Interrupt

→ The interrupt is an event or exception typically generated by hardware or software that requires the cpu to stop executing the code it is currently running and perform some service (code) related to the event.

* interrupt (event) types:.
  1- Asynchronous ⟶ Hardware Interrupt
  2- synchronous ⟶ exceptions

1- Asynchronous (Hardware Interrupt)
→ raised by a hardware device (internal or external) to the MCU
→ these events: generated at arbitrary [random] times with respect to the cpu clock signals

2- Synchronous (exceptions):
→ produced by the cpu control unit while executing instructions, as it detects an error or exception in the current instruction.
→ the processor issues them only after terminating the execution of an instruction.

* Hardware interrupt:
  • internal Hardware:
    → the [EUSART] communication protocol module received a new byte of data
    → the [Timer] module complete the required time
    → the (ADC) module finishes the ADC conversion.
  • external Hardware:
    → The [INTX] pins → (INT0, 1, 2) Raising / falling edge detection
    → The on change [KBI0, KBI1, KBI2, KBI3] pins: interrupt on every voltage change on these pins

* Hardware interrupt procedures :
  - → the device asserts (issues) interrupt request to the CPU.
  - → the processor suspends the currently executing task
  - → the processor executes an Interrupt service routine (ISR) to service the device
  - → the processor resumes the previously suspended task.

* EXCeptions are divided into three group :-
  1) Faults 2) traps 3) Aborts
  - → Exceptions → generated when the CPU detects an anomalous (odd) condition while executing an instruction

→ When an exception occurs, the processor writes a specific value in a certain register in memory ~~[illegible]~~. Based on this value the exception is classified as either Faults, traps or aborts. this value is the address of the instruction that caused the exception to occur.

[1] Faults :
- → they are generally correctable, and the program can be restarted with no loss of continuity
- → when the exception handler (ISR) is able to correct the anomalous condition that caused the exception, the instruction that caused the fault can be resumed.
- → common examples of faults include
  - divide error → division by 0
  - invalid OP-code → CPU not supports this OP code
  - floating-point error
  - Alignment check
- → the saved value in the register is the address of the instruction that caused the fault

2 Traps :.

→ the saved value in the register is the address of the instruction that should be executed after the one that caused the trap
→ the main use of traps is for depugging purposes
→ Interrupt signal → notify the debugger that a specific instruction has been executed
→ the debugger → provides data for examination by the user
→ the user → may ask that execution of the debugged program resume, starting from the next instruction

* example:
- Breakpoint - overflow

3 Aborts :-

→ A serious error occurred, such as a hardware failure
→ the control unit may be unable to store in the register the exact location of the instruction causing the exception
→ interrupt signal → sent by the control unit → is an emergency signal used to switch control to the corresponding abort exception handler.

* example:
- Double fault - machine-check exception

* Software interrupt :-

→ you can trigger exceptions or interrupts using software code.
• executing this code causes an interrupt

* Interrupt service routine (ISR / ESR)

→ is a software routine (function) that the processor invokes in response to an interrupt OR exception

→ any interrupt or exception must own its routine
→ when an interrupt or exception happened, the cpu will jump / execute the ISR / ESR

→ an ISR must perform very quickly to avoid slowing down the operation of the device.

→ the address of ISR / ESR determined while compile the application and stored in a fixed location in the flash memory (ROM)

- the address of the ISR / ESR called a "vector"
- All vectors are grouped in a table called "vector table"
- the vector table location is known and may be fixed or can be mapped
- ISR functions have no arguments passed into them and they can never return a value.

---

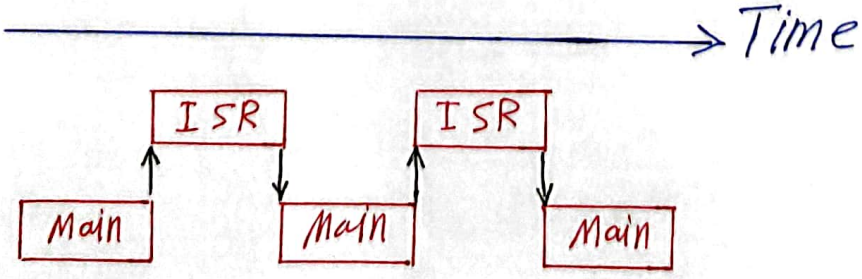
* executing (handling) an interrupt request.

1- the CPU finishes the instruction it is currently executing and saves the address of the next instruction (program counter) on the stack

2- it jumps to a fixed location in memory "vector table" which directs the CPU to the address of the interrupt service routine (ISR)

3- the CPU starts to execute the (ISR) untill it reaches the last Instruction of the routine, which is RETI (return from interrupt)

4- upon executing the [RETI] instruction, the CPU returns to the place where it was interrupted. first, it gets the PC address from the stack by popping the top bytes of the stack into the PC, then it starts to execute from that address



* Interrupt ~~Masking~~ Masking and Interrupt pending :

→ two types of interrupts masking :

1- Maskable interrupt 2- Non-Maskable interrupt

1) Maskable interrupt :-

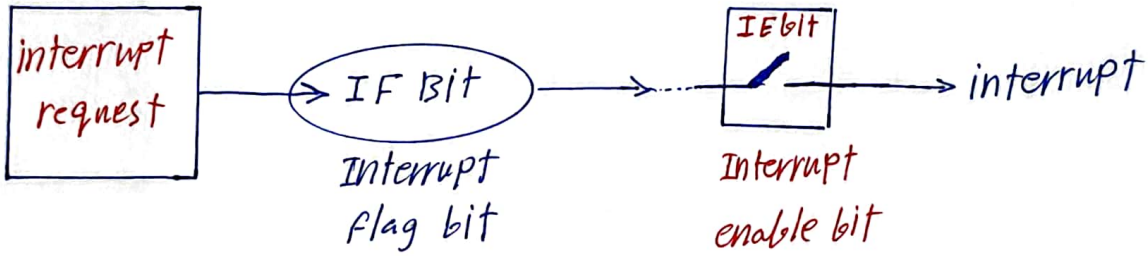
- This type of interrupts can be (enabled or disabled)
- each interrupt has a "flag bit" to indicate that the event occurred.
- each interrupt has a "enable bit" to enable or disable the interrupt.

→ flag bit raised "enabled" ⟶ = 1

→ enable bit raised "enabled" ⟶ = 1

if flag bit = 1 and enable bit = 0 ⟶ the CPU was not interrupted

if flag bit = 1 and enable bit = 1 ⟶ the CPU was interrupted



2) Non-Maskable interrupts :

- this kind of interrupts can't be disabled "must be served"
- each interrupt has a (flag bit) to indicate that the event occured
- example : → reset

* Interrupt pending :

* interrupt handlers (routines) : have priorities associated with interrupts, this priority determines which interrupt will be handled first if multiple interrupts occur at the same time.
* if multiple interrupts occur at the same time, the interrupts will be handled in priority order - highest to lowest - the lower priority interrupts will remain in a "pending state" untill higher priority interrupts have been serviced
* interrupt priority : is set based on the importance and urgency of the interrupt.

* Polling and interrupt:

→ a micro controller can serve multiple devices using interrupts or polling.
→ in the interrupt method, devices notify the micro controller when they need service and the micro controller stops its current task to serve the device based on the priority.
→ in polling, the micro controller continuously checks the status of devices and serves them when a condition is met
* interrupt is a hard ware mechanism / polling is a protocol
* interrupts are more preferable as they allow the microcontroller to perform other tasks while waiting for devices requests and can also ignore devices requests. this saves time and allows for more efficient use of the microcontroller.

* Polling:

Advantage → efficient if the "events are rapidly → High rate.
disadvantage → takes CPU time even when no "pending request".

* interrupt priority:

→ if the microcontroller support multiple interrupt sources, then the CPU has to decide which interrupt should receive service first in this situation.
→ the solution is to prioritize all interrupt sources based on their priority levels.
→ the priority level of an interrupt is typically determined by its importance to the system
→ an interrupt with higher priority always receives service befor interrupts at lower priorityes.

* microcontrollers interrupt prioritization can be:

→ fixed by the hardware and can't change it by software → Traps
→ changed by the software (pre-configurations and run-time)

* priority levels:

PIC 18F → 1-bit → (0/1) → low or high priority
PIC 24F → 3-bit → (0:7) → 0 < 1 < 2 .... < 7

* interrupt Controller: (PIC : Programmable Interrupt Controller) or (NVIC : Nested Vector Interrupt controller)
→ it controls interrupt behavior, such as enabling or disabling interrupts, setting the interrupt priority, and selecting the interrupt source.

* What if 2 interrupts with the same priority happens at the same time?
→ they are handled using one of two methods:
1- Round-robin: the interrupts are handled on an alternating basis. the cpu will handle one interrupt for a few cycle, then switch to the other interrupt, and so on. this ensure that both interrupts are serviced in a timely manner (توقيت مناسب)

2- priority queuing or First come, First served: whichever interrupt occurred first in time will be handled first, even if by just a few nano seconds. the second interrupt will have to wait its turn.

→ in some cases, the interrupts may also be masked temporarily to ensure the current interrupt is fully handled before allowing another interrupt of the same priority
→ the round-robin approach is often considered "fairer" since it ensures both interrupts get serviced within a reasonable time frame.

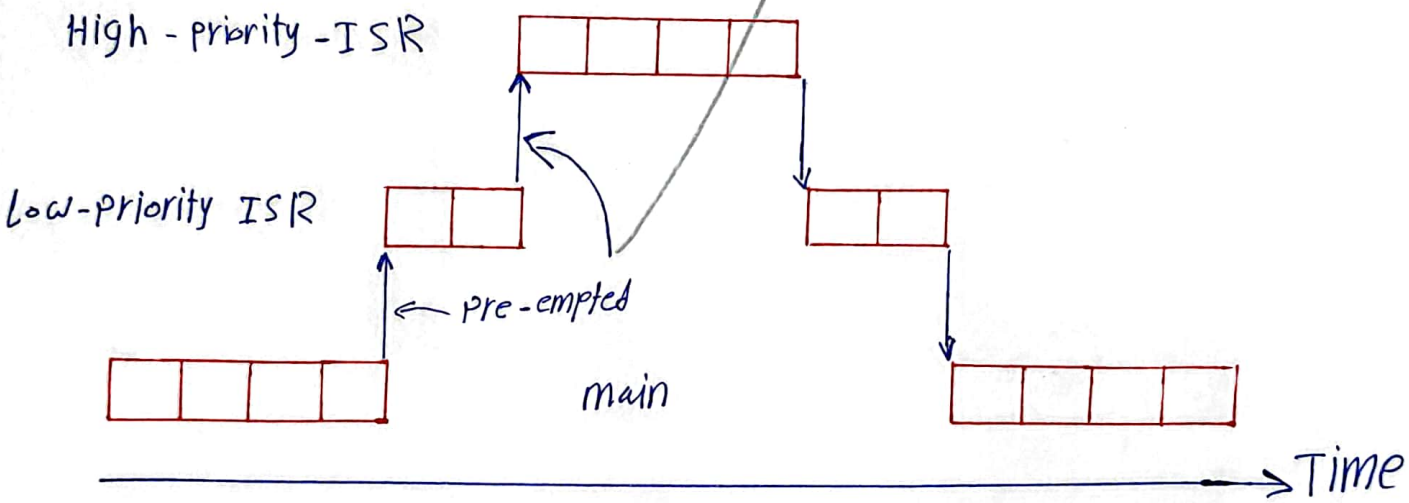
* Interrupt Nesting:-
→ refers to the situation where an interrupt occurs while the processor is already servicing another interrupt

* What if a high priority interrupt happened while executing a low priority interrupt??
→ the processor will save the context of the low-priority interrupt service routine and immediately start executing the high priority interrupt service routine.

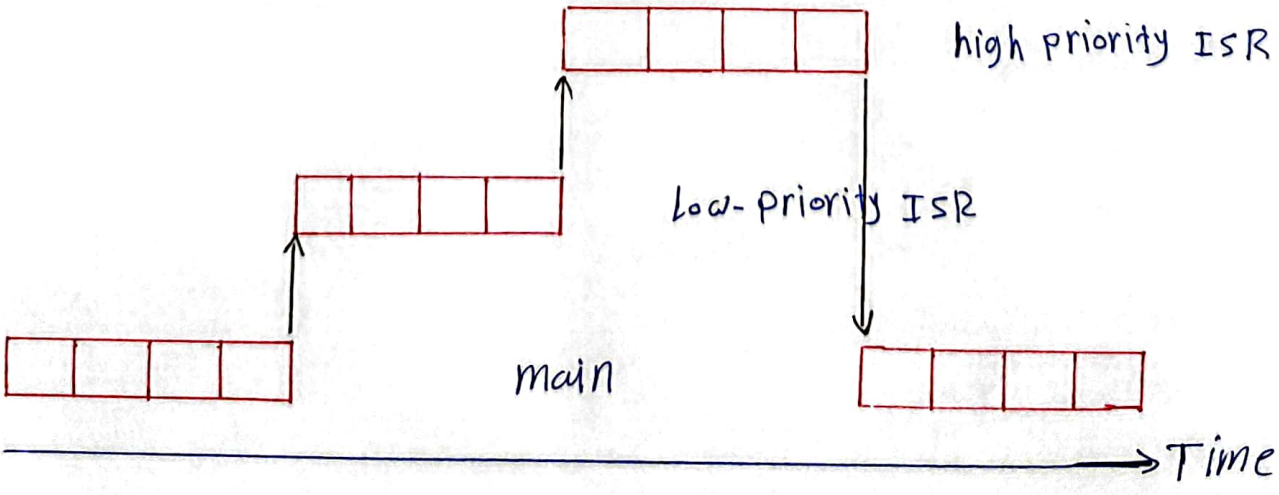
→ once the high-priority interrupt is serviced, the processor will return to the low-priority interrupt service routine, and continue from where it left off this process is known as "interrupt preemption"

* if the low-priority interrupt occurs while a high priority interrupt is being serviced, the low-priority interrupt is masked or preempted until the high priority interrupt handler finishes. this is a common approach
* interrupts, by default, are nestable ::

* Nested interrupt schedule :



* in the Non-Nested interrupt, if the high priority interrupt occurs while a low-priority interrupt is being serviced, the high-priority interrupt is pending until the low-priority interrupt is completed



Interrupt handling

* Vectored Arbitration system:-
  1- interrupt vector table (IVT) ⟶ Vectored interrupt
  2- non vectored interrupt ⟶ (Non IVT)

1 Vectored interrupt:

→ is a type of interrupt handling mechanism, when an interrupt occurs, the interrupt controller (NVIC) identifies the source of the interrupt and directs the processor to the corresponding interrupt handler (ISR) using a vector address which is a pointer to the memory location where the ISR is stored, and each device that can generate interrupts has its own unique vector address.

→ this allows the processor to quickly and efficiently handle the interrupt without having to search through a table or list of interrupt sources.

→ IVT (interrupt vector table): is a table of addresses that the
- processor uses to locate the appropriate interrupt handler (ISR) when an interrupt occurs.
- When an interrupt is triggered, the interrupt controller looks up the corresponding entry in the IVT to find the address of the ISR associated with that interrupt, the processor then jumps to that address to execute the ISR
- the IVT is typically a fixed-size table located in a specific memory address in flash memory

ex:

vector No.	program address	source	interrupt definition
1	0X 00	RESET	.....
2	0X 02	INT0	.....
3	0X 04	Timer0...	.....

* the address of ISR ⟶ vector table ⟶ flash memory
* The ISR definition code ⟶ • Code area ⟶ // //

* if the microcontroller support 30 different interrupt sources, and address size is 2 byte ⟶ IVT size = 30 * 2 = 60 bytes (Allocated to the flash memory)
→ IVT allocated to the lowest Address in the flash memory (. vector section)
→ the IVT also determines the priority levels of the different interrupts.
→ the lower address the higher is the priority level
→ the RESET has the highest priority

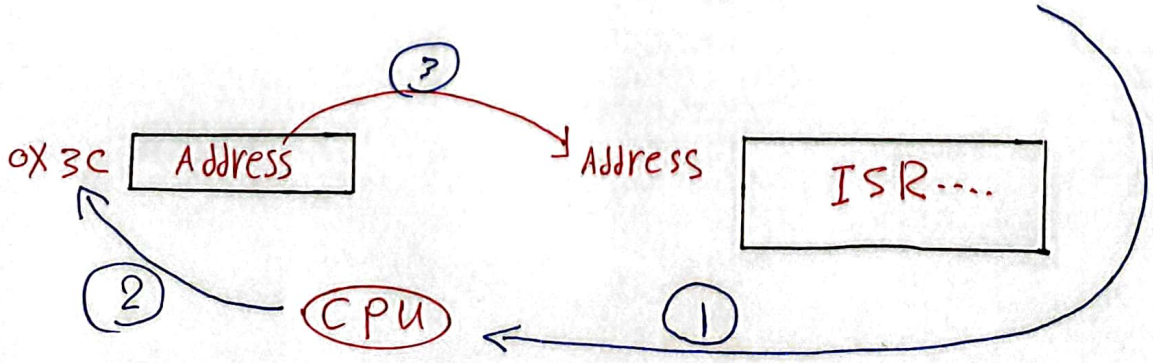
* IRQ (interrupt request line): is a signal sent by an peripheral device to the processor to request service.

* Vector table is an array of function pointers written in the "start up file/code".

* IVT in STM32F4 (cortexM4):
→ ISR vector address calculations:
Vector address = 64 + (address size * N)
64 ⟶ location of into ⟶ 0x40 in hex
N ⟶ IRQ number ⟶ from datasheet
address size ⟶ address bus width.

ex: A request happens on ⟶ IRQ=0
Vector address = 64 + (4*0) = 64 ⟶ 0x40
ex: A request happens on ⟶ IRQ = 9
Vector address = 64 + (4*9) = 100 ⟶ 0X64

ex: A request happens on ⟶ IRQ = -1
Vector address = 64 + (4 * (-1)) = 60 ⟶ 0X3C



* Start up Code:- responsible for initializing the microcontroller's hardware and software environment. it is usually written in assembly language and it is located in flash memory

→ the main tasks performed by the startup code:

1- set the initial stack pointer value = _estack → end address of stack
2- set the initial program counter value = address of reset handler function which is the first function that is executed after a reset.
PC == Reset_Handler
3- setup the vector table entries with the exceptions ISR address that the microcontroller supports. this is necessary for handling interrupts and exceptions

4- set up the microcontroller to start executing the main function in the C library, which in turn calls the user application's main function.

* RESET_HANDLER : is the first function that is executed after a reset of the microcontroller

the main tasks performed by Reset_Handler:

1- set up the stack pointer to the correct location in memory, which is the address of the current stack frame (top of the stack)
2- • the data segment contains initialized global and static variables that are stored in flash memory
• the Reset_Handler copies these variables to SRAM so that they can be accessed and modified during program execution.
3- • the bss segment contains unintialized global and static variables that are allocated but not initialized
• the Reset_Handler sets all the bytes in the bss segment to zero to ensure that these variables are intialized to zero.
4- Call the clock system initialization function, which configures the microcontroller's clock source, and sets up the system clock to a specific frequency, this is necessary for synchronizing the microcontroller with other devices and for calculating time intervals.

5- Pass control to the application to start executing the user's program

---

Non-Vectored interrupt system (No IVT)

→ in this system, there is no interrupt vector table (IVT). when an ~~interrupt~~ interrupt occurs, the PC jumps to a specific address called the interrupt vector ⟹ (specific vector)

→ At least 3 interrupt vectors need to be defined:

1- Reset Vector → Reset address → initialize the sys and start the progra
2- High priority vector
3- Low priority vector
] → used to handle interrupts

→ at each of these vectors, there is a jump instruction that jumps to a specific (ISR)

→ in this system, there are only 2 ISRs:

1- High priority ISR : to handle high priority interrupts
2- Low priority ISR : to handle Low priority interrupts.

→ when an interrupt occurs, the PC jumps to a specific vector, which points to the location of the (ISR)

→ at the (ISR), the system needs to check all the possible interrupts sequentially to determine which one caused the interrupt. this can be a slow process, especially if there are many possible interrupts that need to be checked

→ there can be a large delay between the time the interrupt occurs and the time it is serviced.

→ this delay can occur because the system needs to check all the possible interrupts sequentially before it can determine which one caused the interrupt and then handle it

→ overall, the non-IVT system can be slow and inefficient when handling interrupts compared to an IVT system, where the interrupt handling process is faster and more simplify

---

* steps to service an interrupt in more details :.

1- An exception occured (sync or Async event)
2- The interrupt detected by the processor
3- The current instruction being executed is completed by the processor before responding to the interrupt and suspend the main program execution this ensures that the main program's execution is not interrupted in the middle of an instruction
4- the processor prepares to transfer control to the interrupt routine by saving information needed to resume the current program at the point of interrupt. this process is called a "context switch"
   * to perform a context switch process :-
      1- saving the status of the processor, which is contained in a register called the program status word / register (PSW / PSR) this information is copied to shadow register
      2 - saving the location of the next instruction to be executed which is called "Return Address", this is typically done by pushing the return address onto the stack
5- in a single interrupt sys (Low / high), two situations can occur:
   1- if interrupt priority is disabled, the global interrupt enable will be disabled to prevent any interrupt from occuring while responding to the current interrupt
   2- if interrupt priority is enabled, the generated interrupt's priority is checked. If it is a high priority interrupt, the general high priority interrupt will be disabled. if it is a low priority interrupt, the general low priority interrupt will be disabled.
6- the program counter is set to the vector address specific to that interrupt. here, there are two interrupt schemas : vectored and nonvectored
   • An ISR is placed in the vector location of an interrupt to handle it.
7- execute the ISR of this interrupt until reaching a RETI "return from interrupt instruction"
8- the "PSW" or "PSR" is restored from the shadow register, and the return address is restored from the stack or from the internal backup register and loaded into the program counter.

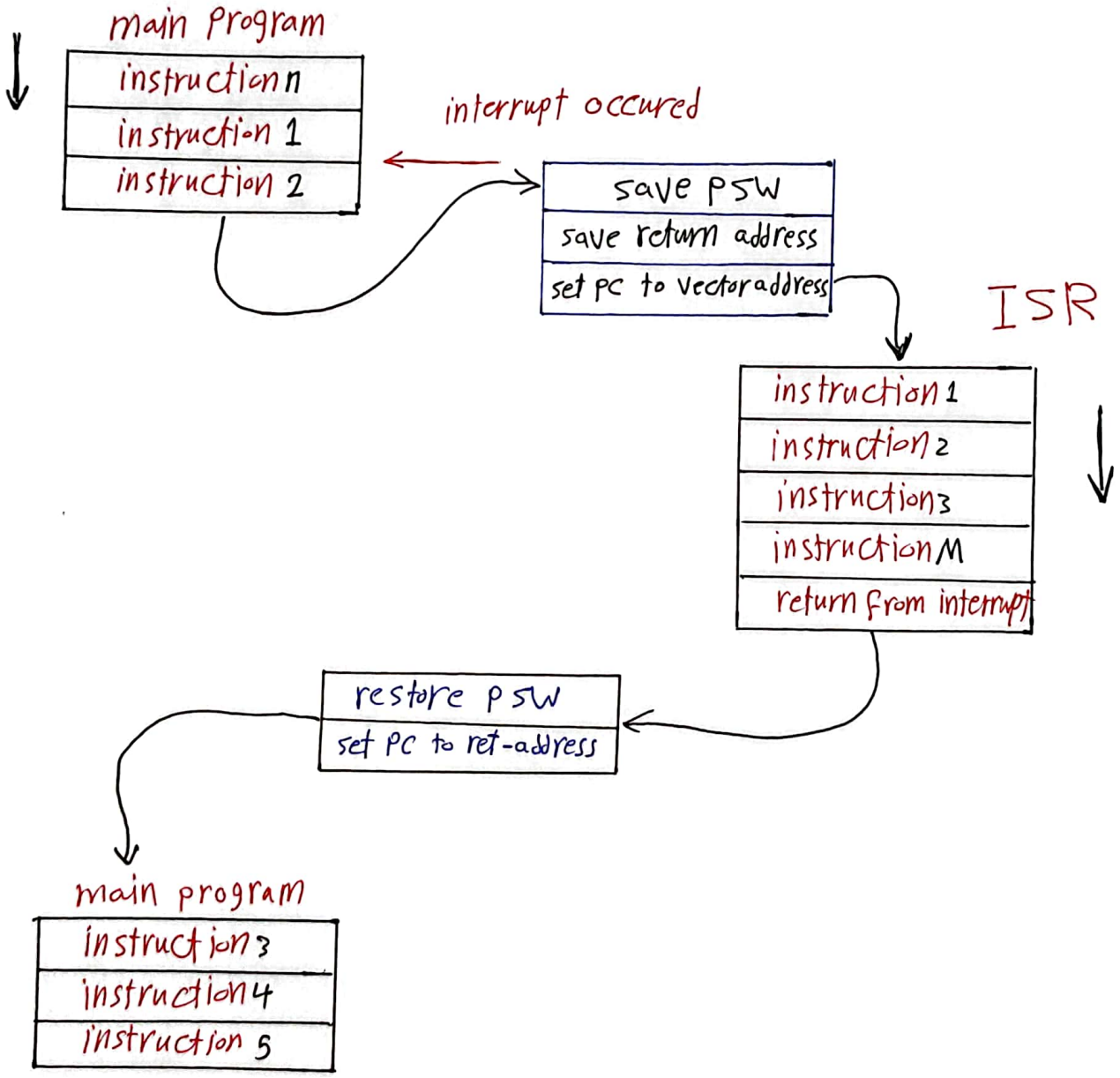
9- in single interrupt system (low/high), we have two situation.

1- interrupt priority is disabled : the global interrupt enable will be enabled back

2- interrupt priority is enabled :

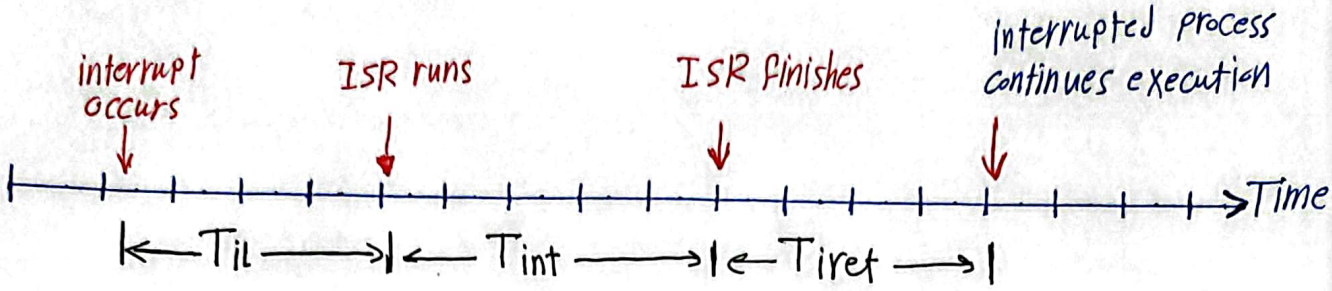
- if the generated interrupt is a high priority interrupt, so the general high priority interrupt will be enabled back
- if the generated interrupt is a low priority interrupt, so the general low priority interrupt will be enabled back.

10- the CPU start executes the interrupted code again.



interrupt Latency

→ refers to the delay from the start of the interrupt request to the start of interrupt handler execution.



(T_{il}) : interrupt latency.
(T_{int}) : interrupt processing time.
(T_{iret}) : interrupt termination time.

* interrupt latency or interrupt responding time (IRT) : the delay from the start of the interrupt request to the start of interrupt handler execution
* ~~interrupt response time~~ §
* interrupt processing time : the delay from the start of the ISR execution to the end of ISR execution
* interrupt termination time : the delay from the end of the ISR execution to start continue execution the interrupt process.

---

* Shared data and race condition :

→ A common issue when designing software that uses interrupts is how to share data between the ISR and the main program.

→ A race condition is a situation where the outcome varies depending on the precise order (ترتيب دقيق) in which the instructions of the main code and the ISR are executed. This is a problem and should be prevented.

→ it can be extremely difficult to find race condition bugs because interrupts are Asynchronous events, and sometimes they don't happen at all.

→ this can make it tricky to spot the bug during testing. the bug might not show up until after the software has been shipped to the customer, even if it passed all the tests during development.

* Here is an example of race condition :-

```
volatile int gIndex = 0;
interrupt void serial_receiver_ISR (void) {
    gIndex++;
}
int main ( ) {
   while (1) {
        if (gIndex) {
            gIndex--;
        }
   }
}
```

* imagine that the serial port ISR serial_receiver_ISR is invoked when an incoming character arrives
* As characters are received, gIndex variable is incremented to keep track of the number of characters stored in the memory buffer.

→ the main function also uses gIndex variable, by decrementing it when it processes the received characters in the memory buffer.

→ assume that the gIndex = (3) when the line of code decrements the value of this variable

→ this line of code gets turned into a set of instructions in assembly-language
Load → Modify → Store

Load: read the value of gIndex → 3, from its location in RAM into a processor register
Modify: the register value is decremented, resulting in a value of (2)

→ now suppose a serial port receive interrupt occurs before the new value of gIndex is stored in memory → load → modify → stored
interrupt

→ the processor stops executing main and executes the serial port ISR
→ the ISR increments gIndex to a value of (4)
→ then the processor resumes execution of main after the ISR exits.
→ main executes the line of code that stores the register value → (2), back into the variable gIndex
→ now gIndex = 2, as if the latest interrupt never occured to increment the variable.
→ this race condition can cause the program to lose incoming characters.

operation	start	Load	Modify (dec)	interrupt	store (reg→var)
gIndex	3	3	3	4	2
register	—	3	2	—	—

→ the decrement code in the main program is called a "critical section"
→ A critical section : is a part of a program that must be executed in order and automatically → (indivisible or un interrupted)

* So, how is this problem corrected?
→ because an interrupt can occur at any time, the only way to ensure is to disable interrupts during the critical section
→ interrupts are disabled before the critical section execution and then enabled after.

```
while (1) {
    InterruptDisable();
    if (gIndex) {
      gIndex--;
    }
    InterruptEnable();
}
```

* in embedded system, and especially real-time systems, it is important to keep interrupts enabled as much as possible to avoid hindering (ببطئ) the responsiveness of the system

* so you should try to minimize the number of critical sections and the length of critical section code

→ and the safest solution is to save the state of the interrupt enable flag, disable interrupts, execute the critical section, and the restore the state of the interrupt enable flag. Assuming no interrupts enabled at start.