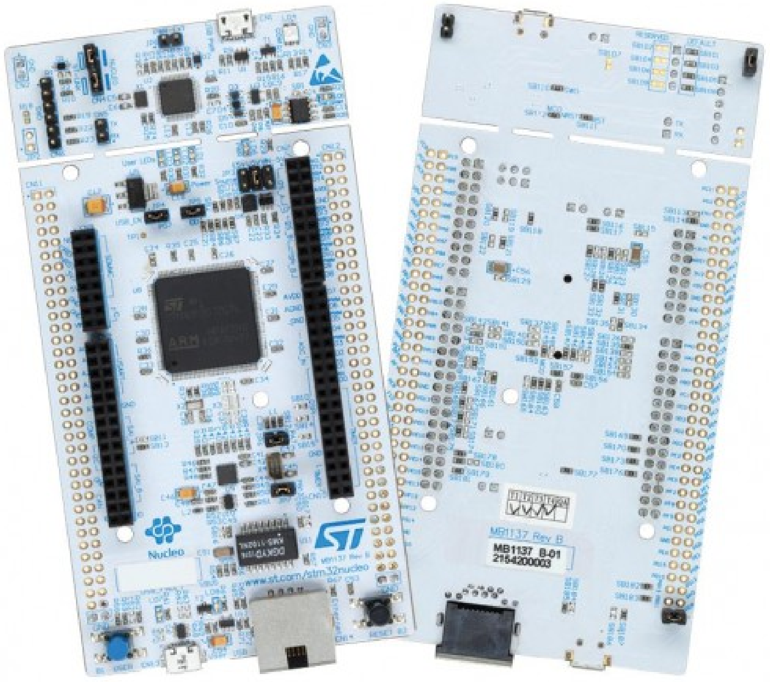
STM 32 F767ZI

GPIO, INTERRUPTS AND UART

ÁLVARO CASTRO LEITE
DEC 2021



Requirements

Micro-USB cable

STM32 development board

PC with all the tools installed

Serial Port terminal

Agenda

Skill level: Beginner

Introduction to HAL API

STMCubeMX walkthrough

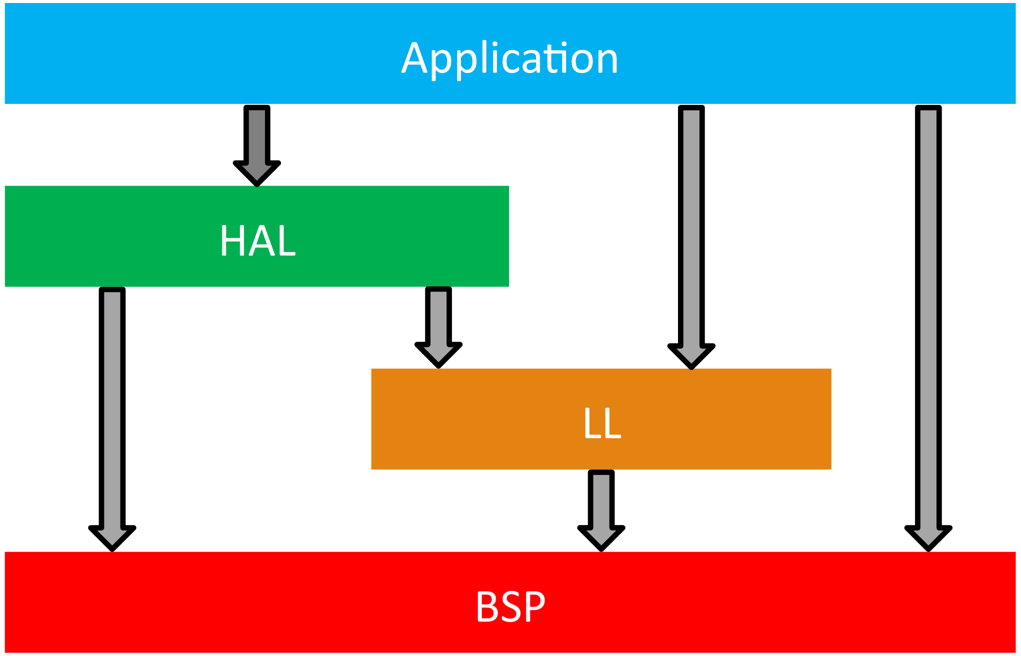
GPIO

Interrupt management

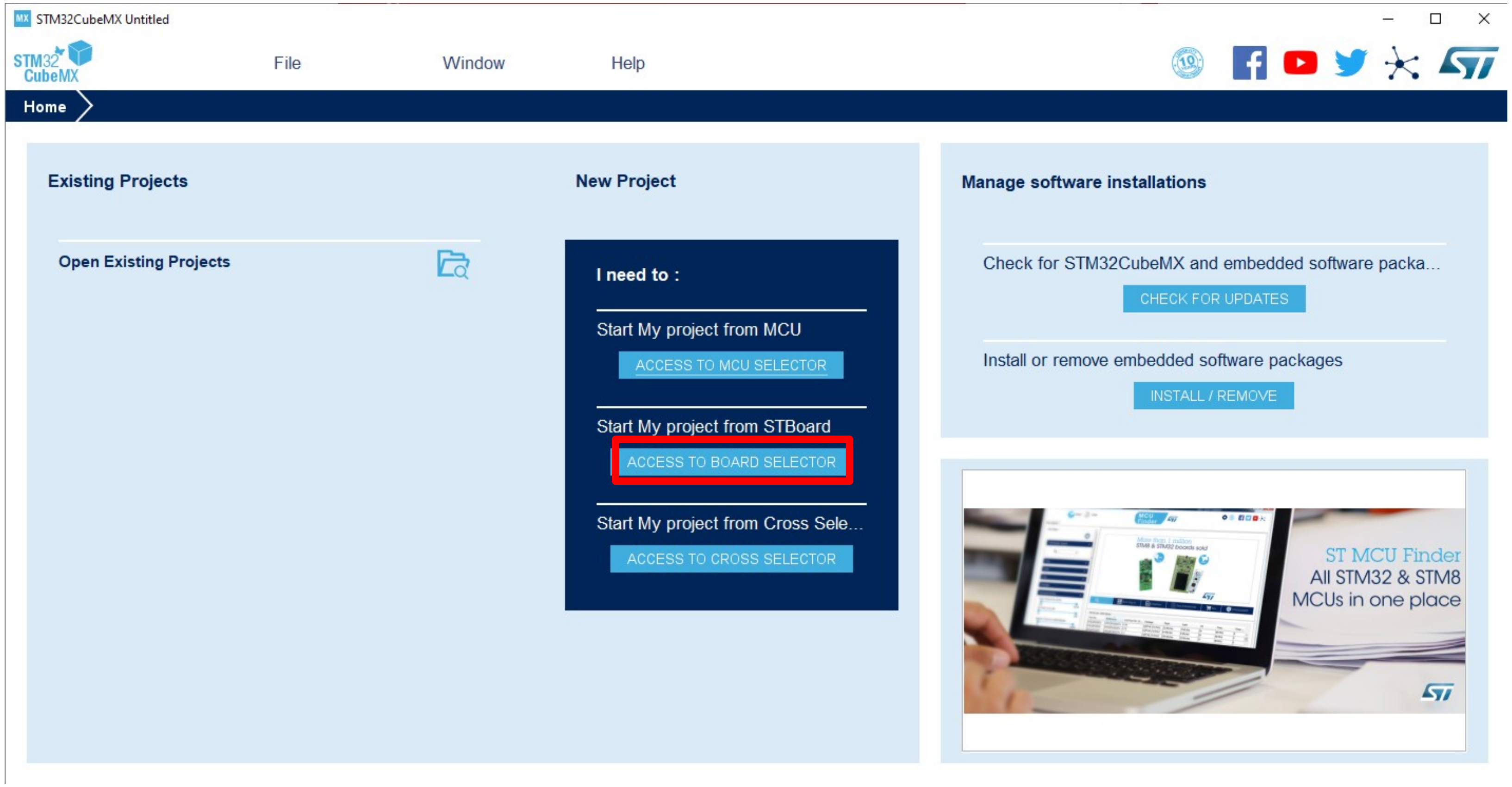
On-chip debug

UART

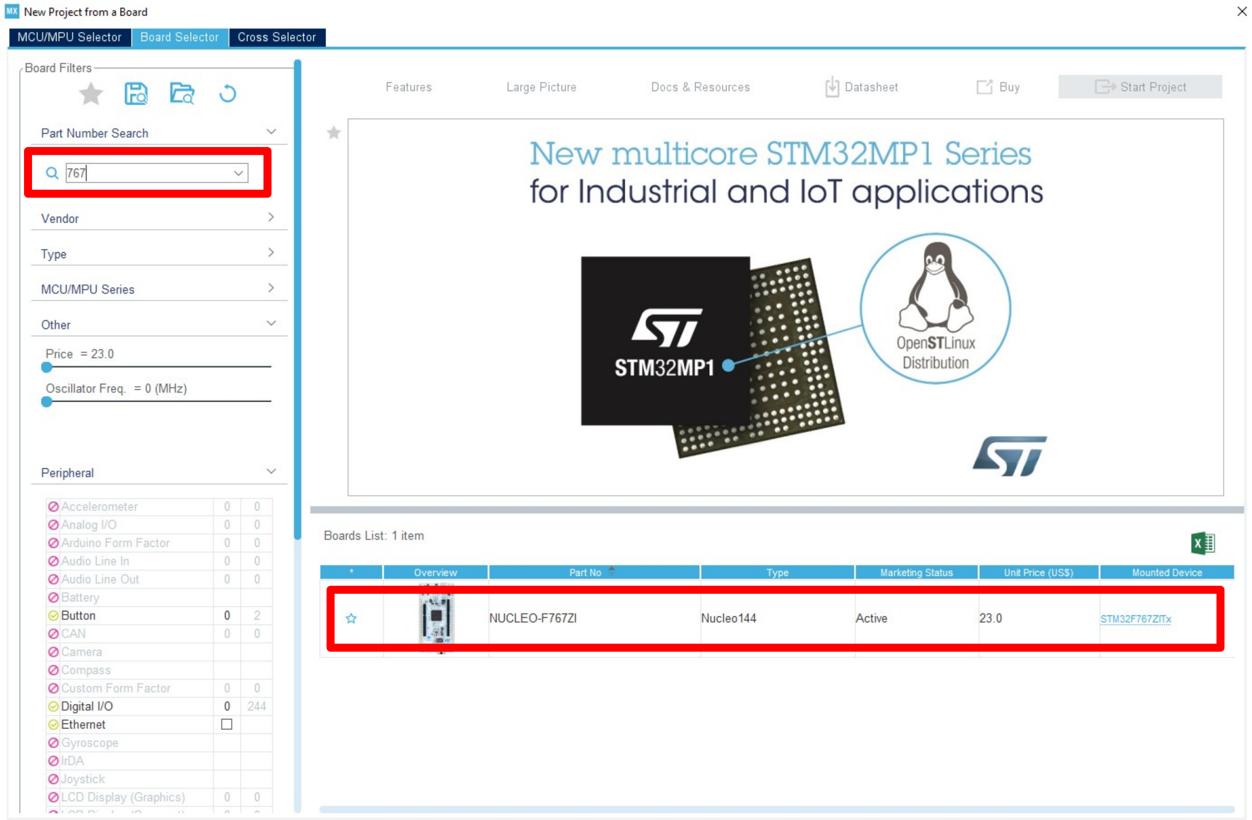
Introduction HAL API



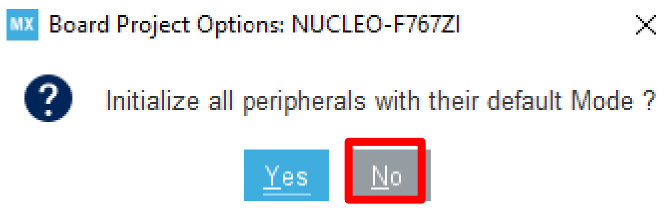
STMCubeMX walkthrough



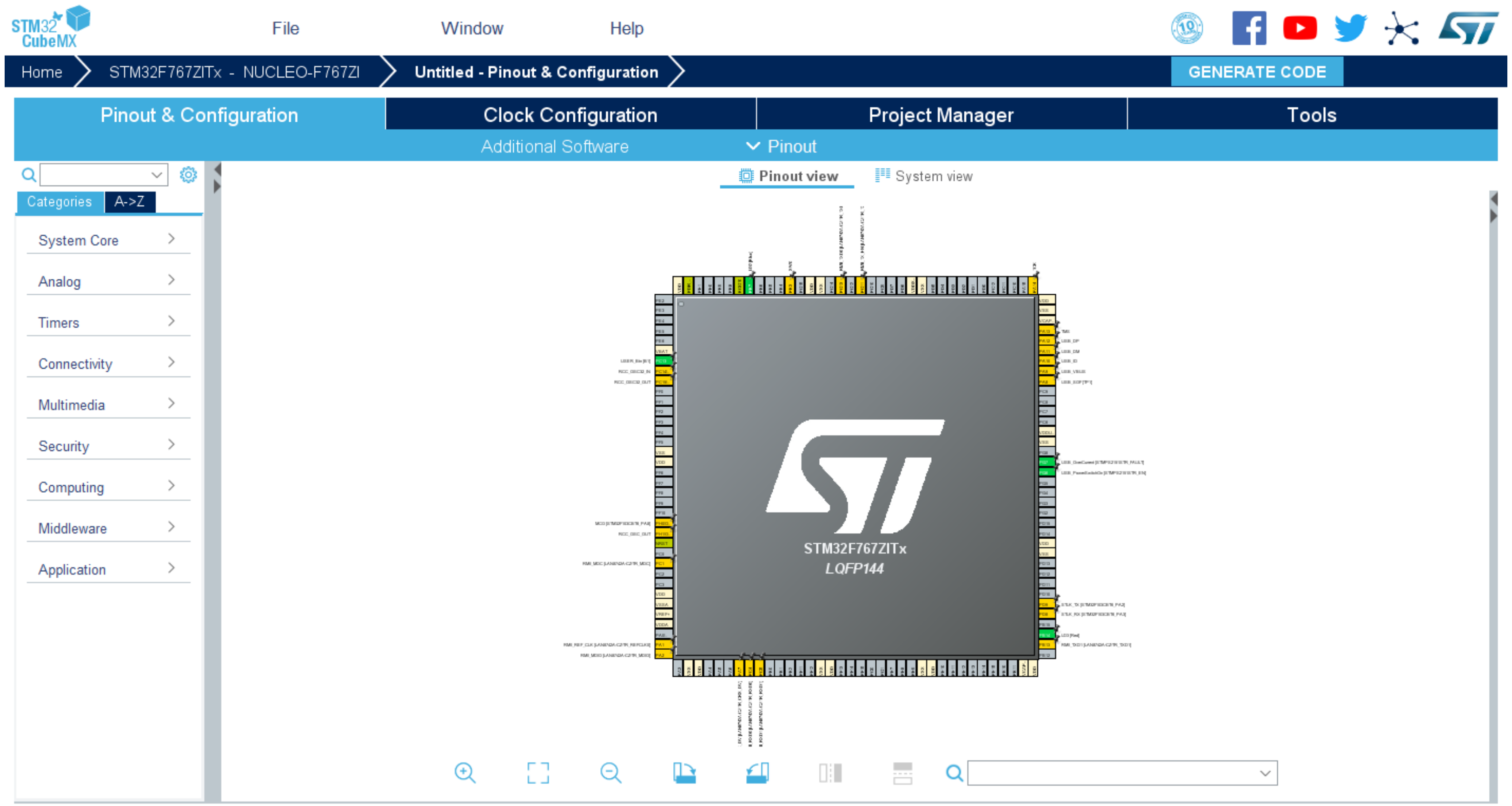
STMCubeMX walkthrough



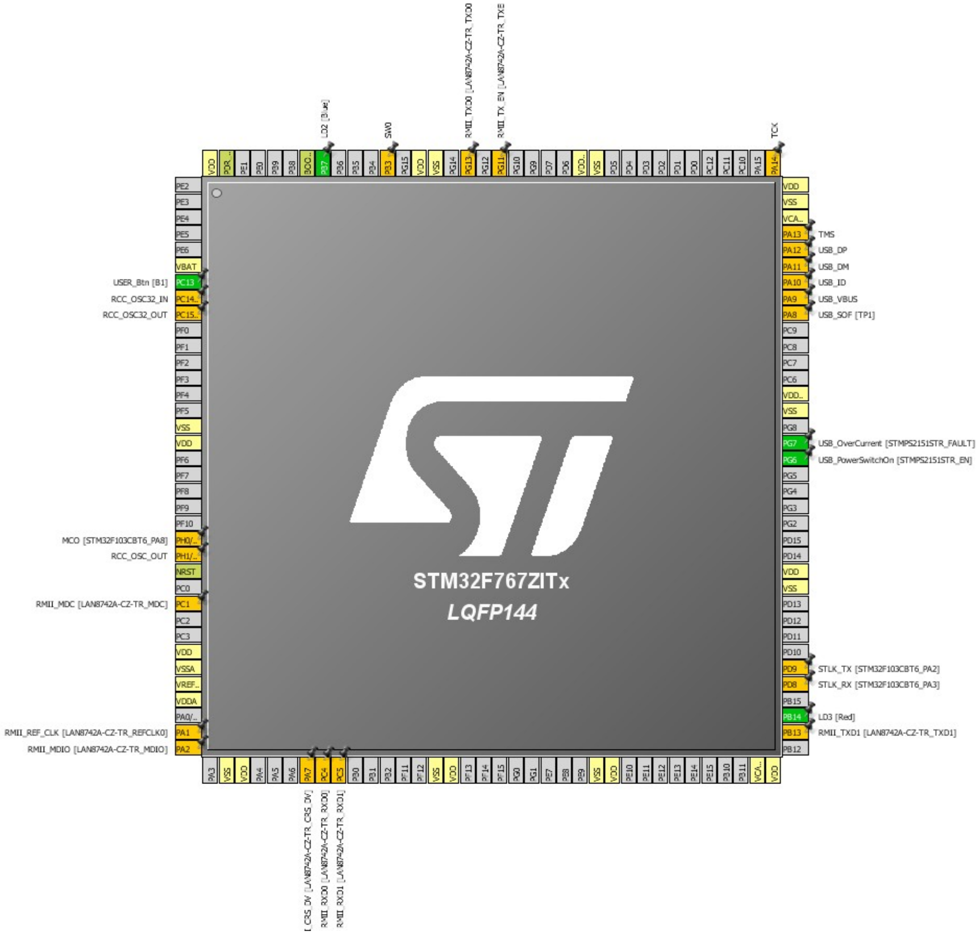
STMCubeMX walkthrough



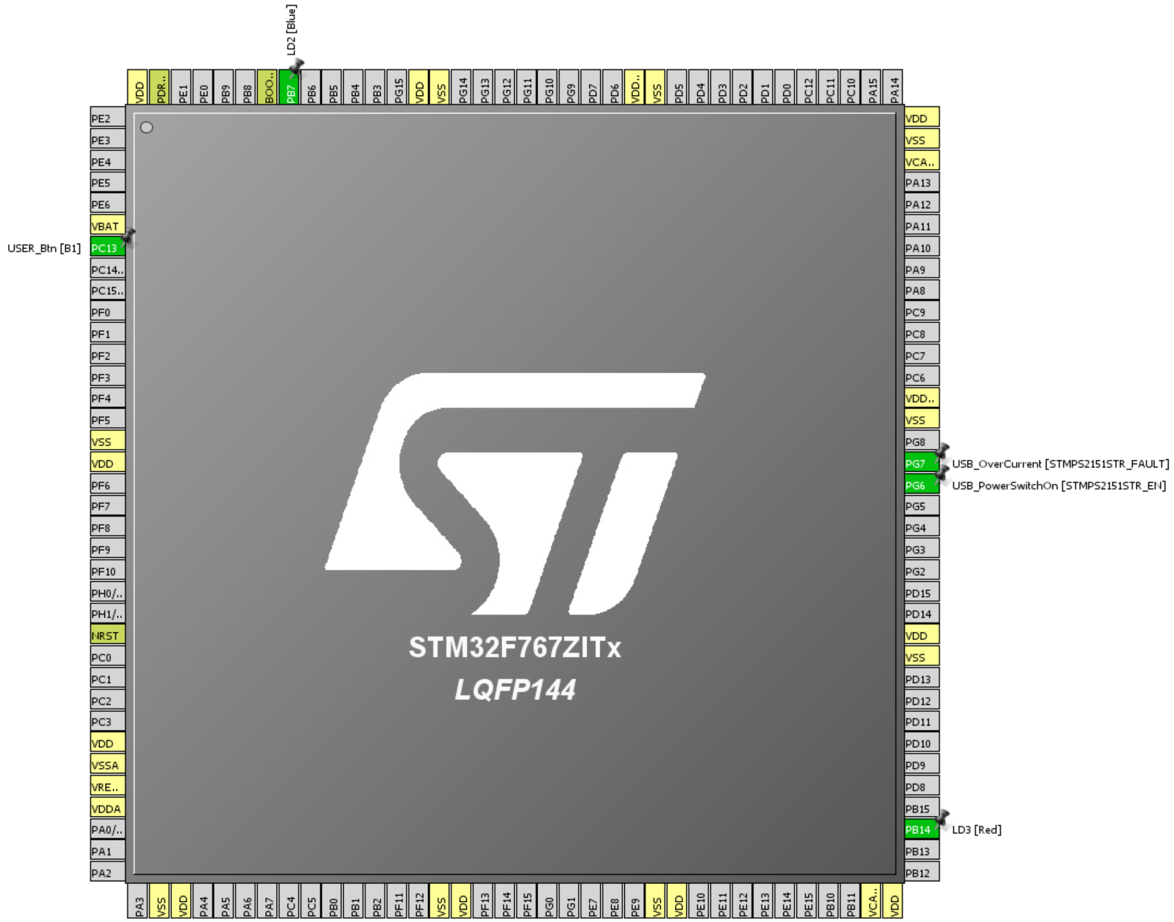
STMCubeMX walkthrough



STMCubeMX walkthrough



STMCubeMX walkthrough



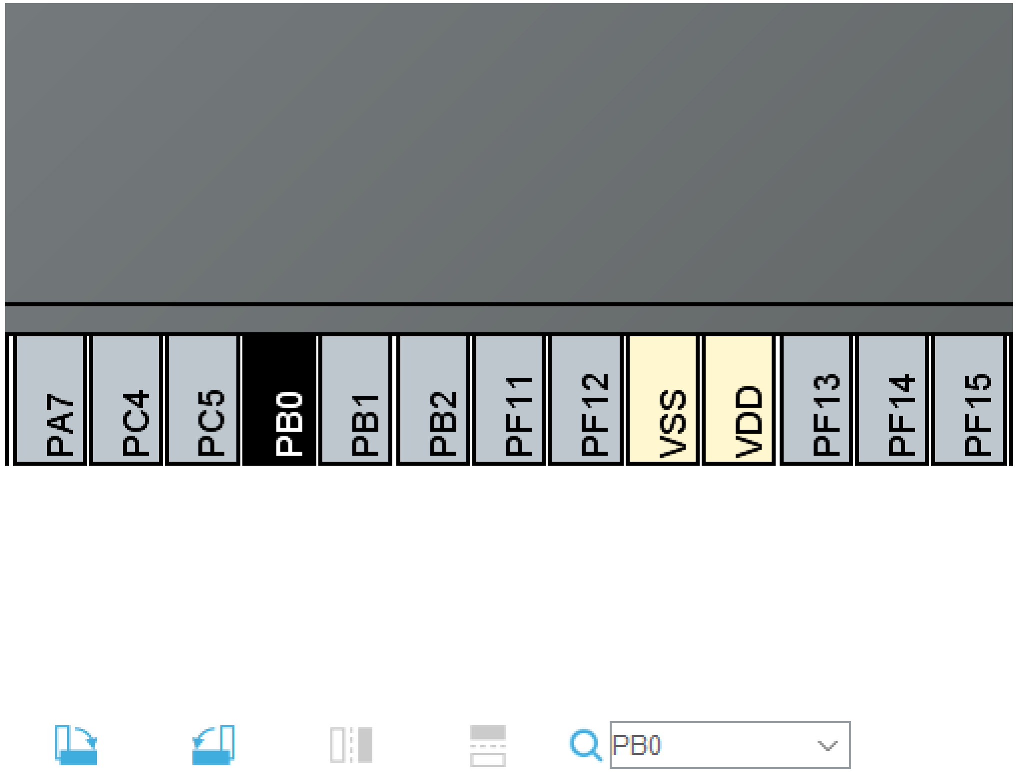
STMCubeMX walkthrough

Question: How many user LED's the board has? 3 right? Where's the green LED?

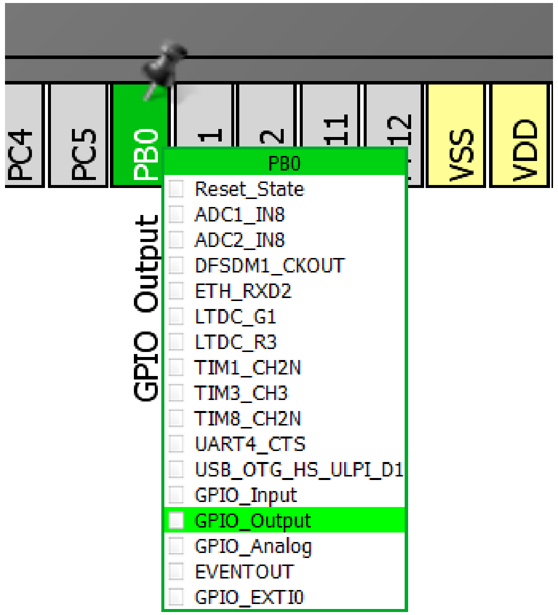
Tip: Open UM1974 and go to page 24.

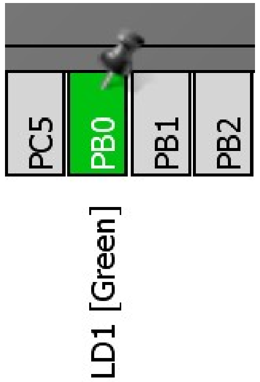
Answer: PB0

STMCubeMX walkthrough

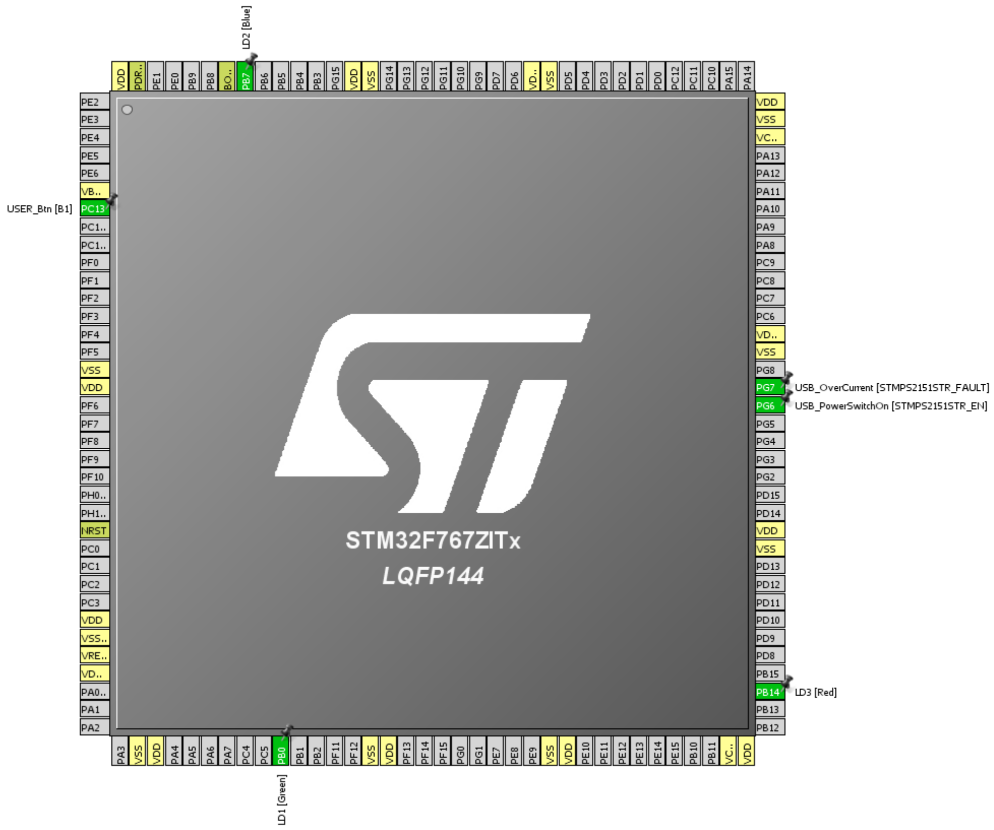


STMCubeMX walkthrough

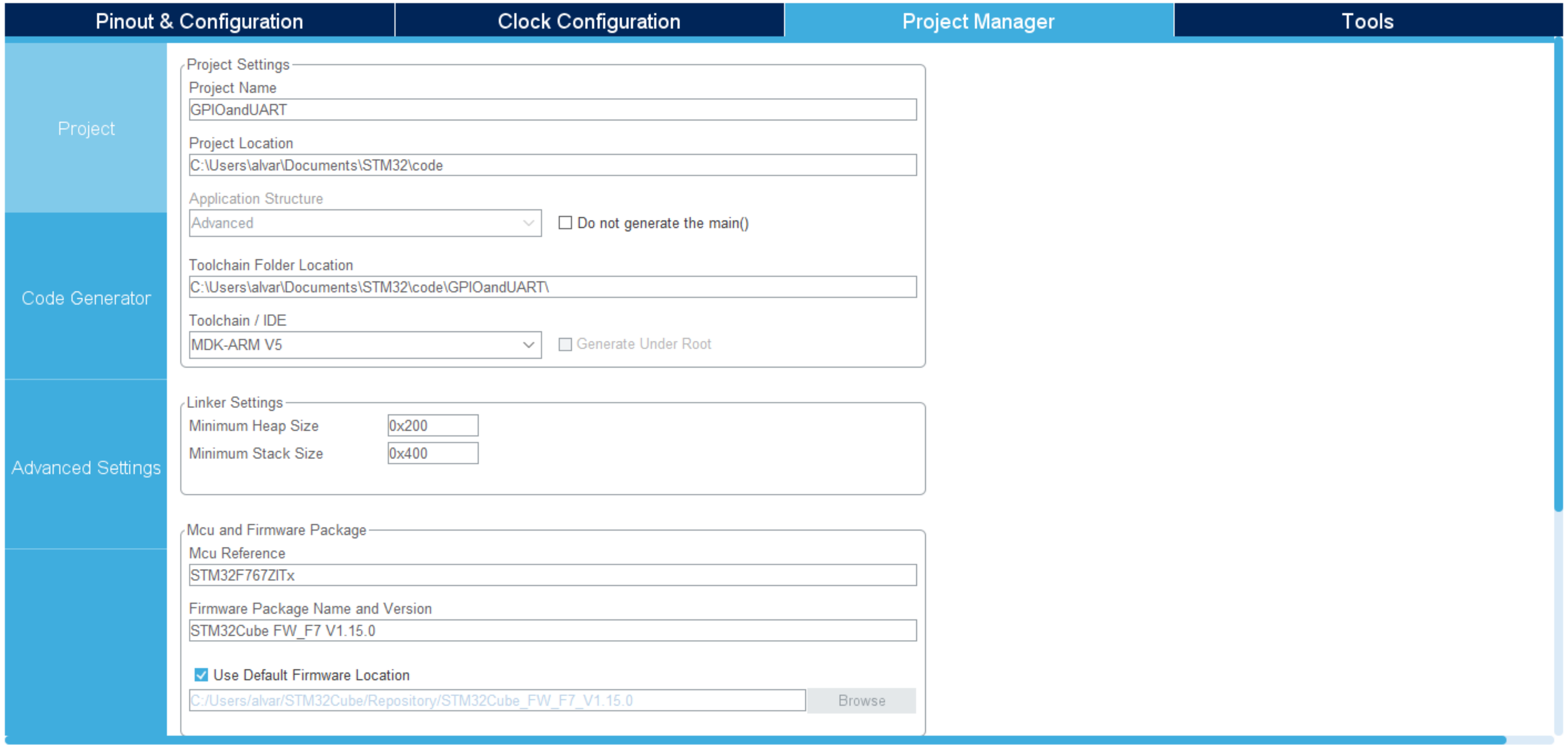




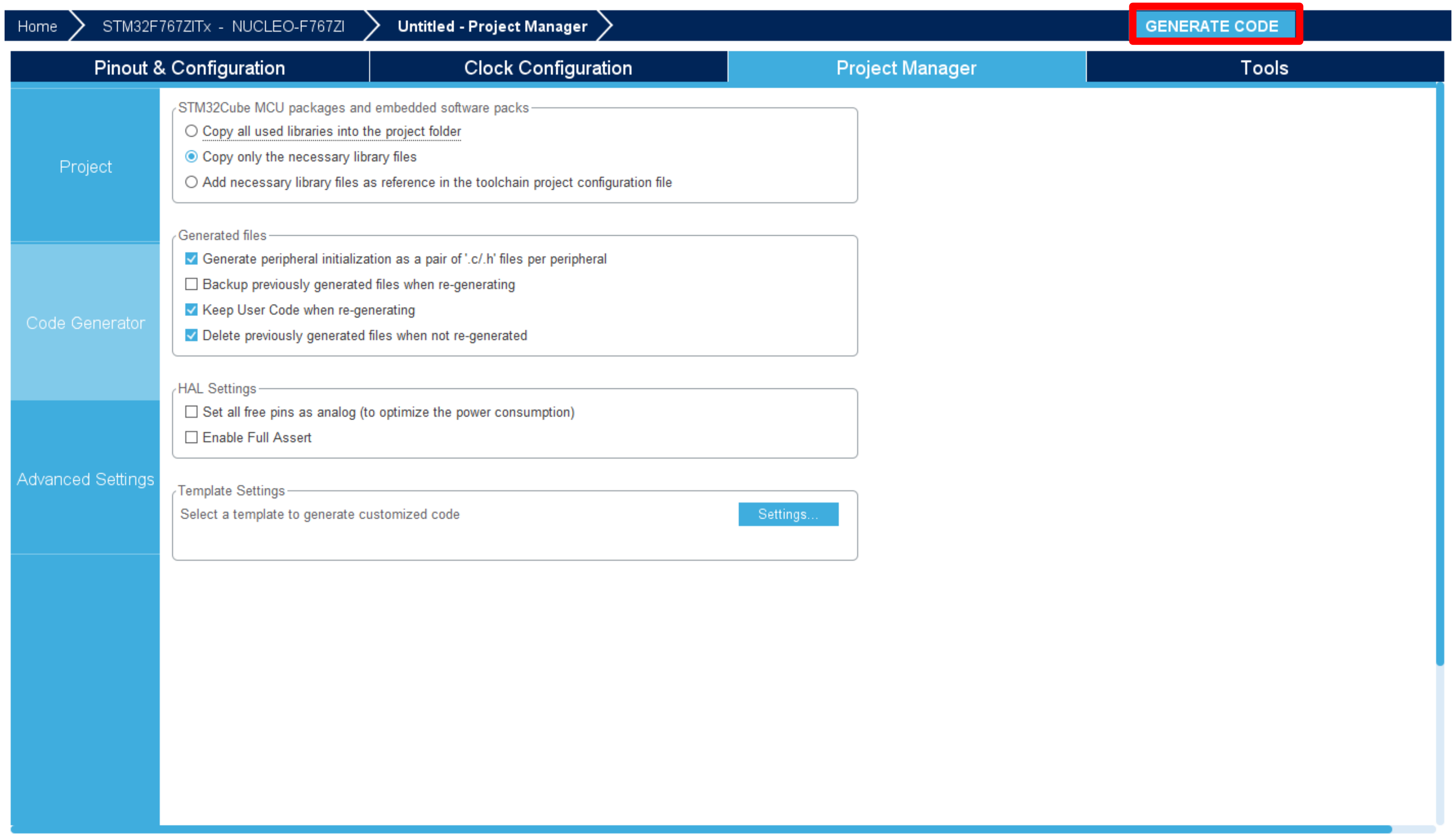
STMCubeMX walkthrough



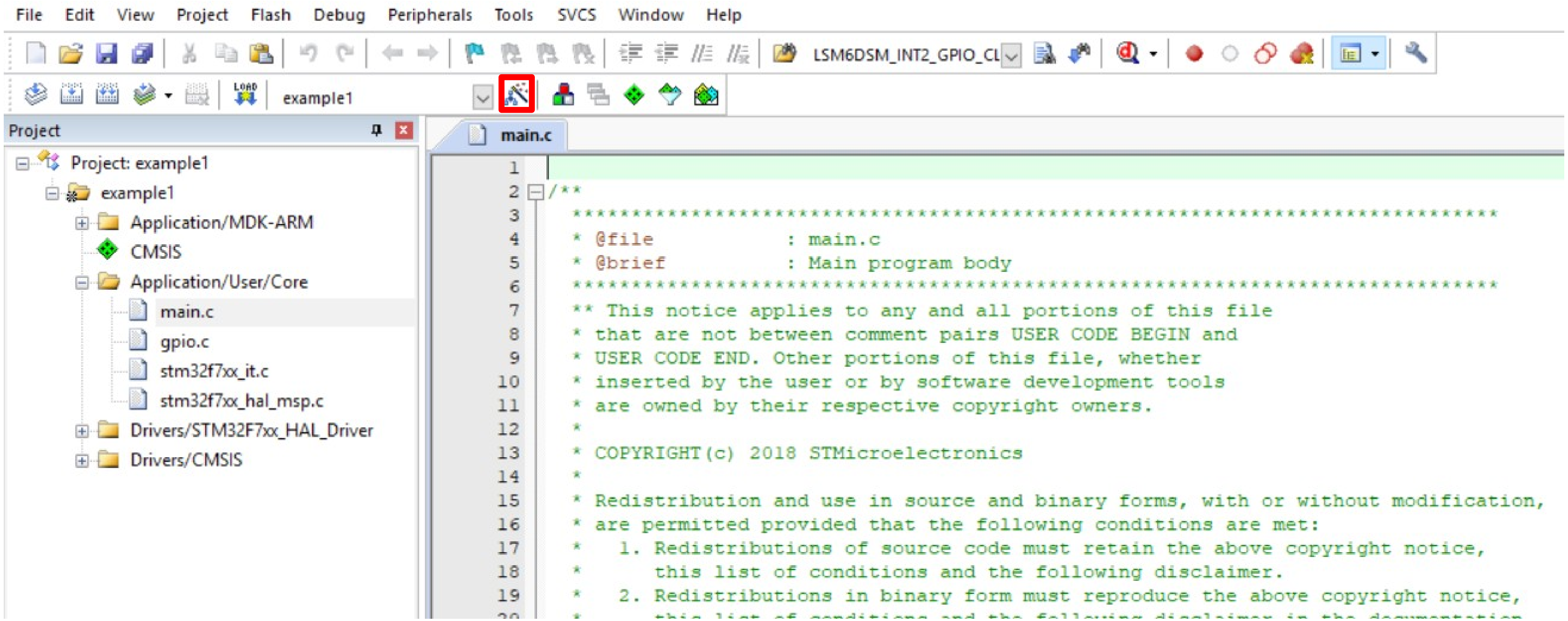
STMCubeMX walkthrough



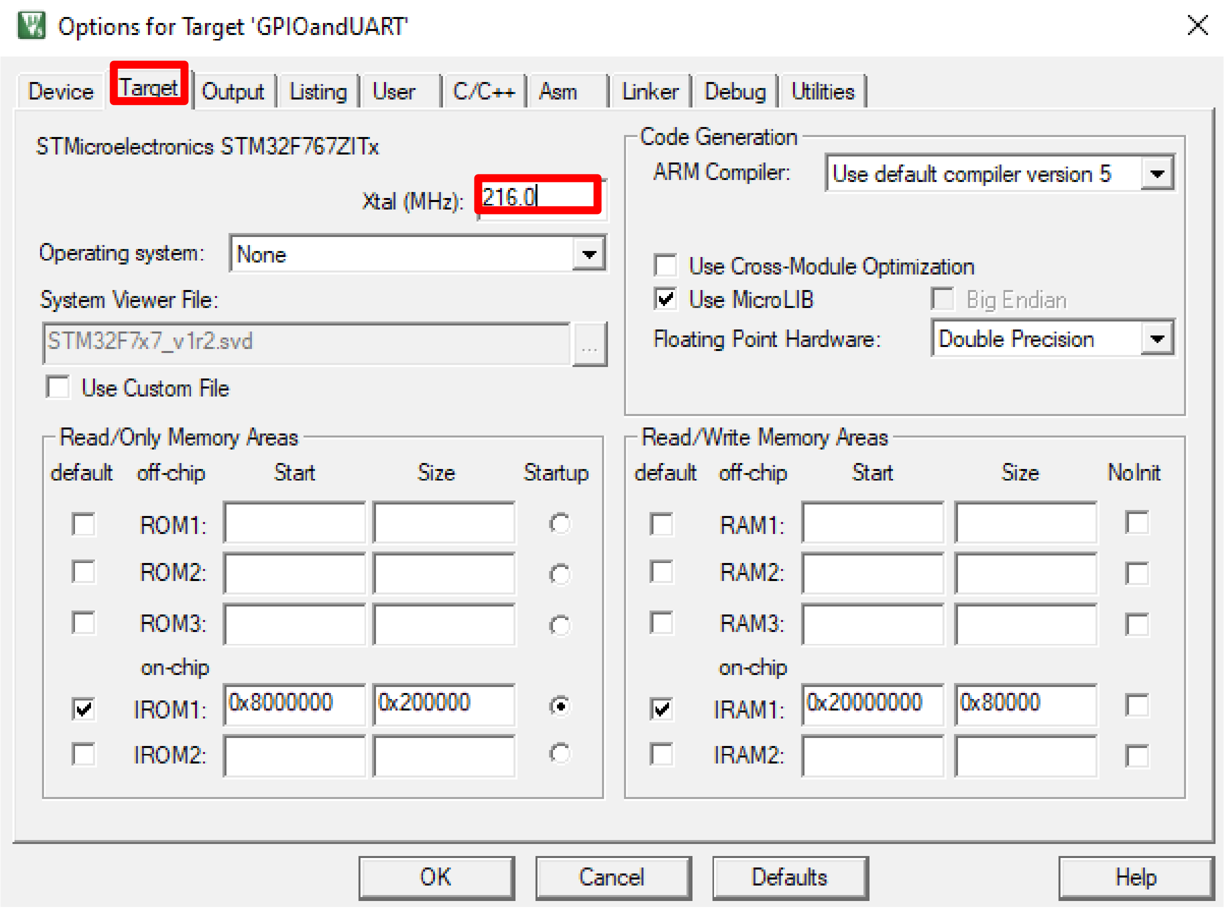
STMCubeMX walkthrough



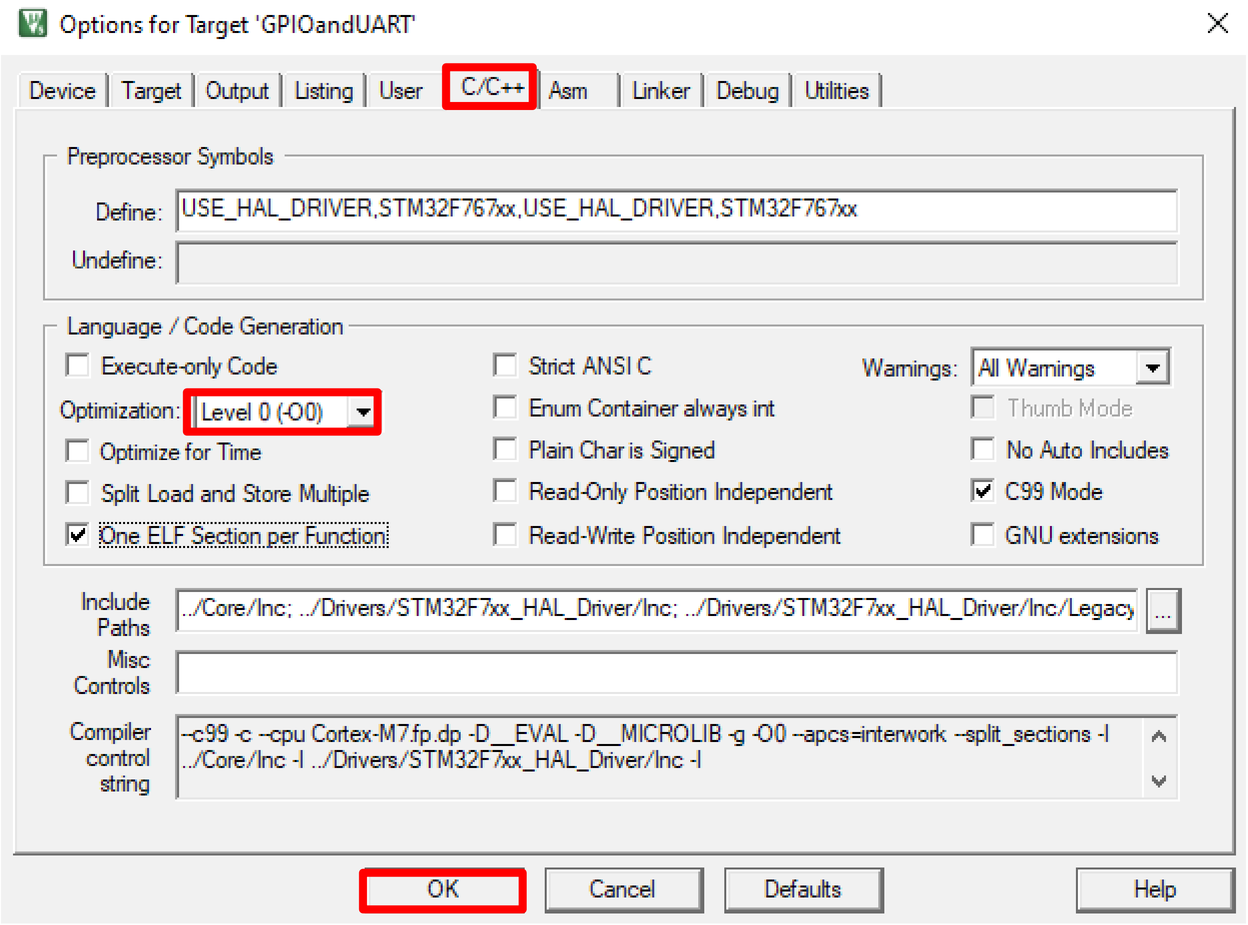
STMCubeMX walkthrough



STMCubeMX walkthrough



STMCubeMX walkthrough



GPIO

In UM1905 User Manual, on Chapter 26 (page 372), you can find the information about the GPIO HAL API.

It's important that you understand the structure associated with each I/O port and the API (set of functions to handle all GPIO).

HAL provides four manipulation routines to read, change and lock the state of an I/O.

GPIO

HAL provides four manipulation routines to read, change and lock the state of an I/O and two for configuration.

To read the status of an I/O we can use the function:

- `GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)`

which accepts the GPIO descriptor and the pin number. It returns GPIO_PIN_RESET when the I/O is low or GPIO_PIN_SET when high.

GPIO

Conversely, to change the I/O state, we have the function:

- `void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState)`

which accepts the GPIO descriptor, the pin number and the desired state.

GPIO

If we want to simply invert the I/O state, then we can use this convenient routine:

- `void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)`

NEEEICUM
núcleo de estudantes de engenharia eletrónica industrial e computadores da universidade do minho

GPIO

Example 1: Make a blinking led with the three LEDs reconfigured before.

```
while (1)
{
  HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_0);
  HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_7);
  HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_14);
  HAL_Delay(500);
  /* USER CODE END WHILE */
```

After writing the following lines, compile and flash, and the leds should blink.

GPIO

Example 2: Make the green LED to change the state when you push the user button.

```
while (1)
{
  if(HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13))
    HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_0);
  HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_7);
  HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_14);
  HAL_Delay(500);
  /* USER CODE END WHILE */
```

The problem: the system isn't analyzing the button all the time, most of it is doing other stuff (in the case: the delay).

The solution could be, use ***interrupts***.

Interrupt management

ARM architecture defines events as something that can be originated both by the hardware and the software itself. ARM architecture events are distinguished between two types: interrupts originated by the hardware, exceptions by the software (e.g., invalid access to the memory location).

NEEEICUM
núcleo de estudantes de engenharia eletrónica industrial e computadores da universidade do minho

Interrupt management

Cortex-M processors provide a unit dedicated to event management. This is called Nested Vectored Interrupt Controller (NVIC). NVIC is a dedicated hardware unit inside the Cortex-M based microcontrollers that is responsible for handling exceptions. You can find more information about NVIC in RM0410, chapter 10.

A dedicated programmable controller, named External Interrupt/Event Controller (EXTI), is responsible for the interconnection between the external I/O signals and the NVIC controller, as we will see next.

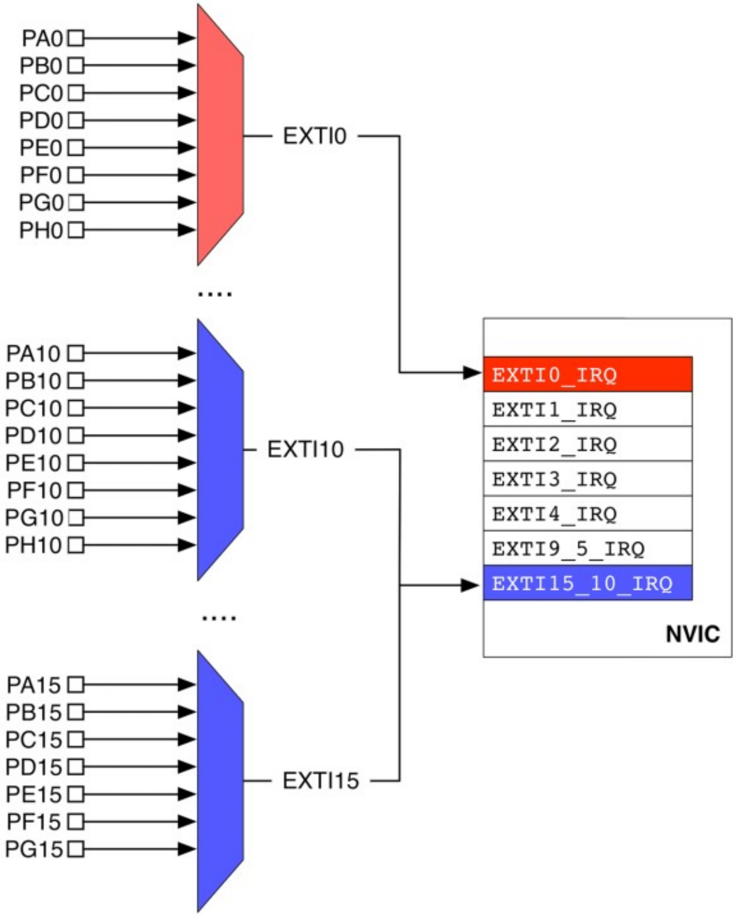
Interrupt management

STM32 microcontrollers provide a variable number of external interrupt sources connected to the NVIC through the EXTI controller, which in turn is capable of managing several EXTI lines. The number of interrupt sources and lines depends on the specific STM32 family.

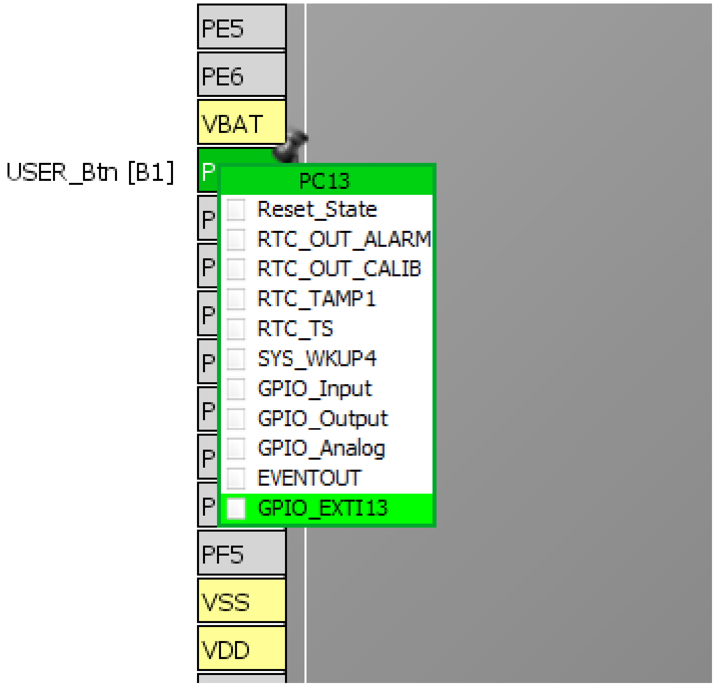
GPIO is connected to the EXTI lines, and it is possible to enable interrupts for every MCU GPIO, even if most of them share the same interrupt line. For example, for an STM32F7 MCU, up to 168 GPIOs are connected to 16 EXTI lines. However, there are only 11 independent interrupt in the EXTI.

You can find more information about EXTI in RM0410, chapter 11.

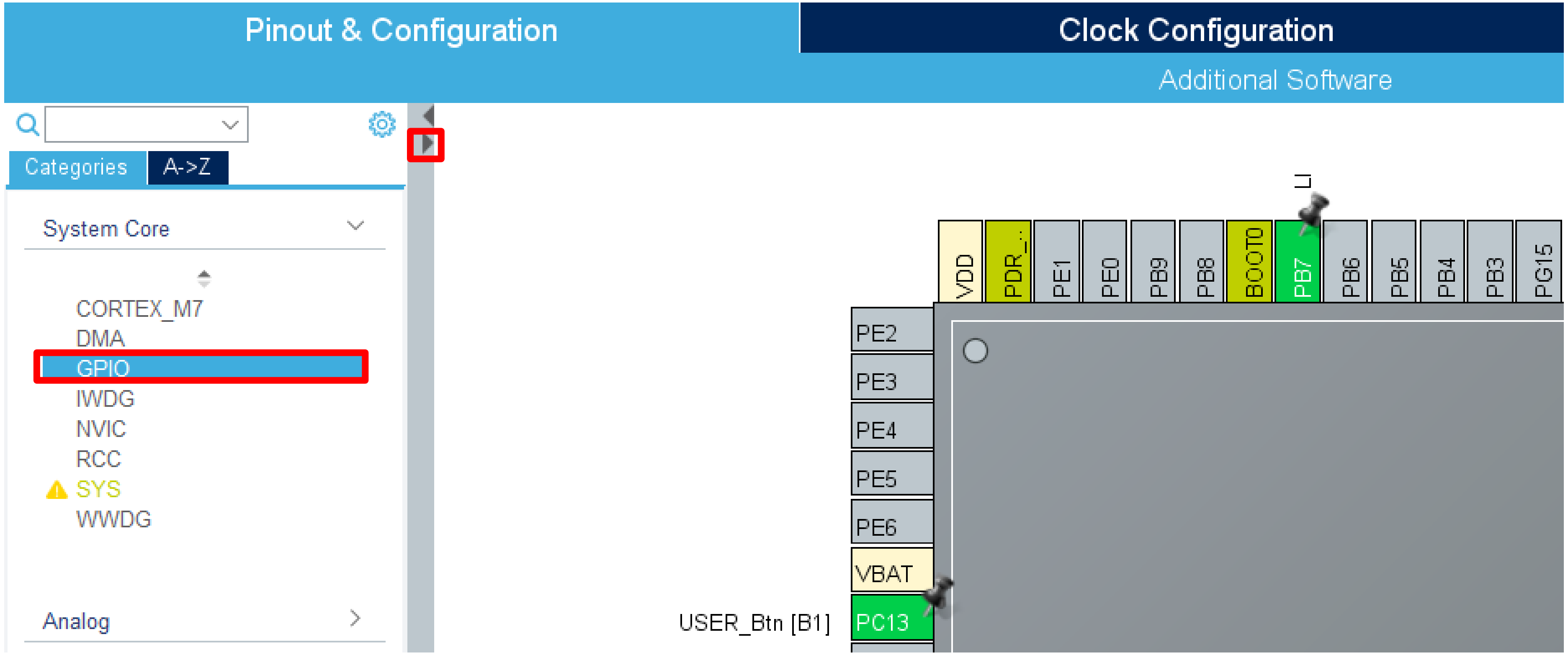
Interrupt management



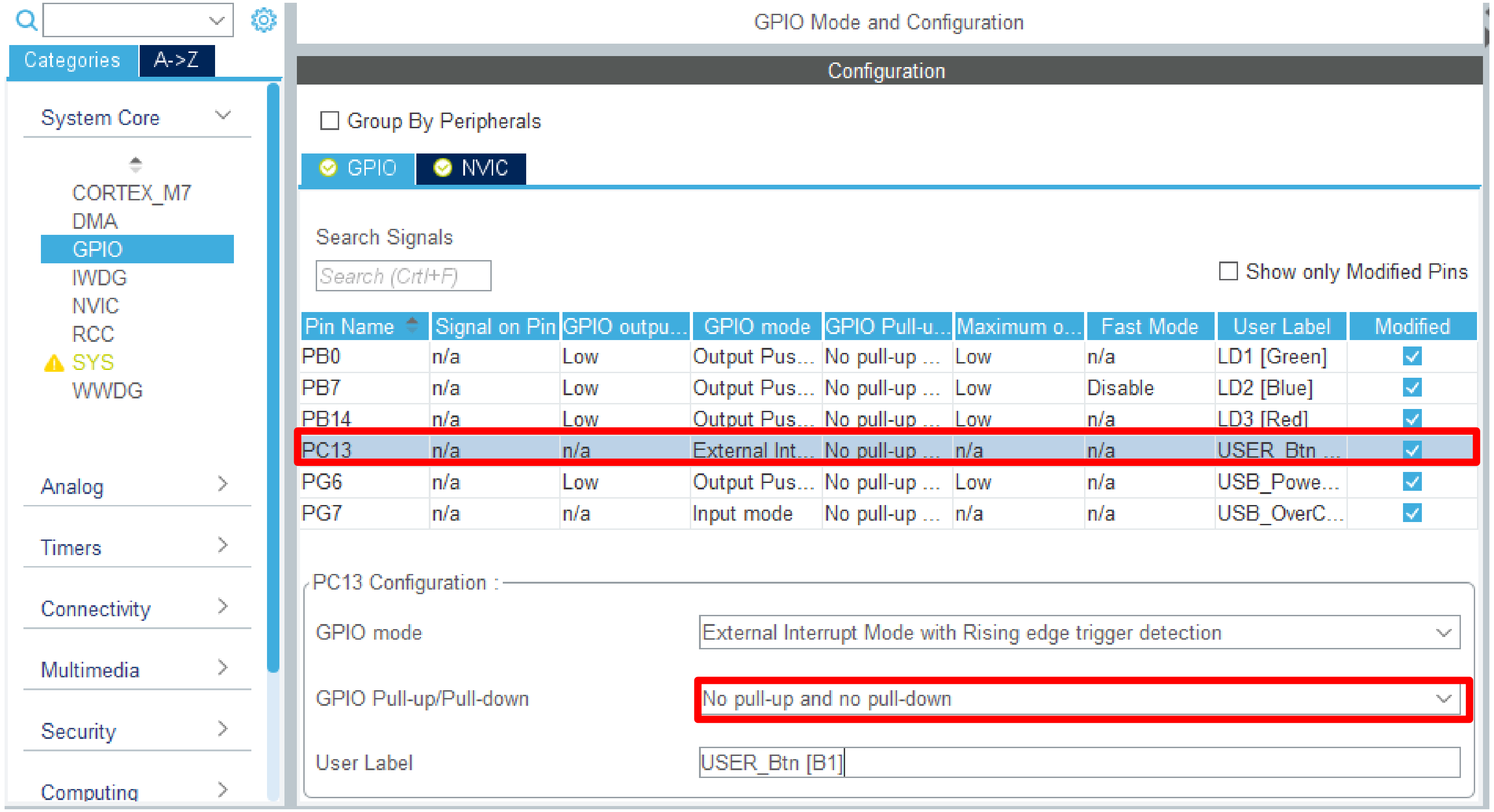
Interrupt management



Interrupt management



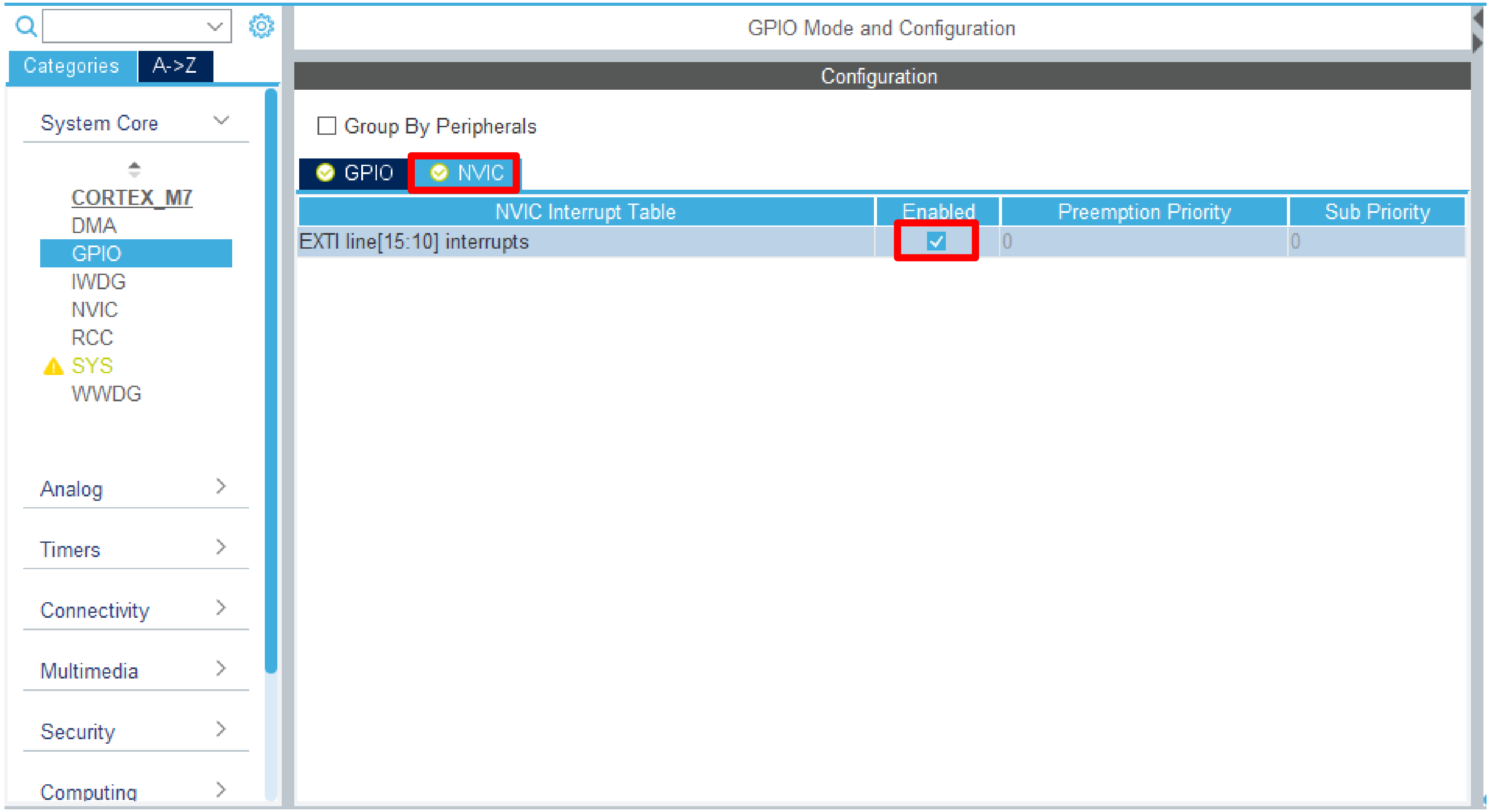
Interrupt management



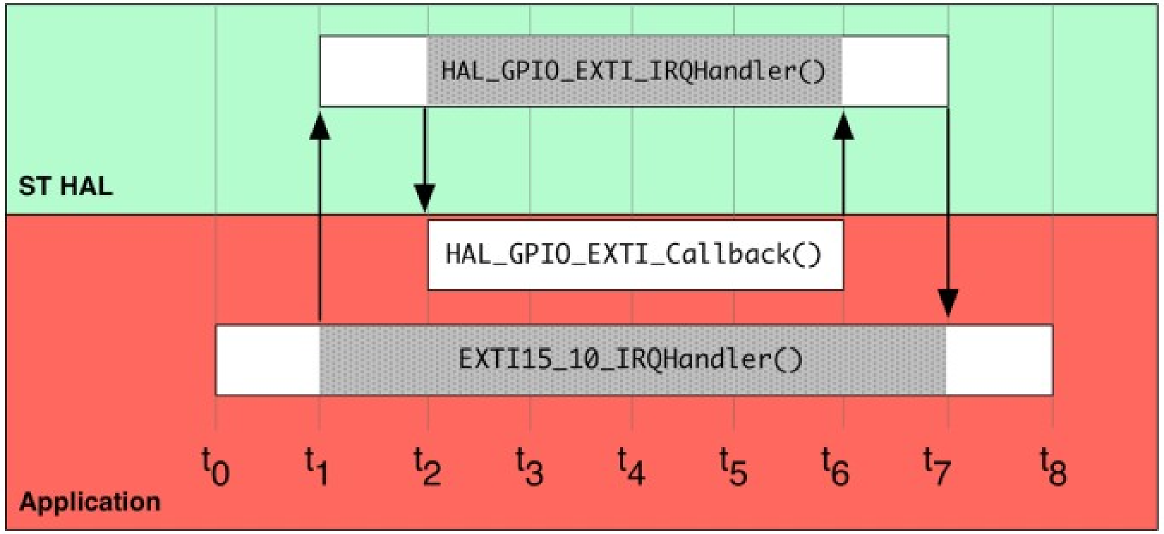
??

UM1974
page 75

Interrupt management



Interrupt management



Interrupt management

The STM32CubeMX already added implementation code of the EXTI15_10_IRQHandler function, this code is on stm32f7xx_it.c file.

```c
void EXTI15_10_IRQHandler(void)
{
  /* USER CODE BEGIN EXTI15_10_IRQn 0 */

  /* USER CODE END EXTI15_10_IRQn 0 */
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
  /* USER CODE BEGIN EXTI15_10_IRQn 1 */

  /* USER CODE END EXTI15_10_IRQn 1 */
}
```

Interrupt management

You must implement the HAL_GPIO_Callback function on GPIO file.

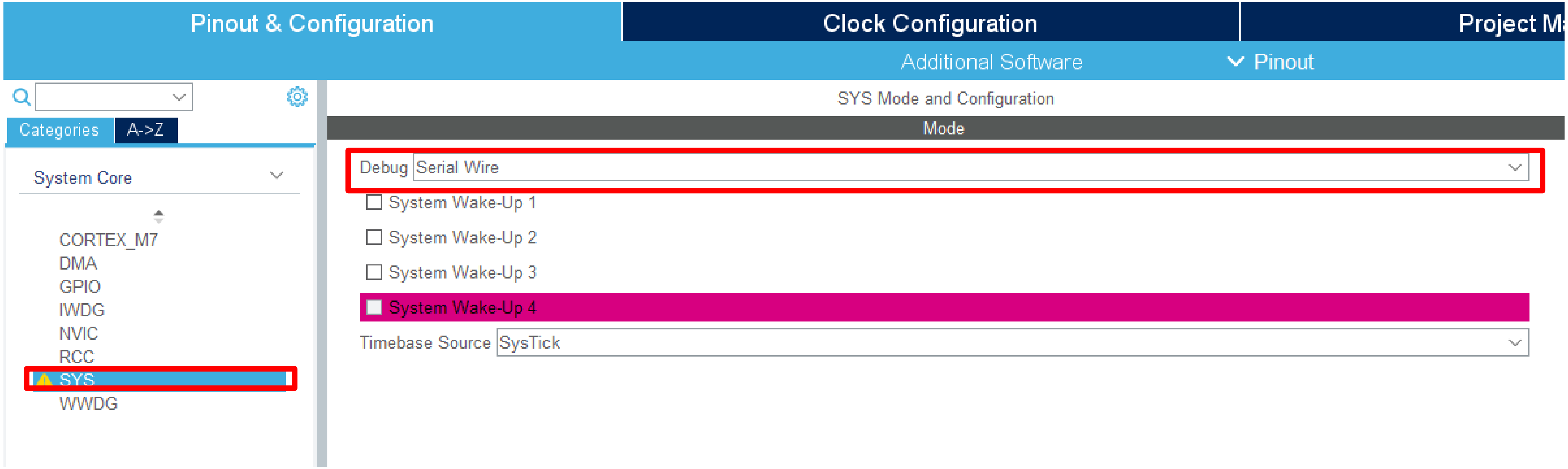
```
/* USER CODE BEGIN 2 */

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){
  if (GPIO_Pin == GPIO_PIN_13)
    HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_0);
}

/* USER CODE END 2 */
```

On-chip debug

In order to debug our code we need to prepare the peripherals for the communication.

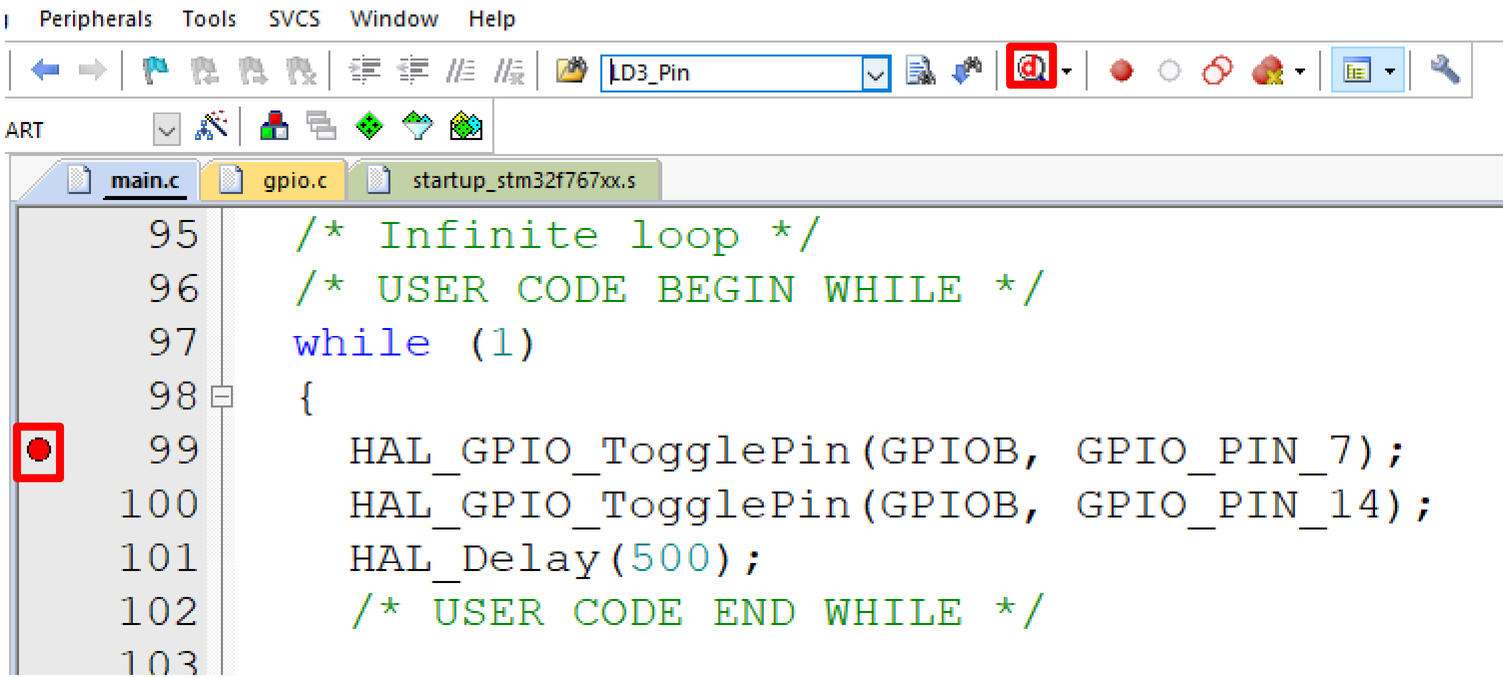
To do this it's necessary to enable the Single Wire Debug in the SYS(tem) peripheral.

On-chip debug

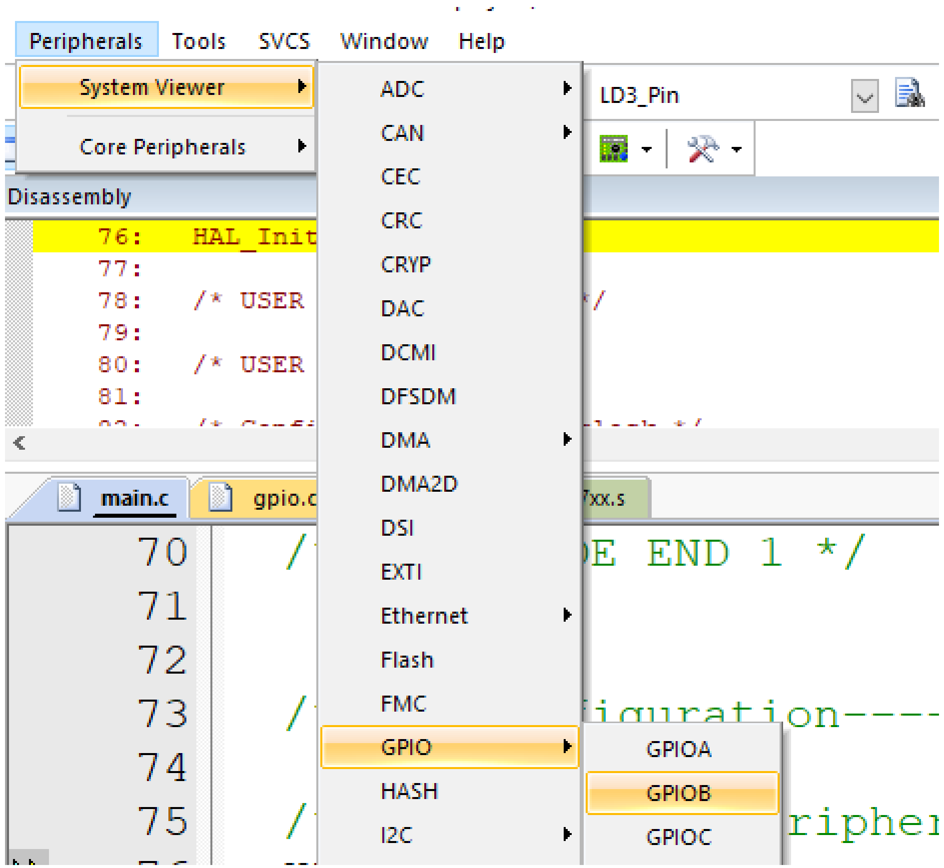


Generate the code again.

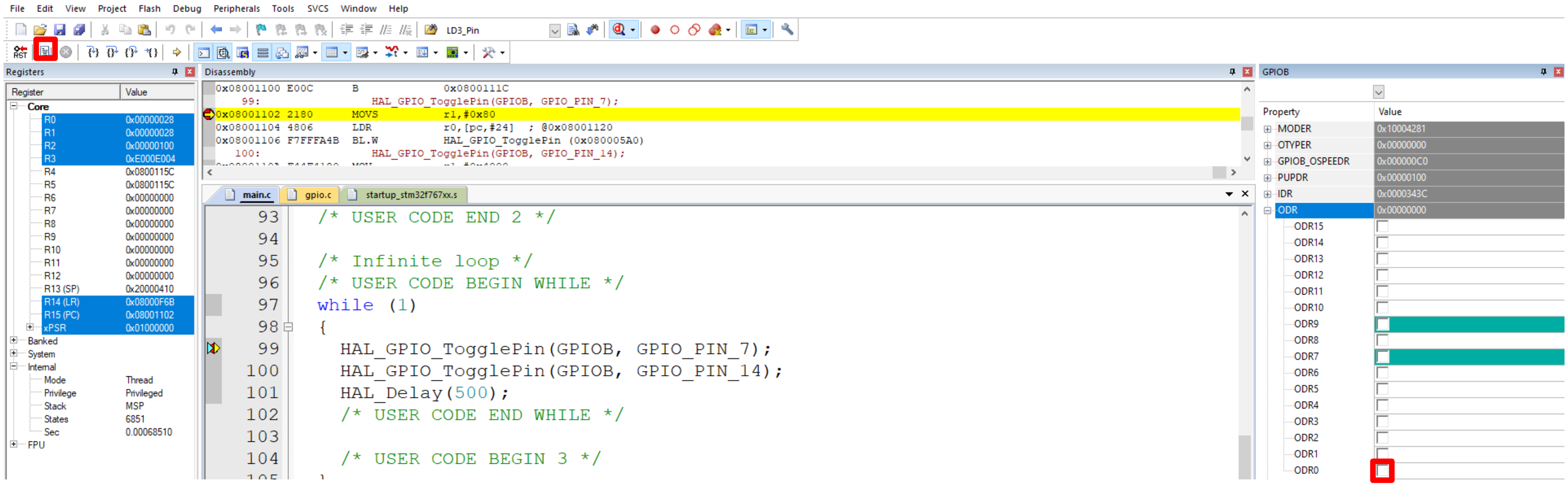
On-chip debug



On-chip debug



On-chip debug



UART

In order to use UART and USART peripherals, HAL API offers a set of functions.

To transmit a sequence of bytes over the USART in polling mode the HAL provides the function:

```
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

UART

To receive a sequence of bytes over the USART in polling mode the HAL provides the function:

```
HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef
*huart, uint8_t *pData, uint16_t Size, uint32_t
Timeout);
```

UART

To transmit a sequence of bytes in interrupt mode, the HAL defines the function:

```
HAL_StatusTypeDef
HAL_UART_Transmit_IT(UART_HandleTypeDef *huart,
uint8_t *pData, uint16_t Size);
```

UART

To receive a sequence of bytes over the USART in interrupt mode the HAL provides the function:

```
HAL_StatusTypeDef
HAL_UART_Receive_IT(UART_HandleTypeDef *huart, uint8_t
*pData, uint16_t Size);
```

UART

Example 3: Create a program that communicates with the computer through the RS232 protocol, echoing whatever the computer sent, using an UART (or USART) peripheral in the STM32 and a terminal program in the computer. The program may send a message on startup and should change the state of the red led every time a message is receved.

UART

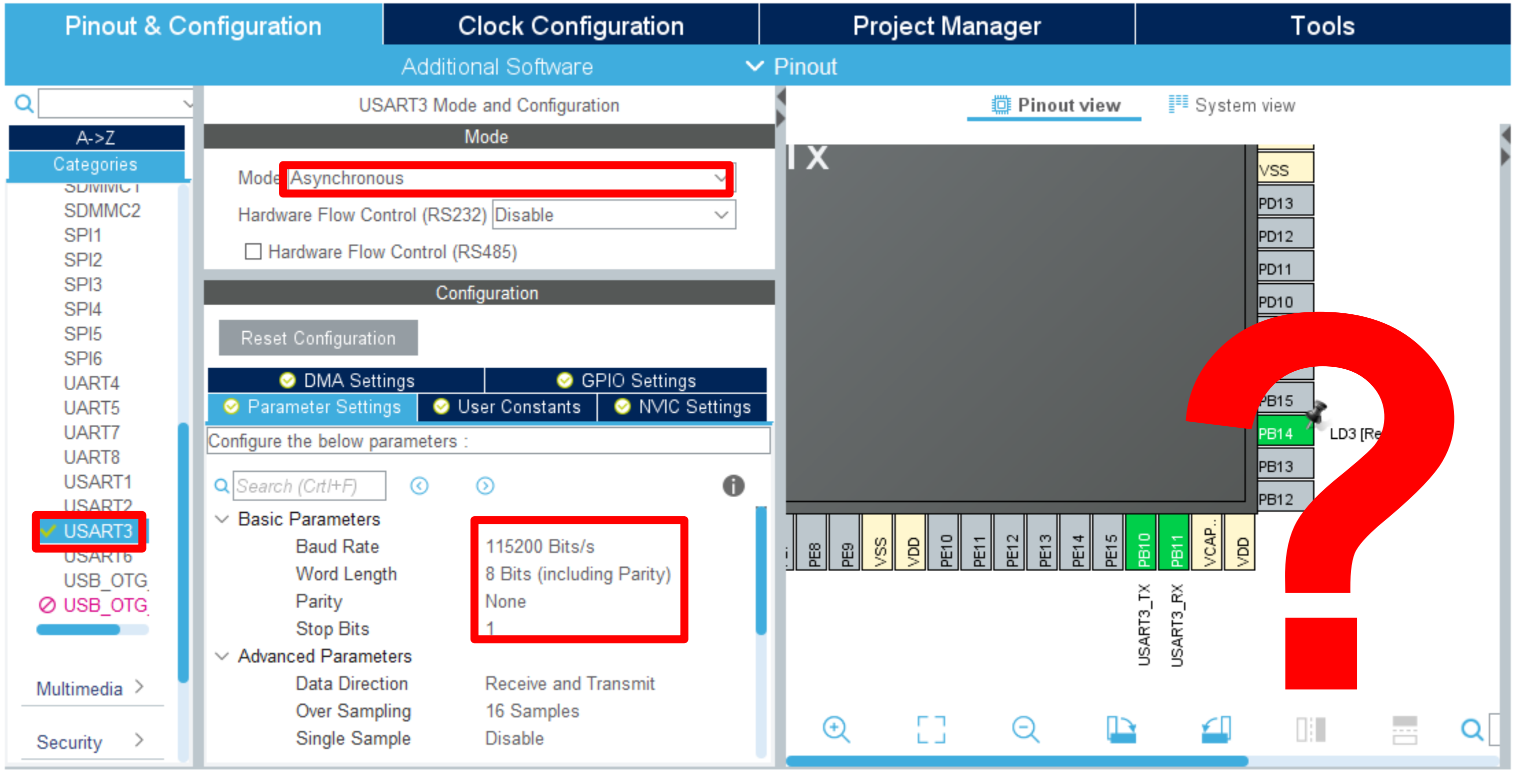
Question: The STM32F767ZI has 8 UART/USART peripherals, what we should use?

Tip: Open UM1974, go to chapter 6 (page 11), try to understand Figure 3, and after go to section 6.9 (page 26).

Answer: UART3

UART

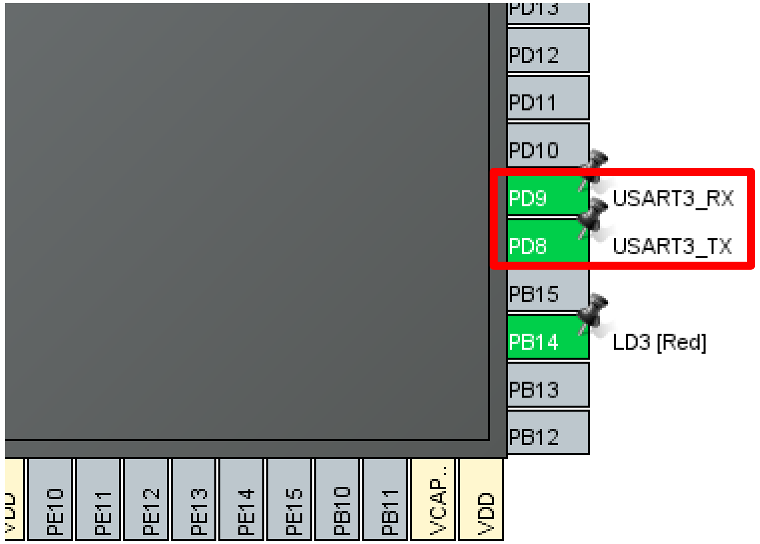
Let’s edit the project in STM32CubeMX.



UART

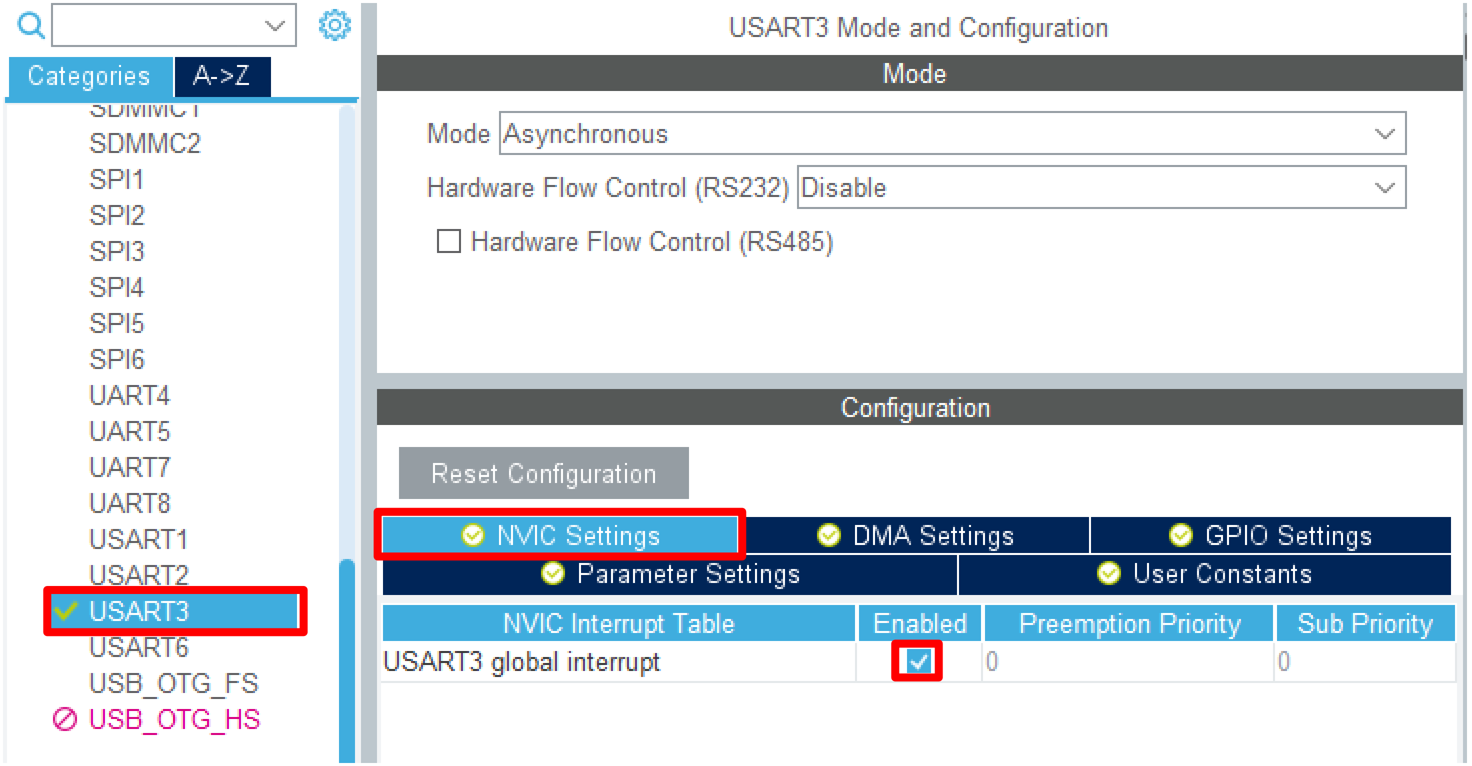
Let’s correct the GPIO Pins.

Tip: To discover alternatives pins you can press CTRL and click over the pin.



UART

Do not forget to enable the USART3 interrupt



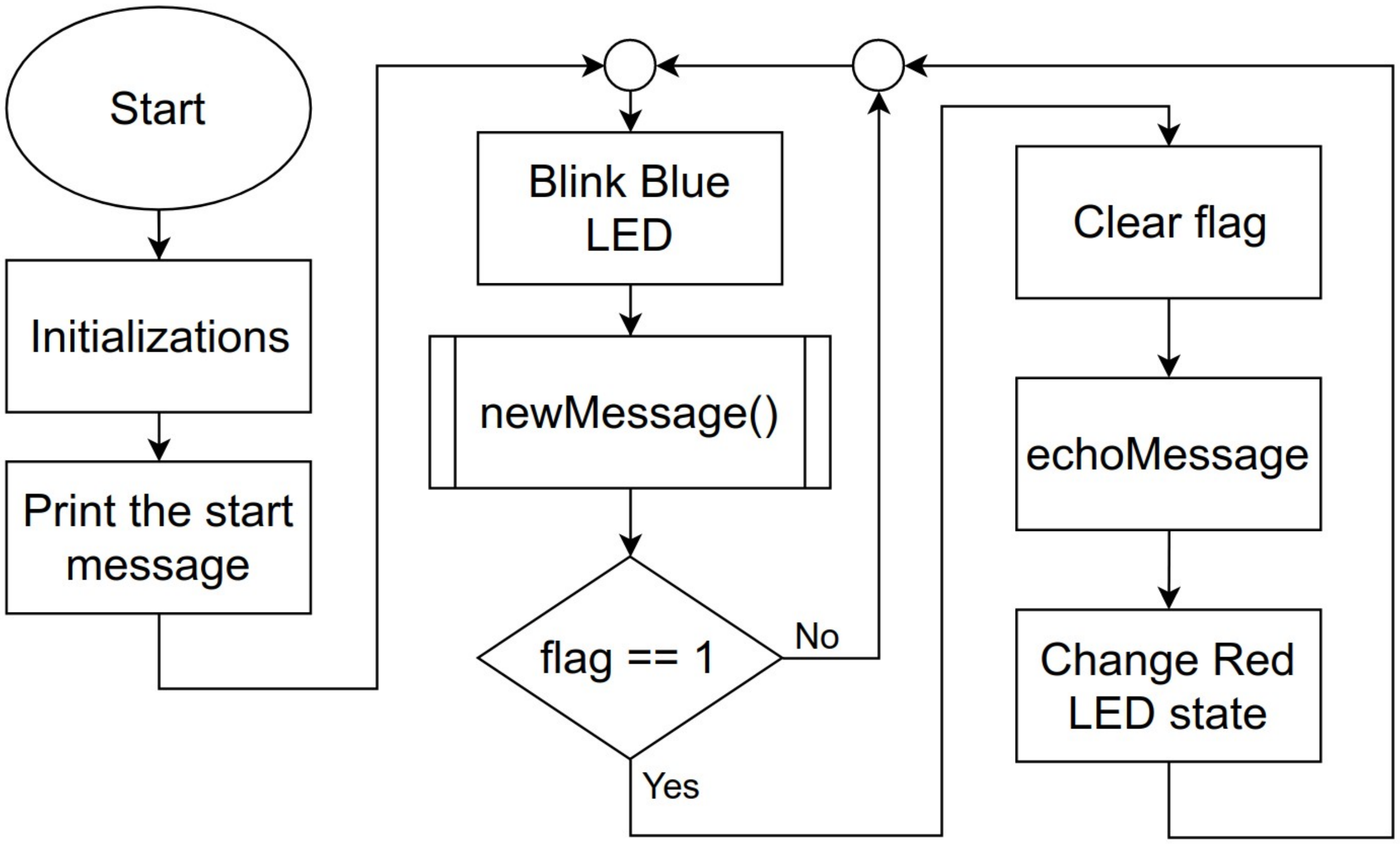
UART

Question: How to design the application?

Tip: Use some flowchart or algorithm

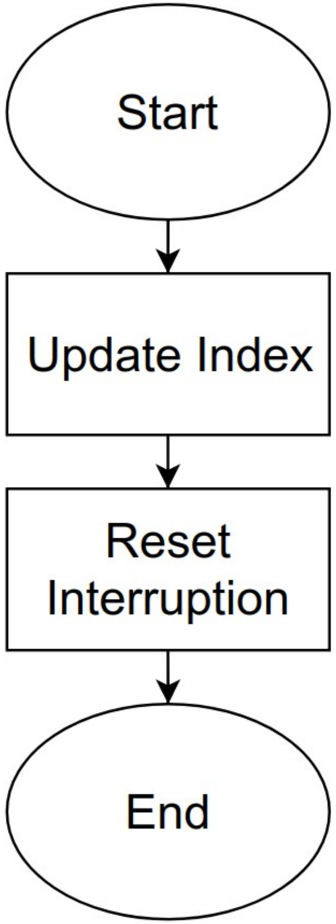
UART

main():


Start
Initializations
Print the start message
Blink Blue LED
newMessage()
flag == 1
No
Yes
Clear flag
echoMessage
Change Red LED state

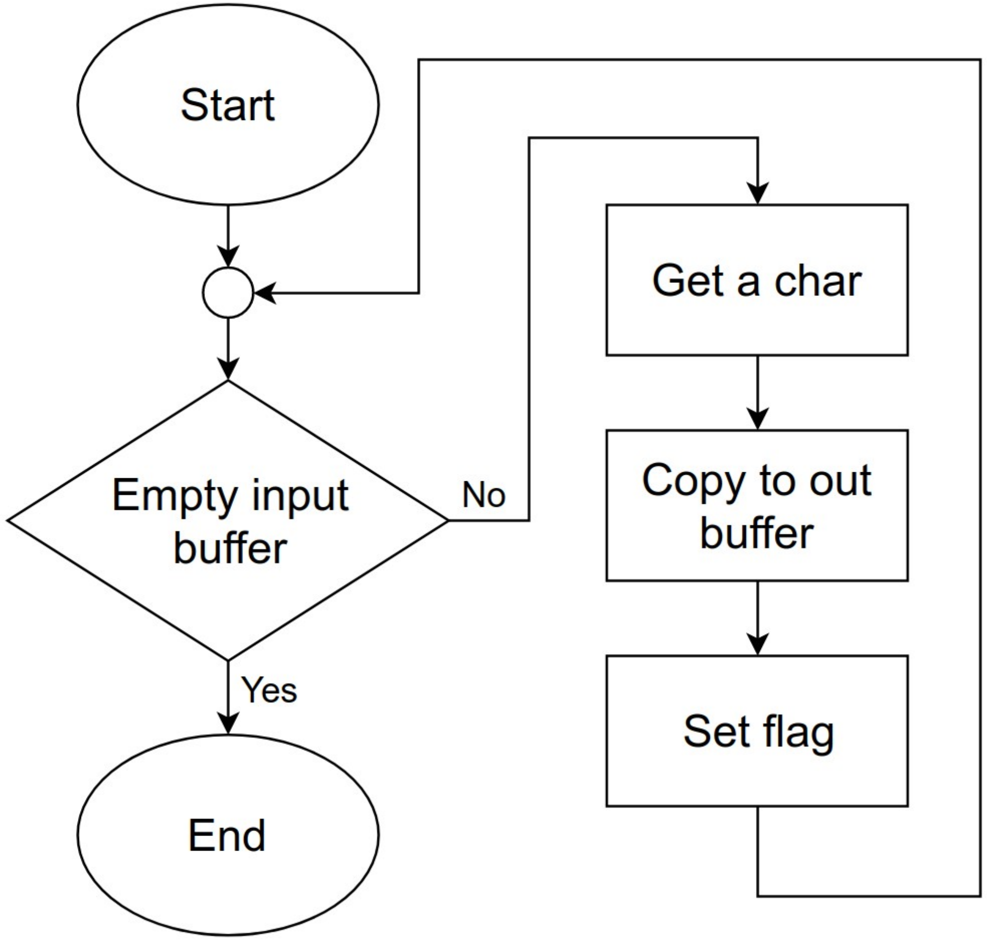
UART

USART Callback:



UART

newMessage():



UART

Let’s implement...

Usart.c

```c
/* USER CODE BEGIN 0 */

//UART3 RX data structures
uint8_t UART3Rx_Buffer[128];
uint8_t Rx_Buffer[128];
int receve_flag = 0;
volatile uint8_t UART3Rx_index = 0;
```

UART

Usart.h

```c
/* USER CODE BEGIN Private defines */
extern uint8_t Rx_Buffer[128];
extern int receve_flag;
/* USER CODE END Private defines */

void MX_USART3_UART_Init(void);

/* USER CODE BEGIN Prototypes */

void newMessage(void);
void init_UART3(void);

/* USER CODE END Prototypes */
```

UART

At usart.c file you must add some functions.

```c
/* USER CODE BEGIN 1 */

void init_UART3(){
  // set the interrupt for UART3 Rx
  HAL_UART_Receive_IT(&huart3, &UART3Rx_Buffer[UART3Rx_index], 1);
}
```

UART

The Usart Callback

```c
//implamantation of UART ISR
void HAL_UART_RxCpltCallback(UART_HandleTypeDef* huart){
  if (huart->Instance == USART3){ //current UART?
    UART3Rx_index++;
    UART3Rx_index &= ~(1<<7); //keep index inside the limits
    // set the interrupt for UART3 Rx again
    HAL_UART_Receive_IT(&huart3, &UART3Rx_Buffer[UART3Rx_index], 1);
  }
}
```

UART

newMessage()

```c
void newMessage(){
    static int local_index = 0;
    int out_index = 0;
    while(local_index != UART3Rx_index){
        Rx_Buffer[out_index] = UART3Rx_Buffer[local_index];
        out_index++;
        local_index++;
        local_index &= ~(1<<7);
        receve_flag = 1;
    }
    Rx_Buffer[out_index] = '\0';
}
```

UART

To make the printf work you must change the output of the function by redefining the fputc function in usart.c file.

```
//redifine the stdout
int fputc(int ch, FILE *f){
  HAL_UART_Transmit(&huart3, (uint8_t*)&ch, 1, 100);
  return ch;
}

/* USER CODE END 1 */
```

UART

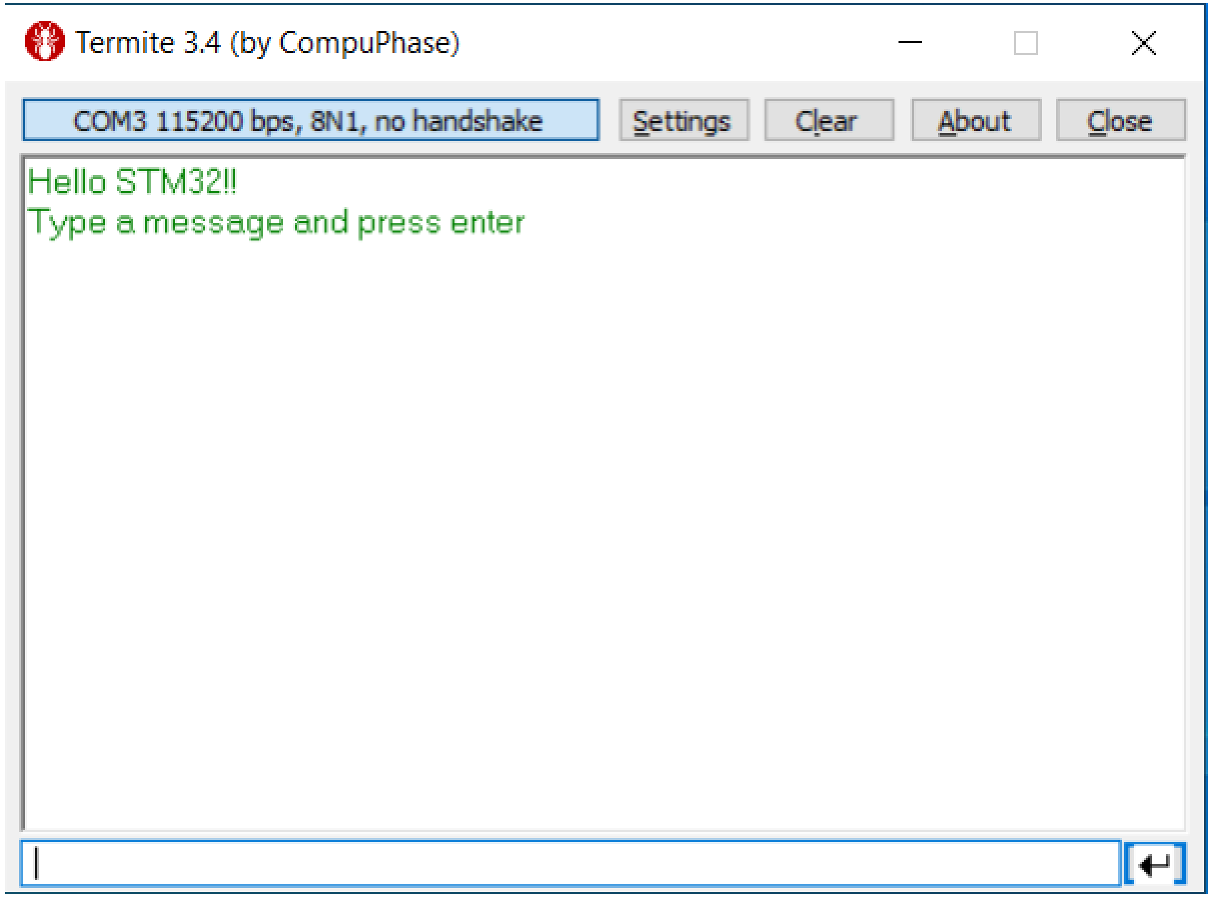
At main.c implement the main function

```
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_USART3_UART_Init();
/* USER CODE BEGIN 2 */

init_UART3();
printf("Hello STM32!!\r\n");
printf("Type a message and press enter\r\n");

/* USER CODE END 2 */
```

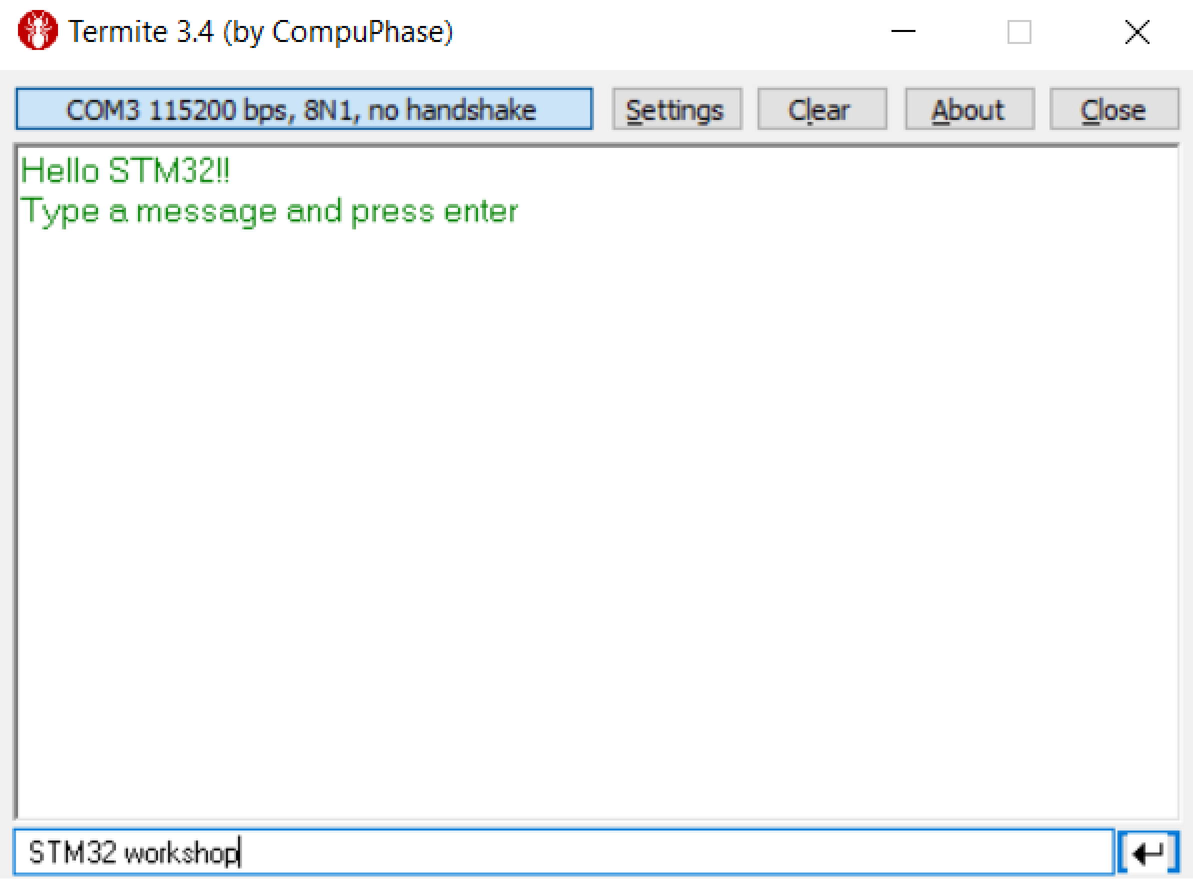
UART

```c
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
  HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_7);
  newMessage();
  if(receve_flag){
    receve_flag = 0;
    printf("The mesage was: [\"%s\"]\r\n", Rx_Buffer);
    printf("Type another message and press enter\r\n");
    HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_14);
  }
  HAL_Delay(500);
  /* USER CODE END WHILE */
```

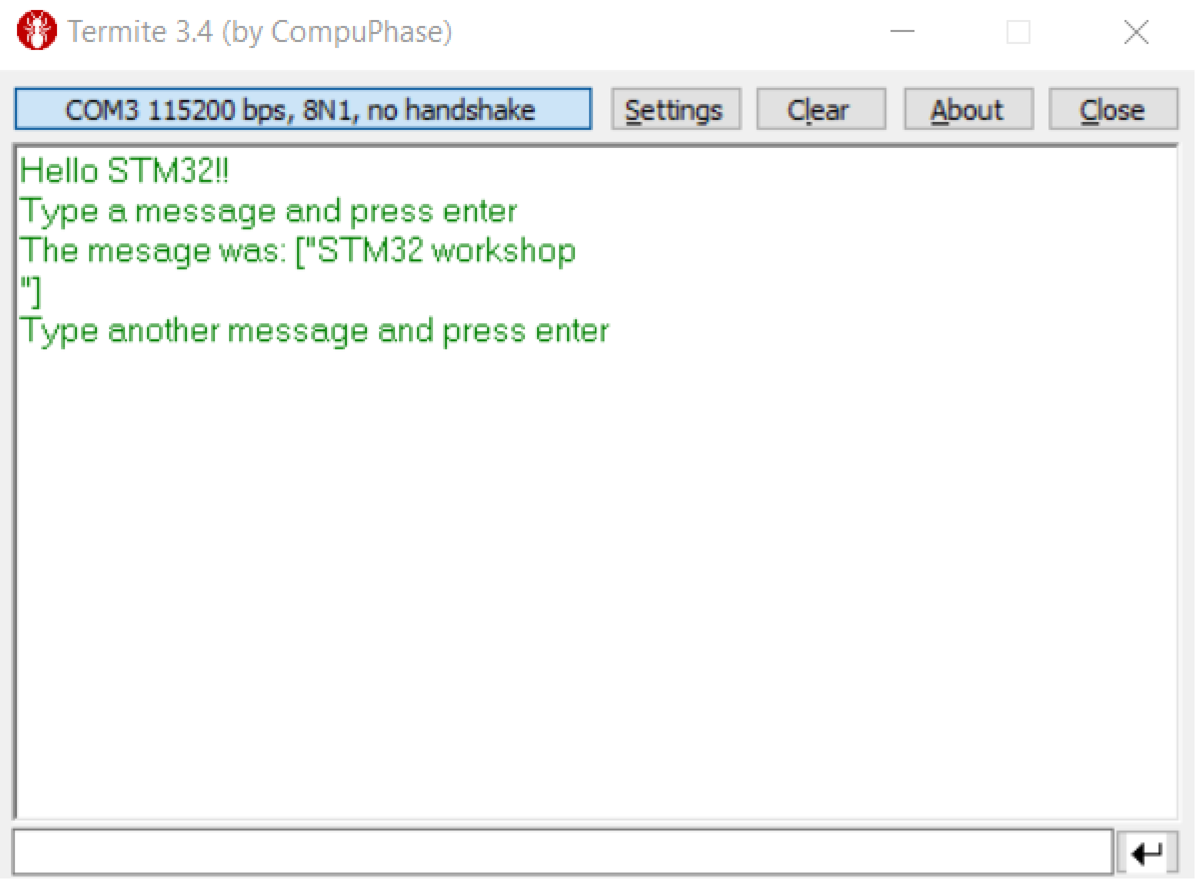
UART


Termite 3.4 (by CompuPhase)
COM3 115200 bps, 8N1, no handshake
Settings
Clear
About
Close
Hello STM32!!
Type a message and press enter

UART


Termite 3.4 (by CompuPhase)
COM3 115200 bps, 8N1, no handshake
Settings
Clear
About
Close
Hello STM32!!
Type a message and press enter
STM32 workshop

UART


Termite 3.4 (by CompuPhase)
COM3 115200 bps, 8N1, no handshake
Settings
Clear
About
Close
Hello STM32!!
Type a message and press enter
The mesage was: ["STM32 workshop
"]
Type another message and press enter

Conclusion

With this you've learned enough to continue your study of the GPIO, interrupt management and the USART peripherals.

The tools shouldn't be a problem now.

Keep in mind, there is much more to learn, this is just the beginning and the only way to improve is doing it yourselves.