

10. 디지털 출력 확장 - 74595

아두이노 우노의 경우 14개의 디지털 입출력 핀과 6개의 아날로그 입력 핀을 제공하고 있으며 이들을 모두 디지털 입출력으로 사용하는 경우 20개의 디지털 출력을 사용할 수 있다. 하지만 1자리 7세그먼트 표시장치는 8개의 독립된 LED로 구성되어 있어 이를 제어하기 위해서는 8개의 디지털 출력을 필요로 하는 등 20개의 디지털 입출력으로는 제어할 수 있는 주변장치가 그리 많지 않은 것이 사실이다. 따라서 많은 입출력을 가지는 시스템을 구성하고자 하는 경우 디지털 출력 핀이 부족한 경우를 흔히 볼 수 있다. 더 많은 디지털 출력 핀이 필요하다면 아두이노 메가(Arduino Mega)와 같이 더 많은 수의 디지털 출력 핀을 제공하는 아두이노 보드를 사용할 수도 있다. 아두이노 메가의 경우 70개의 디지털 입출력을 사용할 수 있으므로 아두이노 우노에 비해 연결할 수 있는 주변장치가 많지만 아두이노 우노에서 한두 개의 디지털 입출력 핀만이 더 필요한 경우 아두이노 메가를 사용하기에 가격이나 크기 등의 면에서 적합하지 않을 수 있다. 이런 경우 사용할 수 있는 것이 디지털 입출력 확장으로, 이 장에서는 직렬 입력 병렬 출력 레지스터인 74595를 사용하여 디지털 출력을 확장하는 방법에 대해 알아본다.

1. 74595

74595 칩은 16핀의 칩으로 아두이노에서 하나의 데이터 입력 핀과 두 개의 제어 핀, 총 세 개의 핀을 사용하여 8개의 디지털 출력이 가능하게 해준다.

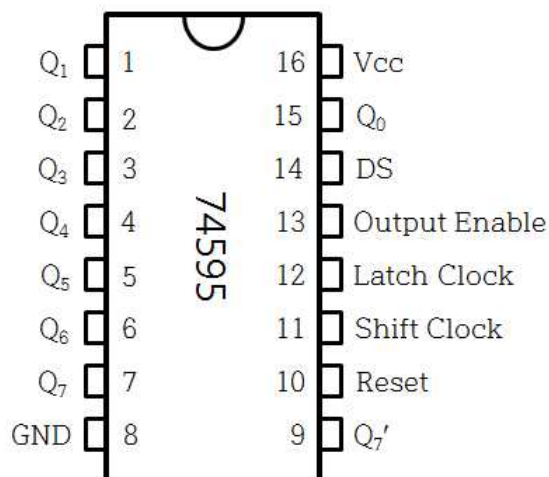


그림 1. 74595 핀 배치도

74595 칩은 직렬로 8비트의 데이터를 입력 받아 이를 병렬로 출력하는 (Serial In Parallel Out) 역할을 한다. 그림 1에서 Q_0 에서 Q_7 는 데이터 출력 핀이며 14번 핀 DS가 입력에 해당한다. 74595에는 11번과 12번 핀에 두 개의 CLOCK이 필요하다. 11번 핀의 Shift Clock은 송신단에서 전송하는 데이터를 동기화하기 위해 사용되는 핀이다. 반면 12번 Latch Clock은 74595 칩에서 병렬로 데이터가 출력되는 시점을 결정한다. 12번 Latch Clock은 13번 Output Enable 핀과 함께 사용된다. Output Enable은 LOW 입력이 가해질 때 출력단의 래치(latch)를 활성화 시켜 출력이 가능하도록 해주는 역할을 하므로 Output Enable 핀에 LOW가 가해진 상태에서 12번 Latch Clock이 주어지는 경우 74595 칩에서 병렬로 출력이 일어난다. 따라서 Output Enable을 GND(LOW 상태)에 연결하고 출력 시점은 Latch Clock으로 조절하는 것이 일반적이다. 10번 Reset 핀은 비동기 초기화 핀으로 Vcc에 연결시키면 아무런 동작도 수행하지 않는다. 9번 Q_7' 는 쉬프트 레지스터의 값을 직렬로 출력하는 핀으로 74595를 여러 개 연결시킬 수 있도록 해준다. 그림 2는 74595 칩의 동작을 나타낸 것이다.

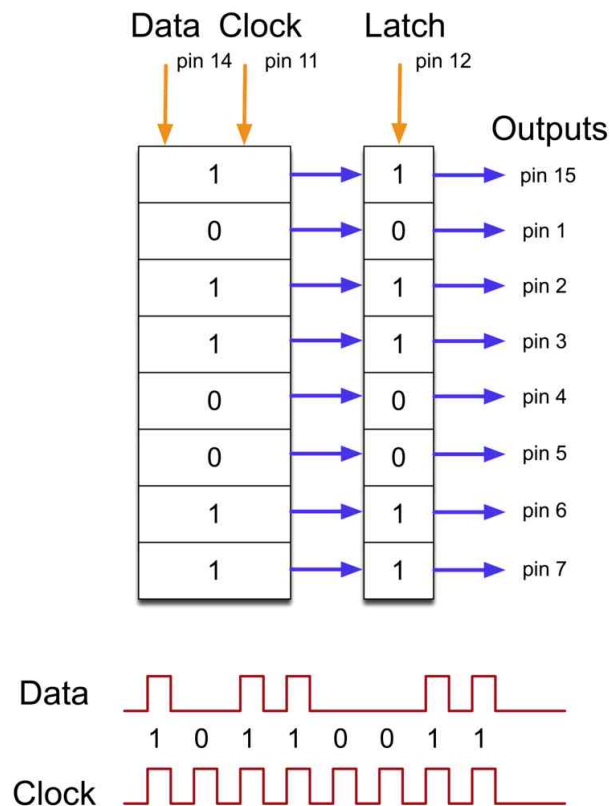


그림 2. 74595 칩의 데이터 전송

74595를 아두이노와 연결하기 위해서는 3개의 핀이 필요하다. 먼저 14번 데이터 입력 핀을 아두이노의 dataPin(아두이노 13번 디지털 핀)에 연결한다. 12번 Latch Clock은 아두이노의 latchPin(아두이노 12번 디지털 핀)에 연결하고 11번 Shift Clock은 아두이노의 clockPin(아두이노 13번 디지털 핀)에 연결한다. 13번 Output Enable은 GND를 입력하여 출력을 활성화 시키고(active low) latchPin으로 입력되는 클록에 의해 최종 출력이 조절되도록 한다. DM-UStyle EX 보드에는 74595 직렬 입력 병렬 출력 레지스터가 포함되어 있다.

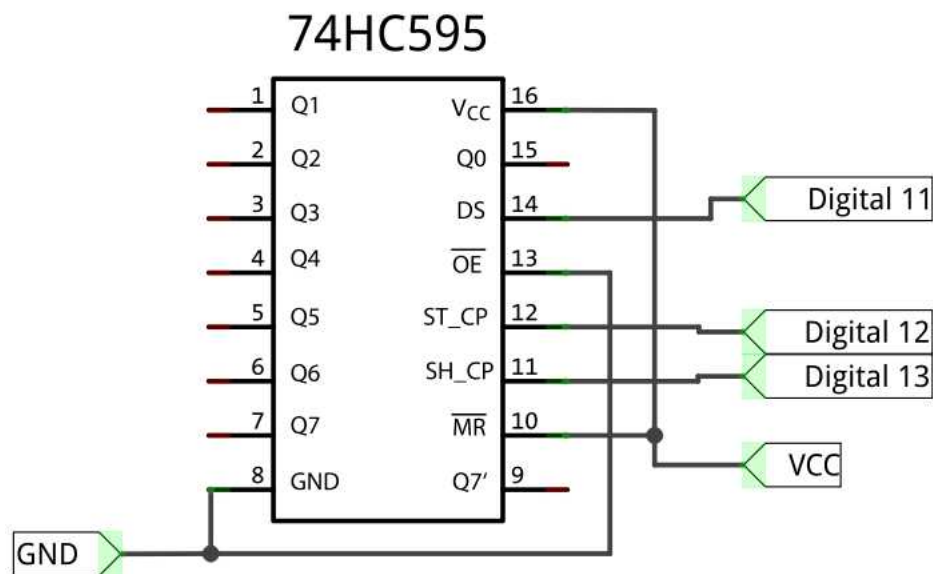


그림 2. 74595 칩과 아두이노 연결 회로도

74595 칩의 8개 출력을 1자리 7세그먼트 표시장치 제어를 위한 8개의 제어선과 연결해 보자. DM-UStyle EX 보드에는 74595 칩에 연결된 공통 양극 방식의 1자리 7세그먼트 표시장치 역시 포함되어 있다.

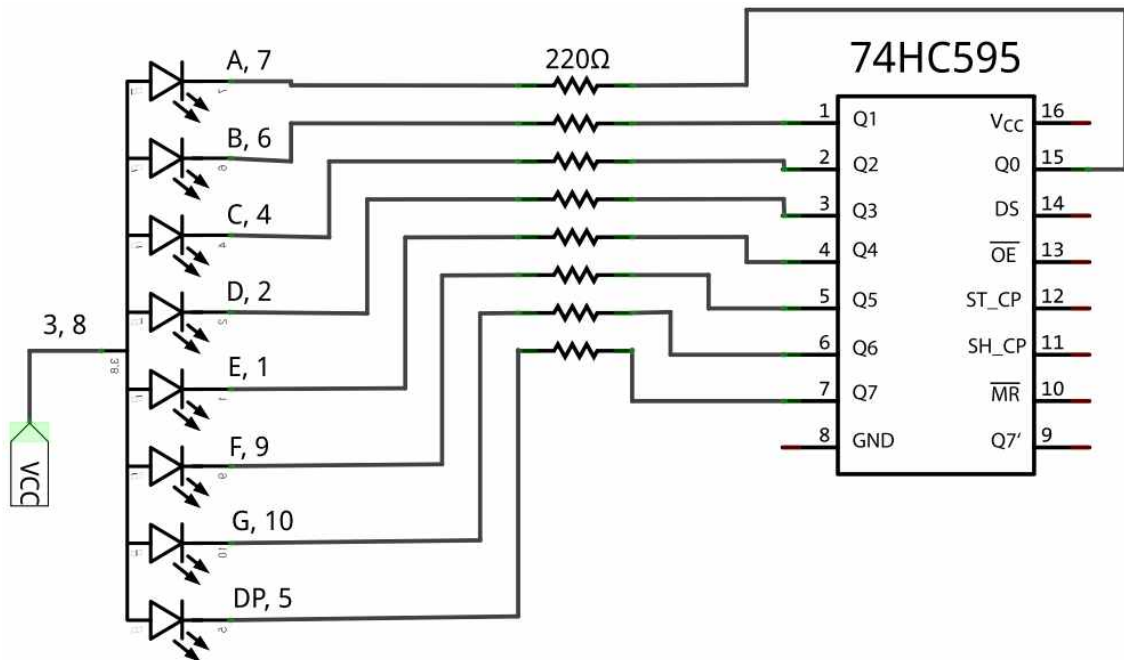


그림 3. 74595 칩과 1자리 7세그먼트 표시장치 연결 회로도

74595 칩을 사용하는 방법은 간단하다. 먼저 Latch Clock에 LOW 값을 출력하고, 데이터 (DS)와 동기화 클럭(Shift Clock)을 출력한 후 Latch Clock에 HIGH 값을 출력하면 실제로 74595 칩의 8개 출력 핀으로 동시에 출력이 발생한다. 스케치 1은 74595 칩에 연결된 1자리 7세그먼트 표시장치로 0에서 9까지 1초 간격으로 표시하는 예이다.

스케치 1. 74595 칩으로 1자리 7세그먼트 표시장치 제어

```
// 0에서 9까지 숫자 표현을 위한
// 세그먼트 a, b, c, d, e, f, g, dp의 패턴
byte patterns[] = {
    0xFC, 0x60, 0xDA, 0xF2, 0x66, 0xB6, 0xBE, 0xE4, 0xFE, 0xE6
};

int dataPin = 11;
int latchPin = 12;
int clockPin = 13;

int index = 0;
```

```

void setup() {
    pinMode(dataPin, OUTPUT);
    pinMode(clockPin, OUTPUT);
    pinMode(latchPin, OUTPUT);
}

void loop() {
    index = (index + 1) % 10;

    digitalWrite(latchPin, LOW);                // 래치 출력 금지
    shiftOut(dataPin, clockPin, LSBFIRST, ~patterns[index]); // 데이터 출력
    digitalWrite(latchPin, HIGH);               // 래치 출력 허용

    delay(1000);
}

```

스케치 1에서 데이터와 동기화 클럭 출력을 위해 shiftOut 함수를 사용하였다. shiftOut 함수의 원형은 다음과 같다.

```

void shiftOut(uint8_t dataPin, uint8_t clockPin, uint8_t bitOrder, uint8_t value)
    dataPin : 비트 출력이 일어날 핀
    clockPin : 한 비트 출력 후 데이터 출력을 알려주기 위해 펄스가 출력되는 핀
    bitOrder : 비트 출력 순서, MSBFIRST 또는 LSBFIRST
    value : 출력될 데이터 (byte)
    반환값 : 없음

```

shiftOut 함수는 지정한 데이터 값(value)를 지정한 비트 순서(bitOrder)에 따라 데이터(dataPin)와 동기화 클럭(clockPin)을 출력하는 함수이다. shiftOut 함수는 바이트 단위의 출력 함수이므로 255 이상의 값을 출력하기 위해서는 바이트 단위로 나누어 여러 번 출력하여야 한다. 비트 순서는 MSB나 LSB부터 출력하도록 지정할 수 있다. 0b11110000의 값을 출력하는 경우 MSBFIRST가 지정되면 1부터 출력되고 LSBFIRST가 지정되면 0부터 출력된다. 스케치 1에서도 LSB부터 출력하도록 되어 있으며 숫자 패턴이 세그먼트 a부터 세그먼트 dp 순

서로 지정되어 있으므로 세그먼트 a 제어 신호가 Q_0 에서, 세그먼트 dp 제어 신호가 Q_7 에서 출력된다.

세그먼트	LSBFIRST에서 출력 순서	74595 출력 핀
a	8	Q_0
b	7	Q_1
c	6	Q_2
d	5	Q_3
e	4	Q_4
f	3	Q_5
g	2	Q_6
dp	1	Q_7

그림 4. 74595로 1자리 7세그먼트 표시장치 제어

1자리 7세그먼트 모듈을 74595 칩의 출력 핀 헤더에 연결해 보자. 1자리 7세그먼트 모듈에 사용된 7세그먼트 역시 DM-UStyle EX 보드에 포함되어 있는 것과 동일한 공통 양극 방식이다. 이 때 주의할 점은 세그먼트 a에 해당하는 제어 신호를 74595 칩의 Q_0 출력과 연결하여야 한다는 점이다. 숫자가 제대로 표시되지 않는다면 연결 순서가 잘못된 것이므로 연결 순서를 반대로 하거나 스케치 1에서 LSBFIRST를 MSBFIRST로 바꾸고 시도해 보자.

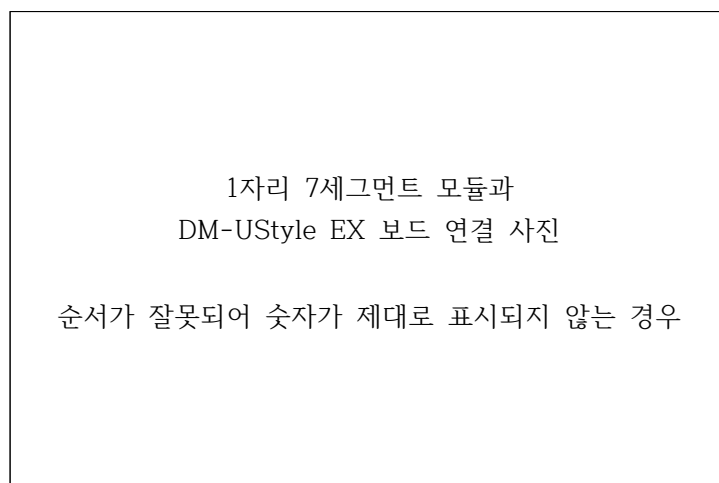


그림 5. 1자리 7세그먼트 모듈을 74595에 연결

2. LED 모듈 제어

74595 칩의 데이터 출력 핀 헤더에 LED 모듈을 연결해 보자.

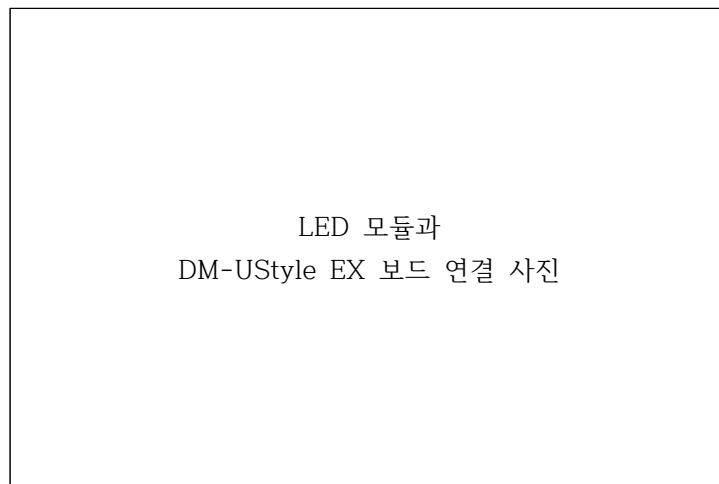


그림 6. LED 모듈을 74595에 연결

스케치 2는 74595에 연결된 LED 모듈에 그림 7의 패턴을 0.5초 간격으로 반복해서 출력하는 스케치의 예이다.

패턴	LED 1	LED 2	LED 3	LED 4	LED 5	LED 6	LED 7	LED 8
1	ON	OFF	ON	OFF	OFF	ON	OFF	ON
2	OFF	ON	OFF	ON	ON	OFF	ON	OFF
3	OFF	OFF	ON	OFF	OFF	ON	OFF	OFF
4	OFF	OFF	OFF	ON	ON	OFF	OFF	OFF
5	OFF	OFF	ON	OFF	OFF	ON	OFF	OFF
6	OFF	ON	OFF	ON	ON	OFF	ON	OFF

그림 7. LED 출력 패턴 (: ON, : OFF)

스케치 2. LED 모듈에 패턴 나타내기

```
// 출력 패턴
byte patterns[] = {
    0b10100101,
    0b01011010,
    0b00100100,
    0b00011000,
    0b00100100,
    0b01011010 };

int dataPin = 11;
int latchPin = 12;
int clockPin = 13;

int index = 0;

void setup() {
    pinMode(dataPin, OUTPUT);
    pinMode(clockPin, OUTPUT);
    pinMode(latchPin, OUTPUT);
}

void loop() {
    index = (index + 1) % 6;

    digitalWrite(latchPin, LOW);           // 래치 출력 금지
    shiftOut(dataPin, clockPin, MSBFIRST, patterns[index]); // 데이터 출력
    digitalWrite(latchPin, HIGH);          // 래치 출력 허용

    delay(500);
}
```

1자리 7세그먼트 모듈이 공통 양극 방식인 반면 LED 모듈은 공통 음극 방식으로 연결되어 있으므로 패턴을 반전시키지 않고 출력한다는 점에 유의하여야 한다. 또한 그림 7의 패턴은 좌우 대칭이므로 shiftOut 함수에서 MSB 또는 LSB 우선 출력을 선택하여도 동일한 결과를 가져온다.

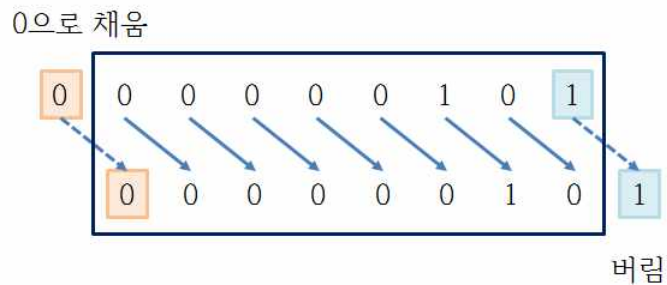
3. 원형 비트 쉬프트 연산

지금까지 LED 모듈에 패턴을 나타내기 위해서는 모든 패턴을 배열로 저장한 후 이를 이용하였다. 하지만 자리 이동만 있는 패턴들의 경우에는 쉬프트 연산자를 이용하여서도 가능하다. 그림 8의 패턴을 생각해 보자.

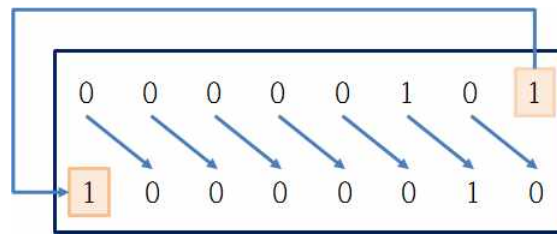
패턴	LED 1	LED 2	LED 3	LED 4	LED 5	LED 6	LED 7	LED 8
1	ON	OFF	ON	OFF	OFF	OFF	OFF	OFF
2	OFF	ON	OFF	ON	OFF	OFF	OFF	OFF
3	OFF	OFF	ON	OFF	ON	OFF	OFF	OFF
4	OFF	OFF	OFF	ON	OFF	ON	OFF	OFF
5	OFF	OFF	OFF	OFF	ON	OFF	ON	OFF
6	OFF	OFF	OFF	OFF	OFF	ON	OFF	ON
7	ON	OFF	OFF	OFF	OFF	OFF	ON	OFF
8	OFF	ON	OFF	OFF	OFF	OFF	OFF	ON

그림 8. LED 출력 패턴 ( : ON,  : OFF)

8개의 패턴을 저장하고 표현할 수도 있지만 패턴의 표현 순서를 살펴보면 1번에서 6번 패턴까지는 우측 쉬프트 연산(>>)을 사용하면 가능하다. 하지만 6번 패턴 0b00000101을 우측 쉬프트하면 0b00000010을 얻게 되고 이는 7번 패턴과는 다르다. 이 경우 사용할 수 있는 연산이 원형 쉬프트 연산이다. 그림 9는 쉬프트 연산과 원형 쉬프트 연산을 비교한 것이다.



(a) 쉬프트 연산

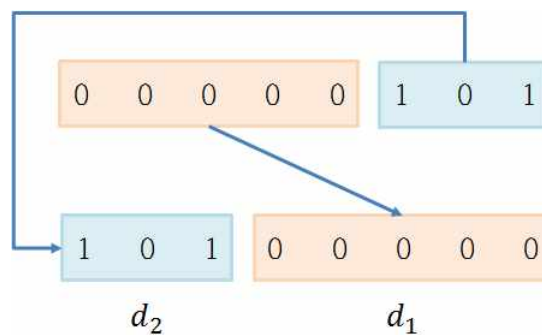


(b) 원형 쉬프트 연산

그림 9. 쉬프트 연산과 원형 쉬프트 연산

원형 쉬프트 연산을 사용하면 그림 8의 패턴을 간단하게 구현이 가능하지만 C/C++ 언어는 원형 쉬프트를 위한 별도의 연산자를 제공하지는 않는다. 따라서 C/C++ 언어의 비트 연산자들을 사용하여 원형 쉬프트 연산을 구현하여야 한다.

8비트의 데이터 d 를 오른쪽으로 $n(= 3)$ 비트 원형 쉬프트 하는 연산자를 생각해 보자. 먼저 데이터 d 를 $n(= 3)$ 비트와 $8 - n(= 5)$ 비트로 나누어야 한다.

그림 10. $n(= 3)$ 비트 오른쪽 원형 쉬프트 연산

$8 - n(= 5)$ 비트 부분(d_1)은 d 를 $n(= 3)$ 비트 오른쪽으로 쉬프트 연산을 수행하면 얻을 수 있다. 반면 $n(= 3)$ 비트 부분(d_2)은 d 를 $8 - n(= 5)$ 비트 왼쪽으로 쉬프트 연산을 수행하면 얻을 수 있다. 쉬프트된 최종 결과는 d_1 과 d_2 의 비트 단위 OR 연산으로 구할 수 있다.

```
d1 = d >> n;
d2 = d << (8 - n);

result = d1 | d2;
```

오른쪽 원형 쉬프트 연산을 사용하여 그림 8의 패턴들을 0.5초 간격으로 LED 모듈에 표시하도록 하는 스케치의 예가 스케치 3이다.

스케치 3. 오른쪽 원형 쉬프트 연산으로 LED 모듈에 패턴 나타내기

```
byte pattern = 0b10100000;           // 기본 출력 패턴

int dataPin = 11;
int latchPin = 12;
int clockPin = 13;

void setup() {
  pinMode(dataPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(latchPin, OUTPUT);
}

byte circular_right_shift(byte d, int n)
{
  byte d1 = d >> n;
  byte d2 = d << (8 - n);

  return (d1 | d2);
}

void loop() {
  pattern = circular_right_shift(pattern, 1);

  digitalWrite(latchPin, LOW);         // 래치 출력 금지
  shiftOut(dataPin, clockPin, MSBFIRST, pattern); // 데이터 출력
  digitalWrite(latchPin, HIGH);        // 래치 출력 허용

  delay(500);
}
```

유사한 방법으로 왼쪽 원형 쉬프트 연산도 구현할 수 있다.

```

d1 = d >> (8 - n);
d2 = d << n;

result = d1 | d2;

```

스케치 3은 오른쪽 원형 쉬프트 연산을 사용하고 있다. 이를 수정하여 버튼이 눌러진 경우 오른쪽 및 왼쪽 원형 쉬프트 연산을 전환하도록 해보자. 디지털 9번 핀에 풀다운 저항을 통해 버튼을 연결한다.

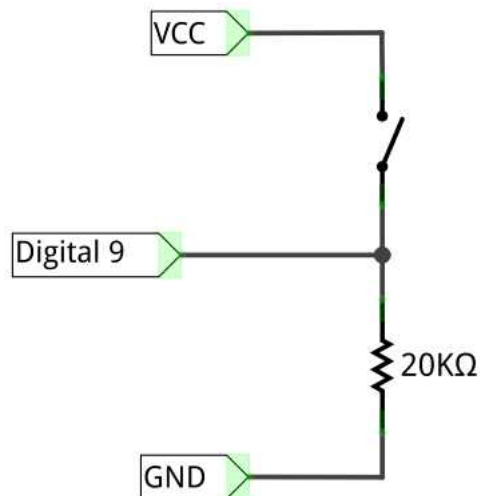


그림 11. 풀다운 저항으로 연결된 버튼

스케치 4. 원형 쉬프트 연산으로 LED 모듈에 패턴 나타내기

```

byte pattern = 0b10100000;           // 기본 출력 패턴

int dataPin = 11;
int latchPin = 12;
int clockPin = 13;

int button_pin = 9;                   // 버튼 연결 핀
boolean state = true;                 // 오른쪽 원형 쉬프트 사용

```

```
void setup() {
    pinMode(dataPin, OUTPUT);
    pinMode(clockPin, OUTPUT);
    pinMode(latchPin, OUTPUT);

    pinMode(button_pin, INPUT);          // 버튼 연결 핀을 입력으로 설정
}

byte circular_right_shift(byte d, int n)
{
    byte d1 = d >> n;
    byte d2 = d << (8 - n);

    return (d1 | d2);
}

byte circular_left_shift(byte d, int n)
{
    byte d1 = d >> (8 - n);
    byte d2 = d << n;

    return (d1 | d2);
}

void loop() {
    if(digitalRead(button_pin)){          // 버튼이 눌려진 경우
        state = !state;                  // 오른쪽 및 왼쪽 쉬프트 전환
        delay(500);                      // 디바운스
    }

    if(state){                            // 오른쪽 원형 쉬프트
        pattern = circular_right_shift(pattern, 1);
    }
    else{                                  // 왼쪽 원형 쉬프트
        pattern = circular_left_shift(pattern, 1);
    }
}
```

```

}

digitalWrite(latchPin, LOW);           // 래치 출력 금지
shiftOut(dataPin, clockPin, MSBFIRST, pattern); // 데이터 출력
digitalWrite(latchPin, HIGH);          // 래치 출력 허용

delay(500);
}

```

4. 1차원 세포 자동자 (cellular automaton)

1차원 세포 자동자를 만들어서 LED 모듈에 나타내 보자. 1차원 세포 자동자는 한 줄로 세포들을 배열한 것으로 세포는 살아 있거나 죽은 두 가지 상태 중 하나를 가진다. 세포는 왼쪽과 오른쪽 이웃 세포의 상태를 알아낼 수 있으며 이웃하는 두 세포들과 자신의 현재 상태에 따라 다음 상태를 결정한다. 예를 들어 이웃하는 세포들과 자신이 현재 살아 있는 상태이면 너무 갑갑해서 죽어 버리고, 죽은 상태에서 이웃한 두 세포 중 하나만 살아 있다면 다시 살아나는 식이다. 이웃한 세포들과 자신의 상태에 따라 삶과 죽음을 결정하는 규칙은 모두 8가지가 존재하며 그림 12와 같은 규칙을 사용한다고 가정해 보자.

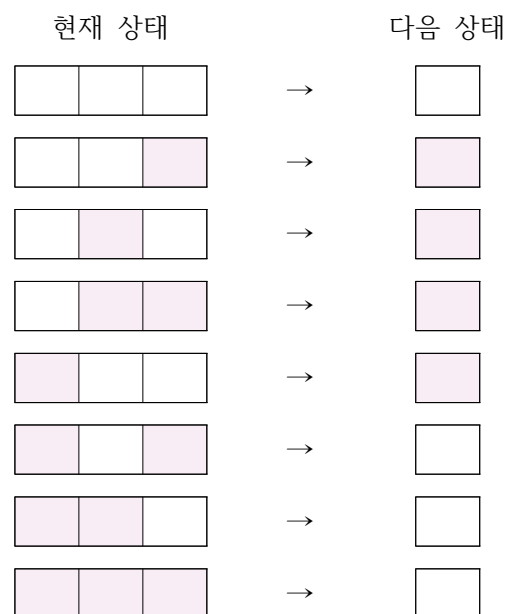


그림 12. 1차원 세포 자동자의 규칙 ( : 살아 있는 상태,  : 죽은 상태)

LED 모듈의 각 LED는 하나의 세포에 해당하므로 byte 형의 데이터로 8개의 세포를 표현할 수 있다. 하지만 각 세포의 다음 상태를 결정할 때 MSB의 경우 왼쪽, LSB의 경우 오른쪽 이웃 세포가 존재하지 않는다. 이처럼 이웃하는 세포가 존재하지 않는 경우에는 랜덤 함수를 사용하여 살아있거나 또는 죽은 세포를 임의로 생성하여 사용하도록 한다. 스케치 5는 그림 12의 규칙을 사용하여 세포 자동자를 LED 모듈에 나타내는 스케치의 예이다.

스케치 5. 세포 자동자

```
byte pattern_previous, pattern_current;
int rules[] = {0, 1, 1, 1, 1, 0, 0, 0}; // 세포의 생성 소멸 규칙

int dataPin = 11;
int latchPin = 12;
int clockPin = 13;

void setup(){
  pinMode(dataPin, OUTPUT);
  pinMode(clockPin, OUTPUT);
  pinMode(latchPin, OUTPUT);

  pattern_previous = 0b00010000; // 세포 자동자 초기 상태
  randomSeed(analogRead(A5)); // 랜덤 함수 초기화
}

void loop(){
  // 현재 상태 계산
  pattern_current = 0;
  for(int i = 0; i < 8; i++){
    int left = i + 1; // 왼쪽 세포 인덱스
    int right = i - 1; // 오른쪽 세포 인덱스

    int value_left, value_right; // 이웃 세포의 상태
```

```

    if(left >= 8){                                // 왼쪽 이웃 세포가 없는 경우
        value_left = random(0, 2);                // 임의로 결정
    }
    else{                                          // 왼쪽 이웃 세포의 상태
        value_left = bitRead(pattern_previous, left);
    }

    if(right < 0){                                // 오른쪽 이웃 세포가 없는 경우
        value_right = random(0, 2);               // 임의로 결정
    }
    else{                                          // 오른쪽 이웃 세포의 상태
        value_right = bitRead(pattern_previous, right);
    }

    // 자신의 상태
    int value_center = bitRead(pattern_previous, i);

    // 자신과 이웃 세포들의 상태로 생성 규칙 번호 결정
    int index = value_left * 4 + value_center * 2 + value_right;
    // 현재 상태 기록
    bitWrite(pattern_current, i, rules[index]);
}

pattern_previous = pattern_current;

digitalWrite(latchPin, LOW);                     // 래치 출력 금지
shiftOut(dataPin, clockPin, MSBFIRST, pattern_current); // 데이터 출력
digitalWrite(latchPin, HIGH);                    // 래치 출력 허용

delay(500);
}

```

세포 자동자는 계산가능성 이론(computability theory)과 수학에서 연구되는 이산 모델 중 하나로 John Conway가 생명 게임(Life Game)을 만들어 더욱 유명해졌다. LED 모듈에 표시하는 세포 자동자는 Conway의 2차원 생명 게임을 단순화시킨 것으로 2차원 생명 게임의 경

우 단순한 계산에 의해 다양한 패턴이 나타나므로 해당 사이트¹⁾를 방문해 보기를 추천한다.

1) http://en.wikipedia.org/wiki/Conway's_Game_of_Life
