INSA INSTITUT NATIONAL DES SCIENCES APPLIQUÉES LYON

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

Bertrand Massot bertrand.massot@insa-lyon.fr

membre de UNIVERSITÉ DE LYON

Foreword

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

#	Session title	Date / Time
1	Introduction	12/09
2	Input / Output	02/10
3	Interrupt #1	07/10
4	Interrupt #2 – Time Management	08/10
5	Time Management (2)	06/11

#	Session title	Date / Time
6	Interrupt #3 – Peripherals	02/12
7	**Exam (50 mn)**, Project setup	10/12
8	Project #1	11/12
9	Project #2	15/01
10	Project #3, **Exam (50 mn)**	19/01

Use A.I. wisely – from ChatGPT itself

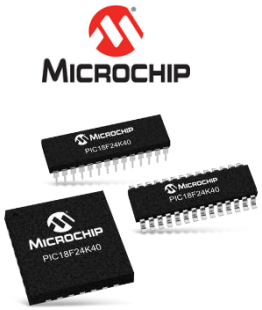
Why avoid AI in learning embedded programming?

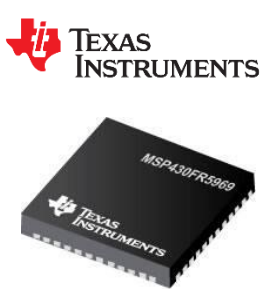
- Essential foundations: risk of copying without understanding (hardware, registers, interrupts).
- Autonomy: learning debugging and problem analysis.
- Real-world constraints: AI may ignore memory limits, real-time, power consumption.
- Reliability: generated code can look correct but be wrong or incomplete.
- 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 are small integrated computers (Core + Peripherals)

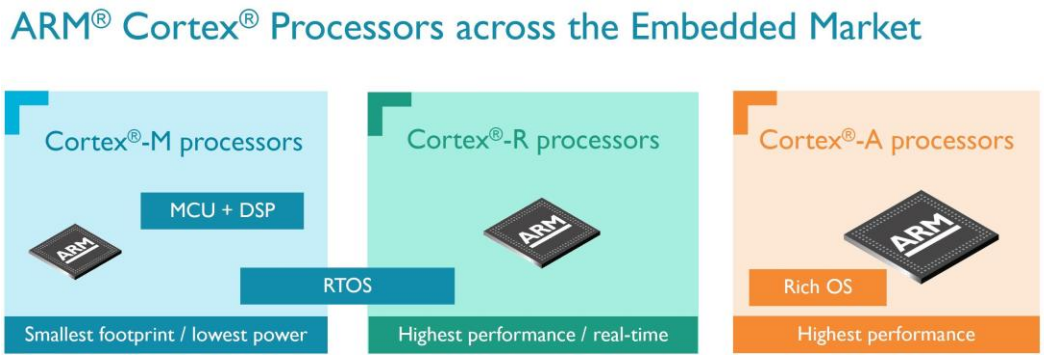




Key figures :

- > $15 billion market in 2020
- Automotive, Industrial, Consumer Electronics, Healthcare, Aerospace & Defense
- > 80% programmed in C language
- > 46 000 references on Mouser / Digikey

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



arm

arm CORTEX®-M0+

Ultra-low power

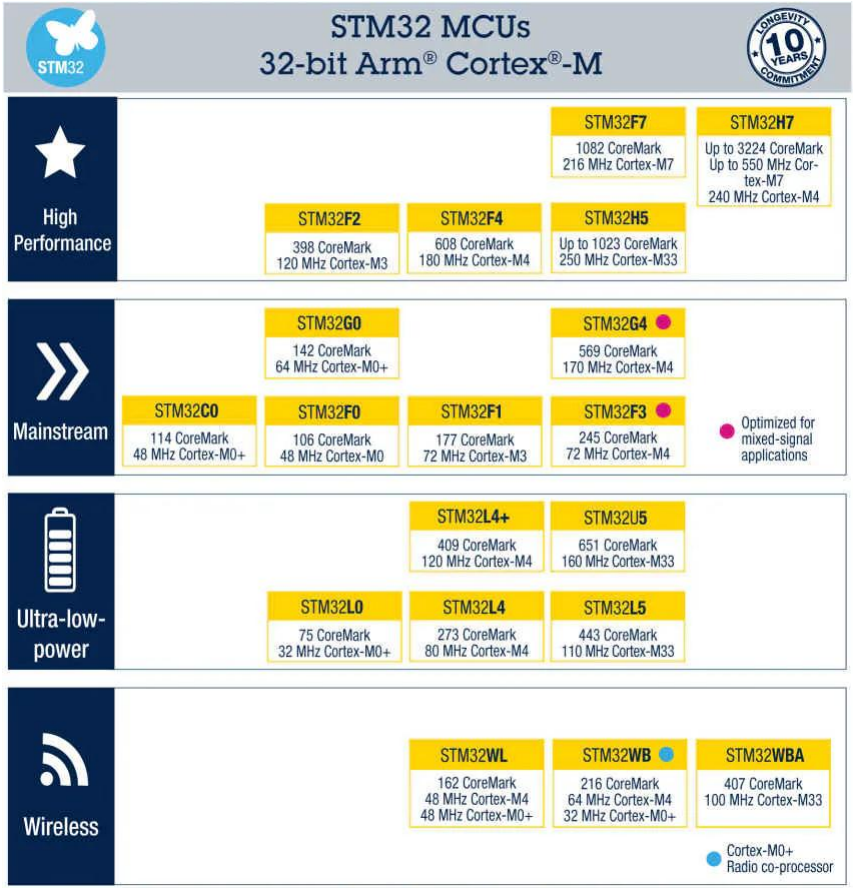
arm CORTEX®-M4

Most deployed, FPU

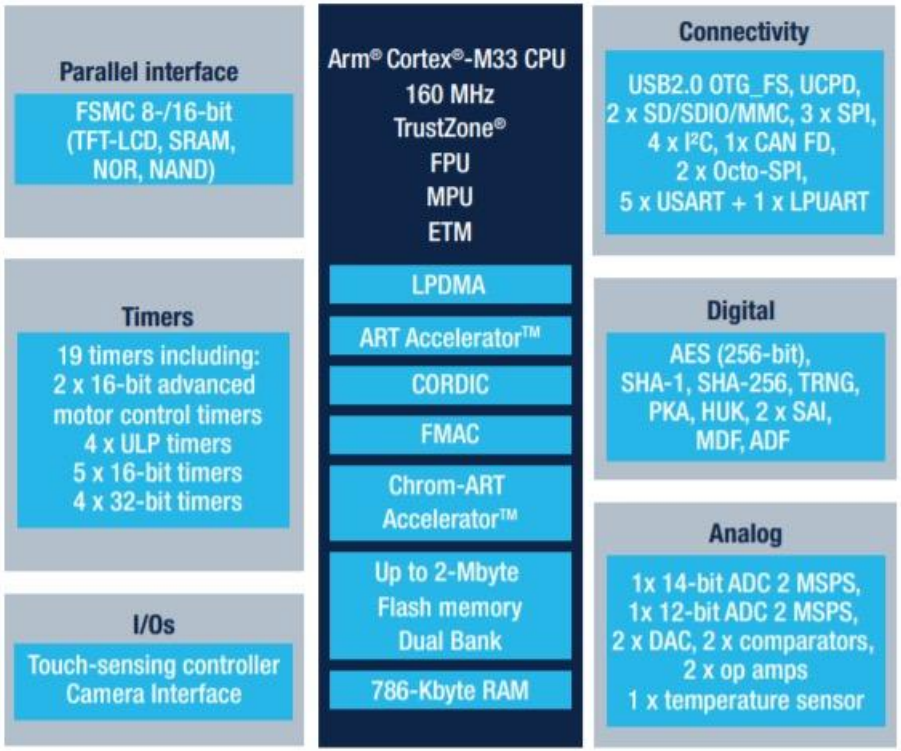
arm CORTEX-M33

M4 + Security

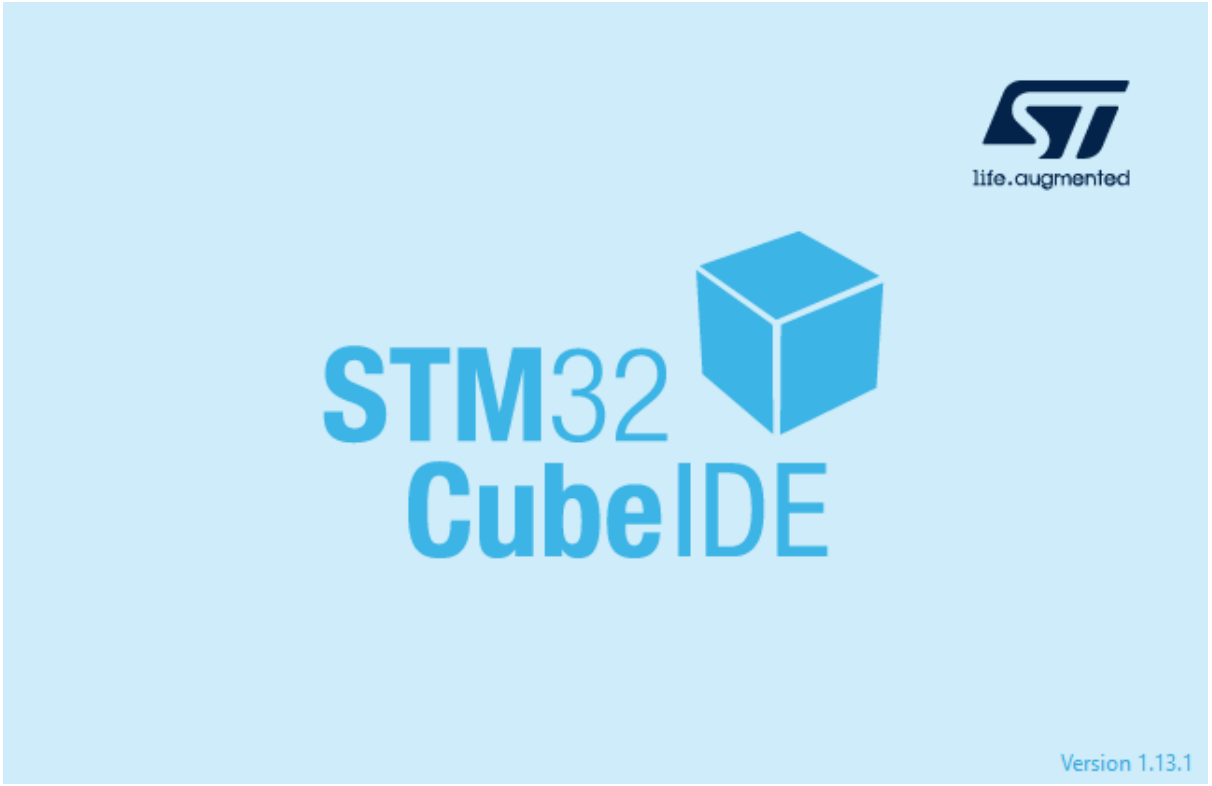
STM32 MCUs portfolio



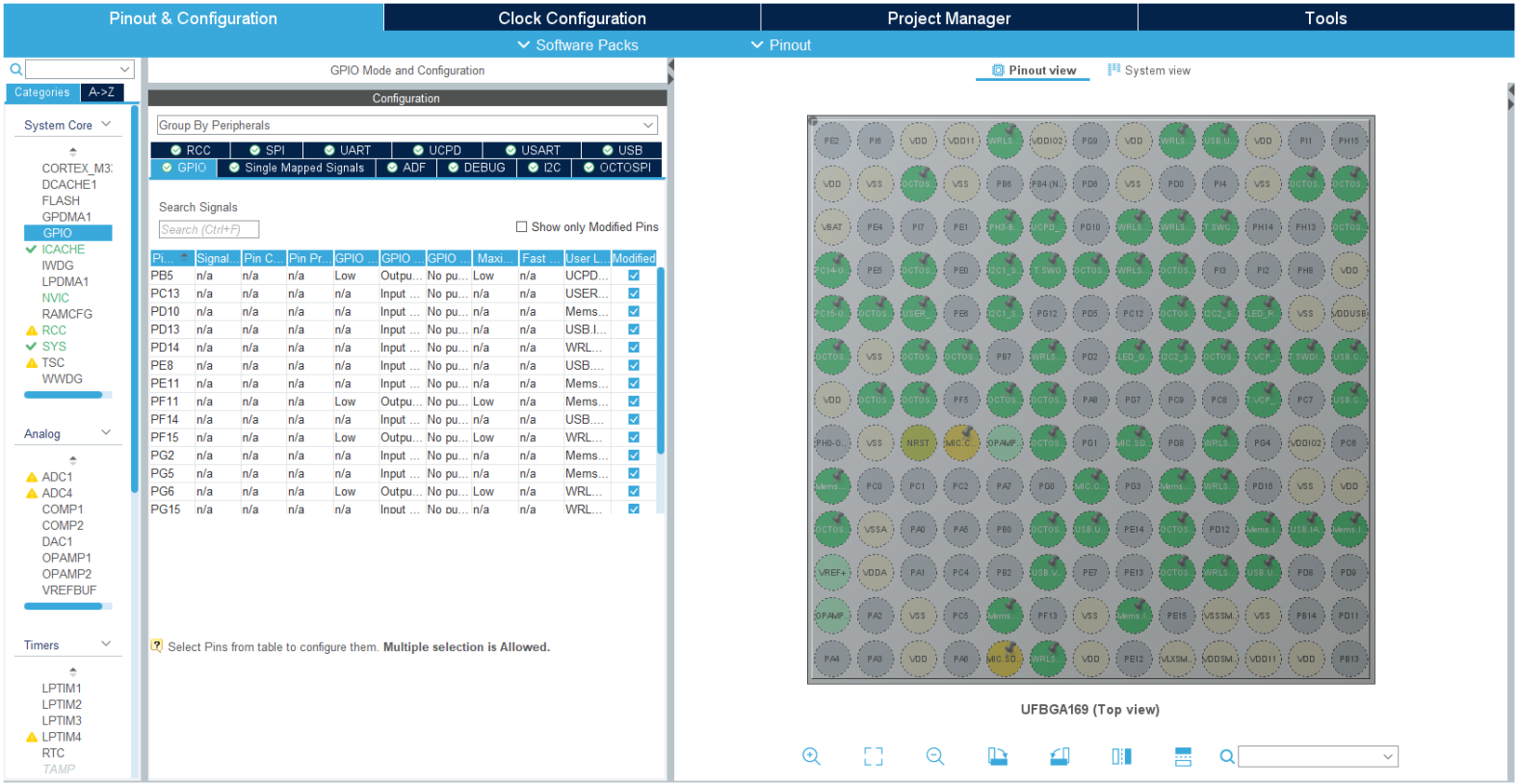
STM32U585 are 32-bit microcontrollers based on Cortex-M33 core



STM32U585 are programmed using STM32CubeIDE

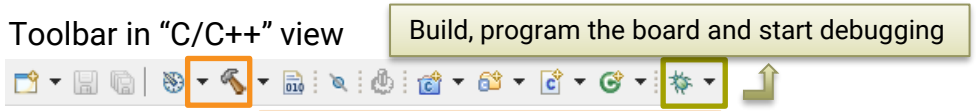


STM32U585 are programmed using STM32CubeIDE – and Cube MX

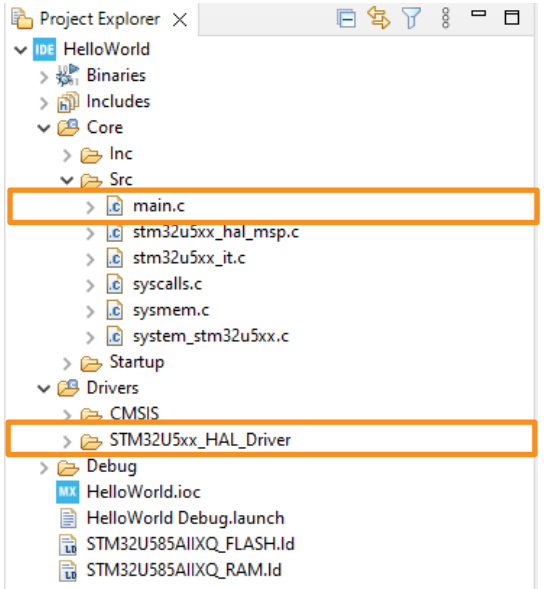


STM32U585 are programmed using STM32CubeIDE

Toolbar in “C/C++” view



Navigate inside your project files



Edit your source files

main.c

```c
#include "main.h"

int main(void)
{

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

    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
    HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin, GPIO_PIN_SET);
    HAL_Delay(500);
    HAL_GPIO_WritePin(LED_RED_GPIO_Port, LED_RED_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(LED_GREEN_GPIO_Port, LED_GREEN_Pin, GPIO_PIN_RESET);
    HAL_Delay(500);
  }
  /* USER CODE END 3 */
}
```

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

```
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
	0x0C00 0000 0x0FFF FFFF
SRAM	0x2000 0000 0x2FFF FFFF
	0x3000 0000 0x3FFF FFFF
Peripherals	0x4000 0000 0x4FFF FFFF
	0x5000 0000 0x5FFF FFFF

Program instructions

Data memory (variables)

Peripheral control

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 ?

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

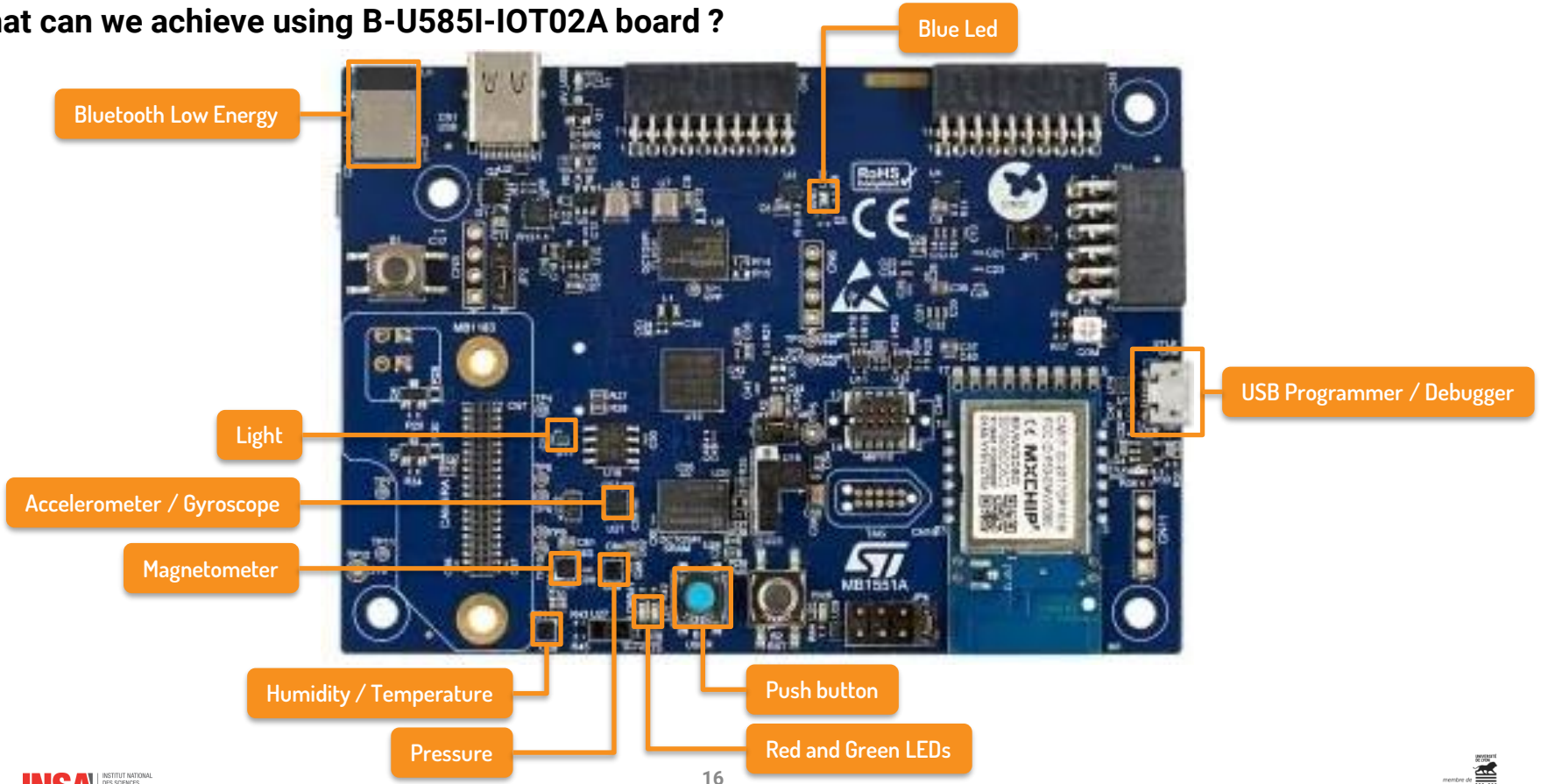
Register level programming

```
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_InitStruct =
  {
    .Pin = GPIO_PIN_0,
    .Mode = GPIO_MODE_OUTPUT_PP
  };
  HAL_GPIO_Init(GPIOH, &GPIO_InitStruct);

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

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 opertions
  - ST website link

Practical #1

Objectives :

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

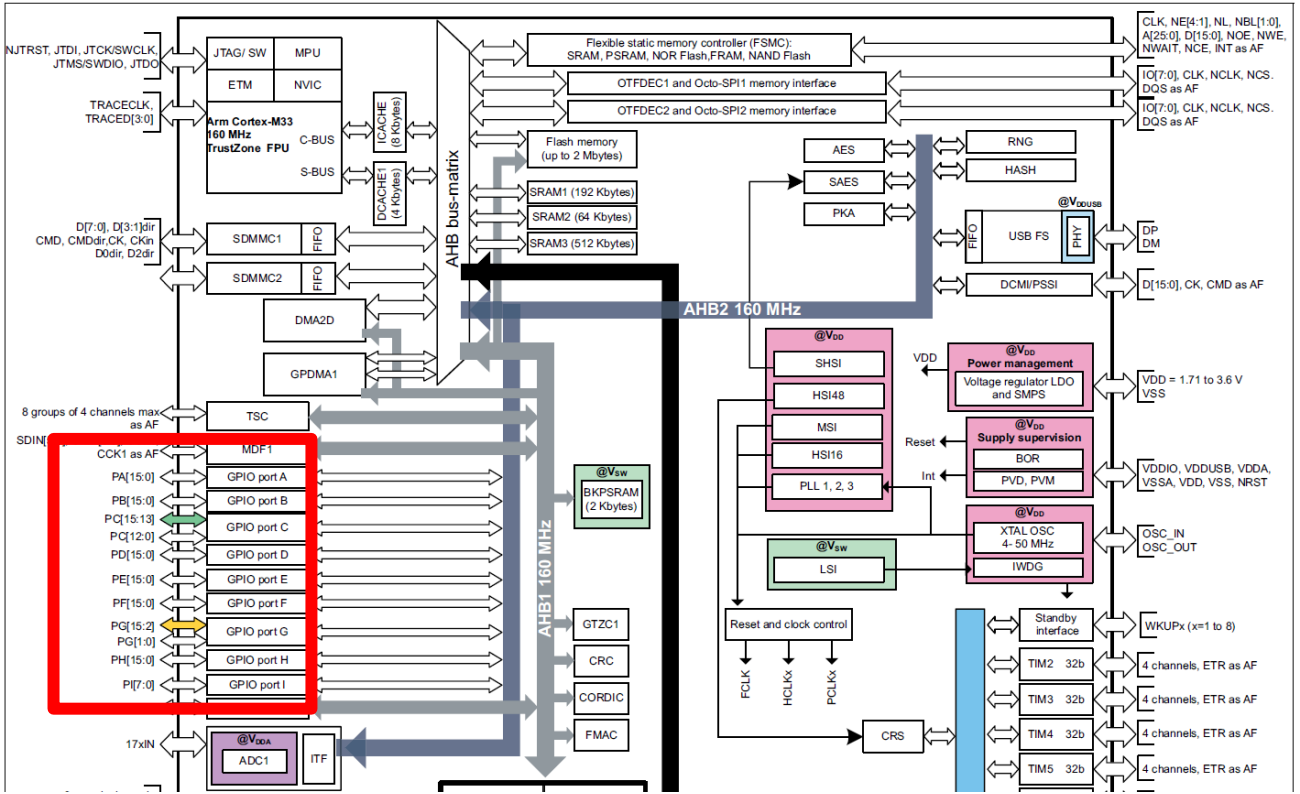
Constraints :

- Use HAL functions

General Purpose Inputs / Outputs (GPIO)

Handling digital inputs and outputs of the STM32U585

General Purpose Inputs / Outputs



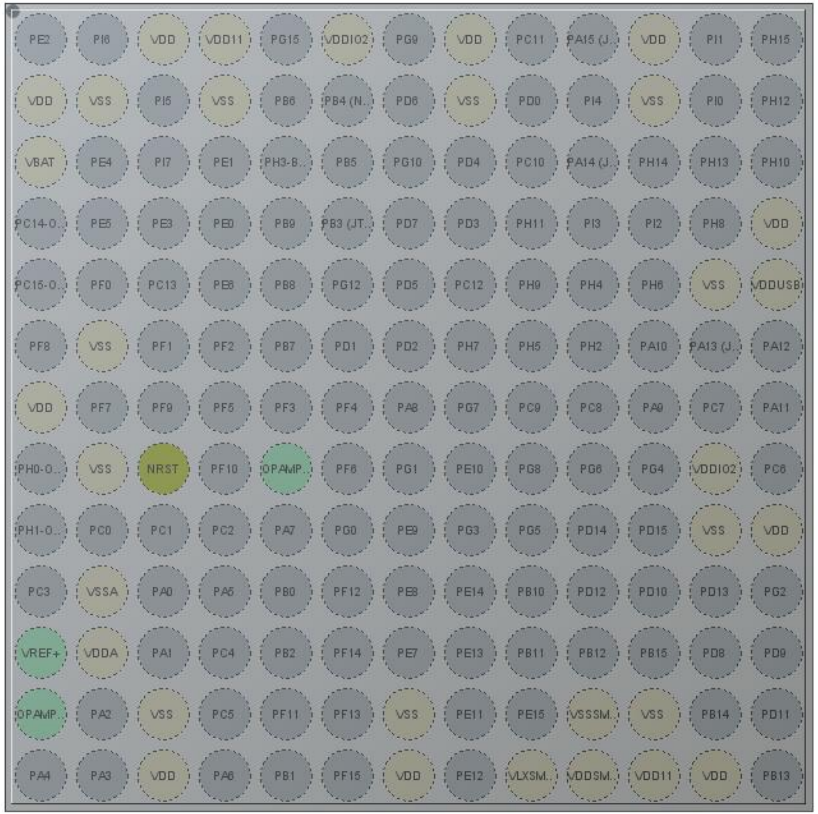
133 over 169 pins on the STM32U585AII6Q are GPIOs

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

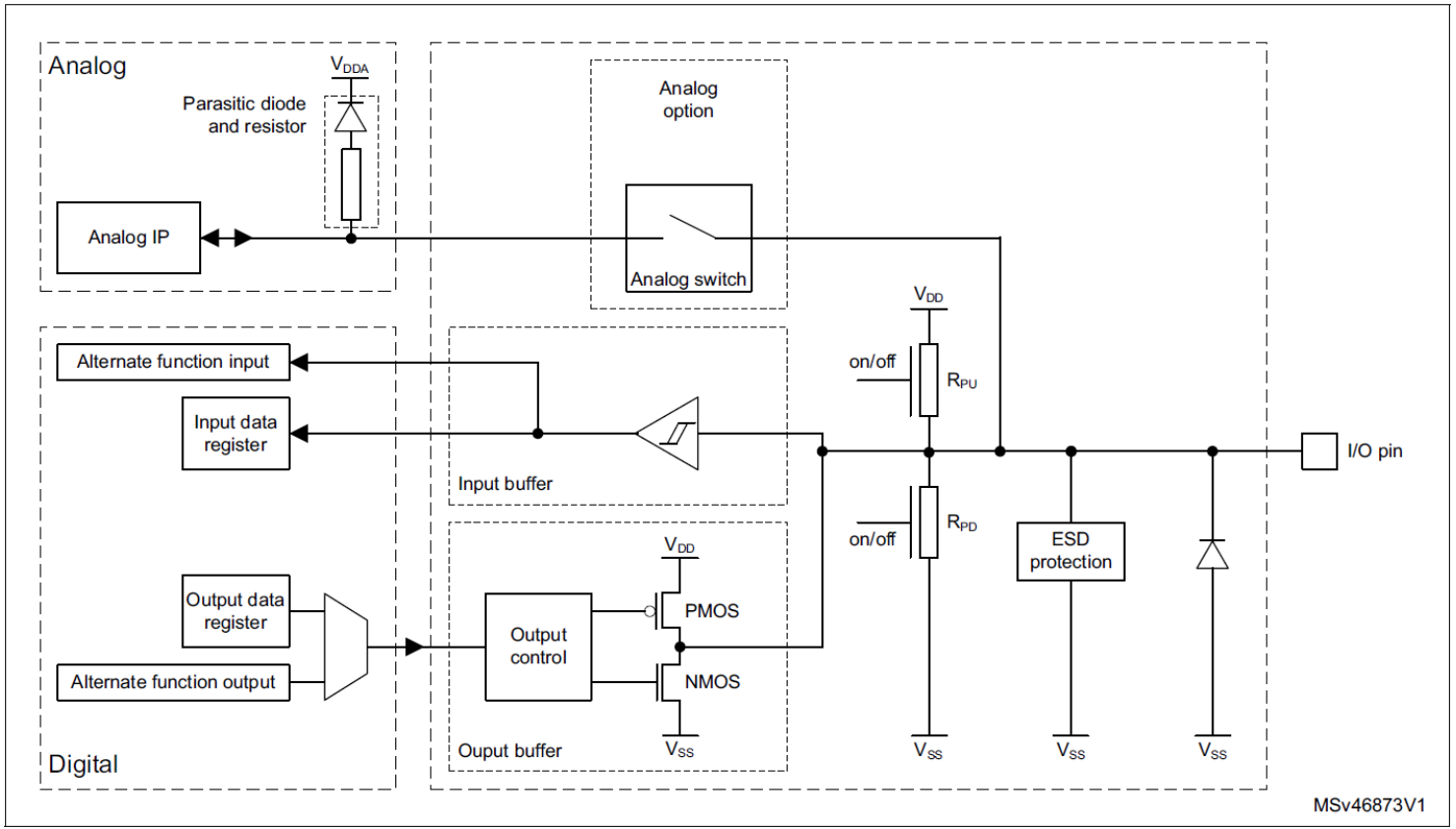
- Input floating (Hi-Z, default)
- Output low or high (logical 0 or 1)
- Analog
- Alternate function input / output

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.

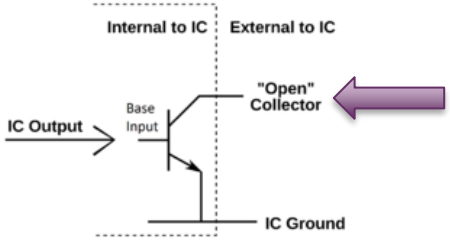


General Purpose Inputs / Outputs



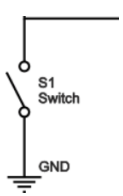
Reminder on pull-up / pull-down inputs

- 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.)



This output can be High-Z or low

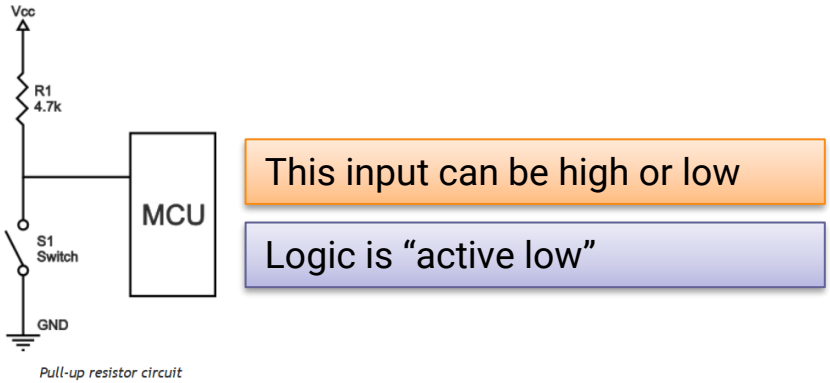
- Useful for switch buttons



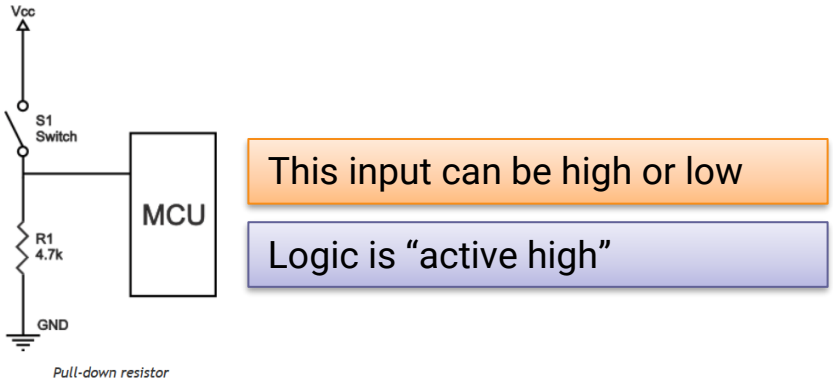
This input can be High-Z or low

Reminder on pull-up / pull-down inputs

- Useful for switch buttons



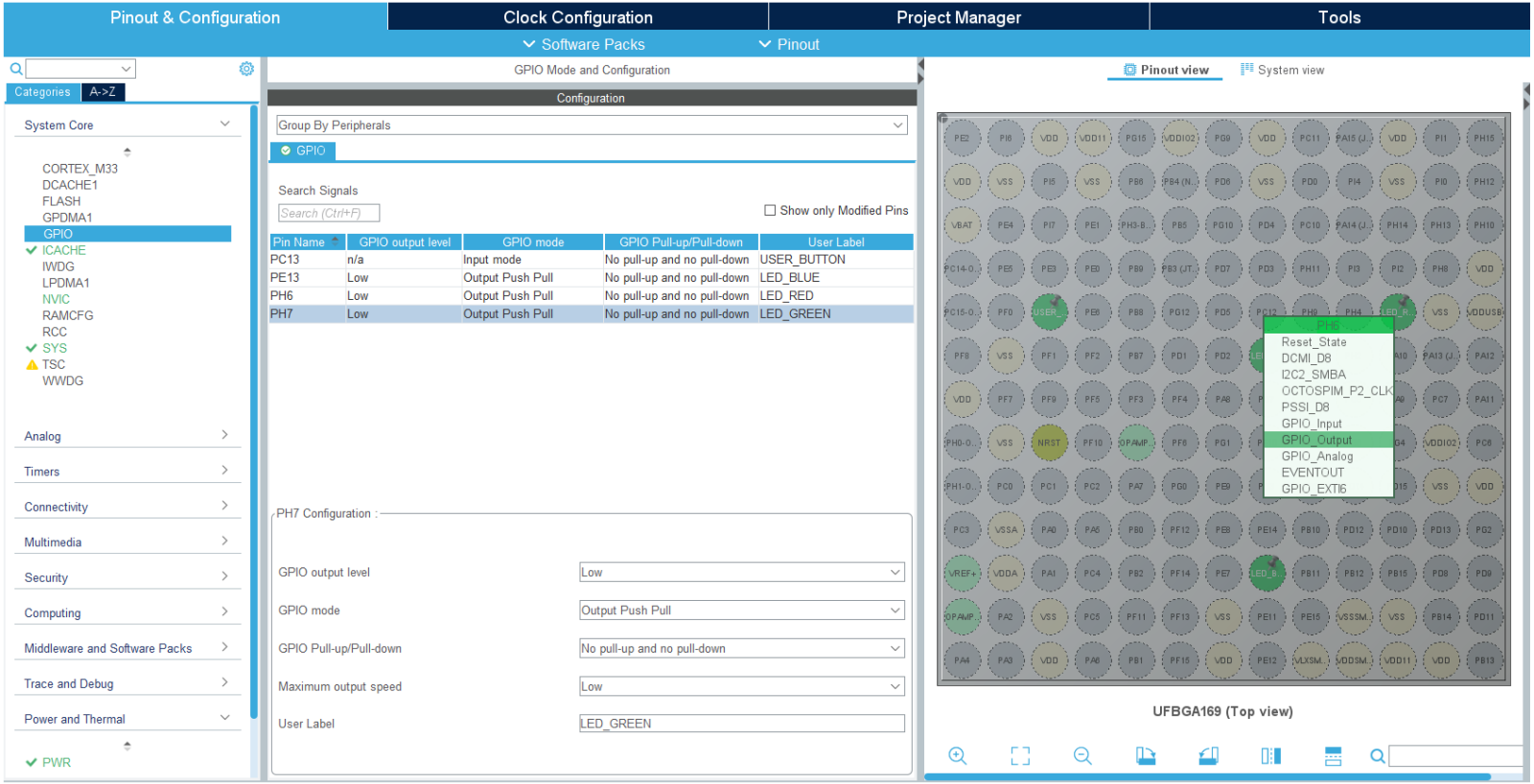
Pull-up resistor circuit



Pull-down resistor

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

Cube MX configuration



General Purpose Inputs / Outputs

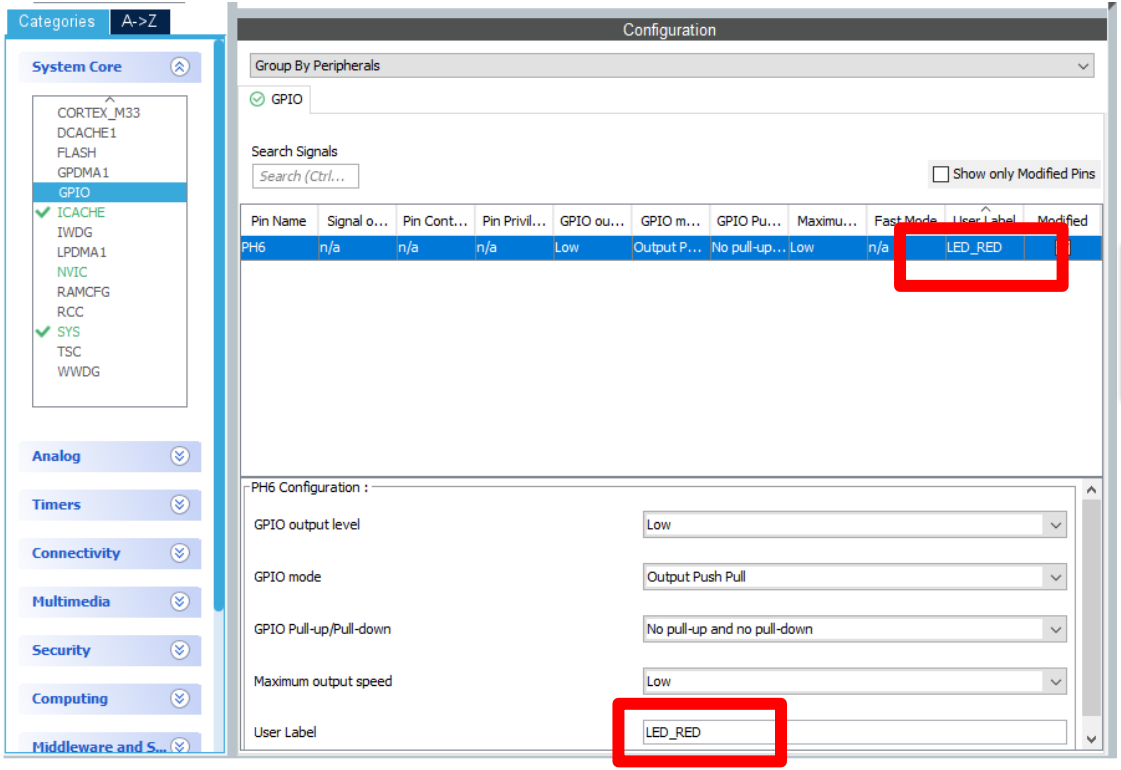
- **With Cube MX (.ioc), configuration / initialization is automatically generated**
- **With the HAL library, register access is handled by the library**

```
// 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




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

STM32CubeIDE – Hints

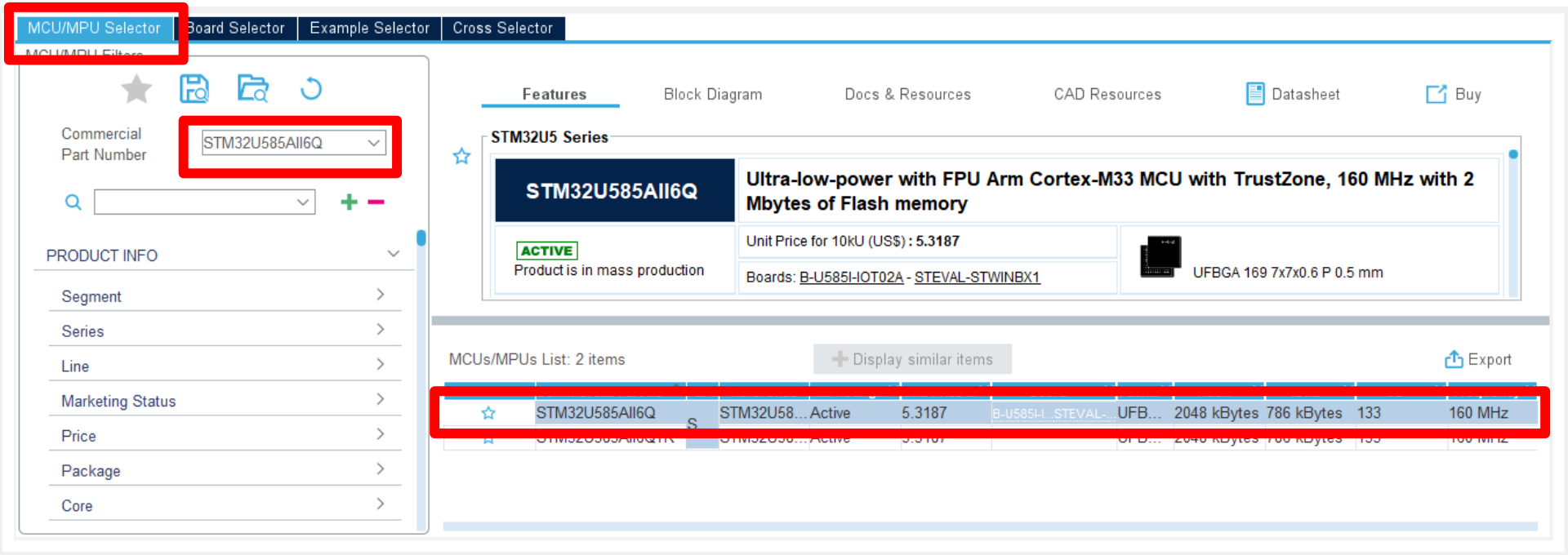
```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
/* USER CODE END WHILE */


/* 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

STM32CubeIDE – Blank project

Cube MX configuration



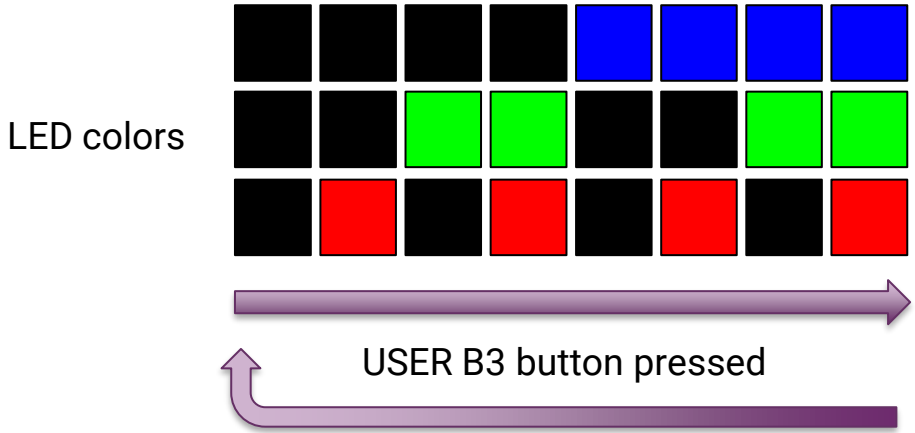
Practical #2

Objectives :

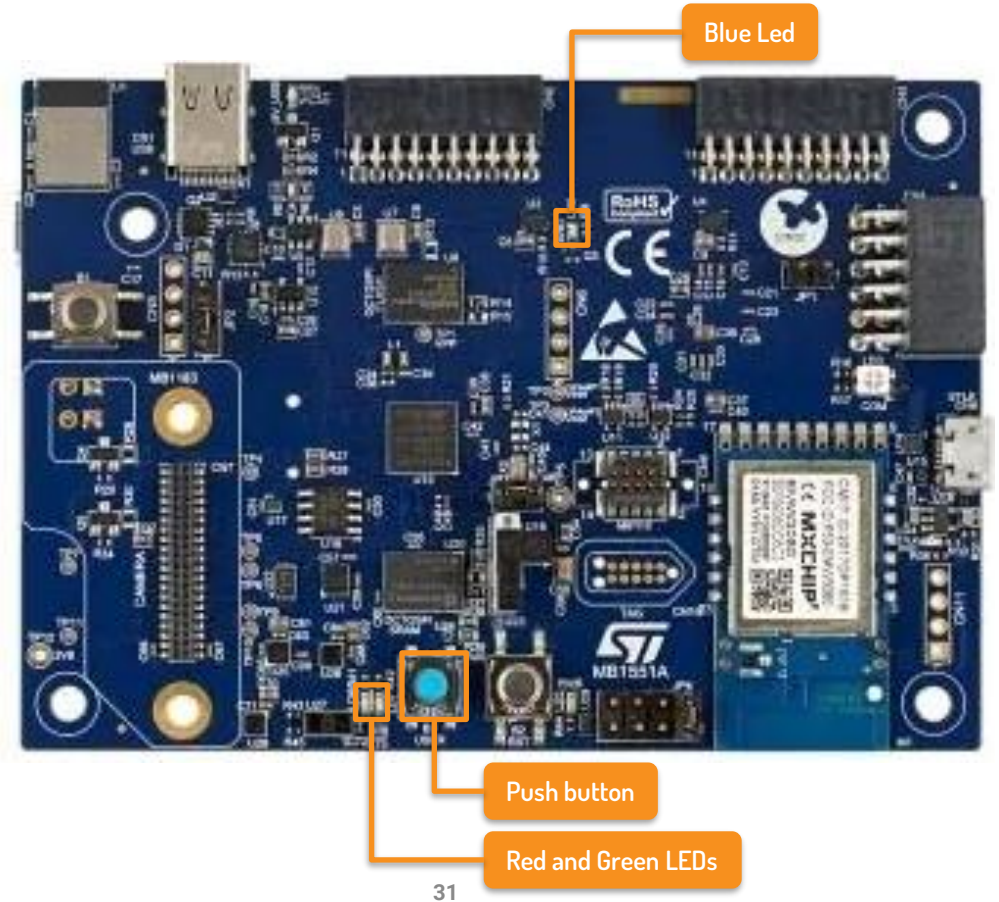
- Implement a program to control color of the red, green and blue LED using the switch button

Constraints :

- Start with empty Cube MX (select STM32U585AII6Q)
- Use HAL library (no direct access to peripheral registers)
- Use GPIO input / output only

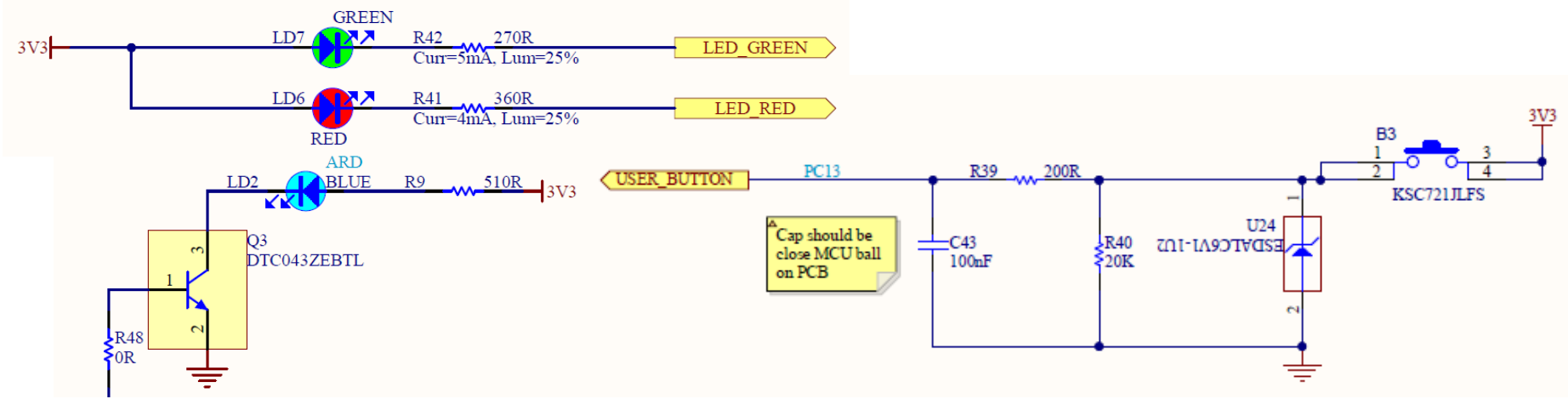


Practical #2



Practical #2

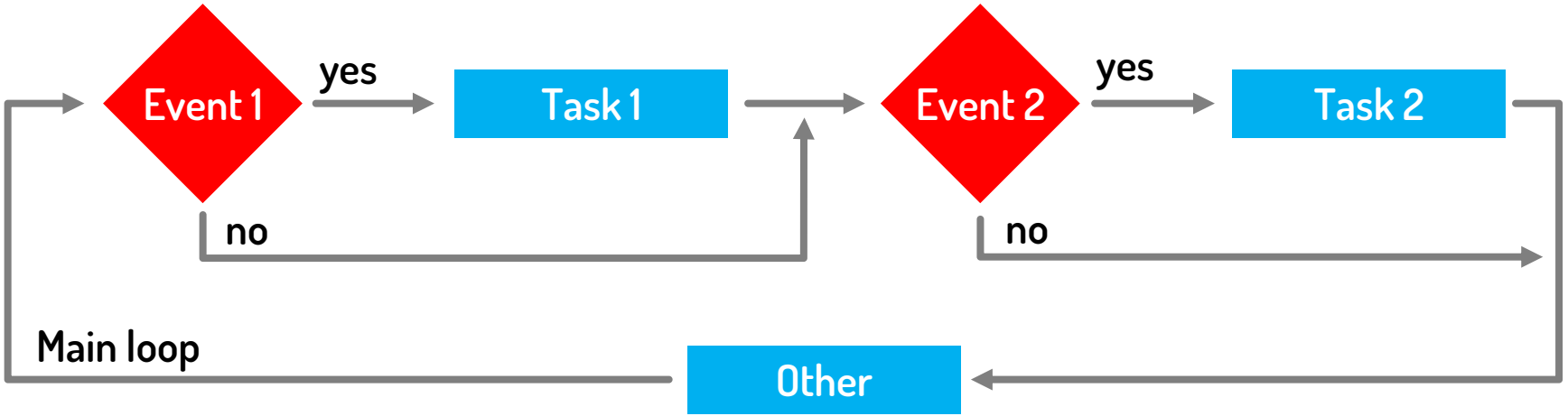
I/O	Reference	Color	Name	Comment
PC13	B3	Blue	USER	User button
PH7	LD7	Green	LD7	User LED lights up when PH7 is set to 0.
PH6	LD6	Red	LD6	User LED lights up when PH6 is set to 0.
PE13	LD2	Blue	ARD	ARDUINO® LED lights up when PE13 is set to 1.



Interruptions - Part #1

Basic mechanisms of interruptions and event-based programming principles

Executing a program sequentially without interrupts is called *polling*

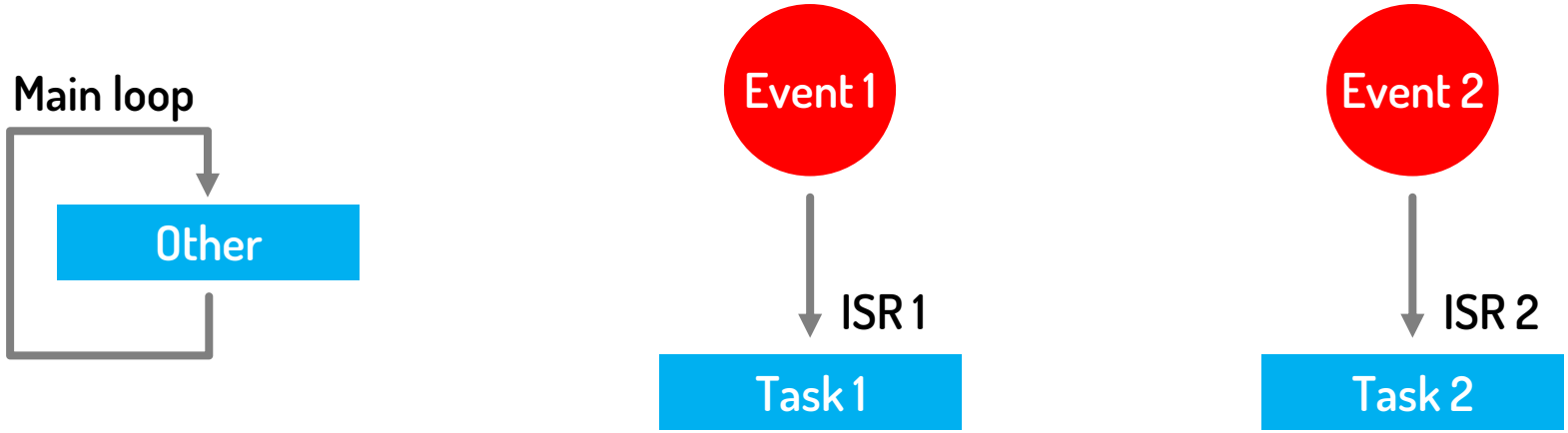


- Events occurring during execution, such a button pushed, are detected by successive testing of the inputs in a loop.
- When an event is detected, an appropriate function is called to handle the required behavior
- Meanwhile other events can not be detected anymore !
- CPU is **always busy**

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**

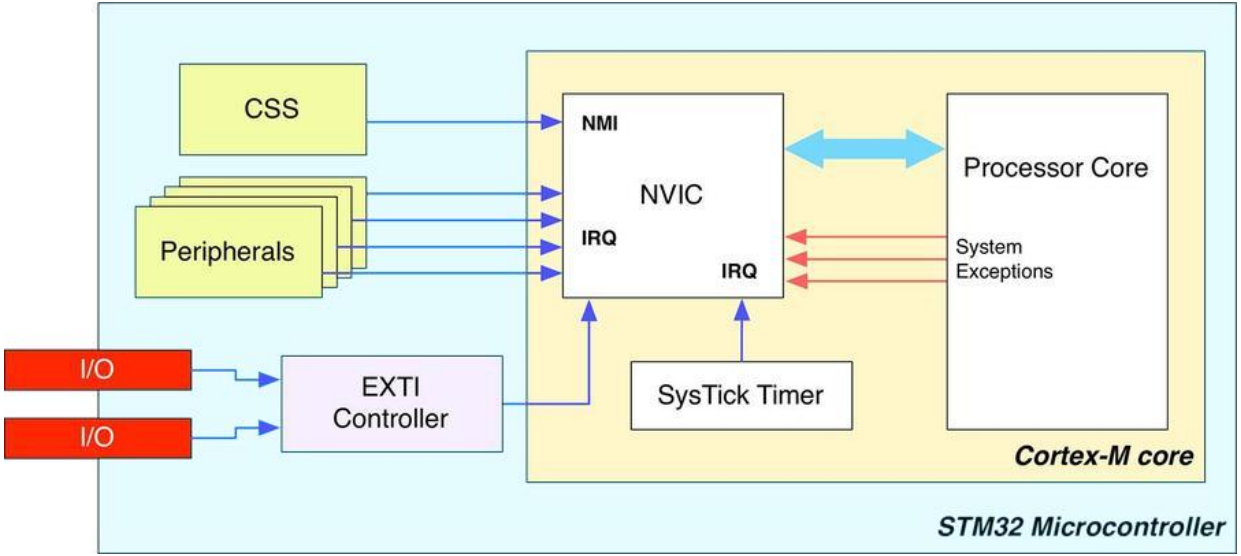


- Whatever the program is doing, events will be detected and handled accordingly
- This provides much more reactivity for tasks with high ***priority***
- The CPU can be put in **idle state** if there is no action to do, except waiting for events

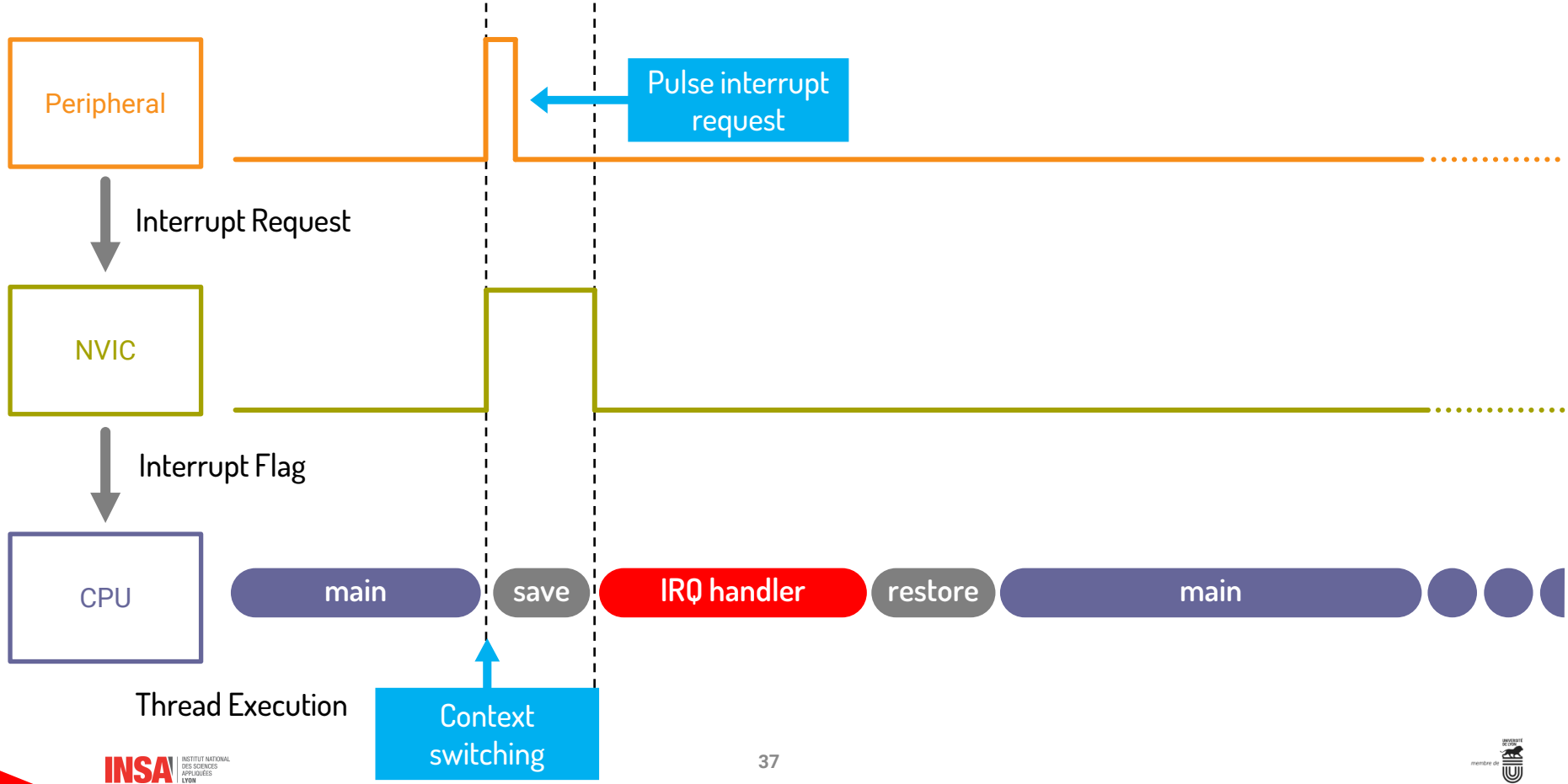
A dedicated hardware block is responsible for handling interrupts

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

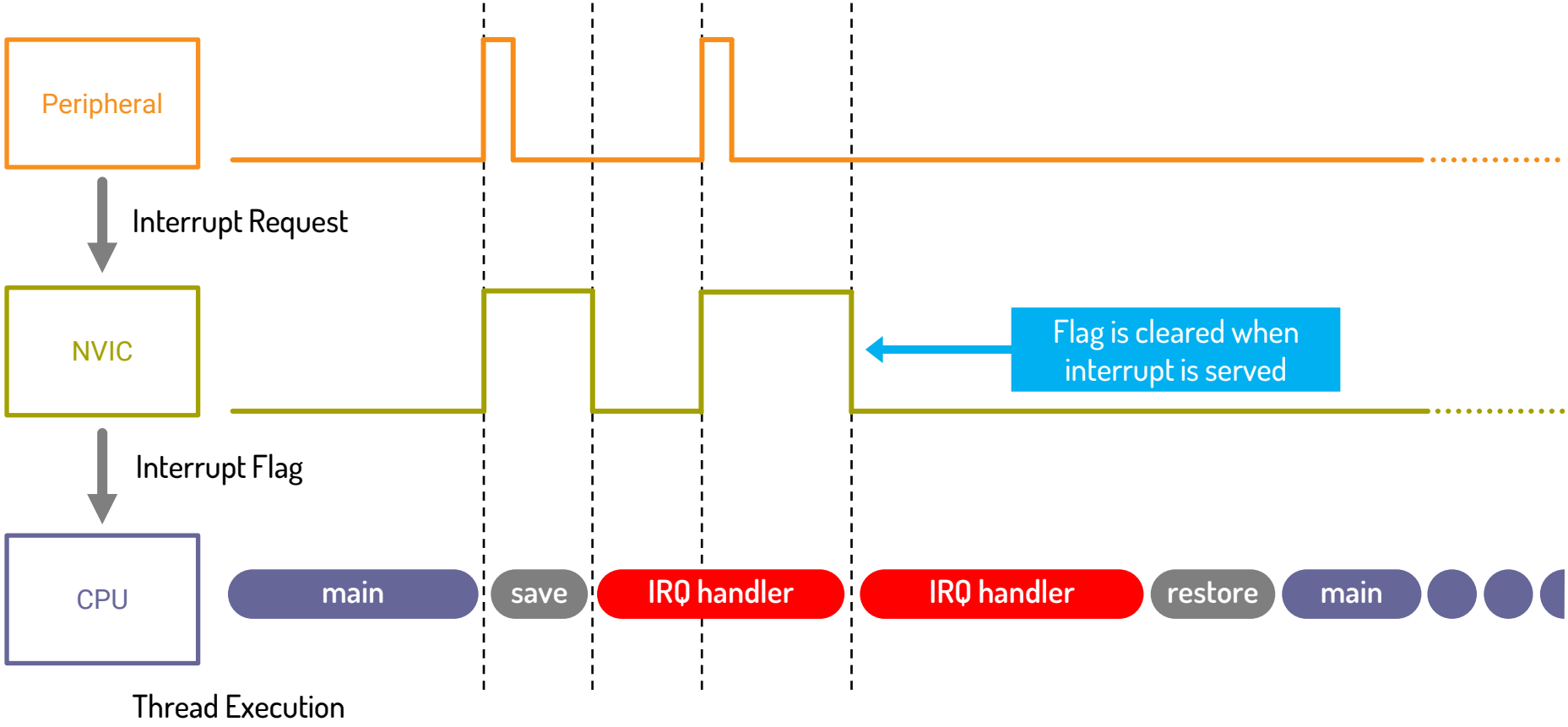
- The NVIC collects all interrupt signals from peripherals
- Saves current CPU state (registers, stack, etc.)
- Sets the next instruction pointer to the address of the *interrupt service routine*
- Restores previous saved CPU state when the ISR returns



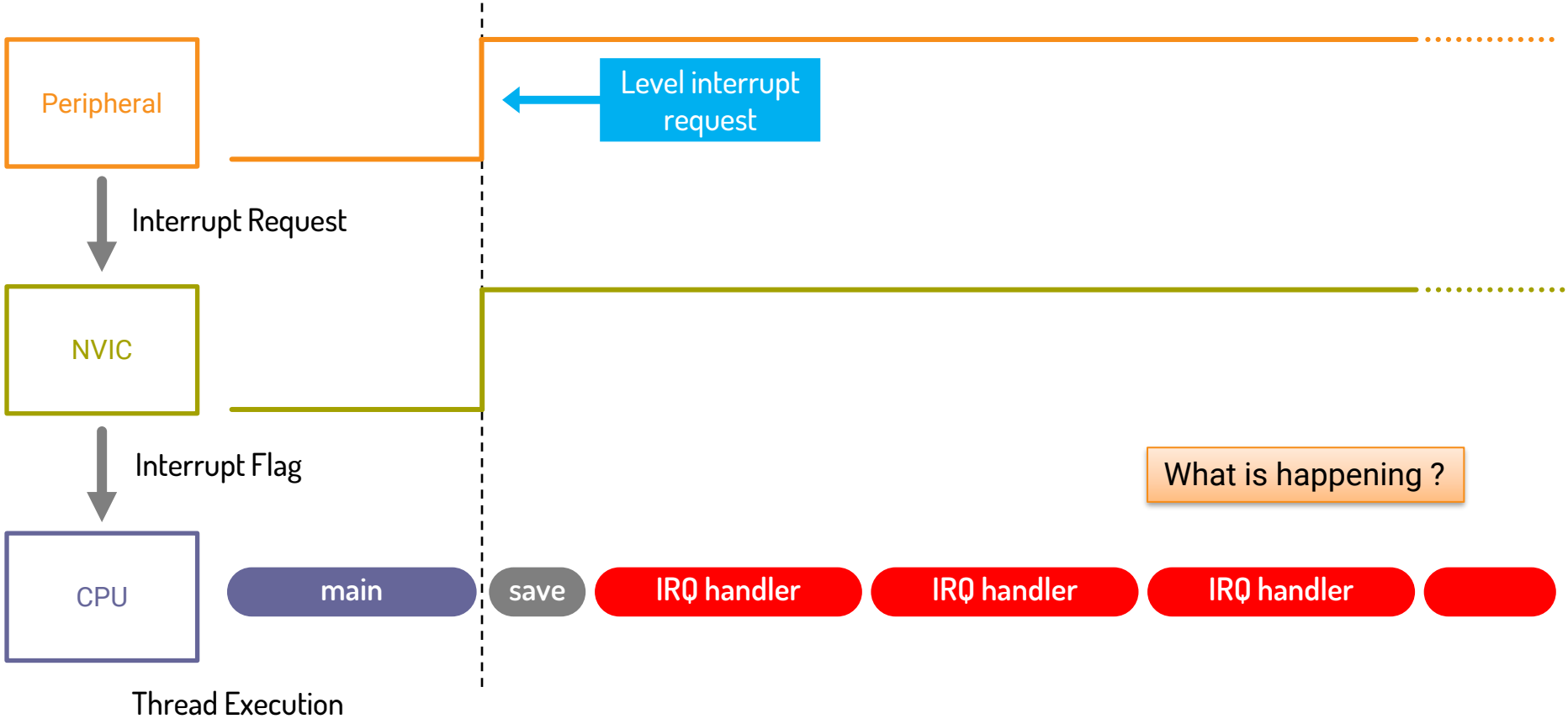
Serving a single Interrupt Request (IRQ)



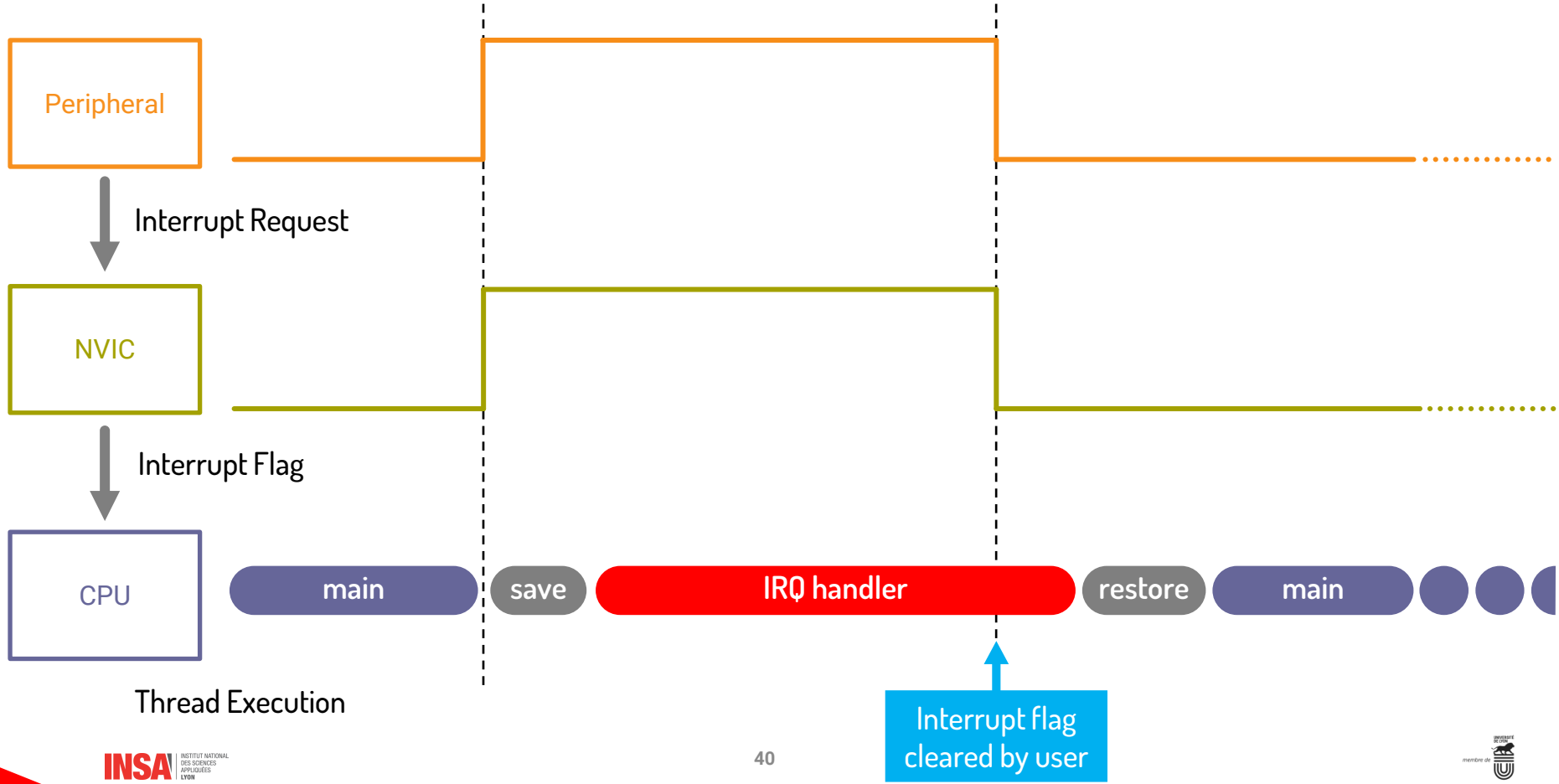
Serving a single Interrupt Request (IRQ)



Serving a single Interrupt Request (IRQ)



Serving a single Interrupt Request (IRQ)

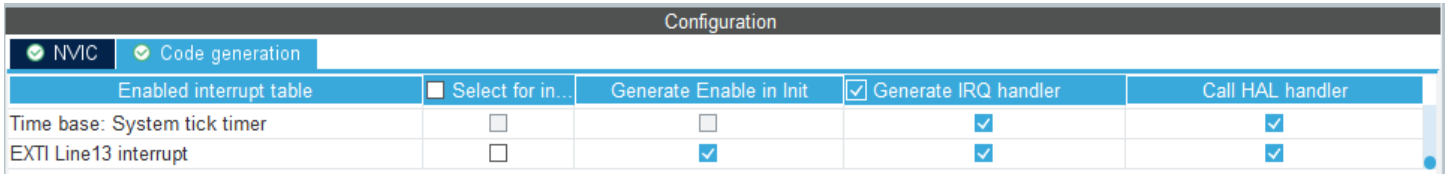


Some examples of interrupt sources:

- 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



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

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 Premption Priority and Sub Priority | Sort by interrupts names

Search: Search (Ctrl+F) | 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

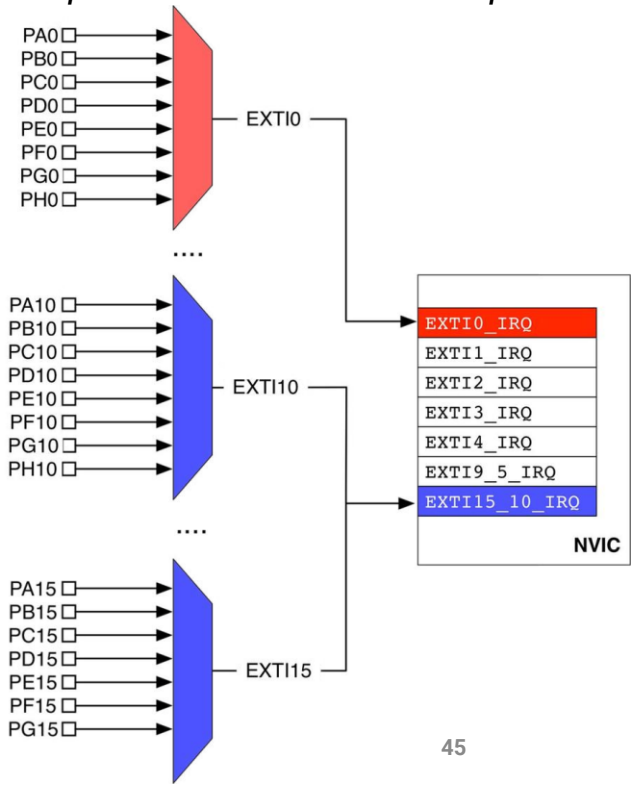
Interrupts can be configured and activated from Cube MX depending on peripheral configuration

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	☐	☑	☑	☑

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.



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 */
}
```

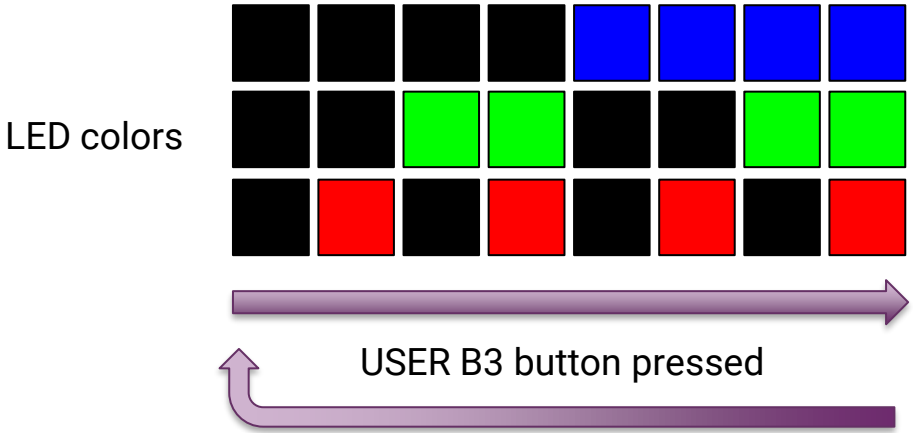
Practical #3

Objectives :

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

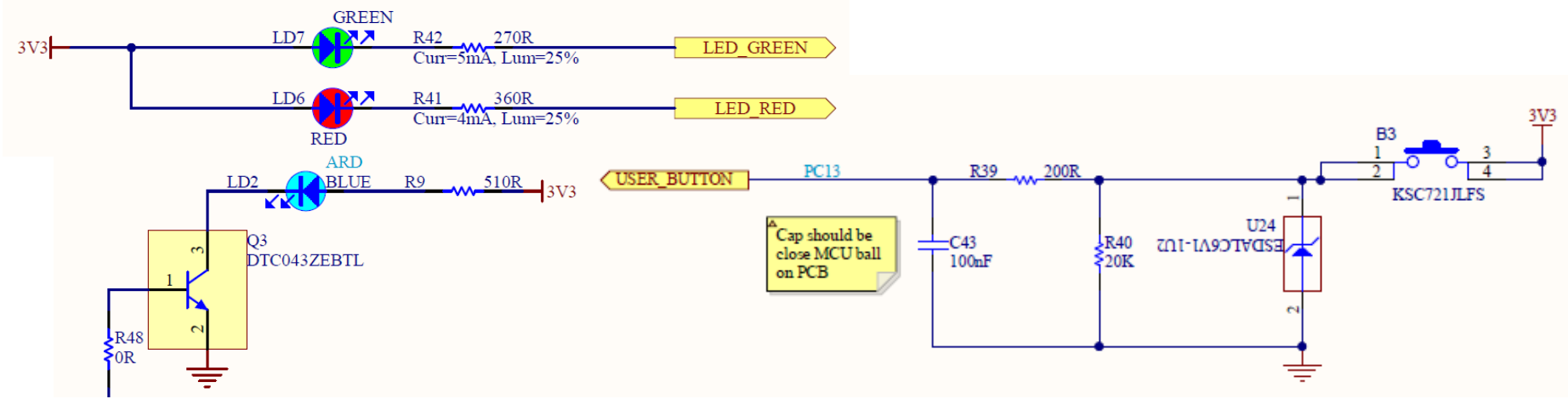
Constraints :

- Start with empty Cube MX (select STM32U585AII6Q)
- 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.

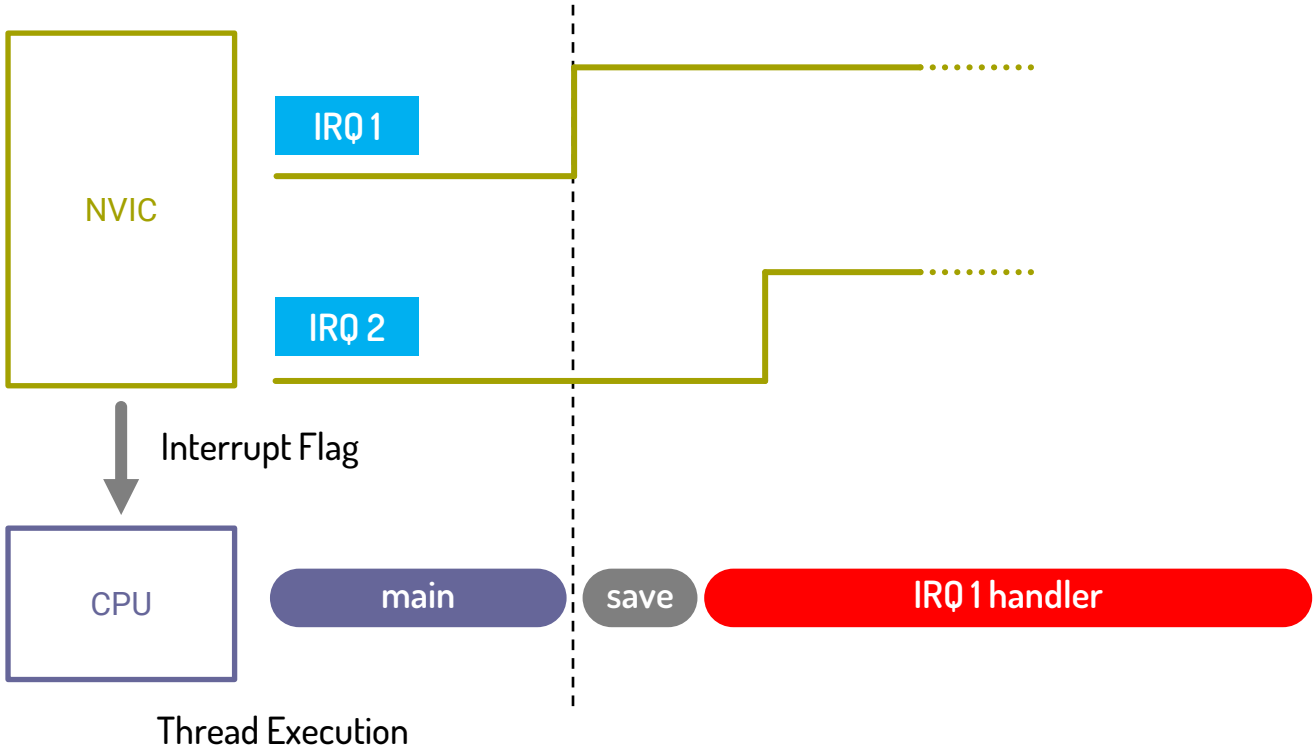


Interruptions - Part #2

Nested interrupts, priorities

Serving a single Interrupt Request (IRQ)

What happens if two interrupts occur closely / simultaneously ?



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
[…]	[…]	[…]

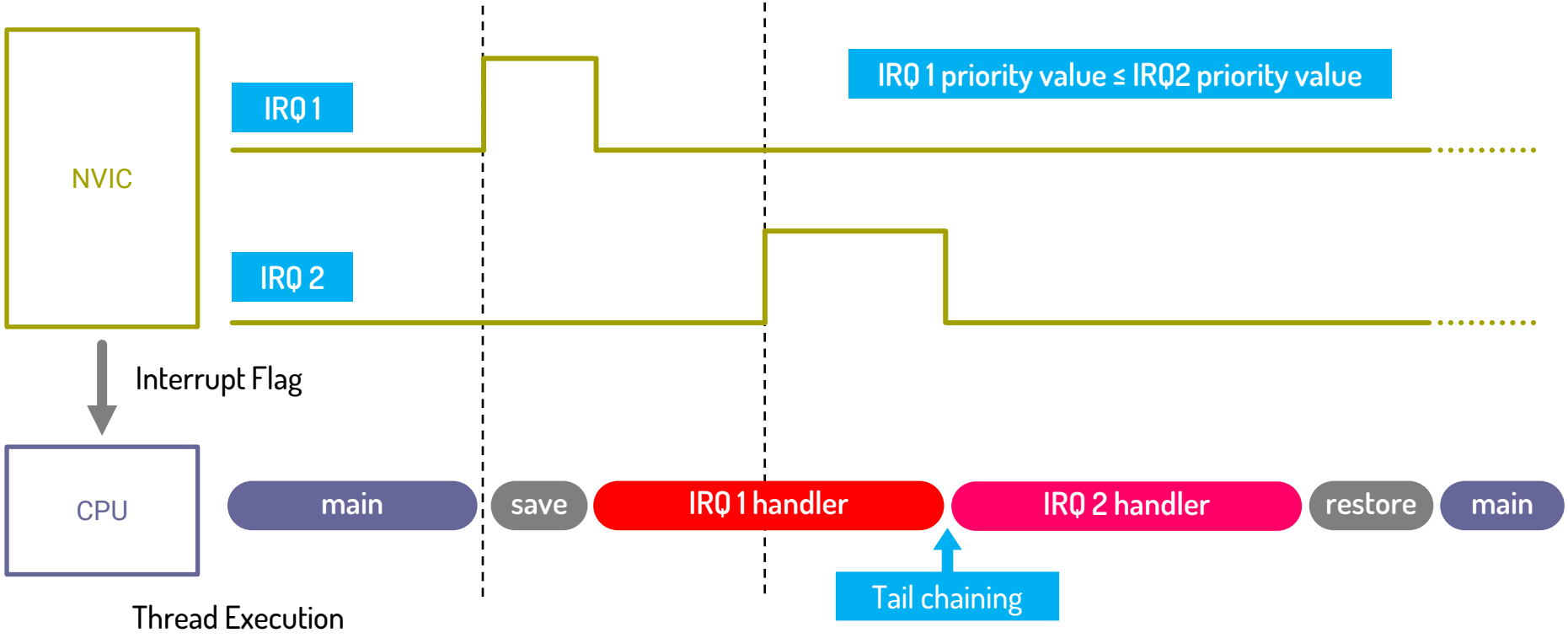
Priority value is comprised between 0 (highest priority) and 15 (lower priority)

User programmable priorities

The lower the number, the higher the priority

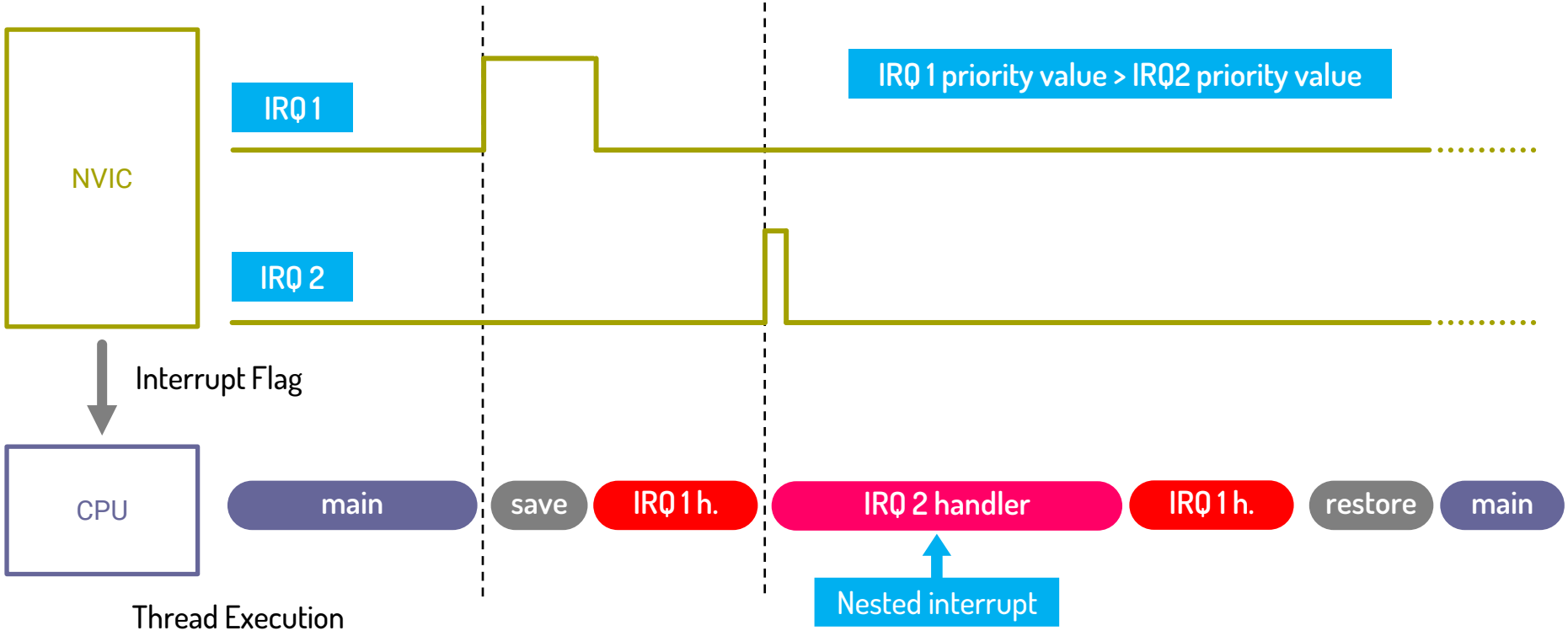
Serving a single Interrupt Request (IRQ)

What happens if two interrupts occur closely / simultaneously ?

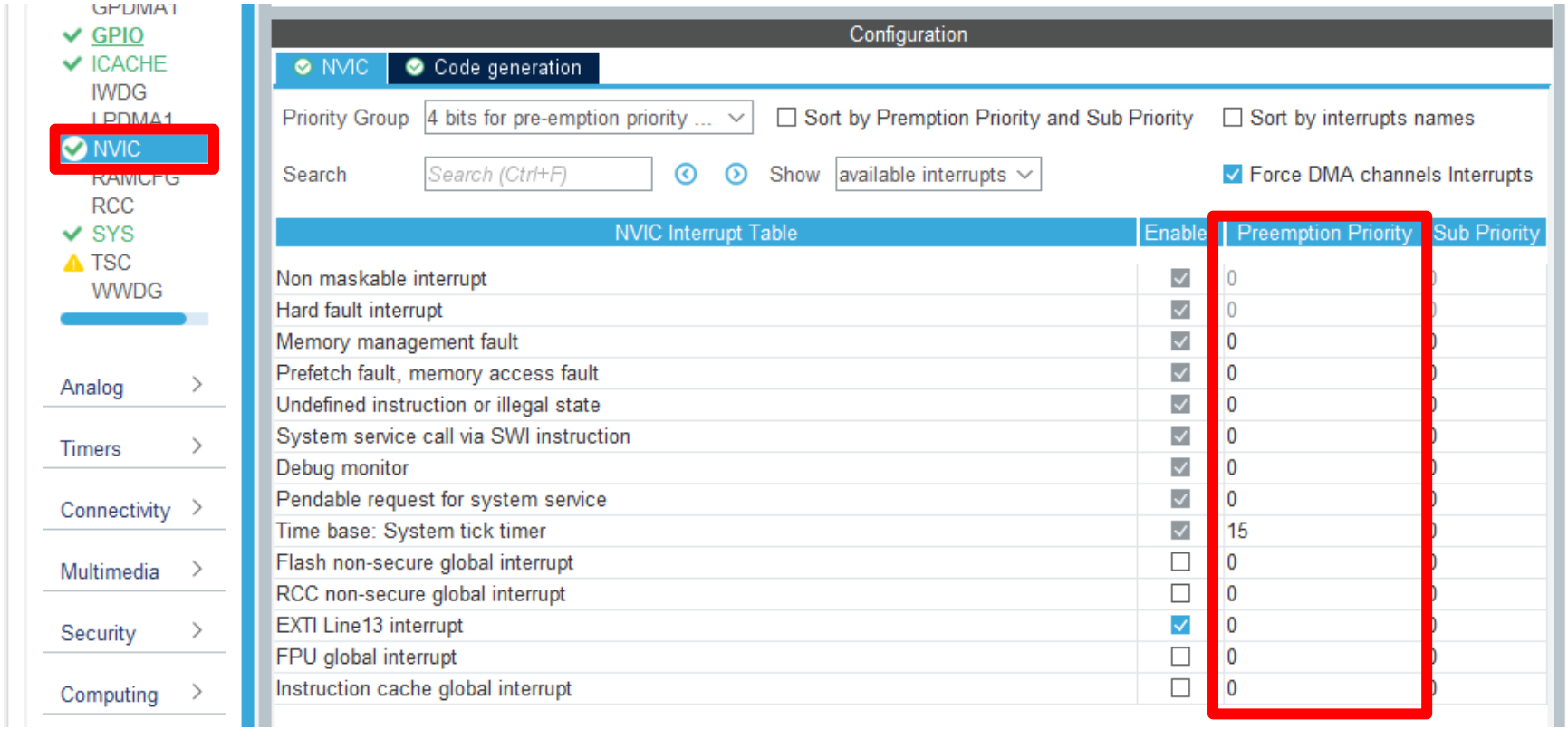


Serving a single Interrupt Request (IRQ)

What happens if two interrupts occur closely / simultaneously ?



Configure interrupt priorites in Cube/X



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

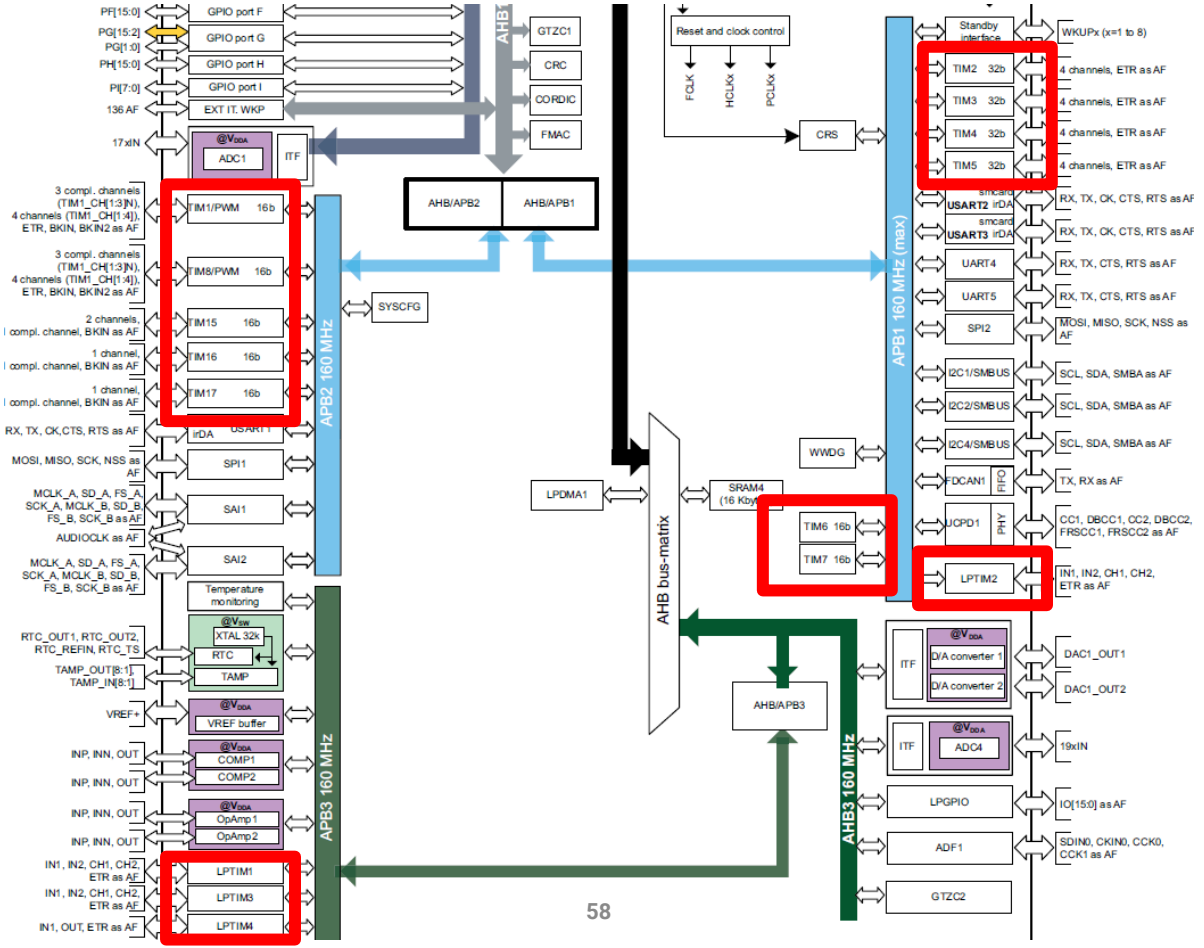
- The time counting is not precise (system clock frequency, compiler optimization)
- This function is blocking
- This is a terrible timer…

Hardware timers enable parallel counting and let the CPU free

Time management on microcontrollers

Timers

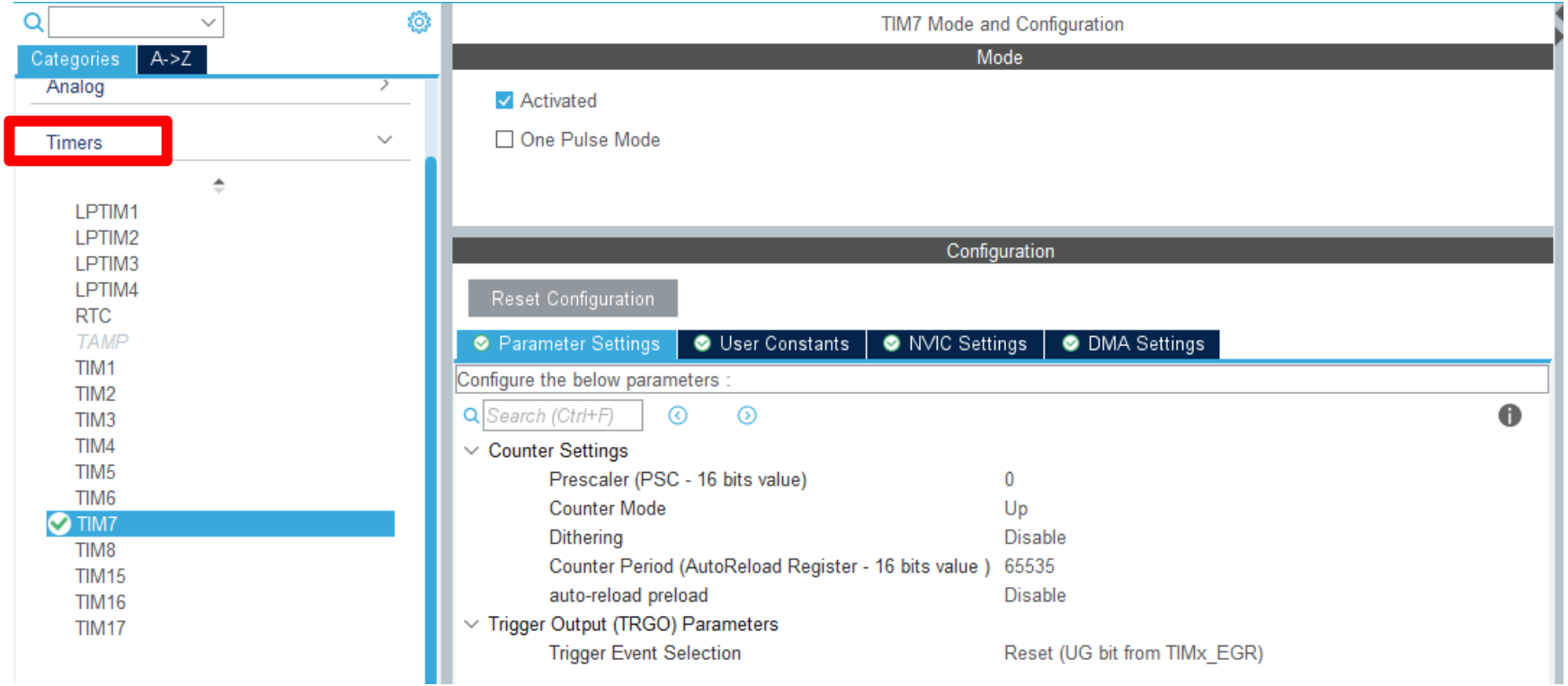
- LPTIM1
- LPTIM2
- LPTIM3
- LPTIM4
- RTC
- *TAMP*
- TIM1
- TIM2
- TIM3
- TIM4
- TIM5
- TIM6
- TIM7
- TIM8
- TIM15
- TIM16
- TIM17



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



SysTick Timer

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);

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

Wait for a number of tick

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

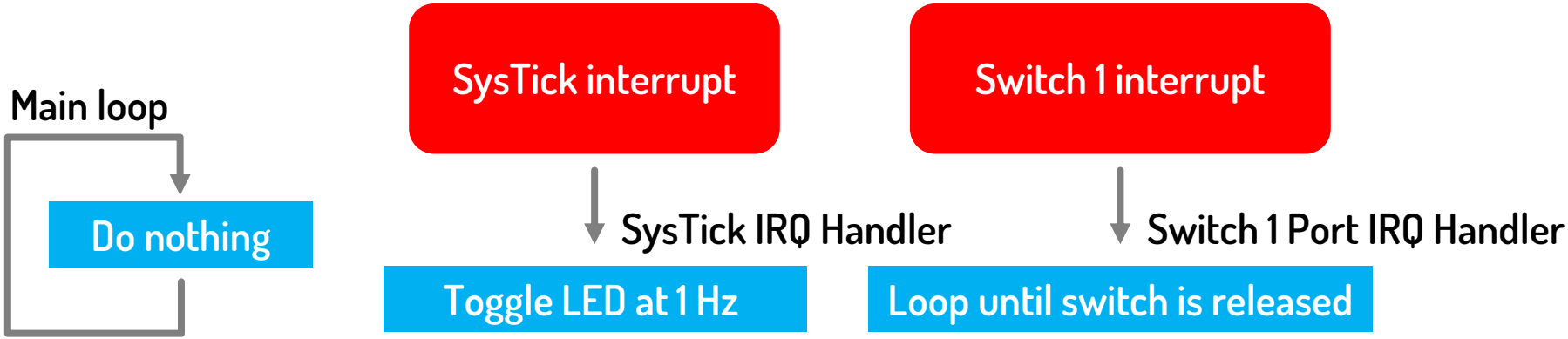
```
void user_button_interrupt(void)
{
    /* Anti rebond */
    HAL_Delay(10);
}
```

This will hang program execution

Practical #4

Objectives :

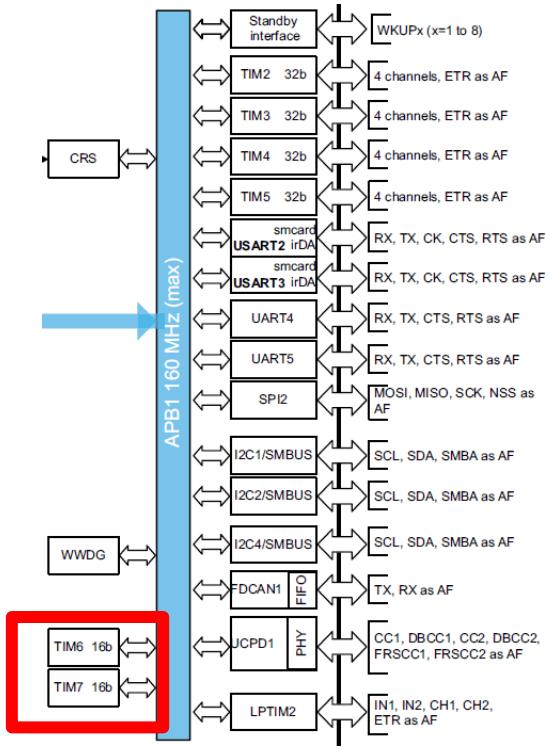
- Implement a program to blink the red led using SysTick timer interruption at 1 Hz.
- Implement a blocking loop in a Switch interruption (code stays in the interrupt while button is pushed)
- Use priorities to **force** or **prevent** the Switch interrupt to block SysTick timer interrupts



TIM6, TIM7 simple timers

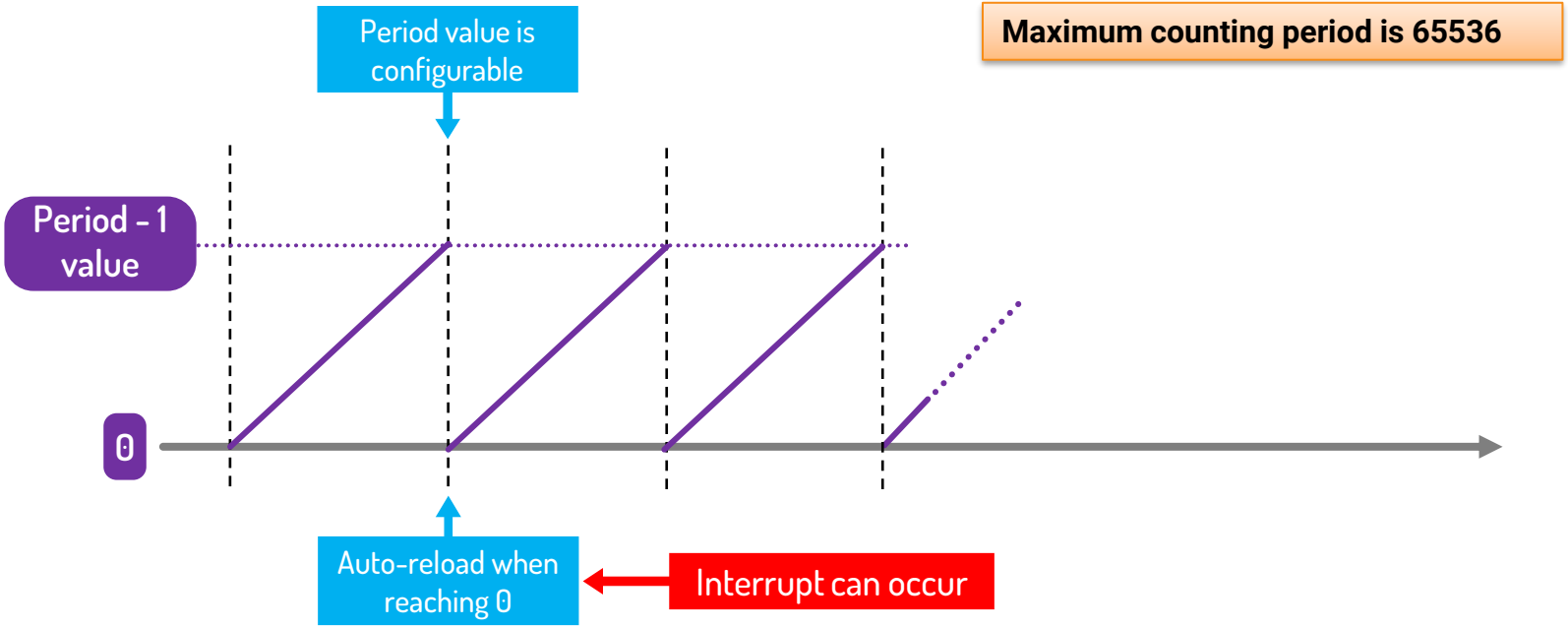
TIM6 and TIM7 timers are simple 16-bit up timers with no input / output connectivity

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



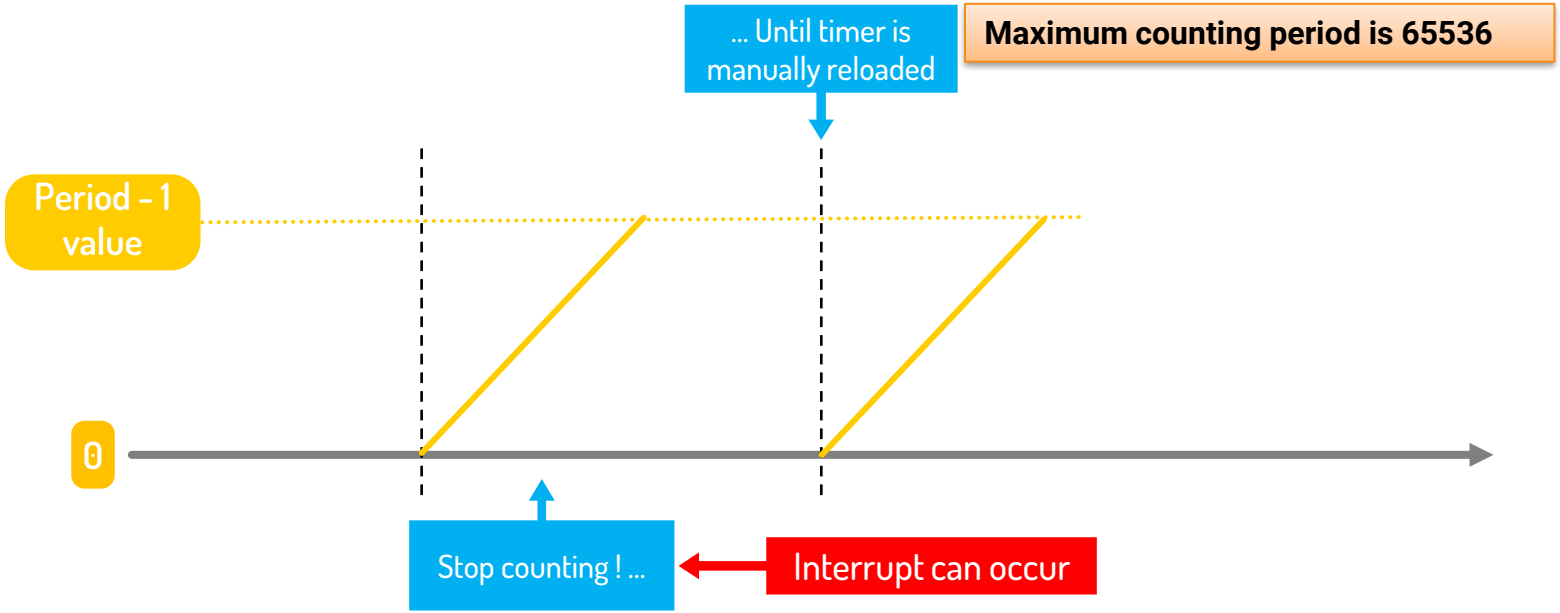
TIM6 and TIM7 contains one 16-bit counter, counting up in 2 possible configurations

Periodic mode (*One Pulse Mode* disabled)



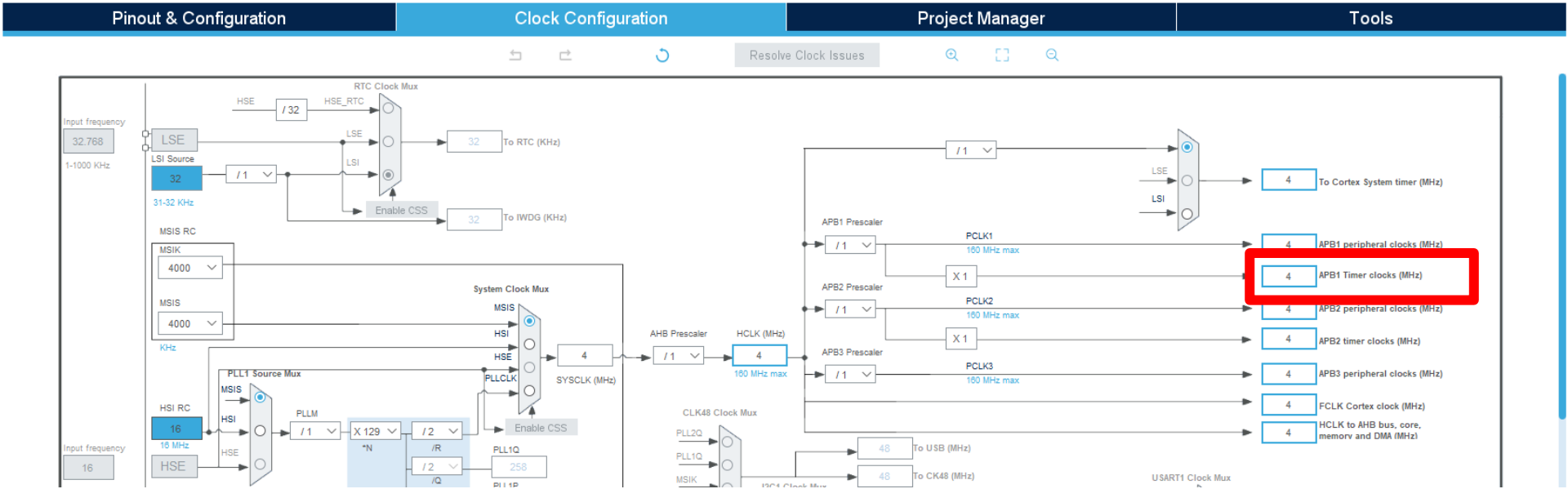
TIM6 and TIM7 contains one 16-bit counter, counting up in 2 possible configurations

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



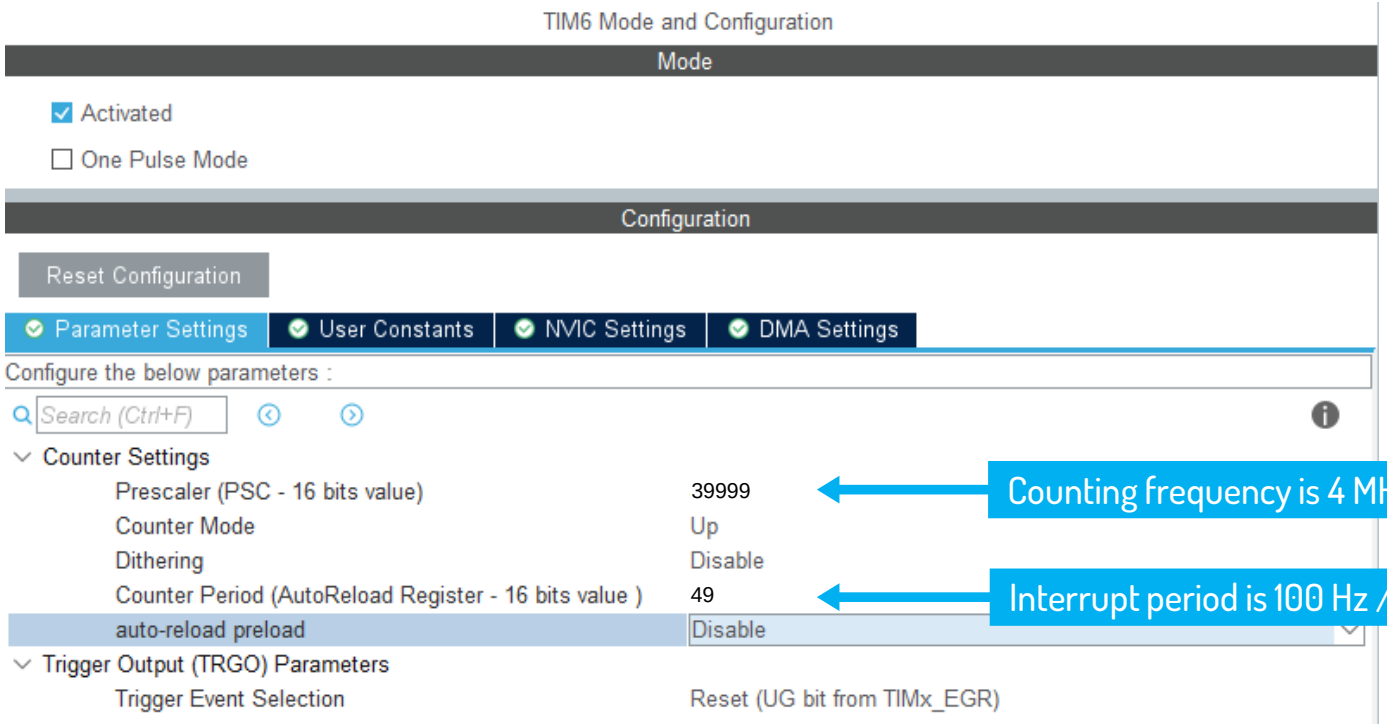
TIM6, TIM7 clock speed

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



TIM6, TIM7 clock speed

Timer has an internal pre-scaler to reduce counting frequency



Counting frequency is 4 MHz / (39999 + 1) = 100 Hz

Interrupt period is 100 Hz / (49 + 1) = 2 Hz

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 :

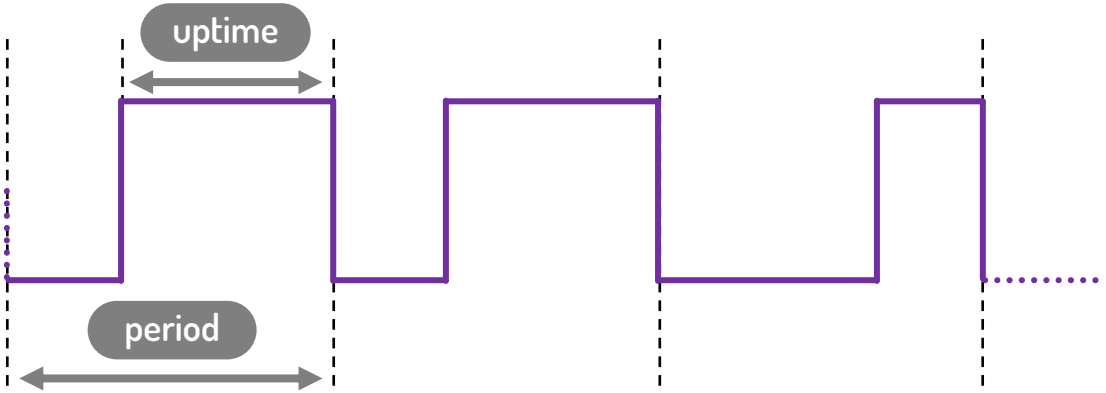
- Start from **Practical #4** project
- Main “while” loop is empty

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

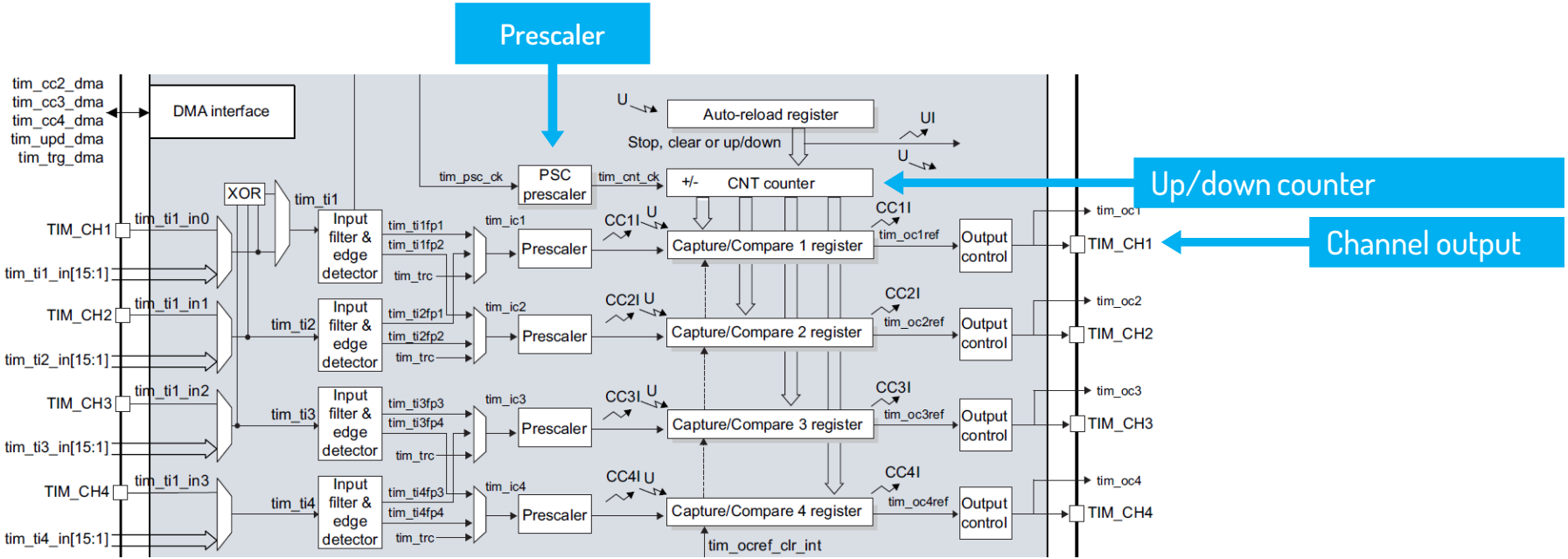
- **period** is constant
- **frequency** is 1 / (PWM period)
- **uptime** is variable
- **duty cycle** is uptime / period



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

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

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

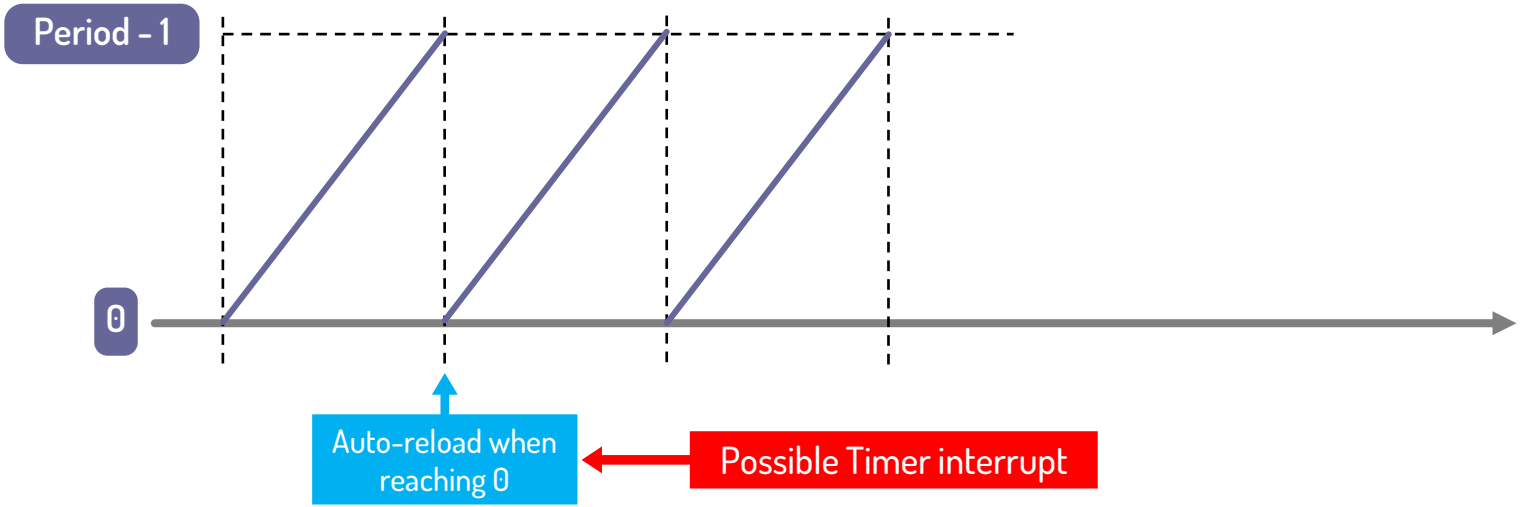
- The outputs TIMx_CHy are set/reset when counter reaches the corresponding CCy register

Details on the PWM Mode

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

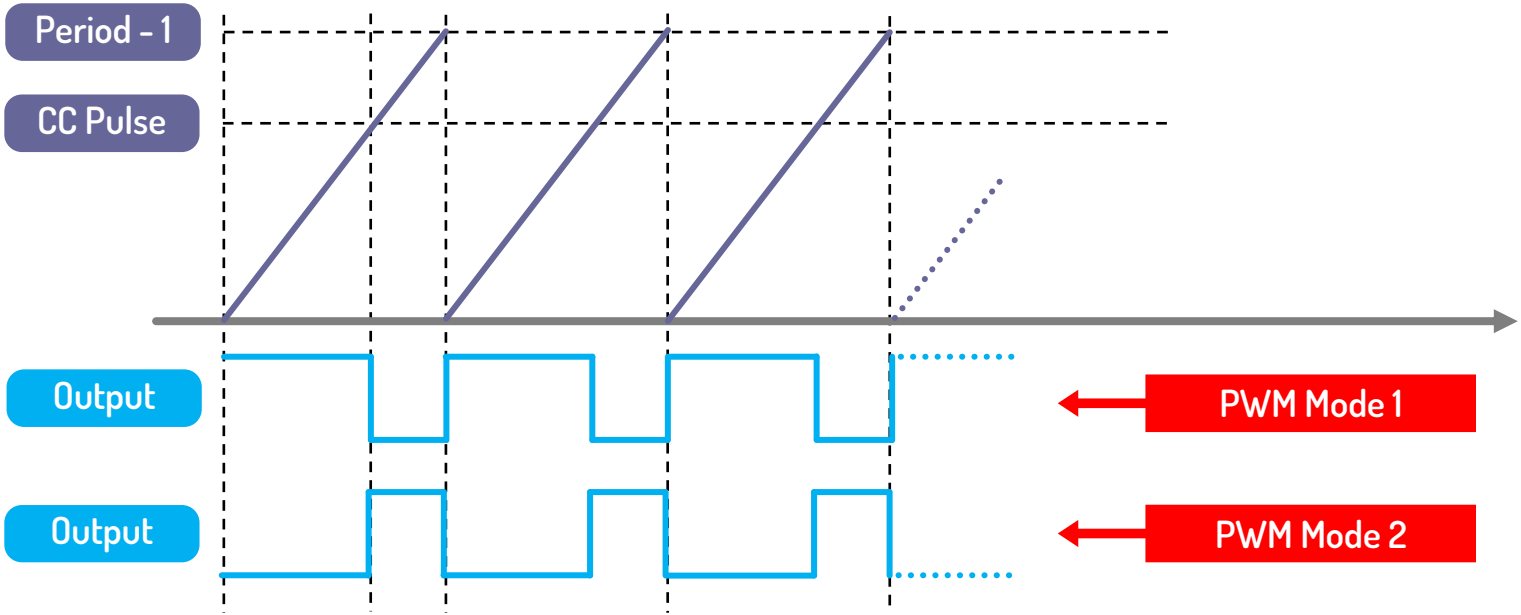
PWM period is set by the timer period (*Counter period* setting in Cube MX)

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.



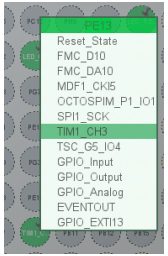
PWM duty cycle is set by the CC register value (*Pulse* setting in Cube MX)

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**.



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

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

Adjust clock and period settings

Set Mode and initial Pulse (duty cycle)

PWM Generation Channel 3

Mode	PWM mode 1
Pulse (16 bits value)	0
Output compare preload	Enable
Fast Mode	Disable
CH Polarity	High
CH Idle State	Reset

Practical #5a

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)

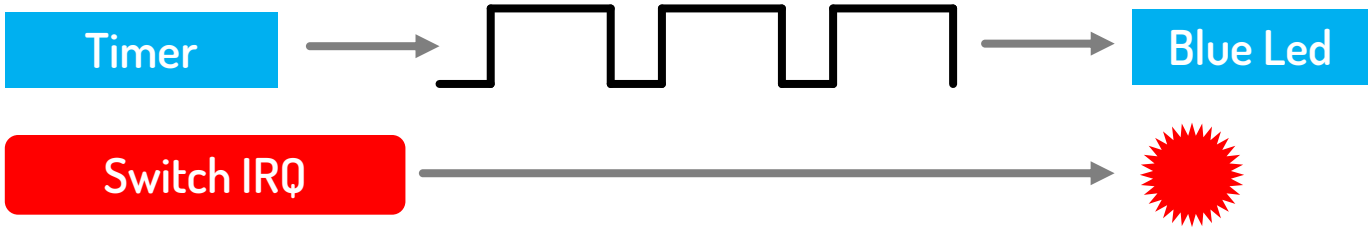
Practical #5b

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%



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 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

Deferred interrupts

Generally, on MCUs it is not good practice to :

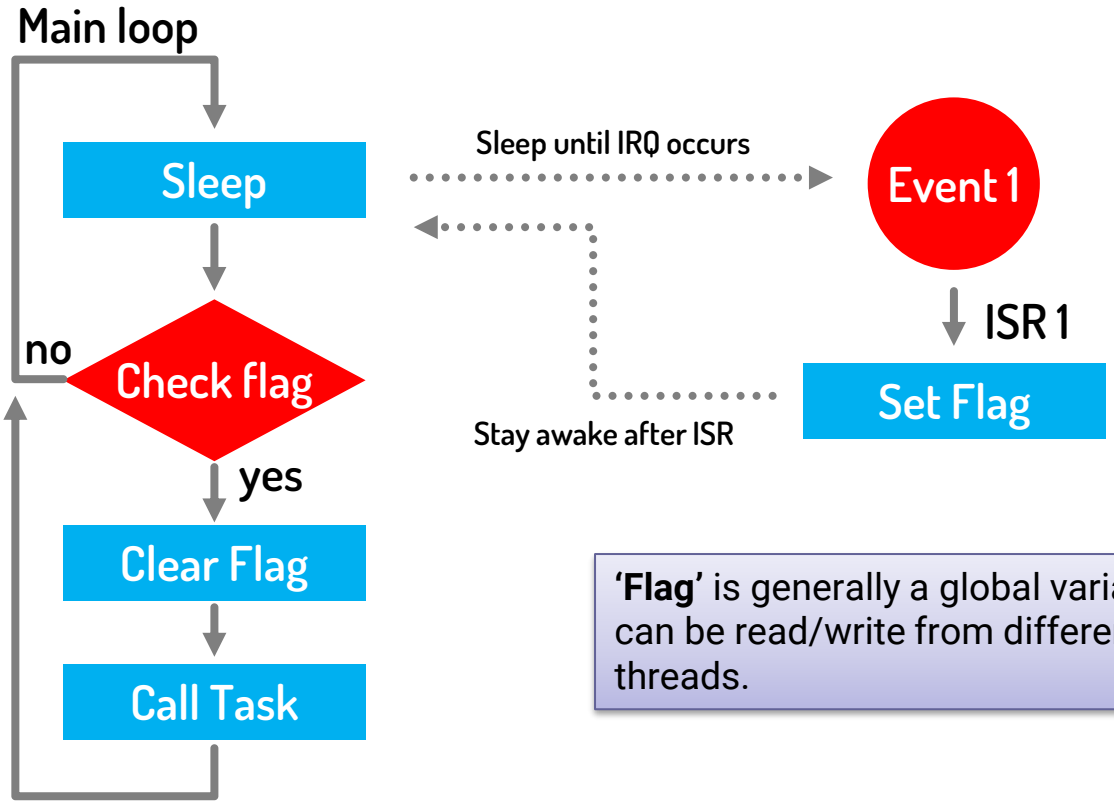
- Execute long code procedures within the ISR
- Calling external function from the ISR

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.

Code template

```c
/* 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 occured, 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;
        }
    }
}
```

```c
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.

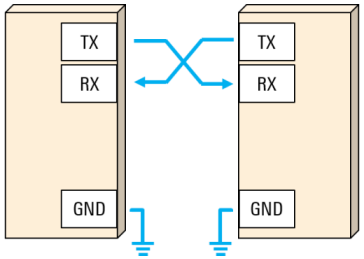
UART communication

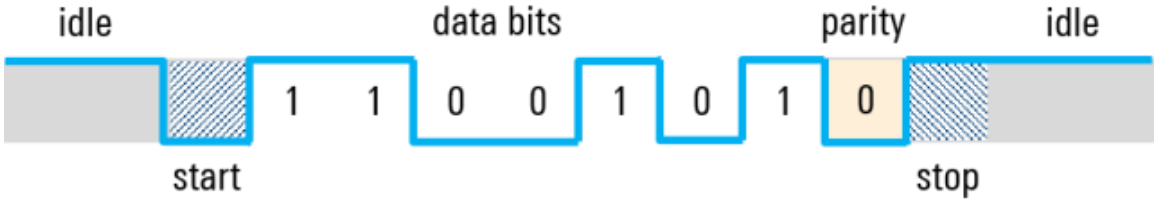
Communication with remote computer

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

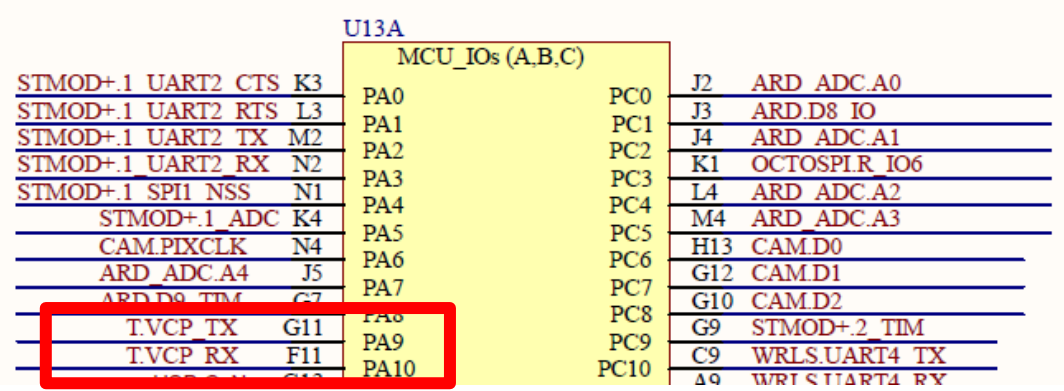




UART1 is connected to ST Link debug probe

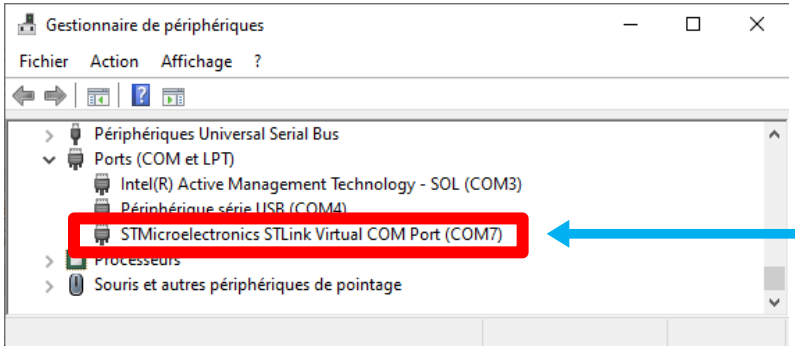
- **On-board** ST Link v3 is used both for programming / debugging
- It also has 2 GPIOs dedicated to UART communication between STM32U585 and computer

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



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

Check your peripheral manager (*gestionnaire de périphériques*)

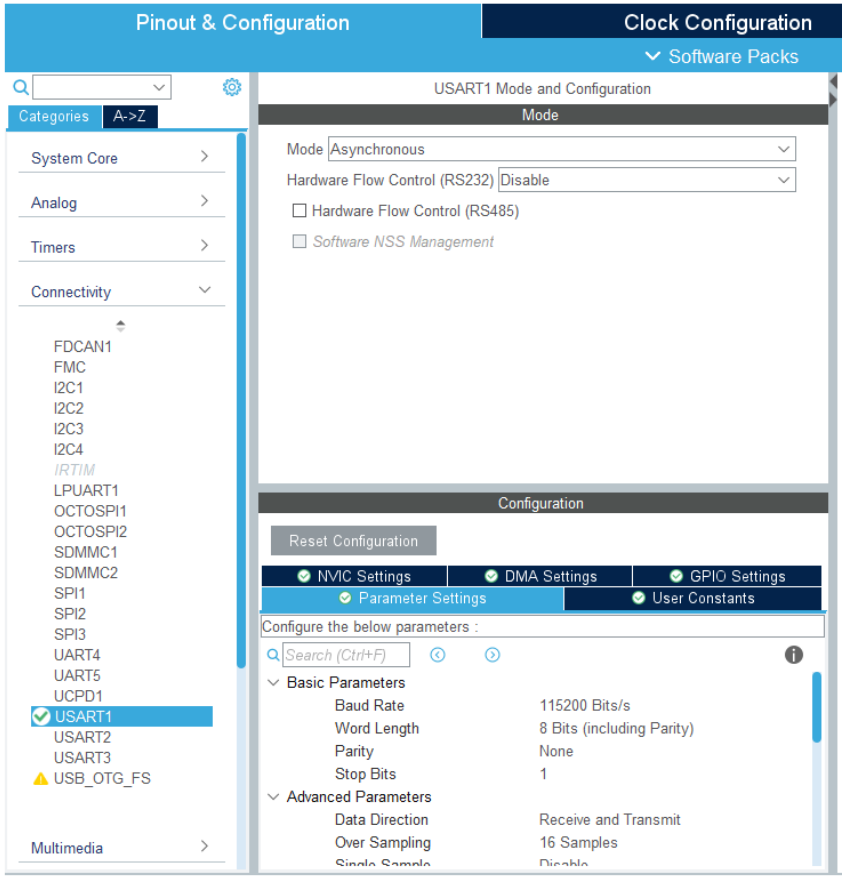


You can connect to this serial port by using any terminal such as:

- 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

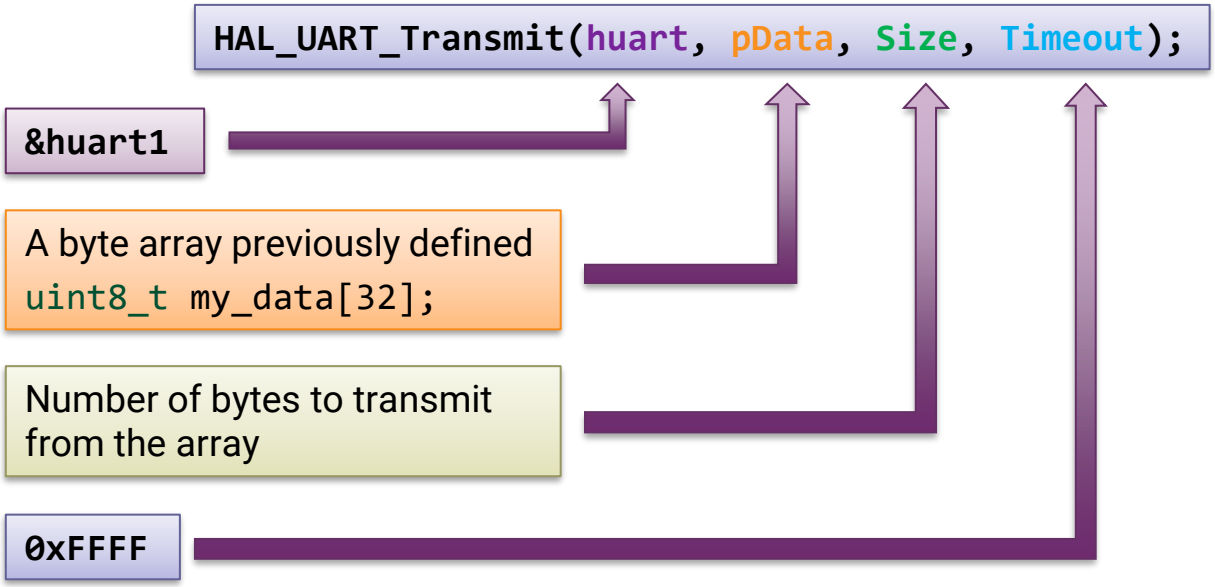


Configure PA9, PA10 and USART1 in Cube MX

USART1 is then initialized in main.c :

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

Send bytes arrays using transmit function :



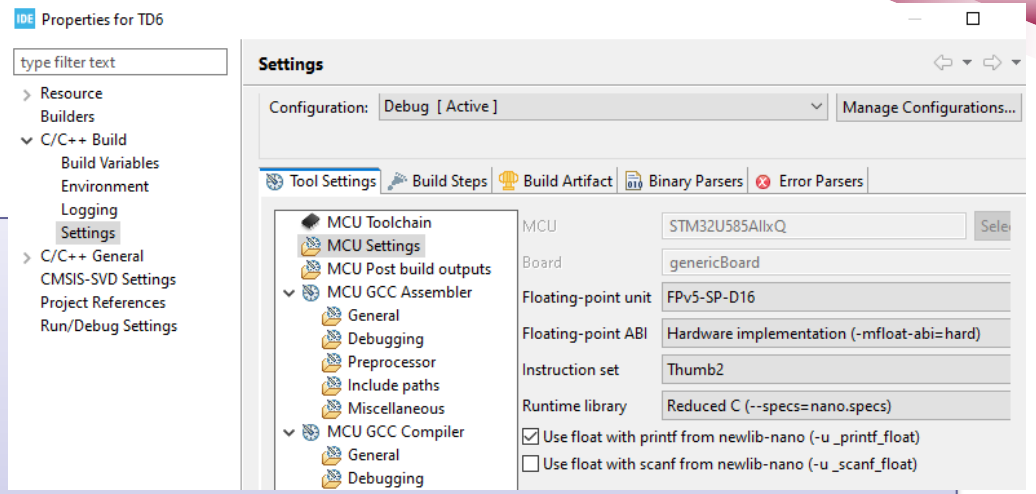
Send bytes arrays using transmit function :

```
// 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



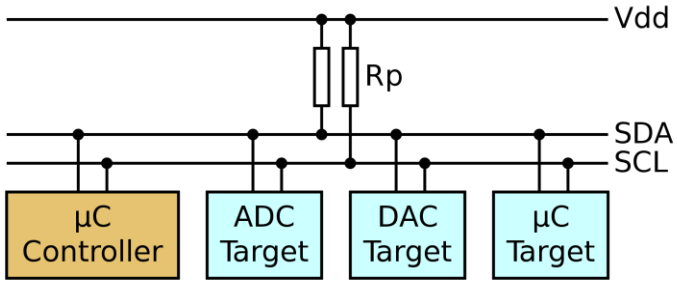

```
%d   // signed 8 or 16 bits integer
%ld  // signed 32 bits integer
%lld // signed 64 bits integer
%u   // unsigned 8 or 16 bits integer
%lu  // unsigned 32 bits integer
%llu // unsigned 64 bits integer
%[+][0][7][.3]f  // float (must be enabled project properties, see above)
The value 1,1352 would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

I2C communication

Communication with board peripherals

I2C Protocol :

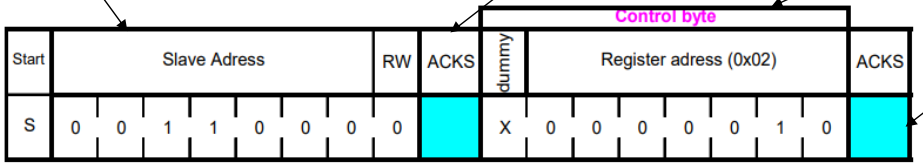
- Synchronous transmission
- Usual clocks: 100 kHz (Standard Mode), 400 kHz (Fast Mode)
- Half duplex (only one data line for emitter and receiver)



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

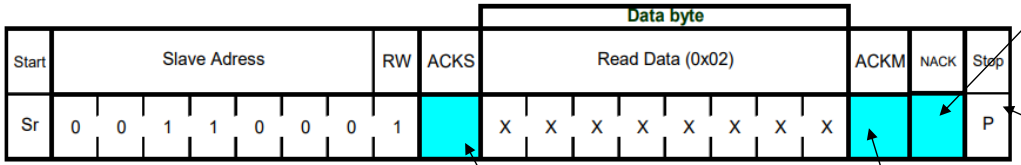
Slave ack

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



Slave ack

Master wishes to terminate the current transaction



End of transaction

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

Slave ack

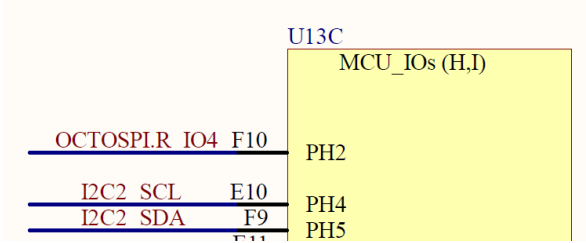
Master gets data

Master ack

Communication with board peripherals

I2C2 is used to communicate MEMS on board

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



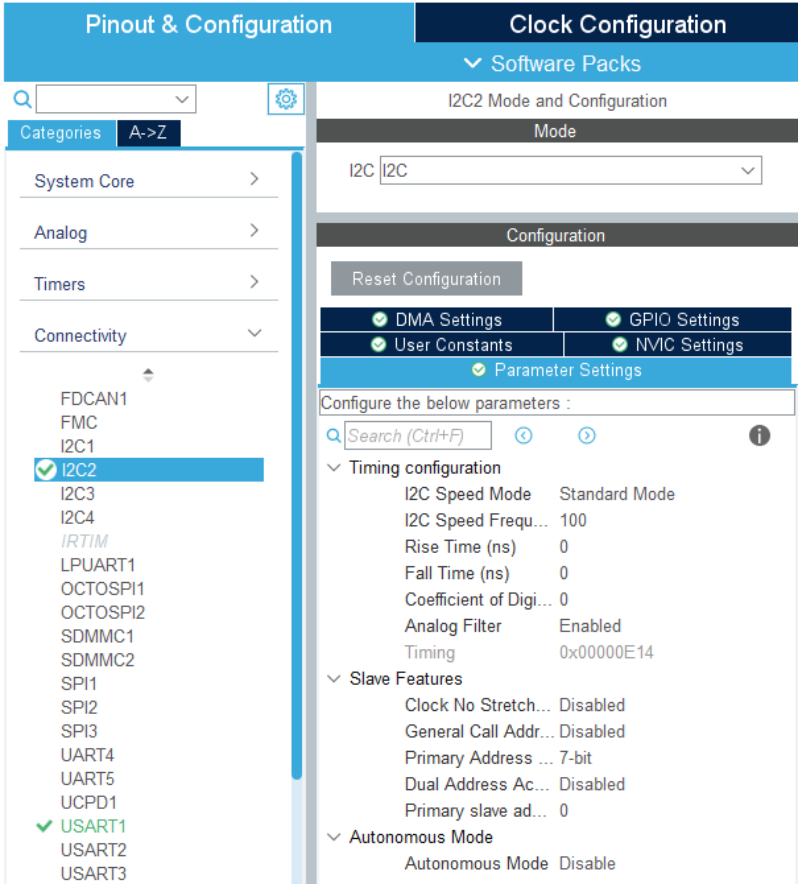
HTS221
Humidity & Temperature
8-bit addr. 0xBE

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



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;
uitn8_t addr = 0xBE;
HAL_I2C_Mem_Read(&hi2c2, addr, reg, 1, value, 2, 0xFFFF);
```

Practical #6

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 !**