

5. I²C 통신 (센서 데이터 인터페이스)

5.1 I²C 기본

• SDA/SCL 동작, Start/Stop/ACK 프레임

다음 내용은 I²C(Inter-Integrated Circuit) 버스의 물리·논리 동작을 규격서 스타일로 정리한 것이다. 목표는 SDA(데이터 선)와 SCL(클럭 선)의 신호 동작, Start/Stop 조건, ACK/NACK 프레임의 정확한 의미와 생성/판단 방법을 완전하게 이해시키는 것이다.

1. 개요 — 기본 원리

- I²C는 2선식 직렬 통신 버스이다. 두 선은 SDA(Serial Data)와 SCL(Serial Clock)이다.
- 두 선 모두 **오픈-드레인(open-drain)**(또는 오픈-콜렉터) 출력으로 동작한다. 즉, 어떤 장치도 선을 적극적으로 High로 구동하지 않으며, Low는 능동적으로 드라이브하고 High는 풀업 저항에 의해 형성된다.
- 버스 레벨: 논리 '1'은 선이 풀업에 의해 High(릴렉스 상태), 논리 '0'은 장치가 선을 Low로 드라이브한 상태.
- I²C는 마스터-슬레이브 구조(멀티-마스터 가능). 마스터는 클럭을 생성하고 전송을 제어한다. 멀티마스터 환경에서는 **arbitration(중재)**와 **clock synchronization(클럭 동기화/스트레칭)** 규칙이 적용된다.

2. 물리적 요구사항(하드웨어)

- SDA, SCL 라인에 **풀업 저항(R_pullup)** 필요. 값은 버스 속도 및 총 정전 용량(C_{bus})에 따라 결정된다. 대략:
 - 표준 모드(100 kHz): 1 kΩ ~ 10 kΩ 범위 사용 가능
 - 고속 모드(400 kHz 이상): 풀업 저항을 더 낮춰야 함(더 큰 전류 소모)
- 버스 정전용량(C_{bus})은 라인 길이와 디바이스 수에 비례한다. 높은 C_{bus}는 상승시간(t_r)을 증가시켜 타이밍 위반 유발.
- 슬레이브/마스터는 SDA/SCL을 **GPIO 오픈드레인** 또는 전용 I²C 하드웨어로 구성. 출력은 Low 드라이브 또는 Hi-Z(릴렉스) 상태만 사용.

3. 버스 상태 정의

- **버스 유휴(Idle):** SCL = High, SDA = High. 이 상태가 기본 대기 상태.
- **Start 조건 (START, S):** SDA가 High → Low로 변하면서 동시에 SCL이 High인 순간. (SCL High 동안 SDA가 하강)
 - Start는 버스의 독점 권한을 선언하는 신호이며, 이후부터 바이트 전송이 시작된다.
- **Stop 조건 (STOP, P):** SDA가 Low → High로 변하면서 SCL이 High인 순간. (SCL High 동안 SDA가 상승)
 - Stop은 버스 트랜잭션의 종료를 알리고 버스를 유휴 상태로 복귀시킨다.
- **Repeated Start (Sr):** Stop 대신 다시 Start를 발생시키는 것. 마스터가 버스 제어권을 유지한 채 다른 전송(예: 읽기 전환)을 시작할 때 사용.

4. Start/Stop의 정확한 신호 타이밍(정의적 조건)

- Start: 조건은 SDA가 SCL이 High인 동안 High→Low 전환일 때 유효. SCL이 Low일 때 SDA 전환은 데이터 비트 전송으로 해석된다(데이터 비트 전환 규칙과 충돌하지 않도록 주의).
 - Stop: SDA가 SCL이 High인 동안 Low→High 전환일 때 유효.
 - Start/Stop 타이밍에는 세부적인 setup/hold 요구시간이 존재(규격서의 t_HD;STA, t_SU;STA, t_SU;STO 등). 구현 시 MCU의 I²C 하드웨어와 트랜시버 스펙을 확인.
-

5. 바이트 전송과 ACK/NACK 프레임 — 기본 흐름

1. 마스터가 Start를 발생시킨다.
 2. 마스터는 8비트(1바이트)를 MSB(최상위 비트)부터 SDA에 설정하고, 각 비트마다 SCL을 Low→High→Low로 토글한다.
 - 데이터 유효 시점: SDA는 SCL의 상승 전(또는 정의된 setup 시간)까지 안정되어 있어야 하며, SCL이 High인 동안 SDA는 변하면 안 된다(hold).
 3. 8비트 전송이 끝나면 송신 쪽은 SDA를 릴렉스(Hi-Z) 상태로 두고, 수신 쪽(ACK/NACK을 줄 장치)은 9번째 클럭에 지 동안 SDA를 드라이브할 수 있다.
 - 9번째 클럭: SCL이 Low에서 High로 상승하는 동안 ACK/NACK 비트가 읽힌다.
 4. **ACK(응답)**: 수신 장치가 해당 바이트를 올바로 수신했음을 알리기 위해 SDA를 Low로 드라이브한다(논리 0).
 5. **NACK(비응답)**: 수신 장치가 응답하지 않거나 더 이상 데이터를 원하지 않을 경우 SDA를 릴렉스(High) 상태로 유지 한다(논리 1).
 6. ACK/NACK 판정 후, 송신자는 다음 바이트 전송 또는 Stop/Repeated Start를 수행한다.
-

6. 주소 전송과 R/W 비트

- 첫 번째 바이트(주소 바이트)는 7-bit 주소 + R/W 비트(LSB)로 구성되거나, 확장형의 경우 10-bit 주소 모드가 사용된다.
 - 7-bit 주소 방식: [A6 A5 A4 A3 A2 A1 A0][R/W] (총 8비트)
 - R/W = 0 → 쓰기(마스터가 슬레이브로 데이터 전송)
 - R/W = 1 → 읽기(마스터가 슬레이브로부터 데이터 수신)
 - 주소 전송 후 슬레이브는 주소 비교에 성공하면 ACK(0)를 보낸다. ACK가 없으면(= NACK) 그 주소에는 응답 장치 없음.
-

7. ACK의 발생 주체와 생성 규칙

- ACK는 항상 **수신자가** 생성한다.
 - 마스터가 주소/데이터를 보냈다면 슬레이브가 ACK를 보낸다.
 - 슬레이브가 데이터를 보낼 때(읽기 모드), 마스터가 각 수신 바이트에 대해 ACK를 보낸다(마지막 바이트 수신 후 마스터는 NACK를 보내며 읽기 종료를 알린다).
- ACK를 생성하는 구체적 방법:
 - ACK 비트 시간(9번째 클럭) 동안 수신자는 SDA를 Low로 드라이브.

- 송신자는 이 9번째 클럭 동안 SDA 레벨을 읽어서 ACK(0) 또는 NACK(1)를 판정.
 - ACK를 수신하지 못하면(= NACK) 송신자는 보통 재시도하거나 Stop을 발생시킨다.
-

8. 읽기 흐름(마스터가 슬레이브에서 읽을 때) — ACK/NACK 역할

- 마스터가 슬레이브에게 읽기 요청(R/W=1)을 보낸 뒤,
 - 슬레이브는 버퍼에서 바이트를 준비하여 SCL에 맞춰 SDA로 전송.
 - 각 바이트 전송 후 마스터는 응답으로 **ACK(0)**를 보내면 슬레이브는 다음 바이트 전송을 계속한다.
 - 마지막으로 마스터가 더 이상 바이트를 원하지 않으면 NACK(1)을 보낸 뒤 Stop 또는 Repeated Start로 트랜잭션을 종료한다.
 - 따라서 마스터는 읽기 시 **바이트 수 제어자** 역할을 ACK/NACK로 수행.
-

9. Repeated Start(연속 Start)의 용도와 동작

- Repeated Start는 한 트랜잭션 내에서 Stop 없이 Start를 다시 발생시켜 다른 동작으로 전환할 때 사용.
 - 전형적 사용 예: 슬레이브 내부 레지스터에 쓰기(주소 지정) → Repeated Start → 슬레이브에서 읽기(같은 주소의 레지스터 읽기).
 - Repeated Start는 버스 제어권을 유지하면서 슬레이브가 버스 상태를 재초기화해야 하는 오버헤드를 제거한다.
 - 신호적으로는 Start와 동일한 조건(SCL High 동안 SDA High→Low).
-

10. 멀티마스터, 중재(Arbitration) 및 클럭 동기화

- 멀티마스터 환경에서는 여러 마스터가 동시에 Start를 발생시킬 수 있다. 다음 규칙 적용:
 - **Arbitration:** 각 마스터는 전송 중 SDA 레벨을 모니터링하여 자신이 Low를 보냈으나 버스가 High로 유지되면(= 다른 마스터가 High로 남겼다) 패배로 간주하고 전송을 포기해야 한다. 반대의 경우(자신이 High였는데 버스가 Low이면)도 패배. 핵심: 장치는 본인이 보내려는 비트와 실제 버스 비트가 다른 경우 전송을 중단.
 - Arbitration은 비트 단위로 진행되며, SCL 라인의 동기화 규칙에 의해 멀티마스터의 SCL은 Low-OR(AND?) 메커니즘으로 조합된다. (각 마스터는 SCL을 Low로 드라이브할 수 있고, High는 오픈 상태여서 풀업에 의해 형성)
 - **Clock synchronization / Clock stretching:**
 - 슬레이브 또는 다른 마스터가 처리량이 부족할 때 SCL을 Low로 유지(드라이브)함으로써 클럭을 지연시킬 수 있다. 이 동작을 **clock stretching**이라 한다.
 - 마스터는 SCL을 다시 High로 올렸을 때 실제로 High인지(=선이 릴렉스되어 풀업으로 High가 되었는지) 확인해야 하며, 계속 Low이면 슬레이브가 스트레칭 중으로 판단하고 대기해야 한다.
-

11. 타이밍 파라미터(중요 항목)

- I²C 규격에는 많은 타이밍 파라미터가 있음(이하는 대표적 항목, 정확 수치는 규격서 참조):
 - **f_SCL:** SCL 클럭 주파수 (standard-mode: up to 100 kHz, fast-mode: up to 400 kHz, fast-mode plus: up to 1 MHz, high-speed: 3.4 MHz)
 - **t_r:** SDA/SCL 상승시간 (rise time) — 풀업과 버스 정전용량에 좌우.

- **t_f**: 하강시간 (fall time) — 장치가 Low로 드라이브할 때의 시간.
 - **t_HD;STA**: Start hold time — Start 후 마스터는 적어도 이 시간만큼 SDA를 Low로 유지.
 - **t_SU;STA**: Start setup time — Start 발생 전에 SDA가 Stable해야 하는 최소 시간.
 - **t_SU;DAT**: 데이터 setup 시간 — 데이터가 SCL 상승 전에 안정해야 하는 시간.
 - **t_HD;DAT**: 데이터 hold 시간 — SCL이 상승 후 데이터가 유지되어야 하는 최소 시간.
 - **t_SU;STO**: Stop setup time — Stop을 생성하기 전에 SDA가 Low여야 하는 최소 시간.
 - 구현 시 MCU 레퍼런스 매뉴얼과 I²C 디바이스 데이터시트의 타이밍 요구사항을 반드시 확인.
-

12. 에러 사례 및 예외 처리

- **NACK 발생**:
 - 주소 NACK: 슬레이브 없음 또는 슬레이브가 바쁨 → 재시도 또는 오류 보고.
 - 데이터 NACK: 슬레이브 내부 버퍼 풀 또는 읽기 완료 신호 → 전송 중단.
 - **Bus busy 이상**: SDA 또는 SCL가 Low로 유지되어 Start/Stop 생성 불가 → 모든 마스터가 라인을 릴렉스시키고, 소프트웨어적으로 Bus Clear 루틴(클럭 펄스를 여러 번 생성하여 슬레이브 복구 시도)을 수행.
 - **Clock stretching 과다**: 슬레이브가 과도하게 SCL을 Low로 유지하면 마스터 측 타임아웃 필요.
 - **Bus contention(결선 충돌)**: 하드웨어 결함으로 라인이 강제로 Low로 유지되는 경우, 버스가 비정상 상태로 고착될 수 있음.
-

13. 소프트웨어/펌웨어 구현시 체크리스트

- SDA/SCL GPIO는 **오픈드레인**, 풀업 활성화로 설정.
 - 풀업저항 값과 버스 정전용량 사이의 균형 확인(속도·전류·타이밍 트레이드오프).
 - Start, Stop, Repeated Start의 타이밍 조건을 준수.
 - 데이터 전송 후 **9번째 클럭(ACK 비트)**를 반드시 읽어 ACK/NACK 판단.
 - 읽기 동작 시 마스터가 마지막 바이트를 수신하면 NACK를 보내고 Stop을 발생.
 - 멀티마스터 기능 사용 시 arbitration 모니터링 루틴 구현.
 - 클럭 스트레칭을 허용하도록 SCL 읽기/확인 루틴 구현(마스터가 SCL을 High로 올린 후 실제 High인지 확인).
 - 에러 발생시 재시도 정책(횟수 제한, 지수 백오프 등)과 버스 클리어 방법을 포함.
-

14. 예제 타이밍 다이어그램 (ASCII)

```
1 Idle: SCL=1, SDA=1
2 Start: SDA falls while SCL=1
3
4 SCL   _____ \_____/ \_____
5 SDA   / \____/ \____/ \____/ \____/ (bits)
6     ^ bit0 ^ bit1 ^ ...     ^ ACK bit (9th)
7 Start: |_____
8      | b7 b6 b5 b4 b3 b2 b1 b0
9
10 After 8 bits, master releases SDA (Hi-Z)
11 ACK bit: slave pulls SDA low during 9th SCL high
12 Stop: SDA rises while SCL=1
```

(위 다이어그램은 원리 설명용이며 실제 타이밍 값은 규격 참조)

15. 실제 설계 팁 / 권장사항

- 풀업 저항은 보드 설계 초기에 결정. 만약 여러 장치가 다른 풀업을 갖고 있다면 병렬 저항값이 너무 낮아질 수 있음 (전류 소모 증가).
- 라인 길이가 길면 신호 반사 및 잡음 유입 가능 — 배선은 짧게, 필요 시 차폐 고려.
- I²C 트랜시버(레벨 시프터, 버퍼)를 사용하면 여러 전압 도메인(3.3V ↔ 5V) 연결에 유리.
- 슬레이브 장치의 데이터시트에 명시된 ACK/NACK 동작을 반드시 확인(특정 슬레이브는 특정 시퀀스에서 독특하게 동작).
- 디버깅 시 로직 분석기 또는 오실로스코프로 SDA/SCL 파형을 캡쳐하여 Start/Stop, ACK 타이밍, 클럭 스트레칭을 확인.

16. 요약(핵심 포인트)

- SDA와 SCL은 오픈-드레인 구조로, High는 풀업, Low는 드라이브로 형성된다.
- Start는 SCL=High에서 SDA High→Low, Stop은 SCL=High에서 SDA Low→High로 규정된다.
- 데이터는 8비트 MSB 우선 전송, 9번째 클럭에서 ACK/NACK 판정.
- ACK는 수신자가 Low로 드라이브해서 보낸다(0 = ACK, 1 = NACK).
- Repeated Start는 Stop 없이 트랜잭션 전환을 가능하게 함.
- 멀티마스터 환경에서는 arbitration과 clock stretching을 고려해야 함.
- 물리적 설계(풀업, 버스 용량, 라인 길이)와 타이밍(Setup/Hold, rise/fall time)이 통신 신뢰성에 결정적 역할을 함.

• 7-bit, 10-bit Address 구조

1. 개요

I²C 버스는 슬레이브 디바이스를 식별하기 위해 주소 프레임(address frame)을 사용한다.

I²C 표준은 **7-bit 주소 체계**와 **10-bit 주소 체계** 두 가지를 정의한다.

두 방식 모두 첫 바이트의 최하위 비트(LSB)는 **R/W 비트**로 사용되며, 상위 비트는 주소 비트로 채워진다.

2. 7-bit 주소 구조

2.1 비트 구성

1		A6		A5		A4		A3		A2		A1		A0		R/W	
---	--	----	--	----	--	----	--	----	--	----	--	----	--	----	--	-----	--

- **A6~A0:** 슬레이브 주소 (7비트)
- **R/W:** 읽기(1) / 쓰기(0) 비트

2.2 동작

- 마스터는 Start 신호 이후, 위 구조의 1바이트를 전송한다.
- 슬레이브는 이 중 자신의 주소와 상위 7비트를 비교한다.
- 일치하면 슬레이브는 ACK(0)를 보낸다.
- 이후의 데이터 바이트는 이 주소에 해당하는 슬레이브와만 주고받는다.

2.3 주소 범위

- 유효 주소 범위: 0x00 ~ 0x7F (128개)
- 실제 사용 가능한 주소 수는 약 112개로, 일부 주소는 예약됨(예: 0x00~0x07, 0x78~0x7F 등).

2.4 예약 주소 (예시)

주소 (7-bit)	용도
0000 000	일반 호출(General Call)
0000 001	CBUS 주소
0000 010	Reserved for future use
0000 011	Reserved for future use
0000 100	Hs-mode master code
1111 1xx	Reserved for 10-bit addressing

3. 10-bit 주소 구조

3.1 도입 배경

- 7-bit 주소로는 최대 128개의 슬레이브만 식별 가능하다.
- 산업/자동차/센서 네트워크 등에서 더 많은 디바이스를 수용하기 위해 10-bit 주소 모드가 추가됨.

3.2 비트 구성 (두 바이트 전송)

- 1 첫 번째 바이트: 11110 A9 A8 R/W
- 2 두 번째 바이트: A7 A6 A5 A4 A3 A2 A1 A0

3.3 전송 절차

1. 마스터는 Start 신호 발생.
2. 첫 바이트($11110 + A9 + A8 + R/W=0$)를 전송.
 - 상위 5비트 11110은 **10-bit 주소 모드 식별 패턴**.
3. 해당 주소를 인식한 슬레이브가 ACK(0)를 응답.
4. 두 번째 바이트로 나머지 8비트 주소(A7~A0) 전송.
5. 슬레이브가 다시 ACK(0) 응답.
6. 이후 R/W 비트가 1인 읽기 동작이 필요한 경우, **Repeated Start**를 발생시켜 같은 10-bit 주소로 읽기 요청을 보낸다.
 - 두 번째 Start 후 첫 바이트의 R/W=1로 다시 전송.

3.4 요약 타이밍

- 1 Start → [11110 + A9 + A8 + 0] → ACK
- 2 → [A7~A0] → ACK
- 3 → Data bytes ...

(읽기 시: Repeated Start 후 R/W=1로 첫 바이트 재전송)

4. 비교 요약표

항목	7-bit 주소	10-bit 주소
주소 공간	128개 (일부 예약)	1024개 (0x000~0x3FF)
첫 바이트 구조	[A6~A0][R/W]	[11110 A9 A8][R/W]
두 번째 바이트	없음	[A7~A0]
R/W 전환 방식	단일 Start	Repeated Start 필요
지원 디바이스	대부분 기본 지원	일부 MCU/센서만 지원
주소 예약 영역	0000xxx, 1111xxx	11110xxx (10-bit 식별)

5. 실제 동작 예시

5.1 7-bit 주소 예 (주소 = 0x48, 쓰기)

```
1 Start
2 → 0x90 (0b10010000) // 0x48 << 1 | 0
3 → ACK
4 → Data byte(s)
5 → Stop
```

5.2 10-bit 주소 예 (주소 = 0x2D3, 읽기)

```
1 주소 = 0b10 1101 0011 (A9..A0)
2
3 Start
4 → 11110 10 0 (0xF4) // 첫 바이트: 상위 5비트 + A9,A8 + R/W=0
5 → ACK
6 → 1101 0011 (0xD3) // 두 번째 바이트
7 → ACK
8 → Repeated Start
9 → 11110 10 1 (0xF5) // 첫 바이트, R/W=1
10 → ACK
11 → Data from slave...
12 → NACK
13 → Stop
```

6. HAL 드라이버 관점 (STM32 예시)

- HAL에서는 항상 8비트 주소 프레임(`addr << 1`)을 인자로 사용한다.
 - 즉, HAL 함수 내에서 R/W 비트는 자동으로 추가된다.
 - 예: `HAL_I2C_Master_Transmit(&hi2c1, 0x48 << 1, buf, len, 100);`
- 10-bit 모드를 지원하는 경우 `hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_10BIT;`로 설정.

7. 주요 유의사항

- 일부 디바이스는 10-bit 주소를 인식하지 못하므로 호환성 문제 발생 가능.
- 10-bit 주소의 상위 패턴(11110xx)은 7-bit 주소 예약 영역과 중복되므로, 7-bit 모드 장치는 해당 프레임을 무시한다.
- 멀티슬레이브 버스에서 7-bit과 10-bit 디바이스 혼용 가능하나, 마스터가 각 모드별 전송 시퀀스를 구분해야 함.

8. 결론

- 7-bit 주소는 단순하고 거의 모든 I²C 장치가 지원한다.
- 10-bit 주소는 대규모 또는 고밀도 시스템에서 유용하지만, 구현 복잡도 증가.
- 주소 구조 차이를 정확히 이해해야 HAL 함수 호출 시 `addr << 1` 오류, NACK, 통신 실패를 방지할 수 있다.

• HAL I²C 주소 시프트 규칙 (`addr << 1`)

1. 개요

STM32 HAL I²C 드라이버에서는 슬레이브 주소를 함수 인자로 전달할 때, **7-bit 주소를 왼쪽으로 1비트 시프트(`addr << 1`) 해야 한다.**

이 규칙은 **I²C 주소 프레임의 구조적 정의**와 HAL 내부 전송 루틴의 설계 방식에서 기인한다.

2. I²C 주소 프레임의 구조

I²C 버스에서 마스터는 Start 조건 이후 **8비트의 주소 프레임**을 전송한다.

이 8비트 프레임은 다음과 같이 구성된다.

```
1 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | R/W |
```

- 상위 7비트(A6~A0): 슬레이브의 주소.
- 최하위 1비트: R/W 비트.
 - 0 → 쓰기(Write)
 - 1 → 읽기(Read)

즉, 실제 버스 상에서 전송되는 주소는 항상 **8비트 단위이며, 마지막 비트는 R/W로 예약되어 있다.**

따라서 MCU 내부에서 7비트 주소만 알고 있다면, 실제 전송 시 **R/W 비트가 들어갈 자리 확보를 위해 1비트 왼쪽으로 시프트해야 한다.**

3. HAL 내부 구현 원리

STM32 HAL 드라이버(`stm32f1xx_hal_i2c.c`)는 하드웨어 레지스터에 전송 주소를 그대로 기록하지 않고, 사용자가 전달한 주소 값을 그대로 **8비트 프레임으로 간주한다.**

HAL 함수 예시:

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

- `DevAddress`: 전송할 대상 디바이스 주소 (8비트 프레임 형태로 기대)
- HAL 내부에서는 다음과 같이 처리된다:

```
1 | hi2c->Instance->DR = DevAddress & I2C_OAR1_ADD7_0; // 8비트 전송
```

- 여기서 하드웨어가 R/W 비트를 자동으로 붙이지 않는다.
→ 즉, HAL은 사용자가 이미 R/W 비트가 들어갈 자리를 확보한 주소(8비트 형태)를 넘겨줄 것을 요구한다.

4. 주소 시프트 규칙의 이유

7-bit 주소 체계에서 HAL이 요구하는 `addr << 1` 연산은 다음을 의미한다.

항목	설명
원래 주소	7-bit 슬레이브 주소 (예: 0x48)
시프트 후	0x90 (<code>0x48 << 1 = 0x90</code>)
최하위 비트	HAL 내부에서 R/W 비트로 채워짐 (<code>0x90</code> =Write, <code>0x91</code> =Read)

즉, `addr << 1` 은 "R/W 비트가 들어갈 공간을 확보한다"는 의미.

5. 실제 전송 예시

5.1 슬레이브 주소: 0x48

- HAL 함수 호출:

```
1 | HAL_I2C_Master_Transmit(&hi2c1, 0x48 << 1, tx_buf, 1, 100);
```

- 내부 전송 주소: 0x90 (0b10010000)
- 버스 상 실제 시퀀스:

```
1 | Start → 0x90 (Address + Write) → ACK → Data → Stop
```

5.2 동일 슬레이브, 읽기 동작:

```
1 | HAL_I2C_Master_Receive(&hi2c1, 0x48 << 1, rx_buf, 1, 100);
```

- 내부 전송 주소: 0x90 이지만, HAL 내부에서 읽기 명령 시 R/W=1로 전환되어 전송.
- 버스 상 실제 시퀀스:

```
1 | Start → 0x91 (Address + Read) → ACK → Data ← Slave → NACK → Stop
```

6. 흔한 실수

잘못된 코드	결과	원인
<code>HAL_I2C_Master_Transmit(&hi2c1, 0x90, buf, 10, 100);</code>	NACK 발생	이미 시프트된 주소(8비트)를 또 시프트하지 않아야 하는데, 중복됨

잘못된 코드	결과	원인
<code>HAL_I2C_Master_Transmit(&hi2c1, 0x48, buf, len, 100);</code>	NACK 발생	7-bit 주소 그대로 전달 → R/W 비트 자리 없음
<code>HAL_I2C_Master_Transmit(&hi2c1, 0x48 << 1, buf, len, 100);</code>	정상	HAL 규칙에 맞게 시프트 처리

7. HAL 내부 R/W 비트 처리 로직

HAL은 R/W 비트를 함수 종류로 구분한다:

- `HAL_I2C_Master_Transmit()` → R/W=0 (Write)
- `HAL_I2C_Master_Receive()` → R/W=1 (Read)
- `HAL_I2C_Mem_Read()` / `HAL_I2C_Mem_Write()` → 내부적으로 위 두 함수를 조합

즉, R/W는 사용자가 지정할 필요가 없으며,

HAL이 함수 종류에 따라 자동으로 최하위 비트를 0 또는 1로 세팅한다.

8. HAL의 10-bit 주소 모드 예외

10-bit 주소 모드(`hi2c.Init.AddressingMode = I2C_ADDRESSINGMODE_10BIT`)에서는

`addr << 1` 규칙이 일부 다르게 동작한다. HAL이 내부적으로 10-bit 전송 시퀀스(11110 + A9 + A8 + R/W 등)를 자동 구성하므로,

이 경우에는 주소 시프트를 사용자가 직접 수행하지 않는다.

즉:

- 7-bit 모드 → `addr << 1`
- 10-bit 모드 → `addr` 그대로

9. 정리

구분	7-bit 주소 모드	10-bit 주소 모드
HAL 전달 방식	<code>(addr << 1)</code>	<code>addr</code> 그대로
예시	<code>HAL_I2C_Master_Transmit(&hi2c1, 0x48 << 1, buf, len, 100);</code>	<code>HAL_I2C_Master_Transmit(&hi2c1, 0x2D3, buf, len, 100);</code>
내부 R/W 처리	함수별 자동	함수별 자동
이유	8비트 프레임의 R/W 비트 자리 확보	HAL이 내부적으로 10-bit 프레임 구성

10. 핵심 요약

- HAL은 8비트 주소 프레임을 인자로 기대한다.
- 7-bit 주소 모드에서는 `addr << 1` 해야 한다.
- R/W 비트는 HAL이 함수 종류에 따라 자동 세팅한다.
- 10-bit 모드에서는 시프트하지 않는다.
- 주소 프레임을 잘못 전달하면 즉시 NACK 오류 또는 통신 실패 발생.

• 타임아웃, 버스 정지 복구 기법

1. 개요

I²C 버스는 양방향 오픈드레인(open-drain) 구조를 가지며,

모든 라인은 풀업 저항을 통해 High 상태로 유지된다.

이 구조상, 한 디바이스라도 SCL 또는 SDA를 Low로 유지하면
전체 버스가 정지(Bus Hang)될 수 있다.

이 현상은 일반적으로 다음 상황에서 발생한다:

- 슬레이브가 ACK 응답 도중 전원 다운 또는 리셋됨.
- 마스터 전송 중 예외 발생으로 Stop 조건 미발생.
- 전기적 노이즈로 인한 SCL/SDA 라인 유지 오류.
- 클럭 스트레칭(Clock Stretching) 중 슬레이브 오동작.

따라서, 타임아웃 감지 및 버스 복구 절차는 실시간 시스템 안정성 확보를 위해 필수적으로 구현되어야 한다.

2. 타임아웃(Timeout) 감지 원리

2.1 HAL 레벨 기본 구조

STM32 HAL I²C 드라이버는 모든 전송 함수에 타임아웃 인자를 포함한다:

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

- `Timeout`은 밀리초 단위 시간 제한값이다.
- 내부적으로 HAL은 전송 중 상태 레지스터를 폴링(polling)하며,
지정 시간 내에 ACK, STOP, TXE 등의 이벤트가 발생하지 않으면
`HAL_TIMEOUT` 상태를 반환한다.

2.2 동작 과정 요약

1. 전송 시작
2. BUSY 플래그 확인
3. ADDR, TXE, BTF 등의 이벤트 대기

4. 지정 시간 경과 시 타임아웃 리턴

2.3 HAL 반환 상태 예시

반환 값	의미
HAL_OK	정상 전송 완료
HAL_ERROR	오류 발생 (예: NACK, Arbitration Lost 등)
HAL_BUSY	I ² C 하드웨어가 아직 점유 중
HAL_TIMEOUT	지정 시간 내 응답 없음

3. 타임아웃 발생 시 시스템 상태

타임아웃 이후 버스 라인은 다음 두 경우로 나뉜다:

상황	버스 상태
슬레이브 응답 없음, SDA High	버스 유휴(Idle), 재시도 가능
슬레이브 SDA Low 유지	버스 정지(Bus Stuck), 수동 복구 필요

4. 버스 정지(Bus Hang) 원인 분석

4.1 SDA Low 유지

- 슬레이브가 마지막 비트 전송 후 릴리스하지 못함.
- 클럭 스트레칭 중 슬레이브가 SCL을 계속 Low로 유지.
- 전원 불안정으로 슬레이브 내부 상태기계(State Machine) 고착.

4.2 MCU 측면 영향

- I²C_SR2.BUSY 비트가 1에서 해제되지 않음.
- HAL 함수가 BUSY 상태로 리턴하거나, 이후 통신 불가 상태 지속.
- Stop 신호가 유효하지 않아 I²C 주변장치가 재초기화되지 않음.

5. 버스 복구 절차 (Bus Recovery Sequence)

5.1 기본 개념

SDA가 Low에 고착된 경우, 마스터가 SCL을 수동으로 토글하여

슬레이브의 내부 시퀀스를 완결시킨다.

슬레이브가 마지막 비트를 정상적으로 인식하면 SDA를 해제하고,

이후 Stop 조건을 강제로 생성하여 버스를 유휴 상태로 되돌린다.

5.2 하드웨어 제어 절차

단계별 절차:

1. I²C 주변장치 비활성화

```
1 | __HAL_I2C_DISABLE(&hi2c1);
```

2. SCL, SDA 핀을 GPIO 출력 모드로 재구성

```
1 | GPIO_InitStruct.Pin = GPIO_PIN_6 | GPIO_PIN_7; // 예: PB6=SCL, PB7=SDA
2 | GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_OD;
3 | GPIO_InitStruct.Pull = GPIO_PULLUP;
4 | HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
```

3. SCL을 수동으로 9회 토글

```
1 | for (int i = 0; i < 9; i++) {
2 |     HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
3 |     HAL_Delay(1);
4 |     HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
5 |     HAL_Delay(1);
6 | }
```

💡 9회 이유: 슬레이브가 8비트 데이터 + 1 ACK를 인식할 기회를 제공.

4. Stop 조건 강제 생성

- SDA를 Low 유지한 상태에서 SCL High → SDA High 전환:

```
1 | HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_RESET);
2 | HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
3 | HAL_Delay(1);
4 | HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_SET);
```

5. SCL, SDA를 입력 모드로 복원

```
1 | GPIO_InitStruct.Mode = GPIO_MODE_AF_OD;
2 | HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
```

6. I²C 주변장치 재활성화 및 재초기화

```
1 | __HAL_I2C_ENABLE(&hi2c1);
2 | HAL_I2C_Init(&hi2c1);
```

6. 하드웨어 복구 루틴 예시 코드

```
1 void I2C_Bus_Recovery(void)
2 {
3     GPIO_InitTypeDef GPIO_InitStruct = {0};
4
5     // 1. I2C 비활성화
6     __HAL_I2C_DISABLE(&hi2c1);
7
8     // 2. GPIO 출력 설정
9     GPIO_InitStruct.Pin = GPIO_PIN_6 | GPIO_PIN_7;
10    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_OD;
11    GPIO_InitStruct.Pull = GPIO_PULLUP;
12    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
13
14    // 3. SCL 9회 토글
15    for (int i = 0; i < 9; i++) {
16        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
17        HAL_Delay(1);
18        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_RESET);
19        HAL_Delay(1);
20    }
21
22    // 4. Stop 조건 생성
23    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_RESET);
24    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);
25    HAL_Delay(1);
26    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_7, GPIO_PIN_SET);
27
28    // 5. I2C 모드 복원
29    GPIO_InitStruct.Mode = GPIO_MODE_AF_OD;
30    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
31
32    // 6. I2C 재초기화
33    __HAL_I2C_ENABLE(&hi2c1);
34    HAL_I2C_Init(&hi2c1);
35 }
```

7. HAL 타임아웃 후 자동 복구 절차 통합 예시

```
1 if (HAL_I2C_Master_Transmit(&hi2c1, addr << 1, data, len, 100) == HAL_TIMEOUT)
2 {
3     I2C_Bus_Recovery();
4     HAL_I2C_Master_Transmit(&hi2c1, addr << 1, data, len, 100);
5 }
```

8. 고급 복구 기법 (하드웨어 의존)

방법	설명
I ² C Software Reset (SWRST 비트)	I ² C_CR1.SWRST = 1 → 0 으로 내부 상태기계 초기화
Peripheral Reinit	HAL_I2C_DeInit() 후 HAL_I2C_Init() 수행
Watchdog 기반 복구	장시간 BUSY 유지 시 시스템 리셋 트리거
Clock Stretch Detection	SCL Low 유지 시간 측정 → 슬레이브 오류 감지

9. 타임아웃 및 복구 전략 요약

항목	권장 조치
통신 실패(NACK)	재시도 또는 오류 리턴
타임아웃 발생	Bus Recovery 수행
BUSY 지속	I ² C 리셋 및 재초기화
반복적 오류	디바이스 전원 리셋 또는 하드웨어 리커버리 절차 실행

10. 결론

- I²C 통신은 타임아웃 및 버스 정지에 취약하므로,
소프트웨어 레벨 타임아웃 감시와 하드웨어 레벨 복구 시퀀스를 반드시 병행해야 한다.
- STM32 HAL은 기본 타임아웃 감지는 제공하지만, 버스 복구는 사용자가 직접 구현해야 한다.
- 실제 산업용 시스템에서는 9클럭 토클 + 강제 Stop + 주변장치 리셋 조합이 가장 신뢰성이 높다.

5.2 HAL I²C 함수

• HAL_I2C_Master_Transmit()

1. 개요

HAL_I2C_Master_Transmit() 함수는 I²C 마스터(Master)가 특정 슬레이브(Slave) 디바이스로 데이터를 전송하기 위한 HAL 계층의 표준 API이다.

이 함수는 Polling 방식으로 동작하며, 전송 완료 또는 타임아웃이 발생할 때까지 CPU가 전송 상태를 지속적으로 확인한다.

2. 함수 원형

```
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 );
```

3. 매개변수 설명

매개변수	자료형	설명
hi2c	I2C_HandleTypeDef*	I ² C 주변장치를 나타내는 핸들 구조체 포인터. CubeMX 초기화 시 자동 생성된다.
DevAddress	uint16_t	전송 대상 슬레이브 주소. 7비트 주소는 좌측으로 1비트 시프트 (<code>addr << 1</code>) 해야 한다.
pData	uint8_t*	전송할 데이터 버퍼의 시작 주소.
Size	uint16_t	전송할 데이터의 바이트 수.
Timeout	uint32_t	최대 대기 시간(밀리초). 지정 시간 내 완료되지 않으면 <code>HAL_TIMEOUT</code> 반환.

4. 반환값

반환 값	의미
HAL_OK	정상적으로 전송 완료
HAL_ERROR	오류 발생 (NACK, Arbitration Lost 등)
HAL_BUSY	I ² C 하드웨어가 점유 중
HAL_TIMEOUT	지정 시간 내 완료되지 않음

5. 내부 동작 순서

함수 내부는 다음 순서로 진행된다.

1. 상태 확인

- `hi2c->State` 가 `HAL_I2C_STATE_READY` 인지 검사.
- BUSY 상태이면 즉시 `HAL_BUSY` 리턴.

2. Bus 상태 확인

- I2C_SR2.BUSY 플래그 확인.
- BUSY 상태 시 하드웨어 점유로 간주.

3. Start 조건 생성

- I2C_CR1.START 비트를 세트.
- 하드웨어가 Start Condition(S) 생성 후 SB (Start Bit) 플래그 세트.

4. 슬레이브 주소 전송

- I2C_DR에 (DevAddress & 0xFE) 전송.
- 7비트 주소의 경우 (addr << 1).
- Write 모드 → 마지막 비트 0.

5. ADDR 플래그 대기

- 슬레이브 ACK 수신 시 ADDR 플래그 세트.
- 응답이 없을 경우 타임아웃 → HAL_TIMEOUT.

6. 데이터 전송 루프

- TXE (Transmit Empty) 플래그 대기 후 바이트 전송.
- 내부 루프에서 BTF (Byte Transfer Finished) 확인.
- 모든 데이터 전송 완료 시 종료 절차로 이동.

7. Stop 조건 생성

- I2C_CR1.STOP 비트를 세트.
- Stop Condition(P) 출력 후 BUSY 비트 클리어.

8. 상태 복구 및 반환

- 내부 상태를 HAL_I2C_STATE_READY로 복원.
- HAL_OK 반환.

6. 동작 타이밍 다이어그램

1	S Addr+W ACK Data1 ACK Data2 ACK ... DataN NACK P
2	<----- Transmit Sequence ----->

- **S** : Start condition
- **Addr+W** : 슬레이브 주소(Write 모드)
- **ACK/NACK** : 슬레이브 응답
- **P** : Stop condition

7. 사용 예제

```
1 uint8_t data[2] = {0x01, 0x7F}; // 전송할 데이터
2 uint16_t dev_addr = 0x48; // 7-bit 주소
3
4 if (HAL_I2C_Master_Transmit(&hi2c1, dev_addr << 1, data, 2, 100) != HAL_OK)
5 {
6     // 오류 처리 루틴
7     Error_Handler();
8 }
```

8. 내부 레지스터 연동 요약

동작 단계	주요 레지스터	설정 비트
Start 생성	I2C_CR1	START
주소 전송	I2C_DR	주소 바이트 기록
ACK 감지	I2C_SR1	ADDR
데이터 전송	I2C_DR, I2C_SR1	TXE, BTF
Stop 생성	I2C_CR1	STOP
상태 확인	I2C_SR2	BUSY

9. 예외 및 주의사항

- 슬레이브 응답 없음(NACK) 시 `HAL_ERROR` 발생.
- BUSY 상태 지속 시 이전 통신이 완전히 종료되지 않은 것.
- 클럭 스트레칭이 긴 디바이스는 타임아웃 값을 충분히 크게 설정해야 함.
- 멀티마스터 환경에서는 Arbitration Lost(ARLO) 플래그 발생 시 즉시 재시도 필요.

10. 확장 응용

- Non-blocking 전송:**

`HAL_I2C_Master_Transmit_IT()` (인터럽트 기반)

- DMA 전송:**

`HAL_I2C_Master_Transmit_DMA()` (고속 데이터 송신용)

- 메모리 접근 전용 함수:**

`HAL_I2C_Mem_Write()` — 내부 레지스터 접근 시 사용 (예: 센서 설정 레지스터).

11. 결론

`HAL_I2C_Master_Transmit()`은 STM32 HAL 계층에서 가장 기본적인 I²C 송신 루틴으로, **Polling 기반 단발성 송신**에 적합하다.

복수의 I²C 트랜잭션이 동시에 수행되는 멀티태스킹 환경에서는 FreeRTOS Mutex와 타임아웃 처리를 결합하여 **버스 충돌 방지 및 재시도 로직**을 반드시 함께 설계해야 한다.

• `HAL_I2C_Master_Receive()`

1. 개요

`HAL_I2C_Master_Receive()` 함수는 **I²C 마스터(Master)**가 지정한 **슬레이브(Slave)**로부터 데이터를 **Polling 방식으로 수신(Read)** 하는 HAL 계층 표준 API이다.

이 함수는 마스터가 **Start → Address + Read → Data → Stop** 순으로 프레임을 생성하며, CPU가 직접 플래그 상태를 모니터링하여 모든 수신 과정이 완료될 때까지 대기한다.

2. 함수 원형

```
1 HAL_StatusTypeDef HAL_I2C_Master_Receive(
2     I2C_HandleTypeDef *hi2c,
3     uint16_t DevAddress,
4     uint8_t *pData,
5     uint16_t Size,
6     uint32_t Timeout
7 );
```

3. 매개변수 설명

매개변수	자료형	설명
<code>hi2c</code>	<code>I2C_HandleTypeDef*</code>	I ² C 주변장치 핸들 구조체 포인터. CubeMX 생성 시 자동 정의됨.
<code>DevAddress</code>	<code>uint16_t</code>	슬레이브 주소. 7-bit 주소는 반드시 좌측 시프트(addr << 1) 해야 함.
<code>pData</code>	<code>uint8_t*</code>	수신 데이터를 저장할 버퍼의 시작 주소.
<code>Size</code>	<code>uint16_t</code>	수신할 데이터의 바이트 수.
<code>Timeout</code>	<code>uint32_t</code>	타임아웃 시간(밀리초). 응답 지연 시 <code>HAL_TIMEOUT</code> 반환.

4. 반환값

반환 값	의미
HAL_OK	정상적으로 수신 완료
HAL_ERROR	오류 발생 (NACK, ARLO 등)
HAL_BUSY	I ² C 주변장치가 점유 중
HAL_TIMEOUT	지정된 시간 내 응답 없음

5. 내부 동작 순서

1. 상태 확인

- `hi2c->State` 가 `HAL_I2C_STATE_READY` 인지 검사.
- BUSY 상태일 경우 즉시 `HAL_BUSY` 리턴.

2. Start 조건 생성

- `I2C_CR1.START` 비트를 세트하여 Start Condition 생성.
- SB (Start Bit) 플래그 세트 시 다음 단계로 진행.

3. 슬레이브 주소 전송 (Read 모드)

- `I2C_DR`에 `(DevAddress << 1) | 1`을 기록.
→ LSB = 1 → Read 명령.
- 슬레이브가 ACK 시 `ADDR` 플래그 세트.

4. ADDR 클리어 및 수신 준비

- `I2C_SR1`, `I2C_SR2`를 읽어 ADDR 플래그 클리어.
- 이후 수신 루프 진입.

5. 데이터 수신 루프

- `RXNE` (Receive Buffer Not Empty) 플래그 감시.
- 각 바이트 수신 시 `I2C_DR`에서 읽어 `pData[]`에 저장.
- 마지막 바이트 직전에 NACK 및 STOP 조건 제어.

6. 마지막 바이트 처리

- Size=1일 경우:
 - `ACK` 비트 클리어 → `STOP` 비트 세트 → 데이터 읽기.
- Size>1일 경우:
 - N-1번째 바이트까지 ACK 유지, 마지막 1바이트 전 `ACK` 클리어.

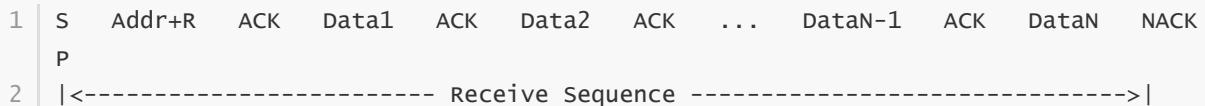
7. Stop 조건 생성 및 종료

- `I2C_CR1.STOP` 비트 세트 → Stop Condition 출력.
- `BUSY` 플래그 해제 후 완료.

8. 상태 복구 및 반환

- 내부 상태를 `HAL_I2C_STATE_READY`로 복귀.
- 결과값 `HAL_OK` 반환.

6. 수신 타이밍 다이어그램



- **S** : Start condition
- **Addr+R** : 슬레이브 주소 + Read 비트
- **ACK/NACK** : 마스터가 슬레이브에 전송
- **P** : Stop condition

주의: 마지막 바이트 수신 후 마스터는 **NACK** → **Stop** 시퀀스를 반드시 수행해야 한다.
슬레이브는 NACK를 수신하면 더 이상 데이터를 전송하지 않는다.

7. 예제 코드

```

1 | uint8_t recv_data[4];
2 | uint16_t dev_addr = 0x48; // 예: 온도센서
3 |
4 | if (HAL_I2C_Master_Receive(&hi2c1, dev_addr << 1, recv_data, 4, 100) != HAL_OK)
5 | {
6 |     // 오류 처리
7 |     Error_Handler();
8 | }
  
```

8. 내부 레지스터 흐름 요약

단계	레지스터	관련 비트	설명
Start 생성	<code>I2C_CR1</code>	<code>START</code>	Start condition 출력
주소 전송	<code>I2C_DR</code>	—	슬레이브 주소 + Read 비트
ACK 제어	<code>I2C_CR1</code>	<code>ACK</code>	수신 바이트마다 ACK/NACK 전송
수신 상태	<code>I2C_SR1</code>	<code>RXNE</code>	수신 버퍼에 데이터 존재
데이터 읽기	<code>I2C_DR</code>	—	수신 바이트 읽기
Stop 생성	<code>I2C_CR1</code>	<code>STOP</code>	통신 종료 신호

9. Size 별 내부 제어 로직

(1) 1바이트 수신

- Start → Addr+R → NACK → Stop → Read

(2) 2바이트 수신

- ACK 유지 후 마지막 2바이트 전 NACK 설정
- Stop 출력 후 마지막 데이터 읽기

(3) 다중 바이트 수신 (≥ 3)

- 모든 바이트에 대해 ACK 응답
- 마지막 2바이트 전 NACK 준비
- Stop 출력 후 종료

10. 예외 처리 및 타임아웃

상황	처리
슬레이브 응답 없음 (NACK)	즉시 HAL_ERROR 반환
BUSY 상태 유지	이전 전송 미완료 → 복구 또는 재시도 필요
RXNE 미세트 (지연 수신)	지정 타임아웃 내 대기
클럭 스트레칭	SCL Low 유지로 지연될 수 있으므로 타임아웃 여유 필요

11. 확장 응용

함수	설명
HAL_I2C_Master_Receive_IT()	인터럽트 기반 비동기 수신
HAL_I2C_Master_Receive_DMA()	DMA 기반 대용량 수신
HAL_I2C_Mem_Read()	슬레이브 내부 메모리 주소 지정 후 수신 (예: 센서 레지스터)

12. 응용 예 — 센서 레지스터 읽기

```
1 uint8_t reg = 0x01;
2 uint8_t val;
3
4 HAL_I2C_Master_Transmit(&hi2c1, 0x48 << 1, &reg, 1, 100);
5 HAL_I2C_Master_Receive(&hi2c1, 0x48 << 1, &val, 1, 100);
```

일반적으로 I²C 센서의 레지스터 접근은 위와 같이
"쓰기 후 읽기" 형태로 구성된다.

13. 결론

`HAL_I2C_Master_Receive()`은 마스터-슬레이브 구조의 기본 Read 트랜잭션을 수행하며,
Polling 기반으로 안정적이지만, CPU 점유율이 높다.

FreeRTOS 기반 시스템에서는

- 인터럽트 기반(`HAL_I2C_Master_Receive_IT()`)
 - 또는 DMA 기반(`HAL_I2C_Master_Receive_DMA()`)
- 방식으로 전환하여 병렬성을 확보하는 것이 바람직하다.

• `HAL_I2C_Mem_Read()`, `HAL_I2C_Mem_Write()`

1. 개요

`HAL_I2C_Mem_Read()`은 I²C 마스터가 특정 슬레이브 디바이스의 내부 메모리 주소(Register Address)로부터
데이터를 읽기(Read) 위해 사용하는 HAL 계층의 고수준 API이다.

이 함수는 다음과 같은 두 단계 트랜잭션을 자동으로 수행한다.

1. Write 단계 — 내부 주소 전송 (Memory Address)
2. Read 단계 — 해당 주소에서 데이터 수신

2. 함수 원형

```
1 HAL_StatusTypeDef HAL_I2C_Mem_Read(
2     I2C_HandleTypeDef *hi2c,
3     uint16_t DevAddress,
4     uint16_t MemAddress,
5     uint16_t MemAddSize,
6     uint8_t *pData,
7     uint16_t Size,
8     uint32_t Timeout
9 );
```

3. 매개변수 설명

매개변수	자료형	설명
<code>hi2c</code>	<code>I2C_HandleTypeDef*</code>	I ² C 핸들 구조체 포인터
<code>DevAddress</code>	<code>uint16_t</code>	슬레이브 주소 (addr << 1 규칙 동일)
<code>MemAddress</code>	<code>uint16_t</code>	슬레이브 내부 메모리(또는 레지스터) 주소

매개변수	자료형	설명
<code>MemAddSize</code>	<code>uint16_t</code>	내부 주소 크기 (I2C_MEMADD_SIZE_8BIT 또는 I2C_MEMADD_SIZE_16BIT)
<code>pData</code>	<code>uint8_t*</code>	수신 데이터 버퍼 포인터
<code>size</code>	<code>uint16_t</code>	읽을 바이트 수
<code>Timeout</code>	<code>uint32_t</code>	타임아웃(ms 단위)

4. 반환값

값	의미
<code>HAL_OK</code>	정상 수신 완료
<code>HAL_ERROR</code>	오류 발생
<code>HAL_BUSY</code>	주변장치 점유 중
<code>HAL_TIMEOUT</code>	지정 시간 초과

5. 내부 동작 순서

1. 상태 확인

`hi2c->State` 확인 → READY 상태인지 검사

2. 1차 Write 프레임 (주소 전송)

- Start 조건 생성
- `DevAddress` + Write 비트 전송
- ACK 수신 후 내부 주소(`MemAddress`) 전송
- `MemAddSize`에 따라 1바이트 또는 2바이트로 분할 전송

3. Repeated Start 생성

- Stop 없이 Re-Start 생성 ($SR = 1$)
- 슬레이브 주소 + Read 비트 전송

4. 데이터 수신 루프

- `size` 만큼 바이트 수신 (`RXNE` 감시)
- 마지막 바이트 전 NACK 및 STOP 제어

5. 종료 처리

- `STOP` 조건 출력
- 상태를 READY로 복귀

6. 트랜잭션 타이밍 다이어그램

```
1 | S Addr(w) ACK MemAddr ACK Sr Addr(R) ACK Data1 ACK Data2 ACK ... DataN NACK P
```

- S : Start
- Sr : Repeated Start
- P : Stop

7. 사용 예

```
1 | uint8_t data[2];
2 | uint16_t dev_addr = 0x48;      // 예: 온도센서
3 | uint8_t reg_addr = 0x01;
4 |
5 | if (HAL_I2C_Mem_Read(&hi2c1, dev_addr << 1, reg_addr,
6 |                         I2C_MEMADD_SIZE_8BIT, data, 2, 100) != HAL_OK)
7 | {
8 |     Error_Handler();
9 | }
```

8. 내부 제어 포인트

단계	주요 레지스터	비트	설명
Start	I2C_CR1	START	첫 Start 생성
내부 주소 전송	I2C_DR	—	MemAddress 전송
Repeated Start	I2C_CR1	START	Read 시퀀스 재시작
수신 상태	I2C_SR1	RXNE	수신 버퍼 데이터 존재
Stop	I2C_CR1	STOP	통신 종료

9. 예외 및 주의사항

- `MemAddsize` 가 올바르지 않으면 슬레이브가 ACK를 보내지 않음.
- 일부 EEPROM은 Write 후 내부 Write-Cycle 시간(수 ms) 필요 → Read 즉시 시도 금지.
- Repeated Start를 지원하지 않는 장치에서는 사용 불가.

HAL 함수 — `HAL_I2C_Mem_Write()`

1. 개요

`HAL_I2C_Mem_Write()` 는 I²C 마스터가 슬레이브 내부의 특정 주소로 데이터 쓰기(Write)를 수행하는 고수준 HAL API이다.

EEPROM, RTC, DAC 등 내부 레지스터에 설정 값을 기록할 때 사용된다.

2. 함수 원형

```
1 HAL_StatusTypeDef HAL_I2C_Mem_Write(
2     I2C_HandleTypeDef *hi2c,
3     uint16_t DevAddress,
4     uint16_t MemAddress,
5     uint16_t MemAddSize,
6     uint8_t *pData,
7     uint16_t Size,
8     uint32_t Timeout
9 );
```

3. 매개변수

매개변수	자료형	설명
hi2c	<code>I2C_HandleTypeDef*</code>	I ² C 핸들 구조체
DevAddress	<code>uint16_t</code>	슬레이브 주소 (<code>addr << 1</code>)
MemAddress	<code>uint16_t</code>	내부 레지스터 또는 메모리 주소
MemAddSize	<code>uint16_t</code>	주소 크기 (8/16비트)
pData	<code>uint8_t*</code>	송신 데이터 버퍼
Size	<code>uint16_t</code>	송신 바이트 수
Timeout	<code>uint32_t</code>	타임아웃 시간

4. 내부 동작 순서

1. 상태 확인

READY 상태 확인

2. Start 조건 생성

- `DevAddress + Write` 전송
- 슬레이브 ACK 수신

3. 내부 주소 전송

- `MemAddSize` 크기에 따라
1 또는 2바이트 전송 (MSB → LSB)

4. 데이터 바이트 전송 루프

- TXE, BTF 플래그 감시
- 각 데이터 바이트 I2C_DR에 기록

5. Stop 조건 출력

- I2C_CR1.STOP 세트
- BUSY 플래그 해제 후 완료

5. 타이밍 다이어그램

```
1 | S Addr(W) ACK MemAddr ACK Data1 ACK Data2 ACK ... DataN ACK P
```

6. 사용 예

```
1 | uint8_t config[2] = {0x01, 0x60};  
2 | uint16_t dev_addr = 0x48; // 예: 센서  
3 |  
4 | if (HAL_I2C_Mem_Write(&hi2c1, dev_addr << 1, 0x02,  
5 |                 I2C_MEMADD_SIZE_8BIT, config, 2, 100) != HAL_OK)  
6 | {  
7 |     Error_Handler();  
8 | }
```

7. 주의사항

- 일부 EEPROM은 Page Write 제한 존재 (예: 16바이트 단위).
페이지 경계를 넘는 쓰기는 자동으로 Wrap-around됨.
- Write 후 내부 Write-Cycle 완료까지 추가 대기 필요 (보통 5~10ms).
- BUSY 상태나 NACK 발생 시 재시도 루틴 권장.

8. 반환 상태

반환 값	의미
HAL_OK	정상 완료
HAL_ERROR	NACK 등 오류 발생
HAL_BUSY	장치 점유 중
HAL_TIMEOUT	시간 초과

9. 확장 함수

함수	설명
HAL_I2C_Mem_Read_IT()	인터럽트 기반 Read
HAL_I2C_Mem_Write_IT()	인터럽트 기반 Write
HAL_I2C_Mem_Read_DMA()	DMA 기반 Read
HAL_I2C_Mem_Write_DMA()	DMA 기반 Write

10. 결론

이 두 함수는 단순한 `Transmit` / `Receive` 함수보다 상위 계층이며,
“주소 기반 I²C 디바이스”(EEPROM, 센서, RTC, ADC 등)의 표준 접근 인터페이스이다.

STM32 HAL은 내부적으로

`HAL_I2C_Master_Transmit()` + `HAL_I2C_Master_Receive()` 의
시퀀스를 자동 조합하여 “쓰기 후 읽기(Read from Memory)” 트랜잭션을 완전하게 구현한다.

5.3 실습

• 24C02 EEPROM 읽기/쓰기

1. 개요

24C02는 **2Kbit(=256Byte)** 용량의 **シリアル EEPROM**으로,
I²C(Inter-Integrated Circuit) 프로토콜을 통해 데이터를 저장/읽기한다.
비휘발성(Non-Volatile) 메모리로 전원이 제거되어도 데이터가 유지된다.
EEPROM 내부는 **8비트 주소 공간 (0x00 ~ 0xFF)** 으로 구성되어 있으며,
페이지 단위(보통 8~16Byte)로 쓰기가 가능하다.
통신은 **SDA (Serial Data)** 와 **SCL (Serial Clock)** 두 선만으로 수행되며,
각 디바이스는 하드웨어 주소 비트(A0, A1, A2)에 의해 구분된다.

2. 하드웨어 연결 구조

핀명	설명	연결 예시
VCC	전원 (1.8~5.5V)	+3.3V
GND	접지	GND
SDA	シリアル 데이터	STM32 PB7 (I ² C1 SDA)
SCL	シリ얼 클록	STM32 PB6 (I ² C1 SCL)
WP	Write Protect	GND (비활성화)

핀명	설명	연결 예시
A0~A2	디바이스 주소 설정	GND (모두 0으로 설정)

3. 디바이스 주소 (Slave Address)

24C02의 7비트 기본 주소는 `0b1010xxx` 형태이다.

여기서 `xxx`는 A2~A0 핀에 연결된 값으로 결정된다.

예를 들어 A0=A1=A2=GND인 경우:

```
1 | Device Address = 0b1010000 = 0x50
```

HAL 함수 호출 시에는 **왼쪽으로 1비트 시프트**(`addr << 1`)하여

R/W 비트를 포함한 8비트 주소로 전송한다.

즉:

```
1 | uint8_t devAddr = 0x50 << 1; // HAL 함수용 주소
```

4. 내부 메모리 구조

주소 범위	설명
0x00 ~ 0xFF	256 Byte EEPROM 저장 공간
페이지 크기	8 Byte (Page Write 단위)

Page Write 시 주의사항:

한 번의 Write 명령으로 8바이트를 초과하면, 페이지 경계 이후의 데이터는 해당 페이지의 시작 주소로 래핑되어 덮어쓴다.

5. 쓰기 동작 (Write Operation)

5.1 순차 개요

1. Start 조건 생성
2. Slave Address + Write 비트 전송
3. 내부 주소(1Byte) 전송
4. 데이터 바이트 전송 (최대 8바이트)
5. Stop 조건 생성
6. 내부 Write Cycle 진행 (5~10ms 소요)

5.2 HAL 코드 예시

```
1 #include "stm32f1xx_hal.h"
2
3 extern I2C_HandleTypeDef hi2c1; // I2C1 핸들 선언
4
5 #define EEPROM_ADDR (0x50 << 1)
6
7 void EEPROM_Write(uint16_t memAddr, uint8_t *pData, uint16_t size)
8 {
9     if (HAL_I2C_Mem_Write(&hi2c1, EEPROM_ADDR, memAddr,
10                     I2C_MEMADD_SIZE_8BIT, pData, size, 100) != HAL_OK)
11     {
12         Error_Handler();
13     }
14
15     HAL_Delay(10); // 내부 Write Cycle (T_WR) 대기
16 }
```

5.3 예시 사용

```
1 uint8_t txBuf[8] = {'H', 'E', 'L', 'L', 'O', '!', 0x00, 0x00};
2 EEPROM_Write(0x00, txBuf, 8);
```

이 코드는 0x00 번지부터 “HELLO!” 데이터를 기록한다.

쓰기 완료 후 약 5~10ms 동안 EEPROM이 내부 쓰기를 수행하므로,

즉시 다음 Write 명령을 보내면 NACK이 발생할 수 있다.

6. 읽기 동작 (Read Operation)

6.1 순차 개요

1. Start 조건 생성
 2. Slave Address + Write 비트 전송
 3. 내부 주소(1Byte) 전송
 4. Repeated Start 조건 생성
 5. Slave Address + Read 비트 전송
 6. 데이터 바이트 수신
 7. Stop 조건 생성
-

6.2 HAL 코드 예시

```
1 void EEPROM_Read(uint16_t memAddr, uint8_t *pData, uint16_t size)
2 {
3     if (HAL_I2C_Mem_Read(&hi2c1, EEPROM_ADDR, memAddr,
4                         I2C_MEMADD_SIZE_8BIT, pData, size, 100) != HAL_OK)
5     {
6         Error_Handler();
7     }
8 }
```

6.3 예시 사용

```
1 uint8_t rxBuf[8] = {0};
2
3 EEPROM_Read(0x00, rxBuf, 8);
4 printf("EEPROM: %s\n", rxBuf);
```

출력:

```
1 | EEPROM: HELLO!
```

7. 내부 Write Cycle 관리

24C02는 데이터가 EEPROM 셀에 실제로 기록될 때까지 **Write Cycle Time (tWR)** 동안 버스를 점유하며 NACK을 반환한다.

이를 감지하여 **Write 완료를 풀링(Polling)** 하는 방식이 일반적이다.

예시 코드

```
1 void EEPROM_WaitForwrite(void)
2 {
3     while (HAL_I2C_IsDeviceReady(&hi2c1, EEPROM_ADDR, 10, 100) != HAL_OK)
4     {
5         HAL_Delay(1);
6     }
7 }
```

`HAL_I2C_IsDeviceReady()` 함수는 내부적으로 SLA+W 전송을 시도하고 ACK 응답을 확인하여, 디바이스가 Write Cycle을 마칠 때까지 반복 대기한다.

8. 전체 예제

```
1 void EEPROM_Test(void)
2 {
3     uint8_t txData[] = "STM32 EEPROM TEST";
4     uint8_t rxData[32] = {0};
5
6     EEPROM_Write(0x10, txData, sizeof(txData));
7     EEPROM_WaitForwrite();
8
9     EEPROM_Read(0x10, rxData, sizeof(txData));
10
11    printf("Read Data: %s\n", rxData);
12 }
```

9. 다중 EEPROM 구성 (Address Bit 활용)

여러 개의 24C02를 병렬로 연결할 경우,
A0~A2 핀을 개별적으로 조합하여 최대 8개(0x50~0x57) 장치를 구분할 수 있다.

A2	A1	A0	주소(7bit)	실제 HAL 주소
0	0	0	0x50	0xA0
0	0	1	0x51	0xA2
0	1	0	0x52	0xA4
...

10. 참고 사항

- Page Write 시 `memAddr` 가 페이지 경계를 넘으면 래핑 발생.
- Write Protect(WP) 핀이 HIGH이면 쓰기 불가.
- I²C 속도는 일반적으로 100kHz(Standard Mode) 권장.
- 고속 모드(400kHz) 사용 시, Pull-up 저항(4.7kΩ 이하) 조정 필요.

11. 타이밍 다이어그램

```
1 Write Cycle:
2 S | DevAddr(W) | ACK | MemAddr | ACK | Data... | ACK | P
3 Read Cycle:
4 S | DevAddr(W) | ACK | MemAddr | ACK | Sr | DevAddr(R) | ACK | Data | NACK | P
```

12. 결론

- `HAL_I2C_Mem_Write()` → 내부 주소 지정 후 데이터 전송
- `HAL_I2C_Mem_Read()` → 내부 주소 지정 후 Re-Start로 데이터 수신
- `HAL_I2C_IsDeviceReady()` → Write Cycle 완료 대기

이 세 가지 함수 조합만으로
STM32에서 24C02 EEPROM을 완전하게 제어할 수 있다.

• OLED SSD1306 “Hello” 출력

1. 개요

SSD1306은 128×64 또는 128×32 해상도를 지원하는 모노크롬 OLED 디스플레이 드라이버 IC로,
I²C, SPI, Parallel 등 여러 인터페이스를 제공한다.
STM32 환경에서는 주로 I²C(400kHz) 모드를 사용한다.

디스플레이는 내부 **GRAM (Graphic RAM)**을 보유하고 있으며,
MCU는 이 RAM에 데이터를 전송하여 픽셀을 제어한다.

2. 하드웨어 연결 구조 (I²C형 모듈 기준)

OLED 핀	기능	STM32 연결 예시
VCC	전원 (3.3V 또는 5V)	3.3V
GND	접지	GND
SCL	I ² C 클록 입력	PB6 (I ² C1_SCL)
SDA	I ² C 데이터 입력	PB7 (I ² C1_SDA)

⚠ 일부 모듈은 **0x3C** 또는 **0x3D** 주소를 사용한다.
(SA0 핀 연결 상태에 따라 달라짐)

3. 디바이스 주소

- **7-bit Address** : `0x3C` (일반적)
- HAL 호출 시 : `0x3C << 1 = 0x78`

```
1 | #define SSD1306_ADDR (0x3C << 1)
```

4. 명령과 데이터 전송 규칙

SSD1306은 **Control Byte**로 명령과 데이터를 구분한다.

Control Byte	의미
0x00	명령(Command)
0x40	데이터(Data)

전송 포맷:

```

1 | S | DevAddr(W) | 0x00 (Control) | Command Byte | P
2 | S | DevAddr(W) | 0x40 (Control) | Display Data | P

```

5. 초기화 시퀀스 (Initialization Sequence)

SSD1306은 전원 인가 후 내부 레지스터를 설정해야 한다.

아래는 일반적인 128×64 모듈 기준 초기화 명령 세트이다.

```

1 | static const uint8_t SSD1306_InitCmd[] = {
2 |     0xAE, // Display OFF
3 |     0x20, 0x00, // Memory addressing mode: Horizontal
4 |     0xB0, // Page Start Address
5 |     0xC8, // COM Output Scan Direction: remapped
6 |     0x00, // Low column start
7 |     0x10, // High column start
8 |     0x40, // Start line address
9 |     0x81, 0x7F, // Contrast
10 |    0xA1, // Segment remap
11 |    0xA6, // Normal display
12 |    0xA8, 0x3F, // Multiplex ratio (1/64)
13 |    0xA4, // Display follows RAM content
14 |    0xD3, 0x00, // Display offset
15 |    0xD5, 0x80, // Display clock divide ratio
16 |    0xD9, 0xF1, // Pre-charge period
17 |    0xDA, 0x12, // COM pins config
18 |    0xDB, 0x40, // VCOMH deselect level
19 |    0x8D, 0x14, // Charge pump enable
20 |    0xAF // Display ON
21 | };

```

6. 명령 전송 함수

```

1 | void SSD1306_SendCommand(uint8_t cmd)
2 | {
3 |     uint8_t control[2] = {0x00, cmd};
4 |     HAL_I2C_Master_Transmit(&hi2c1, SSD1306_ADDR, control, 2, HAL_MAX_DELAY);
5 | }

```

7. 초기화 함수

```
1 void SSD1306_Init(void)
2 {
3     HAL_Delay(100); // 전원 안정 대기
4
5     for (uint8_t i = 0; i < sizeof(SSD1306_InitCmd); i++)
6     {
7         SSD1306_SendCommand(SSD1306_InitCmd[i]);
8     }
9 }
```

8. 문자 출력 개념

SSD1306은 픽셀 단위 디스플레이로, 문자를 표시하기 위해서는 **폰트 데이터 배열(Font Table)**을 RAM으로 전송해야 한다.

예시 — 5x7 ASCII 폰트 일부:

```
1 static const uint8_t Font5x7[][5] = {
2     // ' ' ~ '~'
3     {0x00,0x00,0x00,0x00,0x00}, // Space (0x20)
4     {0x00,0x00,0x5F,0x00,0x00}, // !
5     {0x00,0x07,0x00,0x07,0x00}, // "
6     ...
7     {0x7C,0x12,0x11,0x12,0x7C}, // A
8     {0x7F,0x49,0x49,0x49,0x36}, // B
9 };
```

9. 문자 전송 함수

```
1 void SSD1306_SendData(uint8_t *data, uint16_t size)
2 {
3     uint8_t buffer[size + 1];
4     buffer[0] = 0x40; // control byte: Data
5     memcpy(&buffer[1], data, size);
6     HAL_I2C_Master_Transmit(&hi2c1, SSD1306_ADDR, buffer, size + 1, HAL_MAX_DELAY);
7 }
8
9 void SSD1306_DrawChar(char c)
10 {
11     if (c < 32 || c > 126) c = ' ';
12     SSD1306_SendData((uint8_t*)Font5x7[c - 32], 5);
13     uint8_t space = 0x00;
14     SSD1306_SendData(&space, 1); // 글자 간격
15 }
```

10. 커서 위치 설정

디스플레이는 페이지(Page) 단위로 구성된다.

한 페이지는 세로 8픽셀, 가로 128픽셀이다.

```
1 void SSD1306_SetCursor(uint8_t x, uint8_t y)
2 {
3     SSD1306_SendCommand(0xB0 + y);           // Page address
4     SSD1306_SendCommand(0x00 + (x & 0x0F)); // Low column
5     SSD1306_SendCommand(0x10 + ((x >> 4) & 0x0F)); // High column
6 }
```

11. “Hello” 출력 예제

```
1 void SSD1306_PrintHello(void)
2 {
3     SSD1306_Init();
4     SSD1306_SetCursor(0, 0);
5
6     char *str = "Hello";
7     while (*str)
8     {
9         SSD1306_DrawChar(*str++);
10    }
11 }
```

결과:

디스플레이 첫 줄 좌측 상단에 “Hello” 출력됨.

12. 전체 시퀀스 요약

단계	내용
1	I ² C1 초기화
2	SSD1306 초기화 명령 전송
3	페이지/컬럼 위치 설정
4	문자 데이터 송신
5	화면 표시

13. 디버깅 포인트

증상	원인	해결
화면에 아무 것도 표시되지 않음	전원 또는 주소 불일치	0x3C ↔ 0x3D 확인
깨진 문자	폰트 인덱스 범위 오류	c - 32 보정
깜박임	빠른 화면 간접	DMA 전송 고려
일부 줄 안보임	페이지 주소 설정 오류	0xB0 + y 확인

14. 확장 기능

- SSD1306_Clear() : 전체 0x00으로 채워 화면 지우기
- SSD1306_DrawString() : 문자열 전체 출력
- SSD1306_DrawBitmap() : 이미지 렌더링
- SSD1306_UpdateScreen() : 버퍼 기반 프레임 전송

15. 결론

- HAL_I2C_Master_Transmit() 기반의 단순 구조로 동작
- 제어 바이트(0x00/0x40)를 구분하여 명령/데이터 송신
- “Hello” 출력은 최소한의 폰트 및 위치 제어만으로 가능

STM32 + SSD1306은 I²C 실습 및 임베디드 UI 구축의 기본 예제로,
FreeRTOS 환경에서는 DisplayTask로 분리하여 주기적 업데이트를 수행할 수 있다.

• VL53L0X 거리센서 데이터 읽기

1. 개요

VL53L0X는 STMicroelectronics에서 개발한 **ToF(Time-of-Flight)** 방식의
레이저 거리 센서로, 적외선 펄스를 발사한 뒤 반사광의 왕복 시간을 측정하여
대상 물체까지의 거리를 mm 단위로 계산한다.

센서는 **940nm VCSEL** 레이저를 사용하며,
일반적으로 **30~1000mm** 범위에서 ±3% 수준의 정밀도를 가진다.
통신은 **I²C** 인터페이스를 사용한다.

2. 하드웨어 연결 구조

VL53L0X 핀	기능	STM32 연결 예시
VIN	전원 입력 (2.6~5.5V)	3.3V
GND	접지	GND

VL53L0X 핀	기능	STM32 연결 예시
SDA	I ² C 데이터	PB7 (I ² C1_SDA)
SCL	I ² C 클록	PB6 (I ² C1_SCL)
XSHUT	센서 활성화	PB5 (GPIO Output)
GPIO1	인터럽트 출력 (옵션)	미사용 가능

⚠️ VL53L0X는 내부 풀업이 존재하지만, 안정적인 동작을 위해 SDA/SCL 라인에 4.7kΩ 외부 풀업 저항을 권장한다.

3. I²C 주소

VL53L0X의 기본 7비트 I²C 주소는 **0x29**이다.

HAL 함수 사용 시에는 왼쪽으로 한 비트 시프트해야 한다.

```
1 | #define VL53L0X_ADDR (0x29 << 1)
```

4. 동작 원리 요약

- 센서 활성화 (`XSHUT = HIGH`)
- 초기화 명령 전송
- 거리 측정 모드 설정 (Single 또는 Continuous)
- 측정 요청 후 결과 레지스터에서 거리(mm) 읽기

VL53L0X 내부는 MCU 명령 기반으로 동작하며,
데이터시트에 정의된 여러 **레지스터 맵(Register Map)**을 통해 제어된다.

5. 주요 레지스터

레지스터 주소	의미	크기	설명
0x0000	IDENTIFICATION_MODEL_ID	8bit	칩 ID (0xEE)
0x0014	SYSTEM_INTERRUPT_CLEAR	8bit	인터럽트 클리어
0x001E	SYSTEM_SEQUENCE_CONFIG	8bit	측정 시퀀스 구성
0x0062	RESULT_INTERRUPT_STATUS	8bit	측정 완료 플래그
0x0064	RESULT_RANGE_STATUS	8bit	거리 측정 결과 상태
0x006E	RESULT_RANGE_VALUE	16bit	거리(mm) 데이터

6. 초기화 시퀀스

VL53L0X는 부팅 직후 내부 펌웨어를 로드하므로,
전원 인가 후 약 **2ms**의 안정화 시간이 필요하다.

```
1 void VL53L0X_Init(void)
2 {
3     HAL_Delay(5);
4     // 기본적으로 ST가 제공하는 API 사용을 권장하지만,
5     // 간단한 거리 읽기용 최소 시퀀스는 아래와 같다.
6 }
```

실제 구현에서는 ST의 공식 드라이버([VL53L0X_api.c](#)) 사용을 권장한다.
그러나 아래 예시는 최소한의 I²C 전송으로 직접 읽는 방식이다.

7. I²C 읽기/쓰기 헬퍼 함수

```
1 void VL53L0X_WriteReg(uint16_t reg, uint8_t data)
2 {
3     uint8_t buf[3];
4     buf[0] = reg >> 8;
5     buf[1] = reg & 0xFF;
6     buf[2] = data;
7     HAL_I2C_Master_Transmit(&hi2c1, VL53L0X_ADDR, buf, 3, HAL_MAX_DELAY);
8 }
9
10 uint8_t VL53L0X_ReadReg(uint16_t reg)
11 {
12     uint8_t regBuf[2] = { reg >> 8, reg & 0xFF };
13     uint8_t val = 0;
14     HAL_I2C_Master_Transmit(&hi2c1, VL53L0X_ADDR, regBuf, 2, HAL_MAX_DELAY);
15     HAL_I2C_Master_Receive(&hi2c1, VL53L0X_ADDR, &val, 1, HAL_MAX_DELAY);
16     return val;
17 }
18
19 uint16_t VL53L0X_ReadReg16(uint16_t reg)
20 {
21     uint8_t regBuf[2] = { reg >> 8, reg & 0xFF };
22     uint8_t val[2];
23     HAL_I2C_Master_Transmit(&hi2c1, VL53L0X_ADDR, regBuf, 2, HAL_MAX_DELAY);
24     HAL_I2C_Master_Receive(&hi2c1, VL53L0X_ADDR, val, 2, HAL_MAX_DELAY);
25     return (val[0] << 8) | val[1];
26 }
```

8. 거리 데이터 읽기 함수

VL53L0X는 기본적으로 내부 연속 측정 모드로 설정 가능하다.

단일 거리 측정을 수행하려면 “Single Ranging Measurement”를 요청해야 한다.

```
1 uint16_t VL53L0X_ReadDistance(void)
2 {
3     uint16_t distance = 0;
4
5     // 측정 시작 명령 전송
6     VL53L0X_WriteReg(0x000E, 0x01); // SYSRANGE_START = 1
7
8     // 측정 완료 대기
9     uint8_t status = 0;
10    do {
11        status = VL53L0X_ReadReg(0x0013); // RESULT_INTERRUPT_STATUS
12        HAL_Delay(1);
13    } while ((status & 0x07) == 0);
14
15    // 거리 데이터 읽기
16    distance = VL53L0X_ReadReg16(0x0014 + 10); // RESULT_RANGE_VALUE (0x006E)
17
18    // 인터럽트 플래그 클리어
19    VL53L0X_WriteReg(0x0015, 0x01); // SYSTEM_INTERRUPT_CLEAR
20
21    return distance;
22 }
```

9. 사용 예시

```
1 void VL53L0X_Test(void)
2 {
3     uint16_t dist;
4
5     // XSHUT 핀 활성화
6     HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_SET);
7     HAL_Delay(5);
8
9     VL53L0X_Init();
10
11    while (1)
12    {
13        dist = VL53L0X_ReadDistance();
14        printf("Distance: %d mm\r\n", dist);
15        HAL_Delay(100);
16    }
17 }
```

출력 예시:

1	Distance: 215 mm
2	Distance: 217 mm
3	Distance: 216 mm

10. 측정 모드 비교

모드	설명	특징
Single Ranging Mode	명령 1회당 1회 측정	MCU 제어 용이
Continuous Ranging Mode	자동 반복 측정	주기적 거리 모니터링 가능
High Accuracy Mode	평균 200ms / 샘플	±2mm 수준 정밀
High Speed Mode	평균 20ms / 샘플	±5mm 수준 정밀

11. 성능 및 제약

조건	성능
측정 범위	30mm ~ 2000mm
정밀도	±3%
I ² C 속도	400kHz 권장
소비 전류	측정 시 약 19mA

12. 디버깅 포인트

증상	원인	해결
거리값 0 또는 8190 출력	초기화 누락	<code>VL53L0X_Init()</code> 재확인
응답 없음	주소 불일치	0x29 ↔ 0x52 확인
랜덤 값 출력	측정 완료 플래그 미확인	STATUS 폴링 루프 추가
일정 주기 후 정지	XSHUT 핀 Floating	내부 Pull-down 또는 GND 연결

13. 결론

- VL53L0X는 I²C 기반의 ToF 거리 센서로,
0x29 주소에서 측정 데이터를 직접 읽을 수 있다.
- HAL 레벨에서는 단순 I²C Read/Write로 제어 가능하지만,
ST의 공식 **VL53L0X API**를 사용하면 자동 보정 및 보정상수 로딩이 포함되어
더 안정적인 측정이 가능하다.

- “거리(mm)” 단위의 실시간 데이터는 약 33Hz 속도로 갱신 가능하다.

• I²C 듀얼센서 (0x48, 0x49) 수위센서 병렬 테스트

1. 개요

정전식(Capacitive) 수위센서 모듈은 내부적으로
수조 내 액체의 높이에 따라 정전용량이 변하며,
이를 아날로그 전압 또는 디지털 데이터로 변환하여 출력한다.
일부 모델은 I²C 인터페이스를 통해 정량화된 수위 데이터를
디지털 형태로 제공한다.

두 개의 동일한 수위센서를 동시에 측정하기 위해
I²C 버스에 병렬 연결되며,
각 센서가 서로 다른 주소(예: 0x48, 0x49)를 사용하도록 설정한다.
이로써 동일한 I²C 채널에서 두 센서 데이터를 독립적으로 읽을 수 있다.

2. 하드웨어 구성

구성 요소	핀명	STM32 포트	설명
센서 1	SDA / SCL	PB7 / PB6	I ² C1 공유
센서 2	SDA / SCL	PB7 / PB6	I ² C1 공유
센서 1	ADDR	GND	주소 0x48
센서 2	ADDR	VCC	주소 0x49
전원	VCC / GND	3.3V / GND	안정 전원 공급
풀업	4.7kΩ	SDA/SCL 라인	I ² C 안정화용

I²C는 버스형 구조이므로 SDA/SCL 라인은 두 센서가 공통으로 사용한다.

단, 각 센서의 **ADDR** 핀 상태를 달리하여 주소를 구분한다.

3. I²C 주소 체계

- 센서 1 : 0x48 (ADDR = GND)
- 센서 2 : 0x49 (ADDR = VCC)

HAL 통신 함수에서는 7-bit 주소를 8-bit 형태로 변환해야 하므로,
아래와 같이 시프트한다.

```
1 | #define SENSOR1_ADDR (0x48 << 1)
2 | #define SENSOR2_ADDR (0x49 << 1)
```

4. 센서 데이터 레지스터 구조 (예시)

주소	설명	크기	단위
0x00	센서 ID	1 byte	-
0x01	수위 데이터 (Low)	1 byte	mm
0x02	수위 데이터 (High)	1 byte	mm
0x03	상태 플래그	1 byte	bit field

실제 레지스터 맵은 센서 제조사에 따라 다르나,
대부분 16비트 수위 데이터를 0x01~0x02 주소에 저장한다.

5. 데이터 읽기 함수 구현

```
1 uint16_t Read_Water_Level(uint8_t devAddr)
2 {
3     uint8_t reg = 0x01;
4     uint8_t buf[2];
5     HAL_I2C_Master_Transmit(&hi2c1, devAddr, &reg, 1, HAL_MAX_DELAY);
6     HAL_I2C_Master_Receive(&hi2c1, devAddr, buf, 2, HAL_MAX_DELAY);
7     return (buf[0] << 8) | buf[1];
8 }
```

이 함수는 지정된 I²C 주소(`devAddr`)에서
0x01~0x02 레지스터 값을 읽어 16비트 수위(mm 단위)로 반환한다.

6. 듀얼 센서 병렬 테스트 루틴

```
1 void Dual_WaterSensor_Test(void)
2 {
3     uint16_t level1, level2;
4
5     printf("Dual I2C Water Level Sensor Test Start\r\n");
6
7     while (1)
8     {
9         level1 = Read_Water_Level(SENSOR1_ADDR);
10        level2 = Read_Water_Level(SENSOR2_ADDR);
11
12        printf("[0x48] Level1: %u mm\t[0x49] Level2: %u mm\r\n", level1, level2);
13
14        HAL_Delay(500);
15    }
16 }
```

출력 예시:

```
1 [0x48] Level1: 25 mm    [0x49] Level2: 75 mm
2 [0x48] Level1: 26 mm    [0x49] Level2: 74 mm
3 [0x48] Level1: 27 mm    [0x49] Level2: 73 mm
```

7. 에러 및 타임아웃 처리

센서 응답이 없거나 I²C 라인이 비정상 상태일 경우,

`HAL_I2C_Master_Transmit()` 또는 `HAL_I2C_Master_Receive()`에서

`HAL_TIMEOUT` 또는 `HAL_ERROR` 코드가 반환될 수 있다.

예외 처리 루틴 예시:

```
1 if (HAL_I2C_Master_Transmit(&hi2c1, devAddr, &reg, 1, 50) != HAL_OK)
2 {
3     printf("I2C Tx Error at 0x%X\r\n", devAddr >> 1);
4     HAL_I2C_DeInit(&hi2c1);
5     HAL_I2C_Init(&hi2c1);
6 }
```

이 방식으로 버스 정지를 감지한 후,

I²C 주변장치를 재초기화하여 복구할 수 있다.

8. 측정 데이터 필터링

실제 수위는 센서 노이즈나 액체 흔들림에 의해

±1~3mm 정도의 변동이 발생할 수 있다.

이를 보정하기 위해 이동평균 필터(Moving Average Filter)를 적용한다.

```
1 #define FILTER_SIZE 10
2 uint16_t level_buf1[FILTER_SIZE], level_buf2[FILTER_SIZE];
3
4 uint16_t Average(uint16_t *buf)
5 {
6     uint32_t sum = 0;
7     for (int i = 0; i < FILTER_SIZE; i++) sum += buf[i];
8     return sum / FILTER_SIZE;
9 }
```

주기적으로 최신 데이터를 버퍼에 저장하고 평균을 산출하여

보다 안정적인 수위를 출력한다.

9. 타이밍 및 버스 로드

항목	권장값
I ² C 속도	100~400kHz

항목	권장값
샘플링 주기	200~500ms
응답 시간	약 5~10ms
버스 풀업 저항	4.7kΩ (공통)

두 센서가 동시에 버스에 접근하므로,
통신 충돌 방지를 위해 MCU는 항상 마스터로 동작해야 한다.

10. 디버깅 포인트

현상	원인	조치
센서 중 하나만 응답	ADDR 핀 설정 오류	각 센서의 주소 확인
I ² C 버스 정지	SDA/SCL 노이즈	풀업 저항 강화 또는 차폐선 사용
랜덤 값 출력	필터 미적용	이동평균 필터 적용
측정 주기 불안정	HAL 타이머 간섭	I ² C 호출 시 타이밍 보장

11. 결론

- 두 개의 동일한 I²C 센서를 단일 버스에서 병렬 운용하기 위해 서로 다른 주소(0x48, 0x49)를 사용한다.
- STM32 HAL I²C 함수만으로 각 센서의 데이터를 독립적으로 읽어들일 수 있다.
- 노이즈 환경에서는 이동평균 필터 및 타임아웃 복구 루틴을 병행하는 것이 안정적인 운용에 필수적이다.
- 이 구조는 수위, 온도, 압력 등 다중 계측 노드 설계에 적용 가능하다.