for Arduino often also called Serial) usually consists of 2 pins:
TX: This line is used to send data to the device at the other end.
RX: This line is used to receive data from the device at the other end.

Note that the naming of these two pins depends on the chosen perspective and can be ambiguous. For example, while the ESP might send (TX) on pin A and receive (RX) data on pin B, from the other device's perspective these two pins are switched (i.e. *it* sends on pin B and receives on pin A). So you might need to try with the two pins switched if it doesn't work immediately.
Additionally, each UART bus can operate at different speeds (baud rates), so ESP needs to know what speed to receive/send data at using the baud_rate option. The most common baud rates are 9600 and 115200.
In some cases only **TX** or **RX** exists as the device at the other end only accepts data or sends data.

The ESP32 chip has three UART controllers (UART0, UART1, and UART2) that feature an identical set of registers for ease of programming and flexibility.

Each UART controller is independently configurable with parameters such as baud rate, data bit length, bit ordering, number of stop bits, parity bit etc. All the controllers are compatible with UART-enabled devices from various manufacturers and can also support Infrared Data Association protocols (IrDA).

4.4.5 Main Functions of ESP in FOTA project:

1. **Connecting to the OEM server**

- Establish a connection with Firebase and retrieve data.
- Utilize Wi-Fi with full TCP/IP stack and microcontroller capabilities.
- Continuously check for updates from the OEM (Original Equipment Manufacturer) server which connected to firebase.

2. **User Interface Connection**

- Ensure user acceptance of the update process.

3. **Downloading and Decrypting Update Files**

- Download the update file into memory.
- Decrypt the downloaded file for further processing.

4. **Connecting to the Gateway**

- Determine the specific application requiring an update.
- Transmit identifying numbers to the gateway to specify the unit being updated.
- Convert the update file into a string format suitable for transmission via UART (Universal Asynchronous Receiver-Transmitter) communication protocol.
- Implement management protocols to ensure accurate and reliable file transmission to the gateway.

5. **The ESP serves as a crucial intermediary**

ESP is ensuring that the user commands from the GUI are accurately translated and communicated to the ECUs, enabling the proper functioning of various car systems

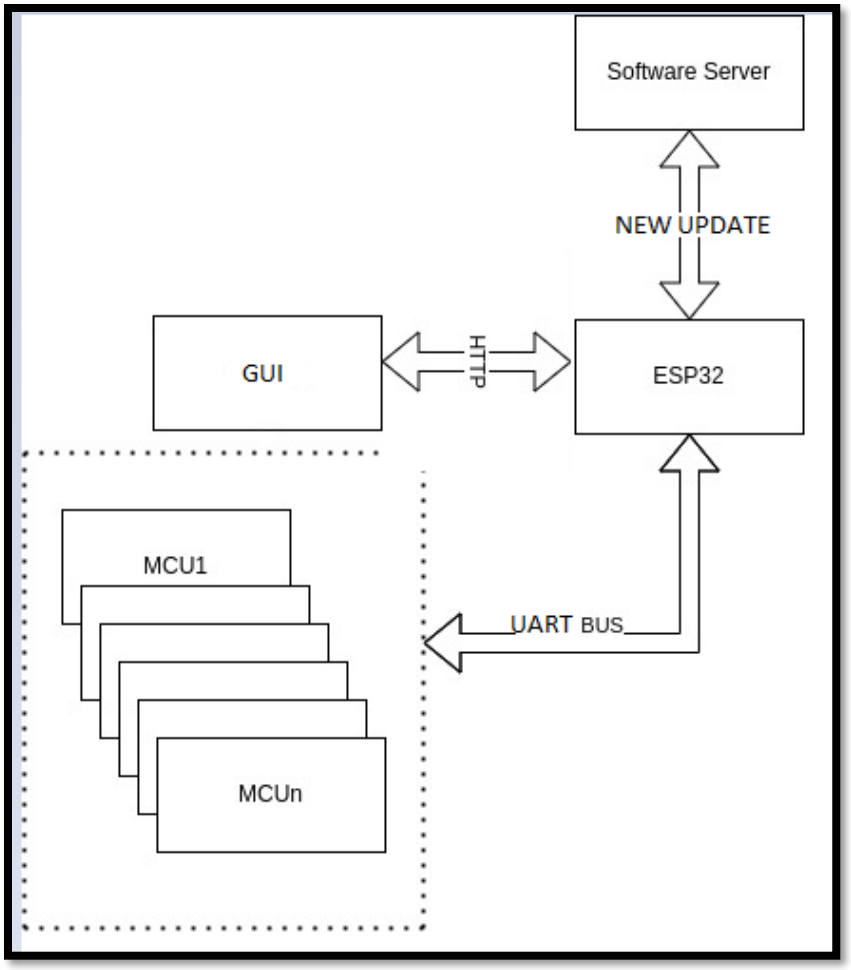

Software Server
NEW UPDATE
GUI
HTTP
ESP32
MCU1
UART BUS
MCUn

Figure 35

4.5 Software implementation

4.5.1 Wireless communication part

- ESP32 to connect to local Wi-Fi network for internet access

```
// Wi-Fi credentials
const char *WIFI_SSID = "******";
const char *WIFI_PASSWORD = "********";
```

  - WIFI_SSID and WIFI_PASSWORD are variables storing the credentials (SSID and password) of your Wi-Fi network.

  - In setup function():

```
void setup() {
    Serial.begin(9600);

    delay(1000); // Add a delay after initializing Serial

    // Connect to Wi-Fi
    WiFi.begin(WIFI_SSID, WIFI_PASSWORD);

    Serial.print("Connecting to Wi-Fi");
    unsigned long ms = millis();
    while (WiFi.status() != WL_CONNECTED)
    {
        Serial.print(".");
        delay(300);
        if (millis() - ms > 10000)
            break;
    }
    Serial.println();
    Serial.print("Connected with IP: ");
    Serial.println(WiFi.localIP());
    Serial.println();
```

- the ESP32 is configured as an Access Point to connect to GUI (Mobile Application)

```
// AP credentials
const char *AP_SSID = "ESP32_Hotspot";
const char *AP_PASSWORD = "123456789";

// Static IP configuration for the AP
IPAddress local_IP(192, 168, 4, 1);
IPAddress gateway(192, 168, 4, 1);
IPAddress subnet(255, 255, 255, 0);
```

  - **AP_SSID** and **AP_PASSWORD** are variables storing the credentials for the Access Point (AP) mode on your ESP32 device. When the ESP32 is configured as an Access Point, these credentials define the network name (SSID) and password ("123456789") that client devices can use to connect to the ESP32's Wi-Fi network.
  - IPAddress local_IP(192, 168, 4, 1),This line assigns a static IP address to the Access Point (AP). Devices connecting to this AP will be assigned IP addresses within the same subnet.
  - IPAddress gateway(192, 168, 4, 1) sets the gateway IP address. It serves as the access point through which devices connected to the AP can access other networks or the internet.
  - IPAddress subnet(255, 255, 255, 0) defines the subnet mask. A subnet mask of 255.255.255.0 indicates that the first three octets (192.168.4 in this case) represent the network portion of the IP address, and the last octet (1 for the gateway and local IP) represents the host portion. This allows devices to communicate within the same local network segment.

- Access point In setup :

```
server.begin();
Serial.println("HTTP server started");

Serial.print("Connected to WiFi. IP address: ");
Serial.println(WiFi.localIP());

// Set up the ESP32 as an access point with static IP
WiFi.softAP(AP_SSID, AP_PASSWORD);
WiFi.softAPConfig(local_IP, gateway, subnet);

Serial.print("AP IP address: ");
Serial.println(WiFi.softAPIP());
```

4.5.2 Firebase connection

Connect to fire base by using authentication data about database

```
const char *API_KEY = "AIzaSyA5YnTypXaPBCWh4dN1RofPbfdLOpwQKbE";
const char *USER_EMAIL = "********@gmail.com";
const char *USER_PASSWORD = "********";
const char *STORAGE_BUCKET_ID = "fota-c3f87.appspot.com";
```

- keys is used for authentication and authorization purposes when accessing between Firebase services and ESP
  - API_KEY is a constant pointer to a character array storing the Firebase API key. USER_EMAIL is a constant pointer to a character array containing the email address associated with your Firebase account. This email is used for user authentication when interacting with Firebase services that require user-specific access
  - USER_PASSWORD is a constant pointer to a character array storing the password associated with the USER_EMAIL for Firebase authentication

purposes. It's used to securely authenticate and authorize access to Firebase services.

- STORAGE_BUCKET_ID is a constant pointer to a character array representing the ID of the Firebase Storage bucket (**fota-c3f87.appspot.com in this case)**. Firebase Storage provides secure file uploads and downloads for Firebase applications, and this identifier specifies the bucket where your application's files are stored.

4.5.3 Checking for updates

Checking downloaded file in firebase by specific name (code1)& (Code2) only name examples for different codes towards to different updating Application

- Main function in checking update process

```
void checkForUpdates() {
    Serial.println("\nChecking for updates...\n");

    if (!Firebase.Storage.download(&fbdo, STORAGE_BUCKET_ID, "code.hex", "/file.hex", mem_storage_type_flash)) {
        Serial.println(fbdo.errorReason());
        i1 = false;
    } else {
        i1 = true;
        fileDownloaded = false;
    }
}

void checkForUpdates2() {
    Serial.println("\nChecking for updates2...\n");

    if (!Firebase.Storage.download(&fbdo, STORAGE_BUCKET_ID, "code2.hex", "/file.hex", mem_storage_type_flash)) {
        Serial.println(fbdo.errorReason());
        i2 = false;
    } else {
        i2 = true;
        fileDownloaded = false;
    }
}
```

- **checkForUpdates** and **checkForUpdates2** are two functions that check for updates by downloading files from Firebase storage. They attempt to download different files (code.hex and code2.hex respectively) to a local file named /file.hex.
  When file is successifully downloaded , set boolean flags i1(when code1 is exist) or i2((when code1 is exist) to true and return fileDownloaded to false to keep checking updates ,If file is failed downloaded, they set boolean flags i1 or i2 or both to false

- Main sequence in checking for updating process

```
void loop() {
    if (Firebase.ready()) {
        unsigned long currentTime = millis();
        if (!fileDownloaded && currentTime - lastUpdateTime >= CHECK_UPDATE_INTERVAL) {
            lastUpdateTime = currentTime;
            checkForUpdates();
            if (!i1) {
                checkForUpdates2();
            }
            handleOff();
            if(UpdateCommandfromUser == true) {
                FCS_DownloadStatusInfo info;
                info.status = firebase_fcs_download_status_complete; // Mock the status as complete
                fcsDownloadCallback(info);  // Call the callback function
                fileDownloaded = false;
            }
        }
    }

    server.handleClient();
}
```

This checks if the Firebase connection is ready to be used. If Firebase is not ready, the rest of the code inside this if block will not execute.

- fileDownloaded: A boolean flag indicating if the file has been downloaded already. currentTime - lastUpdateTime >= CHECK_UPDATE_INTERVAL: Checks if the interval since the last update check has passed.
- checkForUpdates()This function checks for updates and attempts to download a file from Firebase. If it fails to download code 1(not found in firebase storage), it sets i1 to false and calls checkForUpdates2 to try downloading code2.
- (UpdateCommandfromUser == true) If the user has accepted the update, mock the completion of the download:
  UpdateCommandfromUser: A boolean flag indicating the user has accepted the update. FCS_DownloadStatusInfo info Creates an info object to store the download status. info.statusfirebase_fcs_download_status_complete Sets the download status to complete.fcsDownloadCallback(info): Calls a callback function with the download status information.
  fileDownloaded = false: Resets the fileDownloaded flag, indicating the process should be handled further.

4.5.4 HTTP requests

Send data between ESP and mobile application server is through http requests

- WebServer server(80)

```
WebServer server(80);
```

This line creates an instance of the WebServer class, initializing it to listen on port 80. The WebServer instance allows your device to handle HTTP requests, receive data from ESP, or respond to API requests

- Main function that using http request
    1) **The handleOff function** checking if there is any updating files

```
void handleOff()
{
    if ((i1 == true) || (i2 == true))
    {
        server.send(200, "text/plain", "FileDownloaded");
    }
    else if ((i1 == false) || (i2 == false))
    {
        Serial.println("File not Downloaded. Wait");
        String message = "File Not Found\n\n";
        server.send(404, "text/plain", message);
    }
}
```

The handleOff function checks the status of i1(which become true if code1 of Application 1 downloaded in ESP memory) and i2 (which become true if code2 of Application 2 downloaded in ESP memory)
If either i1 or i2 is true, it sends an HTTP 200 response indicating that the file has been downloaded.
If both i1 and i2 are false, it prints a message to the Serial Monitor and sends an HTTP 404 response indicating that the file was not found or downloaded.

2) The handleNotFound function is used to show that an error has occurred, and the requested file could not be found.

```
void handleNotFound() {
    Serial.println("ERRORRRRRR");
    String message = "File Not Found\n\n";
    server.send(404, "text/plain", message);
}
```

The handleNotFound function is used to handle HTTP requests for resources that do not exist on the server. When such a request is received, it prints an error message to the Serial Monitor for debugging purposes. It then sends a 404 Not Found HTTP response to the client, along with a plain text message indicating that the requested file was not found.

3) Updatecommand function indicate that the updating file is accepted by user

```
void updatecommand() {
    UpdateCommandfromUser = true;
}
```

4) Led on & led off function is used in manual mode in Air condition application (more detailed in chapter 6)

```
void ledon() {
    Serial.write('5');
    Serial.write('\n');
}

void ledoff() {
    Serial.write('6');
    Serial.write('\n');
}
```

5) Autotimerhalf & Autotimerone functions are used after updating the air condition application

```
void autotimerhalf() {
    Serial.write('8');
    Serial.write('\n');
}
void autotimerone() {
    Serial.write('4');
    Serial.write('\n');
}
```

6) These functions managed by the ESP module control the car's movement both before and after updates, the ESP sending char to the gateway via UART, Each character transmitted corresponds to a specific movement or action known by ECU in the car.

```
void a1() {
    Serial.write('W');
    Serial.write('\n');
}
void a2() {
    Serial.write('S');
    Serial.write('\n');
}
void a3() {
    Serial.write('A');
    Serial.write('\n');
}
void a4() {
    Serial.write('O');
    Serial.write('\n');
}
```

- Sets up various functions for an HTTP server requests and executed it on an ESP3.

```
server.on("/update", HTTP_GET, updatecommand);
server.on("/ledon", HTTP_GET, ledon);
server.on("/ledoff", HTTP_GET, ledoff);
server.on("/on", HTTP_GET, handleOff);
server.onNotFound(handleNotFound);

server.on("/autotimer", HTTP_GET, autotimer);

server.on("/a1", HTTP_GET, a1);
server.on("/a2", HTTP_GET, a2);
server.on("/a3", HTTP_GET, a3);
server.on("/a4", HTTP_GET, a4);

server.on("/autotimerhalf", HTTP_GET, autotimerhalf);
server.on("/autotimerone", HTTP_GET, autotimerone);
```

Each route ('/…. ') is associated with a specific function that handles the corresponding HTTP request to GUI.

4.5.5 ESP and Gateway connection

When update request has accepted by client , the file is sending to gateway through UART as sequence of strings, but before that it needed to indicate the updating file is for which application to send towards the right microcontroller by gateway.

1. Determine updating file for which Application to gateway

```
void fcsDownloadCallback(FCS_DownloadStatusInfo info) {
    if (info.status == firebase_fcs_download_status_complete) {
        File hexFile = LittleFS.open("/file.hex", "r");

        if (hexFile) {
            Serial.begin(9600);
            if (i1 == true) {
                Serial.write('1');
            } else if (i2 == true) {
                Serial.write('2');
            }

            Serial.write('\n');
            Serial1.write('1');
            Serial1.write('\n');
```

If code 1 is updating send 1 to gateway to send file to Application 1and if code 2 is updating send 2 to gateway to send file to code 2

2. Splitting function

The split Lines function is designed to split the file code into multiple lines based on newline ('\n') characters to send file as sequence of strings to make the communication between the ESP and gateway more reliable and efficient

```
void splitLines(String input) {
    linesCount = 0;
    String tempLine = "";
    for (int i = 0; i < input.length(); i++) {
        if (input[i] == '\n') {
            lines[linesCount++] = tempLine;
            tempLine = "";
        } else {
            tempLine += input[i];
        }
    }
}
```

3. Control the follow of sending file line by line

That determine proper error handling and logical structuring ensure the code is more readable and maintainable.
The code reads data from the serial input and waits for specific characters to excute specific operation due to every character

```
char c = Serial.read();
while (c != 'B') {
    c = Serial.read();
}

if (c == 'B') {
    delay(900);
    String lastLine = "";
    String currentLine = "";
    bool retransmitFromStart = false;

    while (true) {
        if (retransmitFromStart) {
            hexFile.seek(0, SeekSet);
            retransmitFromStart = false;
        }

        if (!hexFile.available())
            break;
        char c = hexFile.read();
        currentLine += c;
        Serial.write(c);
        Serial1.write(c);

        if (c == '\n') {
            lastLine = currentLine;
            currentLine = "";
```

- The code reads data from the serial input and waits for the character 'B' That mean flag 1 or 2 is successfully sending to microcontroller and boot loader in chip is ready to receive hex file of code
- Upon receiving 'B', it starts reading from a file (hex File)
- lastLine and currentLine are used to keep track of the lines being read.
- **Retransmission Flag** : The retransmitFromStart flag indicates whether to restart reading from the beginning of the file.
- **Reading and Transmitting Data**: The loop reads characters from a hex file and transmits them over serial
- **End of Line Handling**: When a newline character ('\n') is encountered, the current line is stored in lastLine, and currentLine is reset for the next line.

```
                char rec = Serial.read();
                while (rec != 'K') {
                    if (rec == 'N') {
                        Serial.print(lastLine);
                        Serial1.print(lastLine);
                    } else if (rec == 'F') {
                        retransmitFromStart = true;
                        break;
                    }
                    rec = Serial.read();
                }
            }

            if (retransmitFromStart) {
                continue;
            }
        }
        UpdateCommandfromUser = false;
        updated = true;
        Serial.write('\n');
        hexFile.close();
    }
```

- The code reads a hex file, line by line, transmitting each line over serial connections. It handles transmissions based on specific serial input commands
- When receiving character ‘k’
- That mean the line is successfully sending to chip and store in its memory and require to send the next line
- When receiving character ‘N’
  That meaning the line that was sent didn’t arrived successfully in memory and required resending the line again .
- 'F' for restarting from the beginning
- The process continues until the end of the file is reached or a specific condition triggers a retransmission. The flags and variables ensure proper flow and control over the file reading and transmission process.

Chapter 5: Mobile Application

5.1 Introduction

Mobile applications play a crucial role in facilitating Firmware Over-The-Air (FOTA) updates, particularly for devices such as the ESP32, a popular microcontroller used in IoT projects. FOTA updates enable the remote delivery of firmware updates, ensuring that devices are always running the latest version of their software without the need for physical access.While using a Raspberry Pi for FOTA updates is considered more professional due to its advanced capabilities, it is also significantly more expensive. To reduce costs and maintain efficiency, we opted to use a mobile application for FOTA updates. By choosing this approach, we were able to save 5000, making the project more cost-effective without compromising on functionality. This process is integral to the success of FOTA projects as it offers a streamlined, secure, and user-friendly method to manage firmware updates, ensuring devices are up-to-date and operating efficiently. As technology advances, these applications will continue to evolve, providing even more robust and intelligent features to support the growing landscape of IoT devices.

Figure 36

In this project, Flutter, an open-source UI software development kit by Google, is used to develop the mobile application, ensuring a seamless and efficient FOTA process.

Figure 37

5.2 Flutter

Flutter is an open-source UI software development kit (SDK) created by Google, designed to facilitate the development of natively compiled applications for mobile, web, and desktop from a single codebase. Launched in 2017, Flutter has quickly gained popularity among developers due to its ability to create high-quality, visually appealing, and performant applications efficiently.

5.2.1 Features of Flutter

1. Cross-Platform Development:
   Single Codebase: Developers can write one set of code that runs on multiple platforms, including iOS, Android, web, and desktop. This approach significantly reduces development time and effort.
   Consistent UI: Flutter ensures that the application's look and feel are consistent across different platforms, providing a uniform user experience.
2. High Performance:
   Native Compilation: Flutter applications are compiled directly to native ARM code for both iOS and Android, ensuring high performance and smooth animations.
   Fast Rendering: Powered by the Skia graphics engine, Flutter enables fast and efficient rendering of UI components.
3. Rich Set of Widgets:
   Customizable Widgets: Flutter provides a wide range of pre-designed widgets that can be customized to fit the specific needs of the application.
   Material Design and Cupertino: Flutter supports both Material Design for Android and Cupertino for iOS, allowing developers to create applications with a native look and feel.
4. Hot Reload:
   Real-Time Development: The hot reload feature allows developers to see changes in the code immediately without restarting the application. This accelerates the development process and makes it easier to experiment and iterate.
5. Strong Community and Ecosystem:
   Extensive Libraries and Plugins: Flutter has a rich ecosystem of libraries and plugins that extend its functionality, making it easier to integrate with various services and APIs.
   Active Community: A large and active community of developers contributes to Flutter's continuous improvement, providing support, resources, and best practices.

5.2.2 Benefits of Using Flutter

1. Faster Time-to-Market:Efficient Development: The single codebase approach and hot reload feature enable rapid development and quicker iteration cycles, reducing the time-to-market for applications.
   Reduced Costs: Developing one codebase for multiple platforms lowers development and maintenance costs.
2. Enhanced User Experience:
   Customizable UI: Flutter's rich set of widgets and flexible design capabilities allow developers to create highly customized and engaging user interfaces.

Smooth Performance: Native compilation and fast rendering ensure smooth performance and a responsive user experience.

3. Future-Proofing:
   Platform Independence: Flutter is not tied to a specific platform, which future-proofs applications against changes in operating systems and platform requirements.
   Constant Updates: Regular updates and contributions from Google and the developer community ensure that Flutter stays current with the latest technology trends and standards.

5.3 Basics of Dart Language for Flutter

When you write flutter apps, you use Dart, a programming language that was developed by google. It's relatively new; the first version of Dart was released on November 14, 2013.

It's now also official ECMA standard. It's open source, object oriented, strongly typed, class defined, and uses C-style syntax which is to say, it's like many other modern programming languages, including Java or C#, and to some extent, even JavaScript.

So, you might be wondering: why another language? I'm sure there isn't a unique answer to that question, but there are some features worth mentioning here that make Dart noteworthy, even without considering Flutter:

- It's easy to learn: if you have some knowledge of Java, C#, or JavaScript, Dart will be extremely easy to learn, as you'll see in the next few pages.
- It's aimed at productive: It's syntax is exceptionally concise and easy to read and debug.
- It can transpile to JavaScript, in order to maximize compatibility with web development.
- It has a general purpose: You can use it for client-side, server-side, and mobile development.
- Google is deeply involved in this project and has some big plans for Dart, including a new operating system, called Google Fuchsia.

Let's see Dart in action, which will make it easier to build your first Flutter project.

5.4 Getting Started with Flutter

After installing the necessary software, the next step is to set up your development environment and start building your first Flutter application. Here's how to get started:

Start the Android Studio:

- Go to Tools.
- Open SDK Manager.
- Under Appearance and Behavior, select System Settings.
- Click on Android SDK.
- In the SDK Tools tab, ensure that you choose both the Android Emulator and Android SDK Platform-Tools.

Setting Up the Flutter Environment:

- Install the Flutter SDK from the official Flutter website.
- Extract the downloaded Flutter SDK to the desired location on your system.
- Add the Flutter SDK path to your system's environment variables.

Creating a New Flutter Project:

- Open Android Studio.
- Select "Start a new Flutter project" from the welcome screen.
- Choose "Flutter Application" and click "Next."
- Configure your new Flutter project by providing a name, description, and desired location.
- Click "Finish" to create the project.

Running Your First Flutter App:

- Connect your mobile device or start an Android emulator.
- Click on the green "Run" button in Android Studio to build and run your Flutter application.
- You should see a sample Flutter app running on your device or emulator.

With these steps, you are now ready to start developing and experimenting with Flutter to create mobile app.

5.5 System Overview: ESP32 Firmware Update via Mobile App

In our system, we have integrated the ESP32 microcontroller with a mobile application using Firebase as the intermediary for update notifications. This setup ensures that the ESP32 can efficiently check for firmware updates and seamlessly communicate with the mobile app to keep the user informed and in control of the update process.

System Workflow:

The ESP32 is programmed to periodically check Firebase for any available firmware updates. It queries the Firebase database to determine if a new update is available for the car's system. If an update is found in Firebase, the ESP32 sends a flag to the mobile application indicating that a new update is available. If no update is found, a flag indicating that no updates are available is sent instead. Upon receiving the update notification flag, **the mobile app displays a message to the user:**

- If there is no update, the message reads, “No update available for your car.”

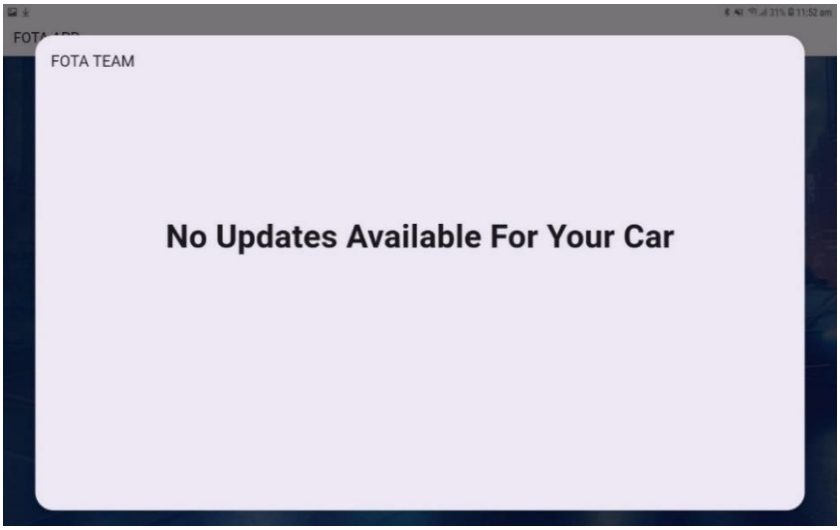


Figure 38

- If there is an update, the message reads, “New update available for your car” and presents two buttons: "Update Now" and "Update Later".

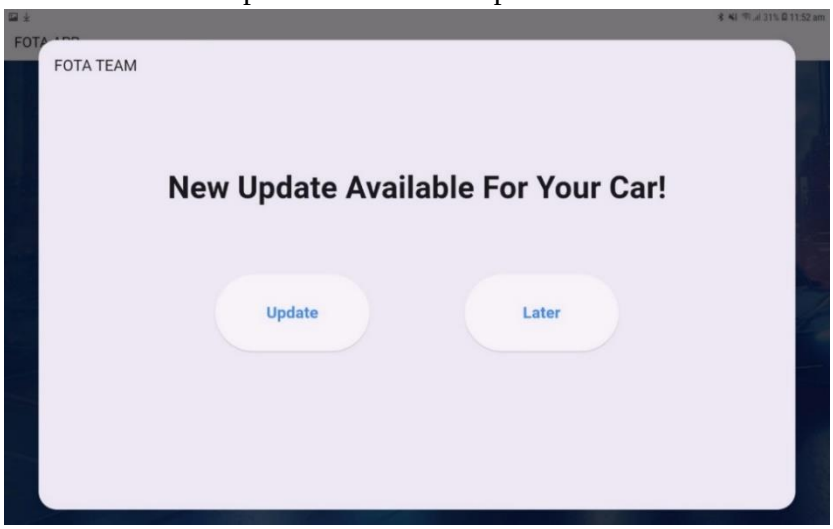


Figure 39

If the user selects "Update Later," the app sends a feedback flag back to the ESP32 indicating that the user chose to delay the update. If the user selects "Update Now," the app sends a feedback flag to the ESP32 instructing it to proceed with the firmware download. Upon receiving the "Update Now" flag from the mobile app, the ESP32 initiates the firmware download process. It begins downloading the new firmware. The ESP32 downloads the firmware via its UART interface and performs the update. During this process, it can communicate status updates back to the mobile app, providing real-time feedback to the user.

5.6 Mobile App Design and Icons

A graphical user interface (GUI) system in a car is crucial for modern vehicle functionality and user experience. A well-designed GUI allows users to interact with the car's features and systems efficiently and intuitively, enhancing safety, convenience, and overall satisfaction. It enables drivers to access and control various functionalities such as climate control, entertainment, navigation, and communication with ease, minimizing distractions and allowing them to focus on driving. Our mobile app integrates a comprehensive GUI system that includes intuitive icons for easy access to all essential features.

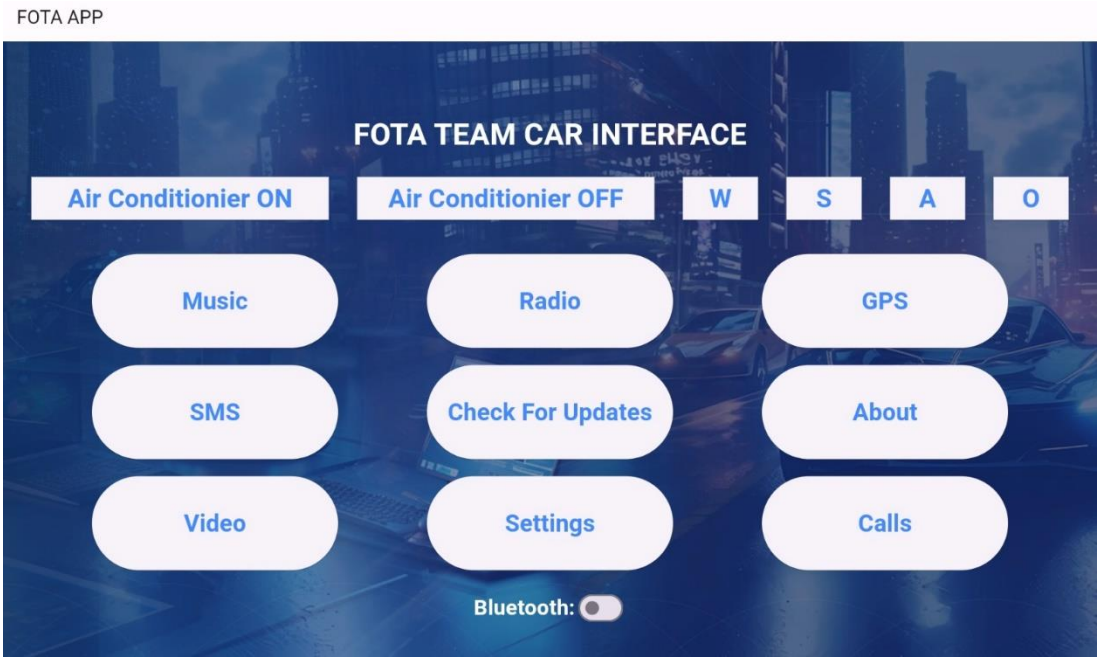


Figure 40

Music: The music icon opens the music player interface within the app. Here, you can browse your music library, play your favorite songs, create playlists, and control playback options such as play, pause, skip, and shuffle. It integrates seamlessly with your car's audio system for an enhanced listening experience.

Radio: Tapping the radio icon takes you to the radio interface, where you can tune into various radio stations. You can search for stations by frequency or browse through preset channels. This feature allows you to listen to live broadcasts, news, music, and talk shows while on the go.

GPS: The GPS icon opens the navigation system of the app. This feature provides real-time maps, turn-by-turn directions, traffic updates, and points of interest. It helps you navigate to your destination efficiently, offering alternative routes and voice guidance to ensure you stay on track.

SMS: Tapping the SMS icon opens the text messaging interface. You can read incoming messages, send new texts, and manage your conversations directly from the app. This feature is

designed to keep you connected while driving, with voice-to-text options available for safer communication.

Check for Updates: The check for updates icon allows you to see if there are any new firmware updates available for your car. When tapped, the app communicates with Firebase to check for updates. If an update is available, it notifies you with options to update now or later, ensuring your car's software is always up to date.

About: The about icon provides information about the app, including the version number, developers, and contact information. It is a resource for users who want to learn more about the app's background, check for updates, or find support and troubleshooting information.

Video: Tapping the video icon opens the video player within the app. This feature allows you to watch saved videos or stream video content directly on your mobile device. It supports various video formats and provides playback controls such as play, pause, fast-forward, and rewind.

Settings: The settings icon opens the app's settings menu. Here, you can customize various app preferences, such as notification settings, user profiles, and connectivity options. It allows you to tailor the app's functionality to your personal needs and ensure a personalized user experience.

Calls: The calls icon provides access to the phone call interface. You can make and receive calls directly from the app, sync your contacts, and use hands-free calling features. This ensures that you stay connected while driving, with safety and convenience in mind.

Air Conditioner On and Air Conditioner off: These icons control the manual mode of the car's air conditioning system. The "Air Conditioner On" icon activates the system, allowing you to cool or heat the interior of your vehicle to your desired temperature. You can prepare your car's climate before you start your journey or to adjust it while driving. The "Air Conditioner Off" icon deactivates the air conditioning system, helping conserve battery power or fuel when climate control is not needed. This function reverts to natural ventilation, it is an efficient way to stop the climate control and save energy.

When the update for the air conditioning system is downloaded, the "Air Conditioner On" icon will change, triggering a pop-up window **'Timer For Air Conditioner'**. This window will notify you that it is time to adjust the system, providing options for durations of half an hour or one hour to save energy.

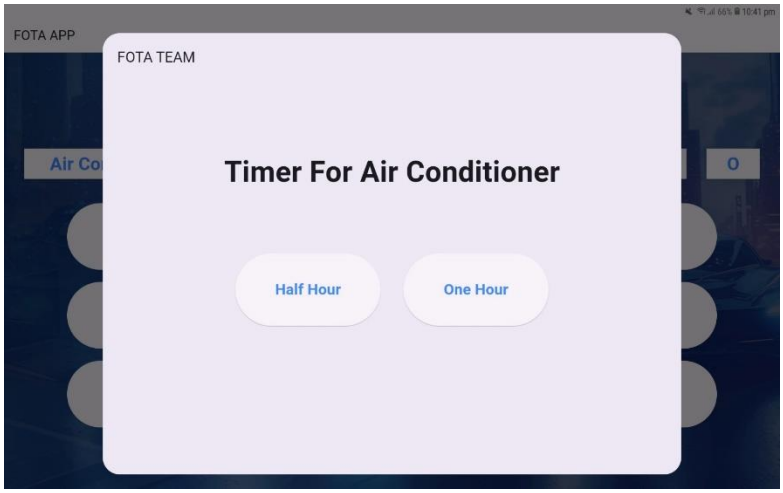


Figure 41

Mobile App Functionality for Car Control: The mobile application functions as a graphical user interface (GUI) to control the car's movements. The app provides buttons for specific commands that manage the car's operations directly. Here's a breakdown of the functionalities assigned to each button:

- **W for Forward Movement**: This button commands the car to move forward. When the user presses the "W" button, the app sends a flag to the car's control system to drive forward through esp.
- **S for Backward Movement**: This button commands the car to move backward. When the user presses the "S" button, the app sends a flag to the car's control system to reverse through esp.
- **A for Auto Parking**: This button activates the car's autoparking feature. When the user presses the "A" button, the app sends a flag to the car's control system through esp to initiate the automatic parking sequence, allowing the car to park itself in a designated spot.
- **O for Object Avoidance**: This button enables the car's object avoidance system. When the user presses the "O" button, the app sends a flag to the car's control system through esp to start monitoring for obstacles and avoid them during movement.

The mobile app uses these buttons to send commands to the car, allowing for easy and intuitive control. The app's interface is designed to be user-friendly, with clearly labeled buttons that correspond to the different functions.

5.7 Code Explanation and Functionality

This section explains the Flutter code, detailing each component's purpose and functionality within the FOTA (Firmware Over-The-Air) mobile application for controlling a car's features via an ESP32 module.

5.7.1 Flutter Imports

```
import 'package:flutter/material.dart';
import 'package:flutter/services.dart';
import 'package:http/http.dart' as http;
```

5.7.2 'Main' Functions

The **main()** function initializes the Flutter app and sets preferred screen orientations. Ensures the app starts with specific screen orientations suitable for landscape and portrait modes.

Code:

```
void main() {
  WidgetsFlutterBinding.ensureInitialized();
  SystemChrome.setPreferredOrientations([
    DeviceOrientation.portraitUp,
    DeviceOrientation.portraitDown,
    DeviceOrientation.landscapeLeft,
    DeviceOrientation.landscapeRight,
  ]);
```

```
  runApp(MyApp());
}
```

Explain:

The ‘WidgetsFlutterBinding.ensureInitialized();’ function ensures that Flutter bindings are initialized before executing further, which is crucial for the Flutter application to be fully ready to run and manage widgets. ‘WidgetsFlutterBinding’ is Flutter's binding class that connects the framework to the Flutter engine, and ‘ensureInitialized()’ ensures that all necessary initializations are completed, such as setting up communication channels with the native platform.

The ‘SystemChrome.setPreferredOrientations([...])’ function is used to set the preferred screen orientations for the application, configuring system-level settings on the device. It defines the orientations that the app should support, including ‘DeviceOrientation.portraitUp’ (portrait mode with the device upright), ‘DeviceOrientation.portraitDown ‘(portrait mode with the device upside down), ‘DeviceOrientation.landscapeLeft’ (landscape mode with the device rotated to the left), and ‘DeviceOrientation.landscapeRight’ (landscape mode with the device rotated to the right). This setup ensures that the app can display correctly in both portrait and landscape orientations, accommodating different device orientations.

The ‘runApp(MyApp());’ function runs the Flutter application with ‘MyApp’ as the root widget. ‘runApp()’ is a function provided by Flutter that starts the application and specifies the root widget. ‘MyApp()’ is an instance of the ‘MyApp’ class, which extends StatelessWidget. ‘MyApp’ represents the main structure of the application and defines its initial configuration, including the app bar, theme, and initial screen (‘CarTouchscreenUI’).

5.7.3 ‘MyApp’ Class

Code:

```
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('FOTA APP'),
        ),
        body: Container(
          decoration: BoxDecoration(
            image: DecorationImage(
              image: AssetImage('images/car_image.png'),
              fit: BoxFit.cover,
            ),
          ),
          child: CarTouchscreenUI(),
        ),
      ),
    );
  }
}
```

Explain:

The class declaration class ‘MyApp’ extends ‘StatelessWidget’ declares a new class named ‘MyApp’ which extends ‘StatelessWidget’. A ‘StatelessWidget’ is a widget that does not require mutable state, meaning it does not change once built.

The build method, annotated with ‘@override’, indicates that it overrides the same method in its superclass (‘StatelessWidget’). The ‘Widget build(BuildContext context)’ method describes the part of the user interface represented by this widget. The ‘BuildContext’ context provides the context in which the widget is built, allowing access to theme data, navigation, and other context-specific features.

The ‘MaterialApp’ widget, returned by the ‘build’ method, wraps several widgets commonly required for Material Design applications, providing basic functionalities like navigation and theming.

The home property, specified as ‘Scaffold’, defines the default route of the app and is typically the first screen displayed to users. The ‘Scaffold’ widget implements the basic visual structure of the Material Design layout, providing a framework for building UI elements such as an app bar, body content, and bottom navigation. It serves as a foundational structure for organizing and presenting the app's content.

The ‘appBar’ property, assigned with ‘AppBar’, defines a material design app bar that can accommodate widgets such as titles, icons, and actions. Within the ‘AppBar’, the title property is set to ‘Text('FOTA APP')’, specifying that the title displayed in the app bar is "FOTA APP". This configuration ensures that the app bar provides a consistent and recognizable interface for users, displaying essential information and actions at the top of the screen.

The Container widget is a flexible widget used to customize the layout and appearance of its child widget. In this context, the body property of ‘MaterialApp’ is set to a Container widget. Within the Container, the decoration property is defined with a ‘BoxDecoration’, allowing customization of the background behind its child. Inside ‘BoxDecoration’, a ‘DecorationImage’ is used to specify an image to display, loaded from the local assets using ‘AssetImage('images/car_image.png')’. The fit property of ‘DecorationImage’ is set to ‘BoxFit.cover’, ensuring that the image covers the entire area of the Container while maintaining its original aspect ratio. This setup allows the app to display a visually appealing background image within its main content area.

The ‘child’ property of the Container widget in this context sets ‘CarTouchscreenUI()’ as its child widget. ‘CarTouchscreenUI’ is a custom widget designed to represent the primary user interface of the car's touchscreen application. It encapsulates both the functionality and visual design tailored specifically for the application's requirements. By assigning ‘CarTouchscreenUI()’ as the child of the Container, the app ensures that the content and behavior defined within ‘CarTouchscreenUI’ are displayed and interactable within the container's area on the screen. This approach allows for a structured and modular design where different parts of the application's UI can be managed independently while maintaining a cohesive user experience.

5.7.4 ‘CarTouchscreenUI’ Class

Code:

```
class CarTouchscreenUI extends StatefulWidget {
  @override
  _CarTouchscreenUIState createState() => _CarTouchscreenUIState();
  }
```

The ‘CarTouchscreenUI’ class in Flutter is a special kind of widget that can change its appearance and behavior over time. It's marked with ‘StatefulWidget’, meaning it can hold and manage its own internal state, which includes things like data or settings that can change. The ‘@override’ part before ‘createState()’ indicates that we're customizing how this widget creates and handles its state. The ‘_CarTouchscreenUIState’ part is where the actual logic and state management happen behind the scenes. This setup allows the widget to update itself when needed, making it dynamic and responsive to user interactions or changes in the app.

```
class _CarTouchscreenUIState extends State<CarTouchscreenUI> {
  double volume = 0.5;
  bool isBluetoothConnected = false;
  late BuildContext builderContext;
```

State Class Declaration: ‘_CarTouchscreenUIState’ extends ‘State<CarTouchscreenUI>’, indicating that it manages the state for the ‘CarTouchscreenUI’ widget. It initializes variables like volume and ‘isBluetoothConnected’, which are used to manage the UI state within the widget.

```
@override
Widget build(BuildContext context) {
  return Center(
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        Text(
          '\nFOTA TEAM CAR INTERFACE',
          style: TextStyle(fontSize: 35,color: Colors.white, fontWeight:
FontWeight.bold),
        ),
```

Build Method: The ‘build’ method overrides the method from the State class and returns the UI components that make up the ‘CarTouchscreenUI’ widget. It centers a column of widgets vertically and horizontally in the screen.

Title Text Widget: Displays the title "FOTA TEAM CAR INTERFACE" with specific styling.

5.7.5 Air Conditioner Buttons:

Code:

```
Row(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: [
    CarButton2(
      label: 'Air Conditionier ON',
      onPressed: () {
        fdownload2(context);
      },
    ),
    CarButton2(
      label: 'Air Conditionier OFF',
      onPressed: () {
        fdownload3(context);
      },
    ),
  ],
),
SizedBox(height: 20),
SizedBox(height: 20),
```

- **CarButton2 Class:**

```
class CarButton2 extends StatelessWidget {
  final String label;
  final VoidCallback onPressed;
  CarButton2({required this.label, required this.onPressed});
  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.only(top: 25.0), // Add top margin here
      child: ElevatedButton(
        style: ElevatedButton.styleFrom(
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.zero, // Makes the button rectangular
          ),
        ),
        onPressed: onPressed,
        child: Container(
          width: 300, // Adjust the width as needed
          height: 50, // Adjust the height as needed
          alignment: Alignment.center, // Center the label text
          child: Text(
            label,
            style: TextStyle(
              fontSize: 30,
              color: Colors.blueAccent,
              fontWeight: FontWeight.bold,
            ), // Adjust the font size as needed
          ),
        ),
      ),
    );
```

```
  }
}
```

- **fdownload2 Class:**

```
Future<void> fdownload2(BuildContext context) async {
  Uri uri = Uri.parse('http://192.168.4.1/ledon');
  http.Response res = await http.get(uri);
}
```

Purpose: Sends an HTTP GET request to http://192.168.4.1/ledon.To turn the led on
Steps:

1. Parses the URI for the request endpoint.
2. Uses `http.get(uri)` to asynchronously send the GET request and await its response.

- **fdownload3 class:**

```
Future<void> fdownload3(BuildContext context) async {
  Uri uri = Uri.parse('http://192.168.4.1/ledoff');
  http.Response res = await http.get(uri);
}
```

Purpose: Initiates an HTTP GET request to http://192.168.4.1/ledoff.To turn the led off
Steps:

1. Parses the URI for the request endpoint.
2. Uses `http.get(uri)` to asynchronously send the GET request and await its response.

- **When we update the Air Conditioner App:(pop up window)**

```
void showUpdateDialog3(BuildContext context) {
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: Text('FOTA TEAM'),
        content: Column(
          children: [
            Row(
              children: [
                Expanded(
                  child: Text(
                    '\n\nTimer For Air Conditioner',
                    style: TextStyle(fontSize: 48, fontWeight:
FontWeight.bold),
                  ),
                ),
              ],
            ),
            SizedBox(
```

```
              height: 100,
              width: 200,
            ),
            Row(
              mainAxisAlignment: MainAxisAlignment.spaceAround,
              children: [
                ElevatedButton(
                  onPressed: () {
                    halfh(context);
                  },
                  style: ElevatedButton.styleFrom(
                    padding: EdgeInsets.all(20),
                  ),
                  child: Container(
                    width: 200,
                    height: 80,
                    child: Center(
                      child: Text(
                        'Half Hour',
                        style: TextStyle(fontSize: 25,color:
Colors.blueAccent, fontWeight: FontWeight.bold),
                      ),
                    ),
                  ),
                ),
                ElevatedButton(
                  onPressed: () {
                    oneh(context);
                  },
                  style: ElevatedButton.styleFrom(
                    padding: EdgeInsets.all(20),
                  ),
                  child: Container(
                    width: 200,
                    height: 80,
                    child: Center(
                      child: Text(
                        'One Hour',
                        style: TextStyle(fontSize: 25,color:
Colors.blueAccent, fontWeight: FontWeight.bold),
                      ),
                    ),
                  ),
                ),
              ],
            ),
          ],
        ),
        contentPadding: EdgeInsets.symmetric(horizontal: 200),
      );
    },
  );
}
```

The ‘showUpdateDialog3’ function displays a dialog in the context of a Flutter application. It uses the ‘showDialog’ function to create an ‘AlertDialog’ with a title "FOTA TEAM" and a main content section consisting of a column layout. Within this column, a row displays the text "Timer For Air Conditioner" styled with a font size of 48 and bold weight. A ‘SizedBox’ is used to provide spacing with a height of 100 and width of 200. Another row contains two ‘ElevatedButton’ widgets aligned with ‘mainAxisAlignment: spaceAround’. Each button is styled with padding, a specified width and height, and displays centered text in blue with a font size of 25 and bold weight. The first button triggers the ‘halfh’ function when pressed, and the second triggers the ‘oneh’ function. The overall content padding of the dialog is set with horizontal padding of 200 units to properly space the dialog's content within the screen.

- **Halfh Class:**

```
Future<void> halfh(BuildContext context) async {
  Uri uri = Uri.parse('http://192.168.4.1/autotimerhalf');
  http.Response res = await http.get(uri);
}
```

Explain:

1. **Function Declaration and Asynchronicity**:

- **`Future<void> halfh(BuildContext context) async { ... }`:** Declares an asynchronous function `halfh` that returns a `Future<void>`. The `async` keyword indicates that this function will perform asynchronous operations.

2. **Creating the URI**:

- **`Uri uri = Uri.parse('http://192.168.4.1/autotimerhalf');`:** Creates a `Uri` object by parsing the provided URL string `'http://192.168.4.1/autotimerhalf'`. This URL is assumed to be the endpoint for setting a timer for half an hour on a device, such as an ESP32.

3. **Sending the HTTP GET Request**:

- **`http.Response res = await http.get(uri);`:** Sends an HTTP GET request to the specified URI. The `await` keyword is used to wait for the response from the server before proceeding. The response is stored in the `res` variable of type `http.Response`.

- **oneh Class:**

```
Future<void> oneh(BuildContext context) async {
  Uri uri = Uri.parse('http://192.168.4.1/autotimerone');
  http.Response res = await http.get(uri);
}
```

Explain:

1. **Function Declaration and Asynchronicity**:

- **`Future<void> oneh(BuildContext context) async { ... }`:** Declares an asynchronous function `oneh` that returns a `Future<void>`. The `async` keyword indicates that this function will perform asynchronous operations.

2. **Creating the URI**:

- **`Uri uri = Uri.parse('http://192.168.4.1/autotimerone');`:** Creates a `Uri` object by parsing the provided URL string `'http://192.168.4.1/autotimerone'`. This URL is assumed to be the endpoint for setting a timer for one hour on a device, such as an ESP32.

3. **Sending the HTTP GET Request**:

- **`http.Response res = await http.get(uri);`:** Sends an HTTP GET request to the specified URI. The `await` keyword is used to wait for the response from the server before proceeding. The response is stored in the `res` variable of type `http.Response`.

- **autotimer**

```
Future<void> autotimer(BuildContext context) async {
  Uri uri = Uri.parse('http://192.168.4.1/autotimer');
  http.Response res = await http.get(uri);
  if (res.statusCode == 200 && res.body == "autotimer") {
    showUpdateDialog3(context);
  } else {
    fdownload2(context);
  }
}
```

Explain:

1. **Function Declaration and Asynchronous Nature**:

- **`Future<void> autotimer(BuildContext context) async`:** The function `autotimer` is declared to return a `Future<void>`, indicating it is asynchronous and will perform operations that

might take time to complete without blocking the execution of other code. The `async` keyword allows the use of `await` within the function.

2. **Parsing the URI**:

- **`Uri uri = Uri.parse('http://192.168.4.1/autotimer');`**: A `Uri` object is created by parsing the string `'http://192.168.4.1/autotimer'`. This string represents the endpoint where the request will be sent.

3. **Sending the HTTP GET Request**:

- **`http.Response res = await http.get(uri);`**: An HTTP GET request is sent to the parsed URI. The `await` keyword ensures the function waits for the response before continuing.

4. **Handling the Response**:

- **`if (res.statusCode == 200 && res.body == "autotimer") { ... }`**: The function checks if the response status code is 200, indicating a successful request, and if the response body contains the string `"autotimer"`. If both conditions are true, the function `showUpdateDialog3(context)` is called to display a dialog.

5.7.6 Check For Updates Button:

```
CarButton(
  label: 'Check For Updates',
  onPressed: () {
    fdownload(context);
  },
),
```

- **CarButton Class:**

```
class CarButton extends StatelessWidget {
  final String label;
  final VoidCallback onPressed;
  CarButton({required this.label, required this.onPressed});
  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: onPressed,
      child: Container(
        width: 240, // Adjust the width as needed
        height: 110, // Adjust the height as needed
        child: Center(
          child: Text(
            label,
            style: TextStyle(fontSize: 28,color: Colors.blueAccent,
```

```
fontWeight: FontWeight.bold), // Adjust the font size as needed
          ),
        ),
      ),
    );
  }
}
```

The ‘CarButton’ class is a ‘StatelessWidget’ that creates customized buttons for a car interface. It takes two required parameters in its constructor: label, which specifies the button's text, and ‘onPressed’, a function executed when the button is pressed. Inside the ‘build’ method, it returns an ‘ElevatedButton’ wrapped in a Container that defines the button's size and alignment. The Text widget inside the ‘Container’ displays the label text with specified style attributes such as font size, color (‘blueAccent’), and font weight (‘bold’). This setup allows for easy creation of consistent, visually appealing buttons tailored for different actions within the car's touchscreen UI.

- **Fdownload:**

```
Future<void> fdownload(BuildContext context) async {
  Uri uri = Uri.parse('http://192.168.4.1/on');
  http.Response res = await http.get(uri);
  if (res.statusCode == 200 && res.body == "FileDownloaded") {
    showUpdateDialog(context);
  } else {
    showUpdateDialog2(context);
  }
}
```

Purpose: Initiates an HTTP GET request to http://192.168.4.1/on.
Steps:

1. Parses the URI for the request endpoint.
2. Uses http.get(uri) to asynchronously send the GET request and await its response.
3. Checks if the response status code is 200 and if the body of the response equals "FileDownloaded".
4. Calls showUpdateDialog(context) if the conditions are met, indicating a new update is available.
5. Otherwise, calls showUpdateDialog2(context) if the conditions are not met, indicating no updates are available.

- **UpdateCommandTrue Class:**

```
Future<void> updateCommandTrue(BuildContext context) async {
  Uri uri = Uri.parse('http://192.168.4.1/update');
  http.Response res = await http.get(uri);
}
```

Purpose: Sends an HTTP GET request to http://192.168.4.1/update. When the user chooses ‘update now’ it goes to execute this command and return to esp to update the car.

Steps:

1. Parses the URI for the request endpoint.
2. Uses http.get(uri) to asynchronously send the GET request and await its response.

- **If there is a new update:(pop up window)**

```
void showUpdateDialog(BuildContext context) {
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: Text('FOTA TEAM'),
        content: Column(
          children: [
            Row(
              children: [
                Expanded(
                  child: Text('\n\nNew Update Available For Your Car!',
                    style: TextStyle(fontSize: 48, fontWeight:
FontWeight.bold),
                  ),
                ),
              ],
            ),
            SizedBox(
              height: 100,
              width: 200,
            ),
            Row(
              mainAxisAlignment: MainAxisAlignment.spaceAround,
              children: [
                ElevatedButton(
                  onPressed: () {
                    updateCommandTrue(context); // Hena hn7ot el update
                  },
                  style: ElevatedButton.styleFrom(
                    padding: EdgeInsets.all(20),
                  ),
                  child: Container(
                    width: 200, //
                    height: 80, //
                    child: Center(
                      child: Text('Update',
                        style: TextStyle(fontSize: 25,color:
Colors.blueAccent, fontWeight: FontWeight.bold),
                      ),
                    ),
                  ),
                ),
                ElevatedButton(
                  onPressed: () {
                    Navigator.of(context).pop();
                  },
                  style: ElevatedButton.styleFrom(
```

```
                    padding: EdgeInsets.all(20),
                  ),
                  child: Container(
                    width: 200, // Adjust the width as needed
                    height: 80, // Adjust the height as needed
                    child: Center(
                      child: Text('Later',
                        style: TextStyle(fontSize: 25,color:
Colors.blueAccent, fontWeight: FontWeight.bold),
                      ),
                    ),
                  ),
                ),
              ],
            ),
          ],
        ),
        contentPadding: EdgeInsets.symmetric(horizontal: 200),
      );
    },
  );}
```

- **If there is a no update:(pop up window)**

```
void showUpdateDialog2(BuildContext context) {
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: Text('FOTA TEAM'),
        content: Column(
          children: [
            Row(
              children: [
                Expanded(
                  child:Center(
                    child: Text(
                      '\n\n\nNo Updates Available For Your Car\n',
                      style: TextStyle(fontSize: 50, fontWeight:
FontWeight.bold),
                    ),
                  ),
                ),
              ],
            ),
            SizedBox(
              height: 100,
              width: 200,
            ),
          ],
        ),
        contentPadding: EdgeInsets.symmetric(horizontal: 200),
      );
    },
```

5.6.7 Auto parking car interface buttons

- **Button Appearance and Functionality**

```
CarButton3(
  label: 'W',
  onPressed: () {
    a1(context);
  },
),
CarButton3(
  label: 'S',
  onPressed: () {
    a2(context);
  },
),
CarButton3(
  label: 'A',
  onPressed: () {
    a3(context);
  },
),
CarButton3(
  label: 'O',
  onPressed: () {
    a4(context);
  },
),


class CarButton3 extends StatelessWidget {
  final String label;
  final VoidCallback onPressed;

  CarButton3({required this.label, required this.onPressed});

  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.only(top: 25.0), // Add top margin here
      child: ElevatedButton(
        style: ElevatedButton.styleFrom(
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.zero, // Makes the button rectangular
          ),
        ),
        onPressed: onPressed,
        child: Container(
          width: 40, // Adjust the width as needed
          height: 50, // Adjust the height as needed
          alignment: Alignment.center, // Center the label text
          child: Text(
            label,
            style: TextStyle(
              fontSize: 30,
              color: Colors.blueAccent,
              fontWeight: FontWeight.bold,
```

```
Future<void> a1(BuildContext context) async {
  Uri uri = Uri.parse('http://192.168.4.1/a1');
  http.Response res = await http.get(uri);
}

Future<void> a2(BuildContext context) async {
  Uri uri = Uri.parse('http://192.168.4.1/a2');
  http.Response res = await http.get(uri);
}



Future<void> a3(BuildContext context) async {
  Uri uri = Uri.parse('http://192.168.4.1/a3');
  http.Response res = await http.get(uri);
}


Future<void> a4(BuildContext context) async {
  Uri uri = Uri.parse('http://192.168.4.1/a4');
  http.Response res = await http.get(uri);
}
```

Chapter 6 : Bootloader

6.1. Introduction:

A bootloader is an application whose primary purpose is to allow a systems software to be updated without the use of specialized hardware such as a JTAG programmer and SWD programmer to load the user application into MCU. In certain cases, it can also be the earliest point at which the integrity of an embedded system can be checked. The bootloader manages the systems images. There are many different sizes and flavors to embedded bootloaders. They can communicate over a variety of protocols such as USART, CAN, I2C, Ethernet, USB and the list goes on for as many protocols that exist. stores it in the Flash memory, usually using the Flash Controller Peripheral of the microcontroller. Systems with bootloaders have at least two program images coexisting on the same micro-controller and must include branch code that performs a check to see if an attempt to update software is in progress.
such as :
* JTAG (Joint Test Action Group) was originally used to test chips. This group initiated discussions in the late 1980s and formally released documents interpreting IEEE standards in 1990. It is used for boundary scans, checking faults in chips /boards in productions debugging and flashing micros was an evolution in its applications over the time. JTAG may be necessary to gain more control and functionality
* SWD, Serial Wire Debug, is a protocol designed by ARM for programming and debugging their microcontrollers. It is based on the ARM Core Sight debugging architecture and enables reading and writing of chip registers through data packets. SWD is a two-wire protocol for accessing ARM debugging interfaces. It is part of the ARM Debug Interface Architecture Specification and serves as an alternative to JTAG and SWD may be a better fit for resource-constrained systems.

Protocol	SWD	JTAG
Supported CPU	ARM Cortex-M	Independent group "not limited to ARM"
Special feature	Printing debugging information	Not supported
Complexity	simpler, more efficient interface	complex systems with comprehensive debugging needs.

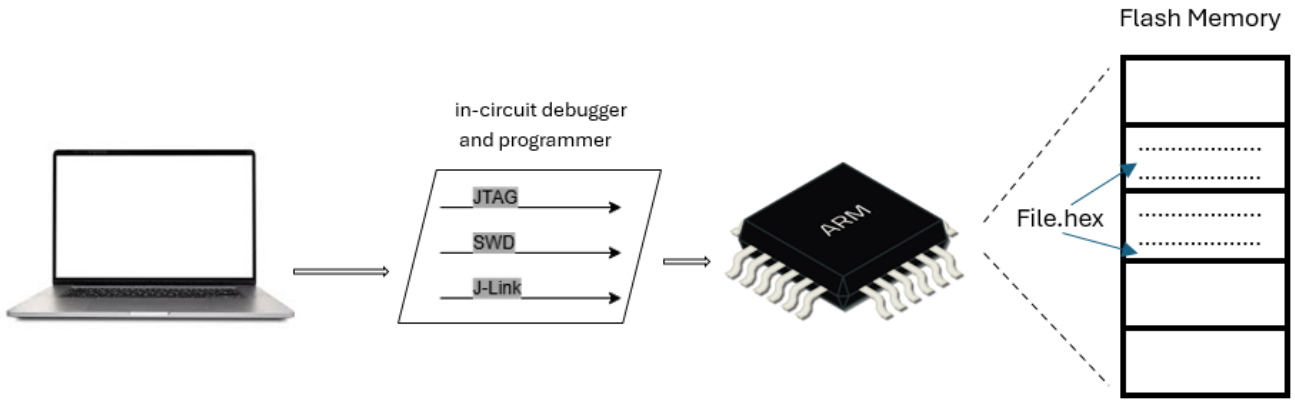


Figure 42

6.2. Memory Architecture

Every microcontroller has some form of non-volatile memory that is used to store the program. The most used type of memory is flash. Flash is broken up into divisible sections. Sectors are in turn organized into larger structures known as blocks. Each microprocessor is different as to how these sections of flash can be manipulated. Most will allow you to write a single byte to flash at a time. Others may require 8 bytes or 256 bytes be written at one time.

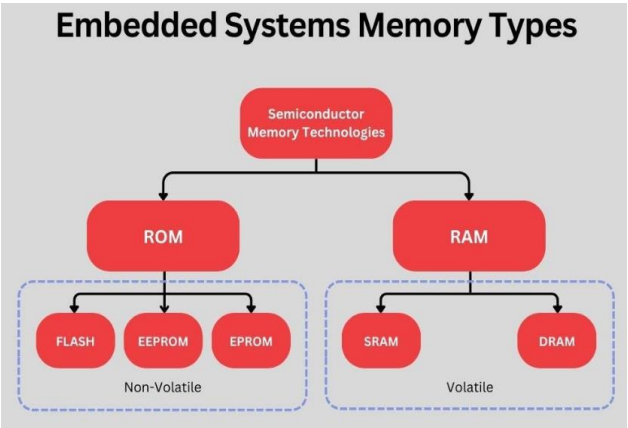


Figure 43

The main flash memory starting from address 0x0800000 to 0x0801FFFF should typically contain three sections: Branching Section, Bootloader Section and Application Section. The branching section is used to jump to either the application or the bootloader and it can be a part of the bootloader or has an individual section in the Flash Memory. In this project, the branching code is a part of the bootloader section, thus, the main flash memory has two sections: Bootloader section

and Application Section.
The microcontroller used in the project, “STM32F401RC”, is a medium-density device with a 256KB flash memory with 6 sectors memory starting from memory address 0x0800000 to memory address 0x0803FFFF . The Information block (System Memory) in the flash memory starts from memory address 0x1FFF0000. It has 30KB. They aren’t erased after reset which can be handy in storing information without the need of an external EEPROM.
The standard STM32F401RC Bootloader supports several communication protocols, however it doesn’t support CAN Bootloader, so a custom bootloader in the program memory is developed to flash the program in the memory.

Table 5. Flash module organization (STM32F401xB/C and STM32F401xD/E)

Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	Sector 7	0x0806 0000 - 0x0807 FFFF	128 Kbytes
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

Figure 44

It is important that before a designer gets too far in their boot-loader design that they pull out the microcontroller datasheet and read through the flash and memory organization chapters. The primary purpose for detailed examination of the memory map is to determine what sections of flash are available and best used for the bootloader.
There are several factors that should be considered when selecting where to locate the bootloader.
1- Bootloader size.
2- Vector table location.
3- Write protection and code security flash section.

6.2.1 Memory Partitions:

a) **Bootloader**:
It is a small application that is used to flash application in flash memory.
b) **Application 1**:
It is the version one of application, these space from the flash, the bootloader loads the application hex file in it.
c) **Application 2**:
It is the second version of the application 1, it run if the data corruption when the bootloader flash update or another file in the space of applicatin1.

d) **Updated flag**:

It is the indication that indicates if there is any application and its sector to determine which sector to erased.

Bootloader **Sector 0 : 16Kb** **0x0800 0000 - 0x0800 3FFF**	**Updated flag** **Sector 3** **0x0800 C000**
Application 1 **Sector 4 : 64Kb** **0x0801 0000 - 0x0801 FFFF**	**Application 2** **Sector 5 : 128Kb** **0x0802 0000 - 0x0803 FFFF**

6.3. Bootloader Requirements

Each bootloader will have its own unique set of requirements based on the type of application; however, there are still a few general requirements that are common to all bootloaders.

1-Ability to switch or select the operating mode (Application or bootloader).

2- Communication protocol requirements (USB, CAN, I2C, UART).

3- Record parsing requirement (S-Record, hex).

4- Flash system requirements (erase, write, read).

5- Error detection mechanism (checksum, CRC, …).

6.4. Intel Hex Format

Now that we understand the use of the Flash Memory Controller, the next step is to study the hex file format to be able to flash the program in the flash memory. Intel hexadecimal object file format, Intel hex format or Intellec Hex is a file format that conveys binary information in ASCII text form. It is commonly used for programming microcontrollers, EPROMs and other types of programmable logic devices. In a typical application, a compiler or assembler converts a program's source code (such as in C or assembly language) to machine code and outputs it into a HEX file. Common file extensions used for the resulting files are HEX or H86. The HEX file is then read by a programmer to write the machine code into a PROM or is transferred to the target system for loading and execution.

Intel HEX consists of lines of ASCII text that are separated by line feed or carriage return characters or both. Each text line contains hexadecimal characters that encode multiple binary numbers. The binary numbers may represent data, memory addresses, or other values, depending on their position in the line and the type and length of the line. Each text line is called a record.

The Record structure is 6 parts as follows:

1- Start code, one character, an ASCII colon ':'.

2- Byte count, two hex digits (one hex digit pair), indicating the number of bytes (hex digit pairs) in the data field. The maximum byte count is 255 (0xFF). 16 (0x10) and 32 (0x20) are commonly used byte counts. 16 is the most used in the hex files used in our project, however other byte numbers may also be found.

3- Address, four hex digits, representing the 16-bit beginning memory address offset of the data. The physical address of the data is computed by adding this offset to a previously established base address, thus allowing memory addressing beyond the 64 kilobyte limit of 16-bit addresses. The base address, which defaults to zero, can be changed by various types of records. Base addresses and address offsets are always expressed as big endian values.
4- Record type, two hex digits, 00 to 05, defining the meaning of the data field. The record types are explained in details after Record Structure.
5- Data, a sequence of n bytes of data, represented by 2n hex digits. Some records omit this field (n equals zero). The meaning and interpretation of data bytes depends on the application.
6- Checksum, two hex digits, a computed value that can be used to verify the record has no errors.

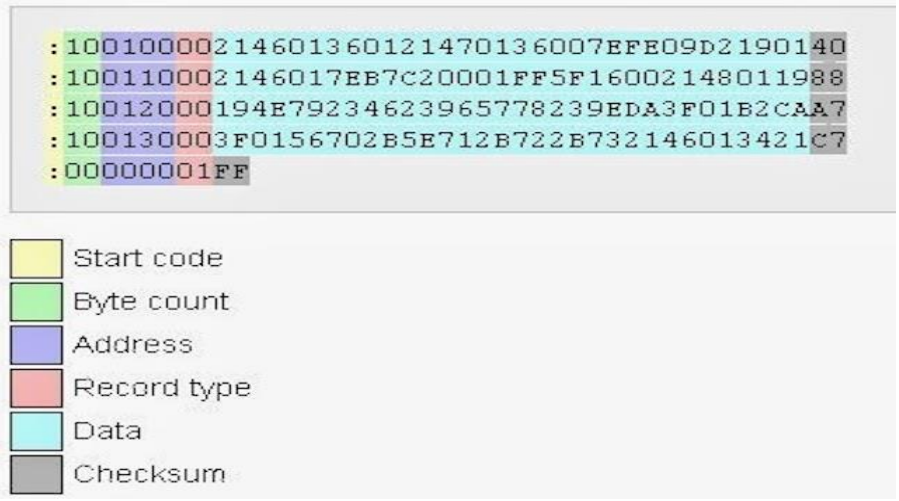


Figure 5.4 hex record

Figure 45

As mentioned, Record type may have a value from **00** to **05** as follows:
1- **00:** Data. Contains data and a 16-bit starting address for the data. The byte count specifies number of data bytes in the record. Example: (:10280000005000202 12A0008272A00082B2A00084F)
This line contains hex data: 0x02000500, 0x08002A21, 0x08002A27, 0x08002A2B respectively. Their memory locations are determined according to the Extended Linear Address record before them.
2- **01:** End of File. It must occur exactly once per file in the last line of the file. The data field is empty (thus byte count is 00) and the address field is typically 0000. Example: (:00000001FF)
3- **02:** Extended Segment Address. The data field contains a 16-bit segment base address (thus byte count is always 02) compatible with 80x86 real mode addressing. The address field (typically 0000) is ignored. The segment address from the most recent 02 record is multiplied by 16 and added to each subsequent data record address to form the physical starting address for the data. This allows addressing up to one megabyte of address space.
4- **03:** Start Segment Address. For 80x86 processors, specifies the initial content of the CS (code segment): IP (instruction pointer) registers (i.e., the starting execution address). The address field is 0000, the byte count is always 04, the first two data bytes are the CS value, the latter two are the IP value. Since the startup code manages the memory sections and code segments and the bootloader

manages the execution start address of the application, this record is also not present in the hex files used in the project.
5- **04:** Extended Linear Address. Allows for 32 bit addressing (up to 4GiB). The record's address field is ignored (typically 0000) and its byte count is always 02. The two data bytes (big endian) specify the upper 16 bits of the 32 bit absolute address for all subsequent type 00 records; these upper address bits apply until the next 04 record. The absolute address for a type 00 record is formed by combining the upper 16 address bits of the most recent 04 record with the low 16 address bits of the 00 record. If a type 00 record is not preceded by any type 04 records then its upper 16 address bits default to 0000. This record is usually the first record in the hex file; however, it can be repeated anywhere in the hex file if the base address if the data is changed. Example: (:020000040800F2)
In this record, the address is 0x0800, which is concatenated to the address in the data record to get the final address. For example, in the previous data record example:
(:10280000005000202 12A0008272A00082B2A00084F)
The data will be stored starting from address 0x08002800.
6- **05:** Start Linear Address. The address field is 0000 (not used) and the byte count is always 04. The four data bytes represent a 32-bit address value (big-endian). [18] It represents the start address of the program. Example: (:040000050800290DB9)
The start address of this program is 0x0800290D.

6.4.1. The Hex Data Processor Module

It is responsible for taking the hex record and extracting the data and addresses and flashing the data in the flash memory. The module is divided into 4 states/functions in order to decrease the code complexity:
1- Check Validation**:**
In this step, the software checks if the hex record is a valid hex record by checking the start code ‘:’ and checking that the number of the characters in the record is an even number and not exceeding the maximum record number.
2- Convert to Hex:
Since the hex file is received in ASCII format, this step converts the ASCII format into hexadecimal bytes format.
3- Parse Line:
This step parses the hex data into corresponding fields of the record structure into a structure and determine the record type. It also stores the data in a 32-bit array in the structure since the flash memory architecture and flash memory module deals with 32-bit data type Since the data in the hex line is in big endian format, but the data format in the system is little endian, it rearranges the data as desired. The step of rearranging the data is only done in case of a data record as the other records don’t need this step. It also adds 0xFF if the data length doesn’t complete a 32-bit variable, so that the next record can program the rest of the data. This step also checks the CRC Checksum and returns a NOK in case of error.

4- Write Data to flash:
In this step, an action is done according to the record type: in case odd data, 32-bit chunks of data are sent one by one to the flash controller to program them according to the last extended linear address and the address in the data record. In case of extended linear address, the address is stored in a static variable to be used when storing the data.

6.5. The Bootloader System

Bootloaders can come in many different sizes and in many different flavors but in general the operation of a system with a bootloader is relatively standard. There are three major components to these systems that can be seen in Figure 1. They are the branching code (green), the application code (blue) and the bootloader code (red). For most systems we prefer them to be executing the application path the majority of their operational life. Figure 1 highlights the execution path to get to the application by the dashed red line. The orange block is a common block used by both the bootloader and the application to reset the system.

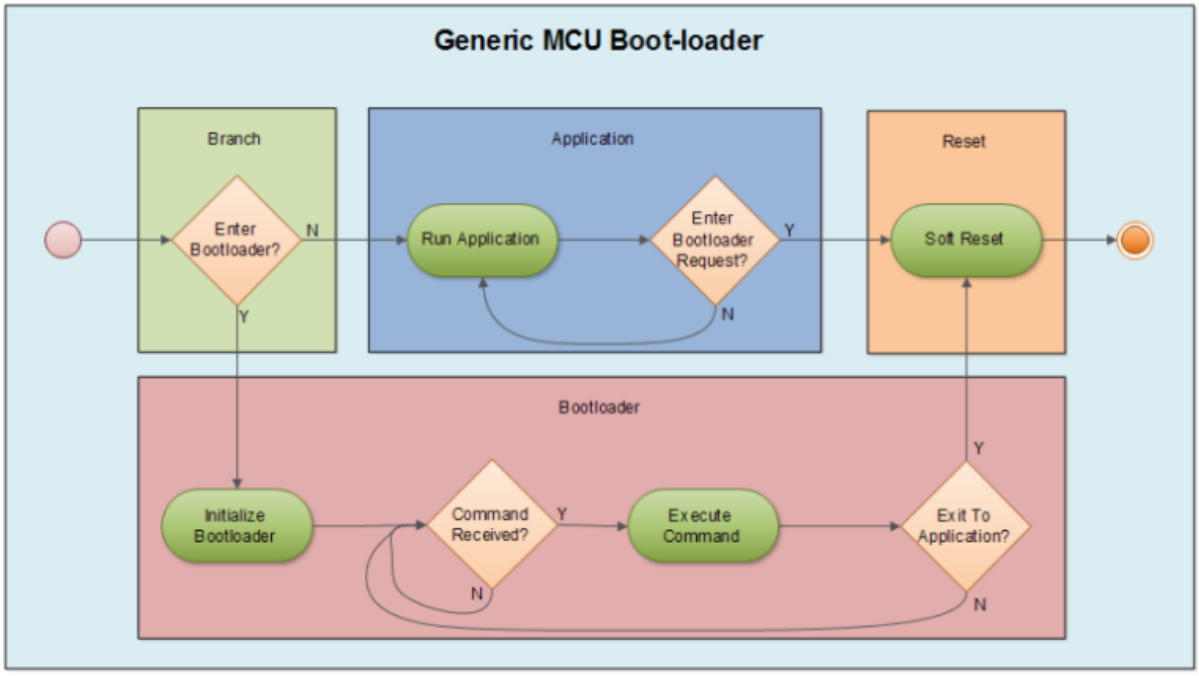


Figure 46

The branching code (green) makes the decision as to whether the bootloader is loaded or whether the application is loaded. The branching code in a very simple system can be nothing more than checking a GPIO for a certain state to enter the bootloader; however, such basic systems are usually only used by chip manufacturers in order to demonstrate the functionality of the chip and is usually not recommended if a robust boot-loading solution is required. In a more complex system, the bootloader may completely load itself into memory and perform some basic system functions for a short period of time before making the branching decision possibly to perform a system integrity check. For this reason, in most

cases the branching code is included in the bootloader. The application code (blue) is executed only after the branching code has made the decision that there are no requests to enter the bootloader and that the application code is safe to execute.

Even while the application is loading and performing its duties, it should be designed so that it can accept a command to enter the bootloader. When the application receives such a request the application will perform any cleanup that needs to be performed and when it is safe to do so the application will perform a soft reset of the system. In most cases this is done by writing incorrect values to the watchdog timer that in turn will then cause a power-on-reset of the system.
Additional details of how to make the branching code recognize. The boot-loader code (red) performs all the functions of the bootloader. When the bootloader is loaded into memory it will begin initializing a basic subset of peripherals that are required for the bootloader to perform all its functions. These peripherals are typically the system clock, interrupt service routines and tables, a communication peripheral and perhaps a state-machine or basic task scheduler. This allows the bootloader to communicate with the outside world and accept commands to perform flashing instructions.

6.5.1 Bootloader Main Sequence

when the microcontroler is powerd , the bootlodar starts and waits for some seconds if there any update or not. if there is any update . it begins to check flag to know which applications runs so there are three cases according to updated flag :

1-if updated flag ‘s value equal to “0xFFFFAAAA” , the application 1 is active ,so the bootloader will erase other sector “5” to write new update on.
2- if updated flag ‘s value equal to “0xFFFFBBBB” , the application 2 is active ,so the bootloader will erase other sector “4” to write new update on.
3- if updated flag ‘s value equal to “0xFFFFFFFF” , no application exists so the bootloader will erase other sector “4” to write new update on.

After erasing specific sector ,
1-it starts to recive records through UART.
2- hex parser makes a checksum to verify the data is received correctly.
3- Then write the record in flash.
4- With each iteration repeat this step from receive record
the bootloader initiates the main application after successfully updating the firmware. Provide insights into how the bootloader jumps to the application start address and begins the execution of the new firmware.

6.6. State machine diagram

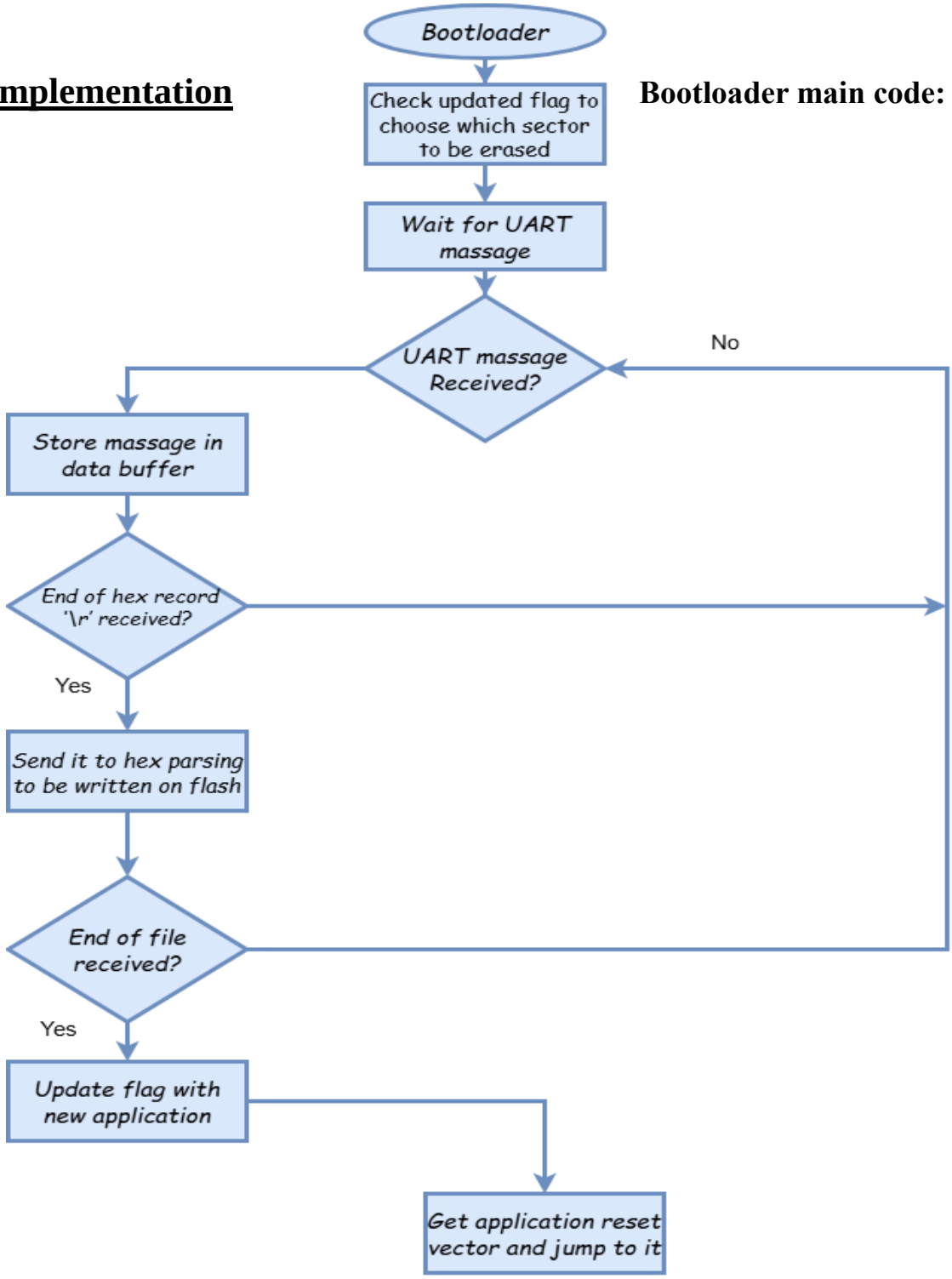


Figure 47

6.7. implementation

Bootloader main code:

```c
/* USER CODE BEGIN Header */
/******************************************************************************
 * @file           : main.c
 * @author         : Samy&Rashad@GP_FOTA_TEAM
 * @Data           : 14 Feb 2024
 * @brief          : Bootloader_V3
 ******************************************************************************/
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "Libraries/STD_TYPES.h"
#include "Libraries/BIT_MATH.h"

#include "MCAL/MRCC/MRCC_interface.h"
#include "MCAL/MGPIO/MGPIO_interface.h"
#include "MCAL/MSTK/MSYSTICK_Interface.h"
#include "MCAL/MNVIC/MNVIC_Interface.h"
#include "MCAL/MEXTI/MEXTI_interface.h"
#include "MCAL/MUSART/MUSART_Interface.h"
#include "MCAL/MFMI/MFMI_interface.h"

#include "HAL/LED/LED_interface.h"

#include "APP/HexParser/HexParser_interface.h"
#include "APP/BootLoader/BootLoader_interface.h"

/* USER CODE END Includes */
/*************************************************************************************************/
/*                                    MCAL Components                                            */
/* externs -------------------------------------------------------------*/
/* USER CODE BEGIN externs */
extern const LedX_t Led_1,Led_2,Led_3;

/* USER CODE END externs */
/* Global variables  ---------------------------------------------------*/
volatile u16 APP_1=APP_1_ID;
volatile u16 APP_2=APP_2_ID;

volatile u8 CurrantAPP=0;
volatile u8 UpdateAPP =0;

volatile u8 u8RecBuffer[100] = {0}   ;
volatile u8 u8RecCounter     =  0    ;
volatile u8 TimeOutFlag      =  0    ;
volatile u8 u8BLWriteReq     =  1    ;
u8 CS=No_ERROR;
u8 Error_counter=0; //same line receive error counter

/* Global variables  END */
/*functions deceleration ----------------------------------------------------*/

void Func_CallBack(void);
void Jumper       (void);
void Disables (void);
/* functions deceleration  END */
/* Private user code ---------------------------------------------------*/
/* USER CODE BEGIN 0 */
int main(void)
{
        /*Initialize System Clock to be 16MHz from HSI*/
        MRCC_voidInitSystemClk();
        /*Initialize  USART GPIO clock*/
        MRCC_voidEnablePeripheralClock(AHB1,GPIOA_PORT);
        /*Initialize the USART*/
        MRCC_voidEnablePeripheralClock(APB2,PERIPHERAL_EN_USART1);
        MGPIO_voidSetPinMode (GPIOA_PORT, PIN_9,  MODE_ALTF);
        MGPIO_voidSetPinMode (GPIOA_PORT, PIN_10, MODE_ALTF);
        MGPIO_voidSetPinAltFn(GPIOA_PORT, PIN_9,  ALTFN_7);
```

```
MGPIO_voidSetPinAltFn(GPIOA_PORT, PIN_10, ALTFN_7);
        /*USART initialize and enable*/
        MUSART_voidInit();
        MUSART_voidEnable(USART1);
        /*SysTick initialize*/
        MSTK_voidInit();
        /*Initialize the LEDS*/
        LED_Init(&Led_1);
        //LED_Init(&Led_3);

        /* begin ------------------------------------------------------*/
        LED_On(&Led_1);
        /*check on the currant App (in sector 4 and 5 )*/
        if ( UpDate_Flag == No_APP )
        {
                CurrantAPP=0;// determined which App active
                UpdateAPP =1;// determined which App will boot
        }
        else if ( UpDate_Flag == APP_1_MID )
        {
                CurrantAPP=1;// determined which App active
                UpdateAPP =2;// determined which App will boot
        }
        else if (UpDate_Flag == APP_2_MID )
        {
                CurrantAPP=2;// determined which App active
                UpdateAPP =1;// determined which App will boot
        }


        /*Defending on the used communication protocol*/
#if (BL_communication == USART_COMM)
        /*send ACK to the gateway that BOOTLOADER is active now*/
        MUSART_u8Send_Byte(USART1,(u8*)'B');
        /*7.5 sec wait then call the callback function*/
        MSTK_voidSetIntervalSingle((u32)15000000,Func_CallBack);
        /*USART receive statuses*/
        u8 LOC_u8RecStatus = NoReceive;

        while( TimeOutFlag == 0 )
        {
                LOC_u8RecStatus =MUSART_u8Receive_Byte(USART1,& u8RecBuffer[u8RecCounter]);        //from
the gateway
                //-------------------------------------------------------------------------------------
                //--------------------------------       Gateway receive --------------------------------
-
                if( LOC_u8RecStatus == Receive )
                {
                        /*Stop timer Application code in processing*/
                        MSTK_voidStopInterval();
                        /*Collecting data in the buffer*/
                        if( u8RecBuffer[u8RecCounter] == '\n' ){ //Indicate The end of the Record
                                if( u8BLWriteReq == 1 ){
                                        if(u8RecBuffer[0] == '1' || u8RecBuffer[0] == '2'){
                                                MUSART_u8Send_Byte(USART1,(u8*)'B');
                                                u8RecCounter = 0 ;
                                                continue;
                                        }
                                        /*Erase The Application Area*/
                                        if(UpdateAPP==1){
                                                MFMI_voidSectorErase(SECTOR_4);
                                        }
                                        else if(UpdateAPP==2){
                                                MFMI_voidSectorErase(SECTOR_5);
                                        }
                                        /*Set WriteReq Flag = 0 */
                                        u8BLWriteReq = 0 ;
                                }
                                /*Collecting check sum and send ACK to the gateway*/
                                CS=HexParser_CheckSumOfData( u8RecBuffer );
                                if(CS==No_ERROR){
                                        /*Parse The record and write it to flash*/
                                        HexParser_vParseRecord ( u8RecBuffer );
                                        /*send ok to the tool to send the next record*/
```

```
#if     (BL_IMT_DEBUGGING == ENABLE)
                                        MUSART_u8Send_Data(USART1,(u8 *)"ok");
#else
                                        MUSART_u8Send_Byte(USART1,(u8 *)'K');
#endif
                                        Error_counter=0;
                                }
                                else{
                                        // send ACK to the gateway to send it again if the error happened
3 times it fail and back to the app
                                        Error_counter++;
                                        if(1==Error_counter || 2==Error_counter){
                                                MUSART_u8Send_Byte(USART1,(u8 *)'N');
                                        }
                                        else if(3==Error_counter){
                                                MUSART_u8Send_Byte(USART1,(u8 *)'F');
                                                Error_counter=0;
                                                Func_CallBack();
                                        }
                                }
                                u8RecCounter = 0 ;
                        }
                        else{ u8RecCounter++ ; }
                        /*End of hex file */
                        if( u8RecBuffer[8] == '1' ){
                                //indicate to successful flashing
                                //LED_On(&Led_3);
                                MUSART_u8Send_Byte(USART1,(u8 *)'D');
                                /*update the flag and jump  */
                                CurrantAPP=UpdateAPP;
                                Func_CallBack();
                        }
                }
        }
#endif

#if (BL_communication == CAN_COMM)
        /*for CAN communication*/
#endif

        if(TimeOutFlag == 1){
                Jumper();
        }
}

/* USER CODE END 0 */
/* Functions -----------------------------------------------------------*/
void Func_CallBack(void){
        TimeOutFlag = 1 ;
        /*update the flag sector and be ready to jump */
        switch(CurrantAPP)
        {
        case 0:
                /*---NO APPs--- */
                TimeOutFlag = 0;
                MSTK_voidReSetInterval((u32)15000000);
                break;

        case 1:
                /*----APP1_S4--- */
                MFMI_voidSectorErase(SECTOR_3);
                MFMI_voidFlashWrite(APP_FLAG_ADD,&APP_1,1);
                MSTK_voidStopInterval();
                break;

        case 2:
                /*----APP2_S5--- */
                MFMI_voidSectorErase(SECTOR_3);
                MFMI_voidFlashWrite(APP_FLAG_ADD,&APP_2,1);
                MSTK_voidStopInterval();
                break;
        }
}
```

```c
void Jumper(void){
	switch(CurrantAPP)
	{
	case 0:
		/*---NO APPs--- */
		break;

	case 1:
		/*----APP1_S4--- */
		Disables();
		BL_voidJumpToAPP_1();
		break;

	case 2:
		/*----APP1_S5--- */
		Disables();
		BL_voidJumpToAPP_2();
		break;
	}
}

void Disables (void){
	/*turn off BootLoader LEDs */
	LED_Off(&Led_1);
	//LED_Off(&Led_3);
	/*disable peripherals */
	MUSART_voidDisable(USART1);
	MRCC_voidDisablePeripheralClock(AHB1,GPIOA_PORT);
	MRCC_voidDisablePeripheralClock(APB2,PERIPHERAL_EN_USART1);
}


/* GIDE for the characters send from the BOOTLOADER -------------------------*/
/*
	'B'	---	ACK to inform the gateway this node is in the BOOTLOADER now 'BOOTLOADER'
	'K'	---	ACK to inform the gateway this it received the record correct'OK'
	'N'	---	ACK to inform the gateway this it received the record wrong  'NO'
	'D'	---	ACK to inform the gateway this it received the last record   'DONE'
	'F'	---	ACK to inform the gateway this it received failed 3 times  'FAILED'
 */
```

Hex parser Code:

1- parsing record to hex

```
/*********************************************************************************************************/
static u8 HexParser_u8Ascii2Hex(u8 A_u8Ascii)
{
	/* local variable that holds the returned value */
	u8 L_u8HexValue = 0;

	/* Check if this Ascii value is equivalent to number from 0 to 9 */
	if(A_u8Ascii >= '0' && A_u8Ascii <= '9')
	{
		L_u8HexValue = A_u8Ascii - '0' ;
	}
	/* Check if this Ascii value is char A to F */
	else if(A_u8Ascii >= 'A' && A_u8Ascii <='F')
	{
		/* 55 to transform A to 10, B to 11 and etc.. */
		L_u8HexValue = A_u8Ascii - 55 ;
	}
	return L_u8HexValue;
}
```

```
/*********************************************************************************************************/
void HexParser_vParseRecord(u8*Copy_BufRecord)
{
	/* Check on record type */
	switch(Copy_BufRecord[8])
	{
	case '0' :

		/* Start parsing the record */
		HexParser_voidParseData(Copy_BufRecord);
		break;

	case '1':
		/* End of file - Do nothing */
		break;

	case '4':
		/*set high address part*/
		if(UpDate_Flag  == No_APP)
		{
			FLASH_BASE_ADDR=0x08010000;
		}

		else if ( UpDate_Flag  == APP_1_MID )
		{
			FLASH_BASE_ADDR=0x08020000;
			//count++;
			/*if(count==2)
			{
				FLASH_BASE_ADDR=0x08030000;
				count=0;
			}*/
		}

		else if (UpDate_Flag  == APP_2_MID )
		{
			FLASH_BASE_ADDR=0x08010000;
		}
		break;

	case '5':
		/* End of file - Do nothing */
		D_5_Flag_lastLine=1;
		break;


	default:
		/* Do nothing - defensive programming */
		break;
	}
}

/*********************************************************************************************************/
void HexParser_voidParseData(u8 * A_pu8Data)
{
	/* local variable to hold the character count high byte ,count low byte & count */
	u8 CC_high, CC_low, CC ;

	/* 4 digits for conversion */
	u8 digit0,digit1,digit2,digit3 ;

	/* To hold the address value */
	u32 address = 0;

	/*** Getting the character count ***/

	/* Get the high byte */
	CC_high = HexParser_u8Ascii2Hex(A_pu8Data[1]);
	/* Get the low byte */
	CC_low = HexParser_u8Ascii2Hex(A_pu8Data[2]);
	/* Get the character count */
	CC = (CC_high<<4)|CC_low ;

	/*** Extracting the address ***/
	digit0 = HexParser_u8Ascii2Hex(A_pu8Data[3]);
```

```
	digit1 = HexParser_u8Ascii2Hex(A_pu8Data[4]);
	digit2 = HexParser_u8Ascii2Hex(A_pu8Data[5]);
	digit3 = HexParser_u8Ascii2Hex(A_pu8Data[6]);


	/* Insert the low address into the least significant 4 bytes */
	address = address & 0xFFFF0000;
	address = (FLASH_BASE_ADDR) |
			(digit0 << 12)    |
			(digit1 << 8 )    |
			(digit2 << 4 )    |
			(digit3 << 0 );

	/*** Extracting the data ***/

	/* Writes 2 bytes per time */
	for(u8 i=0; i<(CC/2) ; i++)  //CC/2 step every two byte
	{
		/* Get the first byte */
		digit0 = HexParser_u8Ascii2Hex(A_pu8Data[(4*i)+9]);
		digit1 = HexParser_u8Ascii2Hex(A_pu8Data[(4*i)+10]);
		/* Get the second byte */
		digit2 = HexParser_u8Ascii2Hex(A_pu8Data[(4*i)+11]);
		digit3 = HexParser_u8Ascii2Hex(A_pu8Data[(4*i)+12]);

		if(digit0==0 && digit1==1 && digit2==0 && digit3==8){
			if ( UpDate_Flag == APP_1_MID) {
				digit1=2;
			}
		}
		DataBuffer[i] = (digit2 << 12) |
				(digit3 << 8 ) |
				(digit0 << 4 ) |
				(digit1 << 0 );
/*		else if(digit0==0 && digit1==2 && digit1==0 && digit1==8){

				}
		else if(digit0==0 && digit1==0 && digit1==0 && digit1==8){

				}
*/
	}
	// every byte is divided to two (upper and lower)
	// flash write in 16 bit mode
	// store 2 byte (4 digit) every time
	// each byte should be arranged to higher and lower

	/*
	 * Load the second byte then first byte using MSB first
	 * Example: FE BA should be loaded as BA FE
	 */

	/* Write the data in the flash memory */
	MFMI_voidFlashWrite(address, DataBuffer, CC/2);
}
```

Check-Sum Code:

```
/*********************************************************************************************************/
u8 HexParser_CheckSumOfData (u8 * Copy_u8BufData)
{
	/* local variable */
	u8 CharCount_H_Byte = 0, CharCount_L_Byte = 0 ,CharCount = 0;
	u8 No_ofRecord_Digits_without_CS_digits = 0;
	u8 Sum_of_Digits_without_CS_digits = 0;
	int CheckSum = 0;
	u8 Check_No_Error;

	/*** Getting the character count ***/

	/* Get the high byte */
	CharCount_H_Byte = HexParser_u8Ascii2Hex(Copy_u8BufData[1]);
	/* Get the low byte */
	CharCount_L_Byte = HexParser_u8Ascii2Hex(Copy_u8BufData[2]);
	/* Get the character count */
	CharCount = (CharCount_H_Byte << 4) | CharCount_L_Byte;

	/*
    CharCount_digits = 2 digit
     Address_digits  = 4 digit
     Type_digit      = 2 digit
     so we add 8 to sum of data digits
         */

	/* Calculate number of digits */
	No_ofRecord_Digits_without_CS_digits = (CharCount * 2) + 8;
	/*
	  Start from 1 to neglect the ':'
	  Increment by 1 bytes "2 digit"
	 */

	for (int i = 1 ;i < No_ofRecord_Digits_without_CS_digits; i+=2)
	{
		/* Accumulate the sum byte by byte */
		Sum_of_Digits_without_CS_digits += ( HexParser_u8Ascii2Hex(Copy_u8BufData[i]) << 4)
|HexParser_u8Ascii2Hex((Copy_u8BufData[i+1]));
	}

	/*
	  Checksum is the 2s-complement of the sum of the number of bytes, plus the address plus the data
	  Add up the number of bytes, the address and all the data and discard any carry to give 8-bit
total
	  Then invert each digit to give 1s-complement by XOR with 0xFF then add 1 to get the 2s-
complement
	 */

	Sum_of_Digits_without_CS_digits = (((Sum_of_Digits_without_CS_digits ^ 0xFF)) + 1);

	/*   Get the checkSum byte from Record */

	int CheckSum_HByte =
HexParser_u8Ascii2Hex(Copy_u8BufData[No_ofRecord_Digits_without_CS_digits+1]);
	int CheckSum_LByte =
HexParser_u8Ascii2Hex(Copy_u8BufData[No_ofRecord_Digits_without_CS_digits+2]);
	CheckSum = ( CheckSum_HByte << 4) | CheckSum_LByte ;

	/* Compare between Calculated checksum and checksum in record*/
	if ((CheckSum & 0xFF) == (Sum_of_Digits_without_CS_digits& 0xFF))
	{
		Check_No_Error = No_ERROR;
	}
	else
	{
		Check_No_Error = ERROR;
	}
	return Check_No_Error;

}
```

Chapter 7 : Gateway

7.1. Introduction:

The gateway is a microcontroller (stm32) that control the way between ESP32 and the two microcontrollers that have the applications (node 1 and 2), , it is very important especially with application having many ECUs like cars, gateway receive the code from ESP through UART protocol and forward it to the target ECU through a UART protocol too.

We can imagine it like a door and behind it two different doors and the gateway choose which door to enter depending on the information coming from the main door (ESP door) and also pass the coming information from node 1 or 2 to ESP door without looking at it.

The gateway is a central hub that reliably interconnects and processes data across these heterogeneous networks. It provides physical isolation and protocol translation to route data between functional domains that share data to enable new features. Gateways allow engineers to design more robust and functional vehicle networks.

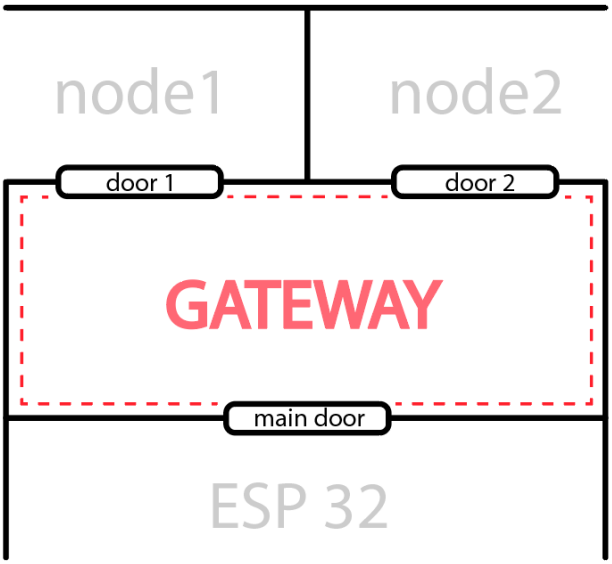


Figure 48

7.2. Gateway functions:

The gateway have four main functions and the three are important to complete the FOTA update for the target node by Determine which ECU this code is for to make sure the code is delivered to the correct ECU, as stated before the gateway has information about all the ECUs including ECUs IDs so it can determine which ECU the update is targeted to.

So we can break that to this four functions:

1- Connect with ESP32 and two STM32f401rc throw UART.
2- Choose the target node and deliver the hex file line by line.

3- Receive the feedback from the two node and deliver them to ESP32.
4- To indicate the nodes with the application that must execute.

7.3. Communication and Connections:

At the end of our communication build we found that we built a communication protocol similar to the reliable data transfer 1(rdt.1), by having acknowledgements coming from nodes to ESP throw gateway that happened throw UART protocol.

The different microcontrollers communicate with USART protocol and the code is ready to change to different communication way like CAN (will be discussed in the future features).

7.3.1. Communication (UART):

Universal Asynchronous Receiver/Transmitter protocol (UART), it's a dedicated hardware associated with serial communication. The hardware for the UART could be a dedicated IC or it could be an integrated circuit on the microcontroller. Unlike SPI and I2C, that are just communication protocols.

UART is one of the most commonly used Serial Communication techniques as it's very simple. Today, UART is being used in many applications like GSM, Modems, GPS Receivers, Wireless Communication Systems, Bluetooth Modules, and GPRS, RFID based applications etc. Most of the microcontrollers have dedicated UART hardware built in to their architecture. The main reason for integrating the UART hardware in to microcontrollers' architecture is that it is a serial communication and requires only two wires for communication, one for transmission and the other for reception.

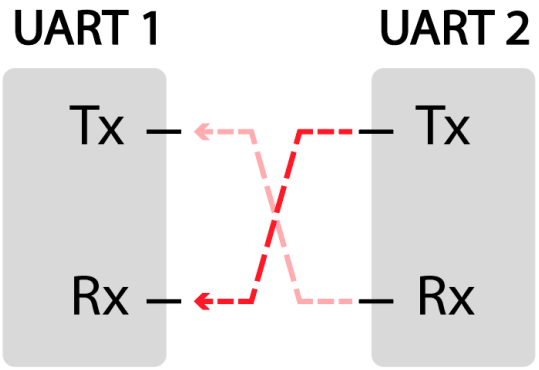


Figure 49

7.3.1.1. UART advantages:

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

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

3. No need for clock or any other timing signal (asynchronous).

4. Parity bit ensures basic error checking is integrated in to the data packet frame.4

5. Baud rate (speed) can be (2400, 4800, 9600, 14400, …., 115200)

7.3.1.2. UART frame:

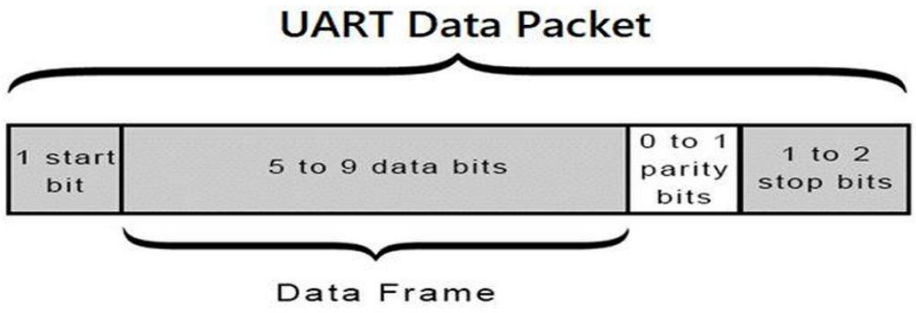


Figure 50

Start Bits the IDLE state of the UART transmission line is high voltage when it's not transmitting data. To start the transfer of data, the transmitting 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.

Data Bits the data bits contain the actual data that is being transferred. Its length can vary from 5 bits up to 8 bits long if a parity bit is used. If there is no parity bit, the data length can be 9 bits long. In most cases, the data is sent with the least significant bit first.

Parity Bits describes the evenness or oddness of a number. The parity bit is a way for the receiver of the UART to detect if any data has been changed during data transmission. Data could be changed due to mismatched baud rates, electromagnetic radiation, or long-distance data transfers. After the receiver reads the data frame, it counts the number of ones in it and checks if there is an even or odd number of ones. If the parity bit is a 0 this indicates even parity, which means that the total number of ones in the data should be an even number. If the parity bit is 1 which indicates odd parity means that the total number of ones in the data should be an odd number. If the parity bit matches the data, the UART knows that the transmission is free of errors and has been done successfully. But if the parity bit is the frame is 0, and the total number of ones in the data is odd; or the parity bit is a 1, and the total number of ones is even, the UART knows that an error occurred during transmission and bits in the data frame have changed.

Stop Bits To signal the end of the data packet, the transmitter signals the data transmission line from a low voltage to a high voltage to make it in the IDLE state again for at least two-bit durations.

method	asynchronous
parity	disable
Data length	8 bits
Stop bits	1 bit
Over sampling	16
Baud rate	9600

(Table show UART configuration used in the gateway)

7.3.2. Connections:

The gateway need two communicate with three microcontrollers, two stm32f401rc and one ESP32 and every communication need 2 lines (Tx & Rx)

- Gateway with Node 1: will be connected on port A pin 9 & 10
- Gateway with Node 2: will be connected on port A pin 2 & 3
- Gateway with ESP32 : will be connected on port A pin 11 & 12

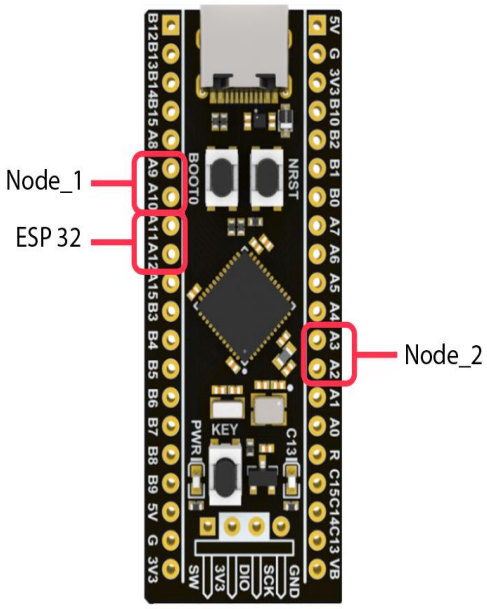


Figure 51

7.4. Gateway software:

7.4.1. Software drivers:

As the gateway doesn't have any sensors or any buttons to interact with, it is the communication bridge so it need the basic drivers and the communication ones to communicate properly.

The drivers:

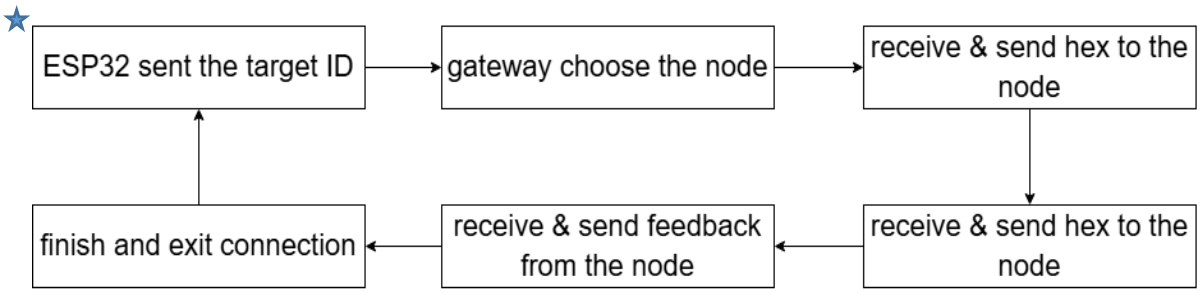
1. RCC : for the management of the clock of microcontroller.
2. GPIO : to interface with microcontroller pins.
3. STK : for the timer.
4. NVIC : mange the interrupts.
5. EXTI : mange the external interrupts.
6. UART : protocol to communicate with different microcontrollers.
7. LED : for hardware led.

7.4.2. Software sequence:

The gateway face two different missions, the first one to update the nodes (1 or 2) and the second one to inform the node with the application must be executed.

7.4.2.1. Update mode:

This mode is the one that for the FOTA to update one of the two nodes, as it was explained that the gateway is connecting between the ESP that have the hex file and the node that need this file, so gateway first receive the node ID that need to be updated from ESP, after that the gateway decide which UART will be activated (UART 1 or 2, UART 6 is for ESP32), after that gateway receive a acknowledge indicate that the node entered the bootloader and ready it be updated (this acknowledge pass to the ESP to start sending the hex file line by line), the gateway receive the first line and send it to the node, it will receive again an acknowledge from the node indicate if the line was correct or not (this ACK will be sent to ESP to retransmit the line if not ok or send the next one if ok), and so on to the last line (:00000001FF) ,after the last line the communication end and wait for new order.



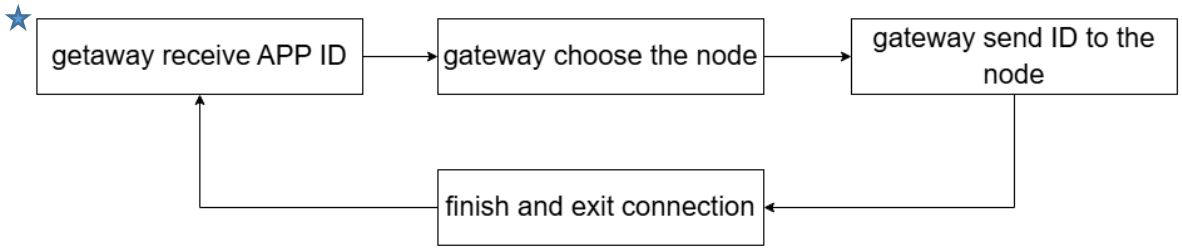
(Figure 4.5 Flowchart show the gateway program sequence to update node)

Acknowledgement guide from the code in the update mode:

'B' --- ACK to inform the ESP that the node BOOTLOADER respond
'K' --- ACK to inform the ESP that the node BOOTLOADER received the record correct
'N' --- ACK to inform the ESP that the node BOOTLOADER received the record wrong
'D' --- ACK to inform the ESP that the node BOOTLOADER received the last record
'F' --- ACK to inform the ESP that the node BOOTLOADER received failed 3 times

7.4.2.2. Application execution mode:

This mode is the one that for the applications (auto park, air conditioner), first the gateway receive the application ID, second gateway decide the node that have this application to inform it with the application id.



(Figure 4.6 Flowchart show the gateway program sequence to execute app)

ID guide from the code in the application mode:

'W' --- ACK to inform the NODE to start FORWARD application
'S' --- ACK to inform the NODE to start BACKWORD application
'A' --- ACK to inform the NODE to start AUTOPARK application
'C' --- ACK to inform the NODE to start CONDITIONER application

7.5. Implementation:

7.5.1. Gateway main code:

```
/* USER CODE BEGIN Header */
/**************************************************************************
 * @file           : main.c
 * @author         : Samy&Rashad&Huda@GP_FOTA_TEAM
 * @Data           : 17 Feb 2024 -      8 May 2024
 * @brief          : GETWAY_V0
 **************************************************************************/
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "Libraries/STD_TYPES.h"
#include "Libraries/BIT_MATH.h"

#include "MCAL/MRCC/MRCC_interface.h"
#include "MCAL/MGPIO/MGPIO_interface.h"
#include "MCAL/MSTK/MSYSTICK_Interface.h"
#include "MCAL/MNVIC/MNVIC_Interface.h"
#include "MCAL/MEXTI/MEXTI_interface.h"
#include "MCAL/MUSART/MUSART_Interface.h"

#include "HAL/LED/LED_interface.h"

/* USER CODE END Includes */
/**************************************************************************
******************/
/*                                      MCAL Components
*/
#define No_ERROR     1
#define ERROR        0

#define ENABLE              1
#define DISABLE             0
```

```c
#define USART_COMM   1
#define CAN_COMM     2
#define GW_communication  USART_COMM

#define GW_DEBUGGING            DISABLE
#define GW_IMT_DEBUGGING  DISABLE
/* externs ------------------------------------------------------------------*/
/* USER CODE BEGIN externs */
extern const LedX_t Led_3;

/* USER CODE END externs */
/* Global variables  ---------------------------------------------------------*/

volatile u8 u8RecBuffer_6[45]  = {NULL};
volatile u8 u8RecCounter_6     =  0    ;
volatile u8 u8FirstLineReq     =  1    ;
volatile u8 USART=0;

volatile u8 u8RecBuffer_1    =  0    ;
volatile u8 u8RecBuffer_2    =  0    ;
/* Global variables  END */
/*functions deceleration ----------------------------------------------------*/

void Reset_Flags(void);
/* functions deceleration  END */
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

int main(void)
{
	/*Initialize System Clock to be 16MHz from HSI*/
	MRCC_voidInitSystemClk();
	MSTK_voidInit();
	_delay_ms(1500);
	/*Initialize  USARTs GPIO clock*/
	MRCC_voidEnablePeripheralClock(AHB1,GPIOA_PORT);
	MRCC_voidEnablePeripheralClock(AHB1,GPIOC_PORT);
	/*USART 1 initializing*/                                        //for
Node_1
	MRCC_voidEnablePeripheralClock(APB2,PERIPHERAL_EN_USART1);
	MGPIO_voidSetPinMode (GPIOA_PORT, PIN_9,  MODE_ALTF);
	MGPIO_voidSetPinMode (GPIOA_PORT, PIN_10, MODE_ALTF);
	MGPIO_voidSetPinAltFn(GPIOA_PORT, PIN_9,  ALTFN_7);
	MGPIO_voidSetPinAltFn(GPIOA_PORT, PIN_10, ALTFN_7);
	/*USART 2 initializing*/                                        //for
Node_2
	MRCC_voidEnablePeripheralClock(APB1,PERIPHERAL_EN_UART2);
	MGPIO_voidSetPinMode (GPIOA_PORT, PIN_2,  MODE_ALTF);
	MGPIO_voidSetPinMode (GPIOA_PORT, PIN_3,  MODE_ALTF);
	MGPIO_voidSetPinAltFn(GPIOA_PORT, PIN_2,  ALTFN_7);
	MGPIO_voidSetPinAltFn(GPIOA_PORT, PIN_3,  ALTFN_7);
	/*USART 6 initializing*/                                        //for
ESP32
	MRCC_voidEnablePeripheralClock(APB2,PERIPHERAL_EN_USART6);
	MGPIO_voidSetPinMode (GPIOA_PORT, PIN_11,  MODE_ALTF);
	MGPIO_voidSetPinMode (GPIOA_PORT, PIN_12,  MODE_ALTF);
	MGPIO_voidSetPinAltFn(GPIOA_PORT, PIN_11,  ALTFN_8);
	MGPIO_voidSetPinAltFn(GPIOA_PORT, PIN_12,  ALTFN_8);
```

```
/*USARTs initialize and enable*/
	MUSART_voidInit();
	MUSART_voidEnable(USART1);
	MUSART_voidEnable(USART2);
	MUSART_voidEnable(USART6);
	/*SysTick initialize*/
	MSTK_voidInit();
	/*Initialize the LEDS*/
	LED_Init(&Led_3);

	/* begin -----------------------------------------------------------*/
	LED_Off(&Led_3);
	/*Defending on the used communication protocol*/
#if (GW_communication == USART_COMM)

#if    (GW_DEBUGGING == ENABLE)
	MUSART_u8Send_Data(USART6,(u8*)"---GETEWAY_READY---");            //---
	MUSART_u8Send_Data(USART6,MUSART_NewLine);                          //---
#endif
	/*USARTs receive statuses*/
	u8 LOC_u8RecStatus_1 = NoReceive;      //from Node_1
	u8 LOC_u8RecStatus_2 = NoReceive;      //from Node_2
	u8 LOC_u8RecStatus_6 = NoReceive;      //from ESP32

	while(1)
	{
		LOC_u8RecStatus_1 =MUSART_u8Receive_Byte(USART1,& u8RecBuffer_1);
                        //from Node_1
		LOC_u8RecStatus_2 =MUSART_u8Receive_Byte(USART2,& u8RecBuffer_2);
                       //from Node_2
		LOC_u8RecStatus_6 =MUSART_u8Receive_Byte(USART6,&
u8RecBuffer_6[u8RecCounter_6]);      //from ESP32
		//-------------------------------------------------------------------------
-----------------
		//------------------------------------ ESP32 -------------------------
---------------
		if( LOC_u8RecStatus_6 == Receive )
		{
			/*Collecting data in the buffer*/
			if( u8RecBuffer_6[u8RecCounter_6] == '\n' ){
				/*indicate the target node and send the restart order to
the app*/
				if( u8FirstLineReq == 1 ){

					/*--------UPDATE  NODE 1 or 2--------*/
					if (u8RecBuffer_6[0]=='1') {
#if    (GW_DEBUGGING == ENABLE)
                                        MUSART_u8Send_Data(USART6,(u8 *)"Start USART
1");    //---
#endif

                                        USART=USART1;
                                        u8FirstLineReq   = 0 ;
                                }
                                else if (u8RecBuffer_6[0]=='2') {
#if    (GW_DEBUGGING == ENABLE)
                                        MUSART_u8Send_Data(USART6,(u8 *)"Start USART
2");    //---
#endif
```

```
                                    USART=USART2;
                                    u8FirstLineReq   = 0 ;
                              }

                              /*--------applications on NODE_1
      (FORWARD,BACKWORD,AUTOPARK)--------*/
                              else if(u8RecBuffer_6[0]=='W'){
                                    USART=USART1;
                                    u8FirstLineReq   = 0 ;
                              }
                              else if(u8RecBuffer_6[0]=='S'){
                                    USART=USART1;
                                    u8FirstLineReq   = 0 ;
                              }
                              else if(u8RecBuffer_6[0]=='A'){
                                    USART=USART1;
                                    u8FirstLineReq   = 0 ;
                              }

                              /*--------applications on NODE_2 (CONDITIONER)------
--*/
                              else if(u8RecBuffer_6[0]=='C'){
                                    USART=USART2;
                                    u8FirstLineReq   = 0 ;
                              }
                              else{ /*for future fetcher*/ }
                        }
                        u8RecBuffer_6[u8RecCounter_6+1]='\0';  // for not sending
anything from the older data in the buffer
                        /*send to the target*/
                        if(USART==USART1 || USART==USART2){
                              MUSART_u8Send_Data(USART,u8RecBuffer_6);
                              /*rest usart after ordering the app*/
                              if((u8FirstLineReq  == 0) && (u8RecBuffer_6[0]=='W'
|| u8RecBuffer_6[0]=='A' || u8RecBuffer_6[0]=='S' || u8RecBuffer_6[0]=='C')){
                                    u8FirstLineReq  = 1;
                                    USART=0;
                              }
                        }
                        Led_Toggle(&Led_3); //led to visualizes the send process
                        /*Set buffer counter to 0 */
                        u8RecCounter_6   = 0 ;
                  }
                  else{ u8RecCounter_6++ ; }
            }
            //-------------------------------------------------------------------
-----------------
            //------------------------------------ Node 1      -------------------
----------------------
            if( LOC_u8RecStatus_1 == Receive )
            {
                  /*Receive the ACK from the BOOTLOADER*/
                  if(u8RecBuffer_1=='B'){                //node1 in the BOOTLOADER
                        MUSART_u8Send_Byte(USART6,'B');
                  }
                  else if(u8RecBuffer_1=='K'){     //record correct
#if    (GW_IMT_DEBUGGING == ENABLE)
                        MUSART_u8Send_Data(USART6,(u8 *)"ok");
```

```
#else
                        MUSART_u8Send_Byte(USART6,'K');
#endif
                }
                else if (u8RecBuffer_1=='N' ) {  //error in the record
                        MUSART_u8Send_Byte(USART6,'N');
                }
                else if (u8RecBuffer_1=='D') {   //End of hex file
                        MUSART_u8Send_Byte(USART6,'D');
                        Reset_Flags();                          //restart & send
something to show on the tablet
                }
                else if(u8RecBuffer_1=='F'){
                        //Bootloader received wrong 3 times ACK for ESP to re
download the file or ask the server to fix the problem
                        MUSART_u8Send_Byte(USART6,'F');
                        Reset_Flags();
                }
                else{
#if    (GW_DEBUGGING == ENABLE)
                        MUSART_u8Send_Data(USART6,&u8RecBuffer_1);          //---
                        MUSART_u8Send_Data(USART6," --- ERRORR1\n");  //---
#endif
                }
        }
        //-------------------------------------------------------------------------
-----------------
        //------------------------------------ Node 2      -------------------
----------------------
        if( LOC_u8RecStatus_2 == Receive )
        {
                /*Receive the ACK from the BOOTLOADER*/
                if(u8RecBuffer_2=='B'){                 //node2 in the BOOTLOADER
                        MUSART_u8Send_Byte(USART6,'B');
                }
                else if(u8RecBuffer_2=='K'){     //record correct
#if    (GW_IMT_DEBUGGING == ENABLE)
                        MUSART_u8Send_Data(USART6,(u8 *)"ok");
#else
                        MUSART_u8Send_Byte(USART6,'K');
#endif
                }
                else if (u8RecBuffer_2=='N' ) {  //error in the record
                        MUSART_u8Send_Byte(USART6,'N');
                }
                else if (u8RecBuffer_2=='D') {   //End of hex file
                        MUSART_u8Send_Byte(USART6,'D');
                        Reset_Flags();                          //restart send
something to show on the LCD
                }
                else if(u8RecBuffer_2=='F'){
                        //Bootloader received wrong 3 times ACK for ESP to re
download the file or ask the server to fix the problem
                        MUSART_u8Send_Byte(USART6,'F');
                        Reset_Flags();
                }
                else{
#if    (GW_DEBUGGING == ENABLE)
                        MUSART_u8Send_Data(USART6,&u8RecBuffer_2);          //---
```

```
                        MUSART_u8Send_Data(USART6," --- ERRORR2\n");  //---
#endif
                    }
                }
        }
#endif

#if (GW_communication == CAN_COMM)
        /*for CAN communication*/
#endif
}


/* USER CODE END 0 */
/* Functions -----------------------------------------------------------*/

void Reset_Flags(void)
{
        u8FirstLineReq = 1;
        USART=0;
        LED_Off(&Led_3);
}


/* GIDE for the characters send from the GETWAY -------------------------*/
/*
        'B'    ---    ACK to inform the ESP that the node BOOTLOADER respond
                        'BOOTLOADER'
        'K'    ---    ACK to inform the ESP that the node BOOTLOADER received the
record correct'OK'
        'N'    ---    ACK to inform the ESP that the node BOOTLOADER received the
record wrong  'NO'
        'D'    ---    ACK to inform the ESP that the node BOOTLOADER received the last
record   'DONE'
        'F'    ---    ACK to inform the ESP that the node BOOTLOADER received failed 3
times    'FAILED'


        'W'    ---    ACK to inform the NODE to start FORWARD     application
'FORWARD'
        'S'    ---    ACK to inform the NODE to start BACKWORD    application
'BACKWORD'
        'A'    ---    ACK to inform the NODE to start AUTOPARK    application
'AUTOPARK'
        'C'    ---    ACK to inform the NODE to start CONDITIONER application
'CONDITIONER'
 */
```

Chapter 8: Air Conditioner Application :

8.1 Introduction:

The air conditioning (AC) system in a car is crucial for providing comfort to the driver and passengers by controlling the temperature and humidity of the air inside the vehicle. Here's a detailed overview of how a car's air conditioning system works and its main components:

8.1.1 Components of the Car Air Conditioning System:

1. **Compressor**:
   - Acts as the heart of the AC system.
   - Compresses and circulates the refrigerant gas, raising its pressure and temperature.
2. **Condenser**:
   - Located in front of the car's radiator.
   - Cools the high-pressure refrigerant gas, converting it into a high-pressure liquid.
3. **Receiver/Drier**:
   - Removes moisture and contaminants from the refrigerant.
   - Contains a desiccant to absorb moisture.
4. **Expansion Valve (or Orifice Tube)**:
   - Regulates the amount of refrigerant entering the evaporator.
   - Reduces the pressure of the liquid refrigerant, cooling it down rapidly.
5. **Evaporator**:
   - Located inside the car's cabin.
   - Absorbs heat from the air inside the cabin, causing the refrigerant to evaporate and cool the air.
6. **Blower Fan**:
   - Pushes air over the evaporator fins and into the car's interior, delivering cooled air.

8.1.2 How the Car Air Conditioning System Works

1. **Compression**:
   - The AC system starts when the driver turns on the AC. The compressor compresses the refrigerant gas (often R-134a or R-1234yf), increasing its pressure and temperature.
2. **Condensation**:
   - The high-pressure, high-temperature gas flows into the condenser, where it is cooled by the air flowing through the condenser coils (often aided by the car's cooling fan), turning the gas into a high-pressure liquid.
3. **Expansion**:
   - The high-pressure liquid refrigerant passes through the expansion valve or orifice tube, which reduces its pressure and allows it to expand. This process cools the refrigerant significantly.

4. **Evaporation**:
    - The low-pressure, cold refrigerant flows into the evaporator inside the car's cabin. As warm air from the cabin is blown over the evaporator fins by the blower fan, the refrigerant absorbs heat from the air and evaporates into a gas. This process cools the air, which is then circulated back into the cabin.
5. **Circulation**:
    - The blower fan continues to push the cooled air into the car's interior. The now-warm refrigerant gas returns to the compressor to repeat the cycle.

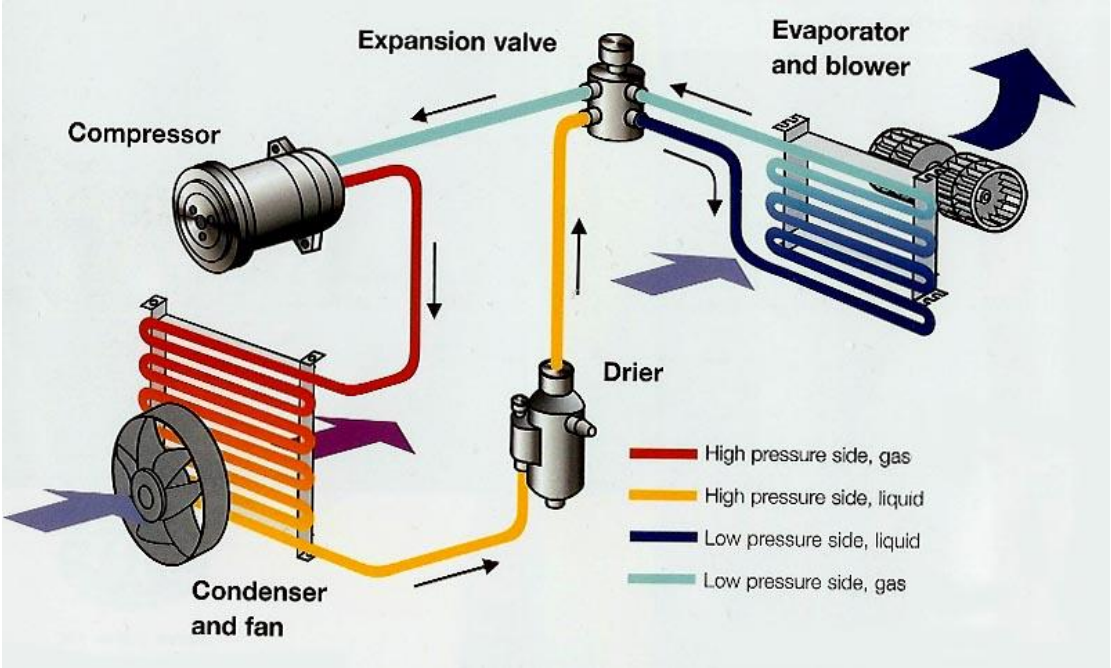


Figure 52

In our graduation project , we simulate it by small led and switch.The driver can push the switch to turn on the led which means the air conditioning in the car is turned on , this is the first one which called manual air conditioning

8.2 Manual air conditioning:

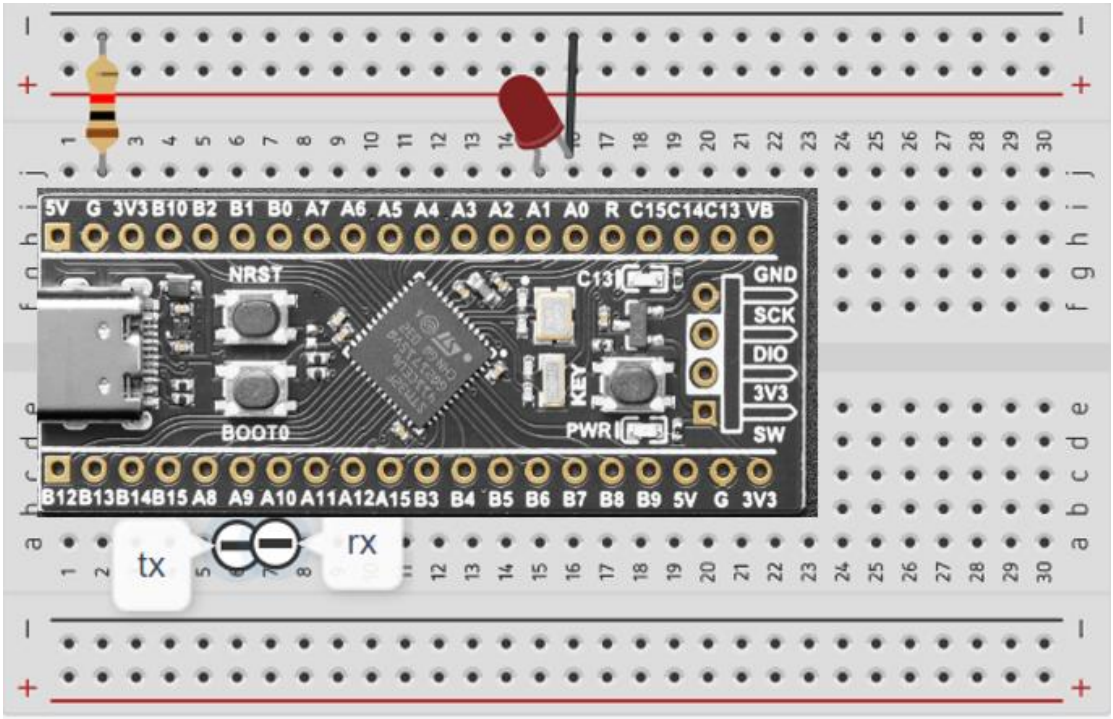


Figure 53

This figure show the connection of stm32 with led , tx and rx connected to rx and tx of the stm

8.2.1 How does manual mode work:

1- The driver push the button in tablet (Air Conditioner ON)
2- The Button send http request to esp32
3- Esp32 send number (5) as string to stm serially using UART communication protocol
4- 451stm turn the led on , this means the air conditioning is turned on (in the code saved in stm ,string 5 makes the led on)

The same process is done when the driver push the switch off on the tablet , in this case esp32 send number (6) as string to stm

8.3 UART

(Universal Asynchronous Receiver/Transmitter) is a hardware communication protocol used for asynchronous serial communication between devices. It is widely used in embedded systems for communication between microcontrollers, computers, and other peripherals. Here's an overview of the UART communication protocol:

8.3.1 How UART Works

1. **Asynchronous Communication**:
    - UART does not require a clock signal to synchronize the sender and receiver. Instead, both devices must agree on a common baud rate (bit rate) for data transmission.
2. **Data Transmission**:
    - Data is transmitted in packets called frames, which typically include a start bit, data bits, an optional parity bit, and stop bits.

8.3.2 UART Frame Structure

1. **Start Bit**:
    - The start bit signals the beginning of a data frame. It is a single low (0) bit that indicates the line is transitioning from idle (high state, 1) to active.
2. **Data Bits**:
    - These bits represent the actual data being transmitted. A UART frame can contain 5 to 9 data bits (usually 8 bits).
3. **Parity Bit (Optional)**:
    - This bit is used for error checking. It can be even parity (the number of 1s in the data bits plus the parity bit is even) or odd parity (the number of 1s is odd). It's optional and may not be used in all UART configurations.
4. **Stop Bits**:
    - These bits signal the end of the data frame. They can be 1, 1.5, or 2 bits long. The line returns to the idle state (high) during stop bits.

8.3.3 Typical UART Frame Example (8N1 Configuration)

- **8 Data Bits**
- **No Parity Bit**
- **1 Stop Bit**

In this configuration, a data frame would look like this:

```sql
| Start Bit | Data Bit 0 | Data Bit 1 | ... | Data Bit 7 | Stop Bit |
|     0     |      1     |      0     | ... |      1     |     1    |
```

Figure 54

8.3.4 Communication Flow

1. **Baud Rate**:
   - Both the transmitting and receiving devices must be set to the same baud rate. Common baud rates include 9600, 115200, etc.
2. **Transmission**:
   - The transmitter sends data serially one bit at a time. The receiver reads these bits at the agreed baud rate.
3. **Error Checking**:
   - If a parity bit is used, the receiver checks it to ensure data integrity. If the parity bit does not match the expected value, an error is detected.
4. **Buffering**:
   - Both transmitter and receiver often use buffers to temporarily store data. This helps manage the flow of data, especially if one device is faster than the other.

8.3.5 Advantages of UART

1. **Simplicity**:
   - UART is straightforward to implement and requires minimal external components.
2. **No Clock Signal**:
   - Asynchronous operation means there is no need for a shared clock signal, simplifying the wiring.
3. **Flexible**:
   - UART can be used for full-duplex communication (simultaneous transmission and reception).

8.3.6 Disadvantages of UART

1. **Limited Distance**:
   - UART is not suitable for long-distance communication without additional drivers or converters (like RS-232, RS-485).
2. **Limited Speed**:
   - UART is slower compared to other communication protocols like SPI or I2C, especially at higher baud rates where signal integrity can be an issue.

3. **No Multi-Master**:
    - UART supports only point-to-point communication, not multi-master configurations.

8.3.7 Common Applications

- **Microcontroller Communication**:
    - Communication between microcontrollers and sensors, modules, or other microcontrollers.
- **Embedded Systems**:
    - Used in debugging and programming microcontrollers.
- **Serial Devices**:
    - Connecting serial devices like GPS modules, Bluetooth modules, and other peripherals to a computer or microcontroller.

8.4 Air conditioning with timer:

In this type , the air conditioning turns on and off with timer

8.4.1 Timer Feature Implementation

Implementing the timer feature involved several steps. Firstly, the firmware was modified to detect when the "Air Conditioning On" button was pressed and to subsequently display the dialog box with the timer options. The dialog box was designed to be simple and user-friendly, with clear labels for the "Half Hour" and "One Hour" options. Once a selection was made, the firmware set a timer corresponding to the chosen duration. This timer was implemented using a countdown mechanism, where the system's internal clock was used to decrement the timer value every second. The firmware continuously monitored the timer, ensuring that the air conditioning system (represented by the LED) remained on for the specified duration. Upon timer expiration, the firmware automatically turned off the air conditioning system, thereby completing the timer-controlled operation. This process required careful coordination between the user interface and the underlying control logic to ensure seamless functionality.

8.4.2 FOTA Update Mechanism

The introduction of the timer feature was facilitated through a FOTA update. This process involved creating a new version of the firmware with the timer functionality integrated. The updated firmware was then packaged and prepared for over-the-air delivery. Implementing a secure and reliable FOTA delivery mechanism was crucial to ensure that the firmware update could be transmitted and applied without any issues. The microcontroller was programmed to receive the update, verify its integrity, and apply the new firmware, all without user intervention. This capability highlights the significant advantages of FOTA technology, allowing for remote updates and enhancements to be made to embedded systems without the need for physical access. Extensive testing was conducted to validate the updated firmware, ensuring that the timer feature worked as intended and that the update process was smooth and error-free.

8.4.3 How does timer mode work:

1- Oem upload the new update of the automatic air conditioning on the server
2- The server save the file in the fire base
3- Esp32 download the file which has the new update
4- The user push the button of check for update and choose update
5- The button send http request to esp32
6- Esp32 communicate serially using UART communication protocol with stm to download the file successfully and save it in stm
7- After the operation of download finishes , the user push button air conditioner on and choose half hour or one hour
8- In case the user choose half hour , the air conditioning turn on for 30 minutes but in our code we make the led turn on for 30 seconds
9- In case the user choose one hour , the air conditioning turn on for 60 minutes but in our code we make the led turn on for 60 seconds
10- In case the user push button air conditioner off , the air conditioning turn off (the led turn off)

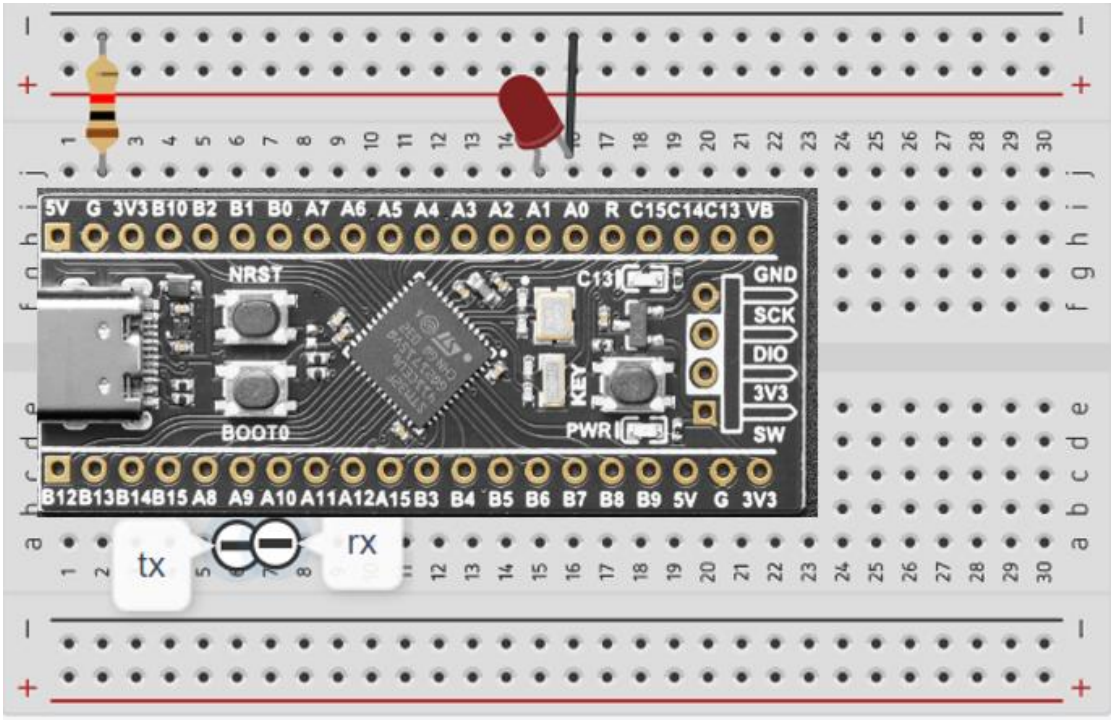


Figure 55

This figure shows connection between stm32 , led , tx , rx connected to rx, tx of stm same as mamual

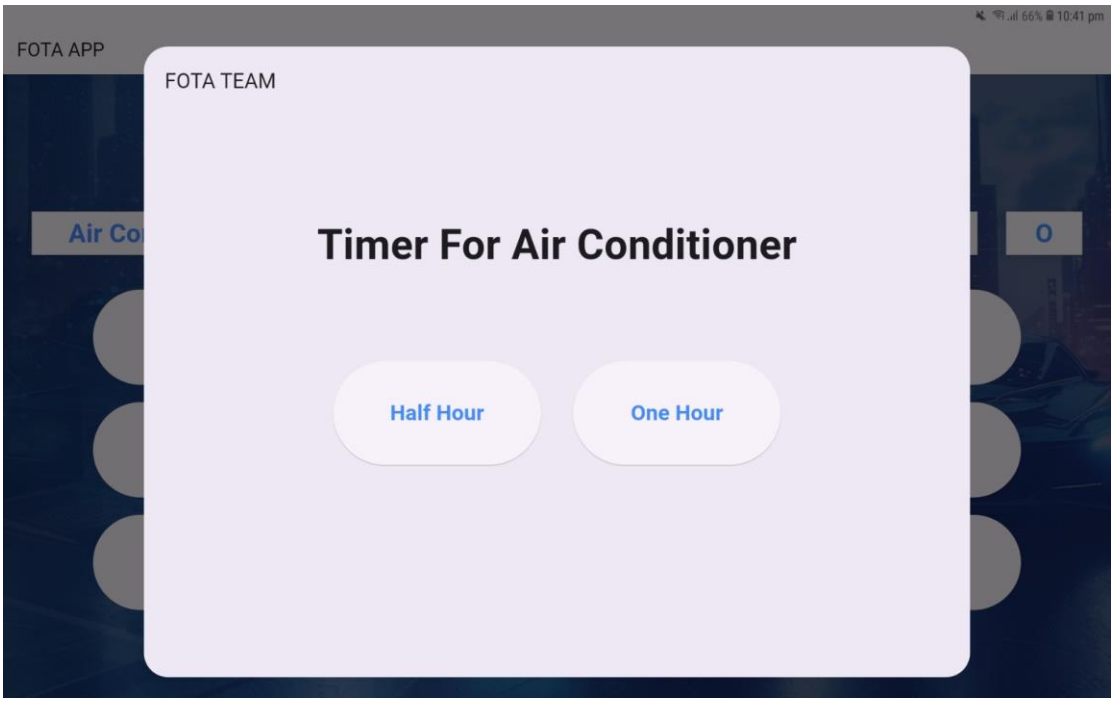


Figure 56

This figure shows the update of the application which make the user choose to turn on the air conditioning for half hour or one hour.

8.5 Advantages of timer air conditioning:

- **Energy Savings:** By setting the air conditioner to turn off after specified time , you can significantly reduce energy consumption .
- **Convenience:** You can program the air conditioner to turn on to avoid unnecessary operation .
- **Prolonged Lifespan:** Using the timer to avoid continuous operation can help reduce wear and tear on the unit, potentially extending its lifespan.
- **Cost Efficiency**: Reducing the operational hours of the air conditioner can result in lower maintenance and repair costs over time.

8.6 Future Work

Looking ahead, there are several avenues for further enhancing the application. Additional features, such as more complex scheduling options or the integration of temperature sensors for automated control, could be explored. Expanding the FOTA capabilities to support a wider range of devices and functionalities presents another exciting opportunity. This project serves as a foundational step towards leveraging FOTA technology in various applications, demonstrating its potential to revolutionize the way embedded systems are maintained and updated. Future work will build on this foundation, exploring new possibilities and pushing the boundaries of what can be achieved with remote firmware updates

8.7 Manual air conditioning Code :

```
#include "Libraries/STD_TYPES.h"
#include "Libraries/BIT_MATH.h"

#include "MCAL/MRCC/MRCC_interface.h"
#include "MCAL/MGPIO/MGPIO_interface.h"
#include "MCAL/MSTK/MSYSTICK_Interface.h"
#include "MCAL/MNVIC/MNVIC_Interface.h"
#include "MCAL/MEXTI/MEXTI_interface.h"
#include "MCAL/MUSART/MUSART_Interface.h"
#include "MCAL/MTIMER/MTIMER_interface.h"

#include "HAL/LED/LED_interface.h"
/* USER CODE END Includes */

/* externs ------------------------------------------------------------*/
/* USER CODE BEGIN externs */
extern const LedX_t                Led_0,Led1;//,Led_1,Led_2,Led_3,Leds_F,Leds_B;//embedded blue
,1,2,3 red

/* USER CODE END externs */

/* Global variables  ---------------------------------------------------*/
u8  R_data=0;                                  //USART date
u8  R_data_Flag=0;                  //USART date flag

/* Global variables  END */

/*functions deceleration ----------------------------------------------*/
```

```
void RECIVE_INT(void);
void WWDT_voidMakeSoftWareReset(void);
/* functions deceleration  END */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
int main(void)
{
	//u8 Parking_Available=0;
	/*System Clock is 16MHz from HSI*/
	MRCC_voidInitSystemClk();
	MSTK_voidInit();
	//--------LEDs --------//
	MRCC_voidEnablePeripheralClock(AHB1,GPIOC_PORT );
	LED_Init(&Led_0);
	LED_Off (&Led_0);

	MRCC_voidEnablePeripheralClock(AHB1,GPIOA_PORT );
	LED_Init(&Led_1);
	LED_Off (&Led_1);
	//--------USART--------//
	MRCC_voidEnablePeripheralClock(AHB1,GPIOA_PORT);
	MRCC_voidEnablePeripheralClock(APB2,PERIPHERAL_EN_USART1);
	MGPIO_voidSetPinMode (GPIOA_PORT, PIN_9,  MODE_ALTF);
	MGPIO_voidSetPinMode (GPIOA_PORT, PIN_10, MODE_ALTF);
	MGPIO_voidSetPinAltFn(GPIOA_PORT, PIN_9,  ALTFN_7);
	MGPIO_voidSetPinAltFn(GPIOA_PORT, PIN_10, ALTFN_7);
	MUSART_voidInit();
```

```
MUSART_voidEnable(USART1);
USART1_voidSetCallBack(RECIVE_INT);



//---------------------------------------*/
//---------------------------------------*/



//--------------------TEST AREA-------------------------*/
//-------------------------------------------------------*/



//--------Enable Interrupts-------------//
MNVIC_voidEnableInterrupt(MNVIC_USART1);              //UART
//---------------------------------------*/
//Indicate stop condition at start
while(1){
        //---------------//
        Led_Toggle(&Led_0);//Indicate the App is not stuck in something
        _delay_ms(100);
        //---------------//



        //-------------------------------------------------------------------------------
        //-----------------------------------     FORWARD APP  ----------------------------------


        if ('5'==R_data_Flag) {
```

```
                        LED_On(&Led_1);
                        //MUSART_u8Send_Data(USART1,"STOP ");//--
                        R_data_Flag='0';
                        //MNVIC_voidEnableInterrupt(MNVIC_USART1);
                }
                else if('6'==R_data_Flag){
                        LED_Off(&Led_1);
                        R_data_Flag='0';
                }
                else{
                        //Forbidden//
                }


        }




}


/* USER CODE END 0 */
/* Functions -----------------------------------------------------*/


void RECIVE_INT(void)
{
```

```
	MUSART_u8INT_Receive_Byte(USART1,&R_data);
	// software rest to the bootloader
	if('2'==R_data){
		LED_Off(&Led_0);
		MNVIC_voidDisableInterrupt(MNVIC_USART1);
		MUSART_voidDisable(USART1);
		//reset soft//
		WWDT_voidMakeSoftWareReset();
	}
	// Forward parking
	else if('5'==R_data){
		R_data_Flag='5';
	}
	// Backward parking
	else if('6'==R_data){
		R_data_Flag='6';
	}
	//
	else{
		//MUSART_u8Send_Data(USART1,"ERROR ");//-----
	}
	//MUSART_u8Send_Data(USART1,"recived int \n ");//-----
	MUSART_u8Send_Data(USART1,MUSART_NewLine);                    //-----//-----//---
--//-----
}



#define WWDT_CR          *((volatile u32 *)(0x40002C00))
void WWDT_voidMakeSoftWareReset(void)
```

```
{
	MRCC_voidEnablePeripheralClock(APB1,PERIPHERAL_EN_WWDG);
	WWDT_CR=0xBF;
	while(1);
}
```

8.7.1 Explanation of manual air conditioning code :

This code is for an embedded system application that involves communication via UART and controlling LEDs. It is written in C and uses various hardware abstraction layers and microcontroller abstraction layers (HAL and MCAL).

The function handles received UART data, performing actions based on the data:

- If data is '2', it triggers a software reset.
- If data is '5', it sets the flag to turn on Led_1.
- If data is '6', it sets the flag to turn off Led_1.

8.8 Air conditioning with timer Code :

```
/* USER CODE BEGIN Header */
/**********************************************************************************
 * @file         : main.c
 * @author       : Mohamed_Samy@GP_FOTA_TEAM
 * @brief        : Auto_Controll_APP
 **********************************************************************************/
/* USER CODE END Header */

/* Includes ------------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "Libraries/STD_TYPES.h"
#include "Libraries/BIT_MATH.h"

#include "MCAL/MRCC/MRCC_interface.h"
#include "MCAL/MGPIO/MGPIO_interface.h"
```

```
#include "MCAL/MSTK/MSYSTICK_Interface.h"
#include "MCAL/MNVIC/MNVIC_Interface.h"
#include "MCAL/MEXTI/MEXTI_interface.h"
#include "MCAL/MUSART/MUSART_Interface.h"
#include "MCAL/MTIMER/MTIMER_interface.h"

#include "HAL/LED/LED_interface.h"
/* USER CODE END Includes */

/* externs ------------------------------------------------------------*/
/* USER CODE BEGIN externs */
extern const LedX_t        Led_0,Led1;//,Led_1,Led_2,Led_3,Leds_F,Leds_B;//embedded blue  ,1,2,3 red

/* USER CODE END externs */

/* Global variables  ---------------------------------------------------*/
u8  R_data=0;              //USART date
u8  R_data_Flag=0;         //USART date flag

/* Global variables  END */

/*functions deceleration ----------------------------------------------*/
void RECIVE_INT(void);
void WWDT_voidMakeSoftWareReset(void);
/* functions deceleration  END */

/* Private user code ---------------------------------------------------*/
/* USER CODE BEGIN 0 */
```

```
int main(void)
{
   //u8 Parking_Available=0;
   /*System Clock is 16MHz from HSI*/
   MRCC_voidInitSystemClk();
   MSTK_voidInit();
   //--------LEDs --------//
   MRCC_voidEnablePeripheralClock(AHB1,GPIOC_PORT );
   LED_Init(&Led_0);
   LED_Off (&Led_0);

   MRCC_voidEnablePeripheralClock(AHB1,GPIOA_PORT );
   LED_Init(&Led_1);
   LED_Off (&Led_1);
   //--------USART--------//
   MRCC_voidEnablePeripheralClock(AHB1,GPIOA_PORT);
   MRCC_voidEnablePeripheralClock(APB2,PERIPHERAL_EN_USART1);
   MGPIO_voidSetPinMode (GPIOA_PORT, PIN_9,  MODE_ALTF);
   MGPIO_voidSetPinMode (GPIOA_PORT, PIN_10, MODE_ALTF);
   MGPIO_voidSetPinAltFn(GPIOA_PORT, PIN_9,  ALTFN_7);
   MGPIO_voidSetPinAltFn(GPIOA_PORT, PIN_10, ALTFN_7);
   MUSART_voidInit();
   MUSART_voidEnable(USART1);
   USART1_voidSetCallBack(RECIVE_INT);

   //---------------------------------------*/
   //---------------------------------------*/
```

```
//--------------------TEST AREA-------------------------*/
//------------------------------------------------------*/


//--------Enable Interrupts-------------//
MNVIC_voidEnableInterrupt(MNVIC_USART1);       //UART
//----------------------------------------*/
//Indicate stop condition at start
while(1){
   //---------------//
   Led_Toggle(&Led_0);//Indicate the App is not stuck in something
   _delay_ms(100);
   //---------------//

   //-------------------------------------------------------------------------------
   //-----------------------------------  FORWARD APP  ---------------------------------


   if ('8'==R_data_Flag) {
      LED_On(&Led_0);
      LED_On(&Led_1);
      _delay_s(30);
      LED_Off(&Led_1);
      R_data_Flag='0';
   }
   else if('4'==R_data_Flag){
      LED_On(&Led_0);
      LED_On(&Led_1);
      _delay_s(55);
      LED_Off(&Led_1);
```

```
        R_data_Flag='0';
      }
      else if('6'==R_data_Flag){
        LED_On(&Led_0);
        LED_Off(&Led_1);
        R_data_Flag='0';
      }
      else{
        //Forbidden//
      }   }}
/* USER CODE END 0 */
/* Functions -----------------------------------------------------------*/

void RECIVE_INT(void)
{
   MUSART_u8INT_Receive_Byte(USART1,&R_data);
   // software rest to the bootloader
   if('2'==R_data){
      LED_Off(&Led_0);
      MNVIC_voidDisableInterrupt(MNVIC_USART1);
      MUSART_voidDisable(USART1);
      //reset soft//
      WWDT_voidMakeSoftWareReset();
   }
   else if('8'==R_data){
      R_data_Flag='8';
   }
   else if('4'==R_data){
```

```
    R_data_Flag='4';
  }
  else if('6'==R_data){
    R_data_Flag='6';
  }
  //
  else{
    //MUSART_u8Send_Data(USART1,"ERROR ");//-----
  }
  //MUSART_u8Send_Data(USART1,"recived int \n ");//-----
  MUSART_u8Send_Data(USART1,MUSART_NewLine);          //-----//-----//-----//-----
}

#define WWDT_CR          *((volatile u32 *)(0x40002C00))
void WWDT_voidMakeSoftWareReset(void)
{
  MRCC_voidEnablePeripheralClock(APB1,PERIPHERAL_EN_WWDG);
  WWDT_CR=0xBF;
  while(1);
}
```

8.8.1 Explanation of air conditioning with timer code:

Same as pervious code but we add timer 30 sec or 60 sec respect to the choose of user, if we send "4 "== led turn on for 30 seconds

"8 "== led turn on for 60 seconds

Chapter 9 : Auto Parking Application

9.1 Auto-parking

9.1.1 Introduction

9.1.1.1 Background

Autonomous parking systems, known as auto-parking, are a major advancement in car technology. They help solve the challenges of parking in busy cities where spaces are tight and parking is stressful. These systems use sensors and smart algorithms to guide cars into parking spots with less need for human input. They make parking easier and safer, reducing the chances of accidents.

9.1.1.2 Objective

The main aim of this project is to create an auto-parking system showcasing modern autonomous vehicle technology. It will accurately park vehicles in different spaces like parallel, perpendicular, and angled spots. Key goals include:

- Designing a strong algorithm that uses data from various sensors to detect and maneuver into parking spots.
- Ensuring the system works well in different environments.
- Making an easy-to-use interface so drivers can start and watch the auto-parking process.

9.1.1.3 Scope

This project focuses on creating and testing an auto-parking system prototype. Here's what it includes:

I. System Design: Planning how hardware like ultrasonic sensors, cameras, and microcontrollers will work together, along with the software needed for auto-parking.
II. Implementation and Testing: Making a working model and testing it in controlled settings to see how well it parks in different situations.
III. Performance Evaluation: Checking how accurate, reliable, and fast the autoparking system is, and comparing it to other systems.
IV. User Interface Development: Creating an easy interface for drivers to use when they want to use the autoparking system.

9.1.2 Methodology

9.1.2.1 System Design

The autoparking system is designed to automate parking using hardware and software:

I. Sensor Arrays: Collect real-time data about the vehicle's surroundings.
II. Control Units: Process sensor data and control vehicle movements.
III. Software Algorithms: Analyze data to plan paths and make decisions for parking.

9.1.2.2 Hardware Components

The hardware components are carefully selected to ensure accurate detection and control:

1. **Ultrasonic sensors** are important for autoparking systems as they give real-time feedback on the vehicle's surroundings. They work by sending out high-frequency sound waves and measuring how long it takes for the waves to bounce back from nearby objects. This helps the system figure out how far away objects are and detect obstacles accurately. Placement and Configuration:

the configuration includes:

Front Bumper: 2 sensors

Rear Bumper: 2 sensors

Sides (near the front and rear wheels): 2 sensors on each side

This setup ensures that the vehicle can detect obstacles in all directions

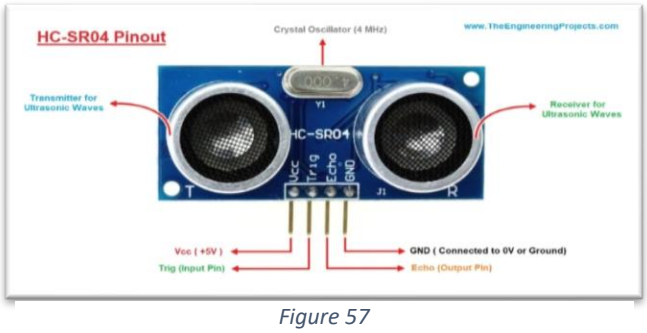


Figure 57

- Functionality:

Obstacle Detection: Ultrasonic sensors continuously emit sound waves and listen for their echoes. When the vehicle approaches an obstacle, the sensor detects the echo sooner, indicating a shorter distance.

➢ **Parking Space Measurement:** When searching for a parking space, the sensors measure the distance between parked vehicles or other obstacles to identify suitable gaps for parking.

- Technical Specifications:

➢ **Frequency:** Typically, ultrasonic sensors operate at frequencies between 40 kHz and 50 kHz.
➢ **Range:** The effective range of these sensors is usually between 20 cm to 5 meters.
➢ **Resolution:** The sensors can detect distance changes as small as a few millimeters, providing precise measurements.
➢ **Beam Angle:** The emission angle of the sound waves is typically narrow (around 15-30 degrees), which helps in focusing on specific areas and reducing interference from surrounding noise.

- Integration with Control Systems:

Ultrasonic sensors are connected to the vehicle's microcontroller or control unit. The integration process involves:

➢ **Data Collection:** The sensors transmit distance data to the microcontroller at regular intervals.
➢ **Signal Processing:** The microcontroller processes the raw data, filtering out noise and averaging multiple readings to improve accuracy.
➢ **Decision Making:** The processed data is used by the path planning and control algorithms to make real-time decisions about steering, acceleration, and braking during the parking maneuver.

➢ **Feedback Loop:** Continuous feedback from the sensors is used to adjust the vehicle's movements dynamically, ensuring safe and precise parking.

2. **Servo Motors:** Servo motors are used to control the angular motion of the front wheels in an autoparking system. By precisely adjusting the steering angle, servo motors enable the vehicle to maneuver into parking spaces with high accuracy. The servo motor's ability to receive and act upon precise control signals makes it an ideal choice for this application.

- **Functionality:**

Servo motors operate by receiving Pulse Width Modulation (PWM) signals from the microcontroller, which dictate the desired position or angle of the motor. The motor then moves to the corresponding position and holds it, providing precise control over the steering angle of the vehicle's front wheels.

- **Hardware Integration:**

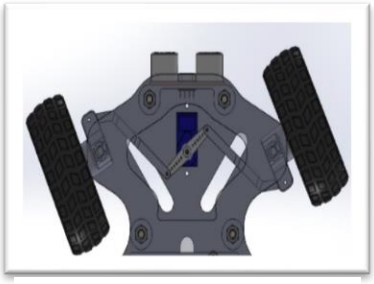


Figure 58

➢ Mounting the Servo Motor: The servo motor is attached to the steering column or linked to the steering mechanism. This setup lets the servo motor steer the front wheels.
➢ Mechanical Linkage: A strong connection is made between the servo motor and the front wheels. This ensures that any adjustments in the motor's angle are precisely translated into wheel movements.
➢ Electrical Connections: The servo motor connects to the vehicle's power source and the microcontroller. It usually needs three wires: one for power, one for ground, and one for receiving control signals.

- **Control System:**

➢ Microcontroller: The central control unit of the autoparking system, which processes sensor data and generates control signals for the servo motor.
➢ PWM Signal Generation: The microcontroller generates PWM signals based on the desired steering angle calculated by the path planning algorithm.
➢ Feedback Loop: Continuous feedback from the servo motor's position sensor ensures the motor achieves and maintains the desired angle.
➢ Sensors: Ultrasonic sensors and cameras provide real-time data about the vehicle's surroundings, feeding this information to the microcontroller.
➢ Path Planning Algorithm: Calculates the optimal path for the vehicle to follow, including the precise angles required for steering at each stage of the parking maneuver.

- **Technical Implementation:**

➢ Pulse Width Modulation (PWM): PWM signals are used to control the position of the servo motor. The width of the pulses determines the angle of the motor.
➢ Duty Cycle: The percentage of time the signal is high during each cycle determines the motor's position. For example, a 1.5 ms pulse might correspond to a neutral position (straight wheels), while 1.0 ms and 2.0 ms pulses correspond to maximum left and right steering angles, respectively.

3.DC motor for the Back Wheel

- **Definition:** A DC motor is an electric motor that runs on direct current (DC) electricity for move the car forward and backword. Plastic motor with 1:48 gearbox. The dual shaft turns up to about 600rpm. Powered by 3-12 volts, draws an average of 190 mA of current (max 250 mA).

Figure 59

- **Features:**
  - Power supply voltage: 3-12 V
  - Power consumption: 190 mA (max. 250 mA)
  - Gearbox: 1:48
  - Speed: 600 ± 10 rpm (at 12V)
  - Shaft diameter: 5.4 mm
  - The shaft is cut on both sides
  - The shaft has a threaded hole to fit M3 screws (not supplied)
  - The motor has a double-shaft

4. DC Motor Driver

Definition: A DC motor driver is an electronic device that controls the direction and speed of a DC motor. It acts as an interface between the microcontroller and the motor.

- **Pinout Configuration**

in Name	escription
N1 & IN2	lotor A input pins. Used to control the spinning direction of Motor A
N3 & IN4	lotor B input pins. Used to control the spinning direction of Motor B
NA	nables PWM signal for Motor A
NB	nables PWM signal for Motor B
UT1 & OUT2	utput pins of Motor A
UT3 & OUT4	utput pins of Motor B
2V	2V input from DC power Source
V	upplies power for the switching logic circuitry inside L298N IC
ND	round pin

Pinout of Motor Driver

Figure 60

- **Features & Specifications**
  - Motor Supply Voltage (Maximum): 46V
  - Motor Supply Current (Maximum): 2A
  - Logic Voltage: 5V
  - Driver Voltage: 5-35V
  - Driver Current:2A
  - Logical Current:0-36mA
  - Maximum Power (W): 25W

- Current Sense for each motor
- Heatsink for better performance

- **Communication Between DC Motor and Microcontroller**

  Control Signals: Typically, two types of signals are used: direction control signals and speed control signals (PWM).

  a. H-Bridge Circuit

- **Function:** An H-Bridge allows the motor to run forward and backward by reversing the polarity of the voltage applied to the motor.
- **Pins and Connections:**
  - Enable Pin (EN): Controls the motor's speed via PWM.
  - Input Pins (IN1 and IN2): Control the direction of the motor.

  b. Pulse Width Modulation (PWM)

- **Function:** PWM controls the speed of the motor by varying the duty cycle of the voltage signal.
- **Implementation:** The microcontroller generates a PWM signal to the motor driver's enable pin.

5. ARM STM32f401c Microcontroller

- **Features:**
- **High Processing Power:** Capable of handling real-time data processing and complex algorithms.
- **Rich Peripherals:** Includes ADCs (Analog-to-Digital Converters), PWM outputs, GPIOs (General Purpose Input/Output), I2C, SPI, UART interfaces, etc.
- **Low Power Consumption:** Efficient power management features suitable for automotive applications.
- **Scalability:** Available in various models to suit different performance and memory requirements.

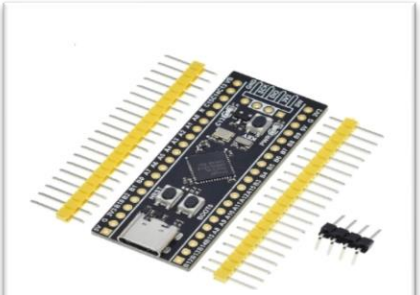


Figure 61

- Functionality in Autoparking Systems: The ARM STM microcontroller acts as the central processing unit, coordinating the activities of sensors, servo motors, and other components to execute the autoparking function.

- **System Architecture:**

1. **Sensor Integration:**
   - **Ultrasonic Sensors:** Connected via GPIO or ADC pins to measure distances to obstacles.
2. **Actuator Control:**
   - **Servo Motors:** Controlled using PWM signals generated by the microcontroller to adjust the steering angle.
   - **Throttle and Brake Actuators:** Controlled similarly to manage vehicle speed and stopping.
3. **Algorithm Execution:**

- **Sensor Fusion:** Combines data from ultrasonic sensors and cameras to create a comprehensive understanding of the environment.
- **Path Planning:** Executes algorithms to calculate the optimal path for parking.
- **Control Algorithms:** Uses PID controllers to ensure smooth and accurate actuator control.

9.1.3. Implementation

The autoparking system uses Firebase to get updates. These updates go to an ESP32 module that's connected to the internet and a gateway. The gateway understands the commands and sends them to a microcontroller. The microcontroller then performs parking maneuvers according to the commands it receives.

9.1.3.1 Step-by-Step Process

I. **Firebase Configuration:** Set up Firebase to receive parking commands from the user interface, storing them in a specific database location.

II. **ESP32 Configuration:** Configure the ESP32 to connect to Firebase and check for new commands periodically. When a command is received, the ESP32 sends it to the gateway.

2. Gateway Communication:

I. **Command Interpretation:** The gateway receives and interprets the command from the ESP32, deciding which action to take.

II. Routing Commands:

A. If the command is 'A', the gateway sends it to the microcontroller.

B. The microcontroller then performs one of three actions: self-parking, backward parking, or forward parking.

3.Microcontroller Parking Functions:

I. Self-Parking ('p'):

I. The vehicle initially moves forward, using ultrasonic sensors on the right side to measure the distance from potential parking spaces.

II. When the distance is greater than 12 cm, indicating a possible parking space, the vehicle stops and measures the space to ensure it is suitable for parking.

III. If the space is suitable, the vehicle maneuvers into the space by moving forward, adjusting the front wheels using a servo motor, and then reversing into the parking spot. The vehicle then straightens the wheels and moves forward slightly to finalize the parking.

II. Backward Parking ('B'):

I. The vehicle uses the back ultrasonic sensor to measure the distance behind it.

II. If the distance is greater than 12 cm, the motor moves the vehicle backward until the distance is less than 12 cm.

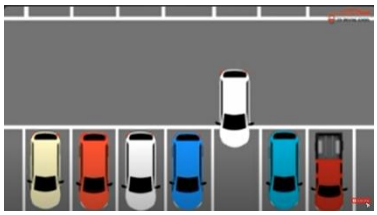
III. During this maneuver, the ultrasonic sensors on all sides ensure that the vehicle does not get too close to any obstacles.

III.Forward Parking ('f'):

I. The vehicle uses the front ultrasonic sensor to measure the distance in front of it.
II. If the distance is greater than 12 cm, the motor moves the vehicle forward until the distance is less than 12 cm.
III. The ultrasonic sensors on all sides are used to ensure the vehicle maintains a safe distance from any obstacles.

9.1.3.2 Steps with Pictures

For forward Parking

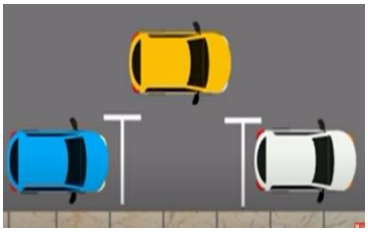


Sensing The Distance and go Forward

For Backward Parking

Sensing The Distance and go Backward

For Self_parking

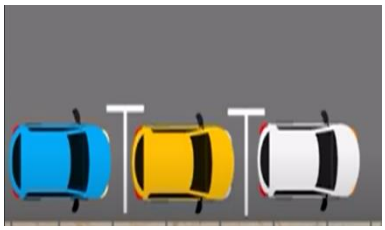


step1,sense the distance and measure it to be suitable for parking

step2,going forward to start parking

step3,Returnning the front Wheel

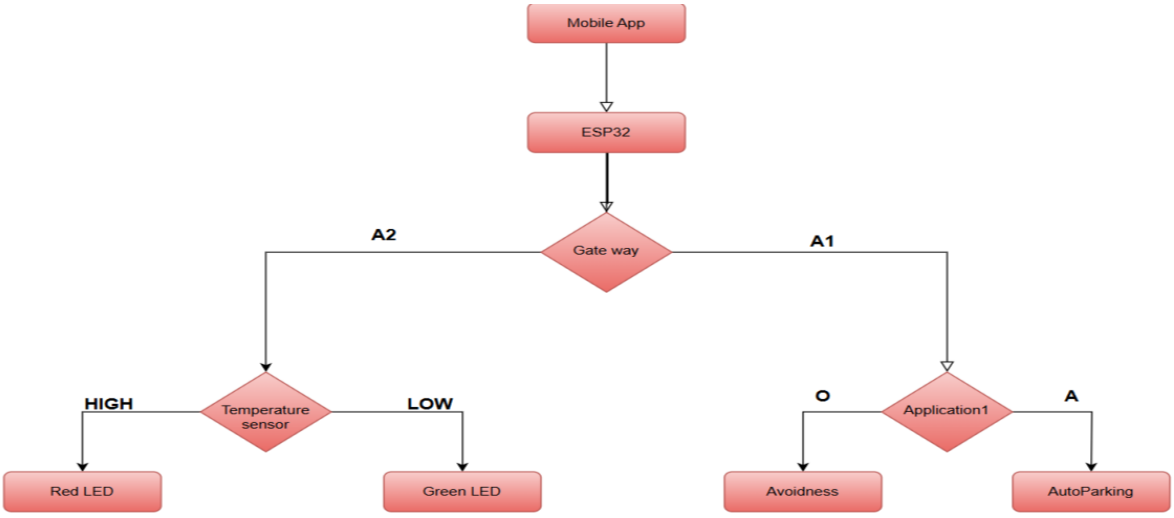
step4,going back with this angle



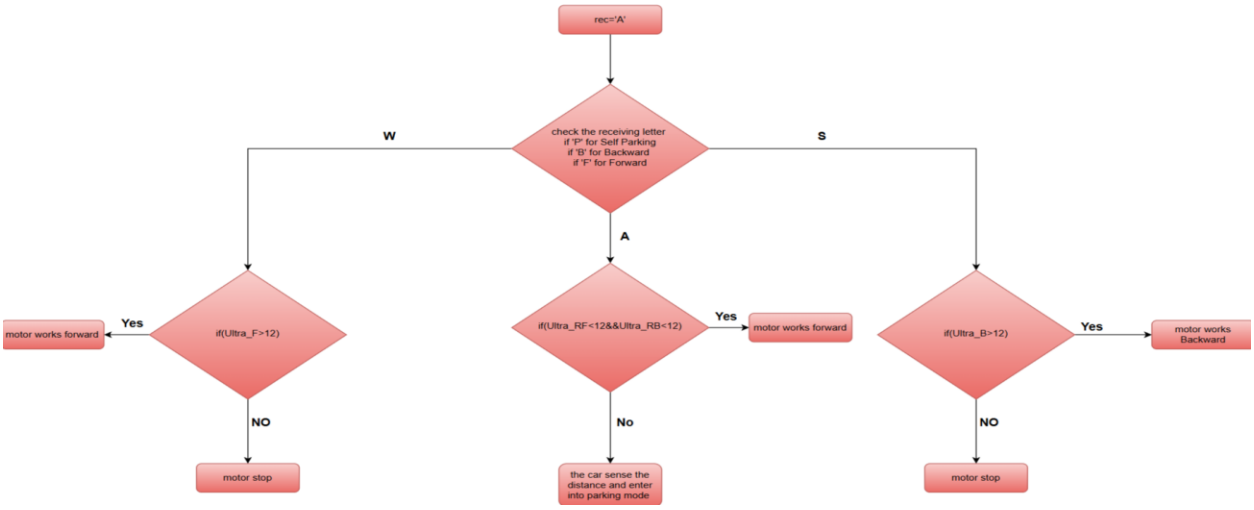
step5,correct parking with suitable distance

9.1.3.3 Flow Chart

- **For all system**



Flow Chart: For all system



9.1.3.5 Code

```
/* USER CODE BEGIN Header */
/**************************************************************************
 * @file           : main.c
 * @author         : Mohamed_Samy@GP_FOTA_TEAM
 * @brief          : Auto Parking APP
 **************************************************************************/
/* USER CODE END Header */

/* Includes ------------------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "Libraries/STD_TYPES.h"
#include "Libraries/BIT_MATH.h"

#include "MCAL/MRCC/MRCC_interface.h"
#include "MCAL/MGPIO/MGPIO_interface.h"
#include "MCAL/MSTK/MSYSTICK_Interface.h"
#include "MCAL/MNVIC/MNVIC_Interface.h"
#include "MCAL/MEXTI/MEXTI_interface.h"
#include "MCAL/MUSART/MUSART_Interface.h"
#include "MCAL/MTIMER/MTIMER_interface.h"

#include "HAL/LED/LED_interface.h"
#include "HAL/Ultrasonic/Ultrasonic_interface.h"
#include "HAL/MotorDriver/MotorDriver_interface.h"
#include "HAL/Servo_motor/Servo_motor_interface.h"
/* USER CODE END Includes */

/* externs ------------------------------------------------------------------*/
/* USER CODE BEGIN externs */
extern const LedX_t              Led_0;//,Led_1,Led_2,Led_3,Leds_F,Leds_B;//embedded blue  ,1,2,3 red
extern const UltraSX_t   UltraS_t_F_B;                    //Ultrasonics trigger 1&2..6
extern const MotorX_t            Motor_1,Motor_2;         //DC Motors  1,2
/* USER CODE END externs */

/* Global variables  ---------------------------------------------------------*/
u8  R_data=0;                           //USART date
u8  R_data_Flag=0;                //USART date flag
u8  currunt_turn=NO_TURN;   //servo motion flag
u8  currunt_motion=STOP; //dc motion flag
u8  SPEED=80;                           //dc motors speed 0-100

f64 Distance_F =0; //distance from ultraS_1 (front)                      -
f64 Distance_B =0; //distance from ultraS_2      (back)                   _
f64 Distance_Rf=0; //distance from ultraS_3      (right front)    .       |
f64 Distance_Rb=0; //distance from ultraS_4      (right back)     .       |
f64 Distance_Lf=0; //distance from ultraS_5      (left front)     |       .
f64 Distance_Lb=0; //distance from ultraS_6      (left back)              |       .
/* Global variables  END */

/*functions deceleration ----------------------------------------------------*/
void HAL_Rear_Drive(u8 DIRCTION);
void RECIVE_INT(void);
void WWDT_voidMakeSoftWareReset(void);
/* functions deceleration  END */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
int main(void)
{
        u8 Parking_Available=0;
        /*System Clock is 16MHz from HSI*/
        MRCC_voidInitSystemClk();
        MSTK_voidInit();
        //--------LEDs --------//
        MRCC_voidEnablePeripheralClock(AHB1,GPIOC_PORT );
```

```
LED_Init(&Led_0);
LED_Off (&Led_0);
/*      MRCC_voidEnablePeripheralClock(AHB1,GPIOA_PORT );
LED_Init(&Led_1);
LED_Off (&Led_1);
LED_Init(&Led_2);
LED_Off (&Led_2);
LED_Init(&Led_3);
LED_Off (&Led_3);*/
//--------USART--------//
MRCC_voidEnablePeripheralClock(AHB1,GPIOA_PORT);
MRCC_voidEnablePeripheralClock(APB2,PERIPHERAL_EN_USART1);
MGPIO_voidSetPinMode (GPIOA_PORT, PIN_9,  MODE_ALTF);
MGPIO_voidSetPinMode (GPIOA_PORT, PIN_10, MODE_ALTF);
MGPIO_voidSetPinAltFn(GPIOA_PORT, PIN_9,  ALTFN_7);
MGPIO_voidSetPinAltFn(GPIOA_PORT, PIN_10, ALTFN_7);
MUSART_voidInit();
MUSART_voidEnable(USART1);
USART1_voidSetCallBack(RECIVE_INT);
//--------ULTRASONIC echo & its Timer--------// F  B
// the front one (B3)
MRCC_voidEnablePeripheralClock(AHB1_BUS, PERIPHERAL_EN_GPIOB);
MRCC_voidEnablePeripheralClock(APB1_BUS, PERIPHERAL_EN_TIM2);
MTIMER_vStartTime(TIMER2);
MNVIC_voidEnableInterrupt(MNVIC_TIM2);
HULTRA_vInitialize(ULTRA_SONIC2, TIMER2, CH2);
// the back one          (A15)
MRCC_voidEnablePeripheralClock(AHB1_BUS, PERIPHERAL_EN_GPIOA);
MRCC_voidEnablePeripheralClock(APB1_BUS, PERIPHERAL_EN_TIM2);
MTIMER_vStartTime(TIMER2);
MNVIC_voidEnableInterrupt(MNVIC_TIM2);
HULTRA_vInitialize(ULTRA_SONIC1, TIMER2, CH1);
// the front right       (A3)
MRCC_voidEnablePeripheralClock(AHB1_BUS, PERIPHERAL_EN_GPIOA);
MRCC_voidEnablePeripheralClock(APB1_BUS, PERIPHERAL_EN_TIM2);
MTIMER_vStartTime(TIMER2);
MNVIC_voidEnableInterrupt(MNVIC_TIM2);
HULTRA_vInitialize(ULTRA_SONIC3, TIMER2, CH4);
// the front back        A2
MRCC_voidEnablePeripheralClock(AHB1_BUS, PERIPHERAL_EN_GPIOA);
MRCC_voidEnablePeripheralClock(APB1_BUS, PERIPHERAL_EN_TIM2);
MTIMER_vStartTime(TIMER2);
MNVIC_voidEnableInterrupt(MNVIC_TIM2);
HULTRA_vInitialize(ULTRA_SONIC4, TIMER2, CH3);
//--------Motor Driver--------//         PWM pin:B6      B7
HAL_MOTOR_Init(&Motor_1);
HAL_MOTOR_STOP(&Motor_1);
HAL_MOTOR_Init(&Motor_2);
HAL_MOTOR_STOP(&Motor_2);
//--------Servo motor --------//         A6
MRCC_voidEnablePeripheralClock(AHB1_BUS, PERIPHERAL_EN_GPIOA);
MRCC_voidEnablePeripheralClock(APB1_BUS, PERIPHERAL_EN_TIM3);
HSERVO_vServoInit(TIMER3,CH1);
HSERVO_vServoDeg(NO_TURN);
//--------------------------------------*/
//--------------------------------------*/
//--------------------TEST AREA-------------------------*/

//-----------------------------------------------------*/
//--------Enable Interrupts------------//
MNVIC_voidEnableInterrupt(MNVIC_USART1);         //UART
//--------------------------------------*/
LED_On(&Led_3);          //Indicate stop condition at start
while(1){
        //---------------//
        Led_Toggle(&Led_0);//Indicate the App is not stuck in something
        //---------------//
        //get ULTRA_SONIC_X distances_X
```

```
		HULTRA_vSendTrigger(&UltraS_t_F_B);                              //send trigeer for both
	1,2,3,4

		HULTRA_vGetDistance(ULTRA_SONIC2, &Distance_F);
		HULTRA_vGetDistance(ULTRA_SONIC1, &Distance_B);
		HULTRA_vGetDistance(ULTRA_SONIC3, &Distance_Rf);
		HULTRA_vGetDistance(ULTRA_SONIC4, &Distance_Rb);

		//-------------------------------------------------------------------------------------
		//-----------------------------------   FORWARD APP  ----------------------------------

		if ((('W'==R_data_Flag)||(currunt_motion==FORWARD)) && (Distance_F<=20)) {
			HAL_Rear_Drive(STOP);
			HSERVO_vServoDeg(NO_TURN);
			//MUSART_u8Send_Data(USART1,"STOP ");//--
			R_data_Flag='0';
			currunt_motion=STOP;
			currunt_turn=NO_TURN;
			//MNVIC_voidEnableInterrupt(MNVIC_USART1);
		}
		else if(('W'==R_data_Flag) && (Distance_F>20)){
			//MNVIC_voidDisableInterrupt(MNVIC_USART1);
			HAL_Rear_Drive(FORWARD);
			HSERVO_vServoDeg(NO_TURN);

			//MUSART_u8Send_Data(USART1,"FORWARD ");//--
			R_data_Flag='0';
			currunt_motion=FORWARD;
			currunt_turn=NO_TURN;
		}
		else{
			//Forbidden//
		}

		//-------------------------------------------------------------------------------------
		//-----------------------------------   BACKWARD APP  ---------------------------------
		//make it to stop after a delay and with the ultra sonic to be logiacly
		if ((('S'==R_data_Flag)||(currunt_motion==BACKWARD)) &&
((Distance_B<=20)||(Distance_Rf>=20))) {
			HAL_Rear_Drive(STOP);
			HSERVO_vServoDeg(NO_TURN);
			//MUSART_u8Send_Data(USART1,"STOP ");//--
			R_data_Flag='0';
			currunt_motion=STOP;
			currunt_turn=NO_TURN;
			//MNVIC_voidEnableInterrupt(MNVIC_USART1);
		}
		else if(('S'==R_data_Flag) && (Distance_B>20)){
			//MNVIC_voidDisableInterrupt(MNVIC_USART1);
			HAL_Rear_Drive(BACKWARD);
			HSERVO_vServoDeg(NO_TURN);
			//MUSART_u8Send_Data(USART1,"FORWARD ");//--
			R_data_Flag='0';
			currunt_motion=BACKWARD;
			currunt_turn=NO_TURN;
		}
		else{
			//Forbidden//
		}

		//-------------------------------------------------------------------------------------
		//-----------------------------------   AUTOPARK APP  ---------------------------------
		if('A'==R_data_Flag ){
			if((Distance_F <= 20)){
				HAL_Rear_Drive(STOP);

			}
			// first block
```

```
if((Distance_Rb <= 20) && (Distance_Rf <= 20) && (Parking_Available == 0))
{
	HAL_Rear_Drive(FORWARD);

}
//finding position
else if((Distance_Rf > 20) && (Distance_Rb > 20) && (Parking_Available == 0))
{
	/*	HAL_Rear_Drive(STOP);
	_delay_ms(250);*/
	HAL_Rear_Drive(FORWARD);
	Parking_Available = 1;

}

if((Distance_Rb <= 25) && (Distance_Rf <= 25) && (Parking_Available == 1))
{
	HAL_Rear_Drive(STOP);
	_delay_ms(50);
	HAL_Rear_Drive(BACKWARD);
	_delay_ms(200);
	HAL_Rear_Drive(STOP);
	_delay_ms(100);
	HSERVO_vServoDeg(110);
	_delay_ms(200);
	HAL_Rear_Drive(BACKWARD);
	_delay_ms(1500);
	HAL_Rear_Drive(STOP);
	_delay_ms(100);
	HSERVO_vServoDeg(LEFT);
	_delay_ms(100);
	HAL_Rear_Drive(BACKWARD);
		HULTRA_vSendTrigger(&UltraS_t_F_B);
	//send trigeer for both 1,2,3,4
	HULTRA_vGetDistance(ULTRA_SONIC4, &Distance_Rb);
	_delay_ms(20);
	HULTRA_vGetDistance(ULTRA_SONIC1, &Distance_B);

	while(Distance_Rb > 25 || Distance_B > 20)
	{
		HAL_Rear_Drive(BACKWARD);
		_delay_ms(10);
		HULTRA_vSendTrigger(&UltraS_t_F_B);
	//send trigeer for both 1,2,3,4

		HULTRA_vGetDistance(ULTRA_SONIC4, &Distance_Rb);
		_delay_ms(10);
		HULTRA_vGetDistance(ULTRA_SONIC1, &Distance_B);
	}
	HAL_Rear_Drive(STOP);
	_delay_ms(20);
	//HSERVO_vServoDeg(RIGHT);
	//_delay_ms(50);
	HAL_Rear_Drive(FORWARD);
	//_delay_ms(1500);
	HULTRA_vSendTrigger(&UltraS_t_F_B);
		//send trigeer for both 1,2,3,4
	HULTRA_vGetDistance(ULTRA_SONIC4, &Distance_Rb);
	_delay_ms(10);
	HULTRA_vGetDistance(ULTRA_SONIC3, &Distance_Rf);
	_delay_ms(10);
	HULTRA_vGetDistance(ULTRA_SONIC2, &Distance_F);

	while ((Distance_Rb > 20 || Distance_Rf > 20) || Distance_F >18)
	{
		_delay_ms(20);
		HSERVO_vServoDeg(RIGHT);
		_delay_ms(20);
```

```
HAL_Rear_Drive(FORWARD);
HULTRA_vSendTrigger(&UltraS_t_F_B);
//send trigeer for both 1,2,3,4
if((1.2 >= (Distance_Rb/Distance_Rf)) && (0.8<=(Distance_Rb/Distance_Rf)))
        {
                                                HAL_Rear_Drive(STOP);
                                                break;
                                        }
                                        HULTRA_vGetDistance(ULTRA_SONIC4, &Distance_Rb);
                                        _delay_ms(10);
                                        HULTRA_vGetDistance(ULTRA_SONIC3, &Distance_Rf);
                                        _delay_ms(10);
                                        HULTRA_vGetDistance(ULTRA_SONIC2, &Distance_F);
                                }

                                HAL_Rear_Drive(STOP);
                                _delay_ms(20);
                                HSERVO_vServoDeg(NO_TURN);

                                //_delay_ms(700);
                                Parking_Available =2;
                                }
                                }
                        }
                }
        }
}
/* USER CODE END 0 */
/* Functions -----------------------------------------------------------*/

void HAL_Rear_Drive(u8 DIRCTION)
{
        HAL_MOTOR_MOVE(&Motor_1, DIRCTION, SPEED);
        HAL_MOTOR_MOVE(&Motor_2, DIRCTION, SPEED);

}
void RECIVE_INT(void)
{
        MUSART_u8INT_Receive_Byte(USART1,&R_data);
        // software rest to the bootloader
        if('1'==R_data){
                LED_Off(&Led_0);
                MNVIC_voidDisableInterrupt(MNVIC_USART1);
                MUSART_voidDisable(USART1);
                //reset soft//
                WWDT_voidMakeSoftWareReset();
        }
        // Forward parking
        else if('W'==R_data){
                R_data_Flag='W';
        }
        // Backward parking
        else if('S'==R_data){
                R_data_Flag='S';
        }
        // Auto parking
        else if('A'==R_data){
                R_data_Flag='A';
        }
        //
        else{
                //MUSART_u8Send_Data(USART1,"ERROR ");//-----
        }

        MUSART_u8Send_Data(USART1,MUSART_NewLine);                              //-----//-----//-----//-
----
}
```

```c
#define WWDT_CR            *((volatile u32 *)(0x40002C00))
void WWDT_voidMakeSoftWareReset(void)
{
        MRCC_voidEnablePeripheralClock(APB1,PERIPHERAL_EN_WWDG);
        WWDT_CR=0xBF;
        while(1);
}


/*_____________________________________________
_____________________________________________
_____________________ ... _____________________
____________________| W |____________________
____________________...____________________
_____________________________________________
_____________ -         ...     - _____________
____________| A |__| S |__| D |_____________
_____________ -         ...     - _____________
_____________________________________________
_____________________________________________*/
```

9.2 Avoidance

9.2.1 Introduction

9.2.1.1 Background

Self-driving cars utilize ultrasonic sensors, microcontrollers, motors, and motor drivers to navigate autonomously. These components work together to sense surroundings, process data, and control movement, aiming to enhance safety and efficiency on the road.

9.2.1.2 Objectives

- **Develop an Obstacle Detection System:** Design and make a system using ultrasonic sensors to find obstacles around the car accurately.
- **Implement Real-Time Data Processing:** Make software that can quickly use sensor data to decide what to do right away.
- **Integrate Control Algorithms:** Create algorithms that can change how fast and which way the car moves based on the obstacles it finds.

9.2.1.3 Scope

- **Sensor Integration:** Using ultrasonic sensors to detect obstacles around the vehicle in real-time.
- **Microcontroller-Based Control:** Using a microcontroller (STM32F401C) to handle sensor data and control how the vehicle moves.
- **Software Development:** Creating software to process data from sensors, detect obstacles, plan paths, and control the vehicle.

9.2.2 Methodology

9.2.2.1 Design and Development

The design and development of the avoidance car involved careful selection and integration of sensors, software, and hardware components. This section details each aspect of the design process, focusing on the use of ultrasonic sensors and a microcontroller to achieve obstacle avoidance.

9.2.2.2 Sensors

To detect obstacles and navigate safely, the avoidance car utilizes multiple ultrasonic sensors. These sensors are chosen for their reliability, cost-effectiveness, and ability to measure distances accurately.

- **Ultrasonic Sensors:** Ultrasonic sensors use sound waves to detect objects. They emit a high-frequency sound wave and measure the time it takes for the echo to return after bouncing off an object. This time delay is used to calculate the distance to the object. The car is equipped with several ultrasonic sensors placed strategically around the vehicle:

I. **Front Bumper:** one sensors are placed on the front bumper to detect obstacles directly ahead.
II. **Rear Bumper:** one sensors are mounted on the rear bumper to assist with backward movement and detect obstacles behind the car.
III. **Side Panels:** two sensors are placed on every side panels to detect obstacles on either side, ensuring 360-degree coverage.

9.2.2.3 Software

The software component is critical for processing the data from the ultrasonic sensors and making real-time driving decisions. The software architecture includes the following key elements:

I. **Sensor Data Processing:** The software reads the distance measurements from the ultrasonic sensors continuously. It filters and processes this data to identify potential obstacles around the vehicle.
II. **Obstacle Detection Algorithms:** Simple threshold-based algorithms are used to determine if an object is within a critical distance that requires the vehicle to take action. If the distance to an object is less than a pre-defined threshold, the software recognizes it as an obstacle.
III. **Path Planning and Decision Making:** Basic path planning algorithms are implemented to decide the vehicle's movements. The software determines whether to stop, slow down, or change direction based on the location and proximity of detected obstacles. For instance, if an obstacle is detected in the front, the car may either stop or turn to avoid it.

9.2.2.3 Hardware

- **Microcontroller:** The core of the system is a microcontroller, such as (STM32F401c). The microcontroller is responsible for interfacing with the ultrasonic sensors, processing the sensor data, and controlling the vehicle’s actuators.

- **Motor Drivers:** Motor driver modules are used to control the speed and direction of the car's motors based on commands from the microcontroller. These drivers provide the necessary power and control signals to the motors.
- **Actuators:** The car's movement is controlled by DC motors or servo motors connected to the wheels. These actuators receive signals from the motor drivers to adjust speed and direction.

9.2.3 Implementation

To implement the system where Firebase sends update files to an ESP32, which then triggers actions on a microcontroller responsible for controlling a car's speed based on ultrasonic sensor data

9.2.3.1 Step-by-Step Process

1) Firebase Setup

2) ESP32 Configuration

- Program ESP32 to listen for Firebase updates and send a signal ('o') when an update is received.

3) Gateway Communication

- Connect ESP32 to a gateway device.
- Define a protocol for ESP32 to send commands ('o') to the gateway.

4) Microcontroller (STM32F401C) Setup

- Receive commands from the gateway on the STM32F401C.
- Initialize ultrasonic sensors (front, rear, and sides) connected to the microcontroller.

5) Distance Measurement and Speed Control

- Read distance data from ultrasonic sensors.
- Implement a speed control algorithm:

I. If distance < 20 cm, set lowest speed.
II. If distance > 20 cm, increase to highest speed.
III. Emergency stop if the front sensor detects a sudden stop ahead.

9.2.3.2 Steps with Pictures

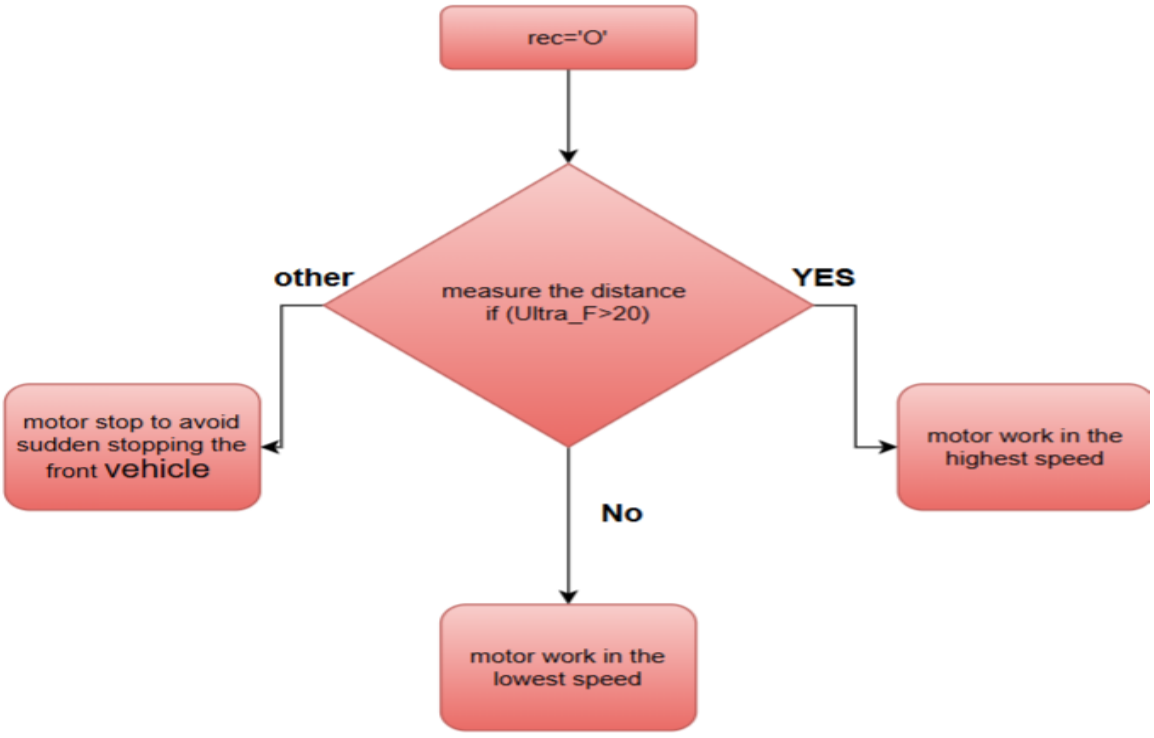
step1, for start motion

: step2, start detect the front distance

step3, when the distance is large the car move in high speed

step4, when the distance is small the car move in low speed

9.2.3.3 Flow Chart



9.2.3.5 Code

```
//-------------------------------------------------------------------------------------
                //-----------------------------------   Collision Avoidance APP  ----------------------
-----------
                if('O'==R_data_Flag ){

                        HULTRA_vGetDistance(ULTRA_SONIC2, &Distance_F);

                        if(Distance_F<=150 && Distance_F>=60)
                        {
                                HAL_Rear_Drive_Collision(FORWARD,95);
                        }
                        else if(Distance_F<60 && Distance_F>=35)
                        {
                                HAL_Rear_Drive_Collision(FORWARD,80);
                        }
                        else if(Distance_F<35 && Distance_F>=25)
                        {
                                HAL_Rear_Drive_Collision(FORWARD,70);
                        }
                        else if(Distance_F<25)
                        {
                                HAL_Rear_Drive_Collision(STOP,0);
                                //R_data_Flag='0';

                        }
                }
```

Chapter 10 : Security

10.1 Introduction

Computer and network security is complex due to its potential attacks, and counterintuitive procedures. It involves designing security mechanisms, deciding their placement, and ensuring the protection of secret information. Security is a battle of wits between attackers and designers, with attackers finding weaknesses and designers eliminating them. Users and system managers often perceive little benefit from security investment until a failure occurs. Regular monitoring is difficult, and security is often an afterthought rather than an integral part of the design process.

Cryptographic algorithms and protocols are essential in network and Internet security, focusing on deterring, preventing, detecting, and correcting security violations related to information transmission. These algorithms can be categorized into symmetric encryption, asymmetric encryption, data integrity algorithms, and authentication protocols. Examples of security violations include transmitting sensitive information, intercepting messages, invalidating accounts, and allowing customers to deny sending messages.

So, in our approach we are protecting our messages and data by Cryptography algorithms which are characterized by three dimensions: operations used to transform plaintext into ciphertext, the number of keys used (symmetric, single-key, secret-key, conventional), and the processing method of the plaintext, such as block or stream ciphers.

The encryption part for the application

- The code Encrypts/Decrypts files with AES algorithm (Cipher Feedback Mode).
- Uses secure key handling (hashed with random salt).
- Shows encryption/decryption progress.
- User-friendly GUI for file selection, key input, and progress monitoring.
- Handles errors and allows cancellation during encryption/decryption.

10.1.1 Security Algorithms

10.1.1.1 AES:

Advanced Encryption Standard (AES) is a symmetric encryption which uses the same key for encryption and decryption. It's known as block cipher, which means it takes a plaintext block (message) of a fixed number of bits and produces a ciphertext block of the same size. AES lays out 16 bytes (128 bits) of data in a 4×4 grid this is known as the "state" of the algorithm, and we will make updates to the state repeatedly until we get our encrypted block.

AES has three types according to the key size used (in bits). The three different key lengths that AES allows for: 128, 192, or 256 bits, the main thing that changes in AES is how you generate the key schedule from the key. We are working with key length equals to 128 bits. Encryption consists of 10 rounds of processing for 128-bit keys, 12 rounds for 192-bit keys, and 14 rounds for 256-bit keys. Except for the last round in each case, all other rounds are identical. Each round of processing includes one single-byte based substitution step, a row-wise permutation step, a

column-wise mixing step, and the addition of the round key. The order in which these four steps are executed is different for encryption and decryption. AES also has the notion of a word. A word consists of four bytes, that is 32 bits. Therefore, each column of the state is a word, as is each row. Each round of processing works on the input state and produces an output state.

AES uses is a substitution-permutation network - substitution-permutation network (SPN) is a fundamental structure used in block cipher algorithms, including AES - in a more general sense. Each round of processing in AES involves byte-level substitutions followed by word level permutation.

AES is an iterated block cipher in which plaintext is subject to multiple rounds of processing, with each round applying the same overall transformation function to the incoming block. AES process contains:

1. Key expansion
2. Substitution cipher
3. Shift rows
4. Mix columns

Key Expansion:

AES specifies a process that involves starting with the original key and determining the other bytes in the expanded key by combining values from previous bytes. The goal is to get an expanded set of keys where the bits of the expansion depend on many or all the bits from the original key.

Each round has its own round key that is derived from the original 128-bit encryption key in the manner described in this section. One of the four steps of each round, for both encryption and decryption, involves XORing of the round key with the state array

The implementation of the transformation function in each round involves a key that is specific to that round. This key is known as the round key. Round keys are derived from the user-supplied encryption key. In such ciphers, each round first applies a diffusion-achieving transformation operation which may be a combination of linear and nonlinear steps to the entire incoming block, which is then followed by the application of the round key to the entire block.

$$\begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix}$$

$$\Downarrow$$

$$\begin{bmatrix} w_0 & w_1 & w_2 & w_3 \end{bmatrix}$$

The 128-bit input block is arranged in the form of a state array, the algorithm first arranges the 16 bytes of the encryption key in the form of a 4 × 4 array of bytes.

The first four bytes of the encryption key constitute the word w_0, the next four bytes the word w_1, and so on. The remaining 40 words of the key schedule are used four words at a time in each of the 10 rounds.

How does the Key Expansion Algorithm expand four words into the 44 words w_0, w_1, w_2, w_3, w_4, w_5, ..., w_{43}?

As shown in figure 1, the key expansion takes place on a four-word to four-word basis, in the sense that each grouping of four words decides what the next grouping of four words will be.

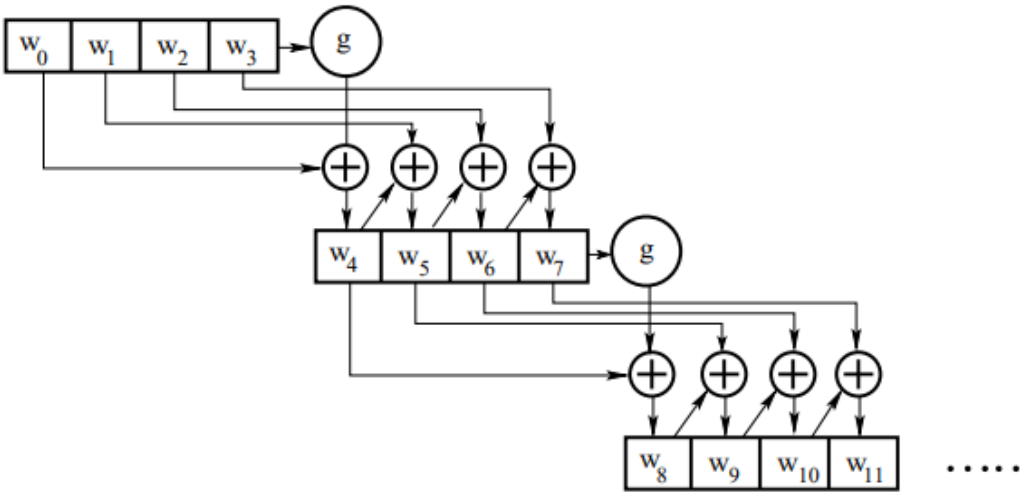


Figure 62

Substitution Cipher:

AES uses a substitution step called "SubBytes". Each byte in the state is substituted for a different byte. The encryption algorithm defines exactly what substitution to be performed for each byte with different substitution for every byte value.
This is a byte-by-byte substitution using a rule that stays the same in all encryption rounds. The byte-by-byte substitution rule is different for the decryption chain, but again it stays the same for all the rounds.

The presentation in the rest of this section is organized as follows:

- The modern way of explaining the byte substitution step that allows us to find the substitute byte for a given byte by simply looking up a pre-computed 256-element array of numbers.
- The traditional way of explaining the byte substitution step that involves using a 16 × 16 lookup table.
- My Perl and Python implementations of the byte substitution step. These implementations are based on the modern explanation of the step. Obviously, as you would expect, both explanations lead to the same final answer for byte substitution.

In the modern way of explaining the byte substitution step for the encryption chain, let x_{in} be a byte of the state array for which we seek a substitute byte x_{out}. We can write $x_{out} = f(x_{in})$. The function f() involves two nonlinear operations:

1. We first find the multiplicative inverse $x' = x_{in} - 1$ in $GF(2^8)$

2. then we scramble the bits of x ′ by XORing x ′ with four different circularly rotated versions of itself and with a special constant byte c = 0x63. The four circular rotations are through 4-, 5-, 6-, and 7-bit positions to the right. As you will see later in this section, this bit scrambling step can be expressed by the relation: $x_{out} = A \cdot x' + c$.

When using my BitVector module, the byte substitution step as explained above can be implemented with just a couple of calls to the module functions. The first operation of the step, which involves calculating the multiplicative inverse of a byte x in GF(2^8), can be carried out by invoking the function gf MI() on the BitVector representation of x. The second operation that requires XORing a byte with circularly shifted versions of itself is even more trivial.
The modern explanation of the byte substitution step as presented above applies equally well to the decryption chain, except for the fact that you first apply the bit scrambling operation to the byte and then you find its multiplicative inverse in GF(2^8).
The goal of the substitution step is to reduce the correlation between the input bits and the output bits at the byte level. The bit scrambling part of the substitution step ensures that the substitution cannot be described in the form of evaluating a simple mathematical function.

Shit Rows:

AES shifts the values of each row by a different amount. For Encryption, the Shift Rows transformation consists of

1. Not shifting the first row of the state array at all.
2. Circularly shifting the second row by one byte to the left.
3. Circularly shifting the third row by two bytes to the left.
4. Circularly shifting the last row by three bytes to the left.

This operation on the state array can be represented by:

$$\begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} ===> \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{bmatrix}$$

For decryption, the corresponding step shifts the rows in exactly the opposite fashion.

1. The first row is left unchanged.
2. The second row is circularly shifted to the right by one byte.
3. The third row is circularly shifted to the right by two bytes.
4. The last row is circularly shifted to the right by three bytes.

$$\begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2.0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3.0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} ===> \begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.3} & s_{1,0} & s_{1,1} & s_{1,2} \\ s_{2.2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3.1} & s_{3,2} & s_{3,3} & s_{3,0} \end{bmatrix}$$

Mix Columns:

This step replaces each byte of a column by a function of all the bytes in the same column., essentially diffusing information throughout the values. Mixing is by multiplication and addition but not the normal multiplication and addition, the multiplication and addition are limited, they work with a finite 256 (= 28) possible values starting from 0 to 255.

More precisely, for encryption, each byte in a column is replaced by two times that byte, plus three times the next byte, plus the byte that comes next, plus the byte that follows. The words "next" and "follow" refer to bytes in the same column, and their meaning is circular, in the sense that the byte that is next to the one in the last row is the one in the first row.

- For the bytes in the first row of the state array
$$s'_{0,j} = (0\mathbf{x}02 \times s_{0,j}) \otimes (0\mathbf{x}03 \times s_{1,j}) \otimes s_{2,j} \otimes s_{3,j}$$
- For the bytes in the second row of the state array
$$s'_{1,j} = s_{0,j} \otimes (0\mathbf{x}02 \times s_{1,j}) \otimes (0\mathbf{x}03 \times s_{2,j}) \otimes s_{3,j}$$
- For the bytes in the third row of the state array
$$s'_{2,j} = s_{0,j} \otimes s_{1,j} \otimes (0\mathbf{x}02 \times s_{2,j}) \otimes (0\mathbf{x}03 \times s_{3,j})$$
- For the bytes in the fourth row of the state array
$$s'_{3,j} = (0\mathbf{x}03 \times s_{0,j}) \otimes s_{1,j} \otimes s_{2,j} \otimes (0\mathbf{x}02 \times s_{3,j})$$

Finally, the column operations can be shown as

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2.0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3.0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0.0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1.0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2.0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3.0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

On the left-hand side, when a row of the leftmost matrix multiplies a column of the state array matrix, additions involved are meant to be XOR operations.

The corresponding transformation during decryption is given by

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \times \begin{bmatrix} s_{0.0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1.0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2.0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3.0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0.0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1.0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2.0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3.0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

THE OVERALL STRUCTURE OF AES:

The number of rounds shown in the figure is 10 for the case when the encryption key is 128 bit long. Before any round-based processing for encryption can begin, the input state array is XORed with the first four words of the key schedule. The same thing happens during decryption except that now we XOR the ciphertext state array with the last four words of the key schedule.

For encryption, each round consists of the following four steps:
1) Substitute bytes. 2) Shift rows.
3) Mix columns. 4) Add round key.

The last step consists of XORing the output of the previous three steps with four words from the key schedule.

For decryption, each round consists of the following four steps:
1) Inverse shift rows 2) Inverse substitute bytes.
3) Add round key 4) Inverse mix columns.

The third step consists of XORing the output of the previous two steps with four words from the key schedule.

Note the differences between the order in which substitution and shifting operations are carried out in a decryption round vis-a-vis the order in which similar operations are carried out in an encryption round. The last round for encryption does not involve the “Mix columns” step. The last round for decryption does not involve the “Inverse mix columns” step.

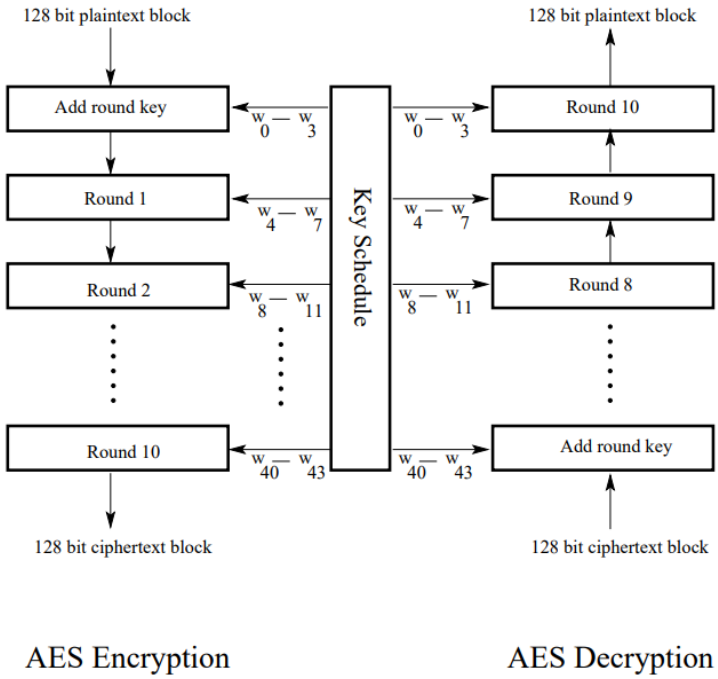


Figure 63

Efficiency: It is highly efficient for encrypting large amounts of data.

Strength: AES is a robust and widely adopted encryption standard known for its strength and performance. It has been scrutinized and approved by security experts globally.

Cipher Feedback Mode (CFB):

- A mode of operation for block ciphers like AES.
- Converts the block cipher into a stream cipher, which is useful for encrypting data streams of any size (unlike block ciphers which have a fixed block size).
- Works by feeding the previous ciphertext block back into the cipher along with the plaintext to generate the next ciphertext block. This creates a chaining effect where each ciphertext block depends on the previous ones.
- Offers a good balance between security and efficiency.
- While there are other modes for AES (like CBC - Cipher Block Chaining), CFB is a popular choice for its relative simplicity and error propagation property (errors in the ciphertext stream affect only subsequent blocks).

Why CFB with AES in this code?

- The code likely deals with files of varying sizes. CFB mode allows the AES block cipher to handle the file data as a stream, overcoming the limitation of fixed block sizes.
- CFB offers a good balance between security and performance, making it suitable for real-world encryption/decryption tasks.

10.1.1.2 Hashing Algorithms:

Hash algorithms - SHA-1, SHA-224, SHA-256, SHA-384, SHA-512, SHA-512/224 and SHA-512/256 - for computing a condensed representation of electronic data (message). When a message of any length less than 2^{64} bits (for SHA-1, SHA-224 and SHA-256) or less than 2^{128} bits (for SHA-384, SHA-512, SHA-512/224 and SHA-512/256) is input to a hash algorithm, the result is an output called a message digest. The message digests range in length from 160 to 512 bits, depending on the algorithm. Secure hash algorithms are typically used with other cryptographic algorithms, such as digital signature algorithms and keyed-hash message authentication codes, or in the generation of random numbers (bits).

The hash algorithms are computationally infeasible:

1) to find a message that corresponds to a given message digest.
2) to find two different messages that produce the same message digest. Any change to a message will, with a very high probability, result in a different message digest. This will result in a verification failure when the secure hash algorithm is used with a digital signature algorithm or a keyed-hash message authentication algorithm.

SHA-256 -which we are using in the code- may be used to hash a message, *M*, having a length of λ bits, where $0 \leq \lambda < 2^{64}$. The algorithm uses:

1) A message schedule of sixty-four 32-bit words.

2) Eight working variables of 32 bits each.
3) A hash value of eight 32-bit words. The result of SHA-256 is a 256-bit message digest.

The words of the message schedule are labeled W_0, W_1…, W_{63}. The eight working variables are labeled **a**, **b**, **c**, **d**, **e**, **f**, **g**, and **h**. The words of the hash value are labeled $H_0^{(i)}$, $H_1^{(i)}$ ……., $H_7^{(i)}$ which will hold the initial hash value, $H^{(0)}$, replaced by each successive intermediate hash value (after each message block is processed), $H^{(i)}$, and ending with the final hash value, $H^{(N)}$. SHA-256 also uses two temporary words, T_1 and T_2.

SHA-256 Preprocessing:

1. Set the initial hash value, H (0).
   For SHA-256, the initial hash value, H (0), shall consist of the following eight 32-bit words, in hex:

   $H_0^{(0)}$ = 6a09e667 $H_1^{(0)}$ = bb67ae85
   $H_2^{(0)}$ = 3c6ef372 $H_3^{(0)}$ = a54ff53a
   $H_4^{(0)}$ = 510e527f $H_5^{(0)}$ = 9b05688c
   $H_6^{(0)}$ = 1f83d9ab $H_7^{(0)}$ = 5be0cd19

   These words were obtained by taking the first thirty-two bits of the fractional parts of the square roots of the first eight prime numbers.

2. The message is padded and parsed.
   The purpose of this padding is to ensure that the padded message is a multiple of 512 or 1024 bits, depending on the algorithm. Padding can be inserted before hash computation begins on a message, or at any other time during the hash computation prior to processing the block(s) that will contain the padding.

SHA-256 Hash Computation:

The SHA-256 hash computation uses functions and constants previously defined in Sec. 4.1.2 and Sec. 4.2.2, respectively. Addition (+) is performed modulo 2^{32}. Each message block, $M^{(1)}$, $M^{(2)}$, …, $M^{(N)}$, is processed in order, using the following steps:

For i=1 to N:

1. Prepare the message schedule, $\{W_t\}$:

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

2. Initialize the eight working variables, a, b, c, d, e, f, g, and h, with the $(i-1)^{st}$ hash value:

$$a = H_0^{(i-1)} \qquad e = H_4^{(i-1)}$$
$$b = H_1^{(i-1)} \qquad f = H_5^{(i-1)}$$
$$c = H_2^{(i-1)} \qquad g = H_6^{(i-1)}$$
$$d = H_3^{(i-1)} \qquad h = H_7^{(i-1)}$$

3. For t=0 to 63:

$$T_1 = h + \sum_1^{\{256\}}(e) + Ch(e,f,g) + K_t^{\{256\}} + W_t$$
$$T_2 = \sum_0^{\{256\}}(a) + Maj(a,b,c)$$

$$h = g \qquad d = c$$
$$g = f \qquad c = b$$
$$f = e \qquad b = a$$
$$e = d + T_1 \qquad a = T_1 + T_2$$

4. Compute the i^{th} intermediate hash value $H^{(i)}$:

$$H_0^{(i)} = a + H_0^{(i-1)} \qquad H_4^{(i)} = e + H_4^{(i-1)}$$
$$H_1^{(i)} = b + H_1^{(i-1)} \qquad H_5^{(i)} = f + H_5^{(i-1)}$$
$$H_2^{(i)} = c + H_2^{(i-1)} \qquad H_6^{(i)} = g + H_6^{(i-1)}$$
$$H_3^{(i)} = d + H_3^{(i-1)} \qquad H_7^{(i)} = h + H_7^{(i-1)}$$

After repeating steps one through four a total of N times (i.e., after processing $M^{(N)}$), the resulting 256-bit message digest of the message, *M*, is

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}$$

10.1.2 Libraries and Classes

10.1.1.3 Libraries:

- os: Provides functions for interacting with the operating system.
- sys: Provides access to system parameters and functions.
- hashlib: Provides functions for various hashing algorithms.
- tkinter: Provides functions for creating graphical user interfaces (GUI) in Python.
- filedialog: Provides functions for opening and saving file dialogs.
- messagebox: Provides functions for displaying message boxes.
- Crypto.Cipher: Provides classes for various encryption ciphers (AES used here).
- secrets: Provides functions for generating cryptographically secure random numbers.

10.1.1.4 Classes:

EncryptionTool:

- This class handles encryption and decryption logic. It takes the file path, key, and salt as input and performs the following actions:
    - It takes the file path and user-provided key as input.
    - It generates a secure random salt for key derivation.
    - Hashes the key and salt using SHA256.
    - Creates a cipher object using AES in CFB mode.
    - Reads the input file in chunks and encrypts/decrypts it using the cipher object.
    - Writes the encrypted/decrypted data to the output file.
    - Provides a function to cancel the ongoing encryption/decryption process.
- Functions used in it:
    - def __init__(self, user_file, user_key)
    - def read_in_chunks(file_object, chunk_size=1024)
    - def hash_key_salt(self)
    - def encrypt(self)
    - def decrypt(self)
    - def abort(self)

MainWindow:

- This class creates the GUI elements and handles user interactions. It includes:
    - File selection button and entry field.
    - Key entry field (password field with asterisks).
    - Encrypt and Decrypt buttons.
    - Reset button.
    - Status label to display messages.
    - Menu bar with "Owners" and "Exit" options.
- Functions used in it:
    - def __init__(self, root)
    - def selectfile_callback(self)
    - def freeze_controls(self)
    - def unfreeze_controls(self)
    - def encrypt_callback(self)
    - def decrypt_callback(self)
    - def reset_callback(self)
    - def cancel_callback(self)
    - def show_help_callback(self)

10.1.3 Functions used in the code

10.1.1.5 "__init__" function from Encryption Tool Class:

It is a known function in python that initializes attributes.

```
def __init__(self, user_file, user_key):
    # Get the path to input file
    self.user_file = user_file
    self.input_file_size = os.path.getsize(self.user_file)
    self.chunk_size = 1024
    self.total_chunks = (self.input_file_size / self.chunk_size) + 1
    self.file_extension = self.user_file.split(".")[-1]
    self.encrypt_output_file = self.user_file + ".kryp"
    self.user_key = bytes(user_key, "utf-8")
    self.user_salt = secrets.token_bytes(16)  # Generate a 16-byte secure random salt
    self.decrypt_output_file = self.user_file[:-5].split(".")
    self.decrypt_output_file = (".".join(self.decrypt_output_file[:-1])
                                + "_dekrypted_." + self.decrypt_output_file[-1])
    self.hash_type = "SHA256"
    self.hashed_key_salt = dict()
    self.hash_key_salt()
```

- **self.user_file = user_file**
  user_file: Path to the input file to be encrypted or decrypted.
- **self.input_file_size = os.path.getsize(self.user_file)**
  Get the path to input file
- **self.chunk_size = 1024**
  This line sets the chunk size to 1024 bytes. This value determines how much data will be read from the input file at a time during encryption. Chunk size of 1024 bytes provides a good balance between memory usage and performance, making the encryption and decryption processes efficient and manageable even for large files.
- **self.total_chunks = (self.input_file_size / self.chunk_size) + 1**
  Calculates the total number of chunks required to read the entire input file. It divides the input file size by the chunk size and adds 1 to ensure that all data is accounted for, even if it doesn't evenly divide into chunks.
- **self.file_extension = self.user_file.split(".")[-1]**
  This line extracts the file extension from the 'user_file' path by splitting the path based on the dot (.) separator and taking the last element of the resulting list. This assumes that the file extension is the part of the file name following the last dot. [-1] index is used to access the last element in the list returned by the split method
- **self.encrypt_output_file = self.user_file + ".kryp"**
  It adds ".kryp" the original file extension.

- **self.user_key = bytes(user_key, "utf-8")**
  Converts the user_key provided by the user (string) into a bytes object using UTF-8 encoding. This ensures that the key is in a suitable format for cryptographic operations.
  Note:
  UTF-8 (Unicode Transformation Format, 8-bit) is a variable-width character encoding capable of encoding all 1,112,064 valid character code points in Unicode. It uses a minimum of one byte (8 bits) to encode ASCII characters, and up to four bytes for other characters. It's widely used on the internet and in computing systems as it's backward compatible with ASCII and supports a vast range of characters from various languages and symbols.
- **self.user_salt = secrets.token_bytes(16)**
  Usesd to generate 16 cryptographically secure random bytes for the salt.
- **self.decrypt_output_file = self.user_file[:-5].split(".")**
  Removes the ".kryp" extension splits the resulting string into a list based on '.'
- **self.decrypt_output_file = (".".join(self.decrypt_output_file[:-1])**
  + "_dekrypted_."
  +self.decrypt_output_file[-1])
  Used to construct the output filename for the decrypted file, if the original filename is "data.txt", the decrypted output filename would become "data_dekrypted.txt".
- **self.hash_type = "SHA256"**
  Set "SHA256" to which defines the hashing algorithm used for key derivation.
- **self.hashed_key_salt = dict()**
  Empty dictionary to store hashed key and salt.
  Note:
  Salting enhances the security of hashed passwords and encrypted data by adding randomness and uniqueness to each hashed value, making it significantly more challenging for attackers to crack passwords using various attack methods.
- self.hash_key_salt()
  Calls the hash_key_salt method to perform the hashing.

10.1.1.6 "**hash_key_salt**" function:

This function is responsible for hashing the user-provided key and the randomly generated salt to enhance security in the encryption process.

```
def hash_key_salt(self):
    """Converts key to hash"""
    hasher = hashlib.new(self.hash_type)            # Create a new hash object
    hasher.update(self.user_key)
    self.hashed_key_salt["key"] = hasher.digest()
    hasher.update(self.user_salt)
    self.hashed_key_salt["salt"] = hasher.digest()
```

- **hasher = hashlib.new(self.hash_type)**
  This line creates a new hashing object using the hashlib library. The specific hashing algorithm is determined by self.hash_type, which is likely set to "SHA256" in the class constructor.
- **hasher.update(self.user_key)**

This line updates the hash object with the user's secret key. The update method adds data to the hasher object, and the key is converted to bytes before being fed into the hasher since hashing algorithms operate on byte data.

- **self.hashed_key_salt["key"] = hasher.digest()**

This line retrieves the digest (hashed output) from the hasher object and stores it in the class dictionary self.hashed_key_salt under the key named "key". This hashed key will be used later in the encryption process.

- **hasher.update(self.user_salt)**

The hash object is updated again, but this time with the self.user_salt value. Recall that user_salt is a random byte string generated earlier in the class constructor to improve key security.

- **self.**hashed_key_salt[**"salt"**] = hasher.digest()

The final hashed output (digest) containing both the key and salt is retrieved from the hasher and stored in the class dictionary self.hashed_key_salt under the key "salt".

10.1.1.7 "**read_in_chunks"** function:

It is a generator function used to read a file in chunks.

```
def read_in_chunks(file_object, chunk_size=1024):
    while True:
        data = file_object.read(chunk_size)
        if not data:
            break           # If the end of the file is reached, the loop breaks
        yield data
        # This allows you to iterate over the file content chunk by chunk without reading the entire file
        # into memory at once.
```

It takes two arguments:

- **file_object:** This is a file object opened in read mode.
- **chunk_size:** This is the size of each chunk to be read from the file. Defaults to 1024 bytes (1 KB).
- Why 1024 bytes?

  Chunk size of 1024 bytes provides a good balance between memory usage and performance, making the encryption and decryption processes efficient and manageable even for large files.

Benefits of using read_in_chunks:

- **Memory efficiency:** By reading the file in chunks, it avoids loading the entire file into memory at once. This is especially beneficial for large files.
- **Performance improvement:** For some operations, it can be more efficient to process data chunk by chunk, especially for network transfers.

10.1.1.8 "**encrypt**" function:

It handles the encryption process of a file in the "EncryptionTool" class.

```
def encrypt(self):
    iv = secrets.token_bytes(16)
    # Create a cipher object
    cipher_object = AES.new(self.hashed_key_salt["key"], AES.MODE_CFB, iv)

    self.abort()                        # Remove existing output files if any
    input_file = open(self.user_file, "rb")
    output_file = open(self.encrypt_output_file, "wb")
    output_file.write(self.user_salt)  # Write the salt at the beginning of the file
    output_file.write(iv)              # Write the IV after the salt
```

```
    done_chunks = 0
    for piece in read_in_chunks(input_file, self.chunk_size):
        encrypted_content = cipher_object.encrypt(piece)
        output_file.write(encrypted_content)
        done_chunks += 1
        yield (done_chunks / self.total_chunks) * 100

    input_file.close()              # Closes the input file after reading.
    output_file.close()             # Closes the output file after writing the encrypted data
```

Step of encryption process:

1. Initialization Vector (IV) generation:

    - **iv = secrets.token_bytes(16)**
      This line generates a random 16-byte initialization vector (IV) using the secrets module. The IV is a crucial element in AES encryption mode (CFB used here) to ensure that identical plaintexts produce different ciphertexts.

2. Cipher object creation:
    - **cipher_object = AES.new(self.hashed_key_salt["key"], AES.MODE_CFB,iv)**
      This line creates a new AES cipher object using the Crypto.Cipher module. The arguments provided are:

- self.hashed_key_salt["key"]: The previously hashed key retrieved from the class dictionary.
- AES.MODE_CFB: This specifies the AES Cipher Feedback Block (CFB) mode, which is a popular mode for stream ciphers.
- iv: The initialization vector generated earlier.

3. Cleaning up existing output files:
   - **self.abort()**
     This line calls the abort method from the class (not shown in the provided code). The purpose of this call is likely to remove any partially created or leftover encrypted files from previous attempts to prevent confusion or data corruption.

4. Opening files:
   - **input_file = open(self.user_file, "rb")**
     This line opens the user-specified file (**self**.user_file) in binary read mode ("rb").
   - **output_file = open(self.encrypt_output_file, "wb")**
     This line opens a new file (**self**.encrypt_output_file - likely with a ".kryp" extension) in binary write mode ("wb") for writing the encrypted content.

5. Writing salt and IV:
   - **output_file.write(self.user_salt)**
     The randomly generated salt (**self**.user_salt) is written to the beginning of the output file. This ensures the salt is included in the encrypted data for decryption purposes.
   - **output_file.write(iv)**
     The initialization vector (IV) is then written to the output file, following the salt.

6. Encrypting file chunks:
   - **Looping through read_in_chunks(input_file, self.chunk_size)**
     This part iterates over the user's file in chunks of a specific size (**self**.chunk_size - likely 1024 bytes as defined elsewhere).
   - **encrypted_content = cipher_object.encrypt(piece):**
     Inside the loop, each chunk (piece) of the file is fed to the encrypt method of the cipher object. This encrypts the chunk using the AES-CFB mode with the provided key and IV.
   - **output_file.write(encrypted_content)**
     The encrypted chunk (encrypted_content) is then written to the output file.

7. Progress counter:
   - **done_chunks += 1:**
     This line keeps track of the number of processed chunks.
   - **yield (done_chunks / self.total_chunks) * 100**
     This line utilizes the yield keyword, suggesting this function might be used within a generator context. It calculates the encryption progress as a percentage and

potentially yields this value to an external function that can update a progress bar or display status information to the user.

8. Closing files:
   - **input_file.close()**
     After processing all chunks, the input file is closed.
   - **output_file.close()**
     The output file containing the encrypted data is also closed.

10.1.1.9 "d**ecrypt**" function:

```
def decrypt(self):
    # The Exact same as above function except in reverse
    input_file = open(self.user_file, "rb")
    self.user_salt = input_file.read(16)  # Read the salt from the beginning of the file
    iv = input_file.read(16)              # Read the IV after the salt
    self.hash_key_salt()                  # Re-hash the key with the read salt

    cipher_object = AES.new(self.hashed_key_salt["key"], AES.MODE_CFB, iv)

    self.abort()            # if the output file already exists, remove it first
    output_file = open(self.decrypt_output_file, "wb")

    done_chunks = 0
    for piece in read_in_chunks(input_file):
        decrypted_content = cipher_object.decrypt(piece)
        output_file.write(decrypted_content)
        done_chunks += 1
        yield (done_chunks / self.total_chunks) * 100

    input_file.close()      # Closes the input file after reading.
    output_file.close()     # Closes the output file after writing the encrypted data
```

Decryption code is like encryption code.

1. **Opening files:**
   - **input_file = open(self.user_file, "rb")**
     Like encryption, the user-specified file (self.user_file) is opened in binary read mode ("rb") for reading the encrypted content.

2. **Reading salt and IV:**
   - **self.user_salt = input_file.read(16)**
     The first 16 bytes of the encrypted file are read and stored in self.user_salt. This is the salt that was written during encryption.
   - **iv = input_file.read(16)**

The next 16 bytes are read and assigned to iv. This represents the initialization vector used in encryption.

3. **Key re-hashing:**
   - **self.hash_key_salt()**
     This line is called the hash_key_salt function from the class (already explained previously). However, in decryption, the user_salt just read from the file is used for hashing along with the user's key. This ensures the correct key is derived for decryption.

4. **Cipher object creation:**
   - **cipher_object = AES.new(self.hashed_key_salt["key"], AES.MODE_CFB, iv)**
     A new AES cipher object is created using the re-hashed key, the same AES-CFB mode, and the retrieved initialization vector.

5. **Cleaning Up Existing Output File (optional):**
   - **self.abort()**
     Like encryption, this line might optionally remove any leftover decrypted files from previous attempts.

6. **Opening output file for decryption:**
   - **output_file = open(self.decrypt_output_file, "wb")**
     A new file with the appropriate name (self.decrypt_output_file - likely with a "_dekrypted" extension) is opened in binary write mode ("wb") to write the decrypted content.

7. **Decrypting file chunks:**
   - **For piece in read_in_chunks(input_file)**
     The encrypted file is again processed in chunks using the read_in_chunks function.
   - **decrypted_content = cipher_object.decrypt(piece)**
     Each chunk (piece) is fed to the decrypt method of the cipher object. This reverses the encryption using the AES-CFB mode with the recreated key and IV.
   - **output_file.write(decrypted_content)**
     The decrypted chunk (decrypted_content) is written to the output file.

8. **Progress counter:**
   Like encryption, the code keeps track of processed chunks (done_chunks) and calculates the decryption progress as a percentage, potentially yielding it for progress updates.

9. **Closing files:**
   - **input_file.close()**
     After processing all chunks, the input file is closed.
   - **output_file.close()**
     The output file containing the encrypted data is also closed.

10.1.1.10 "**abort**" function:

```
def abort(self):
    if os.path.isfile(self.encrypt_output_file):
        os.remove(self.encrypt_output_file)
    if os.path.isfile(self.decrypt_output_file):
        os.remove(self.decrypt_output_file)
```

The provided function abort is designed to handle the removal of potentially existing output files before encryption or decryption starts in the Encryption Tool class. Here's a breakdown of its functionality:

Functionality:

- This function doesn't take any arguments as it operates on internal class variables.
- It uses two if statements to check for specific files:
    - **if os.path.isfile(self.encrypt_output_file)**
      This line checks if a file path stored in self.encrypt_output_file (likely the path for the encrypted output file) exists using the os.path.isfile function from the os library. If the file exists, the os.remove(self.encrypt_output_file) statement is executed to remove the file.

    - The second if statement performs the same check but for the self.decrypt_output_file path (likely the path for the decrypted output file).

Purpose:

- The primary purpose of this function is to prevent conflicts during encryption or decryption. If the output file already exists from a previous operation, it might contain partially written data or unexpected content. Deleting it before starting a new operation ensures a clean slate for the encryption or decryption process.

10.1.1.11 "**__init__**" function from Main Window Class:

```
def __init__(self, root):
    self.root = root
    self._file_url = tk.StringVar()
    self._secret_key = tk.StringVar()
    self._status = tk.StringVar()
    self._status.set("---")
    self.should_cancel = False
```

```
    root.title("FOTA: Encryption and Decryption Tool")
    root.configure(bg="#eeeeee", relief=tk.FLAT)

    self.menu_bar = tk.Menu(root, bg="#eeeeee", relief=tk.FLAT)
    self.menu_bar.add_command(label="Owners", command=self.show_help_callback)
    self.menu_bar.add_command(label="Exit", command=root.quit)
    root.configure(menu=self.menu_bar)
    self.file_entry_label = tk.Label(root, text="FOTA-File", bg="#eeeeee", anchor=tk.W)
    self.file_entry_label.grid(padx=12, pady=(8, 0), row=0, column=0, columnspan=4,
                               sticky=tk.W + tk.E + tk.N + tk.S)

    self.file_entry = tk.Entry(root, textvariable=self._file_url, bg="#fff", relief=tk.FLAT)

    self.file_entry.grid(padx=15, pady=6, ipadx=8, ipady=8,
                         row=1, column=0, columnspan=4, sticky=tk.W + tk.E + tk.N + tk.S)
    self.select_btn = tk.Button(root, text="SELECT FILE", command=self.selectfile_callback,
                            width=42, bg="#1089ff", fg="#ffffff", bd=2, relief=tk.FLAT)

    self.select_btn.grid(padx=15, pady=8, ipadx=8, ipady=8,
                         row=2, column=0, columnspan=4, sticky=tk.W + tk.E + tk.N + tk.S)

    self.key_entry_label = tk.Label(root, text="Enter Key", bg="#eeeeee", anchor=tk.W)
    self.select_btn = tk.Button(root, text="SELECT FILE", command=self.selectfile_callback,
                            width=42, bg="#1089ff", fg="#ffffff", bd=2, relief=tk.FLAT)

    self.select_btn.grid(padx=15, pady=8, ipadx=8, ipady=8,
                         row=2, column=0, columnspan=4, sticky=tk.W + tk.E + tk.N + tk.S)

    self.key_entry_label = tk.Label(root, text="Enter Key", bg="#eeeeee", anchor=tk.W)

    self.key_entry_label.grid(padx=12, pady=(8, 0), row=3, column=0, columnspan=4,
                              sticky=tk.W + tk.E + tk.N + tk.S)
    self.key_entry = tk.Entry(root, textvariable=self._secret_key, bg="#fff", relief=tk.FLAT, show="*")
    self.key_entry.grid(padx=15, pady=6, ipadx=8, ipady=8,
                        row=4, column=0, columnspan=4, sticky=tk.W + tk.E + tk.N + tk.S)

    self.encrypt_btn = tk.Button(root, text="ENCRYPT", command=self.encrypt_callback, bg="#ed3833",
                                 fg="#ffffff", bd=2, relief=tk.FLAT)
    self.encrypt_btn.grid(padx=(15, 6), pady=8, row=7, column=0, columnspan=2, sticky=tk.W + tk.E + tk.N + tk.S)

    self.decrypt_btn = tk.Button(root, text="DECRYPT", command=self.decrypt_callback, bg="#00bd56",
                                 fg="#ffffff", bd=2, relief=tk.FLAT)
    self.decrypt_btn.grid(padx=(6, 15), pady=8, row=7, column=2, columnspan=2, sticky=tk.W + tk.E + tk.N + tk.S)

    self.reset_btn = tk.Button(root, text="RESET", command=self.reset_callback, bg="#aaaaaa",
                               fg="#ffffff", bd=2, relief=tk.FLAT)
```

```
self.reset_btn.grid(padx=15, pady=(4, 12), ipadx=8, ipady=8,
                    row=8, column=0, columnspan=4, sticky=tk.W + tk.E + tk.N + tk.S)

self.status_label = tk.Label(root, textvariable=self._status, bg="#eeeeee", anchor=tk.W,
                             justify=tk.LEFT, relief=tk.FLAT, wraplength=350)
self.status_label.grid(padx=12, pady=(0, 12), row=9, column=0, columnspan=4, sticky=tk.W + tk.E + tk.N + tk.S)

tk.Grid.columnconfigure(root, 0, weight=1)
tk.Grid.columnconfigure(root, 1, weight=1)
tk.Grid.columnconfigure(root, 2, weight=1)
tk.Grid.columnconfigure(root, 3, weight=1)
```

This code snippet initializes the main window (**MainWindow**) of the encryption and decryption tool.

1. **Setting up root window:**
    - Takes the root window (**root**) as input.
    - Stores the reference to the root window (**self.root**).
    - Creates string variables (**tk.StringVar()**) to hold the file path **(_file_url**), secret key (**_secret_key**), and status message (_status).
    - Sets the initial status message to an empty string ("---").
    - Creates a flag (**should_cancel**) to indicate if an ongoing encryption/decryption process should be cancelled.

2. **Configuring root window:**
    - Sets the title of the window to "FOTA: Encryption and Decryption Tool".
    - Configures the background color and relief style of the window.

3. **Creating Menu Bar:**
    - Creates a menu bar (**self.menu_bar**) with a flat relief style and light gray background.
    - Adds two menu items:
        - "Owners" - triggers the **show_help_callback** function to display a message box with owner information.
        - "Exit" - triggers the **root.quit()** method to close the application.
    - Assigns the menu bar to the root window.

4. **Creating Labels and Entry Fields:**
    - Creates labels for "FOTA-File" and "Enter Key".
    - Creates entry fields for the file path (**self.file_entry**) and secret key (**self.key_entry**). The key entry displays asterisks (*) to hide the characters as the user types.

5. **Creating Buttons:**
    - Creates three buttons:
        - "SELECT FILE" - triggers the selectfile_callback function to open a file dialog for selecting the file to encrypt/decrypt.

- "ENCRYPT" - triggers the encrypt_callback function to initiate the encryption process.
- "DECRYPT" - triggers the decrypt_callback function to initiate the decryption process.
- "RESET" - triggers the **reset_callback** function to clear the file path, key, and status message.

6. **Creating Status Label:**
   - Creates a label (**self.status_label**) to display status messages during encryption/decryption or errors.
   - Sets the background color, justification (left-aligned), relief style, and wraps text to fit within 350 pixels.

7. **Configuring Grid Layout:**
   - Configures the columns of the grid layout in the root window to have equal weights for a balanced look.
   - Uses grid geometry manager to arrange all the labels, entry fields, buttons, and the status label in their respective positions within the window.

10.1.1.12 "**selectfile_callback**" function:

```
def selectfile_callback(self):
    try:
        file_path = filedialog.askopenfilename()
        self._file_url.set(file_path)
    except Exception as e:  # as e: This part assigns the encountered exception to the variable e
        # This allows you to access details about the error if needed.
        self._status.set(str(e))
```

It's defined within the Main Window class. This function will be called whenever the "SELECT FILE" button is clicked.

- This block uses a try-except statement to handle potential errors during file selection.
- file_path = filedialog.askopenfilename()
  This line calls the askopenfilename function from the filedialog module. This function opens a file dialog window where the user can browse and select a file. The selected file path is then stored in the file_path variable.
- self._file_url.set(file_path)
  This line updates the internal state of the application with the chosen file path. Here, self._file_url is a tk.StringVar() variable that holds the file path. By setting its value to file_path, the application now knows which file the user has selected.
- self._status.set(str(e))
  This line displays an error message in the status label. It converts the exception (e) to a string using str(e) and then sets it as the status message using self._status.set().

10.1.1.13 “**freeze_controls**” function:

```
def freeze_controls(self):
    for widget in (self.file_entry, self.key_entry, self.select_btn, self.encrypt_btn,
                   self.decrypt_btn, self.reset_btn):
        widget.configure(state="disabled")
    self.status_label.update()
```

This function disables various controls in the user interface to prevent user interaction during encryption or decryption, while potentially updating the status label to indicate a busy state. This helps maintain program stability and prevents unexpected behavior if the user tries to modify settings or perform actions while the encryption/decryption process is in progress.

- **self.file_entry**: Entry field for the file path.
- **self.key_entry**: Entry field for the secret key.
- **self.select_btn**: "SELECT FILE" button.
- **self.encrypt_btn**: "ENCRYPT" button.
- **self.decrypt_btn**: "DECRYPT" button.
- **self.reset_btn**: "RESET" button.
- **widget.configure(state="disabled")**
  Disables various UI elements (file path entry, key entry, buttons) to prevent user interaction while encryption/decryption is happening.
- **self.status_label.update()**
  Refreshes the status label to ensure any updates are shown immediately after disabling the controls.

10.1.1.14 “**unfreeze_controls**” function:

```
def unfreeze_controls(self):
    for widget in (self.file_entry, self.key_entry, self.select_btn, self.encrypt_btn,
                   self.decrypt_btn, self.reset_btn):
        widget.configure(state="normal")
    self.status_label.update()
```

Same as “freeze_controls” function but the here the function re-enables user interaction with the controls after encryption or decryption processes

10.1.1.15 “**encrypt_callback**” function:

```
def encrypt_callback(self):
    self.freeze_controls()
    try:
        cipher = EncryptionTool(self._file_url.get(), self._secret_key.get())
        for percentage in cipher.encrypt():
            if self.should_cancel:
                break
            percentage = "{0:.2f}%".format(percentage)
            self._status.set(percentage)
            self.status_label.update()
        self._status.set("File Encrypted!")
        if self.should_cancel:
            cipher.abort()
            self._status.set("Cancelled!")
        self.should_cancel = False
    except Exception as e:
        # print(e)
        self._status.set(str(e))
    self.unfreeze_controls()
```

The encrypt_callback function is triggered when the user clicks the "ENCRYPT" button in the graphical user interface.

- **self.freeze_controls()**
  Disables UI elements like file entry, key entry, and buttons to prevent user interaction during encryption.

- **cipher = EncryptionTool(self._file_url.get(), self._secret_key.get())**
  - Creates an instance of the EncryptionTool class, likely defined elsewhere in the code.
  - **self._file_url.get()** retrieves the current file path from the _file_url string variable.
  - **self._secret_key.get()** retrieves the current secret key from the _secret_key string variable.
  - Initializes the encryption process using the provided file path and key.
- **for percentage in cipher.encrypt()**
  - Calls the encrypt method of the cipher object.
  - This method handles the encryption process in chunks and returns an inerrable of percentages (0.0 to 100.0) representing progress.
  - The “for” loop iterates through each percentage value.
- **if self.should_cancel:**
  Checks a flag **self.should_cancel**. This flag might be set by another part of the code (e.g., a "Cancel" button) to indicate that the user wants to interrupt the encryption process.
- **percentage = "{0:.2f}%".format(percentage):**

Formats the percentage value (a number between 0.0 and 1.0) to a string with two decimal places and a "%" sign for visual representation of progress (e.g., "50.00%").

- **self._status.set(percentage)**
  Updates the _status string variable with the formatted percentage string. This variable holds the text displayed in the status label.
- **self.status_label.update()**
  Refreshes the status label to ensure the progress update is visible to the user.
- **self._status.set("File Encrypted!")**
  After the loop completes (assuming no cancellation), sets the status message to "File Encrypted!" indicating successful completion.
- **if self.should_cancel:**
  Checks the cancellation flag again.
- **self._status.set("Cancelled!"):**
  If cancellation occurred, sets the status message to "Cancelled!" to inform the user.
- **self.should_cancel = False:**
  Resets the cancellation flag to False, assuming it's only used for the current encryption operation.
- **self._status.set(str(e))**
  Converts the exception (e) to a string and sets it as the status message. This displays an error message to the user if the encryption fails.
- **self.unfreeze_controls()**
  Regardless of success or failure, reenables the UI controls that were disabled earlier using freeze_controls. This allows the user to interact with the application again.

10.1.1.16 “**decrypt_callback**” function:

```
def decrypt_callback(self):
    self.freeze_controls()
    try:
        cipher = EncryptionTool(self._file_url.get(), self._secret_key.get())
        for percentage in cipher.decrypt():
            if self.should_cancel:
                break
            percentage = "{0:.2f}%".format(percentage)
            self._status.set(percentage)
            self.status_label.update()
        self._status.set("File Decrypted!")
        if self.should_cancel:
            cipher.abort()
            self._status.set("Cancelled!")
        self.should_cancel = False
    except Exception as e:
        # print(e)
        self._status.set(str(e))

    self.unfreeze_controls()
```

The same as “encrypt_callback” for handling progress, user interaction, error handling, and user feedback. However, they differ in the specific methods called on the Encryption Tool object (encrypt for encryption, decrypt for decryption) and the success message displayed to the user.

10.1.1.17 “**reset_callback**” function:

```
def reset_callback(self):
    self._file_url.set("")
    self._secret_key.set("")
    self._status.set("---")
```

This function essentially resets the user interface by clearing the displayed file path, secret key, and status message. This allows the user to start over with a clean slate for a new encryption or decryption task.

10.1.1.18 “**cancel_callback**” function:

```
def cancel_callback(self):
    """Sets a flag to cancel the ongoing encryption/decryption process."""
    self.should_cancel = True
```

This function doesn't take any arguments as it modifies an internal variable within the class.

The **should_cancel** flag is likely checked within the “**encrypt_callback**” and “**decrypt_callback**” functions. These functions are responsible for carrying out the encryption and decryption operations. By setting should_cancel to True, the “**encrypt_callback**” and

"**decrypt_callback**" functions can interrupt their execution and stop processing any further data chunks.

10.1.1.19 "show_help_callback" function:

```
1 usage
@staticmethod
def show_help_callback():
    messagebox.showinfo( title: "Owners",  message: "Eman - Esraa - Huda - Mohamed - Omar - Omnia - Rashad - Sagy - Yasmeen")
```

The static method displays a message box with a list of owners when clicking on the "Owner" list in Menu Bar.

"show_help_callback" is a static method. Static methods are methods that don't receive an implicit first argument (self) which refers to the class instance. They are typically used for utility functions that don't rely on class attributes or methods.

10.1.4 The Code

```
import os          # provides a portable way of using operating system-dependent     functionality.
import sys         # provides access to some variables used or maintained by the Python interpreter
import hashlib  # provides a common interface to many secure hashes and message digest
algorithms.
import tkinter as tk
from tkinter import filedialog
from tkinter import messagebox
from Crypto.Cipher import AES
import secrets    # provides functions for generating cryptographically secure random numbers.

class EncryptionTool:
    def __init__(self, user_file, user_key):
        # Get the path to input file
        self.user_file = user_file
        self.input_file_size = os.path.getsize(self.user_file)
        self.chunk_size = 1024
        self.total_chunks = (self.input_file_size / self.chunk_size) + 1
        self.file_extension = self.user_file.split(".")[-1]
        self.encrypt_output_file = self.user_file + ".kryp"
        self.user_key = bytes(user_key, "utf-8")
        self.user_salt = secrets.token_bytes(16)  # Generate a 16-byte secure random salt
        self.decrypt_output_file = self.user_file[:-5].split(".")
        self.decrypt_output_file = (".".join(self.decrypt_output_file[:-1])
                                    + "_dekrypted_." + self.decrypt_output_file[-1])
        self.hash_type = "SHA256"
        self.hashed_key_salt = dict()
        self.hash_key_salt()

    def hash_key_salt(self):
```

```
        """Converts key to hash"""
        hasher = hashlib.new(self.hash_type)  # Create a new hash object
        hasher.update(self.user_key)
        self.hashed_key_salt["key"] = hasher.digest()
        hasher.update(self.user_salt)
        self.hashed_key_salt["salt"] = hasher.digest()

    def encrypt(self):
        iv = secrets.token_bytes(16)
        # Create a cipher object
        cipher_object = AES.new(self.hashed_key_salt["key"], AES.MODE_CFB, iv)

        self.abort()                    # Remove existing output files if any
        input_file = open(self.user_file, "rb")
        output_file = open(self.encrypt_output_file, "wb")
        output_file.write(self.user_salt)  # Write the salt at the beginning of the file
        output_file.write(iv)             # Write the IV after the salt

        done_chunks = 0
        for piece in read_in_chunks(input_file, self.chunk_size):
            encrypted_content = cipher_object.encrypt(piece)
            output_file.write(encrypted_content)
            done_chunks += 1
            yield (done_chunks / self.total_chunks) * 100

        input_file.close()              # Closes the input file after reading.
        output_file.close()             # Closes the output file after writing the encrypted data

    def decrypt(self):
        # The Exact same as above function except in reverse
        input_file = open(self.user_file, "rb")
        self.user_salt = input_file.read(16)  # Read the salt from the beginning of the file
        iv = input_file.read(16)              # Read the IV after the salt
        self.hash_key_salt()                  # Re-hash the key with the read salt

        cipher_object = AES.new(self.hashed_key_salt["key"], AES.MODE_CFB, iv)

        self.abort()            # if the output file already exists, remove it first
        output_file = open(self.decrypt_output_file, "wb")

        done_chunks = 0
        for piece in read_in_chunks(input_file):
            decrypted_content = cipher_object.decrypt(piece)
            output_file.write(decrypted_content)
            done_chunks += 1
            yield (done_chunks / self.total_chunks) * 100
```

```
        input_file.close()      # Closes the input file after reading.
        output_file.close()     # Closes the output file after writing the encrypted data

    def abort(self):
        if os.path.isfile(self.encrypt_output_file):
            os.remove(self.encrypt_output_file)
        if os.path.isfile(self.decrypt_output_file):
            os.remove(self.decrypt_output_file)

def read_in_chunks(file_object, chunk_size=1024):
    while True:
        data = file_object.read(chunk_size)
        if not data:
            break         # If the end of the file is reached, the loop breaks
        yield data       # This allows you to iterate over the file content chunk by chunk without
reading the  # entire file into memory at once.


class MainWindow:
    """ GUI Wrapper """
    THIS_FOLDER_G = ""
    if getattr(sys, "frozen", False):
        # frozen
        THIS_FOLDER_G = os.path.dirname(sys.executable)
    else:
        # unfrozen
        THIS_FOLDER_G = os.path.dirname(os.path.realpath(__file__))

    def __init__(self, root):
        self.root = root
        self._file_url = tk.StringVar()
        self._secret_key = tk.StringVar()
        self._status = tk.StringVar()
        self._status.set("---")
        self.should_cancel = False

        root.title("FOTA: Encryption and Decryption Tool")
        root.configure(bg="#eeeeee", relief=tk.FLAT)

        self.menu_bar = tk.Menu(root, bg="#eeeeee", relief=tk.FLAT)
        self.menu_bar.add_command(label="Owners", command=self.show_help_callback)
        self.menu_bar.add_command(label="Exit", command=root.quit)

        root.configure(menu=self.menu_bar)
```

```
        self.file_entry_label = tk.Label(root, text="FOTA-File", bg="#eeeeee", anchor=tk.W)
        self.file_entry_label.grid(padx=12, pady=(8, 0), row=0, column=0, columnspan=4,
                                   sticky=tk.W + tk.E + tk.N + tk.S)

        self.file_entry = tk.Entry(root, textvariable=self._file_url, bg="#fff", relief=tk.FLAT)

        self.file_entry.grid(padx=15, pady=6, ipadx=8, ipady=8,
                             row=1, column=0, columnspan=4, sticky=tk.W + tk.E + tk.N + tk.S)

        self.select_btn = tk.Button(root, text="SELECT FILE", command=self.selectfile_callback,
                                    width=42, bg="#1089ff", fg="#ffffff", bd=2, relief=tk.FLAT)

        self.select_btn.grid(padx=15, pady=8, ipadx=8, ipady=8,
                             row=2, column=0, columnspan=4, sticky=tk.W + tk.E + tk.N + tk.S)

        self.key_entry_label = tk.Label(root, text="Enter Key", bg="#eeeeee", anchor=tk.W)

        self.key_entry_label.grid(padx=12, pady=(8, 0), row=3, column=0, columnspan=4,
                                  sticky=tk.W + tk.E + tk.N + tk.S)

        self.key_entry = tk.Entry(root, textvariable=self._secret_key, bg="#fff", relief=tk.FLAT,
show="*")

        self.key_entry.grid(padx=15, pady=6, ipadx=8, ipady=8,
                            row=4, column=0, columnspan=4, sticky=tk.W + tk.E + tk.N + tk.S)

        self.encrypt_btn = tk.Button(root, text="ENCRYPT", command=self.encrypt_callback,
bg="#ed3833", fg="#ffffff", bd=2, relief=tk.FLAT)

        self.encrypt_btn.grid(padx=(15, 6), pady=8, row=7, column=0, columnspan=2,
                                    sticky=tk.W + tk.E + tk.N + tk.S)

        self.decrypt_btn = tk.Button(root, text="DECRYPT", command=self.decrypt_callback,
                                             bg="#00bd56", fg="#ffffff", bd=2, relief=tk.FLAT)

        self.decrypt_btn.grid(padx=(6, 15), pady=8, row=7, column=2, columnspan=2,
                                    sticky = tk.W + tk.E + tk.N + tk.S)

        self.reset_btn = tk.Button(root, text="RESET", command=self.reset_callback,
bg="#aaaaaa",
                                           fg="#ffffff", bd=2, relief=tk.FLAT)

        self.reset_btn.grid(padx=15, pady=(4, 12), ipadx=8, ipady=8,
                                  row=8, column=0, columnspan=4, sticky=tk.W + tk.E + tk.N + tk.S)
```

```
    self.status_label = tk.Label(root, textvariable=self._status, bg="#eeeeee", anchor=tk.W,
                                 justify=tk.LEFT, relief=tk.FLAT, wraplength=350)

    self.status_label.grid(padx=12, pady=(0, 12), row=9, column=0, columnspan=4,
                           sticky = tk.W + tk.E + tk.N + tk.S)

    tk.Grid.columnconfigure(root, 0, weight=1)
    tk.Grid.columnconfigure(root, 1, weight=1)
    tk.Grid.columnconfigure(root, 2, weight=1)
    tk.Grid.columnconfigure(root, 3, weight=1)

def selectfile_callback(self):
    try:
        file_path = filedialog.askopenfilename()
        self._file_url.set(file_path)
    except Exception as e:
        self._status.set(str(e))

def freeze_controls(self):
    for widget in (self.file_entry, self.key_entry, self.select_btn, self.encrypt_btn,
                   self.decrypt_btn, self.reset_btn):
        widget.configure(state="disabled")
    self.status_label.update()

def unfreeze_controls(self):
    for widget in (self.file_entry, self.key_entry, self.select_btn, self.encrypt_btn,
                   self.decrypt_btn, self.reset_btn):
        widget.configure(state="normal")
    self.status_label.update()

def encrypt_callback(self):
    self.freeze_controls()
    try:
        cipher = EncryptionTool(self._file_url.get(), self._secret_key.get())
        for percentage in cipher.encrypt():
            if self.should_cancel:
                break
            percentage = "{0:.2f}%".format(percentage)
            self._status.set(percentage)
            self.status_label.update()
        self._status.set("File Encrypted!")
        if self.should_cancel:
            cipher.abort()
            self._status.set("Cancelled!")
        self.should_cancel = False
    except Exception as e:                # print(e)
```

```
            self._status.set(str(e))
        self.unfreeze_controls()

    def decrypt_callback(self):
        self.freeze_controls()
        try:
            cipher = EncryptionTool(self._file_url.get(), self._secret_key.get())
            for percentage in cipher.decrypt():
                if self.should_cancel:
                    break
                percentage = "{0:.2f}%".format(percentage)
                self._status.set(percentage)
                self.status_label.update()
            self._status.set("File Decrypted!")

            if self.should_cancel:
                cipher.abort()
                self._status.set("Cancelled!")
            self.should_cancel = False
        except Exception as e:
            # print(e)
            self._status.set(str(e))

        self.unfreeze_controls()

    def reset_callback(self):
        self._file_url.set("")
        self._secret_key.set("")
        self._status.set("---")

    def cancel_callback(self):
        """Sets a flag to cancel the ongoing encryption/decryption process."""
        self.should_cancel = True

    @staticmethod
    def show_help_callback():
        messagebox.showinfo("Owners", "Eman - Esraa - Huda - Mohamed - Omar - Omnia –
                                        Rashad -   Sagy - Yasmeen")
if __name__ == "__main__":
    ROOT = tk.Tk()
    MAIN_WINDOW = MainWindow(ROOT)
    ROOT.mainloop()
```

The decryption part for the car

- The code Decrypts files with AES algorithm (Cipher Feedback Mode).
- Uses the same secure key handling (hashed with random salt) that used in encryption part.

10.2.1 Libraries:

```
#include <SD.h>
#include <SPI.h>
#include <Crypto.h>
#include <AES.h>
#include <SHA256.h>
```

- #include <SD.h>: This includes the SD library, which refers to the header file that provides functions and definitions for working with SD (Secure Digital) cards. This library allows Arduino boards to read and write data to SD cards, which are commonly used for storing large amounts of data in a compact format.
- #include <SPI.h>: This includes the SPI library, which provides functions for using the Serial Peripheral Interface (SPI) bus on Arduino boards. SPI is a synchronous serial communication protocol commonly used to communicate between microcontrollers and peripheral devices such as sensors, displays, memory devices (including SD cards), and other microcontrollers.
- #include <Crypto.h>: This includes the Crypto library, refers to a header file that provides cryptographic functions and algorithms for secure communication and data integrity. It is part of the Arduino Cryptography Library (ArduinoCrypto) and allows Arduino boards to perform cryptographic operations such as encryption, decryption, hashing, and message authentication.
- #include <AES.h>: This includes the AES library, which provides functions and definitions specific to the AES encryption algorithm. AES (Advanced Encryption Standard) is a symmetric encryption algorithm widely used for securing data.
- #include <SHA256.h>: This includes the SHA256 library, which provides functions for calculating the SHA-256 hash of data. SHA-256 is a cryptographic hash function that generates a fixed-size hash value from input data, commonly used for data integrity verification.

10.2.2 Explanation of the code:

```
AES256 aes256;
SHA256 sha256;
```

- AES256 aes256;: This declares an object aes256 of type AES256. In the context of the Crypto library, AES256 likely represents a class or structure that provides AES-256 encryption and decryption capabilities.
- SHA256 sha256;: This declares an object sha256 of type SHA256. Similarly, in the Crypto library, SHA256 represents a class or structure that provides SHA-256 hashing capabilities.

```
unsigned char key[32];
unsigned char iv[16];
unsigned char salt[16];
unsigned char hashed_key_salt[32];
unsigned char block[16];
```

- unsigned char key[32];: This declares an array key of 32 bytes (unsigned char), which is likely intended to store an AES encryption key. AES-256, for example, uses a 256-bit (32-byte) key.
- unsigned char iv[16];: This declares an array iv of 16 bytes (unsigned char), which represents the initialization vector used for AES encryption. The initialization vector is a random or semi-random value used to ensure that encrypted data generated from the same plaintext with the same key are distinct.
- unsigned char salt[16];: This declares an array salt of 16 bytes (unsigned char). A salt is typically used in password hashing and key derivation functions to ensure that the same input (e.g., password) generates different hashes or keys.
- unsigned char hashed_key_salt[32];: This declares an array hashed_key_salt of 32 bytes (unsigned char). This is likely intended to store the result of hashing the combination of key and salt, which is a common practice in key derivation functions to derive cryptographic keys securely.
- unsigned char block[16];: This declares an array block of 16 bytes (unsigned char). In the context of AES encryption, block likely represents a data block of plaintext or ciphertext that is processed during encryption or decryption operations.

These declarations set up the necessary variables and objects to perform AES encryption and SHA-256 hashing using the Crypto library on an Arduino or similar platform capable of handling these cryptographic operations.

```
void setup() {
    Serial.begin(9600);
```

Serial.begin(9600);: Initializes serial communication at a baud rate of 9600. This allows the Arduino to communicate with an external device, typically a computer via USB.

```
    if (!SD.begin(10)) {
        Serial.println("SD card initialization failed!");
        return;
    }
```

if (!SD.begin(10)) { ... }: Attempts to initialize communication with the SD card connected to pin 10. If initialization fails, it prints an error message ("SD card initialization failed!") to the serial monitor and exits the setup function using return;.

```
    // Open the encrypted file
    File encryptedFile = SD.open("input.txt.kryp", FILE_READ);
    if (!encryptedFile) {
        Serial.println("Failed to open file");
        return;
    }
```

File encryptedFile = SD.open("input.txt.kryp", FILE_READ);: Opens a file named "input.txt.kryp" on the SD card for reading (FILE_READ is a constant indicating read mode). If opening the file fails (likely due to non-existence or permission issues), it prints an error message ("Failed to open file") and exits the setup function.

```
    // Read the salt and IV from the file
    encryptedFile.read(salt, 16);
    encryptedFile.read(iv, 16);
```

- encryptedFile.read(salt, 16);: Reads 16 bytes from the encryptedFile and stores them in the salt array.

- encryptedFile.read(iv, 16);: Reads the next 16 bytes from the encryptedFile and stores them in the iv array. These bytes typically represent the salt and initialization vector needed for decryption.

```
// Get the key from the user (or hardcoded for this example)
String userKey = "your_secret_key";
userKey.getBytes(key, 32);
```

- String userKey = "your_secret_key";: Defines a string userKey containing the secret key used for decryption. In this example, the key is hardcoded.
- userKey.getBytes(key, 32);: Converts the string userKey into bytes and copies them into the key array, which is expected to be 32 bytes long (AES-256 key size).

```
// Hash the key and salt using SHA256
sha256.reset();
sha256.update(key, strlen((char *)key));
sha256.update(salt, 16);
sha256.finalize(hashed_key_salt, sizeof(hashed_key_salt));
```

- sha256.reset();: Resets the SHA-256 hashing object to its initial state.
- sha256.update(key, strlen((char *)key));: Updates the SHA-256 hash computation with the bytes in the key array.
- sha256.update(salt, 16);: Updates the SHA-256 hash computation with the bytes in the salt array.
- sha256.finalize(hashed_key_salt, sizeof(hashed_key_salt));: Finalizes the SHA-256 hash computation and stores the resulting hash in the hashed_key_salt array, which is used as the AES key.

```
// Initialize AES with the hashed key
aes256.setKey(hashed_key_salt, sizeof(hashed_key_salt));
```

aes256.setKey(hashed_key_salt, sizeof(hashed_key_salt));: Initializes the AES-256 encryption object (aes256) with the hashed_key_salt array, which now contains the hashed key derived from the user-provided key and salt.

```
// Determine the size of the encrypted data
size_t fileSize = encryptedFile.size() - 32; // Exclude salt and IV
unsigned char* encryptedData = (unsigned char*)malloc(fileSize);
unsigned char* decryptedData = (unsigned char*)malloc(fileSize);
if (!encryptedData || !decryptedData) {
    Serial.println("Failed to allocate memory");
    encryptedFile.close();
    return;
}
```

- size_t fileSize = encryptedFile.size() - 32;: Calculates the size of the encrypted data by subtracting the sizes of salt (16 bytes) and iv (16 bytes) from the total file size.
- unsigned char* encryptedData = (unsigned char*)malloc(fileSize);: Allocates dynamic memory (malloc) to store the encrypted data.
- unsigned char* decryptedData = (unsigned char*)malloc(fileSize);: Allocates dynamic memory (malloc) to store the decrypted data.
- Checks if memory allocation was successful. If not (!encryptedData || !decryptedData), it prints an error message ("Failed to allocate memory"), closes the encryptedFile, and exits the setup function.

```
// Read the encrypted data
encryptedFile.read(encryptedData, fileSize);
encryptedFile.close();
```

- encryptedFile.read(encryptedData, fileSize);: Reads fileSize bytes of encrypted data from encryptedFile into the encryptedData array.
- encryptedFile.close();: Closes the encryptedFile after reading its content.

```
// Decrypt the data using CFB mode
for (size_t i = 0; i < fileSize; i += 16) {
    // Encrypt the IV
    aes256.encryptBlock(block, iv);

    // XOR the encrypted IV with the ciphertext block to get the plaintext block
    for (size_t j = 0; j < 16 && (i + j) < fileSize; j++) {
        decryptedData[i + j] = encryptedData[i + j] ^ block[j];
    }

    // Update the IV to the current ciphertext block
    memcpy(iv, encryptedData + i, 16);
}
```

This section decrypts the encryptedData using AES-256 in CFB (Cipher Feedback) mode:

- aes256.encryptBlock(block, iv);: Encrypts the current iv using AES and stores the result in block.
- decryptedData[i + j] = encryptedData[i + j] ^ block[j];: XORs each byte of the current ciphertext block (encryptedData) with the corresponding byte of block to obtain the plaintext (decryptedData).
- memcpy(iv, encryptedData + i, 16);: Updates the iv with the current ciphertext block to be used in the next iteration.

here isn't a standard function named `encryptBlock` in the core libraries or standard functions. However, when dealing with cryptographic operations, especially with libraries like AES (Advanced Encryption Standard) or other block ciphers, you might encounter similar methods or functions. These functions are typically part of specific libraries that implement cryptographic algorithms.

Here's a general explanation of what `.encryptBlock` might refer to in the context of cryptographic libraries:

- **Block Ciphers**: Block ciphers are cryptographic algorithms that encrypt fixed-size blocks of data (often 128 bits in AES). The process involves several rounds of substitution and permutation operations.
- **Encryption Function**: The `.encryptBlock` function would likely be part of a class or library that implements a specific block cipher algorithm (such as AES). It takes a block of plaintext data and encrypts it using a given encryption key.
- **Usage**: In practice, you would initialize the cryptographic library, set the encryption key, and then use `.encryptBlock` to encrypt each block of plaintext data sequentially. This is essential for applications requiring secure communication or data storage on microcontrollers like Arduino.

```
  // Output the decrypted data
  Serial.println("Decrypted Data:");
  Serial.write(decryptedData, fileSize);
```

Outputs the decrypted data to the serial monitor.

```
    // Free the memory
    free(encryptedData);
    free(decryptedData);
}
```

free(encryptedData); and free(decryptedData);: Releases the dynamically allocated memory used for encryptedData and decryptedData to prevent memory leaks.

10.2.3 Code

```
#include <SD.h>
#include <SPI.h>
#include <Crypto.h>
#include <AES.h>
#include <SHA256.h>

AES256 aes256;
SHA256 sha256;

unsigned char key[32];
unsigned char iv[16];     //initialization vector
unsigned char salt[16];
unsigned char hashed_key_salt[32];
unsigned char block[16];

void setup() {
   Serial.begin(9600);
   if (!SD.begin(10)) {
      Serial.println("SD card initialization failed!");
      return;
   }

   // Open the encrypted file
   File encryptedFile = SD.open("input.txt.kryp", FILE_READ);
   if (!encryptedFile) {
      Serial.println("Failed to open file");
      return;
   }

   // Read the salt and IV from the file
   encryptedFile.read(salt, 16);
   encryptedFile.read(iv, 16);

   // Get the key from the user (or hardcoded for this example)
   String userKey = "your_secret_key";
   userKey.getBytes(key, 32);

   // Hash the key and salt using SHA256
```

```
  sha256.reset();
  sha256.update(key, strlen((char *)key));
  sha256.update(salt, 16);
  sha256.finalize(hashed_key_salt, sizeof(hashed_key_salt));

  // Initialize AES with the hashed key
  aes256.setKey(hashed_key_salt, sizeof(hashed_key_salt));

  // Determine the size of the encrypted data
  size_t fileSize = encryptedFile.size() - 32; // Exclude salt and IV
  unsigned char* encryptedData = (unsigned char*)malloc(fileSize);
  unsigned char* decryptedData = (unsigned char*)malloc(fileSize);
  if (!encryptedData || !decryptedData) {
    Serial.println("Failed to allocate memory");
    encryptedFile.close();
    return;
  }
  // Read the encrypted data
  encryptedFile.read(encryptedData, fileSize);
  encryptedFile.close();
  // Decrypt the data using CFB mode
  for (size_t i = 0; i < fileSize; i += 16) {
    // Encrypt the IV
    aes256.encryptBlock(block, iv);

    // XOR the encrypted IV with the ciphertext block to get the plaintext block
    for (size_t j = 0; j < 16 && (i + j) < fileSize; j++) {
      decryptedData[i + j] = encryptedData[i + j] ^ block[j];
    }

    // Update the IV to the current ciphertext block
    memcpy(iv, encryptedData + i, 16);
  }

  // Output the decrypted data
  Serial.println("Decrypted Data:");
  Serial.write(decryptedData, fileSize);
  // Free the memory
  free(encryptedData);
  free(decryptedData);
}
void loop() {
  // Do nothing
}
```

Chapter 11 : Problems and Challenges

One of the primary challenges encountered in the Firmware Over-The-Air (FOTA) project, particularly with the integration of a mobile application (GUI) connected to an ESP32 module over Wi-Fi, is the difficulty in configuring a static IP address for the ESP32 while maintaining a reliable connection to Firebase. The ESP32, a versatile and popular microcontroller with integrated Wi-Fi and Bluetooth capabilities, often requires a dynamic IP address assigned by a DHCP server to communicate with Firebase efficiently. When attempting to assign a static IP, numerous connection failures and communication breakdowns occur, largely because Firebase expects a dynamically assigned IP and can encounter issues with static configurations due to network routing and address resolution problems. The static IP configuration often conflicts with the dynamic nature of cloud-based services like Firebase, leading to intermittent connectivity and data transmission issues.

To address this issue, one effective solution is to configure the ESP32 to use a reserved dynamic IP address within the DHCP settings of the router. By reserving an IP address for the ESP32's MAC address, the device effectively operates with a static IP from the router's perspective while still benefiting from the DHCP management. This approach ensures that the ESP32 can maintain a stable connection to Firebase without the disruptions caused by manually setting a static IP. Additionally, implementing a robust error-handling mechanism in the mobile app can help detect and recover from connection failures more gracefully, thereby improving the overall reliability of the FOTA process.

Another significant challenge in the FOTA project is ensuring seamless and secure firmware updates over the air. The process of updating firmware remotely introduces risks such as data corruption, incomplete updates, and security vulnerabilities. During the transmission of firmware data over Wi-Fi, packets can be lost or corrupted due to interference or network instability, potentially leading to a bricked device if the update is not properly verified. Moreover, ensuring the integrity and authenticity of the firmware is crucial to prevent malicious code from being uploaded to the ESP32, which could compromise the entire system.

To mitigate these risks, implementing a checksum or hash verification process is essential. By verifying the integrity of the firmware file before and after transmission, the system can detect and correct errors caused by packet loss or corruption. Additionally, employing secure communication protocols such as HTTPS for data transmission can help protect against eavesdropping and tampering. Incorporating cryptographic signatures into the firmware ensures that only authorized and authenticated updates are applied to the ESP32, safeguarding the device from malicious attacks.

The FOTA project also faces challenges related to the compatibility and stability of the mobile application across different devices and operating systems. Developing a mobile app that interacts seamlessly with the ESP32 requires addressing various issues, such as differences in Wi-Fi hardware, operating system versions, and network configurations. Users may experience

connectivity problems, app crashes, or inconsistent performance depending on their device and environment.

To tackle these issues, extensive testing across a wide range of devices and network scenarios is necessary. Ensuring compatibility with different Wi-Fi standards and thoroughly testing the app on multiple versions of Android and iOS can help identify and resolve potential issues before deployment. Additionally, gathering user feedback and implementing regular updates to address reported problems can significantly enhance the app's stability and performance.

Inconsistent Firebase Connection

Challenge: Establishing and maintaining a stable connection to Firebase can be challenging due to network issues. The ESP32 module's Wi-Fi connection can be disrupted by various factors such as signal interference, router issues, or network congestion. These disruptions lead to inconsistent data transmission between the ESP32 and Firebase, resulting in delays, data loss, and communication failures. This instability can severely impact the functionality of applications relying on real-time data synchronization, such as monitoring systems, IoT devices, and remote control applications.

Solution: Implement error handling mechanisms and retries to manage connectivity issues effectively. This can involve setting up periodic reconnection attempts when a connection drop is detected and employing exponential backoff algorithms to space out retries and reduce network load. Additionally, using local caching of data on the ESP32 ensures that critical information is not lost during temporary disconnections. Once the connection is restored, the cached data can be synchronized with Firebase. This approach ensures robust and resilient communication between the ESP32 and Firebase, maintaining data integrity and enhancing the reliability of the overall system.

Accurate Distance Measurement

Challenge: Ultrasonic sensors require precise calibration to provide accurate distance measurements, which can be affected by environmental factors such as temperature, humidity, and surface reflectivity. Variations in these conditions can lead to incorrect readings, causing erroneous distance calculations that may compromise the performance of systems dependent on accurate measurements, such as obstacle detection, level measurement, and automated parking systems.

Solution: Properly calibrate the ultrasonic sensors and ensure optimal placement and orientation to minimize interference and false readings. Calibration involves adjusting the sensor settings to account for environmental variations and verifying the accuracy of measurements against known distances. Additionally, placing the sensors in positions that minimize obstructions and reflections can improve measurement accuracy. Regular maintenance, such as cleaning the sensor surfaces and checking for physical damage, coupled with periodic testing, can help maintain the accuracy of the sensors over time. Implementing algorithms that filter out noise and compensate for environmental factors can further enhance the reliability of the distance measurements.

Smooth Motor Control

Challenge: Achieving smooth motor control is crucial for precise maneuvering and parking, particularly in automated systems. Motors can exhibit jerky movements, inconsistent speeds, and abrupt starts and stops, making it difficult to achieve smooth and controlled motion. These issues can arise from inadequate control algorithms, power supply fluctuations, or mechanical constraints, affecting the performance of robotics, automated guided vehicles, and other motor-driven applications.

Solution: Implement PWM (Pulse Width Modulation) control techniques to regulate motor speed accurately. PWM allows for precise control of motor power by varying the duty cycle of the input signal, enabling smooth acceleration and deceleration. This technique helps in achieving fine-grained control over motor speed and direction, reducing jerky movements and enhancing the overall smoothness of motor operation. Additionally, using feedback mechanisms such as encoders or sensors to monitor motor performance in real-time allows for dynamic adjustments to the control signals, ensuring consistent and smooth motor behavior. Integrating these control strategies into the motor driver software and conducting thorough testing and tuning can significantly improve the performance and reliability of motor control systems.

Chapter 12 : Future Plans

12.1 Enhanced Security Features

The security of the FOTA system is of utmost importance, as it directly affects the integrity and functionality of the devices it manages. Moving forward, the FOTA project should prioritize the implementation of end-to-end encryption protocols to safeguard the data transmitted during the update process. By employing advanced encryption standards like AES-256 for both data in transit and at rest, the system can ensure that sensitive information remains confidential and immune to tampering. Additionally, the development of robust mechanisms for authenticating update sources and authorizing devices to receive updates is crucial. This could involve integrating multi-factor authentication, leveraging digital certificates, and exploring blockchain-based authentication methods to ensure updates are only deployed from trusted sources. Automated vulnerability scanning tools should be a core part of the process, scrutinizing firmware for potential security flaws before deployment. Regular security audits and penetration testing will be essential to proactively identify and address vulnerabilities, thereby maintaining the highest standards of security. These measures will not only protect the devices from malicious attacks but also build user trust and confidence in the FOTA system.

12.2 Improved Reliability and Resilience

Reliability and resilience are critical components of a successful FOTA system, ensuring that devices remain operational and stable during and after updates. Future plans should include the creation of comprehensive fail-safe mechanisms and rollback procedures. These will allow devices to revert to a stable state if an update fails, preventing disruptions and potential data loss. Implementing dual-bank or A/B system architectures, where devices maintain a copy of the previous firmware version, can provide a reliable fallback option. Moreover, deploying redundant server architectures will enhance the availability and reliability of the update service, minimizing downtime and ensuring continuous service even during high-traffic periods or server outages. Delta updates, which involve transferring only the differences between firmware versions, should be prioritized to reduce data transfer amounts and minimize error risks, making the update process quicker and more efficient. Continuous monitoring and logging of update activities can help in swiftly identifying and resolving issues, further enhancing the system's reliability. By focusing on these areas, the FOTA project can ensure a seamless and dependable update experience for users.

12.3 Scalability and Performance Optimization

As the number of connected devices continues to grow exponentially, the FOTA system must be scalable and optimized for high performance. Future development should focus on deploying load balancers to distribute update requests evenly across servers, preventing overloads and ensuring efficient handling of high traffic volumes. Leveraging edge computing is another vital strategy, allowing updates to be processed closer to the devices, which reduces latency and conserves bandwidth. This approach can significantly improve the overall user experience by providing faster and more reliable updates. Optimizing bandwidth utilization is crucial,

especially for large-scale deployments. This can be achieved through advanced data compression techniques, efficient data transfer protocols, and intelligent scheduling of updates during off-peak times. Designing the FOTA system for horizontal scalability will allow it to accommodate an increasing number of devices and data without compromising performance. Investing in real-time performance monitoring tools will provide insights into system performance, enabling proactive adjustments to maintain optimal efficiency and responsiveness.

12.4 Advanced Device Management

The management of large fleets of devices requires sophisticated tools and capabilities. Future plans for the FOTA project should include the development of comprehensive fleet management systems that provide administrators with real-time status monitoring, detailed analytics, and flexible update scheduling. Intuitive dashboards can offer a centralized view of all devices, simplifying the management process and enabling quick decision-making. Allowing users to define custom update policies based on criteria such as device groups, geographic locations, or usage patterns will provide greater control and adaptability. Integrating remote diagnostic tools will be essential for identifying and troubleshooting issues without physical access to devices, reducing downtime and support costs. These tools can offer detailed logs and diagnostic data, aiding in swift problem resolution. Automated inventory management and device tracking features can streamline operations, ensuring that all devices are accounted for and managed efficiently throughout their lifecycle. By focusing on these areas, the FOTA project can provide a robust and scalable solution for managing large numbers of connected devices.

12.5 Enhanced User Experience

A seamless and intuitive user experience is critical for the widespread adoption and success of the FOTA system. Future plans should prioritize the development of user-friendly interfaces for both administrators and end-users. This involves designing intuitive dashboards with clear progress indicators and straightforward update procedures, making the update process easy to understand and execute. Implementing robust notification systems is also crucial, as it will keep users informed about the status of updates, any issues encountered, and necessary actions. These notifications can be delivered through various channels, including email, push notifications, and in-app alerts, ensuring users are always up-to-date. Providing comprehensive documentation, tutorials, and support resources will enhance the user experience by helping users navigate and utilize the FOTA system effectively. Additionally, offering multi-language support and incorporating accessibility features will ensure the system is usable by a diverse audience, further broadening its reach and adoption. By focusing on these enhancements, the FOTA project can significantly improve user satisfaction and engagement.

12.6 Integration with Other Systems

To maximize the utility and efficiency of the FOTA system, seamless integration with other enterprise systems is essential. Future development should focus on creating robust APIs that enable integration with ERP, CRM, and other enterprise management systems. This integration

can facilitate automated workflows, improve data consistency, and enhance overall operational efficiency. Ensuring cross-platform compatibility is also critical to broaden the range of devices that can be updated via FOTA. This involves supporting various operating systems, hardware platforms, and communication protocols. Extensive testing and validation should be conducted to ensure that the FOTA system operates seamlessly across different environments, providing a reliable and consistent experience for all users. Additionally, developing partnerships with key industry players can enhance the system's capabilities and reach. By focusing on these areas, the FOTA project can create a more cohesive and integrated solution that meets the diverse needs of modern enterprises.

12.7 Machine Learning and AI

Integrating machine learning and AI into the FOTA system can provide significant enhancements in predictive capabilities and operational efficiency. Future plans should include the development of predictive maintenance algorithms that analyze device performance data to forecast potential failures and schedule updates proactively. This can reduce downtime and extend the lifespan of devices. Analyzing usage patterns with AI can optimize update times, ensuring updates are performed during periods of low activity to minimize disruptions. Additionally, AI-based anomaly detection can identify unusual behavior post-update, allowing for quick corrective actions and reducing the impact of potential issues. These intelligent features can make the FOTA system more adaptive and responsive, significantly improving its efficiency and reliability. Investing in AI and machine learning capabilities can also provide valuable insights and analytics, driving continuous improvement and innovation in the FOTA project.

12.8 Regulatory Compliance and Standards

Ensuring that the FOTA system complies with relevant industry standards and regulations is essential for maintaining user trust and avoiding legal issues. Future plans should focus on aligning the FOTA processes with standards such as GDPR for data protection and ISO 27001 for information security management. This involves implementing robust data protection measures, conducting regular compliance audits, and staying updated with regulatory changes. Establishing certification programs for devices and firmware can ensure they meet quality and security standards before deployment. Regular compliance reviews should be conducted to ensure ongoing adherence to these standards, and necessary adjustments should be made to stay compliant with evolving regulations. By focusing on regulatory compliance and standards, the FOTA project can build trust with users and stakeholders, ensuring the system's long-term success and reliability.

12.9 Community and Ecosystem Building

Building a strong community and ecosystem around the FOTA project can drive continuous innovation and improvement. Encouraging contributions from the open-source community can bring fresh ideas and enhance the system's capabilities. Establishing partnerships with hardware manufacturers, IoT platform providers, and other stakeholders can create a robust ecosystem that

supports the FOTA system's growth and adoption. Hosting developer conferences, hackathons, and other community events can foster collaboration and knowledge sharing, helping to build a vibrant and engaged community. Providing clear guidelines, support, and incentives for community contributions can further enhance participation and drive the project's success. By focusing on community and ecosystem building, the FOTA project can leverage collective expertise and resources, driving continuous improvement and innovation.

12.10 Implementing Delta Updates

In the future, a primary focus for the FOTA project will be the implementation and optimization of delta updates. Delta updates, which involve transmitting only the differences between the current firmware version and the new version, are poised to revolutionize the update process. This approach will significantly reduce the amount of data that needs to be transferred, resulting in faster update times and reduced bandwidth usage. This efficiency is especially crucial as the number of connected devices continues to grow, making it imperative to manage network resources effectively. By minimizing the data load, delta updates will also enhance the reliability of the update process, reducing the likelihood of transmission errors and ensuring a smoother user experience. Additionally, implementing delta updates will support large-scale deployments by lessening the strain on network infrastructure, allowing for simultaneous updates to numerous devices without overwhelming the system. This method will also lead to shorter device downtime during updates, which is critical for maintaining continuous operation and user satisfaction. As part of our future plans, we will invest in developing robust algorithms and tools to generate and apply delta updates efficiently. This will include extensive testing to ensure compatibility across various devices and firmware versions, as well as enhancing our infrastructure to handle the new update method seamlessly. By focusing on delta updates, the FOTA project will achieve greater scalability, efficiency, and reliability, meeting the evolving demands of an increasingly connected world.

References:

https://www.avsystem.com/firmware-over-the-air/

https://www.nxp.com/applications/automotive/software-defined-vehicle/firmware-over-the-air-fota:FOTA

https://www.samsung.com/my/support/mobile-devices/what-are-the-requirements-when-upgrade-firmware-with-fota-firmware-over-the-air/

https://www.soracom.io/iot-definitions/what-is-firmware-over-the-air-fota/

https://www.embitel.com/firmware-over-the-air-fota-updates-for-iot-and-automotive-devices

https://www.faststreamtech.com/products/firmware-over-the-air/

https://www.rohde-schwarz.com/us/products/test-and-measurement/essentials-test-equipment/digital-oscilloscopes/understanding-uart_254524.html#:~:text=UART%20stands%20for%20universal%20asynchronous,and%20receive%20in%20both%20directions.

https://en.wikipedia.org/wiki/Universal_asynchronous_receiver-transmitter

https://www.techtarget.com/whatis/definition/UART-Universal-Asynchronous-Receiver-Transmitter

https://www.espressif.com/sites/default/files/documentation/esp32_datasheet_en.pdf

https://en.wikipedia.org/wiki/Advanced_Encryption_Standard

https://www.techtarget.com/searchsecurity/definition/Advanced-Encryption-Standard

https://en.wikipedia.org/wiki/Automatic_parking

https://www.ford.com.ph/engineering/active-park-assist/

https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html

https://www.st.com/en/microcontrollers-microprocessors/stm32-high-performance-mcus.html

https://flutter.dev/

https://dart.dev/

https://www.irishrail.ie/en-ie/about-us/iarnrod-eireann-services/dart-commuter

https://community.st.com/t5/stm32-mcus-embedded-software/official-stm32-bootloader/td-p/591959

https://www.reddit.com/r/vuejs/