11. 통합 실습 프로젝트

11.1 Smart Tank (스마트 수조)

- ## 초음파 + 수위센서 + HX711 통합

1. 개요

본 장에서는 **초음파 거리센서 (HC-SR04)**, **정전식 수위센서 (I²C 타입)**, **무게센서 (HX711)** 를
STM32 시스템에서 통합 운용하는 구조를 설명한다.

각 센서는 물리적으로 서로 다른 입력(초음파: 타이머, 수위: I²C, 무게: SPI 유사 인터페이스)을 사용하지만,
주기적 측정 루프와 데이터 필터링, Fail-Safe 보호, FreeRTOS 기반 Task 병렬 처리로 통합 제어가 가능하다.

---

2. 시스템 구조

센서	인터페이스	주요 함수	측정 단위	주기 (ms)
초음파(HC-SR04)	GPIO + Timer IC	`Ultrasonic_GetDistance()`	mm	200
수위센서(Capacitive, 0x48/0x49)	I²C	`LevelSensor_ReadAll()`	단계(0~20) → mm	500
무게센서(HX711)	GPIO(Bit-Bang)	`HX711_GetWeight()`	g	1000

---

3. 데이터 흐름

```
[센서 입력 계층]
   ├─ Ultrasonic_Read()
   ├─ I2C_LevelSensor_Read()
   └─ HX711_ReadRaw()

       ↓ (보정/필터링 계층)
   ├─ Offset & Scale Calibration
   ├─ Moving Average / Kalman Filter
   ├─ Timeout & Fail-Safe

       ↓ (통합 변환 계층)
   ├─ water_level_mm
   ├─ tank_height_mm
   ├─ weight_g

       ↓ (출력/제어 계층)
   ├─ DisplayTask (UART/OLED)
   ├─ ControlTask (밸브, 펌프)
```

```
19      └ RTCTask (주기 스케줄링)
```

4. 센서별 측정 함수

(1) 초음파 거리 측정

```
1  float Ultrasonic_GetDistance(void)
2  {
3      uint32_t echo_time = Get_Echo_Pulse_Width(); // Timer Input Capture
4      if (echo_time == 0) return -1.0f; // Timeout → Fail
5      return (float)echo_time * 0.017f; // 340m/s → 0.017 mm/us
6  }
```

(2) 수위센서 I²C 측정

```
1  float LevelSensor_GetHeight(void)
2  {
3      uint8_t level_data[20];
4      if (HAL_I2C_Master_Receive(&hi2c1, LEVEL_SENSOR_ADDR << 1, level_data, 20, 100)
   != HAL_OK)
5          return -1.0f;
6
7      uint8_t last_on = 0;
8      for (int i = 0; i < 20; i++)
9          if (level_data[i] > THRESHOLD) last_on = i;
10
11     return last_on * 5.0f + 3.0f; // 1단 ≈ 5mm
12 }
```

(3) 무게센서 (HX711) 측정

```
1  float HX711_GetWeight(void)
2  {
3      long raw = HX711_ReadRaw(&hx);
4      if (raw == 0) return -1.0f;
5
6      float weight = (raw - hx.offset) * hx.scale;
7      return ExponentialSmooth(weight, &hx.filter);
8  }
```

5. 통합 루틴

```
1  typedef struct {
2      float distance_mm;
3      float level_mm;
4      float weight_g;
```

```
    uint32_t timestamp;
} SensorData_t;

SensorData_t sensorData;

void Sensor_UpdateAll(void)
{
    sensorData.distance_mm = Ultrasonic_GetDistance();
    sensorData.level_mm    = LevelSensor_GetHeight();
    sensorData.weight_g    = HX711_GetWeight();
    sensorData.timestamp   = HAL_GetTick();

    // Fail-safe 처리
    if (sensorData.distance_mm < 0 || sensorData.level_mm < 0 ||
sensorData.weight_g < 0)
        System_FailSafe("Sensor Read Error");
}
```

6. FreeRTOS 기반 병렬 Task

```
void SensorTask(void *argument)
{
    for (;;)
    {
        Sensor_UpdateAll();
        osDelay(500);
    }
}

void DisplayTask(void *argument)
{
    for (;;)
    {
        printf("[DIST] %.1f mm | [LEVEL] %.1f mm | [WEIGHT] %.1f g\n",
               sensorData.distance_mm, sensorData.level_mm, sensorData.weight_g);

        OLED_ShowFloat(0, 0, sensorData.distance_mm, 1);
        OLED_ShowFloat(0, 2, sensorData.level_mm, 1);
        OLED_ShowFloat(0, 4, sensorData.weight_g, 1);
        osDelay(1000);
    }
}
```

7. 보정 및 필터링

항목	방법	함수
Offset 보정	무부하/기준점 측정	`HX711_SetOffset()`

항목	방법	함수
Scale 보정	표준추 기반	`HX711_SetScale()`
Exponential Smoothing	단기 잡음 제거	`ExponentialSmooth()`
Kalman Filter (1차)	측정 안정화	`Kalman_Update()`

---

8. 통합 제어 로직 예시

```
void ControlTask(void *argument)
{
    for (;;)
    {
        if (sensorData.level_mm < 50.0f)
            Pump_ON();
        else if (sensorData.level_mm > 180.0f)
            Pump_OFF();

        if (sensorData.weight_g > MAX_WEIGHT)
            Valve_Close();

        osDelay(200);
    }
}
```

---

9. Fail-Safe 연동

조건	Fail-Safe 동작
초음파 응답 없음	최근 정상값 유지 후 재시도
I²C 수위센서 NACK	I2C 재초기화 및 재시도
HX711 Deadlock	GPIO 재설정 후 복구
3회 이상 실패	System_Reset() 호출

---

10. 데이터 로그 구조

항목	단위	설명
Distance	mm	초음파 기반 수면~센서 거리
Level	mm	정전식 수위센서 기준 수위
Weight	g	탱크 무게 (HX711)

항목	단위	설명
Status	flag	FAIL / OK
Time	ms	측정 타임스탬프

UART / EEPROM / BLE 등 다양한 출력 채널로 전송 가능.

---

11. 시스템 요약

모듈	인터페이스	주기	오류 복구
HC-SR04	Timer + GPIO	200 ms	ECHO Timeout
Capacitive Level	I²C (0x48/0x49)	500 ms	Bus Recovery
HX711	GPIO Bit-Bang	1000 ms	Channel Reset
Display	OLED / UART	1000 ms	None
Control	Relay / Pump	200 ms	Fail-Safe Mode

---

12. 결론

본 통합 구조는

- **비동기 측정 구조 (FreeRTOS Task)**
- **필터링 및 보정 체계 (Offset, Scale, Kalman)**
- **Fail-Safe 복구 루틴**
- **주기적 데이터 통합 및 표시 루프**

를 모두 포함하는 안정적인 센서 융합 프레임워크이다.

이 구조를 기반으로 수위 제어, 유량 감시, 탱크 충전량 추정 등
다양한 **스마트 계측 시스템**으로 확장 가능하다.

• FreeRTOS Task 병렬 실행

---

1. 개요

FreeRTOS는 **다중 Task 기반의 선점형(Preemptive) RTOS 커널**로,
여러 개의 루프(태스크)를 병렬적으로 실행시켜 각 기능(센서, 디스플레이, 제어 등)을
독립적으로 동작시키는 구조를 제공한다.

각 Task는 독립적인 **스택 공간**과 **우선순위**를 가지며,
커널의 **스케줄러(Scheduler)** 가 CPU를 시분할(Time Slice)하여
동시에 실행되는 것처럼 관리한다.

---

2. Task 생성 구조

(1) CMSIS-RTOS2 API 기반

STM32CubeMX에서 FreeRTOS를 활성화하면,
`cmsis_os2.h` 기반의 API(`osThreadNew`, `osDelay` 등)를 사용한다.

```
void SensorTask(void *argument);
void DisplayTask(void *argument);
void ControlTask(void *argument);

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_FREERTOS_Init(); // CubeMX 생성 코드

    osKernelInitialize();

    osThreadNew(SensorTask, NULL, NULL);
    osThreadNew(DisplayTask, NULL, NULL);
    osThreadNew(ControlTask, NULL, NULL);

    osKernelStart();

    while (1); // 커널 시작 후 도달하지 않음
}
```

---

3. Task 실행 원리

FreeRTOS는 기본적으로 **선점형 스케줄링(Preemptive Scheduling)** 을 사용한다.

- 높은 우선순위의 Task가 실행 중이면 낮은 우선순위 Task는 대기 상태
- 동일 우선순위의 Task는 Tick(기본 1ms) 단위로 시분할
- `osDelay()` 또는 `vTaskDelay()` 호출 시 → Task Ready Queue로 이동

실행 상태 전이도

```
┌──────────────┐
|   Ready      |
└──────┬───────┘
       |(스케줄러)
       ▼
┌──────────────┐
|   Running    |
└──────┬───────┘
       |(osDelay, Block)
       ▼
┌──────────────┐
|   Blocked    |
```

```
└──────┬──────┘
       |(Timeout/Event)
       ▼
┌──────────────┐
|   Ready      |
└──────────────┘
```

---

4. Task 간 병렬 실행 예제

```c
void SensorTask(void *argument)
{
    for (;;)
    {
        Sensor_UpdateAll();     // 초음파 + 수위 + 무게 센서
        osDelay(200);           // 200ms 주기
    }
}

void DisplayTask(void *argument)
{
    for (;;)
    {
        OLED_ShowSensorData();  // 센서값 OLED 표시
        UART_LogSensorData();   // UART 전송
        osDelay(1000);
    }
}

void ControlTask(void *argument)
{
    for (;;)
    {
        Control_PumpAndValve(); // 수위/무게 기반 제어
        osDelay(300);
    }
}
```

위 세 Task는 서로 독립적으로 병렬 실행된다.
실제 CPU는 단일 코어지만, RTOS 스케줄러가 **Tick 단위로 Task를 전환**하여
논리적으로 동시에 동작하는 것처럼 보인다.

---

5. Task 우선순위 설정

우선순위는 각 Task 생성 시 속성 구조체로 지정한다.

```
const osThreadAttr_t sensorTask_attributes = {
    .name = "SensorTask",
    .priority = (osPriority_t) osPriorityHigh,
    .stack_size = 256 * 4
};

const osThreadAttr_t displayTask_attributes = {
    .name = "DisplayTask",
    .priority = (osPriority_t) osPriorityLow,
    .stack_size = 256 * 4
};
```

Task	우선순위	의미
SensorTask	High	주기적 측정, 실시간성 요구
ControlTask	Normal	펌프/밸브 동작 제어
DisplayTask	Low	화면 출력, 로그 처리

스케줄러는 항상 **가장 높은 우선순위의 Ready 상태 Task**를 실행한다.

---

6. 병렬 실행 중 동기화

여러 Task가 동일 자원(OLED, UART, I²C 등)을 공유할 경우,
Mutex 나 **Semaphore** 를 사용하여 동기화해야 한다.

```
osMutexId_t i2cMutex;

void SensorTask(void *argument)
{
    for (;;)
    {
        osMutexAcquire(i2cMutex, osWaitForever);
        I2C_ReadSensor();
        osMutexRelease(i2cMutex);
        osDelay(500);
    }
}
```

이렇게 하면 DisplayTask가 동시에 OLED를 I²C로 접근해도 충돌이 발생하지 않는다.

---

7. 병렬 실행 타이밍

Task	주기(ms)	CPU 점유율(%)	동작 역할
SensorTask	200	30	실시간 측정
ControlTask	300	25	액추에이터 제어

Task	주기(ms)	CPU 점유율(%)	동작 역할
DisplayTask	1000	10	OLED/UART 출력
IdleTask	—	나머지	Tickless Sleep, 전력절감

---

8. 병렬 실행 모니터링

FreeRTOS는 런타임 통계 기능(`vTaskGetRunTimeStats()`)을 제공한다.

```
char stats[256];
vTaskGetRunTimeStats(stats);
printf("%s\n", stats);
```

예시 출력:

```
Task Name     Time%
SensorTask    32.1
ControlTask   25.4
DisplayTask   12.7
IDLE          29.8
```

이를 통해 각 Task의 점유율과 스케줄링 상태를 분석할 수 있다.

---

9. 병렬 실행의 이점

항목	설명
독립성	각 기능 루프가 분리되어 유지보수 용이
응답성	실시간 이벤트(IRQ, 센서 입력)에 빠르게 반응
안정성	하나의 Task 오류가 전체 시스템 중단으로 이어지지 않음
확장성	새로운 기능 추가 시 기존 루프 영향 최소화

---

10. 결론

FreeRTOS의 **Task 병렬 실행 구조**는 STM32 기반 계측·제어 시스템에서
센서 읽기, 디스플레이, 통신, 제어를 완전히 독립적으로 병행 실행하게 해준다.

$$\boxed{\text{독립적 실행} + \text{스케줄링 제어} = \text{실시간 병렬 시스템 완성}}$$

이를 기반으로 **센서 융합, 저전력 관리, Fail-Safe 보호** 등 고신뢰 시스템 아키텍처로 발전시킬 수 있다.

- # RTC 기반 주기 측정

1. 개요

RTC(Real-Time Clock)는 저전력 하드웨어 타이머로,
초 단위의 절대 시간 추적뿐 아니라 **주기적 측정 스케줄링**에 활용할 수 있다.
STM32F103 시리즈의 RTC는 **LSE(32.768 kHz)** 클럭을 기준으로 동작하며,
전원이 차단되어도 백업 배터리를 통해 시간을 유지한다.

주기 측정은 RTC를 기준으로 "매 n초마다 측정 루틴 수행"하도록 하여
FreeRTOS나 일반 HAL 루프보다 **정확하고 저전력**인 시간 기반 제어를 가능하게 한다.

---

2. RTC 주기 측정 개념

RTC를 주기적 트리거로 활용하는 두 가지 방법은 다음과 같다.

1. **RTC Alarm 기능 사용**
   - 특정 시각(예: HH:MM:SS)에 알람을 설정하고 인터럽트 발생 시 측정 루틴 실행
   - 알람 완료 후 다음 알람 시각을 갱신하여 주기적 동작 구현
2. **RTC Wakeup Timer (F4 이상 시리즈)**
   - STM32F1에는 직접적 Wakeup Timer 없음 → Alarm 반복 갱신으로 동일 효과 구현

결과적으로, 일정한 시간 간격(예: 1분, 5분, 10분 등)마다
센서 측정, 데이터 저장, 송신 등을 **RTC 알람 기반으로 정확하게 수행**할 수 있다.

---

3. 시스템 흐름

```
[RTC Tick (초 단위)]
       ↓
[Alarm Interrupt 발생]
       ↓
[Measurement Task Notify]
       ↓
[센서 측정 → OLED 표시 → EEPROM 저장]
       ↓
[다음 알람 설정 (현재 + Δt)]
```

---

4. 구현 절차

(1) RTC 초기화

RTC를 LSE 32.768kHz 기준으로 초기화하고, 시간을 설정한다.

```
RTC_TimeTypeDef sTime = {0};
RTC_DateTypeDef sDate = {0};

sTime.Hours = 12;
sTime.Minutes = 0;
sTime.Seconds = 0;
HAL_RTC_SetTime(&hrtc, &sTime, RTC_FORMAT_BIN);
HAL_RTC_SetDate(&hrtc, &sDate, RTC_FORMAT_BIN);
```

(2) 알람 설정

주기적 트리거를 위해 현재 시각 기준으로 Δt (예: 10초 후) 알람을 설정한다.

```
RTC_AlarmTypeDef sAlarm = {0};
HAL_RTC_GetTime(&hrtc, &sTime, RTC_FORMAT_BIN);

uint8_t next_sec = (sTime.Seconds + 10) % 60;
sAlarm.AlarmTime.Hours = sTime.Hours;
sAlarm.AlarmTime.Minutes = sTime.Minutes;
sAlarm.AlarmTime.Seconds = next_sec;
sAlarm.Alarm = RTC_ALARM_A;

HAL_RTC_SetAlarm_IT(&hrtc, &sAlarm, RTC_FORMAT_BIN);
```

(3) 알람 인터럽트 처리

알람이 발생하면 인터럽트 콜백에서 측정 루틴을 호출하거나
FreeRTOS Task에 Notify 신호를 보낸다.

```
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    vTaskNotifyGiveFromISR(measureTaskHandle, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

(4) 주기적 알람 갱신

측정이 완료되면, 다시 다음 주기 알람을 설정한다.

```c
void ScheduleNextAlarm(uint8_t interval_sec)
{
    RTC_TimeTypeDef now;
    HAL_RTC_GetTime(&hrtc, &now, RTC_FORMAT_BIN);
    RTC_AlarmTypeDef next;

    next.AlarmTime.Hours = now.Hours;
    next.AlarmTime.Minutes = now.Minutes;
    next.AlarmTime.Seconds = (now.Seconds + interval_sec) % 60;
    next.Alarm = RTC_ALARM_A;
    HAL_RTC_SetAlarm_IT(&hrtc, &next, RTC_FORMAT_BIN);
}
```

---

5. FreeRTOS 기반 측정 루틴

```c
void MeasureTask(void *argument)
{
    for (;;)
    {
        ulTaskNotifyTake(pdTRUE, portMAX_DELAY);  // RTC 알람 이벤트 대기
        PerformSensorMeasurement();               // 초음파, 수위, 무게 측정
        OLED_DisplayData();
        ScheduleNextAlarm(60);                    // 60초 후 재설정
    }
}
```

이 구조는 `vTaskDelay()` 대신 RTC 기반 정확한 트리거를 사용하므로
슬립 모드 진입 및 저전력 주기 동작이 가능하다.

---

6. 주기 측정 정확도

클럭	주파수	오차(typical)	1시간 누적 오차
LSE	32.768 kHz	±20 ppm	±0.072초
HSE / LSI	8MHz / 40kHz	±1% 이상	±36초 이상

RTC를 LSE에 연결하면 장시간 동작 시에도 안정된 주기성을 확보할 수 있다.

---

7. 예시 로그

```
[RTC] Alarm Triggered @ 12:00:00
[Measure] Distance=132.4mm, Weight=1.246kg
[RTC] Next Alarm Set @ 12:01:00
[Sleep] Entering STOP Mode...
```

---

8. 장점 요약

항목	설명
정확성	32.768kHz LSE 기반으로 ±20ppm 수준의 시간 오차
저전력	메인 클럭 비활성 상태에서도 RTC 동작 유지
독립성	FreeRTOS Tick과 무관하게 실시간 스케줄 유지
유연성	동적 간격 조절 가능 (예: 10초, 1분, 5분 등)

---

9. 결론

RTC 기반 주기 측정은 단순한 delay 기반 주기 루프보다 훨씬 **정확하고 안정적**이며,
특히 저전력 모드(STOP, STANDBY)와 결합할 경우
배터리 기반 장시간 센서 노드 구현의 핵심이 된다.

$$\boxed{\text{RTC Alarm} + \text{Task Notify} = \text{정확하고 저전력인 주기 측정 구조}}$$

OLED + UART 상태 출력

1. 개요

센서 데이터나 시스템 상태를 시각적으로 확인하기 위해 **OLED(SSD1306)**와 **UART 터미널**을 동시에 활용할 수 있다.
두 출력을 병행하면 —

- OLED는 **현장용 디스플레이**,
- UART는 **PC 디버깅 로그** 역할을 하여
  테스트 및 유지보수 효율을 크게 높인다.

---

2. 출력 구조

```
┌──────────────┐
|  SensorTask |
|  ──────────  |
|  거리, 무게, 수위 측정 |
└──────┬───────┘
       |
       ▼
┌──────────────┐
| DisplayTask |
|  OLED + UART 출력  |
└──────────────┘
```

- `SensorTask`는 주기적으로 센서 데이터를 생성
- `DisplayTask`는 Queue나 전역 구조체를 통해 최신 값을 받아 OLED와 UART로 표시

---

3. OLED 초기화

```
#include "ssd1306.h"
#include "fonts.h"

void OLED_Init_Display(void)
{
    ssd1306_Init();
    ssd1306_Fill(Black);
    ssd1306_UpdateScreen();
}
```

---

4. OLED 출력 함수

```
void OLED_ShowStatus(float distance, float weight, float level)
{
    ssd1306_Fill(Black);
    ssd1306_SetCursor(0, 0);
    ssd1306_WriteString("SENSOR STATUS", Font_7x10, White);

    char buf[32];
    sprintf(buf, "Dist: %.1f mm", distance);
    ssd1306_SetCursor(0, 16);
    ssd1306_WriteString(buf, Font_6x8, White);

    sprintf(buf, "Weight: %.3f kg", weight);
    ssd1306_SetCursor(0, 26);
    ssd1306_WriteString(buf, Font_6x8, White);

    sprintf(buf, "Level: %.1f mm", level);
    ssd1306_SetCursor(0, 36);
    ssd1306_WriteString(buf, Font_6x8, White);

    ssd1306_UpdateScreen();
}
```

---

5. UART 출력 함수

```
void UART_PrintStatus(float distance, float weight, float level)
{
    char msg[64];
    sprintf(msg, "[DATA] D=%.1fmm, W=%.3fkg, L=%.1fmm\r\n",
            distance, weight, level);
    HAL_UART_Transmit(&huart1, (uint8_t *)msg, strlen(msg), HAL_MAX_DELAY);
}
```

---

6. FreeRTOS DisplayTask

```
void DisplayTask(void *argument)
{
    SensorData_t data;

    for (;;)
    {
        if (xQueueReceive(sensorQueueHandle, &data, portMAX_DELAY) == pdPASS)
        {
            OLED_ShowStatus(data.distance, data.weight, data.level);
            UART_PrintStatus(data.distance, data.weight, data.level);
        }
        osDelay(500); // 디스플레이 갱신 주기
    }
}
```

---

7. 통합 로그 예시

OLED 화면

```
SENSOR STATUS
Dist: 132.4 mm
Weight: 1.246 kg
Level: 48.5 mm
```

UART 터미널 (115200bps)

```
[DATA] D=132.4mm, W=1.246kg, L=48.5mm
[INFO] RTC Next Alarm: 12:01:00
```

---

8. 포인트 요약

구분	기능	비고
OLED	현장 디스플레이	I²C 기반 SSD1306
UART	실시간 로깅	PC 연결 (115200bps)
Queue	데이터 전달	SensorTask → DisplayTask
FreeRTOS	병렬 동작	주기별 Task 분리

---

9. 결론

`OLED + UART` 병행 출력은
센서 상태를 현장과 원격 양쪽에서 동시에 모니터링할 수 있는 실용적인 구조다.

$$\boxed{\text{DisplayTask} = \text{OLED 표시} + \text{UART 로그 출력}}$$

- # 밸브 제어 (Relay/MOSFET)

1. 개요

밸브(Valve) 제어는 유체, 공기, 또는 액체의 흐름을 제어하기 위해 **릴레이(전기적 스위치)** 또는 **MOSFET(반도체 스위치)**를 이용해 수행된다.
임베디드 시스템에서는 **GPIO 핀을 통해 밸브의 전원선을 On/Off** 하며, 센서 입력(수위, 압력, 시간)에 따라 자동 제어 루틴이 동작한다.

---

2. 하드웨어 구성

제어 방식	구분	제어 소자	구동 전류	특성
릴레이 방식	전기적 접점	Mechanical Relay	수십 mA	절연 강함, 동작 느림
MOSFET 방식	반도체 스위칭	N-channel MOSFET	수 mA	빠른 스위칭, 소음 없음

(1) 릴레이 회로

```
MCU GPIO ──> NPN 트랜지스터 ──> 릴레이 코일
                          |
                      다이오드(역기전력 보호)
```

(2) MOSFET 회로

```
MCU GPIO ──> Gate (10kΩ Pull-down)
Drain ──> 밸브 음극(–)
Source ──> GND
밸브 양극(+) ──> +12V
```

---

3. GPIO 초기화 (STM32 HAL 예시)

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

    __HAL_RCC_GPIOB_CLK_ENABLE();

    GPIO_InitStruct.Pin = GPIO_PIN_0;          // 밸브 제어 핀
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
```

```
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_0, GPIO_PIN_RESET);  // 초기 OFF
}
```

---

4. 밸브 제어 함수

```
#define VALVE_PORT   GPIOB
#define VALVE_PIN    GPIO_PIN_0

void Valve_On(void)
{
    HAL_GPIO_WritePin(VALVE_PORT, VALVE_PIN, GPIO_PIN_SET);
}

void Valve_Off(void)
{
    HAL_GPIO_WritePin(VALVE_PORT, VALVE_PIN, GPIO_PIN_RESET);
}
```

---

5. 제어 로직 예시

(1) 단순 수위 제어

```
if (level < LEVEL_MIN)
    Valve_On();    // 수위 부족 → 밸브 열기
else if (level > LEVEL_MAX)
    Valve_Off();   // 수위 충분 → 밸브 닫기
```

(2) 타이머 기반 자동 차단

```
uint32_t valve_start_time = 0;
bool valve_active = false;

void Valve_Control_Auto(void)
{
    if (!valve_active && level < LEVEL_MIN)
    {
        Valve_On();
        valve_start_time = HAL_GetTick();
        valve_active = true;
    }

    if (valve_active)
    {
        if (level > LEVEL_MAX || (HAL_GetTick() - valve_start_time > 10000))
        {
            Valve_Off();
```

```
            valve_active = false;
        }
    }
}
```

6. FreeRTOS Task 예시

```
void ControlTask(void *argument)
{
    SensorData_t data;
    for (;;)
    {
        if (xQueueReceive(sensorQueueHandle, &data, portMAX_DELAY) == pdPASS)
        {
            if (data.level < 40.0f)      Valve_On();
            else if (data.level > 80.0f) Valve_Off();
        }
        osDelay(200);
    }
}
```

7. Fail-Safe 보호 로직

조건	제어 동작
센서 통신 오류	밸브 자동 차단
측정 타임아웃	밸브 Off
과열, 과전류 검출	밸브 Off
Watchdog 리셋	초기 Off 상태 유지

```
void FailSafe_Check(void)
{
    if (sensor_error || timeout_flag)
    {
        Valve_Off();
        UART_Log("FailSafe: Valve OFF (Sensor Fault)\r\n");
    }
}
```

8. PWM 제어 (비례 제어용 MOSFET)

일부 전자밸브는 **유량을 제어하기 위한 비례 제어(PWM Duty)**를 지원한다.

```
void Valve_PWM_Set(uint16_t duty)
{
    __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1, duty);
}
```

Duty(%)를 수위 편차에 비례시켜 다음과 같이 적용할 수 있다.

$$Duty = K_p \times (Level_{target} - Level_{current})$$

---

9. 상태 피드백

OLED 및 UART에 밸브 상태 표시:

```
ssd1306_SetCursor(0, 46);
ssd1306_WriteString(Valve_State() ? "Valve: ON" : "Valve: OFF", Font_6x8, White);
```

UART 로그:

```
[VALVE] ON (Auto Control)
```

---

10. 결론

밸브 제어는 단순한 GPIO On/Off이지만,
센서 피드백 • 타이머 • Fail-safe 로직을 결합하면 안정적인 자동 제어 시스템이 된다.

$$\boxed{\text{Valve} = \text{Sensor Feedback} + \text{Time Limit} + \text{Safety Override}}$$

11.2 Sensor Node (저전력 계측 노드)

- # RTC 알람 기반 Wake-up

1. 개요

RTC(Real-Time Clock) 알람 기반 Wake-up은
저전력 모드(SLEEP, STOP, STANDBY)에서 시스템을 완전히 정지시킨 뒤,
RTC 알람 신호를 이용해 **주기적으로 MCU를 깨워 측정 및 제어 루틴을 수행**하는 방식이다.

이 방식은 **배터리 구동 환경**이나 **장기 동작 IoT 시스템**에서 전력 소모를 극적으로 줄이기 위해 사용된다.

---

2. 동작 개요

```
┌──────────────────┐
|    Active Mode |
| - 센서 측정      |
| - OLED/UART 출력 |
└──────┬───────────┘
       |
       ▼
┌──────────────────┐
|   Sleep Mode   |
| - CPU 정지       |
| - LSE 32.768kHz 유지 |
| - RTC 알람 대기    |
└──────┬───────────┘
       |(RTC Alarm IRQ)
       ▼
┌──────────────────┐
| Wake-up        |
| - RTC_Alarm_IRQHandler |
| - 측정 루틴 재실행 |
└──────────────────┘
```

---

3. 하드웨어 요구 조건

항목	설명
LSE 크리스털	32.768kHz 외부 크리스털 (RTC 클럭 소스)
RTC Alarm A (or B)	알람 인터럽트 트리거
NVIC 설정	`RTC_Alarm_IRQn` Enable
전원 모드	STOP 또는 STANDBY 지원 MCU

---

4. RTC 알람 설정 코드 예시

```c
void Set_Alarm_AfterSeconds(uint32_t seconds)
{
    RTC_AlarmTypeDef sAlarm = {0};

    HAL_RTC_GetTime(&hrtc, &sTime, RTC_FORMAT_BIN);
    HAL_RTC_GetDate(&hrtc, &sDate, RTC_FORMAT_BIN);

    uint8_t new_seconds = (sTime.Seconds + seconds) % 60;
    uint8_t carry_minutes = (sTime.Seconds + seconds) / 60;

    uint8_t new_minutes = (sTime.Minutes + carry_minutes) % 60;
    uint8_t carry_hours  = (sTime.Minutes + carry_minutes) / 60;
```

```
    uint8_t new_hours = (sTime.Hours + carry_hours) % 24;

    sAlarm.AlarmTime.Hours = new_hours;
    sAlarm.AlarmTime.Minutes = new_minutes;
    sAlarm.AlarmTime.Seconds = new_seconds;
    sAlarm.AlarmTime.SubSeconds = 0;
    sAlarm.AlarmTime.DayLightSaving = RTC_DAYLIGHTSAVING_NONE;
    sAlarm.AlarmTime.StoreOperation = RTC_STOREOPERATION_RESET;

    sAlarm.AlarmMask = RTC_ALARMMASK_DATEWEEKDAY;
    sAlarm.AlarmSubSecondMask = RTC_ALARMSUBSECONDMASK_ALL;
    sAlarm.AlarmDateWeekDaySel = RTC_ALARMDATEWEEKDAYSEL_DATE;
    sAlarm.AlarmDateWeekDay = 1;
    sAlarm.Alarm = RTC_ALARM_A;

    HAL_RTC_SetAlarm_IT(&hrtc, &sAlarm, RTC_FORMAT_BIN);
}
```

5. 알람 인터럽트 핸들러

```
void RTC_Alarm_IRQHandler(void)
{
    HAL_RTC_AlarmIRQHandler(&hrtc);
}

void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    // Wake-up 루틴 진입
    printf("[RTC] Wake-up Alarm Triggered!\r\n");

    // 필요한 센서 측정 및 Task Notify 수행
    xTaskNotifyFromISR(RTCTaskHandle, 0, eNoAction, NULL);
}
```

6. 저전력 진입

```
void Enter_LowPower_Mode(void)
{
    printf("Entering STOP Mode...\r\n");
    HAL_SuspendTick();                      // SysTick 중단
    HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
    HAL_ResumeTick();                       // 재개
    SystemClock_Config();                   // 클럭 재설정
}
```

7. Wake-up 흐름

1. `Set_Alarm_AfterSeconds(60);` — RTC 알람을 60초 후로 예약
2. `Enter_LowPower_Mode();` — MCU STOP 모드 진입
3. RTC는 계속 동작 (LSE 기반)
4. 알람 발생 → `RTC_Alarm_IRQHandler()` 호출
5. 시스템이 깨어나며 측정 Task 재개
6. 측정 완료 후 다시 Sleep 반복

---

8. FreeRTOS 연동

RTC 인터럽트 콜백 내에서 `xTaskNotifyFromISR()`을 이용해
측정 Task를 깨워 주기적으로 동작시킬 수 있다.

```
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    vTaskNotifyGiveFromISR(SensorTaskHandle, &xHigherPriorityTaskWoken);
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
}
```

---

9. OLED/UART 상태 표시

```
ssd1306_SetCursor(0, 54);
ssd1306_WriteString("Mode: STOP->WAKE", Font_6x8, White);
ssd1306_UpdateScreen();
UART_Log("[RTC] Wake-up from Alarm\r\n");
```

---

10. 결론

RTC 알람 기반 Wake-up은

- **센서 주기적 동작**,
- **배터리 절약**,
- **정확한 시간 기반 스케줄링**

을 동시에 달성하는 저전력 설계 핵심 기법이다.

$$\boxed{\text{Sleep} \rightarrow \text{RTC Alarm} \rightarrow \text{Wake} \rightarrow \text{Measure} \rightarrow \text{Sleep}}$$

STOP 모드 진입 / 복귀

1. 개요

STOP 모드(Stop Mode)는 STM32 마이크로컨트롤러의 **가장 대표적인 저전력 모드** 중 하나로,
CPU 클록과 대부분의 주변장치를 중단시키되, **SRAM, 레지스터, RTC, 백업 도메인**은 유지되는 상태이다.

이 모드는 **RTC 알람, 외부 인터럽트, 또는 특정 이벤트(EXTI, WKUP 핀 등)** 에 의해 복귀할 수 있다.
일반적으로 **RTC 알람 기반 Wake-up 시스템**과 결합하여 사용한다.

---

2. STOP 모드 특징

항목	설명
소비 전류	수십 μA 수준 (MCU 코어 정지)
유지 자원	SRAM, 레지스터, 백업 도메인 유지
클럭 상태	HSI/HSE 정지, LSE/Lsi만 유지 가능
복귀 트리거	RTC 알람, EXTI 라인, WKUP 핀 등
복귀 후 동작	`SystemClock_Config()` 재호출 필요
복귀 후 코드 위치	Sleep 전 지점 이후 코드 계속 실행

---

3. STOP 모드 진입 절차

1. **주요 주변장치 정지**
   - ADC, Timer, I²C, SPI 등 클럭 의존 장치 종료
   - OLED, UART 등 비필수 모듈 Sleep 처리
2. **SysTick 중단**

```
HAL_SuspendTick();
```

3. **STOP 모드 진입**

```
HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
```

4. **복귀 후 클럭 복원**

```
SystemClock_Config();
HAL_ResumeTick();
```

---

4. 예제 코드

```
void Enter_StopMode(void)
{
    printf("Entering STOP Mode...\r\n");

    // 주변장치 및 통신 모듈 비활성화
    HAL_I2C_DeInit(&hi2c1);
    HAL_UART_DeInit(&huart1);

    // SysTick 중단
    HAL_SuspendTick();

    // STOP 모드 진입
    HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);

    // 이후 RTC 알람 또는 EXTI에 의해 복귀
}

void Exit_StopMode(void)
{
    // 클럭 복원
    SystemClock_Config();
    HAL_ResumeTick();

    // 주변장치 재초기화
    MX_I2C1_Init();
    MX_USART1_UART_Init();

    printf("Woke up from STOP Mode\r\n");
}
```

---

5. RTC 알람과 연계

RTC 알람 인터럽트는 STOP 모드에서도 동작 가능하다.
아래 순서로 구성하면 완전한 주기적 Sleep-Wake 사이클이 가능하다.

```
// 1. 60초 후 알람 설정
Set_Alarm_AfterSeconds(60);

// 2. STOP 모드 진입
Enter_StopMode();

// 3. 알람 발생 시 RTC_Alarm_IRQHandler() 호출
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    Exit_StopMode();
    Sensor_Measure_Routine();
}
```

---

6. 복귀 동작 흐름

```
┌──────────────────────────┐
| Active Mode              |
| - 센서 측정, 통신 수행   |
└──────────┬───────────────┘
           |
           ▼
┌──────────────────────────┐
| STOP Mode                |
| - 클럭 정지 (HSI/HSE)    |
| - RTC 동작 유지          |
| - 전력소모 최소화        |
└──────────┬───────────────┘
           | (RTC Alarm / EXTI)
           ▼
┌──────────────────────────┐
| Wake-up                  |
| - 클럭 복구, HAL 재개    |
| - 측정 루틴 재실행       |
└──────────────────────────┘
```

---

7. 주의사항

항목	설명
클럭 재설정	STOP 복귀 시 `SystemClock_Config()` 반드시 호출
디버그 불가	STOP 모드에서는 JTAG/SWD 연결이 일시 중단됨
전원 안정성	외부 안정화 회로(전원, RTC LSE 필터 등) 필요
FreeRTOS 사용 시	Idle Task에서 Tickless Idle Mode와 병행 가능

---

8. FreeRTOS 연동 예시

FreeRTOS 환경에서는 Idle Task에서 STOP 모드 진입을 자동화할 수 있다.

```
void vApplicationIdleHook(void)
{
    __HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);
    HAL_SuspendTick();
    HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
    HAL_ResumeTick();
    SystemClock_Config();
}
```

---

9. 결론

STOP 모드는 STM32의 전력 효율을 극대화하는 핵심 기능으로,
RTC 알람 기반 Wake-up과 결합하면 **센서 측정 주기 제어**, **배터리 수명 연장**,
완전한 저전력 IoT 시스템 구현이 가능하다.

$$\boxed{\text{Active} \rightarrow \text{STOP} \rightarrow \text{RTC Alarm} \rightarrow \text{Wake} \rightarrow \text{Active}}$$

- # 측정 → 전송 → Sleep

🧩 개요

저전력 IoT 시스템(예: 수위 모니터링, 무선 로드셀 등)은
"**측정 → 데이터 전송 → 절전(Sleep)**" 주기로 동작하여
배터리 수명을 극대화한다.

이 구조는 **RTC 알람 기반 주기 제어** + **STOP 모드 진입/복귀** 로 구현한다.

⚙️ 전체 동작 흐름

```
┌──────────────────┐
| 1. 측정 단계     |
| - 초음파 센서    |
| - HX711 로드셀   |
| - RTC 타임스탬프 |
└────────┬─────────┘
         ▼
┌──────────────────┐
| 2. 전송 단계     |
| - BLE/UART 통신  |
| - OLED 상태 표시 |
└────────┬─────────┘
         ▼
┌──────────────────┐
| 3. Sleep 단계    |
| - 주변장치 종료  |
| - STOP 모드 진입 |
| - RTC 알람 대기  |
└──────────────────┘
```

📡 핵심 순서

```c
void Main_Cycle(void)
{
    /* 1. 측정 단계 ---------------------------------- */
    Sensor_Read_All();     // 초음파 + 수위 + HX711
    Process_Data();        // 오프셋/스무딩/보정

    /* 2. 전송 단계 ---------------------------------- */
    UART_Send_Data();      // PC/로거 전송
```

```
    BLE_Transmit_Data();   // 무선 송신
    OLED_Display_Status(); // 간략 상태 표시

    /* 3. Sleep 단계 ---------------------------------- */
    Prepare_Sleep();       // 주변장치 종료, I²C/UART DeInit
    Set_Alarm_AfterSeconds(60); // 다음 주기 (예: 60초)
    Enter_StopMode();      // STOP 모드 진입
}
```

🧭 1. 측정 단계 (Sensor_Read_All)

```
void Sensor_Read_All(void)
{
    float level = Ultrasonic_GetDistance();     // 초음파 거리
    float weight = HX711_GetWeight();           // 로드셀
    float temp = Read_TempSensor();             // 보조 센서

    data.level = level;
    data.weight = weight;
    data.temp = temp;

    printf("Level: %.1f cm, Weight: %.2f kg, Temp: %.1f°C\r\n",
           level, weight, temp);
}
```

> 💡 보정 알고리즘 (Offset, Kalman, Exponential Smoothing 등)은
> `Process_Data()` 내부에서 수행한다.

🔄 2. 전송 단계 (BLE/UART)

```
void UART_Send_Data(void)
{
    char buffer[64];
    sprintf(buffer, "L=%.1f, W=%.2f, T=%.1f\r\n",
            data.level, data.weight, data.temp);
    HAL_UART_Transmit(&huart1, (uint8_t*)buffer, strlen(buffer), 100);
}

void BLE_Transmit_Data(void)
{
    if (ble_connected)
        BLE_SendMeasurement(data.level, data.weight, data.temp);
}
```

> OLED에는 “전송 완료” 또는 “Sleep 진입 중...” 메시지를 표시해
> 상태 모니터링을 쉽게 할 수 있다.

🌙 3. Sleep 단계 (STOP Mode)

```
void Prepare_Sleep(void)
{
    HAL_I2C_DeInit(&hi2c1);
    HAL_UART_DeInit(&huart1);
    HAL_SuspendTick();
}

void Enter_StopMode(void)
{
    printf("Entering STOP Mode...\r\n");
    HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
}

void Wakeup_From_Stop(void)
{
    SystemClock_Config();
    HAL_ResumeTick();
    MX_I2C1_Init();
    MX_USART1_UART_Init();
    printf("Wake-up complete\r\n");
}
```

⏰ 4. RTC 알람 기반 Wake-up

```
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    Wakeup_From_Stop();
    Main_Cycle();  // 다음 주기 수행
}
```

🔋 전력 절감 포인트

항목	절전 조치
OLED	Sleep 명령 전송 후 전원 차단
UART/BLE	DeInit 또는 전원차단
I²C 센서	전원 GPIO OFF
MCU	STOP 모드 진입
RTC	LSE 기반 유지, 알람으로 복귀

📘 주기 동작 개념

상태	전력소모	주요 동작
Active	5–15 mA	측정, 연산, 통신
Sleep (STOP)	<100 µA	RTC 유지, 대기
Wake-up	1–2 ms	클럭 재복원, Task 재개

✅ 최종 구조 예시

```c
int main(void)
{
    System_Init();
    RTC_Init();
    OLED_Init();
    BLE_Init();

    while (1)
    {
        Main_Cycle();      // 측정 → 전송 → Sleep
    }
}
```

🔋 결론

이 구조는 배터리 기반 IoT 장치(예: **원격 수위/하중 측정기**)에서
측정 정확도 유지 + 수개월 이상 동작 가능하게 하는 핵심 루틴이다.

$$\boxed{\text{측정} \rightarrow \text{전송} \rightarrow \text{Sleep (RTC 알람} \rightarrow \text{Wake-up} \rightarrow \text{측정 반복)}}$$

- # 소비전류 측정 및 최적화

🧭 개요

저전력 시스템에서 **소비전류(Current Consumption)** 측정과 최적화는
배터리 수명, 발열, 안정성을 좌우하는 핵심 단계다.

STM32 기반의 IoT/센서 시스템에서는 다음 세 가지 측면에서 접근한다:

1. **정확한 소비전류 측정 방법**
2. **소자별 소비전류 분석**
3. **펌웨어 및 하드웨어 최적화 기법**

⚙️ 1. 소비전류 측정 환경 구성

측정 방법

방법	설명	장점	단점
멀티미터 직렬 측정	전원선에 직렬로 연결하여 DC 전류 측정	간단, 빠름	Sleep/Active 전환 구간 측정 불가
샌드위치 저항(Shunt Resistor)	저항에 걸린 전압을 오실로스코프로 측정	시간 분해능 높음	보정 필요, 회로 삽입 필요
전류 프로브 (Current Probe)	전류 클램프 방식으로 비접촉 측정	비침입식, 실시간	고가, 노이즈 민감
전력분석기 (Power Analyzer)	전류 + 전압 동시 분석	정확, 통합 분석	고가 장비 필요

기본 구성 예시

```
DC Power Supply (3.3V)
      |
      ├──[Shunt 1Ω]──▶ DUT (STM32 보드)
      |
      └──▶ Differential Probe (오실로스코프)
```

> 저항값은 0.1–1 Ω 권장.
> 전압 강하 최소화 및 전류 변동 분석 가능.

🧪 2. 측정 절차

(1) 시스템 상태별 전류 측정

상태	주요 구성요소	예상 전류 (예시)
Active Mode	MCU + OLED + I²C 센서	8–15 mA
Sleep (STOP)	RTC + 백업레지스터 유지	50–100 µA
Standby	RTC Off	<10 µA
Wake-up	클럭 복원, 주변장치 초기화	1–3 mA (단기)

(2) 파형 분석

- **측정 도구:** 오실로스코프 + 차동 프로브
- **파형 확인 항목:**
  - 측정 루틴 동안의 전류 피크
  - STOP 모드 진입 후 안정 전류

- Wake-up 트랜지언트 구간
- BLE/UART 전송 시 스파이크

> 파형 분석을 통해 **불필요한 주변장치가 Sleep 중에도 활성화되어 있는지** 식별 가능.

⚡ 3. 주요 소비원 분석

구성 요소	소비전류	절전 방법
MCU (STM32F1)	3–10 mA (Run) / 20–50 µA (STOP)	HSI→MSI 전환, STOP 모드
OLED (SSD1306)	5–10 mA	Sleep 명령, 전원 차단
HX711 (ADC)	1.5 mA	Power Down 핀 제어
VL53L0X (ToF)	10–20 mA (Active) / 5 µA (Idle)	`VL53L0X_StopMeasurement()`
BLE Module (ESP32-C3 등)	40–120 mA (Tx) / 10 µA (Deep Sleep)	연결 유지 시간 최소화
RTC	<1 µA	유지 필요, 별도 절전 불가

🔧 4. 펌웨어 절전 최적화 기법

(1) 클럭 제어

```
__HAL_RCC_GPIOA_CLK_DISABLE();
__HAL_RCC_SPI1_CLK_DISABLE();
__HAL_RCC_ADC1_CLK_DISABLE();
```

- 사용하지 않는 주변장치 클럭 비활성화
- SystemClock_Config() 내에서 PLL/HSI 주파수 조정

(2) GPIO 상태 관리

- 불필요한 Floating Input 방지 → **Pull-down 설정**
- LED 핀 OFF 상태 유지
- Sensor Power 핀 OFF로 전원 차단

```
HAL_GPIO_WritePin(SENSOR_PWR_GPIO_Port, SENSOR_PWR_Pin, GPIO_PIN_RESET);
```

(3) Sleep / STOP / Standby 모드 활용

모드	특징	복귀시간	소비전류
Sleep	CPU만 정지	1–5 µs	수 mA
STOP	RAM 유지, 클럭 정지	1–2 ms	수십 µA

모드	특징	복귀시간	소비전류
Standby	RAM 소멸, 최소 유지	수십 ms	수 μA

```
HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);
```

(4) RTC 알람 기반 Wake-up

STOP 모드에서도 RTC 유지, 알람으로 복귀 가능.

```
Set_Alarm_AfterSeconds(60);
```

🧮 5. 측정 데이터 분석 예시

구간	동작	전류 (mA)	지속시간 (ms)	에너지 (mC)
측정	초음파 + HX711	10.5	200	2.1
전송	BLE Tx	40.0	50	2.0
Sleep	STOP 모드	0.05	58000	2.9
합계	1분 주기	—	60000	**7.0 mC**

> 평균 전류 = 7.0 mC / 60 s = **0.116 mA**
> → 1000 mAh 배터리 사용 시 약 **360일** 동작 가능.

🧰 6. 하드웨어 절전 설계 포인트

항목	최적화 방법
전원공급	LDO 대신 DC-DC 사용
센서 전원	MOSFET으로 개별 제어
Pull-up 저항	10 kΩ 이상으로 상향
LED 표시	필요시만 점등
디커플링	전원 안정화용 100nF + 10μF
배터리 보호	저전압 차단 회로

📊 7. 최적화 검증 루틴

1. **Baseline 측정:** 모든 기능 활성 상태
2. **Stop Mode 측정:** MCU만 절전
3. **Peripheral Off 측정:** I²C/OLED 차단
4. **Sleep 루프 반복:** 평균값 산출
5. **각 구간 전류 로그 저장** (예: INA219 전류센서 사용)

✅ 결론

STM32 기반 시스템에서 소비전류를 줄이기 위해서는
하드웨어, 펌웨어, 주기 설계 세 가지를 통합적으로 최적화해야 한다.

$$I_{avg} = \frac{(I_{active} \cdot t_{active}) + (I_{sleep} \cdot t_{sleep})}{T_{cycle}}$$

즉,
활성 시간(t_active)을 최소화하고, Sleep 시간(t_sleep)을 최대화
하는 것이 절전 설계의 핵심이다.

11.3 자동 수위 제어 시스템

• 목표 수위 입력

🧭 개요

시스템이 자동으로 밸브나 펌프를 제어하기 위해서는
사용자가 **목표 수위(Target Level)** 를 설정해야 한다.
이 값은 측정된 실제 수위(`measured_level`)와 비교되어,
제어 루프(예: 펌프 ON/OFF, 밸브 개폐)의 기준으로 사용된다.

⚙️ 1. 변수 정의

```
float target_level = 100.0f;   // 목표 수위 (mm 단위)
float measured_level = 0.0f;   // 센서 측정값
```

> 기본값은 100 mm 등 시스템에 맞게 초기화한다.
> 이 값은 EEPROM 또는 Flash에 저장하여 전원 재인가 시에도 유지한다.

🧩 2. 입력 방법

입력 방식	설명	구현 방식
UART 명령 입력	PC 시리얼 모니터에서 명령으로 설정	`SET LVL 120`
OLED + 버튼 UI	버튼으로 수위 조정	Up/Down 버튼 입력

입력 방식	설명	구현 방식
BLE (앱 연동)	BLE Characteristic 통해 설정	스마트폰 앱 연동
EEPROM 초기값 로드	부팅 시 저장값 로드	`EEPROM_ReadFloat()`

💬 3. UART 명령 기반 예시

```
void UART_CommandHandler(char *cmd)
{
    if (strncmp(cmd, "SET LVL", 7) == 0)
    {
        float val = atof(cmd + 8);
        if (val >= 0 && val <= 300)
        {
            target_level = val;
            EEPROM_WriteFloat(ADDR_TARGET_LEVEL, target_level);
            printf("Target Level set to %.1f mm\r\n", target_level);
        }
        else
        {
            printf("Invalid range (0-300 mm)\r\n");
        }
    }
    else if (strncmp(cmd, "GET LVL", 7) == 0)
    {
        printf("Target Level: %.1f mm\r\n", target_level);
    }
}
```

> 사용 예:
>
> ```
> > SET LVL 120
> Target Level set to 120.0 mm
> > GET LVL
> Target Level: 120.0 mm
> ```

📟 4. OLED + 버튼 인터페이스 예시

```
void Handle_LevelSetting(void)
{
    static float temp_target = target_level;

    if (Button_Up_Pressed())    temp_target += 5.0f;
    if (Button_Down_Pressed())  temp_target -= 5.0f;

    if (temp_target < 0) temp_target = 0;
    if (temp_target > 300) temp_target = 300;
```

```
10
11      OLED_ShowString(0, 0, "Set Level:");
12      OLED_ShowFloat(80, 0, temp_target, 1);
13      OLED_ShowString(120, 0, "mm");
14
15      if (Button_OK_Pressed())
16      {
17          target_level = temp_target;
18          EEPROM_WriteFloat(ADDR_TARGET_LEVEL, target_level);
19          OLED_ShowString(0, 16, "Saved!");
20      }
21  }
```

> 버튼 UI 예시
>
> - ▲ : +5 mm
> - ▼ : −5 mm
> - OK : 저장

---

5. EEPROM 저장 및 로드

```
1   #define ADDR_TARGET_LEVEL 0x10
2
3   void Save_TargetLevel(void)
4   {
5       EEPROM_WriteFloat(ADDR_TARGET_LEVEL, target_level);
6   }
7
8   void Load_TargetLevel(void)
9   {
10      target_level = EEPROM_ReadFloat(ADDR_TARGET_LEVEL);
11      if (isnan(target_level) || target_level <= 0)
12          target_level = 100.0f; // 기본값
13  }
```

> EEPROM 접근 시 `HAL_I2C_Mem_Write()` / `HAL_I2C_Mem_Read()` 사용.

🧠 6. 제어 루프에서의 활용

```
void ControlTask(void)
{
    measured_level = Get_Level_mm();

    if (measured_level < target_level - 5)
        Pump_On();
    else if (measured_level > target_level + 5)
        Pump_Off();

    printf("Level: %.1f / %.1f mm\r\n", measured_level, target_level);
}
```

> ±5 mm의 **히스테리시스(Hysteresis)** 를 적용해
> 펌프의 빈번한 On/Off를 방지한다.

🪫 7. 예시 시나리오

상황	측정 수위	목표 수위	제어 상태
초기	90 mm	100 mm	Pump ON
상승	98 mm	100 mm	유지
도달	101 mm	100 mm	Pump OFF
하강	95 mm	100 mm	Pump ON (재시작)

✅ 결론

목표 수위 입력 기능은

- **사용자 맞춤 제어 설정**
- **EEPROM 영구 저장**
- **UI/명령/BLE 인터페이스 통합**

을 가능하게 하며,
자동 급수·배수 시스템의 **중추 제어 기준**으로 작동한다.

$$\boxed{\text{Target Level (User Input)} \Rightarrow \text{Control Reference (Pump/Valve Logic)}}$$

• PID 제어 (간단 비례제어)

🧭 개요

PID 제어는 **비례(Proportional)**, **적분(Integral)**, **미분(Derivative)** 요소를 이용해
목표값에 빠르고 안정적으로 도달하도록 제어하는 알고리즘이다.
여기서는 가장 단순한 형태인 **비례제어(P-제어)** 중심으로 구현을 다룬다.

수위, 압력, 온도, 속도 등의 연속 제어에 적용 가능하며
본 시스템에서는 **목표 수위(target_level)** 와 **현재 수위(measured_level)** 의 차이를
비례적으로 펌프 구동량(PWM Duty)으로 환산한다.

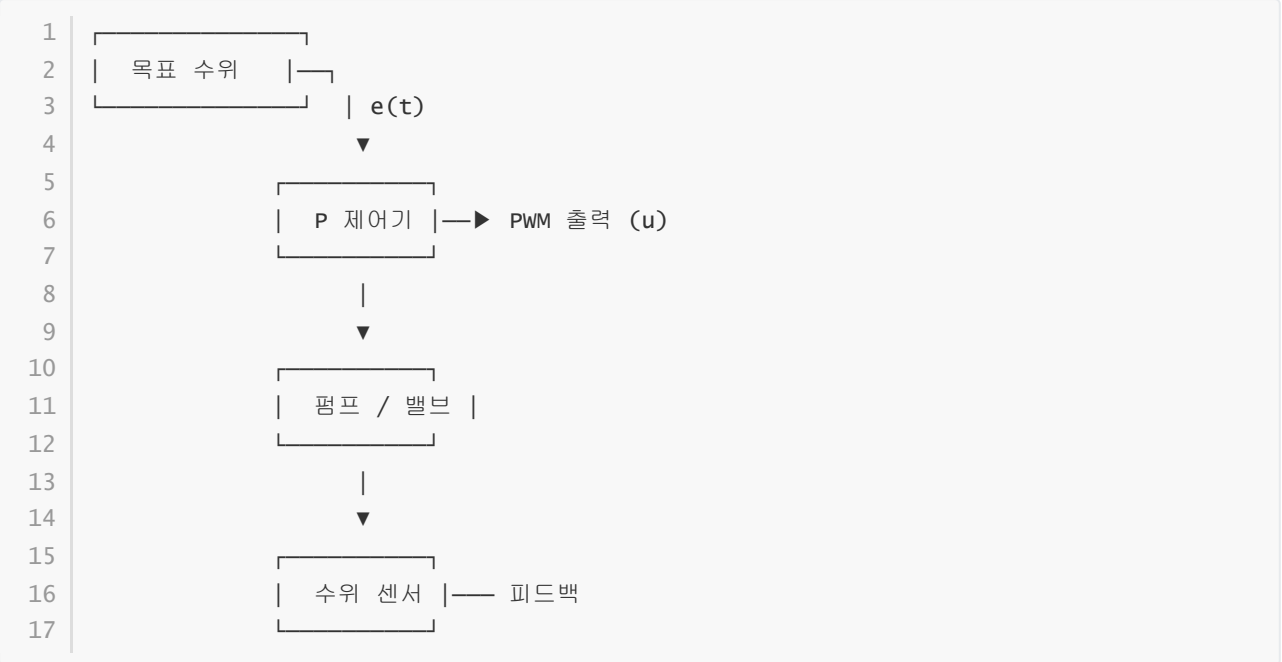
⚙️ 1. 기본 수식

$$u(t) = K_p \cdot e(t)$$

- $e(t) = \text{Target} - \text{Measured}$
- $u(t)$: 제어 출력 (펌프 PWM, 밸브 개도율 등)
- K_p: 비례 이득 (Gain, 민감도 조절)

> P 제어에서는 오차가 0이 되어야 출력이 0이 되므로,
> 잔류 오차(Steady-state Error)가 존재할 수 있다.

⚡ 2. 제어 루프 구조



🔧 3. 비례제어 구현 예시

```c
float target_level = 100.0f;   // 목표 수위 (mm)
float measured_level = 0.0f;   // 센서 입력
float Kp = 2.5f;               // 비례 이득
float control_output = 0.0f;   // PWM Duty (0~100%)

void PID_Control(void)
{
    float error = target_level - measured_level;

    // 단순 비례제어
    control_output = Kp * error;
```

```
12
13      // PWM Duty 제한
14      if (control_output > 100.0f)
15          control_output = 100.0f;
16      else if (control_output < 0.0f)
17          control_output = 0.0f;
18
19      Pump_SetPWM(control_output);
20  }
```

> 예:
>
> - `target_level = 120`, `measured_level = 100`, `Kp = 2.5` → `output = 50%`
> - 수위가 근접할수록 `error ↓`, 펌프 출력 자동 감소.

⚙️ 4. 실제 구동 시 주기적 호출

```
1  void ControlTask(void *argument)
2  {
3      for (;;)
4      {
5          measured_level = Get_Level_mm();
6          PID_Control();
7          osDelay(500);  // 0.5초 간격 제어
8      }
9  }
```

> FreeRTOS 환경에서는 **ControlTask** 내에서 주기적 실행.
> Sleep 모드 기반 시스템이라면 Wake-up 후 1회 수행 후 Sleep 재진입.

🧠 5. Gain 튜닝 (Kp 설정)

Kp 값	특징	현상
너무 작음	반응 둔함	목표 수위 도달 느림
적정	안정 수렴	정상 제어
너무 큼	진동, 오버슈트	수위가 반복 변동

> 초기에는 작은 값(예: 1.0)부터 시작하여 점진적으로 조정한다.
> 최적 Kp는 펌프 유량, 탱크 용량, 센서 응답속도에 따라 달라진다.

🧮 6. 제어 출력 변환 (PWM)

```
void Pump_SetPWM(float duty)
{
    __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1, (uint16_t)(duty * 10));
}
```

- 0–100% Duty → 0–1000 Timer Compare 값으로 변환
- PWM 주파수: 1~5 kHz 권장 (모터 진동 최소화)

📉 7. 동작 예시

목표(mm)	실제(mm)	오차(mm)	출력(%)
100	80	20	50
100	90	10	25
100	98	2	5
100	100	0	0

> 수위가 목표에 가까워질수록 펌프 출력이 점진적으로 줄어드는
> **소프트 제어(Smooth Control)** 특성을 갖는다.

⚙️ 8. 개선 방향

추가 항목	역할	효과
I (적분제어)	누적 오차 보정	잔류 오차 제거
D (미분제어)	급격한 변화 억제	진동 억제
Anti-Windup	적분 과포화 방지	안정성 향상
Feedforward	외란 예측 보정	응답속도 개선

> 실제 산업 제어에서는 PI 또는 PID 형태로 확장하여
> 장기 안정성과 응답성을 함께 확보한다.

✅ 결론

간단한 비례제어(P-control)는
센서 입력을 기반으로 한 펌프/밸브 제어의 **가장 기본적이고 효과적인 제어 방식**이다.
설계 시에는 Kp 값과 히스테리시스 범위를 조정해
수위 진동 최소화, 안정적 목표 도달을 달성해야 한다.

$$\boxed{u(t) = K_p \cdot (h_{\text{target}} - h_{\text{measured}})}$$

펌프 구동 및 차단

⚙️ 개요

펌프 구동(Pump Drive)은 **수위 제어 시스템의 핵심 동작부**로,
센서에서 측정한 수위를 바탕으로 목표 수위에 맞게 **자동으로 물을 공급하거나 차단**한다.
제어 방식은 단순한 **임계값 기반 온/오프 제어** 또는 **비례제어(P-control)**를 적용할 수 있다.

실제 하드웨어는 **릴레이(저전력 구동)** 또는 **MOSFET(고속 스위칭)** 으로 구동된다.

🔌 1. 하드웨어 구성

항목	설명	비고
제어 핀	GPIO 출력	예: PB12
구동 소자	NPN 트랜지스터, MOSFET, SSR, 릴레이 등	로드 종류에 따라 선택
보호 소자	Flyback 다이오드 (릴레이 코일 보호용)	역전압 방지
전원	5 V / 12 V / 24 V	펌프 정격에 따라 선택

릴레이 구동 회로 예시

```
MCU GPIO ─┬─> NPN 트랜지스터 → 릴레이 코일 → +12V
          └─> Flyback 다이오드 (역전압 보호)
```

MOSFET 구동 회로 예시

```
MCU GPIO → Gate
Drain → Pump(-)
Source → GND
Pump(+) → +12V
```

🔧 2. 제어 로직

(1) 임계값 기반 ON/OFF 제어

```c
#define LEVEL_LOW_THRESHOLD   40.0f   // 펌프 ON 기준(mm)
#define LEVEL_HIGH_THRESHOLD  90.0f   // 펌프 OFF 기준(mm)

void Pump_Control(float current_level)
{
    static uint8_t pump_state = 0; // 0:OFF, 1:ON

    if (current_level < LEVEL_LOW_THRESHOLD && pump_state == 0)
    {
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_SET);  // 펌프 ON
        pump_state = 1;
```

```
    }
    else if (current_level > LEVEL_HIGH_THRESHOLD && pump_state == 1)
    {
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_RESET); // 펌프 OFF
        pump_state = 0;
    }
}
```

- 히스테리시스(40 ↔ 90 mm)를 두어 펌프가 짧은 주기로 반복 ON/OFF되지 않도록 함.
- `GPIO_PIN_SET` 상태에서 릴레이/트랜지스터가 구동되도록 설계.

---

(2) 비례 제어 기반 PWM 구동

```
float Kp = 2.5f; // 비례 이득
float target_level = 100.0f;
float control_output = 0.0f;

void Pump_PWM_Control(float measured_level)
{
    float error = target_level - measured_level;
    control_output = Kp * error;

    if (control_output < 0) control_output = 0;
    if (control_output > 100) control_output = 100;

    __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1, (uint16_t)(control_output * 10));
}
```

- PWM으로 펌프 속도를 제어하여 **수위 근처에서 부드럽게 감속** 가능.
- MOSFET 기반 PWM 제어 회로에서 유용.

🧠 3. FreeRTOS Task 예시

```
void PumpTask(void *argument)
{
    for (;;)
    {
        float level = Get_Level_mm();

        #ifdef USE_PWM_CONTROL
            Pump_PWM_Control(level);
        #else
            Pump_Control(level);
        #endif

        osDelay(500); // 0.5초 주기
    }
}
```

> 주기적으로 수위를 확인하여 자동으로 구동/차단 제어 수행.

⚡ 4. 수동 제어 인터페이스

UART 또는 BLE 명령으로 수동 제어 모드를 제공할 수 있다.

```
void Pump_ManualControl(bool on)
{
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, on ? GPIO_PIN_SET : GPIO_PIN_RESET);
}
```

- `"PUMP ON"`, `"PUMP OFF"` 명령 수신 시 직접 구동
- 비상 상황 또는 디버깅용으로 사용

🧩 5. 보호 및 안전 로직

항목	내용
Dry-Run 보호	일정 시간 동안 수위 변화 없으면 펌프 차단
Overcurrent 보호	전류 센서 기반 과전류 감지 시 차단
Timeout 보호	일정 시간 이상 연속 구동 시 자동 정지
Low-Voltage Cutoff	전원 강하 시 구동 금지

예시:

```
if (pump_state == 1 && run_time > MAX_PUMP_RUN_TIME)
{
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_RESET);
    pump_state = 0;
}
```

📊 6. 상태 모니터링

UART 로그 또는 OLED에 펌프 상태 표시

```
printf("[PUMP] State: %s, Level: %.1f mm\r\n",
        pump_state ? "ON" : "OFF", current_level);
```

OLED 예시

```
PUMP STATUS
ON
LEVEL: 58.4mm
```

✅ 결론

펌프 제어는

- 임계값 기반 온/오프,
- 또는 비례제어(PWM) 기반 속도 제어로 구현 가능하다.

안정성을 위해 **히스테리시스, 타임아웃, 보호로직**을 반드시 추가해야 하며,
FreeRTOS 환경에서는 **PumpTask**로 분리하여 병렬 제어 루프를 구성하는 것이 이상적이다.

$$\boxed{\text{Level} \rightarrow \text{Error} \rightarrow \text{Control Output} \rightarrow \text{Pump ON/OFF or PWM}}$$

• 경보 / 상태 모니터링

⚙️ 개요

경보(Alarm) 및 상태 모니터링은 **센서 데이터 이상 감지, 시스템 오류, 한계값 초과 상황을 실시간으로 감시하고 사용자에게 알림을 제공**하는 기능이다.
수위, 무게, 전원, 통신 등 다양한 요소를 통합적으로 확인하여 **안전한 자동제어 시스템**을 유지한다.

🔍 1. 경보 조건 정의

구분	조건	설명
수위 과저하 (Low Level)	`level < MIN_LEVEL`	탱크가 너무 비어 있음
수위 과충만 (High Level)	`level > MAX_LEVEL`	오버플로 위험
센서 오류	I²C, HX711 Timeout 발생	측정 실패 또는 통신 불량
펌프 과구동	펌프 동작 시간 > `MAX_RUN_TIME`	드라이런 또는 고착 위험
RTC 알람 미동작	일정 주기 내 미기동	타이머 오류 가능성
전원 불안정	전압 < 4.5V (예시)	전원 부족

⚠️ 2. 경보 상태 구조체

```
typedef struct {
    uint8_t low_level;
    uint8_t high_level;
    uint8_t sensor_error;
    uint8_t pump_timeout;
    uint8_t power_fault;
} AlarmFlags_t;

AlarmFlags_t alarm_flags = {0};
```

> 개별 경보는 플래그 단위로 관리되어, LED, OLED, BLE 등으로 손쉽게 표시 가능.

🧮 3. 경보 판정 루틴

```
#define MIN_LEVEL 20.0f
#define MAX_LEVEL 120.0f
#define MAX_PUMP_RUN_TIME 30000 // 30초

void Alarm_Check(float level, uint8_t sensor_ok, uint32_t pump_run_time)
{
    alarm_flags.low_level   = (level < MIN_LEVEL);
    alarm_flags.high_level  = (level > MAX_LEVEL);
    alarm_flags.sensor_error = !sensor_ok;
    alarm_flags.pump_timeout = (pump_run_time > MAX_PUMP_RUN_TIME);
}
```

> 매 주기마다 측정값을 기반으로 경보 조건을 재평가.

💡 4. 시각/음성 알림

(1) LED 표시

```
void LED_Alarm_Display(void)
{
    if (alarm_flags.low_level)
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_SET); // 빨강 LED ON
    else
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_0, GPIO_PIN_RESET);

    if (alarm_flags.high_level)
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_1); // 황색 LED 점멸
}
```

(2) 부저 경보

```
void Buzzer_Alarm(void)
{
    if (alarm_flags.high_level || alarm_flags.sensor_error)
    {
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_8, GPIO_PIN_SET);
        HAL_Delay(200);
        HAL_GPIO_WritePin(GPIOB, GPIO_PIN_8, GPIO_PIN_RESET);
    }
}
```

(3) OLED / UART 출력

```
void Display_Alarm_Status(void)
{
    printf("[ALARM] ");
    if (alarm_flags.low_level)   printf("LOW ");
    if (alarm_flags.high_level)  printf("HIGH ");
    if (alarm_flags.sensor_error) printf("SENSOR ");
    if (alarm_flags.pump_timeout) printf("TIMEOUT ");
    printf("\r\n");
}
```

OLED 예시 화면

```
ALARM STATUS
HIGH LEVEL !!
PUMP TIMEOUT
```

🔄 5. FreeRTOS 기반 모니터링 Task

```
void AlarmTask(void *argument)
{
    for (;;)
    {
        float level = Get_Level_mm();
        uint8_t sensor_ok = Check_Sensor_Status();
        uint32_t pump_time = Get_Pump_RunTime();

        Alarm_Check(level, sensor_ok, pump_time);
        LED_Alarm_Display();
        Display_Alarm_Status();

        osDelay(1000); // 1초 주기 확인
    }
}
```

> 경보 판정, 시각화, 로그 전송을 주기적으로 수행.

📶 6. 원격 모니터링 (BLE / UART / IoT 연동)

(1) BLE / UART 전송 프레임 예시

```
typedef struct {
    float level;
    float weight;
    uint8_t pump_state;
    AlarmFlags_t alarm;
} SystemStatus_t;
```

BLE나 UART를 통해 실시간 상태를 외부로 송신

```
[STATUS]
LEVEL=58.3mm
WEIGHT=4.12kg
PUMP=ON
ALARM=HIGH_LEVEL
```

(2) IoT 확장 (MQTT or HTTP)

- ESP32 또는 STM32+ESP 모듈을 통해 클라우드로 전송 가능
- 주기적 상태 업로드 + 경보 시 즉시 푸시 가능

🧠 7. 경보 복구 및 초기화

경보가 해제되면 자동으로 복귀하거나, 수동 리셋 가능하도록 설계.

```
void Alarm_Reset(void)
{
    memset(&alarm_flags, 0, sizeof(alarm_flags));
}
```

> 예: 버튼 입력, BLE 명령 `"RESET ALARM"` 수신 시 호출

✅ 결론

경보 및 상태 모니터링은

- **안전성 확보**,
- **실시간 진단**,
- **원격 알림 연계**를 위한 핵심 요소이다.

다음과 같은 흐름으로 구성된다:

센서데이터 → 이상감지 → 경보플래그설정 → 시각/음성알림 → 전송 → 복구

💬 **추천 확장**

- RTC 로그에 경보 발생 시각 저장
- OLED 화면 자동 전환 (정상 ↔ 경보 상태)
- BLE Notification 기능으로 모바일 경고 연동