COPYRIGHT

DISCLAIMER

This content is provided without warranty or representation of any kind. The use of the content is entirely at your own risk and Stellenbosch University (SU) will have no liability directly or indirectly as a result of this content.

The content must not be assumed to provide complete coverage of the particular study material. Content may be removed or changed without notice.

The video is of a recording with very limited post-recording editing. The video is intended for use only by SU students enrolled in the particular module.

UNIVERSITEIT
iYUNIVESITHI
STELLENBOSCH
UNIVERSITY

100
1918 · 2018

forward together · saam vorentoe · masiye phambili

Computer Systems / Rekenaarstelsels 245 - 2020

Lecture 8

Interrupts and Exceptions: Part 2
Onderbrekings en Uitsonderings: Deel 2

Dr Rensu Theart & Dr Lourens Visagie

Lecture Overview

- Exception handling sequence on the Cortex M4
- Nested Vectored Interrupt Controller (NVIC)
- Interrupt priority
- High-level coding of interrupts

Interrupt considerations

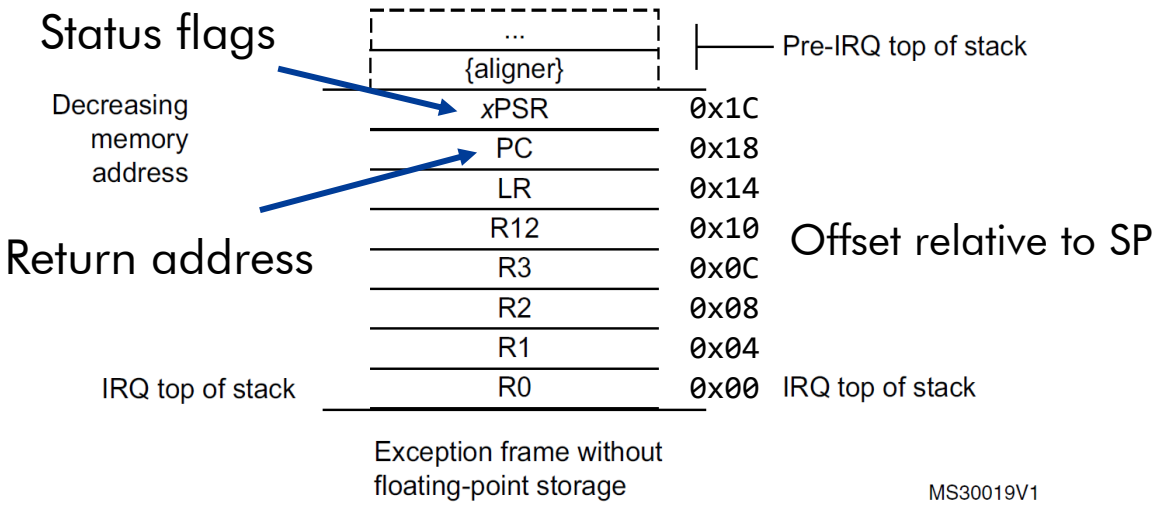
- How does the processor know which function to execute (and where to find it in memory)?
- How do we ensure that the interrupted program does not get confused (i.e. register and status flag values unchanged)?
- Which interrupt will get preference if more than one occur at the same time?
- How do we ensure that no interrupt can occur during critical parts of the program?

Exception handling sequence on the Cortex M4 / Uitsondering hantering van die Cortex M4

When the processor takes an exception **(exception entry):**

1. The processor automatically pushes eight data words onto the current stack.
   - This operation is referred as **stacking** and the structure of eight data words is referred as **stack frame**.
   - The stack frame includes the return address. This is the address of the next instruction in the interrupted program.
   - This value is restored to the PC when the exception returns so that the interrupted program resumes.

Figure 12. Cortex-M4 stack frame layout



Exception handling sequence on the Cortex M4 / Uitsondering hantering van die Cortex M4

2. In parallel to the stacking operation, the processor performs a **vector fetch** that reads the exception handler start address from the vector table.
3. When stacking is complete, the processor **starts executing** the exception handler code.
   - Also changes the status of the corresponding interrupt as active.
4. At the same time, the processor writes an EXC_RETURN value to the LR.
   - This indicates which stack pointer corresponds to the stack frame and what operation mode the processor was in before the entry occurred.

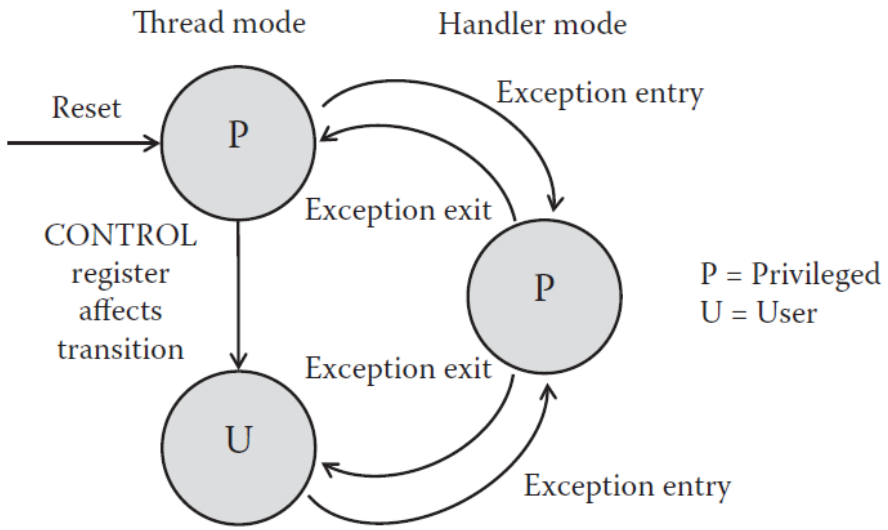
Exception handling sequence on the Cortex M4 / Uitsondering hantering van die Cortex M4

- Cortex M4 modes are:
  - **Thread mode**: normal program execution
  - **Handler mode**: exception handler execution
    - Returns to Thread mode when it has finished exception processing
- The privilege levels of software execution are:
  - **Unprivileged/User:** Limited access to special instructions working with core registers, timers, memory and peripherals, etc.
  - **Privileged:** has access to all instructions and resources
    - Always used for interrupts

Exception handling sequence on the Cortex M4 / Uitsondering hantering van die Cortex M4

- There are two different stack pointers (both accessed as R13, but depends on mode)
  - Mainly when you have an OS

	Privileged	User
Handler mode	Use: Exception handling Stack: Main	
Thread mode	Use: Applications Stack: Main or Process	Use: Applications Stack: Main or Process



Exception handling sequence on the Cortex M4 / Uitsondering hantering van die Cortex M4

How do we return from an interrupt?

- **Exception return** occurs when the processor is in Handler mode and executes LDM, POP or LDR with PC as destination or BX.
  - This causes the equivalent of BX LR, which loads the EXC_RETURN value into the PC.
- EXC_RETURN is the value loaded into the LR on exception entry. The exception mechanism relies on this value to detect when the processor has completed an exception handler.
  - The lowest 5 bits gives information on the return stack and processor mode.
  - (LR does not contain a copy of the address where to return to, that was stored on the stack)

TABLE 15.2
EXC_RETURN Value for the Cortex-M4 with Floating-Point Hardware

EXC_RETURN[31:0]	State	Return to	Using Stack Pointer
0xFFFFFFE1	Floating-point	Handler mode	MSP
0xFFFFFFE9	Floating-point	Thread mode	MSP
0xFFFFFFED	Floating-point	Thread mode	PSP
0xFFFFFFF1	Non-floating-point	Handler mode	MSP
0xFFFFFFF9	Non-floating-point	Thread mode	MSP
0xFFFFFFFD	Non-floating-point	Thread mode	PSP

Exception handling sequence on the Cortex M4 / Uitsondering hantering van die Cortex M4

Exception return:

- When the processor is in Handler mode, and one of the possible EXC_RETURN values from the previous table is written to the PC (which is what will happen with BX LR instruction in the interrupt handler), the processor will automatically:
  - Restore the mode based on the EXC_RETURN value
  - Pop the stack frame from the stack into the original registers.
  - The stack frame contains the PC. This will cause PC to get the value it had at the time the exception occurred
  - End of exception – program continues

- The return instruction from an exception can thus still use conventional way of returning from subroutine (i.e. BX LR) even though LR does not contain the address to return to. The end effect is the same.

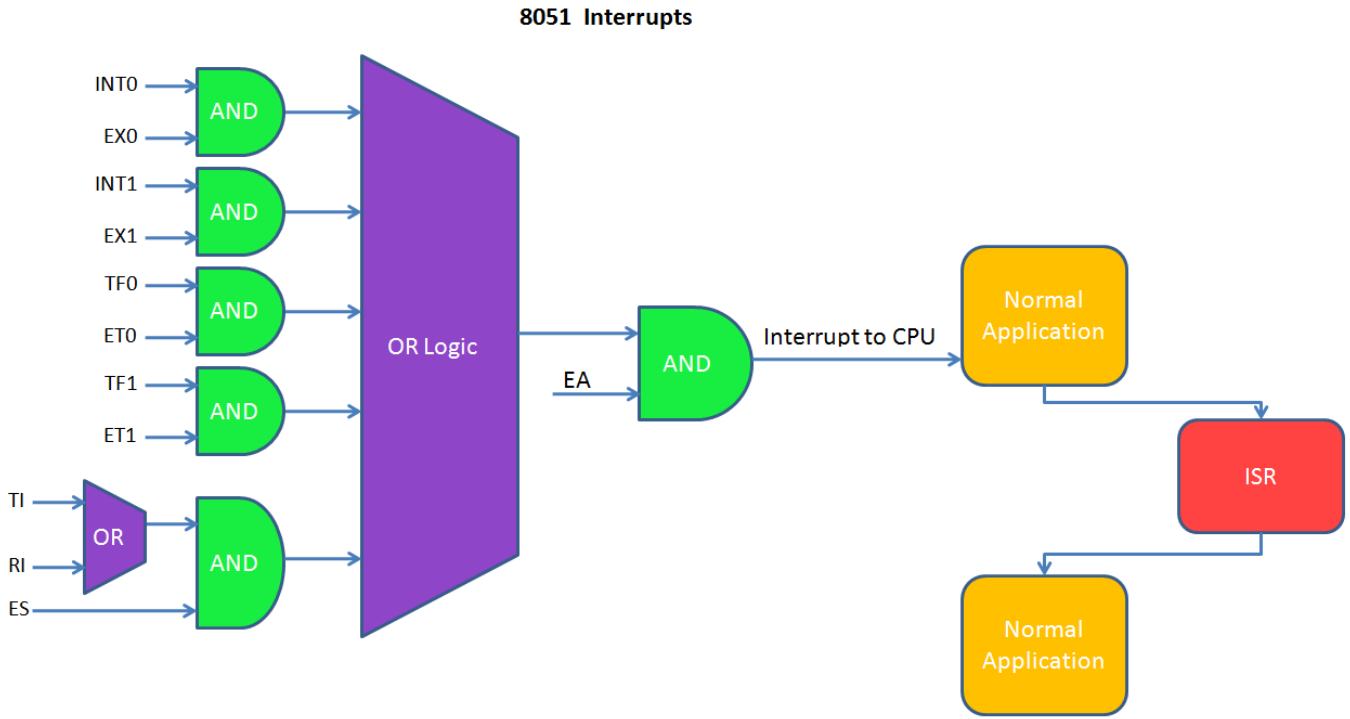
Side Note

How does a single core processor execute multiple processes / tasks / applications? (i.e. Windows)

- Execute little bits of each task in turn – makes it look like they are running concurrently.
- In exception handlers (timer that expires and others) determine if the current process has run long enough (give another one a turn) or if the current process is blocking – waiting for something to happen.
- Execution switch to another process is called a **context switch**.
- 4 cores operating at 400MHz, each executing a different process – or a single core running at 1.6GHz, time-sliced process execution.
- Context switches are usually computationally intensive, and much of the design of operating systems is to optimize the use of context switches.
- Switching from one process to another requires a certain amount of time for doing the administration – saving and loading registers and memory maps, updating various tables and lists, etc.

Interrupts / Onderbrekings

- Remember, interrupts come from connected peripherals (timers, GPIO, etc.)
- The processor only has one interrupt signal – how does it know which interrupt occurred?
- Older microcontrollers – OR all separate interrupt lines together and connect to processor interrupt request line.
- Processor has to check the state of each peripheral to find out which one signaled the interrupt.



Interrupts / Onderbrekings

- Modern solution - **Interrupt Controller**: Logic device, integrated circuit, that combines multiple interrupt sources.
- Allows setting priorities on various sources, so that CPU switches execution to the most appropriate ISR.
- Also allows chaining pending interrupts (let them wait for current interrupt to finish).
- Most importantly – provides the source of the interrupt (i.e. register that must be read)
- **Vectored Interrupt Controller (VIC)** – provides to the microprocessor the address of the handler routine (the interrupt vector).
- On the Cortex M4 the VIC is tightly interwoven with the rest of the microcontroller core – called a **Nested Vectored Interrupt Controller (NVIC)**. A bus between the processor and VIC is used to relay the handler function address.
- NVIC allows the processor to automatically decide which exception handler to execute – no code required.

Interrupts / Onderbrekings

For STM32F411 the NVIC supports:

- 52 maskable interrupt channels
- A programmable priority level of 0-15 for each interrupt.
  - A higher level corresponds to a lower priority, so level 0 is the highest interrupt priority.
- Level and pulse detection of interrupt signals.
- Dynamic reprioritization of interrupts.
- Grouping of priority values into group priority and subpriority fields
- Interrupt tail-chaining
  - This mechanism speeds up exception servicing. On completion of an exception handler, if there is a pending exception that meets the requirements for exception entry, the stack pop is skipped and control transfers to the new exception handler.
- An external *Non-maskable interrupt* (NMI)
  - This is the highest priority exception other than reset. It is permanently enabled and has a fixed priority of -2.
  - It cannot be masked (disabled) or prevented by another exception

Interrupt priority / Onderbreking prioriteit

- **Which interrupt will get preference if more than one occur at the same time?**

- We can set the priority of individual interrupts by using the **Interrupt priority registers (NVIC_IPRx)**
  - Each NVIC_IPRx register provide an 8-bit priority field for each interrupt. These registers are byte-accessible. Each register holds four priority fields, that map to four elements in the interrupt priority array IP[0] to IP[67], as shown in Figure 19.

- Exception priorities can be programmatically specified. (except for a few fixed ones).
- Exceptions with higher priority may interrupt another exception.

Figure 19. NVIC_IPRx register mapping

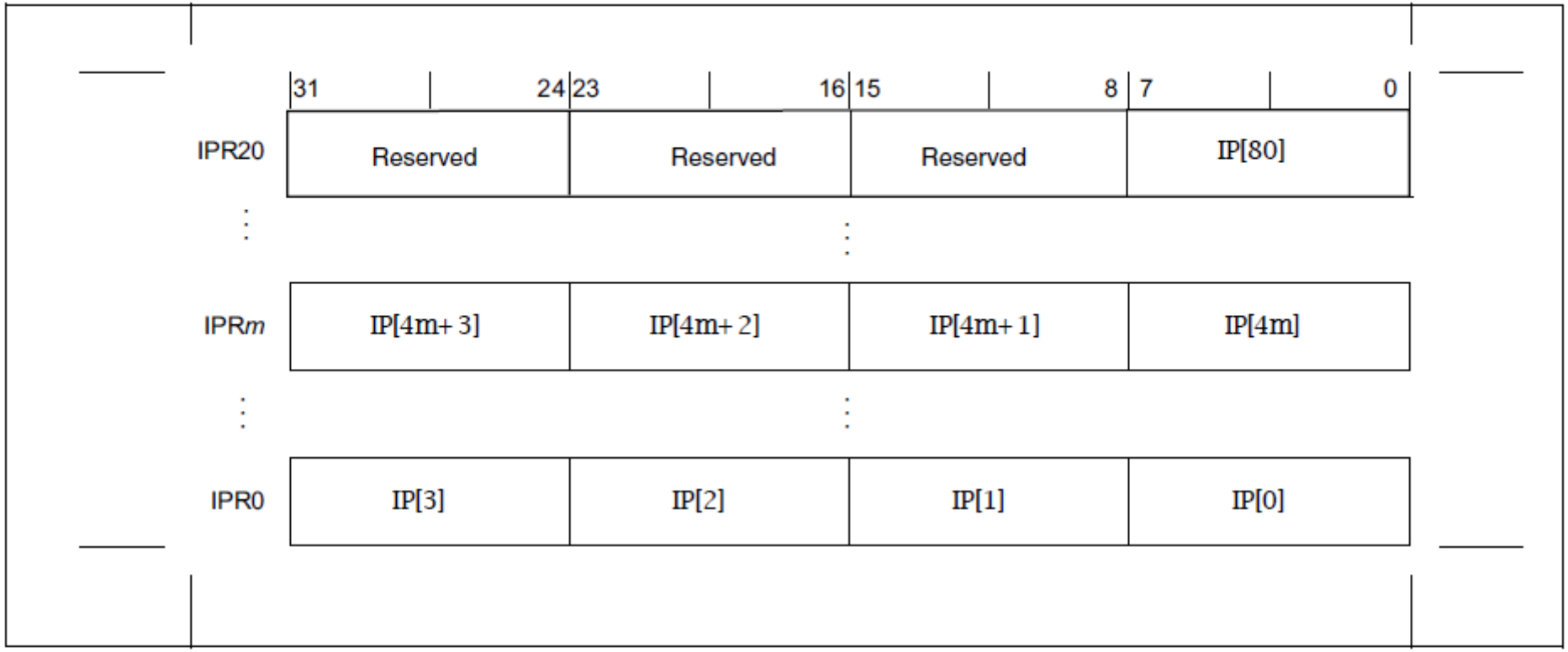


Table 46. IPR bit assignments

Bits	Name	Function
[31:24]	Priority, byte offset 3	Each priority field holds a priority value, 0-255. The lower the value, the greater the priority of the corresponding interrupt. The processor implements only bits[7:4] of each field, bits[3:0] read as zero and ignore writes.
[23:16]	Priority, byte offset 2	
[15:8]	Priority, byte offset 1	
[7:0]	Priority, byte offset 0	

Interrupt priority / Onderbreking prioriteit

- On reset, the Cortex M4 will read two values from vector table. The first one (at offset 0) is the initial stack pointer. The second is the value that is loaded into PC – the address of the reset exception handler (which has the highest priority).

TABLE 15.1
Exception Types and Vector Table

Exception Type	Exception Number	Priority	Vector Address	Caused by…
—	—	—	0x00000000	Top of stack
Reset	1	– 3 (highest)	0x00000004	Reset
NMI	2	– 2	0x00000008	Non-maskable interrupt
Hard fault	3	– 1	0x0000000C	All fault conditions if the

Interrupt priority / Onderbreking prioriteit

- **How do we ensure that no interrupt can occur during critical parts of the program?**

- By setting the appropriate bits in the **Interrupt set-enable registers (NVIC_ISERx)** register we can disable individual interrupts.
- To disable all exceptions *below* a certain priority, you can use the BASEPRI register.
- To disable all exceptions with configurable priority you can use the PRIMASK register.
  - Can be set with `__enable_irq();` and `__disable_irq();`

NVIC Functions / NVIC Funksies

- There are many convenient functions that the NVIC library provides to set these registers.

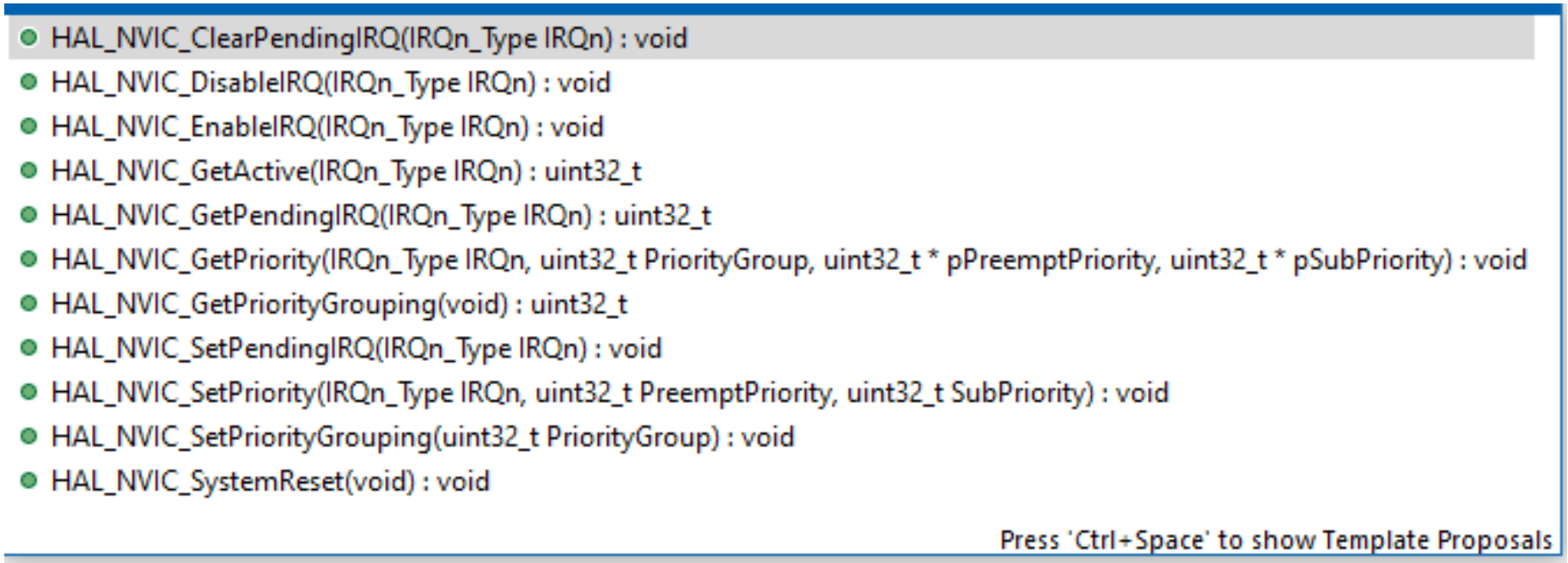
Table 45. CMSIS access NVIC functions

CMSIS function(1)	Description
void NVIC_EnableIRQ(IRQn_Type IRQn)	Enables an interrupt or exception.
void NVIC_DisableIRQ(IRQn_Type IRQn)	Disables an interrupt or exception.
void NVIC_SetPendingIRQ(IRQn_Type IRQn)	Sets the pending status of interrupt or exception to 1.
void NVIC_ClearPendingIRQ(IRQn_Type IRQn)	Clears the pending status of interrupt or exception to 0.
uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn)	Reads the pending status of interrupt or exception. This function returns non-zero value if the pending status is set to 1.
void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)	Sets the priority of an interrupt or exception with configurable priority level to 1.
uint32_t NVIC_GetPriority(IRQn_Type IRQn)	Reads the priority of an interrupt or exception with configurable priority level. This function return the current priority level.

1. The input parameter IRQn is the IRQ number,

NVIC Functions / NVIC Funksies

- The NVIC functions are wrapped by HAL to include some basic validation and error checking of the supplied arguments, but ultimately simply calls the NVIC functions.



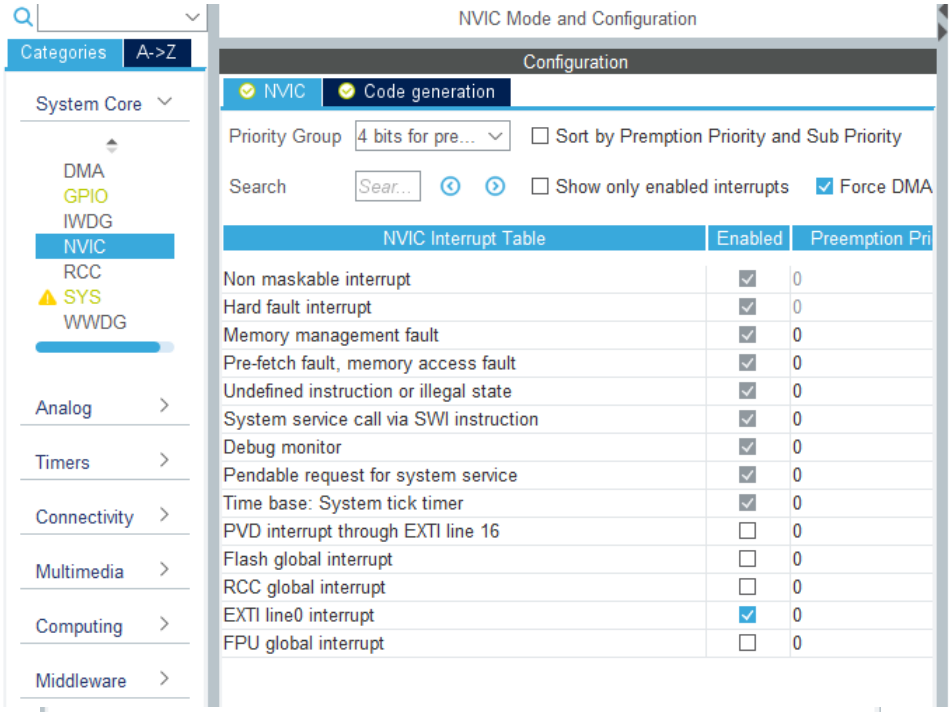
NVIC Functions / NVIC Funksies

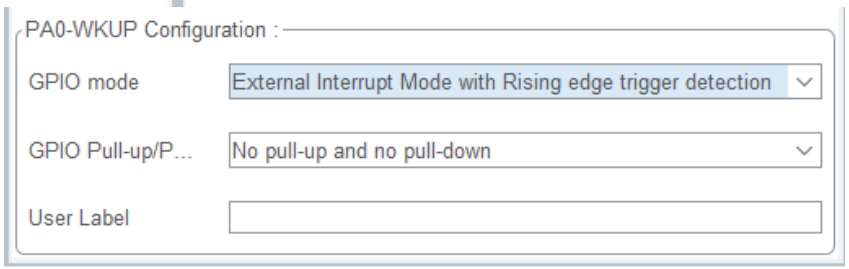
- The NVIC Functions change the physical memory contents of where the registers are mapped.

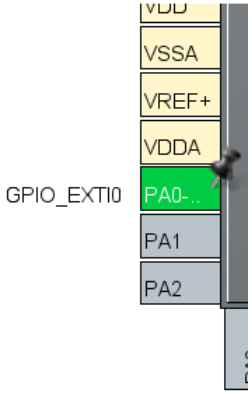
```
/**
  \brief   Set Interrupt Priority
  \details Sets the priority of a device specific interrupt or a processor exception.
           The interrupt number can be positive to specify a device specific interrupt,
           or negative to specify a processor exception.
  \param [in]      IRQn  Interrupt number.
  \param [in]  priority  Priority to set.
  \note    The priority cannot be set for every processor exception.
 */
__STATIC_INLINE void __NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority)
{
  if ((int32_t)(IRQn) >= 0)
  {
    NVIC->IP[((uint32_t)IRQn)]               = (uint8_t)((priority << (8U - __NVIC_PRIO_BITS)) & (uint32_t)0xFFUL);
  }
  else
  {
    SCB->SHP[(((uint32_t)IRQn) & 0xFUL)-4UL] = (uint8_t)((priority << (8U - __NVIC_PRIO_BITS)) & (uint32_t)0xFFUL);
  }
}
```

High-level setting up of interrupts

- Usually we simply use the STM32CubeIDE to edit the .ioc file which automatically generates the interrupt code for us.







Generate code

main.c

```
/*Configure GPIO pin : PA0 */
GPIO_InitStruct.Pin = GPIO_PIN_0;
GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

/* EXTI interrupt init*/
HAL_NVIC_SetPriority(EXTI0_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(EXTI0_IRQn);
```

High-level setting up of interrupts

- The IDE also generates a C function where we can write the Interrupt Handler in stm32f4xx_it.c
- This start address of this this function is automatically placed at the correct place in the vector table in startup_stm32f411vertx.s

```
/**
  * @brief This function handles EXTI line0 interrupt.
  */
void EXTI0_IRQHandler(void)
{
  /* USER CODE BEGIN EXTI0_IRQn 0 */

  /* USER CODE END EXTI0_IRQn 0 */
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
  /* USER CODE BEGIN EXTI0_IRQn 1 */

  /* USER CODE END EXTI0_IRQn 1 */
}
```

```
/* External Interrupts */
.word     WWDG_IRQHandler                   /* Window WatchDog              */
.word     PVD_IRQHandler                    /* PVD through EXTI Line detection */
.word     TAMP_STAMP_IRQHandler             /* Tamper and TimeStamps through the EXTI line */
.word     RTC_WKUP_IRQHandler               /* RTC Wakeup through the EXTI line */
.word     FLASH_IRQHandler                  /* FLASH                        */
.word     RCC_IRQHandler                    /* RCC                          */
.word     EXTI0_IRQHandler                  /* EXTI Line0                   */
.word     EXTI1_IRQHandler                  /* EXTI Line1                   */
.word     EXTI2_IRQHandler                  /* EXTI Line2                   */
.word     EXTI3_IRQHandler                  /* EXTI Line3                   */
.word     EXTI4_IRQHandler                  /* EXTI Line4                   */
```