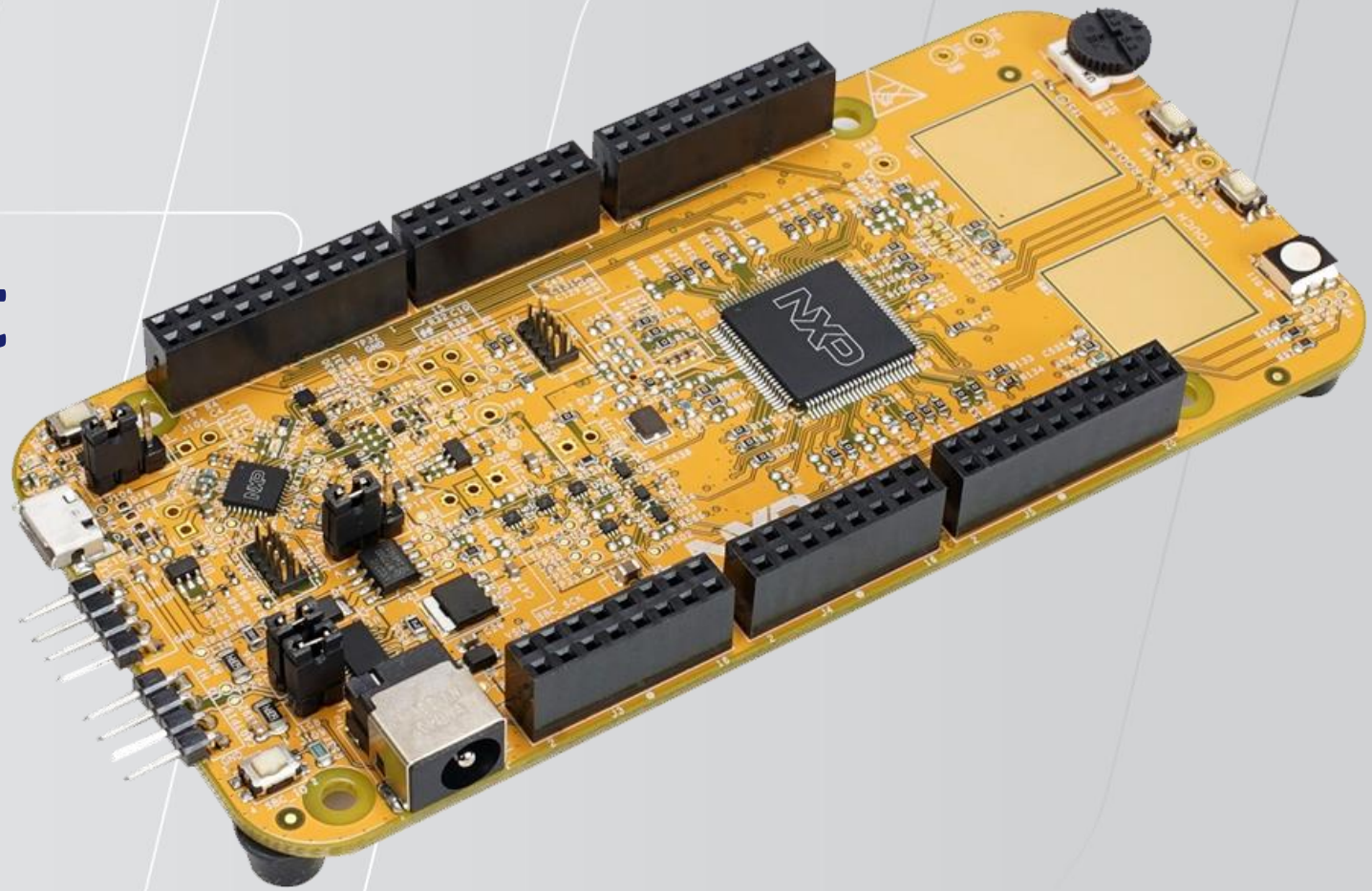
Exception/Interrupt



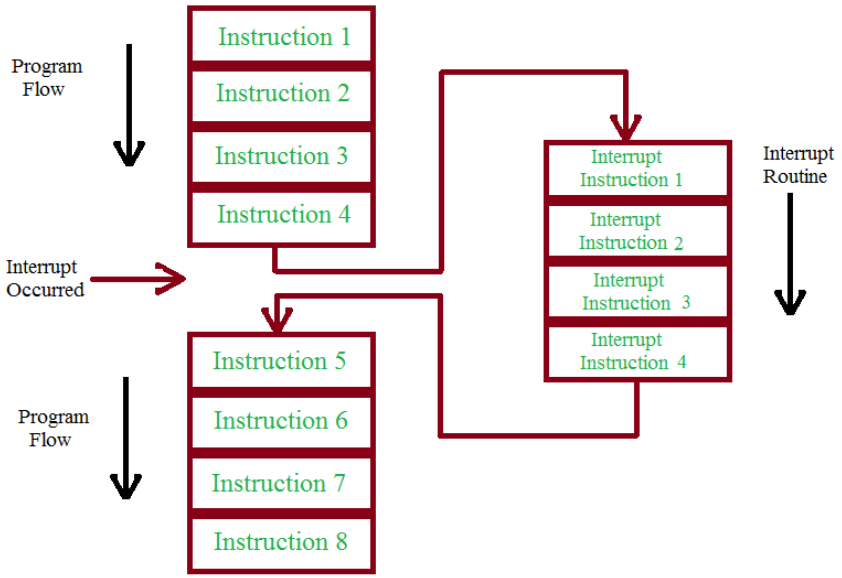
29 Aug 2024

OUTLINE

1. Interrupt Overview
2. Exception model
    - 2.1. Exception states
    - 2.2. Exception types
    - 2.3. Exception handlers
    - 2.4. Vector table
    - 2.5. Exception priorities
    - 2.6. Interrupt priority grouping
    - 2.7. Exception entry and return
3. Nested Vectored Interrupt Controller(NVIC)
4. System Control Block (SCB)
5. Exception mask registers

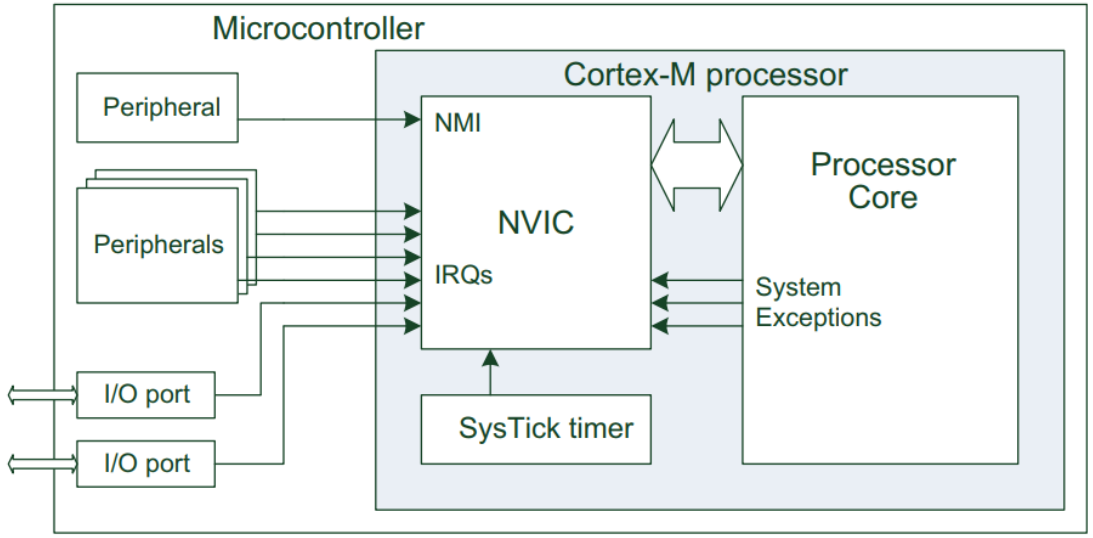
1. Interrupt Overview

- **Interrupts** are a common **feature** available in almost all **microcontrollers**.
- Interrupts are **events** typically **generated** by **hardware** (e.g., peripherals or external input pins) that cause changes in program flow control outside a normal programmed sequence (e.g., to provide service to a peripheral).
- 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 **currently** executing **task**.
  3. The processor **executes** an Interrupt Service Routine (**ISR**) to service the peripheral, and optionally clear the interrupt request by software if needed.
  4. The processor **resumes** the **previously** suspended **task**.



1. Interrupt Overview

- All Cortex-M processors provide a **Nested Vectored Interrupt Controller (NVIC)** for interrupt handling. In addition to interrupt requests, there are other **events** that need servicing and we called them "**exceptions**."
- In ARM terminology, an **interrupt** is **one type of exception**.
- Other exceptions in Cortex-M processors included **fault exceptions** and other **system exceptions** to support the OS (e.g., SVC instruction).
- The **pieces of program code** that **handle exceptions** are often called **exception handlers**.
  In a typical Cortex-M microcontroller, the NVIC receives interrupt requests from various sources, as shown the below figure:



2.1. Exception states

No.	States	Descriptions
1	Inactive	The exception is **not active and not pending**.
2	**Pending**	The exception is **waiting** to be serviced by the processor. An interrupt request from a peripheral or from software can change the state of the corresponding interrupt to pending.
3	**Active**	An exception that is **being serviced** by the processor but has not completed.
4	**Active and pending**	The exception is being serviced by the processor and there is a pending exception from the **same source**.

2.2. Exception types

- Exceptions are numbered **1-15** for **system exceptions** and **16 and above for interrupt** inputs (inputs to the processor)
- Different Cortex-M3 or Cortex-M4 microcontrollers can have different numbers of interrupt sources (from 1-240) and different numbers of priority levels.
- This is because chip designers can configure the Cortex-M3 or Cortex-M4 design source code for different application requirements.
- The value of the currently running exception is indicated by the special register Interrupt Program Status Register (IPSR) or by one of the registers in the NVIC called the Interrupt Control State Register (the VECTACTIVE field).

Table 2-5 IPSR bit assignments

Bits	Name	Function
[31:9]	-	Reserved
[8:0]	ISR_NUMBER	This is the number of the current exception: 0 = Thread mode 1 = Reserved 2 = NMI 3 = HardFault 4 = MemManage 5 = BusFault 6 = UsageFault 7-10 = Reserved 11 = SVCall 12 = Reserved for Debug 13 = Reserved 14 = PendSV 15 = SysTick 16 = IRQ0. . . . n+15 = IRQ(n-1)[a]

2.2. Exception types

No.	Exception types	Descriptions
1	**Reset**	Reset is invoked on power up or a warm reset. The exception model treats reset as a special form of exception. When reset is asserted, the operation of the processor stops, potentially at any point in an instruction. When reset is deasserted, execution restarts from the address provided by the reset entry in the vector table. Execution restarts as privileged execution in Thread mode.
2	**NMI**	A Non Maskable Interrupt (NMI) can be signaled by a peripheral or triggered by software. This is the highest priority exception other than reset. It is permanently enabled and has a fixed priority of -2. NMIs cannot be: • masked or prevented from activation by any other exception • preempted by any exception other than Reset.
3	**HardFault**	A HardFault is an exception that occurs because of an error during exception processing, or because an exception cannot be managed by any other exception mechanism. HardFaults have a fixed priority of -1, meaning they have higher priority than any exception with configurable priority.

2.2. Exception types

No.	Exception types	Descriptions
4	**MemManage**	A MemManage fault is an exception that occurs because of a memory protection related fault. The fixed memory protection constraints determines this fault, for both instruction and data memory transactions. This fault is always used to abort instruction accesses to *Execute Never* (XN) memory regions.
5	**BusFault**	A BusFault is an exception that occurs because of a memory related fault for an instruction or data memory transaction. This might be from an error detected on a bus in the memory system.
6	**UsageFault**	A UsageFault is an exception that occurs because of a fault related to instruction execution. This includes: • an undefined instruction • an illegal unaligned access • invalid state on instruction execution • an error on exception return. The following can cause a UsageFault when the core is configured to report them: • an unaligned address on word and halfword memory access • division by zero.

2.2. Exception types

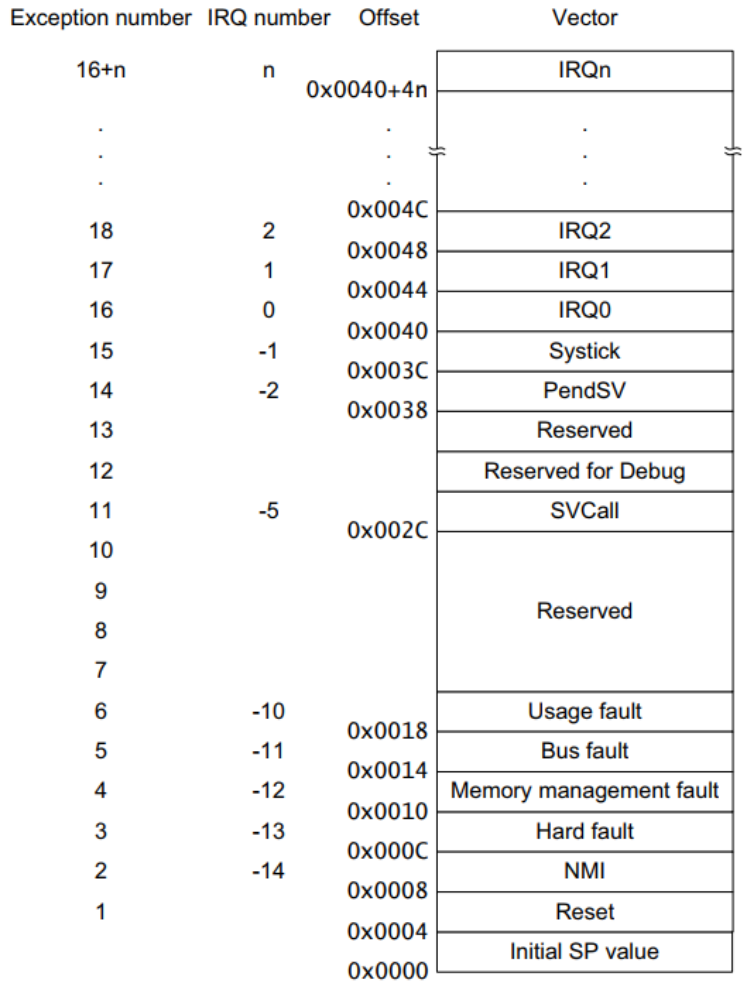
No.	Exception types	Descriptions
7	**SVCall**	A *supervisor call* (SVC) is an exception that is triggered by the SVC instruction. In an OS environment, applications can use SVC instructions to access OS kernel functions and device drivers.
8	**PendSV**	PendSV is an interrupt-driven request for system-level service. In an OS environment, use PendSV for context switching when no other exception is active.
9	**SysTick**	A SysTick exception is an exception the system timer generates when it reaches zero. Software can also generate a SysTick exception. In an OS environment, the processor can use this exception as system tick.
10	**Interrupt (IRQ)**	A interrupt, or IRQ, is an exception signalled by a peripheral, or generated by a software request. All interrupts are asynchronous to instruction execution. In the system, peripherals use interrupts to communicate with the processor.

2.3. Exception handlers

No.	Handler	Descriptions
1	**Interrupt Service Routines (ISRs)**	The IRQ interrupts are the exceptions handled by ISRs.
2	**Fault handlers**	HardFault, MemManage fault, UsageFault, and BusFault are fault exceptions handled by the fault handlers.
3	**System handlers**	NMI, PendSV, SVCall, SysTick, and the fault exceptions are all system exceptions that are handled by system handlers.

2.4. Vector table

- When the Cortex-M processor accepts an exception request, the processor needs to **determine** the **starting address** of the **exception handler** (or ISR if the exception is an interrupt). This information is **stored** in **the vector table** in the memory.
- The vector table contains the **reset value of the stack pointer**, and the **start addresses**, also called exception vectors, for all **exception handlers**
- **By default**, the vector table starts at memory **address 0**, and the vector address is arranged according to the exception number times four. The vector table is normally defined in the **startup codes** provided by the microcontroller vendors. The least-significant bit of each vector must be 1, indicating that the exception handler is Thumb code.
- The **Vector Table Relocation feature** provides a programmable register called the Vector Table Offset Register (**VTOR**)



2.4. Vector table

Exception Num	IRQ Number	Exception type	Priority	Vector address or offset	Activation
0	-	Initial SP value	-	0x00000000	-
1	-	Reset	-3, the highest	0x00000004	Asynchronous
2	-14	NMI	-2	0x00000008	Asynchronous
3	-13	HardFault	-1	0x0000000C	-
4	-12	MemManage	Configurable	0x00000010	Synchronous
5	-11	BusFault	Configurable	0x00000014	Synchronous when precise, asynchronous when imprecise
6	-10	UsageFault	Configurable	0x00000018	Synchronous
7 - 10	-	Reserved	-	-	-
11	-5	SVCall	Configurable	0x0000002C	Synchronous
12 - 13	-	Reserved	-	-	-
14	-2	PendSV	Configurable	0x00000038	Asynchronous
15	-1	SysTick	Configurable	0x0000003C	Asynchronous
16	0	Interrupt (IRQ0)	Configurable	0x00000040	Asynchronous
17	1	Interrupt (IRQ…)	Configurable	0x0000000…	Asynchronous
18	2	Interrupt (IRQn)	Configurable	0x0000000…	Asynchronous

2.5. Exception priorities

All exceptions have an associated priority, with:

- A **lower** priority value indicating a **higher** priority.
- Configurable priorities for all exceptions **except** Reset, HardFault, and NMI.
- If software does **not configure** any priorities, then all exceptions with a configurable priority have a priority of **ZERO**.
- **Reset, HardFault, and NMI** exceptions, with **fixed negative priority** values, always have higher priority than any other exception.

2.6. Interrupt priority grouping

To increase priority control in systems with interrupts, the NVIC supports priority grouping. This divides each interrupt priority register entry into two fields:

- An upper field that defines the group priority
- A lower field that defines a subpriority within the group.

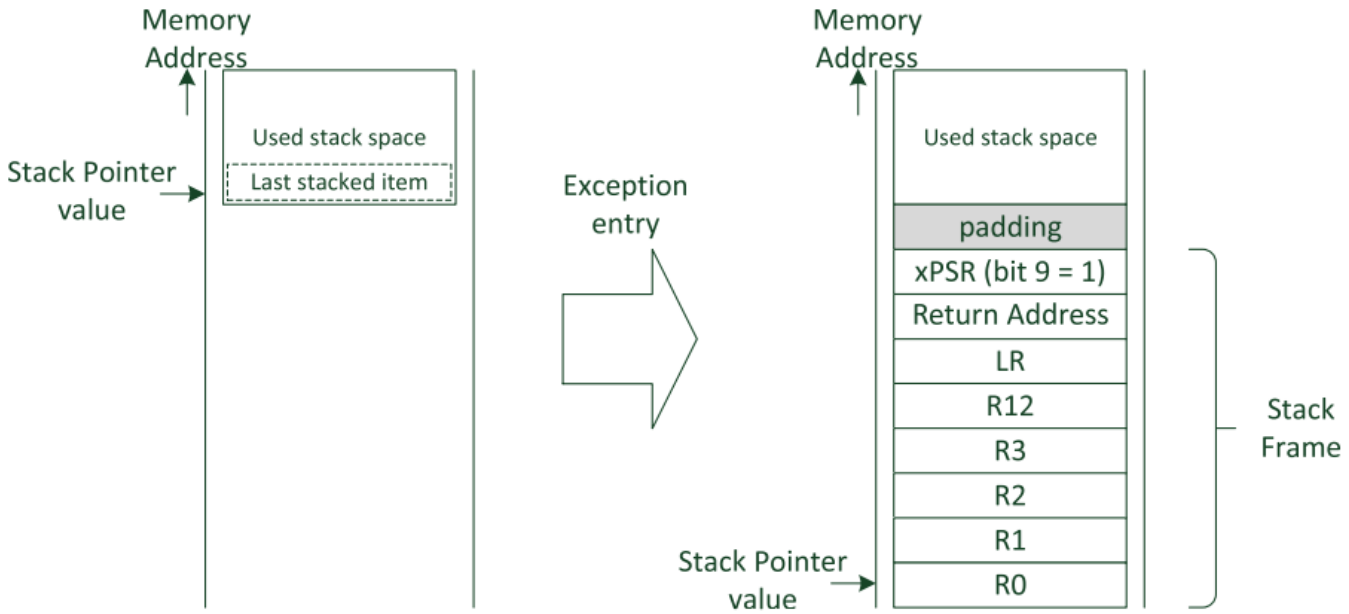
Interrupt priority register

- ✓ When the processor is executing an interrupt exception handler, another interrupt with the **same group priority** as the interrupt being handled does **not preempt** the handler
- ✓ If multiple pending interrupts have the **same group** priority, the **subpriority** field determines the order in which they are processed
- ✓ If multiple pending interrupts have the **same group** priority and **subpriority**, the interrupt with the lowest IRQ number is processed first.

2.7. Exception entry and return

Stacking and Unstacking:

- The exception **entry** sequence, a number of **registers** are **pushed** onto the **stack** automatically. This is called **stacking**.
- When the exception **return** mechanism is triggered, the processor accesses the previously stacked register values in the stack memory during exception entrance and **restores** them back to the **register bank**. This is called **unstacking**.

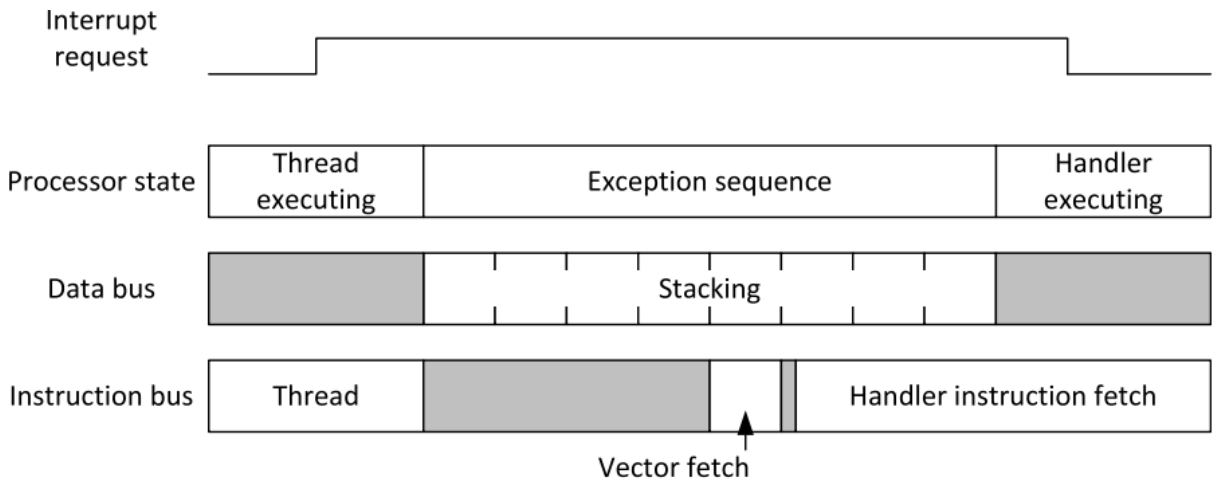


2.7. Exception entry and return

Exception entry :

❖ An exception entrance sequence contains several operations:

1. **Stacking** of a number of registers, including return address to the currently selected stack. This enables an exception handler to be written as a normal C function.
2. **Fetching the exception vector** (starting address of the exception handler/ISR). This can happen in parallel to the stacking operation to reduce latency.
3. Fetching the instructions for the **exception handler** to be **executed**. After the starting address of the exception handler is determined, the instructions can be fetched.
4. **Update** of various **NVIC registers** and **core registers**. This includes the **pending** status and **active** status of the **exception**, and **registers** in the **processor** core including the Program Status Register (PSR), Link Register (LR), Program Counter (PC), and Stack Pointer (SP).



2.7. Exception entry and return

Exception handler execution:

- Within the exception handler, you can carry out services for the peripheral that requires service. The processor is in Handler mode when executing an exception handler. In Handler mode:
  - The **Main Stack Pointer (MSP)** is used for stack operations
  - The processor is executing in **privileged access level**
- If a **higher-priority** exception arrives during this stage, the **new interrupt** will be **accepted**, and the **currently executing handler** will be **suspended** and **pre-empted** by the higher-priority handler. This is called a **nested exception.**
- If another exception with the **same** or **lower** priority arrives during this stage, the **newly** arrived exception will stay in the **pending** state and will be **serviced** when the **current** exception handler is **completed**.

2.7. Exception entry and return

Exception return:

This **occurs** when the exception **handler** is **completed**, and:

- There is **no pending** exception with sufficient priority to be serviced
- The completed exception handler was **not** handling a **late-arriving exception**.

The processor **pops** the **stack** and restores the processor state to the state it had before the interrupt occurred.

- The **exception return** mechanism is triggered using a special **return address** called **EXC_RETURN**. This value is generated at exception entry and is **stored** in the **Link Register** (LR). When **EXC_RETURN** is written to the PC, it triggers the exception return.
- When the exception return mechanism is triggered, the processor accesses the previously stacked register values in the stack memory during exception entrance and **restores** them back to the **register bank**.
- In parallel to the **unstacking** operation, the processor can start **fetching** the **instructions** of the **previously** interrupted program to allow the program to **resume** operation as soon as possible.

2.7. Exception entry and return

What is **EXC_RETURN**?

EXC_RETURN is the value loaded into the LR on exception entry

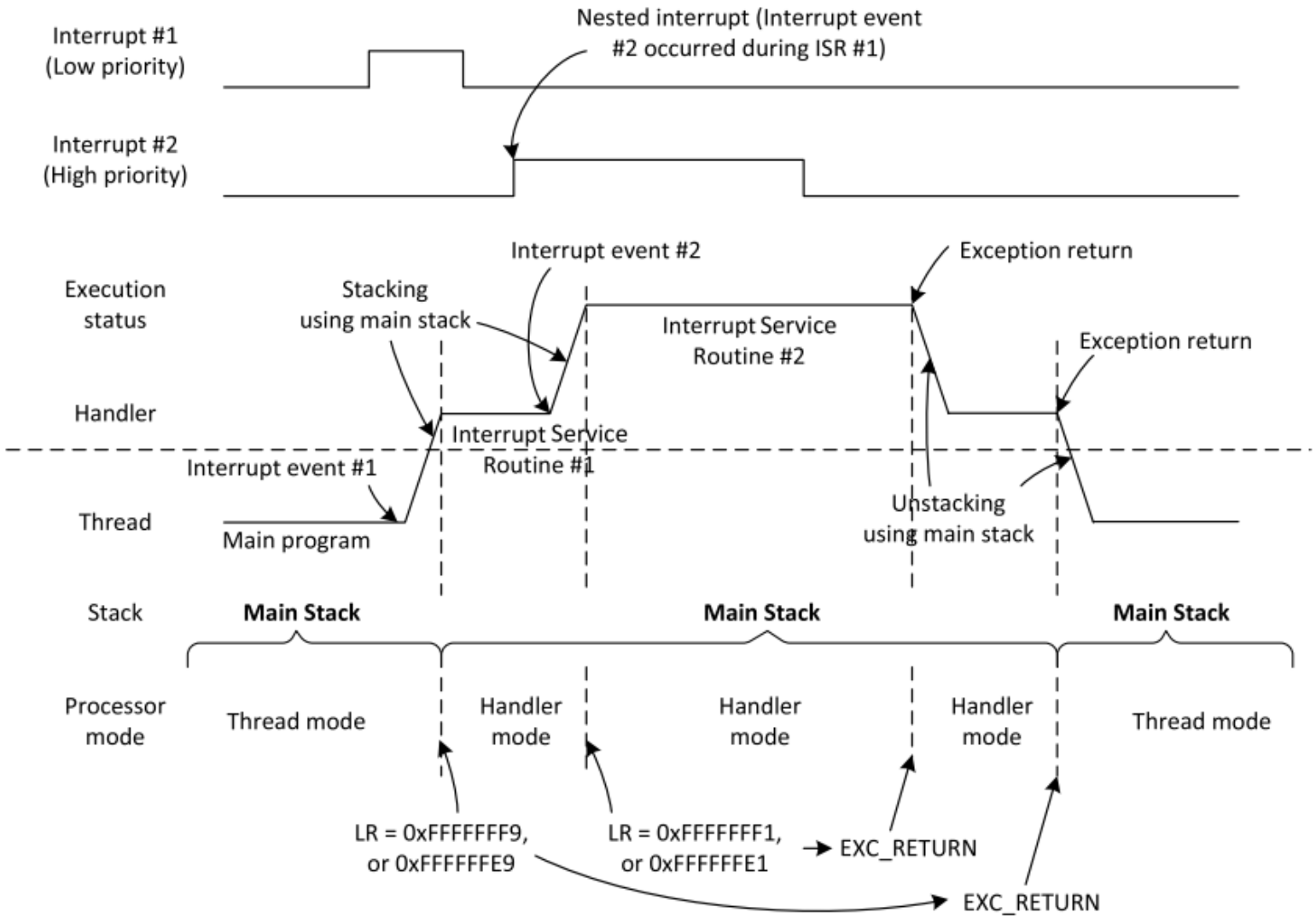
This indicates:

- ❑ Stack pointer corresponds to the stack frame
- ❑ Operation mode the processor was in before the entry occurred.

EXC_RETURN[31:0]	Description
0xFFFFFF**F**1	Return to Handler mode, exception return uses **non-floating-point** state from the MSP and execution uses MSP after return.
0xFFFFFF**F**9	Return to Thread mode, exception return uses **non-floating-point** state from MSP and execution uses MSP after return.
0xFFFFFF**F**D	Return to Thread mode, exception return uses **non-floating-point** state from the PSP and execution uses PSP after return.
0xFFFFFFE1	Return to Handler mode, exception return uses floating-point state from MSP and execution uses MSP after return.
0xFFFFFFE9	Return to Thread mode, exception return uses floating-point state from MSP and execution uses MSP after return.
0xFFFFFFED	Return to Thread mode, exception return uses floating-point state from PSP and execution uses PSP after return.

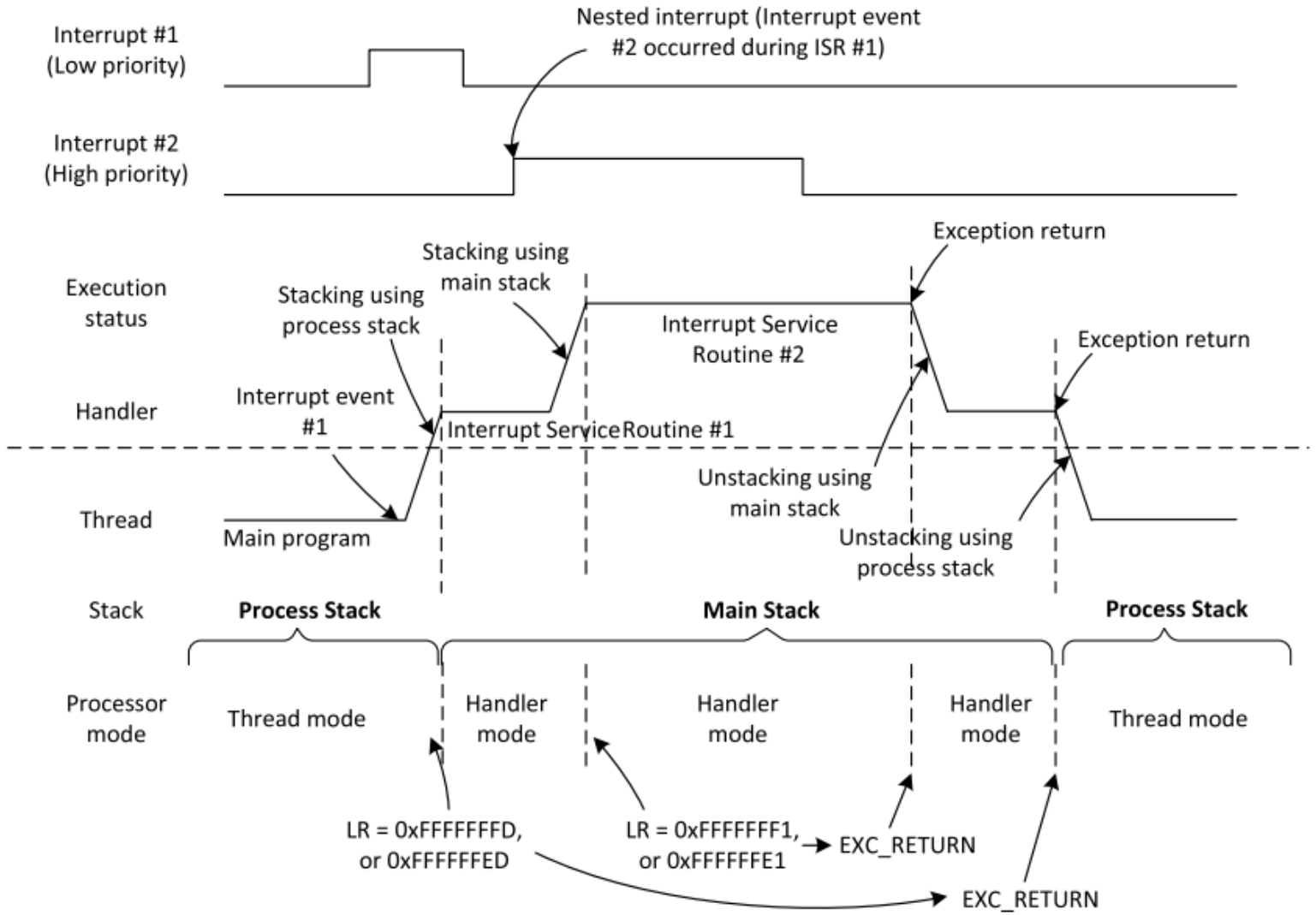
2.7. Exception entry and return

Example:



2.7. Exception entry and return

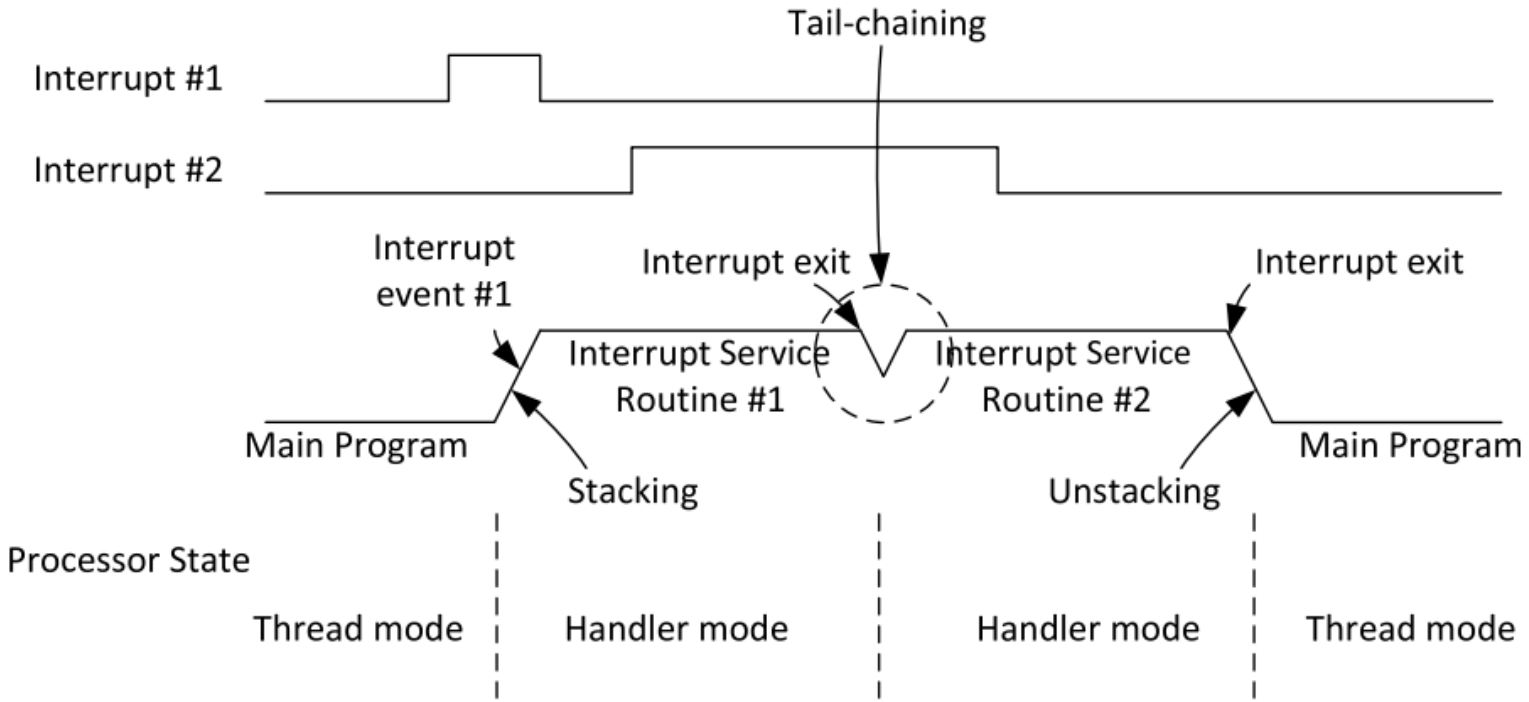
Example:



2.7. Exception entry and return

Tail-chaining:

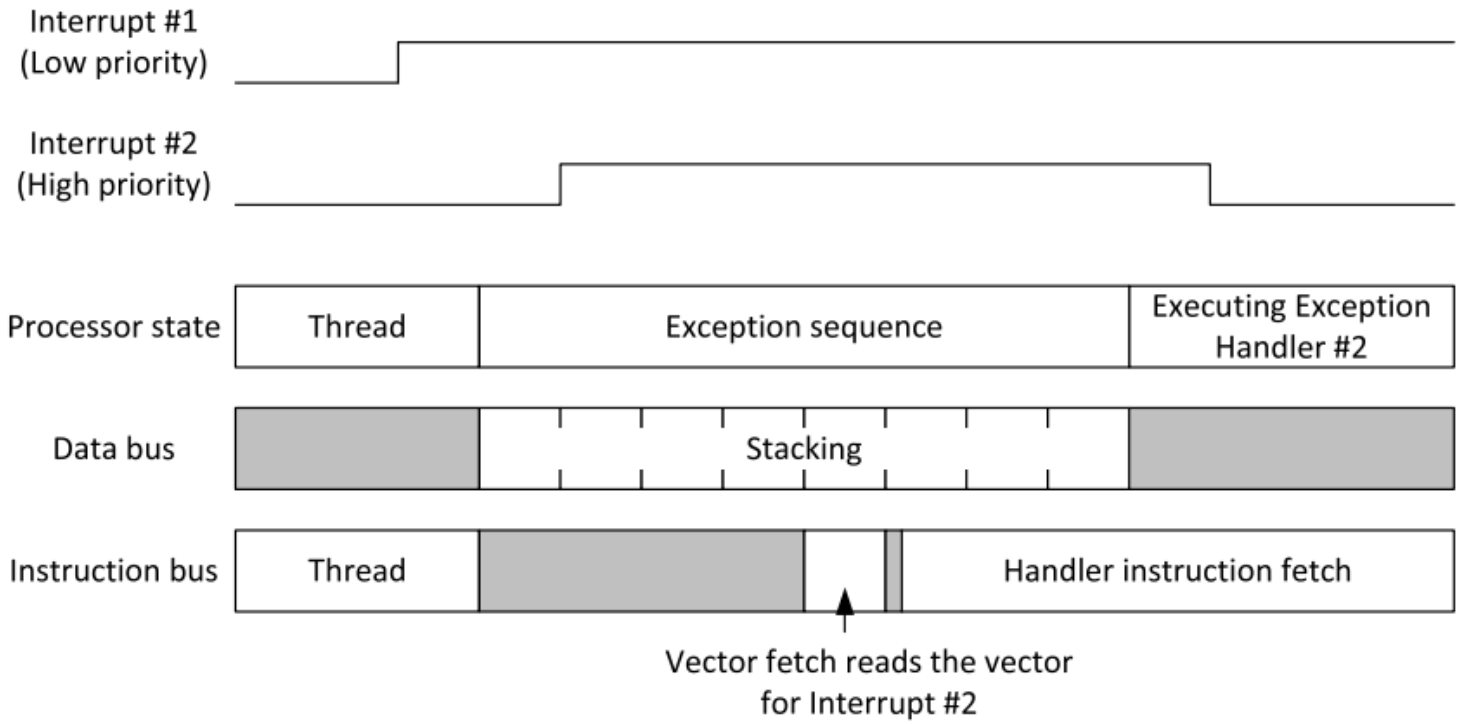
This mechanism **speeds up** exception servicing. On **completion** of an **exception** handler, if there is a **pending** exception that meets the requirements for exception **entry**, the **stack pop** is **skipped** and control transfers to the **new exception** handler.



2.7. Exception entry and return

Late-arriving:

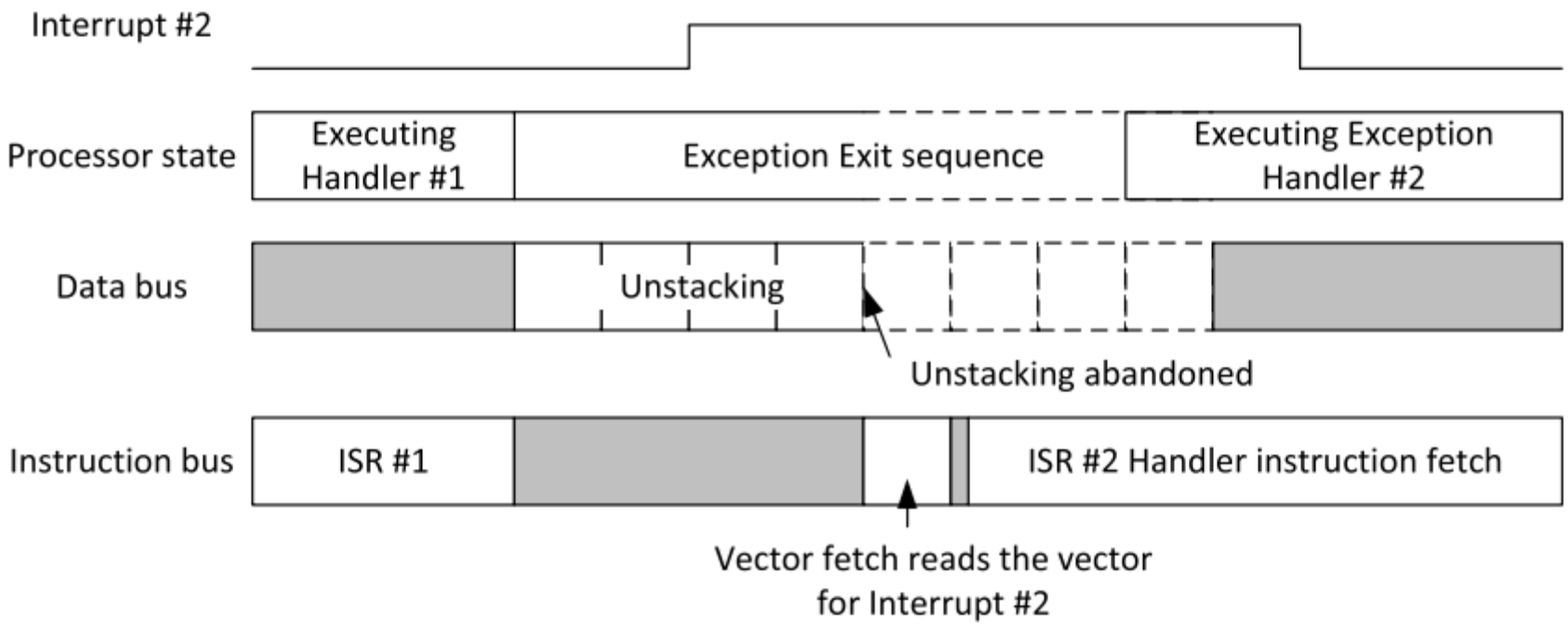
This mechanism **speeds up** preemption. If a **higher** priority exception **occurs** during state **saving** for a previous exception, the processor **switches** to **handle** the **higher** priority exception and initiates the vector fetch for that exception. State saving is not affected by late arrival because the state saved is the same for both exceptions. Therefore the state saving continues uninterrupted. The processor can accept a late arriving exception until the first instruction of the exception handler of the original exception enters the execute stage of the processor. On return from the exception handler of the late-arriving exception, the normal tail-chaining rules apply.



2.7. Exception entry and return

Pop preemption:

If an **exception** request **arrives** during the **unstacking** process of another exception handler that has just finished, the **unstacking operation** would be **abandoned** and the **vector fetch** and **instruction fetch** for the next **exception** service **begins**. This optimization is called pop pre-emption.



3. Nested Vectored Interrupt Controller(NVIC)

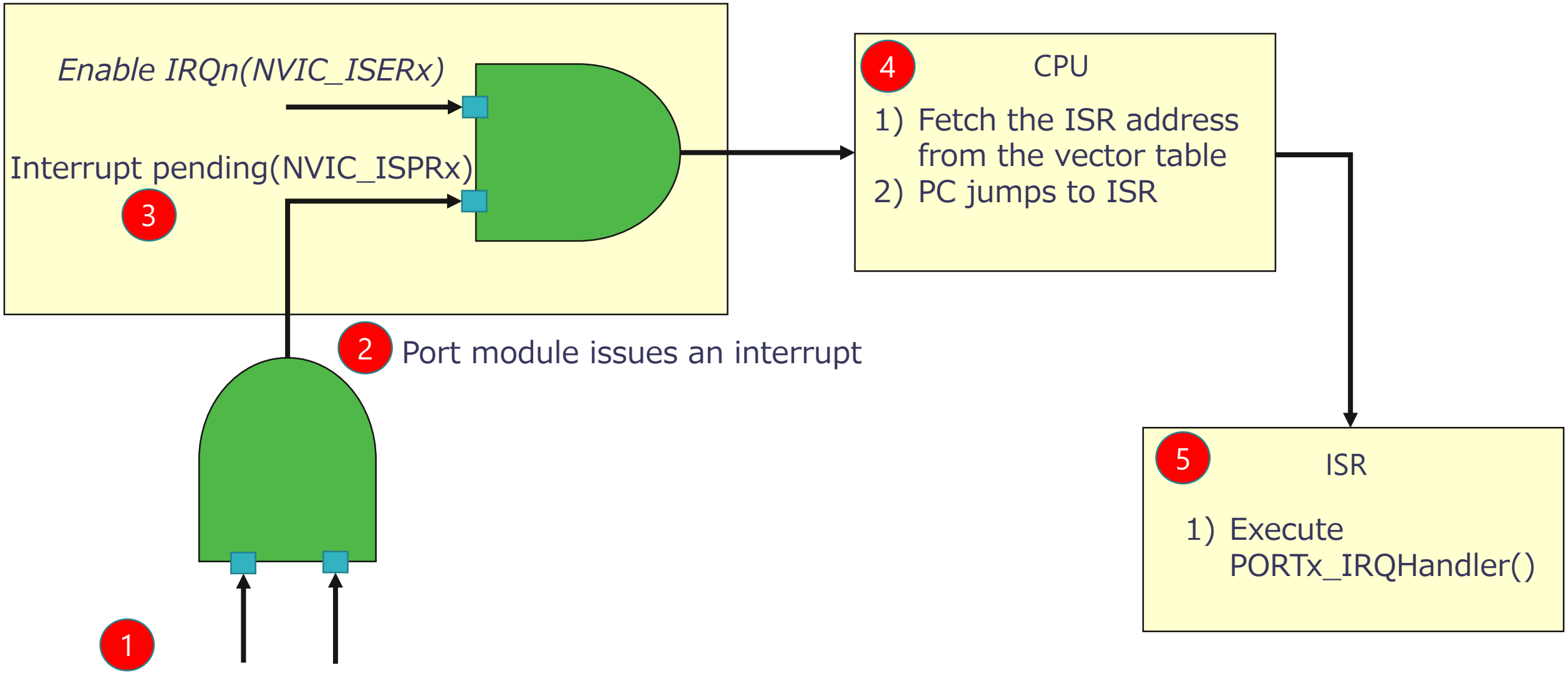
No.	Supported Feature
1	An implementation-defined number of interrupts, in the range **1-240** interrupts
2	A **programmable priority level** of 0-255
3	Level and pulse detection of interrupt signals.
4	Dynamic reprioritization of interrupts.
5	**Grouping of priority**
6	Interrupt tail-chaining.
7	An external Non Maskable Interrupt (NMI)
8	Optional WIC, providing ultra-low power sleep mode support.

3. Nested Vectored Interrupt Controller(NVIC)

FPT Software

NVIC Register Summary					
Address	**Name**	**Type**	**Required privilege**	**Reset value**	**Description**
0xE000E100 - 0xE000E11C	NVIC_ISER0 - NVIC_ISER7	RW	Privileged	0x00000000	Interrupt Set-enable Registers
0XE000E180 - 0xE000E19C	NVIC_ICER0 - NVIC_ICER7	RW	Privileged	0x00000000	Interrupt Clear-enable Registers
0XE000E200 - 0xE000E21C	NVIC_ISPR0 - NVIC_ISPR7	RW	Privileged	0x00000000	Interrupt Set-pending Registers
0XE000E280 - 0xE000E29C	NVIC_ICPR0 - NVIC_ICPR7	RW	Privileged	0x00000000	Interrupt Clear-pending Registers
0xE000E300 - 0xE000E31C	NVIC_IABR0 - NVIC_IABR7	RW	Privileged	0x00000000	Interrupt Active Bit Registers
0xE000E400 - 0xE000E4EF	NVIC_IPR0 - NVIC_IPR59	RW	Privileged	0x00000000	Interrupt Priority Registers
0xE000EF00	STIR	WO	Configurable	0x00000000	Software Trigger Interrupt Register

3. Nested Vectored Interrupt Controller(NVIC)



3. Nested Vectored Interrupt Controller(NVIC)

Address	Name	Type	Required privilege	Reset value	Description
0xE000E100 - 0xE000E11C	NVIC_ISER0 - NVIC_ISER7	RW	Privileged	0x00000000	Interrupt Set-enable Registers

NVIC_ISER0 - NVIC_ISER7:

- 8 registers 32 bits
- 8 * 32 = 256 bits support for 256 interrupts
- Each bit controls 1 IRQn(Interrupt Request)

The NVIC_ISER0-NVIC_ISER7 registers enable interrupts, and show which interrupts are enabled. See the register summary in Table 4-2 on page 4-3 for the register attributes.

The bit assignments are:

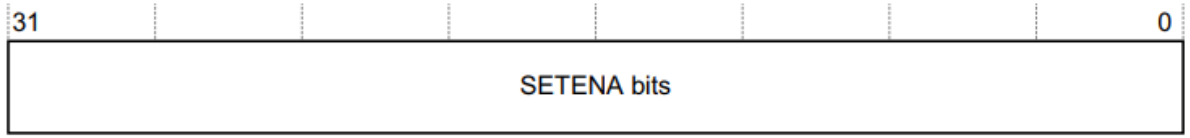


Table 4-4 ISER bit assignments

Bits	Name	Function
[31:0]	SETENA	Interrupt set-enable bits. Write: 0 = no effect 1 = enable interrupt. Read: 0 = interrupt disabled 1 = interrupt enabled.

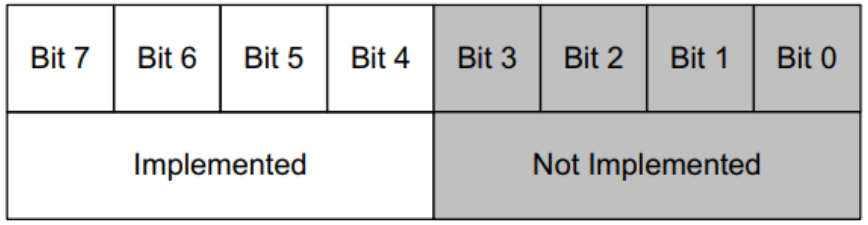
If a pending interrupt is enabled, the NVIC activates the interrupt based on its priority. If an interrupt is not enabled, asserting its interrupt signal changes the interrupt state to pending, but the NVIC never activates the interrupt, regardless of its priority.

3.1. Priority and Group priority

Address	Name	Type	Required privilege	Reset value	Description
0xE000E400 - 0xE000E4EF	NVIC_IPR0 - NVIC_IPR59	RW	Privileged	0x00000000	Interrupt Priority Registers

NVIC_IPR0 - NVIC_IPR59:

- 60 registers 32 bits
- Each register sets priority for 4 IRQn
- Register priority value fields are 8-bits wide for each interrupt
- The actual number of available programmable priority levels is decided by silicon chip designers
- **Actual**, a **priority-level** register with **4 bits** implemented (16 programmable priority levels)



The NVIC_IPR0-NVIC_IPR59 registers provide an 8-bit priority field for each interrupt and each register holds four priority fields. These registers are byte-accessible. See the register summary in Table 4-2 on page 4-3 for their attributes. Each register holds four priority fields as shown:

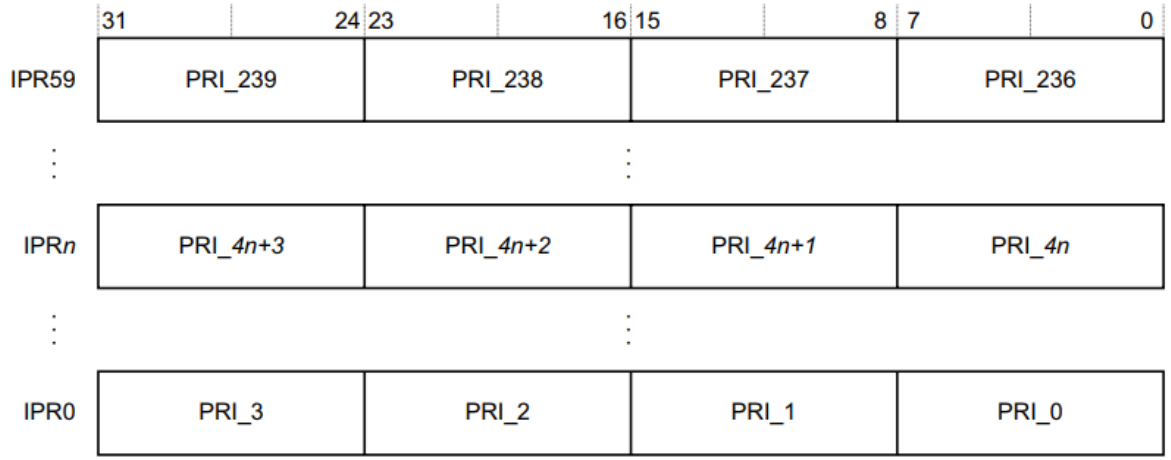


Table 4-9 IPR bit assignments

Bits	Name	Function
[31:24]	Priority, byte offset 3	Each implementation-defined priority field can hold a priority value, 0-255. The lower the value, the greater the priority of the corresponding interrupt. Register priority value fields are eight bits wide, and non-implemented low-order bits read as zero and ignore writes.
[23:16]	Priority, byte offset 2	
[15:8]	Priority, byte offset 1	
[7:0]	Priority, byte offset 0	

4. System Control Block (SCB)

❖ The *System Control Block* (SCB) provides system implementation information, and **system control**. This includes **configuration**, **control**, and **reporting** of the **system exceptions**.

Address	Name	Type	Access	Description
0xE000E008	ACTLR	RW	Privileged	Auxiliary Control Register
0xE000ED00	CPUID	RO	Privileged	CPUID Base Register
0xE000ED04	ICSR	RW	Privileged	Interrupt Control and State Register
0xE000ED08	VTOR	RW	Privileged	Vector Table Offset Register
0xE000ED0C	AIRCR	RW	Privileged	Application Interrupt and Reset Control Register
0xE000ED10	SCR	RW	Privileged	System Control Register
0xE000ED14	CCR	RW	Privileged	Configuration and Control Register
0xE000ED18	SHPR1	RW	Privileged	System Handler Priority Register
0xE000ED1C	SHPR2	RW	Privileged	System Handler Priority Register
0xE000ED20	SHPR3	RW	Privileged	System Handler Priority Register
0xE000ED24	SHCRS	RW	Privileged	System Handler Control and State Register
0xE000ED28	CFSR	RW	Privileged	Configurable Fault Status Register
0xE000ED28	MMSR	RW	Privileged	MemManage Fault Status Register
0xE000ED29	BFSR	RW	Privileged	BusFault Status Register
0xE000ED2A	UFSR	RW	Privileged	UsageFault Status Register
0xE000ED2C	HFSR	RW	Privileged	HardFault Status Register
0xE000ED34	MMAR	RW	Privileged	MemManage Fault Address Register
0xE000ED38	BFAR	RW	Privileged	BusFault Address Register
0xE000ED3C	AFSR	RW	Privileged	Auxiliary Fault Status Register

- The VTOR indicates the offset of the **vector table base address** from memory address 0x00000000.
- The AIRCR provides **priority grouping** control for the exception model, endian status for data accesses, and **reset** control of the **system**.

5. Exception mask registers

- The exception mask registers **disable** the **handling** of **exceptions** by the processor.
- Disable exceptions where they might **impact** on **timing critical tasks.**

❖ ***Priority Mask Register - PRIMASK:***

- The **PRIMASK** register **prevents activation** of **all exceptions with configurable priority.**
- The PRIMASK register is used to **disable all exceptions** except **NMI** and **HardFault**. It effectively changes the current priority level to 0 (highest programmable level).
- In C programming, you can use the functions provided in CMSIS-Core to set and clear PRIMASK:

```
void __enable_irq();   // Clear PRIMASK
void __disable_irq();   // Set PRIMASK
```

- In assembly language programming, you can change the value of PRIMARK register using CPS (Change Processor State) instructions:

```
CPSIE I ; Clear PRIMASK (Enable interrupts)
CPSID I ; Set PRIMASK (Disable interrupts)
```

5. Exception mask registers

❖ *Fault Mask Register - FAULTMASK:*

- The **FAULTMASK** register **prevents activation** of **all exceptions** except for **Non-Maskable Interrupt (NMI),** so that even the **HardFault** handler is **blocked**. Only the **NMI** exception handler can be **executed** when FAULTMASK is set
- In **C** programming, you can use the following CMSIS-Core functions to **set and clear the FAULTMASK**:
  void __enable_fault_irq(void); // Clear FAULTMASK
  void __disable_fault_irq(void); // Set FAULTMASK to disable
- In **assembly** language programming, you can change the current status of the FAULTMASK using CPS (Change Processor State) instructions:
  CPSIE F ; Clear FAULTMASK
  CPSID F ; Set FAULTMASK
- FAULTMASK is cleared automatically upon exiting the exception handler except return from NMI handler. This characteristic provides an interesting usage for FAULTMASK: if in a lower-priority exception handler we want to trigger a higher-priority handler (except NMI), but want this higher-priority handler to start AFTER the lower-priority handler is completed, we can:
  - Set the FAULTMASK to disable all interrupts and exceptions.
  - Set the pending status of the higher-priority interrupt or exception
  - Exit the handler

  Because the pending higher-priority exception handler cannot start while the FAULTMASK is set, the higher-priority exception stays in the pending state until FAULTMASK is cleared, which happens when the lower-priority handler finishes. As a result, you can force the higher-priority handler to start after the lower-priority handler ends.

5. Exception mask registers

❖ ***Base Priority Mask Register - BASEPRI:***

- The **BASEPRI** register defines the **minimum priority for exception processing**. When **BASEPRI** is **set** to a **nonzero** value, it **prevents** the **activation** of all **exceptions** with the **same** or **lower priority level** as the BASEPRI value.
- For example, block all exceptions with priority level equal to or lower than 0x60
- In C programming, you can use the functions provided in CMSIS-Core to you can write the value to BASEPRI:

```
__set_BASEPRI(0x60); // Disable interrupts with priority 0x60-0xFF
```

- In **assembly** language programming, you can write:

```
MOVS R0, #0x60
MSR BASEPRI, R0 ; Disable interrupts with priority 0x60-0xFF
```


Thank you