

# 01 Interrupt Feature Explanation

☰ Tags	Handling Interrupt Request	Interrupt Latency	Interrupt Maskability
	Interrupt Pending	Interrupt Priority	Interrupt Service Routine
	Interrupt Types	Non Vectored Interrupt	Polling vs Interrupt
	Shared Data and Race Condition	Vectored Interrupt	

## ▼ Table of Content

### Interrupt Feature

#### Types of Interrupts

Asynchronous Interrupts → Called Interrupts

Synchronous Interrupts → Exceptions

#### Interrupt Service Routine (ISR)

Handling an Interrupt Request

#### Interrupt Maskability

Maskable Interrupt

Non-Maskable Interrupt

Interrupt in PIC18F46K20

#### Interrupt Pending

#### Interrupt vs Polling

Polling

#### Interrupt Priority

Interrupt Nesting

#### Interrupt Vector Table (IVT)

Nested Vector Interrupt Controller (NVIC) in STM32F4

How Reset Operation Works?

#### Non-Vector Interrupt in PIC18F46K20

#### Interrupt Handling

Interrupt Latency

#### Shared Data and Race Condition

Shared Data

Race Condition

Critical Section

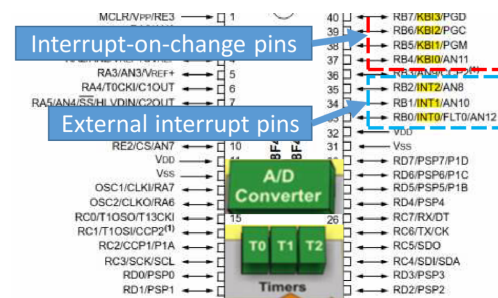
# Interrupt Feature

- An interrupt is an Event [Exception] typically generated by the hardware that requires the CPU to stop normal program execution and start some service (Code) related to the event

## Types of Interrupts

### Asynchronous Interrupts → Called Interrupts

- Raised by a Internal\External hardware device to the **MCU**
- **Asynchronous** events: Generated at random times with respect to the CPU clock signals
- **Internal Interrupt**
  - Raised by internal peripherals inside the microcontroller as:
    1. EUSART: Module received a new byte of data
    2. Timer: Module complete the required time
    3. ADC: Module finishes the ADC conversion
    4. SPI: Module received/transmitted a complete byte of data
    5. I2C: Module received/transmitted a complete byte of data
- **External Interrupt**
  - It has two types:
    1. The [INTx] pins : **Rising/Falling** edge detection.
    2. The [On Change] pins : Interrupt on every **voltage change** on these pins.



### Synchronous Interrupts → Exceptions

- Raised by the **CPU control unit** while executing instructions
- **Synchronous** events: The CPU issues them only after terminating the execution an instruction (Generated by the CPU)
- **There are 3 types**
  1. **Faults**

- The saved value in the register is the address of the instruction that caused the fault.
- Can generally be corrected and once corrected, the program is allowed to restart with no loss of continuity.
- That instruction can be resumed when the exception handler terminates.
- Resuming the same instruction is necessary whenever the handler is able to correct the anomalous condition that caused the exception.
- Examples :
  - **Divide error**: Raised when a program issues an integer division by 0.
  - **Invalid opcode**: The CPU execution unit has detected an invalid opcode (the part of the machine instruction that determines the operation performed).
  - **Floating-point error**: The floating-point unit integrated into the CPU chip has signaled an error condition, such as numeric overflow or division by 0.
  - **Alignment check**: The address of an operand is not correctly aligned (for instance, the address of a long integer is not a multiple of 4).

## 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.
- After the kernel returns control to the program, it is allowed to continue its execution with no loss of continuity.
- A trap is triggered only when there is no need to re-execute the instruction that terminated.
- The main use of traps is for **debugging purposes**.
- The role of the interrupt signal, in this case, is to notify the debugger that a specific instruction has been executed (for instance, a breakpoint has been reached within a program).
- Once the user has examined the data provided by the debugger, the user may ask that execution of the debugged program resume, starting from the next instruction.
- Examples :

- **Breakpoint**: Caused by an int3 (breakpoint) instruction (usually inserted by a debugger).
- **Overflow**: A (check for overflow) instruction has been executed while the OF (overflow) flag is set.

### 3. Aborts

- A serious error occurred.
- Aborts are used to report severe errors, such as **hardware failures**.
- The interrupt signal sent by the control unit is an emergency signal used to switch control to the corresponding abort exception handler.
- This handler has no choice but to **force the affected process to terminate**.
- Examples :
  - **Double fault**: Normally, when the CPU detects an exception while trying to call the handler for a prior exception, the two exceptions can be handled serially. In a few cases, however, the processor cannot handle them serially, so it raises this exception.
  - **Machine-check exception (MCE)**: is a type of computer hardware error that occurs when a computer's central processing unit detects an unrecoverable hardware error in the processor itself, the memory, the I/O devices, or on the system bus.

---

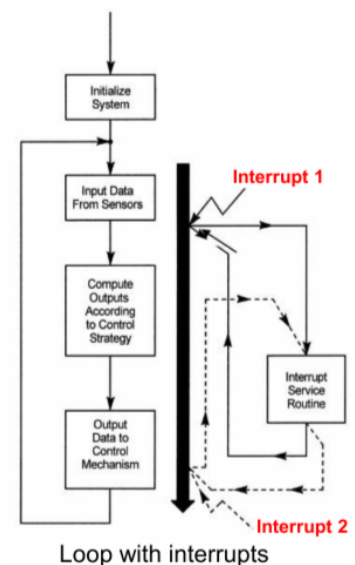
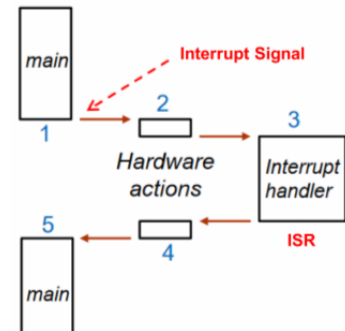
## Interrupt Service Routine (ISR)

- An interrupt service routine (ISR) is a software routine (function) that the processor invokes in response to an interrupt.
- An Exception service routine (ESR) is a software routine (function) that the processor invokes in response to an exception.
- Each **interrupt must own its Service** Routine
- Because the ISR terminate the main program. So, **ISR must perform very quickly** to avoid slowing down the operation of the device.
- The address of this ISR / ESR determined while compile the application and stored in a fixed location in the Flash memory (In the vector table).
  - 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.

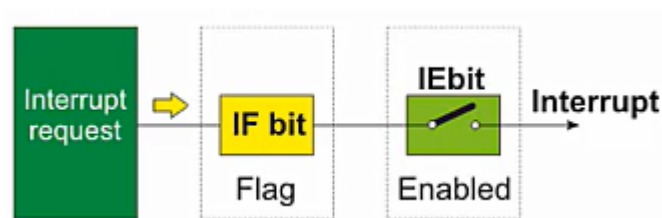
## Handling an Interrupt Request

1. Suspend main thread
  - Stop the execution of the main thread
2. Save CPU state
  - Save the status of the CPU (the data stored in registers) in the **stack** to not to lose our progress in the main thread
3. Execute interrupt handler
  - Picking the address of ISR (Function) of our Interrupt from the vector table
  - Execute the Interrupt handler
4. Restore CPU state
  - After this the previous state will be loaded from the stack to CPU’s registers
5. Resume main thread



## Interrupt Maskability

### Maskable Interrupt



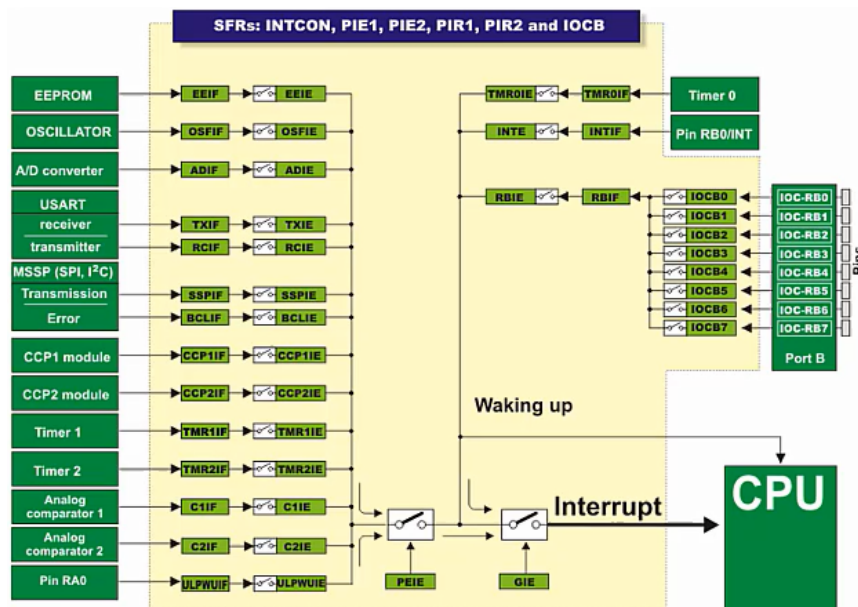
- A maskable interrupt is the interrupt we can (Enable/Disable) it
- Maskable Interrupt has (Enable/Disable) bit

- Storing zero → Even the interrupt signal is sent it doesn't interrupt the CPU
- Storing one → The interrupt signal can be delivered by the CPU
- The interrupt is a Microcontroller specific So we must read the data sheet to know our interrupt features
- Each interrupt has a **flag bit** indicates that the event occurred
- **Example:** Most of internal peripheral

## Non-Maskable Interrupt

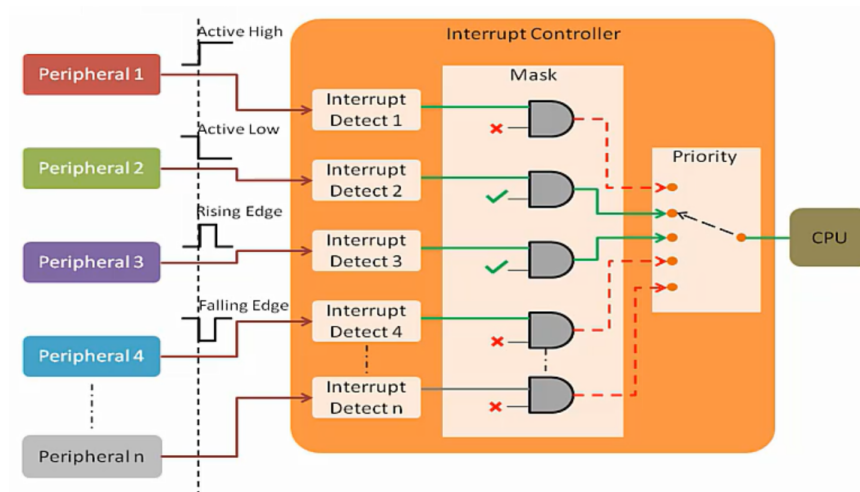
- The kind of interrupt cannot be disabled and must be served
- Each interrupt has a **flag bit** indicates that the event occurred
- **Example:** Reset Button

## Interrupt in PIC18F46K20



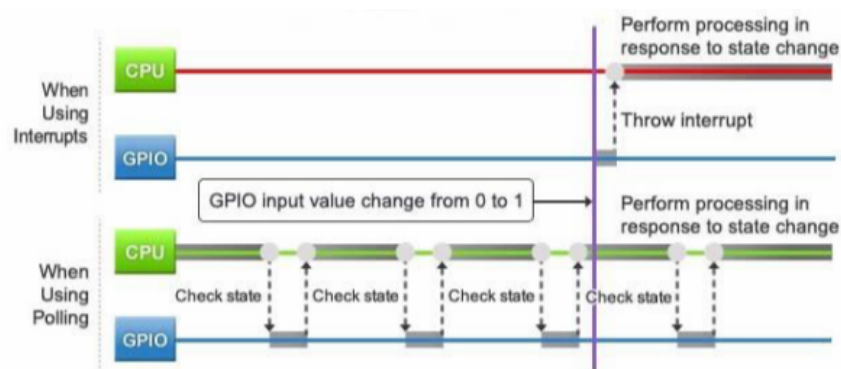
- Each peripheral inside the Microcontroller has its own interrupt with Enable/Disable bit and a flag bit
- The General Interrupt Enable (**GIE**) bit is used to Enable/Disable all the interrupt signals from interrupting the CPU
- The Peripheral Interrupt Enable (**PEIE**) bit is used to Enable/Disable all the interrupt signals from coming from the internal peripheral from interrupting the CPU

## Interrupt Pending



- If the CPU receives more than one interrupt, it handles the interrupt with the highest priority and pend the rest of interrupts until finishes it.
- After handling the highest priority interrupt, it goes to the next interrupt and pend the rest.

## Interrupt vs Polling



## Polling

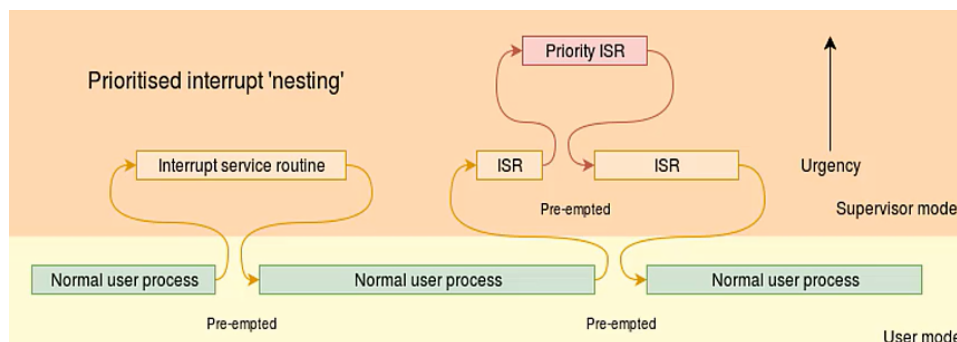
- Polling is not a hardware mechanism, its a protocol in which CPU **steadily checks** whether the device needs attention.
- The device is serviced by CPU.
- Advantage → Efficient if the “Events are rapidly, Happens with High Rate”
  - So it saves the time of **Context Switching** (Stacking & Unstacking)

- Disadvantage → Takes CPU time even when no “Pending Request”

## Interrupt Priority

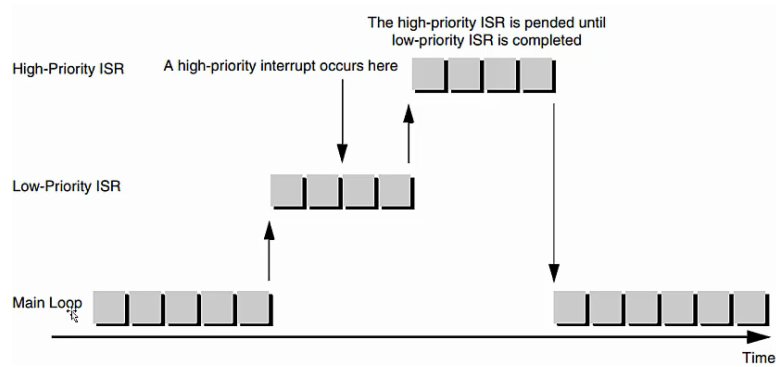
- If the Microcontroller receives more than one interrupt it has to decide which interrupt should receive service first in this situation
- Microcontroller Prioritization can be:
  1. **Fixed** by the hardware and cannot be changed by a software
  2. Changed by the software (**Pre-configured** & **Run-time**)
- We can have priority levels depending on the number of bits the microcontroller provide for prioritization
  - **PIC18**: Provide one bit for prioritization (2 Levels [0 → 1])
  - **PIC24**: Provide three bits for prioritization (7 Levels [0 → 7])

## Interrupt Nesting

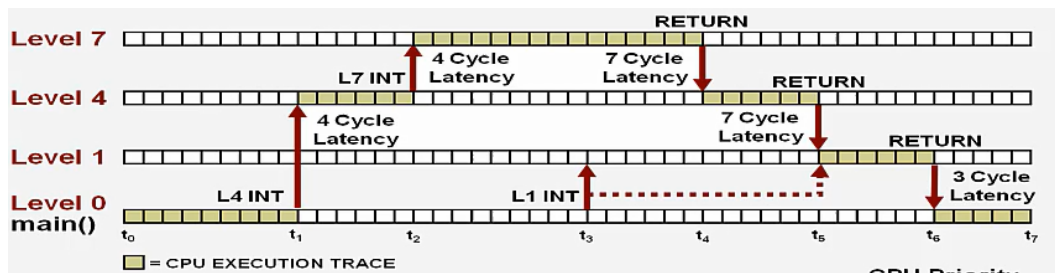


- Interrupt by default are nestble
- Any ISR in a progress may be interrupted by another interrupt source having a higher programmed priority
- If the interrupt is Non-Nested interrupt so the higher priority interrupt cannot interrupt an ISR in a progress and it's pended whatever its priority

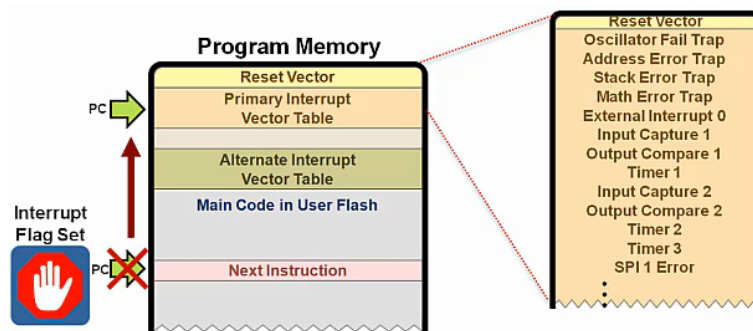




- Example for 7 levels interrupt schedule



## Interrupt Vector Table (IVT)



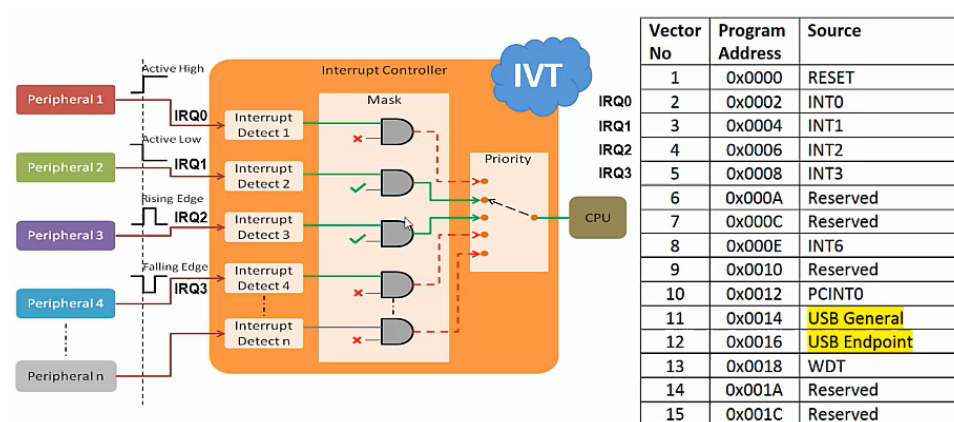
- This location is not the same for all microcontrollers.
- The start location can't be changed by the programmer.
- The IVT is allocated in **flash** memory.
- Only its content can be changed.
  - In Embedded Programming, Vector means memory address.



- The start address of the ISR added to it's Vector **by the compiler** when allocating program memory for the ISRs.
- **How** the processor determines where the ISR is located in code memory for the specific interrupt?
  - When an interrupt is thrown by UART peripheral (for example) so the address of the vector storing its ISR address in the IVT is known and provided in the data sheet.
  - So when UART peripheral sends an interrupt signal the PC immediately points to vector address of UART peripheral in the IVT
- The list also determines **the priority levels of the different interrupts**.
  - The **lower the address the higher is the priority level**.
  - The RESET has the highest priority, and next is INT0 – the External Interrupt Request 0.

Vector No	Program Address	Source	Interrupt Definition	Arduino/C++ ISR() Macro Vector Name
1	0x0000	RESET	Reset	
2	0x0002	INT0	External Interrupt Request 0 (pin D0)	(INT0_vect)
3	0x0004	INT1	External Interrupt Request 1 (pin D1)	(INT1_vect)
4	0x0006	INT2	External Interrupt Request 2 (pin D2)	(INT2_vect)
5	0x0008	INT3	External Interrupt Request 3 (pin D3)	(INT3_vect)

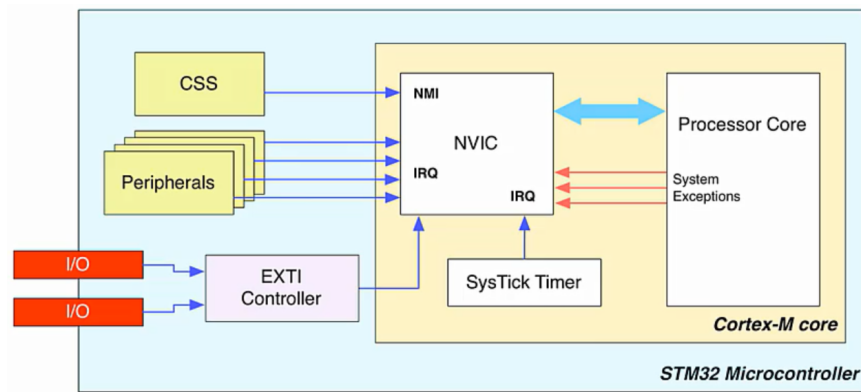
- How the CPU knows which vector in the IVT to jump into for an Interrupt request?



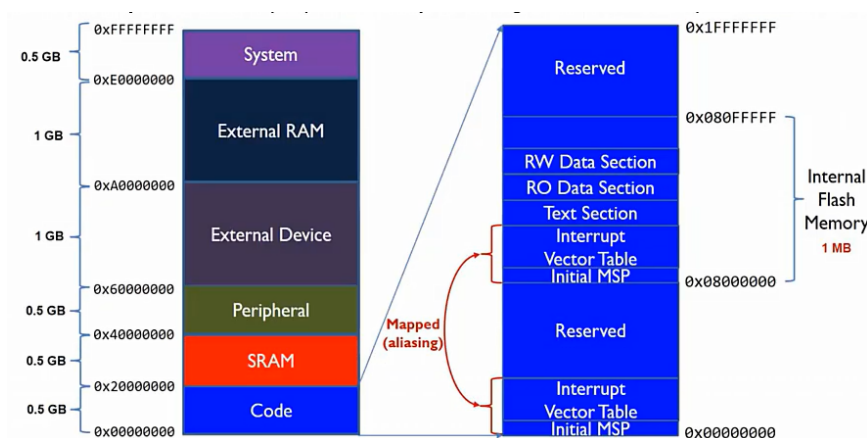
- Each peripheral is connected to the Programmable Interrupt Controller (PIC) with Interrupt Request Line (IRQ).
- When an interrupt signal received by the (PIC) the CPU detects which (IRQ) sends the signal.

- Depending on the IRQ the CPU jumps to a specific address in the IVT with **the help of an equation**

## Nested Vector Interrupt Controller (NVIC) in STM32F4

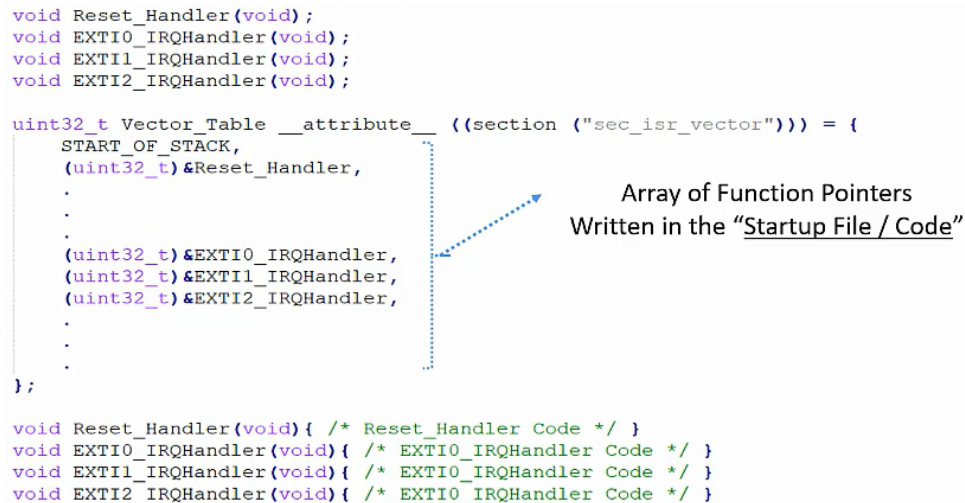


- We have the **Clock Security System** (CSS) which is a non-maskable interrupt for aborting clock's failures
- Peripherals Connected to the NVIC through IRQ lines
- We have I/O Modules connected to NVIC through the **External Interrupt Controller** (EXTI Controller)
- The Processor Core connected with NVIC through IRQ and can throw Synchronous Interrupts **"Exceptions"**



- **Something Specific for STM32F4:** The interrupt vector table in the CortexM4 microprocessor is mapped in the flash memory
  - If we access the address 0x00000000 the microcontroller will jump to 0x08000000 in the flash memory

- This memory aliasing is done for the interrupt vector table only
- For the ARM CortexM4 the interrupts from -14 → -1 are fixed. Interrupts From 0 → 81 are depending on the Microcontroller.



```
void Reset_Handler(void);
void EXTI0_IRQHandler(void);
void EXTI1_IRQHandler(void);
void EXTI2_IRQHandler(void);

uint32_t Vector_Table __attribute__((section("sec_isr_vector"))) = {
    START_OF_STACK,
    (uint32_t)&Reset_Handler,
    .
    .
    (uint32_t)&EXTI0_IRQHandler,
    (uint32_t)&EXTI1_IRQHandler,
    (uint32_t)&EXTI2_IRQHandler,
    .
    .
};

void Reset_Handler(void) { /* Reset_Handler Code */ }
void EXTI0_IRQHandler(void) { /* EXTI0_IRQHandler Code */ }
void EXTI1_IRQHandler(void) { /* EXTI1_IRQHandler Code */ }
void EXTI2_IRQHandler(void) { /* EXTI2_IRQHandler Code */ }
```

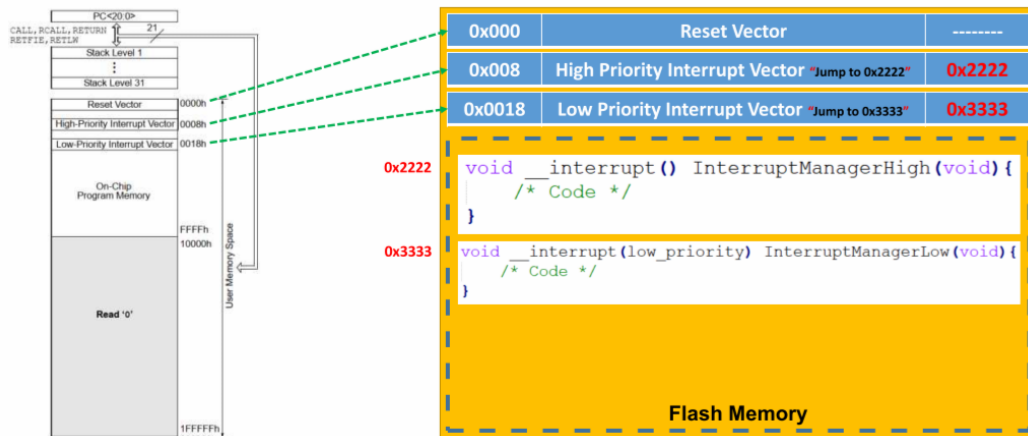
Array of Function Pointers  
Written in the "Startup File / Code"

- So if you open the Data sheet of CortexM4, you will find interrupts from 0 → 81 are not initialized.
- To know about interrupts from 0 → 81 you have to open microcontroller data sheet.
- For our STM32F4, we write ISRs' addresses in the **Startup File/Code** as an **array of pointers to functions** to be stored in our IVT.
- The highest address of SRAM "stack" is stored at 0x00000000 in IVT.
- At the beginning of the program execution START\_OF\_STACK stored in stack pointer register.

## How Reset Operation Works?

- After the non-maskable reset event occurred a set of steps happens
  1. The **PC initialized** with the Reset Handler address
  2. The Stack Pointer **"SP" Initialized** with the End of the SRAM Address
  3. **Loading the global and static data** from Flash .data section to SRAM .data section
  4. **Initializing the .bss** section with zeros
  5. **Initializing the Configuration bits and the Clock system**
  6. **Calling the main** Function

# Non-Vector Interrupt in PIC18F46K20



- When the “Interrupt Occurs”, the PC jump to a specific address “Specific Vector”.
- At least “3 Interrupt Vectors” needs to be defined
  1. Reset Address → **Reset Vector**
  2. High Priority Address → **High Priority Vector**
  3. Low Priority Address → **Low Priority Vector**
- At these “Vectors” there is a “**Jump Instruction**” to a specific ISR.
  - At the “**High Priority Vector**”, There is a “Jump Instruction” to the “**High Priority Vector ISR**”.
  - At the “**Low Priority Vector**”, There is a “Jump Instruction” to the “**Low Priority Vector ISR**”.
  - Note : The system has only “2 ISRs”
    1. High Priority ISR
    2. Low Priority ISR
- This scheme can be **very slow** and there can be a large delay between the time the interrupt occurs and the time it is serviced.
- Supported by many 8-Bit Microcontrollers where the interrupt sources not large ~5 : ~8.
- If we have only two ISRs, How we determine which interrupt occurred if we have 10 interrupt sourced for example?

```
void __interrupt(low_priority) InterruptManagerLow(void){
    if(INTERRUPT_ENABLE == INTCONbits.INT2IE && INTERRUPT_
```

```

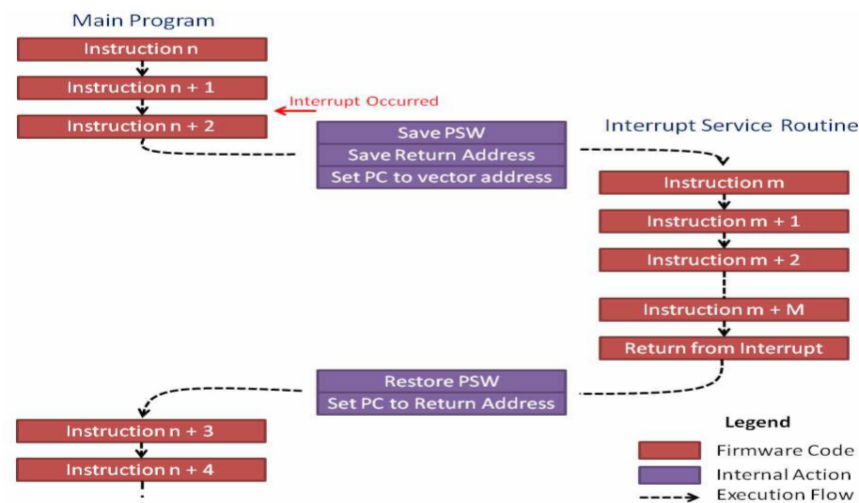
        INT2_Service_Code();
    } else { /* Nothing */}

    if(INTERRUPT_ENABLE == INTCONbits.INT3IE && INTERRUPT_
        INT3_Service_Code();
    } else { /* Nothing */}
    .
    .
    .
}

```

- We divide our interrupts as low/high priority interrupts.
- After an interrupt occurred we jump to its ISR and check flag bits for interrupts.
- This approach will result a large delay between the time the interrupt occurs and the time it is serviced.

## Interrupt Handling

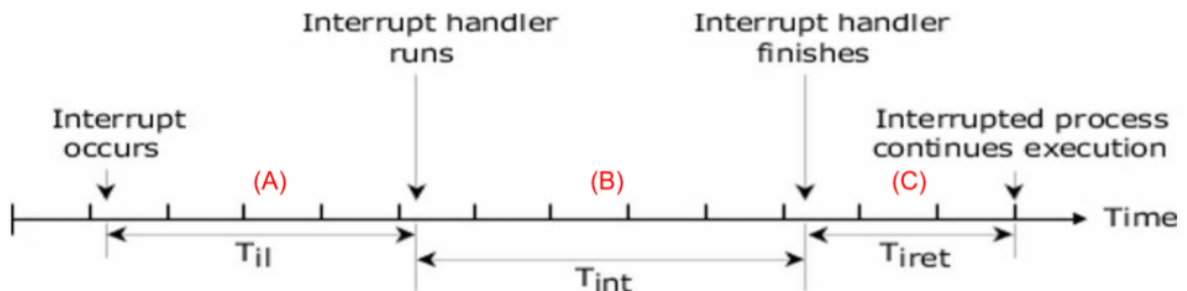


1. An exception occurred (Synchronous event or Asynchronous event) b) The interrupt detected by the processor.
2. The current instruction being executed is completed by the processor before responding to the interrupt and suspend the main program execution.
3. The processor now needs to prepare to transfer control to the interrupt routine. o

- It needs to save information needed to resume the current program at the point of interrupt.
  - This process called “Context Switch”
    - Saving the status of the processor, which is contained in a register called the program status word/register (PSW/PSR) → Copied to the **Shadow Registers**.
    - Saving the location of the next instruction to be executed “**Return Address**”. → **Pushing to the stack**.
4. In single interrupt systems (Low / High), **we have 2 situations**.
    - Interrupt **priority is disabled**: the global interrupt enable will be disabled to prevent any interrupt occurs while responding to the current interrupt.
    - Interrupt **priority is enabled**: if the generated interrupt is a high priority interrupt, so the general high priority interrupt will be disabled and if the generated interrupt is a low priority interrupt, so the general low priority interrupt will be disabled.
  5. The Program counter is set to the vector address specific to that interrupt.
    - Here we have 2 interrupt schema (Vectored and Non Vectored).
    - A piece of code called the Interrupt Service Routine (ISR) is placed in the vector location of an interrupt, to handle it.
  6. Execute the ISR of this interrupt → Until reaching a **RETI** “Return from Interrupt” instruction.
  7. Restore the internal register called PSW “Program Status Word” or “PSR : Program Status Register” from the shadow registers.
  8. Restore the return address from the stack or from the internal backup register and load it to the PC.
  9. In single interrupt systems (Low / High), **we have 2 situations**.
    - Interrupt priority is disabled: the global interrupt enable will be enabled back.
    - 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 executing the interrupted code again.

## Interrupt Latency



- **Interrupt Latency (A)**: The delay from the start of the interrupt request to the start of interrupt handler execution.
- **Interrupt Response Time (IRT) (A)**: The delay from the start of the interrupt request to the start of interrupt handler execution.
- **Interrupt Processing Time (B)**: The delay from the start of the ISR execution to the end of ISR execution.
- **Interrupt Termination Time (C)**: The delay from the end of the ISR execution to start continue execution the interrupted process.

## Shared Data and Race Condition

### Shared Data

- We say two functions have shared data if they both can read/write for one or more variable(s)
  - Example

```
volatile uint8 var = 0;

void isr1(void) {var++;}
void isr2(void) {var--;}
// Here var is shared data between isr1 and isr2
int main(void) {
    while (1) { /* Code */ }
```

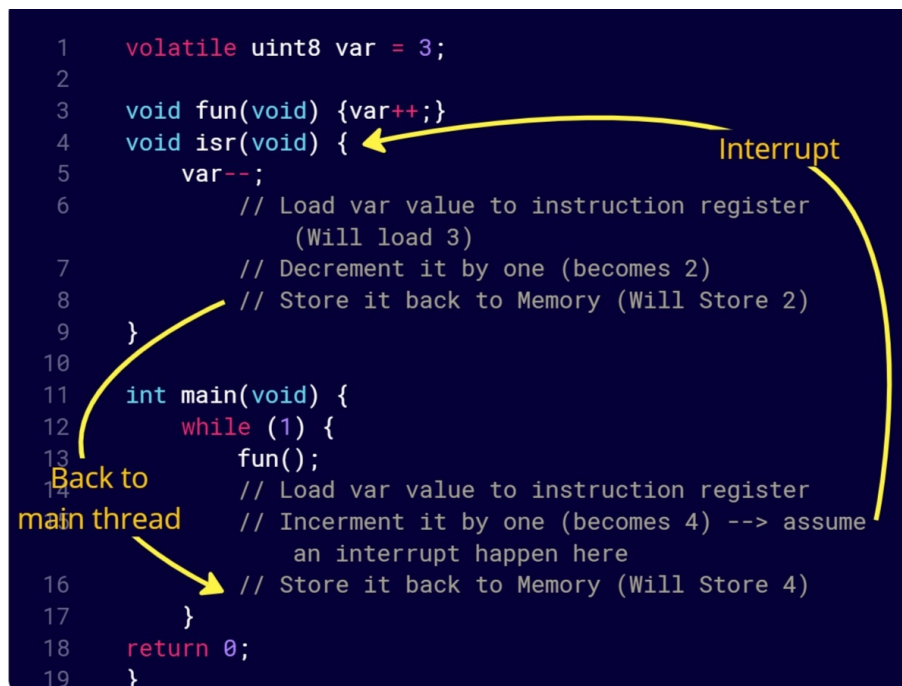


```
return 0;
}
```

- **Shared Resources:** a peripheral or one of peripheral's register if they are shared between more than one API is called shared resources

## Race Condition

- A race condition is a situation where the outcome varies depending on the precise order in which the instructions of the main code and the ISR are executed.
  - This should be **strenuously avoided**.
- It can be extremely **difficult to find race condition bugs** because interrupts are asynchronous events and, to make matters worse, the race condition doesn't occur every time the code executes.
- Your software can run for days and pass all production tests without exhibiting this bug but then, once the unit is shipped to the customer, it is certain to show up.
- Race conditions can be a pitfall with interrupts and Operating Systems (as RTOS)
- **Example**



- Here in this example the `var` will store 4 in the memory
- In this situation the effect of ISR is **erased**

## Critical Section

- 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 atomically, or without interruption.
  - A line of C code (even as trivial as increment or decrement) is not necessarily atomic, as we’ve seen in this example.
- So, how is this problem corrected ?
  - Because an interrupt can occur at any time, the only way to make such a guarantee is to disable interrupts during the critical section.
  - In order to protect the critical section as in the previous example code, interrupts are disabled before the critical section executes and then enabled after.

```
int main(void)
{
    while (1)
    {
        InterruptDisable();
        if (gIndex)
        {
            /* Process receive character in memory buffer. */
            gIndex--;
        }
        InterruptEnable();
    }
}
```