6. 센서 드라이버 개발

6.1 초음파 센서 (HC-SR04)

- ## TRIG 펄스 생성, ECHO 펄스폭 측정

1. 개요

HC-SR04는 초음파 송수신 모듈로,
40 kHz 초음파를 방사한 후 반사되어 돌아오는 신호의 왕복 시간을 측정하여
대상 물체까지의 거리를 계산한다.

센서는 두 개의 주요 제어 신호를 가진다.

- **TRIG** : 초음파 발사를 시작하는 입력 신호
- **ECHO** : 반사파 수신 시간에 따라 High 상태로 유지되는 출력 신호

ECHO 펄스의 폭(duration)이 길수록 물체까지의 거리가 멀다는 것을 의미한다.

---

2. 동작 원리 요약

1. **TRIG 핀**에 최소 **10 µs** 이상의 High 펄스를 인가한다.
2. 센서가 40 kHz 초음파를 약 8 회 발사한다.
3. 반사 신호가 돌아올 때까지 **ECHO 핀**은 High 상태를 유지한다.
4. 반사 신호 수신 완료 시 ECHO가 Low로 떨어진다.
5. **ECHO의 High 지속 시간(Δt)** 으로 거리 계산이 가능하다.

거리 계산식은 다음과 같다.

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

또는

$$\text{Distance (mm)} = \frac{\text{Pulse width } (\mu\text{s})}{5.8}$$

(음속 = 약 343 m/s, 공기 중 기준)

---

3. 하드웨어 연결 예시

핀명	STM32 포트	방향	설명
VCC	5 V	입력	전원
GND	GND	입력	접지
TRIG	PB9	출력	초음파 발사 트리거
ECHO	PB8	입력 (Timer Input Capture)	반사 신호 입력

> ⚠️ ECHO 신호는 5 V 레벨로 출력되므로,
> 3.3 V MCU에서 사용할 경우 **10 kΩ / 20 kΩ 분압** 또는 **레벨 시프터**를 사용해야 한다.

---

4. TRIG 펄스 생성 (GPIO 제어)

TRIG 핀은 단순한 디지털 출력으로 제어한다.
아래 예제는 정확한 10 µs 펄스를 생성하는 코드이다.

```
void HCSR04_Trigger(void)
{
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_9, GPIO_PIN_RESET);
    HAL_Delay(1); // 안정화
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_9, GPIO_PIN_SET);
    delay_us(10);  // 10 µs High 유지
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_9, GPIO_PIN_RESET);
}
```

여기서 `delay_us()`는 Timer 기반 마이크로초 단위 지연 함수이다.
Timer2 또는 Timer3를 사용해 구현할 수 있다.

---

5. ECHO 펄스폭 측정 (Timer Input Capture)

ECHO 핀의 상승엣지(High 시작)와 하강엣지(High 종료)를
Timer의 Input Capture 기능으로 검출하여, 펄스폭을 계산한다.

1. **TIMx_CHx**를 **Input Capture 모드**로 설정
2. **Rising Edge**에서 첫 번째 카운터 값 저장
3. **Falling Edge**에서 두 번째 카운터 값 저장
4. 두 값의 차이를 이용해 시간(µs)을 계산

예시 코드:

```
uint32_t ic_val1 = 0, ic_val2 = 0;
uint8_t is_first_captured = 0;
uint32_t difference = 0;

void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1)
    {
        if (is_first_captured == 0)
        {
            ic_val1 = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
            is_first_captured = 1;
            __HAL_TIM_SET_CAPTUREPOLARITY(htim, TIM_CHANNEL_1,
TIM_INPUTCHANNELPOLARITY_FALLING);
        }
        else
        {
```

```
            ic_val2 = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
            if (ic_val2 > ic_val1)
                difference = ic_val2 - ic_val1;
            else
                difference = (0xFFFF - ic_val1) + ic_val2;

            is_first_captured = 0;
            __HAL_TIM_SET_CAPTUREPOLARITY(htim, TIM_CHANNEL_1,
TIM_INPUTCHANNELPOLARITY_RISING);
            __HAL_TIM_DISABLE_IT(htim, TIM_IT_CC1);
        }
    }
}
```

---

6. 거리 계산

Timer 주파수가 1 MHz (즉, 1 tick = 1 μs)라면,
`difference`는 곧 **ECHO 펄스폭(μs)** 이 된다.

따라서 거리 계산식은 다음과 같다.

```
float HCSR04_GetDistance(void)
{
    float distance = (float)difference / 58.0f; // cm 단위
    return distance;
}
```

---

7. 측정 시퀀스

```
void HCSR04_Read(void)
{
    HCSR04_Trigger();
    __HAL_TIM_ENABLE_IT(&htim3, TIM_IT_CC1);
    HAL_Delay(60); // 최대 거리 약 400cm 기준 대기시간
    float distance = HCSR04_GetDistance();
    printf("Distance: %.2f cm\r\n", distance);
}
```

---

8. 측정 주기 및 제한사항

항목	값
최소 트리거 간격	60 ms
최대 측정 거리	약 400 cm
분해능	약 3 mm

항목	값
전원 전압	5 V (±10 %)
동작 전류	약 15 mA

9. 디버깅 포인트

현상	원인	조치
측정값 0 cm 고정	ECHO 미검출	레벨 시프터 확인
무작위 값 출력	Timer 오버플로	프리스케일러 재설정
거리 항상 일정	TRIG 펄스 실패	GPIO 모드/지연 확인
400 cm 초과 후 0 cm 반복	타임아웃 미처리	MAX_TIMEOUT 설정

10. 결론

- HC-SR04는 **TRIG 신호로 발사**, **ECHO 신호의 펄스폭으로 거리 계산**하는 구조이다.
- STM32의 **Timer Input Capture**를 활용하면 μs 단위의 정밀 측정이 가능하다.
- 실무에서는 필터링 및 타임아웃 처리를 결합하여
  안정적인 거리 센서 시스템을 구축할 수 있다.

Timer Input Capture 기반 거리 계산

1. 개요

초음파 센서(HC-SR04 등)의 **ECHO 핀**은 물체까지의 왕복 시간에 비례하는
High 펄스를 출력한다.
STM32의 **Timer Input Capture** 기능을 이용하면 이 펄스폭을
하드웨어 수준에서 정밀하게 측정할 수 있다.

Timer는 내부 클록을 기반으로 주기적으로 카운트하며,
입력 신호의 상승엣지(Rising Edge) 또는 하강엣지(Falling Edge) 발생 시
현재 카운터 값을 자동으로 캡처(Capture)하여 저장한다.
이를 이용하면 High 펄스의 시작과 끝을 하드웨어 정확도로 계산할 수 있다.

2. 동작 원리

1. Timer가 일정 주파수(F_timer)로 연속 카운트한다.
2. ECHO 핀을 Timer의 Input Capture 채널에 연결한다.
3. Rising Edge에서 첫 번째 카운터 값(Capture1)을 저장한다.
4. Falling Edge에서 두 번째 카운터 값(Capture2)을 저장한다.
5. 두 값의 차이(ΔCount)가 곧 펄스폭(Δt)에 해당한다.

$$\Delta t = \frac{\Delta Count}{F_{timer}}$$

---

3. Timer 설정

3.1 기본 타이머 주파수 설정

Timer 클록 주파수를 1 MHz (1 μs 해상도)로 설정한다.
예: APB1 Timer Clock = 72 MHz (STM32F103 기준)

```
htim3.Instance = TIM3;
htim3.Init.Prescaler = 72 - 1;      // 72MHz / 72 = 1MHz (1μs)
htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
htim3.Init.Period = 0xFFFF;         // 최대 65535 μs = 65.535 ms
htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
HAL_TIM_IC_Init(&htim3);
```

3.2 Input Capture 채널 설정

ECHO 신호를 입력받는 채널을 설정한다.
다음은 TIM3_CH1 (PB4) 예시이다.

```
sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;   // 상승엣지부터 측정 시작
sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
sConfigIC.ICFilter = 0;
HAL_TIM_IC_ConfigChannel(&htim3, &sConfigIC, TIM_CHANNEL_1);
```

이후 인터럽트를 활성화한다.

```
HAL_TIM_IC_Start_IT(&htim3, TIM_CHANNEL_1);
```

---

4. 캡처 콜백 처리

Rising → Falling 엣지 전환 방식으로 펄스폭을 구한다.

```
uint32_t ic_val1 = 0, ic_val2 = 0;
uint8_t is_first_captured = 0;
uint32_t difference = 0;

void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Channel == HAL_TIM_ACTIVE_CHANNEL_1)
    {
        if (is_first_captured == 0)
        {
            ic_val1 = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
            is_first_captured = 1;
```

```
13
14              __HAL_TIM_SET_CAPTUREPOLARITY(htim, TIM_CHANNEL_1,
    TIM_INPUTCHANNELPOLARITY_FALLING);
15          }
16          else
17          {
18              ic_val2 = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
19
20              if (ic_val2 > ic_val1)
21                  difference = ic_val2 - ic_val1;
22              else
23                  difference = (0xFFFF - ic_val1) + ic_val2;
24
25              is_first_captured = 0;
26
27              __HAL_TIM_SET_CAPTUREPOLARITY(htim, TIM_CHANNEL_1,
    TIM_INPUTCHANNELPOLARITY_RISING);
28              __HAL_TIM_DISABLE_IT(htim, TIM_IT_CC1);
29          }
30      }
31  }
```

5. 거리 계산

Timer가 1 MHz로 동작 중이라면 `difference` 값은
펄스폭(μs)과 동일하다.
이를 이용하여 거리 계산을 수행한다.

```
1  float HCSR04_GetDistance(void)
2  {
3      float distance = (float)difference / 58.0f; // cm 단위
4      return distance;
5  }
```

6. 측정 시퀀스

```
1  void HCSR04_Read(void)
2  {
3      HCSR04_Trigger();  // 10 μs 트리거 펄스
4      __HAL_TIM_ENABLE_IT(&htim3, TIM_IT_CC1);
5      HAL_Delay(60);     // 최대 대기 시간 (400 cm 기준)
6      float distance = HCSR04_GetDistance();
7      printf("Distance: %.2f cm\r\n", distance);
8  }
```

7. 측정 타이밍

항목	값
타이머 주파수	1 MHz
측정 해상도	1 μs
최대 펄스폭	38 ms (약 6.5 m)
권장 측정 간격	≥ 60 ms
오차 요인	온도, 습도, 기압, 반사각

---

8. 오류 처리 및 안정화

오류	원인	해결 방법
거리 0 고정	ECHO 미검출	연결, 레벨 시프터 확인
불규칙 측정값	클록 불안정	프리스케일러 재확인
매우 큰 값 출력	Falling 엣지 미검출	타임아웃 설정 필요
주기적 노이즈	다중 반사	이동평균 필터 적용

---

9. 고급 응용

1. **DMA 기반 Input Capture**
   - 연속 거리 데이터를 자동 수집 가능
   - CPU 부하 최소화
2. **FreeRTOS Task 통합**
   - 주기적 거리 측정 Task + 데이터 처리 Task 분리
3. **온도 보정 알고리즘**
   - 음속 $v = 331.4 + 0.6 \times T(^{\circ}C)$
   - 실시간 온도센서 연동 시 ±1% 정확도 향상

---

10. 결론

- Timer Input Capture는 ECHO 신호의 펄스폭을 μs 단위로 정밀하게 측정한다.
- 하드웨어 캡처 방식을 사용하면 소프트웨어 폴링 대비 지터가 거의 없고,
  안정적인 거리 산출이 가능하다.
- 이후 단계로는 DMA 또는 FreeRTOS 기반 자동화 구조를 설계할 수 있다.

- # MAX_TIMEOUT 처리

1. 개요

초음파 센서(HC-SR04)는 물체까지의 거리에 따라 **ECHO 핀의 High 지속 시간**이 변한다.
그러나 **측정 범위를 초과하거나 반사체가 없는 경우**,
ECHO 핀은 Low로 돌아오지 않거나 비정상적인 펄스폭을 유지할 수 있다.

이러한 상황을 방치하면 펌웨어가 **무한 대기 상태(Hang)** 에 빠질 수 있으므로,
측정 루틴에 **최대 대기 시간(MAX_TIMEOUT)** 을 설정하여
안정적인 동작을 보장해야 한다.

---

2. 원리

1. TRIG 펄스 출력 후 ECHO 핀 상승엣지를 대기한다.
2. 상승엣지 감지 시 Timer를 시작한다.
3. 하강엣지를 감지하기 전까지 대기하되,
   지정된 최대 시간(예: 40 ms)을 초과하면 타임아웃으로 간주한다.
4. 타임아웃 발생 시 ECHO 신호를 강제로 종료하고,
   측정값을 0 또는 이전 유효값으로 대체한다.

---

3. 이론적 기준값

HC-SR04의 최대 감지 거리는 약 **4m**이며,
왕복 시간은 다음과 같이 계산된다:

$$t_{max} = \frac{2 \times 4\ \text{m}}{343\ \text{m/s}} = 0.0233\ \text{s} \approx 23.3\ \text{ms}$$

따라서 25~30ms 정도를 타임아웃으로 설정하면 충분하다.
안정성을 위해 **40ms (40000μs)** 으로 설정하는 경우가 많다.

---

4. 코드 예시 (Polling 방식)

```
#define MAX_TIMEOUT 40000  // 40 ms

uint32_t HCSR04_Read(void)
{
    uint32_t timeout = 0;
    uint32_t start_time, echo_time;

    // TRIG High (10μs)
    HAL_GPIO_WritePin(TRIG_GPIO_Port, TRIG_Pin, GPIO_PIN_SET);
    delay_us(10);
    HAL_GPIO_WritePin(TRIG_GPIO_Port, TRIG_Pin, GPIO_PIN_RESET);

    // ECHO Rising Edge 대기
    timeout = 0;
```

```
    while (HAL_GPIO_ReadPin(ECHO_GPIO_Port, ECHO_Pin) == GPIO_PIN_RESET)
    {
        if (timeout++ > MAX_TIMEOUT)
            return 0;  // 타임아웃
        delay_us(1);
    }

    start_time = getMicros();  // Timer 기반 μs 단위 시간

    // ECHO Falling Edge 대기
    timeout = 0;
    while (HAL_GPIO_ReadPin(ECHO_GPIO_Port, ECHO_Pin) == GPIO_PIN_SET)
    {
        if (timeout++ > MAX_TIMEOUT)
            return 0;  // 타임아웃
        delay_us(1);
    }

    echo_time = getMicros() - start_time;

    // 거리 계산 (cm)
    return (uint32_t)(echo_time / 58);
}
```

---

5. 코드 예시 (Timer Input Capture + Timeout)

Input Capture 방식에서도,
Timer Overflow Interrupt 또는 **Software Timer** 를 이용해
타임아웃을 감시할 수 있다.

```
#define ECHO_TIMEOUT 40000 // 40ms

void HCSR04_Trigger(void)
{
    HAL_GPIO_WritePin(TRIG_GPIO_Port, TRIG_Pin, GPIO_PIN_SET);
    delay_us(10);
    HAL_GPIO_WritePin(TRIG_GPIO_Port, TRIG_Pin, GPIO_PIN_RESET);

    __HAL_TIM_ENABLE_IT(&htim3, TIM_IT_CC1);
    __HAL_TIM_SET_COUNTER(&htim3, 0);
    HAL_TIM_Base_Start_IT(&htim3);
}

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if (htim->Instance == TIM3)
    {
        if (__HAL_TIM_GET_COUNTER(htim) > ECHO_TIMEOUT)
        {
            // ECHO 미검출 → 타임아웃 처리
            __HAL_TIM_DISABLE_IT(htim, TIM_IT_CC1);
```

```
            HAL_TIM_Base_Stop_IT(htim);
            is_first_captured = 0;
            distance = 0.0f;
        }
    }
}
```

---

6. 예외 처리 전략

상황	증상	처리 방법
반사체 없음	무한 대기	MAX_TIMEOUT 초과 시 강제 종료
반사 신호 약함	펄스 미검출	타임아웃 후 “0 cm” 리턴
타이머 오버플로	측정 불능	타임아웃 조건 포함
잡음 감지	비정상 펄스	2회 이상 연속 측정 후 평균

---

7. 개선 기법

1. **Timer 기반 µs 단위 Timeout**
   - HAL_GetTick()은 ms 단위로 부정확하므로, Timer 또는 DWT 기반 µs 카운터 사용
2. **FreeRTOS Task 통합 시**
   - osDelay()로 대기 중에도 Watchdog Reset 방지 필요
   - Timeout 발생 시 Queue에 “TIMEOUT” 이벤트 전달
3. **Fail-safe 거리 보정**
   - 최근 유효 거리 값을 캐시하여, 타임아웃 시 이전 값 유지
   - “거리 유효 플래그(valid flag)”로 측정 품질 표시

---

8. 결론

- MAX_TIMEOUT 처리는 초음파 거리 측정 시스템의 **안정성 확보 핵심 요소**이다.
- 타임아웃을 적용하지 않으면 MCU가 **무한 루프 또는 블로킹 상태**에 빠질 수 있다.
- 실시간 운영 환경(FreeRTOS 등)에서는 **비차단(Non-blocking) 구조 + Timeout 이벤트 전송** 방식이 이상적이다.

6.2 로드셀 (HX711)

- # 24bit ADC 인터페이스

1. 개요

HX711은 로드셀(Load Cell) 신호를 고정밀로 읽기 위해 설계된
24비트 ΔΣ(Delta-Sigma) ADC이다.
내장된 **프로그래머블 게인 앰프(PGA)** 와 **전원 제어 회로**를 포함하여
아날로그 전압 차(ΔV = V+ – V−)를 디지털 값으로 변환해 MCU로 전송한다.

이 칩은 **클럭 입력을 외부에서 제공하지 않으며**,
MCU가 직접 **SCK 신호를 구동**하고,
그에 따라 **DOUT 핀으로 직렬 비트 스트림을 읽는 구조**이다.

---

2. 핀 구성 및 역할

핀명	방향	설명
VCC	입력	2.6V ~ 5.5V 전원 입력
GND	입력	기준 접지
DT (DOUT)	출력	변환 완료 데이터 출력
SCK	입력	클록 입력 (MCU 제어)
AVDD, DVDD	내부 전원 분리 (아날로그 / 디지털)	
A+, A−, B+, B−	로드셀 브리지 입력 (채널 A, B)	

---

3. 동작 원리

3.1 ΔΣ 변환 구조

HX711은 **ΔΣ ADC** 구조를 사용한다.
입력 아날로그 전압을 연속적으로 오버샘플링하고,
노이즈를 평균화하여 고분해능(24bit)의 디지털 데이터를 생성한다.

변환 주기는 설정에 따라 **10 SPS(Hz)** 또는 **80 SPS**로 동작한다.

3.2 데이터 전송 프로토콜

1. 변환이 완료되면 **DOUT 핀 = Low** 로 전환된다.
2. MCU가 SCK를 24회 토글하면,
   DOUT에서 **MSB → LSB 순으로 24bit 데이터**가 출력된다.
3. 24비트 이후 SCK를 **1~3회 추가 토글**하여
   다음 변환의 **게인 및 채널 설정**을 결정한다.

추가 클록 수	선택 채널	게인
25	A	128
26	B	32
27	A	64

---

4. 데이터 프레임 구조

비트	설명
Bit[23:0]	2의 보수(24bit signed) 데이터
Bit[23]	부호 비트 (1 = 음수)
Bit[22:0]	유효 측정 데이터
이후 SCK 펄스	게인/채널 설정

데이터는 **MSB First**, **2's Complement 형식**으로 제공된다.

---

5. MCU 인터페이스 시퀀스

5.1 측정 준비

1. **DOUT 핀**이 Low로 떨어질 때까지 대기
   (변환 완료 신호)
2. 이후 **24회의 SCK 펄스**를 생성
3. 각 펄스의 상승엣지 또는 하강엣지에서 DOUT 값을 시프트 입력

5.2 데이터 수집 루틴

```
#define HX711_DT_Pin   GPIO_PIN_1
#define HX711_SCK_Pin  GPIO_PIN_2
#define HX711_DT_Port  GPIOA
#define HX711_SCK_Port GPIOA

long HX711_ReadRaw(void)
{
    long data = 0;

    while (HAL_GPIO_ReadPin(HX711_DT_Port, HX711_DT_Pin)); // DOUT=LOW 대기

    for (int i = 0; i < 24; i++)
    {
        HAL_GPIO_WritePin(HX711_SCK_Port, HX711_SCK_Pin, GPIO_PIN_SET);
        data = (data << 1) | HAL_GPIO_ReadPin(HX711_DT_Port, HX711_DT_Pin);
        HAL_GPIO_WritePin(HX711_SCK_Port, HX711_SCK_Pin, GPIO_PIN_RESET);
```

```
    }

    // 게인 설정용 추가 클럭 (128배)
    HAL_GPIO_WritePin(HX711_SCK_Port, HX711_SCK_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(HX711_SCK_Port, HX711_SCK_Pin, GPIO_PIN_RESET);

    // 24비트 부호 확장 (2의 보수)
    if (data & 0x800000)
        data |= 0xFF000000;

    return data;
}
```

---

6. 보정 및 스케일 적용

6.1 Offset Calibration (제로점 보정)

로드셀에 아무 하중도 없을 때의 평균값을 측정하여 `offset`으로 저장.

```
long HX711_GetOffset(void)
{
    long sum = 0;
    for (int i = 0; i < 10; i++)
        sum += HX711_ReadRaw();
    return sum / 10;
}
```

6.2 Scale Calibration (스케일 보정)

기준 중량(예: 1kg)을 올려서 변화량을 측정,
비례 상수 `scale` 계산:

$$scale = \frac{(raw_{known} - offset)}{known\_weight}$$

---

7. 무게 계산 함수

```
float HX711_GetWeight(void)
{
    long raw = HX711_ReadRaw();
    long value = raw - offset;
    float weight = (float)value / scale;
    return weight;
}
```

---

8. 동작 시 주의사항

항목	설명
전원 안정화	HX711은 전원 노이즈에 민감함. 별도 100nF + 10µF 바이패스 필요
변환 주기	10 Hz 모드 권장 (노이즈 억제)
MCU Sleep 시	SCK Low 유지, HX711 자동 절전
온도 변화	장기 Drift 보정 필요
ADC 오버플로	입력 전압 ±20mV 초과 시 포화 발생

---

9. 고급 응용

1. **DMA 기반 시퀀스 수집**
   - FreeRTOS Task 또는 Timer에 의해 주기적 샘플링
2. **Moving Average 필터 적용**
   - `N`회 샘플 평균으로 노이즈 제거
3. **EEPROM Calibration 저장**
   - 오프셋 및 스케일 값 비휘발성 메모리에 보존

---

10. 결론

- HX711은 외부 클럭이 없는 독립형 24bit ADC로,
  MCU가 직접 SCK를 제어하여 데이터를 읽는다.
- 변환 주기(10/80 Hz)는 안정성과 속도의 균형을 결정하며,
  신호선 노이즈 차단 및 전원 안정화가 정확도에 큰 영향을 미친다.
- 펌웨어에서는 타임아웃, 오프셋, 스케일 보정, 필터링을 결합해
  신뢰성 높은 무게 측정 시스템을 구현할 수 있다.

Offset / Scale 보정

1. 개요

HX711 기반 로드셀 시스템에서 **보정(calibration)** 은
센서 출력값(ADC raw data)을 실제 물리 단위(gram, kg 등)로 변환하기 위한
2단계 절차이다.

1. **Offset Calibration** — 무부하(Zero Load) 보정
2. **Scale Calibration** — 기준 중량(Reference Weight)을 통한 비례 스케일 보정

두 보정값은 시스템에 고유하며,
하드웨어 변경(센서 교체, 배선 변경 등) 시 반드시 재측정해야 한다.

---

2. Offset Calibration (제로점 보정)

2.1 목적

무부하 상태에서 ADC 출력이 0에 근접하도록 보정한다.
 이는 로드셀의 영점 전위차 및 회로 오프셋 전류를 제거하기 위함이다.

2.2 절차

1. 로드셀 위에 아무 하중도 올리지 않는다.
2. HX711이 안정된 상태(약 3~5회 샘플링)까지 대기한다.
3. 10~20회 이상 ADC 값을 읽어 평균을 구한다.
4. 이 평균값을 `offset` 으로 저장한다.

2.3 예시 코드

```
long HX711_GetOffset(void)
{
    long sum = 0;
    for (int i = 0; i < 20; i++)
    {
        sum += HX711_ReadRaw();
        HAL_Delay(50);
    }
    return (sum / 20);
}
```

2.4 적용 방식

보정 후 측정 시마다 다음과 같이 적용한다.

$$\text{Raw}_{corr} = \text{Raw}_{read} - \text{Offset}$$

---

3. Scale Calibration (스케일 보정)

3.1 목적

ADC 출력 단위를 실제 질량 단위(g, kg)로 변환하기 위한
 비례 상수(Scale factor) 를 계산한다.

3.2 절차

1. Offset 보정 완료 후, 기준 중량 W_{ref} (예: 1 kg)을 로드셀 위에 올린다.
2. 해당 상태에서 ADC 출력을 10~20회 읽고 평균을 구한다.
3. 다음 계산식으로 스케일을 산출한다.

$$\text{Scale} = \frac{\text{Raw}_{avg} - \text{Offset}}{W_{ref}}$$

1. 이후 실제 측정 시 다음 공식을 사용한다.

$$\text{Weight} = \frac{\text{Raw}_{read} - \text{Offset}}{\text{Scale}}$$

---

4. 예시 코드

```
float HX711_CalibrateScale(float ref_weight)
{
    long raw = 0;
    for (int i = 0; i < 10; i++)
    {
        raw += HX711_ReadRaw();
        HAL_Delay(50);
    }
    raw /= 10;
    float scale = (float)(raw - offset) / ref_weight;
    return scale;
}
```

측정 시:

```
float HX711_GetWeight(void)
{
    long raw = HX711_ReadRaw();
    long value = raw - offset;
    float weight = (float)value / scale;
    return weight;
}
```

---

5. 장기 Drift 보정

로드셀은 **온도, 장력 피로, 기계적 히스테리시스** 등의 영향으로
 시간에 따라 영점이 서서히 변할 수 있다.

이를 보완하기 위해 다음 방법을 병행한다.

항목	설명
주기적 Zero-Offset 재보정	무부하 상태에서 일정 주기로 offset 재측정
Auto-Tare 알고리즘	하중이 일정 시간 이상 0 근처일 경우 offset 보정 자동 수행
EEPROM 저장	offset / scale 값을 비휘발성 메모리에 저장 및 부팅 시 복원
평균 필터	샘플 단위 노이즈 및 미세 드리프트 완화

---

6. EEPROM 저장 예시

```
typedef struct {
    long offset;
    float scale;
} HX711_Calibration;

void Save_Calibration(HX711_Calibration *cal)
{
    EEPROM_Write(0x00, (uint8_t*)cal, sizeof(HX711_Calibration));
}

void Load_Calibration(HX711_Calibration *cal)
{
    EEPROM_Read(0x00, (uint8_t*)cal, sizeof(HX711_Calibration));
}
```

---

7. 결론

- **Offset** 은 측정 기준점을 정렬하는 역할,
- **Scale** 은 단위 변환 계수를 정의하는 역할을 수행한다.
- 두 보정값은 반드시 짝을 이루어야 하며,
  잘못된 조합은 절대 오차 및 비선형성을 유발한다.

정확한 보정을 위해서는 **열 안정화**, **기계적 평형 유지**,
전원 노이즈 차단 이 필수적이다.

• 평균 필터링 (Moving Average)

1. 개요

Moving Average Filter (이동 평균 필터) 는
센서 측정값의 단기 노이즈(thermal noise, quantization noise 등)를 억제하기 위해
가장 널리 사용되는 **선형 저역통과 필터(Low-Pass Filter)** 이다.

과거 N개의 샘플을 순차적으로 평균내어
출력값의 단기 변동을 완화하고, 안정적인 측정치를 제공한다.

---

2. 기본 원리

2.1 수학적 정의

$$y[n] = \frac{1}{N} \sum_{k=0}^{N-1} x[n-k]$$

여기서

- $x[n]$: 입력 시퀀스 (센서 측정값)
- $y[n]$: 필터링된 출력

- N: 윈도우 크기 (샘플 개수)

윈도우 크기 N이 클수록 노이즈 억제는 강하지만,
응답 속도(Response time)는 느려진다.

---

3. 동작 예시

입력 시퀀스 (노이즈 포함)

```
100, 103, 99, 101, 102, 98, 100, 104
```

4포인트 평균 필터 적용 결과

```
(100+103+99+101)/4 = 100.75
(103+99+101+102)/4 = 101.25
(99+101+102+98)/4 = 100.00
...
```

노이즈 피크가 완화되어
출력값이 점진적으로 안정화된다.

---

4. MCU 구현 방식

4.1 단순 평균 (Naive Implementation)

```
#define FILTER_SIZE 10
long buffer[FILTER_SIZE];
uint8_t index = 0;

long MovingAverage(long new_value)
{
    buffer[index++] = new_value;
    if (index >= FILTER_SIZE) index = 0;

    long sum = 0;
    for (int i = 0; i < FILTER_SIZE; i++)
        sum += buffer[i];

    return sum / FILTER_SIZE;
}
```

단점: `O(N)` 반복합 계산 → CPU 부하 증가.

4.2 누적합 기반 최적화 (Incremental)

```
#define FILTER_SIZE 10
long buffer[FILTER_SIZE];
uint8_t index = 0;
long total = 0;

long MovingAverage_Optimized(long new_value)
{
    total -= buffer[index];
    buffer[index] = new_value;
    total += new_value;
    index++;
    if (index >= FILTER_SIZE) index = 0;

    return total / FILTER_SIZE;
}
```

특징:

- 연산량 `O(1)`
- 실시간 필터링에 적합
- FreeRTOS Task, Timer ISR, DMA 콜백 등에서 효율적

---

5. 필터 창(Window) 크기 선택 기준

윈도우 크기 (N)	특징	권장 적용
4~8	빠른 응답, 약한 필터링	실시간 거리 센서, 트리거 검출
10~20	안정적 평균, 적당한 응답 속도	로드셀, 온도 센서
50 이상	매우 안정적, 느린 응답	환경 측정, 장기 로그용

---

6. 고급 응용

6.1 이동 표준편차 계산 (Noise 측정용)

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \bar{x})^2}$$

이를 통해 **측정 안정도(Stability)** 또는 **품질 관리(Noise Level Detection)** 가능.

6.2 가중 이동 평균 (Weighted Moving Average)

최근 샘플에 더 큰 가중치를 부여:

$$y[n] = \frac{\sum_{k=0}^{N-1} w_k x[n-k]}{\sum_{k=0}^{N-1} w_k}$$

가중치 w_k는 최근값일수록 크게 설정한다.
(예: $w = [1, 2, 3, 4]$)

---

7. 예시: 로드셀 측정 루틴 통합

```
float HX711_GetStableWeight(void)
{
    long raw = HX711_ReadRaw();
    long filtered = MovingAverage_Optimized(raw);
    float weight = (float)(filtered - offset) / scale;
    return weight;
}
```

---

8. 성능 특성 요약

항목	영향
노이즈 제거	높음
응답 지연	N/2 샘플
연산 복잡도	낮음 (O(1))
메모리 사용량	N × 데이터 크기
위상 왜곡	없음 (선형 위상)

---

9. 결론

- **Moving Average** 는 구현이 단순하면서도
  노이즈 억제 성능이 우수한 1차 저역통과 필터이다.
- 샘플 수 N의 선택은 **노이즈 환경 vs 응답성** 의 균형에 따라 결정한다.
- 실시간 제어계에서는 누적합 방식의 최적화 버전이 권장된다.
- 로드셀, 온도, 압력, 거리 측정 등 대부분의 센서 인터페이스에
  기본 필터 레이어로 사용된다.

- # 무게 계산 함수 `HX711_Get_Value()`

1. 개요

HX711 은 로드셀(저항형 스트레인게이지)의 미세 전압을
24bit ADC (ΔΣ 타입, Sigma-Delta) 로 변환하여
디지털 신호로 출력하는 전용 A/D 변환기이다.

`HX711_Get_Value()` 함수는
이 디지털 출력을 읽어 보정(Offset/Scale)을 적용하고,
실제 무게(Weight, g 단위) 로 변환하는 핵심 루틴이다.

---

2. 기본 동작 순서

1. **데이터 레디 신호 대기 (DOUT Low)**
   HX711은 변환 완료 시 DOUT 핀을 Low로 유지한다.
   이때까지는 변환이 진행 중이며, 읽으면 안 된다.
2. **24bit 데이터 읽기 (MSB First)**
   `CLK` 핀을 24회 토글하며 각 비트를 순차적으로 수신한다.
   DOUT에서 24비트를 읽어 합산한 뒤, **부호 확장(Sign Extend)** 처리한다.
3. **게인 설정을 위한 추가 CLK 토글 (25~27회차)**
   마지막 클럭 입력 개수에 따라 다음 변환의 **Gain/Channel** 이 설정된다.
   - 25회: Channel A, Gain 128
   - 26회: Channel B, Gain 32
   - 27회: Channel A, Gain 64
4. **Offset / Scale 보정 적용**
   - **Offset**: 무게가 0일 때의 ADC 값 (영점 보정값)
   - **Scale**: 1g당 ADC 변화량 (비례 계수)
5. **무게 계산 및 반환**

---

3. 데이터 읽기 루틴 예시

```
#define HX711_CLK_Pin   GPIO_PIN_1
#define HX711_DOUT_Pin  GPIO_PIN_2
#define HX711_GPIO_Port GPIOA

long HX711_ReadRaw(void)
{
    while (HAL_GPIO_ReadPin(HX711_GPIO_Port, HX711_DOUT_Pin)); // DOUT Low 대기

    long value = 0;
    for (int i = 0; i < 24; i++)
    {
        HAL_GPIO_WritePin(HX711_GPIO_Port, HX711_CLK_Pin, GPIO_PIN_SET);
        value = (value << 1) | HAL_GPIO_ReadPin(HX711_GPIO_Port, HX711_DOUT_Pin);
```

```
        HAL_GPIO_WritePin(HX711_GPIO_Port, HX711_CLK_Pin, GPIO_PIN_RESET);
    }

    // 추가 클럭 1회 → 채널 A, Gain 128 유지
    HAL_GPIO_WritePin(HX711_GPIO_Port, HX711_CLK_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(HX711_GPIO_Port, HX711_CLK_Pin, GPIO_PIN_RESET);

    // 부호 확장 (24bit → 32bit)
    if (value & 0x800000)
        value |= 0xFF000000;

    return value;
}
```

---

4. Offset / Scale 보정식

$$Weight = \frac{(Raw - Offset)}{Scale}$$

- **Raw** : HX711 24bit 원시 데이터
- **Offset** : 영점 보정값 (tare 측정)
- **Scale** : 1g당 ADC 변화량 (Calibration)

Calibration 시 다음 절차를 수행한다.

```
void HX711_Calibrate(long known_weight)
{
    long raw_now = HX711_ReadRaw();
    Scale = (float)(raw_now - Offset) / known_weight;
}
```

---

5. 무게 계산 함수 구현

```
float HX711_Get_Value(void)
{
    long raw = HX711_ReadRaw();
    float weight = (float)(raw - Offset) / Scale;
    return weight;
}
```

특징:

- 단일 호출로 실시간 무게 측정 가능
- 보정값(Offset, Scale)은 EEPROM 또는 Flash에 저장 가능
- 필터 적용 시 `MovingAverage_Optimized()` 와 병합하여 안정화 가능

---

6. 고급형: 평균 필터 포함 버전

```
#define AVG_SAMPLES 10

float HX711_Get_Value(void)
{
    long sum = 0;
    for (int i = 0; i < AVG_SAMPLES; i++)
        sum += HX711_ReadRaw();

    long avg = sum / AVG_SAMPLES;
    float weight = (float)(avg - Offset) / Scale;
    return weight;
}
```

- 노이즈 제거 및 안정된 값 확보
- HX711 내부 변환 속도(10~80Hz)에 맞춰 딜레이 조정 필요

---

7. 동작 시퀀스 요약

단계	동작	설명
1	DOUT Low 대기	변환 완료 감지
2	24bit 데이터 수신	MSB부터 순차 읽기
3	부호 확장	24bit → 32bit 변환
4	Offset 보정	영점 보정 제거
5	Scale 변환	실제 무게로 변환
6	반환	g 또는 kg 단위 출력

---

8. 주의사항

- DOUT가 High일 때 데이터 읽으면 **잘못된 측정** 발생
- HX711은 내부 필터로 인해 약 10~80Hz 속도로만 업데이트됨
- 전원 노이즈, ADC 클럭 지터에 민감하므로 GND 평면 확보 필요
- 로드셀 브리지 전압(VBG)은 5V 기준, 전압 강하 시 Scale 값 변동 가능

---

9. 결론

`HX711_Get_Value()` 는
24bit ADC 데이터 획득 → 보정 → 실무 단위 변환 의
표준 절차를 구현한 핵심 함수이다.
펌웨어 구조상 다음과 같이 위치한다:

```
Application
 ├─ Task_Sensor()
 |    └─ HX711_Get_Value()
 ├─ Filter()
 |    └─ MovingAverage()
 ├─ Calibration()
 |    └─ HX711_SetOffset(), HX711_SetScale()
```

즉, 단순한 ADC 리더가 아니라
보정된 물리량 변환기(Weight Converter) 역할을 수행한다.

6.3 ToF 센서 (VL53L0X)

- # I²C 레지스터 접근

1. 개요

대부분의 I²C 디바이스(센서, 메모리, 디지털 포트 확장 등)는
내부 레지스터 맵(Register Map) 을 갖는다.
호스트 MCU(STM32)는 I²C 프로토콜을 통해
해당 레지스터 주소를 지정하고,
데이터를 읽거나 쓰는 방식으로 제어한다.

이 접근 방식은 SPI의 "주소 + 데이터 프레임"과 유사하지만,
I²C에서는 **Start** → **Slave Address** → **Register Address** → **Data** 의
일련의 시퀀스로 수행된다.

---

2. 레지스터 접근 시퀀스

2.1 레지스터 쓰기(Write)

단계	설명
①	**Start** 조건 발생
②	Slave Address + Write(0) 전송
③	Slave의 **ACK** 수신
④	Register Address 전송
⑤	Slave의 **ACK** 수신
⑥	Data Byte 전송
⑦	Slave의 **ACK** 수신
⑧	**Stop** 조건 전송

즉,

$$\text{Start} \rightarrow \text{Addr(W)} \rightarrow \text{RegAddr} \rightarrow \text{Data} \rightarrow \text{Stop}$$

2.2 레지스터 읽기(Read)

단계	설명
①	**Start** 조건 발생
②	Slave Address + Write(0) 전송
③	Slave의 ACK 수신
④	Register Address 전송
⑤	**Repeated Start** 발생
⑥	Slave Address + Read(1) 전송
⑦	Slave의 ACK 수신
⑧	Slave로부터 Data 수신
⑨	Master에서 NACK + **Stop**

즉,

$$\text{Start} \rightarrow \text{Addr(W)} \rightarrow \text{RegAddr} \rightarrow \text{Repeated Start} \rightarrow \text{Addr(R)} \rightarrow \text{Data}$$

---

3. HAL 기반 접근 함수

STM32 HAL에서는 위 과정을 자동화한
레지스터 단위 접근 API 를 제공한다.

3.1 레지스터 읽기 (`HAL_I2C_Mem_Read()`)

```
HAL_StatusTypeDef HAL_I2C_Mem_Read(
    I2C_HandleTypeDef *hi2c,
    uint16_t DevAddress,
    uint16_t MemAddress,
    uint16_t MemAddSize,
    uint8_t *pData,
    uint16_t Size,
    uint32_t Timeout
);
```

파라미터 설명

인자	설명
`DevAddress`	7-bit 디바이스 주소 (왼쪽 시프트 필요 없음)
`MemAddress`	내부 레지스터 주소

인자	설명
`MemAddSize`	`I2C_MEMADD_SIZE_8BIT` or `I2C_MEMADD_SIZE_16BIT`
`pData`	수신 버퍼 포인터
`Size`	수신 바이트 수
`Timeout`	타임아웃(ms)

3.2 레지스터 쓰기 (`HAL_I2C_Mem_Write()`)

```
HAL_StatusTypeDef HAL_I2C_Mem_Write(
    I2C_HandleTypeDef *hi2c,
    uint16_t DevAddress,
    uint16_t MemAddress,
    uint16_t MemAddSize,
    uint8_t *pData,
    uint16_t Size,
    uint32_t Timeout
);
```

특징

- 내부적으로 `Start → Write → RegAddr → Data → Stop` 순서 수행
- 단일 바이트 또는 버스트 쓰기 모두 지원

---

4. 예시: VL53L0X 레지스터 접근

4.1 레지스터 쓰기 예시

```
uint8_t data = 0x01;
HAL_I2C_Mem_Write(&hi2c1, 0x52, 0x000E, I2C_MEMADD_SIZE_16BIT, &data, 1, 100);
```

- 디바이스 주소: `0x52` (7-bit 주소 0x29 << 1)
- 레지스터 주소: `0x000E`
- 데이터: `0x01`

4.2 레지스터 읽기 예시

```
uint8_t data;
HAL_I2C_Mem_Read(&hi2c1, 0x52, 0x0014, I2C_MEMADD_SIZE_16BIT, &data, 1, 100);
```

이 호출은 다음 I²C 프레임으로 변환된다:

```
START → 0x52(W) → 0x00 → 0x14 → RESTART → 0x53(R) → [DATA] → STOP
```

---

5. 일반적인 패턴

센서의 데이터시트는 다음과 같이 구성된다:

Register	Name	R/W	Description
0x000E	SYSTEM_MODE_START	W	측정 시작 트리거
0x0014	RESULT_RANGE_STATUS	R	거리 측정 상태
0x0016	RESULT_DISTANCE	R	측정된 거리 값 (mm)

이 경우 펌웨어는 다음과 같이 접근한다.

```
void VL53L0X_StartMeasurement(void)
{
    uint8_t cmd = 0x01;
    HAL_I2C_Mem_Write(&hi2c1, 0x52, 0x000E, I2C_MEMADD_SIZE_16BIT, &cmd, 1, 100);
}

uint16_t VL53L0X_ReadDistance(void)
{
    uint8_t buf[2];
    HAL_I2C_Mem_Read(&hi2c1, 0x52, 0x0016, I2C_MEMADD_SIZE_16BIT, buf, 2, 100);
    return (buf[0] << 8) | buf[1];
}
```

---

6. 레지스터 접근 에러 처리

에러 코드	의미	조치
`HAL_TIMEOUT`	Slave 응답 없음	배선 확인, 전원 안정성 점검
`HAL_ERROR`	NACK 응답	주소/레지스터 오류
`HAL_BUSY`	버스 점유 중	지연 후 재시도 또는 `HAL_I2C_DeInit()` 후 복구
`HAL_OK`	정상 통신	데이터 유효

---

7. 디버깅 팁

- 오실로스코프로 SDA/SCL 파형 확인 시
  Start / Repeated Start / ACK / Stop 파형을 반드시 검증한다.
- 주소 NACK 발생 시,
  7-bit 주소가 `<<1` 되어 있는지(즉, `addr << 1`) 확인한다.
- 멀티 슬레이브 환경에서는 버스 충돌 시
  Stop 조건 미발생 → **Bus Hang** 상태가 될 수 있다.

---

8. 결론

I²C 레지스터 접근은
"**메모리 맵 기반 명령 전송 구조**"이며,
HAL의 `Mem_Read()` / `Mem_Write()` API를 통해
센서 내부 상태 제어 및 데이터 취득을 표준화할 수 있다.

이 접근 방식은
VL53L0X, MPU6050, SHT3x, BMP280 등
대부분의 I²C 센서 드라이버에서 공통적으로 사용된다.

- # 초기화 및 Calibration

1. 개요

센서 초기화(Initialization)는 전원 인가 후
정상적인 동작 상태를 보장하기 위한 일련의 설정 과정이다.
Calibration(보정)은 센서의 출력값을 기준 상태로 정렬하여
정확한 물리량을 산출하도록 하는 단계이다.

이 두 과정은 시스템 부팅 시 가장 먼저 수행되며,
센서의 안정성·정확도·재현성을 결정짓는 핵심 절차이다.

---

2. 초기화 (Initialization)

초기화는 하드웨어 및 내부 레지스터의 기본 상태를
유효한 측정 모드로 설정하는 과정이다.

2.1 일반 순서

단계	항목	설명
①	**전원 안정화 대기**	센서 전원 인가 후 내부 회로 안정화(보통 10~50ms)
②	**통신 인터페이스 초기화**	I²C/SPI/UART 설정, GPIO 모드 설정
③	**디바이스 ID 확인**	WHO_AM_I / ID 레지스터 읽기 후 장치 식별
④	**리셋 명령 전송 (선택적)**	SW_RESET 또는 POWER_ON_RESET 수행
⑤	**기본 레지스터 설정**	측정 모드, 샘플링 주기, 게인, 오프셋 등 초기값 적용
⑥	**Status 확인**	Boot 완료 플래그, Ready 비트 확인
⑦	**Self-test 수행 (선택적)**	내부 기준 회로 또는 참조 값 비교 검사
⑧	**Calibration 단계 진입**	이후 단계에서 오프셋, 스케일, 기준점 조정 수행

2.2 HAL 기반 예시

```
void Sensor_Init(void)
{
    uint8_t id = 0;

    // (1) 장치 ID 확인
    HAL_I2C_Mem_Read(&hi2c1, DEV_ADDR, REG_ID, I2C_MEMADD_SIZE_8BIT, &id, 1, 100);
    if (id != EXPECTED_ID) Error_Handler();

    // (2) 리셋
    uint8_t reset_cmd = 0x01;
    HAL_I2C_Mem_Write(&hi2c1, DEV_ADDR, REG_RESET, I2C_MEMADD_SIZE_8BIT,
&reset_cmd, 1, 100);
    HAL_Delay(10);

    // (3) 측정 파라미터 설정
    uint8_t config = 0x30;
    HAL_I2C_Mem_Write(&hi2c1, DEV_ADDR, REG_CONFIG, I2C_MEMADD_SIZE_8BIT, &config,
1, 100);
}
```

---

3. Calibration (보정)

보정은 센서의 원시(raw) 출력값을
실제 물리량과 일치하도록 조정하는 절차이다.
보정 방식은 센서의 종류에 따라 다르지만,
일반적으로 다음 세 가지 요소를 포함한다.

3.1 Offset 보정

- 센서 출력의 **영점 오차(zero offset)** 제거
- 입력이 0일 때의 측정값을 기록하고,
  모든 측정값에서 이를 감산

$$\text{Corrected Value} = \text{Raw Value} - \text{Offset}$$

예: HX711 무게센서에서 빈 저울판 상태의 ADC 평균값을 Offset으로 저장

3.2 Scale 보정 (Gain Calibration)

- 기준 물리량(known reference)을 이용하여 스케일링 비율 계산
- 실제 물리량과 측정값 간의 비례 관계를 보정

$$\text{Scale} = \frac{\text{Reference Weight}}{\text{Raw Output Difference}}$$

3.3 Temperature Compensation (온도 보정)

- 온도에 따른 출력 드리프트를 보정하기 위해
  내부 또는 외부 온도센서 값을 함께 사용
- 보정식 예시:

$$V_{corr} = V_{meas} - k_T(T - T_{ref})$$

여기서
`k_T` = 온도계수,
`T_ref` = 기준 온도 (보정 기준)

---

4. 예시: HX711 무게센서

```
typedef struct {
    long offset;
    float scale;
} HX711_Calibration_t;

HX711_Calibration_t cal;

void HX711_Calibration(void)
{
    long raw_empty = HX711_Get_Value(10);
    cal.offset = raw_empty;

    // 기준추 1000g 올림
    long raw_load = HX711_Get_Value(10);
    cal.scale = 1000.0f / (raw_load - raw_empty);
}

float HX711_Get_Weight(void)
{
    long raw = HX711_Get_Value(5);
    return (raw - cal.offset) * cal.scale;
}
```

---

5. 예시: VL53L0X 거리센서

VL53L0X는 전원 인가 후 다음 절차로 Calibration 수행한다:

단계	내용
①	SPAD 보정 (SPAD calibration)
②	Reference SPAD 및 crosstalk 측정
③	VHV Calibration
④	Temperature offset 보정

단계	내용
⑤	Range offset 적용

STM32 펌웨어에서는 ST의 API를 호출하여 수행한다.

```
VL53L0X_StaticInit(&dev);
VL53L0X_PerformRefSpadManagement(&dev, &count, &is_aperture);
VL53L0X_PerformRefCalibration(&dev, &vhv_settings, &phase_cal);
VL53L0X_SetReferenceSpads(&dev, count, is_aperture);
```

---

6. Calibration 결과 저장

보정 후 계산된 `Offset`, `Scale`, `TempCoeff` 등은
비휘발성 메모리(NVRAM, Flash, EEPROM) 에 저장해
다음 부팅 시 재사용해야 한다.

예시:

```
typedef struct {
    int32_t offset;
    float scale;
} SensorCal_t;

SensorCal_t cal_data __attribute__((section(".cal_data")));

void Save_Calibration(void)
{
    Flash_Write(CAL_ADDR, (uint8_t*)&cal_data, sizeof(cal_data));
}
```

---

7. 결론

센서의 **Initialization**과 **Calibration**은
단순 설정이 아닌,
시스템 신뢰도를 결정하는 핵심 단계이다.

초기화는 하드웨어를 동작 가능한 상태로 만들고,
Calibration은 실제 환경에 대한 정밀 보정을 수행한다.

두 절차가 올바르게 수행되어야만
측정값은 **선형성, 반복성, 정확성**을 확보할 수 있다.

단발/연속 측정 모드

1. 개요

센서의 측정 방식은 일반적으로
단발(Single-shot) 과 **연속(Continuous)** 두 가지 모드로 구분된다.

이 두 모드는 MCU의 제어 구조, 전력 소모, 응답 지연(latency),
데이터 처리량 등의 트레이드오프를 결정하는 핵심 파라미터다.

---

2. 단발 측정 모드 (Single-shot Mode)

2.1 정의

단발 모드는 MCU가 명령을 내릴 때마다
센서가 한 번의 측정 시퀀스를 수행하고 결과를 반환하는 방식이다.
즉, **측정 요청 → 변환 수행 → 결과 반환 → 대기 상태**의 구조다.

2.2 특징

항목	설명
트리거 방식	MCU가 명령으로 직접 트리거
소비 전력	매우 낮음 (대기 중 Sleep)
응답 속도	변환 시간만큼 지연 발생
측정 주기	MCU가 직접 제어
사용 예	저전력 시스템, 간헐 측정 장치 (배터리 기반)

2.3 일반 시퀀스

1. **Start 조건 발생**
2. **Slave Address(W)** 전송
3. **측정 시작 레지스터(예: 0x00, 0x18 등)** 에 Trigger 명령 전송
4. **변환 완료 대기 (Delay or Status Polling)**
5. **결과 레지스터 읽기**
6. **Stop 조건**

2.4 예시 (VL53L0X 거리센서)

```
void VL53L0X_SingleShot(void)
{
    uint8_t cmd = 0x01; // Trigger single measurement
    HAL_I2C_Mem_Write(&hi2c1, 0x52, 0x000E, I2C_MEMADD_SIZE_16BIT, &cmd, 1, 100);

    // 변환 완료 대기
    HAL_Delay(30);

    uint8_t buf[2];
    HAL_I2C_Mem_Read(&hi2c1, 0x52, 0x0016, I2C_MEMADD_SIZE_16BIT, buf, 2, 100);
    uint16_t distance = (buf[0] << 8) | buf[1];
}
```

---

3. 연속 측정 모드 (Continuous Mode)

3.1 정의

연속 모드는 센서 내부에서 **주기적 측정 루프**가 자동 수행되는 방식이다.
MCU는 최초에 “연속 측정 시작” 명령을 보내고,
이후 결과 레지스터를 반복적으로 읽어들인다.

센서는 변환 완료 후 자동으로 다음 측정을 개시한다.

3.2 특징

항목	설명
트리거 방식	초기 1회 명령 후 자동 반복
소비 전력	상대적으로 높음
응답 속도	일정한 주기로 즉시 결과 제공
측정 주기	센서 내부 타이머 또는 설정된 인터벌에 따름
사용 예	실시간 거리 측정, 움직임 추적, 데이터 스트리밍

3.3 일반 시퀀스

1. **Start 조건 발생**
2. **Slave Address(W)** 전송
3. **연속 모드 설정 레지스터에 모드 비트 작성**
4. **센서 내부에서 주기적 변환 수행**
5. **MCU는 일정 주기로 결과 레지스터 Polling**
6. **필요 시 Stop 명령으로 종료**

3.4 예시 (VL53L0X 연속 모드)

```
void VL53L0X_ContinuousStart(void)
{
    uint8_t cmd = 0x02; // Continuous mode command
    HAL_I2C_Mem_Write(&hi2c1, 0x52, 0x000E, I2C_MEMADD_SIZE_16BIT, &cmd, 1, 100);
}

uint16_t VL53L0X_ContinuousRead(void)
{
    uint8_t buf[2];
    HAL_I2C_Mem_Read(&hi2c1, 0x52, 0x0016, I2C_MEMADD_SIZE_16BIT, buf, 2, 100);
    return (buf[0] << 8) | buf[1];
}

void VL53L0X_ContinuousStop(void)
{
    uint8_t cmd = 0x00; // Stop command
    HAL_I2C_Mem_Write(&hi2c1, 0x52, 0x000E, I2C_MEMADD_SIZE_16BIT, &cmd, 1, 100);
}
```

---

4. 모드 비교 요약

구분	단발 모드 (Single)	연속 모드 (Continuous)
트리거 제어	MCU 수동 명령	자동 반복
응답 지연	있음 (측정 시간 필요)	거의 없음
전력 소비	매우 낮음	상대적으로 높음
데이터율	불규칙 (MCU 주기 의존)	일정 (센서 타이밍 기준)
제어 복잡도	간단	중간 (중단 조건 필요)
권장 환경	배터리 기반, 저속 시스템	실시간 모니터링, 연속 처리 시스템

---

5. 인터럽트 기반 연속 측정

일부 센서는 측정 완료 시 **Interrupt Pin (INT, DRDY)** 를 제공한다.
MCU는 I²C Polling 대신 외부 인터럽트를 통해
데이터 갱신 시점을 감지할 수 있다.

```c
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if (GPIO_Pin == SENSOR_INT_Pin)
    {
        uint8_t buf[2];
        HAL_I2C_Mem_Read(&hi2c1, DEV_ADDR, REG_RESULT, I2C_MEMADD_SIZE_8BIT, buf, 2, 100);
        uint16_t value = (buf[0] << 8) | buf[1];
    }
}
```

---

6. 종료 절차

연속 모드 사용 시 반드시
정상 종료(Stop Command) 를 통해
센서를 대기 상태로 되돌려야 한다.

이는 I²C 버스 안정성 유지 및
센서 발열, 전력 누적 방지를 위해 필수적이다.

---

7. 결론

- **단발 모드**는 저전력, 저주기 환경에 최적
- **연속 모드**는 실시간, 고속 샘플링에 적합
- 대부분의 I²C 센서는 두 모드를 모두 지원하며,
  초기화 시점에서 선택적으로 설정 가능하다.

센서의 레지스터 맵에서
`MODE`, `CTRL`, `POWER` 비트를 정확히 해석하고
HAL I²C 함수로 제어 시퀀스를 구현하는 것이 핵심이다.

• 신호율, VCSEL 파라미터 이해

(Understanding Signal Rate and VCSEL Parameters in Time-of-Flight Sensors)

---

1. 개요

VL53L0X와 같은 **ToF(Time-of-Flight)** 센서는
VCSEL(Vertical Cavity Surface Emitting Laser)을 광원으로 사용하며,
반사된 신호를 SPAD(Single Photon Avalanche Diode) 어레이로 수광하여
비행 시간(Time of Flight) 을 계산한다.

이 과정에서 측정 품질, 측정 거리, 노이즈 수준은
VCSEL 파라미터 설정과 **신호율(Signal Rate)** 에 의해 결정된다.

---

2. 신호율 (Signal Rate)

2.1 정의

신호율은 수신기에 검출된 **유효 포톤(Event)의 비율**을 나타내는 물리적 지표로,
단위는 **Mcps (Mega counts per second)** 로 표현된다.

이는 센서가 수신한 반사광 강도와 반비례하여
노이즈, 반사율, 거리, 표면 상태 등의 영향을 받는다.

$$\text{Signal Rate} = \frac{\text{Detected Photon Events}}{\text{Integration Time}}$$

2.2 신호율의 의미

구분	설명
높은 신호율 (High Signal Rate)	반사광이 강함 → 근거리 / 고반사 표면
낮은 신호율 (Low Signal Rate)	반사광이 약함 → 원거리 / 저반사 표면
Zero Signal Rate	측정 불가 또는 과도한 외란(외광, 포화 등)

2.3 신호율의 역할

1. **측정 유효성 판단**
   - 센서는 신호율이 특정 임계값(`SignalRateLimit`) 이상일 때만 유효 측정으로 간주한다.
2. **거리 보정 인자 계산**
   - 신호율은 거리 계산 알고리즘의 Gain Factor로 사용된다.
3. **자동 노출 제어 (Auto Exposure)**
   - 신호율이 낮을 경우, 통합시간(Integration Time) 또는 VCSEL Pulse 수가 증가한다.

2.4 기본 설정 예시

파라미터	권장 값	설명
`SignalRateLimit`	0.25 Mcps	최소 유효 신호율
`Reference Signal Rate`	1.0 Mcps	공장 캘리브레이션 기준
`AmbientRate`	≤ 0.1 Mcps	주변광 영향 허용치

---

3. VCSEL 파라미터 (VCSEL Configuration Parameters)

3.1 VCSEL 개요

VCSEL(Vertical Cavity Surface Emitting Laser)은
수직 공진 구조를 갖는 반도체 레이저로,
ToF 센서의 송신부에서 짧은 펄스를 방출한다.

VL53L0X의 VCSEL은

- 파장(Wavelength): 약 **940 nm (근적외선)**
- 펄스폭(Pulse Width): 수십 ns
- 반복 주기(Pulse Period): 수백 μs 수준

으로 동작한다.

3.2 주요 설정 파라미터

항목	레지스터	설명
VCSEL Period Pre-Range	0x50	사전 거리측정(Pre-range) 구간의 VCSEL 펄스 주기
VCSEL Period Final-Range	0x70	최종 거리측정(Final-range) 구간의 VCSEL 펄스 주기
Laser Pulse Count	0x52 / 0x72	각 구간당 VCSEL 펄스 개수
Cross-Talk Compensation	0x20~0x22	내부 반사/렌즈 누설 보정
VCSEL Calibration Register	0xB0~0xB3	VCSEL 전류 및 위상 정렬값

VCSEL Period는 **타임베이스 단위(TIMEOUT_MClks)** 로 설정되며,
짧을수록 더 짧은 펄스 주기(고주파 펄스)를 의미한다.
이는 **해상도 향상**에 유리하나, **소비전력 증가 및 발열 증가**의 트레이드오프를 가진다.

3.3 VCSEL 파라미터 튜닝의 영향

설정 항목	값 변화 방향	영향
VCSEL Period ↓	더 짧은 펄스	거리 분해능 향상, 발열 증가
VCSEL Period ↑	더 긴 펄스	장거리 감도 향상, 속도 저하
Pulse Count ↑	더 많은 샘플	안정도 향상, 응답 지연 증가
Pulse Count ↓	적은 샘플	응답 속도 향상, 노이즈 증가
Cross-talk Compensation ↑	반사 억제	근거리 정밀도 향상
Cross-talk Compensation ↓	억제 약화	장거리 감도 향상

---

4. 신호율과 VCSEL의 상관 관계

신호율은 VCSEL 구동 조건과 직접적으로 연동된다.

$$\text{Signal Rate} \propto \text{VCSEL Power} \times \text{Target Reflectivity} \times \frac{1}{\text{Distance}^2}$$

따라서 VCSEL Period와 Pulse Count를 조정하면
거리 감도와 반사 신호 세기를 균형 있게 제어할 수 있다.

예를 들어,

- **근거리 고정밀 측정:** 짧은 VCSEL Period, 낮은 Pulse Count
- **장거리 탐지:** 긴 VCSEL Period, 높은 Pulse Count

구조로 설정된다.

---

5. ST의 캘리브레이션 절차 요약

1. **SPAD Reference Calibration**
   - 내부 감도 기준 설정
2. **XTalk Compensation Calibration**
   - 렌즈 내부 반사 보정
3. **Offset Calibration**
   - 측정 오차 정렬
4. **Signal Rate Reference Calibration**
   - VCSEL 출력과 신호율 간 상관 관계 등록

이 절차는 **공장 초기화 루틴** 혹은
`VL53L0X_PerformRefCalibration()` API 호출을 통해 수행된다.

---

6. 실제 적용 예 (HAL 기반 설정)

```
VL53L0X_SetVcselPulsePeriod(VL53L0X_VCSEL_PERIOD_PRE_RANGE, 14);
VL53L0X_SetVcselPulsePeriod(VL53L0X_VCSEL_PERIOD_FINAL_RANGE, 10);
VL53L0X_SetSignalRateLimit(0.25); // 0.25 Mcps
VL53L0X_PerformRefCalibration(&calData);
```

이 구성은 **정밀도와 측정 속도의 균형**을 위해
ST에서 권장하는 중간 레벨 설정이다.

---

7. 요약

항목	역할	주요 영향
Signal Rate	반사광 강도 판단	유효 측정 여부 결정
VCSEL Period	펄스 주기 제어	거리 분해능 / 감도 조절
Pulse Count	펄스 샘플 수	노이즈 안정성
Cross-Talk	내부 반사 보정	근거리 정확도 향상
Calibration	시스템 정렬	보정 오차 최소화

결론적으로,
신호율(Signal Rate) 은 **측정 신뢰도**를 정량화하는 핵심 지표이며,
VCSEL 파라미터는 센서의 **물리적 거리 성능**을 결정짓는
핵심 설정 변수이다.

이 두 요소의 최적 조합은
응용 목적(정밀 계측 / 장거리 감지 / 저전력)에 따라
별도로 튜닝되어야 한다.

6.4 수위센서 (Capacitive Water Sensor)

- # 20단 정전식 수위 검출

20단 정전식 수위 검출 시스템은 수조 내 높이를 20개의 연속 또는 계단식 감지 지점(segments)으로 분해하여 **디지털 on/off 신호**로 수위 상태를 얻는다.
각 단계는 전극(또는 링, 스트립)으로 구성되고, 액체가 전극에 가까워지면 정전용량이 증가하여 임계값을 넘어설 때 해당 단계가 “ON”으로 보고된다.
이 접근은 단순·저비용이며, NO contact(비접촉)·내구성·다중 레벨 검출에 유리하다.

---

2. 동작 원리(요약)

- 각 전극과 기준 전극(일반적으로 탱크 바닥의 그라운드 또는 레퍼런스 플레이트) 사이의 정전용량을 측정한다.
- 액체(물)는 공기보다 유전율(ε_r)이 크므로 액체가 전극을 감쌀수록 정전용량이 증가.
- 정전용량 증가는 센서 회로(또는 전용 IC)의 측정치 증가로 이어지며, 사전 설정한 **임계값(threshold)** 을 넘으면 해당 채널을 ON으로 판정한다.

---

3. 하드웨어 구현 옵션 (권장 우선순위)

옵션 A — 전용 Capacitive-to-Digital IC (권장)

- 칩 예: AD7150 계열, FDC1004, MPR121(터치), CAP1188 등(제품별 채널/특징 상이).
- 장점: 정밀도·안정성·잡음 저감·다중 채널 지원·I²C 인터페이스.
- 설계: 전극 → 센서 IC 채널 → I²C → MCU.
- 다수 채널(20개)일 경우 멀티-IC(예: FDC1004 4ch ×5개) 또는 멀티플렉서 사용.

옵션 B — MCU 자체 TSC(터치 센싱 컨트롤러) 사용 (STM32)

- STM32 TSC 모듈 사용(가능한 MCU에서).
- 장점: 외부 IC 불필요, 비용 절감.
- 단점: 채널 수·설계 복잡도·보정 로직 필요.

옵션 C — RC 타이밍 / Charge-Transfer 기법(소프트웨어 측정)

- GPIO와 저항을 이용해 전극의 충전/방전 시간(또는 주파수 변화)을 측정(Timer IC)
- 단가 낮음, 그러나 노이즈·온도 영향 크고 캘리브레이션·샘플링 오버헤드 필요.

---

4. 전극(센서) 물리 설계 지침

- 전극 수: 20개(수직 스트립 또는 링 형태). 각 단계 높이 = 전체 측정 높이 ÷ 20.
- 전극 폭/길이: 수조 벽 두께·전극 간격에 따라 결정. 충분한 면적 확보가 감도에 유리.
- 전극 간격: 전기적 커플링 최소화(겹침·너무 가까움 주의).
- 재료: 스테인리스 스틸, 구리 테이프, PCB 패턴(그라운드 쉴드 + 가드 아일 지도 권장).
- 가드 링(guard trace): 전극 주변에 동일 신호의 가드 레이어를 두어 누설을 줄임.
- 절연: 절연층(플라스틱, 에폭시)으로 전극을 보호하되 너무 두꺼우면 감도 감소.
- 케이블/배선: 짧게, 쉴드 사용 권장. 풀업/풀다운 불필요(전용 IC 사용 시).

---

5. 신호 처리 흐름 (IC 또는 MCU 쪽)

1. **샘플링**: 모든 채널에 대해 정기적으로 측정값(정전용량값 또는 카운트)을 읽음.
2. **평활화(필터)**: 이동평균 또는 지수평활(EMA)으로 노이즈 제거. (예: N=8 이동평균)
3. **임계값 비교**: 각 채널에 대해 `meas_value > threshold[channel]` 이면 ON.
4. **디바운스/히스테리시스**: ON→OFF 전환시 다른 임계값(또는 일정 기간 연속 판정)을 사용.
5. **최종 수위 판단**: 가장 높은 인덱스(탱크에서 위쪽 기준)에서 ON인 마지막 채널 `last_on`을 찾음.
6. **거리/높이 계산**: 예) `height_mm = last_on * step_mm + offset_mm` (outline에서 제시한 식: `height = last_on * 5 + 3` 사용 가능).

---

6. 임계값(Threshold) 설정 및 보정

- **초기 캘리브레이션**: 장비 설치 후(빈 탱크 상태) 각 채널의 베이스라인(Baseline)을 측정.
- **threshold[channel] = baseline[channel] + margin** (margin은 허용 신호 대비 노이즈 레벨에 따라 결정; 예: 3σ 또는 고정 오프셋).
- **온도보상**: 온도에 따른 유전율/드리프트 보정을 위해 온도센서 값으로 baseline 보정.
- **오토-캘리브레이션(운영 중)**: 장기간 변동을 대비해 낮은 활동 시간대(예: 시스템 부팅 또는 비사용 기간)에 baseline 업데이트(증분 평균) — 단, 물이 있을 때는 하지 않도록 조건부 수행.
- **히스테리시스 적용**: ON→OFF 임계값을 다르게 하여 깜박임 방지(예: ON_thresh = base + 20, OFF_thresh = base + 10).

---

7. 디바운스 및 로직(예시)

- **연속 판정 요구**: n_on 연속 판정 후 실제 ON으로 반영(예: n_on = 3).
- **타임아웃**: 판정 불가시 이전 유효값 유지 또는 경고 상태로 전환.
- **에러 플래그**: 모든 채널 값이 이상(너무 높거나 낮음)일 때 센서 실패로 처리.

```
// 간단한 의사 코드 (pseudo-C)
#define CHANNELS 20
uint16_t baseline[CHANNELS];
uint16_t thresh[CHANNELS];
```

```
uint16_t buffer[CHANNELS][N];
uint8_t idx = 0;
uint8_t stable_count[CHANNELS];

void process_sample(uint16_t raw[CHANNELS]) {
    for (int ch=0; ch<CHANNELS; ch++) {
        // 이동평균
        uint32_t sum = 0;
        for (int k=0;k<N;k++) sum += buffer[ch][k];
        uint16_t avg = sum / N;

        // 임계값 비교 & 디바운스
        if (avg > thresh[ch]) {
            if (stable_count[ch] < DEBOUNCE_MAX) stable_count[ch]++;
        } else {
            if (stable_count[ch] > 0) stable_count[ch]--;
        }
        bool on = (stable_count[ch] >= DEBOUNCE_N);
        state[ch] = on;
    }
    idx = (idx+1) % N;
}
```

---

8. 수위(mm) 변환식 예시

- 만약 전체 측정 높이 100 mm를 20단으로 나눈다면 `step_mm = 100 / 20 = 5 mm`.
- 예시 (outline에서 제시한 식 활용):

  ```
  height_mm = last_on * 5 + 3
  ```

  여기서 `last_on`은 0~19 인덱스, `+3`은 설치/offset 보정 값(현장 실측에 따라 조정).

---

9. 통신 및 시스템 통합

- MCU와 상위 시스템(제어기/로거)은 I²C/SPI/UART를 통해 다음 정보를 제공:
  - 각 채널의 디지털 상태(20비트 비트맵)
  - 보정된 높이(mm)
  - 상태/에러 플래그(센서 결함, 캘리브레이션 필요 등)
- 다중 센서를 병렬/다중 주소로 구성 시 I²C 주소 충돌 주의 및 버스 복구 로직 적용.

---

10. 잡음·간섭·장애 대응

- **전기적 잡음**: 전원 필터링(LC, RC), 전극에 병렬 콘덴서 금지(감도 저하), 차폐 케이블 권장.
- **외부 전계 간섭**: 주변 고전압/모터 등에서 유도 노이즈 영향 -> 접지/차폐 필요.
- **액체 전도성 변화**: 물의 전도도(염분)에 따라 감도 변화 → 캘리브레이션 필요.

- **결로/스프레이**: 표면에 물방울이 맺히면 false ON 발생 → 전극 위치·형상 재검토 또는 물 튐 방지 구조 권장.

---

11. 성능·사양(설계 시 고려값)

- 분해능: 전극당 1단(예: 5 mm)
- 반복성: ±1~3 mm (설계·환경에 따라 변동)
- 응답 속도: 필터 창·디바운스에 따라 50 ms ~ 500 ms
- 온도계수: 온도 변화 시 baseline drift — 보정 권장
- 전력소모: 전용 IC 사용시 대기전류수준 매우 낮음

---

12. 추천 설계 절차(순서)

1. 전극 레이아웃 설계(20단, step 높이 결정).
2. 전용 Capacitive IC 선정(FDC1004 등) 또는 MCU TSC 사용 여부 결정.
3. 초기 베이스라인 측정 루틴 및 자동 캘리브레이션 구현.
4. 임계값·히스테리시스 값 실험적으로 결정(현장 테스트).
5. 필터·디바운스 파라미터 튜닝 (응답성 vs 안정성 균형).
6. 온도 보상 루틴 추가(필요 시).
7. 통합 테스트(실제 액체 레벨 변화, 흔들림, 잡음 환경).
8. EEPROM/Flash에 캘리브레이션 값 저장 및 부팅 복원.

---

13. 결론 및 권장사항

- **가장 현실적이고 안정적인 방법**은 전용 capacitive-to-digital IC를 사용하고, I²C로 MCU와 통신하는 것.
- 전극의 물리적 설계(크기·절연·가드)의 품질이 결국 측정 신뢰도를 좌우한다.
- 캘리브레이션(빈 탱크 baseline + reference height)은 필수이며, 운영 중 자동 보정 정책을 신중히 설계해야 한다.
- 소프트웨어 측면에서는 **이동평균 + 히스테리시스 + 디바운스** 조합이 실용적이다.

• Dual I²C Address 병합

1. 개요

“Dual I²C Address 병합”은 동일 타입 또는 서로 다른 센서가 **두 개(또는 다수)의 I²C 주소**를 갖는 경우, MCU 쪽에서 이들 주소의 데이터를 합쳐(merge) 하나의 의미 있는 센서 값으로 만들거나, 동일 주소 충돌을 해결하기 위한 설계 기법을 말한다.
목적은 신뢰성 향상(중복 측정), 분해능 확대(세그먼트 병합), 주소 충돌 회피 또는 다중 채널 센서 통합이다.

---

2. 사용 시나리오 분류

1. **독립 장치(동일 기능) → 데이터 융합**
   - 예: 동일형 수위센서 2개(주소 0x48, 0x49)를 병렬로 설치해 노이즈/신뢰도 보강.
2. **하나의 물리적 센서가 논리적으로 2주소 제공 → 채널 병합**
   - 예: 대형 정전식 패널이 A, B 두 주소로 분할 출력 → 전체 레벨 계산.
3. **동일 주소 복수장치(하드웨어 주소 변경 불가) → 멀티플렉서 사용**
   - 예: 여러 동일 I²C 장치를 동일 버스에 연결할 수 없을 때 TCA9548A 등 사용.
4. **주소 변경 가능한 장치 → 하드웨어로 주소 재할당 후 단순 읽기**
   - A0/A1 핀, XSHUT 등으로 주소 바꿔 운영.

---

3. 병합(데이터 융합) 전략 — 원칙과 기법

3.1 검증(Validity) 우선

- 각 장치로부터 읽은 값에 대해 **타임스탬프, 상태 플래그(에러/온도/신호율)** 를 확인.
- 유효성 검사: NACK, 타임아웃, 범위밖 값(예: 음수, 과포화) 등은 배제.

3.2 병합 방식 (대표)

- **단순 평균(Average)** : noise가 랜덤일 때 효과적.
  `merged = (v1 + v2) / 2`
- **가중 평균(Weighted average)** : 센서별 신뢰도(예: signal_rate, SNR)에 따라 가중치 부여.
  `merged = (w1*v1 + w2*v2) / (w1 + w2)`
- **최우선(valid-first) 선택** : 한 쪽이 유효하고 다른 쪽이 타임아웃이면 유효 값 채택.
- **다중검출/다수결(Majority) / Median** : 임계치 이상일 때 이상치 제거.
- **센서 융합(센서퓨전)** : 칼만 필터, 가중치 필터 등 동적 필터 적용(시간상관성 고려).

3.3 히스테리시스 & 디바운스 적용

- 연속 판정(예: N 연속 샘플이 동일 상태일 때 반영)으로 깜박임/스파이크 방지.
- ON/OFF 임계값을 별도로 두어 hysteresis 적용.

3.4 동기화

- 동시 샘플 필요 시 **타임스탬프 동기화** 또는 같은 소프트웨어 트리거 순서로 읽기.
- 연속 모드 센서는 읽기 직전 최신값을 보장하도록 모드 확인.

---

4. I²C 레벨 해결책 (주소 충돌 / 동일 주소 장치 문제)

4.1 주소 핀 변경 가능한 경우

- 장치의 A0/A1 핀으로 주소를 변경하여 충돌 회피. (하드웨어·회로 변경 필요)

4.2 I²C 멀티플렉서 사용 (권장: 동일 주소 다수 지원)

- 예: **TCA9548A** (8채널 I²C switch) — 채널 선택 레지스터로 특정 채널의 장치만 버스에 연결.
- 장점: 동일 주소 장치 다수 사용 가능, 펌웨어에서 채널 전환만으로 접근.
- 단점: 멀티플렉서 선택 지연(레지스터 쓰기 후 안정화 시간 필요).

TCA9548A 사용 예 (HAL):

```
#define TCA_ADDR (0x70 << 1)  // 7-bit 0x70
// 채널 0 선택
uint8_t ch = (1 << 0);
HAL_I2C_Master_Transmit(&hi2c1, TCA_ADDR, &ch, 1, 100);
```

4.3 소프트웨어 주소 재맵핑(부팅 시 시퀀스)

- 일부 장치는 XSHUT, RESET 라인으로 개별 리셋 후 순차적으로 주소를 재할당 가능(예: VL53L0X 다중 사용 방법).

---

5. 구현 예제 — STM32 HAL 코드 패턴

5.1 단순 병합(두 주소 평균) — 동기 폴링

```
#define ADDR1 (0x48 << 1)
#define ADDR2 (0x49 << 1)

bool read_sensor(uint8_t dev, uint16_t reg, uint16_t *out) {
    uint8_t buf[2];
    if (HAL_I2C_Mem_Read(&hi2c1, dev, reg, I2C_MEMADD_SIZE_8BIT, buf, 2, 50) != HAL_OK)
        return false;
    *out = (buf[0] << 8) | buf[1];
    return true;
}

bool read_and_merge(uint16_t *merged_mm) {
    uint16_t v1, v2;
    bool ok1 = read_sensor(ADDR1, 0x01, &v1);
    bool ok2 = read_sensor(ADDR2, 0x01, &v2);

    if (ok1 && ok2) {
        *merged_mm = (v1 + v2) / 2; // 단순 평균
        return true;
    } else if (ok1) {
        *merged_mm = v1;
        return true;
    } else if (ok2) {
```

```
        *merged_mm = v2;
        return true;
    } else {
        return false;
    }
}
```

5.2 TCA9548A 멀티플렉서로 동일 주소 장치 접근

```
#define MUX_ADDR (0x70 << 1)
void mux_select(uint8_t channel) {
    uint8_t cmd = 1 << channel;
    HAL_I2C_Master_Transmit(&hi2c1, MUX_ADDR, &cmd, 1, 100);
}

// 사용 예
mux_select(0); // channel 0에 연결된 장치 접근
HAL_Delay(1);  // 안정화
HAL_I2C_Mem_Read(&hi2c1, (0x50<<1), reg, I2C_MEMADD_SIZE_8BIT, buf, len, 100);
```

---

6. 동시성 / RTOS 고려사항

- **버스 상호배제(Mutex)**: FreeRTOS 환경에서는 I²C 버스와 멀티플렉서 사용 시 반드시 Mutex로 보호.
- **채널 전환 시 안정화 대기**: 멀티플렉서 레지스터 쓰기 후 1~2ms 대기 권장.
- **비차단 방식**: 긴 I²C 타임아웃은 Task 지연 초래 → 비동기(인터럽트/DMA) 또는 별도 I²C Task 권장.

---

7. 오류 대응 및 복구

7.1 타임아웃 / NACK 발생

- 재시도 로직(재시도 횟수 1~3회, 지수 백오프) 후 복구 시도.
- 연속 실패 시 해당 장치 비활성 플래그 설정 및 알람.

7.2 버스 정지(Bus Hang)

- SCL/SDA 강제 토글(9클럭) + Stop 생성(버스 복구 루틴) 수행.
- 필요 시 I²C 주변장치 리-이니셜라이즈(HAL_I2C_DeInit / HAL_I2C_Init).

7.3 데이터 불일치(두 센서 값 큰 차이)

- Outlier 탐지(예: |v1-v2| > threshold) → 신뢰도 기반 선택 또는 재측정.
- 필터(칼만/가중평균) 적용.

---

8. 성능/타이밍 주의사항

- I²C 속도(100/400kHz)와 장치 응답 시간을 고려해 읽기 간격 조정.
- 멀티플렉서 채널 전환 O(1~ms) 오버헤드 존재 — 고속 다중 읽기엔 영향.
- 전원/접지 공유 시 간섭 주의 — 센서 다수 연결 시 전력 설계 중요.

---

9. 권장 설계 흐름(요약)

1. 요구사항 정의: 중복 신뢰성? 해상도 증강? 동일 주소 문제 해결?
2. 하드웨어 선택: 주소핀/멀티플렉서/전원/레벨시프터 결정.
3. 통신 패턴 설계: 동기화·타임아웃·재시도·Mutex 규칙 정의.
4. 병합 알고리즘 선정: 평균/가중/median/kalman 등.
5. 오류복구, 복구 우선순위 및 알람 설계.
6. 현장 튜닝: threshold·가중치·필터 파라미터 보정.

---

10. 결론

Dual I²C Address 병합은 단순히 두 값을 더하거나 평균 내는 일을 넘어서, **유효성 검사·타이밍·동기화·오류대응·하드웨어 제약**을 함께 고려해야 신뢰성 있는 시스템을 만든다. 동일 주소 장치 문제는 멀티플렉서(TCA9548A) 사용이 실무적으로 가장 직관적이며 안정적이다. 센서 융합(merge)은 응용 목적(정확도 vs 응답성)에 따라 적절한 필터/가중치 전략을 선택하면 된다.

- # Threshold 기반 마지막 감지단 탐색

정전식 수위센서는 여러 개의 **측정 세그먼트(감지단, electrode)** 로 구성되어 있으며, 각 세그먼트는 물 접촉 시 **정전용량(capacitance)** 값이 증가한다.
이 값은 I²C 인터페이스를 통해 MCU로 전달되며, 일반적으로 8~20단의 채널이 순차적으로 읽힌다.

센서의 각 채널은 물의 존재 여부를 기준으로 “ON(1)” 또는 “OFF(0)”로 표현된다.
“Threshold 기반 마지막 감지단 탐색”은 다음의 과정을 통해 **수면 높이(level height)** 를 계산하는 핵심 절차이다.

---

2. 기본 원리

1. **채널별 정전용량 값(또는 ADC 값)** 을 읽는다.
2. **기준 임계값(Threshold)** 을 설정한다.
   - 예: 물 접촉 시 정전용량 ≥ 2000 → 감지(ON)
   - 공기 상태 시 정전용량 < 2000 → 비감지(OFF)
3. 모든 채널을 **하단 → 상단 방향**으로 스캔하며 마지막으로 감지된(ON) 세그먼트의 인덱스를 찾는다.
4. 해당 인덱스에 **세그먼트 간 간격(distance per segment)** 을 곱해 수위를 산출한다.

$$h = N_{last} \times d_{seg} + c_{offset}$$

여기서

- N_{last}: 마지막 감지단 인덱스 (0~19)

- d_{seg}: 각 세그먼트 높이 간격 (예: 5mm)
- c_{offset}: 보정 오프셋 (예: 3mm)

---

3. 데이터 처리 절차

3.1 데이터 수집

센서가 20채널 값을 I²C 메모리 공간에 저장한다고 가정한다.
MCU는 다음과 같이 모든 채널을 읽는다.

```
uint16_t sensor_raw[20];
HAL_I2C_Mem_Read(&hi2c2, SENSOR_ADDR, 0x00, I2C_MEMADD_SIZE_8BIT,
                 (uint8_t*)sensor_raw, sizeof(sensor_raw), 100);
```

3.2 Threshold 적용

```
#define THRESHOLD 2000
uint8_t state[20];

for (int i = 0; i < 20; i++) {
    state[i] = (sensor_raw[i] >= THRESHOLD) ? 1 : 0;
}
```

3.3 마지막 감지단 탐색

```
int last_on = -1;
for (int i = 0; i < 20; i++) {
    if (state[i] == 1)
        last_on = i;  // 감지된 가장 높은 index를 저장
}
```

3.4 수위 계산

```
float height_mm = last_on * 5.0f + 3.0f;  // 5mm 간격, 3mm 오프셋
```

---

4. 예시 출력

채널	값(raw)	상태(ON/OFF)
0	2531	ON
1	2602	ON
2	2478	ON
3	2184	ON
4	1902	OFF

채널	값(raw)	상태(ON/OFF)
…	…	…

→ 마지막 감지단 `i = 3`
→ 수위 계산: $h = 3 \times 5 + 3 = 18\text{mm}$

---

5. 안정화 기법

5.1 히스테리시스(Hysteresis)

센서의 잡음으로 임계값 근처가 진동할 수 있으므로, ON/OFF 전환 시 상하한값을 분리한다.

```
#define TH_ON  2100
#define TH_OFF 1900
```

이 방식은 수면 경계 근처의 불안정한 감지를 방지한다.

5.2 이동평균 필터

각 채널의 최근 N회 데이터를 평균하여 안정성을 향상시킨다.

```
filtered[i] = (old[i]*3 + new[i]) / 4;
```

5.3 중복 판정

마지막 감지단이 일정 시간 동안 동일하면 수위가 안정된 것으로 판단한다.
(예: 3회 연속 동일 인덱스 유지 시 확정)

---

6. 예외 처리

상황	원인	대응
모든 채널 OFF	수위 0 또는 센서 분리	경고 표시 및 재시도
모든 채널 ON	수위 초과 또는 센서 오염	상한 알람
중간 채널만 ON	노이즈 또는 공기포켓	평균/필터링으로 보정
연속 불량 데이터	I²C 통신 문제	버스 복구 및 재시도

---

7. FreeRTOS 환경에서의 구현 예

센서 읽기와 연산을 주기적으로 수행하는 `WaterTask` 구성:

```
void WaterTask(void *argument)
{
    for (;;) {
        Read_WaterSensor(sensor_raw);
        Process_WaterLevel(sensor_raw);
        osDelay(1000); // 1초 주기 측정
    }
}
```

---

8. 결론

Threshold 기반 마지막 감지단 탐색은 **단순하면서도 신뢰도 높은 수위 산출 방식**이다.
센서 개별 오차를 보정하고 히스테리시스·필터링을 적용하면
수 mm 단위의 수위 안정도를 확보할 수 있다.

이 기법은 후속 단계인 **다중 센서 융합(초음파 + 정전식)** 의 기초 연산에도 사용된다.

- # 수위(mm) 계산식: `height = last_on * 5 + 3`

1. 개요

정전식 수위센서의 각 감지단은 일정한 높이 간격으로 배열되어 있으며,
센서의 출력값을 기준으로 마지막으로 활성화된(ON) 감지단의 위치를 찾아 수위를 계산한다.

이때 계산식은 다음과 같은 선형 모델로 표현된다.

$$h = N_{last} \times d_{seg} + c_{offset}$$

여기서,

- h: 실제 수위(mm)
- N_{last}: 마지막으로 감지된 세그먼트 인덱스 (0부터 시작)
- d_{seg}: 세그먼트 간 거리(mm)
- c_{offset}: 센서 하단 오프셋(mm)

---

2. 상수 정의

센서 구조를 기준으로 다음과 같이 정의할 수 있다.

변수	의미	예시값
`d_seg`	각 감지단 간 높이 간격	5.0 mm
`c_offset`	하단 보정 오프셋	3.0 mm
`N_last`	마지막 감지된 채널 인덱스	3

---

3. 계산 예시

감지단 인덱스가 `3`인 경우:

$$h = 3 \times 5.0 + 3.0 = 18.0 \text{ mm}$$

즉, 네 번째 감지단(0~3)이 마지막으로 활성화되었다면
수면은 약 18mm 지점에 위치한다고 판단한다.

---

4. 오프셋(`c_offset`) 의미

센서의 하단부(첫 번째 전극)와 실제 탱크 바닥 사이에는
물리적 간격 또는 비감지 구간(dead zone)이 존재한다.
이 오프셋은 다음 목적을 가진다.

- **기계적 장착 여유**: 전극 하단이 바닥보다 약간 위에 위치
- **비선형 감도 보정**: 하단부 감도 저하 구간 보완
- **보정 시험 결과 반영**: 실험적 보정값

오프셋은 경험적 보정으로 얻으며, 일반적으로 2~5 mm 범위에서 설정된다.

---

5. 확장식: 비선형 보정 적용

일부 센서는 전극 간 간격 또는 감도 분포가 일정하지 않으므로,
구간별 보정계수를 적용해 비선형 모델로 확장할 수 있다.

$$h = (a \times N_{last} + b \times N_{last}^2) + c_{offset}$$

- a: 기본 단위 높이(mm)
- b: 비선형 보정 계수(mm/단²)

이때 $b > 0$이면 상부로 갈수록 거리 간격이 약간 넓어진다.

---

6. 구현 예시 (STM32 HAL)

```
#define SEGMENT_GAP_MM   5.0f
#define OFFSET_MM        3.0f

float Get_WaterLevel(uint8_t last_on)
{
    return (last_on * SEGMENT_GAP_MM) + OFFSET_MM;
}
```

혹은 비선형 보정이 필요한 경우:

```c
#define COEFF_A 5.0f
#define COEFF_B 0.02f
#define OFFSET_MM 3.0f

float Get_WaterLevel_Nonlinear(uint8_t last_on)
{
    return (COEFF_A * last_on + COEFF_B * last_on * last_on) + OFFSET_MM;
}
```

---

7. 단위 변환

필요 시, 수위 단위를 cm로 변환하여 표시할 수 있다.

$$h_{cm} = \frac{h_{mm}}{10.0}$$

```c
float height_mm = Get_WaterLevel(last_on);
float height_cm = height_mm / 10.0f;
printf("Water Level = %.2f cm\n", height_cm);
```

---

8. 결론

수위 계산식 `height = last_on * 5 + 3` 은
정전식 다단 감지센서의 단순·선형 모델로,

- **물리적 간격(세그먼트 간 거리)**
- **기계적 오프셋(센서 하단 보정)**

을 고려한 실제 수위 추정에 사용된다.
필요 시 온도, 감도, 매질 변화에 따른 보정 계수를 추가함으로써
정확도를 ±1mm 수준까지 향상시킬 수 있다.

6.5 센서 결합 테스트

- # 초음파 + 수위 + 무게 통합

목적: HC-SR04(초음파) + 20단 정전식 수위센서 + HX711(로드셀) 데이터를 결합하여 **신뢰성 높은 수위 및 질량(무게) 상태**를 제공하는 임베디드 통합 아키텍처와 구현 지침을 제시한다.
문체는 기술문서 형식이며, 존댓말 사용하지 않음.

---

1 설계 개요

통합 시스템은 다음 목표를 가진다.

- 각 센서의 강점을 살리고 약점을 보완하여 신뢰도 높은 **수위(mm)**와 **무게(g/kg)** 산출
- 실시간성 확보(주기적 샘플링) + 오류(타임아웃, NACK, 이상치) 견고성
- FreeRTOS 기반이면 태스크 분리, 아니면 타이머 기반 폴링 구조로 구현
- I²C와 GPIO/Timer 리소스의 동시 접근 보호(뮤텍스/임계영역)

구성요소

- HC-SR04: 거리(mm) → 보조 수위(원거리나 장애물 보정용)
- 정전식 20단: 단계별 ON/OFF → 최종 수위 지점(last_on) → 기본 수위 계산
- HX711: 로드셀 무게 → 밀도 추정 및 수위 보정(필요 시)

데이터 융합 목적: 서로 다른 물리적 양(거리/단계/무게)을 이용해 **정합성 검사**, **보정/추정**, **결측 보완**을 수행.

---

2 데이터 플로우 요약

1. 각 센서 동기 또는 반동기(비동기) 샘플링
2. 전처리: 필터(이동평균), 임계값/Hysteresis 적용, 타임아웃 처리
3. 품질지표 산출: `q_ultrasonic`, `q_capacitive`, `q_loadcell` (0..1)
4. 융합 알고리즘 적용
    - 정상 컨디션: 가중평균(품질가중)
    - 불일치(> threshold): 이상치 제거 또는 재시도
    - 결측: 다른 센서로 보정(예: 정전식 실패 시 초음파로 대체)
5. 출력: `WaterLevel_mm`, `Mass_g`, `MassValidity`, `LevelValidity`, `FusionQuality`

---

3 센서별 처리 권장사항

3.1 HC-SR04 (초음파)

- TRIG: 10μs 펄스, ECHO는 Timer Input Capture로 측정
- 타임아웃: 40 ms 권장 (MAX_TIMEOUT)
- 샘플 주기: 200–500 ms (실시간 모니터링: 100 ms 가능하나 반향·노이즈 고려)
- 전처리: 이동평균(N=3~5), 이상치 제거(값=0 또는 >max_range)
- 품질지표 `q_ultrasonic`:
    - 1.0 : 반복 샘플 표준편차 < small_thresh && 값 within expected_range
    - 0.5 : 표준편차 moderate
    - 0.0 : 타임아웃 / 0 또는 불합리 값

3.2 정전식 20단

- 각 채널은 baseline 보정 후 threshold + hysteresis로 ON/OFF 판정
- 필터: 채널별 이동평균(샘플 N=4~10) + 디바운스(연속 N 판정)
- last_on 얻은 뒤 기본 수위: `level_mm = last_on * step_mm + offset_mm`
- 샘플 주기: 200–1000 ms (노이즈·디바운스에 따라)
- 품질지표 `q_capacitive`:
    - 1.0 : 연속 판정 안정(3회 동일)
    - 0.5 : 변화 중(상승/하강 직전)

- 0.0 : 모든 채널 OFF/ON(비정상) 또는 통신 오류

3.3 HX711 (로드셀)

- 읽기: HX711_ReadRaw(), 부호 확장 처리, 평균 샘플링(예: 5~10회)
- 보정: Offset, Scale은 EEPROM/Flash에 저장
- 필터: 이동평균 또는 중간값 필터(중간값은 이상치에 강함)
- 샘플 주기: 500 ms ~ 2 s (로드셀 변동이 천천히 일어남)
- 품질지표 `q_loadcell` :
  - 1.0 : 표준편차 작음(안정)
  - 0.5 : 변동 있음(흔들림)
  - 0.0 : 읽기 실패/포화/오류

---

4 품질(신뢰도) 기반 융합 알고리즘

목표: 센서 신뢰도에 기반하여 최종 수위를 산출. 기본 전략은 **가중평균 + 이상치 필터링 + 백업/대체 규칙**.

정의:

- `L_c` = level from capacitive (mm)
- `L_u` = level from ultrasonic (mm) (거리→수위 변환 필요)
- `M` = mass from HX711 (g)

단계:

1. **유효성 판단**
   - `valid_c = (q_capacitive > 0)`
   - `valid_u = (q_ultrasonic > 0)`
   - `valid_m = (q_loadcell > 0)`
2. **거리→수위 변환 (초음파)**
   - 초음파가 탱크 상단에서 아래로 재는 경우 센서 설치 방식에 맞게 변환:
     `L_u = sensor_to_liquid_distance_mm` → 수위 = `tank_total_height_mm - L_u - sensor_mount_offset`
3. **가중평균 (품질가중)**
   - 가중치: `w_c = q_capacitive`, `w_u = q_ultrasonic`, `w_m = f_mass(M)` (mass 신뢰도는 수위 예측력에 따라)
   - `f_mass(M)` 는 물의 밀도 가정(예: 밀도 1 g/cm³)과 탱크 단면적(A)으로부터 예상 수위를 역산할 때 사용될 수 있음:
     - 예상 수위 from mass: `L_m = (M / (A * rho)) * 1000` (mm)
     - mass 가중치는 mass 안정성 기반: `w_m = q_loadcell * gain_m` (gain_m ≤ 1)
   - 융합:

```
W = w_c + w_u + w_m
If W == 0 -> Fusion invalid
L_fused = (w_c*L_c + w_u*L_u + w_m*L_m) / W
FusionQuality = W / 3.0  // 정규화
```

4. **이상치/불일치 처리**
   - 만약 |L_c - L_u| > DELTA_THRESHOLD (예: 50 mm) 이고 한 쪽 품질 낮으면 낮은 쪽 버림
   - 모든 센서 불일치 시: 우선순위 로직 예: capacitive > ultrasonic > mass (기본) 또는 선택적(구조에 따라)
5. **결측 보정(대체)**
   - capacitive 결측 → ultrasonic & mass로 보정
   - ultrasonic 결측 → capacitive 우선
   - mass 결측 → 수위만 제공(무게 미포함)

---

5 시스템 타이밍 권장치

- 정전식(20단): 200–500 ms 주기
- 초음파: 200–500 ms 주기(정전식과 동기화 권장)
- HX711: 500 ms ~ 2 s (비동기, 융합 시 최신 값 사용)
- Fusion 주기: 정전식/초음파 중 빠른 쪽 주기(예: 200–500 ms)로 동작, mass는 별도로 업데이트 시 반영

FreeRTOS 권장:

- `SensorTask` (200 ms): 정전식 + 초음파 샘플링 → 전처리 → 큐에 push
- `WeightTask` (500~1000 ms): HX711 읽기 → 필터 → 큐에 push
- `FusionTask` (200 ms): 가장 최근 센서 데이터 POP → 융합 → publish

---

6 구현 예제 (의사 코드 — FreeRTOS + STM32 HAL)

> 핵심 포인트: I²C 버스 공유 시 `i2c_mutex` 사용, HX711은 GPIO 시퀀스(핀 제어), 초음파는 TIM IC 사용.

```
// 공유 리소스
SemaphoreHandle_t i2c_mutex;

typedef struct {
    float level_cap_mm;
    float level_ultra_mm;
    float mass_g;
    float q_cap, q_ultra, q_mass;
    uint32_t timestamp;
} SensorSnapshot_t;

QueueHandle_t sensor_queue; // recent snapshot queue

// SensorTask: capacitive + ultrasonic
void SensorTask(void *arg) {
    SensorSnapshot_t snap;
    for(;;) {
```

```
        // Read capacitive (I2C)
        if (xSemaphoreTake(i2c_mutex, pdMS_TO_TICKS(50)) == pdTRUE) {
            bool ok = Read_Capacitive_All(ch_vals); // user function
            xSemaphoreGive(i2c_mutex);
            if (ok) {
                Process_Capacitive(ch_vals, &snap.level_cap_mm, &snap.q_cap);
            } else {
                snap.q_cap = 0;
            }
        }

        // Trigger ultrasonic and read (Timer IC + callback sets ic_difference)
        Trigger_Ultrasonic();
        if (WaitForUltrasonicResult(&snap.level_ultra_mm, 60 /*ms*/)) {
            snap.q_ultra = EvaluateUltraQuality(...);
        } else {
            snap.q_ultra = 0;
        }

        snap.timestamp = HAL_GetTick();
        xQueueOverwrite(sensor_queue, &snap); // keep last snapshot

        vTaskDelay(pdMS_TO_TICKS(200));
    }
}

// WeightTask: HX711
void WeightTask(void *arg) {
    for(;;) {
        float mass = HX711_Get_Value_Averaged(5, &ok);
        SensorSnapshot_t snap;
        if (xQueuePeek(sensor_queue, &snap, 0) == pdTRUE) {
            snap.mass_g = ok ? mass : 0;
            snap.q_mass = ok ? EvaluateMassQuality(...) : 0;
            snap.timestamp = HAL_GetTick();
            xQueueOverwrite(sensor_queue, &snap);
        }
        vTaskDelay(pdMS_TO_TICKS(800));
    }
}

// FusionTask
void FusionTask(void *arg) {
    SensorSnapshot_t snap;
    for(;;) {
        if (xQueuePeek(sensor_queue, &snap, portMAX_DELAY) == pdTRUE) {
            float Lm = MassToLevel(snap.mass_g); // use tank area & rho
            float w_c = snap.q_cap;
            float w_u = snap.q_ultra;
            float w_m = snap.q_mass * 0.5f; // mass weight scaling
            float W = w_c + w_u + w_m;
            if (W < 0.01f) {
                PublishInvalid();
```

```
            } else {
                float fused = (w_c*snap.level_cap_mm + w_u*snap.level_ultra_mm + w_m*Lm)/W;
                PublishLevelMass(fused, snap.mass_g, W/3.0f);
            }
        }
        vTaskDelay(pdMS_TO_TICKS(200));
    }
}
```

---

7 데이터 구조 및 저장 (Calibration / 상태)

- EEPROM 구조(예시):

```
typedef struct {
    uint32_t magic;
    float cap_baseline[20];
    float cap_thresholds[20];
    float hx_offset;
    float hx_scale;
    float tank_area_mm2;
    float level_offset_mm;
} CalibStore_t;
```

- 저장/불러오기: 부팅 시 `LoadCalibration()` 호출. 보정값 변경 시 `SaveCalibration()` 호출.

---

8 오류 처리 및 복구 정책

- I²C 실패: 3회 재시도 → 버스 복구 루틴(9클럭 토글) → 재초기화 → 알람
- HX711 DOUT 미하강(읽기 불가): 타임아웃 → 재전원(옵션) 또는 에러 플래그
- 초음파 타임아웃: 값 무시 → last valid 사용 → 알람
- 융합 불가능(W==0): `level = last_valid_level` 또는 `NaN`으로 인터페이스에 통보
- Watchdog: 장시간 Task 블록/데드락 시 시스템 리셋

---

9 로깅, UI, 알람

- 출력: `{timestamp, level_mm, mass_g, fusion_quality, sensor_status_bits}` JSON 라인 로그
- 알람 레벨: INFO/WARNING/ERROR (예: ERROR: all sensors invalid)
- OLED/UART: 주기적 상태 출력(예: FusionTask가 1s 간격으로 요약 출력)

---

10 튜닝 가이드 / 실험 프로토콜

1. 각 센서 개별 안정화/보정 수행:
    - 정전식: 빈탱크 baseline 측정, threshold 설정
    - HX711: offset & scale (tare + known weight)
    - 초음파: 센서-수면 거리 baseline 및 offset 측정
2. 실험: 정량적 테스트(다양한 수위에서 mass 측정 → 예측수위와 비교)
3. 파라미터 튜닝: 이동평균 크기, 디바운스 갯수, DELTA_THRESHOLD(센서 불일치 허용치), mass weight scaling
4. 극단 테스트: 파도/거품/물흐름 환경에서 안정성 확인

---

11 예시 수치(참고)

- tank_area = 0.1 m² (100,000 mm²) → mass(kg) → level(mm): `L_m = (mass_g / 1000) / 0.1 * 1000 = mass_g / 0.1 = mass_g * 10` (이 값은 실제 탱크 단면적으로 조정 필요)
- DELTA_THRESHOLD for sensor disagreement: 50–100 mm (작은 탱크면 줄여라)

---

12 결론 및 권장 우선순위

1. 먼저 각 센서의 개별 보정(Offset/Scale/Threshold)을 완성하라.
2. I²C 버스 접근은 뮤텍스(FreeRTOS)로 보호. 멀티슬레이브 있으면 주소 관리/멀티플렉서 고려.
3. 융합은 단순 품질가중 평균으로 시작하고, 필요 시 칼만필터로 확장.
4. 오류/타임아웃 정책을 엄격히 하여 시스템 신뢰성 확보.
5. 튜닝은 현장 실험 기반으로 진행.

I²C 충돌 방지 및 Multi-Slave 테스트

I²C 버스는 다중 슬레이브(Multi-Slave) 구조를 지원하지만, 실제 하드웨어 환경에서는 전기적 간섭·펌웨어 타이밍 문제로 충돌이 자주 발생한다. 충돌 방지를 위해 하드웨어, 소프트웨어, 프로토콜 수준에서의 보호 전략이 필요하다.

---

1. 기본 개념

I²C는 **멀티마스터·멀티슬레이브** 구조를 지원하지만, 일반적으로 STM32는 **싱글마스터·멀티슬레이브**로 동작한다.
마스터는 Start/Stop/ACK 프레임을 완전히 제어하며, 모든 슬레이브는 SCL, SDA 라인을 공유한다.
따라서 하나의 버스에서 여러 장치가 존재하더라도, 주소 충돌(Address Conflict)이나 통신 중 SCL 라인 홀드가 발생하지 않도록 관리해야 한다.

---

2. 하드웨어 충돌 방지

1. 풀업 저항(Rpull-up) 정합

- SDA, SCL 라인은 Open-Drain 구조이므로 풀업 저항이 반드시 필요하다.
- 권장값: 3.3V 시스템 기준 4.7 kΩ ~ 10 kΩ
- 다수의 I²C 디바이스가 풀업 내장 시 병렬 효과로 저항값이 과도하게 낮아질 수 있음 → 실제 합성 저항 2 kΩ 이하로 떨어지면 파형 왜곡, 홀드 발생.
- 해결: 외부 풀업 저항을 4.7 kΩ 하나만 남기거나 내장 풀업 비활성화.

2. 배선 길이 및 커패시턴스

- 총 커패시턴스(Cbus)는 400 pF 이하 유지 권장.
- 장거리(>20 cm) 또는 병렬 센서 많을 경우 신호 상승시간 느려짐 → 타이밍 오류 발생.
- 필요 시 I²C 버퍼(IC, 예: PCA9515A) 또는 분리된 버스 사용.

3. 주소 충돌 방지

- 각 슬레이브는 고유 7-bit 주소를 가져야 함.
- 동일 IC 여러 개 사용 시 Address 핀(A0, A1, A2)로 주소 변경 가능.
- 예: ADS1115(0x48~0x4B), SHT31(0x44/0x45), VL53L0X(0x29 → 변경 가능)
- 주소 변경 불가 장치는 I²C 멀티플렉서(TCA9548A)로 분리.

4. 노이즈 차단

- SDA, SCL 라인에 100 Ω 직렬저항 + 100 pF 필터를 권장.
- 전원 노이즈 시 페라이트 비드(Ferrite Bead) 추가 고려.

---

3. 소프트웨어 충돌 방지

1. 버스 점유 보호 (뮤텍스)

- FreeRTOS 환경에서는 여러 Task가 I²C 함수를 동시에 호출할 수 있다.
- 반드시 `xSemaphoreTake(i2c_mutex, timeout)`으로 진입 보호 후 사용.
- 예시:

```
if (xSemaphoreTake(i2c_mutex, pdMS_TO_TICKS(50)) == pdTRUE) {
    HAL_I2C_Mem_Read(&hi2c1, addr<<1, reg, I2C_MEMADD_SIZE_8BIT, data, len, 100);
    xSemaphoreGive(i2c_mutex);
}
```

- HAL 내부에서는 re-entrancy 보장되지 않음.

2. 타임아웃 처리

- HAL 함수의 마지막 인자는 timeout(ms).
- 슬레이브 응답 없음 → `HAL_TIMEOUT` 리턴 시 버스 정지 가능.
- 즉시 복구 루틴 호출 필요(아래 참고).

3. 버스 정지 복구

- 슬레이브가 SCL을 계속 LOW로 유지할 때 마스터는 버스 접근 불가.
- 복구 절차:
  1. SCL 핀을 GPIO로 임시 재설정
  2. SCL을 수동으로 9회 토글
  3. SDA가 HIGH로 복귀하면 STOP 조건을 재발행
  4. I²C 재초기화 (`HAL_I2C_DeInit() → HAL_I2C_Init()`)

```
void I2C_Bus_Recovery(GPIO_TypeDef* scl_port, uint16_t scl_pin,
GPIO_TypeDef* sda_port, uint16_t sda_pin) {
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_OD;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    GPIO_InitStruct.Pin = scl_pin;
    HAL_GPIO_Init(scl_port, &GPIO_InitStruct);

    for (int i = 0; i < 9; i++) {
        HAL_GPIO_WritePin(scl_port, scl_pin, GPIO_PIN_SET);
        HAL_Delay(1);
        HAL_GPIO_WritePin(scl_port, scl_pin, GPIO_PIN_RESET);
        HAL_Delay(1);
    }

    HAL_GPIO_WritePin(scl_port, scl_pin, GPIO_PIN_SET);
}
```

4. 전송 실패 후 복구 정책

- `HAL_I2C_Mem_Read()`나 `HAL_I2C_Master_Transmit()`이 `HAL_ERROR` 반환 시:
  - 1회 버스 복구 루틴 호출
  - 이후 `HAL_I2C_DeInit()` + `HAL_I2C_Init()`으로 재설정
  - 3회 이상 연속 실패 시 시스템 리셋 또는 에러로그 저장.

---

4. Multi-Slave 테스트 시나리오

1. **목표**
   - 다수 슬레이브(예: OLED 0x3C, VL53L0X 0x29, Water Sensor 0x48/0x49) 병렬 연결 시 충돌 여부 및 응답 안정성 검증.
2. **테스트 단계**
   - (1) I²C Scan 수행 → 모든 주소 응답 확인
   - (2) 슬레이브별 순차 접근:

```
for (addr = 0x03; addr < 0x77; addr++) {
    if (HAL_I2C_IsDeviceReady(&hi2c1, addr<<1, 1, 100) == HAL_OK)
        printf("Found 0x%02X\r\n", addr);
}
```

   - (3) 주기적 읽기 Loop: 각 슬레이브에 교대로 접근하며 타임아웃 발생률 측정
   - (4) 부하 테스트: OLED 업데이트 + 센서 읽기 + EEPROM 접근을 동시에 실행 → 버스 충돌 감시
3. **검증 항목**
   - ACK/NACK 발생률
   - 통신 오류 횟수
   - HAL 리턴코드(`HAL_TIMEOUT`, `HAL_BUSY`, `HAL_ERROR`) 발생 빈도
   - 회복성(버스 복구 루틴 적용 후 정상 복귀 여부)
4. **테스트 자동화**
   - FreeRTOS 환경: TestTask 주기 500 ms, 각 슬레이브 순회
   - 오류 검출 시 로그 기록:

```
if (status != HAL_OK) {
    printf("[I2C ERR] addr=0x%02X code=%d\r\n", addr, status);
    I2C_Bus_Recovery(...);
}
```

---

5. 결론

- I²C는 물리적으로 단순하지만, 타이밍·풀업·동시 접근 문제로 충돌이 자주 발생한다.
- **핵심 방지 요소**는 다음과 같다.
  1. 풀업 정합 및 커패시턴스 관리
  2. 주소 중복 금지 및 필요 시 멀티플렉서 분리
  3. Task 간 접근 보호(뮤텍스)
  4. 타임아웃·복구 루틴 자동화
- 테스트는 슬레이브를 모두 연결한 실제 회로에서 반드시 수행해야 하며, 일정 시간(>10분) 이상 반복 통신을 통해 신뢰성 확인이 필요하다.