

Chapter 211. SPI - OLED 디스플레이

1. SPI

SPI는 Serial Peripheral Interface의 약어로 고속의 주변장치 연결을 위한 시리얼 통신 방식 중 하나이다. 이전 장에서 UART 시리얼 통신에 대해 살펴보았다. UART는 간단하면서도 널리 사용되는 직렬 통신 방법 중 하나이지만 일대일 통신만이 가능하다는 한계가 있으며 아두이노 우노는 하나의 UART 시리얼 통신만을 지원하므로 여러 개의 UART 시리얼 통신을 사용하는 장치를 사용하고자 하는 경우 소프트웨어를 통한 SoftwareSerial 라이브러리를 사용해야 했다. 이에 비해 SPI는 마스터 장치 하나에 슬레이브 장치 여러 개가 연결되는 일대다 통신이 가능하다. 마스터 장치는 연결을 시작하고 통신을 제어하는 책임을 지는 장치로 연결이 성립된 이후에는 두 개의 데이터 선을 통해 전이중 방식으로 데이터를 동시에 주고받을 수 있다. UART 통신은 비동기 방식의 통신인 반면 SPI는 동기 통신 방식으로 동기화를 위해 별도의 클록 신호를 전송하므로 별도의 데이터 선을 필요로 한다. 또한 일대다 연결에서 특정 슬레이브 장치를 선택하기 위해 데이터 선을 사용하므로 총 4개의 연결선을 가진다.

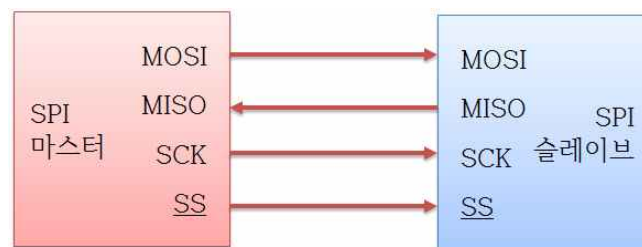


그림 1. 일대일 SPI 연결

그림 1에서 SCK (Serial Clock)는 직렬 클록으로 동기화를 위한 클록 전송을 위해 사용된다. MOSI는 Master Out Slave In의 약어로 마스터 장치에서 슬레이브 장치로 데이터를 전송하기 위해 사용되며, MISO는 Master In Slave Out의 약어로 슬레이브 장치에서 마스터 장치로의 데이터 전송을 위해 사용된다. SS는 Slave Select의 약어로 마스터 장치가 데이터를 주고받을 슬레이브 장치를 선택하기 위해 사용한다. 선택되지 않은 슬레이브 장치의 SS는 HIGH 상태에 있는 반면 선택된 슬레이브 장치의 SS는 LOW 상태에 있다. 이는 여러 개의 슬레이브 장치가 연결된 경우 보다 명확히 알 수 있다.

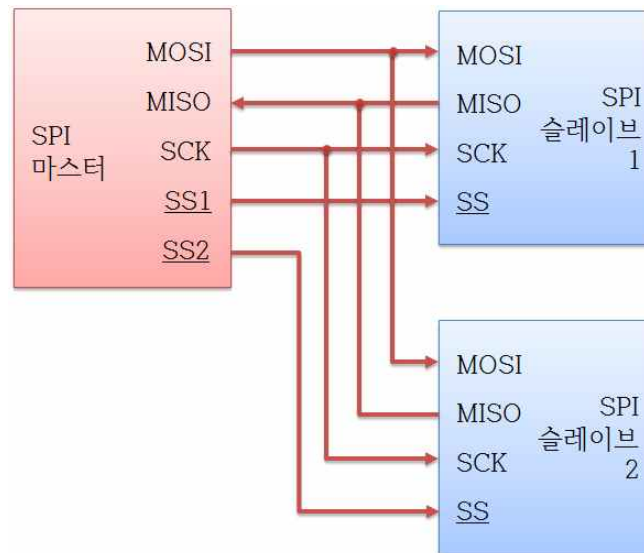


그림 2. 일대다 SPI 연결

그림 2는 하나의 마스터 장치에 2개의 슬레이브 장치가 연결된 예이다. 모든 슬레이브 장치는 마스터로부터의 SCK, MOSI, MISO 연결선을 공유하고 있다. 즉, 마스터 장치의 데이터가 모든 슬레이브 장치로 전달된다. 하지만 실제로 데이터를 받아들이는 슬레이브는 SS 라인이 LOW 상태에 있는 슬레이브이다. 즉, 일대다 연결이 가능하지만 특정 순간에는 일대일 통신만이 가능하다. 또한 슬레이브 장치의 수가 늘어날수록 전용의 SS 라인은 슬레이브 장치의 수와 동일하게 증가한다.

SPI의 데이터 전송 방식은 다른 직렬 통신 방법과 차이가 있으며 항상 송신과 수신이 동시에 일어난다는 특징을 가진다. UART 시리얼 통신에서도 송신과 수신은 동시에 일어날 수 있지만 항상 그런 것은 아니며 송신과 수신은 별개로 동작한다. SPI는 동기 방식으로 동작하므로 동기화를 위해 클럭이 필요하고 이 클럭은 마스터 장치에 의해 공급된다. 슬레이브 장치에서 마스터 장치로 데이터를 보내고자 하는 경우에도 슬레이브 장치는 마스터 장치의 클럭을 기준으로 데이터를 전송해야 하지만 슬레이브 장치는 클럭을 만들지 않는다. 따라서 SPI에서는 동기화의 문제점을 송신과 수신에 동시 진행되도록 함으로써 해결하고 있다. SPI의 마스터와 슬레이브 장치의 데이터 버퍼는 원형 큐를 이루고 있다고 생각하면 이해가 쉽다. 마스터 장치에서 슬레이브 장치로 보낼 데이터 A가 마스터 장치의 큐에 저장되어 있고, 슬레이브 장치에서 마스터 장치로 보낼 데이터 B가 슬레이브 장치의 큐에 저장되어 있다고 가정해 보자.

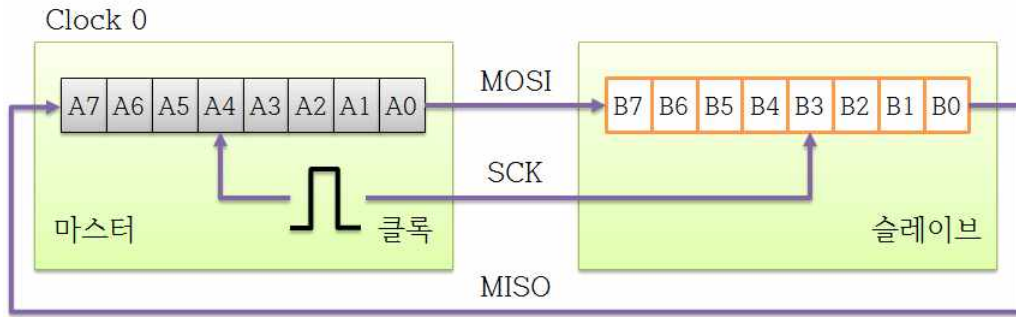


그림 3. 데이터 전송 준비

데이터가 준비된 상태에서 하나의 클럭이 발생하였다고 가정해 보자. 마스터 장치의 1비트 데이터 A0는 MOSI를 통해 슬레이브 장치로 전달되며 동시에 슬레이브 장치의 1비트 데이터 B0는 MISO를 통해 마스터 장치로 전달된다. 이 때 마스터 장치와 슬레이브 장치의 데이터는 데이터 A와 B가 오른쪽 원형 이동 연산을 수행한 것과 동일한 결과를 얻게 된다. 다만 데이터 A와 B가 저장된 버퍼는 마스터와 슬레이브 장치에 물리적으로 분리되어 있다.

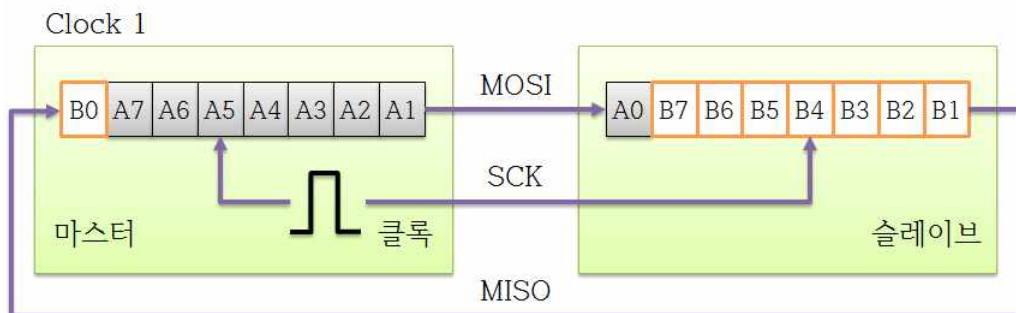


그림 4. 1비트 데이터 전송

비슷하게 8개의 클럭이 발생하게 되면 마스터의 8비트 데이터 A는 슬레이브로 전달되고, 슬레이브의 8비트 데이터 B는 마스터로 전달되어 1바이트의 데이터가 ‘교환’된다. 따라서 마스터가 슬레이브로 데이터를 보내는 경우에도 마스터는 슬레이브로부터 의미 없는 데이터를 받게 되며, 슬레이브로부터 데이터를 받는 경우에도 마스터는 슬레이브로 의미 없는 데이터를 보내야 한다.

데이터를 교환함에 있어 유의할 점 한 가지는 클럭의 역할이다. 클럭은 데이터가 전달되는 시점을 알려주어 전달된 데이터가 안정적인 상태를 유지할 때 수신 장치가 데이터를 샘플링하도록 지시하기 위해 사용된다. 하지만 클럭의 어느 부분에서 데이터 전송이 완료되었는지에 따라 수신 장치가 데이터를 읽는 시점이 달라지며, 이를 위해 클럭의 극성(Clock Polarity,

CPOL)과 클럭 위상(Clock Phase, CPHA)이 사용된다.

- CPOL : SPI 버스가 유힬 상태일 때의 클럭 값을 결정한다. CPOL = 0이면 비활성 상태일 때 SCK는 LOW 값을 가지며, CPOL = 1이면 비활성 상태일 때 SCK는 HIGH 값을 가진다.
- CPHA : 데이터를 샘플링 하는 시점을 결정한다. CPHA = 0이면 데이터는 비활성 상태에서 활성 상태로 바뀌는 에지에서 샘플링 되고, CPHA = 1이면 데이터는 활성 상태에서 비활성 상태로 바뀌는 에지에서 샘플링 된다.

그림 5는 CPHA = 0인 경우, 즉, 클럭이 비활성 상태에서 활성 상태로 바뀌는 시점에서 샘플링 되는 경우를 나타낸다. CPOL = 0이면 상승 에지에서, CPOL = 1이면 하강 에지에서 데이터가 샘플링 된다.

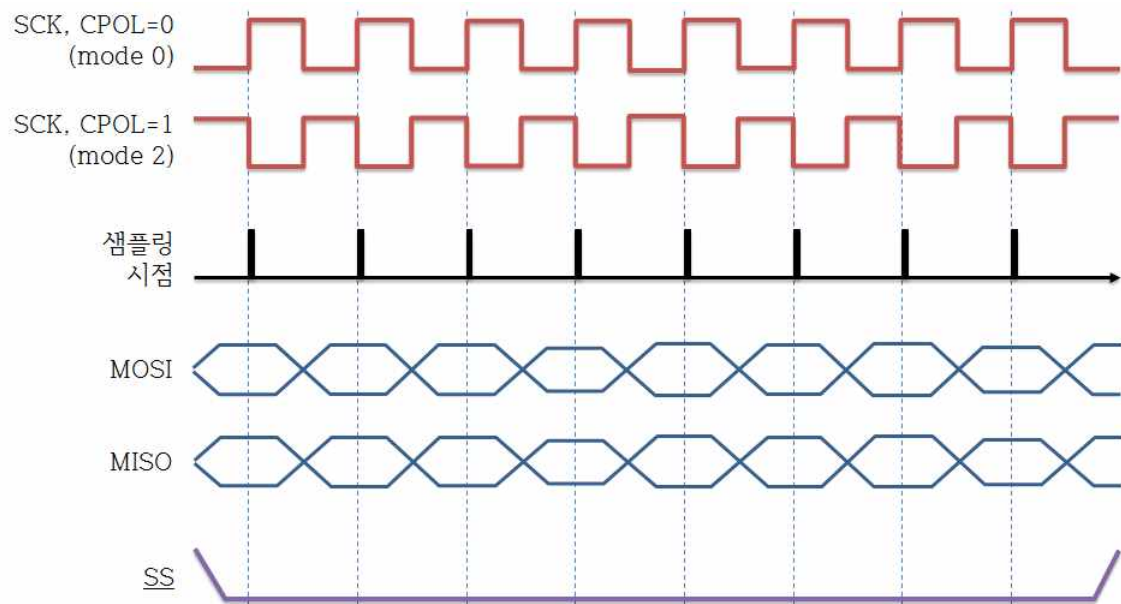


그림 5. CPHA = 0인 경우 데이터 전송 다이어그램

그림 6은 CPHA = 1인 경우, 즉, 클럭이 활성 상태에서 비활성 상태로 바뀌는 시점에서 샘플링 되는 경우를 나타낸다. CPOL = 0이면 하강 에지에서, CPOL = 1이면 상승 에지에서 샘플링 된다.

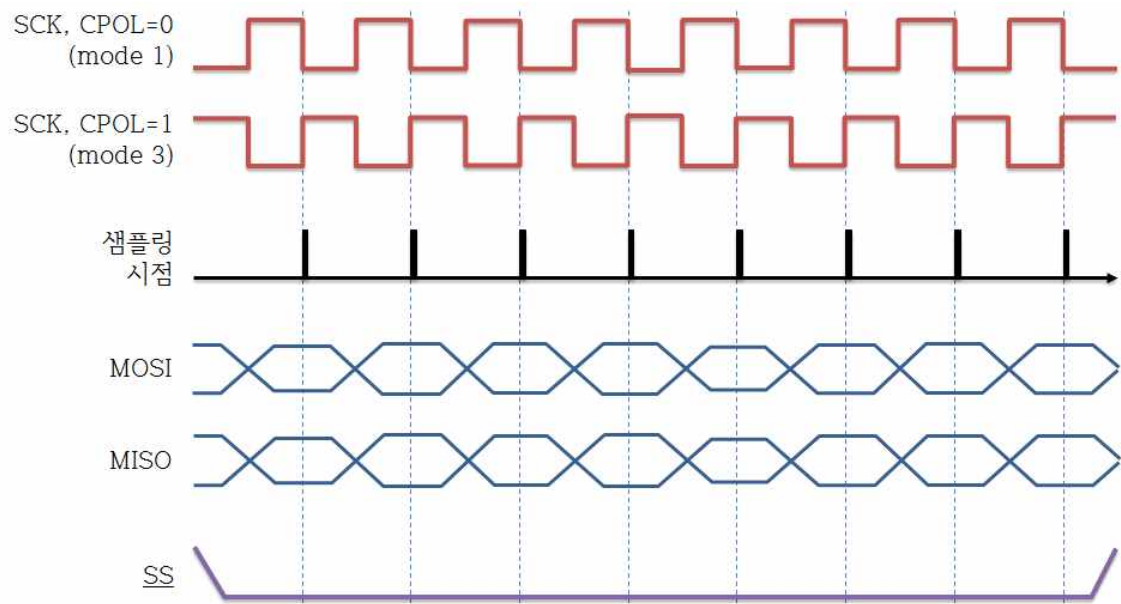


그림 6. CPHA = 1인 경우 데이터 전송 다이어그램

그림 5와 그림 6에서 알 수 있듯이 수신 장치가 데이터를 샘플링 하는 시점은 데이터가 안정화 된 이후, 즉 데이터 비트의 가운데에서 얻어진다. UART의 경우 비동기적으로 데이터가 전송되므로 한 비트의 데이터 확인을 위해 여러 번의 샘플링을 통해 실제 전송된 데이터를 파악하지만 SPI의 경우에는 동기화를 위한 클록이 별도로 존재하므로 한 번의 샘플링만으로 충분하다. 한 가지 더 유의할 점은 데이터가 샘플링 되는 동안 SS는 LOW 상태를 유지하고 있어야 한다는 점이다.

2. 아두이노 우노의 SPI 통신

아두이노 우노에 사용된 ATmega328은 SPI 통신을 지원하고 있으며 이를 위해 전용 핀을 정의하고 있다. 아두이노 우노에서 SPI 통신을 위해 사용하는 핀은 표 1과 같다.

SPI 신호	Arduino UNO	비고
MOSI	디지털 11	데이터
MISO	디지털 12	데이터
SCK	디지털 13	클록
SS	디지털 10	변경 가능

표 1. SPI 연결 핀

표 1에서 MOSI, MISO, SCK는 변경할 수 없지만 SS는 변경이 가능하다. 여러 개의 슬레이브를 연결하기 위해서는 여러 개의 SS 연결선이 필요하므로 디지털 10번 핀만을 사용하는 것은 불가능하다. 아두이노에서 SPI 장치 연결을 위한 SS 연결핀으로 디지털 10번 핀을 흔히 사용하므로 10번 핀이 할당되어 있을 뿐 특별한 의미가 있는 것은 아니다. 다만 언급하였다시피 다른 핀들은 변경할 수 없다. 아두이노 우노에서 SPI 통신을 위한 각 핀은 다음과 같이 정의되어 있다.

```
static const uint8_t SS      = 10;
static const uint8_t MOSI    = 11;
static const uint8_t MISO    = 12;
static const uint8_t SCK     = 13;
```

아두이노에는 SPI 통신을 지원하기 위해 기본 라이브러리 중 하나로 SPI 라이브러리를 제공하고 있다. SPI 통신을 통해 송수신되는 데이터의 의미는 연결하는 장치에 따라 달라지므로 SPI 라이브러리에는 기본적인 설정 및 데이터 전송 함수들만을 정의하고 있다. 또한 SPI 통신은 하나의 경로를 통해서만 데이터를 송수신하므로 데이터 송수신을 위한 객체는 하나만 생성할 수 있다. 즉, SPI 라이브러리에서 정의하고 있는 클래스는 SPIclass이며 이의 전역 객체인 SPI를 통해 SPI 통신이 이루어진다. 이는 UART 시리얼 통신을 담당하는 유일한 객체가 Serial인 점과 동일하다. SPI 클래스에는 다음과 같은 멤버 함수들이 정의되어 있다.

- begin

```
void begin(void)
- 매개변수 : 없음
- 반환값 : 없음
```

SPI 통신을 초기화한다.

- end

```
void end(void)
- 매개변수 : 없음
- 반환값 : 없음
```

SPI 통신을 종료한다.

- setBitOrder

```
void setBitOrder(uint8_t bitOrder)
- 매개변수
  bitOrder : 데이터 송수신시 비트 순서 (LSBFIRST 또는 MSBFIRST)
- 반환값 : 없음
```

직렬로 전송되는 데이터 비트의 전송 순서를 설정한다. LSBFIRST (LSB 우선) 또는 MSBFIRST (MSB 우선) 중 하나의 값을 가진다.

- setClockDivider

```
void setClockDivider(uint8_t rate)
- 매개변수
  rate : 분주비율
- 반환값 : 없음
```

시스템에서 사용하는 클록에 상대적인 분주비를 설정한다. AVR 기반의 보드에서는 2, 4, 8, 16, 32, 64, 128 중 하나의 값을 사용할 수 있으며 이 값들은 SPI_CLOCK_DIV2에서 SPI_CLOCK_DIV128 까지의 상수로 정의되어 있다. 디폴트값은 SPI_CLOCK_DIV4로 시스템 클록의 1/4 속도로 데이터를 전송한다. 아두이노 우노는 16MHz 클록을 사용하므로 그 1/4인 4MHz가 SPI의 기본 주파수에 해당한다.

- setDataMode

```
void setDataMode(uint8_t mode)
```

- 매개변수
mode : 전송 모드
- 반환값 : 없음

클록 극성(CPOL)과 클록 위상(CPHA)에 따른 전송 모드를 지정한다. 모드 번호는 그림 5와 그림 6을 참고하면 되며 각 모드는 표 2의 상수로 정의되어 있다.

Mode	극성 (CPOL)	위상 (CPHA)
SPI_MODE0	0	0
SPI_MODE1	0	1
SPI_MODE2	1	0
SPI_MODE3	1	1

표 2. SPI 통신에서의 전송 모드

클록 극성과 클록 위상의 의미를 정확히 이해하지 못해도 사용에 지장은 없지만 마스터와 슬레이브가 동일한 모드를 사용하여야만 정상적인 통신이 가능하므로 아두이노에 주변 장치를 연결하는 경우 주변 장치에서 사용하는 모드를 데이터시트를 확인하고 이에 맞게 아두이노의 동작 모드를 설정해야 한다.

- transfer

```
byte transfer(byte data)
```

- 매개변수
data : 송신할 데이터
- 반환값 : 수신된 데이터

SPI 버스를 통해 한 바이트의 데이터를 보내고 한 바이트의 데이터를 받아서 반환한다.

아두이노에서는 SPI 라이브러리를 제공하고 있지만 연결된 주변장치를 SPI 라이브러리를 이용하여 직접 제어하는 경우는 드물다. 그렇다면 SPI 라이브러리는 어디에 사용할까? 아두이노와 함께 사용할 수 있는 SPI 통신을 사용하는 주변장치는 어렵지 않게 찾아볼 수 있지만 대부분의 장치들은 전용의 라이브러리가 제공되고 있다. 하지만 이들 전용의 라이브러리들은 아

두이노의 SPI 라이브러리를 바탕으로 만들어져 있으므로 제공되는 라이브러리의 기능을 수정하거나 보완하고 싶다면 SPI 라이브러리에 대해 알고 있어야 한다.

SPI 라이브러리를 사용함에 있어 한 가지 주의할 점은 SPI 라이브러리가 마스터 모드만을 지원한다는 점이다. 따라서 슬레이브 장치로 다른 장치에 아두이노를 연결할 수 없으며, 슬레이브 모드로 사용하고자 하는 경우에는 ATmega328 데이터시트를 참고하여 레지스터를 설정하여야 한다.

3. OLED 디스플레이

최근 스마트폰에서 사용되어 뛰어난 화질로 주목을 받고 있는 디스플레이에 OLED가 있다. OLED는 유기 발광 다이오드(Organic Light-Emitting Diode)로 이전의 LCD와는 픽셀의 구현 방식에서 차이가 있다. LCD는 백라이트에서 발산한 빛이 액정에 의해 굴절되어 투과하는 빛의 양을 조절함으로써 픽셀을 구현하는 방식을 사용한다. 이에 비해 OLED는 스스로 빛을 내는 유기 소자로 픽셀을 구성하고 있으므로 백라이트가 필요하지 않다. 따라서 OLED는 LCD에 비해 얇게 만드는 것이 가능하며 곡면 디스플레이나 투명한 디스플레이 역시 제작할 수 있다. 이외에도 OLED는 명암비가 높고, 시야각이 넓으며, 응답 속도가 빠른 등 LCD에 비해 많은 장점을 가지고 있다. 하지만 OLED는 수율이 LCD에 비해 낮아 가격이 비싼 단점이 있다. 이 장에서 사용한 디스플레이는 SPI 통신을 사용하는 0.96인치 크기의 OLED 디스플레이로 128×64의 해상도를 가지고 있다.



그림 7. OLED 디스플레이

OLED 디스플레이 모듈은 7개의 연결선을 가지며 이 중 2개는 전원, 1개는 리셋, 나머지 4개

는 데이터 선에 해당한다. 4개의 데이터 선 중 3개는 SPI 통신을 위한 연결선이다. SPI 통신을 위해서는 4개의 연결선이 필요하지만 MISO는 슬레이브에서 마스터로, 즉, OLED 디스플레이에서 아두이노 우노로 데이터를 전달하는 핀으로 정보 표현을 위해서는 필요하지 않으므로 연결하지 않고 MOSI, SCK, CS(또는 SS)의 3개 연결만을 사용한다. 나머지 하나인 DC는 디스플레이 드라이버 칩 제어를 위해 사용된다. 일반적으로 디스플레이를 제어는 드라이버 칩으로 정보를 전송한 후 드라이버 칩이 동작하도록 지시함으로써 이루어진다. 이 때 드라이버 칩으로 전달하는 정보는 (화면을 지우거나 커서를 특정 위치로 옮기는 등의) 명령어이거나 (화면에 출력할 문자를 나타내는 등의) 데이터 두 가지가 있으며 이를 구분하기 위해 DC 핀이 사용된다. 텍스트 LCD 제어를 위한 LiquidCrystal 라이브러리의 RS(Register Select), 그래픽 LCD 제어를 위한 GLCD 라이브러리의 DI(Data Instruction) 등도 기본적으로 이와 동일한 목적으로 사용되는 핀이다.

OLED 모듈 핀	설명	아두이노 연결 핀	비고
GND		GND	
VCC		VCC	
D0	SCK	디지털 13번	
D1	MOSI	디지털 11번	
RES	RESET	RESET	
DC	Data/Command Selection	디지털 9번	디스플레이 드라이버 칩 제어용
CS	SS	디지털 10번	Slave Select 또는 Chip Select

표 3. OLED 디스플레이 핀 설명

OLED 디스플레이 제어를 위해서는 전용 라이브러리인 U8glib를 사용한다. 먼저 해당 홈페이지에서 아두이노를 위한 라이브러리를 다운받아¹⁾ 스케치북 디렉터리 아래 libraries 디렉터리에 설치하자. 먼저 그림 8과 같이 OLED 디스플레이를 아두이노 우노와 연결해 보자.

1) <https://code.google.com/p/u8glib/>

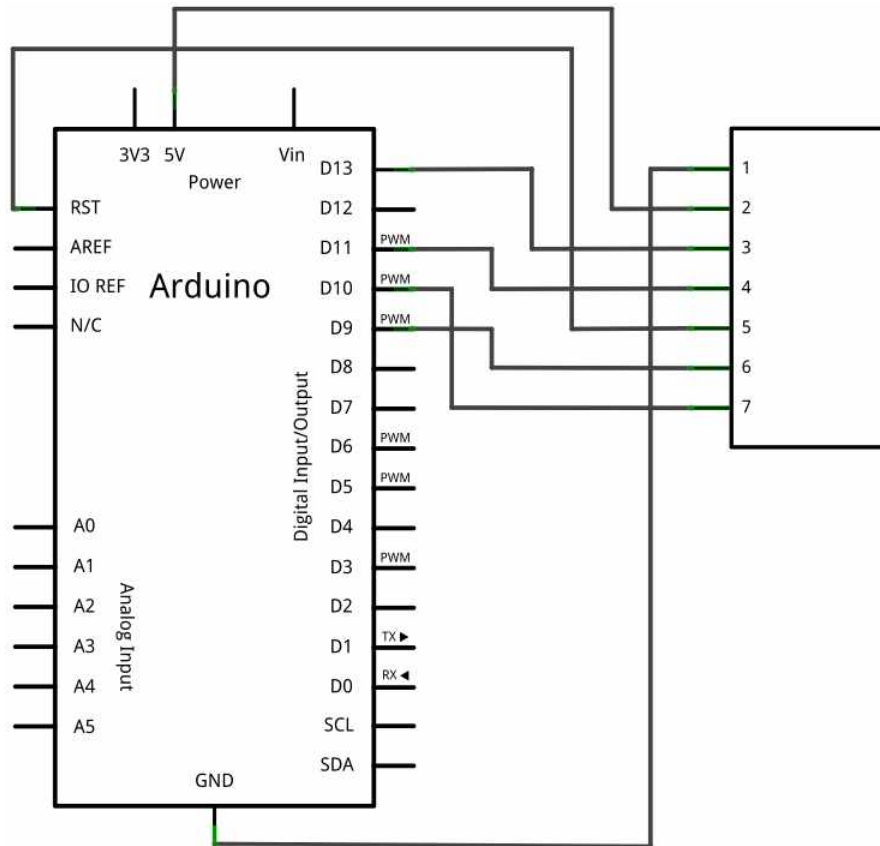


그림 8. OLED 디스플레이와 아두이노 우노 연결 회로도

U8glib 라이브러리는 다양한 형태와 다양한 연결 방식의 디스플레이를 지원하고 있으며 SPI 방식의 OLED 디스플레이는 그 중 하나이다. U8glib 라이브러리에서는 다양한 디스플레이를 지원하기 위해 마이크로컨트롤러의 메모리에 버퍼를 생성하고, 버퍼에 먼저 그리기를 완료한 후 디스플레이 버퍼로 옮기는 방식을 사용하고 있다. 이를 위해 화면에 나타낼 내용들은 그리기 명령어들을 모아놓은 함수를 사용하고 아두이노의 loop 함수에서 그리기 전용 함수를 호출하여 그리기를 수행한다. 그리기 함수는 일반적으로 draw라는 이름으로 만들어지며 원한다면 다른 이름으로 작성해도 무방하다. loop 함수에서 draw를 호출할 때에도 주의할 점이 있다. 그리기를 수행하기 위해서는 draw를 두 번 이상 호출해 주어야 한다는 점이다. U8glib 라이브러리는 그리기를 위해 마이크로컨트롤러의 메모리를 버퍼로 사용한다. 하지만 마이크로컨트롤러의 메모리는 제한적이어서 디스플레이의 크기가 커지면 메모리 사용이 많아져서 문제가 될 수 있다. 이러한 점을 해결하기 위해 U8glib 라이브러리는 화면을 몇 개의 영역으로 나누어서 반복해서 그리기 방식을 사용하고 있다. 스케치 1은 OLED 디스플레이에 문자와 직선을 표시하는 예이다.

스케치 1. OLED 디스플레이 테스트

```
#include "U8glib.h"

// OLED 디스플레이 연결
// SCK = 13, MOSI = 11, CS = 10, A0 = 9
U8GLIB_SH1106_128X64 u8g(13, 11, 10, 9);

void draw(void){                                // 그리기 명령어 집합
    u8g.setFont(u8g_font_unifont);              // 폰트 설정
    u8g.setPrintPos(10, 20);                     // 문자열 출력 시작 위치
    u8g.print("Hello OLED...");
    u8g.setPrintPos(30, 50);
    u8g.print("by hgycap");
    u8g.drawLine(10, 10, 120, 60);               // 직선 그리기 (x1, y1, x2, y2)
}

void setup(void) {
}

void loop(void) {
    // 그리기 루틴
    u8g.firstPage();                             // 첫 번째 영역
    do{
        draw();
    }while(u8g.nextPage());                       // 다음 영역

    delay(500);                                  // 이미지 갱신
}
```

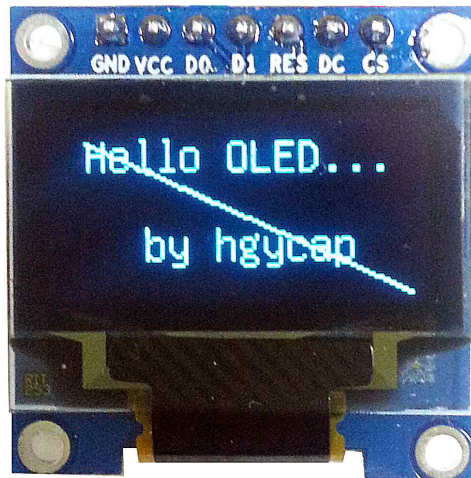


그림 9. 스케치 1 실행 결과

draw 함수를 작성할 때에 주의할 점 중 한 가지는 전역변수의 값을 수정해서는 안된다는 점이다. 만약 1초 간격으로 증가하는 카운터 변수 count를 선언하고 이 내용을 표시하고 싶다면 draw 함수가 아니 loop 함수에서 count 값을 증가시키고 draw 함수에서는 count 값을 사용하기만 하여야 한다. 스케치 2는 1초 간격으로 증가하는 카운터의 값을 OLED 디스플레이에 표시하는 스케치의 예이다.

스케치 2. OLED 디스플레이

```
#include "U8glib.h"

// OLED 디스플레이 연결
// SCK = 13, MOSI = 11, CS = 10, A0 = 9
U8GLIB_SH1106_128X64 u8g(13, 11, 10, 9);

int count = 0;
unsigned long time_previous, time_current;

void draw(void){
    u8g.setFont(u8g_font_unifont);
    u8g.setPrintPos(10, 20);
    u8g.print("Count : ");
    u8g.setPrintPos(20, 35);
    // 그리기 명령어 집합
    // 폰트 설정
    // 문자열 출력 시작 위치
```

```
    u8g.print(count);
}

void setup(void) {
    time_previous = millis();
}

void loop(void) {
    time_current = millis();
    // 1초에 한 번 카운트 값을 증가시키고 카운터 값이 변한 경우 디스플레이 갱신
    if(time_current - time_previous > 1000){
        time_previous = time_current;
        count++;

        // 그리기 루틴
        u8g.firstPage();                // 첫 번째 영역
        do{
            draw();
        }while(u8g.nextPage());        // 다음 영역
    }
}
```

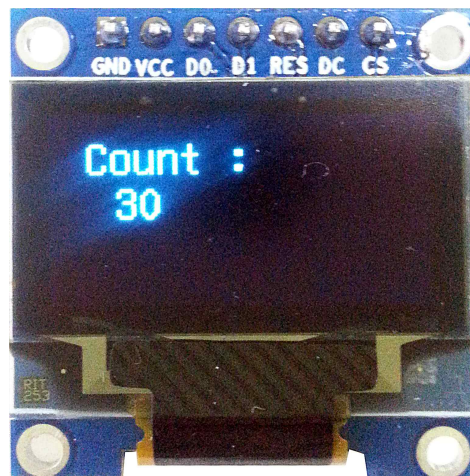


그림 10. 스케치 2 실행 결과