9. 인터럽트 / 예외 / Fault Handling

9.1 예외 (Exceptions)

- ## HardFault / BusFault / UsageFault 분석

1. 개요

Cortex-M 계열(예: STM32F 시리즈)에서 발생하는 **Fault Exception**은
시스템 오류나 잘못된 접근을 감지하기 위한 **CPU 보호 메커니즘**이다.
이 예외들은 모두 **"Fault Handler"** 에 의해 처리되며,
각각의 원인은 다음과 같이 구분된다.

Fault 종류	주요 원인	우선순위
HardFault	복구 불가한 심각한 오류 (다른 Fault로부터 승격된 경우 포함)	가장 높음
BusFault	잘못된 메모리 접근 (Bus 오류)	중간
UsageFault	잘못된 명령어, 정렬 오류, 0으로 나눗셈 등	낮음
MemManage Fault	보호 영역 위반 (MPU 관련)	낮음

이러한 Fault들은 **SCB(System Control Block)** 내부의 전용 상태 레지스터를 통해 진단할 수 있다.

---

2. Fault 발생 흐름

1. 프로그램 실행 중 예외 상황 발생
2. CPU가 Fault의 원인을 판별
3. 해당 Fault 활성화 → NVIC를 통해 예외 진입
4. Fault Handler에서 `stack frame` 저장 (PC, LR, PSR 등)
5. 복귀 불가 시 시스템 리셋 또는 무한 루프 진입

---

3. 주요 Fault 상세 분석

3.1 HardFault

개요

HardFault는 **복구 불가능한 예외**이며,
다른 Fault가 처리 불가능할 때 자동으로 승격되어 발생한다.

대표 원인

- 잘못된 함수 포인터 호출
- NULL 포인터 접근
- 스택 오버플로우
- BusFault / UsageFault 발생 후 핸들러 비활성 상태
- 잘못된 벡터 테이블 또는 인터럽트 반환 주소

진단 방법

```
void HardFault_Handler(void)
{
    __asm volatile
    (
        "TST lr, #4        \n"
        "ITE EQ            \n"
        "MRSEQ r0, MSP     \n"
        "MRSNE r0, PSP     \n"
        "B HardFault_HandlerC \n"
    );
}

void HardFault_HandlerC(uint32_t *hardfault_args)
{
    uint32_t stacked_pc = hardfault_args[6];
    uint32_t stacked_lr = hardfault_args[5];
    printf("HardFault at PC=0x%08lx LR=0x%08lx\r\n", stacked_pc, stacked_lr);

    uint32_t cfsr = SCB->CFSR;
    uint32_t hfsr = SCB->HFSR;
    uint32_t mmfar = SCB->MMFAR;
    uint32_t bfar = SCB->BFAR;

    printf("CFSR=0x%08lx, HFSR=0x%08lx, MMFAR=0x%08lx, BFAR=0x%08lx\r\n",
           cfsr, hfsr, mmfar, bfar);

    while (1);
}
```

참고 레지스터

레지스터	의미
`SCB->HFSR`	HardFault 상태
`SCB->CFSR`	세부 Fault 원인 (BusFault/UsageFault 통합 정보 포함)
`SCB->BFAR`	BusFault 접근 주소
`SCB->MMFAR`	MemManage Fault 주소

3.2 BusFault

개요

BusFault는 **잘못된 메모리 접근 시** 발생한다.
CPU가 유효하지 않은 주소를 읽거나 쓰려고 시도하면 BusFault가 발생하며,
보통 Flash, SRAM, 주변장치 접근 오류로 이어진다.

대표 원인

- 존재하지 않는 주소 접근 (예: 0xFFFFFFFF)
- 정렬되지 않은 32bit 접근
- 주변장치 Clock Enable 미설정 상태에서 접근
- DMA/버스 충돌로 인한 오류

진단 레지스터 (`SCB->CFSR` 의 [15:8] 영역)

비트	이름	설명
15	`BFARVALID`	`BFAR` 유효 여부
13	`STKERR`	스택 저장 시 오류
12	`UNSTKERR`	스택 복원 시 오류
11	`IMPRECISERR`	지연된 Bus 오류 (정확한 PC 없음)
10	`PRECISERR`	정확한 주소 Bus 오류
8	`IBUSERR`	명령어 Fetch 중 오류

디버깅 팁

- `BFARVALID` 가 1이면, `SCB->BFAR` 에 오류 주소 존재.
- `PRECISERR` 이면 Fault 시점 PC가 정확히 유효.
- `IMPRECISERR` 이면 DMA/버퍼링된 접근이 원인일 가능성 큼.

3.3 UsageFault

개요

UsageFault는 **명령어 실행 단계에서 잘못된 연산이 수행될 때** 발생한다.
CPU 명령어 수준 오류, 정렬 문제, 0 나눗셈 등 프로그램 논리 오류가 대부분이다.

대표 원인

- 정의되지 않은 명령어 실행
- Thumb/ARM 상태 전환 오류
- 0으로 나누기

- 정렬되지 않은 메모리 접근
- 잘못된 Exception Return

진단 레지스터 (`SCB->CFSR`의 [31:16] 영역)

비트	이름	설명
18	`DIVBYZERO`	0으로 나누기
17	`UNALIGNED`	정렬되지 않은 접근
16	`UNDEFINSTR`	정의되지 않은 명령어
25	`NOCP`	FPU 명령 사용 중, FPU 미활성화
24	`INVPC`	잘못된 PC 복귀
23	`INVSTATE`	잘못된 실행 상태 전환

---

4. Fault 진단 절차

1. **CFSR, HFSR, BFAR, MMFAR 레지스터 확인**

```
printf("CFSR=0x%08lx, HFSR=0x%08lx, BFAR=0x%08lx\r\n",
       SCB->CFSR, SCB->HFSR, SCB->BFAR);
```

2. **PC, LR 스택 값 추출 → Fault 발생 위치 식별**
   - HardFault 핸들러 내 스택 프레임에서 PC 추출 후 디버거로 주소 매핑.
3. **문제 코드 분석**
   - 잘못된 포인터, 배열 인덱스 초과, 주변장치 초기화 누락 점검.
4. **중단점(Breakpoint) 삽입 후 재현 테스트**
   - Fault 발생 직전 명령어 추적.

---

5. 예방 및 디버깅 전략

항목	권장 설정
Fault 활성화	`SCB->SHCSR
Stack 보호	MPU로 Stack 영역 보호 설정
HardFault 로그	Fault Context를 UART로 출력하여 문제 원인 기록
Assert 사용	HAL_ASSERT / configASSERT로 잘못된 인자 조기 감지
Watchdog 병행	Fault 발생 시 자동 리셋 및 로그 저장

---

6. 결론

HardFault, BusFault, UsageFault는
STM32 시스템의 **치명적 오류 원인 추적의 핵심 도구**이다.

- HardFault는 “최종 방어선”
- BusFault는 “잘못된 메모리 접근”
- UsageFault는 “명령어/연산 오류”

각 Fault Handler에 **CFSR 분석 루틴과 스택 정보 출력 기능**을 추가하면,
실제 원인을 빠르게 추적할 수 있다.

이를 통해 디버깅 효율을 극대화하고, 안정적인 펌웨어 동작을 보장할 수 있다.

- # Stack Frame 복구 (PC, LR, PSR)

1. 개요

Cortex-M 코어는 **예외(Interrupt/Fault) 발생 시 자동으로 CPU 레지스터 상태를 스택에 저장**한다.
이 과정에서 생성되는 데이터 집합을 **Stack Frame**이라 하며,
Fault 발생 당시의 **프로그램 카운터(PC)**, **링크 레지스터(LR)**, **프로그램 상태 레지스터(PSR)** 값을 복원하면
문제 발생 지점을 정확히 추적할 수 있다.

Stack Frame 복구는 HardFault, BusFault, UsageFault 등
모든 예외 분석의 기본 절차로 사용된다.

---

2. Stack Frame 구조

예외가 발생하면, Cortex-M은 **자동으로 8개의 워드(32bit × 8 = 32bytes)** 를 스택에 푸시한다.

순서	레지스터	설명
1	R0	첫 번째 함수 인자 / 일반 레지스터
2	R1	두 번째 함수 인자
3	R2	세 번째 함수 인자
4	R3	네 번째 함수 인자
5	R12	서브루틴 내 임시 변수
6	LR (R14)	호출 복귀 주소
7	PC (R15)	예외 발생 시 실행 중이던 명령어 주소
8	xPSR	프로그램 상태 (Condition, Thumb Bit 등 포함)

> 이 구조는 ARMv7-M 아키텍처(Cortex-M3/M4)에서 동일하게 적용된다.
> 스택 정렬(8-byte aligned)이 보장되지 않으면 Fault가 중첩될 수 있다.

---

3. 스택 선택 (MSP / PSP)

Cortex-M 코어는 **두 개의 스택 포인터**를 사용한다.

스택 종류	용도	예외 진입 시 사용 조건
MSP (Main Stack Pointer)	예외 처리, 초기 진입 시 기본 스택	대부분의 예외에서 기본 사용
PSP (Process Stack Pointer)	Thread 모드에서 사용	FreeRTOS 등 RTOS 환경에서 각 Task 전용

Fault Handler 진입 시, **LR의 2비트(비트 2)** 값으로
현재 스택 포인터가 MSP인지 PSP인지 구분할 수 있다.

```
TST lr, #4        ; LR의 2비트 검사
ITE EQ
MRSEQ r0, MSP     ; MSP 사용 시
MRSNE r0, PSP     ; PSP 사용 시
```

---

4. Stack Frame 복원 절차

Fault 핸들러 내부에서 현재 스택을 복원하려면,
아래 절차에 따라 스택의 8개 워드 값을 읽어 분석한다.

```
void HardFault_Handler(void)
{
    __asm volatile
    (
        "TST lr, #4        \n"
        "ITE EQ            \n"
        "MRSEQ r0, MSP     \n"
        "MRSNE r0, PSP     \n"
        "B HardFault_Decode \n"
    );
}

void HardFault_Decode(uint32_t *stack_addr)
{
    uint32_t r0  = stack_addr[0];
    uint32_t r1  = stack_addr[1];
    uint32_t r2  = stack_addr[2];
    uint32_t r3  = stack_addr[3];
    uint32_t r12 = stack_addr[4];
    uint32_t lr  = stack_addr[5];
    uint32_t pc  = stack_addr[6];
    uint32_t psr = stack_addr[7];

    printf("R0 = 0x%08lx\nR1 = 0x%08lx\nR2 = 0x%08lx\nR3 = 0x%08lx\n", r0, r1, r2,
r3);
```

```
    printf("R12= 0x%08lx\nLR = 0x%08lx\nPC = 0x%08lx\nPSR= 0x%08lx\n", r12, lr, pc, psr);
}
```

이 출력값을 통해 Fault가 발생한 **정확한 명령어 주소(PC)** 와
예외 직전 복귀 주소(LR) 를 복원할 수 있다.

---

5. PSR (Program Status Register) 분석

xPSR은 CPU의 실행 상태를 나타내며,
특히 **T 비트(Thumb Bit, Bit[24])** 가 반드시 `1`이어야 한다.
이 비트가 0이면 ARM 명령어로 오인되어 HardFault가 발생한다.

비트	이름	설명
31	N	Negative flag
30	Z	Zero flag
29	C	Carry flag
28	V	Overflow flag
24	T	Thumb 상태 비트 (반드시 1이어야 함)
8~0	ISR 번호	현재 실행 중인 예외 번호

예시:

```
xPSR = 0x61000000 → Thumb 모드 정상
xPSR = 0x21000000 → Thumb 비트 손실 → HardFault 가능
```

---

6. PC 및 LR 추적

(1) PC (Program Counter)

- Fault 발생 당시 실행 중이던 명령어 주소.
- 디버거 또는 `.map` 파일을 사용해 해당 주소를 C 코드 라인으로 역추적 가능.

```
arm-none-eabi-addr2line -e project.elf 0x08001234
```

(2) LR (Link Register)

- 예외 발생 직전의 복귀 주소 저장.
- `0xFFFFFFF9`, `0xFFFFFFFD` 등 특별한 값은
  Exception Return Code 를 의미한다.

LR 값	의미
`0xFFFFFFF9`	MSP에서 복귀 (Handler 모드 → Thread 모드)
`0xFFFFFFFD`	PSP에서 복귀 (Thread 모드 유지)
`0xFFFFFFE1`	FPU 컨텍스트 포함 복귀

---

7. 실전 Fault 분석 예시

UART 출력 결과:

```
HardFault!
PC  = 0x08001234
LR  = 0x08001029
PSR = 0x61000000
CFSR= 0x00008200
HFSR= 0x40000000
```

분석:

- PC = 0x08001234 → 코드 매핑 결과, `memcpy()` 내부 포인터 접근 시점.
- CFSR = 0x00008200 → `PRECISERR` (정확한 버스 오류)
- 결론: 잘못된 포인터 접근으로 인한 BusFault → HardFault 승격.

---

8. 결론

Stack Frame 복구는 HardFault 분석의 핵심이며,
PC, LR, PSR 값을 추출함으로써 **Fault 발생 코드 위치**와 **실행 컨텍스트**를 명확히 식별할 수 있다.

- PC: 오류 발생 지점
- LR: 복귀 경로 또는 Exception Return 코드
- PSR: CPU 상태 및 Thumb 모드 여부

이 정보를 기반으로 소스 코드 매핑 및 재현 테스트를 수행하면
원인 분석과 디버깅 시간을 획기적으로 단축할 수 있다.

• 디버깅 시 Fault 추적 방법

1. 개요

Fault 발생 시 시스템은 예외 벡터를 통해 해당 **Fault Handler**로 진입한다.
이 시점에서 단순히 MCU가 “멈췄다”는 현상만 관찰되는 경우가 많지만,
스택 프레임과 시스템 제어 블록(SCB) 레지스터를 분석하면
실제 오류 발생 지점과 원인을 정밀하게 역추적할 수 있다.

효율적인 Fault 디버깅은 다음 세 단계를 따른다.

1. **Fault 상태 레지스터 확인**
2. **Stack Frame 복구 (PC, LR, PSR 등)**

3. **코드 매핑 및 원인 분석**

---

2. Fault 분석에 필요한 주요 레지스터

모든 Fault 상태 정보는 **System Control Block (SCB)** 내부의 다음 레지스터에 저장된다.

레지스터	설명
`SCB->CFSR`	Configurable Fault Status Register — Usage, Bus, MemManage Fault 세부 정보
`SCB->HFSR`	HardFault Status Register — HardFault 발생 원인
`SCB->BFAR`	BusFault Address Register — BusFault 발생 주소
`SCB->MMFAR`	MemManage Fault Address Register — 메모리 보호 위반 주소
`SCB->SHCSR`	System Handler Control and State Register — Fault 활성화 상태 제어

> 디버깅 시 이들 값을 UART로 출력하거나 IDE Watch 창에서 확인하면
> Fault 원인 파악이 크게 단축된다.

---

3. Fault 핸들러 내 분석 루틴

다음 코드는 Fault 발생 시 스택 프레임과 SCB 레지스터를 함께 출력하는 예제다.

```
void HardFault_Handler(void)
{
    __asm volatile
    (
        "TST lr, #4        \n"
        "ITE EQ            \n"
        "MRSEQ r0, MSP     \n"
        "MRSNE r0, PSP     \n"
        "B HardFault_Trace \n"
    );
}

void HardFault_Trace(uint32_t *stack)
{
    uint32_t pc   = stack[6];
    uint32_t lr   = stack[5];
    uint32_t psr  = stack[7];

    uint32_t cfsr = SCB->CFSR;
    uint32_t hfsr = SCB->HFSR;
    uint32_t bfar = SCB->BFAR;
    uint32_t mmfar= SCB->MMFAR;

    printf("\r\n===== HardFault Detected =====\r\n");
    printf("PC  = 0x%08lx\r\n", pc);
    printf("LR  = 0x%08lx\r\n", lr);
```

```
    printf("PSR = 0x%08lx\r\n", psr);
    printf("CFSR= 0x%08lx\r\n", cfsr);
    printf("HFSR= 0x%08lx\r\n", hfsr);
    printf("BFAR= 0x%08lx\r\n", bfar);
    printf("MMFAR=0x%08lx\r\n", mmfar);
    printf("==============================\r\n");

    while (1);
}
```

이 출력만으로도 Fault 발생 위치(PC), 예외 원인(CFSR 비트), 잘못된 접근 주소(BFAR)를 직관적으로 확인할 수 있다.

---

4. Fault 로그 해석 예시

출력 예:

```
===== HardFault Detected =====
PC  = 0x080045F8
LR  = 0x0800412D
PSR = 0x21000000
CFSR= 0x00008200
HFSR= 0x40000000
BFAR= 0x20001004
MMFAR=0x00000000
==============================
```

해석

- **CFSR=0x00008200 → PRECISERR (정확한 BusFault)**
- **BFAR=0x20001004 → 잘못된 메모리 접근 주소**
- **PC=0x080045F8 → Fault가 발생한 명령어 주소**

해당 PC 주소를 `.elf` 파일과 매핑하여 정확한 C 코드 라인을 식별할 수 있다.

```
arm-none-eabi-addr2line -e project.elf 0x080045F8
```

결과 예:

```
Src/main.c: line 152 → *(uint32_t*)0x20001004 = 0x1234;
```

→ 실제 원인은 초기화되지 않은 포인터 접근으로 인한 BusFault.

---

5. IDE를 활용한 Fault 추적

(1) STM32CubeIDE Debug 모드

- **Run → Debug As → STM32 Cortex-M C/C++ Application**
- Fault 발생 시 “HardFault_Handler()”로 자동 진입
- “Registers” 탭에서 SCB, PC, LR 값 실시간 확인 가능

(2) Call Stack + Disassembly 뷰

- “PC” 레지스터를 더블클릭 → Fault 발생 명령어 확인
- “Call Stack”으로 Fault 직전 호출 함수 추적 가능

(3) Watch 창 등록

- `SCB->CFSR`, `SCB->HFSR`, `SCB->BFAR` 추가
- Fault 발생 시 레지스터 변화 즉시 모니터링 가능

---

6. 일반적인 Fault 원인별 추적 요령

Fault 유형	주요 원인	추적 포인트
HardFault	NULL 포인터, 잘못된 인터럽트 반환	PC / LR 값 확인 후 소스 매핑
BusFault	존재하지 않는 주소 접근, 클럭 미설정	BFAR, PRECISERR 비트
UsageFault	잘못된 명령어, 0 나누기, 정렬 오류	CFSR 상위 16비트
MemManage Fault	MPU 보호 영역 접근	MMFAR 주소
Stack Overflow	FreeRTOS Task 스택 초과	LR = 0xFFFFFFFD 형태, RTOS 설정 점검

---

7. 실전 디버깅 전략

1. **CFSR 해석**
   - CFSR 비트를 세분화하여 원인 판별 (PRECISERR, DIVBYZERO 등)
2. **PC 주소 매핑**
   - `addr2line` 또는 IDE에서 정확한 코드 위치 확인
3. **문제 함수 역추적**
   - Call Stack 또는 RTOS Task Context 확인
4. **조건부 재현**
   - 동일 입력 시 반복 Fault 발생 확인
5. **보호 루틴 삽입**
   - `assert()`, NULL 체크, 범위 검증 코드 추가

---

8. Fault 예방을 위한 사전 설정

항목	코드 예시
모든 Fault 활성화	`SCB->SHCSR
0 나누기 감지	`SCB->CCR
BusFault 활성화	`SCB->SHCSR
Stack 오버플로 검사	FreeRTOS `configCHECK_FOR_STACK_OVERFLOW` 사용
MPU 보호	Stack 및 Peripheral 영역 보호 설정

---

9. 결론

Fault 디버깅은 단순한 "시스템 멈춤"을
PC/LR/PSR/SCB 레지스터 기반의 원인 분석 단계로 전환하는 과정이다.

- Stack Frame 복구로 예외 당시의 상태를 재현하고,
- SCB 레지스터로 구체적인 Fault 원인을 파악하며,
- 코드 매핑(`addr2line` 또는 CubeIDE Debug)으로 실제 소스 위치를 식별한다.

이 절차를 표준화하면, **하드웨어 오류·펌웨어 버그·RTOS 스택 초과** 등
복잡한 문제의 원인을 신속하게 추적하고 해결할 수 있다.

9.2 인터럽트 처리

- # NVIC 우선순위 및 Enable

1. 개요

NVIC (Nested Vectored Interrupt Controller) 는 Cortex-M 코어의 핵심 인터럽트 제어기다.
모든 외부 및 시스템 예외(IRQ)는 NVIC를 통해 관리되며,
인터럽트 벡터 테이블, **우선순위(Preemption/Subpriority)**, **Enable/Disable** 상태로 제어된다.

STM32의 HAL 및 CMSIS 계층에서는 `HAL_NVIC_...()` 혹은 `NVIC_...()` API를 통해 설정할 수 있다.

---

2. 인터럽트 벡터 구조

Cortex-M은 다음 구조로 인터럽트를 식별한다.

구분	번호	예시
시스템 예외	0 ~ 15	Reset, NMI, HardFault, SysTick 등
외부 인터럽트 (IRQ)	16 ~ n	EXTI0, TIM2, USART1, DMA 등

각 IRQ는 고유한 벡터 엔트리를 가지며, `startup_stm32fxxx.s` 파일에 선언되어 있다.

---

3. 우선순위 구조

NVIC의 우선순위는 **중첩 허용 수준**을 결정한다.
Cortex-M3/M4/M7 기준, 최대 256단계(8bit)까지 지원하나,
STM32는 **4비트만 사용 (0~15)** 하는 경우가 많다.

- **Preemption Priority (선점 우선순위)**:
  인터럽트 중첩 여부를 결정 (낮을수록 우선순위 높음)
- **Sub Priority (서브 우선순위)**:
  동일한 Preemption 내에서 발생 순서 조정

---

4. 우선순위 그룹 (Priority Grouping)

`SCB->AIRCR` 레지스터의 `PRIGROUP` 필드로 그룹 비트를 정의한다.

Group 설정	Preemption	Sub	비트 예시
Group 0	0	4	전부 SubPriority
Group 1	1	3	1비트 Preemption
Group 2	2	2	균형
Group 3	3	1	Preemption 우선
Group 4	4	0	전부 Preemption

HAL에서는 다음 API로 설정 가능하다.

```
HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_2);
```

---

5. 인터럽트 우선순위 설정

특정 IRQ의 Preemption/SubPriority를 지정한다.

```
HAL_NVIC_SetPriority(IRQn_Type IRQn, uint32_t PreemptPriority, uint32_t
SubPriority);
```

예시:

```
// TIM2 Interrupt: 높은 우선순위
HAL_NVIC_SetPriority(TIM2_IRQn, 0, 0);

// USART1 Interrupt: 낮은 우선순위
HAL_NVIC_SetPriority(USART1_IRQn, 1, 0);
```

---

6. 인터럽트 Enable / Disable

Enable

```
HAL_NVIC_EnableIRQ(IRQn_Type IRQn);
```

Disable

```
HAL_NVIC_DisableIRQ(IRQn_Type IRQn);
```

예시:

```
HAL_NVIC_EnableIRQ(EXTI0_IRQn);  // EXTI0 인터럽트 활성화
HAL_NVIC_DisableIRQ(USART2_IRQn); // USART2 인터럽트 비활성화
```

---

7. 우선순위 변경 시 주의사항

1. **PriorityGroup 변경은 Reset 이후 적용됨**
   → 실행 중 변경 시 예상치 못한 동작 가능
2. **FreeRTOS 사용 시**
   `configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY` 보다 높은 IRQ에서는
   RTOS API 호출 금지
3. **Nested Interrupts 활성화 시**
   높은 우선순위 인터럽트가 낮은 인터럽트를 중단시킬 수 있음

---

8. 실전 예: UART + Timer 인터럽트

```
void MX_NVIC_Init(void)
{
    HAL_NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_2);

    // UART1 Rx Interrupt (Preemption=1, Sub=0)
    HAL_NVIC_SetPriority(USART1_IRQn, 1, 0);
    HAL_NVIC_EnableIRQ(USART1_IRQn);

    // TIM3 Interrupt (Preemption=0, Sub=0)
    HAL_NVIC_SetPriority(TIM3_IRQn, 0, 0);
    HAL_NVIC_EnableIRQ(TIM3_IRQn);
}
```

설명:

- TIM3 인터럽트는 Preemption=0 → UART1보다 항상 우선 실행됨.
- UART1은 Preemption=1 → TIM3 실행 중에는 대기함.

---

9. 디버깅 시 확인

- `NVIC->ISER[n]` : Enable 상태 비트
- `NVIC->ICER[n]` : Disable 상태
- `NVIC->IP[x]` : 실제 적용된 우선순위 값

예시 (Memory Watch):

```
NVIC->ISER[0] = 0x00004004  // TIM2, USART1 활성화
NVIC->IP[28]  = 0x20        // USART1 Priority = 0x20
```

---

10. 결론

NVIC 설정은 실시간성 제어의 핵심이다.

- **우선순위 그룹**을 명확히 구분하고,
- **Preemption/Sub Priority**를 체계적으로 배치하며,
- **Enable/Disable 제어**로 인터럽트 타이밍을 조정한다.

이 구성을 정확히 이해하면, RTOS Task, Timer, DMA, UART 인터럽트 간의 충돌 없이
안정적이고 예측 가능한 동작 시퀀스를 설계할 수 있다.

HAL_IRQ_Handler 호출 체계

1. 개요

STM32 HAL 드라이버는 **CMSIS 예외 벡터 구조**를 기반으로,
하드웨어 인터럽트 → HAL IRQ Handler → 사용자 콜백 순으로 호출되는 계층적 구조를 가진다.

이 체계는 하드웨어 독립성과 코드 유지보수성을 확보하기 위해 설계되었으며,
각 주변장치(Peripheral)는 고유의 HAL IRQ Handler 루틴을 제공한다.

---

2. 인터럽트 호출 흐름 전체 구조

다음은 실제 인터럽트 발생 시 제어 흐름이다.

```
[하드웨어 인터럽트 발생]
        ↓
NVIC 벡터 테이블 진입
        ↓
(1) XXX_IRQHandler()   ← startup_stm32fxxx.s에 정의됨
        ↓
(2) HAL_XXX_IRQHandler()  ← HAL 드라이버 내부 구현
        ↓
(3) 내부 상태 처리 및 콜백 호출
        ↓
(4) User Callback (예: HAL_UART_RxCpltCallback)
```

---

3. 예시: USART 인터럽트 처리 흐름

(1) NVIC 벡터 엔트리

`startup_stm32f103xb.s` 등에서 다음과 같이 정의됨:

```
...
.word USART1_IRQHandler   /* USART1 global interrupt */
...
```

(2) CMSIS Handler (사용자 코드 레벨)

```
void USART1_IRQHandler(void)
{
    HAL_UART_IRQHandler(&huart1);
}
```

(3) HAL 드라이버 레벨 (`stm32f1xx_hal_uart.c`)

```
void HAL_UART_IRQHandler(UART_HandleTypeDef *huart)
{
    uint32_t isrflags   = READ_REG(huart->Instance->SR);
    uint32_t cr1its     = READ_REG(huart->Instance->CR1);

    // RXNE Flag 확인
    if (((isrflags & USART_SR_RXNE) != RESET) && ((cr1its & USART_CR1_RXNEIE) != RESET))
    {
        UART_Receive_IT(huart);
        return;
    }

    // TXE Flag 확인
    if (((isrflags & USART_SR_TXE) != RESET) && ((cr1its & USART_CR1_TXEIE) != RESET))
    {
        UART_Transmit_IT(huart);
        return;
    }

    // Error 처리 등
}
```

(4) 콜백 함수 호출

```
static void UART_Receive_IT(UART_HandleTypeDef *huart)
{
    *huart->pRxBuffPtr++ = (uint8_t)(huart->Instance->DR & (uint8_t)0x00FF);
    huart->RxXferCount--;

    if (huart->RxXferCount == 0U)
```

```
    {
        __HAL_UART_DISABLE_IT(huart, UART_IT_RXNE);
        huart->RxState = HAL_UART_STATE_READY;

        // 사용자 콜백 호출
        HAL_UART_RxCpltCallback(huart);
    }
}
```

(5) 사용자 레벨 콜백 구현

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance == USART1)
    {
        printf("UART1 RX Complete: %c\r\n", rxData);
        HAL_UART_Receive_IT(&huart1, &rxData, 1); // 다시 수신 대기
    }
}
```

---

4. 일반적인 HAL IRQ Handler 호출 패턴

주변장치	NVIC 핸들러	HAL 핸들러	대표 콜백
EXTI (GPIO)	`EXTI15_10_IRQHandler()`	`HAL_GPIO_EXTI_IRQHandler(pin)`	`HAL_GPIO_EXTI_Callback(pin)`
TIM	`TIM2_IRQHandler()`	`HAL_TIM_IRQHandler(&htim2)`	`HAL_TIM_PeriodElapsedCallback()`
ADC	`ADC1_2_IRQHandler()`	`HAL_ADC_IRQHandler(&hadc1)`	`HAL_ADC_ConvCpltCallback()`
I2C	`I2C1_EV_IRQHandler()` / `I2C1_ER_IRQHandler()`	`HAL_I2C_EV_IRQHandler()` / `HAL_I2C_ER_IRQHandler()`	`HAL_I2C_MasterRxCpltCallback()` 등
DMA	`DMA1_Channel1_IRQHandler()`	`HAL_DMA_IRQHandler(&hdma_adc1)`	`HAL_DMA_XferCpltCallback()`
UART	`USARTx_IRQHandler()`	`HAL_UART_IRQHandler(&huartx)`	`HAL_UART_TxCpltCallback()` 등

---

5. 인터럽트 Enable 및 등록

IRQ 핸들러가 동작하려면 NVIC 설정이 선행되어야 한다.

```
HAL_NVIC_SetPriority(USART1_IRQn, 1, 0);
HAL_NVIC_EnableIRQ(USART1_IRQn);
```

HAL 드라이버 초기화 (`HAL_UART_Init()`, `HAL_TIM_Base_Start_IT()` 등) 시
자동으로 인터럽트 비트(예: `TXEIE`, `UIE`)가 설정된다.

---

6. 인터럽트 핸들러 구조 요약

계층	역할	코드 위치
NVIC / CMSIS Handler	IRQ 진입점, HAL Handler 호출	`stm32fxxx_it.c`
HAL Driver Handler	상태 플래그 확인, 내부 함수 호출	`stm32fxxx_hal_*.c`
Callback Dispatcher	사용자 이벤트 알림	HAL 내부
User Callback	응용 로직 처리	`main.c` 또는 `user_code.c`

---

7. 주의사항

1. HAL 콜백은 **Interrupt Context**에서 실행됨 → 짧고 빠르게 처리해야 함.
2. FreeRTOS 사용 시, 콜백 내부에서 RTOS API (`osDelay()`, `printf()`) 직접 호출 금지.
3. 여러 인터럽트 공유 시(예: `DMA1_Channel1_IRQHandler()`가 여러 채널 처리),
   HAL 내부에서 핸들 핀 포인터로 구분되므로 `&hdma_xx` 매칭 필수.
4. NVIC에서 IRQ 비활성화 후 HAL IRQ 호출 시 콜백이 작동하지 않음.

---

8. 결론

HAL 인터럽트 호출 체계는 다음과 같이 요약된다:

```
[하드웨어 IRQ]
   ↓
[NVIC → IRQHandler()]
   ↓
[HAL_XXX_IRQHandler()]
   ↓
[내부 이벤트 처리 및 Flag Clear]
   ↓
[사용자 콜백 호출]
```

이 구조를 이해하면 **사용자 코드 수정 없이 인터럽트 흐름을 제어**할 수 있으며,
특정 이벤트만 필터링하거나, ISR 기반 비동기 시스템을 안정적으로 설계할 수 있다.

- # RTC Alarm IRQ, UART RX IRQ, EXTI IRQ 실습

1. 개요

이 절에서는 **RTC 알람**, **UART 수신 완료**, **GPIO 외부 인터럽트(EXTI)** 의 세 가지 대표적인 인터럽트를
STM32 HAL 기반으로 직접 구성하고, 각 인터럽트의 처리 흐름과 콜백 구조를 실습한다.

세 인터럽트는 모두 **NVIC → HAL Handler → User Callback** 계층 구조를 따르며,
이를 통해 MCU가 이벤트 중심으로 동작하는 실시간 반응형 시스템을 구현할 수 있다.

---

2. RTC Alarm 인터럽트

(1) 구성 개요

RTC는 내부 또는 외부 저속 클럭(LSE/LSI)을 기반으로 동작하며,
 알람(Alarm A / Alarm B) 이벤트를 NVIC를 통해 인터럽트로 발생시킬 수 있다.

(2) CubeMX 설정

- Peripherals → RTC → 활성화
- Clock Source: **LSE** 선택
- NVIC → “RTC Alarm” Interrupt Enable

(3) 코드 예시

```
RTC_AlarmTypeDef sAlarm;

void Set_RTC_Alarm(void)
{
    sAlarm.AlarmTime.Hours = 0x00;
    sAlarm.AlarmTime.Minutes = 0x00;
    sAlarm.AlarmTime.Seconds = 0x10;  // 10초 후 알람
    sAlarm.AlarmMask = RTC_ALARMMASK_DATEWEEKDAY;
    sAlarm.Alarm = RTC_ALARM_A;
    HAL_RTC_SetAlarm_IT(&hrtc, &sAlarm, RTC_FORMAT_BCD);
}
```

(4) 인터럽트 핸들러

```
void RTC_Alarm_IRQHandler(void)
{
    HAL_RTC_AlarmIRQHandler(&hrtc);
}
```

(5) 사용자 콜백

```
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    printf("RTC Alarm Interrupt Triggered!\r\n");
    HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_12); // 펌프 제어 예시
}
```

(6) 실험 결과

- 알람 시간 도래 시 UART 콘솔에 메시지 출력
- PB12 릴레이 제어 핀 토글로 알람 기반 제어 확인 가능

---

3. UART RX 인터럽트

(1) 개요

UART는 비동기 통신 방식으로, **RXNE(Receive Not Empty)** 플래그가 설정되면
인터럽트를 발생시켜 수신 데이터를 실시간으로 처리할 수 있다.

(2) CubeMX 설정

- USART1 → Asynchronous Mode
- NVIC → USART1 global interrupt Enable
- Baud rate: 115200, 8N1

(3) 코드 예시

```
uint8_t rxData;

void UART_Start_Receive_IT(void)
{
    HAL_UART_Receive_IT(&huart1, &rxData, 1);
}
```

(4) IRQ Handler

```
void USART1_IRQHandler(void)
{
    HAL_UART_IRQHandler(&huart1);
}
```

(5) 콜백 함수

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance == USART1)
    {
        printf("Received: %c\r\n", rxData);
        HAL_UART_Receive_IT(&huart1, &rxData, 1); // 재수신 대기
    }
}
```

(6) 실험 결과

- UART 터미널에서 문자 입력 시 MCU가 인터럽트로 즉시 처리
- 입력된 문자를 UART를 통해 즉시 에코(Echo) 출력

---

4. EXTI (외부 인터럽트)

(1) 개요

GPIO 핀을 외부 인터럽트 소스로 설정하면, Rising/Falling Edge에 따라
이벤트를 감지하고 즉시 콜백을 실행할 수 있다.

(2) CubeMX 설정

- GPIO Input 핀 지정 (예: PA0: Button)
- External Interrupt Mode with Rising Edge
- NVIC → EXTI Line0 Enable

(3) 코드 예시

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if (GPIO_Pin == GPIO_PIN_0)
    {
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13); // LED 토글
        printf("EXTI Interrupt Triggered on PA0!\r\n");
    }
}
```

(4) IRQ Handler

```
void EXTI0_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
}
```

(5) 실험 결과

- PA0 버튼을 눌렀을 때 LED가 즉시 반전되고 UART에 이벤트 메시지 출력
- 인터럽트 디바운스 처리를 위해 약간의 소프트웨어 지연 또는 타이머 기반 필터 적용 가능

---

5. 통합 동작 흐름

세 인터럽트를 동시에 동작시킬 경우,
MCU는 각 이벤트 발생 시 독립적인 콜백을 수행한다.

```
RTC Alarm Event   → HAL_RTC_AlarmAEventCallback()
UART RX Event     → HAL_UART_RxCpltCallback()
GPIO EXTI Event   → HAL_GPIO_EXTI_Callback()
```

이 방식은 **실시간 멀티이벤트 시스템** 구현의 핵심이며,
FreeRTOS 환경에서는 각 콜백에서 **Task Notify** 또는 **Queue 전송**을 통해
RTOS Task 간 비동기 동작을 트리거할 수 있다.

---

6. 주의사항

1. 모든 콜백은 인터럽트 컨텍스트에서 실행되므로 **짧고 비차단적**으로 작성해야 한다.
2. UART 콜백 내에서 `printf()` 사용 시 버퍼링이 필요할 수 있음 (DMA 기반 권장).
3. EXTI 핀은 디바운스가 없으므로 버튼이나 스위치 입력 시 **소프트웨어 필터링** 필요.
4. RTC 알람은 **Backup Domain Reset** 시 초기화되므로 전원 관리 시 유의.

---

7. 요약

인터럽트 종류	NVIC 핸들러	HAL Handler	사용자 콜백	주요 용도
RTC Alarm	`RTC_Alarm_IRQHandler()`	`HAL_RTC_AlarmIRQHandler()`	`HAL_RTC_AlarmAEventCallback()`	주기 측정 / Wake-up
UART RX	`USARTx_IRQHandler()`	`HAL_UART_IRQHandler()`	`HAL_UART_RxCpltCallback()`	비동기 통신
EXTI	`EXTIx_IRQHandler()`	`HAL_GPIO_EXTI_IRQHandler()`	`HAL_GPIO_EXTI_Callback()`	버튼 / 센서 트리거

이 세 가지 인터럽트는 STM32에서 **시간, 통신, 이벤트 감지**의 세 축을 구성하며,
FreeRTOS 또는 비RTOS 환경 모두에서 핵심적인 실시간 제어 루틴의 기반이 된다.

9.3 실습

- # RTC 알람 인터럽트 → Task Notify

1. 개요

RTC 알람 인터럽트를 단순한 HAL 콜백 수준이 아닌,
FreeRTOS Task에 이벤트를 전달(Notify) 하여 **측정 루틴, 디스플레이 업데이트, 로깅 등**을
Task 단위로 처리할 수 있도록 확장한다.

이 방식은 “인터럽트는 트리거만, 작업은 Task에서” 수행하는 RTOS 설계 원칙을 따른다.

---

2. 기본 흐름도

```
[RTC Alarm 발생]
        ↓
RTC_Alarm_IRQHandler()
        ↓
HAL_RTC_AlarmIRQHandler()
        ↓
HAL_RTC_AlarmAEventCallback()
        ↓
xTaskNotifyFromISR(RTCTaskHandle, ...)
        ↓
RTCTask (대기 상태 → 실행)
        ↓
측정 / 로깅 / 제어 루틴 수행
```

---

3. 코드 구조

(1) RTCTask 생성

```
#include "cmsis_os.h"

TaskHandle_t RTCTaskHandle;

void RTCTask(void *argument)
{
    uint32_t ulNotifyValue;

    for (;;)
    {
        // 알람 발생 시까지 대기
        xTaskNotifyWait(0x00, 0xFFFFFFFF, &ulNotifyValue, portMAX_DELAY);

        if (ulNotifyValue == 0x01)
        {
            printf("RTC Alarm Triggered → Measurement Task Start\r\n");
            // 측정 루틴, EEPROM 저장 등 수행
            Perform_Measurement();
        }
    }
}
```

> `xTaskNotifyWait()`은 Task가 특정 Notify 신호를 받을 때까지 블록 대기한다.
> Notify 값은 `xTaskNotifyFromISR()`에서 전달한다.

---

(2) 메인 초기화

```
void MX_FREERTOS_Init(void)
{
    xTaskCreate(RTCTask, "RTCTask", 256, NULL, 2, &RTCTaskHandle);
}
```

---

(3) RTC 알람 설정

```
void Set_RTC_Alarm(void)
{
    RTC_AlarmTypeDef sAlarm;
    sAlarm.AlarmTime.Seconds = 0x10;
    sAlarm.AlarmMask = RTC_ALARMMASK_DATEWEEKDAY;
    sAlarm.Alarm = RTC_ALARM_A;
    HAL_RTC_SetAlarm_IT(&hrtc, &sAlarm, RTC_FORMAT_BCD);
}
```

---

(4) 인터럽트 콜백에서 Task Notify

```
extern TaskHandle_t RTCTaskHandle;

void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    // Task에 알람 신호 전달
    vTaskNotifyGiveFromISR(RTCTaskHandle, &xHigherPriorityTaskWoken);

    // 필요시 즉시 컨텍스트 스위치
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

또는 `xTaskNotifyFromISR()` 버전으로 더 구체적인 값 전달도 가능하다.

```
xTaskNotifyFromISR(RTCTaskHandle, 0x01, eSetValueWithOverwrite,
&xHigherPriorityTaskWoken);
```

---

4. 동작 순서

1. `Set_RTC_Alarm()`에서 알람 타이머 설정
2. 지정된 시간 도래 → `RTC_Alarm_IRQHandler()` 발생
3. HAL이 콜백(`HAL_RTC_AlarmAEventCallback`) 호출
4. 콜백에서 `vTaskNotifyGiveFromISR()` 실행
5. RTCTask가 블록 해제되어 실행됨
6. RTCTask에서 측정 루틴 수행 후 다시 대기

---

5. 주의사항

구분	설명
ISR vs Task Context	HAL_RTC 콜백은 ISR 컨텍스트에서 실행되므로 FreeRTOS API 중 `_FromISR()` 계열만 사용해야 함
즉시 전환	`portYIELD_FROM_ISR()` 호출 시, 알람 발생 즉시 RTCTask로 스위치 가능
Stack 여유	RTCTask 내에서 측정/연산을 수행하므로 Stack 크기를 넉넉히 할당 (>=256 words 권장)
동시 알람 방지	다음 알람 예약 전 기존 알람 clear 필요 (`__HAL_RTC_ALARM_CLEAR_FLAG()`)

---

6. 통합 예시 로그

```
[System Boot]
RTC Initialized
RTC Alarm set for +10 sec

-- 10초 후 --
[ISR] RTC Alarm Triggered
[RTCTask] Alarm Notify Received
[RTCTask] Performing Measurement...
[RTCTask] Completed and Waiting Next Alarm
```

---

7. 확장 아이디어

- RTC 알람 주기를 동적으로 조정 (`HAL_RTC_SetAlarm_IT()` 재호출)
- 측정 Task와 Display Task 간 Queue 연결
- Sleep 모드에서 알람으로 Wake-up 후 Notify 처리 (Tickless Idle 연계)
- EEPROM에 측정 주기별 타임스탬프 로깅

요약:

RTC 알람 인터럽트를 FreeRTOS Task로 Notify하는 구조는
MCU가 슬립 상태에서도 저전력으로 대기하다가
정해진 시점에 필요한 루틴만 수행하도록 만드는 **이벤트 기반 스케줄링 핵심 구조**이다.

• 초음파 ECHO 인터럽트 기반 측정

1. 개요

초음파 센서(HC-SR04 등)는 **TRIG 핀**으로 짧은 트리거 펄스를 송신하면,
반사된 신호가 돌아올 때까지 **ECHO 핀**을 High로 유지한다.
이 High 구간의 **펄스폭**이 바로 초음파가 왕복한 시간에 비례한다.

일반적인 방법은 `HAL_GPIO_ReadPin()`을 이용한 **폴링 방식**이지만,
정확도 향상과 CPU 점유율 절감을 위해 **외부 인터럽트(EXTI)** 혹은 **타이머 입력 캡처**로 처리할 수 있다.
여기서는 EXTI 기반 인터럽트 방식을 중심으로 설명한다.

---

2. 측정 원리

1. MCU가 **TRIG 핀**을 10 µs 동안 High 출력 → 초음파 송신
2. 초음파 반사 신호 수신 시 **ECHO 핀**이 High 상태로 유지
3. ECHO가 Low로 돌아올 때까지의 시간을 측정
4. 거리 계산식 적용

$$\text{Distance (cm)} = \frac{Time(\mu s)}{58.0}$$

---

3. 하드웨어 연결

HC-SR04	STM32 핀	설명
VCC	5V	전원
GND	GND	공통 접지
TRIG	PA1 (예시)	출력 핀
ECHO	PA2 (예시)	EXTI 입력 핀

> ECHO는 5V 출력이므로, STM32 3.3V 입력 핀에 연결 시
> 반드시 **레벨 시프터** 또는 **저항 분압기(10k:10k)** 사용.

---

4. 소프트웨어 구성

(1) TRIG 펄스 생성

```
void Ultrasonic_Trigger(void)
{
    HAL_GPIO_WritePin(TRIG_GPIO_Port, TRIG_Pin, GPIO_PIN_SET);
    delay_us(10);   // 최소 10µs 유지
    HAL_GPIO_WritePin(TRIG_GPIO_Port, TRIG_Pin, GPIO_PIN_RESET);
}
```

---

(2) EXTI 인터럽트 설정

CubeMX에서

- ECHO 핀을 `GPIO_Input` + `External Interrupt Mode with Rising/Falling edge`로 설정
- NVIC에서 `EXTI2_IRQn`(또는 해당 핀) 활성화

---

(3) 인터럽트 콜백 처리

```
volatile uint32_t echo_start = 0;
volatile uint32_t echo_end = 0;
volatile uint8_t  echo_captured = 0;

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if (GPIO_Pin == ECHO_Pin)
    {
        if (HAL_GPIO_ReadPin(ECHO_GPIO_Port, ECHO_Pin) == GPIO_PIN_SET)
        {
            // 상승 에지: 시작 시각 저장
            echo_start = __HAL_TIM_GET_COUNTER(&htim2);
        }
        else
```

```
        {
            // 하강 에지: 종료 시각 저장
            echo_end = __HAL_TIM_GET_COUNTER(&htim2);
            echo_captured = 1;
        }
    }
}
```

> 타이머는 마이크로초 단위 동작을 위해 **1 MHz (1 µs 단위)** 설정 권장.
> CubeMX에서 TIM2 Prescaler 조정:
>
> ```
> Prescaler = (SystemCoreClock / 1,000,000) - 1
> Period = 0xFFFF
> ```

---

(4) 거리 계산 루틴

```
float Ultrasonic_GetDistance(void)
{
    if (echo_captured)
    {
        uint32_t duration;
        if (echo_end >= echo_start)
            duration = echo_end - echo_start;
        else
            duration = (0xFFFF - echo_start) + echo_end;

        echo_captured = 0;

        // 거리(cm) 계산
        float distance = duration / 58.0f;
        return distance;
    }
    return -1.0f;  // 아직 데이터 없음
}
```

---

(5) 메인 루프

```
while (1)
{
    Ultrasonic_Trigger();
    HAL_Delay(100);

    float dist = Ultrasonic_GetDistance();
    if (dist > 0)
        printf("Distance: %.2f cm\r\n", dist);
}
```

---

5. 동작 시퀀스

```
TRIG ↑————10µs————↓
             ↓
          ECHO ↑————————————————↑
                 <—— 펄스폭 t ——>
```

- 상승 에지에서 `echo_start` 저장
- 하강 에지에서 `echo_end` 저장
- 차이(`echo_end - echo_start`) = 펄스폭
- 펄스폭 → 거리 변환

---

6. 특징 및 장점

항목	설명
정확도	인터럽트 방식은 소프트웨어 폴링 지연이 없어 µs 단위 정밀 측정 가능
CPU 효율	측정 대기 동안 CPU가 다른 작업 수행 가능
확장성	여러 초음파 센서 병렬 처리 시, 타이머 채널 또는 EXTI 핀 분리 가능

---

7. Fault 대응

문제	원인	해결
펄스폭 0 또는 과도한 값	반사 신호 없음 / 타임아웃 누락	타이머 오버플로 확인 및 `MAX_TIMEOUT` 설정
잡음 트리거	근접 반사 or 전원 노이즈	Rising/Falling Edge 디바운스 처리
CPU Block	잘못된 while 대기 루프	인터럽트 기반으로 대체 (본 구조)

---

8. 결론

ECHO 핀을 EXTI로 연결하고 타이머를 활용하여
상승·하강 에지를 시간차로 계산하면,
µs 단위의 초음파 거리 측정이 가능하다.

이는 FreeRTOS 환경에서도 `xTaskNotifyFromISR()` 등으로
측정 완료 이벤트를 Task로 전달해
비동기 측정 루틴으로 확장할 수 있다.

UART RX Complete Callback 로깅

1. 개요

UART 통신에서 수신 데이터 처리를 효율적으로 수행하기 위해
STM32 HAL은 **비동기 수신 인터럽트 기반 콜백 구조**를 제공한다.

그중 `HAL_UART_RxCpltCallback()` 은 **한 바이트 또는 지정된 버퍼 수신 완료 시 자동 호출되는 콜백 함수**로,
이 콜백 내에서 **데이터 로깅, 명령 파싱, Task Notify, Queue 전송** 등의 처리를 수행할 수 있다.

UART RX 콜백 로깅 구조는
"폴링 없이 이벤트 기반으로 UART 입력을 처리하는"
핵심 메커니즘이다.

---

2. UART 수신 구조

UART 수신에는 세 가지 모드가 존재한다.

모드	함수	특징
폴링	`HAL_UART_Receive()`	Blocking 방식, CPU 점유율 높음
인터럽트	`HAL_UART_Receive_IT()`	1바이트 또는 N바이트 단위 수신 후 콜백 호출
DMA	`HAL_UART_Receive_DMA()`	대용량 수신 가능, CPU 부하 최소화

여기서는 **Interrupt 방식 (`HAL_UART_Receive_IT()`)** 기반으로 설명한다.

---

3. 초기 설정

(1) UART 핸들 초기화

```
UART_HandleTypeDef huart2;

void MX_USART2_UART_Init(void)
{
    huart2.Instance = USART2;
    huart2.Init.BaudRate = 115200;
    huart2.Init.WordLength = UART_WORDLENGTH_8B;
    huart2.Init.StopBits = UART_STOPBITS_1;
    huart2.Init.Parity = UART_PARITY_NONE;
    huart2.Init.Mode = UART_MODE_TX_RX;
    huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart2.Init.OverSampling = UART_OVERSAMPLING_16;
    HAL_UART_Init(&huart2);
}
```

---

(2) 인터럽트 기반 수신 시작

```
uint8_t rx_data;

void UART_StartReceive(void)
{
    HAL_UART_Receive_IT(&huart2, &rx_data, 1);  // 1바이트씩 수신
}
```

> 반드시 초기화 후 즉시 호출해야 첫 수신 이벤트를 놓치지 않는다.
> 콜백 내에서 다음 `HAL_UART_Receive_IT()`를 재시작하는 구조를 취한다.

---

4. 콜백 함수 구현

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance == USART2)
    {
        // (1) 수신 데이터 로깅
        printf("[UART RX] %c\r\n", rx_data);

        // (2) 버퍼에 누적 저장
        UART_LogBuffer_Append(rx_data);

        // (3) 명령어 종료 문자 검사
        if (rx_data == '\n')
        {
            UART_ProcessCommand(UART_LogBuffer_Get());
        }

        // (4) 다음 바이트 수신 재개
        HAL_UART_Receive_IT(&huart2, &rx_data, 1);
    }
}
```

---

5. 로깅 버퍼 구조 예시

```
#define UART_LOG_SIZE 128
static uint8_t log_buf[UART_LOG_SIZE];
static uint8_t log_idx = 0;

void UART_LogBuffer_Append(uint8_t data)
{
    if (log_idx < UART_LOG_SIZE - 1)
    {
        log_buf[log_idx++] = data;
    }
    else
    {
```

```
        log_idx = 0; // Overflow 방지
    }
}

char* UART_LogBuffer_Get(void)
{
    log_buf[log_idx] = '\0';
    log_idx = 0;
    return (char*)log_buf;
}
```

---

6. FreeRTOS 환경 연동

콜백에서 직접 처리하지 않고 **Task로 이벤트 전달**할 수도 있다.

```
extern TaskHandle_t UARTTaskHandle;

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    if (huart->Instance == USART2)
    {
        xTaskNotifyFromISR(UARTTaskHandle, rx_data, eSetValueWithOverwrite, &xHigherPriorityTaskWoken);
        HAL_UART_Receive_IT(&huart2, &rx_data, 1);

        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    }
}
```

Task 내에서 수신 문자를 누적하고 명령어 단위로 파싱 가능하다.

---

7. 동작 로그 예시

```
[UART RX] A
[UART RX] T
[UART RX] +
[UART RX] C
[UART RX] M
[UART RX] D
[UART RX] ?
[UART RX] OK
Received Command: AT+CMD?
```

---

8. Fault 및 예외 처리

문제	원인	해결
데이터 누락	콜백 내에서 `HAL_UART_Receive_IT()` 재시작 누락	콜백 마지막에 반드시 재호출
중복 출력	`HAL_UART_Receive_IT()` 중복 호출	수신 완료 후에만 재개
프레임 오류 (FE)	잡음 또는 속도 불일치	Baudrate 정합 확인, `HAL_UART_ErrorCallback()` 처리 추가
버퍼 오버플로	긴 명령어 입력	버퍼 크기 증가 또는 DMA 모드 전환

---

9. 확장 기능

- DMA + Idle Line Detection을 사용한 가변 길이 패킷 수신
- 명령어 기반 CLI(Command Line Interface) 구현
- 로깅 데이터를 SD 카드 / EEPROM에 저장
- BLE 또는 Wi-Fi 모듈 디버그 로깅에 활용

정리:

`HAL_UART_RxCpltCallback()`은 **UART 수신의 핵심 인터럽트 후처리 루틴**으로,
여기서 로그 기록, 명령 파싱, Task 통지 등을 수행함으로써
효율적이고 실시간성이 높은 UART 기반 시스템을 구성할 수 있다.