Benha University
Shoubra Faculty of Engineering

SMART FIRMWARE OVER THE AIR SYSTEM

BY

Abdulrhman Mohamed Abdelhamed
Ahmed Khaled Ahmed Zaky
Mazen Khaled Salah
Sayed Reda El Sayed
Yousef Mohamed Anwar

COMPUTER ENGINEERING DEPARTMENT

SUPERVISED BY

Dr. May Ahmed

Year: 2023 / 2024

Contents

Chapter 1: Introduction

1.1 Firmware Over the Air

1.1.1 What is the FOTA system?

Firmware Over-The-Air (FOTA) allows for the remote updating of firmware, enabling manufacturers to seamlessly deploy improvements, bug fixes, and security patches to devices in the field without requiring physical intervention. FOTA streamlines the update process, reducing the need for costly and time-consuming manual interventions, and contributes to the overall efficiency and adaptability of electronic devices.

One of the key advantages of FOTA lies in its ability to extend the lifespan of devices. By providing a mechanism for manufacturers to deliver timely updates. The system ensures that devices remain relevant and secure over time. FOTA not only benefits manufacturers by allowing them to respond swiftly to emerging threats or opportunities but also enhances the user experience by offering a more reliable, secure, and feature-rich environment for connected devices ***as in Figure 1***.

Figure 1. Vehicles get updates.

1.1.2 Importance of FOTA system

1. **Remote Maintenance**
   Devices or Vehicles are often deployed in locations that may be hard to reach or in environments where physical access is limited. FOTA allows manufacturers to perform maintenance and updates without requiring direct access to the devices, reducing operational challenges and costs.

2. **Continuous Improvement**
   FOTA enables manufacturers to continuously improve and refine the functionality of Devices or Vehicles. This is essential for addressing bugs, enhancing features, and ensuring that devices stay current with the latest standards and technologies ***as in figure 2.***

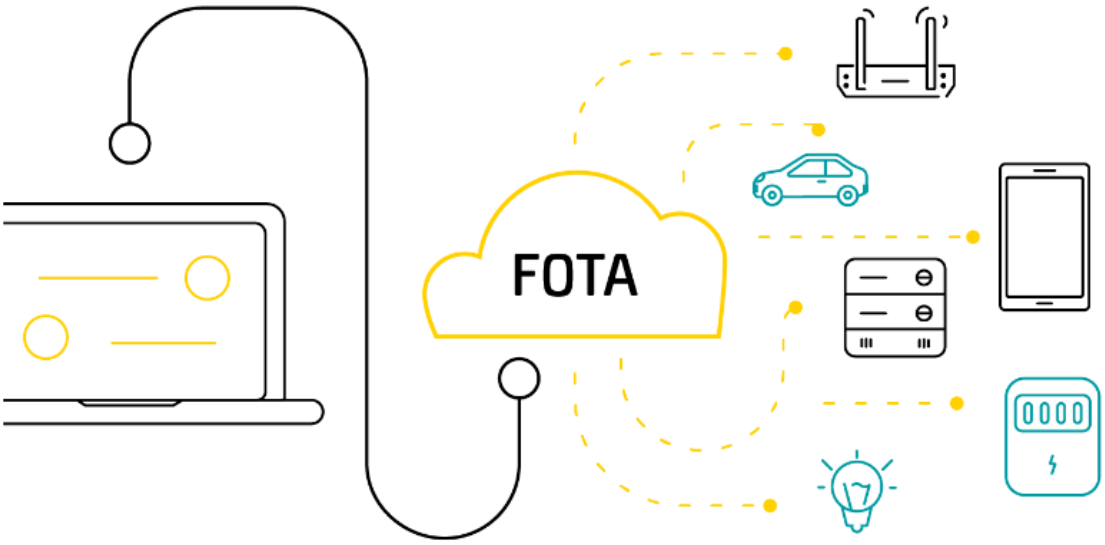


Figure 2. Devices that can connect to FOTA system for Improve the functionality.

3. **Cost-Efficiency**
   Conducting updates in Vehicles traditionally involved significant costs associated with manual intervention, on-site visits, or recalls. FOTA reduces these costs by allowing updates to be delivered remotely, minimizing the need for physical maintenance.
4. **Extended Product Lifecycle(reliable)**
   FOTA helps extend the useful life of Vehicles by enabling ongoing software support and updates.

5. **Reduced Downtime and Service Disruption**:
   Vehicles are often critical components of larger systems, and downtime can have significant consequences. FOTA updates can be applied without disrupting the normal operation of Vehicles, ensuring continuous functionality.

1.1.3 The main features of a FOTA system typically include

1. **Wireless Updates:**
   FOTA systems allow devices to receive and install updates wirelessly, eliminating the need for physical connections like USB cables. This is particularly useful for devices that are deployed remotely or are not easily accessible ***as in figure (3).***

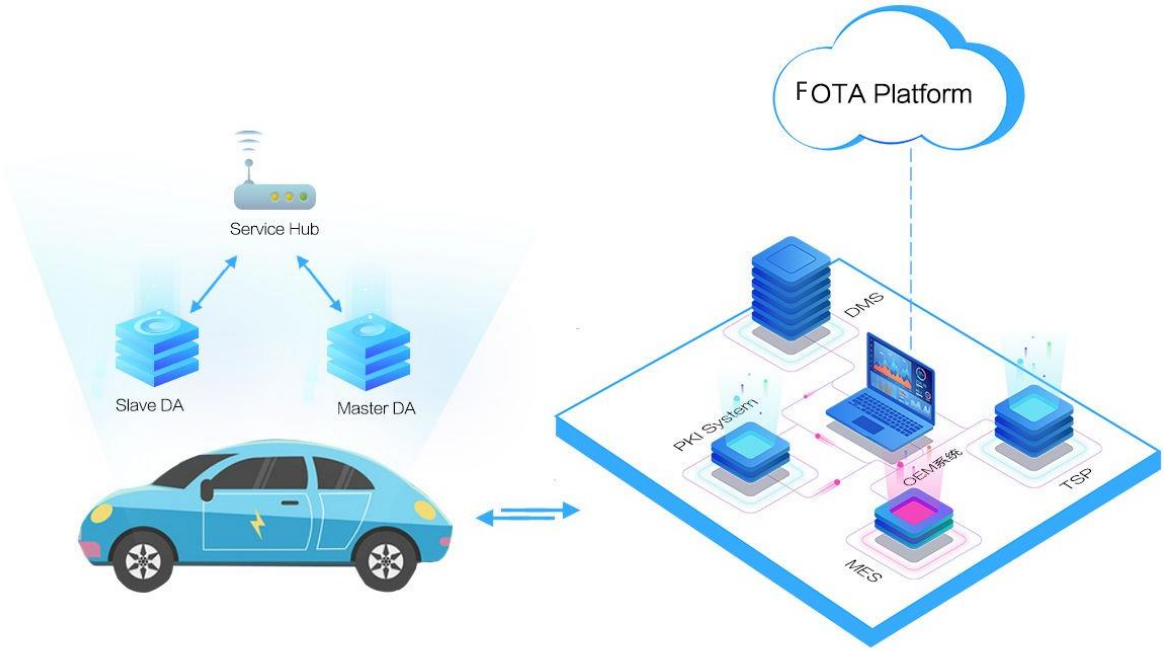


Figure 3. Updating the software of a car using wireless

2. **Rollback Mechanism:**
   In case an update causes unexpected issues or errors, a FOTA system may include a rollback mechanism. This allows the device to revert to the previous firmware version, ensuring system stability.
3. **Notification and User Interaction:**
   Users may be notified about available updates, and the FOTA system may provide options for users to schedule updates at convenient times. This ensures that updates do not disrupt critical operations.
4. **Over-the-air diagnosis:**

The ability to Use hardware and software diagnosis to collect data from devices over the air. This data can be used to troubleshoot problems and to improve the firmware update process.

1.1.4 Solving Software faults remote

In the automotive landscape, FOTA technology represents a significant departure in how we handle issues within vehicles. This initiative leverages Firmware Over-the-Air (FOTA) technology, allowing for the remote updating of car software without the necessity for manufacturers (OAM) to initiate global recalls for fixes. Imagine it as akin to a software update for your smartphone but tailored for automobiles.

FOTA eliminates the need to physically recall all sold cars globally merely to address a software glitch, presenting a more streamlined and resource-efficient process. The project underscores a commitment to an enhanced, cost-effective approach to resolving software-related challenges in vehicles, highlighting FOTA's potential to bring about a transformative shift in how we approach the maintenance and optimization of automotive performances ***as in figure 5***, and everything will be solved remotely.

Figure 4. Updating Vehicle with traditional connection

1.1.5 Problem Statement

Our primary challenge involves remotely updating vehicle software via WI-FI, accessing the server to download the latest software without necessitating a recall to the manufacturing facility.

Additionally, some vehicles may be unable to connect to the server for updates, prompting the utilization of V2V technology to update the software of those cars that cannot establish a direct connection to the server.

1.1.6 The proposed FOTA new features

The V2V communication feature allows vehicles to exchange firmware updates when they are unable to connect to the internet. This feature ensures that even in situations where a car cannot establish an internet connection, it can still receive the latest firmware updates from nearby vehicles that have already downloaded them ***as shown in figure (4).***

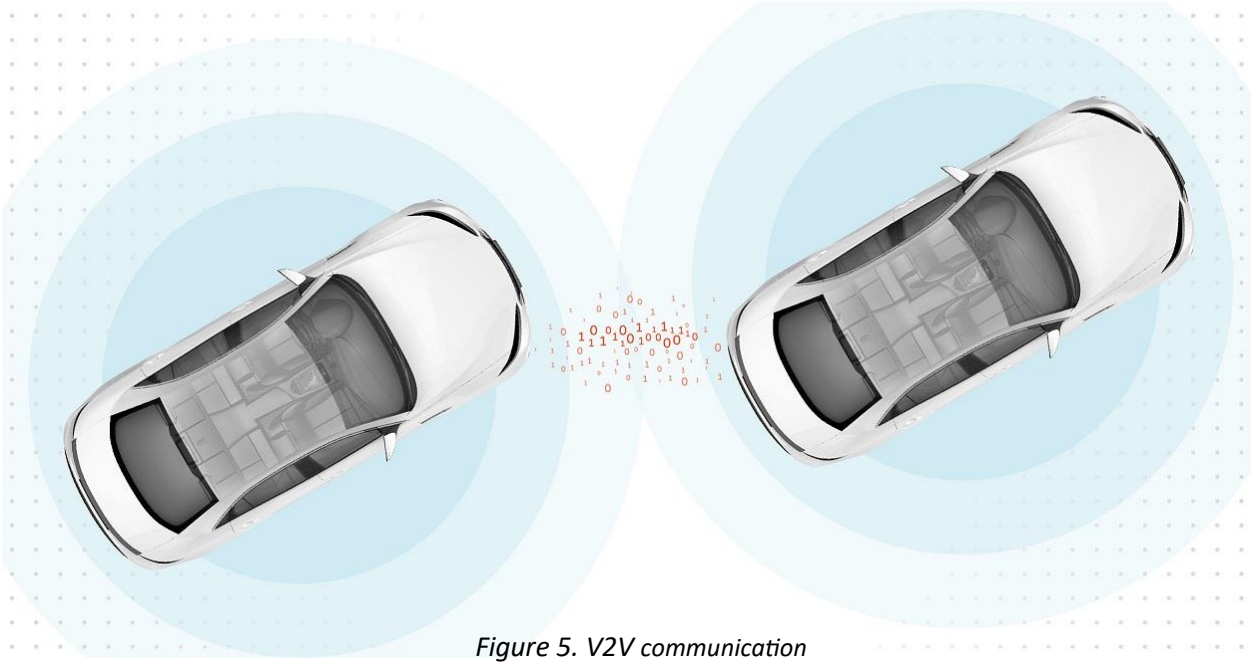


Figure 5. V2V communication

Through the utilization of V2V functionality, cars can create a network and securely share firmware data, ensuring that all vehicles in the vicinity have access to the latest software enhancements and bug fixes. The V2V feature offers several advantages.

Firstly, it reduces reliance on internet connectivity, enabling updates to be received in remote areas or during network connectivity issues. Secondly, it facilitates faster deployment of firmware updates as vehicles within proximity can share the latest software versions without relying on centralized servers. This improves the overall efficiency of the FOTA system.

1.2 Flashing Techniques for Microcontrollers

1.2.1 Off-Circuit Programming

In this technique, to program the microcontroller, the microcontroller shall be removed from its application circuit, and then connected to a burner, which is a hardware kit that sometimes has a socket on which the microcontroller can be placed, or simply connected to the required pins with jumpers. It uploads the program to the flash memory, and when it finishes, the microcontroller can now go back to its application circuit and start working, hence called off circuit programming as it requires removing the microcontroller from its application circuit.

The flash interface is external outside the microcontroller and the flash memory programming pins are connected to some microcontroller pins to be programmed through. When the file is uploaded to the flash using the burner, the microcontroller can be connected again to its application circuit.

1.2.2 In-Circuit Programming

In-system programming (ISP), also called in-circuit serial programming (ICSP), is the ability of some programmable logic devices, microcontrollers, and other embedded devices to be programmed while installed in a complete system, rather than requiring the chip to be programmed before installing it into the system. It also allows firmware updates to be delivered to the on-chip memory of microcontrollers and related processors without requiring specialist programming circuitry on the circuit board and simplifies design work.

There is no standard for in-system programming protocols for programming microcontroller device. Almost all manufacturers of microcontrollers support this feature, but all have implemented their protocols, which often differ even for different devices from the same manufacturer.

The primary advantage of in-system programming is that it allows manufacturers of electronic devices to integrate programming and testing into a single production phase, and save money, rather than requiring a separate programming stage before assembling the system. This may allow manufacturers to program the chips in their own system's production line instead of buying pre-programmed chips from a manufacturer or distributor, making it feasible to apply code or design changes in the middle of a production run. The other advantage is that production can always use the latest

firmware, and new features, as well as bug fixes, can be implemented and put into production without the delay occurring when using pre-programmed microcontrollers.

1.2.3 In-Circuit Programming with Bootloader

A Bootloader is a program that allows you to load other programs via a more convenient interface like a standard USB cable. When you power up or reset your microcontroller board, the bootloader checks to see if there is an upload request. If there is, it will upload the new program and burn it into Flash memory. If not, it will start running the last program that you loaded. So, instead of different interfaces with different microcontrollers, we use a bootloader to create a unified interface with the system.

1.3 Software Development Life cycle

SDLC, or Software Development Life Cycle, is a set of steps used to create software applications. These steps divide the development process into tasks that can then be assigned, completed, and measured.

1.3.1 What is Software Development Lifecycle?

Software Development Life Cycle is the application of standard business practices to building software applications. It's typically divided into six to **Eight** steps: Planning, Requirements, Design, Build, Document, Test, Deploy, and Maintain. Some project managers will combine, split, or omit steps, depending on the project's scope. These are the core components recommended for all software development projects.

SDLC is a way to measure and improve the development process. It allows a fine-grain analysis of each step of the process. This, in turn, helps companies maximize efficiency at each stage. As computing power increases, it places a higher demand on software and developers. Companies must reduce costs, deliver software faster, and meet or exceed their customers' needs. SDLC helps achieve these goals by identifying inefficiencies and higher costs and fixing them to run smoothly.

1.3.2 How the Software Development Life Cycle Works

The Software Development Life Cycle simply outlines each task required to put together a software application. This helps to reduce waste and increase the efficiency of the development process. Monitoring also ensures the project stays on track and continues to be a feasible investment for the company.

Many companies will subdivide these steps into smaller units. Planning might be broken into technology research, marketing research, and a cost-benefit analysis. Other steps can merge. The Testing phase can run concurrently with the Development phase since developers need to fix errors that occur during testing.

1.3.3 The Seven Phases of the System Development Life Cycle (SDLC) as in figure 5:

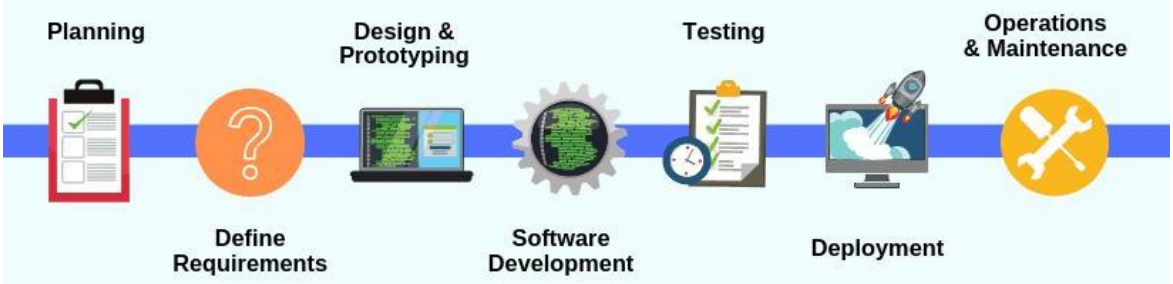


Figure 6: SDLC

1) Planning

In the Planning phase, project leaders evaluate the terms of the project. This includes calculating labor and material costs, creating a timetable with target goals, and creating the project's teams and leadership structure.

Planning can also include feedback from stakeholders. Stakeholders are anyone who stands to benefit from the application. Try to get feedback from potential customers, developers, subject matter experts, and sales reps.

Planning should clearly define the scope and purpose of the application. It plots the course and provisions the team to effectively create the software. It also sets boundaries to help keep the project from expanding or shifting from its original purpose.

2) Define Requirements

Defining requirements is considered part of planning to determine what the application is supposed to do and its requirements. For example, a social media application would require the ability to connect with a friend. An inventory program might require a search feature.

Requirements also include defining the resources needed to build the project. For example, a team might develop software to control a custom manufacturing machine. The machine is a requirement in the process.

3) Design and Prototyping

The Design phase models the way a software application will work. Some aspects of the design include:

- **Architecture –** Specifies programming language, industry practices, overall design, and use of any templates or boilerplate.
- **User Interface –** Defines the ways customers interact with the software, and how the software responds to input.
- **Platforms –** Defines the platforms on which the software will run, such as Apple, Android, Windows version, Linux, or even gaming consoles.
- **Programming –** Not just the programming language but including methods of solving problems and performing tasks in the application.
- **Communications –** Defines the methods by which the application can communicate with other assets, such as a central server or other instances of the application.
- **Security –** Defines the measures taken to secure the application, and may include SSL traffic encryption, password protection, and secure storage of user credentials.

Prototyping can be a part of the Design phase. A prototype is like one of the early versions of software in the Iterative software development model. It demonstrates a basic idea of how the application looks and works. This "hands-on" design can be shown to stakeholders. Use feedback to improve the application. It's less expensive to change the Prototype phase than to rewrite code to make a change in the Development phase.

4) Software Development

This is the actual writing of the program. A small project might be written by a single developer, while a large project might be broken up and worked by several teams. Use an Access Control or Source Code Management application in this phase. These systems help developers track changes to the code. They also help ensure compatibility between different team projects and to make sure target goals are being met.

The coding process includes many other tasks. Many developers need to brush up on skills or work as a team. Finding and fixing errors and glitches is critical. Tasks often hold up the development process, such as waiting for test results or compiling code so an application can run. SDLC can anticipate these delays so that developers can be tasked with other duties.

Software developers appreciate instructions and explanations. Documentation can be a formal process, including wiring a user guide for the application. It can also be informal, like comments in the source code that explain why a developer used a certain procedure.

Even companies that strive to create software that's easy and intuitive benefit from the documentation.

Documentation can be a quick guided tour of the application's basic features that are displayed on the first launch. It can be video tutorials for complex tasks. Written documentation like user guides, troubleshooting guides, and FAQs help users solve problems or technical questions.

5) Testing

It's critical to test an application before making it available to users. Much of the testing can be automated, like security testing. Another testing can only be done in a specific environment – consider creating a simulated production environment for complex deployments. Testing should ensure that each function works correctly. Different parts of the application should also be tested to work seamlessly together—performance test, to reduce any hangs or lags in processing. The testing phase helps reduce the number of bugs and glitches that users encounter. This leads to higher user satisfaction and a better usage rate.

6) Development

In the deployment phase, the application is made available to users. Many companies prefer to automate the deployment phase. This can be as simple as a payment portal and download link on the company website. It could also be downloading an application on a smartphone.

Deployment can also be complex. Upgrading a company-wide database to a newly developed application is one example. Because there are several other systems used by the database, integrating the upgrade can take more time and effort.

7) Operations and Maintenance

At this point, the development cycle is almost finished. The application is done and being used in the field. The Operation and Maintenance phase is still important, though. In this phase, users discover bugs that weren't found during testing. These errors need to be resolved, which can spawn new development cycles.

In addition to bug fixes, models like Iterative development plan additional features in future releases. For each new release, a new Development Cycle can be launched.

Chapter 2: Literature review

2.1 Related papers

Paper 1:

Title: Firmware Update Over the Air (FOTA) for Automotive Industry.
Author(s): Moshe Shavit, Andy Gryc, Radovan Miucic.
Conference Name: Asia Pacific Automotive Engineering Conference.

This paper deals with automotive software update (re-programming), describing the process as of today and suggesting some alternatives, based on experience from the telecom industry. Specifically, we analyze the Firmware Over the Air (FOTA) technology and its applicability to the automotive industry.

Paper 2:

Title: A new code compression method for FOTA.
Author(s): Youngcheul Wee, Taehwa Kim.
Conference Name: IEEE Transactions on Consumer Electronics, Vol. 56, No. 4, November 2010.

This paper presents a new compression method for compressed Firmware-over-the-air (FOTA) that reconstructs a new compressed version from an old, compressed version of firmware stored in NAND flash memory.
The compressed FOTA requires a small delta that minimizes the amount of update data and an efficient compression method that provides fast encoding and decoding to achieve fast updating and extraction. A fast dictionary-based compression algorithm for the program code is introduced. With a high compression ratio, our method provides relatively fast encoding and very fast decoding.
We demonstrate that the compression time during the update process can be reduced by utilizing the copy information stored in the delta without increasing the size of the delta. We present some experimental results to show the impact of this method in terms of the compression ratio, extraction time and updating time.

Paper 3:

Title: Secure Automotive On-Board Protocols: A Case of Over-the-Air Firmware Updates.
Author(s): Idrees, M. S., Schweppe, H., Roudier, Y., Wolf, M., Scheuermann, D., & Henniger, O. (2011).
Discusses the Communication Protocol of vehicular on-board IT architectures very heterogeneous landscape of communication network technologies, e.g., CAN, LIN, Flex-Ray and MOST. Internet Protocol (IP) based communication, new external

communication interfaces e.g., with other vehicles – V2V, or with the infrastructure – V2I based on V2X communications.

Secure On-Board Network Architecture, Implementing Security Primitives

Secure Firmware Update Protocol and Algorithms for Installed, Securing SW Successfully.

Paper 4:

Title: Firmware Over the Air for Automotive, FOTAMOTIVE.

Author(s): Hesham A. Odat and Subra Ganesan.

OEMs are struggling with customer dissatisfaction, SW issues, recall and huge after production cost. FOTAMOTIVE is the solution for customer satisfaction with reduced OEM costs. Most OEMs are currently seeking vendors for FOTA. It is the best time to implement this technology and establish cutting-edge technology. VCM will take the connected car to the next level as well, where it will enable OEMs to extend the SW update process to its component vendors and enable the use of vehicle data in all stages of the vehicle's life cycle.

In This paper discusses the problems facing OEMs, dealer, and consumers. How can Firmware Over the Air (FOTA) solve it, FOTA definition, Process, System Components, Vehicle Cloud Management (VCM), Issue about FOTA.

Paper 5:

Title: Firmware Over the Air Ad-Hoc Network, FOTANET.

Author(s): Hesham A. Odat, Amjad Nsour and Subra Ganesan.

Conference Name: International Conference on Electro/Information Technology (EIT).

Discusses Technology for FOTA by using FOTANET.

Explaining FOTANET definition, Process, System Components, Vehicle Cloud Management (VCM), Issue about FOTA.

Introduced a new solution to lower the cost of the over the air download by downloading the update files from other vehicles that were within a reasonable range and that already have the update files. The solution presented in this paper is expected to lower the cost of software related recalls by 93%.

Paper 6:

Title: Implementation and Research of Bootloader for Automobile ECU Remote.

Author(s): Ji Zhang, Xiangyu Zhu, Yong Peng.

Conference Name: Clean Energy Automotive Engineering Center of Tongji University, Shanghai 201804, China.

This paper discusses the implementation of a bootloader for automobile ECU remote update, focusing on the speed of this method.

The paper introduces the technology of the bootloader and outlines the system design project for the automobile ECU bootloader.

The study also discusses the requirements for the bootloader, including memory layout, specific software running processes, and software structures.

The paper also analyzes the related function requirements for the bootloader, including security, operation reliability, accuracy, and novelty.

The bootloader must be able to deal with external abnormal faults, such as power failures, and ensure the system runs correctly. The study concludes that the bootloader design is more efficient and faster than traditional methods.

Paper 7:

Title: Over-the-Air (OTA) Updates in Embedded Microcontroller Applications.

Author(s): By Benjamin Bucklin Brown.

The OTA update process involves transferring new software from the server to the client, which is then converted into a binary format. This process compiles source code files, links them into executable files, and converts the executable into a portable binary file format.

The over-the-air (OTA) update process in embedded systems presents three major challenges. The first challenge is memory organization, which requires organizing the new software application into volatile or nonvolatile memory of the client device. A previous version of the software must be kept as a fallback application in case the new software has problems. Additionally, the state of the client device between resets and power cycles must be retained.

The second major challenge is **communication.** The new software must be sent from the server to the client in discrete packets, each targeting a specific address in the client's memory. The scheme for packetizing, the packet structure, and the protocol used to transfer the data must all be accounted for in the software design.

The third major challenge is security, which requires ensuring that the server is a trusted party, obfuscating the new software to any observers, and ensuring integrity.

Paper 8:

Title: Research Firmware Update Over the Air from the Cloud.

Author(s): Neven Nikolov.

Conference Name: 2018 IEEE XXVII International Scientific Conference Electronics - ET.

This article describes the study of Firmware update of IoT embedded systems. It is showing a way of updating management software of embedded system from cloud. In the article are research Firmware Update over The Air for embedded system - esp8266.

Paper 9:

Title: Applicable Protocol for Updating Firmware of Automotive HVAC Electronic Control Units (ECUs) Over the Air.

Author(s): Tina Mirfakhraie, Giuliano Vitor, Ken Grogan

Conference Name: 2018 IEEE Confs on Internet of Things, Green Computing and Communications.

Example about FOTA for Updating Automotive HVAC Electronic Control Units based on SAE J1939 standard for automotive industry applications. This reprogramming system uses cellular network standards to transfer the data and Using MCC-FT (standard two-way connectivity gateway for the MCC-HVAC system and other ECUs that are connected to the vehicle CAN bus network).

Paper 10:

Title: Reliability-Oriented Distributed Test Strategy for FOTA/SOTA Enabled Edge Device.

Author(s): Prateek Bajaj, Anoop Dharmarajan, Venkatesh Naik.

Conference Name: Symposium on International Automotive Technology.

To enhance customer experience and to reduce time to market, the manufacturers are constantly in need of being able to update software/firmware of the Electronic Control units (ECU) when the vehicle is in field operations.

The updates could be a bug fix or a new feature release. Until recent years, the updating of software/firmware used to be done using a physical hardwired connection to the Vehicle in a workshop. However, with the element of connectivity being added to the vehicle, the updating of software can be done remotely and wirelessly over the air using a feature called (FOTA).

Paper 11:

Title: Over-the-Air Software-Defined Vehicle Updates Using Federated Fog Environment.

Author(s): Asad Waqar Malik, Anis U. Rahman, Arsalan Ahmad, Max Mauro Dias Santos.

Conference Name: IEEE Transactions on Network and Service Management (Volume: 19, Issue: 4, December 2022).

In this paper, they propose a dissemination framework for OTA updates using a federated fog environment. Pushing the update to all vehicles via the fog nodes may congest the network, leading up to a single-point failure for update dissemination, as well as disruption to other installed services at the fog nodes. We propose a software-defined

mechanism to select a pivot, which gets the update from the fog node and streams it to other vehicles.

Moreover, a timer-barrier mechanism is proposed to identify any malicious vehicles that are barred from participating in pivot selection. The experimental evaluation demonstrates that the proposed scheme improves network convergence time by 40% with 95% node trustworthiness.

Paper 12:

Title: A Secure Firmware Update over the Air of ESP32 using MQTT Protocol from Cloud.

Author(s): Neven Nikolov, Daniela Gotseva.

Conference Name: 2023 XXXII International Scientific Conference Electronics (ET).

In this paper are made a scientifically applied experiment that update controlling firmware of an IoT embedded system. This is achieved by using the MQTT protocol and encrypting the connection between the IoT Device and the IoT Cloud. TLS SSL certificates were used to encrypt the data through the MQTT protocol.

Paper 13:

Title: Study of Hybrid Cryptographic Techniques for Vehicle FOTA System.

Author(s): Manas Borse, Parth Shendkar, Yash Undre, Atharva Mahadik & Rachana Patil.

Conference Name: Mobile Computing and Sustainable Informatics.

Here is a brief of the key points raised in the paper:

Processor Shortcomings: There may be limitations in terms of processor capabilities. The devices often have smaller RAM, making it challenging to handle the relatively large code required for FOTA updates. Ensuring the integrity and authenticity of these devices is also problematic.

Power Usage: IoT devices are typically powered by batteries, and power consumption is a significant concern. Extended operation in harsh environments can lead to increased power usage. Energy-efficient solutions are needed to address this issue.

Scant Utility: IoT devices come in various forms and use different communication protocols, making it difficult to establish a standardized solution for FOTA updates. Heterogeneity among devices complicates the process, and there is no one-size-fits-all approach.

Untrusted Network: Establishing a trusted network to facilitate secure over-the-air updates. However, achieving end-to-end security and mutual authentication is challenging, and not all devices can implement the necessary security mechanisms.

Security is a critical concern for FOTA updates. Ensuring the integrity and authenticity of firmware is essential.

2.2 FOTA References

Project name	Com. protocol	Network	Server	Microcontroller	References	V TO V
FOTA	CAN	ESP8266 Wi-Fi module	N/A	STM32	Ain Shams University	None
FOTA	CAN	N/A	Django	STM32F1/RPI	Ain Shams University	None
FOTA	CAN	ESP8266 Wi-Fi module	Firebase	STM32/RPI 4	Alexandra University	None
FOTA	CAN/UART	ESP8266 Wi-Fi module	Firebase	STM32F103	Mansoura University	None
FOTA	UART	BUILT IN WIFI	Google Cloud	Raspberry Pi 3B (simulates the car) STMF103 (simulates car's ECU)	ITI	None
Paper 1	Shavit, M., Gryc, A., and Miucic, R. (2007)					None
Paper 2	IEEE Transactions on Consumer Electronics (2010)					None
Paper 3	Idrees, M. S., Schweppe, H., Roudier, Y., Wolf, M., Scheuermann, D., & Henniger, O. (2011)					None
Paper 4	Odat, H. A., & Ganesan, S. (2014)					None
Paper 5	Odat, H. A., Nsour, A., & Ganesan, S. (2015)					None
Paper 6	AASRI International Conference on Industrial Electronics and Applications (IEA 2015)					None
Paper 7	Brown, B.B. (2018)					None
Paper 8	IEEE XXVII International Scientific Conference Electronics (2018)					None
Paper 9	Mirfakhraie, T., Vitor, G., & Grogan, K. (2018)					None
Paper 10	Bajaj, P., Dharmarajan, A., Naik, V. (2021).					None
Paper 11	IEEE Transactions on Network and Service Management (2022)					None
Paper 12	XXXII International Scientific Conference Electronics (ET) (2023)					None
Paper 13	Borse, M., Shendkar, P., Undre, Y., Mahadik, A., Patil, R. (2023)					None

2.3 V To V References

Project name	Communication protocol	Module	Microcontroller	Security	References
V2V system Emergency Electronic Feedback	**WIFI**	ESP8266	STM32	**MQTT**	**Ain Shams University**
OTA Feedback on Cars System	**WIFI**	ESP8266	STM32	**MQTT**	**Ain Shams University**
Collision warning system	**WIFI**	ESP8266	TM4C123G	**MQTT**	**Ain Shams University**
Enhancing Road Safety	**WIFI**	ESP8266	STM32	**MQTT**	**ITI**
Enhancing Road Safety	**WIFI**	ESP8266	STM32	None	**ITI**
FOTA System with V To V	**WIF**	ESP8266	STM32	None	**ITI**
ADAS-with-V2V-Communication	**WIFI**	CC3200	TMS57012	None	**mahmud samiunlu**
V2V Communication Safety Aware Application	**WIFI**	ESP8266	Raspberry Pi	None	**Ain Shams University**

Chapter 3: Background

3.1 FOTA with V2V Updates

Firmware Over-The-Air (FOTA) updates have revolutionized the automotive industry by introducing a dynamic approach to software management in vehicles. Traditionally, updating the firmware or software in vehicles required physical access through service centers or dealership visits. However, with FOTA technology, updates can be seamlessly delivered over wireless networks, eliminating the need for physical connections, and greatly improving convenience and efficiency.

FOTA in vehicles enables automakers to continuously enhance the performance, functionality, and security of onboard systems and electronic control units (ECUs). These updates can include bug fixes, feature enhancements, performance optimizations, and critical security patches. By deploying FOTA, automakers can swiftly address software issues, respond to emerging cybersecurity threats, and introduce new features without requiring vehicle owners to visit service centers, resulting in improved customer satisfaction, and reduced operational costs.

Figure 7. FOTA Updates

V2V updates leverage the growing connectivity of modern vehicles and the emergence of sophisticated onboard communication systems. With V2V updates, vehicles can transmit software patches, firmware updates, and other relevant data to nearby vehicles in real-time, creating a dynamic network for distributing critical software improvements and security enhancements.

3.2 UART Communication Protocol:

3.2.1 What is Communication ?

1. **Communication** in embedded systems refers to the **exchange** of **data** or **information between different components** within the embedded system itself or between the **embedded** system and **external** devices.
2. **Serial Communication**: like **UART** (Universal Asynchronous Receiver-Transmitter), **SPI** (Serial Peripheral Interface), and **I2C** (Inter-Integrated Circuit) are **frequently** used in embedded systems for **connecting various components** due to their **simplicity** and **efficiency**.

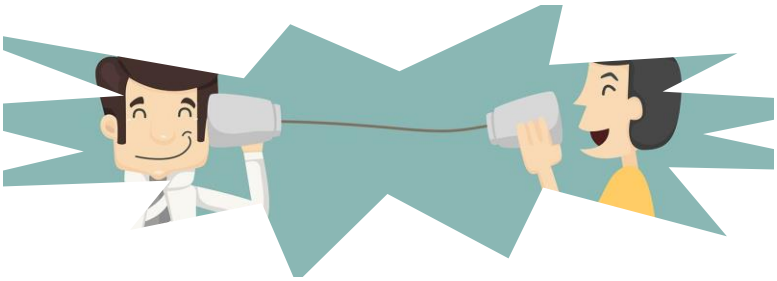


Figure 8. Communication Example

3. There are 4 points we need to know while we are making a communication:
    1. **Data:** Size of data in Bits/Bytes.
    2. **Bit Time:** Time taken To transmit a bit.
    3. **Bit Rate:** Number of bits transmitted per second.
    4. **Baud Rate:** Symbol (Group of bits) transmitted per second.

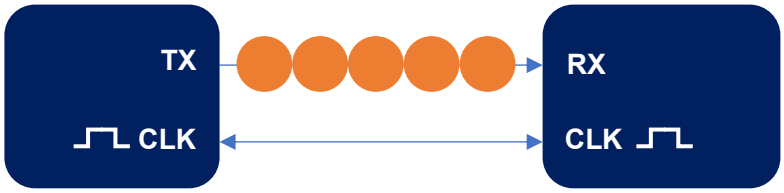
3.2.2 What is USART & UART ?

3.2.2.1 UART (Universal Asynchronous Receiver/Transmitter):

- It operates asynchronously, meaning there is no separate clock signal used to synchronize data transmission.
- In UART communication, data is transmitted one bit at a time, with start and stop bits framing each byte of data.
- UART (Universal Asynchronous Receiver/Transmitter):
- It operates asynchronously, meaning there is no separate clock signal used to synchronize data transmission.
- In UART communication, data is transmitted one bit at a time, with start and stop bits framing each byte of data.

3.2.2.2 USART (Universal Synchronous/Asynchronous Receiver/Transmitter):

- USART is an advanced version of UART that supports both synchronous and asynchronous communication modes.
- In asynchronous mode, it functions similarly to UART, transmitting data with start and stop bits.



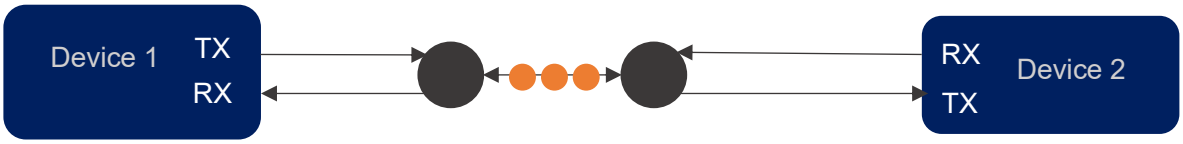
3.2.3 How does UART work?

3.2.3.1 Simplex communication:

- Information can only be transmitted from one device (the transmitter) to other devices (the receivers). The receivers cannot send data back to the transmitter.

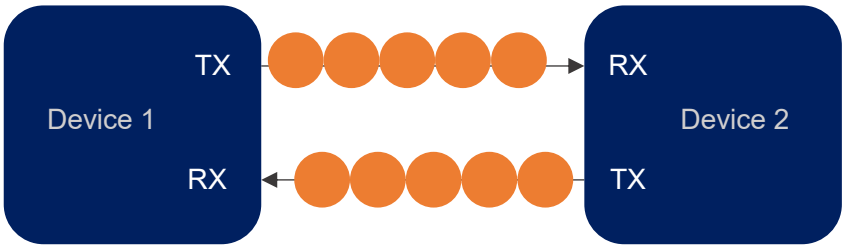
3.2.3.2 Half Duplex communication:

- Can Send and receive but not at the same time.

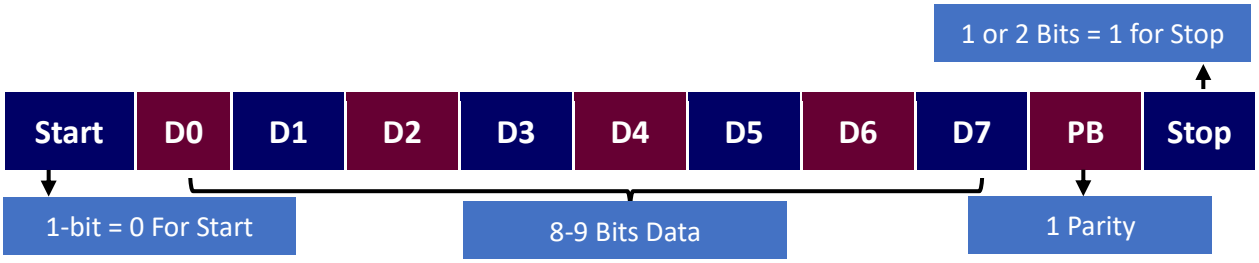


3.2.3.3 Full Duplex communication:

- Can Send and receive at the same time.



3.2.4 UART Frame:



1. **Start bit:** The start bit indicates the beginning of the frame and is always a logic low (0). It prepares the receiver for incoming data.
2. **Data bits:** These are the actual bits of data being transmitted, typically ranging from 5 to 9 bits per character. The number of data bits is configurable and determines the amount of information that can be transmitted in each frame.
3. **Parity bit (optional):** The parity bit is used for error checking. It can be set to odd, even, mark, space, or none, depending on the desired error detection scheme. The parity bit is typically used for detecting transmission errors, although it doesn't correct them.
4. **Stop bit(s):** The stop bit indicates the end of the frame and is always a logic high (1). It gives the receiver time to prepare for the next frame.

3.2.4 UART inside stm32f103c8t6:

The STM32F103C8T6 boasts three UART peripherals (UART1, UART2, and UART3). These UARTs share common functionalities:

- **Asynchronous mode:** Supports data transmission without a clock signal.
- **Configurable baud rate:** Adjustable to match the communication speed of the connected device.
- **Data format:** Selectable data bit size (8-bits or 9-bits), parity (even, odd, none), and stop bits (1 or 2).
- **Interrupt handling:** Offers interrupt generation for events like transmit complete, receive data available, and error conditions.

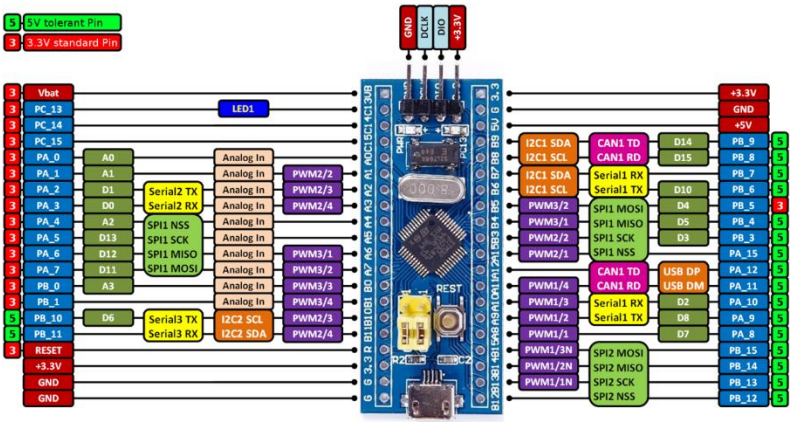


Figure 9. STM Pin Out

3.3 Flash Memory:

Flash memory is a type of non-volatile memory that retains its data even when power is removed. It is commonly used in microcontrollers like the STM32F103C8T6 to store program code, configuration settings, and other essential data.

We will delve into what flash memory is, its significance in the STM32F103C8T6 system, contents typically stored in it, the location of the bootloader within the flash memory, and the process of programming flash memory in STM32F103C8T6.

3.3.1 What is Flash Memory?

Flash memory is a type of non-volatile computer memory that can be electrically erased and reprogrammed. It is commonly used in microcontrollers.

In the context of the STM32F103C8T6, flash memory serves as the primary storage for the firmware and other essential data.

3.3.2 Flash memory inside STM32F103C8T6:

In the STM32F103C8T6 microcontroller, flash memory holds the firmware that controls the operation of the microcontroller. This firmware includes the program code, initialization data, and configuration settings. The flash memory's significance lies in its ability to retain data even when the power is turned off, making it ideal for storing the microcontroller's essential software components.

3.3.3 Contents Typically Stored in Flash Memory:

The flash memory of the STM32F103C8T6 typically stores the following contents:

1. **Bootloader**: The bootloader is a small program that initializes the microcontroller and allows the user to program the flash memory through various interfaces such as UART, USB, or CAN. It is usually located at a specific address in the flash memory.
2. **Firmware:** The main application code, including interrupt vectors, initialization routines, and application-specific code, is stored in the flash memory.
3. **Configuration Settings:** Parameters and settings specific to the application, such as clock configurations, peripheral settings, and user-defined constants, are typically stored in flash memory.

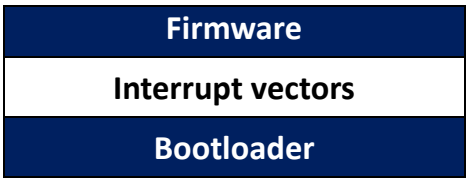


Figure 10. Flash Memory Content

3.3.4 Process of Programming Flash Memory

The general process of programming the flash memory in the STM32F103C8T6 involves the following steps:

1. **Compile Code:** Write and compile the firmware code using an Integrated Development Environment (IDE) such as Keil, IAR, or STM32CubeIDE.
2. **Erase Flash:** Erase the existing contents of the flash memory to prepare for writing new data.
3. **Program Flash:** Load the compiled firmware code into the flash memory using the programming tool and appropriate software.
4. **Verify and Debug:** Verify the programmed data and debug the firmware as necessary to ensure proper functionality.

3.4 ESP-NOW Protocol

ESP-NOW is a communication protocol developed by Express if Systems specifically for peer-to-peer communication between ESP32 devices, offering low-latency, power-efficient, and robust data transmission. Unlike traditional Wi-Fi protocols, ESP-NOW operates in a standalone manner, bypassing the need for Wi-Fi association and authentication, which significantly reduces the communication overhead.

Figure 11. ESP-NOW Protocol

3.4.1 Key Features of ESP-NOW:

1. **Efficiency**: ESP-NOW is designed for efficient data transmission, making it suitable for applications where low latency and minimal power consumption are critical factors. By eliminating the overhead associated with Wi-Fi association and authentication, ESP-NOW optimizes the communication process for quick and reliable data exchange.
2. **Peer-to-Peer:** Communication: ESP-NOW enables direct communication between ESP32 devices without requiring a centralized access point or router. This peer-to-peer architecture simplifies network setup and management, making it ideal for scenarios where devices need to communicate directly with each other in a decentralized manner, such as in V2V (Vehicle-to-Vehicle) communication systems.
3. **Simple Configuration**: Setting up ESP-NOW communication between devices is straightforward, requiring minimal configuration. Devices can be paired using pre-shared keys (PSK) or dynamically generated keys for secure communication. Once paired, devices can exchange data seamlessly without the need for further setup.
4. **Customizable Payload**: ESP-NOW allows for the transmission of custom payloads, enabling developers to send various types of data, including sensor readings, control commands, and status updates, between ESP32 devices. This flexibility makes ESP-NOW suitable for a wide range of IoT (Internet of Things) applications beyond V2V communication.

5. **Low Power Consumption**: ESP-NOW is optimized for low-power operation, making it suitable for battery-powered devices and energy-efficient applications. Devices can enter low-power modes when idle and wake up periodically to send or receive data, prolonging battery life and enabling long-term deployment in remote or mobile environments.

3.4.2 ESP-NOW operating modes

1. Unicast Mode:

- In unicast mode, data is transmitted from one ESP32 device (the sender) to another specific ESP32 device (the receiver).
- This mode is suitable for point-to-point communication, where two devices need to exchange data exclusively with each other.
- Unicast mode allows for direct communication between two devices without the need for a centralized network infrastructure.

2. Broadcast Mode:

- In broadcast mode, data is transmitted from one ESP32 device to all other ESP32 devices within range.
- This mode is useful for scenarios where a device needs to broadcast information to multiple recipients simultaneously.
- Broadcast mode enables efficient dissemination of data to multiple devices without the need to individually address each recipient.

3. Group Mode:

- Group mode allows multiple ESP32 devices to form a communication group, where data can be exchanged among all members of the group.
- Each device in the group can send and receive data to and from all other devices within the same group.
- Group mode facilitates collaborative communication among a defined set of devices, such as in mesh networks or multi-device applications.

3.5 Cryptography

3.5.1 Cryptography explanation:

Cryptography, the practice, and study of techniques for secure communication in the presence of third parties, plays a crucial role in the development and maintenance of secure software systems. As digital information exchange grows exponentially, cryptographic methods have become indispensable in ensuring the confidentiality, integrity, authenticity, and non-repudiation of data. This essay explores the fundamental principles of cryptography and its essential role in software systems, supported by academic references.

Fundamental Principles of Cryptography

Cryptography is founded on several core principles that include confidentiality, integrity, authenticity, and non-repudiation. Confidentiality ensures that information is accessible only to those authorized to have access. This is typically achieved through encryption techniques such as the Advanced Encryption Standard (AES) and Rivest-Shamir-Adleman (RSA) algorithm (Menezes et al., 1996). Integrity guarantees that data has not been altered in transit, often implemented through hash functions like SHA-256 (Stallings, 2017). Authenticity verifies the identity of the entities involved in communication, commonly facilitated by digital signatures and public key infrastructures (PKI). Non-repudiation ensures that once a transaction has occurred, neither party can deny its occurrence, which is critical in legal and financial contexts.

3.5.2 Issues

- **Interception and Eavesdropping**:

In wireless communication, data is broadcasted through radio waves, making it accessible to anyone with the right type of receiver within range. This openness makes interception particularly easy compared to wired networks. Eavesdropping could result in the leakage of confidential information, such as intellectual property, user data, or credentials.

- **Unauthorized Access**:

Wireless networks are inherently more vulnerable to unauthorized access because attackers don't need physical access to the network medium. They can attempt to connect to your network from a distance. If successful, they can steal data, inject malicious data, or even take control of your MCUs.

• **Data Integrity Attacks**:

This involves an attacker modifying the data sent between two devices without the sender or receiver being aware. The attacker could alter commands or data, leading to the devices performing unintended actions. This is particularly dangerous in systems controlling physical devices or collecting sensitive data.

• **Replay Attacks**:

In a replay attack, an attacker captures valid data transmissions and retransmits them to create unauthorized events or transactions. For example, if a device sends a command to unlock a door, capturing and replaying this command could allow an attacker repeated unauthorized access.

• **Man-in-the-Middle (MitM) Attacks**:

This occurs when an attacker intercepts communications between two devices and inserts themselves into the conversation. The attacker can listen to, modify, and relay messages between the parties, who believe they are communicating directly with each other. This can compromise confidentiality and integrity of the data.

• **Denial of Service (DoS) Attacks**:

DoS attacks aim to disrupt service by overwhelming the network or devices with a flood of traffic. In wireless environments, this could be done by jamming the wireless signal or flooding the network with excessive data packets, preventing legitimate communications.

Solutions

• **Encryption**: Implement strong encryption for data in transit to protect it from eavesdropping and interception. Common encryption protocols for wireless communication include WPA2/WPA3 for WiFi and AES for general data encryption.

• **Authentication Protocols**: Ensure that both devices authenticate each other before establishing a connection. Techniques like mutual TLS can be used where both parties verify each other's certificates.

• **Secure Pairing and Key Exchange**: Use secure methods for initial pairing and key exchange, such as Diffie-Hellman, to prevent unauthorized access and ensure that the encryption keys cannot be intercepted.

• **Data Integrity Checks**: Implement mechanisms like checksums, hashes, or MAC (Message Authentication Code) to ensure that the data has not been altered during transmission.

• **Regular Security Updates and Patch Management**: Keep the firmware of the MCUs updated to protect against known vulnerabilities and exploits.

• **Monitoring and Anomaly Detection**: Monitor network traffic for unusual patterns that could indicate a security breach or an attack.

3.5.4 Our Solutions

Advanced Encryption Standard:

The Advanced Encryption Standard (AES) has emerged as a cornerstone for securing data in various domains, particularly in embedded systems where resource constraints pose significant challenges. This abstract delves into the integration of AES-128, a widely adopted variant of the AES algorithm, within embedded systems, elucidating its benefits and implementation nuances.

AES-128 offers a balanced trade-off between security and computational efficiency, making it particularly suitable for resource-constrained embedded platforms. Its fixed block size of 128 bits and key lengths of 128, 192, or 256 bits provide robust encryption while minimizing computational overhead.

Embedded systems, characterized by limited processing power, memory, and energy resources, require lightweight cryptographic solutions without compromising security. AES-128 fulfills this requirement by providing a high level of security with relatively low computational overhead, making it suitable for a wide range of embedded applications, including IoT devices, wearable gadgets, and automotive systems.

Furthermore, AES-128's simplicity facilitates ease of implementation and integration within embedded environments, ensuring compatibility with diverse hardware architectures and software platforms. Its standardized nature enables interoperability across different systems and enhances the scalability of embedded solutions.

In addition to its security and efficiency benefits, AES-128's widespread adoption and support within cryptographic libraries and hardware accelerators streamline the development process for embedded systems engineers. Leveraging existing implementations and optimizations further reduces development time and enhances the overall reliability of embedded cryptographic solutions.

However, successful integration of AES-128 within embedded systems necessitates careful consideration of various factors, including memory footprint, processing overhead, and

power consumption. Optimizing these parameters requires a holistic approach that balances security requirements with resource constraints, ensuring optimal performance without compromising system integrity.

In conclusion, the adoption of AES-128 in embedded systems offers numerous benefits, including robust security, computational efficiency, ease of implementation, and interoperability. By leveraging these advantages, embedded systems designers can enhance the security and reliability of their products while meeting stringent resource constraints in diverse application domains.

Why AES?

Using AES128 encryption is a common and efficient choice for securing data, including file encryption. AES (Advanced Encryption Standard) is a widely adopted symmetric encryption algorithm that provides a good balance between security and performance. Here are some reasons why AES128 is considered efficient for file encryption:

Security: AES is a well-established encryption algorithm that has undergone extensive scrutiny by cryptographers worldwide. AES128 provides a high level of security for most applications, making it suitable for protecting sensitive data.

Performance: AES encryption and decryption operations are efficiently implemented in hardware and software, allowing for fast processing speeds even on resource-constrained devices like microcontrollers. AES128 typically offers a good balance between security and performance.

Resource Efficiency: AES encryption and decryption require relatively low computational resources compared to other encryption algorithms with similar security levels. This makes AES suitable for embedded systems and IoT devices with limited processing power and memory.

Standardization: AES is a standardized encryption algorithm recommended by government agencies and industry standards bodies, providing interoperability and compatibility across different platforms and systems.

However, it's essential to consider your specific use case and security requirements when choosing an encryption algorithm. While AES128 is suitable for many applications, there may be scenarios where stronger encryption (e.g., AES256) or other cryptographic techniques are necessary.

Additionally, the efficiency of file encryption also depends on factors such as the size of the files, the capabilities of the hardware platform, and the implementation of the encryption algorithm. It's essential to benchmark and evaluate the performance of the encryption solution in your specific environment to ensure it meets your requirements.

Feature	RSA	AES
Type	Asymmetric encryption algorithm	Symmetric encryption algorithm
Primary Use	Encryption, digital signatures, key exchange	Encrypting and decrypting data
Common Applications	Securing communications (e.g., SSL/TLS for websites), digital signatures	Protecting data at rest (e.g., files, hard drives), securing data in transit (e.g., network communications)
Computational Cost	High (computationally expensive, especially for large data)	Low (computationally efficient for large amounts of data)
Key Size	Typically 1024 to 4096 bits	Typically 128, 192, or 256 bits

3.5.5 One Time Password

What is a One-Time Password?

A one-time password is a password that is valid for only one login session or transaction. It typically expires after a short period of time or after it is used. OTPs help to mitigate the risks associated with static passwords, such as replay attacks, where an intercepted password can be reused by malicious actors.

How OTPs are Generated

OTPs can be generated in several ways:

•**Time-Based OTPs (TOTP):** These passwords are generated by applying a cryptographic function to the current timestamp, using a shared secret key. TOTPs are commonly valid for 30 seconds before a new code is generated.

•**HMAC-Based OTPs (HOTP**): This method uses a counter and a shared secret key. Each time a password is generated, the counter is incremented, ensuring that each OTP is unique.

•**SMS and Email OTPs**: These are sent to the user's registered mobile phone or email address. They are less secure than TOTP and HOTP due to vulnerabilities in the delivery channels but are widely used for their convenience.

Uses of OTPs

OTPs are used in various authentication processes, including:

•**Two-Factor Authentication (2FA):** In addition to a regular password, an OTP provides a second layer of security. Even if the regular password is compromised, the OTP still protects the account.

Benefits of Using OTPs

•**Multi-factor Authentication (MFA)**: OTPs can be part of a multi-factor authentication strategy, combined with other factors like biometrics or hardware tokens.

•**Transaction Authentication:** OTPs are used to authenticate financial and other high-value transactions, ensuring that the transaction is explicitly authorized by the user. Enhanced Security: OTPs reduce the risk of unauthorized access because even if an OTP is intercepted, it cannot be reused.

•**Mitigation of Phishing and Other Attacks**: Phishing attacks often aim to steal user credentials. Since OTPs are not reusable and expire quickly, the effectiveness of these attacks is reduced.

•**User Convenience**: While providing an additional security layer, OTPs are relatively straightforward for users to handle, especially with the advent of smartphone apps designed for OTP generation.

Considerations and Drawbacks

•**Dependence on Devices**: Users need access to their mobile device or hardware token to receive or generate OTPs, which can be problematic if the device is lost, out of battery, or not in network coverage.

•**Potential for Intercept**: SMS and email OTPs can be intercepted through various means, including SIM swapping and email hacking.

OTPs significantly enhance security in digital environments by ensuring that access rights are dynamically validated, making unauthorized access much more difficult for attackers. They are a vital part of modern cybersecurity strategies, especially for securing sensitive transactions and data access.

3.5.6 References

Bonneau, J., Herley, C., Van Oorschot, P. C., & Stajano, F. (2012). The quest to replace passwords: A framework for comparative evaluation of web authentication schemes. IEEE Symposium on Security and Privacy, 553-567.

Chen, L., Chen, L., Jordan, S., Liu, Y. K., Moody, D., Peralta, R., & Smith-Tone, D. (2016). Report on post-quantum cryptography. NISTIR 8105. National Institute of Standards and Technology.

Dierks, T., & Rescorla, E. (2008). The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246. Internet Engineering Task Force.

Fouque, P. A., Joux, A., & Mavromati, L. (2013). Multi-user collisions: Applications to discrete logarithm, Even-Mansour and PRINCE. Advances in Cryptology–CRYPTO 2013, 44-61.

Katz, J., & Lindell, Y. (2020). Introduction to Modern Cryptography. CRC Press.

Menezes, A. J., van Oorschot, P. C., & Vanstone, S. A. (1996). Handbook of Applied Cryptography. CRC Press.

Nakamoto, S. (2008). Bitcoin: A peer-to-peer electronic cash system. Bitcoin.org.

Stallings, W. (2017). Cryptography and Network Security: Principles and Practice. Pearson.

US Department of Health and Human Services. (2013). Summary of the HIPAA Security Rule. HHS.gov.

Bellovin, S. M. (2011). Frank Miller: Inventor of the One-Time Pad. Cryptologia, 35(3).

Oregon State University. (2021). Chapter 1: One-Time Pad. The Joy of Cryptography.

SpringerLink. (2021). One-Time Pad. SpringerLink.

CSU Ohio. (2021). Theoretical and Practical Significance of One-Time-Pad Cryptography. CSU Ohio.

Chapter 4: Proposed methodology

4.1 FOTA

4.1.1 Bootloader

Writing a bootloader is a critical aspect of firmware development for embedded systems. Bootloaders enable the firmware to be updated or replaced without the need for specialized hardware.

4.1.2 Different Techniques Of Bootloaders

There are several ways to categorize bootloaders, depending on the specific functionalities they offer.

4.1.2.1 Vector Table Relocation:

Explanation:

Vector table relocation involves writing a bootloader that relocates the interrupt vector table to a different location in memory before jumping to the main application. This method allows the bootloader and the application to coexist in the same memory space.

Advantages:

Allows the main application to occupy the lower portion of memory, leaving the upper portion for the bootloader.

Provides flexibility in memory allocation for both the bootloader and the application.

Disadvantages

Requires careful management of memory and interrupts to ensure proper operation.

May complicate the bootloader code due to the relocation process.

Example

An example of vector table relocation is seen in the bootloader development for ARM Cortex-M microcontrollers, where the bootloader relocates the vector table to a specific address in memory before jumping to the main application.

Figure 12. Vector Table Relocation

4.1.2.2 Dual Bank Memory

Explanation:

Dual bank memory refers to a memory architecture that divides the memory into two separate banks, allowing for concurrent access to different memory locations. This architecture is commonly used in embedded systems and microcontrollers to improve performance and efficiency.

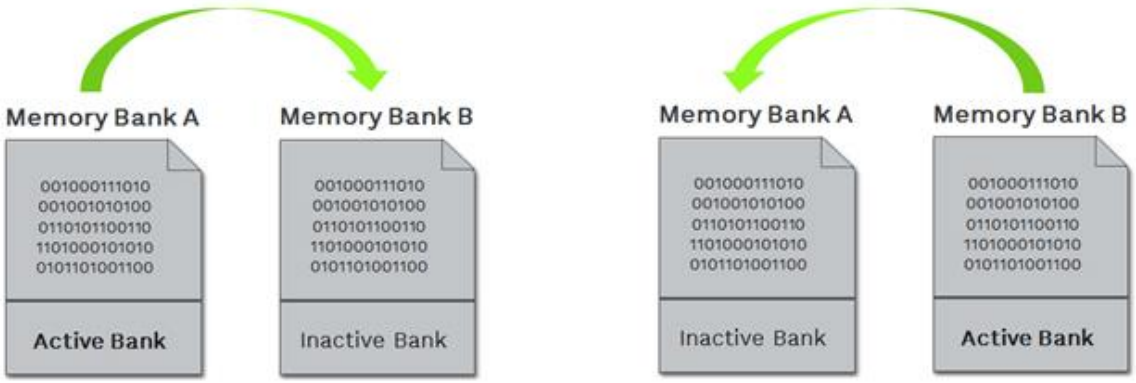


Figure 13. Dual Bank Memory

Advantages:

Provides a fail-safe mechanism as the bootloader and application are stored in separate memory banks.

Enables seamless firmware updates with minimal risk of corruption.

Disadvantages:

Requires careful management of memory banks and address switching.

May result in increased flash memory usage due to the need for redundancy.

4.1.2.3 In-Application Programming (IAP)

Explanation

In-Application Programming (IAP) involves integrating the bootloader code within the application firmware. This allows the microcontroller to update its own firmware without the need for external programming tools.

Figure 14. In-Application Programming (IAP)

Advantages

Simplifies the firmware update process as it eliminates the need for external programming hardware.

Provides flexibility for over-the-air (OTA) updates in connected devices.

Disadvantages

Requires careful memory allocation and error handling to ensure the reliability of the update process.

Security considerations are crucial to prevent unauthorized firmware updates.

Considerations

Robust error handling mechanisms and security measures must be implemented to ensure the integrity of the firmware update process.

Careful memory allocation and activation mechanisms are necessary to avoid unintentional firmware updates.

Example

The integration of OTA firmware updates in Automotive often utilizes the IAP approach, enabling devices to update their firmware over the air without physical access or external programming hardware.

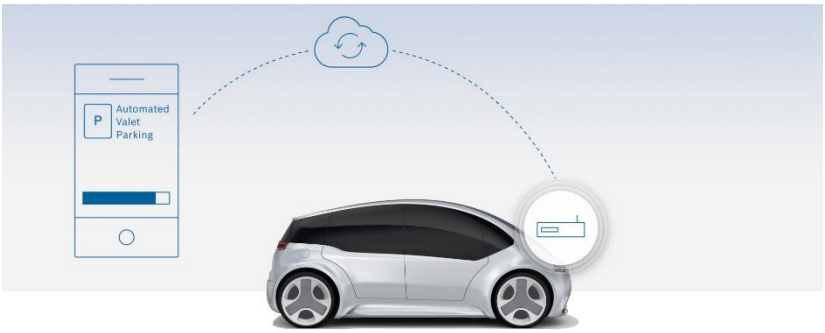
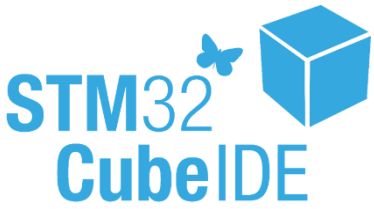


Figure 15. Over The air (IAP)

How do we implement our bootloader?

1) **Setting Up the Development Environment:**
   - install the necessary software tools, including an Integrated Development Environment (IDE) STM32CubeIDE.
   - Download the STM32CubeMX tool, which can generate initialization code and configuration files for STM32 microcontrollers.



2) **Writing the Bootloader Code:**
   - Create a new project in your IDE and configure it for your STM32 microcontroller.
   - Write the bootloader code.
3) **Testing and Debugging:**
   - Test the bootloader by sending firmware update files to the STM32 board using USART Communication protocol.

4.1.2 Write the bootloader code:

First, we will need our bootloader to perform this task:

- Erase flash
- Write on flash
- Get Version
- Get Chip ID

To perform a certain task, we will send across UART an IP represent every command.

This is the commands table of our bootloader (as in figure 8):

```
STM32F407 Custome BootLoader
============================
Which command you need to send to the bootLoader :
   CBL_GET_VER_CMD              --> 1
   CBL_GET_CID_CMD              --> 3
   CBL_FLASH_ERASE_CMD          --> 6
   CBL_MEM_WRITE_CMD            --> 7

Enter the command code :
```

Figure 16. Bootloader Commands

Every choose will convert to a certain value for simplicity:

- CBL_GET_VER_CMD = 0x10
- CBL_GET_CID_CMD = 0x12
- CBL_FLASH_ERASE_CMD = 0x15
- CBL_MEM_WRITE_CMD = 0x16

If we send 1 to STM32 board using UART2 we will receive the **current bootloader version** (As in figure 9).

```
Enter the command code : 1
Request the bootloader version
Host CRC =  0x4cec8427
   0x05    0x10    0x27    0x84    0xec    0x4c
   Received Acknowledgement from Bootloader
   Preparing to receive ( 4 ) bytes from the bootloader

   Bootloader Vendor ID :  100
   Bootloader Version   :  1 . 1 . 0

Please press any key to continue ...
```

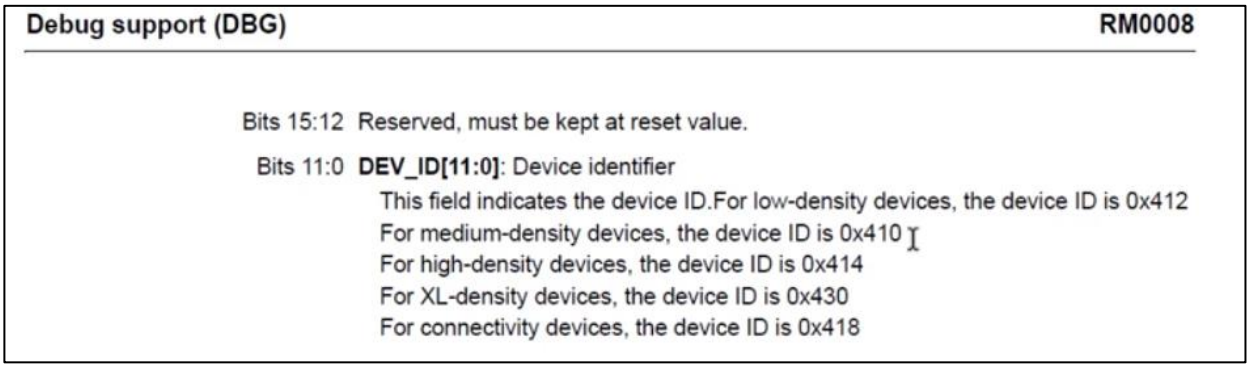
Figure 17. Get Version Command

If we send 3 to STM32 board using UART2 we will receive the **MCU ID** (As in figure 10).

```
Enter the command code : 3
Read the MCU chip identification number
   0x05    0x12    0x49    0xbf    0x6e    0x45
   Received Acknowledgement from Bootloader
   Preparing to receive ( 2 ) bytes from the bootloader

   Chip Identification Number :  0x410
```

Figure 18. Get Chip ID Command

From Datasheet of MCU we will find that stm32f103c8t6 32-bit arm development board Chip id is 0x410 (As in figure 11):



Debug support (DBG) **RM0008**

Bits 15:12 Reserved, must be kept at reset value.

Bits 11:0 **DEV_ID[11:0]**: Device identifier

This field indicates the device ID.For low-density devices, the device ID is 0x412
For medium-density devices, the device ID is 0x410
For high-density devices, the device ID is 0x414
For XL-density devices, the device ID is 0x430
For connectivity devices, the device ID is 0x418

Figure 19. Chip ID of stm32f103c8t6

If we want to **erase all pages**, we send address **0xff** which will erase the flash (As in figure 12).

```
Enter the command code : 6
Mass erase or sector erase of the user flash command

   Please enter start Adress          : 0xff
   0x0a    0x15    0xff    0x00    0x00    0x00    0x00    0x20    0x61    0xcd    0x43
   Received Acknowledgement from Bootloader
   Preparing to receive ( 1 ) bytes from the bootloader

   Erase Status -> Successfule Erase
```

Figure 20. Erase Flash Memory of stm32f103c8t6

If we send 6, we must send an address which we need begin erase from then it will **erase a page starting from this address (As in Figure 13 and 14)**.

```
Enter the command code : 6
Mass erase or sector erase of the user flash command

   Please enter start Adress          : 0x8008000

   Please enter number of pages to erase (12 Max): 1
   0x0a    0x15    0x00    0x80    0x00    0x08    0x01    0x32    0x4f    0x12    0xd5
   Received Acknowledgement from Bootloader
   Preparing to receive ( 1 ) bytes from the bootloader

   Erase Status -> Successfule Erase
```

Figure 21. Erase a page from Flash Memory of stm32f103c8t6

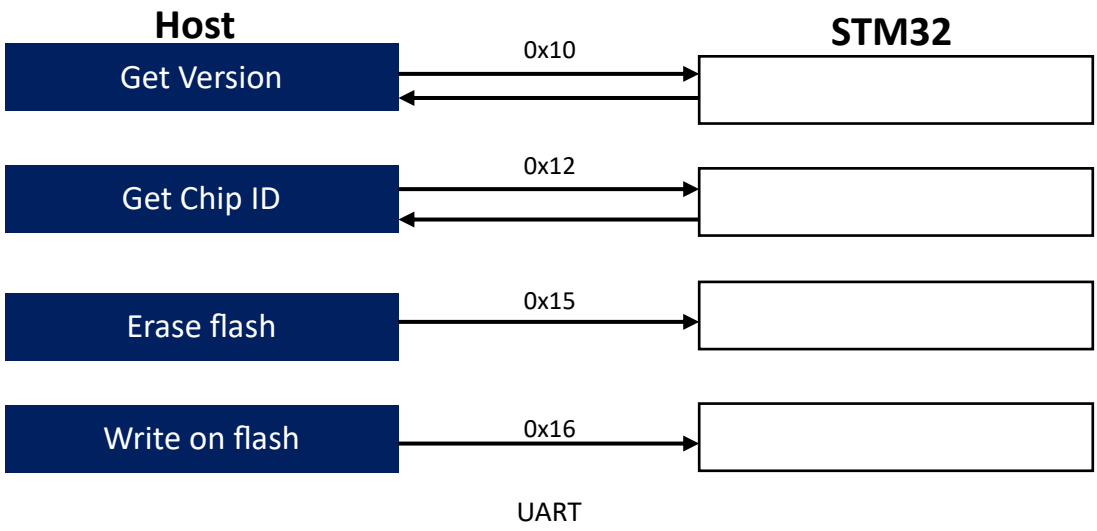
Address	0	4	8	C	ASCII
0x08008000	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x08008010	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x08008020	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x08008030	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x08008040	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x08008050	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x08008060	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x08008070	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x08008080	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
0x08008090	FFFFFFFF	FFFFFFFF	FFFFFFFF	FFFFFFFF	ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ

Figure 22. Erase a page from Flash Memory of stm32f103c8t6

If we send 7 to STM32 board using UART2 we will **Write a new code** on STM32 Flash (As in figure 15).

```
Enter the command code : 7
Write data into different memories of the MCU command
   Preparing writing a binary file with length ( 3920 ) Bytes

   Enter the start address : 0x8008000
2111236098
   0x4a    0x16    0x00    0x80    0x00    0x08    0x40    0xc0    0x06    0x00    0x20    0x89    0x81    0x00    0x08
   0x41    0x8e    0x00    0x08    0x99    0x8d    0x00    0x08    0x3d    0x8e    0x00    0x08    0x91    0x82    0x00
   0x08    0x65    0x8e    0x00    0x08    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00    0x00
   0x00    0x00    0x00    0x00    0x00    0x00    0x49    0x8e    0x00    0x08    0x95    0x82    0x00    0x08    0x00
   0x00    0x00    0x00    0x45    0x8e    0x00    0x08    0x4d    0x8e    0x00    0x08    0x02    0xe8    0xd6    0x7d

   Bytes sent to the bootloader :64

   Received Acknowledgement from Bootloader
   Preparing to receive ( 1 ) bytes from the bootloader

   Write Status -> Write Successfule
792361142
```

Figure 23. Writing a new code on Flash Memory of stm32f103c8t6



Let go deep how these functions performed:

4.1.2.1 Get Version:

```c
static void BL_Get_Version(uint8_t* Host_buffer)
{
    // Define an array containing the bootloader version information
    uint8_t Version[4] = { CBL_VENDOR_ID, CBL_SW_MAJOR_VERSION, CBL_SW_MINOR_VERSION, CBL_SW_PATCH_VERSION };

    // Initialize variables
    uint16_t Host_Packet_Len = 0;
    uint32_t CRC_value = 0;

    // Retrieve the length of the host buffer
    Host_Packet_Len = Host_buffer[0] + 1;

    // Extract the CRC value from the end of the host buffer
    CRC_value = *(uint32_t*)(Host_buffer + Host_Packet_Len - 4);

    // Verify the CRC checksum of the received data
    if (CRC_VERIFING_PASS == BL_CRC_verfiy((uint8_t*)&Host_buffer[0], Host_Packet_Len - 4, CRC_value))
    {
        // Send an acknowledgment (ACK) response
        BL_Send_ACK(4);

        // Transmit the bootloader version information to the host
        HAL_UART_Transmit(&huart2, (uint8_t*)Version, 4, HAL_MAX_DELAY);

        // Jump to the application code (if available)
        Jump_Application();
    }
    else
    {
        // If CRC verification fails, send a negative acknowledgment (NACK) response
        BL_Send_NACK();
    }
}
```

Figure 24. Implementation of Get Version Function

- The function retrieves the bootloader version information and transmits it to the host device. It also verifies data using a CRC checksum.
- An array containing the bootloader version information, including vendor ID, major version, minor version, and patch version.
- If the CRC verification passes, it sends an acknowledgment (ACK) response to the host, transmits the bootloader version information.
- If the CRC verification fails, it sends a negative acknowledgment (NACK) response to the host.

4.1.2.2 MCU ID:

```c
static void BL_Get_Chip_Idendification_nNumber(uint8_t *Host_buffer)
{
    uint16_t Chip_ID = 0;       // Variable to store the chip identification number
    uint16_t Host_Packet_Len = 0;  // Variable to store the length of the host buffer
    uint32_t CRC_value = 0;     // Variable to store the CRC checksum value

    // Retrieve the length of the host buffer
    Host_Packet_Len = Host_buffer[0] + 1;

    // Extract the CRC value from the end of the host buffer
    CRC_value = *(uint32_t*)(Host_buffer + Host_Packet_Len - 4);

    // Verify the CRC checksum of the received data
    if (CRC_VERIFING_PASS == BL_CRC_verfiy((uint8_t*)&Host_buffer[0], Host_Packet_Len - 4, CRC_value))
    {
        // If CRC verification passes, retrieve the chip identification number from the DBGMCU_IDCODE register
        Chip_ID = (uint16_t)(DBGMCU->IDCODE & 0x00000FFF);

        // Send an acknowledgment (ACK) response to the host
        BL_Send_ACK(2);

        // Transmit the chip identification number to the host device
        HAL_UART_Transmit(&huart2, (uint8_t*)&Chip_ID, 2, HAL_MAX_DELAY);
    }
    else
    {
        // If CRC verification fails, send a negative acknowledgment (NACK) response to the host
        BL_Send_NACK();
    }
}
```

Figure 25. Implementation of Chip id Function

- This function retrieves the chip identification number from the DBGMCU peripheral of the STM32 microcontroller.
- It then verifies the integrity of the received data using a CRC checksum.
- If the CRC verification passes, the chip identification number is transmitted to the host device.
- If the CRC verification fails, a negative acknowledgment (NACK) response is sent to the host.

4.1.2.3 Erase:

```c
static void BL_Flash_Erase(uint8_t *Host_buffer)
{
    // Print debug message indicating the start of flash memory erasure
    BL_SendMessage("Start erasing flash memory\r\n");

    // Initialize variables
    uint8_t Erase_status = UNSUCCESSFUL_ERASE;
    uint16_t Host_Packet_Len = 0;
    uint32_t CRC_value = 0;

    // Retrieve the length of the host buffer
    Host_Packet_Len = Host_buffer[0] + 1;

    // Extract the CRC value from the end of the host buffer
    CRC_value = *(uint32_t*)(Host_buffer + Host_Packet_Len - 4);

    // Verify the CRC checksum of the received data
    if (CRC_VERIFING_PASS == BL_CRC_verfiy((uint8_t*)&Host_buffer[0], Host_Packet_Len - 4, CRC_value))
    {
        // Perform flash memory erase operation based on the address and size received from the host
        Erase_status = Perform_Flash_Erase(*((uint32_t*)&Host_buffer[2]), Host_buffer[6]);

        // Send an acknowledgment (ACK) response
        BL_Send_ACK(1);

        // Transmit the erase status to the host device
        HAL_UART_Transmit(&huart2, (uint8_t*)&Erase_status, 1, HAL_MAX_DELAY);
    }
    else
    {
        // If CRC verification fails, send a negative acknowledgment (NACK) response
        BL_Send_NACK();
    }
}
```

Figure 26. Implementation of Erase Function

- The function begins by sending a message indicating the start of the flash memory erasure process.
- Initialized a variable to store the erase status.
- CRC Verification: The function calculates and verifies the CRC checksum of the received data.
- Flash Memory Erasure: If the CRC verification passes, the function **performs a flash memory erase operation** based on the address and size received from the host.
- Acknowledgment: It sends an acknowledgment (ACK) response to the host device to indicate successful reception and processing of the erase request.
- Negative Acknowledgment (NACK): If CRC verification fails, indicating a potential data corruption or tampering, the function sends a negative acknowledgment (NACK) response to the host device.

performs a flash memory erase operation Function:

```c
static uint8_t Perform_Flash_Erase(uint32_t PageAddress, uint8_t page_Number)
{
    // Initialize variables
    FLASH_EraseInitTypeDef pEraseInit;
    HAL_StatusTypeDef Hal_status = HAL_ERROR;
    uint32_t PageError = 0;
    uint8_t PageStatus = INVALID_PAGE_NUMBER;

    // Check if the page number is within the valid range
    if (page_Number > CBL_FLASH_MAX_PAGE_NUMBER)
    {
        PageStatus = INVALID_PAGE_NUMBER;
    }
    else
    {
        PageStatus = VALID_PAGE_NUMBER;

        // Check if the specified page number is valid or if a mass erase is requested
        if (page_Number <= (CBL_FLASH_MAX_PAGE_NUMBER - 1) || PageAddress == CBL_FLASH_MASS_ERASE)
        {
            // Configure flash erase parameters
            if (PageAddress == CBL_FLASH_MASS_ERASE)
            {
                pEraseInit.TypeErase = FLASH_TYPEERASE_PAGES; // Mass erase operation
                pEraseInit.Banks = FLASH_BANK_1;
                pEraseInit.PageAddress = 0x8008000; // Start address for mass erase
                pEraseInit.NbPages = 12; // Number of pages to be erased (example value)
            }
            else
            {
                pEraseInit.TypeErase = FLASH_TYPEERASE_PAGES; // Page erase operation
                pEraseInit.Banks = FLASH_BANK_1;
                pEraseInit.PageAddress = PageAddress; // Start address of the page to be erased
                pEraseInit.NbPages = page_Number; // Number of pages to be erased
            }

            // Unlock flash memory for erase operation
            HAL_FLASH_Unlock();

            // Perform flash memory erase operation
            Hal_status = HAL_FLASHEx_Erase(&pEraseInit, &PageError);

            // Lock flash memory after erase operation
            HAL_FLASH_Lock();

            // Check erase operation status
            if (PageError == HAL_SUCCESSFUL_ERASE)
            {
                PageStatus = SUCCESSFUL_ERASE;
            }
            else
            {
                PageStatus = UNSUCCESSFUL_ERASE;
            }
        }
        else
        {
            PageStatus = INVALID_PAGE_NUMBER;
        }
    }

    // Return the status of the flash memory erase operation
    return PageStatus;
}
```

- **Function Purpose:** Performs flash memory erase operation based on the specified page address and page number.
- **Parameters:** Page Address is the starting address of the flash memory page to be erased, and page Number is the number of flash memory pages to be erased.
- **Return Value:** The status of the flash memory erase operation, which can be *SUCCESSFUL_ERASE*, *UNSUCCESSFUL_ERASE*, or *INVALID_PAGE_NUMBER*.
- **Flash Erase Initialization:** Initializes flash erase parameters based on the specified page address and page number.
- **Flash Erase Operation:** Unlocks flash memory, performs the flash memory erase operation, and locks flash memory again after completion.
- **Error Handling:** Checks for errors during the erase operation and updates the Page Status accordingly.

4.1.2.4 Write:

```
static uint8_t FlashMemory_Paylaod_Write(uint16_t *pdata, uint32_t StartAddress, uint8_t Payloadlen)
{
    // Initialize variables
    uint32_t Address = 0;
    uint8_t UpdataAdress = 0;
    HAL_StatusTypeDef Hal_status = HAL_ERROR;
    uint8_t payload_status = FLASH_PAYLOAD_WRITE_FAILED;

    // Unlock flash memory for writing
    HAL_FLASH_Unlock();

    // Iterate over the payload data and write it to flash memory
    for (uint8_t payload_count = 0, UpdataAdress = 0; payload_count < Payloadlen / 2; payload_count++, UpdataAdress += 2)
    {
        // Calculate the address for writing
        Address = StartAddress + UpdataAdress;

        // Write data to flash memory
        Hal_status = HAL_FLASH_Program(FLASH_TYPEPROGRAM_HALFWORD, Address, pdata[payload_count]);

        // Check if the write operation was successful
        if (Hal_status != HAL_OK)
        {
            payload_status = FLASH_PAYLOAD_WRITE_FAILED;
        }
        else
        {
            payload_status = FLASH_PAYLOAD_WRITE_PASSED;
        }
    }

    // Lock flash memory after writing
    HAL_FLASH_Lock();

    // Return the status of the flash memory write operation
    return payload_status;
}
```

Figure 27. Implementation of Write Function

- **Function Purpose:** Writes payload data to the flash memory starting from the specified address.
- **Parameters:** pdata is a pointer to the payload data to be written, StartAddress is the starting address in flash memory where the data will be written, and Payloadlen is the length of the payload data.
- **Return Value:** The status of the flash memory write operation, which can be *FLASH_PAYLOAD_WRITE_PASSED or FLASH_PAYLOAD_WRITE_FAILED*.
- **Flash Memory Unlocking:** Unlocks flash memory before writing.
- **Data Writing Loop:** Iterates over the payload data and writes it to flash memory using the HAL_FLASH_Program function.
- **Error Handling:** Checks if the write operation was successful and updates the payload_status accordingly.
- **Flash Memory Locking**: Locks flash memory after writing to prevent further modification.
- **Return Status:** Returns the status of the flash memory write operation.

4.1.2.5 Perform the command Function:

```
BL_status BL_FeatchHostCommand()
{
    BL_status status = BL_NACK; // Initialize status as NACK (Not acknowledged)
    HAL_StatusTypeDef Hal_status = HAL_ERROR;
    uint8_t DataLen = 0;
    // Clear the host buffer before receiving new data
    memset(Host_buffer, 0, HOSTM_MAX_SIZE);
    // Prompt the host to enter the data length
    BL_SendMessage("Enter the length \r\n");
    // Receive the data length from the host
    Hal_status = HAL_UART_Receive(&huart2, Host_buffer, 1, HAL_MAX_DELAY);
    // Check if the received data length indicates a jump to application
    if (Host_buffer[0] == 4) {
        Jump_Application();
        return BL_ACK; // ACK if jump is successful (assuming Jump_Application returns without errors)
    }
    // Check for errors during data length reception
    if (Hal_status != HAL_OK) {
        status = BL_NACK;
    } else {
        // Extract the data length from the received byte
        DataLen = Host_buffer[0];
        // Prompt the host to enter the data
        BL_SendMessage("Enter the Data \r\n");
        // Receive the actual data from the host
        Hal_status = HAL_UART_Receive(&huart2, &Host_buffer[1], DataLen, HAL_MAX_DELAY);
        // Include the data length byte for processing
        DataLen++;
        // Check for errors during data reception
        if (Hal_status != HAL_OK) {
            status = BL_NACK;
        } else {
            // Process the received command based on the first byte (command code)
            switch (Host_buffer[1]) {
            case CBL_GET_VER_CMD:
                BL_Get_Version(Host_buffer);
                status = BL_ACK; // Update status to ACK if processing is successful (assuming BL_Get_Version
                    returns without errors)
                break;
            case CBL_GET_CID_CMD:
                BL_Get_Chip_Idendification_nNumber(Host_buffer);
                status = BL_ACK; // Update status to ACK if processing is successful (assuming
                    BL_Get_Chip_Idendification_nNumber returns without errors)
                break;
            case CBL_FLASH_ERASE_CMD:
                BL_Flash_Erase(Host_buffer);
                status = BL_ACK; // Update status to ACK if processing is successful (assuming BL_Flash_Erase
                    returns without errors)
                break;
            case CBL_MEM_WRITE_CMD:
                BL_Write_Data(Host_buffer);
                status = BL_ACK; // Update status to ACK if processing is successful (assuming BL_Write_Data
                    returns without errors)
                break;
            default:
                status = BL_NACK; // NACK for unsupported commands
            }
        }
    }

    return status;
}
```

Figure 28. Implementation of Perform the command Function

The BL_FeatchHostCommand function fetches a command from the host device through UART:

1. **Receives data length:**
    - Waits for host to send data length (1 byte).
2. **Receives command data:**
    - Waits for host to send actual command data based on received length.
3. **Processes command based on code:**
    - Uses Second byte of data (command code) to determine the action:
        - Supported commands call specific functions (e.g., BL_Get_Version).
        - Unsupported commands: returns NACK (Not Acknowledged).
    - Updates status to ACK (Acknowledged) if processing is successful (assuming called functions succeed).
4. **Returns status:**
    - Returns BL_ACK for successful command or BL_NACK for errors.

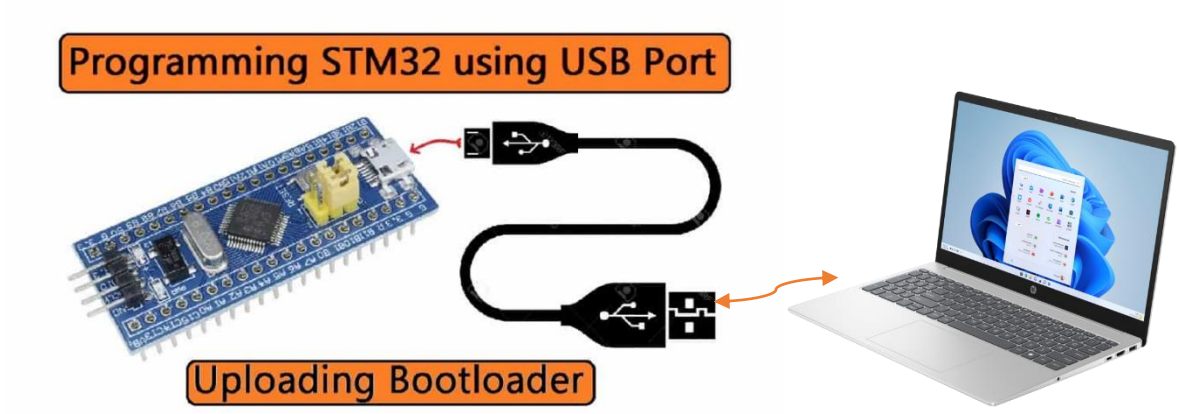
Packet Length	Command	Data	CRC

Figure 29. UART Frame

Packet Length (1 Byte) **Command (1 Byte)** **CRC (4 Bytes)**

Data composed of:

Data Address (4 Bytes) Data Length (1 Bytes) Data (Depend on data)



4.1.3 Firmware Over The Air (FOTA):

What is FOTA?

FOTA stands for Firmware Over-the-Air. It refers to the wireless delivery of software or firmware updates to devices. This eliminates the need to physically connect the device to a computer for updates.

How does FOTA work?

FOTA typically involves Four main components:

1. **Server**: Where the File or (firmware/software) is hosted.
2. **Client**: The vehicle receiving the file.
3. **Communication Protocol**: The method used to transfer the file from the server to the client.
4. **Flashing:** Through a Bootloader

When an FOTA file is initiated, the client device connects to the server, checks for available updates, downloads the update package, and then installs it.

How do we implement our FOTA?

Using ESP32 microcontroller (Figure 11): ESP32, a powerful microcontroller with built-in Wi-Fi and Bluetooth capabilities, Here is the Steps we use for implementing FOTA using ESP32:

1. Establish a connection between the ESP32 and a server (Firebase).
2. Develop firmware update logic on the ESP32 to check for updates and perform the update process.
3. Store firmware updates securely on the server and ensures integrity during transmission.

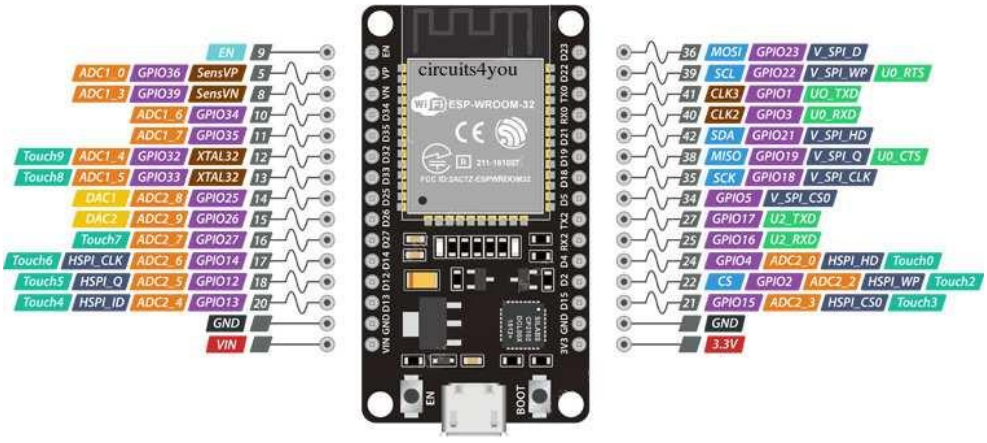


Figure 30. ESP32 MCU

Creation of server will be discussed in 4.1.5, Let go deep how we receive file from our server:

4.1.4 Write ESP32 code Receiving a File from Firebase Over The Air:

4.1.4.1 File Libraries and Constant Declarations

Libraries: This part includes the necessary libraries we use, including Wi-Fi, Firebase, and file system management.

1. **stdint.h**: Provides definitions for standard integer types (e.g., uint8_t, uint32_t).
2. **Arduino.h**: Includes core functionalities for Arduino boards.
3. **WiFi.h**: Includes functions for connecting to Wi-Fi networks.
4. **Firebase_ESP_Client.h**: Library for interacting with Firebase services from an ESP microcontroller.
5. **LittleFS.h**: Library for accessing the LittleFS file system on the board.
6. **addons/TokenHelper.h** and **addons/RTDBHelper.h**: These likely include helper functions for printing Firebase Realtime Database information.

Definitions: It defines API key, database URL, storage bucket ID, and Wi-Fi credentials needed for connecting to Firebase and Wi-Fi network.

1. **API_KEY**: Replace with your actual Firebase API key.
2. **DATABASE_URL**: Replace with the URL of your Firebase Realtime Database.
3. **STORAGE_BUCKET_ID**: Replace with the ID of your Firebase Storage bucket.
4. **WIFI_SSID**: The name of the Wi-Fi network the device should connect to.
5. **WIFI_PASSWORD**: The password for the Wi-Fi network.

```
#include <stdint.h>

#include <Arduino.h>
#include <WiFi.h>
#include <Firebase_ESP_Client.h>
#include <LittleFS.h>

#include "addons/TokenHelper.h"
#include "addons/RTDBHelper.h"

#define API_KEY "AIzaSyC1T33X-TFgmt9LfCcqdp887FFPRTVs0jI"
#define DATABASE_URL "https://fota-2024-default-rtdb.firebaseio.com/"
#define STORAGE_BUCKET_ID "fota-2024.appspot.com"

#define WIFI_SSID "FOTA"
#define WIFI_PASSWORD "FOTA123"
```

4.1.4.2 Global variables used:

Declare global variables and objects needed for Firebase, such as **FirebaseData**, **FirebaseAuth**, and **FirebaseConfig** objects, as well as **flags** indicating **task completion**.

1. **fbdo**: A Firebase data object used for storing data retrieved from Firebase.
2. **auth**: A Firebase authentication object for managing user authentication (optional in this case).
3. **config**: A Firebase configuration object used to set up the connection with Firebase services.
4. **RXp2** and **TXp2**: Define the RX (receive) and TX (transmit) pins used for serial communication on Serial2.
5. **taskCompleted**: A flag (bool) to track the completion of a task (likely the download process)

```
FirebaseData fbdo;
FirebaseAuth auth;
FirebaseConfig config;

#define RXp2 16
#define TXp2 17

bool taskCompleted = false;
```

4.1.4.3 Initialize WI-FI and Connect to Firebase Function:

Initializes Wi-Fi in station mode and connects to the specified network using the provided SSID and password.

- Wi-Fi station mode refers to a networking configuration where ESP32 microcontroller, operates as a client device on a Wi-Fi network rather than as an access point.
- **Firebase.reconnectWiFi(true)**: Enables automatic Wi-Fi reconnection by the Firebase library. This ensures that if the Wi-Fi connection drops, the library will attempt to reconnect automatically.

Connecting to Firebase:

- **config.api_key = API_KEY** : Sets the Firebase API key in the config object. This key is essential for authentication with Firebase services.
- **config.database_url = DATABASE_URL** : Sets the Firebase Realtime Database URL in the config object. This URL specifies the location of our database.
- **if (Firebase.signUp(&config, &auth, "car1", "123")) { ... }**: This code block attempts to sign up with Firebase using the Firebase.signUp function.

- If successful, "ok" is printed to the serial monitor, and the taskCompleted flag is set to true.
- If signup fails, the error message is retrieved and printed using config.signer.signupError.message.c_str().

- **Firebase.begin(&config, &auth):** Initializes the Firebase library with the configured settings in the config object and the authentication object (auth).

```
void initWiFi() {
  WiFi.mode(WIFI_STA); // Set Wi-Fi mode to station
  WiFi.begin(WIFI_SSID, WIFI_PASSWORD); // Connect to Wi-Fi network
  Serial.print("Connecting to Wi-Fi ...");
  while (WiFi.status() != WL_CONNECTED) {
    Serial.print('.');
    delay(1000);
  }
  Serial.println();
  Serial.print("Connected with IP: ");
  Serial.println(WiFi.localIP());

  config.api_key = API_KEY; // Set API key in configuration
  config.database_url = DATABASE_URL; // Set database URL in configuration

  if (Firebase.signUp(&config, &auth, "car1", "123")) {
    Serial.println("ok");
    taskCompleted = true;
  } else {
    Serial.printf("%s\n", config.signer.signupError.message.c_str());
  }

  config.token_status_callback = tokenStatusCallback;
  Firebase.begin(&config, &auth); // Initialize Firebase
  Firebase.reconnectWiFi(true); // Enable automatic Wi-Fi reconnection
}
```

4.1.4.4 File Error checker using CRC Function:

Simple check sum error on (Length, Command, Data).

```
uint32_t Calculate_CRC32(uint8_t* Buffer, size_t Buffer_Length) {
  uint32_t CRC_Value = 0xFFFFFFFF;
  for (size_t i = 0; i < Buffer_Length; i++) {
    CRC_Value = CRC_Value ^ Buffer[i];
    for (int j = 0; j < 32; j++) {
```

```
      if (CRC_Value & 0x80000000) {
        CRC_Value = (CRC_Value << 1) ^ 0x04C11DB7;
      } else {
        CRC_Value = (CRC_Value << 1);
      }
    }
  }
  CRC_Value = CRC_Value & 0xFFFFFFFF;
  return CRC_Value;
}
```

4.1.4.5 Downloaded File State:

This is a callback function used in Firebase Storage to handle download status updates.

- **If the download status is firebase_fcs_download_status_init:**
  It prints information about the file being downloaded, including its name, size, and destination.
- **Calculates values related to the download**:
  - Calculates downloaded_file_size as the size of the file being downloaded.
  - Divides downloaded_file_size by 128 to calculate result.
  - Updates last_packet_round as result + 1.
  - Multiplies result by 128 and subtracts it from downloaded_file_size to update last_data_size.
  - Sets last_packet_size as last_data_size + 10.
  - Prints out the calculated values.
- **If the download status is firebase_fcs_download_status_download**:
  It prints out the progress of the download as a percentage and the elapsed time.
- **If the download status is firebase_fcs_download_status_complete**: It prints a completion message.
- **If the download status is firebase_fcs_download_status_error**:
  It prints out the error message associated with the download failure.

```
int last_data_size;
int last_packet_size;
int last_packet_round;

void fcsDownloadCallback(FCS_DownloadStatusInfo info){
    // Check the status of the download
    if (info.status == firebase_fcs_download_status_init){
    // If the download is initializing, print out file information being downloaded
        Serial.printf("Downloading file %s (%d) to %s\n",
```

```
                    info.remoteFileName.c_str(),
                    info.fileSize,
                    info.localFileName.c_str());

        // Calculate some values related to the download
        int downloaded_file_size = info.fileSize;
        int result = downloaded_file_size / 128;
        last_packet_round = result + 1;
        result = result * 128;
        last_data_size = downloaded_file_size - result;
        last_packet_size = last_data_size + 10;

        // Print out the calculated values
        Serial.println();
        Serial.println(last_packet_round);
        Serial.println(last_data_size);
        Serial.println(last_packet_size);
    }
    else if (info.status == firebase_fcs_download_status_download){
        // If the download is in progress, print out the progress and elapsed time
        Serial.printf("Downloaded %d%s, Elapsed time %d ms\n",
                    (int)info.progress, "%",
                    info.elapsedTime);
    }
    else if (info.status == firebase_fcs_download_status_complete){
        // If the download is complete, print a completion message
        Serial.println("Download completed\n");
    }
    else if (info.status == firebase_fcs_download_status_error){
        // If there's an error during the download, print out the error message
        Serial.printf("Download failed, %s\n", info.errorMsg.c_str());
    }
}
```

4.1.4.6 Combine our Frame like bootloader Function:

We need to make a frame like the one we made for the Bootloader:

Packet Length	Command	Data	CRC

So, we need an array of type uint8 hold this frame:

Packet Length (1 Byte) **Command (1 Byte)** **CRC (4 Bytes)**

Data composed of:

Data Address (4 Bytes) Data Length (1 Bytes) Data (Depend on data)

Packet length (index 0) + Command (index 1) + Data (index 2) + CRC (Packet length – 4).

Since we will use write command to flash onto our controller so, the **command** will be **0x16**.

The array which will hold all of this is **resultArray[MAX_BUFFER_SIZE].**

Summery:

1. The Function Will receive data which needs to be written in arry2.
2. Need to make an array hold our frame.

```
bool last_packet = false;
uint32_t address = 0x08008000; //Application Start Address

void combineArraysAndCRC( uint8_t* array2, size_t array2Size,
  uint8_t* resultArray) {

  uint8_t data_length;
  uint8_t length;
  // Check if it's the last packet
  if(last_packet){
    length = last_packet_size; // Set the packet length to the last packet size
    data_length= last_data_size; // Set the data length to the last data size
  } else{
     length = 0x8a; // Set the packet length to default value
     data_length= 0x80; // Set the data length to default value
  }
  // Make Our Frame
  resultArray[0]=length; // Store packet length [Command + Data + CRC]
  resultArray[1]= 0x16; // Store a Command
  resultArray[2]= address & 0xFF; // Store the lower byte of the address
  resultArray[3]=(address >> 8) & 0xFF; // Store the second byte of the address
  resultArray[4]=(address >> 16) & 0xFF; // Store the third byte of the address
  resultArray[5]=(address >> 24) & 0xFF; // Store the upper byte of the address
  resultArray[6]=data_length; // Store data length in the result array

  address= address+0x80; // Increment the address for the next packet by 0x80
  size_t i = 7; // Initialize index variable for result array
  // Copy the second array into the result array
  for (size_t j = 0; j < array2Size; j++, i++) {
    resultArray[i] = array2[j]; // Copy elements of array2 into result array
  }

  // Calculate CRC for the combined arrays
  uint32_t combinedCRC = Calculate_CRC32(resultArray, data_length + 7);
  // Copy the combined CRC into the result array
  resultArray[i++] = combinedCRC & 0xFF; // Store CRC's lower byte
```

```
  resultArray[i++] = (combinedCRC >> 8) & 0xFF; // Store CRC's second byte
  resultArray[i++] = (combinedCRC >> 16) & 0xFF; // Store CRC's third byte
  resultArray[i] = (combinedCRC >> 24) & 0xFF; // Store CRC's upper byte
}
```

4.1.4.7 Read Data Function:

This Function open file which contain data need to be written on MCU:

1. **Open** the specified file **'rb'** in read mode.
2. If the file is successfully opened, print a success message.
3. **Initialize** a buffer (**chunk**) to **store** a chunk of **file data**, and a variable (bytesRead) to store the number of bytes read from the file.
4. **Read** the file in **chunks** of size CHUNK_SIZE and process each chunk **until** the **end** of the **file** is reached.
5. **Process each chunk** by calling the **combineArraysAndCRC** function to **add chunk data to** the **Frame**.
6. Check if the current chunk is the last packet. If so, set a flag (last_packet) to indicate it's the last packet and print the content of the result array in hexadecimal format.
7. Print the content of the result array.
8. Increment a **counter** (counter) to keep track of the **number** of **processed chunks**.
9. **Perform** some **write** operation (**write()**).
10. Close the file after processing. If the file can't be opened, print an error message.

```
#define CHUNK_SIZE 128 // Adjust the chunk size as needed
#define MAX_BUFFER_SIZE 140
uint8_t resultArray[MAX_BUFFER_SIZE];

void read_binary_file_in_chunks(const char* filename) {
  // Open the file in read mode
  File file = LittleFS.open(filename, "rb");
  if (file) {
    Serial.println("File opened successfully");
    uint8_t chunk[CHUNK_SIZE]; // Buffer to store a chunk of file data
    size_t bytesRead; // store the number of bytes read from the file

    // Read and process the file in chunks until the end of file is reached
    while ((bytesRead = file.read(chunk, CHUNK_SIZE)) > 0) {
      Serial.println("File content:");
      size_t chunk_Size = sizeof(chunk); // Get the size of the chunk
      combineArraysAndCRC(chunk, chunk_Size, resultArray); // Combine The Frame

      // Check if the current chunk is the last packet
```

```
      if (counter == last_packet_round) {
        last_packet = true; // Set the flag indicating it's the last packet
        // Print the result array content in hexadecimal format
        for (int i = 0; i <= last_packet_size; i++) {
          Serial.print(resultArray[i], HEX);
          Serial.print(" ");
        }
      } else {
        // Print the result array content
        for (int i = 0; i <= 138; i++) {
          Serial.print(resultArray[i]);
          Serial.print(" ");
        }
      }
      Serial.println();
      counter++; // Increment the counter

      write(); // Perform some write operation
      // Wait until a specific byte (0xAA) is received from Serial2
      while (Serial2.read() != 0xAA);
    }

    file.close(); // Close the file
  } else {
    Serial.println("Error opening file"); // if the file couldn't be opened
  }
}
```

4.1.4.8 Write Function:

This function sends Write frame using result array which burn code packet by packet.

```
void write(){
  Serial2.write(resultArray[0]);
  delay(10);
  if(last_packet)
  {
    for(int i=1;i<=last_packet_size;i++)
    {
      Serial.println("I AM IN");
      Serial2.write(resultArray[i]);
      delay(10);
    }
  }else
  {
```

```
    for(int i=1;i<=138;i++)
    {
      Serial2.write(resultArray[i]);
      delay(10);
    }
  }
}
```

4.1.4.9 Erase Function:

This function establishes an Erase Frame to stm32 through uart2.

```
uint8_t erase_frame[]={0x15 ,  0xff,  0x00, 0x00, 0x00, 0x00,
0x20,  0x61,  0xcd,  0x43};

void earse_App(){
  /* send the length of the packet*/
  Serial2.write(0x0a);
  delay(100);
  /* send the Frame */
  for(int i=0;i<10;i++)
  {
    Serial2.write(erase_frame [i]);
    delay(100);
  }
}
```

4.1.4.10 Main Function in ESP32:

1. Setup communication Through UART2 with rate 115200.
2. Initialize WI-FI and Log in into Firebase.
3. When STM32 Send 5 through UART Begin to flash.
   a. Download the file from Firebase.
   b. Ensure a successful download.
   c. Erase The past application.
   d. Write new code through `read_binary_file_in_chunks()`

```
void loop() {
  // Check if the input from Serial monitor is 5
  if(Serial.read() == 5) {
    // If the previous task is completed
    if (taskCompleted) {
      taskCompleted = false;
      Serial.println("\nDownload file...\n");

      // Download a file from Firebase Storage
      if (Firebase.Storage.download(&fbdo, STORAGE_BUCKET_ID,
     "application.bin",
     "/downloaded_file.bin",
     mem_storage_type_flash,
     fcsDownloadCallback)) {
        Serial.println("File downloaded successfully");

        // Erase old application
        earse_App();
        delay(1000); // Delay for 1 second
        read_binary_file_in_chunks("/downloaded_file.bin");//Write pack by pack

      } else {
        Serial.println("File download failed");
      }
    }
  }
}
```

4.1.5 Firebase:

Using Firebase as a server (Figure 12) a mobile and web application development platform, offers robust backend services, including cloud storage and real-time database, making it suitable for hosting FOTA updates, Here is the Steps we use for making a server for FOTA using Firebase:

1. Set up a Firebase project and configure cloud storage for storing firmware updates.
2. Develop a backend service or function to manage update requests and serve firmware updates to client devices.

Figure 31. Firebase Server

Firebase security system (Figure 13): Ensuring the secure storage of firmware updates on Firebase and maintaining their integrity during transmission is very important here is how we setup the security system for our database:

1. **Firebase Security Rules:** Set up rules to control access to your Firebase database and storage, allowing only authorized users to upload and download firmware updates.
2. **Firebase Authentication:** Authenticate users accessing firmware updates to ensure only authorized users can interact with them.
3. **HTTPS Transmission:** Use HTTPS protocol when transmitting firmware updates to and from Firebase to encrypt data during transmission.
4. **Access Controls:** Restrict access to firmware updates based on user roles or permissions, allowing only authorized users to modify or access them.

Figure 32. Firebase security system

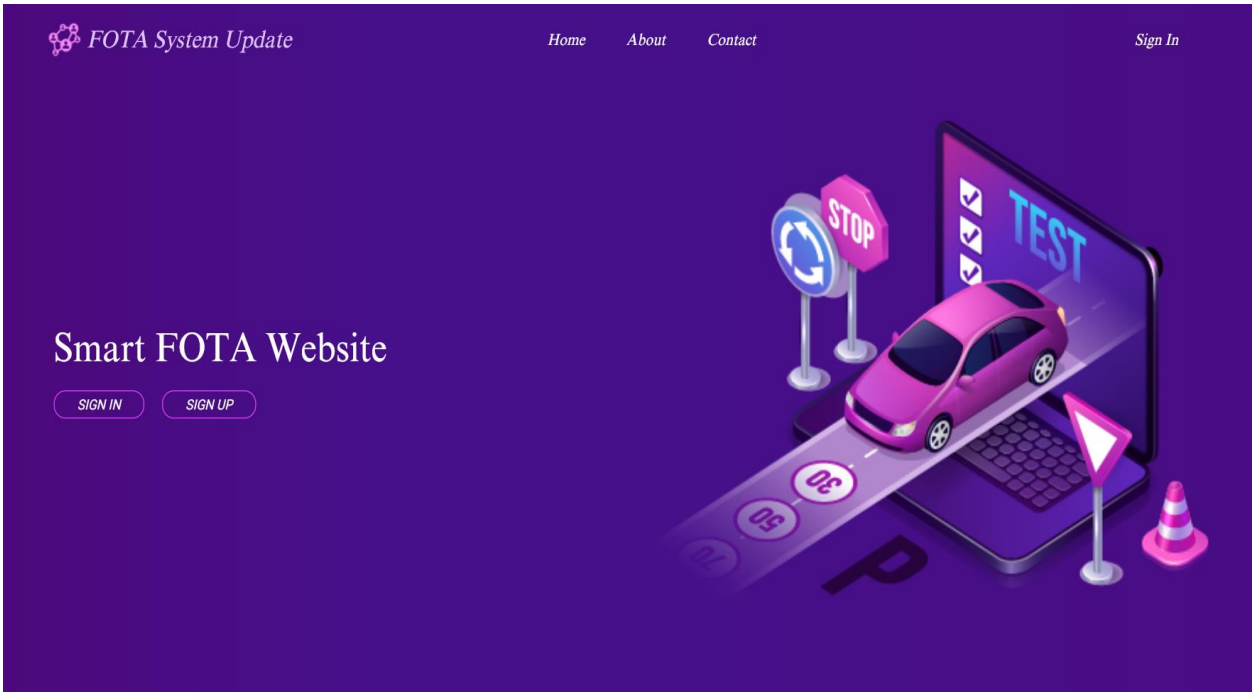


Figure 33. FOTA Sign in page

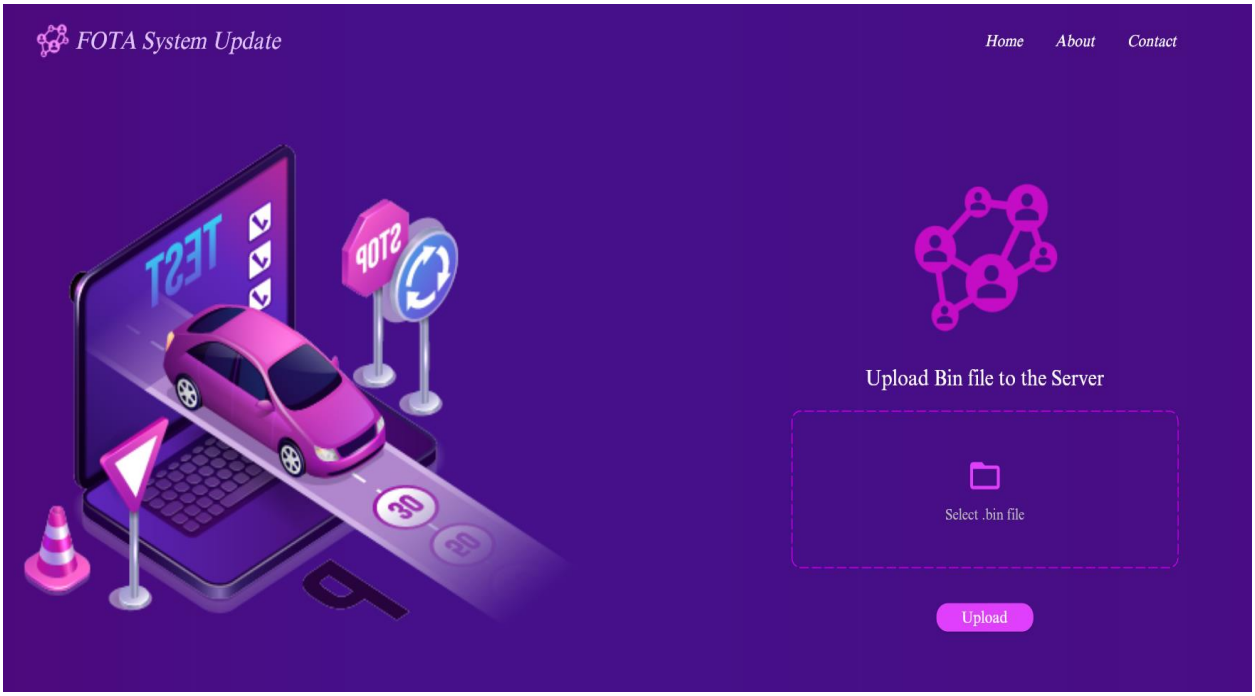


Figure 34. FOTA Upload page

4.2 V2V

The V2V (Vehicle-to-Vehicle) communication system is designed to enable direct peer-to-peer communication between vehicles using ESP32 microcontrollers and the ESP-NOW protocol. The architecture consists of the following key components:

4.2.1 System Architecture

1. ESP32 Devices

- Each vehicle in the system is equipped with an ESP32 microcontroller module, which serves as the primary communication node.
- The ESP32 devices are responsible for collecting and transmitting vehicle data, receiving data from neighboring vehicles, and executing control commands.

2. ESP-NOW Protocol:

- The ESP-NOW protocol is utilized for establishing communication links between ESP32 devices.
- ESP-NOW operates in a peer-to-peer mode, allowing direct communication between vehicles without the need for a centralized network infrastructure.

3. Communication Flow:

- When a vehicle needs to transmit data, it initiates a data packet containing relevant information using ESP-NOW.
- Nearby vehicles equipped with ESP32 devices receive the data packet and process the information accordingly.
- Vehicles respond with acknowledgment packets or transmit their own data packets in response to received information.

4.2.2 Implementation

4.2.2.1 initwifi2() function

- The **initwifi2** function configures the ESP32 as a Wi-Fi Station (**WIFI_STA**) and initializes ESP-NOW communication.
- It registers a callback function, **OnDataSent**, to handle data transmission events.
- The function sets up a peer configuration with the broadcast address and channel information.
- It then adds the peer to the ESP-NOW peer list. If any initialization or peer addition fails, appropriate error messages are displayed.

```
void initwifi2() {
  // Set ESP32 as a Wi-Fi Station
  WiFi.mode(WIFI_STA);

  // Initilize ESP-NOW
```

```
  if (esp_now_init() != ESP_OK) {
    Serial.println("Error initializing ESP-NOW");
    return;
  }
  // Register the send callback
  esp_now_register_send_cb(OnDataSent);
  // Register peer
  memcpy(peerInfo.peer_addr, broadcastAddress, 6);
  peerInfo.channel = 0;
  peerInfo.encrypt = false;
  // Add peer
  if (esp_now_add_peer(&peerInfo) != ESP_OK){
    Serial.println("Failed to add peer");
    return;
  }
}
```

4.2.2.2 file_to_arrays() function

The **file_to_arrays** function serves to read data from a specified file and process it in chunks, ultimately storing the processed data into a multidimensional array. Upon invocation, the function takes a filename as its input parameter, representing the name of the file to be processed.

Using the LittleFS file system, the function attempts to open the specified file in read mode ("rb"). Upon successful file opening, the function proceeds to read the contents of the file in fixed-size chunks, each of size defined by the constant **CHUNK_SIZE**.

Within a loop iterating over the file's contents, data is read into a temporary buffer **chunk**, and the number of bytes read is stored in the variable **bytesRead**.

Each chunk of data is then passed to a processing function, **combineArraysAndCRC**, which performs required operations such as combining arrays and calculating the CRC (Cyclic Redundancy Check) for ensuring data integrity.

The processed data, represented by **resultArray**, is subsequently copied into a multidimensional array **myarr**, indexed by the variable **i**, signifying the current iteration. This process continues until the entirety of the file has been read and processed.

The function concludes its execution once all chunks of data have been processed and stored in **myarr**. In the event of a failed file operation, appropriate error handling is implemented to ensure the function's robustness and reliability.

```
void file_to_arrays(const char* filename)
{
    File file = LittleFS.open(filename, "rb");
  if (file) {
    Serial.println("File opened successfully");
```

```
    uint8_t chunk[CHUNK_SIZE];
    size_t bytesRead;
int i=0;
    // Read and process the file in chunks
    while ((bytesRead = file.read(chunk, CHUNK_SIZE)) > 0) {
      Serial.println("File content:");
      size_t chunk_Size = sizeof(chunk);
      combineArraysAndCRC( chunk, chunk_Size, resultArray);
      memcpy(myarr[i], resultArray, sizeof(resultArray));
i++;
    }
}
}
```

4.2.2.3 v2v send

This part sends data using ESP-NOW to the broadcast address and we use it in the are that has the update and it will send it to another car. It then prints the contents of the data array to the Serial Monitor for debugging purposes.

After sending the data, it checks the result of the transmission operation. If successful, it prints "Sending confirmed"; otherwise, it prints "Sending error".

The index i is incremented to prepare for the next transmission, and there's a delay of 200 milliseconds before the next iteration.

```
// Send message via ESP-NOW
  esp_err_t result = esp_now_send(broadcastAddress, (uint8_t *) &myarr[i],
sizeof(myarr[i]));
      for(int j=0 ; j<= 138 ; j++)
        {
          Serial.print(myarr[i][j] );
          Serial.print(" ");
        }
    Serial.println(" ");
  if (result == ESP_OK) {
    Serial.println("Sending confirmed");
  }
  else {
    Serial.println("Sending error");
  }
  i++;
  delay(200);
  }
```

4.2.2.4 v2v receive

The OnDataRecv function is an event handler triggered used as a call back function for the car that needs the update when data is received via ESP-NOW.

It iterates through the received data array, printing each element to the Serial Monitor for debugging purposes.

The received data is then stored in a multidimensional array of receivedArrays.

The function increments a counter k and sets a flag to indicate that data has been received. After processing the data, it prints a newline character to the Serial Monitor.

```
uint8_t receivedArrays[NUM_ARRAYS][MAX_BUFFER_SIZE];
// Callback function executed when data is received
int i=0,j=0;
void OnDataRecv(const uint8_t * mac, const uint8_t *incomingData, int len) {
  j=0;

  memcpy(receivedArrays[i], incomingData, sizeof(incomingData));
  Serial.println("Data received: ");

        for(int j=0 ; j<= 138 ; j++)
        {
          Serial.print(incomingData[j] );
          Serial.print(" ");
        }

}
```

4.3 Security System

Our program provides functionality for encrypting and decrypting files using AES encryption in CBC mode. It includes utility functions for padding and unpadding data to conform to AES block size requirements. The program also generates random OTPs for authentication purposes and sends them to both an ESP32 device and a mobile number via SMS using the Twilio API. The main function orchestrates user interaction, allowing the user to select files for encryption or decryption and subsequently handling OTP generation and transmission. The environment variables needed for Twilio API integration are loaded at the start.

4.3.1 System architecture

1. Desktop Application:

The program is responsible for encrypting the update and generate the OTPs and send them to users via SMS message, Vehicle via https protocol.

2. HTTPS Server:

It's for make secure channels for send OPTs to vehicles, and it's implemented in Vehicle's system (ESP32).

3. Twilio Server:

It's cloud service for sending the OPTs and notify the user about updates.

4.3.2 Communication Flow:

1. After running the program, it asks the user for enter the path for the update it wants to encryption then ask for file name.
2. When the encryption process ends, the program fire event to generate OTPs automatically one to V2V communication, other for downloaded file from the server.
3. Then the program asks user for enter the for Vehicle to start the channel via https protocol.
4. In the end of program, it asks for phone number for sending SMS message contains update name and OTPs.

4.5.3 System Implementation

1. Environment Variables Loading:

```
load_dotenv()
```

Function: load_dotenv() loads environment variables from a .env file into the program's environment.

Purpose: This is typically used to keep sensitive information like API keys or database credentials secure and out of the source code.

2. Padding Function:

```
def pad(s):
    return s + b"\0" * (AES.block_size - len(s) % AES.block_size)
```

Function: pad(s) takes a byte string s and pads it with null bytes (\0) to make its length a multiple of AES.block_size.

Parameter:

s: The byte string to be padded.

Purpose: Ensures that the data size is compatible with the block size required by AES encryption.

3. Unpadding Function:

```
def unpad(s):
    return s.rstrip(b"\0")
```

Function: unpad(s) removes the padding (null bytes) from the byte string s.

Parameter:

s: The padded byte string to be unpadded.

Purpose: Restores the original data size after decryption.

4. File Encryption Function

```
def encrypt_file(key, iv, file_path, output_file_path):
    chunk_size = 64 * 1024
    encryptor = AES.new(key, AES.MODE_CBC, iv)
    with open(file_path, 'rb') as infile:
        with open(output_file_path, 'wb') as outfile:
            outfile.write(iv)
            while True:
                chunk = infile.read(chunk_size)
                if len(chunk) == 0:
                    break
                elif len(chunk) % AES.block_size != 0:
                    chunk = pad(chunk)
                outfile.write(encryptor.encrypt(chunk))
    print(f"File encrypted and saved in path : {output_file_path}")
```

Function: encrypt_file encrypts a file using AES encryption in CBC mode.

Parameters:

key: The encryption key (must be 16, 24, or 32 bytes long).

iv: The initialization vector (must be 16 bytes long).

file_path: Path to the file to be encrypted.

output_file_path: Path where the encrypted file will be saved.

Purpose: Reads the input file in chunks, encrypts each chunk, and writes the encrypted data to a new file, prepending the IV to the encrypted data.

5. File Decryption Function

```
def decrypt_file(key, iv, file_path, output_file_path):
    chunk_size = 64 * 1024
    decryptor = AES.new(key, AES.MODE_CBC, iv)
    with open(file_path, 'rb') as infile:
        original_iv = infile.read(16)  # The first 16 bytes contain the IV
        with open(output_file_path, 'wb') as outfile:
            while True:
                chunk = infile.read(chunk_size)
                if len(chunk) == 0:
                    break
                outfile.write(unpad(decryptor.decrypt(chunk)))
    print(f"File decrypted and saved in path : {output_file_path}")
```

Function: decrypt_file decrypts an AES-encrypted file.

Parameters:

key: The encryption key.

iv: The initialization vector.

file_path: Path to the encrypted file.

output_file_path: Path where the decrypted file will be saved.

Purpose: Reads the encrypted file, decrypts it in chunks, and writes the decrypted data to a new file, removing padding from the final chunk.

6. OTP Generation Function:

```
def generate_otp(length=4):
    """Generate a random OTP of specified length."""
    digits = "0123456789"
    otp = "".join(random.choice(digits) for _ in range(length))
    return otp
```

Function: generate_otp generates a random numeric One-Time Password (OTP) of specified length.

Parameters:

length: The length of the OTP to be generated (default is 4).

Purpose: Creates a random OTP for authentication or validation purposes.

7. Send OTP via SMS Function:

```
def send_otp(mobile_number, dotp, votp, update_name):
    """Send the OTPs to the specified mobile number using Twilio."""
    account_sid = os.getenv('TWILIO_ACCOUNT_SID')
    auth_token = os.getenv('TWILIO_AUTH_TOKEN')
    client = Client(account_sid, auth_token)

    try:
        message = client.messages.create(
            body=f"Your update is {update_name}\nYour OTP for download file is
            {dotp}\nand Your OTP for V2V update is {votp}",
            from_=os.getenv('TWILIO_PHONE_NUMBER'),
            to=mobile_number
        )
        print(f"Message sent: {message.sid}")
    except TwilioRestException as e:
        print(f"Failed to send message: {e}")
```

Function: send_otp sends the generated OTPs to a specified mobile number using the Twilio API.

Parameters:

mobile_number: The recipient's mobile number.

dotp: The OTP for file download.

votp: The OTP for V2V (vehicle-to-vehicle) update.

update_name: Name of the update.

Purpose: Uses Twilio's messaging service to send an SMS containing the OTPs to the recipient.

8. Send OTPs to ESP32 Function

```
def send_otps_to_esp32(dotp, votp, esp32_ip):
    """Send the OTPs to the ESP32 via HTTP POST."""
    url = f"http://{esp32_ip}/otp"
    data = {'Dotp': dotp, 'Votp': votp}
    response = requests.post(url, data=data)
    print(f"Response from ESP32: {response.text}")
```

Function: send_otps_to_esp32 sends the OTPs to an ESP32 device via an HTTP POST request.

Parameters:

dotp: The OTP for file download.

votp: The OTP for V2V update.

esp32_ip: The IP address of the ESP32 device.

Purpose: Communicates with an ESP32 device by sending the OTPs to it over a network.

9. Main Function

```
def main():
    key = b'\x2b\x7e\x15\x16\x28\xae\xd2\xa6\xab\xf7\x97\x27\x5d\x8b\x3e\x2b'
    iv = b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\x0c\x0d\x0e\x0f'

    directory = prompt("Enter the directory to search for
      files:",completer=PathCompleter(only_directories=True))
    #directory  = os.path.dirname(os.path.abspath(__file__))
    while True:
        print("1. Encrypt a file")
        print("2. Decrypt a file")
        choice = input("Select an operation (1-Encrypt, 2-Decrypt): ")

        if choice not in ['1', '2']:
            print("Invalid choice. Please enter 1 or 2.")
            continue

        completer = PathCompleter(only_directories=False, get_paths=lambda: [directory])
        file_name = prompt("Enter the name of the file: ", completer=completer)
        file_path = directory+"\\"+ file_name

        if not os.path.exists(file_path):
            print("File not found.")
            continue

        if choice == '1':
            output_file_path = f"{file_path}.enc"
            encrypt_file(key, iv, file_path, output_file_path)
        elif choice == '2':
            if file_path.endswith(".enc"):
                output_file_path = file_path[:-4]
```

```
            decrypt_file(key, iv, file_path, output_file_path)
        else:
            print("The selected file does not have an .enc extension.")
            continue

    # Generate 4-digit OTPs
    dotp = generate_otp()
    print("Generated DOTP:", dotp)
    votp = generate_otp()
    print("Generated VOTP:", votp)
    update_name = file_name
    # ESP32 IP address
    esp32_ip = str(input ("enter ip :"))  # Replace with the actual IP address of ESP32
    # Send OTPs to ESP32
    send_otps_to_esp32(dotp, votp, esp32_ip)
    # Mobile number to send OTP to
    mobile_number = '+201066973032'
    # Send OTP via SMS
    send_otp(mobile_number, dotp, votp, update_name)


    repeat = input("Press 1 to exit, 2 to perform another operation: ")
    if repeat == '1':
        break
```

Function: main serves as the entry point of the program, handling user interaction, file encryption/decryption, and OTP generation and sending.

Steps:

1.Sets up encryption key and IV.

2.Prompts the user to select a directory and then to choose an operation (encrypt or decrypt).

3.Handles file selection, encryption, and decryption based on user input.

4.Generates OTPs and sends them to both an ESP32 device and a specified mobile number via SMS.

5.Repeats or exits based on user input.

4.4 Application

4.4 Making our V2V FOTA PCB Board:

Advantages of PCBs

- PCBs offer numerous advantages over other wiring methods, such as point-to-point wiring or wire wrapping.
- They enhance reliability by reducing the likelihood of loose connections and short circuits.
- PCBs also enable miniaturization, allowing for the creation of smaller and more compact electronic devices.

Tool we used for making a PCB:

- **Altium Design**:
  - Altium Designer is a comprehensive PCB design software suite that offers a unified environment for schematic capture, PCB layout, and design validation.
  - It provides a range of powerful tools and features to streamline the entire PCB development process.

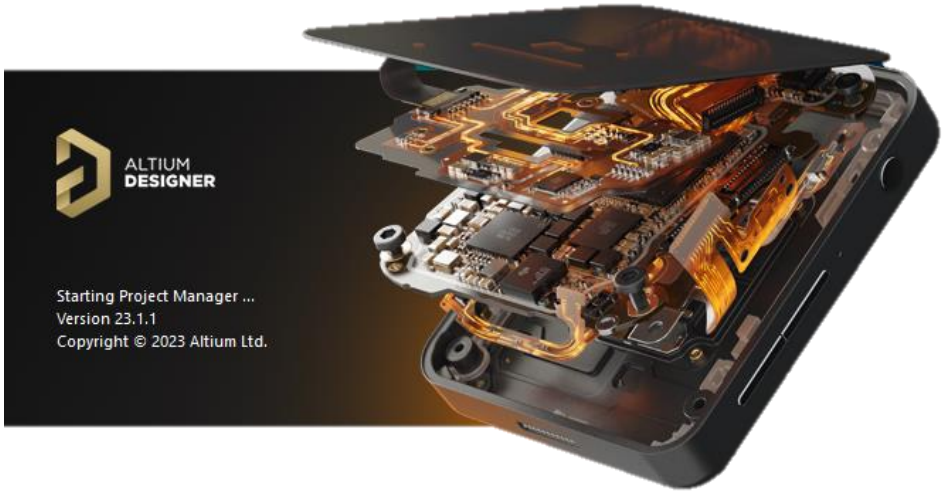


Figure 35. Altium Designer Program

4.4.1 Making our Schematic Sheet:

The schematic sheet is a fundamental aspect of PCB design, serving as a roadmap for the layout and interconnection of electronic components.

The components we used in our project are:

- LCD 16*2 **(Showing the progress of Flashing)**.
- STM32 Development board **(Target Bootloader MCU)**.
- ESP32 Development Board **(Wi-Fi Modul)**.
- 4 Switches with 4 Pull Up Resistors **(For Interfacing)**.

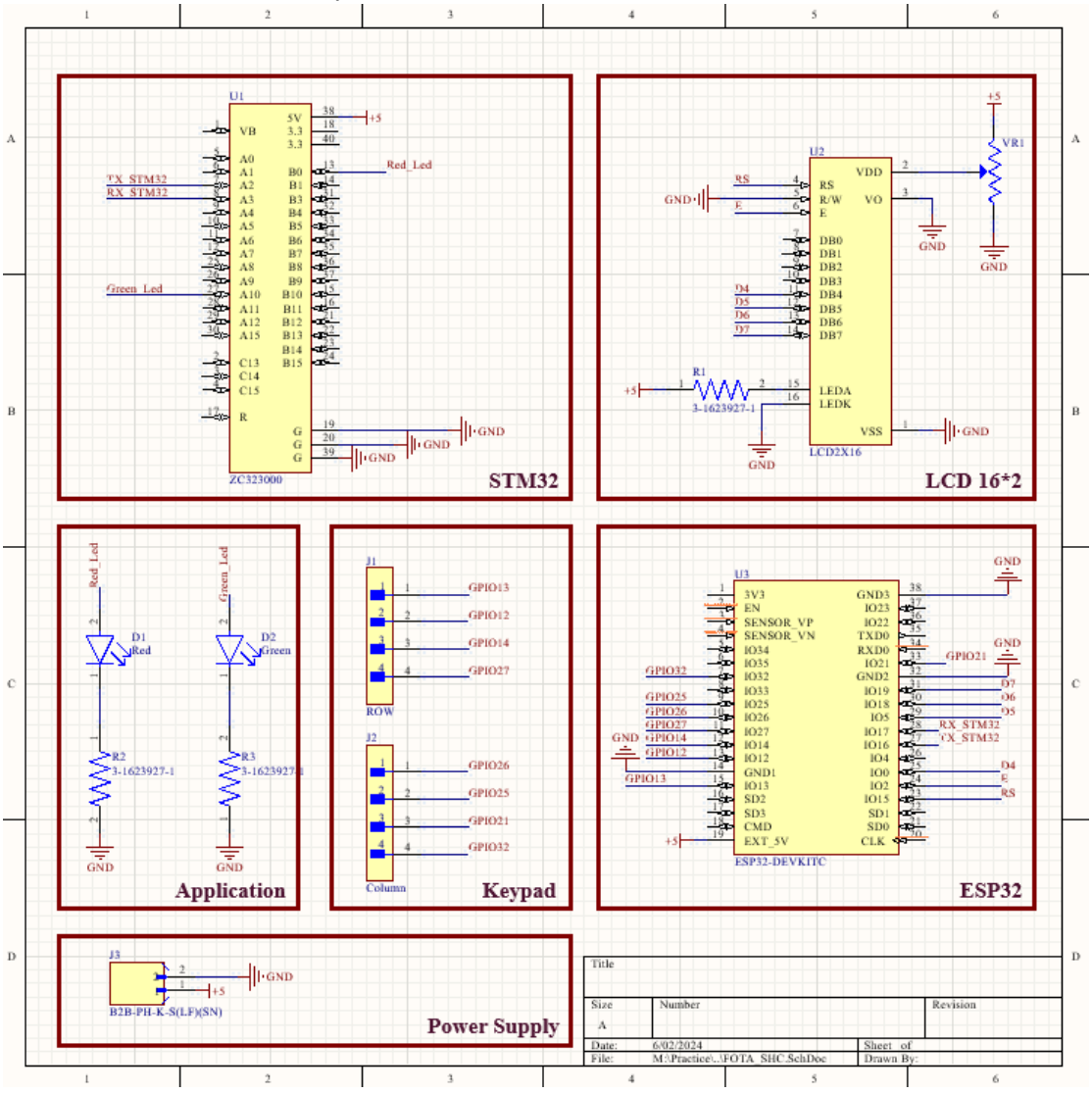


Figure 36. Schematic Sheet

4.4.2 Making Our PCB Board:

After making the schematic, Executed our components to PCB Sheet then:

- Place components beside each other for easier route (As in Figure 10).
- Route all components on Top and Button Layers (As in Figure 11).
- Create a ground plane on the button and Top layer for improved electrical performance and noise reduction (As in Figure 12 and 13).
- Execute our 3D PCB Model (As in Figure 14).

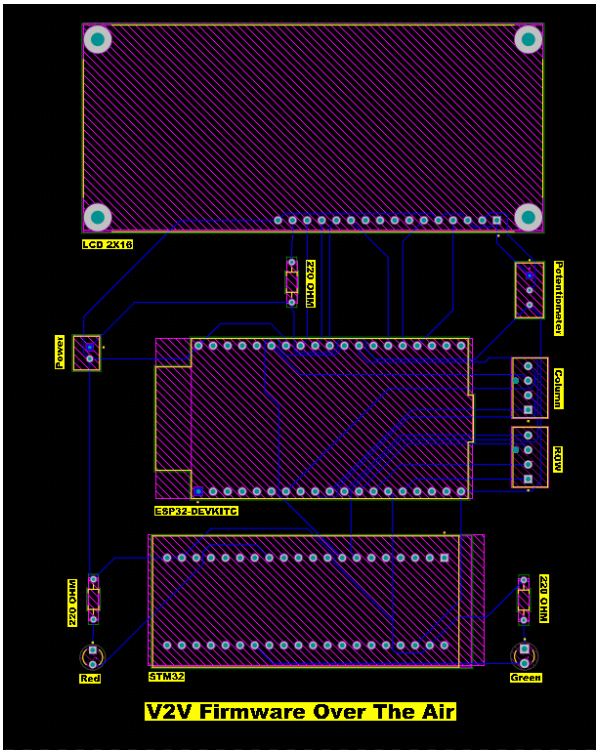


Figure 37. Routing

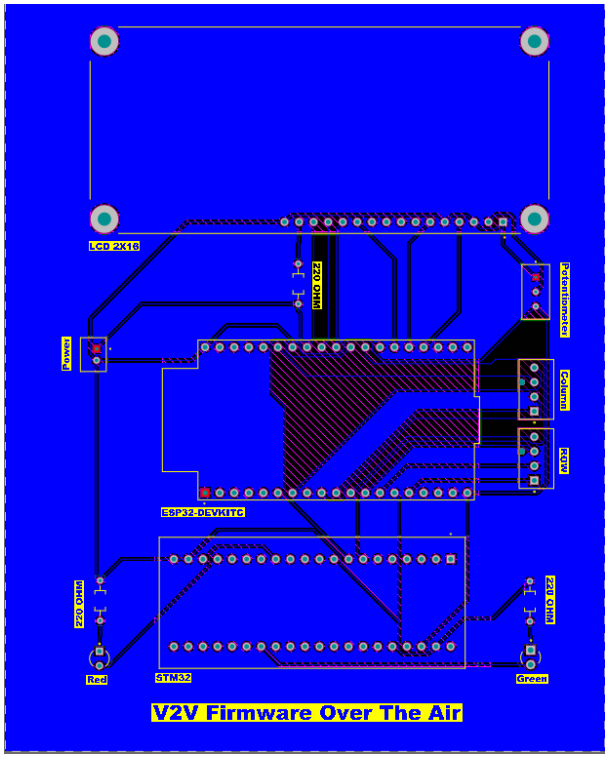


Figure 38. Mask

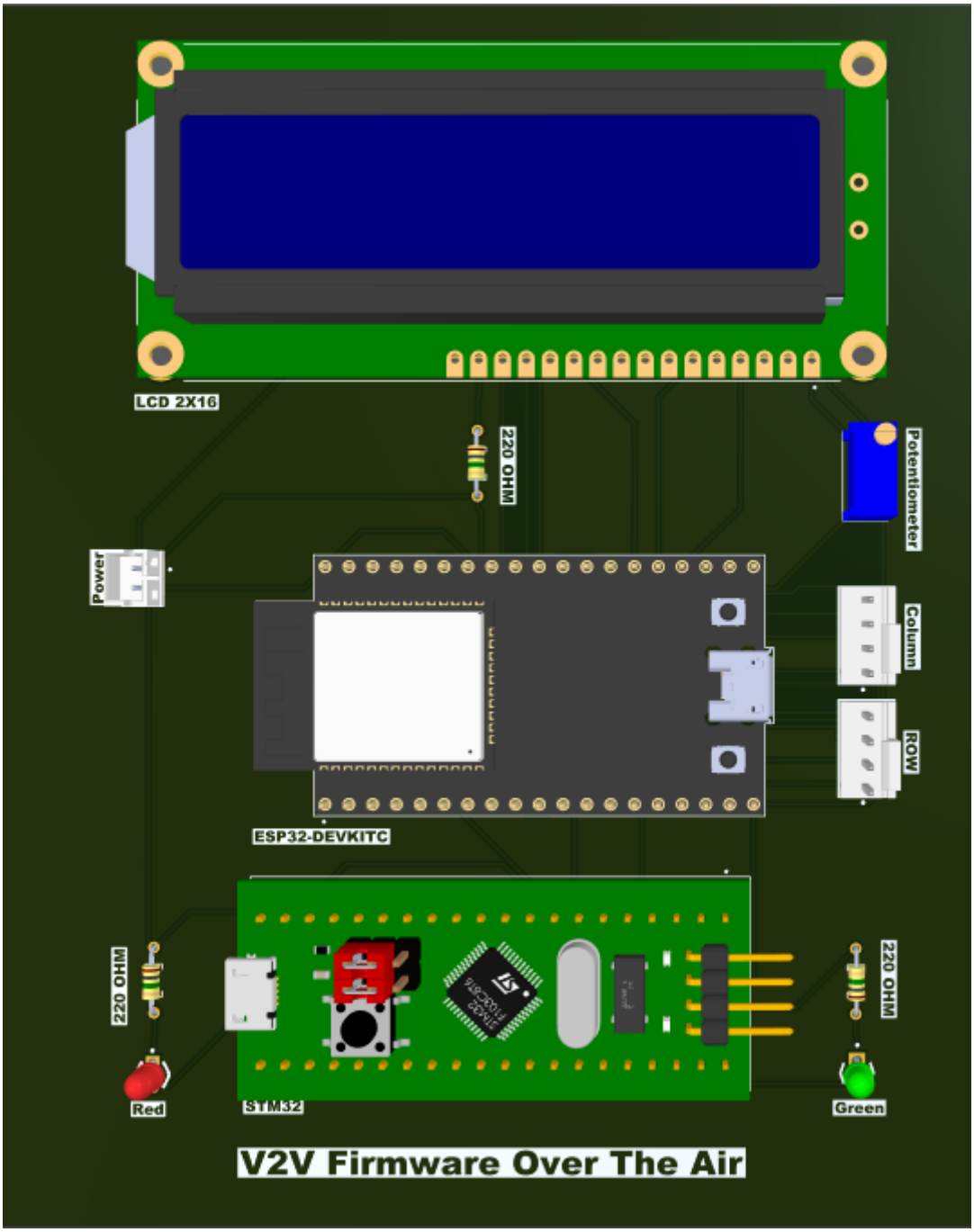


Figure 39. 3D PCB

Chapter 5: Running and Testing

5.1 Working Flow

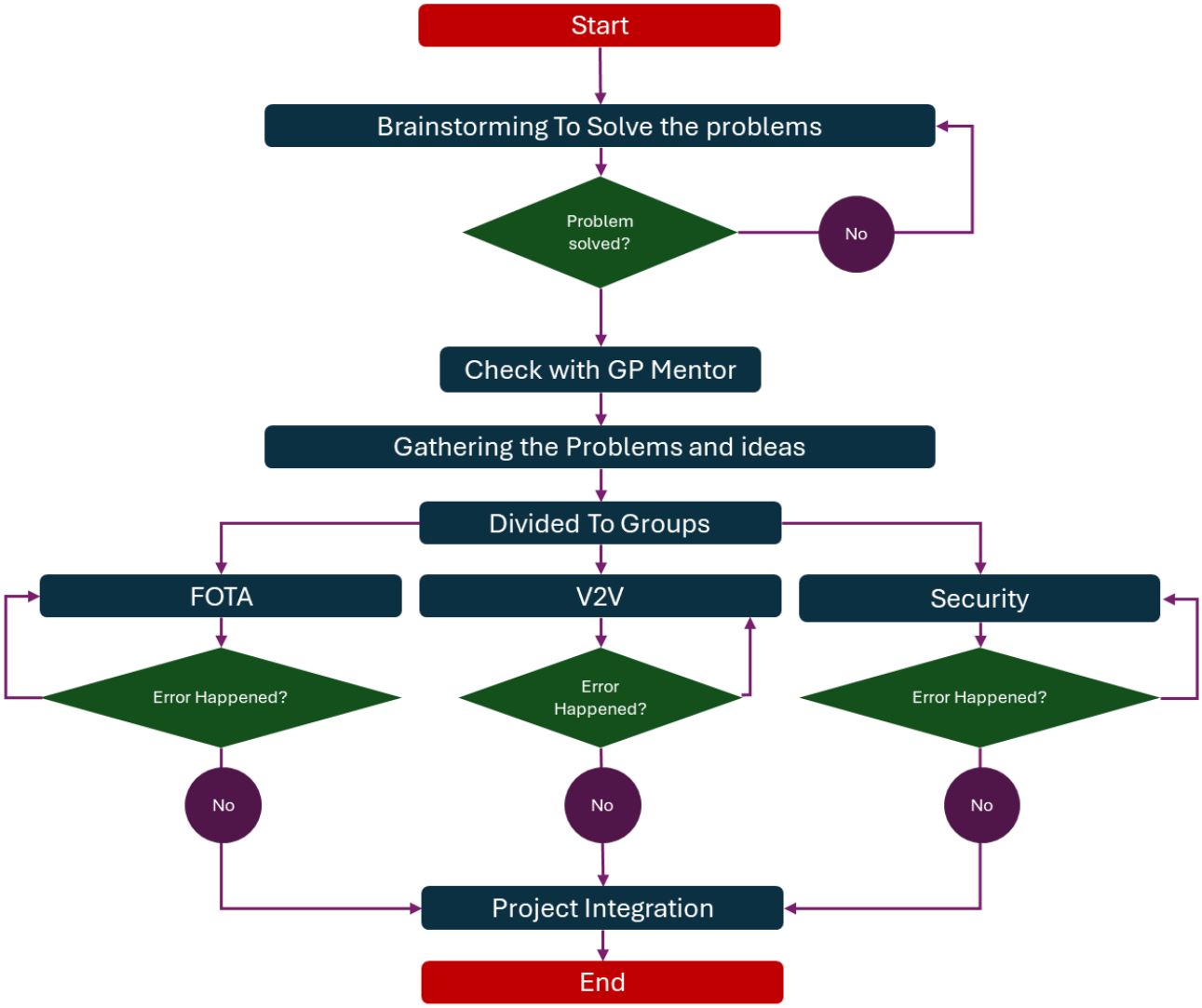


Figure 40. Working flow diagram

5.2 Test cases

Case 1 : When An update occurs through server:

Figure 41. Cars state from firebase server (Both cars not updated)

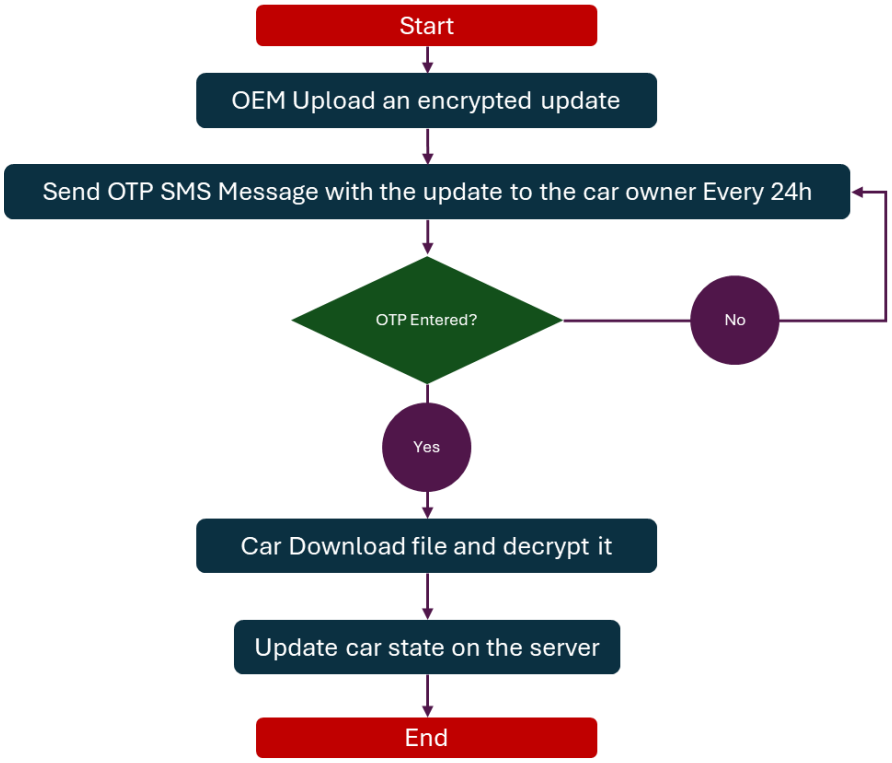


Figure 42. Flow chart for how the car updated from server

Figure 43. Cars state from firebase server (Both cars are updated)

Case 2 : When An update occurs through V2V:

Figure 44. Cars state from firebase server (Car1 updated and Car2 Not updated)

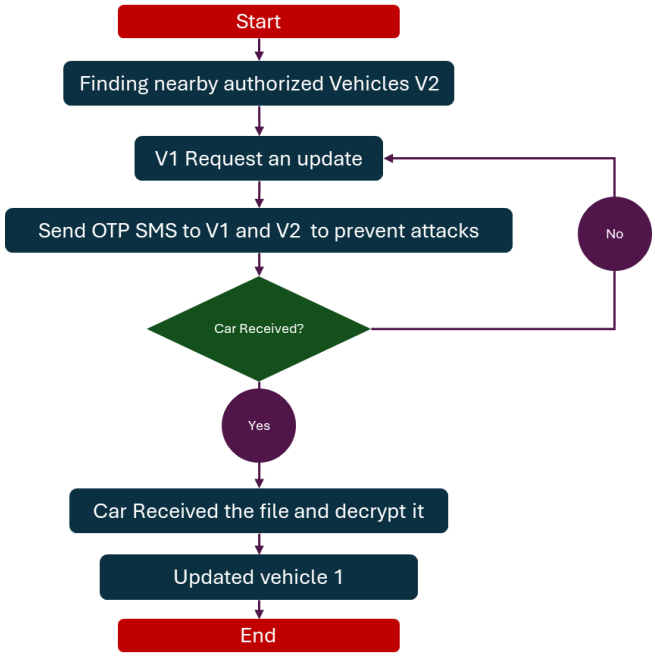


Figure 45. Cars state from firebase server (Both cars are updated)