Mansoura University

Faculty of Engineering

Electronics and communication department

2020-2021

FOTA Project Handbook

(Flash over the air)

Supervisor

Prof. Dr. Sherif Keishk

Prepared by

Abdelrhman Mosad Abdelhady

Mohamed Hafez Mohamed

Osama Salah Hijazi

Abstract

In this age almost everything is connected to internet, and we can see the growth of fields like IoT in our life starting from the appliances at our homes to our cars and phones, and with this growth the number of applications is increasing and the process of developing a stable software is becoming harder and harder and the number of lines of code per application is increasing rapidly.

One domain of applications is having more trouble than the rest and that is the Embedded systems domain; mainly because of the lack of the means to deliver new software updates to the embedded devices through wireless communication and that made the development cycle of embedded software applications takes long time and a long test process to make sure there is no faults or bugs in software as we cannot fix faulty software after shipping.

If we looked at other domains, we will not find this problem and the answer is over air update technology, Mobile and PC companies don’t provide really a complete software free of bugs, their software will be lacking a lot features they intended to do and will have some bugs, but after providing the software they can access the devices and provide updates fixing the bugs or adding new features.

So, in the project we try to accomplish the same concept but with a new domain a more restricted and limited domain the embedded systems domain.

Table of contents

List of Abbreviations

IoT	Internet of things
FOTA	Flash over the air
OTA	Over the air update
ECU	Electronics control unit
SDLC	Software development life cycle
OEM	Original Equipment Manufacturer
CRC	Cyclic redundancy check
UART	Universal asynchronous receiver transmitter
GPIO	General Purpose Input/Output
SPI	Serial Peripheral Interface
API	Application Programming Interface
HLD	High Level Design
SRS	Software Requirements specification
CDD	Component Design Document
BSW	Basic Software
AUTOSAR	Automotive Open System Architecture

Chapter (1): Introduction

1.1. Introduction To FOTA

1.1.1. What is FOTA?

Firmware-over-the-air (FOTA) is a technology that enables the operating firmware of a device to be upgraded and updated wirelessly over the network ("over the air") without the need to connect directly to the device.

FOTA-capable devices can download updates from service providers or manufacturers in short time, depending on file size and connection speed. This saves businesses the time and money spent sending a technician to have each one of their devices physically upgraded or updated.

So simply A flash-over-the-air update is the wireless delivery of new software or data to mobile devices.

1.1.2. FOTA examples

FOTA technology has increased in significance, as mobile devices evolve, and applications emerge. Mobile operators and telecommunication third parties can send OTA updates through SMS to configure data updates in SIM cards; distribute system updates; or access services, such as wireless access protocol.

In a well-known example from 2016, Tesla used FOTA to update each of their cars with the ability to self-park. Without FOTA, each car would have had to either be recalled or visited by a Tesla technician for these updates to be installed.

FOTA is particularly useful when it comes to IoT systems, especially those with large numbers of connected devices that require frequent updates. For example, updating hundreds of sensors measuring soil moisture levels across a large farm would be a near-impossible task using the traditional method, in which each sensor would have to be retrieved, connected to a

computer or handheld device, reprogrammed with the new update, and placed back in field, creating unnecessary costs and performance disruption.

1.2. Flashing Techniques for microcontrollers

So, without FOTA how we can deliver a new software to a device? the traditional method is to connect directly to device through a defined interface and we have different types of connection.

1.2.1. Off-Circuit Programming

This is the basic old technique, here the flash interface is external outside the microcontroller and the flash memory programming pins are connected to some microcontroller pins to be programmed through.

A device called burner is used, usually a hardware kit that sometimes has a socket on which the microcontroller can be placed, or simply connected to the required pins with jumpers, This burner kit includes the flash driver, it also includes the communication port with the computer, So the task of the burner is to get the executable file from the computer via its communication port, then it applies the required high power on the flash memory as if it uploads the file to flash memory.

When the burner finishes uploading the file, the microcontroller can now be removed and connected to the application circuit.

So, to summarize this technique, to program the microcontroller, the microcontroller shall be removed from its application circuit, then connected to the burner which 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.

But what if the process of removing the microcontroller from its application circuit for reprogramming is not that easy?! For example, in firmware update applications, where the new program shall be uploaded to a lot of microcontrollers inside their machines, this will be a big time and effort consuming task.

Figure 1.1

1.2.2. In-Circuit Programming

Also called In System Programming (ISP), here the microcontroller has an internal flash interface, that can generate any necessary programming power from main power supply to provide the required high power for the flash programming, which is inside the microcontroller also, so the flashing process is done inside the microcontroller which means there is no need to remove the microcontroller from the system circuit for programming, hence its name.

Of course this feature has provided a lot of advantages as now the microcontroller can be programmed while installed in a complete system, rather than requiring the chip to be programmed prior to installing it into the system, hence it allows firmware updates to be delivered to the on-chip flash memory of microcontrollers without requiring specialist programming circuitry on the circuit board, which simplifies design work, also it allows manufacturers of electronic devices to integrate programming and testing into a single production phase, and save money.

But how to communicate with this on-chip flash driver i.e., how to send it the required executable file?

Firstly, the flash driver is needed to communicate with the external world to get the required application code, this is done by serial communication, so the flash driver manufacturer defines one or more communication protocols through which the flash driver can interface, for example, stm32f103 there are 2 protocols that are provided to communicate with the in-circuit flash programmer: JTAG and SWD.

This technique we use the ST-Link programmer and debugger as shown in figure 1.1, and this device works as a translator from the USB protocol to the SWD protocol, Note that this device is target-specific which means that changing the microcontroller to another one with different flash interface communication protocol will result in changing the translator device, this will be a big headache if it is required to reprogram a number of different microcontrollers in the same system, i.e. a car with 100 ECUS (electronic control units), this is one of the main reasons why the bootloader is used.

Figure 1.2

1.2.3. In-Circuit Programming with bootloader

Booting (also known as booting up) is the initial set of operations that a computer system performs when electrical power is switched on.

The processor initializes a communication port in the microcontroller to make it ready to receive data, a computer starts sending the application code through this communication port to our microcontroller, the processor reads the data, then sends it to the in-circuit flash interface which in turn writes this data in the flash, eventually the target controller is updated with the code.

So now, instead of different interfaces with different microcontrollers we use bootloader to create unified interface with the system, for example if we have 100 ECU in a system, we will design a bootloader for each of them with standard communication protocol like UART or Ethernet.

1.3. Software development life cycle

Consistently developing a quality embedded system within budgetary and timing constraints is a challenging endeavor and good software cycle will make this challenge easier.

1.3.1. What is software development life cycle?

The Software Development Life Cycle can be defined as: “The processes used to consistently achieve the desired software quality for a system within reasonable budgetary and timing constraints”.

The SDLC defines processes and procedures that help teams to avoid common pitfalls that would otherwise result in software rework and debugging. We all know that rework and debugging can have significant business ramifications such as being late to market, budget overages and customer backlash just to name a few. The SLDC is meant to help teams go faster not slow them down!

1.3.2. Software development life cycle phases

The SDLC defines processes that fall across several different development phases. These phases have traditionally consisted of the following:

- Requirements
- Design
- Coding
- Testing
- Deployment

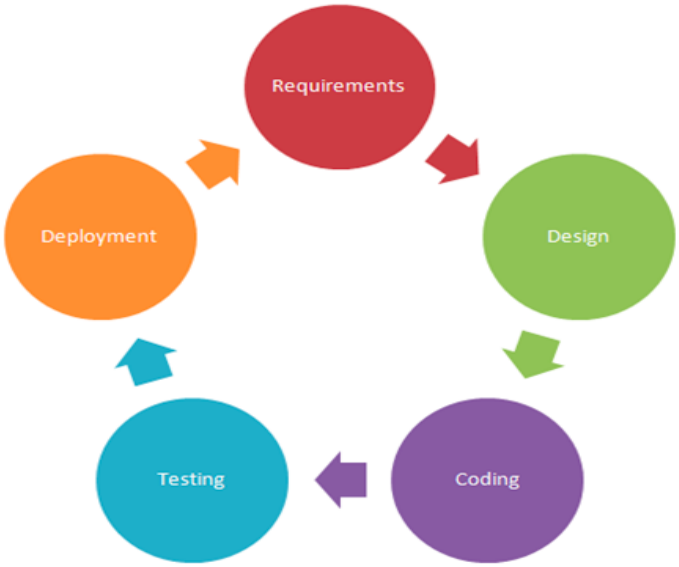

Figure 1.3

Note that in typical embedded device the cycle stops at deployment of the code and there is no room for maintenance in the cycle because normally we cannot access the device after shipping and therefore, we cannot fix bugs.

For this reason, the testing phase will consume a long time to ensure that the code is free of bugs and function correctly.

Development too will consume a long time as we must implement all the features before shipping.

1.3.3. How to improve development life cycle

When applying FOTA concept with embedded device we can add maintenance to the development life cycle which will improve the code quality and will add more value, like the mobile applications which get to market without finishing all features and these features will be added later with OTA technology.

Chapter (2): Problem definition, challenges

2.1. Problem statement

During the past twenty-five years, computer-based electronic control units (ECUs) have gradually replaced many of the mechanical and pneumatic control systems in vehicles. A 2013 study released by Frost & Sullivan found that mass market cars by then had at least 20 million to 30 million lines of software code, while premium cars could have as much as 100 million lines controlling essential systems. According to Frost & Sullivan, the average cost of the software code is $10 per line, and it is steadily increasing. They estimate that by 2020 that the amount of software will increase by as much as 50 percent.

It is essential for vehicle OEMs to manage the software efficiently over the lifecycle of the vehicle, both to provide improvements in performance and to deliver corrections to faulty software that endanger lives or the environment and which could result in expensive product recalls. It is estimated that between 60 and 70 per cent of vehicle recalls in North America and Europe today are due to software problems, so this issue is clearly one that must be addressed aggressively by the OEMs. A topical case in point is Volkswagen and the eleven million diesel vehicles it has sold during the past eight years with _faulty' emissions control software. A portion of these vehicles also require hardware changes, but if VW could have corrected the problem with only a software update, the process would have taken far less time, cost much less and reduced the environmental impact.

FOTA is already in use among some vehicle OEMs (e.g., Tesla and Mercedes-Benz) as an alternative to performing the software updates using a vehicle workshop system, however this practice is still very new and limited. In a future OTA scenario, whether the updates are performed in the workshop or at the dealer with OTA technology, or remotely, there remain both technical and business reasons for using the dealer network for performing the updates. Unlike a company like Tesla Motors, which sells cars directly to customers and which does not have a dealer network, vehicle OEMs are dependent on their dealers and National Sales Companies for customer contact information. Many, if not most, car OEMs do not have a central database containing the names and

most recent contact information on the owners or drivers of their vehicles. Dealers with workshops want to continue to have the direct relationships with customers to sell service and accessories, and to eventually sell the customer a new car. They will resist any attempt to short circuit them.

2.2. Major challenges

Based on this high-level description of the OTA update process, major challenges arise that the OTA update solution must address:

- Challenge related to memory.
- Challenge in communication.
- Challenge in security.
- Challenge in design
- Challenge in systems that have many ECUs

2.2.1. Challenge related to memory

The software solution must organize the new software application into volatile or nonvolatile memory of the client device so that it can be executed when the update process completes. The solution must ensure that a previous version of the software is kept as a fallback application in case the new software has problems. Also, we must retain the state of the client device between resets and power cycles, such as the version of the software we are currently running, and where it is in memory.

2.2.2. Challenge in 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.

2.2.3. Challenge in security.

With the new software being sent wirelessly from the server to the client, we must ensure that the server is a trusted party. This security challenge is known as authentication. We also must ensure that the new software is obfuscated to any observers, since it may contain sensitive information. This security challenge is known as confidentiality. The final element of security is integrity, ensuring that the new software is not corrupted when it is sent over the air.

2.2.4. Challenge in design

The software must be designed to be highly portable to different microcontrollers and must be configurable to change smoothly to different situations, if we needed to change the communication protocol for example, we do not need to rewrite the whole code.

The design must be reliable to ensure correct functionality.

2.2.5. Challenge in systems that have many ECUs

In systems like cars, we have around 100 – 200 ECU so we must have some sort of a system that mange the coming update and can forward it to the right ECU, this system will be probably an ECU with high functionality and will manage everything related to FOTA

This ECU in the automotive domain called FOTA master because it controls the whole process.

2.3. FOTA Requirements

1 - Safety

No image can be downloaded until it has been verified as the correct image for the specific ECU; utilizing CRCs, checksums, and other release version compatibilities used.

CRC is a powerful check mechanism to make sure that the new update is correct, and the flashing process is successful.

2 - Security

Data should be sent utilizing security protocol and encryption, for security of the code it will be encrypted and then will be decrypted at the devise before flashing the update.

Protocols used in wireless communication must be secure.

3 - Reliability

Should make sure the data is transferred from one ECU to another ECU is correct and must wait till all data is transferred first before processing the data, so a reliable communication protocol must be used.

4 - Scalability

Should be designed to support large number of embedded devices by providing multicast and broadcast capabilities in addition to unicast.

The software must be designed in a way to support scaling too.

5 - Flexibility

Should support national and regional requirements. Should provide an efficient mechanism to support multiple devices.

2.4. FOTA Benefits

1. Cost efficient

OEMs can seamlessly manage firmware updates across a fleet of IoT devices from one unified interface. The costs significantly decrease over the entire lifecycle of the product, and no needs to recalls and in-person maintenance.

2. Continuous improvements

Bugs can be fixed, and product behavior can be enhanced after the device lands in the hands of your consumers. This can potentially eliminate costly recalls and in-person maintenance.

For example, if device have a major bug which affect the performance of the device instead of collecting all the devices to fix the bug, the update will be sent through any wireless communication.

3. Improved scalability.

FOTA updates enable manufacturers to add new features to infrastructure after the release without physical access to upgrade firmware, which will lead to a better user experience.

Features can be added like in cars for example auto parking feature can be added without problems.

4. Faster time-to-market.

Developers can test new features on selected devices and deploy frequently knowing that the products will remain stable. Firmware updates can be dispatched even while the vehicle is still on the production line or in dealer center.

5. Improved safety and compliance.

OEMs can use updates to patch the known vulnerabilities, e.g., defective adapter plugs, instead of recalling the vehicles. Your business can respond to the legal and regulatory responsibilities faster and more cost-effectively.

6. Better software quality.

Updates allow you to continuously amend your systems. Whenever you discover a new opportunity for improvement (e.g., a way to reduce vehicles fuel consumption), you can instantly deliver it to customers, instead of waiting to incorporate it in a new batch of vehicles.

8. Two-way communication.

OTA data exchanges can happen both ways. By collecting data on the vehicle usage or performance and deploying analytics tools, OEMs can promote better customer experience and show customers that they care about them by issuing regular system updates. Additionally, you can gather intel for R&D and ensure preventive maintenance. And you can enrich your solutions with additional data sources or white-label solutions purchased through this service.

Chapter (3): System design

3.1. Overall system view

3.1.1. An abstract view

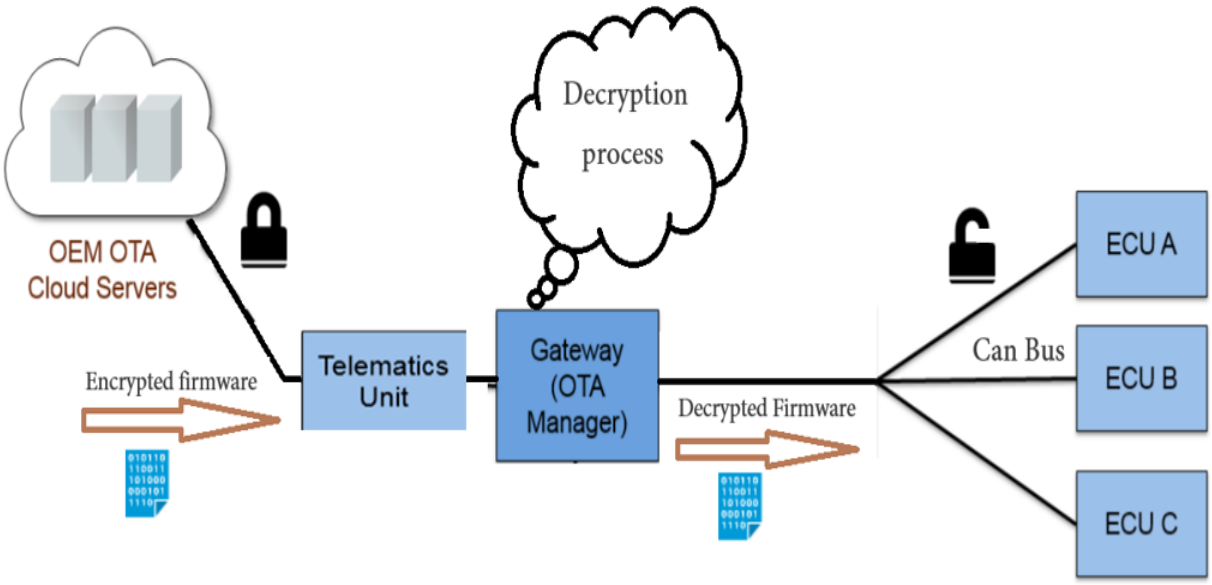

Figure 3.1

In (Figure 3.1) we can see the overall view of the system; in an abstracted way we see four main components:

1 - OEM cloud server

It is where we will upload the new update through a defined user interface and will have the needed information about the products and the new update.

2 - Telematics unit

It is our way to connect to server and download the code it has the capability to connect to Wi-Fi for example to has access to internet and will access the server easily.

3 - Gateway

Will receive the update from telematics unit and important data about the update like the CRC and target ECU ID and will decrypt the code before sending it to the target ECU.

4 - The target ECU

It is the ECU which the update is directed to and connected to the gateway through a communication bus like UART or CAN.

There is a fifth hidden component which is crucial for the whole process, it is the bootloader. A bootloader which supports FOTA updates must be installed in the ECU before shipping the product, the definition and mechanism of bootloader will be discussed later.

The OEM will have to a defined interface to access the server and manage the updates easily. A graphical user interface will make it easier and more reliable.

3.1.2. The view implemented

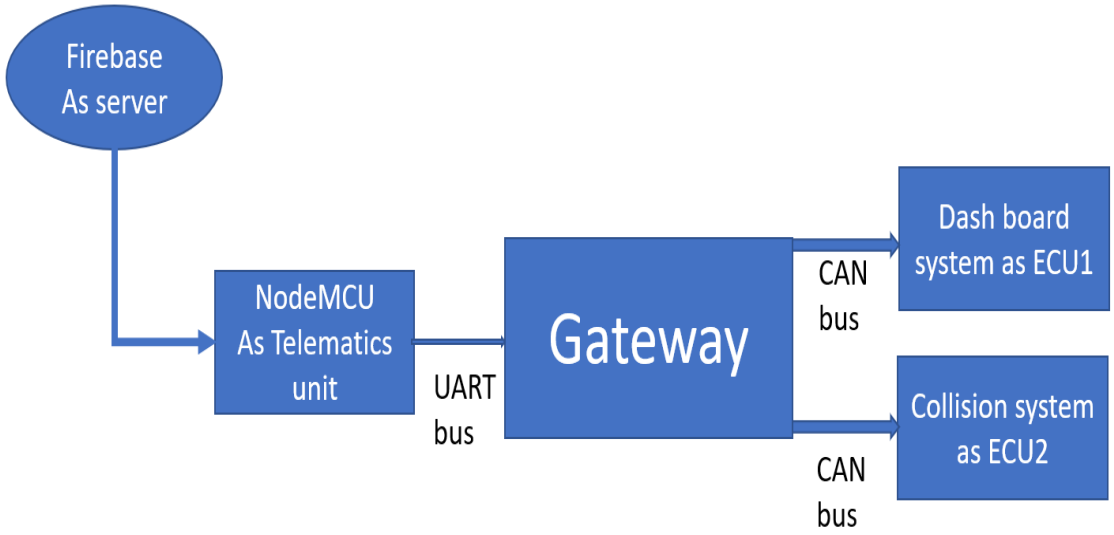

Figure 3.2

For the server firebase is used as it provides real time database and free large storage, and it is a google platform.

NodeMCU is used as a telematics unit it has a Wi-Fi module which will make it easy to connect to the server and will be connected to gateway through a UART bus.

For test cases we implemented two systems used in cars a dashboard system and collision system both ECUs is connected to CAN bus to access the gateway. And both use stm32f103 microcontroller.

Gateway also will be implemented using the same microcontroller as it has the required communication protocols, and the micro has the required processing power.

3.2. System components

The system can be divided into 4 major subsystems and all these systems are connected through a defined protocols like UART or CAN, these systems are:

- Telematics unit system.
- Gateway system.
- Dashboard system.
- Collision system.

3.2.1. Telematics unit system

The major component is NodeMCU, it is an open-source development board and firmware based in the widely used ESP8266 -12E Wi-Fi module. It allows you to program the ESP8266 Wi-Fi module with the simple and powerful LUA programming language or Arduino IDE.

With just a few lines of code you can establish a Wi-Fi connection and define input/output pins according to your needs exactly like Arduino, turning your ESP8266 into a web server and a lot more. It is the Wi-Fi equivalent of ethernet module. Now you have internet of things (IoT) real tool.

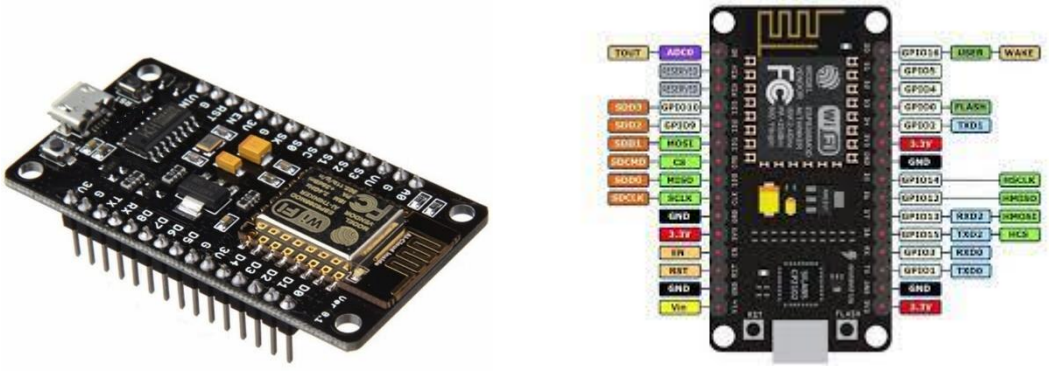

Figure 3.3

Features

- Programable Wi-Fi module.
- Arduino-like (software defined) hardware IO.
- Can be programmed with the simple and powerful Lua programming language or Arduino IDE.
- USB-TTL included plug & play.
- 10 GPIOs D0-D10, PWM functionality, IIC and SPI communication, 1-Wire and ADC A0 etc. all in one board.
- Wi-Fi networking (can be used as access point and/or station, host a web server), connect to internet to fetch or upload data.
- Event-driven API for network applications.
- PCB antenna.

3.2.2. Gateway system

One of the most important systems in FOTA, it has a lot of functions like decryption of the code, forward the update to the right ECU and make sure that the update completed successfully to inform back the server, one of the important functions is to keep track of software versions and mange rollback in case of code failure in one of the ECUs.

Gateway will also inform the user of new update to give the user the choice to accept or reject the update request, that will be achieved through a graphical interface, if the user reject the update the server must drop the update for some time, if user accepted the update the updated will be downloaded and processed.

Gateway is the connecting point between the telematics unit and the target ECU and must be designed in scalable efficient way.

So, to achieve these features gateway has 3 main components:

1. STM32f103c8t6 microcontroller
2. TFT display
3. Buttons

1 - STM32f103c8t6 microcontroller

The STM32F103C8T6 is a medium density performance line, ARM Cortex-M3 32bit microcontroller in 48 pin LQFP package. It incorporates high performance RISC core with 72MHz operating frequency, high speed embedded memories, extensive range of enhanced I/Os and peripherals connected to two APB buses. The STM32F103C8T6 features 12bit ADC, timers, PWM timer, standard and advanced communication interfaces. A comprehensive set of power saving mode allows the design of low power applications.

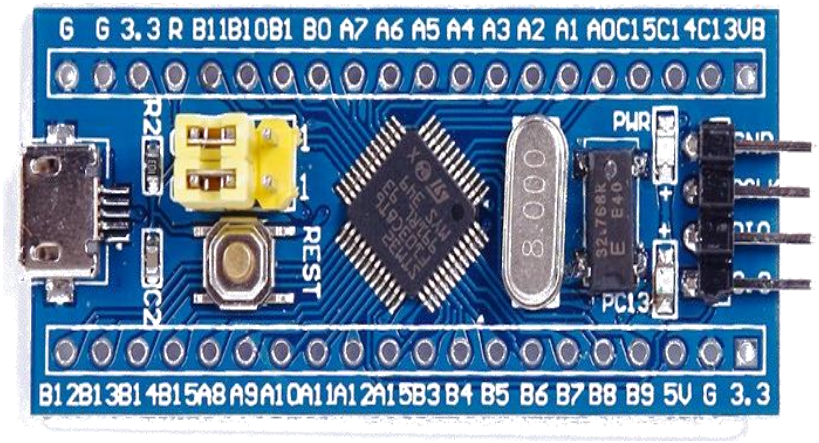

Figure 3.3

2 - TFT display

A thin-film-transistor liquid-crystal display (TFT LCD) is a variant of a liquid-crystal display (LCD) that uses thin-film-transistor (TFT) technology to improve image qualities such as addressability and contrast. A TFT LCD is an active-matrix LCD, in contrast to passive matrix LCDs or simple, direct-driven LCDs with a few segments.

Features of TFT Displays

- Bright LED Backlights. The bright LED backlights that are featured in TFT displays are most often used for mobile screens.
- Deliver Sharp Images.
- Stable and Much Quicker Response Times.

- Wide Viewing Angles.
- Better Color Reproduction and Representation

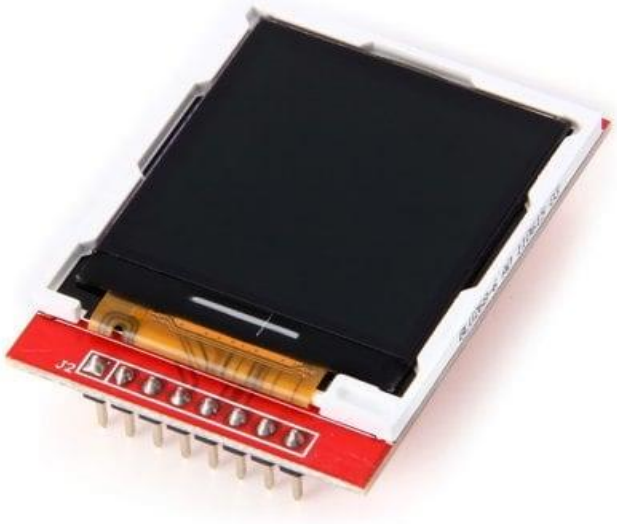

Figure 3.4

2 - Buttons

A push-button (also spelled pushbutton) or simply button is a simple switch mechanism to control some aspect of a machine or a process. Buttons are typically made from hard material, usually plastic or metal. The surface is usually flat or shaped to accommodate the human finger or hand, to be easily depressed or pushed.

It is the user way to interact with the gateway to accept or reject the update.

Figure 3.5

3.2.3. Dashboard system

It is ECU1 which will be implemented and used as a test case for FOTA, A dashboard is a control panel set within the central console of a vehicle or small aircraft. Usually located directly ahead of the driver, it displays instrumentation and controls for the vehicle's operation.

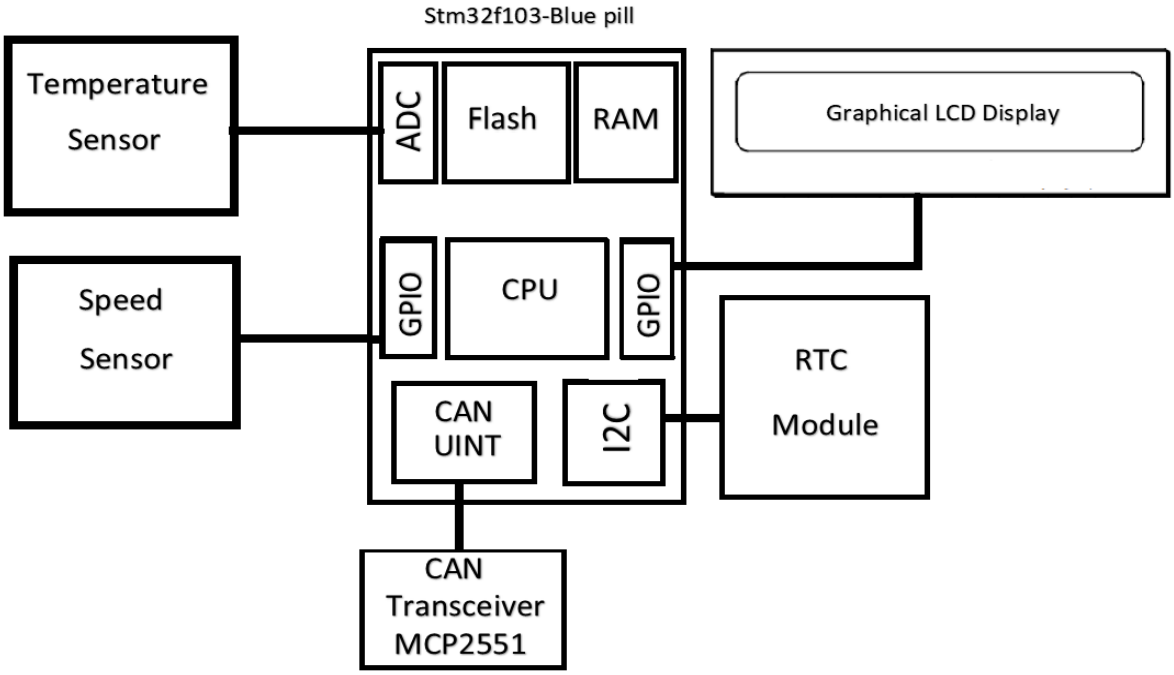

Figure 3.6

The dashboard we designed will show temperature, speed, and time to do so the components used are:

1. Graphical LCD
2. RTC module
3. Speed sensor
4. Temperature sensor
5. STM32f1c8

1 - Graphical LCD:

A Graphic LCD display is just as its name implies. This LCD module can display images, letters and numbers that are generated through the customer's software. Dot Matrix displays are identified by two sets of numbers. An example of this is a 128 x 64. This display contains 128 dots along the X axis, or horizontal, and 64 dots along the Y axis or Vertical. Each of these dots, sometimes referred to as a pixel, can be turned ON and OFF independently of each other. The customer makes use of software to tell each dot when to turn ON and OFF. This is like the old 'light bright' toy.

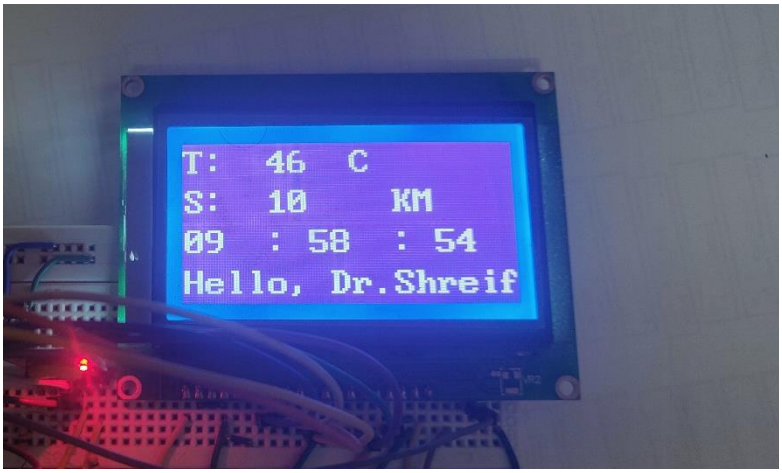

Figure 3.7

2 - RTC module:

Stands for real time clock used to calculate the time and usually accessed by an I2C protocol.

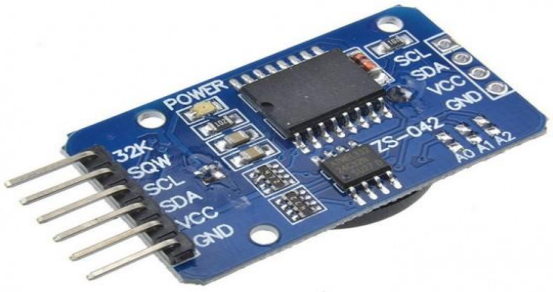

Figure 3.8

3 - Temperature sensor:

A temperature sensor is an electronic device that measures the temperature of its environment and converts the input data into electronic data to record, monitor, or signal temperature changes. There are many different types of temperature sensors.

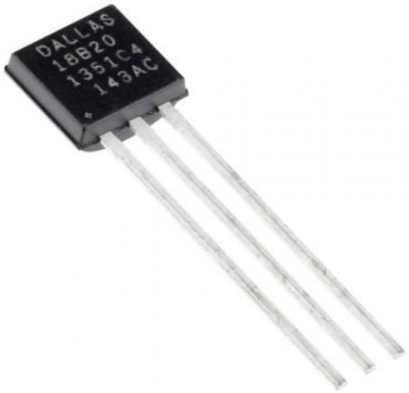

Figure 3.9

3.2.4. Collision system

It is ECU2 which will be implemented and used as a test case for the FOTA, A collision system is a simple system which warns the driver in case of the driver is about to collide with another car or object.

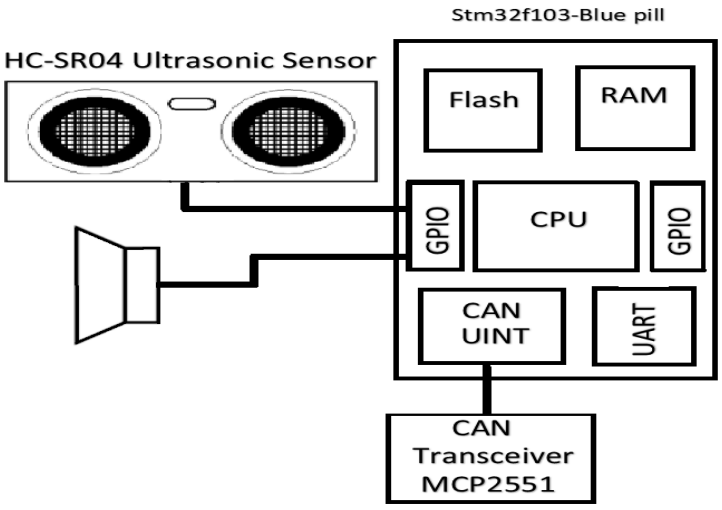

Figure 3.10

The used components:

1. Ultrasonic module
2. Buzzer
3. STM32F103c8

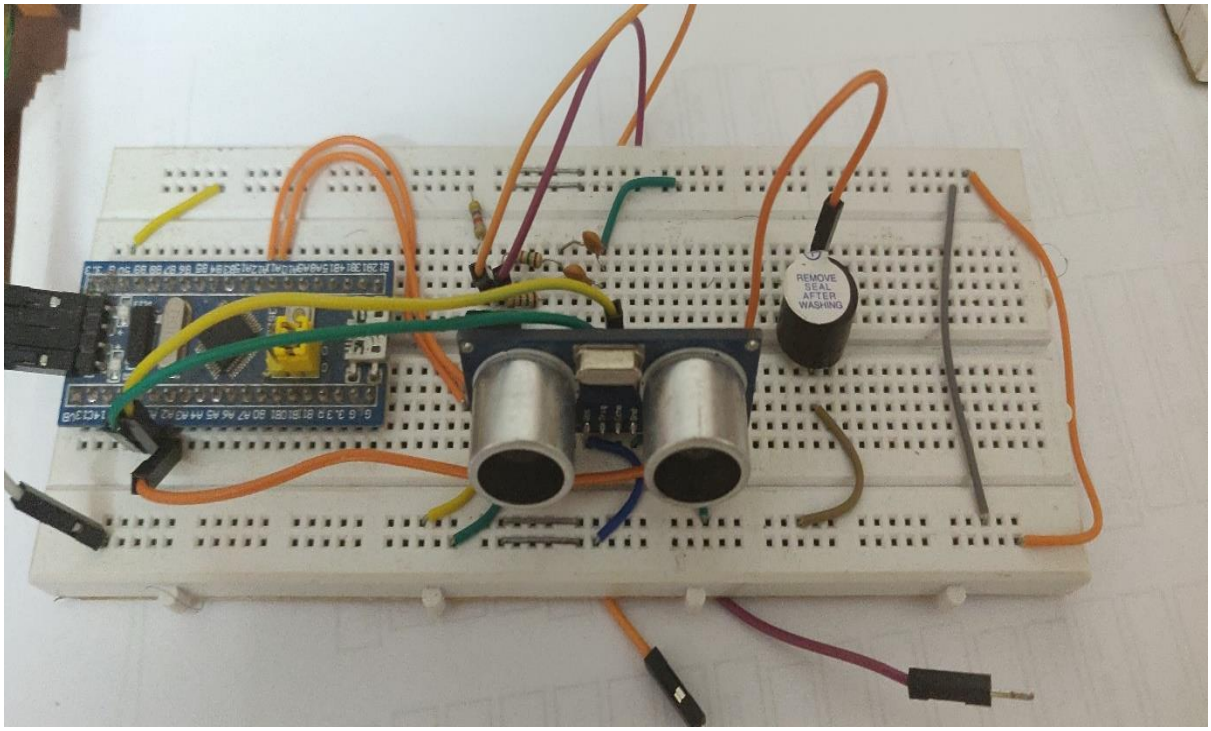

Figure 3.11

Ultrasonic module can detect existence of an object by sending ultrasound signals and monitoring its reflection. It is one of the most common techniques to detect an obstacle in front of a robot. Another known usage is to measure the distance from an object. The alarm in your car that warns you when you get close to an obstacle during the parking process, is mainly implemented using these sensors.

When the ultrasonic sensor detects an object, the buzzer will fire a sound to warn the driver when the object is out of range the buzzer will stop.

Ultrasonic ranging module HC - SR04 provides 2cm - 400cm non-contact measurement function, the ranging accuracy can reach to 3mm.

3.3. Software design

3.3.1. Software design process

Software design is the process of envisioning and defining software solutions to one or more sets of problems.

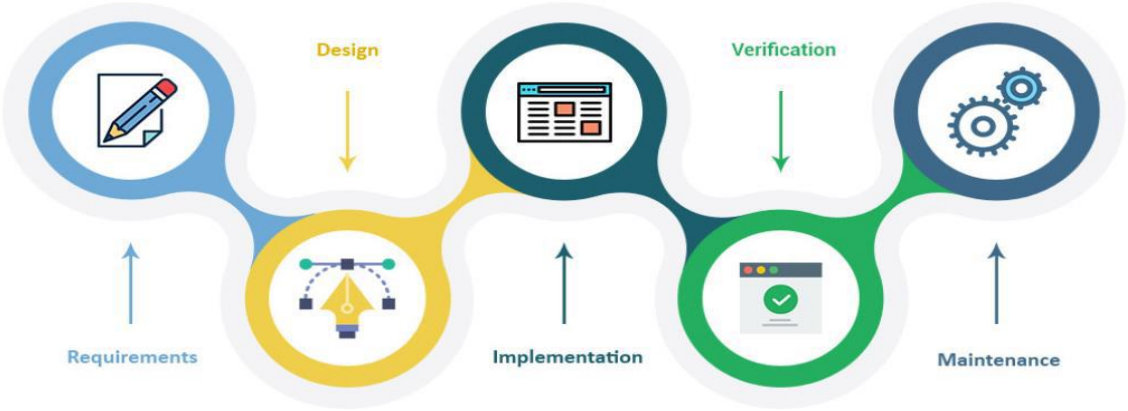

Figure 3.12

3.3.2. Software design approaches

Bottom-up design

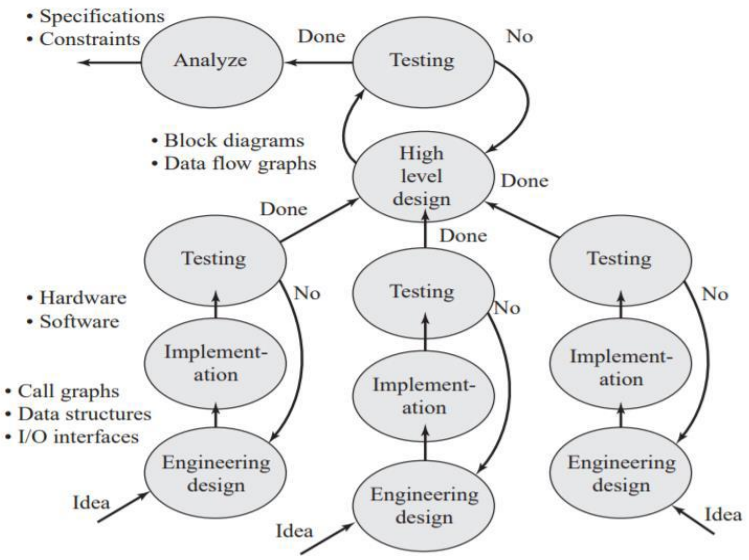

Figure 3.13

Top-down design

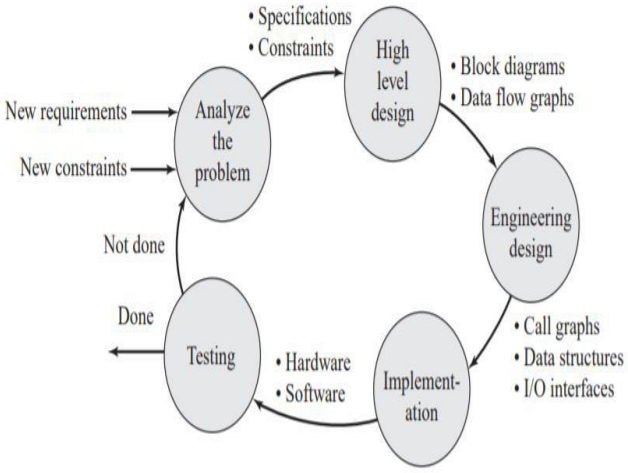

Figure 3.14

3.3.3. Software characteristics

- Compatibility **(**backward compatible**)**
- Extendibility (add new code)
- Modularity (drag and drop)
- Maintainability (easy for debug)
- Reusability (Do you can use it in different project?)
- Robustness (work hard)
- Usability (user interface)
- Scalability (increase input data)

3.3.4. Software design types

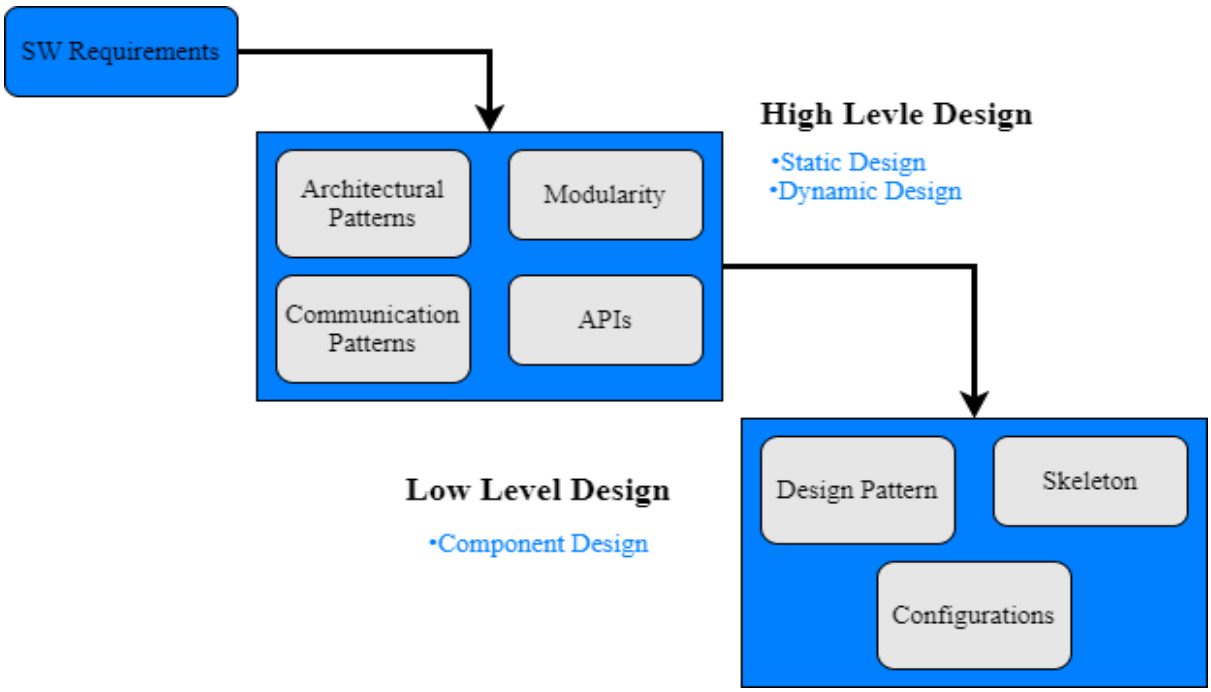

Figure 3.15

3.3.4.1. High level design

The system engineer analysis customer requirements and extract the software specifications and writes it in a document called SRS.

The architect takes the SRS and put his imagination about the system software from the point view of static design, physical design, and dynamic design.

1- Static design

It represents some important concepts like modularity and architectural patterns.

- Modular programming define how SW system is divided into SW components.
- Architectural patterns: define how to organize modules.
  - Layered architecture.
  - Event triggered.
  - Micro services.

2- Physical design

What ‘s the role of physical design?

- Design folder structure for the project.
- Design make system for the project.
- Design the output from the project.
- Define files dependencies and implement constraints on them.

How physical design reduces time to market?

- It optimizes compilation time.
  If a change occurred in the logic of a module, you do not have to recompile the hole project files again but recompile the changed module.

- It optimizes liking time.
  By creating a library for each folder. Link the object files in each folder into one file. At the final stages, link all libraries. So, if a change occurred in a folder, you recompile these files and link them to produce a library to be linked with other libraries.

- It optimizes the flashing timing by dividing the output file and define addresses for each part and if a change occurred, flash the changed part only.

3- Dynamic design

- Dynamic Design always answer how the modules talk with each other by implementing communication pattern which define 2
- It defines when module APIs' will be triggered.
  - Event or Time triggered.
  - Client server.
  - Service pr Signal oriented.

3.3.4.2. Low level design (Logic design)

Logical design is that which pertains to language constructs such as classes, operators, functions, and so on. For example, whether a particular class should or should not have a copy constructor is a logical design issue.

- Module skeleton: getter API to get data, setter API to set data and main function that run module state machine.
- Design pattern: implement the actual logic of the module.
- Configuration: configure configuration parameters.

The designer writes CDD that explain the logic of the module.

The developer takes the CDD and writes the code that match this logic.

3.3.5. **AUTOSAR Architecture**

AUTOSAR (AUTomotive Open System Architecture) is a worldwide development partnership of vehicle manufacturers, suppliers, service providers and companies from the automotive electronics, semiconductor and software industry.

AUTOSAR aims to standardize the software architecture of Electronic Control Units (ECUs). AUTOSAR paves the way for innovative electronic systems that further improve performance, safety, and security.

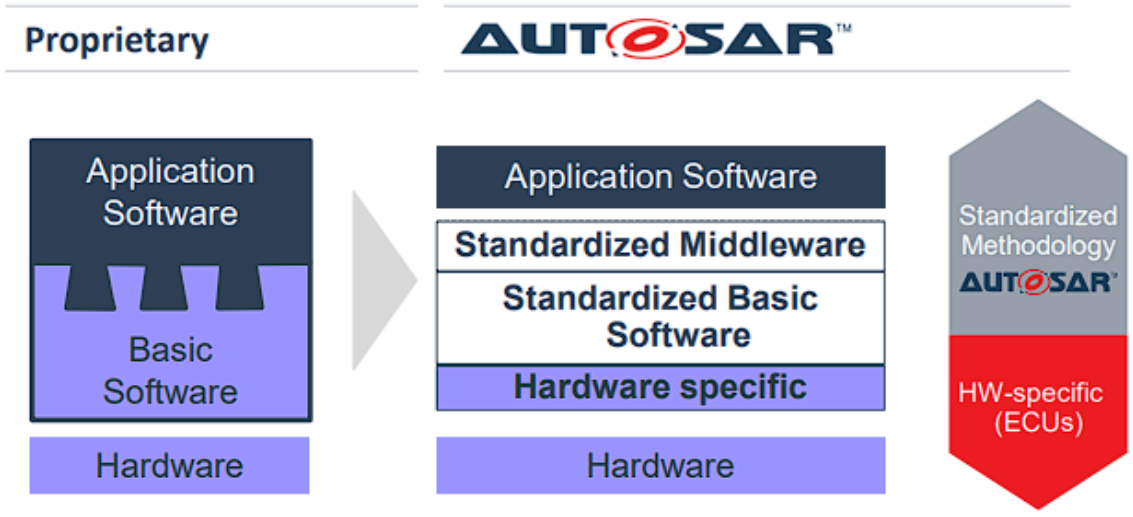

Figure 3.16

3.3.5.1. Significant benefits of standard

• Hardware and software – widely independent of each other.

• Development can be decoupled (through abstraction) by horizontal layers, reducing development time and costs.

• Reuse of software enhances quality and efficiency.

• Establish development distribution among suppliers.

• Compete on innovative functions with increased design flexibility.

• Simplify software and system integration.

• Reduce overall software development costs.

• Enable more efficient variant handling.

• Reuse software modules across OEMs.

• Increase efficiency of application development.

• Invent new business models.

• Interface with development processes.

• Embed tools into an overall tool environment.

• Enable new business models by means of standardized interfaces.

• Easily understand how automotive software is developed.

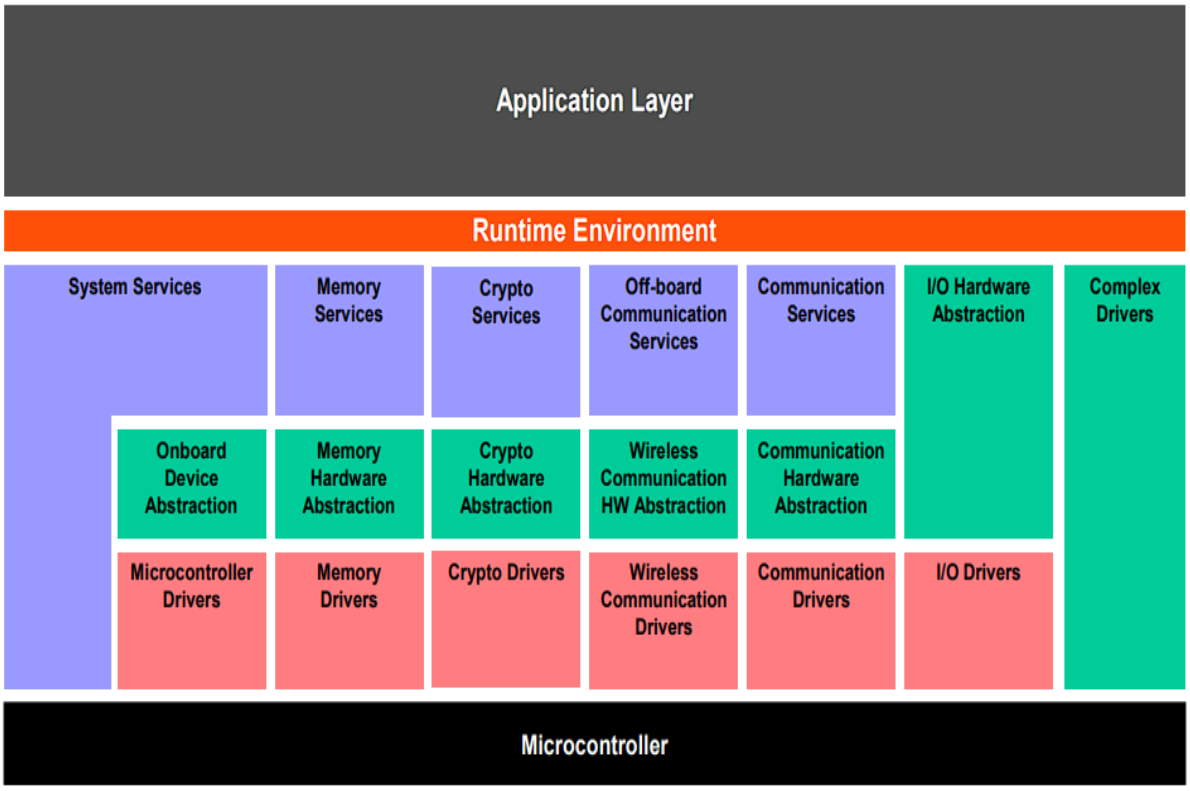

Figure 3.17

3.3.5.2. RTE layer

The Run-Time Environment (RTE) is at the heart of the AUTOSAR ECU architecture. The RTE is the realization (for a particular ECU) of the interfaces of the AUTOSAR Virtual Function Bus (VFB). The RTE provides the infrastructure services that enable communication to occur between AUTOSAR software-components as well as acting as how AUTOSAR software-components access basic software modules including the OS and communication service.

The RTE encompasses both the variable elements of the system infrastructure that arise from the different mappings of components to ECUs as well as standardized RTE services.

In principle the RTE can be logically divided into two sub-parts realizing:

- the communication between software components
- the scheduling of the software components.

3.4. Software design of dashboard

3.4.1. Layered architecture

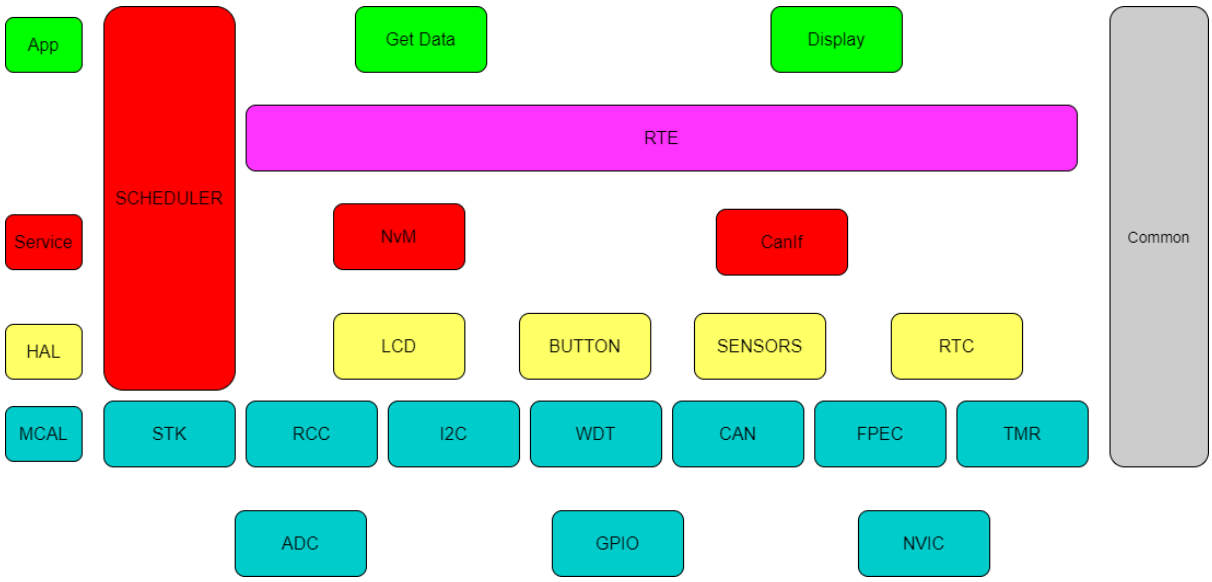

Figure 3.18

3.4.2. Dynamic Logic design

There are more modules in the project so, the best way is to use a scheduler to handle all tasks of the system. This means we used time triggered as a design for this project. Each task (main function of the Application module that run the module logic) has a release time and a period for executing.

We used a RTE layer to abstract the application from the lower layers (BSW)

So, we can move any module of application to another ECU with change in RTE only.

Chapter (4): Implementation

4.1. Cloud connection

The first fundamental step to apply FOTA technology is to connect to the internet, to do so we will need some prerequisites, we first need a server to upload the update to and handle the backend services, and a GUI to interface with this server and control it more easily and to make standard interface with cloud, GUI can be accessed from any remote PC and finally the embedded device must have a hardware that can connect to internet like a Wi-Fi module or GSM module the connection must be stable and secure.

So, the three main components to accomplish a connection to the cloud are:

1. Server for that we used firebase.
2. GUI for that we implemented simple GUI using python.
3. Wi-Fi hardware for that we used NodeMCU.

4.1.1. Firebase

Firebase is a Backend-as-a-Service (Baas). It provides developers with a variety of tools and services to help them develop quality apps, grow their user base, and earn profit. It is built on Google's infrastructure.

Firebase is google platform, free and easy to use, it have a real time database.

Firebase lets you automatically run backend code in response to events triggered by Firebase features and HTTPS requests. Your code is stored in Google's cloud and runs in a managed environment.

Firebase is categorized as a NoSQL database program, which stores data in JSON-like documents. Firebase is powerful and offers several services, including:

- Analytics – Google Analytics for Firebase offers free, unlimited reporting on as many as 500 separate events. Analytics presents data about user behavior in iOS and Android apps, enabling better decision-making about improving performance and app marketing.

- Authentication – Firebase Authentication makes it easy for developers to build secure authentication systems and enhances the sign-in and onboarding experience for users. This feature offers a complete identity solution, supporting email and password accounts, phone auth, as well as Google, Facebook, GitHub, Twitter login and more.

- Cloud messaging – Firebase Cloud Messaging (FCM) is a cross-platform messaging tool that lets companies reliably receive and deliver messages on iOS, Android, and the web at no cost.

- Realtime database – the Firebase Realtime Database is a cloud-hosted NoSQL database that enables data to be stored and synced between users in real time. The data is synced across all clients in real time and is still available when an app goes offline.

- Performance – Firebase Performance Monitoring service gives developers insight into the performance characteristics of their iOS and Android apps to help them determine where and when the performance of their apps can be improved.

- Storage – Firebase provide large storge where we can store data like the software update to access it later and download it.

4.1.1.1. Firebase real-time database

Figure 4.1

Used to store important data about the update like CRC which is used to check the validity of the code, the ECU ID in case the system has many ECUs, a flag to notify the device if there is a new update and the download link of the update, these variables will be changed through the GUI without the need to access the database directly through the firebase interface.

In figure (4.1) we see the database and variables we just mentioned NewUpdate variable is so important to track the update progress.

4.1.1.2. Firebase storage

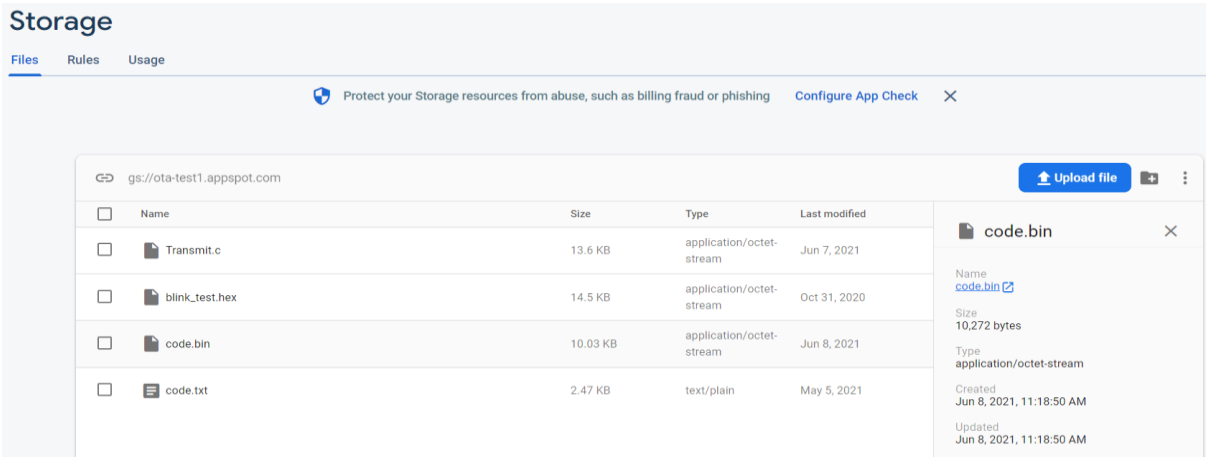

Figure 4.2

Used to store the update itself to download it later in the embedded device we can upload to it directly through the firebase interface or through the GUI provided to access the firebase to abstract the backend work.

4.1.2. GUI

Used to abstract the backend implementation with firebase and create standard interface to deal with the update, GUI is implemented using python language.

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built-in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance.

Python support firebase and with tkinter library we can implement the GUI in (figure 4.3).

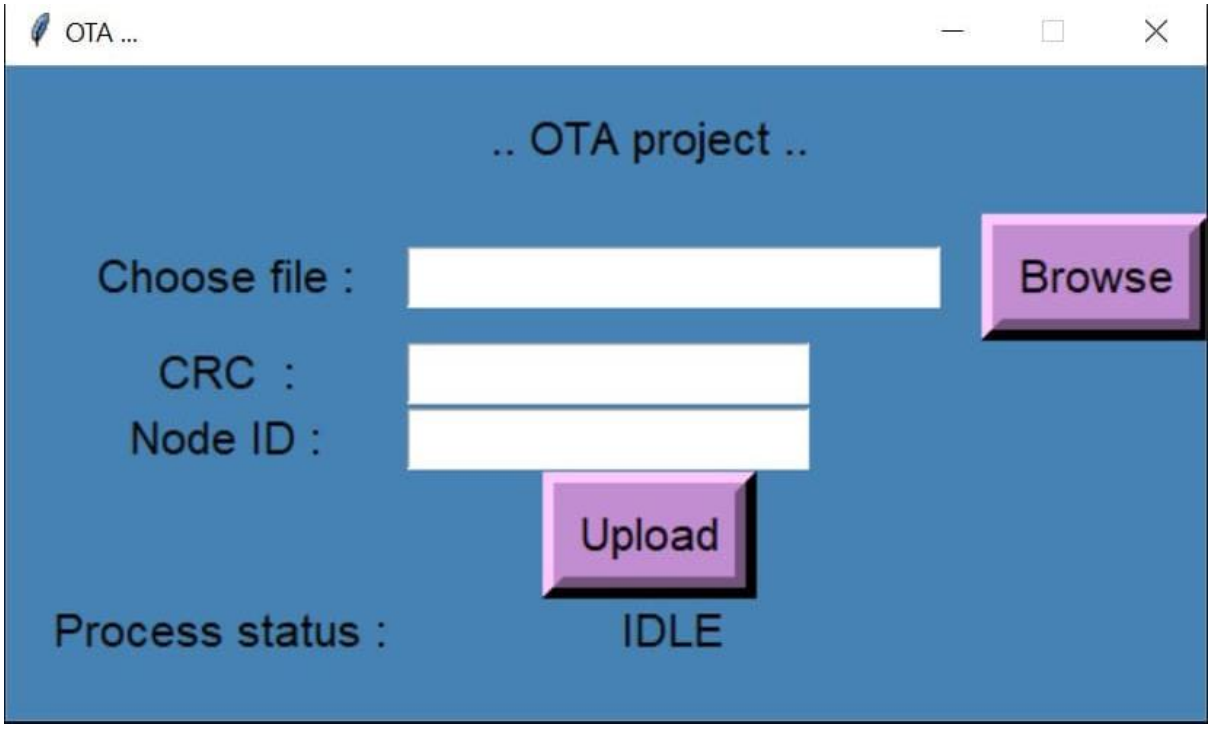

Figure 4.3

We enter three important fields first the code file and must be in binary form, the binary form is efficient and less in size than other forms, second field is CRC of the code and lastly the node ID.

After we enter all the fields, we press the upload file and status bar indicates the progress of the update or block the update and if the server is busy with another update.

4.1.3. NodeMCU

Like we stated before it is an open-source firmware and development kit that helps you to prototype or build IoT products. It includes firmware that runs on the ESP8266 Wi-Fi, and hardware which is based on the ESP-12 module. The firmware uses the Arduino IDE.

NodeMCU is an Arduino based microcontroller with additional feature of ESP8266 Wi-Fi chip with full TCP/IP stack, main advantage of Nodemcu is that it can directly connect to the internet without using any additional peripheral and can connect to cloud using HTTP or MQTT protocols to send and receive data to help IoT systems to be up to date and can monitor systems as well analyze data for the future developments.

We used NodeMCU to connect to the firebase cloud and download the update then will forward the update to the gateway through UART protocol.

To connect to the firebase NodeMCU must have a secret key to authenticate to the cloud to ensure security, and to prevent any other device to access the cloud without permeation.

In (Figure 4.4) we see the firebase hostname and authentication key is define in the code of the NodeMCU.

```
/* Server data */
#define FIREBASE_HOST "ota-test1.firebaseio.com"              // the project name address from firebase id
#define FIREBASE_AUTH "Uqw7UoTw9v1ZZUPr16vAdTKdy6MCwVehXKiHIsor"       // the secret key generated from firebase
```

Figure 4.4

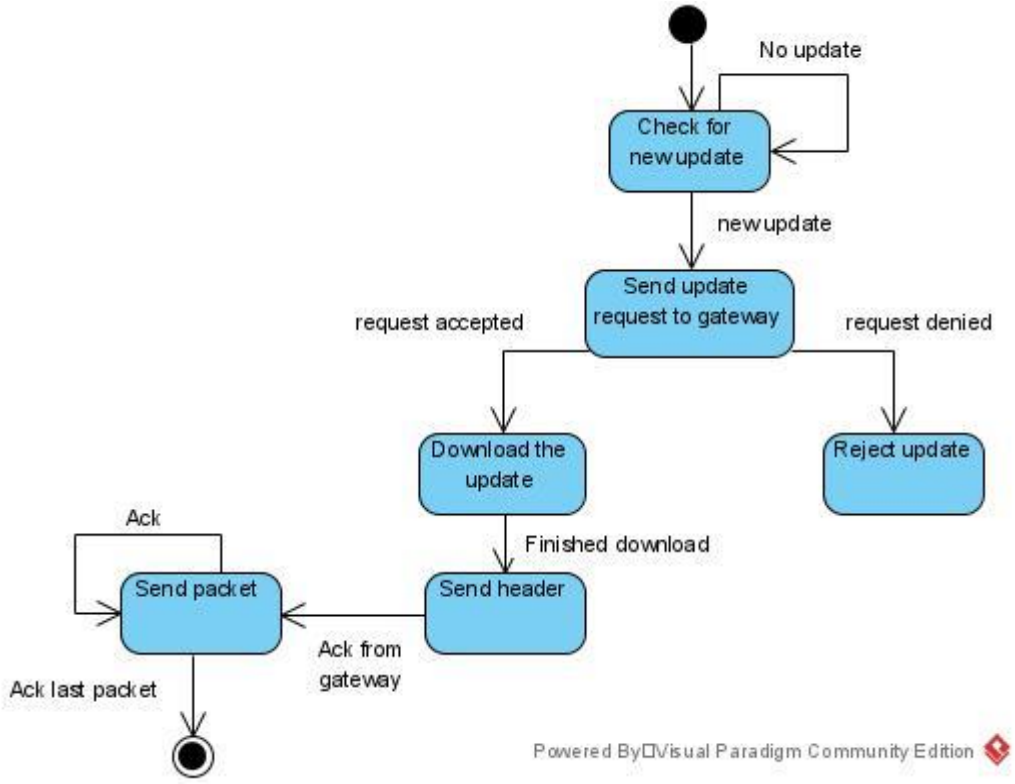

Figure 4.4

In (figure 4.4) we see the state machine of the sequence happening in the nodeMCU, as we see the code will not be downloaded unless the user accepts the update first, when the request is accepted the code will be downloaded and then will be sent to the gateway the header then the packet, the header includes data about the code like its size, CRC, and ECU ID.

The code will be divided into packets and each packet will be sent after the gateway acknowledge the node till all the code is sent.

After the code is sent to gateway and verified, NodeMCU will update the cloud that the update is successful, and the device now will be ready to receive other updates.

4.2. Security of software

4.2.1. Why we secure firmware?

We explained previously that we need to upload the Data to Firebase Cloud storage and send it over the air to Telematics unit so to prevent hackers attack and manipulation; we must apply the following countermeasures:

Authenticity: We must prevent reprogramming; the authenticity and integrity of the firmware needs to be protected. This ensures that the firmware originates from a trusted source and that it has not been modified.

Encryption of the firmware: Checking the authenticity of firmware guarantees that only trusted firmware is used in an update process, but a hacker may still be able to read the plaintext (original data) firmware to reverse engineer the source code, or steal data, which may lead to IP theft and/or privacy issues. Encryption of the firmware (for example, using AES, RSA, or DES) can prevent this.

4.2.2. Encryption Algorithms:

There are two main encryption methods. First, Asymmetric Encryption in which the encryption key is published for anyone to use and for encrypting messages. Only the receiving party has access to the decryption key that enables messages to be read. Public-key encryption was first described in a secret document in 1973. Before that, all encryption schemes were symmetric-key (also called private-key). Second, Symmetric Encryption in which the encryption and decryption keys are the same. Communicating parties must have the same key to achieve secure communication. We choose AES Symmetric method to encrypt the firmware.

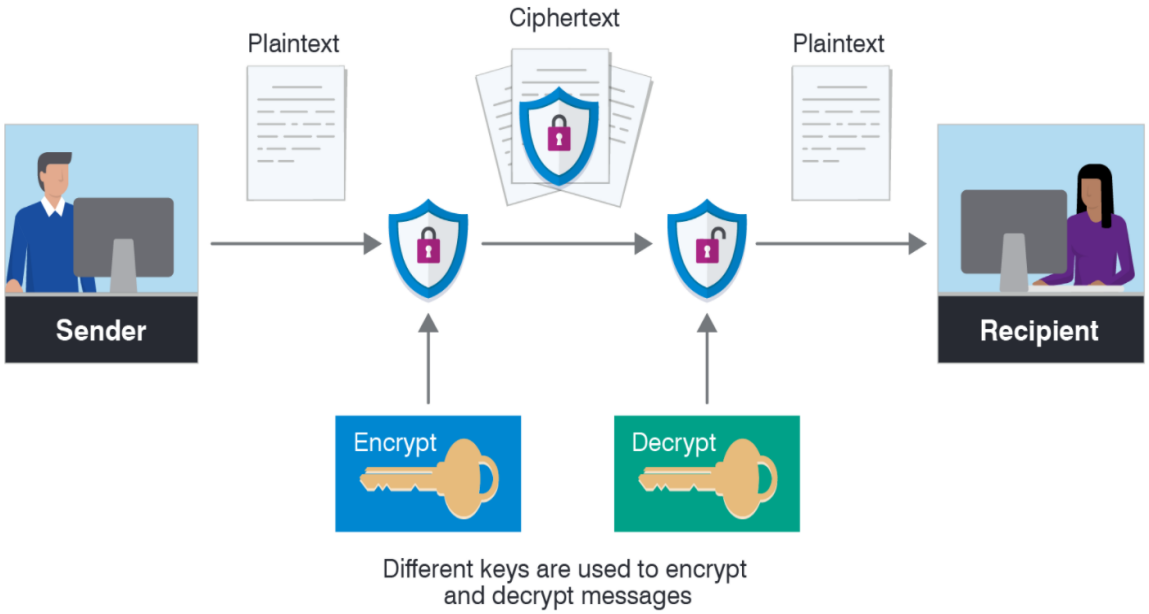

Figure 4.5

4.2.3. Advanced Encryption Standard (AES):

AES is a symmetric encryption cipher. This means that the same key used to encrypt the data is used to decrypt it. Over the years, AES has proven itself to be reliable and effective method of safeguarding sensitive information Some of the key benefits of using AES include the following:

- This robust security algorithm may be implemented in both hardware and software.

- It is resilient against hacking attempts, thanks to its higher-length key sizes (128, 192, and 256 bits).

- It is an open-source solution. Since AES is royalty-free, it remains highly accessible for both the private and public sectors

- AES is the most used security protocol today, used for everything from encrypted data storage to wireless communications.

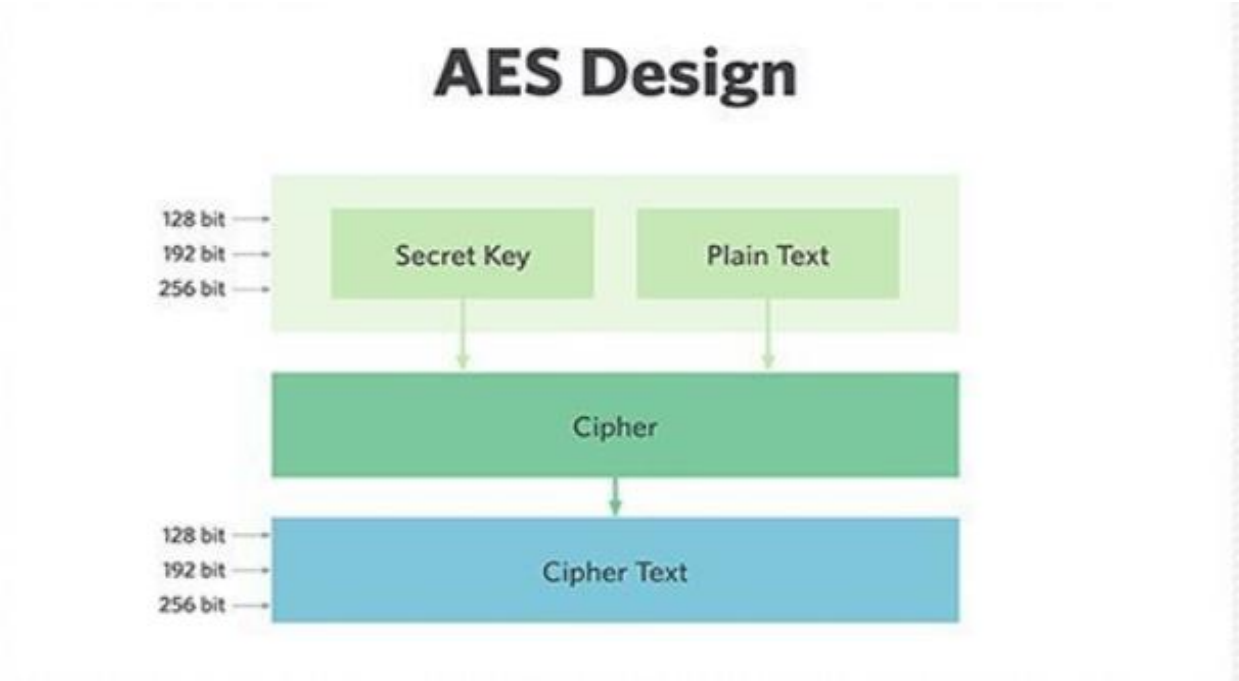

Figure 4.6

4.2.4. Encryption Process:

Encryption is a popular technique that plays a major role to protect data from intruders. AES algorithm uses a particular structure to encrypt data to provide the best security. To do that it relies on several rounds and inside each round comprise of four sub-processes. Each round consists of the following four steps to encrypt 128-bit block. Let us discuss these steps in details as shown in (figure 4.7)

a) Substitute Bytes Transformation:

The first stage starts with Sub Bytes transformation. This stage is depending on nonlinear S-box to substitute a byte in the state to another byte. According to diffusion and confusion Shannon's principles for cryptographic algorithm design it has important roles to obtain much more security.

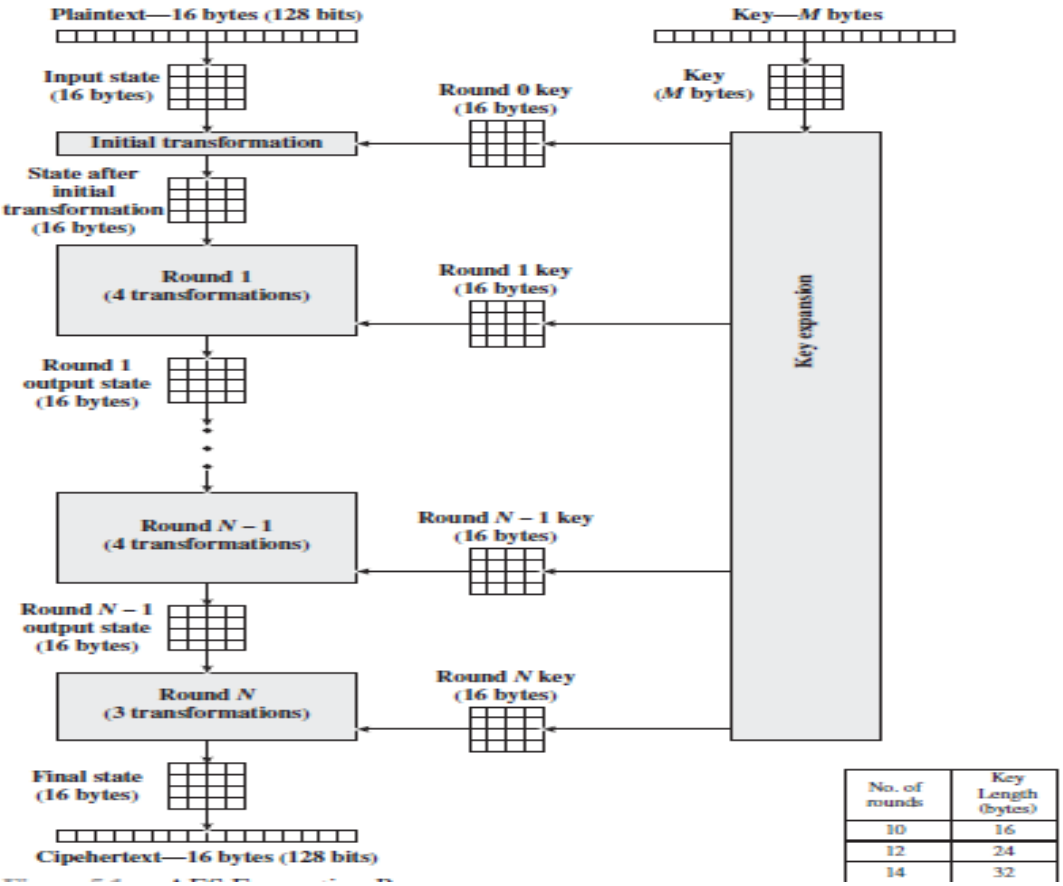

Figure 4.7 (Encryption process)

b) Shift Rows Transformation:

The next step after Sub Byte that perform on the state is Shift Row. The main idea behind this step is to shift bytes of the state cyclically to the left in each row rather than row number zero. In this process the bytes of row number zero remain and does not carry out any permutation. In the first row only one byte is shifted circular to left. The second row is shifted two bytes to the left. The last row is shifted three bytes to the left the size of new state is not changed that remains as the same original size 16 bytes but shifted the position of the bytes in state.

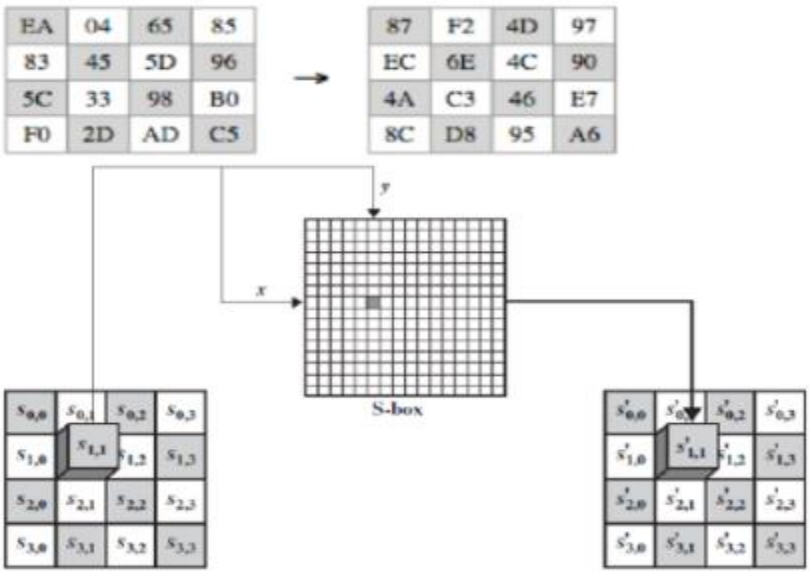

*Figure 4.8 (*Substitute Bytes*)*

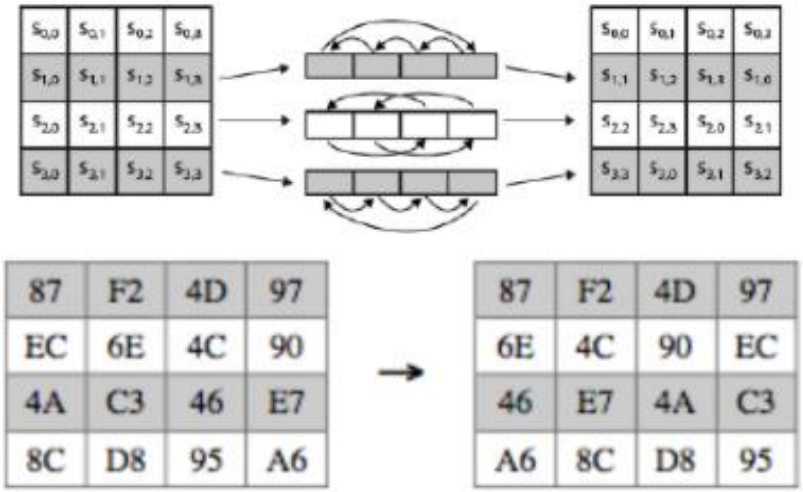

*Figure 4.9 (*Substitute Bytes*)*

c) Mix Columns Transformation:

Another crucial step occurs of the state is Mix Columns. The multiplication is carried out of the state each byte of one row in matrix transformation multiply by each value (byte) of the state column. In another word, each row of matrix transformation must multiply by each column of the state. The results of these multiplications are used with XOR to produce a new four bytes for the next state. In this step, the size of state is not changed that remained as the original size 4x4.

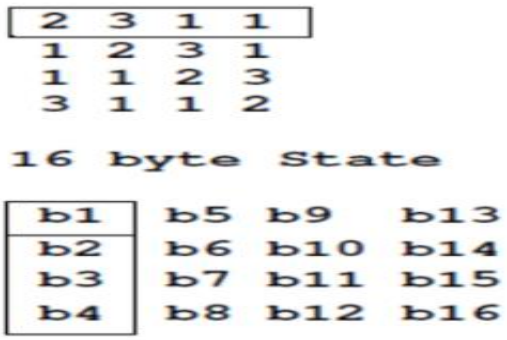

*Figure 4.10 (*Multiplication Matrix*)*

b1 = (b1 * 2) XOR (b2 *3) XOR (b3 * 1) XOR (b4 * 1), and so on until all columns of the state are exhausted.

d) Add Round Key Transformation:

Add Round Key is the most vital stage in AES algorithm. Both the key and the input data (also referred to as the state) are structured in a 4x4 matrix of bytes. (Figure 4.11) shows how the 128-bit key and input data are distributed into the byte matrices. Add Round Key can provide much more security during encrypting data. This operation is based on creating the relationship between the key and the cipher text. The cipher text is coming from the previous stage. The Add Round Key output exactly relies on the key that is indicated by users.

*Figure 4.11 (*Add Round Key*)*

e) AES Key Expansion:

AES algorithm is based on AES key expansion to encrypt and decrypt data. It is another most important step in AES structure. Each round has a new key. In this section concentrates on AES Key Expansion technique. The key expansion routine creates round keys word by word, where a word is an array of four bytes. The routine creates 4X (Nr+1) words. Where Nr is the number of rounds.

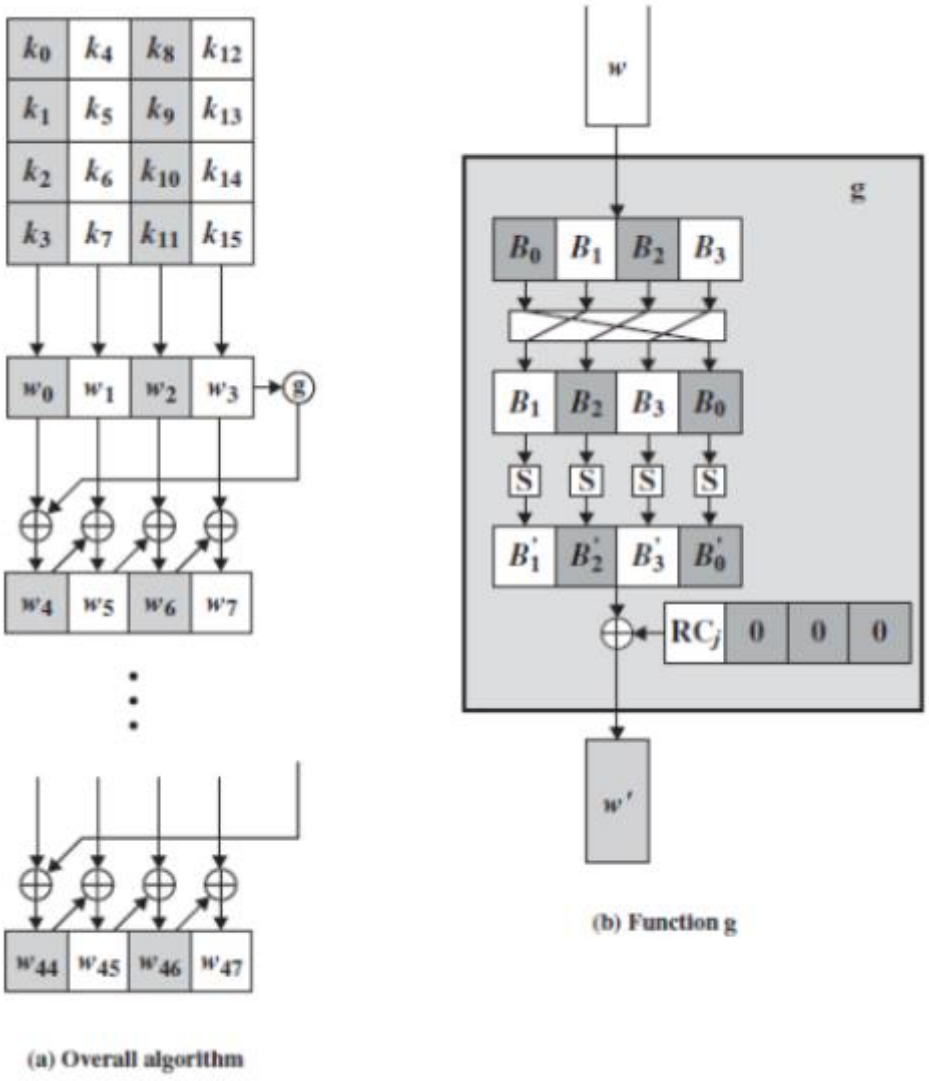

*Figure 4.12 (*Add Key Expansion*)*

4.2.5 Decryption Process:

The decryption is the process to obtain the original data that was encrypted. This process is based on the key that was received from the sender of the data. The decryption processes of an AES are like the encryption process in the reverse order and both sender and receiver have the same key to encrypt and decrypt data. The last round of a decryption stage consists of three stages such as Inverse Shift Rows, Inverse Sub Bytes, and Add Round Key as illustrated in (Figure 4.13).

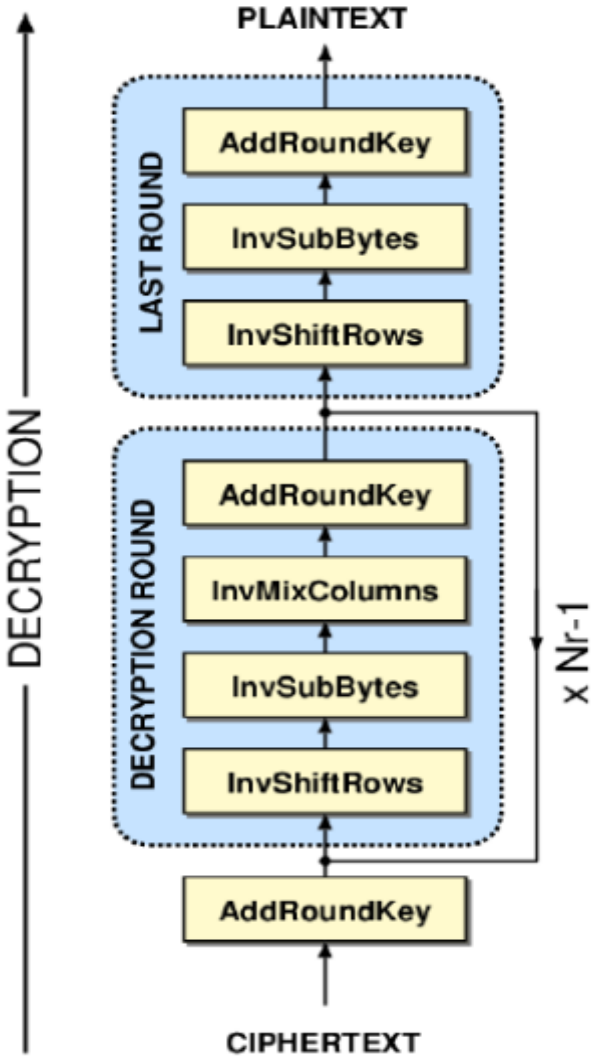

*Figure 4.13 (*Decryption Process*)*

4.2.6 Encryption and Decryption Stages:

Now we learned how to perform AES Encryption and Decryption, and to apply the second countermeasure. We will encrypt the firmware before uploading it to the storage and convert it to cipher text. After receiving the cipher text successfully by Telematics unit, we will verify and decrypt it to its plain text then forwards it to the communication network to its final target destination.

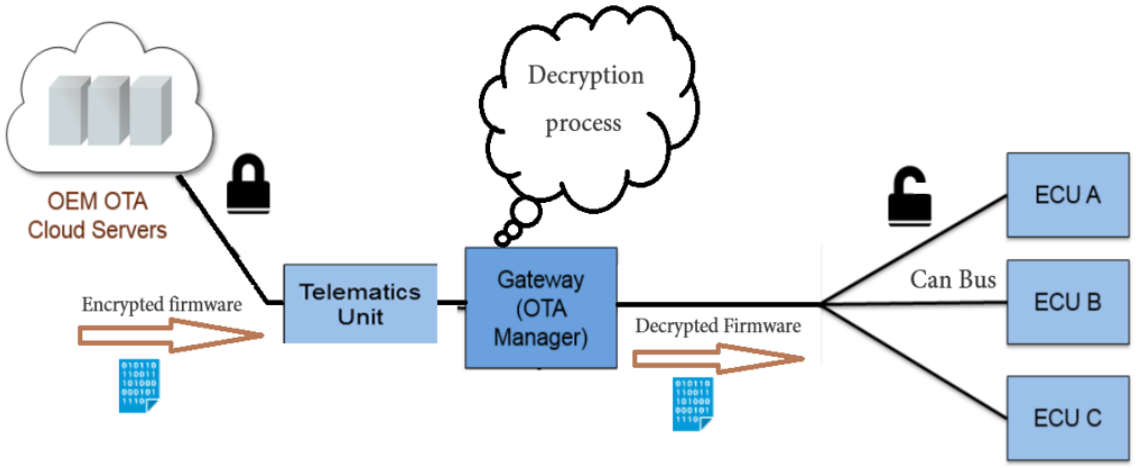

*Figure 4.14 (*Decryption & Encryption Stages*)*

4.3. Communication Protocol CAN

4.3.1. Introduction:

To make the car smart and more safety new ECUs were added to the cars, it is not a problem. The problem was how to connect between these different ECUs to deliver the data from ECU to another. To solve the problem of wiring in the cars, Bosch started to develop a new serial communication protocol in the early1980s. In 1986, Bosch introduced The Controller Area Network (CAN) bus to the world, the first use of CAN bus was in 1988 BMW 8 Series cars. The last version of CAN bus is CAN with Flexible Data-Rate (CAN FD) and it released from Bosch in 2012, this version speed reach to 8 Megabit/sec and allows to sending 64 bytes of data in every message. The CAN

bus is also using to run diagnostics on the cars to report the status of each ECU in the car this feature makes the fix of car's problem easier. Today all cars in the world use CAN bus as a main serial communication protocols between most of ECUs for several reasons:

- Message priority assignment and guaranteed maximum latencies.
- Multicast communication with bit-oriented synchronization.
- System-wide data consistency.
- Bus multi-master access.
- Error detection and signaling with automatic retransmission of corrupted messages.
- Detection of permanent failures in nodes, and automatic switch-off to isolate faulty node.

4.3.2. Can Network physical connectors:

ISO-11898-2 does not specify the mechanical wires and connectors. However, the specification does require that the wires and connectors meet the electrical specification. The specification also requires 120Ω (nominal) terminating resistors at each end of the bus as shown in (Figure 4.15). MCP2551 Transceiver is used to adjust the voltages across the bus network.

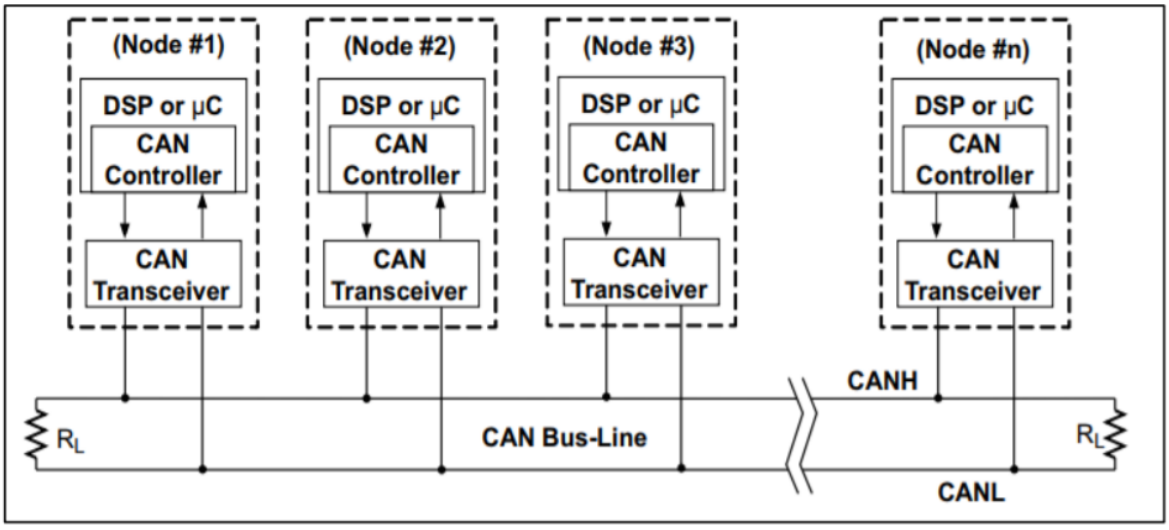

Figure4.15 (Can nodes connection)

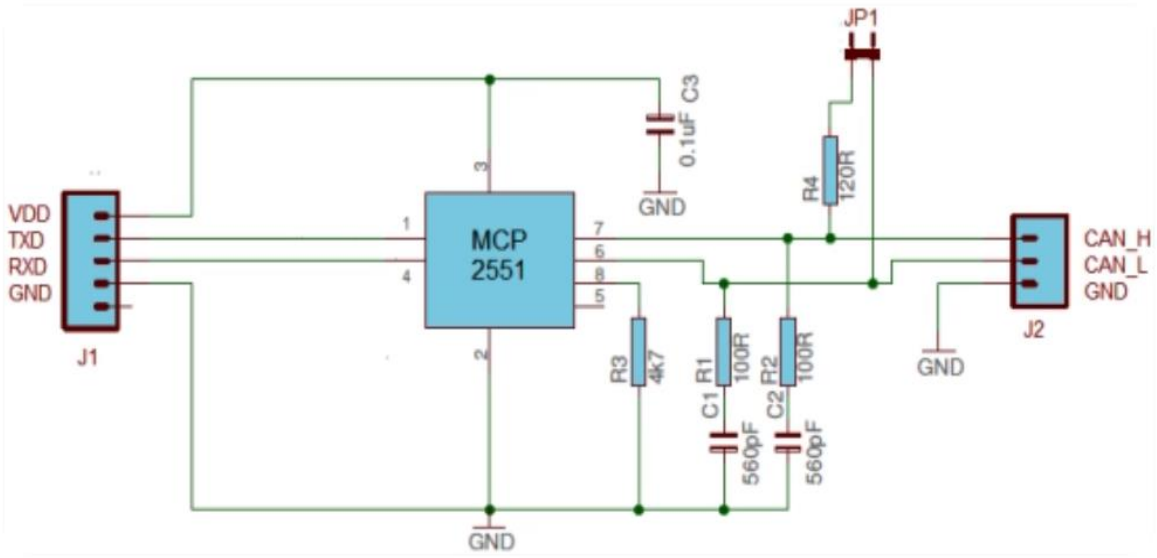

Figure4.16 (MCCP connection)

4.3.3. Can Network Message Format:

a) Data Frame Format:

Data Frame are used to transmit information between a source node and one or more receivers.

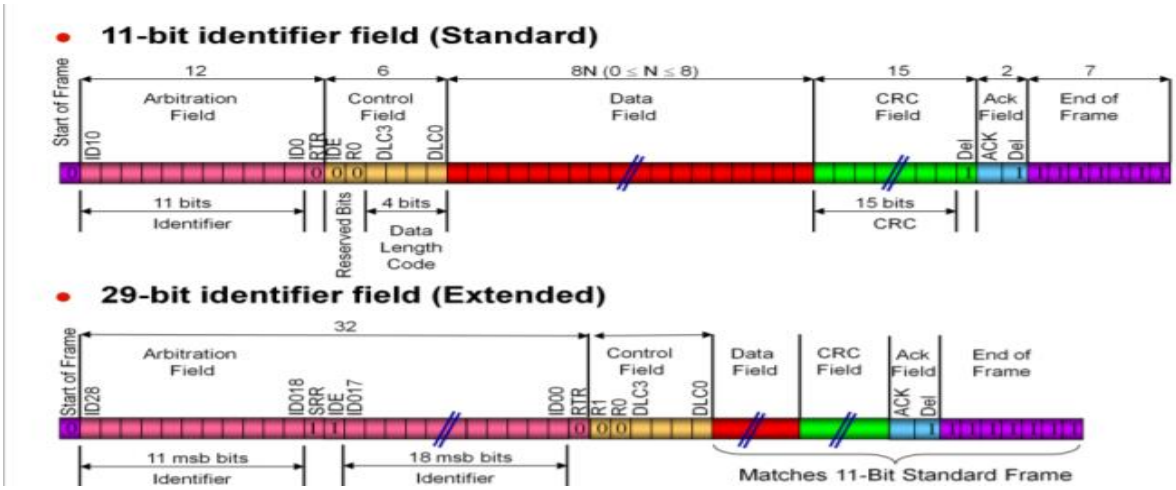

Figure4.17 (Data Frame Format)

b) Remote Frame Format:

Remote Frame requests the transmission of a message by another node.it also detected by a high RTR bit, each receiving node in a can network when detecting a high RTR bit will know that the receiving message is a remote frame.

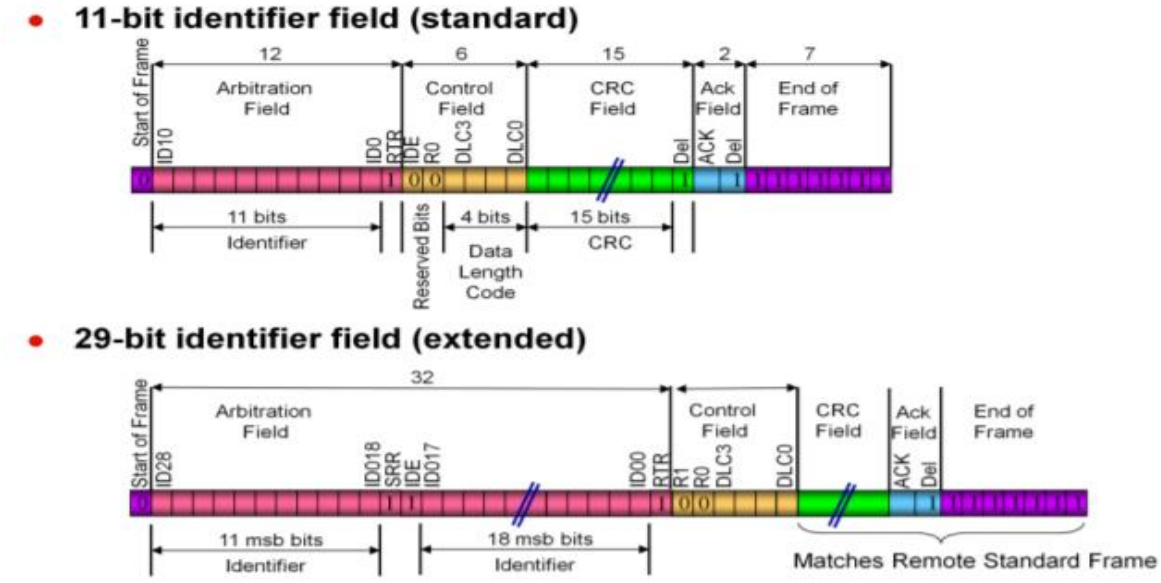

Figure4.18 (Remote Frame Format)

c) Error frame:

Any Node can transmit error frame when detect an error with data or remote frames.

4.3.4. Interface between Nodes and Communication Network:

System Nodes are stm32f103 boards, which support can units. The system contains three nodes, every one of them must have its unique message id so that all other nodes in the system can communicate with it. The messages must contain this id so the corresponding node can ack. We should know that all messages in the systems will be sent through the same network bus, but the node will response and save only the messages that hold its id. There are techniques in Can unit itself that can handle transmitting and receiving messages.

a) Can Filtering:

The filtering is done by arbitration identifier of the CAN frame. This technique is also used when monitoring a CAN bus, to focus in on messages of importance using an identifier and mask. These allow a range of IDs to be accepted with a single filter rule. When a CAN frame is received, the mask is applied. This determines which bits of the identifier will be used to determine if the frame matches the filter.

b) Can FIFO Buffer:

First in First Out concept using in CAN to Store multiple Messages Received in RAM as Receiving each message Individually keeps interrupting the CPU for every single Message receives. It will make system overhead especially in multitasking systems.

4.3.5. Can In Action:

For the next couple of flow chart, we will illustrate how to transmit and receive a message based on our blue pill board and our Can unit Driver.

a) Transmitting a Can message:

Transmitting a message typically requires loading the identifier, RTR, ID, Data length and the Data into Transmit structure.

b) Receiving a can message:

Receiving a message typically requires loading filtering id to set filter structure and monitoring the FMP bit with interrupt or polling to notify user application when a can message has been received.

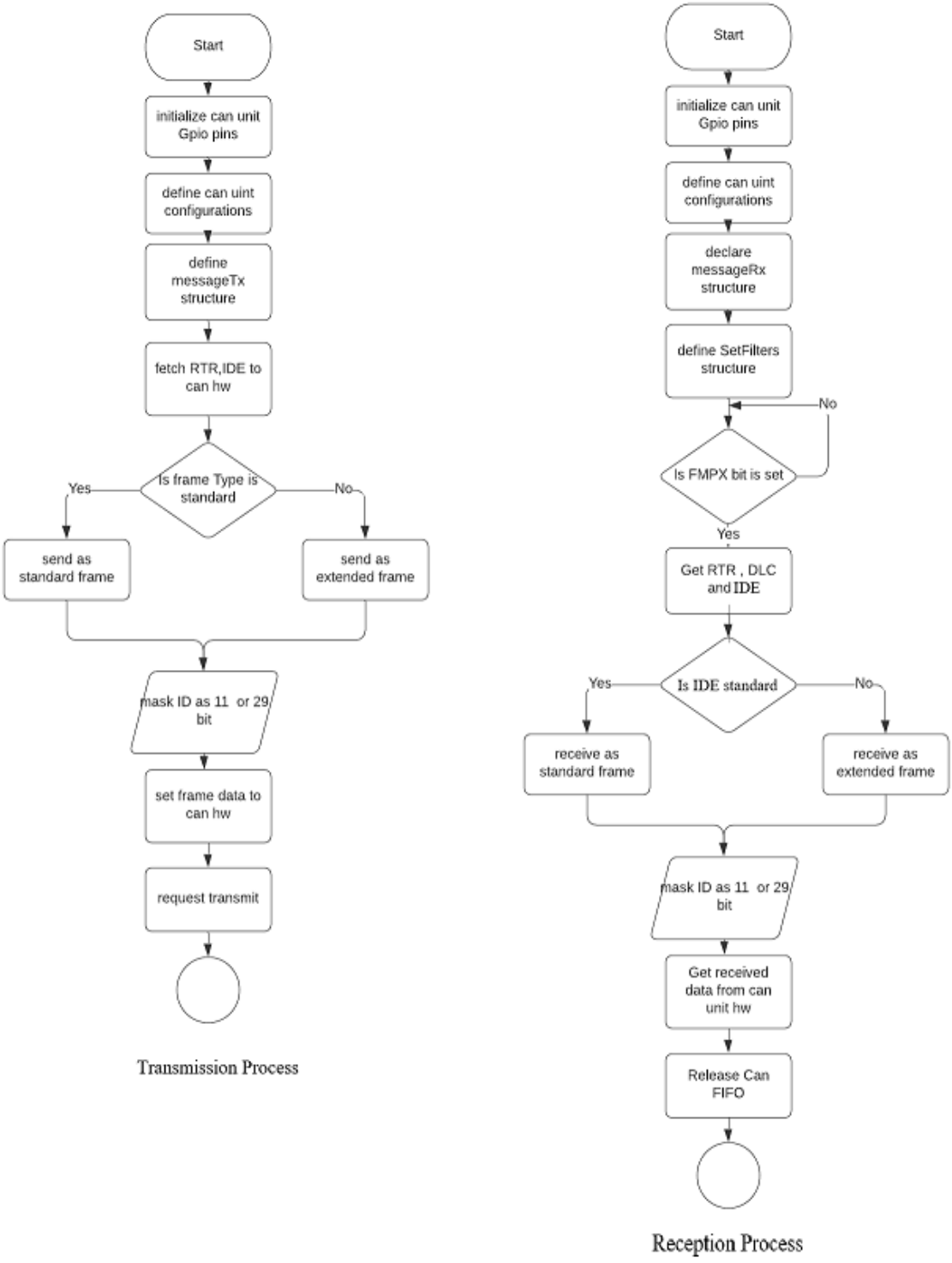

Figure 4.19 (Can Transmission &Reception)

4.4. Gateway

Gateway is the bridge between the telematics unit and the target ECUs, it is very important especially with application having many ECUs like cars, gateway receive the code from NodeMCU through UART protocol and forward it to the target ECU through a CAN bus.

A gateway is a central hub that securely and reliably interconnects and processes data across these heterogeneous networks. It provides physical isolation and protocol translation to route data between functional domains (powertrain, chassis and safety, body control, infotainment, telematics, ADAS) that share data to enable new features. Gateways allow engineers to design more robust and functional vehicle networks.

4.4.1. Gateway functions

- Save information about connected ECUs, this information includes ECUs IDs, software version of every ECU and any important data related to the connected ECUs.

- Decrypt the code before sending it to the target ECU, gateway in most cases will be more powerful microcontroller than the other ECUs so it will hold the decryption algorithm and make sure the code had been decrypted correctly before forwarding it, and like stated before decryption is very necessary step to as the code will be encrypted in the server to security reasons.

- Check if code is valid, it is important to check before sending to ECU to avoid sending buggy code or incorrect data.

- Determine which ECU this code is for, and that is one of the main tasks to make sure the code is delivered to the correct ECU, as stated before the gateway has information about all the ECUs including ECUs IDs so it can determine which ECU the update is targeted to.

- Resolve dependencies in case of rollback, and that is why the gateway save information about every ECU's software version, sometime a rollback will be needed due to the failure of the current software, and that software will be relying on other ECUs, so if this ECU will execute a rollback should inform first the gateway to send a command to rollback all the ECUs depending on that ECU.

- Notify the node when the update is complete so the node can notify the server too that the code update is done.

4.4.2. Gateway software

As gateway has many important functions and deals with both the NodeMCU and target ECUs the software will be complexed and must be designed and implemented carefully, so we followed some of AUTOSAR concepts mentioned in last chapter and implemented it with a modular way to make easy to edit and add features in the future.

4.4.2.1. Gateway layered architecture

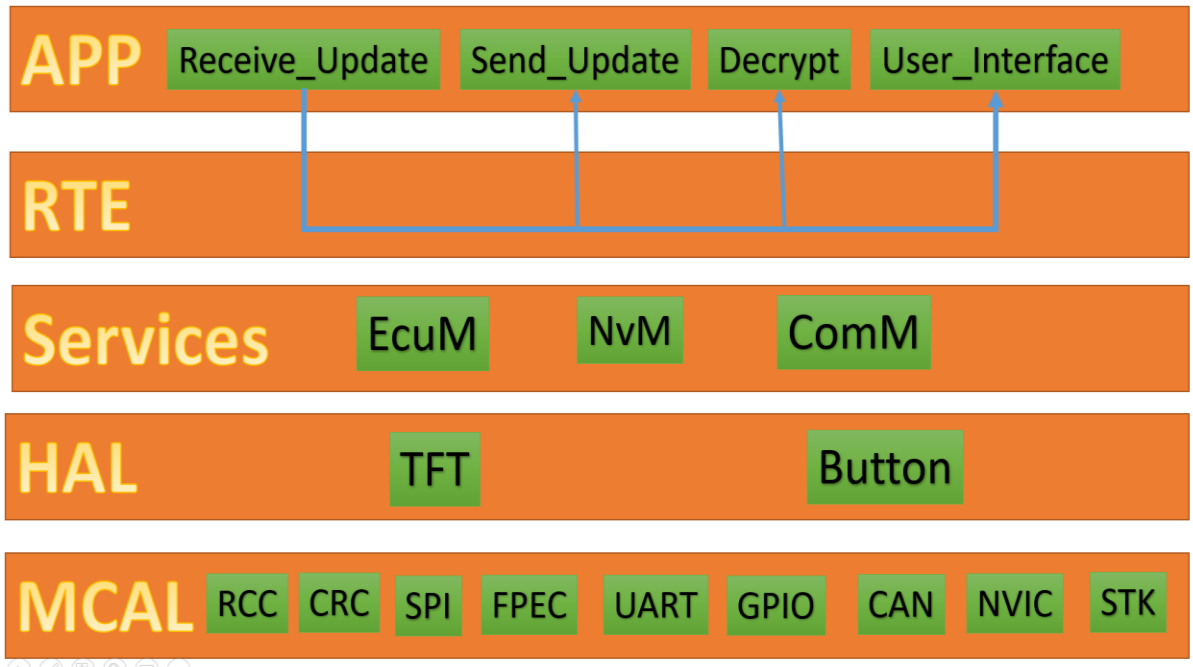

Figure 4.20

As we can see in (Figure 4.20) the architecture of the code consists of four layers every layer has some important modules needed to implement the application:

1- Lower layer is MCAL layer it the layer dealing with software that mange the hardware of the microcontroller to abstract the hardware from the upper layers we implemented several modules:

1. RCC: for the management of the clock of microcontroller.
2. CRC: for calculations of CRC.
3. SPI: a protocol to communicate with TFT.
4. FPEC: to write and erase flash.
5. UART: protocol to communicate with NodeMCU.
6. GPIO: to interface with microcontroller pins.
7. CAN: communication protocol to connect gateway with many ECUs.
8. NVIC: mange the interrupts.
9. STK: a timer.

2- Second layer HAL deals with any hardware outside the microcontroller and used also to abstract any details related to the hardware has only two modules:

1. TFT: to manage TFT display.
2. Button: to manage the buttons interface.

3- Third layer is the services layer and is used also to abstract all the detail related to the lower layers and provides standard interfaces for the application to deal with hardware without worrying about any dependency, it has three modules:

1. EcuM: mange everything related to clock and power and interrupts.
2. NvM: mange everything related to memory.
3. ComM: mange everything related to communication.

4- Fourth layer is RTE we mentioned it in the last chapter and its important to connect modules in the application layer.

5- The most upper layer is the application layer it has the logic of all the functions we stated before and we tried to implement features in separate modules, it had four important modules:

1. Receive_Update: handle the connection with NodeMCU.
2. Decrypt: handle the decryption.
3. Send_Update: mange forwarding the code to target ECU.
4. User_Interface: mange the interface with user.

4.4.2.2. Gateway state machine

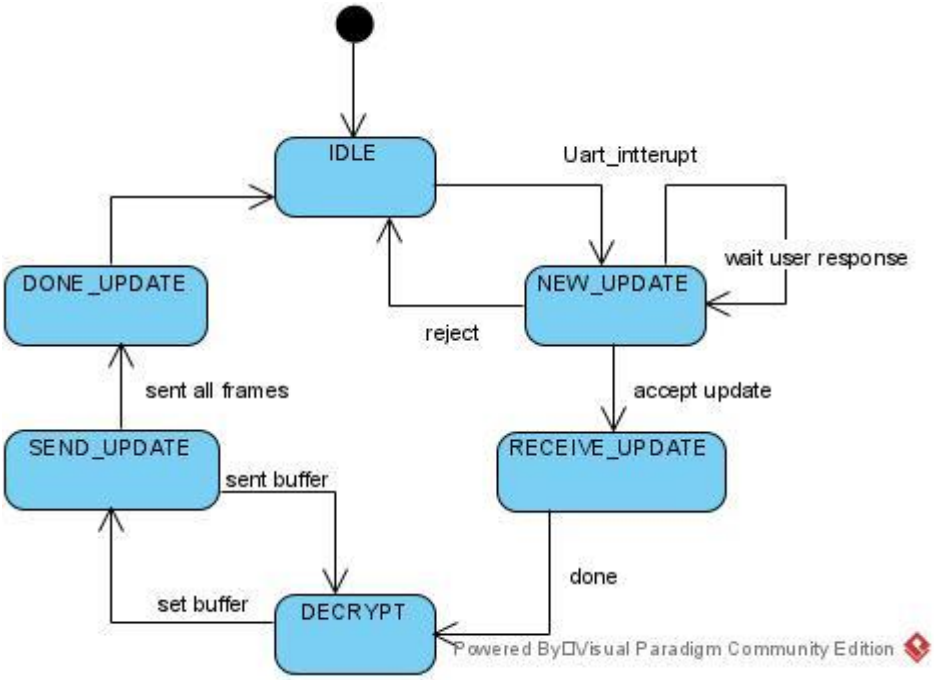

Figure 4.21

In (Figure 4.21) we see the state machine of the whole gateway system.

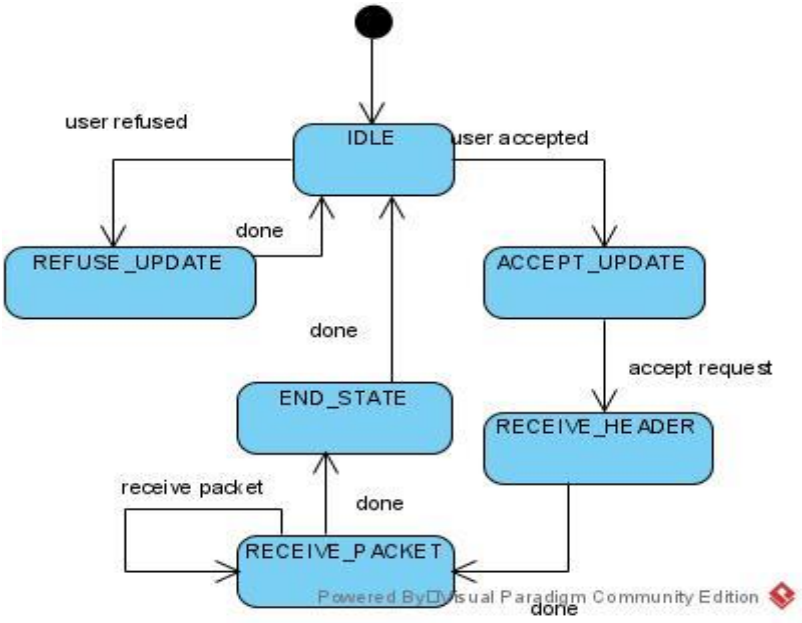

Figure 4.22

In (Figure 4.22) the state machine of the Receive_Update module that handles connection with NodeMCU.

4.4.3. Gateway sequence

1. Gateway remains in idle state waiting a notification from NodeMCU in (Figure 4.23) we can see the TFT screen in idle state

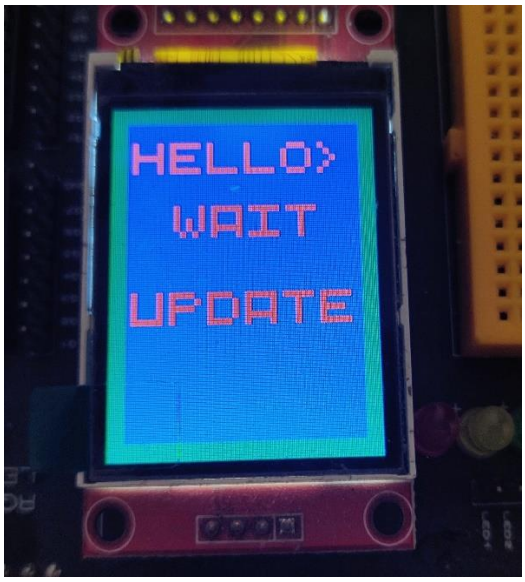

Figure 4.23

2. When gateway receive UART interrupt from NodeMCU indicating that a new update available the gateway will notify the user as we see in (Figure 4.24) the user can accept or reject the update

Figure 4.24

3. If the user rejected the update the gateway will return to the idle state but if the user accepted the update the code will be downloaded and sent to gateway and the progress will be shown like in (Figure 4.25).

Figure 4.25

4. When the download finishes and the update will be installed by forwarding the update after decryption to the target ECU.

Figure 4.26

5. When the installation is done the gateway will notify the user that update is successful and will return to idle state

Figure 4.27

4.5. Bootloader

4.5.1. Why updating a firmware?

Even though firmware is not designed to be changed, updates in a bootloader or firmware are usually needed to correct bugs or to add new functionalities. Indeed, there are firmware that are not correctly designed and so contain bugs which can sometimes be critical. Sometimes, the firmware is just too old and does not comply with the client desires any more. In all these situations, a new firmware is needed.

4.5.2. Why In application programming?

Using Over-The-Air update: the new firmware is download thanks to wireless communication using a very specific protocol just made for the update operation. The user of embedded device does not have to go to the maintenance center to update his device, The new update will come to the device remotely over the internet.

4.5.3. What is bootloader?

A bootloader is an application whose primary purpose is to allow a systems software to be updated without the use of specialized hardware such as a JTAG programmer.

The type of bootloader related to the communication protocol that receive the update from it.

4.5.4. Bootloader requirements

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

Requirements:

1- Ability to switch or select the operating mode (Application or bootloader).
2- Communication protocol requirements (USB, CAN, I2C, UART).
3- Record parsing requirement (S-Record, hex).
4- Flash system requirements (erase, write, read).
5- EEPROM requirements (erase, read, write).
6- Error detection mechanism (checksum, CRC, …).
7- Security and encryption.

4.5.5. Bootloader system

Bootloaders can come in many different sizes and in many different flavors but in general the operation of a system with a bootloader is relatively standard. There are three major components to these systems that can be seen in the Figure. They are the branching code (green), the application code (blue) and the boot-loader code (red). For most systems we prefer them to be executing the application path most of their operational life. (Figure 4.28) highlights the execution path to get to the application by the dashed red line. The orange block is a common block used by both the bootloader and the application to reset the system.

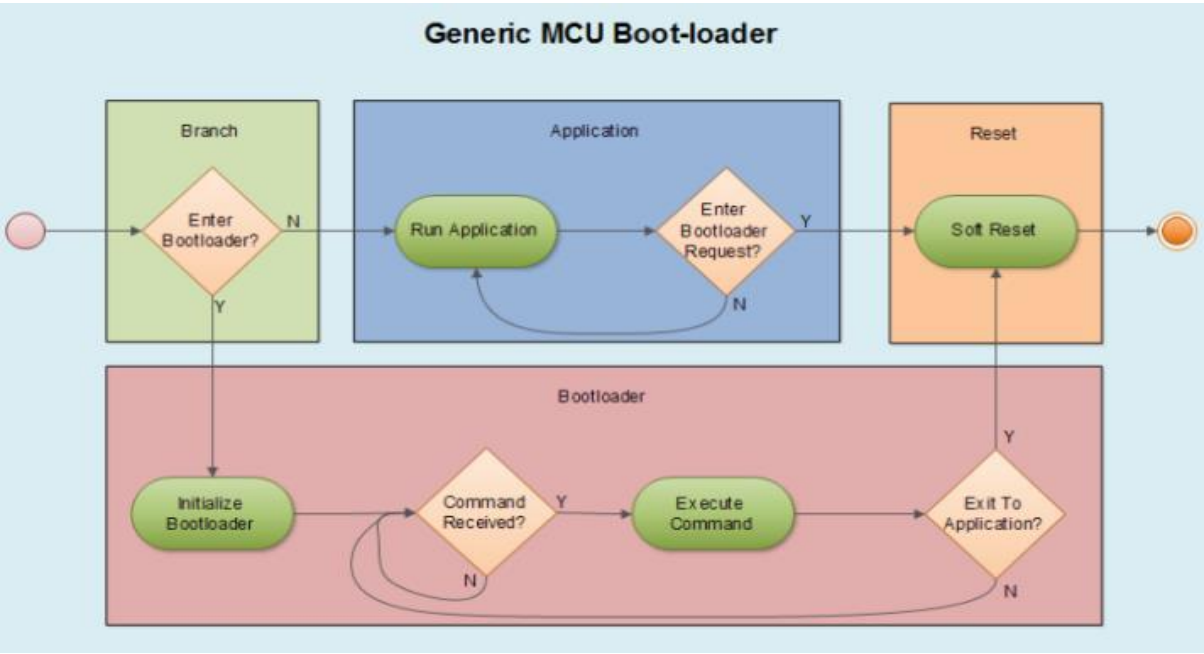

Figure 4.28

4.5.6. Bootloader behavior

A bootloader itself is not that different from a standard application; in fact, it is a standard application. What makes a boot-loader special is that it is sharing flash space with another application and has the capability to erase and program a new application in its place. Like every other application, one of the first priorities of the bootloader is to initialize the processor and the minimum number of peripherals required to carry out boot-loading functions. It's best to keep the use of peripherals to as few as possible in the bootloader in order to attempt to maximize the flash space that will be available for the application code.

The typical sequence for programming a device can be:

- Start the branch code to determine which code 'll be executed based on a flag in EEPROM memory or flash memory.
- If bootloader will be executed
    1- Move the active image to backup image if exist.
    2- Receive code packet by packet from can protocol and store it in the active region in the flash memory.
    3- Reset branching flag and make software reset.
- If there is an application.
    1- Generate CRC and check it.
    2- If CRC is ok, it jumps to the application to be executed.
    3- If CRC is not ok, it moves the backup image to the active region to be executed.

4.5.7. Application behavior

The behavior of the application image is of for the most part not of any interest to the bootloader designer except in one aspect; the application needs to be capable of receiving a command to enter the bootloader.

This means that the application needs to have two bootloader like capabilities:

1- set a piece of information that the bootloader can detect to enter bootloader mode.
2- Reset the system to initiate a branching decision.

4.5.8. Start-up branching

When the system starts up, there are at least two different software images that can be loaded and executed by the micro-controller, the bootloader, the application and possibly a backup application image. It is therefore necessary that as part of the boot-loader image code a branching algorithm be included that can handle the decision-making process of which image to load.
There are many different methods that can be used to decide which image to load. The simplest method that can be used and is most often used in example code from chip manufacturers is the use of a GPIO line to make the decision. For example, if the GPIO signal is high, load the application; if it is low then load the bootloader.
A single I/O line being used to branch to the bootloader is not a very robust solution. A user could accidentally push the bootloader button, or a noise event could cause the system to enter the bootloader. Instead, a common method used to detect a request to enter the bootloader is to change an EEPROM value.

The robust solution to the branching code is that the code will calculate the application image checksum. When it is completed, the branching code will check "Does the application reset vector exist?"

4.5.9. Memory partitioning

Every microcontroller has some form of non-volatile memory that is used to store the program. The most used type of memory is flash. Flash is broken up into divisible sections. The smallest section of flash is often referred to as a page. Pages are organized into larger structures known as sectors. Sectors are in turn organized into larger structures known as blocks. Each microprocessor is different as to how these sections of flash can be manipulated. Most will allow you to write a single byte to flash at a time. Others may require that 8 bytes or 256 bytes be written at one time. In most cases, the smallest section of flash that can be erased at a time is a single sector that often consists of 4kB. It is important that before a designer gets too far in their boot-loader design that they pull out the microcontroller datasheet and read through the flash and memory organization chapters.

The primary purpose for detailed examination of the memory map is to determine what sections of flash are available and best used for the bootloader.

There are several factors that should be considered when selecting where to locate the bootloader.

1- Bootloader size.
2- Vector table location.
3- Write protection and code security flash section.

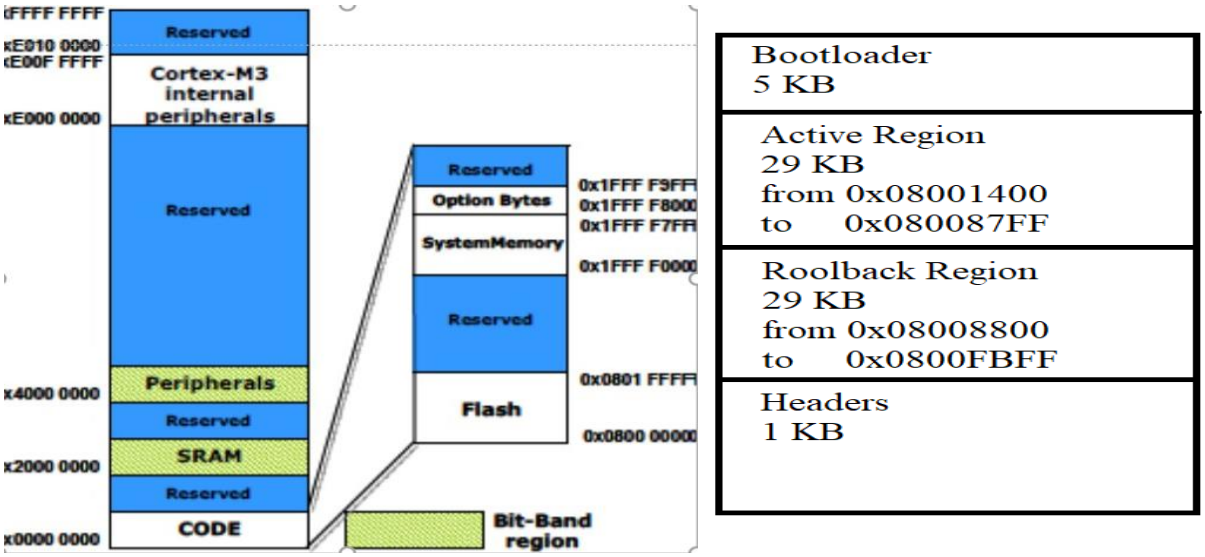

Figure 4.29

4.5.10. Embedded application setup

For the embedded application to function properly there is an important change that needs to be made so that it plays nice with the bootloader. The linker file needs to be updated so that it does not try to locate code or data within the boot-loader memory space.

The linker file is used by the compiler to decide where to place functions in RAM and flash. While the boot-loader memory space would be protected within flash, if the application tries to locate itself in the flash space the bootloader would ignore those memory locations. The application would never get written correctly to flash. This could lead to a unit that is completely non-functional. It is therefore imperative that the linker be modified so that it does not occupy the same space.

4.5.11. Layered architecture

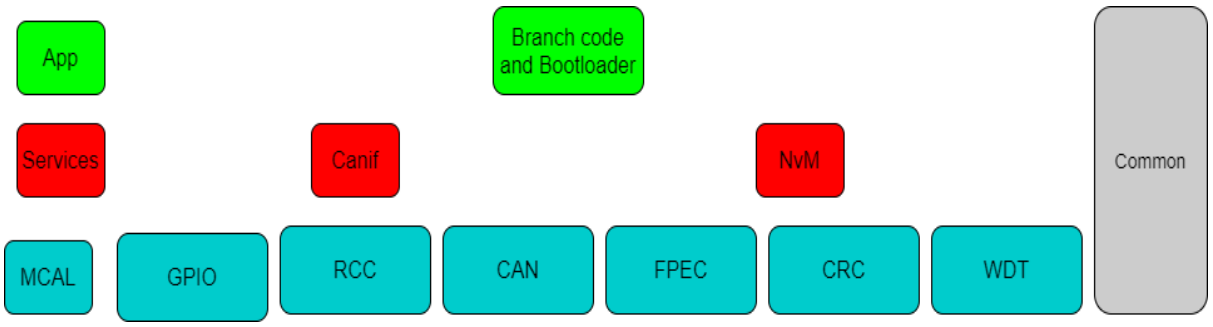

Figure 4.30

4.5.12. State machine diagram

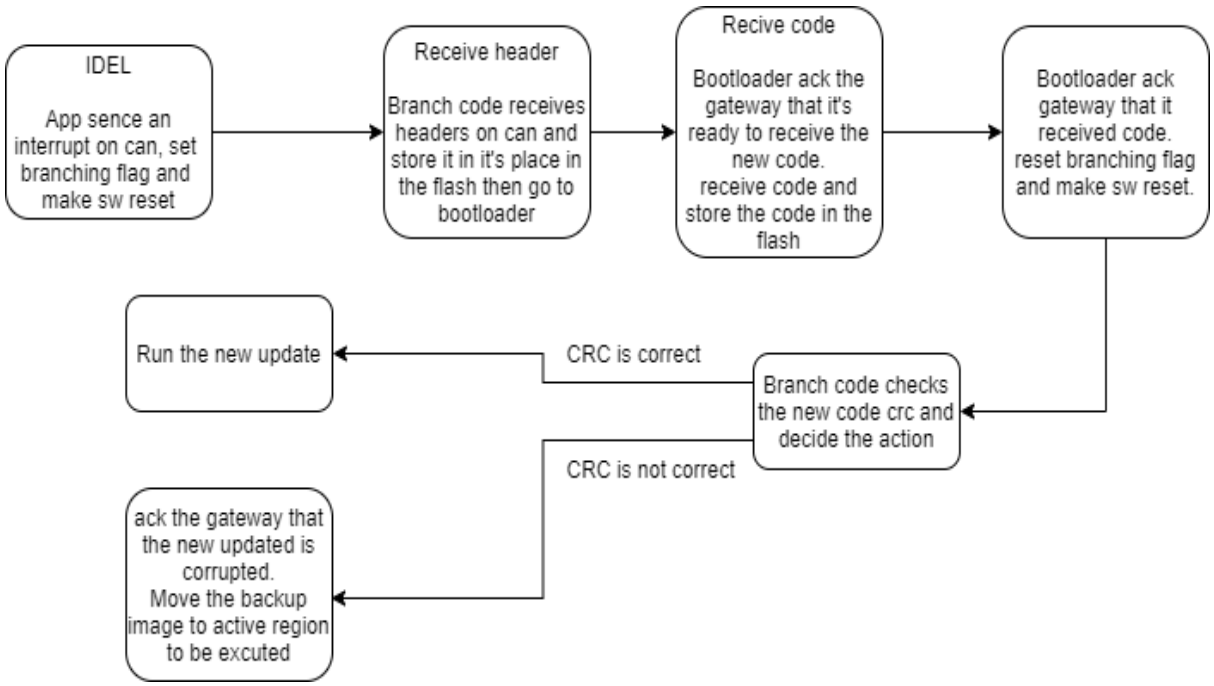

Figure 4.31

4.5.13. Sequence diagram

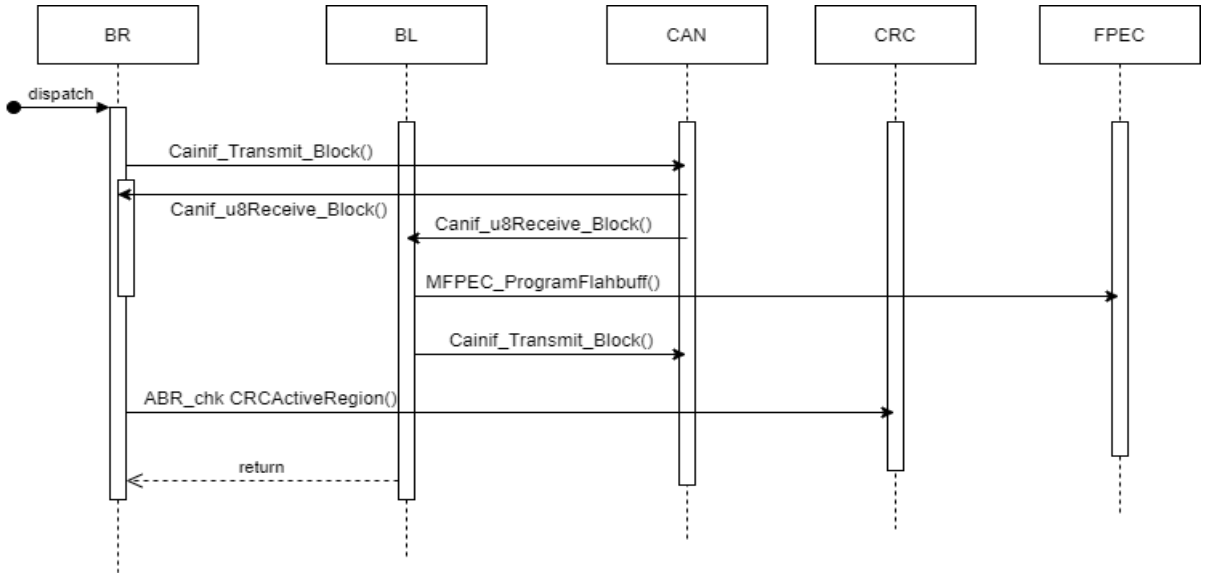

Figure 4.32

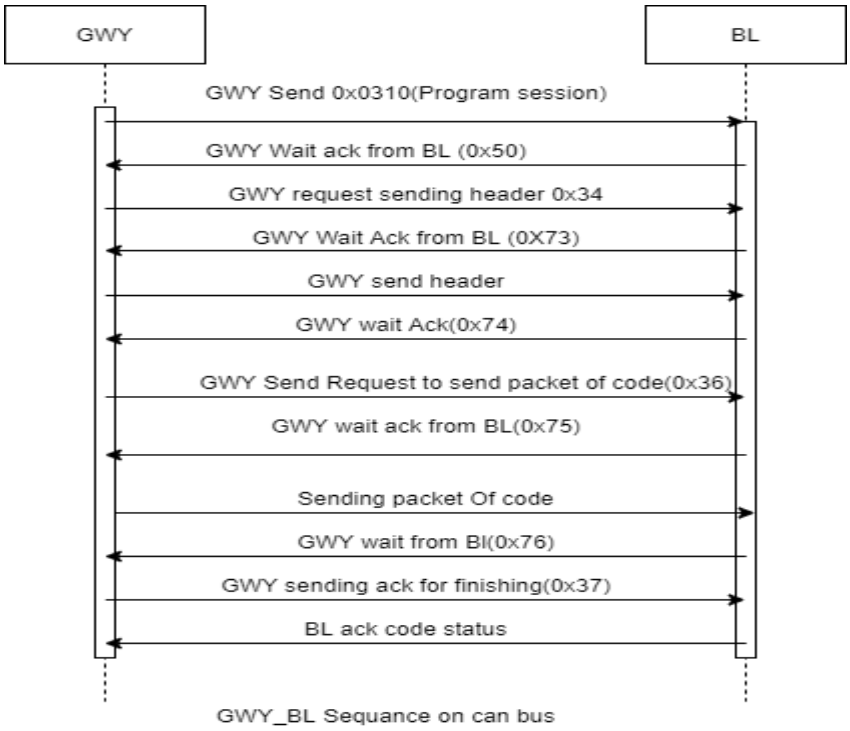

Figure 4.33

Chapter (5): Conclusion

5.1. Test cases results

The project had two phases, at the first phase we tested the concept of bootloader with establishing a connection with the server using only the NodeMCU and one STM32f103 and a simple test case of flashing a led the update code was 2,528 byte and successfully flashed the link blow has a video to the first phase.

https://github.com/mohamed-hafez-mohamed/Graduation_Project_2021/blob/master/01-%20Docs/Videos/Phase1.mp4

The second phase was the implementation of gateway and using bootloader with CAN communication instead of UART and update the test case to use systems close to the systems of cars the FOTA was tested was the two systems we stated in chapter 2, dashboard system and collision system and successfully updated both through FOTA.

	Dashboard system	Collison system
Code size	10.1 KB	3.60 KB
Code CRC	0x6C990BF6	0x8697B25D
Node ID	1	0
Flashing time	52 sec	23 sec

The flashing time can decrease a lot when changing UART and CAN speeds, at the tested speeds UART speed was 9600 bit/sec, and CAN with bit rate 500KHZ, the link below contain video of the test.

https://github.com/mohamed-hafez-mohamed/Graduation_Project_2021/blob/master/01-%20Docs/Videos/Phase2.mp4

5.2. What did we achieve?

Like we stated the FOTA technology is very important to develop and integrate with embedded systems solutions and we tried to implement the concept of FOTA in a way that can be applied in different fields, that is why we developed the solution into two phases the first phase was meant to simple applications with one ECU and the second phase was more complex with a system that has more than one ECU like vehicles and we did simulate systems from vehicles to test the solution with more complex software with specific requirements.

We can sum up the achievements as:

1 - Designed a robust bootloader with many features, the most important one is a rollback feature.

2 - Designed the bootloader support both CAN and UART protocols.

3 - Software of the bootloader designed in a modular manner.

4 - Designed and implemented a robust software to the gateway.

5 - Applied some AUTOSAR concepts with the software.

6 - Implemented all the drivers dealing with hardware from scratch.

7 - Made the software portable as possible to support many platforms.

8 - To ensure security of the code we applied encryption and decryption algorithms.

9 - All the software was tested with the hardware.

10 - designed two complex test cases from real life scenarios.

11 - all software was managed by using professional tools like git and GitHub.

The repo link:

https://github.com/mohamed-hafez-mohamed/Graduation_Project_2021

5.3. Overall cost

Here is a table that comprise Hardware components and their price:

Serial no.	Component	Number of pieces	Price LE
1.	Node-MCU(ESP8266 Wi-Fi Programming & Development Kit)	1	140
2.	Stm32f103c8(blue pill) microcontroller based on Arm Cortex-M3	3	3*140
3.	ST-LINK/V2 in-circuit debugger/Programmer	3	3*200
4.	TFT display	1	350
5.	Graphical LCD	1	400
6.	Ultrasonic sensor	1	40
7.	RTC module	1	50
8.	LM35DZ(Temperature Sensor)	1	25
9.	MCP2551(Can bus Transceiver)	3	3*45
10.	Test Breadboards	5	5*25
11.	Cable & Wire Box	1	60
Total cost		16	2345 LE

References

1. Bootloader Design for Microcontrollers in Embedded Systems By Jacob Beningo.

2. AUTOSAR Documents
https://www.autosar.org/standards/classic-platform/

3. STM32F103 datasheet.

4. Jacob Beningo - Reusable Firmware Development_ A Practical Approach to APIs, HALs and Drivers-Apress (2017)

5. ST7735S datasheet for TFT.

6. HCSR04-datasheet-version-1 for ultrasonic.

7. Understanding and Using the Controller Area Networkn Communication Protocol by A. Ghosal, Haibo Zeng, and Marco Di Natale.

8. Think Python: An Introduction to Software Design.

9. Online AES encryption tool:
https://the-x.cn/en-us/cryptography/Aes.aspx
http://aes.online-domain-tools.com/

10. Advanced Encryption Standard (AES) Algorithm
https://www.researchgate.net/directory/publications

11. STM Cryptographic firmware library software
https://www.st.com/en/embedded-software/x-cube-cryptolib.html

Tools

1. KEIL: complete software development environment for a range of Arm Cortex-M based microcontroller devices, helps with debugging.

2. Arduino IDE: it is an open-source IDE makes it easy to write code and upload it to the board, used for NodeMCU.

3. Python 3.8: used to create the GUI as it is a stable release of Python.

4. Visual studio code: as an editor.

5. STM32 ST-LINK Utility: used to burn code to microcontroller.

6. Draw.io: to draw software diagrams.

7. Git & GitHub: for version control.