PROJECT DOCUMENT FOR

FOTA

Firmware Over-The-Air

SPONSORED BY:

APPRAID TECH

Supervised By

Prof\ Tamer Mostafa

Sponsored By

Team Members

Mahmoud Sayed Mahmoud Helmy	**1804992**
Youssef Mamdouh Abdelaty	**1809348**
Ziad Ashraf Taha	**1805076**
Yosra Mahmoud Mohammed Attaher	**1801971**
Eslam Mohamed Sayed	**1807422**
Manar Ahmed Mohamed Omran	**1806133**
Amr Emad Eldeen	**1803188**
Seif El Dine Atef Shebl	**1803605**
Hussein Mahmoud Foaad	**1700462**
Yasmin Mohamed Abdelfattah	**1801886**
Nour Abdulnasser Fathelbab	**1806417**
Mahmoud Taher Rashad	**1809769**

Contents

Table of figures

1. INTRODUCTION

Nowadays equipment with Embedded systems is very common, it is normal to buy a fridge with heat sensor, light sensor to remind you when the door is not closed, also heaters, air conditioner and even cars either.

Cars have very complicated computer system it is based on many MCUs each has unique and important task, tasks in automotive are usually critical and should not be delayed.

so, what would we do if there is a bug in one of the MCUs code? or even if the publish company has an update to keep its system useful and meet customer requirements?

we all know that we can reprogram any MCU by hard wiring to JTAG programmer or by connecting the MCU to pc that has Bootloader to this MCU, but do we really need our customers to waste their time and money to keep their devices updated?! time and money can be saved if hard wiring is replaced by another wireless way, this way should be able to flash the code on the car's MCU without interrupting the main program which is already running. so we need to send firmware over the air. FOTA is abbreviation to firmware over the air, it means that the software will be sent to the target MCU in the car through wireless method and then be flashed and run on it in a secure way. FOTA is particularly useful when it comes to IOT systems, especially those with large numbers of connected devices that require frequent updates, **FOTA technology** allows manufactures to provide efficient and timely firmware updates for handsets, which increases customer satisfaction and reduces technical support requirements, it also allows manufactures to repair bugs in sold devices by remotely installation of new software.

In this project FOTA technology will be implemented from scratch, to do so we have to take in consideration some challenges, for example memory in target MCU should be able to keep the running project, the new installed one and also the bootloader, memory management and remapping should be done as the memory in embedded MCUs are partially limited so we have to use it efficiently, we also should make sure that the running program is not interrupted while flashing the new one to apply FOTA technology. another challenge is the design itself as we are working with many MCUs which are target MCU (car MCU) and telematics unit which receives the updated hex file from the server and send it to the target MCU, so we must program each of the MCUs separately then integrate all work.

2. PROJECT OVERVIEW

This system consists of four main blocks which are server, telematics unit, CAN module and target MCU.

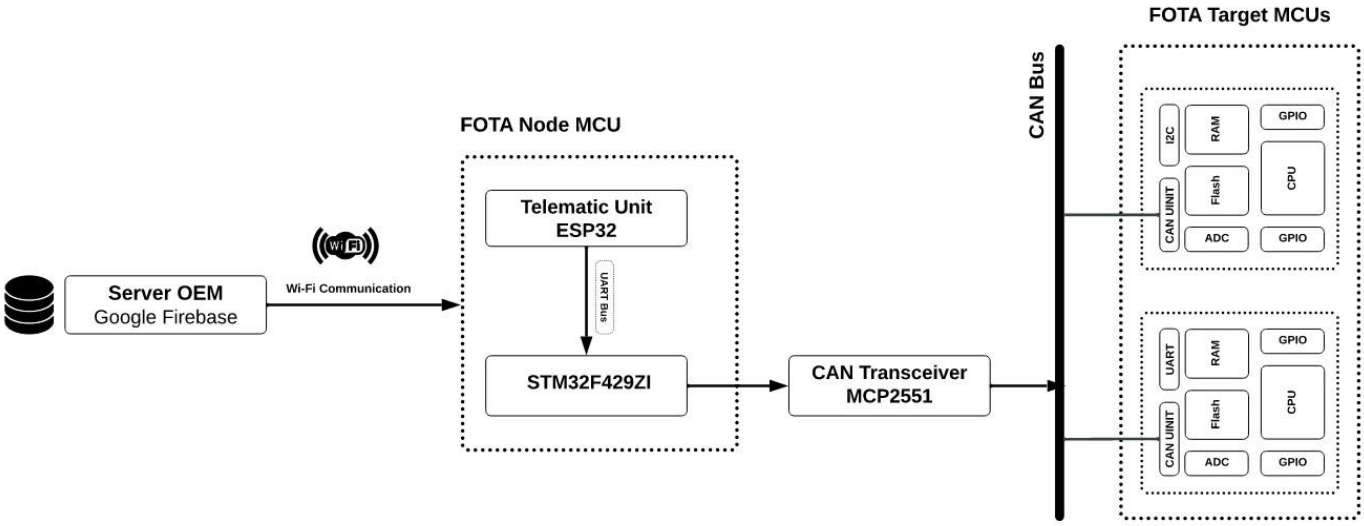


Figure 1.FOTA Block Diagram

2.1. Server

- **Cloud server** will be used as a database to keep the **.HEX file** of the new code, telematics unit will keep checking for updates on the server every now and then.
- **Firebase** is a free, open-source platform provided by google it is efficient to be used in small and medium scaled projects like ours as it is easy and has a real time database, it also lets you automatically run back-end code in response to event triggers by fire-base features and HTTPS requests. It stores code in Google cloud and run in a managed environment.
- **.HEX file** of the new code will be uploaded to the firebase through a GUI, this GUI will be simple, and it doesn't matter which programming language will be used to write it as all its function is to get the hex file and upload it to the firebase, then the firebase will store it in its database.

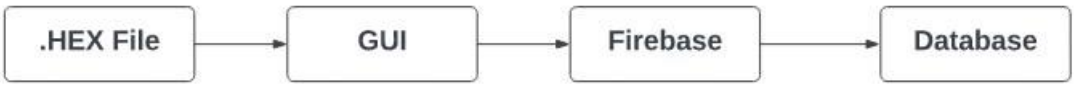


Figure 2.Server Block Diagram

2.2. Telematics Unit

- This unit is put in the car (target device) also, its main function is to keep checking if there is any update available on the server, when there is an available update, it should be downloaded as a hex file and stored in a puffer then sent to the main target MCU through CAN_BUS, the MCU used to implement this concept is known as Node MCU.
- To implement **Node MCU** concept we are using **ESP32** as it is open-source firmware and development kit that is used in prototyping and building IOT products, it also has built in WIFI module which allow it to keep connected to the internet without any additional peripherals to keep checking the server, it can connect to the cloud using HTTP and **MQTT** protocols, it also supports 2 wire interfacing (CAN Protocol).

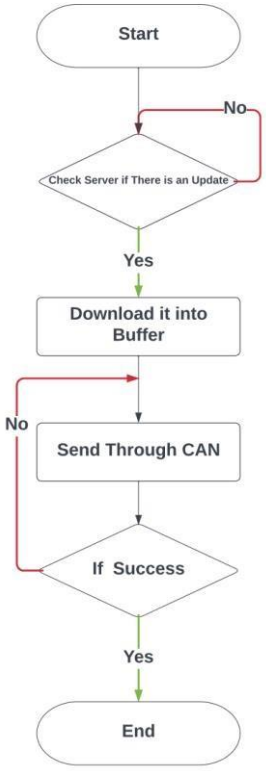


Figure 3.Telematics Unit Flow Chart

2.3. CAN Module

We use CAN to establish a connection between Node MCU and Target MCU, the next figure shows the connection using CAN transceiver which will be discussed later in detail.

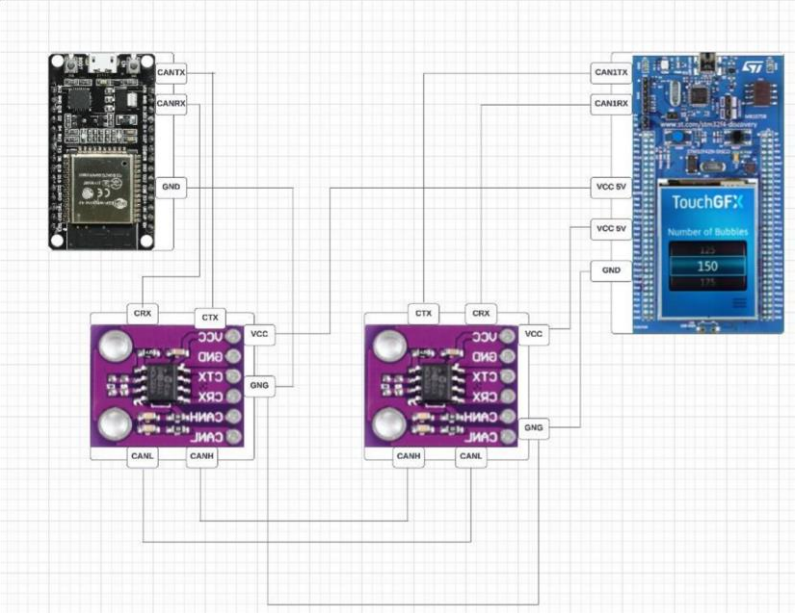


Figure 4.CAN Schematic

2.4. Target MCU

It is the main MCU in the target device e.g., **car**. which perform any specific task like measuring sensor readings to help user with parking, this MCU is connected to the Node MCU through CAN Bus, when it receives the **.HEX File** of the new code it flashes it on the memory then run it.

In order to make flashing process there should be a bootloader system in this MCU, and in order to make sure that the already running program will not be interrupted even if the MCU is receiving code from CAN Bus we should use **dual bank memory MCU**, so if the code is running is bank 1 the new code received from the CAN Bus can be flashed independently on bank 2. Each bank memory should be big enough to hold the target program and its own bootloader code.

For this purpose, **STM32F429** is used as the target MCU as it supports dual-bank memory which allow us to implement **OTA** technology, each bank is **1MB** memory so both target program and bootloader can fit in each bank. STM32F429 also supports TWI (Two Wire Interfacing or CAN Protocol), so it is compatible with the project needs.

3. CONTROLLER AREA NETWORK - CAN

CAN bus is a **message-based** protocol designed to allow the Electronic Control Units (ECUs) found in today's automobiles, as well as other devices, to communicate with each other in a reliable, priority-driven fashion.

3.1. CAN Characteristics

1. **Serial**
2. **Multi-Master**
3. **Message Broadcast System:** this means that all nodes can "hear" all transmissions. There is no way to send a message to just a specific node.
4. **Operate at speeds from 20 kb/s to 1 Mb/s.**
5. **Half Duplex**
6. **In this protocol, smaller IDs have higher priority, so we send MSB first.**
   e.g., Light protocol management with built-in features for error detection and retransmission.
7. **Used in many application domains including automotive, medical devices, robotics and more.**
8. **CAN support is included in automotive standards including OSEKCom and AUTOSAR.**

3.2. The Development of CAN

The development of CAN began when more and more electronic devices were implemented into modern motor vehicles. Examples of such devices include engine management systems, active suspension, ABS, gear control, lighting control, air conditioning, airbags, and central locking. All this means more safety and more comfort for the driver and of course a reduction in fuel consumption and exhaust emissions.

To improve the behavior of the vehicle even further, it was necessary for the different control systems (and their sensors) to exchange information. This was usually done by discrete interconnection of the different systems (i.e., point to point wiring). The requirement for information exchange has then grown to such an extent that a cable network with a length of up to several miles and many

connectors was required. This produced growing problems concerning material cost, production time and reliability.

Before and After CAN

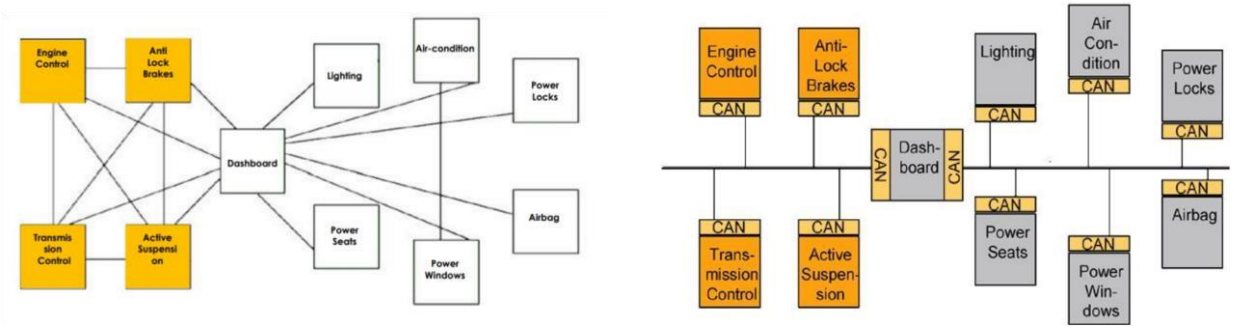


Figure 5.before and after can

The figures shown demonstrate the numerous advantages which the CAN Communication offers. As more and more features are added to the machine (e.g. an automobile), more devices require sharing information back and forth. For example, if you need to display analytics about the conditions of your automobile's features (such as ALB or lighting), you'll have to connect every single feature to your dashboard. Not to mention if one device is required to be connected to more than one device. Thus, point to point communication (fig. 6A) becomes tedious, as too many wires are used to connect between devices, which in turn gives rise to increasing resistances, delays and high vulnerabilities to noise.

CAN communication (fig. 6B) is a lot more straightforward, where each of the devices which are required to communicate are simply connected to one bus, and each device has its own CAN controller, altogether called a 'node.' The way that works is through broadcasting: the data is broadcast through the bus, and all the nodes 'hear' the transmitted data, then each node filters the message that interests it using the CAN controller. A message consists of a group of frames, each frame with its own function. The CAN controller uses the ID frame to filter the messages it requires. This is in the case of message-based protocol.

3.3. Basic Configuration

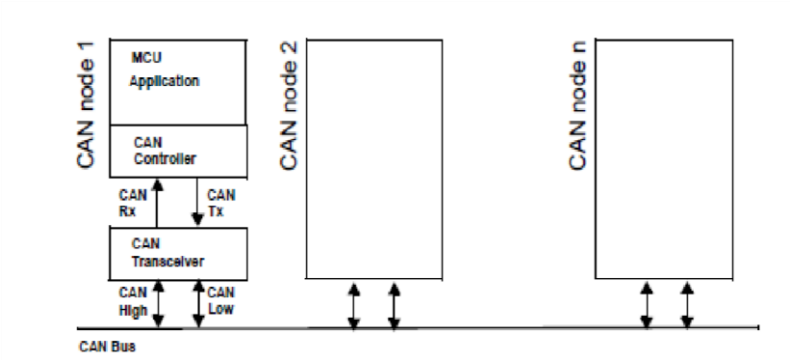


Figure 6.CAN Basic Configuration

The figure (fig. 7) shows a typical CAN network. It consists of several nodes. Each device has a host controller (ECU/MCU), which is responsible for the function of a specific node, and the CAN controller and the transceiver.

In automobiles, you want to cover a greater distance in order to connect the devices together to allow communication. This may be 1m, 2m or even up to 40 meters in some cases. CAN controllers work with digital signals, which are preferred for short ranges and not the cases mentioned. So, this where the CAN transceiver comes into play. The transceiver translates the digital signals into CAN High (CANH) and CAN Low (CANL) signals, which are differential and complementary signals. This makes great room for noise immunity, as in the case of noise, both CANH and CANL will experience the same noise or at least roughly identical noise, and when subtracted at the other transceiver terminal, they cancel out automatically.

Terminating resistances should be connected between the CANH and CANL at the CAN bus terminals. The use they perform is as follows:

-Ensure that the bus quickly reaches the recessive state, allowing the parasitic capacitance's energy to dissipate more quickly.

-Enhance anti-interference performance by allowing high-frequency and lowenergy signals to fade away fast.

The points discussed allow us to deduce many advantages of CAN communication:

- ✦ High noise immunity.
- ✦ Wiring simplicity.

✦ One can use as many nodes as needed, hence scalability. ✦ Multi-master (any device can communicate with any other) ✦ Long ranges.

✦ Reliability.

3.4. CAN Encoding

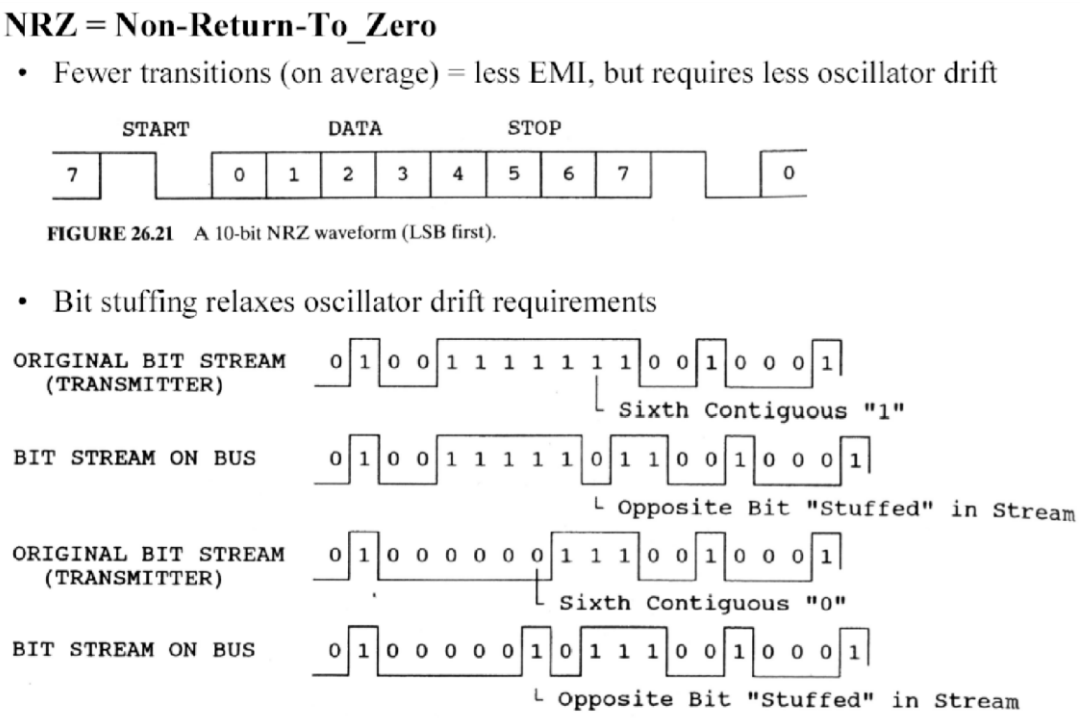


Figure 7.can

3.5. CAN Bus Characteristics

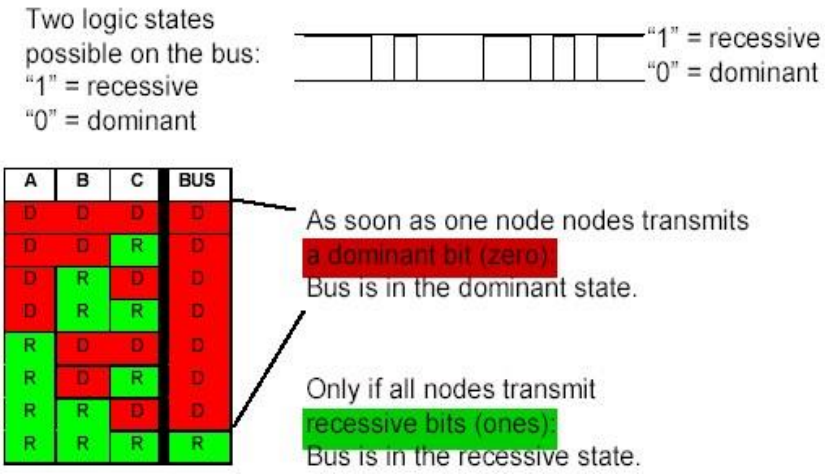


Figure 8. Example to clarify dominant and recessive bits concept

3.6. Bus Characteristics – Wired AND

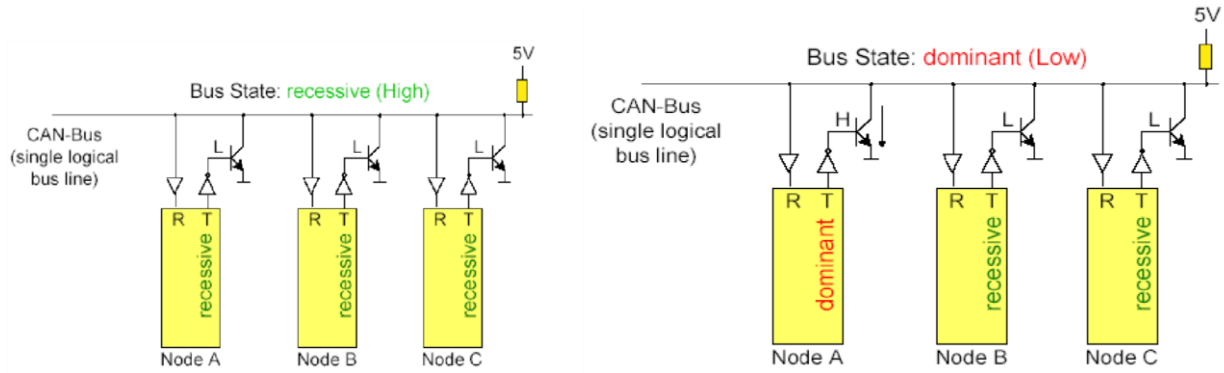


Figure 9.shape of nodes during T and R

T is Transmitter, R is receiver. Note: nodes can therefore check the line while transmitting. This is important particularly during arbitration.

3.7. CAN Arbitration

CAN nodes wait for “bus idle” before starting transmission.

Each node starts to transmit its message ID.

If a node transmits “1” and detects “0” on the bus, it stops transmitting (loses arbitration).

The node that wins arbitration completes transmission till the end of its frame.

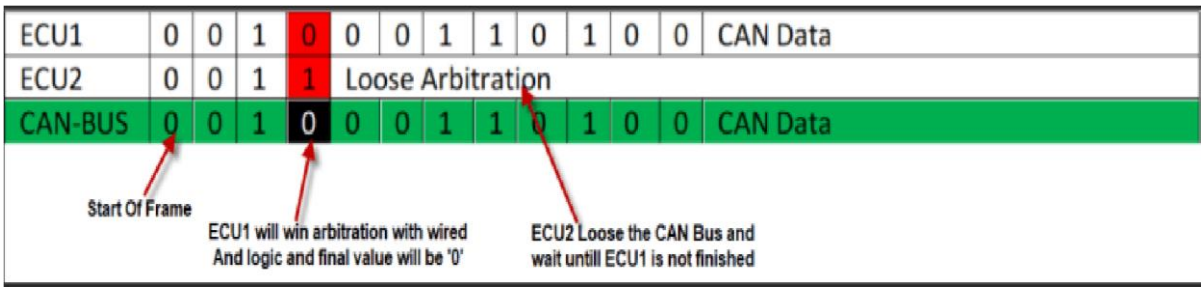


Figure 10. Example to clarify Bus arbitration

3.8. CAN Standard

There are four types of CAN frames (By usage):

- **Data Frame**
- **Remote Frame**
- **Error Frame**
- **Overload Frame**

There are **two types** of CAN frames (By ID):

1) **Standard Frame**: 11-bit identifier

2) **Extended Frame**: 29-bit identifier

The messages use a clever scheme of bit-wise arbitration to control access to the bus, and each message is tagged with a priority.

The CAN standard also defines an elaborate scheme for error handling and confinement.

1. Data Frame

The Data Frame is the most common message type.

The Arbitration Field, which determines the priority of the message when two or more nodes are contending for the bus. The Arbitration Field contains:

- For CAN 2.0A, an 11-bit Identifier and one bit, the RTR bit, which is dominant for data frames.
- For CAN 2.0B, a 29-bit Identifier (which also contains two recessive bits: SRR and IDE) and the RTR bit.

The Data Field, which contains zero to eight bytes of data.

The CRC Field, which contains a 15-bit checksum calculated on most parts of the message. This checksum is used for error detection.

An Acknowledgement Slot: any CAN controller that has been able to correctly receive the message sends an Acknowledgement bit at the end of each message. The transmitter checks for the presence of the Acknowledge bit and retransmits the message if no acknowledge was detected.

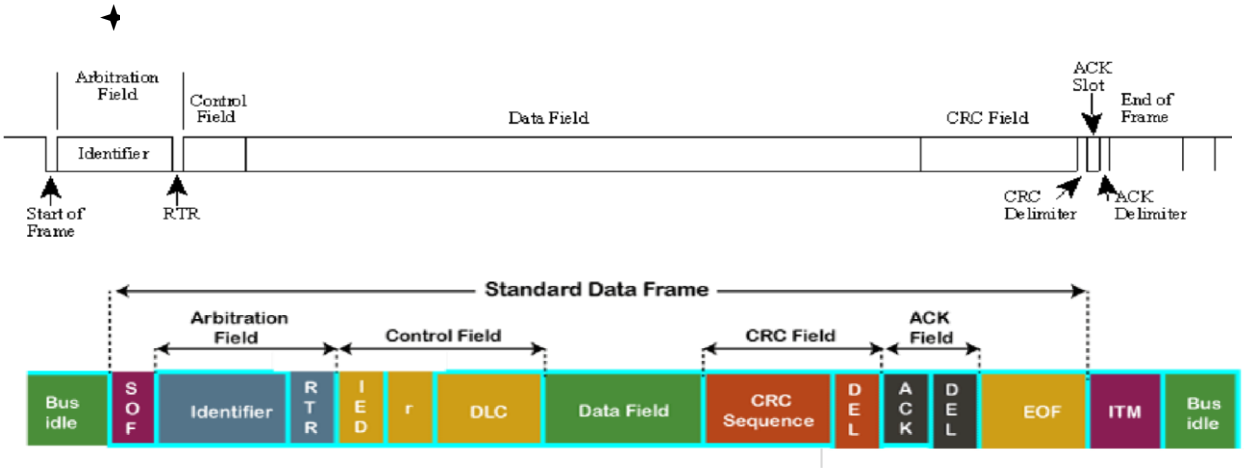


Figure 11.standard can 11-bit id

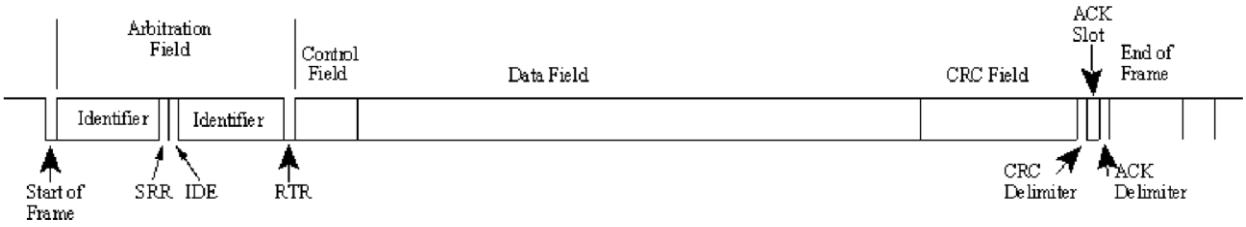


Figure 12.standard can 29-bit id

2. Remote Frame

Remote Frames are used to request data from others.

The Remote Frame is just like the Data Frame, with two important differences:

- It is explicitly marked as a Remote Frame (the RTR bit in the Arbitration Field is recessive).
- There is no Data Field.

DLC represents length of expected data frame. A
Remote Frame (2.0A type):

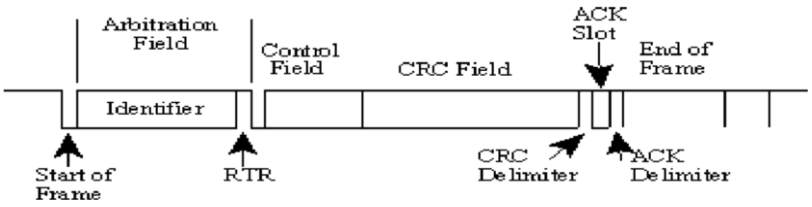


Figure 13. Remote frame

3. Error Frame

Transmitted by an ECU when it detects errors on bus.

A special message that violates the encoding rules (no bit stuffing occurs) to make other nodes detect a fault.

Consists of 2 fields:

- **Error Flag field** (6 to 12 bits of same level)
- **Error Delimiter field** (8 recessive bits)

4. Overload frame

Used to provide extra delay between successive data or remote frames sent on bus.

At most two overload frames may be generated per time.

The Overload Frame is mentioned here just for completeness. It is very similar to the Error Frame regarding the format, and it is transmitted by a node that becomes too

busy. The Overload Frame is not used very often, as today's CAN controllers are clever enough not to use it.

Consists of 2 fields:

- **Overload Flag field** (6 to 12 bits dominant)
- **Overload Delimiter field** (8 recessive bits)

3.9. Identifier Filters modes

Mask mode: Identifier registers are associated with mask registers specifying which bits of the identifier are identified as must match or as don’t care.

Identifier list mode: Mask registers are used as identifier registers. All bits of the incoming identifier must match the bits specified in the filter registers.

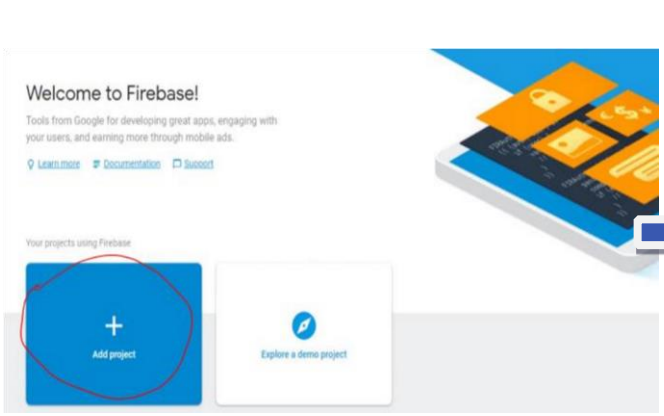
3.10. Major Tasks

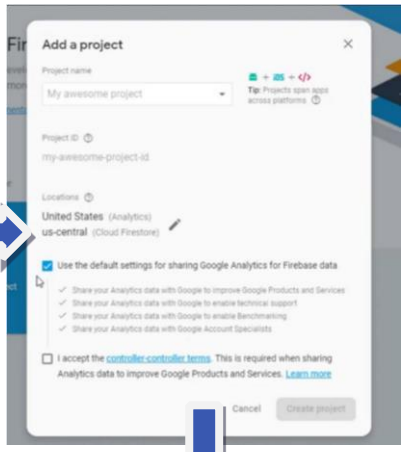
We can summarize the work done in CAN in two major points:

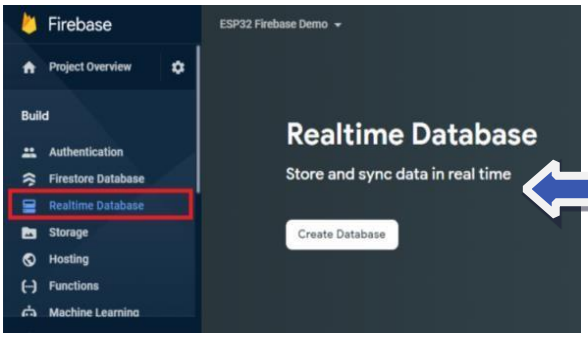
- **Loop back mode test for STM32:** This mode is provided for self-test functions.
- **Normal Mode test:** This mode is for communication between STM32 and the telematics units (ESP32).

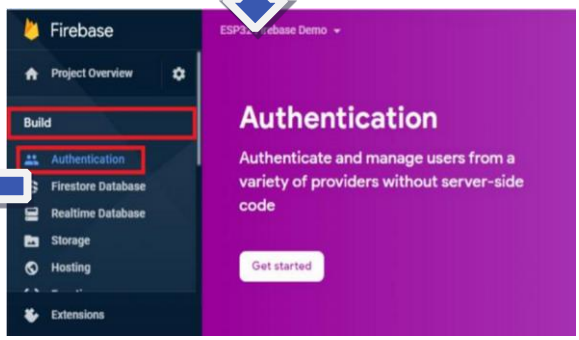
4. DESIGN & IMPLEMENTATION

4.1. Server



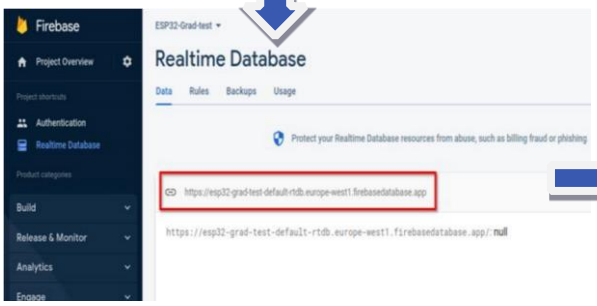


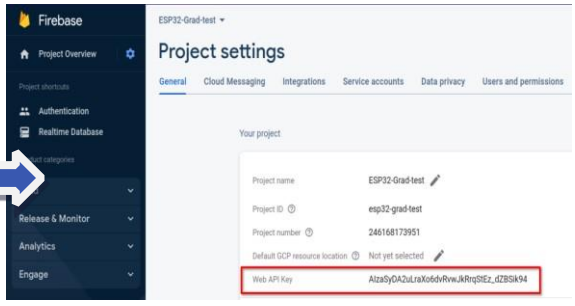




Then creating real-time database

Then set authentication mode.





we need to copy and save the database URL—highlighted in the following image—because you'll need it later in your ESP32 code.

Then get project API.

4.2. Telematics Unit

ESP32 is Arduino base, so first we had to set configurations of Arduino-IDE in order to get start, to make the setup we followed the following steps:

1. In your Arduino IDE, go to **File**>**Preferences.**
2. Enter the following into the "**Additional Board Manager** URLs" field: **https://raw.githubusercontent.com/espressif/arduino-esp32/ghpages/package_esp32_index.json**
3. Then, click the "**OK**" button:
4. Open the Boards Manager. Go to **Tools > Board > Boards Manager**.
5. Search for **ESP32** and press **install** button.

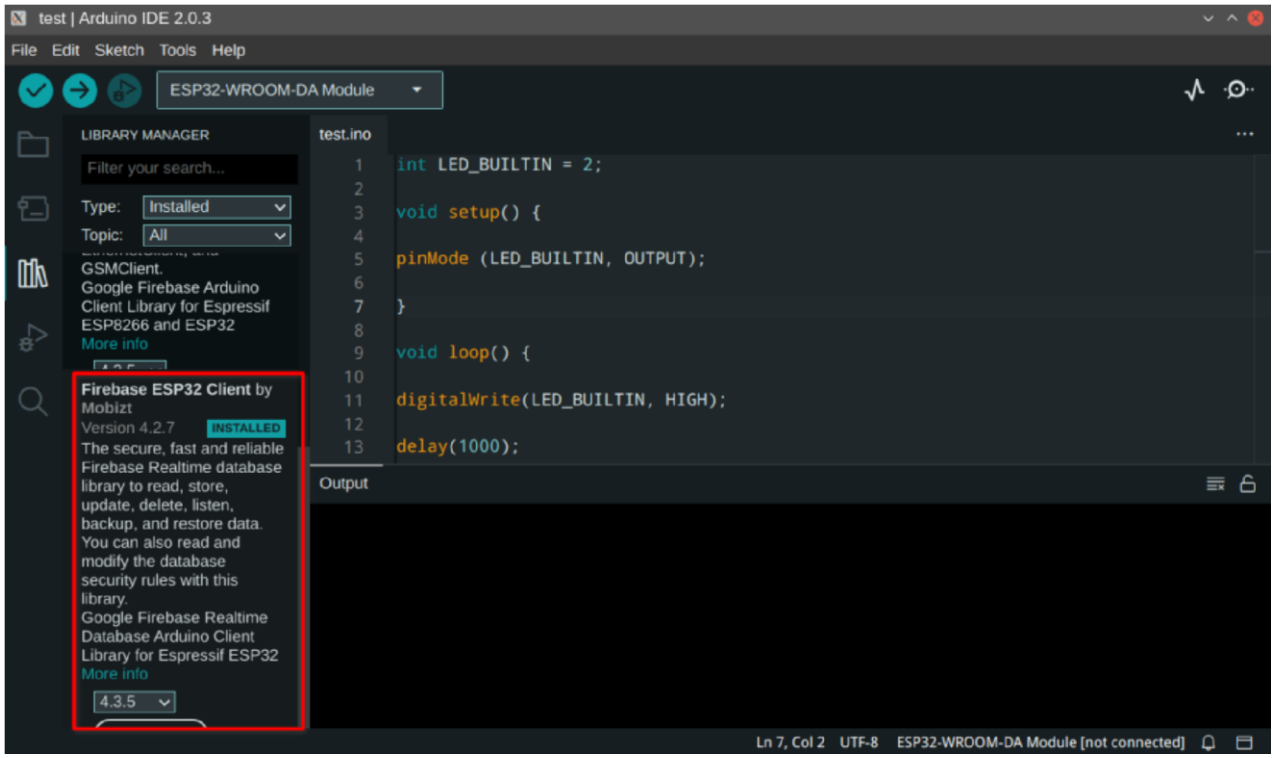


Figure 14. ESP 32 package installed on Arduino ide

To connect Firebase to ESP32 we must Install the Firebase-ESP-Client Library.

1. Go to **Sketch > Include Library > Manage Libraries**
2. Search for Firebase ESP Client **and install the** Firebase Arduino Client Library for ESP8266 and ESP32 **by Mobitz.**

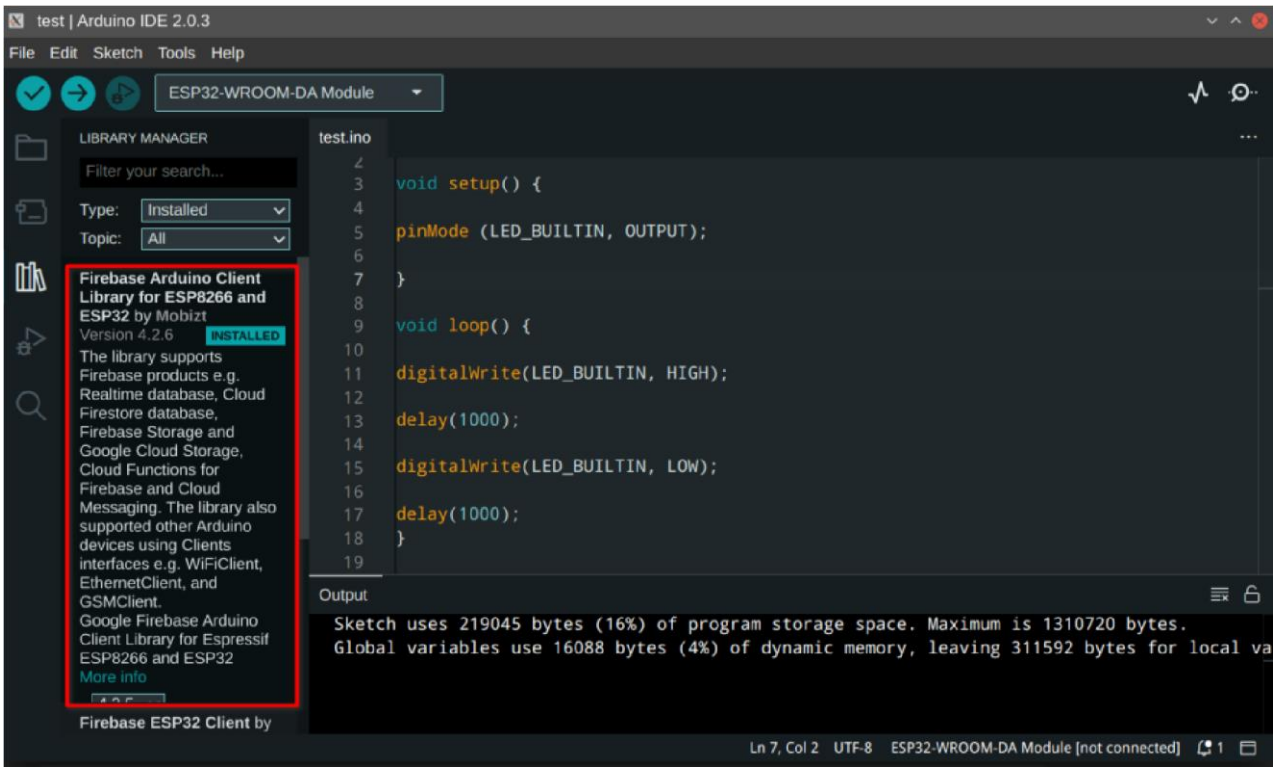


Figure 15. firebase package installed on Arduino ide

To test this configuration a simple sketch is made to inserts an int and a float number into the database every 15 seconds. This is a simple example showing you how to connect the ESP32 to the database and store data.

```
#include <Arduino.h>

#if defined(ESP32)
#include <WiFi.h>
#endif
#include <Firebase_ESP_Client.h>

//Provide the token generation process info.
#include "addons/TokenHelper.h"
//Provide the RTDB payload printing info and other helper functions.
#include "addons/RTDBHelper.h"

// Insert your network credentials
#define WIFI_SSID "REPLACE_MY_YOUR_SSID"
#define WIFI_PASSWORD "REPLACE_WITH_MY_PASSWORD"

// Insert Firebase project API Key
#define API_KEY "AIzaSyDA2uLraXo6dvRvwJkRrqStEz_dZBSik94 "

// Insert RTDB URLefine the RTDB URL */
#define DATABASE_URL "https://esp32-grad-test-default-rtdb.europe west1.firebasedatabase.app/"

//Define Firebase Data object
FirebaseData fbdo;

FirebaseAuth auth;
FirebaseConfig config;

unsigned long sendDataPrevMillis = 0;
int count = 0;
bool signupOK = false;

void setup(){
Serial.begin(115200);
WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
Serial.print("Connecting to Wi-Fi");
while (WiFi.status() != WL_CONNECTED){
Serial.print(".");
delay(300);
}
Serial.println();
Serial.print("Connected with IP: ");
Serial.println(WiFi.localIP());
Serial.println();

/* Assign the api key (required) */
config.api_key = API_KEY;

/* Assign the RTDB URL (required) */
config.database_url = DATABASE_URL;

/* Sign up */
if (Firebase.signUp(&config, &auth, "", "")){
Serial.println("ok");
signupOK = true;
}
else{
Serial.printf("%s\n", config.signer.signupError.message.c_str());
}
/* Assign the callback function for the long running token generation task */
config.token_status_callback = tokenStatusCallback; //see addons/TokenHelper.h
Firebase.begin(&config, &auth);
```

Figure 16.test code

```
Firebase.reconnectWiFi(true);
}
void loop(){
if (Firebase.ready() && signupOK && (millis() - sendDataPrevMillis > 15000 || sendDataPrevMillis == 0)){
sendDataPrevMillis = millis();
// Write an Int number on the database path test/int
if (Firebase.RTDB.setInt(&fbdo, "test/int", count)){
Serial.println("PASSED");
Serial.println("PATH: " + fbdo.dataPath());
Serial.println("TYPE: " + fbdo.dataType());
}
else {
Serial.println("FAILED");
Serial.println("REASON: " + fbdo.errorReason());
}
count++;
// Write an Float number on the database path test/float
if (Firebase.RTDB.setFloat(&fbdo, "test/float", 0.01 + random(0,100))){
Serial.println("PASSED");
Serial.println("PATH: " + fbdo.dataPath());
Serial.println("TYPE: " + fbdo.dataType());
}
else {
Serial.println("FAILED");
Serial.println("REASON: " + fbdo.errorReason());
}
}
}
```

This part is still in progress.
Part related to CAN initialization in the following chapter.

Figure 17.test code

5. TARGET MCU MEMORY

As mentioned in the overview, target MCU memory is divided into 2 banks, when an application is running only one bank is activated (the bank that has the software flashed on it by the vendor) the other bank is inactive until a new code is received from CAN_BUS, when receiving a code (hex file) the boot-loader of bank 2 is activated and start flashing it on its own bank, all these steps obeys FOTA technology.

In order to do so, there should be bank swapping technique between banks and we have to construct boot-loader software and put it in each bank, each bank also will has a branching unit which takes the decision if the application or the boot-loader will run, there will be a reset unit in both application and boot-loader sector to allow going back to the branching unit if needed.

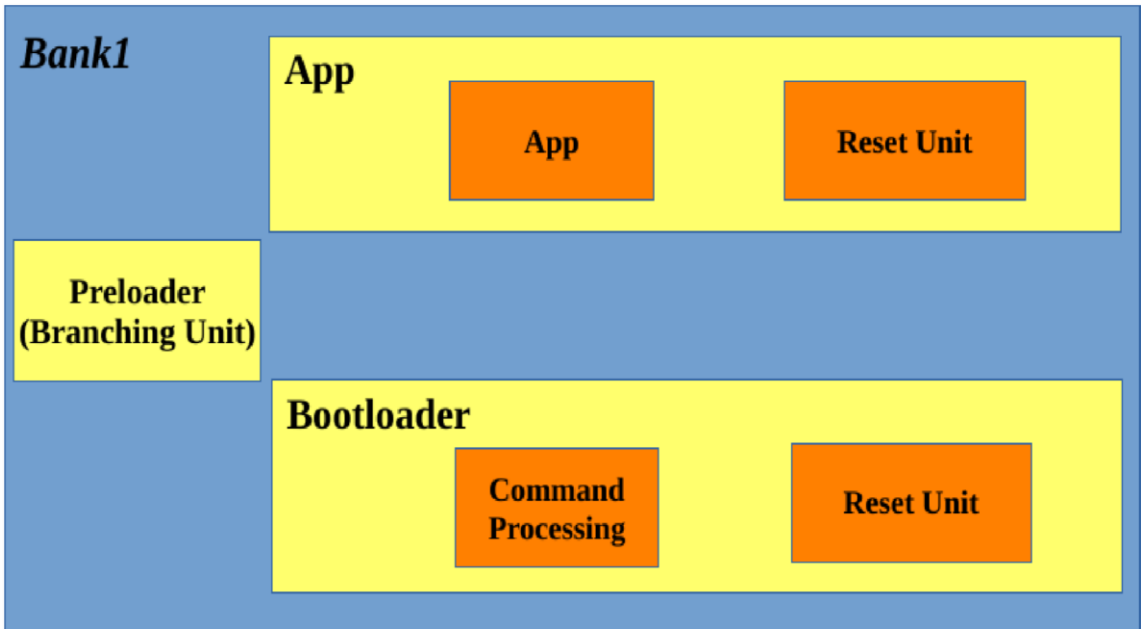


Figure 18.content of a bank

It is important to clarify that each of application, branching unit and bootloader are separated programs to prevent any issues when a new application is flashed, each of them has its own vector table and initialization, and are flashed in specific sectors in each bank using direct addressing for the memory sectors, this shown in the following project explorer.

5.1. Branching Unit

We can name this application as the decision maker, as it decides which program in the bank will run by checking a push button if it is pushed boot-loader mode will run else application mode runs.

```c
static void preloader(void)
    {
    if(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0)== GPIO_PIN_SET)
    {
    // Jump to bootloader
  goto_bootloader();
  }
    else
    {
  // Jump to application
  goto_application();
  }
}
```

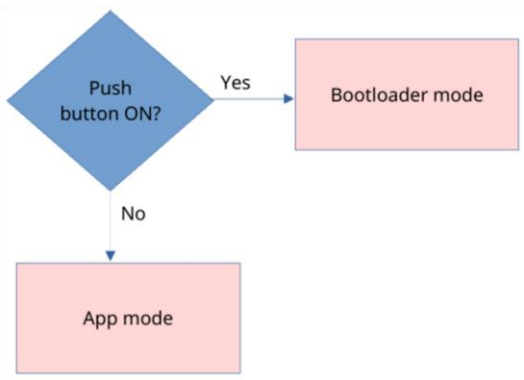


Figure 19.branching unit

As show in the previous code, there are 2 main functions one to go to boot-loader mode and other to the application mode both functions change the stack pointer and reset handler addresses then make soft reset.

```c
static void goto_bootloader(void)
{
    // Turn ON the Green Led
    HAL_GPIO_WritePin(GPIOG, GPIO_PIN_13, GPIO_PIN_SET );

    // tell the user that Bootloader is running
  printf("Starting Bootloader \n");

    void (*app_reset_handler)(void) = (void*)(*((volatile uint32_t*) (0x08002000 + 4U)));

  __set_MSP(*(volatile uint32_t*) 0x08002000);

  app_reset_handler();    //call the app reset handler
}
```

```c
static void goto_application(void)
{
    // Turn OFF the Green Led to tell the user that Bootloader is not running
  HAL_GPIO_WritePin(GPIOG, GPIO_PIN_13, GPIO_PIN_RESET );

    //tell the user that jummping to app is running
  printf("Starting Application\n");

  void (*app_reset_handler)(void) = (void*)(*((volatile uint32_t*) (0x08040000 + 4U)));

  __set_MSP(*(volatile uint32_t*) 0x08008000);

  app_reset_handler();    //call the app reset handler
}
```

Figure 20.goto_application & goto_bootloader functions

5.2. Application Unit

This application is the main task that should be running all time by the MCU, in this project we can disregard what exactly is this application doing therefore we can make a simple application that toggle a led just for testing.

```
  while (1)
  {
    /* USER CODE END WHILE */
    HAL_GPIO_WritePin( GPIOG, GPIO_PIN_14, GPIO_PIN_SET );
    HAL_Delay(1000);  //1 Sec delay
    HAL_GPIO_WritePin( GPIOG, GPIO_PIN_14, GPIO_PIN_RESET );
    HAL_Delay(1000);    //1 Sec delay
    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

Figure 21.app unit

Application should also raise a flag when anything is received by CAN Bus, this flag starts the reset unit.

5.3. Reset Unit

It is a reset function in the application project and also the boot-loader project, its main purpose is to set push button on and go back to branching unit by changing stack pointer an reset handler.

5.4. Bootloader

Bootloader is an application whose primary purpose is to allow the system software to be updated without the use of hard wiring programmer like JTAG programmer, boot-loader should be compatible to the communication protocol from which it receive the new code (CAN), it should also has the ability to switch between memory banks and select the operating mode (Application / Boot Mode), there should be a flashing system that can erase, read or write code in memory and also error detection mechanisms should be implemented.

To make the boot-loader able to flash any new code, flashing mechanism should be obeyed, first the flash memory should be erased, then write the new code (flashing), then read the code during execution.

To perform erasing to specific sectors (sectors that carry application only) the following steps should be followed:

- Check if the sector number valid (from 0 to 11) as there are 12 sectors in each bank, if sector number is 0xff that means mass erase to all sections in bank.
- Calculate how many sectors need to be erased. ✦ Choose erase type (sector, block, bank) ✦ Set sectors number.
- Unlock flashing.
- Erase
- Lock flashing

5.4.1 Erasing function

```
uint8_t execute_flash_erase(uint8_t sector_number , uint8_t number_of_sector)
{
    //we have totally 12 sectors in one bank .. sector[0 to 11]
    //number_of_sector has to be in the range of 0 to 11
    // if sector_number = 0xff , that means mass erase !
    FLASH_EraseInitTypeDef flashErase_handle;
    uint32_t sectorError;
    uint8_t status = HAL_ERROR;

    if( number_of_sector > 12 )
        return INVALID_SECTOR;
    if( (sector_number == 0xff ) || (sector_number <= 12) )
    {
        if(sector_number == (uint8_t) 0xff)
        {
            flashErase_handle.TypeErase = FLASH_TYPEERASE_MASSERASE;
        }
        else
        {
            /*Here we are just calculating how many sectors needs to erased */
            uint8_t remanining_sector = 12 - sector_number;
            if( number_of_sector > remanining_sector)
            {
                number_of_sector = remanining_sector;
            }
            flashErase_handle.TypeErase = FLASH_TYPEERASE_SECTORS;
            flashErase_handle.Sector = sector_number; // this is the initial sector
            flashErase_handle.NbSectors = number_of_sector;
        }
        flashErase_handle.Banks = FLASH_BANK_1;

        /*Get access to touch the flash registers */
        HAL_FLASH_Unlock();
        flashErase_handle.VoltageRange = FLASH_VOLTAGE_RANGE_3;  // our MCU will work on this voltage range
        status = (uint8_t) HAL_FLASHEx_Erase(&flashErase_handle, &sectorError);
        HAL_FLASH_Lock();

        return status;
    }
```

Figure 22.erasing function

5.4.2 Writing function

```
uint8_t execute_mem_write(uint8_t *pBuffer, uint32_t mem_address, uint32_t len)
{
    uint8_t status = HAL_ERROR;

    //We have to unlock flash module to get control of registers
    HAL_FLASH_Unlock();

    for(uint32_t i = 0 ; i <len ; i++)
    {
        //Here we program the flash byte by byte
        status = HAL_FLASH_Program(FLASH_TYPEPROGRAM_BYTE,mem_address+i,pBuffer[i] );
    }

    HAL_FLASH_Lock();

    return status;
}
```

Figure 23.writing function

All these steps are shown in the following code.

HAL_FLASH_Program is a function generated by STM32CubeIDE it is found in **stm32f4xxx_hal_flash.c**, it follows the flashing steps which are: ✦ Lock the process to prevent being interrupted with any other process ✦ Check the parameters.

- ✦ Wait for last operation to be completed.
- ✦ Choose way of flashing (byte / half word / word / double word) ✦ Start flashing.
- ✦ Wait till flashing ends.
- ✦ Unlock process.

```
HAL_StatusTypeDef HAL_FLASHEx_Erase(FLASH_EraseInitTypeDef *pEraseInit, uint32_t *SectorError)
{
  HAL_StatusTypeDef status = HAL_ERROR;
  uint32_t index = 0U;

  /* Process Locked */
  __HAL_LOCK(&pFlash);

  /* Check the parameters */
  assert_param(IS_FLASH_TYPEERASE(pEraseInit->TypeErase));

  /* Wait for last operation to be completed */
  status = FLASH_WaitForLastOperation((uint32_t)FLASH_TIMEOUT_VALUE);

  if (status == HAL_OK)
  {
    /*Initialization of SectorError variable*/
    *SectorError = 0xFFFFFFFFU;

    if (pEraseInit->TypeErase == FLASH_TYPEERASE_MASSERASE)
    {
      /*Mass erase to be done*/
      FLASH_MassErase((uint8_t) pEraseInit->VoltageRange, pEraseInit->Banks);

      /* Wait for last operation to be completed */
      status = FLASH_WaitForLastOperation((uint32_t)FLASH_TIMEOUT_VALUE);

      /* if the erase operation is completed, disable the MER Bit */
      FLASH->CR &= (~FLASH_MER_BIT);
    }
    else
    {
      /* Check the parameters */
      assert_param(IS_FLASH_NBSECTORS(pEraseInit->NbSectors + pEraseInit->Sector));

      /* Erase by sector by sector to be done*/
      for (index = pEraseInit->Sector; index < (pEraseInit->NbSectors + pEraseInit->Sector); index++)
      {
        FLASH_Erase_Sector(index, (uint8_t) pEraseInit->VoltageRange);

        /* Wait for last operation to be completed */
        status = FLASH_WaitForLastOperation((uint32_t)FLASH_TIMEOUT_VALUE);

        /* If the erase operation is completed, disable the SER and SNB Bits */
        CLEAR_BIT(FLASH->CR, (FLASH_CR_SER | FLASH_CR_SNB));

        if (status != HAL_OK)
        {
          /* In case of error, stop erase procedure and return the faulty sector*/
          *SectorError = index;
          break;
        }
      }
    }
    /* Flush the caches to be sure of the data consistency */
    FLASH_FlushCaches();
  }

  /* Process Unlocked */
  __HAL_UNLOCK(&pFlash);

  return status;
}
```

Figure 24.cubemx function assistant for erasing

✦ **All these steps are shown in the following code.**

```
HAL_StatusTypeDef HAL_FLASH_Program(uint32_t TypeProgram, uint32_t Address, uint64_t Data)
{
  HAL_StatusTypeDef status = HAL_ERROR;
  /* Process Locked */
  __HAL_LOCK(&pFlash);
  /* Check the parameters */
  assert_param(IS_FLASH_TYPEPROGRAM(TypeProgram));
  /* Wait for last operation to be completed */
  status = FLASH_WaitForLastOperation((uint32_t)FLASH_TIMEOUT_VALUE);
  if(status == HAL_OK)
  {
    if(TypeProgram == FLASH_TYPEPROGRAM_BYTE)
    {
      /*Program byte (8-bit) at a specified address.*/
      FLASH_Program_Byte(Address, (uint8_t) Data);
    }
    else if(TypeProgram == FLASH_TYPEPROGRAM_HALFWORD)
    {
      /*Program halfword (16-bit) at a specified address.*/
      FLASH_Program_HalfWord(Address, (uint16_t) Data);
    }
    else if(TypeProgram == FLASH_TYPEPROGRAM_WORD)
    {
      /*Program word (32-bit) at a specified address.*/
      FLASH_Program_Word(Address, (uint32_t) Data);
    }
    else
    {
      /*Program double word (64-bit) at a specified address.*/
      FLASH_Program_DoubleWord(Address, Data);
    }
    /* Wait for last operation to be completed */
    status = FLASH_WaitForLastOperation((uint32_t)FLASH_TIMEOUT_VALUE);
    /* If the program operation is completed, disable the PG Bit */
    FLASH->CR &= (~FLASH_CR_PG);
  }
  /* Process Unlocked */
  __HAL_UNLOCK(&pFlash);
  return status;
```

Figure 25.cubemx function assistant for writing

5.5 Error checking

Error checking is a very important task implemented by bootloader as in digital communication systems, it is perfectly possible that data gets corrupted or not completely received, Corruption can be from the source or in the communication medium, and there are lots of techniques to detect the errors.

Cyclic redundancy check (CRC)

CRC is widely-used technique for detecting errors in digital data, both during transmission and storage, it is calculated based on the data transmitted and a polynomial, and transmitted with the data, There are 4-bit & 8-bit & 16-bit and 32bit CC, If the width of the CRC is M then The width of the polynomial is M+1, The receiver can determine whether or not the data is corrupted by recalculating the CRC and compare it with the received CRC, if that CRC equal to the received one, there is no bit occurred during transmission or by computing the CRC over the whole received data (data and CRC), if the new CRC is zero, there is no bit occurred during transmission.

There are two methods for implementation:

- **Shifting** it handles one bit at a time, make register with the polynomial width

, shift the register left by one bit, reading the next bit of the message into register bit position 0, if the MSB equal 1 and is popped out then " register = register XOR polynomial "

- **Table-Driven**
    - It handles **one byte** at a time.

For one byte: Get the CRC of the first byte from the tables depending on our polynomial.

- Calculate the previous CRC XOR 2nd byte.
- Get the CRC of the previous result from the tables depending on our polynomial and so on.

For 2 or 4 bytes:

- Get the CRC of the first byte from the tables depending on our polynomial.
- Calculate the previous CRC **XOR** 2nd byte.

- ✦ Get the CRC of the previous result from the tables depending on our polynomial and so on/
- ✦ Calculate the previous CRC **XOR** (shifting the CRC of the first byte) and so on.

STM32F4 calculation unit

- Use 32-bit CRC.
- Its polynomial is 0x4C11DB7.
- Single input/output 32-bit data register
- Each write operation into the data register creates a combination of the previous CRC value and the new one.
- The write operation is stalled until the end of the CRC computation, thus allowing back-to back write accesses or consecutive write and read accesses.
- The CRC calculator can be reset to 0xFFFF FFFF with the RESET control bit in the CRC_CR register. This operation does not affect the
- Contents of the CRC_IDR register.

CRC functional description

The CRC calculation unit mainly consists of a single 32-bit data register, which: Used as an input register to enter new data in the CRC calculator (when writing into the register) holds the result of the previous CRC calculation (when reading the register) Each write operation into the data register creates a combination of the previous CRC value and the new one (CRC computation is done on the whole 32-bit data word, and not byte per byte).

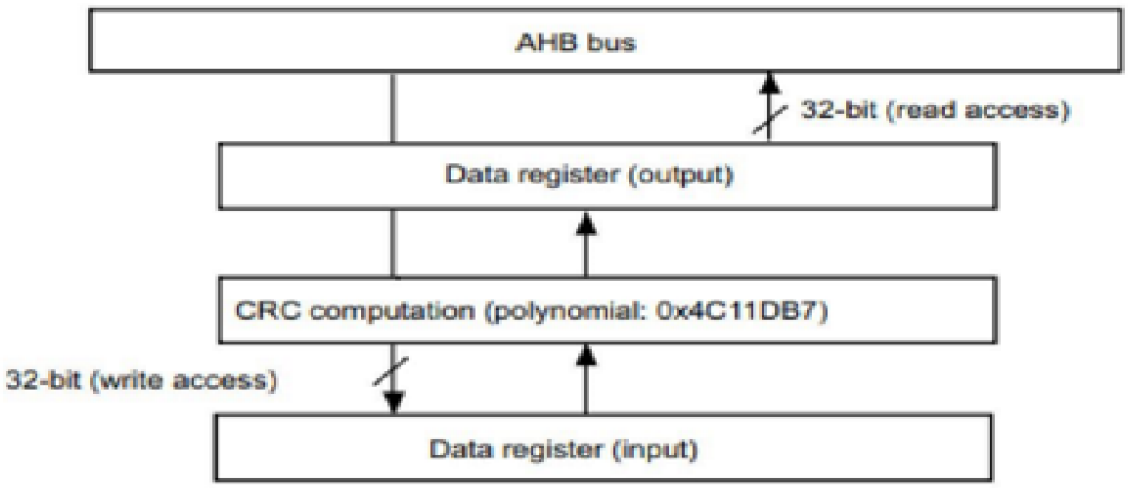


Figure 26.calculation unit diagram

DATA REGISTER (CRC_DR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
DR [31:16]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DR [15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:0 **Data register bits**

Used as an input register when writing new data into the CRC calculator.
Holds the previous CRC calculation result when it is read.

INDEPENDENT DATA REGISTER (CRC_IDR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved								IDR[7:0]							
								rw	rw	rw	rw	rw	rw	rw	rw

Bits 7:0 **General-purpose 8-bit data register bits**

Can be used as a temporary storage location for one byte.
This register is not affected by CRC resets generated by the RESET bit in the CRC_CR register.

CONTROL REGISTER (CRC_CR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved															RESET
															w

Bits 31:1 Reserved, must be kept at reset value.

Bit 0 **RESET bit**

Resets the CRC calculation unit and sets the data register to 0xFFFF FFFF.
This bit can only be set, it is automatically cleared by hardware.

Figure 27.registers of calculation unit

```
  */
uint32_t HAL_CRC_Accumulate(CRC_HandleTypeDef *hcrc, uint32_t pBuffer[], uint32_t BufferLength)
{
  uint32_t index;      /* CRC input data buffer index */
  uint32_t temp = 0U;  /* CRC output (read from hcrc->Instance->DR register) */

  /* Change CRC peripheral state */
  hcrc->State = HAL_CRC_STATE_BUSY;

  /* Enter Data to the CRC calculator */
  for (index = 0U; index < BufferLength; index++)
  {
    hcrc->Instance->DR = pBuffer[index];
  }
  temp = hcrc->Instance->DR;

  /* Change CRC peripheral state */
  hcrc->State = HAL_CRC_STATE_READY;

  /* Return the CRC computed value */
  return temp;
}
```

```
871
872  /*This verifies the CRC of the given buffer in pData */
873  uint8_t bootloader_verify_crc (uint8_t *pData, uint32_t len, uint32_t crc_host)
874  {
875      /* Reset CRC Calculation Unit */
876        __HAL_CRC_DR_RESET(&hcrc);
877
878      uint32_t CRCValue=0xFF;
879
880      for (uint32_t i=0 ; i < len ; i++)
881      {
882          uint32_t i_data = pData[i];
883          CRCValue = HAL_CRC_Accumulate(&hcrc, &i_data, 1);
884      }
885
886      if( CRCValue == crc_host)
887      {
888          return VERIFY_CRC_SUCCESS;
889      }
890
891      return VERIFY_CRC_FAIL;
892  }
893
```

Figure 28.CRC function

5.6 Challenges in bootloader implementation

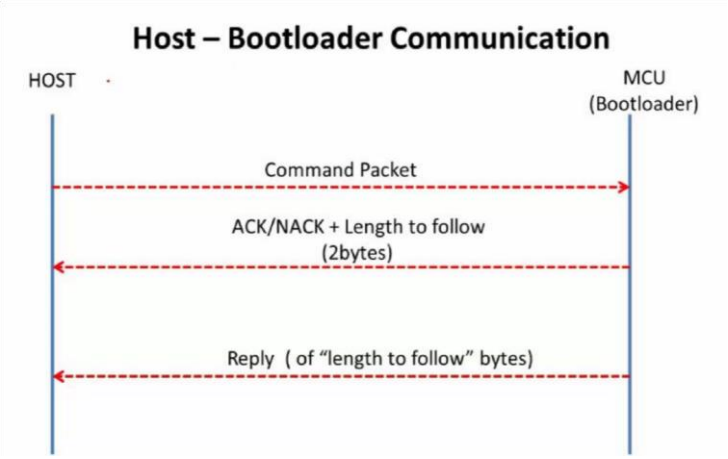
5.6.1 Communication protocol

According to project design boot-loader should receive code packets from

CAN_BUS, to simulate this process we had to use flashing tool, flashing tool function is to simulate the communication way if it is not ready to work, you just plug the MCU to the PC and send data, the MCU then accepts the data as it is using a specific

communication protocol (CAN / UART /……), each communication protocol has a specific flashing tool.

As we are using CAN communication protocol so we need can flashing tool, but the only one we found was produced by VECTOR organization and it was not available for free, so this was a challenge as we cannot test using boot-loader compatible with CAN, to overcome this challenge we made functions and commands suitable for UART protocol and started testing.



Supported Bootloader Commands

Host Sends	Command Code	Bootloader Replies	Notes
BL_GET_VER	0x51	Bootloader version number (1byte)	This command is used to read the bootloader version from the MCU
BL_GO_TO_ADDR	0x55	Success or Error Code (1byte)	This command is used to jump bootloader to specified address.
BL_FLASH_ERASE	0x56	Success or Error Code (1byte)	This command is used to mass erase or sector erase of the user flash .
BL_MEM_WRITE	0x57	Success or Error Code (1byte)	This command is used to write data in to different memories of the MCU

Figure 29.Host-bootloader communication & supported commands

5.6.2 Supported commands

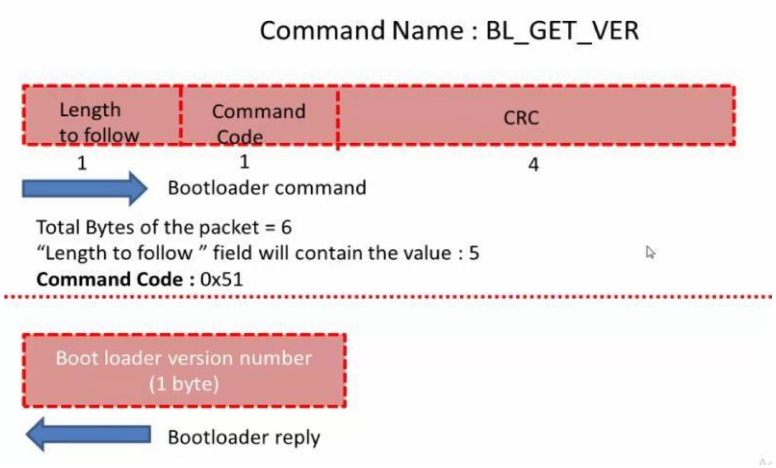


Figure 30.BL_GET_VER frame

In the **BL_GO_TO_ADDR** command we will jump to the specific address of the SRAM, FLASH, or external memory.

It will be as following:

- The host (node MCU\pc) send the length of the following data which be sent.
- The code of the BL_GO_TO_ADDR
- The memory address which we want jump to it.
- The 32-bit CRC of the previous data
- After checking the CRC by the target MCU we will send the ack\nack
- In case of sending the ack then the target sends the status

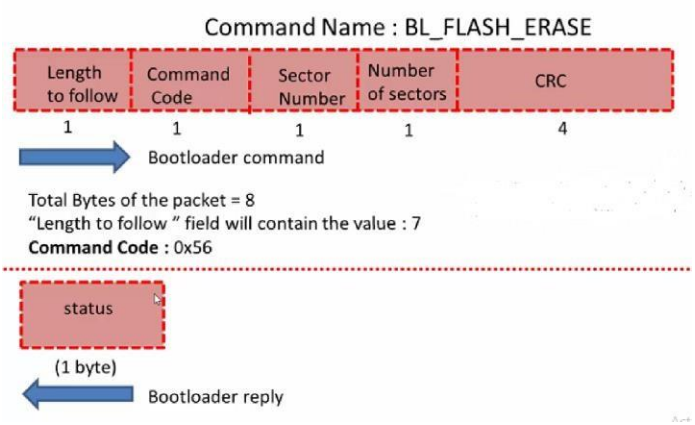


Figure 31.BL_FLASH_ERASE frame

In the **BL_FLASH_ERASE** command we will erase a specific sector of a bank or erasing the whole bank It will be as following:

- The host (node MCU\pc) send the length of the following data which be sent

- ✦ The code of the BL_FLASH_ERASE
- ✦ The sector number which we will start the erasing from it ✦ The no of sectors which we want to erase.
- ✦ The 32-bit CRC of the previous data
- ✦ After checking the CRC by the target MCU we will send the ack\nack ✦ In case of sending the ack then the target sends the status

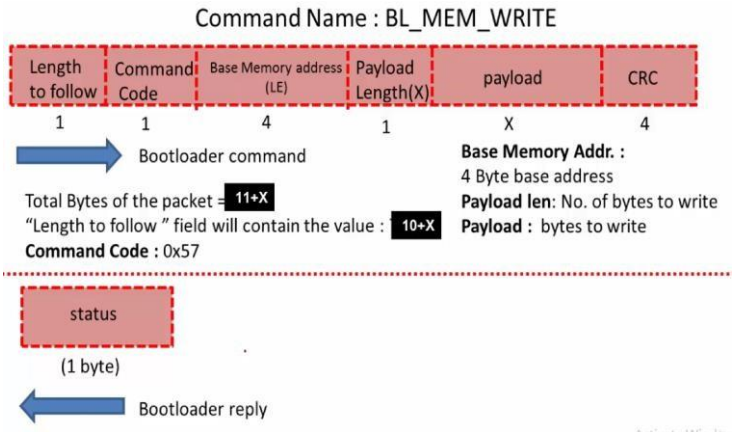


Figure 32.BL_MEM_WRITE frame

In the **BL_MEM_WRITE** command we will write a specific addresses of a bank It will be as following:

- ✦ The host (node MCU\pc) send the length of the following data which be sent
- ✦ The code of the BL_MEM_WRITE
- ✦ The address from which we will start writing.
- ✦ The length of the data which we want to write. ✦ The data which we want to write.
- ✦ The 32-bit CRC of the previous data
- ✦ After checking the CRC by the target MCU we will send the ack\nack ✦ In case of sending the ack then the target sends the status. ✦

- `void bootloader_uart_read_data(void);` this function is to read if there is any packet received by UART, and according to the received command it acts like flash erasing, flashing, or reading from buffer.

- **`void bootloader_uart_write_data (uint8_t *pBuffer, uint8_t len);`** this function writes data to C_UART
- **`bootloader_send_ack (uint8_t command_code, uint8_t follow_len);`** this function sends acknowledge if there is no error.
- **`void bootloader_send_nack(void);`** this function sends acknowledge if there is error.

```c
void bootloader_uart_read_data(void)
{
    uint8_t rcv_len = 0;

    while(1)
    {
        memset(bl_rx_buffer,0,200);
        HAL_UART_Receive(D_C_UART,bl_rx_buffer,1,HAL_MAX_DELAY);
        rcv_len = bl_rx_buffer[0];
        HAL_UART_Receive(D_C_UART,&bl_rx_buffer[1],1,HAL_MAX_DELAY);
        switch(bl_rx_buffer[1])
        {
            case BL_GET_VER:
                bootloader_handle_getver_cmd(bl_rx_buffer);
                break;
            case BL_FLASH_ERASE:
                bootloader_handle_flash_erase_cmd(bl_rx_buffer);
                break;
            case BL_MEM_WRITE:
                bootloader_handle_mem_write_cmd(bl_rx_buffer);
                break;
            default:
                printf("Invalid command received from the host");
                break;
        }

    }
}
```

```c
/*This function sends NACK */
void bootloader_send_nack(void)
{
    uint8_t nack_buf[1];
    nack_buf[0] = BL_ACK;
    HAL_UART_Transmit(D_C_UART,nack_buf,1,HAL_MAX_DELAY);
}
```

```c
/*This function sends ACK if CRC matches along with 'len' to follow*/
void bootloader_send_ack(uint8_t command_code , uint8_t follow_len)
{
    uint8_t ack_buf[2];
    ack_buf[0] = BL_ACK;
    ack_buf[1] = follow_len;
    HAL_UART_Transmit(D_C_UART,ack_buf,2,HAL_MAX_DELAY);
}
```

```c
/* This function writes data in to C_UART */
void bootloader_uart_write_data(uint8_t *pBuffer , uint8_t len)
{
    HAL_UART_Transmit(D_C_UART,pBuffer,len,HAL_MAX_DELAY);
}
```

Figure 33.ack & nack & read & write functions

The previous tasks use commands to make flashing and erasing, commands are functions written to deal with bootloader in easy way.

Erasing command is implemented as following:

```c
/*Helper function to handle BL_FLASH_ERASE command */
void bootloader_handle_flash_erase_cmd(uint8_t *pBuffer)
{
    uint8_t erase_status = 0x00;

    //Total length of the command packet
    uint32_t command_packet_len = bl_rx_buffer[0]+1 ;

    //extract the CRC32 sent by the Host
    uint32_t host_crc = *((uint32_t * ) (bl_rx_buffer+command_packet_len - 4) ) ;

    if (! bootloader_verify_crc(&bl_rx_buffer[0],command_packet_len-4,host_crc))
    {
        printf("checksum success \n");
        bootloader_send_ack(pBuffer[0],1);
        erase_status = execute_flash_erase(pBuffer[2] , pBuffer[3]);
        bootloader_uart_write_data(&erase_status,1);
    }
    else
    {
        printf("checksum fail !!\n");
        bootloader_send_nack();
    }
}
```

Figure 34.handle erase command function

Flashing command is implemented as following:

```
void bootloader_handle_mem_write_cmd(uint8_t *pBuffer)
{
        uint8_t addr_valid = ADDR_VALID;
        uint8_t write_status = 0x00;
        uint8_t chksum =0, len=0;
        len = pBuffer[0];
        uint8_t payload_len = pBuffer[6];
        uint32_t mem_address = *((uint32_t *) ( &pBuffer[2]) );
        chksum = pBuffer[len];
    //Total length of the command packet
        uint32_t command_packet_len = bl_rx_buffer[0]+1 ;
        //extract the CRC32 sent by the Host
        uint32_t host_crc = *((uint32_t * ) (bl_rx_buffer+command_packet_len - 4) ) ;
        if (! bootloader_verify_crc(&bl_rx_buffer[0],command_packet_len-4,host_crc))
        {
        bootloader_send_ack(pBuffer[0],1);
                if( verify_address(mem_address) == ADDR_VALID )
                {
            //execute mem write
            write_status = execute_mem_write(&pBuffer[7],mem_address, payload_len);
            //inform host about the status
            bootloader_uart_write_data(&write_status,1);
                }
                else
                {
            write_status = ADDR_INVALID;
            //inform host that address is invalid
            bootloader_uart_write_data(&write_status,1);
                }
        }
        else
        {
        bootloader_send_nack();
        }
```

Figure 35.handle writing command

6. BANK SWAPPING

6.1. What is Bank Swapping

Bank Swapping is a **memory-switching technique** that uses each bank to store a different version of SW. It is useful for increasing the amount of usable memory beyond the amount directly addressable by processor instructions. It can be used to configure a system differently at different times; for example, if one version is outdated and needs to be updated, we can switch to the other bank, which supposedly has the updated SW version.

Block	Bank	Name	Block base addresses	Size
Main memory	Bank 1	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
		Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
		Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
		Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbyte
		Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
		Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
		Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
		-	-	-
		-	-	-
		-	-	-
		Sector 11	0x080E 0000 - 0x080F FFFF	128 Kbytes
	Bank 2	Sector 12	0x0810 0000 - 0x0810 3FFF	16 Kbytes
		Sector 13	0x0810 4000 - 0x0810 7FFF	16 Kbytes
		Sector 14	0x0810 8000 - 0x0810 BFFF	16 Kbytes
		Sector 15	0x0810 C000 - 0x0810 FFFF	16 Kbytes
		Sector 16	0x0811 0000 - 0x0811 FFFF	64 Kbytes
		Sector 17	0x0812 0000 - 0x0813 FFFF	128 Kbytes
		Sector 18	0x0814 0000 - 0x0815 FFFF	128 Kbytes
			-	-
			-	-
			-	-
		Sector 23	0x081E 0000 - 0x081F FFFF	128 Kbytes
System memory			0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP			0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes	Bank 1		0x1FFF C000 - 0x1FFF C00F	16 bytes
	Bank 2		0x1FFE C000 - 0x1FFE C00F	16 bytes

Figure 36.memory structure

6.2. Bank Swapping approaches

Its default is to boot from **bank 1**, but to boot from **bank 2** we must swap the two banks, this can be achieved using two methods:

- ✦ Switching using **native bootloader**
- ✦ Switching using **user-defined application**

1.Switching using native bootloader:

Set **BFB2** bit in **option bytes** to boot from System memory (Native bootloader) and then it checks if there is a valid code at bank 2 (Sector 12) and then swap the banks and go to sector 0 (bank 2 now), if not it boots from sector 0 but without remapping (Bank 1 by default).

Bit 4	**BFB2**: Dual bank boot 0: Boot from Flash memory bank 1 or system memory depending on boot pin state (Default). 1: Boot always from system memory (Dual bank boot mode).

Figure 37.bfb2 bit modes

2.Switching using user-defined application:

Boot from bank 1 (sector 0 by default) then check for non-volatile variable stored in flash memory or backup register, depending on that variable it decides to remap the banks by using **FB_MODE** in **MEMREM** register.

Table 41. SYSCFG register map and reset values (STM32F42xxx and STM32F43xxx)

Offset	Register	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0x00	SYSCFG_ MEMRMP	Reserved																				SWP_FMC		Reserved	FB_MODE	Reserved					MEM_ MODE		
	Reset value																					0	0		0						x	x	x

Bit 8 **FB_MODE**: Flash Bank mode selection

Set and cleared by software. This bit controls the Flash Bank 1/2 mapping.
0: Flash Bank 1 is mapped at 0x0800 0000 (and aliased at 0x0000 0000) and Flash Bank 2 is mapped at 0x0810 0000 (and aliased at 0x0010 0000)
1: Flash Bank 2 is mapped at 0x0800 0000 (and aliased at 0x0000 0000) and Flash Bank 1 is mapped at 0x0810 0000 (and aliased at 0x0010 0000)

Figure 38.fb_mode description

```
void Start_Active_Bank(void) {

    // disable interrupts
    __disable_irq();
    // disable ART
    CLEAR_BIT(FLASH->ACR, FLASH_ACR_PRFTEN);
    CLEAR_BIT(FLASH->ACR, FLASH_ACR_ICEN | FLASH_ACR_DCEN);
    // clear cache
    SET_BIT(FLASH->ACR, FLASH_ACR_ICRST | FLASH_ACR_DCRST);
    // Don't need to switch vector table as BANK2 effectively becomes BANK1 (so both programs are linked
    //as though they live in BANK1)
    //              SCB->VTOR = (uint32_t)vector2;
    if (Read_Active_Bank_No() == 2) {
        SET_BIT(SYSCFG->MEMRMP, SYSCFG_MEMRMP_UFB_MODE);
    }

    // at this point, the instruction pointer will continue at the same address BUT on the other bank
    // so this initial bit of code HAS to be in the same location (ie offset) on both Banks
    // re-enable ART
    SET_BIT(FLASH->ACR, FLASH_ACR_PRFTEN);
    SET_BIT(FLASH->ACR, FLASH_ACR_ICEN | FLASH_ACR_DCEN);
    // restore interrupt capability
    __enable_irq();

}
```

Figure 39. function to swap banks using FB_MODE

To boot from bank 2, we must remap bank 2 address to bank 1 address because the execution of the code always starts from the address of bank 1. After researching the two methods, we picked method one (using native bootloader), as it is more suitable to our project, giving more functionality and flexibility.

The following diagram shows the flow of the native boot-loader method:

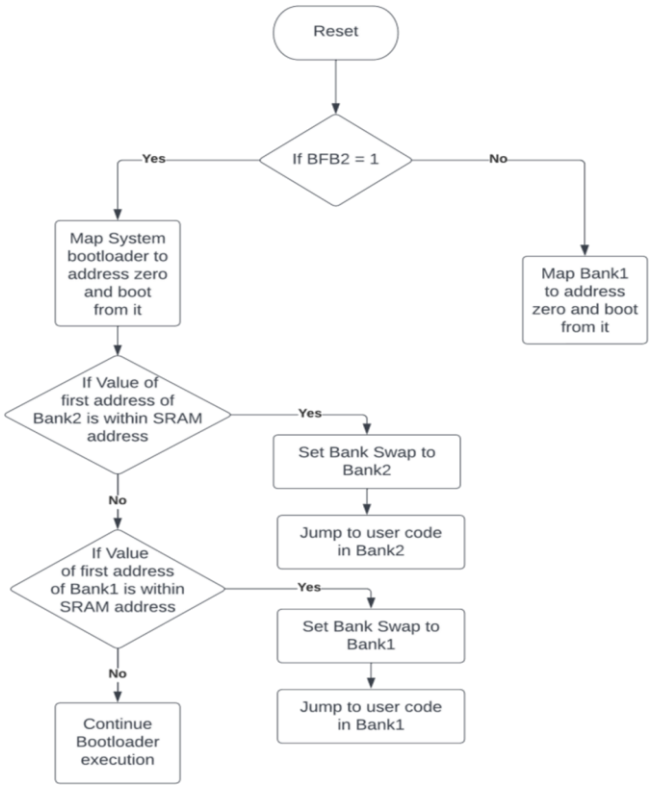


Figure 40.flowchart showing bfb2 bit modes

6.3. Challenges in bank swapping

1. Vector Table handling

✦ Problem statement

The code's default vector table address is at memory address 0, so if we booting from bank 1 its aliased to address 0, in this case if any ISR occur it checks vector table which is currently at address 0 which is correct in this case.

If we are booting from bank 2, system memory bootloader is aliased to address 0 with its vector table, then system bootloader swaps the two banks, and jumps to bank 2, in this case if any ISR occur, it checks vector table at address 0 which is currently system memory bootloader vector table, which is wrong in this case as it accessed a wrong vector table with wrong addresses.

Note: this problem occurs only while using method 1 of bank swapping (Native bootloader).

✦ Solution

The following solution is needed when using bank 2, but we implement it for both banks to make the system more flexible and portable operating on both banks all the same:

Adjust the vector table offset to address **0X08000000**, in this case if any ISR occur, it checks vector table at address 0X08000000 which is necessary for booting from bank 2, but not for booting from bank 1, now booting from any of the two banks will be successful.

2. Hardware limitations

✦ Problem statement

Bank Swapping using stm32f4291 is not fully hardware dependent (depends on SW); The method we are using for bank swapping (Native bootloader) depends system bootloader to do the swapping, which depends on BFB2 bit value (as mentioned above), this makes our system software dependent.

✦ Solution

There's little we can do about this problem using stm32f4291, but to solve this we need to upgrade to a higher board(e.g. STMH7XX), another issue occurred which was the price of the higher boards.

- **STMH7XX**

  The Embedded Flash memory bank 1 and bank 2 can be swapped in the memory map accessed by **AXI interface**, this feature can be used after a firmware update to restart the device on the newly updated firmware, bank swapping is controlled by the **SWAP_BANK** bit of the **FLASH_OPTCR** register.

3. Testing problem

Flash memory area	Flash memory corresponding bank		Start address	End address	Size (bytes)	Region Name
	SWAP_ BANK=0	SWAP_ BANK=1				
User main memory	Bank 1	Bank 2	0x0800 0000	0x0801 FFFF	128 K	Sector 0
			0x0802 0000	0x0803 FFFF	128 K	Sector 1
			...	...	...	...
			0x080E 0000	0x080F FFFF	128 K	Sector 7
	Bank 2	Bank 1	0x0810 0000	0x0811 FFFF	128 K	Sector 0
			0x0812 0000	0x0813 FFFF	128 K	Sector 1
			...	...	...	...
			0x081E 0000	0x081F FFFF	128 K	Sector 7
System memory	Bank 1		0x1FF0 0000	0x1FF1 FFFF	128 K	System Flash memory (bank 1)
	Bank 2		0x1FF4 0000	0x1FF5 FFFF	128 K	System Flash memory (bank 2)

Figure 41.stm32h7 memory structure

✦ Problem Statement

While we were testing toggling of the BFB2 bit, a problem occurred, where if any variation happens from the algorithm used for toggling the BFB2 bit, read/write protection will be enabled (protection level 1), and flash memory won't be accessible.

Table 16. Description of the option bytes (STM32F42xxx and STM32F43xxx)

Option bytes (word, address 0x1FFF C000)	
RDP: *Read protection option byte.* The read protection is used to protect the software code stored in Flash memory.	
Bit 15:8	0xAA: Level 0, no protection 0xCC: Level 2, chip protection (debug and boot from RAM features disabled) Others: Level 1, read protection of memories (debug features limited)
USER: User option byte This byte is used to configure the following features:	

Figure 42.protection levels

Level 1 Protection

At level 1 protection, we have no access to read/write on the flash memory, and we can return from level 1 to level 0, but we can't return from level 2 to any of the previous levels.

✦ Solution

We use **STM32 ST-LINK Utility** to reset the option byte values (return to level 0 protection); Thus, unlocking the read/write on the flash memory.

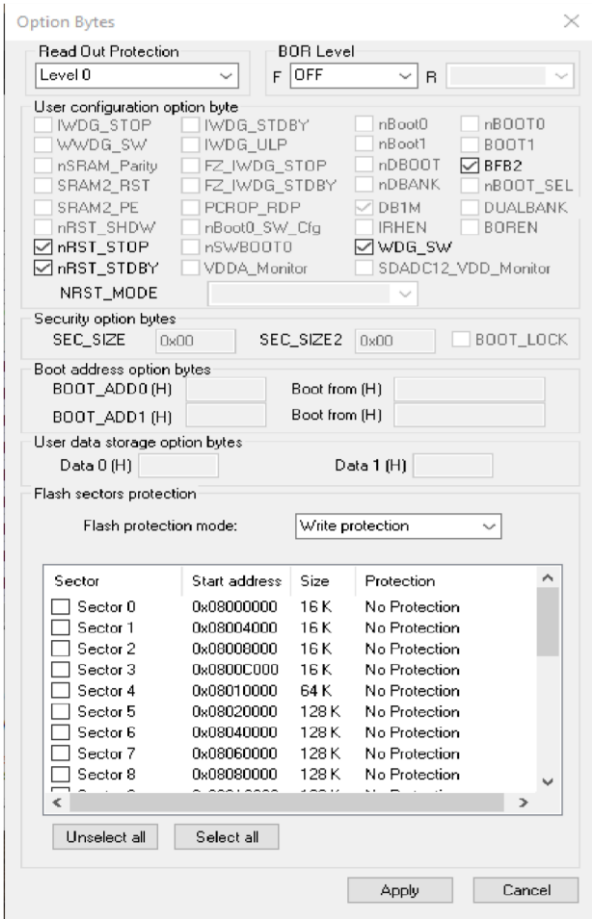


Figure 43.st-link utility

4.Building process problem

✦ Problem Statement

While burning the **.HEX/.ELF** files using STM32 ST-LINK Utility, the start address of the code is specified at the linker file, which means all the **.HEX/.ELF** files can be flashed at the specified start address only.

(This problem occurs only while using the board debugger to burn the code).

✦ **Solution**

We use the .bin file instead of **.HEX/.ELF** files, so we can flash the code in any sector of memory instead of a specified address.

✦ **.HEX files**

.hex files are Ascii files written using the IntexHex file format and is used for putting code into memory of microcontrollers. This file type contains metadata about where to load the file, the length of the file, the type of stuff being loaded, etc. This makes it possible to do more things with .hex file than is possible with a .bin file. Also, when putting a .hex file onto a microcontroller you do not have to specify where to load the code, because the address is specified as part of the file.

✦ **.bin files**

.bin files are straight binary data (and therefore require a special editor to edit / make sense of), when loading them onto a microcontroller you must specify the address to load the data to.

6.4. Implementation

Steps used to swap the two banks:

- ✦ Unlock Flash.
- ✦ Unlock option bytes.
- ✦ Use a declared structure to get the current option bytes value.
- ✦ Check for BFB2 bit if it’s enabled then disable it or if it’s disabled the enabled it.
- ✦ Flash program using the modified structure.
- ✦ Flash option bytes launch.
- ✦ Lock options bytes.
- ✦ Lock Flash.
- ✦ Software system reset.

The approach used to switch the two banks is based on toggling BFB2 bit and then manually resetting the MCU, In order to edit the register that contains this bit we must first unlock flash registers access this is done by putting the following keys in

FLASH KEY register key1(0x45670123U) then key2(0xCDEF89ABU), then we unlock option byte register programing by putting the following keys in FLASH OPTKEYR register key1(0x08192A3BU) then key2 (0x08192A3BU), now we can get option byte value and modify it according to our needs, after finishing option byte programing we must again lock flash and option byte register and then manually resetting the MCU so the changes can take effect.

Keep in mind that these steps need to be executed in that exact order otherwise the MCU will activate read/write protection level 1, and this can be solved as discussed before.

Function used to switch between the two banks.

```
337 void toggleBankAndReset() {
338     FLASH_AdvOBProgramInitTypeDef OBInit;
339     HAL_FLASH_Unlock();
340     //__HAL_FLASH_CLEAR_FLAG(FLASH_FLAG_OPTVERR);
341     HAL_FLASH_OB_Unlock();
342     HAL_FLASHEx_AdvOBGetConfig(&OBInit);
343     OBInit.OptionType = OPTIONBYTE_BOOTCONFIG;
344 
345     if (((OBInit.BootConfig) & (OB_DUAL_BOOT_ENABLE)) == OB_DUAL_BOOT_ENABLE) {
346         OBInit.BootConfig = OB_DUAL_BOOT_DISABLE;
347     } else {
348         OBInit.BootConfig = OB_DUAL_BOOT_ENABLE;
349     }
350     if (HAL_FLASHEx_AdvOBProgram(&OBInit) != HAL_OK) {
351         uint32_t errorCode = HAL_FLASH_GetError();
352         while (1) {
353             HAL_Delay(1000);
354             HAL_GPIO_TogglePin(GPIOG, GPIO_PIN_14);
355         }
356     }
357     if (HAL_FLASH_OB_Launch() != HAL_OK) {
358         uint32_t errorCode = HAL_FLASH_GetError();
359         while (1) {
360             HAL_Delay(100);
361             HAL_GPIO_TogglePin(GPIOG, GPIO_PIN_14);
362         }
363     }
364     HAL_FLASH_OB_Lock();
365     HAL_FLASH_Lock();
366     HAL_NVIC_SystemReset();
367 
368 }
```

Figure 44.funtion to switch two banks using bfb2 bit.

7. CAN Initiation

7.1. Identifier Filtering for STM32F429I

- 28 configurable and scalable filter banks (27-0)
- Each filter bank x consists of two 32-bit registers, **CANFxR0** and **CANFxR1.**
- Each filter bank can be scaled independently:
- One 32-bit filter for **STDID [10:0]**, **EXTID [17:0]**, **IDE** and **RTR** bits.
- Two 16-bit filters for **STDID [10:0], IDE, RTR** and **EXTID [17:15]**.

7.2. Loop back mode test for STM32

A loopback test is the process of sending digital data streams from a source back to the same point without any intentional modifications. It's generally performed to determine whether the ECU with the hardware components work properly, and it could send and receive messages successfully.

To be independent of external events, CAN Core ignores acknowledge errors (no dominant bit sampled in the acknowledge slot of a data / remote frame) in Loop Back Mode.

In this mode, the **bxCAN** performs feedback from its **TX** output to its **RX** input.

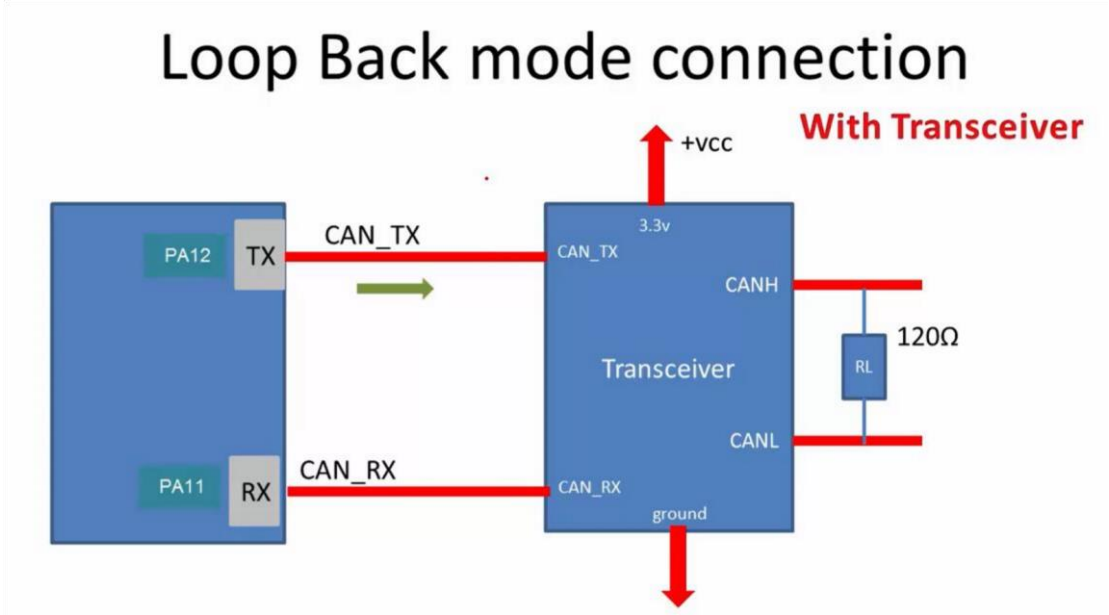


Figure 45.loop back mode connection

7.3. Transmission steps

✦ First, we should include the HAL module driver, #include "stm32f4xx_hal.h" to call CAN functions is the HAL (Hardware Abstraction Layer) ✦ Start the CAN module.

- By calling the function: `HAL_CAN_Start(&hcan);` ✦ Enable CAN interrupts.
- To get an interrupt when a message is received.
- By calling the function:

`HAL_CAN_ActivateNotification(&hcan, CAN_IT_RX_FIFO0_MSG_PENDING);`

✦ **TxHeader** Initialization

- Data Length Code with 1 byte
- Specify that this is a Data Frame
- Standard ID with 0x103 ✦ Specify the data to be sent **0xf3**.

✦ Add a message to the first free Tx mailbox and activate the corresponding transmission request.

```
HAL_CAN_Start(&hcan);

HAL_CAN_ActivateNotification(&hcan, CAN_IT_RX_FIFO0_MSG_PENDING);

TxHeader.DLC = 1;
TxHeader.ExtId = 0;
TxHeader.IDE = CAN_ID_STD;
TxHeader.RTR = CAN_RTR_DATA;
TxHeader.StdId = 0x103;
TxHeader.TransmitGlobalTime = DISABLE;

TxData[0] = 0xf3;
HAL_CAN_AddTxMessage(&hcan, &TxHeader, TxData, &TxMailbox);
```

Figure 46. TxHeader initialization and adding Message on CAN bus

7.4. Reception Steps

✦ CAN filter configuration:

- Activate filter.
- Specify the filter bank that will be initialized.
- Filter on ID: **0x103**
- Note: This is the ID of the transmitted message to make sure that this message passes the filtered and get received successfully.
- Specify the mask with the value of the **ID(0x103)** to make sure that the ID of the received message must match the filter ID.

- Specify filter mode (Mask Mode)
- Specify filter scale (One 32-bit filter)

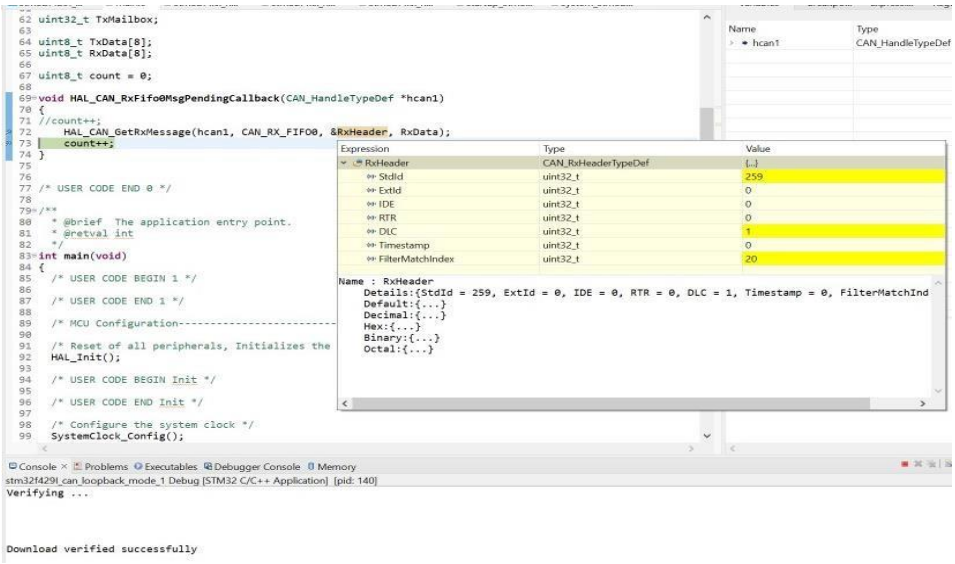
Call HAL_CAN_ConfigFilter () and pass to it the structure of the filter configurations.

```
CAN_FilterTypeDef canfilterconfig;

canfilterconfig.FilterActivation = CAN_FILTER_ENABLE;
canfilterconfig.FilterBank = 10;  // anything between 0 to SlaveStartFilterBank
canfilterconfig.FilterFIFOAssignment = CAN_RX_FIFO0;
canfilterconfig.FilterIdHigh = 0x103<<5;
canfilterconfig.FilterIdLow = 0x0000;
canfilterconfig.FilterMaskIdHigh = 0x103<<5;
canfilterconfig.FilterMaskIdLow = 0x0000;
canfilterconfig.FilterMode = CAN_FILTERMODE_IDMASK;
canfilterconfig.FilterScale = CAN_FILTERSCALE_32BIT;
canfilterconfig.SlaveStartFilterBank = 13;  // 13 to 27 are assigned to slave CAN (CAN 2) OR 0 to 12 are assgned to CAN1

HAL_CAN_ConfigFilter(&hcan, &canfilterconfig);
```

Figure 47. Filter configurations

To guarantee the success of the transmission and reception, we will put a breakpoint at the message pending call back function:

- As we have activated the notification for the message pending interrupt, the callback function will be called if and only if the transmitted message ID matches the ID configured for the filter.
- When the break point is hit, this will help us to see the details of the received message to make sure that it's received successfully as shown in the following screenshot.



Expression	Type	Value
RxData	uint8_t [8]	0x2000005c <RxData>
RxData[0]	uint8_t	243 '6'
RxData[1]	uint8_t	0 '\0'
RxData[2]	uint8_t	0 '\0'
RxData[3]	uint8_t	0 '\0'
RxData[4]	uint8_t	0 '\0'
RxData[5]	uint8_t	0 '\0'
RxData[6]	uint8_t	0 '\0'
RxData[7]	uint8_t	0 '\0'

Figure 48. Frame received successfully

As shown in the two previous screenshots, the breakpoint was hit and the ID of the received message is 259 (decimal) that equals **0x103**, and the data received is 243 (decimal) that equals **0xf3**, which are the same ID and data of the message we sent before.

Message transmission and reception is successfully done.

7.5. Normal Mode between STM32 and ESP32

After initialization, the software must request the hardware to enter Normal mode to be able to synchronize on the CAN bus and start reception and transmission. The **bxCAN** enters Normal mode when it has synchronized with the data transfer on the CAN bus.

Configurations of ESP32

- TX: **GPIO pin 5**
- RX: **GPIO pin 4**
- Baud rate: **500Kbps**

Configurations of STM3232F429I

- TX: **PINA12**
- RX: **PINA11**
- Baud rate: **500Kbps**

STM32 Set up (To Send)

- Start the CAN module.
- Initialization of TxHeader:
- Data Length Code with 2 bytes
- Data Frame
- ID: **0x446**
- Add a message to the first free TX mailbox and activate the corresponding transmission request.

```
HAL_CAN_Start(&hcan1);

  // Activate the notification
  HAL_CAN_ActivateNotification(&hcan1, CAN_IT_RX_FIFO0_MSG_PENDING);


  TxHeader.DLC = 2;  // data length
  TxHeader.IDE = CAN_ID_STD;
  TxHeader.RTR = CAN_RTR_DATA;
  TxHeader.StdId = 0x446;  // ID

  HAL_CAN_AddTxMessage(&hcan1, &TxHeader, TxData, &TxMailbox);
```

Figure 49. TxHeader initialization and adding Message on CAN bus

ESP32 Set up (To receive)

- ✦ If the message is transmitted successfully, a printed message containing the verification, ID, format, and length appears on the monitor window.
- ✦ If unsuccessful, a "**Failed to receive**" message is printed.

```
can_message_t messagerx;
if (can_receive(&messagerx, pdMS_TO_TICKS(10000)) == ESP_OK) {
    printf("Message received\n");
} else {
    printf("Failed to receive message\n");
    return;
}


//Process received message
if (messagerx.extd) {
    printf("Message is in Extended Format\n");
} else {
    printf("Message is in Standard Format\n");
}
printf("ID is %d\n", messagerx.identifier);
if (!(messagerx.rtr)) {
    for (int i = 0; i < messagerx.data_length_code; i++) {
        printf("Data byte %d = %d\n", i, message.data[i]);
    }
}

}
```

Figure 50. esp32 receive frame successfully from stm32

Verification of Reception

- A message (with format, ID and length configured) is transmitted to the receiving microcontroller.

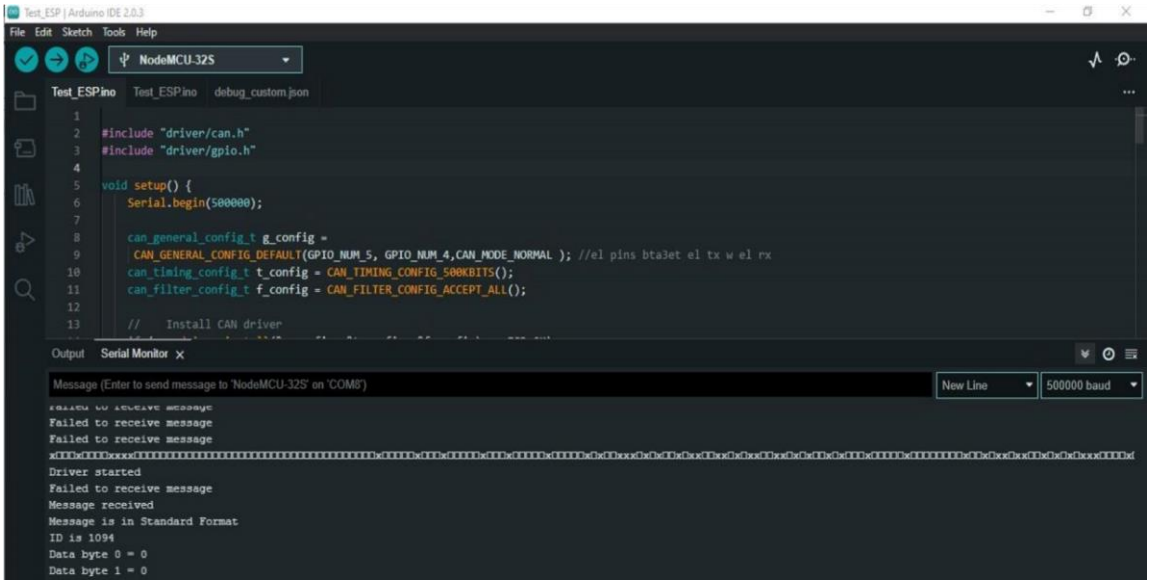


Figure 51. esp32 receive frame code

As shown, The ID of the received message is 1094 (decimal), this equals **0x446**, which is the same ID of the message sent by STM32.

Communication between STM32 and ESP32 is established successfully.

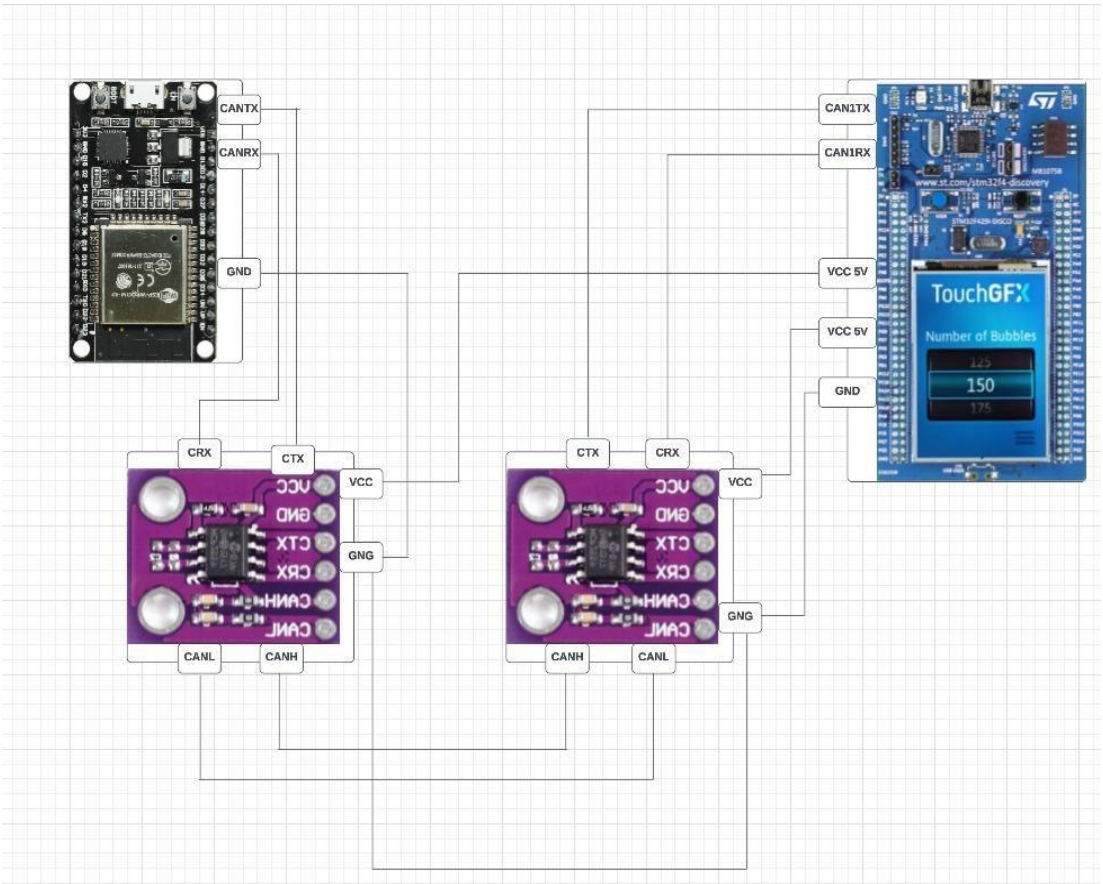
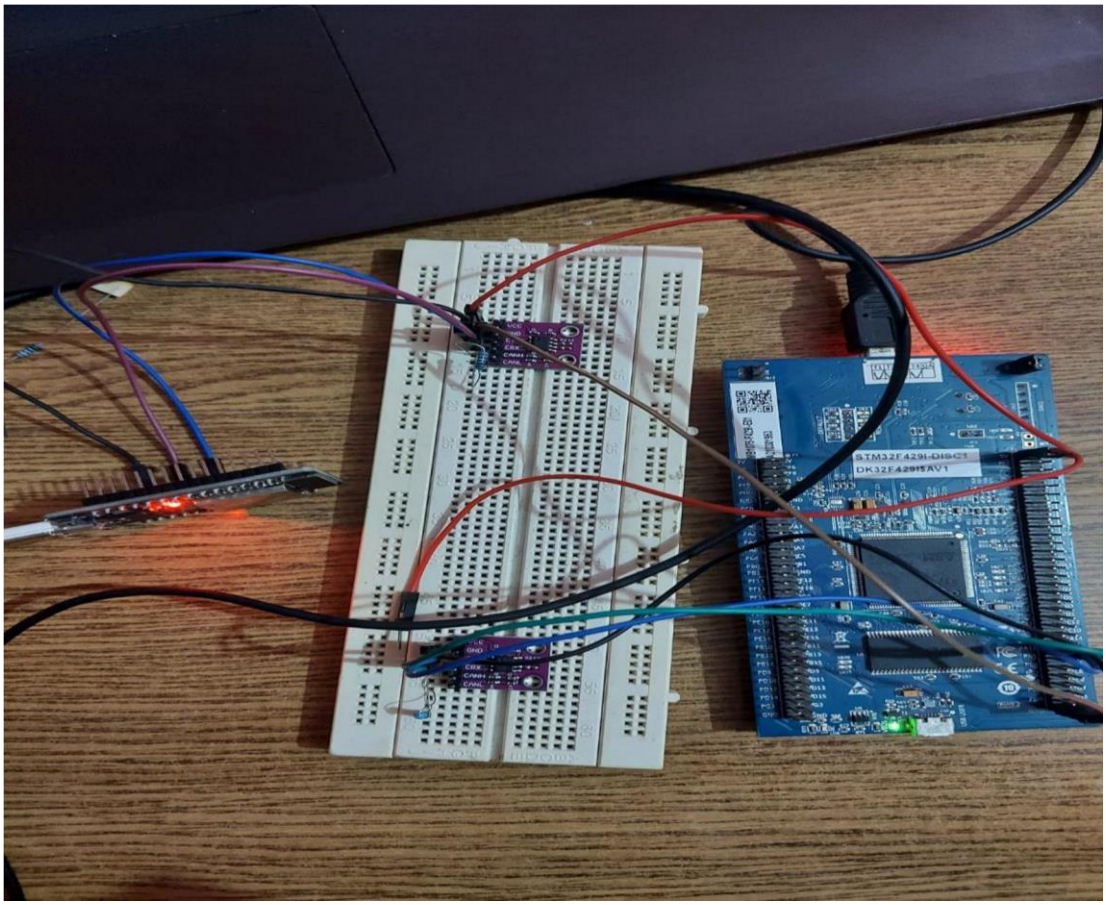


Figure 52. communication between telematics unit (esp32) and Target (STM32)



8. REAL-TIME OPERATING SYSTEM (RTOS)

8.1. Why use a Real-time operating system?

- to ensure the embedded system operates in an efficient and reliable manner by managing hardware and software resources.
- to provide an abstraction layer to simplify the process of developing higher layers of software.
- to act as a partitioning tool.

The embedded OS (operating system) achieves these functions via a **kernel** that includes: process management, memory management, and I/O system management components.

8.2. Popular RTOS

- **Deos** (DDC-I).
- **FreeRTOS** (Amazon).
- **Nucleus** (Mentor Graphics).

We picked **FreeRTOS** as it's free, and has the most online resources available, we also studied FreeRTOS in college, which gave us prior experience.

FreeRTOS in our project

In this project, the OS will have 2 main tasks, task 1 being the main task, task 2 will be OTA task, task 1 should be as simple as a LED toggle task, the OTA task will be responsible for receiving the new application code from Node MCU using CAN communication protocol, and then flash it on the inactive bank.

8.3. Implementation of FREERTOS

OS code generation

We implemented the OS to be located at Sector 3.

We modified the Vector Table accordingly.

```
/* Memories definition */
MEMORY
{
  CCMRAM    (xrw)    : ORIGIN = 0x10000000,   LENGTH = 64K
  RAM    (xrw)    : ORIGIN = 0x20000000,   LENGTH = 192K
  FLASH    (rx)    : ORIGIN = 0x800C000,   LENGTH = 2048K
}
```

Figure 53.linker file of OS application

OS code test

Created a task to toggle the board's LED.

We used RTC backup registers to allow jumping to user code or staying in the bootloader.

```
void LED_Test(void *pvparameters) {
    TickType_t xDelay = 1000;
    for (;;) {
        HAL_GPIO_WritePin(GPIOG, GPIO_PIN_14, GPIO_PIN_RESET);
        vTaskDelay(xDelay);
        HAL_GPIO_WritePin(GPIOG, GPIO_PIN_14, GPIO_PIN_SET);
        vTaskDelay(xDelay);

        // Write Back Up Register 1 Data
        HAL_PWR_EnableBkUpAccess();
        //Writes a data in a RTC Backup data Register 1
        HAL_RTCEx_BKUPWrite(&hrtc, RTC_BKP_DR1, 0xBEBE);
        HAL_PWR_DisableBkUpAccess();
    }
}
```

Figure 54.simple led task in freeRTOS

We created a simple LED toggle task to test our operating system, **later in the project an OTA task should be added**

9. References

1) https://chat.openai.com/

2) https://www.udemy.com/course/stm32f4-arm-cortex-mx-custom-bootloader-development/

3) https://www.st.com/en/evaluation-tools/32f429idiscovery.html#documentation

4)https://community.st.com/s/question/0D53W00001XuYBgSAN/with-bfb2-set-on-an-stm32f429-ie-the-rom-bootloader-runs-before-the-application-code-it-is-not-possible-to-start-an-application-that-runs-from-the-flash-alias-at-address-0-that-works-fine-with-bfb20

5)https://community.st.com/s/question/0D50X0000ALvG9LSQV/using-hal-ll-to-set-bfb2-bit-on-stm32l4xx-help-needed

6)https://community.st.com/s/question/0D50X0000A4qcSDSQY/stm32f427-flash-memory-bank-swapping

7) https://e2e.ti.com/

8) https://copperhilltech.com/

9) https://elearning.vector.com/

10)https://www.youtube.com/watch?v=qzoLA5wlHVg&list=PLPFp2zIiTnxWaZioZMFrb2Cw1wiacD_oT/

11) https://www.csselectronics.com/