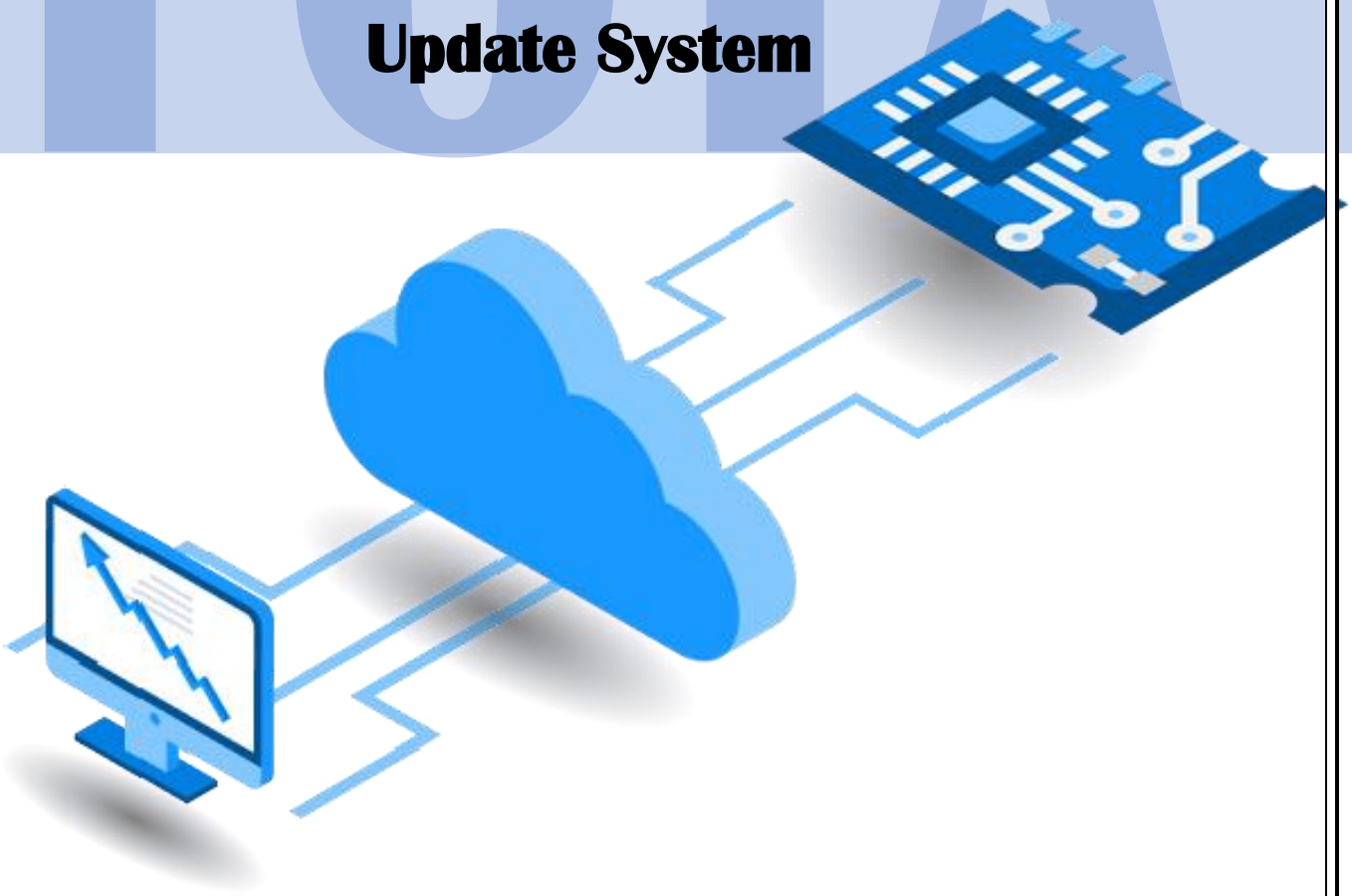
Ministry of Communications and Information Technology

egypt makes electronics

NTI National Telecommunication Institute

Firmware Over-the-Air

Update System



Contents

1. Abstract

In our rapidly advancing technological era, embedded systems and the Internet of Things (IoT) have seamlessly intertwined with various aspects of our lives, ushering in a new wave of innovation. This relentless pursuit of progress aims to enhance user experiences with devices encountered in daily life. The inherent complexity associated with increased features and requirements typically poses challenges in system design. However, the use of Embedded Arm technology, a focal point in our project, defies expectations, providing a solution that efficiently manages and adapts to evolving technologies.

As technological advancements continue to unfold, a responsive system becomes imperative to keep pace with the dynamic landscape. Our project focuses on the utilization of Firmware Over-the-Air (FOTA), a pivotal feature that establishes a cloud-based connection for devices. This connection enables seamless communication with the manufacturer, facilitating regular updates via Over-the-Air (OTA) transmissions. The essence of FOTA lies in ensuring that devices remain aligned with the latest technological developments. Beyond updates, FOTA serves as a mechanism for users to safeguard their devices, detecting and alerting them to both software and potential hardware issues.

In the not-so-distant past, software maintenance and updates incurred substantial time and costs. FOTA has revolutionized this landscape, streamlined the process and significantly reducing associated expenses. Notably, the automotive industry stands as a prominent beneficiary of FOTA, with our team contributing to this domain. Our efforts extend beyond conventional FOTA implementation; we have incorporated enhanced security features to fortify the development cycle. Additionally, we prioritize user-friendliness, ensuring that these advanced features are easily accessible and navigable.

This project encapsulates our commitment to advancing embedded systems, particularly in the realm of FOTA, where security and user convenience converge. As we navigate the intricacies of this venture, we aspire to contribute meaningfully to the continual evolution of technology in the automotive sector and beyond.

2. Introduction

In the ever-evolving landscape of embedded systems, the ability to seamlessly update firmware over the air has become a pivotal requirement for maintaining and enhancing the functionality of devices deployed in the field. This documentation serves as a comprehensive guide to the design, implementation, and deployment of a Firmware Over-the-Air (FOTA) update system utilizing MQTT (Message Queuing Telemetry Transport) as the communication protocol.

2.1 Project Overview

The FOTA update system presented herein comprises a robust architecture with distinct components, including a Node-RED-based Publisher for firmware file uploads, a Mosquitto MQTT Broker for efficient message routing, and an ESP8266-powered Subscriber responsible for transferring firmware updates to the target Embedded Control Unit (ECU) using UART (Universal Asynchronous Receiver-Transmitter) protocol.

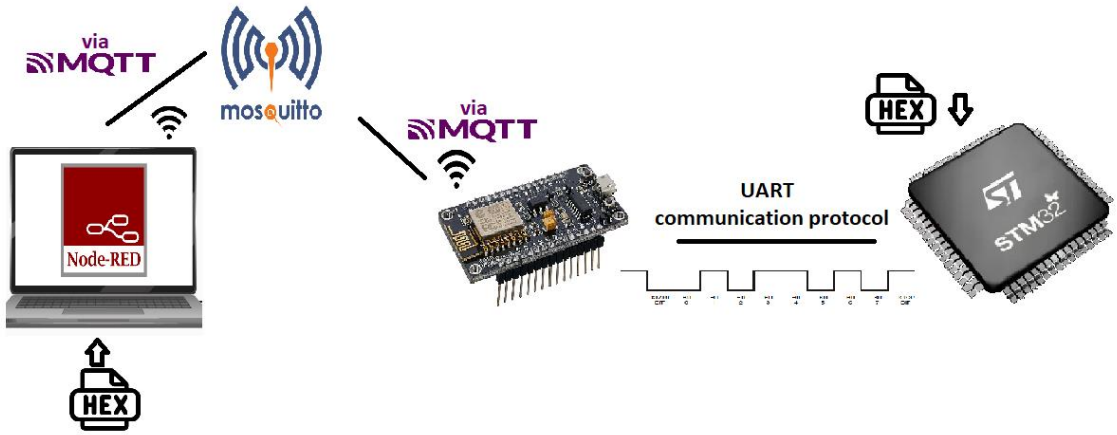


Figure 1: Project layout

The initial segment of our project, the Internet of Things (IoT) part, serves as the user's portal for initiating the firmware update process. Users interact with an intuitive Graphical User Interface (GUI) tool embedded in Node-RED, seamlessly uploading firmware files to the server. Leveraging MQTT, a lightweight and efficient messaging protocol, the firmware traverses the communication channel from Node-RED to the ESP8266 through the Mosquitto broker. Once securely stored in the ESP8266's RAM, the firmware is poised for the subsequent phases of our FOTA update system.

At the heart of our FOTA project, the ESP32 acts as the communication hub, bridging the gap between the user-initiated firmware updates and the ultimate implementation on the target ECU. The ESP8266, a versatile and cost-effective Wi-Fi module, plays a pivotal role in our Firmware Over-the-Air (FOTA) update system. Its integration into the project adds a layer of connectivity, intelligence, and adaptability, enabling seamless communication between the MQTT broker and the target Embedded Control Unit (ECU).

//////// bootloader passage /////////

2.2 Objective

The primary objective of this project is to enable the remote and secure updating of firmware in embedded systems. The update is uploaded to the server using Node-Red GUI. The encryption process takes place in Node-Red before the update is sent to the ESP using MQTT. Through MQTT, a lightweight and efficient messaging protocol, we establish a reliable communication channel that facilitates the seamless transfer of firmware updates over the air. The utilization of ESP8266 as a Wi-Fi module ensures flexibility and compatibility with a wide range of embedded systems. Once the ESP receives the secure firmware, the decryption process begins to deliver the actual hex file to be delivered to the target ECU (stm32 cortex M4) through UART communication protocol and to be flashed on the target ECU to eventually update the old firmware with the updated version.

2.3 Document Structure

This documentation is structured to provide a comprehensive understanding of the FOTA update system, encompassing design considerations, implementation details, security measures, testing methodologies, and best practices. Each section is tailored to guide both developers and stakeholders through the various aspects of the project, offering insights into the decision-making process and technical intricacies.

As we delve into the intricacies of FOTA updates using MQTT, we invite readers to explore the innovative features, challenges overcome, and lessons learned during the development of this robust and scalable firmware update system.

3.0 IoT

In such a case, performing any firmware update could be a challenging task that involves multiple revisions and modifications. Thus, an over-the-air update through a wireless network eases the task while lowering any additional cost and time consumed for multiple software firmware updates during the entire life cycle of a car. IHS automotive had estimated that the total OEM cost savings from OTA (firmware and software) update process will grow to more than$35billion in 2022 from $2.7 billion in 2015. The components used in the project are discussed in details below.

3.1 GUI Tool in Node-RED

The GUI tool in Node-RED (Publisher) provides users with a user-friendly interface, facilitating the seamless upload of firmware files. It serves as the control center, initiating firmware updates and acting as the interface connecting users with the underlying FOTA infrastructure.

The light-weight runtime is built on Node.js, taking full advantage of its event-driven, non-blocking model. This makes it ideal to run at the edge of the network on low-cost hardware such as the ESP8266 as well as in the cloud. With over 225,000 modules in Node's package repository, it is easy to extend the range of palette nodes to add new capabilities.

The main advantage of Node-Red is that it allows having a local host freely and without hassling in having a cloud server/host. Also, the variety of blocks in Node-Red provides massive potential functionalities that can help in any further development.

In our project we use the node red GUI to open the user a window to upload the desired updated firmware, Node-red must be built using blocks that illustrates the objective required from it, the following figure shows the required blocks needed to achieve the goal.

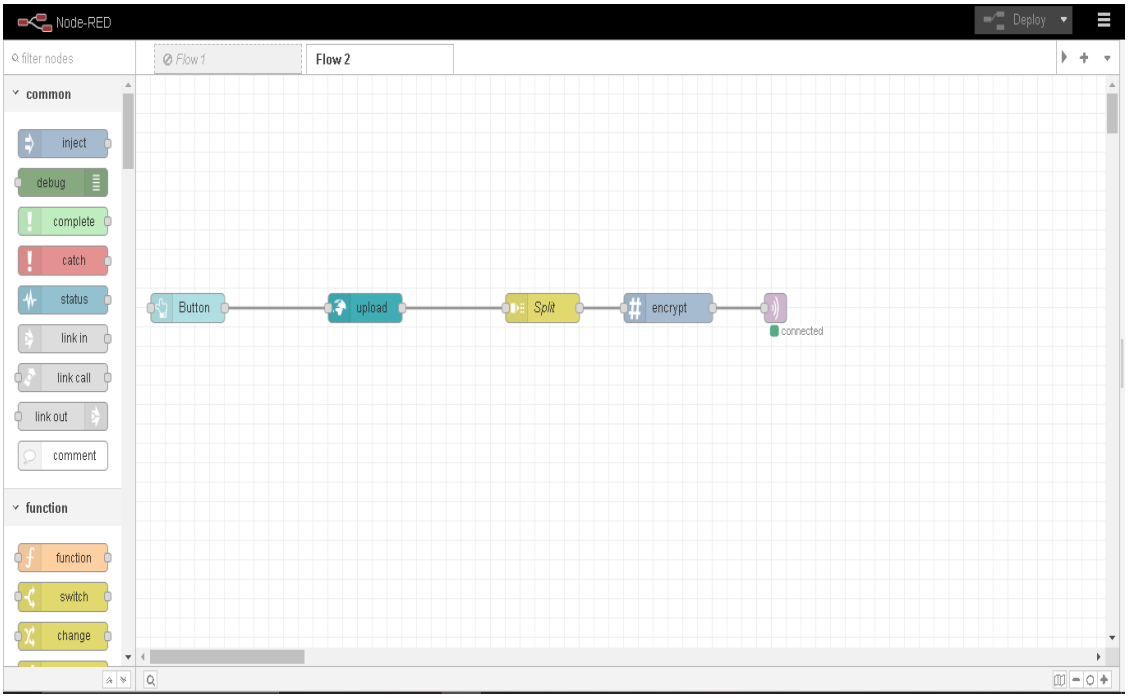


Figure 2: NODE-RED workplace showing the used blocks

The flow of our project has been created using five blocks as follows:

1. **Button:**
    - **Purpose:** The Button node is used to create a clickable button in the Node-RED dashboard. It provides a simple way to trigger an action or a sequence of actions when the button is pressed.
    - **Library:** The Button node is part of the "dashboard" library in Node-RED.
    - **Essential Information:** Configure the button's label, color, and the action to be performed when the button is clicked.
2. **Upload:**
    - **Purpose:** The Upload node is used to upload files through the Node-RED dashboard. It's commonly used when you want to allow users to upload files and process them within your flow. It exempts us from the usage of cloud servers or google storage.
    - **Library:** The Upload node is also part of the "dashboard" library.
    - **Essential Information:** Specify the target folder for file uploads and configure additional settings such as file types allowed.
3. **Split:**
    - **Purpose:** The Split node is used to split an array or a message property with multiple values into separate messages. It's useful when you want to process each element of an array individually in your flow.
    - **Library:** The Split node is part of the core Node-RED nodes, so it doesn't belong to a specific library.
    - **Essential Information:** Configure the property containing the array, and the Split node will output a separate message for each array element.
4. **Encrypt:**
    - **Purpose:** Encryption nodes are typically not standard in Node-RED, but they can be added through additional nodes or custom nodes. Encryption is used to secure sensitive data by converting it into a format that is not easily readable without the appropriate decryption key.
    - **Library:** The specific encryption node you use will determine the library it belongs to. Common encryption libraries include crypto, bcrypt, or other encryption modules in Node.js.
    - **Essential Information:** Configuration will depend on the encryption node you choose. You'll typically need to provide details such as the algorithm, key, and input data.
5. **MQTT Out:**
    - **Purpose:** The MQTT Out node is used to send messages(publish) to an MQTT broker. MQTT (Message Queuing Telemetry Transport) is a lightweight messaging protocol commonly used in IoT applications.
    - **Library:** The MQTT Out node is part of the core Node-RED nodes, so it doesn't belong to a specific library.
    - **Essential Information:** Configure the MQTT broker connection details, topic, and payload to be sent. This node is crucial for integrating Node-RED with MQTT-based communication in IoT scenarios.

6. **Authentication**
   UserName, Password

3.2 MQTT Communication

MQTT, a lightweight IoT protocol, seamlessly integrates with Node-RED for Over-The-Air (OTA) firmware updates on ESP8266 via the Arduino IDE. In Node-RED's visual interface, an MQTT Out block configures broker details, topic, and payload for firmware updates. On the ESP8266, the Arduino IDE facilitates the MQTT communication using libraries like "PubSubClient" to subscribe to the designated topic. Additionally, ESP8266's ArduinoOTA library ensures a secure and reliable OTA update process. As Node-RED triggers an OTA firmware update, it publishes the update message via MQTT. The ESP8266, with MQTT and ArduinoOTA integration, receives and successfully implements the firmware update, showcasing the synergy between Node-RED and ESP8266 in enabling user-friendly, remote firmware management for IoT devices. This comprehensive approach ensures a seamless and secure pathway for delivering firmware updates to ESP8266 devices, enhancing their flexibility and ease of remote management.

3.3 Mosquitto broker

In the context of Over-The-Air (OTA) firmware updates for the ESP8266 Wi-Fi module facilitated by Node-RED and the MQTT protocol, the Mosquitto MQTT broker plays a pivotal role in ensuring seamless communication. Mosquitto acts as the intermediary between Node-RED, serving as the user interface, and the ESP8266 device awaiting firmware updates.

In the Node-RED graphical user interface (GUI), an MQTT Out node is configured to establish a connection with the Mosquitto broker. This node is responsible for transmitting the firmware update payload, along with essential information such as the topic and broker details. As the user initiates an OTA firmware update from Node-RED, the MQTT Out node publishes the update message to the specified MQTT topic.

Simultaneously, the ESP8266, programmed using the Arduino IDE, is equipped with an MQTT client using libraries like "PubSubClient." The ESP8266 subscribes to the designated MQTT topic, awaiting incoming messages. Upon the publication of the firmware update message by Node-RED, Mosquitto relays this message to the ESP8266 through the subscribed MQTT topic.

To ensure the security and integrity of the OTA process, the ArduinoOTA library is employed on the ESP8266. This library facilitates the over-the-air update mechanism, allowing the ESP8266 to receive the firmware update, extract the payload, and apply the update without requiring a physical connection. The MQTT protocol acts as the conduit, efficiently transmitting the firmware update commands from Node-RED to the ESP8266, all orchestrated by the Mosquitto broker.

This orchestrated interaction between Node-RED, Mosquitto MQTT broker, and the ESP8266 through MQTT ensures a reliable, secure, and user-friendly approach to remotely managing firmware updates. The MQTT protocol, coupled with the Mosquitto broker, becomes a powerful

enabler for the seamless flow of information, allowing users to effortlessly conduct Over-The-Air firmware updates for ESP8266 devices via the Node-RED GUI.

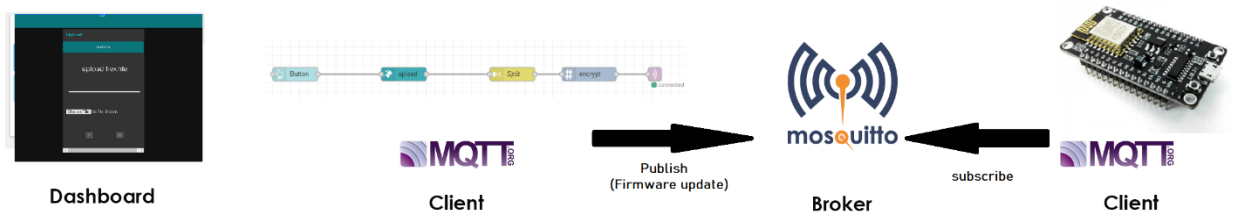


Figure 3: Relation between IoT components in details

In conclusion, the collaboration between Node-RED, Mosquitto MQTT broker, and the ESP8266 in facilitating Over-The-Air (OTA) firmware updates establishes a robust and efficient framework for remote device management. Node-RED's intuitive graphical interface simplifies user interaction, while the MQTT protocol serves as the communication backbone. The Mosquitto MQTT broker seamlessly bridges Node-RED and the ESP8266, orchestrating the transmission of firmware update commands.

Node-RED's MQTT Out node, configured within its GUI, acts as the trigger point for initiating firmware updates. This node communicates with the Mosquitto broker, publishing update messages containing vital information like the firmware payload and topic details. On the ESP8266 side, the Arduino IDE, coupled with MQTT client libraries like "PubSubClient" and the ArduinoOTA library, ensures a secure and reliable reception of these updates.

Mosquitto MQTT broker stands as the linchpin, efficiently relaying information between the Node-RED GUI and ESP8266. It establishes a streamlined communication channel that allows for the seamless flow of firmware update commands, ultimately enabling users to remotely and securely manage firmware updates for their ESP8266 devices. This cohesive integration of tools not only enhances user convenience through Node-RED's visual interface but also showcases the power of MQTT and Mosquitto in facilitating flexible, real-time, and secure Over-The-Air firmware updates for IoT devices like the ESP8266.

Some modifications need to be made to the configuration file in the mosquito medium in order to get the large hexagon shape correctly. The initial enhancement involves enabling the broker to listen on our secure port, namely "8883". Subsequently, we adjusted the configuration to accommodate anonymous connections, ensuring proper functionality on the newly assigned port.

Following this, we modified the maximum queued messages to align with the quantity of hex registers being transmitted, addressing limitations associated with QOS 1. This adjustment is crucial to mitigate potential losses of transmitted messages, particularly when utilizing a partitioned Node-Red block (e.g., sending hex registers individually).

The last alteration involves setting the maximum onboard messages to zero. This precautionary measure allows for an unlimited number of messages that may be in progress during the handshake or those undergoing retries, eliminating any set maximum.

3.4 Challenges

NODE-RED

Limitation in Uploading Blocks

Issue: The Upload node faces a limitation when attempting to send the entire hex file (in kilobytes) when connected solely to the MQTT-out node.

Resolution: Utilizing the Split node, the hex file can be divided into records, allowing for the transmission of the complete hex file record by record. This approach ensures the successful transfer of the entire hex file, regardless of its size.

Encryption Node

Issue: The encryption node supports multiple algorithms like AES or DES encryption but lacks specific categorization. This absence makes decryption challenging, as, for instance, AES is referenced without specifying the algorithm, such as 128.

Solutions:

1. Download an additional node for AES encryption from the package named "node-red-contrib-crypto-blue," specifically a node called Cipher. This allows for the selection of the desired algorithm.

2. Opt for DES encryption, as it can be decrypted using libraries available in the Arduino IDE.

Mosquitto Broker

Connection Issues with Clients ("Node_Red" publish, ESP "subscribe")

Issue: The MQTT-out node fails to connect to the broker, as does the ESP.

Solution: Edit the Mosquitto configuration file.

1. Configure the broker to listen to port 8883 using the line [listener 8883].

2. Modify the line [allow_anonymous true] to permit clients to connect to the broker.

Incomplete Hex File Reception

Issue: When dealing with a hex file consisting of 3000 records, the system receives a random number of records, typically above 2500, but never the entire file.

Solution: Further adjustments to the configuration file are required.

1. For the example mentioned, edit the line [max_queued_messages 3000], precisely indicating the number of messages sent through QOS1.

2. Optionally, modify the line [max_inflight_messages 0] to eliminate any maximum restriction.

4. ESP32 Dev Modual

In the dynamic landscape of embedded systems and the Internet of Things (IoT), the ESP32 emerges as a key enabler in our Firmware Over-the-Air (FOTA) update system. This section delves into the integral role played by the ESP32, a versatile and open-source IoT platform, in facilitating seamless communication, secure updates, and enhanced user interactions.

4.1 Handling Hex File in ESP32

One of the critical tasks performed by the ESP32 in our FOTA project is the handling of the hex file. Here's a detailed breakdown of this process:

4.1.2 Firmware Storage

Upon receiving the firmware update via MQTT, ESP32 stores the hex file securely in its file system. The SPIFFS (Serial Peripheral Interface Flash File System) is commonly used for this purpose. SPIFFS allows ESP32 to manage and organize files in its flash memory, ensuring efficient storage and retrieval of firmware updates.

4.1.3 Decryption and Processing

Once the hex file is securely stored, ESP32 initiates the decryption process using the embedded decryption key. This step is crucial for converting the encrypted hex file into a format that can be effectively transmitted to the target ECU.

4.1.4 UART Communication with Target ECU

With the decrypted firmware ready, ESP32 establishes UART communication with the target Embedded Control Unit (ECU). This communication channel, typically employing the UART protocol, ensures the reliable transfer of the firmware update to the ECU.

5. ECU Bootloader

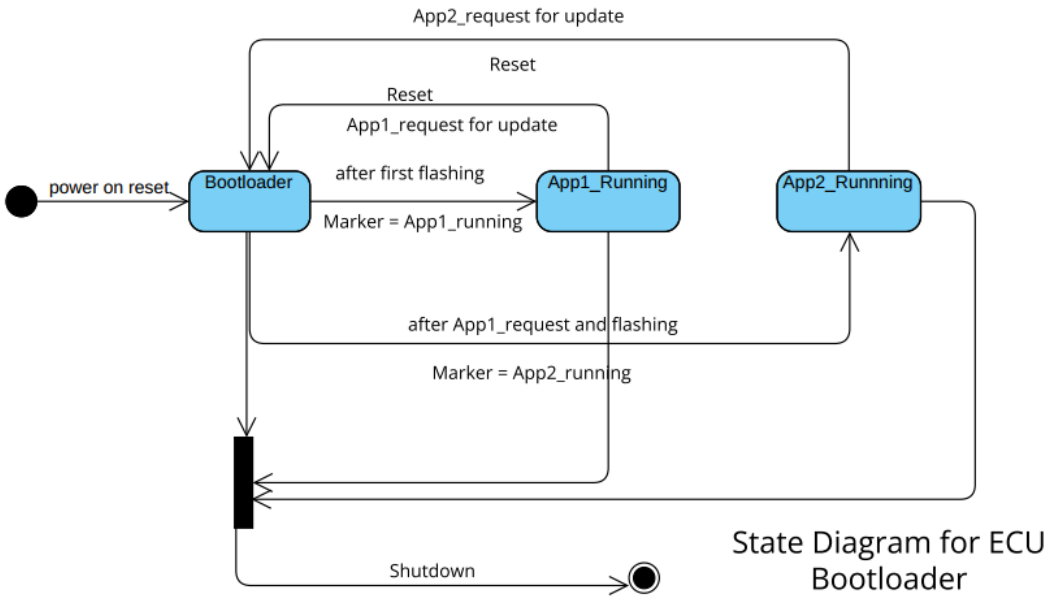


Figure 4: ECU Bootloader State Diagram.

5.1 Bootloader state :

Step 1 :

It initialize the system clock and peripherals (UART, GPIO,..,and SysTick).

Step 2 :

It checks the status of the system by comparing the ROM_Marker with the predefined states of the system (NO_APP , APP1_Running , APP2_Running , APP1_request, and APP2_request).

Step 3:

It takes a decission according to the state of the system:

* NO_APP → The system will remain in the bootloader until receiving application for the first time.

* APP1-Running → The system will jump to application 1 and application 1 will work until receiving an update request from the NodeMCU ESP82 .

* APP2-Running → The system will jump to application 2 and application 2 will work until receiving an update request from the NodeMCU ESP82 .

* APP1_request → The system will start communication with NodeMCU to receive new application and flashing it in the application 2 sectors then jumping to run application 2 if the flashing has been done successfully and let application 1 as a backup application but if the flashing is faild , the system will jump to run application 1 .

* APP2_request → The system will start communication with NodeMCU to receive new application and flashing it in the application 1 sectors then jumping to run application 1 if the flashing has been done successfully and let application 2 as a backup application but if the flashing is faild , the system will jump to run application 2.

5.2 APP1_Running :

The application 1 is running until receiving request from NodeMCU ESP82 .

After receiving request , it will change the ROM_Marker to APP1_request and jump to the bootloader .

5.3 APP2_Running :

The application 2 is running until receiving request from NodeMCU ESP82 .

After receiving request , it will change the ROM_Marker to APP2_request and jump to the bootloader .

6. V-model

Software require

- Design server tool that is the first step to upload file with simple GUI.
- Implement bootloader code that take hex file from the point (ESP) between server and STM MCU and flashing it over the air on MCU.
- Implement the bootloader app by RTOS design.
- Implement simple app that simulating the signs direction leds in real car.

Validation Test Plan:-

(Test all system as a black box)

Software Design

HIGH Level Design

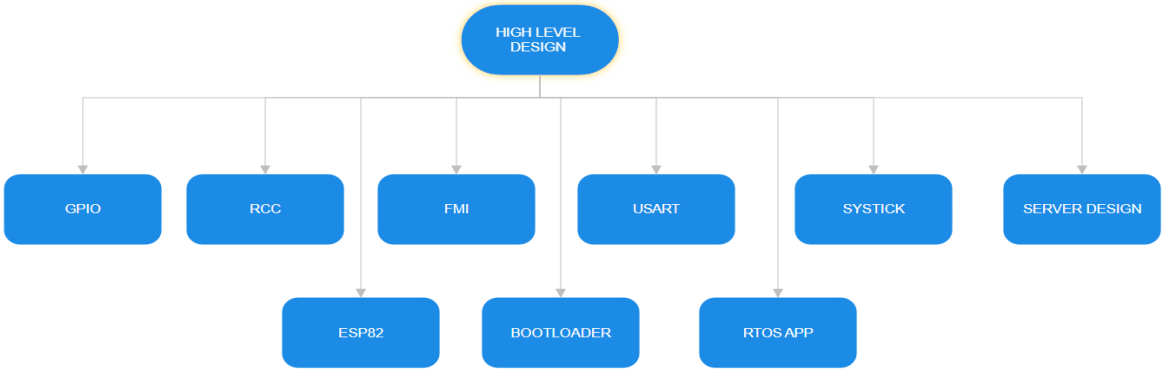


Figure 6 - Software Design

GPIO APIs:-

```
void MGPIO_vInit(MGPIO_Config_t* A_xPinConfig)
// configuration in link time of pin by passing struct
typedef struct
{
      u8 Port ;
      u8 Pin;
      u8 Mode;
      u8 OutputType;
      u8 OutputSpeed;
      u8 InputPull;
      u8 AltFunc;

}MGPIO_Config_t;
```

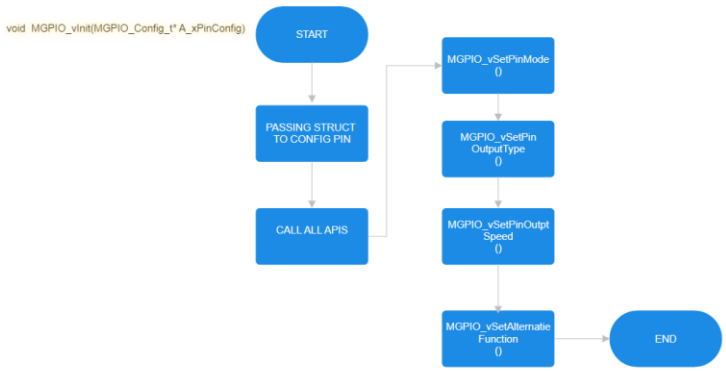


Figure 6- GPIO_INTIAL

RCC APIs:-

```
void MRCC_vInit(void)

// Define system clock
```

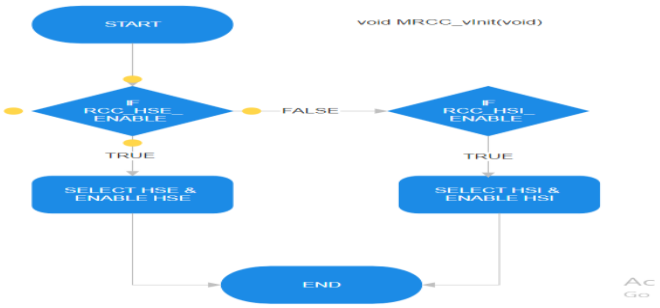


Figure 7-RCC INTIAL

```
void MRCC_vEnableClock(u32 A_u32BusId, u32 A_u32PeripheralId)
// Enable peripherals clock
```

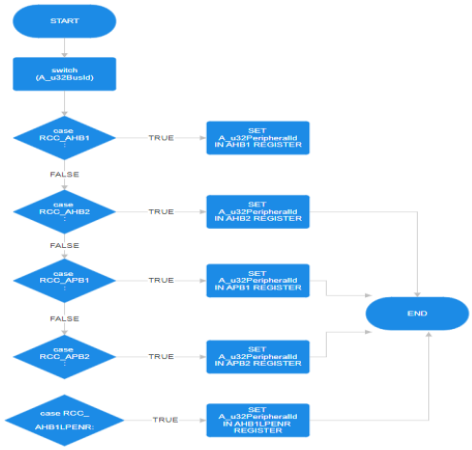
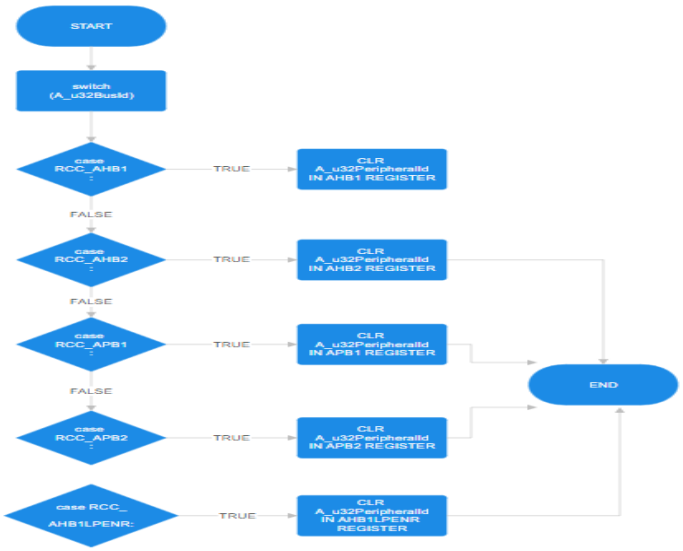


Figure 8-RCC Enble clock algorithm

```
Void MRCC_vDisableClock(u32 A_u32BusId, u32 A_u32PeripheralId)
// Disable peripherals clock
```

Figure 9- RCC Disable clock algorithm



Flash Memory Driver (FMI) APIs:-

```
void FMI_vSectorErase(u8 A_u8SectorNo)
// Earse sector area
```

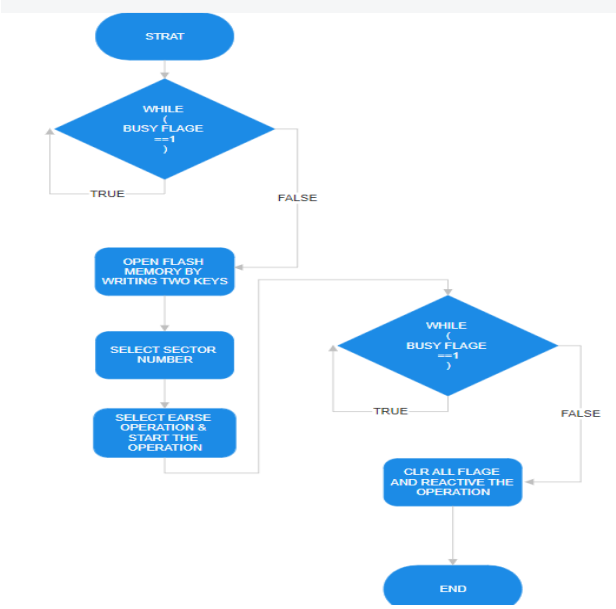


Figure 10- FMI erase sector algorithm

```
void FMI_vEraseAppArea(u8 FROM , u8 TO)
//  Earse selected area
```

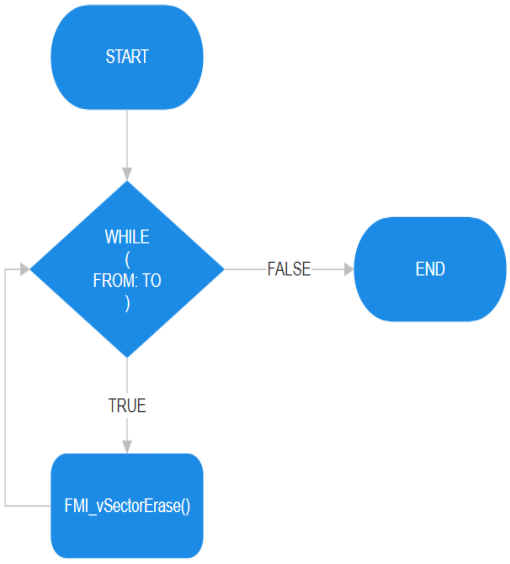


Figure 11-FMI AppErea erease algorithm

```
//void FMI_vFlashWrite(u32 A_u32Address,u16* A_pu16Dat, u16
A_u16Length)
// Flash and update marker area
```

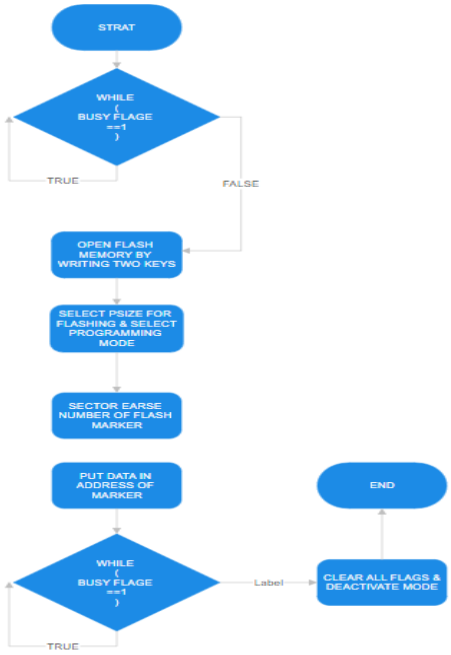


Figure 12-FMI FlashWrite algorithm

- USART APIs:-

```
void MUSART_voidInit(USART_InitType *A_InitStruct)
// Initial configuration of USART by passing struct as a link time
config
```

```
typedef struct
{
	u32 BaudRate;
	u8 DataWidth;
	u8 StopBits;
	u8 TransferDirection;
	u8 Oversampling;
}USART_InitType;
```

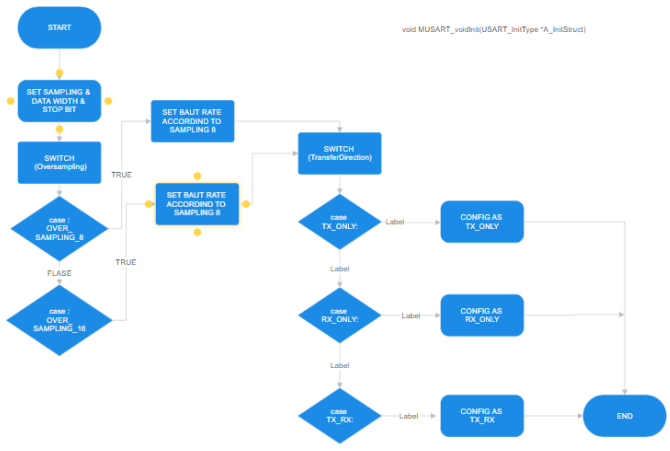


Figure 13-UART Initialization

```
void MUSARTAnd_Interrupte_Enable(void)
```
// Enable USART and Interrupte

```
void MUSARTAnd_Interrupte_Disable(void)
```
// Disable USART and Interrupte

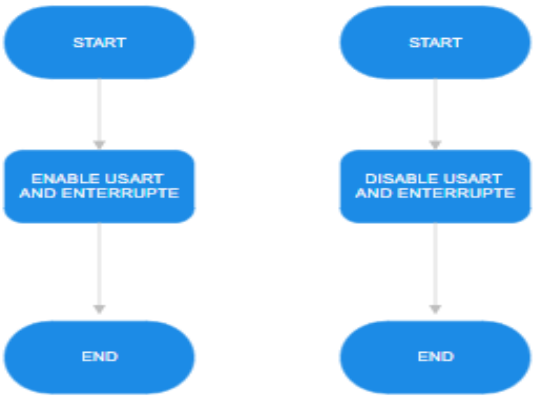


Figure 14-UART Interrupt enable and disable algoritm

```
void MUSART_voidTransmitByte (u8 A_u8Byte)
```
// Transmate single byte

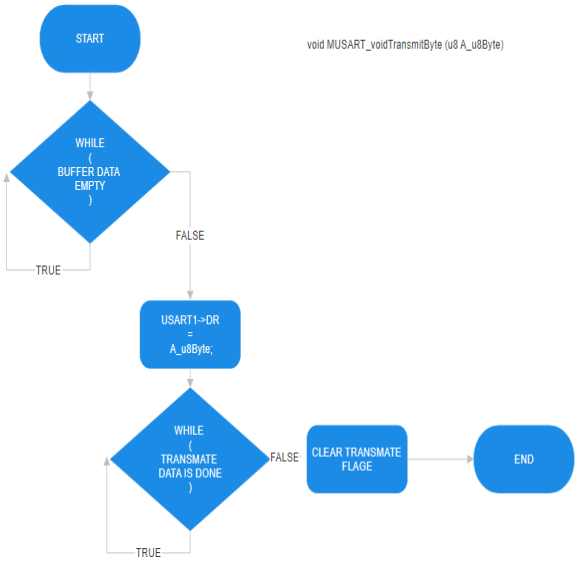


Figure 15-UART TrasmitByte algorithm

```
void MUSART_voidTransmitString ( u8 *A_ptru8String )
// Transmate string
```

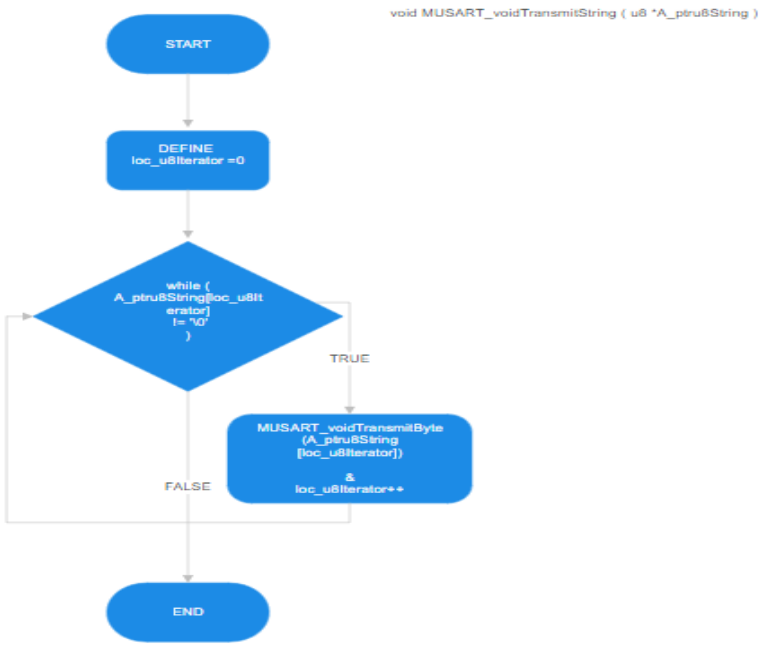


Figure 16-UART Trasmit String algorithm

6.1.1 Systick APIs:-

```
void MSTK_vInit(void)
// Define systick clock to config tick time
```

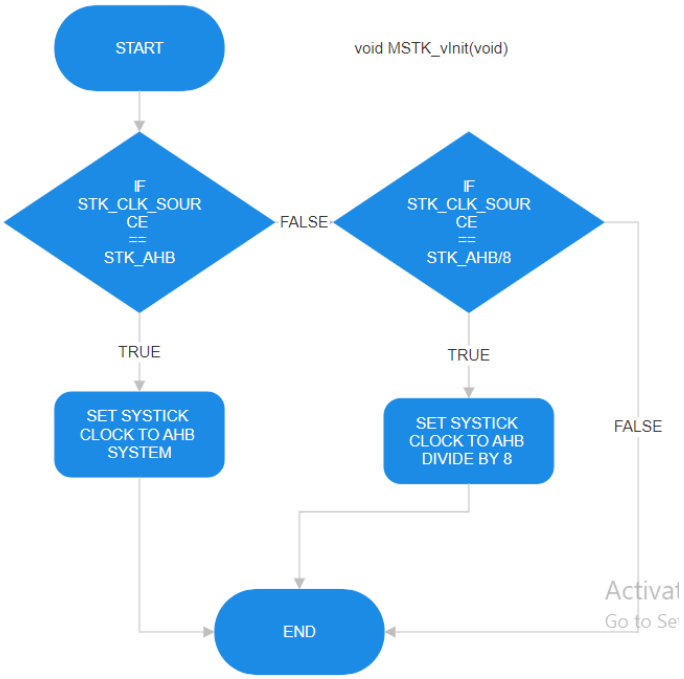


Figure 17- SYSTICK INIT

```
void MSTK_vResetTimer(void)
// Reset time to count from zero
```

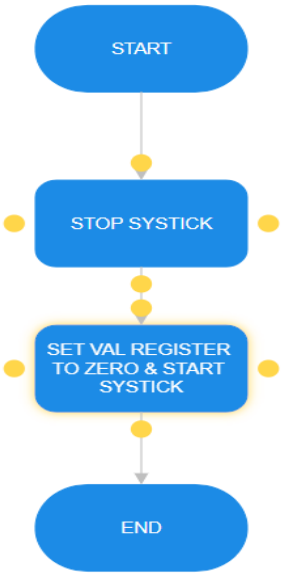


Figure 18 - SYSTICK RESET

```
void MSTK_vStoptimerAnd_Enterrupte(void)
// Disable enterrupte and stop systick
```

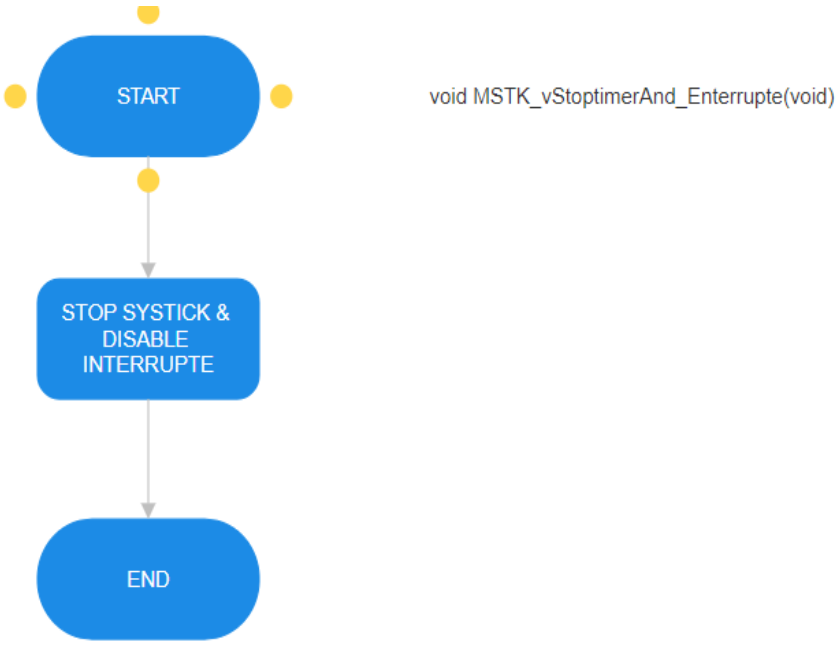


Figure 19- STOP SYSTICK AND INTERRUPTE

```
void MSTK_vSetInterval_single
(u32 A_u32Ticks, void(*CallbackFunction)(void))
// Define time to call app after it
```

Figure20 - SET CALL BACK FUNCTION AND START SYSTICK

Parse APIs:

```
void HexParser_vParseData_2D(u8 (*A_pu8Data)[50])
// Task 2D array and flashing it
```

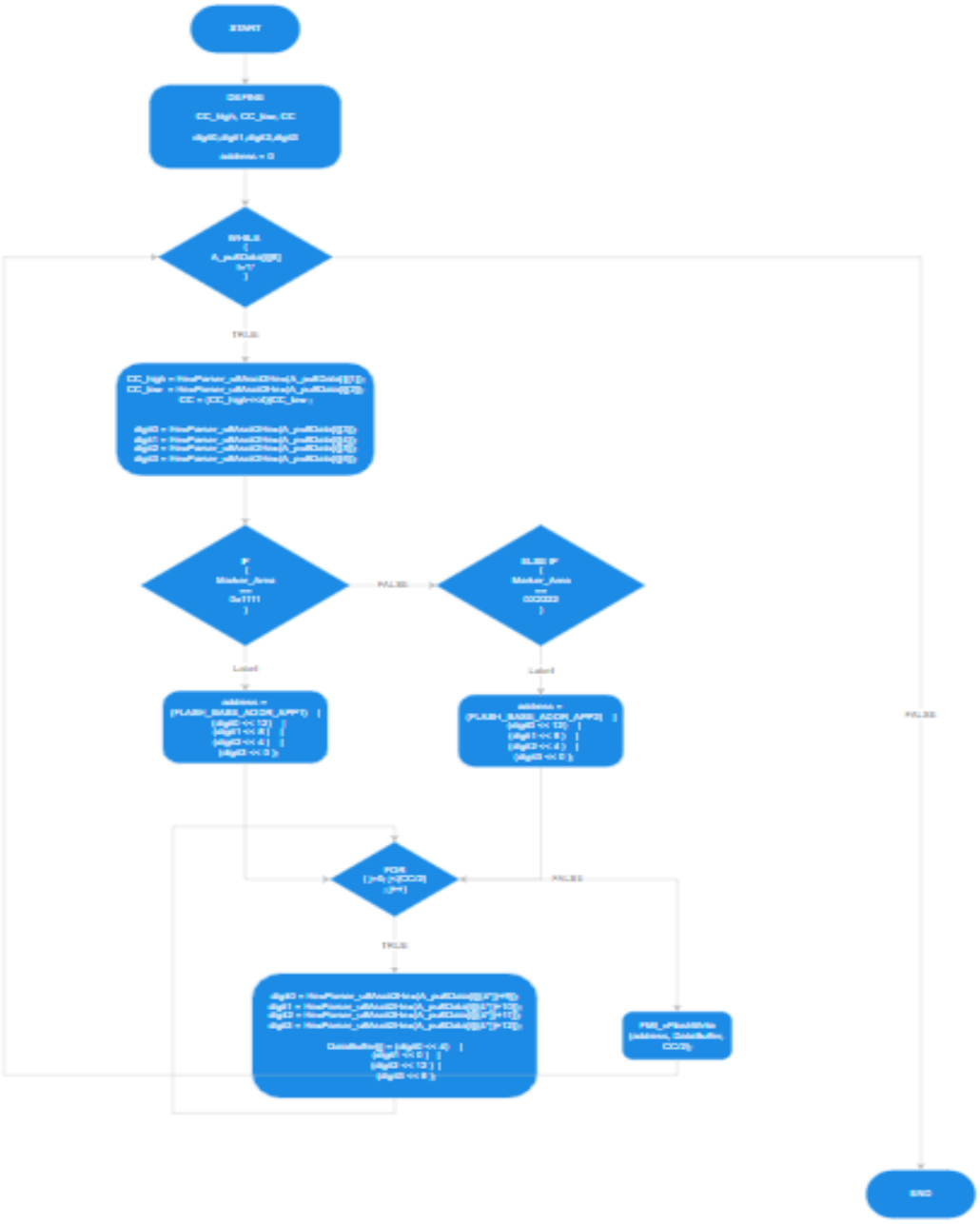


Figure 21-HexParser to parse data in 2D array algorithm

Low Level Design

(Static and private APIs)

```
static void MGPIO_vSetPinMode(u8 A_u8PortId,u8 A_u8PinNo ,u8 A_u8Mode)
// Config pin mood
```

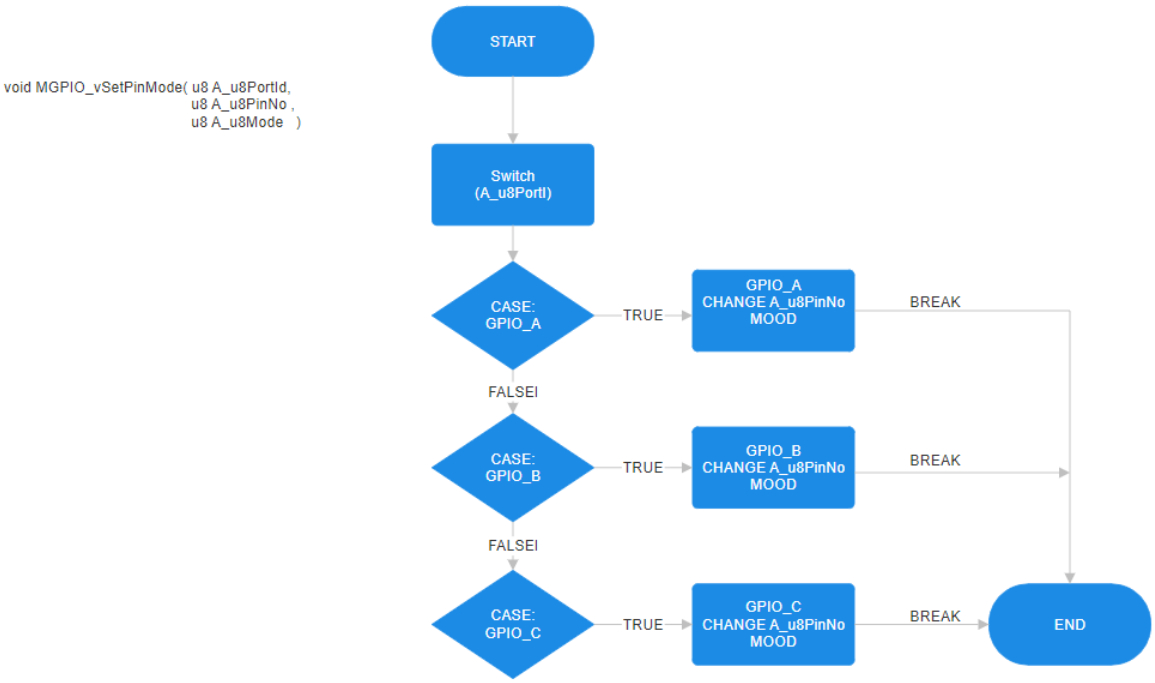


Figure 22-GPIO Set piin mode algorithm

```
static void MGPIO_vSetPinOutputType(u8 A_u8PortId,u8
A_u8PinNo,u8A_u8OutType)
// Config output type
```

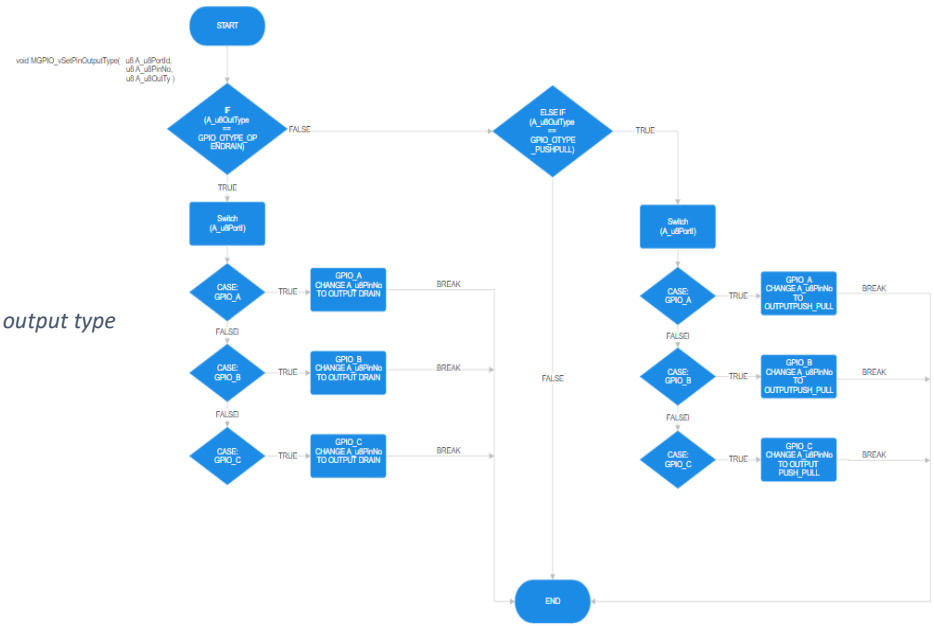


Figure 23-GPIO set pin output type algorithm

```
static void void MGPIO_vSetPinOutputSpeed(u8 A_u8PortId,u8 A_u8PinNo
,u8A_u8OutSpeed)
// Config output speed
```

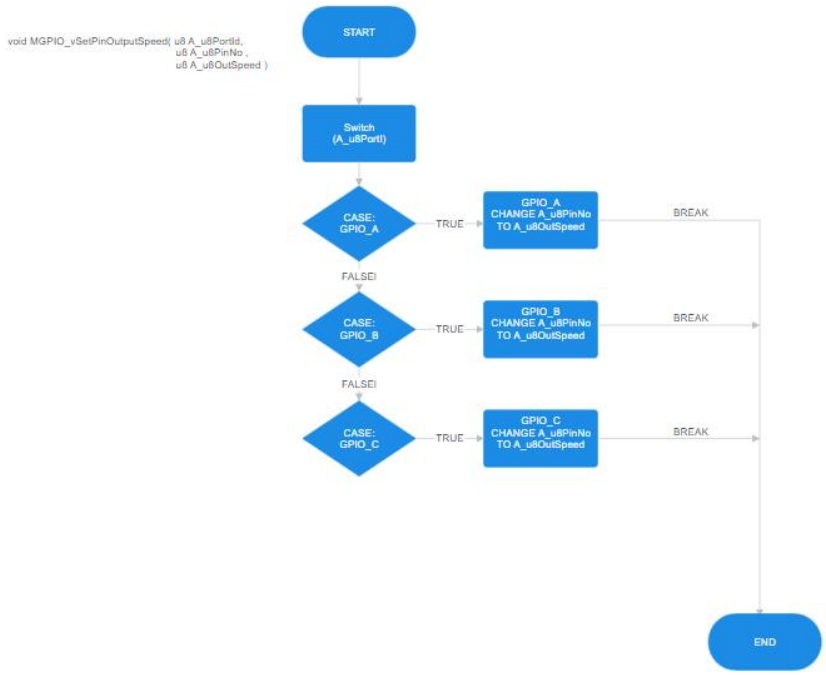


Figure 24- GPIO set pin output speed

```
static void MGPIO_vSetAlternativeFunction(u8 A_u8PortId,A_u8PinNo,u8
A_u8AltFun)
// Config AlternativeFunction for pin
```

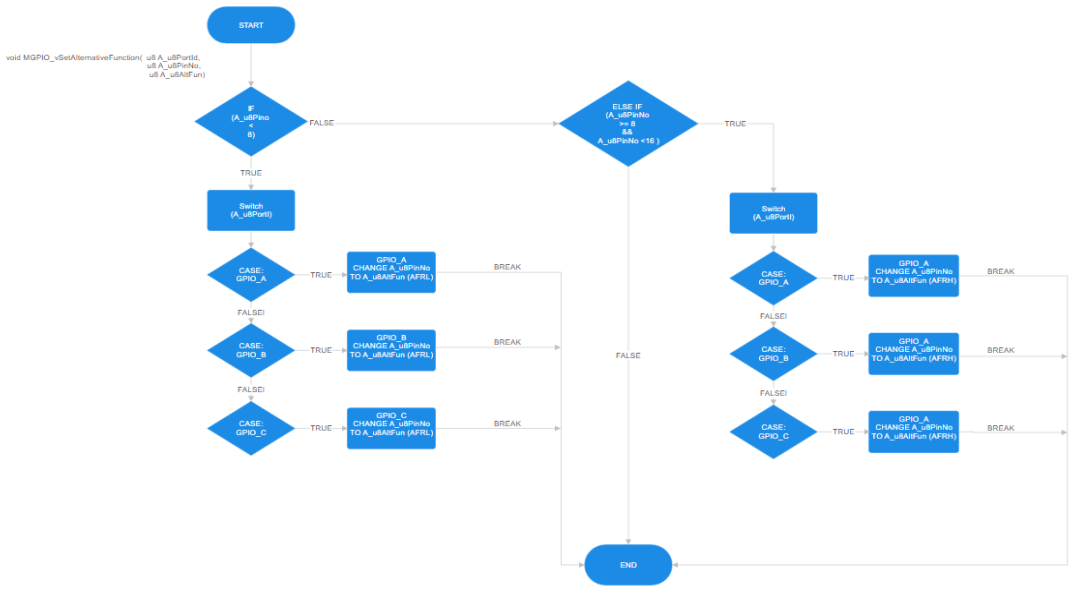


Figure 25-GPIO set alternative function algorithm

```
static void FMI_vFlashWrite(u32 A_u32Address,u16* A_pu16Data,u16
A_u16Length)
// Flashing new record in it address
```

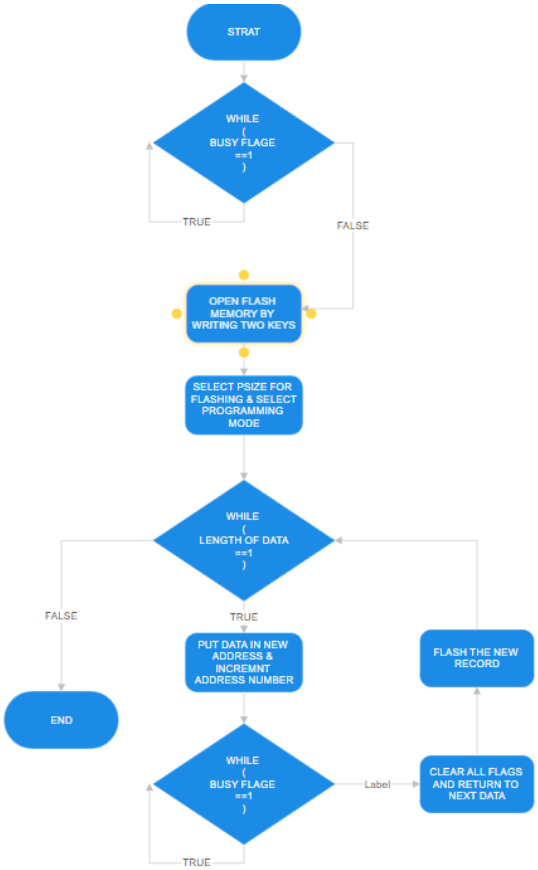


Figure 26-FMI flash write algorithm

```
static u8 HexParser_u8Ascii2Hex(u8 A_u8Ascii)
// Convert from ASCI to decimal
```

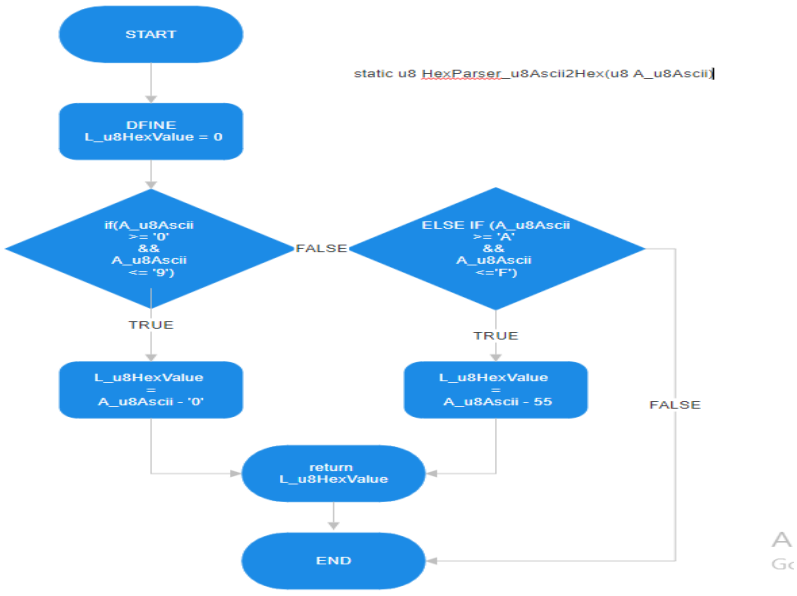


Figure 27-HexParser convert ascii to hex algorithm

- Integration Test Plan

- Test GPIO With RCC
- Test GPIO With RCC With Systick
- Test GPIO With RCC With Systick With Flash Driver
- Test GPIO With RCC With Systick With Flash Driver With Parse App
- Test GPIO With RCC With Systick With Flash Driver With Parse App With USART
- Test Bootloader Code
- Test Bootloader Code That Implemented By RTOS
- Test Server
- Test Server With ESP82
- Test Server With ESP82 With USART
- Test Server With ESP82 With USART With Bootloader Code
- Test RTOS App
- Test Server With ESP82 With USART With Bootloader Code With RTOS App

6.2 Components design

In low level design implemented all function as flow charts.

ESP82 Flow chart:-

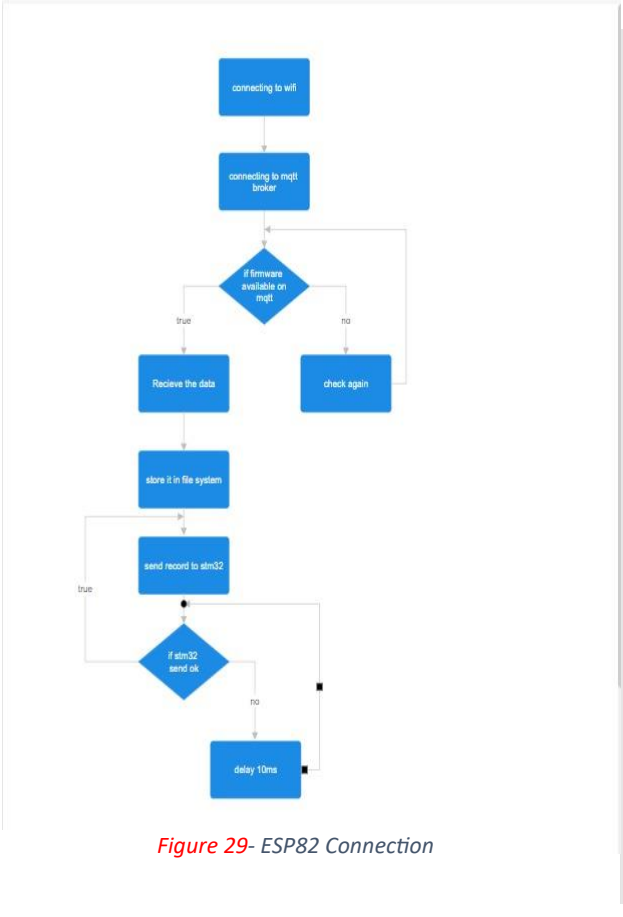


Figure 29- ESP82 Connection

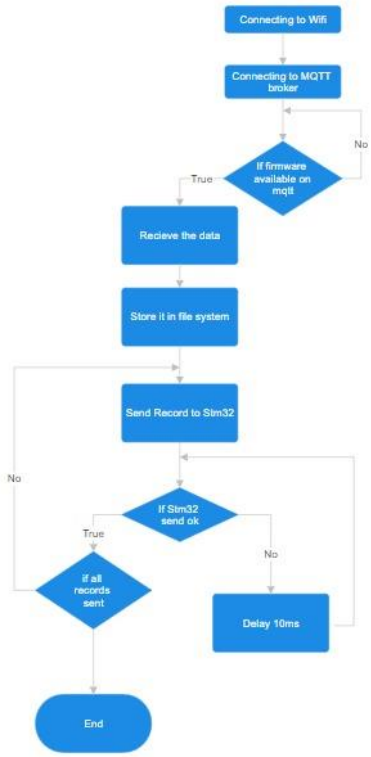


Figure 28- ESP82 Diagram

Bootloader Flow chart

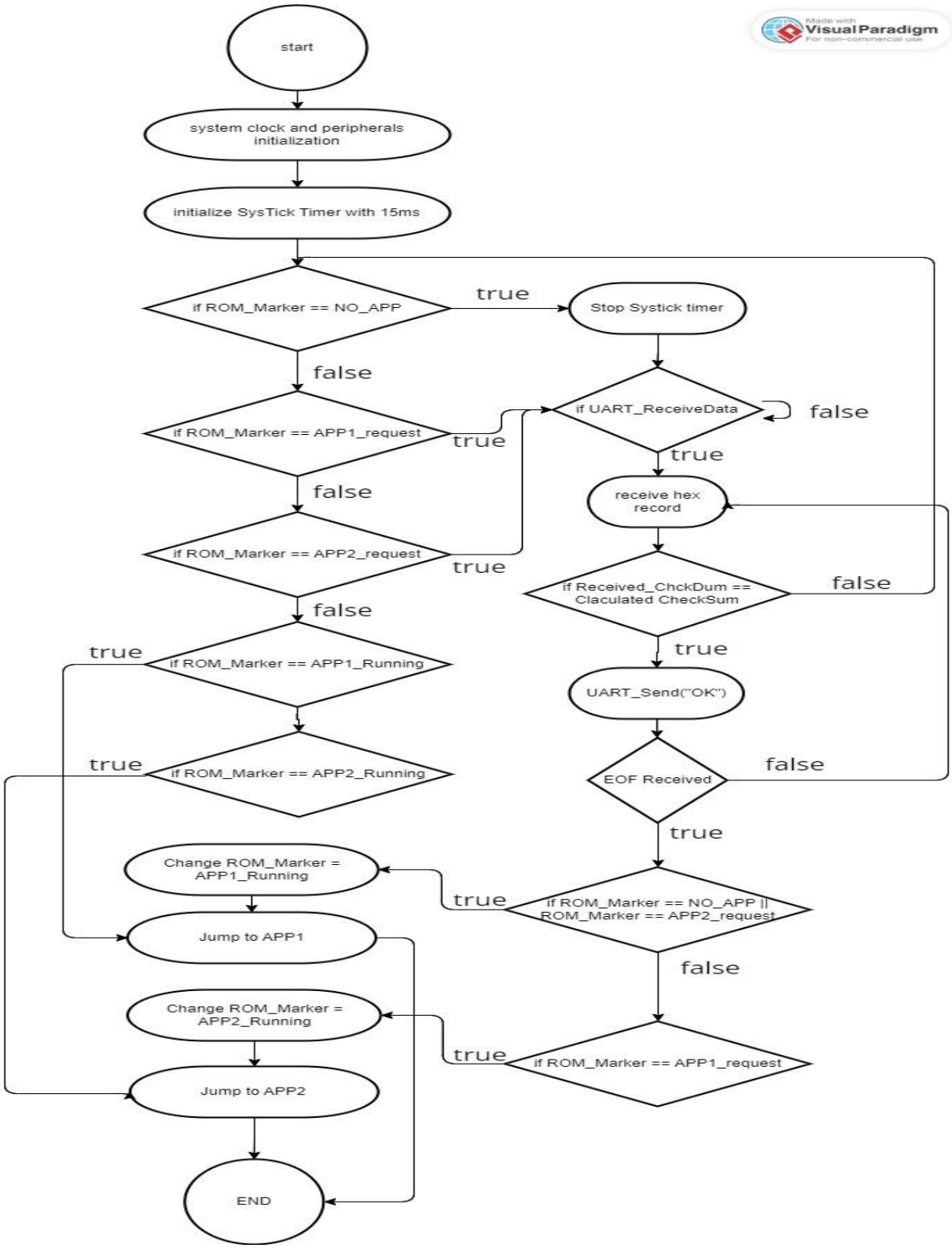


Figure 4- Bootloader Flow chart

RTOS App State Machine

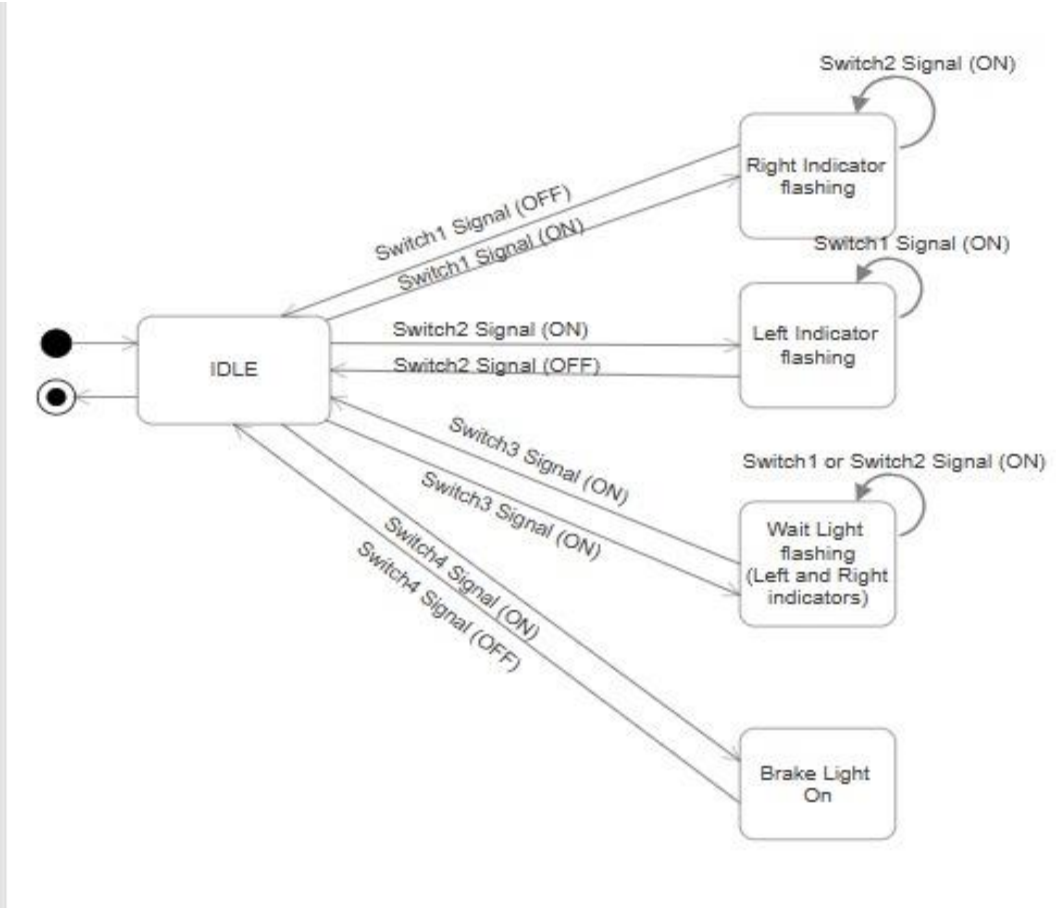


Figure 5 - RTOS App State Machine

- **Unit test plan :-**

Test all function as a white box

With two state (Boundary Analysis)

- Branch Converge
- Branch decision