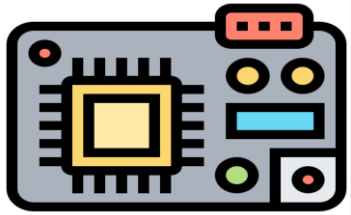
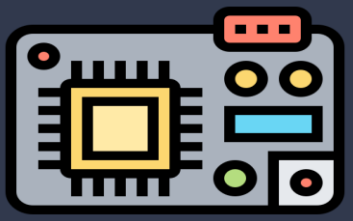


Interrupts

CSE 3105 - Computer Interfacing & Embedded System

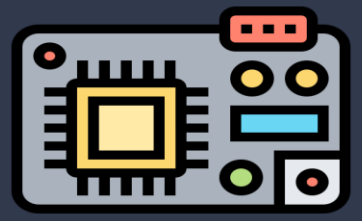
Md. Nasif Osman Khansur
Lecturer, Dept. of CSE
Rajshahi University of Engineering & Technology, Rajshahi - 6204



Introduction to Interrupts

An interrupt is a signal to the processor indicating an event that needs **immediate attention**, causing the processor to temporarily halt its current execution and jump to execute an Interrupt Service Routine (ISR) before returning to its previous activity.

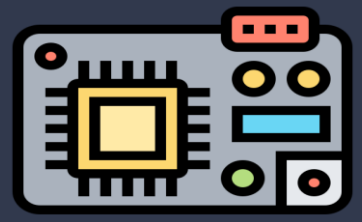
The halted process continues as if nothing had happened.



Preemptive Interrupt

A **preemptive interrupt** allows a higher-priority task to interrupt and temporarily halt a lower-priority task so that the higher-priority task can be executed immediately.

Example: In a microcontroller managing both a temperature sensor and a motor control, if the temperature sensor's reading indicates a critical condition, a preemptive interrupt can halt the motor control task to immediately process the temperature data and take necessary actions to prevent overheating.

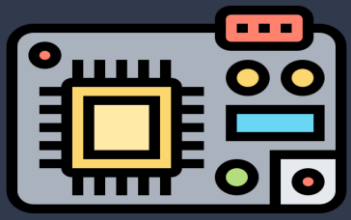


Non-preemptive Interrupt

A **non-preemptive interrupt** allows the processor to complete the current task before switching to handle the interrupt, ensuring that tasks are executed sequentially without interruption.

Example: If a microcontroller is executing a routine to read a sensor value and a non-preemptive interrupt occurs to log data to memory, the microcontroller will finish reading the sensor value before logging the data.

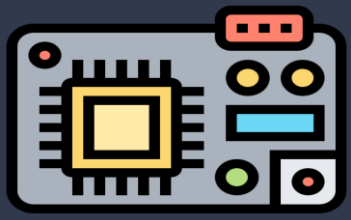
Interrupt vs Polling/Busy Wait



Interrupt: **An interrupt is a signal sent to the processor by hardware or software to indicate that an event needs immediate attention**. Instead of constantly checking (polling) whether a condition has occurred, the system gets interrupted and responds accordingly. **This allows the CPU to perform other tasks until the interrupt occurs, making it more efficient**.

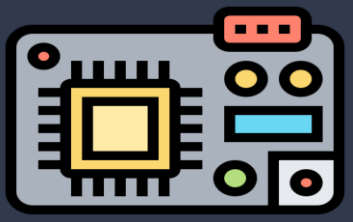
For example, ***A keyboard input system***: When a key is pressed, the keyboard sends an interrupt signal to the CPU. The CPU pauses its current task, processes the keystroke, and then resumes its task. The CPU doesn't waste time constantly checking (polling) for key presses.

Interrupt vs Polling/Busy Wait



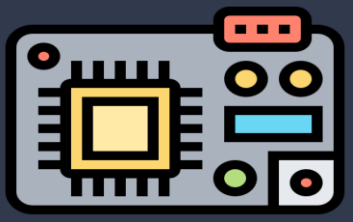
Polling: **Busy waiting or polling is a process where the CPU repeatedly checks a condition or a status flag in a loop to see if an event has occurred**. This method keeps the CPU busy, even when the event hasn't occurred, **leading to inefficiency because the CPU cannot perform other tasks during this time**.

For example, ***A network device:*** If the CPU needs to know when data is available from the network, in a polling system, it continuously checks the network buffer to see if new data has arrived. This keeps the CPU occupied and unable to perform other tasks efficiently, even when no new data is available.



Interrupt Numbers

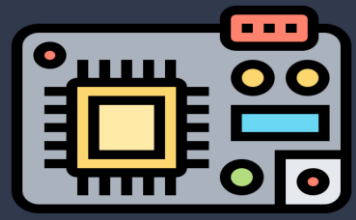
- **Cortex-M processors support up to 256 types of interrupts.**
- Each interrupt type, excluding the **reset** interrupt, is identified by a unique number, ranging from -15 to 240.
- Interrupt numbers are defined by ARM and chip manufacturers collectively. These numbers are fixed and software cannot redefine them.
- **Blue Pill has 43 interrupts (Excluding 16 system interrupts.)**
- **Interrupt numbers are divided into two groups.**
  - **System Interrupts**
  - **Peripheral Interrupts**



System Interrupts

- **The first 16 interrupts are system interrupts, also called system exceptions.** *Exceptions are the interrupts that come from the processor core.*
- These interrupt numbers are defined by ARM. Specifically, the ARM CMSIS library defines all system exceptions by using negative values.
- CMSIS stands for Cortex Microcontroller Software Interface Standard.

System Interrupts

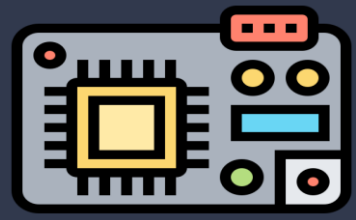


System interrupts in Cortex-M microcontrollers are hardware-triggered events that temporarily halt normal execution flow to handle time-critical tasks or respond to external stimuli. Some important system interrupts are:

- ★ Reset (Reset Handler):
  - Triggered when the system is reset due to power-on, external reset, or a software-induced reset.
  - Initializes the microcontroller system to its default state and begins program execution from the reset vector.

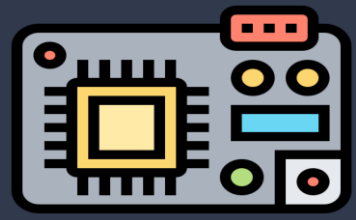
- ★ Non-Maskable Interrupt (NMI):
  - A high-priority interrupt, used for events that cannot be delayed, like system faults or power failures. For example, NMI can be tied to a watchdog timer that triggers when the system fails to reset it in time.

System Interrupts

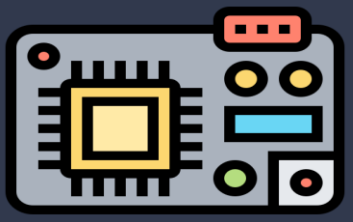


- ★ Hard Fault:
  - Triggered by a memory access violation, divide-by-zero, or any other critical fault in the system.
- ★ SysTick Timer Interrupt:
  - A configurable timer interrupt built into the Cortex-M core, typically used for system timekeeping or task scheduling in an RTOS (Real-Time Operating System).
- ★ PendSV (Pendable Service Call):
  - A low-priority interrupt, often used in conjunction with SysTick for context switching in an RTOS.

System Interrupts



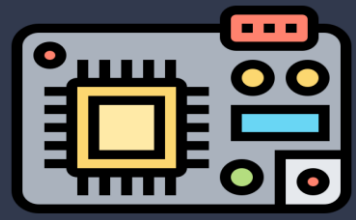
- ★ Bus Fault:
  - Triggered when an invalid access occurs on the system bus.
- ★ Usage Fault:
  - Useful for debugging incorrect operations in code, especially related to invalid or erroneous instructions.
- ★ SVCall (Supervisor Call):
  - Used for RTOS services, like triggering a system call or switching to supervisor mode for certain tasks (system-level services or privileged tasks).



Peripheral Interrupts

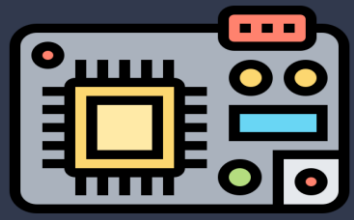
- The remaining 240 interrupts are peripheral interrupts, also called non-system exceptions.
- Blue Pill has 43 peripheral interrupts.
- **The peripheral interrupt numbers start at 0**. Peripheral interrupts are defined by chip manufacturers.
- The total number of peripheral interrupts supported varies among chips.

Peripheral Interrupts



- UART (Universal Asynchronous Receiver-Transmitter) Interrupts
  - TX (Transmit) Interrupt
  - RX (Receive) Interrupt
  - Error Interrupt
- Timer Interrupts
  - General Purpose Timer Interrupt
  - Watchdog Timer Interrupt
- GPIO (General Purpose Input/Output) Interrupts
- ADC (Analog-to-Digital Converter) Interrupts
  - End of Conversion (EOC) Interrupt
  - Analog Watchdog Interrupt
- SPI (Serial Peripheral Interface) Interrupts
- I2C (Inter-Integrated Circuit) Interrupts
- DMA (Direct Memory Access) Interrupts
- USB Interrupts

CMSIS Definition of interrupt numbers for STM32L4



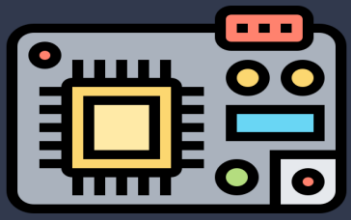
Cortex-M4 Processor Exceptions Numbers

-14	Non-maskable interrupt
-13	Hard fault
-12	Memory management
-11	Bus fault
-10	Usage fault
-5	Supervisor call (SVCall)
-4	Debug monitor
-2	PendSV
-1	SysTick

STM32L4 specific Interrupt Numbers

0	WWDG	16	DMA1_CH6	32	I2C1_ER	48	FMC	64	COMP	80	RNG				
1	PVD	17	DMA1_CH7	33	I2C2_EV	49	SDMMC1	65	LPTIM1	81	FPU				
2	TAMPER_STAMP	18	ADC1_ADC2	34	I2C2_ER	50	TIM5	66	LPTIM2						
3	RTC_WKUP	19	CAN1_TX	35	SPI1	51	SPI3	67	OTG_FS						
4	FLASH	20	CAN1_RX0	36	SPI2	52	UART4	68	DMA2_Channel6						
5	RCC	21	CAN1_RX1	37	USART1	53	UART5	69	DMA2_Channel7						
6	EXTI0	22	CAN1_SCE	38	USART2	54	TIM6_DAC	70	LPUART1						
7	EXTI1	23	EXTI9_5	39	USART3	55	TIM7	71	QUADSPI						
8	EXTI2	24	TIM1_BRK	40	EXTI15_10	56	DMA2_Channel1	72	I2C3_EV						
9	EXTI3	25	TIM1_UP	41	RTC_Alarm	57	DMA2_Channel2	73	I2C3_ER						
10	EXTI4	26	TIM1_TRG	42	DFSDM3	58	DMA2_Channel3	74	SAI1						
11	DMA1_CH1	27	TIM1_CC	43	TIM8_BRK	59	DMA2_Channel4	75	SAI2						
12	DMA1_CH2	28	TIM2	44	TIM8_UP	60	DMA2_Channel5	76	SWPMI1						
13	DMA1_CH3	29	TIM3	45	TIM8_TRG	61	DFSDM0	77	TSC						
14	DMA1_CH4	30	TIM4	46	TIM8_CC	62	DFSDM1	78	LCD						
15	DMA1_CH5	31	I2C1_EV	47	ADC3	63	DFSDM2	79							

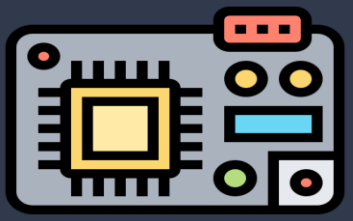
Interrupt Numbers



When an interrupt is processed, the interrupt number is stored in the program status register (PSR).

ARM Cortex-M does not store interrupt numbers in two's complement. Instead, the interrupt number in PSR adds a positive offset of 15 to the CMSIS interrupt number.

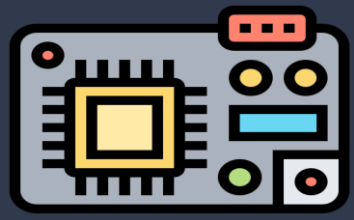
Interrupt number in PSR = CMSIS interrupt number + 15



Interrupt Service Routines

- An Interrupt Service Routine (ISR) is a function that is called in response to an interrupt.
- It handles the specific event that caused the interrupt and typically performs tasks such as updating variables, processing data, or interacting with peripherals.

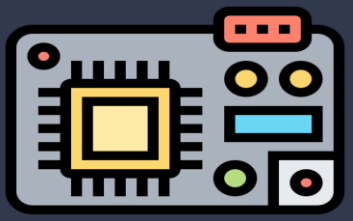
Interrupt Service Routines



The default implementation of most ISRs is simply a **dead loop. For example, interrupt handler for the system timer:**

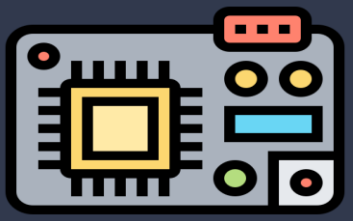
```
SysTick_Handler   PROC
                  EXPORT  SysTick_Handler  [WEAK]
                  B       .                ; dead loop
                  ENDP
```

The keyword **weak** means that another non-weak subroutine with the same name defined elsewhere can override this one.



Interrupt Vector Table

- **Cortex-M stores the starting memory address of every ISR in a special array called the interrupt vector table.**
- An interrupt number is used as an index into the interrupt vector table to locate the corresponding interrupt service routine.
- For a given interrupt number *i* defined in CMSIS, the memory address of its corresponding ISR is located at the $(i + 16)^{th}$ entry in the interrupt vector table.
- The interrupt vector table is stored at the memory address 0x00000004. **Because each entry in the table represents a memory address, each entry takes four bytes in memory.** [0x00000000 is for the initial stack pointer (MSP)]



Interrupt Vector Table

Address of ISR = InterruptVectorTable[i + 16]

For example,

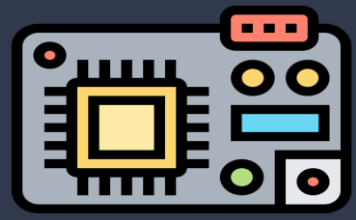
1. the interrupt number of SysTick is -1, the memory address of SysTick_Handler can be founding by reading the word stored at the following address.

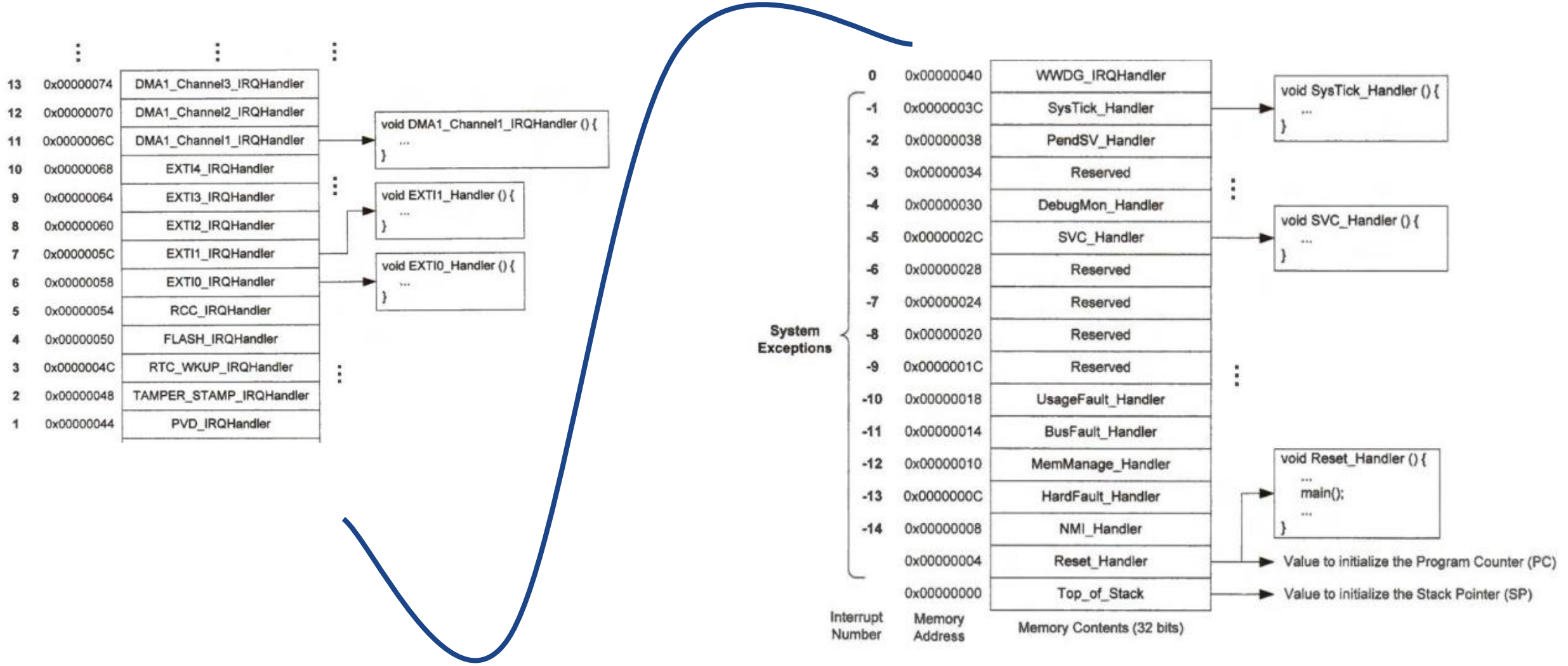
Address of SysTick_Handler
= 0x00000000 + 4 x (-1 + 16)
= 0x0000003C

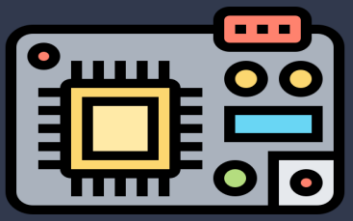
2. The interrupt number of reset is -15. Thus, the memory address of Reset_Handler is

Address of Reset_Handler
= 0x00000000 + 4 x (-15 + 16)
= 0x00000004

Interrupt Vector Table

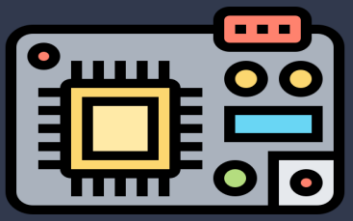






The Booting Process Of Cortex-M

- When an ARM Cortex processor is turned on or reset, the processor fetches two words located at 0x00000000 and 0x00000004 in memory.
- The processor uses the word located at 0x00000000 to initialize the main stack pointer (MSP), and the other one at 0x00000004 to set up the program counter (PC).
- The word stored at 0x00000004 is the memory address of the Reset_Handler(). Typically, Reset_Handler() calls **main()**, which is the user's application code. After PC is initialized, the program begins execution.

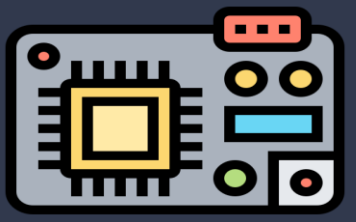


Interrupt Stacking and Unstacking

Interrupt Stacking

When an interrupt occurs, the processor saves the current context, which includes registers, program counter, and status register, onto the stack, ensuring that the system can return to the exact state it was in before the interrupt. This process is known as **stacking**.

- Before executing the interrupt handler, the stacking process automatically pushes **eight registers** to preserve the running environment.
- These eight registers include the lowest four registers (r0, r1, r2, and r3) and four other registers (r12, LR, PSR, and PC).



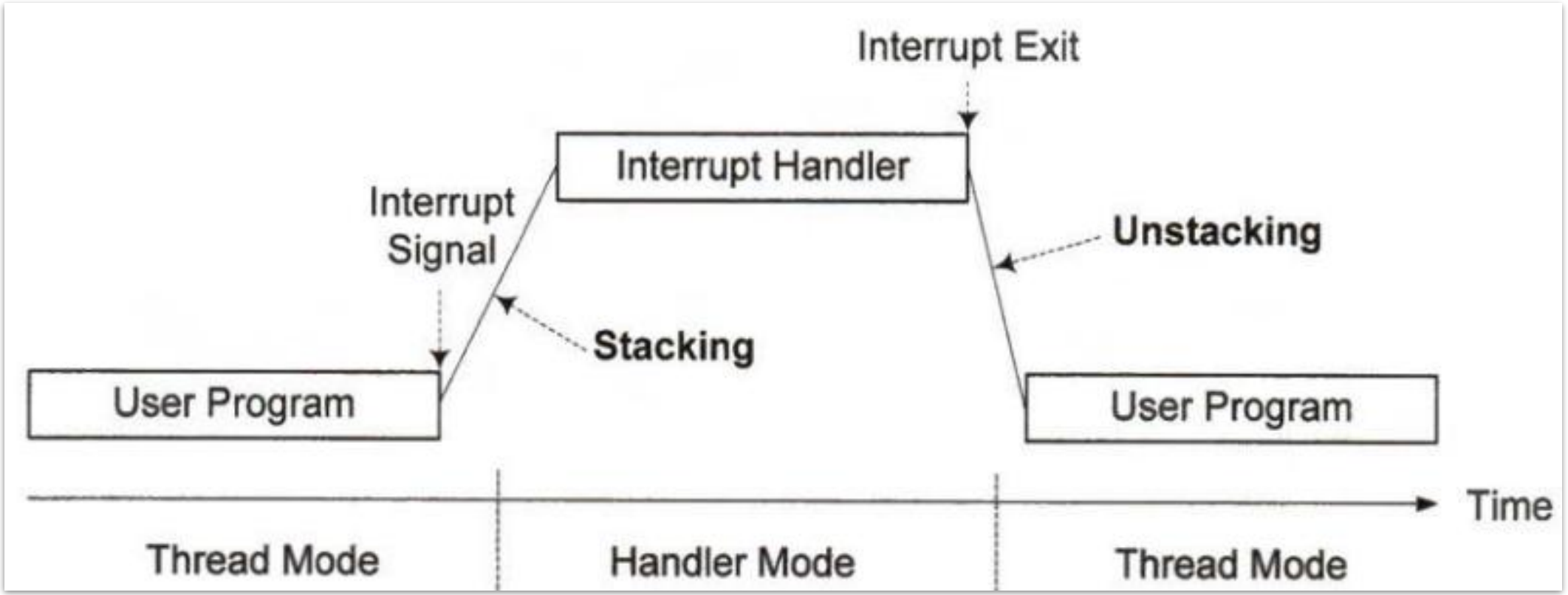
Interrupt Stacking and Unstacking

Interrupt Unstacking

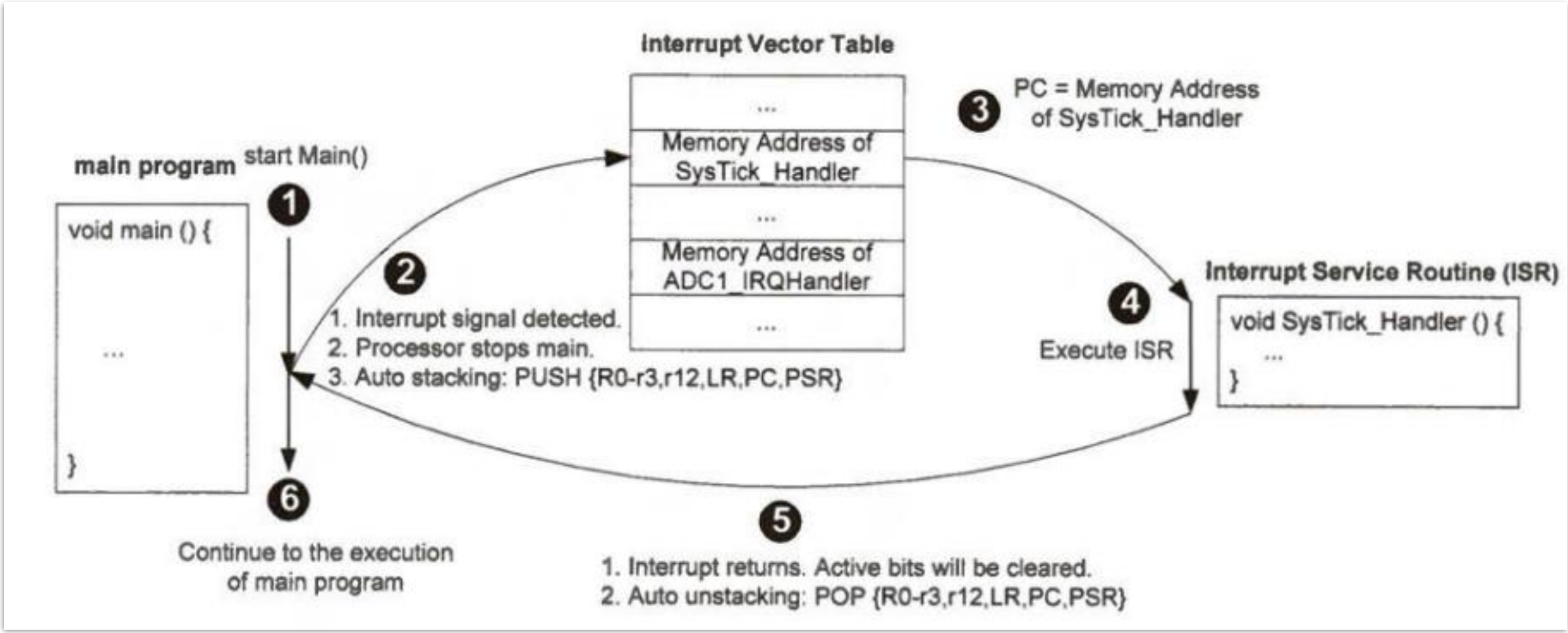
Once the interrupt service routine (ISR) completes, the **unstacking** process restores the saved context from the stack, allowing the processor to resume normal operation seamlessly.

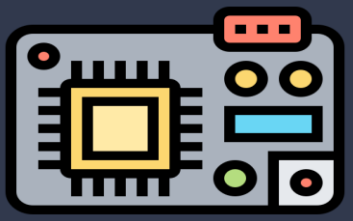
- After the interrupt handler completes, the unstacking process automatically pops the values of these eight registers off the stack.
- At the same time, the processor clears the corresponding active bits in the NVIC status registers.

Automatic stacking and unstacking for interrupt handler



Steps of stacking and unstacking for interrupt handler



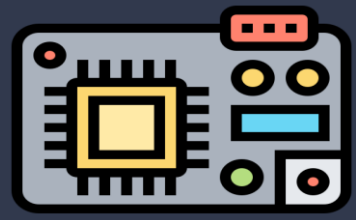


Nested Vectored Interrupt Controller (NVIC)

The nested vectored interrupt controller (NVIC) is built into Cortex-M cores to manage all interrupts. It offers three key functions:

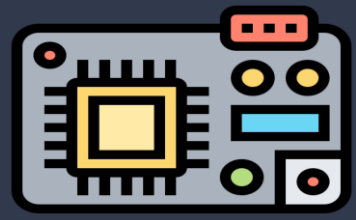
1. Enable and disable interrupts.
2. Configure the preemption priority and subpriority of a specific interrupt.
3. Set and clear the handing bit of a specific interrupt.

Interrupt control bits



Interrupt control bit	Corresponding register (32 bits)
Enable bit	Interrupt set enable register (ISER)
Disable bit	Interrupt clear enable register (ICER)
Pending bit	Interrupt set pending register (ISPR)
Un-pending bit	Interrupt clear pending register (ICPR)
Active bit	Interrupt active bit register (IABR)
Software trigger bit	Software trigger interrupt register (STIR)

Nested Vectored Interrupt Controller (NVIC)



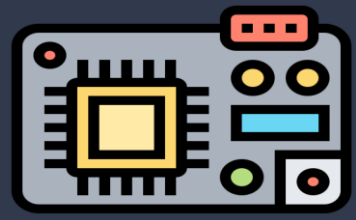
Enable and disable an interrupt

Writing an **enable bit to 1** can enable the corresponding interrupt. Writing an **enable bit to 0** does not turn off the corresponding interrupt. Write a **disable bit to 1** can disable the interrupt. Writing a **disable bit to 0** has no impacts on the related interrupt. Separating enable bits and disable bits allows us to disable an interrupt conveniently without affecting the other interrupts.

ISER = Interrupt Set Enable Register

ICER = Interrupt Clear Enable Register

Nested Vectored Interrupt Controller (NVIC)



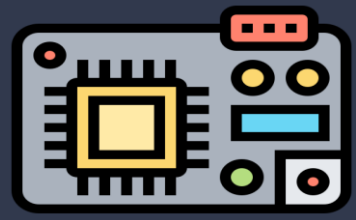
Pend and clear an interrupt

If an interrupt occurs, the corresponding pending **bit is set** if the microcontroller cannot process this interrupt immediately. Writing the **clear pending bit to 1** removes the corresponding interrupt from the pending list.

ISPR = Interrupt Set Pending Register

ICPR = Interrupt Clear Pending Register

Nested Vectored Interrupt Controller (NVIC)

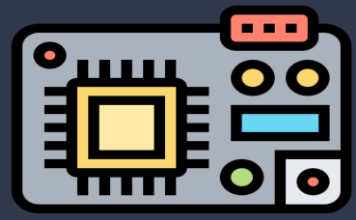


Trigger an interrupt

Setting an active bit by software or hardware activates the related interrupt, and the microcontroller starts the corresponding interrupt handler. If software writes a trigger bit of the software trigger interrupt register **(STIR) to 1**, the related interrupt is also activated.

IABR = Interrupt Active Bit Register

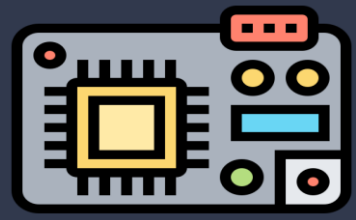
Nested Vectored Interrupt Controller (NVIC)



Cortex-M processors can have up to–

1. Eight 32-bit Interrupt Set-Enable Registers (ISER). ISER[0] - ISER[7]
2. Eight 32-bit Interrupt Clear-enable Register (ICER). ICER[0] - ICER[7]
3. Eight 32-bit Interrupt Set-Pending Registers (ISPR). ISPR[0] - ISPR[7]
4. Eight 32-bit Interrupt Clear-Pending Register (ICPR). ICPR[0] - ICPR[7]
5. Eight 32-bit Interrupt Active Bit Register (IABR). IABR[0] - IABR[7]

Enable and Disable Peripheral Interrupts

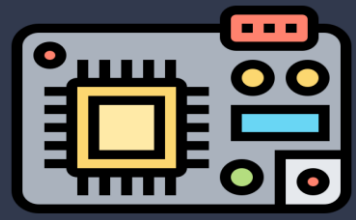


Two steps you need to follow:

1. Find out the appropriate ISER[i] or ICER[i] register. For this, apply: **ISER[i] = [IRQn / 32]** **[Note: IRQn stands for Interrupt Request Number]**
2. Find out the bit position in the ISER[i] or ICER[i] register. For this, apply: **IRQn % 32.**

* **Same steps go for ISPR, ICPR, and IABR.**

Enable and Disable Peripheral Interrupts

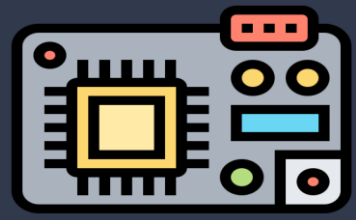


For example, Given IRQn = 40.

1. The corresponding registers are ISER[1] and ICER[1]. Since, IRQn /32 = 40 / 32 ≈ 1
2. To enable or disable the interrupt, the bit position in ISER[1] is IQRn % 32 =8

That means, to enable the interrupt, you need to set bit 8 in NVIC_ISER[1] and to disable the interrupt, you need to clear bit 8 in NVIC_ICER[1]

Enable and Disable Peripheral Interrupts

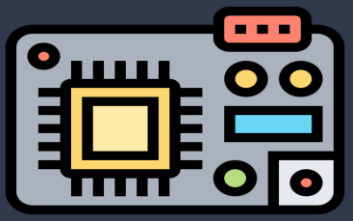


```
#define IRQn_40 40
void NVIC_EnableIRQ(uint32_t IRQn)
{
  // Calculate which ISER register to use and which bit to set
  NVIC->ISER[IRQn / 32] = (1 << (IRQn % 32));
}
NVIC_EnableIRQ(IRQn_40); // Enable IRQn = 40
```

IRQn % 32 or IRQn >> 5: often addressed as **word offset.**
IRQn / 32 or IRQn & 0x1F: often addressed as **bit offset.**

(1 << (IRQn % 32)): This expression shifts a 1 to the correct bit position within the selected register.

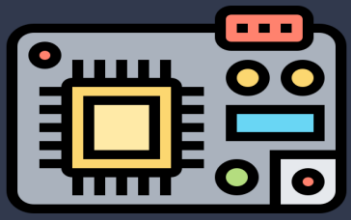
```
#define IRQn_40 40
void NVIC_DisableIRQ(uint32_t IRQn)
{
  // Calculate which ICER register to use and which bit to clear
  NVIC->ICER[IRQn / 32] = (1 << (IRQn % 32));
}
NVIC_DisableIRQ(IRQn_40); // Disable IRQn = 40
```



Interrupt Priority

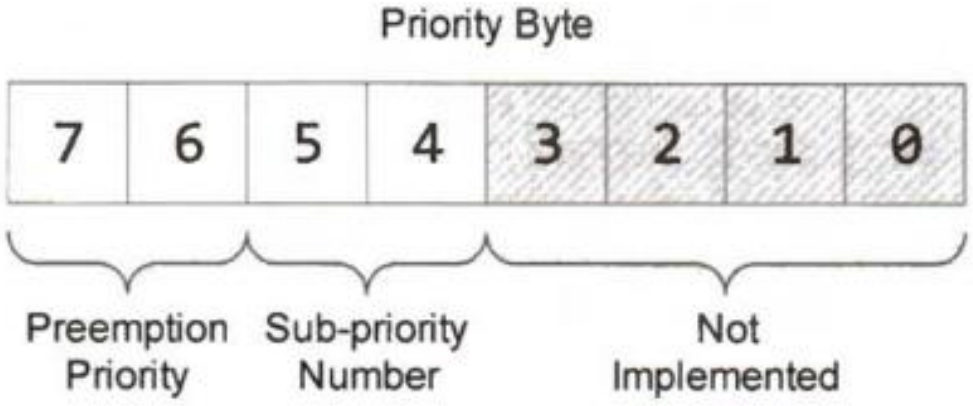
- **The processor serves all interrupts based on their priority levels.**
- The processor stops the currently running interrupt handler if a new interrupt with a higher priority occurs.
- The new interrupt task preempts the current lower-priority task, and the processor resumes the low-priority task when the handler of the new interrupt completes.
- **A higher value of the interrupt priority number represents a lower priority (or urgency). The Reset_Handler() has top priority, and its priority number is -3.**

Interrupt Priority

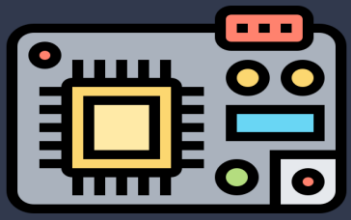


Each interrupt has an interrupt priority register (IP), which has a width of 8 bits. Each consists of two fields: the preemption priority number and the sub-priority number. While Cortex-M processors use eight bits to store the priority number, STM32L processors only implement four bits.

If we use n bits for the preempt priority number, then the sub-priority number has (4 - n) bits, where n = 0, 1, 2, 3, or 4. By default, two bits are used for the preempt priority number, and two bits are used for the sub-priority number, as shown in Figure.



References



Ref. Book: Embedded Systems with ARM Cortex-M Microcontrollers in Assembly Language and C 3rd Ed - Yifeng Zhu - Eman (2018) **[Chapter 11]**


Thank
you