جامعة بيرزيت
BIRZEIT UNIVERSITY

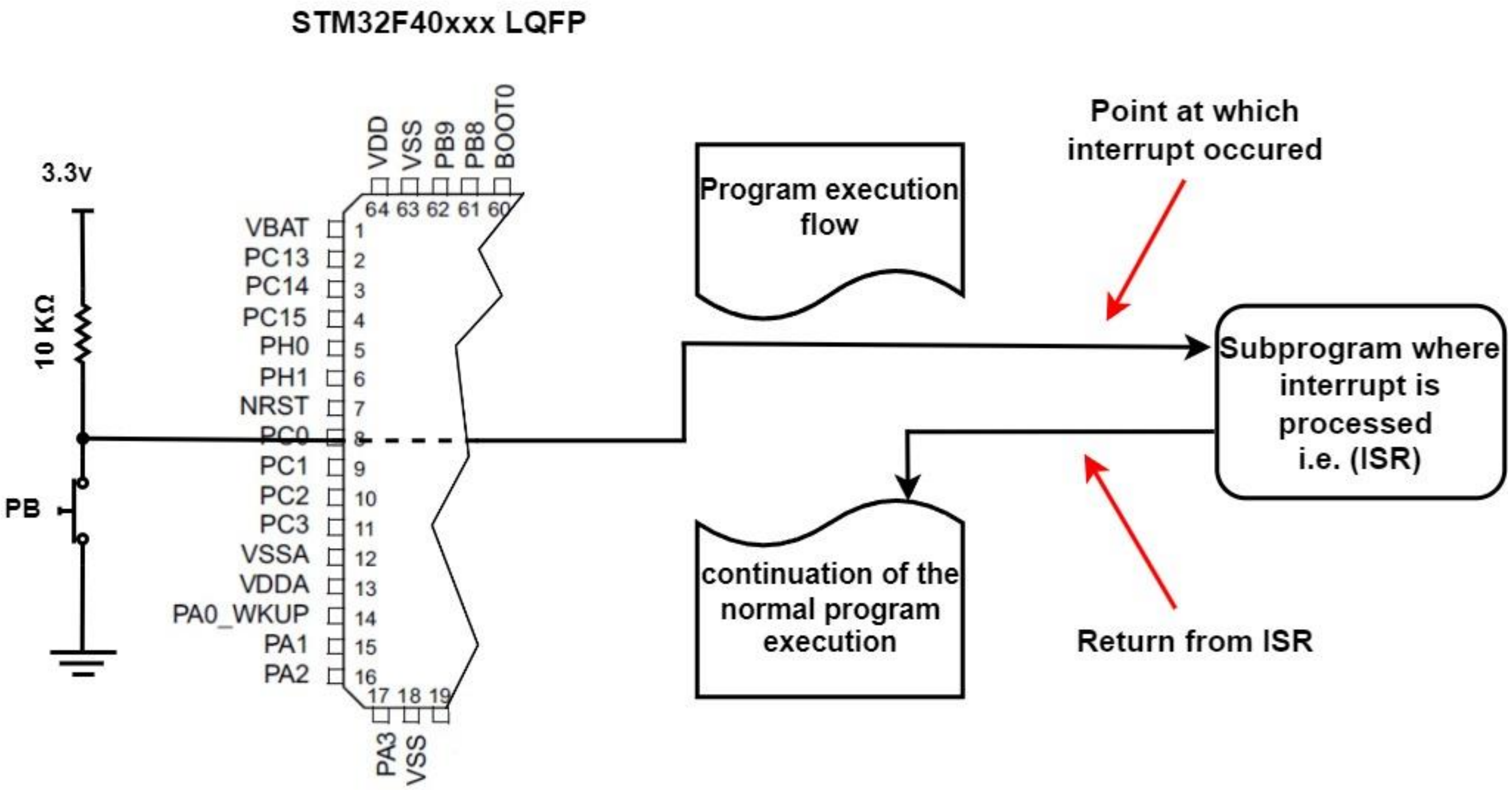
INTERRUPTS AND EVENTS

Shadi Daana

Interrupts

- Interrupt is a mechanism that allows the processor to temporarily suspend its current execution of instructions and switch to a different set of instructions in response to a specific event or condition
- Interrupts are a common feature available in almost all microcontrollers.
- Interrupts are usually generated from on-chip peripherals (e.g., a timer) or external inputs (e.g. a tactile switch connected to a GPIO), and in some cases they can be triggered by software

Interrupts



STM32F40xxx LQFP
3.3v
10 KΩ
PB
VDD
VSS
PB9
PB8
BOOT0
64 63 62 61 60
VBAT 1
PC13 2
PC14 3
PC15 4
PH0 5
PH1 6
NRST 7
PC0 8
PC1 9
PC2 10
PC3 11
VSSA 12
VDDA 13
PA0_WKUP 14
PA1 15
PA2 16
17 18 19
PA3
VSS
Program execution flow
Point at which interrupt occured
Subprogram where interrupt is processed i.e. (ISR)
continuation of the normal program execution
Return from ISR

Interrupts

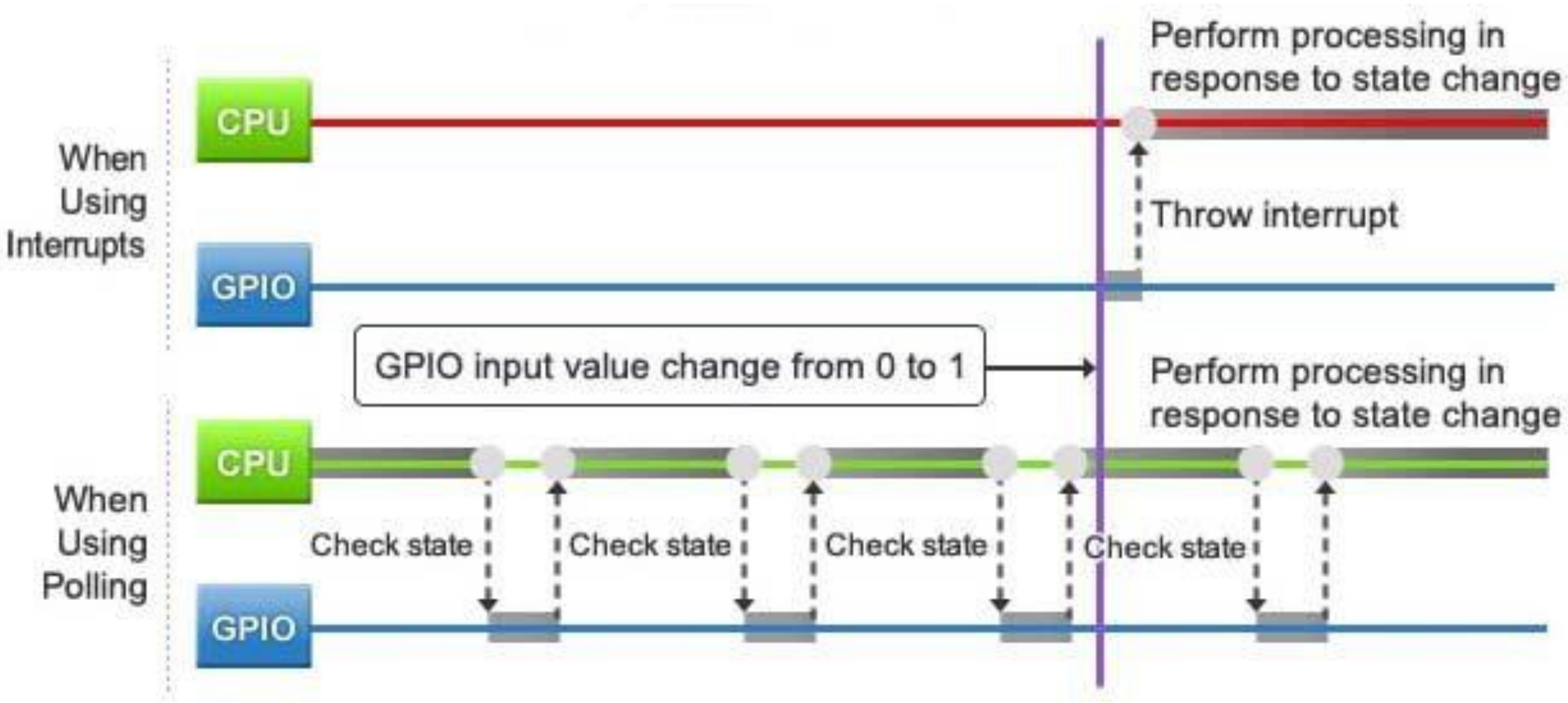
- When a peripheral or hardware needs service from the processor, typically the following sequence would occur:

1. The peripheral asserts an interrupt request to the processor
2. The processor suspends the execution of the current task, saves its context (that is, its stack pointer)
3. The processor starts the execution of an **Interrupt Service Routine (ISR)** to handle the interrupting event
4. The processor resumes the previously suspended task

Interrupts and Polling

- Interrupts and polling are two different approaches used in computer systems to handle external events, particularly in the context of input and output operations or event-driven processing.
- **Polling** is a technique where the program actively checks the status of a condition or device in a loop, while **interrupts** are events that prompt the program to respond immediately without actively checking.
- Polling can introduce delay and inefficiency, especially when dealing with infrequent events, while interrupts provide real-time responsiveness and efficient resource utilization.

Interrupts and Polling



Source: https://www.renesas.com/

Interrupts and Polling

```
…. . .
int main(void)
{
…. . .
  while (1)
  {
…. . .
  }
}
void xxx_IRQHandler(void )
{
    if(GPIO_Pin == GPIO_PIN_9) )
    {
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_8);
    }
}
```

```
…. . .
int main(void){
…. . .
  while (1){
   if(HAL_GPIO_ReadPin (GPIOA, GPIO_PIN_9)){
      HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_8, GPIO_PIN_SET);
   }
   …. . .
  }
}
```

- You can program your **interrupt handlers** or **Interrupt Service Routines (ISR)** as normal C routines/functions.
- The ISR **is automatically called** when an interrupt event occurs. It is not called from the main function or any other functions!!

Interrupt Masking

- Cortex-M processors have different numbers of interrupt sources including GPIO pins, timers, resets, and more.
- In most microcontrollers, there are often **interrupt mask registers** associated with each interrupt source.
- These registers allow you to **enable** or **disable** specific interrupts at the **hardware level**
- When an interrupt is **masked (disabled)**, it won't be recognized or serviced by the processor, even if the corresponding event occurs
- You need to **enable (unmask)** the interrupt before it can be used.

Interrupt Masking

- Some interrupts **cannot be disabled** by software; these are referred to as **Non-Maskable Interrupts (NMIs)**
- These are high-priority interrupts that are used for critical events like hardware faults, system resets, and emergencies that need immediate attention

Interrupt Priority

- Most of interrupts have **programmable priorities**, and a few system interrupts have **fixed priority**
- **What do we mean by the priority of an interrupt?**
- The **priority of an interrupt** determines the order in which interrupts are handled when multiple interrupts occur at the same time or when an interrupt occurs while another one is already being processed.
- Each programmable interrupt can be assigned a priority level, where a value of '0' represents the highest priority

Interrupt Types in Cortex M

- Cortex-M interrupts can fall into the following categories:
  - **IRQs (Interrupt Requests):** generated by peripherals such as timers, I/O ports, and communication interfaces (e.g., UART, I2C).
  - **Non-Maskable Interrupt (NMI):** generated from internal peripherals (e.g. watchdog timer).
  - **SysTick (System Tick**) timer interrupt.
  - **System exceptions.**

Interrupt Types in Cortex M

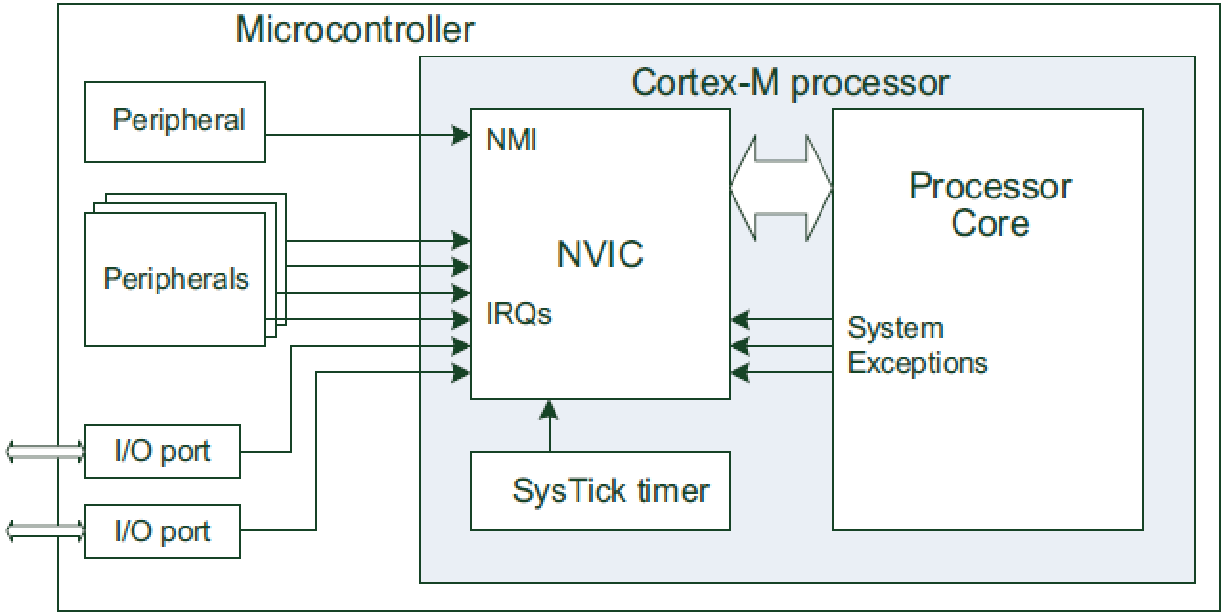
Table D.1 Quick Summary of Cortex®-M3/M4 Exception Types and their Configurations

Exception Type	Name	CMSIS-Core Exception Enum	Priority (Level Address)	Enable
1	Reset	–	−3	Always
2	NMI	NonMaskableInt_IRQn	−2	Always
3	Hard fault	HardFault_IRQn	−1	Always
4	MemManage	MemoryManagement_IRQn	Programmable (SCB->SHP[0], 0xE000ED18)	SCB->SHCSR (0xE000ED24) bit[16]
5	BusFault	BusFault_IRQn	Programmable (SCB->SHP[1], 0xE000ED19)	SCB->SHCSR (0xE000ED24) bit[17]
6	Usage fault	UsageFault_IRQn	Programmable (SCB->SHP[2], 0xE000ED1A)	SCB->SHCSR (0xE000ED24) bit[18]
7–10	–	–	–	–
11	SVC	SVCall_IRQn	Programmable (SCB->SHP[7], 0xE000ED1F)	Always
12	Debug monitor	DebugMonitor_IRQn	Programmable (SCB->SHP[8], 0xE000ED20)	CoreDebug->DEMCR (0xE000EDFC) bit[16]
13	–	–	–	–
14	PendSV	PendSV_IRQn	Programmable (SCB->SHP[10], 0xE000ED22)	Always
15	SysTick	SysTick_IRQn	Programmable (SCB->SHP[1], 0xE000ED23)	SysTick->CTRL (0xE000E010) bit[1]
16–255	IRQ	(device specific)	Programmable (NVIC->IP[*n*], 0xE000E400)	NVIC->ISER[] (0xE000E100)

Source: Yiu, Joseph. The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors. Germany: Elsevier Science, 2013.

Nested Vectored Interrupt Controller (NVIC)

- How does the processor manage interrupts from different sources ?



Source: Yiu, Joseph. The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors. Germany: Elsevier Science, 2013.

- Inside Cortex processors, there is a hardware component called **Nested Vectored Interrupt Controller (NVIC)** that is responsible for managing and prioritizing interrupt requests from various sources

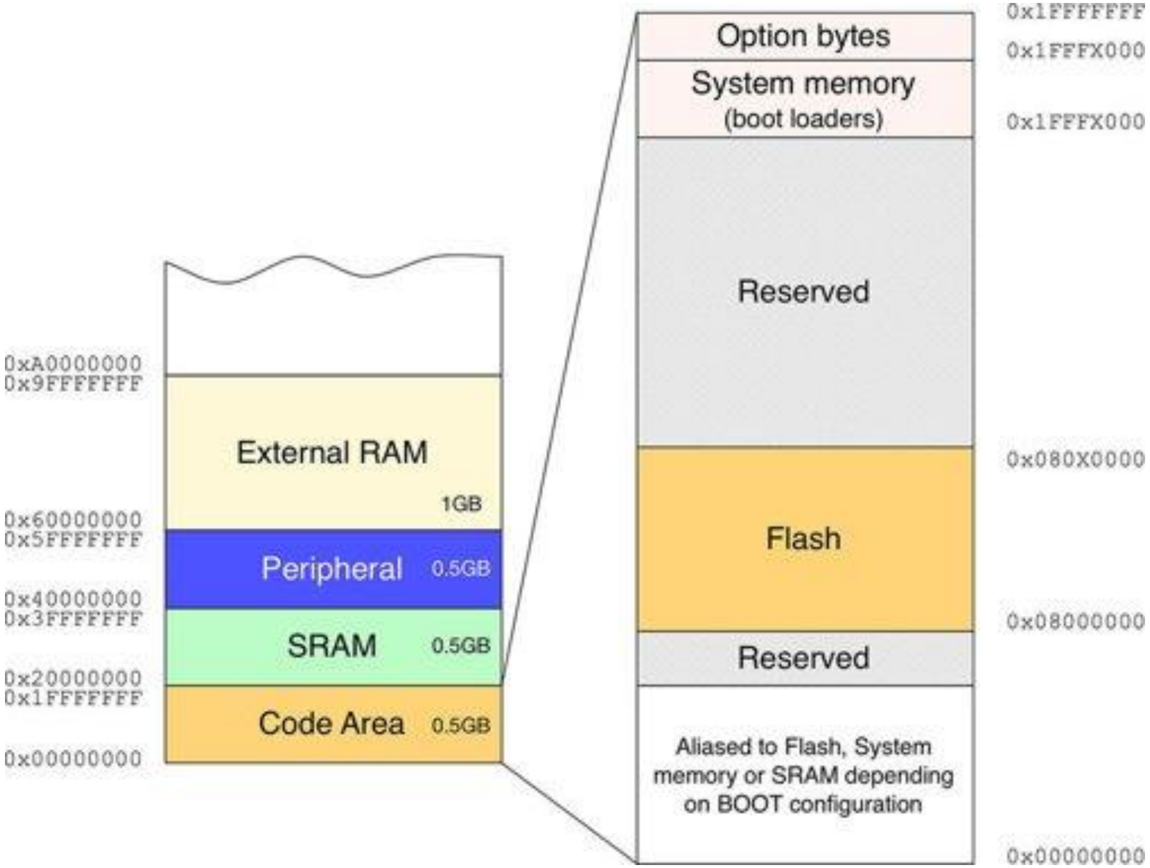
Nested Vectored Interrupt Controller (NVIC)

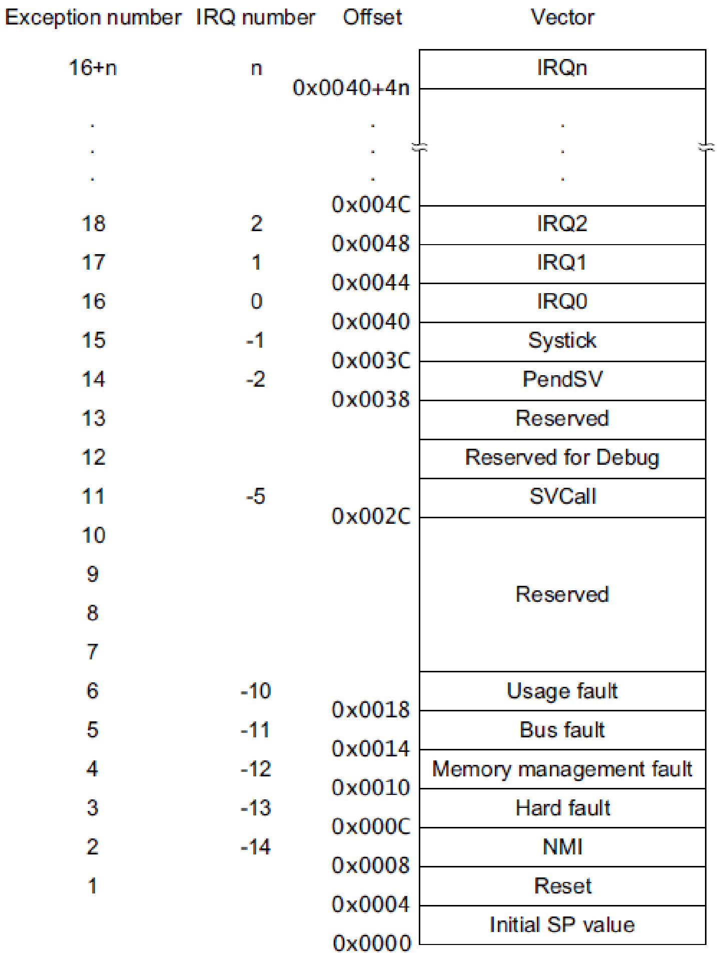
- The NVIC controls how interrupts are handled, their priorities, and the order in which they are executed
- The term "**nested**" reflects the ability to manage interrupts in a layered manner
- The term “**vector**” in nested vector interrupt control refers to the way in which the CPU finds the program, or ISR, to be executed when an interrupt occurs.

Vector Table

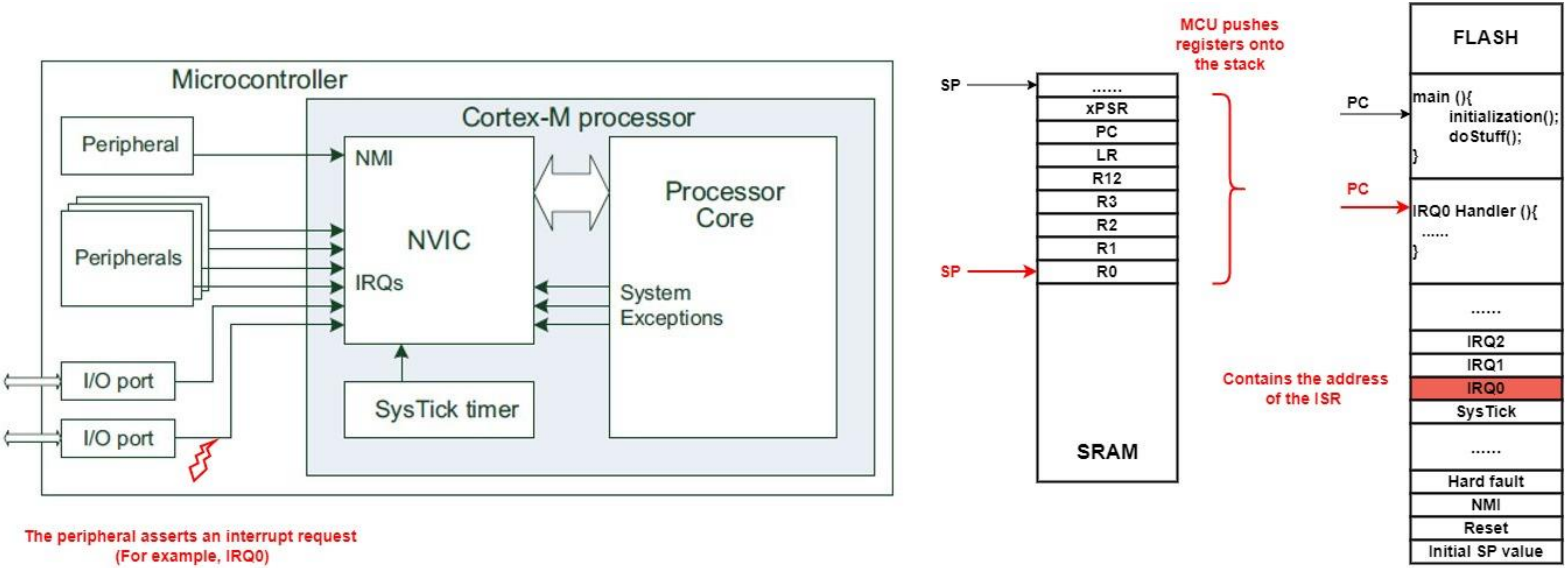
- In order to understand how interrupts are managed, let’s take a review of the memory map of ARM cortex M4 microcontrollers.
- The address space is 4 GB wide.
- It is organized into several subregions such as code, SRAM, peripherals, external memory, system memory, etc.
- The code memory region contains an **interrupt vector table (IVT)**.
- When an interrupt occurs, the processor looks up the address of the corresponding ISR in the vector table and jumps to that address to execute the interrupt service routine

Vector Table





What Happens When an Interrupt Occurs?



What Happens When an Interrupt Occurs?

- The processor finishes or terminates the current instruction
- Microcontroller pushes registers R0, R1, R2, R3, R12, LR, Program counter (PC), and program status register (PSR) onto the stack
- The processor reads the interrupt number from the xPSR register and finds the corresponding address of the ISR from the **vector table**.
- The processor then starts to execute ISR instructions and uses the registers that were previously pushed to stack
- The last step is exiting the ISR, popping all eight registers from the stack, and returning to the same instruction where the ISR occurs
- *Note: The interrupt service routines or exception handlers in ARM Cortex-M4 microcontrollers do not use R4-R11 registers during ISR execution*

Stacking

ISR Execution

Return

Writing Your ISR Implementation

- You don’t need to manage the low-level interrupt handling process directly.
- You can focus on implementing your own ISR logic for specific actions.
- Usually, the HAL library (or other frameworks) takes care of the underlying details.
- The library also provides the necessary ISR definitions you can use.

Writing Your ISR Implementation

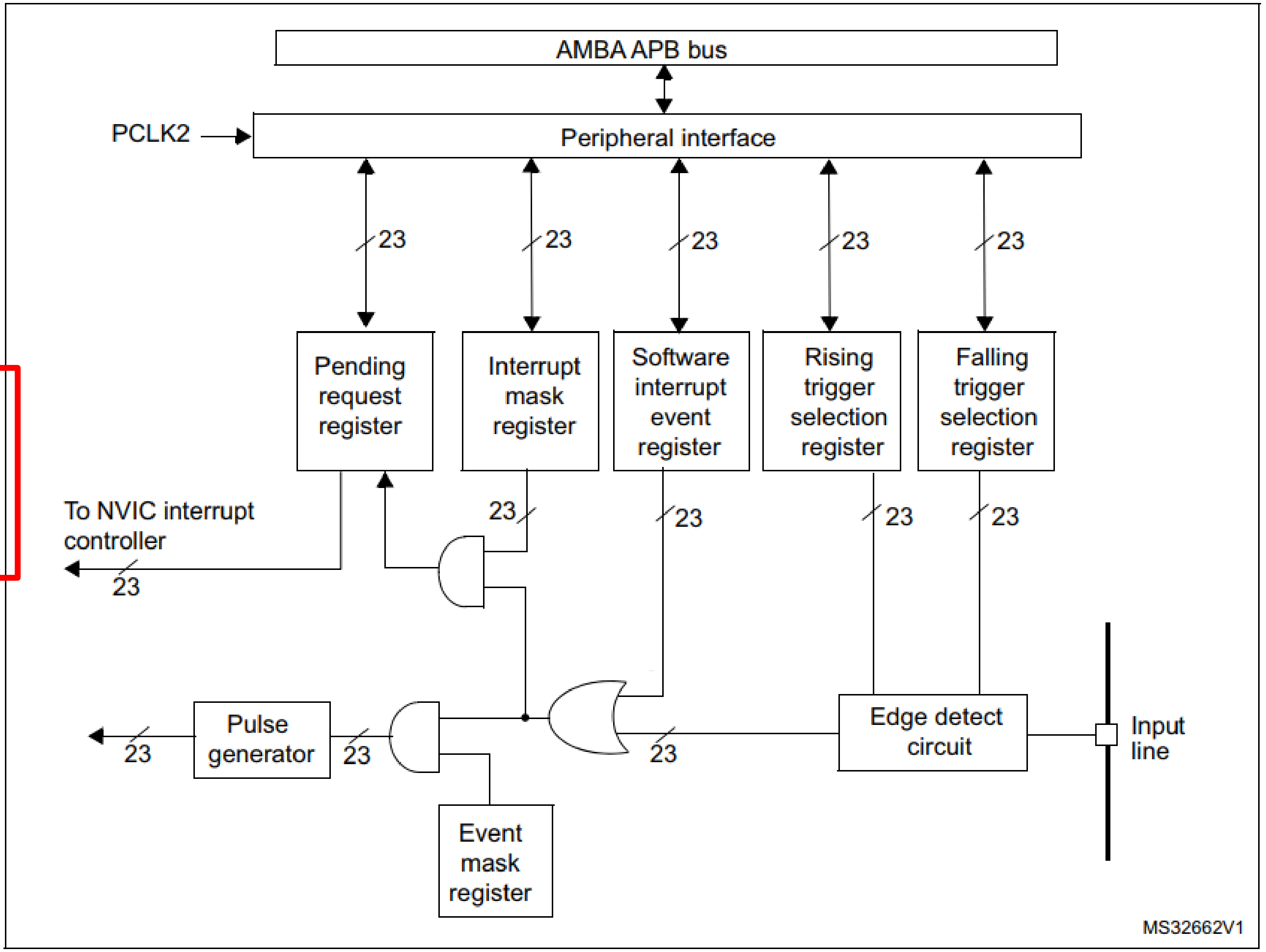
- Before diving into ISR implementation, it's important to understand how the STM32 handles interrupts from **external sources**.
- Inside the microcontroller, each input line is connected to special circuit that is designed to **detect changes** in input signals and **handle external interrupts and events**.
- This circuit is called the **External Interrupt/Event Controller (EXTI).**

Writing Your ISR Implementation

- When we say EXTI detects changes in input signals, what does it actually detect?
  - It monitors **signal transitions** on its connected GPIO pins.
    - **Rising Edge**: The signal changes from **low (0)** to **high (1)**.
    - **Falling Edge**: The signal changes from high **(1)** to low **(0)**.
    - **Both Edges**: The signal changes in either direction.
- The EXTI allows you to configure which type of edge it should detect for each interrupt source.
- When the configured edge is detected (button presses, sensor signals, etc.), the EXTI generates an **interrupt**.

External Interrupt/Event Controller (EXTI)

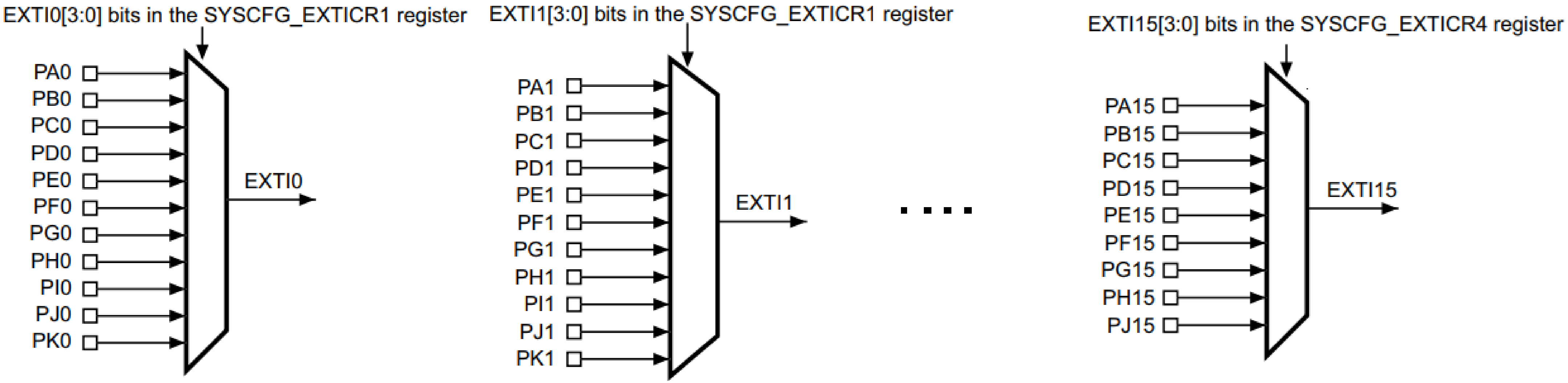
Figure 41. External interrupt/event controller block diagram



EXTI just tells the NVIC that an interrupt has occurred on line x

EXTI consists of up to 23 edge detectors for generating event/interrupt requests

External Interrupt/Event Controller (EXTI)



The seven other EXTI lines are connected as follows:

- EXTI line 16 is connected to the PVD output
- EXTI line 17 is connected to the RTC Alarm event
- EXTI line 18 is connected to the USB OTG FS Wakeup event
- EXTI line 19 is connected to the Ethernet Wakeup event
- EXTI line 20 is connected to the USB OTG HS (configured in FS) Wakeup event
- EXTI line 21 is connected to the RTC Tamper and TimeStamp events
- EXTI line 22 is connected to the RTC Wakeup event

Interrupt Handling Using STM32 HAL

- STM32's HAL driver provides an **interrupt handler function** for each peripheral
- (e.g.,`EXTIx_IRQHandler` for external inputs, `TIMx_IRQHandler` for timers, `USARTx_IRQHandler` for UART, etc.).
- These handlers are predefined by the HAL and **automatically call** and execute when the corresponding interrupt occurs.
- Within these interrupt handlers, the HAL **<u>usually</u>** includes calls to specific **callback functions** that the user can **override.**

Callback Functions

- For example, in the STM32Cube framework `EXTIx_IRQHandler()` calls `HAL_GPIO_EXTI_IRQHandler()` which calls `HAL_GPIO_EXTI_Callback(GPIO_Pin)`
- `HAL_GPIO_EXTI_Callback()` has "weak linkage", meaning you can **override** by defining your own version.

stm32f4xx_hal_gpio.c

```
/**
  * @brief  EXTI line detection callbacks.
  * @param  GPIO_Pin Specifies the pins connected EXTI line
  * @retval None
  */
__weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  /* Prevent unused argument(s) compilation warning */
  UNUSED(GPIO_Pin);
  /* NOTE: This function Should not be modified, when the callback is needed,
           the HAL_GPIO_EXTI_Callback could be implemented in the user file
   */
}

/**
  * @}
  */
```

Overriding Interrupt Callback Functions

- Override means you include your own implementation of a function to replace the default behavior provided by a library

```c
int main(void)
{
…...
  while (1)
  {
…...
  }
}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin){

// Your code here, for example, toggle an LED or perform any other action.
  if (GPIO_Pin==GPIO_PIN_0) {
    HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_12);
  }
}
```

- There is no explicit call to `HAL_GPIO_EXTI_Callback` in the main function

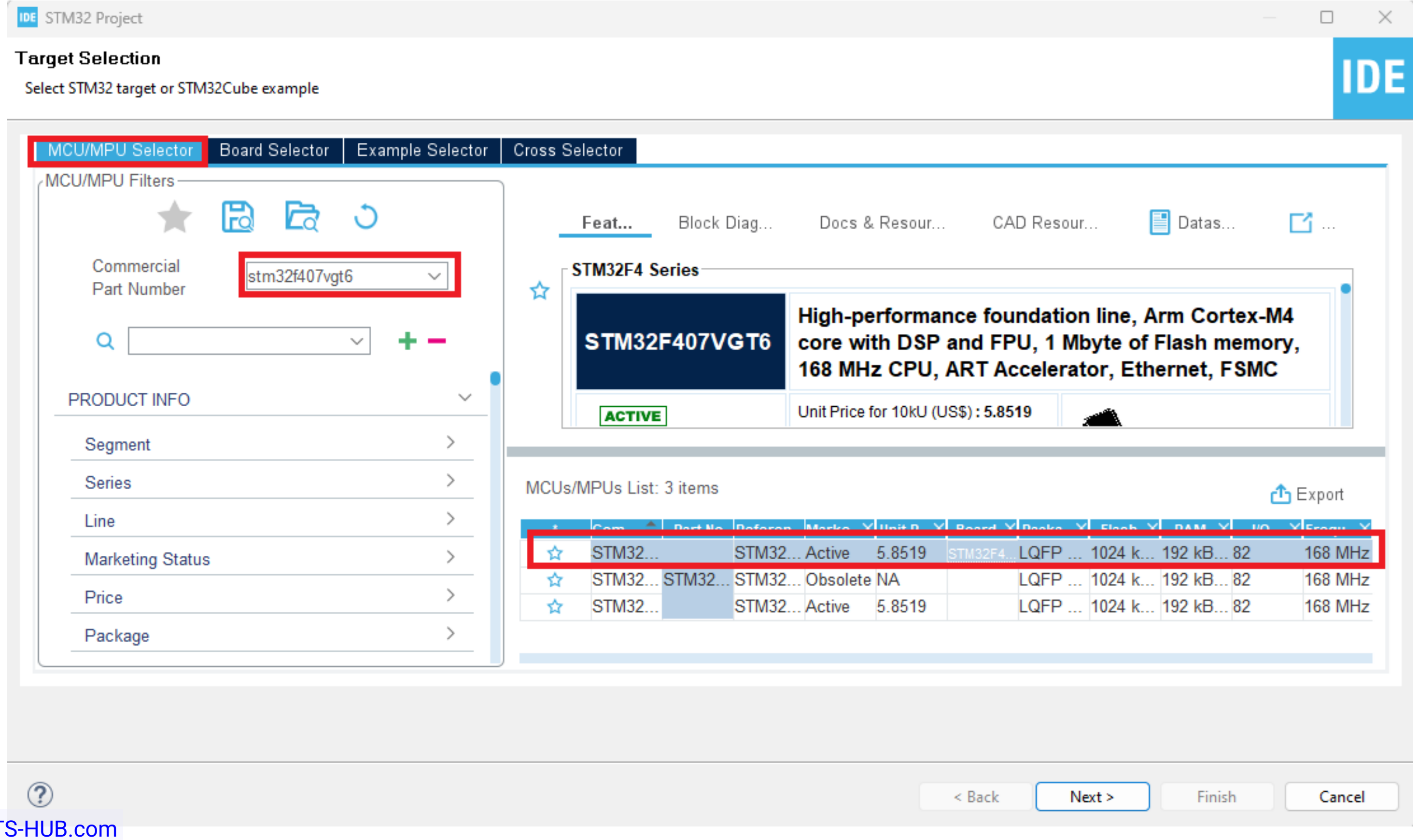
- Whenever an interrupt occurs, the callback function is executed **automatically**
- The user implementation defines the actions that will take place inside the function
- Here, whenever an interrupt detected on the EXTI line 0, the pin PD12 will toggle

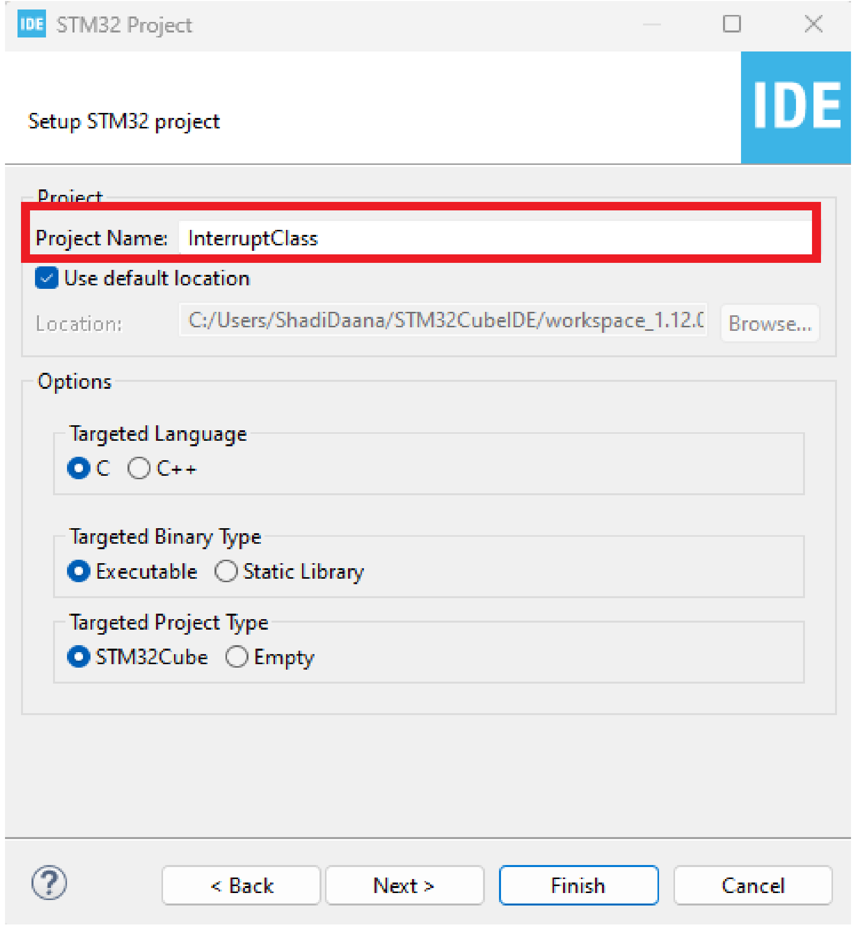
Key Considerations for Interrupt Configuration

- You should enable (unmask) the interrupt before using it.
- You need to define the interrupt priority (highest priority is 0).
- The priority of each interrupt should be unique
- The ISR should be kept as simple and brief as possible (Do not write long code inside the ISR).
- The global variables that are modified by interrupts should be **`volatile`**

Example: Configure EXTI to toggle a LED when a user button is pressed

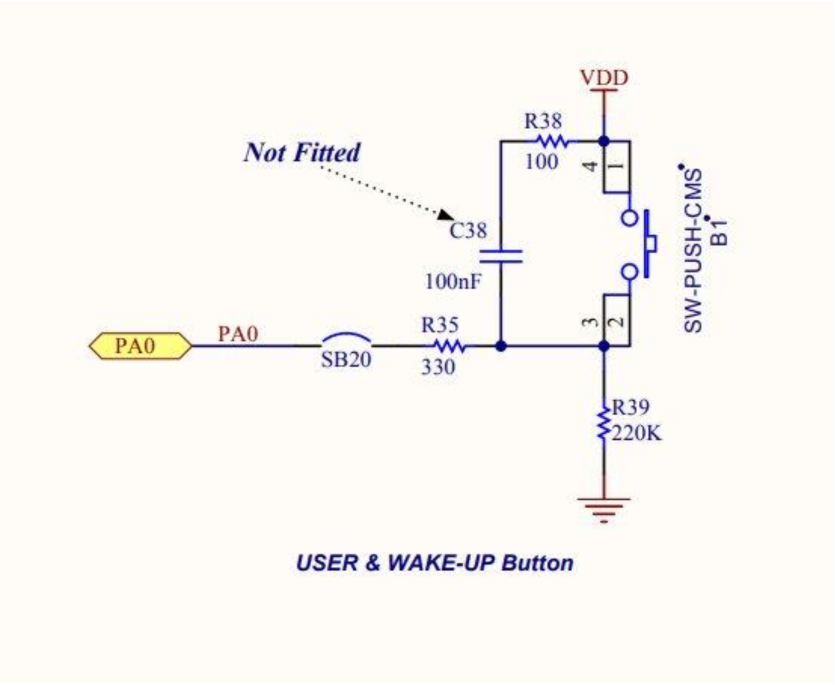
- Create a new STM32CubeIDE project

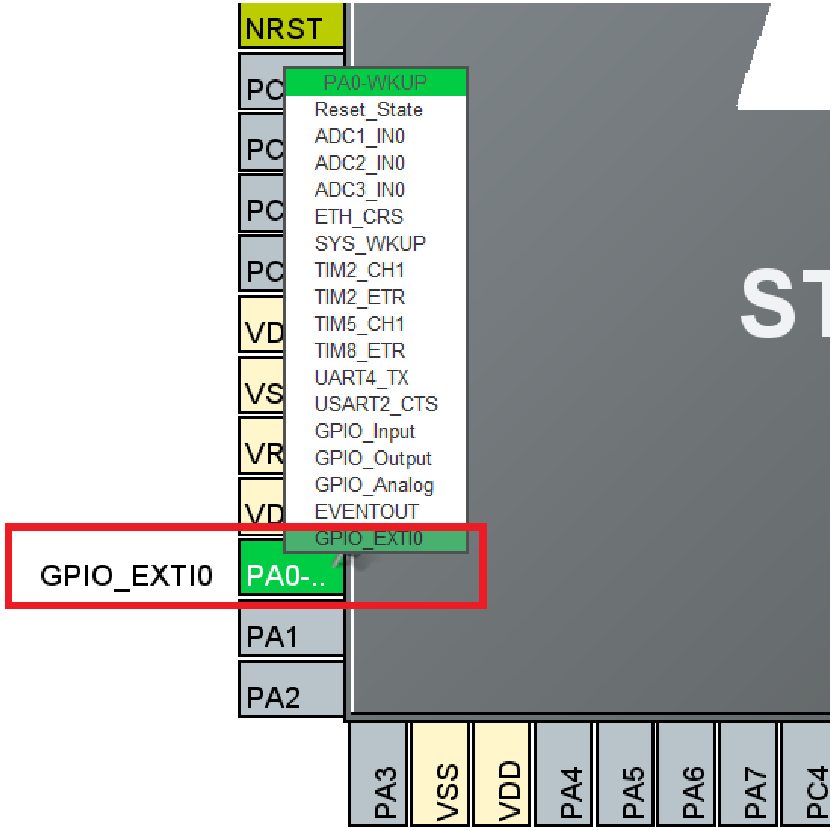




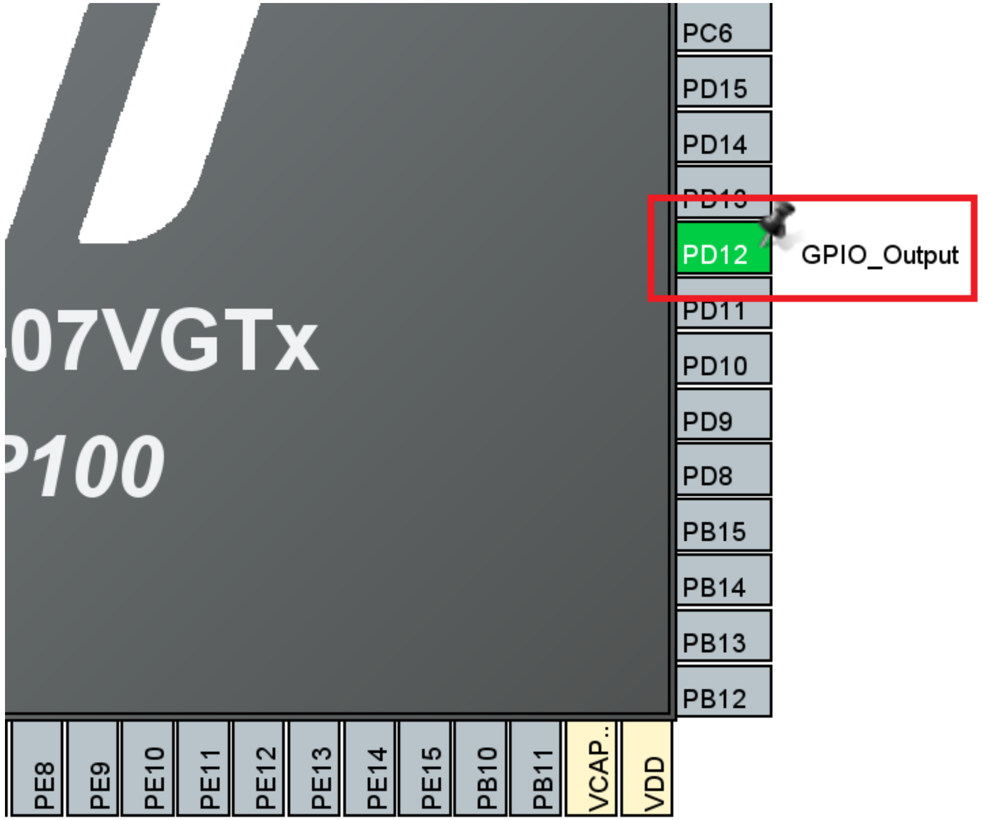
Example: Continued

- Configure a button pin (PA0) as GPIO_EXTI

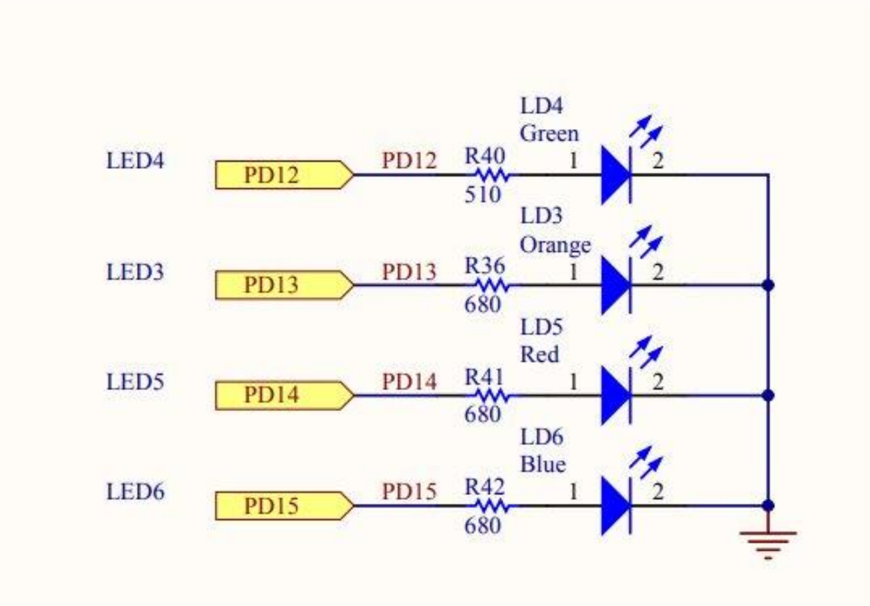




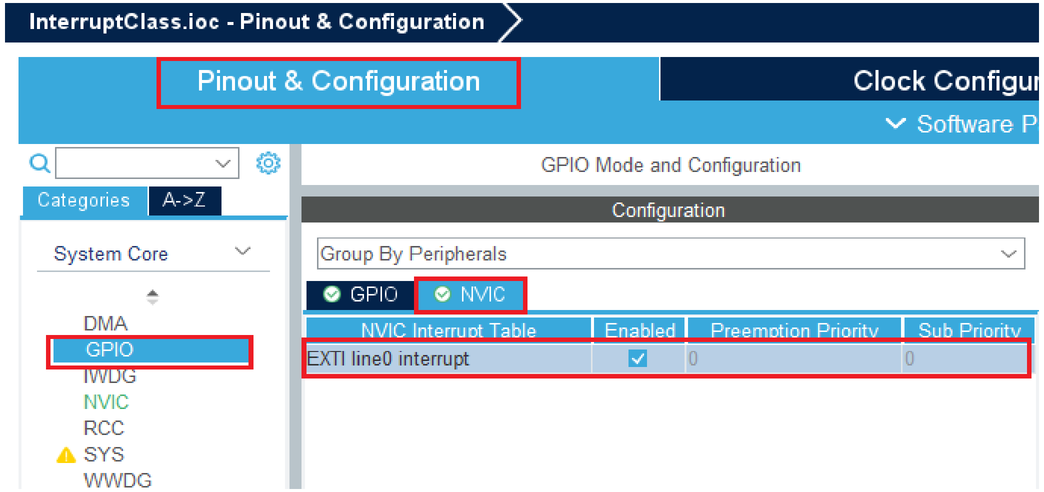
Example: Continued



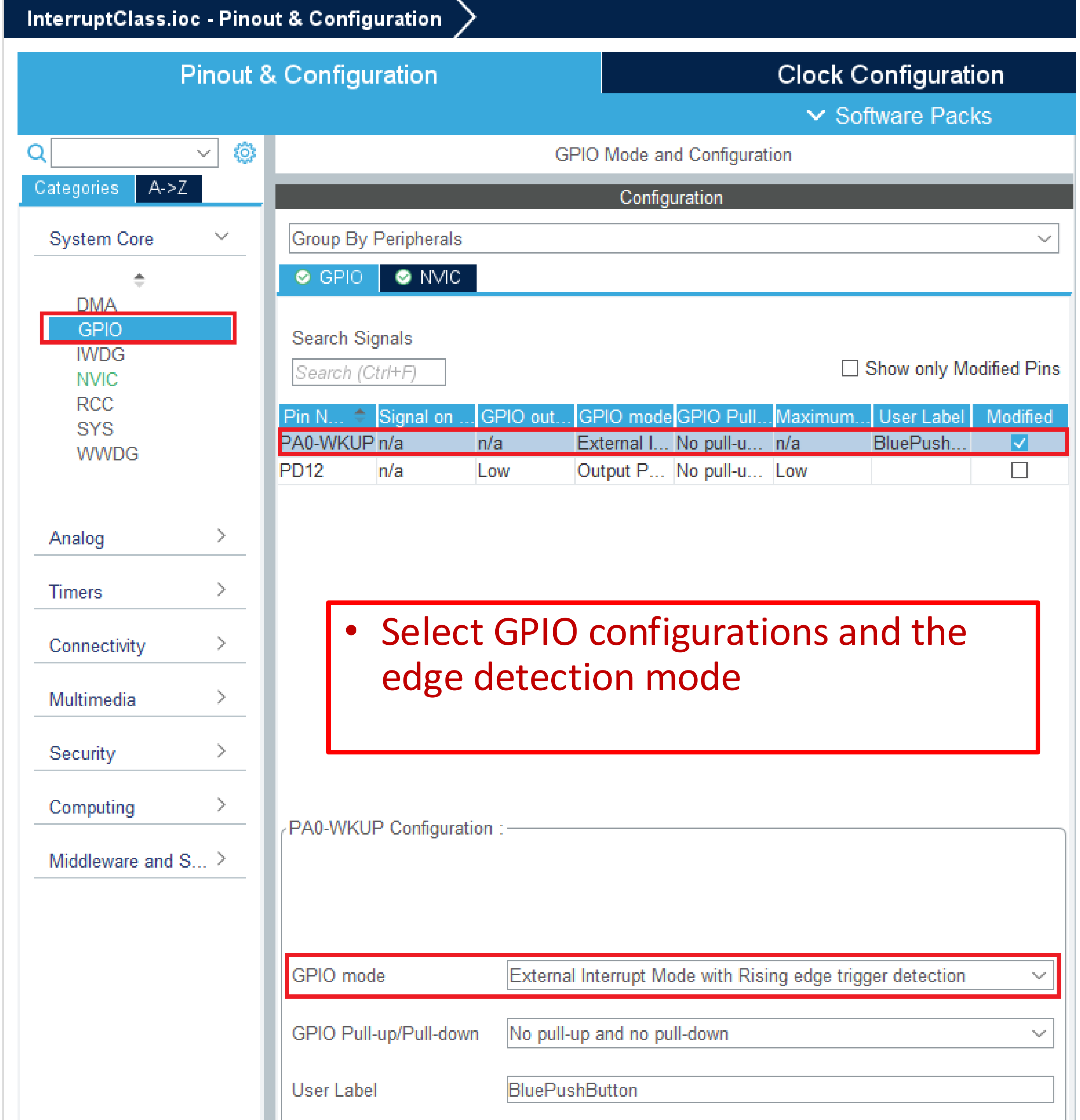
- Configure the LED pin (PD12) as GPIO_Output



Example: Continued



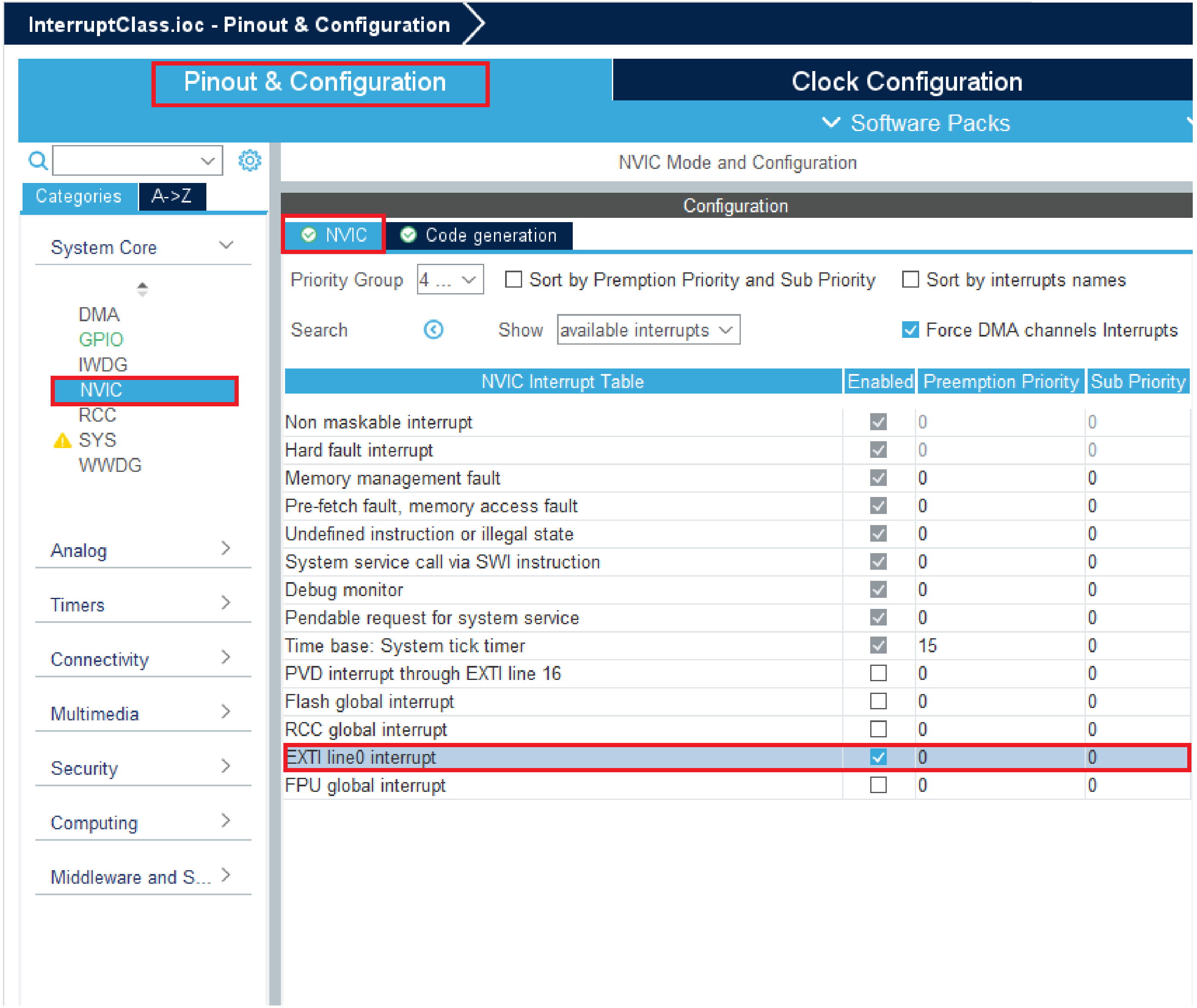
- Enable the interrupt for EXTI!!



- Select GPIO configurations and the edge detection mode

Example: Continued

- Set the priority of the interrupt
- If you have multiple interrupt sources, each must have a unique priority number
- Generate the code



Example: Program Code

```
…. . .
int main(void)
{
…. . .
  while (1)
  {
…. . .
  }
}
…. . .
/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    if (GPIO_Pin == BluePushButton_Pin){
        HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_12);
    }
}
/* USER CODE END 4 */
…. . .
```

Example: Practical Version

```
…...
/* USER CODE BEGIN PD */
#define DEBOUNCE_TIME 100
/* USER CODE END PD */
…...
/* USER CODE BEGIN PV */
volatile uint32_t currentTick=0;
volatile uint32_t previousTick=0;
volatile uint8_t toggleFlag=0;
/* USER CODE END PV */

int main(void)
{
…...
  while (1)
  {
    if (toggleFlag==1){
        HAL_GPIO_TogglePin(GPIOD, GPIO_PIN_12);
        toggleFlag=0;
    }
  }
}
…...
```

If the button press is valid (meaning it passed the debounce check), `toggleFlag` is set to 1

Gets the current system tick count in milliseconds (each time the interrupt occurs, we record the time)

```
…...

/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin) {
    if (GPIO_Pin == BluePushButton_Pin) {
        currentTick = HAL_GetTick();
        if ((currentTick - previousTick) > DEBOUNCE_TIME)
{
          toggleFlag=1;
        }
        previousTick = currentTick;
    }
}
/* USER CODE END 4 */

…...
```

This code handled the bouncing, it won't consider any changes for some period of time (DEBOUNCE_TIME)

HAL interrupt handler and callback functions

- Besides the APIs, HAL peripheral drivers include:
  - HAL_PPP_IRQHandler() peripheral interrupt handler that should be called from stm32f4xx_it.c
  - User callback functions.
- The user callback functions are defined as empty functions with “weak” attribute. They have to be defined in the user code.

```
HAL_PPP_EventTypeCallback
```

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin);
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart);
void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart);
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim);
void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc);
```

Questions?

References

- Yiu, Joseph. The Definitive Guide to ARM® Cortex®-M3 and Cortex®-M4 Processors. Germany: Elsevier Science, 2013.
- Motion Control Tips. (n.d.). What is nested vector interrupt control (NVIC)? Retrieved from https://www.motioncontroltips.com/what-is-nested-vector-interrupt-control-nvic/
- Renesas. (n.d.). MCU programming: Peripherals - 04. Interrupts. Retrieved from https://www.renesas.com/en/support/engineer-school/mcu-programming-peripherals-04-interrupts?srsltid=AfmBOooaPIEDO5nE4bsR0lUf9D5JhGniBHTyLNPy6sjqHIOky3bWW1En