

Chapter 105. 비트 연산자

1. 포트? 핀?

이전 장에서는 DDRx(아두이노 스타일의 경우 pinMode 함수)와 PORTx(아두이노 스타일의 경우 digitalWrite 함수) 레지스터를 통해 LED를 제어하는 간단한 프로그램을 통해 C 스타일과 아두이노 스타일 프로그램을 비교해 보았다. 이를 확장시켜 이 장에서는 크리스마스 트리까지는 아니더라도 여러 개의 LED를 제어해 보자. 여러 개의 LED를 제어하기 위해서는 제어하고 싶은 LED의 개수만큼 ATmega328의 범용 입출력(GPIO) 핀이 필요하다. C 스타일의 프로그램에서는 (8개 핀을 묶은 바이트 단위인) 포트 단위로 GPIO 핀을 다루는 반면, 아두이노 스타일 프로그램에서는 (비트 단위인) 핀 단위로 GPIO 핀을 다룬다는 차이를 기억하는가? 여러 개의 LED를 제어하기 위해서는 포트 단위로 LED를 연결하는 것이 유리하므로 먼저 포트와 아두이노 핀 번호 사이의 관계를 살펴보자.

ATmega328의 28개 핀 중 5개는 전원 관련 핀으로 나머지 23개가 범용 입출력으로 사용할 수 있는 핀이다. 하지만 이 중 2개는 외부 크리스탈 연결을 위해, 1개는 리셋 용도로 사용하므로 아두이노에서는 20개의 핀을 범용 입출력으로 사용한다. ATmega328에는 B, C, D의 3개 포트가 존재하지만 포트 D만 8개 핀을 전부 범용 입출력으로 사용할 수 있고 포트 B와 C는 비트 0에서 비트 5까지 6개 핀만을 범용 입출력으로 사용할 수 있다. 아두이노에서는 'D → B → C' 포트의 순서로 0번에서 19번까지의 번호를 할당하여 사용하고 있다. 이 중 포트 C의 범용 입출력 핀에는 아날로그-디지털 변환 장치(Analog-Digital Converter, ADC)가 연결되어 있어 아날로그 입력을 받아들이는 용도로 사용할 수 있으므로 아두이노에서는 범용 입출력 핀 14번부터 19번까지를 아날로그 입력 핀이라고 부른다. 반면 범용 입출력 핀 0번부터 13번까지를 디지털 입출력 핀이라고 부른다. 표 1은 포트와 아두이노 핀 번호 사이의 관계를 요약한 것이다.

포트 및 포트 내 비트 번호	아두이노 핀 번호	ATmega328 칩에서의 핀 번호	비고
PB0	8	14	
PB1	9	15	
PB2	10	16	
PB3	11	17	
PB4	12	18	
PB5	13	19	
PB6	XTAL	9	외부 크리스탈 연결 핀
PB7	XTAL	10	
PC0	14	23	아날로그 입력 핀
PC1	15	24	
PC2	16	25	
PC3	17	26	
PC4	18	27	
PC5	19	28	
PC6	RESET	1	리셋 핀
PD0	0	2	시리얼 방식 업로드에 사용
PD1	1	3	
PD2	2	4	
PD3	3	5	
PD4	4	6	
PD5	5	11	
PD6	6	12	
PD7	7	13	

표 1. ATmega328의 포트와 아두이노 핀 번호 사이의 관계

표 1에서는 ATmega328 칩의 핀 번호 역시 나타내었지만 이 책에서 칩의 핀 번호를 사용하는 경우는 흔하지 않으며 C 스타일의 프로그램에서는 “포트와 비트 번호”로, 아두이노 스타일의 프로그램에서는 “핀 번호”로 범용 입출력 핀을 구별한다. 특히 ‘1번 핀’ 또는 ‘디지털 1번 핀’이라고 지칭하는 핀은 아두이노에서 포트 D의 1번 비트에 해당하는 ATmega328 칩의 3번 핀을 지칭한다는 점에 주의하여야 한다.

한 가지 유념해야 할 점은 아두이노의 디지털 0번과 1번 핀은 일반적으로 범용 입출력으로 사용하지 않는다는 점이다. 아두이노에서는 일반적으로 시리얼 방식의 프로그램 업로드를 사용하며 이는 UART 통신을 통해 이루어진다. ATmega328에는 UART 통신을 위한 전용 핀으로 디지털 0번과 1번 핀을 사용하므로 여기에 다른 외부 장치를 연결하는 경우 시리얼 업로드 장치와 충돌로 정상적으로 동작하지 않는 경우가 있으므로 0번과 1번 범용 입출력 핀은 프로그램 업로드 전용으로 사용하는 것이 일반적이다.

2. 8개 LED 제어하기

여러 개의 LED를 제어하기 위해 포트 D(0번에서 7번까지의 디지털 핀)에 8개의 LED를 연결해 보자.

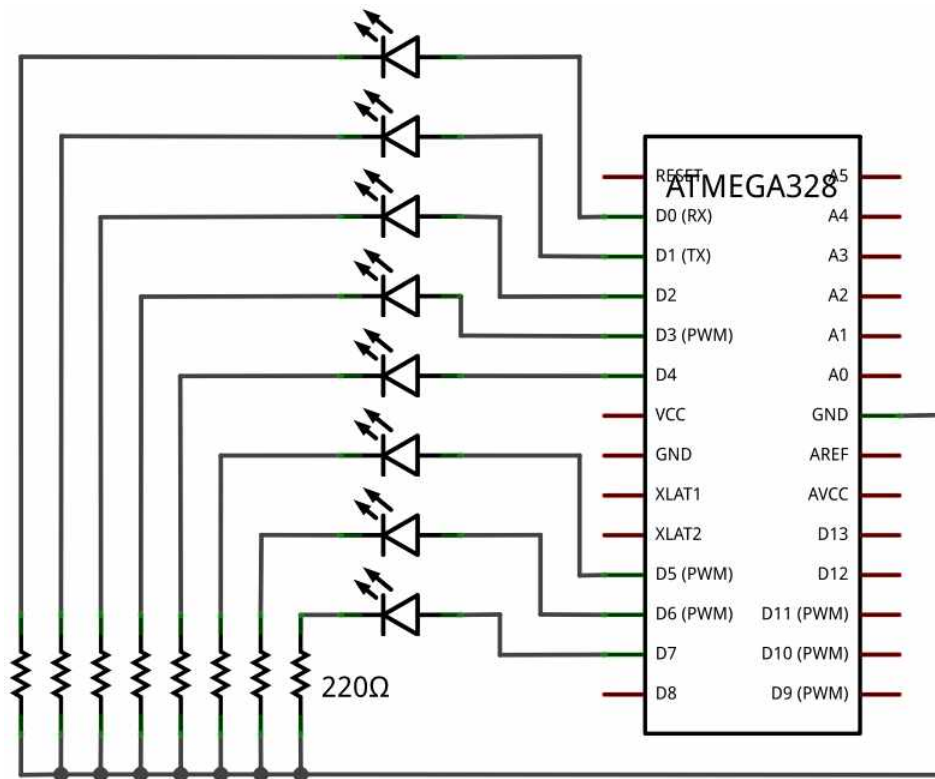


그림 1. 포트 D에 8개 LED 연결을 위한 회로도

코드 1은 1초 간격으로 홀수 번 째 LED와 짝수 번 째 LED를 번갈아 켜는 C 스타일 코드이며 코드 2는 동일한 동작을 하는 아두이노 스타일 코드이다.

코드 1. 포트 D의 8개 LED 점멸 I - C 스타일

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

int main(void)
{
    DDRD = 0xFF;          // PORT D 8개 핀을 출력으로 설정

    while(1)
    {
        PORTD = 0x55;     // 홀수 번 째 LED 켜기
        _delay_ms(1000);
        PORTD = 0xAA;     // 짝수 번 째 LED 켜기
        _delay_ms(1000);
    }

    return 1;
}
```

코드 2. 포트 D의 8개 LED 점멸 I - 아두이노 스타일

```
int pins[] = {0, 1, 2, 3, 4, 5, 6, 7};    // 포트 D의 아두이노 핀 번호

void setup()
{
    for(int i = 0; i < 8; i++){            // 포트 D의 핀들을 출력으로 설정
        pinMode(pins[i], OUTPUT);
    }
}

void loop()
{
    for(int i = 0; i < 8; i++){            // 홀수 번 째 LED 켜기
```

```

    if(i % 2 == 0) digitalWrite(pins[i], HIGH);
    else digitalWrite(pins[i], LOW);
}
delay(1000);

for(int i = 0; i < 8; i++){                // 짝수 번 째 LED 켜기
    if(i % 2 == 0) digitalWrite(pins[i], LOW);
    else digitalWrite(pins[i], HIGH);
}
delay(1000);
}

```

코드 1과 2는 두 가지의 표시 패턴만을 가지고 있으므로 간단하게 코드를 작성할 수 있다. 하지만 그림 2와 같이 8가지의 표시 패턴을 주기적으로 표현하려고 한다면 어떻게 해야 할까?

비트 번호	7	6	5	4	3	2	1	0
패턴 1	■							
패턴 2		■						
패턴 3			■					
패턴 4				■				
패턴 5					■			
패턴 6						■		
패턴 7							■	
패턴 8								■

그림 2. 8가지 LED 표시 패턴 (■ : LED ON, □ : LED OFF)

코드 3과 같이 패턴을 반복하거나 코드 4와 같이 배열을 사용하여 구현할 수도 있다.

코드 3. 포트 D의 8개 LED 점멸 II-1

```
#define F_CPU 16000000L
```

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

int main(void)
{
    DDRD = 0xFF;          // PORT D 8개 핀을 출력으로 설정

    while(1)
    {
        PORTD = 0x80;      // 1번 패턴
        _delay_ms(1000);
        PORTD = 0x40;      // 2번 패턴
        _delay_ms(1000);
        PORTD = 0x20;      // 3번 패턴
        _delay_ms(1000);
        PORTD = 0x10;      // 4번 패턴
        _delay_ms(1000);
        PORTD = 0x08;      // 5번 패턴
        _delay_ms(1000);
        PORTD = 0x04;      // 6번 패턴
        _delay_ms(1000);
        PORTD = 0x02;      // 7번 패턴
        _delay_ms(1000);
        PORTD = 0x01;      // 8번 패턴
        _delay_ms(1000);
    }

    return 1;
}
```

코드 4. 포트 D의 8개 LED 점멸 II-2

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

```

int main(void)
{
    DDRD = 0xFF;          // PORT D 8개 핀을 출력으로 설정

    char patterns[] = {0x80, 0x40, 0x20, 0x10, 0x08, 0x04, 0x02, 0x01};
    while(1)
    {
        for(int i = 0; i < 8; i++){
            PORTD = patterns[i];      // i번 패턴
            _delay_ms(1000);
        }
    }

    return 1;
}

```

그림 2의 8개 패턴은 그 형태가 단순하여 간단한 계산으로 만들어 내는 것도 가능하며 이를 위해서는 비트 연산이 필요하다. 우선 코드 5의 입력해서 실행해 보자.

코드 5. 포트 D의 8개 LED 점멸 II-3

```

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

int main(void)
{
    DDRD = 0xFF;          // PORT D 8개 핀을 출력으로 설정

    while(1)
    {
        for(int i = 0; i < 8; i++){
            PORTD = 0x01 << (7 - i); // i번 패턴

```

```

        _delay_ms(1000);
    }
}

return 1;
}

```

코드 5는 코드 4와 비교했을 때 패턴을 계산을 통해 만들어 내는 점 이외에는 동일하다. 코드 5를 이해하기 위해서는 먼저 비트 연산자를 이해하여야 한다.

3. 비트 연산자

디지털 데이터를 다루는 마이크로컨트롤러에서 최소의 데이터 단위는 비트(bit)로 하나의 입출력 핀은 한 비트의 데이터로 제어할 수 있다. 하지만 마이크로컨트롤러를 포함하여 중앙처리장치에서 비트 단위로 데이터를 처리할 수 있는 방법은 없으므로 주변장치와의 데이터 입출력을 위해서 바이트 단위인 포트를 사용한다. 포트 B의 여섯 번째 핀으로 1의 값을 출력하기 위해 사용한 문장이 기억나는가?

```
PORTB = 0x20;
```

실제로는 8개의 비트, 즉, 바이트 단위의 데이터가 할당되고 있지만 그 중 의미가 있는 값은 여섯 번째 비트의 값뿐이며 나머지 7개 비트는 비트 단위로 데이터를 입출력할 수 없기 때문에 사용되었을 뿐 의미는 없다. 아두이노의 경우는 어떤가?

```
digitalWrite(13, HIGH);
```

digitalWrite 함수는 핀 번호와 HIGH 또는 LOW의 출력값을 매개변수로 가지므로 핀 단위 또는 비트 단위로 데이터를 출력하는 것처럼 보인다. 하지만 이는 말 그대로 ‘처럼’ 보일 뿐 실제로는 포트 B에 8비트의 데이터를 출력하고 있다. digitalWrite 함수의 정의를 살펴보면 이

를 보다 명확히 알 수 있다. digitalWrite 함수는 단순히 데이터를 출력하는 이상의 작업을 하고 있으므로 데이터 출력 부분만을 살펴보자.¹⁾

```
void digitalWrite(uint8_t pin, uint8_t val)
{
    ...
    uint8_t bit = digitalPinToBitMask(pin); // 비트 마스크
    volatile uint8_t *out;                  // PORTx 레지스터로 출력할 값

    ...
    out = portOutputRegister(port);        // 현재 PORTx 레지스터의 값

    ...
    // 비트 단위 데이터 조작
    if (val == LOW){
        *out &= ~bit;                      // PORTx 레지스터의 해당 비트를 0으로 설정
    } else {
        *out |= bit;                      // PORTx 레지스터의 해당 비트를 1로 설정
    }
    ...
}
```

무언가 복잡해 보이지 않는가? 아두이노에서 지정한 핀 번호는 해당 레지스터에서의 비트 번호로 매핑 되고, 비트 단위의 데이터 조작을 통해 값이 설정된 후, 바이트 단위로 데이터가 출력된다. 아두이노 스타일 프로그램에 마술은 없다. C 스타일 코드를 보다 이해하고 사용하기 쉽게 만들어 놓았을 뿐 핵심은 C 스타일 코드와 동일하다. 위의 코드를 전부 이해하지 못하더라도 실망할 필요는 전혀 없지만 비트 단위의 데이터 조작은 반드시 이해해야 한다.

1) digitalWrite 함수의 정의는 아두이노 설치 디렉터리 아래 'hardware\arduino\cores\arduino' 디렉터리에 'wiring_digital.c' 파일에서 확인할 수 있다.

C 언어에서 정의된 기본 자료형 중에서 가장 적은 메모리를 필요로 하는 자료형은 한 글자를 나타내는 char형으로 1바이트의 메모리를 필요로 한다. 마이크로컨트롤러의 경우 비트 단위의 데이터 입출력이 가능하다면 보다 효율적으로 프로그램을 작성할 수 있겠지만 여러 번 언급하였다시피 이는 불가능하다.

아두이노에는 비트 단위의 데이터를 표현하기 위해 boolean형이 존재하며 'true'나 'false' 값을 가질 수 있다. 어떻게 이것이 가능할까? 'Arduino.h' 파일을 살펴보면 프로그래밍의 편의를 위해 실제로는 1바이트의 메모리를 필요로 하지만 1비트의 메모리를 사용할 것처럼 보이는 boolean형의 정의를 찾아볼 수 있다.

```
typedef uint8_t boolean;
```

실제로는 1바이트의 메모리를 사용하고 있다. 또한 true와 false 역시 일(one)과 영(zero)의 정수로 정의되어 있음을 확인할 수 있다.

```
#define true 0x1
#define false 0x0
```

이들은 아두이노에서만 사용 가능하며 C 스타일의 프로그램에서는 사용할 수 없다는 점도 명심해야 한다.

마이크로프로세서에서 모든 자료는 이진값으로 처리되며 처리의 최소 단위는 비트이다. C 언어에서는 비트 값을 처리할 수 있는 비트 연산자들을 통해 비트 단위의 연산을 수행할 수 있도록 해주지만 실제로 연산은 최소한 8개의 비트를 묶은 바이트 단위로 이루어진다는 점을 잊지 말아야 한다. 표 2는 C 언어에서 제공하는 비트 연산자를 나타낸다. 비트 AND 및 비트 OR 연산자는 논리 AND 및 논리 OR 연산자와 유사하므로 혼돈하지 않도록 한다.

구분	연산자	종류	결과
비트 논리 연산자	$a \& b$	비트 AND	a와 b의 비트 단위 AND
	$a b$	비트 OR	a와 b의 비트 단위 OR
	$a \wedge b$	비트 XOR	a와 b의 비트 단위 XOR
	$\sim a$	비트 NOT	a의 비트 단위 NOT (1의 보수)
비트 이동 연산자	$a \ll n$	왼쪽으로 이동	a를 n비트 왼쪽으로 이동하고 오른쪽은 0으로 채움
	$a \gg n$	오른쪽으로 이동	a를 n비트 오른쪽으로 이동하고 왼쪽은 0으로 채움

표 2. C 언어의 비트 연산자

비트 연산자는 불대수에서 정의된 논리 연산을 비트 단위로 시행하는 연산자와 비트를 좌우로 이동시키는 연산자로 이루어져 있다. 비트 단위 논리 연산은 참 또는 거짓을 나타내는 1 또는 0의 값에 대하여 표 3의 진리표에 정의된 방식으로 연산을 수행한다.

a1	b1	$a1 \& b1$	$a1 b1$	$a1 \wedge b1$	$\sim a1$
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

표 3. 비트 논리 연산자를 위한 진리표

비트 이동 연산은 지정한 비트 수 만큼을 좌 또는 우로 이동시키는 연산으로, 이동으로 인해 밀려 나오는 비트들은 버려지고 빈 칸은 0으로 채워진다. 그림 3은 왼쪽으로 한 비트 이동시키는 연산의 예를 보여준다.

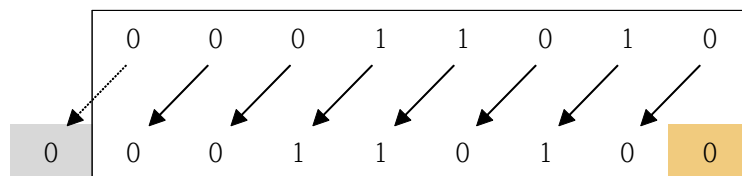


그림 3. 왼쪽 비트 이동 연산

왼쪽 이동 연산에 의해 이동 전 값 $00011010_2 = 26$ 은 이동 후에 $00110100_2 = 52$ 로 2배로 바뀐 점에 주목하여야 한다. 하지만 실수값에 대해서는 비트 이동 연산을 사용할 수 없으며 오버플로가 발생하는 경우에도 값이 2배가 되지 않는다. 그림 3과 유사하게 그림 4는 오른쪽으로 한 비트 이동시키는 연산의 예를 보여준다.

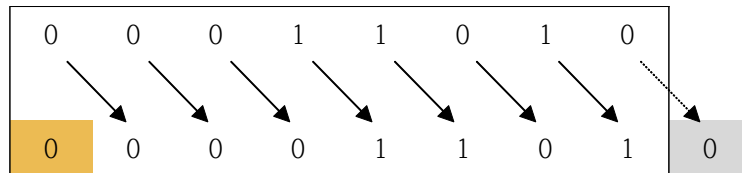


그림 4. 오른쪽 비트 이동 연산

왼쪽 이동 연산과는 반대로 이동 전 값 $00011010_2 = 26$ 은 이동 후에 $00001101_2 = 13$ 으로 $1/2$ 로 바뀌었다. 하지만 왼쪽 이동 연산과 마찬가지로 실수값에서는 사용할 수 없으며 정수 나눗셈만 가능하다.

마이크로컨트롤러 프로그래밍 과정에서는 레지스터를 조작하는 작업이 많으며 레지스터는 비트별로 그 의미가 다르게 지정되어 있는 경우가 많아 비트 단위 연산을 통해 원하는 값을 얻어내거나 바꾸는 작업을 흔히 볼 수 있다. 아두이노의 경우 레지스터 조작의 많은 부분을 추상화된 함수로 제공하고 있지만 비트 연산을 이해함으로써 코드의 이해도를 높일 수 있을 뿐만 아니라 함수로 제공되지 않는 작업을 수행하기 위해서는 비트 연산을 사용하여 하는 경우가 있다. 일반적으로 특정 비트를 1로 만드는 작업을 '세트(set)', 특정 비트를 0으로 만드는 작업을 '클리어(clear)'라고 표현하며 마이크로컨트롤러 프로그래밍에서 많이 사용되는 비트 연산에는 다음과 같은 것들이 있다.

- 특정 비트 세트
- 특정 비트 클리어
- 특정 비트 반전
- 특정 비트 검사/읽기

① 특정 비트 세트

특정 비트 세트는 특정 위치의 비트만을 1로 설정하고 나머지는 현재 값을 그대로 유지하는 것을 말한다. 비트 OR 연산을 수행하면 그림 5에서 알 수 있듯이 1과 OR 시킨 결과는 항상 1이 되고 0과 OR 시킨 결과는 현재 값이 그대로 유지되며, 이를 통해 원하는 위치의 비트만

을 1로 만들 수 있다.

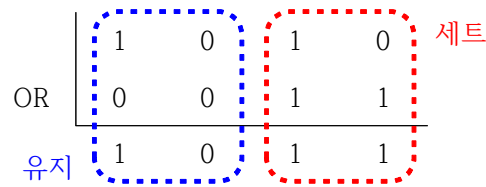


그림 5. 비트 OR 연산

변수 a에 $a = 0bxxxxxxx$ 값이 저장되어 있는 경우 3번째 비트만은 1로 설정하고 나머지 비트들은 현재 값을 그대로 유지하기 위해서는 그림 6에서와 같이 $0b00000100$ 과 OR 연산을 수행하면 된다.

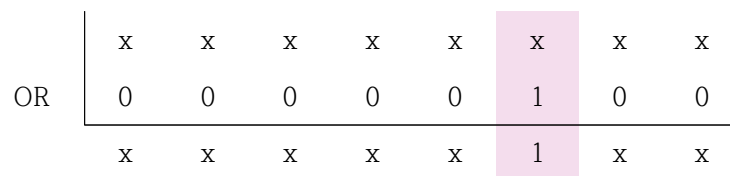


그림 6. 특정 비트 세트

그림 6을 코드로 나타내면 다음과 같다.

```
a = a | 0b00000100;
a = a | 0x04;
```

위의 코드에서 $0b00000100$ 또는 $0x04$ 를 비트 연산을 위한 '마스크(mask)'라고 한다. 마스크는 직접 숫자로 기입하기도 하지만 숫자를 직접 기입하는 것은 종종 오류의 원인이 되므로 비트 이동 연산을 통해 마스크를 만들어 사용하기도 하며, 마스크 생성을 위해서는 왼쪽 비트 이동 연산자인 '<<'를 사용할 수 있다. 비트 이동 연산자를 사용하면 그림 6의 비트 세트 연산은 다음과 같이 표시할 수 있다.

```
a = a | (0x01 << 2);
```

위의 코드에서 (1 << 2) 부분이 0x04 마스크를 만들어내는 부분으로 3번째 비트만을 1로 설정하기 위해 2만큼 왼쪽으로 이동시킨다. 이 때 세 번째 비트가 특별한 의미를 가지는 경우, 예를 들어 3번째 비트를 1로 세트하는 것이 3번째 LED를 켜는 동작인 경우 코드의 가독성을 높이기 위해 #define 문장을 사용하기도 한다.

```
#define LED_3_ON      2
a = a | (0x01 << LED_3_ON);
```

아두이노에서는 비트 세트를 위해 bitSet이라는 매크로 함수를 제공하고 있으며 축약된 복합 연산자를 사용하는 점을 제외하면 앞의 코드와 동일하다.

```
bitSet(x, n)
매개변수
  x : 데이터
  n : 기록할 비트 위치, LSB부터 0에서 시작
반환값 : 없음

#define bitSet(value, bit) ((value) |= (1UL << (bit)))
```

아두이노에서는 마스크 생성을 위해 사용할 수 있는 bit 함수 역시 제공하고 있다.

```
bit(n)
  n : 계산하고자 하는 비트 위치
반환값 : n에 의해 지정된 비트의 비트값

#define bit(b) (1UL << (b))
```

bit 함수를 사용하여 bitSet 함수를 정의하면 다음과 같이 나타낼 수 있다.

```
#define bitSet(value, _bit) ((value) |= bit(_bit))
```

② 특정 비트 클리어

특정 비트 클리어는 특정 위치의 비트만을 0으로 설정하고 나머지는 현재 값을 그대로 유지하는 것을 말한다. 비트 AND 연산을 수행하면 그림 7에서 알 수 있듯이 0과 AND 시킨 결과는 항상 0이 되고 1과 AND 시킨 결과는 현재 값이 그대로 유지되며, 이를 통해 원하는 위치의 비트만을 0으로 만들 수 있다.

AND	클리어	1	0	1	0	유지
		0	0	1	1	
		0	0	1	0	

그림 7. 비트 AND 연산

변수 a에 a = 0bxxxxxxxx 값이 저장된 경우 5번째 비트만은 0으로 클리어하고 나머지 비트들은 현재 값을 그대로 유지하기 위해서는 그림 8에서와 같이 0b11101111과 AND 연산을 수행하면 된다.

AND	x	x	x	x	x	x	x	x
	1	1	1	0	1	1	1	1
	x	x	x	0	x	x	x	x

그림 8. 특정 비트 클리어

그림 8을 코드로 나타내면 다음과 같다.

```
REG = REG & 0b11101111;
REG = REG & 0xEF;
REG = REG & ~(0x01 << 4);
```

비트 클리어의 경우 마스크에서 클리어 하는 위치의 비트 값을 0으로 하고 나머지 비트들은 1로 설정해야 한다. 이는 비트 세트와는 반대의 경우로 비트 이동 연산자를 사용하는 경우 비트 세트에서와 동일한 방식으로 마스크를 만들고 이를 반전(~, 비트 NOT)시켜 사용하여야 한다. 아두이노에서는 비트 클리어를 위해 bitClear라는 매크로 함수를 제공하고 있다.

```
bitClear(x, n)
매개변수
  x : 데이터
  n : 기록할 비트 위치, LSB부터 0에서 시작
반환값 : 없음

#define bitClear(value, bit) ((value) &= ~(1UL << (bit)))
```

③ 특정 비트 반전

특정 비트 반전은 특정 위치의 비트만을 반전시키고 나머지는 현재 값을 그대로 유지하는 것을 말한다. 비트 XOR 연산을 수행하면 그림 9에서 알 수 있듯이 1과 XOR 시킨 결과는 비트가 반전되며 0과 XOR 시킨 결과는 현재 값이 그대로 유지되며, 이를 통해 원하는 위치의 비트만을 반전시킬 수 있다.

XOR	유지		반전	
	1	0	1	0
	0	0	1	1
	1	0	0	1

그림 9. 비트 XOR 연산

변수 a에 a = 0bxxxxxxx 값이 저장된 경우 4번째 비트만을 반전시키고 나머지 비트들은 현

재 값을 그대로 유지하기 위해서는 그림 10에서와 같이 0b00001000과 XOR 연산을 수행하면 된다.

XOR	x	x	x	x	x	x	x	x
	0	0	0	0	1	0	0	0
	x	x	x	0	$\bar{x}$	x	x	x

그림 10. 특정 비트 반전

그림 10을 코드로 나타내면 다음과 같다.

```
REG = REG ^ 0b00001000;
REG = REG ^ 0x08;
REG = REG ^ (1 << 3);
```

아두이노에도 비트 반전을 위한 매크로 함수는 정의되어 있지 않다.

④ 특정 비트 검사/읽기

특정 비트 검사/읽기는 특정 위치의 비트 값만을 알아내는 것을 말한다. 특정 비트 클리어에 서와는 반대로 구하고자 하는 비트 위치에 해당하는 마스크 값을 1로하고 나머지 비트 값을 0으로 설정하고 비트 AND 연산을 수행함으로써 특정 위치의 비트 값을 알아낼 수 있다. 변수 a에 a = 0bxxxxxxxx 값이 저장된 경우 6번째 비트 값을 알아내기 위해서는 그림 11에 서와 같이 0b00001000과 AND 연산을 수행하면 된다.

AND	x	x	x	x	x	x	x	x
	0	0	1	0	0	0	0	0
	0	0	x	0	0	0	0	0

그림 11. 특정 비트 검사

그림 11을 코드로 나타내면 다음과 같다.

```
if( (REG & 0x20) == 0x20 ){ // 6번째 비트 검사
    // 비트가 1인 경우
}
else{
    // 비트가 0인 경우
}
```

위의 코드에서 나타나 있듯이 특정 비트를 검사한 결과는 마스크와 동일한 값으로 비교하는 것이 안전하다. C 언어에서는 0이 아닌 모든 정수 값은 참(true)으로 간주하므로 마스크 값과 비교하지 않아도 결과는 동일하지만, 이 경우 정수값과 논리값이 혼재되어 나타나므로 피하는 것이 좋다.

```
if( REG & 0x20 ){ // 6번째 비트 검사
    // 비트가 1인 경우
    // 실제 if 내의 식은 32 또는 0의 값을 가지는 정수값이므로
    // 논리값과 정수값의 혼돈을 유발할 수 있으므로 추천하지 않는다.
}
else{
    // 비트가 0인 경우
}
```

특정 비트를 검사한 결과는 오른쪽 비트 이동(>>) 연산자를 이용하여 결과를 0 또는 1로 만들어 사용하기도 한다.

```

if( (REG & 0x20) >> 5) == 1 ){ // 6번째 비트 검사
    // 비트가 1인 경우
}
else{
    // 비트가 0인 경우
}

```

또는 값을 먼저 이동시킨 후 비트 연산을 수행하여도 동일한 결과를 얻을 수 있다.

```

if( (REG >> 5) & 0x01) == 1 ){ // 6번째 비트 검사
    // 비트가 1인 경우
}
else{
    // 비트가 0인 경우
}

```

아두이노에서는 비트 검사/읽기를 위해 bitRead라는 매크로 함수를 제공하고 있다.

```

bitRead(x, n)
매개변수
x : 데이터
n : 읽어낼 비트 위치로 LSB부터 0에서 시작
반환값 : 0또는 1의 비트값

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

```

4. 비트 패턴 만들기

코드 5로 돌아가 보자. 이제 패턴을 만드는 문장을 이해할 수 있겠는가?

```
PORTD = 0x01 << (7 - i);
```

변수 i 값에 따라 켜지는 LED의 위치를 조정하기 위해 비트 이동 연산을 사용하고 있다. 그림 12의 패턴을 생각해 보자.

비트 번호	7	6	5	4	3	2	1	0
패턴 1	■							■
패턴 2		■					■	
패턴 3			■			■		
패턴 4				■	■			
패턴 5				■	■			
패턴 6			■			■		
패턴 7		■					■	
패턴 8	■							■

그림 12. 8가지 LED 표시 패턴 (■ : LED ON, □ : LED OFF)

그림 12의 패턴은 왼쪽에서 오른쪽으로 움직이는 LED 하나와 오른쪽에서 왼쪽으로 움직이는 LED 하나로 구성되어 있다. 두 개의 LED 위치를 비트 이동 연산에 의해 결정하고 이를 OR 시킴으로써 패턴을 생성할 수 있다. 코드 6은 이를 구현한 예이다.

코드 6. 포트 D의 8개 LED 점멸 III - C 스타일

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

int main(void)
{
    DDRD = 0xFF;          // PORT D 8개 핀을 출력으로 설정

    while(1)
```

```

{
    for(int i = 0; i < 8; i++){
        char p1 = 0x01 << i;
        char p2 = 0x01 << (7 - i);
        PORTD = p1 | p2;
        _delay_ms(1000);
    }
}

return 1;
}

```

코드 7은 코드 6을 아두이노 스타일로 구현한 예이다. 아두이노에서는 핀 단위로 LED를 제어하여야 하므로 코드 6에 비해 알고리즘은 오히려 복잡하게 느껴질 수 있다.

코드 7. 포트 D의 8개 LED 점멸 III - 아두이노 스타일

```

int pins[] = {0, 1, 2, 3, 4, 5, 6, 7};          // 포트 D의 아두이노 핀 번호

void setup()
{
    for(int i = 0; i < 8; i++){                  // 포트 D의 핀들을 출력으로 설정
        pinMode(pins[i], OUTPUT);
    }
}

void loop()
{
    for(int i = 0; i < 8; i++){                  // 8개 패턴
        for(int j = 0; j < 8; j++){              // 패턴별 8개 LED
            if(j == i || j == (7 - i))           // 패턴 당 2개 LED만 켜고
                digitalWrite(pins[j], HIGH);
            else                                   // 나머지 6개는 끄
                digitalWrite(pins[j], LOW);
        }
    }
}

```

```
    delay(1000);  
  }  
}
```