UNIVERSITÉ SAVOIE MONT BLANC

Bootloader

Hardware used for this lecture:

- A STM32F446 Nucleo board + mini USB cable
- A serial-USB bridge + mini USB cable
- 2 wire female-female wires (or two jumpers)

1 Initial Problem

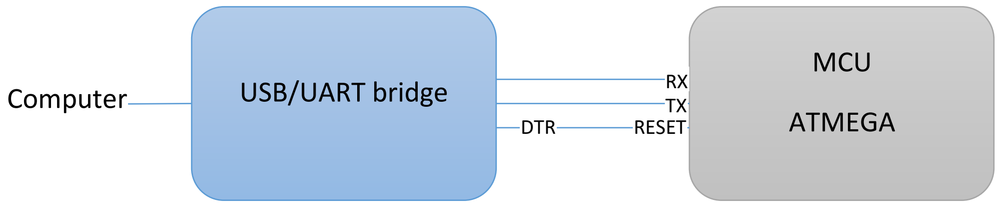
There are 2 ways to program a microcontroller:

1. With a programmer: **SWD** for STM32 (ST Link is the Debugger), **ICSP** for microchip
2. With a bootloader: A bootloader is a small piece of code stored in the MCU Flash. It runs first and checks if the user wants to update the application running on the MCU.

2 What is a bootloader?

2.1 Two examples

1. Arduino: On every reset the bootloader runs. Then the Arduino IDE sends the user application through the UART. Finally, there is another reset, and the application runs. The first reset is managed by the RTS signal.



2. STM32: It comes with a bootloader in its flash memory, but it has to be activated by hardware (Pin BOOT0 and BOOT1).

2.2 The STM32F446 memory

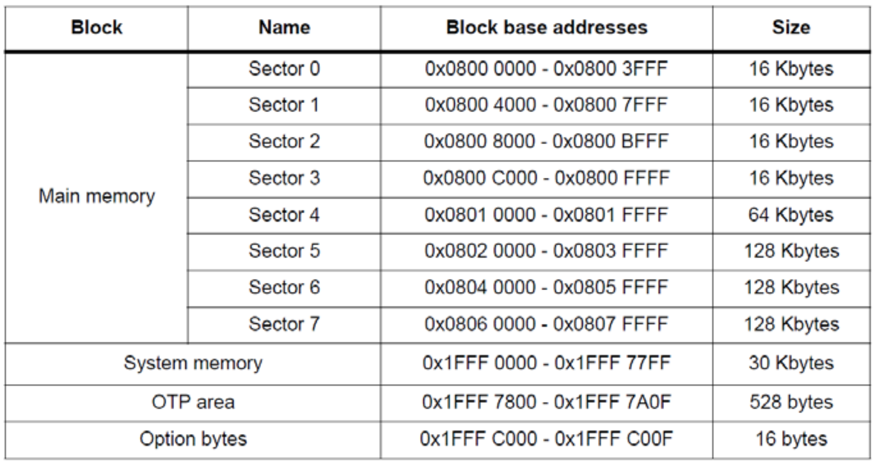
2.2.1 Memories available

- **512 Kb Flash:** Program, constants, interrupt vector table.
- **112 Kb SRAM1:** Heap, stack, data, bss.
- **16 Kb SRAM2:** Heap, stack, data, bss.
- **30 Kb System Memory (ROM): This is where the bootloader is!**

- **528 bytes OTP**: One Time Programmable.
- **16 bytes** of Optional bytes
- **4Kb of Backup RAM:** Ram battery powered for very low power application.

2.2.2 STM32F446 memory map

Details of the Flash memory:



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

Let's check the 0x0800 0000 address of the flash memory with the STM32CubeIDE memory browser

2.3 Reset Sequence

The first 2 steps of the reset sequence are done by hardware:

1. The MCU considers the value at the address 0x0800 0000 as the MSP (Main Stack Pointer). In our case, the stack pointer is therefore 0x20020000 (last address of the RAM memory)
2. The PC is loaded with the Reset_Handler (reset vector) stored at the address 0x0800 0004. This is where the start-up code is.

We can check the Reset handler address in the PC with a breakpoint on the start-up code (View > Register).

Note: Actually, the MCU considers its start address as 0x0000 0000. But this is a processor called memory aliasing which targets the same physical address 0x0800 0000. So 0x0000 000 = 0x0800 0000.

Note: Addresses of all function pointers (like interrupt vectors) are always incremented by one to specify that the targeted instruction is a thumb (ARM) instruction.

At the end of the startup code, the main function is called (bl main).

2.4 Boot Mode

The MCU can start from different memory depending on the BOOT0 and BOOT1 Pins status.

Table 2. Boot modes

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

On the Nucleo board, BOOT1 is already grounded, so we just need to set BOOT0 (CN7 PIN7) to 3V3 (CN7 PIN5) with a jumper.

The bootloader is a special code from ST that answer to specific command. Only a Host program sending these specific commands can communicate with this program. The Application Note AN2606 explains the bootloader configuration.

Possible peripherals for the bootloader: USART, I2C, SPI, CAN, USB.

2.5 Bootloader connection through USART

The internal bootloader in the uses only USART1/3 (no way to use USART2). We will use USART3, and to connect our PC to USART3, we obviously need a USART/USB bridge:

- USART3_RX (PC11 - CN7 pin 2)
- USART3_TX (PC10 - CN7 pin1)

On the PC side, we use **STM32Cube Programmer**.

Let's try to connect to our bootloader with USART3 and load a new application.

3 Writing a custom bootloader

We want to write our own bootloader for a STM32F446. It will be a very simple one with very few functionalities. The main (and only) purpose is to be able to update our user application. Of Our bootloader won't be compatible with STM32CubeProgrammer, so we will also have to write our PC application.

3.1 Hardware configuration

We will use USART2 as the serial link connected to our bootloader. It would be great to have a separated debug serial link (see Figure 1) but to keep things really simple, we will use USART2 for both the bootloader and Debug.

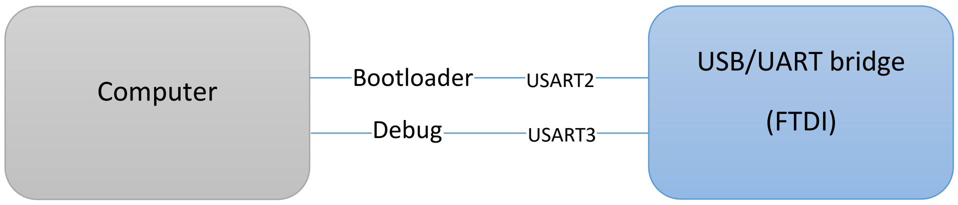


Figure 1: Hardware configuration of our project

3.2 Bootloader and user application

They will both be stored in the flash memory. But to get things tidy; we place them in different sectors:

- Sector 0 and 1 (32 Kb): our bootloader firmware
- Sector 2 to 7: our application firmware

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

Let's try to create 2 projects. One called *bootloader0* (slow led blinking) and the other one called *userApplication0* (fast led blinking) and program them in their sectors.

Problem: The user application is never launched (except after programming). Indeed, the MCU start at 0x0800 0000 so only the bootloader is running.

3.3 Bootloader user interface

When the MCU starts, the the bootloader runs. Then its behaviour depends on the user choice:

1. It can start a firmware update (we will see that later).
2. If no action is taken by the user, the bootloader has nothing to do. So it needs to jump to the user application address (reset handler of the user application).

We keep the same userApplication0 (fast led blinking). Let's build a new bootloader 1 that check the user button after reset. If the button is pressed, then we run the bootloader code (later there will be the firmware update here). If not, we jump to the user application 0.

To call the user application reset handler, we are going to use function pointer. In the bootloader project:

```
void (*userApplication) (void);
userApplication = ADDRESS OF THE USER APP;
```

```
userApplication();
```

3.4 Interrupt function in the user Application

We now want to add another functionality to our user application (User Application1). The string "Push Button pressed in User application 1" shall be sent on USART2 whenever the push button PC13 is pressed. We want to use interrupt for this functionality.

First, we try the User Application 1 at the address 0x08000 0000 of the MCU. We can check that everything works fine.

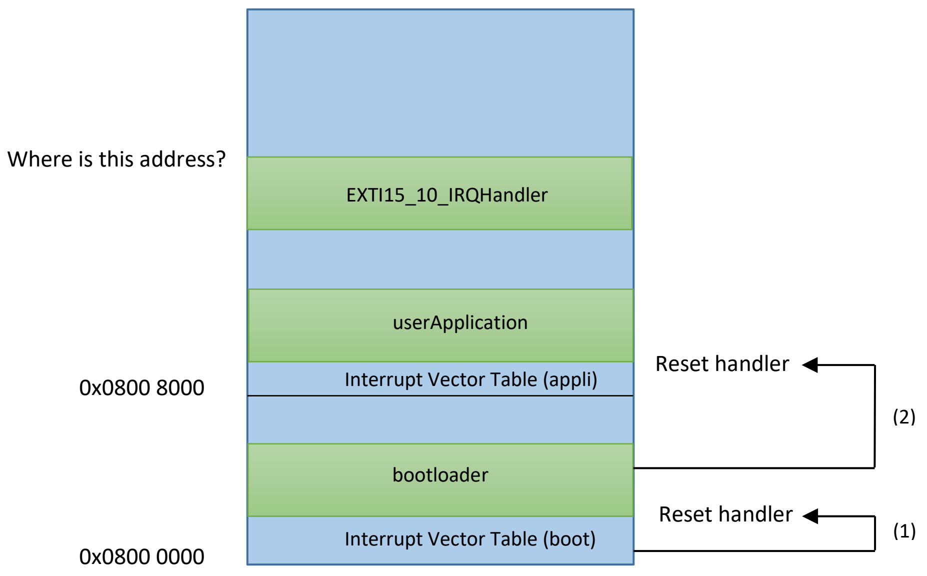
Info: Here is the process to set up an external GPIO interrupt in a STM32 MCU:

1. The GPIO PIN must be enabled as input.
2. The PORT number must be redirect to the corresponding EXTI line (SYSCFG register). /!\ We must enable the clock for SYSCFG before using it.
3. The EXTI line interrupt must be unmask (EXTI->IMR register).
4. The falling/rising edge that triggered the interrupt must be selected. (EXTI->FTSR register)
5. NVIC EXTI must be enabled. A priority can also be set (the lower the priority number, the higher the priority).

When this application works fine, we try to replace both bootloader 1 and User Application 1 in the MCU:

- **Bootloader 1** at 0x0800 0000 (sector 0)
- **User Application1** at 0x0800 8000 (sector 2)

We realize that the User Application 1 is launched properly but the PB interrupt is not working anymore. Why is that?



(1) : The bootloader jumper to its reset handler (from its Interrupt Vector Table). The PC is updated with bootloader reset handler address and the bootloader starts.

(2) : If the PB is not pushed, the bootloader jumps to the application reset handler and the application starts.

(3) When an IT is triggered, where the MCU gets the address of the interrupt routine? **Answer**: In the bootloader vector table. But in the bootloader vector table, have you set the address of this corresponding EXTI interrupt? **Answer**: No, because we did it in the interrupt vector table of the application. So, when the IT fires, the MCU goes to an unknown address and get lost.

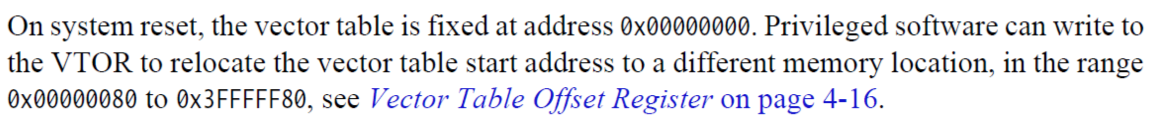
➡ We are going to update our bootloader (bootloader 2) to take care of this.

<u>What do we have to?</u>

When the MCU launches the application, we need to tell the application to use a new vector table.

<u>How do we do that?</u>

We need to refer to the generic Cortex M4 documentation (below)



On system reset, the vector table is fixed at address `0x00000000`. Privileged software can write to the VTOR to relocate the vector table start address to a different memory location, in the range `0x00000080` to `0x3FFFFF80`, see *Vector Table Offset Register* on page 4-16.

<u>When do we do it?</u>

It's better to do it in the bootloader just before jumping to the user application.

Let's try this. New firmware bootloader 2 at 0x0800 0000 and the User Application 1 at 0x0800 8000.

3.5 Main stack pointer

Relocating the vector table is not enough. We must do in software the same hardware reset sequence seen in the paragraph 2.3:

1. Set the Main Stack Pointer (To Do)
2. Set the PC with the application reset Handler (done when we call the application from the bootloader).

We are going to update our bootloader (we keep bootloader 2) to take care of this.

<u>What happen if we don't change the MSP?</u>

May be nothing if the application and bootloader use the same start address for the stack. But that is not necessarily the case. Of course, if the stack doesn't work properly, the application is corrupted. In the bootloader, Main stack pointer for the User Application should be set just before jumping to the user application.

<u>How do we know the main stack pointer of the user application?</u>

Again, the MSP is at the first address of the user application. In our case 0x0800 8000. To update the MSP, we can use the CMSIS macro:

```
void __set_MSP(uint32_t topOfMainStack)
```

3.6 User Application firmware update.

The last mission of the bootloader is to get the new code from the PC and to flash it in sector 2.

Which data do we send from the PC?

We need to send to the MCU the exact content of the overall flash memory. This is exactly what .hex and .bin files are.

Let's check the content of the MCU memory and compare it to the userApplication.bin and userApplication.hex.

We need to take two actions:

1. Build a PC application to send the content of the memory (.hex or .bin) to the MCU USART2.
2. Build a function in our bootloader that takes the content received on the USART2 and writes its own flash memory to update the application.

3.7 PC application

We can build a C application or a python script. It shall send the exact content of the .bin file on the COM port. Python is a good choice for its simplicity. Indeed, we don't require any performances for this application. Here is an example of a simple python script (IDLE IDE):

```
import serial
import os

print('Bootloader')
fileName = "STM32F446_UserApplication_Update.bin"
fileSize = os.path.getsize(fileName)
print(f'Binary file is {fileSize} bytes')


s = serial.Serial("COM5",115200,timeout=1)
f = open(fileName,"rb")

for x in range(5156):
    s.write(f.read(1))
    print(s.read(1))
```

3.8 Receiving the binary code: Bootloader 3

For this new bootloader firmware (bootloader 3), each received byte is loopbacked to the PC.

Let's check this new bootloader with the python script.

3.9 Writing to the Flash memory : Bootloader 4

- **The Flash memory has a small number of write cycle. We need to be very careful not to use any flash memory write process in an infinite loop. Otherwise, the MCU will have corrupted sectors and won't work properly anymore.**

Now, each received byte is also flashed to the memory. The serial link transmission is slow comparing to the MCU clock. We assume that we have time to write each byte in the flash memory before receiving a new one from the PC.

Each byte received from the serial communication shall be written in the Flash memory at 0x0800 8000. To make things easier, we will use the HAL libraries for the Flash process. However, if you feel comfortable, you can dive in the Reference Manual and configure the registers. It's not complicated but obviously a bit more time consuming at first.

The HAL libraries for the STM32F4xx MCU are available on the NAS. Before writing all the binary code in the Flash, please, try first with a small amount of data (10?, 100?) to check if the process works well.

- **Let's check this new bootloader 4.**

- **Be careful not to launch the bootloader before after updating. The HAL_FLASH_Erase is called and the Flash is erased.**

- **TODO for next year: Acknowledge the update before erasing the application sector 2.**