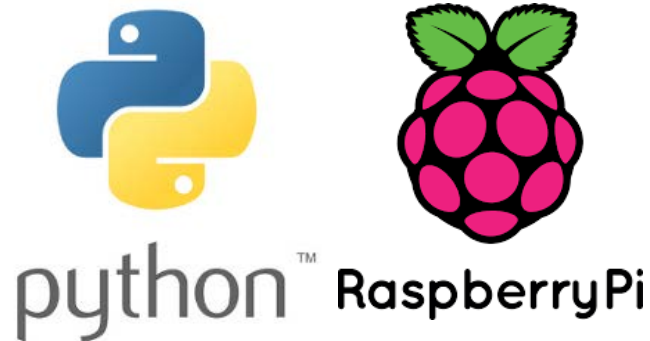


Python 기반 센서 제어



Sensor Programming

센서 프로그래밍

라즈베리파이 개발환경

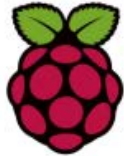


RaspberryPi



RASPBIAN

라즈베리파이 개발환경



Raspberry Pi 3 Model B



RASPBERRYPI3-MODB-1GB

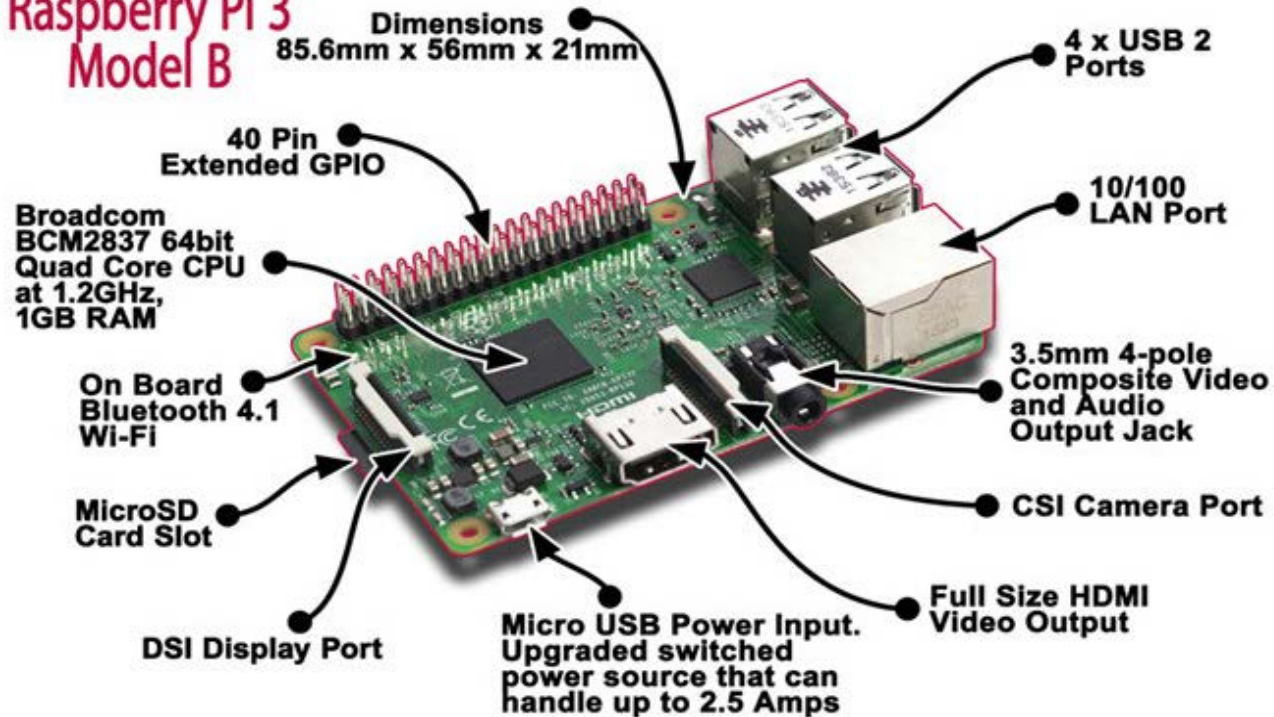


RPI3-MODB-16GB-NOOBS

- 2012 영국 라즈베리파이 재단에서 출시
- <https://www.raspberrypi.org/>

라즈베리파이 개발환경

Raspberry Pi 3 Model B



라즈베리파이 개발환경

icore-peri0 모듈

GPIO

I2C

SPI

Character LCD

FND

LED

초음파

EEPROM

IR 수신

DC 모터

GPIO 커넥터

가변저항

조이스틱

사운드

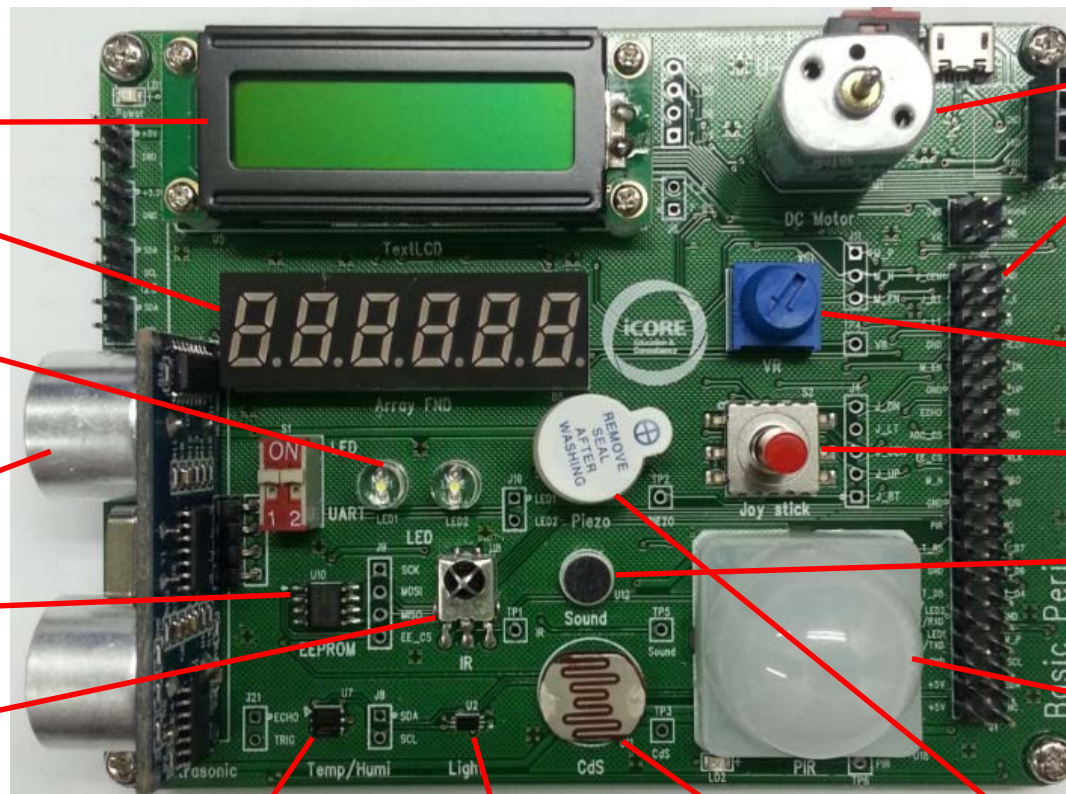
인체감지

온습도

빛 감지

조도센서

Piezo



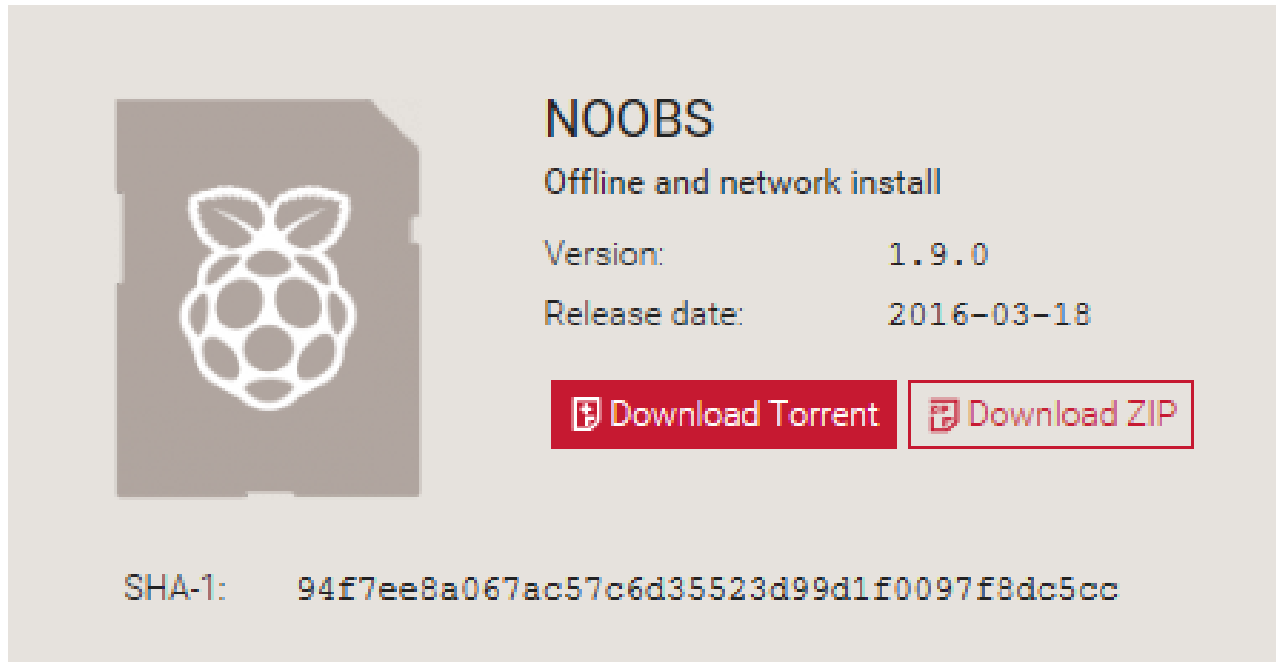
GPIO 확장 커넥터

라즈베리파이 개발환경

NOOBS로 OS 설치하기

<https://www.raspberrypi.org/downloads/noobs/>

"Download ZIP" 을 눌러 NOOBS_v1_9_0.zip 파일을 다운



The image is a screenshot of the NOOBS (New Out Of The Box Secure) download page. On the left, there is a large icon of a Raspberry Pi SD card with the Raspberry Pi logo on it. To the right of the icon, the text 'NOOBS' is displayed in a large, bold font. Below this, it says 'Offline and network install'. Further down, the 'Version:' is listed as '1.9.0' and the 'Release date:' is '2016-03-18'. At the bottom of this section, there are two red buttons: 'Download Torrent' and 'Download ZIP'. Below the buttons, the 'SHA-1:' hash is provided as '94f7ee8a067ac57c6d35523d99d1f0097f8dc5cc'.

NOOBS
Offline and network install

Version: 1.9.0
Release date: 2016-03-18

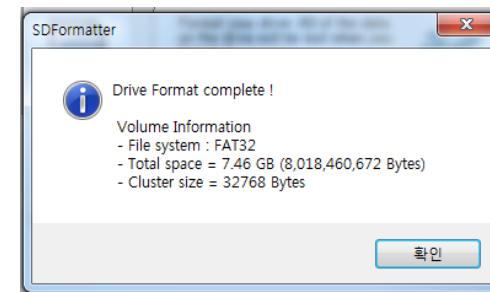
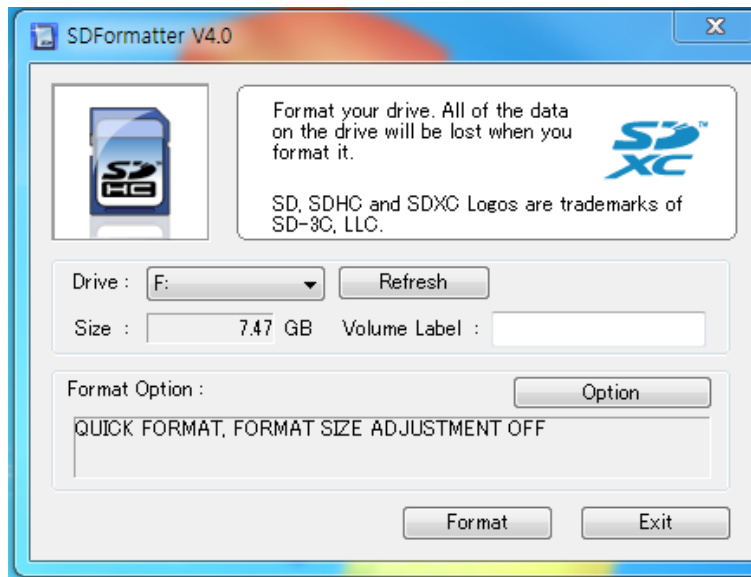
[Download Torrent](#) [Download ZIP](#)

SHA-1: 94f7ee8a067ac57c6d35523d99d1f0097f8dc5cc

라즈베리파이 개발환경

NOOBS로 OS 설치하기

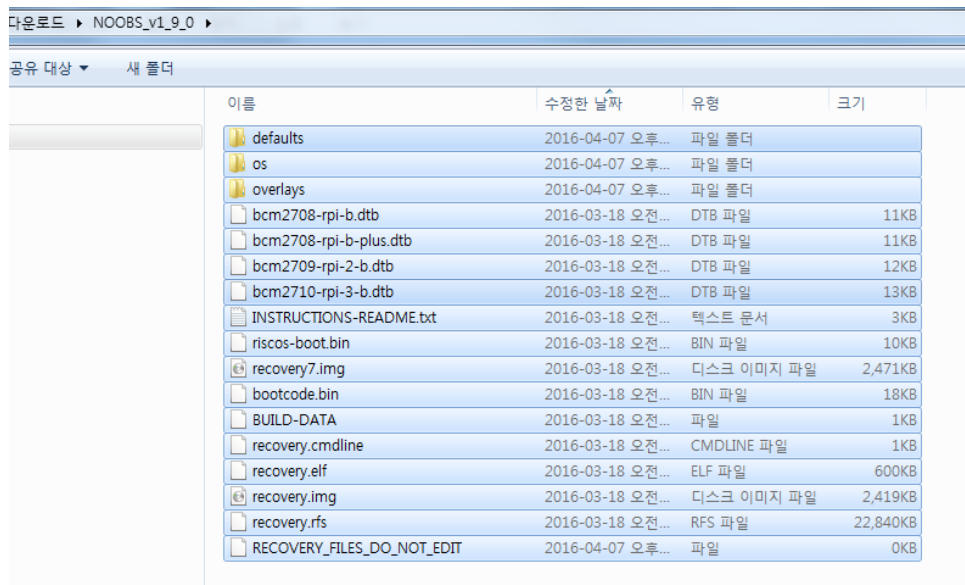
SDFormatterV4.0.exe 프로그램을 설치하고 USB타입 SD리더기에 Micro SD 카드를 장착하고 PC에 연결하여 아래와 같이 포맷한다



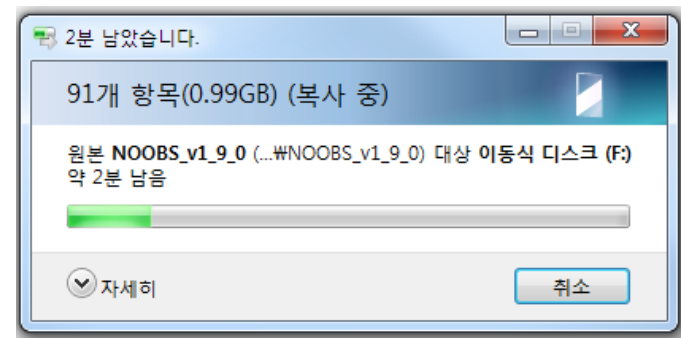
라즈베리파이 개발환경

NOOBS로 OS 설치하기

다운받은 NOOBS_v1_9_0.zip 파일을 압축을 풀고 디렉토리 안의 전체 파일을 선택하여 SD카드의 최상위 디렉토리에 그대로 모두 복사한다



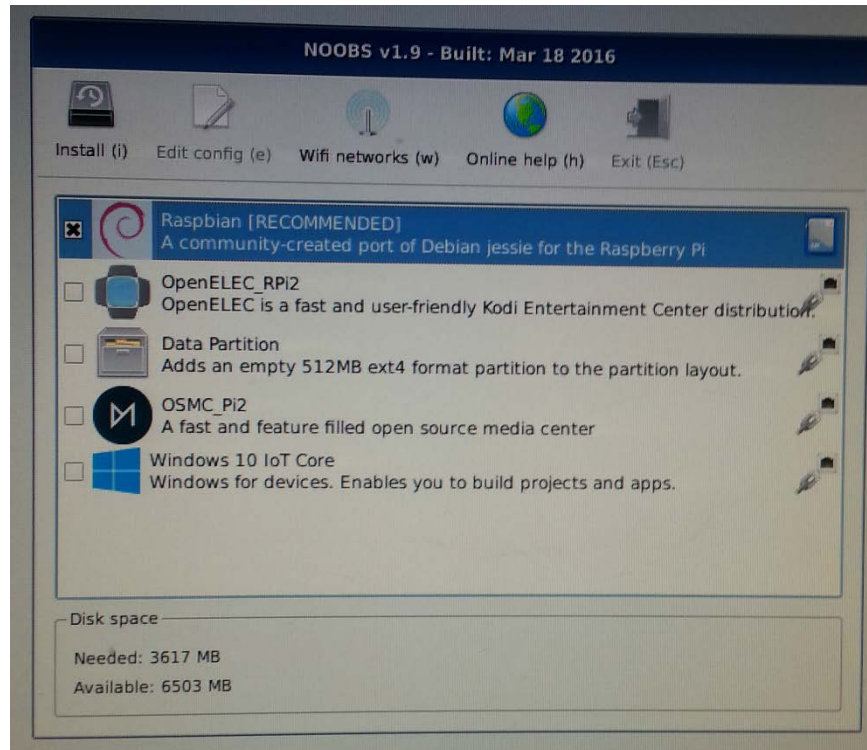
이름	수정된 날짜	유형	크기
defaults	2016-04-07 오후...	파일 폴더	
os	2016-04-07 오후...	파일 폴더	
overlays	2016-04-07 오후...	파일 폴더	
bcm2708-rpi-b.dtb	2016-03-18 오전...	DTB 파일	11KB
bcm2708-rpi-b-plus.dtb	2016-03-18 오전...	DTB 파일	11KB
bcm2709-rpi-2-b.dtb	2016-03-18 오전...	DTB 파일	12KB
bcm2710-rpi-3-b.dtb	2016-03-18 오전...	DTB 파일	13KB
INSTRUCTIONS-README.txt	2016-03-18 오전...	텍스트 문서	3KB
riscos-boot.bin	2016-03-18 오전...	BIN 파일	10KB
recovery7.img	2016-03-18 오전...	디스크 이미지 파일	2,471KB
bootcode.bin	2016-03-18 오전...	BIN 파일	18KB
BUILD-DATA	2016-03-18 오전...	파일	1KB
recovery.cmdline	2016-03-18 오전...	CMDLINE 파일	1KB
recovery.elf	2016-03-18 오전...	ELF 파일	600KB
recovery.img	2016-03-18 오전...	디스크 이미지 파일	2,419KB
recovery.rfs	2016-03-18 오전...	RFS 파일	22,840KB
RECOVERY_FILES_DO_NOT_EDIT	2016-04-07 오후...	파일	0KB



라즈베리파이 개발환경

NOOBS로 OS 설치하기

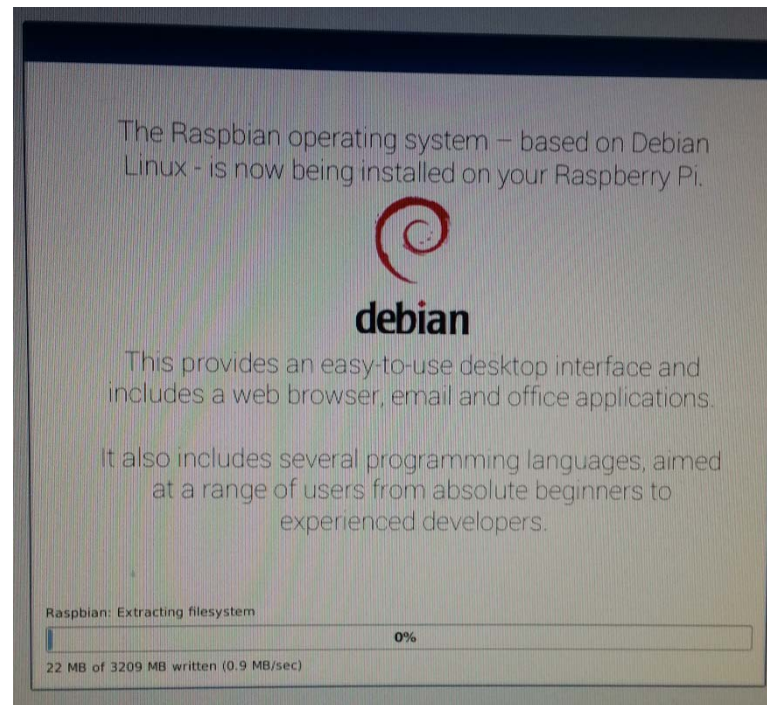
부팅 중 모니터에서 아래와 같은 메뉴가 나오면 메뉴 가장 위에 보여지는 Raspbian 을 선택하고 상단의 <Install> 버튼을 클릭



라즈베리파이 개발환경

NOOBS로 OS 설치하기

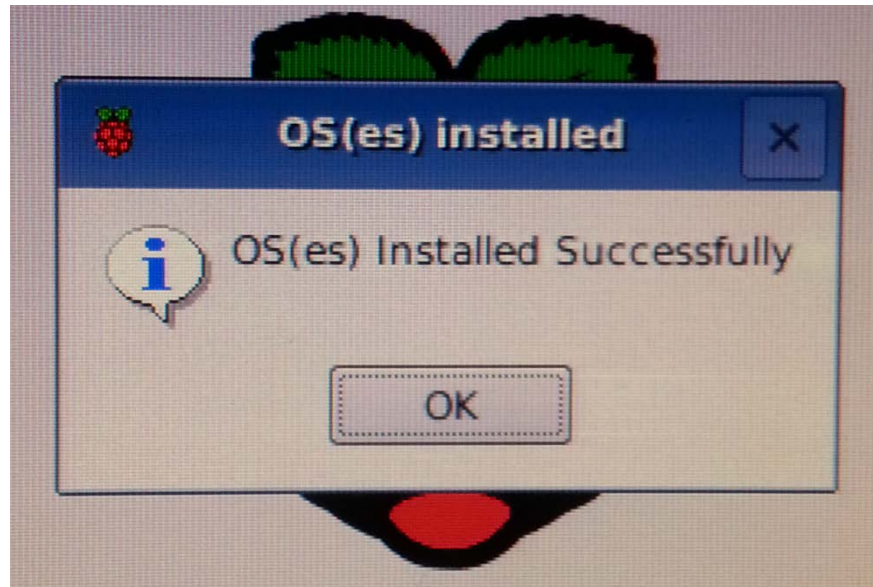
설치 과정이 진행되는 동안 10여분 정도 기다린다



라즈베리파이 개발환경

NOOBS로 OS 설치하기

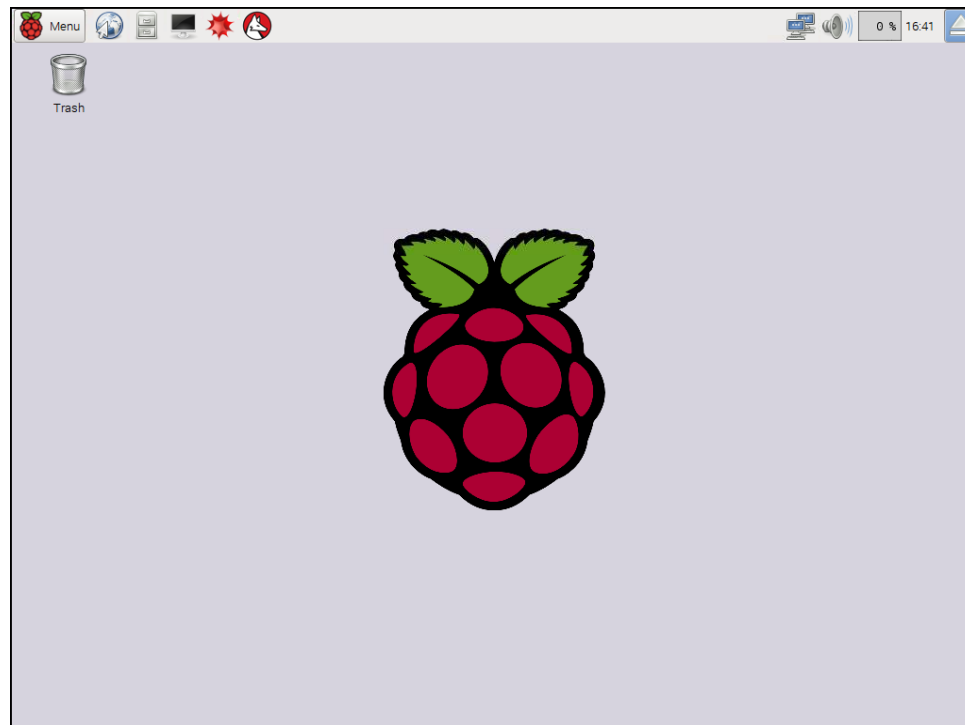
설치가 끝나면 [OK] 버튼을 누르면 재부팅된다



라즈베리파이 개발환경

NOOBS로 OS 설치하기

재부팅하면 자동으로 로그인하여 GUI 화면이 실행된다



Sensor Programming

센서 프로그래밍

GPIO



RaspberryPi



RASPBIAN

GPIO

- **GPIO – General Purpose Input and Output**
 - 범용 입/출력 장치를 지칭하는 용어
 - 일반적으로 AP(application processor)는 물리적인 형태를 가지는 IC로 구현될 때 (IC package) 제한된 개수의 핀을 가짐
 - AP 내부 기능은 물리적인 핀의 숫자보다 많음 (alternated function)
 - Ex) IC pin은 208개이고 내장된 기능이 약 600개가 될 때 약 1:3의 비율.
 - 핀 하나에 최소 3개의 기능과 입/출력 모드 설정까지 5개의 동작 모드를 지정함
 - 핀은 한 순간에 하나의 기능으로만 동작

GPIO

- **GPIO – General Purpose Input and Output** (continued)
 - GPIO는 IC에 내장된 기능과 입/출력 동작을 설정하고, 각 동작의 세부 상태를 설정하는 용도로 사용됨.
 - AP의 물리적인 특정 핀(AP의 물리적인 형태를 가지는 IC 실제 핀 하나에 대해)의 동작 상태를 입력 또는 출력으로 설정
 - 입력 상태와 출력 상태를 동시에 가질 수 없음
 - 한 순간에는 하나의 동작 상태만을 가짐
- H/W 제어 프로그램의 기본은 프로그래머가 의도한 시점에 지정된 핀의 상태를 High 또는 Low로 설정하는 것(또는 프로그래머가 의도한 시점에 특정 핀의 상태를 읽을 수 있는 것) → 연결된 장치에 따른 동작 발생(또는 동작 인식)

GPIO

- **GPIO – General Purpose Input and Output** (continued)

1. 핀 동작 설정 : 신호의 출력 또는 입력

GPIO는 프로그래머가 사용하는 MCU(micro control unit) 또는 AP의 물리적인 핀에 대해 입력 또는 출력 상태를 지정

2. 설정된 입력/출력에 대한 MCU 또는 AP에 내장된 기능 선택

MCU 또는 AP 에 따라 입/출력과 기능 선택을 같은 단계에서 지정할 수 있음

GPIO

BCM	wPi	Name	Peri #1	Pin No		Peri #1	Name	wPi	BCM
				1	2		5V		
2	8	SDA_1	SHT20_SDA1	3	4		5V		
3	9	SCL_1	SHT20_SCL1	5	6		GND		
4	7	GPIO_7	DCMotor_P	7	8	LED1, UART	TxD	15	14
		GND		9	10	LED2, UART	RxD	16	15
17	0	GPIO_0	LCD_D4	11	12	LCD_D5	GPIO_1	1	18
27	2	GPIO_2	LCD_D6	13	14		GND		
22	3	GPIO_3	LCD_D7	15	16	TLCD_RS	GPIO_4	4	23
				17	18	TLCD_RW	GPIO_5	5	24
10	12	MOSI	EE_MOSI	19	20		GND		
9	13	MISO	EE_MISO	21	22	DCMotor_N	GPIO_6	6	25
11	14	SCLK	EE_SCLK	23	24	EE_CS	CE_0	10	8
		GND		25	26	ADC_CS	CE_1	11	7
0	30	SDA_0		27	28		SCL_0	31	1
5	21	GPIO_21	JOG_UP	29	30		GND		
6	22	GPIO_22	JOG_DN	31	32	DC Moto PWM	GPIO_26	26	12
13	23	GPIO_23	PIEZO1	33	34		GND		
19	24	GPIO_24	IRLED_IN	35	36	JOG_LT	GPIO_27	27	16
26	25	GPIO_25	TLCD_E	37	38	JOG_RT	GPIO_28	28	20
		GND		39	40	JOG_CENTER	GPIO_29	29	21

Sensor Programming

센서 프로그래밍

GPIO LED



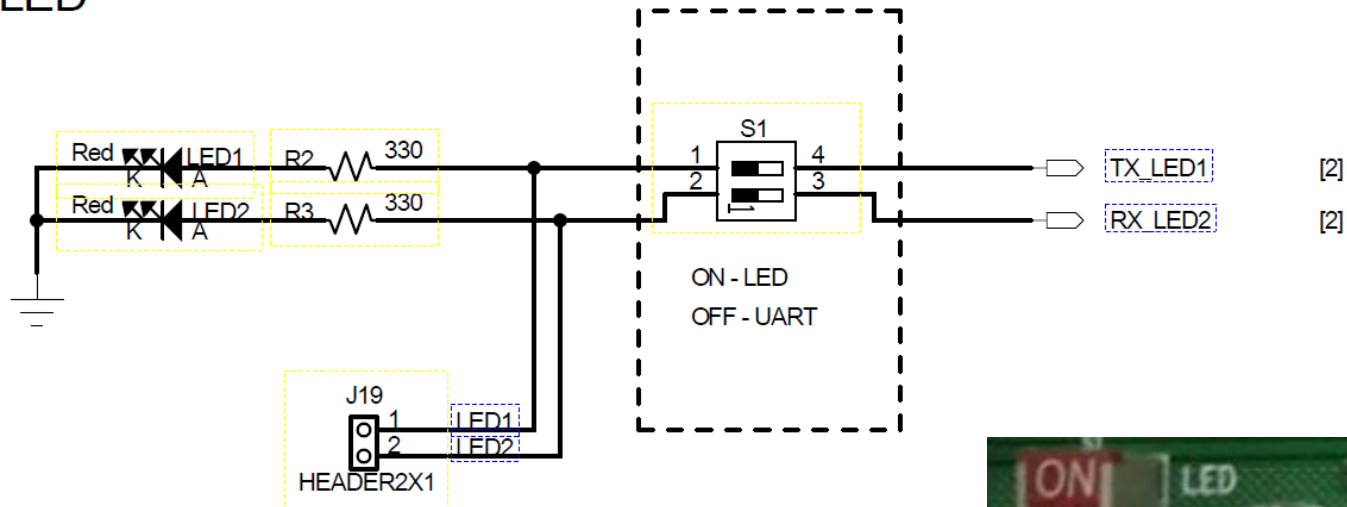
RaspberryPi



RASPBIAN

GPIO LED

LED



- TX_LED1는 BCM GPIO14와 연결
- RX_LED2는 BCM GPIO15와 연결
- 스위치를 사용하여 LED와 UART를 선택하여 사용할 수 있음
- 출력 전용 회로에 330옴 저항이 연결된 이유는?
 - LED 스펙의 최대 허용 볼트가 2.6v인 반면 GPIO는 3.3V
- J19 는 오실로 스코프로 신호 값 측정을 위한 용도



GPIO LED

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)

led_pin1 = 14
led_pin2 = 15

GPIO.setup(led_pin1, GPIO.OUT)
GPIO.setup(led_pin2, GPIO.OUT)

try:
    while True:
        GPIO.output(led_pin1, False)
        GPIO.output(led_pin2, False)
        time.sleep(1)
        GPIO.output(led_pin1, True)
        GPIO.output(led_pin2, True)
        time.sleep(1)
finally:
    print("Cleaning up")
    GPIO.cleanup()
```


GPIO LED

- `import RPi.GPIO as GPIO`
 - 라즈비안 OS에 기본 포함된 GPIO 라이브러리
- `import time`
 - 시간 지연 함수를 사용하기 위해 포함
- `GPIO.setmode(GPIO.BCM)`
 - 확장 커넥터의 핀 번호 할당 방식을 지정
- LED GPIO 핀 번호 지정
 - `led_pin1 = 14`
 - `led_pin2 = 15`
- LED 1의 GPIO 를 on/off로 설정
 - `GPIO.output(led_pin1, False)`
 - `GPIO.output(led_pin1, True)`

GPIO LED

- `time.sleep(1)`
 - 1초 시간 지연함수
 - 100ms 는 0.1
- `GPIO.cleanup()`
 - 사용된 모든 gpio의 출력 설정을 초기 상태의 입력으로 되돌림
- try/finally 구문
 - 프로그램이 시작되면 try 구문을 실행함
 - Ctrl+ C가 입력되면 finally 구문이 실행됨

Sensor Programming

센서 프로그래밍

GPIO 보터



RaspberryPi



RASPBIAN

GPIO 모터

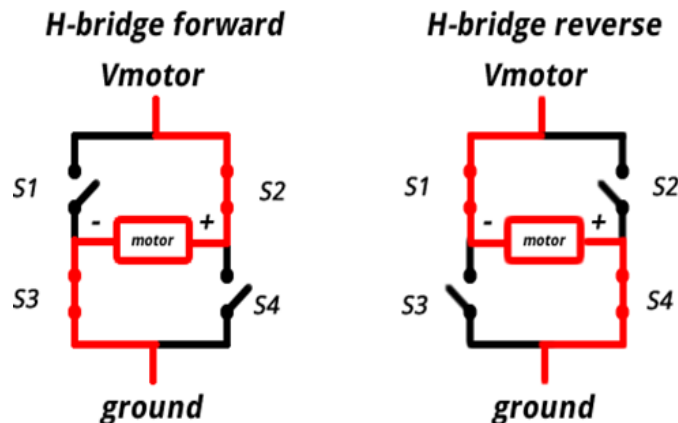
- **DC Motor 구동**

- DC Motor는 공급 전류에 비례하여 회전 속도 및 회전력이 증가함
- DC Motor가 구현되면 내부 저항이 고정되므로 공급 전압에 따라 전류가 비례함
- 따라서 공급 전압에 따라 회전력, 회전 속도가 변함.
- LED 밝기 제어 예제와 같은 코드를 사용하여 일정한 짧은 시간 간격으로 ON 구간과 OFF 구간의 비율을 달리하여 DC motor에 공급되는 전압을 가변하는 것과 동일한 효과 발생 → 회전 속도 제어

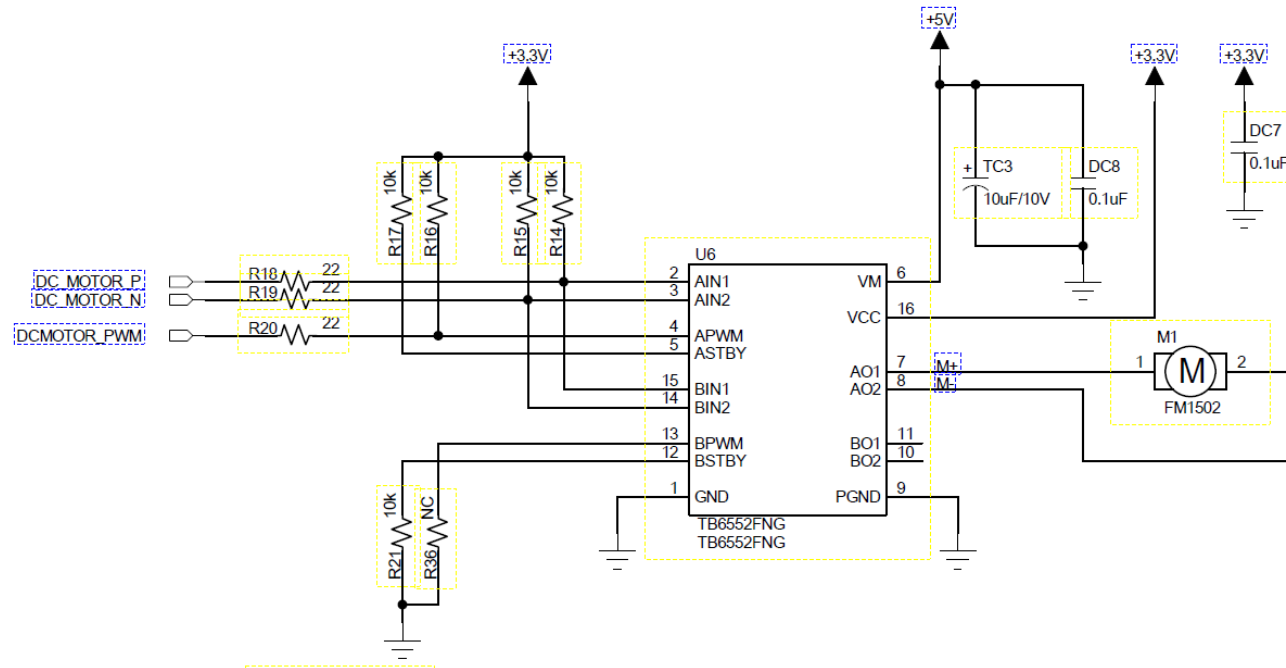
GPIO 모터

- DC Motor의 구동 회로

- DC motor의 회전 방향은 두 단자에 공급하는 전원의 극성에 따라 CW/CCW로 결정
- 회전 방향을 반전 시키려면 공급 전원의 극성을 변경해야 함
- H-Bridge
 - 전자 회로에서 4개의 스위치를 사용하여 ON/OFF 설정으로 DC 모터의 단자에 공급되는 전원의 극성을 제어할 수 있는 회로
 - DC motor를 회로 가운데 배치하고 4개의 스위치(Transistor)를 H 자 모양으로 배치한 것에서 유래됨

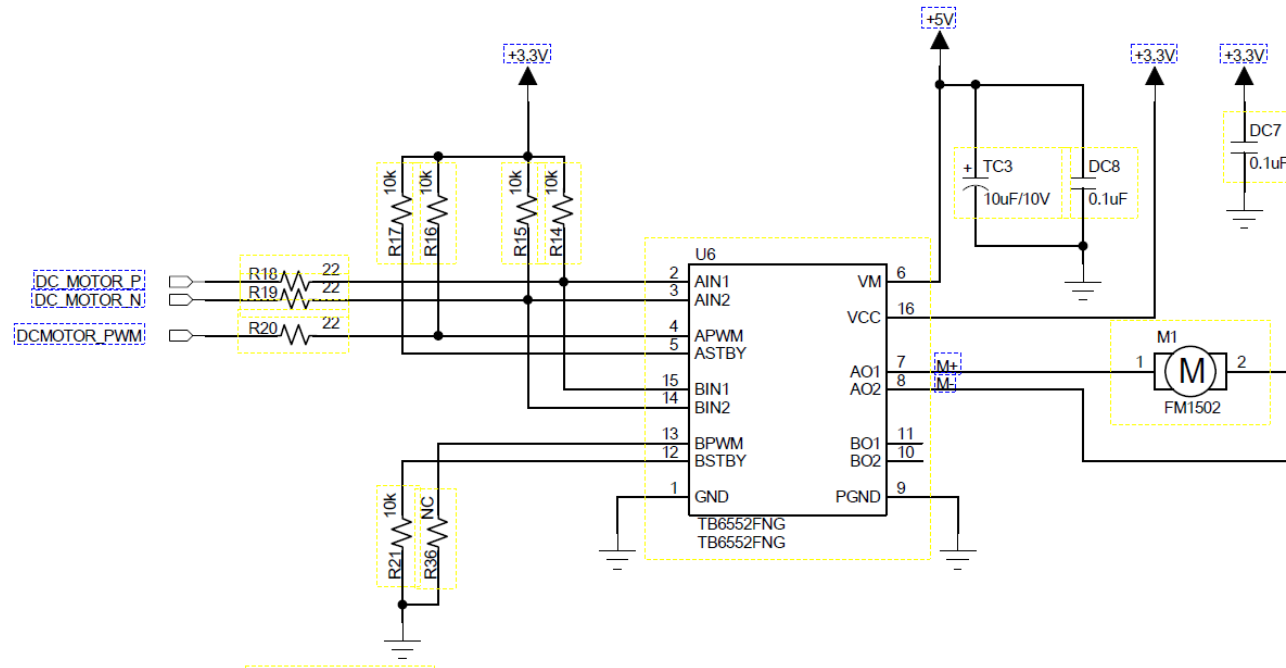


GPIO 모터



- VM : 모터의 파워를 위한 전원 (전류는 PGND로 흐름)
- VCC : 모터 드라이버의 컨트롤 로직의 전원 (전류는 GNC로 흐름)
- 캐패시터
 - TC3, DC8, DC7 소자는 캐패시터라고 부름
 - 갑자기 최대 속도로 모터를 회전하면 순간적으로 전류를 많이 소비하게 됨
 - 이런 상황이 발생하면 전류가 부족하여 일시적으로 모터를 구동할 수 없게 됨
 - 전류를 일정 용량 저장하여 공급 전원의 안정화 역할을 함

GPIO 모터



- TB6652FNG는 채널이 2개임 (A채널, B채널) 모터 2개를 연결 가능, 현재 회로는 A채널만 사용함.
- AIN1, AIN2 : 모터의 방향을 위한 설정을 위한 핀 (AP쪽으로 연결)
- APWM : 모터의 속도 설정을 위한 핀 (AP쪽으로 연결)
- ASTBY : 전류를 절약하기 위한 설정 핀, 현재는 사용 안함으로 3.3V에 연결됨
- AO1 : 모터는 극성이 없으므로 모터의 핀 중 한쪽을 연결
- AO2 : 모터의 핀 중 다른 한쪽을 연결

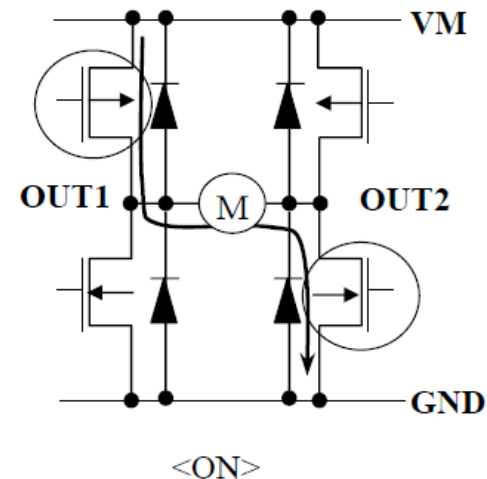
GPIO 모터

- **DC Motor의 구동 실습**
 - 실습 B/D의 DC Motor 구동 회로
 - H-Bridge 를 사용한 구동 회로
 - 모터는 입력의 DC_MOTOR_P, DC_MOTOR_N, DCMOTOR_PWM 신호에 동작
 - DC_MOTOR_P, DC_MOTOR_N 는 각각 High, Low 또는 Low, High의 조합일 경우에만 동작
 - CW (Clock Wise), CCW (Counter Clock Wise)로 회전 방향 설정
 - DC_MOTOR_P, DC_MOTOR_N 가 동작 상태를 만족할 때 DCMOTOR_PWM 에 따라 회전
 - High일 때 최대 속도, Low일 때 정지

GPIO 모터

- **DC Motor의 구동 실습 (continued)**
 - 실습 보드의 DC motor 구동 제어
 - 입력 신호에 따른 출력 동작

Input				Output		
IN1	IN2	PWM	STBY	OUT1	OUT2	Mode
H	H	H/L	H	L	L	Short brake
L	H	H	H	L	H	CCW
		L	H	L	L	Short brake
H	L	H	H	H	L	CW
		L	H	L	L	Short brake
L	L	H	H	OFF (High impedance)		Stop
H/L	H/L	H/L	L	OFF (High impedance)		Standby



GPIO 모터

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)

GPIO_RP = 4
GPIO_RN = 25
GPIO_EN = 12

GPIO.setup(GPIO_RP, GPIO.OUT)
GPIO.setup(GPIO_RN, GPIO.OUT)
GPIO.setup(GPIO_EN, GPIO.OUT)
```

GPIO 모터

```
try:
    while True:
        print 'forward'
        GPIO.output(GPIO_RP, True)
        GPIO.output(GPIO_RN, False)
        GPIO.output(GPIO_EN, True)
        time.sleep(1)

        print 'stop'
        GPIO.output(GPIO_EN, False)
        time.sleep(1)

        print 'backward'
        GPIO.output(GPIO_RP, False)
        GPIO.output(GPIO_RN, True)
        GPIO.output(GPIO_EN, True)
        time.sleep(1)

        print 'stop'
        GPIO.output(GPIO_EN, False)
        time.sleep(1)

finally:
    GPIO.cleanup()
```

Sensor Programming

센서 프로그래밍

GPIO 조이스틱

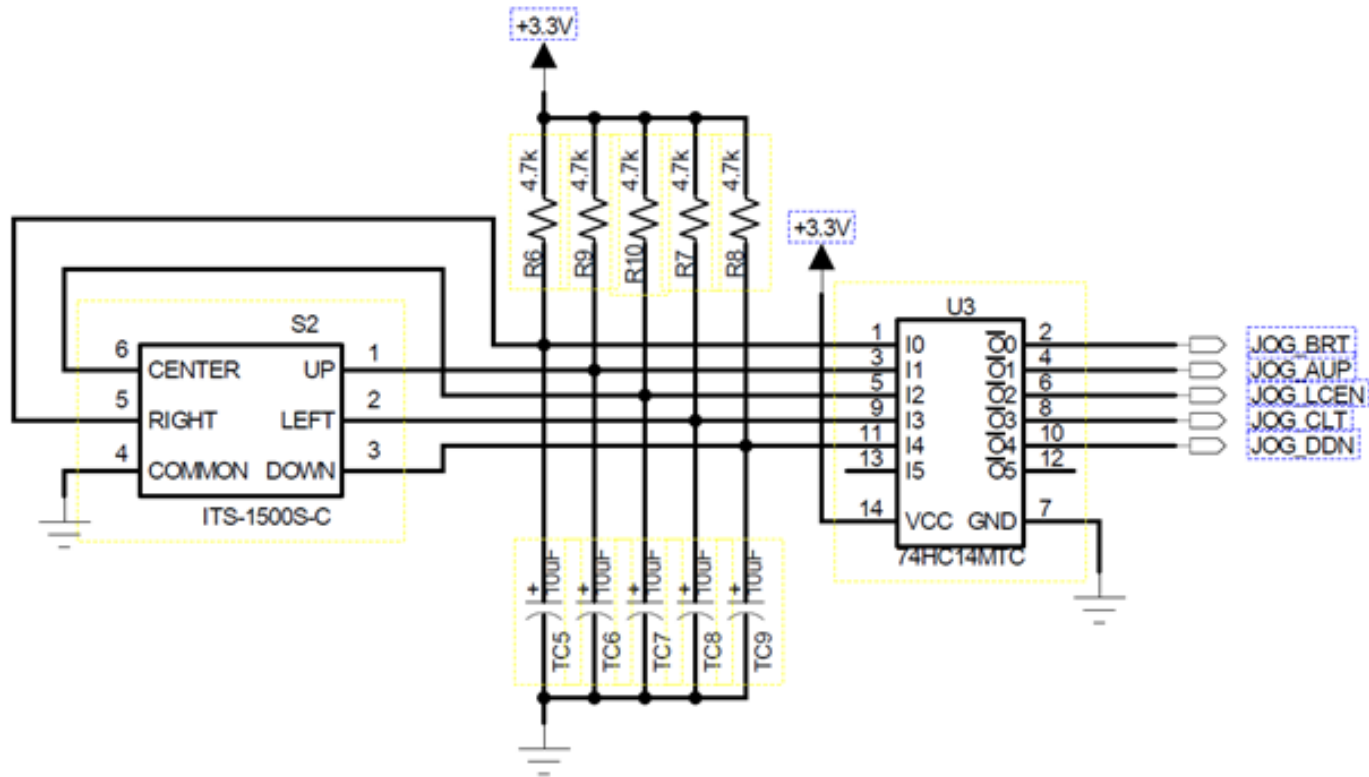


RaspberryPi



RASPBIAN

GPIO 조이스틱



- ITS-1500S-C : 4방향과 center 의 입력 감지가 가능한 부품
- R6 ~ R10
 - 3.3v에 연결되어 있으므로 풀업 저항이라 부름 (GND에 연결되면?)
 - 입력 회로에 연결될 경우 초기 상태를 HI신호로 결정하기 위함
 - 동작이 될 경우(버튼이 감지되면) 는 low신호가 되므로 low active라고 함
- 74HC14MTC : 채터링 (입력신호의 노이즈로 인한 오동작) 방지를 위해 사용됨

GPIO 조이스틱

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
gpio = [ 5, 6, 16, 20, 21]
stat = [ 0, 0, 0, 0, 0]

def print_jog_all():
    print ('up : %d, down: %d, left: %d, right : %d, cen: %d'%
           %(stat[0], stat[1], stat[2], stat[3], stat[4]))
```


GPIO 조이스틱

- `gpio = [5, 6, 16, 20, 21]`
 - BCM gpio pin 번호를 저장하기 위한 리스트 정의 (반복문을 활용을 위함)
 - up, down, left, right, center 순으로 정의
- `stat = [0, 0, 0, 0, 0]`
 - 현재의 값을 위해 저장할 리스트 정의
- `def print_jog_all():`
 - 이전값과 현재값이 변경될 경우 각 gpio의 상태를 모두 출력하기 위한 용도

GPIO 조이스틱

```
try:
    for i in range(5):
        GPIO.setup(gpio[i], GPIO.IN)

    cur_stat = 0

    while True:
        for i in range(5):
            cur_stat = GPIO.input(gpio[i])
            if cur_stat != stat[i]:
                stat[i] = cur_stat
                print_jog_all()

finally:
    print("Cleaning up")
    GPIO.cleanup()
```

GPIO 조이스틱

- `GPIO.setup(gpio[i], GPIO.IN)`
 - 리스트의 모든 GPIO를 입력으로 설정
- `cur_stat = 0`
 - 조작전 출력을 하지 않기 위한 초기값 설정
- `while True:`
 - 무한 반복
- `for i in range(5):`
 - 폴링으로 모든 핀의 변경을 감시
- `cur_stat = GPIO.input(gpio[i])`
 - 해당 핀의 현재 값을 읽어 `cur_stat` 에 저장함
- `if cur_stat != stat[i]:`
 - `stat[i] = cur_stat`
 - `print_jog_all()`
 - 이전 값과 현재 값이 다를 경우(변경된 경우) 현재 값을 `stat[i]`에 업데이트 후 출력

Sensor Programming

센서 프로그래밍

GPIO Piezo



RaspberryPi

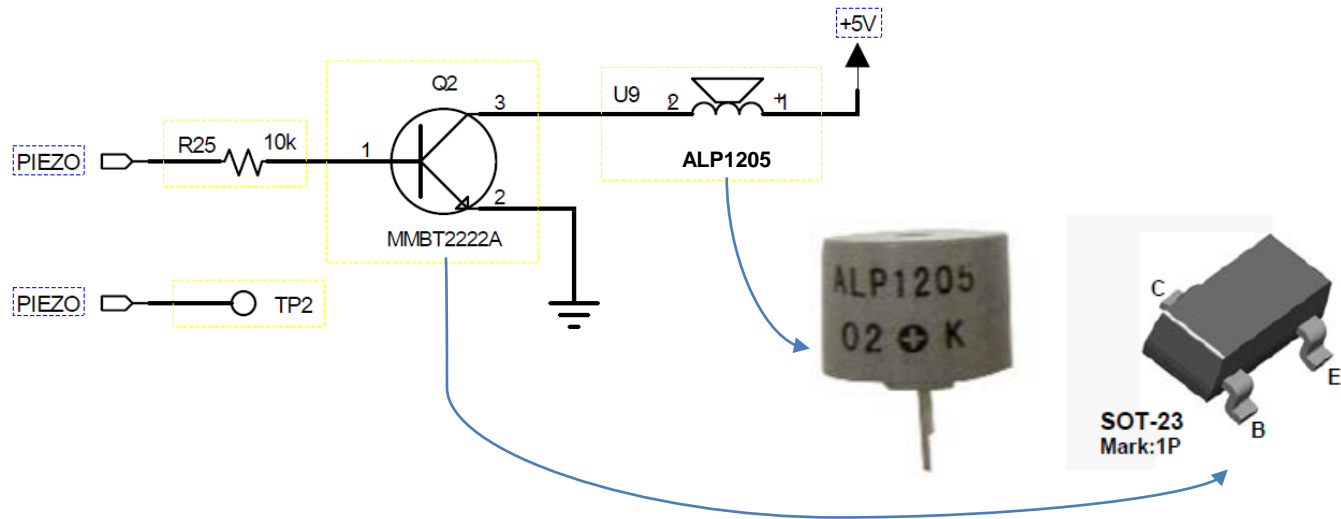


RASPBIAN

GPIO Piezo

- **Piezo 동작 제어**
 - Piezo는 Active type과 Passive type 으로 구분
 - Active type은 발진 회로 내장으로 정해진 Beep 음 발생
 - Passive type은 별도의 구동 회로 사용
 - 일정한 시간 간격으로 On/Off 할 때 음 발생
 - 발생하는 음은 음계별 표준 주파수 table 참조
 - 스위칭 주기(시간 간격) = on 시간 + off 시간

GPIO Piezo



- KPX-1205 : 마그네틱 부저
- MMBT2222A
 - IC 핀에서 흘릴 수 있는 전류는 보통 수십 mA
 - 부저같이 전류를 많이 필요로 하는 부품에 많은 양의 전류를 흘릴 수 있는 IC
 - 스펙은 최대 500mA까지 허용함.

GPIO Piezo

- **Piezo 동작 제어** (continued)
 - 발생하는 음은 음계별 표준 주파수 table

음계 \ 옥타브	1	2	3	4	5	6	7	8
C(도)	32.7032	65.4064	130.8128	261.6256	523.2511	1046.502	2093.005	4186.009
C#	34.6478	69.2957	138.5913	277.1826	554.3653	1108.731	2217.461	4434.922
D(레)	36.7081	73.4162	146.8324	293.6648	587.3295	1174.659	2349.318	4698.636
D#	38.8909	77.7817	155.5635	311.1270	622.2540	1244.508	2489.016	4978.032
E(미)	41.2034	82.4069	164.8138	329.6276	659.2551	1318.510	2637.020	5274.041
F(파)	43.6535	87.3071	174.6141	349.2282	698.4565	1396.913	2793.826	5587.652
F#	46.2493	92.4986	184.9972	369.9944	739.9888	1479.978	2959.955	5919.911
G(솔)	48.9994	97.9989	195.9977	391.9954	783.9909	1567.982	3135.963	6271.927
G#	51.9130	103.8262	207.6523	415.3047	830.6094	1661.219	3322.438	6644.875
A(라)	55.0000	110.0000	220.0000	440.0000	880.0000	1760.000	3520.000	7040.000
A#	58.2705	116.5409	233.0819	466.1638	932.3275	1864.655	3729.310	7458.620
B(시)	61.7354	123.4708	246.9417	493.8833	987.7666	1975.533	3951.066	7902.133

GPIO Piezo

- **Piezo 소자 특징**
 - 실습 장치에 사용되는 Piezo 소자의 특징
 - Passive Piezo
 - 동작 주파수에 따라 음량 편차
 - 음계 별로 서로 다른 크기의 소리 발생
 - 고품질 음악 발생용으로는 부적합
 - 단순 멜로디 발생 용도
 - 동작 시작 / 종료 및 오류 상태 등의 상태 표시 용도

GPIO Piezo

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
gpio_pin=13
scale = [ 261, 294, 329, 349, 392, 440, 493, 523 ]
GPIO.setup(gpio_pin, GPIO.OUT)

try:
    p = GPIO.PWM(gpio_pin, 100)
    p.start(100)          # start the PWM on 100% duty cycle
    p.ChangeDutyCycle(90) # change the duty cycle to 90%

    for i in range(8):
        print (i+1)
        p.ChangeFrequency(scale[i])
        time.sleep(1)

    p.stop()              # stop the PWM output

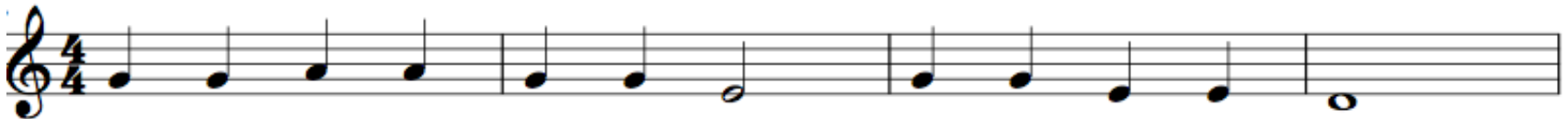
finally:
    GPIO.cleanup()
```

GPIO Piezo

- `scale = [261, 294, 329, 349, 392, 440, 493, 523]`
 - 4옥타브 도 부터 5옥타브 도까지의 음계를 위한 list 선언
- `p = GPIO.PWM(gpio_pin, 100)`
 - `gpio_pin`의 주파수 100인 `pwm` 인스턴스를 생성
- `p.start(100)`
 - Duty cycle을 100%로 시작 설정
- `p.ChangeDutyCycle(90)`
 - Duty cycle을 90%로 변경
- `for i in range(8):`
 - 4옥타브 도부터 `scale` list의 음계를 반복 수행하기 위한 루프
- `p.ChangeFrequency(scale[i])`
 - 주파수를 `scale`의 음계로 변경
- `p.stop()`
 - `pwm` 정지

GPIO Piezo

- Piezo 제어 실습
 - 음계 발생
 - Piezo 연결된 BCM 13 핀의 ON/OFF
 - 일정한 시간 간격으로 High, Low 출력
 - 박자 제어
 - 발생 음의 지속 시간
 - 연속 음의 구분을 위한 일정 휴지 구간 설정
 - 아래의 악보를 실습 장치의 buzzer를 사용하여 음 발생 실습



Sensor Programming

센서 프로그래밍

GPIO character LCD



RaspberryPi

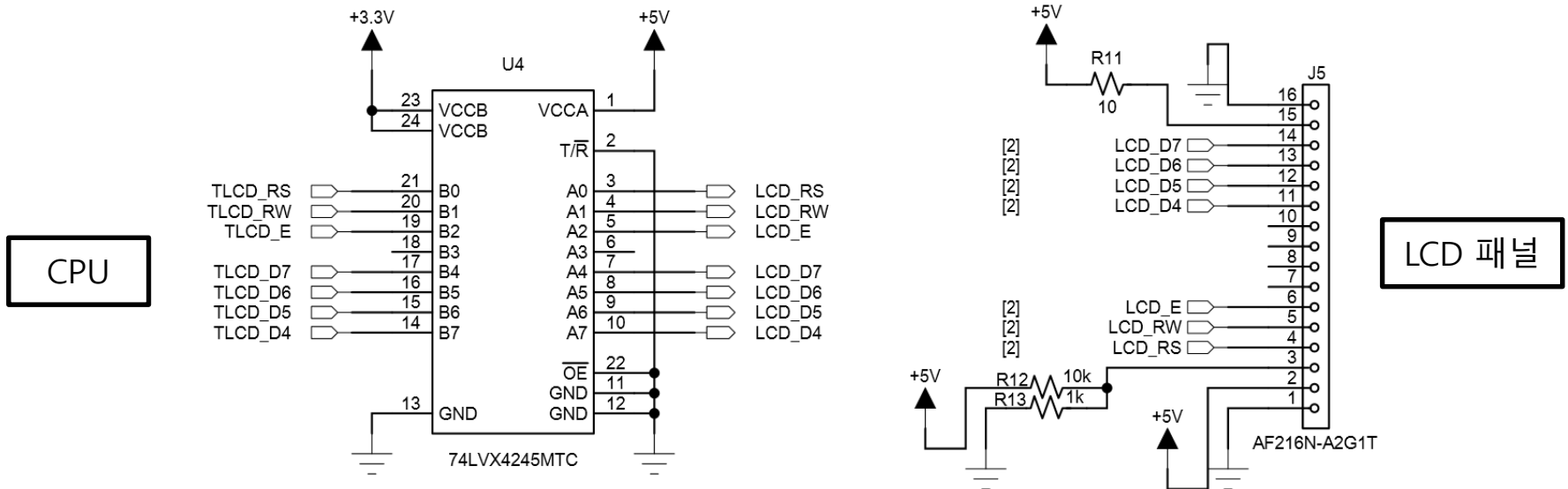


RASPBIAN

GPIO Character LCD

- Character LCD
 - 기억된 내용을 눈으로 직접 볼 수 있게 만든 디스플레이 장치.
 - 디스플레이 부와 제어 부가 하나로 통합된 모듈 형태로 판매.
 - 실습 보드는 16 x 2 문자 LCD 사용.
 - 4비트, 8 비트 마이크로프로세서와 인터페이스 가능.
 - 5x8, 5x10 도트 디스플레이 가능.
 - 80x8비트의 디스플레이 램(최대 80글자).
 - 240 문자 폰트를 위한 문자 발생기 ROM.
 - 64x8비트 문자 발생기 RAM.
 - +5V 전원 사용.

GPIO Character LCD

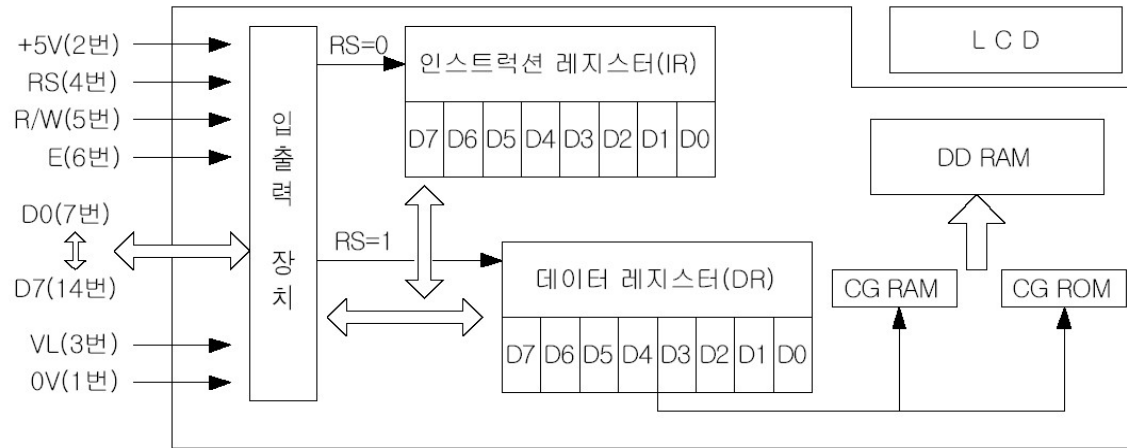


74LVX4245MTC

- 3v(CPU, B port) 와 5v(LCD패널, A port) 버스 사이의 인터페이스를 위한 IC

GPIO Character LCD

- Character LCD 구조



- IR (instruction register) : LCD on/off, clear, function set 등을 설정하는 레지스터.
- DR (data register) : LCD 모듈에 문자를 나타내기 위한 데이터 값이 들어가는 레지스터.
- DD RAM : 최대 80 글자(80x8비트)를 출력할 수 있는 디스플레이 RAM
- CG RAM : 사용자 정의 64 글자(64x8비트) 발생기용 RAM
- CG ROM : 240 문자 폰트 ROM
- D0 ~ D7 : 명령(rs = 0) 또는 데이터(rs = 1) 입력 핀 (4비트일 경우 D4~D7)
- RS (resister select) : 명령 또는 데이터 선택 핀

GPIO Character LCD

pin	Signal Name	기 능		
1	VSS	전원 GND		
2	VDD	전원 +5VDC		
3	VEE	Contrast 제어 전압 레벨 (VDD-VEE = 13.5 ~ 0V)		
4	RS	Register Select (0 = instruction, 1 = data)		
5	R/W	Read/Write (0 = write, 1 = read)		
6	E	Enable Signal for read/write LCD		
7	DB0 (LSB)	4비트 데이터 버스 이용 시 사용 안함	8비트 데이터 버스 이용 시 모두 사용	
8	DB1			
9	DB2			
10	DB3			
11	DB4	4비트 데이터 버스 이용 시 사용		
12	DB5			
13	DB6			
14	DB7 (MSB)			
15	A	+LED (backlight LED용 전원 +4.4V ~ +4.7V)		
16	K	-LED (backlight LED용 전원 GND)		

GPIO Character LCD

- RS와 RW 관계

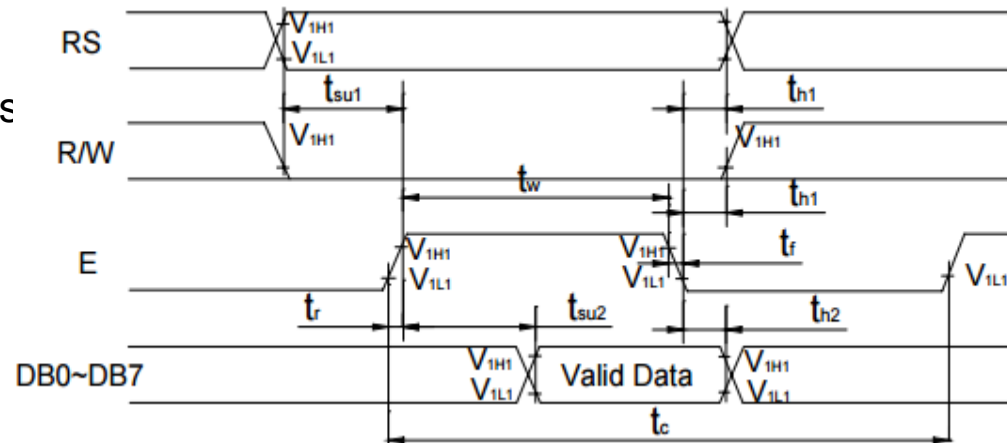
RS	R/W	LCD 동작 상태
0	0	IR 선택하여 제어 명령 쓰기 예) LCD 화면 클리어, 커서 시프트, LCD ON/OFF 등등.
0	1	D7로부터 비지 플래그 읽기 어드레스 카운터를 D0~D6으로부터 읽기
1	0	DR이 선택되어 데이터 값 쓰기
1	1	DR이 선택되어 데이터 값 읽기

- rs = 0 : 명령, rs = 1 : 데이터
- r/w = 0 : 쓰기, rw = 1 : 읽기
- 주소 버스는 없고 데이터 버스만 존재.
- 데이터 버스에 제어 명령과 데이터 정보가 함께 전달.
- rs로 데이터 버스에 실리는 정보가 제어 명령인지 데이터인지 구분 필요.

GPIO Character LCD

- 쓰기 모드 타이밍
 - rs를 high 또는 low로 설정
 - r/w를 low로 설정
 - 40ns 이후 e를 high로 설정
 - 80ns 이후 db0~7에 데이터 설정
 - 데이터 전송 완료
 - e를 low로 설정

Mode	Symbol	Min.	Typ.	Max.	Unit
E Cycle Time	t_c	500	-	-	ns
E Rise / Fall Time	t_r, t_f	-	-	20	ns
E Pulse Width (High, Low)	t_w	230	-	-	ns
RW and RS Setup Time	t_{su1}	40	-	-	ns
RW and RS Hold Time	t_{h1}	10	-	-	ns
Data Setup Time	t_{su2}	80	-	-	ns
Data Hold Time	t_{h1}	10	-	-	ns

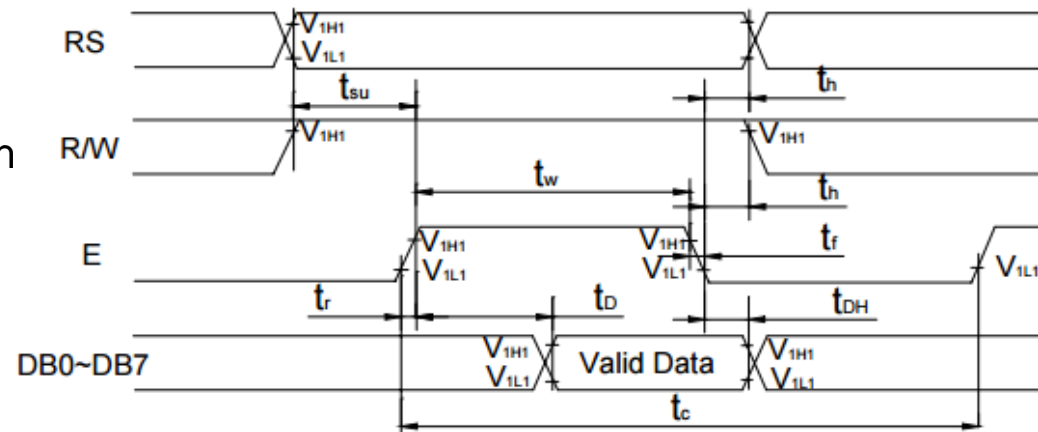


- 데이터 쓰기 구간은 최소 500ns

GPIO Character LCD

- 읽기 모드 타이밍
 - rs를 high 또는 low로 설정
 - r/w를 high로 설정
 - 40ns 이후 e를 high로 설정
 - 120ns 이후 db0~7에 데이터 읽기
 - 데이터 읽기 완료
 - e를 low로 설정
 - 데이터 읽기 구간은 최소 500n

Mode	Symbol	Min.	Typ.	Max.	Unit
E Cycle Time	t_c	500	-	-	ns
E Rise / Fall Time	t_{r, t_f}	-	-	20	ns
E Pulse Width (High, Low)	t_w	230	-	-	ns
RW and RS Setup Time	t_{su1}	40	-	-	ns
RW and RS Hold Time	t_{h1}	10	-	-	ns
Data Output Delay Time	t_{su2}	-	-	120	ns
Data Hold Time	t_{h1}	5	-	-	ns



GPIO Character LCD

- Character LCD 표시 제어 명령

기능	제어 신호		제어 명령							
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0
Clear Display	0	0	0	0	0	0	0	0	0	1
Return Home	0	0	0	0	0	0	0	0	1	0
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	S
Display ON/OFF Control	0	0	0	0	0	0	1	D	C	B
Cursor or Display Shift	0	0	0	0	0	1	S/C	R/L	0	0
Function Set	0	0	0	0	1	DL	N	F	0	0
Set CG RAM address	0	0	0	1	CG RAM address					
Set DD RAM address	0	0	1	DD RAM address						
Read busy flag and address	0	1	BF	Address Counter						
Data write to CG RAM or DD RAM	1	0	Write address							
Data read from CG RAM or DD RAM	1	1	Read address							

GPIO Character LCD

- Character LCD 표시 제어 명령 (continued)
 - Character LCD는 전송 받은 각 명령을 실행하기 위해 일정 시간 요구.
 - 다음 명령을 전송하기 전에 충분히 시간 지연.
 - BUSY 플래그를 읽어 선행된 명령이 완료되었는지 확인.
 - Clear Display
 - 모든 디스플레이 상태를 소거하고 커서를 Home 위치 이동.
 - Return Home
 - DD RAM의 내용은 변경하지 않고 커서만 Home으로 이동.
 - Entry Mode SET
 - 데이터를 읽거나 쓸 때 커서 위치 증가(I/D=1), 감소(I/D=0) 결정
 - 이때 화면을 이동 할지(S=1) 아닌지(S=0) 결정
 - Display ON/OFF Control
 - 화면 표시 ON/OFF(D), 커서 ON/OFF(C) 커서 깜박임(B) 설정

GPIO Character LCD

- Character LCD 표시 제어 명령 (continued)
 - Cursor or Display Shift
 - 화면(S/C=1) 또는 커서(S/C=0)를 오른쪽(R/L=1), 왼쪽(R/L=0)으로 이동.
 - Function SET
 - **데이터 길이**를 8비트(DL=1) 또는 **4비트(DL=0)로 지정.**
 - 화면 표시 행수를 2행(N=1) 또는 1행(N=0)으로 지정.
 - 문자 폰트를 5 x 10 도트(F=1) 또는 5 x 7도트(F=0)로 지정.
 - 전원 투입 후 초기화 코드에서 사용.
 - Character LCD reset에 약50ms가 소요되므로 충분히 기다린 후 명령 전송.
 - Set CG RAM Address
 - Character Generator RAM의 어드레스 지정.
 - 이후 송수신하는 데이터는 CG RAM 데이터

GPIO Character LCD

- Character LCD 표시 제어 명령 (continued)
 - Set DD RAM Address
 - Display Data RAM의 어드레스 지정.
 - 이후 송수신하는 데이터는 DD RAM 데이터.
 - Read Busy Flag & Address
 - Character LCD가 내부 동작 중임을 나타내는 Busy Flag(BF) 읽기.
 - 어드레스 카운터 내용 읽기.
 - Character LCD가 각 제어 코드를 실행하는 데는 일정한 시간이 필요.
 - 프로세서가 BF를 읽어 '1'이면 기다리고 '0'이면 다음 제어 코드 전송

GPIO Character LCD

- Character LCD 모듈 초기화
 - 전원 인가
 - Character LCD가 reset되려면 약 30ms 이상 소요되므로 이 시간 동안 대기.
 - Function set 명령(001X_XX00) 전송 후 39us 이상 대기.
 - Display ON/OFF control 명령(0000_1XXX) 전송 후 39us 이상 대기.
 - Display Clear 명령(0000_0001) 전송 후 1.53ms 이상 대기.
 - Entry mode set 명령(0000_01XX) 전송.
 - 필요에 따라 DD RAM address 전송 후 문자 데이터 연속 전송.

GPIO Character LCD

```
#!/usr/bin/python

import RPi.GPIO as GPIO
import time

# Define GPIO to LCD mapping
LCD_RS = 23
LCD_RW = 24
LCD_E  = 26
LCD_D4 = 17
LCD_D5 = 18
LCD_D6 = 27
LCD_D7 = 22
```

GPIO Character LCD

```
# Define some device constants
```

```
LCD_WIDTH = 16    # Maximum characters per line
```

```
LCD_CHR = True
```

```
LCD_CMD = False
```

```
LCD_LINE_1 = 0x80 # LCD RAM address for the 1st line
```

```
LCD_LINE_2 = 0xC0 # LCD RAM address for the 2nd line
```

```
# Timing constants
```

```
E_PULSE = 0.0005
```

```
E_DELAY = 0.0005
```

GPIO Character LCD

```
def main():  
    GPIO.setwarnings(False)  
    GPIO.setmode(GPIO.BCM)      # Use BCM GPIO numbers  
    GPIO.setup(LCD_E, GPIO.OUT) # E  
    GPIO.setup(LCD_RS, GPIO.OUT) # RS  
    GPIO.setup(LCD_D4, GPIO.OUT) # DB4  
    GPIO.setup(LCD_D5, GPIO.OUT) # DB5  
    GPIO.setup(LCD_D6, GPIO.OUT) # DB6  
    GPIO.setup(LCD_D7, GPIO.OUT) # DB7  
  
    # Initialise display  
    lcd_init()
```

GPIO Character LCD

```
# def main(): (continued)
    while True:

        # Send some test
        lcd_string("Raspberry Pi",LCD_LINE_1)
        lcd_string("16x2 LCD Test",LCD_LINE_2)
        time.sleep(3) # 3 second delay

        # Send some text
        lcd_string("1234567890123456",LCD_LINE_1)
        lcd_string("abcdefghijklmnop",LCD_LINE_2)
        time.sleep(3) # 3 second delay
```

GPIO Character LCD

```
def lcd_init():  
    # Initialise display  
    lcd_byte(0x33,LCD_CMD) # 110011 Initialise  
    lcd_byte(0x32,LCD_CMD) # 110010 Initialise  
    lcd_byte(0x06,LCD_CMD) # 000110 Cursor move direction  
    lcd_byte(0x0C,LCD_CMD) # 001100 Display On,Cursor Off, Blink Off  
    lcd_byte(0x28,LCD_CMD) # 101000 Data length, number of lines, font size  
    lcd_byte(0x01,LCD_CMD) # 000001 Clear display  
    time.sleep(E_DELAY)
```

GPIO Character LCD

```
def lcd_byte(bits, mode):  
    # Send byte to data pins  
    # bits = data  
    # mode = True  for character  
    #           False for command  
  
    GPIO.output(LCD_RS, mode) # RS
```

GPIO Character LCD

```
# def lcd_byte (bits, mode): (continued)
# High bits
GPIO.output(LCD_D4, False)
GPIO.output(LCD_D5, False)
GPIO.output(LCD_D6, False)
GPIO.output(LCD_D7, False)
if bits&0x10==0x10:
    GPIO.output(LCD_D4, True)
if bits&0x20==0x20:
    GPIO.output(LCD_D5, True)
if bits&0x40==0x40:
    GPIO.output(LCD_D6, True)
if bits&0x80==0x80:
    GPIO.output(LCD_D7, True)
# Toggle 'Enable' pin
lcd_toggle_enable()
```

GPIO Character LCD

```
# def lcd_byte (bits, mode): (continued)
# Low bits
GPIO.output(LCD_D4, False)
GPIO.output(LCD_D5, False)
GPIO.output(LCD_D6, False)
GPIO.output(LCD_D7, False)
if bits&0x01==0x01:
    GPIO.output(LCD_D4, True)
if bits&0x02==0x02:
    GPIO.output(LCD_D5, True)
if bits&0x04==0x04:
    GPIO.output(LCD_D6, True)
if bits&0x08==0x08:
    GPIO.output(LCD_D7, True)
# Toggle 'Enable' pin
lcd_toggle_enable()
```


GPIO Character LCD

```
def lcd_toggle_enable():
    # Toggle enable
    time.sleep(E_DELAY)
    GPIO.output(LCD_E, True)
    time.sleep(E_PULSE)
    GPIO.output(LCD_E, False)
    time.sleep(E_DELAY)

def lcd_string(message,line):
    # Send string to display
    message = message.ljust(LCD_WIDTH," ")
    lcd_byte(line, LCD_CMD)
    for i in range(LCD_WIDTH):
        lcd_byte(ord(message[i]),LCD_CHR)
```

GPIO Character LCD

```
if __name__ == '__main__':  
    try:  
        main()  
    except KeyboardInterrupt:  
        pass  
    finally:  
        lcd_byte(0x01, LCD_CMD)  
        lcd_string("Goodbye!",LCD_LINE_1)  
        GPIO.cleanup()
```

Sensor Programming

센서 프로그래밍

ultrasonic
Sensor



RaspberryPi



RASPBIAN

Ultrasonic Sensor

- **Ultrasonic Sensor : 초음파 센서**

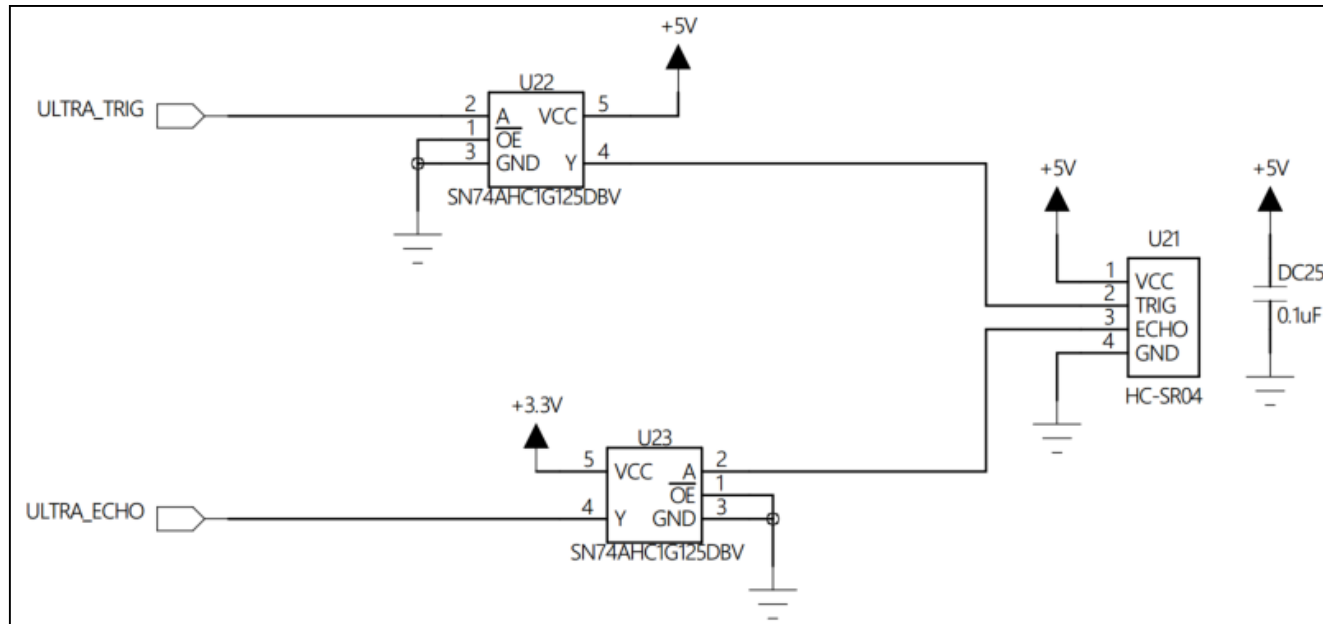
초음파 센서는 박쥐처럼 초음파를 공기 중에 방사한 후 물체에 반사되어 돌아오는 시간을 계산해 거리를 파악한다. Perio 모듈에 포함된 초음파 센서 (HC-SR04)는 송수신기가 결합된 모듈 타입으로 햇빛이나 검은색 물질에 영향을 덜 받으며 2cm ~ 400cm 범위의 거리를 측정할 수 있다.



Ultrasonic Sensor

- 회로 구성

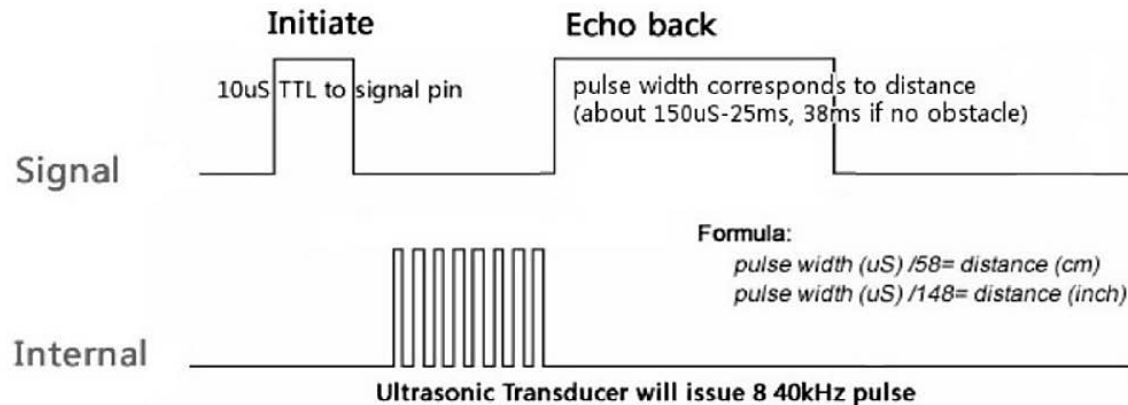
동작 전압은 5V이므로 송신기와 연결되는 ULTRA_TRIG는 3.3V GPIO 출력을 5V로 변환하기 위해 레벨 버퍼 U22를 거치며 수신기와 연결되는 ULTRA_ECHO는 5V 출력을 GPIO의 유효 입력 전압인 3.3V로 변경하기 위해 레벨 버퍼 U23을 거쳐 들어온다.



Ultrasonic Sensor

- TRIG/ECHO 신호

초음파 센서의 TRIG 핀에 최소한 10마이크로 초 동안 HIGH 신호를 전달하면 송신기는 40kHz 초음파 8개를 방사한 후 반사되어 돌아오는 것을 기다린다. 수신기에서 반사되어 돌아온 초음파를 감지하면 ECHO를 HIGH로 설정한 다음 거리에 비례하는 마이크로 초(ms) 동안 유지(Time)한다. 따라서 ECHO 핀으로 수신되는 약 150us ~ 25ms 범위의 HIGH 레벨 펄스 폭은 거리에 해당하며, 장애물이 없는 경우 38ms 동안 유지된다.



Ultrasonic Sensor

- 거리 계산
초음파의 속도는 약 340m/s (1초에 340m 이동)

속도 = 거리 / 시간

거리(distance) = 속도 * 시간

공식에 따라 속도는 340m/s, 거리는 distance ,
시간은 duration / 2 (왕복거리이므로)

$$\text{distance} = 340 * 100 * \text{duration} / 2 = \text{duration} * 17000$$

단위가 meter였으므로 cm로 나타내기위해 100을 곱하면
17000

round를 이용하여 소숫점 아래 3째자리에서 반올림

Ultrasonic Sensor

```
import RPi.GPIO as GPIO
import time

GPIO.setmode(GPIO.BCM)
GPIO.setwarnings(False)

trig = 0
echo = 1

GPIO.setup(trig, GPIO.OUT)
GPIO.setup(echo, GPIO.IN)          # Peri v 2.1
#GPIO.setup(echo, GPIO.IN,GPIO.PUD_UP) # Peri v 2.0

try :

    while True :

        GPIO.output(trig, False)
        time.sleep(0.5)

        GPIO.output(trig, True)
        time.sleep(0.00001)
        GPIO.output(trig, False)
```


Ultrasonic Sensor

```
while GPIO.input(echo) == False : # Peri v 2.1
#while GPIO.input(echo) == True : # Peri v 2.0
    pulse_start = time.time()
```

```
while GPIO.input(echo) == True : # Peri v 2.1
#while GPIO.input(echo) == False : # Peri v 2.0
    pulse_end = time.time()
```

```
pulse_duration = pulse_end - pulse_start
distance = pulse_duration * 17000
distance = round(distance, 2)
```

```
print ("Distance : ", distance, "cm")
```

```
except :
    GPIO.cleanup()
```

Sensor Programming

센서 프로그래밍

IR(Infrared)
Receiver



RaspberryPi

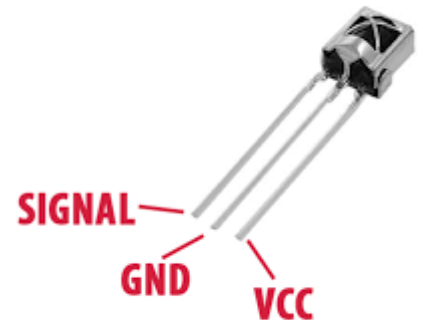


RASPBIAN

IR Receiver

- IR (infrared) Receiver

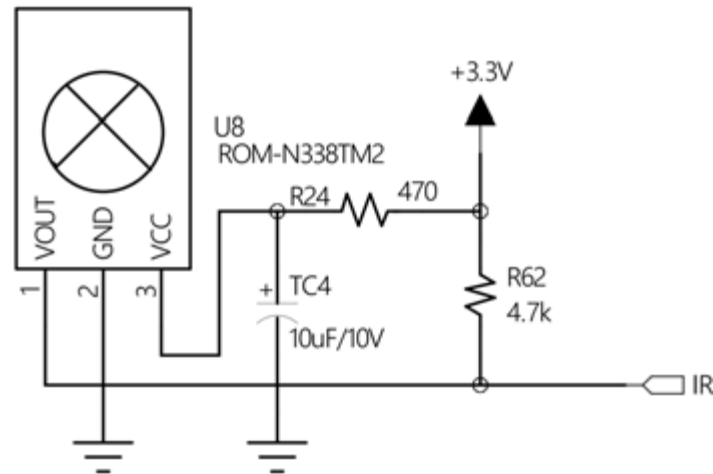
적외선 Infrared Rays은 파장이 약 780nm ~ 1mm 범위의 전자기파를 일컫는 말로 가시광선 영역에서 파장이 가장 긴 빨간색 밖에 위치한다. 적외선은 다시 파장이 길어지는 순서로 근적외선, 중적외선, 원적외선으로 나뉘며, 이 중 대부분의 영역은 열선이라 불리는 원적외선($4\mu\text{m}\sim 1\text{mm}$)이 차지한다. 780nm ~ 2000nm 범위의 근적외선은 가시광선보다 파장이 길어서 회절이 잘 일어나고 사람의 눈으로 인지할 수 없으며 원적외선과 달리 열을 발생시키지 않으므로 가전제품의 제어에 주로 사용한다.



IR Receiver

- 회로 구성

Peri0의 IR Receiver는 적외선을 감지하는 포토다이오드 Phototransistor와 대역 통과 필터 Band Pass Filter가 통합된 모듈 타입으로 수신되는 적외선 신호 중 특정 주파수의 적외선 신호만 분리해 출력 핀(VOUT)으로 내보낸다. 입력 핀(IR)이 풀 업 상태이므로 기본값은 HIGH이고 적외선 신호가 감지될 때마다 Low(1) 신호가 IR로 전달된다.



IR Receiver

- LIRC(Linux Infrared Remote Control)

/etc/lirc/ 디렉터리에 위치하는 iCORE-SDP의 기존 LEG 리모컨 용 키 맵 파일 lircd.conf를 백업한 후 새로 생성한 키 맵 파일을 복사한다.

```
tea@planx:~ $ sudo mv /etc/lirc/lircd.conf    /etc/lirc/lge_lircd.conf
```

```
tea@planx:~ $ sudo cp /etc/lirc/car_lird.conf  /etc/lirc/lircd.conf
```

lircrc는 lircd.conf에 정의된 키 이름을 특정 문자열로 변환해 응용프로그램에 전달할 때 참조한다. /dev/lirc/ 디렉터리에 위치하는 iCORE-SDP의 기존 LEG 리모컨 용 lircrc를 백업한 후 새로 작성한다.

```
tea@planx:~ $ sudo mv /etc/lirc/lircrc    /etc/lirc/lge_lircrc
```

```
tea@planx:~ $ sudo mv /etc/lirc/car_lircrc  /etc/lirc/lircrc
```

<... 각 항목은 begin ~ end로 묶이고 주요 요소는 button, prog, config임 ...>

IR Receiver

- LIRC(Linux Infrared Remote Control)

LIRC 서비스를 재 시작한 후 새로 만든 키 맵을 테스트한다.

```
tea@planx:~ $ sudo service lirc restart
```

```
tea@planx:~ $ irw
```

<수신기 키를 누를 때마다 코드와 연속 횟수, 이름, 키 맵 이름 순으로 출력>

```
0000000000ffa25d 00 KEY_CHANNELDOWN custom.conf
```

```
0000000000ffa25d 01 KEY_CHANNELDOWN custom.conf
```

```
0000000000ffa25d 02 KEY_CHANNELDOWN custom.conf
```

```
0000000000ffe21d 00 KEY_CHANNELUP custom.conf
```

```
0000000000ffe21d 01 KEY_CHANNELUP custom.conf
```

...

<Ctrl> + <C>

IR Receiver

```
import lirc
import time

sockid = lirc.init("Peri0", "/etc/lirc/lircrc", blocking=False)

while True:
    try:
        button = lirc.nextcode()

        if len(button) == 0:
            continue

        print(button[0])

    except KeyboardInterrupt:
        lirc.deinit()
        break
```

Sensor Programming

센서 프로그래밍

PIR(Passive Infrared)
Sensor



RaspberryPi



RASPBIAN

PIR Sensor

- PIR (Passive infrared) Sensor

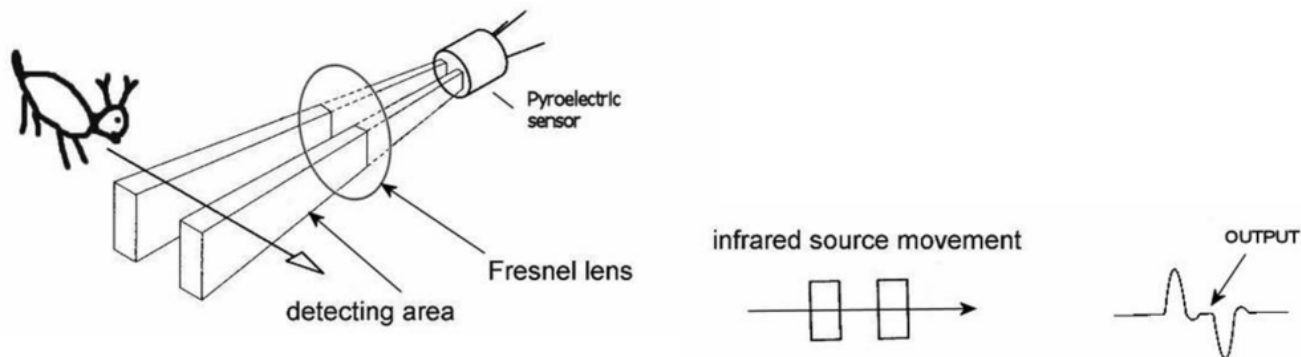
절대 온도를 초과하는 모든 물체는 사람 눈에는 보이지 않는 적외선 파장과 함께 열에너지를 방출하는데 PIR는 물체에서 방출되거나 반사되는 적외선 파장의 움직임에 반응하는 센서이다.



PIR Sensor

- 움직임 감지

PIR Sensor는 프레넬 렌즈 Fresnel Lens 와 적외선 검출기(U20), 판별 회로 부로 구성되는데 프레넬 렌즈가 적외선만 모아 적외선 검출기로 전달하면 적외선 검출기는 수평 면을 기준으로 첫 번째 감지 점과 두 번째 감지 점을 구분해 판별 회로로 전달한다. 판별 회로는 두 입력 값의 변화를 비교한 후 디지털 결과를 출력한다.

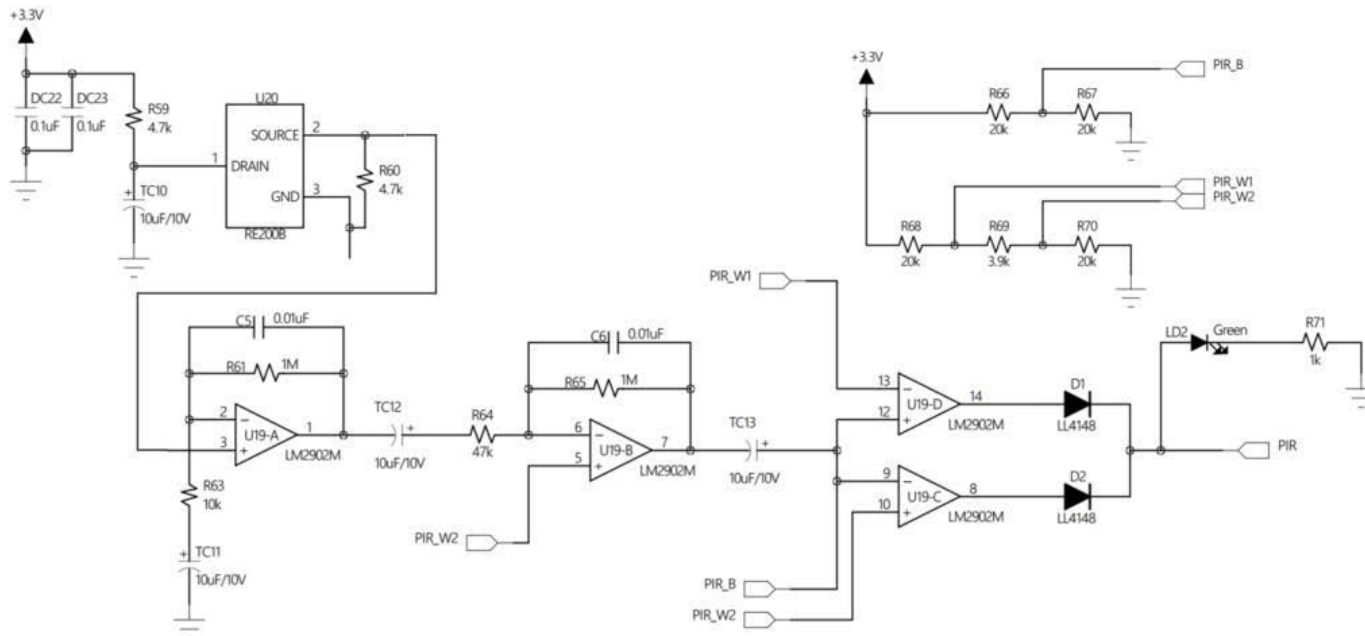


PIR Sensor

- 회로 구성

Peri0의 적외선 검출기 소스는 다운Pull Down 상태이므로 적외선의 움직임을 감지하면 미세 전류를 오른쪽 증폭기(U19-A, U18-B)로 보낸다. 첫 번째 증폭기는 입력 신호와 피드백을 통해 노이즈를 제거한 증폭 신호 만들어 두 번째 증폭기로 보내고, 두 번째 증폭기는 입력 신호와 기준 전압(PIR_W2)을 통해 노이즈를 제거한 증폭 신호를 만들어 비교기로(U19-C, U19-D)로 보낸다

비교기는 2개의 기준 전압(PIR_W1, PIR_W2)과 입력 신호의 비교 결과를 다이오드(D1, D2)로 보내면 다이오드를 통해 움직임이 감지될 때마다 최종 결과인 High 값이 출력 핀(PIR)으로 보내진다. 출력 핀은 풀 다운 상태이므로 움직임이 감지되지 않으면 LOW, 감지되면 HIGH가 된다.



PIR Sensor

```
import RPi.GPIO as GPIO
import time

pir = 24
GPIO.setmode(GPIO.BCM)
GPIO.setup(pir, GPIO.IN)

def loop():
    cnt = 0
    while True:
        if (GPIO.input(pir) == True):
            print ('detected %d' %cnt)
            cnt+=1
            time.sleep(0.1)

try:
    loop()

except KeyboardInterrupt:
    pass

finally:
    GPIO.cleanup()
```

Sensor Programming

센서 프로그래밍

I2C



RaspberryPi



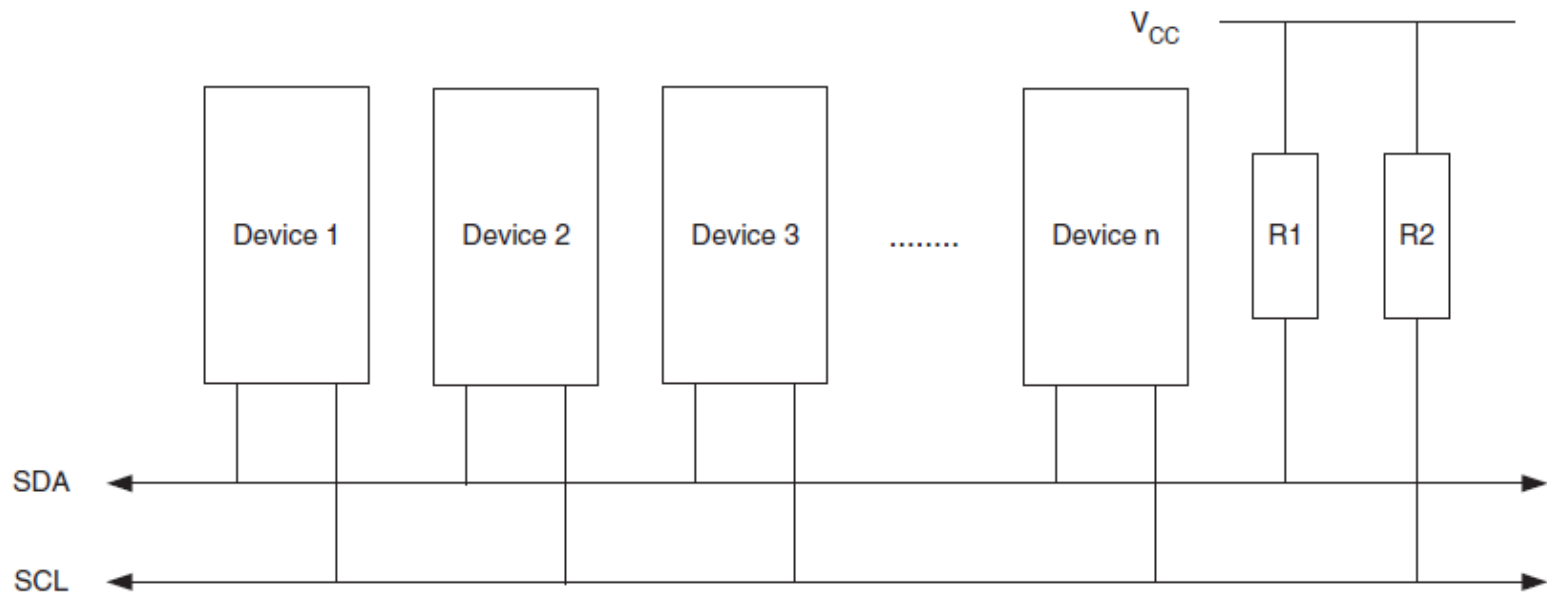
RASPBIAN

I2C

- I2C (Inter-IC Communication)
 - I2C는 I-Square-C 라고 부르며 필립스에서 제안한 통신 방식
 - 두 개의 신호를 사양하는 양방향 직렬 통신 방식
 - 하나의 Master와 다수의 Slave로 구성
 - 전송 속도
 - 표준 모드 : 100kbps
 - 고속 모드 : 400kbps
 - 초고속 모드 : 3.4Mbps
 - 하위 호환성 유지. 프로세서 뿐만 아니라 연결된 장치에 따라 속도 상이할 수 있으므로 datasheet 확인 필요.
 - 각 장치마다 7-bits 또는 10-bits의 주소 사용.
 - 일반적으로 7-bits 주소 사용.
 - 하나의 I2C 버스에 연결되는 개별 장치의 식별 용도로 활용

I2C

- I2C (Inter-IC Communication) (continued)
 - SDA(Serial Data line), SCL(Serial Clock Line) 두 개의 신호 사용
 - 양방향 데이터 전송을 지원하므로 SDA, SCL은 open drain 으로 반드시 외부에 pull-up 저항이 필요.(SDA는 read, write에 따라 핀의 상태가 입력 또는 출력으로 동작이 변함)



I2C

- I2C (Inter-IC Communication) (continued)
 - SDA, SCL 두 개의 신호에 다수의 장치들이 부착되어 Master의 제어에 따라 데이터 송/수신.
 - 디바이스 주소
 - I2C 통신은 Master, Slave로 구성되며 동일한 Line에 연결된 다수의 장치를 식별하기 위한 목적으로 사용.
 - 7비트 또는 10비트의 주소 체계를 사용할 수 있음.
 - 대부분 7비트 주소 체계를 사용하며 이 경우 하나의 Master에 연결 가능한 장치는 이론적으로 128개가 됨.
 - 디바이스 주소는 IC 제조사에서 고정시켜 놓거나 외부 핀에 의해 설정 가능. 반드시 해당 IC의 datasheet 참조.

I2C

- I2C (Inter-IC Communication) (continued)
 - SCL
 - Serial Clock Line으로 Master에서 Slave로 전송되는 단방향 신호
 - SDA
 - Serial Data line으로 write 동작에서는 Master에서 Slave로, read 동작에서는 Slave에서 Master로 전송되는 양방향 신호.
 - Read 동작은 I2C의 표준 포맷에 따라 SDA의 전송 방향은 Master → Slave, Slave → Master로 각각 변화.

I2C

- I2C 포트 활성화

```
# sudo raspi-config
```

- 'Advanced Options ' 을 선택
- I2C 옵션 선택 후 Finish

- I2C 유틸리티 및 라이브러리 설치

```
# sudo apt-get install python-smbus i2c-tools
```

- 커널 모듈 설정

```
# sudo vi /etc/modules
```

추가 후 저장

```
i2c-bcm2708  
i2c-dev
```

- 재부팅

```
# sudo reboot
```

I2C

- I2C API
 - long write_quick(int addr)
 - Send only the read / write bit
 - long read_byte(int addr)
 - Read a single byte from a device, without specifying a device register.
 - long write_byte(int addr,char val)
 - Send a single byte to a device
 - long read_byte_data(int addr,char cmd)
 - Read Byte Data transaction.
 - long write_byte_data(int addr,char cmd,char val)
 - Write Byte Data transaction.
 - long read_word_data(int addr,char cmd)
 - Read Word Data transaction.
 - long write_word_data(int addr,char cmd,int val)
 - Write Word Data transaction.

I2C

- I2C API (continued)
 - `long process_call(int addr,char cmd,int val)`
 - Process Call transaction.
 - `long[] read_block_data(int addr,char cmd)`
 - Read Block Data transaction.
 - `write_block_data(int addr,char cmd,long vals[])`
 - Write up to 32 bytes to a device. This function adds an initial byte indicating the length of the vals array before the vals array. Use `write_i2c_block_data` instead!
 - `long[] block_process_call(int addr,char cmd,long vals[])`
 - Block Process Call transaction.

Sensor Programming

센서 프로그래밍

I2C light



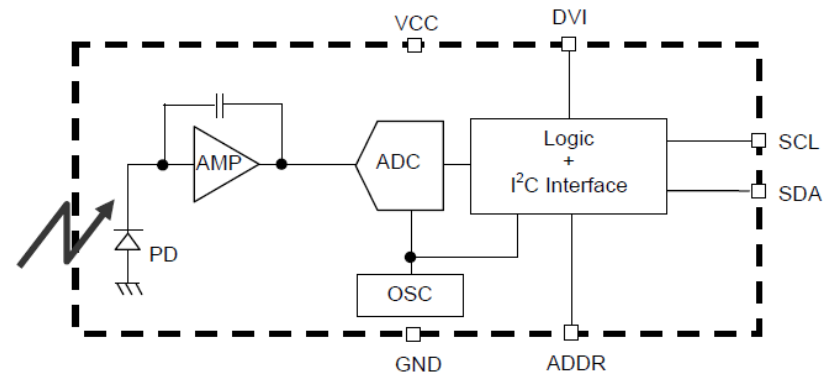
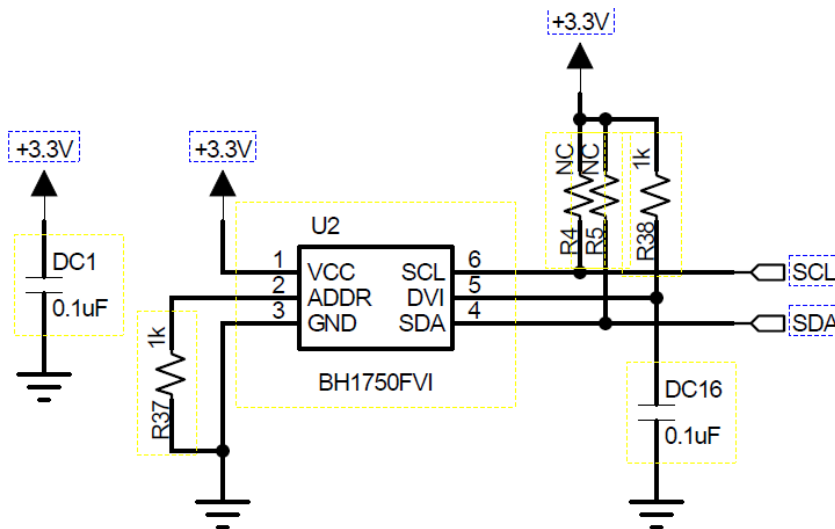
RaspberryPi



RASPBIAN

I2C light

- BH1750FVI I2C 디지털 조도센서
 - ROHM사의 BH1750FVI 칩을 사용한 디지털 조도센서 모듈
 - 조도 측정 범위는 0~65535 룩스(lx)
 - 내부에는 16비트 A/D가 내장되어 직접 디지털 출력
 - 디지털 출력 방식은 I2C 표준



I2C light

Instruction	Opecode	Comments
Power Down	0000_0000	No active state.
Power On	0000_0001	Waiting for measurement command.
Reset	0000_0111	Reset Data register value. Reset command is not acceptable in Power Down mode.
Continuously H-Resolution Mode	0001_0000	Start measurement at 1lx resolution. Measurement Time is typically 120ms.
Continuously H-Resolution Mode2	0001_0001	Start measurement at 0.5lx resolution. Measurement Time is typically 120ms.
Continuously L-Resolution Mode	0001_0011	Start measurement at 4lx resolution. Measurement Time is typically 16ms.
One Time H-Resolution Mode	0010_0000	Start measurement at 1lx resolution. Measurement Time is typically 120ms. It is automatically set to Power Down mode after measurement.
One Time H-Resolution Mode2	0010_0001	Start measurement at 0.5lx resolution. Measurement Time is typically 120ms. It is automatically set to Power Down mode after measurement.
One Time L-Resolution Mode	0010_0011	Start measurement at 4lx resolution. Measurement Time is typically 16ms. It is automatically set to Power Down mode after measurement.
Change Measurement time (High bit)	01000_MT[7,6,5]	Change measurement time. ※ Please refer "adjust measurement result for influence of optical window."
Change Measurement time (Low bit)	011_MT[4,3,2,1,0]	Change measurement time. ※ Please refer "adjust measurement result for influence of optical window."

※ Don't input the other opecode.

I2C light

- I2C light 실습

```
import smbus2 as smbus
import time

bus = smbus.SMBus(1)
addr = 0x23
reset = 0x07
con_hr_mode = 0x10

data1 = 0
data2 = 0
val = 0
light_val = 0
```


I2C light

- I2C light 실습 (Continued)
 - import smbus
 - smbus 모듈 임포트
 - bus = smbus.SMBus(1)
 - I2C 채널 설정 및 초기화
 - 0 = /dev/i2c-0 (port I2C0),
 - 1 = /dev/i2c-1 (port I2C1)
 - addr = 0x23
 - slave addr 저장
 - reset = 0x07
 - reset command 저장
 - con_hr_mode = 0x10
 - hr mode 저장
 - 1lx resolution으로 설정

I2C light

- I2C light 실습 (continued)

```
try:
```

```
    bus.write_byte(addr, reset)
```

```
    time.sleep(0.05)
```

```
while True:
```

```
    bus.write_byte(addr, con_hr_mode)
```

```
    time.sleep(0.2)
```

```
    data1 = bus.read_byte(addr)
```

```
    data2 = bus.read_byte(addr)
```

```
    val = (data1 << 8) | data2
```

```
    light_val = val / 1.2
```

```
    print 'light_val = %.2f' % light_val
```

```
    time.sleep(1)
```

I2C light

- I2C light 실습 (continued)

```
except KeyboardInterrupt:  
    # do not anything  
    pass  
finally:  
    pass
```

- bus.write_byte(addr, reset)
 - Reset command 실행
- time.sleep(0.05)
 - Reset timing 지연
- data1 = bus.read_byte(addr)
 - 상위 바이트 읽음
- data2 = bus.read_byte(addr)
 - 하위 바이트 읽음

Sensor Programming

센서 프로그래밍

I2C TEMP / HUMI



RaspberryPi



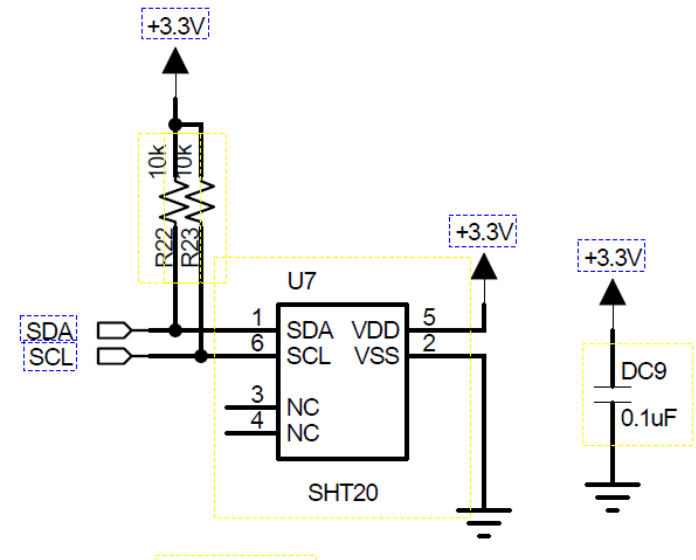
RASPBIAN

I2C TEMP/HUMI

온/습도 센서

SHT20 by Sensirion

- IIC Address : 0x40 (7-bit)
 - Read_address : 0x81
 - Write_address : 0x80
- Command Register



Command	Comment	Code
Trigger T measurement	hold master	1110'0011
Trigger RH measurement	hold master	1110'0101
Trigger T measurement	no hold master	1111'0011
Trigger RH measurement	no hold master	1111'0101
Write user register		1110'0110
Read user register		1110'0111
Soft reset		1111'1110



I2C TEMP/HUMI

Datasheet 참고

- Conversion of Signal Output
 - Default resolution is set to 12 bit relative humidity and 14 bit temperature reading.
 - Relative Humidity Conversion
- Temperature Conversion

$$T = -46.85 + 175.72 \cdot \frac{S_T}{2^{16}}$$

I2C TEMP/HUMI

- I2C TEMP/HUMI 실습

```
import smbus2 as smbus
```

```
import time
```

```
bus = smbus.SMBus(1)
```

```
addr = 0x40
```

```
cmd_temp = 0xf3
```

```
cmd_humi = 0xf5
```

```
soft_reset = 0xfe
```

```
temp = 0.0
```

```
humi = 0.0
```

```
val = 0
```

```
data = [0, 0]
```

I2C TEMP/HUMI

- I2C TEMP/HUMI 실습 (Continued)

try:

```
bus.write_byte(addr, soft_reset)
```

```
time.sleep(0.05)
```

while True:

```
# temperature
```

```
bus.write_byte(addr, cmd_temp)
```

```
time.sleep(0.260)
```

```
for i in range(0,2,1):
```

```
    data[i] = bus.read_byte(addr)
```

```
val = data[0] << 8 | data[1]
```

```
temp = -46.85+175.72/65536*val
```

$$T = -46.85 + 175.72 \cdot \frac{S_T}{2^{16}}$$

I2C TEMP/HUMI

- I2C TEMP/HUMI 실습 (Continued)

```
# humidity
```

```
bus.write_byte(addr, cmd_humi)
```

```
time.sleep(0.260)
```

```
for i in range(0,2,1):
```

```
    data[i] = bus.read_byte(addr)
```

```
val = data[0] << 8 | data[1]
```

```
humi = -6.0+125.0/65536*val;
```

$$RH = -6 + 125 \cdot \frac{S_{RH}}{2^{16}}$$

```
print 'temp : %.2f, humi : %.2f' %(temp, humi)
```

```
time.sleep(1)
```

```
except KeyboardInterrupt:
```

```
    pass
```

Sensor Programming

센서 프로그래밍

I2C FND



RaspberryPi



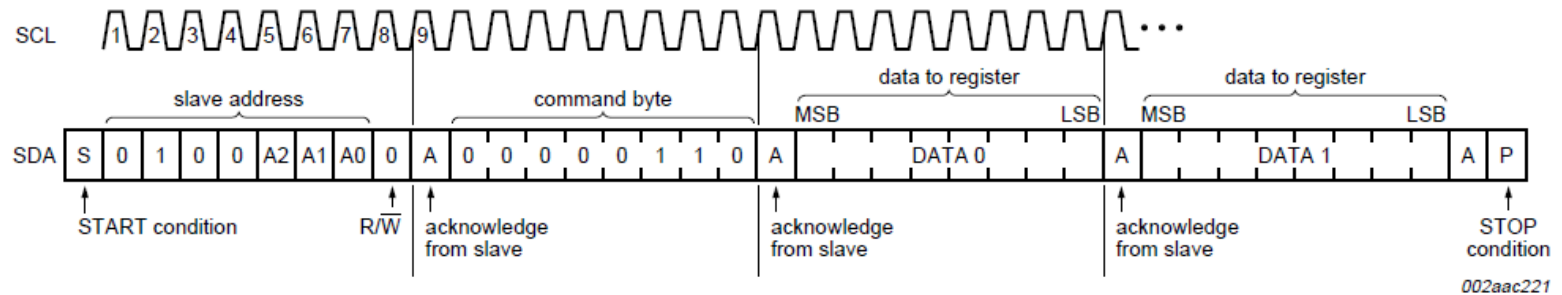
RASPBIAN

I2C FND

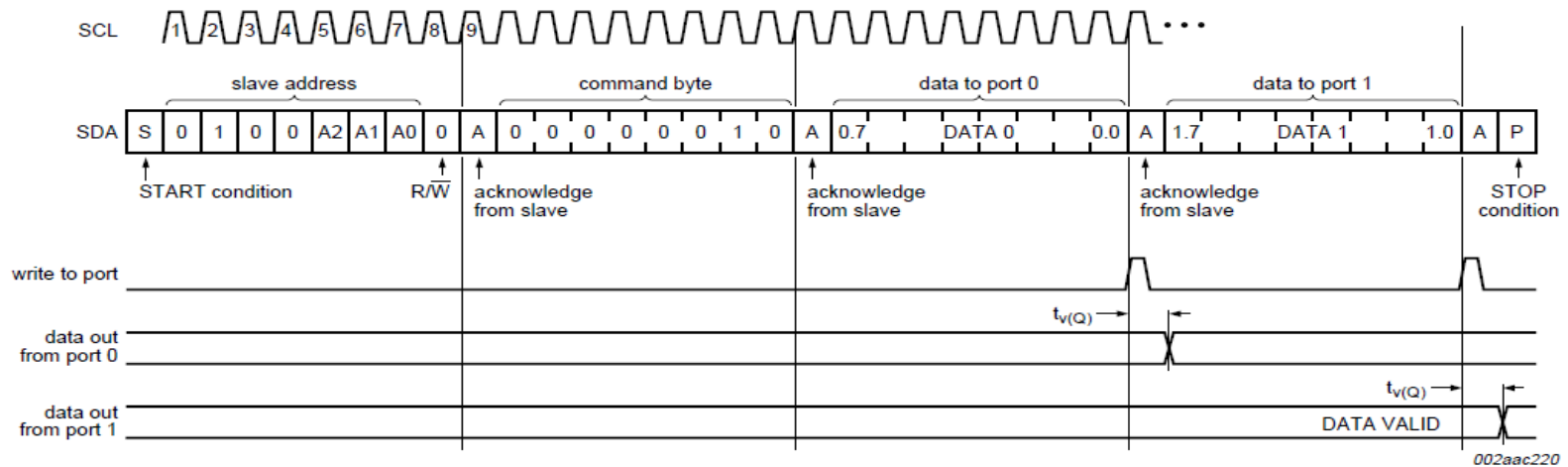
- **FND (Flexible Numeric Display)**
 - 7개의 LED를 배열하여 0 ~ 9의 10진 수를 표시하도록 구현한 장치
 - Dot를 포함하여 8개의 직사각형 LED를 10진수를 표시할 수 있는 모양으로 배치하여 특정 위치의 LED를 켜므로 10진수 1자리를 표현
 - Dot를 제외하고 7개의 LED로 수를 표현하므로 7-segment라 부르기도 함
 - FND의 개별 요소 하나씩은 LED와 동일한 구조이나 직사각형의 모양으로 구현됨
 - LED와 동일한 제어 방법 사용. 8개의 출력을 하나로 연관하여 생각해야 의미가 있음

I2C FND

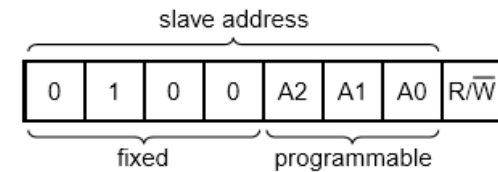
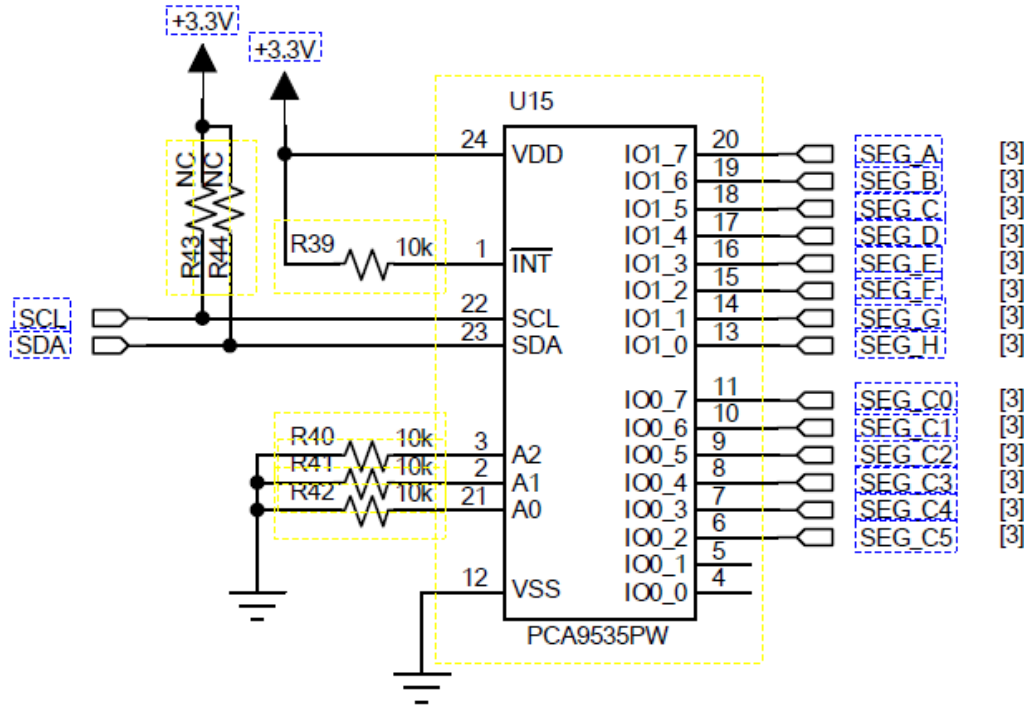
- Output port 0, 1 설정을 위한 command byte는 0x06



- Output port 0, 1로 데이터를 쓰기 위한 command byte는 0x02



I2C FND



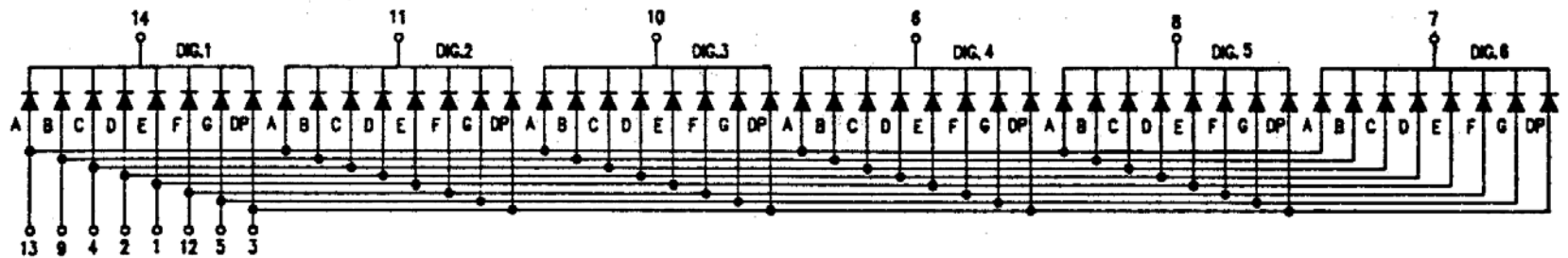
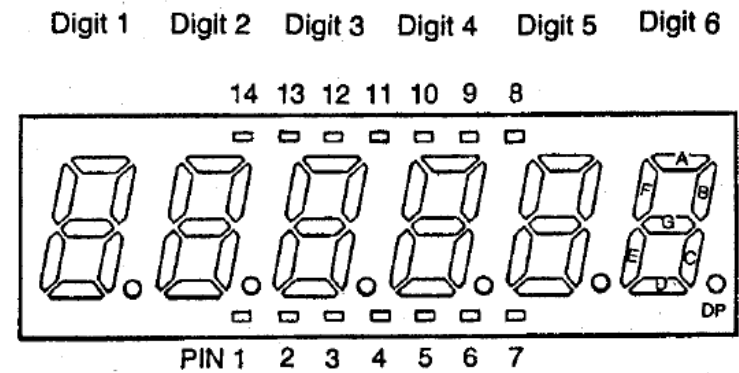
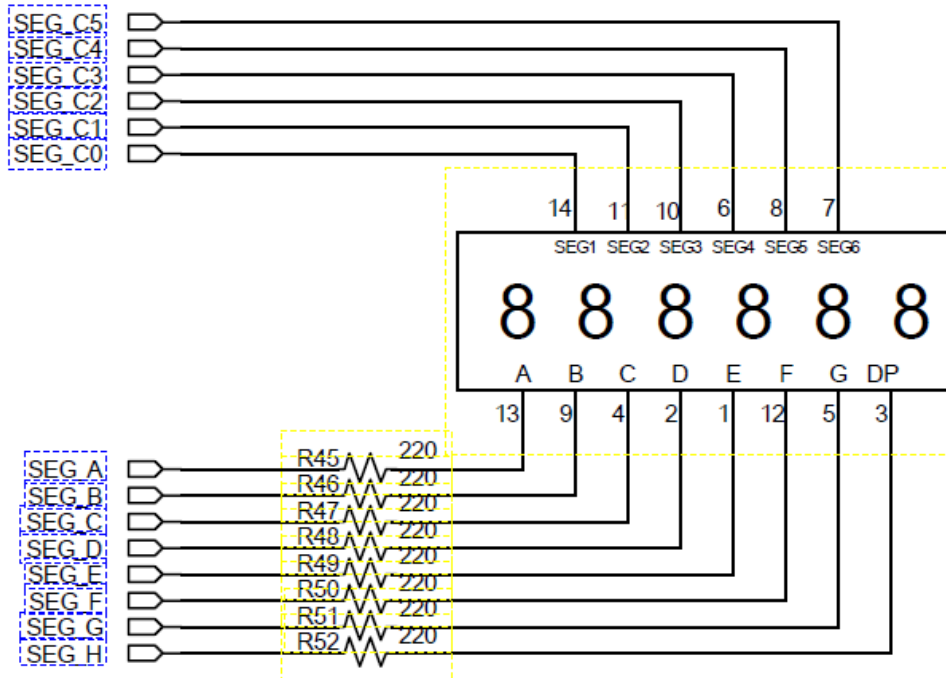
A0, A1, A2 pin

- 7비트 slave addr 변경 가능 핀, 현재 회로의 slave addr은 ??

INT pin

- IO port의 변경이 발생할 경우 내부 로직이 동작하도록 함 (low power 지원)
- Open drain이므로 pull up 저항 연결 (low active)

I2C FND



I2C FND

- I2C FND 실습

```
import smbus2 as smbus
import time

bus = smbus.SMBus(1)
addr = 0x20
config_port = 0x06
out_port = 0x02
#           0   1   2   3   4   5   6   7   8   9
data = (0xFC,0x60,0xDA,0xF2,0x66,0xB6,0x3E,0xE0,0xFE,0xF6)
#         seg1, seg2, seg3, seg4, seg5, seg6
digit = (0x7F, 0xBF, 0xDF, 0xEF, 0xF7, 0xFB)

out_disp=0
```

I2C FND

- I2C FND 실습 (continued)

```
try:
    bus.write_word_data(addr, config_port, 0x0000)

    # seg1 부터 seg6까지, 0~9까지 반복하면서 출력
    for i in range (0,6,1):
        for j in range (0,10,1):
            out_disp = data[j] << 8 | digit[i]
            bus.write_word_data(addr, out_port, out_disp)
            time.sleep(0.1)

except KeyboardInterrupt:
    pass
```


Sensor Programming

센서 프로그래밍

SPI

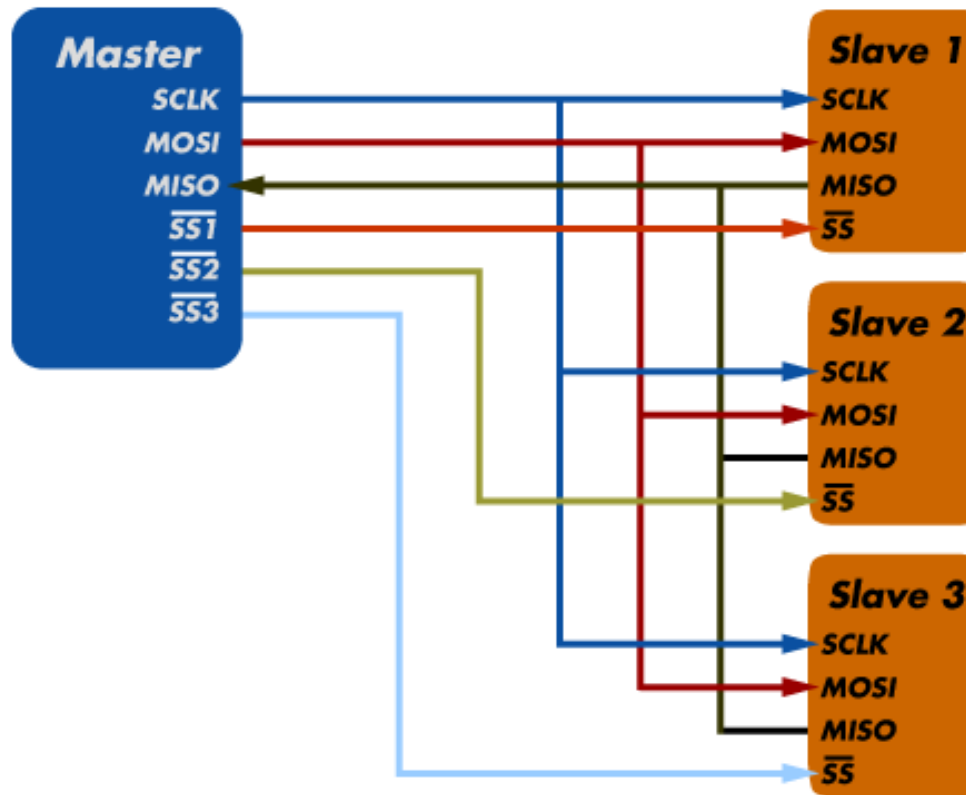


RaspberryPi



RASPBIAN

SPI



SPI

- SPI (Serial Peripheral Interface Bus)
 - 아키텍처 전이중 통신 모드로 동작.
 - 모토로라 아키텍처에 이름을 딴 동기화 직렬 데이터 연결 표준.
 - 장치들은 마스터 슬레이브 모드로 통신하며 마스터 장치는 데이터 프레임을 초기화한다.
 - 여러 슬레이브 장치들은 개별 슬레이브 셀렉트 (칩 셀렉트) 라인과 함께 동작할 수 있다.
- SPI 버스는 4가지 논리 신호를 지정한다
 - SCLK: 직렬 클럭 (마스터로부터의 출력)
 - MOSI; SIMO: 마스터 출력, 슬레이브 입력 (마스터로부터의 출력)
 - MISO; SOMI: 마스터 입력, 슬레이브 출력 (슬레이브로부터의 출력)
 - SS: 슬레이브 셀렉트 (active low, 마스터로부터의 출력).

- 장점
 - 완전한 전이중 통신
 - 전송되는 비트에 대한 완전한 프로토콜 유연성
 - 전송기가 필요하지 않음
 - 매우 단순한 하드웨어 인터페이스 처리
 - IC 패키지에 4개의 핀만 사용하며 이는 병렬 인터페이스에 비해 수가 적은 것이다.
- 단점
 - 하드웨어 슬레이브 인식이 없음
 - 슬레이브에 의한 하드웨어 흐름 제어가 없음
 - 오류 검사 프로토콜이 정의되어 있지 않음
 - 일반적으로 노이즈 스파이크에 영향을 받는 경향이 있음
 - RS-232, RS-485, CAN 버스보다 비교적 더 짧은 거리에서 동작
 - 하나의 마스터 장치만 지원

SPI

- SPI blacklist 제거

```
# sudo nano /etc/modprobe.d/raspi-blacklist.conf
```

```
# blacklist spi and i2c by default (many users don't need them)

#blacklist spi-bcm2708
#blacklist i2c-bcm2708
```

저장 : control-X 누른 후 Y 누름

- 재부팅

```
# sudo reboot
```

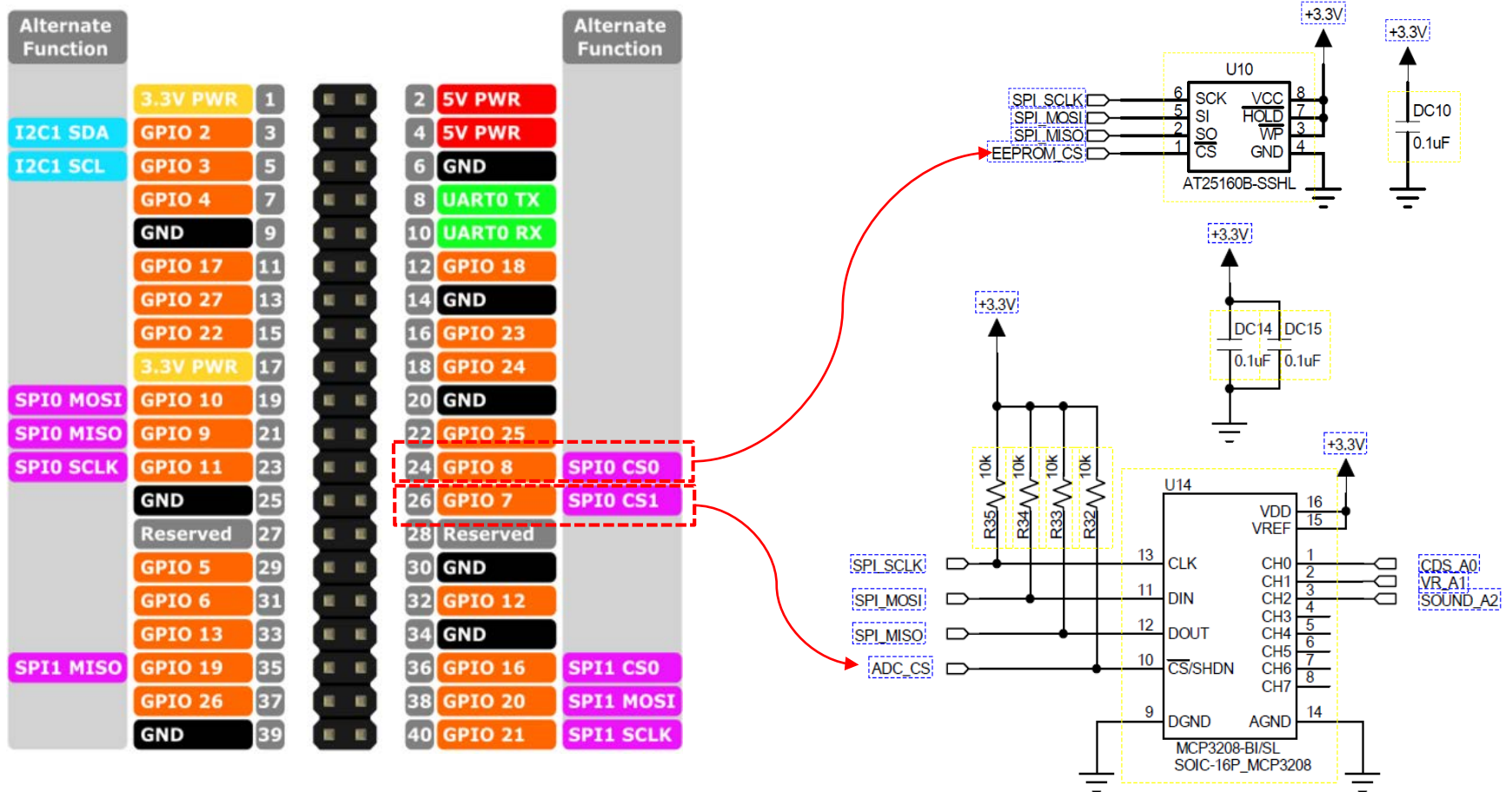
- SPI 활성화 확인

```
pi@pi2 ~ $ ls -l /dev/spidev*
crw-rw---T 1 root spi 153, 0 Jan  1  1970 /dev/spidev0.0
crw-rw---T 1 root spi 153, 1 Jan  1  1970 /dev/spidev0.1
```

SPI

- `spi = spidev.SpiDev()`
 - SPI 객체 생성
- `spi.open(bus, device)`
 - `bus` ; spi port,
 - `device` : chip select number
- `spi.close`
 - 종료 전 port close
- `spi.max_speed_hz`
 - HZ로 bus 최대 버스 속도 설정
- `spi.readbytes(len)`
 - Len 만큼의 바이트를 디바이스에서 읽어 list로 리턴함
- `spi.writebytes([value,value1,...])`
 - List에 저장된 값들을 디바이스에게 전송함
- `spi.xfer2 ([value, value1, ...])`
 - 리스트의 개수만큼 디바이스에 전송하고 **전송한 개수**만큼 list로 리턴함

SPI 마스터 연결



- SPI0, SPI1 : 2개의 채널 지원, CS0 : 칩 셀렉터 0번, CS1 : 칩 셀렉터 1번
- 현재 스펙에서 지원 가능한 SPI 슬레이브 디바이스 개수는? 3개

Sensor Programming

센서 프로그래밍

SPI EEPROM



RaspberryPi



RASPBIAN

SPI EEPROM

```
import spidev
import time

spi = spidev.SpiDev()
spi.open(0, 0) # SPI 채널 0번 , cs0 (칩 셀렉터 0번)
spi.max_speed_hz = 1000000 # 1Mhz

WREN = 0x06
WRITE = 0x02
READ = 0x03
WRDI = 0x04
RDSR = 0x05
WRSR = 0x01

dummy = 0
max_size = 15
```

SPI EEPROM

```
try:
    spi.writebytes( [WREN] )
    time.sleep(0.001)

    buff = [WRITE, 0x00, 0x11]
    for i in range(max_size):
        buff.append(i)
    spi.writebytes(buff)
    time.sleep(0.001)

    spi.writebytes ( [WRDI] )
    time.sleep(0.001)
```

SPI EEPROM

```
#try: (continue)
    buff = [READ, 0x00, 0x11]
    for i in range(max_size):
        buff.append(dummy)

    read = spi.xfer2( buff )
    time.sleep(0.001)
    print read

except KeyboardInterrupt:
    pass

finally:
    spi.close()
```

Sensor Programming

센서 프로그래밍

SPI MPU3208

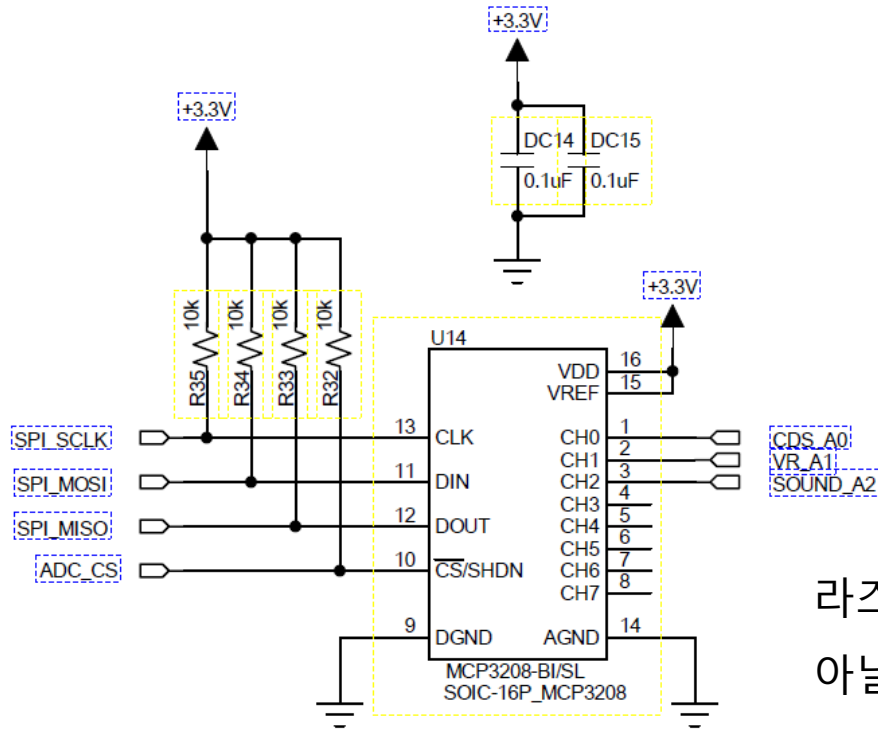


RaspberryPi



RASPBIAN

SPI MCP3208

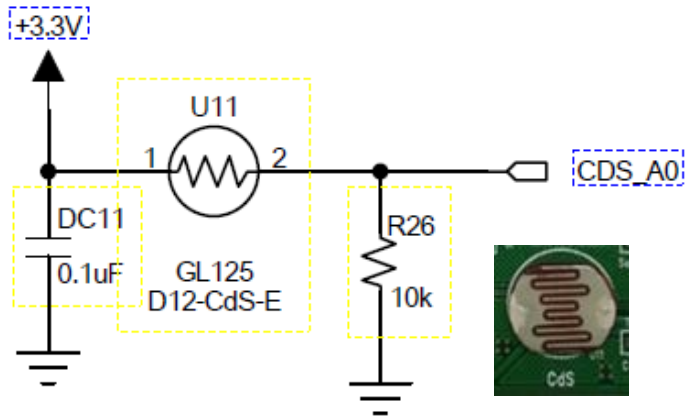


라즈베리파이는 칩 내부에 ADC 기능이 없으므로
아날로그 센서 디바이스를 직접 연결 불가

MCP3208

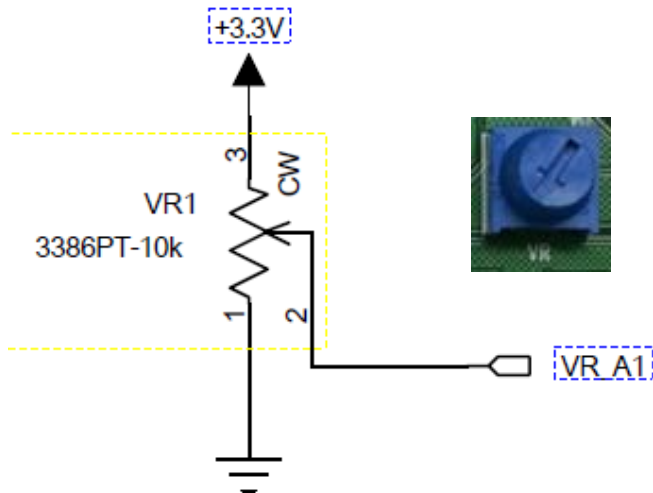
- 0v ~ 3.3v 의 센서 출력 값을 0 ~ 4095 (12bit) 의 디지털값으로 변환해주는 IC
- SPI 통신을 통해 각 채널에 연결된 디바이스의 센싱 값과 명령을 READ/WRITE 가능
- 최대 8-Channel 사용 가능, 현재 회로에서는 3개의 디바이스만 연결됨

SPI MPU3208



GL125

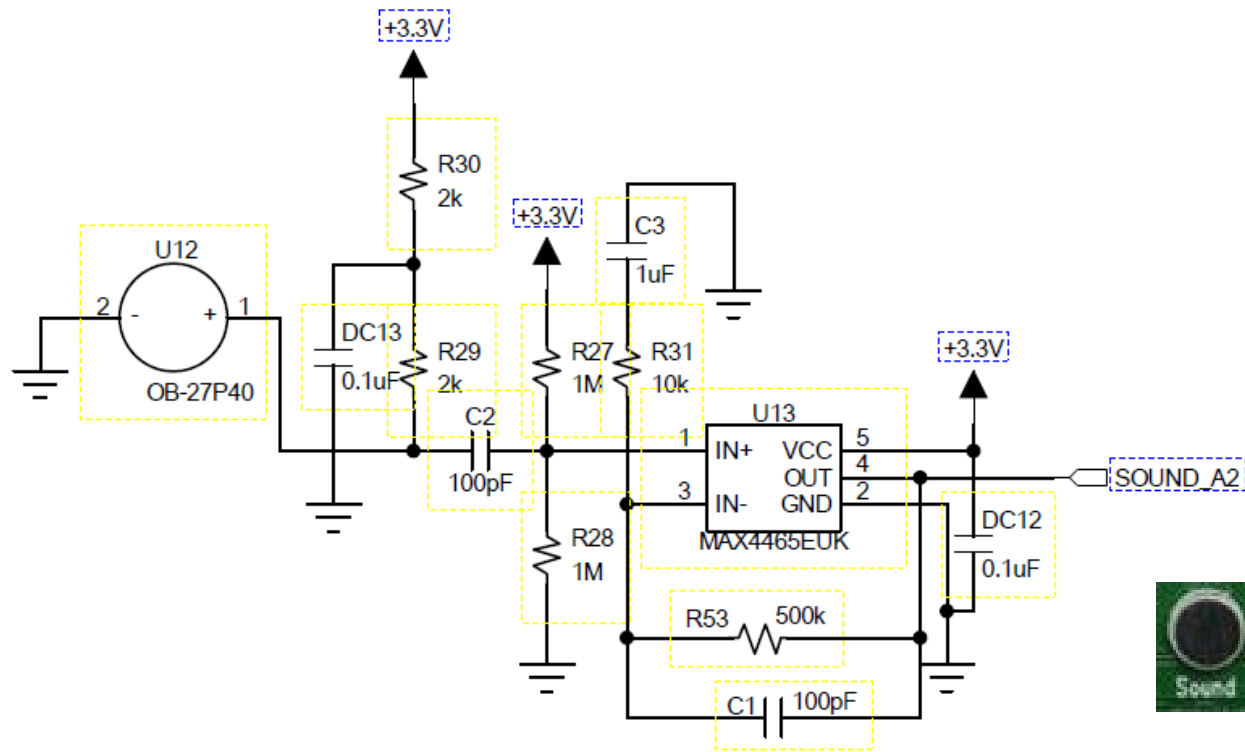
- Cadmium Sulfide (CdS)
- 빛에 따라 저항 값 변경
- 저항에 대한 전압을 CDS_A0로 (MCP3208 채널 0번) 출력



3386PT (가변저항)

- 조절 단자의 회전 각도에 따라 저항값 변화
- 저항에 대한 전압을 VR_A1로 (MCP3208 채널 1번) 출력

SPI MPU3208



OB-27P40

- 소리의 크기를 전압 신호로 변환해 주는 마이크

MAX4465EUK

- 마이크의 출력 신호를 adc에 입력 전압에 맞도록 증폭시키는 역할을 함

SPI MPU3208

```
import spidev
import time

spi = spidev.SpiDev()
spi.open(0,1) # SPI 채널 0번, CS1 : 칩 셀렉터 1번
adc_read =[0,0,0]

def analog_read(ch):
    # 12 bit
    r = spi.xfer2([0x6 | (ch & 0x7) >> 2, ((ch & 0x7) << 6),0 ])
    adcout= ((r[1] & 0xf) << 8) + r[2]

    # 10 bit
    #r= spi.xfer2([1, (8+channel)<<4, 0])
    #ret = ((r[1]&3) << 8) + r[2]
    return adcout
```


SPI MPU3208

```
try:
    while 1:
        adc_read[0] = analog_read(0)
        adc_read[1] = analog_read(1)
        adc_read[2] = analog_read(2)
        print("[csd vr sound]")
        print(adc_read)
        time.sleep(1)

finally:
    spi.close()
```