



GEA-4-IF3: Event-based programming on microcontrollers

bertrand.massot@insa-lyon.fr

Course of the sessions: 10 x 2h sessions mixing theory, practical, exam and project

#	Session title	Date / Time
1	Introduction	08/09
2	Input / Output	01/10
3	Interrupt #1	06/10
4	Interrupt #2 – Time Management	07/10
5	Time Management (2)	24/11

#	Session title	Date / Time
6	Interrupt #3 – Peripherals	08/12
7	Exam (50 mn) , Project setup	16/12
8	Project #1	17/12
9	Project #2	14/01
10	Project #3, Exam (50 mn)	18/01

Use A.I. wisely – from ChatGPT itself

Why avoid AI in learning embedded programming?

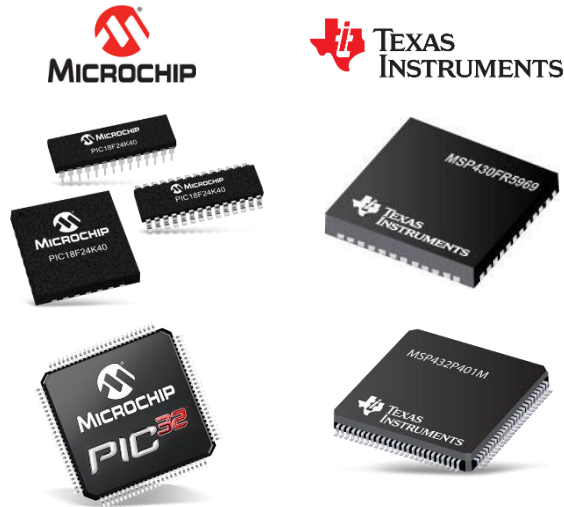
- Essential foundations: risk of copying without understanding (hardware, registers, interrupts).
- Autonomy: learning debugging and problem analysis.
- Real-world constraints: AI may ignore memory limits, real-time, power consumption.
- Reliability: generated code can look correct but be wrong or incomplete.
- Critical thinking: reading datasheets, understanding trade-offs → lasting skills.

Introduction to STM32U585 family

General introduction to the device, hardware and software used in this course

Microcontrollers and C programming

Microcontrollers are small integrated computers (Core + Peripherals)



Key figures :

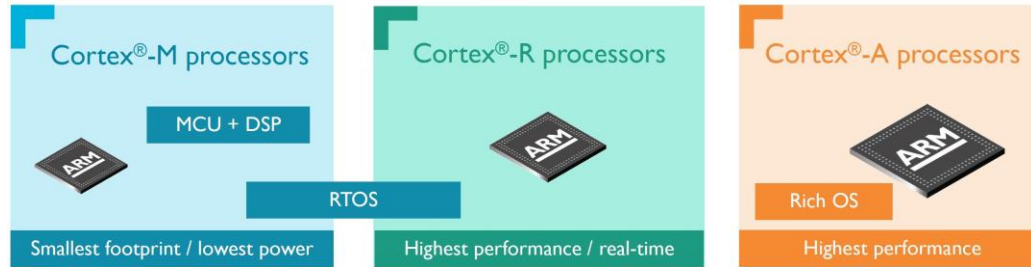
- > \$94 billion market in 2026*
- Automotive, Industrial, Consumer Electronics, Healthcare, Aerospace & Defense
- > 80% programmed in C language
- > 46 000 references on Mouser / Digikey

*<https://www.factmr.com/report/4556/microcontroller-market>

Microcontrollers and C programming

Most new microcontroller architectures are based on ARM (M0+, M4F, M23, M33)

ARM® Cortex® Processors across the Embedded Market



arm

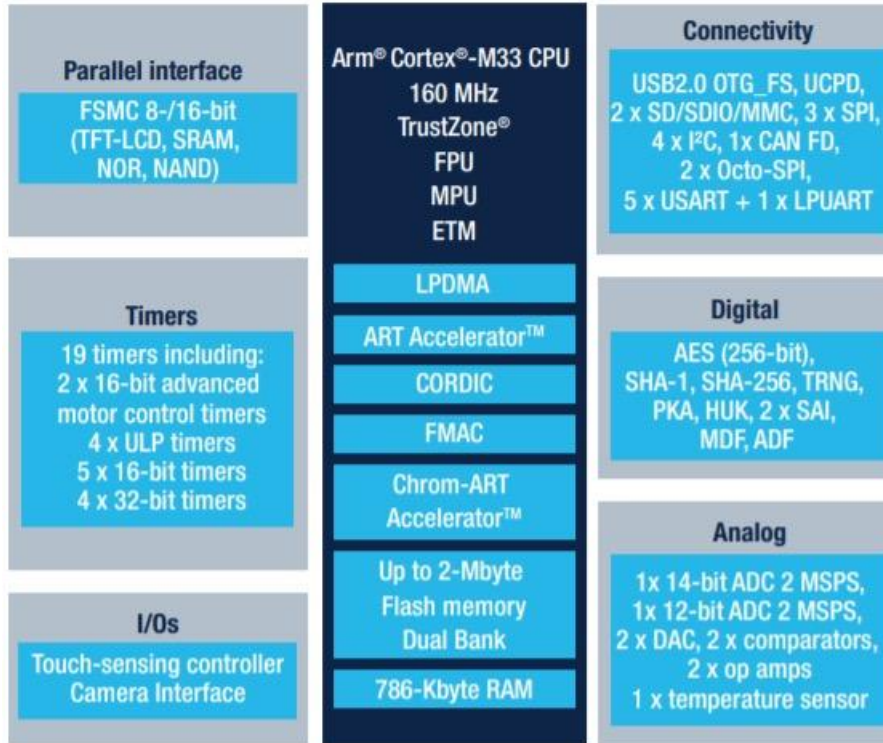


STM32 MCUs portfolio

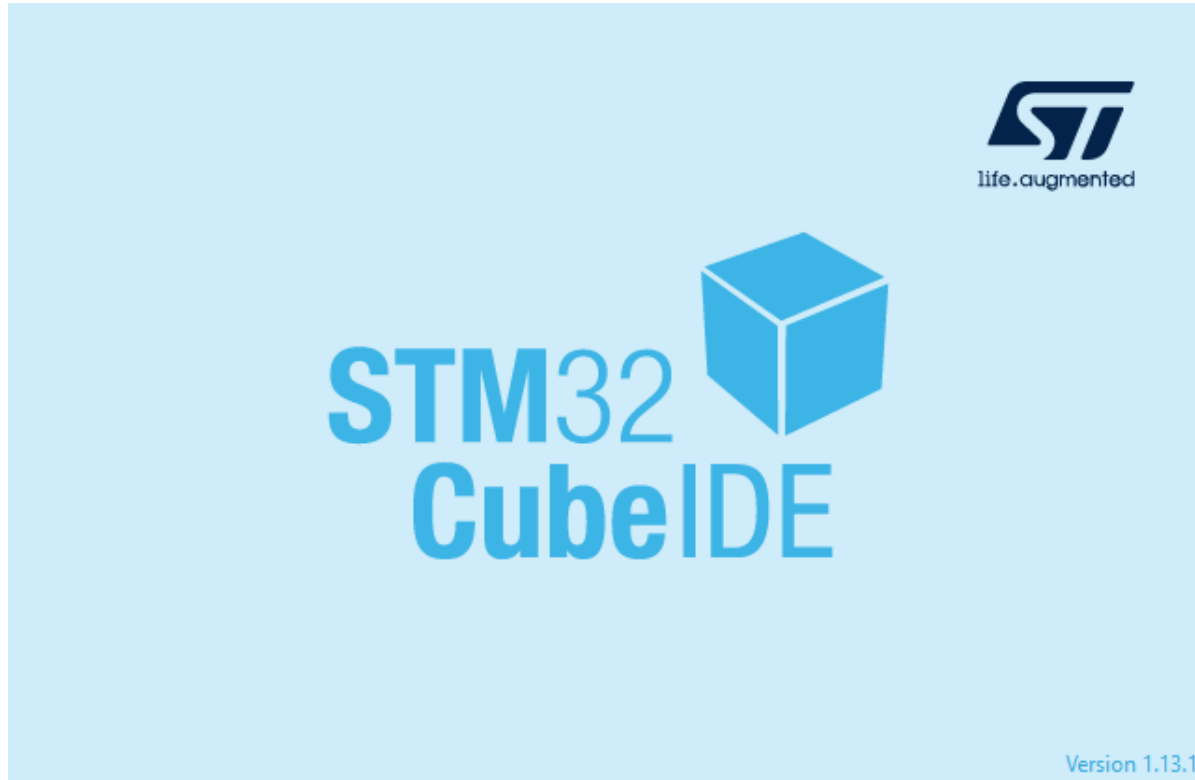
 STM32 MCUs 32-bit Arm® Cortex®-M 				
★ High Performance	STM32F2 398 CoreMark 120 MHz Cortex-M3		STM32F7 1082 CoreMark 216 MHz Cortex-M7	STM32H7 Up to 3224 CoreMark Up to 550 MHz Cortex-M7 240 MHz Cortex-M4
	STM32F4 608 CoreMark 180 MHz Cortex-M4	STM32H5 Up to 1023 CoreMark 250 MHz Cortex-M33		
»» Mainstream	STM32G0 142 CoreMark 64 MHz Cortex-M0+	STM32G4 569 CoreMark 170 MHz Cortex-M4		● Optimized for mixed-signal applications
	STM32C0 114 CoreMark 48 MHz Cortex-M0+	STM32F0 106 CoreMark 48 MHz Cortex-M0	STM32F1 177 CoreMark 72 MHz Cortex-M3	
🔋 Ultra-low-power	STM32L0 75 CoreMark 32 MHz Cortex-M0+		STM32L4 273 CoreMark 80 MHz Cortex-M4	STM32U5 651 CoreMark 160 MHz Cortex-M33
	STM32L4+ 409 CoreMark 120 MHz Cortex-M4		STM32L5 443 CoreMark 110 MHz Cortex-M33	
📶 Wireless	STM32WL 162 CoreMark 48 MHz Cortex-M4 48 MHz Cortex-M0+		STM32WB 216 CoreMark 64 MHz Cortex-M4 32 MHz Cortex-M0+	STM32WBA 407 CoreMark 100 MHz Cortex-M33
			● Cortex-M0+ Radio co-processor	

Introduction to STM32U585 architecture

STM32U585 are 32-bit microcontrollers based on Cortex-M33 core



STM32U585 are programmed using STM32CubeIDE



Introduction to STM32U585 programming

STM32U585 are programmed using STM32CubeIDE – and Cube MX

Pinout & Configuration

Clock Configuration

Project Manager

Tools

Software Packs

Pinout

GPIO Mode and Configuration

Configuration

Group By Peripherals

RCC

SPI

UART

UCPD

USART

USB

GPIO

Single Mapped Signals

ADF

DEBUG

I2C

OTG

Search Signals

Search (Ctrl+F)

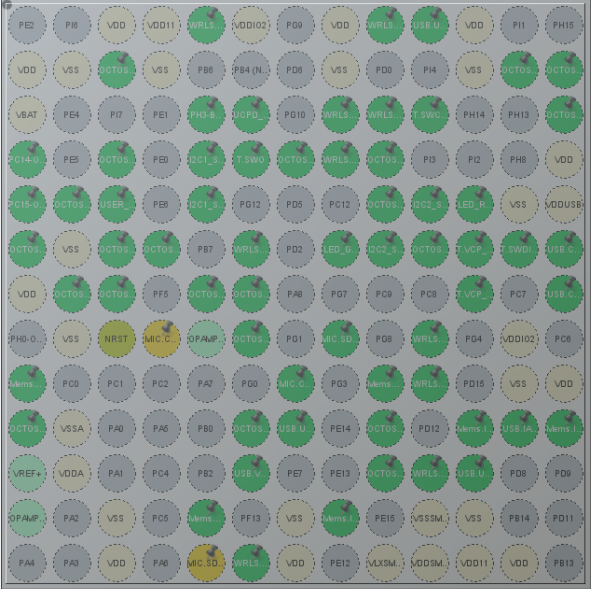
☐ Show only Modified Pins

Pin	Signal	Pin C.	Pin Pr.	GPIO	GPIO	GPIO	Maxi.	Fast	User L.	Modified
PB5	n/a	n/a	n/a	Low	Output...	No pu...	Low	n/a	UCPD...	☒
PC13	n/a	n/a	n/a	n/a	Input ...	No pu...	n/a	n/a	USER...	☒
PD10	n/a	n/a	n/a	n/a	Input ...	No pu...	n/a	n/a	Mems...	☒
PD13	n/a	n/a	n/a	n/a	Input ...	No pu...	n/a	n/a	USB.I...	☒
PD14	n/a	n/a	n/a	n/a	Input ...	No pu...	n/a	n/a	WRL...	☒
PE8	n/a	n/a	n/a	n/a	Input ...	No pu...	n/a	n/a	USB...	☒
PE11	n/a	n/a	n/a	n/a	Input ...	No pu...	n/a	n/a	Mems...	☒
PF11	n/a	n/a	n/a	Low	Output...	No pu...	Low	n/a	Mems...	☒
PF14	n/a	n/a	n/a	n/a	Input ...	No pu...	n/a	n/a	USB...	☒
PF15	n/a	n/a	n/a	Low	Output...	No pu...	Low	n/a	WRL...	☒
PG2	n/a	n/a	n/a	n/a	Input ...	No pu...	n/a	n/a	Mems...	☒
PG5	n/a	n/a	n/a	n/a	Input ...	No pu...	n/a	n/a	Mems...	☒
PG6	n/a	n/a	n/a	Low	Output...	No pu...	Low	n/a	WRL...	☒
PG15	n/a	n/a	n/a	n/a	Input ...	No pu...	n/a	n/a	WRL...	☒

Select Pins from table to configure them. Multiple selection is Allowed.

Pinout view

System view



UFBGA169 (Top view)

Search

Pinout

System

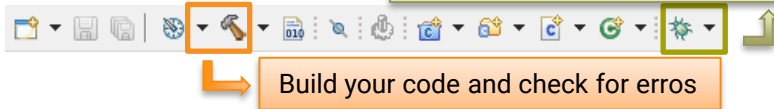
Search

Introduction to STM32U585 programming

STM32U585 are programmed using STM32CubeIDE

Toolbar in "C/C++" view

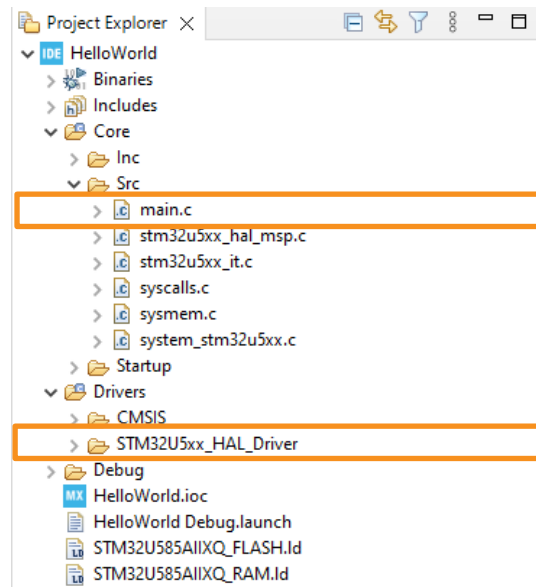
Build, program the board and start debugging



Build your code and check for errors

Edit your source files

Navigate inside your project files



```
main.c X
1
2 #include "main.h"
3
4 int main(void)
5 {
6
7     /* Infinite loop */
8     /* USER CODE BEGIN WHILE */
9     while (1)
10    {
11
12        /* USER CODE END WHILE */
13
14        /* USER CODE BEGIN 3 */
15        HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin, GPIO_PIN_RESET);
16        HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin, GPIO_PIN_SET);
17        HAL_Delay(500);
18        HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin, GPIO_PIN_SET);
19        HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin, GPIO_PIN_RESET);
20        HAL_Delay(500);
21    }
22    /* USER CODE END 3 */
23 }
```

Introduction to STM32U585 programming

STM32U585 are programmed using STM32CubeIDE

Toolbar in "Debug" view



Step by step execution when program is paused

Start, pause, stop program execution

Place breakpoints into your code to pause execution and check values

```
main.c X
135  /* Infinite loop */
136  /* USER CODE BEGIN WHILE */
137  while (1)
138  {
139
140      /* USER CODE END WHILE */
141
142      /* USER CODE BEGIN 3 */
143      HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin, GPIO_PIN_RESET);
144      HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin, GPIO_PIN_SET);
145      HAL_Delay(500);
146      HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin, GPIO_PIN_SET);
147      HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin, GPIO_PIN_RESET);
148      HAL_Delay(500);
149  }
150  /* USER CODE END 3 */
151 }
```

Manual switch perspective



STM32 programming is done in C language (or C++)

- Learned in 3GEA

The microcontroller peripherals are accessed using hardware registers

- You simply write a value at a specific address in memory to control a peripheral
- All registers are described in the *Reference Manual* (**3637 pages**)

Region description	Address range	
Code - Flash and SRAM	0x0800 0000 0x0BFF FFFF	Program instructions
	0x0C00 0000 0x0FFF FFFF	
SRAM	0x2000 0000 0x2FFF FFFF	Data memory (variables)
	0x3000 0000 0x3FFF FFFF	
Peripherals	0x4000 0000 0x4FFF FFFF	Peripheral control
	0x5000 0000 0x5FFF FFFF	

Peripheral Access Register: example for reading / writing digital input / output

32-bit MODER register at address 0x42021C00 controls pin direction of port H pins

```
((uint32_t *) (0x42021C00)) |= 0x01;    // set PH.0 as output
GPIOH->MODER |= 0x01;                    // same using stm32u585xx.h definition
```

32-bit ODR register at address 0x42021C14 controls output logic level of port H pins

```
((uint32_t *) (0x42021C14)) |= 0x01;    // set PH.0 level high
GPIOH->ODR |= 0x01;                      // same using stm32u585xx.h definition
```

What do you think ?

Introduction to STM32U585 programming

Access device's peripheral functions through a software development kit : Hardware Abstraction Layer (HAL) library

- Instead of raw registers access
- Improve readability and portability of your code
- Easier procedures for complex peripherals

Register level programming

```
void blink_led_reg(void)
{

    GPIOH->MODER |= 0x01;
    GPIOH->OTYPER &= ~0x01;

    while(1)
    {
        GPIOH->ODR ^= 0x01;
        volatile i = 7500000;
        while(i--);
    }
}
```

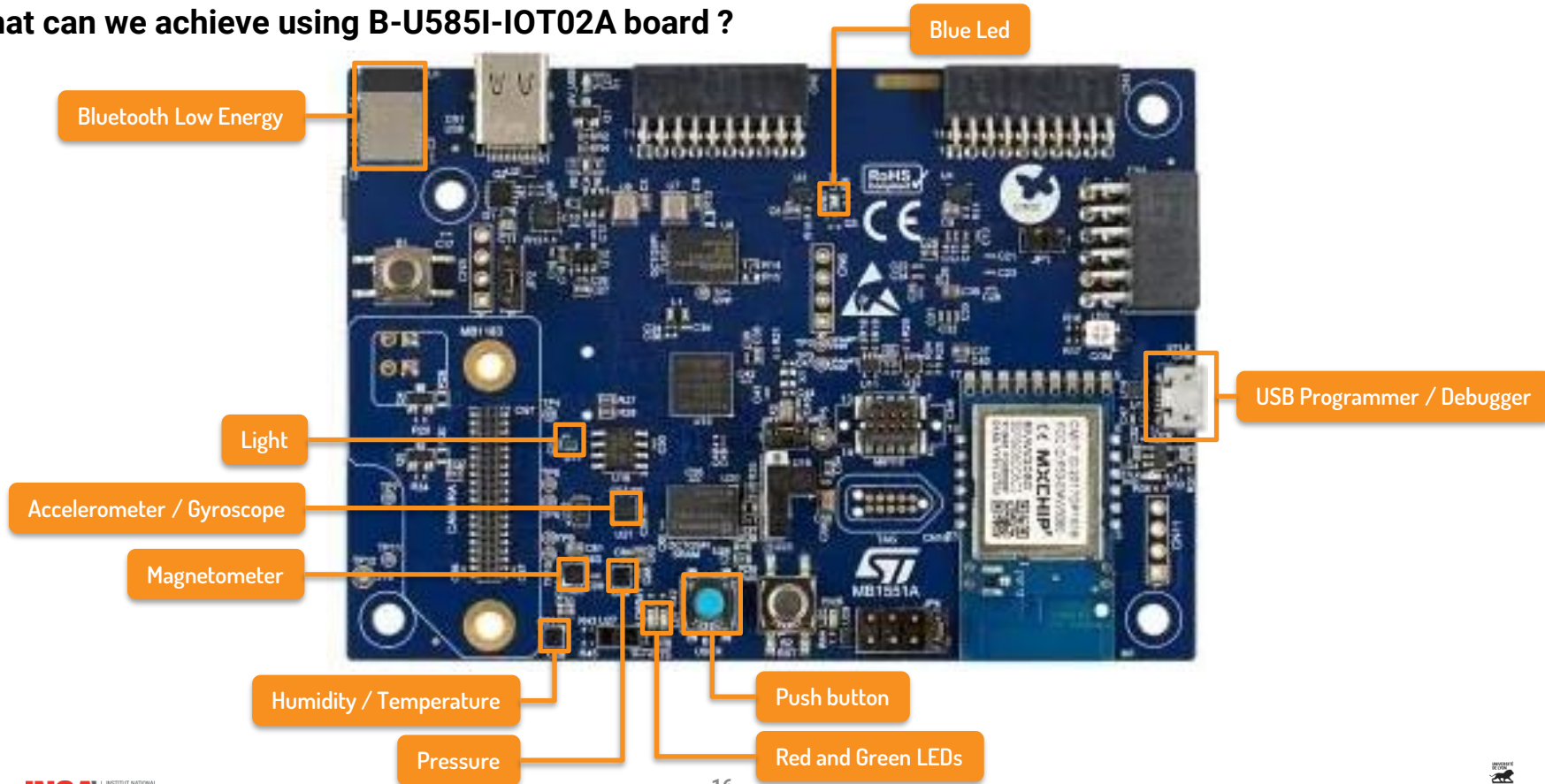
Hardware abstraction layer programming

```
void blink_led_hal(void)
{
    GPIO_InitTypeDef GPIO_InitStructure =
    {
        .Pin = GPIO_PIN_0,
        .Mode = GPIO_MODE_OUTPUT_PP
    };
    HAL_GPIO_Init(GPIOH, &GPIO_InitStructure);

    while(1)
    {
        HAL_GPIO_TogglePin(GPIOH, GPIO_PIN_0);
        HAL_Delay(500);
    }
}
```

Hardware setup

What can we achieve using B-U585I-IOT02A board ?



Reference documentation (everything is on Moodle)

- ***B-U585I-IOT02A User Manual***
 - This gives you details on the board peripherals / interconnections
 - [ST website link](#)
- ***STM32U5 HAL User Manual***
 - This explains all the functions contained in the HAL library for peripheral usage
 - [ST website link](#)
- ***STM32U585 Datasheet***
 - Describes which peripherals are implemented in this specific STM32U5 reference
 - [ST website link](#)
- ***STM32U5 Reference Manual***
 - Provides details about core / peripheral operations
 - [ST website link](#)

Practical #1

Objectives :

- Create a program which makes red and green LEDs blink alternately at 2 Hz

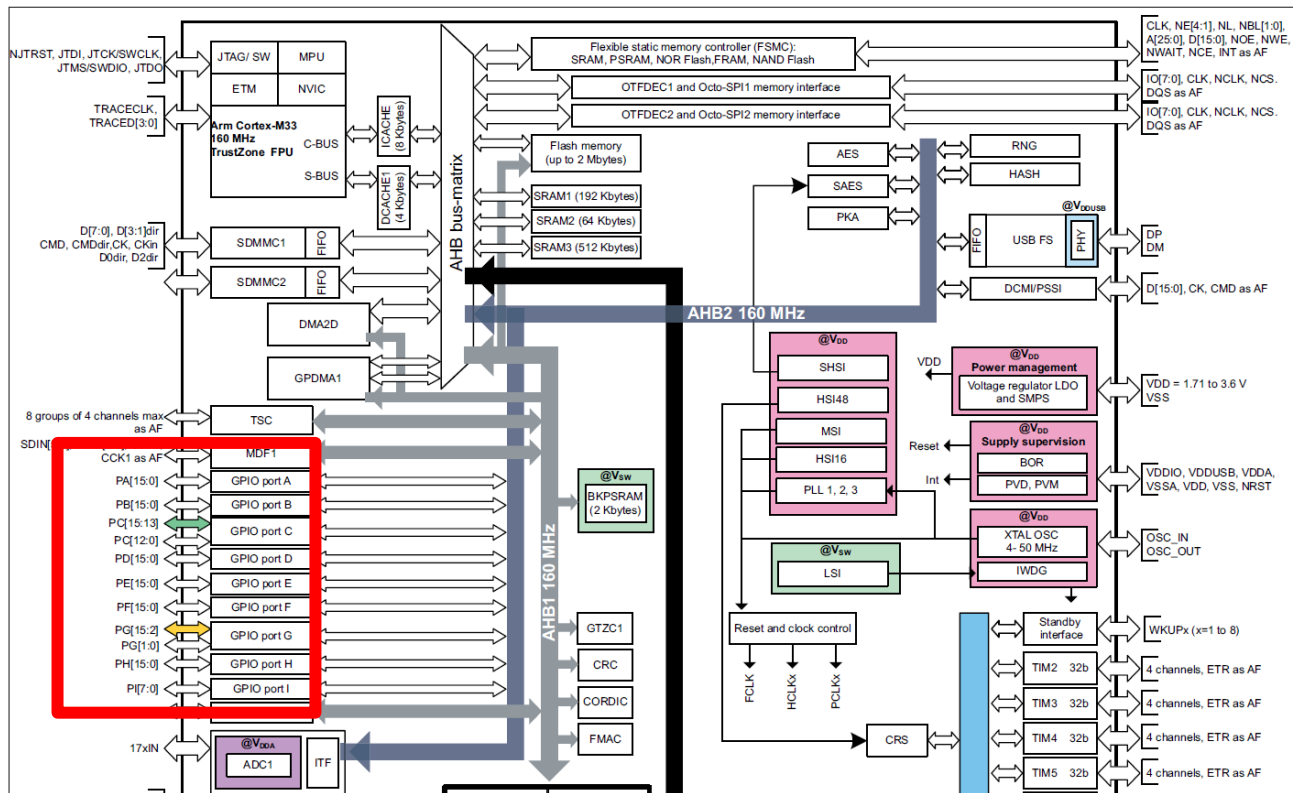
Constraints :

- Use HAL functions

General Purpose Inputs / Outputs (GPIO)

Handling digital inputs and outputs of the STM32U585

General Purpose Inputs / Outputs



General Purpose Inputs / Outputs

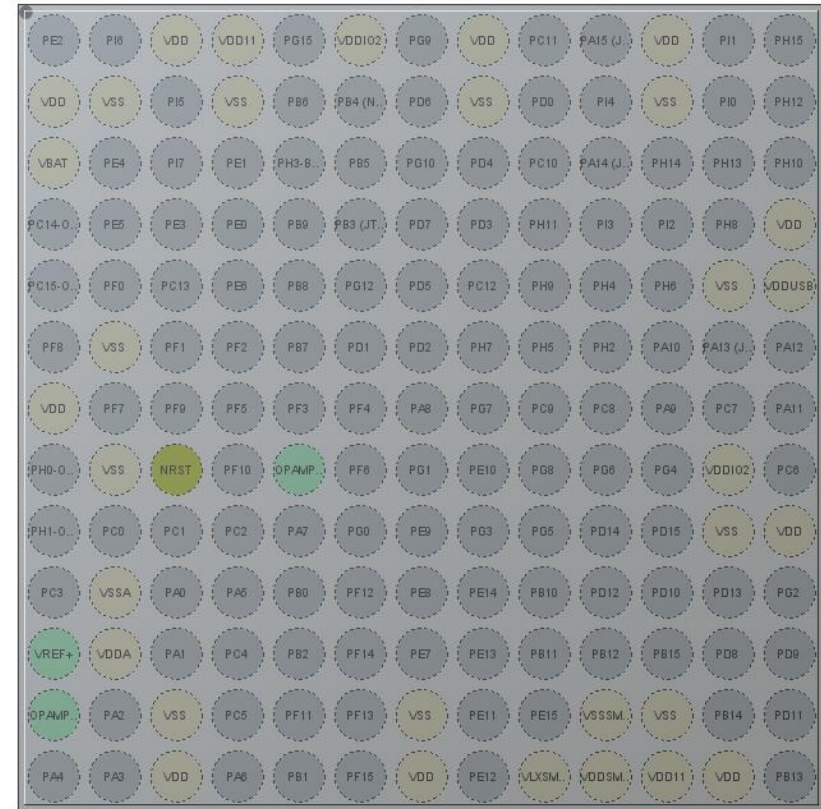
133 over 169 pins on the STM32U585AI16Q are GPIOs

Basically, each GPIO can be configured in one of the following possible states :

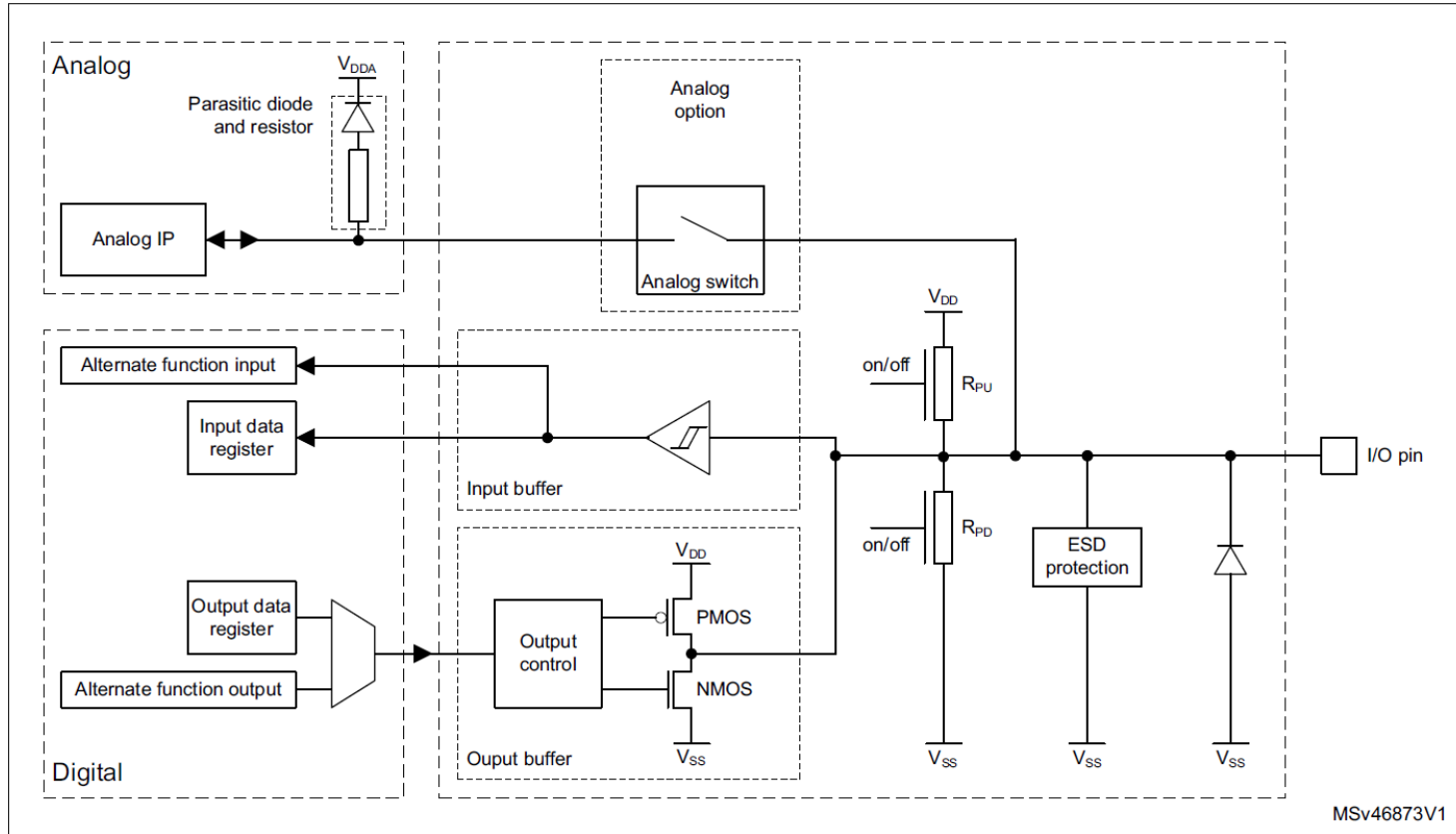
- Input floating (Hi-Z, default)
- Output low or high (logical 0 or 1)
- Analog
- Alternate function input / output

Input or Output mode can also be configured with an internal pull-up or pull-down resistance

Most of them have alternate functions other than just being a logic input or output : PWM, serial bus, ADC input, etc.



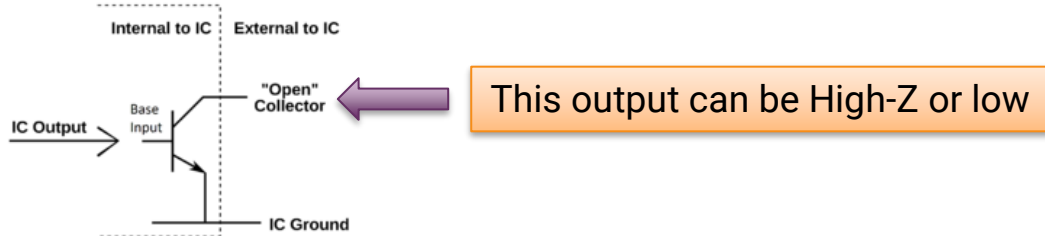
General Purpose Inputs / Outputs



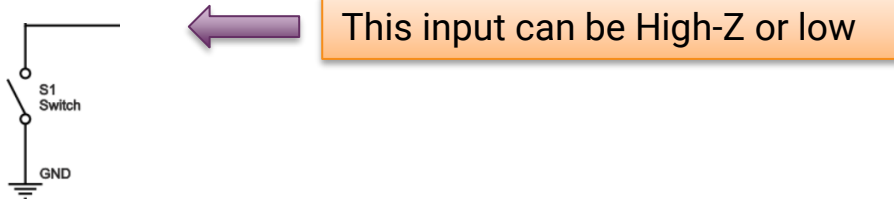
General Purpose Inputs / Outputs

Reminder on pull-up / pull-down inputs

- Useful for open-drain or open-collector devices which drives only one state (low / high)
- Commonly found in logical buses (SPI or I2C buses, SD Cards, etc.)



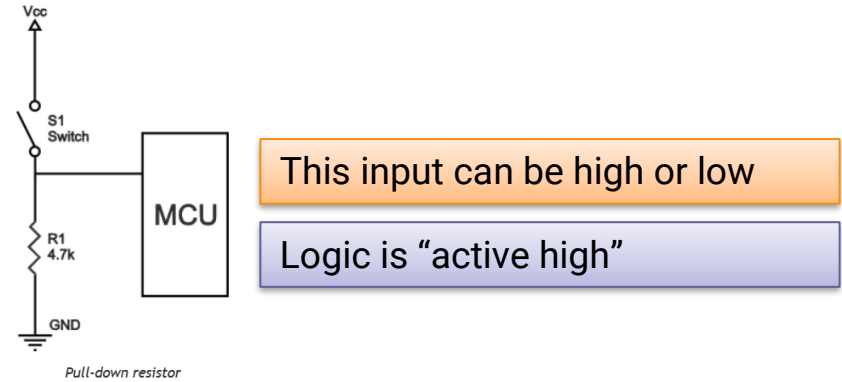
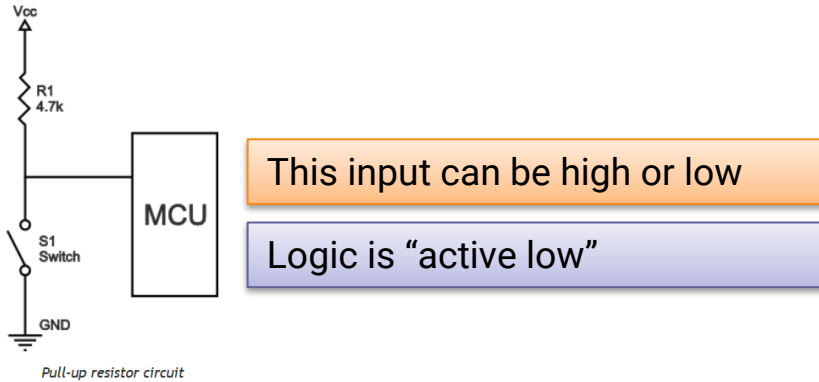
- Useful for switch buttons



General Purpose Inputs / Outputs

Reminder on pull-up / pull-down inputs

- Useful for switch buttons



- In most microcontrollers, those resistors can be set internally (no need for external components)

Cube MX configuration

Pinout & Configuration

Categories

System Core

CORTEX_M33

DCACHE1

FLASH

GPDMA1

GPIO

ICACHE

IWDG

LPDMA1

NVIC

RAMCFG

RCC

SYS

TSC

WWDG

Analog

Timers

Connectivity

Multimedia

Security

Computing

Middleware and Software Packs

Trace and Debug

Power and Thermal

PWR

Clock Configuration

Software Packs

Pinout

GPIO Mode and Configuration

Configuration

Group By Peripherals

GPIO

Search Signals

Search (Ctrl+F)

Show only Modified Pins

Pin Name	GPIO output level	GPIO mode	GPIO Pull-up/Pull-down	User Label
PC13	n/a	Input mode	No pull-up and no pull-down	USER_BUTTON
PE13	Low	Output Push Pull	No pull-up and no pull-down	LED_BLUE
PH6	Low	Output Push Pull	No pull-up and no pull-down	LED_RED
PH7	Low	Output Push Pull	No pull-up and no pull-down	LED_GREEN

PH7 Configuration :

GPIO output level

Low

GPIO mode

Output Push Pull

GPIO Pull-up/Pull-down

No pull-up and no pull-down

Maximum output speed

Low

User Label

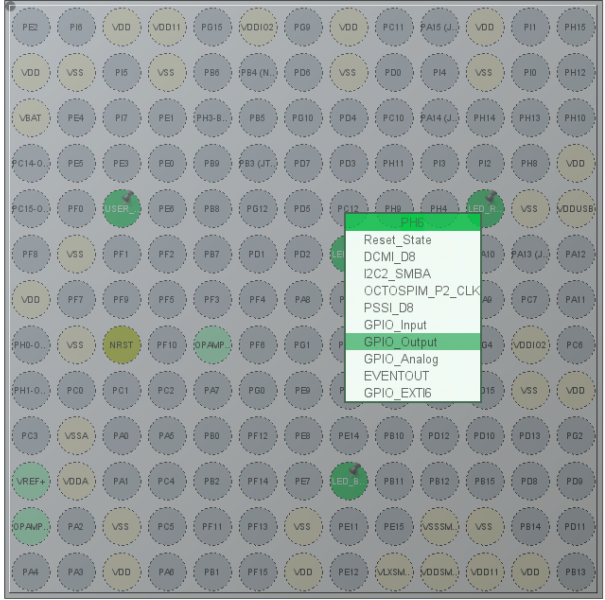
LED_GREEN

Project Manager

Tools

Pinout view

System view



UFBA169 (Top view)

General Purpose Inputs / Outputs

- With Cube MX (.ioc), configuration / initialization is automatically generated
- With the HAL library, register access is handled by the library

```
// Functions for writing pins as output
HAL_GPIO_WritePin(GPIOE, GPIO_PIN_13, GPIO_PIN_SET); // or GPIO_PIN_RESET (= 0)
HAL_GPIO_TogglePin(GPIOE, GPIO_PIN_13);

// Functions for reading pins as input
GPIO_PinState pin_state = HAL_GPIO_ReadPin(GPIOC, GPIO_PIN_13);
```

Parameters are simple, readable constants, such as `GPIOA`, `GPIO_PIN_0`, `GPIO_PIN_SET` (see `stm32u5xx_hal_gpio.h`)

Cube MX also enables user labels for GPIOs (`LED_GPIO_Port`, `LED_GPIO_Pin`)

General Purpose Inputs / Outputs

Categories A-Z

System Core

- CORTEX_M33
- DCCACHE1
- FLASH
- GPDMA1
- GPIO
- ICACHE
- IWDG
- LPDMA1
- NVIC
- RAMCFG
- RCC
- SYS
- TSC
- WWDG

Analog

Timers

Connectivity

Multimedia

Security

Computing

Middleware and Services

Configuration

Group By Peripherals

GPIO

Search Signals

Search (Ctrl...)

Show only Modified Pins

Pin Name	Signal o...	Pin Cont...	Pin Privil...	GPIO ou...	GPIO m...	GPIO Pu...	Maximu...	Fast Mode	User Label	Modified
PH6	n/a	n/a	n/a	Low	Output P...	No pull-up...	Low	n/a	LED_RED	

PH6 Configuration :

GPIO output level: Low

GPIO mode: Output Push Pull

GPIO Pull-up/Pull-down: No pull-up and no pull-down

Maximum output speed: Low

User Label: LED_RED

```
HAL_GPIO_WritePin(LED_RED_GPIO_Port,  
LED_RED_Pin,  
GPIO_PIN_SET);
```

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
/* USER CODE END WHILE */

/* USER CODE BEGIN 3 */
}
/* USER CODE END 3 */
```



NEVER write code outside
of BEGIN / END zones

- Only code within BEGIN / END zone will be kept after new generation from Cube MX
- Code outside will be overwritten

Cube MX configuration

The screenshot shows the STM32CubeMX interface. The 'MCU/MPU Selector' tab is active, and the 'Commercial Part Number' dropdown is set to 'STM32U585AI6Q'. The 'PRODUCT INFO' section on the left lists various attributes like Segment, Series, Line, Marketing Status, Price, Package, and Core. The main area displays the 'STM32U5 Series' details, including the 'STM32U585AI6Q' model, its features (Ultra-low-power with FPU Arm Cortex-M33 MCU with TrustZone, 160 MHz with 2 Mbytes of Flash memory), unit price (5.3187), and boards (B-U585H-IOT02A - STEVAL-STWINBX1). Below this, a table lists the 'MCUs/MPUs List: 2 items', with the first item, 'STM32U585AI6Q', highlighted in blue and red.

MCU/MPU Selector | Board Selector | Example Selector | Cross Selector

Commercial Part Number: **STM32U585AI6Q**

PRODUCT INFO

- Segment
- Series
- Line
- Marketing Status
- Price
- Package
- Core

STM32U5 Series

STM32U585AI6Q Ultra-low-power with FPU Arm Cortex-M33 MCU with TrustZone, 160 MHz with 2 Mbytes of Flash memory

ACTIVE Product is in mass production

Unit Price for 10kU (US\$) : 5.3187

Boards: [B-U585H-IOT02A](#) - [STEVAL-STWINBX1](#)

UFPGA 169 7x7x0.6 P 0.5 mm

MCUs/MPUs List: 2 items

	Part Number	Status	Unit Price	Boards	Flash	SRAM	EEPROM	Frequency
★	STM32U585AI6Q	Active	5.3187	B-U585H-IOT02A - STEVAL-STWINBX1	2048 kBytes	786 kBytes	133	160 MHz
★	STM32U585AI6Q	Active	5.3187	B-U585H-IOT02A - STEVAL-STWINBX1	2048 kBytes	786 kBytes	133	160 MHz

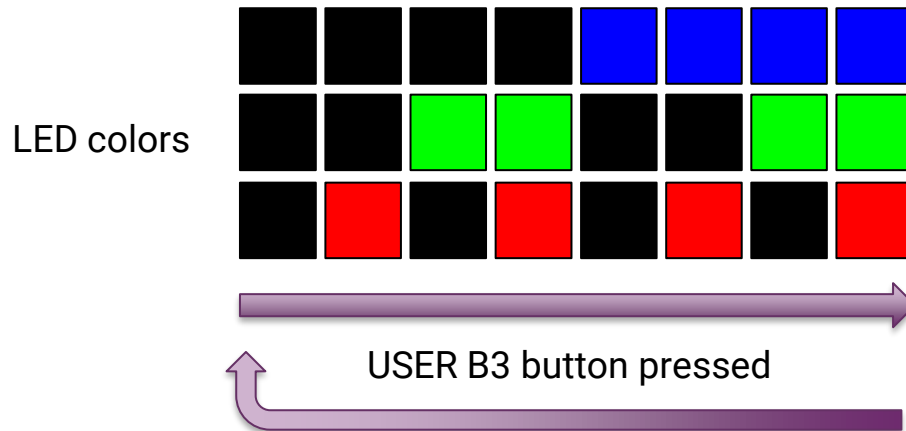
Practical #2

Objectives :

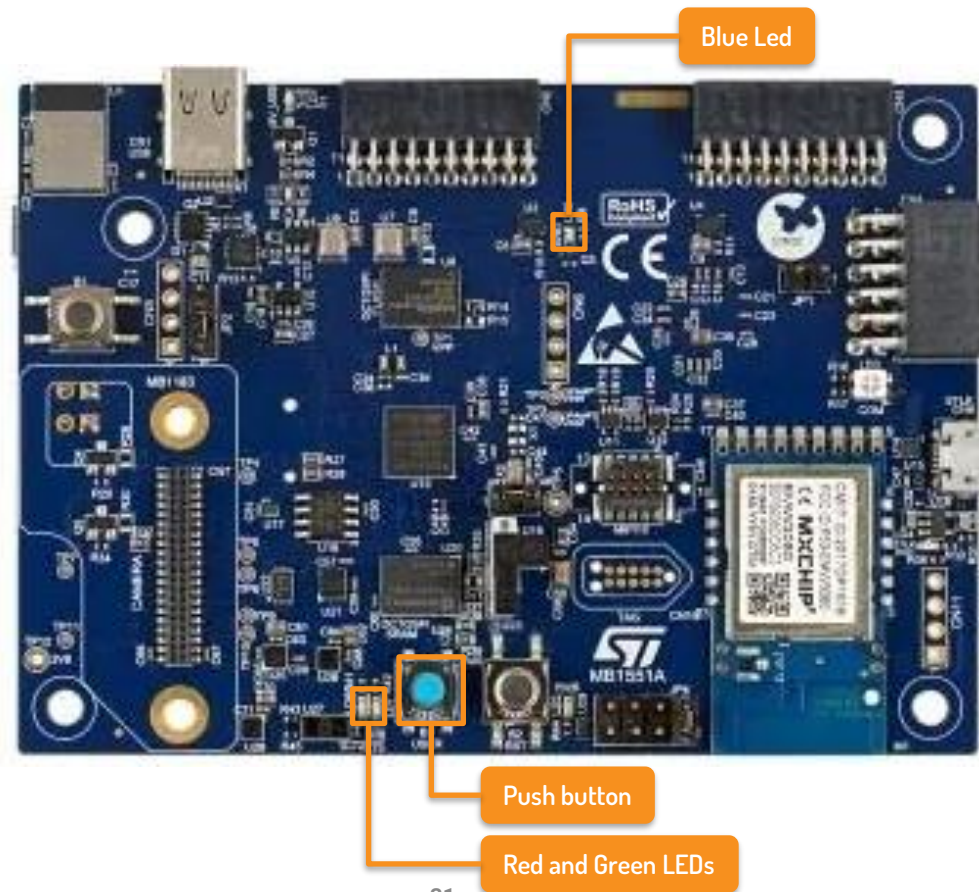
- Implement a program to control color of the red, green and blue LED using the switch button

Constraints :

- Start with empty Cube MX (select STM32U585AI16Q)
- Use HAL library (no direct access to peripheral registers)
- Use GPIO input / output only

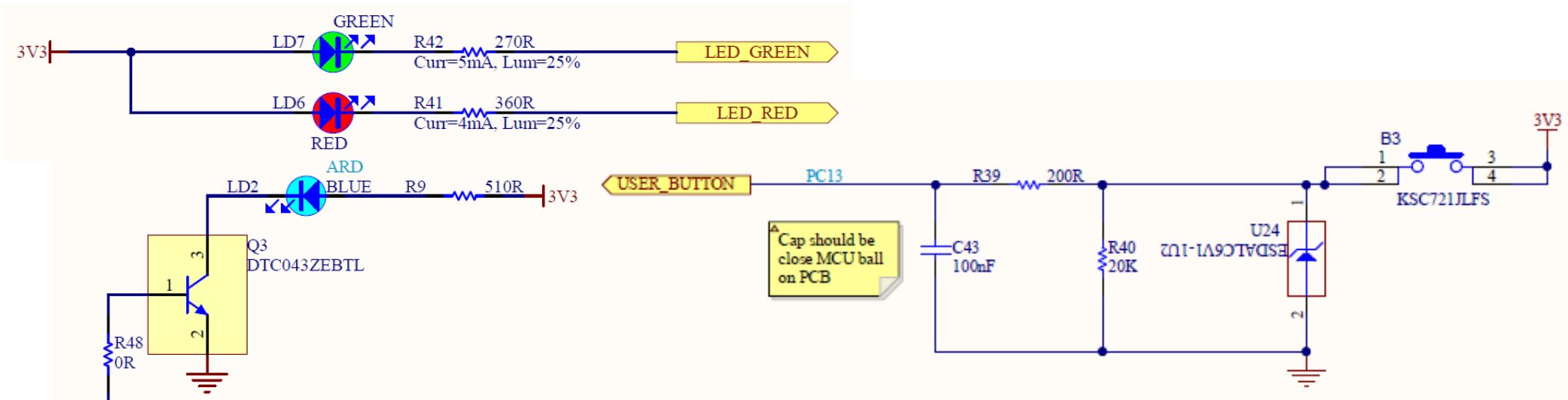


Practical #2



Practical #2

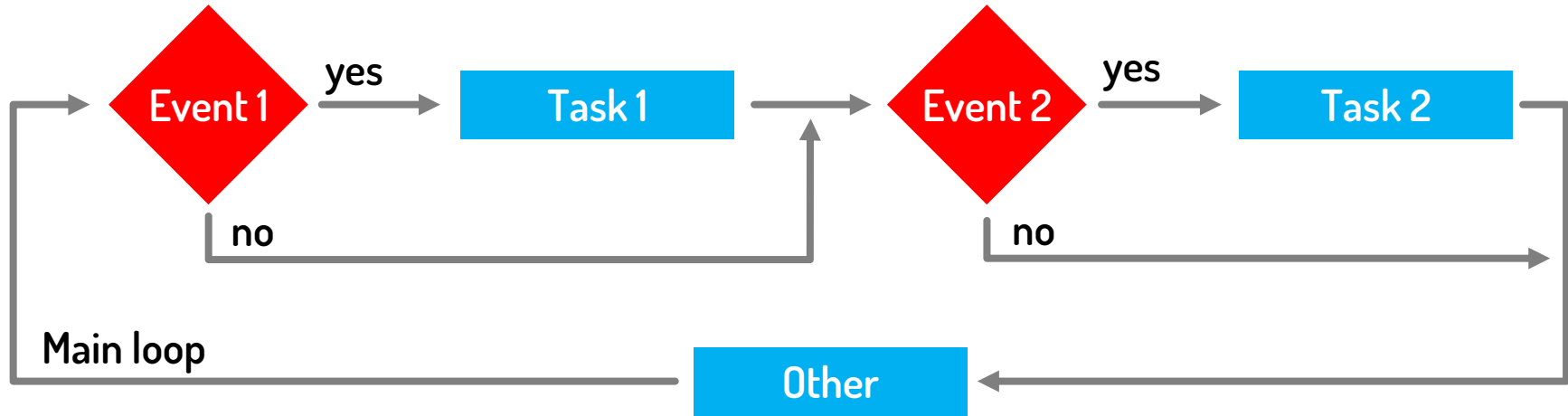
I/O	Reference	Color	Name	Comment
PC13	B3	Blue	USER	User button
PH7	LD7	Green	LD7	User LED lights up when PH7 is set to 0.
PH6	LD6	Red	LD6	User LED lights up when PH6 is set to 0.
PE13	LD2	Blue	ARD	ARDUINO® LED lights up when PE13 is set to 1.



Interruptions - Part #1

Basic mechanisms of interruptions and event-based programming principles

Executing a program sequentially without interrupts is called *polling*



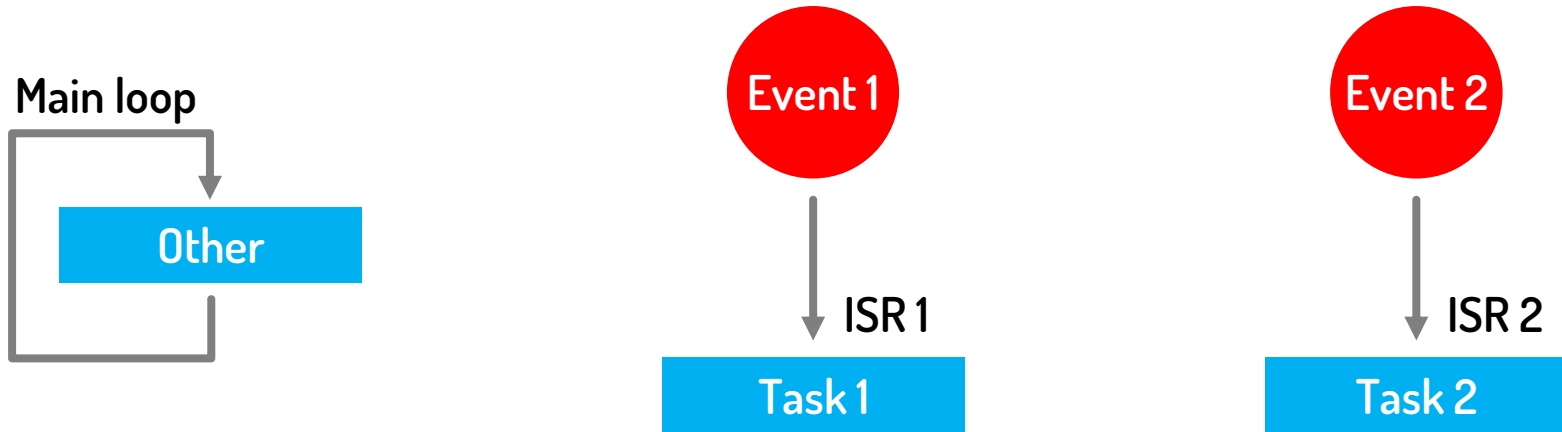
- Events occurring during execution, such as a button pushed, are detected by successive testing of the inputs in a loop.
- When an event is detected, an appropriate function is called to handle the required behavior
- Meanwhile other events can not be detected anymore !
- CPU is **always busy**

Introduction to hardware interrupts

On events, interruptions can stop the current running program and execute a sub-program

The current program will be resumed after execution of the sub-program.

Such a sub-program is called an **Interrupt Service Routine (ISR)** or an **Interrupt Handler**

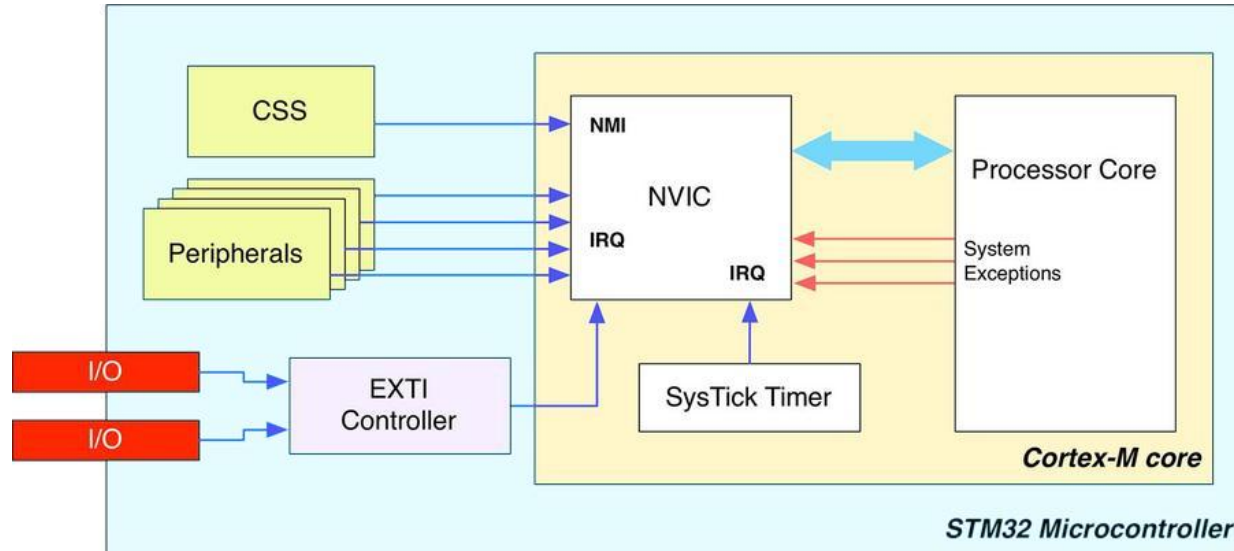


- Whatever the program is doing, events will be detected and handled accordingly
- This provides much more reactivity for tasks with high **priority**
- The CPU can be put in **idle state** if there is no action to do, except waiting for events

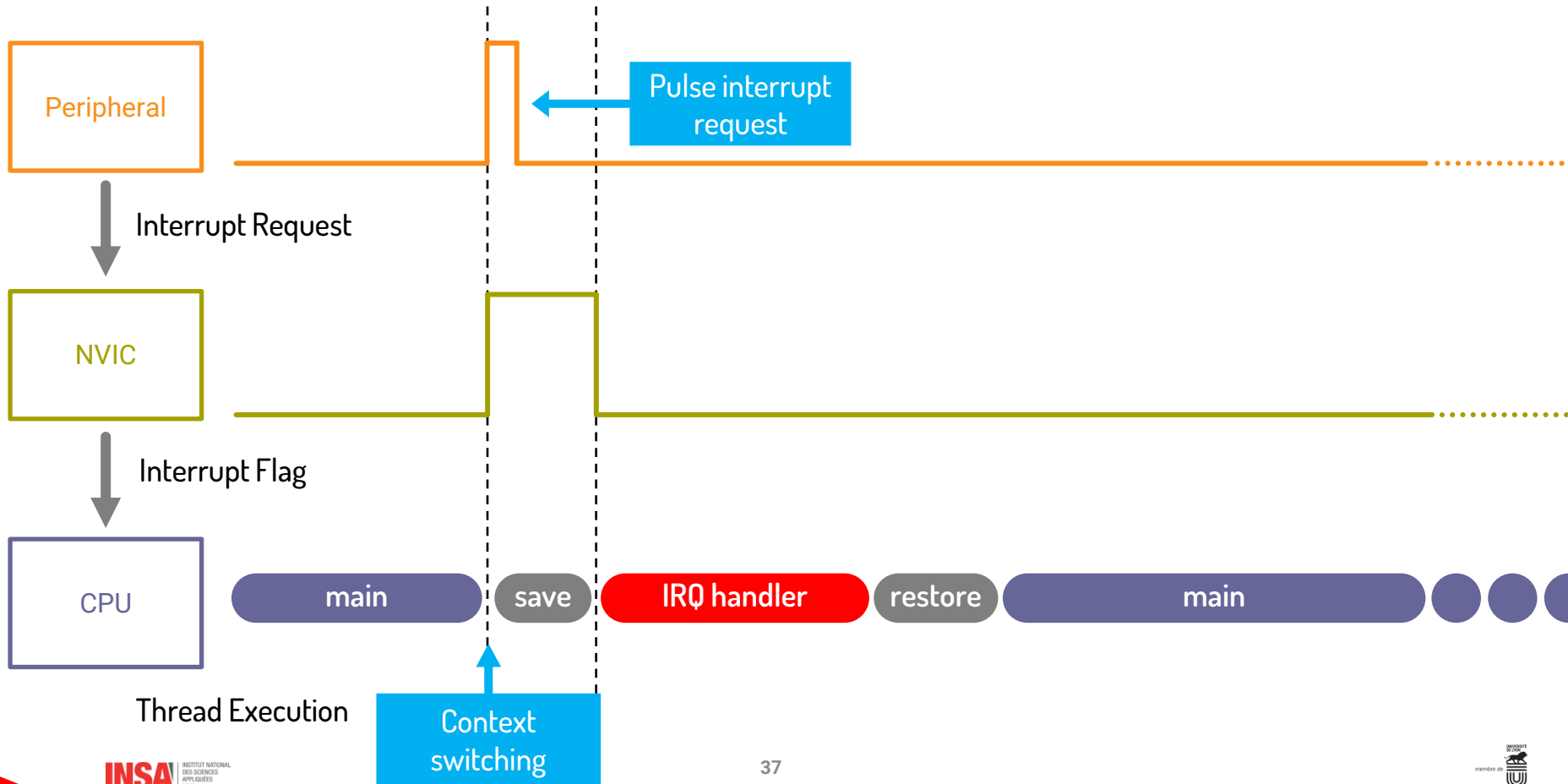
A dedicated hardware block is responsible for handling interrupts

This block is called the **Nested Vector Interrupt Controller (NVIC)**

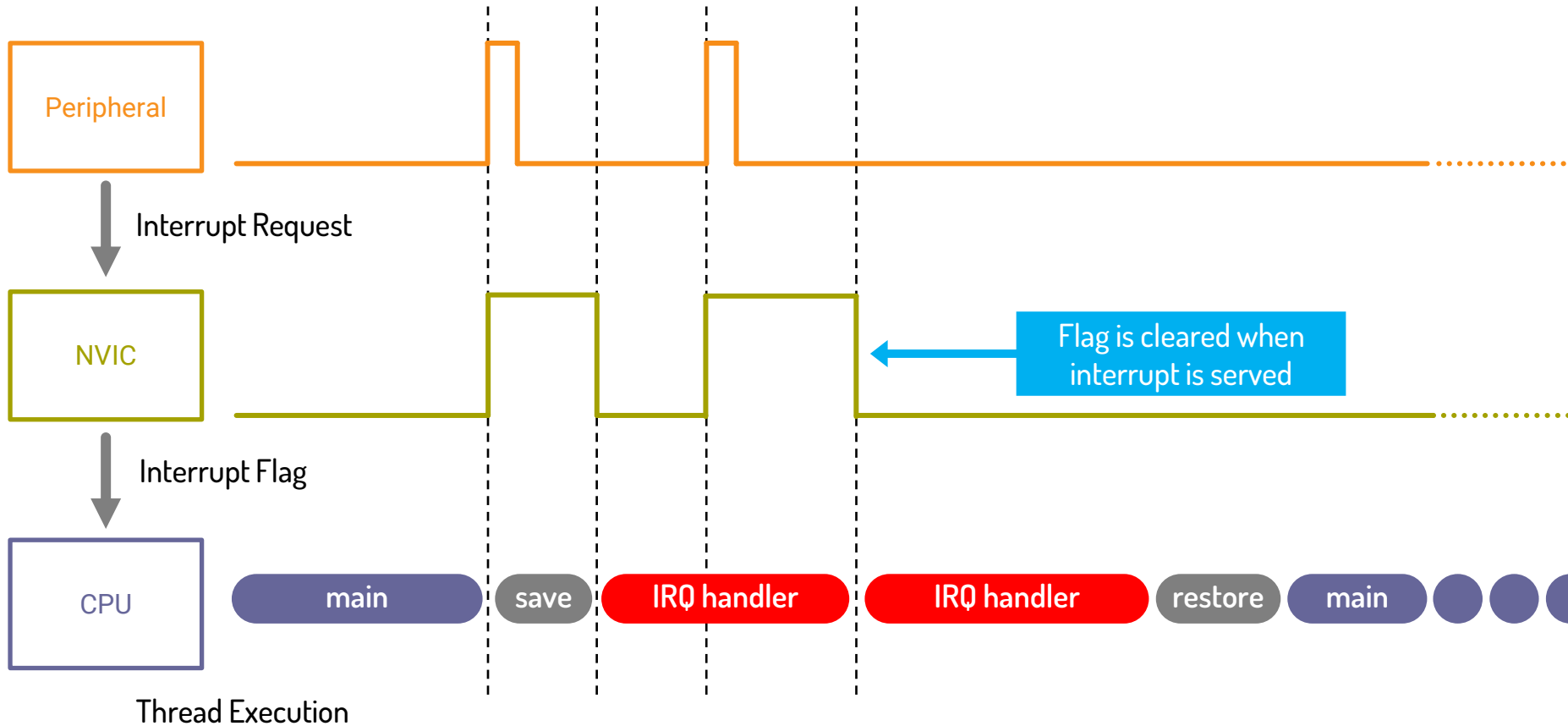
- The NVIC collects all interrupt signals from peripherals
- Saves current CPU state (registers, stack, etc.)
- Sets the next instruction pointer to the address of the *interrupt service routine*
- Restores previous saved CPU state when the ISR returns



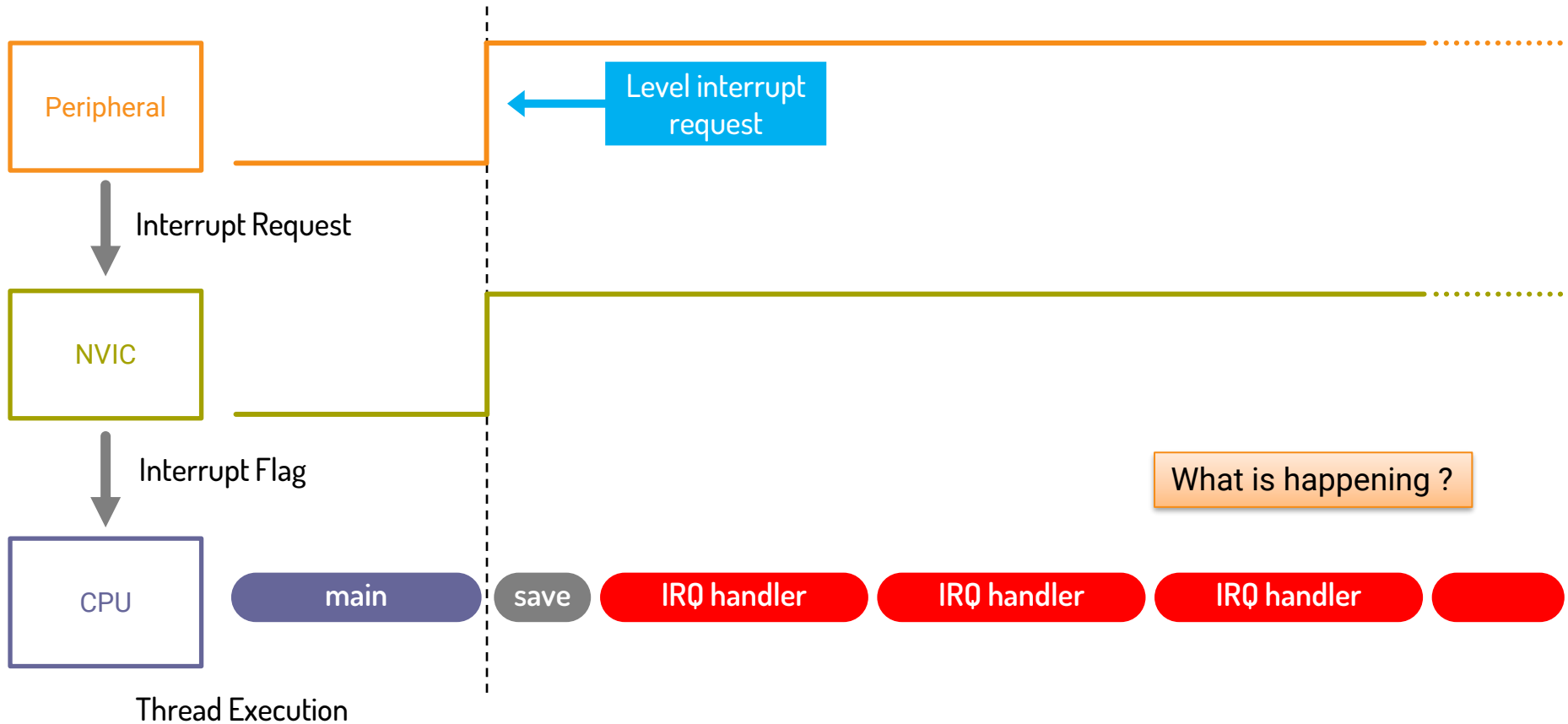
Serving a single Interrupt Request (IRQ)



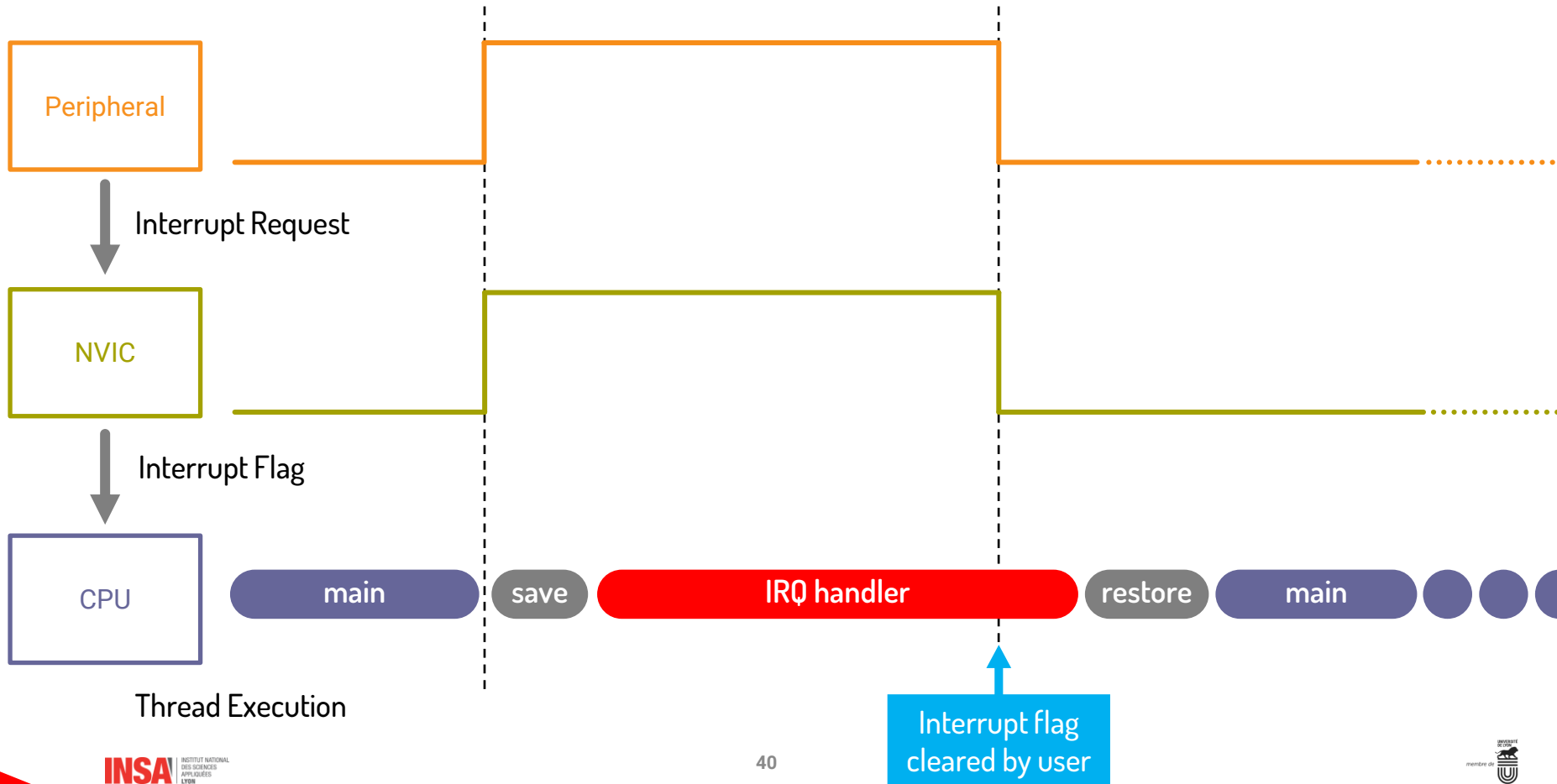
Serving a single Interrupt Request (IRQ)



Serving a single Interrupt Request (IRQ)



Serving a single Interrupt Request (IRQ)



Interrupt handling in STM32U585

Some examples of interrupt sources:

- An input pin changing state (low to high, or high to low)
- ADC end of sample conversion
- Timer expiration
- Data received on serial bus
- Etc.

Most of the peripheral events which occur in your program can be detected with interruptions !

It is then possible to build your application based on events without continuously checking them

Using STM32 Cube MX, code is usually already generated for flag clearance

Configuration				
☒ NVIC	☒ Code generation			
Enabled interrupt table	☐ Select for in...	Generate Enable in Init	☒ Generate IRQ handler	Call HAL handler
Time base: System tick timer	☐	☐	☒	☒
EXTI Line13 interrupt	☐	☒	☒	☒

Interrupt handling in STM32U585

The NVIC contains an Interrupt Vector Table, which stores addresses of ISRs for each interrupt source

Interrupt Number in the table	Interrupt Source	
-15	-	
-14	Reset	
-13	Non Maskable Interrupt	
-12	Hard Fault	
-11	Memory Management	
[...]	[...]	
-1	SysTick	
[...]	[...]	
11	EXTI1 (GPIOx Pin 1)	
12	EXTI2 (GPIOx Pin 2)	
[...]	[...]	

Specific / CPU exceptions (1 – 15)

External / Peripheral interrupts (16 – 255)

When an interrupt occurs, the main program is immediately stopped, and the corresponding handler is called

Interrupt handling in STM32U585

Interrupts can be configured and activated from Cube MX depending on peripheral configuration

✔ NVIC ✔ Code generation

Priority Group 4 bits for pre-emption priority 0 bit... ☐ Sort by Preemption Priority and Sub Priority ☐ Sort by interrupts names

Search ⏪ ⏩ Show available interrupts ☒ Force DMA channels Interrupts

NVIC Interrupt Table	Enabled	Preemption Priority	Sub Priority
Non maskable interrupt	☒	0	0
Hard fault interrupt	☒	0	0
Memory management fault	☒	0	0
Prefetch fault, memory access fault	☒	0	0
Undefined instruction or illegal state	☒	0	0
System service call via SWI instruction	☒	0	0
Debug monitor	☒	0	0
Pendable request for system service	☒	0	0
Time base: System tick timer	☒	15	0
Flash non-secure global interrupt	☐	0	0
RCC non-secure global interrupt	☐	0	0
EXTI Line13 interrupt	☒	0	0
FPU global interrupt	☐	0	0
Instruction cache global interrupt	☐	0	0

Interrupt handling in STM32U585

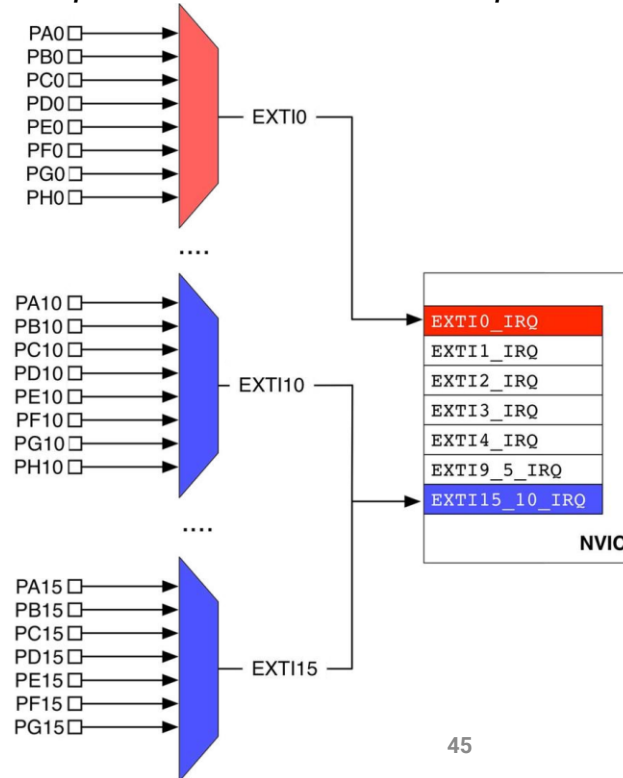
Interrupts can be configured and activated from Cube MX depending on peripheral configuration

✓ NVIC		✓ Code generation			
Enabled interrupt table	☐ Select for init ...	Generate Enabl...	☒ Generate IRQ handler	Call HAL h...	
Non maskable interrupt	☐	☐	☒	☐	
Hard fault interrupt	☐	☐	☒	☐	
Memory management fault	☐	☐	☒	☐	
Prefetch fault, memory access fault	☐	☐	☒	☐	
Undefined instruction or illegal state	☐	☐	☒	☐	
System service call via SWI instruction	☐	☐	☒	☐	
Debug monitor	☐	☐	☒	☐	
Pendable request for system service	☐	☐	☒	☐	
Time base: System tick timer	☐	☐	☒	☒	
EXTI Line13 interrupt	☐	☒	☒	☒	

Interrupt handling in STM32U585

All GPIO ports share same line interrupts ! Two GPIO interrupts for the same pin of different ports can not be enabled at the same time

Example : GPIOA Pin 1 interrupt and GPIOB Pin 1 interrupt can not be enabled at the same time.



Interrupt handling in STM32U585

GPIO interrupts are special GPIO mode in Cube MX

Configuration

Group By Peripherals

GPIO NVIC

Search Signals

Search (Ctrl+F)

☐ Show only Modified Pins

Pin Na...	Signal on ...	Pin Conte...	Pin Privile...	GPIO outp...	GPIO mode	GPIO Pull...	Maximum ...	Fast Mode	User Label	Modified
PC13	n/a	n/a	Non-privile...	n/a	External I...	No pull-up...	n/a	n/a		☐

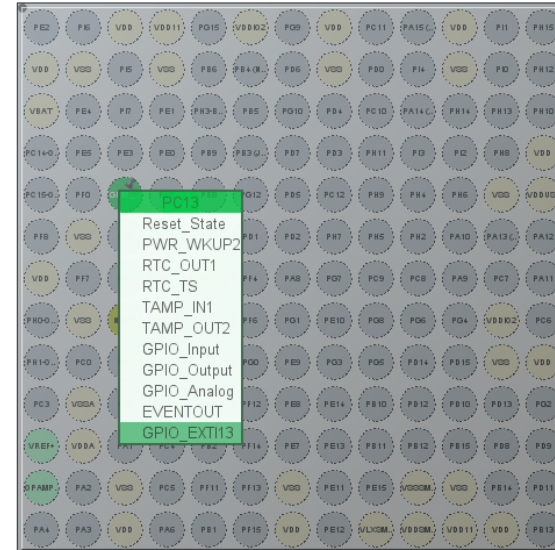
PC13 Configuration :

Pin Privilege access: Non-privileged access

GPIO mode: External Interrupt Mode with Rising edge trigger detection

GPIO Pull-up/Pull-down: No pull-up and no pull-down

User Label:



UFBGA169 (Top view)

Implementation of interrupts in STM32U585

Implement service routines in your code that can be called from generated interrupt handlers

- Create a custom function in *main.c*
- Declare the function prototype in *main.h*
- Call the function from the interrupt handler in *stm32u5xx_it.c*

main.h

```
/* USER CODE BEGIN EFP */  
void user_button_interrupt(void);  
/* USER CODE END EFP */
```

main.c

```
/* USER CODE BEGIN PFP */  
void user_button_interrupt(void)  
{  
    // Do something useful here !  
}  
/* USER CODE END PFP */
```

stm32u5xx_it.c

```
void EXTI13_IRQHandler(void)  
{  
    /* USER CODE BEGIN EXTI13_IRQn 0 */  
    /* USER CODE END EXTI13_IRQn 0 */  
    HAL_GPIO_EXTI_IRQHandler(USER_BUTTON_Pin);  
    /* USER CODE BEGIN EXTI13_IRQn 1 */  
    user_button_interrupt();  
    /* USER CODE END EXTI13_IRQn 1 */  
}
```

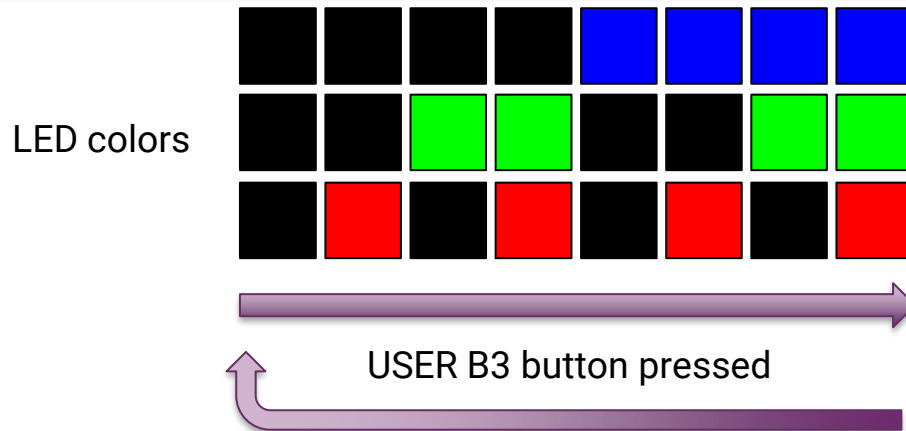
Practical #3

Objectives :

- Implement a program to control color of the red, green and blue LED using the switch button

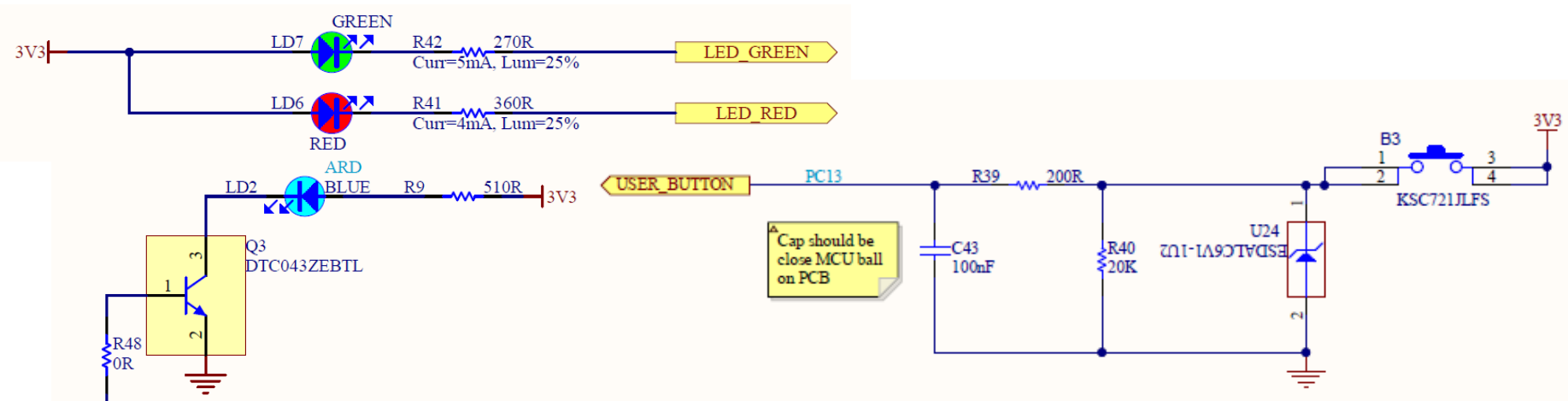
Constraints :

- Start with empty Cube MX (select STM32U585AI16Q)
- Use HAL library (no direct access to peripheral registers)
- Use GPIO input / output only
- **Use interrupts – the infinite while loop must be empty !**



Practical #3

I/O	Reference	Color	Name	Comment
PC13	B3	Blue	USER	User button
PH7	LD7	Green	LD7	User LED lights up when PH7 is set to 0.
PH6	LD6	Red	LD6	User LED lights up when PH6 is set to 0.
PE13	LD2	Blue	ARD	ARDUINO® LED lights up when PE13 is set to 1.

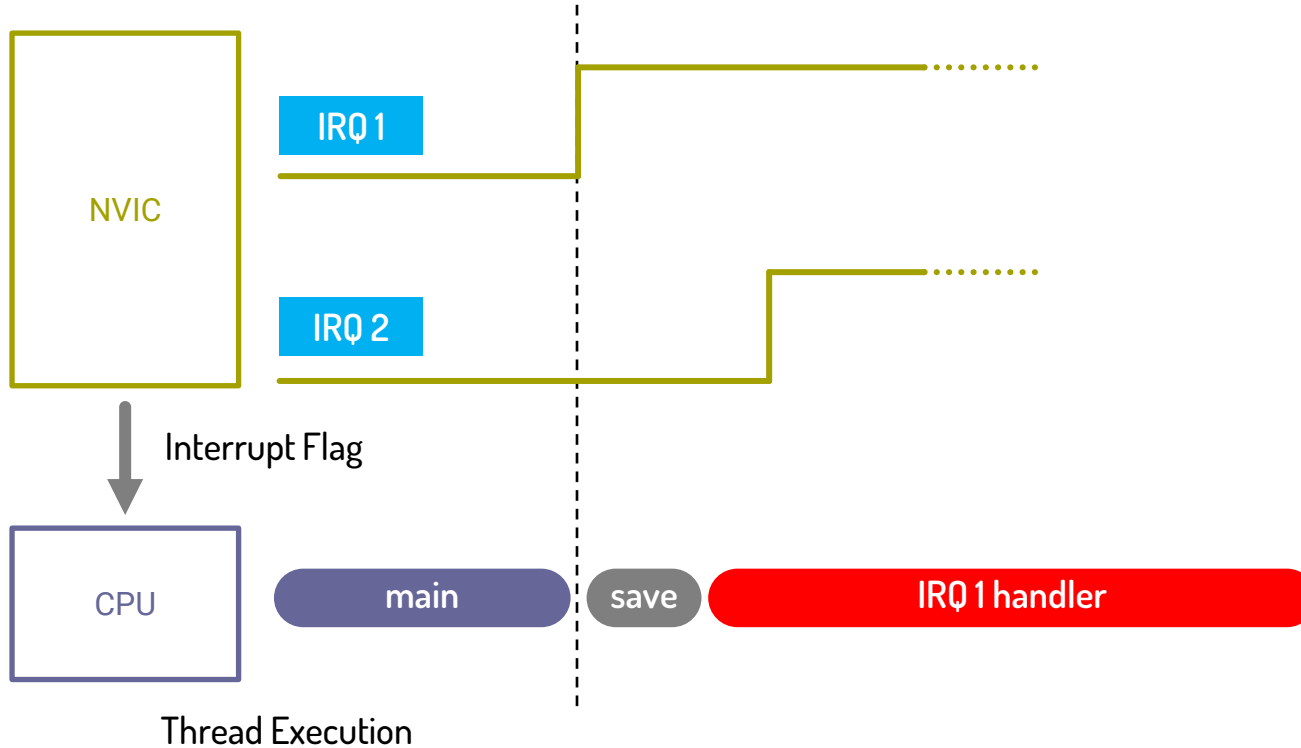


Interruptions - Part #2

Nested interrupts, priorities

Serving a single Interrupt Request (IRQ)

What happens if two interrupts occur closely / simultaneously ?



Serving a single Interrupt Request (IRQ)

In Cortex-M4 core, interrupts have priorities

A lower priority interrupt handler can be interrupted by a higher priority one

#	Source	Priority
-15	Reset	0
-14	Non Maskable Interrupt	0
-13	Hard Fault	0
-12	Memory Management	0 – 15
[...]	[...]	[...]
-1	SysTick	0 – 15
[...]	[...]	[...]
11	EXTI1 (GPIOx Pin 1)	0 – 15
12	EXTI2 (GPIOx Pin 2)	0 – 15
[...]	[...]	[...]



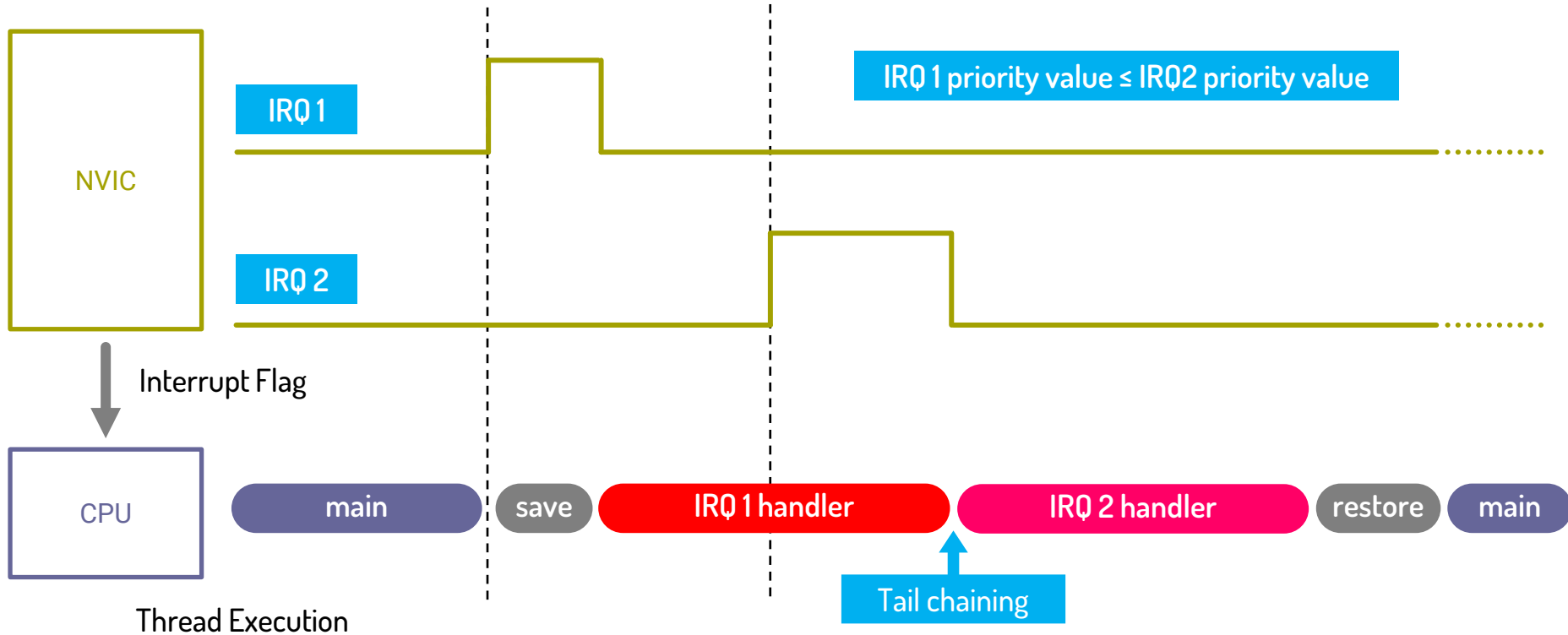
Priority value is comprised between 0 (highest priority) and 15 (lower priority)

User programmable priorities

The lower the number, the higher the priority

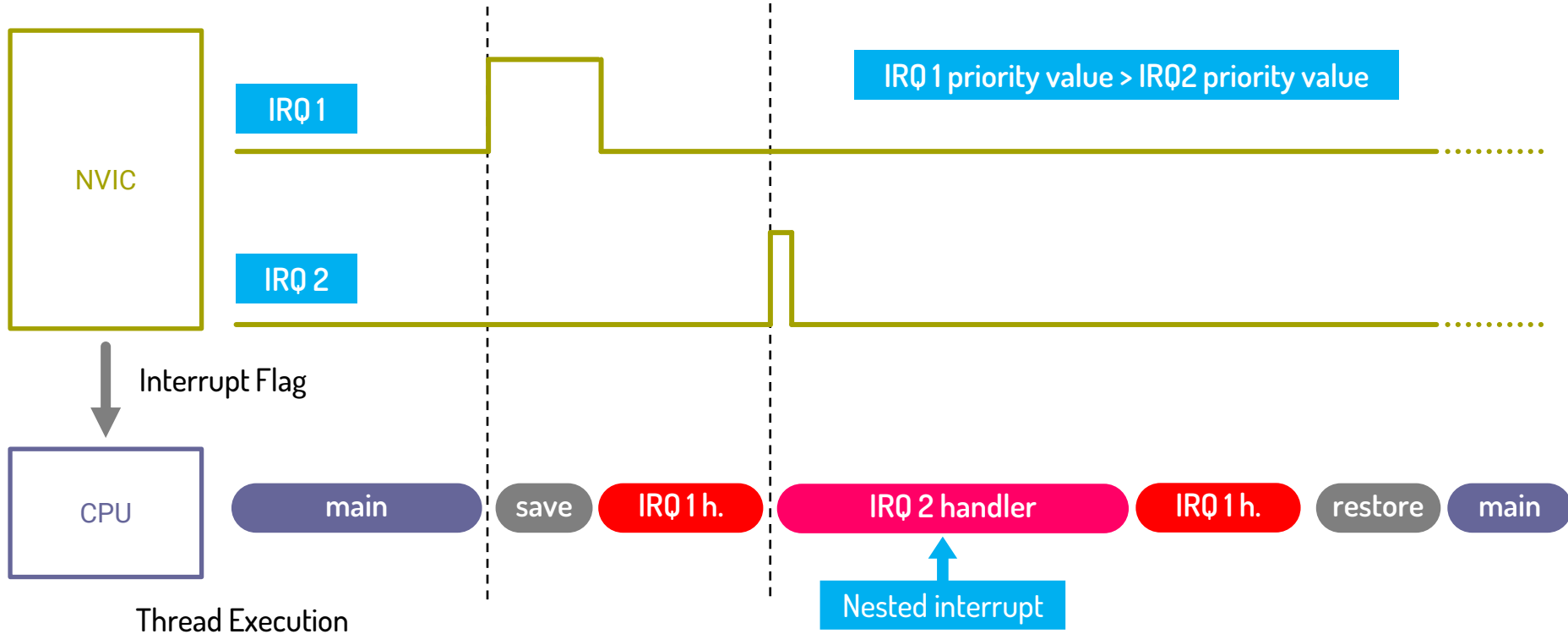
Serving a single Interrupt Request (IRQ)

What happens if two interrupts occur closely / simultaneously ?



Serving a single Interrupt Request (IRQ)

What happens if two interrupts occur closely / simultaneously ?



Configure interrupt priorities in Cube/X

GPDMA1
✓ GPIO
✓ ICACHE
IWDG
LPDMA1
✓ **NVIC**
RAMCFG
RCC
✓ SYS
⚠ TSC
WWDG

Analog >
Timers >
Connectivity >
Multimedia >
Security >
Computing >

Configuration

✓ NVIC ✓ Code generation

Priority Group 4 bits for pre-emption priority ... ☐ Sort by Preemption Priority and Sub Priority ☐ Sort by interrupts names

Search Show available interrupts ☒ Force DMA channels Interrupts

NVIC Interrupt Table	Enable	Preemption Priority	Sub Priority
Non maskable interrupt	☒	0	0
Hard fault interrupt	☒	0	0
Memory management fault	☒	0	0
Prefetch fault, memory access fault	☒	0	0
Undefined instruction or illegal state	☒	0	0
System service call via SWI instruction	☒	0	0
Debug monitor	☒	0	0
Pendable request for system service	☒	0	0
Time base: System tick timer	☒	15	0
Flash non-secure global interrupt	☐	0	0
RCC non-secure global interrupt	☐	0	0
EXTI Line13 interrupt	☒	0	0
FPU global interrupt	☐	0	0
Instruction cache global interrupt	☐	0	0

Time Management

SysTick, Basic and General Purpose Counters

A basic software time counter can be done using CPU loop

```
void Timer(uint32_t time)
{
    volatile uint32_t i;

    while (time > 0)
    {
        for (i = 100000; i > 0; i --);
        time --;
    }
}
```

Adjust value according to time unit

- The time counting is not precise (system clock frequency, compiler optimization)
- This function is blocking
- This is a terrible timer...

Hardware timers enable parallel counting and let the CPU free

Time management on microcontrollers

Timers

✓ LPTIM1

LPTIM2

LPTIM3

LPTIM4

RTC

TAMP

TIM1

TIM2

TIM3

TIM4

TIM5

TIM6

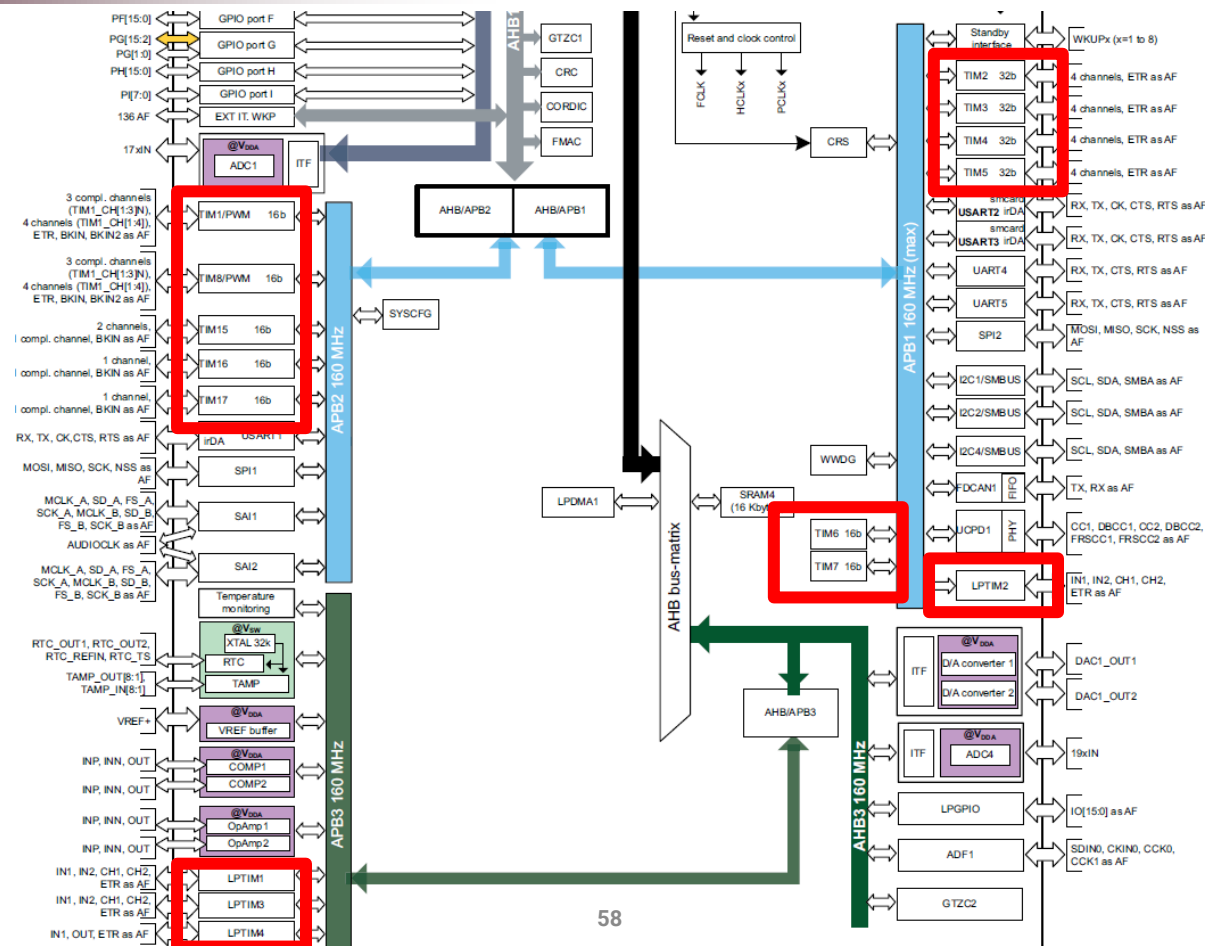
TIM7

TIM8

TIM15

TIM16

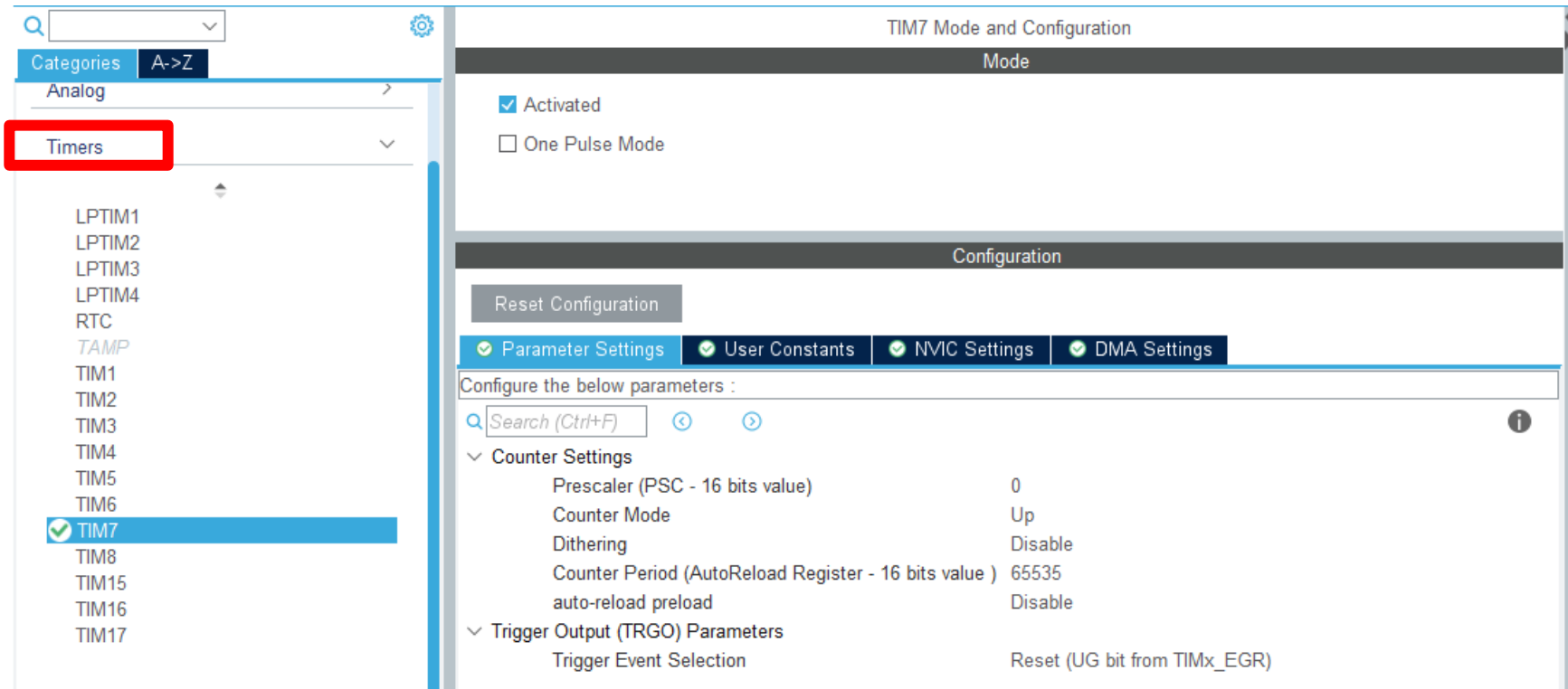
TIM17



Category of timers in STM32U585

Labels	Type	Comments
SysTick	24-bit (down) Simple	Always active, interrupt already enabled, 1 kHz default frequency
TIM1, TIM8	16-bit (up/down) Advanced Timer, Counter, PWM	Complex PWM generators
TIM2, TIM3, TIM4, TIM5, TIM15, TIM16, TIM17	32-bit (up/down) Generic Timer, Counter, PWM	Connected to input / output
TIM6, TIM7	16-bit (up) Timer only	No input / output Only to count time
LPTIM1, LPTIM2, LPTIM3, LPTIM4	16-bit (up) Timer, Counter, PWM	Active in low power modes

Timers of STM32U585 in CubeMX



The screenshot displays the STM32CubeMX configuration tool for the TIM7 timer. On the left, a sidebar lists various components, with 'Timers' highlighted in a red box and 'TIM7' selected with a green checkmark. The main panel is titled 'TIM7 Mode and Configuration' and is divided into two sections: 'Mode' and 'Configuration'.

Mode Section:

- ☒ Activated
- ☐ One Pulse Mode

Configuration Section:

A 'Reset Configuration' button is present. Below it, four tabs are visible: 'Parameter Settings' (active), 'User Constants', 'NVIC Settings', and 'DMA Settings'.

The 'Parameter Settings' tab contains the following configuration parameters:

Configure the below parameters :

Search (Ctrl+F) [Navigation arrows] [Info icon]

- Counter Settings**
 - Prescaler (PSC - 16 bits value): 0
 - Counter Mode: Up
 - Dithering: Disable
 - Counter Period (AutoReload Register - 16 bits value): 65535
 - auto-reload preload: Disable
- Trigger Output (TRGO) Parameters**
 - Trigger Event Selection: Reset (UG bit from TIMx_EGR)

SysTick Timer

Time management on microcontrollers

SysTick timer is simple, convenient timer to use for time counting

- Already configured and started by the system
- Interrupts is activated and used to increment a counter
- Default period is 1 millisecond

stm32u5xx_it.c

```
void SysTick_Handler(void)
{
    HAL_IncTick();
}
```

Increment an internal variable

```
uint32_t delay_ms = 10;
HAL_Delay(delay_ms);
```

Wait for a number of tick

```
uint32_t ticks_num;
ticks_num = HAL_GetTick();
```

Get number of ticks since beginning



Warning: Pay attention to SysTick interrupt priority

- Increment counter used by HAL_Delay()
- If HAL_Delay() is used in an interrupt, code execution will be block if SysTick interrupt priority value is lower or equal to the interrupt in which it is called !
- Ex. Button debouncer:

NVIC Interrupt Table	Enabled	Preemption Priority
Non maskable interrupt	☒	0
Hard fault interrupt	☒	0
Memory management fault	☒	0
Prefetch fault, memory access fault	☒	0
Undefined instruction or illegal state	☒	0
System service call via SWI instruction	☒	0
Debug monitor	☒	0
Pendable request for system service	☒	0
Time base: System tick timer	☒	15
Flash non-secure global interrupt	☐	0
RCC non-secure global interrupt	☐	0
EXTI Line13 interrupt	☒	0
FPU global interrupt	☐	0
Instruction cache global interrupt	☐	0

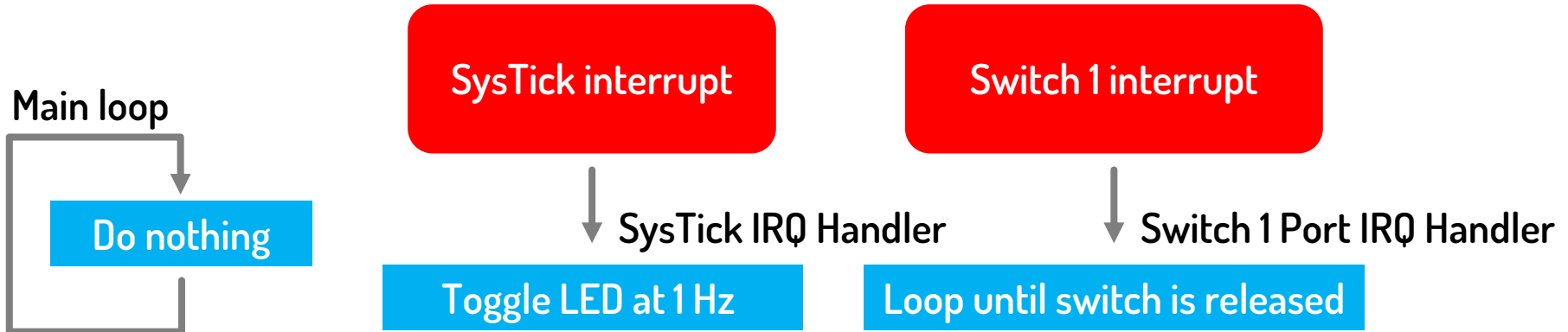
```
void user_button_interrupt(void)
{
    /* Anti rebond */
    HAL_Delay(10);
}
```

This will hang program execution

Practical #4

Objectives :

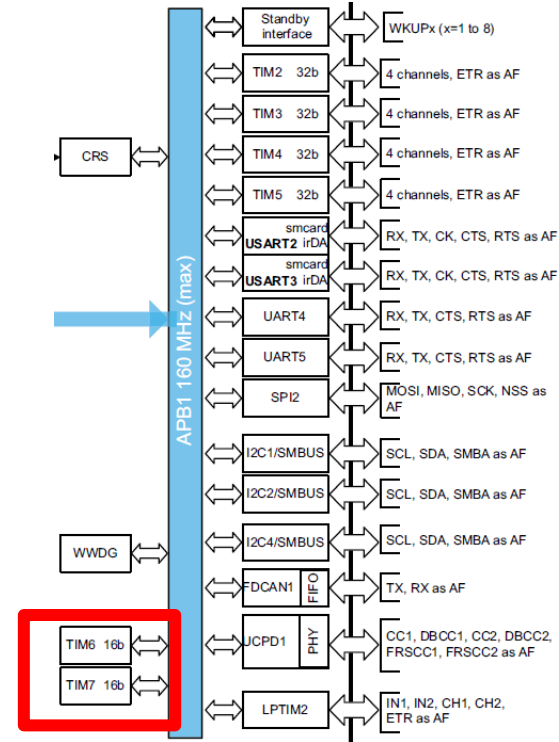
- Implement a program to blink the red led using SysTick timer interruption at 1 Hz.
- Implement a blocking loop in a Switch interruption (code stays in the interrupt while button is pushed)
- Use priorities to **force** or **prevent** the Switch interrupt to block SysTick timer interrupts



TIM6, TIM7 simple timers

TIM6 and TIM7 timers are simple 16-bit up timers with no input / output connectivity

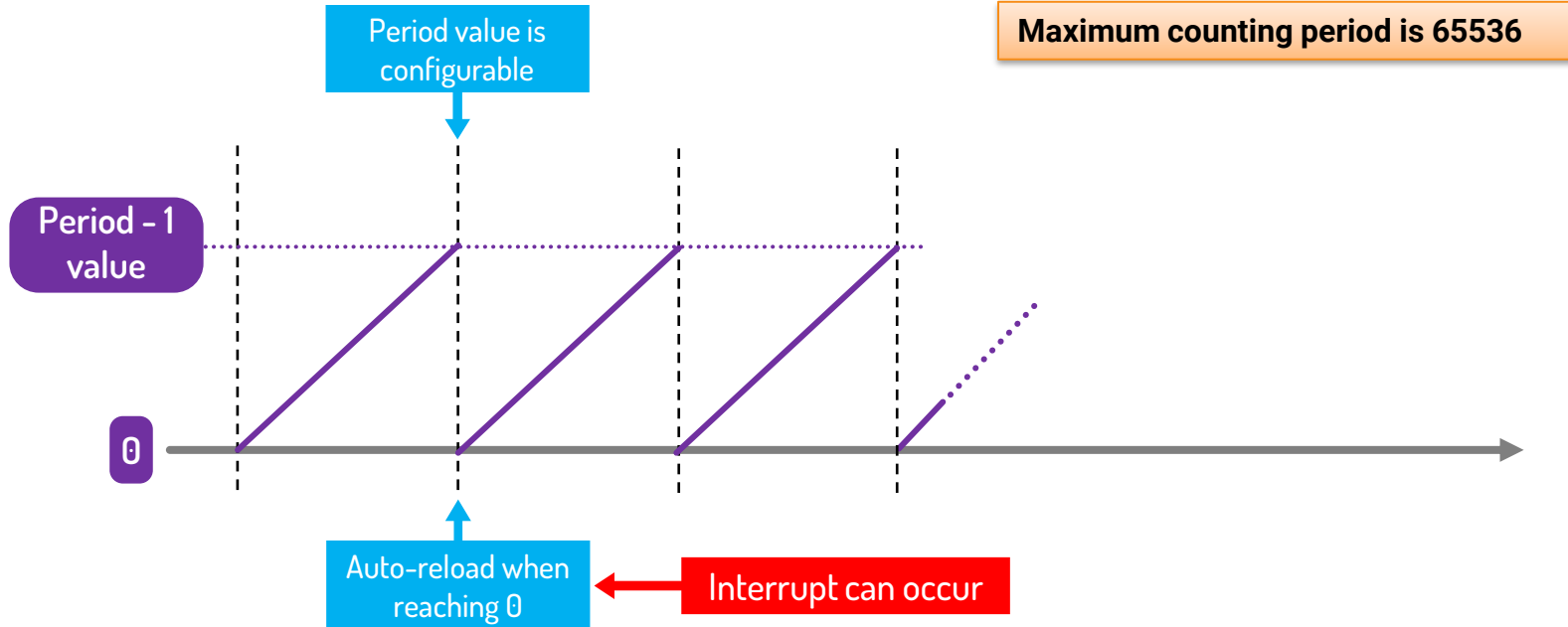
- Their practical usage is reserved for internal clock counting
- They can trigger interrupts when reloading
- They are clocked with APB1 Timer Clock



Time management on microcontrollers

TIM6 and TIM7 contains one 16-bit counter, counting up in 2 possible configurations

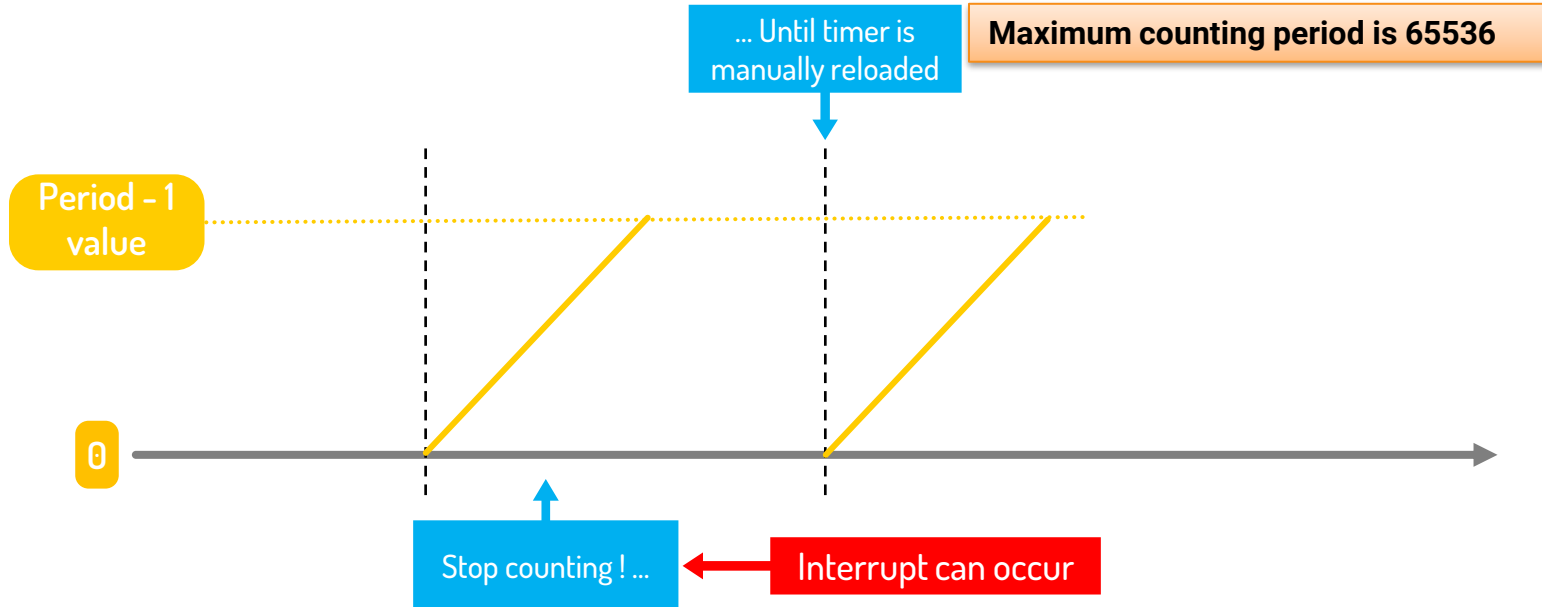
Periodic mode (*One Pulse Mode* disabled)



Time management on microcontrollers

TIM6 and TIM7 contains one 16-bit counter, counting up in 2 possible configurations

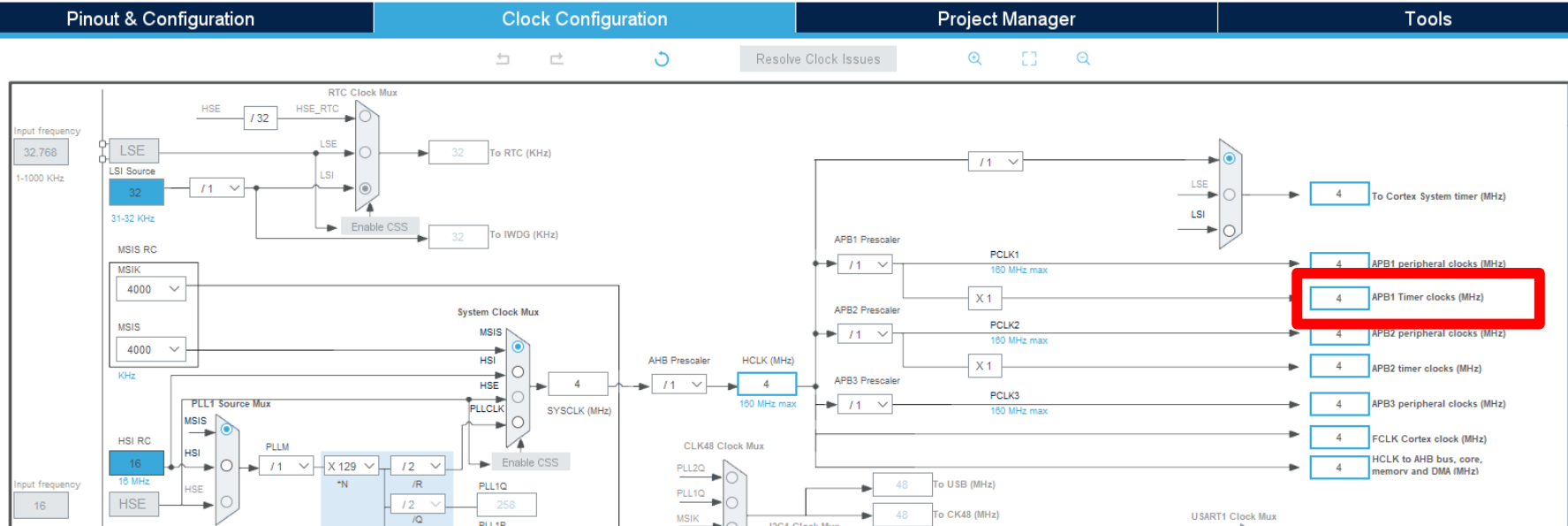
One-shot mode (*One Pulse Mode* enabled)



Time management on microcontrollers

TIM6, TIM7 clock speed

Timer counts at APB1 Timer clock speed, default is 4 MHz



Time management on microcontrollers

TIM6, TIM7 clock speed

Timer has an internal pre-scaler to reduce counting frequency

TIM6 Mode and Configuration

Mode

☒ Activated

☐ One Pulse Mode

Configuration

Reset Configuration

☒ Parameter Settings ☒ User Constants ☒ NVIC Settings ☒ DMA Settings

Configure the below parameters :

Search (Ctrl+F) ◀ ▶ ⓘ

Counter Settings

Prescaler (PSC - 16 bits value)	39999	Counting frequency is $4 \text{ MHz} / (39999 + 1) = 100 \text{ Hz}$
Counter Mode	Up	
Dithering	Disable	
Counter Period (AutoReload Register - 16 bits value)	49	Interrupt period is $100 \text{ Hz} / (49 + 1) = 2 \text{ Hz}$
auto-reload preload	Disable	

Trigger Output (TRGO) Parameters

Trigger Event Selection

Reset (UG bit from TIMx_EGR)

Usage of TIM6 and TIM7 with interrupts

- Ensure interrupts are enabled in NVIC settings
- Implement a interrupt service routine and declare it in `main.h`
- Call your ISR from the Timer handler function in `stm32u5xx_it.c`
- Start the timer with `HAL_TIM_Base_Start_IT()` function

Practical #4b

Objectives :

- Blink the green LED at 2 Hz using TIM6 (250 ms ON – 250 ms OFF)
- Blinking **must not be blocked** by user button
- Red LED blinking at 1 Hz using SysTick **has to be blocked** by user button

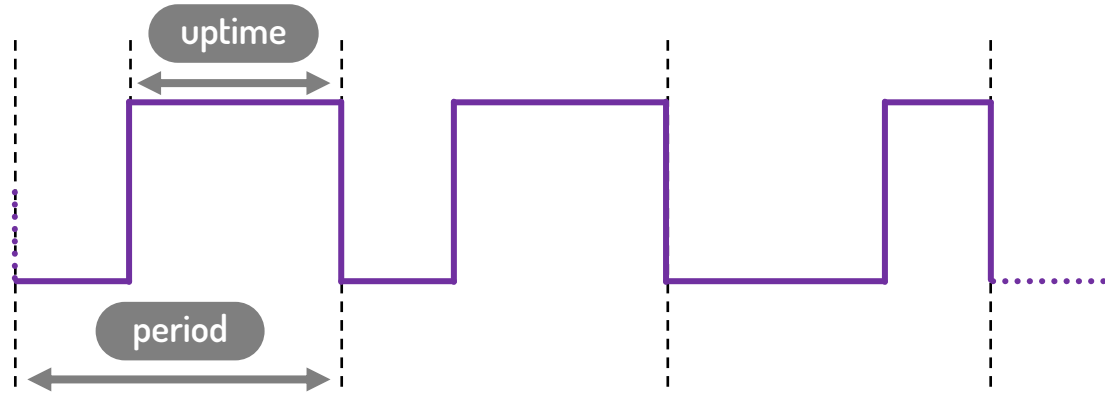
Constraints :

- Start from **Practical #4** project
- Main “while” loop is empty

TIMx generic timers (other than TIM6 & TIM7)

Reminder on Pulse Width Modulation signals (PWM)

- **PWM signals are convenient for driving analog/digital peripherals**
 - LED intensity
 - Motor speed, drone flight controls
 - Sound generation
- **period** is constant
- **frequency** is $1 / (\text{PWM period})$
- **uptime** is variable
- **duty cycle** is $\text{uptime} / \text{period}$

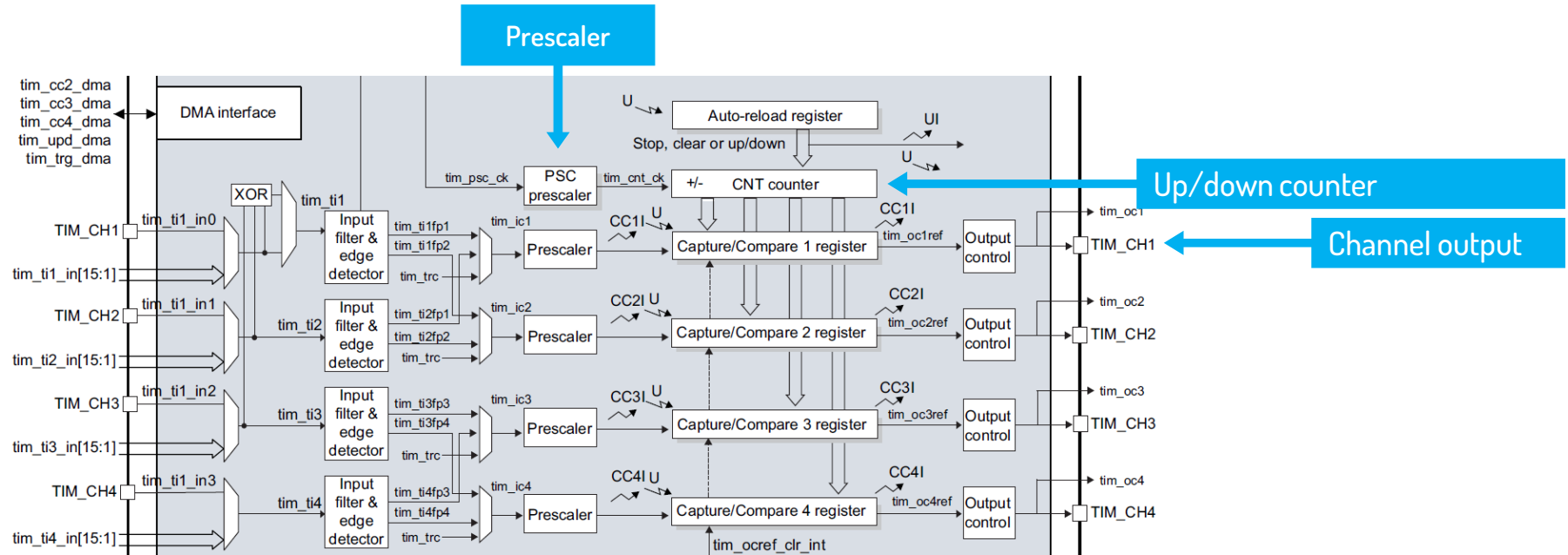


Average value of PWM is an analog signal proportional to duty cycle

Capture / Compare counters like TIM1, etc. can be used to generate PWM

Time management on microcontrollers

TIMx are 16-bit or 32-bit timers/counters



An interrupt can be generated when counter loops back to 0

An interrupt can be generated when counter reach channel CC register

Each TIMx possesses capture/compare registers (CCy)

Capture mode: made to measure time or events

- The current value of the counter is stored in the CC register on rising / falling / toggle of external input TIMx_CHy

Compare mode: made to generate PWM

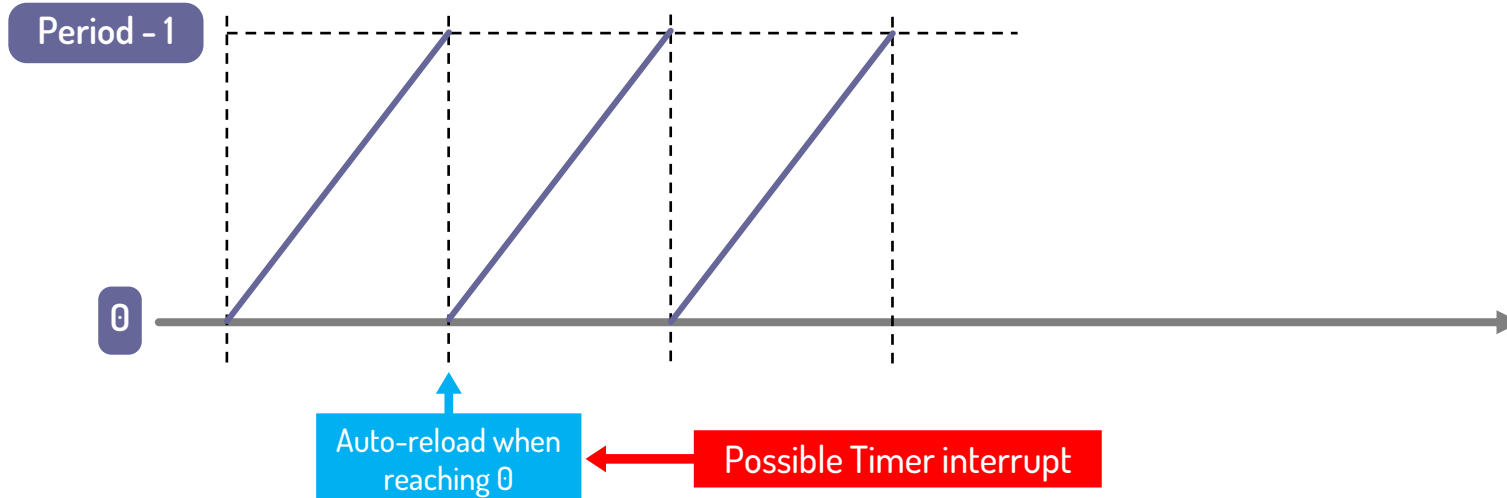
- The outputs TIMx_CHy are set/reset when counter reaches the corresponding CCy register

Details on the PWM Mode

This mode is dedicated to generate PWM signals using timer period and CC register.

PWM period is set by the timer period (*Counter period* setting in Cube MX)

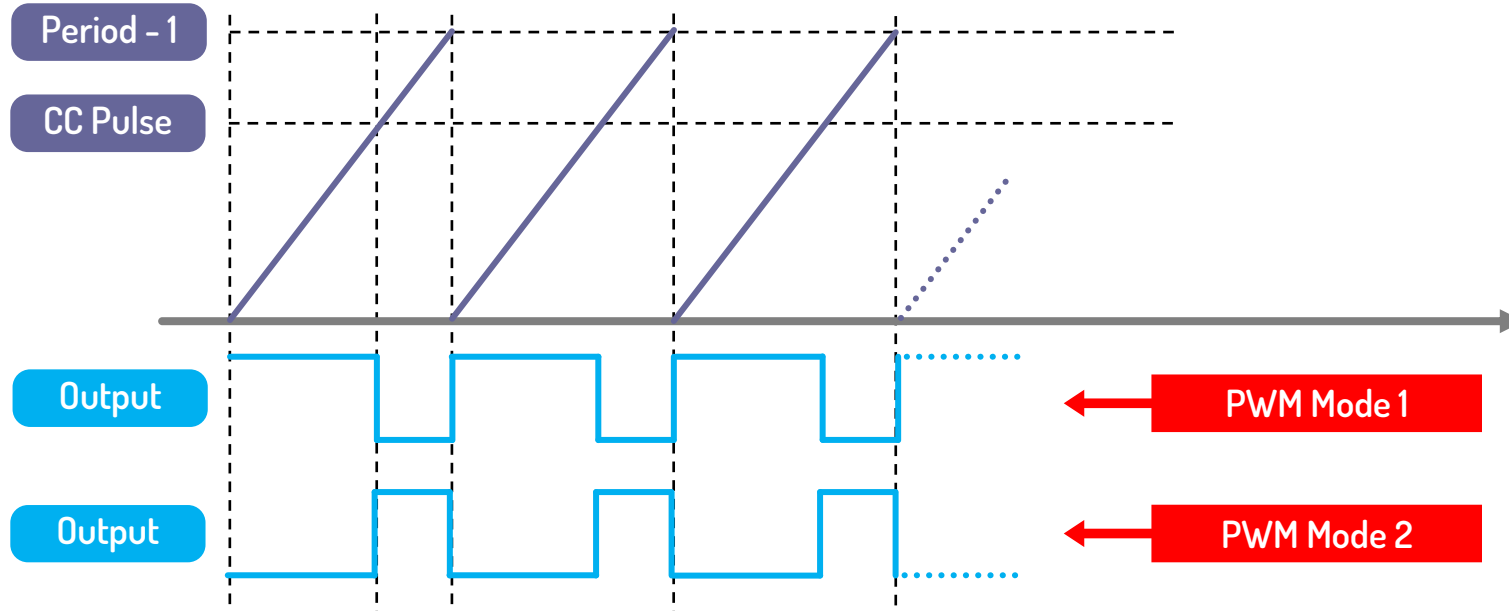
Counter of the timer is incremented on each clock source rising edge until reaching the value stores in the period register, where the counter restart counting from 0.



Time management on microcontrollers

PWM duty cycle is set by the CC register value (*Pulse setting in Cube MX*)

An output is generated based on the comparison between current counter value and a value stored in CC register. **Pulse** and **Period** is the **ducty cycle**.



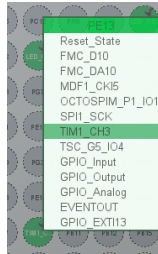
Time management on microcontrollers

Configuration of Timer for PWM generation in Cube MX

Select GPIO to be used for PWM output

Select the right Timer

Inputs and outputs for each Timer are fixed !



TIM1 Mode and Configuration

Mode

Slave Mode	Disable
Trigger Source	Disable
Clock Source	Disable
Channel1	Disable
Channel2	Disable
Channel3	PWM Generation CH3

Configure the channel corresponding to GPIO

Adjust clock and period settings

Parameter Settings | User Constants | NVIC Settings | DMA Settings | GPIO Settings

Configure the below parameters :

Search (Ctrl+F)

Counter Settings

Prescaler (PSC - 16 bits value)	39999
Counter Mode	Up
Dithering	Disable
Counter Period (AutoReload Register - 16 bits v...	49

Set Mode and initial Pulse (duty cycle)

▼ PWM Generation Channel 3

Mode

Pulse (16 bits value)

Output compare preload

Fast Mode

CH Polarity

CH Idle State

PWM mode 1

0

Enable

Disable

High

Reset

Objectives :

- Find the correct function to call to start timer in PWM Mode
- Find how to modify duty cycle (look at the code generated by CubeMX)

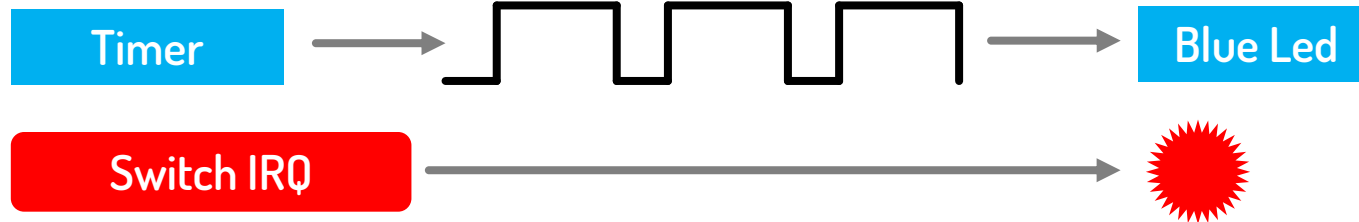
Practical #5b

Objectives :

- Control Blue LED intensity using the user switch button

Constraints :

- Implement a PWM using a Timer to control Blue Led intensity
- Use the user button switch to increment the duty cycle by steps of 10%



Practical #5

Hints :

- Think about what should be the PWM clock and period ?
 - A period of 100 enables to set duty cycle directly as a percentage (0: always off – 100: always on)
 - With a period of 100, clock should be at least $100 * 1 \text{ kHz}$ so a human eye can't see off/on state but an average of the intensity

Advanced:

- Change switch behavior so intensity varies if the switch is kept pushed

Interruptions – Part #3

Deferred interrupts, global event architecture

Generally, on MCUs it is not good practice to :

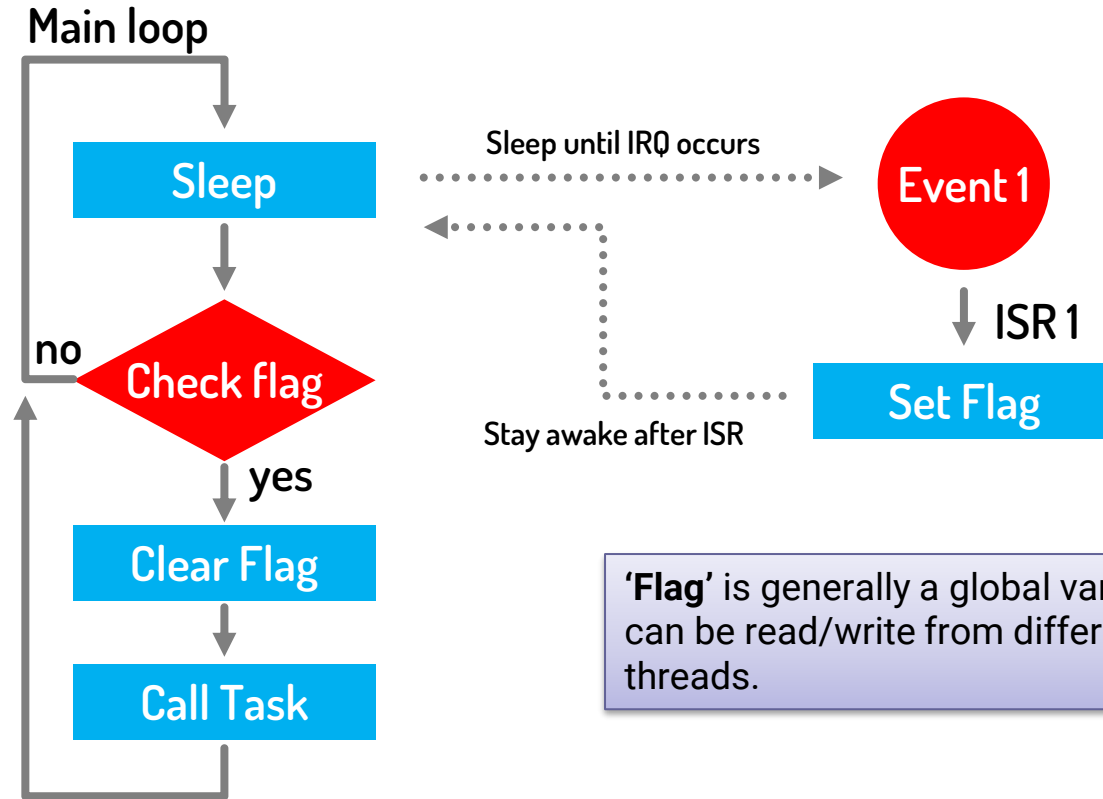
- Execute long code procedures within the ISR
- Calling external function from the ISR

For safe code execution, it is always better to run code from the main thread rather than from an ISR. When such code is triggered by an interrupt, a **flag** can be used to synchronize the interrupt event (from the ISR) with the execution of code in the main thread.

Use it for code which must be triggered from an interrupt and:

- Take very long processing time
- Should be interruptible by any IRQ (lowest priority)
- Must run in the main thread (safe execution)

Deferred interrupts



'Flag' is generally a global variable in the program so it can be read/write from different function and execution threads.

Deferred interrupts

Code template

```
/* This variable is declared out of a function, so it is global to the file */
uint8_t flag;

int main(void)
{
    /* Initialize the flag */
    flag = 0;

    while(1)
    {
        /* Stop the CPU until next interrupt */
        HAL_SuspendTick();
        HAL_PWR_EnterSLEEPMode(PWR_MAINREGULATOR_ON,PWR_SLEEPENTRY_WFI);
        HAL_ResumeTick();

        /* After an interrupt occurred, the CPU stay awake */
        /* So we check the flag */
        if (flag != 0)
        {
            /* Do what you want to do */
            call_some_task();
            /* Clear the flag */
            flag = 0;
        }
    }
}
```

```
void myIRQ_Handler(void)
{
    /* Set the flag */
    flag = 1;
}
```

Summary of event-based programming

Event handling type	High priority interrupt	Low priority interrupt	Deferred interrupt	Event polling
Priority				
Type	• Short code • Lowest latency • Not interruptible	• Mostly short code • Low latency • Interruptible	• Longer code • No latency constraint • Interruptible	Do not use event polling, please.

UART communication

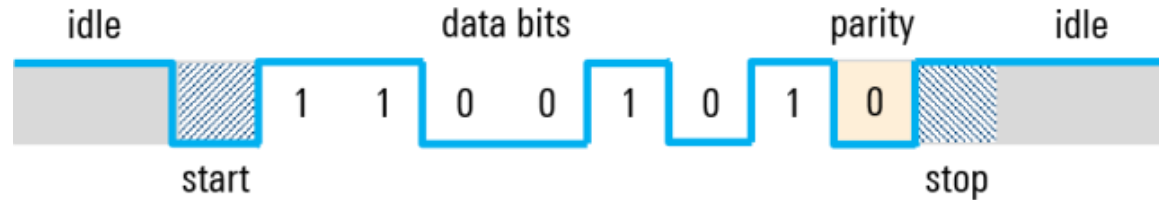
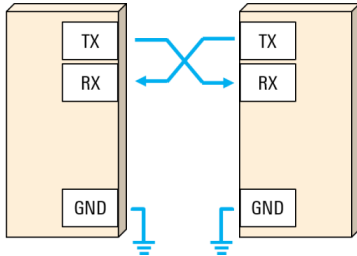
Communication with remote computer

Communication with remote computer

UART Protocol :

- Asynchronous transmission (no clock)
- Usual baud rates : 9600, 57600, 115200 bdps
- Full duplex (independent data line for emitter and receiver)

Emitter and receiver must be set to same speed



Communication with remote computer

UART1 is connected to ST Link debug probe

- **On-board** ST Link v3 is used both for programming / debugging
- It also has 2 GPIOs dedicated to UART communication between STM32U585 and computer

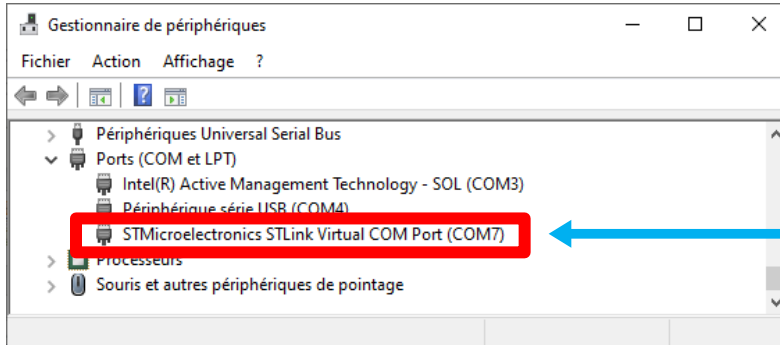
This UART communication over USB port is called **Virtual Serial Port (COM Port on Windows, TTY/ACM on Linux / Mac OS)**

U13A			
MCU_IOs (A,B,C)			
STMOD+.1 UART2 CTS K3	PA0	PC0	J2 ARD ADC.A0
STMOD+.1 UART2 RTS L3	PA1	PC1	J3 ARD.D8 IO
STMOD+.1 UART2 TX M2	PA2	PC2	J4 ARD ADC.A1
STMOD+.1 UART2 RX N2	PA3	PC3	K1 OCTOSPLR_IO6
STMOD+.1 SPI1 NSS N1	PA4	PC4	L4 ARD ADC.A2
STMOD+.1 ADC K4	PA5	PC5	M4 ARD ADC.A3
CAMP.PIXCLK N4	PA6	PC6	H13 CAMD0
ARD.ADC.A4 J5	PA7	PC7	G12 CAMD1
ARD.D8.TM C7	PA8	PC8	G10 CAMD2
T.VCP.TX G11	PA9	PC9	G9 STMOD+.2 TIM
T.VCP.RX F11	PA10	PC10	C9 WRLS.UART4 TX
			A9 WRLS.UART4 RX

- UART1 TX pin is connected to **PA9**
- UART1 RX pin is connected to **PA10**

Communication with remote computer

Check your peripheral manager (*gestionnaire de périphériques*)



Virtual COM Port created by the STLink v3

You can connect to this serial port by using any terminal such as:

- Putty <https://www.putty.org/>
- MobaXterm <https://mobaxterm.mobatek.net/>
- Tabby <https://tabby.sh/> (cross-platform)
- Or go to <https://bipes.net.br/aroca/web-serial-terminal/> using Chrome or Edge



Data sent from STM32 will be “readable” on a serial monitor ONLY if it is text

Communication with remote computer

The screenshot shows the STM32CubeMX Pinout & Configuration window. The left sidebar lists various components, with USART1 selected. The main area shows the USART1 Mode and Configuration settings. The Mode is set to Asynchronous, and Hardware Flow Control (RS232) is disabled. The Configuration section shows the Parameter Settings tab selected, with the following parameters:

Parameter	Value
Baud Rate	115200 Bits/s
Word Length	8 Bits (including Parity)
Parity	None
Stop Bits	1
Data Direction	Receive and Transmit
Over Sampling	16 Samples
Single Sample	Disable

Configure PA9, PA10 and USART1 in Cube MX

USART1 is then initialized in main.c :

```
UART_HandleTypeDef huart1;  
static void MX_USART1_UART_Init(void);
```

Communication with remote computer

Send bytes arrays using transmit function :

```
HAL_UART_Transmit(huart, pData, Size, Timeout);
```

&huart1

A byte array previously defined
`uint8_t my_data[32];`

Number of bytes to transmit
from the array

0xFFFF

Send bytes arrays using transmit function :

```
// be sure that your array is large enough !
char byte_array[32];
int str_size = 0;
unsigned int value = 0;
while (1)
{
    // put some text in the byte array
    str_size = snprintf(byte_array, sizeof(byte_array), "Hello World ! %u\r\n", value);
    HAL_UART_Transmit(&huart1, (uint8_t *)byte_array, str_size, 0xFFFF);
    value = value + 1;
    HAL_Delay(1000);
}
```

snprintf() is included with `#include <stdio.h>`

Communication with remote computer

snprintf() function specifiers to convert numbers into characters

```
%d    // signed 8 or 16 bits integer
%d    // signed 32 bits integer
%lld  // signed 64 bits integer
%u    // unsigned 8 or 16 bits integer
%lu   // unsigned 32 bits integer
%llu  // unsigned 64 bits integer
%+[0][7][.3]f // float (must be enabled project properties, see above)
The value 1,1352 would be printed +01.135 (7 characters total, 3 decimals)
%s    // character string
%%    // write % character
\r\n  // new line
```

IDE Properties for TD6

type filter text

- > Resource
- Builders
- ▼ C/C++ Build
 - Build Variables
 - Environment
 - Logging
 - Settings
- > C/C++ General
- CMSIS-SVD Settings
- Project References
- Run/Debug Settings

Settings

Configuration: Debug [Active]

Manage Configurations...

Tool Settings Build Steps Build Artifact Binary Parsers Error Parsers

MCU Toolchain	MCU	STM32U585AllxQ	Select
MCU Settings	Board	genericBoard	
MCU Post build outputs	Floating-point unit	FPv5-SP-D16	
▼ MCU GCC Assembler	Floating-point ABI	Hardware implementation (-mfloat-abi=hard)	
General	Instruction set	Thumb2	
Debugging	Runtime library	Reduced C (--specs=nano.specs)	
Preprocessor	☒ Use float with printf from newlib-nano (-u _printf_float)		
Include paths	☐ Use float with scanf from newlib-nano (-u _scanf_float)		
Miscellaneous			
▼ MCU GCC Compiler			
General			
Debugging			

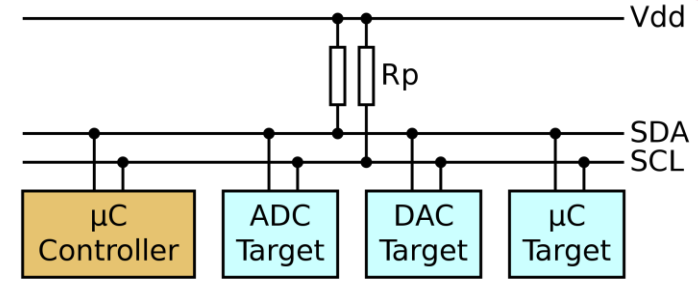
I2C communication

Communication with board peripherals

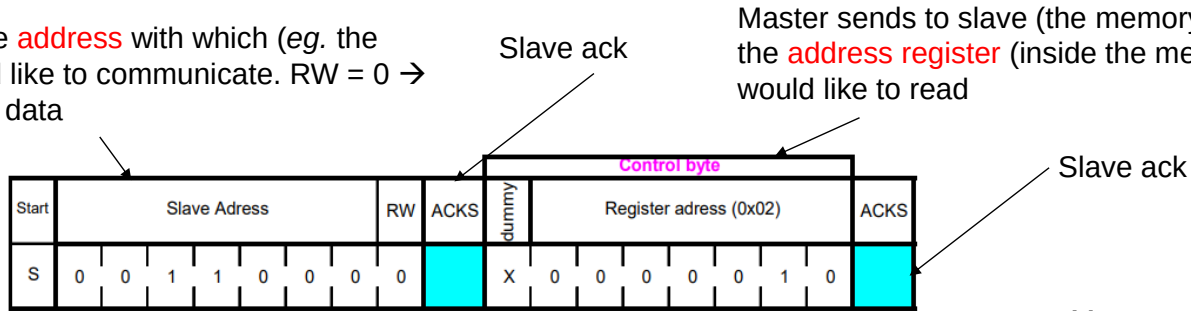
Communication with board peripherals

I2C Protocol :

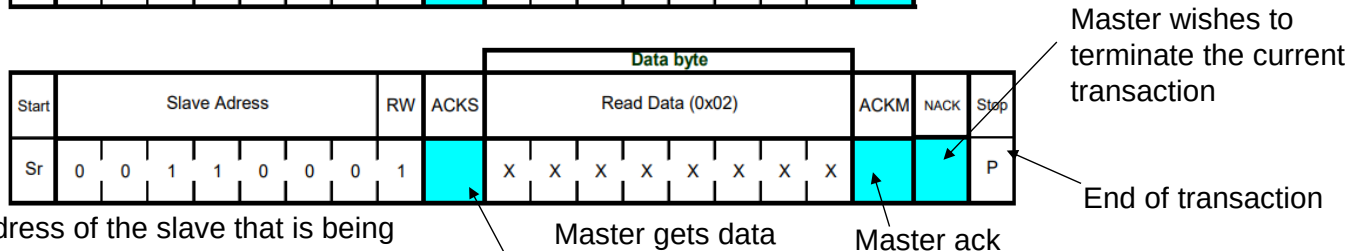
- Synchronous transmission
- Usual clocks: 100 kHz (Standard Mode), 400 kHz (Fast Mode)
- Half duplex (only one data line for emitter and receiver)



Master sends the slave **address** with which (eg. the memory bank) it would like to communicate. $RW = 0 \rightarrow$ master wishes to write data



Master sends to slave (the memory bank here) the **address register** (inside the memory!) it would like to read

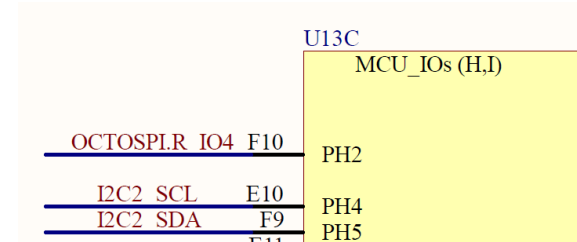


Master sends the address of the slave that is being ready now to communicate $\rightarrow$ master wishes to get data ($RW = 1$) from the register which address is declared above

Communication with board peripherals

I2C2 is used to communicate MEMS on board

- SCL is connected to **PH4**
- SDA is connected to **PH5**



HTS221

Humidity & Temperature
8-bit addr. 0xBE

IIS2MDCTR

3-axis Magnetometer
8-bit addr. 0x3C

LPS22HH

Pressure & Temperature
8-bit addr. 0xBA

ISM330DHCX

6-axis Inertial Unit
8-bit addr. 0xD6

Communication with board peripherals

Pinout & Configuration

Clock Configuration

Software Packs

I2C2 Mode and Configuration

Mode

I2C I2C

Configuration

Reset Configuration

DMA Settings

GPIO Settings

User Constants

NVIC Settings

Parameter Settings

Configure the below parameters :

Search (Ctrl+F)

Timing configuration

I2C Speed Mode Standard Mode

I2C Speed Frequency 100

Rise Time (ns) 0

Fall Time (ns) 0

Coefficient of Digital Filter 0

Analog Filter Enabled

Timing 0x00000E14

Slave Features

Clock No Stretch... Disabled

General Call Address... Disabled

Primary Address ... 7-bit

Dual Address Acknowledge... Disabled

Primary slave address... 0

Autonomous Mode

Autonomous Mode Disable

Configure PH4, PH5 and I2C2 in Cube MX

I2C2 is then initialized in main.c :

```
I2C_HandleTypeDef hi2c2;  
static void MX_I2C2_Init(void);
```

```
// To read 2 bytes from peripheral's memory register 0x3D
uint8_t value[2];
uint8_t reg = 0x3D;
uint8_t addr = 0xBE;
HAL_I2C_Mem_Read(&hi2c2, addr, reg, 1, value, 2, 0xFFFF);
```

Objectives :

- Read ID register of the 4 available sensors and print them in a serial monitor
- **Check that IDs correspond to value provided in datasheet !**