Embedded Systems Course - Interrupts and Exception handling

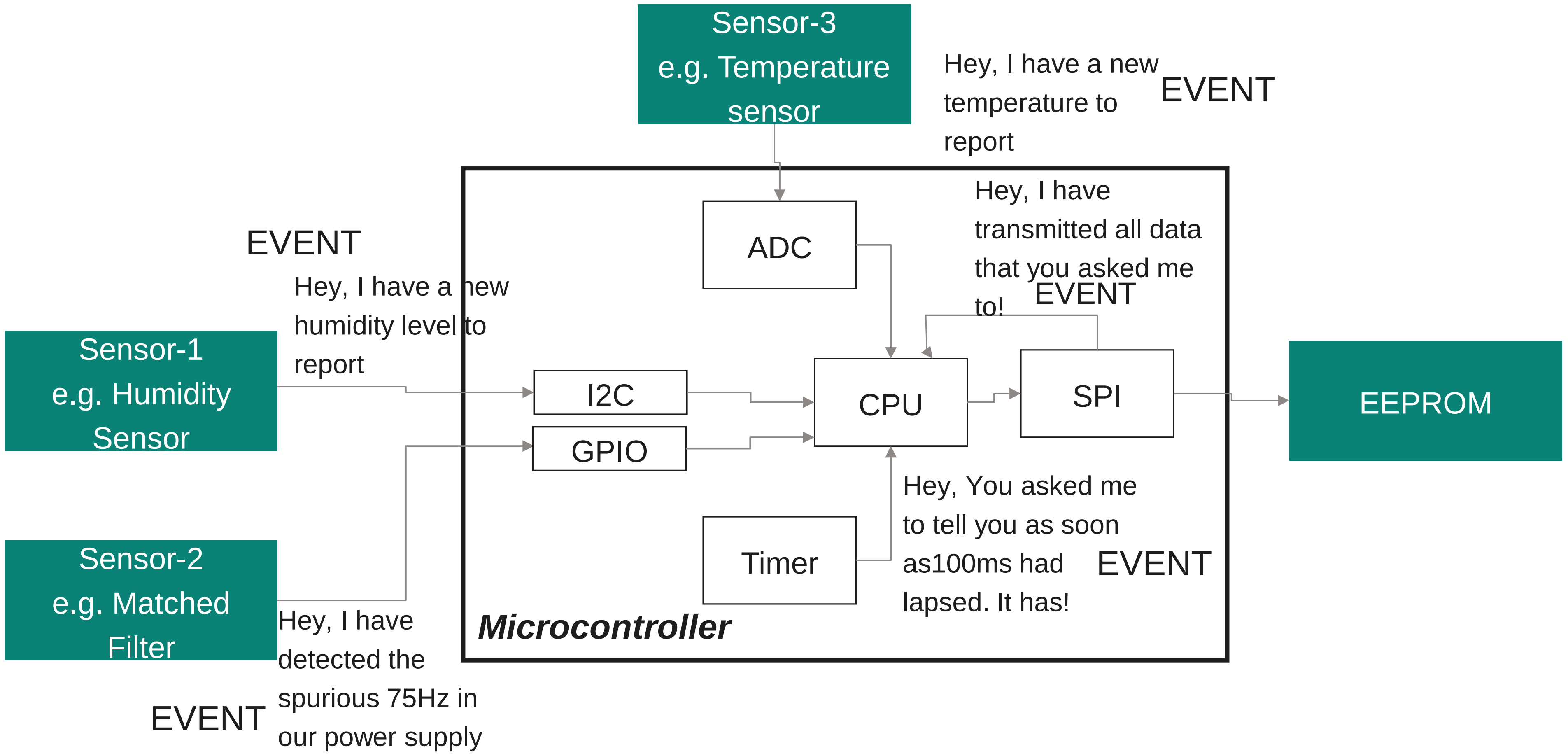
Siddhesh Raut

11 May 2024

Outline

- Event Driven System.
- Polling Method of event.
- Exception, Interrupt and Exception handler.
- Different types of Exceptions
- Operation Modes
- Register Bank
- Interrupt Controller
- Brief overview of NVIC
- NVIC Registers
- Vector Table
- Exception Entry Sequence
- Exception Exit Sequence
- Interrupt Process(complete)

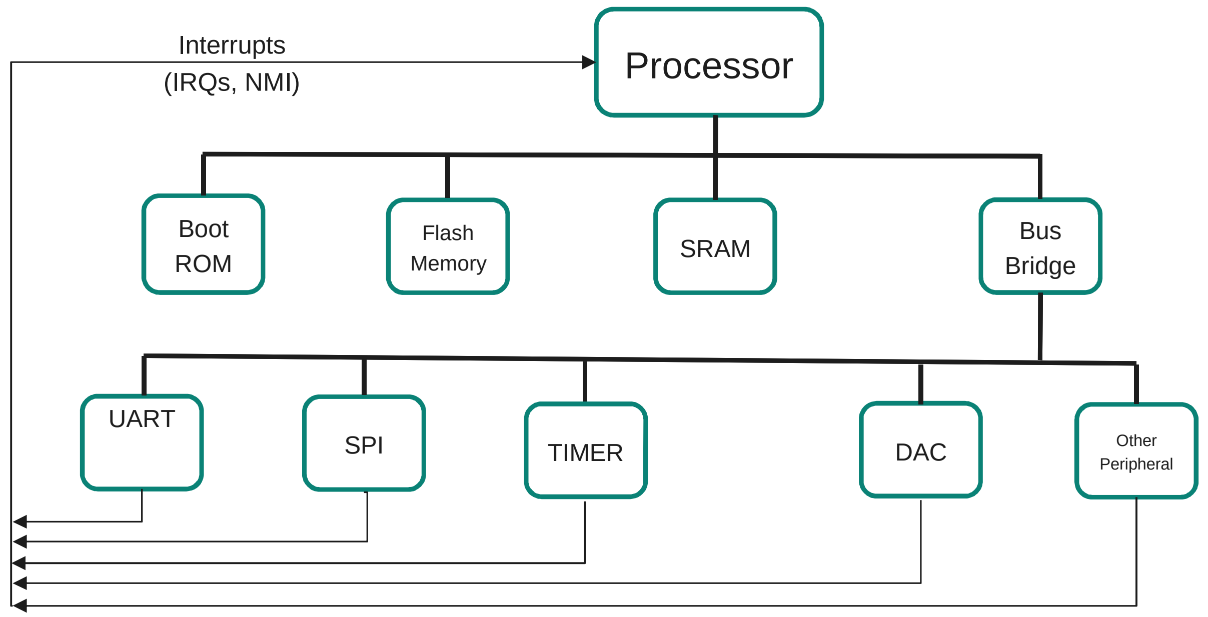
Event driven embedded system – 1/2



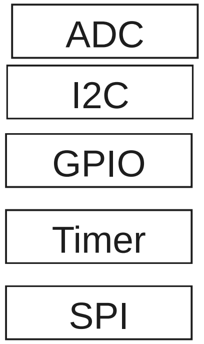
Sensor-3 e.g. Temperature sensor
Hey, I have a new temperature to report
EVENT
ADC
Hey, I have transmitted all data that you asked me to!
EVENT
EVENT
Hey, I have a new humidity level to report
Sensor-1 e.g. Humidity Sensor
I2C
GPIO
CPU
SPI
EEPROM
Hey, You asked me to tell you as soon as100ms had lapsed. It has!
EVENT
Timer
Sensor-2 e.g. Matched Filter
Hey, I have detected the spurious 75Hz in our power supply
EVENT
Microcontroller

Event driven embedded system – 2/2

- Event represents significant occurrence or state changes within a system
  - It represented as signals that convey information about particular occurrence
- Event are asynchronous in nature
- Events can occur at any time!



Methods of handling events – 1/2



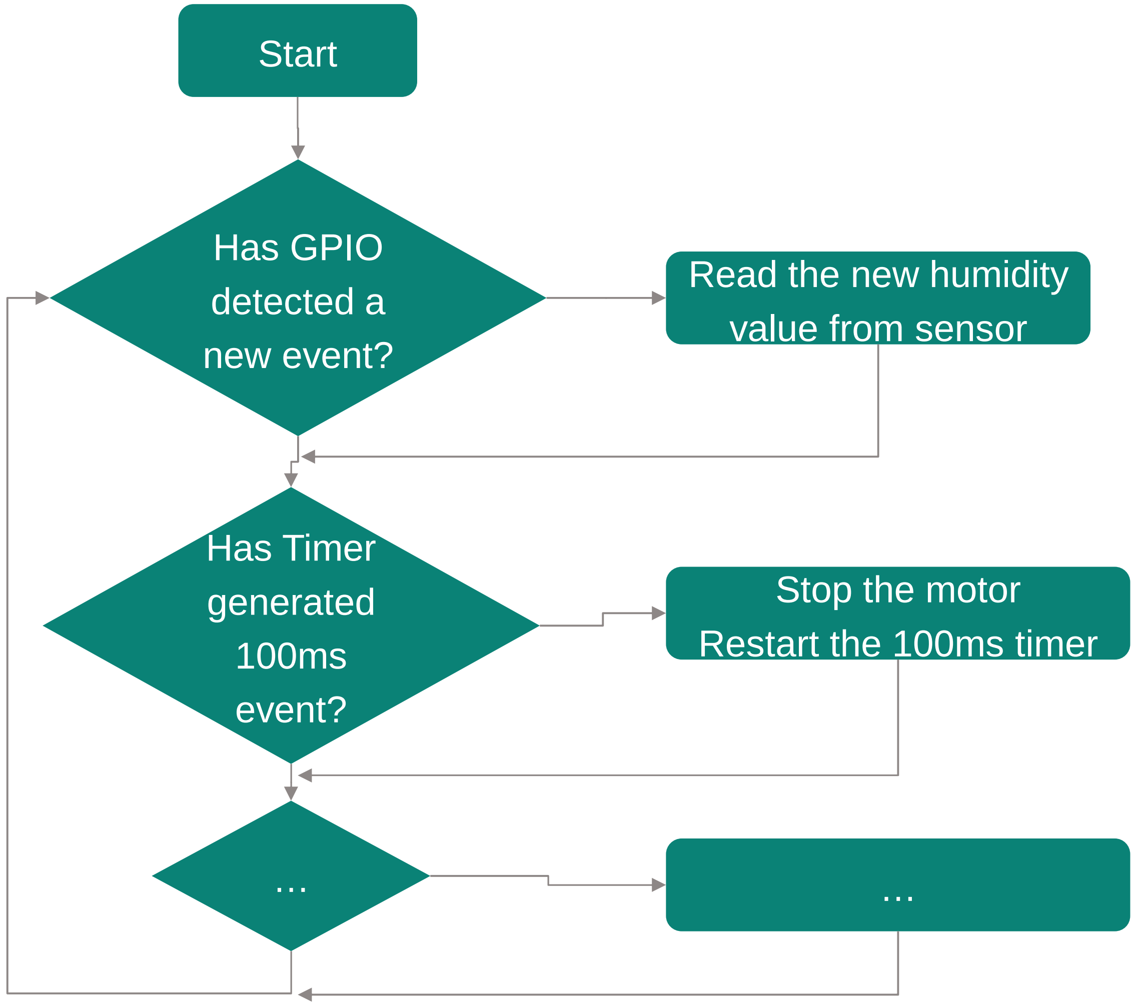
Polling mode

What should be the correct order of polling?

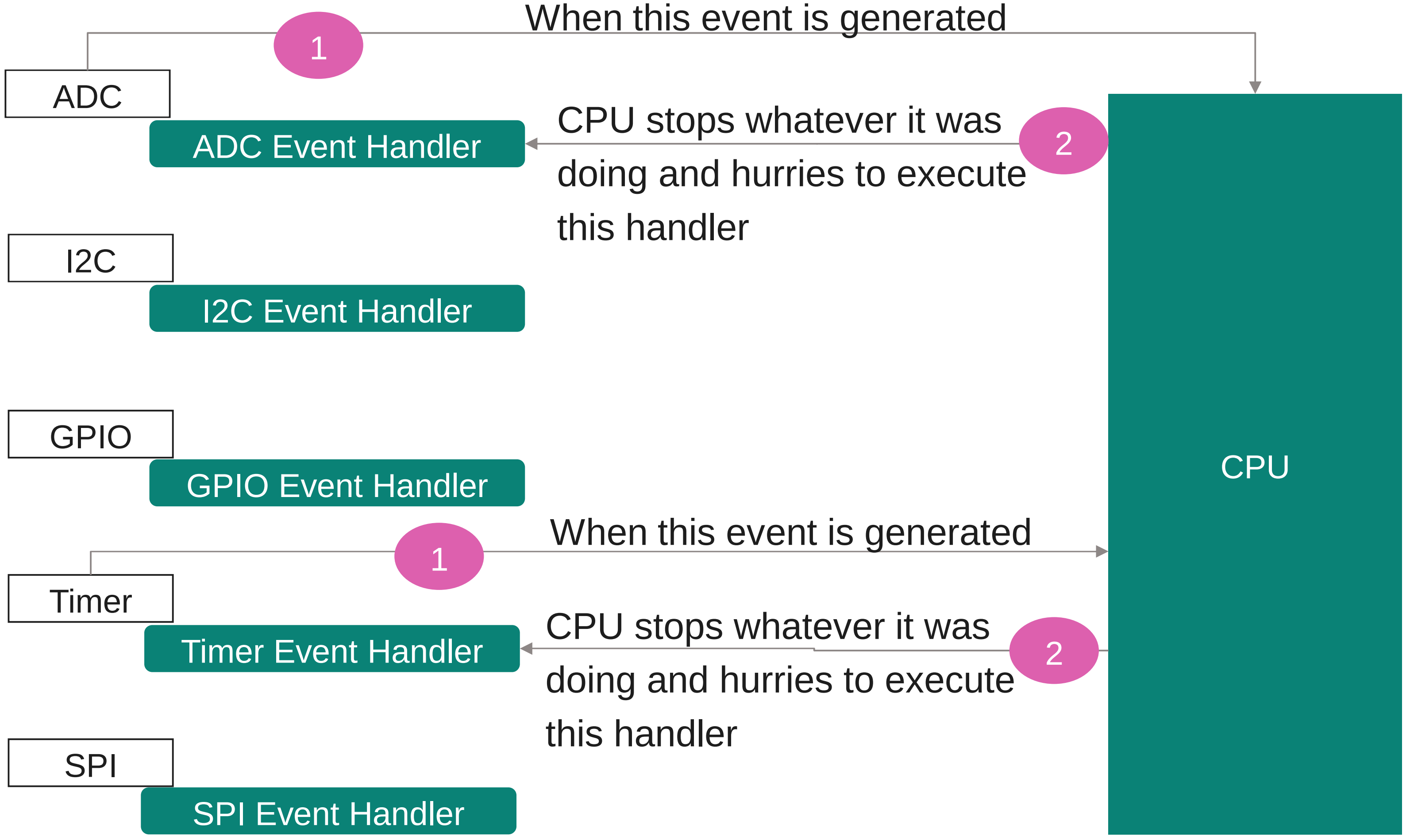
Can the event be lost by the time we get to a peripheral?

Energy inefficient. CPU is always burning power

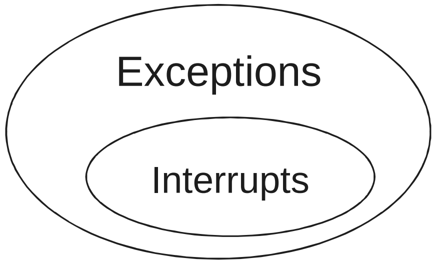
Use polling mode as necessary. It is NOT criminal to do so!


Start
Has GPIO detected a new event?
Read the new humidity value from sensor
Has Timer generated 100ms event?
Stop the motor
Restart the 100ms timer
…
…

Methods of handling events – 2/2


When this event is generated
1
ADC
ADC Event Handler
CPU stops whatever it was doing and hurries to execute this handler
2
I2C
I2C Event Handler
GPIO
GPIO Event Handler
When this event is generated
1
Timer
Timer Event Handler
CPU stops whatever it was doing and hurries to execute this handler
2
SPI
SPI Event Handler
CPU

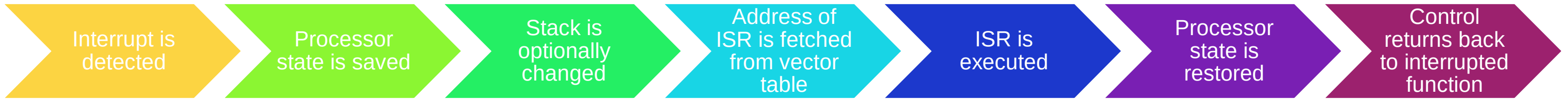
Events are exceptions!



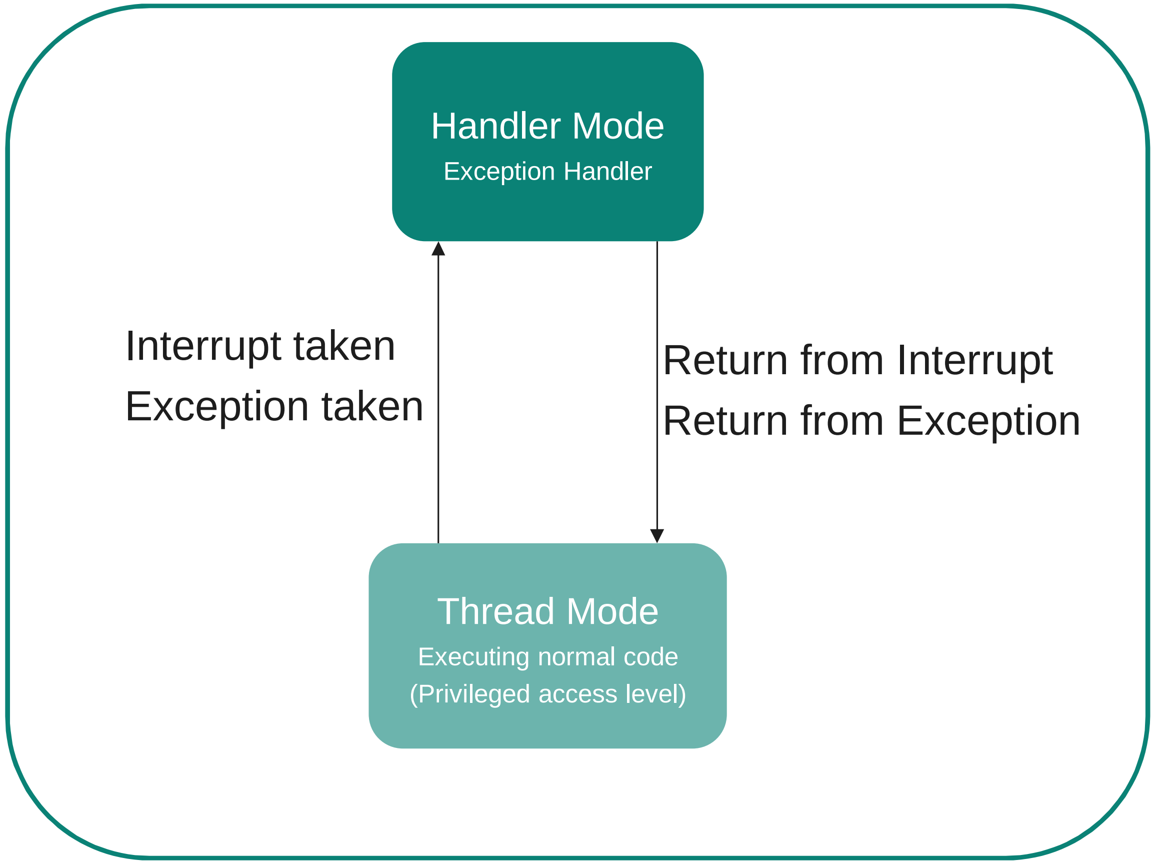
Exception, Interrupt and Exception handler

- Any condition that needs to halt the normal sequential execution of instructions.
- In Exception the device notices the CPU that it requires its attention.
- An exception is not a protocol, its a hardware mechanism.
- Interrupts are a subset of exceptions. External to processor core.
- when an exception occurred, instead of continuing program execution, the processor suspends the current executing task and executes a part of the program code called the *exception handler*.
- If the exception handler is associated with an interrupt event, then it can also be called as interrupt handler, or Interrupt Service Routine (ISR).

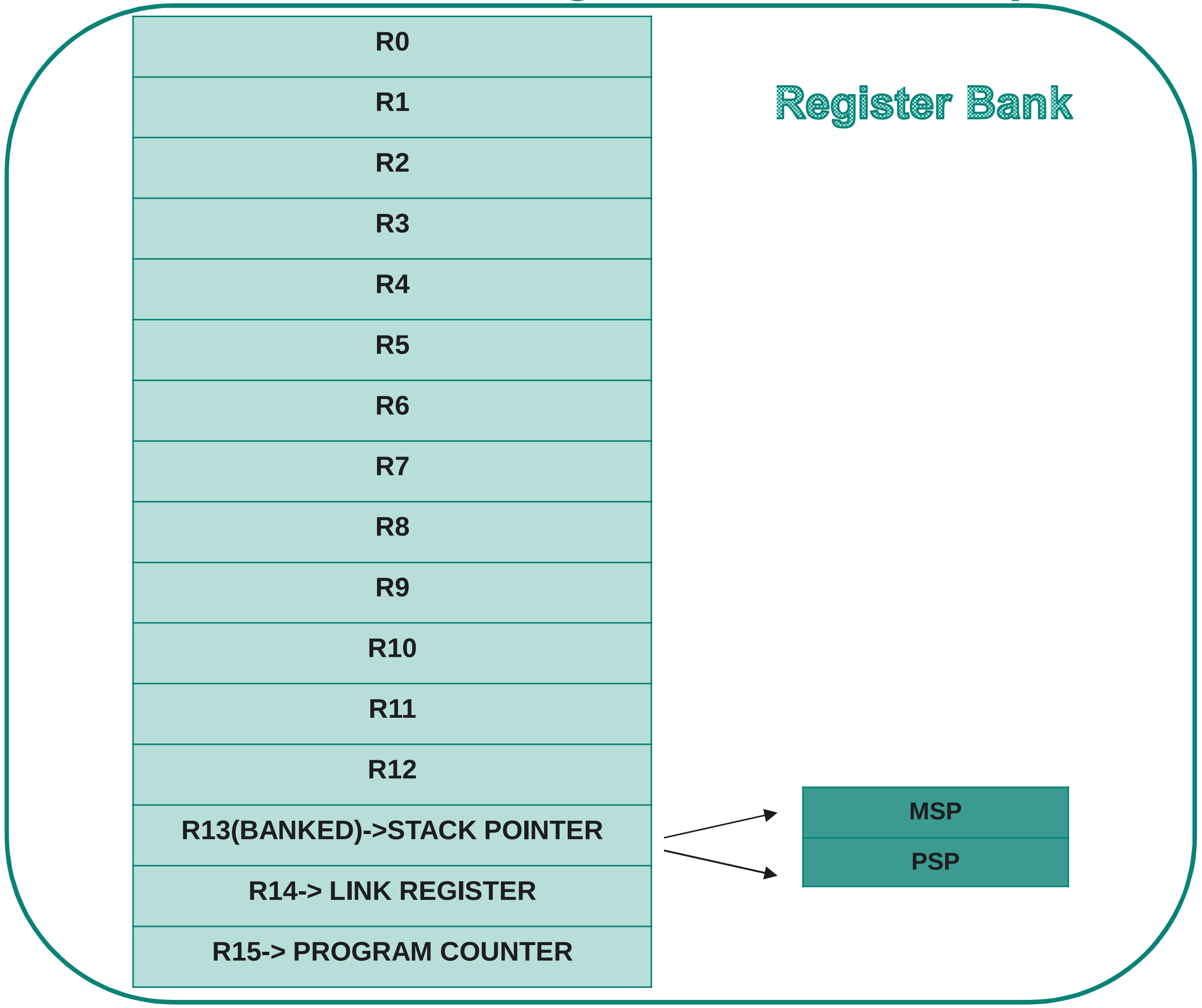
Overview of interrupt handling

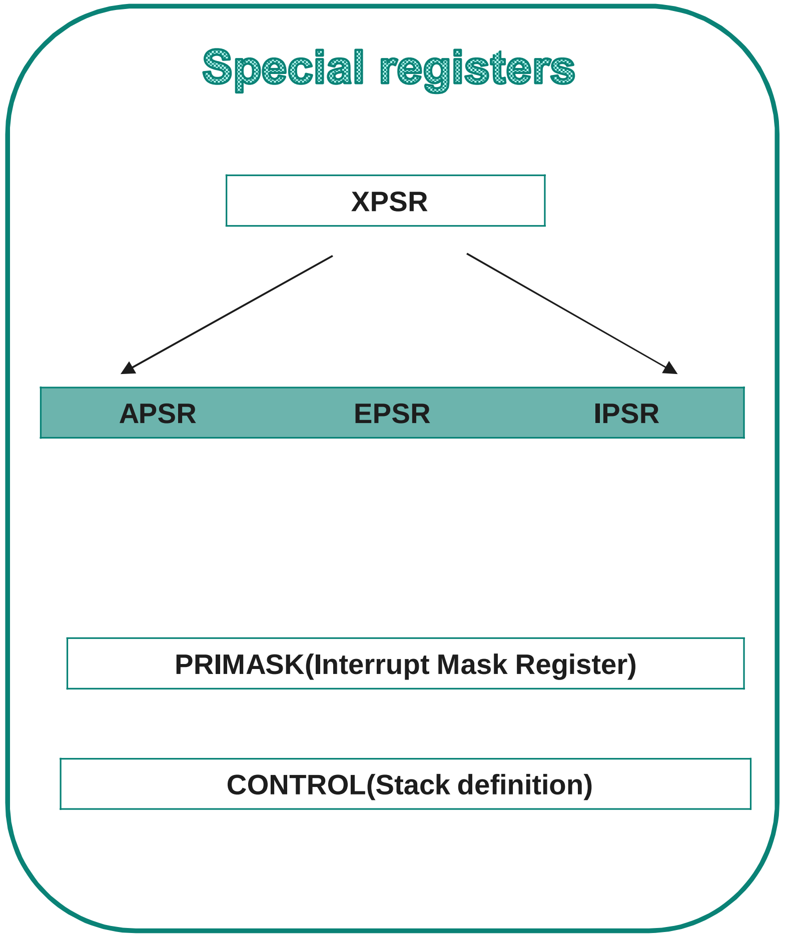

Interrupt is detected
Processor state is saved
Stack is optionally changed
Address of ISR is fetched from vector table
ISR is executed
Processor state is restored
Control returns back to interrupted function

Operation Modes and States



Processor state - Register Bank and special registers



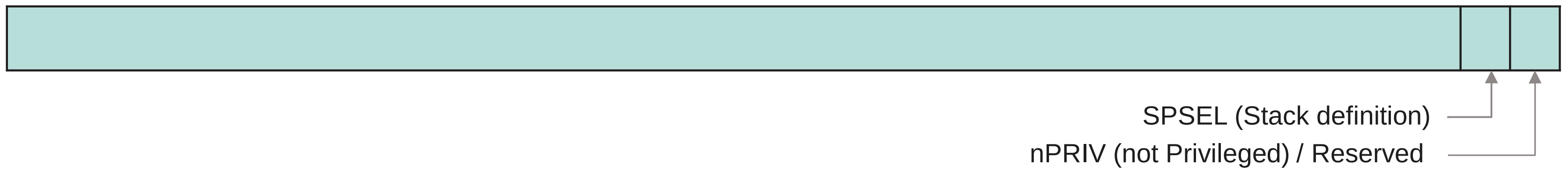


Stack management by CONTROL register

- It is used for accessing the stack memory via PUSH and POP operations..

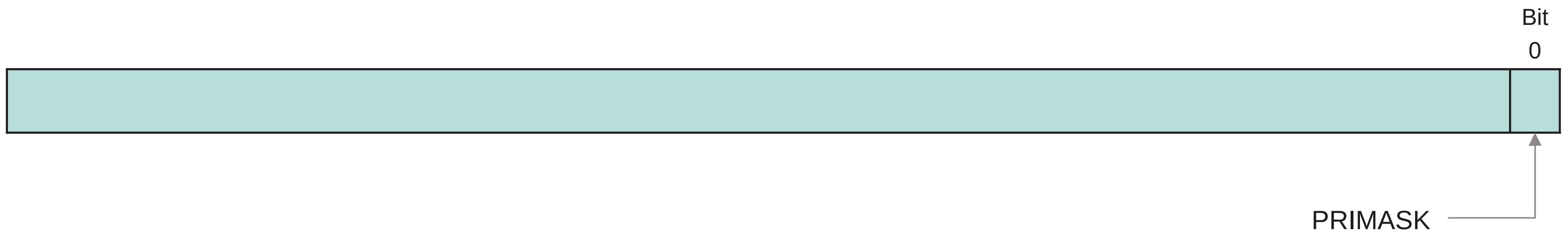
- There are physically two different stack pointers:
  - The Main Stack Pointer (MSP, or SP_main in ARM documentation) is the default Stack Pointer after reset and is used when running exception handlers.
  - The Process Stack Pointer (PSP, or SP_process in ARM documentation) can only be used in Thread mode (when not handling exceptions).

- The stack pointer selection is determined by the CONTROL register, one of the special registers
  - SPSEL -1 (PSP)
  - SPSEL-0 (MSP)



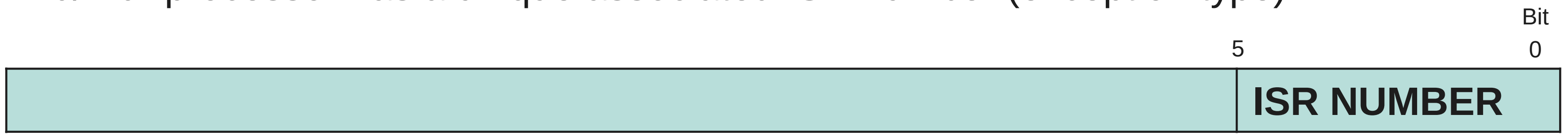
CPU Level interrupt management - PRIMASK and IPSR

– The PRIMASK register is a 1-bit wide interrupt mask register. When set, it blocks all interrupts apart from the Non-Maskable Interrupt (NMI) and the HardFault exception.



PRIMASK Register

– The IPSR contains the current executing ISR (Interrupt Service Routine) number. Each exception on the Cortex-M0/M0$^+$ processor has a unique associated ISR number (exception type).

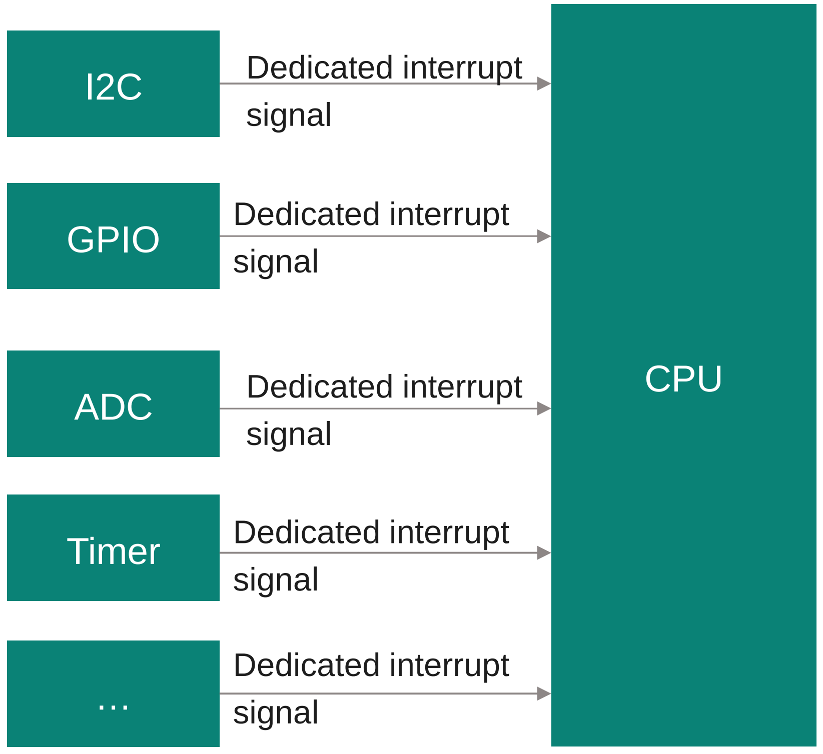


IPSR Register

Different types of Exceptions

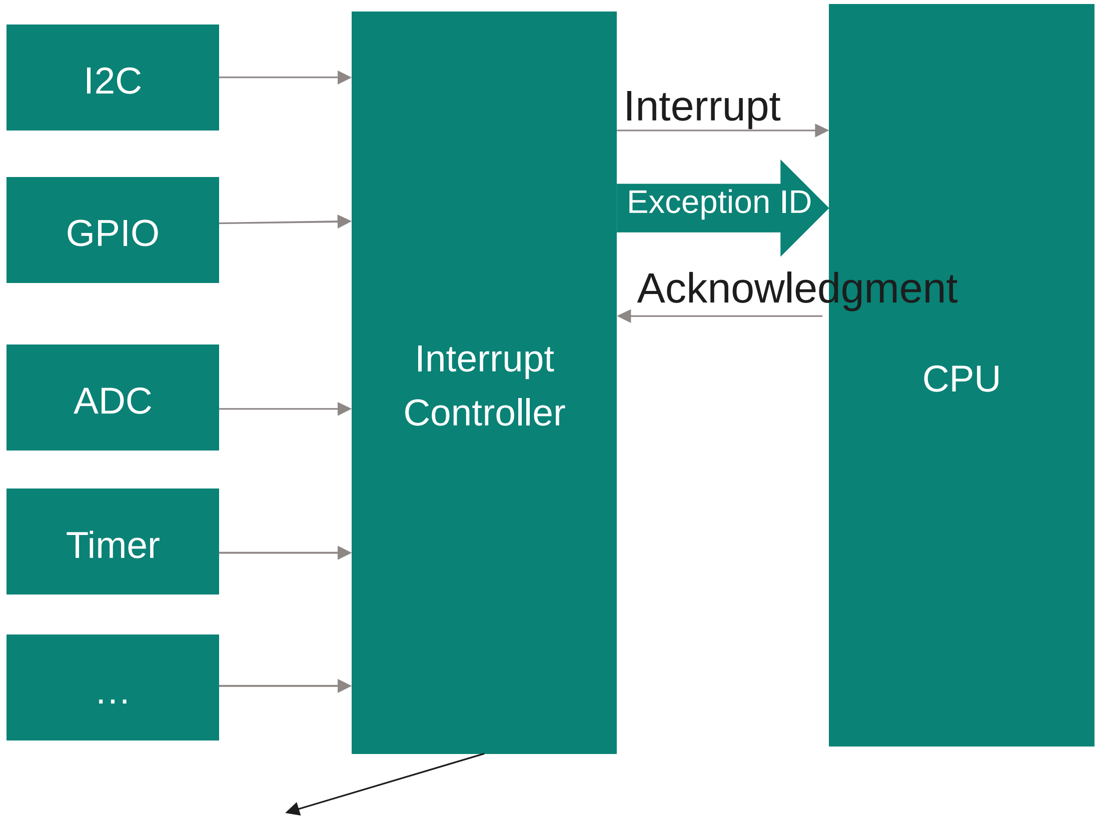
- There are seven types of exception:
  - **Non-Maskable Interrupt:** The NMI is like IRQ, but it cannot be disabled and has the highest priority apart from the reset. It is very useful for safety critical systems like industrial control or automotive.
  - **HardFault**: HardFault is an exception type dedicated for handling fault conditions during program execution
  - **SVCall (Supervisor Call):** SVCall exception takes place when the SVC instruction is executed. SVC is usually used in system with Operating System (OS), allowing applications to access to system services.
  - **Pending Service Call:** Pending Service Call (PendSV) is another exception for applications with OS. But PendSV can be delayed
  - **System Tick Timer:** The SysTick Timer inside the NVIC is another feature for OS application. Almost all OS need a timer to generate periodic interrupt for system maintenance works like context switching etc.
  - **Interrupt**

How to manage several interrupts?


I2C
Dedicated interrupt signal
GPIO
Dedicated interrupt signal
ADC
Dedicated interrupt signal
Timer
Dedicated interrupt signal
…
Dedicated interrupt signal
CPU

Too many signals directly landing on CPU

Never used in practice



Enable/Disable individual interrupts

Assign priorities to interrupts

Maintains pending interrupt status – No interrupt event is lost

Interrupt Controller on Cortex M0+ : NVIC

- The NVIC is a programmable unit that allows software to manage interrupts and exceptions.

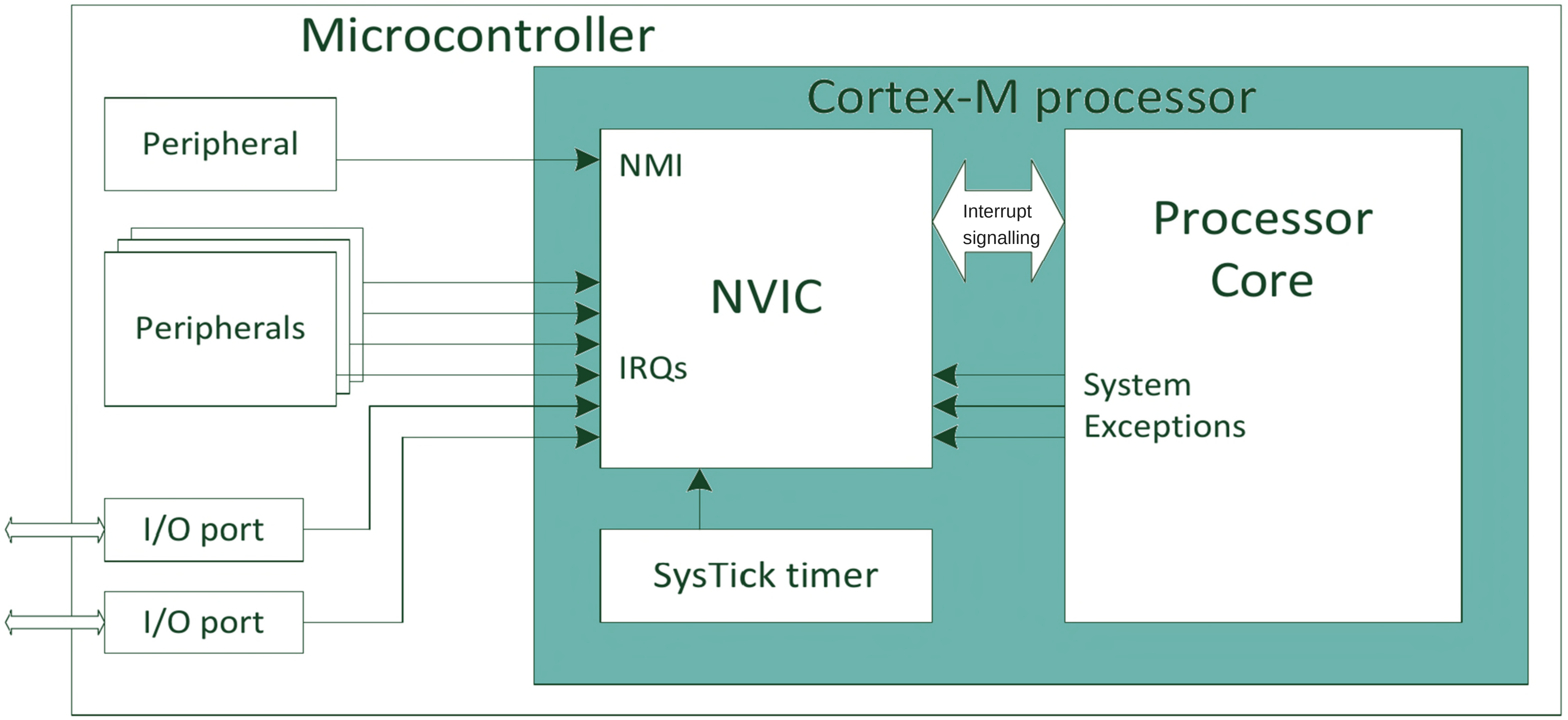
- It has a number of memory mapped registers for the following:

  - Enabling or disabling of each of the interrupts

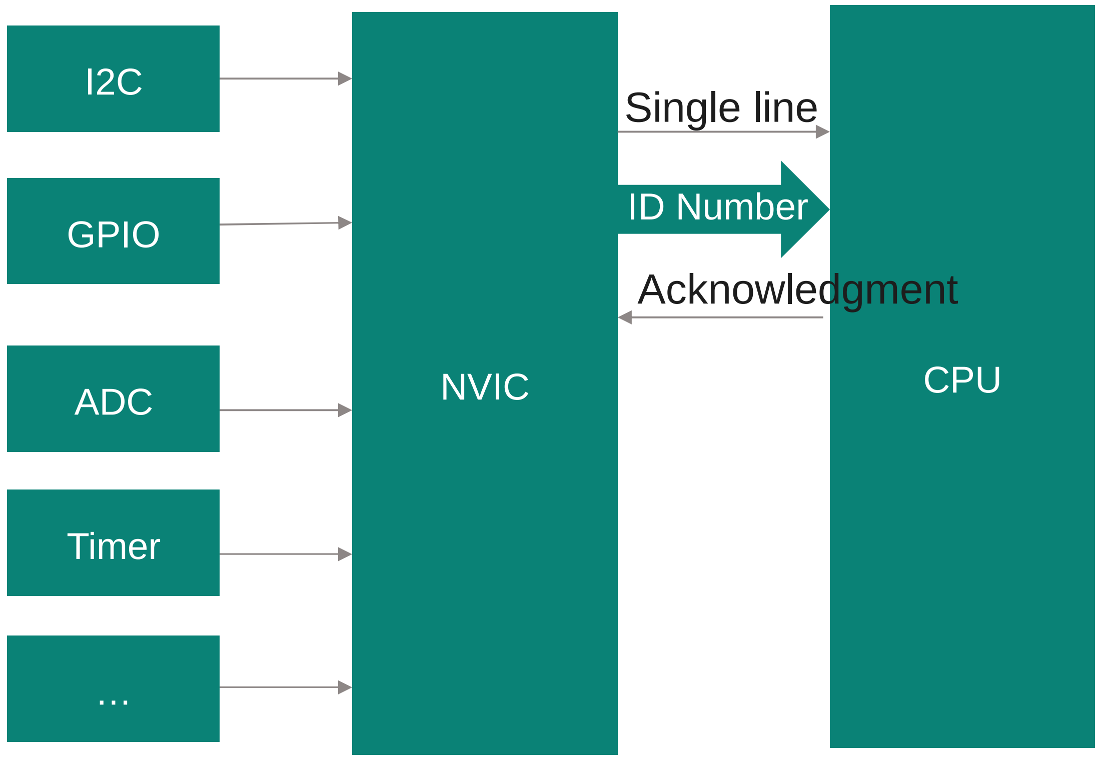
  - Defining the priority levels of each interrupts and some of the system exceptions

  - Enabling the software to access the pending status of each interrupt, including the capability to trigger interrupts by setting pending status in software.

Interrupt Controller



How to enable/disable interrupts?



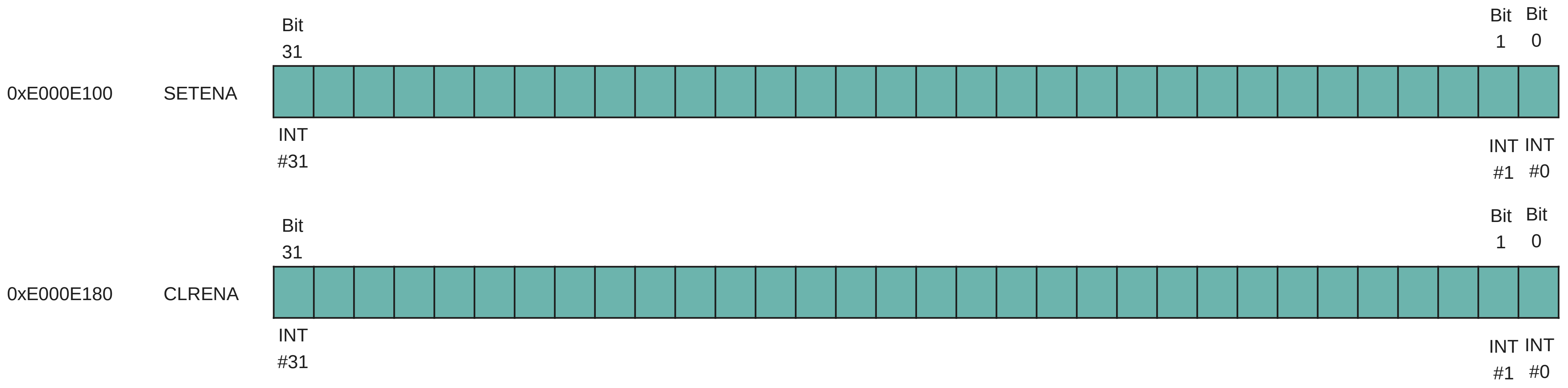
Peripherals generate events. Enable them first

Enable interrupt generation for desired peripheral events

Enable interrupt detection at CPU level

NVIC Registers -> Interrupt Enable and Clear Enable

- Used to control the enable/disable of the IRQs (exception 16 and above).
- Width of this register is 32 bit and minimum size is 1 bit.
- To enable an interrupt, the SETENA address is used, and to disable an interrupt, the CLRENA address is used.



NVIC Registers -> Interrupt Enable and Clear Enable

- /* Enable interrupt #2 */
  - *((volatile unsigned long *)(0xE000E100)) = 0x4;
- /* Disable interrupt #2 */
  - *((volatile unsigned long *)(0xE000E180)) = 0x4;

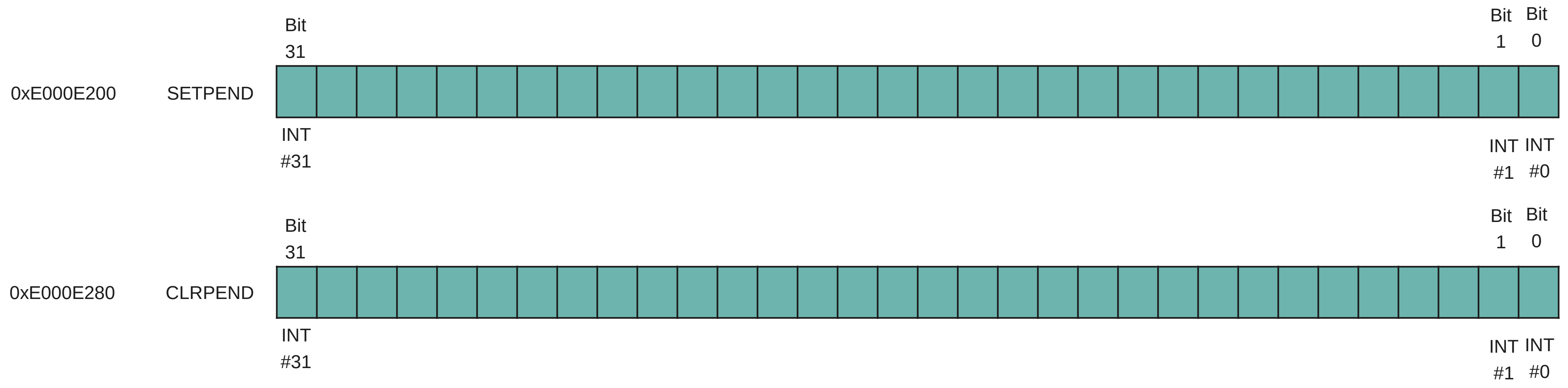
- By using CMSIS Function

- /* Enable Interrupt IRQn value of 0 refer to Interrupt #0 */
  - void NVIC_EnableIRQ(IRQn_Type IRQn);

- /* Disable Interrupt e IRQn value of 0 refer to Interrupt #0 */
  - void NVIC_DisableIRQ(IRQn_Type IRQn);

NVIC Registers -> Interrupt Pending Set and Clear Register

– Interrupt takes place but cannot be processed immediately, for example, if the processor is serving another higher priority interrupt, the IRQ will be pended
– The interrupt pending status can be accessed, or modified, through the Interrupt Set Pending (SETPEND) and Interrupt Clear Pending (CLRPEND) register addresses



Interrupt Pending Set and Clear Register

- /* Enable interrupt #2 */
  - *((volatile unsigned long *)(0xE000E100)) =0x4;
- /* Pend interrupt #2 */
  - *((volatile unsigned long *)(0xE000E200)) =0x4;

- /* Clear interrupt #2 pending status */
  - *((volatile unsigned long *)(0xE000E280)) = =0x4;

- By using a library function (e.g. CMSIS Library)
- /* Set pending status of a interrupt */
  - void NVIC_SetPendingIRQ(IRQn_Type IRQn);
- /* Clear pending status of a interrupt */
  - void NVIC_ClearPendingIRQ(IRQn_Type IRQn);

- /* Return true if the interrupt pending status is 1 */
  - uint32_t NVIC_GetPendingIRQ(IRQn_Type IRQn);

NVIC Registers -> Interrupt Priority Level

- Each external interrupt has an associated priority level register. Each of them is 2 bit wide, occupying the two MSBs of the Interrupt Priority Level Registers.

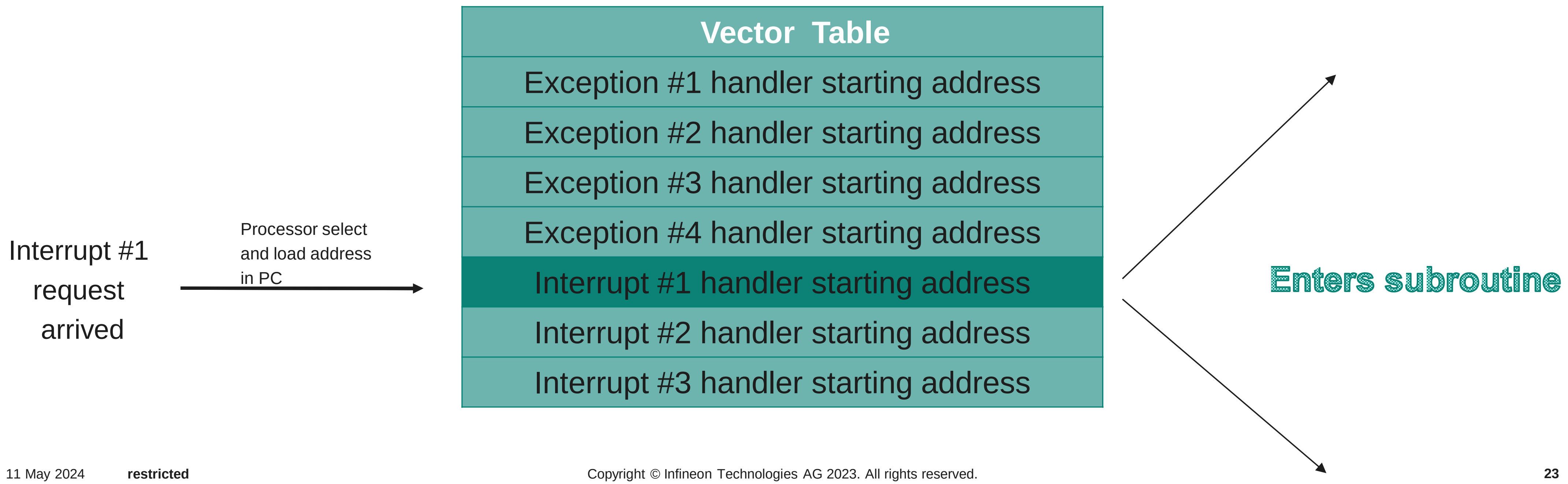
Bit	31 30	24	23 22	16	15 14	8	7 6	0	
0xE000E41C	31		30		29		28		IPR7
0xE000E418	27		26		25		24		IPR6
0xE000E414	23		22		21		20		IPR5
0xE000E410	19		18		17		16		IPR4
0xE000E40C	15		14		13		12		IPR3
0xE000E408	11		10		9		8		IPR2
0xE000E404	7		6		5		4		IPR1
0xE000E400	IRQ 3		IRQ 2		IRQ 1		IRQ 0		IPR0

- By using CMSIS function
- // Set the priority level of an interrupt or a system exception
- void NVIC_SetPriority(IRQn_Type IRQn, uint32_t priority);
- // return the priority level of an interrupt or a system exception
- uint32_t NVIC_GetPriority(IRQn_Type IRQn);

Vector Table

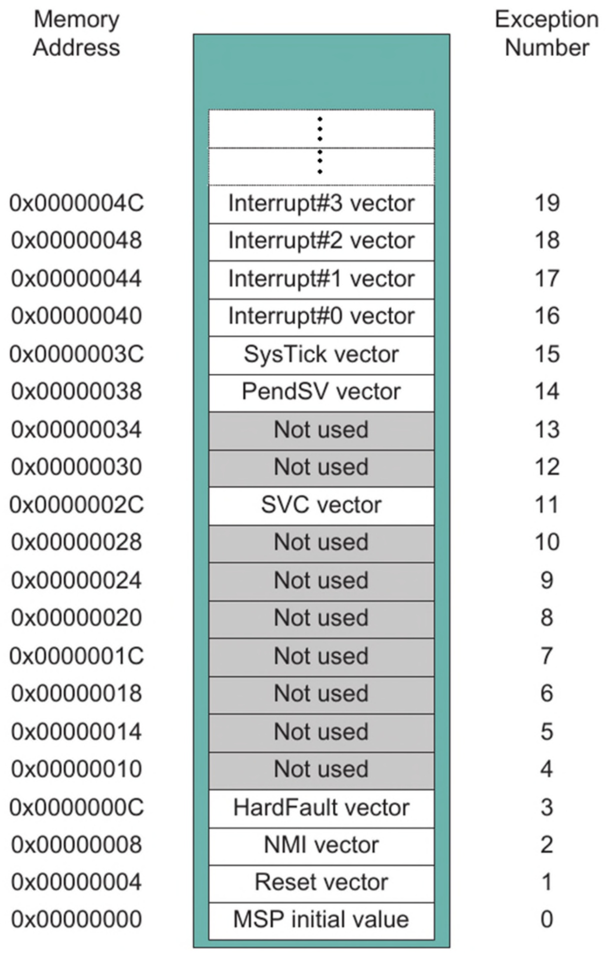
- The interrupt handling in the Cortex-M Processor is vectored, which means the processor's hardware automatically determines which interrupt or exception to service.

- After receiving an IRQ of exception event, it will need to know the starting address of the handler, and the vector table is a lookup table in the memory that provides such information.


Interrupt #1 request arrived
Processor select and load address in PC
Enters subroutine

Vector Table
Exception #1 handler starting address
Exception #2 handler starting address
Exception #3 handler starting address
Exception #4 handler starting address
Interrupt #1 handler starting address
Interrupt #2 handler starting address
Interrupt #3 handler starting address

Vector Table(conti.)



Memory Address		Exception Number
	⋮	
	⋮	
0x0000004C	Interrupt#3 vector	19
0x00000048	Interrupt#2 vector	18
0x00000044	Interrupt#1 vector	17
0x00000040	Interrupt#0 vector	16
0x0000003C	SysTick vector	15
0x00000038	PendSV vector	14
0x00000034	Not used	13
0x00000030	Not used	12
0x0000002C	SVC vector	11
0x00000028	Not used	10
0x00000024	Not used	9
0x00000020	Not used	8
0x0000001C	Not used	7
0x00000018	Not used	6
0x00000014	Not used	5
0x00000010	Not used	4
0x0000000C	HardFault vector	3
0x00000008	NMI vector	2
0x00000004	Reset vector	1
0x00000000	MSP initial value	0

Note : LSB of each vector must be set to 1 to indicate Thumb state

Exception Entry Sequence

- When an exception takes place, a number of things happen as follows:

Stacking and update of one of the Stack Pointers (SPs)

Vector fetch (determine starting address of ISR) and update R15 (PC)

Registers update (LR, Internal Program Status Register (IPSR), NVIC registers)

Stacking – 1/2

– When an exception takes place, eight registers are pushed:

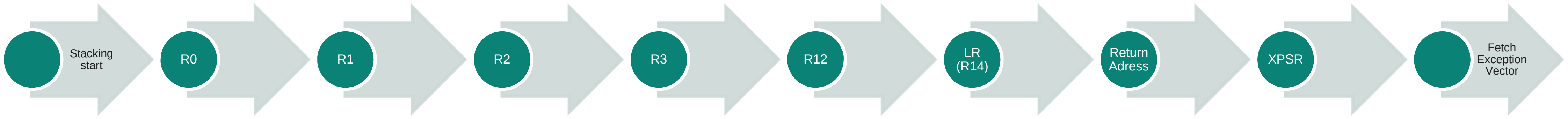
R0 – R3

R12

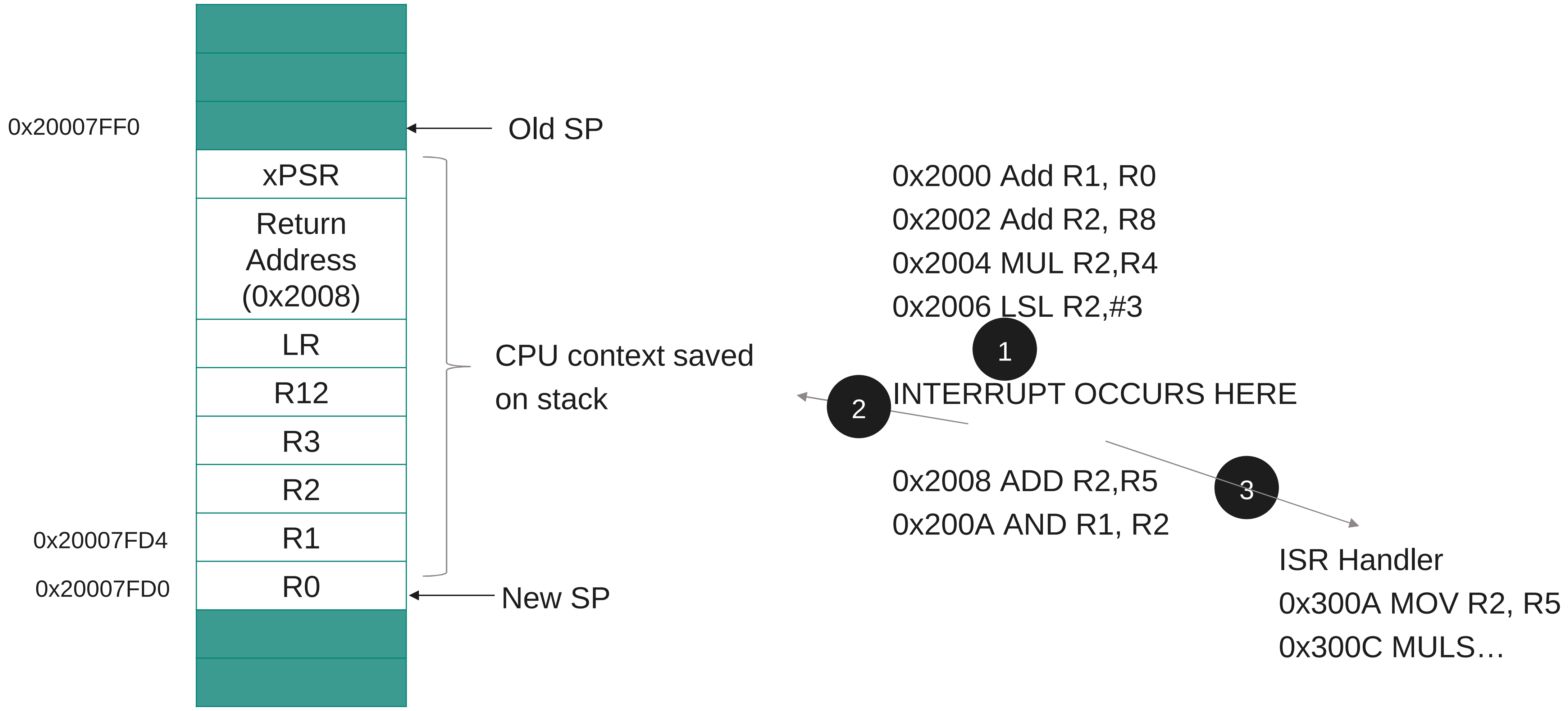
R14(LINK REGISTER)

The return address/PC

xPSR



Stacking – 2/2



0x2000 Add R1, R0
0x2002 Add R2, R8
0x2004 MUL R2,R4
0x2006 LSL R2,#3

INTERRUPT OCCURS HERE

0x2008 ADD R2,R5
0x200A AND R1, R2

ISR Handler
0x300A MOV R2, R5
0x300C MULS…

Vector Fetch and Update PC

I

- After the stacking, the processor then fetches the exception vector (starting address of the ISR) from the vector table

II

- Vector updated to the PC.

III

- Instruction fetch of the exception handler execution starts from this address.

Additional Registers Updates

II

- the IPSR is also updated to the exception number of current serving exception.

III

- In addition, a few NVIC registers get updated

Returning back to the interrupted subroutine

I

- Registers originally pushed to stack during interrupt entry sequence are popped and written back to the registers
- Interrupt stack pointer adjusted

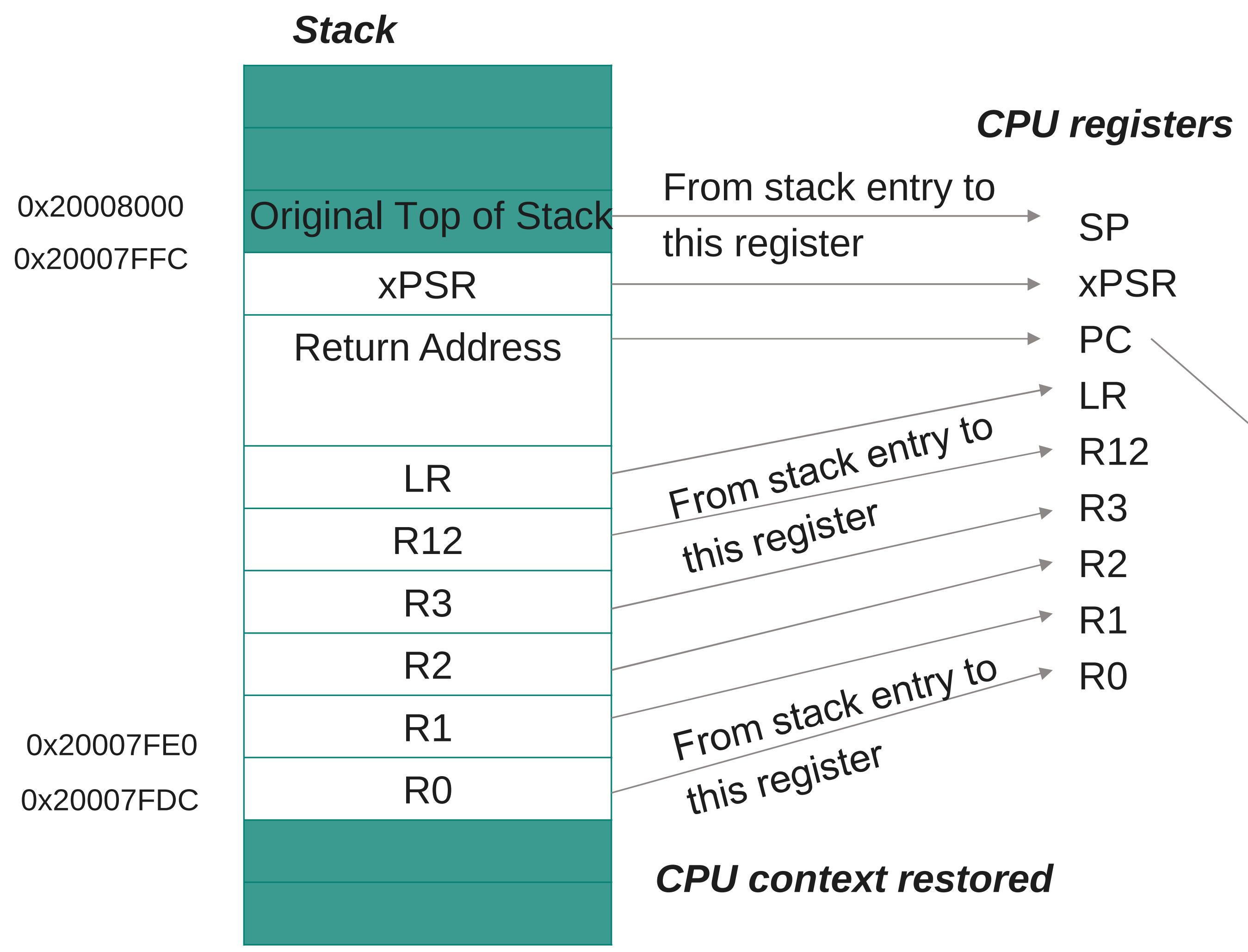
II

- Program counter is loaded with return address taken from Link Register

III

- CPU executes instructions fetched from the address of the interrupted subroutine

Returning back from interrupt

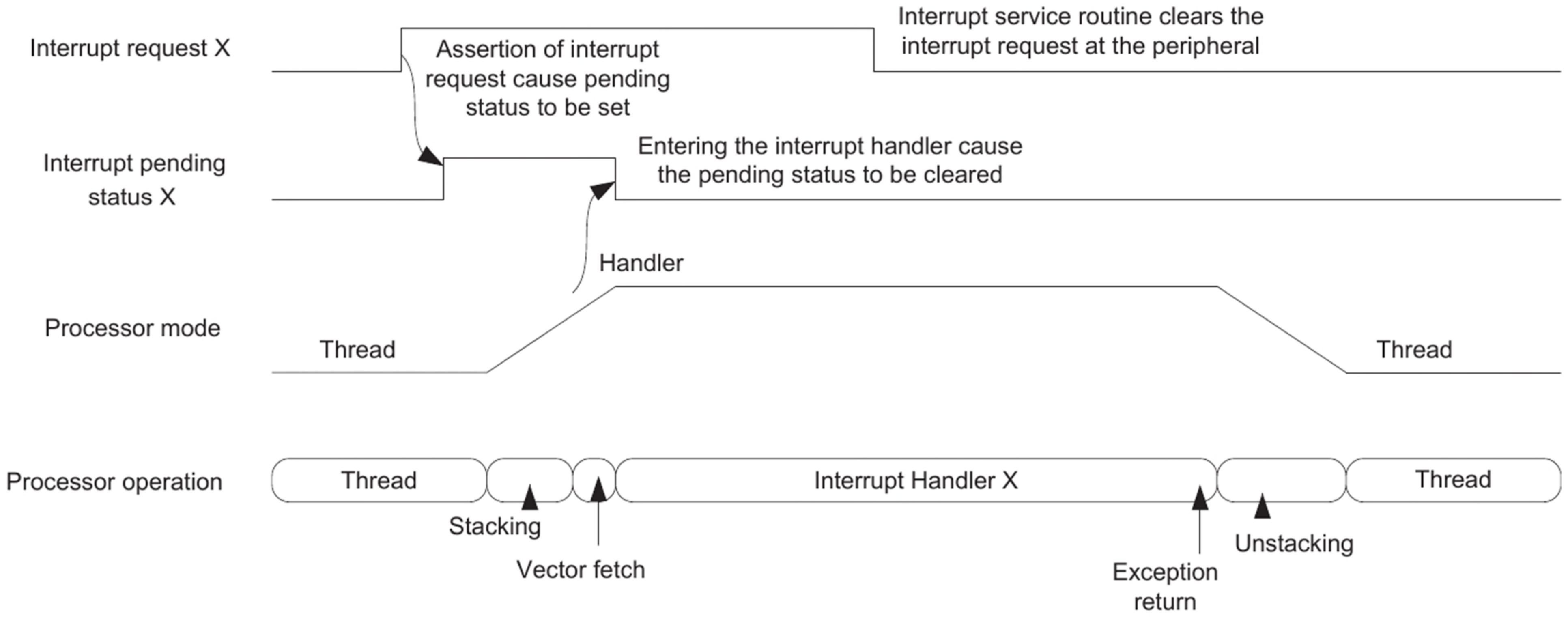


```
0x2000 Add R1, R0
0x2002 Add R2, R8
0x2004 MUL R2,R4
0x2006 LSL R2,#3

0x2008 ADD R2,R5
0x200A AND R1, R2
```

CPU resumes execution of interrupted routine correctly

Interrupt Process


Interrupt request X
Assertion of interrupt request cause pending status to be set
Interrupt service routine clears the interrupt request at the peripheral
Interrupt pending status X
Entering the interrupt handler cause the pending status to be cleared
Handler
Processor mode
Thread
Thread
Processor operation
Thread
Interrupt Handler X
Thread
Stacking
Vector fetch
Exception return
Unstacking

infineon