Session 4/4

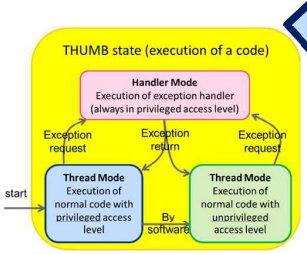
Microprocessor 2 Course

Session 4: Exception mechanisms and NVIC

Version STM32L476

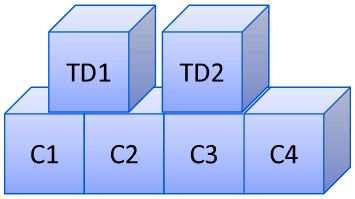
(and microcontrollers)

2024-v1



ENSEA S7

Laurent MONCHAL



Session Menu

- STM32 **interrupts**
  - Switching when processing an interrupt
  - The Interrupt Controller (NVIC)
    - Different states of an interrupt
    - Setting of interrupts
      - Allow / prohibit an interrupt
      - Priorities
      - Interrupt masking
      - Deleting an interrupt request
    - Implementation of NVIC
  - Mode and access level

Response to Exercise Session 3

- Where is the first instruction executed during interrupt processing of a request from the EXTI1 device? (VTOR is 0x08000000)

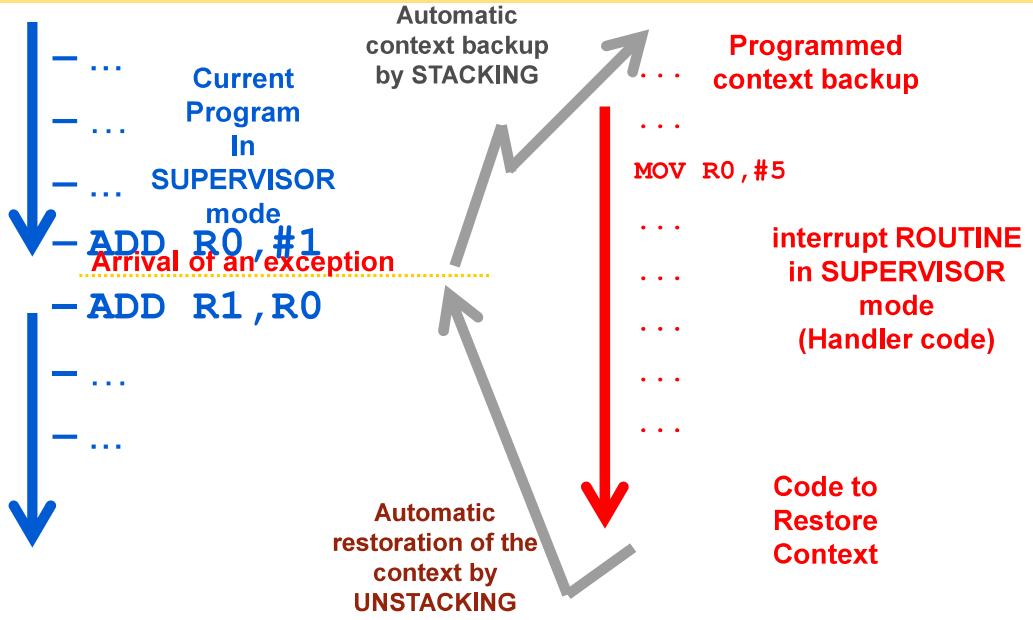
- According to the slides of session 2, the EXTI1 device generates an interrupt request identified by the number 7.
- In the vector table, the vector associated with EXTI1 is therefore at the address VTOR+(16x4) + (7x4) =VTOR+ 92 =0x08000000+0x5C
- The vector of EXTI1 is read here at address 0x0800005C: it is 0x0800ICI0.
- The first instruction executed during a request from the EXTI1 device (if accepted by NVIC) is therefore located at address **0x08001C10**.

Address	Data (32 bits)
0x08000048	0x08000204
0x0800004C	0x080013F0
0x08000050	0x08000408
0x08000054	0x08000B24
0x08000058	0x08000A00
0x0800005C	0x08001C10
0x08000060	0x0800220C

Microcontroller resources

- ROM
  - To store the code (algorithm)
    - 2 or 4 bytes for one ASM instruction
    - 2 to 12 bytes for one C instruction
  - To store constants
  - 512 kB (mega bytes) for STM32F401RE
  - 1 MB (mega bytes) for STM32F746 or STM32L476
- RAM
  - For the variables
    - Global variables: directly in an allocated part of the RAM
    - Local variables: in the Stack (allocated in the RAM)
  - 96 kB (kilo bytes) for STM32F401RE
  - 128 kB (kilo bytes) for STM32L476
  - 320 kB (kilo bytes) for STM32F746
- CPU
  - To execute ASM instructions (from functions in C)
  - 32-bit registers : R0 to R15 for variables and CPU execution

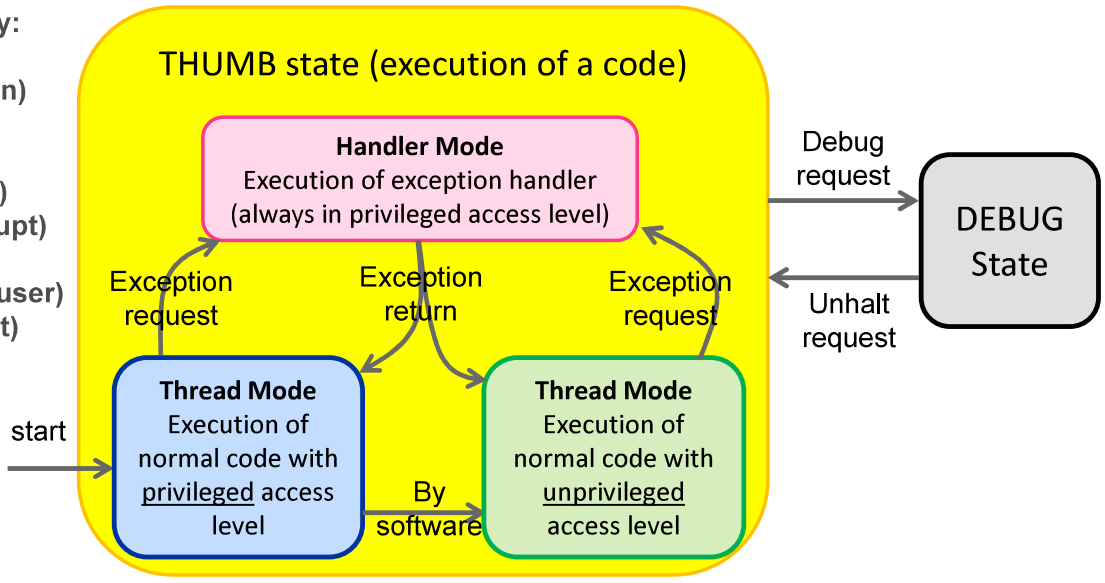
Scenario for handling an exception



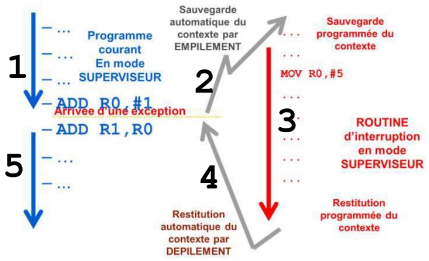
Operation State, Execution Mode and Access Level

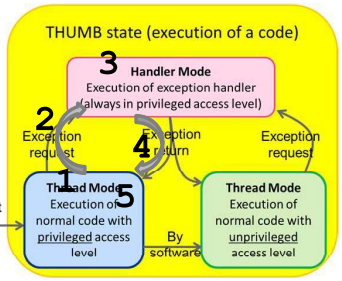
The CPU is defined by:

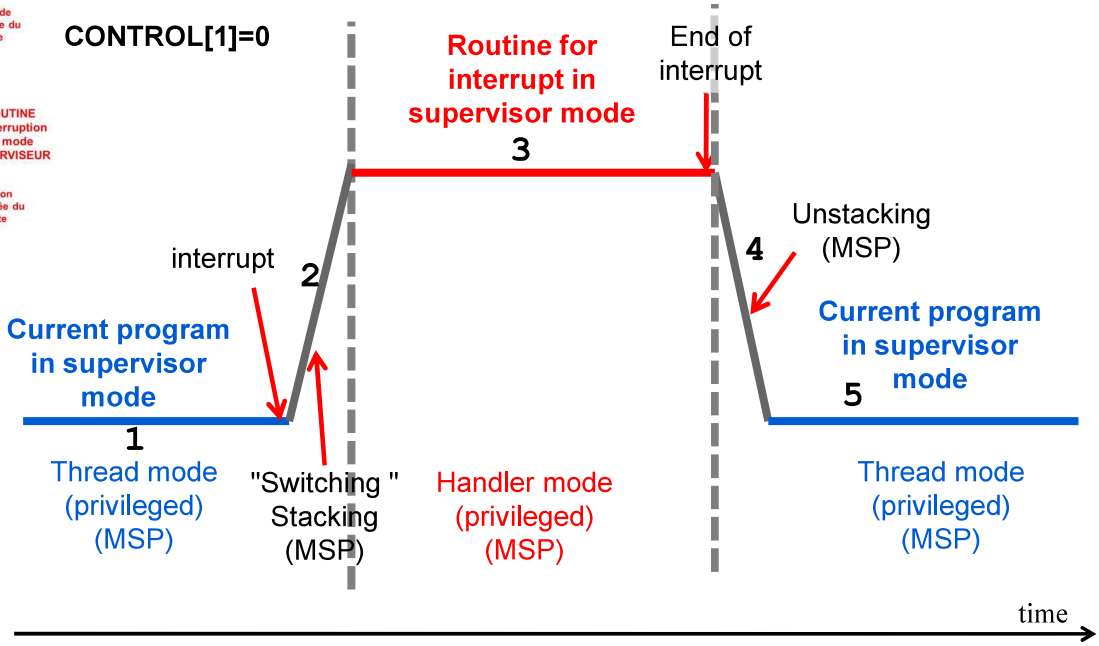
- A STATE
  - THUMB (execution)
  - DEBUG
- A MODE
  - THREAD (Normal)
  - HANDLER (Interrupt)
- An ACCESS LEVEL
  - UNPRIVILEGED (user)
  - PRIVILEGED (root)



Evolution of the modes when processing an exception







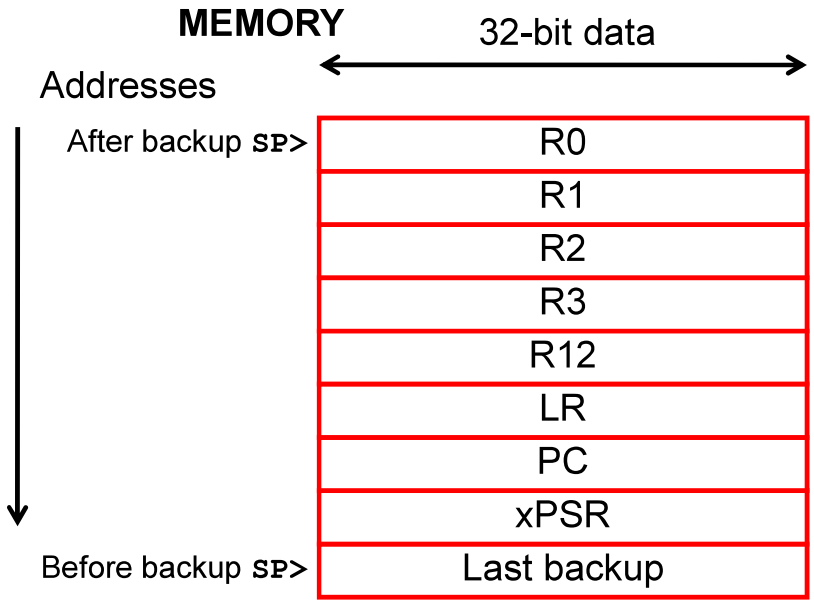
Automatic µP actions during switching

- When an N-number exception or interrupt occurs, the following steps are performed when switching between the current program and the code Handler:
  - **Context backup** by saving 8 registers in the stack (with evolution of the SP stack pointer)
  - **Recovery of the vector** corresponding to the running exception or interrupt
  - **Update of the LR** link register and the **PC** register

Automatic context saving

- 8 registers are stored in the current stack
  - R0 to R3, R12, LR, PC and PSR
  - This choice was made because these registers constitute a common (but incomplete) minimal context.
  - Selecting the stack pointer to perform the backup:
    - MSP if the interrupted program is in supervisor mode
    - PSP if the interrupted program is in user mode
- What about the other resources?
  - In C language: compiler strategy => Low level instructions and registers
  - The other registers are those saved by instructions when calling a C function according to the AAPCS standard (ARM Architecture Procedure Call Standard).
  - Thus the interrupt processing program is a C function according to the AAPCS standard because it allows the backup of the other registers.

Registers saved in a stack (automatic)



Updated registers

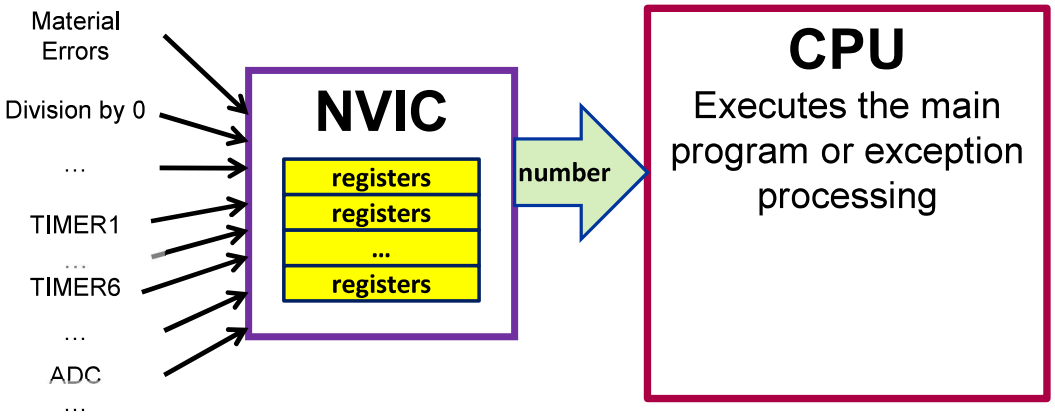
- When responding to an exception or interrupt, certain registers are updated
  - 8 registers being stacked, the concerned stack register (**MSP or PSP**) is updated: SP SP-8x4
  - The IPSR part of the **xPSR** is updated with the exception or interrupt number.
  - The program pointer (**PC**) is of course set to the value corresponding to the exception routine that is going to be started (so that the exception program is started).
  - The Link Register (**LR**) loads a special value: EXC_RETURN (=0xFFFFFFFFX)
    - That's what allows you to write an interrupt routine as a simple function
    - Indeed the return is always made by the instruction BX LR (or equivalent).

The return from interrupt handler code

- This is done by loading the value of LR into PC:
  - By BX LR
  - Or POP {PC}
- The special value of LR (EXC_RETURN=0xFFFFFFFX) then triggers:
  - Restoring the 8 registers (R0 to R3, R12, LR, PC and PSR)
  - Updating of NVIC records for the interrupt whose routine has just ended

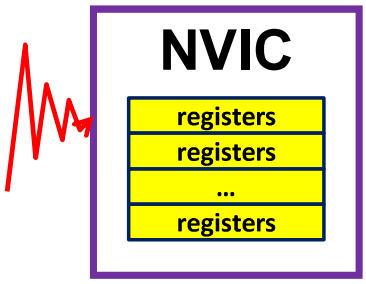
Between peripherals and CPU: NVIC

- The CPU "delegates" the management of interrupts to the NVIC block.



The role and means of NVIC

An exception or interrupt identifiable issues a **request**



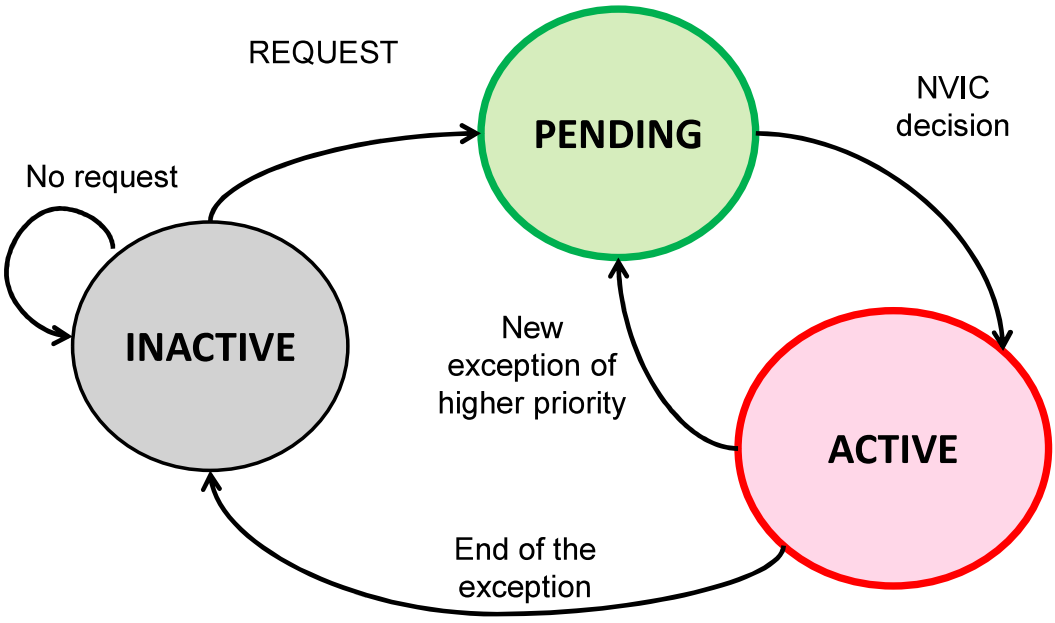
The NVIC checks whether the interrupt is PROHIBITED or AUTHORIZED => it transmits only authorized requests
Records: `NVIC->ISER[x]`

In case of multiple requests, the NVIC checks the priority level of the interrupts concerned.
Records: `NVIC->IPR[x]`

NVIC Decision

- If the request originates from a DISABLED interrupt
  - Then the interrupt goes into standby mode (PENDING).
- If the request originates from an AUTHORIZED interrupt
  - If there are no interrupts during processing or if the request has a higher priority than the interrupt being processed
    - Then the interrupt from which the request originated goes into ACTIVE mode.
  - Otherwise
    - The interrupt goes into PENDING mode

Interrupt states



Main NVIC registers

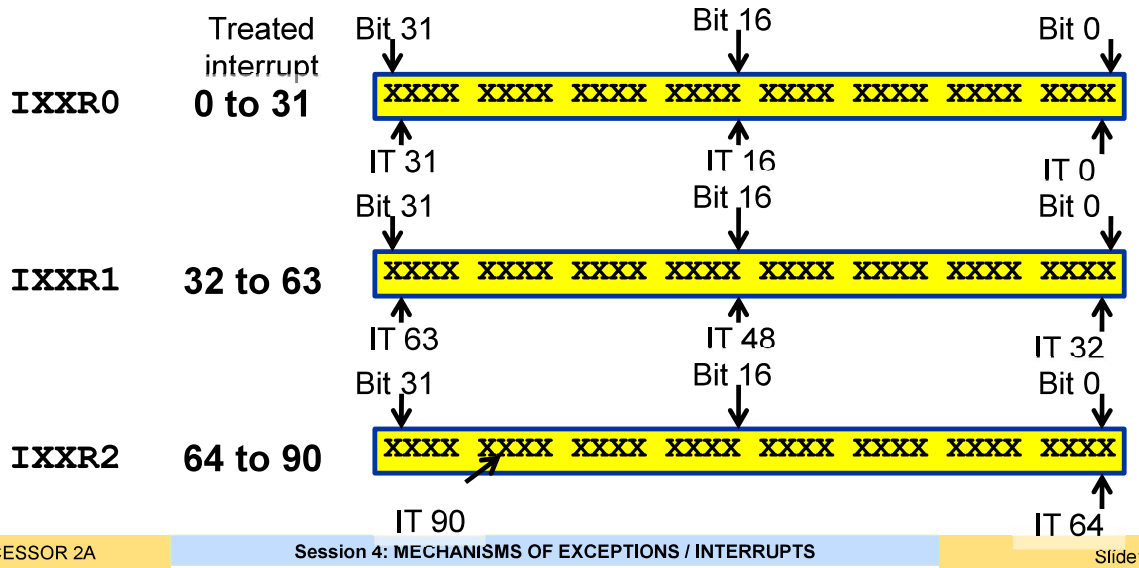
- These registers can be grouped according to interrupt status management:
  - For AUTHORIZATIONS:
    - **NVIC_ISERx and NVIC_ICERx**
  - For the PENDING state:
    - **NVIC_ISPRx and NVIC_ICPRx**
  - To know the status: ACTIVE or not
    - **NVIC_IABRx**
  - To set the priority:
    - **NVIC_IPx**

Authorization of interrupts

- NVIC_ISER0 to NVIC_ISER2
  - Interrupt Set Enable Register
  - EnC: `NVIC->ISER[0]` to NVIC->ISER`[2]`
  - Each bit corresponds to an interrupt
  - 1 32-bit register ↔ management of 32 interrupts
  - Write
    - 1: **allows** the corresponding interrupt
    - 0: no effect
    - The use of masks is not necessary.
  - Reading
    - 1: the corresponding interrupt is allowed
    - 0: The corresponding interrupt is not allowed.
    - The use of masks is not necessary.

Correspondence between registers and interrupts

- The registers of types NVIC_IXXRn are used to manage requests, wait states and interrupt activity for interrupts numbered 0 to 90.



Disabling interrupts

- NVIC_ICER0 to NVIC_ICER2
  - Interrupt Clear Enable Register
  - EnC: NVIC->ICER[0] to NVIC->ICER[2]
  - Each bit corresponds to an interrupt
  - Write
    - 0: no effect
    - 1: **prohibits** the corresponding interrupt
  - Reading
    - 0: The corresponding interrupt is not allowed.
    - 1: the corresponding interrupt is allowed

Forcing in pending state

- NVIC_ISPR0 to NVIC_ISPR2
  - Interrupt Set-Pending Register
  - EnC: `NVIC->ISPR[0] to` NVIC->ISPR`[2]`
  - Each bit corresponds to an interrupt to be forced in pending state.
  - Write
    - 0: no effect
    - 1: **forces** the corresponding interrupt to the **pending state.**
  - Reading
    - 0: The corresponding interrupt is not pending.
    - 1: the corresponding interrupt is pending

Canceling a wait state

- NVIC_ICPR0 to NVIC_ICPR2
  - Interrupt Clear-Pending Register
  - EnC: NVIC->ICPR`[0] to` NVIC->ICPR`[2]`
  - Each bit corresponds to an interrupt whose waiting state is cancelled (thus becoming inactive).
  - Write
    - 0: no effect
    - 1: Cancel **standby status**
  - Reading
    - 0: The corresponding interrupt is not pending.
    - 1: the corresponding interrupt is pending

Registers : active interrupts

- NVIC_IABR0 to NVIC_IABR2
  - Interrupt Active Bit Register
  - EnC: **`NVIC->IABR[0] to`** NVIC->IABR**`[2]`**
  - This read-only register identifies active interrupts.
  - Write
    - Unauthorized
  - Reading
    - 0: The corresponding interrupt is not active.
    - 1: the corresponding interrupt is active (or active and pending).

Need for priorities

- Some events have a higher priority than others and require a quick response.
- Example in a plane, which event has the highest priority?
  - Displaying a message on a screen
  - Transmit a thrust order on the engines

Priority levels

- Exceptions/Disruptions are classified among them by priority levels
  - The priorities are materialized by integers (thus positive or negative values).
  - The **larger the integer, the lower the priority**:
    - 1 is a higher priority than 64
    - -2 has higher priority than 1
    - Negative priorities are only used for the first 3 exceptions
    - Priority Level=Rank
  - They allow the design of a pre-emptive system

Priority management and pre-emption

- Establishment of a priority ranking
- A higher-priority interrupt interrupts a lower-priority interrupt in progress: this is **pre-emption.**
- A lower-priority interrupt cannot interrupt a higher-priority interrupt.
- Preemption occurs when the processing of Interrupt 1 is in progress and is stopped to allow the processing of Interrupt 2 to be executed.
- It is said that interrupt 2 preempts interrupt 1.
- This pre-emption is implemented by initializing the NVIC registers: NVIC_IPRn

Priority Level Registers

- NVIC_IPR0 to NVIC_IPR23
  - Interrupt Priority Register
  - For interrupts: 8 bit unsigned integers! (so positive)
  - In C: **`NVIC->IP[0] to NVIC->IP[96]`**
  - **`NVIC->IP[n]`** is an 8-bit value

NVIC_IPRn registers

IPR0

32-bit register

IP[3]	IP[2]	IP[1]	IP [0]
xxxx0000	xxxx0000	xxxx0000	xxxx0000

RPI1

32-bit register

IP[7]	IP [6]	IP [5]	IP[4]
xxxx0000	xxxx0000	xxxx0000	xxxx0000

IPRn

32-bit register

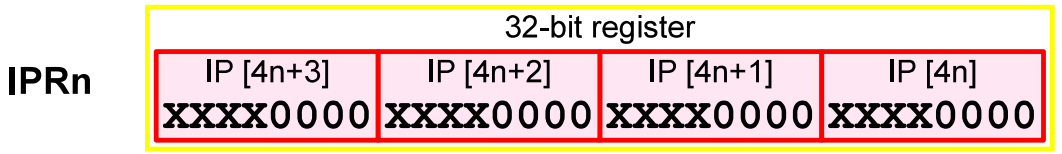
IP [4n+3]	IP [4n+2]	IP [4n+1]	IP [4n]
xxxx0000	xxxx0000	xxxx0000	xxxx0000

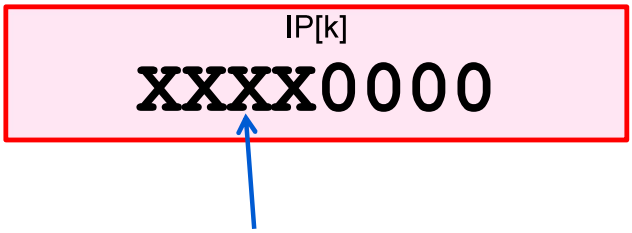
IPR16

32-bit register

IP [67]	IP [66]	IP [65]	IP [64]
xxxx0000	xxxx0000	xxxx0000	xxxx0000

Detail of an 8-bit priority value

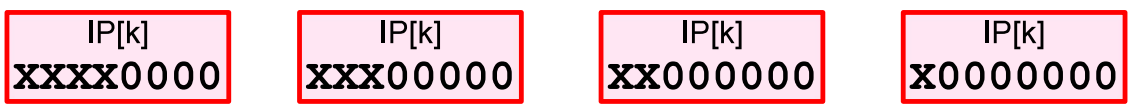




The 4 bits can be used to define 16 different combinations of interrupt levels

Size of the priority group

- The SCB_AIRCR register contains a PRIGROUP field which allows to modify the number of bits to be taken into account for the priority.
- A priority can be read on 4, 3, 2 or 1 bits



Priority and sub-priority

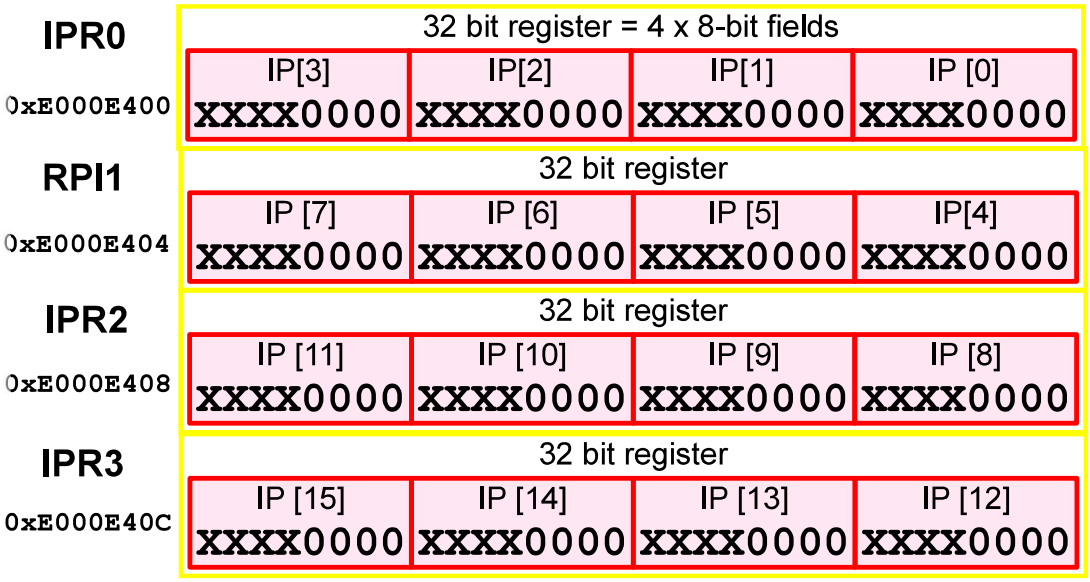
- The STM32 uses 4 bits to set the priority
- On these 4 bits
  - the upper part allows to define the priority of **preemption**
  - The lower part allows to define a sub-priority in case of equality when 2 interrupts appear at the same time.
  - In case of equality of the sub-priorities, the order of the vector table takes precedence.

Location of NVIC registers

- The registers of the NVIC device are located at address `0xE000 E100.`
- NVIC_ISER0 in `0xE000 E100`
  - So NVIC_ISER1 in `0xE000 E104`
  - So NVIC_ISER2 in `0xE000 E108`
- NVIC_ICER0 in `0xE000 E180`
- NVIC_ISPR0 in `0xE000 E200`
- NVIC_ICPR0 in `0xE000 E280`
- NVIC_IABR0 in `0xE000 E300`
- NVIC_IPR0 en `0xE000 E400`
- NVIC_STIR in `0xE000 EF00`

Example: implementation of NVIC_IP[n]

The priorities are defined in the 4 MSBs of each field.



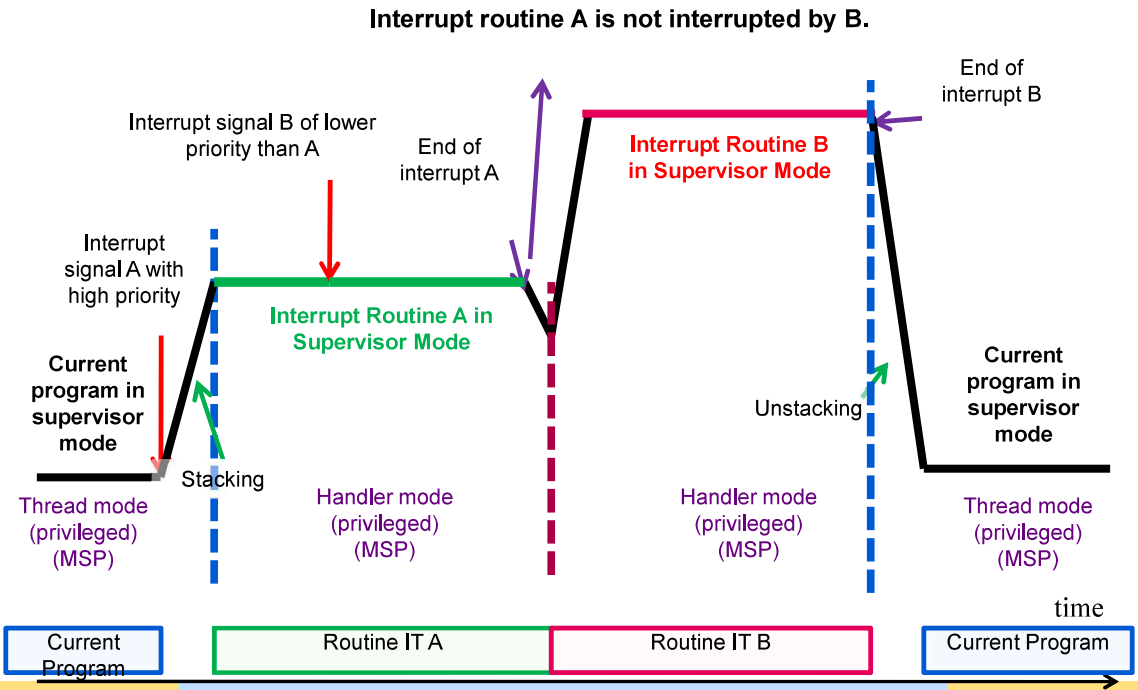
The 4 LSBs in each field are: 0000

With HAL, it is possible to read or write 8-bit fiels IP[n] by using "`NVIC->IP[n]`"

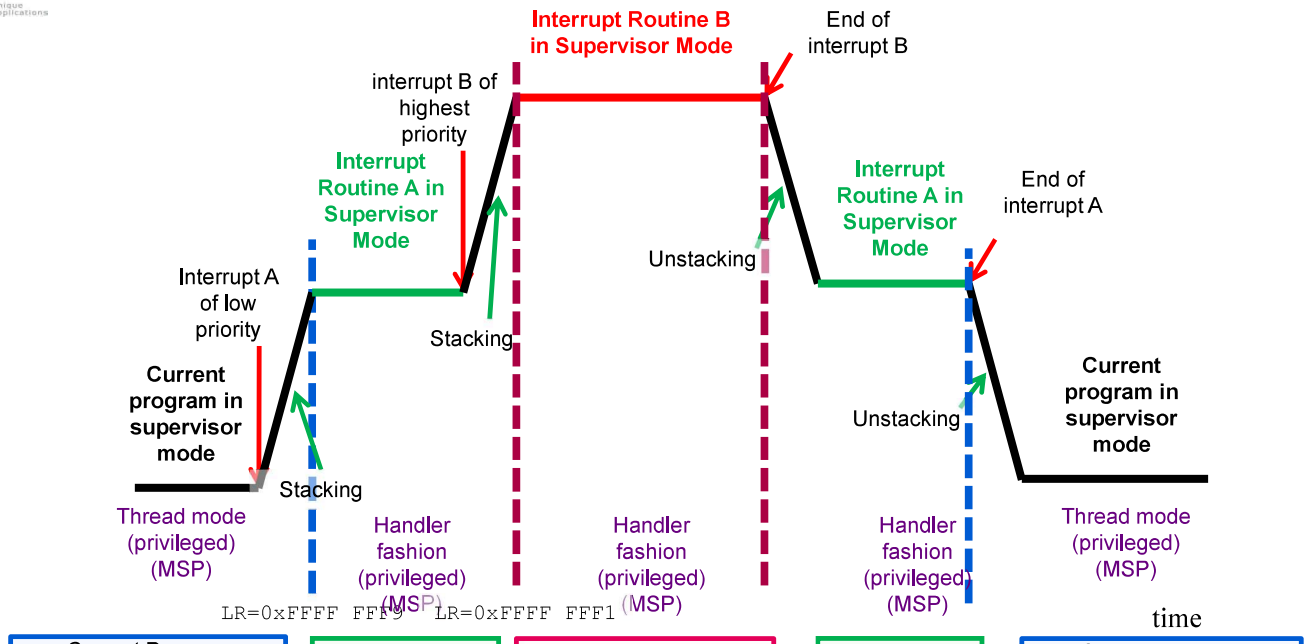
HAL functions for NVIC

- The NVIC is programmable using C functions described in "**User Manual, Description of STM32FXxx HAL drivers**".
- Programming the NVIC:
  - To set the priority group : `void` **`HAL_NVIC_SetPriorityGrouping`**`(uint32_t PriorityGroup)`
    - Example: `HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4)`
  - To set the priority level of an interrupt: `void` **`HAL_NVIC_SetPriority`** `(IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t SubPriority)`
  - To allow an interrupt at the NVIC: `void` **`HAL_NVIC_EnableIRQ`**`(IRQn_Type IRQn)`
  - To prohibit an interrupt to the NVIC: `void` **`HAL_NVIC_DisableIRQ`**`(IRQn_Type IRQn)`

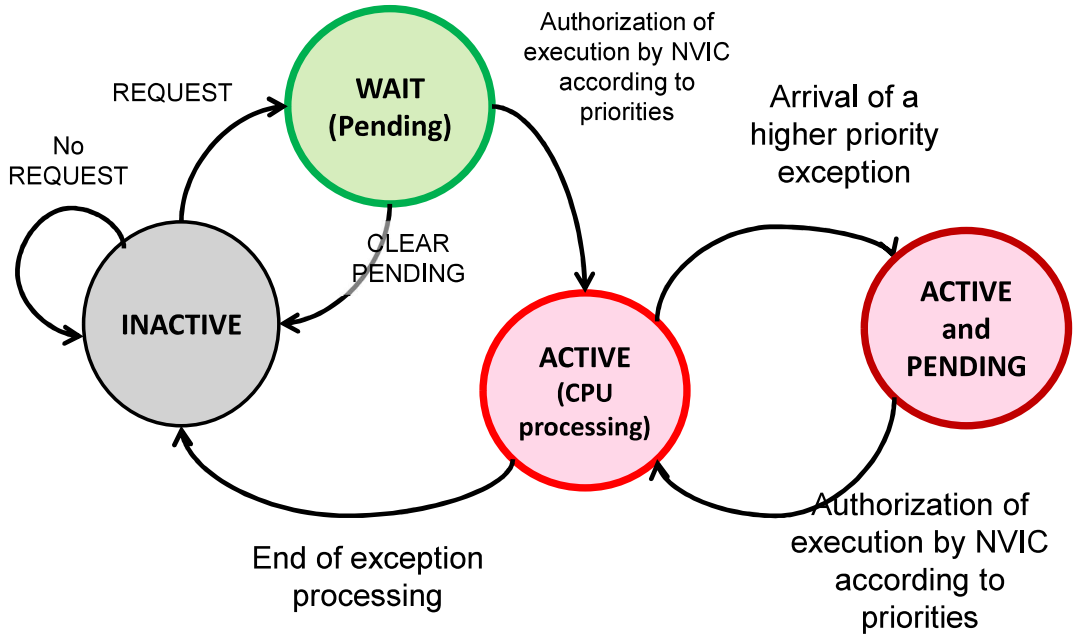
Detail when responding to an interrupt of lower priority



Detail when responding to a higher priority interrupt

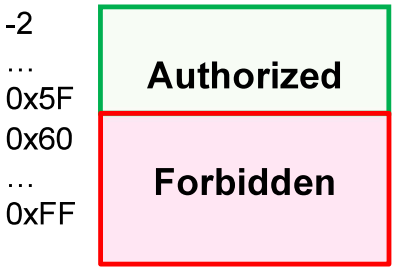


The states of an interrupt or exception



Principle of interrupt masking

- The priority mask is an integer value that allows to **prohibit** all interrupts whose level is higher or equal to the mask (cad less priority or equal).
- This therefore prohibits all interrupts that have a priority value greater than or equal to
  - Example: If the mask is set to 0x60, then all interrupts with a priority value of 0x60 and higher are prohibited.
  - Mask set by CPU



The masking register: BASEPRI

- The BASEPRI register contains the value of the mask
  - It cannot be written or modified in user mode!
  - Example:
    - `MOV RO,#0x60`
    - `MSR BASEPRI,R0`
    - This makes it possible to prohibit interrupts of priority values from 0x60 to 0xFF.

Hide almost all interrupts

- **PRIMASK** Register
  - Register 1 bit!
  - 1: Prohibits all interrupts and exceptions except non-maskable interrupts (NMI) and hard faults.
    - changes the priority level to 0
    - Useful in a critical task
    - `MOV R0,#1`
    - `MSR PRIMASK,R0`

Priority	Type of priority	Acronym
-2	fixed	NMI
-1	fixed	HardFault
0	settable	MemManage
1	settable	BusFault
2	settable	UsageFault

Authorized

Forbidden

Hide interrupts

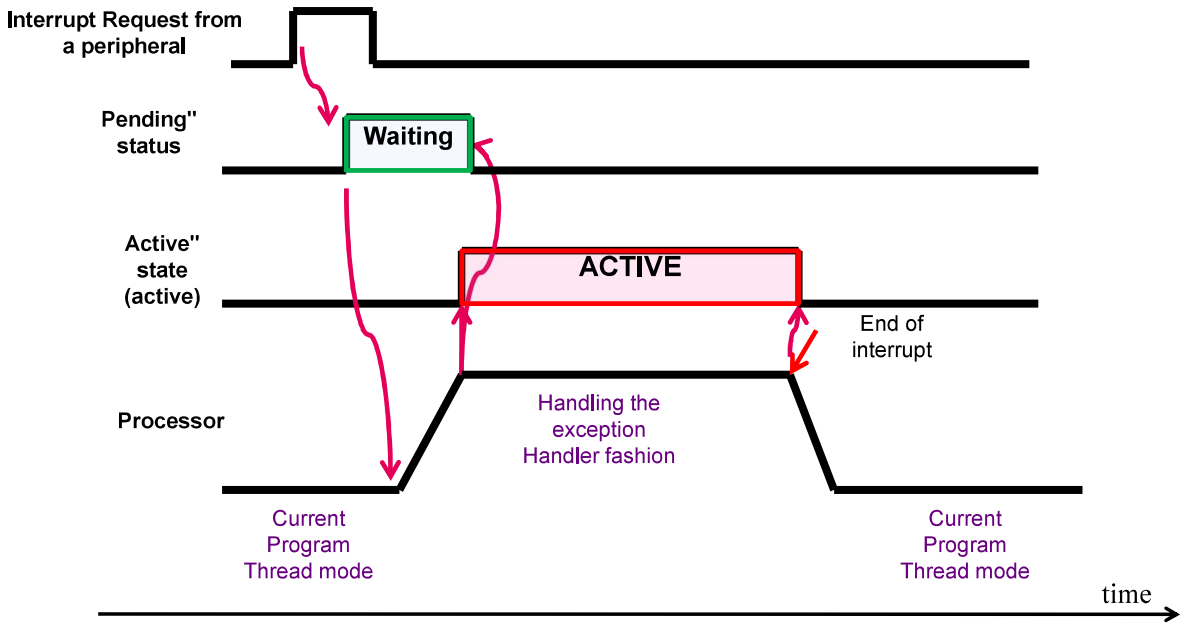
- **FAULTMASK** register
  - Register with 1 bit!
  - 1: Prohibits all interrupts and exceptions except non-maskable interrupts (NMI).
    - changes the priority level to -1
    - Useful in a critical task
    - `MOV R0,#1`
    - `MSR FAULTMASK,R0`

Priority	Type of priority	Acronym
-2	fixed	NMI
-1	fixed	HardFault
0	settable	MemManage
1	settable	BusFault
2	settable	UsageFault

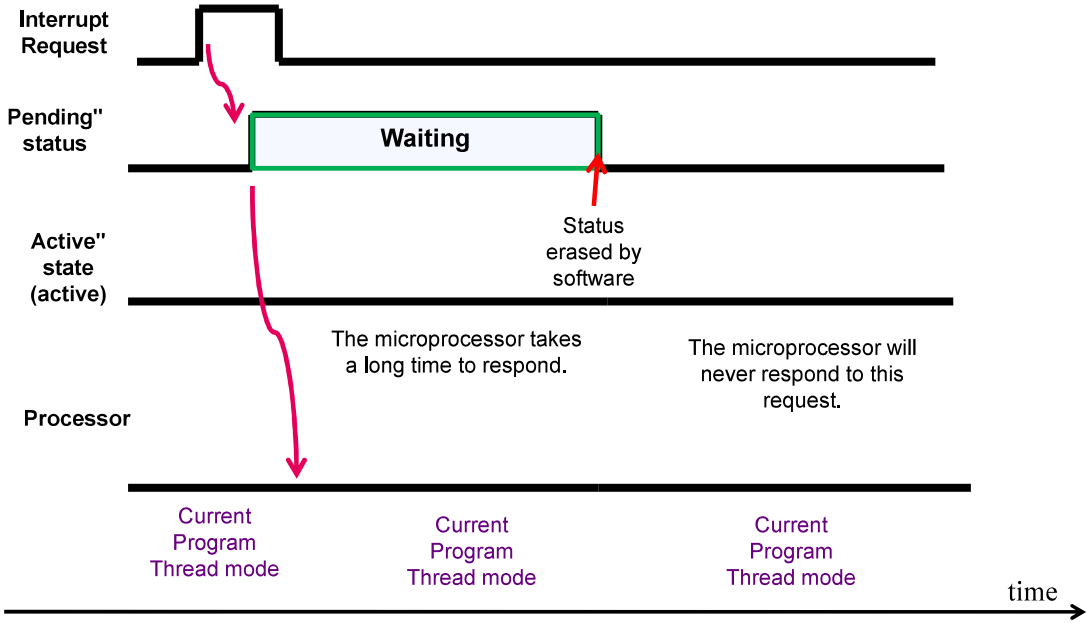
Authorized

Forbidden

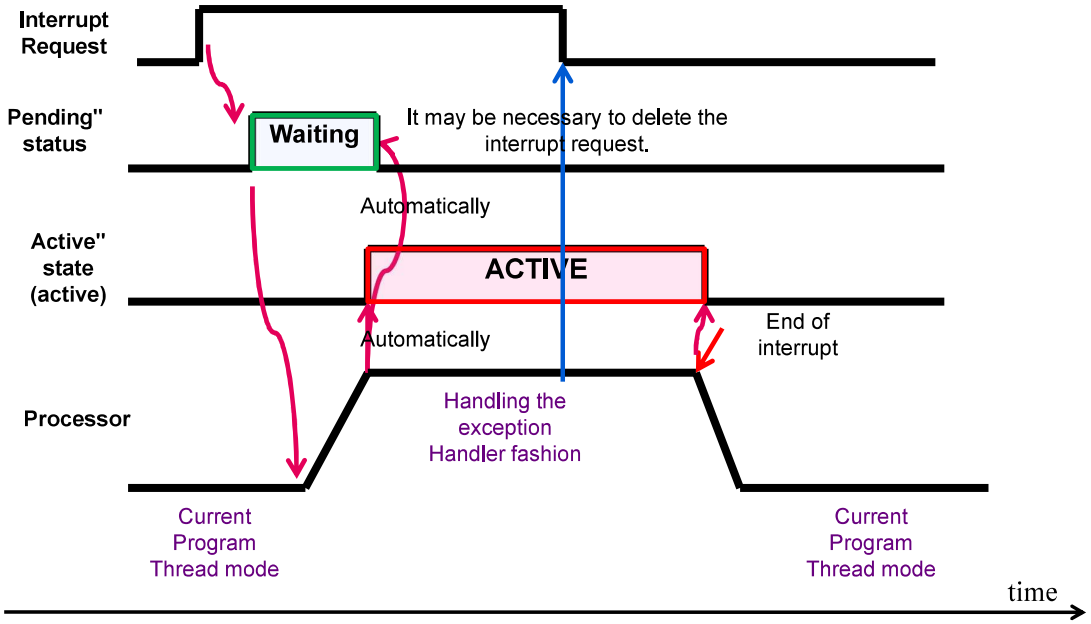
The states of an interrupt on a request



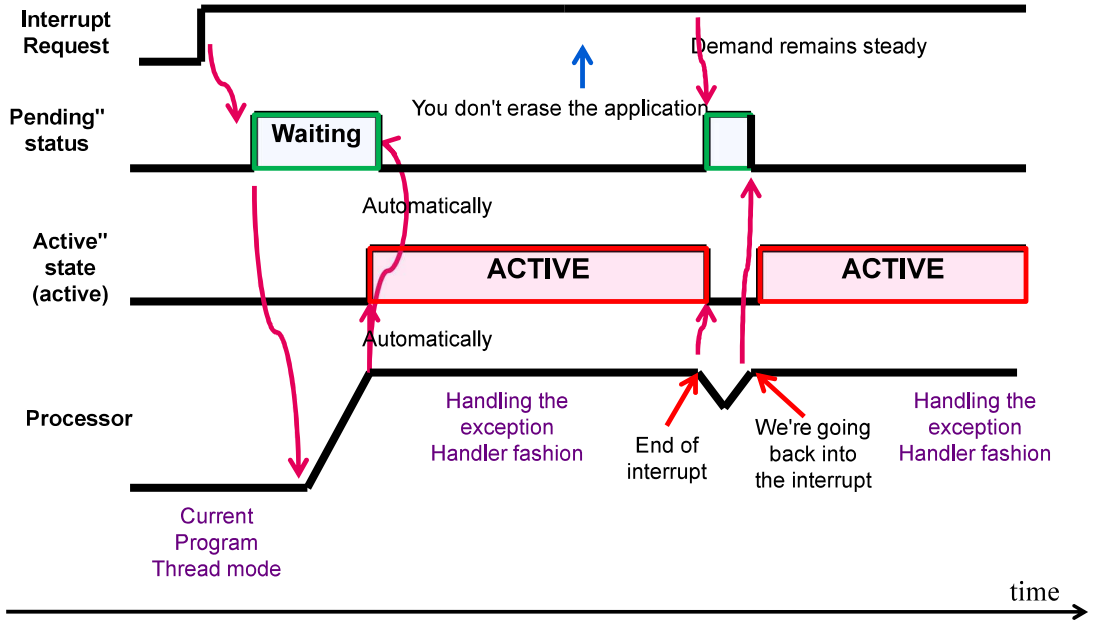
Software erased pending state before processor response



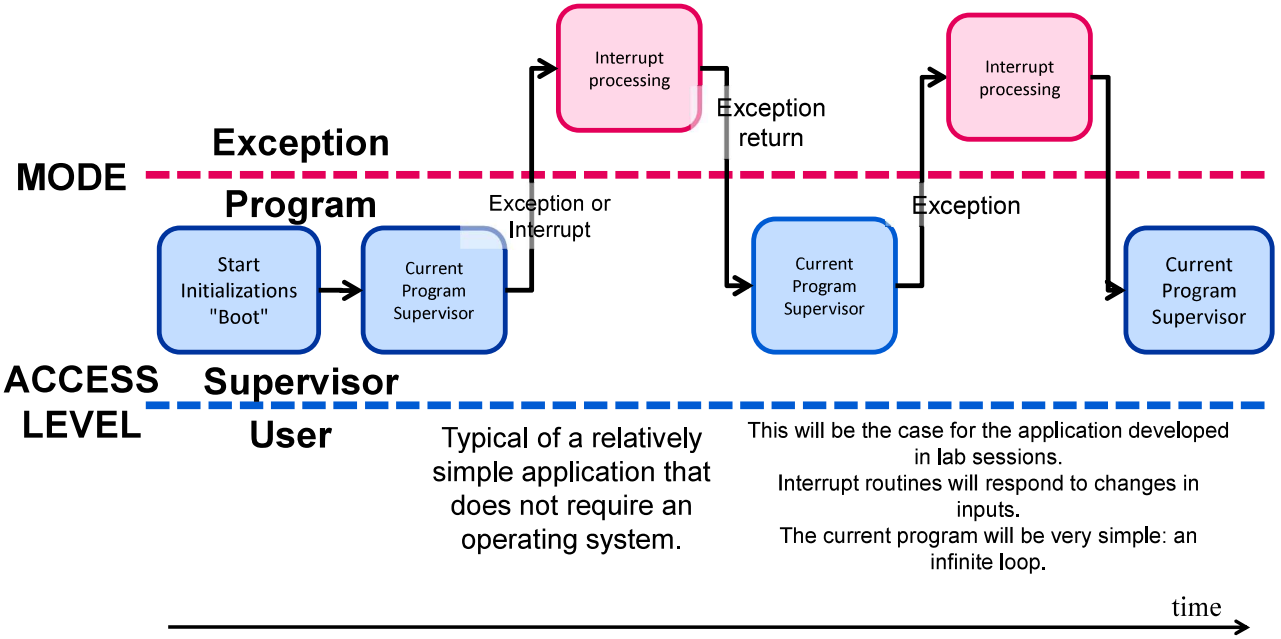
Erasing demand (classic scenario)



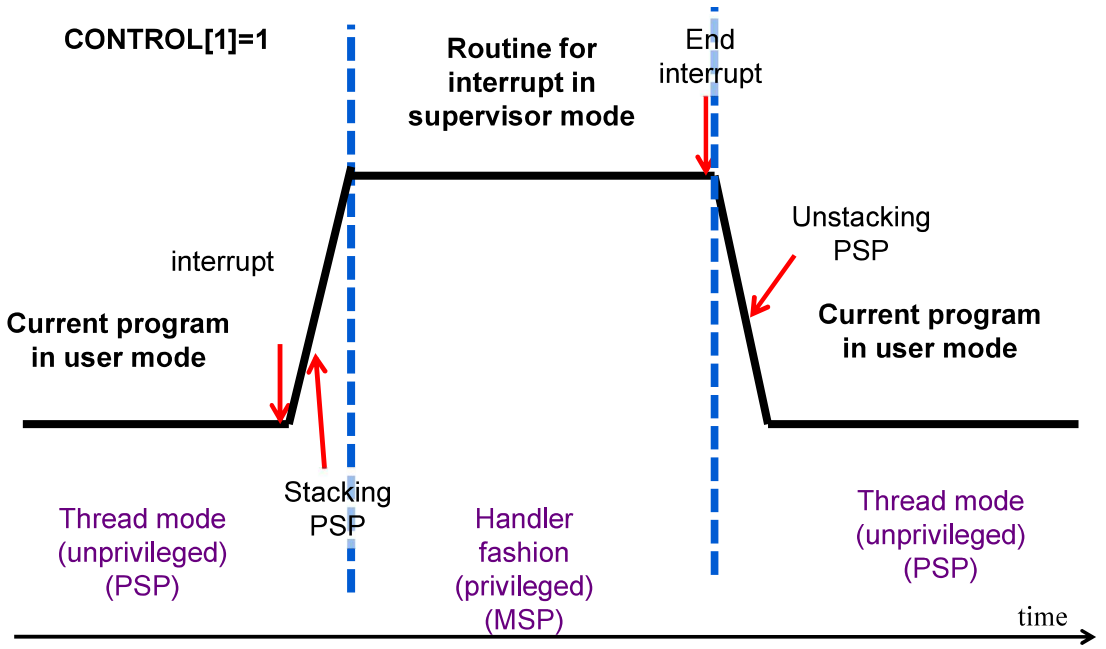
Consequence of not erasing the interrupt request (if the request is maintained)



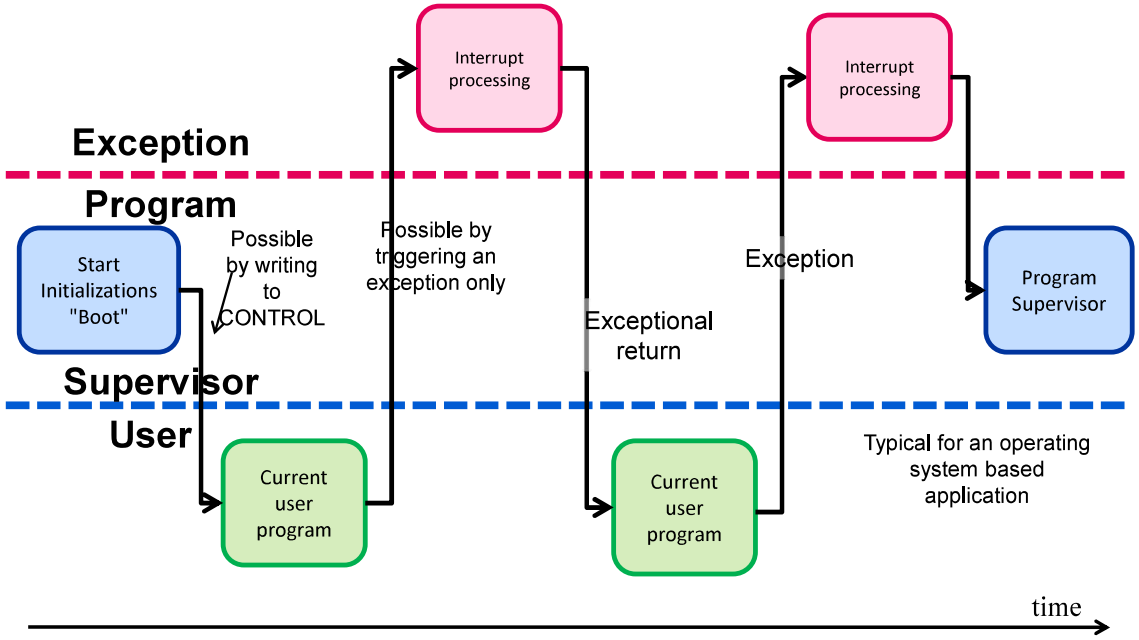
Example of code switching



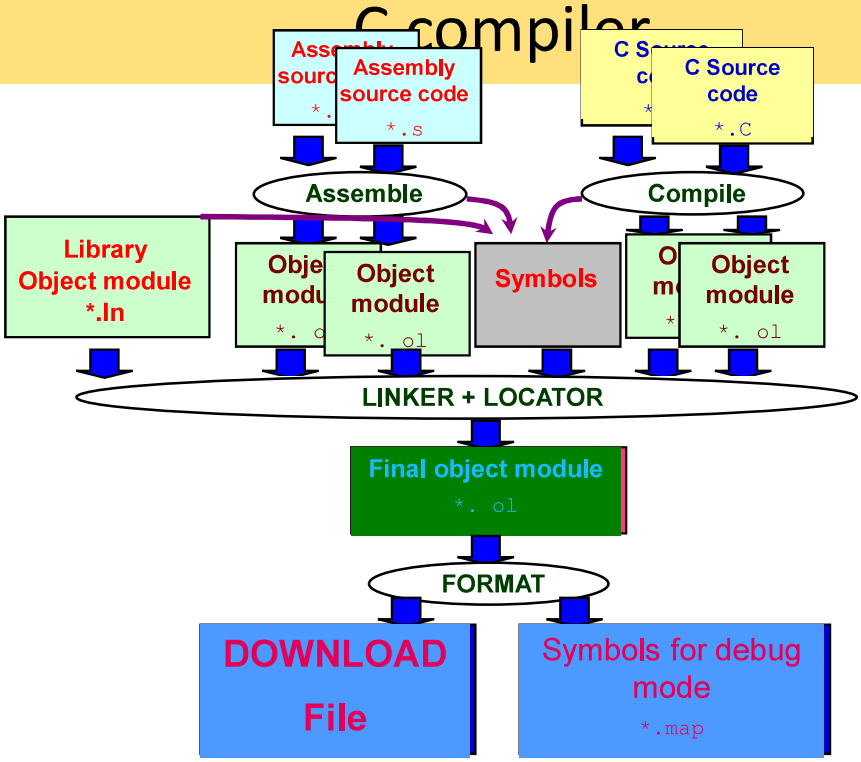
Switching from unprivileged level



Example of time switching



C compiler



Reminder on priority level

- Each interrupt has its own priority level
  - It is represented by an integer value
  - The lower the value, the higher the priority level.
  - It's unnatural (I'll grant you that).
  - It's only natural if you think of the priority level as a **waiting line.**
  - Therefore, the priority level should not be confused with the value of priority

The states of an interrupt (reminders)

- An exception or interrupt can be in different states:
  - **Inactive**: no request has been sent
  - Pending: a request has been sent but the corresponding routine is not running because an exception of higher priority is being served.
  - **Active**: a request has been sent and the corresponding routine is running.

interrupt request management (reminder)

- A request from a device
  - End of counting for a timer (overflow)
  - End of conversion for ADC (EOC) .
- This request is maintained to the NVIC until a command is sent to the device to remove it.
- NVIC "registers" the application, processes it...
- The device can remove the request:
  - If writing a particular bit (for Timer)
  - If reading the converted data (for ADC)...

Software deletion of a pending state (recall)

- It is possible to clear a pending state by writing to the NVIC interrupt control registers:
  - NVIC_ICPRx: in writing
    - Writing a 0: no effect
    - Write a 1: Clears the pending status of the corresponding interrupt.
  - NVIC_ICPRx: read
    - Reading a 0: interrupt not pending
    - Playing a 1: Interrupt in pending mode

Stack pointer(s)

R13 (MSP)	R13 (PSP)	Stack pointers: SP

- 2 registers for 2 stacks
  - MSP (Main Stack Pointer)
    - In **privileged** access (supervisor): for OS or exception handling
    - Default at startup
  - PSP (Process Stack pointer)
    - In **non-privileged** access (user): user code execution
  - Each pointer evolves from 4 to 4 starting from 0: 0, 4, 8, 12, 16...because it goes from word to word (32 bits = 4 bytes) the last 2 bits are always at 0: `00000`, `00100`, `01000`, `01100`, `10000...`

Modes of operation

- The Cortex M4 / M7 have 2 levels of access:
  - User (unprivileged)
  - Supervisor (privileged)
- What's that all about?
  - To enable the creation of operating systems
    - A program manages and controls the material resources with full intervention rights (supervisor)
    - It accepts that users run programs but with limited rights (user).

Stack and mode

- Stack access:
  - MSP only in supervisor mode
  - PSP in user and supervisor mode
- What's that all about?
  - To prevent a program from accessing the supervisor stack in user mode (by mistake or with the intention of harming it)

2 execution states

- "Debug state"
  - Debugging mode that allows step-by-step or breakpoint execution
  - In this state the microprocessor can be switched off.
- "Thumb state"
  - Code execution: the microprocessor never stops.

2 modes of execution

- "Handler mode"
  - This mode is used when executing the program of an interrupt or exception.
- "Thread mode"
  - Execution other than an exception or interrupt: classic execution of a program.

2 levels of access (or priority)

- "supervisor mode"
  - Access to all records and instructions
- "user mode".
  - Limited access to certain registers and instructions (in particular, does not allow changing the priority level)

Switches

- An exception always runs in privileged mode
- When an exception is returned, the program returns to the mode in which it was previously running.
- A user cannot change modes
- The PSP stack is dedicated to user mode and the MSP stack to supervisor mode.

Memory ressources of a STM32

- Ressources for code
  - In ROM Flash: 2Mbytes for STM32H7A3
  - For code and constants
- Ressources for variables
  - In RAM (SRAM):
    - For STM32H7A3: 1.18 Mbytes for user RAM
  - STACK: user heap stack
    - For local variables
    - With STM32CubeIDE: (1.5 Kbytes chosen by STM32CubeIDE)
  - Data block: user data
    - For global variables (static only!)

Know-how at the end of session 4

- Know how to program the NVIC to manage interrupts by writing to the NVIC registers and deduce the implementation of the NVIC.
- Example
  - Allow interrupt from an EXTI1 request by writing to a NVIC register (NVIC->)
  - Which bit is set to 1 (and at which address?) to get authorization?
  - Set its pre-emption priority to 8 per entry in a NVIC register
  - What should be written in the memory that confirms that this action has been carried out?
  - Write the calls to the HAL functions allowing to carry out these 2 actions

For the Labs / Season 2024

- Read the first part of the text
- Bring back your STM32L476 NUCLEO kit!
- With the right cable: USB-micro
- You'll need 2 or 3 cables to connect pins on your board
- Install STM32Cube IDE (Follow the Guide)
- Install Tera Term (if possible)
- Use the plotter developed by Antoine Tauvel