

Chapter 21. 아두이노-아두이노 SPI 통신

1. SPI

SPI는 Serial Peripheral Interface의 약어로 고속의 주변장치 연결을 위한 시리얼 통신 방식 중 하나이다. 이전 장에서 살펴본 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 사용하기 어렵다. 이에 비해 SPI는 마스터 장치 하나에 슬레이브 장치 여러 개가 연결되는 1:n 통신이 가능하다. 마스터 장치는 연결을 시작하고 통신을 제어하는 책임을 지는 장치로 연결이 성립된 이후에는 두 개의 데이터 선을 통해 송신과 수신이 동시에 진행될 수 있는 전이중 방식으로 데이터를 주고받을 수 있다. UART 통신은 비동기 방식의 통신인 반면 SPI는 동기 통신 방식으로 동기화를 위해 별도의 클록 신호를 전송하므로 클록을 위한 별도의 연결선을 필요로 한다. 또한 1:n 연결에서 특정 슬레이브 장치를 선택하기 위해 데이터 선을 사용하므로 총 4개의 연결선을 가진다.

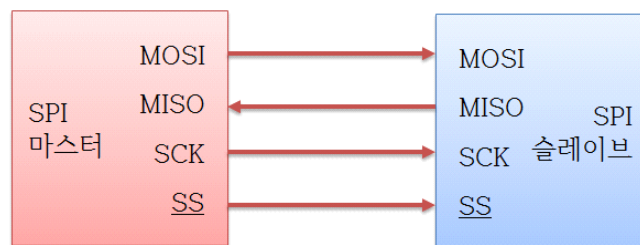


그림 1. 일대일 SPI 연결

그림 1에서 SCK (Serial Clock)는 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. 2개의 슬레이브 장치가 연결된 경우를 살펴보자.

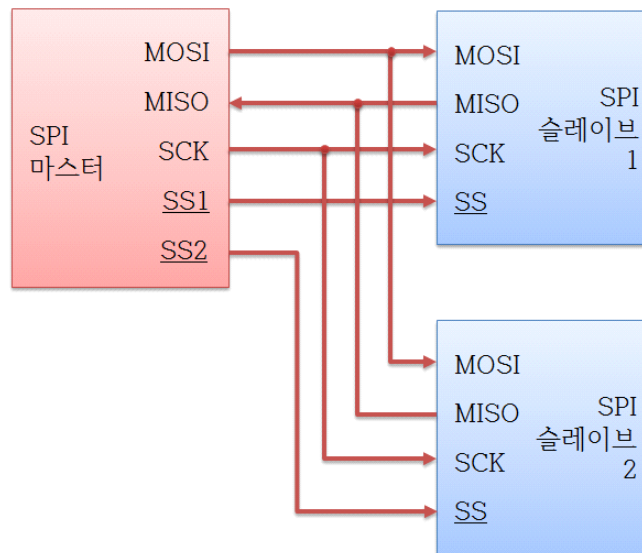


그림 2. 일대다 SPI 연결

그림 2는 하나의 마스터 장치에 2개의 슬레이브 장치가 연결된 경우이다. 모든 슬레이브 장치는 마스터로부터의 SCK, MOSI, MISO 연결선을 공유하고 있다. 하지만 실제로 데이터를 받아들이는 슬레이브는 SS 라인이 LOW 상태에 있는 슬레이브이다. 즉, 일대다 연결이 가능하지만 특정 순간에는 일대일 통신만 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 수신에 동시에 일어난다는 특징을 가진다. UART 시리얼 통신에서도 송신과 수신에 동시에 일어날 수 있지만 항상 그런 것이 아니라 필요한 경우에만 데이터를 전송하므로 송신과 수신은 별개로 동작한다. SPI는 동기 방식으로 동작하므로 동기화를 위해 클록이 필요하고 이 클록은 마스터 장치에 의해 공급된다. 따라서 슬레이브 장치에서 마스터 장치로 데이터를 보내고자 하는 경우에도 슬레이브 장치는 마스터 장치의 클록을 기준으로 데이터를 전송해야 한다. 즉, 슬레이브가 자발적으로 마스터로 데이터를 전송하는 것은 불가능하며, 마스터의 요구가 있는 경우에만 슬레이브에서 마스터로 데이터 전송이 가능하다.

2. 아두이노에서의 SPI

아두이노에 사용된 마이크로컨트롤러는 하드웨어적으로 SPI 통신을 지원하고 있으므로 전용 핀을 통해 연결하여야 한다. SPI 연결을 위해서는 4개의 연결선이 필요하며 이 중 3개의 연결선인 MOSI, MISO, SCK는 변경할 수 없다. 반면 SS는 임의로 선택할 수 있다.

SPI 신호	아두이노 우노	아두이노 메가2560	비고
MOSI	디지털 11	디지털 51	데이터
MISO	디지털 12	디지털 50	데이터
SCK	디지털 13	디지털 52	클록
SS	디지털 10	디지털 53	변경 가능

표 1. SPI 연결 핀

아두이노는 기본 라이브러리 중 하나로 SPI 라이브러리를 포함하고 있으며 이를 통해 SPI 통신이 가능하다. SPI 라이브러리에는 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. 2, 4, 8, 16, 32, 64, 128 중 하나의 값을 사용할 수 있으며 이 값들은 SPI_CLOCK_DIV2에서 SPI_CLOCK_DIV128까지의 상수로 정의되어 있다. 디폴트값은 SPI_CLOCK_DIV4로 시스템 클록의 1/4 속도로 데이터를 전송한다.

- setDataMode

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

- 매개변수
mode : 전송 모드 (SPI_MODE0 ~ SPI_MODE3)
- 반환값 : 없음

클록 극성(CPOL)과 클록 위상(CPHA)에 따른 전송 모드를 지정한다. 동일한 전송 모드를 사용하지 않는 경우에는 정상적인 통신이 불가능하므로 마스터와 슬레이브를 반드시 동일한 모드로 설정하여야 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 주변장치에 따라 통신 방식에 차이가 있어

SPI 라이브러리를 이용하여 직접 장치를 제어하는 경우는 드물다. 대신 SPI 라이브러리를 바탕으로 만들어진 전용 라이브러리를 사용하는 경우가 대부분이다. SPI 라이브러리를 사용할 때 한 가지 주의할 점은 SPI 라이브러리가 마스터 모드만을 지원한다는 점이다. 따라서 SPI 라이브러리를 사용하여 아두이노를 다른 장치에 슬레이브로 연결할 수 없으며, 슬레이브 모드로 사용하고자 하는 경우에는 데이터시트를 참고하여 레지스터를 설정해 주어야 한다.

3. 아두이노-아두이노 SPI 통신

아두이노 메가2560을 마스터로, 아두이노 우노를 슬레이브로 연결하여 데이터를 송수신하는 스케치를 작성해 보자. 학습보드에는 아두이노 우노와 아두이노 메가2560이 SPI 통신이 가능하도록 연결되어 있다. 학습보드를 사용할 때 한 가지 유념할 점은 학습보드에 포함된 외장 EEPROM 역시 SPI 통신을 사용하며 SS 연결로 디지털 42번 핀을 사용하고 있다는 점이다. 따라서 아두이노 우노와의 통신을 위해서는 디지털 42번 핀에 HIGH 값을 출력하여 EEPROM이 선택되지 않은 상태로 두어야 한다.

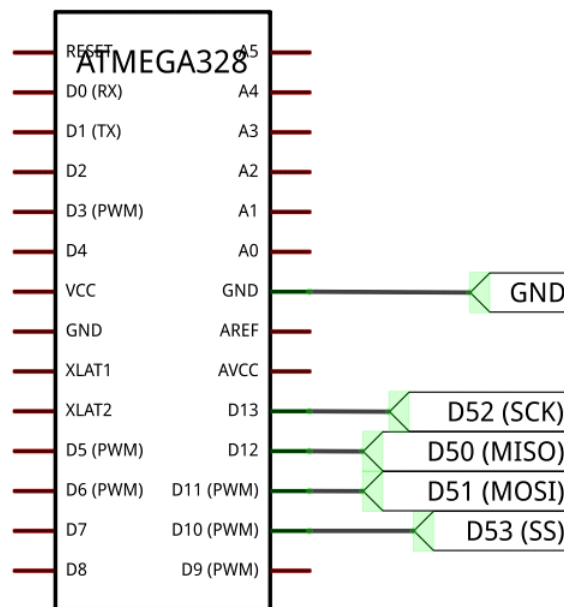


그림 3. 아두이노 우노와 아두이노 메가2560의 SPI 통신 연결 회로도

아두이노 메가2560에 업로드 할 마스터를 위한 스케치는 간단하다. 스케치 1은 SPI 라이브러리의 마스터 모드를 통해 슬레이브로 1초에 한 번 메시지를 전송하는 스케치의 예이다.

스케치 1. SPI 통신 테스트 - 메가2560

```
#include <SPI.h>

int SS_arduino_uno = 53;
int SS_EEPROM = 42;

void setup (void){
  pinMode(SS_arduino_uno, OUTPUT);
  pinMode(SS_EEPROM, OUTPUT);
  digitalWrite(SS_arduino_uno, HIGH);    // 슬레이브가 선택되지 않은 상태
  digitalWrite(SS_EEPROM, HIGH);

  SPI.begin();                          // SPI 통신 초기화
  // 안정적인 전송을 위해 분주비를 높여 전송 속도를 낮춤
  SPI.setClockDivider(SPI_CLOCK_DIV16);
}

void loop (void){
  const char *p = "Hello, World\n";
  digitalWrite(SS_arduino_uno, LOW);    // 슬레이브 선택
  for (int i = 0; i < strlen(p); i++){  // 문자열 전송
    SPI.transfer(p[i]);
  }
  digitalWrite(SS_arduino_uno, HIGH);    // 슬레이브 선택 해제

  delay(1000);
}
```

SPI 라이브러리는 슬레이브 모드를 지원하지 않으므로 해당 레지스터를 직접 설정하여야 하며 3가지 설정이 필요하다.

- SPI 통신을 사용 가능하도록 설정
- SPI 통신에서 슬레이브로 동작하도록 설정
- SPI 통신을 통해 데이터가 수신된 경우 인터럽트가 발생하도록 설정

이 세 가지는 모두 SPI Control Register인 SPCR의 해당 비트를 설정해 줌으로써 가능하다. SPCR 레지스터 구조는 그림 5와 같다.

비트	7	6	5	4	3	2	1	0
	SPIE	SPE	DORD	MSTR	CPOL	CPHA	SPR1	SPR0
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초깃값	0	0	0	0	0	0	0	0

그림 4. SPCR 레지스터 구조

SPCR 레지스터 중 6번 비트인 SPE(SPI Enable) 비트를 1로 설정함으로써 SPI 통신이 가능하다. 4번 비트인 MSTR(Master/Slave Select) 비트가 1인 경우 마스터 모드로 동작하며 0인 경우 슬레이브 모드로 동작한다. 마지막으로 7번 비트인 SPIE(SPI Interrupt Enable) 비트를 1로 설정하면 SPI 통신에 의한 인터럽트 발생이 허용된다.

슬레이브는 인터럽트 방식으로 동작한다. SPI 통신을 통해 데이터가 수신되면 자동으로 인터럽트 서비스 루틴(ISR, Interrupt Service Routine)이 호출되며 ISR의 매개변수는 SPI 통신을 통한 데이터 수신 인터럽트에 해당하는 인터럽트 번호 SPI_STC_vect가 전달된다. 스케치 2는 마스터로부터 수신된 데이터를 시리얼 모니터로 출력하는 스케치의 예이다.

스케치 2. SPI 통신 테스트 - 우노

```
#include <SPI.h>

char buf[100]; // 수신된 문자 저장을 위한 버퍼
// pos와 process_it은 인터럽트 처리 루틴에서 값을 바꾸는 변수이므로
// volatile 선언을 통해 매번 값을 참조하도록 설정하여야 한다.
volatile byte pos = 0; // 수신 버퍼에 문자를 기록할 위치
volatile boolean process_it = false; // 수신 데이터 종료 알림

void setup (void){
    Serial.begin (9600);

    // MOSI, SCLK, SS는 입력으로 설정하지 않아도 된다.
    // MISO 역시 이 예에서는 사용하지 않으므로 생략할 수 있다.
```

```
pinMode(MISO, OUTPUT);
pinMode(MOSI, INPUT);
pinMode(SCK, INPUT);
pinMode(SS, INPUT);

// 마스터의 전송 속도에 맞추어 통신 속도를 설정
SPI.setClockDivider(SPI_CLOCK_DIV16);

// SPI 통신을 위한 레지스터 설정
// SPCR : SPI Control Register
SPCR |= (1 << SPE);           // SPE : SPI Enable
SPCR &= ~(1 << MSTR);        // MSTR : Master Slave Select
SPCR |= (1 << SPIE);         // SPIE : SPI Interrupt Enable
}

// SPI 통신으로 데이터가 수신된 경우 자동 호출
ISR (SPI_STC_vect)
{
    byte c = SPDR;             // 수신된 데이터
    if (pos < sizeof(buf)){
        buf[pos++] = c;

        if (c == '\n'){       // 데이터 수신 종료
            process_it = true;
        }
    }
}

void loop (void){
    if (process_it){           // 데이터 수신이 종료된 경우
        buf[pos] = 0;          // 문자열의 끝 표시
        Serial.print(buf);

        pos = 0;
        process_it = false;
    }
}
```

```
}
```

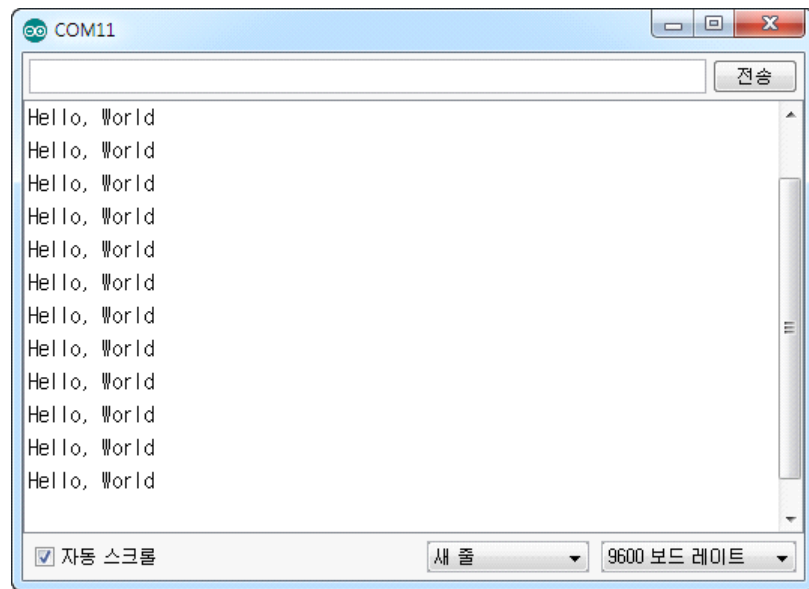


그림 5. 스케치 1과 스케치 2 실행 결과 - 우노