7. RTC (Real-Time Clock)

7.1 RTC 기본

- **LSE 32.768kHz 클럭 설정**
- **백업 도메인 및 배터리 유지**
- **RTC 구조체 (`RTC_TimeTypeDef`, `RTC_AlarmTypeDef`)**

7.2 HAL 함수

- **`HAL_RTC_SetTime()`, `HAL_RTC_GetTime()`**

1. 개요

LSE (Low-Speed External) 클럭은 32.768 kHz 외부 크리스털을 사용하여
RTC(Real-Time Clock), 백업 도메인, 저전력 타이머(LPTIM) 등에 안정적이고 정밀한 기준 주파수를 제공한다.
내부 RC 오실레이터인 **LSI (≈37 kHz)** 보다 정확도가 높아,
장시간의 시간 유지가 필요한 애플리케이션(예: 시계, 데이터 로거, 저전력 타이머)에 필수적으로 사용된다.

STM32의 LSE는 **Backup Domain** 영역에 속하며,
VBAT 전원이 유지되는 한 RTC 동작 및 시간 카운트가 지속된다.

---

2. 하드웨어 구성

1. **크리스털 사양**
   - 주파수: 32.768 kHz
   - 부하 정전용량 (CL): 6 pF ~ 12.5 pF (데이터시트 참조)
   - ESR (등가직렬저항): 50 kΩ 이하 권장
   - 드라이브 레벨: 최대 1 µW 이하
2. **PCB 설계 권장사항**
   - X1, X2 핀은 짧고 대칭적으로 배치
   - 노이즈 간섭 방지를 위해 주변 고주파 신호선과 격리
   - 그라운드 쉴드(GND guard ring)로 감싸는 것이 안정적
   - LSE 부하 커패시터(C1, C2)는 다음 식으로 산출

$$C_L = \frac{C_1 \times C_2}{C_1 + C_2} + C_{stray}$$

예: CL = 12.5 pF, 기생정전용량 Cstray ≈ 2 pF →
C1 = C2 ≈ 18 pF

---

3. 펌웨어 설정 (HAL 기반)

(1) CubeMX 설정 절차

1. **RCC → LSE 설정: "LSE Crystal/Ceramic Resonator" 선택**
2. **RTC Clock Source → LSE 선택**
3. **Enable RTC**
4. 코드 생성 시 `MX_RTC_Init()` 및 `HAL_RCC_OscConfig()` 자동 삽입

(2) 직접 코드 설정

```
RCC_OscInitTypeDef RCC_OscInitStruct = {0};
RCC_PeriphCLKInitTypeDef PeriphClkInitStruct = {0};

// 1. LSE 활성화
RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_LSE;
RCC_OscInitStruct.LSEState = RCC_LSE_ON;
RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
HAL_RCC_OscConfig(&RCC_OscInitStruct);

// 2. RTC 클럭 소스 설정
PeriphClkInitStruct.PeriphClockSelection = RCC_PERIPHCLK_RTC;
PeriphClkInitStruct.RTCClockSelection = RCC_RTCCLKSOURCE_LSE;
HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct);

// 3. RTC 클럭 활성화
__HAL_RCC_RTC_ENABLE();
```

---

4. 상태 확인 및 예외 처리

```
if (__HAL_RCC_GET_FLAG(RCC_FLAG_LSERDY) == RESET) {
    // LSE Ready 안 됨 → 예비로 LSI 사용 가능
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_LSI;
    RCC_OscInitStruct.LSIState = RCC_LSI_ON;
    HAL_RCC_OscConfig(&RCC_OscInitStruct);
    PeriphClkInitStruct.RTCClockSelection = RCC_RTCCLKSOURCE_LSI;
    HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct);
}
```

- **LSERDY 플래그**는 LSE 발진 안정화 완료 시 `SET` 된다.
- 안정화에는 약 300 ms ~ 1 s 필요할 수 있음.

---

5. 주의 사항

1. **Backup Domain Write Protection**
   - RTC, LSE 설정 시 백업 도메인은 보호되어 있음.
   - 변경 전 반드시 보호 해제 필요:

```
HAL_PWR_EnableBkUpAccess();
```

2. **VBAT 및 백업 레지스터 유지**
   - 메인 전원(VDD) 차단 시에도 VBAT이 유지되면 RTC 카운터 동작 지속.
3. **LSE 드라이브 레벨 조정**
   - 일부 MCU는 드라이브 강도 설정 가능:

```
__HAL_RCC_LSEDRIVE_CONFIG(RCC_LSEDRIVE_LOW);
```

   - 크리스털 사양에 따라 LOW / MEDIUM / HIGH 선택

---

6. RTC 동작 예시

```
RTC_HandleTypeDef hrtc;

void MX_RTC_Init(void)
{
    hrtc.Instance = RTC;
    hrtc.Init.HourFormat = RTC_HOURFORMAT_24;
    hrtc.Init.AsynchPrediv = 127;
    hrtc.Init.SynchPrediv = 255;  // LSE 32.768kHz 기준 → 1Hz tick
    hrtc.Init.OutPut = RTC_OUTPUT_DISABLE;
    HAL_RTC_Init(&hrtc);
}
```

---

7. 결론

- LSE 32.768 kHz는 RTC의 정확한 시간 유지에 필수적이다.
- 안정적 동작을 위해 크리스털 ESR, 부하 커패시터, 배선 길이를 세심히 고려해야 한다.
- LSE 실패 시 LSI로 대체 가능하지만, 정확도는 수 % 단위로 저하된다.
- 저전력 모드에서 LSE는 RTC를 통한 **실시간 웨이크업 기반 타이머**로 매우 유용하게 활용된다.

- `HAL_RTC_SetAlarm_IT()`

1. 개요

`HAL_RTC_SetAlarm_IT()` 함수는 **RTC Alarm 인터럽트 기반 알람 설정**을 수행한다.
RTC는 LSE(32.768 kHz) 또는 LSI(≈37 kHz) 클럭을 기준으로 1초 단위 카운트를 유지하며,
알람 시간(Alarm A 또는 Alarm B)을 지정하면 해당 시점에 인터럽트를 발생시킨다.

이 함수는 내부적으로 **RTC Alarm 레지스터(ALARMA 또는 ALARMB)** 에 값을 설정하고,
`RTC_IT_ALRA` 또는 `RTC_IT_ALRB` 인터럽트를 활성화한다.

---

2. 함수 원형

```
HAL_StatusTypeDef HAL_RTC_SetAlarm_IT(
    RTC_HandleTypeDef *hrtc,
    RTC_AlarmTypeDef *sAlarm,
    uint32_t Format
);
```

인자명	설명
`hrtc`	RTC 핸들 포인터 (`RTC_HandleTypeDef`)
`sAlarm`	알람 설정 구조체 포인터 (`RTC_AlarmTypeDef`)
`Format`	시간 포맷: `RTC_FORMAT_BIN` 또는 `RTC_FORMAT_BCD`

---

3. 구조체 정의

```
typedef struct {
  RTC_TimeTypeDef  AlarmTime;      // 알람 시각 (시, 분, 초)
  uint32_t         AlarmMask;      // 비교 마스크 비트 (예: 초만 비교)
  uint32_t         AlarmSubSecondMask;
  uint32_t         AlarmDateWeekDaySel; // 날짜 기반 / 요일 기반 선택
  uint8_t          AlarmDateWeekDay;    // 일(day) 또는 요일(week day)
  uint32_t         Alarm;          // ALARM_A 또는 ALARM_B 선택
} RTC_AlarmTypeDef;
```

---

4. 사용 절차

(1) RTC 초기화

```
RTC_HandleTypeDef hrtc;

void MX_RTC_Init(void)
{
    hrtc.Instance = RTC;
```

```
    hrtc.Init.HourFormat = RTC_HOURFORMAT_24;
    hrtc.Init.AsynchPrediv = 127;
    hrtc.Init.SynchPrediv = 255;  // LSE 32.768kHz → 1Hz
    hrtc.Init.OutPut = RTC_OUTPUT_DISABLE;
    HAL_RTC_Init(&hrtc);
}
```

(2) 현재 시간 설정

```
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);

sDate.WeekDay = RTC_WEEKDAY_THURSDAY;
sDate.Month = RTC_MONTH_NOVEMBER;
sDate.Date = 6;
sDate.Year = 25;
HAL_RTC_SetDate(&hrtc, &sDate, RTC_FORMAT_BIN);
```

(3) 알람 시간 설정

```
RTC_AlarmTypeDef sAlarm = {0};

sAlarm.AlarmTime.Hours   = 12;
sAlarm.AlarmTime.Minutes = 0;
sAlarm.AlarmTime.Seconds = 10;
sAlarm.AlarmMask = RTC_ALARMMASK_DATEWEEKDAY;  // 날짜 무시
sAlarm.AlarmSubSecondMask = RTC_ALARMSUBSECONDMASK_ALL;
sAlarm.AlarmDateWeekDaySel = RTC_ALARMDATEWEEKDAYSEL_DATE;
sAlarm.AlarmDateWeekDay = 0;
sAlarm.Alarm = RTC_ALARM_A;

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

이 설정은 **12:00:10** 시점에 **Alarm A 인터럽트**를 발생시킨다.

---

5. 인터럽트 콜백 처리

`HAL_RTC_SetAlarm_IT()` 호출 후 RTC 알람 발생 시
다음 콜백 함수가 자동 호출된다:

```
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    printf("RTC Alarm A Triggered!\r\n");
    // 여기에 사용자 코드 삽입 (예: 센서 측정, 디스플레이 업데이트 등)
}
```

6. NVIC 설정

알람 인터럽트를 정상적으로 받기 위해 NVIC 설정 필요:

```
void HAL_RTC_MspInit(RTC_HandleTypeDef* rtcHandle)
{
    if (rtcHandle->Instance == RTC)
    {
        HAL_NVIC_SetPriority(RTC_Alarm_IRQn, 0, 0);
        HAL_NVIC_EnableIRQ(RTC_Alarm_IRQn);
    }
}
```

그리고 ISR 핸들러:

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

7. 반복 알람 (주기적 알람)

RTC는 **하드웨어적으로 자동 반복 기능이 없다**.
주기 알람을 구현하려면 콜백 내부에서 알람을 갱신해야 한다.

```
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    RTC_AlarmTypeDef sAlarm = {0};

    // 다음 알람: 5초 후
    RTC_TimeTypeDef now;
    HAL_RTC_GetTime(hrtc, &now, RTC_FORMAT_BIN);

    sAlarm.AlarmTime.Hours = now.Hours;
    sAlarm.AlarmTime.Minutes = now.Minutes;
    sAlarm.AlarmTime.Seconds = (now.Seconds + 5) % 60;
    sAlarm.AlarmMask = RTC_ALARMMASK_DATEWEEKDAY;
    sAlarm.Alarm = RTC_ALARM_A;
    HAL_RTC_SetAlarm_IT(hrtc, &sAlarm, RTC_FORMAT_BIN);
}
```

8. 주의 사항

1. **Backup Domain 접근 허용**

```
HAL_PWR_EnableBkUpAccess();
```

2. **LSE Ready 확인 후 RTC 초기화**
    - LSE 안정화 전에 RTC 설정 시 실패 가능.
3. **중복 알람 방지**
    - 새 알람 설정 전 `HAL_RTC_DeactivateAlarm()` 호출 권장.
4. **저전력 모드에서 동작**
    - STOP/STANDBY 모드에서도 LSE 기반 RTC는 동작 유지.
    - 알람 인터럽트로 MCU 자동 웨이크업 가능.

---

9. 결론

`HAL_RTC_SetAlarm_IT()` 은 **정밀한 시각 기반 이벤트 트리거**를 구현하기 위한 핵심 함수이다.

- RTC 시간·날짜를 설정한 후 알람 시점을 지정하면 인터럽트 발생
- 콜백 내부에서 센서 측정, 로그 저장, 슬립 복귀 등 수행 가능
- LSE 기반일 경우 ±20 ppm 수준의 정확도(≈1.7초/하루) 확보 가능

저전력 시스템에서는 이 알람 인터럽트를 활용하여 **주기적 웨이크업**과 **에너지 절약형 데이터 로깅**을 구현한다.

• 알람 인터럽트 동작 (`RTC_Alarm_IRQHandler`)

1. 개요

`RTC_Alarm_IRQHandler()` 함수는 **RTC 알람 이벤트 발생 시 NVIC를 통해 호출되는 인터럽트 서비스 루틴(ISR)** 이다.
이 핸들러는 HAL 계층의 `HAL_RTC_AlarmIRQHandler()`를 호출하여, 내부적으로 **알람 플래그 클리어**, **콜백 실행**, **후처리**를 수행한다.

이 함수는 RTC 알람 인터럽트가 NVIC에 등록되어 있어야만 정상 동작하며,
알람이 발생하면 **RTC 알람 플래그(ALRAF)** 가 세트되어, 해당 IRQ가 트리거된다.

---

2. 기본 동작 흐름

RTC 알람 발생 시 내부 동작 순서는 다음과 같다:

1. **RTC 하드웨어가 알람 시간(Alarm A/B)에 도달**
2. **ALRAF 또는 ALRBF 플래그**가 RTC_ISR 레지스터에 세트됨
3. **NVIC**에서 `RTC_Alarm_IRQn` 인터럽트 요청 발생
4. MCU가 `RTC_Alarm_IRQHandler()` 함수로 벡터링
5. HAL 레벨에서 인터럽트 처리 (`HAL_RTC_AlarmIRQHandler()`)
6. 사용자 콜백 함수 (`HAL_RTC_AlarmAEventCallback()` 또는 `HAL_RTC_AlarmBEventCallback()`) 호출
7. 인터럽트 플래그 클리어 → IRQ 종료

---

3. 함수 구현 예시

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

이 함수는 HAL 계층에 정의된 핸들러를 호출한다.
`hrtc`는 전역으로 선언된 RTC 핸들 구조체 포인터이다.

---

4. HAL 내부 처리 과정

`HAL_RTC_AlarmIRQHandler()`는 다음과 같은 순서로 작동한다.

```
void HAL_RTC_AlarmIRQHandler(RTC_HandleTypeDef *hrtc)
{
    // 1. 알람 A 플래그 확인
    if(__HAL_RTC_ALARM_GET_FLAG(hrtc, RTC_FLAG_ALRAF) != RESET)
    {
        // 2. 알람 인터럽트 플래그 클리어
        __HAL_RTC_ALARM_CLEAR_FLAG(hrtc, RTC_FLAG_ALRAF);

        // 3. 사용자 콜백 호출
        HAL_RTC_AlarmAEventCallback(hrtc);
    }

    // 4. 알람 B 플래그 확인
    if(__HAL_RTC_ALARM_GET_FLAG(hrtc, RTC_FLAG_ALRBF) != RESET)
    {
        __HAL_RTC_ALARM_CLEAR_FLAG(hrtc, RTC_FLAG_ALRBF);
        HAL_RTC_AlarmBEventCallback(hrtc);
    }

    // 5. EXTI 라인(17번) 인터럽트 플래그 클리어
    __HAL_RTC_ALARM_EXTI_CLEAR_FLAG();
}
```

핵심은 RTC와 EXTI(외부 인터럽트 라인 17번)를 함께 관리하는 점이다.
RTC 알람은 내부적으로 **EXTI Line 17**을 통해 NVIC에 전달된다.

---

5. EXTI 라인 매핑

기능	EXTI 라인	NVIC IRQ 이름
RTC Alarm	17	`RTC_Alarm_IRQn`

따라서, RTC 알람이 발생하면 EXTI17 라인이 세트되고,
`HAL_RTC_AlarmIRQHandler()`는 이를 클리어하여 다음 알람 이벤트에 대비한다.

6. NVIC 설정 예시

```
void HAL_RTC_MspInit(RTC_HandleTypeDef* hrtc)
{
    if(hrtc->Instance == RTC)
    {
        HAL_NVIC_SetPriority(RTC_Alarm_IRQn, 0, 0);
        HAL_NVIC_EnableIRQ(RTC_Alarm_IRQn);
    }
}
```

이 설정은 RTC 알람 인터럽트를 NVIC에서 우선순위 0으로 활성화한다.

---

7. 사용자 콜백 함수

`HAL_RTC_AlarmIRQHandler()` 는 알람 이벤트 발생 시
자동으로 다음 콜백 중 하나를 호출한다:

```
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    printf("RTC Alarm A Triggered!\r\n");
    // 사용자 코드 삽입
}
```

또는

```
void HAL_RTC_AlarmBEventCallback(RTC_HandleTypeDef *hrtc)
{
    printf("RTC Alarm B Triggered!\r\n");
}
```

콜백 내부에서 센서 측정, OLED 업데이트, FreeRTOS Task Notify 등의 로직을 수행할 수 있다.

---

8. 인터럽트 후 플래그 클리어

- **RTC_FLAG_ALRAF / ALRBF** : 알람 발생 플래그
- **EXTI_PR(17번)** : 외부 인터럽트 펜딩 레지스터

HAL은 두 플래그를 모두 클리어하여 **중복 인터럽트 재발생**을 방지한다.
사용자가 직접 플래그를 조작할 필요는 없으며,
`HAL_RTC_AlarmIRQHandler()` 내부에서 자동 처리된다.

---

9. 디버깅 및 주의 사항

1. `HAL_RTC_SetAlarm_IT()` 호출 이후에 NVIC가 비활성화되어 있으면 인터럽트가 발생하지 않는다.
2. LSE가 안정화되지 않은 상태에서 RTC를 초기화하면 알람 타이밍이 부정확해질 수 있다.
3. 콜백 내부에서 긴 연산을 수행하면 ISR 지연이 발생하므로,
   Task Notify 또는 플래그 전달 후 메인 루프에서 처리하는 것이 바람직하다.
4. RTC 알람은 **전원 리셋 후에도 백업 도메인이 유지**될 수 있으므로,
   재부팅 시 불필요한 알람 인터럽트 발생 가능성에 유의해야 한다.

---

10. 요약

단계	설명
1	RTC 시간 설정 및 알람 등록 (`HAL_RTC_SetAlarm_IT()`)
2	알람 시간 도달 시 RTC 내부 ALRAF 세트
3	EXTI Line 17 트리거 → NVIC 호출
4	`RTC_Alarm_IRQHandler()` 실행 → `HAL_RTC_AlarmIRQHandler()` 호출
5	플래그 클리어 및 사용자 콜백 호출

✅ 결론

`RTC_Alarm_IRQHandler()` 는 **RTC 알람 이벤트의 진입점**이다.
이 핸들러는 HAL 레벨에서 인터럽트 플래그를 자동으로 관리하며,
사용자는 콜백 함수 내부에서 필요한 작업을 수행하면 된다.

저전력 시스템에서는 이 알람 IRQ를 통해 MCU를 **STOP/STANDBY 모드에서 자동 웨이크업**시켜,
주기적 데이터 측정 및 전력 최적화를 실현할 수 있다.

7.3 실습

- # `Set_RTC()` / `Set_Alarm()` 함수 구현

1. 개요

STM32에서 RTC(Real-Time Clock)를 이용하기 위해서는
1️⃣ RTC 시간 및 날짜를 설정하는 `Set_RTC()`,
2️⃣ 특정 시각에 알람을 발생시키는 `Set_Alarm()`
함수를 구현해야 한다.

이 두 함수는 `HAL_RTC_SetTime()`, `HAL_RTC_SetDate()`, `HAL_RTC_SetAlarm_IT()` API를 활용하며,
RTC가 **LSE(32.768 kHz)** 클럭을 기준으로 동작할 때 가장 높은 정확도를 제공한다.

---

2. RTC 초기화 구조

RTC는 다음과 같은 구성 요소를 가진다.

구성 요소	설명
LSE (Low-Speed External)	32.768 kHz 외부 크리스털
Asynch/Synch Prediv	1Hz 타이밍 생성용 분주기
Time/Date Registers	시간 및 날짜 저장
Alarm Registers	알람 시간 저장 (A/B)
Backup Domain	VBAT으로 전원 유지 가능

초기화는 한 번만 수행되며, 이후에는 백업 도메인에 의해 값이 유지된다.

---

3. `Set_RTC()` 함수 구현

(1) 목적

현재 시간을 설정하는 함수로, `RTC_TimeTypeDef` 및 `RTC_DateTypeDef` 구조체를 사용한다.

(2) 코드 예시

```
#include "rtc.h"
#include "stdio.h"

RTC_HandleTypeDef hrtc;

/**
  * @brief  RTC 시간/날짜 설정 함수
  * @param  hour, min, sec, year, month, date
  * @retval None
  */
void Set_RTC(uint8_t hour, uint8_t min, uint8_t sec,
             uint8_t year, uint8_t month, uint8_t date)
{
    RTC_TimeTypeDef sTime = {0};
    RTC_DateTypeDef sDate = {0};

    // 시간 설정
    sTime.Hours   = hour;
    sTime.Minutes = min;
    sTime.Seconds = sec;
    sTime.TimeFormat = RTC_HOURFORMAT12_AM;
    HAL_RTC_SetTime(&hrtc, &sTime, RTC_FORMAT_BIN);

    // 날짜 설정
    sDate.WeekDay = RTC_WEEKDAY_THURSDAY;  // 필요 시 수정
    sDate.Month   = month;
```

```
    sDate.Date    = date;
    sDate.Year    = year;
    HAL_RTC_SetDate(&hrtc, &sDate, RTC_FORMAT_BIN);

    printf("[RTC] Time Set → %02d:%02d:%02d\r\n", hour, min, sec);
    printf("[RTC] Date Set → %02d/%02d/20%02d\r\n", date, month, year);
}
```

(3) 주요 동작

- 시간 및 날짜를 각각 독립적으로 설정
- `RTC_FORMAT_BIN`은 BCD 변환 없이 이진수로 직접 입력
- 설정 시점의 백업 도메인 클리어 필요 시 `HAL_PWR_EnableBkUpAccess()` 호출

---

4. `Set_Alarm()` 함수 구현

(1) 목적

특정 시각에 인터럽트를 발생시켜 MCU가 이벤트를 처리하거나
저전력 모드에서 깨어나도록 트리거한다.

(2) 코드 예시

```
/**
  * @brief  RTC 알람 설정 함수 (Alarm A, 인터럽트 기반)
  * @param  hour, min, sec : 알람 시각
  * @retval None
  */
void Set_Alarm(uint8_t hour, uint8_t min, uint8_t sec)
{
    RTC_AlarmTypeDef sAlarm = {0};

    sAlarm.AlarmTime.Hours   = hour;
    sAlarm.AlarmTime.Minutes = min;
    sAlarm.AlarmTime.Seconds = sec;
    sAlarm.AlarmMask = RTC_ALARMMASK_DATEWEEKDAY;  // 날짜 비교 무시
    sAlarm.AlarmSubSecondMask = RTC_ALARMSUBSECONDMASK_ALL;
    sAlarm.AlarmDateWeekDaySel = RTC_ALARMDATEWEEKDAYSEL_DATE;
    sAlarm.AlarmDateWeekDay = 0;
    sAlarm.Alarm = RTC_ALARM_A;

    if (HAL_RTC_SetAlarm_IT(&hrtc, &sAlarm, RTC_FORMAT_BIN) == HAL_OK)
    {
        printf("[RTC] Alarm Set → %02d:%02d:%02d\r\n", hour, min, sec);
    }
    else
    {
        printf("[RTC] Alarm Set Failed\r\n");
    }
}
```

(3) NVIC 및 MSP 설정

RTC 알람 인터럽트는 **EXTI Line 17**을 통해 NVIC으로 전달되므로,
아래 설정이 필요하다.

```
void HAL_RTC_MspInit(RTC_HandleTypeDef* hrtc)
{
    if (hrtc->Instance == RTC)
    {
        HAL_NVIC_SetPriority(RTC_Alarm_IRQn, 0, 0);
        HAL_NVIC_EnableIRQ(RTC_Alarm_IRQn);
    }
}
```

---

5. 알람 인터럽트 핸들러

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

void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    printf("[RTC] Alarm Triggered!\r\n");
    // 사용자 동작 삽입 (예: 센서 측정, TaskNotify, Wake-up 등)
}
```

---

6. 주의 사항

1. **백업 도메인 보호 해제 필요**

   ```
   HAL_PWR_EnableBkUpAccess();
   ```

2. **LSE Ready 대기**
   RTC가 LSE 기반으로 설정되어야 안정적인 주기 제공.
   CubeMX 코드에서는 `MX_RTC_Init()` 내에서 `HAL_RCC_OscConfig()` 후
   LSE 안정화 확인 루틴 포함.

3. **STOP 모드 웨이크업**
   알람 인터럽트는 STOP/STANDBY 모드에서도 동작 가능.
   저전력 주기 측정 루프에 활용 가능.

4. **다음 알람 자동 재설정**
   RTC는 반복 알람 기능이 없으므로,
   콜백에서 `Set_Alarm()`을 다시 호출해야 주기적 동작 가능.

---

7. 전체 예제

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_RTC_Init();

    Set_RTC(12, 0, 0, 25, 11, 6);      // 2025-11-06 12:00:00
    Set_Alarm(12, 0, 10);              // 12:00:10 알람

    while (1)
    {
        HAL_Delay(1000);
        RTC_TimeTypeDef now;
        HAL_RTC_GetTime(&hrtc, &now, RTC_FORMAT_BIN);
        printf("[RTC] %02d:%02d:%02d\r\n", now.Hours, now.Minutes, now.Seconds);
    }
}
```

✅ 결론

- `Set_RTC()` : RTC 시간과 날짜를 직접 지정하여 시스템 기준 시각을 초기화
- `Set_Alarm()` : 특정 시각에 알람 인터럽트를 등록하고, 콜백 내에서 동작 수행
- 두 함수는 HAL API 기반으로 안정적이며, FreeRTOS나 저전력 시스템에서 주기적 동작 트리거에 적합하다.
- LSE 32.768kHz 사용 시 장기 안정도 ±20ppm, 약 ±1.7초/일 수준의 정확도 확보 가능.

• 슬립 → 알람 깨움 → 측정 루틴 수행

1. 개요

STM32의 **RTC Alarm 인터럽트**는 MCU가 **저전력(SLEEP/STOP/STANDBY)** 모드에 있을 때
자동으로 시스템을 깨우는 **Wake-up Source**로 사용할 수 있다.

이 구조를 이용하면 MCU는 다음 순서로 동작한다:

1️⃣ RTC 알람 예약
2️⃣ 저전력 모드 진입 (`HAL_PWR_EnterSTOPMode()` 등)
3️⃣ 알람 발생 시 자동 깨움
4️⃣ 깨어난 후 센서 측정 루틴 수행
5️⃣ 다음 주기 알람 재설정 및 재수면

이 주기를 통해 **주기적 측정/로깅/통신**을 초저전력으로 실행할 수 있다.

---

2. 동작 순서 요약

단계	설명
①	RTC를 LSE 32.768kHz 기준으로 초기화
②	`Set_Alarm()` 으로 다음 측정 시각 등록
③	불필요한 주변장치 전원 차단
④	`HAL_PWR_EnterSTOPMode()` 진입
⑤	RTC 알람 인터럽트 발생 시 자동 Wake-up
⑥	`HAL_RTC_AlarmAEventCallback()` 에서 측정 루틴 수행
⑦	다음 알람 재등록 후 다시 슬립

---

3. RTC 알람 기반 슬립 시스템 구조

```
/* RTC 알람 콜백 - 측정 루틴 수행 */
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef *hrtc)
{
    printf("[RTC] Alarm Triggered - Wake Up\r\n");

    // ==== 측정 루틴 ====
    Measure_All_Sensors();     // 예: 초음파 + 수위 + 무게

    // ==== 다음 알람 예약 ====
    RTC_TimeTypeDef now;
    HAL_RTC_GetTime(hrtc, &now, RTC_FORMAT_BIN);

    uint8_t next_min = (now.Minutes + 1) % 60; // 1분 후 알람
    Set_Alarm(now.Hours, next_min, 0);

    printf("[RTC] Next Alarm Set at %02d:%02d:00\r\n", now.Hours, next_min);
}
```

---

4. 슬립 진입 루틴

RTC 알람을 설정한 뒤 MCU를 **STOP 모드**로 전환한다.
STOP 모드는 클럭이 중지되고, **RTC / EXTI / I²C WAKE** 등만 유지된다.

```
void Enter_Sleep_Mode(void)
{
    printf("[PWR] Entering STOP Mode...\r\n");

    HAL_SuspendTick(); // SysTick 정지 (전력 절감)

    // STOP 모드 진입, RTC 알람으로만 깨어남
```

```
    HAL_PWR_EnterSTOPMode(PWR_LOWPOWERREGULATOR_ON, PWR_STOPENTRY_WFI);

    // 깨어난 후
    HAL_ResumeTick();
    SystemClock_Config(); // STOP 복귀 시 클럭 재설정

    printf("[PWR] Wake-up from STOP Mode\r\n");
}
```

5. 측정 루틴 예시

각 센서 드라이버를 순차적으로 호출하여 데이터를 취득하고,
OLED 또는 BLE로 전송할 수 있다.

```
void Measure_All_Sensors(void)
{
    float dist = Get_Ultrasonic_Distance();
    float level = Get_WaterLevel();
    float weight = HX711_Get_Weight();

    printf("[MEASURE] Distance=%.1f cm | Level=%.1f mm | Weight=%.1f g\r\n",
           dist, level, weight);

    // 필요 시 데이터 로깅 또는 BLE 전송
    // BLE_Send_Data(dist, level, weight);
}
```

6. 전체 동작 흐름 예시

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_RTC_Init();
    MX_GPIO_Init();
    MX_I2C1_Init();

    // RTC 기준 시각 설정
    Set_RTC(12, 0, 0, 25, 11, 6);

    // 1분 후 알람 등록
    Set_Alarm(12, 1, 0);

    while (1)
    {
        Enter_Sleep_Mode();  // 슬립 진입 → 알람 시 자동 복귀
        // 복귀 후 알람 콜백에서 측정 루틴 수행
    }
}
```

7. 주의 사항

1. **STOP 모드 복귀 시 클럭 재설정 필수**
   STOP에서 복귀하면 PLL이 꺼지므로, `SystemClock_Config()`를 다시 호출해야 한다.
2. **RTC는 LSE 클럭으로 구동**
   LSI는 ±5% 이상 오차 가능 → 장기 주기 측정에는 부적합.
3. **GPIO 상태 보존**
   STOP 모드 진입 전 필요한 핀을 `GPIO_MODE_ANALOG`으로 설정하면 누설전류 최소화.
4. **FreeRTOS와 병행 시**
   STOP 진입 전 `vTaskSuspendAll()` 또는 IDLE Hook을 통해 제어해야 함.

✅ 결론

- **RTC Alarm + STOP Mode** 조합은 초저전력 주기 동작의 핵심.
- MCU는 **슬립 중 소비 전류 수 μA 수준**,
  알람 발생 시 자동으로 복귀하여 **센서 측정 루틴 수행**.
- 이 구조는 **배터리 구동 환경**에서 수개월~수년의 작동 시간을 실현할 수 있다.

저전력 시스템 구현

1. 개요

STM32 기반 시스템에서 **저전력(Power Saving)** 은
배터리 구동 환경이나 장기 동작을 요구하는 센서 노드, IoT 기기에서 핵심적인 설계 요소다.

저전력 시스템 구현은 단순히 전류를 줄이는 것이 아니라,
동작 모드 제어 + 클럭 최적화 + 주변장치 관리 + 인터럽트 웨이크업 구조
를 유기적으로 결합하는 과정이다.

STM32는 다음과 같은 저전력 모드를 지원한다.

모드	소비 전류 (typ.)	특징
RUN	수 mA	CPU 활성, 모든 주변장치 구동
SLEEP	수백 μA	CPU 정지, 주변장치 유지
STOP	수 μA	메모리 유지, 대부분 클럭 정지
STANDBY	수십 nA	RAM 유지 안 함, RTC/Backup만 활성
SHUTDOWN	수 nA	모든 회로 차단, RTC까지 비활성 가능

---

2. 저전력 설계의 핵심 구성요소

저전력 시스템은 **하드웨어 설계, 클럭 설정, 전원 관리, 펌웨어 루프 구조**로 나뉜다.

(1) 하드웨어 설계

- LDO 대신 **DC/DC 변환기** 사용 시 전력 효율 향상
- 풀업/풀다운 저항 최적화로 누설전류 방지
- 불필요한 LED, 디버그 회로 제거

(2) 클럭 관리

- 고속 클럭(PLL, HSE) 사용 구간 최소화
- 유휴 상태에서는 **MSI/LSE** 저속 클럭으로 전환
- `HAL_RCCEx_PeriphCLKConfig()` 로 주변장치 클럭 분리

(3) 주변장치 제어

- 사용하지 않는 주변장치는 RCC 게이트를 비활성화

```
__HAL_RCC_I2C1_CLK_DISABLE();
__HAL_RCC_ADC1_CLK_DISABLE();
```

- GPIO는 **Analog 모드 + Pull 없음** 설정으로 누설 최소화

```
GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
GPIO_InitStruct.Pull = GPIO_NOPULL;
```

(4) 인터럽트 기반 설계

- **Polling 제거** → 인터럽트 기반 이벤트 처리
- RTC, EXTI, UART, I²C 이벤트가 MCU를 깨우는 구조로 설계

---

3. 저전력 동작 루프 구조

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_RTC_Init();
    MX_I2C1_Init();
    MX_GPIO_Init();

    printf("[SYS] Low Power System Init\r\n");

    // 초기 측정 및 알람 예약
    Measure_All_Sensors();
    Set_Alarm(12, 10, 0);  // 다음 주기 알람 등록

```

```
    while (1)
    {
        // 불필요한 주변장치 차단
        __HAL_RCC_I2C1_CLK_DISABLE();

        // STOP 모드 진입
        Enter_Sleep_Mode();

        // 알람 인터럽트로 복귀 시
        __HAL_RCC_I2C1_CLK_ENABLE();

        // 센서 측정 루틴 수행
        Measure_All_Sensors();

        // 다음 알람 재설정
        RTC_TimeTypeDef now;
        HAL_RTC_GetTime(&hrtc, &now, RTC_FORMAT_BIN);
        Set_Alarm(now.Hours, (now.Minutes + 1) % 60, 0);
    }
}
```

4. STOP 모드 진입 절차

```
void Enter_Sleep_Mode(void)
{
    HAL_SuspendTick();   // SysTick 정지 → 불필요한 wake 제거

    // GPIO 최소 전력 상태로 설정
    Set_All_GPIO_Analog();

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

    // 복귀 시 클럭 복원
    HAL_ResumeTick();
    SystemClock_Config();

    printf("[PWR] Wake-up from STOP\r\n");
}
```

5. GPIO 최소 전력 설정

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

    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();
    __HAL_RCC_GPIOC_CLK_ENABLE();
```

```
    GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Pin = GPIO_PIN_All;

    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
}
```

---

6. 전력 최적화 포인트

항목	권장 설정	설명
RTC 클럭	LSE 32.768 kHz	안정적이며 μA 수준 유지
SysTick	SLEEP 전 정지	불필요한 인터럽트 차단
GPIO	Analog, Pull None	누설전류 최소화
I²C/SPI/UART	필요 시만 Enable	RCC 게이트 제어
ADC/Timer	Sleep 중 Disable	클럭 공급 중단
Wake Source	RTC Alarm / EXTI	주기성 또는 외부 이벤트

---

7. 소비 전류 비교 예시 (STM32L0/L4 기준)

모드	조건	전류 (typ.)
RUN (16MHz, HSI)	모든 주변장치 활성	2.4 mA
SLEEP	CPU 정지	0.4 mA
STOP	LSE 유지, RTC on	5~10 μA
STANDBY	RTC on	0.8 μA
SHUTDOWN	RTC off	0.2 μA

---

8. FreeRTOS 환경에서의 저전력

FreeRTOS를 사용할 경우,
Tickless Idle Mode를 활성화하여 유휴 시 자동으로 STOP/SLEEP 진입이 가능하다.

```
#define configUSE_TICKLESS_IDLE 1
#define configEXPECTED_IDLE_TIME_BEFORE_SLEEP 5
```

`vPortSuppressTicksAndSleep()` 함수가 HAL STOP 모드를 호출하며,
RTC 또는 SysTick에 의해 재개된다.

---

9. 실측 예시 (STM32L476, LSE 기반)

구성	조건	평균 전류
주기적 센서 측정	10초 주기, STOP 모드 사용	**8.5 μA**
BLE 송신 포함	1분 주기, 5ms 송신	**25 μA**
지속 RUN 모드	센서 상시동작	**1.9 mA**

✅ 결론

- 저전력 시스템은 **Sleep/Stop 모드 + RTC Alarm Wake-up** 구조가 핵심.
- `RCC`, `GPIO`, `SysTick`, `RTC`를 정밀 제어하여 μA 수준 소비 전류 달성 가능.
- 정기 측정, BLE 통신, 데이터 로깅 등 모든 동작을 **주기적 활성 + 자동 수면 루프**로 설계하면
  배터리 기반 시스템에서 **수개월~수년 단위 동작**이 가능하다.