Ataberk ÖKLÜ

STG Stajyer

USART - BootLoader

Solution to the Problem An Example Program

İçindekiler

[The bootloader via USART / UART interface 2](#_Toc51528275)

[How to enter the bootloader process 2](#_Toc51528276)

[Where is this Boot0 Pin? 3](#_Toc51528277)

[Boot0 pin problem? 4](#_Toc51528278)

[First Step: UART RX Interrupt 4](#_Toc51528279)

[Reset2BootLoader Function Definition 4](#_Toc51528280)

[Flash Write Function Definition 4](#_Toc51528281)

[Second Step: Soft-Reset Handling 5](#_Toc51528282)

[Memory Mapping 6](#_Toc51528283)

[How to Connect Devices 7](#_Toc51528284)

[Which Port the device is using 7](#_Toc51528285)

[What Happens in Bootloader Process 8](#_Toc51528286)

[How can we order a command 11](#_Toc51528287)

[Communication Safety 11](#_Toc51528288)

[Receiving Information via Bootloader 12](#_Toc51528289)

[How to Write our code - Write Memory command 13](#_Toc51528290)

[Where to write our code 14](#_Toc51528291)

[Some constraints we need to obey 14](#_Toc51528292)

[Example Code to be Written 15](#_Toc51528293)

[An Example Program – STM32 Flasher 16](#_Toc51528294)

[Connection Properties 16](#_Toc51528295)

[Read or Write Protection 17](#_Toc51528296)

[GET Information from the device via bootloader 18](#_Toc51528297)

[WRITE CMD 19](#_Toc51528298)

[Utility Tools 20](#_Toc51528299)

[Hex Reader 20](#_Toc51528300)

[UART BootLoader Trigger 21](#_Toc51528301)

# The bootloader via USART / UART interface

## How to enter the bootloader process

The STM32L47xxx/48xxx bootloader is activated by applying Pattern 7.

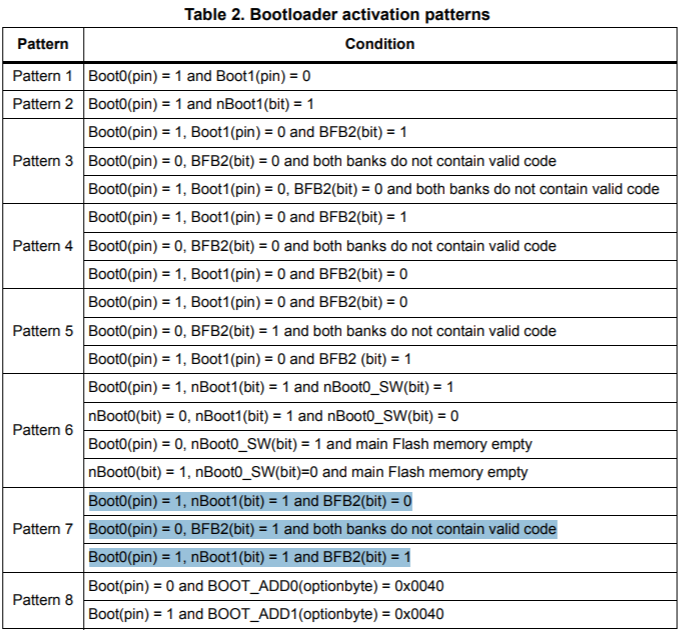
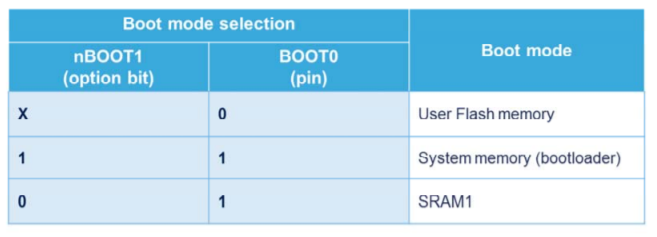


Figure 1 - BootLoader Activation Patterns - [Source](https://www.st.com/resource/en/application_note/cd00167594-stm32-microcontroller-system-memory-boot-mode-stmicroelectronics.pdf#page=23)

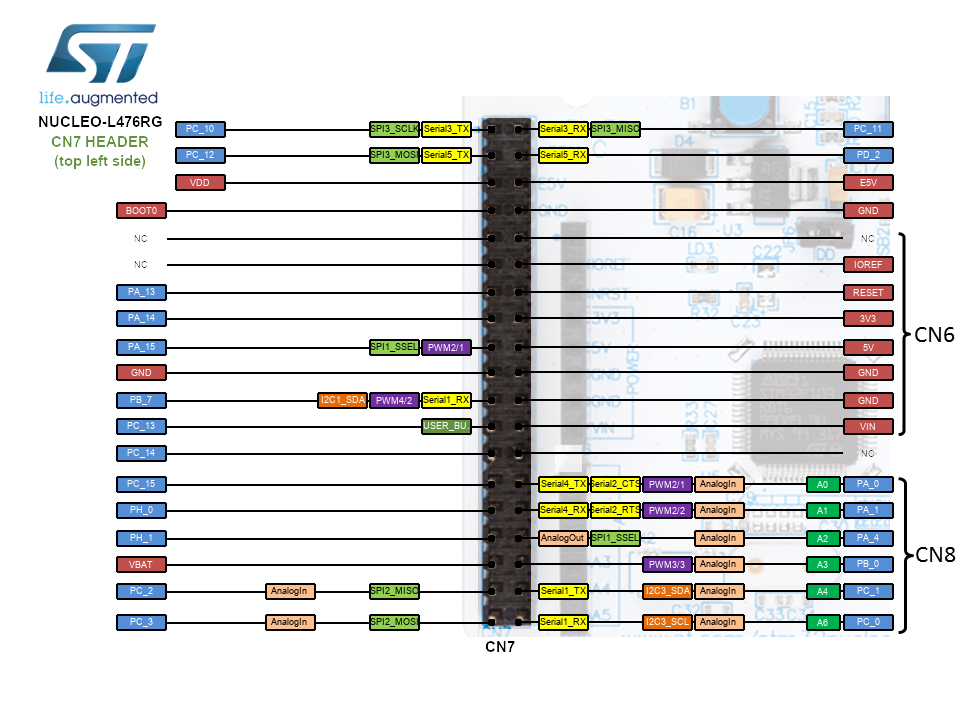


**DEBUG**

**DEFAULT**

Figure 2 - Boot Modes - [Source](https://www.st.com/content/ccc/resource/training/technical/product_training/08/4c/83/1b/56/bd/45/34/STM32L4_System_SYSCFG.pdf/files/STM32L4_System_SYSCFG.pdf/jcr:content/translations/en.STM32L4_System_SYSCFG.pdf#page=8)

## Where is this Boot0 Pin?



For convention, pushing Boot0 pin to HIGH, then resetting results in BootLoader Mode.

## Boot0 pin problem?

Since Boot0 pin should be pushed HIGH by physically, it requires at least one more pin accessed from outside world other than GND, RX, and TX, reserved UART pins. We needed to bypass this requirement to achieve jump to BootLoader @ System Memory (0x1FFF0000).

The constructed bypasser is using the method of “Cipher Check.” The method is merely checking the value at the predefined memory location, whether it is the predetermined cipher, triggering the jump to BootLoader @ System Memory (0x1FFF0000) (See [Memory Mapping](#_Memory_Mapping)). If it is not the case, Reset\_Handler @ startup.s file initiates the main program as default. However, when cipher is caught at the predefined memory location, then Reset\_Handler, mention above, executes the Reboot\_Loader routine in the startup.s file. And, the only way the cipher to be written to the specified location is triggering the RX Interrupt of reserved UART pins. Moreover, the following executions guarantee that cipher is invalidated to prevent repetitive executions of Reboot\_Loader, mentioned above. Let us examine the method elaborately.

## First Step: UART RX Interrupt

When RX of the reserved accessible UART interface is triggered, it calls USARTx\_IRQHandler, which is executing the Reset2BootLoader function defined in main.c file:

### Reset2BootLoader Function Definition

void Reset2BootLoader**(**void**)**

**{**

FlashWrite**(**CIPHER\_ADDR **,** MAGIC\_CIPHER**);** // Write Special Code "ATABERK" to End of the SRAM2 0x2000 0000

HAL\_NVIC\_ClearPendingIRQ**(**USART1\_IRQn**);** // USART1 Pending Bit RESET

\_\_DSB**();** // Blocking The Program until every memory instructions are done.

NVIC\_SystemReset**();** // Soft-RESET -> startup.s file -> RESET\_HANDLER + REBOOT\_LOADER

**}**

### Flash Write Function Definition

void FlashWrite**(**uint32\_t address**,** uint32\_t data**){**

// WHEN ADDR IS @ SRAM1, IT IS SUFFICIENT

**\*((**volatile uint32\_t**\*)(**address**))** **=** data**;**

// IF FLASH IS SELECTED, THE CODE BELOW

/\* FLASH WRITER START

uint32\_t PAGEError = 0;

FLASH\_EraseInitTypeDef EraseInitStruct;

EraseInitStruct.TypeErase = FLASH\_TYPEERASE\_PAGES;

EraseInitStruct.Page = 255;

EraseInitStruct.NbPages = 1;

EraseInitStruct.Banks = FLASH\_BANK\_1;

HAL\_FLASH\_Unlock();

\_\_HAL\_FLASH\_CLEAR\_FLAG(FLASH\_FLAG\_EOP | FLASH\_FLAG\_OPERR | FLASH\_FLAG\_WRPERR | FLASH\_FLAG\_PGAERR | FLASH\_FLAG\_PGSERR );

if (HAL\_FLASHEx\_Erase(&EraseInitStruct, &PAGEError) != HAL\_OK)

HAL\_FLASH\_GetError();

HAL\_FLASH\_Program(FLASH\_TYPEPROGRAM\_DOUBLEWORD, address, data);

HAL\_FLASH\_Lock();

FLASH WRITER END \*/

**}**

## Second Step: Soft-Reset Handling

When the device is reset, Reset\_Handler @ startup.s file runs:

; Reset\_Handler

Reset\_Handler PROC

EXPORT Reset\_Handler **[**WEAK**]**

IMPORT SystemInit

IMPORT \_\_main

LDR R0, **=**0x2000FFF0 ; CIPHER\_ADDR @ END\_OF\_SRAM1

LDR R1, **=**0xA7ABE12C ; ATABE R K **-** The MAGIC\_CIPHER

LDR R2, **[**R0**]** ; Take the value CIPHER\_ADDR

STR R0, **[**R0**]** ; Write itself onto itself

CMP R2, R1 ; CHECKING PROCCESS

BEQ Reboot\_Loader ; IF true**:** Execute Reboot\_Loader

LDR R0, **=**SystemInit

BLX R0

LDR R0, **=**\_\_main

BX R0

ENDP

; Reboot\_Loader

Reboot\_Loader PROC

EXPORT Reboot\_Loader

LDR R0, **=**0x40021060 ; RCC\_APB2ENR

LDR R1, **=**0x00000001 ; ENABLE SYSCFG CLOCK

STR R1, **[**R0**]**

LDR R0, **=**0x40010000 ; SYSCFG\_MEMRMP

LDR R1, **=**0x00000001 ; MAP ROM AT ZERO

STR R1, **[**R0**]**

LDR R0, **=**0x1FFF0000 ; SYSTEM\_MEMORY\_STARTING\_ADDR

LDR SP,**[**R0, #0] ; SP @ +0

LDR R0,**[**R0, #4] ; PC @ +4 - RESET VECTOR

BX R0

ENDP

Firstly, the cipher and the address are 0xA7ABE12C and 0x2000FFF0, respectively. The memory address is selected to be at the end of the SRAM1 portion of the STM32L476 MCU so that we can safely overwrite even there is a variable using this address. (See [Memory Mapping](#_Memory_Mapping)).

In Reset\_Handler routine, we check if this address holds any but the cipher. In the case of the cipher existence, indicating that UART RX Interrupt has occurred, Reset\_Handler executes the Reboot\_Loader routine. In the Reboot\_Loader, we first enable RCC, Clock, and Memory Initiations then jump to 0x1FFF0000 address holding the BootLoader @ System Memory (See [Memory Mapping](#_Memory_Mapping)), and goes its Reset\_Vector lying 4 bytes offset from the SP. Moreover, writing the cipher address into itself performs invalidation of the cipher at each reset, avoiding recursive occurrence.

On the other hand, not founding the cipher in the address means no UART RX Interrupt triggered, therefore, no need to jump to BootLoader. Then, hence the condition is not satisfied; Reset\_ Handler continues with loading SystemInit and jumps to the \_\_main vector – the main program vector.

## Memory Mapping

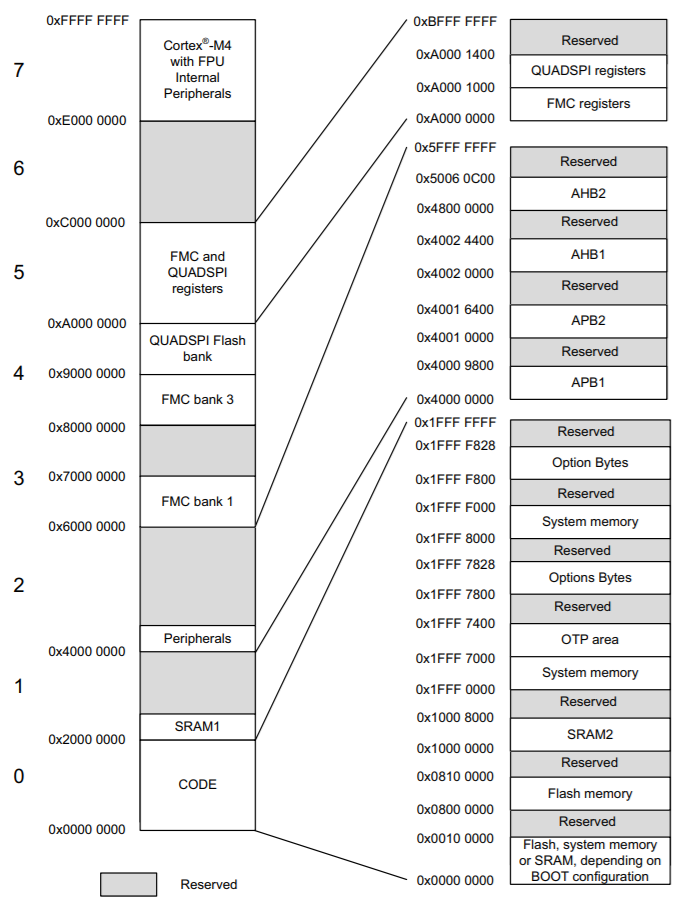


Figure 3 - Memory Map - [Source](https://www.st.com/resource/en/reference_manual/dm00083560-stm32l47xxx-stm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf#page=76)

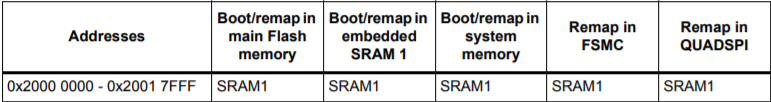
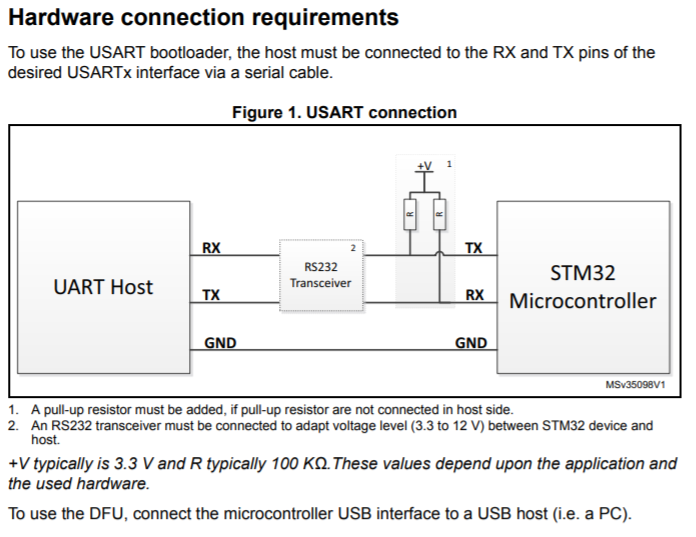


Figure 4 - SRAM1 Memory Addresses in different boots - [Source 1](https://www.st.com/resource/en/reference_manual/dm00083560-stm32l47xxx-stm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf#page=92) – [Source 2](https://www.st.com/content/ccc/resource/training/technical/product_training/08/4c/83/1b/56/bd/45/34/STM32L4_System_SYSCFG.pdf/files/STM32L4_System_SYSCFG.pdf/jcr:content/translations/en.STM32L4_System_SYSCFG.pdf#page=4)

## How to Connect Devices



**Pull-up**

For TTL connection from PC only RX, TX and GND connections are sufficient if you are using UART TTL Converter. If you are using the USB interface, no further connections are needed.

### Which Port the device is using

If you are using your PC to connect to the device, by using either USB TLL converter or direct USB connection, the Port likely to be in the form of “COMx”. To check to port COM number, you can use “Device Manager” on WindowsOS. Under “Connection Ports”, you can see your device and port number listed here. If not the case, you may need to install the device driver. Here is the driver for the [PL2303 USB TTL converter](http://www.prolific.com.tw/US/ShowProduct.aspx?p_id=225&pcid=41).

## What Happens in Bootloader Process

When we jump to BootLoader via our Reboot\_Loader routine, the device is searching all receiver channels to catch a communication request. The protocol list is given below:

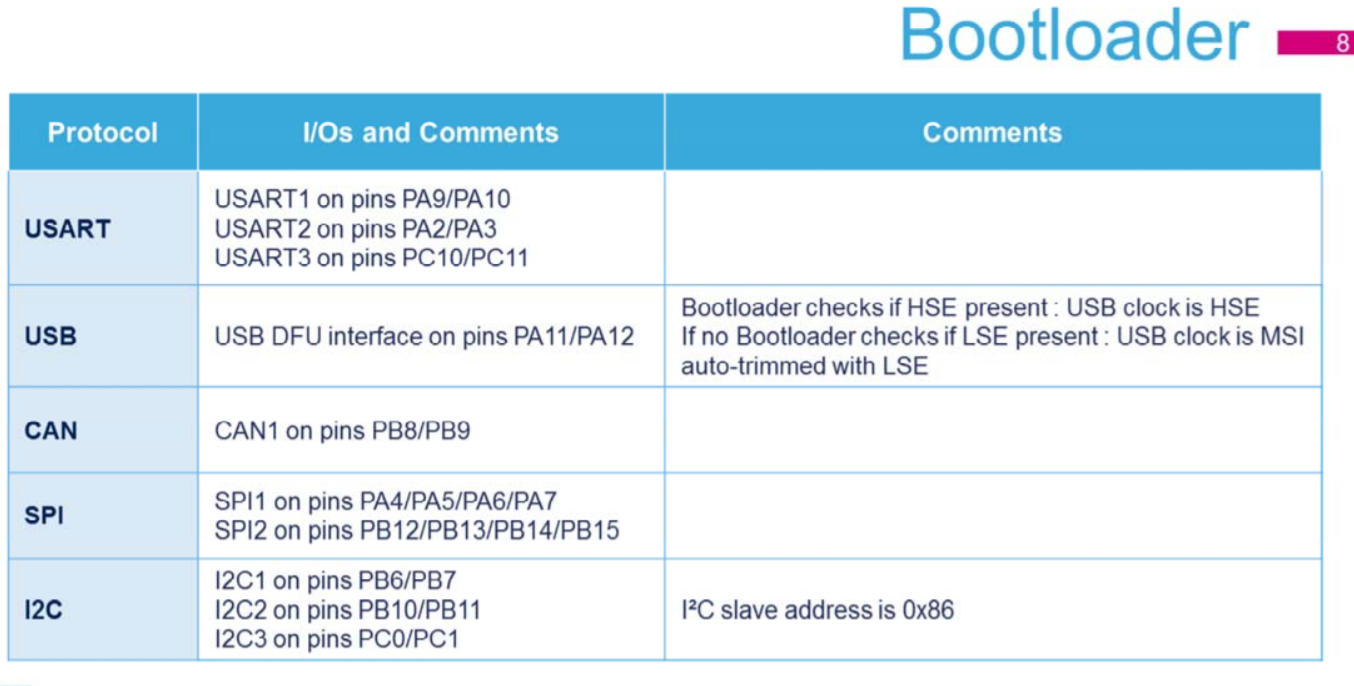


Figure 5 - BootLoader Communication Protocols - [Source](https://www.st.com/content/ccc/resource/training/technical/product_training/08/4c/83/1b/56/bd/45/34/STM32L4_System_SYSCFG.pdf/files/STM32L4_System_SYSCFG.pdf/jcr:content/translations/en.STM32L4_System_SYSCFG.pdf#page=10)

We focus on USART1 connection, for further information, please refer to the table and the source below:

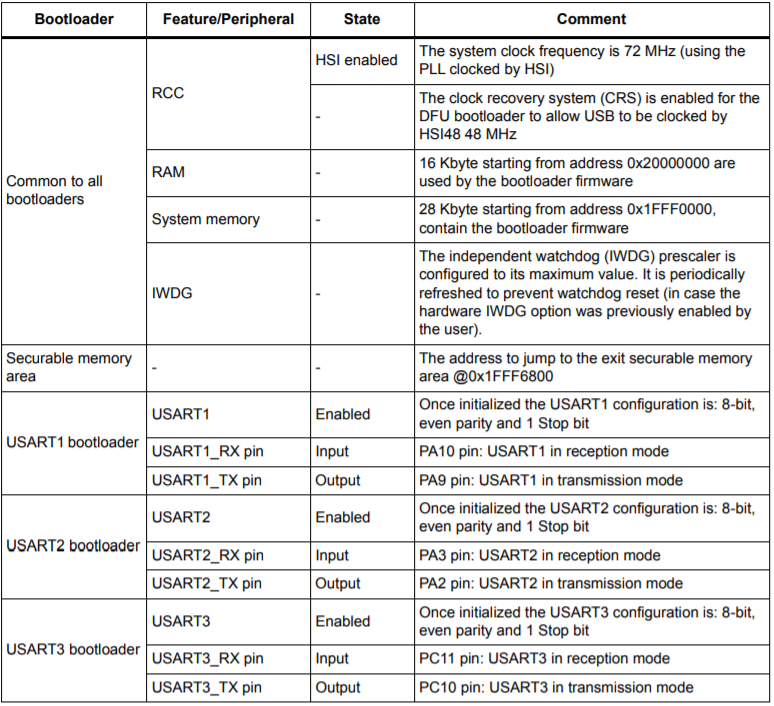
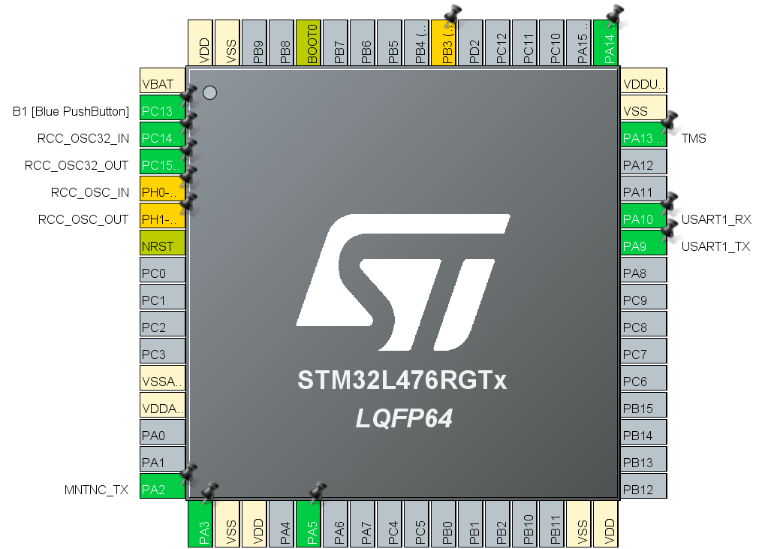
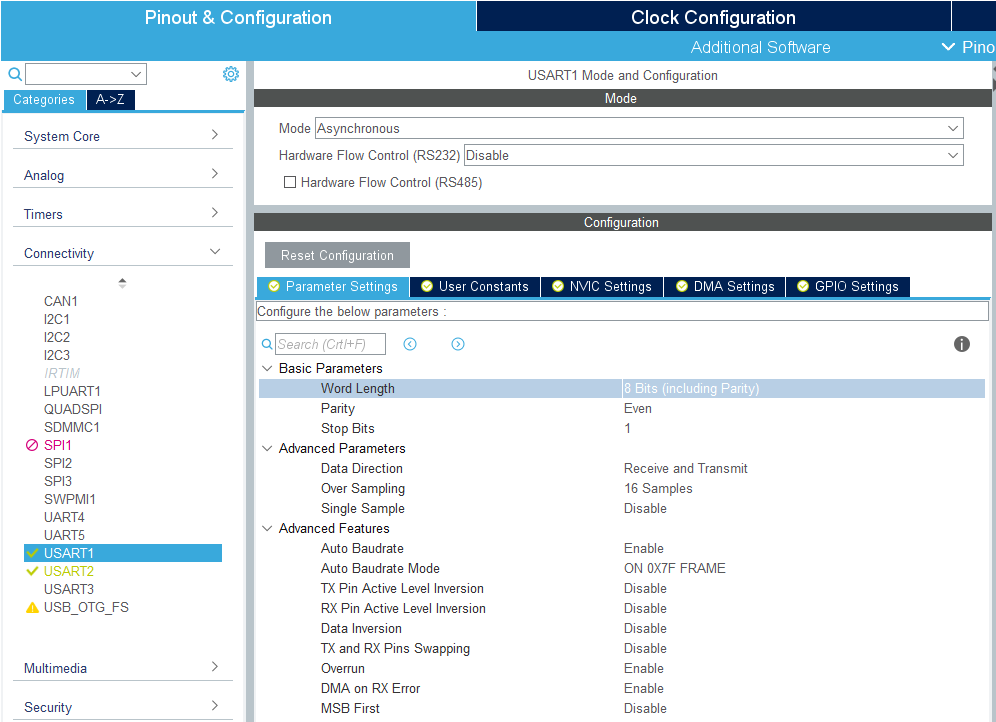


Figure 6 - Detailed Explanations For USART Connection - [Source](https://www.st.com/resource/en/application_note/cd00167594-stm32-microcontroller-system-memory-boot-mode-stmicroelectronics.pdf#page=214)



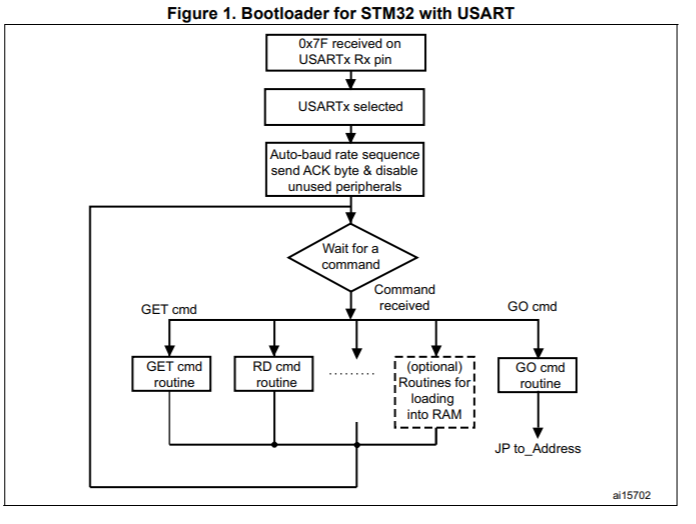
As stated, first, we need to initialize our USART1 in proper settings. To do this, we facilitate an ST software called [STM32CubeMX](https://www.st.com/en/development-tools/stm32cubemx.html). By setting PA9 and PA10 pins, RX, and TX, respectively. The software offers much more convenience.

Then we define our USART1 parameters obeying the given rules above:



Since the BootLoader is going to use this port also for Auto Boudrate finding, you may need to set this parameter.

To actively communicate and use the commands of bootloader, we need to follow the flow below:



**115200**

Figure 7 - BootLoader Protocol Selection - UART - [Source](https://www.st.com/resource/en/application_note/cd00264342-usart-protocol-used-in-the-stm32-bootloader-stmicroelectronics.pdf#page=5)

We activate the communication over USART1 by sending a 0x7F data frame, consisting of one start bit, 0x7F data, even parity bit, and one stop bit. According to the [Application Note – AN3155](https://www.st.com/resource/en/application_note/cd00264342-usart-protocol-used-in-the-stm32-bootloader-stmicroelectronics.pdf#page=8), the returned message is either ACK or NACK, which are 0x79 and 0x1F, respectively.

## How can we order a command

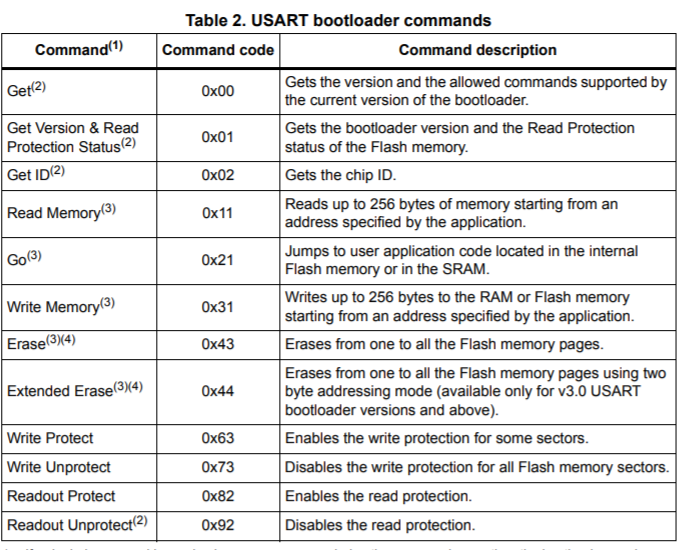
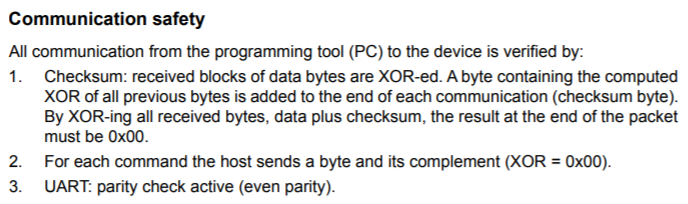


Figure 8 - Command List - [Souce](https://www.st.com/resource/en/application_note/cd00264342-usart-protocol-used-in-the-stm32-bootloader-stmicroelectronics.pdf#page=7)

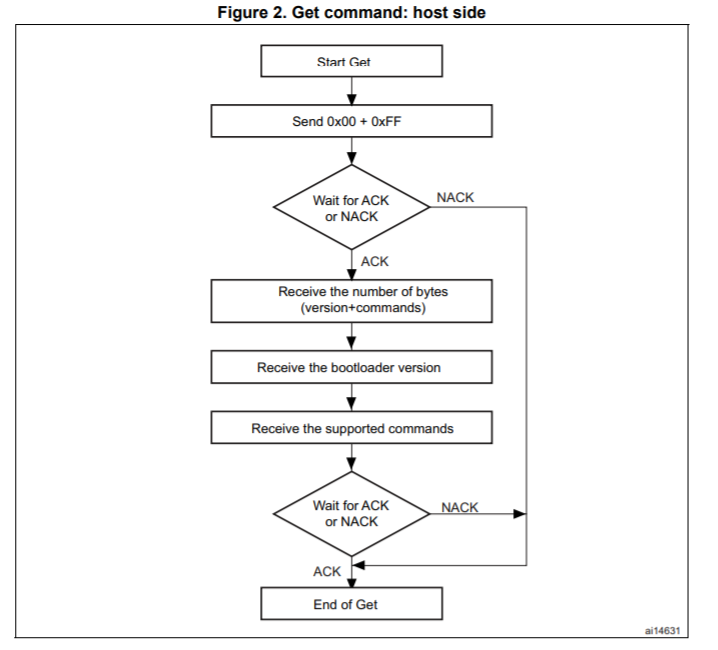
### Communication Safety

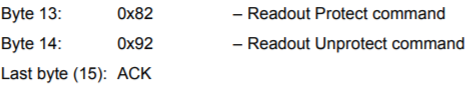
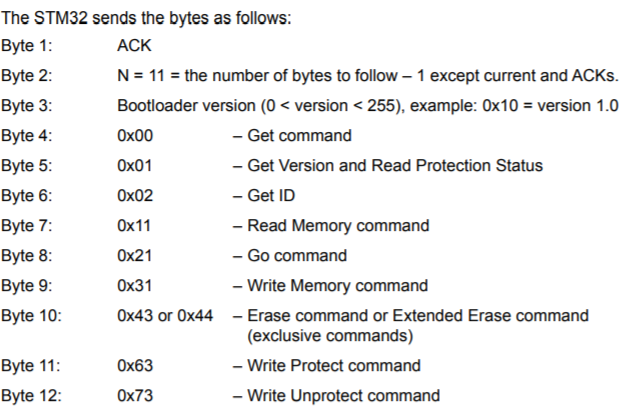


Hence we send our command byte followed by its complementary byte. For example, the “GET” command 0x00 is sent with its complement 0xFF, so that we establish secure communication.

### Receiving Information via Bootloader

There is a flowchart for the “GET” Command showing how the communication is handled.

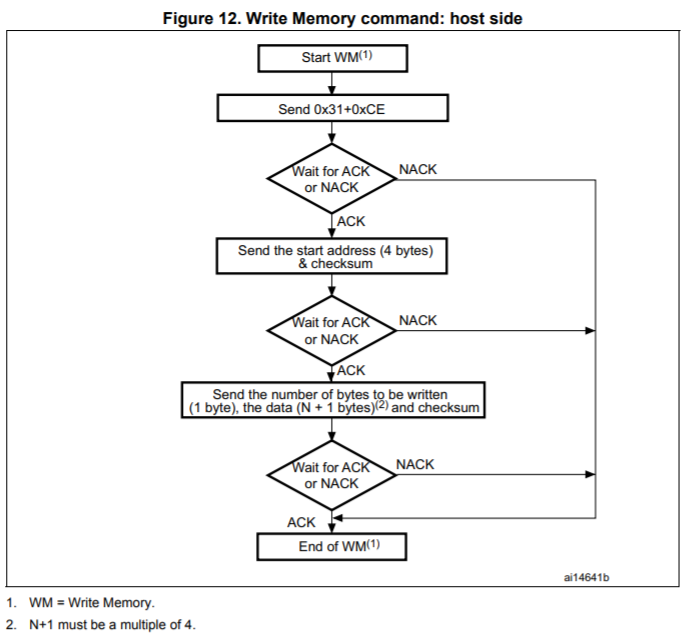


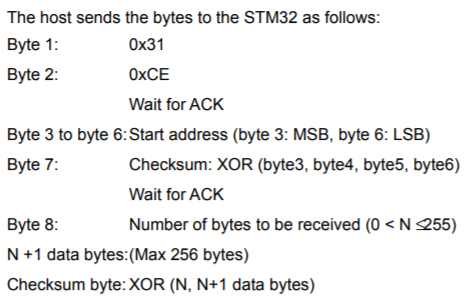


## How to Write our code - Write Memory command

The maximum length of the block to be written for the STM32 is 255 bytes, according to [AN3115](https://www.st.com/resource/en/application_note/cd00264342-usart-protocol-used-in-the-stm32-bootloader-stmicroelectronics.pdf#page=18).

If the Write Memory command is issued to the option byte area, all bytes are erased before writing the new values, and at the end of the command, the bootloader generates a system reset to take into account the new configuration of the option bytes.



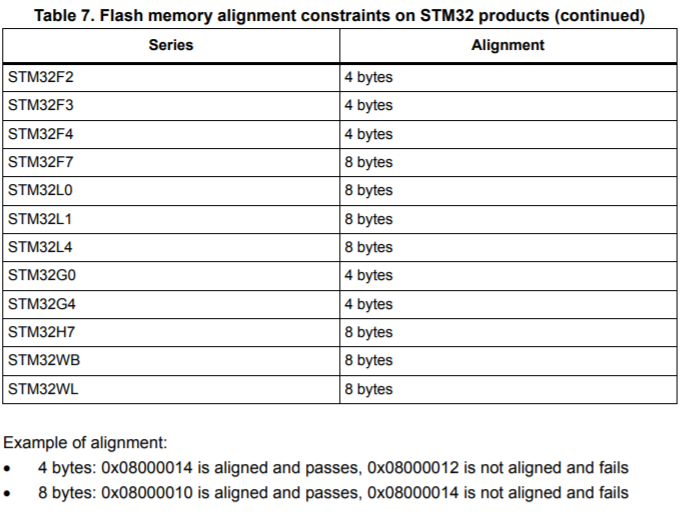


## Where to write our code

We cannot write the code directly to an arbitrary memory location. First, we need to compile and build the code to obtain HEX or BIN translation of the code. For codding IDE, I use [KEIL µVisionV5](http://www2.keil.com/mdk5/uvision/) Software. After we built the code, we obtain the HEX file, ready to be written.

The program must be written starting from the beginning of the FLASH Memory @0x08000000 memory address (See [Memory Mapping](#_Memory_Mapping)).

## Some constraints we need to obey



## Example Code to be Written

The code generated by KEIL uVision Software:



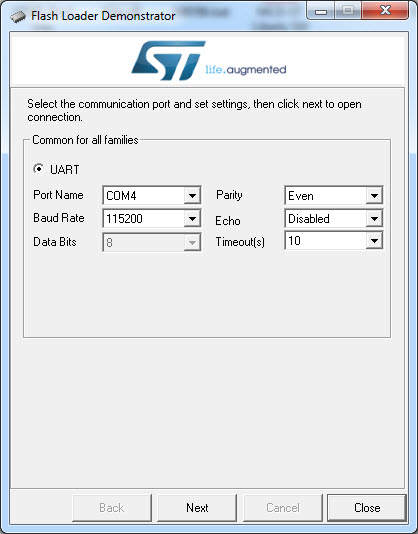
32 HEX\_CODED FORM == 16 BYTES

This code is obtained via Flasher Program, which is going to be discussed next section. The whole flash memory, in the first boot run, dumbed into code.hex and code.bin file.

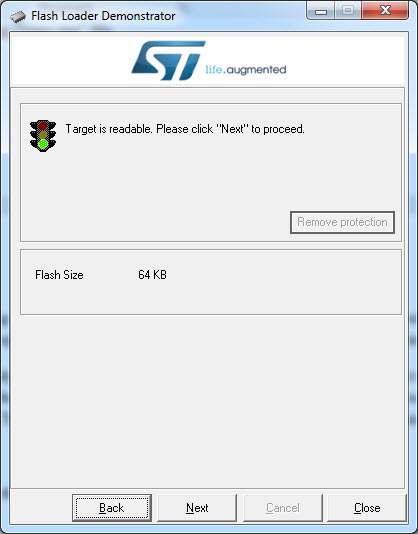


## An Example Program – STM32 Flasher

### Connection Properties



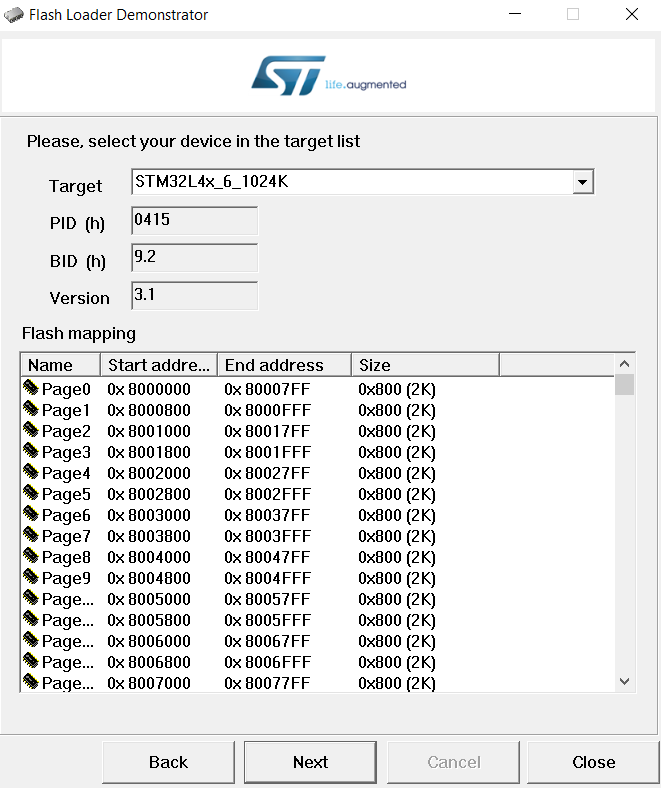
### Read or Write Protection



**PROTECTION:**

* **Read Protection**
* **Write Protection**

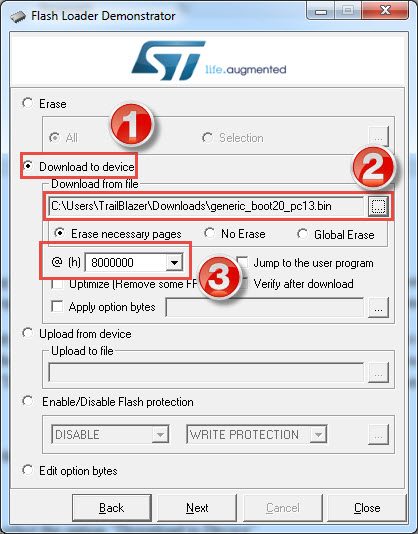
### GET Information from the device via bootloader



**FLASH MEMORY ADDRESS**

**GET CMD**

### WRITE CMD



## Utility Tools

### Hex Reader

The compiler creates HEX file of the compiled program that should be written to the user flash memory address to start the MCU with main program. An example for compiled HEc file is given in the [Example Code](#_Example_Code_to) section, above. Since the HEX file contains more informaiton, like memory address and memory page address, a utility tool should extract the program HEX Codes from the file in order to send via UART while programming the MCU.

The Python script I wrote uses IntelHex library. The \_BUFFER\_SIZE setting holds the max buffer size information. The HEX file is written, main program code is extracted then dived into chunks with size of \_BUFFER\_SIZE.

from intelhex import IntelHex  
  
# CONFIGURATIONS  
\_BUFFER\_SIZE = 256  
  
# IntelHex Object Initiation  
  
intelHex = IntelHex()  
  
# HEX FILE SELECTION  
  
\_file\_name = str(input("HEX File Name to be uploaded:"))  
  
# Get Data From HEX File  
intelHex.fromfile(\_file\_name+".hex", format='hex') # Read Hex File  
hex\_dict = intelHex.todict() # Dump into DICT Object  
  
hex\_byte\_list = list(hex\_dict.values()) # Convert to list  
hex\_byte\_list.pop() # POP: {'EIP': 134218121}  
print("FILE-Decimal Bytes:", hex\_byte\_list) # All bytes in decimal form  
  
hex\_chunk\_list = [hex\_byte\_list[i: i + \_BUFFER\_SIZE] # Creating new list: Chunk List  
 for i in range(0, len(hex\_byte\_list), \_BUFFER\_SIZE)] # Each containing specified many  
  
print("\nChunks:", hex\_chunk\_list) # See Chunks  
print('\n1st Chunk (HEX): [{}]'.format(', ' # See First Chunk in HEX  
 .join(hex(x) for x in hex\_chunk\_list[0]))) # HEX Conversion (CHECKING)  
  
# Some Other Information  
print("\nGeneral Code Information:")  
print("Total # of Bytes:\t\t", len(hex\_byte\_list))  
print("Buffer Size:\t\t\t", \_BUFFER\_SIZE)  
print("Total # of Chunks:\t\t", len(hex\_chunk\_list))  
print("# of Last Chunk bytes:\t", len(hex\_chunk\_list[-1]))

General Code Information:

Total # of Bytes: 7372

Buffer Size: 256

Total # of Chunks: 29

# of Last Chunk bytes: 204

### UART BootLoader Trigger

The first UART receive interrupt forces MCU to boot in bootloader mode. After MCU is in BootLoader mode, the communication can be establish by the guide of the protocol discussed in [Bootloader Process](#_What_Happens_in) section.

The Python code I wrote uses both time and serial libraries. Communicaiton constants are constructed as defined in [Commands](#_How_can_we) sections.

import serial  
from time import sleep  
  
  
# Color Class  
class Bcolors:  
 HEADER = '\033[95m'  
 OKBLUE = '\033[94m'  
 OKGREEN = '\033[92m'  
 WARNING = '\033[93m'  
 FAIL = '\033[91m'  
 ENDC = '\033[0m'  
 BOLD = '\033[1m'  
 UNDERLINE = '\033[4m'  
  
# CONFIGURATIONS  
\_BAUD\_RATE = 115200  
\_PORT = "COM5"  
\_SERIAL\_TIMEOUT = 10  
\_BYTE\_SIZE = 8  
\_STOP\_BITS = serial.STOPBITS\_ONE  
\_PARITY = serial.PARITY\_EVEN  
  
# COMM CONSTANTS  
\_ACK = b'\x79'  
\_NACK = b'\x1F'  
\_GET\_CMD = b'\x00\xFF'  
\_GV\_CMD = b'\x01\xFE'  
\_GED\_ID\_CMD = b'\x02\xFD'  
\_WRITE\_CMD = b'\x31\xCE'  
\_READ\_CMD = b'\x11\xEE'  
\_GO\_CMD = b'\x21\xDE'  
\_ERASE\_CMD = b'\x43\xBC'  
  
\_UART\_SELEC = b'\x7F'  
ACK\_counter = 0  
  
# Serial Object Init with proper parameters  
  
  
serialPort = serial.Serial(port=\_PORT,  
 baudrate=\_BAUD\_RATE,  
 timeout=\_SERIAL\_TIMEOUT,  
 stopbits=\_STOP\_BITS,  
 bytesize=\_BYTE\_SIZE,  
 parity=\_PARITY)  
  
# serialPort.open()  
# First Step: Trigger The USART1 RX:  
print(f"{Bcolors.HEADER}UART1 RX Interrupt:", \_ACK, f"{Bcolors.ENDC}")  
serialPort.write(\_ACK)  
sleep(0.5) # Sleep For 500ms to give some time to device  
  
# Second Step: UARTx Selection Command  
print(f"{Bcolors.OKBLUE}UARTx SELECTION CMD:", \_UART\_SELEC, f"{Bcolors.ENDC}")  
serialPort.write(\_UART\_SELEC)  
sleep(1) # Sleep For 1 sec to give some time to human  
  
# Third Step: Comm Check  
char = serialPort.read()  
if char == \_ACK:  
 print(f"{Bcolors.OKGREEN}Received ACK | UARTx SUCCESS{Bcolors.ENDC}")  
elif char == \_NACK:  
 print(f"{Bcolors.FAIL}Received NACK | UARTx FAILED{Bcolors.ENDC}")  
sleep(1) # Sleep For 1 sec to give some time to human  
  
# Forth Step: Get Command  
print(f"{Bcolors.OKBLUE}Sending GET CMD:{Bcolors.ENDC}", \_GV\_CMD)  
serialPort.write(\_GV\_CMD)  
sleep(1)  
while True:  
 char = serialPort.read()  
 if char == \_NACK:  
 print(f"{Bcolors.FAIL}Received NACK | CMD FAILED{Bcolors.ENDC}")  
 break  
 elif char == \_ACK and ACK\_counter == 0:  
 print(f"{Bcolors.OKGREEN}Received ACK | CMD STARTED{Bcolors.ENDC}")  
 ACK\_counter += 1  
 elif char == \_ACK and ACK\_counter:  
 print(f"{Bcolors.OKGREEN}Received ACK | CMD SUCCESS{Bcolors.ENDC}")  
 break  
 else:  
 print(f"{Bcolors.HEADER}Response:", char, f"{Bcolors.ENDC}")