ARDUINO

Open Physical Computing Platform

오탈자, 문의 및 보완이 필요한 내용은 hgycap@hotmail.com으로 알려주세요.

Chapter 20. I2C와 SPI 통신을 이용한 아두이노 연결

SPI(Serial Peripheral Interface)는 I2C(Inter-Integrated Circuit)와 더불어 마이크로컨트롤러와 주변 장치 사이에 디지털 정보를 간편하게 전송할 수 있는 방법을 제공하기 위해 만들어진 통신 프로토콜이다. 아두이노 소프트웨어에는 기본적으로 SPI와 I2C를 위한 라이브러리인 SPI 라이브러리와 Wire 라이브러리가 포함되어 있다. SPI와 I2C 중 어느 것을 선택하느냐는 연결하고자 하는 장치에 따라 결정된다. 두 가지 모두 지원하는 장치도 있지만 대부분의 장치는 둘 중 하나만을 지원한다.

I2C의 장점은 신호선 2개만 연결하면 사용할 수 있다는 점이다. 두 신호선을 바탕으로 여러 장치를 연결하여 제어할 수 있으며 신호가 올바르게 수신되었음을 알려주는 승인 신호도 받을 수 있다. 하지만 I2C는 SPI에 비해 속도가 느리다는 단점이 있다. 또한 I2C는 반이중(half-duplex) 방식으로 송신과 수신이 동시에 이루어질 수 없기 때문에 양방향 통신이 필요한 경우에는 전송 속도가 더 느려진다. 또한 신뢰할 수 있는 신호 전송을 위해서는 풀업저항을 사용하여야 한다. 이에 비해 SPI의 장점은 속도가 빠르며 전이중(full-duplex) 방식으로 송신과 수신이 동시에 이루어질 수 있다. 하지만 활성 장치를 선택하기 위해 추가 연결이 필요하므로 여러 장치를 연결하여 사용하기 위해서는 연결선이 증가하는 단점이 있다. 아두이노에서는 일반적으로 고속의 데이터 전송이 필요한 경우에는 SPI를 사용하며 많은 데이터 전송을 요구하지 않는 센서의 경우 I2C가 사용된다. 한 가지 주의할 점은 SPI나 I2C는 통신 방법만을 정의할 뿐 전송되는 데이터는 장치에 따라 달라진다는 점이다. 즉, 동일한 데이터가 전송된다고 하여도 송수신하는 장치에서의 의미는 전혀 달라질 수 있으므로 장치에 따른 데이터의 의미는 datasheet를 참고하여야 한다.

1. I2C

I2C는 필립스에서 저속의 주변 기기를 연결하기 위해 개발한 규격으로 SCL(serial clock)과 SDA(serial data) 두 개의 연결을 가진다. 표준 Arduino UNO 보드에서는 SCL을 위해 아날로그 5번 핀을, SDA를 위해 아날로그 4번 핀을 사용하며 Arduino Mega의 경우 SCL을 위해 디지털 20번 핀을, SDA를 위해 디지털 21번 핀을 사용한다.

I2C 버스의 한 쪽에는 유일한 마스터(master) 장치가 연결되어 다른 슬레이브(slave) 장치들과의 정보 전송을 제어하게 되며 일반적으로 아두이노가 마스터의 역할을 수행한다. 슬레이브 장치들은 7비트의 고유 주소에 의해 식별되므로 최대 $2^7 = 128$개의 슬레이브 장치가 연결될

수 있다. 그림 1은 I2C 버스를 통해 여러 개의 슬레이브 장치들이 연결된 예를 보여주고 있다.

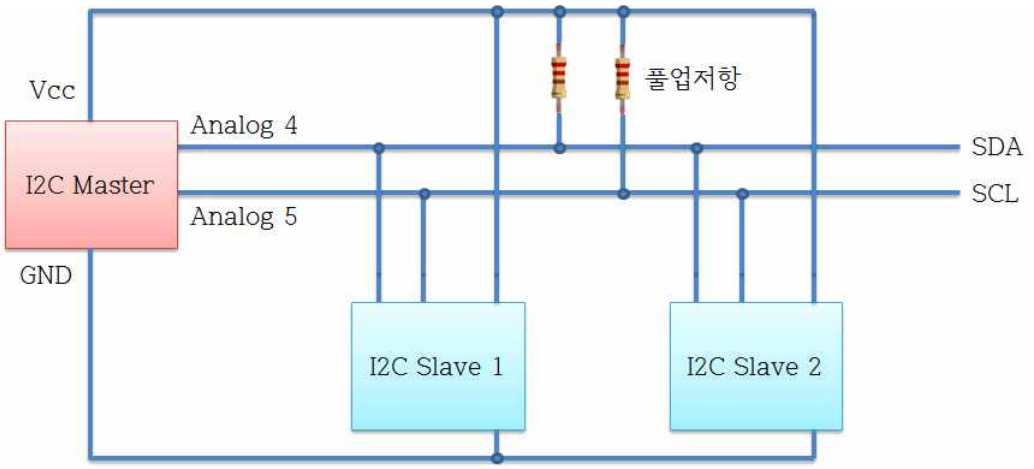

그림 1. I2C 연결

2. I2C를 이용한 아두이노간 통신

두 대의 아두이노를 SPI 통신을 이용하여 연결해 보자. 두 대의 아두이노를 먼저 그림 2에서와 같이 연결한다.

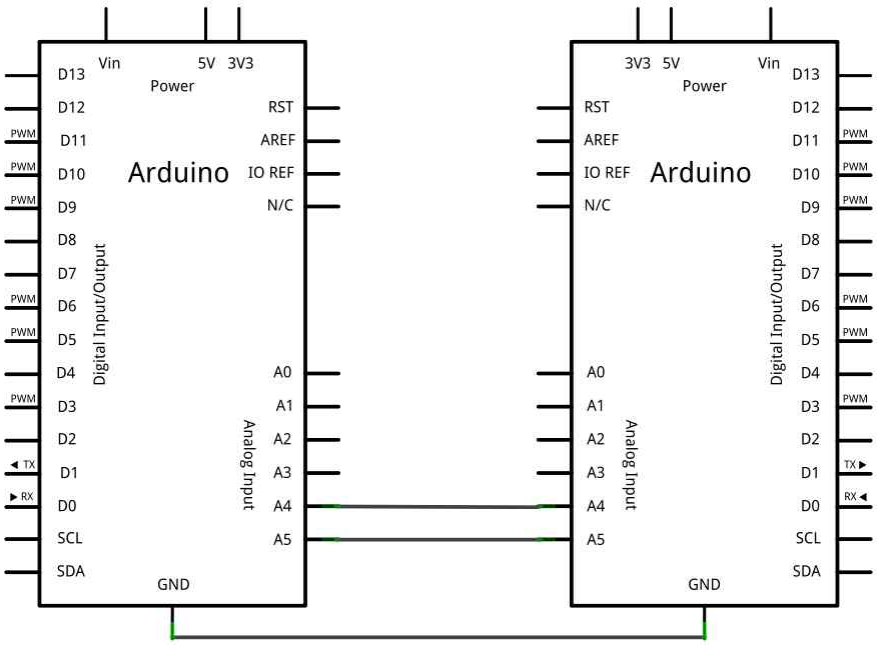

그림 2. 두 대의 아두이노를 I2C 통신을 통해 연결한 회로도

I2C 통신은 클록과 데이터를 위한 두 개의 핀 만을 연결하면 된다. 마스터는 슬레이브로 1초 간격으로 requestFrom 함수로 1바이트 크기의 데이터를 요구하고 데이터를 수신하여 Serial 로 출력한다. 마스터는 데이터 요구를 위해 슬레이브의 주소를 알고 있어야 하며 슬레이브 주소는 4번으로 설정하였다.

코드 1. Master

```
#include <Wire.h>

#define SLAVE 4                    // 슬레이브 주소

void setup() {
  Wire.begin();                    // I2C 통신을 위한 Wire 라이브러리 초기화
  Serial.begin(9600);              // 직렬 통신 초기화
}

void loop() {
  i2c_communication();             // 슬레이브로 데이터 요구 및 수신 데이터 처리
  delay(1000);                     // 1초 대기
}
void i2c_communication() {
  Wire.requestFrom(SLAVE, 1);      // 1 바이트 크기의 데이터 요구

  char c = Wire.read();            // 수신 데이터 읽기
  Serial.println(String(c, DEC));  // 수신 데이터 출력
}
```

슬레이브는 Wire 라이브러리를 초기화할 때 마스터에서 식별 가능한 주소를 지정해주어야 한다. 또한 마스터에서 데이터 전송 요구가 발생한 경우 이를 처리하는 핸들러 함수를 onRequest 함수를 통해 등록해주어야 한다.

코드 2. Slave

```
#include <Wire.h>
#define SLAVE 4

byte count = 0;           // 마스터로 송신할 카운터 데이터

void setup() {
  Wire.begin(SLAVE);      // Wire 라이브러리 초기화. 주소를 지정해야 한다.
  // 마스터의 데이터 전송 요구가 있을 때 처리할 함수 등록
  Wire.onRequest(sendToMaster);
}

void loop () {
}

void sendToMaster() {
  // 카운터 값을 증가시키고 마스터로 전송
  Wire.write(++count);
}
```

그림 3. 코드 1 실행 결과

3. SPI

SPI는 I2C와 다르게 송신과 수신을 위한 별도의 연결선인 MOSI(Master Out Slave In)와 MISO(Master In Slave Out) 그리고 클록(SCLK)이 존재한다. IC2에서는 특정 슬레이브를 지정하기 위해 소프트웨어적인 주소를 사용하는 것과 달리 SPI는 하드웨어적인 연결인 SS(Slave Select)가 존재하여 총 4개의 연결선을 가진다. MOSI, MISO, SCLK는 모든 슬레이브에 공통이며 Arduino UNO의 경우 디지털 11, 12, 13번 핀으로 연결한다. Arduino UNO의 경우 디지털 10번 핀이 SS 핀으로 정의되어 있어 많이 사용하지만 마스터로 사용하는 경우 슬레이브 별로 하나의 SS 연결이 필요하므로 연결 가능한 디지털 핀은 모두 SS로 사용할 수 있다. 10번 핀은 아두이노가 SPI 통신에서 슬레이브로 선택될 수 있도록 하기 위해 사용하는 핀이기도 하다. 아두이노가 슬레이브로 동작하기 위해서는 10번 핀이 INPUT으로 설정된 상태에서 LOW 값이 인가되어야 한다. (SS가 LOW인 장치는 현재 마스터와 통신할 수 있는 장치를 나타낸다.) 아두이노 프로그램에서 제공되는 SPI 라이브러리는 아두이노의 마스터 모드만을 지원하며 초기화 과정에서 해당 핀의 입출력 상태를 자동으로 설정하므로 마스터 모드에서 사용할 때에는 초기화 이후 슬레이브에 연결된 SS 핀들을 HIGH 상태로 설정해주어야 한다는 점만 주의하면 된다.

SPI 신호	Arduino UNO	Arduino Mega
MOSI	11	51
MISO	12	50
SCLK	13	52
SS	10	53

표 1. SPI 연결 핀

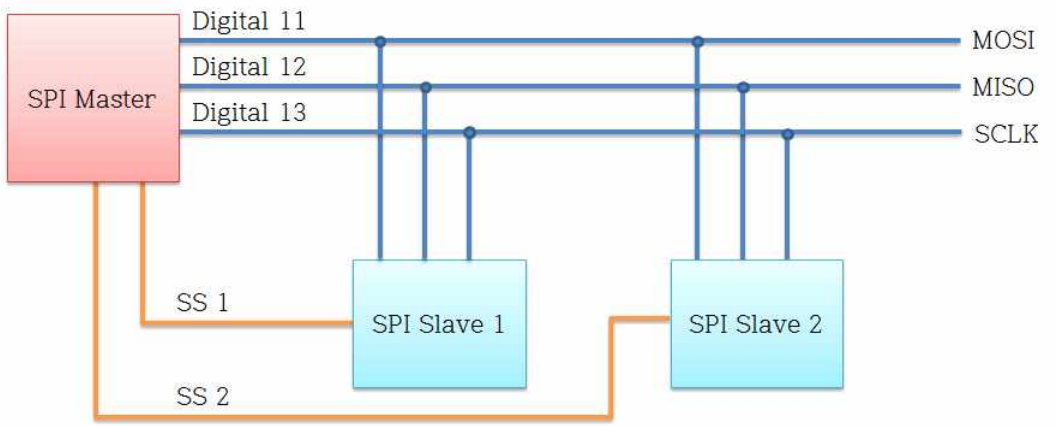

그림 4. SPI 연결

SPI는 데이터 전송을 위해 세부적인 내용까지 정의하고 있지는 않으며 장치에 따라 약간씩 다른 방법으로 구현하고 있으므로 데이터 직렬화 방법 (MSB 우선 또는 LSB 우선), 클록 동기화 방법, 전송 속도 등에 관하여 장치의 data sheet를 면밀히 살펴보아야 한다. SPI 라이브러리의 전송속도는 디폴트값으로 4MHz로 설정되어 있다.

4. SPI 라이브러리

SPI는 전이중 방식의 동기식 직렬 통신 프로토콜로 짧은 거리에 있는 주변 기기들과 통신하는 용도로 만들어졌으며 아두이노 프로그램에는 SPI 라이브러리가 포함되어 있다. SPI 통신을 통해 송수신되는 데이터의 의미는 주변 장치에 따라 달라지므로 SPI 라이브러리에는 기본적인 설정 및 데이터 전송 함수들만을 정의하고 있다. 실제 SPI 라이브러리에서 정의하고 있는 클래스는 SPIclass이며 이의 전역 객체인 SPI를 통해 SPI 통신이 이루어진다.

- begin

```
void begin(void)
  반환값 : 없음
```

SPI 통신을 초기화한다.

- end

```
void end(void)
  반환값 : 없음
```

SPI 통신을 종료한다.

- setBitOrder

```
void setBitOrder(uint8_t bitOrder)
    bitOrder : 데이터 송수신시 비트 순서
  반환값 : 없음
```

SPI 버스로 데이터가 직렬화 되어 전송될 때 전송 순서를 설정한다. LSBFIRST 또는 MSBFIRST 중 하나의 값을 가진다.

- setClockDivier

```
void setClockDivier(uint8_t rate)
    rate : 분주비율
  반환값 : 없음
```

시스템에서 사용하는 클록에 상대적인 분주비율을 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. Arduino UNO의 경우 16MHz 클록을 사용하므로 4MHz가 SPI의 기본 주파수에 해당한다.

- setDataMode

```
void setDataMode(uint8_t mode)
    mode : 전송 모드
  반환값 : 없음
```

데이터의 전송 모드를 설정한다. 전송 모드는 클록의 phase와 polarity 조합에 따라 4 가지로 나눌 수 있으며 각각 CPHA (Clock PHAse)와 CPOL (Clock POLarity) 또는 SPH (Signal PHase)와 SPO (Signal POlarity)라고 부른다. phase는 데이터가 샘플링 되는 시점을 결정한다. CPHA = 0이면 클록의 상승 에지에서 데이터가 샘플링 되고 CPHA = 1이면 하강 에지에서 데이터가 샘플링 된다. polarity는 클록이 비활성 상태일 때의 기본값을 결정한다. CPOL = 0이면 기본값은 0이며 CPOL = 1이면 기본값은 1이 된다. 이 두 가지 조합에 의해 모두 네 가지의 전송 모드가 존재한다.

Mode	Polarity (CPOL)	Phase (CPHA)
SPI_MODE0	0	0
SPI_MODE1	0	1
SPI_MODE2	1	0
SPI_MODE3	1	1

표 2. SPI 통신에서의 전송 모드

그림 5는 각 모드에 따라 샘플링이 이루어지는 시점과 클록의 관계를 나타낸 것이다.

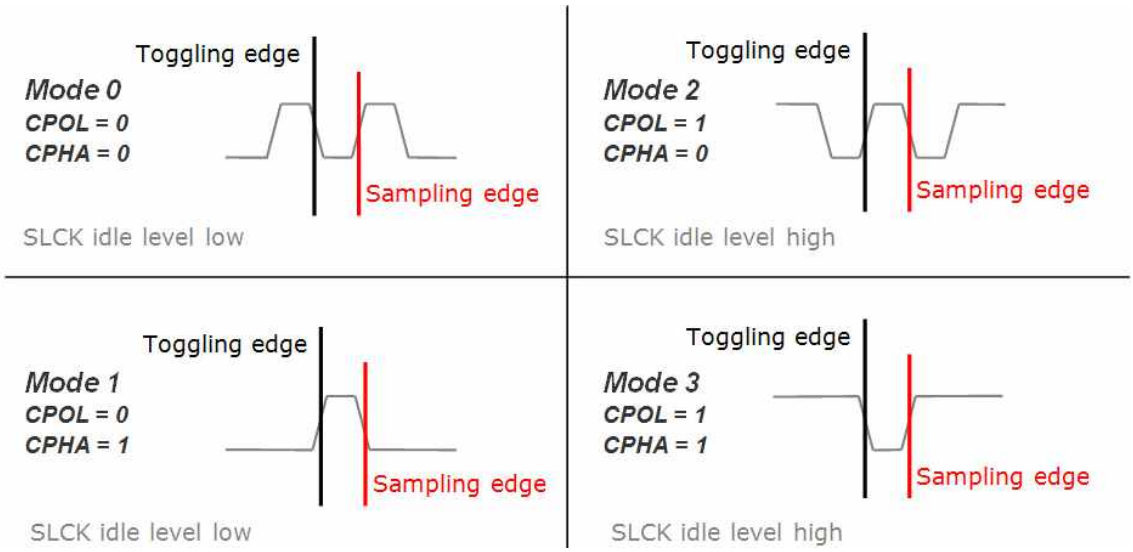

그림 5. 모드에 따른 클록 및 샘플링 시점

polarity와 phase의 의미를 완전히 이해할 필요는 없지만 마스터와 슬레이브 사이에 동일한 모드를 사용하여야만 정상적인 통신이 가능하므로 기기가 사용하는 모드를 data sheet에서 반드시 확인하여야 한다는 점은 기억하기 바란다.

- transfer

```
byte transfer(byte data)
    data : 송송할 데이터
  반환값 : 수신된 데이터
```

SPI 버스를 통해 한 바이트의 데이터를 보내고 받는다. 송신과 수신은 동시에 진행된다.

5. SPI를 이용한 아두이노간 통신

두 대의 아두이노를 SPI 통신을 이용하여 연결해 보자. 두 대의 아두이노를 먼저 그림 6에서와 같이 연결한다.

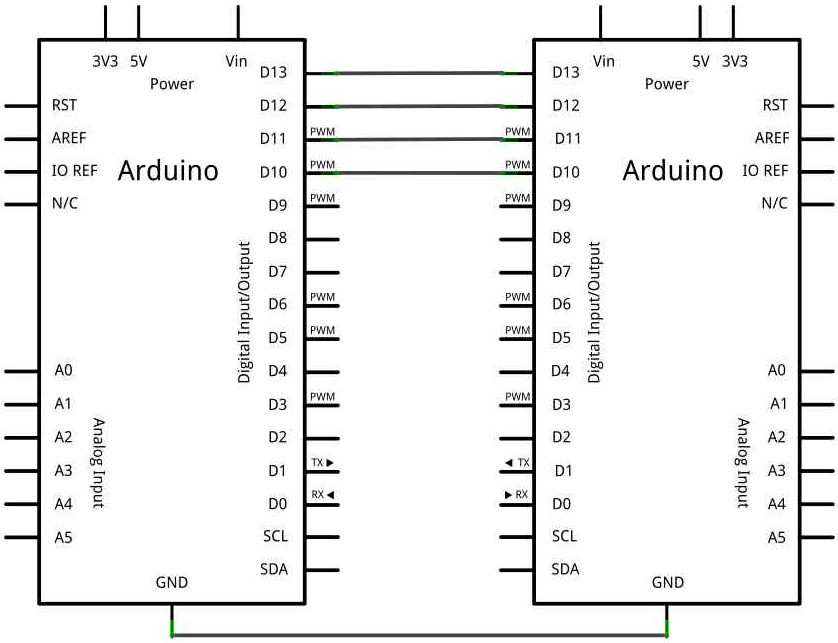

그림 6. 두 대의 아두이노를 SPI 통신으로 연결하는 회로도

두 대의 아두이노 중 하나는 마스터로 동작하며 1초 간격으로 'Hello World' 문자열을 전송한다. 수신측에서는 개행 문자인 '\n'을 문자열의 끝으로 인식하여 처리하므로 문자열 끝에 개행 문자가 추가되어 있다. 아두이노 사이의 SPI 통신에서 전송 속도가 높은 경우 데이터가 정상적으로 송수신되지 않으므로 분주비를 낮추어 전송 속도를 낮춰준다. 이 경우 슬레이브에서도 마스터에서와 동일한 속도로 설정해주어야 한다. 마스터에서 문자열의 전송하는 코드의 예는 코드 3과 같다.

코드 3. Master

```
#include <SPI.h>
```

```
void setup (void)
{
  SPI.begin ();                    // SPI 통신 초기화
  // pins_arduino.h에 SS는 기본적으로 10번 핀으로 정의되어 있다.
  digitalWrite(SS, HIGH);          // 슬레이브가 선택되지 않은 상태로 유지

  // 속도가 빠른 경우 데이터가 정확하게 전달되지 않으므로
  // 분주비를 높여 전송 속도를 낮춘다.
  SPI.setClockDivider(SPI_CLOCK_DIV16);
}

void loop (void)
{
  const char *p = "Hello, World\n";

  digitalWrite(SS, LOW);  // 슬레이브를 선택한다.
  for (int i = 0; i < strlen(p); i++){          // 문자열을 전송한다.
    SPI.transfer(p[i]);
  }

  digitalWrite(SS, HIGH); // 슬레이브 선택을 해제한다.

  delay(1000);
}
```

슬레이브 측은 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 번호로 SPI_STC_vect로 정의되어 있다. SPI 라이브러리의 경우 마스터 모드만을 지원하므로 슬레이브 모드로 동작하기 위해서는 해당 레지스터를 직접 설정하여야 하며 세 가지 설정이 필요하다.

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

이 세 가지는 모두 SPI Control Register인 SPCR의 해당 비트를 설정해 줌으로써 가능하다. SPCR 레지스터는 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 7. SPCR 레지스터 구조

SPCR 레지스터 중 먼저 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. 디폴트값이 0이므로 설정하지 않아도 된다. 마지막으로 7번 비트인 SPIE(SPI Interrupt Enable) 비트를 1로 설정하면 인터럽트 발생이 허용된다. MSTR, SPE, SPIE는 각각 4, 6, 7로 정의되어 있다.

```
#define MSTR 4
#define SPE 6
#define SPIE 7
```

각 비트는 _BV 매크로를 이용한 비트 연산을 통해 설정할 수 있다. _BV 매크로는 해당 위치의 비트만을 1로 하는 마스크를 생성하는 매크로로 다음과 같이 정의되어 있다.

```
#define _BV(bit) (1 << (bit))
```

따라서 _BV 매크로를 통해 생성된 마스크와 비트 OR 연산을 수행함으로써 해당 비트만을 1로 설정할 수 있으며 마스크를 반전시켜 비트 AND 연산을 수행함으로써 해당 비트만을 0으로 설정할 수 있다.

코드 4. Slave

```
#include <SPI.h>

char buf[100];                    // 수신된 문자 저장을 위한 버퍼
// pos와 process_it은 인터럽트 처리 루틴에서 값을 바꾸는 변수이므로
// volatile 선언을 통해 업데이트된 값이 정확하게 반영되도록 한다.
volatile byte pos;                // 수신 버퍼에 문자를 기록할 위치
volatile boolean process_it;      // 개행 문자를 만난 경우 출력하기 위한 플래그

void setup (void)
{
  Serial.begin (9600);            // 수신 문자열 출력을 위한 직렬 통신 초기화

  // 디지털 핀은 디폴트값으로 입력으로 설정되어 있으므로
  // MOSI, SCLK, SS는 입력으로 설정하지 않아도 된다.
  // MISO 역시 이 예에서는 사용하지 않으므로 생략할 수 있다.
  pinMode(MISO, OUTPUT);

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

  // SPI 통신을 사용할 수 있도록 레지스터를 설정한다.
  // SPCR : SPI Control Register
  SPCR |= _BV(SPE);               // SPE : SPI Enable

  // SPI 통신에서 슬레이브로 동작하도록 설정한다.
  // 디폴트값이 0이므로 생략할 수 있다.
  SPCR &= ~_BV(MSTR);             // MSTR : Master Slave Select

  pos = 0;                // 버퍼가 비어 있으므로 0번부터 수신 문자 기록
  process_it = false;     // Serial로 출력할 문자열 없음

  // SPI 통신으로 문자가 수신될 경우 인터럽트 발생을 허용한다.
  SPCR |= _BV(SPIE);              // SPIE : SPI Interrupt Enable
}
```

```
// SPI 통신으로 문자가 수신될 때 발생하는 인터럽트 처리 루틴
ISR (SPI_STC_vect)
{
  byte c = SPDR;          // 수신된 문자를 얻어온다.

  if (pos < sizeof(buf)){   // 현재 버퍼에 저장할 공간이 있는 경우
    buf[pos++] = c;         // 버퍼에 수신된 문자 기록

    if (c == '\n'){         // 개행 문자를 만나면 수신된 문자열을 Serial로 출력
      process_it = true;
    }
  }
}

void loop (void)
{
  if (process_it){          // Serial로 출력할 문자열이 있는 경우
    buf[pos] = 0;           // 문자열의 끝 표시
    Serial.print(buf);      // 문자열을 Serial로 출력
    pos = 0;                // 버퍼가 비었음을 표시
    process_it = false;     // Serial로 출력할 문자가 없음을 표시
  }
}
```

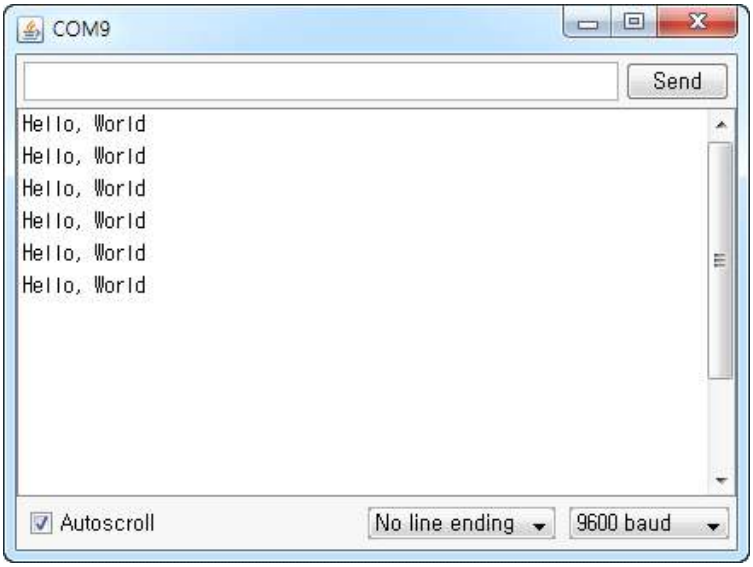
COM9
Send
Hello, World
Hello, World
Hello, World
Hello, World
Hello, World
Hello, World
Autoscroll
No line ending
9600 baud

그림 8. 코드 4 실행 결과