Chapter 20. I2C

1. I2C

지금까지 살펴본 시리얼 통신 방법으로는 UART(Universal Asynchronous Receiver/Transmitter)와 SPI(Serial Peripheral Interface)가 있었다. 이 장에서는 UART, SPI와 더불어 ATmega128에서 하드웨어로 지원하는 시리얼 통신 방식인 I2C(Inter-Integrated Circuit)를 살펴본다. I2C는 저속의 주변 장치 사이의 통신을 위해 필립스에서 만든 규격으로 IIC, I^2C 등으로도 표기된다. UART는 1960년대에 구현된 통신 표준으로 통신 방식이 간단할 뿐만이 아니라 역사가 길어 아직도 다양한 장치들이 지원하고 있다. 하지만 1:1 통신만 가능하여 여러 개의 주변장치를 연결하기가 불편하다. 이를 보완하기 위한 방법으로 여러 가지 시리얼 통신 방법이 제안되었고 이 중 흔히 볼 수 있는 방법이 SPI와 I2C이다. I2C는 저속의 여러 주변 장치들이 최소한의 연결선만을 사용하여 통신할 수 있도록 만들어졌다. 반면 SPI는 고속의 통신을 목표로 하는 점에서 차이가 있다. 표 1은 세 가지 시리얼 통신 방식을 간략하게 비교한 것이다.

		UART	ISP	I2C
동기/비동기		비동기	동기	동기
전이중/반이중		전이중	전이중	반이중
연결 선	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

I2C는 SPI와 마찬가지로 마스터-슬레이브 구조를 가지지만 연결된 슬레이브 장치의 개수와

무관하게 데이터 전송을 위한 SDA (Serial Data), 클록 전송을 위한 SCA (Serial Clock) 두 개의 선만을 필요로 한다. SDA는 데이터가 전송되는 통로로 하나만 존재한다. 송신과 수신은 SDA를 통해 이루어지므로 송신과 수신은 동시에 이루어질 수 없는 반이중(half-duplex) 방식을 사용한다. SCL은 클록 전송을 위한 통로로 SPI에서와 마찬가지로 마스터가 클록을 생성하고 데이터 전송의 책임을 진다.

I2C 역시 SPI와 마찬가지로 동기 방식의 프로토콜이지만 SPI에서와는 달리 위상(phase)과 극성(polarity)에 따른 여러 가지 전송 모드가 존재하지는 않는다. 수신된 데이터는 SCL이 HIGH인 경우에만 샘플링이 가능하다. 따라서 SCL이 HIGH인 경우 SDA의 데이터는 안정된 상태에 있어야만 한다. 데이터 전이는 SCL이 LOW인 상태에서만 가능하다.

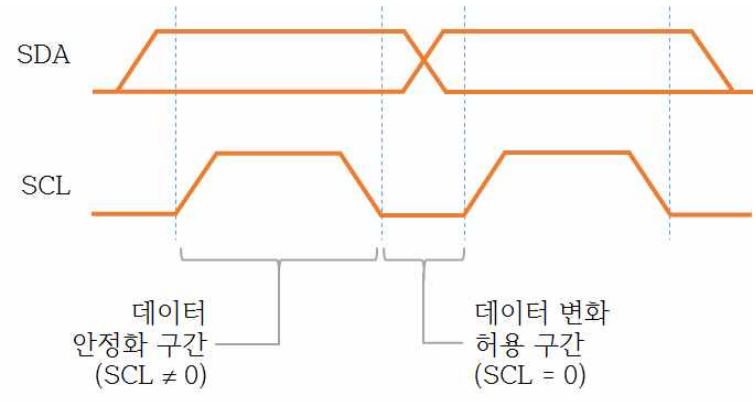

그림 1. 데이터 샘플링 및 전이

하지만 SCL이 HIGH인 경우에도 전이가 발생하는 두 가지 예외 상황이 있으며 바로 데이터 전송 시작과 종료를 나타내는 경우이다. SCL이 HIGH인 경우 SDA가 HIGH에서 LOW로 바뀌는 경우는 데이터가 전송되기 시작함을 나타내고, LOW에서 HIGH로 바뀌는 경우는 데이터 전송이 끝났음을 나타낸다.

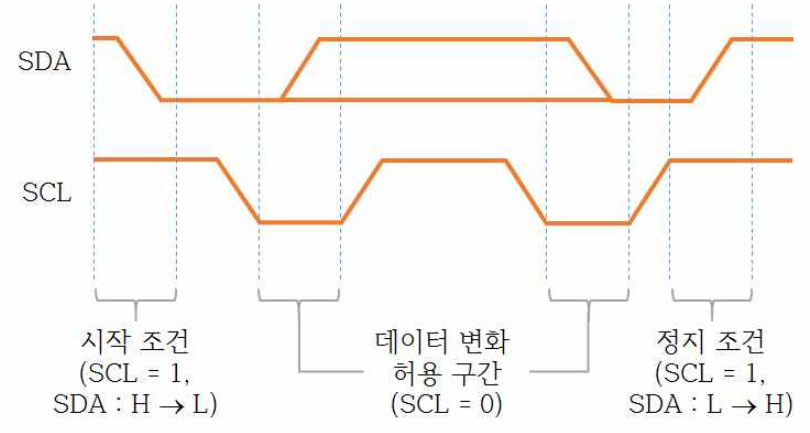

그림 2. 데이터 전송 시작 및 종료

I2C는 SPI와 마찬가지로 1:N 통신을 지원한다. SPI에서는 각각의 슬레이브가 전용의 SS(Slave Select) 또는 CS(Chip Select) 라인을 가지고 하드웨어적으로 데이터를 송수신할 슬레이브를 선택한다. 반면 I2C에서는 슬레이브가 고유의 주소를 가지고 소프트웨어적으로 데이터를 송수신할 슬레이브를 선택한다. 이는 인터넷에 연결된 컴퓨터를 구별하기 위해 IP(Internet Protocol) 주소를 사용하는 것과 비슷하다.

I2C는 7비트 주소를 사용한다. 7비트가 어색할지 모르지만 나머지 한 비트는 읽기/쓰기를 선택하기 위해 사용된다. 즉, HIGH 값이 주어진 경우 마스터는 지정한 슬레이브로부터 전송되는 데이터를 SDA 라인에서 읽어 들일 것임을 나타내고, LOW 값이 주어진 경우 마스터는 지정한 슬레이브로 SDA 라인을 통해 데이터를 전송할 것임을 나타낸다. 7비트의 주소 중 '0000 000'은 마스터가 여러 개의 슬레이브에게 동시에 메시지를 보내는 용도(general call)로 사용하기 위해 예약되어 있으므로 사용할 수 없다. '1111 xxx' 주소 역시 이후 사용을 위해 예약되어 있어 사용할 수 없다.

그림 3. 어드레스 지정

마스터가 시작 신호(S)와 7비트의 주소를 보내고 LOW 값($\overline{W}$)을 보냈다면 지정한 주소를 가

지는 슬레이브는 마스터가 1바이트의 데이터를 전송할 것임을 인식하고 수신을 대기하게 된다. 또한 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 SDA 라인은 HIGH 상태에 있다. 바이트 단위 데이터가 전송된 이후 수신 장치는 정상적인 수신을 알리기 위해 9번째 비트를 송신 장치로 전송한다. LOW 값은 'ACK (acknowledge) 비트'로 수신 장치가 송신 장치에게 정상적으로 데이터를 수신했음을 알려주기 위해 사용되며, HIGH 값은 'NACK (not acknowledge) 비트'로 정상적인 데이터 수신 이외의 상황이 발생했음을 알려주기 위해 사용된다. 데이터를 수신한 장치가 정상적으로 데이터를 수신하지 못하면 수신 확인인 ACK 비트를 전송하지 않는다. 데이터 전송이 완료된 이후 SDA는 HIGH 상태를 유지하고 있을 것이므로 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 신호를 보낸다. 지정된 주소의 슬레이브는 데이터를 수신할 준비를 시작하면서 9번째 비트인 수신 확인 신호를 보낸다. 이후 마스터는 n 바이트의 데이터를 송신하게 되며 매 바이트가 수신된 이후에 슬레이브는 수신 확인 비트를 마스터로 전송한다. 데이터 전송이 끝나면 데이터 송신 종료 비트(P)를 전송함으로써 통신을 끝낸다.

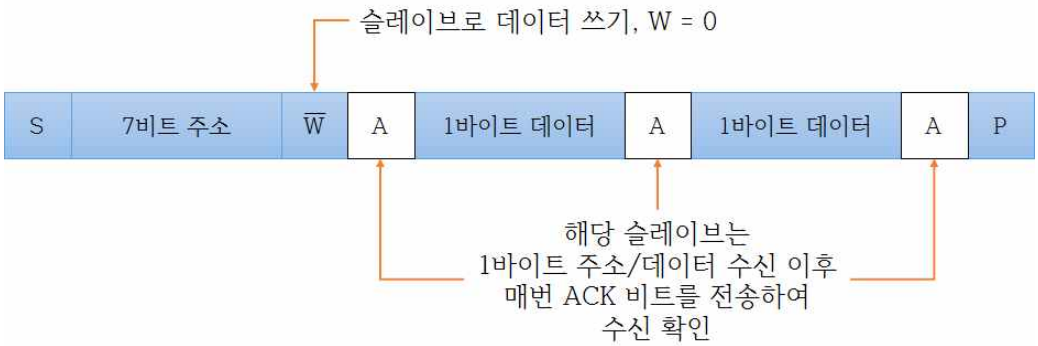

그림 4. 마스터의 n 바이트 데이터 쓰기 (: 마스터 전송, : 슬레이브 전송)

마스터가 슬레이브로부터 n 바이트의 데이터를 수신하는 경우도 이와 유사하다. 먼저 데이터 전송 시작 비트와 7비트 주소, 그리고 데이터 수신 신호를 보낸다. 지정된 주소의 슬레이브는 데이터를 송신할 준비를 시작하면서 9번째 비트인 수신 확인 신호를 보낸다. 이후 슬레이브는

n 바이트의 데이터를 송신하게 되며 매 바이트가 수신된 이후에 마스터는 수신 확인 비트를 슬레이브로 전송한다. 다만 마지막 n 번째 바이트가 수신된 이후 마스터는 NACK를 슬레이브로 전송하여 수신이 완료되었음을 알린다. 데이터 전송이 끝나면 데이터 송신 종료 비트를 전송함으로써 통신을 끝낸다. 데이터 송신의 경우와 마찬가지로 데이터 전송이 끝났음을 나타내는 종료 비트(P)는 마스터에 의해 보내진다는 점에 유의하여야 한다.

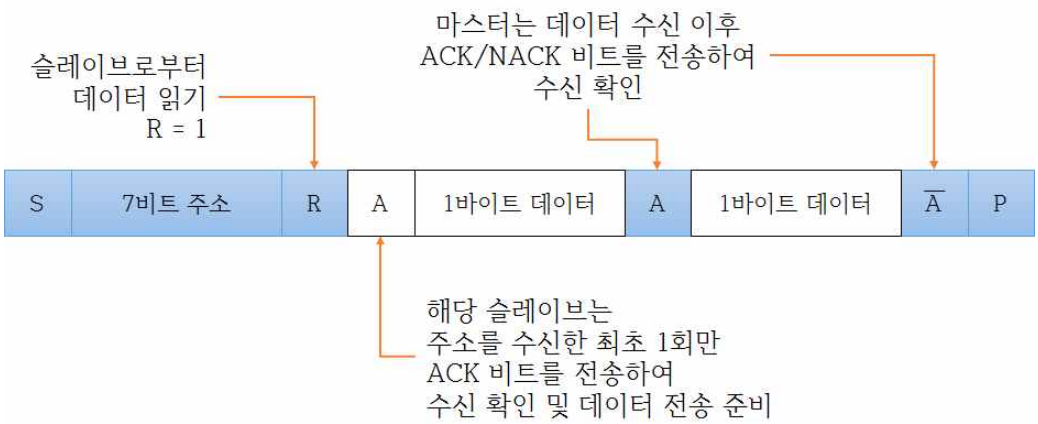

그림 5. 마스터로의 n 바이트 데이터 읽기 (: 마스터 전송, : 슬레이브 전송)

그림 6은 2개의 슬레이브 장치가 I2C를 통해 연결된 예를 나타내는 것으로 SDA와 SCL에 풀업 저항이 연결되어 있다는 점에 유의하여야 한다. 또한 ATmega128은 SDA를 위해 PD1 핀을, SCL을 위해 PD0 핀을 사용하도록 지정되어 있다는 점도 기억해야 한다.

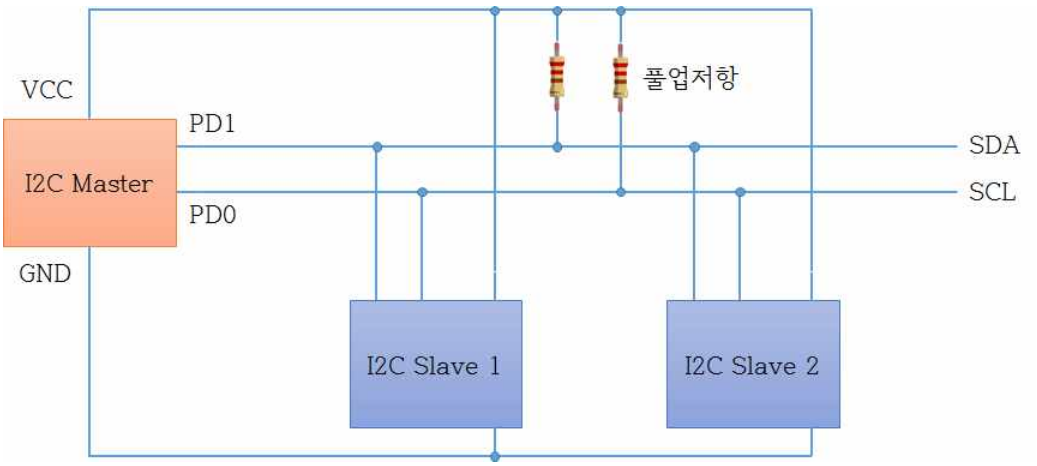

그림 6. I2C 연결 예

2. DS1307, RTC (Real Time Clock) 칩

RTC란 현재 시간을 유지하고 있는 시계를 말한다. 일반적으로 RTC는 컴퓨터나 임베디드 시스템에 존재하는 시계를 이야기 하지만 이외에도 현재 시간을 유지하기 위해 다양한 전자 장치에서 사용되고 있다. RTC는 일반적으로 별도의 전원과 클록을 가지고 있어 시스템의 전원이 공급되지 않는 경우에도 현재 시간을 유지할 수 있다. 마이크로컨트롤러에서 RTC를 사용하는 방법에는 소프트웨어적로 구현하는 방법과 별도의 하드웨어를 사용하는 방법이 있다. 소프트웨어로 시계를 구현한 경우에는 마이크로컨트롤러의 클록을 기준으로 시간을 계산하는 방법으로 간편하고 별도의 하드웨어를 필요로 하지 않는 장점은 있지만 마이크로컨트롤러에 전원이 공급되지 않으면 시간을 유지할 수 없는 단점이 있다. 이 장에서는 RTC를 위한 전용의 하드웨어를 사용하는 방법에 대해 알아본다. RTC를 위해 사용할 수 있는 칩은 여러 가지가 있으며 DS1307 칩이 흔히 사용된다. DS1307 칩은 I2C 프로토콜을 사용하며 외부에 수정 발진기와 전용 전원을 통해 현재 시간을 유지한다. DS1307 칩의 핀 배치는 그림 7과 같다.

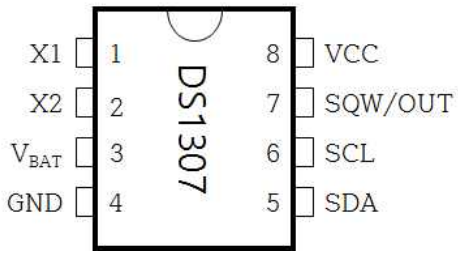

그림 7. DS1307 핀 배치

DS1307 칩의 각 핀 기능은 표 2와 같이 요약할 수 있다.

핀 번호	이름	설명	비고
1	X1	RTC용 발진기	32.768KHz
2	X2		
3	V_{BAT}	RTC용 전원	3.0V
4	GND		
5	SDA	I2C 데이터	
6	SCL	I2C 클록	
7	SQW/OUT	구형파 출력	1, 4, 8, 32KHz 출력
8	VCC		4.5 ~ 5.5V

표 2. DS1307 핀 설명

DS1307에는 두 개의 전원이 연결되며 VCC는 5V를, V_{BAT}는 3.0V를 사용한다. SQW/OUT(Square Wave Output)은 1, 4, 8, 32KHz 중 하나의 주파수를 갖는 구형파를 출력하도록 설정될 수 있다. DS1307을 마이크로컨트롤러와 연결한 전형적인 예는 그림 8과 같다.

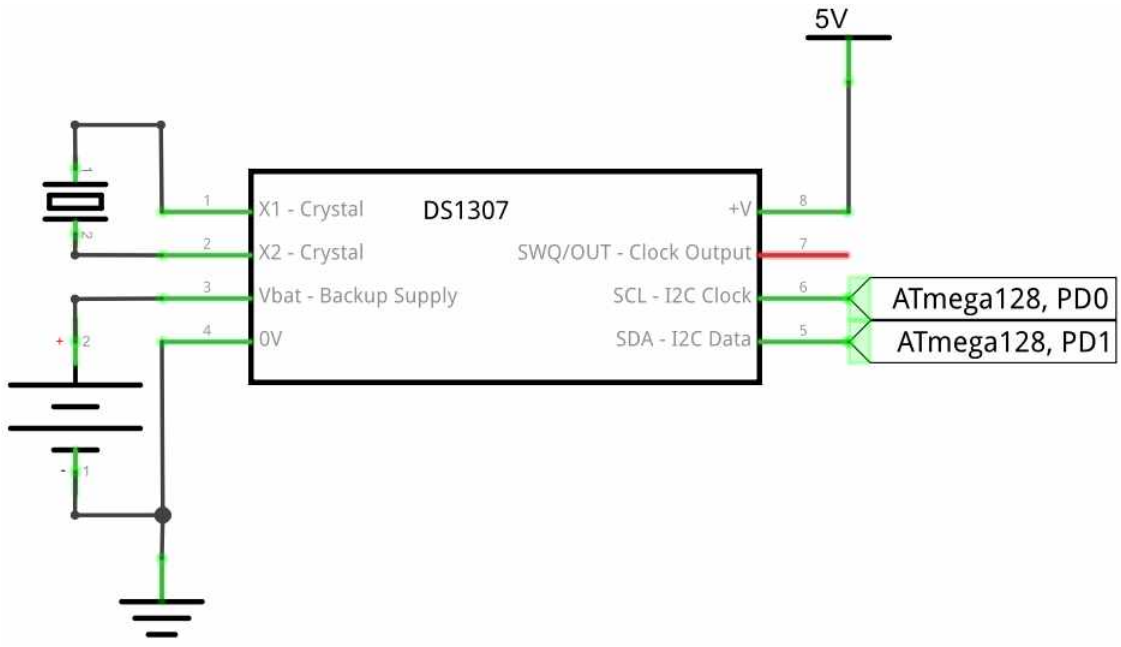

그림 8. 마이크로컨트롤러와 DS1307을 연결한 예

이 장에서는 DS1307 칩에 발진기와 전원 등을 추가하여 만들어진 Tiny RTC 모듈을 사용한

다. Tiny RTC 모듈의 I2C 주소는 0x68로 설정되어 있으며 변경할 수 없다.

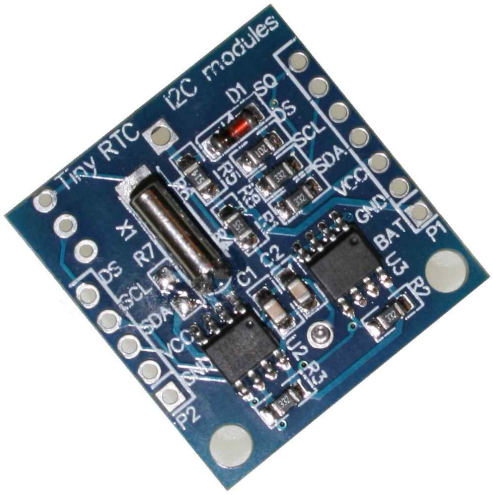

그림 9. Tiny RTC 모듈

Tiny RTC 모듈을 마이크로컨트롤러와 연결하기 위해서는 기본적으로 SCL, SDA, VCC, GND 4개의 연결만이 필요하다. 그림 8에서는 동일한 이름의 연결 단자가 좌우에 존재하며 어디에 연결하여도 무방하다. SQ는 SQW/OUT 출력 핀이며, DS는 디지털 온도 센서인 DS18B20의 출력 핀이다. 하지만 Tiny RTC 모듈에는 DS18B20를 연결할 수 있는 자리만 마련되어 있고 실제 포함되어 있지는 않은 경우가 대부분이다. 그림 10은 Tiny RTC 모듈을 ATmega128에 연결한 예를 나타낸다.

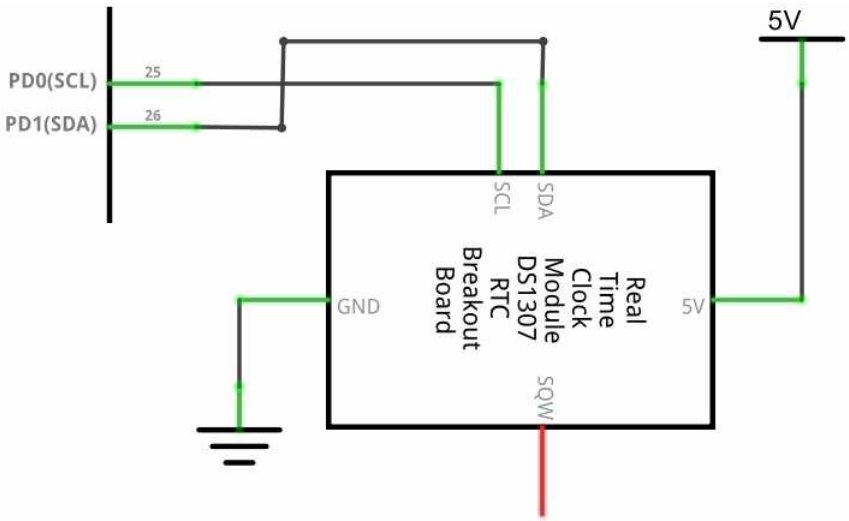

그림 10. Tiny RTC 연결 회로도

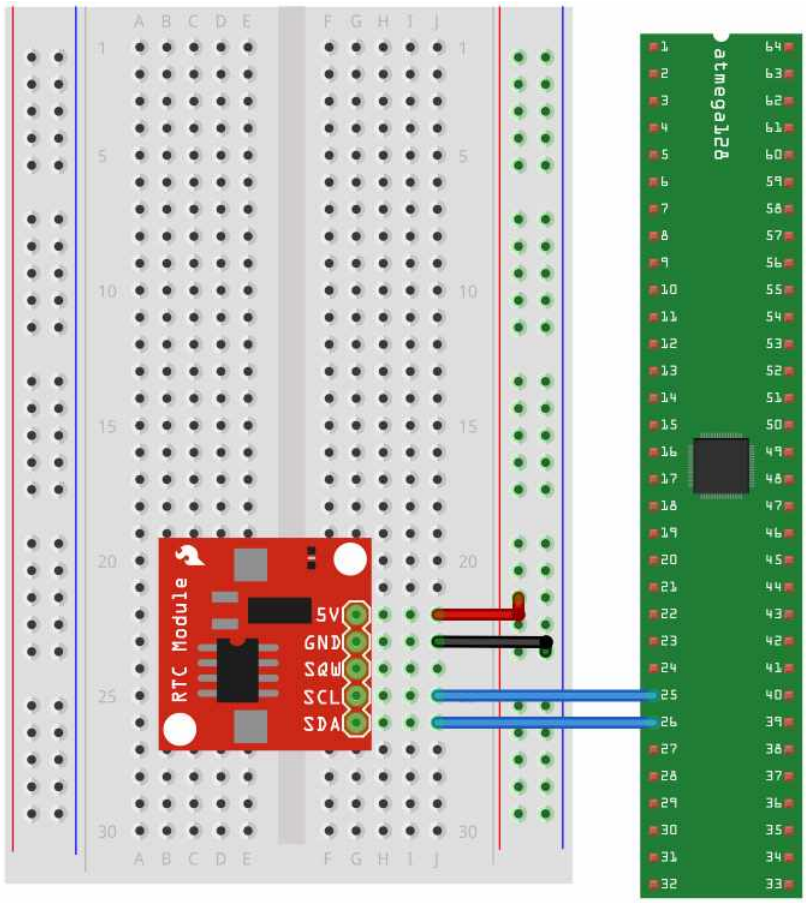

그림 11. Tiny RTC 연결 회로

DS1307 칩에는 64바이트의 메모리가 포함되어 있으며 이 중 시간 및 날짜와 관련하여 사용하는 메모리는 0번부터 6번까지의 메모리이며, 메모리 7번은 구형파 출력과 관계된 메모리이다. 나머지 56바이트 메모리는 범용으로 사용할 수 있는 메모리이다. 표 3은 DS1307 칩의 메모리 구조를 나타낸다.

주소	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0	내용	값 범위
0x00	CH	초(10의 자리)			초(1의 자리)				초	00-59
0x01	0	분(10의 자리)			분(1의 자리)				분	00-59
0x02	0	0	시(10의 자리)		시(1의 자리)				시	00-23
		1								1-12
0x03	0	0	0	0	0	요일			요일	01-07
0x04	0	0	일(10의 자리)		일(1의 자리)				일	01-31
0x05	0	0	0	월(10의 자리)	월(1의 자리)				월	01-12
0x06	연(10의 자리)				연(1의 자리)				연	00-99
0x07	OUT	0	0	SQWE	0	0	RS1	RS0	제어	-
0x08 ~ 0x3F									RAM	00h ~ FFh

표 3. DS1307 메모리 구조

날짜와 시간은 0번에서 6번까지의 메모리에 저장된다. 이 때 0번 메모리의 7번 비트인 CH(Clock Holt) 비트를 1로 설정되면 발진기가 동작하지 않으므로 0으로 설정하여야 한다. 또 한 가지 주의할 점은 숫자는 일반적인 이진수 방식이 아니라 BCD (Binary Coded Decimal) 형식으로 저장된다는 점이다. 12라는 숫자를 이진 형식으로 저장하는 경우 $0000\ 1100_2$가 된다. BCD 형식으로 저장하는 경우에는 4비트로 10진수 한 자리를 나타내므로 상위 4비트는 10의 자리를, 하위 4비트는 1의 자리를 나타내어 $0001_2\ 0010_2$로 저장된다.

7번 메모리는 구형파 출력을 제어한다. 7번 비트 OUT은 구형파 출력이 금지된 경우 SQ 핀으로 출력되는 출력 레벨을 나타내며 1인 경우 논리 1, 0인 경우 논리 0이 출력된다. 디폴트값은 0이다. 4번 비트 SQWE(Square Wave Enable)은 구형파 출력을 제어하는 비트로 1인 경우 구형파가 출력되고 0인 경우 7번 비트에 의해 설정된 레벨 값이 출력된다. 디폴트값은 0으로 구형파는 출력되지 않도록 설정되어있다. 1번과 0번 RS(Rate Select) 비트는 구형파의 출력 주파수를 결정한다. 비트 설정에 따라 출력되는 구형파의 주파수는 표 4와 같다.

SQWE	OUT	RS1	RS0	출력 주파수	비고
1	×	0	0	1Hz	
1	×	0	1	4.096KHz	
1	×	1	0	8.192KHz	
1	×	1	1	32.768KHz	
0	0	×	×	-	항상 0 출력
0	1	×	×	-	항상 1 출력

표 4. RS 설정에 따른 구형파 출력 주파수

DS1307에 현재 날짜와 시간을 설정하고 읽어오고 설정하는 순서는 그림 4 및 그림 5의 순서에 따른다. 먼저 I2C를 초기화하는 방법을 살펴보자.

코드 1. 현재 날짜의 시간 설정 및 읽기

```
#define I2C_SCL                 PD0
#define I2C_SDA                 PD1

void I2C_init(void)
{
  DDRC |= (1 << I2C_SCL);               // SCL 핀을 출력으로 설정
  DDRC |= (1 << I2C_SDA);               // SDA 핀을 출력으로 설정

  TWBR = 32;                            // I2C 클록 주파수 설정 200KHz

  TWCR = (1 << TWEN) | (1 << TWEA);     // I2C 활성화, ACK 허용
}
```

ATmega128에서 SCL과 SDA를 위한 핀이 정해져 있으므로 먼저 이 핀들을 출력으로 설정한다. 다음으로는 I2C 클록 주파수를 설정하여야 한다. I2C의 클록은 마스터가 책임지므로 슬레이브 모드로 설정된 장치는 여기에 영향을 받지 않으며, 메인 클록은 최소한 I2C 클록의 16배 이상이어야 한다. I2C 클록 주파수는 비트율을 결정하는 TWBR (TWI Bit Rate Register) 레지스터와, 분주율을 결정하는 TWPS (TWI Prescaler Bit) 비트에 의해 다음과 같디 결정된다.

$$SCL\ frequency = \frac{System\ Clock}{16 + 2 \cdot TWBR \cdot 4^{TWPS}}$$

TWBR (TWI Bit Rate Register) 레지스터의 구조는 다음과 같다.

비트	7	6	5	4	3	2	1	0
	TWBR7..0							
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. TWBR 레지스터 구조

분주율을 결정하는 TWPS 비트는 TWI 상태 레지스터인 TWSR (TWI Status Register) 레지스터에서 설정한다. TWSR 레지스터의 구조는 그림 13과 같다.

비트	7	6	5	4	3	2	1	0
	TWS7	TWS6	TWS4	TWS4	TWS3	-	TWPS1	TWPS0
읽기/쓰기	R	R	R	R	R	R	R/W	R/W
초깃값	1	1	1	1	1	0	0	0

그림 13. TWSR 레지스터 구조

비트 7에서 비트 3까지 5개 비트는 TWI의 상태를 나타내는 TWS 비트(TWI Status Bit)로 TWI의 전송 상태나 오류 등을 나타내기 위해 사용된다. 분주비는 TWPSn (n = 0, 1) (TWI Prescaler Bit) 비트에 의해 설정되며 설정값에 따른 분주비는 표 5와 같이 4의 거듭제곱으로 계산된다.

TWPS1	TWPS0	분주비
0	0	1
0	1	4
1	0	16
1	1	64

표 5. SCL 주파수의 분주비 설정

코드 1에서는 TWBR을 32로 설정하고 TWPSn은 디폴트값을 사용하였으므로 I2C의 주파수는 200KHz가 된다.

$$SCL\ frequency = \frac{16MHz}{16 + 2 \cdot 32 \cdot 4^{0}} = 200KHz$$

TWI의 전체적인 통신 과정은 TWI 제어 레지스터인 TWCR (TWI Control Register) 레지스터에 의해 제어된다. TWCR 레지스터의 구조는 그림 14와 같다.

비트	7	6	5	4	3	2	1	0
	TWINT	TWEA	TWSTA	TWSTO	TWWC	TWEN	-	TWIE
읽기/쓰기	R/W	R/W	R/W	R/W	R	R/W	R	R/W
초깃값	0	0	0	0	0	0	0	0

그림 14. TWCR 레지스터 구조

TWCR 레지스터 각 비트의 의미는 표 6과 같다.

비트	이름	설명
7	TWINT	TWI Interrupt Flag : TWI의 현재 작업이 완료되고 응용 소프트웨어의 응답을 기다릴 때 하드웨어에 의해 세트된다. SREG 레지스터의 I 비트가 세트되어 전역적으로 인터럽트가 허용되고 TWCR 레지스터의 TWIE 비트가 세트되어 있을 때, 현재 작업이 완료되어 TWINT 비트가 세트되면 인터럽트 서비스 루틴이 실행된다.
6	TWEA	TWI Enable Acknowledge Bit : ACK 펄스의 생성을 제어한다. TWEA 비트가 세트되어 있을 때, 다음과 같은 상황에서 ACK 펄스가 생성된다. 1. 슬레이브 모드에서, 자신의 주소를 수신한 경우 2. 슬레이브 모드에서, TWAR 레지스터의 TWGCE 비트가 세트되어 있고, 모든 슬레이브로 전달되는 일반 호출(general call)을 수신한 경우 3. 마스터나 슬레이브 모드에서, 한 바이트의 데이터를 수신한 경우

5	TWSTA	TWI Start Condition Bit : 마스터 모드에서 전송을 시작하기 위해 TWSTA 비트를 세트한다.
4	TWSTO	TWI Stop Condition Bit : 마스터 모드에서 전송을 끝내기 위해 TWSTO 비트를 세트한다.
3	TWWC	TWI Write Collision Flag : TWINT 비트가 0인 경우, 즉, 데이터가 전송 작업이 완료되지 않은 경우 TWI 데이터 레지스터인 TWDR 레지스터에 데이터를 기록하려고 하면 세트된다. TWINT 비트가 1인 경우 TWDR 레지스터에 데이터를 기록하면 클리어 된다.
2	TWEN	TWI Enable Bit : TWI를 활성화시킨다.
1	-	-
0	TWIE	TWI Interrupt Enable : TWI 데이터 전송이 완료된 경우 인터럽트 발생을 허용한다. SREG 레지스터의 I 비트가 세트되고, TWIE 비트가 세트된 경우 전송이 완료되면 인터럽트가 발생한다.

표 6. TWCR 레지스터 비트

코드 1에서는 TWI를 활성화시키고 ACK 펄스 발생을 허용하고 있으며, 인터럽트는 사용하지 않는다.

코드 2는 I2C 전송을 시작하기 위해 시작 비트를 전송하고 대기하는 함수이다. I2C_start 함수에서 시작 조건 비트 TWSTA와 TWINT 비트를 세트하지만, 실제로 TWINT 비트가 세트되는 시점은 시작 조건이 전송되고 난 이후이므로 시작 조건이 실제 전송 완료되기를 기다려야 한다.

코드 2. TWI 시작 함수

```
void I2C_start(void)
{
   TWCR = _BV(TWINT) | _BV(TWSTA) | _BV(TWEN) | _BV(TWEA);

   while( !(TWCR & (1 << TWINT)) );        // 시작 완료 대기
}
```

시작 조건이 전송된 후 데이터를 슬레이브로 전송되는 경우를 생각해 보자. 먼저 I2C 주소를 전송하고 다음으로 데이터를 전송한다. 의미상으로 이들은 서로 다르지만 전송되는 방식은 동

일하다. 즉, 1바이트 데이터를 전송하고 ACK 비트를 수신될 때까지 대기한다. 코드 3은 TWI를 통해 1바이트의 데이터를 전송하고 ACK 비트를 수신하였는지 확인하는 함수이다.

코드 3. TWI 바이트 송신 함수

```
void I2C_transmit(uint8_t data)
{
  TWDR = data;
  TWCR = _BV(TWINT) | _BV(TWEN) | _BV(TWEA);

  while( !(TWCR & (1 << TWINT)) );        // 전송 완료 대기
}
```

바이트 전송 함수는 시작 함수인 I2C_start와 마찬가지로 TWINT 비트를 세트하고 실제 전송이 완료되고 ACK 비트를 수신할 때까지, 즉, TWINT 비트가 세트될 때까지 대기한다. TWDR (TWI Data Register) 레지스터는 송수신되는 데이터가 저장되는 레지스터로 TWDR 레지스터의 구조는 그림 15와 같다.

비트	7	6	5	4	3	2	1	0
	TWD7..0							
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초깃값	1	1	1	1	1	1	1	1

그림 15. TWDR 레지스터 구조

필요한 정보가 데이터 전송이 완료된 경우에는 정지 비트를 전송하여야 한다. 코드 4는 정지 조건을 전송하여 TWI 통신을 종료하는 함수이다. I2C_start 함수와는 달리 I2C_stop 함수에서는 TWINT 비트가 세트될 때까지 대기하지 않아도 된다.

코드 4. TWI 정지 함수

```
void I2C_stop(void)
{
```

```
    TWCR = _BV(TWINT) | _BV(TWSTO) | _BV(TWEN) | _BV(TWEA);
}
```

데이터를 수신하는 경우도 송신하는 경우와 유사하다. 다만 슬레이브의 I2C 주소를 지정할 때 LSB는 1로 설정되어야 한다는 점에서 차이가 있다. 코드 5는 1바이트의 데이터를 수신하는 함수이다. 코드 5는 코드 3과 설정이 동일하다. 다만 코드 3에서는 데이터 송신을 위해 TWDR 레지스터에 값을 대입하고 있는 반면, 코드 5에서는 데이터 수신을 위해 TWDR 레지스터의 값을 읽는 차이가 있다. 그림 4와 그림 5에서 알 수 있듯이 I2C 통신에서 송신과 수신은 주소를 전송할 때 LSB에 의해 이미 결정되어 있으므로 데이터 송신과 수신 함수에서의 설정에는 차이가 없다.

코드 5. TWI 바이트 수신 함수

```
uint8_t I2C_receive(void) {
    TWCR = _BV(TWINT) | _BV(TWEN) | _BV(TWEA);

    while( !(TWCR & (1 << TWINT)) );        // 수신 완료 대기

    return TWDR;
}
```

I2C 사용을 위해 필요한 기본적인 함수는 여기까지이며, I2C를 사용하는 장치에 따라 위의 함수들을 이용하여 데이터를 송수신하는 방법에는 차이가 있으므로 장치에 따른 데이터시트를 참고하여야 한다.[1] 위의 코드들로 I2C 통신 위한 라이브러리를 다음과 같이 작성할 수 있다.

코드 6. I2C_RTC.h

```
#ifndef I2C_RTC_H_
#define I2C_RTC_H_
```

1) 한 가지 더 언급하고 싶은 점은 위의 코드들에서 코드를 간단히 하기 위해 오류 검사는 수행하지 않았다는 점이다. 오류의 발생 빈도가 높지는 않지만 보다 안전한 코드를 작성하기 위해서는 오류 검사를 수행하여야 하며, 오류 검사에 대해서는 데이터시트의 예를 참고하도록 한다.

```
#define I2C_SCL                 PD0
#define I2C_SDA                 PD1

#include <avr/io.h>

void I2C_init(void);                        // I2C 초기화
void I2C_start(void);                       // I2C 시작
void I2C_transmit(uint8_t data);            // 1바이트 전송
uint8_t I2C_receive(void);                  // 1바이트 수신
void I2C_stop(void);                        // I2C 정지

#endif
```

코드 7. I2C_RTC.c

```
#include "I2C_TRC.h"

void I2C_init(void)
{
  DDRC |= (1 << I2C_SCL);       // SCL 핀을 출력으로 설정
  DDRC |= (1 << I2C_SDA);       // SDA 핀을 출력으로 설정

  TWBR = 32;                    // I2C 클록 주파수 설정 200KHz

  TWCR = (1 << TWEN) | (1 << TWEA);          // I2C 활성화, ACK 허용
}

void I2C_start(void)
{
  TWCR = _BV(TWINT) | _BV(TWSTA) | _BV(TWEN) | _BV(TWEA);

  while( !(TWCR & (1 << TWINT)) );      // 시작 완료 대기
}
```

```
void I2C_transmit(uint8_t data)
{
  TWDR = data;
  TWCR = _BV(TWINT) | _BV(TWEN) | _BV(TWEA);

  while( !(TWCR & (1 << TWINT)) );        // 전송 완료 대기
}

uint8_t I2C_receive(void)
{
  TWCR = _BV(TWINT) | _BV(TWEN) | _BV(TWEA);

  while( !(TWCR & (1 << TWINT)) );        // 수신 완료 대기

  return TWDR;
}

void I2C_stop(void)
{
  TWCR = _BV(TWINT) | _BV(TWSTO) | _BV(TWEN) | _BV(TWEA);
}
```

코드 6 및 코드 7과, UART 통신을 위한 UART1.h 및 UART1.c를 사용하여 RTC 모듈의 시간을 설정하고 2초 후에 이를 다시 읽어 출력하는 코드의 예가 코드 8이다. 코드 8에서는 일반적인 이진수 형식의 숫자와 BCD 형식의 숫자를 상호 변환하기 위해 bcd_to_decimal 함수와 decimal_to_bcd 함수를 사용하고 있다.

코드 8. RTC 모듈의 시간 설정 및 읽기

```
#define F_CPU 16000000UL

#include <avr/io.h>
#include <util/delay.h>
#include <stdio.h>
```

```
#include "UART1.h"
#include "I2C_RTC.h"

FILE OUTPUT
  = FDEV_SETUP_STREAM(UART1_transmit, NULL, _FDEV_SETUP_WRITE);
FILE INPUT
  = FDEV_SETUP_STREAM(NULL, UART1_receive, _FDEV_SETUP_READ);

uint8_t bcd_to_decimal(uint8_t bcd)        // BCD 형식 -> 이진수 형식
{
  return (bcd >> 4) * 10 + (bcd & 0x0F);
}

uint8_t decimal_to_bcd(uint8_t decimal)    // 이진수 형식 -> BCD 형식
{
  return ( ((decimal / 10) << 4) | (decimal % 10) );
}

int main(void)
{
  uint8_t i;

  I2C_init();                              // I2C 초기화
  UART1_init();                            // UART 초기화

  stdout = &OUTPUT;
  stdin = &INPUT;

  uint8_t address = 0x68;                  // RTC 모듈의 I2C 주소

  // 초, 분, 시, 요일, 일, 월, 년
  // 2014년 9월 1일 월요일 12시 34분 56초
  uint8_t data[] = {56, 34, 12, 2, 1, 9, 14};

  // RTC 모듈에 시간 설정
```

```
I2C_start();                                // I2C 시작
I2C_transmit(address << 1);                 // I2C 주소 전송. 쓰기 모드
// RTC에 데이터를 기록할 메모리 시작 주소 전송
I2C_transmit(0);

printf("* Setting RTC module...\r\n");
for(i = 0; i < 7; i++){
  printf(" %dth byte writen...\r\n", i);

  I2C_transmit(decimal_to_bcd(data[i])); // 시간 설정
}

I2C_stop();                                 // I2C 정지

_delay_ms(2000);                            // 2초 대기

I2C_start();                                // I2C 시작
I2C_transmit(address << 1);                 // I2C 주소 전송. 쓰기 모드
// RTC에서 데이터를 읽어올 메모리 시작 주소 전송
I2C_transmit(0);
I2C_stop();                                 // I2C 정지

I2C_start();                                // I2C 읽기 모드로 다시 시작
I2C_transmit( (address << 1) + 1 );         // I2C 주소 전송. 읽기 모드

printf("* Time/Date Retrieval...\r\n");

printf(" %d second\r\n", bcd_to_decimal(I2C_receive()));
printf(" %d minute\r\n", bcd_to_decimal(I2C_receive()));
printf(" %d hour\r\n", bcd_to_decimal(I2C_receive()));
printf(" %d day of week\r\n", bcd_to_decimal(I2C_receive()));
printf(" %d day\r\n", bcd_to_decimal(I2C_receive()));
printf(" %d month\r\n", bcd_to_decimal(I2C_receive()));
printf(" %d year\r\n", bcd_to_decimal(I2C_receive()));

I2C_stop();                                 // I2C 정지
```

```
    while(1);
    return 0;
}
```

그림 16. 코드 8 실행 결과

RTC 모듈에 전용 전원이 연결되어 있다면 한 번 시간을 설정한 이후 필요할 때마다 현재 시간을 읽어서 알아낼 수 있다.