Session 3/4

Microprocessors course
Session 3: Interrupt concept

Version STM32L476

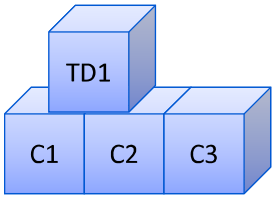
(microcontroller)

2024-v1

INTERRUPTS 1

ENSEA S7-2A

Laurent MONCHAL

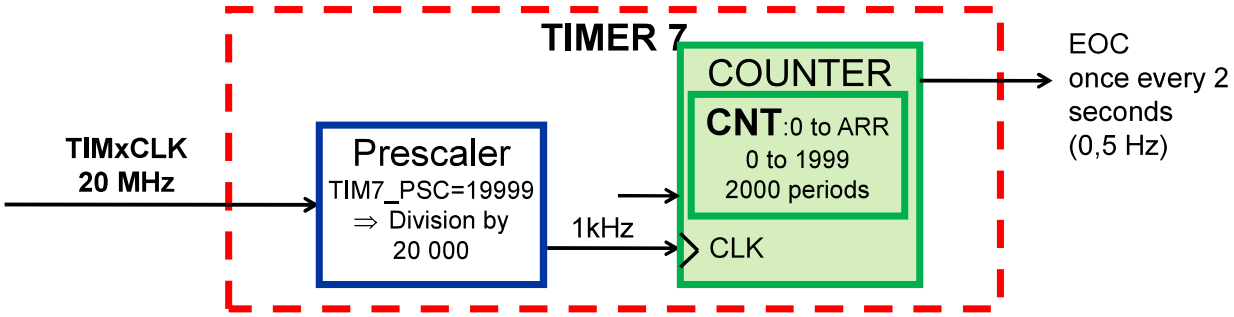


Session Menu

- Previously seen: architecture and peripherals of a STM32F4xx
- The sources of **INTERRUPTS in** the STM32
- Examples:
  - Implementation of LED blinking **without** and **with** Timer
  - Button press detection by polling and interrupt
- The principle of exceptions/ interrupts
  - The triggering event
  - The vector table
  - Interrupted treatment
- Implementation of interrupts with STM32CubeIDE

Correction of the exercise of session 2

- Frequency of Timer7 interrupt requests
  - Timer 7 synchronized at 20 MHz (CK_PSC)
  - TIM7_PSC is 19999, so TIM7_CNT evolves at a rate of 20,000,000 / (19,999 +1) or 1,**000 times per second** (1000 Hz)
  - TIM7_ARR is 1999 thus it implies that the count reaches its end (end of count) after (1999+1)/1000 = 2s
  - So we can say that Timer 7 will send an interrupt request signal **every 2 seconds** (0,5 Hz)**.**
  - We'll see that we can execute a code on this interrupt request from Timer7 (or any other timer).



(Bad) example of LED blinking

- Goal
  - Creating LED blinking
- Main software structure
  - In the form of an infinite loop:

```
void main(){
 LED_Init(); // initializations for LEDs
 while(1){ // always true
        LED_Toggle();     // to change LED
         WaitN(20000);       // to wait
       }
}
```

Time to wait between each LED to be switched on

The functions used by the display

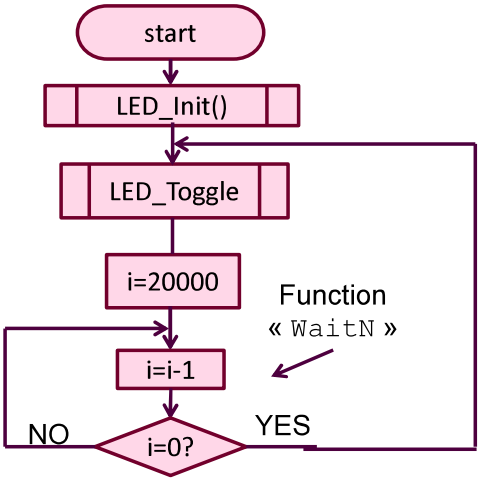
- The "LED_Toggle()" function
  - Changes the status of the LED
    - Lights ON if it is OFF
    - Lights OFF if it is ON
- The "WaitN" function

```
void WaitN(int n){
 int i;
    for(i=n;i>0;i--){
       ;
    }
}
```


Global equivalent algorithm

The Problem:
the microprocessor
(only 1 available)
is used 100% of
the time

CPU usage=100%!
To flash an LED.

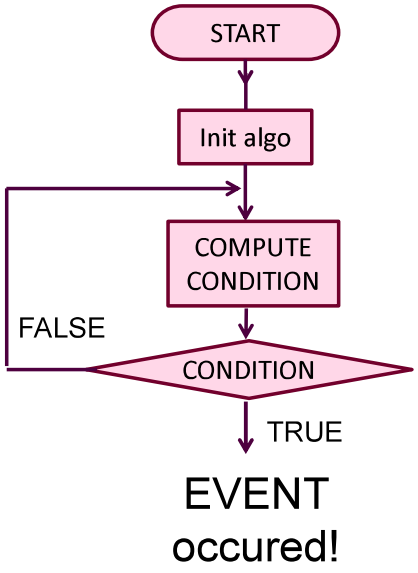


« Exclusive »
implementation.

We're not gonna "get
very far"
with such a method!

Vocabulary: POLLING

- POLLING
  - The goal is to detect an event
  - The basic algorithm for polling is a loop
  - It remains in the loop as the condition revealing an event is false
  - So the algorithm ends as the event occurs
  - This algorithm is executed by the CPU



Events detected by POLLING

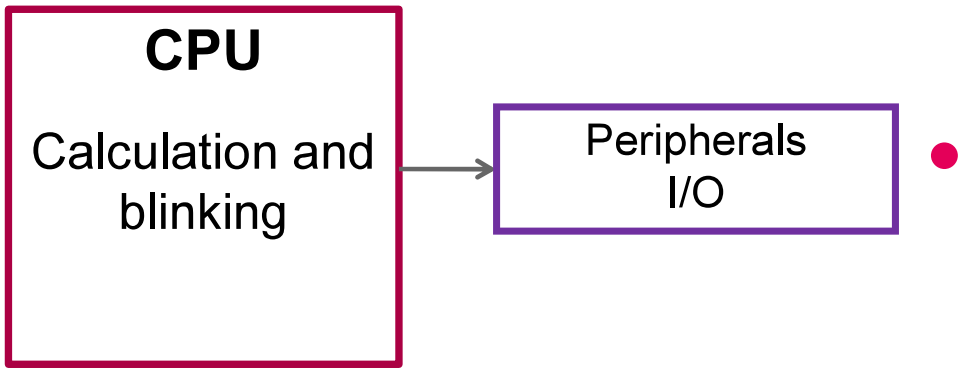
- Duration
  - Done by adding 1 to a variable until a given value is reached
- Change of a digital Input: rising or falling edge
  - Done by reading the input and comparing it to an expected value
- Reception of data on an UART line
  - Done by reading the reception bit RXNE of USARTx_SR
- End of an ADC conversion
  - Done by reading the end of conversion bit EOC of ADCx_SR

Critical analysis of active waiting (or polling)

- The solution implements a wait per count.
  - It works!
  - This is a natural step in a linear procedural writing (framework of the teaching of C in 1st year).
  - The microprocessor is constantly being used (mainly to count time: "`WaitN`").
  - Very complicated to have the microprocessor do another job.
  - The waiting time cannot be easily adjusted precisely.
  - This solution is time consuming for the CPU

Synoptic of "counting in a loop".

- Single CPU usage



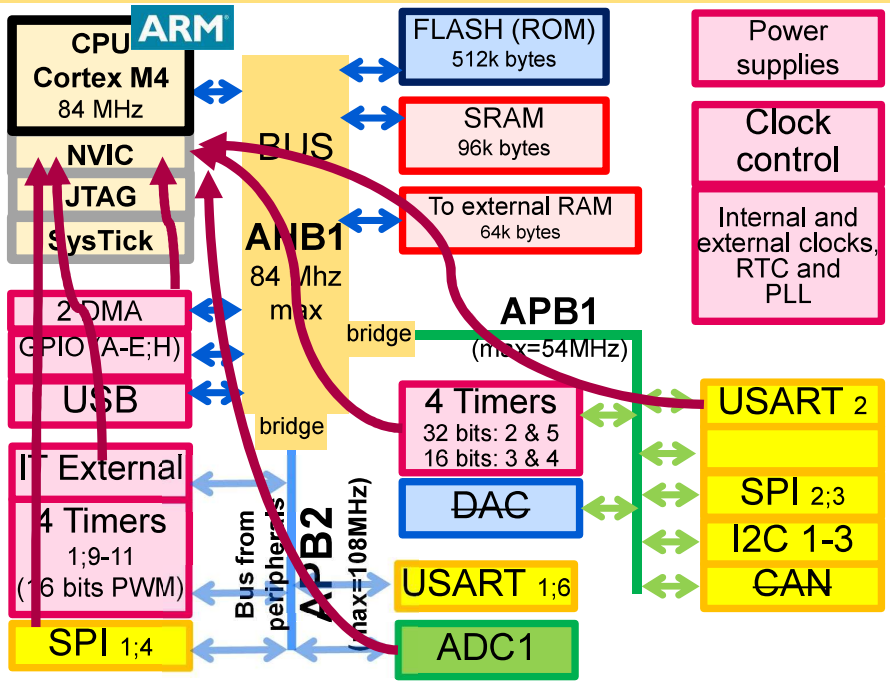
3 Solutions for baking a cake

- Objective: to bake a cake for 50 minutes.
- Means: there is an oven that can be switched on or off but only manually.
- Solution 1: Waiting by counting ("polling" type)
  - You put the cake in the oven.
  - You count 50 times up to 60 (at a rate of one value per second).
  - You cut off the gas supply.
- Solution 2: Waiting by polling
  - You put the cake in the oven and note the time.
  - You constantly scan a watch to check that 50 minutes have not elapsed.
  - When 50 minutes have elapsed you turn off the gas supply.
- Solution 3: You use an interrupt driven timer.
  - You set the timer for 50 minutes.
  - You put the cake in the oven and start the timer...
  - You're going to do something else
  - The timer bell **will interrupt** you in your task so that you can turn off the gas.

Solution analysis

- In any case the cake will be baked (it is assumed that you count at the rate of 1 digit / second).
  - Both solution 1 and solution 2 have in common the **fact that they require your full attention during the entire** cooking process.
  - One can even say that solution 1 is frankly ridiculous!
  - Solution 2 requires additional equipment but you are **fully mobilized** (not efficient!).
  - The solution 3 per timer also requires additional hardware (timer) but, in addition, **you could do something else** for 50 minutes.

Interrupt sources in STM32F401



« Conventional » sources of interrupt

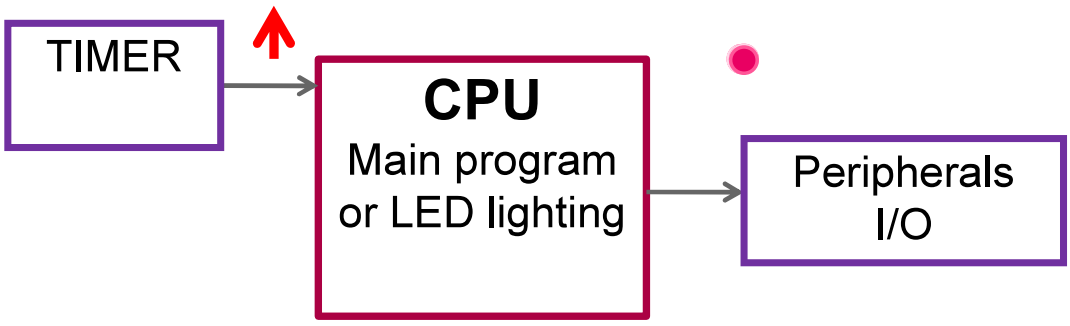
- They emanate **from events** associated with peripherals of the microcontroller:
  - **End of counting of** a counter (SysTick or Timer)
  - **End of conversion of** an ADC
  - Receiving new data on a transmission bus (UART, SPI, I2C)
  - End of sending data on a transmission bus
  - **Modification of an external signal** on the pin of a GPIO
    - Example: detection of a rising or falling edge from a button
  - End of DMA transfer

A new strategy

- How can you do something else at the same time?
  - How to continue to make an LED blink while allowing the microcontroller to execute another code?
  - How, for example, can I retrieve the values of an analog voltage to manage another peripheral? Or how to react to changes in a keyboard?
- Strategy close to that used to allow timer cooking:
  - The microprocessor will be interrupted regularly by a "timer" (see above).
  - Each interrupt of the timer will trigger a new display.
  - Then the microprocessor will be able to do something else while waiting for a new interrupt.

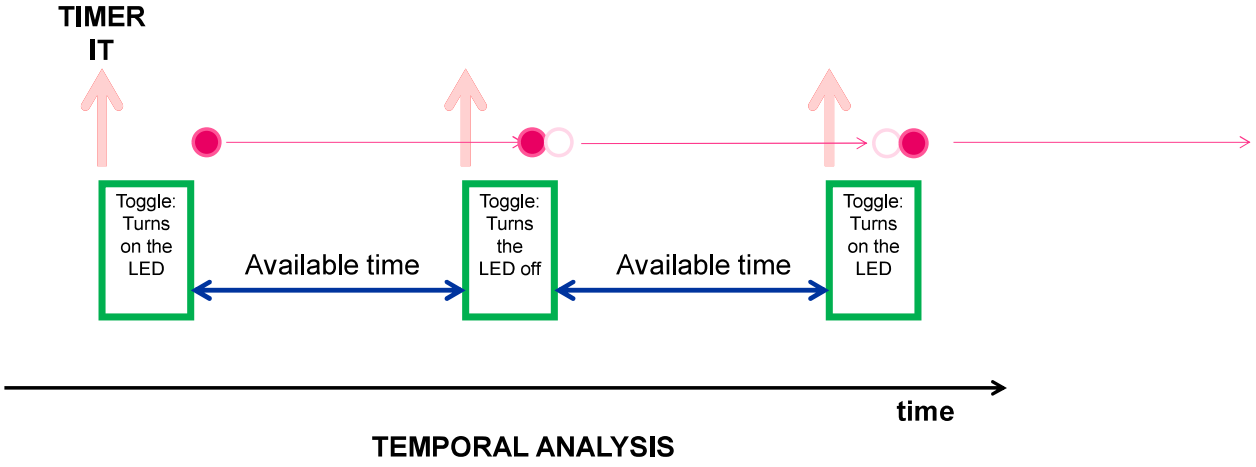
Synoptic of interrupted operation

- The CPU "delegates" the waiting to a peripheral specialized in counting: the TIMER



- The TIMER "wakes" or "interrupts" the CPU at regular and precise time intervals.
- The CPU then triggers a program to calculate the next state of the LED (here simple inversion of its state).

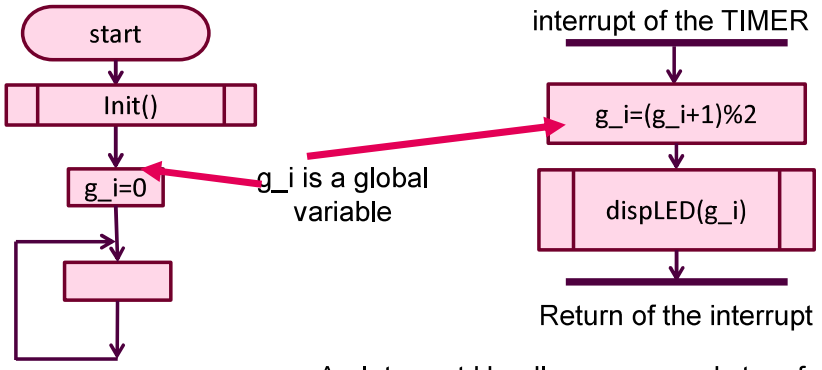
CPU time dedicated to the LED sequence



TEMPORAL ANALYSIS

Note: during the time available the microprocessor is not switched off.
It executes a pointless code: infinite waiting loop. It's not doing anything interesting.
The code of interest will be written in routines triggered by interrupt requests.

LED: Equivalent algorithm



It's the interrupt processing (Interrupt Handler)

This code should be executed as soon as possible.
Here the status of the LED is reversed (toggle).

The main program doesn't "do anything"

But this microprocessor time can now be used for something else!

An Interrupt Handler corresponds to a function (C).
It is a function that is NEVER CALLED in the classical sense but is TRIGGERED by an interrupt request.

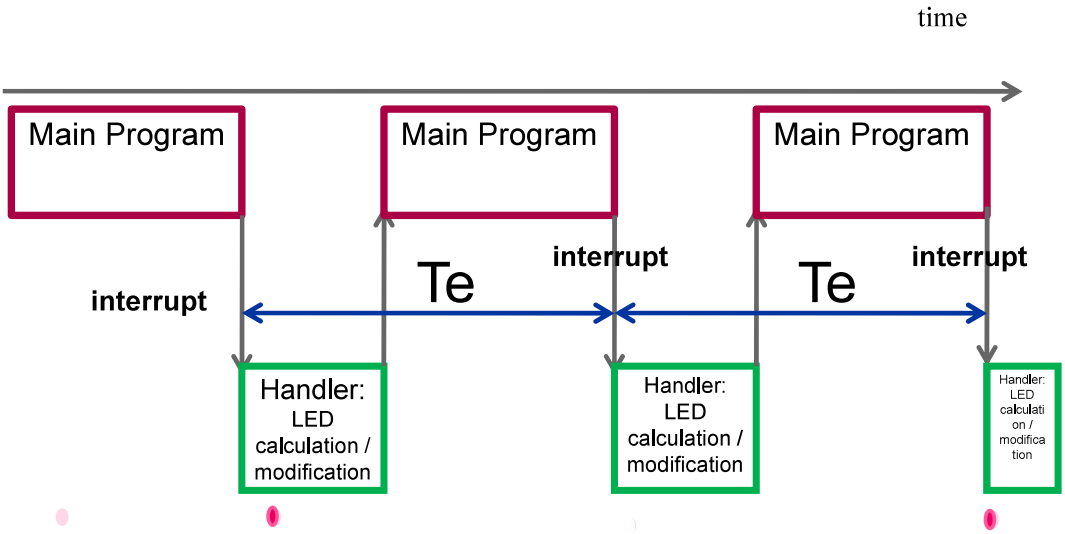
It behaves like a function in the sense that its local variables are initialized at each new launch.

The use of **global variables** is therefore necessary.
The global variables allow to keep in memory a value or a state between 2 successive calls of the interrupted treatment.

interrupt (more details)

- It's a harware event that causes
  - **stopping the normal execution of** a program: the program is interrupted.
  - **saving the state of** the running program: the execution context
  - **the execution of a specific routine** linked to this event: the Interrupt Service Routine (ISR)
  - **restoring the** execution **context**
  - the **return to the execution of the interrupted program**
- **In short: a hardware signal causes a specific code to be executed.**
  - The signal triggers execution: SIGNAL INTERRUPT
  - The code executed is the processing of the termination: HANDLER CODE or ISR

Handler and main program



Notes on the LED toggle in interrupt Handler

- Remarks
  - The main program and the interrupted program share "CPU time". We are in a SINGLE-PROCESSOR context.
  - The program (=interrupt Handler) must be **thought of in a different way** than in its "linear and procedural" version: it is **an event** that **triggers the execution of a code.**
  - The event here is a periodic event: end of counting of a counter.
  - The frequency Te of the interrupts can be set very precisely.
  - A non-periodic and asynchronous event can also trigger an interrupt request: pressing a button for example.

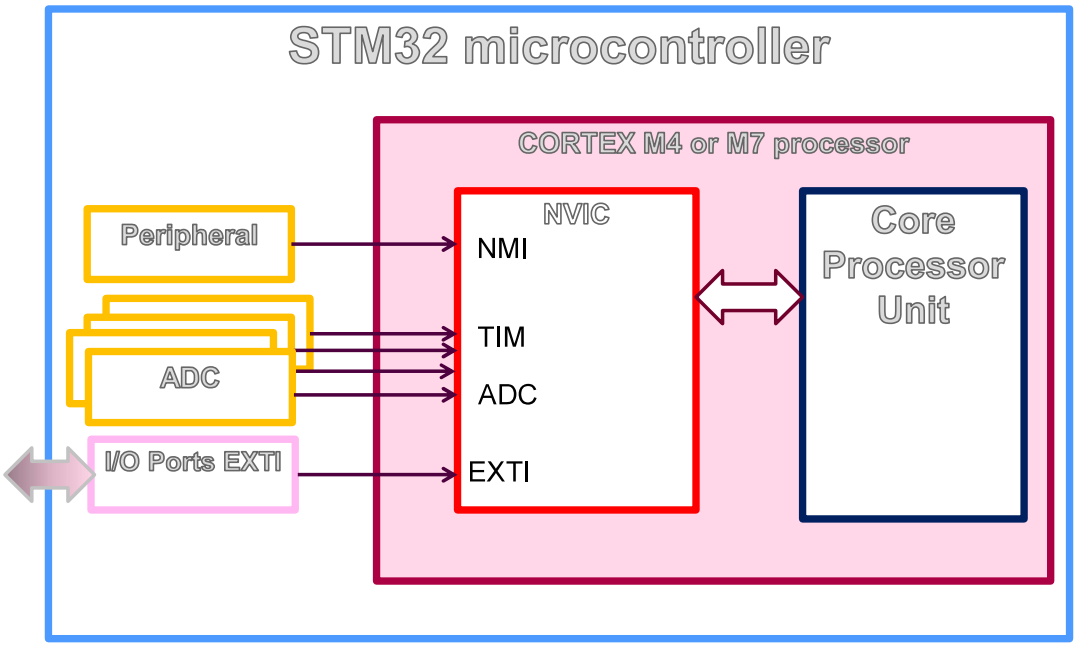
Events and modules

- Duration
  - A Timer is the right module for that: it can generate an interrupt request after a programmed duration
- Change of a digital Input
  - The EXTI module is specialized to detect rising or falling edges: it generates an interrupt request from it
- Reception of data on an UART line
  - An USART interrupt request can be generated from RXNE of USARTx_SR
- End of an ADC conversion
  - An ADC request is sent at the End Of a Conversion (EOC)

Exceptions and interrupts

- Common points
  - PURPOSE: to manage "unexpected" events by triggering an appropriate code.
- The **EXCEPTIONS**
  - They are generated **by the system** (the microprocessor).
  - There's a link to the Cortex M7 processor for the STM32:
    - Instruction in progress: Division by 0, Instruction code does not exist, Reading at a non-existent or prohibited address...
    - Associated microprocessor unit: RESET, Sys Tick
  - 16 exception lines on a Cortex M7: Reset, NMI, HardFault, MemManage, BusFault, SVCall... SysTick
- **Interrupts**
  - They are **of hardware origin external** to the CPU (Cortex M7).
    - Changing a level at the touch of a button
    - End signal of a counter (Timer)
    - There are 98 different sources from your STM32F746
  - No relation with the instruction being executed, so in relation to the CPU clock: they are said to be **asynchronous.**

Architecture for Interrupts & Exceptions

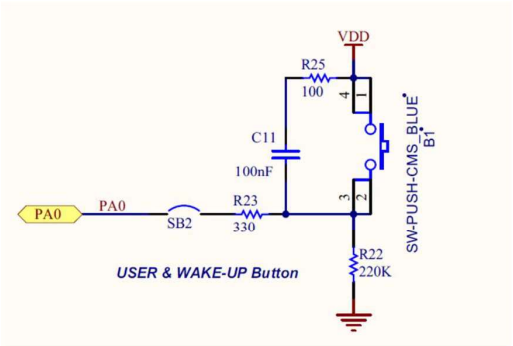


Overview of exceptions and interrupts mechanism

1. The event of a peripheral triggers an interrupt request
2. The processor suspends the currently executing task
3. The processor executes the Interrupt Service Routine (ISR) of the event
4. The processor resumes the previously suspended task

(Bad) example to trigger an action on a button

- Goal
  - To trigger an event as you push a button
- Your software

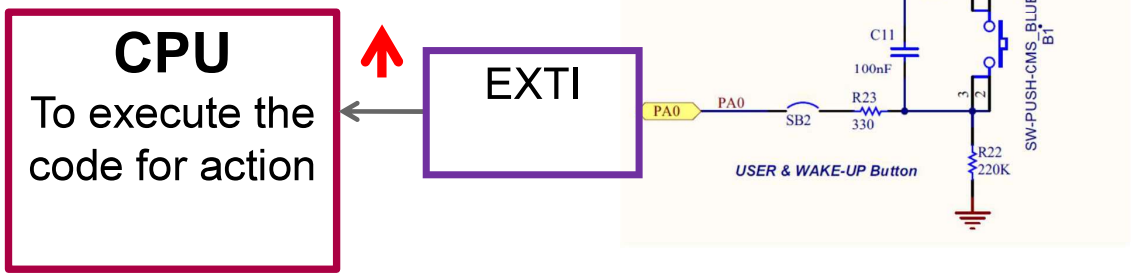


POLLING or BUSY WAIT

```
void main(){
     int button;
 Init(); // to use IOs
     do
          button=HAL_GPIO_ReadPin(GPIOA,GPIO_PIN_0);
     while(button==0);
     action();
}
```

Triggering an action using EXTI and IT

- The CPU is not polling the PA0 pin



- The EXTI module detects the rising edge from the switch
- The CPU executes the code as it is triggered by EXTI signal

Software to trigger an action using EXTI

- The CPU is in an endless loop:

```
void main(){
 Init(); // to use IOs, EXTI and IT
      while(1);{
}
```

- The CPU will stop the previous code each time an EXTI signal is generated (meaning that you push the button)

```
void EXTI0_IRQHandler(){
      action();
}
```

Exceptions and interrupts ID

- Interrupts are a special type of exception.
- Exceptions (internal and external) are identified by a number:
  - For system or internal (so-called "exceptions") exceptions (caused by a hardware or software error in the CPU)
    - it's an exception number from 0 to 15.
  - For external exceptions (i.e. interrupts) (from a peripheral request)
    - It is an **interrupt number from 0 to 80**
    - Or an exception number from 16 to 100.
    - Exception number 16 corresponds to interrupt number 0 etc.

Identification of the Exceptions in the STM32

Number	Priority	NAME	Description
0	-	-	Reserved for stack initialization value
1	Fixed at -3	Reset	Reset
2	Fig. at -2	NMI	Non-Maskable Interrupt from clock system
3	Fig. at -1	Hardfault	All classes of fault
4	Settable 0	MemManage	Memory access fault
5	Settable 1	BusFault	Error on a bus
6	Settable 2	UsageFault	Undefined instruction or prohibited condition
7 to 10		-	
11	Settable 3	SVCall	System call
12	Settable 4	Debug	Monitor
13		-	
14	Settable 5	PendSV	Service Request
15	Settable 6	SysTick	Sys Tick Timer

Some STM32F4xx Interrupt IDs

IT Number: N	NAME	Description	Vector address
0	WWDG	Watchdog	0x0000 0040
1	PVD	Supply voltage monitoring	0x0000 0044
2	TAMPER	Modification on the TAMPER spindle	0x0000 0048
3	RTC	Real time clock	0x0000 004C
...	...	...	...
6	EXTI0	Detecting a change on EXTI0	0x0000 0058
7	EXTI1	Detection of a modification on EXTI1	0x0000 005C
8	EXTI2	Detection of a modification on EXTI2	0x0000 0060
9	EXTI3	Detection of a modification on EXTI3	0x0000 0064
10	EXTI4	Detection of a modification on EXTI4	0x0000 0068
11	DMA1_Stream0	DMA1 channel 0	0x0000 006C
...	...	...	...
18	ADC	Digital-analog converters 1, 2 and 3	0x0000 0088

Other STM32F4xx Interrupt IDs

Number N	NAME	Description	Vector address
21	CAN1_RX1	CAN Bus (≠ Analog Digital Conv)	0x0000 0094
...	...	...	...
28	TIM2	Counter (or Timer) n°2	0x0000 00B0
29	TIM3	Counter (or Timer) n°3	0x0000 00B4
...	...	...	...
34	I2C2_ER	Error on I2C bus n°2	0x0000 00C8
35	SPI1	SPI Bus No. 1	0x0000 00CC
36	SPI2	SPI Bus No. 2	0x0000 00D0
...	....	...	...
52	UART4	Asynchronous serial link 4	0x0000 0110
55	UART5	Asynchronous serial link 5	0x0000 0114
...	...	...	...
90	DMA2D	DMA for graphic accelerator	0x0000 01A8

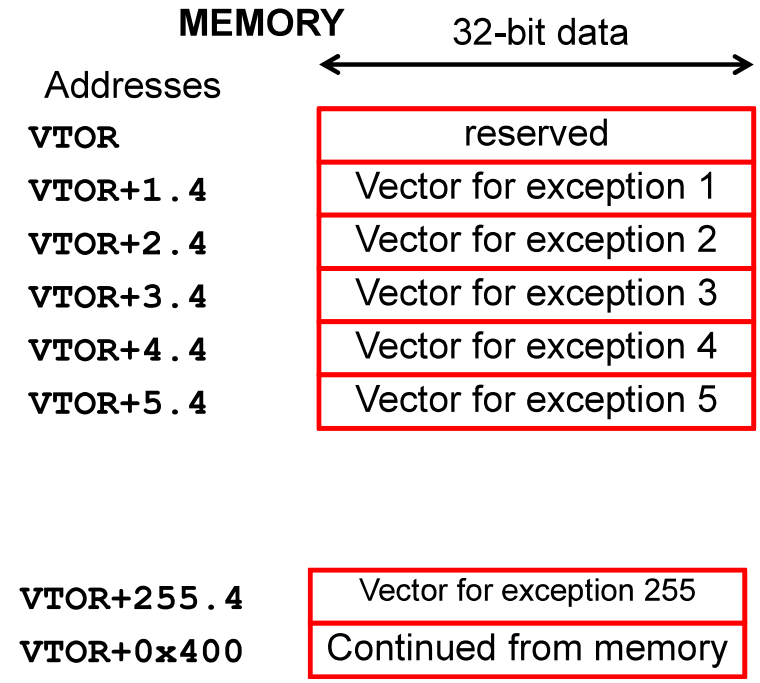
Vectorization of STM32 exceptions / interrupts

- The interrupts of the STM32 are "vectorized":
  - An exception handling program (Interrupt Handler) starts (like any program) at an address:
    - This address is the location of the first instruction in the program.
    - This address, for the STM32 is expressed on 32 bits (thus 4 bytes).
    - This address is called the **VECTOR of the** interrupt.
  - **For each interrupt source** the microcontroller associates
    - an identifier (a value) (0 to 97)
    - a **32-bit=4-byte** interrupt vector, i.e. an interrupt handler or interrupt service routine.
  - In other words:
    - An **interrupt vector** is **the address of the interrupt service routine**, i.e. the beginning of the interrupt code: location of the interrupt handler.
    - Knowing the vector associated with an interrupt is knowing where it starts.
  - Interrupt vectors are stored in a table (or array)

Interrupt vector

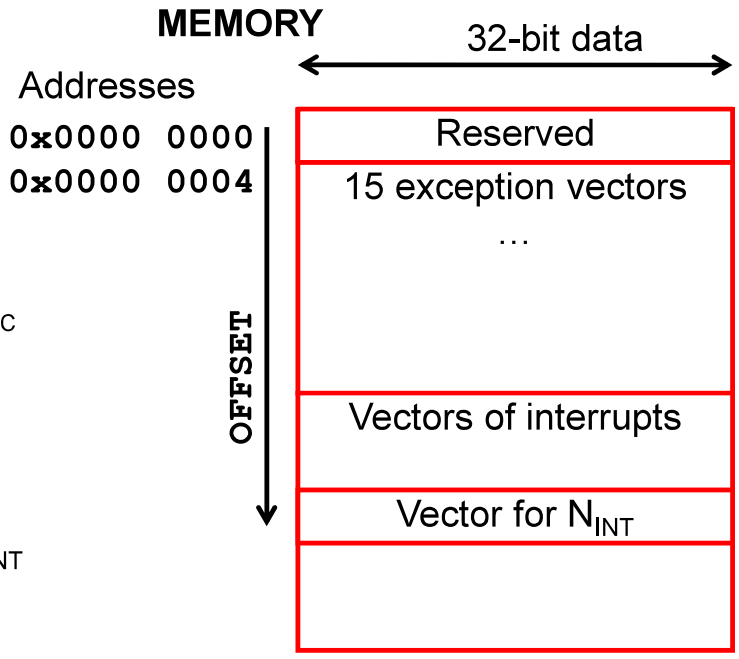
- Definition
  - The "vector" of an interrupt is the address at which the code (or routine) associated (triggered) with this interrupt starts.
  - It is therefore a 32-bit value (=4 bytes).
- Example:
  - If the vector of TIMER6 is `0x0800 0240` or (`0x0800 0241`)
  - The first instruction of the interrupt processing associated with TIMER6 is therefore located in memory at address `0x0800 0240`.

Location of the vector table



Vector arrangement: vector table

- Exception vectors are stored in memory.
  - successively
  - in an array structure called **vector table**
  - they are located individually by an offset depending on their N_{EXC} exception number:
  - OFFSET = N_{EXC} x 4 (because 1 address = 4 bytes)
    - Or their interrupt number N_{INT}
    - $N_{INT=}$ = N_{EXC} -16
  - OFFSET = (N_{INT} +16) x 4 = 4 N_{INT} + 4 x 16



Location of the vector table

- By default it starts at the address 0x00000000
- This address is in ROM (Flash memory).
- It can sometimes be interesting to modify the vectors during the execution of a program: that's why the table is "relocatable".
- The **SCB_VTOR** register contains the address at which the vector table starts.
  - Default, at startup: SCB_VTOR=0x0000 0000
  - The vector of an interrupt is therefore marked by:
    - The **start address** of array SCB_VTOR
    - + an **offset** calculated according to interrupt number N

SCB_VTOR: a register to locate Vectors

Vector table offset register (SCB_VTOR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved		TBLOFF[29:16]													
		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TBLOFF[15:9]							Reserved								
rw	rw	rw	rw	rw	rw	rw									

- Bits 0 to 8 then 30 and 31 must remain at 0.
- Bits 9 to 29 determine VTOR
- If bit 29 is set to 1, it means that the table is located in RAM. In fact the beginning of the RAM is: 0x2000 0000

Use of the vector table by the CPU

- When an interrupt is triggered the CortexM4 (or M7) runs automatically:
  - Automatic **backup** and restore of the execution context (registers used):
  - Automatic reading of a so-called "**vector table**" which contains the addresse of the service routines associated with the calling interrupt.
  - Note: The vector allows the microprocessor to identify the location of the code to be executed as an exception/interrupt.

Functionnalities of the NVIC

- NVIC: Nested Vectored Interrupt Controller
  - A module that controls interrupts
  - Each request is identified by a number by NVIC
  - NVIC authorizes or not, individually, each interrupt
  - It sets priorities between each interrupt
  - Its behavior will be seen in the next session

Writing a code to be executed as an exception

- Each exception or interrupt triggers its own code.
  - This is the exception handler
  - ISR: Interrupt Service Routine
  - Or **Interrupt Handler**
- When using STM32Cube or Keil µVision and associated libraries, **the name of each interrupted process is fixed** (see list in documents).
  - The list of names is available in the file "startup_stm32fXxx.s".
  - For example: "`TIM3_IRQHandler`" for the code associated with Timer 3.
- To write interrupt code in C using the STM32Cube compiler
  - You must write a C function with the predefined name for the exception
  - That returns nothing
  - Which has no input parameters

Example of a code and vector

- We've written the interrupt program for Timer 3:

```
void TIM3_IRQHandler(void){
   g_i=g_i+1;
…
}
```

- The debugger provides the following implementation in memory:

```
TIM3_IRQ_Handler:
```

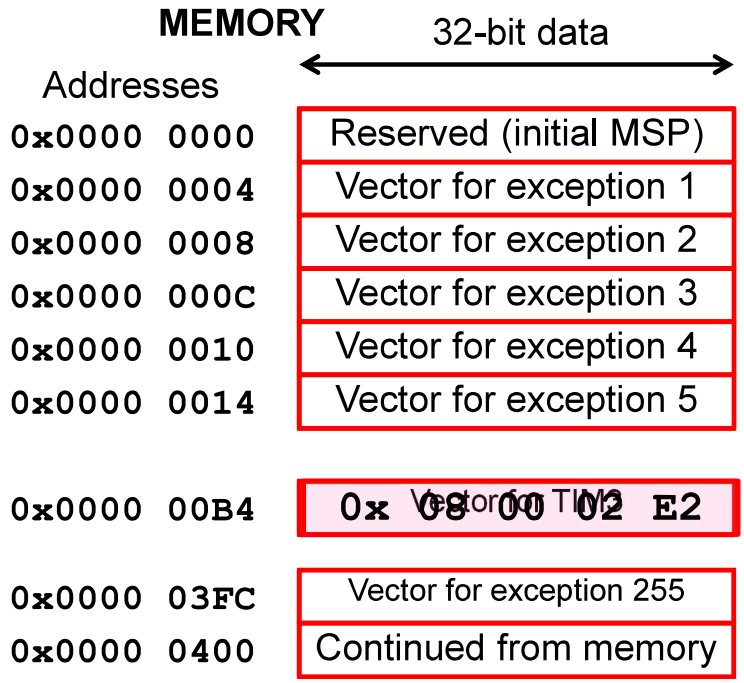
Address	*Assembly*	*instruction code*
0x080002E2	4808	LDR r0, [pc,#32]
0x080002E4	6800	LDR r0, [r0,#0x00]
0x080002E6	1C40	ADDS r0,r0,#1

The Timer3 vector is therefore: **0x080002E2**

This is the address where the processing associated with Timer3 starts.

Calculating the location of a vector

- The vectors of the interrupts are arranged in the vector table after the 16 places reserved for exceptions with An offset from the beginning of the table:
  - 16x4 + N x 4 (1 word = 4 bytes)
  - Example
    - TIM3 is interrupt n°29 (called « position » in Reference Manual)
    - Its vector is located in the vector table with an offset: 16x4 + 29x4 = 180=0xB4



How to read the Vector Table?

- Document
  - In the RM (Reference Manual)
  - Chapter about NVIC
- Position
  - Identification number of an interrupt
  - not available for exceptions
- Priority
  - Given by default at Reset for Exceptions
  - It can be used to separate priority level
- Type of priority
  - Fixed or not: (only 3 exceptions are fixed)
  - The other priorities can be modified by software
- Acronym
  - The name of the Interrupt source
- Address
  - This is where is stored the Vector of the interrupt

Complete table of vectors 1/6 (STM32L47x)

Table 58. STM32L47x/L48x/L49x/L4Ax vector table

Position=N_{INT}

Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000 0000
-	-3	fixed	Reset	Reset	0x0000 0004
-	-2	fixed	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000 0008
-	-1	fixed	HardFault	All classes of fault	0x0000 000C
-	0	settable	MemManage	Memory management	0x0000 0010
-	1	settable	BusFault	Pre-fetch fault, memory access fault	0x0000 0014
-	2	settable	UsageFault	Undefined instruction or illegal state	0x0000 0018
-	-	-	-	Reserved	0x0000 001C - 0x0000 0028
-	3	settable	SVCall	System service call via SWI instruction	0x0000 002C
-	4	settable	Debug	Monitor	0x0000 0030
-	-	-	-	Reserved	0x0000 0034
-	5	settable	PendSV	Pendable request for system service	0x0000 0038
-	6	settable	SysTick	System tick timer	0x0000 003C
0	7	settable	WWDG	Window Watchdog interrupt	0x0000 0040

EXCEPTIONS

N_{INT}=- (not an interrupt)

INTERRUPTS

N_{INT}=0 (first interrupt)

Complete vector table 2/6 (STM32L47x)

N_{INT}					
0		settable	WWDG	Window Watchdog interrupt	0x0000 0040
1		settable	PVD_PVM	PVD/PVM1/PVM2/PVM3/PVM4 through EXTI lines 16/35/36/37/38 interrupts	0x0000 0044
2		settable	RTC_TAMP_STAMP /CSS_LSE	RTC Tamper or TimeStamp /CSS on LSE through EXTI line 19 interrupts	0x0000 0048
3		settable	RTC_WKUP	RTC Wakeup timer through EXTI line 20 interrupt	0x0000 004C
4		settable	FLASH	Flash global interrupt	0x0000 0050
5		settable	RCC	RCC global interrupt	0x0000 0054
6		settable	EXTI0	EXTI Line0 interrupt	0x0000 0058
7		settable	EXTI1	EXTI Line1 interrupt	0x0000 005C
8		settable	EXTI2	EXTI Line2 interrupt	0x0000 0060
9		settable	EXTI3	EXTI Line3 interrupt	0x0000 0064
10		settable	EXTI4	EXTI Line4 interrupt	0x0000 0068
11		settable	DMA1_CH1	DMA1 channel 1 interrupt	0x0000 006C
12		settable	DMA1_CH2	DMA1 channel 2 interrupt	0x0000 0070
13		settable	DMA1_CH3	DMA1 channel 3 interrupt	0x0000 0074
14		settable	DMA1_CH4	DMA1 channel 4 interrupt	0x0000 0078
15		settable	DMA1_CH5	DMA1 channel 5 interrupt	0x0000 007C

Complete vector table 3/6 (STM32L47x)

N_{INT}					
16		settable	DMA1_CH6	DMA1 channel 6 interrupt	0x0000 0080
17		settable	DMA1_CH7	DMA1 channel 7 interrupt	0x0000 0084
18		settable	ADC1_2	ADC1 and ADC2 global interrupt	0x0000 0088
19		settable	CAN1_TX	CAN1_TX interrupts	0x0000 008C
20		settable	CAN1_RX0	CAN1_RX0 interrupts	0x0000 0090
21		settable	CAN1_RX1	CAN1_RX1 interrupt	0x0000 0094
22		settable	CAN1_SCE	CAN1_SCE interrupt	0x0000 0098
23		settable	EXTI9_5	EXTI Line[9:5] interrupts	0x0000 009C
24		settable	TIM1_BRK/TIM15	TIM1 Break/TIM15 global interrupts	0x0000 00A0
25		settable	TIM1_UP/TIM16	TIM1 Update/TIM16 global interrupts	0x0000 00A4
26		settable	TIM1_TRG_COM /TIM17	TIM1 trigger and commutation/TIM17 interrupts	0x0000 00A8
27		settable	TIM1_CC	TIM1 capture compare interrupt	0x0000 00AC
28		settable	TIM2	TIM2 global interrupt	0x0000 00B0
29		settable	TIM3	TIM3 global interrupt	0x0000 00B4
30		settable	TIM4	TIM4 global interrupt	0x0000 00B8
31		settable	I2C1_EV	I2C1 event interrupt	0x0000 00BC

Complete vector table 4/6 (STM32L47x)

N_{INT}					
32		settable	I2C1_ER	I2C1 error interrupt	0x0000 00C0
33		settable	I2C2_EV	I2C2 event interrupt	0x0000 00C4
34		settable	I2C2_ER	I2C2 error interrupt	0x0000 00C8
35		settable	SPI1	SPI1 global interrupt	0x0000 00CC
36		settable	SPI2	SPI2 global interrupt	0x0000 00D0
37		settable	USART1	USART1 global interrupt	0x0000 00D4
38		settable	USART2	USART2 global interrupt	0x0000 00D8
39		settable	USART3	USART3 global interrupt	0x0000 00DC
40		settable	EXTI15_10	EXTI Line[15:10] interrupts	0x0000 00E0
41		settable	RTC_ALARM	RTC alarms through EXTI line 18 interrupts	0x0000 00E4
42		settable	DFSDM1_FLT3	DFSDM1_FLT3 global interrupt	0x0000 00E8
43		settable	TIM8_BRK	TIM8 Break interrupt	0x0000 00EC
44		settable	TIM8_UP	TIM8 Update interrupt	0x0000 00F0
45		settable	TIM8_TRG_COM	TIM8 trigger and commutation interrupt	0x0000 00F4
46		settable	TIM8_CC	TIM8 capture compare interrupt	0x0000 00F8
47		settable	ADC3	ADC3 global interrupt	0x0000 00FC

Complete vector table 5/6 (STM32L47x)

48	settable	FMC	FMC global interrupt	0x0000 0100
49	settable	SDMMC1	SDMMC1 global interrupt	0x0000 0104
50	settable	TIM5	TIM5 global interrupt	0x0000 0108
51	settable	SPI3	SPI3 global interrupt	0x0000 010C
52	settable	UART4	UART4 global interrupt	0x0000 0110
53	settable	UART5	UART5 global interrupt	0x0000 0114
54	settable	TIM6_DACUNDER	TIM6 global and DAC1 underrun interrupts	0x0000 0118
55	settable	TIM7	TIM7 global interrupt	0x0000 011C
56	settable	DMA2_CH1	DMA2 channel 1 interrupt	0x0000 0120
57	settable	DMA2_CH2	DMA2 channel 2 interrupt	0x0000 0124
58	settable	DMA2_CH3	DMA2 channel 3 interrupt	0x0000 0128
59	settable	DMA2_CH4	DMA2 channel 4 interrupt	0x0000 012C
60	settable	DMA2_CH5	DMA2 channel 5 interrupt	0x0000 0130
61	settable	DFSDM1_FLT0	DFSDM1_FLT0 global interrupt	0x0000 0134
62	settable	DFSDM1_FLT1	DFSDM1_FLT1 global interrupt	0x0000 0138
63	settable	DFSDM1_FLT2	DFSDM1_FLT2 global interrupt	0x0000 013C

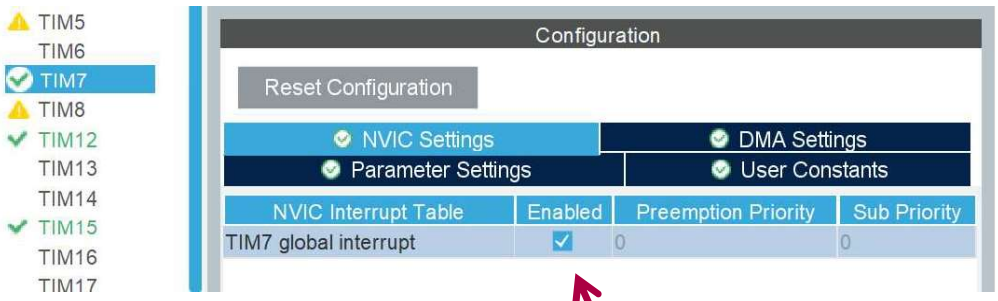
Complete vector table 6/6 (STM32L47x)

64	settable	COMP	COMP1/COMP2 through EXTI lines 21/22 interrupts	0x0000 0140
65	settable	LPTIM1	LPTIM1 global interrupt	0x0000 0144
66	settable	LPTIM2	LPTIM2 global interrupt	0x0000 0148
67	settable	OTG_FS(1)	OTG_FS global interrupt	0x0000 014C
68	settable	DMA2_CH6	DMA2 channel 6 interrupt	0x0000 0150
69	settable	DMA2_CH7	DMA2 channel 7 interrupt	0x0000 0154
70	settable	LPUART1	LPUART1 global interrupt	0x0000 0158
71	settable	QUADSPI	QUADSPI global interrupt	0x0000 015C
72	settable	I2C3_EV	I2C3 event interrupt	0x0000 0160
73	settable	I2C3_ER	I2C3 error interrupt	0x0000 0164
74	settable	SAI1	SAI1 global interrupt	0x0000 0168
75	settable	SAI2	SAI2 global interrupt	0x0000 016C
76	settable	SWPMI1	SWPMI1 global interrupt	0x0000 0170
77	settable	TSC	TSC global interrupt	0x0000 0174
78	settable	LCD(2)	LCD global interrupt	0x0000 0178
79	settable			0x0000 017C
80	settable	RNG	RNG interrupt	0x0000 0180

How to generate Handler code with STM32CubeIDE?

2: Choose a peripheral

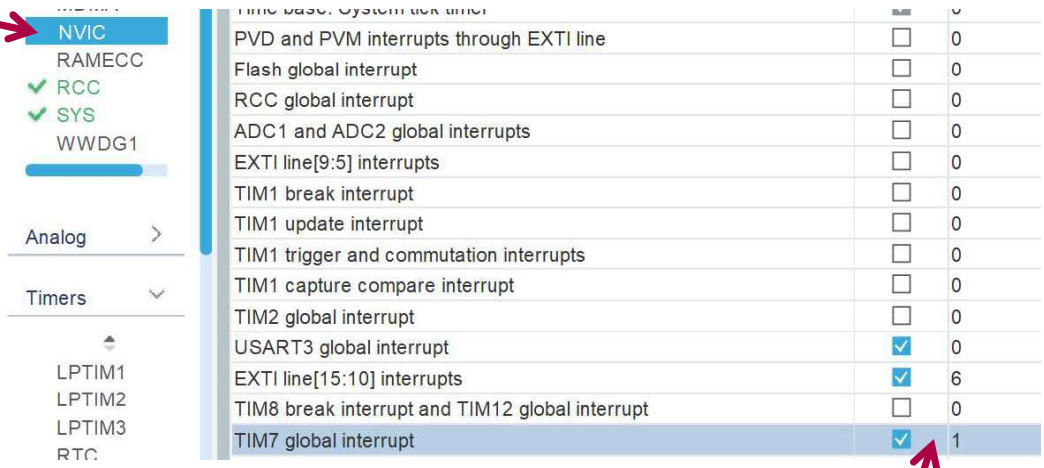
1: Open ioc view



3: Enable the event

Modification of the priority level

Choose NVIC



Save to generate the code

Modify the preemption priority

Write your handler code

After generation, you should find in « stm32Xxx_it.c » the place to write your code

```
void TIM7_IRQHandler(void)
{
  /* USER CODE BEGIN TIM7_IRQn 0 */

  /* USER CODE END TIM7_IRQn 0 */
  HAL_TIM_IRQHandler(&htim7);
  /* USER CODE BEGIN TIM7_IRQn 1 */

  /* USER CODE END TIM7_IRQn 1 */
}
```

Insert your code there

Insert your code there

Do not insert code elsewhere

Exercise 3 for Session 4

- Know how to locate in memory where interrupt processing associated with a peripheral begins
- Example:
  - Where is the first instruction executed during interrupt processing of a request from the EXTI1 peripheral?

Addresses	Data (32 bits)
0x00000048	0x08000204
0x0000004C	0x080013F0
0x00000050	0x08000408
0x00000054	0x08000B24
0x00000058	0x08000A00
0x0000005C	0x08001C10
0x00000060	0x0800220C