ECE/CS 5780-6780 : Embedded Systems Design

Lect. 06: Memory manipulation - Interrupts

Pierre-Emmanuel Gaillardon
Department of Electrical and Computer Engineering – University of Utah

Spring 2017
Salt Lake City, UT, USA

Announcements – HW1

- HW1: So far, 25 submissions have been reviewed – we will continue to review and provide feedbacks today
- Feedbacks are provided as comments on Canvas
- When you re-upload your submission, put a note on Canvas so it gets easier to track.
- Questions regarding HW

Objectives for Today!

- Get familiar with the ARM M processor architecture
- Understand the architecture of a microcontroller
- **Acquire the fundamentals of the hardware/software interface**
- Acquire the fundamentals for sensing and controlling the physical world
- Learn modeling techniques for embedded system design
- **Understand the usage of several peripherals** through labs
- Design a complete embedded system – from specs to realization
  - Realization of a PCB
  - Behavioral modeling of the system
  - Complete SW realization

Memory-manipulation in Embedded Software con't

Bit-wise Data manipulation

- Why does it matter?
- Let's have a look at the MCU datasheet
- We need to be able to read/write single bits in a word without disturbing the initial content!

- **Bit-wise Data Manipulation**
  - & and | → Bit-wise AND and OR
  - << and >> → Bit shifting

- Do not get these operators mixed up with && and || the logic level AND and OR operators

Manipulation Examples

- Let's take an example on a 8-bit data
- Data to manipulate: D=0x34 (0011 0100)
- How can I extract Bit 4?

Tmp = D & 0x10 (0001 0000) = 0x10 (0001 0000)

Tmp = Tmp >> 4 = 1 (0000 0001)

- How can I force Bit 4 to be set to 0?

D & 0xEF (1110 1111) = 0x24 (0010 0100)

- How can I force Bit 3 to be set to 1?

D | 0x08 (0000 1000) = 0x3C (0011 1100)

Variables : Local or Global?

- In ES, variables are more important than in regular PC, as we are resources-constrained.
- Local variables
  - Defined within a C block { }
  - Visible only within this block.
- Global variables
  - Defined outside C blocks
  - Visible everywhere
  - Assignment of a fixed memory location by the compiler
  - Need *extern* prefix to see it in any other C files.
- In ES, we make a large use of global functions – better than pointers

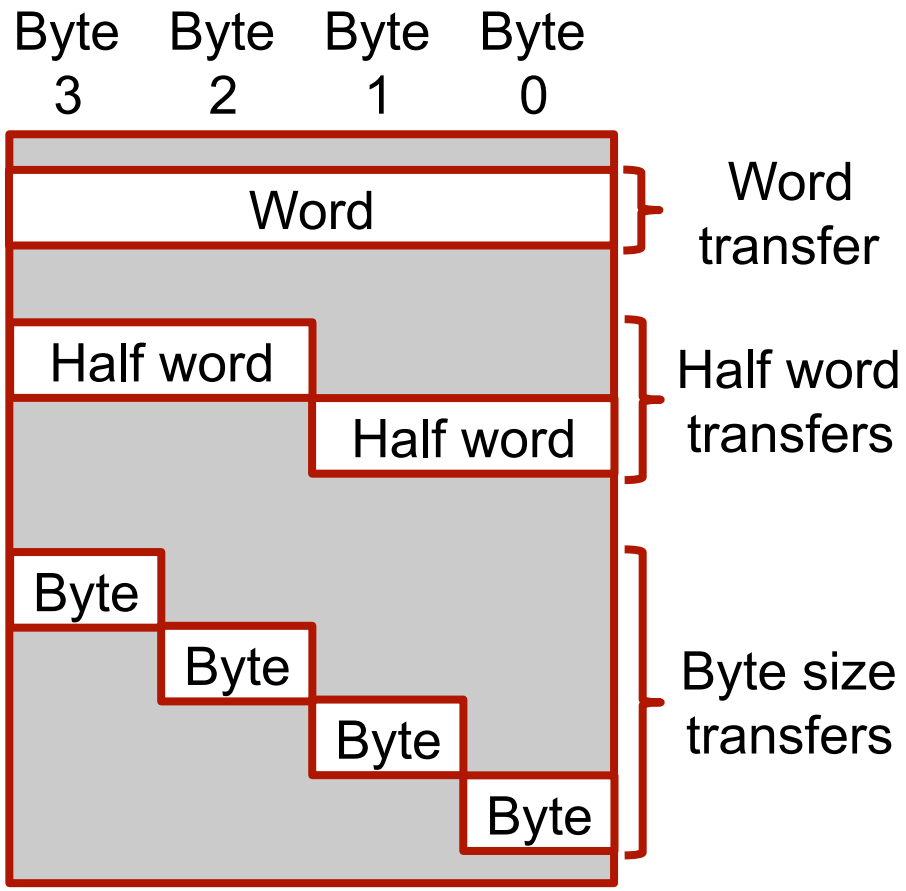
Some Useful C Keywords

- Const
  - Makes variable value or pointer parameter unmodifiable
  - *const int foo = 32;*
- Register
  - Tells compiler to locate variables in a CPU register if possible
  - *register int x;*
- Static
  - Preserve variable value after its scope ends
  - Does not go on the stack
  - *static int x;*
- Volatile
  - Opposite of const
  - Can be changed in the background
  - *volatile int i;*

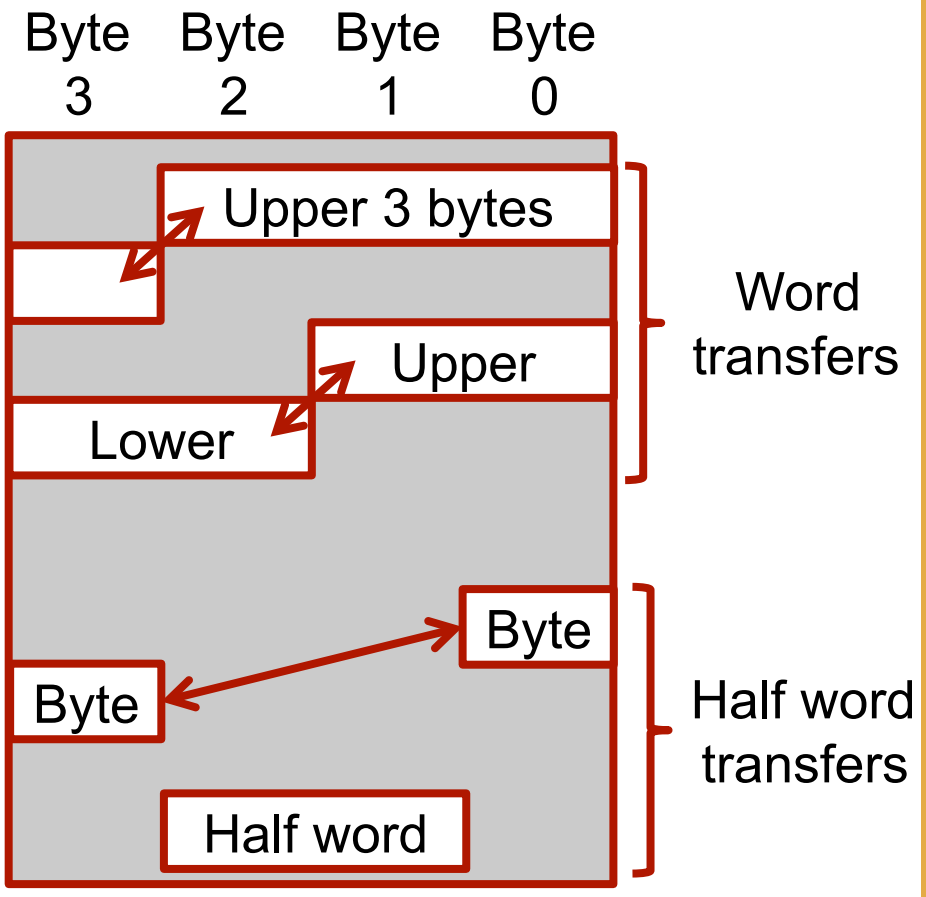
Effect of Hardware Behavior on Programming

Data Alignment

With Thumb instructions, only aligned data transfers are allowed.



Aligned transfers



Unaligned transfers

Misaligned Transfers

- What happens if you initiates an unaligned transfer?

  → you get an HardFault

  (equivalent of a segmentation fault on a PC…)

- How could I get that?

char a[4]; // a 4 byte character array

Althought the size is 4 bytes, these bytes are not necessary aligned to word boundary.

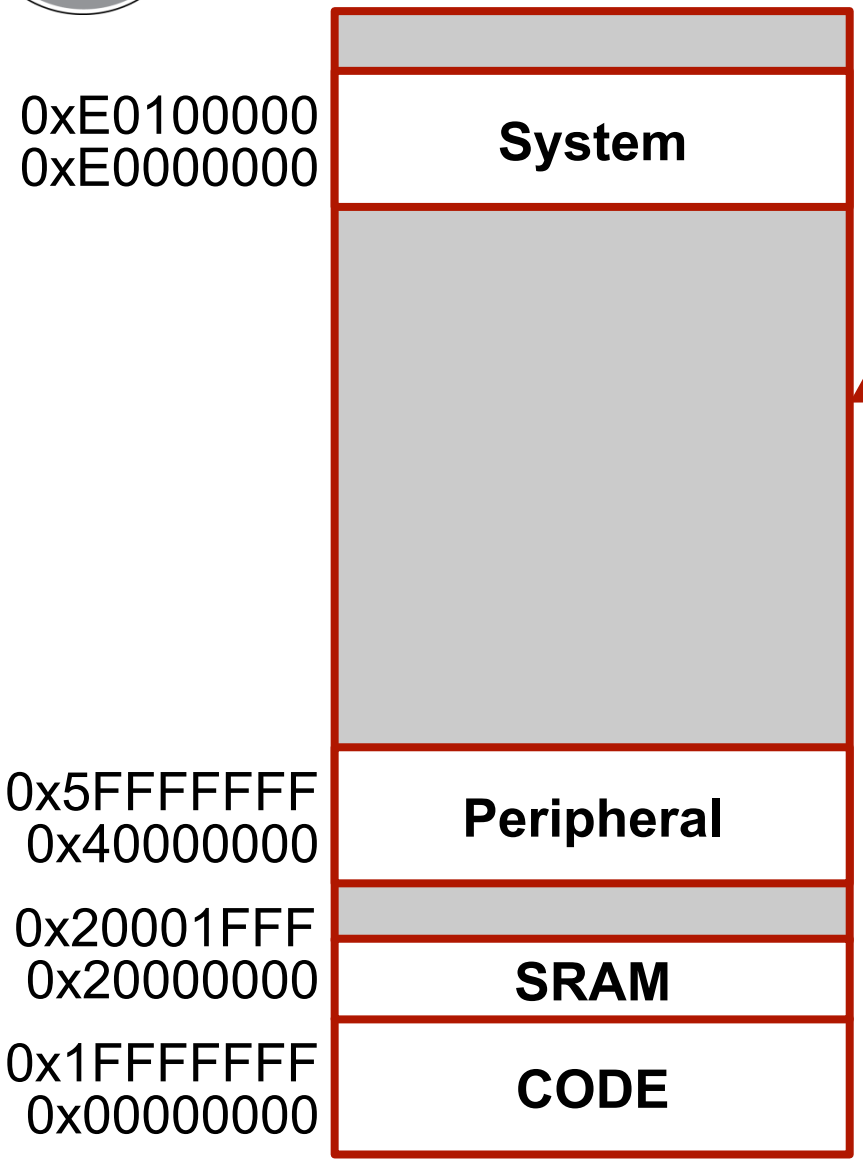
***a = (int) tmp;**

You cast a 32-bit data point to this 32-bit array.

If the array is unaligned → HardFault!

Be careful with pointers!

Access to Invalid Addresses



What happens if you initiate a read operation here?

HardFault exception!

Quite a useful feature for debugging or creating secured systems!

Wait States

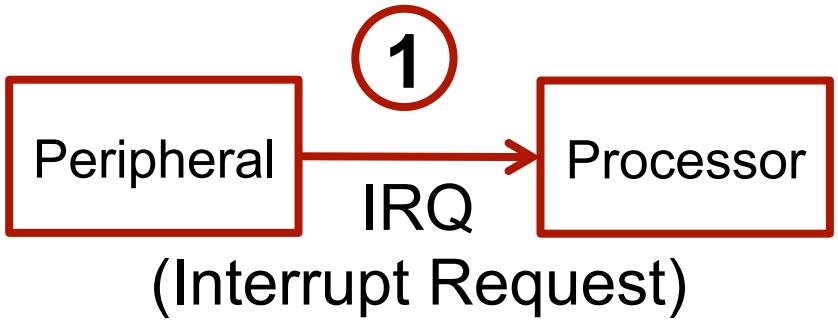
- Some of the memory accesses might take several clock cycles to complete.
- For example, a Flash memory is usually slower than the core of the processor – so during an operation the processor need to wait for the peripheral to be ready.
- Wait states are basically wasted clock cycles.
- It affects:
  - The performance of the system
  - The energy efficiency of the system
  - The interrupt latency
  - The system behavior is less deterministic
- Prefer using the on-chip resources than external ones.

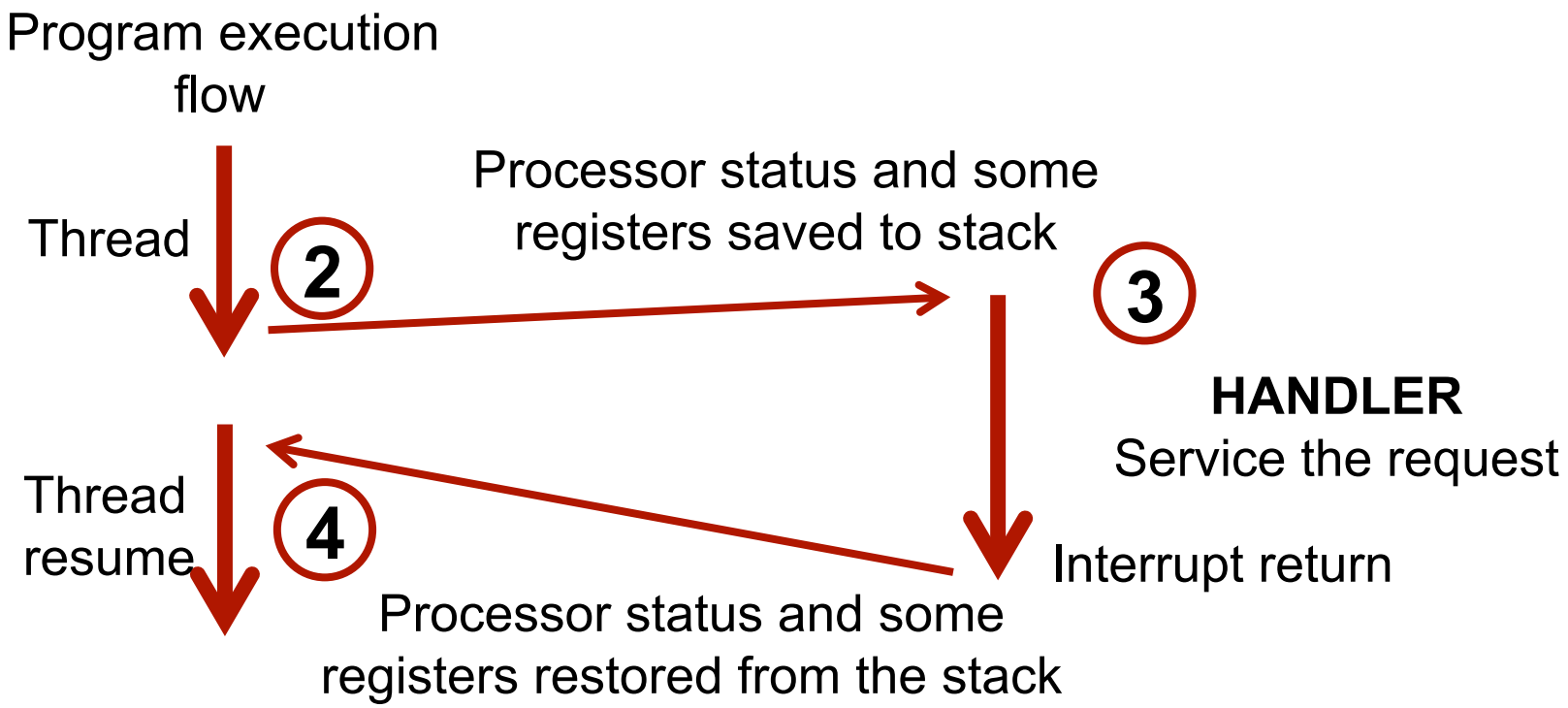
Interrupts

Exceptions and Interrupts

- In most microcontrollers, the interrupt feature enables a peripheral (internal or external) to execute a piece of code to service the request.
- The process involves suspending the current task, branching to a specific piece of code (called exception handler), and resuming back.
- This can be seen as an unexpected function call triggered by some third party.
- Exceptions and interrupts have similar mechanisms:
  - Interrupts: Come from external (of the core) hardware
  - Exceptions: Come from the internal system (div by 0)

Interrupt Handling Concept





Some Definitions

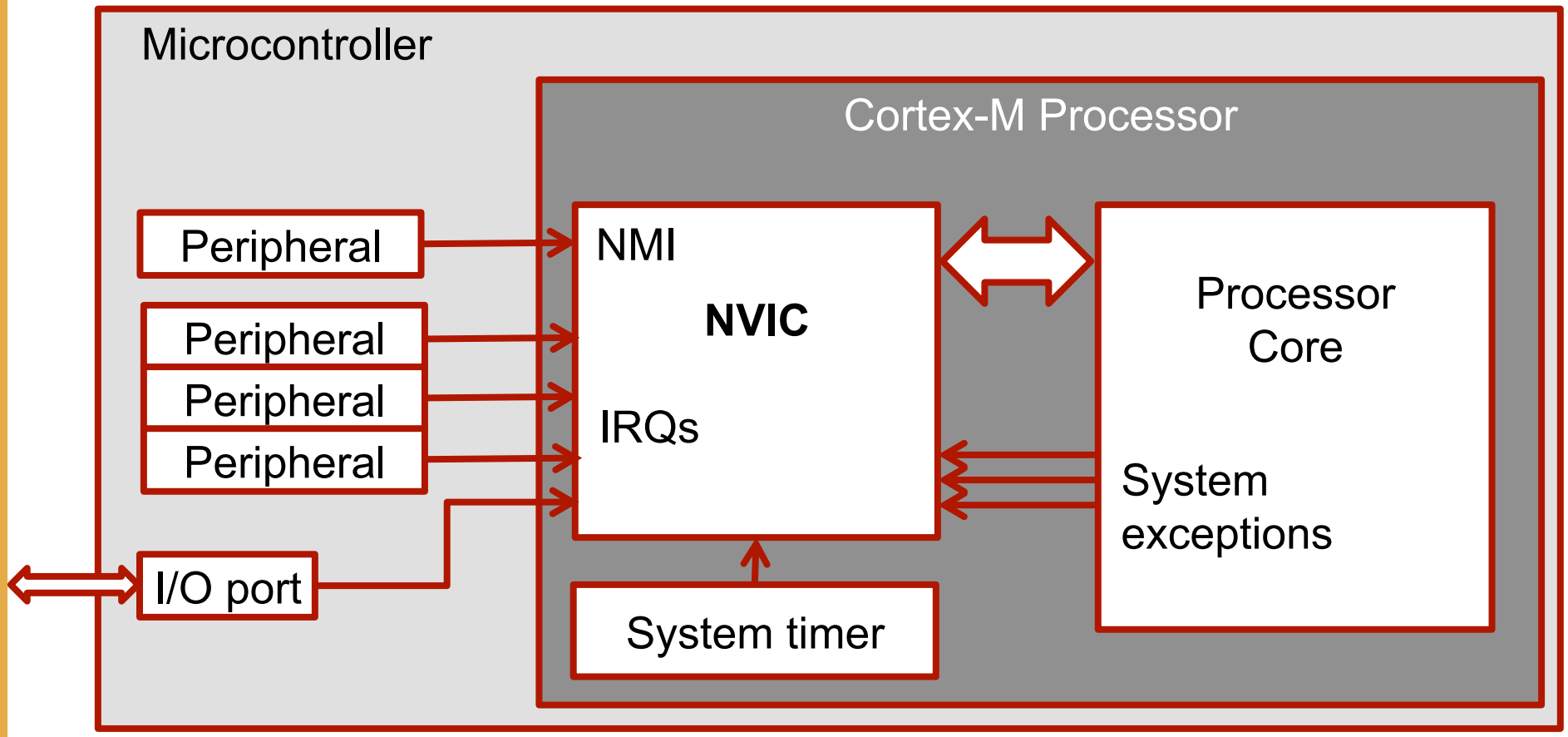
- **Interrupt Requests** (IRQs): One of the exception type associated with peripherals (including external INT pins).

  The Cortex-M0 processors support up to 32 IRQ inputs
- **Non-Maskable Interrupt** (NMI): A special IRQ that cannot be disabled and associated with highest priority level.

  Typically generated by peripherals like watchdog timer or Brown Out Detector (BOD)
- **Handlers**: The software code that gets executed.
- **Nested Interrupts**: When an interrupt with a higher priority takes priority over another interrupt.

How do Interrupts work?

You need some special hardware to control that.
The **Nested Vectored Interrupt Controller** (NVIC)



NVIC Exception Types

Exception number	Exception type	Priority	Descriptions
1	Reset	-3 (Highest)	Reset
2	NMI	-2	Non-Maskable Interrupt
3	HardFault	-1	Fault handling exception
4 – 10	Reserved	NA	-
11	SVCall	Programmable	Supervisor call (OS)
12 – 13	Reserved	NA	-
14	PendSV	Programmable	Pendable request call (OS)
15	SysTick	Programmable	System Tick timer (OS)
16	IRQ #0	Programmable	External interrupt #0
17	IRQ #1	Programmable	External interrupt #1
…	…	…	…
47	IRQ #31	Programmable	External interrupt #31

NVIC Exception Types – STM32F072

Exception number	Exception type	Priority	Descriptions
1	Reset	-3 (Highest)	Reset
2	NMI	-2	**Clock Security System**
3	HardFault	-1	Fault handling exception
4 – 10	Reserved	NA	-
11	SVCall	Programmable	Supervisor call (OS)
12 – 13	Reserved	NA	-
14	PendSV	Programmable	Pendable request call (OS)
15	SysTick	Programmable	System Tick timer (OS)
16	IRQ #0	Programmable	**Watchdog Timer**
17	IRQ #1	Programmable	**Programmable V Detector**
…	…	…	**…**
47	IRQ #31	Programmable	**USB**

Nested Vectored Interrupt Controller

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

- It has a number of memory mapped registers to:
  - Enable/disable each of the interrupts
  - Define the priority levels
  - Enable the software to access the pending status of each interrupts (and to trigger them by software)

- It also has a special PRIMASK register to disable all interrupts and exceptions with positive priorities.

Priority Levels

- Each exception has a priority level.
- The priority level affects whether the exception will be carried out or waits until later.
- The Cortex-M0 supports 3 fixed highest priority levels and 4 programmable levels.
- The programmable priority levels are implemented in 8-bit wide registers, but only the 2 MSBs are implemented.

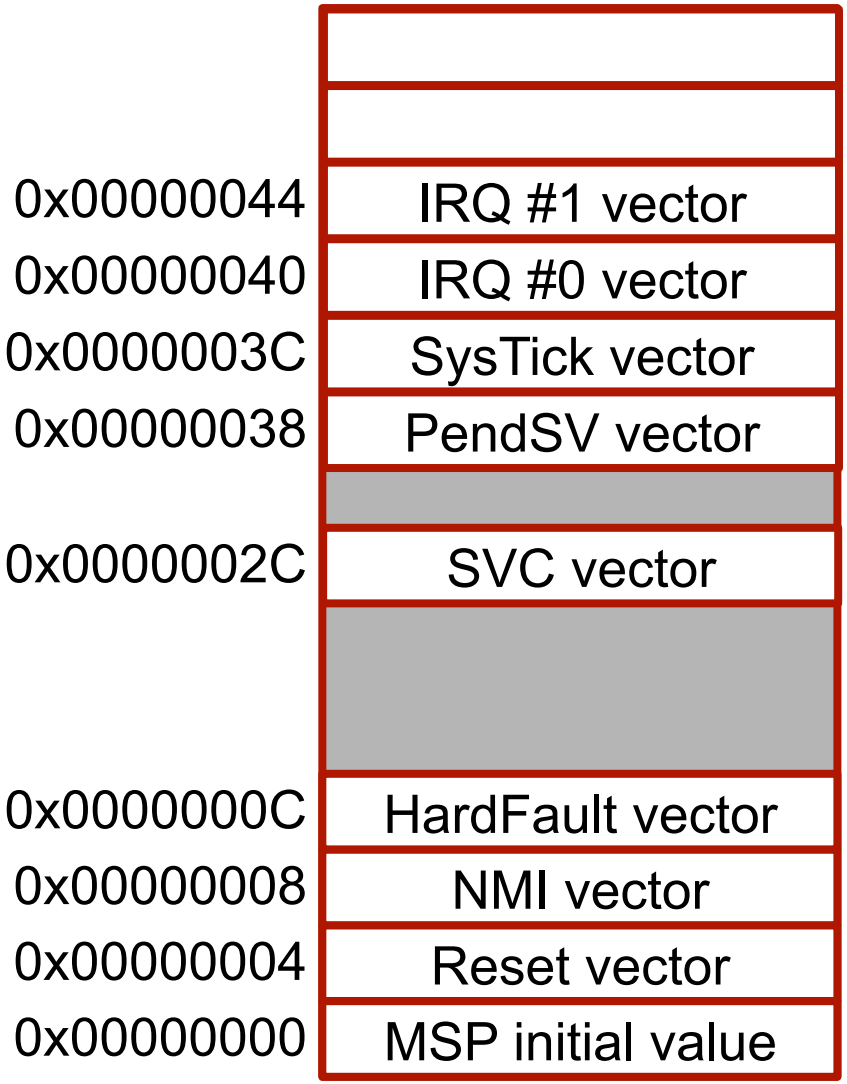
- Priority values: 0x00 (highest), 0x40, 0x80, 0xC0

Impact of Priority Levels

If the processor is already servicing an interrupt,

- but receive a new exception with higher priority
  - then preemption will take place.
  - The running exception handler is suspended and the new exception handler is executed. This is called **Nested Interrupts**.
  - Once serviced, the processor goes back to the previous interrupt handler.
- and receive a new exception with the same priority (or lower)
  - Then the new exception will have to wait by entering a pending state.

Vector Table



Once the NVIC accepts to service an exception request, it branches to the associated handler.

The vector table, located in the CODE memory space, contains the addresses of all the handlers.

How do I put code in the Vector Table?

- It is already done in **vectors_stm32f0xx.c**

- This file contains the function name used for each entry of the vector table.
- It also provides a weak default definition of the handler function that only contains a while(1) loop.

- As a general conclusion, you do not need to worry on how to put an address in the vector table, as you simply have to use the right function name.
- For example, **void USART1_IRQHandler(void);**

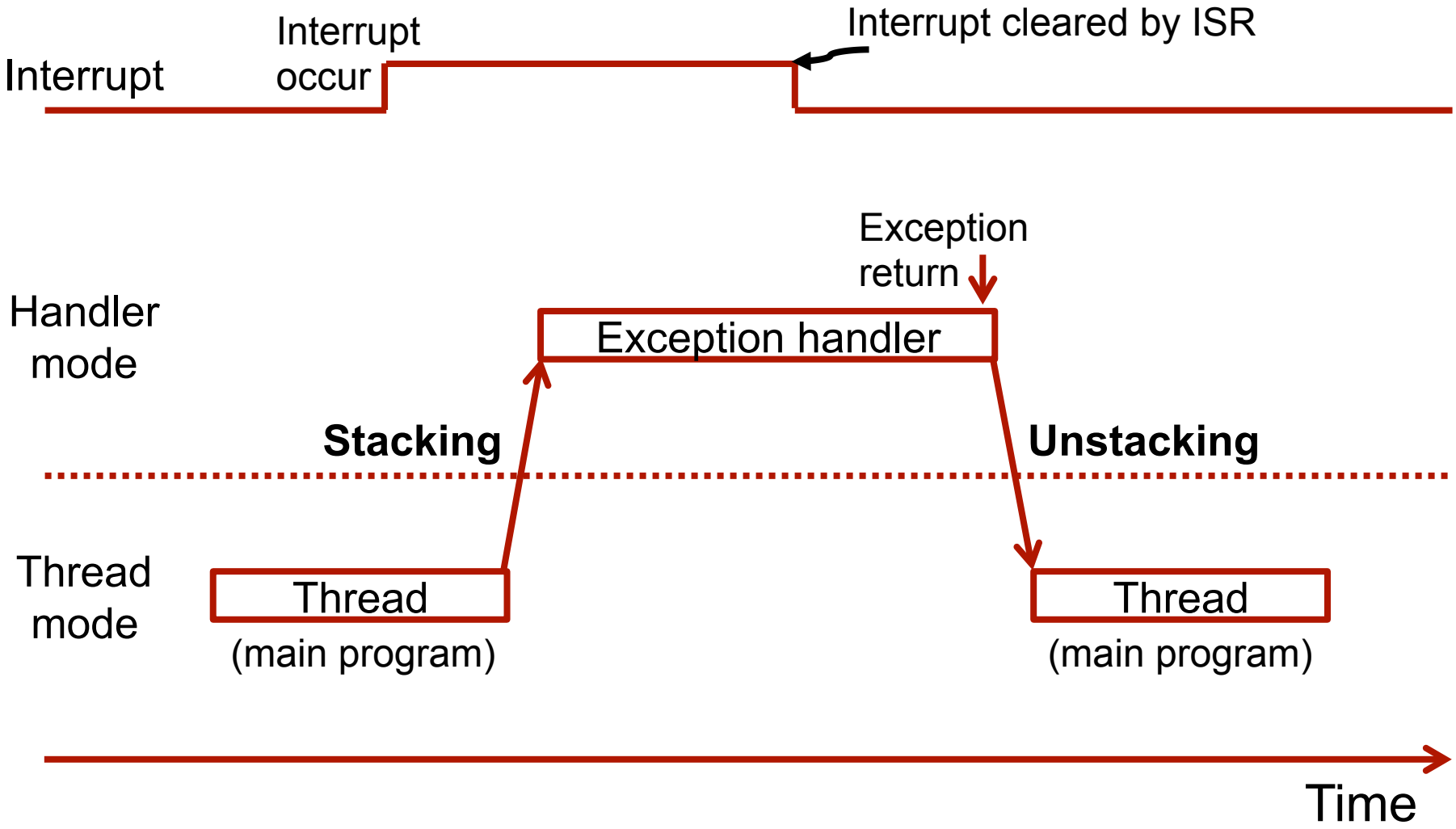
Exception Sequence: Conditions

- The processor accepts an exception if the following conditions are satisfied:
  - The processor is not halted for debugging.
  - For interrupt and SysTick IRQs, the interrupt has been enabled.
  - The processor is not running an exception handler of same or higher priority.
  - The exception is not block by the PRIMASK interrupt masking register.

Exception Sequence: Stacking/Unstacking

- To allow an interrupted program to be resumed correctly, the current state of the processor must be saved before branching to the handler.
- The Cortex-M processors use a mixture of automatic hardware arrangement and, if necessary, additional software steps.
- The interrupt latency is 16 clock cycles.
- When an exception is accepted,
  - the registers *r0* to *r3*, *r12*, *r14 (LR)*, *PC*, *xPSR* are pushed to the stack.
  - LR is updated with a special value indicating how to exit the handler (returning to a previous handler or reading back the branching address from the stack).

Exception Sequence: Mechanism



Exception Sequence: Stacking/Unstacking

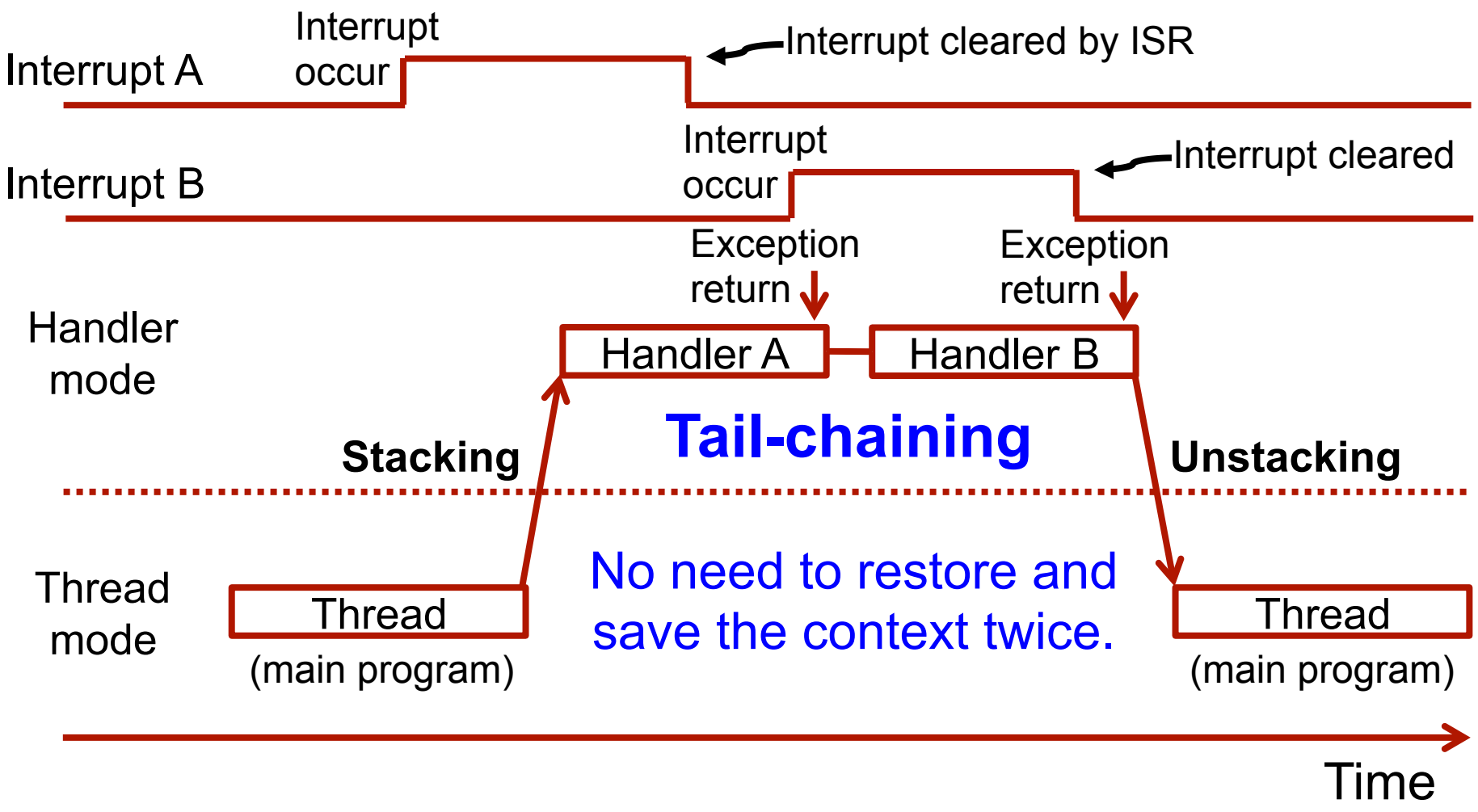
- We mentioned that the Cortex-M processors use a mixture of automatic hardware arrangement and, if necessary, **additional software steps**.

- In particular, the registers not saved by the automatic stacking software (*r4* to *r11*) process have to be saved and restored by software in the exception handler if modified by the exception handler.

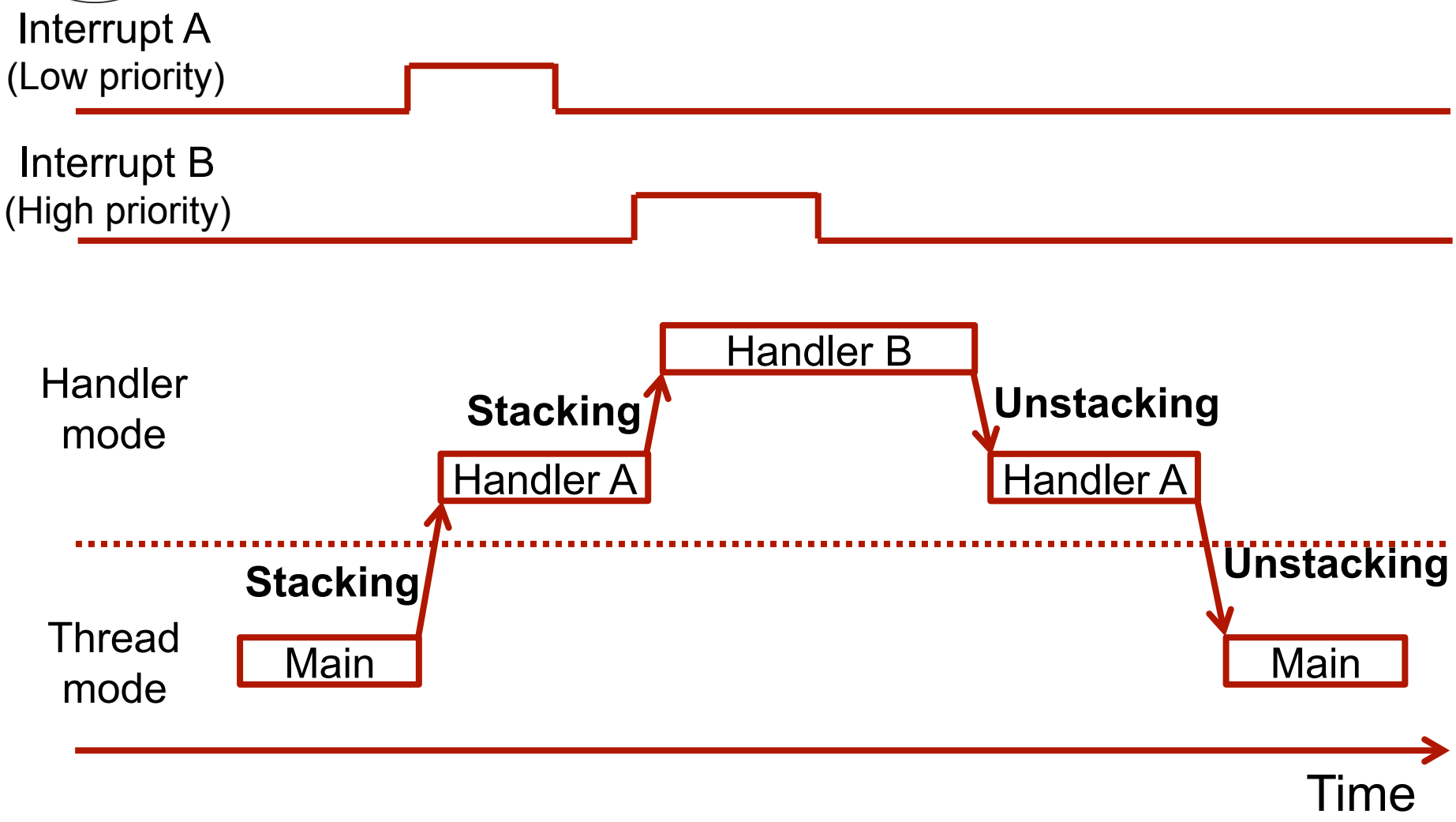
- The good news is that it doesn't usually affect normal C functions because it is a requirement for C compilers to save and restore these registers if they will be modified during the C function execution.

Tail Chaining

This happens when an exception is pending when another handler is completed



Priority Preemption Mechanism



The return context mechanism is controlled by special value in *LR*.

NVIC Control Registers

- The NVIC control registers are memory mapped.
- Their addresses are part of the System space.
- These registers take care of:
  - Enabling/disabling interrupts
  - Controlling the priority level of interrupts
  - Accessing to the pending status of each interrupts
- Possibility to either:
  - Control the registers manually
  - Use the standardized APIs

Interrupt Enable and Clear Enable

Address	Name	Type	Reset value	Descriptions
0xE000E100	SETENA	R/W	0x00000000	Set enable for interrupt 0 to 31. Write 1 to set bit to 1. Write 0 has no effect. Bit[0] for Interrupt #0 (exception 16) …
0xE000E180	CLRENA	R/W	0x00000000	Clear enable for interrupt 0 to 31. Write 1 to clear bit to 0. Write 0 has no effect. Bit[0] for Interrupt #0 (exception 16) …

The NVIC provides an interesting mechanism to set and clear bits.

The two registers SETENA and CLRENA acts on an internal register effectively controlling the interrupt lines.

***((volatile unsigned long *) (0xE000E100)) = 0x04.**

Interrupt Pending Set and Clear Register

Address	Name	Type	Reset value	Descriptions
0xE000E200	SETPEND	R/W	0x00000000	Set pending for interrupt 0 to 31. Write 1 to set bit to 1. Write 0 has no effect. Bit[0] for Interrupt #0 (exception 16) …
0xE000E280	CLRPEND	R/W	0x00000000	Clear pending for interrupt 0 to 31. Write 1 to clear bit to 0. Write 0 has no effect. Bit[0] for Interrupt #0 (exception 16) …

These registers indicate if the interrupt has to be serviced.

Entering an handler clears automatically the associated pending bit.

Note that these bits are manipulated by software (you) and by the hardware requesting interrupt – The hardware interrupt request must be cleared manually.

Interrupt Priority Level

Bit	31 30	24	23 22	16	15 14	8	7 6	0
0xE000E41C	31		30		29		28	
0xE000E418	27		26		25		24	
0xE000E414	23		22		21		20	
0xE000E410	19		18		17		16	
0xE000E40C	15		14		13		12	
0xE000E408	11		10		9		8	
0xE000E404	7		6		5		4	
0xE000E400	IRQ3		IRQ2		IRQ1		IRQ0	

The NVIC registers can only be accessed using word-size transfers.

Need to use bit masking!

```
unsigned long temp;
temp = *((volatile unsigned long *)(0xE000E400));
temp = temp & (0xFF00FFFF) | (0xC0 << 16);
*((volatile unsigned long *)(0xE000E400)) = temp;
```

Or use an API!

```
NVIC->IP[USART1_IRQn] = 0x00;
```

Thank you for your attention

Questions?

Laboratory for NanoIntegrated Systems
Department of Electrical and Computer Engineering
MEB building – University of Utah – Salt Lake City – UT – USA