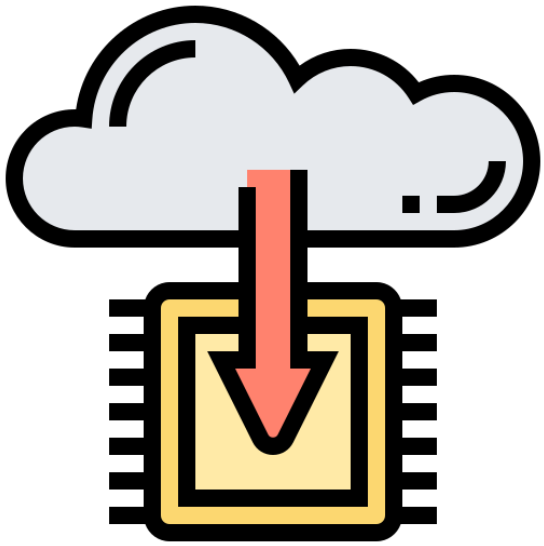
FOTA (Firmware Over The Air)



Prepared by:

Abdulrahman-Elneshwy

Abulrahman_Khalaf

Aya Ahmed Samir

Seba Ammar

John Emile

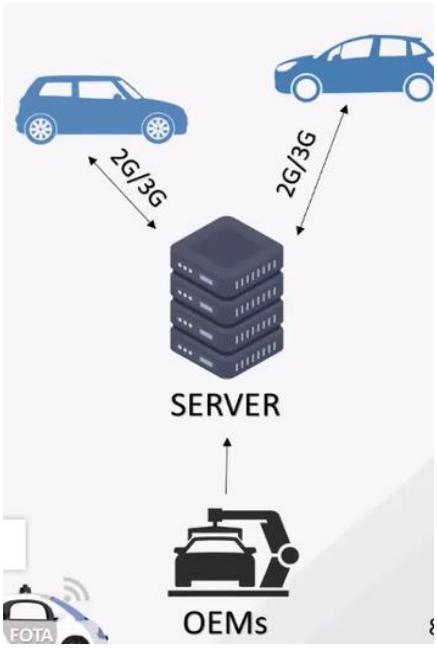
Ziad Gomaa

Smart village- Group 2

Table of Contents

What is FOTA in automotive industry?

FOTA comes from Firmware Over the Air, and is a clear trend that all OEM's will follow because it will be more and more Software in cars, and it will be more updates. FOTA is nothing but a remote software management technology where it allows OEM's to remotely update the new firmware version. OEM's has access on a server and can download the new software on the server and there is a connection either 2G or 3G between the server and the cars which can remotely update the new software.



Firmware over the air (FOTA) plays a crucial role in the automotive industry, offering several important benefits and functionalities. Some key reasons why FOTA is significant in the automotive sector:

1- Software updates: modern vehicles are equipped with numerous electronic control units (ECUs) that manage various functions, including engine control. Infotainment systems, safety features, and more. FOTA enables automakers to remotely update the software running on these ECUs. This allows for bug fixes, performance improvements, and the addition of new features without requiring the vehicle to be physically brought to a service center or dealership.
2- Cost reduction: FOTA reduces costs associated with software updates and recalls. It eliminates the need for physical service visits, reduces the number of vehicles involved in recalls, and streamlines the updates process. This translates to cost savings for both automakers and vehicle owners.
3- Product improvement: FOTA allows automakers to continuously enhance their product after they have been sold. By gathering data from vehicles in the fields, manufacturers can identify areas for improvement and release software updates to address them. This helps maintain customer satisfaction and loyalty while keeping vehicles up to date with the latest technology and features.

Design and Development of FOTA components

- The development of a bootloader to allow download of the firmware and reprogram the application device.

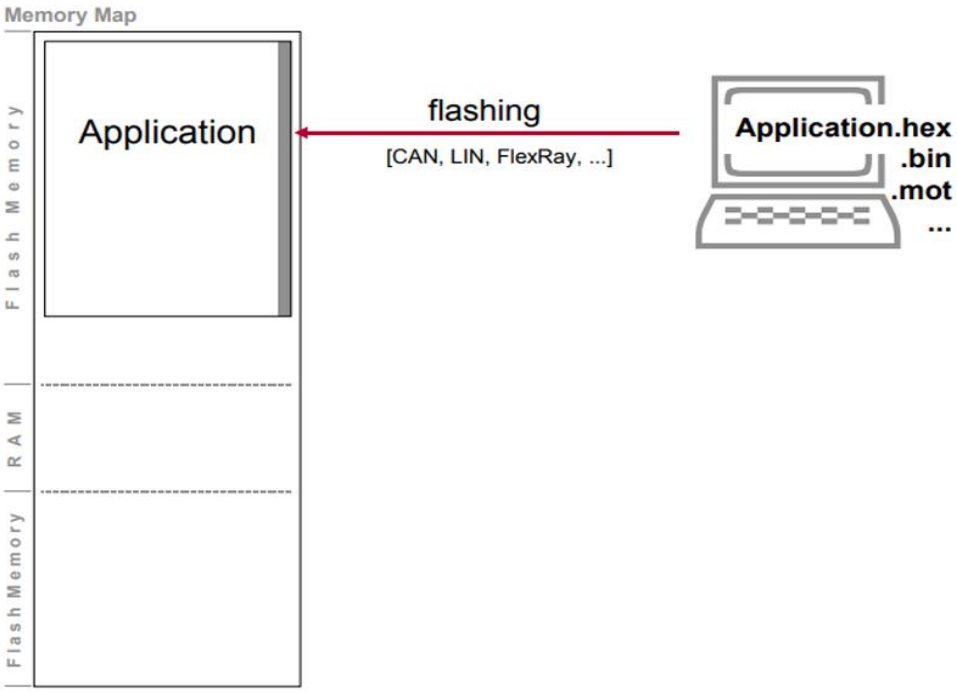
- Designing and developing the FOTA management server interface.

- Design and development of GUIs to build, schedule, deploy and manage the update package.

1. Bootloader

1.1.What is Flashing?

Flashing refers to the process of programming the firmware or software onto the non-volatile memory of a microcontroller or microprocessor-based device. During the flashing process an application (or a part of it), which must be available in hex format, is transferred into the ECU‘s memory. This transfer is done via a bus protocol like UART, CAN, LIN, FlexRay, etc. The Flash Bootloader downloads the application as a hex file to the ECU.



To transfer your application to the target platform you simply need the Flash Bootloader, a PC or alaptop and the Flash Tool. There are three options to flash an application:

1) Off Circuit programming: the chip is removed from the system to be programmed and flash an application. Where the flash driver is in the burner and there is not an internal flash driver inside the microcontroller.

2) In Circuit programming: the chip will not be removed from the system, but it still needs a debugger to be programmed and flash an application. Where there is an internal flash driver and data will be sent to it through communication protocol and the burner of debugger is used in this case as a translator.

3) In Application programming: an application (Bootloader) is used to flash a new application without needing a hardware programming tool (debugger).

1.2.What is Bootloader?

A bootloader is a program that runs on a microcontroller or other embedded device and is responsible for loading and executing the main firmware or operating system. It is typically stored in a non-volatile memory such as flash memory or EEPROM. It is an essential component of many embedded systems, allowing for easy and secure firmware updates and ensuring that devices can be kept up-to-date with the latest features and bug fixes.

The main functionality of a bootloader is to provide a way to update the firmware or software of the microcontroller without requiring a specialized programming tool or hardware and without requiring the device to be physically removed or returned to the manufacturer. This is accomplished by allowing the device to be updated over a communication channel such as UART, USB, Ethernet, or wireless connections.

The bootloader typically works by presenting a user interface over the communication channel that allows the user to initiate a firmware update, specify the new firmware file, and manage the update process. The bootloader will then erase the existing firmware and program the new firmware into the microcontroller's memory. Once the new firmware is programmed, the bootloader will verify its integrity and then transfer execution to the new firmware. Some bootloaders may also include additional features such as security mechanisms to ensure that only authorized firmware updates are installed, or recovery mechanisms to allow the device to be restored to a known good state in the event of a failed firmware update.

Note that The Bootloader is a stand-alone program. It is compiled, linked, and downloaded to the ECU separately from your application. The Bootloader and your application never run simultaneously and it has his own startup code so there is no comparison between the bootloader and the startup code.

In our project we've used Bootloader over **UART** as the bootloader receives record by record through UART and it is implemented through **three versions**, Stm32F401CC microcontroller was used and it has the flash memory organization as shown in the following figure:

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

1.3. Bootloader versions implemented in the project

- **Version 1**, it was the simplest version where the flash and the run mechanism, both were in sector 1 according to the algorithm of the bootloader, where the bootloader runs and receives record by record , at the first time it erases the area in sector 1 to enable the bootloader to flash , and then it flashes record by record and after it finishes , the code jumps to the application in sector 1.

- **Version 2**, a new feature is added which is the two banks, where there is a sector for flashing and a sector for running which is responsible for running the application because if there is error in the updated software, the running application still runs as it is and will not be erased. So, sector 2 is responsible for flashing the new hex file and sector 1 is for running the application. Data will be moved from sector 2 to sector 1 when a restart occurs.

- **Version 3**, another feature is implemented where the target was to increase the application size so sectors from 1 to 4 is responsible for running the application and sector 5 is responsible for flashing the hex file, when the hex file has no errors, there is a jump to sector 1 to run the application. Also, in this version there is a very important feature added which is the check sum, to check that the updated software has no errors and ready to be moved to the sector responsible for running and erase the old application.

1.4. Bootloader features

1- The use of 2 separate banks to hold the new updated version in a separate bank from which the application executes, that gives flexibility and stability as we keep the new updated version from OEM until the code is checked using check sum mechanism. The 2 banks are divided as follows:
Bank 1: Sector 1-2-3-4 and this bank is where the application gets executed.
Bank 2: Sector 5 that receives the update and then copied to Bank 1 if the code is functional or deleted if the code is corrupted.

2- The support of high size applications that can be done by using multiple sectors to run the application from instead of using only one. Though there's a limitation that the size of application doesn't exceed 112 KB (size of bank 1).

3- Error checking using check sum mechanism to determine the integrity of the data transmitted over a network. The hex file is sent as address + data + checksum result thus the receiver can calculate the check sum and compare it with that embedded in the frame. This ensures that the system is always working even if a corrupted update is sent from the OEM. And also if the updated software has an error, the running application will not be erased according to the two banks concept where the flashing is done in sector and the execution of the application is done in another sector.

1.5. Flash memory

The flash memory requires a high power to write on it, so a flash driver is used to write on the flash memory. So, a flash driver is implemented to write on the flash memory and also to erase data. Flash memory is organized into pages, and each page must be erased before it can be written to again. The erase operation clears all bits in the page to 1, which allows new data to be written to the page without interference from the old data. The write operation then sets specific bits in the page to 0 to store the new data. Therefore, before writing new data to a page, the page must be erased to ensure that there is no old data remaining. The FPEC provides a page erase operation that can be used to erase a specific Flash memory page before new data is written to it. The Flash memory is divided into pages, and each page typically contains a fixed number of bytes or words. The programming operation writes one or more words of data to a specific address in the Flash memory, which can be a single byte or an entire page, depending on the size of the data being written.

2. Raspberry Pi &GUI

- Hardware **and Software Requirements**

  - Raspberry Pi 3B+
  - Micro-controller
  - GUI Interface
  - Communication Protocols
  - Internet Connection

- System **Architecture**

  - Raspberry Pi Configuration
  - Micro-controller Integration
  - GUI Design
  - Update Server

- 4 FOTA **Workflow**

  - Update Trigger
  - Downloading Firmware
  - User Confirmation
  - Firmware Transfer
  - Installation Confirmation

- 5 Security **Measures**

  - Secure Communication
  - Authentication and Authorization
  - Data Integrity

We worked on Raspberry Pi 3B+, using Linux image we introduced a program that manages the downloading and installation of firmware on the microcontroller.'

2.1. Raspberry Pi 3B+

The Raspberry Pi 3B+ serves as the central hub, coordinating communication between the user interface and the microcontroller we used it for.
The Raspberry Pi 3B+ is a versatile and powerful single-board computer that can be used for a wide variety of tasks.

Affordable price: The Raspberry Pi 3B+ is very affordable, making it a great entry point into single-board computing.

Compact size: The Raspberry Pi 3B+ is compact, making it ideal for small projects and embedded systems.

Powerful processor: The Raspberry Pi 3B+ features a quad-core 1.4GHz ARM Cortex-A53 processor, which is powerful enough to run most applications and workloads.

Versatile I/O: The Raspberry Pi 3B+ has a wide variety of I/O ports, including HDMI, USB, Ethernet, GPIO, and CSI/DSI. This makes it easy to connect a wide variety of devices and peripherals.

Wide range of software support: The Raspberry Pi 3B+ is supported by a wide range of software, including operating systems, programming languages, and applications. This makes it a great platform for our project.

2.2. Micro-controller

The micro-controller in the RC car must be compatible with the FOTA system, supporting the necessary communication protocols. And here are some features of STM32F104

High Performance:

- The STM32F401RC is built on the ARM Cortex-M4 core, providing high processing power for embedded applications.

Clock Speed:

- The microcontroller operates at a high clock speed, allowing for rapid execution of instructions and efficient performance.

Rich Peripheral Set:

- It comes equipped with a wide range of peripherals, including GPIO, USART, SPI, I2C, ADC, and more, providing flexibility for various applications.

Memory Options:

- The STM32F401RC offers different memory options, including Flash memory for program storage and SRAM for data storage, catering to the diverse needs of embedded systems.

Advanced Connectivity:

- It supports various communication interfaces such as USART, SPI, and I2C, enabling seamless integration with other devices and peripherals.

Analog-to-Digital Converter (ADC):

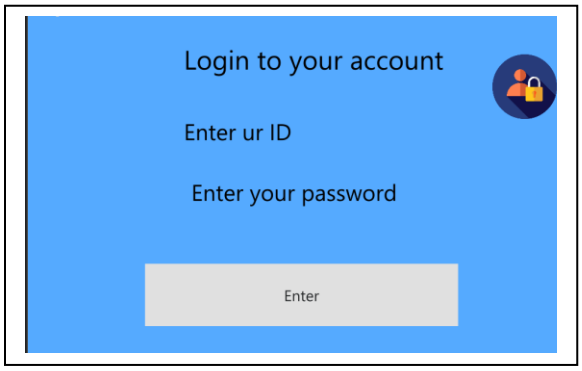
- Integrated high-resolution ADC allows for precise analog signal measurements, crucial for applications requiring accurate sensor readings.

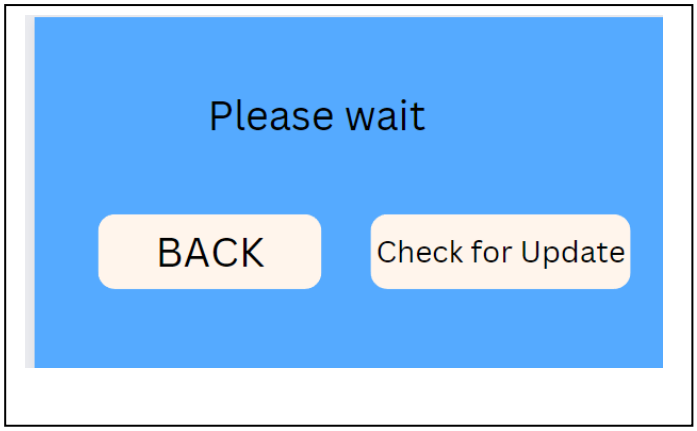
Rich Set of Timers:

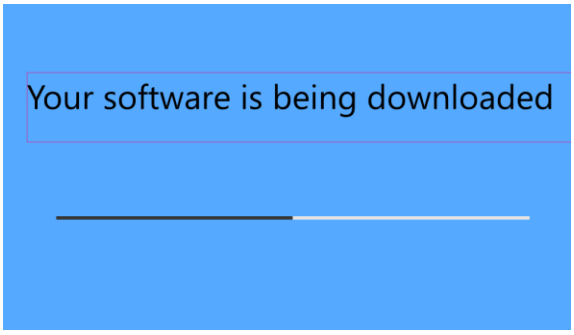
- Multiple timers with various operating modes are available, facilitating tasks like PWM generation, timekeeping, and event counting.

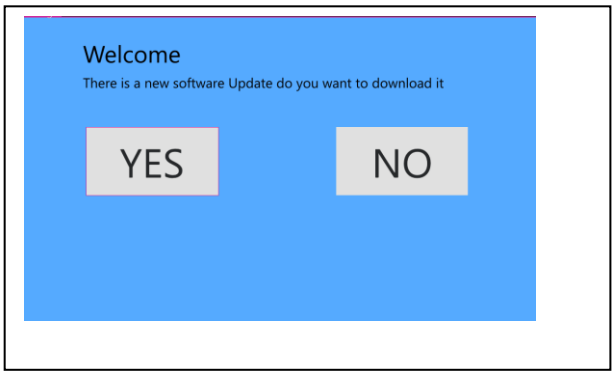
2.3 GUI Interface

The GUI is crucial for user interaction, facilitating update confirmation and providing progress feedback. We used the QT platform to build the GUI and QML language combined with python









Architecture section, considering the context of using a Linux image for the Raspberry Pi, systemd for service management, and GitHub as the update server:

2.4 Raspberry Pi Configuration

The Raspberry Pi needs specific configurations to function as the central hub for managing FOTA. This involves setting up networking, ensuring access to the update server, and configuring systemd for service management.

- **Example:** The Raspberry Pi is configured with a static IP address on the local network, ensuring consistent communication with the update server on GitHub. The systemd service, named "fota_manager," is configured to automatically start on boot, ensuring seamless FOTA functionality.

2.5 Micro-controller Integration

Integrating the micro-controller with the FOTA system involves ensuring compatibility, setting up communication protocols, and adapting the micro-controller firmware to support remote updates.

- **Example:** The micro-controller in the RC car is programmed to communicate over Bluetooth Low Energy (BLE) to receive firmware updates. The FOTA system uses standardized communication protocols to establish a secure connection with the micro-controller, ensuring reliable data transfer during firmware updates.

2.6 GUI Design

The graphical user interface (GUI) is a crucial element for user interaction and confirmation of firmware updates. This involves designing a user-friendly interface that communicates the update status and prompts user actions.

- **Example:** The FOTA GUI presents a clear and intuitive interface on the connected display, displaying real-time update progress, and prompting the user with a dialog box for firmware confirmation. The design incorporates visual indicators such as progress bars and status messages for a user-friendly experience.

Update Server

The update server is responsible for hosting the firmware updates and facilitating download requests from the Raspberry Pi. GitHub is used as the hosting platform for firmware files.

- **Example:** The firmware updates are stored in a designated repository on GitHub. The FOTA system, running on the Raspberry Pi, utilizes Git commands to fetch the latest firmware files from the GitHub repository. The GitHub repository serves as a centralized and version-controlled location for managing firmware updates.

Update Trigger

The update trigger is the event or condition that initiates the firmware update process. This could be a user-initiated action, a scheduled check for updates, or a system-triggered event. For example:

- The user opens the FOTA application on their mobile device and selects the "Check for Updates" button.

Downloading Firmware

After the update is triggered, the system needs to download the new firmware from a remote server. This involves establishing a secure connection and fetching the latest firmware package.

- The FOTA system connects to the designated update server over HTTP and downloads the firmware update file named "RC_Update"

User Confirmation

Once the new firmware is downloaded, the user is prompted to confirm whether they want to proceed with the installation. This step ensures that the user is aware of the update and agrees to apply it.

- The FOTA GUI displays a message stating, "A new firmware update is available. Do you want to install RC Car Firmware v2.0?" with options for "Yes" and "No."

Firmware Transfer

Upon user confirmation, the FOTA system transfers the firmware update to the micro-controller in the RC car. This step involves secure and reliable transmission of the firmware file.

- Upon user confirmation, the Raspberry Pi establishes a secure SPI connection with the RC car's micro-controller and transfers the firmware update file.

Installation Confirmation

After the firmware transfer is completed, the FOTA system confirms the successful installation of the new firmware. This step provides feedback to the user, ensuring transparency in the update process.

Secure Communication

Ensuring secure communication is crucial for preventing unauthorized access and tampering during the transmission of firmware updates.

- The FOTA system uses secure protocols as HTTPS for communication between the Raspberry Pi and the GitHub update server. All data exchanged, including firmware files and update statuses, is encrypted, preventing interception and tampering.

Authentication and Authorization

Authentication and authorization mechanisms are implemented to verify the legitimacy of the entities involved in the FOTA process.

- Before allowing access to firmware updates, the GitHub repository requires authentication using secure credentials (e.g., OAuth tokens). Additionally, the micro-controller in the RC car verifies the authenticity of the received firmware update using digital signatures, ensuring that only authorized updates are accepted.

Data Integrity

Maintaining data integrity ensures that the firmware updates remain unchanged and reliable throughout the entire update process.

- The FOTA system employs cryptographic hash functions to generate checksums for the firmware files. Before installation, the micro-controller verifies the integrity of the received firmware by comparing the calculated checksum with the expected value. Any discrepancies trigger an alert, preventing the installation of corrupted or tampered firmware.

3. Server & Security:

This part provides an overview of the Fernet encryption script designed for the Firmware Over-The-Air (FOTA) project. The script utilizes the Fernet encryption algorithm to secure data stored in a source file by encrypting it and saving the encrypted data in a destination file.

Purpose

The primary purpose of this script is to enhance the security of sensitive data within the FOTA project by employing the Fernet encryption algorithm. The script enables users to easily encrypt the contents of a file before transmission or storage, ensuring the confidentiality of firmware updates and sensitive information.

Encryption Overview

The Fernet encryption algorithm serves as the cornerstone of the security measures implemented in the FOTA project. Fernet, recognized for its efficacy and simplicity, operates as a symmetric encryption algorithm. Symmetric encryption involves the use of a single key for both encryption and decryption processes. This characteristic aligns seamlessly with the security objectives of the FOTA project.

Fernet Key Management

Effectively managing Fernet keys is pivotal for maintaining a secure and reliable encryption infrastructure within the FOTA project. This section provides detailed insights into key generation, secure storage guidelines, and key rotation practices to ensure a resilient key management process.

Key Generation

The process of key generation serves as the foundation for a secure Fernet encryption framework. To uphold stringent security standards, it is imperative to employ best practices during key generation. This procedure, ideally executed during the project's initialization phase, should be a one-time event. The generated Fernet key, derived from cryptographically secure random number generators and conforming to recommended key length guidelines, must be treated with the utmost confidentiality.

Key Storage

The secure storage of Fernet keys is paramount to prevent unauthorized access and maintain the overall integrity of the encryption system. This section outlines guidelines for securely storing the Fernet key, emphasizing the implementation of robust access controls, encryption of stored keys, and exploration of secure key management solutions. Treating the Fernet key as a sensitive asset and employing measures to protect against potential breaches or unauthorized disclosures is essential.

Key Rotation

Proactive key rotation practices are employed to enhance the resilience of the encryption system over time. Periodically rotating encryption keys is a strategic measure that mitigates potential risks associated with prolonged key usage. This section establishes clear procedures for key rotation, defining the frequency and steps involved inhe process. By adopting key rotation practices, the project ensures that, even in the event of a compromised key, the exposure window is limited, fortifying the overall security posture of the FOTA project.

In summary, Fernet Key Management encompasses key generation, secure storage guidelines, and key rotation practices. By adhering to these best practices, the FOTA project can establish and maintain a robust key management framework, vital for ensuring the ongoing confidentiality and integrity of encrypted data during Firmware Over-The-Air updates.

Encryption Process

The Encryption Process section provides a comprehensive guide to the steps involved in encrypting data within the FOTA project, leveraging the robust Fernet encryption algorithm. Additionally, this section delves into an in-depth explanation of the Fernet encryption algorithm, shedding light on its security features and its appropriateness for the project.

Steps for Encryption

Effectively securing data before transmission is a critical aspect of the FOTA project. The following steps outline the meticulous process of encrypting data, supported by illustrative code snippets for practical implementation:

Step 1: Fernet Key Initialization

Initiate the Fernet key, ensuring adherence to best practices for secure key generation.

Step 2: Fernet Cipher Initialization

Create a Fernet cipher object using the generated key.

Step 3: Read Data for Encryption

Retrieve the data to be encrypted from the source file

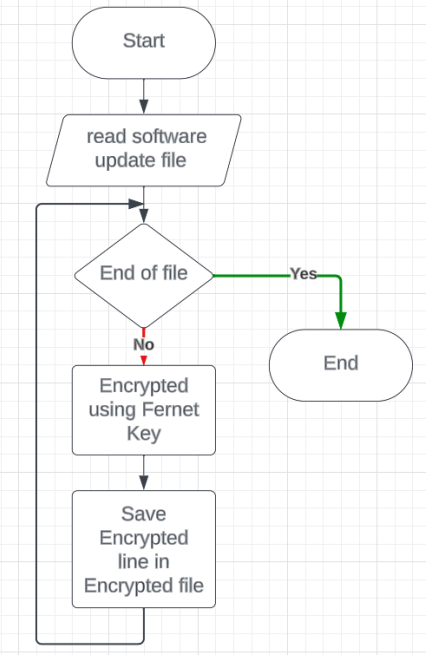
Step 4: Data Encryption

Leverage the Fernet cipher to encrypt the data securely.

Step 5: Write Encrypted Data

Persist the encrypted data to the destination file.

(illustrate follow-chart)



Encryption Algorithm

A fundamental understanding of the Fernet encryption algorithm is crucial for implementing a secure encryption process within the FOTA project. This subsection provides a nuanced exploration of the algorithm, emphasizing its robust security features and its alignment with the project's requirements.

Fernet Encryption Algorithm Overview

The Fernet encryption algorithm operates as a symmetric encryption model, utilizing a single secret key for both encryption and decryption processes. Key features include:

- Symmetry: Utilizing a single secret key for both encryption and decryption.
- Secure Key Management: Incorporating secure key generation practices to bolster the overall security framework.
- Cryptography Standards: Adhering to established cryptographic standards to provide a resilient defense against unauthorized access and data breaches.
- Efficiency: Tailored for resource-constrained environments, Fernet ensures efficient encryption and decryption processes without imposing undue computational or storage burdens.

In conclusion, the Encryption Process section serves as a comprehensive resource for users and developers, guiding them through the intricacies of encrypting data with the Fernet encryption algorithm. This knowledge empowers the FOTA project with a secure and efficient encryption mechanism for safeguarding data during Firmware Over-The-Air updates.

Decryption Process

The Decryption Process section provides an in-depth exploration of the steps involved in decrypting received data within the FOTA project. This section encompasses detailed steps for decryption, supported by code snippets for practical implementation. Additionally, it delves into an explanation of the Fernet decryption algorithm, shedding light on its security features and appropriateness for the project.

Steps for Decryption

Decrypting received data is a crucial aspect of the FOTA project's functionality. The following steps outline the meticulous process of decrypting data, complemented by illustrative code snippets for implementation:

Step 1: Fernet Key Initialization

Initialize the Fernet key, ensuring it corresponds to the key used during encryption.

Step 2: Fernet Cipher Initialization

Create a Fernet cipher object using the initialized key.

Step 3: Read Encrypted Data

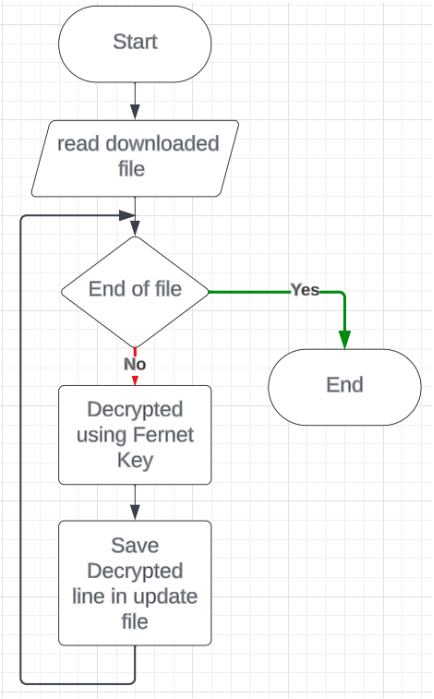
Retrieve the encrypted data from the received file.

Step 4: Data Decryption

Utilize the Fernet cipher to decrypt the received data.

Step 5: Handle Decrypted Data

Process or utilize the decrypted data as needed within the FOTA project.



Decryption Algorithm

A nuanced understanding of the Fernet decryption algorithm is essential for the secure implementation of the decryption process within the FOTA project. This subsection provides an in-depth exploration of the algorithm, emphasizing its robust security features and alignment with the project's requirements.

Fernet Decryption Algorithm Overview

The Fernet decryption algorithm mirrors its encryption counterpart, operating as a symmetric encryption model with a single secret key for both encryption and decryption processes. Key features include:

- Symmetry: Utilizing the same secret key for both encryption and decryption tasks.
- Secure Key Management: Consistent with secure key generation practices to maintain a robust security posture.
- Cryptography Standards: Adhering to established cryptographic standards, ensuring a resilient defense against unauthorized access and data breaches.
- Efficiency: Aligned with the efficiency required for resource-constrained environments, Fernet facilitates seamless decryption without imposing significant computational or storage burdens.

In conclusion, the Decryption Process section serves as a comprehensive guide for users and developers, elucidating the steps for decrypting received data within the FOTA project. The accompanying explanation of the Fernet decryption algorithm empowers the project with the knowledge needed for secure and efficient data decryption during Firmware Over-The-Air updates.

4. GitHub Repository Integration

The integration of a GitHub repository plays a pivotal role in the seamless execution of firmware updates in our FOTA project. GitHub serves as the centralized platform for storing, versioning, and distributing firmware updates, providing a reliable and accessible repository for our system. Here are key aspects of the GitHub integration:

4.1. Repository Structure

The GitHub repository is organized with a structured hierarchy to efficiently manage firmware versions and related assets. Each firmware version is tagged, enabling easy retrieval of specific releases. The repository structure ensures clarity and traceability, essential for maintaining a history of updates and facilitating debugging if issues arise.

4.2 Version Control

Utilizing Git's version control capabilities ensures that each firmware update is tracked, allowing us to roll back to previous versions if needed. This version control mechanism safeguards against unintended changes and supports collaborative development. Developers can easily contribute to firmware updates by branching, making modifications, and submitting pull requests for review and integration.

4.3 Release Management

GitHub's release management features facilitate the organized publication of firmware updates. Each release on GitHub corresponds to a specific version of the firmware, accompanied by release notes detailing the changes and improvements. Users and developers can effortlessly navigate through releases, gaining insights into the evolution of the firmware over time.

4.4. Security Measures

To enhance security, access to the GitHub repository is controlled through authentication mechanisms. OAuth tokens or personal access tokens are utilized to authenticate the Raspberry Pi when fetching updates. Additionally, the repository's settings enforce secure connections (HTTPS), preventing unauthorized entities from intercepting or altering firmware updates during transmission.

4.5 Collaboration and Contribution

GitHub's collaborative features foster teamwork in the development and improvement of firmware updates. Developers can collaborate on the same codebase, contributing enhancements and fixes through pull requests. The transparency provided by GitHub's collaboration tools ensures effective communication among team members, leading to a more robust and reliable FOTA system.

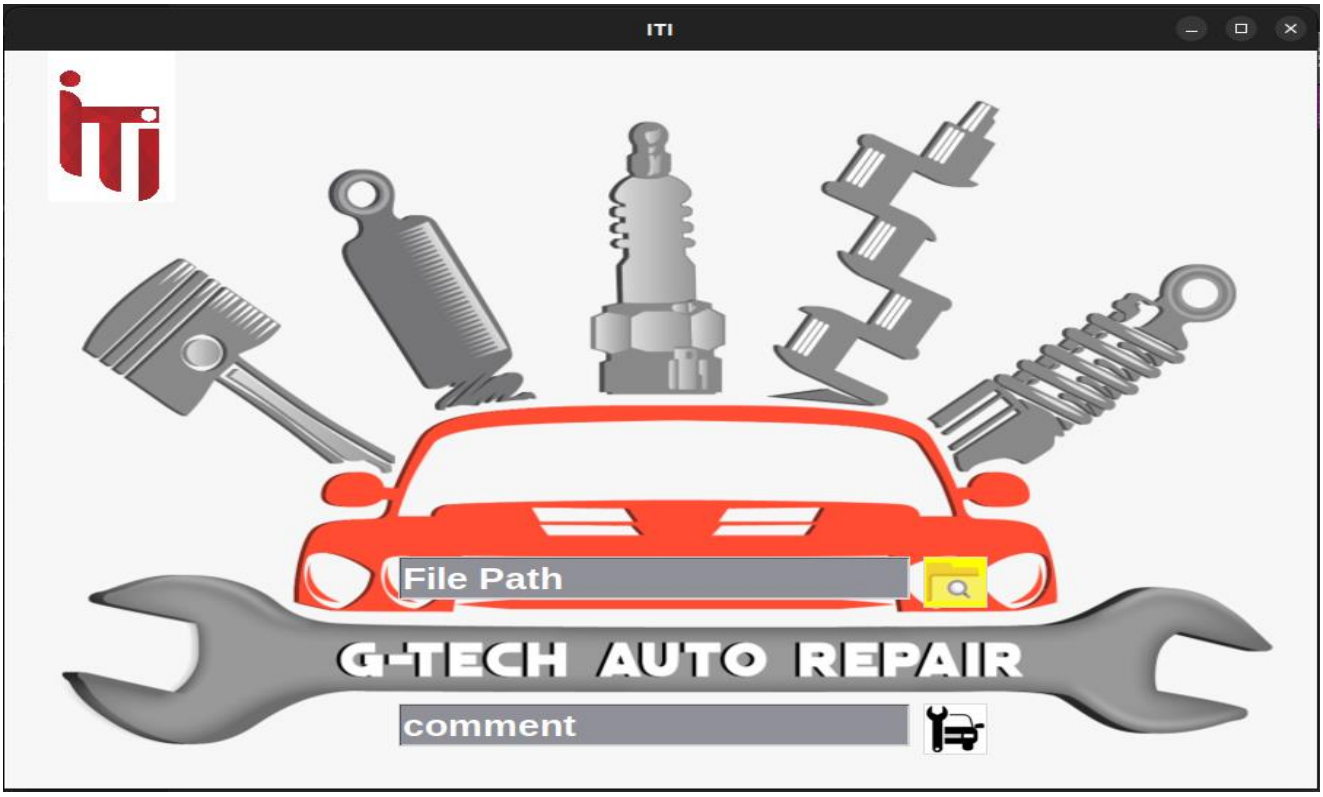
4.6 Automated Workflows

GitHub Actions are leveraged to automate repetitive tasks, such as building firmware binaries, running tests, and deploying updates. Continuous Integration (CI) pipelines ensure that changes introduced to the codebase pass predefined tests before being merged into the main branch. This automation enhances the reliability and quality of firmware updates.

the GitHub repository integration is a cornerstone of our FOTA project, providing a robust platform for version control, release management, security, collaboration, and automation. Leveraging GitHub's features enhances the efficiency, reliability, and security of the firmware update process, contributing to the overall success of the FOTA system.

Developing Tools:

All this Feature integrate in Tool to Simplest updating Sequence for Developer using the Following GUI.



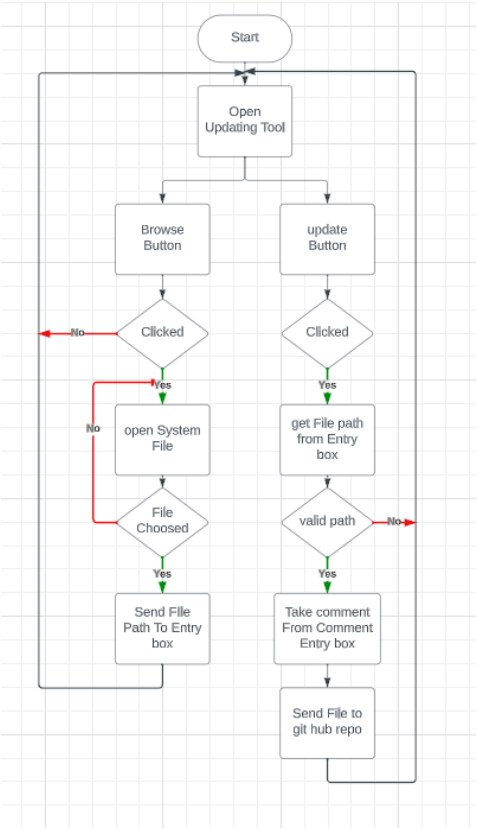
Work Flow:

On Opening Tool we get to Entry box First one is option

To Browse into sytem file and choose the Required File to be update, Second one is to update File system after Commentig the update request.

The tool take the Required File Directory then Encrypte File content and save it in anther File called “Encrypted File ”

Then start upload the File to github Repo using Github Token to have access



5. Application

5.1 Controlling the Car with the Mobile Application

This project is designed to provide users with an intuitive mobile application interface (mode1) for seamless car control, complemented by an autonomous line-following capability utilizing infrared (IR) sensors (mode2). The communication protocol employed between the Bluetooth module (HC-06) and the Microcontroller (STM32F401CC) is UART. The hardware configuration ensures connectivity with RX2 (Pin A3) on the Microcontroller linked to TX on the HC-06, and TX2 (Pin A2) on the Microcontroller connected to RX on the HC-06. Additionally, the grounding of both modules is interconnected for a common reference.

5.1.1 Initializing UART

To initiate communication, UART is initialized using MUART_voidInit();. This sets the stage for the interaction between the mobile application and the Microcontroller.

5.1.2 Receiving Mode Selection

The user communicates the desired mode through the mobile application. The Microcontroller, using the MUART_Recieve(&Local_u8Mode,UART_2); function, awaits this mode signal. If the received mode is 'X', it signifies the user's selection of mode1, granting them control over the car's movements.

5.1.3 Mobile Application Commands

Once in mode 1, the user has the capability to direct the car's movements through the mobile application. The following commands are recognized:

'F': Move the car forward.

'G': Reverse the car.

'R': Turn the car to the right.

'L': Turn the car to the left.

'S': Stop the car.

5.1.4 Mode Switching

Additionally, the user has the option to switch to mode2 by sending 'Y', transitioning from manual control to autonomous line-following.

5.2 Autonomous Line-Following with IR Sensors

In mode 2, the system transitions to autonomous line-following functionality utilizing two IR sensors. These sensors continuously monitor the surface for variations in reflectivity. The line-following algorithm interprets the sensor readings, allowing the car to make intelligent decisions regarding its path.

5.2.1. IR Sensor Integration

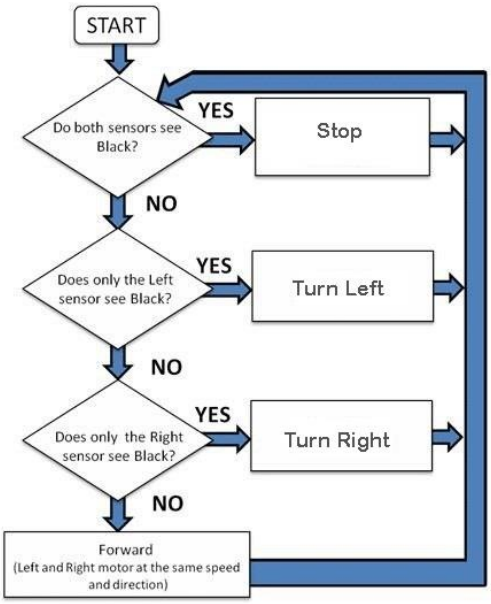
- The Infrared (IR) sensors are employed to detect the contrast between the surface colours, distinguishing between black (IR_BLACK) and white (IR_WHITE).
- The left and right IR sensors' states are continuously checked using HIR_LeftState(); and HIR_RightState(); functions.

5.2.2. Line-Following Logic

- If both sensors detect a white surface (IR_WHITE), the car proceeds forward, maintaining its current trajectory.
- In the event of the left sensor detecting a black line (IR_BLACK) while the right sensor detects white, the car executes a left turn.
- Conversely, if the right sensor detects a black line while the left sensor detects white, the car executes a right turn.
- When both sensors detect a black line, the car comes to a halt, ensuring it stays on the designated path.

5.2.3. Motor Control

The DC motor's behaviour is adjusted based on the line-following algorithm's decisions, enabling the car to autonomously navigate along the predefined route.



5.3 Implementing Application using RTOS.

In advancing the project's capabilities, the seamless integration of FreeRTOS emerges as a pivotal augmentation, elevating the system to a sophisticated multitasking environment. This transformation empowers users to control a car via a mobile application (mode1) and implement autonomous line-following using infrared (IR) sensors (mode2). The integration of FreeRTOS introduces a structured approach to task management, allowing for parallel execution within the embedded systems' constrained environment.

5.3.1. Tasks:

1. TaskModeOption:
    - Responsibility: This task is tasked with receiving the mode from the mobile application and giving the corresponding semaphore.
    - Functionality:
        - Manages the transition between manual and line follower modes.
        - Enables synchronization using semaphores for proper task execution flow.
2. ManualMode:
    - Responsibility: Executing in manual mode, this task controls the car based on user commands received through UART.
    - Functionality:
        - Manages the transition to line follower mode, ensuring a smooth switch between operational modes.
        - Utilizes synchronization semaphore (xsemaphore_1) to coordinate tasks and prevent conflicts.
3. IRMode_Selection:
    - Responsibility: Monitors IR sensors and takes actions based on sensor readings, controlling the car's behaviour in line follower mode.
    - Functionality:
        - Ensures timely response to IR sensor inputs for effective line-following logic.
        - Operates concurrently with other tasks, enhancing the project's real-time capabilities.
4. IRMode_Left:
    - Responsibility: Executing in line follower mode, this task turns the car left based on sensor inputs.
    - Functionality:
        - Ensures smooth left turns while the car is in autonomous line-following mode.
        - Utilizes synchronization semaphore (xsemaphore_3) for coordination with other tasks.
        -

5.3.2. Semaphores:

1. xsemaphore_1:
    - Usage: Used for synchronization in manual mode.
    - Purpose: Ensures a controlled execution flow, avoiding conflicts between tasks operating in manual mode.
2. xsemaphore_2:

- Usage: Used for synchronization in line follower mode.
- Purpose: Coordinates tasks to operate seamlessly in line follower mode, preventing task interference.

3. xsemaphore_3:
    - Usage: Used for synchronization in the IRMode_Left task.
    - Purpose: Coordinates the left-turning behavior in line follower mode, preventing conflicts with other tasks.

This strategic implementation of tasks and semaphores with FreeRTOS ensures the project's enhanced efficiency, responsiveness, and adaptability in diverse operating modes. The structured task management and synchronization mechanisms contribute to a robust and well-coordinated embedded system.

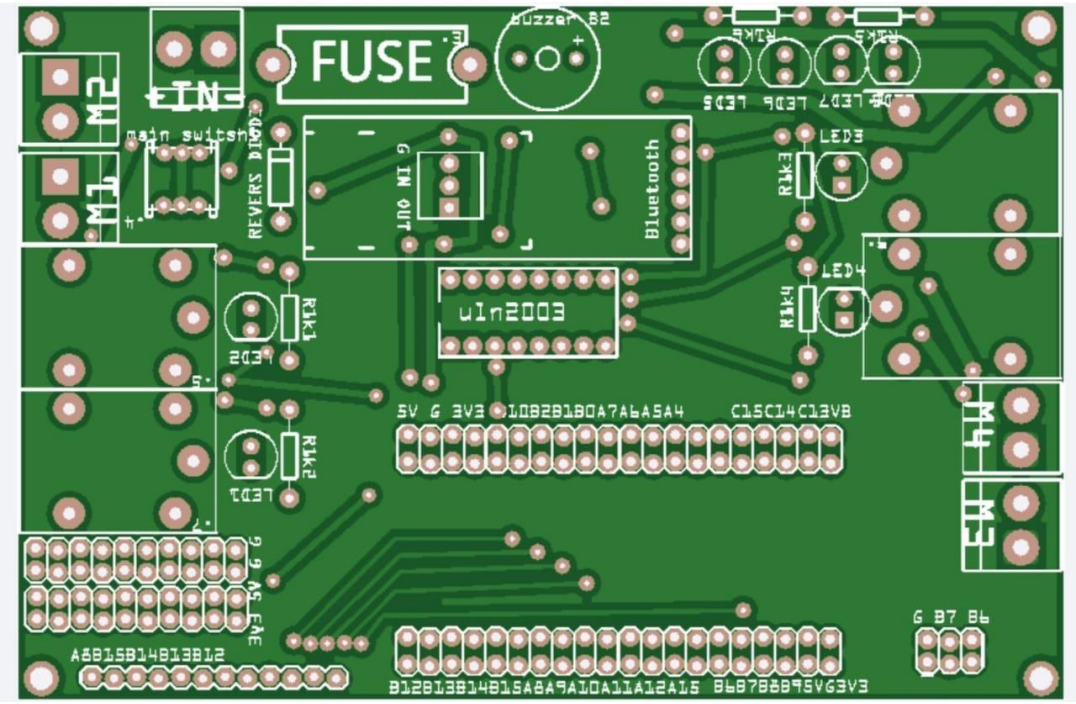
6. Hardware Implementation

Incorporating Firmware Over-the-Air (FOTA) capability enhances the project's flexibility by allowing wireless updates to the system. The Raspberry Pi, along with a touch screen interface, serves as a crucial element in facilitating user interactions for firmware updates. Here is an extended breakdown of the key hardware components :

1. **Microcontroller (STM32F401CC):**
    - Continues to be the primary controller responsible for the core project functionalities.
    - Executes the updated firmware received through FOTA.
2. **Bluetooth Module (HC-06):**
    - Enables wireless communication between the microcontroller and a mobile application.
    - Facilitates remote control of the car through Bluetooth communication.
3. **5 Relays:**
    - Used for switching high-voltage components, allowing for precise control and safety measures.
4. **3 Infrared (IR) Sensors:**
    - Crucial for autonomous line-following functionality.
5. **2 Lithium Batteries:**
    - Power source for the mobile car.
6. **Buzzer:**
    - Provides audio feedback or alerts during the update process.

7. **4 DC Motors:**
    - Drives the movement of the car.
8. **Car Body:**
    - Physical structure designed to house and support components.
9. **Touch Screen Interface:**
    - Facilitates user interaction for initiating and monitoring FOTA updates.
10. **Raspberry Pi 3B+:**
    - Acts as a secondary processing unit for managing FOTA updates.
    - Hosts the interface for user interactions.

The following figure shows a PCB fabrication which was designed by fritzing.



11. **PCB (Printed Circuit Board):**
    - Organized platform for mounting and connecting electronic components.
12. **Fuse:**
    - Protects the circuit from electrical faults.
13. **5V Voltage Regulator L7905:**
    - Stabilizes and regulates voltage.
14. **ULN2003:**
    - Used for driving high-power components.

15. **8 LEDs:**
    - Visual indicators for system status.
16. **Pin Headers:**
    - Facilitate easy interfacing and connectivity.
17. **5 Rosettes:**
    - Connectors for reliable electrical connections.
18. **Power Switch:**
    - Allows the user to control the system's power supply.

Additional Components for FOTA:

19. **Wi-Fi Module:**
    - Enables wireless communication between the Raspberry Pi and external servers for firmware updates.
20. **Storage (SD Card/Flash Memory):**
    - Provides additional storage for firmware updates.
21. **Firmware Update Button:**
    - A dedicated button on the touch screen interface to initiate firmware updates.
22. **Secure Authentication Components:**
    - Components ensuring the security of FOTA updates, preventing unauthorized access.
23. **Update Progress Bar:**
    - Visual representation on the touch screen to inform the user about the progress of the firmware update.

By incorporating these additional components, the project becomes equipped to seamlessly implement FOTA updates, allowing users to enhance their system's functionality over time without physical intervention. The Raspberry Pi and touch screen serve as pivotal components in this user-friendly FOTA implementation, providing a straightforward and interactive experience for users during the update process.