

Chapter 107.5. 버튼 입력

1. 풀업 저항과 풀다운 저항

이전 장에서 LED를 이용하여 이진 데이터를 출력하는 방법에 대해 알아보았다. 하지만 마이크로컨트롤러를 사용하는 데 있어 데이터 출력뿐만이 아니라 데이터를 마이크로컨트롤러로 입력할 수 있는 방법이 있어야 상호작용이 가능해진다. 마이크로컨트롤러가 주변 환경으로부터 데이터를 입력받고 이를 처리하여 처리된 결과를 출력하기 위한 목적을 가진다는 사실을 기억하는가? 이 장에서는 간단한 데이터 입력 방법으로 버튼을 사용하는 방법에 대해 알아본다. 버튼을 통한 데이터 입력 방법은 마이크로컨트롤러에 데이터를 입력하기 위해 사용하는 센서로 확장이 가능하다. 물론 버튼의 경우 1비트의 데이터만을 마이크로컨트롤러로 입력하는 역할을 하지만 센서의 경우에는 여러 비트, 일반적으로 바이트 단위의 데이터를 입력하는 면에서는 차이가 있다.

버튼은 주변에서 흔히 볼 수 있는 입력 장치로 이진 데이터 입력을 위해 사용할 수 있다. 하지만 버튼을 사용하는 경우 몇 가지 주의할 점이 있으며 그 중 하나가 개방 회로(open circuit)를 입력으로 사용해서는 안된다는 점이다. 생각할 수 있는 가장 간단한 버튼 연결 회로는 그림 1과 같이 버튼을 범용 입출력 핀과 VCC 사이에 연결하는 방법이다.

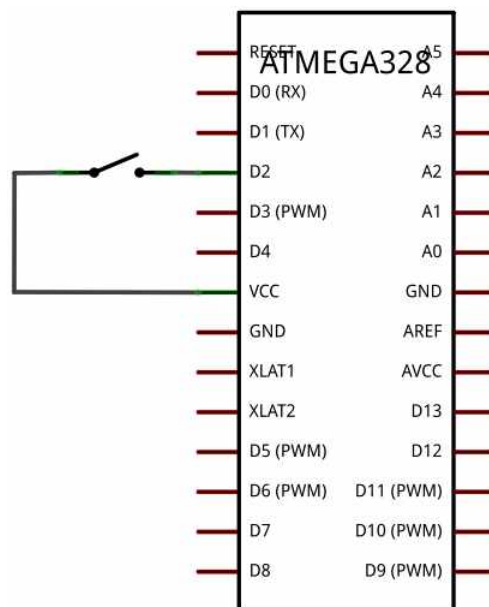


그림 1. 버튼 연결 회로도

그림 1에서 버튼은 PD2 핀 (아두이노 디지털 2번 핀)과 VCC 사이에 연결되어 있으므로 버튼을 누르면 PD2 핀에 5V의 전압이 가해지고, PIN 레지스터를 통해 논리 1의 값을 확인할 수 있다. 하지만 버튼이 눌러지지 않은 경우에는 어떻게 될까? 버튼이 눌러지지 않은 경우 PIN 레지스터를 읽으면 어떤 값이 나올까? 정답은 ‘알 수 없다’이다. 그림 1의 회로에서 버튼이 눌러지지 않은 경우 PD2 핀에 입력되는 값은 인접한 핀의 전압이나 정전기 등의 영향을 받으므로 PIN 레지스터에는 무작위 값이 저장된다. 이처럼 버튼이 눌러지지 않아 개방된 회로는 플로팅(floating)되어 있다고 이야기 한다. 플로팅 상태를 제거하기 위해 사용할 수 있는 방법 중 하나가 풀업 저항을 사용하는 것이다. 그림 2의 회로도를 보자.

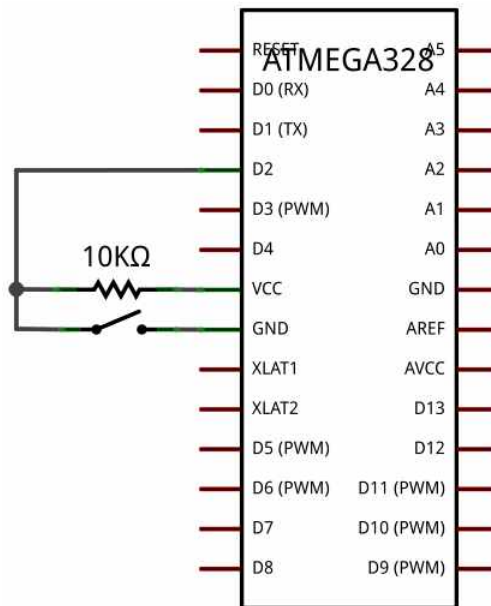


그림 2. 풀업 저항을 사용한 버튼 연결 회로도

그림 2에서 버튼이 눌러지지 않은 경우 PD2 핀에는 10KΩ의 풀업 저항을 통해 VCC가 가해진다. 반면 버튼이 눌러진 경우에는 풀업 저항보다 낮은 저항을 가지는 스위치를 통해 경로가 형성되고 GND가 가해진다. 즉, 풀업 저항을 통해 개방 회로는 해결되었다. 한 가지 주의할 점은 풀업 저항을 사용하는 경우 버튼이 눌러진 경우 논리 0이 가해지고, 버튼이 눌러지지 않은 경우 논리 1이 가해진다는 점이다. 이는 그림 1에서 예상했던 동작과는 반대이며 흔히 생각하는 방식과도 반대로 동작한다. 그렇다면 직관적인 방식으로 동작하도록 버튼을 구성할 수는 없을까? 그림 3의 회로도를 살펴보자.

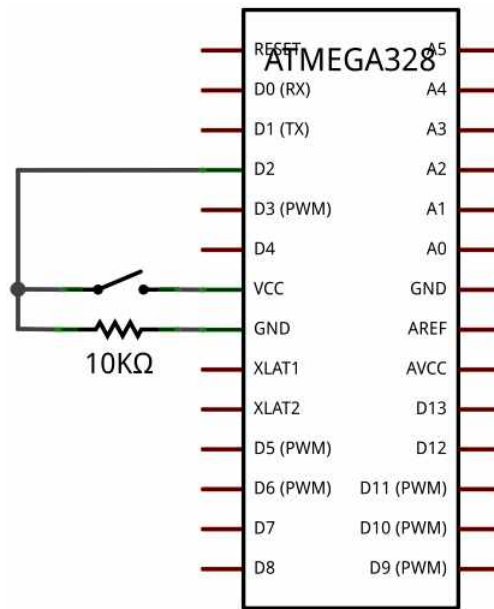


그림 3. 풀다운 저항을 사용한 버튼 연결 회로도

그림 3에서 버튼이 눌러지지 않은 경우에는 저항을 통해 GND가 가해지고 버튼이 눌러진 경우 버튼을 통해 VCC가 가해져서 직관적인 방식과 동일하게 동작한다. 그림 3에서 사용된 저항을 풀다운 저항이라고 부른다.

ATmega328의 범용 입출력 핀에는 프로그램으로 제어할 수 있는 풀업 저항이 포함되어 있다. 풀업 저항이 직관적인 방식과 반대로 동작하기는 하지만 풀업 저항이 풀다운 저항에 비해 구현하기가 쉽고, VCC보다는 GND가 보다 안정적인 전압을 유지하는 등의 장점으로 대부분의 마이크로컨트롤러에서는 풀업 저항을 사용하고 있다. 이 책에서도 회로 구성을 간단히 하고 흔히 사용되는 방식과 동일하게 풀다운 저항보다는 범용 입출력 핀에 내장된 풀업 저항을 사용할 것이다.

2. 버튼 입력

범용 입출력 핀을 사용하기 위해 가장 기본적인 레지스터에는 DDRx, PORTx, PINx (x = B, C, D)의 3개 레지스터가 있다. 이 중 DDRx 레지스터는 범용 입출력 핀의 데이터 이동 방향, 즉, LED를 켜는 동작과 같이 출력으로 사용할 것인지 버튼의 상태를 얻어오는 입력으로 사용할 것인지 설정하기 위해 사용된다. DDRx 레지스터의 디폴트값은 0으로 입력 상태에 있으므로 버튼 입력으로 사용하기 위해서는 별도로 설정할 필요가 없지만 일관적인 사용과 혼란을 방지하기 위해 가능한 DDRx 레지스터 설정을 추천한다.

비트	7	6	5	4	3	2	1	0
비트 이름	DDx7	DDx6	DDx5	DDx4	DDx3	DDx2	DDx1	DDx0
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

(1은 출력, 0은 입력 상태)

그림 4. DDRx 레지스터 구조

범용 입출력 핀이 입력으로 설정되었으면 현재 입력 핀에 가해진 디지털 데이터를 읽어 오기 위해 사용하는 레지스터가 PINx 레지스터이다.

비트	7	6	5	4	3	2	1	0
비트 이름	PINx7	PINx6	PINx5	PINx4	PINx3	PINx2	PINx1	PINx0
읽기/쓰기	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
초깃값	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A

그림 5. PINx 레지스터 구조

데이터를 출력하기 위해 PORTx 레지스터를 사용했던 것을 기억하는가? 버튼 입력을 위해서도 PORTx 레지스터가 사용된다. DDRx 레지스터가 1의 값으로 설정된 경우 PORTx 레지스터는 출력 값을 저장하기 위해 사용된다. 반면 DDRx 레지스터가 0의 값으로 설정된 경우 PORTx 레지스터는 풀업 저항의 사용 여부를 나타내기 위해 사용된다.

비트	7	6	5	4	3	2	1	0
비트 이름	PORTx7	PORTx6	PORTx5	PORTx4	PORTx3	PORTx2	PORTx1	PORTx0
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. PORTx 레지스터 구조

풀업 저항을 사용하기 위해서는 DDRx 레지스터가 (0의 값으로) 입력 상태에 있을 때 PORTx 레지스터에 1의 값을 출력하면 해당 입출력 핀의 풀업 저항이 연결된다. 코드 1은 PD2 핀에 연결된 버튼을 사용하기 위해 입력 상태로 설정하고 풀업 저항을 연결한 후 버튼 상태를 읽어

오는 코드이다.

코드 1. 풀업 저항을 사용하는 버튼 설정

```
DDRD &= ~(1 << PD2);          // PD2 핀(디지털 2번 핀)을 입력으로 설정
PORTD |= (1 << PD2);          // PD2 핀의 풀업 저항 연결
// PD2에 연결된 버튼 상태를 0이나 1 중의 값으로 읽어온다.
buttonState = (PIND & (1 << PD2)) >> PD2;
```

코드 1을 이용하여 버튼이 눌러진 경우 PB5 핀(아두이노 13번 핀)에 연결된 LED를 켜는 프로그램을 작성해 보자. 아두이노 우노 보드의 13번 핀에는 내장 LED가 포함되어 있으므로 별도의 LED를 연결하지 않아도 된다. 또한 풀업 저항이 사용되었으므로 버튼이 눌러진 경우 0의 값을 얻을 수 있으므로 버튼의 상태를 읽어 반전된 값을 LED로 출력하여야 한다는 점에 유의하여야 한다.

코드 2. 버튼으로 LED 제어하기 - C 스타일

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

void INIT_PORT(void)
{
    DDRB = 0x20;          // PB5 (디지털 13번 핀) 출력으로 설정
    PORTB = 0x00;         // LED는 꺼진 상태에서 시작

    DDRD = 0x00;          // 버튼 입력
    PORTD = 0x04;         // PD2 (디지털 2번 핀) 풀업 저항 사용
}

int main(void)
{
    INIT_PORT();          // 포트 설정
```

```

while(1){
    if(PIND & 0x04)    PORTB = 0x00; // LED 끄기
    else               PORTB = 0x20; // LED 켜기
}
}

```

코드 2를 아두이노 스타일의 코드로 작성한 예가 코드 3이다.

코드 3. 버튼으로 LED 제어하기 - 아두이노 스타일

```

void setup()
{
    pinMode(2, INPUT_PULLUP);           // 내부 풀업 저항을 사용하는 버튼 입력

    pinMode(13, OUTPUT);                // 13번 핀을 출력으로 설정
    digitalWrite(13, LOW);              // LED는 꺼진 상태로 시작
}

void loop()
{
    if(digitalRead(2)) digitalWrite(13, LOW); // LED 끄기
    else digitalWrite(13, HIGH);            // LED 켜기
}

```

버튼을 누르면 LED가 켜지고 버튼을 떼면 LED가 꺼지는 것을 볼 수 있을 것이다.

코드 2에서 버튼이 눌러졌는지를 판단하기 위해 비트 AND 연산자를 사용하였다. 비트 연산이 마이크로컨트롤러 프로그래밍에서 흔히 사용되고 기본이 되는 방법이기기는 하지만 이해하기가 쉽지 않으며 잦은 오류의 원인이 되는 것도 사실이다. 아트멜 스튜디오에서 사용하는 컴파일러인 avr-gcc에는 비트 연산을 보다 직관적으로 이해할 수 있는 몇 가지 매크로 함수를 제공하고 있다. 매크로 함수 중에는 특정 비트가 1로 설정되었

는지 검사하는 `bit_is_set`와, 특정 비트가 0으로 설정되었는지 검사하는 `bit_is_clear` 함수가 있다.

```
#define bit_is_set(sfr, bit) (_SFR_BYTE(sfr) & _BV(bit))
#define bit_is_clear(sfr, bit) (!(_SFR_BYTE(sfr) & _BV(bit)))
```

PORTD의 네 번째 비트인 PD3 비트가 1로 설정되었는지 검사하기 위해서는 `bit_is_set(PORTD, PD3)`와 같이 사용할 수 있다. 단, 이들 매크로 함수는 정의에서 보듯이 참인 경우 0이 아닌 값을 반환한다.

이외에도 유용하게 사용할 수 있는 매크로 함수에는 레지스터의 특정 비트가 1로 설정될 때까지 대기하는 `loop_until_bit_is_set`와, 레지스터의 특정 비트가 0으로 설정될 때까지 대기하는 `loop_until_bit_is_clear`가 있다.

```
#define loop_until_bit_is_set(sfr, bit) do { } while (bit_is_clear(sfr, bit))
#define loop_until_bit_is_clear(sfr, bit) do { } while (bit_is_set(sfr, bit))
```

3. 채터링

버튼이 눌러졌을 때 LED에 불을 켜는 코드 2나 코드 3은 실행 과정에서 별다른 문제점을 발견할 수 없을 것이다. 코드 4를 보자. 코드 4는 버튼이 눌러진 횟수를 세는 프로그램으로 버튼이 눌러지지 않은 상태에서 눌러진 상태로 바뀔 때마다 UART 통신을 통해 '*' 문자를 전송하여 터미널에 표시한다. 프로그램을 업로드 하고 버튼을 눌러보자.

코드 4. 버튼이 눌러진 횟수 세기

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

void UART_INIT(void) {
    UCSR0A |= _BV(U2X0);           // 2배속 모드
```

```

UBRR0H = 0x00;                // 통신 속도 (보율) 설정
UBRR0L = 207;

// 비동기, 8비트 데이터, 패리티 없음, 1비트 정지 비트 모드
UCSR0C |= 0x06;

UCSR0B |= _BV(RXEN0);         // 송수신 가능
UCSR0B |= _BV(TXEN0);
}

void UART_transmit(unsigned char data)
{
    while( !(UCSR0A & (1<<UDRE0)) );    // 송신 가능 대기
    UDR0 = data;                        // 데이터 전송
}

void INIT_PORT(void)
{
    DDRB = 0x20;                  // PB5 (디지털 13번 핀) 출력으로 설정
    PORTB = 0x00;                // LED는 꺼진 상태에서 시작

    DDRD = 0x00;                 // 버튼 입력
    PORTD = 0x04;                // PD2 (디지털 2번 핀) 풀업 저항 사용
}

int main(void)
{
    UART_INIT();                 // UART 통신 초기화
    INIT_PORT();                 // 포트 설정
    int state = 0;               // 현재 버튼의 상태

    while(1){
        if(PIND & 0x04){
            state = 0;            // 버튼이 눌러지지 않은 상태로 표시
            PORTB = 0x00;         // LED 끄기
        }
    }
}

```

```

    }
    else{
        // 눌러지지 않은 상태에서 눌러진 상태로 바뀔 때마다 '*' 문자 전송
        if(state == 0) UART_transmit('*');
        state = 1;                // 버튼이 눌러진 상태로 표시
        PORTB = 0x20;             // LED 켜기
    }
}
}
}

```

코드 4를 실행하는 데 어떤 문제가 있는지 눈치 챘는가? 버튼을 한 번 누를 때마다 '*' 표시가 하나씩 늘어나기를 기대하지만 실제로 버튼을 눌러보면 두 개 이상의 문자가 표시되는 현상을 발견할 수 있을 것이다. 버튼을 누르는 경우 OFF 상태에서 ON 상태로 즉시 바뀌기를 기대하지만 실제로는 OFF에서 ON으로 여러 번 바뀐다. 이는 버튼을 누르거나 떼는 순간 두 개의 접점이 완전히 연결되거나 떨어지기 이전에 버튼 내부 스프링의 진동에 의해 미세하게 연결되고 떨어지기를 반복하기 때문에 발생한다. 이러한 현상을 바운스 현상(bounce effect) 또는 채터링(chattering)이라고 한다. 그림 7은 풀업 저항이 사용된 버튼에서 발생하는 채터링 현상의 예를 보인 것이다.

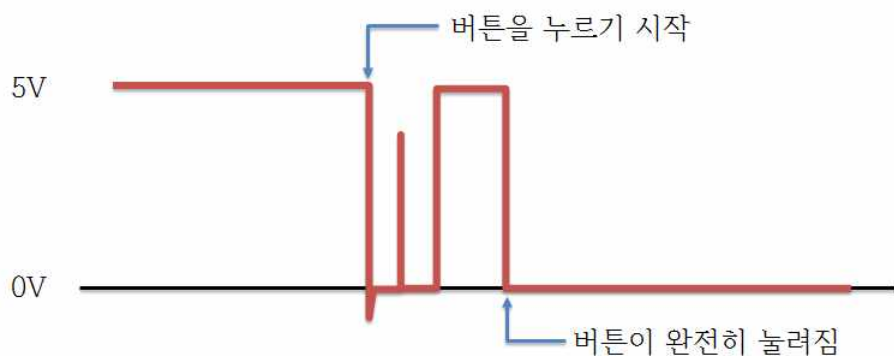


그림 7. 채터링 현상

채터링 현상을 없애는 것을 디바운스(de-bounce)라고 한다. 디바운스는 하드웨어적인 회로를 추가함으로써도 가능하지만 소프트웨어적으로도 처리할 수 있다. 디바운스를 위한 간단한 방법에는 버튼을 누르기 시작하는 시점을 찾아내는 방법과 버튼이 완전히 눌러진 시점을 찾아내는 방법이 있을 수 있다. 버튼을 누르기 시작하는 시점에서 입력은 LOW 값을 가지게 되며

이후 진동을 무시하기 위해서는 LOW 값이 입력된 점 이후 시간 지연 함수를 통해 이후 입력 값을 무시하는 방법이 있을 수 있다. 반면 버튼이 완전히 눌러진 시점을 찾아내기 위해서는 버튼 입력을 짧은 시간 내에 두 번 검사하여 두 번의 검사에서 모두 LOW 값이 입력되는 경우를 찾아내면 된다. 첫 번째 방법의 경우 단순히 시간 지연 함수를 사용함으로써 해결되지만 버튼이 안정화되기까지 최대 수백 ms의 지연 시간이 필요하다. 반면 두 번째 방법의 경우 두 번의 검사 간격은 최대 수십 ms의 지연 시간만이 필요하지만 두 번의 검사를 위해 코드가 복잡해지는 단점이 있다.¹⁾ 코드 5는 버튼이 눌러지기 시작하는 시점을 찾아내는 방법으로 버튼이 눌러진 것으로 판단되는 경우 `_delay_ms` 함수가 추가된 점을 제외하고는 코드 4와 동일하다.

코드 5. 디바운싱 - 버튼이 눌러지기 시작하는 시점 찾기

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

void UART_INIT(void) {
    UCSR0A |= _BV(U2X0);           // 2배속 모드

    UBRRH = 0x00;                  // 통신 속도 (보율) 설정
    UBRRL = 207;

    // 비동기, 8비트 데이터, 패리티 없음, 1비트 정지 비트 모드
    UCSR0C |= 0x06;

    UCSR0B |= _BV(RXEN0);          // 송수신 가능
    UCSR0B |= _BV(TXEN0);
}

void UART_transmit(unsigned char data)
{
    while( !(UCSR0A & (1<<UDRE0)) ); // 송신 가능 대기
    UDR0 = data;                    // 데이터 전송
}
```

1) 최대 지연 시간은 버튼의 종류에 따라 달라질 수 있으므로 사용하고자 하는 버튼에 맞는 시간 간격은 실험을 통해 찾아내어야 한다.

```

}

void INIT_PORT(void)
{
    DDRB = 0x20;           // PB5 (디지털 13번 핀) 출력으로 설정
    PORTB = 0x00;          // LED는 꺼진 상태에서 시작

    DDRD = 0x00;           // 버튼 입력
    PORTD = 0x04;          // PD2 (디지털 2번 핀) 풀업 저항 사용
}

int main(void)
{
    UART_INIT();           // UART 통신 초기화
    INIT_PORT();           // 포트 설정
    int state = 0;         // 현재 버튼의 상태

    while(1){
        if(PIND & 0x04){
            state = 0;      // 버튼이 눌러지지 않은 상태로 표시
            PORTB = 0x00;   // LED 끄기
        }
        else{
            // 눌러지지 않은 상태에서 눌러진 상태로 바뀔 때마다 '*' 문자 전송
            if(state == 0) UART_transmit('*');
            state = 1;       // 버튼이 눌러진 상태로 표시
            PORTB = 0x20;    // LED 켜기
            _delay_ms(100);  // 디바운싱
        }
    }
}

```

코드 6은 버튼이 완전히 눌러진 시점을 찾아내는 방법으로 두 번의 버튼 검사를 위해 button_pressed라는 별도의 함수를 작성하였다.

코드 6. 디바운싱 - 버튼이 완전히 눌러진 시점 찾기

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

void UART_INIT(void) {
    UCSRA |= _BV(U2X0);                // 2배속 모드

    UBRR0H = 0x00;                    // 통신 속도 (보율) 설정
    UBRR0L = 207;

    // 비동기, 8비트 데이터, 패리티 없음, 1비트 정지 비트 모드
    UCSRC |= 0x06;

    UCSRB |= _BV(RXEN0);              // 송수신 가능
    UCSRB |= _BV(TXEN0);
}

void UART_transmit(unsigned char data)
{
    while( !(UCSRA & (1<<UDRE0)) );    // 송신 가능 대기
    UDR0 = data;                       // 데이터 전송
}

void INIT_PORT(void)
{
    DDRB = 0x20;                      // PB5 (디지털 13번 핀) 출력으로 설정
    PORTB = 0x00;                     // LED는 꺼진 상태에서 시작

    DDRD = 0x00;                      // 버튼 입력
    PORTD = 0x04;                     // PD2 (디지털 2번 핀) 풀업 저항 사용
}

unsigned char button_pressed(void)
{

```

```

if(bit_is_clear(PIND, PIND2)){           // 첫 번째 검사
    _delay_ms(10);
    if(bit_is_clear(PIND, PIND2))         // 두 번째 검사
        return 1;
}
return 0;
}

int main(void)
{
    UART_INIT();                          // UART 통신 초기화
    INIT_PORT();                          // 포트 설정
    int state = 0;                        // 현재 버튼의 상태

    while(1){
        if(button_pressed()){
            // 눌러지지 않은 상태에서 눌러진 상태로 바뀔 때마다 '*' 문자 전송
            if(state == 0) UART_transmit('*');
            state = 1;                     // 버튼이 눌러진 상태로 표시
            PORTB = 0x20;                  // LED 켜기
        }
        else{
            state = 0;                     // 버튼이 눌러지지 않은 상태로 표시
            PORTB = 0x00;                  // LED 끄기
        }
    }
}

```

코드 7은 코드 6을 아두이노 스타일로 작성한 예이다.

코드 7. 디바운싱 - 버튼이 완전히 눌러진 시점 찾기 - 아두이노 스타일

```

int state = 0;                          // 현재 버튼의 상태

```

```
void setup()
{
    pinMode(2, INPUT_PULLUP);           // 내부 풀업 저항을 사용하는 버튼 입력
    pinMode(13, OUTPUT);                 // 13번 핀을 출력으로 설정

    Serial.begin(9600);                  // UART 통신 초기화
}

unsigned char button_pressed()
{
    if(!digitalRead(2)){                 // 첫 번째 검사
        delay(10);
        if(!digitalRead(2)) return 1;   // 두 번째 검사
    }

    return 0;
}

void loop()
{
    if(button_pressed()) {               // 버튼이 눌려진 경우
        if(state == 0) Serial.write('*');
        state = 1;
        digitalWrite(13, HIGH);
    }
    else{                                // 버튼이 눌러지지 않은 경우
        state = 0;
        digitalWrite(13, LOW);
    }
}
```