13. 확장 및 통신

13.1 SPI / UART 확장

- ## SPI 기반 IMU (MPU6050)

1. 개요

MPU6050은 3축 자이로스코프와 3축 가속도계를 통합한 6축 IMU(Inertial Measurement Unit) 센서로,
일반적으로 **I²C 인터페이스(0x68/0x69)** 를 사용하지만, 일부 호환 변형 혹은 SPI 래퍼 모듈을 통해 **SPI 모드**로도 구동할 수 있다.
SPI 모드는 I²C보다 빠른 데이터 전송속도를 제공하며, 버스 충돌이나 슬레이브 주소 충돌 위험이 없다.

STM32에서는 **HAL SPI 드라이버**를 통해 MPU6050의 레지스터를 직접 제어하고,
자이로 및 가속도 데이터를 읽어 필터링과 자세 계산(roll, pitch, yaw)에 활용할 수 있다.

---

2. SPI 하드웨어 연결

MPU6050 Pin	SPI 모드 연결	설명
VCC	3.3V	전원 공급
GND	GND	공통 접지
SCL	SCK	SPI 클럭
SDA	MOSI	Master → Slave 데이터
AD0	MISO	Slave → Master 데이터
CS	CS(GPIO)	Chip Select 제어

> 주의: MPU6050 기본형은 I²C 전용이다.
> SPI 버전은 레지스터 매핑 동일하지만 `CS` 핀과 `MISO` / `MOSI` 라인 구성 지원 모듈(예: GY-521 SPI 개조형)을 사용해야 한다.

---

3. SPI 설정 (CubeMX 기준)

- **Mode**: Full Duplex Master
- **Clock Polarity (CPOL)**: Low
- **Clock Phase (CPHA)**: 1 Edge
- **Data Size**: 8-bit
- **Baud Rate Prescaler**: 8 ~ 32 (1~2 MHz 권장)
- **First Bit**: MSB First
- **NSS**: Software (GPIO 제어)

MPU6050의 SPI는 **Mode 0 (CPOL=0, CPHA=0)** 에서 동작한다.

4. SPI 통신 규약

SPI에서 레지스터 접근 시,

- **Bit7 (MSB)** : `1`이면 Read, `0`이면 Write
- 나머지 7비트 : Register Address

예)

동작	전송 바이트	설명
읽기	`[0x3B | 0x80]`	
쓰기	`[0x6B & 0x7F]`	PWR_MGMT_1 설정

5. 주요 레지스터

주소	이름	설명
0x6B	PWR_MGMT_1	전원 관리 / 클럭 설정
0x1B	GYRO_CONFIG	자이로 범위 (±250~2000°/s)
0x1C	ACCEL_CONFIG	가속도 범위 (±2~16g)
0x3B~0x40	ACCEL_X/Y/Z	가속도 출력 (High, Low)
0x43~0x48	GYRO_X/Y/Z	자이로 출력 (High, Low)
0x75	WHO_AM_I	기본값 0x68

6. 코드 예시

```
#define MPU_CS_LOW()    HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4, GPIO_PIN_RESET)
#define MPU_CS_HIGH()   HAL_GPIO_WritePin(GPIOA, GPIO_PIN_4, GPIO_PIN_SET)

extern SPI_HandleTypeDef hspi1;

uint8_t MPU_ReadReg(uint8_t reg)
{
    uint8_t tx[2], rx[2];
    tx[0] = reg | 0x80;  // Read flag
    MPU_CS_LOW();
    HAL_SPI_Transmit(&hspi1, tx, 1, HAL_MAX_DELAY);
    HAL_SPI_Receive(&hspi1, &rx[1], 1, HAL_MAX_DELAY);
    MPU_CS_HIGH();
    return rx[1];
}

void MPU_WriteReg(uint8_t reg, uint8_t data)
```

```
{
    uint8_t tx[2];
    tx[0] = reg & 0x7F;  // Write flag
    tx[1] = data;
    MPU_CS_LOW();
    HAL_SPI_Transmit(&hspi1, tx, 2, HAL_MAX_DELAY);
    MPU_CS_HIGH();
}

void MPU_Init(void)
{
    HAL_Delay(100);
    MPU_WriteReg(0x6B, 0x00);  // Wake up
    MPU_WriteReg(0x1B, 0x00);  // Gyro ±250°/s
    MPU_WriteReg(0x1C, 0x00);  // Accel ±2g
}

void MPU_ReadAccelGyro(int16_t *accel, int16_t *gyro)
{
    uint8_t tx = 0x3B | 0x80;
    uint8_t rx[14];

    MPU_CS_LOW();
    HAL_SPI_Transmit(&hspi1, &tx, 1, HAL_MAX_DELAY);
    HAL_SPI_Receive(&hspi1, rx, 14, HAL_MAX_DELAY);
    MPU_CS_HIGH();

    accel[0] = (rx[0] << 8) | rx[1];
    accel[1] = (rx[2] << 8) | rx[3];
    accel[2] = (rx[4] << 8) | rx[5];
    gyro[0]  = (rx[8] << 8) | rx[9];
    gyro[1]  = (rx[10] << 8) | rx[11];
    gyro[2]  = (rx[12] << 8) | rx[13];
}
```

---

7. 데이터 변환

기본 감도는 다음과 같다:

설정	자이로 감도	가속도 감도
±250°/s	131 LSB/°/s	16384 LSB/g
±500°/s	65.5 LSB/°/s	8192 LSB/g
±1000°/s	32.8 LSB/°/s	4096 LSB/g
±2000°/s	16.4 LSB/°/s	2048 LSB/g

```
float ax = accel[0] / 16384.0f;
float ay = accel[1] / 16384.0f;
float az = accel[2] / 16384.0f;

float gx = gyro[0] / 131.0f;
float gy = gyro[1] / 131.0f;
float gz = gyro[2] / 131.0f;
```

---

8. 필터링 및 자세 계산

- **보정(Offset Calibration)**: 정지 상태 평균값을 0으로 맞춤
- **Exponential Smoothing**: 잡음 완화
- **Complementary Filter** 또는 **Kalman Filter**로 Roll/Pitch 계산

```
float pitch = atan2(ax, sqrt(ay*ay + az*az)) * 180.0 / M_PI;
float roll  = atan2(ay, sqrt(ax*ax + az*az)) * 180.0 / M_PI;
```

---

9. FreeRTOS Task 예시

```
void IMUTask(void *argument)
{
    int16_t accel[3], gyro[3];
    MPU_Init();

    for(;;)
    {
        MPU_ReadAccelGyro(accel, gyro);
        printf("AX:%d AY:%d AZ:%d  GX:%d GY:%d GZ:%d\n",
               accel[0], accel[1], accel[2],
               gyro[0], gyro[1], gyro[2]);
        osDelay(50);
    }
}
```

---

10. 요약

항목	내용
인터페이스	SPI Mode 0
장점	빠른 전송속도, 충돌 없음
주요 레지스터	PWR_MGMT_1, ACCEL_CONFIG, GYRO_CONFIG
측정 데이터	가속도(3축), 자이로(3축)
권장 속도	1~2 MHz

항목	내용
필터링	보정 + Complementary/Kalman
FreeRTOS 연동	DMA 또는 주기 Task로 구현

SPI 기반 MPU6050 구동은 I²C보다 빠르고 안정적이며,
다중 센서 버스 구조나 실시간 제어 루프에서 **저지연 IMU 데이터 취득**에 유리하다.

- # UART 기반 BLE/Wi-Fi 연동

1. 개요

UART 기반 무선 통신은 STM32와 BLE(Bluetooth Low Energy) 또는 Wi-Fi 모듈 간의 데이터 교환을 담당한다.
대표적인 예로 **ESP32, HM-10, HC-05(Bluetooth), ESP8266, ESP32-S3(Wi-Fi/BLE)** 등이 있다.
UART 인터페이스는 **비동기 직렬 통신(Asynchronous Serial Communication)** 형태로, 송신(TX)과 수신(RX) 라인만으로 간단히 구현할 수 있다.

---

2. 하드웨어 구성

항목	STM32 핀	모듈 핀	설명
TX	USARTx_TX	RXD	STM32 → BLE/Wi-Fi 송신
RX	USARTx_RX	TXD	BLE/Wi-Fi → STM32 수신
EN/CH_PD	GPIO	HIGH 유지	모듈 전원 활성화
GND	GND	GND	공통 접지
VCC	3.3V	VCC	전원 공급 (ESP8266은 최소 300mA 필요)

UART 속도는 보통 **115200 bps** 또는 **9600 bps**로 설정하며, 양측의 baud rate 일치가 필수적이다.

---

3. 초기화 코드 (HAL 기반)

```
UART_HandleTypeDef huart2;

void MX_USART2_UART_Init(void)
{
    huart2.Instance = USART2;
    huart2.Init.BaudRate = 115200;
    huart2.Init.WordLength = UART_WORDLENGTH_8B;
    huart2.Init.StopBits = UART_STOPBITS_1;
    huart2.Init.Parity = UART_PARITY_NONE;
    huart2.Init.Mode = UART_MODE_TX_RX;
    huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart2.Init.OverSampling = UART_OVERSAMPLING_16;
    HAL_UART_Init(&huart2);
```

```
}
```

4. 데이터 송신 및 수신

```
uint8_t tx_buf[] = "AT\r\n";
uint8_t rx_buf[64];

HAL_UART_Transmit(&huart2, tx_buf, strlen((char*)tx_buf), HAL_MAX_DELAY);
HAL_UART_Receive(&huart2, rx_buf, sizeof(rx_buf), 1000);
printf("RX: %s\n", rx_buf);
```

BLE 또는 Wi-Fi 모듈은 **AT Command Set**을 통해 제어된다.
예:

- `"AT"` : 응답 확인
- `"AT+GMR"` : 펌웨어 버전 조회
- `"AT+CWMODE=1"` : STA 모드 설정
- `"AT+CWJAP=\"SSID\",\"PASS\""` : Wi-Fi 접속

5. UART 수신 인터럽트 기반 처리

```
uint8_t rx_data;

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if (huart->Instance == USART2)
    {
        ProcessBLECommand(rx_data);
        HAL_UART_Receive_IT(&huart2, &rx_data, 1);
    }
}
```

비동기 수신을 위해 `HAL_UART_Receive_IT()` 또는 DMA 수신(`HAL_UART_Receive_DMA()`)을 활용할 수 있다.
이 방식은 메인 루프를 차단하지 않고 실시간 BLE/Wi-Fi 데이터를 처리할 수 있다.

6. FreeRTOS Task 연동

```
void CommTask(void *argument)
{
    uint8_t msg[64];
    for(;;)
    {
        if (xQueueReceive(uartRxQueue, msg, portMAX_DELAY) == pdTRUE)
        {
            HandleWiFiCommand(msg);
        }
        osDelay(10);
    }
}
```

UART 인터럽트에서 수신된 데이터를 **Queue**를 통해 `CommTask`로 전달하면
BLE/Wi-Fi 통신을 다른 Task와 병렬로 안정적으로 처리할 수 있다.

---

7. Wi-Fi / BLE 통신 응용 예시

응용	설명
BLE Beacon 송신	STM32 → BLE 모듈 → 스마트폰 앱
Wi-Fi MQTT Publish	STM32 → ESP8266 → MQTT 브로커
원격 펌프 제어	BLE/Wi-Fi 명령 수신 → GPIO 제어
데이터 로깅	센서 데이터 → UART 전송 → 클라우드 업로드

---

8. 디버깅 및 주의사항

- 전원 공급 시 전압 강하에 유의 (특히 ESP8266/ESP32는 순간 피크 전류 높음)
- 모듈의 **AT 버전** 및 **펌웨어**에 따라 명령어 차이가 존재
- UART RX 버퍼 오버플로 방지를 위해 DMA 또는 순환 버퍼(Ring Buffer) 사용 권장
- 통신 중 에러(`HAL_UART_ERROR_FE`, `HAL_UART_ERROR_ORE`) 발생 시 반드시 Clear 처리

이 구조를 기반으로 BLE/Wi-Fi 통신을 RTOS 환경에 안전하게 통합하면
원격 제어, 센서 모니터링, OTA 업데이트 등의 고급 기능을 구현할 수 있다.

13.2 EEPROM / Flash 데이터 관리

- # Calibration / Log 저장

1. 개요

센서 측정 시스템에서 **Calibration(보정 데이터)** 및 **운영 Log(측정 이력, 오류 기록)** 저장은
정확도 유지와 진단 기능을 위해 필수적이다.
STM32에서는 비휘발성 메모리인 **EEPROM**, **Flash**, 또는 **외부 I²C EEPROM (예: 24C02)** 을 사용하여
보정 상수와 로그 데이터를 영구 저장할 수 있다.

---

2. 보정 데이터 구조 예시

```
typedef struct {
    float offset;      // 오프셋 보정값
    float scale;       // 스케일(감도) 보정값
    float ref_weight;  // 기준 중량
    uint32_t crc;      // 무결성 검사용 CRC
} CalibrationData_t;

CalibrationData_t cal_data;
```

보정 데이터는 일반적으로 **오프셋**, **스케일**, **기준값**, **CRC** 등을 포함한다.
CRC는 데이터 손상 여부를 부팅 시 검증하는 용도로 사용한다.

---

3. EEPROM 저장 및 로드

(1) 쓰기

```
void SaveCalibration(void)
{
    uint8_t *p = (uint8_t*)&cal_data;
    for (uint16_t i = 0; i < sizeof(CalibrationData_t); i++) {
        HAL_I2C_Mem_Write(&hi2c1, EEPROM_ADDR, i, I2C_MEMADD_SIZE_8BIT, &p[i], 1,
HAL_MAX_DELAY);
        HAL_Delay(5); // EEPROM 쓰기 시간 보장
    }
}
```

(2) 읽기

```
void LoadCalibration(void)
{
    uint8_t *p = (uint8_t*)&cal_data;
    for (uint16_t i = 0; i < sizeof(CalibrationData_t); i++) {
        HAL_I2C_Mem_Read(&hi2c1, EEPROM_ADDR, i, I2C_MEMADD_SIZE_8BIT, &p[i], 1, HAL_MAX_DELAY);
    }
}
```

EEPROM은 1바이트 단위 접근이 가능하지만, 쓰기 시간(약 5ms)을 반드시 고려해야 한다.
쓰기 중 전원 차단 시 데이터 손상 가능성이 있으므로,
전원 감시 회로(Power Fail Detection) 또는 **Double Buffering 방식**으로 안정성을 확보한다.

---

4. Flash 메모리 직접 저장 (EEPROM 미사용 시)

STM32F103과 같은 MCU는 별도의 EEPROM이 없기 때문에 **내부 Flash 영역**을 사용할 수 있다.

```
#define FLASH_USER_START_ADDR  ((uint32_t)0x0801F800)  // 마지막 섹터 예시
#define DATA_SIZE sizeof(CalibrationData_t)

void Flash_SaveCalibration(void)
{
    HAL_FLASH_Unlock();
    FLASH_EraseInitTypeDef erase = { .TypeErase = FLASH_TYPEERASE_PAGES, .PageAddress = FLASH_USER_START_ADDR, .NbPages = 1 };
    uint32_t pageError = 0;
    HAL_FLASHEx_Erase(&erase, &pageError);

    uint32_t *data = (uint32_t*)&cal_data;
    for (uint32_t i = 0; i < DATA_SIZE / 4; i++)
        HAL_FLASH_Program(FLASH_TYPEPROGRAM_WORD, FLASH_USER_START_ADDR + i * 4, data[i]);

    HAL_FLASH_Lock();
}

void Flash_LoadCalibration(void)
{
    memcpy(&cal_data, (void*)FLASH_USER_START_ADDR, DATA_SIZE);
}
```

Flash 쓰기 시에는 **Page Erase 후 Write** 절차가 필수이며,
Erase 단위(페이지 크기)에 맞춰 구조체 크기를 정렬해야 한다.

---

5. Log 저장 구조

```
typedef struct {
    uint32_t timestamp;
    float level;
    float weight;
    uint8_t errorCode;
} LogEntry_t;

#define LOG_MAX_ENTRIES  128
LogEntry_t log_buffer[LOG_MAX_ENTRIES];
uint16_t log_index = 0;

void SaveLog(float level, float weight, uint8_t error)
{
    if (log_index < LOG_MAX_ENTRIES) {
        log_buffer[log_index].timestamp = HAL_GetTick();
        log_buffer[log_index].level = level;
        log_buffer[log_index].weight = weight;
        log_buffer[log_index].errorCode = error;
        log_index++;
    }
}
```

Log는 RAM 상에 임시 보관 후, 주기적으로 EEPROM/Flash에 덤프하거나
UART/BLE/Wi-Fi로 전송해 외부 서버에 저장할 수 있다.

---

6. 무결성 검증 (CRC)

보정 데이터의 신뢰성을 확보하기 위해 **CRC32 또는 Checksum**을 저장 시 함께 기록한다.

```
uint32_t CalcCRC32(uint8_t *data, uint32_t len)
{
    uint32_t crc = 0xFFFFFFFF;
    for (uint32_t i = 0; i < len; i++) {
        crc ^= data[i];
        for (uint8_t j = 0; j < 8; j++)
            crc = (crc >> 1) ^ (0xEDB88320 & -(crc & 1));
    }
    return ~crc;
}
```

시작 시 EEPROM/Flash에서 읽은 데이터의 CRC를 계산해 저장된 값과 비교한다.
일치하지 않으면 기본 보정값으로 초기화하고 재보정을 유도한다.

---

7. FreeRTOS Task 통합 예시

```
void LogTask(void *argument)
{
    for(;;)
    {
        if (xQueueReceive(logQueue, &log_entry, portMAX_DELAY) == pdTRUE)
            SaveLog(log_entry.level, log_entry.weight, log_entry.error);
        osDelay(1000);
    }
}
```

보정 및 로그 저장을 주기적으로 수행하는 별도 Task를 운영하면
 측정 Task의 실시간성을 저하시키지 않고 안정적인 데이터 관리가 가능하다.

---

8. 요약

항목	주요 내용
보정 데이터	Offset, Scale, 기준값, CRC
저장 매체	EEPROM 또는 Flash
로그 데이터	시간, 수위, 무게, 오류코드
무결성 검증	CRC32, Double Buffer
FreeRTOS 연동	LogTask, Queue 기반 처리

정확한 보정과 지속적 로그 저장은 장기 운용 시스템의 신뢰성 확보에 핵심적이다.
 EEPROM/Flash 구조, 전원 이벤트 대응, 무결성 검증이 결합되어야 완전한 영구 데이터 관리가 가능하다.

- # Flash Page Write/Erase 실습

1. 개요

STM32의 내부 Flash 메모리는 프로그램 코드뿐 아니라
 사용자 데이터 저장 영역(예: Calibration, 설정값) 으로도 활용할 수 있다.
 EEPROM이 없는 MCU(STM32F1, F4 계열 등)는 Flash를 EEPROM 대용으로 사용하는 경우가 많다.

Flash는 **Erase 단위가 페이지(Page)** 단위이므로,
 데이터 변경 시 반드시 Erase → Write 절차를 거쳐야 한다.

---

2. 기본 원리

- Flash는 **1Page = 1~2KB (MCU별 상이)**
- **Erase 시 전체 페이지가 0xFF로 초기화**
- **Write는 1Word(4Byte)** 단위로 가능

- 한 번 '0'으로 쓴 비트를 다시 '1'로 변경할 수 없음 → 반드시 Erase 필요
- Flash 접근 시 **HAL_FLASH_Unlock() / HAL_FLASH_Lock()** 으로 보호 제어

---

3. Flash 주소 정의

```
#define FLASH_PAGE_SIZE         1024U
#define FLASH_USER_START_ADDR   ((uint32_t)0x0801F800) // 마지막 2KB 페이지 예시
#define FLASH_USER_END_ADDR     ((uint32_t)0x08020000)
```

주의:

- 애플리케이션 코드가 사용하는 주소 영역을 침범하지 않도록 주의해야 함.
- `LD` 스크립트(`STM32F103C8_FLASH.ld`)에서 Flash 크기 확인 후 남는 영역을 사용.

---

4. Flash 쓰기 함수

```
#include "stm32f1xx_hal.h"

void Flash_Write(uint32_t addr, uint32_t *data, uint32_t length)
{
    HAL_FLASH_Unlock();

    for (uint32_t i = 0; i < length; i++)
    {
        HAL_FLASH_Program(FLASH_TYPEPROGRAM_WORD, addr + (i * 4), data[i]);
    }

    HAL_FLASH_Lock();
}
```

---

5. Flash 읽기 함수

```
void Flash_Read(uint32_t addr, uint32_t *data, uint32_t length)
{
    for (uint32_t i = 0; i < length; i++)
    {
        data[i] = *(volatile uint32_t *)(addr + (i * 4));
    }
}
```

---

6. Flash 페이지 삭제 함수

```
void Flash_Erase_Page(uint32_t pageAddress)
{
    HAL_FLASH_Unlock();

    FLASH_EraseInitTypeDef eraseInitStruct;
    uint32_t pageError = 0;

    eraseInitStruct.TypeErase   = FLASH_TYPEERASE_PAGES;
    eraseInitStruct.PageAddress = pageAddress;
    eraseInitStruct.NbPages     = 1;

    HAL_FLASHEx_Erase(&eraseInitStruct, &pageError);

    HAL_FLASH_Lock();
}
```

---

7. 실습 예제

```
void Flash_Test(void)
{
    uint32_t writeData[4] = {0x12345678, 0xABCDEF00, 0x55555555, 0xAAAAAAAA};
    uint32_t readData[4]  = {0};

    // 1. 페이지 지우기
    Flash_Erase_Page(FLASH_USER_START_ADDR);

    // 2. 데이터 쓰기
    Flash_Write(FLASH_USER_START_ADDR, writeData, 4);

    // 3. 데이터 읽기
    Flash_Read(FLASH_USER_START_ADDR, readData, 4);

    // 4. 검증
    for (int i = 0; i < 4; i++)
        printf("Read[%d] = 0x%08lX\r\n", i, readData[i]);
}
```

출력 예시:

```
Read[0] = 0x12345678
Read[1] = 0xABCDEF00
Read[2] = 0x55555555
Read[3] = 0xAAAAAAAA
```

---

8. 주의사항

항목	설명
Erase 단위	최소 1Page, 개별 Byte 삭제 불가
수명(Lifetime)	약 10,000회 쓰기 가능 (EEPROM보다 낮음)
전원 안정성	쓰기 중 전원 차단 시 데이터 손상 위험
Interrupt	Flash erase/write 중 인터럽트 지연 발생 가능
Protection	옵션바이트(OB) 설정 시 Write 보호 가능

---

9. 실습 확장 아이디어

- Calibration 구조체(`offset`, `scale`)를 Flash에 저장
- Boot 시 Flash에서 값 로드
- UART 명령으로 보정값 변경 및 재저장
- CRC32 무결성 검증 추가

---

10. 정리

단계	내용
①	Flash Page 주소 정의
②	`HAL_FLASH_Unlock()` 수행
③	`HAL_FLASHEx_Erase()` 로 페이지 초기화
④	`HAL_FLASH_Program()` 으로 데이터 기록
⑤	직접 메모리 접근으로 데이터 검증
⑥	완료 후 `HAL_FLASH_Lock()`

Flash Page Write/Erase 실습은 **비휘발성 데이터 관리**,
즉 EEPROM 대체 구현의 핵심이다.
이 과정을 통해 MCU 내부 Flash를 안전하게 제어하고,
센서 보정값이나 시스템 설정을 영구적으로 저장할 수 있다.

13.3 SD카드 로깅

- # SPI-SD 인터페이스

1. 개요

SD 카드(Secure Digital Card)는 비휘발성 저장매체로, STM32 등 임베디드 시스템에서
데이터 로깅, 펌웨어 업데이트, 이미지/사운드 저장 등의 용도로 사용된다.
SD 카드는 기본적으로 **SDIO 모드**와 **SPI 모드**를 모두 지원한다.

SPI 모드는 핀 수가 적고(HW SDIO 불필요), 범용 MCU에서 쉽게 구현 가능하다는 장점이 있다.
속도는 다소 낮지만(수백 kB/s~수MB/s), 대부분의 센서 로깅 및 설정 파일 관리에는 충분하다.

---

2. SPI-SD 핀 구성

SD 카드 핀	SPI 신호	STM32 핀 예시	설명
CS	NSS	PA4	카드 선택 (Low = Active)
DI	MOSI	PA7	MCU → SD (데이터 전송)
DO	MISO	PA6	SD → MCU (데이터 수신)
CLK	SCK	PA5	SPI 클럭
VCC	3.3V	-	전원
GND	GND	-	공통 접지

> ⚠️ SD 카드는 3.3V 동작이므로 5V MCU 사용 시 **레벨시프터** 필요.
> SPI 속도는 초기화 시 400 kHz 이하, 이후 최대 25 MHz까지 가능.

---

3. 초기화 절차 (SPI 모드)

SD 카드는 전원 인가 후 기본적으로 SD 모드로 부팅된다.
SPI 모드로 진입하기 위해 다음 시퀀스를 수행해야 한다.

1. **CS = High**, **CLK ≥ 74 pulses** (Dummy Clock 74개 이상)
2. **CMD0 (GO_IDLE_STATE)** 전송 → 응답 R1 = 0x01
3. **CMD8 (SEND_IF_COND)** 전송 → SDHC 여부 확인
4. **ACMD41 (SD_SEND_OP_COND)** 반복 → R1 = 0x00 될 때까지 (초기화 완료)
5. **CMD58 (READ_OCR)** → 카드 전압 범위 확인
6. 이후 SPI 속도 상향 (최대 25MHz)

---

4. 명령 구조 (SPI Command Frame)

비트	의미
[7]	Always '0' (Start bit)
[6:0]	Command index (예: CMD17 → 17)
[39:8]	Argument (예: Sector address)
[7:1]	CRC7
[0]	End bit = 1

- 예: `CMD17(READ_SINGLE_BLOCK)` = `0x51`, argument = sector address × 512
- 응답(Response) 형식: R1, R3, R7 (1~5 byte)

---

5. 주요 명령 (SPI 모드)

명령	코드	설명
CMD0	0x40	카드 리셋 및 SPI 모드 진입
CMD8	0x48	SDHC 지원 여부 확인
CMD17	0x51	단일 블록(512B) 읽기
CMD24	0x58	단일 블록 쓰기
CMD55	0x77	다음 명령을 ACMD로 지정
ACMD41	0x69	카드 초기화
CMD58	0x7A	OCR 레지스터 읽기

---

6. HAL SPI 예제 코드

```
uint8_t SD_SPI_SendByte(uint8_t data)
{
    uint8_t rx;
    HAL_SPI_TransmitReceive(&hspi1, &data, &rx, 1, HAL_MAX_DELAY);
    return rx;
}

void SD_SPI_SendCommand(uint8_t cmd, uint32_t arg, uint8_t crc)
{
    uint8_t packet[6];
    packet[0] = 0x40 | cmd;
    packet[1] = (arg >> 24);
    packet[2] = (arg >> 16);
```

```
    packet[3] = (arg >> 8);
    packet[4] = (arg);
    packet[5] = crc | 0x01;
    HAL_SPI_Transmit(&hspi1, packet, 6, HAL_MAX_DELAY);
}
```

---

7. FatFs 파일 시스템 통합

STM32CubeIDE에서 **Middleware** → **FATFS**를 활성화하고,
Interface → **SPI**로 설정하면 자동으로 SD SPI 드라이버가 연결된다.

주요 함수

```
f_mount(&SDFatFS, SDPath, 1);                // 파일시스템 마운트
f_open(&file, "log.txt", FA_WRITE | FA_CREATE_ALWAYS);
f_printf(&file, "Sensor=%.2f\r\n", value);
f_close(&file);
```

FatFs는 내부적으로 `diskio.c`의 SPI 함수를 통해 SD 카드 접근을 수행한다.

---

8. SD 카드 데이터 구조

영역	설명
Boot Sector	FAT 파라미터, 볼륨 정보
FAT Table	파일 할당 테이블
Root Directory	파일/폴더 목록
Data Region	실제 파일 데이터 저장 영역

> 일반적인 SD 카드는 **512 Byte/섹터** 단위 접근.
> FAT32 기준 2GB 이상 카드 지원 가능.

---

9. 성능 및 최적화

- DMA를 이용한 SPI 전송 (`HAL_SPI_Transmit_DMA`) 으로 CPU 부하 감소
- Block 단위 버퍼링 (512B) 사용
- `f_sync()` 호출 최소화 (Flash 수명 관리)
- 파일 append 시 `FA_OPEN_APPEND` 사용
- SPI 클럭: 초기화 시 400 kHz, 이후 8~25 MHz

---

10. 오류 처리

오류	원인	조치
`FR_DISK_ERR`	SPI 응답 없음, CMD 실패	배선 점검, 전원 확인
`FR_NOT_READY`	초기화 실패	CMD0~CMD58 시퀀스 재시도
`FR_NO_FILE`	파일 없음	경로 또는 이름 확인
`FR_WRITE_PROTECTED`	SD Write Protect 핀 활성	Write 보호 해제

---

11. 요약

항목	내용
인터페이스	SPI (4-wire)
초기화 절차	CMD0 → CMD8 → ACMD41 → CMD58
블록 크기	512 Byte
파일 시스템	FATFS (SPI Mode 지원)
SPI 속도	400kHz(Init) → 25MHz(Max)
응용	로깅, 설정 저장, OTA, 데이터 백업

SPI-SD 인터페이스는 **범용 MCU 환경에서 손쉽게 대용량 저장 기능을 구현**할 수 있는 방법이다.
Flash 기반 보정보관보다 안정적이며, **FATFS와 결합하면 PC 호환 데이터 로깅 시스템**으로 확장 가능하다.

- # FATFS로 데이터 저장

1. 개요

FATFS는 ChanN님이 개발한 경량 파일시스템으로,
STM32CubeMX에서 **Middleware → FATFS**를 통해 간단히 통합할 수 있다.
SPI 또는 SDIO 기반 SD 카드, USB 메모리, 외부 Flash 등 다양한 저장매체에 사용 가능하다.

FATFS는 PC의 FAT12/16/32 형식을 그대로 사용하므로,
MCU에서 저장한 데이터를 **PC에 SD 카드로 직접 연결해 읽을 수 있는 장점**이 있다.

---

2. CubeMX 설정

(1) Middleware → FATFS 활성화
(2) Interface → `SPI` (또는 `SDIO`) 선택
(3) 사용 SPI 포트 및 CS 핀 설정
(4) 필요 시 DMA 전송 활성화
(5) “Generate Code” 실행

CubeMX는 자동으로 다음 파일을 생성한다.

- `fatfs.c / fatfs.h`
- `sd_diskio.c` (SPI/SDIO 드라이버)

---

3. 주요 함수 흐름

단계	함수	설명
1	`f_mount()`	파일시스템 마운트
2	`f_open()`	파일 열기 또는 생성
3	`f_write()` / `f_printf()`	데이터 쓰기
4	`f_read()`	데이터 읽기
5	`f_close()`	파일 닫기 (Flush)
6	`f_unmount()`	마운트 해제

---

4. 기본 코드 예제

```
#include "fatfs.h"
#include "stdio.h"

FATFS fs;
FIL file;
FRESULT res;
UINT bw;
char buffer[64];

void SD_FATFS_Test(void)
{
    // 1. 파일시스템 마운트
    res = f_mount(&fs, SDPath, 1);
    if (res != FR_OK) {
        printf("Mount failed: %d\r\n", res);
        return;
    }

    // 2. 파일 생성/열기
    res = f_open(&file, "log.txt", FA_WRITE | FA_OPEN_APPEND);
    if (res != FR_OK) {
        printf("Open failed: %d\r\n", res);
        return;
    }

    // 3. 데이터 기록
    sprintf(buffer, "Temp=%.2f, Level=%.1f\r\n", 23.5, 12.7);
```

```
    res = f_write(&file, buffer, strlen(buffer), &bw);
    if (res == FR_OK) {
        printf("Write OK (%d bytes)\r\n", bw);
    }

    // 4. 파일 닫기
    f_close(&file);

    // 5. 마운트 해제
    f_mount(NULL, SDPath, 1);
}
```

> 💡 `FA_OPEN_APPEND` 옵션을 사용하면 자동으로 파일 끝에 추가 기록 가능.

---

5. f_printf() 활용

FatFS는 `f_printf()` 함수도 지원하므로 문자열 포맷 저장이 간편하다.

```
f_printf(&file, "Voltage: %.2f V, Current: %.3f A\r\n", v, i);
```

> 단, `FF_USE_STRFUNC = 1` 이상이어야 하며, CubeIDE 설정의 `ffconf.h`에서 변경 가능.

---

6. 디렉토리 관리

```
DIR dir;
FILINFO fno;

f_opendir(&dir, "/data");
while (1) {
    res = f_readdir(&dir, &fno);
    if (res != FR_OK || fno.fname[0] == 0) break;
    printf("%s\r\n", fno.fname);
}
f_closedir(&dir);
```

---

7. 데이터 로깅 구조 예시

시간	수위(cm)	온도(°C)	유량(L/min)
2025-11-09 10:00	12.3	24.1	3.8
2025-11-09 10:05	12.5	24.3	3.9

```
f_printf(&file, "%04d-%02d-%02d %02d:%02d, %.1f, %.1f, %.2f\r\n",
         year, mon, day, hour, min, level, temp, flow);
```

---

8. 주기적 저장 (FreeRTOS Task)

```
void LogTask(void *argument)
{
    for (;;) {
        SensorData_t data = GetSensorData();
        if (f_mount(&fs, SDPath, 1) == FR_OK) {
            if (f_open(&file, "log.txt", FA_WRITE | FA_OPEN_APPEND) == FR_OK) {
                f_printf(&file, "%.2f,%.2f,%.2f\r\n", data.temp, data.level,
data.flow);
                f_close(&file);
            }
            f_mount(NULL, SDPath, 1);
        }
        osDelay(60000);  // 1분마다 기록
    }
}
```

---

9. FATFS 구조 및 버퍼링

- FATFS는 내부적으로 **512바이트 단위 버퍼(섹터)** 로 동작.
- 파일 단위 캐싱은 없으므로, 쓰기 후 반드시 `f_close()` 또는 `f_sync()` 호출해야 데이터가 반영됨.
- SD 카드 수명(Write Cycle)을 고려해 **버퍼에 일정량 누적 후 저장**하는 것이 바람직하다.

---

10. 오류 처리

에러 코드	의미	조치
`FR_DISK_ERR`	물리적 오류	SPI 배선, 전원 확인
`FR_NOT_READY`	SD 초기화 실패	재시도 또는 카드 재삽입
`FR_WRITE_PROTECTED`	쓰기 금지	SD Lock Switch 확인
`FR_NOT_ENOUGH_CORE`	메모리 부족	Heap 크기 확장
`FR_TIMEOUT`	응답 지연	SPI 속도 낮추기

---

11. 성능 팁

- SPI 전송에 **DMA 사용 (`HAL_SPI_Transmit_DMA`)**
- `FA_OPEN_APPEND` 사용으로 FAT 탐색 최소화
- `f_sync()` 호출은 주기적으로만 수행 (예: 1분 단위)
- SPI 클럭 12~18 MHz 범위 권장
- SDHC 카드 사용 시 주소 변환(섹터 단위) 주의

---

12. 요약

항목	내용
사용 라이브러리	FATFS (ChanN)
인터페이스	SPI 또는 SDIO
단위	512 Byte 섹터
주요 함수	f_mount, f_open, f_write, f_close
응용	데이터 로깅, 설정 파일, 펌웨어 백업
권장 구조	“센서 → 큐 → LogTask → FATFS 쓰기”

FATFS를 이용하면 **데이터 저장 로직을 표준화**하고,
PC에서 직접 파일을 확인할 수 있는 로깅 시스템을 구현할 수 있다.
센서값, 이벤트 로그, 보정값 저장 등 임베디드 프로젝트 전반에서 필수적인 기능이다.

- # CSV 로깅

1. 개요

CSV(Comma-Separated Values) 형식은 **텍스트 기반 표준 데이터 포맷**으로,
엑셀(Excel), 구글 시트(Google Sheets), 파이썬(pandas) 등에서
손쉽게 읽고 분석할 수 있다.

FATFS를 이용해 SD 카드에 **센서 데이터를 CSV로 저장**하면,
PC에서 바로 그래프 분석 및 통계 처리가 가능하다.

---

2. 파일 구조 예시

Timestamp	Level(cm)	Temp(°C)	Weight(g)	Flow(L/min)
2025-11-09 10:00:00	123.4	24.6	1250.3	3.80
2025-11-09 10:01:00	123.6	24.5	1250.8	3.85
2025-11-09 10:02:00	123.8	24.7	1251.1	3.89

> 첫 줄(Header)은 컬럼 이름으로, 이후 각 측정 데이터가 한 줄씩 추가된다.

---

3. 파일 오픈 및 헤더 생성

```
#include "fatfs.h"
#include "stdio.h"

FATFS fs;
FIL file;
```

```
FRESULT res;
UINT bw;

void CSV_Init(void)
{
    res = f_mount(&fs, SDPath, 1);
    if (res != FR_OK) return;

    res = f_open(&file, "data.csv", FA_OPEN_ALWAYS | FA_WRITE);
    if (res == FR_OK) {
        // 파일 크기가 0이면 헤더 추가
        if (f_size(&file) == 0) {
            f_printf(&file, "Timestamp,Level_cm,Temp_C,Weight_g,Flow_Lpm\r\n");
        }
        f_close(&file);
    }

    f_mount(NULL, SDPath, 1);
}
```

4. 데이터 기록 루틴

```
typedef struct {
    float level;
    float temp;
    float weight;
    float flow;
    RTC_TimeTypeDef time;
    RTC_DateTypeDef date;
} SensorData_t;

void CSV_LogData(SensorData_t *data)
{
    char line[128];

    sprintf(line, "%04d-%02d-%02d %02d:%02d:%02d,%.1f,%.1f,%.1f,%.2f\r\n",
            2000 + data->date.Year,
            data->date.Month,
            data->date.Date,
            data->time.Hours,
            data->time.Minutes,
            data->time.Seconds,
            data->level,
            data->temp,
            data->weight,
            data->flow);

    if (f_mount(&fs, SDPath, 1) == FR_OK) {
        if (f_open(&file, "data.csv", FA_OPEN_APPEND | FA_WRITE) == FR_OK) {
            f_write(&file, line, strlen(line), &bw);
            f_close(&file);
```

```
        }
        f_mount(NULL, SDPath, 1);
    }
}
```

> 💡 `FA_OPEN_APPEND` 옵션은 파일 끝으로 자동 이동하여 새 데이터를 추가한다.

---

5. FreeRTOS 기반 주기적 로깅 Task

```
void CSVTask(void *argument)
{
    CSV_Init();

    for (;;) {
        SensorData_t data = GetSensorData();
        CSV_LogData(&data);
        osDelay(60000);  // 1분 간격
    }
}
```

---

6. CSV 파일 읽기 예 (PC 측 파이썬 코드)

```
import pandas as pd

df = pd.read_csv("data.csv")
print(df.head())

# 평균 수위, 온도 계산
print(df["Level_cm"].mean())
print(df["Temp_C"].mean())
```

---

7. CSV 포맷 관리

구분	내용
구분자	`,` (쉼표)
소수점	`.` (영문형)
줄바꿈	`\r\n` (Windows 호환)
인코딩	ASCII 또는 UTF-8
헤더	첫 줄에 필드명 명시

---

8. 메모리 / 속도 최적화

- `sprintf()` 대신 `snprintf()` 사용으로 버퍼 오버플로 방지
- 일정 횟수마다만 `f_mount()` 호출 (매주기 X)
- `f_sync()` 는 10~20회 기록 후 호출하여 SD 수명 보호
- 로그 주기를 FreeRTOS 타이머 기반으로 일정하게 유지

---

9. 예제 결과 (data.csv)

```
Timestamp,Level_cm,Temp_C,Weight_g,Flow_Lpm
2025-11-09 10:00:00,123.4,24.6,1250.3,3.80
2025-11-09 10:01:00,123.6,24.5,1250.8,3.85
2025-11-09 10:02:00,123.8,24.7,1251.1,3.89
```

---

10. 요약

항목	내용
파일 형식	CSV (Comma Separated Values)
장점	PC/Excel 호환, 단순 파싱, 텍스트 기반
주 용도	장기 데이터 로깅, 분석용
구현 방식	FATFS + `f_printf()` / `f_write()`
주의사항	Flush(`f_close` , `f_sync`) 필수, 주기적 Mount 관리

CSV 로깅은 임베디드 환경에서
데이터를 **가볍고 범용적으로 저장**할 수 있는 가장 효율적인 방식이다.
특히 STM32 + FATFS 시스템에서 **센서 로그, 진단, 통계** 등에 최적화되어 있다.

13.4 IoT 연동

- # ESP32 MQTT 브릿지

1. 개요

MQTT (Message Queuing Telemetry Transport) 는
경량 메시징 프로토콜로, 저전력 IoT 장치 간의 실시간 데이터 교환에 널리 사용된다.
ESP32는 Wi-Fi와 BLE를 모두 지원하므로,
STM32 등 유선 센서 보드와 연동 시 **MQTT 브릿지 역할**로 적합하다.

ESP32는 다음과 같은 역할을 수행할 수 있다:

- STM32로부터 UART/I²C로 데이터를 수신
- MQTT Broker로 데이터 중계 (Publish)

- 서버 명령을 MQTT 구독 (Subscribe) 후 STM32로 전달

---

2. 시스템 구성 예시

```
┌──────────────────┐       UART/I²C      ┌──────────────────┐
|   STM32 Board  |  <--------------->  |     ESP32        |
|  Sensor Module |                     | MQTT Bridge      |
|  (Temp, Level) |                     | Wi-Fi + Broker   |
└──────────────────┘                     └──────────┬──────┘
                                                    |
                                          MQTT over TCP/IP
                                                    |
                                    ┌──────────────────────────────┐
                                    | MQTT Broker                  |
                                    | (e.g. Mosquitto, AWS IoT)    |
                                    └──────────────────────────────┘
```

---

3. MQTT 기본 구조

역할	설명
Broker	메시지 중계 서버 (Mosquitto, EMQX, AWS IoT 등)
Publisher	데이터를 전송하는 클라이언트
Subscriber	특정 토픽을 구독하고 메시지를 수신
Topic	메시지 분류 경로 (예: `/sensor/level`)

ESP32는 동시에 Publisher & Subscriber 역할을 수행할 수 있다.

---

4. Arduino 환경 설정

필요 라이브러리

- `WiFi.h`
- `PubSubClient.h` (Nick O'Leary 제작, MQTT 표준 라이브러리)

```
#include <WiFi.h>
#include <PubSubClient.h>
```

---

5. 기본 연결 코드

```
const char* ssid = "YOUR_SSID";
const char* password = "YOUR_PASS";
const char* mqtt_server = "test.mosquitto.org";

WiFiClient espClient;
PubSubClient client(espClient);

void setup_wifi() {
  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) delay(500);
  Serial.println("WiFi connected");
}

void reconnect() {
  while (!client.connected()) {
    if (client.connect("ESP32Bridge")) {
      client.subscribe("/stm32/cmd");
    } else {
      delay(2000);
    }
  }
}

void setup() {
  Serial.begin(115200);
  setup_wifi();
  client.setServer(mqtt_server, 1883);
  client.setCallback(callback);
}

void callback(char* topic, byte* message, unsigned int length) {
  Serial.print("Message arrived [");
  Serial.print(topic);
  Serial.print("]: ");
  for (int i = 0; i < length; i++) Serial.print((char)message[i]);
  Serial.println();
  // STM32로 명령 전달
}
```

6. STM32 데이터 중계 (UART → MQTT Publish)

```
void loop() {
  if (!client.connected()) reconnect();
  client.loop();

  if (Serial.available()) {
    String data = Serial.readStringUntil('\n');
    client.publish("/stm32/data", data.c_str());
    Serial.print("Published: "); Serial.println(data);
  }
}
```

> STM32는 UART로 센서 데이터(예: `"LEVEL=123.4,TEMP=24.5"`)를 전송한다.

---

7. 서버 → STM32 명령 수신

```
void callback(char* topic, byte* payload, unsigned int length) {
  String msg;
  for (int i = 0; i < length; i++) msg += (char)payload[i];
  Serial.print("MQTT CMD: "); Serial.println(msg);
  Serial1.println(msg);  // STM32로 전달
}
```

> 예: `/stm32/cmd` 토픽에 `"PUMP_ON"` 메시지를 보내면 ESP32가 UART로 STM32에 전달.

---

8. 양방향 데이터 구조 예시

방향	토픽	데이터 예시
STM32 → ESP32 → Broker	`/stm32/data`	`{"level":120.5,"temp":24.8,"weight":1250}`
Broker → ESP32 → STM32	`/stm32/cmd`	`{"pump":1}`

ESP32가 실시간 중계 노드로 동작하면서
유선 STM32와 클라우드 간의 완전한 데이터 루프를 형성한다.

---

9. JSON 기반 MQTT 전송

```
#include <ArduinoJson.h>

void sendSensorData(float level, float temp, float weight) {
  StaticJsonDocument<128> doc;
  doc["level"] = level;
  doc["temp"] = temp;
  doc["weight"] = weight;
  char payload[128];
  serializeJson(doc, payload);
  client.publish("/stm32/data", payload);
}
```

> JSON 형식으로 MQTT 메시지를 구성하면
> 서버나 Node-RED, Python 등에서 손쉽게 파싱 가능.

---

10. 오류 및 복구 처리

항목	대응
Wi-Fi 끊김	`WiFi.status()` 체크, 자동 재연결
MQTT 세션 종료	`!client.connected()` 감지 후 `reconnect()` 호출
UART 프레임 손실	CRC 또는 종료문자(`\n`)로 프레이밍
브로커 장애	`client.state()` 확인 후 대체 서버 시도

---

11. 확장 예시

기능	구현 방향
OTA 펌웨어 업데이트	MQTT `/update` 명령 기반
로컬 캐시	Wi-Fi 끊김 시 데이터 버퍼링 후 재전송
BLE ↔ MQTT 브릿지	BLE 센서 데이터 → MQTT Broker 업링크
TLS 보안	`WiFiClientSecure` + `client.setServer(..., 8883)`

---

12. 요약

항목	내용
프로토콜	MQTT (v3.1.1)
라이브러리	PubSubClient

항목	내용
역할	STM32 ↔ ESP32 ↔ Broker 중계
연결 방식	UART or I²C 기반 로컬 링크
주요 함수	`client.connect()`, `client.publish()`, `client.subscribe()`
장점	경량, 저지연, 다중 디바이스 확장 용이

ESP32 MQTT 브릿지는 **STM32의 로컬 측정 데이터**를
클라우드 서비스, Node-RED, Home Assistant 등으로 전송하는
가장 효율적이고 범용적인 IoT 아키텍처이다.

- # RS485 Modbus RTU 연결

1. 개요

RS485는 산업 현장에서 널리 사용되는 **차동(差動) 통신 표준**으로,
노이즈 내성이 뛰어나고 최대 수백 미터 이상의 거리에서도 안정적인 통신이 가능하다.
한 개의 마스터와 여러 개의 슬레이브를 **멀티드롭(Multi-drop)** 방식으로 연결할 수 있다.

STM32는 UART 하드웨어를 이용해 RS485 통신을 구현하며,
프로토콜 계층에는 **Modbus RTU (Remote Terminal Unit)** 가 사용된다.
Modbus RTU는 간결한 구조와 CRC 기반 오류 검출 덕분에
PLC, 센서, HMI 등 다양한 산업 장비와의 호환성이 높다.

---

2. RS485 하드웨어 구성

항목	설명
물리 계층	RS485 차동 라인 (A, B)
트랜시버 IC	MAX485, SN75176, SP3485 등
제어 신호	DE(Data Enable), RE(Receive Enable)
전송 방식	반이중(Half Duplex) – 송신과 수신을 시간분할 수행

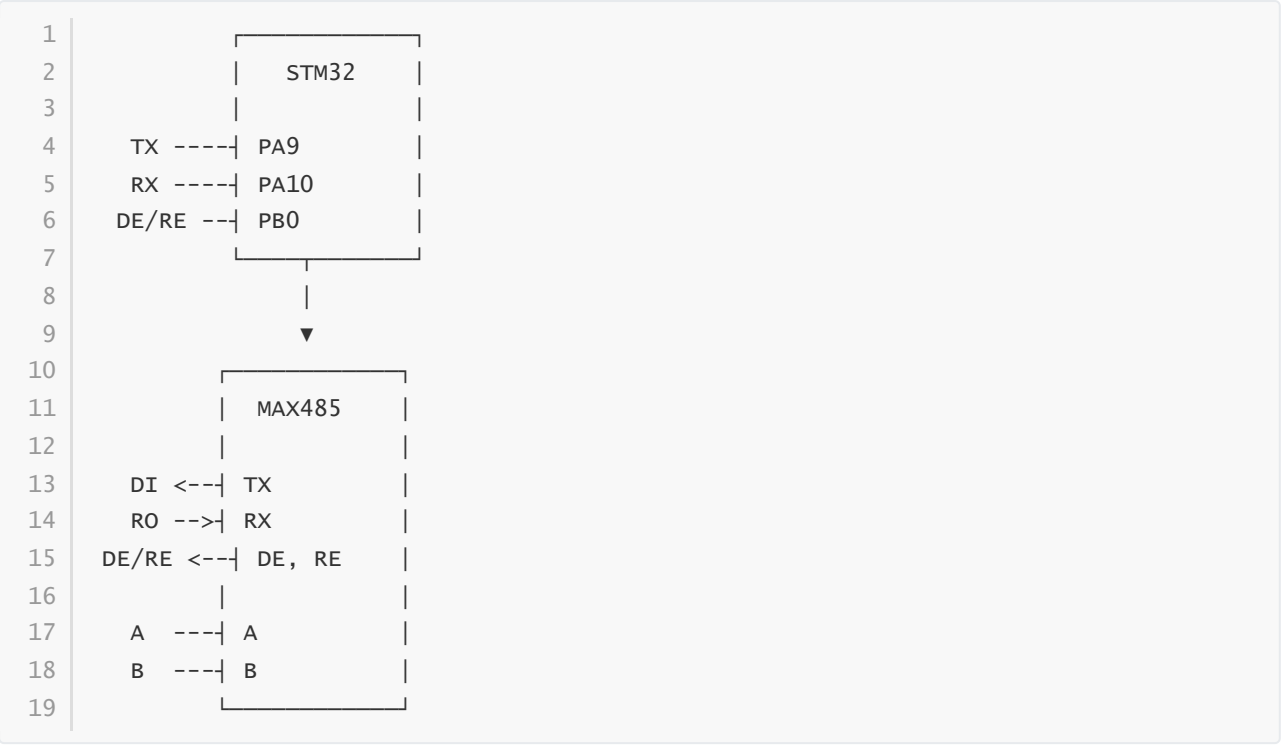
RS485 트랜시버는 STM32의 UART TX/RX와 다음과 같이 연결된다:

```
STM32 TX → DI (Driver Input)
STM32 RX ← RO (Receiver Output)
STM32 GPIO → DE & RE (송수신 전환)
```

> DE/RE 핀은 동일한 GPIO로 묶어 제어하는 것이 일반적이다.
> 송신 시 HIGH, 수신 시 LOW로 설정한다.

---

3. 회로 예시



---

4. Modbus RTU 프레임 구조

필드	크기 (byte)	설명
Slave Address	1	대상 디바이스 주소 (1~247)
Function Code	1	명령 종류 (예: 0x03 – Read Holding Registers)
Data	N	요청 또는 응답 데이터
CRC16	2	하위바이트 + 상위바이트 순서

예시:

```
[01][03][00][10][00][02][C5][CD]
→ Slave 1, Holding Register 0x0010부터 2개 읽기
```

---

5. UART 초기화

CubeMX 또는 HAL 코드를 이용하여 다음과 같이 UART를 설정한다:

- Baud rate: 9600 ~ 115200 (기기 사양에 맞게)
- Data bits: 8
- Parity: None / Even (기기 요구에 따라)
- Stop bits: 1
- Mode: TX/RX

```
huart1.Instance = USART1;
huart1.Init.BaudRate = 9600;
huart1.Init.WordLength = UART_WORDLENGTH_8B;
huart1.Init.StopBits = UART_STOPBITS_1;
huart1.Init.Parity = UART_PARITY_NONE;
huart1.Init.Mode = UART_MODE_TX_RX;
HAL_UART_Init(&huart1);
```

---

6. 송수신 제어 로직

RS485는 반이중 통신이므로, 송신 전후로 DE/RE 제어가 필요하다.

```
#define RS485_DE_RE_PIN GPIO_PIN_0
#define RS485_DE_RE_PORT GPIOB

void RS485_SetTransmit(void) {
    HAL_GPIO_WritePin(RS485_DE_RE_PORT, RS485_DE_RE_PIN, GPIO_PIN_SET);
}

void RS485_SetReceive(void) {
    HAL_GPIO_WritePin(RS485_DE_RE_PORT, RS485_DE_RE_PIN, GPIO_PIN_RESET);
}
```

송신 시퀀스:

```
RS485_SetTransmit();
HAL_UART_Transmit(&huart1, txBuffer, length, 100);
RS485_SetReceive();
```

---

7. CRC16 계산

Modbus RTU는 CRC-16 (Modbus variant) 알고리즘을 사용한다.

```
uint16_t Modbus_CRC16(uint8_t *buf, uint16_t len) {
    uint16_t crc = 0xFFFF;
    for (uint16_t pos = 0; pos < len; pos++) {
        crc ^= buf[pos];
        for (int i = 0; i < 8; i++) {
            if (crc & 1)
                crc = (crc >> 1) ^ 0xA001;
            else
                crc >>= 1;
        }
    }
    return crc;
}
```

---

8. 예제 – Holding Register 읽기 요청

```
uint8_t modbus_tx[8];
uint8_t modbus_rx[32];

void Modbus_ReadRegister(uint8_t id, uint16_t addr, uint16_t len) {
    modbus_tx[0] = id;
    modbus_tx[1] = 0x03;          // Function Code: Read Holding Register
    modbus_tx[2] = addr >> 8;
    modbus_tx[3] = addr & 0xFF;
    modbus_tx[4] = len >> 8;
    modbus_tx[5] = len & 0xFF;

    uint16_t crc = Modbus_CRC16(modbus_tx, 6);
    modbus_tx[6] = crc & 0xFF;
    modbus_tx[7] = crc >> 8;

    RS485_SetTransmit();
    HAL_UART_Transmit(&huart1, modbus_tx, 8, 100);
    RS485_SetReceive();
    HAL_UART_Receive(&huart1, modbus_rx, 7 + len * 2, 100);
}
```

응답 데이터는 다음과 같이 구성된다:

```
[Slave ID][Function][Byte Count][Data…][CRC_L][CRC_H]
```

---

9. Multi-Slave 테스트

RS485 라인에 여러 슬레이브를 병렬 연결할 수 있다.
각 슬레이브는 고유한 **Slave Address**를 갖는다.

```
STM32 Master → RS485 Bus → [Sensor #1: Addr=0x01]
                           [Sensor #2: Addr=0x02]
                           [Sensor #3: Addr=0x03]
```

> 마스터는 슬레이브별로 순차적으로 Modbus 요청을 보내며,
> 한 번에 하나의 장치만 응답하도록 프로토콜이 보장한다.

---

10. 통신 예시

동작	송신 프레임 (HEX)	응답 예시
레지스터 읽기	`01 03 00 00 00 02 C4 0B`	`01 03 04 00 1E 00 32 B8 44`
코일 제어	`01 05 00 10 FF 00 8D FA`	`01 05 00 10 FF 00 8D FA`

---

11. 디버깅 및 주의사항

항목	설명
노이즈 대응	120 Ω 종단저항(Termination) 필수
라인 반전	A/B 라인 교차 시 통신 불가
타임아웃	`HAL_UART_Receive()` Timeout 값 조정
송수신 타이밍	DE 제어 지연 최소화 (송신 직후 즉시 수신 모드 복귀)
전원 노이즈	별도 GND 연결 및 차폐 권장

---

12. FreeRTOS 환경 통합

RS485 통신은 Task 내에서 주기적으로 수행할 수 있다.

```
void ModbusTask(void *argument) {
    for (;;) {
        Modbus_ReadRegister(1, 0x0000, 2);
        vTaskDelay(pdMS_TO_TICKS(500));
    }
}
```

> 송수신 중 Mutex로 UART 자원을 보호해야 한다.

---

13. 요약

항목	내용
물리 계층	RS485 (차동, 반이중)
프로토콜	Modbus RTU
전송 속도	9600~115200bps
오류 검출	CRC-16 (Modbus)
다중 노드	최대 32대 (Repeater 사용 시 확장 가능)
주요 장점	산업용 안정성, 장거리 통신, 간단한 구현

RS485 + Modbus RTU 구조는 **STM32 기반 산업용 제어 시스템**에서
센서 네트워크나 액추에이터 제어를 구현하기 위한
가장 표준적이고 신뢰성 높은 통신 방식이다.

- # 클라우드 모니터링 (Grafana)

1. 개요

Grafana는 센서, 게이트웨이, 서버 등에서 수집된 데이터를
시각적으로 대시보드 형태로 모니터링할 수 있는 오픈소스 플랫폼이다.

STM32나 ESP32 기반 시스템에서 수집한 데이터를
MQTT → Cloud Broker → InfluxDB → Grafana로 전달하면
실시간으로 **수위, 온도, 전류, 밸브 상태** 등을
웹에서 그래프로 확인할 수 있다.

---

2. 시스템 구성 흐름

```
[STM32/ESP32]
   ↓ (MQTT Publish)
[MQTT Broker - Cloud/Mosquitto]
   ↓
[InfluxDB - 시계열 DB]
   ↓
[Grafana Dashboard]
```

요약

- STM32: 센서 데이터 측정 → BLE/Wi-Fi로 ESP32에 전송
- ESP32: MQTT Publish (ex: topic = `/tank/level`)
- Cloud Server: MQTT → InfluxDB 저장 → Grafana 시각화

---

3. MQTT → InfluxDB 브릿지

(1) MQTT 브로커

- 대표 예시: **Eclipse Mosquitto**, **HiveMQ**, **EMQX**, **AWS IoT Core**
- 토픽 예시

```
/tank/level
/tank/pump_status
/tank/temperature
```

(2) InfluxDB

- 시계열(Time-Series) 데이터베이스
- 구조:

```
measurement: tank_status
fields: {level=35.2, pump=1, temp=23.4}
tags: {device="tank01"}
time: 자동
```

(3) Node-RED / Telegraf 연동

- MQTT 데이터를 InfluxDB에 자동 삽입하는 파이프라인 구성

```
[MQTT Input] → [JSON Parse] → [InfluxDB Out]
```

- 또는 **Telegraf** 플러그인 사용

```
[[inputs.mqtt_consumer]]
  servers = ["tcp://broker.emqx.io:1883"]
  topics = ["/tank/#"]
  data_format = "json"

[[outputs.influxdb_v2]]
  urls = ["http://localhost:8086"]
  token = "YOUR_TOKEN"
  organization = "iot"
  bucket = "tank"
```

---

4. Grafana 설정

(1) 데이터 소스 추가

- **Configuration → Data Sources → Add InfluxDB**
- URL: `http://localhost:8086`
- Token, Bucket, Org 입력

(2) 대시보드 생성

- 패널(Panels) 추가
- 쿼리 예시 (Flux):

```
from(bucket: "tank")
  |> range(start: -1h)
  |> filter(fn: (r) => r._measurement == "tank_status")
  |> filter(fn: (r) => r._field == "level")
```

(3) 시각화 예시

항목	시각화 형태
수위(Level)	Line Chart

항목	시각화 형태
펌프 상태	Switch / Gauge
온도	Time Series
전류	Bar Graph
경보	Threshold Alert (e.g., level < 10%)

---

5. 알림(Alerts) 설정

Grafana는 특정 조건에 따라 **Slack / Telegram / Email** 등으로 알림을 보낼 수 있다.

- 예: “수위 10% 이하” 시 Telegram 경보

```
Condition: WHEN avg() OF query(A, 5m, now) IS BELOW 10
Notification: Telegram Bot
```

---

6. 클라우드 배포 예시

플랫폼	용도
AWS EC2	Grafana + InfluxDB 서버
AWS IoT Core / Azure IoT Hub	MQTT Broker
Grafana Cloud (무료)	외부 호스팅형 대시보드
Tailscale / Ngrok	로컬 Grafana를 외부에서 접근 가능하게 함

---

7. 예제 대시보드 구성

패널	내용
Tank Level (%)	실시간 수위 그래프 (Line Chart)
Pump State	ON/OFF 표시 (Stat Panel)
Temperature (°C)	시계열 그래프
Battery Voltage (V)	게이지
Alarm Log	테이블 형태로 이벤트 기록

---

8. FreeRTOS 기반 MQTT 데이터 전송 코드 예시 (ESP32)

```
#include <WiFi.h>
#include <PubSubClient.h>

WiFiClient espClient;
PubSubClient client(espClient);

void mqtt_send(float level, int pump, float temp) {
  char payload[128];
  sprintf(payload, "{\"level\":%.2f,\"pump\":%d,\"temp\":%.2f}", level, pump, temp);
  client.publish("/tank/status", payload);
}
```

STM32 → ESP32 (UART 또는 I²C)
ESP32 → MQTT Broker → Grafana 표시

---

9. 유지보수 및 확장

기능	설명
데이터 보존 정책	InfluxDB Retention 설정 (예: 30일)
백업	InfluxDB `influx backup`
확장성	여러 탱크/라인 추가 시 `device_id` Tag로 구분
보안	MQTT over TLS, Grafana Auth 활성화

---

10. 요약

구성요소	역할
STM32/ESP32	센서 데이터 수집 및 송신
MQTT Broker	데이터 중계
InfluxDB	시계열 데이터 저장
Grafana	실시간 대시보드 시각화
Node-RED/Telegraf	데이터 파이프라인 구성
클라우드 서비스	원격 접속 및 알림 관리

> **결과:**
> 실시간으로 수위·온도·펌프상태를 웹에서 시각화하고,
> 이상 상태 발생 시 즉시 알람을 받아볼 수 있는
> 완전한 **IoT 수위 모니터링 시스템**을 구축할 수 있다.