

Chapter 113. SPI

1. SPI

SPI는 Serial Peripheral Interface의 약어로 주변장치를 위한 직렬 인터페이스를 가리킨다. 앞에서 UART 직렬 통신에 대해 살펴보았다. UART는 전이중 방식의 직렬 통신이지만 일대일 통신만이 가능하다는 한계가 있다. 이에 비해 SPI는 전이중 방식의 직렬 통신인 점은 동일하지만 마스터 장치 하나에 슬레이브 장치 여러 개가 연결된 일대다¹⁾ 통신이 가능하다. 마스터 장치는 연결을 시작하고 통신을 제어하는 책임을 지는 장치로 연결이 성립되면 두 개의 데이터 선을 통해 전이중 방식으로 데이터를 동시에 주고받을 수 있다. 마스터 장치에 하나의 슬레이브 장치가 연결된 경우를 살펴보자.

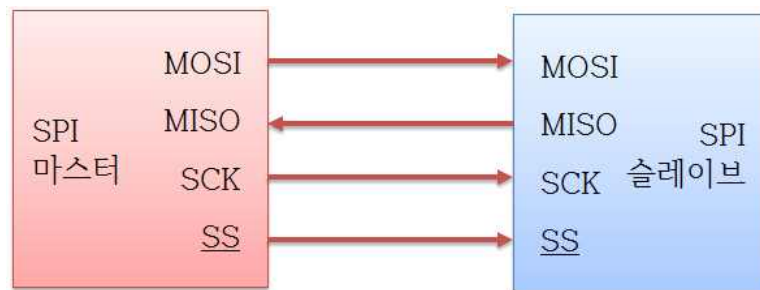


그림 1. 일대일 SPI 연결

그림 1에서 볼 수 있듯이 SPI 연결은 4개 선의 연결을 필요로 한다. SCK (Serial Clock)는 직렬 클록으로 SPI가 동기 전송 방식을 사용하므로 동기화를 위한 클록 전송을 위해 필요하다. MOSI는 Master Out Slave In의 약어로 마스터 장치에서 슬레이브 장치로 데이터를 전송하기 위해 사용되며, MISO는 Master In Slave Out의 약어로 그 반대 방향의 데이터 전송을 위해 사용된다. SS는 Slave Select의 약어로 여러 개의 슬레이브 장치 중 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.

1) SPI 통신은 다대다 구성이 가능하다. 하지만 동기화 방법이 복잡해지므로 거의 사용하지 않는다.

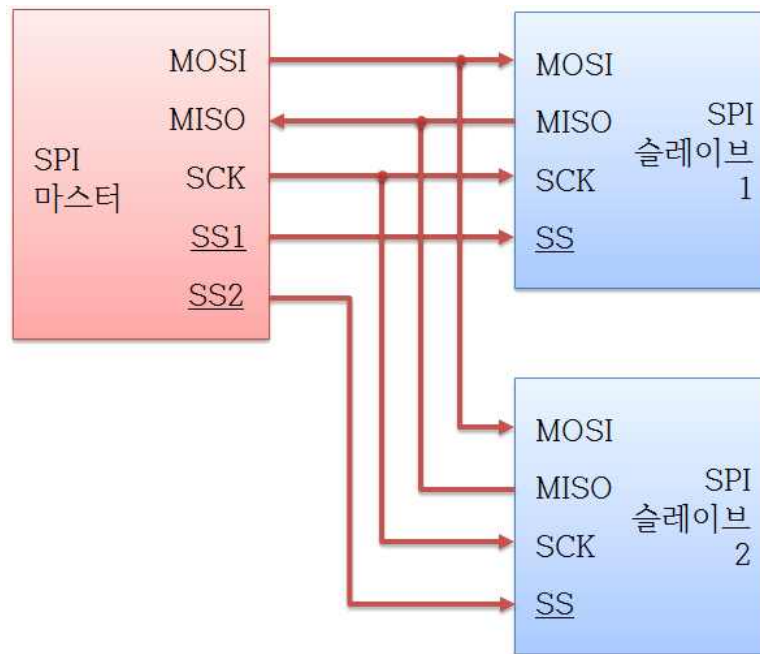


그림 2. 일대다 SPI 연결

그림 2는 하나의 마스터 장치에 2개의 슬레이브 장치가 연결된 예를 보인 것이다. 모든 슬레이브 장치는 마스터로부터의 SCK, MOSI, MISO 라인을 공유하고 있다. 즉, 마스터 장치의 데이터가 모든 슬레이브 장치로 전달된다. 누가 데이터를 받을 것인가? 바로 SS 라인에 의해 결정된다. 데이터를 받을 슬레이브 장치의 SS만 LOW 상태에 있고 나머지 슬레이브 장치의 SS는 HIGH 상태에 있게 된다. 즉, 일대다 연결이 가능하지만 특정 순간에는 일대일 통신만이 가능하다. 또한 슬레이브 장치의 수가 늘어날수록 전용의 SS 라인이 그에 맞게 증가하여야 한다.

SPI의 데이터 전송에 대해 좀 더 살펴보자. SPI의 데이터 전송은 송신과 수신에 항상 동시에 일어나는 특징을 가진다. UART 통신에서도 송신과 수신에 동시에 일어날 수 있지만 ‘항상’ 그런 것은 아니다. 이는 UART 통신이 비동기 방식으로 동작하기 때문이다. SPI는 동기 방식으로 동작하므로 클럭이 필요하며 이 클럭은 마스터 장치에 의해 공급된다. 슬레이브 장치에서 마스터 장치로 데이터를 보내고자 하는 경우에도 마스터 장치의 클럭을 기준으로 데이터를 전송해야 하지만 슬레이브 장치에서는 마스터 장치의 클럭을 알기 어려운 문제점이 있다. SPI에서는 이러한 동기화 클럭의 문제점을 송신과 수신에 항상 동시에 진행되도록 함으로써 해결하고 있다. 어렵게 들릴 수도 있지만 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

이브 장치의 큐에 저장되어 있다고 가정해 보자.

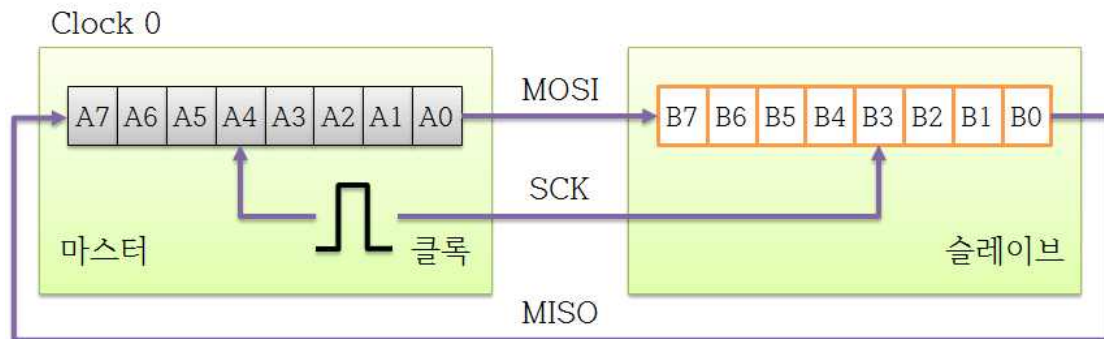


그림 3. 데이터 전송 준비

데이터 전송은 전적으로 마스터 장치가 책임지므로 마스터 장치가 제공하는 클록에 의해 데이터 전송은 이루어진다. 데이터가 준비된 상태에서 하나의 클록이 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 오른쪽 원형 이동 연산을 수행한 것처럼 동작한다. 즉, 원격으로 연결된 원형 큐처럼 동작한다.

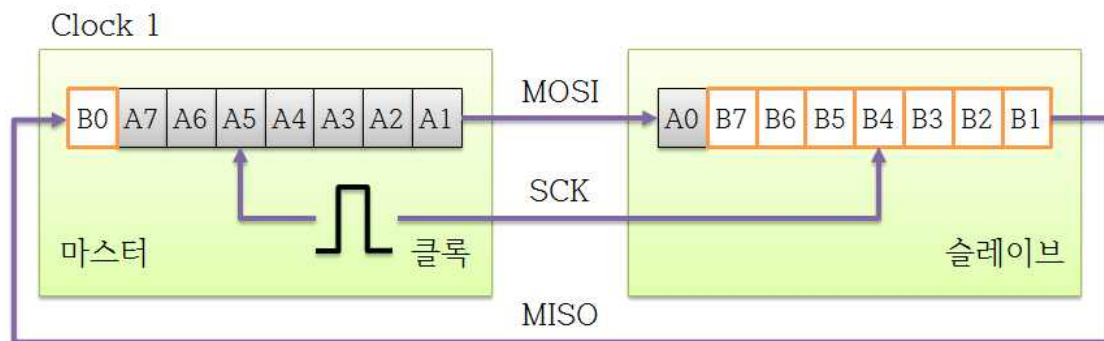


그림 4. 1비트 데이터 전송

유사하게 8개 클록이 발생하면 마스터 장치의 1바이트 데이터는 슬레이브 장치로 전달되고 슬레이브 장치의 1바이트 데이터는 마스터 장치로 전달되는 결과를 얻는다.

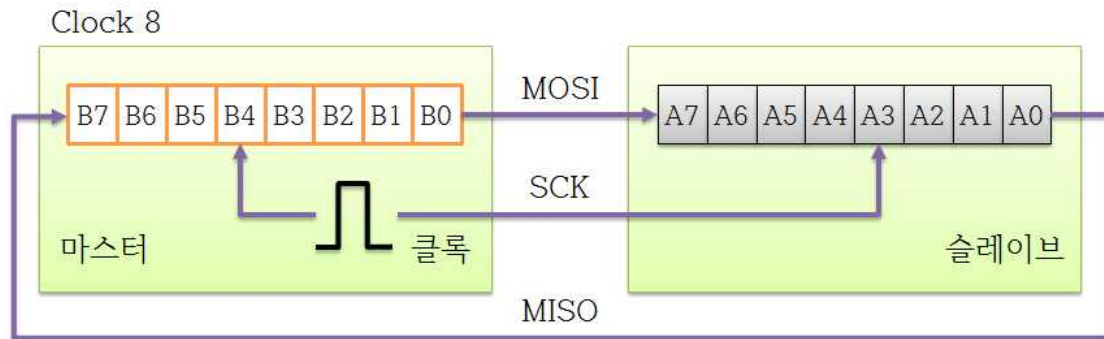


그림 5. 1바이트 데이터 전송

이처럼 마스터 장치와 슬레이브 장치는 항상 데이터를 동시에 주고받음으로써 동기화 문제를 해결하고 있다. 이러한 상황에서 유의할 점 한 가지는 클록의 역할이다. 기본적으로 클록은 데이터가 전달되는 시점을 알려주어 샘플링이 발생하는 시간을 결정하기 위해 사용된다. 이를 위해 클록의 극성(Clock Polarity, CPOL)과 클록 위상(Clock Phase, CPHA)이 사용된다.

- CPOL : SPI 버스가 유희 상태일 때의 클록 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 6은 CPHA = 0인 경우 데이터가 전송되는 방법을 보여준다. 데이터는 CPOL = 0이면 상승 에지에서, CPOL = 1이면 하강 에지에서 샘플링 된다.

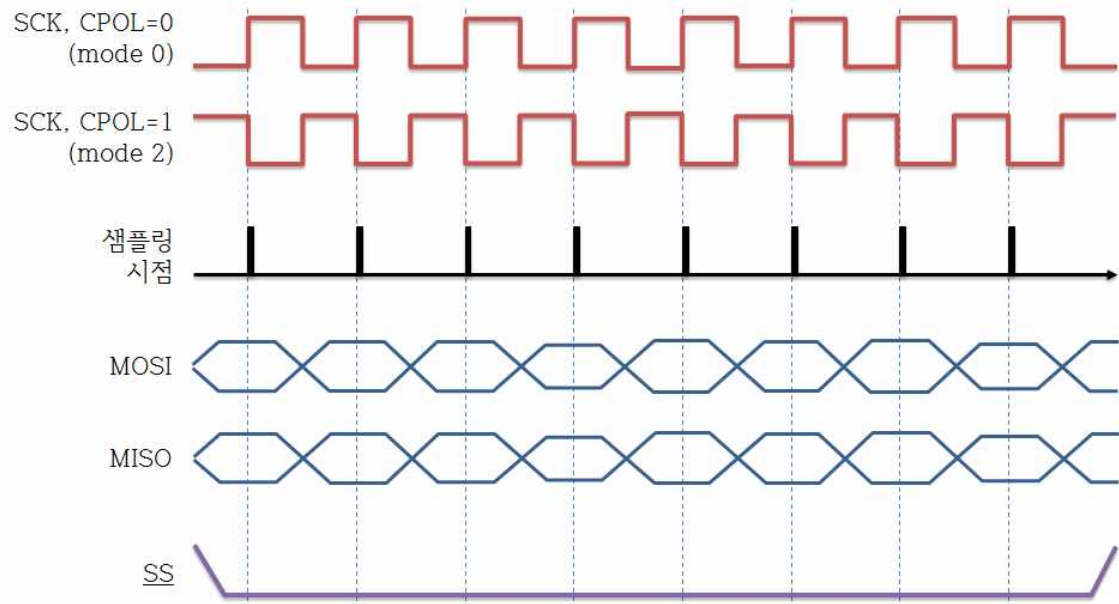


그림 6. CPHA = 0인 경우 데이터 전송 다이어그램

그림 7은 CPHA = 1인 경우 데이터가 전송되는 방법을 보여준다. 데이터는 CPOL = 0이면 하강 에지에서, CPOL = 1이면 상승 에지에서 샘플링 된다.

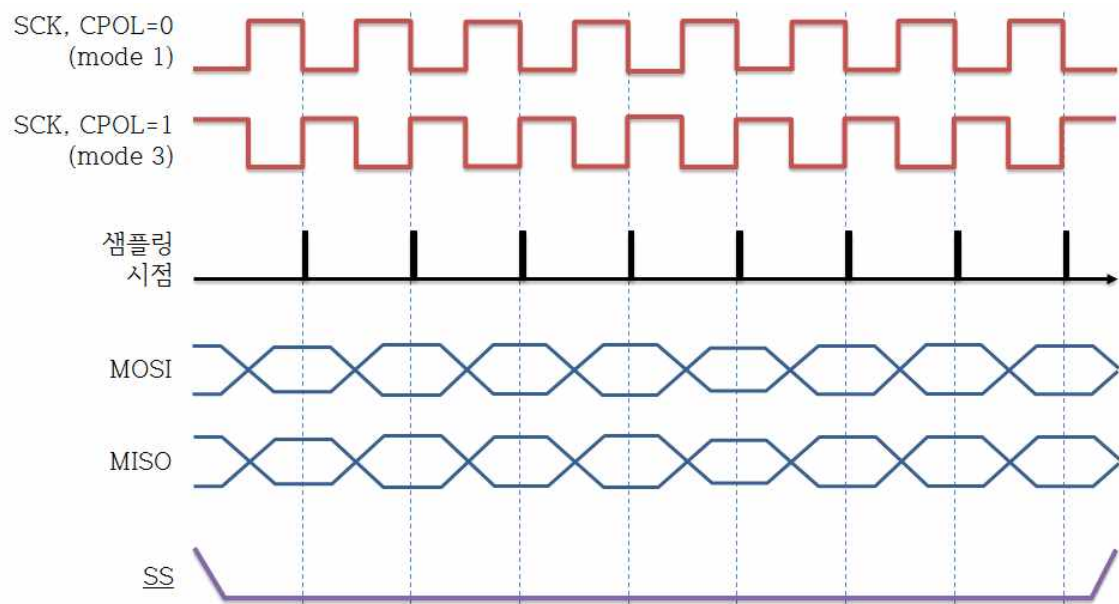


그림 7. CPHA = 1인 경우 데이터 전송 다이어그램

그림 6과 그림 7에서 한 가지 유의해야 할 점은 데이터가 클럭에 따라 샘플링 되는 동안에

SS 라인은 LOW 상태를 유지하고 있다는 점이다. 즉, 선택된 슬레이브만이 데이터를 받을 수 있다.

2. SPI 방식의 EEPROM

이 장에서 사용할 SPI 장치는 EEPROM이다. ATmega328에도 1KByte 크기의 EEPROM이 포함되어 있지만 1KByte는 그리 크지 않으므로 데이터 로깅 등을 위해서 별도의 외부 기억 장치를 사용하는 경우를 흔히 볼 수 있으며, 이 중 EEPROM은 읽고 쓰는 속도가 느린 단점은 있지만 바이트 단위의 간단한 입출력이 가능하며 비휘발성 메모리로 전원이 꺼진 이후에도 기록한 내용이 보존되므로 널리 사용된다. SPI 방식의 EEPROM 중에서도 이 장에서는 마이크로칩(Microchip)에서 제작하여 판매하는 25LCn EEPROM을 사용한다. 'n'은 EEPROM의 크기를 나타내며 1K bit에서 1M bit까지 다양한 크기의 EEPROM이 존재한다. 25LCn EEPROM은 읽고 쓰기에 소요되는 시간이 5ms이며 백만번의 읽고 쓰기를 보장한다. 메모리 크기에 따라 페이지 크기와 최대 동작 속도에서 약간의 차이가 있다. 이 장에서는 1K bit 메모리 크기를 가지는 8핀 DIP 타입의 25LC010A를 사용한다. 25LC010A 칩의 핀 배치는 그림 8과 같다.

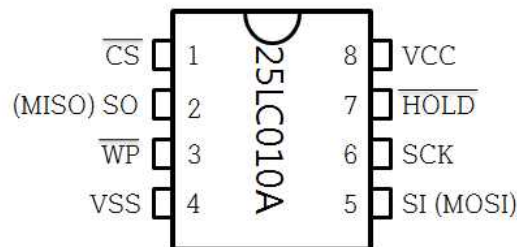


그림 8. 25LC010A 핀 배치

각 핀의 설명은 표 1과 같으며 데이터시트에서는 MISO와 MOSI 대신 SO와 SI로 표시하고 있으므로 혼돈하지 않도록 한다.

핀 번호	이름	설명	비고
1	$\overline{CS}$	Chip Select	SS (Slave Select)
2	SO	Serial Data Output	MISO
3	$\overline{WP}$	Write Protect	
4	VSS	Ground	
5	SI	Serial Data Input	MOSI
6	SCK	Serial Clock	
7	$\overline{HOLD}$	Hold Input	
8	VCC		

표 1. 25LC010A 핀 설명

아두이노 우노는 SPI 연결을 위해 사용할 수 있는 핀이 표 2와 같이 정의되어 있다. 이 중 MOSI, MISO, SCK는 변경할 수 없지만, SS의 경우 기본적으로 사용하는 핀이 디지털 10번 핀이라는 의미이며 다른 핀도 사용 가능하다. 여기서는 디폴트값인 디지털 10번 핀을 사용한다.

SPI 신호	Arduino UNO의 핀	비고
MOSI	디지털 11	PB3
MISO	디지털 12	PB4
SCK	디지털 13	PB5
SS	디지털 10	PB2

표 2. SPI 연결 핀

표 1과 표 2를 참고하여 25LC010A를 그림 9와 같이 ATmega328에 연결한다. $\overline{WP}$ 는 쓰기 금지 기능으로 GND가 가해지면 EEPROM에 내용을 기록할 수 없도록 해준다. 여기서는 VCC에 연결하여 항상 쓰기가 가능하도록 한다. $\overline{HOLD}$ 는 EEPROM과의 통신을 일시 중지시키는 기능을 하는 핀이다. $\overline{WP}$ 가 GND에 연결된 경우 쓰기는 금지되지만 읽기는 가능한 반면 $\overline{HOLD}$ 가 GND에 연결된 경우 읽기와 쓰기 모두 정상적으로 동작하지 않는다.

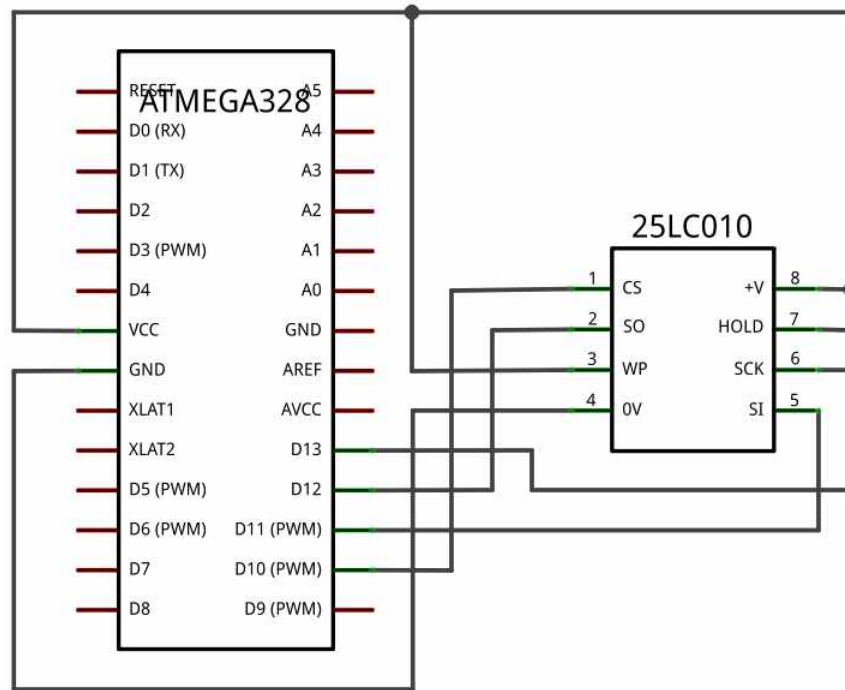


그림 9. 25LC010A 연결 회로도

EEPROM의 기본적인 동작은 바이트 단위의 읽기와 쓰기이다. 먼저 읽기 과정을 살펴보자. EEPROM에서 1바이트의 데이터를 읽기 위해서는 먼저 EEPROM 칩을 선택하여야 하며 이는 $\overline{CS}$ 핀에 LOW를 가함으로써 이루어진다. 이후 읽기에 해당하는 8비트 명령어를 전송하여야 하며, 읽기에 해당하는 8비트 명령어 값은 0b00000011이다.²⁾ 명령어 전송 이후에는 8비트의 주소를 EEPROM으로 전송한다.³⁾ 주소까지 전송되면 EEPROM은 해당 메모리 번지의 바이트 값을 마이크로컨트롤러로 전송하며, 이 때 마스터와 슬레이브는 원형 큐와 같은 방식으로 동작하므로 임의의 바이트 값을 슬레이브로 전송하여야 슬레이브로부터 바이트 값을 읽어 들일 수 있음에 유의하여야 한다. 전송이 끝나면 $\overline{CS}$ 핀에 HIGH를 가함으로써 읽기 동작을 끝낸다. 코드 1은 슬레이브로부터 1바이트 데이터를 읽어 반환하는 함수의 예이다.

코드 1. 바이트 데이터 읽기 함수

2) 명령어에 해당하는 이진수 값은 데이터시트를 참고하도록 한다.

3) 8비트 주소로 구별할 수 있는 메모리는 최대 256바이트이다. 512바이트 이상의 메모리를 가지는 EEPROM의 경우에는 8비트 주소로 어드레싱이 불가능하다. EEPROM은 바이트 단위의 데이터 전송을 기본으로 하므로 이 장에서는 SPI의 동작 방식을 이해하기 위해 8비트 주소를 사용하는 작은 크기의 EEPROM을 선택하였으며, 512바이트 이상의 크기를 가지는 EEPROM을 사용하고자 한다면 데이터시트를 참고하여 예제 코드들, 특히 주소 관련 부분을 수정하여야 한다.

```

#define SPI_SS                PB2
#define SPI_MOSI              PB3
#define SPI_MISO              PB4
#define SPI_SCK               PB5

#define EEPROM_Select()       PORTB &= ~(1 << SPI_SS)    // LOW
#define EEPROM_DeSelect()     PORTB |= (1 << SPI_SS)     // HIGH

uint8_t EEPROM_readByte(uint8_t address) {
    EEPROM_Select();                // EEPROM 선택
    EEPROM_changeByte(EEPROM_READ); // 읽기 명령어 전송
    EEPROM_sendAddress(address);    // 메모리 주소 전송
    // 마스터에서 바이트 값을 전송하여야 슬레이브로부터 바이트 값을 받을 수 있다.
    // 전송하는 값은 의미가 없으므로 0을 전송한다.
    EEPROM_changeByte(0);
    EEPROM_DeSelect();              // EEPROM 선택 해제

    return SPDR;
}

```

먼저 SPI 통신에 사용할 4개의 핀을 표 2에 따라 정의하고, EEPROM_Select 함수는 SS 핀에 LOW를 가하는 매크로 함수로, EEPROM_DeSelect 함수는 SS 핀에 HIGH를 가하는 매크로 함수로 정의하였다. EEPROM_changeByte와 EEPROM_sendAddress 함수는 모두 1바이트 데이터를 교환하는 기능을 한다. 다만 전달하는 데이터의 의미가 데이터와 주소로 다르므로 별도의 함수로 작성하였다.

코드 2. 바이트 데이터 교환 함수

```

void EEPROM_changeByte(uint8_t byte) {
    SPDR = byte;                // 데이터 전송 시작
    loop_until_bit_is_set(SPSR, SPIF); // 전송 완료 대기
}

void EEPROM_sendAddress(uint8_t address) {

```

```
EEPROM_changeByte(address);
}
```

SPI와 관련된 레지스터는 제어를 위한 SPCR (SPI Control Register), 상태 검사를 위한 SPSR (SPI Status Register), 전송되는 데이터를 위한 SPDR (SPI Data Register)의 3개가 있다. SPDR 레지스터의 구조는 그림 10과 같다. SPDR 레지스터에 데이터를 저장하면 데이터 전송은 자동으로 시작된다.

비트	7	6	5	4	3	2	1	0
	SPDR							
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초깃값	x	x	x	x	x	x	x	x

그림 11. SPDR 레지스터 구조

SPSR 레지스터의 구조는 그림 12와 같다.

비트	7	6	5	4	3	2	1	0
	SPIF	WCOL	-	-	-	-	-	SPI2X
읽기/쓰기	R	R	R	R	R	R	R	R/W
초깃값	0	0	0	0	0	0	0	0

그림 11. SPSR 레지스터 구조

비트	이름	설명
7	SPIF	SPI 인터럽트 플래그 (SPI Interrupt Flag) : 데이터 전송이 완료되면 세트된다. 전역 인터럽트 허용 비트가 세트되어 있고 SPCR 레지스터의 인터럽트 허용 비트인 SPIE 비트가 세트되어 있을 때, SPIF 비트가 1이 되면 SPI 인터럽트가 발생한다.
6	WCOL	쓰기 충돌 플래그 (Write COLision flag) : 데이터가 전송되고 있는 도중 SPDR 레지스터에 데이터를 쓰는 경우 세트된다, 즉, 동일한 레지스터에 동시에 데이터를 쓰려는 경우 충돌을 알려주기 위해 사용된다.
0	SPI2X	2배속 모드 (Double SPI Speed Bit) : SPI2X 비트가 세트되어 있는 경우 전송 속도는 2배가 된다. SPCR 레지스터의 SPRn 비트와 함께 사용된다.

표 3. SPSR 레지스터 비트

EEPROM_changeByte에서는 SPDR 레지스터에 전송할 값을 대입함으로써 전송을 시작한다. 데이터 전송 후에는 SPSR 레지스터의 SPIF 비트가 세트되기를, 즉, 데이터 전송이 완료되기를 기다린 후 전송이 완료되면 반환한다.

쓰기 과정을 살펴보자. 데이터를 쓰기 위해서는 먼저 EEPROM으로 WREN (Write Enable) 명령을 전송하여 쓰기 가능 래치를 설정하여야 한다. WREN 명령을 전송할 때는 반드시 전송 전에 $\overline{CS}$ 핀에 LOW를 가하고, 전송 후에는 HIGH를 가해주어야 래치가 정상적으로 설정된다. 이후 데이터를 쓰기 위한 과정은 읽기 과정과 유사하다. 먼저 $\overline{CS}$ 핀에 LOW를 가하여 EEPROM 칩을 선택하고, 쓰기에 해당하는 8비트 명령어 값인 0b00000010를 전송한다. 다음은 데이터를 기록할 8비트 주소값과 기록할 데이터 값을 전송하고 $\overline{CS}$ 핀에 HIGH를 가함으로써 쓰기를 끝낸다. 코드 3은 슬레이브로 1바이트 데이터를 쓰는 함수의 예이다.

코드 3. 바이트 데이터 쓰기 함수	
#define EEPROM_WRITE	0b00000010
#define EEPROM_WREN	0b00000110
<pre>void EEPROM_writeEnable(void) { EEPROM_Select(); // Slave Select를 LOW로 EEPROM_changeByte(EEPROM_WREN); // 쓰기 가능하도록 래치 설정 }</pre>	

```

EEPROM_DeSelect();                // Slave Select를 HIGH로

}

void EEPROM_writeByte(uint8_t address, uint8_t data) {
    EEPROM_writeEnable();          // 쓰기 가능 모드로 설정

    EEPROM_Select();               // EEPROM 선택
    EEPROM_changeByte(EEPROM_WRITE); // 쓰기 명령 전송
    EEPROM_sendAddress(address);    // 주소 전송
    EEPROM_changeByte(data);        // 데이터 전송
    EEPROM_DeSelect();              // EEPROM 선택 해제

    // 쓰기가 완료될 때까지 대기
    while (EEPROM_readStatus() & _BV(EEPROM_WRITE_IN_PROGRESS));
}

```

코드 3에서는 마지막 문장은 EEPROM에서 실제 쓰기 동작이 완료될 때까지 대기하는 문장이다. EEPROM의 상태를 알아내기 위해서는 해당 명령어인 RDSR (Read Status Register) 명령을 전송하면 된다. 반환되는 값의 구조는 그림 12와 같다.

비트	7	6	5	4	3	2	1	0
	-	-	-	-	BP1	BP0	WEL	WIP
읽기/쓰기	-	-	-	-	R/W	R/W	R	R

그림 11. EEPROM의 상태 레지스터 구조

비트	이름	설명
3	BP1	쓰기 금지 (Write Protection) : 쓰기 금지된 블록을 나타낸다.
2	BP0	
1	WEL	쓰기 가능 래치 (Write Enable Latch) : 1인 경우 쓰기 가능 래치가 쓰기 가능하도록 설정되어 있음을 나타낸다.
0	WIP	쓰기 진행 중 (Write In Process) : 1인 경우 EEPROM이 현재 쓰기 동작을 수행 중임을 나타낸다.

표 4. EEPROM의 상태 레지스터 비트

코드 4는 EEPROM의 상태 레지스터 값을 읽어오는 함수의 예이다.

코드 4. 상태 레지스터 읽기 함수

```
#define EEPROM_RDSR          0b00000101

uint8_t EEPROM_readStatus(void) {
    EEPROM_Select();           // EEPROM 선택
    EEPROM_changeByte(EEPROM_RDSR); // 상태 레지스터 읽기 명령 전송
    EEPROM_changeByte(0);      // 상태 레지스터 값 읽기
    EEPROM_DeSelect();         // EEPROM 선택 해제

    return SPDR;
}
```

EEPROM은 비휘발성 메모리로 데이터를 저장하기에는 좋지만 쓰기 속도가 느리다는 단점이 있다. 이를 개선할 수 있는 방법이 페이지 단위로 데이터를 기록하는 방법이다. 25LC010A은 16바이트 크기의 페이지 크기를 갖는다. EEPROM_writeByte 함수는 1바이트 데이터를 기록하기 위해 주소를 지정하지만, 주소를 한번 지정하고 나서는 페이지 크기만큼의 데이터를 연속적으로 전송하여 연속된 주소에 저장할 수 있다. 이 때 한 가지 주의할 점은 페이지 단위의 데이터 기록에서 시작 번지는 XXXX 0000₂이며 페이지의 끝은 XXXX 1111₂이라는 점이다. 즉, 연속된 번지에 데이터가 기록되기는 하지만 페이지 경계를 넘어가지는 못하며 페이지 끝에 도달하면 다시 페이지의 처음으로 되돌아간다. 예를 들어, 0001 0000₂번지를 지정하고 2바이트를 전송하면 0001 0000₂번지와 0001 0001₂번지에 데이터가 기록되지만 0000 1111₂번지를 지정하고 2바이트를 전송하면 0000 1111₂번지와 0000 0000₂번지에 데이터가 기록된다. 코드 5는 페이지 단위의 쓰기를 통해 EEPROM 전체를 0으로 초기화하는 함수의 예이다.

코드 5. EEPROM 초기화 함수

```
#define EEPROM_PAGE_SIZE      16 // 페이지 크기 (byte)
#define EEPROM_TOTAL_BYTE     128 // (byte)
```

```

void EEPROM_eraseAll(void) {
    uint8_t i;
    uint16_t pageAddress = 0;

    while (pageAddress < EEPROM_TOTAL_BYTE){
        EEPROM_writeEnable();           // 쓰기 가능 모드로 설정
        EEPROM_Select();                 // EEPROM 선택
        EEPROM_changeByte(EEPROM_WRITE); // 쓰기 명령 전송
        EEPROM_sendAddress(pageAddress); // 페이지 시작 주소 전송
        for (i = 0; i < EEPROM_PAGE_SIZE; i++) // 페이지 단위 데이터 전송
            EEPROM_changeByte(0);
        EEPROM_DeSelect();               // EEPROM 선택 해제

        pageAddress += EEPROM_PAGE_SIZE; // 페이지 변경
        // 쓰기 완료 대기
        while(EEPROM_readStatus() & _BV(EEPROM_WRITE_IN_PROGRESS));
    }
}

```

마지막으로 SPI 통신을 초기화하는 방법을 살펴보자. SPI 통신 설정은 SPCR 레지스터를 통해 이루어진다. SPCR 레지스터의 구조는 그림 12와 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 12. SPCR 레지스터 구조

비트	이름	설명
7	SPIE	SPI 인터럽트 허용 (SPI Interrupt Enable) : 1로 설정된 경우, SPSR 레지스터의 SPIF 비트가 세트되고 전역 인터럽트가 허용된 경우 SPI 인터럽트가 발생하도록 허용한다.
6	SPE	SPI 활성화 (SPI Enable) : 1로 설정된 경우 SPI 통신이 가능하도록 한다.
5	DORD	데이터 순서 (Data Order) : 1인 경우 데이터의 LSB가 먼저 전송되고, 0인 경우 데이터의 MSB가 먼저 전송된다.
4	MSTR	마스터/슬레이브 선택 (Master/Slave Select) : 1인 경우 마스터 모드로 설정되고, 0인 경우 슬레이브 모드로 설정된다.
3	CPOL	클록 극성 (Clock Polarity)
2	CPHA	클록 위상 (Clock Phase)
1	SPR1	클록 설정 (SPI Clock Rate Select) : 마스터로 설정된 경우 클록을 설정하기 위해 사용된다. SPSR 레지스터의 SPI2X 비트와 함께 사용된다.
0	SPR0	

표 5. SPCR 레지스터 비트

SPRn 비트와 SPI2X 비트 설정에 따른 SPI 클록은 표 6과 같다.

SPSR SPI2X	SPCR		설명
	SPR1	SPR0	
0	0	0	$f_{osc}/4$
0	0	1	$f_{osc}/16$
0	1	0	$f_{osc}/64$
0	1	1	$f_{osc}/128$
1	0	0	$f_{osc}/2$
1	0	1	$f_{osc}/8$
1	1	0	$f_{osc}/32$
1	1	1	$f_{osc}/64$

표 6. SPI 클록 설정

코드 6은 SPI를 설정하는 함수의 예이다.

코드 6. SPI 초기 설정 함수

```

void SPI_Init(void) {
    DDRB |= (1 << SPI_SS);           // SS 핀을 출력으로 설정
    // SS핀은 HIGH로 설정하여 EEPROM이 선택되지 않은 상태로 시작
    PORTB |= (1 << SPI_SS);

    DDRB |= (1 << SPI_MOSI);          // MOSI 핀을 출력으로 설정
    DDRB &= ~(1 << SPI_MISO);         // MISO 핀을 입력으로 설정
    DDRB |= (1 << SPI_SCK);           // SCK 핀을 출력으로 설정

    SPCR |= (1 << MSTR);               // 마스터 모드
    SPCR |= (1 << SPE);               // SPI 활성화
}

```

25LC010A는 최대 10MHz로 동작할 수 있으므로 코드 6에서 분주율은 디폴트값인 클록의 1/4, 즉, 4MHz를 사용하였다. 25LC010A의 경우 CPOL = 0, CPHA = 0인 모드 0에서 동작하므로 클록의 극성과 위상 역시 디폴트값을 사용하였다. 코드 7과 코드 8은 25LC010을 사용하기 위한 함수를 별도의 헤더 및 소스 파일로 작성한 것이다.

코드 7. 25LC010.h

```

#ifndef _25LC010_H_
#define _25LC010_H_

#include <avr/io.h>

// SPI를 위한 핀 정의
#define SPI_SS          PB2
#define SPI_MOSI         PB3
#define SPI_MISO         PB4
#define SPI_SCK          PB5

#define EEPROM_Select()  PORTB &= ~(1 << SPI_SS)    // LOW
#define EEPROM_DeSelect() PORTB |= (1 << SPI_SS)    // HIGH

```

```

// 명령어
#define EEPROM_READ          0b00000011    // 읽기
#define EEPROM_WRITE         0b00000010    // 쓰기
#define EEPROM_WREN          0b00000110    // 쓰기 허용
#define EEPROM_RDSR          0b00000101    // 상태 레지스터 읽기

#define EEPROM_WRITE_IN_PROGRESS      0

#define EEPROM_PAGE_SIZE      16          // 페이지 크기 (byte)
#define EEPROM_TOTAL_BYTE    128          // (byte)

void SPI_Init(void);                    // SPI 초기화
void EEPROM_changeByte(uint8_t byte);   // 1바이트 데이터 교환
void EEPROM_sendAddress(uint8_t address); // 8비트 주소 전송
uint8_t EEPROM_readStatus(void);        // EEPROM 상태 레지스터 읽기
void EEPROM_writeEnable(void);          // 쓰기 허용 래치 설정
uint8_t EEPROM_readByte(uint8_t address); // 1바이트 데이터 읽기
void EEPROM_writeByte(uint8_t address, uint8_t byte); // 1바이트 데이터 쓰기
void EEPROM_eraseAll(void);             // EEPROM을 0으로 초기화

#endif

```

코드 8. 25LC010.c

```

#include "25LC010.h"

void SPI_Init(void) {
    DDRB |= (1 << SPI_SS);           // SS 핀을 출력으로 설정
    // SS핀은 HIGH로 설정하여 EEPROM이 선택되지 않은 상태로 시작
    PORTB |= (1 << SPI_SS);

    DDRB |= (1 << SPI_MOSI);          // MOSI 핀을 출력으로 설정
    DDRB &= ~(1 << SPI_MISO);         // MISO 핀을 입력으로 설정
    DDRB |= (1 << SPI_SCK);           // SCK 핀을 출력으로 설정
}

```

```
SPCR |= (1 << MSTR);           // 마스터 모드
SPCR |= (1 << SPE);            // SPI 활성화
}

void EEPROM_changeByte(uint8_t byte) {
    SPDR = byte;                // 데이터 전송 시작
    loop_until_bit_is_set(SPSR, SPIF); // 전송 완료 대기
}

void EEPROM_sendAddress(uint8_t address) {
    EEPROM_changeByte(address);
}

uint8_t EEPROM_readByte(uint8_t address) {
    EEPROM_Select();            // EEPROM 선택
    EEPROM_changeByte(EEPROM_READ); // 읽기 명령어 전송
    EEPROM_sendAddress(address);   // 메모리 주소 전송
    // 마스터에서 바이트 값을 전송하여야 슬레이브로부터 바이트 값을 받을 수 있다.
    // 전송하는 값은 의미가 없으므로 0을 전송한다.
    EEPROM_changeByte(0);
    EEPROM_DeSelect();          // EEPROM 선택 해제

    return SPDR;
}

void EEPROM_writeEnable(void) {
    EEPROM_Select();            // Slave Select를 LOW로
    EEPROM_changeByte(EEPROM_WREN); // 쓰기 가능하도록 래치 설정
    EEPROM_DeSelect();          // Slave Select를 HIGH로
}

void EEPROM_writeByte(uint8_t address, uint8_t data) {
    EEPROM_writeEnable();        // 쓰기 가능 모드로 설정
```

```

EEPROM_Select();                // EEPROM 선택
EEPROM_changeByte(EEPROM_WRITE); // 쓰기 명령 전송
EEPROM_sendAddress(address);    // 주소 전송
EEPROM_changeByte(data);        // 데이터 전송
EEPROM_DeSelect();              // EEPROM 선택 해제

// 쓰기가 완료될 때까지 대기
while (EEPROM_readStatus() & _BV(EEPROM_WRITE_IN_PROGRESS));
}

uint8_t EEPROM_readStatus(void) {
    EEPROM_Select();                // EEPROM 선택
    EEPROM_changeByte(EEPROM_RDSR); // 상태 레지스터 읽기 명령 전송
    EEPROM_changeByte(0);           // 상태 레지스터 값 읽기
    EEPROM_DeSelect();              // EEPROM 선택 해제

    return SPDR;
}

void EEPROM_eraseAll(void) {
    uint8_t i;
    uint16_t pageAddress = 0;

    while (pageAddress < EEPROM_TOTAL_BYTE){
        EEPROM_writeEnable();        // 쓰기 가능 모드로 설정
        EEPROM_Select();              // EEPROM 선택
        EEPROM_changeByte(EEPROM_WRITE); // 쓰기 명령 전송
        EEPROM_sendAddress(pageAddress); // 페이지 시작 주소 전송
        for (i = 0; i < EEPROM_PAGE_SIZE; i++) // 페이지 단위 데이터 전송
            EEPROM_changeByte(0);
        EEPROM_DeSelect();            // EEPROM 선택 해제

        pageAddress += EEPROM_PAGE_SIZE; // 페이지 변경
        // 쓰기 완료 대기
        while(EEPROM_readStatus() & _BV(EEPROM_WRITE_IN_PROGRESS));
    }
}

```

```
}
```

코드 9는 EEPROM의 0에서 127번지에 0에서 127 까지의 숫자를 기록하고, 이를 다시 읽어서 UART를 통해 컴퓨터로 전송하여 터미널 윈도우에서 확인하는 코드의 예이다.

코드 9. EEPROM에 데이터 읽고 쓰기

```
#define F_CPU 16000000UL

#include <avr/io.h>
#include <string.h>
#include "25LC010.h"

void UART_INIT(void) {
    UCSR0A |= _BV(U2X0);           // 2배속 모드

    UBRR0H = 0x00;                 // 통신 속도 (보율) 설정
    UBRR0L = 207;

    // 비동기, 8비트 데이터, 패리티 없음, 1비트 정지 비트 모드
    UCSR0C |= 0x06;

    UCSR0B |= _BV(RXEN0);          // 송수신 가능
    UCSR0B |= _BV(TXEN0);
}

unsigned char UART_receive(void)   // 1 바이트 수신
{
    while( !(UCSR0A & (1<<RXC0)) ); // 데이터 수신 대기
    return UDR0;
}

void UART_transmit(unsigned char data) // 1 바이트 송신
{
```

```
while( !(UCSR0A & (1<<UDRE0)) );    // 송신 가능 대기
UDR0 = data;                          // 데이터 전송
}

void UART_printString(char *str)      // 문자열 송신
{
    int len = strlen(str);

    for(int i = 0; i < len; i++)
        UART_transmit(str[i]);
}

void UART_print8bitNumber(uint8_t no) // 숫자를 문자열로 변환하여 송신
{
    char numString[4] = "0";
    int i, index = 0;

    if(no > 0){
        for(i = 0; no != 0 ; i++)
        {
            numString[i] = no % 10 + '0';
            no = no / 10;
        }
        numString[i] = '\0';
        index = i - 1;
    }

    for(i = index; i >= 0; i--)
        UART_transmit(numString[i]);
}

int main(void) {
    uint8_t i;

    SPI_Init();                      // SPI 초기화
    UART_INIT();                     // UART 초기화
```

```
for(i = 0; i < 128; i++){                // EEPROM에 쓰기
    EEPROM_writeByte(i, i);
    UART_print8bitNumber(i);
    UART_printString("\n");
}

for(i = 0; i < 128; i++){                // EEPROM에서 읽기
    UART_print8bitNumber(EEPROM_readByte(i));
    UART_transmit('\n');
}
while(1);

return 0;
}
```

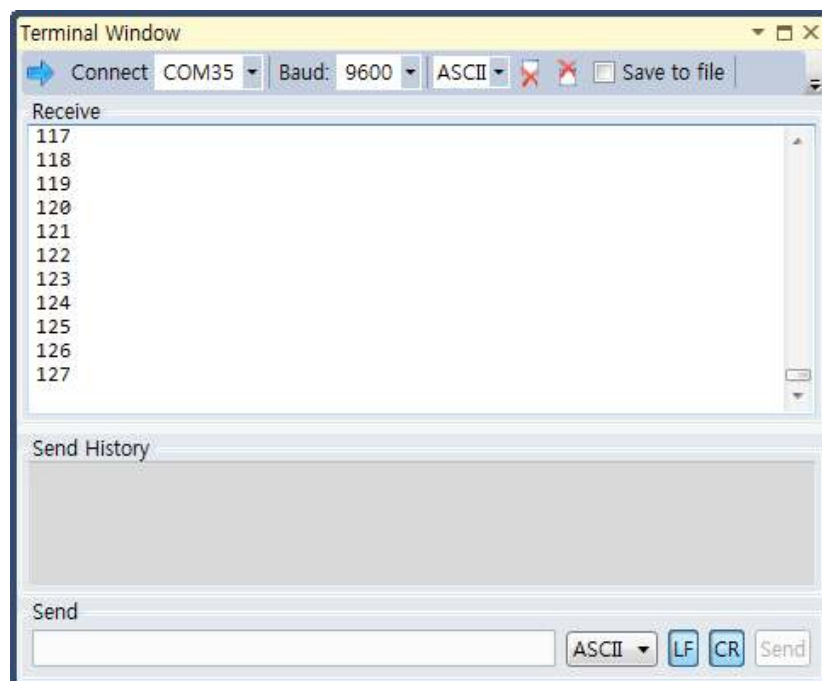


그림 13. 코드 9 실행 결과