

## Chapter 110. 타이머/카운터

### 1. 타이머/카운터

타이머/카운터는 입력되는 펄스를 세는 장치, 즉, 카운터이다. 하지만 주기가 일정한 펄스가 입력된다면 펄스의 개수를 세서 시간을 측정할 수 있으므로 타이머의 역할도 할 수 있어 '타이머/카운터'라고 불린다. ATmega328에서는 8비트 타이머/카운터 2개(Timer/Counter 0, 2)와 16비트 타이머/카운터 1개(Timer/Counter 1)를 제공하고 있다. 타이머/카운터는 다양한 용도로 활용이 가능하지만 그 구조가 복잡하여 사용하기 쉽지 않은 점도 있다. 먼저 카운터/타이머의 전체적인 구조를 살펴보자.

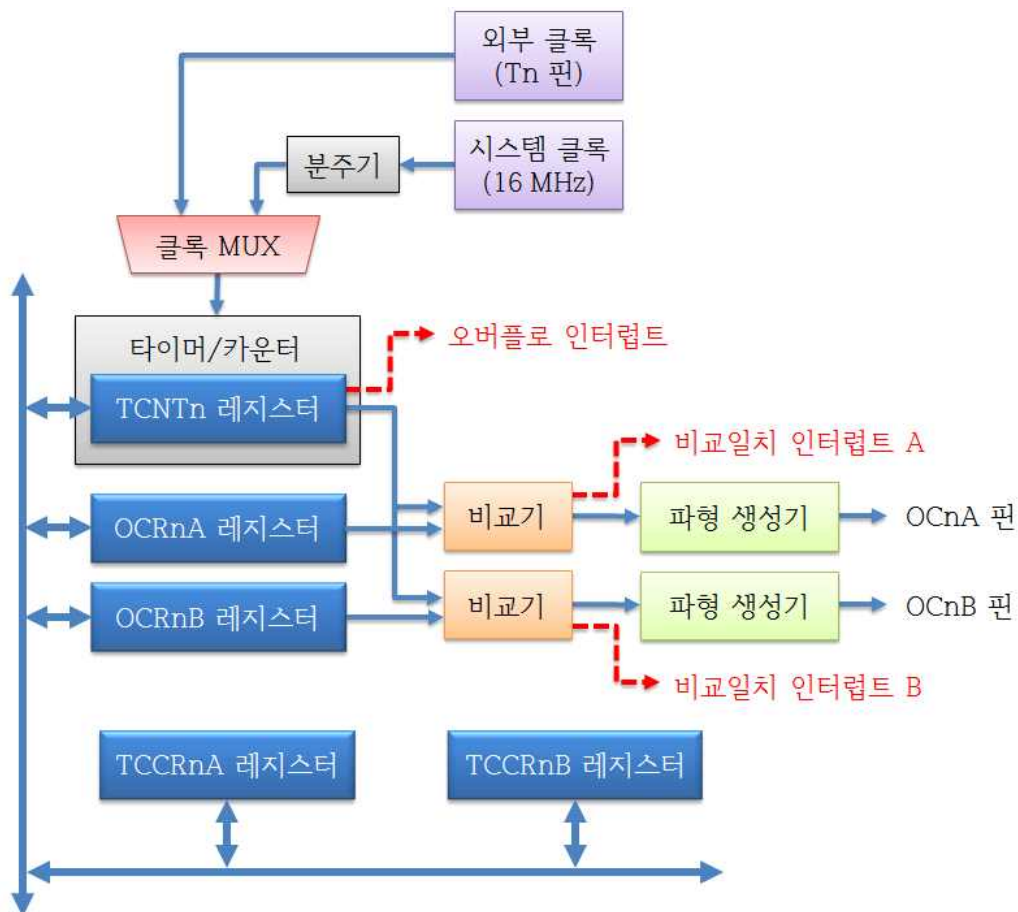


그림 1. 타이머/카운터 블록 다이어그램

타이머/카운터는 기본적으로 입력 펄스를 센다. 펄스를 세기 위해서는 입력 펄스가 존재해야 하며 이는 마이크로컨트롤러의 시스템 클록이나 외부에서 주어지는 클록 중에서 선택해서 사용할 수 있다. 8비트 타이머/카운터의 경우 0에서 255까지만 셀 수 있다. ATmega328의 내부 클록은 16MHz 속도로 동작하므로 256번 카운트하는 시간은  $256/16\text{M} = 0.016\text{ms}$ 로 긴 시간 간격을 가지는 타이머를 만들 수 없다. 따라서 분주기(prescaler)를 사용하여 주기가 긴 클록을 생성하여 긴 시간을 측정할 수 있도록 하고 있다.

현재까지 센 펄스의 수는 TCNTn ( $n = 0, 1, 2$ ) 레지스터(Timer/Counter Register)에 기록된다. 펄스의 개수가 셀 수 있는 최댓값을 넘어서면 TCNTn 레지스터는 0으로 초기화되며 이때 오버플로 인터럽트(Overflow Interrupt)가 발생하게 된다. 오버플로가 발생하는 시간은 시스템 클록과 분주기에 의해 결정되므로 정밀한 시간 간격을 제어하기는 힘들다. 따라서 보다 정밀한 시간 측정을 위해서 사용할 수 있도록 비교일치 인터럽트(Compare Match Interrupt)가 준비되어 있다. 비교일치 인터럽트는 TCNTn 레지스터에 저장된 현재까지의 펄스 개수가 미리 설정된 값과 동일할 때 발생한다. 인터럽트가 발생하는 시점을 나타내는 '미리 설정된 값'은 OCRnx ( $n = 0, 1, 2; x = A, B$ ) 레지스터(Output Compare Register)에 저장된다. 그림 1에서 알 수 있듯이 각 타이머/카운터 별로 두 개의 비교값을 설정할 수 있다. 비교일치 인터럽트 발생과는 별도로 비교일치가 발생하면 지정된 핀을 통해 지정된 파형을 출력하도록 할 수 있다. 타이머/카운터의 세부 동작들은 TCCRnx ( $n = 0, 1, 2; x = A, B$ ) 레지스터(Timer/Counter Control Register)를 통해 제어된다.

## 2. 오버플로 인터럽트

먼저 가장 간단한 오버플로 인터럽트를 사용해 보자. 코드 1은 0.5초 간격으로 PB5 핀(아두이노 13번 핀)에 연결된 LED를 점멸하는 코드이다.

코드 1. 오버플로 인터럽트를 이용한 Blink

```
#include <avr/io.h>
#include <avr/interrupt.h>

int count = 0;                // 오버플로가 발생한 횟수
int state = 0;                // LED 점멸 상태

ISR(TIMER0_OVF_vect)
```

```

{
    count++;
    if(count == 32){                // 오버플로 32회 발생 = 0.5초 경과
        count = 0;                // 카운터 초기화
        state = !state;           // LED 상태 반전
        if(state) PORTB = 0xFF;   // LED 켜기
        else PORTB = 0x00;        // LED 끄기
    }
}

int main(void)
{
    DDRB = 0x20;                  // PB5 핀을 출력으로 설정
    PORTB = 0x00;                // LED는 끈 상태에서 시작

    TCCR0B |= (1 << CS02) | (1 << CS00); // 분주비를 1024로 설정

    TIMSK0 |= (1 << TOIE0);       // 오버플로 인터럽트 허용
    sei();                        // 전역적으로 인터럽트 허용

    while(1){ }
}

```

LED를 일정한 시간 간격으로 점멸하는 코드는 그다지 신기할 것이 없다. 하지만 코드 1은 타이머/카운터 0번의 오버플로 인터럽트를 사용하여 구현하였다는 데 의미가 있다. 현재까지 센 펄스의 개수는 TCNT0에 기록된다. 타이머/카운터 0번은 8비트 타이머/카운터 이므로 0에서 255까지 카운트할 수 있다. TCNT0 레지스터의 구조는 그림 2와 같다.

| 비트    | 7          | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-------|------------|-----|-----|-----|-----|-----|-----|-----|
|       | TCNT0[7:0] |     |     |     |     |     |     |     |
| 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 2. TCNT0 레지스터 구조

타이머/카운터는 디폴트로 아무런 동작도 하지 않으므로 카운트를 시작하기 위해서는 카운트를 시작하도록 설정하여야 한다. 코드 1에서 카운트를 시작하도록 하는 부분은 TCCR0B 레지스터(Timer/Counter Control Register B)에 분주비를 설정하는 부분이다. TCCR0B 레지스터의 구조는 그림 3과 같다.

| 비트    | 7     | 6     | 5 | 4 | 3     | 2    | 1    | 0    |
|-------|-------|-------|---|---|-------|------|------|------|
|       | FOC0A | FOC0B | - | - | WGM02 | CS02 | CS01 | CS00 |
| 읽기/쓰기 | W     | W     | R | R | R/W   | R/W  | R/W  | R/W  |
| 초깃값   | 0     | 0     | 0 | 0 | 0     | 0    | 0    | 0    |

그림 3. TCCR0B 레지스터 구조

7번 비트 FOC0A (Force Output Compare A) 비트와 6번 비트 FOC0B 비트는 비교일치 인터럽트에 사용되는 비트로 코드 1에서는 사용하지 않는다. 3번 비트인 WGM02 (Wave Generation Mode) 비트 역시 비교일치 인터럽트에서 사용되는 비트이다. 코드 1에서 설정한 부분은 비트 2에서 비트 0까지의 CS0n (n = 0, 1, 2) (Clock Select) 비트로 CS0n 비트 설정에 따라 선택되는 클록은 표 1과 같다.

| CS02 | CS01 | CS00 | 설명                              |
|------|------|------|---------------------------------|
| 0    | 0    | 0    | 클록 소스 없음 (타이머/카운터 정지)           |
| 0    | 0    | 1    | 분주비 1                           |
| 0    | 1    | 0    | 분주비 8                           |
| 0    | 1    | 1    | 분주비 64                          |
| 1    | 0    | 0    | 분주비 256                         |
| 1    | 0    | 1    | 분주비 1024                        |
| 1    | 1    | 0    | T0 핀의 외부 클록 소스 사용. 하강 에지에서 동작함. |
| 1    | 1    | 1    | T0 핀의 외부 클록 소스 사용. 상승 에지에서 동작함. |

표 1. CS0n (n = 0, 1, 2) 비트 설정에 따른 클록 선택

CS0n의 디폴트값은 '000<sub>2</sub>'으로 타이머/카운터는 동작하지 않는 상태이다. 코드 1에서는 CS0n에 '101<sub>2</sub>' 값을 설정함으로써 분주비가 1024가 되도록 설정하였다. 아두이노 우노 보드의 경우 16MHz 클록을 사용하므로 분주비가 1024가 되면 16KHz 클록이 타이머/카운터에

공급된다. 타이머/카운터 0번이 255까지 세는데 걸리는 시간은 얼마나 될까? 1초에 16K 개의 펄스가 타이머/카운터 0번에 주어지므로 256개 펄스를 세는 시간은 1/64초이다. 따라서 코드 1에서는 256개 펄스를 세었을 때 발생하는 오버플로 인터럽트를 32번 체크하여 0.5초 경과를 계산하고 LED 상태를 반전시키고 있다. 인터럽트 역시 디폴트로 발생하지 않도록 설정되어 있으므로 타이머/카운터 0번의 인터럽트를 활성화시켜 주어야 하며 이는 TIMSK0 레지스터(Timer/Counter Interrupt Mask Register)에서 설정해 주고 있다. 그림 4는 TIMSK0 레지스터의 구조를 나타낸다.

| 비트    | 7 | 6 | 5 | 4 | 3 | 2      | 1      | 0     |
|-------|---|---|---|---|---|--------|--------|-------|
|       | - | - | - | - | - | OCIE0B | OCIE0A | TOIE0 |
| 읽기/쓰기 | R | R | R | R | R | R/W    | R/W    | R/W   |
| 초깃값   | 0 | 0 | 0 | 0 | 0 | 0      | 0      | 0     |

그림 4. TIMSK0 레지스터 구조

타이머/카운터 0번과 관련된 인터럽트에는 카운터가 카운트할 수 있는 범위를 넘어서는 경우 발생하는 오버플로 인터럽트와, 미리 설정된 값과 동일한 경우 발생하는 비교일치 인터럽트가 있다. 비교일치 인터럽트의 경우 서로 다른 2개의 값을 지정할 수 있으므로 2개가 가능하므로 타이머/카운터 0번에서 사용할 수 있는 인터럽트는 모두 3종류가 된다.

| 벡터 번호 | 인터럽트        | 벡터 이름             | 인터럽트 허용 비트 |                                         |
|-------|-------------|-------------------|------------|-----------------------------------------|
| 15    | 비교일치 인터럽트 A | TIMER0_COMPA_vect | OCIE0A     | Output Compare Match A Interrupt Enable |
| 16    | 비교일치 인터럽트 B | TIMER0_COMPB_vect | OCIE0B     | Output Compare Match B Interrupt Enable |
| 17    | 오버플로 인터럽트   | TIMER0_OVF_vect   | TOIE0      | Overflow Interrupt Enable               |

표 2. 타이머/카운터 0번 인터럽트

### 3. 비교일치 인터럽트

코드 1과 동일한 동작을 하는 프로그램을 비교일치 인터럽트를 사용해서 작성한 예가 코드 2이다.

#### 코드 2. 비교일치 인터럽트를 이용한 Blink 1

```
#include <avr/io.h>
#include <avr/interrupt.h>

int count = 0;           // 비교일치가 발생한 횟수
int state = 0;           // LED 점멸 상태

ISR(TIMER0_COMPA_vect)
{
    count++;
    TCNT0 = 0;           // 자동으로 0으로 변하지 않는다
    if(count == 64){     // 비교일치 64회 발생 = 0.5초 경과
        count = 0;       // 카운터 초기화
        state = !state;   // LED 상태 반전
        if(state) PORTB = 0xFF; // LED 켜기
        else PORTB = 0x00; // LED 끄기
    }
}

int main(void)
{
    DDRB = 0x20;         // PB5 핀을 출력으로 설정
    PORTB = 0x00;        // LED는 끈 상태에서 시작

    TCCR0B |= (1 << CS02) | (1 << CS00); // 분주비를 1024로 설정

    OCR0A = 128;         // 비교일치 기준값

    TIMSK0 |= (1 << OCIE0A); // 비교일치 인터럽트 허용
    sei();               // 전역적으로 인터럽트 허용

    while(1){ }
```

```
}

```

비교일치 인터럽트를 사용하기 위해서는 먼저 OCR0A 또는 OCR0B 레지스터에 비교값을 설정해주어야 한다. OCR0x (x = A, B) 레지스터의 구조는 그림 5와 같다.

| 비트    | 7         | 6   | 5   | 4   | 3   | 2   | 1   | 0   |
|-------|-----------|-----|-----|-----|-----|-----|-----|-----|
|       | OCR0[7:0] |     |     |     |     |     |     |     |
| 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 5. OCR0x (x = A, B) 레지스터 구조

코드 2에서는 OCR0A 레지스터의 값을 128로 설정하여 오버플로 인터럽트보다 2배 빨리 인터럽트가 발생하도록 하였다. 따라서 코드 1에서는 오버플로 인터럽트가 32회 발생한 경우 0.5초가 경과한 것으로 판단하는 반면, 코드 2에서는 비교일치 인터럽트가 64회 발생한 경우 0.5초가 경과한 것으로 판단하고 있다. 한 가지 주의할 점은 타이머/카운터 0번의 경우 비교일치 인터럽트가 발생하여도 현재까지의 카운트는 자동으로 초기화 되지 않는다는 점이다. 따라서 인터럽트 서비스 루틴에서 비교일치 인터럽트가 발생할 때마다 TNCT0 레지스터의 값을 0으로 설정해주고 있다.

코드 2에서는 LED를 점멸하는 코드가 인터럽트 서비스 루틴에 포함되어 있다. 하지만 인터럽트 서비스 루틴은 가능한 짧게 작성하는 것이 좋다는 점을 기억하는가? 인터럽트 서비스 루틴이 실행되는 동안에는 인터럽트가 발생하지 못하며 인터럽트 서비스 루틴 실행에 시간이 걸리므로 정확한 타이밍을 계산하기 어려울 수 있다. 코드 2에서 LED를 점멸하는 코드를 main 함수의 이벤트 루프로 옮겨온 코드가 코드 3이다. 단, 이 경우 count 변수는 volatile로 선언되어야 한다는 점을 잊지 말아야 한다.

#### 코드 3. 비교일치 인터럽트를 이용한 Blink 2

```
#include <avr/io.h>
#include <avr/interrupt.h>

```

```

volatile int count = 0;           // 비교일치가 발생한 횟수
int state = 0;                   // LED 점멸 상태

ISR(TIMER0_COMPA_vect)
{
    count++;
    TCNT0 = 0;                   // 자동으로 0으로 변하지 않는다
}

int main(void)
{
    DDRB = 0x20;                 // PB5 핀을 출력으로 설정
    PORTB = 0x00;                // LED는 끈 상태에서 시작

    TCCR0B |= (1 << CS02) | (1 << CS00); // 분주비를 1024로 설정

    OCR0A = 128;                 // 비교일치 기준값

    TIMSK0 |= (1 << OCIE0A);     // 비교일치 인터럽트 허용
    sei();                       // 전역적으로 인터럽트 허용

    while(1){
        if(count == 64){         // 비교일치 64회 발생 = 0.5초 경과
            count = 0;            // 카운터 초기화
            state = !state;       // LED 상태 반전
            if(state) PORTB = 0xFF; // LED 켜기
            else PORTB = 0x00;    // LED 끄기
        }
    }
}

```

지금까지는 8비트 타이머/카운터를 사용하였으므로 카운트할 수 있는 최댓값이 255로 오버플로 인터럽트가 32회 발생하여야 0.5초가 경과하였다. ATmega328에는 16비트 타이머/카운터 역시 1개가 존재하고 있으므로 긴 시간 주기가 필요한 경우에는 16비트 타이머/카운터를 사용하는 것이 유리하다. 코드 4는 16비트 타이머/카운터 1번을 사용하여 0.5초 간격으로 LED

를 점멸하는 예이다.

#### 코드 4. 비교일치 인터럽트를 이용한 Blink 3

```
#include <avr/io.h>
#include <avr/interrupt.h>

int state = 0; // LED 점멸 상태

ISR(TIMER1_COMPA_vect)
{
    TCNT1 = 0; // 자동으로 0으로 변하지 않는다

    state = !state; // LED 상태 반전
    if(state) PORTB = 0xFF; // LED 켜기
    else PORTB = 0x00; // LED 끄기
}

int main(void)
{
    DDRB = 0x20; // PB5 핀을 출력으로 설정
    PORTB = 0x00; // LED는 끈 상태에서 시작

    TCCR1B |= (1 << CS12) | (1 << CS10); // 분주비를 1024로 설정

    OCR1A = 0x2000; // 비교일치 기준값

    TIMSK1 |= (1 << OCIE1A); // 비교일치 인터럽트 허용
    sei(); // 전역적으로 인터럽트 허용

    while(1){ }
}
```

타이머/카운터 1번은 16비트 타이머/카운터이므로 카운트 값을 저장하는 TCNT1 레지스터는 16비트 길이를 가진다. 하지만 ATmega328은 8비트 마이크로컨트롤러이므로 TCNT1 레지스터는 16비트 값의 상위 바이트에 해당하는 TCNT1H 레지스터와 하위 바이트에 해당하는

TCNT1L 레지스터로 이루어지는 가상의 레지스터에 해당한다. 그림 6은 TCNT1 레지스터의 구조를 나타낸다.

| 비트     | 15           | 14  | 13  | 12  | 11  | 10  | 9   | 8   |
|--------|--------------|-----|-----|-----|-----|-----|-----|-----|
| TCNT1H | TCNT1 [15:8] |     |     |     |     |     |     |     |
| TCNT1L | TCNT1 [7:0]  |     |     |     |     |     |     |     |
| 비트     | 7            | 6   | 5   | 4   | 3   | 2   | 1   | 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 6. TCNT1 레지스터 구조

비교일치 값을 설정하는 OCR1x (x = A, B) 레지스터 역시 16비트이며 그 구조는 TCNT1과 유사하게 상위 바이트를 나타내는 OCR1xH 레지스터와 하위 바이트를 나타내는 OCR1xL 레지스터로 구성된다.

| 비트     | 15           | 14  | 13  | 12  | 11  | 10  | 9   | 8   |
|--------|--------------|-----|-----|-----|-----|-----|-----|-----|
| OCR1xH | OCR1x [15:8] |     |     |     |     |     |     |     |
| OCR1xL | OCR1x [7:0]  |     |     |     |     |     |     |     |
| 비트     | 7            | 6   | 5   | 4   | 3   | 2   | 1   | 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 7. OCR1x (x = A, B) 레지스터 구조

16MHz 클록을 1024로 분주한 경우 타이머/카운터에 공급되는 클록은 1초에 16K개가 되며 0.5초가 경과하기 위해서는 8K개의 클록이 필요하다. 따라서 OCR1A 레지스터에 비교일치가 발생하는 값은  $8 \times 2^{10} = 8192 = 0x2000$ 로 설정되어 있다.

#### 4. 파형 출력

비교일치가 발생하는 경우 인터럽트 이외에도 지정된 핀을 통해 신호를 출력하는 것이 가능하며, 그림 1에서 파형 생성기를 통해 OCnx (n = 0, 1, 2; x = A, B) 핀에 연결된 부분이 여기

에 해당한다.

| 타이머/카운터 | 파형 출력 핀 |     | 아두이노 핀 번호 |
|---------|---------|-----|-----------|
| 0       | OC0A    | PD6 | 6         |
|         | OC0B    | PD5 | 5         |
| 1       | OC1A    | PB1 | 9         |
|         | OC1B    | PB2 | 10        |
| 2       | OC2A    | PB3 | 11        |
|         | OC2B    | PD3 | 3         |

표 3. 비교일치 인터럽트 시 파형 출력 핀

코드 5는 1번 타이머/카운터에서 비교일치 인터럽트가 발생하는 경우 파형 생성 기능을 이용하여 OC1A 핀(아두이노 9번 핀)에 연결된 LED를 0.5초 간격으로 점멸하는 코드이다.

#### 코드 5. 파형생성 1

```
#include <avr/io.h>
#include <avr/interrupt.h>

ISR(TIMER1_COMPA_vect)
{
    TCNT1 = 0;                // 자동으로 0으로 변하지 않는다
}

int main(void)
{
    TCCR1B |= (1 << CS12) | (1 << CS10); // 분주비를 1024로 설정

    OCR1A = 0x2000;           // 비교일치 기준값

    // 비교일치 인터럽트 발생 시 OC1A 핀의 출력을 반전
    TCCR1A |= (1 << COM1A0);

    DDRB |= (1 << PB1);       // OC1A 핀(PB1 핀)을 출력으로 설정
```

```

TIMSK1 |= (1 << OCIE1A);           // 비교일치 인터럽트 허용
sei();                             // 전역적으로 인터럽트 허용

while(1){ }
}

```

디폴트로 비교일치 인터럽트가 발생하여도 OC1A 핀에는 아무런 신호가 출력되지 않는다. OC1A 핀의 동작은 TCCR1A 레지스터(Timer/Counter Control Register)에 의해 설정될 수 있다. TCCR1A 레지스터의 구조는 그림 8과 같다.

| 비트    | 7      | 6      | 5      | 4      | 3 | 2 | 1     | 0     |
|-------|--------|--------|--------|--------|---|---|-------|-------|
|       | COM1A1 | COM1A0 | COM1B1 | COM1B0 | - | - | WGM11 | WGM10 |
| 읽기/쓰기 | R/W    | R/W    | R/W    | R/W    | R | R | R/W   | R/W   |
| 초깃값   | 0      | 0      | 0      | 0      | 0 | 0 | 0     | 0     |

그림 8. TCCR1A 레지스터 구조

1번 비트 WGM11 비트(Waveform Generation Mode)와 0번 비트 WGM10 비트는 PWM (Pulse Width Modulation) 형식의 파형 생성을 위해 사용되는 비트로 코드 5에서는 디폴트 값을 사용한다. 비교일치 인터럽트 A에서 파형 생성을 위해 사용되는 비트는 7번 비트 COM1A1 (Compare Output Mode) 비트와 6번 비트 COM1A0 비트이다. COM1An 비트 설정에 따른 OC1A 핀의 동작은 표 4와 같다.

| COM1A1 | COM1A0 | 설명                                                   |
|--------|--------|------------------------------------------------------|
| 0      | 0      | OC1A 핀으로 데이터가 출력되지 않으며 OC1A 핀은 일반적인 범용 입출력 핀으로 동작한다. |
| 0      | 1      | 비교일치가 발생하면 OC1A 핀의 출력은 반전된다.                         |
| 1      | 0      | 비교일치가 발생하면 OC1A 핀의 출력은 LOW 값으로 바뀐다.                  |
| 1      | 1      | 비교일치가 발생하면 OC1A 핀의 출력은 HIGH 값으로 바뀐다.                 |

표 4. OC1A 핀의 출력

COM1An 비트 값이 '00<sub>2</sub>'이 아닌 경우에는 OC1A 핀은 범용 입출력 핀으로는 사용할 수 없으며, 파형 출력을 위해서는 DDR 레지스터 설정을 통해 OC1A 핀을 출력으로 설정해 주어야 한다.

파형 생성은 비교 출력 모드(Compare Output Mode) 뿐만이 아니라 파형 생성 모드(Waveform Generation Mode)에도 영향을 받으며 파형 생성 모드를 결정하는 비트가 WGM1n (n = 0, 1, 2, 3)이다. 타이머/카운터 1번의 경우 16가지 파형 생성 모드를 사용할 수 있지만 이 중 12개는 PWM과 관련된 모드이다. PWM이 아닌 4개 모드 중 하나는 사용하지 않으며 디폴트값은 정상 모드(Normal Mode)로 구형파 생성을 위해 사용된다. 나머지 2개 모드는 CTC (Clear Timer on Compare Match) 모드라고 불리며 이름에서 알 수 있듯이 비교일치가 발생하는 경우 지금까지의 카운트 값을 영으로 초기화시킨다. 지금까지 비교일치가 발생한 경우 TCNTn 레지스터의 값을 0으로 설정해 준 것을 기억하는가? CTC 모드를 사용하면 자동으로 카운터가 초기화되므로 보다 간단하게 프로그램을 작성할 수 있다. 표 5는 WGM1n 비트 설정에 따른 파형 생성 모드를 나타낸 것으로 16개의 모드 중 PWM과 관련이 없는 모드들만 나열하였으며 PWM은 별도의 장에서 다룬다.

| 모드 | TCCR1B |       | TCCR1A |       | 설명             |
|----|--------|-------|--------|-------|----------------|
|    | WGM13  | WGM12 | WGM11  | WGM10 |                |
| 0  | 0      | 0     | 0      | 0     | 정상 모드 (디폴트 모드) |
| 4  | 0      | 1     | 0      | 0     | CTC 모드         |
| 12 | 1      | 1     | 0      | 0     | CTC 모드         |
| 13 | 1      | 1     | 0      | 1     | Reserved       |

표 5. PWM이 아닌 파형 생성 모드

표 5에서 알 수 있듯이 WGM1n 비트는 TCCR1A 레지스터와 TCCR1B 레지스터에 2비트씩 나뉘어져 있다. 따라서 CTC 모드를 사용하기 위해서는 TCCR1B 레지스터의 WGM12 비트를 설정해 주어야 한다. 코드 6은 코드 5에서 비교일치 인터럽트 처리 루틴을 제거하고 CTC 모드를 통해 자동으로 카운트 값이 0이 되도록 한 예이다.

코드 6. 파형생성 2

```

#include <avr/io.h>
#include <avr/interrupt.h>

int main(void)
{
    TCCR1B |= (1 << CS12) | (1 << CS10); // 분주비를 1024로 설정

    OCR1A = 0x2000; // 비교일치 기준값

    // 비교일치 인터럽트 발생 시 OC1A 핀의 출력을 반전
    TCCR1A |= (1 << COM1A0);
    TCCR1B |= (1 << WGM12); // CTC 모드 선택

    DDRB |= (1 << PB1); // OC1A 핀(PB1 핀)을 출력으로 설정

    while(1){ }
}

```

타이머/카운터 0번과 2번의 경우에도 CTC 모드를 사용할 수 있다. 하지만 타이머/카운터 0번과 2번은 8비트 타이머/카운터로 WGM 비트가 타이머/카운터 1번과는 약간의 차이가 있으므로 데이터시트를 참고하기 바란다.

## 5. 아두이노의 타이머

아두이노에서는 타이머를 위한 함수를 지원하지 않지만 C 스타일의 코드를 통해 타이머를 사용할 수는 있다. 이외에도 아두이노에 기본적으로 포함되어 있지 않은 확장 라이브러리들이 다수 존재하므로 이를 통해 타이머 기능을 사용할 수 있다. 그 중 MsTimer2 라이브러리는 타이머/카운터 2번을 사용하여 구현한 라이브러리이다.

먼저 MsTimer2 라이브러리를 다운받자.<sup>1)</sup> 다운받은 압축 파일은 아두이노의 스케치북 디렉터리 아래에 libraries 디렉터리에 압축을 해제한다. 스케치북 디렉터리는 ‘도구 → Visual Micro (Arduino 1.0.x) → Configure Ide Locations’에서 확인할 수 있다.

1) [http://www.pjrc.com/teensy/td\\_libs\\_MsTimer2.html](http://www.pjrc.com/teensy/td_libs_MsTimer2.html)

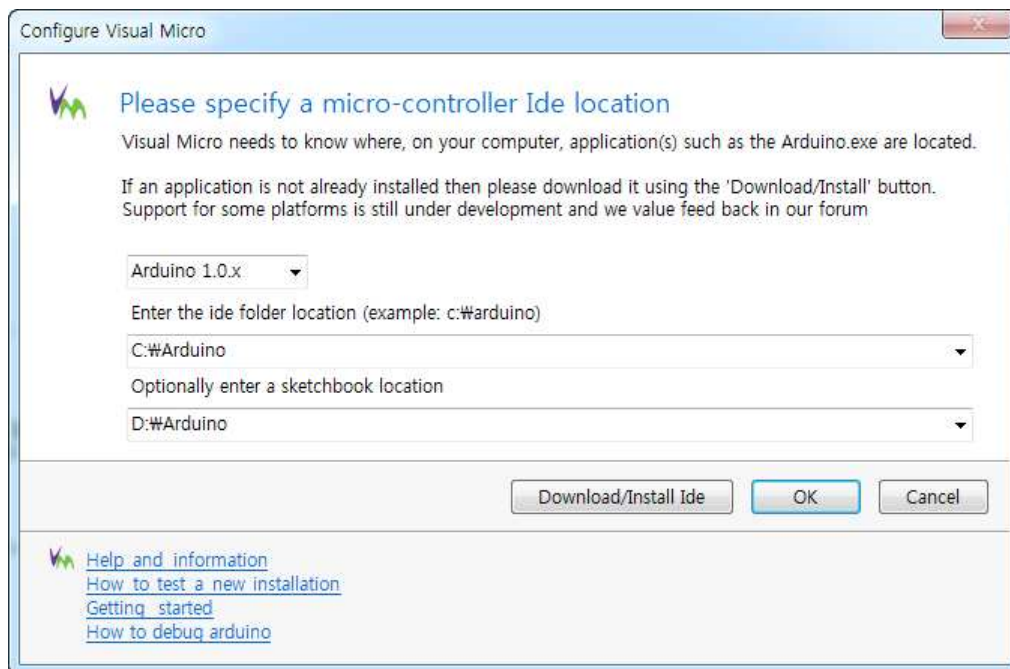


그림 9. 스케치북 디렉터리

라이브러리는 'MsTimer2' 디렉터를 생성하여 그 아래에 압축을 해제하여야 한다.

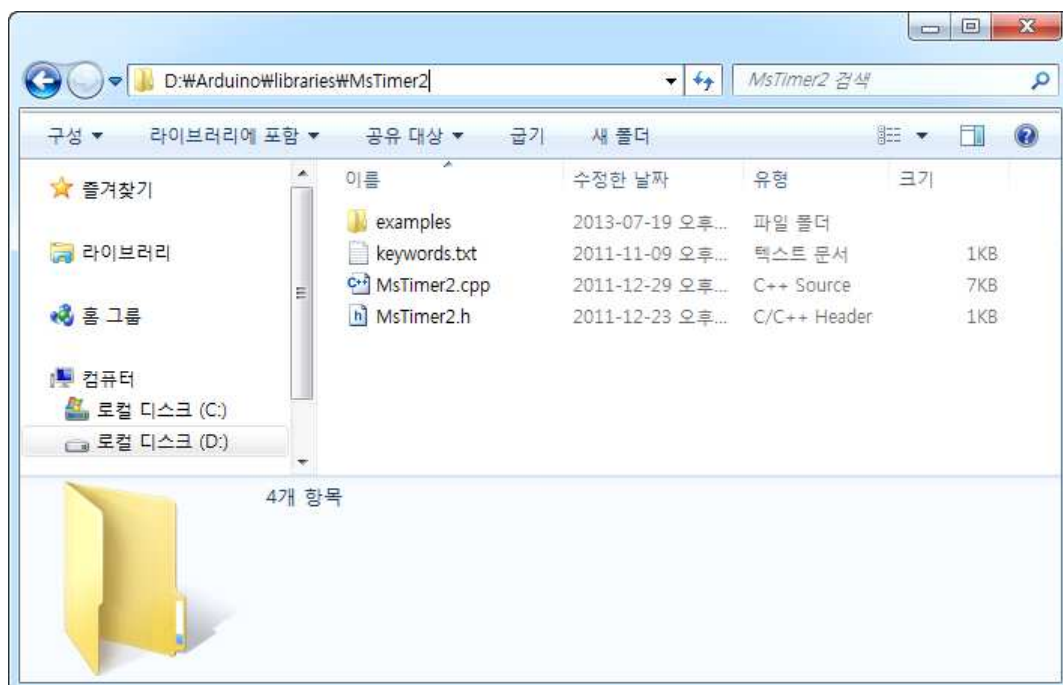


그림 10. MsTimer2 라이브러리 설치 디렉터리

스케치북 디렉터리가 'D:\Arduino'인 경우 압축을 해제할 디렉터리는 'D:\Arduino\libraries\MsTimers'가 된다. 일반적으로 아두이노의 라이브러리는 클래스 형태의 객체지향 스타일로 작성되어 배포되지만 MsTimer2 라이브러리의 경우 MsTimer2 네임스페이스에 타이머 관련 함수들을 정의하고 있으므로 함수 호출을 위해서는 'MsTimer2::function\_name'과 같은 방식으로 사용하여야 한다. MsTimer2 라이브러리에는 다음과 같은 함수들이 정의되어 있다.

- set

```
void set(unsigned long ms, void (*f)())
```

매개변수  
 ms : 경과 시간  
 void (\*f)() : 인터럽트 처리 함수에 대한 포인터  
 반환값 : 없음

지정한 시간 간격으로 타이머를 설정한다. 지정한 시간 간격이 만료되면 인터럽트가 발생되고 인터럽트 서비스 루틴인 함수 f가 자동으로 호출된다. 함수 f는 매개변수를 가질 수 없다.

- start

```
void start(void)
```

매개변수 : 없음  
 반환값 : 없음

인터럽트를 활성화시킨다. 타이머가 만료되면 인터럽트가 발생된다.

- stop

```
void stop(void)
```

매개변수 : 없음  
 반환값 : 없음

인터럽트를 비활성화시킨다. 타이머가 만료되어도 인터럽트가 발생하지 않으며 인터럽트 서비스 루틴은 호출되지 않는다.

아트멜 스튜디오에서 새로운 아두이노 프로젝트를 생성해 보자. 코드 7은 1초 간격으로 만료되는 타이머를 설정하여 사용하는 방법을 보여준다. 설치된 라이브러리를 사용하기 위해서는 해당 헤더 파일을 포함시켜야 하며 '프로젝트 → Add/Import Sketch Library → User' 메뉴를 선택하면 설치한 'MsTimer2' 메뉴 항목이 보일 것이다. 메뉴 항목이 보이지 않는다면 '프로젝트 → Add/Import Sketch Library → Refresh'를 눌러 메뉴 항목을 갱신하면 나타난다. 해당 메뉴 항목을 선택하면 헤더 파일이 자동으로 포함된다.

코드 7. MsTimer2 라이브러리

```
#include <MsTimer2.h>

int count = 0;

void countingFunction(){           // 인터럽트 처리 함수
    Serial.println(count++);
}

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

    // 1초마다 countingFunction 함수가 호출되도록 설정
    MsTimer2::set(1000, countingFunction);
    MsTimer2::start();             // 인터럽트 활성화
}

void loop(){ }
```

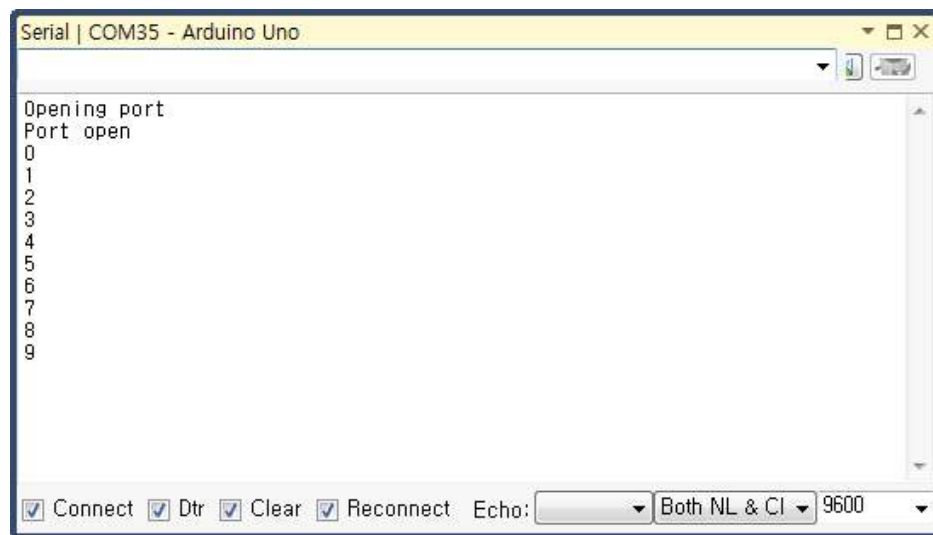


그림 11. 코드 7 실행 결과