

10. 센서 데이터 처리 및 보정

10.1 보정 알고리즘

• Offset Calibration

1. 개요

Offset Calibration은 센서 출력 신호가 0 입력에서도 0이 아닌 값을 갖는 오차(Offset Error)를 보정하는 과정이다. 이는 특히 ADC 기반 측정 장치(예: HX711 로드셀, 아날로그 전압 센서)에서 필수적으로 수행되어야 하며, 하드웨어 오프셋이나 환경적 영향(온도, 회로 불균형 등)에 의해 발생한 기준점의 편차를 제거하는 역할을 한다. 결과적으로 보정된 데이터는 “측정 대상이 0일 때 출력도 0이 되도록” 정렬된다.

2. Offset 오차의 원인

Offset 오차는 다음 요인으로 발생한다.

- 센서 자체의 비이상적 균형
 - 로드셀 브리지 회로의 불균형
 - 영점(Zero Load) 시 전압 편차
- ADC 회로 기생 전압
 - 레퍼런스 노이즈, GND 전위 차이
- PCB 레이아웃 및 전원 잡음
 - HX711, ADS1115 등의 입력단 잡음 누적
- 환경적 요인
 - 온도 변화, 장기 Drift, 케이블 저항 등

3. 보정 원리

Offset 보정은 측정값에서 일정 기준(Offset 평균값)을 빼주는 간단한 선형 연산으로 이루어진다.

$$Value_{calibrated} = Value_{raw} - Offset$$

여기서 **Offset**은 무부하 상태(즉, 입력이 0일 때)의 측정 평균값으로 정의된다.

4. 보정 절차

(1) 무부하 상태 설정

센서에 아무 부하도 걸리지 않은 “기준 상태”로 시스템을 둔다.

(2) 다중 샘플 측정

노이즈 영향을 최소화하기 위해 다수의 샘플을 취득한다.

예: 100회 평균

```
1 long ReadOffset(HX711 *hx)
2 {
3     long sum = 0;
4     for (int i = 0; i < 100; i++)
5     {
6         sum += HX711_ReadRaw(hx);
7         HAL_Delay(10);
8     }
9     return sum / 100;
10 }
```

(3) 평균값 저장

계산된 Offset 평균값을 EEPROM 또는 Flash에 저장한다.

```
1 EEPROM_WriteLong(OFFSET_ADDR, offset_value);
```

(4) 측정 시 적용

이후 실제 측정 시,
읽은 원시값에서 Offset을 빼준다.

```
1 float value = HX711_ReadRaw(hx) - offset_value;
```

5. 보정 결과 검증

Offset Calibration이 완료되면, 무부하 상태에서 측정된 결과는 0 근처의 안정된 값을 가져야 한다.

오차가 ± 5 단위 이하로 유지된다면 정상이다.

예시 결과:

측정 상태	보정 전 (Raw)	보정 후 (Calibrated)
무부하	-8435	0
500g	26530	34965
1000g	54590	63025

6. 주기적 재보정

Offset Drift는 장기 사용 시 필연적으로 발생한다.
따라서 시스템 초기화 시점 또는 일정 주기(예: 24시간, 전원 재투입 시)마다
자동 Offset 재보정 루틴을 실행하는 것이 바람직하다.

자동 보정 루틴 예시:

```
1  if (abs(HX711_GetValue() - last_offset) > 1000)
2  {
3      offset_value = ReadOffset(&hx);
4  }
```

7. 주의사항

항목	설명
안정 대기시간	전원 인가 후 1~2초간 회로 안정화 필요
온도 영향	고정밀 시스템은 온도 보상 병행 필요
EEPROM 쓰기 주기 제한	과도한 저장은 수명 저하 유발 → 변경 시에만 저장

8. 결론

Offset Calibration은 측정 시스템의 **영점 기준 안정성 확보**를 위한 첫 단계이다.
이를 통해 모든 후속 연산(스케일링, 필터링, 제어)이 **정확한 기준점** 위에서 수행되며,
장기적인 데이터 일관성과 신뢰도를 크게 향상시킨다.

• Scale Calibration

• EEPROM에 Calibration 데이터 저장

10.2 필터링

• Moving Average

1. 개요

Scale Calibration은 센서의 출력값을 실제 물리적 단위(예: g, kg, mm, V 등)로 변환하기 위해
스케일링 계수(Scale Factor)를 결정하는 과정이다.
이는 **Offset Calibration** 후 수행되는 2차 보정 절차로,
센서의 감도(Sensitivity)를 기준으로 원시 ADC 값 → 물리값 간의 변환 비율을 정립한다.

대표적으로 로드셀(HX711)이나 아날로그 압력 센서에서는
1 ADC 단위가 몇 g(또는 N)에 해당하는지를 산출하여
실제 무게를 계산한다.

2. 보정 원리

센서 출력의 관계식은 선형으로 가정할 수 있다.

$$\text{Weight (g)} = (\text{ADC}_{\text{raw}} - \text{Offset}) \times \text{Scale}$$

여기서,

- **Offset:** 영점 보정된 기준값 (무부하 시 출력값)
- **Scale:** ADC 한 단위당 무게 변화량 (g/ADC count)

즉, Scale은 다음과 같이 구한다.

$$\text{Scale} = \frac{\text{Known Weight (g)}}{\text{ADC}_{\text{measured}} - \text{Offset}}$$

3. 보정 절차

(1) Offset Calibration 수행

무부하 상태에서 평균값을 구하고 Offset으로 저장한다.

```
1 | long offset = ReadOffset(&hx);
```

(2) 기준 하중(표준추) 올리기

정확히 알려진 무게(예: 1000 g)의 표준추를 로드셀 위에 올린다.

(3) 보정용 원시 데이터 취득

여러 번 측정하여 평균값을 계산한다.

```
1 | long raw = 0;  
2 | for (int i = 0; i < 50; i++)  
3 | {  
4 |     raw += HX711_ReadRaw(&hx);  
5 |     HAL_Delay(20);  
6 | }  
7 | raw /= 50;
```

(4) Scale Factor 계산

```
1 | float scale = (float)known_weight / (raw - offset);
```

(5) Scale Factor 저장

EEPROM이나 Flash에 저장하여 다음 부팅 시 자동 복원되도록 한다.

```
1 | EEPROM_writeFloat(SCALE_ADDR, scale);
```

(6) 측정 변환

이후 모든 측정값에 대해 다음과 같이 변환한다.

```
1 | float weight = (HX711_ReadRaw(&hx) - offset) * scale;
```

4. 예시 데이터

구분	무부하 Raw	1000g Raw	Offset	Scale(g/count)	1000g 환산 값
측정 값	-8500	26500	-8500	$1000 / (26500 - (-8500)) = 0.0286$	$\approx 1000.4g$

5. 다중 구간 보정 (Multi-Point Calibration)

센서가 완전히 선형적이지 않은 경우,

여러 기준점(예: 0g, 500g, 1000g, 2000g)을 사용하여 구간별 Scale을 계산할 수 있다.

$$\text{Scale}(i) = \frac{W_{i+1} - W_i}{ADC_{i+1} - ADC_i}$$

실시간 측정 시, ADC가 속한 구간에 따라 보간(Interpolation)하여
보다 정확한 결과를 얻는다.

6. 검증

보정 후, 여러 표준추를 측정하여 선형성을 검증한다.

표준 무게 (g)	측정값 (g)	오차 (%)
0	0.3	+0.03
500	499.8	-0.04
1000	1000.4	+0.04
2000	2001.5	+0.08

결과: 오차 $\pm 0.1\%$ 이내 → 정상 보정

7. 주의사항

항목	설명
온도 Drift	Scale은 온도 변화에 따라 변동 가능 → 온도 보정 필요
기계적 스트레스	센서 부착 방향 및 하중 중심이 바뀌면 재보정 필요
비선형 구간 회피	센서 사양의 Full Scale 80% 이하에서 사용 권장
EEPROM 저장 주기	자주 변경되는 Scale 값은 RAM에만 유지 후 확정 시 저장

8. 결론

Scale Calibration은 **센서의 물리적 단위 환산 정확도**를 결정짓는 핵심 단계이다.
정확히 수행된 Offset + Scale 보정을 통해
HX711 기반 무게 측정 시스템은 ±0.1% 이하의 정밀도로 안정된 계측을 수행할 수 있다.

Exponential Smoothing

1. 개요

Exponential Smoothing (지수 평활, 지수 이동 평균)은
최근 데이터를 더 큰 비중으로 반영하여 신호의 노이즈를 완화하는 필터링 기법이다.
센서 출력의 단기 변동(노이즈)을 억제하면서도
실제 변화(트렌드)는 빠르게 반응하도록 설계된다.
이는 **Moving Average**보다 메모리 부담이 적고,
실시간 임베디드 시스템에서 효율적인 필터링 방식으로 자주 사용된다.

2. 기본 수식

지수 평활의 핵심 수식은 다음과 같다.

$$S_t = \alpha \cdot X_t + (1 - \alpha) \cdot S_{t-1}$$

여기서,

- S_t : 시점 t에서의 평활된 값 (Filtered Value)
- X_t : 시점 t에서의 실제 입력값 (Raw Data)
- α : 평활 계수 ($0 < \alpha \leq 1$)
- S_{t-1} : 이전 시점의 평활 결과

3. 파라미터 α 의 의미

α 값	특성	반응 속도	노이즈 억제력
0.1 이하	완만한 평활	느림	매우 강함
0.3~0.6	일반적인 센서 필터	보통	적당함
0.8 이상	민감한 반응	빠름	약함

즉,

- $\alpha \rightarrow 1$ 이면 원시 데이터에 근접
- $\alpha \rightarrow 0$ 이면 이전 추세 유지(완만)

임베디드 센서(예: 초음파, 로드셀, 수위)는 보통 $\alpha = 0.2 \sim 0.4$ 정도가 적절하다.

4. 구현 예시 (C 코드)

```
1 float Exponential_Smoothing(float prev, float current, float alpha)
2 {
3     return alpha * current + (1.0f - alpha) * prev;
4 }
```

실제 사용 예:

```
1 float smoothed_value = 0.0f;
2 float alpha = 0.3f;
3
4 while (1)
5 {
6     float raw = HX711_Get_Value(); // 센서 원시 데이터
7     smoothed_value = Exponential_Smoothing(smoothed_value, raw, alpha);
8
9     printf("Filtered: %.2f\n", smoothed_value);
10    HAL_Delay(50);
11 }
```

5. 초기 조건

초기 시점 S_0 은 보통 첫 번째 측정값으로 설정한다.

$$S_0 = X_0$$

혹은 Moving Average로 초기 안정값을 계산한 뒤 S_0 으로 사용한다.

6. 예시 데이터

시점	원시 데이터 (X)	평활 결과 (S, $\alpha=0.3$)
t ₀	100	100
t ₁	108	102.4
t ₂	95	99.18
t ₃	102	99.82
t ₄	98	99.47
t ₅	105	101.33

노이즈가 큰 원시 데이터(X)가 매끄럽게 평활된 형태로 변환된다.

7. Double / Triple Smoothing

보다 정교한 추세 예측이 필요한 경우,
다단계 지수 평활법을 적용한다.

- **Double Exponential Smoothing**: 추세(Trend) 반영
- **Triple (Holt-Winters)**: 계절성(Seasonality) 반영

임베디드에서는 실시간 부하를 고려해 **단일 지수 평활(Single Smoothing)**만 주로 사용한다.

8. 장점과 한계

구분	설명
장점	메모리 절약 (이전 한 값만 필요), 계산 단순 (곱셈 2회)
노이즈 억제	센서의 순간적 이상값(Spike)을 효과적으로 완화
한계	급격한 변화에는 지연 발생, α 설정이 민감함

9. 실무 적용 예

(1) 무게 측정

HX711 출력에 적용

```
1 | weight_filtered = Exponential_Smoothing(weight_filtered, weight_raw, 0.25f);
```

(2) 거리 측정

VL53L0X / 초음파 센서 거리값 평활

```
1 | distance_filtered = Exponential_Smoothing(distance_filtered, distance_raw, 0.35f);
```

(3) 수위 변화 감시

센서 잡음을 억제하여 안정된 변화 감지

10. 결론

Exponential Smoothing은

센서 노이즈 제거, 신호 안정화, 리소스 효율성을 동시에 달성할 수 있는
임베디드 환경에서 매우 실용적인 필터링 기법이다.

적절한 α 설정과 초기화만으로,

복잡한 Kalman Filter 없이도 충분히 부드럽고 안정적인 데이터를 확보할 수 있다.

• Simple Kalman Filter (1차)

1. 개요

Kalman Filter (칼만 필터)는 확률적 상태 추정 알고리즘으로,

센서 측정값에 포함된 노이즈를 최소화하고

실제 상태(참값)에 대한 **최적 추정값**을 계산한다.

1차(Simple) 칼만 필터는 시스템 모델이 단일 상태 변수(예: 위치, 무게, 거리 등)로 구성된

가장 기본적인 형태이며, **임베디드 센서 신호 안정화**에 자주 사용된다.

지수평활(Exponential Smoothing)이 경험적 계수 기반이라면,

칼만 필터는 **통계적 근거(오차 공분산, 측정 불확실성)**를 이용해

보다 이론적으로 최적화된 추정을 수행한다.

2. 상태 모델

1차 칼만 필터는 다음의 시스템을 가정한다.

$$x_k = x_{k-1} + w_k$$

여기서,

- x_k : 실제 상태 (true value)
- z_k : 측정값 (measured value)
- w_k : 프로세스 노이즈 (process noise, 시스템 불확실성)
- v_k : 측정 노이즈 (measurement noise, 센서 불확실성)

3. 변수 정의

변수	의미	초기값 권장
x	상태 추정값 (estimated value)	첫 측정값
P	상태 공분산 (추정 불확실성)	1.0 ~ 10.0
Q	프로세스 노이즈 분산	0.001 ~ 0.01
R	측정 노이즈 분산	0.1 ~ 1.0

4. 필터 단계 요약

칼만 필터는 2단계로 수행된다.

(1) 예측 단계 (Prediction)

$$x_{pred} = x_{k-1}$$

(2) 보정 단계 (Correction)

$$K = \frac{P_{pred}}{P_{pred} + R}$$

여기서 **K**는 칼만 이득(Kalman Gain)으로, 측정값과 예측값 중 어느 쪽을 더 신뢰할지 결정한다.

5. C 구현 예시

```
1 typedef struct {
2     float x; // 추정값
3     float P; // 오차 공분산
4     float Q; // 프로세스 노이즈
5     float R; // 측정 노이즈
6     float K; // 칼만 이득
7 } KalmanFilter;
8
9 void Kalman_Init(KalmanFilter *kf, float q, float r, float p, float initial_value)
10 {
11     kf->Q = q;
12     kf->R = r;
13     kf->P = p;
14     kf->x = initial_value;
15 }
16
17 float Kalman_Update(KalmanFilter *kf, float measurement)
18 {
19     // 예측 단계
20     kf->P += kf->Q;
21 }
```

```

22 // 칼만 이득 계산
23 kf->K = kf->P / (kf->P + kf->R);
24
25 // 보정 단계
26 kf->x = kf->x + kf->K * (measurement - kf->x);
27
28 // 공분산 갱신
29 kf->P = (1.0f - kf->K) * kf->P;
30
31 return kf->x;
32 }

```

사용 예:

```

1  KalmanFilter kf;
2  Kalman_Init(&kf, 0.01f, 0.1f, 1.0f, 0.0f);
3
4  while (1)
5  {
6      float raw = HX711_Get_Value();
7      float filtered = Kalman_Update(&kf, raw);
8
9      printf("Raw: %.2f, Kalman: %.2f\n", raw, filtered);
10     HAL_Delay(50);
11 }

```

6. 변수 조정 가이드

변수	작게 설정 시	크게 설정 시	권장 범위
Q	안정적이지만 느림	빠르지만 진동 가능	0.001 ~ 0.01
R	잡음에 민감	반응 둔감	0.1 ~ 1.0
P	초기 불확실성 크기	수렴 속도 영향	1.0 ~ 10.0

- **Q ↑**: 시스템이 자주 변하는 경우 (예: 빠른 수위 변화)
- **R ↑**: 센서 노이즈가 큰 경우 (예: 초음파 센서)
- **Q ↓, R ↓**: 정적 환경에서 안정적 측정

7. 시뮬레이션 예시

측정값 (z)	추정값 (x)	K (이득)	P
100.0	100.0	0.91	0.09
101.2	100.11	0.48	0.05
99.6	100.01	0.34	0.03

측정값 (z)	추정값 (x)	K (이득)	P
100.8	100.15	0.29	0.02
100.1	100.13	0.25	0.02

노이즈가 ±1 수준으로 변하더라도
필터 결과는 거의 일정하게 유지된다.

8. Exponential Smoothing과 비교

구분	Exponential Smoothing	Simple Kalman Filter
파라미터	α (고정계수)	Q, R (노이즈 공분산)
수학적 기반	경험적	확률·통계 기반
적응성	없음	자동 적응 (K 값 갱신)
계산 부하	낮음	약간 높음
응용	저비용 노이즈 완화	고정밀 추정

9. 실무 적용 예

(1) 초음파 거리 측정

```
1 | distance_filtered = kalman_update(&ultra_kf, distance_raw);
```

(2) 수위 센서 다중 샘플 결합

각 센서별 Kalman 필터를 독립 적용 후, 평균 융합 수행.

(3) 로드셀 무게 측정

장기적 드리프트가 있을 때, 지수평활보다 더 안정된 결과 제공.

10. 결론

Simple Kalman Filter는
잡음 환경에서도 신뢰성 높은 센서 데이터를 얻기 위한
가장 강력한 1차 확률 기반 필터이다.
적절한 Q/R 설정만으로,
초음파, VL53L0X, 로드셀, IMU 등 다양한 센서에 범용적으로 적용할 수 있으며,
실시간 시스템에서도 효율적이다.

“Kalman Filter = 스마트한 Exponential Smoothing”

10.3 타임아웃 및 재시도

• HAL I²C Timeout 처리

1. 개요

HAL_I2C 계열 함수(HAL_I2C_Master_Transmit, HAL_I2C_Mem_Read 등)는 I²C 통신 중 응답 지연, 버스 충돌, 슬레이브 응답 없음 등의 상황에서 무한 대기(lock-up)를 방지하기 위해 **Timeout (시간 제한)** 파라미터를 사용한다.

Timeout은 **SCL 클럭 정지, BUSY 플래그 고착, ACK 미응답** 등의 상황에서 함수를 강제 종료하고 HAL_TIMEOUT 상태를 반환하도록 하는 보호 메커니즘이다.

2. Timeout의 동작 원리

모든 HAL I²C API는 내부적으로 다음 구조를 따른다.

```
1 HAL_StatusTypeDef HAL_I2C_Master_Transmit(  
2     I2C_HandleTypeDef *hi2c,  
3     uint16_t DevAddress,  
4     uint8_t *pData,  
5     uint16_t Size,  
6     uint32_t Timeout  
7 );
```

내부적으로는 다음과 같이 **SDA/SCL 상태 감시 루프**에서 Timeout 카운트를 체크한다.

```
1 tickstart = HAL_GetTick();  
2  
3 while (__HAL_I2C_GET_FLAG(hi2c, I2C_FLAG_BUSY))  
4 {  
5     if ((HAL_GetTick() - tickstart) > Timeout)  
6     {  
7         hi2c->ErrorCode |= HAL_I2C_ERROR_TIMEOUT;  
8         return HAL_TIMEOUT;  
9     }  
10 }
```

즉, **Timeout(ms)** 동안 BUSY나 TXE, RXNE 등의 상태 변화가 없으면 즉시 통신을 중단하고 상위로 HAL_TIMEOUT 을 반환한다.

3. Timeout 파라미터 설정

환경	권장 Timeout	설명
EEPROM (24C02 등)	10~50 ms	페이지 쓰기 대기 시간 포함
OLED (SSD1306)	5~20 ms	명령/데이터 전송

환경	권장 Timeout	설명
VL53L0X / 센서류	50~100 ms	내부 측정 완료 대기
일반 단일 전송	10 ms	기본 통신

보통 HAL 기본 예제는 `HAL_MAX_DELAY` 를 사용하지만,
이 경우 슬레이브 응답이 없으면 **무한 대기**가 발생하므로
실제 제품에서는 반드시 제한 시간(ms)을 명시해야 한다.

예시:

```
1 | HAL_I2C_Mem_Read(&hi2c1, DEV_ADDR, REG_ADDR, 1, buffer, 2, 50);
```

4. Timeout 발생 시 리턴값

반환값	의미	조치
<code>HAL_OK</code>	정상 완료	문제 없음
<code>HAL_TIMEOUT</code>	지정 시간 초과	버스 점유, 슬레이브 미응답 가능
<code>HAL_ERROR</code>	ACK 실패, NACK 수신	어드레스 또는 배선 확인 필요
<code>HAL_BUSY</code>	다른 I2C 작업 중	Mutex 또는 Task 동기화 필요

5. Timeout 발생 원인

원인	설명
SCL 고정 (Low Level)	슬레이브가 Clock Stretch 중이거나 핀 고착
SDA stuck Low	직전 통신 중 비정상 종료
슬레이브 전원 Off	주소 응답 없음, BUSY 유지
풀업 저항 과대/과소	Rise time 이상으로 SCL 타이밍 실패
노이즈 유입	START/STOP 프레임 손상으로 BUSY 유지

6. 복구 절차 (Timeout 이후 버스 정지 복원)

Timeout 이후 I2C 버스가 **BUSY** 상태로 고착되는 경우가 있다.
이때는 펌웨어 수준에서 수동 복구가 필요하다.

(1) SCL 강제 토글로 Bus Release

SCL을 GPIO로 전환하고, 수동으로 클럭 9회를 발생시킨다.

```
1 void I2C_Bus_Recovery(void)
2 {
3     GPIO_InitTypeDef GPIO_InitStruct = {0};
4
5     // SCL, SDA 핀을 GPIO Output으로 변경
6     GPIO_InitStruct.Pin = GPIO_PIN_6 | GPIO_PIN_7;
7     GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_OD;
8     GPIO_InitStruct.Pull = GPIO_NOPULL;
9     GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
10    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
11
12    // SCL 9회 토글 (슬레이브 내부 상태기계 리셋)
13    for (int i = 0; i < 9; i++)
14    {
15        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
16        HAL_Delay(1);
17        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
18        HAL_Delay(1);
19    }
20
21    // STOP 조건 강제 발생
22    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_RESET);
23    HAL_Delay(1);
24    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
25    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_SET);
26 }
```

(2) I²C 재초기화

```
1 HAL_I2C_DeInit(&hi2c1);
2 HAL_I2C_Init(&hi2c1);
```

(3) 재시도 로직

```
1 if (HAL_I2C_Master_Transmit(&hi2c1, addr, data, len, 50) != HAL_OK)
2 {
3     I2C_Bus_Recovery();
4     HAL_I2C_DeInit(&hi2c1);
5     HAL_I2C_Init(&hi2c1);
6 }
```

7. 실무 팁

항목	권장 설정 / 대응
Timeout 관리	HAL_MAX_DELAY 대신 실측 기반 ms 지정
재시도 횟수	최대 3회 정도 (EEPROM 등)
이벤트 로그	HAL_TIMEOUT 시 에러코드/시간 기록
I ² C 상태 확인	통신 전 HAL_I2C_GetState() 체크
FreeRTOS 사용 시	Mutex 보호 + osDelay() 포함

8. 예시 코드

```
1  HAL_StatusTypeDef I2C_Write_WithTimeout(uint8_t addr, uint8_t *pData, uint16_t
   size)
2  {
3      HAL_StatusTypeDef ret;
4      ret = HAL_I2C_Master_Transmit(&hi2c1, addr, pData, size, 50);
5
6      if (ret == HAL_TIMEOUT)
7      {
8          printf("[I2C] Timeout occurred, recovering bus...\n");
9          I2C_Bus_Recovery();
10         HAL_I2C_DeInit(&hi2c1);
11         HAL_I2C_Init(&hi2c1);
12     }
13     return ret;
14 }
```

9. 결론

HAL I²C Timeout은 시스템 응답성을 유지하고 버스 고착을 예방하기 위한 핵심적인 보호 기법이다.

정상적인 임베디드 I²C 설계에서는

- 적절한 Timeout 지정
- 버스 복구 함수 구현
- 재시도 및 로그 처리
를 반드시 포함해야 안정적인 통신이 보장된다.

Timeout은 오류가 아니라 “회복 기회”다.

• 재시도 루프, 오류 카운터

1. 개요

I²C, UART, SPI 등 주변장치 통신에서 간헐적인 전기적 노이즈, 응답 지연, 슬레이브 Busy 상태 등이 발생할 수 있다.

이러한 일시적 오류는 시스템 오류로 처리하기보다

“재시도(Retry) 메커니즘”을 통해 회복시키는 것이 바람직하다.

STM32 HAL 계층은 오류 발생 시 `HAL_ERROR`, `HAL_TIMEOUT`, `HAL_BUSY` 등의 상태를 반환하며, 사용자는 상위 애플리케이션 계층에서 이를 감지하여 **재시도 루프**를 수행해야 한다.

2. 재시도 루프 기본 구조

다음은 전형적인 재시도 루프의 구조이다.

```
1  HAL_StatusTypeDef I2C_write_withRetry(  
2      I2C_HandleTypeDef *hi2c,  
3      uint16_t DevAddr,  
4      uint8_t *pData,  
5      uint16_t Size,  
6      uint32_t Timeout,  
7      uint8_t RetryCount  
8  )  
9  {  
10     HAL_StatusTypeDef ret;  
11  
12     for (uint8_t i = 0; i < RetryCount; i++)  
13     {  
14         ret = HAL_I2C_Master_Transmit(hi2c, DevAddr, pData, Size, Timeout);  
15         if (ret == HAL_OK)  
16             return HAL_OK;  
17  
18         // 오류 발생 시 약간의 지연 후 재시도  
19         HAL_Delay(5);  
20     }  
21  
22     // 모든 시도 실패 시 HAL_ERROR 반환  
23     return ret;  
24 }
```

위 코드에서는 `RetryCount` 횟수만큼 반복 시도하며,
정상 응답(`HAL_OK`)이 들어올 때까지 루프를 수행한다.
모든 시도가 실패하면 마지막 오류 코드를 그대로 반환한다.

3. 오류 카운터 관리

오류가 반복적으로 발생하는 경우를 감지하기 위해
글로벌 또는 정적 변수 기반의 오류 카운터를 두어야 한다.

```
1 static uint16_t i2c_error_count = 0;
2
3 HAL_StatusTypeDef Safe_I2C_Read(uint8_t addr, uint8_t reg, uint8_t *buf, uint16_t
  len)
4 {
5     HAL_StatusTypeDef ret;
6
7     ret = HAL_I2C_Mem_Read(&hi2c1, addr, reg, 1, buf, len, 50);
8     if (ret != HAL_OK)
9     {
10         i2c_error_count++;
11         printf("[I2C] Error #%d at addr 0x%02X\n", i2c_error_count, addr);
12     }
13     else
14     {
15         i2c_error_count = 0; // 성공 시 카운터 리셋
16     }
17
18     return ret;
19 }
```

이 방식으로 통신 안정성을 실시간으로 모니터링할 수 있으며,
지속적 오류 누적 시 Fail-Safe 동작을 수행할 수 있다.

4. 오류 카운터 기반 복구 로직

조건	동작
<code>i2c_error_count < 3</code>	단순 재시도 수행
<code>i2c_error_count ≥ 3</code>	버스 리커버리(<code>I2C_Bus_Recovery()</code>), HAL 재초기화
<code>i2c_error_count ≥ 10</code>	시스템 리셋 또는 Fail-safe 모드 진입

예시:

```
1 if (i2c_error_count >= 3)
2 {
3     printf("[I2C] Reinitializing due to repeated errors...\n");
4     I2C_Bus_Recovery();
5     HAL_I2C_DeInit(&hi2c1);
6     HAL_I2C_Init(&hi2c1);
7 }
```

5. 지연 및 재시도 간격

짧은 시간 간격으로 재시도하면 슬레이브가 여전히 Busy 상태일 수 있다.
따라서 다음과 같은 **재시도 간격(Delay)** 을 두는 것이 바람직하다.

장치 유형	권장 재시도 간격	비고
EEPROM	5~10 ms	내부 Write Cycle 완료 대기
센서 (VL53L0X 등)	10~50 ms	내부 측정 완료 후 응답
OLED	2~5 ms	명령 처리 완료 대기

6. FreeRTOS 환경에서의 재시도

FreeRTOS를 사용하는 경우, 바쁜 루프(`HAL_Delay`) 대신 **비블로킹 지연**을 사용한다.

```
1 for (uint8_t i = 0; i < RetryCount; i++)
2 {
3     ret = HAL_I2C_Master_Transmit(hi2c, addr, pData, len, Timeout);
4     if (ret == HAL_OK) break;
5     osDelay(5); // Task 지연 (CPU 양보)
6 }
```

7. 실무 적용 예시

```
1 #define MAX_I2C_RETRY 3
2
3 HAL_StatusTypeDef I2C_Write_Robust(uint8_t addr, uint8_t reg, uint8_t data)
4 {
5     uint8_t buffer[2] = { reg, data };
6     HAL_StatusTypeDef ret;
7
8     for (int attempt = 1; attempt <= MAX_I2C_RETRY; attempt++)
9     {
10         ret = HAL_I2C_Master_Transmit(&hi2c1, addr, buffer, 2, 50);
11         if (ret == HAL_OK)
12         {
13             printf("[I2C] write success (try %d)\n", attempt);
14             return HAL_OK;
15         }
16
17         printf("[I2C] write failed (try %d)\n", attempt);
18         HAL_Delay(5);
19     }
20
21     printf("[I2C] write permanently failed after %d tries\n", MAX_I2C_RETRY);
22     return ret;
23 }
```

8. 결론

재시도 루프와 오류 카운터는 단순한 신뢰성 향상을 넘어,
임베디드 시스템의 **Fault Tolerance (내결함성)** 을 확보하는 핵심 기법이다.

- 재시도는 일시적 장애를 회복시키는 1차 방어선
- 오류 카운터는 지속적 문제를 감지하는 2차 감시선
- 자동 복구 루틴은 시스템 정지 없는 3차 복구선

“Retry + Error Counter = Self-healing Communication”

• Fail-Safe 제어 루틴

1. 개요

Fail-Safe(안전 실패 제어) 는 센서 오류, 통신 장애, 전원 불안정 등의 비정상 상태에서도
시스템이 **예측 가능한 안전 상태**로 자동 복귀하도록 설계하는 기법이다.

임베디드 시스템에서는 센서 측정 오류, 무응답 장치, 버스 충돌 등이 발생할 수 있으며,
이때 제어 루프가 멈추거나 잘못된 제어를 수행하면 심각한 결과를 초래할 수 있다.
따라서 **Fail-Safe 루틴**은 실시간 안정성 확보의 마지막 방어선이다.

2. Fail-Safe 설계 원칙

원칙	설명
안전 우선 (Safety First)	기능 정지보다 안전 상태 유지가 우선
자동 복구 (Self-Recovery)	일정 조건에서 자동 재시도 또는 재초기화
격리 (Isolation)	오류 모듈을 시스템 전체와 분리
로깅 (Traceability)	오류 발생 시점과 원인 기록

3. 구조 개요

```
1  [정상 동작 루프]
2      ↓
3  [오류 감지]
4      ↓
5  [Fail-Safe 루틴 진입]
6      ↓
7  - 출력 차단 / 밸브 닫기
8  - 통신 재시도
9  - 모듈 재초기화
10     ↓
11 [정상 복귀 or 시스템 리셋]
```

4. 주요 구현 항목

(1) 오류 감지 조건

- 센서 데이터 NaN / out-of-range
- I2C 오류 카운터 ≥ 임계값
- 통신 타임아웃 지속 발생
- 하드웨어 Fault (HardFault, BusFault 등)

예시 코드:

```
1  if (i2c_error_count >= 10 || sensor_value < 0 || sensor_value > MAX_RANGE)
2  {
3      System_FailSafe("Sensor Fault Detected");
4  }
```

(2) Fail-Safe 루틴 함수

```
1  void System_FailSafe(const char *reason)
2  {
3      printf("\n[FAIL-SAFE] Triggered: %s\n", reason);
4
5      // 모든 출력 차단
6      HAL_GPIO_WritePin(VALVE_GPIO_Port, VALVE_Pin, GPIO_PIN_RESET);
7      HAL_GPIO_WritePin(PUMP_GPIO_Port, PUMP_Pin, GPIO_PIN_RESET);
8
9      // 안전 표시 (예: 빨간 LED 점등)
10     HAL_GPIO_WritePin(LED_R_GPIO_Port, LED_R_Pin, GPIO_PIN_SET);
11     HAL_GPIO_WritePin(LED_G_GPIO_Port, LED_G_Pin, GPIO_PIN_RESET);
12
13     // 상태 기록 (EEPROM, Flash, RTC log)
14     Log_SaveFailEvent(reason);
15
16     // 통신 재시도 또는 모듈 재초기화
17     I2C_Bus_Recovery();
18     HAL_I2C_DeInit(&hi2c1);
19     HAL_I2C_Init(&hi2c1);
20
21     // 일정 시간 대기 후 복구 시도
22     HAL_Delay(500);
23 }
```

(3) Fail Counter 기반 복구 로직

상태	조치
<code>fail_count < 3</code>	루틴 진입 후 자동 복구

상태	조치
<code>3 ≤ fail_count < 10</code>	하위 모듈 재초기화 수행
<code>fail_count ≥ 10</code>	시스템 소프트 리셋 수행

```
1 static uint8_t fail_count = 0;
2
3 void Handle_FailSafe(void)
4 {
5     fail_count++;
6
7     if (fail_count < 3)
8         System_FailSafe("Temporary Error");
9     else if (fail_count < 10)
10         HAL_NVIC_SystemReset();
11 }
```

5. Fail-Safe 동작 예시

상황	Fail-Safe 동작	복구 방법
I ² C 통신 단절	I2C_DeInit → ReInit → 재시도	자동 복귀
수위 센서 불량	수위 표시 0, 펌프 정지	수동 점검
초음파 이상치 검출	최근 정상값 유지(Hold)	자동 복귀
무게 ADC 무응답	출력 차단 후 HX711 리셋	자동 복귀
RTC Alarm 미동작	타임베이스 재초기화	자동 복귀

6. FreeRTOS 환경에서의 Fail-Safe

Fail-Safe는 별도의 감시(Task Monitor) 또는 Watchdog Task 에서 주기적으로 감시한다.

```
1 void FailSafeTask(void *argument)
2 {
3     for (;;)
4     {
5         if (i2c_error_count > 5 || sensor_timeout_flag)
6         {
7             System_FailSafe("Peripheral Fault");
8         }
9         osDelay(1000);
10    }
11 }
```

또는 하드웨어 Watchdog과 연동해 치명적 정지 시 자동 리셋:

```
1 void watchdogTask(void *argument)
2 {
3     for (;;)
4     {
5         HAL_IWDG_Refresh(&hiwdg); // 정상 상태일 때만 리프레시
6         osDelay(1000);
7     }
8 }
```

7. 시각적 피드백

시스템이 Fail-Safe 상태로 전환되면 LED, OLED, UART 로그로 표시한다.

```
1 OLED_ShowString(0, 0, "FAIL-SAFE MODE");
2 UART_Log("[SYSTEM] Fail-Safe Mode Entered\n");
```

8. 핵심 요약

핵심 요소	설명
Fail Detection	오류 감지(센서, 통신, 내부 Fault)
Safe Shutdown	위험 동작 차단 (밸브, 펌프, 모터 등)
Recovery Attempt	모듈 재초기화 및 재시도
Persistent Fault	카운터 누적 시 시스템 리셋
Logging & Indication	오류 정보 저장 및 LED/OLED 표시

✅ 정리

Fail-Safe 루틴은 단순 오류 복구가 아니라,
“시스템이 스스로 멈추지 않고 안전하게 살아남는 구조”를 만드는 것이다.

Detect → Isolate → Recover → Resume (DIRR Cycle)