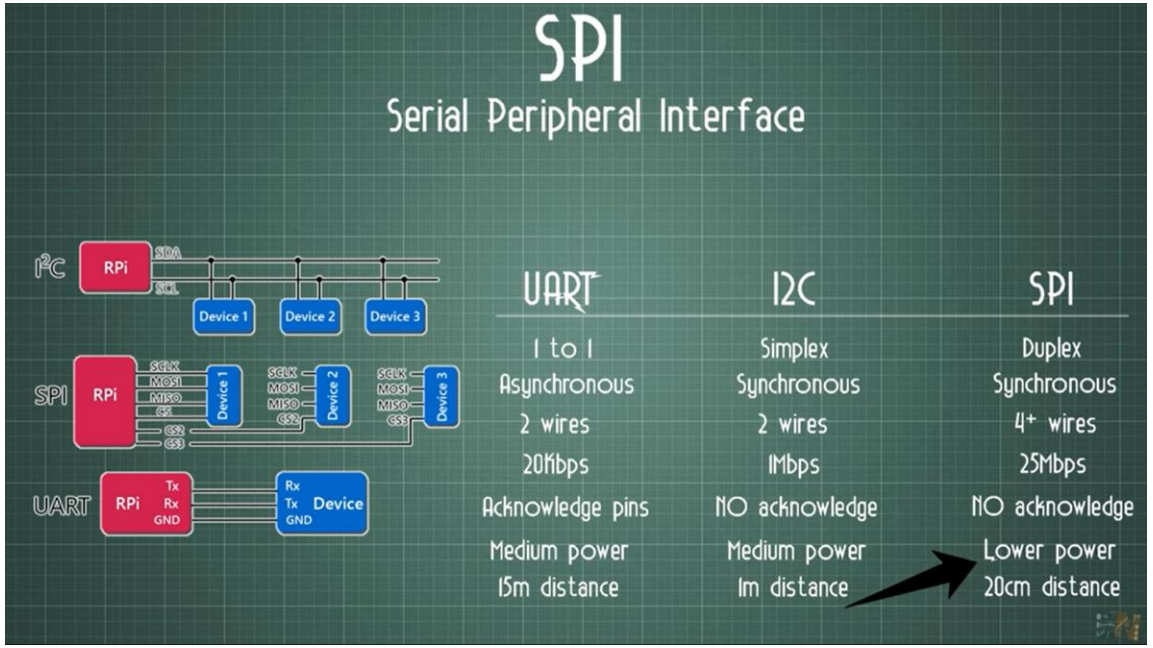
학습열기

- 다음 **I2C, SPI** 인터페이스에 대한 동영상을 시청하시기 바랍니다.
- https://www.youtube.com/watch?v=IyGwvGzrqp8



I2C와 SPI

- **I2C와 SPI**

I2C

- SDA 한 개의 Line으로 데이터 송수신을 하므로 통신 속도가 SPI 방식보다 상대적으로 느림
- 데이터 송수신을 동시에 할 수 없음

➡ 센서모듈을 추가 연결할 때, 전선 한 개를 버스선에 연결하면 됨

SPI

- MOSI와 MISO 두 개의 Line으로 데이터 송수신을 서로 분리하므로 I2C보다 통신속도가 빠름
- 데이터 송수신을 동시에 할 수 있음

➡ 센서모듈을 추가 연결할 때, I2C 보다는 조금 더 많은 연결이 필요함

I2C

- **I2C**

1. Inter Integrated Circuit 버스
2. 마이크로프로세서와 저속 주변장치 사이의 통신을 목적으로 함
3. Philips에서 개발한 규격
4. 2개의 Line만으로 통신하기 때문에 TWI(Two Wire Interface)라고도 함
5. 양방향 오픈 드레인 선인 SCL, SDA로 이루어져 있음
   - SCL: Serial Clock
   - SDA: Serial Data
6. Master-Slave 형태로 동작함

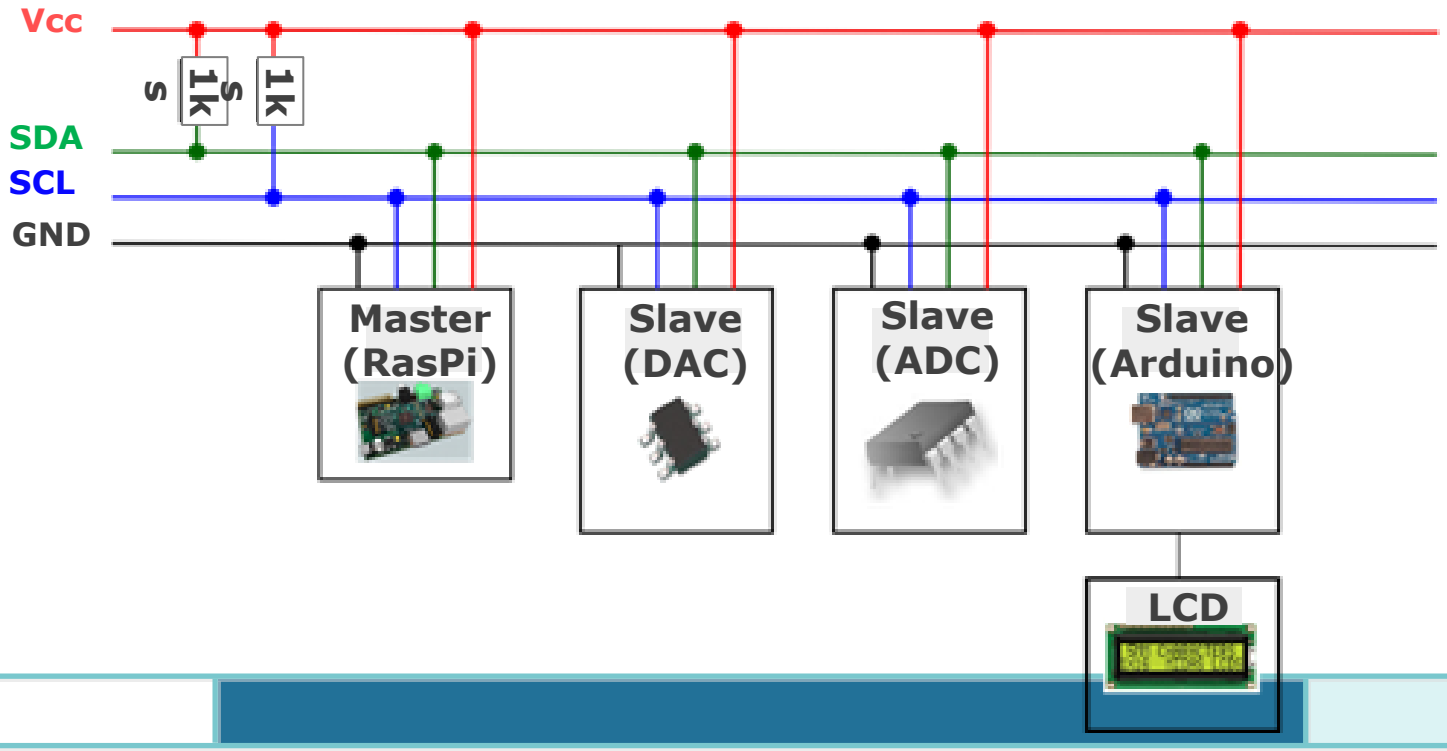
I2C

- **I2C**

7 속도면에서는 다른 방식에 비하여 현저히 느림

BUT 하드웨어적으로 간단한 구성으로 자주 활용됨

8 하나의 버스에 많은 수의 노드를 연결할 수 있다는 장점이 있음



I2C

- **SCL, SDA**

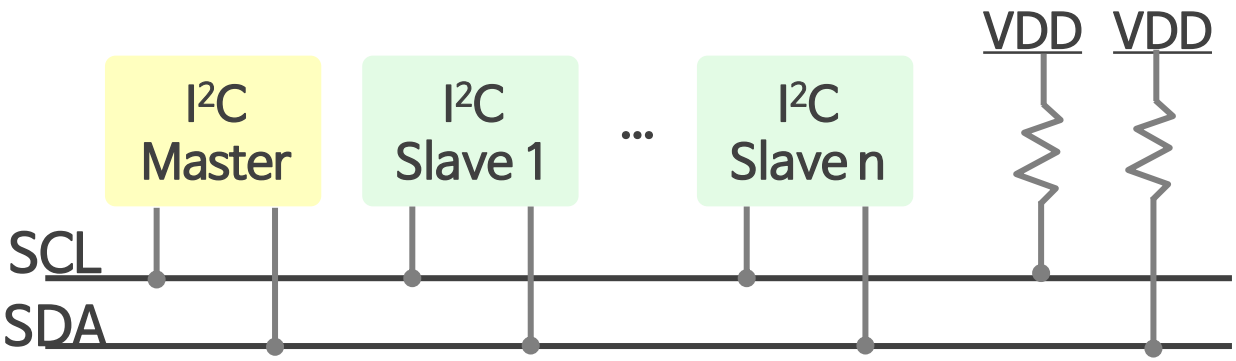
SCL

통신의 동기 (Synchronization)를 위한 클럭용 Line

SDA

데이터 통신을 위한 Line

Master는 SCL로 동기를 위한 클럭을 출력하고, Slave는 SCL로 출력되는 클럭에 맞추어 SDA를 통해 데이터를 출력하거나 입력함



I2C

- **SCL, SDA**

Master는 SCL로 동기를 위한 클럭을 출력함

Slave는 SCL로 출력되는 클럭에 맞추어 SDA를 통해
데이터를 출력하거나 입력 받음

https://eslectures.blog.me/

SDA는 한 선으로만 데이터를 주고 받기 때문에
I2C 버스는 반이중(Half Duplex) 통신만 가능함

SCL선과 SDA선은 모두 오픈 드레인이므로 두 선에는
각각 풀업 저항을 연결해 주어야 함

I2C

- **SCL, SDA**

> 모든 I2C Master와 Slave 장치들의 SCL, SDA는 서로 연결되고, 모든 장치들이 SCL, SDA를 각각 공유하므로 Master가 Slave를 개별적으로 지정하기 위한 방법이 있어야 함

  - Master는 주소로 원하는 Slave를 지정함
  - 주소의 길이는 7비트(bit)이므로 Master는 최대 127개의 슬레이브 장치들과 연결할 수 있음

※ 각 Slave 장치들의 주소는 모두 달라야 함

I2C

I2C 통신방식

1. 한 순간에는 오직 하나의 Master와 하나의 Slave 만이 통신할 수 있음

2. 모든 장치들의 SCL과 SDA는 각각 wired AND로 연결되어 있음

- 어느 한 장치라도 0을 출력하면 해당 신호의 상태는 논리 0이 됨
  - 만일 어떤 장치가 논리 0을 출력하면 다른 장치가 그 신호의 상태를 논리 1로 만들 방법이 없음

➡ 통신에 참여하지 않는 장치가 SCL이나 SDA로 0을 출력하면 Master가 정상적으로 통신할 수 없음

➡ I2C 버스에 연결되어 있지만 현재 통신에 참여하지 않고 있는 장치들은 모두 자신의 출력을 Floating 상태로 유지해야 함

I2C 통신방식

3. 통신이 진행되지 않는 상황에서 모든 장치의 출력은 Floating 상태이므로, SCL과 SDA의 상태는 모두 논리 1임

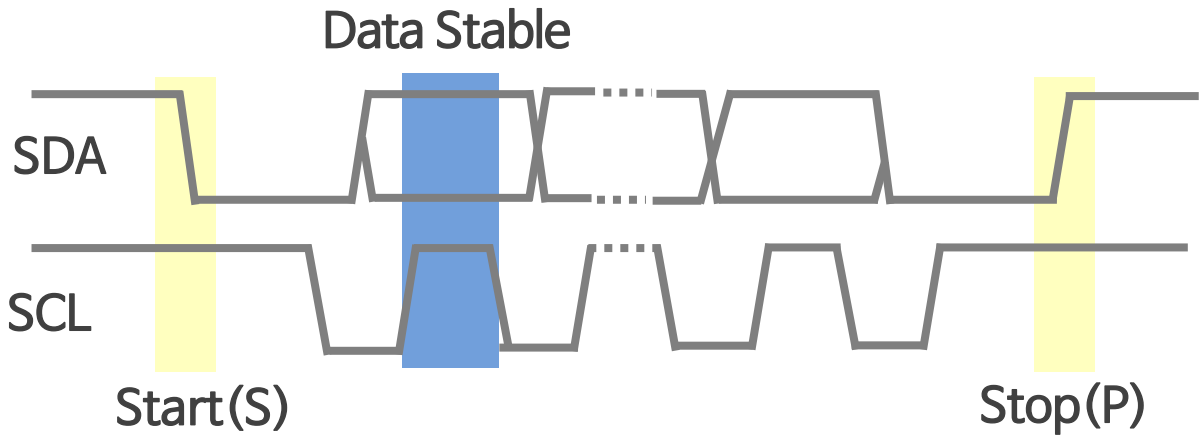
➡ 이 상황에서 I2C 버스의 사용을 원하는 Master는 SCL과 SDA로 시작 조건을 출력하여 버스 소유권을 주장하고 통신을 시작할 수 있음

4. 두 신호의 상태가 모두 논리 1이 아니라면 현재 다른 Master가 버스 소유권을 가지고 통신을 진행 중에 있다는 것을 뜻함

➡ Master가 버스 소유권을 반납할 때까지는 새로운 통신이 시작될 수 없음

I2C

- **SCL, SDA waveform**



[I2C 프로토콜의 시작 조건, 정지 조건, 데이터 안정구간]

Start Condition

- SCL이 1인 동안에 SDA가 1에서 0으로 바뀌는 상황
- 통신의 시작을 다른 장치에 알리는 효과

Stop Condition

- SCL이 1인 동안에 SDA가 0에서 1로 바뀌는 상황
- 버스 소유권 반납을 다른 장치에 알리는 효과

I2C

- **SCL, SDA waveform**

I2C 프로토콜에서 SCL이 0인 구간에서는 SDA의 상태변화가 허용되지만 SCL이 1인 구간에서는 SDA는 안정된 논리 상태를 유지해야 함

Master가 Slave로 데이터를 출력할 때 SCL이 0인 구간에서 SDA의 비트전환을 하여 SCL이 1인 구간에서는 SDA의 상태를 그대로 유지함

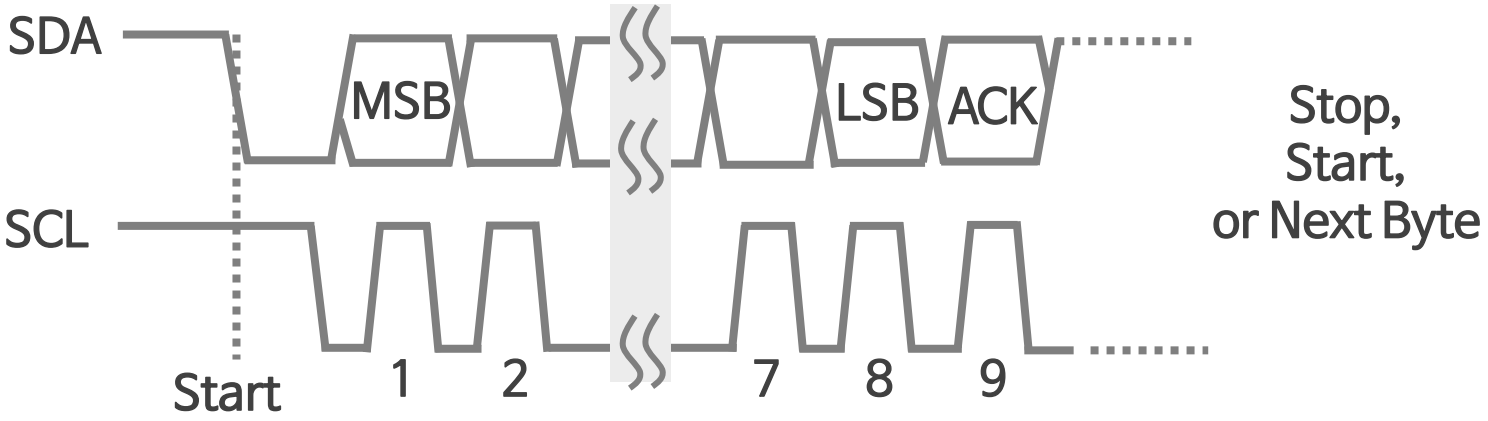
※Slave가 데이터를 출력하고 Master가 그 데이터를 읽을 때도 동일함

I2C

- **I2C protocol**

ACK를 포함한 9비트(Bit)가 I2C 규격에서 통신의 기본 단위임

Master는 시작 조건을 출력하면서 통신의 시작을 알리고, 시작 조건 이후부터는 SCL 상태가 0인 구간에서만 SDA의 논리 값이 바뀜



I2C

I2C protocol

Master가 SCL로 출력하는 클럭에 동기를 맞추어 SDA로는 데이터가 MSB부터 한 비트씩 출력됨

8비트 데이터가 8 클럭 사이클 동안 SDA로 출력되면 그 데이터를 수신한 쪽에서 9번째 클럭에 맞추어 그 8비트 데이터의 수신여부를 확인해주는 ACK를 보냄

- ACK 비트가 0이면 정상 수신
- ACK 비트가 1이면 비정상 수신
  - 값이 0인 ACK와 구분하기 위해서 값이 1인 NACK라 함

I2C

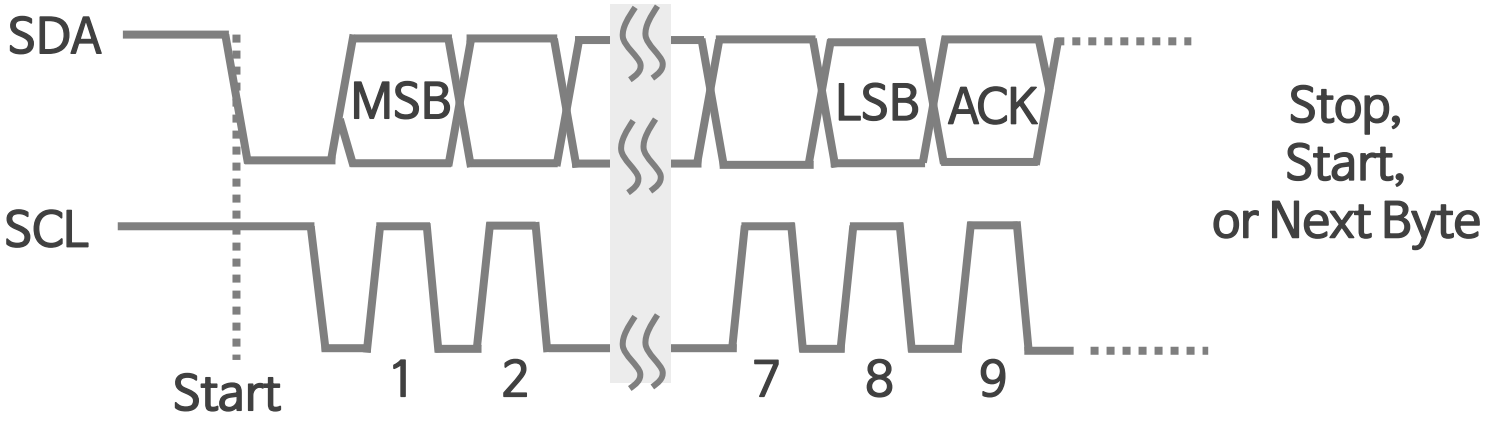
- **I2C** 데이터 전송

I2C에서는 데이터를 전달하기 위해 아래 형식을 따름

① Master가 Slave에 전송을 시작한다는 표현-Start(1bit)
② 데이터 목적지 주소-Address(7bit)
③ 전송 목적-R/'W(읽기 또는 쓰기) (1bit)
④ 전송데이터-Data(8bit)
⑤ Slave가 정상적으로 데이터를 수신했다는 응답-Ack(1bit)
⑥ 전송 종료-Stop(1bit)

I2C

- **I2C Protocol**



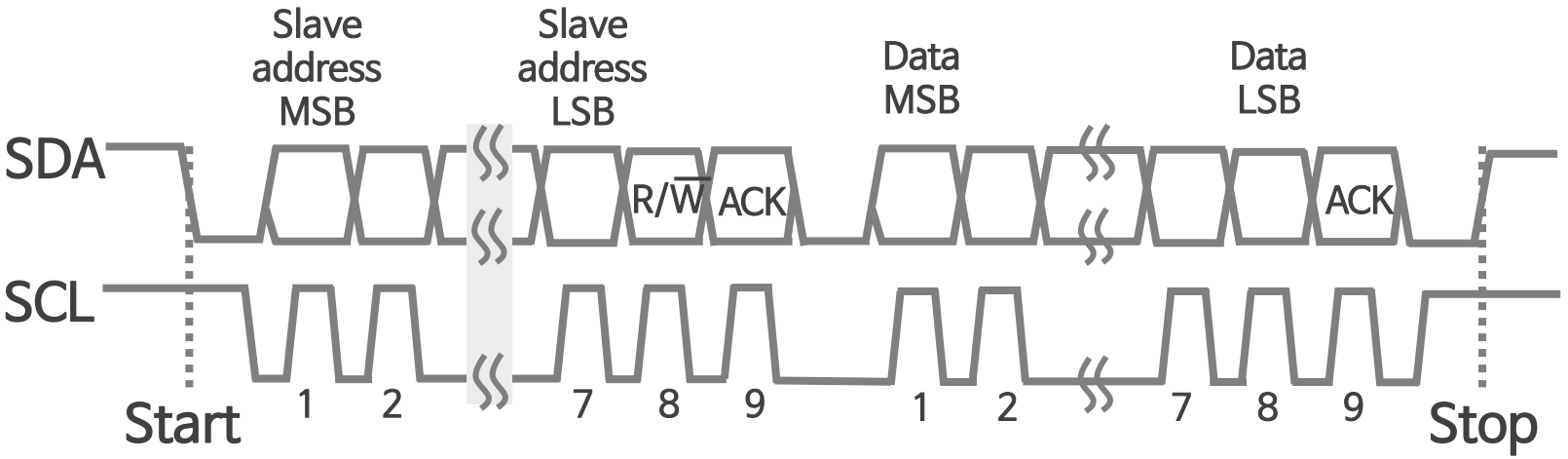
8비트 정보와 이어지는 ACK 비트의 전송이 끝난 다음에는 Master가 정지 조건을 출력하거나 시작 조건을 다시 출력할 수도 있고, 또한 다음 바이트 전송이 이어질 수도 있음

I2C

- I2C 주소 지정 형식

모든 I2C Slave 장치는 7비트의 고유 주소를 가지며, Master는 이 주소를 사용하여 상대 Slave 장치를 지정함

Master는 시작조건에 이어서 자신이 원하는 Slave의 7비트 주소를 출력함



I2C

- I2C 주소 지정 형식

버스에 연결된 모든 Slave는 SDA를 계속 감시하면서 Master가 출력한 주소가 자신의 주소와 일치하는지 검사함

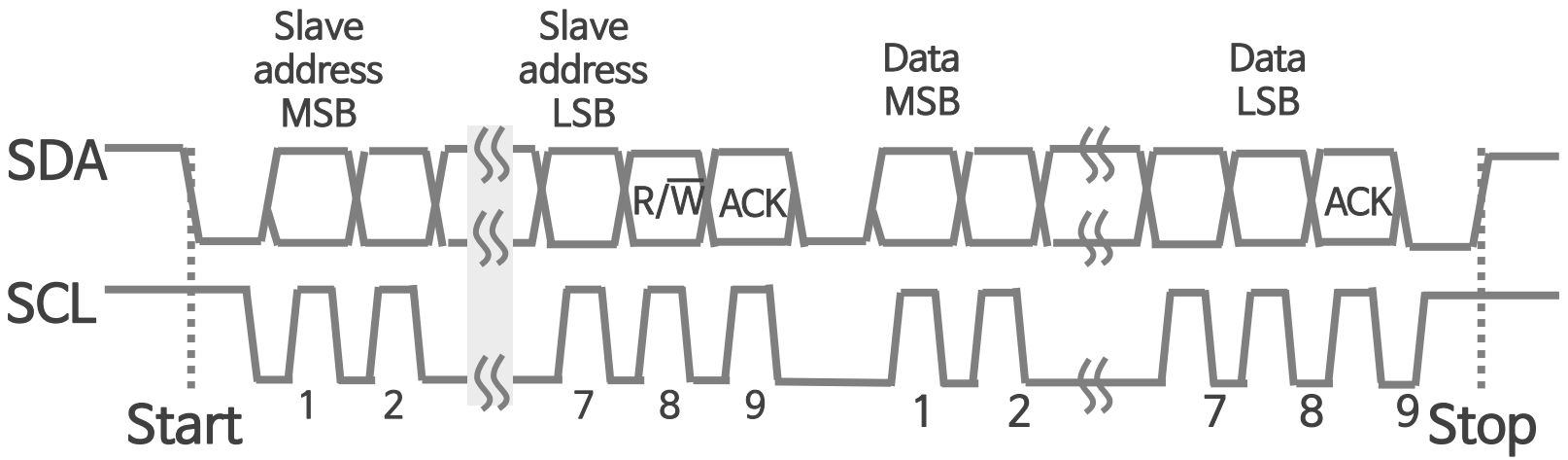
만일 Master가 출력한 주소가 자신의 주소와 같으면 그 Slave는 ACK 비트에 0을 출력하여 Master에게 응답함

  - 아무도 일치하지 않으면 아무도 ACK비트로 0을 출력하지 않으므로 1상태를 유지함 ➡ NACK

I2C

6 Master-Slave 형태로 동작함

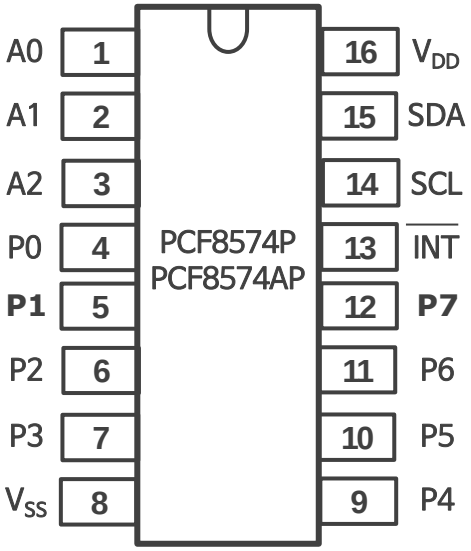
읽기인지 쓰기인지를 가리킴



I2C

- **I2C 활용 사례**

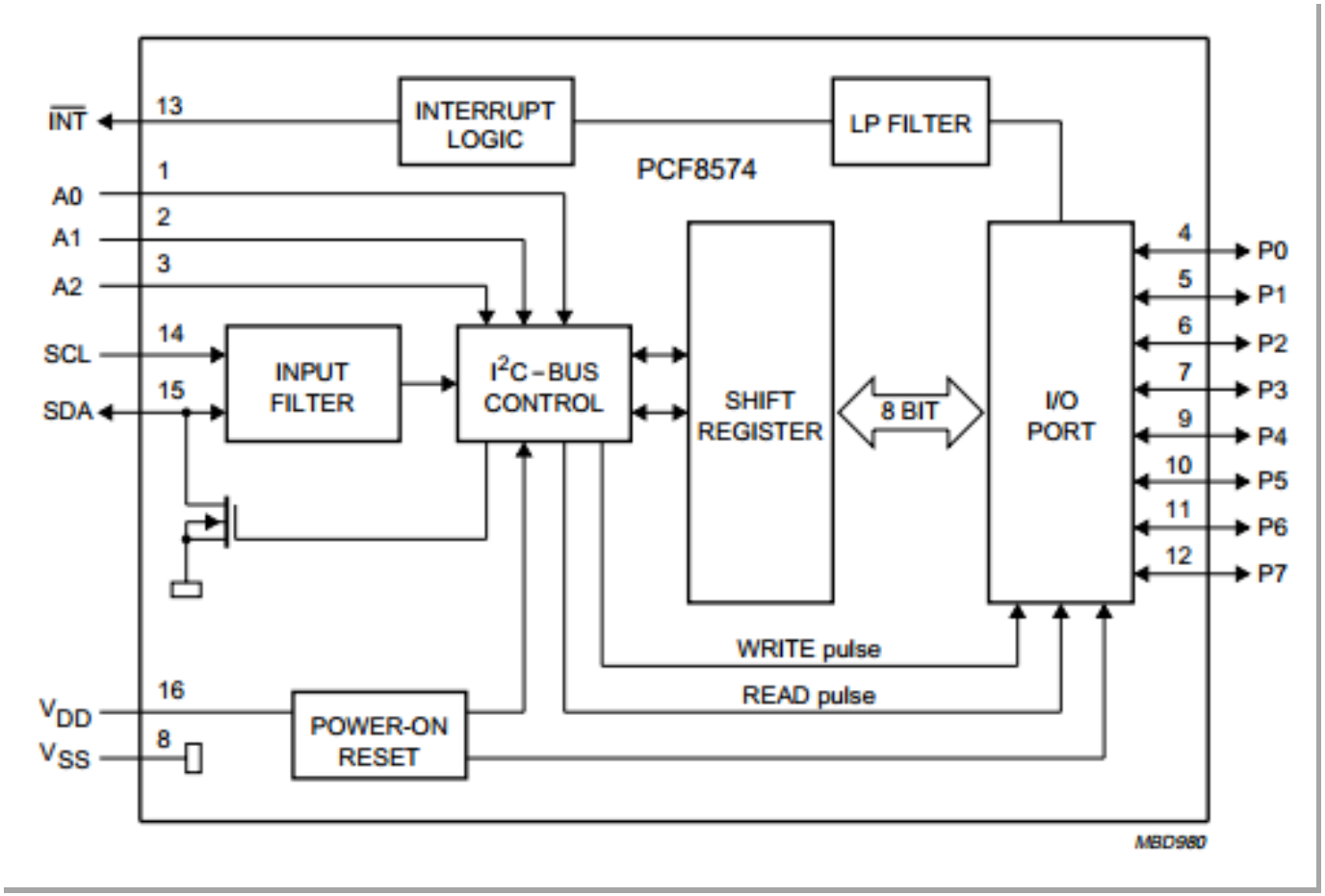
PCF8574AP



SYMBOL	PIN	DESCRIPTION
A0	1	address input 0
A1	2	address input 1
A2	3	address input 2
P0	4	quasi-bidirectional I/O 0
P1	5	quasi-bidirectional I/O 1
P2	6	quasi-bidirectional I/O 2
P3	7	quasi-bidirectional I/O 3
V_{SS}	8	supply ground
P4	9	quasi-bidirectional I/O 4
P5	10	quasi-bidirectional I/O 5
P6	11	quasi-bidirectional I/O 6
P7	12	quasi-bidirectional I/O 7
$\overline{INT}$	13	Interrupt output (active LOW)
SCL	14	Serial clock line
ADA	15	Serial data line
V_{DD}	16	Supply voltage

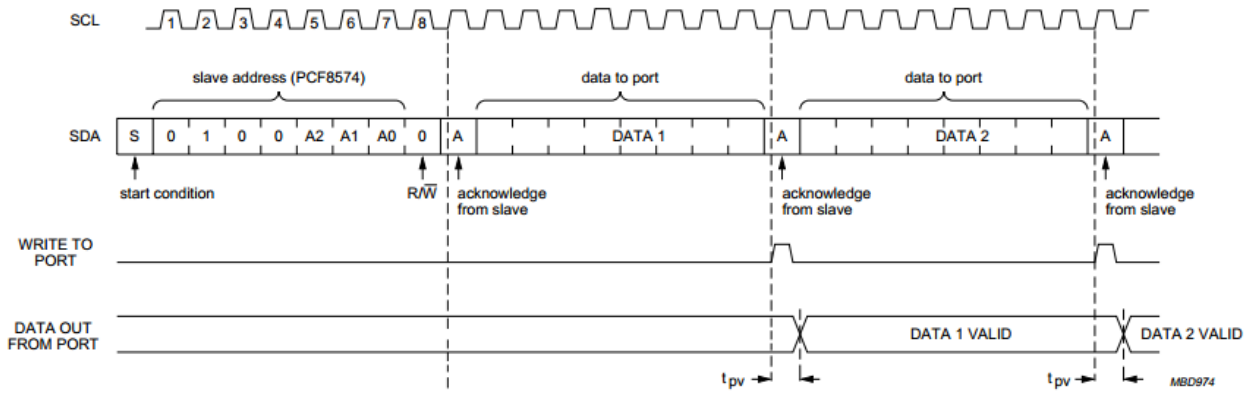
I2C

- PCF8574AP



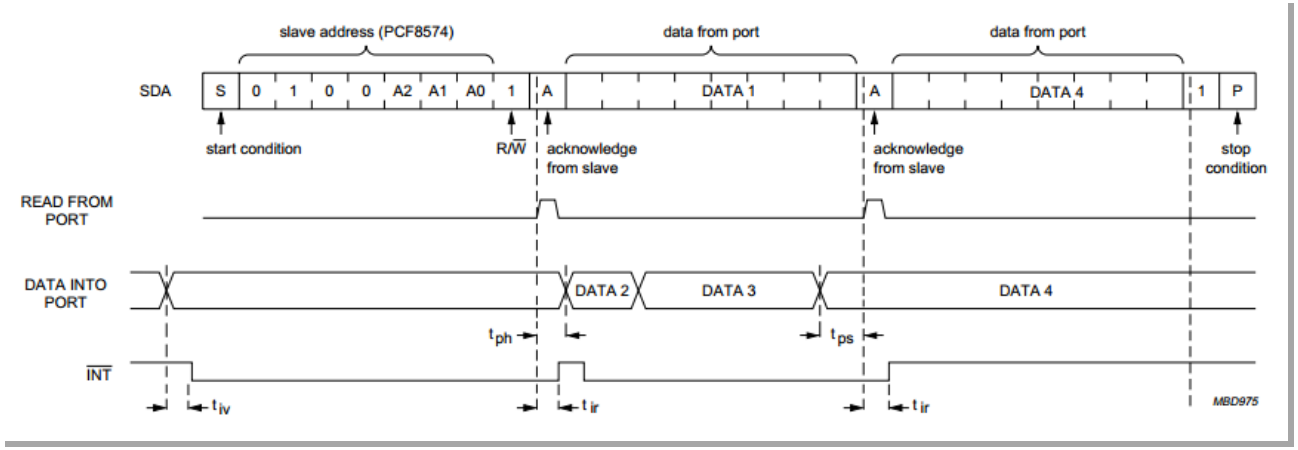
I2C

- PCF8574AP



[Write Mode(Output)]

[Read Mode(Input)]



SPI

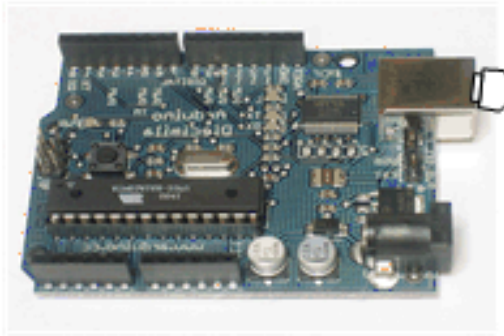
- **SPI(Serial Peripheral Interface)**
  - 하나의 Master와 하나 또는 다수의 Slave Device 간 통신이 가능함
    - SPi는 기본적으로 4개의 Signal Pin을 제공함

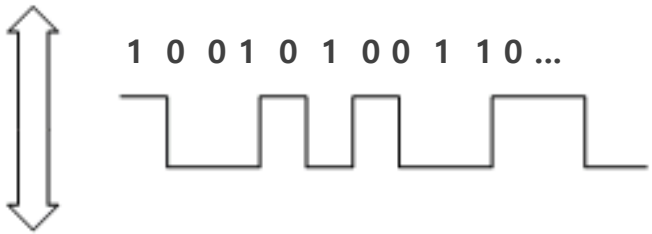
MOSI	Master에서 Slave로 보내는 데이터 핀
MISO	Slave에서 Master로 보내는 데이터 핀
SS(Slave Select)	Slave를 선택하기 위한 핀
SCK(Serial Clock)	Master에서 출력하는 데이터 동기 출력 핀

SPI

Laptop

Cable



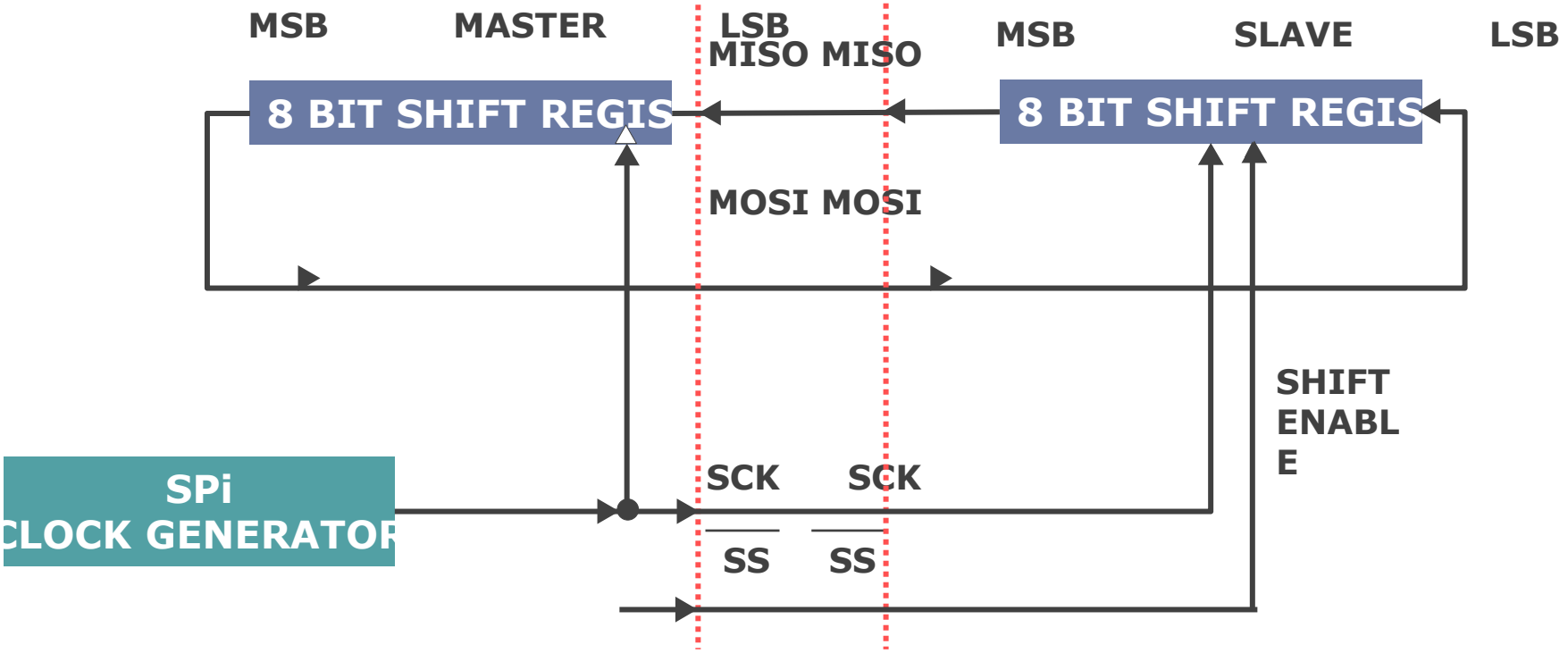


Information passes between the computer and Arduino through the USB cable.

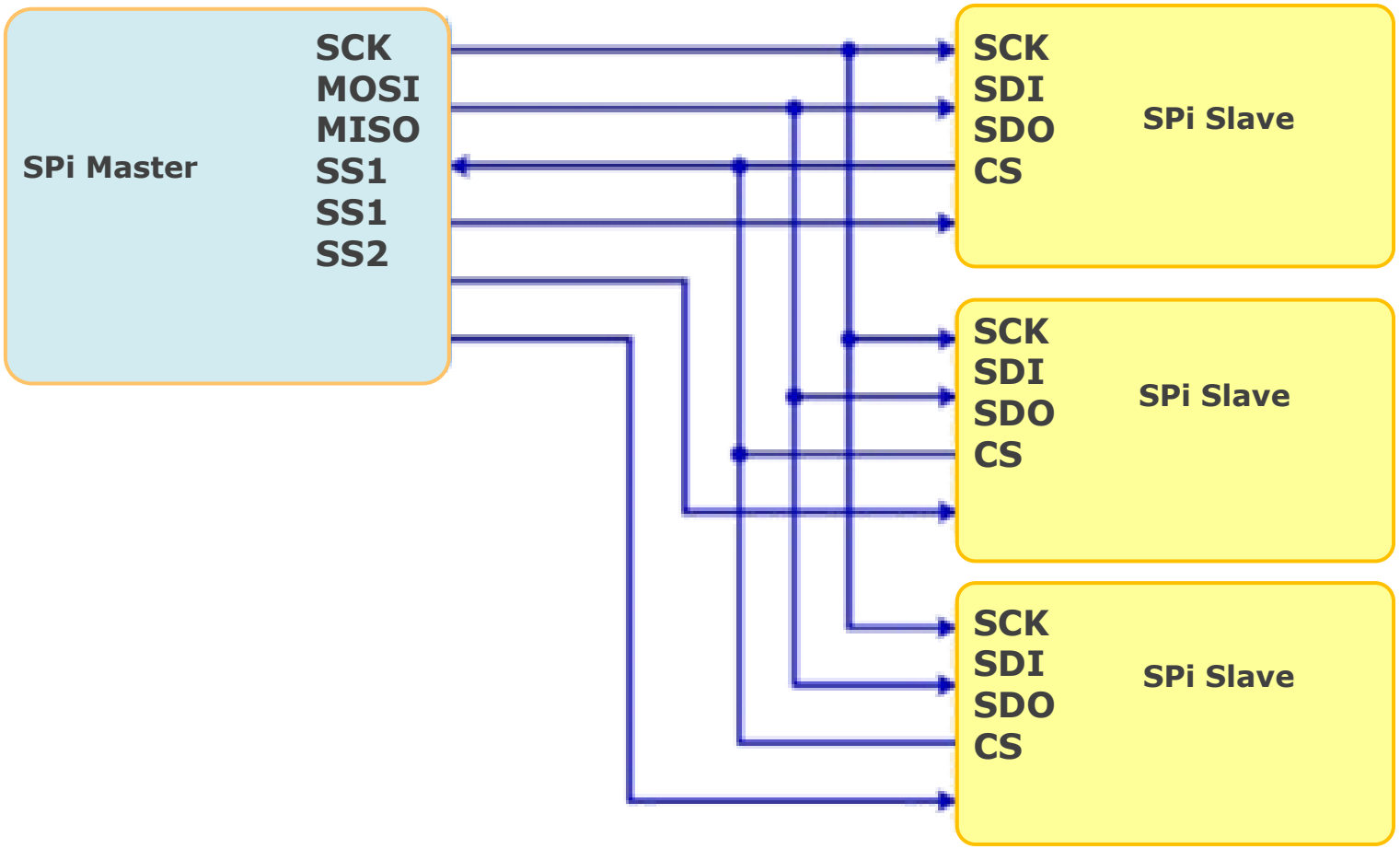
Information is transmitted as zeros(‘0’) and ones(‘1’)... also known as bits!

SPI

- **SPI**의 동작


MSB
MASTER
LSB
MISO MISO
MSB
SLAVE
LSB
8 BIT SHIFT REGIS
8 BIT SHIFT REGIS
MOSI MOSI
SHIFT
ENABL
E
SPi
CLOCK GENERATOR
SCK
SCK
SS
SS

SPI

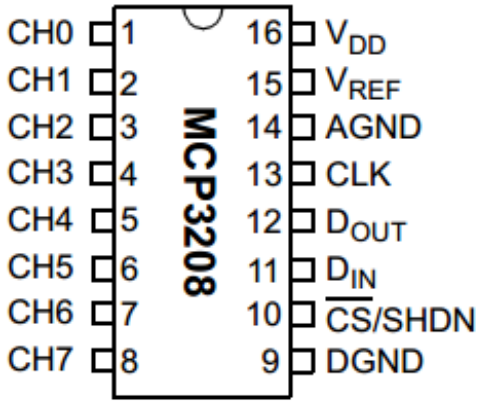

SPi Master
SCK
MOSI
MISO
SS1
SS1
SS2
SCK
SDI
SDO
CS
SPi Slave
SCK
SDI
SDO
CS
SPi Slave
SCK
SDI
SDO
CS
SPi Slave

SPI

- 슬레이브 모드
  - SS가 입력 Pin으로 동작하며 0 레벨상태로 입력되는 경우에만 SPi가 활성화되고 MISO가 출력단자로 됨
  - 외부에서 입력된 SCK 신호에 의하여 데이터 레지스터의 1바이트가 전송되고 나면 SPSR(SPi Status Register)의 비트7(SPIF)이 1로 되면서 SPi 전송완료 인터럽트가 요청됨
- 마스터 모드
  - SPCR(SPi Control Register)의 비트4(MSTR)가 1로 설정되면 마스터가 되며 사용자는 SS 핀의 방향을 DDRB 0 비트를 사용하여 결정할 수 있음
  - 마스터 모드에서 SS Pin이 입력으로 설정되어 있고 0레벨이면 외부에서 다른 마스터가 슬레이브로 지정하고 데이터를 전송하기 시작하는 것으로 해석될 수 있으므로 SS Pin을 입력으로 설정하면 반드시 Pull-up 저항과 같은 외부회로에 의해 1레벨로 유지되어야 함

SPI

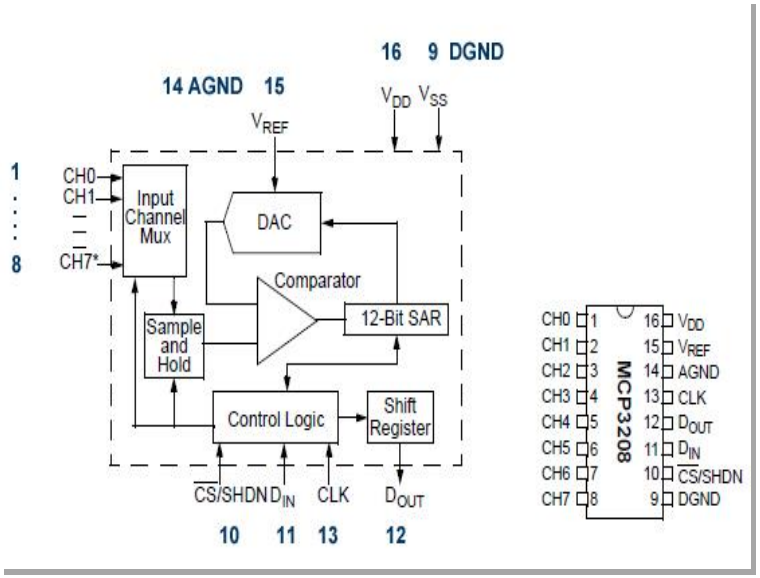
- **MCP3208**

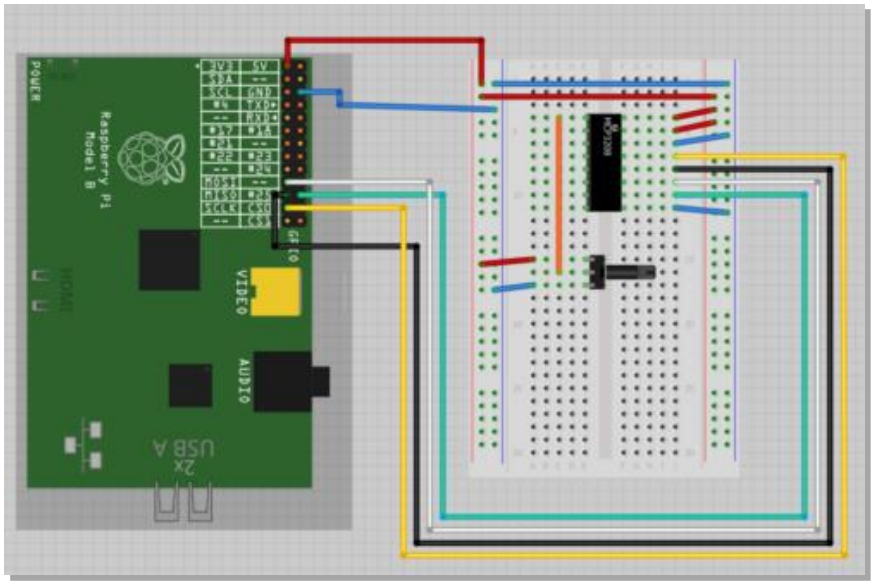


Name	Function
V_{DD}	+2.7V to 5.5V Power Supply
DGND	Digital Ground
AGND	Analog Ground
CH0-CH7	Analog Inputs
CLK	Serial Clock
D_{IN}	Serial Data In
D_{OUT}	Serial Data Out
CS/SHDN	Chip Select/Shutdown Input
V_{REF}	Reference Voltage Input

SPI

- **MCP3208**
  - MCP3208의 ADC 구조와 외부 연결





SPI

- SPi 통신 프로그램
  - 마스터인 ARM 프로세서가 정해진 명령어를 슬레이브인 MCP3208에게 정해진 명령으로 날려주면 그에 해당되는 값을 전송해주는 시스템임
  - 센서 값을 ADC 하여 ARM 프로세서가 받아야 하므로, 정해진 규칙으로 데이터를 전송함
  - 센서 값은 테스트 목적으로 가변저항을 사용함
  - 규칙은 MCP3208 데이터시트, 첨부파일을 참조함

SPI

- **SPI 통신 사례**
  - 데이터 전송 규칙

라즈베리파이는 총 3개의 바이트를 날려줌

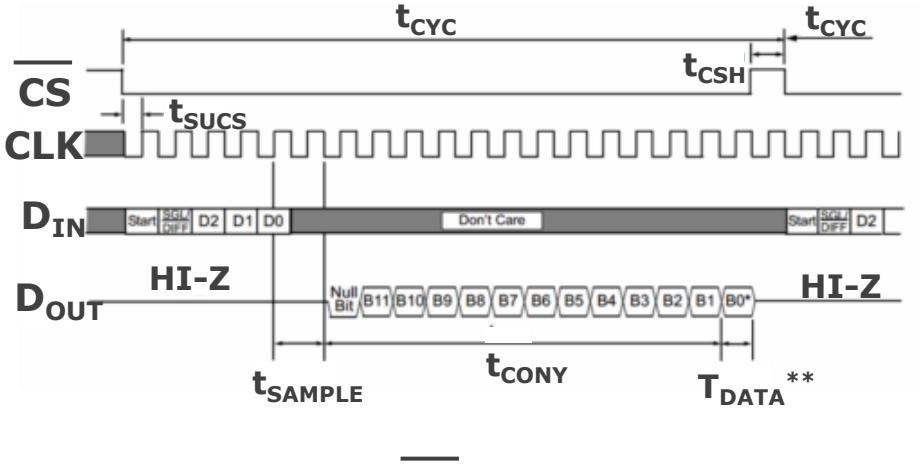
- 1첫째 바이트 0 0 0 0 0 Start SGL D2
- 2번째 바이트 D1 D0 x x x x x x
- 3번째 바이트 x x x x x x x x

※ D2,1, 0은 ADC 채널을 의미하고,
x는 어떠한 값도 상관없는 Don`t Care를 의미함

SPI

- **SPI 통신 사례**

  - 라즈베리파이는 3바이트의 데이터를 보냄과 동시에 MCP3208로부터 3바이트의 데이터를 받음
  - 그 데이터의 밑으로부터 12비트가 ADC 결과값임
  - 이를 위의 소스와 같이 비트연산으로 하나의 데이터 값으로 변환하면 됨

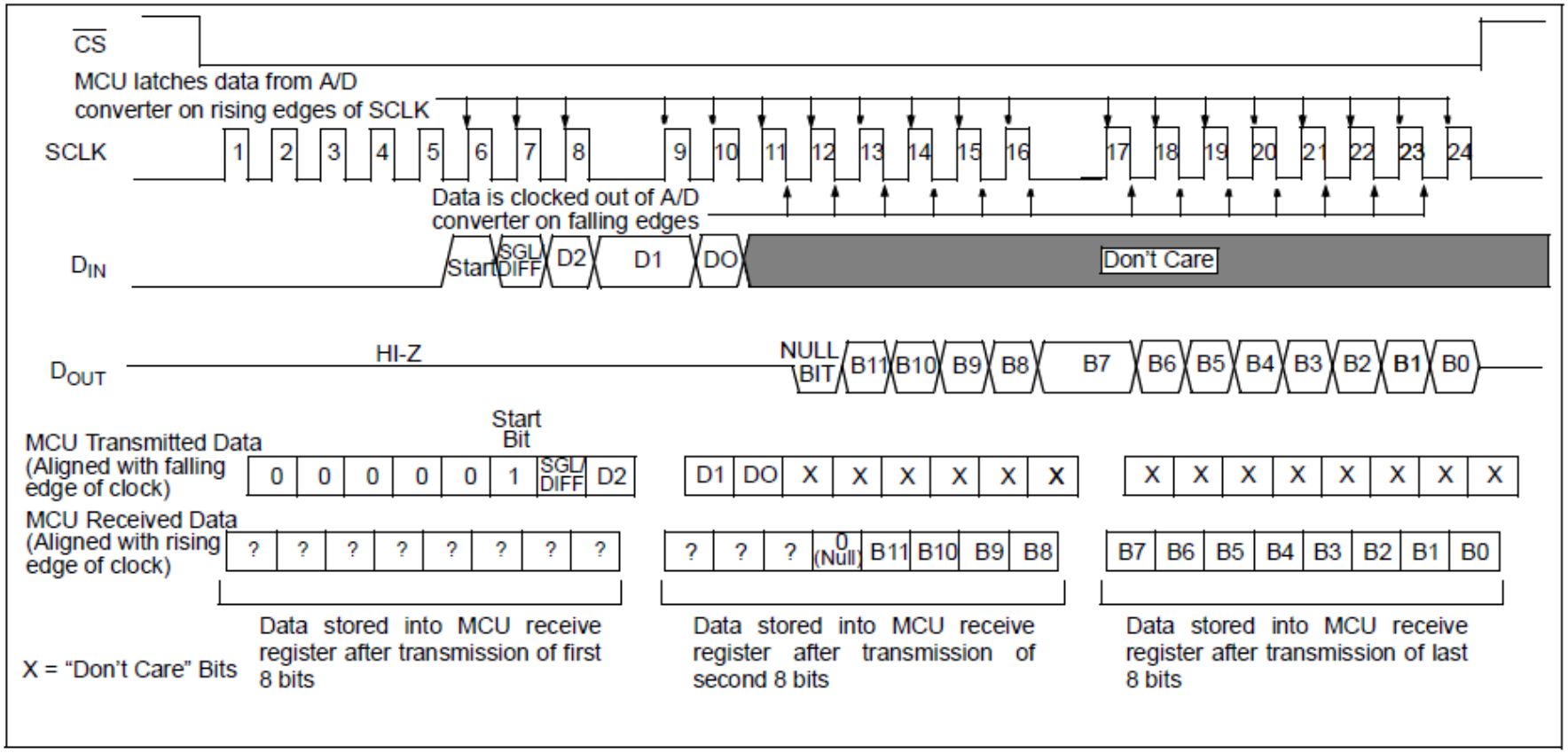


***After completing the data transfer, if further clocks are applied with CS low, the A/D converter will output LSB first data, followed by zeros indefinitely(see Figure 5-2 below)**

****t_{DATA} : during this time, the bias current and the comparator power down while the reference input becomes a high impedance node, leaving the CLK running to clock out the LSB-first data or zeros.**

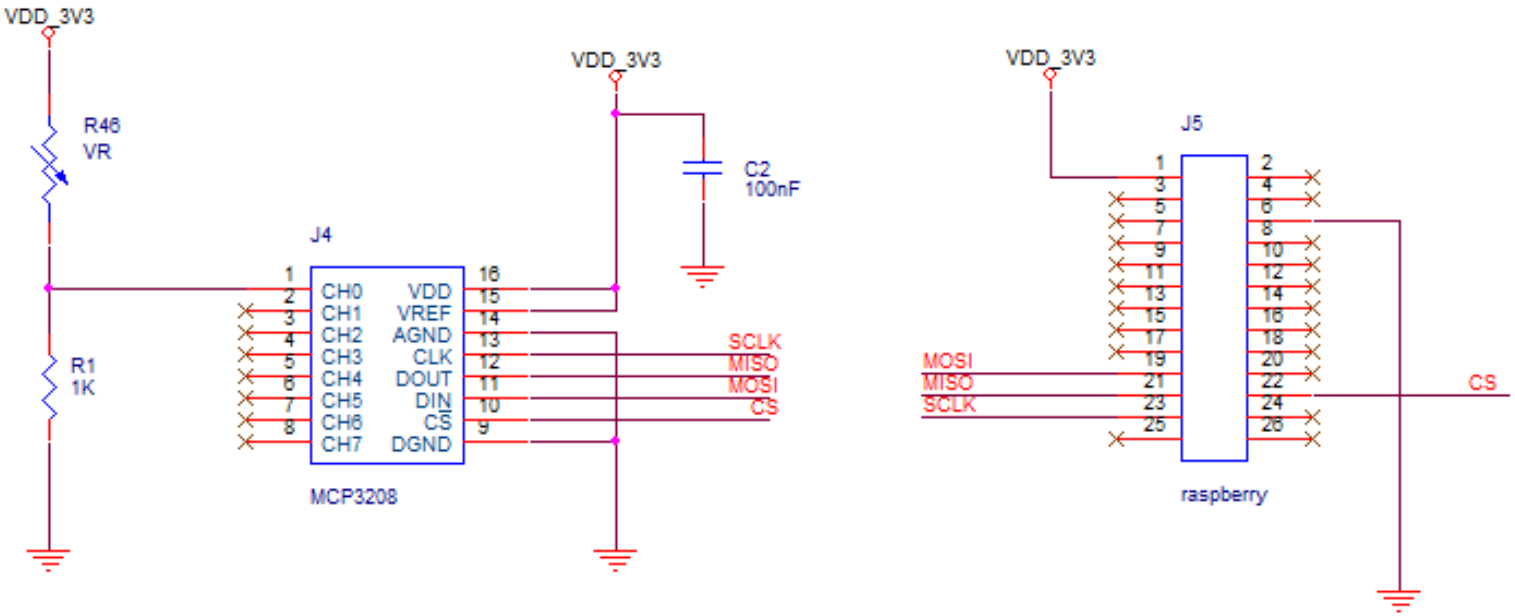
SPI

- **SPi 통신 사례 waveform**



SPI

- **MCP3208과 ARM board와 연결**



SPI

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <wiringPi.h>
#include <wiringPiSPI.h>
#define CS_MCP3208 6 // BCM_GPIO 25
#define SPI_CHANNEL 0
#define SPI_SPEED 1000000 // 1MHz

int read_mcp3208_adc(unsigned char adcChannel){
unsigned char buff[3];
int adcValue = 0;
                                        buff[0] = 0x06 | ((adcChannel & 0x07) >> 2);
                                        buff[1] = ((adcChannel & 0x07) << 6);
                                        buff[2] = 0x00;
                                        digitalWrite(CS_MCP3208, 0); // Low : CS Active
                                        wiringPiSPIDataRW(SPI_CHANNEL, buff, 3);
                                        buff[1] = 0x0F & buff[1];
                                        adcValue = ( buff[1] << 8) | buff[2];
                                        digitalWrite(CS_MCP3208, 1); // High : CS Inactive
                                        return adcValue;
                                }
```

SPI

```
int main (void) {
int adcChannel = 0;
int adcValue = 0;
if(wiringPiSetup() == -1) {
      fprintf (stdout, "Unable to start wiringPi: %s□n", strerror(errno));
      return 1 ;
}
pinMode(CS_MCP3208, OUTPUT);
if(wiringPiSPISetup(SPI_CHANNEL, SPI_SPEED) == -1) {
      fprintf (stdout, "wiringPiSPISetup Failed: %s□n", strerror(errno));
      return 1 ;
}
            while(1) {
                  adcValue = read_mcp3208_adc(adcChannel);
                  printf("adc0 Value = %u□n", adcValue);
            }
            return 0;
            }
```