Unified MCU Firmware Update System

- **Submitted By**

1. Abdelrahman Amin
2. Amr Khaled
3. El-Amir Galal
4. Fady Taha Bassiouni
5. Mohamed Hassan
6. Muhammad Reda

- **Supervised By**
  - **E.S Department Head of Information Technology Institute**
    - Eng. Youssef Nofal
    - Eng. Nour Hassan

- **Institute: Information Technology Institute (ITI)**

Date of Completion:
November 12, 2023

Acknowledgement

We wish to extend our profound gratitude to the Information Technology Institute (ITI) for affording us the invaluable opportunity to undertake this project. ITI's steadfast commitment to the advancement of technology education and innovation has greatly enriched our academic journey.

Our tenure at ITI has been nothing short of transformative, and it is with the utmost appreciation that we acknowledge the unwavering support and expert guidance received from the institute's esteemed faculty and administration. Their dedication to the dissemination of knowledge has equipped us with indispensable skills and a comprehensive understanding of a wide spectrum of cutting-edge technologies.

We reserve our deepest thanks for our project mentors and instructors at ITI, whose sage counsel, encouragement, and mentorship have played an indispensable role in the success of our project. Their wealth of experience and astute insights have significantly contributed to the project's excellence.

The tenure spent at ITI has not only furnished us with a robust academic foundation but has also cultivated an atmosphere of innovation, collaborative engagement, and perpetual learning. This has indeed been a remarkable educational voyage, and we are profoundly thankful for the experiences and knowledge we have amassed.

Our gratitude also extends to our fellow students at ITI, whose camaraderie, collective efforts, and shared passion for technology have been a source of inspiration throughout our academic pursuit.

This project stands as a testament to the opportunities and knowledge we have garnered during our tenure at ITI. We are profoundly grateful for the support and experiences that have shaped our educational odyssey.

[Team Members]

[eme-Embedded Systems Department]

[November 12, 2023]

1 Contents

2 Introduction

Firmware updates are incredibly important for our electronic devices. They play a key role in ensuring device security, efficiency, and functionality. While they may not be very noticeable to users, they are crucial for the reliability and resilience of our devices. In this documentation, we will explore firmware updates and their connection to advanced features like over-the-air updates, secure communication, and centralized management. Our focus will be on enhancing device security.

Firmware updates are often overlooked, but they are essential for keeping our electronic companions reliable and secure. We will highlight the importance of over-the-air updates, which make it easier to deliver critical software improvements while protecting against vulnerabilities. We will also discuss secure communication, revealing how encryption and authentication mechanisms safeguard our devices, protect data integrity, and maintain user trust. Additionally, we will explain centralized management and how it creates a unified system and strengthens firmware security across different devices.

To demonstrate the practical implementation of these principles, we have developed a small application. It showcases how these concepts have a real-world impact on device security and reliability.

Through this exploration of firmware updates and advanced features, you will gain a deep understanding of how they collectively affect the reliability, security, and capabilities of embedded systems. By the end, you will have a comprehensive understanding of how these seemingly small software elements propel our devices forward and protect them against emerging threats.

3 Background

Embedded systems have become integral to our daily lives, seamlessly operating behind the scenes in devices ranging from consumer electronics to industrial machinery. These systems rely on firmware, the software permanently stored in their hardware, to function as intended. While these devices have brought incredible convenience and efficiency to our lives, the landscape of technology is constantly evolving, necessitating regular updates to firmware to address security vulnerabilities, enhance functionality, and adapt to changing needs.

In response to these evolving demands, this documentation explores a comprehensive approach to firmware management, centered around firmware over-the-air (FOTA) updates, bootloader integration, wireless delivery, secure communication, selective microcontroller unit (MCU) reset and bootloader entry, and centralized firmware management. We delve into the nuts and bolts of these essential firmware update features, examining their significance in maintaining the health and performance of embedded systems.

➢ *Firmware Update Features:*

- FOTA Bootloader Integration: Firmware over-the-air updates have emerged as a vital method to deliver improvements and address vulnerabilities remotely. We examine the integration of FOTA capabilities into bootloader systems, enabling efficient and secure updates without the need for physical access to the device.

- Wireless Update Delivery: Wireless communication is at the heart of FOTA updates, enabling seamless delivery of new firmware versions. We explore the technologies and protocols that make wireless updates a reality, from Wi-Fi and Bluetooth to cellular networks.

- Secure Communication: The security of firmware updates is paramount in the connected world. We analyze the techniques and protocols that safeguard communication channels, ensuring the integrity and confidentiality of data during the update process.

- Selective MCU Reset and Bootloader Entry: Understanding how to selectively reset MCUs and enter bootloader mode is a crucial aspect of safe and effective firmware updates. We dive into the intricacies of this process, highlighting its importance in avoiding bricked devices.

- Centralized Firmware Management: Efficiently managing firmware across a fleet of devices is a challenging task. We explore centralized firmware management systems that streamline the distribution of updates, ensuring consistency and reliability across the entire network.

In addition to these fundamental firmware update features, we developed small applications to ensure the functionality of our service. These applications include:

- Temperature Regulation: Embedded systems equipped with temperature regulation can automatically adjust their operations based on environmental conditions. This is invaluable in scenarios like climate control, ensuring optimal performance and energy efficiency.

- Light Regulation: The integration of light regulation allows embedded systems to respond to changes in ambient lighting. This can be vital in applications such as smart lighting, where devices can automatically adjust brightness or color temperature.

- Distance Measurement: Distance measurement features enable embedded systems to gather data about the proximity of objects or people. This is useful in applications like robotics, where precise measurements are essential for safe and efficient operation.

In the ever-evolving landscape of embedded systems, mastering firmware management and embracing sensory integration is essential for both existing and emerging applications. This documentation offers an in-depth exploration of these critical components, providing readers with a comprehensive understanding of the technology that drives these systems and the potential they hold for the future.

4 Design

4.1 Overview

Our project revolves around a comprehensive software update system delivered via Over-the-Air (OTA) updates. The system is orchestrated through a centralized server that is intricately connected to a Master Microcontroller Unit (MCU). A Controller Area Network (CAN) network links all MCUs together, enabling the seamless reprogramming of slave MCUs.

This sophisticated system is accessed through a secure web portal. Once authenticated, users are granted the capability to initiate firmware updates on both the master and slave MCUs. The process of updating slave MCUs is executed through the utilization of CAN network technology and a dedicated Bootloader.

In addition to the core functionality, we have also developed a prototype application as proof of concept for our service. This application boasts the ability to collect and monitor data pertaining to environmental conditions, including temperature, humidity, light levels, and distance measurements.

4.2 Hardware Dependencies

- **Microcontrollers**: A network of microcontrollers (STM32F103C8T6) is employed, with each unit assigned specific roles, all microcontrollers are connected via a CAN network.
- **Web Server**: The centralized microcontroller unit (MCU) interfaces with a web server, allowing for remote firmware updates via a website.
- **CAN Transceivers**: four can transceivers PP230 for building the network.
- **EEPROM Memory Bank**: In cases where firmware updates fail, an EEPROM memory bank provides redundancy.

- **Microcontroller Roles:**
    - **First Node *(Temperature and Humidity Control):*** This microcontroller is responsible for regulating temperature and humidity levels within the environment. It utilizes a *DHT11 sensor* for data acquisition and control.
    - **Second Node *(Light Regulation):*** The second microcontroller manages ambient lighting by responding to changes using *a Light-Dependent Resistor (LDR)* for input and control.
    - **Third Node *(Distance Sensing and Car Alarm):*** This microcontroller monitors incoming vehicles by utilizing an *ultrasonic sensor*. It activates different alarms, each with a distinct sound, based on the vehicle's proximity. LEDs are also employed to provide visual alerts.
- **Main MCU and Communication:**

  The primary MCU serves as the central hub for the system. It connects to the web server via Wi-Fi, using an ESP8266 module for wireless communication. This integration allows for remote access and control of the entire embedded system.

  This well-organized system architecture provides a clear understanding of the project's structure and functionality, ensuring efficient communication between various components and enabling remote management of firmware updates.

4.3 Software Modules

4.3.1 User Interface (Website):

Welcome to our FOTA Front-End hub, where updates meet simplicity. Experience a streamlined interface, seamlessly managing Firmware Over The-Air updates.

This’s a simple website for the service we provide.

- **Technologies used in our Website:**

  - Nuxt(nuxt.com): Full stack framework for building websites with JavaScript.

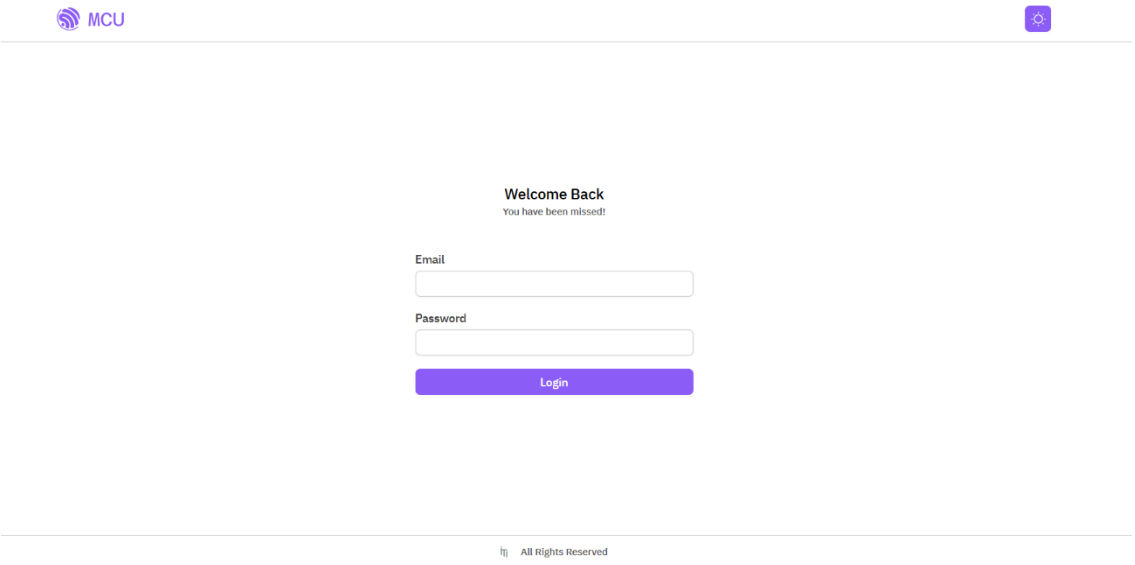
  - socket.io-client: For communicating (sending/receiving data) with the socket server.

  - Supabase: For handling users authentication (Login, Logout, Creating an account, etc..).

  - Nuxt UI: UI component library that provides ready to use components (Buttons, Inputs, Progress bars, etc..).

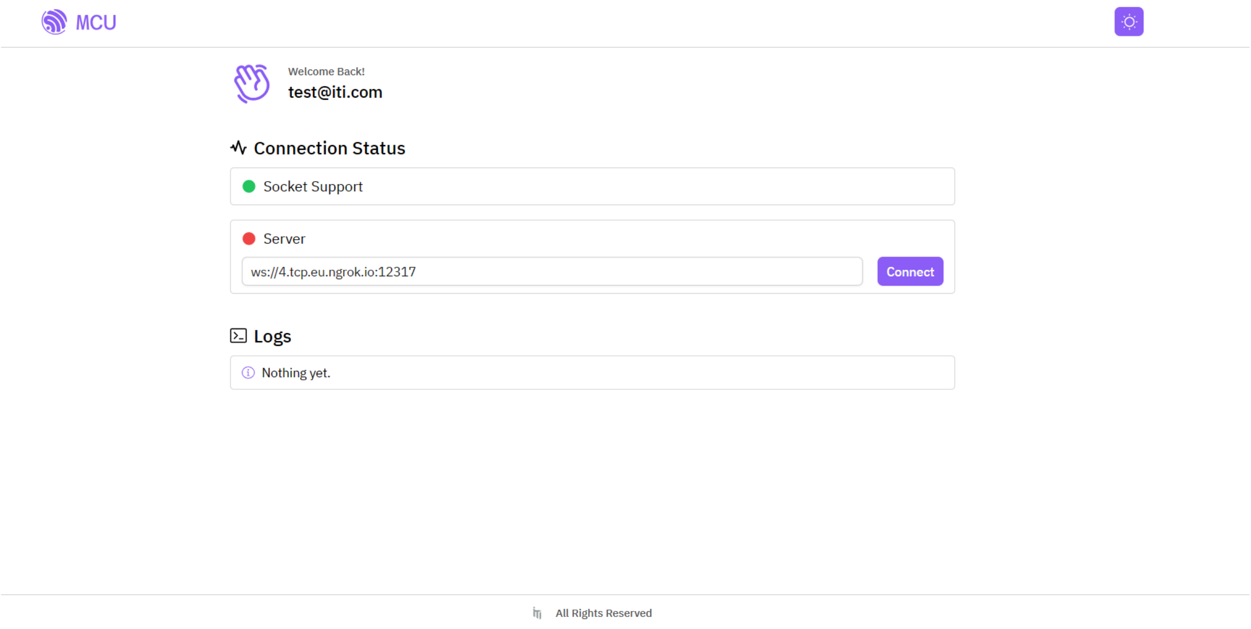
- **Login Page**

  Access to this page requires user authentication within our system. To log in and enter, you must be a registered user.



- ## Connect to MCU Page

Upon successful login, you will be automatically redirected to this page. Please be advised that connectivity to the Microcontroller Unit (MCU) and the internet is mandatory to establish and maintain a connection with the server.



Welcome Back!
test@iti.com

Connection Status

● Socket Support

● Server

ws://4.tcp.eu.ngrok.io:12317

Logs

Nothing yet.

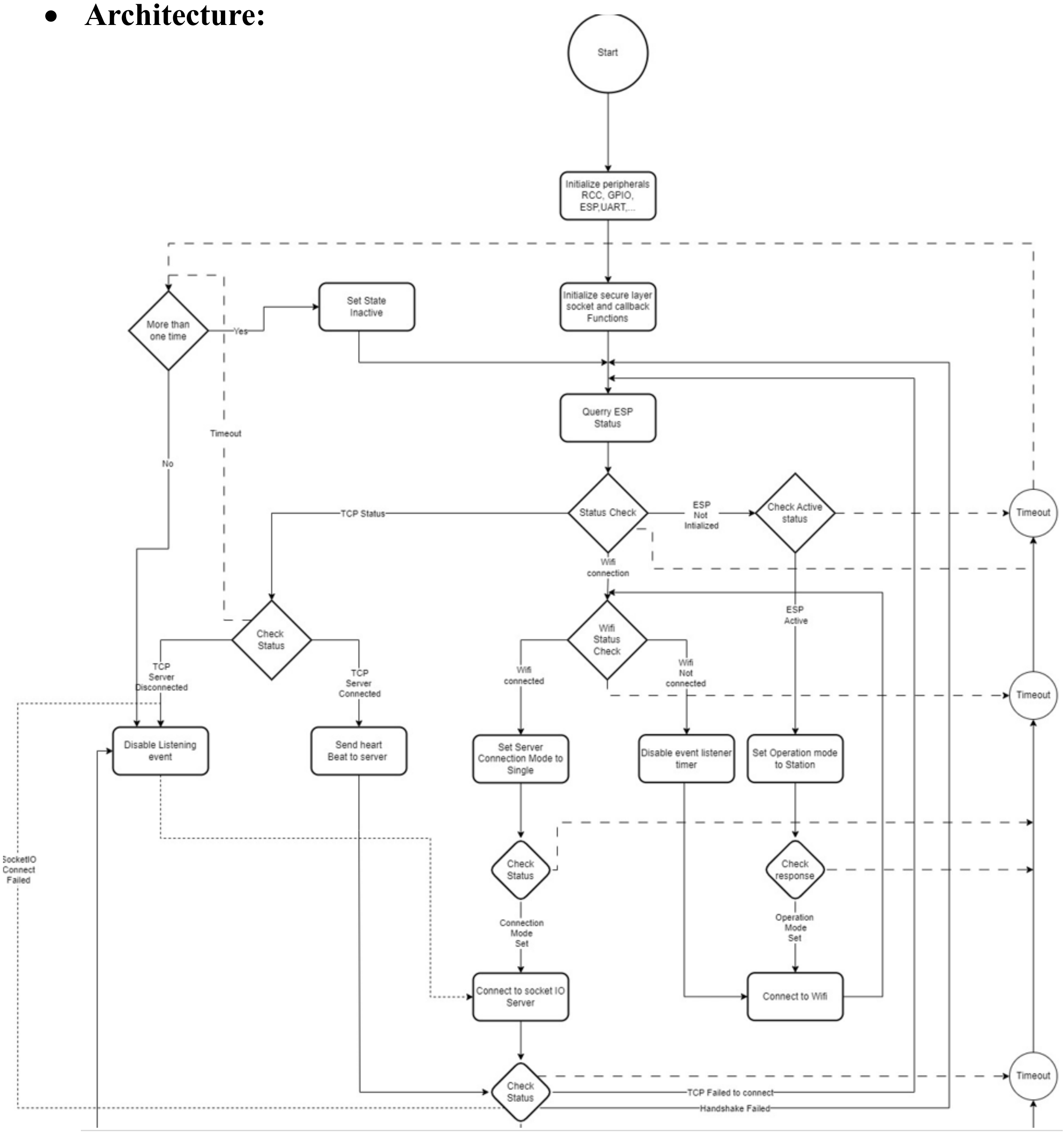
All Rights Reserved

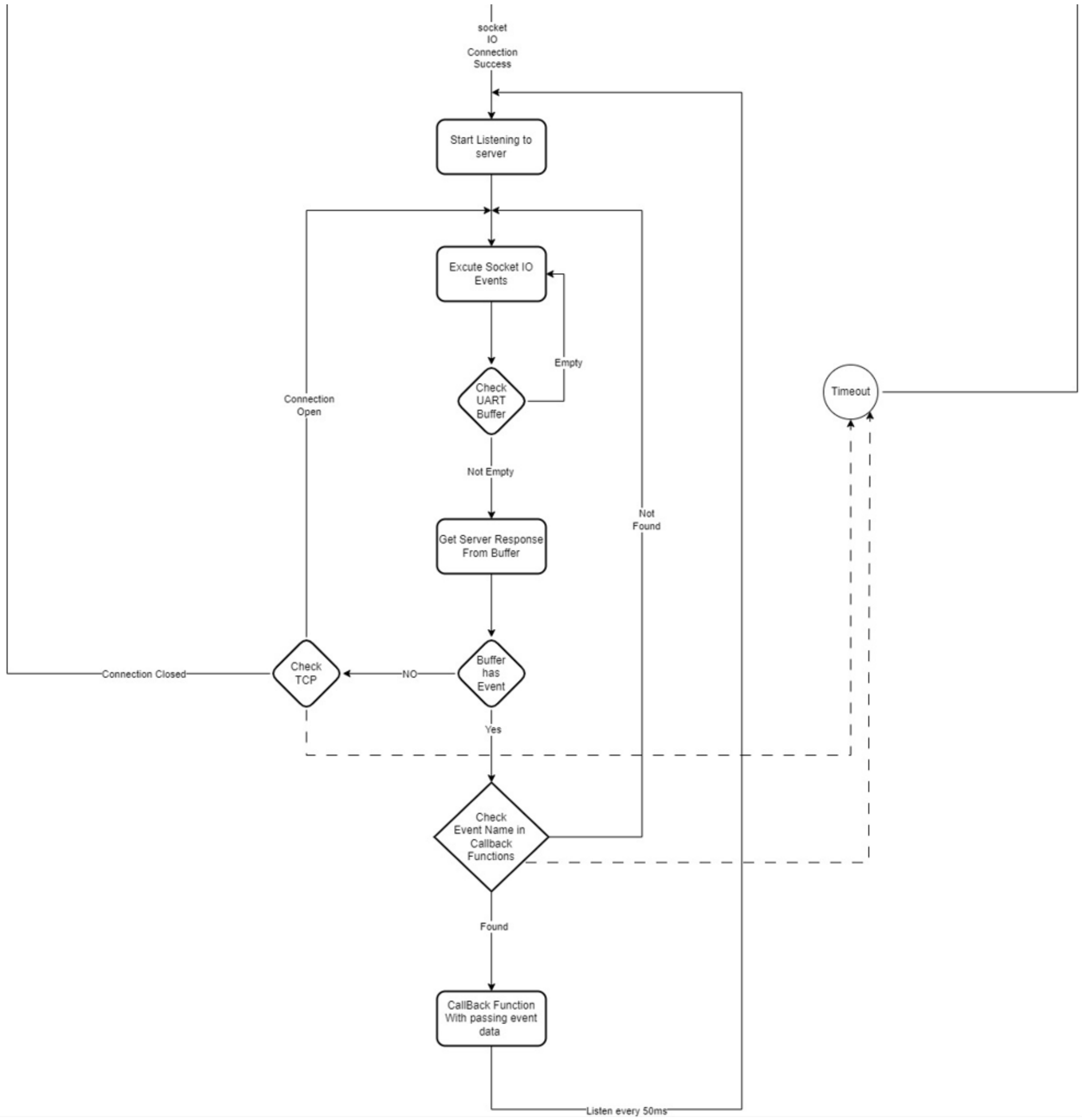
4.3.2 FOTA (Firmware Over-The-Air)

- **Brief**

  Firmware Over-The-Air (FOTA) is a technology that enables wireless updates and upgrades of embedded software on connected devices. It allows manufacturers to remotely deploy enhancements, patches, or new features to devices, eliminating the need for physical interventions. FOTA plays a crucial role in keeping devices secure, up-to-date, and adaptable to evolving technologies without inconveniencing users with manual updates.

- **Architecture:**




socket IO Connection Success
Start Listening to server
Excute Socket IO Events
Empty
Connection Open
Check UART Buffer
Timeout
Not Empty
Not Found
Get Server Response From Buffer
Connection Closed
Check TCP
NO
Buffer has Event
Yes
Check Event Name in Callback Functions
Found
CallBack Function With passing event data
Listen every 50ms

- **FOTA Components:**

- **Server side:**

  Server has Built by Flask Socket io server.

- **Client Side**

- **ESP8266:** A Wi-Fi module that facilitates communication by sending requests to the server.
- **STM32F103C8T6:** An ECU Node responsible for transmitting the Hex file over the CAN Bus to the targeted ECU.
- **CAN Bus Technology**: The technology responsible for efficiently transferring the Hex file to the targeted ECU.

- **Bootloader:** A mechanism responsible for the secure flashing of the Hex file onto the ECU Node.
- **Dual Bank:** In the event of a failed ECU update, this feature ensures the preservation of the last version, allowing for its retrieval and subsequent flashing onto the ECU.

- **Communication Protocols:**

  Server-Device Communication: communication protocol used for transmitting firmware updates is Socket io.

- **Update Process:**

  - **How the FOTA update process is triggered?**

1. Connection Initiation:

   Initiate a connection to the server by accessing the website.

2. Token Authentication:

   - Input the car's unique token on the website.
   - Transmit the token to the server.
   - Validate the token's correctness at the MASTER ECU Node.

3. Connection Establishment:

   - If the token is correct, a "Connection Accepted" message is sent.
   - If the token is incorrect, a "Bad Token" message is sent.

4. ECU Selection:

   - Select the target ECU on the website.

5. Connection Request:

   - Instruct the server to connect to the specified ECU by sending the ECU ID from the website.

6. ECU Validation:

   - The Master ECU validates the specified ECU.

7. Transition to Bootloader Mode:

- Send a reset signal through external interrupt to the specified ECU via the CAN BUS.
- The Target ECU transitions into Bootloader Mode and responds with an 'OK' via the CAN BUS.

These steps collectively describe how the user triggers the update process by initiating a connection, authenticating the vehicle token, selecting the target ECU, and requesting a connection to the specified ECU. The update process unfolds from there, involving validation, signaling, and transitioning to Bootloader Mode.

- **Installation process steps:**

1. Access the website and establish a connection with the server.
2. Input the vehicle's token, a unique identifier assigned to each car.
3. Transmit the token from the website to the server.
4. Select the target ECU on the website.
5. Instruct the server to connect to the specified ECU by sending the ECU ID from the website.
6. Allow the user to upload a Hex file by clicking "Browse Files."
7. Verify the chosen HEX File.
8. Encrypt the HEX file on the website.
9. Transmit the encrypted HEX file to the server by selecting the "Flash" button.

- **Error Handling and Logging:**
  - **Error Codes:**
    - **ESP Not Initialized:**
      Verification of ESP activation status to ensure proper initialization.
    - **Wi-Fi Not Connected:**
      Automated reattempt to establish a connection to the Wi-Fi network.
    - **TCP Connection Failure:**
      In the event of TCP connection establishment failure, a requery of ESP status is triggered based on the provided diagram.
    - **Handshake Failure:**
      Recognition of a failed handshake between the socket io server and ESP, prompting a requery of ESP status as per the diagram.

  - **TCP Server Disconnection:**
    Disabling server listening and initiating a retry to connect to the socket io server based on the diagram.
  - **Socket io Connection Failure:**
    Identification of ESP's failure to connect to the server, managed by disabling server listening and initiating a retry to connect to the socket io server following the diagram.
  - **Buffer Event Not Found:**
    Detection of a lack of events in the UART Buffer, prompting a check on the TCP connection status.
  - **Connection Closure:**
    Recognition of a closed session, leading to the disabling of server listening and a subsequent retry to connect to the Socket io server.
  - **Timeout:**
    Examination of timeout occurrences; if more than one, the system sets the state to inactive and initiates a requery of ESP status.
- **Logging:**
  Any event happens between the server and ECU is sent to server like (Starting flash, Flash done successfully, ECU disconnected, … etc).

- **Security:**
  - Encryption:

    We used Fernet protocol to encrypt data between website and server.

    Here's a small brief about it:

    Fernet is a symmetric key cryptographic algorithm and protocol designed for secure communication and data integrity. It falls under the category of symmetric-key authenticated cryptography, ensuring both confidentiality and integrity of data. Fernet utilizes symmetric key encryption, meaning the same key is used for both encryption and decryption.

    Key features and characteristics of Fernet include:

- Symmetric Encryption: Fernet employs a symmetric encryption algorithm, where a single, shared secret key is used for both encrypting and decrypting data. This key must be kept confidential and secure.

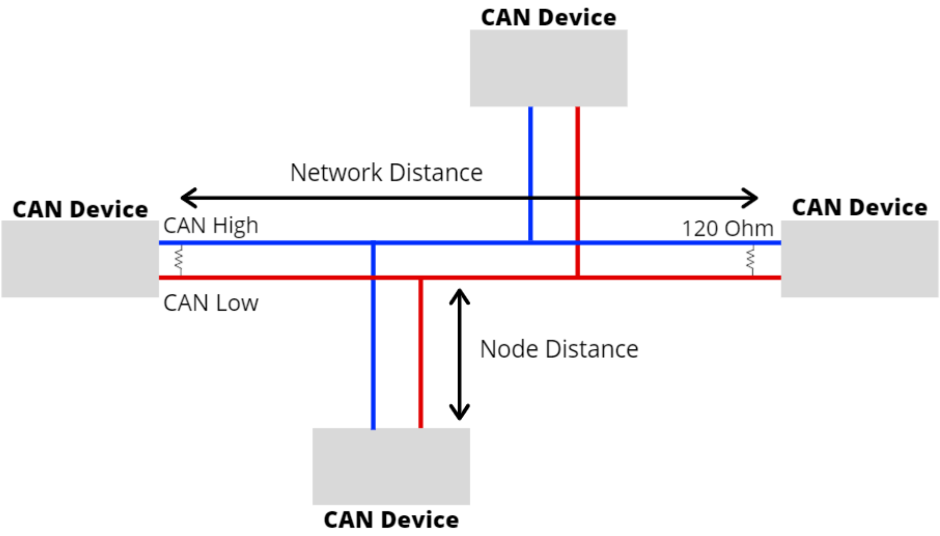
- Authenticated Cryptography: Fernet ensures data integrity by using an authentication code that is generated along with the ciphertext. This code allows the recipient to verify the authenticity of the received data.
- Timestamps for Freshness: Each Fernet token contains a timestamp, preventing the replay of old tokens. This feature enhances security by ensuring that tokens have a limited lifespan.
- URL-Safe Encoding: Fernet tokens are URL-safe, meaning they can be easily included in URLs without the need for additional encoding.
- Ease of Use: Fernet is designed for simplicity and ease of use, making it a popular choice for implementing secure communication protocols. The simplicity of the protocol reduces the risk of implementation errors.
- Python Implementation: Fernet has a widely used Python implementation in the cryptography library, making it accessible for developers working in Python-based projects.

4.3.3 CAN Network:

- **Introduction:**

The Controller Area Network (CAN) is a widely used communication protocol designed for real-time, high-integrity data exchange between electronic control units (ECUs). In the context of STM32F103C8T6 microcontroller, which integrates CAN capabilities, this Section provides an overview of a typical CAN bus network without the need for an external CAN controller.

Components of a CAN Bus Network:



- *STM32F103C8T6 Microcontroller:*

  This STM32 microcontroller serves as the primary node in the CAN bus network. It contains an integrated CAN controller, allowing it to communicate on the CAN bus directly.

- *CAN Transceivers:*

  While traditional CAN networks require external CAN controllers, the STM32F103C8T6 integrates a CAN controller, reducing the need for external components. However, a CAN transceiver is still necessary to interface with the physical CAN bus.

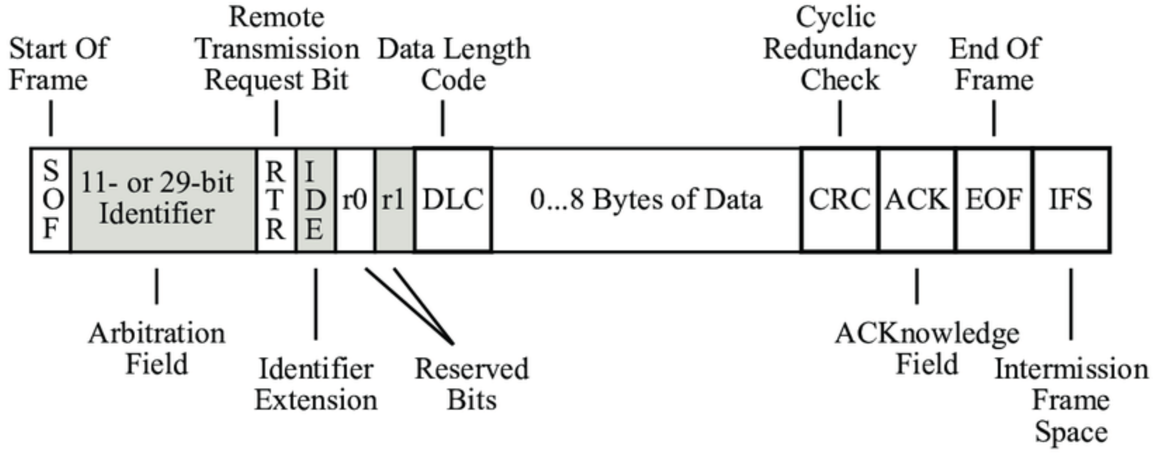
- *CAN Bus:*

  The physical medium for CAN communication. It is typically a twisted-pair cable.

- *CAN Bus Topology:*

  The CAN bus network employs a linear bus topology. Nodes, including the STM32F103C8T6, are connected in series along a single CAN bus cable.

- **CAN Frame Overview:**



In the CAN protocol, communication between nodes occurs through the exchange of frames. A frame is the fundamental unit of data transmission and encapsulates the information to be sent between CAN nodes.

The CAN frame structure consists of several key components:

Start of Frame (SOF): The Start of Frame is the initial bit that marks the beginning of a frame. It serves as a synchronization point for all nodes on the network.

Identifier (ID): The Identifier is a unique value assigned to the frame, determining its priority and facilitating message filtering. In standard CAN, this field can be 11 bits (CAN 2.0A) or 29 bits (CAN 2.0B) long.

Control and Data Fields: The Control Field contains bits that specify frame properties, including the length of the Data Field. The Data Field carries the actual payload of the frame, ranging from 0 to 8 bytes in standard CAN.

CRC (Cyclic Redundancy Check): The CRC field contains a checksum calculated from the frame's contents. It allows nodes to detect errors during transmission.

ACK (Acknowledgment): The ACK field indicates whether the frame was successfully received. It consists of an ACK slot and ACK delimiter.

End of Frame (EOF): The End of Frame signals the conclusion of the frame transmission.

Frame Types:

- **Data Frame:** The most common type of frame, carrying data from the sender to one or more receivers.
- **Remote Frame:** A frame requesting data from another node. It contains only the identifier and the RTR (Remote Transmission Request) bit.
- **Frame Transmission:** Bit Stuffing: To maintain synchronization, a technique called bit stuffing is employed.
  Extra bits are inserted into the data stream to ensure a sufficient number of transitions.

Arbitration: Nodes on the network use the identifier for arbitration. Lower identifiers take precedence, allowing higher-priority messages to interrupt lower-priority ones.

Error Handling: CAN frames include mechanisms for error detection and handling, such as CRC checks and acknowledgment.

CAN BUS Initialization:

The CAN interface is initialized by a dedicated function, preparing the CAN pins A11 and A12 for communication.
This process involves exiting sleep mode, setting the initialization mode, and configuring parameters like Auto Retransmission, Bus-Off Management, Wake-Up Mode, Time-Triggered Communication, Receive FIFO Locking, and Transmit FIFO Priority.
The operational mode is Normal.
The Baud Rate is fixed at 500KBPS.

Transmitting CAN Frame Function

1. **Selecting an Empty Transmit Mailbox:** The function begins by checking the Transmit Status Register (TSR) to identify an available Transmit Mailbox among TME0, TME1, or TME2.
2. **Configuring CAN Header:** After determining an empty mailbox, the function configures the CAN header by setting up the Standard or Extended Identifier, Remote Transmission Request (RTR), and Data Length Code (DLC) in the selected Transmit Mailbox.

3. **Configuring Transmit Global Time (Optional):** If Transmit Global Time is enabled, the function sets the corresponding bit in the Transmit Data Register (TDTR).

4. **Configuring Data Fields:** The data from the provided buffer is written into the Transmit Data Registers (TDHR and TDLR) of the selected Transmit Mailbox.

5. **Requesting Transmission:** To initiate the transmission, the function sets the Transmission Request bit in the Transmit Information Register (TIR). Usage Example Users can leverage this functionality by providing the necessary parameters when calling the CAN_TX function.

- **example:** A CAN_TxHeaderTypeDef structure (txHeader) is populated with details such as Standard Identifier, IDE, RTR, DLC, and Transmit Global Time. A data buffer (dataBuffer) contains the information to be transmitted. The CAN_voidAddTxMsg function is invoked with the populated header and data buffer.

CAN Receive Message Function Overview

- **Get Identifier Extension (IDE):** Determines whether the received message has a Standard or Extended Identifier by examining the Identifier Extension bit (IDE) in the Receive Information Register (RIR).
- **Get Identifier (ID):** If the message has a Standard Identifier, retrieves the Standard Identifier from the Receive Information Register (RIR). If it has an Extended Identifier, retrieves the Extended Identifier from the RIR. Get Remote Transmission
- **Request (RTR):** Retrieves the Remote Transmission Request bit (RTR) from the RIR. Get Data Length Code (DLC): Retrieves the Data Length Code (DLC) from the Receive Data Transfer Register (RDTR).
- **Get Filter Match Index (FMI):** Retrieves the Filter Match Index (FMI) from the RDTR. Get Timestamp: Retrieves the Timestamp from the RDTR.
- **Get Data:** Retrieves the data payload, parsing it from the Receive Data Low and High Registers (RDLR and RDHR).
- **Release FIFO:** Releases the corresponding Receive FIFO (RX FIFO 0 or RX FIFO 1) after processing the received message.
- **Return Flag:** Optionally, sets a return flag if a predefined character (e.g., '*') is found in the received data. This can indicate a specific condition in the data payload.

Note on Buffer Clearing:

The function clears the data buffer at the beginning of each call, ensuring proper data storage.

Configuring CAN Filters

1. **Filter Configuration Structure:** Create a filter configuration structure to hold the parameters necessary for filter initialization.
2. **Filter Activation:** Enable or disable the filter based on the application's requirements.
3. **Filter Bank Selection:** Choose the filter bank to be initialized. Multiple filter banks are often available for use.
4. **Filter FIFO Assignment:** Assign the filter to a specific FIFO (First In, First Out) buffer for handling incoming messages.
5. **Filter Identifier and Mask Configuration:** Define the filter criteria by specifying the filter identifier (ID) and mask values. The filter compares the incoming message ID with the filter ID, considering the mask for filtering.
6. **Filter Mode Configuration:** Select the filter mode to determine how the filter compares the incoming message ID. Common modes include ID list mode and ID mask mode.
7. **Filter Scale Configuration:** Choose the filter scale, indicating the size of the filter ID and mask. Common options include 16-bit and 32-bit scales.
8. **Filter Configuration Execution:** Execute the filter configuration by calling the relevant function with the initialized filter configuration structure.

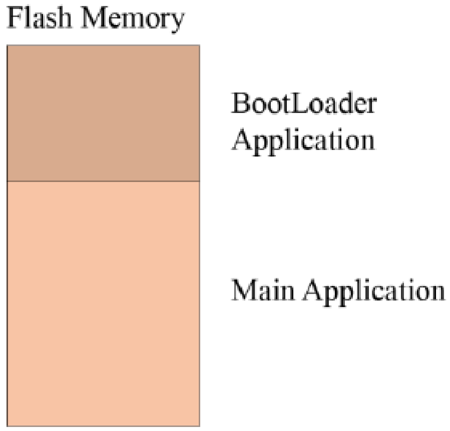
Considerations and Recommendations: Filters should be configured during the initialization phase of the CAN controller. Adjust filter parameters based on the desired filtering criteria for the specific application. Filters are instrumental in optimizing the reception of relevant messages while ignoring irrelevant ones.

4.3.4 Bootloader:

The flashing procedure involves updating the contents of the flash memory. We will employ the bootloader technique to flash our application code onto the flash memories of our Electronic Control Units (ECUs). A bootloader is a compact application specifically created for either downloading firmware into an embedded device or updating our application. It is housed in the Read-Only Memory (ROM) or the flash memory of the Microcontroller Unit (MCU).

The computer transmits the executable file through its USB interface, while the processor can receive it through an alternative communication method. We opt to receive it via the CAN protocol. Once the processor reads data from the CAN interface, it subsequently transfers this information to the Flash Programming/ Erasing Controller (FPEC) registers in the flash interface peripheral, thereby flashing it into the entry point of the flash memory.

Memory architecture

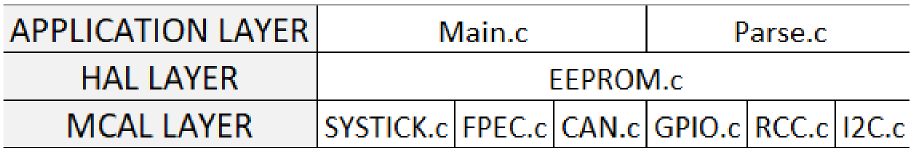


We used STM32F103C8T6 as our ECUs to run the application with 128 Kbytes flash memory.
Our flash memory will be divided into Bootloader section and Main application section. The bootloader will occupy 16kbytes starting from 0x08000000
While the main application will occupy the rest of flash memory
Following power-up or reset, the bootloader takes precedence as the initial code to execute, preceding any other software in the processor's startup sequence.
Bootloaders play a crucial role in programming and flashing a new application to its specified location in the flash memory, subsequently initiating its execution.

Bootloader requirements

- Ability to select the operating mode (Application or bootloader).
- Communication requirements (USB, CAN, I2C, USART, etc).
- Record parsing requirement (S-Record, hex, intel, etc).
- Flash system requirement (erase, write, read, location).
- EEPROM requirement (partition, erase, read, write).
- Application checksum (verifying the app is not corrupt).

Bootloader architecture



APPLICATION LAYER	Main.c			Parse.c		
HAL LAYER	EEPROM.c					
MCAL LAYER	SYSTICK.c	FPEC.c	CAN.c	GPIO.c	RCC.c	I2C.c

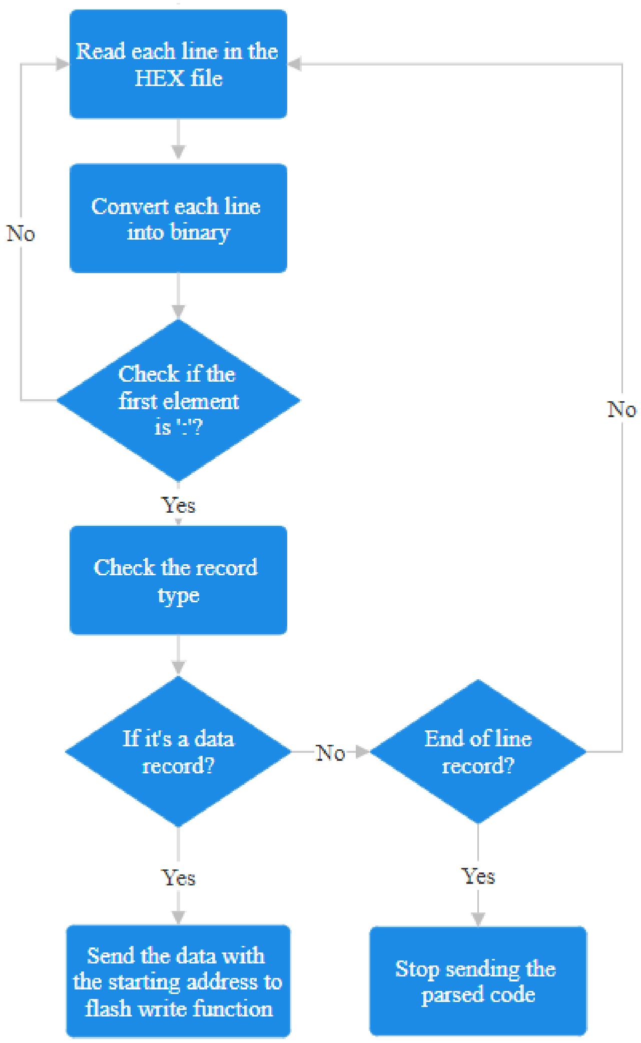
Hex Parser Driver

An assembler or C compiler could just output binary data, which is what a microcontroller needs, and store it as a binary file. If you set your compiler to output a .bin file it will appear as garbage in a text editor, so an Intel Hex file stores data as ASCII characters, which can be read by an editor. Hex is commonly used for programming microcontrollers, EEPROMs, and other types of programmable logic devices. In a typical application, a compiler or assembler converts a program's source code to machine code and outputs it into a HEX file. Thus, we need to write a driver which can parse the HEX file and convert it into ASCII code to download it easily on the microcontroller.

We need to do the following steps:

1. Take the HEX file and then check each record.
2. Check the validation of each record by reading the first element in the array to be ':'.
3. Start to convert each line which is written in HEX format and convert it into an Array and save them as a binary code.
4. Read the next byte that detects the number of data bytes implemented in this line.
5. Read and store the next two bytes which are stored in the array. These two bytes detect the starting address of this data.
6. Check the record type of the HEX file for each line. The record may be a data record or an End of file record.
7. At the end of each line, it will be a byte which represents a checksum which is an error check on the line
8. Start to calculate the checksum of the received data and check if it matches the received checksum then this record was sent correctly.
9. Flash the record in the Flash memory using FPEC driver.

Main Parser Algorithm:


Read each line in the HEX file
Convert each line into binary
No
Check if the first element is ':'?
Yes
Check the record type
If it's a data record?
No
End of line record?
No
Yes
Yes
Send the data with the starting address to flash write function
Stop sending the parsed code

Flashing program and Erase Controller (FPEC):

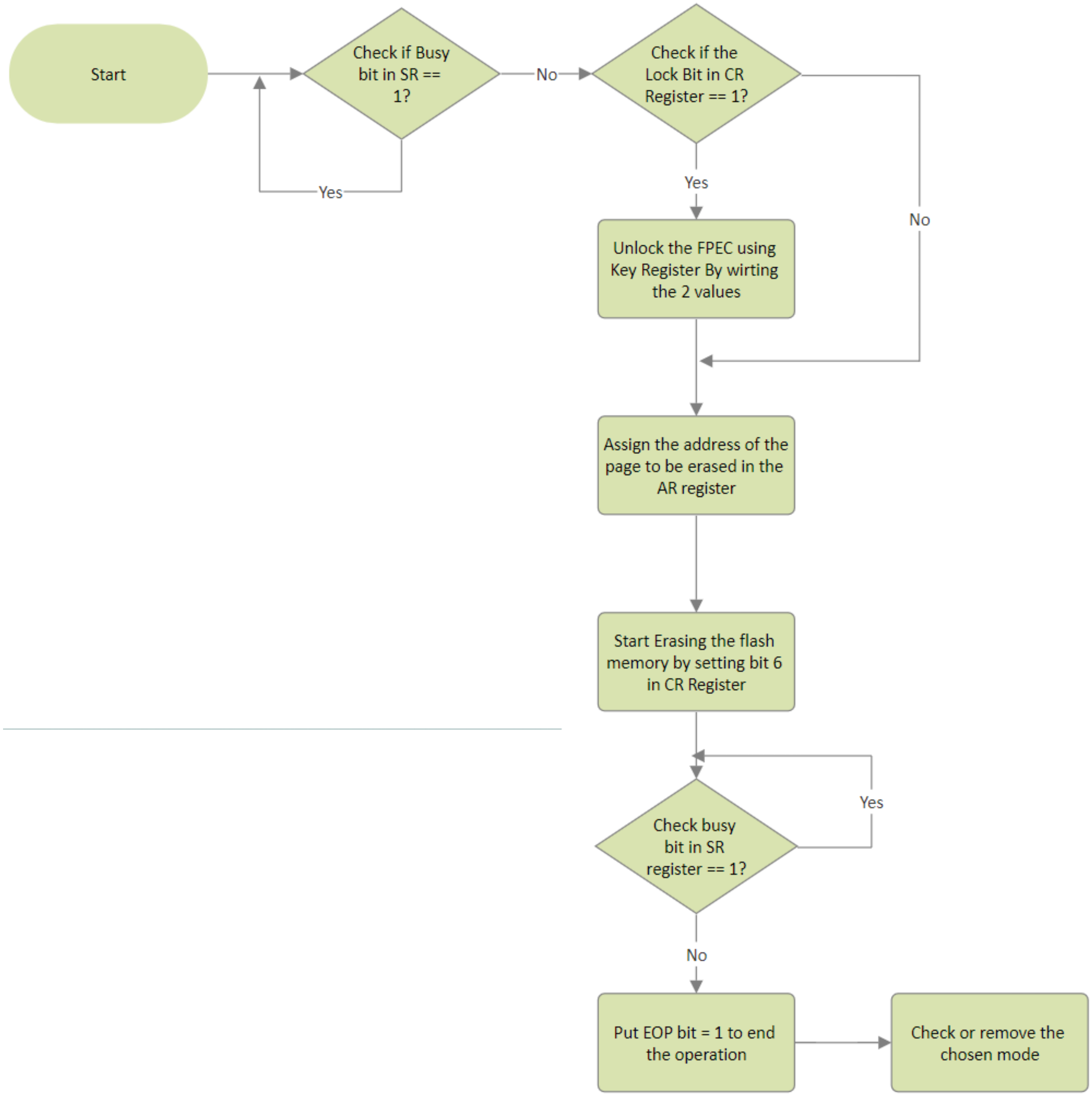
The flashing operation requires an erasing operation first.

In our application, we use a page erase technique. And we write a half word.

Erasing sequence

1. Check Flash Status: Continuously check the Flash Status bit to ensure that there is no ongoing flash operation.
2. Unlock Flash (Optional): Check if the Flash is already unlocked. If it is locked, unlock it by writing specific keys to the KEYR register.
3. Enable Flash Page Erase: Set the Page Erase bit in the Flash control register (CR).
4. Set Flash Page Address: Calculate the flash page address based on the given page number. Write the page address to the Flash Address Register (AR).
5. Initiate Flash Page Erase: Set the Start bit in the Flash control register to initiate the flash page erase operation.
6. Wait for Operation to Complete: Continuously check the Flash Status bit to wait for the erase operation to finish. If it is set, wait until it becomes clear.
7. Mark Operation as Complete: Set the End of Operation bit in the Flash Status Register (SR).
8. Disable Flash Page Erase: Clear the Page Erase bit in the Flash control register.

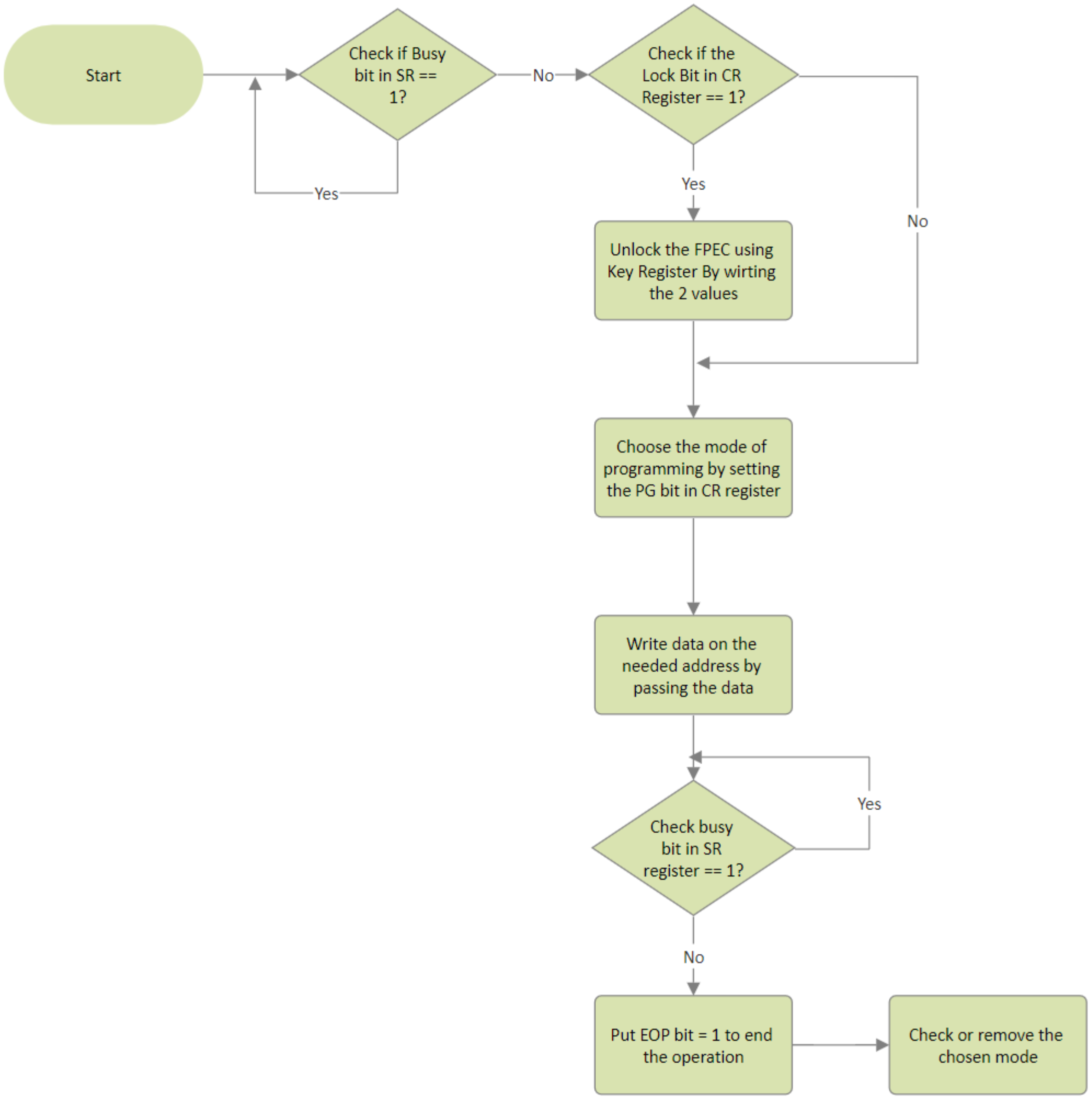
Erasing Algorithm:


Start
Check if Busy bit in SR == 1?
Yes
No
Check if the Lock Bit in CR Register == 1?
Yes
No
Unlock the FPEC using Key Register By wirting the 2 values
Assign the address of the page to be erased in the AR register
Start Erasing the flash memory by setting bit 6 in CR Register
Check busy bit in SR register == 1?
Yes
No
Put EOP bit = 1 to end the operation
Check or remove the chosen mode

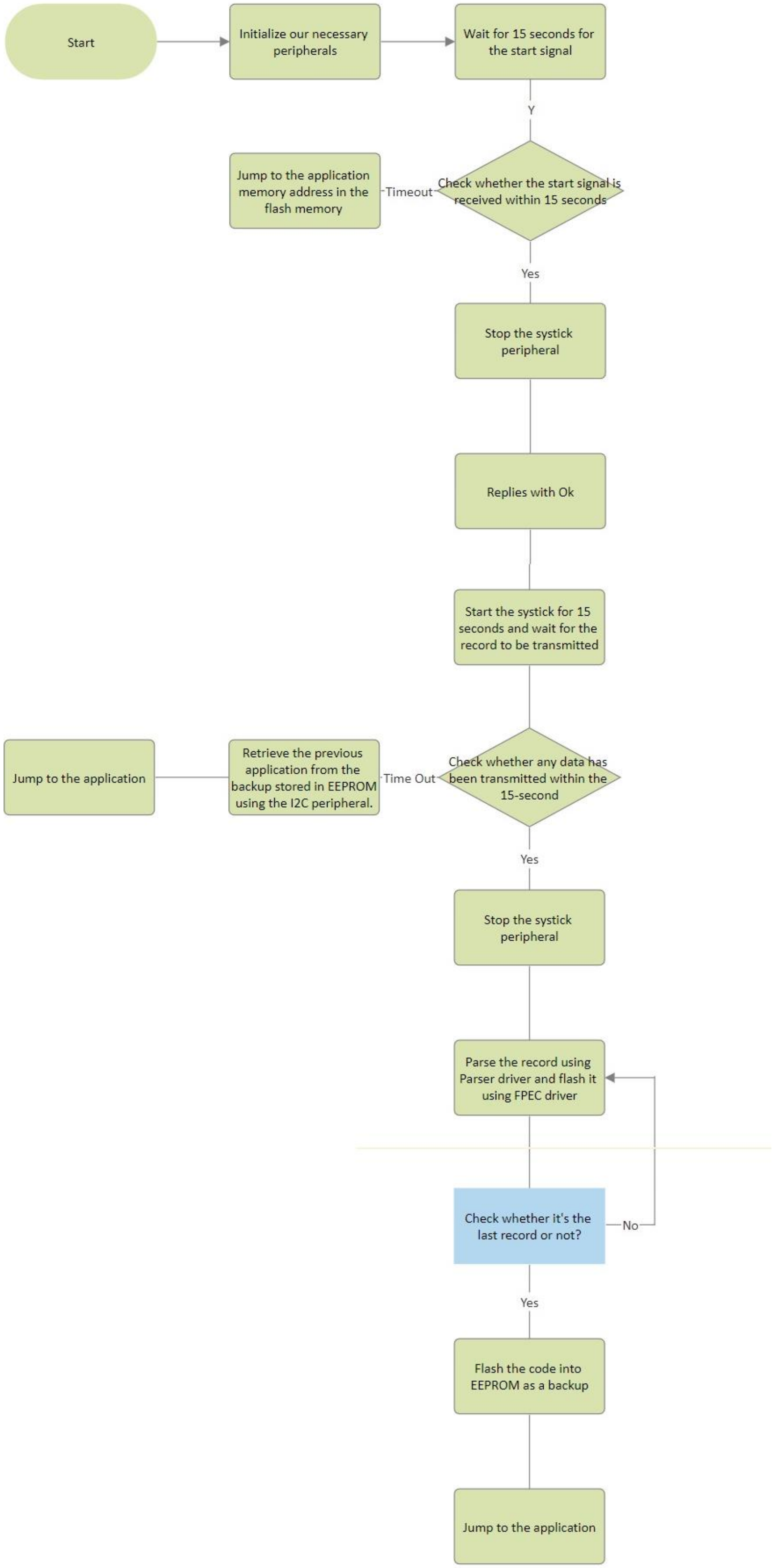
Flashing sequence

1. **Initialize the System:** Set up the system, including configuring the necessary clocks and peripherals, to enable the flash programming process.
2. **Unlock Flash Access:** Before writing to or erasing the flash memory, the FPEC needs to be unlocked. This involves a sequence of commands to specific registers, allowing write and erase operations using a sequence to Unlock:
   1- Key Register = Key1 = 0x45670123
   2- Key Register = Key2 = 0xCDEF89AB
3. **Erase Flash Pages:** erase the flash pages where the new firmware or data will be written.
4. **Program Flash Memory:** Write the new data or firmware into the flash memory. This involves copying the desired content into specific memory addresses.
5. **Verify the Programming:** After writing, it's common to perform a verification step to ensure that the data programmed into the flash memory matches the intended content.
6. **Lock Flash Access:** Once the programming and verification are successful, lock the flash memory to prevent unintended writes or erases.
7. **Run the Application:** After flashing, the processor can jump to the entry point of the newly programmed application in the flash memory.

Flashing Algorithm


Start
Check if Busy bit in SR == 1?
Yes
No
Check if the Lock Bit in CR Register == 1?
Yes
No
Unlock the FPEC using Key Register By wirting the 2 values
Choose the mode of programming by setting the PG bit in CR register
Write data on the needed address by passing the data
Check busy bit in SR register == 1?
Yes
No
Put EOP bit = 1 to end the operation
Check or remove the chosen mode

Thus, our bootloader algorithm will be:


Start
Initialize our necessary peripherals
Wait for 15 seconds for the start signal
Y
Check whether the start signal is received within 15 seconds
Timeout
Jump to the application memory address in the flash memory
Yes
Stop the systick peripheral
Replies with Ok
Start the systick for 15 seconds and wait for the record to be transmitted
Check whether any data has been transmitted within the 15-second
Time Out
Retrieve the previous application from the backup stored in EEPROM using the I2C peripheral.
Jump to the application
Yes
Stop the systick peripheral
Parse the record using Parser driver and flash it using FPEC driver
Check whether it's the last record or not?
No
Yes
Flash the code into EEPROM as a backup
Jump to the application

4.3.5 Proof of Concept Application:

- **Light Regulation**

  - **Purpose:**

    The Light Regulation Application utilizes the STM32F103C8T6 microcontroller along with a Light Dependent Resistor (LDR), Timer1 in PWM mode, and a 12-bit resolution Analog-to-Digital Converter (ADC) to regulate the intensity of light.

  - **Key Components:**

  - *STM32F103C8T6 Microcontroller*

    The STM32F103C8T6 serves as the brain of the system, managing and controlling the overall functionality.

  - *Light Dependent Resistor (LDR)*

    The LDR is used to sense the ambient light conditions, providing analog input to the system.

  - *Timer1 in PWM Mode*

    Timer1 is configured to operate in PWM mode, allowing precise control over the duty cycle for light regulation.

  - *ADC with 12-bit Resolution*

    The 12-bit resolution ADC converts the analog signal from the LDR to a digital value for processing.

- **Design:**

Initialize MCU Peripherals:

This step involves initializing the STM32F103C8T6 microcontroller and its peripherals, including Timer1 and ADC.

Configure Timer1 for PWM Mode:

Set up Timer1 to operate in PWM mode, which will be used to control the brightness of the light.

Configure ADC for 12-bit Resolution:

Configure the Analog-to-Digital Converter (ADC) to have a 12-bit resolution for accurate light measurements.

Initialize LDR GPIO Settings:

Initialize the GPIO settings for the Light Dependent Resistor (LDR) to interface with the microcontroller.

Main Loop:

Enter the main loop where the application continuously operates.

Read ADC Value (Light):

Read the light intensity by converting the analog signal from the LDR using the ADC.

Adjust PWM Duty Cycle based on Light:

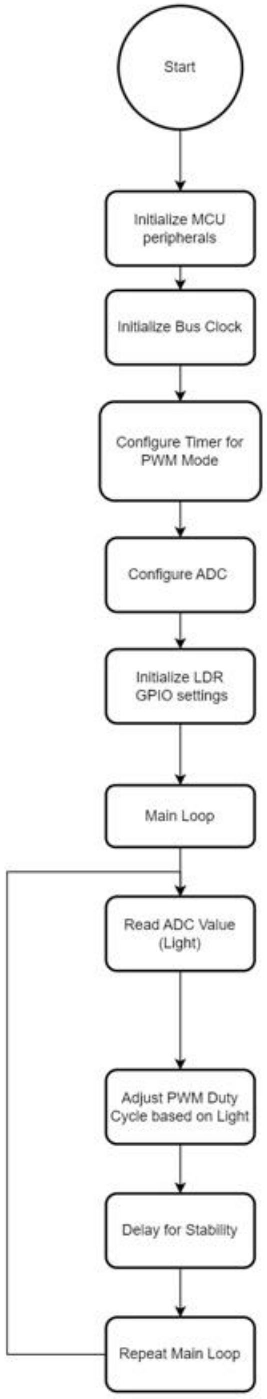
Adjust the PWM duty cycle of Timer1 based on the light intensity read from the ADC.

Delay for Stability:

Introduce a delay to ensure stability in light regulation.

Repeat Main Loop:

Repeat the main loop to continuously regulate the light based on the changing conditions.


Start
Initialize MCU peripherals
Initialize Bus Clock
Configure Timer for PWM Mode
Configure ADC
Initialize LDR GPIO settings
Main Loop
Read ADC Value (Light)
Adjust PWM Duty Cycle based on Light
Delay for Stability
Repeat Main Loop

- **Temperature and humidity control application:**

  - **Purpose:**
    This Section outlines the steps and considerations for interfacing a DHT11 sensor with an STM32F103C8T6 microcontroller to acquire temperature and humidity data.

    1. Initialization:

    1.1 Initialize System:

    Configure GPIO, timers, and any other necessary peripherals on the STM32F103C8T6 microcontroller.

    1.2 Initialize DHT11 Communication:

    Set up communication parameters for interfacing with the DHT11 sensor.

    2. Sensor Communication:

    2.1 Send Start Signal:

    Initiate communication by sending a start signal to the DHT11 sensor.

    2.2 Wait for Response:

    Wait for the DHT11 sensor's response, which involves pulling the data line high.

    2.3 Receive Response Signal:

    Capture the response signal sent by the DHT11 sensor.

    3. Data Transmission:

    3.1 Check Data Validity:

    Verify the validity of the received data using checksum or other error-checking mechanisms.

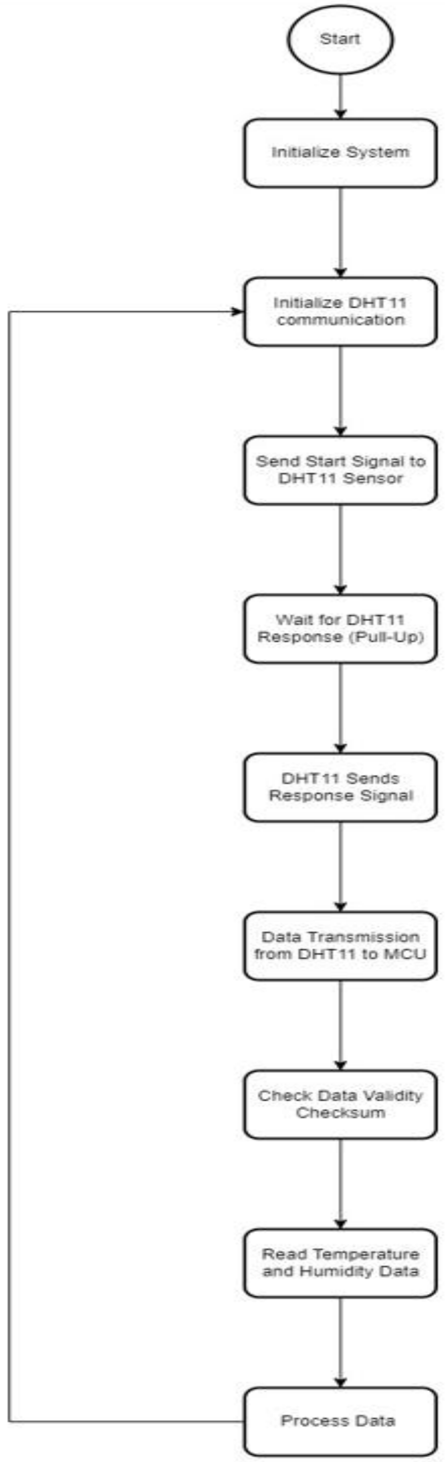
    3.2 Read Temperature and Humidity:

    Extract temperature and humidity data from the received bitstream.

    4.Process Data:

    Perform any necessary processing on the acquired data, such as displaying it, logging it, or triggering specific actions based on the values.

5. Design:


Start
Initialize System
Initialize DHT11 communication
Send Start Signal to DHT11 Sensor
Wait for DHT11 Response (Pull-Up)
DHT11 Sends Response Signal
Data Transmission from DHT11 to MCU
Check Data Validity Checksum
Read Temperature and Humidity Data
Process Data

- **Distance Measurement:**

Purpose:

This documentation outlines the steps and considerations for interfacing an ultrasonic sensor with an STM32F103C8T6 microcontroller to measure distance.

1. Initialization:

1.1 Initialize System:

Configure GPIO, timers, and other necessary peripherals on the STM32F103C8T6 microcontroller.

1.2 Initialize Ultrasonic Sensor Interface:

Set up communication parameters and GPIO pins for interfacing with the ultrasonic sensor.

2. Triggering Ultrasonic Sensor:

2.1 Trigger Ultrasonic Sensor:

Send a pulse to initiate the ultrasonic sensor and start the distance measurement process.

3. Distance Measurement:

3.1 Measure Echo Pulse Width (Time):

Capture the time it takes for the ultrasonic signal to return, representing the echo pulse width.

3.2 Convert Time to Distance:

Utilize the speed of sound to convert the measured time into a corresponding distance value.

4. Data Processing:

4.1 Process Distance Data:

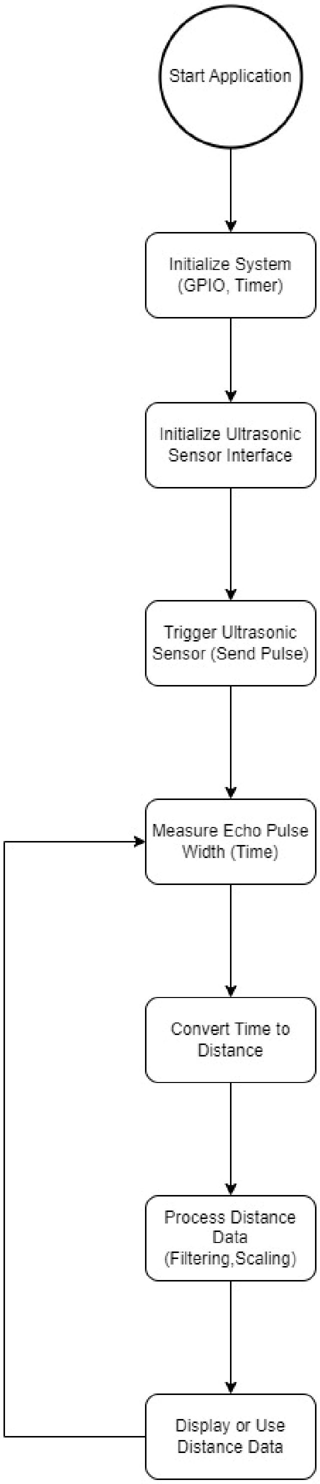
Apply any necessary processing to the distance data, such as filtering or scaling, to enhance accuracy or meet specific requirements.

5. Display or Use Distance Data:

5.1 Display Distance Data:

Present the processed distance data on a display or use it for further actions within the application.

6. Design


Start Application
Initialize System (GPIO, Timer)
Initialize Ultrasonic Sensor Interface
Trigger Ultrasonic Sensor (Send Pulse)
Measure Echo Pulse Width (Time)
Convert Time to Distance
Process Distance Data (Filtering,Scaling)
Display or Use Distance Data

1. *Initialize System:*

   Configure GPIO pins, timers, and other peripherals required for interfacing with the ultrasonic sensor.

2. *Initialize Ultrasonic Sensor Interface:*

   Set up the communication parameters and pins for the ultrasonic sensor.

3. *Trigger Ultrasonic Sensor:*

   Send a pulse to trigger the ultrasonic sensor.

4. *Measure Echo Pulse Width (Time):*

   Capture the time it takes for the ultrasonic signal to return (echo pulse width).

5. *Convert Time to Distance:*

   Utilize the known speed of sound to convert the measured time into a distance value.

6. *Process Distance Data:*

   Apply any necessary processing to the distance data, such as filtering or scaling.

7. *Display or Use Distance Data:*

   Present the distance data on a display or use it for further actions in the application.

8. *End Application.*
9. *Repeat it again.*

5 Potential risks and mitigation strategies

In the course of this project, each module has been meticulously crafted from the ground up, underscoring our commitment to enhancing the educational process. While the alternative of utilizing pre-existing frameworks was available, we consciously elected the more intricate route — constructing bespoke modules to afford us precise control over options and configurations.

A pivotal aspect of our development efforts involved the creation of a server employing Socket.IO, complemented by the establishment of a service layer for seamless communication with the server. Simultaneously, we devised a software solution for the ESP module, tailoring it to effectively interact with the aforementioned server.

In order to streamline the updating process, a CAN Bootloader was developed, contributing to the overall robustness of the project. However, it is pertinent to note that certain modules, such as the ESP8266, posed challenges due to a dearth of comprehensive documentation.

The integration of hardware components with the server proved to be a formidable task. Despite encountering substantial hurdles, we ultimately triumphed by implementing a sequential integration approach, systematically linking modules together to achieve a cohesive and functional system.

In summary, the strategic decision to construct custom modules and confront associated challenges head-on has not only enriched the project but also fortified its foundations, positioning it for sustained success in the realm of educational enhancement.

6 Conclusion

In conclusion, the project centered around implementing Firmware Over-The-Air (FOTA) for Electronic Control Units (ECUs) communicating with each other via the CAN Bus. A pivotal achievement within this initiative was the development of a specialized ESP8266 WiFi module, serving as a crucial component for seamless communication with the designated server.

Recognizing the need for an effective communication interface, a service layer was meticulously devised. Leveraging the capabilities of the Socket.IO server, this layer facilitated robust and real-time communication between the ESP8266 modules and the central server.

A noteworthy aspect of the project involved the integration of a CAN Bus Bootloader, enabling the streamlined and targeted update of firmware to specific ECUs. This CAN Bus-centric approach not only ensured efficient communication but also contributed to the project's overall reliability and scalability.

In essence, the successful execution of this project signifies a significant leap forward in the realm of firmware management for ECUs. The integration of advanced technologies, such as the ESP8266 module and the CAN Bootloader, underscores a commitment to cutting-edge solutions, ultimately paving the way for enhanced efficiency and functionality in the domain of automotive electronics.