

Chapter 212. I2C - 가속도 자이로 센서

1. I2C (Inter-Integrated Circuit)

I2C 통신은 UART, SPI와 더불어 마이크로컨트롤러에서 흔히 사용되는 직렬 통신 방법 중 하나이다. UART는 1960년대에 구현된 오래된 직렬 통신 방식으로 다양한 장치들이 지원하는 장점이 있다. 하지만 UART는 1:1 통신만이 가능한 단점이 있어 여러 개의 주변장치를 연결하기가 불편하다. 이러한 단점을 보완할 수 있는, 1:N 통신이 가능한 통신 방법 중 하나가 SPI 통신이다. SPI에서는 3개의 데이터 선을 모든 장치가 공유하고 N개의 장치 중 하나의 장치를 선택하기 위한 SS (Slave Select) 연결을 장치별로 추가하여 사용하는 방식이다. SPI는 고속의 데이터 전송이 가능하지만 여러 개의 장치를 연결하는 경우 SS 연결이 증가하는 단점이 있다. I2C 역시 1:N 통신이 가능한 직렬 통신 방법이지만 SPI와는 다른 목적에서 만들어졌으므로 그 특성이 다르다. I2C가 SPI와 다른 점 중 하나는 SS 연결이 필요하지 않는 점이다. 그렇다면 어떻게 N개의 장치 중 하나를 선택할 수 있을까? I2C에서는 소프트웨어적인 주소로 특정 장치를 선택한다. 이는 인터넷의 경우와 비슷하다. 서로 연결되어 있는 수많은 컴퓨터로 구성되는 인터넷에서 나의 컴퓨터를 구별하기 위해 IP 주소를 사용하고 있다. SPI에서와 같이 전용 SS 연결에 의해 연결된 장치 중 하나를 선택하는 방식을 하드웨어 주소라고 한다면, I2C에서와 같이 각 장치에 고유의 주소를 지정하는 방식은 소프트웨어 주소라고 할 수 있다. I2C에서는 SS 연결이 필요하지 않으므로 연결된 장치의 수가 늘어나더라도 필요로 하는 연결의 수는 증가하지 않는 장점이 있다. SPI 통신에서 SS 연결을 제외하면 모든 장치들이 공유하고 있는 3개의 연결선이 남는다. 3개의 연결선 중 MOSI와 MISO 2개는 데이터 송수신을 위해 사용되고 SCK는 동기화 클럭 전송을 위해 사용된다. 하지만 I2C에서는 여기서 하나의 연결선을 더 줄일 수 있다. 연결선을 줄일 수 있는 비밀은 I2C가 다른 직렬 통신 방법과 다른 또 한 가지는 반이중(half-duplex) 방식의 통신이라는데 있다. UART와 SPI는 전이중(full-duplex) 방식의 통신이다. 전이중 방식이란 송신과 수신에 동시에 진행될 수 있는 방식으로 이를 위해 2개의 연결선을 필요로 한다. 이에 비해 반이중 방식에서는 1개의 연결선만을 사용하며 송신과 수신은 동시에 진행될 수 없다. 따라서 I2C에서는 데이터 송수신을 위해 1개, 동기화 클럭을 위해 1개 총 2개의 연결선만을 필요로 하며 이는 연결된 장치의 개수가 늘어나도 증가하지 않는다.

			UART	SPI	I2C
동기/비동기			비동기	동기	동기
전이중/반이중			전이중	전이중	반이중
연결선 개수	1개 슬레이브 연결	데이터	2	2	1 (반이중)
		클록	0	1	1
		제어	0	1	0
		합계	2	4	2
	N개 슬레이브 연결		2N	3+N	2
연결 방식			1:1	1:N (마스터-슬레이브)	1:N (마스터-슬레이브)
슬레이브 선택			-	하드웨어 (SS 라인)	소프트웨어 (주소 지정)

표 1. 시리얼 통신 방식 비교

이처럼 I2C는 SPI와는 달리 최소한의 연결선만을 사용하여 여러 개의 장치를 연결하면서 저속으로 데이터를 전송하기 위해 필립스에서 만든 규격으로 IIC, I²C 등으로도 불리고 있다. I2C에서 사용하는 2개의 연결선은 데이터 전송을 위해 사용되는 연결이 SDA (Serial Data), 클록 전송을 위해 사용되는 연결이 SCL(Serial Clock)이라고 불린다.

I2C 역시 SPI와 마찬가지로 동기식 통신으로 전용의 클록을 사용하지만 SPI에서와 같이 위상이나 극성에 따른 전송 모드가 존재하지는 않는다. I2C에서 데이터를 확인하는 방법은 간단하다. 수신된 데이터는 SCL이 HIGH인 경우에만 샘플링 할 수 있다는 규칙을 따른다. 그림 1은 클록에 따른 데이터의 샘플링 시점을 나타낸 것으로 SCL이 LOW가 아닌 구간에서 SDA는 안정된 상태에 있으므로 SCL이 HIGH인 구간에서 한 번 샘플링 하는 것만으로 데이터를 확인할 수 있다. 또한 SDA는 SCL이 LOW인 구간에서는 변화가 가능하다는 점도 눈여겨 볼 필요가 있다.

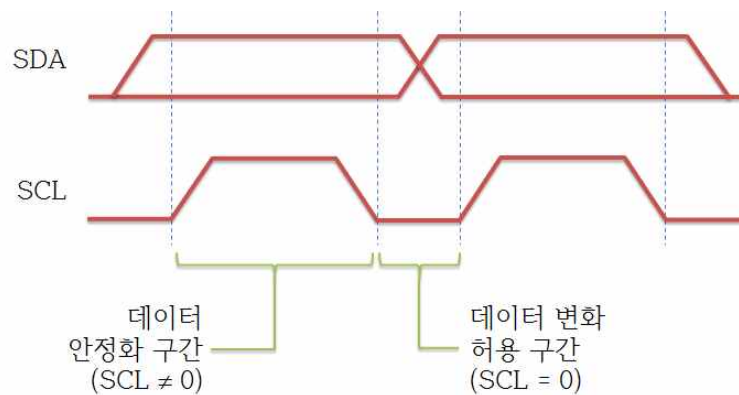


그림 1. 데이터 샘플링 및 전이

하지만 SCL이 HIGH인 구간에서도 SDA의 전이가 발생하는 두 가지 예외 상황이 존재하며 바로 데이터 전송의 시작과 종료를 나타내는 경우이다. SCL이 HIGH인 경우 SDA가 HIGH에서 LOW로 바뀌는 경우는 데이터 전송의 시작을, SDA가 LOW에서 HIGH로 바뀌는 경우는 데이터 전송이 끝을 나타낸다.

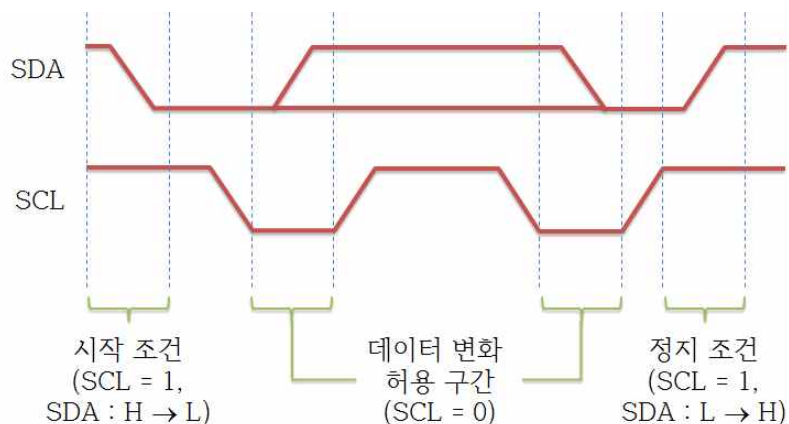


그림 2. 데이터 전송 시작 및 종료

I2C는 소프트웨어적인 주소를 사용하여 연결된 장치를 선택한다고 설명하였다. I2C에서의 주소는 7비트를 사용한다. 7비트의 주소가 익숙하지 않을 수 있지만 여기에 읽기 또는 쓰기를 나타내는 1비트를 추가하여 실제로는 8비트의 주소가 전달된다. 즉, 특정 주소와 읽기 비트를 전송하였다면 슬레이브에서 마스터로의 데이터 전송을, 특정 주소와 쓰기 비트를 전송하였다면 마스터에서 슬레이브로의 데이터 전송을 의미하게 된다.



그림 3. I2C 주소 지정

마스터가 시작 신호(S)와 7비트의 주소를 보내고 LOW 값($\overline{W}$)을 보냈다면 지정한 주소의 슬레이브는 마스터가 1바이트의 데이터를 전송할 것임을 인식하고 수신을 대기하게 된다. 시작 신호(S)와 7비트의 주소를 보내고 마지막에 HIGH 값(R)을 보냈다면 지정한 주소를 가지는 슬레이브는 마스터로 1바이트의 데이터를 전송할 것이다.

마지막으로 한 가지 더 기억해야 할 점은 데이터를 수신한 장치는 데이터를 수신했음을 알려주어야 한다는 점이다. I2C는 바이트 단위로 데이터를 전송하며 8비트의 데이터가 전송된 이후 SDA는 HIGH 상태에 있다. 바이트 단위 데이터가 전송된 이후 수신 장치는 정상적인 수신을 알리기 위해 9번째 비트를 송신 장치로 전송한다. LOW 값은 'ACK (acknowledgement) 비트'로 수신 장치가 정상적으로 데이터를 수신했음을 송신 장치로 알려주기 위해 사용되며, HIGH 값은 'NACK 비트'로 수신 과정에서 오류가 발생하였음을 나타낸다. 수신 장치가 정상적으로 데이터를 수신하지 못하면 수신 확인 ACK 비트를 전송하지 않을 것이고 SDA는 데이터 전송 후 HIGH 상태에 있으므로 별도로 NACK 비트를 전송하지 않아도 효과는 동일하다.

마스터에서 슬레이브로 n 바이트의 데이터를 송신하는 경우를 살펴보자. 먼저 데이터 전송 시작 비트(Start)와 7비트 주소, 그리고 데이터 송신 신호($\overline{W}$)를 보낸다. 지정된 주소의 슬레이브는 데이터를 수신할 준비를 시작하면서 9번째 비트인 수신 확인 신호를 보낸다. 이후 마스터는 n 바이트의 데이터를 송신하며 매 바이트가 전송된 이후 슬레이브는 수신 확인 비트를 마스터로 전송한다. 데이터 전송이 끝나면 데이터 송신 종료 비트(stoP)를 전송함으로써 통신을 끝낸다.

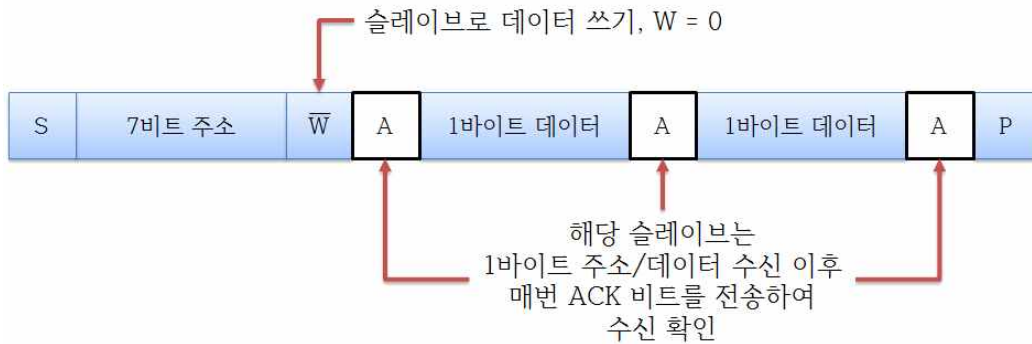


그림 4. 마스터의 n 바이트 데이터 쓰기 (■ : 마스터 전송, □ : 슬레이브 전송)

마스터가 슬레이브로부터 n 바이트의 데이터를 수신하는 경우도 이와 유사하다. 먼저 데이터 전송 시작 비트와 7비트 주소, 그리고 데이터 수신 신호(R)를 보낸다. 지정된 주소의 슬레이브는 데이터를 송신할 준비를 시작하면서 9번째 비트인 수신 확인 신호를 보낸다. 이후 슬레이브는 n 바이트의 데이터를 송신하며 매 바이트가 수신된 이후 마스터는 수신 확인 비트를 슬레이브로 전송한다. 다만 마지막 n 번째 바이트가 수신된 이후 마스터는 NACK를 슬레이브로 전송하여 수신이 완료되었음을 알린다. 데이터 전송이 끝나면 데이터 마스터가 송신 종료 비트를 전송함으로써 통신을 끝낸다. 데이터 송신의 경우와 마찬가지로 데이터 전송이 끝났음을 나타내는 종료 비트(stop)는 마스터에 의해 보내는 점은 SPI에서와 마찬가지로 모든 통신 과정이 마스터의 주도로 이루어짐을 나타낸다.

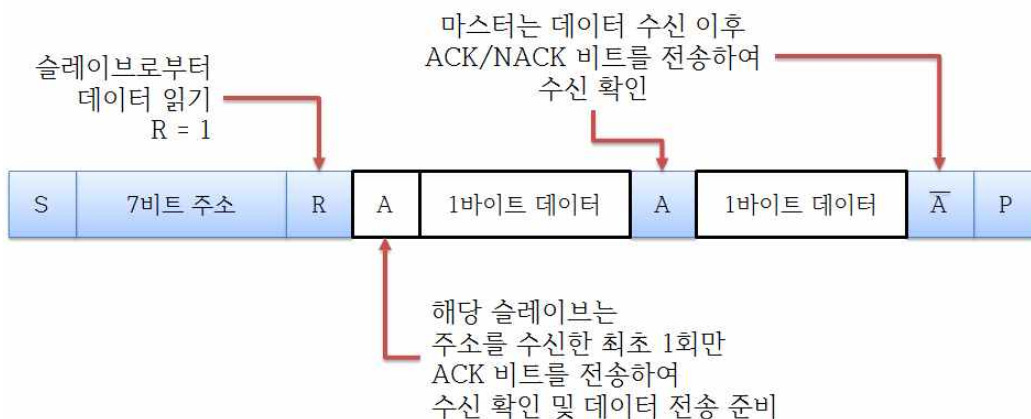


그림 5. 마스터로의 n 바이트 데이터 읽기 (■ : 마스터 전송, □ : 슬레이브 전송)

2. 아두이노 우노에서의 I2C

데이터 전송이 이루어지지 않는 경우 I2C 연결선은 HIGH 상태를 유지하고 있으며 이를 위해 I2C에서는 풀업저항을 사용한다. 두 개의 슬레이브를 아두이노 우노를 마스터로 연결한 예는 그림 6과 같다.

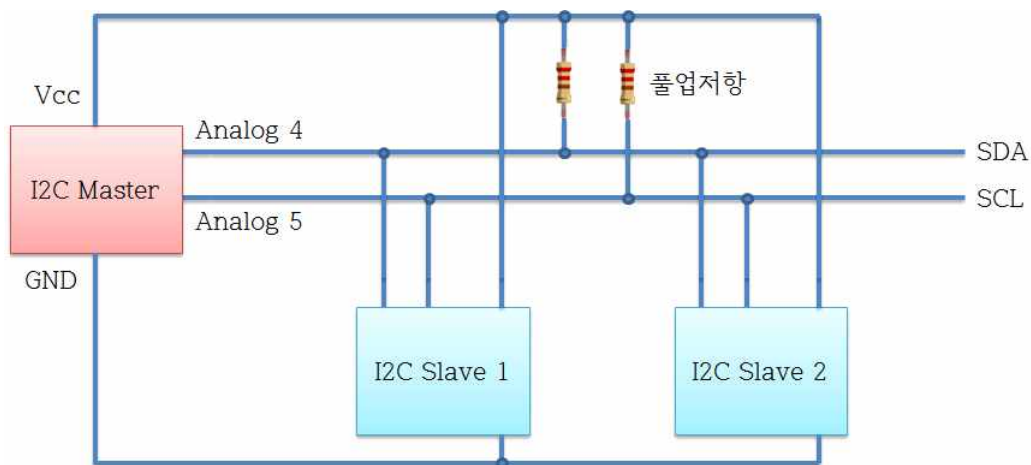


그림 6. I2C 연결 예

그림 6에서 아두이노 우노는 SDA를 위해 아날로그 핀 A4를, SCL을 위해 아날로그 핀 A5를 사용되고 있다. 아두이노 우노는 하드웨어적으로 I2C 통신을 지원하고 있으므로 반드시 전용 핀을 통해 연결하여야 한다. 또한 아두이노에서는 I2C 통신을 지원하기 위해 기본 라이브러리 중 하나로 Wire 라이브러리를 제공하고 있다. I2C와 혼용되어 사용되고 있는 TWI(Two Wire Interface)는 I2C를 지원하는 하드웨어 제작 회사에서 사용하는 I2C의 다른 이름으로 통신을 위해 2개의 연결선을 사용하기 때문에 붙여진 이름이다. Wire 라이브러리에는 I2C 통신을 통해 송신 및 수신을 위해 다음과 같은 멤버 함수들이 정의되어 있다.

- begin

```
void begin(void)
void begin(uint8_t address)
    - 매개변수
        address : I2C 주소
    - 반환값 : 없음
```

Wire 라이브러리를 초기화하고 I2C 버스에 마스터나 슬레이브로 참여한다. 주소가 주어지는 경우에는 슬레이브로 I2C 버스에 참여하는 것을 나타내고, 주소가 주어지지 않은 경우에는 마스터로 참여하는 것을 나타낸다.

- requestFrom

```
uint8_t requestFrom(uint8_t address, uint8_t quantity)
uint8_t requestFrom(uint8_t address, uint8_t quantity, uint8_t sendStop)
    - 매개변수
        address : 슬레이브 주소
        quantity : 요청하는 바이트 수
        sendStop : 요청 완료 후 정지 메시지 전송 여부 (true 또는 false)
    - 반환값 : 슬레이브 장치로부터 전송된 바이트 수
```

마스터 장치가 특정 주소(address)의 슬레이브 장치에 지정한 양(quantity)의 데이터를 요청한다. 매개변수 sendStop에는 true나 false의 값을 지정하며 요청이 완료된 후 정지 메시지를 보낼지의 여부를 나타낸다. false를 지정하면 I2C 마스터는 요청이 완료된 후에도 연결을 유지하여 다른 마스터 장치가 데이터를 요구할 수 없도록 한다. 디폴트는 true이다.

- beginTransmission

```
void beginTransmission(uint8_t address)
    - 매개변수
        address : 슬레이브 주소
    - 반환값 : 없음
```

지정한 주소의 슬레이브 장치로 데이터 전송을 시작한다. 실제 전송은 write 함수에 의해 버퍼에 기록된 후 endTransmission 함수가 호출될 때 일어난다.

- endTransmission

```
uint8_t endTransmission(void)
uint8_t endTransmission(uint8_t sendStop)
    - 매개변수
        sendStop : 요청 완료 후 정지 메시지 전송 여부 (true 또는 false)
    - 반환값 : 전송 상태 메시지
```

write 함수에 의해 큐에 기록된 데이터를 전송함으로써 beginTransmission 함수에 의해 시작된 슬레이브 장치에 대한 전송을 종료한다. sendStop이 false로 설정되면 전송이 완료된 후에도 연결을 유지하여 다른 마스터 장치가 데이터를 전송할 수 없도록 한다. 디폴트는 true 이다. 반환값은 전송 결과를 나타내는 값으로 0이 아닌 값은 전송 과정에서 오류가 발생하였음을 나타낸다. 반환값에 따른 오류 내용은 표 1과 같다.

반환값	의미
0	전송 성공
1	전송 데이터가 버퍼 용량을 초과
2	주소 전송 후 NACK(Negative Acknowledgement) 수신
3	데이터 전송 후 NACK(Negative Acknowledgement) 수신
4	기타 TWI 오류

표 1. I2C 데이터 전송에서의 오류

- write

```
size_t write(uint8_t data)
size_t write(const uint8_t *data, size_t quantity)
    - 매개변수
        data : 전송할 단일 바이트 또는 바이트 배열에 대한 포인터
        quantity : 전송할 바이트 수
    - 반환값 : 전송된 바이트 수
```

마스터 장치의 요청에 따라 슬레이브 장치가 데이터를 전송하거나 마스터 장치에서 슬레이브 장치로의 데이터 전송을 위해 큐에 데이터를 기록하기 위해 사용한다. 마스터 장치에서 슬레이브 장치로의 데이터 전송은 beginTransmission 함수로 시작되고 write 함수에서 큐에 데이터를 기록한 후 endTransmission 함수 호출로 실제 전송이 일어난다.

- available

```
int available(void)
- 매개변수 : 없음
- 반환값 : 유효 바이트 수
```

read 함수로 읽어 들일 수 있는 유효한 바이트 수를 반환한다. available 함수는 마스터 장치에서 requestFrom 함수를 통해 슬레이브 장치로부터 데이터를 요청한 이후 실제 도착한 데이터를 검사하기 위해 사용하거나 슬레이브 장치의 onReceive 함수 내에서 사용된다.

- read

```
int read(void)
- 매개변수 : 없음
- 반환값 : 수신 버퍼의 한 바이트를 읽어서 반환
```

마스터 장치의 requestFrom 함수 호출에 의해 슬레이브 장치가 전송한 데이터 한 바이트를 읽어서 반환한다. 마스터 장치가 전송한 데이터를 슬레이브 장치에서 읽어 들이기 위해서도 사용할 수 있다.

- onReceive

```
void onReceive( void (*)(int) )
- 매개변수 : 없음
- 반환값 : 없음
```

슬레이브 장치가 마스터 장치로부터 데이터를 수신했을 때 호출되는 핸들러(handler) 함수를 등록한다. 핸들러 함수는 반환값이 없으며 수신한 데이터의 바이트 수를 나타내는 int 형식의 매개변수를 가진다.

- onRequest

```
void onRequest( void (*)(void) )
- 매개변수 : 없음
- 반환값 : 없음
```

슬레이브 장치가 마스터 장치로부터 데이터 요청을 받았을 때 호출되는 핸들러(handler) 함수

를 등록한다. 핸들러 함수는 반환값이 없으며 매개변수도 갖지 않는다.

3. 가속도 자이로 센서

가속도 센서와 자이로 센서는 스마트폰에 포함된 이후로 실생활에서 많이 쓰이는, 흔히 접할 수 있는 센서 중 하나가 되었다. 가속도 센서는 직선 방향으로 단위 시간의 속도 증감비, 즉, 가속도를 측정하는 센서를 말하며, 자이로 센서는 축을 기준으로 단위 시간에 물체가 회전한 각도, 즉, 각속도를 측정하는 센서를 말한다.

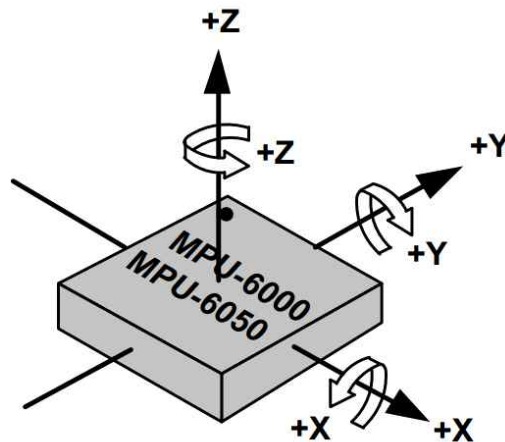


그림 7. 가속도와 각속도

가속도와 각속도를 측정하기 위해 흔히 사용되는 칩으로 MPU6050 칩이 있다. MPU6050은 24개의 핀을 가지는 칩으로 가속도 센서와 자이로 센서를 소형 칩 내에 집적시켜 놓고 있다. 측정된 값은 16비트의 아날로그-디지털 변환기를 통해 변환되며 X, Y, Z 방향의 가속도와 각속도를 동시에 측정할 수 있으며 아두이노와 I2C 통신을 통해 연결될 수 있다. 또한 온도 센서가 포함되어 있어 온도 측정 역시 가능하다. 이 장에서는 MPU6050을 사용하여 1.5×2 cm 크기로 제작된 모듈을 사용한다.



그림 8. 가속도 자이로 센서 모듈

센서 모듈을 그림 9와 같이 아두이노 우노에 연결해 보자. 연결에서 주의할 점은 센서 모듈은 3.3V를 사용하므로 아두이노 우노의 3.3V와 센서 모듈의 VCC를 연결하여야 한다는 점이다. 8개의 핀 중 나머지 4개는 사용하지 않는다.

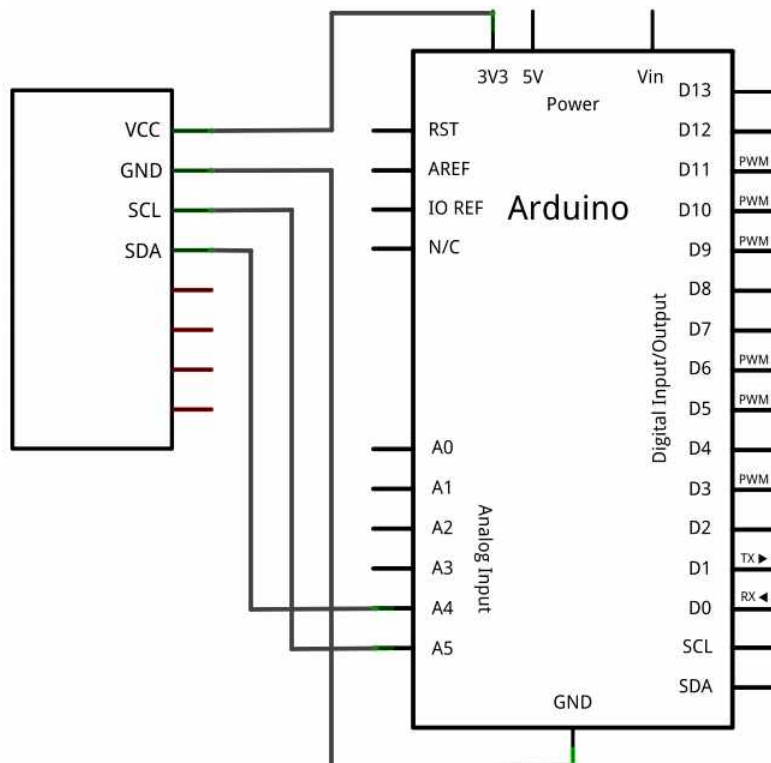


그림 9. 센서 모듈과 아두이노 연결 회로도

MPU6050 칩을 사용하기 위해서는 먼저 MPU6050 칩의 I2C 주소를 지정하여 슬레이브로 연결하여야 한다. 그림 9와 같이 연결한 경우 MPU6050 칩의 I2C 주소는 0x68로 설정되며 변경할 수 없다.

MPU6050 칩에 전원을 인가한 경우 MPU6050 칩은 슬립 모드(sleep mode)로 시작된다. 따라서 센서 데이터를 얻어오기 위해서는 먼저 슬립 모드를 해제하여야 한다. MPU6050 칩에는 13번부터 117번까지의 레지스터가 존재하며 이 중 전원 관리 및 클록 소스 설정을 위해 사용되는 레지스터는 107번(0x6B) 레지스터 PWR_MGMT_1(Power Management 1)이다. 표 1은 PWR_MGMT_1 레지스터의 구조를 나타낸다.

레지스터		Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
HEX	DEC								
6B	107	DEVICE_RESET	SLEEP	CYCLE	-	TEMP_DIS	CLKSEL[2..0]		

표 1. PWR_MGMT_1 레지스터 구조

7번 비트 DEVICE_RESET은 MPU6050 칩을 리셋하기 위해 사용되며, 3번 비트 TEMP_DIS는 온도 센서를 중지시키기 위해 사용된다. 6번 비트 SLEEP이 설정되면 MPU6050 칩은 저전력의 슬립 모드로 설정된다. 슬립 모드가 아닌 경우 5번 비트 CYCLE이 설정되면 슬립 모드와 웨이크업 모드를 전환하면서 일정 시간 간격으로만 센서가 동작한다. 모드 전환 간격은 108번 레지스터 PWR_MGMT_2(Power Management 2)의 LP_WAKE_CTRL 비트(6번 및 7번 비트)에 의해 결정된다. CLKSEL 비트는 MPU6050 칩의 동작 클록을 설정하기 위해 사용되며 디폴트값은 0으로 내부 오실레이터에 의해 동작하도록 설정되어 있다. PWR_MGMT_1 레지스터에 0을 기록하면 디폴트값에서 슬립모드만이 해제된다. PWR_MGMT_1 레지스터에 0을 기록하기 위해서는 먼저 레지스터 번호를 전송한 후 레지스터의 값을 전송하면 된다.

센서 데이터는 3개의 가속도, 3개의 각속도, 1개의 온도가 16비트 값으로 14개의 레지스터에 저장된다. 저장되는 위치는 59번(0x3B) 레지스터부터 72번(0x48) 레지스터까지이다. 표 2는 센서값이 저장되는 레지스터의 구조를 나타낸다.

레지스터		레지스터명	설명	비고
HEX	DEC			
3B	59	ACCEL_XOUT_H	ACCEL_XOUT[15:8]	X축 가속도
3C	60	ACCEL_XOUT_L	ACCEL_XOUT[7:0]	
3D	61	ACCEL_YOUT_H	ACCEL_YOUT[15:8]	Y축 가속도
3E	62	ACCEL_YOUT_L	ACCEL_YOUT[7:0]	
3F	63	ACCEL_ZOUT_H	ACCEL_ZOUT[15:8]	Z축 가속도
40	64	ACCEL_ZOUT_L	ACCEL_ZOUT[7:0]	
41	65	TEMP_OUT_H	TEMP_OUT[15:8]	온도
42	66	TEMP_OUT_L	TEMP_OUT[7:0]	
43	67	GYRO_XOUT_H	GYRO_XOUT[15:8]	X축 각속도
44	68	GYRO_XOUT_L	GYRO_XOUT[7:0]	
45	69	GYRO_YOUT_H	GYRO_YOUT[15:8]	Y축 각속도
46	70	GYRO_YOUT_L	GYRO_YOUT[7:0]	
47	71	GYRO_ZOUT_H	GYRO_ZOUT[15:8]	Z축 각속도
48	72	GYRO_ZOUT_L	GYRO_ZOUT[7:0]	

표 2. 센서값 저장 레지스터

X축 방향의 가속도를 얻고 싶다면 레지스터 번호(59)를 전송한 후, 2바이트의 데이터를 읽어서 16비트 값으로 연결하면 된다. 레지스터 번호를 지정한 후 연속적으로 읽기를 수행하면 레지스터 번호는 자동으로 증가하므로 표 2의 모든 데이터를 읽기 위해서는 read 명령을 14번 실행하면 된다. 스케치 1은 센서 모듈에서 가속도, 각속도 및 온도를 읽어 출력하는 예이다. 그림 10은 스케치 1의 실행 결과로 센서 모듈을 좌우로 흔들면서 출력한 값이므로 AX 값이 양수와 음수로 변하고 있음을 눈여겨 볼 필요가 있다.

스케치 1. MPU6050 가속도 자이로 센서 모듈 테스트	
<pre>#include <Wire.h> int MPU_Address = 0x68; // MPU6050 칩의 I2C 주소 int AcX, AcY, AcZ, Tmp, GyX, GyY, GyZ;</pre>	

```
void setup(){
  Wire.begin();
  Wire.beginTransmission(MPU_Address); // MPU6050 선택
  Wire.write(0x6B);                    // PWR_MGMT_1 레지스터 선택
  Wire.write(0);                        // 슬립 모드 해제
  Wire.endTransmission();
  Serial.begin(115200);                 // UART 직렬 통신 초기화
}

void loop(){
  Wire.beginTransmission(MPU_Address); // MPU6050 선택
  Wire.write(0x3B);                    // X축 방향 가속도 레지스터 선택
  // 연결을 종료하지 않으므로 beginTransmission 없이 읽기 가능
  Wire.endTransmission(false);
  Wire.requestFrom(MPU_Address, 14);   // 14바이트 데이터 요청

  AcX = Wire.read() << 8 | Wire.read(); // ACCEL_XOUT
  AcY = Wire.read() << 8 | Wire.read(); // ACCEL_YOUT
  AcZ = Wire.read() << 8 | Wire.read(); // ACCEL_ZOUT
  Tmp = Wire.read() << 8 | Wire.read(); // TEMP_OUT
  GyX = Wire.read() << 8 | Wire.read(); // GYRO_XOUT
  GyY = Wire.read() << 8 | Wire.read(); // GYRO_YOUT
  GyZ = Wire.read() << 8 | Wire.read(); // GYRO_ZOUT

  Serial.print("AX = "); Serial.print(AcX);
  Serial.print(", AY = "); Serial.print(AcY);
  Serial.print(", AZ = "); Serial.print(AcZ);
  // 레지스터 값을 섭씨 온도로 변환
  Serial.print(", Tmp = "); Serial.print(Tmp / 340.00 + 36.53);
  Serial.print(", GX = "); Serial.print(GyX);
  Serial.print(", GY = "); Serial.print(GyY);
  Serial.print(", GZ = "); Serial.println(GyZ);

  Wire.endTransmission();              // 데이터 읽기 종료

  delay(1000);
```

```
}
```

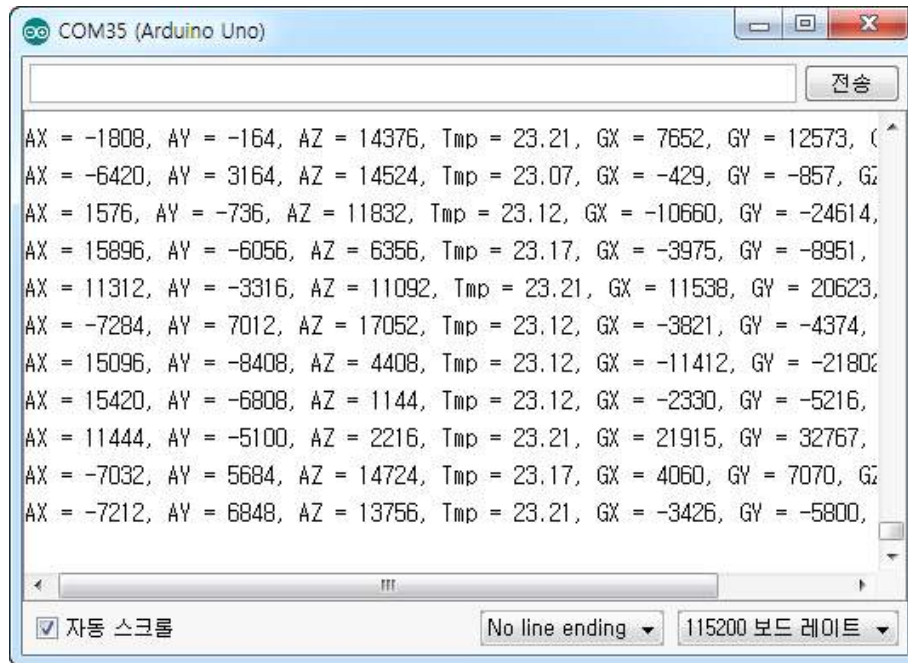


그림 10. 스케치 1 실행 결과