19-1 시스템 프로그래밍

최종 결과 보고서

<Window 10>

10조

201520882 천윤서

201420881 박규동

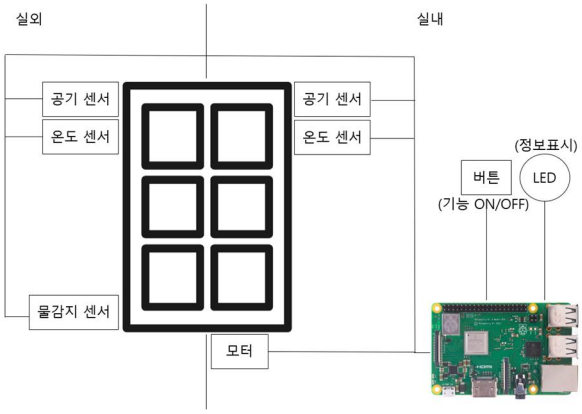
201420966 이레

- 목차 -

6 Challenge Issue 및 소감

팀원 별 프로젝트에서 가장 어려웠던 점을 기술하고 프로젝트 전반적인 소감, 아쉬운 점 등을 종합적으로 기술

1. 프로젝트 개요



[그림] 구 제안서의 시스템 구조도

이번 학기 시스템 프로그래밍의 팀 과제 목표는, 자유로운 주제를 선정하고 그에 맞는 센서들을 시스템 구조상에서 직접 제어하면서 프로젝트를 완성시켜 나가는 것입니다.

저희 조는 자유롭게 브레인 스토밍 기법을 이용해서 제안서에 작성할 주제를 정하면서 의료용 자동 긴급 벨 시스템, 화장실 변기 물내림 감지 시스템, 화재 시 대피 방향 유도 시스템, 수해 재난/화재에 자동으로 개폐되는 문, 창문 자동 개폐 시스템 등을 생각해냈고, 이 중에서 시스템 드라이버를 가장 다양하게 다뤄볼 수 있으며 실용적인 성향이 강한 창문 자동 개폐 시스템인 프로젝트명 "스마트 윈도우"를 선정 하였습니다.

프로젝트를 떠올린 동기는 팀원 중 한 명의 집의 바닥이 나무로 되어있는데, 비가 올 기미가 있는 날에는 나무 바닥이 비에 의해서 곰팡이가 슬 까봐 밤에 창문을 닫고 자야 하는 불편함을 어떻게 해결할 수 있을까 하는 과정에서 떠올리게 되었습니다.

창문은 항상 주변 환경에 대해 열고 닫을 필요가 있는데, 이는 상당히 신경쓰기 귀찮은 일이고, 실제로 환기를 소홀하게 하는 일도 생겨서, 환기를 하자는 공익 관련 광고까지 나오기도 합니다. 저희 조는 온도, 공기 질, 물 감지 세 가지 기능을 이용해서 우선 순위에 따라서 모터를 작동시켜서 창문을 개폐하는 시스템을 만드는 것이 목표였습니다.

이에 따라 한 학기 동안 팀원들이 협동하여 라이브러리 밑의 GPIO 중심의 기능들 만을 이용하여 시스템을 구축하는데 성공하였습니다.

2. 프로젝트 시나리오

프로젝트 목표가 창문의 자동화이기에 사용자가 직접 행동을 해야 하는 것은 On/Off외에 없습니다. 메인 시나리오들은 스마트 윈도우 자체가 판단하여 행동합니다.

1) On/Off 기능

프로그램이 시작된 상황에서 사용자가 버튼을 누르면 모든 동작이 멈춥니다. 이 기능은 실제 상품모델에서는 집에서 외출 시, 사용자가 창문이 열리는 것을 원치 않을 때를 위해 구현했습니다.

2) 온도 비교 기능

프로그램이 시작된 상황에서 실내와 실외의 온도를 비교하여서 환기할지 여부를 결정합니다. 일반적인 상황에서 환기할 때와 유사한 동작입니다.

3) 습도 비교 기능

프로그램이 시작된 상황에서 실내와 실외의 습도를 비교하여 환기할지 여부를 결정합니다. 보통 헬스장 등에서 운동을 할 경우 실내의 습도가 높아져서 불쾌지수가 상승하는 경우가 있는데 그 경우를 상정하였습니다.

4) 비 감지 기능

프로그램이 시작된 상황에서 밖에 비가 내리는지 여부를 감지하고 창문을 닫습니다. 나무바닥으로 되어있는 실내나, 가구가 창문에 인접한 경우 등 다양한 경우에 대응됩니다.

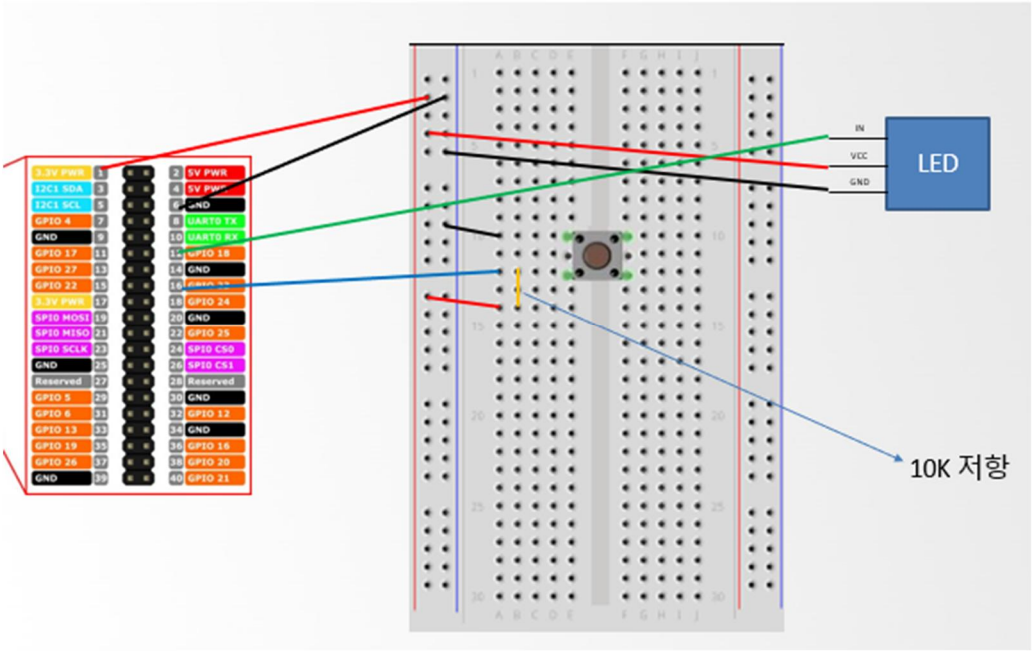
5) 미세 먼지 농도 감지 기능

미세먼지 농도가 지나치게 높은 경우, 창문을 여는 행동을 억제합니다. 오히려 환기하는 것이 건강에 더 악영향을 미칠 수 있는 상황을 가정하였습니다.

3. 시스템 설계

1)하드웨어 구조

a)버튼



[그림] 실습자료-버튼

버튼은 실습용으로 받은 버튼을 그대로 사용하였습니다. 사용자가 기능을 제어하기 위해 사용하기 위해 사용됩니다. 특이 사항으로는 제대로 사용하려면 10K 저항을 사용해야 한다는 점입니다. Button이 On되면 GPIO는 LOW를 감지하고, OFF되면 HIGH를 감지합니다. 3V의 VCC를 필요로 합니다.

b)Servo Motor

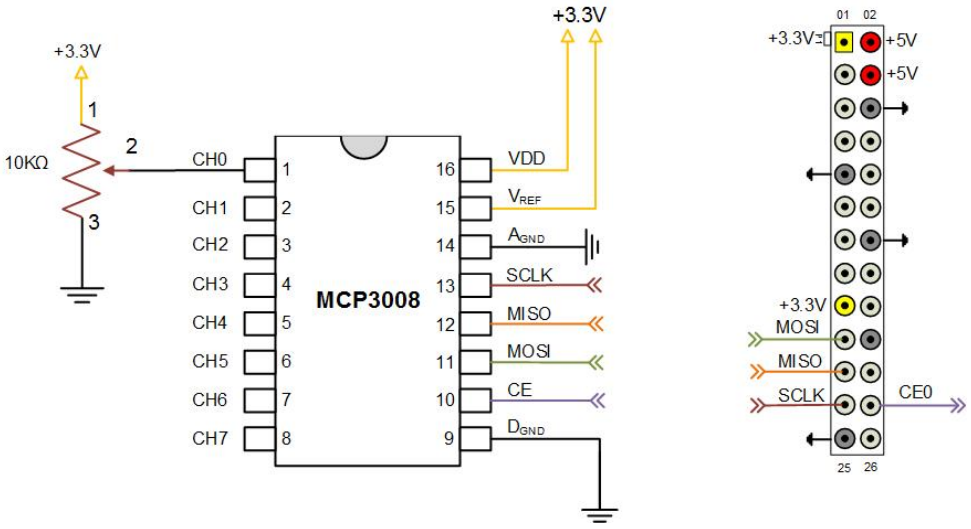
서보 모터는 실습용으로 받은 서보 모터를 그대로 사용하였습니다. PWM을 이용하여 작동하는 서보 모터는, 특정 PWM 입력에 대응하는 특정 각도를 지니고 작동합니다. 자세한 내용은 시스템 구현에서 기술하겠습니다. 저희 프로젝트에서는 창문을 열고 닫음을 표현하기 위해 사용했습니다.

c)온습도 센서

온습도 센서는 온도와 습도를 감지한 다음 아날로그 값이 아닌 디지털 값을 출력합니다. 따라서 ADC 같은 추가적인 구성품이 필요하지 않습니다. 온습도 센서 관련 내용 역시 시스템 구현에서 자세히 다루도록 하겠습니다. 저희 프로젝트에서는 창문 내외부의 온도와 습도를 비교하는 동작을 수행하기 위

해서 사용하였습니다.

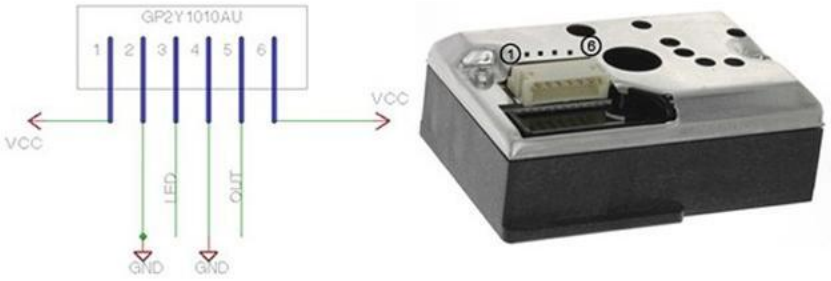
d)수위 센서



[그림] MCP3008 회로도

수위 센서는 물이 은색의 선에 닿을 때의 저항 변화를 통해서 아날로그 값을 돌려줍니다. 라즈베리파이는 이런 아날로그 값을 읽을 수 없기 때문에 MCP3008을 사용해서 디지털 값으로 변환해서 돌려 주어야합니다. MCP3008은 SPI 통신을 이용하며 해당 통신에 필요한 핀들을 라즈베리 파이와 연결해주어야 합니다. 수위 감지 센서는 창 밖에 비가 오는 것을 감지하기 위해서 사용하였습니다.

e)미세 먼지 센서



[그림] GP2Y10 미세먼지 센서와 핀

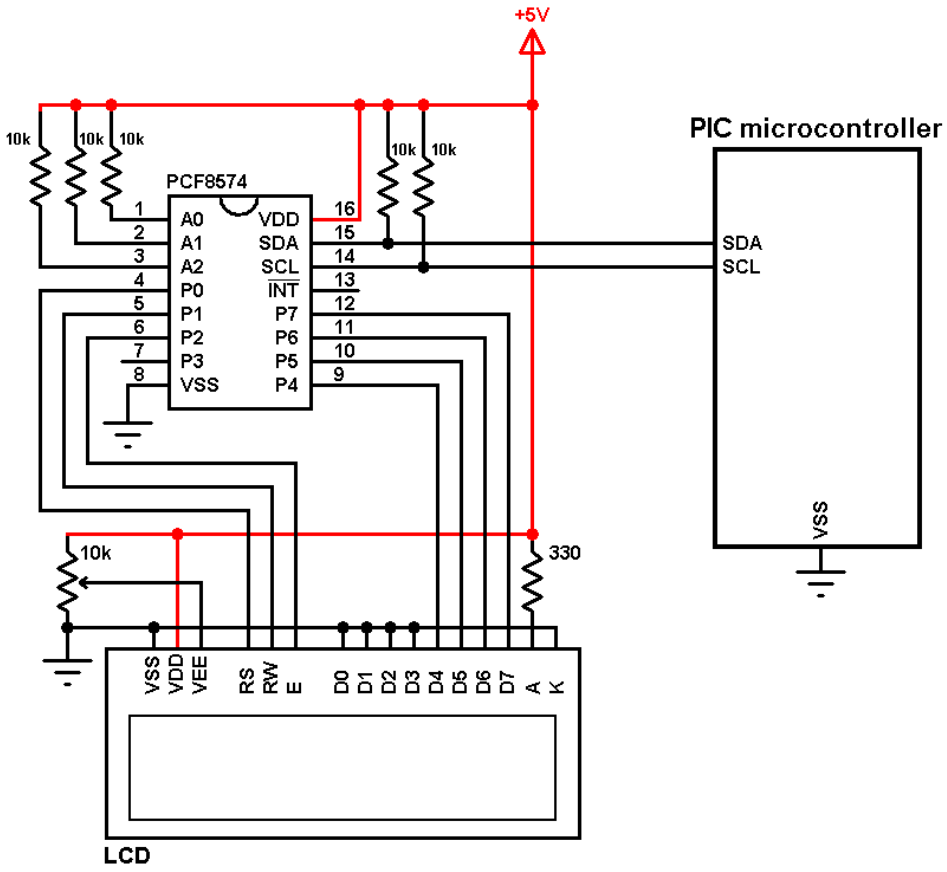
미세먼지 센서는 공기 중의 미세 먼지 농도를 전압에 비례한 출력으로 나타내어줍니다. 총 6개의 선을 사용하는데, 이는 LED로 비춘 뒤, 미세먼지에 의해 산란된 빛을 검출하는 동작 원리상, LED와 검출

기 두가지 파트로 센서가 구성되어 있기 때문입니다.

LED는 출력이 LOW일 때 활성화되고, HIGH일 때 비활성화 되는 특성을 가지며, LED와 검출기 간의 타이밍을 데이터 시트를 읽고 정확하게 맞춰야만 미세먼지를 검출할 수 있습니다.

미세먼지 센서는 창밖의 미세먼지의 농도를 감지하고, 창문을 열고 닫을지 여부를 검사하기 위해서 사용하였습니다.

f) LCD



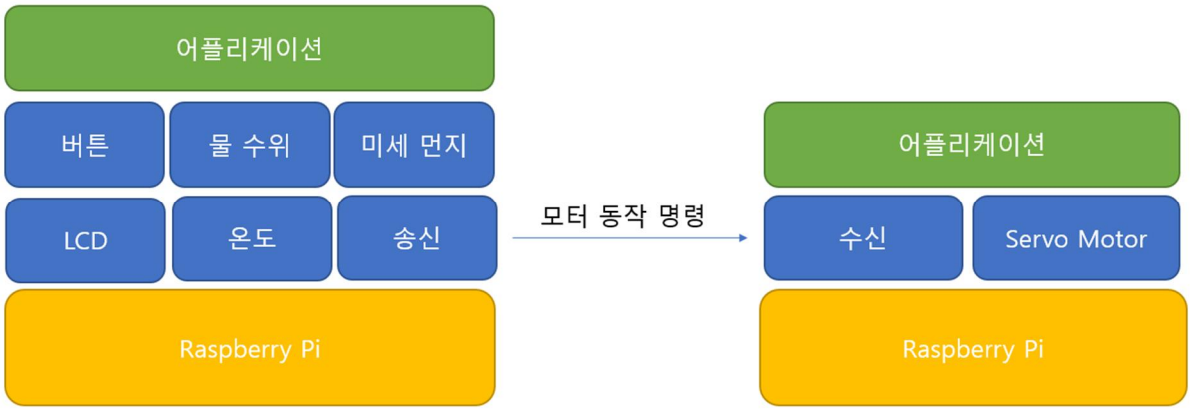
[그림] A1602 LCD의 회로도

LCD는 사용자에게 정보를 전달하기 위한 목적으로 사용했습니다. LCD를 사용할 수 있는 방법은 세가지가 있습니다. 가장 쉬운 방법이 8개의 Pin(D0~D7)을 전부 써서 8-pin 모드로 작동시키는 게 그 방식이고, 이 방식은 최악의 방식으로, GPIO 핀을 10핀 이상 사용하면서 너무 심하게 낭비되기에 이게 원인이 돼서 드라이버를 제대로 구현 못한 조도 있었습니다. 두 번째 방식은 4-pin 모드로 동작시키는 방식입니다. 백 라이트까지 포함한다면 8개의 GPIO 핀을 사용하고, 백 라이트를 제외한다면 7개의 GPIO 핀만으로 동작이 가능합니다. 4-pin 모드로 구동하려면 4개의 비트로 8비트짜리 데이터를 표현할 수 있어야 하기에 데이터를 데이터 시트에 따라서 적절히 처리해야합니다. 가장 최선의 방식은 LCD 기판 뒷면에 달린 PCF8475를 사용해서 통신하는 방식입니다. 2개의 GPIO 핀을 사용하여서 회로가 단순해

지지만, I2C 통신을 사용해야 하며, 중간 역할을 하는 장치가 끼어들어서 간접적인 동작을 하는 만큼 난이도가 어려워집니다.

LCD는 두 줄을 지니고 있고, 캐릭터 테이블이라는 내부에 저장된 테이블을 통해서 화면에 표시되는 글자를 가져옵니다. RS, RW, E는 명령과 동작 모드에 사용되는 핀이고, D0~D7은 데이터 핀입니다.

2) 소프트웨어 구조



소프트웨어는 두 대의 라즈베리 파이가 협업을 하면서 동작합니다. 메인 라즈베리파이는 측정 역할을 합니다. 온습도, 버튼, 물 감지, 미세먼지 감지 등의 기능을 수행하며, 우선 순위를 비교해서 결과를 LCD에 출력하여 보여주며, 2번째 라즈베리파이로 Servo Motor가 수행해야할 동작을 GPIO 핀을 통해서 전송합니다. 이렇게 제안서와 달리 하나의 라즈베리파이를 굳이 더 쓴 이유는, 창문이 여러 개가 있음을 가정했을 때 측정/동작 단을 분리해서 만드는 것이 효율적이라 생각했기 때문입니다.

측정 관련 시스템은 단 하나만 필요하지만, 창문 개수는 어떤 건물의 경우 수많은 창문이 있을 수 있고, 따라서 동작 관련 시스템(Servo Motor)은 여러 군데에 붙을 수가 있습니다.

측정 관련 시스템은 상당히 복잡하기에 비동기적으로 센서 값을 측정하기 위해서 따로 스레드를 지니고 돌아가며, 사용자의 입력을 비동기적으로 받기 위한 버튼 또한 독자적으로 스레드를 따로 가지고 돌아갑니다. LCD와 동작명령은 메인 프로그램에서 돌아가게 됩니다. 동작을 담당하는 라즈베리파이는 기능이 수신과 Servo Motor작동 밖에 없기에 단일 스레드로 동작합니다.

4. 시스템 구현

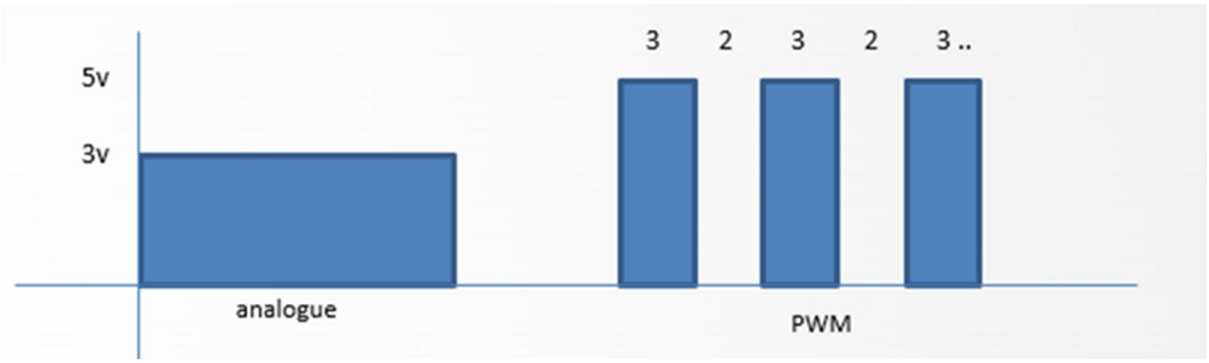
1)디바이스 드라이버

a) 버튼 드라이버

버튼 드라이버는 실습 시간에 진행했던 것과 거의 유사한 코드를 사용하였습니다. 차이점이 있다면 핀 번호가 바뀐 정도입니다. GPSEL을 Input 모드로 설정하기위해서 해당하는 비트를 '000'으로 고쳐주어야 하고, GPLEV을 통해서 값을 읽습니다. 실습 시간에 자세히 강의하셨던 내용이기 때문에 큰 설명 없이 넘어가도록 하겠습니다.

b) Servo Motor 드라이버

servo Motor는 PWM의 output의 밀도에 따라 모터의 각도를 조절합니다. 이를 창문에 결합시킨다면 창문의 개폐를 할 수 있습니다.



위의 그림은 pwm의 아웃풋이 어떻게 진행되는지를 보여주는 그림입니다. 왼쪽과 오른쪽의 전압의 양은 같습니다. 하지만 출력방식이 다르게 진행됩니다. 아날로그를 균일하게 3V로 출력하는 것과 PWM

방식의 5V를 간헐적으로 출력하는 것입니다.

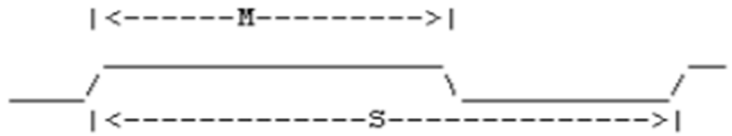
PWM은 위의 사진처럼 전압을 보냈다 안보냈다를 반복합니다. PWM의 전압을 보내는 방식에는 여러 방식이 있습니다. 대표적으로 N/M방식과 M/S방식입니다.

Bad	0	0	0	0	1	1	1	1	0	0	0	0
Fair	0	0	1	1	0	0	1	1	0	0	1	1
Good	0	1	0	1	0	1	0	1	0	1	0	1

위의 사진은 N/M방식을 보여주는 것입니다. duty cycle동안 serial channel을 따라서 전송됩니다. N/M 모드는 M이라는 cycle동안 N개의 output 값이 1이 되어야 합니다. 위에 사진을 살펴보시면 8개가 전송될 때는 반드시 1이 4개가 되는 것을 확인할 수 있습니다. 위의 N/M은 4/8로 볼 수 있습니다. 특정 cycle 동안 output값의 1의 개수가 정해져 있으므로 위와 같이 출력의 경우의 수가 다양한 것을 확인할 수 있습니다.

```
1. Set context = 0
2. context = context + N
3.  if (context >= M)
        context = context – M
        send 1
    else
```

위의 그림은 N/M모드로 전송 시 1과 0의 출력을 교차로 보내기 위해 고안된 알고리즘입니다. 이 알고리즘을 사용하면 'Good' 상태로 N/M모드 output출력을 할 수 있습니다.



위의 사진은 PWM의 M/S모드 사용 그림입니다. M은 data값이며 S는 range입니다. M/S모드는 고주파 변조가 필요하지 않은 경우와 negative 효과가 있는 경우에 바람 직 합니다. 또한 데이터 레지스터가 사용되거나 버퍼가 사용되고 비어 있지 않은 한 채널은 출력을 계속 보냅니다.

	PWM0	PWM1
GPIO 12	Alt Fun 0	-
GPIO 13	-	Alt Fun 0
GPIO 18	Alt Fun 5	-
GPIO 19	-	Alt Fun 5
GPIO 40	Alt Fun 0	-
GPIO 41	-	Alt Fun 0
GPIO 45	-	Alt Fun 0
GPIO 52	Alt Fun 1	-
GPIO 53	-	Alt Fun 1

PWM channel은 위의 그림과 같습니다. 저희는 GPIO 18과 GPIO 12를 둘 다 사용해 보았습니다.

Address	Field Name	Description	Size	Read/ Write
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0000	GPFSEL0	GPIO Function Select 0	32	R/W
0x 7E20 0004	GPFSEL1	GPIO Function Select 1	32	R/W

```
1       #define GPIO_BASE_ADDR 0x3F200000
2       #define GPFSEL1 0X04
```

Bit(s)	Field Name	Description	Type	Reset
31-30	---	Reserved	R	0
29-27	FSEL19	FSEL19 - Function Select 19 000 = GPIO Pin 19 is an input 001 = GPIO Pin 19 is an output 100 = GPIO Pin 19 takes alternate function 0 101 = GPIO Pin 19 takes alternate function 1 110 = GPIO Pin 19 takes alternate function 2 111 = GPIO Pin 19 takes alternate function 3 011 = GPIO Pin 19 takes alternate function 4 010 = GPIO Pin 19 takes alternate function 5	R/W	0
26-24	FSEL18	FSEL18 - Function Select 18	R/W	0
23-21	FSEL17	FSEL17 - Function Select 17	R/W	0
20-18	FSEL16	FSEL16 - Function Select 16	R/W	0
17-15	FSEL15	FSEL15 - Function Select 15	R/W	0
14-12	FSEL14	FSEL14 - Function Select 14	R/W	0
11-9	FSEL13	FSEL13 - Function Select 13	R/W	0
8-6	FSEL12	FSEL12 - Function Select 12	R/W	0
5-3	FSEL11	FSEL11 - Function Select 11	R/W	0
2-0	FSEL10	FSEL10 - Function Select 10	R/W	0

Table 6-3 – GPIO Alternate function select register 1

GPIO 12와 GPIO 18은 GPIO Alternate function select register 1입니다. PWM 채널에 맞추어 GPIO 12는 ALT function select 12에서 alternate function 0인 100을 선택해야 합니다. GPIO 18은 ALT function select 18에서 alternate function 0인 010을 선택해야 합니다. 아래의 코드가 저희 디바이스 코드에서 작성된 부분입니다.

```
*gpsel1 &= ~(1<<1);    //GPIO 12
*gpsel1 |= (1<<8);
*gpsel1 &= ~(1<<1);


*gpsel1 &= ~(1<<1);    //GPIO 18
*gpsel1 |= (1<<25);
*gpsel1 &= ~(1<<1);
```

PWM Address Map			
Address Offset	**Register Name**	**Description**	**Size**
0x0	CTL	PWM Control	32
0x4	STA	PWM Status	32
0x8	DMAC	PWM DMA Configuration	32
0x10	RNG1	PWM Channel 1 Range	32
0x14	DAT1	PWM Channel 1 Data	32
0x18	FIF1	PWM FIFO Input	32
0x20	RNG2	PWM Channel 2 Range	32
0x24	DAT2	PWM Channel 2 Data	32

위의 그림은 PWM의 레지스터입니다. 위의 그림에서 저희가 사용하는 부분은 CTL, RNG1, DAT1 입니다. CTL은 PWM Control과 관련이 있습니다. PWM의 환경을 설정하는 레지스터입니다. DAT1은 PWM channel 1 data입니다. 저희가 channel1만을 사용했기 때문에 DAT1을 사용했습니다. RNG1은 PWM channel 1 range입니다.

7	MSEN1	Channel 1 M/S Enable 0: PWM algorithm is used 1: M/S transmission is used.	RW	0x0
6	CLRF1	Clear Fifo 1: Clears FIFO 0: Has no effect This is a single shot operation. This bit always reads 0	RO	0x0
5	USEF1	Channel 1 Use Fifo 0: Data register is transmitted 1: Fifo is used for transmission	RW	0x0
4	POLA1	Channel 1 Polarity 0 : 0=low 1=high 1: 1=low 0=high	RW	0x0
3	SBIT1	Channel 1 Silence Bit Defines the state of the output when no transmission takes place	RW	0x0
2	RPTL1	Channel 1 Repeat Last Data 0: Transmission interrupts when FIFO is empty 1: Last data in FIFO is transmitted repetedly until FIFO is not empty	RW	0x0
1	MODE1	Channel 1 Mode 0: PWM mode 1: Serialiser mode	RW	0x0
0	PWEN1	Channel 1 Enable 0: Channel is disabled 1: Channel is enabled	RW	0x0

위의 사진은 PWM CTL 레지스터와 관련된 그림입니다. 왼쪽에 보이는 숫자는 CTL 레지스터의 bit 위치입니다. 위 bit의 값을 변경하여 원하는 PWM CTL 모드를 설정합니다. 저희는 위의 그림에서 PWEN1, MODE1, MSEN1을 다루었습니다.

PWEN1는 일치하는 채널을 enable 혹은 disable합니다. 이를 통해서 원하는 채널은 끄고 켤 수 있습니다. bit를 1로 세팅하면 channel과 transmitter state machine을 활성화시킵니다.

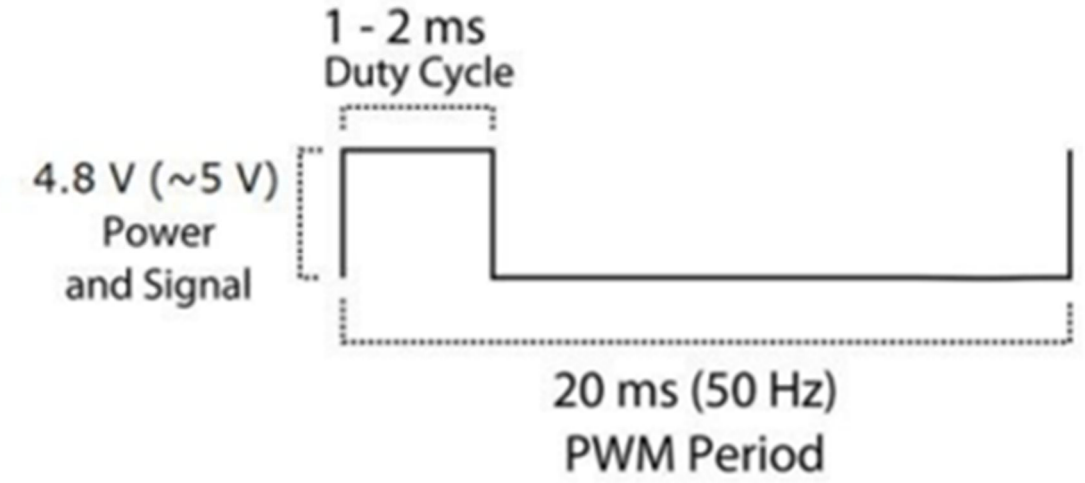
MODE1은 작동의 방식을 결정하는 bit입니다. 0으로 세팅하는 것은 PWM mode를 뜻합니다. 1은 Serial 모드를 선택하는 것입니다. 저희는 PWM 모드를 사용하기 때문에 위 bit를 0으로 설정하였습니다.

MSEN1 M/S모드와 N/M모드를 결정하는데 사용됩니다. 저희는 M/S모드를 사용하였으므로 위 bit를 1로 바꾸었습니다.

31:0	PWM_RNGi	Channel i Range	RW	0x20
31:0	PWM_DATi	Channel i Data	RW	0x0

다음은 RNG와 DAT값을 변경해야 했습니다. 이를 변경하는 이유는 RNG와 DAT의 비율을 통해서 PWM의 모터의 각도가 정해지기 때문입니다. 실습시간에 주어진 코드를 바탕으로 진행한다면 RNG는 320으로 정해져 있습니다. 아래의 그림을 보시면 PWM period를 RNG로 Duty Cycle을 DAT로 해석하면 됩니다. 즉 data는 16~32 사이의 값을 집어넣으면 됩니다. 16은 -90도이고 32는 +90도입니다.

Position "0" (1.5 ms pulse) is middle, "90" (~2ms pulse) is middle,
is all the way to the right, "-90" (~1ms pulse) is all the way to the left.




```
*gpsel1 &= ~(1<<1);
*gpsel1 |= (1<<25);
*gpsel1 &= ~(1<<1);
```

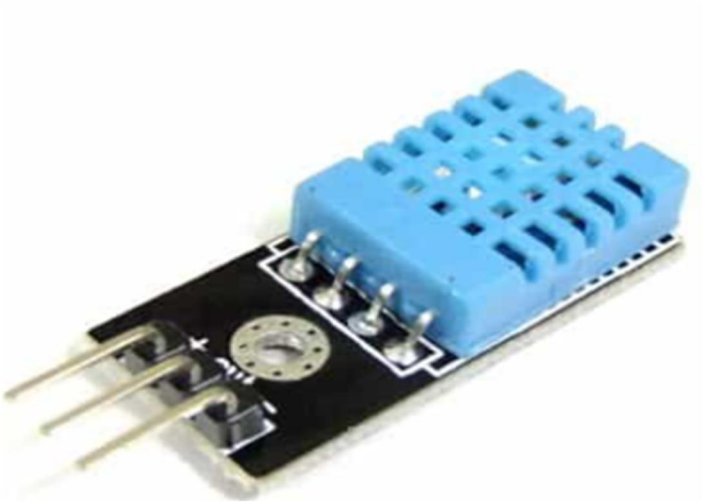
```
*pwmctl |= (1);     //PWEN      1
*pwmctl &= ~(1<<1);   //MODE1     0
*pwmctl |= (1<<7);   //MSEN1     MS  1

*pwmrng1 = 320;     //RANGE     320
*pwmdat1 |= (1<<5);   //DAT     32  (1/10)
```

위의 그림에서 보시면 PWM period는 20ms(50Hz)로 되어있는 것을 확인할 수 있습니다. 위의 그림과 같은 형태로 진행하기 위해선 clk_pwm에 의해서 정의되어 있는 100MHz를 조절해야 합니다. 이는 clock manager를 통해 진행됩니다. BCM 2837을 보시면 CPRMAN이라는 어휘를 볼 수 있는데 clk과

관련된 것입니다. 100MHz는 특정 div을 통해서 줄여져 50hz가 되어야 합니다. 아래의 코드가 그 과정입니다.

```
int init_pwm(void) {
    int pwm_ctrl = *pwmctl;
    *pwmctl = 0; // store PWM control and stop PWM
    msleep(10);//
    *clkctl = BCM_PASSWORD | (0x01 << 5); // stop PWM Clock
    msleep(10);//

    int idiv = (int)(19200000.0f / 16000.0f); // Oscilloscope to 16kHz
    *clkdiv = BCM_PASSWORD | (idiv << 12); // integer part of divisior register
    *clkctl = BCM_PASSWORD | (0x11); //set source to oscilloscope & enable PWM CLK

    *pwmctl = pwm_ctrl; // restore PWM control and enable PWM
}
```

c) 온습도 드라이버

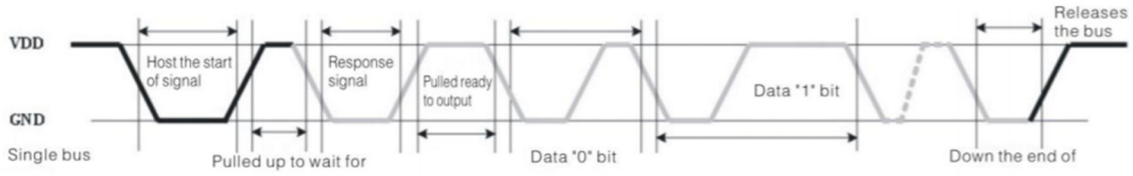


dht11 모델의 온습도 센서를 사용하였습니다. 이 센서는 온도와 습도를 디지털 값으로 반환합니다. 이 디지털 값을 읽어 들여서 사용자가 원하는 온습도의 정보를 알 수 있습니다.

VDD의 파워는 3.5~5.5V를 지원하므로 파이의 있는 모든 출력 전압에 대해서는 지원합니다. LCD는 5V의 전압만을 지원하는 정보와 다릅니다. 데이터는 serial data이며 single bus입니다.

습도의 정확성은 25도인 경우 위아래로 5%의 오차를 가집니다.

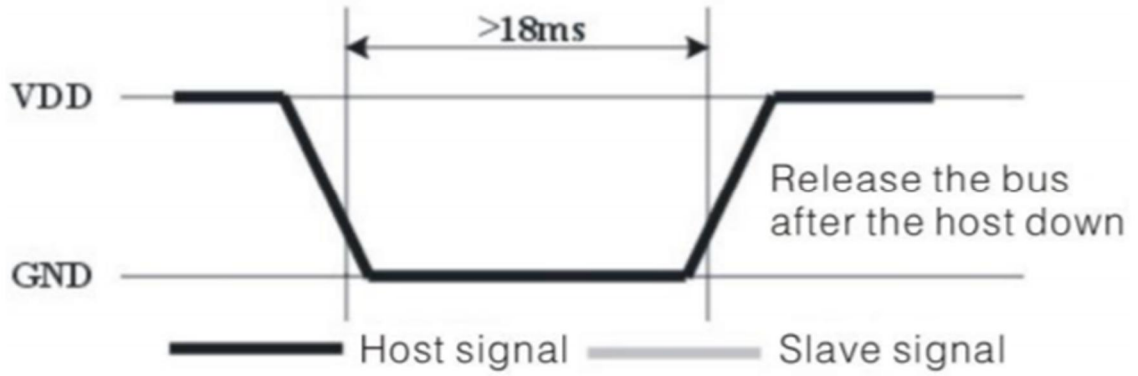
온도의 정확성은 25도인 경우 위아래로 2도의 오차를 가집니다.



위의 그림은 온습도의 DATA Timing Diagram입니다. 굵은 글씨로 작성된 부분은 User host(MCU)가 보내는 신호입니다. 이 신호를 보내게 되면 DHT11은 저전력(low-power) 모드에서 고속 모드(high-speed)로 변환되게 됩니다.

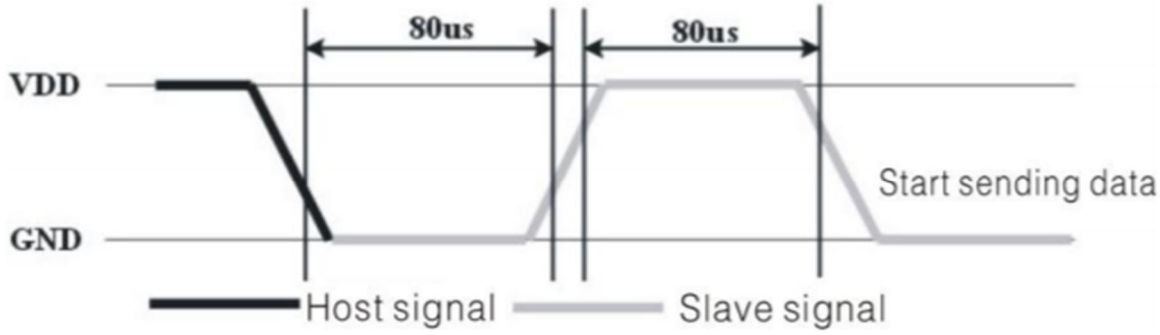
모드가 변환된 DHT11은 반응 신호(Response signal)와 준비 신호(ready to output)를 거쳐 데이터를 전송하게 됩니다.

이와 관련된 코드 작성은 레지스터 하드웨어를 다루기 보다 소프트웨어 적인 방식으로 진행되었습니다. 소프트웨어 적이란 gpclr와 gpset, gplev 이 3가지를 delay를 주어 처리하는 방식입니다. 수업시간에 조교님이 알려주신 정의입니다.



위의 그림은 DHT의 모드를 저전력 모드에서 고속 모드로 변환하는 과정입니다. Host(MCU)는 라즈베리파이입니다. 라즈베리파이는 OUTPUT모드로 설정합니다. 후에 설정된 핀의 value를 LOW로 떨어뜨립니다. 떨어진 LOW값의 시간은 18ms보다 길어야 합니다. 이를 위해 mdelay(18)을 걸어 LOW의 지속시간을 유지합니다. 이 작업이 끝나고 난뒤 핀의 value를 HIGH로 올립니다. udelay(40)을 이용해 40마이크로초동안 HIGH값을 유지한 뒤 라즈베리파이를 IN모드로 변경합니다.

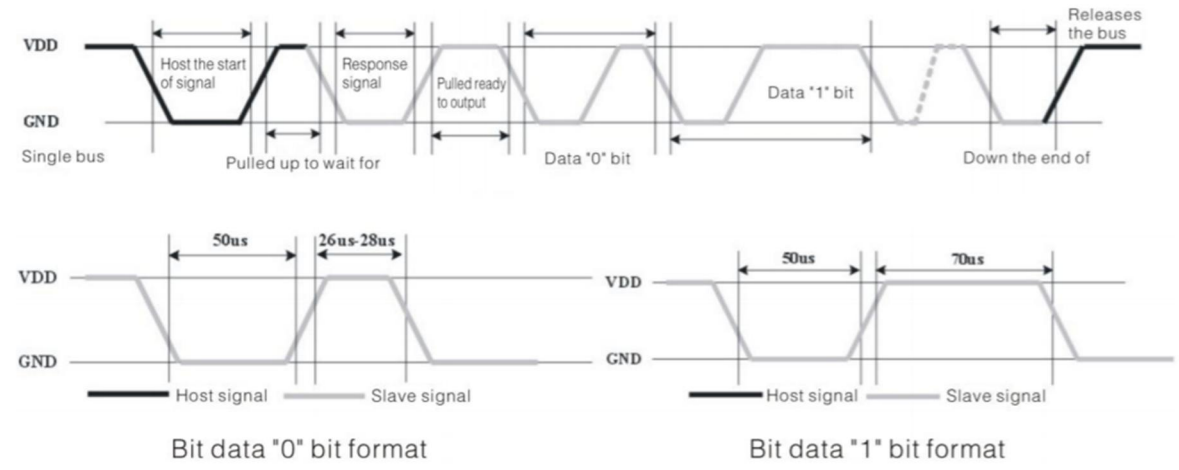
```
*gpsel1 |= (1<<21);    //pinMode(DHpin, OUTPUT);

*gpclr1 |= (1<<17);    //digitalWrite(DHpin, LOW); // bus down, send start signal
mdelay(18);        // delay greater than 18ms, so DHT11 start signal can be detected
*gpset1 |= (1<<17);    //digitalWrite(DHpin, HIGH);
udelay(40);        // Wait for DHT11 response


*gpsel1 &= ~(1<<21);    //pinMode(DHpin, INPUT);
```




```
if(((*gplev0) &( 1 << 17)))      //udelay1씩 주어서 신호가 몇 초간 지속되었는지 확인
    break;
udelay(1);
counter++;
```

위 그림은 응답 신호(Response signal)과 output준비 신호(Pulled ready to output)를 보여주는 단계입니다. 이는 IN모드로 설정된 라즈베리파이가 감지해야 합니다. 80마이크로 세컨드의 LOW가 진행된

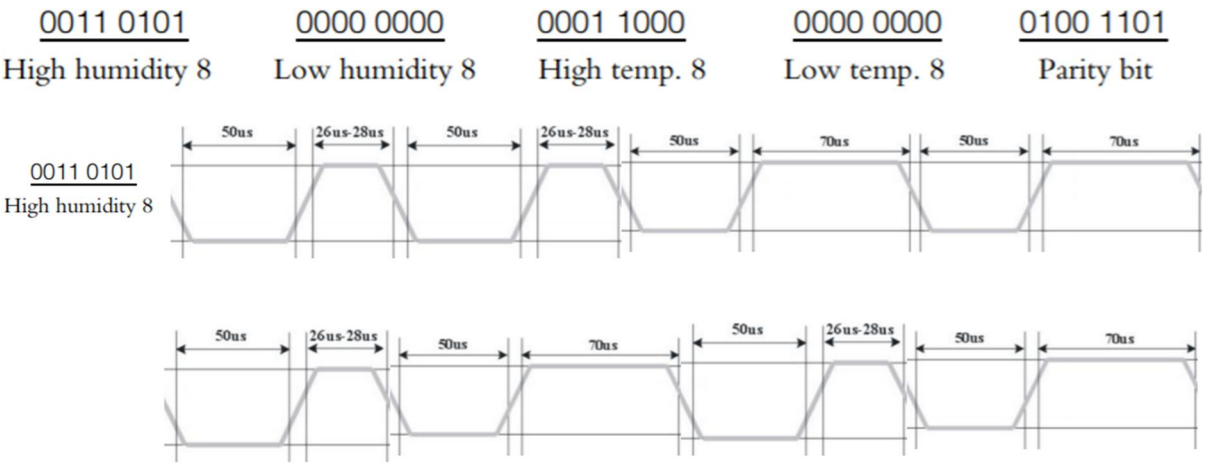
후 80마이크로 세컨드의 HIGH를 받는 것은 확인해야합니다.



위의 그림은 0과 1을 구분하는 방법을 보여줍니다. 자세히 보시면 HIGH인 상태의 시간으로 0과 1을 구분하는 것을 확인할 수 있습니다.

아래의 그림은 한 번의 data 측정이 완료된 예시입니다. 총 40개의 bit가 들어온 것을 확인할 수 있습니다. High humidity는 습도의 정수 부분, Low humidity는 습도의 소수 부분, High temp는 온도의 정수 부분, Low temp는 온도의 소수 부분입니다. 이 4개의 비트에 오류가 있는지 확인하기 위해 Parity bit를 사용합니다.

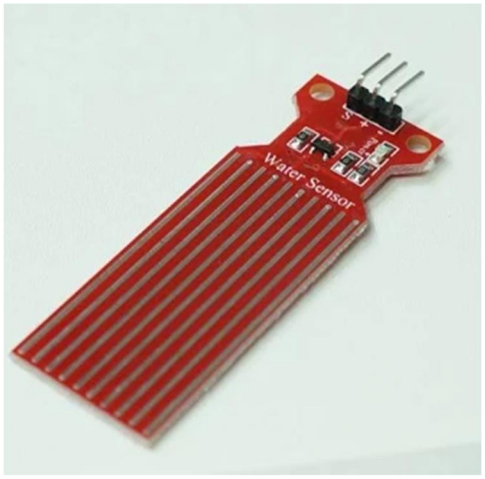
High huminity를 00110101로 표현되었을 때의 그림을 아래에 표현하였습니다.



아래의 코드는 4개의 데이터를 parity bit를 이용해 처리하는 코드입니다.

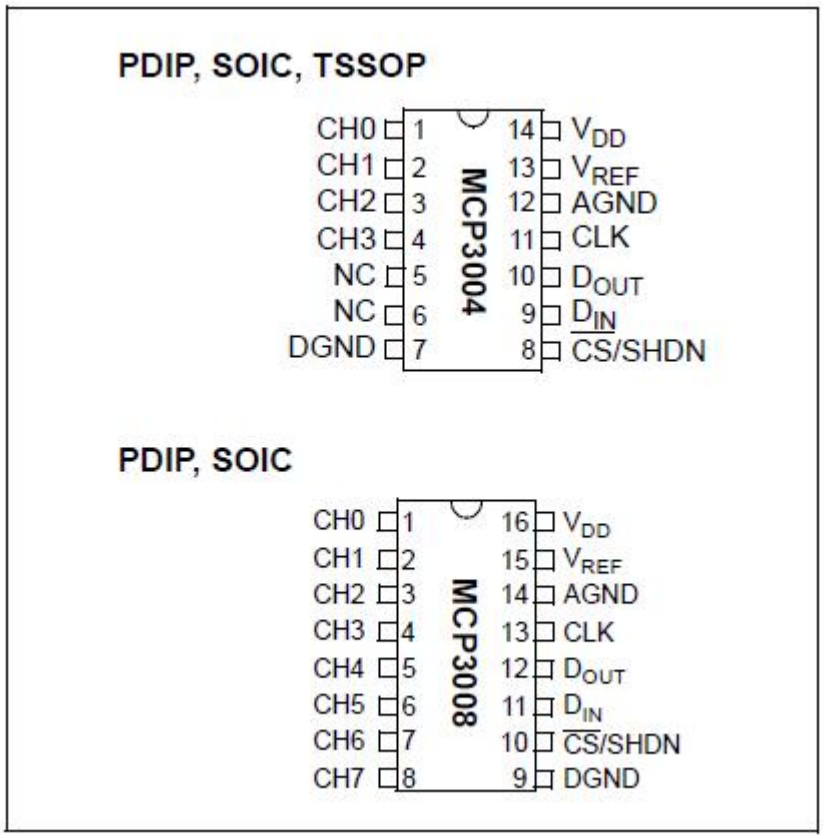
```
if ((dht11_dat[4] == ( (dht11_dat[0] + dht11_dat[1] + dht11_dat[2] + dht11_dat[3]) & 256) )
{

    printk(KERN_ALERT "Humidity = %d.%d %% Temperature = %d.%d ₩n",
        dht11_dat[0], dht11_dat[1], dht11_dat[2], dht11_dat[3]);
```

d) 수위 센서 드라이버



[그림] 수위 감지 센서.

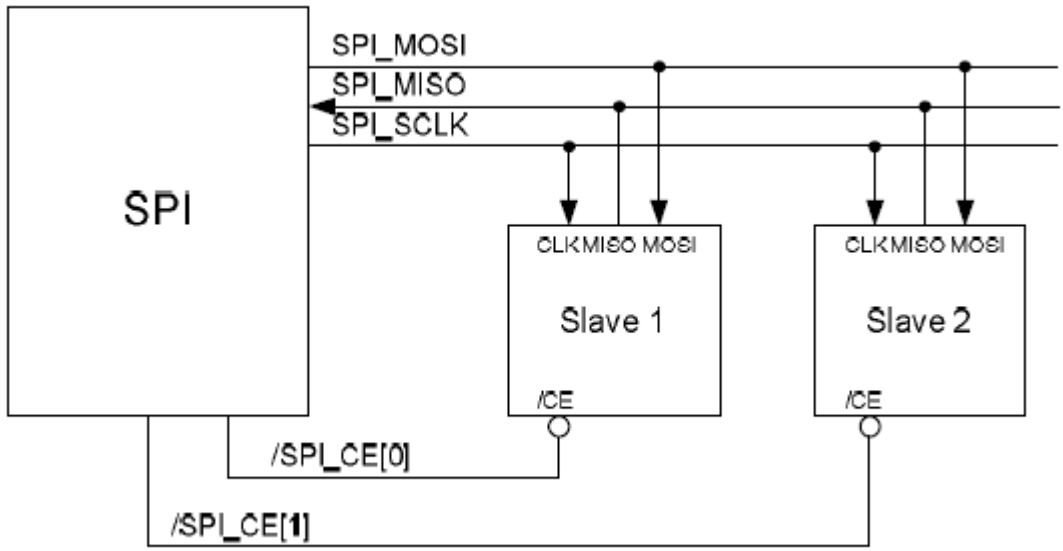
수위 센서 드라이버는 물이 은색의 선에 닿았을 때 저항의 변화를 통해서 아날로그 값을 돌려줍니다. 하지만, 라즈베리파이는 아날로그 값을 읽을 수 있는 기능이 없이, 디지털 값만이 읽을 수 있습니다. 이 때문에 아날로그 값을 디지털 값으로 변환이 가능한 MCP-3008 칩을 사용해야합니다.



[그림] MCP3008 칩

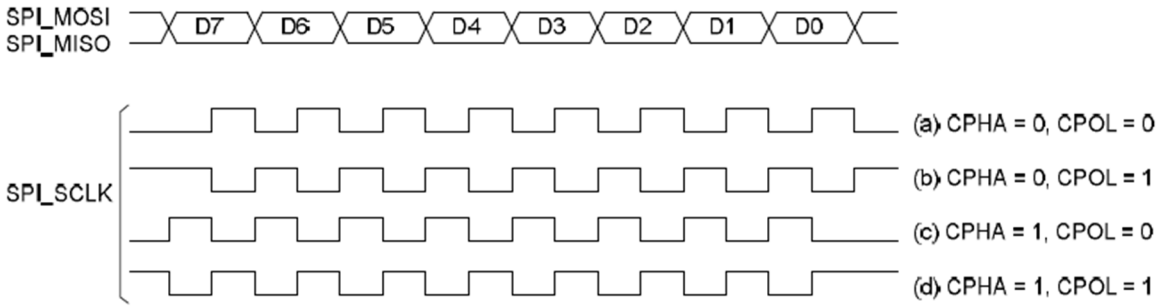
MCP3008칩은 SPI 통신을 이용하여 라즈베리 파이와 통신합니다. SPI 통신을 하기 위해서 CLK는 SPI0 SCLK 핀과, Dout은 SPI0 MISO(Master In/Slave out)핀과 연결되고, Din은 SPI0의 MOSI(Master out/Slave in)과 연결되어야 합니다. SPI에서 마스터가 라즈베리파이, 연결된 디바이스가 Slave라고 생각하면, 직관적으로 이해하기 쉽습니다. CS는 라즈베리파이의 SPI0 CS연결 하면 되는데, 저희 조는 CS0를 사용하기로 하였습니다.

SPI는 Serial peripheral interface의 약자이며 BCM 문서에는 3가지 버전의 SPI 모드들이 서술되어 있으나, MCP3008에서는 Standard Mode를 통해서 통신합니다.



[그림] SPI Standard Mode

Standard Mode에서 라즈베리파이가 Master로서 모든 통신을 제어하고, Master Out-Slave In을 통해서 드라이버에게 값을 전달하며, Master In-Slave out을 통해서 Slave로부터 데이터를 받아들입니다. CE핀을 통해서 작동하는 드라이버를 선택하고, SCLK를 통해서 전체 드라이버에 클락을 제공합니다.



[그림] SPI 클락의 종류

SPI의 클락은 여러가지를 사용할 수 있으며, CPHA를 통해서 클락을 살짝 밀어서 데이터의 클락 중앙에서 R/W를 발생시켜서 정확도를 높일 수 있고, CPOL을 이용해서 클락을 반전시킬 수 있습니다. MCP3008은 0,0과 1,1만을 지원합니다.

SPI를 다루는 알고리즘을 작성하는 방식은 크게 3가지로 분류할 수 있습니다. Interrupt를 발생시키는 방법과, DMA(직접 메모리 접근)을 사용하는 방식, Polling을 하는 방식이 있는데, 저희 조에서 디바이스 드라이버를 구현할 때 사용한 방식은 하드웨어의 변화를 지속적으로 읽어 들이는 Polling 방식입니다.

SPI Address Map			
Address Offset	Register Name	Description	Size
0x0	CS	SPI Master Control and Status	32
0x4	FIFO	SPI Master TX and RX FIFOs	32
0x8	CLK	SPI Master Clock Divider	32

[그림] SPI의 Address

Standard Mode의 SPI 방식과 Polling을 사용한 방식의 SPI 통신을 레지스터 조작만으로 구현하기 위해 사용하는 레지스터는 CS, FIFO, CLK, DLEN, LTOH, DC 총 6가지 중에 3가지입니다. CS 레지스터는 SPI 통신을 설정하면서 상태를 확인하기 위해서 사용되고, FIFO는 SPI의 데이터 전송에 사용됩니다. CLK는 클락을 조정할 때 사용합니다. SDLEN은 오직 DMA 방식에서만 사용하고, LTOH는 LoSSI 모드, DC Register는 Interrupt 방식에서만 사용하기에 저희는 다루지 않았습니다. SPI 통신을 하기 위해서는 SPI의 Base address를 잡아줘야 하며, 이는 0X3F203000입니다. 또한, GPIO핀 7번, 8번, 9번, 10번, 11번 핀을 BCM문서에 따라서 ALT0 모드(100)로 설정해야 합니다.

```
case IOCTL_CMD_WATER_SETTING:
        //gpfsel setting.
        printk("spics:%x\n", *spics);
        *gpsel0 &= ~(1<<21);
        *gpsel0 &= ~(1<<22);
        *gpsel0 |= (1<<23); //GPIO 7 = ALT0

        *gpsel0 &= ~(1<<24);
        *gpsel0 &= ~(1<<25);
        *gpsel0 |= (1<<26); //GPIO 8 = ALT0
......
```

[코드] GPIO 핀을 SPI 모드로 설정하는 코드 조각.

```
...
  char buf1 = 0x01;
  char buf2 = 0x80;
  *spics = 0x00B0; //Initialization
  *spiclk = 64;
...
```

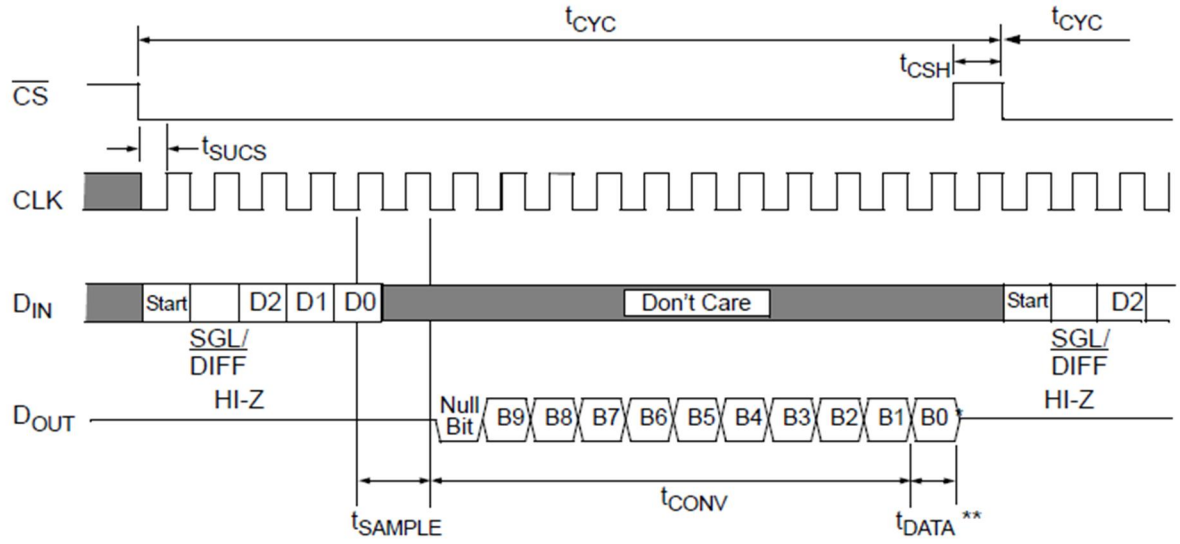
[그림] SPI 통신 시작의 초기 설정

처음 SPI 통신을 시작하면서 CS레지스터의 값을 0x00B0으로 조작합니다.

1	0	1	1	0	0	0	0
TA	CSPOL	CLEAR RX	CLEAR TX	CPOL	CPHA	CS	CS

[표] CS 레지스터의 00**B0**

신경 쓰지 않는 레지스터들을 제외하고 설명하자면, CS 레지스터의 0~1번 비트는 Chip Select(CS)에 해당합니다. 저희는 CS0을 사용하기에 0입니다. CPHA와 CPOL은 MCP3008이 0,0 모드를 지원하기에 0,0으로 설정하였습니다. LSB에서부터 두 번째 바이트인 B는 CLEAR 레지스터를 11로 바꿉니다. 이는 TX FIFO와 RX FIFO를 동시에 클리어 하는 것을 의미합니다. TA는 Transfer Active를 의미하며, SPI 통신을 활성화하여 시작합니다. 하위 비트들이 주로 설정과 관련된 반면, 상위 비트들은 보통 아래에 서술할 알고리즘에서 사용됩니다.



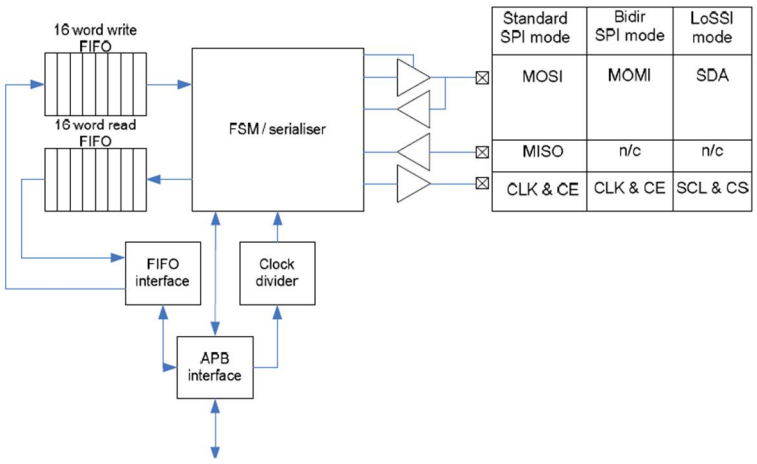
[그림] MCP3008의 통신 구조

위 코드를 참조하면 buf1 = 0x01, buf2 = 0x80이 들어있습니다. 이는 SPI 통신에서 Master가 Slave에게 보낼 비트를 미리 설정해 놓은 것입니다. D_in을 보면 Start 비트로 시작하는데 이것을 보내기 위해서 buf1 = 0001로 설정되어있습니다. 그 다음에 SGL, D2, D1, D0를 순서대로 읽는데 buf_2가 0x80이므로 SGL =1, D2, D1, D0는 0입니다.

Control Bit Selections				Input Configuration	Channel Selection
Single/ Diff	D2*	D1	D0		
1	X	0	0	single-ended	CH0
1	X	0	1	single-ended	CH1

[그림] MCP3008의 Control Bit

이는 MCP3008의 데이터 시트를 참조해보면 채널을 0을 사용하는 것을 의미합니다. 미세먼지 센서는 이와 다르게 buf2 = 0x90를 사용하고 있기에 채널 1핀을 사용하고 있음을 의미합니다. 처음에는 CS를 조작해서 Slave를 여러가지 두는 방식으로 구현하였으나, 이것이 MCP3008 칩을 두 개를 사용하여서 회로를 지나치게 복잡하게 만들어서 MCP3008 하나만을 슬레이브로 두고, 대신 채널을 여러 개 쓰는 방식으로 구현 방식을 변경하였습니다. clk 레지스터에 들어간 값은 메인 클락을 clk 레지스터의 값으로 나눠서 SPI 클락을 쓰게 됩니다.



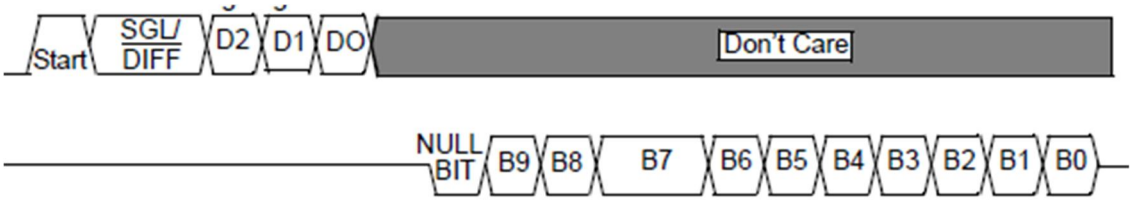
[그림] BCM 문서의 SPI 하드웨어 구조도

SPI로 값을 쓸 때 FIFO 레지스터는 32bit이지만, 실제 R/W 레지스터는 16비트입니다. 우리는 SPI의 FIFO 레지스터를 참조해서 Read FIFO의 값을 꺼내 올 수 있고, 쓰기를 해서 Write FIFO의 값을 꺼내올 수 있습니다. 따라서, SPI의 FIFO 레지스터의 값을 찍어도 이미 쓰여진 Write FIFO의 값은 읽을 수 없습니다. 이 때문에, 제대로 통신을 하고 디버깅을 하려면 이러한 FIFO의 상태를 SPI 통신을 할 때 주기적으로 체크하면서 통신을 해야 합니다.

```
while(1) if(((*spics) &(1<<18)) != 0) break; //tx buffer has space.
 *spififo = buf1; //send mcp3008 start bit(1/2)
 while(1) if(((*spics) & (1<<16)) != 0) break; //wait for done.
 //second cycle
 while(1) if(((*spics) &(1<<18)) != 0) break;
 *spififo = buf2; // send mcp3008 start bit(2/2)
 while(1) if(((*spics) & (1<<16)) != 0) break;
 //third cycle
  while(1) if(((*spics) &(1<<18)) != 0) break;
 *spififo = 0x00; // don't care value
 while(1)
 {
    if(((*spics)&(1<<17)) == 0) break; //check read fifo empty
    buf1 = *spififo; //read bit from mcp(1/2)
 }
 while(1) if(((*spics) & (1<<16)) != 0) break;
 //fourth cycle
 while(1) if(((*spics) &(1<<18)) != 0) break;
 *spififo = 0x00; // don't care value
 while(1)
 {
    if(((*spics)&(1<<17)) == 0) break; //check read fifo empty
    buf2 = *spififo; //read bit from mcp(2/2)
 }
 while(1) if(((*spics) & (1<<16)) != 0) break;
 buf1 = buf1 & 0x0F; //mask
 kbuf = (buf1 <<8) | buf2;
 printk("buf1:%u\n", kbuf);
 *spics = 0x0000; //reset
 return kbuf;
```

[코드] SPI 통신의 메인 알고리즘

위 코드는 SPI 통신의 메인 알고리즘입니다. cs 레지스터의 18번 비트는 TXD를 의미합니다. 이는 TX FIFO가 남아있는 공간이 있는지 없는지 알려줍니다. 즉, TX FIFO가 비는 순간 MCP3008이 요구하는 값을 TX FIFO에 값을 쓰기 시작합니다. 16번 비트는 Done을 의미합니다. while문이 걸려있기 때문에 FIFO에 쓴 값의 Transfer가 끝날 때 까지 기다립니다. 2 번째 사이클을 돌면서 MCP3008이 요구한 값을 전부 전송하면 MCP3008은 자신이 기록한 아날로그 값을 디지털로 변환해서 전송을 시작합니다.



[그림] MCP3008의 통신과정의 일부.

17번째 비트는 RXD입니다. RXD는 RX FIFO가 비어 있는지 검사합니다. 이를 이용해서 MCP3008이 보낸 값을 FIFO가 빌 때 까지 모조리 읽어 들입니다. 읽는 과정은 두 번에 걸쳐서 일어나는데 MCP3008의 데이터 시트를 참조해보면, 총 10개의 비트를 보내기 때문입니다. 앞의 두 비트를 buf1을 이용해서 읽고, 그 다음 여덟 비트를 buf2를 이용해서 읽어 온 뒤, int 자료형인 kbuf에 쉬프트 연산을 이용해서 합쳐서 반환합니다.

```
case IOCTL_CMD_WATER_GET:
        for(i = 0; i < 10; i++)
        {
           kbuf += get_value();
        }
        kbuf /= 10;
         copy_to_user((const void*)arg, &kbuf, 4);
     default:
        return 3;
  }
```

[코드] IOCTL_CMD_WATER 값의 반환

마지막으로 이렇게 받은 kbuf 값을 10번을 받아서 평균을 내서 반환합니다. 평균을 내서 반환하는 이유는, 센서의 값에 잡음이 끼는 것을 어느정도 막기 위해서입니다.

e) 먼지 드라이버

미세 먼지 센서인 GP2Y1010 역시, 먼지의 농도를 아날로그 값으로 출력합니다. 이 때문에 이 또한 MCP-3008을 이용해서 통신해야 합니다. 앞에서 설명한 SPI에 대한 내용은 굳이 중복으로 서술하지 않겠습니다.

```
int get_ad_value(void)
{

   int kbuf = 0;
  char buf1 = 0x01;
  char buf2 = 0x90;
   *spics = 0x00B0; //Initialization
        //first cycle
        while(1) if(((*spics) &(1<<18)) != 0) break; //tx buffer has space.
......
```

[코드] SPI 통신 시작

MCP3008의 0번 채널을 이미 물 수위 센서가 점유하고 있기 때문에, SPI 통신에서 마스터가 보낼 값인 0x80이 0x90으로 바뀌었습니다. D0가 HIGH가 되었기 때문에 1번 채널을 사용한다는 것을 의미합니다. 이 외에 SPI 통신에서 유의미한 변화는 없기 때문에 get_ad_value()라는 함수로 캡슐화하여 사용하고 있습니다.

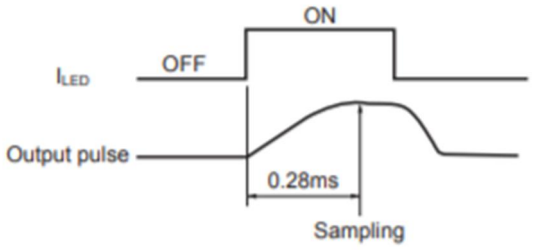
```
case IOCTL_CMD_DUST_GET:
        for(i=0; i<100; i++)
        {
           *gpclr1 |= (0<<18);
           udelay(280);

           buf = get_ad_value();

           udelay(30);
           *gpset1 |= (1<<18);
           mdelay(9);
           udelay(680);
           kbuf += buf;
        }

        kbuf /= 100;
        *spics = 0x0000; //reset
         copy_to_user((const void*)arg, &kbuf, 4);
......
```

[코드] 미세먼지 센서 메인 알고리즘

미세먼지 센서는 SPI 통신 이외에 GPIO 핀 하나를 Output 모드로 사용합니다. 사실 상 두 개의 디바이스를 사용하는 것이나 다름없습니다. 미세먼지를 검출하기 위한 빛을 쏘는 LED와, 먼지에 부딪혀 산

란된 빛을 검출하여 아날로그 값으로 돌려주는 센서가 있기 때문입니다. 이 때문에 검은색 전선을 집어넣고 테스트해보면 검출 값이 덜 나오고, 흰색 전선을 집어넣고 테스트하면 먼지 밀도가 최대로 차 있다고 감지하여서 검출 값이 높게 나오는 것을 볼 수 있습니다. 참고로 해당 센서의 LED는 반대로 0이 들어오면 켜지고, 1이 들어오면 꺼지는 특성을 지닙니다.

Parameter	Symbol	Value	Unit
Pulse Cycle	T	10 ± 1	ms
Pulse Width	P_W	0.32 ± 0.02	ms
Operating Supply voltage	V_{CC}	5 ± 0.5	V



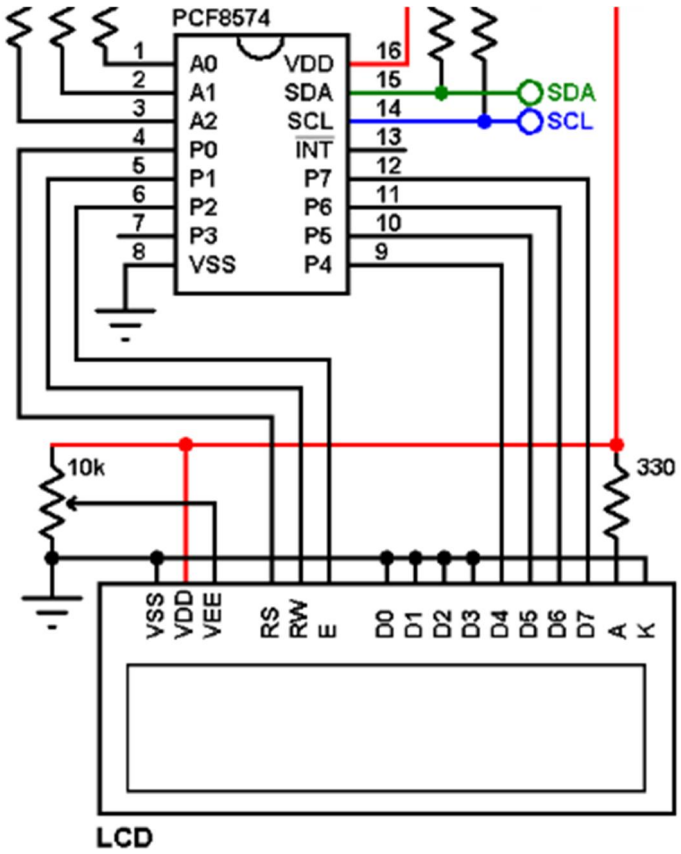
[그림] GP2Y1010의 작동 조건

GP2Y1010이 작동하기 위한 전체 사이클은 약 10ms입니다. 따라서 위 코드의 딜레이를 전부 합쳐보면 10ms에 근접해서 나옵니다. LED가 켜지는 순간 약 0.28ms 이후에 값을 검출해야 합니다. 이후 펄스의 폭을 맞추기 위해서 0.03을 더 쉬는데, 0.31, 0.32, 0.33를 테스트하면서 가장 먼지 검출이 안정적으로 나온 값을 선정하였습니다. 이는, 아마 SPI 통신 등의 코드 자체가 지니는 실행 시간이 영향을 조금이라도 주는 것이 아닐까 생각합니다. 이번 코드는 물 수위 센서가 10번 돌아가는 것과 다르게 100번을 돌아간 뒤 값을 평균 내는데, 이는 센서 자체의 검출 성능이 불안정하면서 좋지 않았기 때문입니다.

[사진] 센서 영점 조절

참고로 센서의 영점 조절을 실시하였으나, 해당 센서가 공기 중 미세먼지 실제 농도가 0일 때 약 0~3에 해당하는 값을 출력해서 영점 조절이 필요 없는 것으로 생각했습니다. 영점을 맞출 때는 고성능 미세먼지 검출기와 전자식 미세먼지 필터를 센서와 비닐 봉투에 넣고 밀봉해서 0으로 떨어지는 순간을 보고 맞추려고 하였습니다.

f) LCD 드라이버



[그림] LCD 연결도

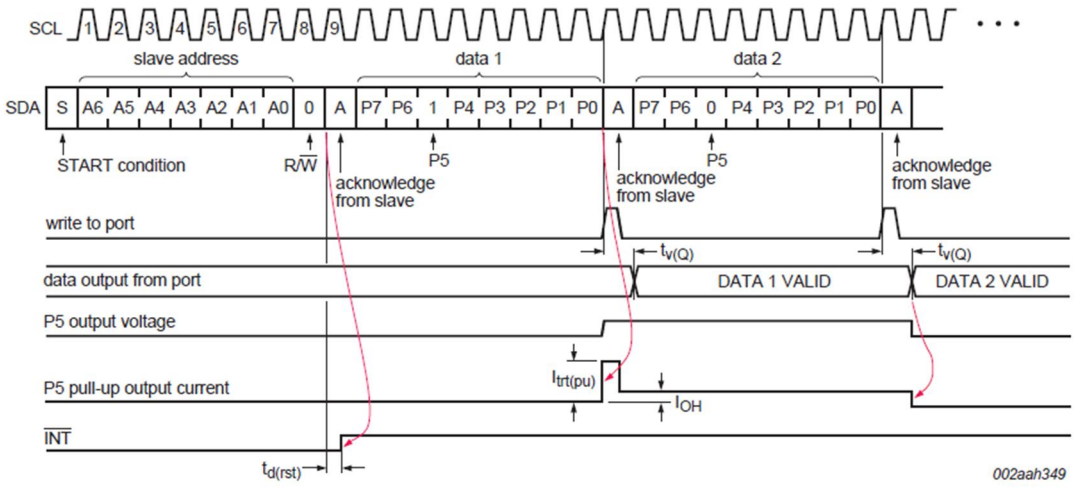
저희 조에서 사용한 LCD는 1602A라는 모델입니다. 이 LCD가 굉장히 많은 I/O 많은 핀을 사용하기 때문에, 추가적으로 PCF8574칩을 사용합니다. PCF8574칩은 8개의 데이터 출력 핀을 가지고, SDA, SCL 두개의 핀을 이용해서 라즈베리파이와 I2C 통신을 하게 됩니다.

I2C또한 SPI와 마찬가지로 라즈베리파이가 통신 전체를 통제하고, 데이터를 R/W하기 전에 주소를 항상 먼저 선언해야 합니다. 이를 이용해서 같은 bus에 여러가지 디바이스를 동시에 사용할 수 있다는 장점이 있습니다. 라즈베리파이에서는 'I2cdetect -y 1' 명령어를 통해서 슬레이브 (디바이스)들의 주소를 획득할 수 있습니다.

0x0	C	Control	32
0x4	S	Status	32
0x8	DLEN	Data Length	32
0xc	A	Slave Address	32
0x10	FIFO	Data FIFO	32
0x14	DIV	Clock Divider	32
0x18	DEL	Data Delay	32

[그림] I2C의 레지스터

저희가 I2C에서 사용한 알고리즘은 SPI와 동일하게 Polling 방식입니다. 사용한 레지스터는 C 레지스터, S 레지스터, DLEN, A 레지스터, FIFO입니다. DIV는 위 SPI 통신의 CLK와 거의 동작이 유사하고, 조작을 하지 않아도 통신이 정상적으로 작동하는 것을 확인했기 때문에 사용하지 않았습니다. C 레지스터는 I2C 통신을 설정하는데 사용됩니다. S 레지스터는 주로 상태를 확인하는데 사용되고, DLEN은 보내는 바이트의 수를 결정합니다. A 레지스터는 슬레이브의 주소 값을 설정하고, FIFO의 동작은 SPI와 거의 유사하게 데이터를 읽고 쓰는데 사용됩니다.



[그림] PCF8574의 통신 구조

PCF8574칩은 라즈베리파이와 I2C를 이용해서 통신을 하는데, 맨 처음 주소를 받아서 시작하며 이 때 한번에 전송되는 데이터는 1바이트입니다. PCF8574는 이 외에 I2C 통신에서 따로 무언가 더 요구하지 않고 바로 통신을 시작하는 것도 특징이라면 특징입니다.

```
case IOCTL_LCD_SETTING:
        *gpsel0 &= ~(1<<6);
        *gpsel0 &= ~(1<<7);
        *gpsel0 |= (1<<8); //ALT 0

        *gpsel0 &= ~(1<<9);
        *gpsel0 &= ~(1<<10);
        *gpsel0 |= (1<<11); //ALT 0

        clear_state();
        *i2cdlen = 1; //1-byte transfer
        *i2ca = 0x3f;//slave address
        *i2cc = 0x00; //clear c register
        *i2cc |= (1<<15); //i2cenable
        *i2cc |= (1<<4); //clear
        *i2cc |= (1<<5); //clear

        wait_done();
        print_all_state(0);
```

[코드] LCD 드라이버의 I2C설정 부분

I2C 통신을 사용하기 위해서는 SDL과 SCL 핀의 설정을 ALT 0로 해주어야 합니다. dlen 레지스터에 들어간 1은 통신의 단위가 1바이트임을 나타냅니다. i2cc 레지스터의 4번, 5번 비트는 FIFO를 시작하기 전에 CLEAR하는 동작을 하고, 15번은 I2C통신을 Enable하는 동작입니다. 0번 비트는 저희는 LCD에 쓰기 때문에 0(Write)으로 두는 것이 적절합니다. 이 모든 작업 앞에서 i2ca에 위에서 언급한 리눅스 명령어로 찍어본 Slave Address를 대입해야 합니다. 이 때, Slave Address가 코드를 돌렸을 때 가끔 사라지기도 하는데, 이는 잘못된 명령으로 인해 I2C 통신의 버스가 점유되어서 그렇다고 합니다. 라즈베리파이를 재 부팅하면 정상으로 돌아오게 됩니다.

DLEN	Data Length. Writing to DLEN specifies the number of bytes to be transmitted/received. Reading from DLEN when TA = 1 or DONE = 1, returns the number of bytes still to be transmitted or received. Reading from DLEN when TA = 0 and DONE = 0, returns the last DLEN value written. DLEN can be left over multiple packets.	RW	0x0

[그림] BCM 문서의 DLEN 설명

```
void print_all_state(int num)
{
  printk("%d: i2cc: %x, i2cs: %x, dlen: %x",num, *i2cc, *i2cs, *i2cdlen);
}
```

[코드] print_all_state

디버그를 하면서 에러가 날 때 가끔 i2cs 레지스터를 printk로 값을 살펴본다면, 8번의 ERR ACK가 1로 설정된 것을 볼 수 있습니다. ERR ACK는 슬레이브가 ACK를 보내지 못할 때 SET되는 비트입니다. ERR ACK이 1로 설정된 시점에서 알고리즘이 잘못 됐다는 것을 알 수 있기 때문에 유용하게 사용될 수 있는 비트입니다. pirnt_all_state()함수는 이를 위해 사용한 함수이며 디버깅에 필요한 3가지 레지스터를 출력하는 함수입니다. dlen 레지스터는 특정 조건 하에서 여전히 전송되거나 받아야 할 바이트의 수를 표시해주기 때문에 이 또한 디버그 용으로 사용할 수 있습니다.

clear_state()함수는 i2cs의 ERR, CLKT, DONE 비트를 초기화하는데 사용합니다. SPI 통신에서는 이들과 비슷한 레지스터들이 Read-Only로서 자동으로 초기화된다고 명시 되어있었는데, I2C에서는 그러한 명시가 없어서 초기화를 수동으로 시켜주었습니다.

```
void wait_done(void)
```

```
{
  int i = 0;
  while(1)
  {
     if((((*i2cs)&(1<<1))!= 0) && (((*i2cs)&(1<<0)) == 0)) break;
     if(i==1000)
     {
        printk("error done register!\n");
        break;
     }
     i++;
  }
}
```

[코드] wait_done 함수

wait_done 함수는 앞서 설명한 SPI 통신에서의 done 레지스터를 기다리는 것과 비슷한 동작을 합니다. Done은 Transfer가 끝났는지 여부를 알려주며, i2cs의 0번 비트는 TA 레지스터를 의미하는데, 이 TA레지스터는 Transfer가 Active되어있는지 알려줍니다. 디바이스 드라이버의 함수가 돌아가는 과정에서는 앱이 강제 종료가 되지 않는데, 이 때문에 타임 아웃을 1000으로 두고, 타임 아웃이 발생할 때 printk를 이용해서 타임 아웃이 발생했음을 알리는 문구를 추가하였습니다.

```
void write_i2c(char value)
{
  int i = 0;
  *i2cc |= (1<<7); //start new transfer
  while(1)
  {
     if(((*i2cs)&(1<<4)) != 0) break;
     if(i>1000)
     {
        printk(KERN_ALERT "Time out check TXD!\n");
        break;
     }
     i++;
  }

  *i2cfifo = value;

  wait_done();
  print_all_state(111);
  clear_state();
}
```

[코드] I2C의 쓰기 함수

i2cc의 7번 비트는 Start Transfer이며, 1로 설정하면, One Shot Operation이기 때문에 자동으로 0으로 돌아옵니다. 이는 쓰기를 새로 할 때마다 항상 1로 설정해주어야 합니다.

i2cs의 4번 비트는 TXD로 FIFO에 쓸 수 있는 공간이 있는지 체크하는 함수로, 기능은 SPI와 동일합니다. 쓸 수 있는 공간이 있자 마자 i2cfifo에 값이 쓰여지고, wait_done()함수를 실행하여 Transfer가 완전히 끝날 때 까지 기다리게 됩니다.

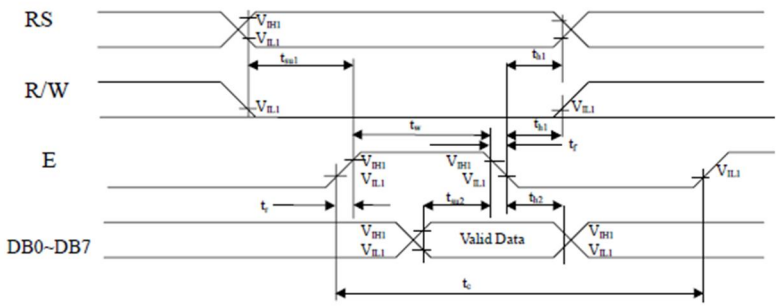
```
void lcd_write(char input)
{
  //backlight;
  char buffer;
  buffer = input | 0x04 | 0x08 ;
  printk("buffer: %x\n", buffer);
  write_i2c(buffer); //enable pulse
  mdelay(1);
  buffer = input & (~0x04) | 0x08 ;
  write_i2c(buffer); //disable pulse
  printk("buffer: %x\n", buffer);
  mdelay(60);

}
```

[코드] lcd_write 함수

P7~P4	P3	P2	P1	P0
D7~D4	BackLight	Enable	RW	RS

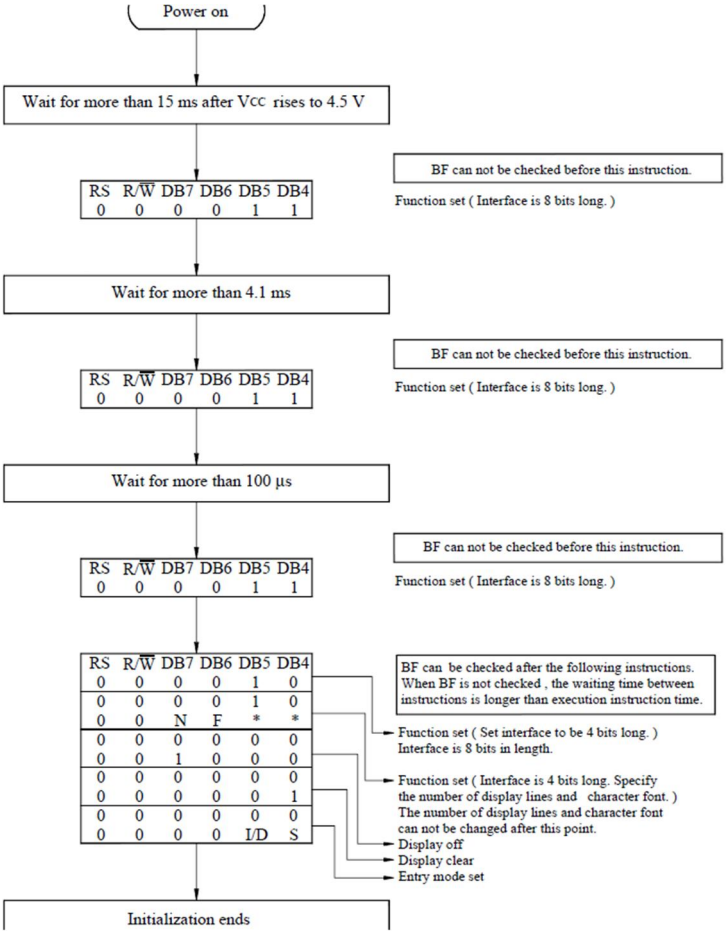
[표] PCF8574와 1602A LCD의 회로 매핑

이렇게 I2C 통신을 이용해서 값을 쓰는데 성공하면, P0~P7에 값이 써집니다. P3이 LCD의 백 라이트를 나타내는 비트이기에, 이를 껐다 키는 것으로 I2C 알고리즘이 잘 동작하는 것을 파악할 수 있습니다. 저희가 쓸 수 있는 데이터 핀은 D7~D4비트이기 때문에 LCD를 4비트 모드로 설정하고 캐릭터를 4비트씩 끊어서 I2C로 보내주어야 합니다. LCD의 4비트 모드를 사용하면 자동으로 D7~D4에 들어온 값 2번을 조합해서 8비트를 쓰듯이 사용할 수 있게 해줍니다. lcd_write 함수는 캐릭터를 앞자리만 이용하며, BackLight On과 OR연산 하여 보내주는 것을 볼 수 있습니다.



[1602A]의 쓰기 통신 구조

이 때, 최소한의 enable 펄스를 인가해주어야 되는데, mdelay와 0x04를 데이터에 논리 연산을 해서 이를 구현하였습니다.



[그림] 1602A의 시작 알고리즘

```
case IOCTL_LCD_START_PRINT:

        lcd_write(0x30);
        mdelay(5);
        lcd_write(0x30);
        mdelay(1);
        lcd_write(0x30);
        mdelay(1);
        lcd_write(0x20); //begin lcd(don't remove!!!)


        lcd_write(0x20);
        lcd_write(0x80); //NF
        lcd_write(0x00);
        lcd_write(0xC0);
        lcd_write(0x00);
        lcd_write(0x60);
        lcd_write(0x00);
        lcd_write(0x10); //Display Clear
        //initialization finish
......
```

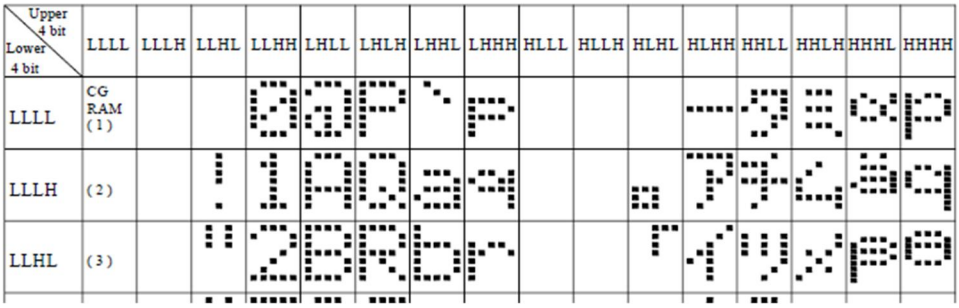
[코드] LCD 초기화 코드

1602A는 초기 설정이 존재하기 때문에, 이 초기 설정을 데이터 시트를 읽고 해줘야만 동작을 시작합

니다. 위 코드는 4비트 모드로 동작하기 위한 코드입니다. 초기 설정을 할 때, 위 설정의 시작 알고리즘을 코드로 구현하고 LCD를 두 줄 모드로 설정하였습니다.

Instruction	Instruction Code										Description	Execution time (fosc=270Khz)
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0		
											× 8 dots)	
Read Busy Flag and Address	0	1	BF	AC6	AC5	AC4	AC3	AC2	AC1	AC0	Whether during internal operation or not can be known by reading BF. The contents of address counter can also be read.	0 μs
Write Data to RAM	1	0	D7	D6	D5	D4	D3	D2	D1	D0	Write data into internal RAM (DDRAM/CGRAM).	43 μs
Read Data from RAM	1	1	D7	D6	D5	D4	D3	D2	D1	D0	Read data from internal RAM (DDRAM/CGRAM).	43 μs

[그림] LCD Instruction Table



[그림] LCD의 캐릭터 테이블

```
void write_letter(char input)
{
  char buffer[2];
  char cbuf;

  if(('0' <= input) && ( '9' >= input))
  {
     cbuf = input - '0';
     cbuf = (cbuf << 4);
     printk("cbuf:%x\n", cbuf);
     buffer[0] = 0x30 | 0x01;
     buffer[1] = cbuf | 0x01;
     lcd_write(buffer[0]);
     msleep(1);
     lcd_write(buffer[1]);
  }
  else if(('A' <= input) && ( 'O' >= input))
```

```
  {
......
```

[코드] write_letter 함수

LCD에 데이터를 쓰는 함수는 input을 받아서 가공하여서 위에서 명령어를 보내는데 썼던 lcd_write() 함수로 보냅니다. 저희 조는 0~9, A~Z만을 이용하여서 해당 데이터가 아니면 공백을 출력하도록 되어 있습니다. LCD는 커서를 한 글자를 쓸 때마다 자동으로 하나씩 넘기기 때문에 따로 LCD의 주소 조작을 할 필요가 없습니다. 4bit 모드이기 때문에 lcd_write()를 두 번 해주면 자동으로 8bit로 합쳐줍니다.

위 Instruction Table을 보면 RS가 1일 때 데이터를 쓰기 때문에, 받은 아스키 코드에 OR 연산으로 0x01을 시켜주는 것을 볼 수 있습니다. I2C 통신에서의 0번째 비트가 RS와 매핑이 되기 때문입니다. 아스키 코드 '1'을 예로 들자면 캐릭터 테이블을 참조해서 상위 비트인 0x30(0011)먼저 보내주고, 하위 비트는 아스키 코드 값 '0'을 빼서 0x01(0001)을 만들어주어서 생성합니다. 여기까지 한다면, 라즈베리파이에서 I2C 통신을 사용해서 PCF 칩에 데이터를 전달하였고, PCF 칩에서 데이터를 가공한 다음 LCD에 전송해주어서, LCD에 글자가 출력됩니다.

1) 응용 프로그램

저희 프로그램의 응용 시스템은 '감지'를 담당하는 app과 '동작'을 담당하는 app로 나뉘어져 있습니다. 먼저 '감지' 앱은 물수위 센서, 온습도 센서, 미세먼지 센서, LCD, 버튼과 연결되어 있고 각각의 드라이버 외에 감지 파이에서 동작 파이로 창문을 여는 명령을 전달하는 send드라이버를 사용했습니다.

i) 감지파이

```
int water =0, dust = 0;
int comp_hu = 0;
int comp_temp = 0;
int cur_hu;
int cur_temp;
int tmp_i[4];
int button_state = 1;
int is_window_open = 0;
```

각 센서들이 보내주는 데이터 값을 전역으로 설정하고 is_window_open을 통해 창문을 열지 닫을지를 결정하게 됩니다.

```
  ioctl(fd_w,WATER_SET,&d);
  ioctl(fd_d,DUST_SET,&d);
  x = ioctl(fd_l,IOCTL_LCD_SETTING, &p);
```

우선 물 수위 센서, 미세먼지 센서, LCD의 드라이버는 초기 설정이 필요하므로 main에서 루프를 돌기 전에 선언합니다.

```
void *detecter(void *data)
{
  ......
  while(1)
  {
     ioctl(fd_w, WATER_GET, &water);
     ioctl(fd_d, DUST_GET, &dust);
     sleep(1);



     ioctl(fd_t, TEMP_SET, &q);
     tmp1[0] = ((0xff00&q)>>8);
     tmp1[1] = (0xff&q);
     tmp_i[0] = (tmp1[0] - (tmp1[0]%10))/10; //huminity 10
     tmp_i[1] = tmp1[0]%10;  //huminity 1
     tmp_i[2] = (tmp1[1] - (tmp1[1]%10))/10;    //T 10
     tmp_i[3] = tmp1[1]%10;


     q=0;
     ioctl(fd_t1, TEMP_SET1, &q);
     tmp2[0] = ((0xff00&q)>>8);
```

```
      tmp2[1] = (0xff&q);

      cur_hu = tmp1[0];
      cur_temp = tmp1[1];
      comp_hu = tmp1[0]-tmp2[0];
      comp_temp = tmp1[1]-tmp2[1];
      sleep(1);
   }

}
```

물 수위 센서와 미세먼지 센서는 디바이스 드라이버의 ioctl get을 할 때 int값으로 유저에게 넘기도록 처리했으므로 각각 water와 dust에 값이 담아지게 됩니다. 다만 밑에 온습도의 경우 디바이스 드라이버에서 온도와 습도 데이터를 한번에 보내고 LCD에 출력해야 하므로 ioctl로 받은 후 tmp 배열로 온도와 습도를 나눠줍니다. 동작파이에 명령을 보낼 때 사용된 계산 값으로 tmp1에 내부 온습도의 데이터가, tmp2에 내부 온습도의 데이터가 int값으로 들어갑니다. 내부 온습도인 tmp1의 경우 LCD에 출력해야 하므로 tmp_i배열에 자리 수마다 한자리 정수로 바꿔서 담아줍니다. 결과적으로 cur_hu에 내부 습도, cur_temp에 내부온도, comp_hu에 비교습도, comp_temp에 비교온도가 들어가게 됩니다.

```
void fill_lcd_buffer()
{
  char input;
  char start = 0;
  char lcd_buf[51];
  int i = 0;
        for(i=0;i<51;i++)
        {
           if(i==0) lcd_buf[i] = 'W';
           else if(i==2)
           {
              if(water < WATER_THRESHOLD) lcd_buf[i] = 'O'; //Water value
              else lcd_buf[i] = 'X';
           }
           else if(i==6) lcd_buf[i] = 'D';
           else if(i==8)
           {
              if(dust < DUST_THRESHOLD) lcd_buf[i] = 'O'; //Dust value
              else lcd_buf[i] = 'X';
           }
           else if(i == 10)
           {
              lcd_buf[i] = 'O';
           }
           else if(i == 11)
           {
               if(button_state == 0) lcd_buf[i] = 'F';
               else lcd_buf[i] = 'N';
           }
           else if(i == 12)
           {
               if(button_state == 0) lcd_buf[i] = 'F';
               else lcd_buf[i] = '@';
           }
           else if(i==40) lcd_buf[i] = 'H';
           else if(i==42) lcd_buf[i] = (char)tmp_i[0]+0x30; //10 Hum value inside
           else if(i==43) lcd_buf[i] = (char)tmp_i[1]+0x30; //1 Hum value inside
```

```
            else if(i==47) lcd_buf[i] = 'T';
            else if(i==49) lcd_buf[i] = (char)tmp_i[2]+0x30; //10 Temp value inside
            else if(i==50) lcd_buf[i] = (char)tmp_i[3]+0x30;//1 Temp value inside
            else lcd_buf[i] = '@';
         }
         i=0;
         ioctl(fd_l, IOCTL_LCD_START_PRINT,&start);

         for(i=0; i<51;i++)
         {
            input = lcd_buf[i];
            ioctl(fd_l, IOCTL_LCD_PRINT_CHAR ,&input);
         }
}
```

LCD 디바이스는 두 줄로 출력이 되는데 첫 줄에 물감지와 먼지센서의 상태를 출력합니다. 저희가 정한 임계 값을 넘어서 창문이 닫혀야 하는 경우 X를 표기하고 열리는 상태는 O로 표기합니다. 온도와 습도는 detecter()에서 받은 tmp_i를 이용해 하나의 char형으로 변환 후 lcd_buf의 지정위치에 담깁니다. lcd_buf에 데이터를 채우는 for문 이후 IOCTL_LCD_START_PRINT로 초기화를 하고 한 글자씩 lcd_buf의 값을 IOCTL_LCD_PRINT_CHAR를 통해 출력합니다. 중간에 lcd_buf[i] = '@'는 LCD 드라이버의 파싱 때 공백으로 처리됩니다.

```
void *detect_button(void *data)
{
  int p = 0;
  int u =32;
  u = ioctl(fd_b, IOCTL_CMD_IS_BUTTON_ON, &p);
  while(1)
  {
     sleep(1);
     u = ioctl(fd_b, IOCTL_CMD_IS_BUTTON_ON, &p);
     if(p==1)
     {
        if(button_state == 0) button_state = 1;
        else button_state = 0;
        printf("button toggle: %d\n", button_state);
     }
  }

}
```

실습시간때 배웠던 버튼 드라이버를 살짝 변형하여 버튼 상태가 0이면 동작을 멈추고 1이면 동작을 지속하게 구현했습니다. 여기서 ON/OFF는 LCD화면에도 출력이 됩니다.

```
  while(1)
  {
     sleep(3);
     printf("water: %d Dust: %d Humi: %d Temp: %d btn State : %d\n", water, dust,
comp_hu, comp_temp, button_state);

    if(button_state == 1)
    {

        if(water > WATER_THRESHOLD)
       {
          is_window_open = 0;
```

```
        }
        else if(dust > DUST_THRESHOLD)
        {
           is_window_open = 0;
        }
        else if(comp_temp > TEMP_THRESHOLD)
        {
           is_window_open = 1;
        }
        else if(comp_hu > MOIST_THRESHOLD)
        {
           is_window_open = 1;
        }

        else is_window_open = 1;
     }
     else is_window_open = 0;

     if(is_window_open == 0) printf("Window Close!\n");
     else printf("Window Open\n");
     x = ioctl(fd_s,IOCTL_SEND,&is_window_open);
     //printf("send: %d\n", x);
     fill_lcd_buffer();
  }
```

메인에서는 이제 최종적으로 받은 데이터를 가지고 is_window_open값을 결정하게 됩니다. 물과 미세먼지의 우선순위를 두어 두 값이 하나라도 임계점을 넘으면 창문은 닫힙니다. 그리고 비교온도 즉, 내부가 너무 덥거나 습하면 창문이 열리게 됩니다. 조건문을 지나 is_window_open값이 정해지면 send 드라이버를 통해 동작파이에 값을 전송하게 됩니다. 그 후에 데이터를 출력하는 fill_lcd_buffer()로 넘어갑니다.

i) 동작파이

동작의 앱 단에서는 모터를 돌리는 하나의 디바이스 드라이버를 사용하게 됩니다. 서보모터의 디바이스 드라이버에서 IOCTL_FLAG_CHECK을 통해 창문을 열어야 하는지 닫아야 하는지에 대한 flag를 감지파이에서 받아옵니다. 그후 flag값을 통해 서보모터 디바이스 드라이버의 어떤 ioctl을 사용해야 하는지 조건문을 통해 해결했습니다.

```
int main(void)
{
  .....

  int flag=0;
  while(1)
  {
     flag=ioctl(fd,IOCTL_FLAG_CHECK,&p);
     printf("d %d",d);
     if(flag==10)
     {
        x=ioctl(fd,IOCTL_CMD_SET_DIRECTION,&
        sleep(5);
```

```
      }
      else
      {
         x=ioctl(fd,IOCTL_CMD_CLEAR_DIRECTION,&p);
         sleep(5);
      }
  }
.....
}
```

디바이스 드라이버에서 ioctl에 리턴 값을 정해주어 10이면 90도(창문이 열림) 다른 값이면 -90도(창문이 닫힘)로 설정하였습니다. 너무 빠른 ioctl의 사용을 막기위해 중간중간에 5초간의 텀을 두고 구현했습니다.

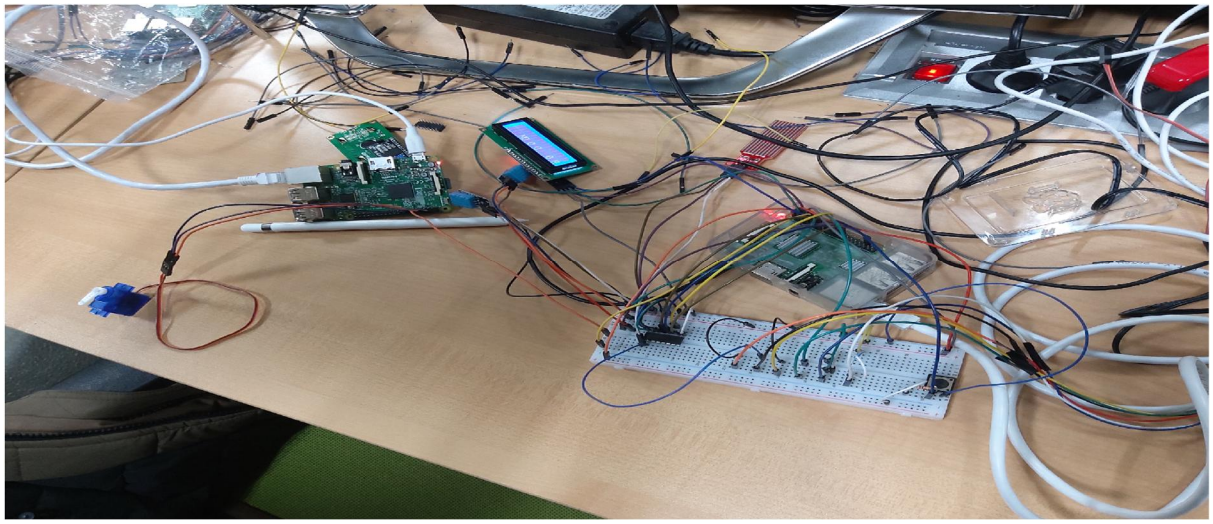
5. 시험 평가

1)프로그램 작동법

메인 파이(센서 처리 부분) 동작 파이(모터 처리 부분)이 동시에 켜져 있어야 합니다. 리눅스의 pthread를 사용하기 때문에 컴파일 옵션 뒤에 -lpthread를 붙여서 작동시켜야 합니다. make를 하고, 각 드라이버(파일명 뒤에 dev.c가 붙는 파일들)를 sudo insmod로 넣어주고, 앱을 실행시키면 됩니다.

커널은 4.19.75-v7+지만, 리눅스 환경에 따라서 Makefile의 해당 부분을 수정해야 할 수도 있습니다.

2)테스트 결과



[사진] 시스템이 정상동작 하는 모습.

모든 센서가 성공적으로 동작하였습니다.

3) 미구현 사항 및 버그 사항

i) 내부 외부 미세먼지 비교

저희가 처음 계획했던 제안서에서는 온습도뿐만 아니라 미세먼지 역시 내부와 외부를 비교해주어 창문이 동작하게 설계하도록 되어있습니다. 그러나 센서를 수령할 때 미세먼지 센서가 하나밖에 오지 않아 부득이하게 시나리오를 수정하고 하나만을 사용했습니다.

ii) 정보 표시 led

정보를 표시하기 위한 수단으로 저희는 처음에 여러 색의 led를 사용하거나 챌린지 이슈로 도트 매트릭스를 사용해보려 했습니다. 그러다가 센서 수령 시 lcd가 같이 포함되어 있었고 도트 매트릭스나 led보다는 lcd가 사용자가 정보를 확인하는데 있어서 직관적이라고 생각해 lcd로 바꾸었습니다. lcd는 i2c

통신에 pcf칩에 lcd디바이스 자체의 데이터시트까지 읽어야 하는 난제였습니다. 처음에는 어려워서 힘들었으나 구현하고 보니 led나 도트 매트릭스의 구조체 배열을 짜는 것보다 시스템 프로그래밍이라는 과목의 취지에 훨씬 잘 맞고 난이도도 훨씬 높았다는 생각이 들었습니다.

6. Challenge Issue 및 소감

구현에 있어서 가장 시간이 많이 걸렸던 부분은 SPI 통신입니다. 제대로 구현하는데 10일의 시간이 걸렸고 수업 시간에서는 대략적인 개요 외의 내용을 다루지 않았기 때문에, 직접 SPI의 구조를 조사해야 했습니다. 가장 처음에 GPIO핀을 비롯한 관련 설정들을 Enable하고 딱 하나의 데이터를 내보내는 것을 목표로 했습니다. 이 과정에서 BCM 문서를 정독 하였고, 다른 라즈베리파이 하나를 GPLEV 값을 측정하는 것을 이용해서 검출기로 사용하려고 했습니다. 하지만 디버깅이 제대로 되지 않았는데, 이는 GPLEV 값 측정을 통한 디버깅은 밀리 초 단위로 이루어지는데 SPI 통신의 동작속도는 그보다 훨씬 빨랐기 때문입니다. 따라서 FIFO를 통해서 값을 보내고 나가는 것을 확인해야 하는데, FIFO에 한 번 들어간 값을 다시 읽을 수 없었습니다. FIFO 레지스터를 참조하면 단지, RX FIFO의 값을 읽어올 뿐입니다.

이러한 구현의 어려움에 의해서 최홍범 조교님께 자문을 구했고 팀원과 역할을 나눠서 한 쪽은 소프트웨어적으로 SPI 통신을 구현하기 시작하였고, 다른 한쪽은 WiringPi의 라이브러리를 분석하기 시작했습니다. 하지만 WiringPi 안에서 linux/spidev.c라는 라이브러리가 존재했고, 해당 라이브러리가 워낙 캡슐화가 잘되어 있어서 분석에 난항을 겪었습니다. 결국 라이브러리를 분석하는 것은 포기하고, 대신 CS레지스터를 통해서 FIFO의 상태를 검사하고, 에러 플래그를 체크하는 방식을 통해서 디버깅을 시작했고, MCP3008에게 데이터값을 반환 받는데 성공하였습니다. 하지만 여전히 문제가 존재했는데, prink()를 넣어서 디버그할때는 정상적으로 돌아가다가 prink를 해제하면 동작을 제대로 안 하는 문제가 발생하였습니다. 이는 딜레이 역할을 해주던 printk가 빠지면서 FIFO 값이 미쳐 빠져나가지 못한 새에 새 값이 들어와 충돌을 일으키는 것으로 추정됐습니다. 이 문제를 해결하기 위해서 FIFO의 동작을 기다려야 한다는 점을 알게 되었고, 결국 DONE, RXD, TXD등을 검사하는 방식을 도입해서 완전한 SPI 통신의 구현에 성공하였습니다.

프로젝트 제안서에는 포함되지 않았던 드라이버를 구현해 본 것도 새로운 도전이었습니다. 원래 LCD 디바이스는 우리 조가 신청하지 않았던 디바이스였는데, 수령할 때 미세먼지와 함께 들어있었습니다. 팀원들끼리 토의한 결과, 도전해보자는 결론이 나왔습니다. LCD 디바이스는 PCF857 칩이 장착되어 있어서 라즈베리 파이와 I2C 통신을 이용해야 했습니다. SPI와 같이 수업 시간에서 대략적인 개요만을 배운 I2C여서 데이터 시트를 보면서 저희 들끼리 레지스터 조작을 깨우쳐야 했습니다. SPI를 구현하면서 데이터시트를 계속 읽었으므로 처음 SPI를 구현 할 때 보다 데이터시트의 레지스터 정보를 읽는 것이 더 익숙해져서 금방 구현 할 수 있을 줄 알았지만, 예상 외로 난관에 봉착했습니다. ADC를 구현할 때는 라즈베리파이의 SPI 데이터시트와 MCP3008의 데이터시트 두 가지를 이해해야 했지만, LCD를 구현하기 위해서는 라즈베리파이 <-> I2C <-> PCF칩 <-> LCD의 구조로 되어있는 3중 통신을 하였기 때문입니다. 저 3가지를 매핑하면서 정확하게 여러 번 데이터의 변환과 쓰기를 진행하여 디버깅하는 것이 상당한 난이도를 지녔습니다. 이 때문에 I2C 통신을 처음 진행할 때는, I2C의 3번째 비트가 LCD의 백 라이트를 ON/OFF할 수 있다는 점을 깨닫아서 LCD의 백 라이트를 끄고 켜는 것부터 시작하였습니다. I2C 또한 LCD의 백 라이트를 키는 것을 에러 플래그를 체크해가면서 디버그하여 성공하였으나, LCD와 PCF칩의 I2C가 어떻게 매핑 되어있는지 제대로 나와있는 공식 문서가 없었고, 결국 구글

에서 매핑 관련 회로도를 하나씩 시도해보면서 어떤 게 올바른 회로도인지 검증해야 했습니다. 이후 LCD의 초기화 조건과 통신 구조도를 해석해서 성공하였고, 3중으로 된 통신/변환 함수를 쓰면서 지금까지 다뤄본 가장 복잡한 장치의 구동에 성공하였습니다.

이번 시스템 프로그래밍 프로젝트를 통해서 임베디드 시스템을 정확히 동작하게 하려면 각각의 디바이스들의 데이터시트를 꼼꼼히 읽고 그 원리를 확실하게 이해해야 한다는 것을 특히 느꼈습니다 항상 코딩할 때 라이브러리의 사용을 당연히 여겼던 우리였는데 이번 프로젝트를 통해 직접 로우레벨의 코딩을 경험할 수 있었던 특별한 시간이었습니다.