



# ARM Based Microcontroller

**EXTI**

Lecture 5



# Advanced RISC Machines

# Introduction

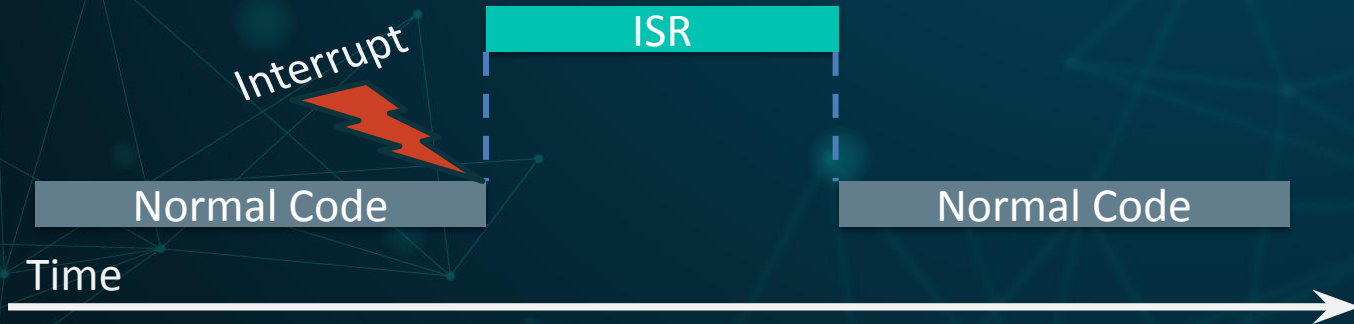
# 01



External Interrupt

# Interrupt

**Interrupt:** Is a combination of software and hardware to force the processor to stop its current activity and begin to execute a particular piece of code called an interrupt service routine (ISR). Which is a response to a specific event generated by hardware change (internally or externally) or even software.



# Exceptions Overview

---

ARM v7 Core supports multiple great features for handling exceptions and interrupts. Which includes the Nested Vectored Interrupt Controller (NVIC).

Micro-Coded Architecture So that interrupt stacking, entry, and exit are done automatically in hardware. Which offloads this work overhead from the CPU.

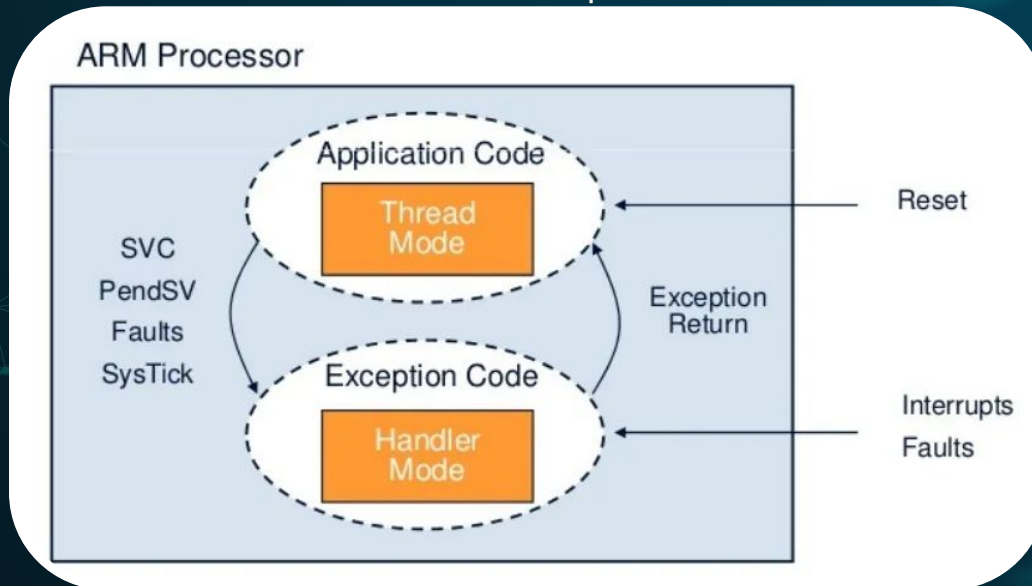
The interrupt architecture and priorities are very flexible and highly configurable to support RTOS.

# Processor Mode

The processor mode can change when exceptions occur. And it can be in one of the following modes:

Thread Mode: Which is entered on reset.

Handler Mode: Which is entered on all other exceptions.





# Micro-Coded Interrupts

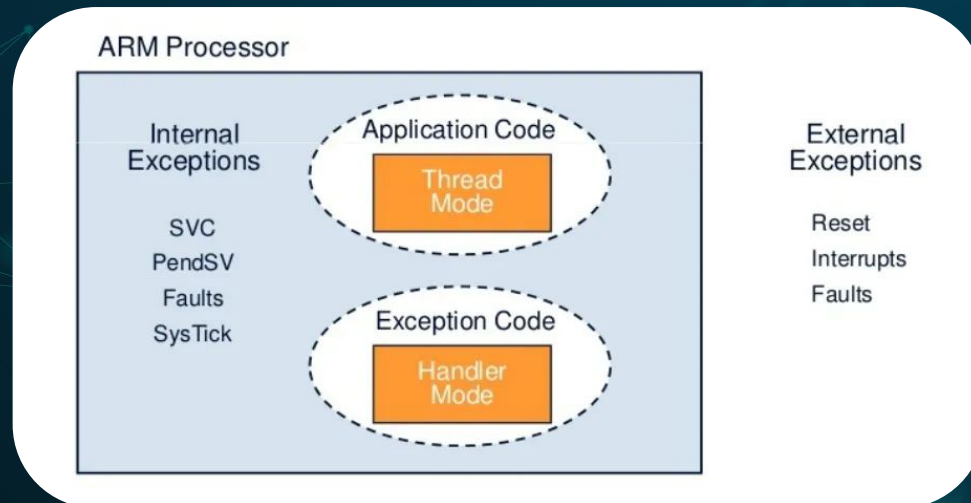
The interrupt entry and exit are hardware implemented in order to reduce the latency and speed up the response. The hardware

- Automatically saves and restores processor context
- Allows late determination of highest priority pending interrupt
- Allows another pending interrupt to be serviced without a full restore/save for processor context (this feature is called tail-chaining)

Exceptions can be fired by various events including:

Internal Exceptions

External Exceptions



# Internal Exceptions

Exceptions that get fired by an internal source to the system and not by any external hardware or peripherals. And this includes:

- **Reset:** This is the handler routine that gets executed when the processor gets out of a reset state whatever the source is.
- **System Service Call (SVC):** Similar to SVC instruction on other ARM cores. it allows non-privileged software to make system calls. This provides protection for critical system functionalities.
- **Pended System Call (PendSV):** Operates with SVC to ease RTOS development as it's intended to be an interrupt for RTOS use.
- **Non-Maskable Interrupt (NMI):** As the name suggests, this interrupt cannot be disabled. If errors happen in other exception handlers, an NMI will be triggered. Aside from the reset exception, it has the highest priority of all exceptions.



# Fault Exceptions

Some of the system exceptions are used to signal and handle specific faults. There are several categories for fault exceptions which include:

Bus Faults

Usage Faults

Memory Management Faults

Hard Faults

# Exception Servicing Model

At the reset state, all interrupts are disabled. The processor begins executing the code instructions with a base execution priority lower than the lowest programmable priority level, so any enabled interrupt can pre-empt the processor.

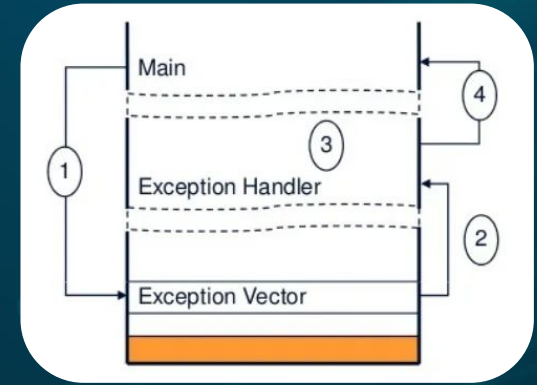
When an enabled interrupt is asserted, the interrupt is serviced by the corresponding ISR handler. The processor runs the handler at the execution priority level of the interrupt. And when the ISR is done, the original priority level is restored.

When an enabled interrupt is asserted with a lower or equal priority level, the interrupt is pended to run.

The interrupt nesting is always enabled, to disable it just set all the interrupts to the same priority level.

# Exception Behavior

1. When an exception occurs, the current instruction stream is stopped and the processor accesses the exceptions vector table.
2. The vector address of that exception is loaded from the vector table.
3. The exception handler starts to be executed in handler mode.
4. The exception handler returns back to main (assuming no further nesting).



# Reset Behavior

---

1. When a reset occurs (Reset input is asserted).
2. The MSP (main stack pointer) register loads the initial value from the address 0x00.
3. The reset handler address is loaded from address 0x04.
4. The reset handler gets executed in thread mode.
5. The reset handler branches to the main program.

# Exception States

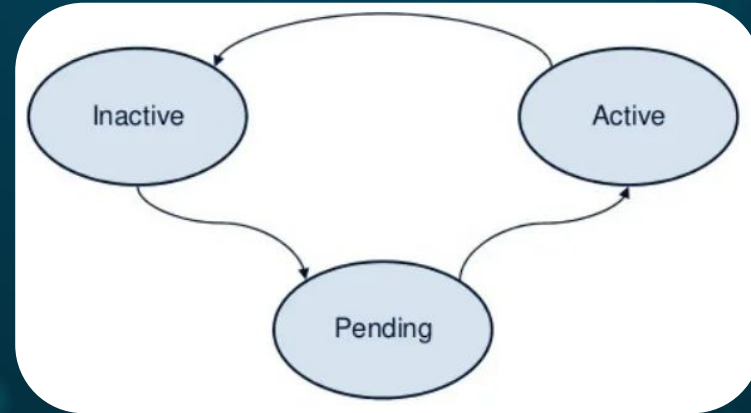
Each exception can be in one of the following states:

**Inactive:** Not pending nor active.

**Pending:** Exception event has been fired but the handler is not executed yet.

**Active:** The exception handler has started execution but it's not over yet. Interrupt nesting allows an exception to interrupt the execution of another exception's handler. In this case, both exceptions are in the active state.

**Active And Pending:** The exception is being serviced by the processor and there is a pending exception from the same source.



# Interrupts Tail-Chaining SpeedUp

# 02

---

External Interrupt

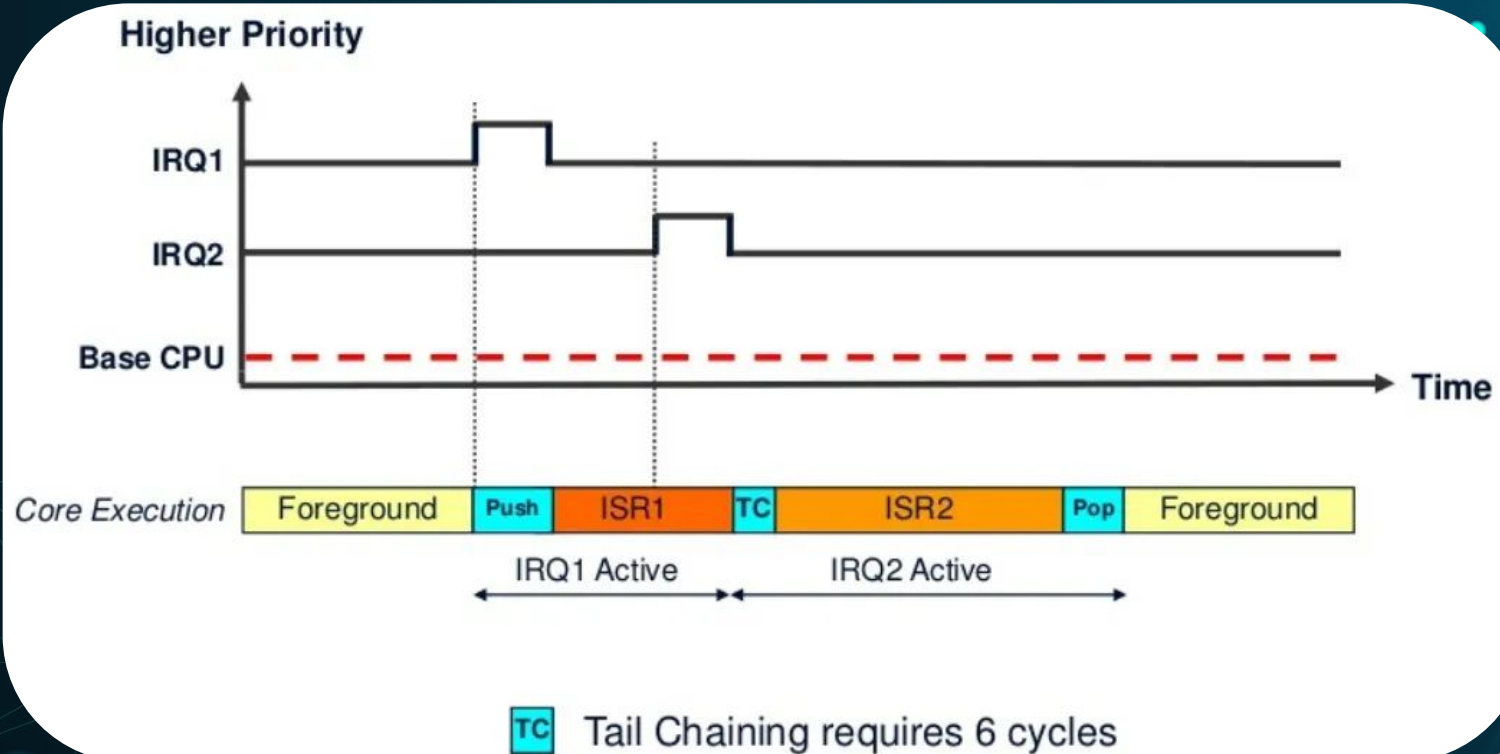


# Interrupts Tail-Chaining SpeedUp

When an interrupt (exception) is fired, the main (foreground) code context is saved (pushed) to the stack and the processor branches to the corresponding interrupt vector to start executing the ISR handler. At the end of the ISR, the context saved in the stack is popped out so the processor can resume the main (foreground) code instructions. However, and if a new exception is already pended, the context push & pop are skipped. And the processor handles the second ISR without any additional overhead. This is called "Tail-Chaining".

And it requires 6 cycles on Cortex-M3/M4 processors. Which is a huge speedup in the performance and enhanced the interrupt response time greatly (reduces the interrupt latency). Here is an example of what happens if the CPU receives a 2nd interrupt request (IRQ2) while it's servicing the 1st one (IRQ1).

# Interrupts Tail-Chaining SpeedUp



# Interrupt Late Arrival SpeedUp

# 03

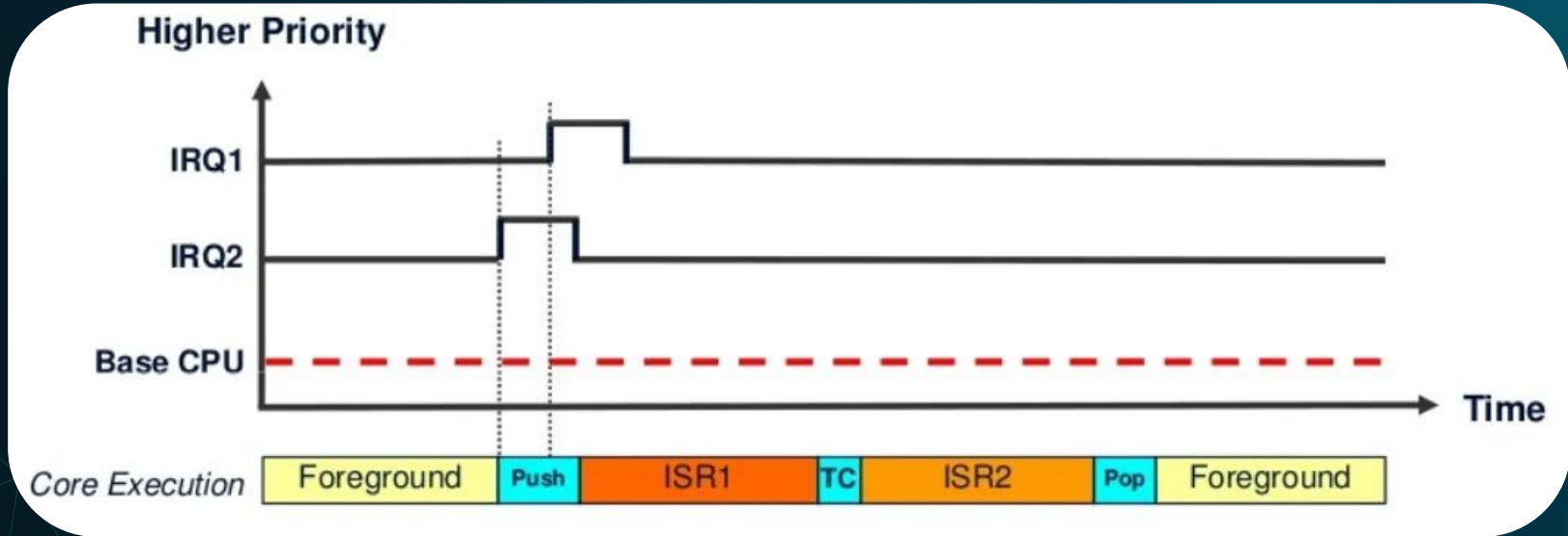
---

External Interrupt

# Interrupt Late Arrival SpeedUp

The ARM core can detect a higher priority exception while in the “exception entry phase” (stacking caller registers & fetching the ISR routine vector to be executed) of another exception. A “late arriving” interrupt is detected during this period. The higher priority ISR can be fetched and executed but the context saving that has been already done can be skipped. This reduces the latency for the higher priority interrupt and, upon completion of the late-arriving exception handler, the processor can then tail-chain into the initial exception that was going to be serviced (the lower priority one).

# Interrupts Tail-Chaining SpeedUp



A pending higher-priority exception is handled before an already pending lower-priority exception even after the exception entry sequence has started. The lower-priority exception is handled after the higher-priority exception.

# Pre-Emption 04



External Interrupt

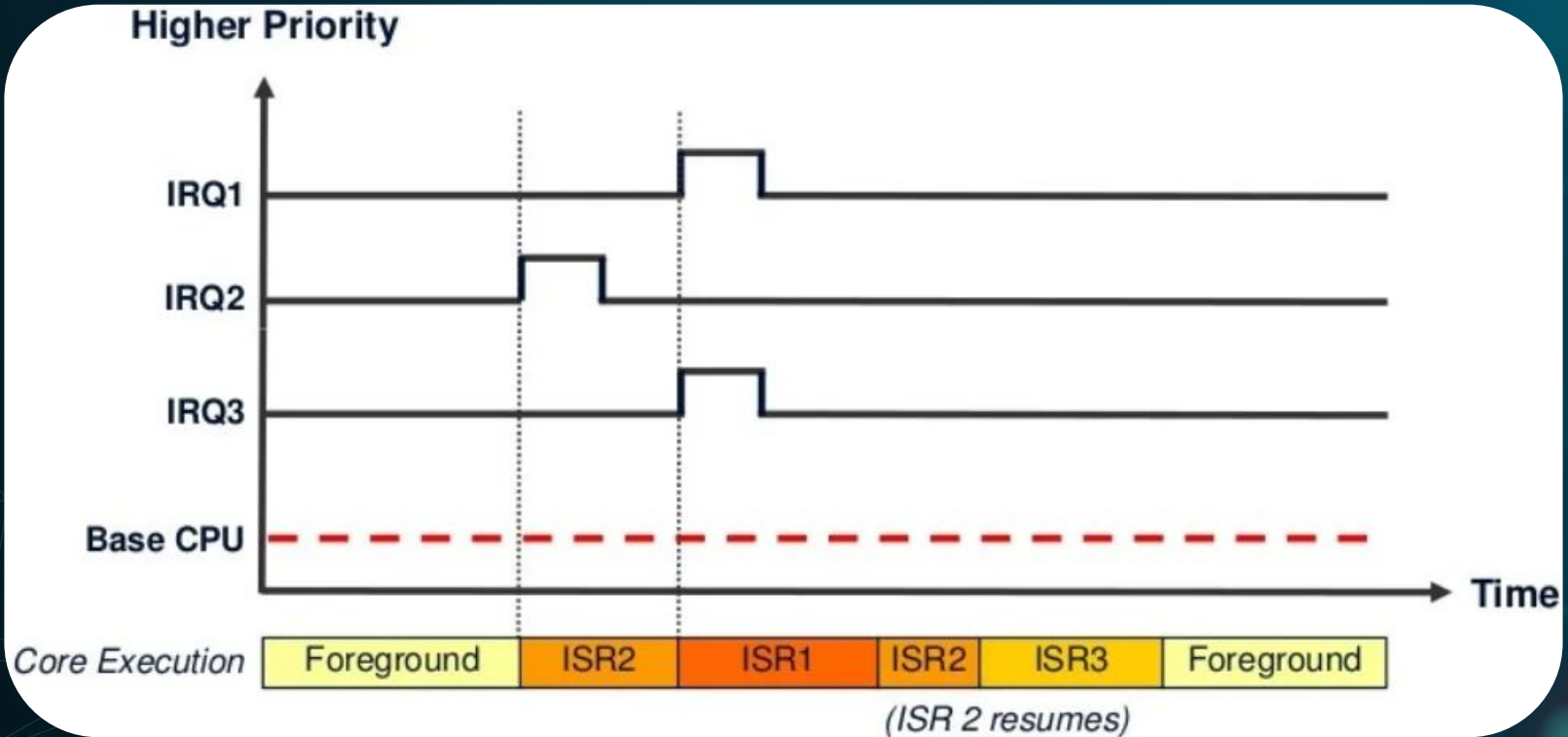


# Pre-Emption

The pre-emption happens when a task is abandoned (gets interrupted) in order to handle an exception. The currently running instruction stream is said to be pre-empted. When multiple exceptions with the same priority levels are pending, the one with the lowest exception number gets serviced first. And once an exception is active and being serviced by the processor, only exceptions with a higher priority level can pre-empt it.

Consider the following example, where 3 exceptions/interrupts are fired with different priority levels. IRQ1 pre-empted IRQ2 and forced IRQ3 to pend until IRQ1 completion. After IRQ1 ISR completion, ISR2 continues where it left off when IRQ1 pre-empted it. And finally, after ISR2 completion, ISR3 starts executions. And the context is restored to the main program (foreground).

# Pre-Emption



# Interrupt Vector Table 05

---

External Interrupt

# Interrupt Vector Table

The first entry in the table (lowest address) contains the initial MSP. All other addresses contain the vectors (addresses) to the start of exception handlers (ISRs), each address is 4-Byte wide. The table has up to 496 external interrupts which is implementation-dependent on each specific target.

The interrupt vector table may be relocated in the memory easily by changing the value of the vector table offset register. The interrupt/exception vector table is usually located in the startup code file. And it looks something like this down below.

Address		Exception #
$0x40 + 4*N$	External N	$16 + N$
...	...	...
0x40	External 0	16
0x3C	SysTick	15
0x38	PendSV	14
0x34	Reserved	13
0x30	Debug Monitor	12
0x2C	SVC	11
0x1C to 0x28	Reserved (x4)	7-10
0x18	Usage Fault	6
0x14	Bus Fault	5
0x10	Mem Manage Fault	4
0x0C	Hard Fault	3
0x08	NMI	2
0x04	Reset	1
0x00	Initial Main SP	N/A

# Interrupt Vector Table

```

1 ; Vector Table Mapped to Address 0 at Reset
2
3         AREA     RESET, DATA, READONLY
4         EXPORT  __Vectors
5         EXPORT  __Vectors_End
6         EXPORT  __Vectors_Size
7
8 __Vectors DCD     __initial_sp                ; Top of Stack
9           DCD     Reset_Handler              ; Reset Handler
10          DCD     NMI_Handler                 ; -14 NMI Handler
11          DCD     HardFault_Handler           ; -13 Hard Fault Handler
12          DCD     MemManage_Handler           ; -12 MPU Fault Handler
13          DCD     BusFault_Handler            ; -11 Bus Fault Handler
14          DCD     UsageFault_Handler          ; -10 Usage Fault Handler
15          DCD     0                           ; Reserved
16          DCD     0                           ; Reserved
17          DCD     0                           ; Reserved
18          DCD     0                           ; Reserved
19          DCD     SVC_Handler                 ; -5 SVCall Handler
20          DCD     DebugMon_Handler            ; -4 Debug Monitor Handler
21          DCD     0                           ; Reserved
22          DCD     PendSV_Handler              ; -2 PendSV Handler
23          DCD     SysTick_Handler             ; -1 SysTick Handler
24
25         ; Interrupts
26 ; ToDo: Add here the vectors for the device specific external interrupts handler
27          DCD     Interrupt0_Handler           ; 0 Interrupt 0
28          DCD     Interrupt1_Handler           ; 1 Interrupt 1
29          DCD     Interrupt2_Handler           ; 2 Interrupt 2
30          DCD     Interrupt3_Handler           ; 3 Interrupt 3
31          DCD     Interrupt4_Handler           ; 4 Interrupt 4
32          DCD     Interrupt5_Handler           ; 5 Interrupt 5
33          DCD     Interrupt6_Handler           ; 6 Interrupt 6
34          DCD     Interrupt7_Handler           ; 7 Interrupt 7
35          DCD     Interrupt8_Handler           ; 8 Interrupt 8
36          DCD     Interrupt9_Handler           ; 9 Interrupt 9
37
38          SPACE   (214 * 4)                   ; Interrupts 10 .. 224 are left out
39 __Vectors_End
40 __Vectors_Size EQU  __Vectors_End - __Vectors

```



# Interrupt Vector Table

The interrupt vector table for the STM32 ARM microcontrollers we're using in this course can be found in the corresponding datasheets of these devices. STM32F103C8 And STM32L432KC, it'll be as shown in the diagram below. It's only one page of it only for reference, the full table is found in the datasheet itself.

Vector table for other STM32F10xxx devices					
Position	Priority	Type of priority	Acronym	Description	Address
-	-	-	-	Reserved	0x0000_0000
-	-3	fixed	Reset	Reset	0x0000_0004
-	-2	fixed	NMI	Non maskable interrupt. The RCC Clock Security System (CSS) is linked to the NMI vector.	0x0000_0008
-	-1	fixed	HardFault	All class of fault	0x0000_000C
-	0	settable	MemManage	Memory management	0x0000_0010
-	1	settable	BusFault	Prefetch fault, memory access fault	0x0000_0014
-	2	settable	UsageFault	Undefined instruction or illegal state	0x0000_0018
-	-	-	-	Reserved	0x0000_001C - 0x0000_002B
-	3	settable	SVCall	System service call via SWI instruction	0x0000_002C
-	4	settable	Debug Monitor	Debug Monitor	0x0000_0030
-	-	-	-	Reserved	0x0000_0034
-	5	settable	PendSV	Pendable request for system service	0x0000_0038
-	6	settable	SysTick	System tick timer	0x0000_003C
0	7	settable	WWDG	Window watchdog interrupt	0x0000_0040
1	8	settable	PVD	PVD through EXTI Line detection interrupt	0x0000_0044
2	9	settable	TAMPER	Tamper interrupt	0x0000_0048
3	10	settable	RTC	RTC global interrupt	0x0000_004C
4	11	settable	FLASH	Flash global interrupt	0x0000_0050
5	12	settable	RCC	RCC global interrupt	0x0000_0054
6	13	settable	EXTI0	EXTI Line0 interrupt	0x0000_0058
7	14	settable	EXTI1	EXTI Line1 interrupt	0x0000_005C
8	15	settable	EXTI2	EXTI Line2 interrupt	0x0000_0060
9	16	settable	EXTI3	EXTI Line3 interrupt	0x0000_0064
10	17	settable	EXTI4	EXTI Line4 interrupt	0x0000_0068
11	18	settable	DMA1_Channel1	DMA1 Channel1 global interrupt	0x0000_006C
12	19	settable	DMA1_Channel2	DMA1 Channel2 global interrupt	0x0000_0070
13	20	settable	DMA1_Channel3	DMA1 Channel3 global interrupt	0x0000_0074



# Interrupt Stacking, ISR, Return

# 06

---

External Interrupt

# Interrupt Stacking (Context Saving)

The main program stack is used to store the program state before an interrupt is received. The stack pointer (SP) is automatically decremented on push operation, and it always points to a non-empty value. The stack is 8-Byte aligned and padding may be inserted if it's required.

When receiving an interrupt signal:

# Interrupt Stacking (Context Saving)

- The processor will finish the current instruction as long as it's not a multi-cycle instruction
- Multi-Cycle instructions may be abandoned by the processor to handle the exception/interrupt then it restarts the abandoned multi-cycle instruction. Alternatively, the processor can stop the multi-cycle instruction and go to handle the exception, then come back to continue it. Multi-cycle instructions include: divide, multiply, load/store double, etc.
- The processor state (context) is automatically saved to the stack. Eight registers are pushed (PC, R0-R3, R12, LR, xPSR).
- During or after context saving, the address of the corresponding ISR is loaded from the exception/interrupt vector table.
- The link register is modified for return after interrupt.
- The first instruction of the ISR starts to be executed by the CPU. For Cortex-M3/M4, the whole latency this process takes is 12 cycles. However, IRQ latency is improved if late-arrival or tail-chaining has occurred.

# Interrupt Service Routine (ISR) Handling

The ISR handler should clear the interrupt source if it's required (Some don't need to be cleared like the SysTick).

Interrupt nesting won't affect the way the ISR is written however, attention should be paid to the main stack overflow that may occur.

The ISR C-Code should be written in a clear way, that's easily readable and easily executed by the processor. Given that certain exceptions/interrupts are to be serviced hundreds or thousands of times per second. So it must run so quickly and no delays are permitted within ISR handlers unless it's a few microseconds and there is a strong reasoning and justification behind it.

# Return From ISR (Context Restoration)

When an exception (ISR) handler is completely executed and no other interrupts are pending, the CPU restores the context of the main (foreground) application code.

The EXC\_RETURN instruction is fetched and gets executed to restore the PC and pop the CPU registers.

If other interrupts are pending, the highest priority will be serviced first, and the context restoration is abandoned to accelerate the interrupt response. This is the tail-chaining feature discussed earlier.

If the context restoration process is interrupted, it gets abandoned. And the new ISR starts execution without the need to save the context because it's already pushed into the stack. The return from interrupt (context restoration) on ARM Cortex-M3/M4 requires 10 clock cycles.

# Exceptions Priorities Overview

# 07

---

External Interrupt




# Exceptions Priorities Overview

The base system execution priority level is lower than the lowest programmable priority level. So any enabled interrupt when gets fired, it'll pre-empt the main code execution. Afterward, the corresponding ISR will get executed.

There exist a few ways in software to change the main code execution priority level to make it higher than the default priority of thread mode or the exception that is currently active. This process is called Priority Boosting. And it can be advantageous in many situations, especially in RTOS. When you need to execute some logic without getting interrupted by any source.

# Exceptions Priorities Overview

Internal and external exceptions table is found in the datasheet. Some exceptions/interrupts are at a fixed priority level and can't be changed programmatically. And the lower the priority level number, the higher the priority is. The priority level is stored in a byte-wide register which is cleared (0x00) on reset. If two or more exceptions/interrupts are of the same priority level value, the priority order is therefore determined based on the exception number itself (Lower exception number has a higher priority).

Name	Exception Number	Exception Priority No.	Lowest
Interrupts #0 - #495 (N interrupts)	16 to 16 + N	0-255 (programmable)	
SysTick	15	0-255 (programmable)	
PendSV	14	0-255 (programmable)	
SVCall	11	0-255 (programmable)	
Usage Fault	6	0-255 (programmable)	
Bus Fault	5	0-255 (programmable)	
Memory Management Fault	4	0-255 (programmable)	
Hard Fault	3	-1	
Non Maskable Interrupt (NMI)	2	-2	
Reset	1	-3	
			Highest

# Interrupts Priority & Priority Grouping

# 08

---

External Interrupt

# Interrupts Priority & Priority Grouping

Each exception/interrupt has associated an 8-bit priority level register. But not all bits are used to set priorities. STM32F103C8 MCU has only 16 priority levels which means that 4 MSB bits are used to set priorities. If needed these bits can be split into two groups where you can create sub-priority levels for each preemptive priority. Sub-priority is used only if the group priority is the same.

Sub-priority levels are useful when two or more same priority level interrupts occur. Then, the one with a higher sub-priority will be handled first. And if two exceptions/interrupts are of the same priority levels exactly, the one with lower vector number gets handled first.

Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
Priority		Sub-priority		Reserved			

# EXTI (External Interrupts) Controller

# 09

---

External Interrupt

# EXTI (External Interrupts) Controller

The external interrupt/event controller consists of up to 20 edge detectors in connectivity line devices, or 19 edge detectors in other devices for generating event/interrupt requests. Each input line can be independently configured to select the type (event or interrupt) and the corresponding trigger event (rising or falling or both). Each line can also be masked independently. A pending register maintains the status line of the interrupt requests.



# EXTI Features

---

The EXTI controller main features are the following:

Independent trigger and mask on each interrupt/event line

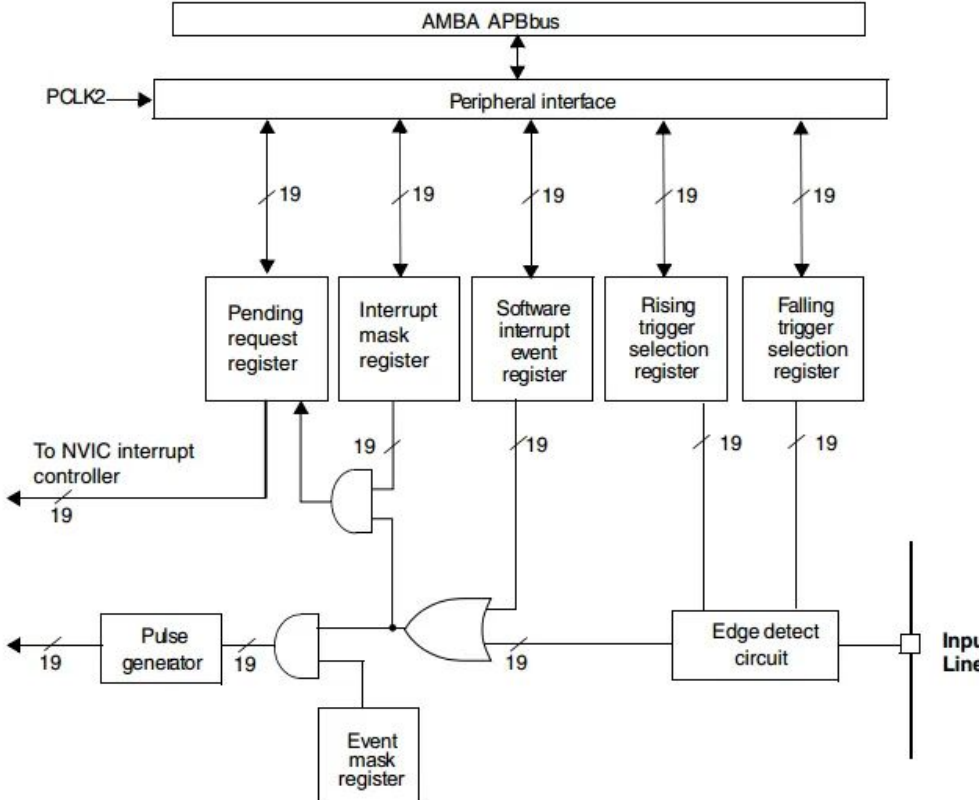
Dedicated status bit for each interrupt line

Generation of up to 20 software event/interrupt requests

Detection of external signal with pulse width lower than the APB2 clock period.

100

### External interrupt/event controller block diagram



# Functional Description

---

## Functional Description

To generate the interrupt, the interrupt line should be configured and enabled. This is done by programming the two trigger registers with the desired edge detection and by enabling the interrupt request by writing a '1' to the corresponding bit in the interrupt mask register. When the selected edge occurs on the external interrupt line, an interrupt request is generated. The pending bit corresponding to the interrupt line is also set. This request is reset by writing a '1' in the pending register.

# Hardware interrupt selection

## Hardware interrupt selection

To configure the 20 lines as interrupt sources, use the following procedure:

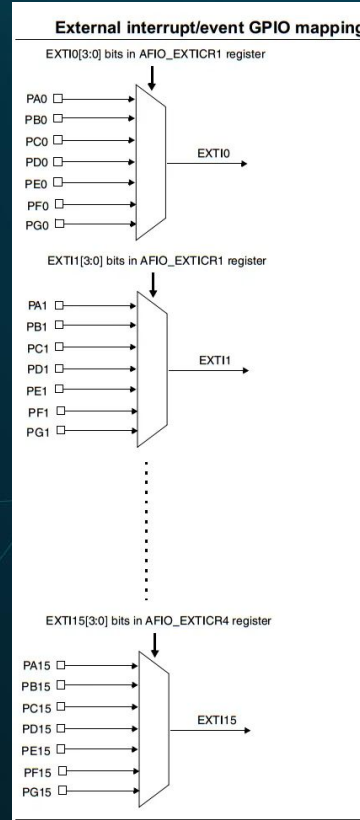
Configure the mask bits of the 20 Interrupt lines (EXTI\_IMR)

Configure the Trigger Selection bits of the Interrupt lines (EXTI\_RTSR and EXTI\_FTSR)

Configure the enable and mask bits that control the NVIC IRQ channel mapped to the External Interrupt Controller (EXTI) so that an interrupt coming from one of the 20 lines can be correctly acknowledged.

# EXTI External Interrupts GPIO Mapping

GPIOs are connected to the 16 external interrupt/event lines in the following manner:



# EXTI Lines

---

The four other EXTI lines are connected as follows:

EXTI line 16 is connected to the PVD output

EXTI line 17 is connected to the RTC Alarm event

EXTI line 18 is connected to the USB Wakeup event

EXTI line 19 is connected to the Ethernet Wakeup event

We'll see how to configure the external interrupt pins using the CubeMX software tool in the next tutorial which is going to be a practical LAB for the external interrupts.





**STM32**  
**Is AWESOME**



# Session LAb





# THANKS!

Do you have any questions?

**[www.imtschool.com](http://www.imtschool.com)**

**[www.facebook.com/imaketechnologyschool/](https://www.facebook.com/imaketechnologyschool/)**

*This material is developed by IMTSchool for educational use only*

*All copyrights are reserved*