# USART - BootLoader

SOLUTION TO THE PROBLEM
AN EXAMPLE PROGRAM

Ataberk ÖKLÜ
METU

# Table of Contents

# The bootloader via USART / UART interface

## Description of the problem

The target device has only three UART pins accessible from outside world, due to security and safety issues. However, built-in Bootloader can only be accessed from dedicated boot0 pin, which is not available for our case. Therefore, the solution should bypass the traditional way and trigger to bootloader mode from UART pins.

| Pattern | Condition |
|---|---|
| Pattern 1 | Boot0(pin) = 1 and Boot1(pin) = 0 |
| Pattern 2 | Boot0(pin) = 1 and nBoot1(bit) = 1 |
| Pattern 3 | Boot0(pin) = 1, Boot1(pin) = 0 and BFB2(bit) = 1 |
| | Boot0(pin) = 0, BFB2(bit) = 0 and both banks do not contain valid code |
| | Boot0(pin) = 1, Boot1(pin) = 0, BFB2(bit) = 0 and both banks do not contain valid code |
| Pattern 4 | Boot0(pin) = 1, Boot1(pin) = 0 and BFB2(bit) = 1 |
| | Boot0(pin) = 0, BFB2(bit) = 0 and both banks do not contain valid code |
| | Boot0(pin) = 1, Boot1(pin) = 0 and BFB2(bit) = 0 |
| Pattern 5 | Boot0(pin) = 1, Boot1(pin) = 0 and BFB2(bit) = 0 |
| | Boot0(pin) = 0, BFB2(bit) = 1 and both banks do not contain valid code |
| | Boot0(pin) = 1, Boot1(pin) = 0 and BFB2 (bit) = 1 |
| Pattern 6 | Boot0(pin) = 1, nBoot1(bit) = 1 and nBoot0_SW(bit) = 1 |
| | nBoot0(bit) = 0, nBoot1(bit) = 1 and nBoot0_SW(bit) = 0 |
| | Boot0(pin) = 0, nBoot0_SW(bit) = 1 and main Flash memory empty |
| | nBoot0(bit) = 1, nBoot0_SW(bit)=0 and main Flash memory empty |
| Pattern 7 | Boot0(pin) = 1, nBoot1(bit) = 1 and BFB2(bit) = 0 |
| | Boot0(pin) = 0, BFB2(bit) = 1 and both banks do not contain valid code |
| | Boot0(pin) = 1, nBoot1(bit) = 1 and BFB2(bit) = 1 |
| Pattern 8 | Boot(pin) = 0 and BOOT_ADD0(optionbyte) = 0x0040 |
| | Boot(pin) = 1 and BOOT_ADD1(optionbyte) = 0x0040 |

*Figure 1 - BootLoader Activation Patterns - Source*

The target MCU is STM32L476. This MCU has Pattern 7 to access bootloader mode.

| Boot mode selection | | Boot mode |
|---|---|---|
| nBOOT1 (option bit) | BOOT0 (pin) | |
| X | 0 | User Flash memory |
| 1 | 1 | System memory (bootloader) |
| 0 | 1 | SRAM1 |

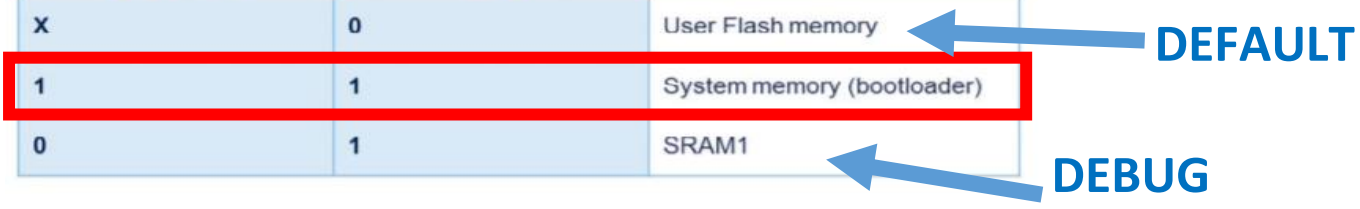
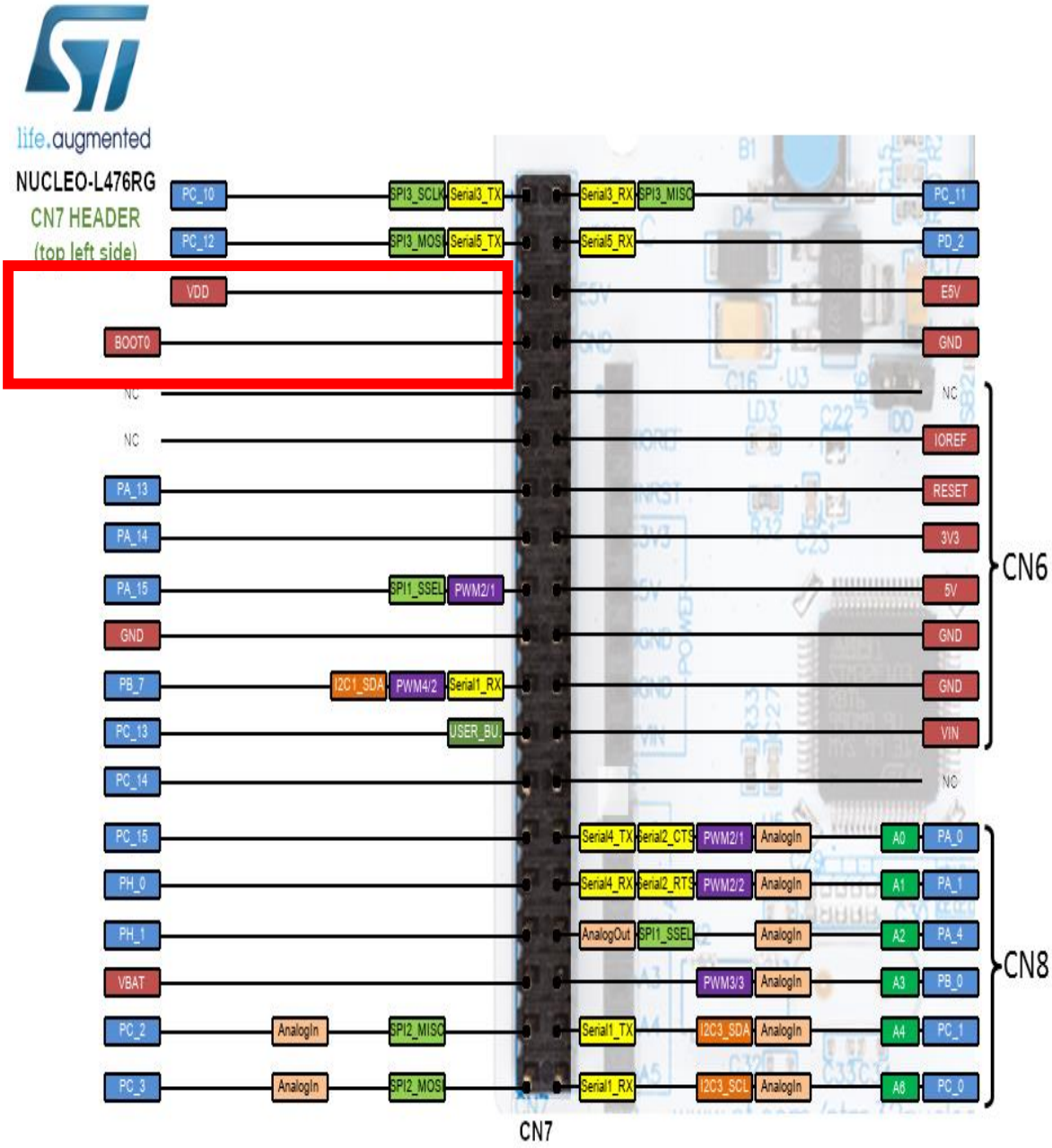



*Figure 2 - Boot Modes - Source*

The Figure 2 shows us that when boot0 pin is low, the MCU start with the User Flash Memory which hold the main program. In order to boot the MCU in bootloader mode, which is kept in System Memory, we need to set boot0 pin. The default of the nboot1 register is set.

## 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). 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).

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), 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

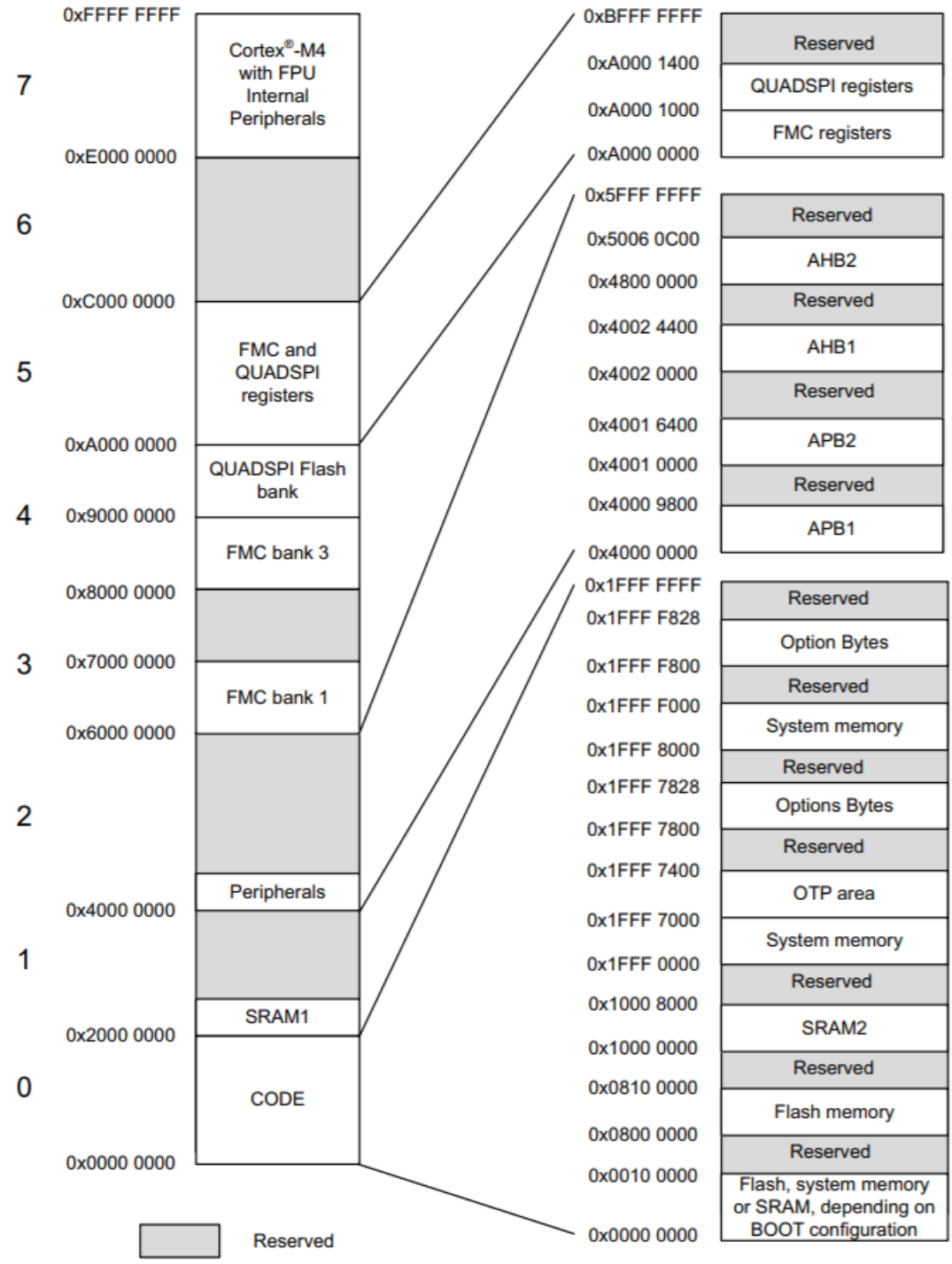



*Figure 3 - Memory Map - Source*

| Addresses | Boot/remap in main Flash memory | Boot/remap in embedded SRAM 1 | Boot/remap in system memory | Remap in FSMC | Remap in QUADSPI |
|---|---|---|---|---|---|
| 0x2000 0000 - 0x2001 7FFF | SRAM1 | SRAM1 | SRAM1 | SRAM1 | SRAM1 |

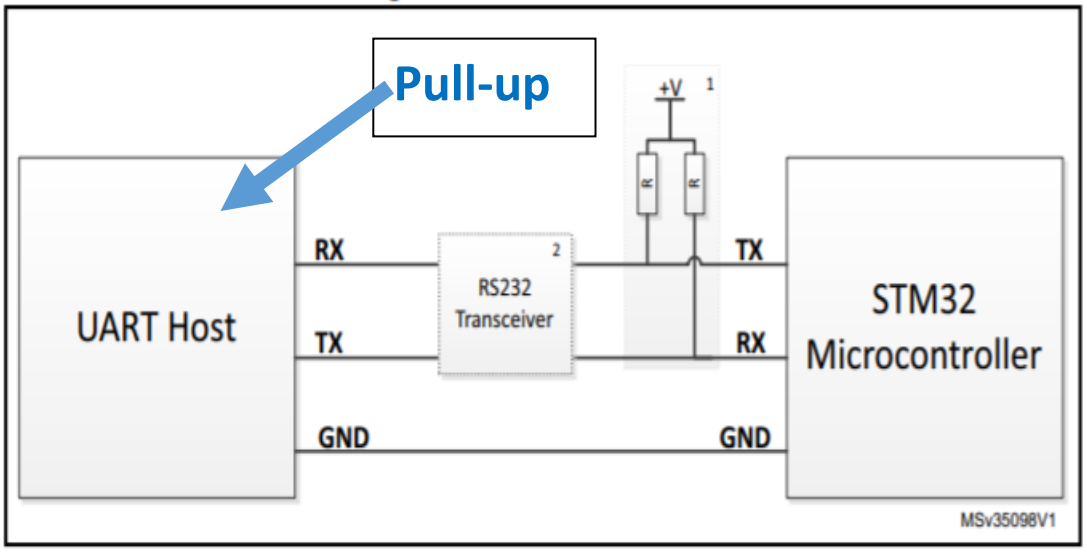
*Figure 4 - SRAM1 Memory Addresses in different boots - Source 1 – Source 2*

## How to Connect Devices

### Hardware connection requirements

To use the USART bootloader, the host must be connected to the RX and TX pins of the desired USARTx interface via a serial cable.

**Figure 1. USART connection**




1. A pull-up resistor must be added, if pull-up resistor are not connected in host side.
2. An RS232 transceiver must be connected to adapt voltage level (3.3 to 12 V) between STM32 device and host.

*+V typically is 3.3 V and R typically 100 KΩ.These values depend upon the application and the used hardware.*

To use the DFU, connect the microcontroller USB interface to a USB host (i.e. a PC).

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.

## 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:

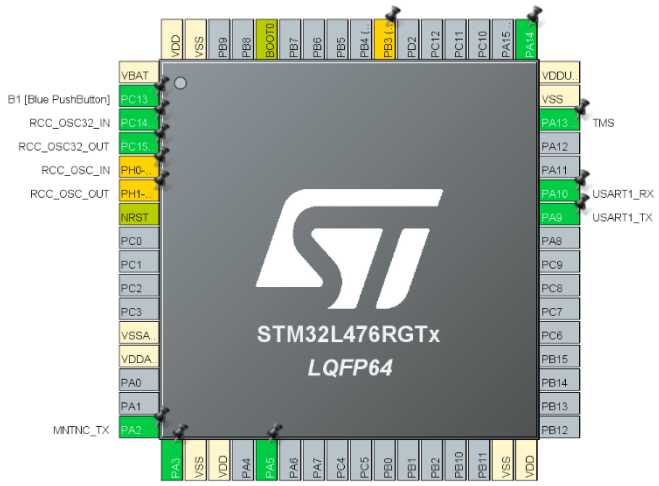
| Protocol | I/Os and Comments | Comments |
|---|---|---|
| **USART** | USART1 on pins PA9/PA10<br>USART2 on pins PA2/PA3<br>USART3 on pins PC10/PC11 | |
| **USB** | USB DFU interface on pins PA11/PA12 | Bootloader checks if HSE present : USB clock is HSE<br>If no Bootloader checks if LSE present : USB clock is MSI auto-trimmed with LSE |
| **CAN** | CAN1 on pins PB8/PB9 | |
| **SPI** | SPI1 on pins PA4/PA5/PA6/PA7<br>SPI2 on pins PB12/PB13/PB14/PB15 | |
| **I2C** | I2C1 on pins PB6/PB7<br>I2C2 on pins PB10/PB11<br>I2C3 on pins PC0/PC1 | I²C slave address is 0x86 |

*Figure 5 - BootLoader Communication Protocols - Source*

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

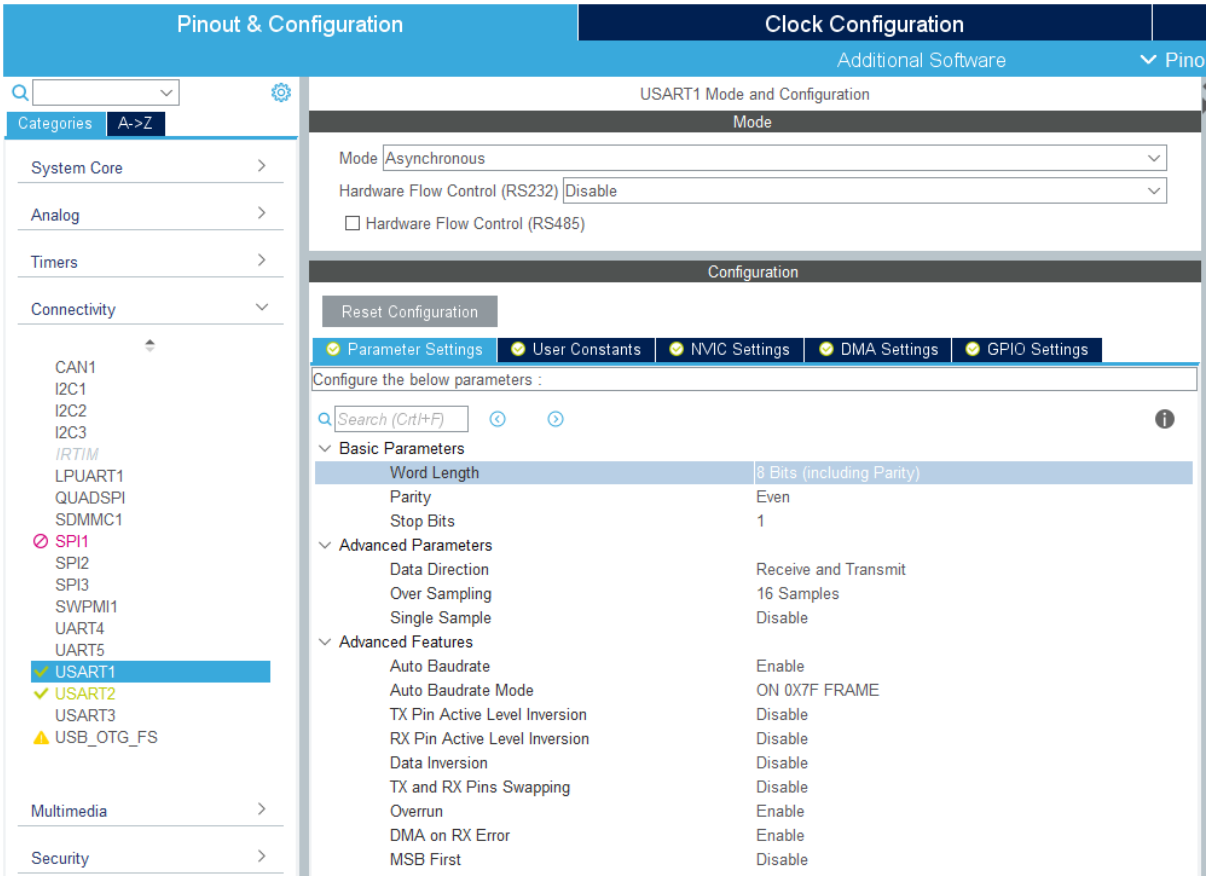
| Bootloader | Feature/Peripheral | State | Comment |
|---|---|---|---|
| Common to all bootloaders | RCC | HSI enabled | The system clock frequency is 72 MHz (using the PLL clocked by HSI) |
| | | - | The clock recovery system (CRS) is enabled for the DFU bootloader to allow USB to be clocked by HSI48 48 MHz |
| | RAM | - | 16 Kbyte starting from address 0x20000000 are used by the bootloader firmware |
| | System memory | - | 28 Kbyte starting from address 0x1FFF0000, contain the bootloader firmware |
| | IWDG | - | The independent watchdog (IWDG) prescaler is configured to its maximum value. It is periodically refreshed to prevent watchdog reset (in case the hardware IWDG option was previously enabled by the user). |
| Securable memory area | - | - | The address to jump to the exit securable memory area @0x1FFF6800 |
| USART1 bootloader | USART1 | Enabled | Once initialized the USART1 configuration is: 8-bit, even parity and 1 Stop bit |
| | USART1_RX pin | Input | PA10 pin: USART1 in reception mode |
| | USART1_TX pin | Output | PA9 pin: USART1 in transmission mode |
| USART2 bootloader | USART2 | Enabled | Once initialized the USART2 configuration is: 8-bit, even parity and 1 Stop bit |
| | USART2_RX pin | Input | PA3 pin: USART2 in reception mode |
| | USART2_TX pin | Output | PA2 pin: USART2 in transmission mode |
| USART3 bootloader | USART3 | Enabled | Once initialized the USART3 configuration is: 8-bit, even parity and 1 Stop bit |
| | USART3_RX pin | Input | PC11 pin: USART3 in reception mode |
| | USART3_TX pin | Output | PC10 pin: USART3 in transmission mode |

*Figure 6 - Detailed Explanations For USART Connection - Source*




As stated, first, we need to initialize our USART1 in proper settings. To do this, we facilitate an ST software called [STM32CubeMX](). 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:

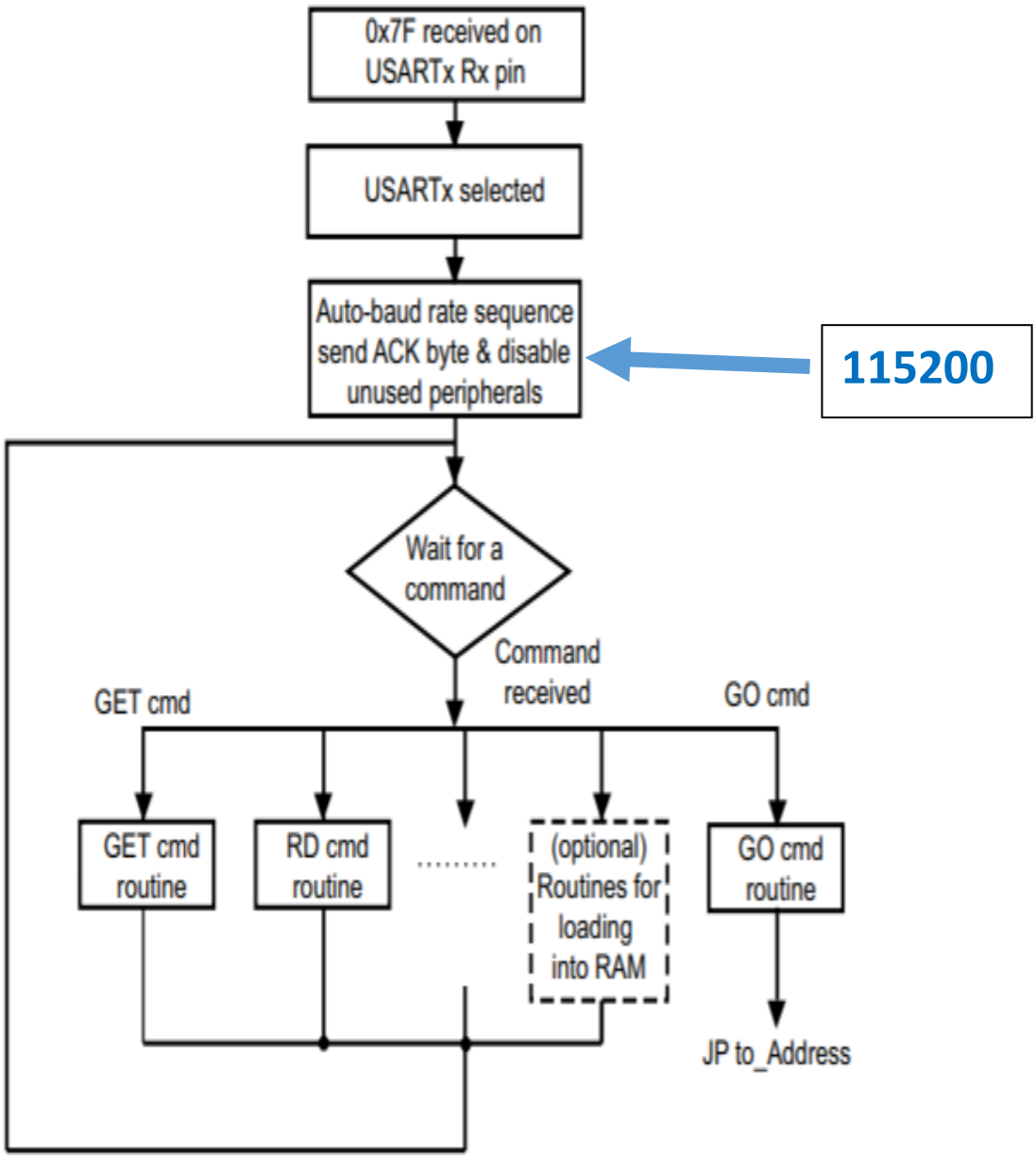



*Figure 7 - BootLoader Protocol Selection - UART - Source*

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, the returned message is either ACK or NACK, which are 0x79 and 0x1F, respectively.

## How can we order a command

| Command[1] | Command code | Command description |
|---|---|---|
| Get[2] | 0x00 | Gets the version and the allowed commands supported by the current version of the bootloader. |
| Get Version & Read Protection Status[2] | 0x01 | Gets the bootloader version and the Read Protection status of the Flash memory. |
| Get ID[2] | 0x02 | Gets the chip ID. |
| Read Memory[3] | 0x11 | Reads up to 256 bytes of memory starting from an address specified by the application. |
| Go[3] | 0x21 | Jumps to user application code located in the internal Flash memory or in the SRAM. |
| Write Memory[3] | 0x31 | Writes up to 256 bytes to the RAM or Flash memory starting from an address specified by the application. |
| Erase[3][4] | 0x43 | Erases from one to all the Flash memory pages. |
| Extended Erase[3][4] | 0x44 | Erases from one to all the Flash memory pages using two byte addressing mode (available only for v3.0 USART bootloader versions and above). |
| Write Protect | 0x63 | Enables the write protection for some sectors. |
| Write Unprotect | 0x73 | Disables the write protection for all Flash memory sectors. |
| Readout Protect | 0x82 | Enables the read protection. |
| Readout Unprotect[2] | 0x92 | Disables the read protection. |

*Figure 8 - Command List - Souce*

## Communication Safety

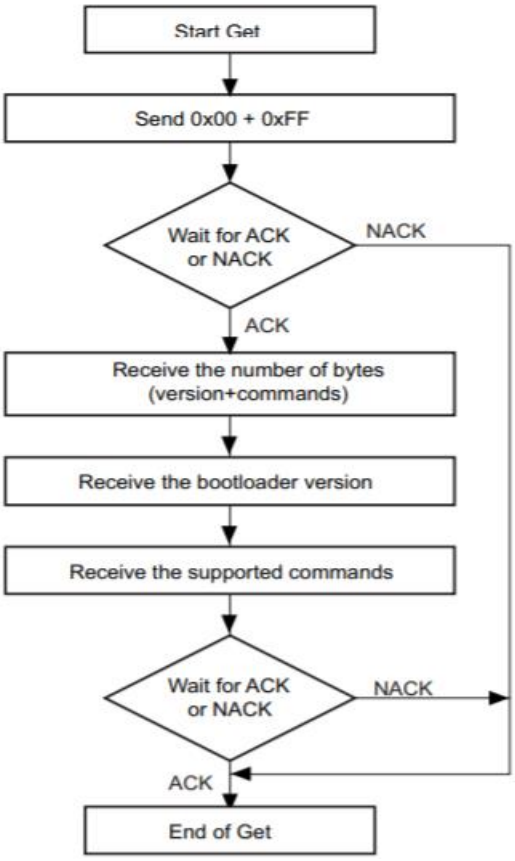
All communication from the programming tool (PC) to the device is verified by:

1. Checksum: received blocks of data bytes are XOR-ed. A byte containing the computed XOR of all previous bytes is added to the end of each communication (checksum byte). By XOR-ing all received bytes, data plus checksum, the result at the end of the packet must be 0x00.
2. For each command the host sends a byte and its complement (XOR = 0x00).
3. UART: parity check active (even parity).

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.



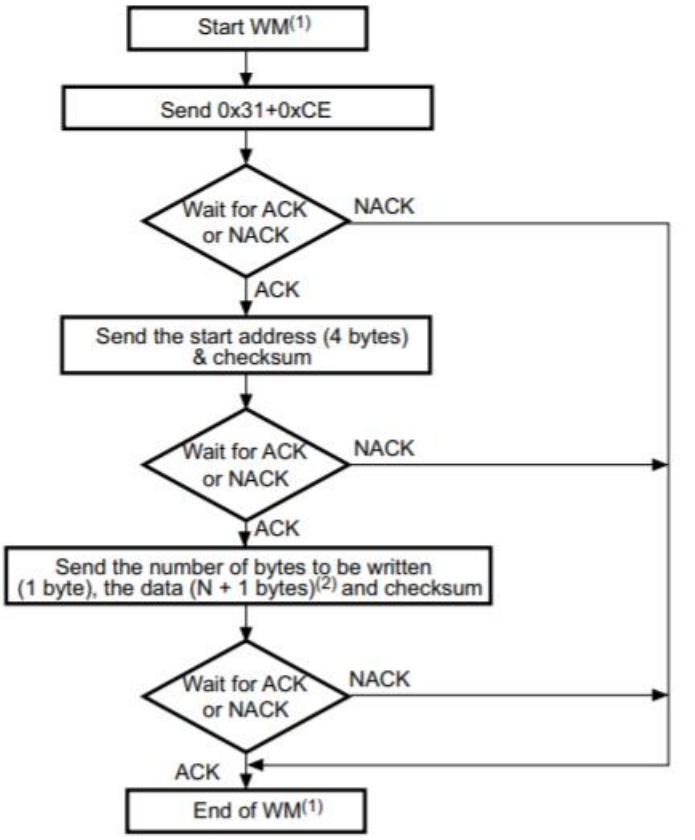

The STM32 sends the bytes as follows:

| | | |
|---|---|---|
| Byte 1: | ACK | |
| Byte 2: | N = 11 = the number of bytes to follow – 1 except current and ACKs. | |
| Byte 3: | Bootloader version (0 < version < 255), example: 0x10 = version 1.0 | |
| Byte 4: | 0x00 | – Get command |
| Byte 5: | 0x01 | – Get Version and Read Protection Status |
| Byte 6: | 0x02 | – Get ID |
| Byte 7: | 0x11 | – Read Memory command |
| Byte 8: | 0x21 | – Go command |
| Byte 9: | 0x31 | – Write Memory command |
| Byte 10: | 0x43 or 0x44 | – Erase command or Extended Erase command (exclusive commands) |
| Byte 11: | 0x63 | – Write Protect command |
| Byte 12: | 0x73 | – Write Unprotect command |
| Byte 13: | 0x82 | – Readout Protect command |
| Byte 14: | 0x92 | – Readout Unprotect command |
| Last byte (15): | ACK | |

## 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.

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.




WM = Write Memory.

N+1 must be a multiple of 4.

The host sends the bytes to the STM32 as follows:

| | |
|---|---|
| Byte 1: | 0x31 |
| Byte 2: | 0xCE |
| | Wait for ACK |
| Byte 3 to byte 6: | Start address (byte 3: MSB, byte 6: LSB) |
| Byte 7: | Checksum: XOR (byte3, byte4, byte5, byte6) |
| | Wait for ACK |
| Byte 8: | Number of bytes to be received (0 < N ≤255) |
| N +1 data bytes: | (Max 256 bytes) |
| Checksum byte: | XOR (N, N+1 data 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 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).

## Some constraints we need to obey

**Table 7. Flash memory alignment constraints on STM32 products (continued)**

| Series | Alignment |
|---|---|
| STM32F2 | 4 bytes |
| STM32F3 | 4 bytes |
| STM32F4 | 4 bytes |
| STM32F7 | 8 bytes |
| STM32L0 | 8 bytes |
| STM32L1 | 8 bytes |
| STM32L4 | 8 bytes |
| STM32G0 | 4 bytes |
| STM32G4 | 4 bytes |
| STM32H7 | 8 bytes |
| STM32WB | 8 bytes |
| STM32WL | 8 bytes |

Example of alignment:

- 4 bytes: 0x08000014 is aligned and passes, 0x08000012 is not aligned and fails
- 8 bytes: 0x08000010 is aligned and passes, 0x08000014 is not aligned and fails

## Example Code to be Written

The code generated by KEIL uVision Software:

```
:020000040800F2
:10000000980400209D010008FB13000821130008 3C
:10001000F9130008B1020008C11B000800000002 2D
:10002000000000000000000000000000B1160008 01
:10003000B302000800000000FD130008B3160008 1A
:10004000B7010008B7010008B7010008B7010008 B0
:10005000B7010008B7010008B7010008B7010008 A0
:10006000B7010008B7010008B7010008B7010008 90
:10007000B7010008B7010008B7010008B7010008 80
:10008000B7010008B7010008B7010008B7010008 70
:10009000B7010008B7010008B7010008B7010008 60
:1000A000B7010008B7010008B7010008B7010008 50
:1000B000B7010008B7010008B7010008B7010008 40
:1000C000B7010008B7010008B7010008B7010008 30
:1000D000B7010008B7010008B7010008B7010008 20
:1000E000B5020008B7010008B7010008B7010008 11
:1000F000B7010008B7010008B7010008B7010008 00
:10010000B7010008B7010008B7010008B7010008 EF
:10011000B7010008B7010008B7010008B7010008 DF
:10012000B7010008B7010008B7010008B7010008 CF
:10013000B7010008B7010008B7010008B7010008 BF
:10014000B7010008B7010008B7010008B7010008 AF
:10015000B7010008B7010008B7010008B7010008 9F
:10016000B7010008B7010008B7010008B7010008 8F
:10017000B7010008B7010008B7010008000000003F
:10018000B7010008B7010008DFF80CD000F07EF8D6
:1001900000480047011C00089804002006488047DA
:1001A00006480047FEE7FEE7FEE7FEE7FEE7FEE75C
```

el HEX binary data | length : 20.788 | lines : 465 | Ln : 2 | Col : 10 | Sel : 32 | 1

32 HEX_CODED FORM == 16 BYTES

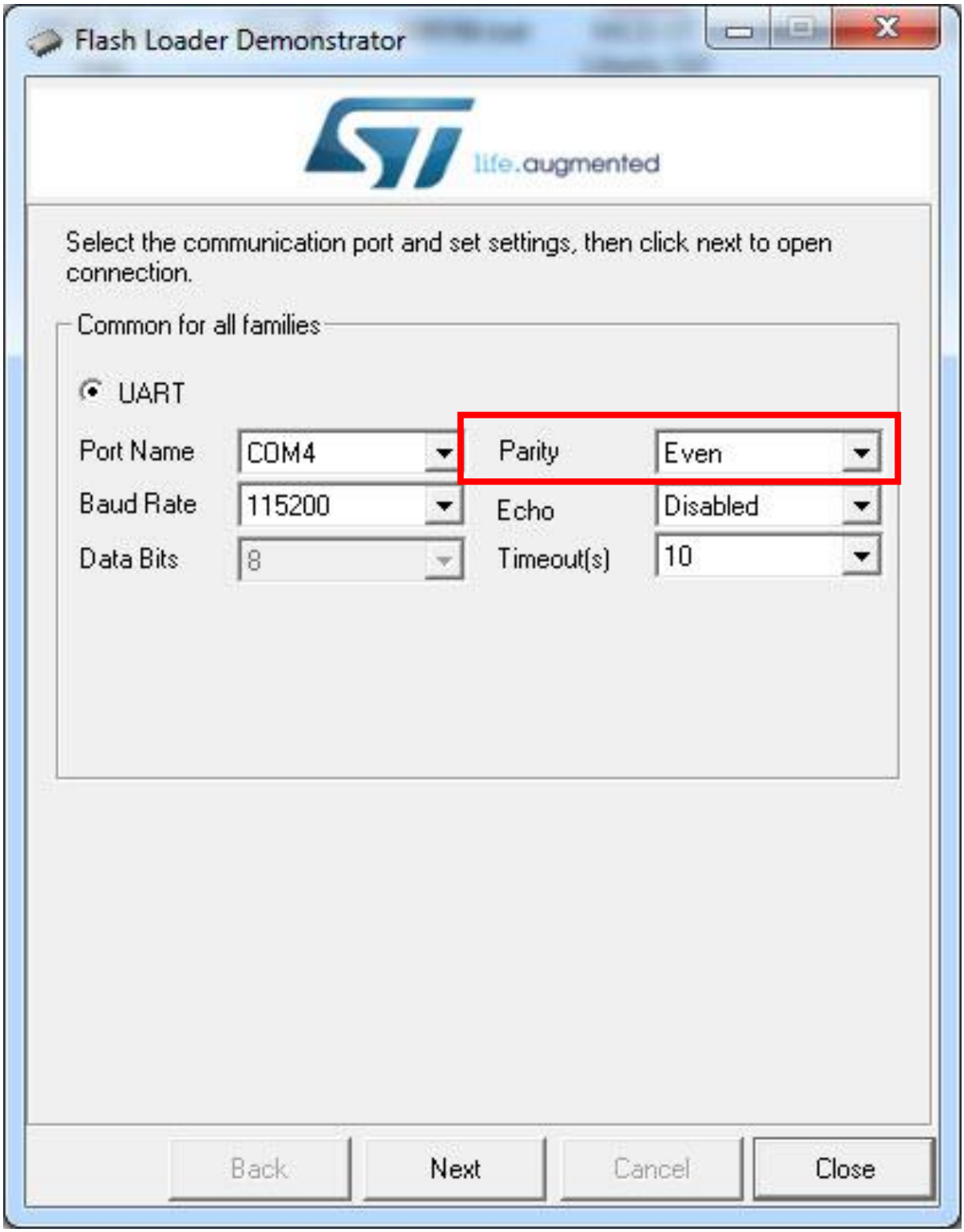
This code is obtained via Flasher Program, which is going to be discussed next section. The whole flash memory, in the first boot run, dumbed into code.hex and code.bin file.

```
:020000040800F2
:020000020000FC
:20000000E00400209D010008EB190008F7180008E919000869050008 4D22000800000001C
:20002000000000000000000000000000351D00086B050008000000 00ED190008371D000884
:20004000B7010008B7010008B7010008B7010008B7010008B7010008B7010008B7010008A0
:20006000B7010008B7010008B7010008B7010008B7010008B7010008B7010008B701000880
:20008000B7010008B7010008B7010008B7010008B7010008B7010008B7010008B701000860
:2000A000B7010008B7010008B7010008B7010008B7010008B7010008B7010008B701000840
:2000C000B7010008B7010008B7010008B7010008B7010008B7010008B7010008B701000820
:2000E000B7010008B7010008B7010008B7010008B7010008B7010008B7010008B701000800
:20010000B7010008B7010008B7010008B7010008B7010008B7010008B7010008B7010008DF
:20012000B7010008B7010008B7010008B7010008B7010008B7010008B7010008B7010008BF
:20014000B7010008B7010008B7010008B7010008B7010008B7010008B7010008B701000089F
:20016000B7010008B7010008B7010008B7010008B7010008B7010008000000003F
:20018000B7010008B7010008DFF80CD000F0DAF900480047 1D2400 08E00400200648804778
:2001A00006480047FEE7FEE7FEE7FEE7FEE7FEE7FEE7FEE7FEE7FEE7B11D00088901000850
:2001C0002DE9F05F05460020924 69B46884606468146402 41BE028464146474622460 0F07C
:2001E0000CF95346 5A46C01A914110D3114618462246 00F0F3F82D1A67EB01084F4622469B
:20020000012000210 0F0EAF817EB00094E41201EA4F10104DFDC484631462A4643460 0F0AF
:2002200000B830BCBDE8C09FD2B201E000F8012B491EFBD270470022F6E710B513460A4635
:20024000044619 46FFF7F0FF204610BD421C10F8011B0029FBD1801A70470346 11F8012B97
:2002600000F8012B002AF9D1184670472DE9FE4F81EA030404F0004421F000410094 4FF01F
:2002800000 0B23F00043 50EA01045ED052EA03045BD0C3F30A54C1F30A552C44A4F2F334CE
:2002A000019 4A0FB0254C1F3130141F48011C3F3130343F4801301FB024400FB034E840A78
:2002C000970A44EA815447EA8357A4FB076802958D0A05FB07854FEA932C04FB0C54270524
:2002E000029D4FEA065847EA1637B5EB08056EEB070C870E920E47EA811742EA8312A7FBD0
:20030000020 1B6EB0B0164EB00042B0D43EA0C335E1844EB1C50DA465146E7FB0201C5F3D1
:20032000013044FEA0B3343EA14534FEA043201 9C43EA0603A4F10C04029400 9CCDE900B418
```

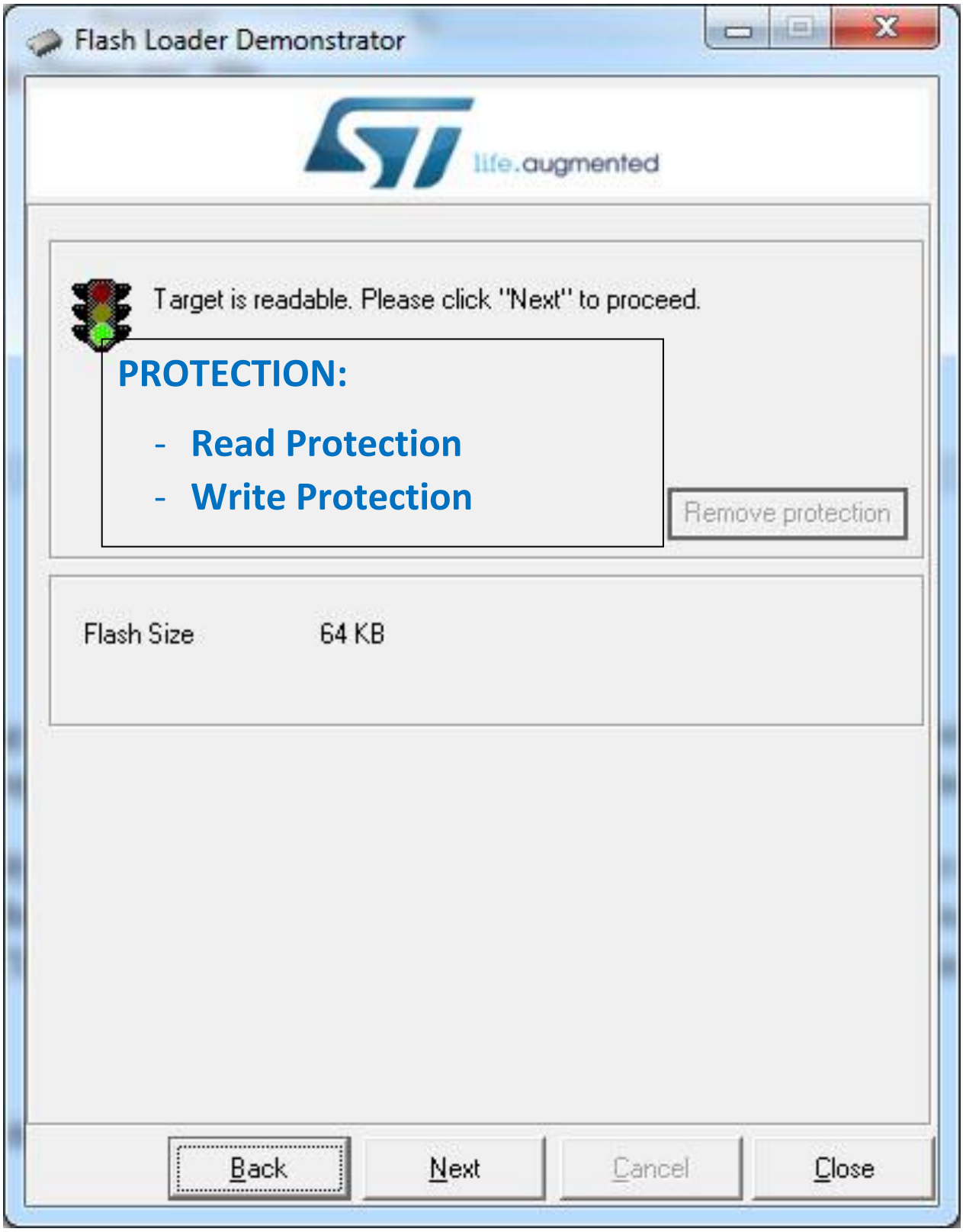
el HEX binary data | length : 2.540.824 | lines : 33.801 | Ln : 11 | Col : 74 | Sel : 64 | 1

## An Example Program – STM32 Flasher

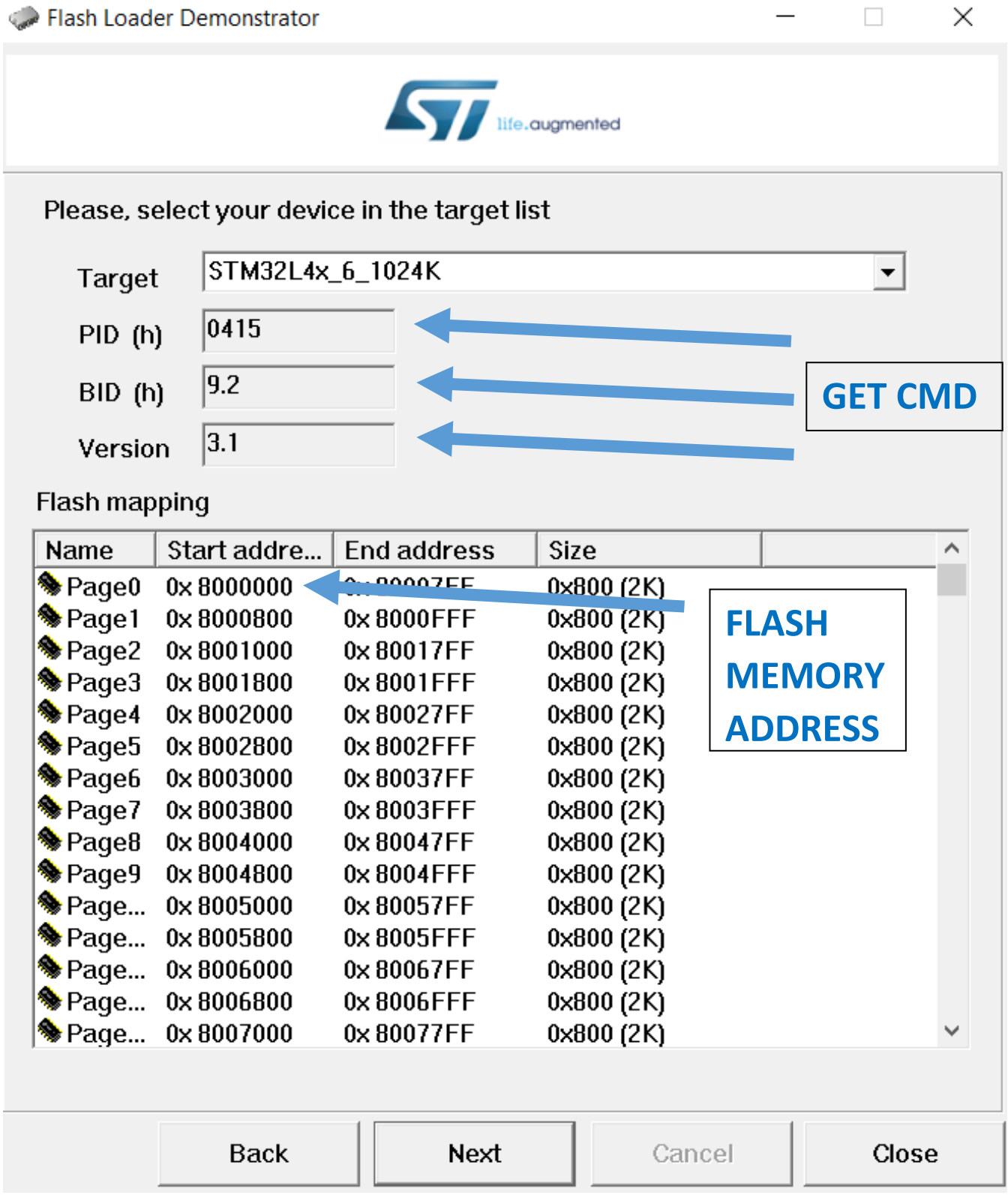
### Connection Properties



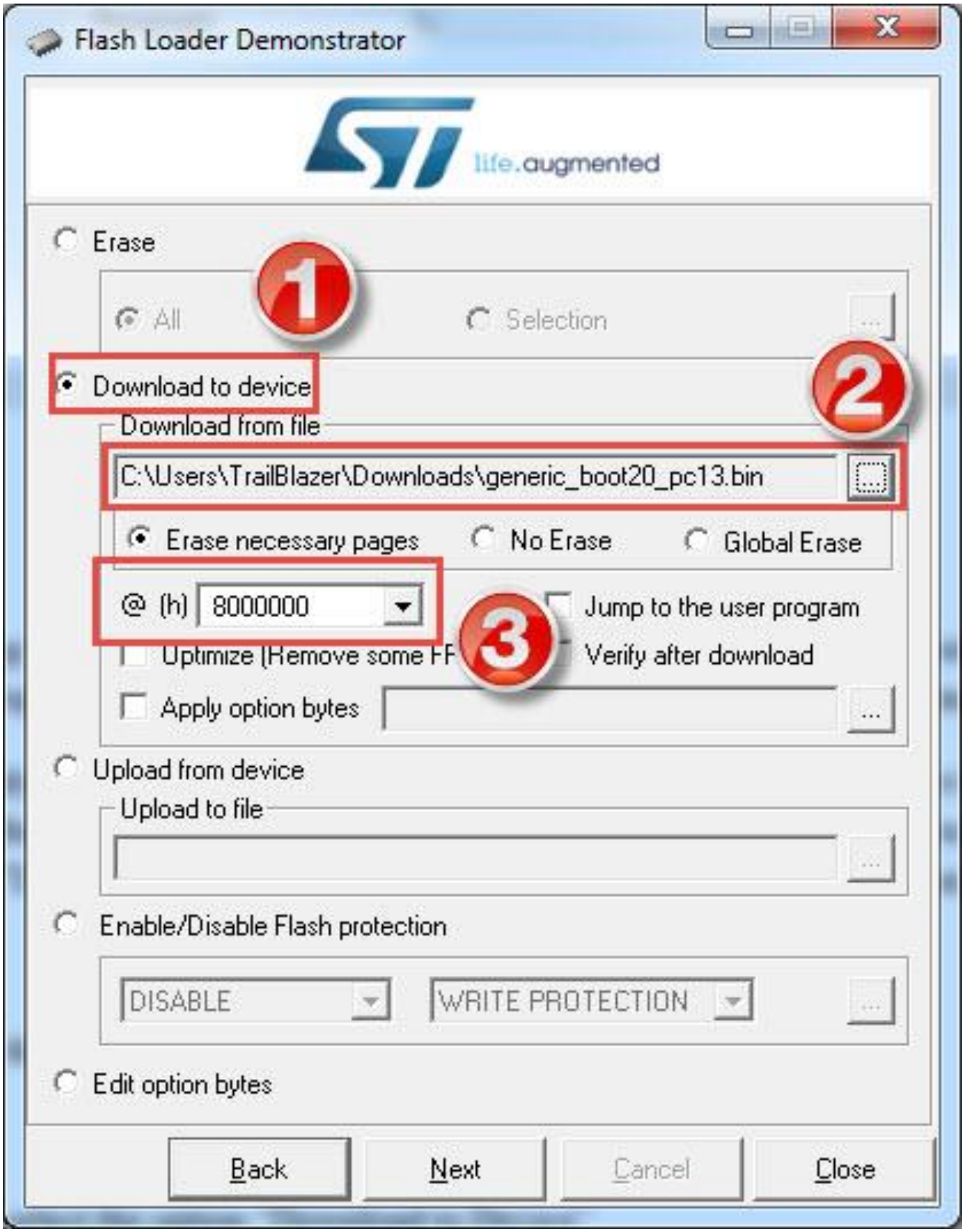
Read or Write Protection



GET Information from the device via bootloader



WRITE CMD


Flash Loader Demonstrator
life.augmented
Erase
All
Selection
1
Download to device
2
Download from file
C:\Users\TrailBlazer\Downloads\generic_boot20_pc13.bin
Erase necessary pages
No Erase
Global Erase
@ (h)
8000000
3
Jump to the user program
Optimize (Remove some FF
Verify after download
Apply option bytes
Upload from device
Upload to file
Enable/Disable Flash protection
DISABLE
WRITE PROTECTION
Edit option bytes
Back
Next
Cancel
Close

## 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 HEX file is given in the Example Code 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 section.

The Python code I wrote uses both time and serial libraries. Communicaiton constants are constructed as defined in Commands 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}")
```