Bootloader Design with The FOTA Feature

Intake	3
Branch	Smart Village
Group	2
Names	Ahmed Ibrahim Almorsy
	Omar Adel Abou Zaid
	Maisara Rashad Abd El Karem
	Moustafa Mohamed Khalil
	Mahmoud Salah Eldin Soliman

Contents

1 Bootloader Design

1.1 Introduction

In embedded systems, a bootloader (often simply referred to as a “boot loader”) is a small program or piece of code that resides in the non-volatile memory of a microcontroller or microprocessor. Its primary function is to initialize the system and load the main application program (firmware) into the system’s RAM or flash memory for execution. Here are the key functions and characteristics of a bootloader in embedded systems:

Initialization: The bootloader is the first code that runs when the embedded system is powered on or reset. Its main task is to initialize the hardware components, set up the necessary peripherals, and prepare the system for the main application.

Loading Firmware: The bootloader’s most critical function is to load the main application firmware into memory. This firmware may be stored in external flash memory, EEPROM, or another non-volatile storage medium. The bootloader reads the firmware from storage and writes it to the appropriate location in RAM or flash memory.

Boot Mode Selection: Some bootloaders provide mechanisms for selecting different boot modes or configurations, which can be useful for tasks such as firmware updates or switching between different application programs.

Communication Interface: Bootloaders often include communication interfaces that allow for firmware updates or debugging. Common communication interfaces include UART (serial), USB, Ethernet, or even wireless connections like Bluetooth or Wi-Fi.

Security: Many embedded systems bootloaders incorporate security features to ensure that only authorized firmware can be loaded onto the device. This can include digital signatures, encryption, or secure boot processes.

Error Handling: Bootloaders typically include error-checking mechanisms to verify the integrity of the firmware image before loading it. If an error is detected, the bootloader may take appropriate action, such as rejecting the firmware or initiating recovery procedures.

Flexibility and Customization: Bootloaders can be customized to meet the specific requirements of the embedded system. This customization might involve adapting the bootloader to work with different hardware configurations or supporting specific communication protocols.

1.2 Type of Programming

1.2.1 OFF-Circuit Programming:

Off-circuit programming, as the name suggests, involves programming a microcontroller or embedded device outside of its intended system or circuit. The device is removed from the target system and connected to a separate programmer or flasher.

Key points about OCP:

- Target Device is Removed:
  The target microcontroller or embedded device is physically removed from the circuit or system and connected to a dedicated programmer.
- Advantages:
  Faster programming speeds compared to ICP.
  Well-suited for mass production, as devices can be programmed in parallel.
- Disadvantages:
  - Requires physical access to the device for removal.
  - May not be suitable for systems where removing the device is difficult or not feasible. In our case, the car contains hundreds of ECUs, if off-circuit is used, we need to remove the MCU from the ECU that we want to program from the vehicle, and this is not a good approach to follow.

1.2.2 In-Circuit Programming

In-circuit programming, also known as in-system programming (ISP), is a method of programming embedded systems while the target device is connected to the system or circuit it is intended to operate in. This method allows you to program the device without removing it from the circuit or the system it is a part of. Processor is halted during the flashing.

Key points about ICP:

- **Target Device is In-Circuit**: The target microcontroller or embedded device remains connected to the circuit and is powered during the programming process.
- **Advantages**:
  - No need to physically remove the device, making it suitable for devices that are permanently soldered onto a PCB (Printed Circuit Board).
  - Simplifies the programming process for devices that are difficult to access or are part of a larger system.
- **Disadvantages**:
  - May require additional circuitry to support programming.
  - The programming speed might be slower than off-circuit programming.

- **Examples**:
  JTAG, SWD (Serial Wire Debug), SPI, and UART are commonly used interfaces for in-circuit programming.

1.2.3 In Application Programming

The processor has the responsibility to communicate with the flash driver in the MCU to flash the Firmware.

The code responsible for flashing that Firmware is the bootloader application.

The bootloader is responsible for flashing the firmware by any communication protocols.

In the Automotive Industry, the vehicle contains hundreds of ECUs and If In-Circuit Programming is used we need several sockets equivalent to the number of ECUs which increase the complexity of the system and increase the cost.

The solution is to use only one socket to program all ECUs with only one Communication Protocol. The Bootloader Application will receive the Firmware with this socket and start communicating with the flash driver inside the ECU.

- **Advantages**:
  - The ECUs will be inside the vehicle and will not be removed during the programming.
  - Using of standardized communication protocol will decrease the complexity of the system and decrease the overall cost.

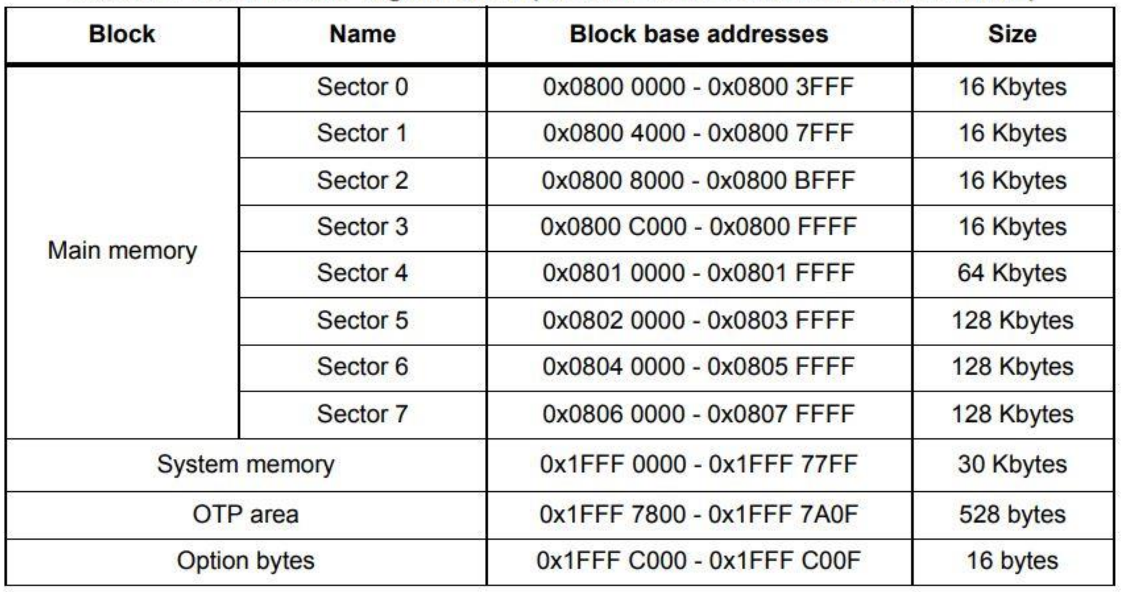
1.3 Flash Memory Layout

Embedded Flash memory in STM32F401xB/C and STM32F401xD/E

The Flash memory has the following main features:

- Capacity up to 256 Kbytes for STM32F401xB/C and up to 512 KBytes for STM32F401xD/E
- 128 bits wide data read.
- Byte, half-word, word and double-word write.
- Sector and mass erase
- Memory organization
  The Flash memory is organized as follows:
  - A main memory block divided into 4 sectors of 16 KBytes, 1 sector of 64 KBytes, 1 sector of 128 KBytes (STM32F401xB/C) or 3 sectors of 128 Kbytes (STM32F401xD/E)
  - System memory from which the device boots in System memory boot mode
  - 512 OTP (one-time programmable) bytes for user data
    The OTP area contains 16 additional bytes used to lock the corresponding OTP data block.
  - Option bytes to configure read and write protection, BOR level, watchdog software/hardware and reset when the device is in Standby or Stop mode.

Table 5. Flash module organization (STM32F401xB/C and STM32F401xD/E)



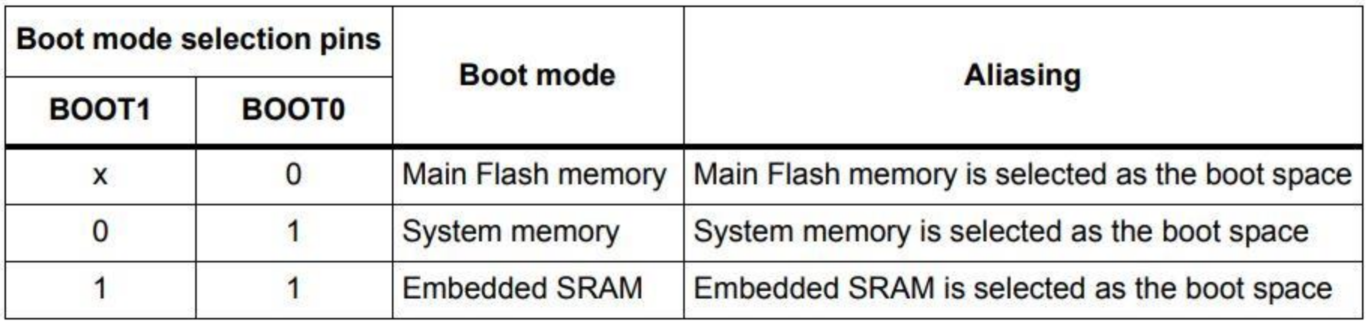
Block	Name	Block base addresses	Size
Main memory	Sector 0	0x0800 0000 - 0x0800 3FFF	16 Kbytes
	Sector 1	0x0800 4000 - 0x0800 7FFF	16 Kbytes
	Sector 2	0x0800 8000 - 0x0800 BFFF	16 Kbytes
	Sector 3	0x0800 C000 - 0x0800 FFFF	16 Kbytes
	Sector 4	0x0801 0000 - 0x0801 FFFF	64 Kbytes
	Sector 5	0x0802 0000 - 0x0803 FFFF	128 Kbytes
	Sector 6	0x0804 0000 - 0x0805 FFFF	128 Kbytes
	Sector 7	0x0806 0000 - 0x0807 FFFF	128 Kbytes
System memory		0x1FFF 0000 - 0x1FFF 77FF	30 Kbytes
OTP area		0x1FFF 7800 - 0x1FFF 7A0F	528 bytes
Option bytes		0x1FFF C000 - 0x1FFF C00F	16 bytes

In our MCU, STM32F401CCU6 is used.

1.4 Boot Configuration Modes

STM32F4xx microcontrollers implement a special mechanism to be able to boot from other memories (like the internal SRAM).

In the STM32F4xx, three different boot modes can be selected through the BOOT [1:0] pins as shown in the below figure.



Boot mode selection pins		Boot mode	Aliasing
BOOT1	BOOT0		
x	0	Main Flash memory	Main Flash memory is selected as the boot space
0	1	System memory	System memory is selected as the boot space
1	1	Embedded SRAM	Embedded SRAM is selected as the boot space

System Memory area used for **STM Native Bootloader**.

From the 2-Pins, we can control from which area the processor will fetch the first instruction.

Note: When the device boots from SRAM, in the application initialization code, you must relocate the vector table in SRAM using the NVIC exception table and the offset register.

1.5 Bootloader Features

- We have a flash memory with 256 Kbytes.
- Bootloader Code area from Sector 0 to Sector 1.
- Application Code area from Sector 2 and Sector 5.
- Bootloader is flashed in the STM32F401CC MCU and the Host will be Raspberry pi 3 that will communicate with the STM32 with package of commands.
- The Bootloader in STM32 MCU will process the command sent by the Host and then reply to the host with ACK or NOT_ACK.
- The Host can send non valid command, and the bootloader in this case, will send NOT_ACK to the Host.
- Bootloader Provided Commands:

  Host can send Commands to:

  1- Read the Bootloader Version From the MCU
  2- Read the supported Command from the MCU
  3- Read Chip Identification Number From the MCU
  4- Read the Protection Level of the MCU
  5- Go To Application Code to start the Vehicle.
  6- Flash Erase of the MCU with start sector and number of sectors to be erased inside the MCU.
  7- Write to specific address inside the flash of the MCU, and to flash the application binary code starting from specified address.
  8- Change Flash Protection Level of the MCU

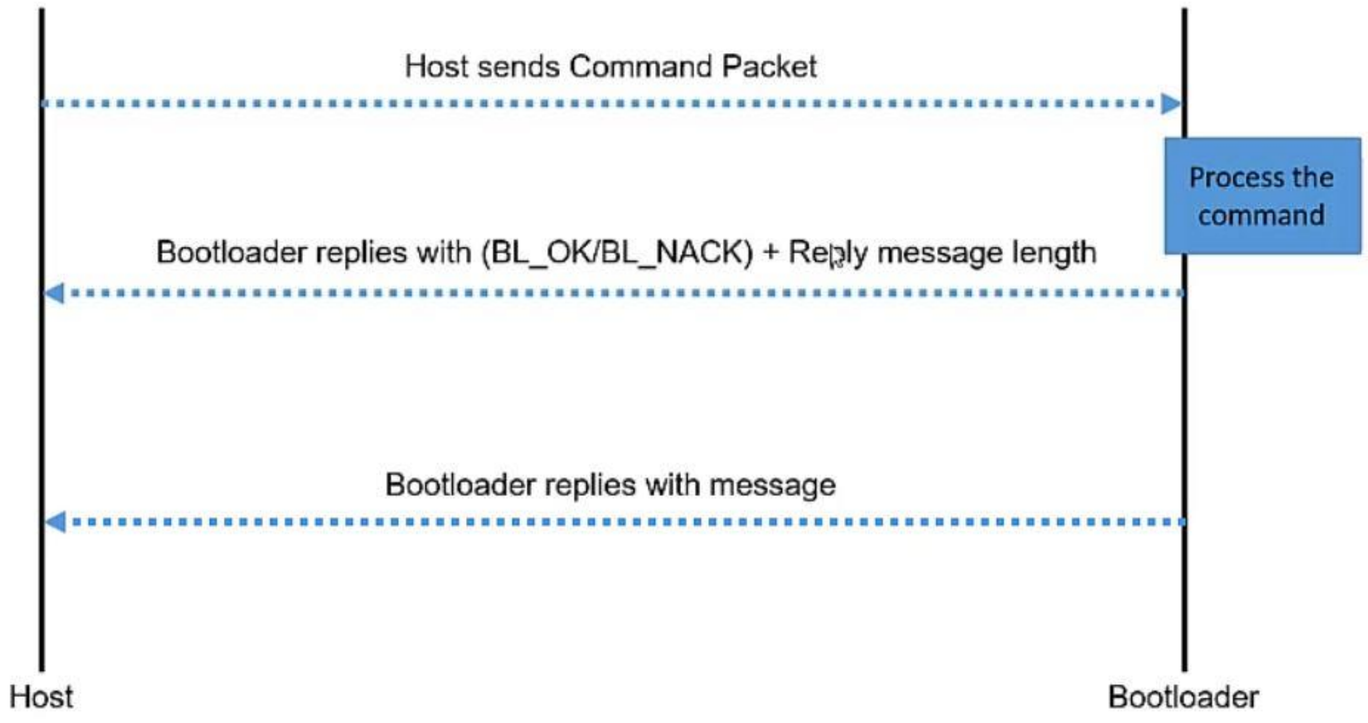
1.6 Bootloader User Guide

To connect to the Bootloader, the host which is **raspberry pi 3** is connected with the **STM32 MCU** through the **UART Protocol**.

The host can send multiple of commands to the Bootloader which will be discussed later in this chapter.

The host is a python script code that will handle all the sending and receiving from the MCU.

1.7 Host and Bootloader Communication Methodology


Host sends Command Packet
Process the command
Bootloader replies with (BL_OK/BL_NACK) + Reply message length
Bootloader replies with message
Host
Bootloader

1- Host will send to the Bootloader a command packet with specified Bytes.

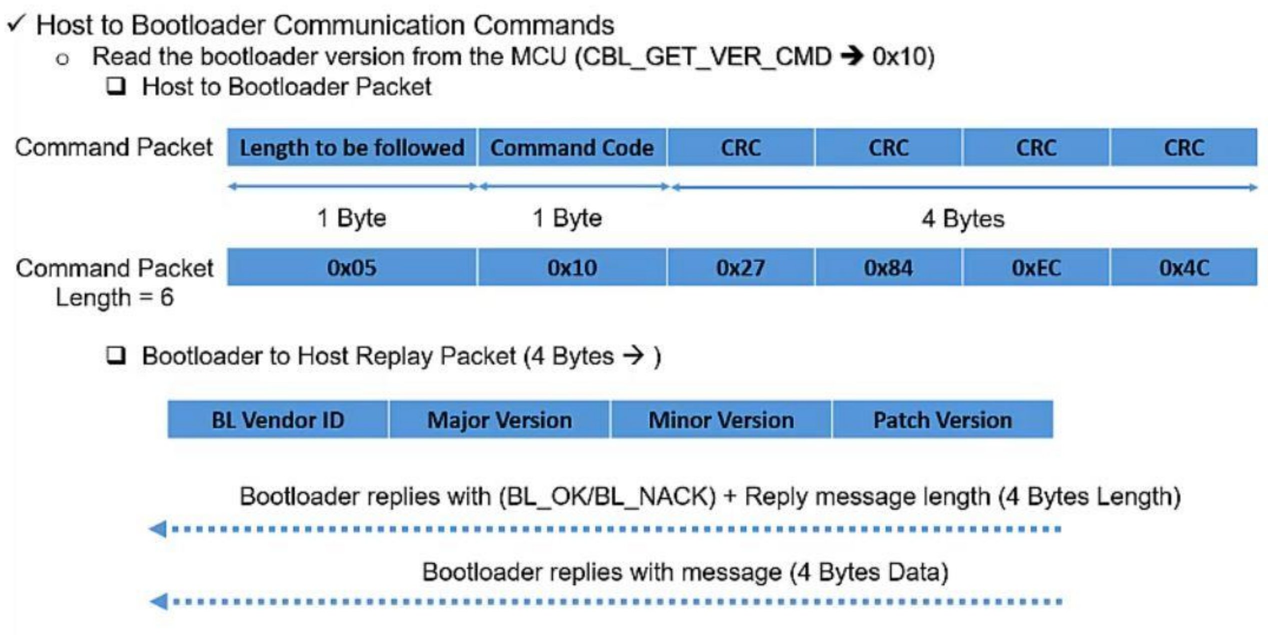
2- Bootloader will Process this Packet and send:

- ACK with the replied message length or
- NACK if the packed is not supported.

3- In case of ACK, Bootloader will send the message with the specified length.

1.7.1 Host and Bootloader Command Packets:

1.7.1.1 Read the Bootloader Version Command Packet


✓ Host to Bootloader Communication Commands
o Read the bootloader version from the MCU (CBL_GET_VER_CMD ➔ 0x10)
❑ Host to Bootloader Packet
Command Packet
Length to be followed
Command Code
CRC
CRC
CRC
CRC
1 Byte
1 Byte
4 Bytes
Command Packet Length = 6
0x05
0x10
0x27
0x84
0xEC
0x4C
❑ Bootloader to Host Replay Packet (4 Bytes →)
BL Vendor ID
Major Version
Minor Version
Patch Version
Bootloader replies with (BL_OK/BL_NACK) + Reply message length (4 Bytes Length)
Bootloader replies with message (4 Bytes Data)

The Host need to read the version of the Bootloader as example:

- Bootloader Vendor ID
- Bootloader Major Version
- Bootloader Minor Version
- Bootloader Patch Version

The command Packet that will be sent by the **Host** is 6 Bytes that contains:

1- Length of the Packet that will be sent to the Bootloader.
2- Command Code ID. Every Command has its own ID that will be processed by the Bootloader.
3- The Host will calculate the CRC and sent it to the bootloader with 4Bytes length.

The Bootloader will receive this Command Packet and Parse it and calculate the CRC to the first 2 Bytes and compare between the CRC sent by the Host and its own calculated CRC to ensure that the 2-CRCs are verified.

If the 2-CRC are the same, The Bootloader will send 2 Bytes which are:

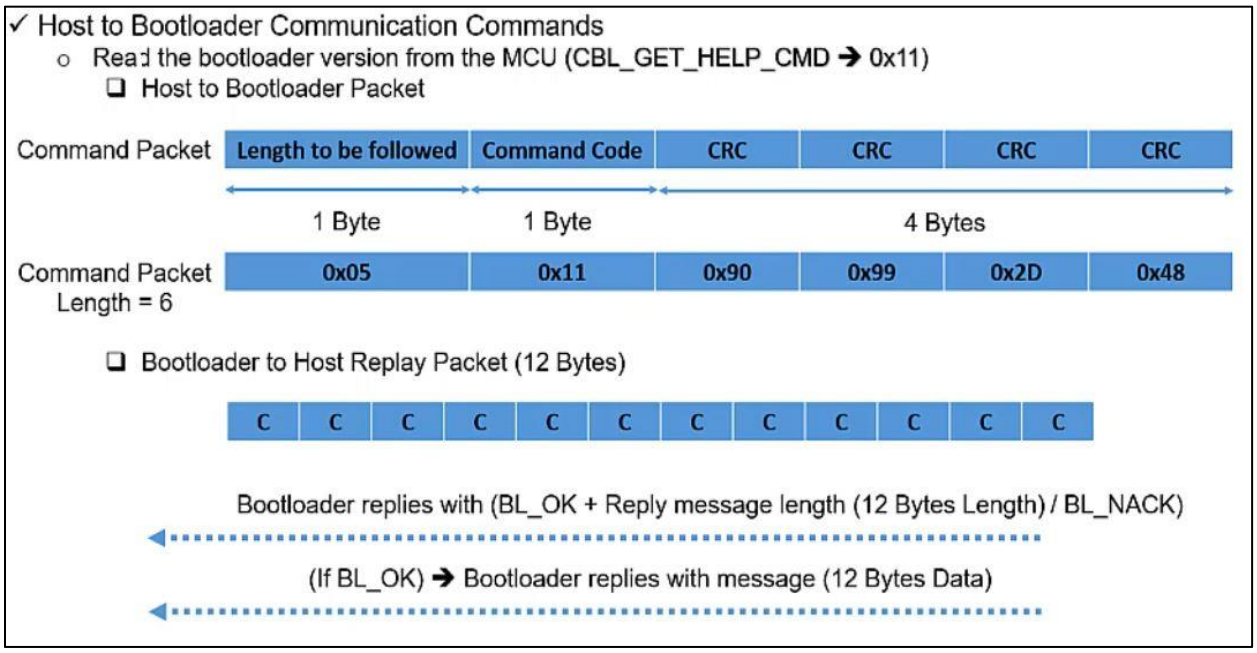
1- ACK
2- Replied Message Length which in this case is 4 Bytes.

The Host will prepare a Buffer to receive that 4 Bytes.

- The Boot Loader will send 4 Bytes contain the Bootloader Version that is mentioned above.
- If the Bootloader didn't sent the 4 Bytes, a timeout will be happened.

If the 2-CRC are not the same, The Bootloader will send only one Byte with **NOTACK**.

1.7.1.2 Read the supported Command from the MCU


✓ Host to Bootloader Communication Commands
Read the bootloader version from the MCU (CBL_GET_HELP_CMD → 0x11)
❑ Host to Bootloader Packet
Command Packet
Length to be followed
Command Code
CRC
CRC
CRC
CRC
1 Byte
1 Byte
4 Bytes
Command Packet Length = 6
0x05
0x11
0x90
0x99
0x2D
0x48
❑ Bootloader to Host Replay Packet (12 Bytes)
C C C C C C C C C C C C
Bootloader replies with (BL_OK + Reply message length (12 Bytes Length) / BL_NACK)
(If BL_OK) → Bootloader replies with message (12 Bytes Data)

The Host need to read supported Commands of the Bootloader that are mentioned at the previous chapter.

The command Packet that will be sent by the **Host** is 6 Bytes that contains:

1- Length of the Packet that will be sent to the Bootloader.
2- Command Code ID. Every Command has its own ID that will be processed by the Bootloader.
3- The Host will calculate the CRC and sent it to the bootloader with 4Bytes length.

The Bootloader will receive this Command Packet and Parse it and calculate the CRC to the first 2 Bytes and compare between the CRC sent by the Host and its own calculated CRC to ensure that the 2-CRCs are verified.

If the 2-CRC are the same, The Bootloader will send 2 Bytes which are:

1- ACK
2- Replied Message Length which in this case is 12 Bytes.

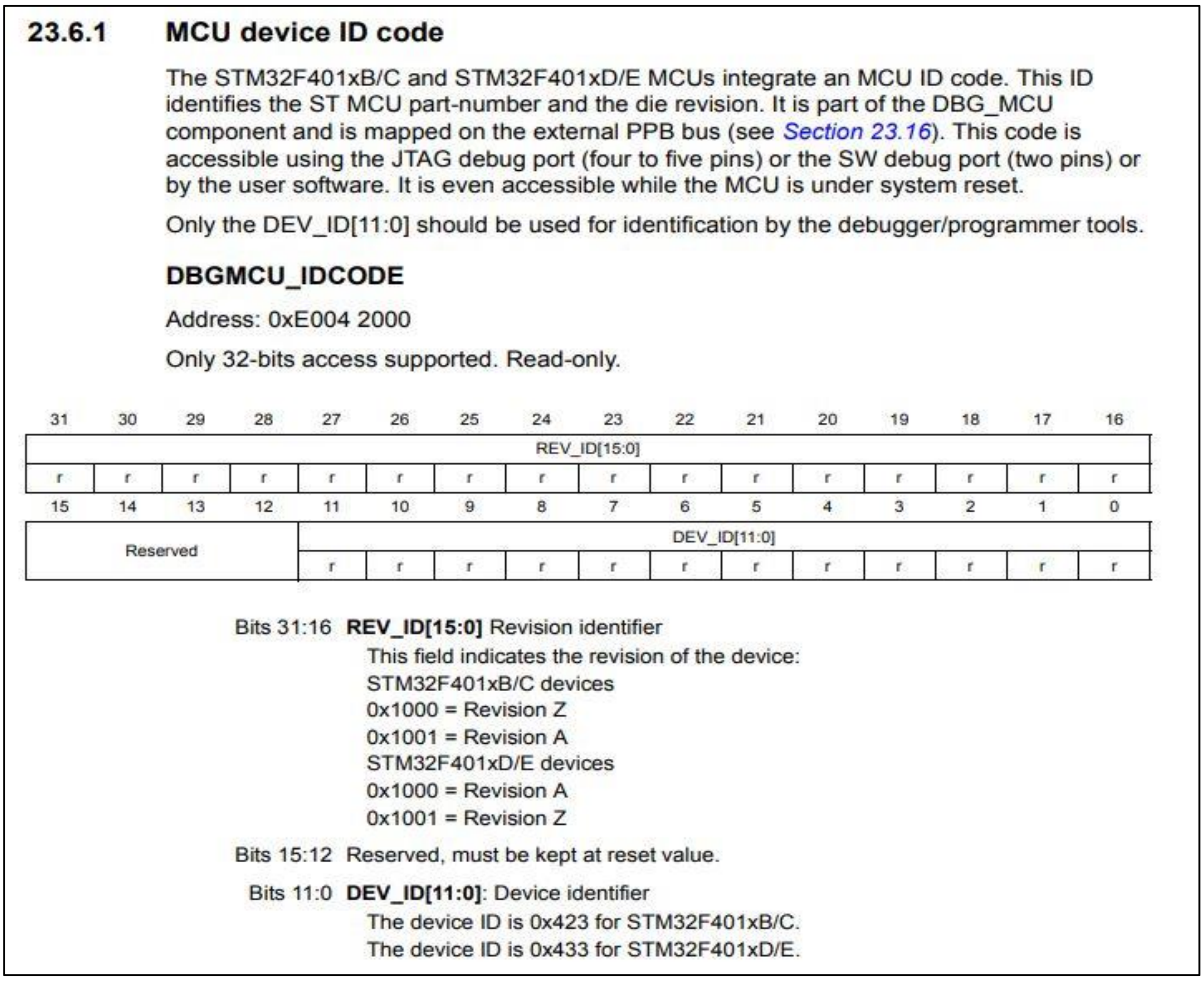
The Host will prepare a Buffer to receive that 12 Bytes.

- The Boot Loader will send 12 Bytes contain the supported commands that is mentioned in the previous chapter.
- If the Bootloader didn’t sent the 12 Bytes, a timeout will be happened.

If the 2-CRC are not the same, The Bootloader will send only one Byte with **NOTACK**.

1.7.1.3 Read the Chip Identification Number

Every MCU has its own Device ID Code which will be inside the DBGMCU_IDCODE Register from Bit0 to Bit11 as shown in the below figure…



23.6.1 MCU device ID code

The STM32F401xB/C and STM32F401xD/E MCUs integrate an MCU ID code. This ID identifies the ST MCU part-number and the die revision. It is part of the DBG_MCU component and is mapped on the external PPB bus (see *Section 23.16*). This code is accessible using the JTAG debug port (four to five pins) or the SW debug port (two pins) or by the user software. It is even accessible while the MCU is under system reset.

Only the DEV_ID[11:0] should be used for identification by the debugger/programmer tools.

DBGMCU_IDCODE

Address: 0xE004 2000

Only 32-bits access supported. Read-only.

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
REV_ID[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				DEV_ID[11:0]											
				r	r	r	r	r	r	r	r	r	r	r	r

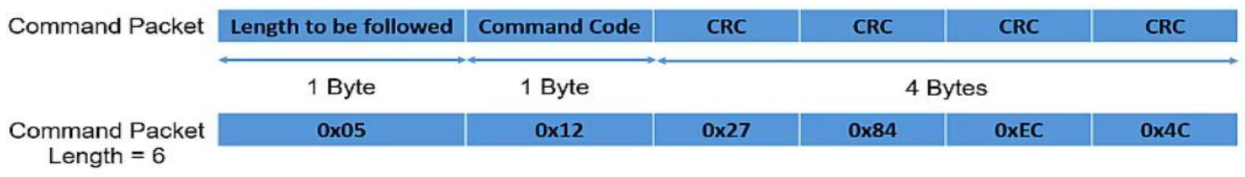
Bits 31:16 **REV_ID[15:0]** Revision identifier
This field indicates the revision of the device:
STM32F401xB/C devices
0x1000 = Revision Z
0x1001 = Revision A
STM32F401xD/E devices
0x1000 = Revision A
0x1001 = Revision Z

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

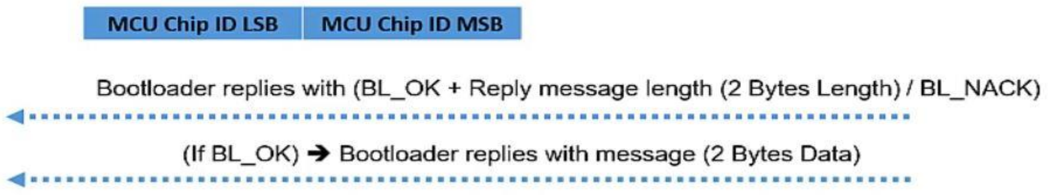
Bits 11:0 **DEV_ID[11:0]**: Device identifier
The device ID is 0x423 for STM32F401xB/C.
The device ID is 0x433 for STM32F401xD/E.

Every Family has its own ID.

✓ Host to Bootloader Communication Commands
- o Read the bootloader version from the MCU (CBL_GET_CID_CMD ➔ 0x12)
  - ❑ Host to Bootloader Packet

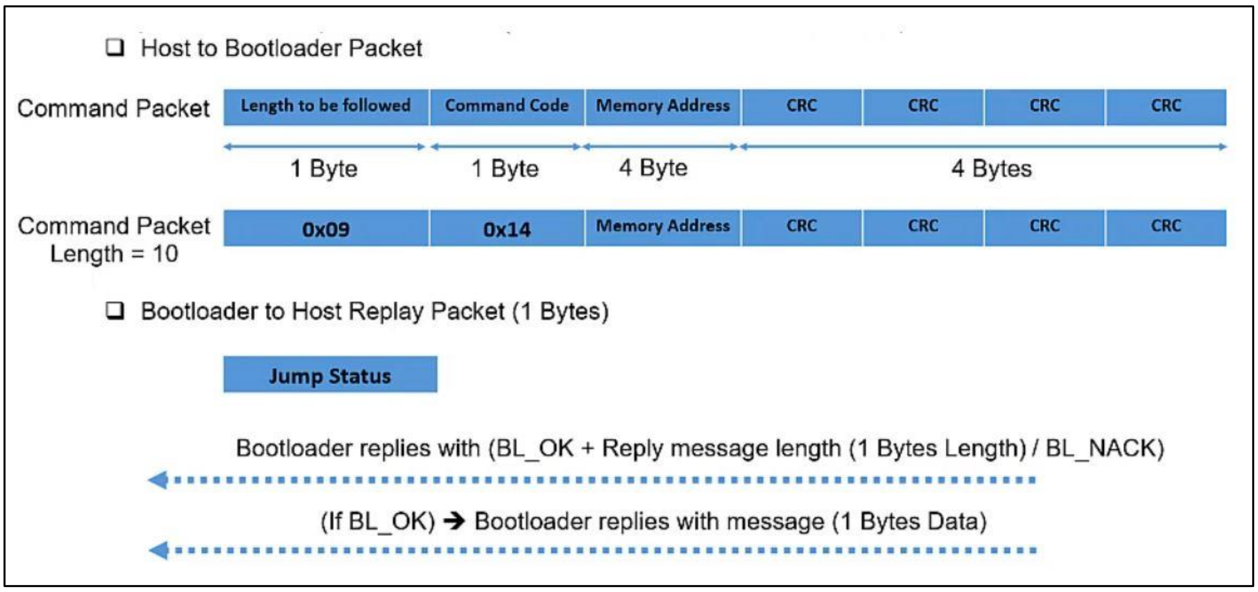


  - ❑ Bootloader to Host Replay Packet (2 Bytes)


MCU Chip ID LSB
MCU Chip ID MSB
Bootloader replies with (BL_OK + Reply message length (2 Bytes Length) / BL_NACK)
(If BL_OK) ➔ Bootloader replies with message (2 Bytes Data)

The Procedures are the same as the previous commands…

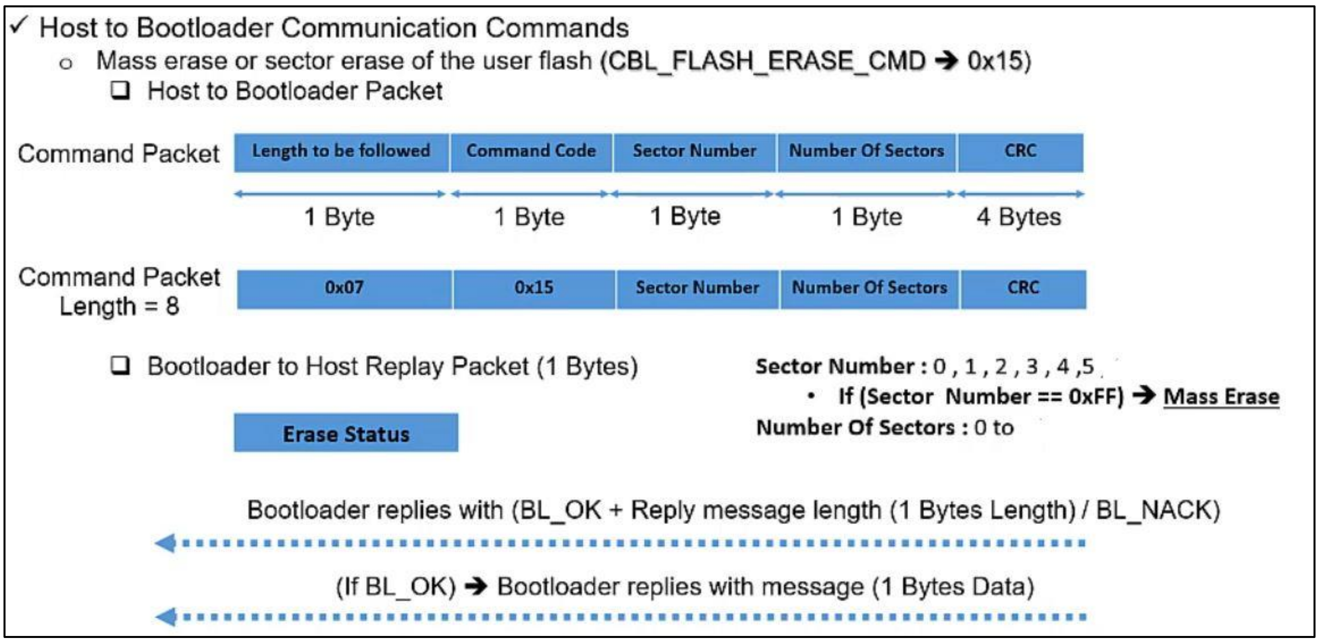
1.7.1.4 Go to Application Code and start the Vehicle.


Host to Bootloader Packet
Command Packet
Length to be followed
Command Code
Memory Address
CRC
CRC
CRC
CRC
1 Byte
1 Byte
4 Byte
4 Bytes
Command Packet Length = 10
0x09
0x14
Memory Address
CRC
CRC
CRC
CRC
Bootloader to Host Replay Packet (1 Bytes)
Jump Status
Bootloader replies with (BL_OK + Reply message length (1 Bytes Length) / BL_NACK)
(If BL_OK) → Bootloader replies with message (1 Bytes Data)

The host will send that command to the Bootloader with the specified Application Address.

- Bootloader will Verify the address first before jumping to that address.
- Bootloader Application will De-initialize all modules and RCC Clock of the bootloader before jumping to the application code.
- Read the first address of Application and this value will be the main stack pointer, then it will set the main stack pointer (SP) to the value given by the first address of the application.
- Bootloader will read the second address of the application which is the location of the **Reset Handler** Function of the application and will jump to that address.

1.7.1.5 Mass Erase or Sector Erase Command Packet.


✓ Host to Bootloader Communication Commands
o Mass erase or sector erase of the user flash (CBL_FLASH_ERASE_CMD → 0x15)
❑ Host to Bootloader Packet
Command Packet
Length to be followed
Command Code
Sector Number
Number Of Sectors
CRC
1 Byte
1 Byte
1 Byte
1 Byte
4 Bytes
Command Packet Length = 8
0x07
0x15
Sector Number
Number Of Sectors
CRC
❑ Bootloader to Host Replay Packet (1 Bytes)
Sector Number : 0 , 1 , 2 , 3 , 4 ,5
• If (Sector Number == 0xFF) → Mass Erase
Number Of Sectors : 0 to
Erase Status
Bootloader replies with (BL_OK + Reply message length (1 Bytes Length) / BL_NACK)
(If BL_OK) → Bootloader replies with message (1 Bytes Data)

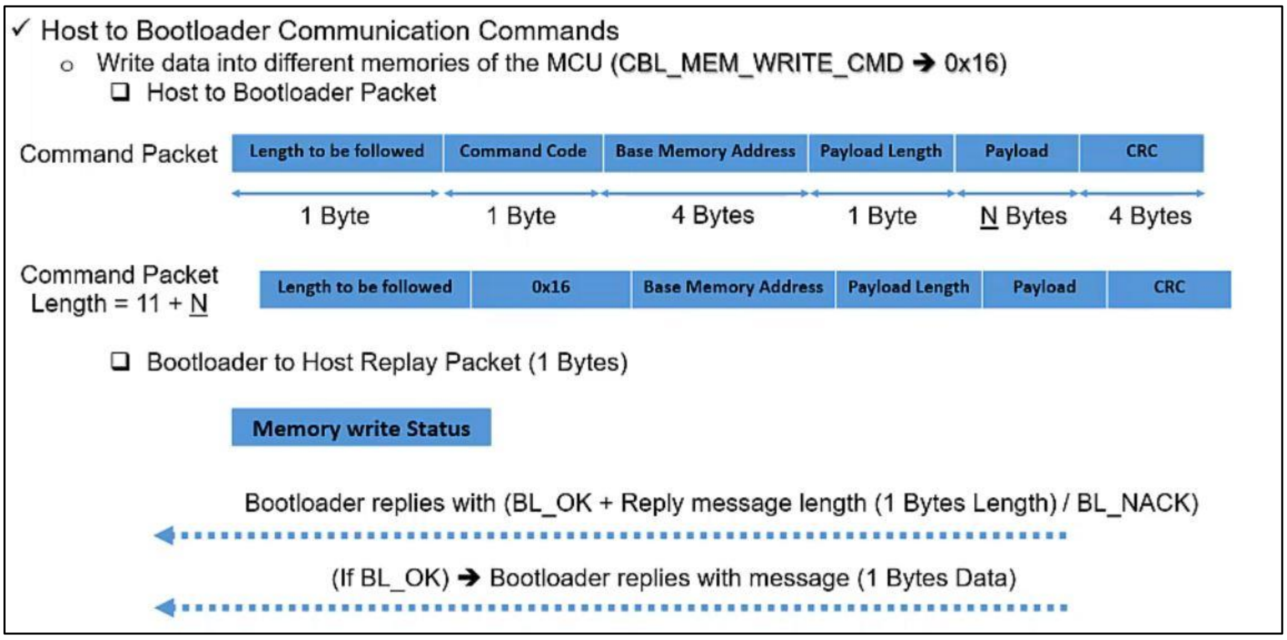
In STM32F401CC, we have only 6 Sectors.

The host will send with the packet, the first sector to be erased and the number of sectors to be erased.

The bootloader will:

- Calculate the CRC to the data sent by the host.
- Compare the 2-CRCs with each other.
- If the 2-CRCs is verified, it will send ACK Byte and the length of the replied messaged that will be the Erase Status.
  Bootloader will verify the Start sector number and the number of sectors to be erased starting from the first sector number.
  If the Sectors within the STM32 MCU, it will perform the Host command erase and send Erase Status success to the Host only if the Flash erase is successful.
  If it is not within the range of the flash sector, it will send Erase Status Fail to the Host.
  Note:
  If the number of Sectors given by the Host is greater than the STM32 MCU sectors, it will erase nothing. And send Erase Status Fail.
  If the start sector is not in the STM32 MCU, it will not erase either.
- If the 2-CRCs is not verified, it will send NACK Byte.

1.7.1.6 Write data into different memories of the MCU Command Packet.


✓ Host to Bootloader Communication Commands
o Write data into different memories of the MCU (CBL_MEM_WRITE_CMD → 0x16)
❑ Host to Bootloader Packet
Command Packet
Length to be followed
Command Code
Base Memory Address
Payload Length
Payload
CRC
1 Byte
1 Byte
4 Bytes
1 Byte
N Bytes
4 Bytes
Command Packet Length = 11 + N
Length to be followed
0x16
Base Memory Address
Payload Length
Payload
CRC
❑ Bootloader to Host Replay Packet (1 Bytes)
Memory write Status
Bootloader replies with (BL_OK + Reply message length (1 Bytes Length) / BL_NACK)
(If BL_OK) → Bootloader replies with message (1 Bytes Data)

The Host has the binary file that will be sent to the STM32 MCU.
The command Packet sent by the Host contains:

1- Length to be followed (1Byte):
   Length of the Packet
2- Command Code (1Byte):
   Command Code of the write Data.
3- Base Memory Address (4Bytes):
   The Fist Address that will be written to.
4- Payload Length (1Byte):
   The length of the data that will be written into the Flash Memory.
   Note: One Byte means that we can send data with 255 Bytes maximum in the first time.
5- Payload:
   It is the data that will be flashed to the flash memory.
6- CRC (4Bytes):
   The CRC calculated for that packet.

The Bootloader will:

1- Calculated the CRC of the packet and compare between the 2-CRCs.
2- If the CRC is failed, it will send NOTACK and stop the writing Process.
3- If the CRC is successful, it sends ACK and the replied length.
4- Bootloader will parse the Base memory address that will start flash software from it. It will also, parse the payload length from the command packet.
5- Bootloader will verify the address sent by the Host with the addresses of the flash memory.
6- If the address is not in the range on the addresses, it will send NOTACK.

7- The data is flashed into the flash memory byte by byte because it send from the Host byte by byte.

1.7.1.7 Read Protection Level of the MCU Command Packet.

The user area in the Flash memory can be protected against read operations and there are three levels of operation.

1- Level 0: no read protection:
When the read protection level is set to Level 0, all read/write operations (if no write protection is set) from/to the Flash memory are possible in all boot configurations (Flash user boot, debug or boot from RAM).

2- Level 1: read protection enabled:
When the read protection Level 1 is set:
– No access (read, erase, program) to Flash memory can be performed while the debug feature is connected or while booting from RAM or system memory bootloader. A bus error is generated in case of read request.
– When booting from Flash memory, accesses (read, erase, program) to Flash memory from user code are allowed. When Level 1 is active, programming the protection option byte (RDP) to Level 0 causes the Flash memory to be mass erased. As a result, the user code area is cleared before the read protection is removed. The mass erase only erases the user code area. The other option bytes including write protections remain unchanged from before the mass-erase operation. The OTP area is not affected by mass erase and remains unchanged. Mass erase is performed only when Level 1 is active and Level 0 requested. When the protection level is increased (0->1, 1->2, 0->2) there is no mass erase.

3- Level 2: debug/chip read protection disabled:
When the read protection Level 2 is set: – All protections provided by Level 1 are active. – Booting from RAM or system memory bootloader is no more allowed. – JTAG, SWV (single-wire viewer), ETM, and boundary scan are disabled. – User option bytes can no longer be changed. – When booting from Flash memory, accesses (read, erase and program) to Flash memory from user code are allowed. Memory read protection Level 2 is an irreversible operation. When Level 2 is activated, the level of protection cannot be decreased to Level 0 or Level 1.

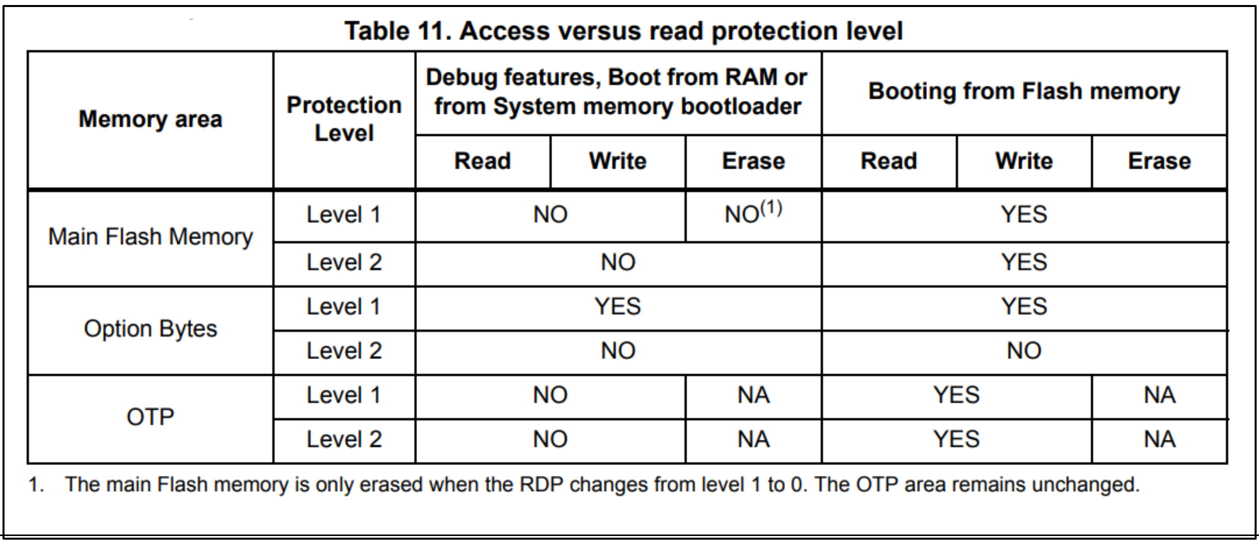
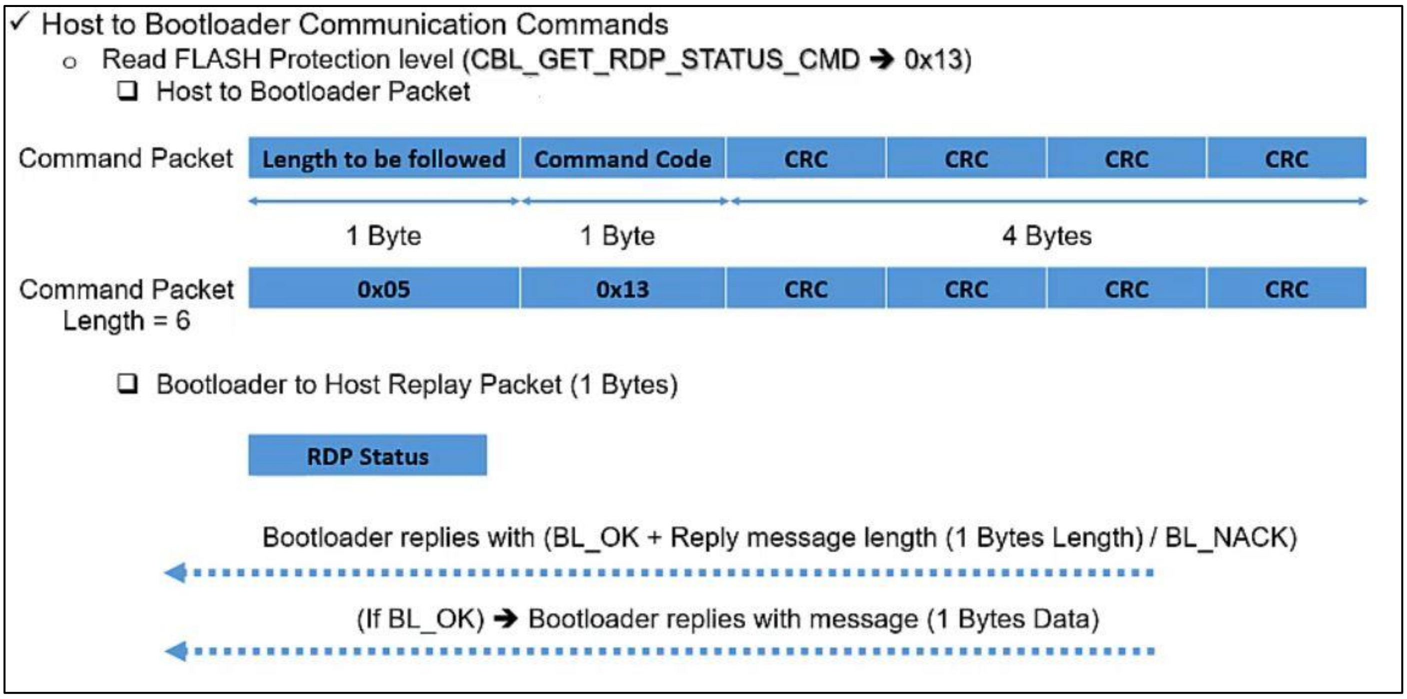


Table 11. Access versus read protection level

Memory area	Protection Level	Debug features, Boot from RAM or from System memory bootloader			Booting from Flash memory		
		Read	Write	Erase	Read	Write	Erase
Main Flash Memory	Level 1	NO		NO[1]	YES		
	Level 2	NO			YES		
Option Bytes	Level 1	YES			YES		
	Level 2	NO			NO		
OTP	Level 1	NO		NA	YES		NA
	Level 2	NO		NA	YES		NA

1. The main Flash memory is only erased when the RDP changes from level 1 to 0. The OTP area remains unchanged.


✓ Host to Bootloader Communication Commands

- Read FLASH Protection level (CBL_GET_RDP_STATUS_CMD → 0x13)
  - ❑ Host to Bootloader Packet

Command Packet
Length to be followed
Command Code
CRC
CRC
CRC
CRC
1 Byte
1 Byte
4 Bytes
Command Packet Length = 6
0x05
0x13
CRC
CRC
CRC
CRC

- ❑ Bootloader to Host Replay Packet (1 Bytes)

RDP Status
Bootloader replies with (BL_OK + Reply message length (1 Bytes Length) / BL_NACK)
(If BL_OK) → Bootloader replies with message (1 Bytes Data)

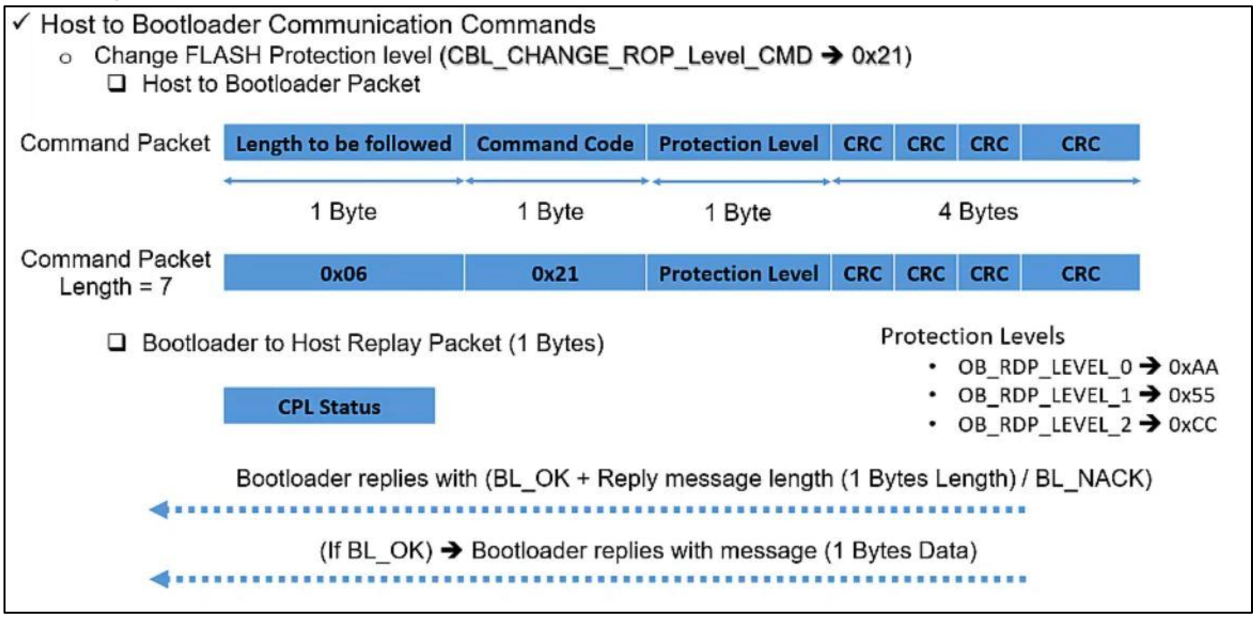
The command Packet sent by the Host contains:

1- Length to be followed (1Byte):
   Length of the Packet
2- Command Code (1Byte):
   Command Code of Read Protection Level.
3- CRC (4Bytes):
   The CRC calculated for that packet.

The Bootloader will:

1- Calculated the CRC of the packet and compare between the 2-CRCs.
2- If the CRC is failed, it will send NOTACK.
3- If the CRC is successful, it sends ACK and the replied length.
4- The Bootloader will send the Protection Level.

1.7.1.8 Change Protection Level of the MCU Command Packet.


✓ Host to Bootloader Communication Commands
o Change FLASH Protection level (CBL_CHANGE_ROP_Level_CMD → 0x21)
❑ Host to Bootloader Packet
Command Packet
Length to be followed
Command Code
Protection Level
CRC
CRC
CRC
CRC
1 Byte
1 Byte
1 Byte
4 Bytes
Command Packet Length = 7
0x06
0x21
Protection Level
CRC
CRC
CRC
CRC
❑ Bootloader to Host Replay Packet (1 Bytes)
CPL Status
Protection Levels
• OB_RDP_LEVEL_0 → 0xAA
• OB_RDP_LEVEL_1 → 0x55
• OB_RDP_LEVEL_2 → 0xCC
Bootloader replies with (BL_OK + Reply message length (1 Bytes Length) / BL_NACK)
(If BL_OK) → Bootloader replies with message (1 Bytes Data)

The command Packet sent by the Host contains:

1- Length to be followed (1Byte):
Length of the Packet

2- Command Code (1Byte):
Command Code of Change the Protection Level.

3- Protection Level (1Byte):
The host will send the protection level which can be Level 0 or Level 1 or Level 2.

4- CRC (4Bytes):
The CRC calculated for that packet.

The Bootloader will:

1- Calculated the CRC of the packet and compare between the 2-CRCs.
2- If the CRC is failed, it will send NOTACK.
3- If the CRC is successful, it sends ACK and the replied length.
The Bootloader will change the protection level and send the change protection level status either it is successes or fail.

2 Firmware Over-The-Air (FOTA)

2.1 Introduction

2.1.1 Overview

Firmware Over-The-Air (FOTA) in embedded systems allows wireless deployment of firmware updates, enhancing the efficiency and reliability of devices. This documentation provides a detailed guide for implementing FOTA on STM32 ARM-based microcontrollers using Raspberry Pi 3 as a WIFI module, UART as the communication protocol, and Fernet encryption in Python for securing firmware files.

2.1.2 Purpose

The purpose of this documentation is to enable seamless, secure, and remote firmware updates for embedded applications, improving device functionality and ensuring a better user experience.

2.2 How FOTA Works

2.2.1 Architecture

- **Client-side components:** STM32 microcontroller, UART communication module.
- **Raspberry Pi 3 components:** WIFI module, UART communication module, Fernet encryption module.

2.2.2 FOTA Workflow for STM32 ARM-based Microcontrollers with Raspberry Pi and Fernet Encryption

- **Initiation of update process:** Triggered remotely via server.
- **Firmware Encryption:** Firmware files are encrypted using Fernet encryption in Python before uploading the firmware to the server.
- **Downloading encrypted packages:** Raspberry Pi downloads encrypted firmware files from the server.
- **Decryption:** Firmware files are decrypted using the Fernet encryption key.
- **Verification and validation:** Firmware integrity check and validation.
- **Installation and rollback mechanisms:** UART communication with STM32 for flashing new firmware and handling failures.

2.3 System Implementation

2.3.1 Device Compatibility

- **STM32 ARM-based microcontroller requirements:** Flash size, RAM, UART pins.
- **Raspberry Pi 3 requirements:** WIFI connectivity, UART pins, adequate processing power.

3.2 FOTA Integration with STM32 Microcontroller and Raspberry Pi 3

Setting up Raspberry Pi 3 as a WIFI Module, UART Communication, and Fernet Encryption

- **Configure WIFI on Raspberry Pi:** Establish a WIFI connection to the local network.
- **UART Configuration:** Set up UART communication between Raspberry Pi and STM32 microcontroller.
- **Firmware Encryption:** Use Fernet encryption in Python to encrypt firmware files before uploading them to the server.
- **Download Files:** Write a python script on Raspberry Pi to download encrypted firmware files from the server using a **python function**.
- **Decryption:** Decrypt firmware files using Fernet encryption key before sending them to STM32 for flashing.
- **File Storage:** Store downloaded and decrypted firmware files accessible by the STM32 microcontroller.

2.3.2 Integration with STM32 Microcontroller

- **UART Communication Protocol:** Implement UART communication protocol between Raspberry Pi and STM32 microcontroller for data transmission.
- **STM32 Firmware Update Logic:** Develop logic to check for available updates, initiate the update process, and confirm successful updates received from Raspberry Pi.
- **Bootloader Flashing:** Utilize the pre-installed bootloader on the microcontroller to enable firmware updates via UART communication. Ensure the bootloader is properly configured and compatible with the update process.
- **Error Handling:** Implement error handling mechanisms to retry downloads, validate firmware integrity, and handle communication failures.

2.3.3 Server Setup

- **Update server configuration:** Set up a secure server for hosting encrypted firmware files, accessible over the internet.
- **Authentication:** Implement authentication mechanisms to restrict access to authorized devices only.

- **File Versioning:** Maintain version control for encrypted firmware files to ensure the correct version is downloaded by devices.

2.4 Benefits of FOTA for STM32 ARM-based Microcontrollers with Raspberry Pi, UART, and Fernet Encryption.

- **Enhanced Security:** Firmware files are encrypted during transmission and storage, requiring a decryption key for access.
- **Wireless Connectivity:** Utilizes Raspberry Pi 3 as a bridge for wireless communication, enabling devices without built-in WIFI modules to connect to the network.
- **Cost-Effectiveness:** Utilizes affordable and widely available hardware components for wireless communication.

2.5 Challenges and Considerations

- **Data Integrity:** Ensuring encrypted firmware packages are not tampered with during transmission.
- **Security:** Safeguarding the encryption key and ensuring secure communication channels to prevent unauthorized access and tampering.
- **Bootloader Compatibility:** Ensuring the compatibility of the pre-installed bootloader with the update process, verifying supported commands, and handling potential limitations.
- **Version Control:** Managing multiple versions of firmware and ensuring seamless transitions between different versions without causing disruptions in device functionality.

2.6 Conclusion

Implementing FOTA for STM32 ARM-based microcontrollers using Raspberry Pi 3 as a WIFI module, UART communication protocol, and Fernet encryption in Python offers a robust, scalable, and highly secure solution for wireless firmware updates. By following the guidelines outlined in this documentation, developers can ensure their embedded applications remain up-to-date, reliable, and secure, even in remote or inaccessible locations.

3 Application

3.1 Modes

The Application can operate in two Modes which are:

- Lane Keeping System
- Obstacle Avoiding System

3.1.1 Lane Keeping System

Lane-Keeping Aid applies steering torque to help direct you back to the center of the lane. Lane-Keeping Alert warns you through steering wheel vibrations to keep the car safe.

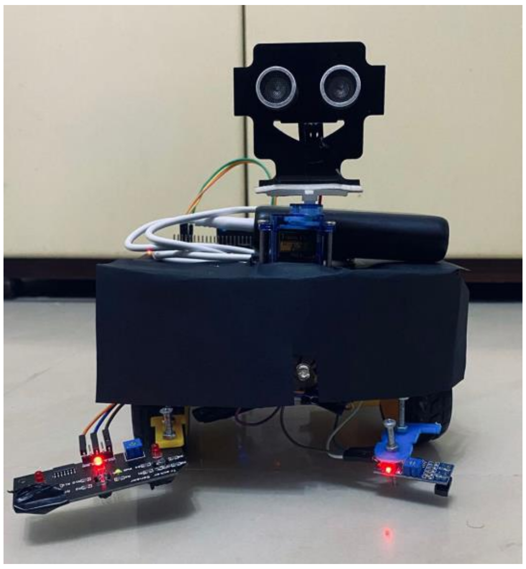
3.1.2 Obstacle Avoiding System

Obstacle avoiding system prevents car accidents by detecting any moving objects. Taking decisions and measuring surrounding distances to select the best route to go through.

3.1.3 Vehicle to Vehicle Communication

The Front Vehicle sends the distance between it and the front object to the rear vehicle. If the distance is less than a specified value, it will send to the rear vehicle to stop immediately.

3.2 Hardware Snapshot

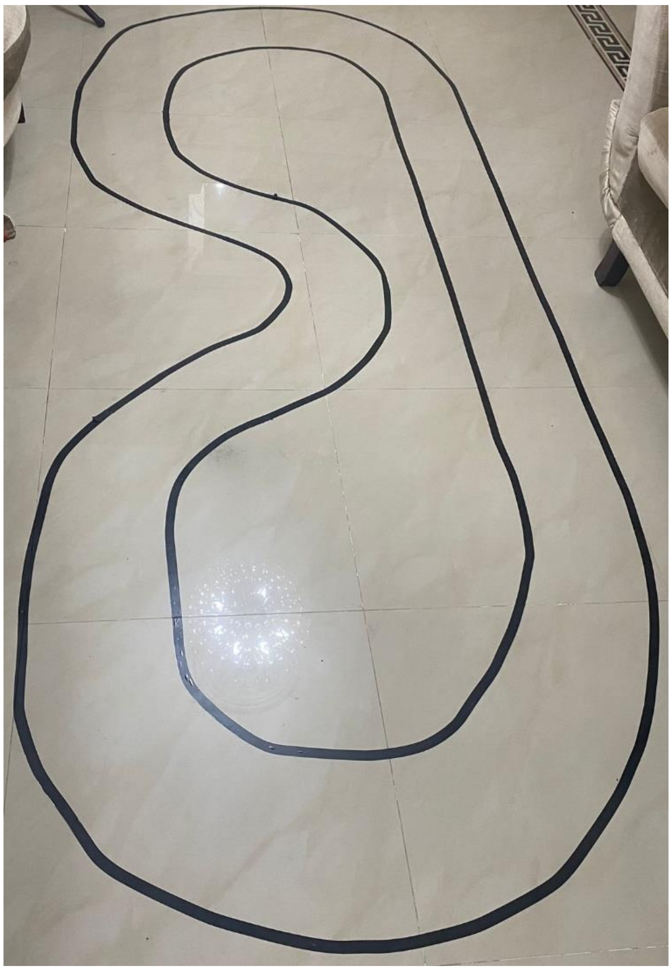


3.3 Component Used

- Four DC motor
- DC motor Driver
- Two Line follower sensors
- Three Batteries
- Power Bank to power the Raspberry pi
- STM32F401CCU6
- Raspberry pi 3
- Servo motor
- Ultrasonic Sensor

3.4 Test Case

A complex Route has been made to test the RC Modes.



3.5 Project Goals

- Helps drivers to keep their vehicle within the lane.
- Avoid obstacles with ultrasonic and find the best route to go through.