7. 타이머 및 입출력 장치

7.1 타이머/카운터의 원리

> Timer vs Counter – 개념부터 실전까지

🧠 타이머/카운터란?

타이머(Timer)와 **카운터(Counter)**는 모두 **정수값을 증가시키거나 감소시키는 하드웨어 블록**이지만,
무엇을 기준으로 증감하느냐에 따라 다름.

기능	기준 신호
타이머	내부 클럭 (시간 흐름 측정)
카운터	외부 이벤트 또는 펄스 (횟수 측정)

> 타이머는 **시간 측정**, 카운터는 **이벤트 수 계산**에 주로 사용됨.

🔩 공통 구성 요소

구성 요소	설명
클럭 소스 (Clock Source)	타이머 동작을 위한 클럭 (보통 내부 클럭 or 외부 클럭)
프리스케일러 (Prescaler)	입력 클럭을 분주하여 느린 주기로 타이머 동작 가능
카운터 레지스터 (TCNT, CNT 등)	현재 카운트 값을 저장
비교 레지스터 (OCR, ARR 등)	특정 값과 비교하여 인터럽트 트리거
오버플로우 처리	카운터가 최대값 초과 시 0으로 리셋, 인터럽트 발생
제어 레지스터	동작 모드, 인터럽트, 클럭 설정 등 제어 기능 포함

⏱️ 타이머의 동작 원리

1. **내부 클럭(PCLK)**이 프리스케일러를 거쳐 타이머 입력으로 들어옴
2. 카운터가 **일정 주기마다 1씩 증가**
3. **비교값**에 도달하거나 오버플로우 시 인터럽트 발생
4. 인터럽트 서비스 루틴에서 특정 동작 수행 (예: LED 토글, 타임스탬프 저장)

예:

- 1MHz 클럭, Prescaler 1000 → 타이머는 1kHz (1ms 간격으로 증가)
- Compare Match 값 1000 → 1초마다 인터럽트 발생

🔢 카운터의 동작 원리

1. 외부 입력 핀(Tn)을 통해 **펄스 또는 이벤트**가 들어옴
2. 이벤트 1개마다 카운터 값이 증가
3. 설정된 비교값에 도달하면 인터럽트 발생

> 예: 회전 수 센서, 버튼 누름 횟수 측정 등

🔁 기본 동작 모드 종류

모드	설명
업 카운터	0부터 최대값까지 증가 후 리셋
다운 카운터	설정된 값부터 0까지 감소
업-다운 카운터	삼각파 형태의 PWM 생성에 사용
PWM 모드	비교값 도달 시 출력 토글 → 정밀한 파형 생성
CTC 모드 (Clear Timer on Compare)	비교 일치 시 자동으로 카운터 리셋 → 주기 제어에 적합

🔔 인터럽트 연동 방식

종류	트리거 조건
오버플로우 인터럽트	카운터가 최대값 넘으면 발생
비교 일치 인터럽트	비교 레지스터와 값이 같아지면 발생
입력 캡처 인터럽트	외부 신호의 상승/하강 엣지를 감지하고 값 저장
출력 비교 인터럽트	정해진 주기마다 출력 핀 상태 변화

🧪 실무 활용 예시

활용 예	설명
1ms 시스템 Tick 생성	주기적 인터럽트를 발생시켜 타임베이스 제공
PWM 파형 생성	비교값 기반 출력 토글로 서보/모터 제어
주기 측정	두 이벤트 사이 시간 측정 (입력 캡처)
이벤트 카운트	외부 이벤트(펄스) 수 세기 (카운터 모드)
비동기 처리	오버플로우마다 이벤트 처리, 디바이스 Polling 회피

📂 MCU별 예시

플랫폼	타이머/카운터 특징
AVR (ATmega328)	8비트, 16비트 타이머 내장, CTC/PWM 등 지원
ARM (STM32)	TIM1~TIMx 다수 내장, 16~32비트, 고속 PWM 가능
PIC	TMR0, TMR1 등, 타이머/카운터 겸용
ESP32	고속 하드웨어 타이머와 소프트웨어 타이머 동시 지원

📌 요약 정리

항목	타이머	카운터
입력	내부 클럭	외부 이벤트
용도	시간 측정, 주기 생성	이벤트 수 측정
주요 기능	인터럽트, 비교, PWM	펄스 카운트, 횟수 기반 제어
동작 모드	업, 다운, CTC, PWM	업, 다운
사용 예	시스템 Tick, LED 블링킹, 모터 제어	회전수 측정, 입력 횟수 세기

7.2 PWM 제어 및 주기 조정

> Pulse Width Modulation (PWM)의 원리와 활용

🧠 PWM(Pulse Width Modulation) 이란?

PWM은 디지털 신호를 일정한 주기로 ON/OFF함으로써
아날로그 출력을 흉내내는 방식이다.
보통 **LED 밝기 조절, 모터 속도 제어, 오디오 출력, 전력 제어**에 쓰인다.

> "신호가 켜져 있는 시간의 비율"을 조정하여 출력 평균값을 조절하는 방식이다.

📦 핵심 개념: 주기와 듀티비

항목	설명
주기 (Period)	전체 PWM 신호의 반복 시간 (T) = 1 / 주파수
듀티비 (Duty Cycle)	한 주기 중 HIGH 상태의 비율 (0% ~ 100%)
주파수	PWM이 초당 반복되는 횟수 (Hz) = 1 / T

✅ 예시

```
PWM 주기: 1ms (1kHz)
듀티비 50% → 0.5ms ON, 0.5ms OFF
듀티비 10% → 0.1ms ON, 0.9ms OFF
```

🔩 PWM 하드웨어 구조

1. **타이머/카운터**: 일정 속도로 증가하는 카운터 (CNT)
2. **비교 레지스터 (CCR, OCR)**: 이 값에 도달하면 출력 변화
3. **자동 리로드 레지스터 (ARR, TOP)**: 한 주기의 끝 → CNT 리셋
4. **출력 핀 제어**: CCR 도달 전까지 HIGH, 이후 LOW

```
CNT:      0 → 1 → 2 ... → TOP → 0
출력핀:   HIGH → ... → CCR값 도달 시 LOW
```

🔧 주기와 듀티비 조정 방법

항목	조절 대상	설명
주기 (Period)	`ARR` 값 설정	전체 PWM 주기 조절
듀티비 (Duty)	`CCR` 값 설정	HIGH 지속 시간 조절
분주기 (Prescaler)	클럭 속도 감소	타이머 입력 주파수 감소로 PWM 주기 확장 가능

📊 예시 계산

조건:

- 타이머 클럭 = 1 MHz (1 µs 단위)
- Prescaler = 0 (분주 없음)
- ARR = 999 → 주기 = (999+1) × 1µs = **1ms (1kHz)**

목표 듀티비	CCR 값
100%	999
50%	499
10%	99

🛠️ 실전 코드 예시

✅ STM32 HAL 예제

```
TIM_OC_InitTypeDef sConfigOC = {0};

htim2.Init.Prescaler = 71;          // 72MHz / (71+1) = 1MHz
htim2.Init.Period = 999;            // 1ms → 1kHz 주기
HAL_TIM_PWM_Init(&htim2);

sConfigOC.OCMode = TIM_OCMODE_PWM1;
sConfigOC.Pulse = 499;              // 50% 듀티비
HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1);
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
```

✅ AVR(ATmega328P) 예제 (Fast PWM, 8비트)

```
TCCR0A = (1<<WGM00) | (1<<WGM01) | (1<<COM0A1);  // Fast PWM, non-inverting
TCCR0B = (1<<CS01);          // Prescaler = 8
OCR0A = 128;                 // 듀티비 50% (0~255 범위)
```

💉 PWM 활용 사례

분야	설명
모터 제어	듀티비 조절 → 전압 평균값 변화 → 속도 제어
LED 밝기 조절	듀티비로 평균 밝기 조절
오디오 신호 생성	고속 PWM으로 아날로그 음성 신호 구현
SMPS 전력 제어	전력 효율 향상을 위한 고주파 스위칭 방식

🧩 PWM 모드 종류

모드	특징
Fast PWM	빠른 카운팅, 주로 고속 제어
Phase Correct PWM	대칭 파형 생성, 잡음 최소화
Center-Aligned PWM	고급 모터 제어에 사용됨 (특히 3상 BLDC 등)

📌 요약 정리

항목	설명
PWM	디지털 신호의 ON 시간 비율로 아날로그 제어
주기 조절	ARR (TOP) 값으로 설정

항목	설명
듀티비 조절	CCR (비교값)로 설정
활용	LED, 모터, 오디오, 파형 생성 등
MCU 구현	대부분의 타이머/카운터가 PWM 출력 모드 지원

7.3 UART, SPI, I2C 등 시리얼 통신

🧠 시리얼 통신이란?

시리얼 통신은 데이터를 **한 번에 1비트씩 순차적으로 전송**하는 통신 방식이다.
병렬 통신보다 **핀 수가 적고, 배선이 간단**하며, **장거리 통신에 적합**하다.

🧩 주요 시리얼 통신 방식 세 가지

프로토콜	용도
UART	비동기 통신, 장거리 전송에 적합 (예: PC ↔ MCU)
SPI	고속 동기 통신, 짧은 거리, 센서/디스플레이
I2C	다중 슬레이브 통신, 주소 기반, 저속 센서 네트워크

📦 기본 비교표

항목	UART	SPI	I2C
통신 방식	비동기	동기	동기
클럭 신호	❌ 없음 (자체 속도 설정)	✅ 있음 (SCLK)	✅ 있음 (SCL)
데이터선	2개 (TX, RX)	최소 3개 (MOSI, MISO, SCLK) + CS	2개 (SDA, SCL)
다중 장치	불편 (1:1 또는 별도 UART 필요)	✅ 가능 (CS로 슬레이브 선택)	✅ 가능 (주소로 다중 슬레이브 지원)
속도	보통 (115200 ~ 1Mbps)	매우 빠름 (10~50Mbps 이상)	느림 (100kbps, 400kbps, 1Mbps)
전송 단위	바이트 단위	비트 스트림	바이트 단위 + ACK/NAK
마스터/슬레이브	없음 (Peer 방식)	있음	있음
거리	중장거리 (~15m)	짧음 (PCB 수준)	짧음 (~1m 이하 권장)
하드웨어 간결성	단순함	제어선 많음	핀 적고 회로 간결

🔌 1. UART (Universal Asynchronous Receiver/Transmitter)

- **비동기** 방식 (클럭 없음)
- TX ↔ RX만 연결하면 통신 가능
- **동작에는 BAUDRATE 동기화가 필수** (예: 9600, 115200)

✅ 기본 프레임 구조

```
Start | Data (8bit) | Optional Parity | Stop
```

✅ 장점

- 간단하고 하드웨어 부담 적음
- PC 통신, GPS 모듈, BLE 등에서 광범위 사용

🔌 2. SPI (Serial Peripheral Interface)

- **동기식, 마스터/슬레이브 구조**
- 마스터가 **클럭(SCLK)**을 제공하며, 데이터는 MOSI/MISO를 통해 교환

✅ 기본 선 구성

신호	역할
MOSI	Master Out → Slave In
MISO	Master In ← Slave Out
SCLK	마스터 클럭
/CS	슬레이브 선택 (Chip Select)

> **하나의 마스터가 여러 슬레이브 선택** 가능 (멀티-CS 방식)

✅ 특징

- 전이중(Full Duplex)
- 초고속 데이터 전송 가능 (10Mbps 이상)
- 센서, Flash, 디스플레이, DAC 등에 사용

🔌 3. I²C (Inter-Integrated Circuit)

- **동기식, 2선 통신, 주소 기반 다중 슬레이브 지원**
- **마스터가 SCL로 클럭을 제공**, 데이터는 SDA에서 **쌍방향으로 전송**

✅ 기본 선 구성

신호	역할
SCL	클럭
SDA	데이터 (양방향)

✅ 프레임 구조 (기본 흐름)

```
Start → [7bit 주소 + R/W] → ACK → 데이터 → ACK → Stop
```

✅ 특징

- 단방향 반이중(Half Duplex)
- **ACK/NACK 프로토콜**로 데이터 유효성 보장
- 저속이지만 센서와 메모리 간 통신에 이상적
- **풀업 저항 필요**

🧪 실전 예: I2C 온도 센서 읽기 흐름

1. 마스터 → Start
2. 마스터 → 슬레이브 주소 + Write
3. 슬레이브 → ACK
4. 마스터 → 레지스터 주소 전송
5. 마스터 → Repeated Start
6. 마스터 → 슬레이브 주소 + Read
7. 슬레이브 → ACK
8. 슬레이브 → 데이터 바이트 전송
9. 마스터 → NACK → Stop

🛠️ 실전 코드 예시 (STM32 HAL 기반)

UART 전송

```
char msg[] = "Hello";
HAL_UART_Transmit(&huart2, (uint8_t*)msg, strlen(msg), HAL_MAX_DELAY);
```

SPI 전송

```
uint8_t tx = 0x9F;
uint8_t rx;
HAL_SPI_TransmitReceive(&hspi1, &tx, &rx, 1, HAL_MAX_DELAY);
```

I2C 읽기

```
uint8_t reg = 0x01;
uint8_t data;
HAL_I2C_Master_Transmit(&hi2c1, 0x48 << 1, &reg, 1, HAL_MAX_DELAY);
HAL_I2C_Master_Receive(&hi2c1, 0x48 << 1, &data, 1, HAL_MAX_DELAY);
```

📌 요약 정리

프로토콜	방식	장점	단점
UART	비동기	간단, 거리 가능	동기화 어려움, 1:1 통신
SPI	동기, 고속	빠름, 풀듀플렉스	선 많음, CS 핀 증가
I2C	동기, 저속	다중 슬레이브, 핀 적음	느림, 프로토콜 복잡

7.4 A/D, D/A 변환 원리

> Analog to Digital, Digital to Analog Conversion

🧠 왜 A/D, D/A 변환이 필요한가?

현실 세계의 대부분은 **아날로그 신호** (연속적인 전압, 온도, 소리 등)지만,
디지털 시스템(CPU, MCU)은 **0과 1만 인식 가능**하기 때문에
→ **아날로그 ↔ 디지털 변환**이 필수이다.

변환	설명	용도 예
ADC (Analog → Digital)	센서값을 CPU가 처리 가능하게 변환	온도, 조도, 전압 측정
DAC (Digital → Analog)	디지털 값을 아날로그 전압으로 출력	음향 출력, 정전압 제어

🔁 전체 흐름

```
[센서 출력 전압] → [ADC] → [MCU/CPU에서 처리] → [DAC] → [아날로그 출력]
```

🔹 1. A/D 변환기 (ADC: Analog to Digital Converter)

✅ 기본 작동 원리

- 입력 아날로그 전압을 **일정 간격으로 샘플링**
- 각 샘플을 **n비트 디지털 값**으로 변환

```
[0V ~ Vref] → 디지털 범위 [0 ~ 2ⁿ - 1]
```

> 예: 10비트 ADC → 0~1023 값으로 전압 표현

✅ 주요 파라미터

항목	설명
분해능 (Resolution)	몇 비트로 아날로그 값을 표현할 수 있는가 (정밀도)
샘플링 속도	초당 샘플링 횟수 (Sample/s 또는 SPS)
레퍼런스 전압 (Vref)	최대 측정 가능한 전압
정확도 (Accuracy)	실제 전압과 디지털 변환값의 오차
입력 채널 수	ADC에 연결할 수 있는 아날로그 입력 개수

✅ 예시: 12비트 ADC, Vref = 3.3V

```
1 LSB = 3.3V / (2¹² - 1) = 약 0.8 mV
입력 1.65V → 디지털 값 ≈ 2048
```

✅ 변환 방식

방식	설명
Successive Approximation (SAR)	MCU에서 가장 흔함, 중간 속도/정밀도
Delta-Sigma (ΔΣ)	매우 정밀, 속도 느림
Flash ADC	매우 빠름 (고속 통신/오디오), 비트 수 제한
Dual-Slope	노이즈 저항성 좋음, 속도 느림 (전력계 등)

✅ 실전 예시 (STM32 HAL)

```
HAL_ADC_Start(&hadc1);
HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
uint32_t value = HAL_ADC_GetValue(&hadc1);
```

🔹 2. D/A 변환기 (DAC: Digital to Analog Converter)

✅ 기본 작동 원리

- 디지털 값을 입력받아 **아날로그 전압 출력**

```
디지털 입력 (0~255, 0~4095 등) → 전압 출력 (0V ~ Vref)
```

✅ 사용 목적

- 사운드 출력 (스피커)
- 전압 제어 장치 (모터 속도, 조명 밝기)
- 파형 발생기 (Sine, Square 등)

✅ 주요 파라미터

항목	설명
분해능	DAC 출력 전압 단계 수
정확도	목표 전압 대비 출력 전압 오차
속도	설정 후 전압이 안정화되기까지의 시간
참조 전압(Vref)	출력 전압의 최대값 결정 기준

✅ 실전 예시 (STM32 HAL)

```
HAL_DAC_Start(&hdac, DAC_CHANNEL_1);
HAL_DAC_SetValue(&hdac, DAC_CHANNEL_1, DAC_ALIGN_12B_R, 2048);
```

→ `2048`은 12비트 DAC에서 50% (중간 전압, 약 1.65V)

📊 A/D vs D/A 비교표

항목	ADC	DAC
입력	아날로그 전압	디지털 값
출력	디지털 값	아날로그 전압
방향	센서 → MCU	MCU → 액츄에이터
용도	측정, 모니터링	제어, 출력
MCU 활용도	매우 흔함 (ADC 내장 많음)	고급 MCU 또는 외장 칩 필요

🧪 실전 활용 예시

예시	ADC	DAC
온도 측정	온도센서 → ADC → 디지털 분석	-
음성 출력	-	디지털 오디오 → DAC → 앰프
조도 조절	광센서 → ADC	PWM으로도 가능, DAC로도 가능
정전압 출력	-	전압 제어식 DAC 출력

📌 요약 정리

항목	설명
ADC	아날로그 입력 → 디지털 값 변환
DAC	디지털 값 → 아날로그 출력 전압 생성

항목	설명
공통 요소	분해능, 참조전압, 정밀도 중요
ADC 구조	SAR, Sigma-Delta, Flash 등 다양
실전 예시	STM32, AVR 등 거의 모든 MCU에서 사용 가능

7.5 외부 입출력 장치 인터페이싱

> 외부 장치와의 하드웨어/소프트웨어 연결 구조

🧠 I/O 인터페이싱이란?

입출력 인터페이싱은 **프로세서와 외부 장치(센서, 스위치, 모터, 디스플레이 등)** 간에
정확하고 안정적인 데이터 교환이 가능하도록 연결해주는 기술과 회로를 의미한다.

> 단순히 "핀을 연결하는 것"이 아니라
> **신호 정합, 타이밍 보정, 주소 디코딩, 레벨 변환**까지 포함하는 고급 회로 설계이다.

🧩 외부 장치의 분류

유형	예시	특징
디지털 입력	버튼, IR 센서	ON/OFF만 감지 (High/Low)
디지털 출력	LED, 릴레이	단순 제어 신호 출력
아날로그 입력	온도센서, 포텐셔미터	ADC 필요
아날로그 출력	DAC → 전압 제어형 소자	PWM or DAC 사용
시리얼 장치	UART, I2C, SPI 기반 센서	시퀀스 기반 통신 필요
병렬 장치	LCD, 메모리, 키패드	복수 비트 데이터 + 제어 신호 필요

🔌 I/O 인터페이스 설계 시 고려사항

요소	설명
전압 레벨	3.3V ↔ 5V ↔ 1.8V 등 레벨 변환 필요 여부
입출력 방향	Input인지 Output인지 명확히 설정
속도/대역폭	타이밍 제약, 고속이면 타이밍 회로 필요
신호 형태	디지털/아날로그, 펄스/레벨, Active Low/High 등
보호 회로	ESD, 과전압 보호 (저항, 쇼트 방지 다이오드 등)
하드웨어 제어 방식	메모리 매핑(MMIO), 포트 맵핑, 버스 연결 등

🧱 I/O 주소 지정 방식

✅ 1. Memory-Mapped I/O (MMIO)

- I/O 장치를 **메모리 주소 공간에 직접 연결**
- MCU/CPU가 `mov`, `ldr`, `str` 명령으로 접근

```
#define LED_PORT  (*(volatile uint8_t*)0x40020014)
LED_PORT |= (1 << 5);  // GPIO 출력 제어
```

✅ 2. Port-Mapped I/O (x86 계열)

- I/O 공간이 메모리와 분리
- `IN`, `OUT` 명령으로만 접근 가능
- 고전적 아키텍처에서 사용됨 (x86 8086~)

🔧 디지털 I/O 예: 버튼 → LED

회로:

- 버튼 → 풀업 저항 → GPIO 입력
- LED → 저항 → GPIO 출력

코드 (예: STM32 HAL)

```
if (HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_0) == GPIO_PIN_SET) {
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_SET);
}
```

🛠️ 아날로그 입력 예: 온도센서 (NTC, LM35 등)

- 출력 전압 → ADC → 디지털 값으로 변환
- 필요시 Vref 기준값 보정

```
HAL_ADC_Start(&hadc1);
HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
uint32_t val = HAL_ADC_GetValue(&hadc1);
float temp = (val / 4095.0f) * 3.3f;  // 전압 계산
```

🧪 외부 장치 예시와 인터페이싱 방법

장치	입력/출력	연결 방식	특이점
버튼	입력	GPIO, 풀업	바운스 제거 필요
LED	출력	GPIO	저항 필수
서보 모터	출력	PWM	50Hz 주기, 1~2ms 펄스

장치	입력/출력	연결 방식	특이점
스텝모터 드라이버	출력	디지털 펄스	방향핀 + 스텝핀 필요
I2C 센서 (MPU6050 등)	입력	SDA/SCL	주소, ACK 처리 필요
SPI Flash	입출력	MISO/MOSI/SCLK/CS	타이밍 제어 중요
ADC 입력	입력	전압 입력 → MCU 내부 ADC	기준 전압 중요
DAC 출력	출력	디지털값 → 전압 출력	로우패스 필터 필요 가능

📊 실시간 인터페이스 고려사항

조건	설계 전략
고속 장치	DMA 또는 인터럽트 기반 I/O 사용
저속 장치	폴링(Polling) 방식도 가능
버스 충돌 방지	Open-Drain, 3-State Buffer 활용
하드웨어 손상 방지	보호 저항, 정류 다이오드 삽입

📌 요약 정리

항목	설명
I/O 인터페이싱	외부 장치 ↔ CPU 연결 위한 신호 조율 기술
디지털/아날로그	디지털: GPIO / 아날로그: ADC or DAC 필요
I/O 제어 방식	MMIO (주소 공간에 직접 연결), Port-Mapped (명령어 기반 접근)
타이밍/레벨	전압 호환성, 속도, 신호 유효 타이밍 중요
MCU 예제	GPIO, ADC, PWM, 시리얼 통신으로 실현 가능

7.6 GPIO 제어 구조

🧠 GPIO란?

GPIO (범용 입출력 핀)은 MCU나 CPU에서 **디지털 신호를 입출력하기 위한 가장 기본적인 포트**이다.
핀 하나하나를 독립적으로 **입력 또는 출력**으로 설정할 수 있고,
외부 장치와의 **디지털 신호 통신**의 기본이 된다.

🧩 GPIO의 주요 기능

기능	설명
디지털 입력	외부에서 High/Low 상태를 읽음 (예: 버튼)

기능	설명
디지털 출력	핀에서 High/Low 신호를 내보냄 (예: LED)
인터럽트 트리거	핀 상태 변화에 따라 인터럽트 발생 가능
풀업/풀다운 설정	미결정 상태 방지용 내부 저항 설정
오픈드레인 출력	I2C 같은 버스에서 다중 장치 연결 가능
고속 출력	일부 MCU는 PWM/클럭 출력을 GPIO로 제공

🧱 GPIO 제어의 기본 구조 (MCU 내부)

MCU에서 GPIO는 보통 다음과 같은 레지스터로 제어된다:

레지스터	기능
MODER / DDR / TRIS	입력/출력 모드 설정
IDR / PIN / IN	입력 데이터 읽기
ODR / PORT / OUT	출력 데이터 설정
PUPDR / Pullup 설정	내부 풀업/풀다운 저항 제어
BSRR / SET/CLR	원자적 출력 제어 (STM32)

🔁 입출력 설정 흐름

```
1. 모드 설정: 입력 / 출력
2. 풀업/풀다운 설정 (입력 시)
3. 핀 상태 읽기 또는 출력 제어
4. (선택) 인터럽트 또는 오픈드레인 설정
```

🧪 실전 예제

✅ STM32 (HAL 라이브러리)

```
// 출력 핀 설정
GPIO_InitTypeDef GPIO_InitStruct = {0};
GPIO_InitStruct.Pin = GPIO_PIN_5;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);

// 출력 제어
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET); // High 출력
HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);              // 토글
```

✅ AVR (ATmega328P)

```
DDRB |= (1 << PB0);   // PB0을 출력으로 설정
PORTB |= (1 << PB0);  // PB0에 High 출력
```

🔌 풀업(Pull-up) / 풀다운(Pull-down) 저항

- GPIO 입력 핀이 공중에 떠 있는 상태(Floating)일 경우 **불안정한 동작** 발생
- 내부/외부 풀업 저항을 통해 상태를 안정화함

상태	설명
풀업	입력이 연결되지 않으면 High 유지
풀다운	입력이 연결되지 않으면 Low 유지

> STM32/AVR은 내부 풀업 저항 설정을 레지스터로 지원함 (`PUPDR`, `PORT |=` 등)

⚙️ 인터럽트 기반 GPIO 입력

- 버튼 누름 등 **상태 변화 이벤트**를 감지하여 인터럽트를 발생시킴

```
GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
```

🧠 오픈드레인(Open Drain) vs 푸시풀(Push-Pull)

방식	설명	예시
Push-Pull	High/Low 모두 직접 구동	일반 출력용 (LED 등)
Open-Drain	Low만 출력, High는 부동 → 풀업 필요	I2C, 멀티 슬레이브 버스
Open-Source	High만 출력 (드문 방식)	일부 특수 회로

📊 GPIO 제어 시 유의사항

주의점	설명
입출력 모드 혼동 금지	출력 핀을 입력으로 읽으면 무의미
속도 설정 고려	고속 출력이 필요할 때만 고속으로 설정 (EMI 방지)
전류 제한	MCU GPIO 당 최대 전류 (10~25mA) 넘지 말 것
디바운싱 처리	버튼 등 신호 변동이 심한 입력은 소프트웨어 안정화 필요

📌 요약 정리

항목	설명
GPIO	범용 디지털 입출력 핀
모드 설정	입력, 출력, 오픈드레인, 인터럽트 등
출력 제어	비트 조작, HAL 함수, BSRR 활용
입력 읽기	High/Low 판별, 내부 풀업 설정 가능
인터럽트 연동	핀 상태 변화 감지에 최적
주의사항	전류 제한, 혼선 방지, 디바운스 필수