6. 인터럽트와 예외 처리

6.1 인터럽트 개념과 종류

🧠 인터럽트란?

인터럽트는 **현재 실행 중인 작업을 중단하고,**
긴급하거나 우선적인 작업을 CPU에게 처리하도록 하는 **메커니즘**이다.

> 즉, CPU는 프로그램이 순차적으로 실행되는 동안, 특정 조건에서 **외부 또는 내부 이벤트**에 의해 실행 흐름을 변경할 수 있다.
> 이를 통해 더 중요한 작업을 즉시 처리할 수 있다.

주요 특징:

1. **비동기적**: 인터럽트는 CPU가 예기치 않게 외부 또는 내부 사건을 처리하도록 유도함.
2. **우선순위**: 다수의 인터럽트가 동시에 발생하면, **우선순위**에 따라 처리됨.
3. **핸들러**: 인터럽트가 발생하면, 해당 인터럽트를 처리하는 **인터럽트 서비스 루틴(ISR)**이 호출됨.

🏁 인터럽트의 동작 과정

1. **인터럽트 발생**: 외부 장치 또는 내부 이벤트가 인터럽트를 발생시킴.
2. **현재 작업 중단**: CPU는 현재 실행 중인 명령어를 중단하고, 인터럽트 처리를 위한 준비를 시작.
3. **인터럽트 서비스 루틴(ISR) 실행**: CPU는 **ISR**을 호출하여 해당 인터럽트를 처리.
4. **복귀**: 인터럽트 서비스 루틴이 완료되면, CPU는 **인터럽트 발생 이전**의 작업으로 돌아감.

> 인터럽트가 발생하면 **현재 상태**(레지스터 값 등)를 **스택에 저장**하고, ISR이 끝나면 이 값을 복원하여 계속 작업을 수행함.

🔧 인터럽트 종류

인터럽트는 크게 **외부 인터럽트**와 **내부 인터럽트**로 구분되며, 각 유형에 따라 세부적으로 나뉜다.

1️⃣ 외부 인터럽트 (External Interrupts)

외부 인터럽트는 외부 하드웨어 장치나 외부 사건에 의해 발생하는 인터럽트이다.

- **I/O 장치 인터럽트**: 키보드, 마우스, 네트워크 카드, 센서 등 외부 장치에서 발생.
- **타이머 인터럽트**: 내장 타이머가 특정 시간 후 발생.
- **전원 관리 인터럽트**: 배터리 부족, 전원 상태 변화 등에서 발생.

예시:

- **키보드 입력**: 키보드에서 키를 누르면, CPU는 키보드 인터럽트를 받아 해당 키값을 처리.
- **타이머 인터럽트**: 시스템이 일정 주기로 인터럽트를 발생시켜 시간을 측정하거나, 주기적인 작업을 실행.

2️⃣ 내부 인터럽트 (Internal Interrupts)

내부 인터럽트는 프로그램 실행 중에 발생하는 오류나 예외에 의해 발생하는 인터럽트이다.

- **예외(Exceptions)**: 프로그램 실행 중 발생할 수 있는 오류(예: 0으로 나누기, 접근 불가 메모리 주소 등)
- **트랩(Traps)**: 소프트웨어나 운영체제에서 사용자가 정의한 조건에 의해 발생하는 인터럽트. 주로 디버깅, 시스템 호출 등에서 사용.

예시:

- **분할 오류 (Divide by Zero)**: 나누기 연산에서 0으로 나누면, 프로그램에 **내부 인터럽트**가 발생하여 오류 처리를 함.
- **페이지 폴트 (Page Fault)**: 프로그램이 접근하려는 메모리 페이지가 없는 경우, 이를 처리하기 위한 **인터럽트**가 발생.

3️⃣ 소프트웨어 인터럽트 (Software Interrupts)

소프트웨어 인터럽트는 **프로그램에 의해 의도적으로 발생**하는 인터럽트야. 보통 시스템 호출을 통해 운영체제와 상호작용할 때 사용된다.

- **시스템 호출**: 사용자 프로그램이 운영체제의 서비스를 요청할 때 사용.
- **디버깅 및 트랩**: 프로그램에서 특정 조건을 확인하고, 디버깅 또는 오류 처리를 위해 인터럽트를 발생시킬 수 있음.

예시:

- **시스템 호출**: `int 0x80` (x86 아키텍처)와 같은 명령어로, 프로그램이 운영체제의 서비스를 요청하는 시스템 콜.
- **디버깅 트랩**: 디버거가 중단점을 설정하여, 프로그램이 특정 지점에 도달하면 중단하고 디버깅을 시작하는 인터럽트.

4️⃣ 하드웨어 인터럽트 (Hardware Interrupts)

하드웨어 인터럽트는 외부 장치에서 발생하는 **실제 하드웨어 이벤트**로 인해 발생하는 인터럽트이다.

- **CPU와 메모리 외부 장치와의 통신**에 필수적이고, CPU의 **작업 중단 없이** 즉시 처리가 필요함.
- **하드웨어 인터럽트 컨트롤러**가 인터럽트 요청을 관리하고 우선순위를 조정.

예시:

- **타이머 인터럽트**: 시스템의 주기적인 작업을 수행하기 위해 타이머에서 발생하는 인터럽트.
- **네트워크 카드 인터럽트**: 네트워크 패킷을 수신하면 해당 인터럽트를 발생시켜 데이터를 처리.

🔑 인터럽트와 폴링 비교

항목	인터럽트	폴링
동작	이벤트 발생 시 CPU에 알림	주기적으로 상태를 확인
CPU 사용	이벤트 발생 시만 사용	상태를 계속 확인하므로 CPU 리소스 소모
속도	이벤트 발생 즉시 처리	대기시간에 따라 느릴 수 있음
응답 시간	짧음 (즉시 처리 가능)	상대적으로 길어짐

📌 요약 정리

항목	설명
인터럽트	현재 실행 중인 작업을 중단하고 우선 작업을 처리하는 메커니즘
외부 인터럽트	외부 장치(키보드, 마우스, 타이머 등)에서 발생
내부 인터럽트	프로그램 실행 중 발생하는 예외나 오류
소프트웨어 인터럽트	프로그램에서 의도적으로 발생시키는 인터럽트
하드웨어 인터럽트	외부 하드웨어 장치에서 발생하는 인터럽트
응답 시간	인터럽트는 이벤트가 발생할 때 즉시 처리, 폴링은 주기적으로 상태를 확인

6.2 인터럽트 벡터 테이블

🧠 인터럽트 벡터 테이블이란?

인터럽트 벡터 테이블이란,
인터럽트 발생 시 해당 인터럽트에 대응하는 처리 루틴(ISR)의 주소를 저장한 테이블이다.

> 즉, 인터럽트 번호(또는 벡터 번호)가 들어오면,
> CPU는 이 벡터 번호를 인덱스로 삼아 **IVT에서 해당 ISR 주소를 읽고 실행**한다.

📦 구조 개념

```
      인터럽트 벡터 번호 (n)
              ↓
+----------------------------+
| Interrupt Vector Table     |
+----------------------------+
| [0] → Reset Handler Addr   |
| [1] → NMI Handler Addr     |
| [2] → HardFault Addr       |
| [3] → Timer Interrupt Addr |
| [4] → UART ISR Addr        |
| ...                        |
+----------------------------+
```

- 각 벡터는 4바이트 또는 8바이트로 구성 (ISA 및 플랫폼에 따라 다름)
- 테이블의 위치는 보통 **고정된 메모리 시작 주소**에 있음

🔄 동작 흐름

1. **인터럽트 발생**
2. **CPU가 인터럽트 벡터 번호 확인**
    - 예: 타이머 인터럽트 → 벡터 번호 14
3. **해당 벡터 번호의 ISR 주소를 IVT에서 참조**

4. **해당 주소로 분기(Jump)하여 ISR 실행**
5. **ISR 종료 후 복귀 → 원래 실행하던 코드로 복귀**

🧬 IVT의 메모리 위치 (시스템별)

시스템/ISA	IVT 위치	비고
x86 Real Mode	`0x0000:0000`	256개 × 4B = 1024B
x86 Protected Mode	IDT (Interrupt Descriptor Table) 사용	IDT는 좀 더 확장된 구조
ARM Cortex-M	`0x00000000` 또는 `0x20000000` (Vector Table Offset Register에 따라 변경 가능)	첫 번째 항목은 SP, 두 번째부터 ISR 주소
AVR(Atmel)	Flash 시작 주소 (`0x0000`)	각 인터럽트 벡터는 고정된 주소에 위치

🧠 ARM Cortex-M 예: 벡터 테이블 예시

```
__attribute__((section(".isr_vector")))
const void* VectorTable[] = {
    &_stack_top,            // 초기 스택 포인터
    Reset_Handler,          // Reset
    NMI_Handler,            // Non-Maskable Interrupt
    HardFault_Handler,      // HardFault
    Timer_IRQHandler,       // 예: SysTick
    UART_IRQHandler,        // 예: UART
    ...
};
```

- 이 테이블은 링커 스크립트에 따라 Flash 시작 주소에 위치
- `__attribute__((section))` 를 이용해 벡터 테이블을 특정 위치로 보내는 게 일반적

🔧 IDT (Interrupt Descriptor Table) - x86 Protected Mode

x86에서는 실주소가 아닌 **디스크립터 기반 벡터 테이블**을 사용한다.

필드	설명
Offset Low	ISR의 하위 주소
Selector	코드 세그먼트 선택자
Type/Attr	게이트 타입, DPL, P(유효 플래그)
Offset High	ISR의 상위 주소

```
struct IDTEntry {
    uint16_t offset_low;
    uint16_t selector;
    uint8_t  zero;
    uint8_t  type_attr;
    uint16_t offset_high;
};
```

> IDT는 `lidt` 명령어로 로드되고, 벡터 번호에 따라 접근됨.

🧩 인터럽트 우선순위와 IVT

- 벡터 번호가 작을수록 보통 **우선순위가 높다**
  (ex: ARM Cortex-M에서 벡터 1 = NMI → 마스크 불가 인터럽트)
- 하지만 **NVIC(ARM)** 같은 시스템에서는 **우선순위 레지스터를 통해 우선순위 직접 설정 가능**

🛠️ 실전 예: STM32 (Cortex-M)

```
void TIM2_IRQHandler(void) {
    if (TIM2->SR & TIM_SR_UIF) {
        TIM2->SR &= ~TIM_SR_UIF;
        // 타이머 오버플로우 처리
    }
}
```

> 이 함수의 이름은 벡터 테이블에 등록되어 `TIM2` 인터럽트가 발생하면 자동으로 호출됨.

📌 요약 정리

항목	설명
IVT	인터럽트 벡터 번호와 ISR 주소를 매핑하는 테이블
위치	보통 메모리 시작 주소, 시스템에 따라 다름
역할	인터럽트 발생 시 ISR로 분기하기 위한 주소 참조
시스템 예시	x86(Real/Protected), ARM Cortex, AVR 등
변경 가능	일부 시스템은 런타임에 벡터 테이블 위치를 바꿀 수 있음

6.3 마스킹 가능한 인터럽트

🧠 개념 요약

마스킹 가능한 인터럽트(Maskable Interrupt)란,
CPU가 명령 또는 제어 레지스터를 통해 '무시할 수 있는(마스킹할 수 있는)' 인터럽트를 말한다.

> CPU는 인터럽트를 잠시 끄고 특정 코드(예: 크리티컬 섹션)를 **중단 없이 실행**할 수 있어야 함.
> 이때 처리 우선순위가 낮은 인터럽트를 **마스킹(Disable)** 하는 기능이 필요하다.

🧩 마스킹 vs 비마스킹 인터럽트

구분	마스킹 가능 인터럽트	비마스킹 인터럽트
영어명	Maskable Interrupt (MI)	Non-Maskable Interrupt (NMI)
무시 가능 여부	✅ CPU에서 끌 수 있음	❌ CPU에서 끌 수 없음
사용 용도	일반 I/O, Timer 등	긴급 오류 처리 (메모리 오류, 전원 이상 등)
우선순위	낮음 또는 가변적	항상 최고 우선순위
명령어로 제어	가능 (`CLI`, `SEI`, `CPSID`, `CPSIE`)	불가능 (하드웨어적으로 고정됨)

🔧 마스킹 방식

✅ 1. 전역 인터럽트 플래그 비트 사용

- **x86**: `IF` (Interrupt Flag)
  - `CLI` → Clear Interrupt (인터럽트 금지)
  - `STI` → Set Interrupt (인터럽트 허용)
- **ARM Cortex-M**: `PRIMASK`, `BASEPRI`, `FAULTMASK`
  - `CPSID i` → 인터럽트 비활성화
  - `CPSIE i` → 인터럽트 활성화
- **AVR**: `I` 비트 (Status Register의 7번 비트)
  - `sei()` → 전체 인터럽트 활성화
  - `cli()` → 전체 인터럽트 비활성화

✅ 2. NVIC 기반 우선순위 마스킹 (ARM Cortex-M)

ARM Cortex-M에서는 **벡터별로 우선순위를 부여하고**,
특정 우선순위 이하의 인터럽트만 **마스킹 가능**하게 함.

- `BASEPRI` 레지스터에 특정 값을 설정
  → 해당 값보다 우선순위가 **낮은 인터럽트는 무시**,
  → 높은 인터럽트는 여전히 허용

🧪 언제 마스킹이 필요한가?

상황	이유
크리티컬 섹션 실행 중	중단되면 데이터 손상 발생 가능 (동기화 오류)
부트 로더, 초기화 코드	하드웨어 초기화 도중 인터럽트 허용은 위험
우선순위 제어	특정 작업 중 낮은 우선순위 인터럽트 무시 필요
디버깅/트랩 보호	인터럽트 처리 중 또 다른 인터럽트 진입 방지

💡 주의할 점

- **인터럽트를 너무 오랫동안 마스킹하면**:
  - **높은 응답성이 필요한 이벤트**(예: UART 수신, 타이머 오버플로우 등)를 놓칠 수 있음
  - 실시간성에 영향 → **RTOS나 실시간 제어 시스템**에서는 주의 필요

🧠 실전 예: AVR (ATmega328P) 코드

```
cli();            // 인터럽트 전역 비활성화
critical_code();  // 중요한 작업
sei();            // 다시 인터럽트 허용
```

🧠 실전 예: ARM Cortex-M (STM32 등)

```
__disable_irq();   // PRIMASK = 1 → 모든 마스크 가능한 인터럽트 비활성화
critical_code();
__enable_irq();    // PRIMASK = 0 → 인터럽트 재활성화
```

또는 `__set_BASEPRI(0x40);` 같은 함수로 **우선순위 기반 마스킹**도 가능

📌 요약 정리

항목	설명
마스킹 가능한 인터럽트	CPU가 비활성화할 수 있는 일반 인터럽트
마스킹 방법	인터럽트 플래그 제어 (IF, PRIMASK 등)
비마스킹 인터럽트	CPU에서도 절대 무시 못함 (하드웨어 오류 등)
사용 목적	중요한 코드 중단 방지, 인터럽트 간 우선순위 관리
실전 예	`CLI/STI`, `CPSID/CPSIE`, `__disable_irq/__enable_irq`

6.4 소프트웨어/하드웨어 인터럽트

🧠 인터럽트 분류 요약

인터럽트는 **"누가 발생시키느냐"**에 따라 아래 두 가지로 나뉜다:

종류	발생 주체	트리거 방식
하드웨어 인터럽트	외부 장치 (I/O, 타이머, 센서 등)	**하드웨어 신호 (핀 변화)**
소프트웨어 인터럽트	프로그램 코드	**명령어 또는 시스템 호출**

🔌 1. 하드웨어 인터럽트 (Hardware Interrupt)

✅ 정의

외부 하드웨어 장치에서 특정 이벤트가 발생하면, 해당 신호를 CPU에 전달하여
현재 코드 실행을 중단하고 인터럽트 서비스 루틴(ISR)을 호출하게 함.

✅ 예시

- **키보드 입력**: 키를 누르면 키보드 컨트롤러가 인터럽트 요청 → ISR 실행
- **타이머 오버플로우**: 일정 시간마다 타이머가 인터럽트를 발생시켜 주기적 작업 수행
- **UART 수신**: 데이터 수신 완료 → RX 인터럽트 발생
- **센서 상태 변화**: GPIO 핀 변화 감지 인터럽트

✅ 흐름 요약

```
[하드웨어 이벤트] → [인터럽트 요청 핀 (IRQ)] → [인터럽트 컨트롤러] → [CPU]
→ [ISR 실행] → [원래 작업 복귀]
```

💻 2. 소프트웨어 인터럽트 (Software Interrupt)

✅ 정의

명령어 또는 코드 상에서 명시적으로 인터럽트를 발생시켜,
운영체제나 특정 핸들러 루틴을 호출하는 방식.
특정 이벤트 없이 프로그래머가 의도적으로 발생시킴.

✅ 예시

상황	설명
시스템 호출(System Call)	유저 코드에서 OS 커널의 기능 요청
→ `int 0x80` (x86), `svc #0` (ARM) 등	
예외 처리	`div 0`, `invalid opcode` → 내부적으로 소프트웨어 인터럽트로 처리
디버깅 트랩	`int 3` (x86) → 디버깅 브레이크포인트
OS Trap	시스템 함수 진입 전에 트랩 설정 (예: trap handler)

✅ 흐름 요약

```
[소프트웨어 명령] → [CPU 트랩] → [인터럽트 벡터 참조] → [ISR 또는 커널 루틴 실행]
```

🔧 실전 코드 예시

🔹 x86 어셈블리에서 소프트웨어 인터럽트

```
mov eax, 1          ; 시스템 콜 번호 (exit)
mov ebx, 0          ; 리턴 코드
int 0x80            ; 소프트웨어 인터럽트로 커널 호출
```

🔹 ARM Cortex-M에서 소프트웨어 인터럽트

```
__asm__("svc #0");   // SVC(Supervisor Call) 명령 → OS 기능 요청
```

🧩 주요 차이점 비교표

항목	하드웨어 인터럽트	소프트웨어 인터럽트
발생 원인	외부 장치 (I/O, 센서, 타이머)	코드 내부 명령 또는 예외
목적	외부 이벤트 감지 및 반응	시스템 호출, 예외 처리, 디버깅
속도	실시간 (비동기)	명령어 실행 기반 (동기)
제어 가능성	불규칙적 (외부 의존)	프로그래머가 제어 가능
우선순위 처리	인터럽트 컨트롤러가 조정	소프트웨어 논리로 조정 가능
예시 명령어	없음 (하드웨어 트리거)	`int`, `svc`, `trap`, `brk` 등

🧠 둘의 공통점

- 둘 다 **인터럽트 벡터 테이블**을 통해 ISR(인터럽트 서비스 루틴)을 실행함
- 둘 다 **현재 작업 상태를 보존하고 → 인터럽트 처리 → 복귀**하는 흐름을 가짐

📌 요약 정리

항목	설명
하드웨어 인터럽트	외부 장치의 신호로 발생, 실시간 이벤트 처리
소프트웨어 인터럽트	명령어 또는 코드 상의 요청으로 발생, 시스템 호출 또는 예외 처리
공통점	CPU가 작업 중단 → ISR 진입 → 복귀
차이점	발생 시점, 트리거 방식, 목적의 차이

6.5 예외(Exception) 처리 메커니즘

🧠 예외(Exception)란?

예외(Exception)는 **CPU가 명령어를 실행하는 도중**
정상적이지 않은 상황(오류, 특수 요청 등)을 감지했을 때 발생하는
비동기적이면서도 내부적인 인터럽트라고 할 수 있다.

> 예외는 프로그램 실행 중 발생하며, CPU가 더 이상 정상적인 명령 실행을 지속할 수 없는 경우
> **즉시 현재 흐름을 중단하고 예외 처리 루틴으로 분기**하게 된다.

📌 예외 vs 인터럽트 차이점

항목	예외 (Exception)	인터럽트 (Interrupt)
발생 시점	명령어 실행 중	명령어 사이
발생 원인	CPU 내부	외부 또는 타이머/IO
예시	0으로 나누기, 페이지 폴트, 불법 명령	키보드 입력, 타이머, UART 수신
제어 여부	일부는 소프트웨어로 발생 (예: `trap`)	하드웨어 이벤트에 의해 발생
일반 목적	오류 처리, OS 요청 (시스템 콜)	비동기 I/O 응답, 정기 작업 처리

🧩 예외의 발생 조건

1. **잘못된 명령어 실행**
   - 예: 존재하지 않는 명령어를 실행하려 할 때
2. **권한 위반**
   - 사용자 모드에서 커널 영역 접근
3. **메모리 접근 오류**
   - 페이지 폴트(Page Fault)
4. **산술 오류**
   - 0으로 나누기, 오버플로우
5. **시스템 호출**
   - 소프트웨어에서 `int`, `trap`, `svc` 명령으로 커널 진입

🧠 예외의 종류

✅ 1. Fault (정정 가능한 오류)

- 예외가 발생한 **명령어는 실행되지 않았고**, 복구 후 **재실행 가능**
- 예: 페이지 폴트 → 운영체제가 메모리 페이지를 올린 후 명령 재시도

✅ 2. Trap (정상적 종료 후 알림)

- 명령어는 **정상적으로 실행된 후**, 예외가 발생
- 예: 디버거의 브레이크포인트, 시스템 콜

✅ 3. Abort (치명적 오류)

- **회복 불가능한 오류**
- 예: 버스 오류(Bus Fault), 명령어 fetch 실패 등

🔧 예외 처리 흐름

1. 예외 발생 (ex. divide by zero, trap)
2. CPU는 현재 명령어 실행을 중단
3. **스택에 현재 상태(PC, PSR, 레지스터 등) 저장**
4. 예외 벡터 테이블에서 해당 예외의 핸들러 주소 참조
5. 해당 **예외 핸들러(Handler/ISR) 실행**
6. 예외 처리 완료 후, `IRET` 또는 `ERET` 등을 통해 복귀

📂 시스템별 예외 처리 방식

🔹 x86 (IA-32)

- **IDT(Interrupt Descriptor Table)** 사용
- 예외마다 고유 벡터 번호 (0~31)
- 예:
  - `#DE` (Divide Error) → 벡터 0
  - `#PF` (Page Fault) → 벡터 14
  - `#GP` (General Protection) → 벡터 13

> 예외 발생 시 CPU는 IDT에서 해당 벡터를 찾고, 그 위치의 핸들러로 점프

🔹 ARM Cortex-M

- 벡터 테이블 구조로 예외 관리
- 우선순위: Reset > NMI > HardFault > MemManage > BusFault > UsageFault ...

예외	설명
Reset	전원 인가 시 진입
NMI	마스킹 불가능한 인터럽트
HardFault	치명적인 오류
MemManage	메모리 접근 오류
BusFault	버스 통신 중 오류

예외	설명
UsageFault	잘못된 명령어 사용 등

> 예: `SCB->SHCSR`로 예외 활성화/비활성화 설정 가능

🛠️ 시스템 호출 = 소프트웨어 예외의 한 형태

- 사용자가 `int 0x80` (x86), `svc #0` (ARM) 명령 실행
- 커널 모드로 진입하여 **시스템 콜 처리 루틴 실행**
- trap handler는 커널 진입점 역할 수행

🧪 실전 예: x86에서 예외 핸들러 예

```
section .text
global divide_by_zero
divide_by_zero:
    ; 오류 처리 코드
    iret    ; 복귀
```

→ IDT의 벡터 0번이 이 함수 주소를 가리키도록 설정

📌 요약 정리

항목	설명
예외(Exception)	명령 실행 중 발생하는 오류 또는 이벤트
종류	Fault (재실행), Trap (종료 후 처리), Abort (복구 불가)
처리 흐름	상태 저장 → 벡터 테이블 참조 → 핸들러 실행 → 복귀
시스템 예	x86: IDT 기반, ARM: 벡터 테이블 기반
활용 예	페이지 폴트, 시스템 콜, 디버깅, 접근 위반 등

6.6 인터럽트 우선순위 처리

> Interrupt Priority and Nesting

🧠 왜 우선순위가 필요한가?

시스템에서는 여러 인터럽트가 **동시에** 발생할 수 있기 때문에
누구를 먼저 처리할지 판단하는 기준이 필요하다.
예를 들어:

> UART 수신 vs 센서 데이터 vs 긴급 시스템 타이머
> → 모두 동시에 발생했을 때, **타이머처럼 중요한 작업을 먼저 처리**해야 함

🔺 우선순위 처리의 목적

목적	설명
실시간성 보장	중요한 작업이 즉시 처리될 수 있도록 함
시스템 안정성	저우선 인터럽트가 고우선 처리를 방해하지 않게 함
중첩 인터럽트 지원	고우선 인터럽트는 저우선 ISR 중에도 처리 가능

🔩 기본 동작 원리

1. 여러 인터럽트가 동시에 발생
2. **인터럽트 컨트롤러**가 우선순위를 평가
3. 가장 높은 우선순위의 ISR부터 실행
4. 실행 중 더 높은 우선순위 인터럽트가 발생하면:
    - 현재 ISR 상태 저장 → 고우선 ISR 실행 → 복귀 후 재개

> 이걸 **중첩 인터럽트(Nested Interrupt)** 라고 해

🧱 우선순위 처리 방식 종류

✅ 1. 고정 우선순위 (Fixed Priority)

- 각 인터럽트 소스에 고정된 우선순위를 부여
- 일반적인 작은 MCU에서 사용 (예: AVR)

✅ 2. 프로그래머블 우선순위 (Programmable Priority)

- 소프트웨어로 우선순위 설정 가능
- ARM Cortex-M의 NVIC (Nested Vectored Interrupt Controller) 구조가 대표적

✅ 3. 동적 우선순위 (Dynamic Priority)

- 상황에 따라 우선순위를 실시간으로 조정
- RTOS 기반 시스템이나 고급 SoC에서 사용

🧩 시스템별 우선순위 구조

🔹 AVR (예: ATmega328P)

- 우선순위 없음 → **위치 기준 처리**
- 낮은 벡터 번호가 우선순위 높음
- 중첩 인터럽트는 기본적으로 불가 (`sei()` 로 명시적 허용해야 함)

🔹 ARM Cortex-M (예: STM32)

- **NVIC** 구조 기반
- 각 인터럽트에 **0~255 단계의 우선순위** 부여 가능
- 우선순위는 **Preemption Priority (선점)** + **Sub Priority (같은 레벨 내 순서)**로 구성

```
HAL_NVIC_SetPriority(USART1_IRQn, 1, 0); // USART1 인터럽트 우선순위 설정
HAL_NVIC_EnableIRQ(USART1_IRQn);
```

- 숫자가 **작을수록 우선순위가 높음** (`0` > `1` > `2`...)

🔹 x86 구조 (8259 PIC, APIC 등)

- 기본적으로 **PIC** 또는 **APIC**에서 우선순위 처리
- x86은 인터럽트 우선순위를 **벡터 번호**에 따라 결정
- 운영체제가 **IDT(Index Descriptor Table)** 를 통해 우선순위 재구성 가능

🧠 중첩 인터럽트(Nested Interrupt)

> 현재 ISR이 실행 중일 때, 더 높은 우선순위 인터럽트가 발생하면
> **현재 ISR의 상태를 스택에 저장하고**, 고우선 ISR을 먼저 처리한 후
> 다시 돌아와 원래 ISR을 계속 실행하는 구조

필요 조건:

조건	설명
인터럽트 중 인터럽트 허용	ISR 내부에서 인터럽트 전역 허용 (`sei()`, `__enable_irq()` 등)
컨트롤러가 우선순위 비교 가능	NVIC, APIC 등
스택 여유 충분	ISR 중 ISR을 실행하려면 상태 보존 필요함

🧪 실전 예제 (ARM Cortex-M)

```
void USART1_IRQHandler(void) {
    HAL_NVIC_SetPriority(TIM2_IRQn, 0, 0);  // TIM2가 더 높은 우선순위
    HAL_NVIC_SetPriority(USART1_IRQn, 1, 0);
}
```

→ `USART1_IRQHandler()` 실행 중이라도 `TIM2_IRQHandler()`는 선점 가능

📊 우선순위 처리 흐름 예시

```
[UART 수신 인터럽트] → 우선순위 2
[타이머 인터럽트]   → 우선순위 1 (더 높음)

→ 타이머 ISR 실행 중 UART 인터럽트 발생 → 무시됨
→ UART ISR 실행 중 타이머 인터럽트 발생 → 중첩 실행 가능
```

📌 요약 정리

항목	설명
우선순위 필요성	동시에 발생하는 인터럽트를 정렬, 중요한 것부터 처리
고정 우선순위	벡터 위치로 결정 (AVR 등)
프로그래머블 우선순위	NVIC, APIC 기반 시스템 (ARM, x86)
우선순위 설정법	ARM: `NVIC_SetPriority()`, x86: IDT 기반
중첩 인터럽트	ISR 실행 중 더 높은 인터럽트가 발생하면 실행 가능