2. GPIO 제어와 디지털 입출력

2.1 GPIO 기본

- ## Port와 Pin 구조

1. 개요

STM32 마이크로컨트롤러는 다수의 **GPIO(General Purpose Input/Output)** 핀을 통해 외부 하드웨어와 상호 작용한다.
이 핀들은 **포트(Port)** 단위로 그룹화되어 있으며, 각 포트는 **최대 16개의 핀(Pin)** 으로 구성된다.
예를 들어, `Port A`에는 `PA0~PA15`가, `Port B`에는 `PB0~PB15`가 존재한다.

STM32F103C8T6(Blue Pill)은
Port A, Port B, Port C 일부 핀을 제공하며, 포트 단위로 독립된 레지스터를 통해 제어된다.

---

2. 포트 구조 (GPIO Port)

2.1 포트 구성

포트명	사용 가능 핀	주요 용도
GPIOA	PA0~PA15	ADC, TIM, USART1 등
GPIOB	PB0~PB15	I²C, SPI, PWM, ADC
GPIOC	PC13~PC15	LED, 외부 인터럽트, RTC

> STM32F103C8T6는 **GPIO D~G 포트가 존재하지 않는다.**

각 포트는 독립된 **레지스터 집합(GPIOx_CR, GPIOx_IDR, GPIOx_ODR, GPIOx_BSRR 등)** 을 가지며,
이 레지스터를 통해 핀 방향, 속도, 입력/출력 상태를 설정할 수 있다.

---

3. 핀 구조 (GPIO Pin)

각 포트의 핀은 **0~15번 인덱스**로 구분된다.
예를 들어, `PA0`는 Port A의 0번 핀, `PB12`는 Port B의 12번 핀이다.

이름	의미
`P`	Port (예: A, B, C)
`xx`	Pin 번호 (0~15)
예시	`PA0`, `PB9`, `PC13`

---

4. GPIO 레지스터 개요

레지스터	설명
GPIOx_CRL	핀 0~7 설정 (Mode + Config)
GPIOx_CRH	핀 8~15 설정 (Mode + Config)
GPIOx_IDR	입력 데이터 레지스터
GPIOx_ODR	출력 데이터 레지스터
GPIOx_BSRR	출력 비트 세트/리셋 (1로 세트, 0으로 리셋)
GPIOx_BRR	출력 비트 리셋 전용
GPIOx_LCKR	핀 설정 Lock 레지스터

> 예: `GPIOA->ODR |= (1 << 5);` → PA5를 High로 설정
> 예: `GPIOA->IDR & (1 << 0);` → PA0 입력 상태 읽기

---

5. 핀 모드와 구성

STM32의 각 핀은 다기능(Multiplexed) 구조로, **입력·출력·대체 기능(Alternate Function)** 중 하나로 설정 가능하다.

모드	설명	예시
Input Mode	외부 신호 입력 (디지털/아날로그)	스위치, 센서 입력
Output Mode	MCU가 신호를 출력	LED 제어, 릴레이 구동
Alternate Function	내부 주변장치 기능 연결	UART, SPI, I²C, PWM
Analog Mode	ADC 입력 / 저전력 상태	센서 아날로그 전압 측정

---

6. 출력 구성 (Output Type)

STM32는 두 가지 출력 타입을 지원한다.

타입	설명	회로 특성
Push-Pull	High/Low 직접 구동	일반적인 LED, 릴레이, 로직 회로
Open-Drain	GND만 구동, High는 외부 Pull-up	I²C, 버스형 회로, 센서 공유라인

> 예: `GPIO_MODE_OUTPUT_PP` → Push-Pull
> 예: `GPIO_MODE_OUTPUT_OD` → Open-Drain

---

7. 입력 구성 (Input Type)

입력 모드에서도 다양한 전기적 특성이 존재한다.

모드	설명	비고
Floating Input	내부 Pull-up/down 없음	외부 회로 필요
Pull-up Input	내부 Pull-up 저항 연결	스위치 입력 기본형
Pull-down Input	내부 Pull-down 저항 연결	Active-High 스위치 대응
Analog Input	ADC 채널 입력	전류 소모 최소화

---

8. GPIO 클록 활성화

각 포트의 GPIO는 **RCC (Reset and Clock Control)** 에 의해 클록이 공급될 때만 동작한다.
클록을 활성화하지 않으면 HAL 함수 호출 시에도 동작하지 않는다.

```
__HAL_RCC_GPIOA_CLK_ENABLE();
__HAL_RCC_GPIOB_CLK_ENABLE();
__HAL_RCC_GPIOC_CLK_ENABLE();
```

> RCC는 주변장치의 전원을 제어하며, 불필요한 포트의 클록을 비활성화하여 전력 소비를 줄인다.

---

9. HAL 구조 내 포트 제어 함수

함수	설명
`HAL_GPIO_WritePin(GPIOx, Pin, State)`	지정 핀 출력 설정
`HAL_GPIO_ReadPin(GPIOx, Pin)`	지정 핀 입력 상태 읽기
`HAL_GPIO_TogglePin(GPIOx, Pin)`	출력 토글
`HAL_GPIO_Init(GPIOx, &GPIO_InitStruct)`	핀 초기화 (Mode, Pull, Speed 등)
`HAL_GPIO_DeInit(GPIOx, Pin)`	핀 설정 해제

예시 코드

```
GPIO_InitTypeDef GPIO_InitStruct = {0};

__HAL_RCC_GPIOC_CLK_ENABLE();

GPIO_InitStruct.Pin = GPIO_PIN_13;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;

HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
```

---

10. 구조 요약

구성 요소	설명
Port (A~C)	GPIO 블록 단위, 각기 별도 레지스터 제어
Pin (0~15)	개별 비트 단위 입출력 제어
Mode / Config	입력·출력·대체 기능 정의
Pull / Speed	전기적 특성 제어
Clock Enable	RCC를 통한 포트 활성화 필요

결론
STM32의 Port/Pin 구조는 단순한 디지털 입출력을 넘어,
다중 기능 선택(Multiplexed Function), 전기적 구성(Push-Pull/Open-Drain), 클록 관리(RCC) 를 통합적으로 포함하는 유연한 구조이다.
이 구조를 정확히 이해해야 **ADC, I²C, PWM, 인터럽트 등** 상위 주변장치의 동작도 올바르게 설정할 수 있다.

- # Input, Output, Alternate, Analog 모드

1. 개요

STM32의 모든 GPIO 핀은 다기능(Multiplexed) 구조를 가지며,
하나의 핀을 **입력(Input)**, **출력(Output)**, **대체 기능(Alternate Function)**, **아날로그(Analog)** 중 하나의 모드로 설정할 수 있다.

이러한 설정은 CubeMX 또는 HAL을 통해 자동으로 생성되며,
저수준에서는 `GPIOx_CRL`, `GPIOx_CRH` 레지스터의 **MODE[1:0]** 및 **CNF[1:0]** 비트 조합으로 결정된다.

---

2. 입력(Input) 모드

(1) 개요

입력 모드는 외부 신호(예: 버튼, 센서, 인터럽트 등)를 MCU로 읽어들이는 용도로 사용된다.
 입력 모드에서는 출력 드라이버가 비활성화되며, 핀 상태를 `GPIOx_IDR`을 통해 읽는다.

(2) 구성 타입

모드	설명	내부저항	사용 예시
Floating Input	내부 풀업/풀다운 없음	없음	외부 풀업 저항이 있는 회로
Input Pull-Up	내부 Pull-up 저항 활성화	VCC로 연결	스위치 입력 (Active Low)
Input Pull-Down	내부 Pull-down 저항 활성화	GND로 연결	스위치 입력 (Active High)
Analog Input	ADC 입력, 디지털 회로 차단	없음	센서 전압 측정, 저전력 모드

(3) HAL 초기화 예시

```
GPIO_InitTypeDef GPIO_InitStruct = {0};

__HAL_RCC_GPIOA_CLK_ENABLE();

GPIO_InitStruct.Pin = GPIO_PIN_0;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_PULLUP;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

(4) 상태 읽기

```
if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
{
    // 버튼 눌림 감지
}
```

---

3. 출력(Output) 모드

(1) 개요

출력 모드는 MCU가 외부로 High 또는 Low 신호를 출력하여 LED, 릴레이, 트랜지스터 등을 구동할 때 사용된다.

(2) 출력 형식

형식	설명	전기적 특성	사용 예시
Push-Pull (PP)	High/Low를 직접 구동	소스/싱크 모두 가능	LED, 릴레이

형식	설명	전기적 특성	사용 예시
Open-Drain (OD)	Low만 구동, High는 외부 Pull-up 의존	싱크만 가능	I²C, 공유 라인
Speed 옵션	출력 상승/하강 속도 제어 (2/10/50MHz)	Slew Rate 조정	EMI 제어, 고속 신호 대응

(3) HAL 초기화 예시

```
GPIO_InitTypeDef GPIO_InitStruct = {0};

__HAL_RCC_GPIOC_CLK_ENABLE();

GPIO_InitStruct.Pin = GPIO_PIN_13;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;

HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
```

(4) 출력 제어

```
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_RESET);  // LED ON
HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);                 // 상태 토글
```

---

4. 대체 기능(Alternate Function, AF) 모드

(1) 개요

Alternate Function은 해당 핀이 GPIO가 아닌 **내부 주변장치(UART, SPI, I²C, PWM 등)** 의 기능을 수행하도록 설정하는 모드이다.
 이는 **AFIO (Alternate Function I/O)** 블록을 통해 제어되며, 각 주변장치가 어떤 핀에 연결될지는 **데이터시트의 핀 멀티플렉싱 표**에 명시된다.

(2) 주요 예시

주변장치	연결 핀	동작 모드
USART1_TX	PA9	AF Output Push-Pull
USART1_RX	PA10	Input Floating
SPI1_SCK	PA5	AF Output Push-Pull
I²C1_SCL	PB6	AF Open-Drain
TIM1_CH1 (PWM)	PA8	AF Output Push-Pull

(3) HAL 초기화 예시

```
GPIO_InitTypeDef GPIO_InitStruct = {0};

__HAL_RCC_GPIOA_CLK_ENABLE();

GPIO_InitStruct.Pin = GPIO_PIN_9; // USART1 TX
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

(4) 주의사항

- Alternate Function 사용 시 해당 주변장치 클록도 반드시 활성화해야 함.
- 동일 핀에 여러 기능이 중첩된 경우 CubeMX가 자동 선택하나, 수동 설정 시 AFIO 리맵 필요.
- 일부 핀은 **AFIO_MAPR** 레지스터를 통해 기능 리맵(Alternate Remap) 가능.

---

5. 아날로그(Analog) 모드

(1) 개요

Analog 모드는 ADC, DAC, Comparator 등 아날로그 회로와 직접 연결할 때 사용된다.
이 모드에서는 디지털 입력 버퍼가 비활성화되어 **전류 소모가 최소화**된다.

(2) 용도

기능	설명
ADC 입력	전압 측정 (센서, 배터리 모니터링 등)
DAC 출력	PWM 대신 전압 생성
저전력 모드	미사용 핀을 Analog로 설정하면 전류 소모 최소화

(3) HAL 초기화 예시

```
GPIO_InitTypeDef GPIO_InitStruct = {0};

__HAL_RCC_GPIOA_CLK_ENABLE();

GPIO_InitStruct.Pin = GPIO_PIN_0;
GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

---

6. 비교 요약

구분	설명	내부회로 동작	사용 예시
Input	외부 신호 입력	입력버퍼 활성화	버튼, 센서
Output	신호 출력	출력버퍼 활성화	LED, 릴레이
Alternate Function	내부 주변장치 기능	AFIO 매핑	UART, SPI, I²C
Analog	아날로그 회로 연결	디지털 회로 비활성	ADC, 저전력

---

7. 결론

STM32의 각 핀은 단순한 입출력 역할을 넘어서,
다기능 하드웨어 인터페이스 노드로 설계되어 있다.
입력(Input)·출력(Output)·대체기능(Alternate Function)·아날로그(Analog) 모드는
레지스터 단위의 비트 조합으로 제어되며,
각 기능 간 충돌 방지 및 전력 효율성을 고려한 설정이 중요하다.

특히, CubeMX를 활용하면 핀 모드를 직관적으로 시각화하여 설정할 수 있으므로,
하드웨어 설계와 펌웨어 구성이 동시에 일관성 있게 유지된다.

- # Push-Pull vs Open-Drain

1. 개요

STM32의 GPIO 출력 핀은 두 가지 대표적인 방식으로 동작한다.
바로 **Push-Pull(푸시풀)** 과 **Open-Drain(오픈드레인)** 출력 구조이다.

이 두 방식은 핀의 전기적 구동 특성이 다르며,
연결 회로, **전류 흐름 방향**, **외부 풀업 필요 여부**, **다중 장치 연결 가능성**에 직접적인 영향을 미친다.

---

2. Push-Pull 출력 (Push-Pull Output)

(1) 원리

Push-Pull 방식은 출력단에 **두 개의 트랜지스터(NMOS + PMOS)** 가 직렬로 연결된 구조를 갖는다.
하나의 트랜지스터가 켜지면 다른 하나는 꺼지는 형태로,
출력 핀이 **VCC(High)** 또는 **GND(Low)** 로 강하게 구동된다.

```
VCC
 |
┌┴┐ PMOS (P)
| |
└┬┘──────> GPIO Pin
┌┴┐ NMOS (N)
| |
└┬┘
 |
GND
```

- **High 출력 시:** PMOS ON, NMOS OFF → 핀에 VCC 출력
- **Low 출력 시:** PMOS OFF, NMOS ON → 핀에 GND 출력

즉, 핀이 **양방향으로 전류를 구동**할 수 있다.

(2) 특징

항목	설명
구동 능력	High/Low 모두 능동적으로 구동
외부 저항	필요 없음
출력 전압	0V 또는 3.3V (또는 5V tolerant)
전류 흐름	양방향 (소스/싱크 모두 가능)
사용 예시	LED, 릴레이, 디지털 출력, SPI, UART 등

(3) 장점

- 회로가 단순하며 별도의 외부 풀업 불필요
- 출력 전류가 크고 신호 전환 속도가 빠름
- 로직 신호 전송에 가장 일반적

(4) 단점

- 다중 출력 병렬 연결 불가능 (Bus Conflict 발생 위험)
- 외부 회로와 전압 레벨이 다를 경우 직접 연결 어려움

(5) HAL 설정 예시

```
GPIO_InitStruct.Pin = GPIO_PIN_13;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;  // Push-Pull
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
```

---

3. Open-Drain 출력 (Open-Drain Output)

(1) 원리

Open-Drain 방식은 출력단에 **NMOS 트랜지스터 하나만 존재**한다.
이 구조는 핀을 **GND로만 끌어내릴 수 있고**, High 상태는 **외부 Pull-up 저항**을 통해 공급받는다.

```
VCC
 |
[Pull-up Resistor]
 |
————————————> GPIO Pin
       |
     NMOS
       |
      GND
```

- **Low 출력:** NMOS ON → 핀이 GND로 연결
- **High 출력:** NMOS OFF → 핀이 풀업저항을 통해 VCC로 상승

(2) 특징

항목	설명
구동 능력	Low만 구동 (GND 방향)
High 출력	외부 Pull-up 저항 필요
전류 흐름	단방향 (싱크 전류만)
전압 범위	외부 풀업에 따라 MCU 전압보다 높을 수 있음
사용 예시	I²C 통신, 다중 디바이스 버스, 알람 신호선 등

(3) 장점

- 여러 장치가 한 신호선을 공유 가능 (Bus 구조에 적합)
- 외부 풀업을 이용하면 **다른 전압 레벨(예: 5V)** 로 인터페이스 가능
- 신호 충돌 없이 OR 동작 가능 (“wired-AND” 구조)

(4) 단점

- 외부 저항 필요, 상승 속도 느림 (RC 특성)
- 전류 소싱 불가 → 하이레벨 유지에 시간이 필요

(5) HAL 설정 예시

```
GPIO_InitStruct.Pin = GPIO_PIN_7;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_OD;  // Open-Drain
GPIO_InitStruct.Pull = GPIO_PULLUP;           // 내부 또는 외부 풀업
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
```

---

4. 두 방식의 비교

구분	Push-Pull	Open-Drain
High 출력 방식	내부 PMOS로 직접 VCC 출력	외부 Pull-up 저항을 통해 VCC로 상승
Low 출력 방식	내부 NMOS로 GND 구동	내부 NMOS로 GND 구동
High 전류 공급 가능성	높음	낮음 (저항 의존)
출력 속도	빠름	상대적으로 느림
외부 저항 필요 여부	불필요	필수 (Pull-up)
다중 장치 병렬 연결	불가 (단락 위험)	가능 (Bus 구조, I²C 등)
대표 사용처	LED, 릴레이, SPI, UART	I²C, 인터럽트 신호선, Fault 신호
전압 호환성	MCU 전압 동일	외부 풀업으로 다른 전압과 호환 가능

---

5. 실제 회로 예시

(1) Push-Pull (LED 제어)

```
3.3V
 |
 |
[LED]
 |
 |——> GPIO (Push-Pull)
 |
GND
```

- `HAL_GPIO_WritePin(GPIOx, Pin, RESET)` → LED ON
- 내부 드라이버가 직접 전류 공급

(2) Open-Drain (I²C Bus)

```
VCC (3.3V)
 |
[4.7kΩ]
 |
 ├── SDA ──┬──> MCU1 (OD)
 |         └──> MCU2 (OD)
 |
[4.7kΩ]
 |
 ├── SCL ──┬──> MCU1 (OD)
 |         └──> MCU2 (OD)
```

- 각 장치가 Low를 출력할 때만 라인이 GND로 떨어짐
- 여러 장치가 동시에 연결되어도 충돌 없음

---

6. 선택 기준

사용 상황	권장 출력 모드
단일 출력 장치 제어 (LED, 릴레이 등)	Push-Pull
다중 디바이스가 한 라인을 공유 (I²C, 알람 라인 등)	Open-Drain
전류 소모 최소화 / 저전력 시스템	Push-Pull (High Drive 효율)
다른 전압 레벨(예: 5V 시스템)과의 인터페이스	Open-Drain (외부 Pull-up)

---

7. 결론

Push-Pull은 **단일 신호 구동용**, Open-Drain은 **공유 버스형 신호용**으로 구분된다.
전자는 **빠르고 강력한 출력 드라이버**, 후자는 **안전한 병렬 연결 및 전압 호환성**을 제공한다.

STM32에서 적절한 출력 모드를 선택하는 것은
회로 안정성, 통신 신뢰성, 전력 효율을 결정짓는 중요한 설계 요소이다.

Pull-Up/Pull-Down 개념 및 회로 연결

1. 개요

Pull-Up과 **Pull-Down**은 입력 핀(Input Pin)이 **떠 있는 상태(Floating)** 에서
불안정하게 신호가 변동되는 것을 방지하기 위한 **기본 전압 기준 설정 회로**이다.

MCU의 입력 핀은 내부 임피던스가 매우 높아(수백 kΩ~수 MΩ),
외부 회로가 연결되지 않으면 핀 전위가 공중에 떠 있게 된다.
이 경우 전자파나 노이즈로 인해 **예기치 않은 High/Low 판정**이 발생할 수 있다.

이를 방지하기 위해 내부 또는 외부에 **풀업(Pull-Up)** 또는 **풀다운(Pull-Down) 저항**을 연결하여
기본 상태를 안정적으로 유지시킨다.

---

2. Pull-Up 회로 (상향 저항, Pull-Up Resistor)

(1) 개념

Pull-Up은 입력 핀을 **기본적으로 High(논리 1)** 상태로 유지하기 위한 방식이다.
외부 입력이 없을 때 핀이 VCC로 연결되어 '1'로 읽히며,
버튼이나 센서가 GND로 연결될 때만 Low로 변한다.

```
   VCC (3.3V)
     |
     R (10kΩ)
     |
     ├──────> MCU Input Pin
     |
   [Button]
     |
    GND
```

(2) 동작 원리

- 평상시: 전류 흐름 없음 → 입력핀은 VCC에 의해 High 유지
- 버튼 누름: GND 연결 → 입력핀 Low

(3) 특징

항목	설명
기본 상태	High
스위치 입력 시	Low로 떨어짐
전류 방향	위에서 아래로 (VCC → Pin → GND)
일반 저항값	4.7kΩ ~ 10kΩ
활용 예	스위치 입력, 오픈 컬렉터 센서 출력, I²C SDA/SCL 라인

(4) CubeMX 설정 예시

```
GPIO_InitStruct.Pin = GPIO_PIN_0;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_PULLUP;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

---

3. Pull-Down 회로 (하향 저항, Pull-Down Resistor)

(1) 개념

Pull-Down은 입력 핀을 **기본적으로 Low(논리 0)** 상태로 유지하는 방식이다.
외부 입력이 없을 때 핀은 GND로 연결되어 '0'으로 읽히며,
버튼이 눌려 VCC에 연결될 때만 High로 변한다.

```
   VCC (3.3V)
     |
   [Button]
     |
     ├──────> MCU Input Pin
     |
     R (10kΩ)
     |
    GND
```

(2) 동작 원리

- 평상시: 입력핀은 GND 쪽으로 풀다운되어 Low 유지
- 버튼 누름: VCC 연결 → High 전위 입력

(3) 특징

항목	설명
기본 상태	Low
스위치 입력 시	High로 상승
전류 방향	위에서 아래로 (VCC → Button → Pin → GND)
일반 저항값	4.7kΩ ~ 10kΩ
활용 예	스위치 입력 (Active-High), 트리거 신호 감지 등

(4) CubeMX 설정 예시

```
GPIO_InitStruct.Pin = GPIO_PIN_1;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
GPIO_InitStruct.Pull = GPIO_PULLDOWN;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

---

4. 내부 Pull-Up / Pull-Down 저항

STM32는 대부분의 GPIO에 **내부 Pull-Up / Pull-Down 저항(약 30~50kΩ)** 이 내장되어 있다.
이는 CubeMX 또는 HAL 설정으로 소프트웨어적으로 제어할 수 있으며,
외부 저항을 생략할 수 있는 간단한 회로 구성에 유용하다.

설정	내부 연결	기본 상태
`GPIO_PULLUP`	핀 → 내부 저항 → VCC	High
`GPIO_PULLDOWN`	핀 → 내부 저항 → GND	Low
`GPIO_NOPULL`	연결 없음 (Floating)	불안정

> ⚠️ 내부 저항값이 크기 때문에, 고속 신호나 노이즈가 심한 환경에서는 외부 저항(4.7kΩ~10kΩ) 사용을 권장한다.

---

5. Floating 상태와 문제점

(1) Floating 상태란?

입력 핀이 VCC나 GND로 연결되지 않아 전위가 부정확한 상태를 의미한다.

(2) 발생 문제

- MCU가 High/Low를 무작위로 판정함
- EMI, 손가락 접촉 등에 의해 불안정한 신호 발생
- 인터럽트 핀이 오작동할 수 있음

(3) 해결 방법

- 내부/외부 Pull-Up 또는 Pull-Down 저항을 설정
- 입력이 불필요한 핀은 Analog 모드로 변경 (노이즈 방지 및 전류 절약)

---

6. 회로 비교 요약

구분	Pull-Up 회로	Pull-Down 회로
기본 상태	High	Low
입력 전환 시	Low (GND 연결)	High (VCC 연결)
내부 저항 설정	`GPIO_PULLUP`	`GPIO_PULLDOWN`
외부 저항 위치	VCC와 입력핀 사이	입력핀과 GND 사이
일반 저항값	4.7kΩ ~ 10kΩ	4.7kΩ ~ 10kΩ
주 사용 예	스위치, 오픈 컬렉터, I²C 버스	Active-High 스위치, 센서 트리거

---

7. 회로 예시 (ASCII 다이어그램)

(1) Pull-Up 버튼 입력

```
3.3V
 |
[10kΩ]
 |
 ├──────> MCU Input Pin (PA0)
 |
[Button]
 |
GND
```

- 기본 상태: High
- 버튼 누름: GND 연결 → Low

(2) Pull-Down 버튼 입력

```
3.3V
 |
[Button]
 |
 ├──────> MCU Input Pin (PA1)
 |
[10kΩ]
 |
GND
```

- 기본 상태: Low
- 버튼 누름: High

---

8. 결론

Pull-Up과 Pull-Down 회로는 단순한 저항 한 개이지만,
디지털 입력의 안정성을 보장하는 핵심 회로 요소이다.
MCU 내부 저항을 적극 활용하면 하드웨어 구성을 간소화할 수 있으나,
정확성과 노이즈 내성을 요구하는 시스템에서는 반드시 외부 저항을 병행하는 것이 바람직하다.

결국 이 개념은 모든 디지털 입력 회로의 기본이며,
GPIO 동작의 신뢰성을 확보하기 위한 **기본 전기적 안전장치**이다.

- # `__HAL_RCC_GPIOx_CLK_ENABLE()` 원리

1. 개요

STM32의 모든 주변장치(Peripheral)는 **RCC (Reset and Clock Control)** 블록을 통해
클록이 공급될 때만 동작한다.

GPIO 포트(GPIOA, GPIOB, GPIOC 등) 역시 **RCC로부터 클록을 분배받아야만**
레지스터 접근 및 입출력 동작이 가능하다.

즉, `__HAL_RCC_GPIOx_CLK_ENABLE()` 매크로는 **RCC의 해당 포트 클록 비트를 "1"로 설정하여**
해당 포트의 **레지스터 접근을 허용**하고, GPIO 기능을 활성화하는 역할을 수행한다.

---

2. RCC (Reset and Clock Control) 구조

RCC는 STM32 내부의 **클록 분배 및 초기화 컨트롤러**로,
MCU 전체의 **동작 주파수, 버스 클록, 주변장치 클록, 리셋 신호**를 제어한다.

RCC 블록 다이어그램 (요약)

```
System Clock Source (HSE, HSI, PLL)
        |
        ▼
  ┌──────────────┐
  | RCC Controller|
  └──────┬───────┘
         |
         ├── AHB Bus → GPIO, DMA, CRC 등
         ├── APB1 Bus → TIM2~7, USART2, I2C1 등
         └── APB2 Bus → GPIOA~E, AFIO, ADC1~2, USART1 등
```

> **GPIO 포트는 APB2 버스에 연결되어 있다.**

---

3. RCC 클록 제어 방식

(1) RCC의 주요 역할

- **각 주변장치 클록 공급 제어**
- **리셋 초기화 제어 (Reset/Enable Bit)**
- **PLL 및 클록 소스 분배**
- **APB1 / APB2 버스 주파수 설정**

(2) 관련 레지스터

- `RCC_APB2ENR` : APB2 버스 주변장치 클록 Enable
- `RCC_APB2RSTR` : APB2 주변장치 Reset
- `RCC_AHBENR` : AHB 버스 주변장치 클록 Enable

> GPIOA, GPIOB, GPIOC는 **RCC_APB2ENR** 레지스터의 제어를 받는다.

---

4. GPIO 클록 활성화의 실제 동작

(1) RCC_APB2ENR 레지스터 구조

비트 번호	비트명	설명
0	AFIOEN	AFIO(Alternate Function I/O) 클록 Enable
2	IOPAEN	GPIOA 클록 Enable
3	IOPBEN	GPIOB 클록 Enable
4	IOPCEN	GPIOC 클록 Enable
5	IOPDEN	GPIOD 클록 Enable
6	IOPEEN	GPIOE 클록 Enable
…	…	…

- 비트를 **1로 설정**하면 해당 GPIO 포트에 클록이 공급됨.
- 0으로 유지되면 해당 포트는 **비활성화 상태**로, 레지스터 접근 시 하드폴트 발생.

---

5. 매크로의 내부 구조 분석

`__HAL_RCC_GPIOA_CLK_ENABLE()` 등의 매크로는 HAL 라이브러리에서 다음과 같이 정의되어 있다.

(1) 정의 위치

stm32f1xx_hal_rcc.h

```
#define __HAL_RCC_GPIOA_CLK_ENABLE()   (RCC->APB2ENR |= (RCC_APB2ENR_IOPAEN))
#define __HAL_RCC_GPIOB_CLK_ENABLE()   (RCC->APB2ENR |= (RCC_APB2ENR_IOPBEN))
#define __HAL_RCC_GPIOC_CLK_ENABLE()   (RCC->APB2ENR |= (RCC_APB2ENR_IOPCEN))
```

(2) 세부 내용

- `RCC` : RCC 레지스터 구조체 포인터
- `APB2ENR` : GPIO 클록 제어용 레지스터
- `RCC_APB2ENR_IOPAEN` : 비트 마스크 상수 (`#define RCC_APB2ENR_IOPAEN (1U << 2)`)

즉,

```
__HAL_RCC_GPIOA_CLK_ENABLE();
```

은 결국 다음 동작을 수행한다:

```
RCC->APB2ENR |= (1U << 2);
```

> 결과적으로 APB2 버스의 2번 비트(IOPAEN)를 '1'로 설정 → GPIOA 클록 공급 개시.

6. HAL 동작 순서

(1) GPIO 사용 시 필수 순서

1. RCC 클록 Enable

```
__HAL_RCC_GPIOA_CLK_ENABLE();
```

2. 핀 모드 설정

```
GPIO_InitTypeDef GPIO_InitStruct = {0};
GPIO_InitStruct.Pin = GPIO_PIN_5;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

3. 입출력 제어 수행

```
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);
```

> 클록이 비활성화된 상태에서 GPIO를 초기화하면 **HardFault Exception** 발생.

---

7. Disable 매크로

클록을 비활성화할 때는 다음 매크로를 사용한다.

```
#define __HAL_RCC_GPIOA_CLK_DISABLE()   (RCC->APB2ENR &= ~(RCC_APB2ENR_IOPAEN))
```

- 클록 공급 중단 → 전력 절감 효과
- 비활성화 후 GPIO 접근 시 즉시 오류 발생

---

8. 예시: GPIOA 활성화 과정

(1) 전(before)

```
RCC->APB2ENR = 0x0000_0000
GPIOA → 클록 없음 → 동작 불가
```

(2) 실행 후(after)

```
__HAL_RCC_GPIOA_CLK_ENABLE();
RCC->APB2ENR = 0x0000_0004  (bit2 = 1)
GPIOA → 클록 공급됨 → 레지스터 접근 가능
```

---

9. RCC 클록 트리 내 GPIO

```
System Clock (72 MHz)
        |
        ▼
     AHB Bus
        |
        ▼
     APB2 Bus (36 MHz)
        |
        ├── GPIOA  ←  __HAL_RCC_GPIOA_CLK_ENABLE()
        ├── GPIOB  ←  __HAL_RCC_GPIOB_CLK_ENABLE()
        ├── GPIOC  ←  __HAL_RCC_GPIOC_CLK_ENABLE()
        └── AFIO   ←  __HAL_RCC_AFIO_CLK_ENABLE()
```

- GPIO는 APB2 버스 하위에서 동작
- RCC가 APB2 클록을 배분하며, 매크로는 해당 비트를 Enable

---

10. 요약

구분	설명
매크로 이름	`__HAL_RCC_GPIOx_CLK_ENABLE()`
역할	RCC 레지스터(APB2ENR)의 GPIOx 클록 비트를 1로 설정
결과	해당 포트에 클록 공급 시작, 레지스터 접근 가능
버스 연결	APB2 버스 (36 MHz 영역)
비활성화 매크로	`__HAL_RCC_GPIOx_CLK_DISABLE()`
예외 상황	클록 비활성 상태에서 GPIO 접근 시 HardFault 발생

---

11. 결론

`__HAL_RCC_GPIOx_CLK_ENABLE()` 은 단순한 매크로처럼 보이지만,
실제로는 **RCC 내부의 APB2ENR 레지스터를 조작하여 하드웨어 레벨의 클록 분배를 제어하는 명령**이다.

이 명령이 실행되어야만 해당 포트(GPIOA~GPIOC)가 동작 가능한 상태로 전환되며,
GPIO 초기화 및 동작의 전제 조건이 된다.

결국 이 매크로는 STM32의 모든 하드웨어 주변장치가 “살아 움직이기” 시작하는
첫 번째 신호(Clock Enable Pulse) 라고 할 수 있다.

2.2 HAL GPIO 함수

- `HAL_GPIO_ReadPin()`, `HAL_GPIO_WritePin()`

1. 개요

STM32 HAL 라이브러리에서 `HAL_GPIO_ReadPin()`과 `HAL_GPIO_WritePin()`은
GPIO 입력/출력의 기본 함수로, MCU 핀의 논리 상태를 읽거나 제어하는 데 사용된다.

이 두 함수는 단순히 논리 값을 읽고 쓰는 것처럼 보이지만,
내부적으로는 **GPIO 레지스터(GPIOx_IDR, GPIOx_BSRR)** 에 직접 접근하여
하드웨어 단위의 비트 조작을 수행한다.

---

2. 함수 원형

stm32f1xx_hal_gpio.h

```
GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, GPIO_PinState
PinState);
```

---

3. `HAL_GPIO_ReadPin()` — 입력 신호 읽기

(1) 함수 정의

stm32f1xx_hal_gpio.c

```
GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
  GPIO_PinState bitstatus;

  if ((GPIOx->IDR & GPIO_Pin) != 0U)
  {
    bitstatus = GPIO_PIN_SET;
  }
  else
  {
    bitstatus = GPIO_PIN_RESET;
  }
  return bitstatus;
}
```

(2) 내부 동작 원리

- `GPIOx->IDR` : **입력 데이터 레지스터(Input Data Register)**
  각 비트는 해당 핀의 현재 입력 상태를 나타냄 (1 = High, 0 = Low)
- `GPIO_Pin` : HAL 매크로(`GPIO_PIN_0`, `GPIO_PIN_1`, ...)로 지정된 **비트 마스크**

- 비트 연산(`&`)을 통해 특정 핀의 입력 값을 추출한 뒤
  결과가 0이 아니면 `GPIO_PIN_SET`, 0이면 `GPIO_PIN_RESET` 반환.

(3) 예시

```
if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_SET)
{
    // 버튼이 눌림 (Active High)
}
else
{
    // 버튼이 눌리지 않음
}
```

(4) 실제 레지스터 관점

```
GPIOA->IDR = 0b0000_0000_0000_0001
                ↑
                └── PA0 핀이 High 상태
```

> → HAL 함수는 위 레지스터의 해당 비트를 읽고 "1"인지 "0"인지 판정한다.

---

4. `HAL_GPIO_WritePin()` — 출력 신호 제어

(1) 함수 정의

stm32f1xx_hal_gpio.c

```
void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, GPIO_PinState
PinState)
{
  if (PinState != GPIO_PIN_RESET)
  {
    GPIOx->BSRR = GPIO_Pin;
  }
  else
  {
    GPIOx->BSRR = (uint32_t)GPIO_Pin << 16U;
  }
}
```

(2) 내부 동작 원리

- `GPIOx->BSRR` : **비트 세트/리셋 레지스터(Bit Set/Reset Register)**
  - 하위 16비트(0~15): `1`로 쓰면 해당 핀을 **SET (High)**
  - 상위 16비트(16~31): `1`로 쓰면 해당 핀을 **RESET (Low)**
- 이 구조 덕분에 **읽기-수정-쓰기(Read-Modify-Write) 문제 없이**
  특정 핀만 원자적으로 제어 가능하다.

(3) 동작 요약

동작	레지스터 기록	결과
`HAL_GPIO_WritePin(GPIOx, Pin, GPIO_PIN_SET)`	`GPIOx->BSRR = Pin`	해당 핀 High
`HAL_GPIO_WritePin(GPIOx, Pin, GPIO_PIN_RESET)`	`GPIOx->BSRR = (Pin << 16)`	해당 핀 Low

(4) 예시

```
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_RESET); // LED ON
HAL_Delay(200);
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);   // LED OFF
```

(5) 실제 레지스터 동작 예시

```
GPIOC->BSRR = 0x00002000      // PC13 High
GPIOC->BSRR = 0x20000000      // PC13 Low
```

> `0x00002000` = bit13 set
> `0x20000000` = bit(13+16)=29 set → bit13 reset

---

5. BSRR vs ODR 비교

구분	ODR (Output Data Register)	BSRR (Bit Set/Reset Register)
접근 방식	전체 레지스터 직접 쓰기	특정 비트만 원자적 조작
쓰기 동작	비트 수정 시 다른 비트 영향 가능	독립적, R-M-W 문제 없음
쓰기 속도	느림 (Read-Modify-Write 필요)	빠름 (1 Cycle Write)
사용 예	대량 비트 변경 시	단일 핀 토글, 인터럽트 기반 I/O

> HAL에서는 안정성과 실시간성을 위해 항상 **BSRR 레지스터**를 사용한다.

---

6. HAL GPIO 함수의 상위 계층 구조

```
┌──────────────────────────────┐
|   HAL_GPIO_WritePin()        |
|      └─ GPIOx->BSRR 제어      |
|                              |
|   HAL_GPIO_ReadPin()         |
|      └─ GPIOx->IDR 읽기       |
└──────────────────────────────┘
```

---

7. HAL_GPIO_TogglePin() (참고)

함수 정의:

```
void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
  GPIOx->ODR ^= GPIO_Pin;
}
```

- ODR의 해당 비트를 XOR로 뒤집어 출력 상태를 반전시킴.
- LED 제어 등 주기적 토글용으로 자주 사용됨.

예시:

```
HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
HAL_Delay(100);
```

---

8. 함수 동작 비교 요약

함수명	동작 대상	내부 레지스터	동작 설명
HAL_GPIO_ReadPin()	입력핀	`GPIOx->IDR`	핀 상태 읽기 (1/0)
HAL_GPIO_WritePin()	출력핀	`GPIOx->BSRR`	핀 상태 설정 (High/Low)
HAL_GPIO_TogglePin()	출력핀	`GPIOx->ODR`	핀 상태 반전

---

9. 실습 예시: 버튼 입력 → LED 출력

회로 구성

- PA0: 버튼 입력 (Pull-up)
- PC13: LED 출력 (Active-Low)

코드 예시

```
void GPIO_Test(void)
{
    if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET) // 버튼 눌림
    {
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_RESET); // LED ON
    }
    else
    {
        HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);   // LED OFF
    }
}
```

---

10. 결론

- `HAL_GPIO_ReadPin()` 은 입력 데이터 레지스터(`IDR`)에서 특정 비트를 읽어 핀의 High/Low 상태를 반환한다.
- `HAL_GPIO_WritePin()` 은 비트 세트/리셋 레지스터(`BSRR`)를 이용해 특정 핀을 원자적으로 제어한다.

이 두 함수는 **STM32 HAL GPIO 계층의 핵심 인터페이스**로,
모든 센서 입력·LED 제어·인터럽트 트리거 등
모든 하드웨어 입출력 제어의 기반이 된다.

• `HAL_GPIO_TogglePin()` 실습

1. 개요

`HAL_GPIO_TogglePin()` 함수는 지정한 GPIO 핀의 **출력 상태를 반전(Toggle)** 시키는 함수이다.
즉, 현재 핀이 HIGH(1)이면 LOW(0)로, LOW이면 HIGH로 자동으로 전환된다.

이 함수는 **LED 점멸(blinking)**, **토글 스위칭 신호 생성**, **디버깅용 신호 출력** 등에 자주 활용된다.

---

2. 함수 원형

stm32f1xx_hal_gpio.h

```
void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
```

매개변수	설명
`GPIOx`	제어할 포트 (예: `GPIOA`, `GPIOB`, `GPIOC`)
`GPIO_Pin`	제어할 핀 번호 (`GPIO_PIN_0` ~ `GPIO_PIN_15`)

---

3. 내부 동작 원리

stm32f1xx_hal_gpio.c

```
void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
  GPIOx->ODR ^= GPIO_Pin;
}
```

- `ODR (Output Data Register)` 의 해당 비트를 **XOR(^)** 연산하여 상태를 반전시킴.
- 한 번 호출할 때마다 해당 핀의 High/Low가 바뀜.

예를 들어:

```
ODR = 0b0000_1000  (기존: 핀3 High)
HAL_GPIO_TogglePin(GPIOx, GPIO_PIN_3)
ODR = 0b0000_0000  (핀3 Low)
```

4. 사전 준비

(1) 하드웨어 구성

구성	핀	설명
LED	PC13	Blue Pill 내장 LED (Active-Low)
전원	3.3V	MCU 전원 공급
디버깅	ST-LINK	프로그램 다운로드 및 디버그용

(2) CubeMX 설정

1. **PC13** → `GPIO_Output`
2. **Mode**: `Output Push-Pull`
3. **Pull**: `No Pull`
4. **Speed**: `Low`
5. `__HAL_RCC_GPIOC_CLK_ENABLE()` 자동 추가

---

5. 코드 예시

main.c

```c
#include "main.h"

int main(void)
{
  HAL_Init();
  SystemClock_Config();

  __HAL_RCC_GPIOC_CLK_ENABLE();  // ① GPIOC 클록 활성화

  GPIO_InitTypeDef GPIO_InitStruct = {0};

  GPIO_InitStruct.Pin = GPIO_PIN_13;          // ② LED 핀 지정
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP; // 출력 모드 (Push-Pull)
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);     // ③ 핀 초기화

  while (1)
  {
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);  // ④ 핀 상태 반전 (LED 토글)
    HAL_Delay(500);                          // ⑤ 0.5초 대기
  }
}
```

---

6. 실행 결과

시간	PC13 상태	LED 동작
0 ms	HIGH (OFF)	LED 꺼짐
500 ms	LOW (ON)	LED 켜짐
1000 ms	HIGH (OFF)	LED 꺼짐
1500 ms	LOW (ON)	LED 켜짐

> Blue Pill의 내장 LED는 **Active-Low** 특성이므로,
> `RESET(0)`일 때 켜지고, `SET(1)`일 때 꺼진다.
> 따라서 토글 시마다 LED가 점멸한다.

---

7. HAL_Delay() 없이 토글 (Timer 활용)

(1) 문제점

`HAL_Delay()`는 **SysTick에 의한 Blocking Delay**로,
FreeRTOS나 인터럽트 기반 시스템에서는 CPU를 점유한다.

(2) 해결

Timer Interrupt 또는 RTOS Task를 사용하여 비차단 방식으로 LED를 토글할 수 있다.

예시: Timer 인터럽트 기반 LED 토글

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(htim->Instance == TIM2)
    {
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
    }
}
```

> TIM2를 500ms 주기로 설정하면, LED가 자동으로 1Hz로 점멸한다.

---

8. FreeRTOS Task에서 토글하기

```
void StartLedTask(void *argument)
{
  for(;;)
  {
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
    osDelay(500);
  }
}
```

- `osDelay()`는 FreeRTOS의 비차단형 지연 함수
- Task 단위로 독립된 LED 점멸 제어 가능

---

9. 디버깅용 활용 예

`HAL_GPIO_TogglePin()`은 LED 제어 외에도,
디버깅 시 특정 이벤트 발생 타이밍을 시각화할 때 매우 유용하다.

예를 들어 인터럽트 처리 시간 측정:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_0); // EXTI 트리거 시점 표시
}
```

오실로스코프로 PB0 파형을 관찰하면
인터럽트 응답 지연시간(Interrupt latency)를 측정할 수 있다.

---

10. 주의사항

항목	설명
Active-Low 핀	LED의 극성이 반대이므로 ON/OFF 반전될 수 있음
빠른 토글 시	LED 눈으로 확인 어려움 (디버그 핀 사용 권장)
다중 Task 환경	동일 핀을 여러 Task에서 제어 시, Mutex 보호 필요
전원 소비	빠른 토글은 전류 소비 및 EMI 증가 유발 가능

---

11. 요약

구분	내용
함수명	`HAL_GPIO_TogglePin(GPIOx, GPIO_Pin)`
기능	지정한 GPIO 핀의 출력 상태를 반전
내부 동작	`GPIOx->ODR ^= GPIO_Pin;` (XOR)
사용 예시	LED 점멸, 디버깅용 트리거, 신호 반전
장점	코드 간결, 빠른 실행, 하드웨어 레벨 제어
주의점	Active-Low 핀, 병행 제어 충돌 주의

---

12. 결론

`HAL_GPIO_TogglePin()` 은 STM32에서 **출력 상태를 즉시 반전**시킬 수 있는 가장 단순하면서 효율적인 함수이다.
LED 점멸 실습은 그 기본이지만, 나아가서는 **신호 타이밍 디버깅**, **주기 신호 발생**,
Task Alive 표시(Heartbeat LED) 등 실무적 활용도가 매우 높다.

이 함수는 STM32의 GPIO 제어에서 "**시스템이 살아있음을 알려주는 하트비트**"와 같은 역할을 한다.

- # 디바운스 처리, 반복 스캔 구현

1. 개요

스위치나 버튼과 같은 **기계식 입력 장치(mechanical input)** 는
누르는 순간이나 떼는 순간에 **접점이 미세하게 진동(Bounce)** 하면서
짧은 시간 동안 High/Low 신호가 빠르게 여러 번 전환되는 현상이 발생한다.

이 현상을 **채터링(Chattering)** 또는 **바운싱(Bouncing)** 이라 하며,
MCU가 이를 그대로 읽으면 **버튼을 한 번 눌렀는데 여러 번 입력된 것처럼** 인식할 수 있다.

이를 방지하기 위해 신호의 불안정 구간을 무시하고
안정된 상태만 유효 입력으로 판정하는 로직을 "**디바운스 처리(Debounce)**"라 한다.

---

2. 바운스 현상 이해

(1) 이상적인 입력

```
버튼 누르기 전:  0 ──────────────────────────────
버튼 누름:         ──────────────────────────────── 1
```

(2) 실제 기계적 신호

```
버튼 누르기 전:  0 ──────────┐
                             ├┐┌┐┌──────  (수 μs~수 ms)
                             ||||||
버튼 눌림 후:                ┘└┘└──────────── 1
```

- 이 미세한 진동이 MCU의 빠른 입력 속도(μs 단위)에서는 여러 번의 입력으로 인식됨.
- 따라서 디바운스를 수행하지 않으면, 단일 클릭으로 2~10회까지 입력 감지 가능.

---

3. 디바운스 처리 방식 분류

방식	설명	장점	단점
하드웨어 방식	RC필터(저항+커패시터) 또는 슈미트 트리거 사용	회로 안정적, MCU 부담 적음	하드웨어 추가 필요
소프트웨어 방식	MCU가 일정 시간 동안 입력을 샘플링하여 안정 상태 판정	유연성 높음, 비용 없음	타이밍 제어 필요

실무에서는 두 방법을 **병행**하는 경우가 많다.
(STM32에서 대부분의 버튼 입력은 소프트웨어 디바운스로 충분함)

---

4. 소프트웨어 디바운스 기본 원리

1. 입력 핀을 주기적으로 읽는다.
2. 신호가 변하면 일정 시간(예: 20ms) 동안 안정되었는지 확인한다.
3. 안정 구간이 지나면 "진짜 입력"으로 판정한다.

---

5. 디바운스 알고리즘 구현 예시

(1) 전역 변수

```
#define DEBOUNCE_DELAY 20  // 20ms 안정 시간

uint8_t button_state = 0;       // 현재 안정된 상태
uint8_t button_last_state = 0;  // 이전 상태
uint32_t last_debounce_time = 0; // 마지막으로 상태가 바뀐 시점
```

(2) 반복 스캔 루프

```
void Button_Scan(void)
{
    uint8_t reading = HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0);  // 버튼 입력 읽기

    if (reading != button_last_state)
    {
        last_debounce_time = HAL_GetTick(); // 상태 변화 감지 시 시각 기록
    }

    // 안정 구간 20ms 유지 확인
    if ((HAL_GetTick() - last_debounce_time) > DEBOUNCE_DELAY)
    {
        if (reading != button_state)
        {
            button_state = reading;

            if (button_state == GPIO_PIN_RESET) // Active-Low 버튼
            {
                // 유효 버튼 입력 발생
                HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13); // LED 토글
            }
        }
    }

    button_last_state = reading;
}
```

(3) main 루프에서 주기적 호출

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();

    while (1)
    {
        Button_Scan();    // 버튼 상태 반복 스캔
        HAL_Delay(1);     // 1ms 주기 (SysTick)
    }
}
```

---

6. HAL_GetTick() 기반의 시간 관리

- `HAL_GetTick()` 은 **SysTick 인터럽트(1ms 단위)** 로 증가하는 카운터를 반환한다.
- 이를 이용하면 **소프트웨어 타이머처럼** ms 단위로 안정시간을 체크할 수 있다.

즉, "20ms 동안 변하지 않으면 안정된 입력으로 판정"하는 것이 핵심이다.

---

7. 하드웨어 + 소프트웨어 병행 회로

(1) 회로 예시

```
3.3V
 |
[10kΩ] ← Pull-up
 |
 ├──> PA0 (Input)
 |
[C 100nF]
 |
GND
```

- 버튼이 눌릴 때 발생하는 진동은 **커패시터 C가 완화**함 (저역통과 필터 역할).
- 이후 MCU에서 10~20ms 디바운스 루틴으로 최종 안정 판정.

---

8. FreeRTOS 환경에서 디바운스

FreeRTOS에서는 Blocking Delay (`HAL_Delay()`) 대신 **비차단형 Task Delay** 를 사용한다.

```
void ButtonTask(void *argument)
{
    for(;;)
    {
        Button_Scan();
        osDelay(1);   // 1ms 주기 반복
    }
}
```

> 이렇게 하면 다른 Task와 병렬로 실행되어 CPU 점유율을 최소화할 수 있다.

---

9. 디바운스 처리 후의 응용 예

응용	설명
토글 버튼	버튼 누를 때마다 LED 상태 전환
길게 누름 감지	일정 시간 이상 눌림 유지 시 이벤트 발생
더블 클릭 감지	짧은 시간 내 2회 입력 시 다른 동작 수행
인터럽트 기반 입력	EXTI Interrupt에서 디바운스 검증 로직 적용

---

10. EXTI 인터럽트 기반 디바운스 예시

스위치가 외부 인터럽트 핀에 연결된 경우,
인터럽트 발생 시점 이후 20ms 동안 동일 입력 확인으로 디바운스 구현 가능.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if (GPIO_Pin == GPIO_PIN_0)
    {
        HAL_Delay(20); // 20ms 안정 대기 (Blocking, 단순 예시)

        if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
        {
            HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
        }
    }
}
```

> 실시간 성능이 중요한 경우에는 Blocking Delay 대신 타이머 콜백 기반 비차단 디바운스를 사용한다.

---

11. 반복 스캔 주기 설계

주기	특징
1ms	가장 정밀, CPU 부하 높음
5ms	권장 (대부분의 스위치 바운스 지속시간 5~20ms)
10ms	느린 입력 시스템에 적합

> 일반적으로 **1~5ms 주기 스캔 + 20ms 디바운스 시간** 조합이 가장 안정적이다.

---

12. 디바운스 실습 전체 코드

main.c (완성 예)

```c
#include "main.h"

#define DEBOUNCE_DELAY 20  // 안정 시간 20ms

uint8_t button_state = 1;       // 현재 안정 상태 (1 = Released)
uint8_t button_last_state = 1;
uint32_t last_debounce_time = 0;

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();

  while (1)
  {
    uint8_t reading = HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0);

    if (reading != button_last_state)
        last_debounce_time = HAL_GetTick();

    if ((HAL_GetTick() - last_debounce_time) > DEBOUNCE_DELAY)
    {
        if (reading != button_state)
        {
            button_state = reading;

            if (button_state == GPIO_PIN_RESET)  // 버튼 눌림
                HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
        }
    }

```

```
    button_last_state = reading;
    HAL_Delay(1);  // 1ms 주기 스캔
  }
}

static void MX_GPIO_Init(void)
{
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOC_CLK_ENABLE();

  GPIO_InitTypeDef GPIO_InitStruct = {0};

  // LED (PC13)
  GPIO_InitStruct.Pin = GPIO_PIN_13;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  // BUTTON (PA0)
  GPIO_InitStruct.Pin = GPIO_PIN_0;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}
```

---

13. 요약

항목	내용
문제 원인	스위치 접점의 기계적 진동 (Bounce)
해결 방법	디바운스 처리 (20~50ms 안정시간 확보)
핵심 함수	`HAL_GPIO_ReadPin()`, `HAL_GetTick()`
반복 스캔 주기	1~5ms
적용 대상	버튼, 센서 트리거, EXTI 입력 등
확장 기능	길게 누름, 더블 클릭, 인터럽트 필터링

---

14. 결론

디바운스 처리는 단순한 버튼 안정화 이상의 의미를 가진다.
이것은 **실시간 제어 시스템에서 신뢰성 있는 입력 판정의 시작점**이며,
스위치 입력뿐만 아니라 **리미트 센서, 엔코더, 릴레이 신호** 등
모든 디지털 입력의 품질을 결정짓는 핵심 기술이다.

STM32에서는 `HAL_GetTick()` 기반의 반복 스캔 구조만으로도
고품질의 소프트웨어 디바운싱을 손쉽게 구현할 수 있다.

2.3 EXTI (외부 인터럽트)

- # EXTI 라인 맵핑 (PA0~PA15)

1. 개요

STM32의 **EXTI (External Interrupt/Event Controller)** 는
외부 핀(GPIO)을 **인터럽트 소스**로 사용할 수 있도록 해주는 하드웨어 블록이다.

즉, **버튼, 센서, 스위치 등 외부 이벤트**가 발생하면
CPU가 이를 즉시 감지하여 **인터럽트 서비스 루틴(ISR)** 을 실행하게 된다.

STM32F103 시리즈에서는 EXTI 라인이 **0~15까지 총 16개** 존재하며,
각 라인은 **하나의 GPIO 핀 번호**와 매핑된다.

---

2. EXTI 구조 요약

(1) 기본 구조

```
┌──────────────┐
|   GPIOA~E    |
| (입력 핀)    |
└──────┬───────┘
       |
       ▼
┌──────────────────────┐
| AFIO_EXTICR[4]       | ← 포트 매핑 (A~E 선택)
└──────────────────────┘
       |
       ▼
┌──────────────────────┐
| EXTI Controller      | ← Falling/Rising Edge 감지
└──────────────────────┘
       |
       ▼
┌──────────────────────┐
| NVIC                 | ← 인터럽트 요청 전송
└──────────────────────┘
```

> 즉, EXTI 라인은 **핀 번호(Pin Number)** 에 의해 결정되며,
> **포트 번호(Port A~E)** 는 AFIO를 통해 선택된다.

---

3. EXTI 라인과 GPIO 핀의 관계

EXTI 라인	연결 가능한 핀	비고
EXTI0	PA0 / PB0 / PC0 / PD0 / PE0	
EXTI1	PA1 / PB1 / PC1 / PD1 / PE1	

EXTI 라인	연결 가능한 핀	비고
EXTI2	PA2 / PB2 / PC2 / PD2 / PE2	
EXTI3	PA3 / PB3 / PC3 / PD3 / PE3	
EXTI4	PA4 / PB4 / PC4 / PD4 / PE4	
EXTI5	PA5 / PB5 / PC5 / PD5 / PE5	공유 라인 (5~9)
EXTI6	PA6 / PB6 / PC6 / PD6 / PE6	공유 라인 (5~9)
EXTI7	PA7 / PB7 / PC7 / PD7 / PE7	공유 라인 (5~9)
EXTI8	PA8 / PB8 / PC8 / PD8 / PE8	공유 라인 (5~9)
EXTI9	PA9 / PB9 / PC9 / PD9 / PE9	공유 라인 (5~9)
EXTI10	PA10 / PB10 / PC10 / PD10 / PE10	공유 라인 (10~15)
EXTI11	PA11 / PB11 / PC11 / PD11 / PE11	공유 라인 (10~15)
EXTI12	PA12 / PB12 / PC12 / PD12 / PE12	공유 라인 (10~15)
EXTI13	PA13 / PB13 / PC13 / PD13 / PE13	공유 라인 (10~15)
EXTI14	PA14 / PB14 / PC14 / PD14 / PE14	공유 라인 (10~15)
EXTI15	PA15 / PB15 / PC15 / PD15 / PE15	공유 라인 (10~15)

- **EXTI 라인은 "핀 번호(Pin Index)"에 대응**
- 동일한 핀 번호를 가진 다른 포트 중 **하나만 활성화 가능**

예:

- PA0 ↔ EXTI0
- PB0 ↔ EXTI0
  → 둘 중 하나만 선택 가능 (AFIO로 매핑)

---

4. AFIO (Alternate Function I/O) 매핑 원리

(1) AFIO_EXTICR 레지스터

AFIO 모듈은 **EXTI 라인과 실제 포트를 연결**하는 역할을 담당한다.
레지스터는 다음과 같이 구성된다.

레지스터	대응 라인	비트 범위	역할
AFIO_EXTICR1	EXTI0~3	[15:0]	4비트씩 각 포트 선택
AFIO_EXTICR2	EXTI4~7	[15:0]	〃
AFIO_EXTICR3	EXTI8~11	[15:0]	〃

레지스터	대응 라인	비트 범위	역할
AFIO_EXTICR4	EXTI12~15	[15:0]	〃

(2) 포트 선택 비트

값	포트	의미
0000	GPIOA	기본
0001	GPIOB	
0010	GPIOC	
0011	GPIOD	
0100	GPIOE	

예를 들어, **PB0 → EXTI0 연결** 시:

```
AFIO_EXTICR1[3:0] = 0001  (PB 선택)
```

---

5. HAL 매핑 과정

HAL 라이브러리에서는 AFIO 설정이 자동으로 수행된다.
CubeMX에서 "GPIO_EXTI" 모드로 핀을 설정하면 다음 코드가 자동 생성된다:

```
GPIO_InitStruct.Pin = GPIO_PIN_0;
GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;   // 또는 RISING
GPIO_InitStruct.Pull = GPIO_PULLUP;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

→ 내부적으로 다음이 자동 수행됨:

1. `AFIO->EXTICR[0]` 에 포트 매핑 (A, B, C 등)
2. `EXTI->IMR` 의 해당 비트 Enable (Interrupt Mask)
3. `EXTI->FTSR` or `RTSR` 설정 (Falling/Rising 감지)
4. NVIC에서 IRQ 활성화 (`EXTI0_IRQn`, `EXTI1_IRQn`, ...)

---

6. EXTI 인터럽트 벡터와 NVIC

EXTI 라인	NVIC 인터럽트 이름	비고
EXTI0	`EXTI0_IRQn`	단독 라인
EXTI1	`EXTI1_IRQn`	단독 라인
EXTI2	`EXTI2_IRQn`	단독 라인

EXTI 라인	NVIC 인터럽트 이름	비고
EXTI3	`EXTI3_IRQn`	단독 라인
EXTI4	`EXTI4_IRQn`	단독 라인
EXTI5~9	`EXTI9_5_IRQn`	5~9 공유
EXTI10~15	`EXTI15_10_IRQn`	10~15 공유

> STM32F1은 EXTI0~4까지는 개별 인터럽트,
> 5~9, 10~15는 그룹 인터럽트로 묶여 처리된다.

---

7. 실습 예: PA0 버튼 → EXTI0 인터럽트

(1) 회로

- PA0 : 버튼 (Pull-up, Active-Low)
- PC13 : LED 출력

(2) 코드

```
void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_AFIO_CLK_ENABLE();

  // LED (PC13)
  GPIO_InitStruct.Pin = GPIO_PIN_13;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  // BUTTON (PA0 → EXTI0)
  GPIO_InitStruct.Pin = GPIO_PIN_0;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;   // Falling Edge 감지
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  // NVIC 설정
  HAL_NVIC_SetPriority(EXTI0_IRQn, 2, 0);
  HAL_NVIC_EnableIRQ(EXTI0_IRQn);
}

// 인터럽트 콜백 함수
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if (GPIO_Pin == GPIO_PIN_0)
```

```
  {
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13); // LED 토글
  }
}

// 인터럽트 핸들러
void EXTI0_IRQHandler(void)
{
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
}
```

(3) 동작

- 버튼을 누르면 PA0 → Low → EXTI0 트리거
- NVIC가 `EXTI0_IRQHandler()` 호출
- HAL이 내부적으로 `HAL_GPIO_EXTI_Callback()` 실행
- LED 토글 발생

---

8. 공유 인터럽트 (예: EXTI9_5_IRQn)

예: PA6, PB7을 동시에 EXTI 입력으로 설정할 경우

- 두 핀 모두 `EXTI9_5_IRQn` 벡터에 매핑됨.
- ISR 내부에서 **어떤 핀에서 발생했는지 구분**해야 함.

```
void EXTI9_5_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_6);
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_7);
}
```

→ 이후 `HAL_GPIO_EXTI_Callback()` 내에서 핀 번호로 구분:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if (GPIO_Pin == GPIO_PIN_6)
        // 처리1
    else if (GPIO_Pin == GPIO_PIN_7)
        // 처리2
}
```

---

9. 레지스터 수준 동작

(1) Falling/Rising Edge 설정

레지스터	역할
`EXTI_RTSR`	Rising Edge 감지
`EXTI_FTSR`	Falling Edge 감지

(2) 인터럽트 허용

레지스터	역할
`EXTI_IMR`	Interrupt Mask Register — 1: 허용
`EXTI_EMR`	Event Mask Register — 이벤트 출력용

(3) Pending 처리

레지스터	역할
`EXTI_PR`	Pending Register — 인터럽트 발생 표시(1로 set)

> ISR 내에서 `EXTI_PR`**의 해당 비트를 1로 써서 클리어**해야 인터럽트 재발생이 가능.

---

10. 전체 요약

구분	설명
EXTI 라인 수	0~15 (총 16개)
매핑 기준	핀 번호(Pin Index)
포트 선택	AFIO_EXTICR[4] 레지스터 (A~E 선택)
NVIC 벡터	EXTI0~4 단독, EXTI5~9 / 10~15 그룹
엣지 감지	Rising, Falling, 또는 Both
활성화 절차	① AFIO 매핑 → ② EXTI 설정 → ③ NVIC Enable
대표 사용 예	버튼, 센서 트리거, 외부 이벤트 감지

---

11. 결론

`EXTI` 는 STM32의 **외부 신호 감지 핵심 하드웨어**로,
각 핀의 번호에 대응하는 EXTI 라인(0~15)을 통해
Rising/Falling 이벤트를 인터럽트로 직접 처리할 수 있다.

즉, PA0~PA15 핀은 각각 EXTI0~EXTI15 라인에 연결 가능하며,
포트 선택은 **AFIO_EXTICR** 레지스터로 결정된다.

이 구조를 이해하면 버튼 입력뿐만 아니라
센서 트리거, 통신 신호, 외부 동기화 등
모든 이벤트 기반 제어를 정밀하게 설계할 수 있다.

- # NVIC 우선순위 설정

1. 개요

STM32F103의 **NVIC (Nested Vectored Interrupt Controller)** 는
ARM Cortex-M3 코어 내부에 포함된 **인터럽트 관리 유닛**으로,
여러 인터럽트가 동시에 발생했을 때 **우선순위(Priority)** 를 기준으로
CPU가 어떤 인터럽트를 먼저 처리할지를 결정한다.

즉, **NVIC는 인터럽트 요청을 관리하고, 중첩(우선순위 기반) 실행을 제어하는 컨트롤러**이다.

---

2. NVIC의 역할

1. **인터럽트 허용 / 비허용 제어 (Enable/Disable)**
2. **인터럽트 우선순위 설정 (Priority Assignment)**
3. **중첩 인터럽트 처리 (Nested Interrupt Handling)**
4. **인터럽트 Pending/Active 상태 관리**

NVIC는 **256개의 인터럽트 우선순위 레벨**을 지원하지만,
STM32F1 시리즈에서는 **최대 4비트(0~15)** 만 실제로 사용된다.
(HAL에서는 이를 **Preemption Priority(선점 우선순위)** 와 **Sub Priority(서브 우선순위)** 로 분리 설정 가능)

---

3. NVIC의 우선순위 개념

(1) Preemption Priority (선점 우선순위)

- 높은 우선순위를 가진 인터럽트가 낮은 우선순위 ISR을 **중단하고 진입할 수 있음.**
- 즉, **인터럽트 간 중첩 허용** 여부를 결정.

(2) Sub Priority (서브 우선순위)

- 동일한 Preemption Priority 내에서 **동시 발생한 인터럽트 간의 처리 순서**를 결정.
- **선점은 불가능**, 단지 **순서 결정용**.

(3) 우선순위 숫자 규칙

- **숫자가 작을수록 우선순위가 높음.**
  예: `0` > `1` > `2` > `3` ...

---

4. NVIC 우선순위 그룹 (Priority Grouping)

NVIC는 **우선순위 비트를 Preemption/Sub로 나누는 비율**을 설정할 수 있다.

그룹	선점비트	서브비트	설명
`NVIC_PRIORITYGROUP_0`	0	4	모두 서브우선순위 (중첩 불가)
`NVIC_PRIORITYGROUP_1`	1	3	
`NVIC_PRIORITYGROUP_2`	2	2	
`NVIC_PRIORITYGROUP_3`	3	1	
`NVIC_PRIORITYGROUP_4`	4	0	모두 선점우선순위 (서브 없음)

> STM32 HAL에서는 기본적으로 `NVIC_PRIORITYGROUP_2` (2비트 + 2비트)가 기본 설정된다.

---

5. HAL에서 NVIC 설정 함수

(1) 인터럽트 우선순위 설정

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

- `IRQn` : 인터럽트 이름 (`EXTI0_IRQn`, `TIM2_IRQn`, `USART1_IRQn`, 등)
- `PreemptPriority` : 선점 우선순위 (0이 가장 높음)
- `SubPriority` : 서브 우선순위

(2) 인터럽트 활성화

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

(3) 예시

```
HAL_NVIC_SetPriority(EXTI0_IRQn, 1, 0);  // EXTI0: Preempt 1, Sub 0
HAL_NVIC_EnableIRQ(EXTI0_IRQn);

HAL_NVIC_SetPriority(EXTI1_IRQn, 0, 0);  // EXTI1: Preempt 0 → 더 높은 우선순위
HAL_NVIC_EnableIRQ(EXTI1_IRQn);
```

---

6. 실제 예시: 버튼 + 타이머 인터럽트

(1) 시나리오

- **PA0 (EXTI0)**: 버튼 눌림 인터럽트
- **TIM2**: 1초 주기 타이머 인터럽트

버튼 인터럽트가 타이머보다 **더 높은 우선순위**를 갖도록 설정한다.

(2) 코드

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

    // TIM2 Interrupt
    HAL_NVIC_SetPriority(TIM2_IRQn, 1, 0);  // 낮은 우선순위
    HAL_NVIC_EnableIRQ(TIM2_IRQn);

    // EXTI0 Interrupt (Button)
    HAL_NVIC_SetPriority(EXTI0_IRQn, 0, 0);  // 높은 우선순위
    HAL_NVIC_EnableIRQ(EXTI0_IRQn);
}
```

(3) 동작 순서

- TIM2 ISR 실행 중 버튼 인터럽트 발생 시,
  EXTI0 (Preempt=0)이 TIM2 ISR(Preempt=1)을 **즉시 중단하고 진입.**

> → 높은 실시간 응답이 필요한 이벤트(예: 긴급 정지, 센서 트리거)는 낮은 숫자의 우선순위로 설정해야 한다.

---

7. NVIC 레지스터 구조 (요약)

레지스터	기능
`NVIC_ISERx`	Interrupt Set Enable Register — 인터럽트 허용
`NVIC_ICERx`	Interrupt Clear Enable Register — 비허용
`NVIC_ISPRx`	Interrupt Set Pending Register
`NVIC_ICPRx`	Interrupt Clear Pending Register
`NVIC_IPRx`	Interrupt Priority Register (8bit × n)

- STM32F103은 60개 이상의 IRQ 지원 (IRQn = 0~59)
- 각 IRQ의 우선순위는 `NVIC_IPR[x]`에서 개별적으로 설정됨

---

8. HAL 기본 예제 (EXTI0)

```
void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_AFIO_CLK_ENABLE();

  GPIO_InitStruct.Pin = GPIO_PIN_0;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  // NVIC 설정
  HAL_NVIC_SetPriority(EXTI0_IRQn, 2, 0);  // 낮은 우선순위
  HAL_NVIC_EnableIRQ(EXTI0_IRQn);
}
```

> `HAL_NVIC_SetPriority()` 와 `HAL_NVIC_EnableIRQ()` 는 항상 쌍으로 사용된다.
> 하나라도 빠지면 인터럽트가 발생하지 않는다.

---

9. NVIC 우선순위 확인 및 디버깅 팁

1. **Priority Group 확인**

```
uint32_t group = NVIC_GetPriorityGrouping();
```

2. **현재 설정 확인**

```
uint32_t preempt = NVIC_GetPriority(EXTI0_IRQn);
```

3. **중첩 인터럽트 테스트**
   - 타이머 ISR 안에서 EXTI를 트리거해보고 선점 동작 관찰.
4. **중요한 ISR은 짧게**
   - NVIC는 ISR 중첩을 허용하지만, **장시간 ISR은 다른 인터럽트를 지연시킨다.**

---

10. FreeRTOS 환경에서 주의사항

FreeRTOS에서는 NVIC 우선순위가 **커널과 연동**되므로
아래 제한이 반드시 지켜져야 한다.

- 커널에서 사용하는 우선순위보다 높은 ISR에서는
  FreeRTOS API 호출 금지 (`xQueueSendFromISR()` 등은 예외)
- `configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY` 값 이하로 설정해야 함.

> 예: `configLIBRARY_MAX_SYSCALL_INTERRUPT_PRIORITY = 5`
> → 5보다 **큰 숫자**(우선순위 낮은 인터럽트)에서만 FreeRTOS API 호출 가능.

---

11. NVIC 우선순위 설계 예시

인터럽트	목적	Preemption	Sub	비고
Emergency Stop (EXTI0)	안전 정지	0	0	최고 우선순위
Sensor Trigger (EXTI5)	측정 트리거	1	0	
UART RX	데이터 수신	2	0	
Timer2	주기적 Task	3	0	
SysTick	OS Tick	15	0	기본값 (가장 낮음)

> 인터럽트 수가 많을수록 우선순위를 **계층적 설계**로 관리해야 한다.

---

12. 결론

`NVIC`는 STM32 인터럽트 시스템의 "심장부"로,
모든 EXTI, Timer, UART, DMA 등 **외부·내부 이벤트의 실행 순서를 결정**한다.

핵심은 다음과 같다:

1. **숫자가 작을수록 우선순위가 높다.**
2. **Preemption Priority**는 중첩 가능성,
   Sub Priority는 동일 레벨 내 순서만 결정한다.
3. `HAL_NVIC_SetPriority()` 와 `HAL_NVIC_EnableIRQ()` 를 반드시 함께 사용한다.
4. 실시간 응답이 중요한 인터럽트는 **가장 낮은 숫자의 우선순위**를 부여한다.

한 줄 요약:

> NVIC 우선순위 설정은 STM32의 인터럽트 아키텍처에서
> "**누가 먼저 CPU를 차지할 것인가**"를 정하는 규칙이며,
> 시스템의 실시간성과 안정성을 결정짓는 핵심 설계 포인트이다.

- # HAL_GPIO_EXTI_Callback() 구조

1. 개요

`HAL_GPIO_EXTI_Callback()` 은
STM32 HAL 라이브러리에서 **외부 인터럽트(EXTI)** 가 발생했을 때
사용자가 정의할 수 있는 **콜백 함수(Callback Function)** 이다.

즉, **GPIO 핀의 Falling/Rising Edge 신호가 감지되면**
HAL 인터럽트 처리 루틴이 자동으로 이 함수를 호출하여
개발자가 직접 작성한 동작(예: LED 토글, 센서 읽기 등)을 수행할 수 있게 한다.

2. 전체 인터럽트 흐름 (EXTI 동작 구조)

(1) 하드웨어 → 소프트웨어 동작 시퀀스

```
외부 이벤트 발생 (버튼, 센서 등)
      |
      ▼
GPIO 핀 상태 변화 (Rising/Falling)
      |
      ▼
EXTI 라인 감지 (EXTI_PR 비트 세트)
      |
      ▼
NVIC 인터럽트 요청 발생 (IRQn)
      |
      ▼
해당 IRQHandler() 실행 (예: EXTI0_IRQHandler)
      |
      ▼
HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_x)
      |
      ▼
HAL_GPIO_EXTI_Callback(GPIO_PIN_x)
      |
      ▼
사용자 정의 코드 실행
```

> 즉, `HAL_GPIO_EXTI_Callback()`은 **HAL 계층의 마지막 단계**이며,
> 사용자가 실제 동작을 구현하는 **핸들러의 진입점**이다.

3. 호출 경로 상세 구조

(1) NVIC 인터럽트 핸들러

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

> 모든 EXTI IRQ 핸들러(`EXTI0_IRQHandler`, `EXTI1_IRQHandler`, `EXTI9_5_IRQHandler` 등)는
> 공통적으로 `HAL_GPIO_EXTI_IRQHandler()`를 호출한다.

(2) HAL 인터럽트 핸들러 내부 구조

stm32f1xx_hal_gpio.c

```c
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
{
  // 인터럽트 펜딩 클리어 (Pending Flag Clear)
  if (__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != RESET)
  {
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);   // EXTI_PR 비트 클리어

    // 사용자 콜백 함수 호출
    HAL_GPIO_EXTI_Callback(GPIO_Pin);
  }
}
```

> `__HAL_GPIO_EXTI_GET_IT()` : 해당 EXTI 라인에서 인터럽트가 발생했는지 확인
> `__HAL_GPIO_EXTI_CLEAR_IT()` : Pending Register (`EXTI_PR`) 비트를 1로 써서 클리어
> `HAL_GPIO_EXTI_Callback()` : 사용자 정의 함수 호출

---

(3) 사용자 정의 콜백 함수

```c
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if (GPIO_Pin == GPIO_PIN_0)
    {
        // 버튼 눌림 이벤트 처리
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
    }
}
```

- `GPIO_Pin` 인자는 어떤 핀에서 인터럽트가 발생했는지를 알려준다.
- 여러 핀이 같은 EXTI 그룹(예: EXTI9_5_IRQn)을 공유할 때 **이 인자로 구분**한다.

---

4. 인터럽트 공유 구조 (EXTI9_5, EXTI15_10)

(1) NVIC 핸들러

```c
void EXTI9_5_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_6);
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_7);
}
```

(2) 콜백에서 구분 처리

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    switch(GPIO_Pin)
    {
        case GPIO_PIN_6:
            // 센서 1 감지
            break;

        case GPIO_PIN_7:
            // 센서 2 감지
            break;
    }
}
```

> 이렇게 하면 여러 핀이 동일한 인터럽트 벡터를 공유하더라도
> 개별 이벤트를 독립적으로 처리할 수 있다.

---

5. 콜백 함수의 위치

> `HAL_GPIO_EXTI_Callback()` 함수는 HAL 라이브러리 내부에 **약한(weak)** 심볼로 선언되어 있다.

stm32f1xx_hal_gpio.c

```
__weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  /* Prevent unused argument(s) compilation warning */
  UNUSED(GPIO_Pin);
  /* NOTE: This function should not be modified, when the callback is needed,
           the HAL_GPIO_EXTI_Callback could be implemented in the user file
   */
}
```

- `__weak` 키워드는 “약한 링크(Weak Linkage)”를 의미한다.
- 즉, 사용자가 같은 이름의 함수를 정의하면 **자동으로 오버라이드(override)** 된다.

📘 **정리:**
HAL 내부 함수는 수정하지 말고,
사용자 코드(`main.c` 또는 `stm32f1xx_it.c`)에 같은 이름으로 함수를 작성해야 한다.

---

6. 전체 코드 예제 (PA0 → EXTI0 → LED 제어)

(1) 회로 구성

핀	기능	설명
PA0	버튼 입력	Pull-up, Active-Low

핀	기능	설명
PC13	LED 출력	Active-Low

(2) 코드 예시

```c
#include "main.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();

  while (1)
  {
    // 메인 루프는 비워둠
  }
}

static void MX_GPIO_Init(void)
{
  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_AFIO_CLK_ENABLE();

  GPIO_InitTypeDef GPIO_InitStruct = {0};

  // LED 출력 (PC13)
  GPIO_InitStruct.Pin = GPIO_PIN_13;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  // 버튼 입력 (PA0) → EXTI0
  GPIO_InitStruct.Pin = GPIO_PIN_0;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  // NVIC 설정
  HAL_NVIC_SetPriority(EXTI0_IRQn, 2, 0);
  HAL_NVIC_EnableIRQ(EXTI0_IRQn);
}

// NVIC 핸들러
void EXTI0_IRQHandler(void)
{
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
```

```
}

// HAL 콜백 함수
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if (GPIO_Pin == GPIO_PIN_0)
  {
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
  }
}
```

(3) 동작

- 버튼을 누르면 PA0 → GND로 연결되어 Falling Edge 발생
- `EXTI0_IRQHandler()` → `HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0)` → `HAL_GPIO_EXTI_Callback(GPIO_PIN_0)` 호출
- LED 상태 토글

---

7. 디바운스 및 이벤트 확장 예시

콜백 함수 내부에서 **디바운스 또는 이벤트 큐 전달**도 가능하다.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if (GPIO_Pin == GPIO_PIN_0)
  {
    // 20ms 디바운스
    HAL_Delay(20);
    if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
    {
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
    }
  }
}
```

또는 FreeRTOS 사용 시:

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if (GPIO_Pin == GPIO_PIN_0)
  {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    xTaskNotifyFromISR(ButtonTaskHandle, 0, eNoAction, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
  }
}
```

---

8. 주의사항

항목	설명
ISR 내에서 HAL_Delay() 사용 금지	블로킹 함수이므로 시스템 타이밍 깨짐
콜백 내 연산 최소화	ISR은 가능한 한 짧게 유지해야 함
공유 인터럽트 구분 필수	`GPIO_Pin` 인자로 반드시 분기
다른 인터럽트 호출 가능	NVIC 우선순위 조절 필요
Weak 함수 중복 정의 금지	한 프로젝트 내에 2개 이상 정의 시 링크 에러 발생

---

9. 전체 호출 관계 요약

```
[EXTI Event Detected]
        ↓
EXTI Controller → NVIC IRQn
        ↓
<stm32f1xx_it.c>
  EXTIx_IRQHandler()
        ↓
HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_x)
        ↓
HAL_GPIO_EXTI_Callback(GPIO_PIN_x)
        ↓
<사용자 코드>
  ├─ LED 제어
  ├─ 센서 트리거 처리
  ├─ FreeRTOS Task Notify
  └─ 디바운스 / 데이터 로깅
```

---

10. 결론

`HAL_GPIO_EXTI_Callback()` 은 HAL 레벨에서 제공하는
EXTI 인터럽트 사용자 진입점으로,
개발자가 직접 이벤트 기반 코드를 작성하는 핵심 함수이다.

핵심 요약:

항목	설명
호출 시점	EXTI 인터럽트 발생 후 HAL 처리 완료 시점
입력 인자	`GPIO_Pin` (예: `GPIO_PIN_0`, `GPIO_PIN_13`)
정의 위치	`main.c` (사용자 정의)
내부 동작	HAL이 Pending Flag 클리어 후 자동 호출

항목	설명
주요 역할	이벤트 처리, LED 제어, Task Notify 등

결론적으로,

> `HAL_GPIO_EXTI_Callback()`은 STM32 외부 인터럽트의 **사용자 정의 처리 루틴의 엔트리 포인트**이며,
> HAL 레이어가 인터럽트의 모든 레지스터 및 Pending 처리를 대신 수행한 뒤
> 최종적으로 개발자 코드가 실행되는 **안전하고 표준화된 인터페이스**이다.

- # 버튼 인터럽트 및 LED 토글 실습

1. 실습 개요

이번 실습은 **외부 인터럽트(EXTI)** 를 이용해
버튼 입력 시 LED를 토글하는 가장 기본적인 STM32 인터럽트 제어 예제이다.

이 과정을 통해

- GPIO 입력 핀을 EXTI 라인으로 설정하고
- Falling Edge(버튼 눌림) 이벤트를 감지하며
- NVIC 우선순위와 콜백 함수를 활용해
- LED를 인터럽트 기반으로 제어하는 전 과정을 학습한다.

---

2. 하드웨어 구성

구성요소	포트	설정	설명
LED	PC13	Output Push-Pull	Blue Pill 기본 내장 LED (Active-Low)
버튼	PA0	Input with Pull-Up	눌렀을 때 GND로 연결 (Active-Low)

회로 연결도 (개념)

```
3.3V
 |
[10kΩ]
 |
 ├──────> PA0 (EXTI0 Input)
 |
[Button]
 |
GND

PC13 ─── LED ─── GND
```

---

3. 핵심 동작 흐름

```
버튼 누름 (PA0 = LOW)
      ↓
EXTI0 라인 트리거 (Falling Edge)
      ↓
NVIC: EXTI0_IRQn 인터럽트 발생
      ↓
HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0)
      ↓
HAL_GPIO_EXTI_Callback(GPIO_PIN_0)
      ↓
사용자 코드: LED 토글 (HAL_GPIO_TogglePin)
```

---

4. CubeMX 또는 코드 수동 설정

(1) RCC 클록 활성화

```
__HAL_RCC_GPIOA_CLK_ENABLE();
__HAL_RCC_GPIOC_CLK_ENABLE();
__HAL_RCC_AFIO_CLK_ENABLE();
```

(2) LED 핀 초기화 (출력)

```
GPIO_InitTypeDef GPIO_InitStruct = {0};

GPIO_InitStruct.Pin = GPIO_PIN_13;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
```

(3) 버튼 핀 초기화 (EXTI 입력)

```
GPIO_InitStruct.Pin = GPIO_PIN_0;
GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;  // Falling Edge에서 인터럽트 발생
GPIO_InitStruct.Pull = GPIO_PULLUP;           // 평상시 High 유지
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

(4) NVIC 설정

```
HAL_NVIC_SetPriority(EXTI0_IRQn, 2, 0);  // 우선순위 설정
HAL_NVIC_EnableIRQ(EXTI0_IRQn);          // 인터럽트 활성화
```

---

5. 인터럽트 핸들러 연결

stm32f1xx_it.c

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

> HAL이 내부적으로 EXTI Pending 비트를 클리어하고
> 자동으로 `HAL_GPIO_EXTI_Callback()`을 호출한다.

---

6. 콜백 함수 구현

main.c

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if (GPIO_Pin == GPIO_PIN_0)
    {
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);  // LED 토글
    }
}
```

---

7. 전체 예제 코드 (main.c)

```
#include "main.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();

  while (1)
  {
    // 메인 루프는 비워둠 (인터럽트 기반 동작)
  }
}

static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOC_CLK_ENABLE();
```

```
  __HAL_RCC_AFIO_CLK_ENABLE();

  // LED (PC13)
  GPIO_InitStruct.Pin = GPIO_PIN_13;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

  // BUTTON (PA0) → EXTI0
  GPIO_InitStruct.Pin = GPIO_PIN_0;
  GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

  // NVIC 설정
  HAL_NVIC_SetPriority(EXTI0_IRQn, 2, 0);
  HAL_NVIC_EnableIRQ(EXTI0_IRQn);
}

// NVIC 인터럽트 핸들러
void EXTI0_IRQHandler(void)
{
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
}

// 콜백 함수
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if (GPIO_Pin == GPIO_PIN_0)
  {
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
  }
}
```

8. 동작 결과

동작	설명
버튼을 누름 (PA0 → LOW)	EXTI0 인터럽트 발생
HAL 인터럽트 루틴 실행	EXTI Pending 클리어 + 콜백 호출
콜백 내부에서 LED 토글	LED가 ON ↔ OFF 전환
버튼을 여러 번 누름	LED가 반복적으로 토글됨

> ⚠️ 버튼이 기계식일 경우, 바운스 현상이 발생할 수 있음 → 디바운스 루틴 병행 필요 (앞선 2.8 참고)

9. 디바운스 포함 확장 예시

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if (GPIO_Pin == GPIO_PIN_0)
  {
    HAL_Delay(20); // 디바운스 대기
    if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
    {
      HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
    }
  }
}
```

---

10. FreeRTOS 환경 확장 예시

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if (GPIO_Pin == GPIO_PIN_0)
  {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    xTaskNotifyFromISR(LEDTaskHandle, 0, eNoAction, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
  }
}
```

> 버튼 누름 → ISR → RTOS Task Notify → LED Task에서 토글 수행
> → 인터럽트는 빠르고 Task는 안정적

---

11. NVIC 및 EXTI 동작 요약

구분	설정 대상	역할
AFIO	EXTI 라인 포트 매핑	PA0 ↔ EXTI0 연결
EXTI_RTSR / FTSR	엣지 감지 설정	Rising/Falling 선택
EXTI_IMR	인터럽트 활성화	1 → Enable
NVIC	인터럽트 우선순위 및 허용	`HAL_NVIC_EnableIRQ()`
콜백 함수	사용자 동작 정의	`HAL_GPIO_EXTI_Callback()`

---

12. 실무 응용 포인트

응용 예	설명
긴급 정지 버튼 (E-Stop)	EXTI + 높은 NVIC 우선순위
센서 트리거 입력	Falling Edge 이벤트 감지
Encoder Phase 입력	Dual EXTI 라인 사용
Power 버튼 제어	EXTI + Sleep 모드 연계
디버그용 버튼	인터럽트로 상태 토글 및 로그 출력

---

13. 결론

이 예제는 STM32 EXTI 시스템의 핵심 구조를 모두 포함한다:

1. **GPIO → EXTI 라인 매핑 (AFIO)**
2. **Rising/Falling Edge 감지**
3. **NVIC 우선순위 설정 및 인터럽트 허용**
4. **HAL 핸들러 → 콜백 호출 → 사용자 정의 처리**

결국, **"버튼 입력 → 인터럽트 → LED 토글"** 은
STM32 외부 이벤트 처리의 가장 기본적인 학습 예제이며,
이를 기반으로 타이머, 센서, 통신 인터럽트 등으로 확장할 수 있다.

✅ 핵심 요약

항목	설정
입력 핀	PA0 (EXTI0, Pull-up, Falling Edge)
출력 핀	PC13 (LED, Push-Pull)
인터럽트	`EXTI0_IRQn`, NVIC Priority 2
함수 흐름	`EXTI0_IRQHandler()` → `HAL_GPIO_EXTI_IRQHandler()` → `HAL_GPIO_EXTI_Callback()`
결과	버튼 누를 때마다 LED 토글

2.4 실습

- # LED 제어

1. 개요

STM32에서 LED 제어는 **GPIO 출력 제어(Output Control)** 의 대표적인 예제이다.
이 실습에서는 HAL 라이브러리를 이용해 **LED를 점멸(ON/OFF, 토글)** 하는 과정을 다룬다.

GPIO 출력은 MCU에서 외부로 신호를 보내는 방식이며,
디지털 제어의 기본 단위로 모터, 릴레이, 센서 트리거 등 모든 출력 장치 제어의 기초가 된다.

---

2. 하드웨어 구성

구성요소	포트	설정	설명
LED	PC13	Output Push-Pull	Blue Pill 기본 내장 LED (Active-Low)

> Blue Pill의 내장 LED는 **VCC → 저항 → LED → PC13 → GND** 형태로 연결되어 있으며,
> 즉, **PC13을 LOW로 출력하면 LED가 켜지고**, HIGH일 때 꺼진다.

회로 요약

```
3.3V ──[저항]──▶ LED ▶── PC13
```

PC13 출력	LED 상태
HIGH (1)	꺼짐
LOW (0)	켜짐

---

3. GPIO 출력 핀 설정

(1) 클록 활성화

```
__HAL_RCC_GPIOC_CLK_ENABLE();
```

(2) 핀 모드 설정

```
GPIO_InitTypeDef GPIO_InitStruct = {0};
GPIO_InitStruct.Pin = GPIO_PIN_13;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;   // Push-Pull 출력
GPIO_InitStruct.Pull = GPIO_NOPULL;           // 내부 풀업/풀다운 없음
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;  // 출력 속도 설정
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
```

(3) 출력 초기화

```
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET); // LED Off
```

---

4. 제어 함수

함수	설명
`HAL_GPIO_WritePin(GPIOx, Pin, State)`	지정 핀을 High/Low로 설정
`HAL_GPIO_TogglePin(GPIOx, Pin)`	핀의 현재 상태를 반전시킴
`HAL_GPIO_ReadPin(GPIOx, Pin)`	현재 입력 상태를 읽음 (출력핀에도 사용 가능)

(1) ON/OFF 제어

```
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_RESET); // LED ON
HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);   // LED OFF
```

(2) 토글 제어

```
HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);  // ON ↔ OFF 반전
```

---

5. 전체 예제 코드

```
#include "main.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();

  while (1)
  {
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);  // LED 상태 반전
    HAL_Delay(500);                          // 500ms 지연 (0.5초)
  }
}

static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  __HAL_RCC_GPIOC_CLK_ENABLE();
```

```
24
25    // LED (PC13)
26    GPIO_InitStruct.Pin = GPIO_PIN_13;
27    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
28    GPIO_InitStruct.Pull = GPIO_NOPULL;
29    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
30    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
31
32    // 초기 상태: LED OFF
33    HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);
34  }
```

(실행 결과)

- 0.5초 간격으로 LED 깜박임 (ON ↔ OFF)
- 전원 공급 후 자동 반복

---

6. 코드 동작 분석

구문	설명
`HAL_Init()`	HAL 라이브러리 초기화
`SystemClock_Config()`	시스템 클록 구성
`MX_GPIO_Init()`	GPIO 핀 초기화 (출력 설정)
`HAL_GPIO_TogglePin()`	LED 상태 반전 (1 → 0 또는 0 → 1)
`HAL_Delay(500)`	소프트웨어 지연 (SysTick 기반 500ms)

> SysTick 타이머는 1ms 단위로 HAL 내부에서 자동 설정되어 있으므로
> `HAL_Delay()` 함수는 실제 시간 기반 LED 점멸을 구현할 수 있다.

---

7. 확장 예제 – 버튼 입력과 결합

앞선 **EXTI 버튼 인터럽트**와 결합하여
버튼을 누를 때마다 LED를 제어할 수도 있다.

```
1  void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
2  {
3    if (GPIO_Pin == GPIO_PIN_0)
4    {
5      HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
6    }
7  }
```

> 버튼 눌림(Falling Edge) → EXTI 인터럽트 발생 → LED 토글

---

8. 전류 및 보호 회로 참고

항목	권장값	설명
LED 전류	5~10mA	STM32 GPIO의 Sink 전류 제한 내
저항 값	330Ω~1kΩ	LED 보호용 시리즈 저항
출력 전류 제한	GPIO 당 최대 20mA, 포트당 총합 80mA	

> 고전류 구동이 필요한 LED나 릴레이의 경우 **트랜지스터 드라이버**를 병행해야 한다.

---

9. 주의사항

항목	설명
Active-Low 설계	PC13은 내부 풀업 회로 구조상 LED가 Active-Low임
지연 시간	`HAL_Delay()` 는 SysTick 기반으로 RTOS 환경에서는 사용 자제
GPIOC 클록 누락 금지	`__HAL_RCC_GPIOC_CLK_ENABLE()` 반드시 필요
전원 핀 확인	Blue Pill은 일부 보드에서 LED 극성이 반대인 경우 존재
전류 초과 주의	직접 LED 구동 시 10mA 이내 유지

---

10. FreeRTOS 환경 예시

```
void vLEDTask(void *pvParameters)
{
  for(;;)
  {
    HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
    vTaskDelay(pdMS_TO_TICKS(500));  // 500ms 주기
  }
}
```

> 타이머 인터럽트나 FreeRTOS Task로 LED를 주기 제어하면
> **CPU 부하 없이 비동기 점멸 구현** 가능.

---

11. 결론

- **GPIO 출력 제어**는 STM32 하드웨어 제어의 가장 기본 단위이다.
- 단일 LED 제어를 통해, 출력 레벨 제어, 지연, 인터럽트, 타이밍 제어의 전체 원리를 배울 수 있다.
- 이후 이 원리는 **릴레이, 모터, 신호등, 센서 트리거, PWM** 등 모든 제어 로직의 기반이 된다.

✅ 핵심 요약

항목	내용
제어 대상	PC13 (LED, Active-Low)
출력 모드	Push-Pull
동작 방식	HAL_GPIO_TogglePin() / HAL_GPIO_WritePin()
동작 예	500ms 간격 LED 점멸
확장	버튼 인터럽트, FreeRTOS Task, PWM 제어

- # 스위치 인터럽트로 펌프 ON/OFF

1. 개요

본 실습은 **외부 스위치 입력(EXTI)** 을 이용해
릴레이(또는 MOSFET)를 통해 펌프를 ON/OFF 제어하는 예제이다.

즉,

- 사용자가 버튼을 누르면 → EXTI 인터럽트 발생
- 인터럽트 콜백에서 펌프 구동 핀을 토글하여
- **전원 장치(펌프, 모터, 밸브 등)** 의 작동을 제어한다.

---

2. 하드웨어 구성

구성 요소	포트	설정	설명
스위치	PA0	EXTI 입력 (Pull-up, Falling Edge)	누름 시 GND로 연결
펌프 릴레이	PB12	Output Push-Pull	릴레이 모듈 제어 (Active-High)
전원	12V	펌프 구동용 외부 전원	
MCU 전원	3.3V	Blue Pill 구동 전원	

릴레이 회로 예시

```
STM32 PB12 ─▶ 릴레이 IN
릴레이 VCC ─▶ 5V
릴레이 GND ─▶ GND (STM32 공통 접지)
릴레이 NO ─▶ 펌프 (+)
릴레이 COM ─▶ 12V 전원 +
펌프 - ──────▶ GND
```

스위치 회로 예시

```
3.3V
 |
[10kΩ]
 |
 ├──> PA0 (EXTI0 Input)
 |
[Button]
 |
GND
```

> 버튼 누름 → PA0 = LOW → EXTI0 인터럽트 발생

---

3. 동작 흐름

```
버튼 누름 (Falling Edge)
      ↓
EXTI0_IRQHandler() 실행
      ↓
HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0)
      ↓
HAL_GPIO_EXTI_Callback(GPIO_PIN_0)
      ↓
현재 릴레이 상태 읽기 → 반전 (ON ↔ OFF)
      ↓
펌프 전원 ON/OFF 제어
```

---

4. 펌프 제어 논리

조건	PB12 출력	릴레이 상태	펌프
ON 상태	HIGH	접점 연결	작동
OFF 상태	LOW	접점 해제	정지

> 대부분의 **Active-High 릴레이 모듈**은 IN 핀에 HIGH 신호를 주면 펌프가 동작한다.

---

5. GPIO 및 EXTI 설정

(1) 클록 설정

```
__HAL_RCC_GPIOA_CLK_ENABLE();
__HAL_RCC_GPIOB_CLK_ENABLE();
__HAL_RCC_AFIO_CLK_ENABLE();
```

(2) 스위치 입력 (PA0)

```
GPIO_InitTypeDef GPIO_InitStruct = {0};

GPIO_InitStruct.Pin = GPIO_PIN_0;
GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;  // Falling Edge
GPIO_InitStruct.Pull = GPIO_PULLUP;           // 평상시 High 유지
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

(3) 릴레이 출력 (PB12)

```
GPIO_InitStruct.Pin = GPIO_PIN_12;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
```

(4) NVIC 설정

```
HAL_NVIC_SetPriority(EXTI0_IRQn, 2, 0);
HAL_NVIC_EnableIRQ(EXTI0_IRQn);
```

---

6. 전체 코드 예제

main.c

```
#include "main.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
  HAL_Init();
  SystemClock_Config();
  MX_GPIO_Init();

  while (1)
  {
    // 인터럽트 기반 동작 → 메인 루프는 비워둠
  }
}

static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};

  __HAL_RCC_GPIOA_CLK_ENABLE();
  __HAL_RCC_GPIOB_CLK_ENABLE();
  __HAL_RCC_AFIO_CLK_ENABLE();
```

```
25
26    // 릴레이 제어핀 (PB12)
27    GPIO_InitStruct.Pin = GPIO_PIN_12;
28    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
29    GPIO_InitStruct.Pull = GPIO_NOPULL;
30    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
31    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
32
33    // 초기 상태: 펌프 OFF
34    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_RESET);
35
36    // 스위치 입력 (PA0)
37    GPIO_InitStruct.Pin = GPIO_PIN_0;
38    GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
39    GPIO_InitStruct.Pull = GPIO_PULLUP;
40    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
41
42    // NVIC 설정
43    HAL_NVIC_SetPriority(EXTI0_IRQn, 2, 0);
44    HAL_NVIC_EnableIRQ(EXTI0_IRQn);
45  }
46
47  // NVIC 인터럽트 핸들러
48  void EXTI0_IRQHandler(void)
49  {
50    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);
51  }
52
53  // EXTI 콜백 함수
54  void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
55  {
56    if (GPIO_Pin == GPIO_PIN_0)
57    {
58      HAL_Delay(30); // 디바운스
59      if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_RESET)
60      {
61        HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_12); // 펌프 ON/OFF 토글
62      }
63    }
64  }
```

---

7. 동작 결과

동작	설명
버튼 누름	EXTI0 발생 (PA0 → GND)
인터럽트 처리	콜백 진입 → PB12 상태 반전
릴레이 구동	PB12 HIGH → 릴레이 ON → 펌프 작동
버튼 다시 누름	PB12 LOW → 릴레이 OFF → 펌프 정지

> 결과적으로, 버튼 입력만으로 **토글 방식의 펌프 제어**가 구현된다.

---

8. 펌프 구동 전류 주의

항목	권장값	설명
GPIO 최대 출력	20mA	직접 펌프 구동 불가
릴레이 IN 전류	5~15mA	GPIO 직접 구동 가능
펌프 전류	수백 mA ~ 수 A	별도 릴레이 또는 트랜지스터 필요

> ⚠️ 펌프를 **직접 GPIO에 연결하면 MCU 손상** 위험이 있으므로, 반드시 릴레이/SSR을 사용해야 한다.

---

9. 보호 회로 및 노이즈 대책

- 릴레이 코일 역기전력 방지를 위해 **다이오드(1N4007)** 병렬 연결
- 릴레이 접점 부 근에 **스파크 억제용 RC Snubber** 사용
- 펌프 전원선에 **Flyback 다이오드 / LC 필터** 추가
- GND 공유 시 **Star Ground 방식**으로 구성

```
[PB12]──▶[릴레이 IN]──▶[릴레이 접점]──▶[펌프]
                          |
                          └──▶ 다이오드 병렬
```

---

10. FreeRTOS 확장 예시

버튼 이벤트를 RTOS Task로 전달해 안정적으로 처리할 수도 있다.

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if (GPIO_Pin == GPIO_PIN_0)
  {
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    xTaskNotifyFromISR(PumpTaskHandle, 0, eNoAction, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
  }
}

void PumpTask(void *argument)
{
  for(;;)
  {
    ulTaskNotifyTake(pdTRUE, portMAX_DELAY);
    HAL_GPIO_TogglePin(GPIOB, GPIO_PIN_12);
  }
}
```

> ISR에서는 짧게 Notify만 수행하고,
> 실질적인 릴레이 제어는 Task에서 처리한다 → **안정성 향상**

---

11. 주의사항

항목	설명
버튼 노이즈	디바운스(Delay 또는 소프트웨어 필터) 필수
릴레이 잔류 전압	OFF 직후 코일 잔류자속 존재 — 재클릭 대기 필요
공통 접지	STM32, 릴레이, 펌프 GND 반드시 공통
펌프 전류 스파이크	전원 라인에 콘덴서(470μF 이상) 병렬 권장
전원 분리	펌프 전원(12V)과 MCU 전원(3.3V)은 전기적으로 분리

---

12. 결론

본 실습은 **스위치 인터럽트(EXTI)** 를 활용하여
릴레이를 통해 펌프를 온·오프 제어하는 완전한 임베디드 제어 예제이다.

핵심 포인트는 다음과 같다.

항목	내용
입력	PA0 (EXTI0, Pull-Up, Falling Edge)
출력	PB12 (릴레이 제어 핀)
인터럽트 처리	HAL_GPIO_EXTI_IRQHandler → HAL_GPIO_EXTI_Callback
동작	버튼 누를 때마다 펌프 ON ↔ OFF
보호	다이오드, RC Snubber, 공통 GND 유지
확장	FreeRTOS Task 연동, 디바운스, 센서 기반 자동 제어

✅ 요약

단계	설명
1	EXTI0 설정 (PA0 버튼 입력)
2	PB12 릴레이 출력 설정
3	NVIC 활성화
4	콜백에서 PB12 토글
5	릴레이 구동으로 펌프 ON/OFF

단계	설명
6	노이즈 및 전원 보호 적용

이 예제는 향후 “**Smart Tank 프로젝트**”의 핵심 블록으로 확장되어
수위센서 기반 자동 펌프 제어, FreeRTOS Task 분리, 안전 제어 로직에 응용된다.

- # 수위센서 트리거 입력 (디지털 감지)

1. 개요

이번 실습은 **수위 센서(레벨 스위치)** 의 **디지털 출력 신호**를
STM32의 **GPIO 입력 핀**으로 받아 감지하는 예제이다.

센서의 출력이 **High/Low로 단순히 변화**하는 경우,
아날로그 ADC가 아니라 **디지털 입력(GPIO)** 으로 충분히 인식할 수 있다.

이 구조는 **탱크의 최대 수위 감지**, **펌프 자동 차단**, **누수 감지** 등에 널리 활용된다.

---

2. 센서 종류별 출력 방식

센서 타입	출력 형태	설명
플로트 스위치 (Float Switch)	디지털 (Open/Close)	수면에 따라 기계식 접점이 ON/OFF
정전식 수위 센서 (Capacitive Type)	디지털 또는 아날로그	물 접촉 시 HIGH 출력
광학식 수위 센서 (Optical Level)	디지털	빛의 굴절로 ON/OFF 신호 생성
압력식 센서	아날로그	수두 압력으로 전압 출력 (ADC 필요)

이번 실습은 **디지털 트리거형 센서**(출력 HIGH/LOW)를 대상으로 한다.

---

3. 하드웨어 구성

구성요소	포트	모드	설명
수위 센서 출력	PA1	GPIO Input	트리거 신호 입력 (High: 감지됨)
LED 표시	PC13	Output Push-Pull	감지 시 ON 표시
펌프 제어 (옵션)	PB12	Output	수위 조건에 따라 자동 ON/OFF

회로 연결도

```
3.3V ──────────────────┐
                       |
               [수위센서 VCC]
               [수위센서 OUT] ──────▶ PA1
               [수위센서 GND] ──────▶ GND
                       |
STM32C8T6 (Blue Pill)
```

> 센서가 **감지되면 PA1 = HIGH**,
> 감지되지 않으면 **PA1 = LOW** 로 출력된다고 가정.

---

4. 입력 핀 설정

(1) 클록 활성화

```
__HAL_RCC_GPIOA_CLK_ENABLE();
__HAL_RCC_GPIOC_CLK_ENABLE();
```

(2) 센서 입력 핀 (PA1)

```
GPIO_InitTypeDef GPIO_InitStruct = {0};

GPIO_InitStruct.Pin = GPIO_PIN_1;
GPIO_InitStruct.Mode = GPIO_MODE_INPUT;    // 단순 입력
GPIO_InitStruct.Pull = GPIO_NOPULL;        // 외부 센서가 신호를 구동
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
```

(3) LED 출력 핀 (PC13)

```
GPIO_InitStruct.Pin = GPIO_PIN_13;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
```

---

5. 기본 코드 예제

```
#include "main.h"

void SystemClock_Config(void);
static void MX_GPIO_Init(void);

int main(void)
{
  HAL_Init();
  SystemClock_Config();
```

```
  MX_GPIO_Init();

  while (1)
  {
    // 센서 입력 감지
    if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_SET)
    {
      HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_RESET);  // LED ON
    }
    else
    {
      HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);    // LED OFF
    }

    HAL_Delay(100);  // 100ms 주기 감지
  }
}
```

(실행 결과)

수위 상태	PA1 입력	LED	설명
물 감지됨	HIGH	ON	수위 도달 표시
물 없음	LOW	OFF	정상 상태

---

6. 인터럽트 기반 확장 (옵션)

수위 변화가 즉시 감지되어야 하는 경우, **EXTI 인터럽트**를 사용할 수 있다.

```
// EXTI 설정
GPIO_InitStruct.Pin = GPIO_PIN_1;
GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING_FALLING;  // 상승/하강 감지
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

HAL_NVIC_SetPriority(EXTI1_IRQn, 2, 0);
HAL_NVIC_EnableIRQ(EXTI1_IRQn);
void EXTI1_IRQHandler(void)
{
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_1);
}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  if (GPIO_Pin == GPIO_PIN_1)
  {
    if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_SET)
    {
      HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_RESET); // 수위 도달
      HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_RESET); // 펌프 OFF
```

```
    }
    else
    {
      HAL_GPIO_WritePin(GPIOC, GPIO_PIN_13, GPIO_PIN_SET);
      HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_SET);   // 펌프 ON
    }
  }
}
```

---

7. 수위 트리거 기반 펌프 제어 로직

수위 감지	PA1 입력	동작
감지됨	HIGH	펌프 OFF (물 가득)
미감지	LOW	펌프 ON (물 보충)

```
if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_SET)
{
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_RESET);  // OFF
}
else
{
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_SET);    // ON
}
```

> 이를 통해 **자동 급수 제어 루프**를 구성할 수 있다.
> (예: “Smart Tank” 프로젝트의 기본 수위 제어 알고리즘)

---

8. 디바운스 및 노이즈 필터링

물 접촉 감지 시 전기적 노이즈가 발생할 수 있다.
다음과 같은 방법으로 안정성을 높일 수 있다:

(1) 소프트웨어 디바운스

```
uint8_t readSensorStable(GPIO_TypeDef* port, uint16_t pin)
{
    uint8_t state1 = HAL_GPIO_ReadPin(port, pin);
    HAL_Delay(20);
    uint8_t state2 = HAL_GPIO_ReadPin(port, pin);
    return (state1 == state2) ? state1 : 0;
}
```

(2) RC 필터 회로

- 입력단에 10kΩ + 0.1μF 콘덴서 추가
- 급격한 신호 변화 완화 (노이즈 필터)

(3) 소프트웨어 카운팅 필터

```
static uint8_t count = 0;
if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1) == GPIO_PIN_SET)
{
    if (count++ > 3)
    {
        sensorState = 1;
        count = 0;
    }
}
else
{
    count = 0;
    sensorState = 0;
}
```

---

9. 센서 종류별 입력 설정 요약

센서 타입	입력모드	풀업/풀다운	엣지 감지	비고
플로트 스위치	`IT_FALLING`	Pull-up	하강엣지	GND 연결형
정전식 수위센서	`IT_RISING`	No Pull	상승엣지	High Active
광학식 센서	`INPUT`	No Pull	Polling	0/1 안정적 출력

---

10. 확장 아이디어

- **상한·하한 수위 이중 감지**
  → 상단 수위 OFF, 하단 수위 ON
  → 2개의 센서(PA1, PA2) 조합으로 펌프 제어

```
if (LOW_SENSOR == 0 && HIGH_SENSOR == 1)
    PUMP = OFF;
else if (LOW_SENSOR == 1 && HIGH_SENSOR == 0)
    PUMP = ON;
```

- **FreeRTOS Task 기반**
  - 수위센서 Task: 감지 및 상태 플래그 갱신
  - 제어 Task: 펌프, 밸브 제어 수행
  - 통신 Task: 상태 송신 (UART/MQTT)

11. 결론

본 실습을 통해 **수위센서의 디지털 신호를 입력 감지**하여
LED 또는 펌프를 자동 제어하는 시스템을 구성할 수 있다.

핵심 개념 요약:

항목	내용
입력 핀	PA1
출력 핀	PC13 (LED), PB12 (릴레이)
감지 방식	디지털 HIGH/LOW
제어 로직	수위 감지 시 펌프 OFF
확장	상·하한 수위, FreeRTOS Task, MQTT 전송

✅ 핵심 요약

단계	내용
1	수위센서 출력 핀을 GPIO 입력으로 설정
2	HAL_GPIO_ReadPin()으로 디지털 감지
3	LED 또는 펌프 릴레이 제어
4	노이즈 필터 및 디바운스 적용
5	상·하한 감지 로직으로 확장 가능

> 이 구조는 이후 "**스마트 탱크 FreeRTOS 프로젝트**"의 핵심 입력부로 통합되어
> 수위 감지 → 펌프 제어 → 알람 → 데이터 로깅 순으로 확장된다.