BMP180 & LCD1602 Project

With STM32

(Telechips) AI 시스템반도체 SW 개발자

박성호 박준영

TABLE OF CONTENTS

01 프로젝트 개요

02 Schematic

03 세부 구현 기능

04 코드 설명

05 결과 및 느낀점

01 프로젝트 개요

프로젝트 목적

- BMP180과 LCD1602의 데이터시트를 보고 Device Driver 구현.
- 데이터 시트에 명시되어 있는 내용을 그대로 코드를 작성하고 구현하는 능력 향상.

프로젝트 내용

- BMP180 Driver
  - I2C 프로토콜 방식으로 동작센서 (기압, 온도 측정)
  - 무응답 시 timeout 처리 기능 추가.
- LCD1602 Driver
  - 4bit 방식으로 구현.
- DHT11 / DS1302 / DOTMATRIX와 Shift-Register 연동.
  - 데이터 시트를 통해 코드 작성 후 동작 구현.

01 프로젝트 개요

주요 부품

STM32 Nucleo-F411RE

BMP180

LCD1602

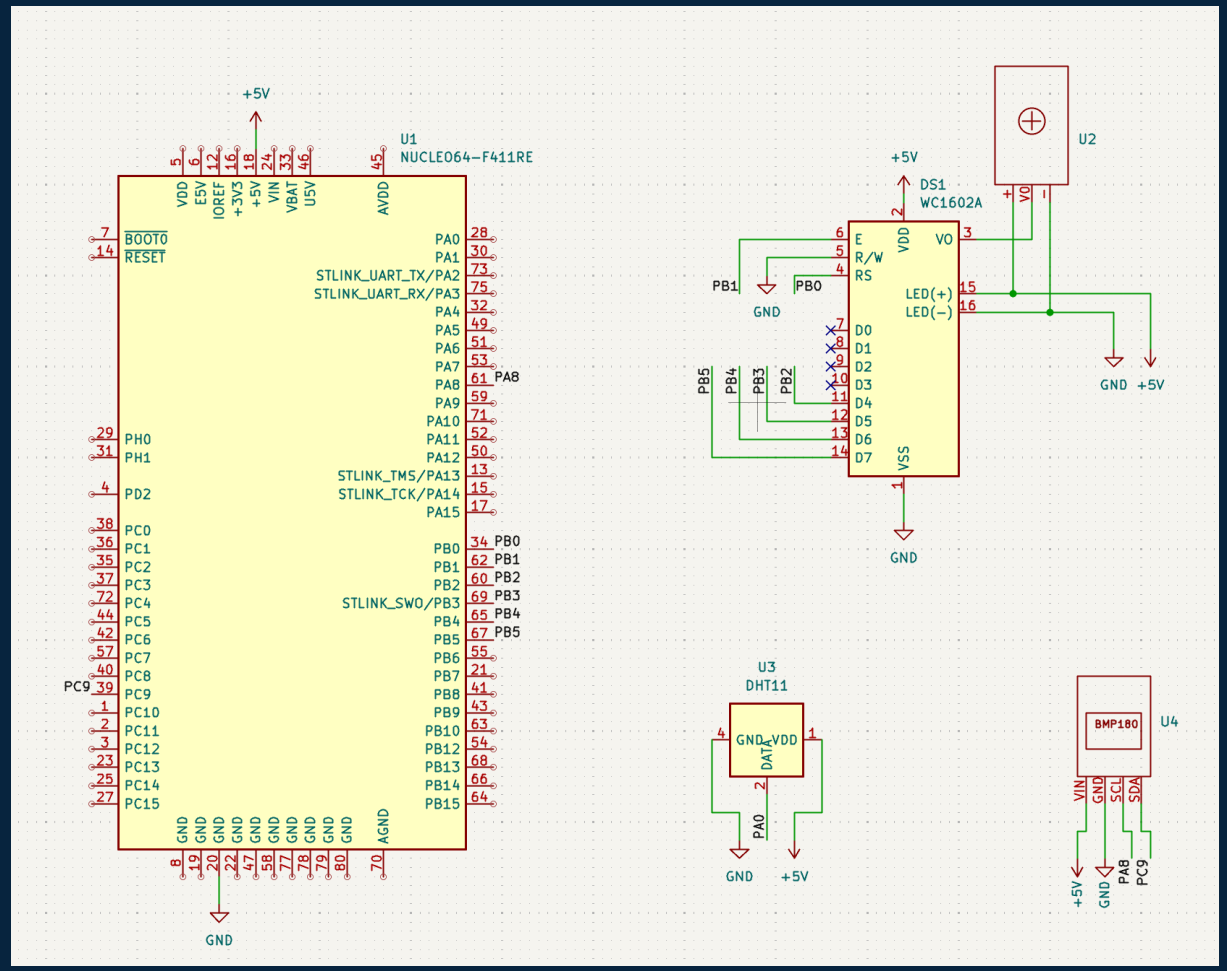
DOTMATRIX

DS1302

DHT11

02 Schematic

Schematic (회로도)



03 세부 구현 기능

기능 설명

- BMP180 - 온도, 기압 측정
  - I2C 통신으로 센서의 보정 데이터를 읽고 원시 온도 및 기압 값을 획득한 후 이를 보정하여 LCD와 UART로 출력.
  - 무응답 시 TIMEOUT.

- LCD1602 - 측정값 출력
  - 4비트 단위로 데이터를 LCD에 전송.
  - LCD는 4비트 모드에서 상위 4비트와 하위 4비트를 별도로 전송하여 한 바이트 구성.

- DHT11 - 온도, 습도 측정
  - 하나의 데이터 라인을 이용하여 매우 정밀한 타이밍으로 신호의 HIGH / LOW를 교환하는 방식.

- DS1302 - 날짜, 시간 출력
  - RTC 모듈과 직접 통신을 통해 실시간 시계 데이터를 읽고 설정하며 LCD와 UART를 통해 현재 시간 정보 제공.

04 코드 설명 (bmp180.c)

Key features

Pressure range:	300 ... 1100hPa (+9000m ... -500m relating to sea level)
Supply voltage:	1.8 ... 3.6V (V_{DD}) 1.62V ... 3.6V (V_{DDIO})
Package:	LGA package with metal lid Small footprint: 3.6mm x 3.8mm Super-flat: 0.93mm height
Low power:	5µA at 1 sample / sec. in standard mode
Low noise:	0.06hPa (0.5m) in ultra low power mode 0.02hPa (0.17m) advanced resolution mode

- Temperature measurement included
- I^2C interface
- Fully calibrated
- Pb-free, halogen-free and RoHS compliant,
- MSL 1

센서 주요 특징

- 압력 범위 : 300 ~ 1100 hPa
- 온도 측정 포함
- I2C 인터페이스 지원
- 내장 보정 계수 (11개의 16비트 값, EEPROM에 저장)

04 코드 설명 (bmp180.c)

```c
// BMP180 I2C 주소 (8비트 주소 체계: 좌측 쉬프트)
// BMP180 센서의 기본 I2C 주소인 0x77을 왼쪽으로 1비트 시프트하여 8비트 형식으로 변환
// HAL 라이브러리에서는 8비트 주소를 사용하므로
#define BMP180_ADDRESS (0x77 << 1)

// hi2c3는 BMP180 센서와 통신하기 위해 사용되는 I2C 인터페이스의 핸들러
extern I2C_HandleTypeDef hi2c3;

// BMP180 레지스터 및 명령어
// 보정 계수(캘리브레이션 데이터)를 저장한 EEPROM의 시작 주소 0xAA
#define BMP180_REG_CALIB_START 0xAA
// 센서 제어 레지스터 주소(명령어 쓰기를 통해 온도/압력 측정 시작)
#define BMP180_REG_CONTROL     0xF4
// 측정 결과(데이터)의 최상위 바이트를 읽는 주소
#define BMP180_REG_OUT_MSB     0xF6
// 온도 측정을 위한  명령어(0x2E를 제어 레지스터에 쓰면 온도 측정 시작)
#define BMP180_CMD_READ_TEMP   0x2E
// 기압 측정을 위한 기본 명령어(oversampling 설정에 따라 변경)
#define BMP180_CMD_READ_PRESS  0x34

// 보정 계수 변수
// BMP180 센서는 개별 칩마다 고유의 보정 계수를 EEPROM에 저장
// 온도와 기압을 정확하게 계산하기 위해 이 계수들을 읽어와서 사용
// AC1, AC2, AC3, B1, B2, MB, MC, MD는 부호가 있는 16비트 정수형
int16_t AC1, AC2, AC3, B1, B2, MB, MC, MD;
// AC4, AC5, AC6는 부호가 없는 16비트 정수형
uint16_t AC4, AC5, AC6;
```

The BMP180 module address is shown below. The LSB of the device address distinguishes between read (1) and write (0) operation, corresponding to address 0xEF (read) and 0xEE (write).

Table 7: BMP180 addresses

A7	A6	A5	A4	A3	A2	A1	W/R
1	1	1	0	1	1	1	0/1

- BMP180 I2C 주소
  - 기본 주소 0x77을 8비트 형식으로 변환
- I2C 인터페이스 핸들러 (I2C3).
- 보정 계수를 저장한 EEPROM의 시작 주소 설정.
- 센서 제어 레지스터 주소 설정.
- 데이터의 최상위 바이트를 읽는 주소 설정.
- 온도 측정을 위한 명령어 설정(0x2E를 제어 레지스터에 쓰면 온도 측정 시작)
- 기압 측정을 위한 기본 명령어 설정.
- 보정 계수 변수 설정 (16비트 정수형)

04 코드 설명 (bmp180.c)

```c
// 보정 계수 읽기 함수
// I2C 인터페이스를 통해 BMP180의 보정 데이터를 읽어오는 함수
void BMP180_ReadCalibrationCoefficients(I2C_HandleTypeDef *hi2c)
{
    // calib_data 배열(22바이트 크기)은 보정 데이터 전체 저장
    uint8_t calib_data[22];
    if (HAL_I2C_Mem_Read(hi2c, BMP180_ADDRESS, BMP180_REG_CALIB_START,
            I2C_MEMADD_SIZE_8BIT, calib_data, 22, BMP180_TIMEOUT_MS) != HAL_OK)
    {
        // 타임아웃 또는 통신 오류 시 처리: 여기서는 간단히 리턴
        return;
    }
    // 보정 데이터는 8비트 단위로 읽어오므로 상우 | 바이트와 하위 바이트를 결합하여 16비트값으로 만듦
    // (calib_data[0] << 8 | calib_data[1])는 첫 번째 16비트 값(AC1)을 만듦
    // 나머지 보정 계수들도 같은 방식으로 계산
    // 부호 있는 값은 (int16_t)로, 부호 없는 값은 (uint16_t)로 캐스팅
    AC1 = (int16_t)(calib_data[0] << 8 | calib_data[1]);
    AC2 = (int16_t)(calib_data[2] << 8 | calib_data[3]);
    AC3 = (int16_t)(calib_data[4] << 8 | calib_data[5]);
    AC4 = (uint16_t)(calib_data[6] << 8 | calib_data[7]);
    AC5 = (uint16_t)(calib_data[8] << 8 | calib_data[9]);
    AC6 = (uint16_t)(calib_data[10] << 8 | calib_data[11]);
    B1  = (int16_t)(calib_data[12] << 8 | calib_data[13]);
    B2  = (int16_t)(calib_data[14] << 8 | calib_data[15]);
    MB  = (int16_t)(calib_data[16] << 8 | calib_data[17]);
    MC  = (int16_t)(calib_data[18] << 8 | calib_data[19]);
    MD  = (int16_t)(calib_data[20] << 8 | calib_data[21]);
}
```

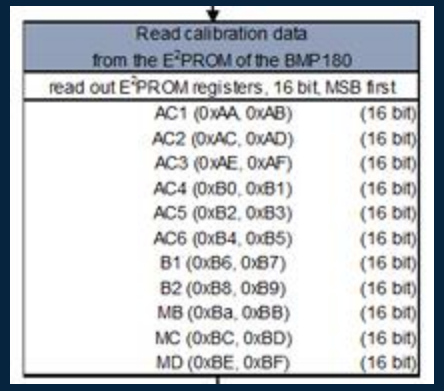


Table 5: Calibration coefficients

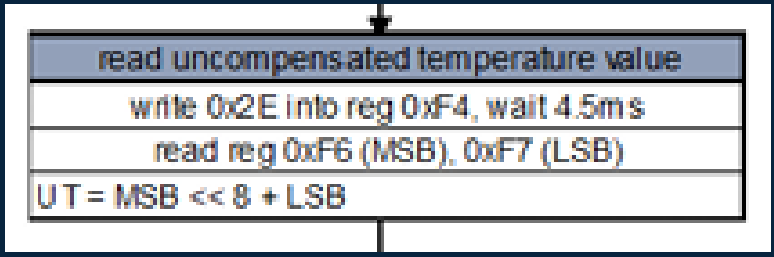
	BMP180 reg adr	
Parameter	MSB	LSB
AC1	0xAA	0xAB
AC2	0xAC	0xAD
AC3	0xAE	0xAF
AC4	0xB0	0xB1
AC5	0xB2	0xB3
AC6	0xB4	0xB5
B1	0xB6	0xB7
B2	0xB8	0xB9
MB	0xBA	0xBB
MC	0xBC	0xBD
MD	0xBE	0xBF

보정 계수 읽기 함수

- HAL_I2C_Mem_Read() 함수 사용.
  - BMP180의 보정 데이터가 저장된 시작 주소부터 22바이트를 읽어옴.
  - I2C 핸들러, I2C 주소, EEPROM의 시작 주소, 메모리 주소 크기, 데이터 저장 배열, 타임아웃 시간.
- 보정 계수 결합 및 저장
  - 보정 데이터를 8비트 단위로 저장.
  - 상위 바이트(MSB)와 하위 바이트(LSB)로 나뉨.
  - 두 바이트를 결합하여 하나의 16비트 값 생성.
- BMP180 데이터시트의 “Calibration coefficients”

04 코드 설명 (bmp180.c)

```c
// 원시 온도 값 읽기
// 온도 측정을 위해 센서에 명령을 보내고 원시 온도 데이터를 읽어오는 함수
int16_t BMP180_ReadRawTemperature(I2C_HandleTypeDef *hi2c)
{
    // cmd 변수에 온도 측정 명령어(0x2E)를 저장
    uint8_t cmd = BMP180_CMD_READ_TEMP;
    if (HAL_I2C_Mem_Write(hi2c, BMP180_ADDRESS, BMP180_REG_CONTROL,
            I2C_MEMADD_SIZE_8BIT, &cmd, 1, BMP180_TIMEOUT_MS) != HAL_OK)
    {
        // 통신 타임아웃 또는 오류 발생 시 -1 반환
        return -1;
    }
    // 센서가 온도 측정을 완료할 때까지 대기 시간
    HAL_Delay(5);
    // 0xF6 주소에서 2바이트를 읽어와 원시 온도 데이터(raw 배열) 저장
    uint8_t raw[2];
    if (HAL_I2C_Mem_Read(hi2c, BMP180_ADDRESS, BMP180_REG_OUT_MSB,
            I2C_MEMADD_SIZE_8BIT, raw, 2, BMP180_TIMEOUT_MS) != HAL_OK)
    {
        return -1;
    }
    // 상위 바이트와 하위 바이트를 결합하여 16비트 정수형 온도 값을 반환
    return (int16_t)((raw[0] << 8) | raw[1]);
}
```



원시 온도 값 읽기

- 온도 측정 시작하려면 제어 레지스터(0xF4)에 0x2E 값 씀.
- HAL_I2C_Mem_Write() 함수 사용
  - I2C 인터페이스 핸들러
  - I2C 주소
  - 제어 레지스터 주소에 명령어 씀.
  - 메모리 주소
  - 전송할 데이터(0x2E 명령어)
  - 1바이트 전송
  - 타임아웃 시간
- HAL_I2C_Mem_Read() 함수 사용
  - 온도 측정 후 결과를 0xF6 주소에서 2바이트로 제공.

04 코드 설명 (bmp180.c)

```c
// 원시 기압 값 읽기 (oss: oversampling setting, 0~3)
// 기압 측정을 위해 원기 기압 데이터를 읽어오는 함수
int32_t BMP180_ReadRawPressure(I2C_HandleTypeDef *hi2c, uint8_t oss)
{
    // oss는 oversampling setting으로 0~3 사이의 값을 사용
    // 기압 명령어에 oversampling 지트를 추가하기 위해 (oss << 6)을 더함
    uint8_t cmd = BMP180_CMD_READ_PRESS + (oss << 6);
    if (HAL_I2C_Mem_Write(hi2c, BMP180_ADDRESS, BMP180_REG_CONTROL,
            I2C_MEMADD_SIZE_8BIT, &cmd, 1, BMP180_TIMEOUT_MS) != HAL_OK)
    {
        return -1;
    }
    // oversampling 설정에 따라 측정 완료까지 대기해야 하는 시간이 달라짐
    // oss가 0이면 5ms, 1이면 8ms, 2이면 14ms, 3이면 26ms 대기
    // 잘못된 값이 들어올 경우 기본적으로 5ms 대기
    switch(oss)
    {
        case 0: HAL_Delay(5);  break;
        case 1: HAL_Delay(8);  break;
        case 2: HAL_Delay(14); break;
        case 3: HAL_Delay(26); break;
        default: HAL_Delay(5); break;
    }
    //0xF6 주소에서 3바이트를 읽어옴, 이 데이터는 기압 측정의 결과를 포함
    uint8_t raw[3];
    if (HAL_I2C_Mem_Read(hi2c, BMP180_ADDRESS, BMP180_REG_OUT_MSB, I2C_MEMADD_SIZE_8BIT, raw, 3, BMP180_TIMEOUT_MS) != HAL_OK)
    {
        return -1;
    }
    // 읽어온 3바이트 데이터를 결합하여 24비트 정수형 값을 만든 후 oversampling 설정에 따라 오른쪽으로 시프트하여 최종 원시 기압 값(up)을 계산
    // 계산된 up 값을 반환
    int32_t up = (((int32_t)raw[0] << 16) | ((int32_t)raw[1] << 8) | raw[2]) >> (8 - oss);
    return up;
}
```

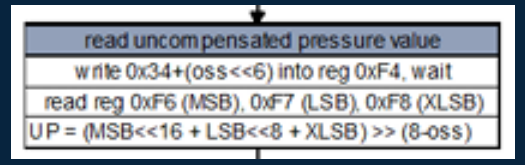


Table 8: Control registers values for different internal oversampling_setting (oss)

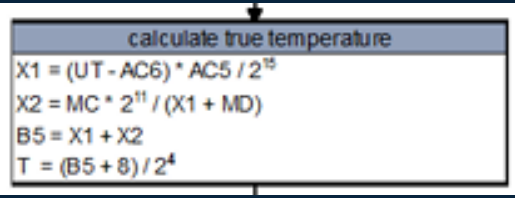
Measurement	Control register value (register address 0xF4)	Max. conversion time [ms]
Temperature	0x2E	4.5
Pressure (oss = 0)	0x34	4.5
Pressure (oss = 1)	0x74	7.5
Pressure (oss = 2)	0xB4	13.5
Pressure (oss = 3)	0xF4	25.5

원시 기압 읽기

- oss << 6 연산을 통해 oversampling 설정 값을 명령어의 상위 비트에 추가.
- HAL_I2C_Mem_Write() 함수 사용.
  - 제어 레지스터(0xF4)에 명령어 전송.
- 센서가 명령어를 받고 실제 압력 데이터를 측정하는 데 걸리는 시간이 oversampling 설정에 따라 달라짐.
- oss 값에 따라 다른 대기 시간 설정 (데이터 시트 참고)
- 측정 완료 후 0xF6 주소부터 3바이트의 데이터 전송.
- 데이터는 MSB부터 순서대로 raw[0], raw[1], raw[2]에 저장.
- 읽어온 3바이트의 데이터를 하나의 24비트 정수형 값으로 결합.

04 코드 설명 (bmp180.c)

```c
// 온도 계산 함수 (BMP180 데이터시트에 따른 계산식)
// 반환값은 0.1°C 단위의 온도 값
// 원시 온도 값(UT)을 이용하여 보정된 온도를 계산하는 함수
int32_t BMP180_ComputeTemperature(int16_t UT)
{
    // 중간 계산에 사용할 변수 X1, X2, B5 선언
    int32_t X1, X2, B5;
    // 데이터 시트에 따른 공식의 일부로 UT에서 AC6을 빼고 AC5를 곱한 후
    // 2^15로 나누기 위해 비트 시프트 연산 (>> 15)을 수행
    X1 = ((UT - AC6) * AC5) >> 15;
    // MC를 2^11(2048)배한 후 X1과 MD의 합으로 나눔
    // 이 계산은 온도 보정을 위한 중간 단계
    X2 = (MC << 11) / (X1 + MD);
    // X1과 X2를 더해 B5값을 얻음, B5는 최종 온도 계산에 중요한 중간 변수
    B5 = X1 + X2;
    // B5에 8을 더한 후 2^4(16)로 나눠 최종 보정 온도 구함
    // 반환 값을 0.1도 단위의 온도
    return (B5 + 8) >> 4;
}
```



온도 계산 함수

- 데이터 시트를 참고하여 온도 계산 함수 작성

X1 계산:

- $X1 = \frac{(UT - AC6) \times AC5}{2^{15}}$

X2 계산:

- $X2 = \frac{MC \times 2^{11}}{X1 + MD}$

B5 계산:

- $B5 = X1 + X2$

최종 온도 계산:

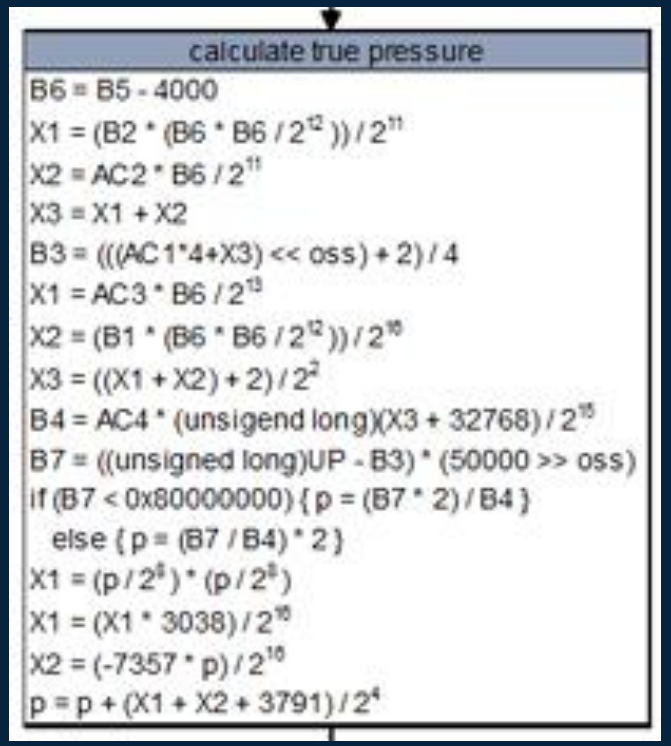
- 온도 (0.1°C 단위) = $\frac{B5 + 8}{16}$

04 코드 설명 (bmp180.c)

```c
// 기압 계산 함수 (BMP180 데이터시트에 따른 계산식)
// 단, 내부에서 온도 보정 값(B5)을 다시 계산합니다.
// 온도 보정을 포함하여 원시 기압(UP)과 온도(UT)를 이용해 보정된 기압 계산
int32_t BMP180_ComputePressure(int32_t UT, int32_t UP, uint8_t oss)
{
    // 여러 중간 변수(X1, X2, B5, B6, X3, p)와 보정용 변수(B4, B7)를 선언
    int32_t X1, X2, B5, B6, X3, p;
    uint32_t B4, B7;

    // 온도 보정 계산
    // 기압 계산 시에도 온도 보정 값(B5)이 필요하므로 앞서 온도 계산과 동일한 공식으로 B5를 재계산
    X1 = ((UT - AC6) * AC5) >> 15;
    X2 = (MC << 11) / (X1 + MD);
    B5 = X1 + X2;
    // 기압 보정 계산
    // B5에서 4000을 빼서 B6을 구함, 기압 보정 공식의 일부
    B6 = B5 - 4000;
    // B6의 제곱을 오른쪽으로 12비트 시프트한 후 B2와 곱하고 다시 2^11(시프트 11)로 나눔
    X1 = (B2 * ((B6 * B6) >> 12)) >> 11;
    // AC2와 B6을 곱한 후 2^11로 나눈 값을 X2에 저장하고 X1과 X2를 더해 X3를 구함
    X2 = (AC2 * B6) >> 11;
    X3 = X1 + X2;
    // AC1에 4를 곱하고 X3를 더한 후 oversampling setting(oss)에 따라
    // 왼쪽 시프트를 수행하고 2를 더한 후 4로 나누어 B3를 계산
    int32_t B3 = (((AC1 * 4 + X3) << oss) + 2) / 4;
    // AC3와 B6를 곱한 값을 오른쪽으로  13비트 시프트하여 X1
    X1 = (AC3 * B6) >> 13;
    // B6의 제곱을 오른쪽으로 12비트 시프트한 후 B1과 곱하고 2^16으로 나눈 값을 X2
    X2 = (B1 * ((B6 * B6) >> 12)) >> 16;
    // X1과 X2를 더하고 2를 더한 후 2^2(4)로 나누어 최종 X3
    X3 = ((X1 + X2) + 2) >> 2;
    // X3에 32768을 더한 후 AC4와 곱하고 오른쪽으로 15비트 시프트하여 B4값을 계산
    // B4는 보정 과정의 분모 역할
    B4 = (AC4 * (uint32_t)(X3 + 32768)) >> 15;
    // 원시 기압 값 UP에서 B3를 뺀 후 50000을 oversampling에 맞게 오른쪽으로 시프트한 값과 곱하여 B7을 계산
    B7 = ((uint32_t)UP - B3) * (50000 >> oss);
    // B7의 값에 따라 두 가지 경우로 나누어 p값을 계산
    // 조건문은 B7이 0x80000000보다 작은지 확인하여 p를 B4로 나누고 2를 곱하는 방식으로 계산
    if(B7 < 0x80000000)
        p = (B7 * 2) / B4;
    else
        p = (B7 / B4) * 2;

    // p를 오른쪽으로  8비트 시프트한 후 제곱하여 X1에 저장하고 다시 3038을 곱한 후 2^16으로 나눔
    X1 = (p >> 8) * (p >> 8);
    X1 = (X1 * 3038) >> 16;
    // X2는 p에 -7357을 곱한 후 2^16으로 나눔
    X2 = (-7357 * p) >> 16;
    // X1, X2 그리고 3791을 더한 후 2^4(16)로 나누어 p에 더함
    // 이 계산을 통해 최종 보정된 기압 값 도출
    p = p + (((X1 + X2 + 3791) >> 4));

    // 계산된 기압 p값을 반환, 단위는 파스칼(Pa)
    return p; // 단위: Pa
}
```

기압 계산 함수

- 데이터 시트를 참고하여 기압 계산 함수 작성



04 코드 설명 (bmp180.c)

```c
// BMP180 센서로부터 데이터를 읽어 LCD와 UART로 출력하는 메인 함수
void bmp_main(void)
{
    // lcd_line1, lcd_line2는 16자를 저장할 문자열 배열, LCD의 각 행에 출력할 내용 저장
    char lcd_line1[17];
    char lcd_line2[17];

    // I2C LCD 초기화
    i2c_lcd_init(); // LCD 초기화 함수

    // BMP180 보정 계수 읽기
    // BMP180 센서의 보정 데이터를 읽어와 전역 변수들(AC1~MD)에 저장
    // hi2c3 핸들러를 사용하여 I2C통신으로 센서의 EEPROM 영역에서 데이터를 읽어옴
    BMP180_ReadCalibrationCoefficients(&hi2c3);

    // 센서 측정 및 출력 작업을 계속 반복하기 위해 무한 루프 시작
    while(1)
    {
        // 원시 온도 및 기압 읽기
        // BMP180_ReadRawTemperature함수를 호출하여 원시 온도 값(UT)를 읽어옴
        int16_t UT = BMP180_ReadRawTemperature(&hi2c3);
        // BMP180_ReadRawPressure함수를 호출하여 원시 기압 값(UP)을 읽음, oversampling setting을 0으로 사용
        int32_t UP = BMP180_ReadRawPressure(&hi2c3, 0);

        // 통신 오류 시 -1 반환된 경우 예외 처리
        if (UT == -1 || UP == -1)
        {
            move_cursor(0, 0);
            lcd_string("                "); // 16칸 공백
            move_cursor(1, 0);
            lcd_string("                "); // 16칸 공백
            sprintf(lcd_line1, "Sensor Error!");
            sprintf(lcd_line2, "Timeout");
            move_cursor(0, 0);
            lcd_string(lcd_line1);
            move_cursor(1, 0);
            lcd_string(lcd_line2);
            printf("BMP180 Timeout or Communication Error\r\n");
            HAL_Delay(1000);
            continue;
        }

        // 온도 계산 (0.1°C 단위)
        // 읽어온 UT값을 이용하여 보정된 온도 계산
        // 반환 값은 0.1도 단위
        int32_t temperature = BMP180_ComputeTemperature(UT);
        // 보정식을 이용한 실제 기압 계산 (Pa 단위 -> hPa로 변환)
        // UT와 UP값을 사용해 보정된 기압 계산
        int32_t pressure = BMP180_ComputePressure(UT, UP, 0) / 100;

        // 첫 번째 줄에 온도값, 두 번째 줄에 기압값을 문자열로 포맷
        sprintf(lcd_line1, "Temp: %2ld.%1ld C", temperature / 10, abs(temperature) % 10);
        sprintf(lcd_line2, "Press: %4ld hPa", pressure);

        // LCD 출력
        move_cursor(0, 0);
        lcd_string(lcd_line1);
        move_cursor(1, 0);
        lcd_string(lcd_line2);
        HAL_Delay(1000);

        // UART를 통해 결과 출력 (온도는 소수 첫째자리까지 출력)
        char msg[100];
        sprintf(msg, "Temp: %2ld.%1ld C, Pressure: %4ld hPa\r\n", temperature / 10, abs(temperature) % 10, pressure);
        printf("%s", msg);
        HAL_Delay(1000); // 1초 주기로 측정
    }
}
```

BMP180 센서로부터 데이터를 읽어 LCD와 UART로 출력하는 메인 함수

- 보정 계수 읽기
  - EEPROM에 저장된 11개의 보정 계수(AC1 ~ MD)를 읽어와 전역 변수에 저장.
- BMP180_ReadRawTemperature() 함수 호출 - 온도 데이터 읽어옴.
- BMP180_ReadRawPressure() 함수 호출 - 압력 데이터 읽어옴.
- LCD, UART 메시지 출력

04 코드 설명 (lcd1602.c)

```c
void LCD_SendCommand(uint8_t cmd)
{
    HAL_GPIO_WritePin(LCD_RS_GPIO_Port, LCD_RS_Pin, GPIO_PIN_RESET);
    LCD_Send4Bits(cmd >> 4);   // 상위 4비트
    LCD_Send4Bits(cmd & 0x0F); // 하위 4비트
    HAL_Delay(2);
}
```

LCD에 명령어를 보내는 함수

RS 핀을 LOW로 설정하여, **명령어 모드**임을 알림.
상위 4비트를 먼저 보내고, 후에 하위 4비트를 보냄.

```c
void LCD_SendData(uint8_t data)
{
    HAL_GPIO_WritePin(LCD_RS_GPIO_Port, LCD_RS_Pin, GPIO_PIN_SET);
    LCD_Send4Bits(data >> 4);
    LCD_Send4Bits(data & 0x0F);
    HAL_Delay(2);
}
```

LCD에 데이터를(문자 등) 보내는 함수

RS 핀을 HIGH로 설정하여, **데이터 모드**임을 알림.

04 코드 설명 (lcd1602.c)

```c
void LCD_Enable(void) {
    HAL_GPIO_WritePin(LCD_E_GPIO_Port, LCD_E_Pin, GPIO_PIN_SET);
    HAL_Delay(1);
    HAL_GPIO_WritePin(LCD_E_GPIO_Port, LCD_E_Pin, GPIO_PIN_RESET);
    HAL_Delay(1);
}
```

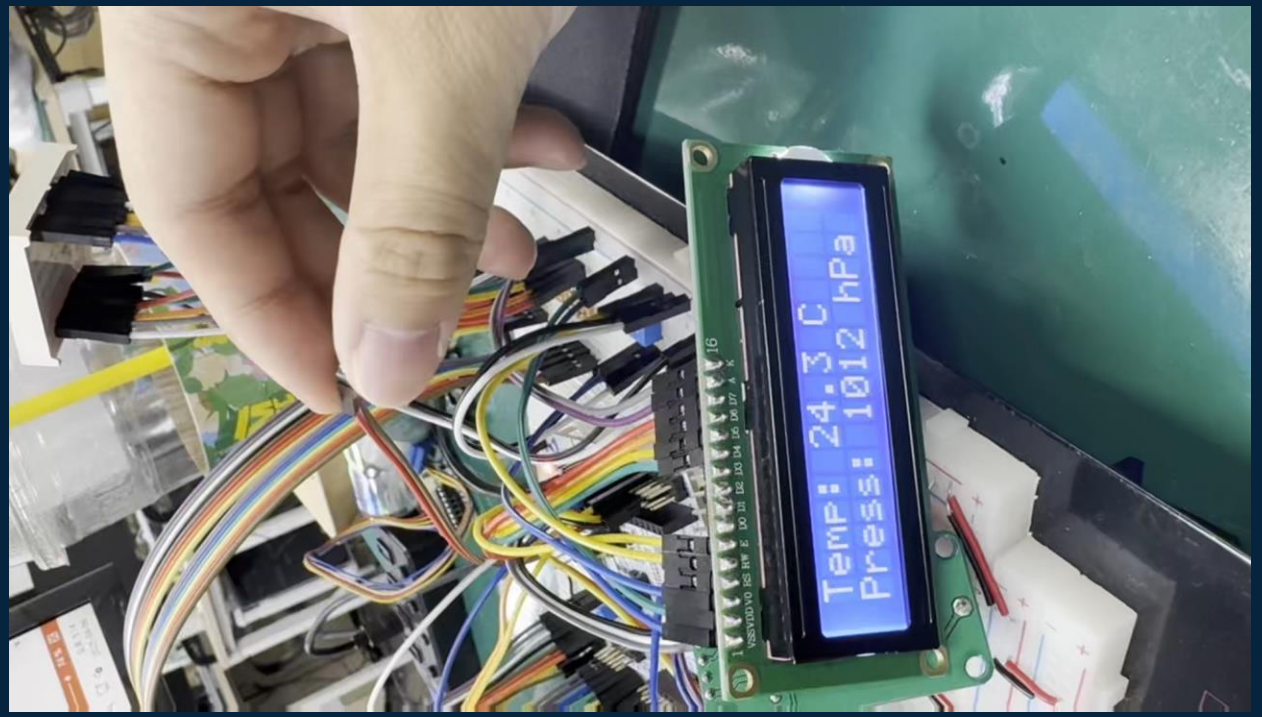
Enable 핀을 짧게 High → Low로 바꿔서 LCD에 데이터를 latch(고정)시키는 역할
LCD는 **E 핀의 Falling Edge**에서 데이터를 읽음.

```c
void LCD_Send4Bits(uint8_t data) {
    HAL_GPIO_WritePin(LCD_D4_GPIO_Port, LCD_D4_Pin, (data >> 0) & 0x01);
    HAL_GPIO_WritePin(LCD_D5_GPIO_Port, LCD_D5_Pin, (data >> 1) & 0x01);
    HAL_GPIO_WritePin(LCD_D6_GPIO_Port, LCD_D6_Pin, (data >> 2) & 0x01);
    HAL_GPIO_WritePin(LCD_D7_GPIO_Port, LCD_D7_Pin, (data >> 3) & 0x01);
    LCD_Enable();
}
```

실제로 4비트 데이터를 LCD에 보내는 함수
입력된 4비트(data)를 각각 D4~D7 핀에 대응시켜 GPIO에 출력
그 후 `LCD_Enable()` 함수를 호출하여 데이터 latch

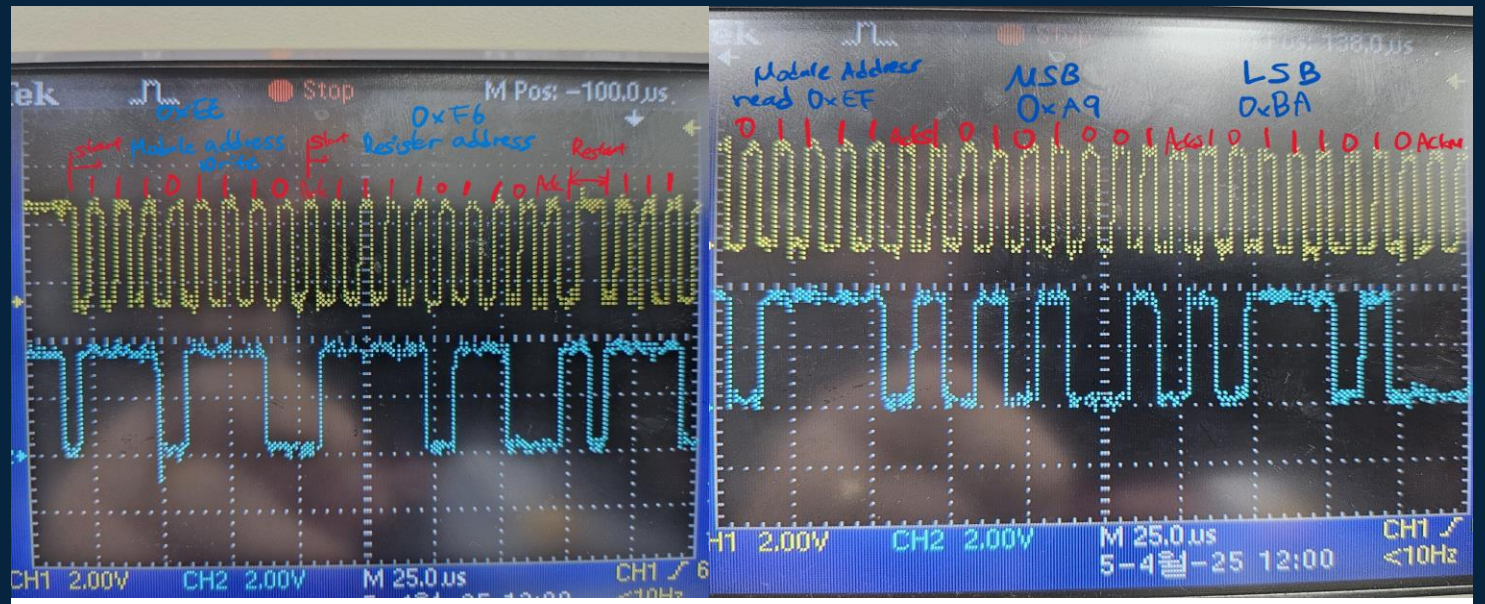
05 결과 및 느낀점

동작 영상 (https://youtube.com/shorts/3SWXgfUtDQU)



05 결과 및 느낀점

오실로스코프 파형 분석 (BMP180)



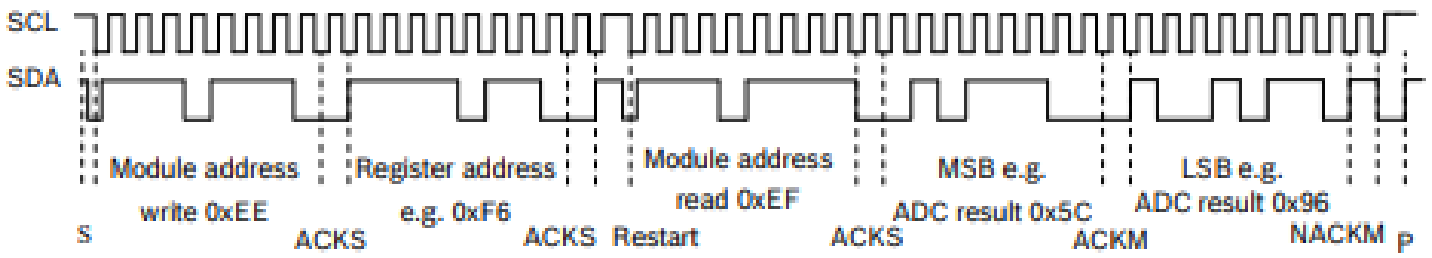
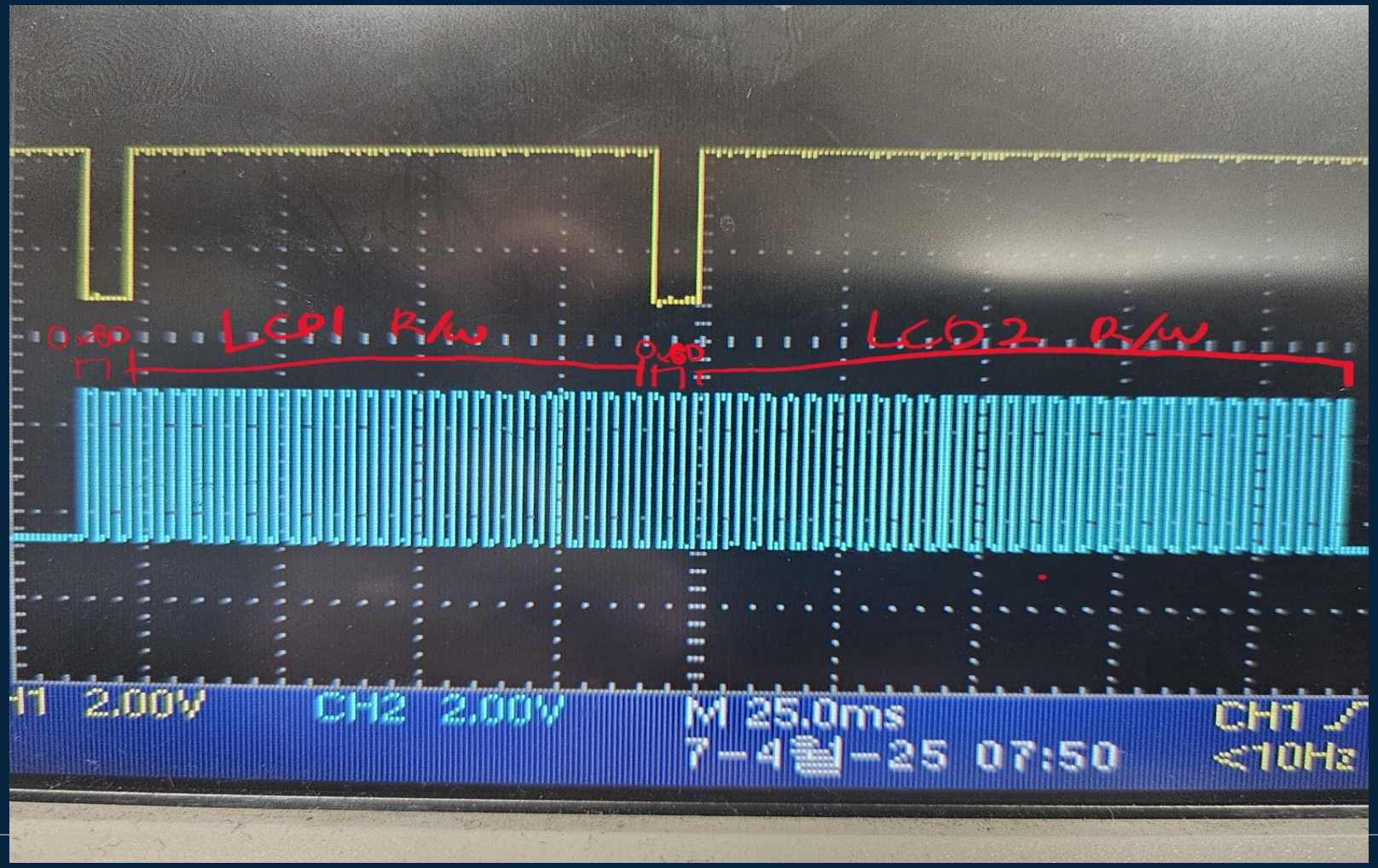


Figure 9: Timing diagram read 16 bit A/D conversion result

05 결과 및 느낀점

오실로스코프 파형 분석 (LCD1602)



05 결과 및 느낀점

느낀점

1. STM32F411RE와 LCD1602A를 직접 결선하고, 드라이버 없이 코드로 제어하는 과정을 통해 임베디드 시스템의 하드웨어와 소프트웨어가 얼마나 밀접하게 연결되어 있는지를 직접 체감함.

2. 4비트 모드로 LCD를 제어하는 방식은 효율적인 GPIO 활용이라는 관점에서 인상 깊었고, 메모리 주소를 직접 다루는 방식이 흥미로움.

3. 라이브러리에만 의존하는 것이 아니라, 하드웨어 동작 원리를 이해한 상태에서 소프트웨어로 제어하는 능력의 중요성을 깨달음.

Thank You

(Telechips) AI 시스템반도체 SW 개발자

박성호 박준영