

Chapter 117. 7 세그먼트 표시장치

1. 1자리 7 세그먼트 표시장치

7 세그먼트 표시장치(7-segment display)는 7개의 선분으로 숫자나 글자를 표시하기 위해 발광 다이오드(LED)를 사용하여 만든 출력 장치의 일종이다. 7세그먼트 표시장치는 이름과 다르게 7개의 선분에 소수점(DP, Decimal Point)을 표시하기 위한 LED가 추가되어 8개의 세그먼트로 구성되는 것이 일반적이며 바이트 단위의 데이터로 제어할 수 있다. 7세그먼트 표시장치는 FND(Flexible Numeric Display)라고도 흔히 불린다. 7세그먼트 장치는 2자리 이상의 숫자를 표시할 수 있도록 만들어진 경우도 많으며 한 자리 숫자를 표시하는 장치와는 제어하는 방법에 차이가 있다.

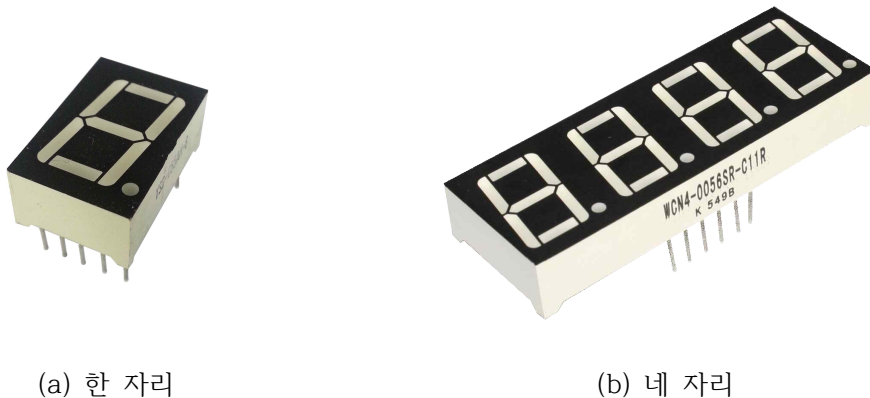


그림 1. 7세그먼트 표시장치

7 세그먼트 표시장치에는 공통 양극 (common cathode) 방식과 공통 음극 (common anode) 방식 두 가지가 있다. 공통 핀에 VCC를 연결하고 각 제어 핀을 GND에 연결하면 불이 들어오는 방식이 공통 양극 방식이며, 공통 핀을 GND에 연결하고 각 제어 핀을 VCC에 연결하면 불이 들어오는 방식이 공통 음극 방식이다.

먼저 한 자리 7세그먼트 표시장치를 살펴보자. 1자리 7 세그먼트 표시장치는 10개의 핀을 가지며 핀 배열은 그림 2와 같다. 그림 2에 나타난 바와 같이 소수점이 있는 면의 가장 왼쪽 핀이 1번 핀에 해당하며 반시계 방향으로 핀 번호는 증가한다.

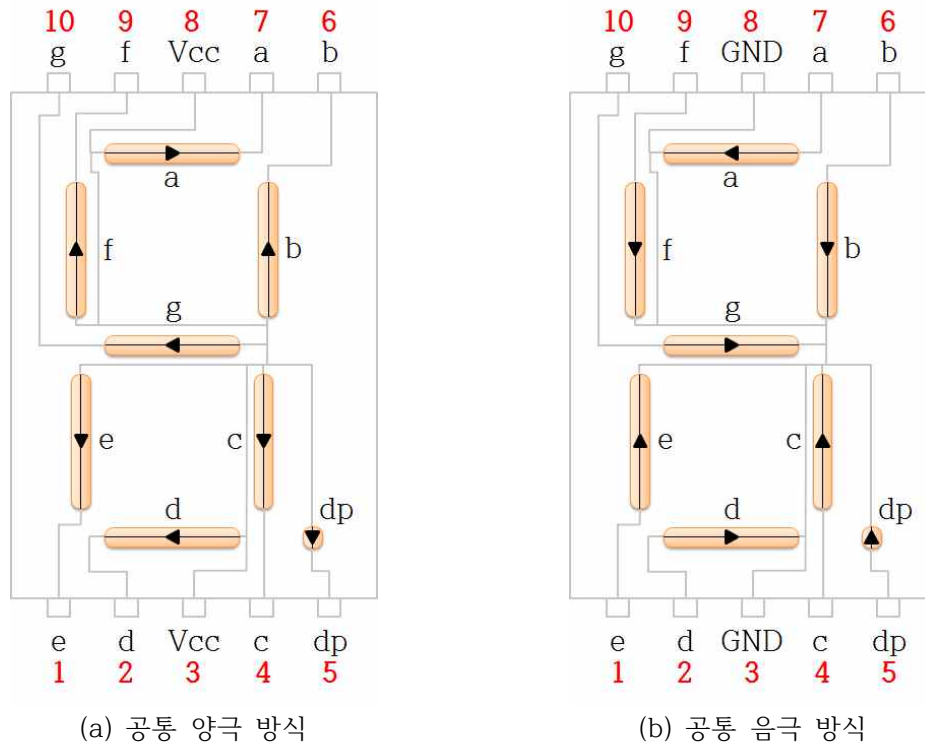


그림 2. 7세그먼트 표시장치 및 핀 배열

1자리 7 세그먼트 표시장치의 10개 핀은 두 개의 공통 핀과 8개의 세그먼트 제어 핀으로 구성된다. 공통 핀은 공통 양극 방식의 경우 VCC에, 공통 음극 방식의 경우 GND에 연결한다. 나머지 8핀은 각 세그먼트를 제어하는 핀으로 공통 양극 방식의 경우 GND 입력이 가해지는 경우 (디지털 출력 LOW) 해당 세그먼트가 켜지는 반면, 공통 음극 방식의 경우 VCC 입력이 가해지는 경우 (디지털 출력 HIGH) 해당 세그먼트가 켜진다. 이처럼 공통 양극 방식과 공통 음극 방식은 제어하는 방식이 서로 반대이므로 7세그먼트 표시장치를 사용하기 이전에 공통 양극 방식인지 공통 음극 방식인지 먼저 확인하여야 하며 이 장에서는 직관적인 사용법과 일치하는 공통 음극 방식의 1자리 7세그먼트를 사용한다. 7 세그먼트 표시장치로 숫자를 표시한 예는 그림 3과 같다.

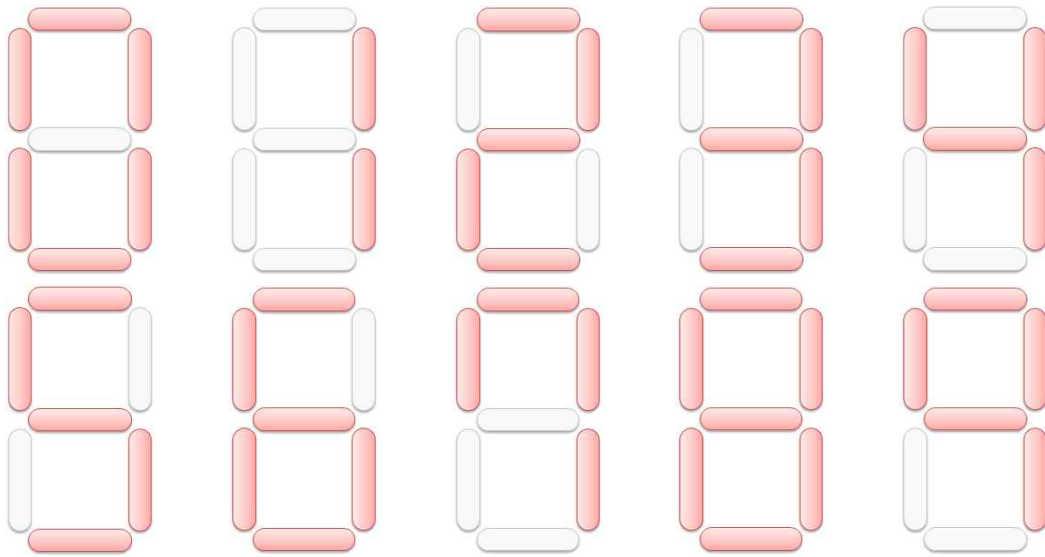


그림 3. 7 세그먼트 표시장치에서 숫자의 표현

그림 3과 같은 모양을 표시하기 위해 공통 음극 방식의 7세그먼트 표시장치 제어 핀으로 출력하여야 하는 데이터는 표 1과 같다.

숫자	세그먼트								제어값	
	a	b	c	d	e	f	g	dp	2진값	16진값
0	○	○	○	○	○	○	×	×	1111 1100	FC
1	×	○	○	×	×	×	×	×	0110 0000	60
2	○	○	×	○	○	×	○	×	1101 1010	DA
3	○	○	○	○	×	×	○	×	1111 0010	F2
4	×	○	○	×	×	○	○	×	0110 0110	66
5	○	×	○	○	×	○	○	×	1011 0110	B6
6	○	×	○	○	○	○	○	×	1011 1110	BE
7	○	○	○	×	×	○	×	×	1110 0100	E4
8	○	○	○	○	○	○	○	×	1111 1110	FE
9	○	○	○	×	×	○	○	×	1110 0110	E6

표 1. 1자리 7세그먼트에 숫자 표시를 위한 데이터 (○ : on, × : off)

1자리 7세그먼트 표시장치를 그림 4와 같이 연결해 보자. 각 세그먼트에는 세그먼트 보호를 위해 220Ω의 저항을 사용하였다. 회로를 구성함에 있어 7세그먼트 표시장치의 각 세그먼트들

은 'a → b → c → d → e → f → g → dp' 순서로 언급하는 반면 이 순서가 7세그먼트 표시장치의 핀 번호와는 서로 다르다는 점에 유의하여야 한다.

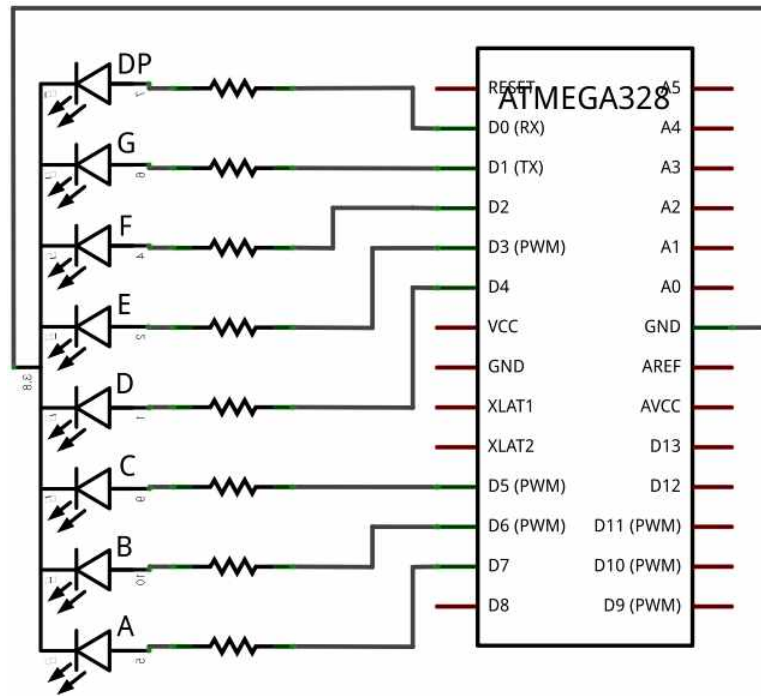


그림 4. 1자리 7세그먼트 표시장치 연결 회로

코드 1은 1자리 7세그먼트 장치에 1초 간격으로 0에서 9까지 반복적으로 표시하는 코드의 예이다. 제어 핀은 포트 D를 사용하므로 PORTD 레지스터를 통해 간단히 제어할 수 있다. 코드 1에서 다음 번 표시할 숫자를 결정함에 있어 '9'가 출력된 다음에는 '0'을 출력하도록 하기 위해 나머지 연산자(%)를 사용하고 있다.

코드 1. 상향 카운터

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    uint8_t numbers[]
```

```

        = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE4, 0xFE, 0xE6};
int count = 0;                                // 현재 표시할 숫자
DDRD = 0xFF;                                  // 제어핀 8개를 출력으로 설정

while(1)
{
    PORTD = numbers[count];                    // 숫자 데이터 출력
    count = (count + 1) % 10;                  // 다음에 표시할 숫자

    _delay_ms(1000);                           // 1초 대기
}

return 1;
}

```

코드 1은 `_delay_ms` 함수를 사용하고 있다. `_delay_ms` 함수를 사용하면 간단히 시간 지연을 구현할 수 있지만 `_delay_ms` 함수가 실행 중인 동안에는 다른 작업을 진행할 수 없는 문제점이 있다. 이를 개선하기 위한 방법 중 한 가지는 타이머를 통한 인터럽트를 이용하는 방법이다. 코드 2는 8비트 타이머/카운터를 사용하여 1초 간격으로 숫자가 바뀌도록 하는 코드의 예이다. 타이머/카운터 1번은 8비트이므로 256 클록에 한 번 오버플로 인터럽트를 발생시키므로 분주비를 1024로 설정하면 1초 동안 인터럽트가 64번 발생한다.

$$\text{경과 시간} = \frac{\text{클록 주파수}}{\text{분주비} \cdot \text{카운터 최댓값} \cdot \text{인터럽트 발생 횟수}} = \frac{16 \times 2^{20}}{2^{10} \times 256 \times 64} = 1 \text{ 초}$$

코드 2. 상향 카운터 - 인터럽트 사용

```

#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/interrupt.h>

volatile int interrupt_count = 0;           // 인터럽트 발생 횟수

```

```

ISR(TIMER0_OVF_vect)                // 타이머/카운터 0번 오버플로 인터럽트
{
    interrupt_count++;
}

int main(void)
{
    uint8_t numbers[]
        = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE4, 0xFE, 0xE6};
    int count = 0;
    DDRD = 0xFF;
    PORTD = numbers[0];                // 0에서 시작

    // 타이머/카운터 0번 인터럽트 설정
    TCCR0B |= (1 << CS02) | (1 << CS00); // 분주비를 1024로 설정
    TIMSK0 |= (1 << TOIE0);              // 오버플로 인터럽트 허용
    sei();                                // 전역적으로 인터럽트 허용

    while(1)
    {
        if(interrupt_count == 64){        // 1초 경과
            interrupt_count = 0;          // 인터럽트 발생 횟수 초기화

            count = (count + 1) % 10;      // 표시할 숫자
            PORTD = numbers[count];        // 숫자 데이터 출력
        }
    }

    return 1;
}

```

아두이노에서 인터럽트를 사용하는 예는 흔하지 않다. 대신 폴링 방식으로 프로그램이 시작된 이후 현재까지의 시간을 millis 함수를 통해 알아내고 원하는 경과 시간과 비교하여 주기적인 작업이 필요한 경우 사용한다. millis 함수는 내부적으로는 타이머/카운터 0번을 사용하고 있으며, 분주비 64의 오버플로 인터럽트를 사용한다. 분주비 64의 오버플로가 발생하기 위해서

는 64×256 개의 클록을 필요로 한다. 아두이노 우노는 16M의 클록을 사용하므로 1 마이크로초에 16개의 클록이 발생한다. 따라서 오버플로가 발생할 때까지의 시간은 $64 \times 256 \div 16 = 1024$ 마이크로초가 된다. 이를 밀리초 단위의 1과 마이크로초 단위의 24로 분리하여 각각 별도의 변수의 저장하고 오버플로 인터럽트가 발생할 때마다 값을 갱신해 준다. 코드 3은 아두이노의 millis 함수를 구현하여 상향 카운터를 만든 예이다. millis 함수는 이외에도 실행 시간 측정이나 시간 비교 등을 위해 다양한 용도로 사용할 수 있다.

코드 3. 상향 카운터 - millis 함수 사용

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <avr/interrupt.h>

// 1 마이크로초 당 클록의 수
#define CLOCKS_PER_MICRO ( F_CPU / 1000000L )
// 클록 수를 마이크로초로 변환
#define CLOCKS_TO_MICROSECONDS(a) ( (a) / CLOCKS_PER_MICRO )
// 타이머0의 분주비를 64로 설정한 경우
// 오버플로 인터럽트가 발생하기까지의 마이크로초 단위 경과 시간
#define MICROSECONDS_PER_TIMER0_OVERFLOW
    ( CLOCKS_TO_MICROSECONDS(64 * 256) )
// 오버플로 인터럽트가 발생한 경우 밀리초 단위 증가 시간
#define MILLIS_INCREMENT_PER_OVERFLOW
    ( MICROSECONDS_PER_TIMER0_OVERFLOW / 1000 )
// 오버플로 인터럽트가 발생한 경우
// 밀리초 단위 증가를 제외하고 남는 마이크로초 단위 증가 시간
#define MICROS_INCREMENT_PER_OVERFLOW
    ( MICROSECONDS_PER_TIMER0_OVERFLOW % 1000 )

// 프로그램 시작 이후의 경과 시간
volatile unsigned long timer0_millis = 0;
volatile int timer0_micros = 0;

ISR(TIMER0_OVF_vect)
{
```

```

unsigned long m = timer0_millis;
int f = timer0_micros;

m += MILLIS_INCREMENT_PER_OVERFLOW;    // 밀리초 단위 시간 증가
f += MICROS_INCREMENT_PER_OVERFLOW;    // 마이크로초 단위 시간 증가

// 마이크로초가 1000을 넘어가면 밀리초를 증가시킴
int micro_to_millis = f / 1000;
m += micro_to_millis;
f = f % 1000;

timer0_millis = m;
timer0_micros = f;
}

unsigned long millis()
{
    unsigned long m;
    uint8_t oldSREG = SREG;            // 상태 레지스터 값 저장

    // timer0_millis 값을 읽는 동안
    // timer0_millis 값이 변하지 않도록 인터럽트를 비활성화
    cli();

    m = timer0_millis;

    SREG = oldSREG;                    // 이전 상태 레지스터 값 복원

    return m;                          // 프로그램 시작 후 경과 시간
}

int main(void)
{
    uint8_t numbers[]
        = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE4, 0xFE, 0xE6};
    int count = 0;

```

```

DDRD = 0xFF;
PORTD = numbers[0];

TCCR0B |= (1 << CS01) | (1 << CS00); // 분주비를 64로 설정
TIMSK0 |= (1 << TOIE0);              // 오버플로 인터럽트 허용
sei();                                // 전역적으로 인터럽트 허용

unsigned long time_previous, time_current;
time_previous = millis();              // 시작 시간

while(1)
{
    time_current = millis();            // 현재 시간

    if((time_current - time_previous) > 1000){ // 1초 경과
        time_previous = time_current;

        count = (count + 1) % 10;
        PORTD = numbers[count];
    }
}

return 1;
}

```

2. 1자리 7 세그먼트 표시장치 - 아두이노

상향 카운터를 만드는 방법 3가지를 위에서 살펴보았다. 이 중 인터럽트를 사용하는 방법을 제외한 두 가지는 아두이노에서 흔히 사용된다. 인터럽트를 사용할 수도 있지만 아두이노는 인터럽트 지원이 원활하지 않으므로 C 스타일의 코드를 사용하여야 한다. 코드 4는 delay 함수를 사용하여 상향 카운터를 구현한 예로 코드 1과 동일한 동작을 수행한다.

코드 4. 상향 카운터 - 아두이노 delay 함수 사용

```
// 세그먼트 연결 핀 'dp → g → f → e → d → c → b → a' 순서
int segment_pins[] = {0, 1, 2, 3, 4, 5, 6, 7};
uint8_t numbers[]
    = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE4, 0xFE, 0xE6};
int count = 0;

void setup() {
    for(int i = 0; i < 8; i++)                // 세그먼트 제어핀을 출력으로 설정
        pinMode(segment_pins[i], OUTPUT);
}

void loop() {
    for(int i = 0; i < 8; i++){
        boolean on_off = bitRead(numbers[count], i); // 비트값 읽기
        digitalWrite(segment_pins[i], on_off);       // 세그먼트 제어
    }

    delay(1000);                                // 1초 대기

    count = (count + 1) % 10;                    // 표시할 숫자
}
```

코드 4에서 bitRead 함수는 주어진 값에서 특정 위치의 비트값을 반환하는 매크로 함수로 LSB가 0번부터 시작된다.

```
#define bitRead(value, bit) (((value) >> (bit)) & 0x01)
```

또 한 가지 다른 점은 C 스타일 코드의 경우에는 포트 단위로 8비트의 데이터를 한꺼번에 출력할 수 있지만 아두이노의 경우에는 핀 별로 비트 단위의 출력을 반복해야 한다는 점이다. 코드 4 역시 delay 함수를 사용하였으며 앞에서 설명한 이유로 delay 함수 사용은 바람직하지 않다. 따라서 아두이노에서도 millis 함수를 사용이 추천된다. 코드 5는 아두이노의 millis 함수를 사용하여 코드 4와 동일한 동작을 하는 코드의 예로 코드 3과 동일하다. 코드 3에서의

millis 함수는 아두이노의 millis 함수 구현을 간단히 구현한 것으로 실제 아두이노의 millis 함수와 기본적인 동작은 동일하지만 세부적인 동작은 다를 수 있다.

코드 5. 상향 카운터 - 아두이노 millis 함수 사용

```
// 세그먼트 연결 핀 'dp → g → f → e → d → c → b → a' 순서
int segment_pins[] = {0, 1, 2, 3, 4, 5, 6, 7};
uint8_t numbers[]
    = {0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE4, 0xFE, 0xE6};
unsigned long time_previous, time_current;
int count = 0;

void setup() {
    for(int i = 0; i < 8; i++)                // 세그먼트 제어핀을 출력으로 설정
        pinMode(segment_pins[i], OUTPUT);

    time_previous = millis();                  // 시작 시간
}

void loop() {
    time_current = millis();                  // 현재 시간

    if(time_current - time_previous > 1000){
        time_previous = time_current;
        for(int i = 0; i < 8; i++){
            boolean on_off = bitRead(numbers[count], i);    // 비트값 읽기
            digitalWrite(segment_pins[i], on_off);          // 세그먼트 제어
        }
        count = (count + 1) % 10;              // 표시할 숫자
    }
}
```