마이크로 컴퓨터

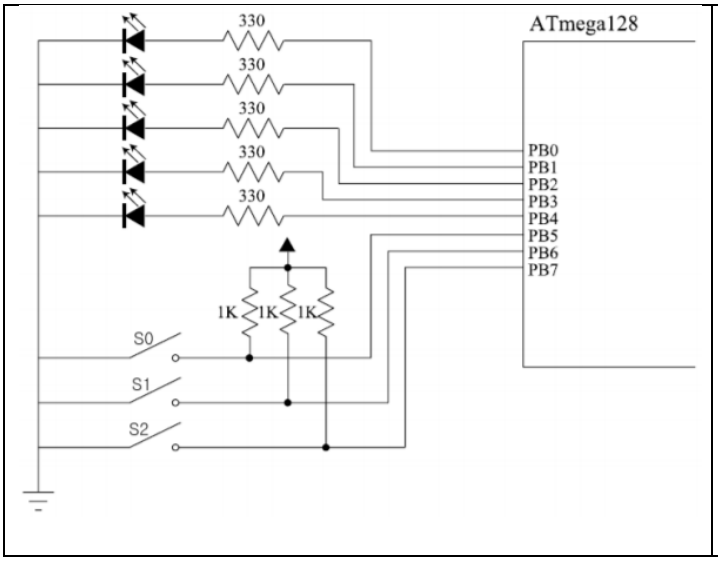
실험 1. LED ON/OFF

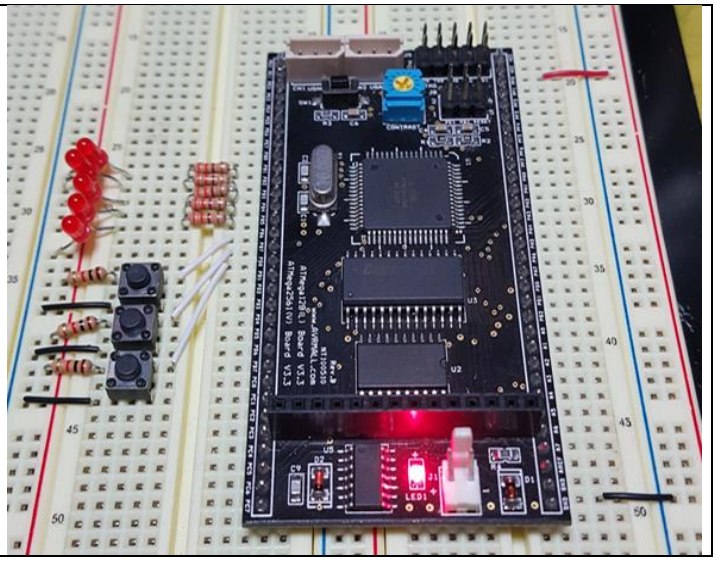
전자공학과 21611591 김난희

실험 목적

1. AVR 환경 구축과 간단한 실험으로 AVR 사용 방법 익히기.
2. 기본 소자(LED, 저항, 스위치)를 이용하여 LED ON/OFF 실험하기.
3. 추가 실험을 통해 풀업 저항의 의미 이해와 풀업 저항을 이용한 인터럽트 이해하기.

1. 실험 회로





1-1. 다이오드 회로 분석

AVR에서 HIGH 전압레벨이 출력될 때 최대 방출되는 전류는 다음과 같다.

Absolute Maximum Ratings*

Operating Temperature	-55°C to +125°C
Storage Temperature	-65°C to +150°C
Voltage on any Pin except $\overline{\text{RESET}}$ with respect to Ground	-0.5V to V_{CC}+0.5V
Voltage on $\overline{\text{RESET}}$ with respect to Ground	-0.5V to +13.0V
Maximum Operating Voltage	6.0V
DC Current per I/O Pin	40.0 mA
DC Current V_{CC} and GND Pins	200.0 - 400.0mA

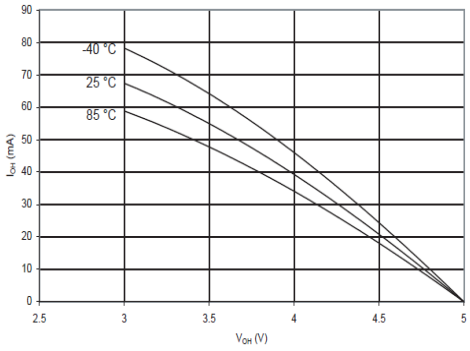
*NOTICE: Stresses beyond those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or other conditions beyond those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

[ATMEGA128 Datasheet, 최대 제한 스펙]

출력 단자의 전류 제한은 최대 40mA까지 가능하며, LED를 충분히 켜고도 남는 전류이다.
Source로 Led를 제어할 때 특성곡선은 다음과 같다.

Pin Driver Strength

Figure 182. I/O Pin Source Current vs. Output Voltage (V_{CC} = 5V)



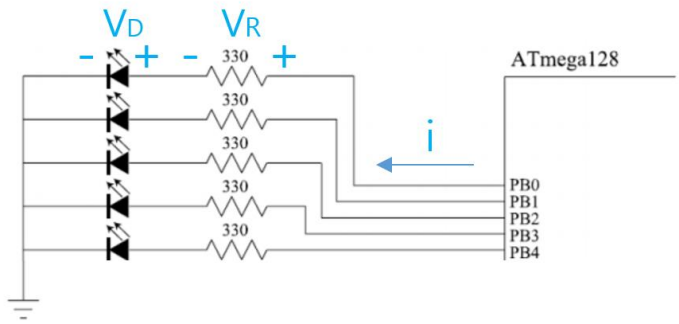
[소스 전원일 때, Source Current와 Output Voltage]

색상	구 분	최소전압	최대전압	전류(일반)	전류(최대)
적	Red	1.8V	2.3V	20 mA	50 mA
등	Orange	2.0V	2.3V	30 mA	50 mA
황	Real Yellow	2.0V	2.8V	20 mA	50 mA
초	emerald Green	1.8V	2.3V	20 mA	50 mA
초	Real Green	3.0V	3.6V	20 mA	50 mA
청	sky Blue	3.4V	3.8V	20 mA	50 mA
청	Real Blue	3.4V	3.8V	20 mA	50 mA
자	Pink	3.4V	3.8V	20 mA	50 mA
백	White	3.4V	4.0V	20 mA	50 mA

[대략적인 LED 사용 전압과 전류]

위 오른쪽 사진의 LED 관련 표는 정확하지 않지만 참고사항으로 첨부해보았다.

PB0핀에 High 전압 레벨을 출력하면 핀에서 전류가 나와 GND로 흐르면서 LED가 켜지게 된다.(Source 제어) LED가 켜지면 2V 정도의 전압에 20mA 정도 소요된다. LED에 5V를 직접 가하면 매우 밝으나 40mA에 가까운 전류가 흘러 AVR 보드에 좋지 않다. 안전을 고려하여 LED에 흐를 전류는 정격 20mA의 절반 정도인 10mA로 잡는 것이 보통이다. 10mA가 흐른다고 가정하여 계산을 해본다.



[실험 회로 일부 - DIODE 부분]

색	첫 번째 띠	두 번째 띠	세 번째 띠 (단위)	4번째 띠 (오차)	열계수
검정	0	0	$\times 10^0$		
갈색	1	1	$\times 10^1$	±1% (F)	100 ppm
빨간색	2	2	$\times 10^2$	±2% (G)	50 ppm
주황색	3	3	$\times 10^3$		15 ppm
노란색	4	4	$\times 10^4$		25 ppm
초록색	5	5	$\times 10^5$	±0.5% (D)	
파란색	6	6	$\times 10^6$	±0.25% (C)	
보라색	7	7	$\times 10^7$	±0.1% (B)	
회색	8	8	$\times 10^8$	±0.05% (A)	
흰색	9	9	$\times 10^9$		
금색			×0.1	±5% (J)	
은색			×0.01	±10% (K)	

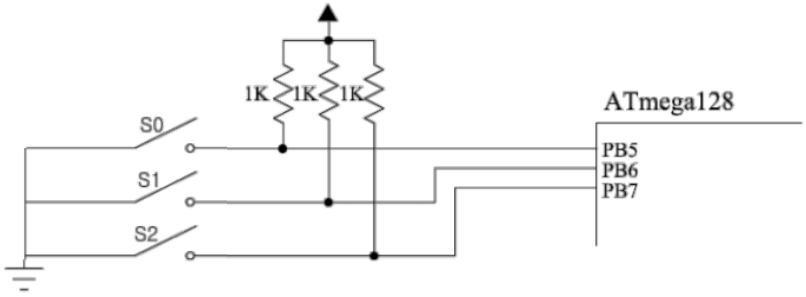
[4줄 저항띠 읽기]

PB0 PIN에서 출력전압을 V라 했을 때, 5V가 출력되므로,

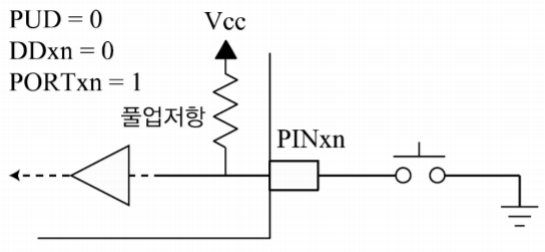
$$V - RI - V_D = 0$$
$$5 - R \times 10mA - 2 = 0$$
$$R = 300\Omega$$

$R = 300\Omega$이 되므로, 시중에 가장 많이 사용되는 LED 제어용 저항은 330Ω이 되는 것이다. 위 오른쪽 사진은 저항의 색띠에 따라 값을 읽는 방법이다. 멀티미터기로 측정하였을 때, 오차를 포함하여 가장 정확한 저항 값을 얻을 수 있지만, 실험 환경에 따라 위 표로 계산해 보았다. 실험 때 사용한 저항은 주주갈금으로, 약 330Ω을 사용하였다.

1-2. 스위치 회로 분석



[실험 회로 일부 - 풀업저항과 스위치 부분]

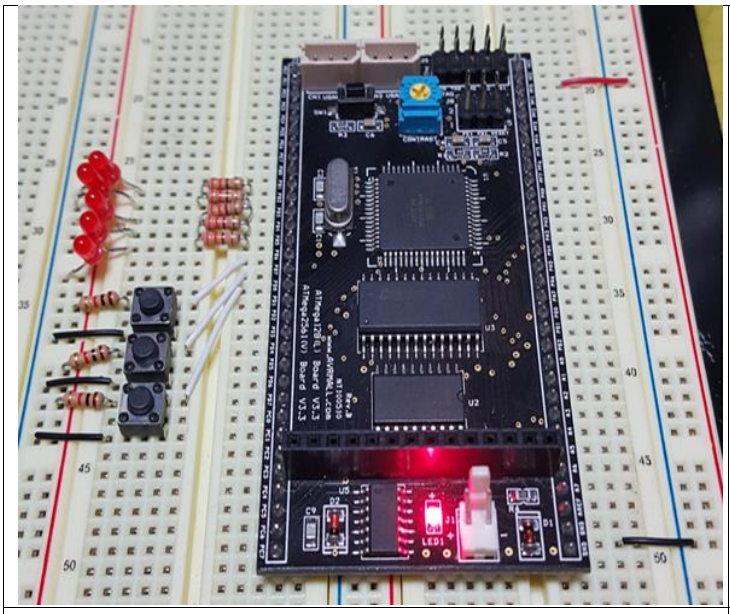


[풀업 저항의 사용]

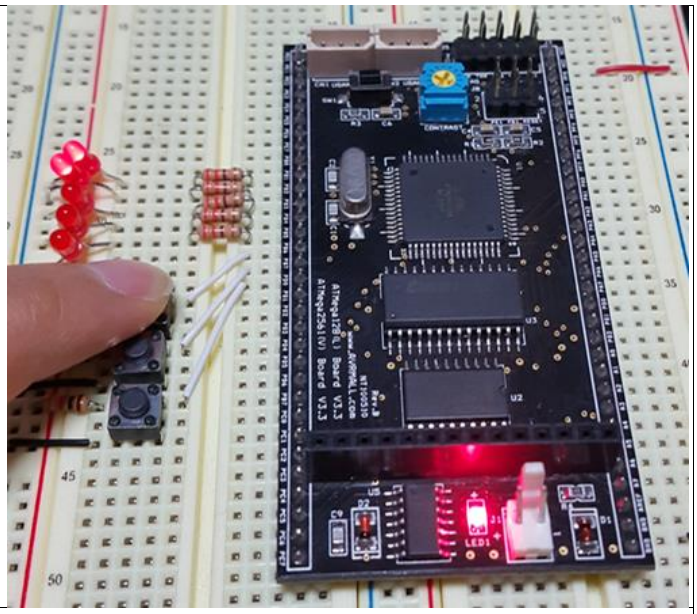
회로에서 설정된 스위치는 풀업 저항으로, 스위치를 누르면 0(LOW), 누르지 않으면 1(HIGH)가 시행된다.
결론적으로 B포트에 연결된 스위치와 LED를 이용하여 입출력 동작을 수행→ 스위치가 눌릴 때 LED ON

스위치 S0 → LED D0, D1을 ON/OFF
스위치 S1 → LED D2를 ON/OFF
스위치 S2 → LED D3을 ON/OFF

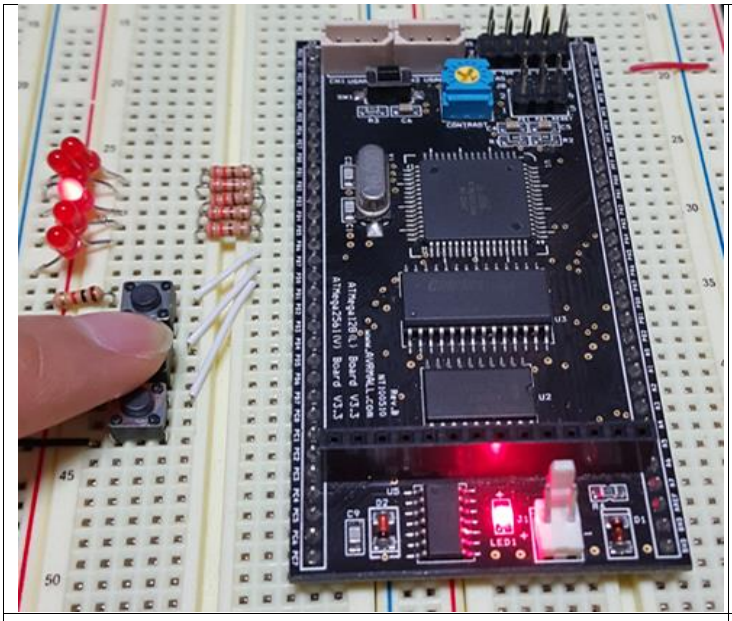
1-3. 실험 결과



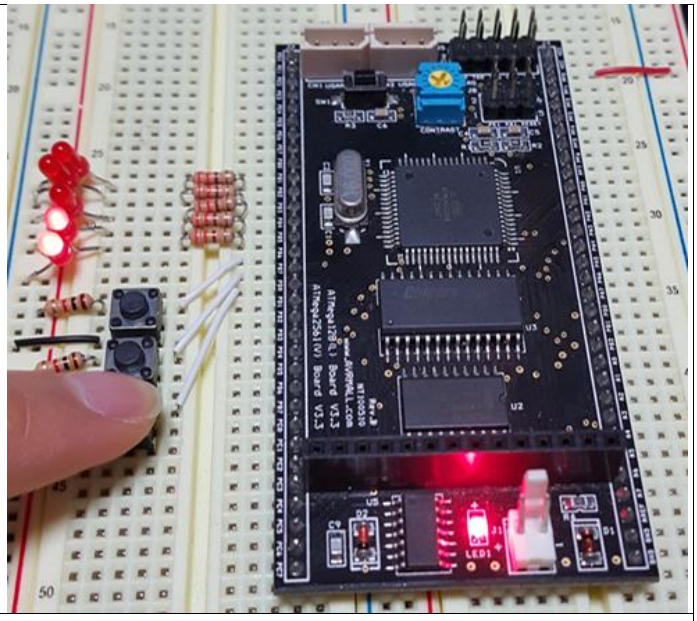
스위치를 누르지 않았을 때



1번째 스위치를 누르면
1, 2번째 스위치가 ON된다.



2번째 스위치를 누르면
3번째 스위치가 ON된다.



3번째 스위치를 누르면
4, 5번째 스위치가 ON된다.

1-4. AVR 소스 코드

레이블 정의

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

#define S0 PB5
#define S1 PB6
#define S2 PB7

#define D0 PB0
#define D1 PB1
#define D2 PB2
#define D3 PB3
#define D4 PB4
```

main 함수

```
int main(void)
{
    DDRB = 1<<D0 | 1<<D1 | 1<<D2 | 1<<D3 | 1<<D4;

    while(1){
        if(!(PINB & (1<<S0)) )
            PORTB |= 1<<D0 | 1<<D1;
        else
            PORTB &= ~(1<<D0 | 1<<D1);

        if(! (PINB &(1<<S1)))
            PORTB |= 1<<D2;
        else
            PORTB &= ~(1<<D2);

        if(! (PINB &(1<<S2)))
            PORTB |= 1<<D3 | 1<<D4;
        else
            PORTB &=~(1<<D3 | 1<<D4);
    }

    return 0;
}
```

1-5. 프로그램 분석

```
#include <avr/io.h> //AVR 기본 입출력 관련 헤더파일

#define S0 PB5 // S는 Switch, PB5 핀을 풀업 저항이 달린 스위치로 사용
#define S1 PB6
#define S2 PB7

#define D0 PB0 // D는 Diode, PB0 핀을 330옴 저항이 달린 LED로 사용
#define D1 PB1
#define D2 PB2
```

```
#define D3 PB3
#define D4 PB4
```

AVR 마이크로 컨트롤러의 포트들은 범용 디지털 입출력 포트로 사용되어질 때 Read-Modify-Write 기능이 있다. 이 기능은 다른 핀들의 값을 수정하지 않고 포트의 특정 핀만 수정할 수 있도록 한다.
다른 핀들의 방향에 영향을 주지 않고 한 포트 핀의 방향을 바꿀 수 있다.

```
int main(void) //main함수. void는 parameters(매개변수)가 갖지 않음을 의미
{
        DDRB = 1<<D0 | 1<<D1 | 1<<D2 | 1<<D3 | 1<<D4; // PB0~4까지 출력으로 지정
```

Port B Data Direction Register – DDRB

Bit	7	6	5	4	3	2	1	0	
	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	DDRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

DDRB(Data Direction Register)는 입/출력 방향 설정을 위한 레지스터로, SET(1)하면 출력이고, CLEAR(0)하면 입력으로 설정한다. 여기서는 PB0~4 PIN까지 모두 출력으로 설정하였다. PB5~7까지는 입력으로 설정된다.

```
        while(1){
          if(!(PINB & (1<<S0)) )
        // S0에 풀업 저항으로 인해 스위치를 누르면, LOW가 되고, PINB에 입력되면
```

Port B Input Pins Address – PINB

Bit	7	6	5	4	3	2	1	0	
	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	PINB
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

PINB(Input Pins Address)는 입력 핀에 해당하는 레지스터로서 입력된 값을 표시한다. S0는 PB5(스위치)를 의미하고, 스위치를 누르면 S0는 LOW(0)로 SET된다. 이것을 PINB에 입력하고, !(비트반전)을 통해 if문에는 True(1)의 값이 된다. 그렇게 if문을 시행한다.

```
                PORTB |= 1<<D0 | 1<<D1;
                // PB0과 PB1의 PIN에 PORTB 레지스터로 HIGH를 내보낸다.
                //첫번째와 두번째 다이오드가 켜진다.
```

Port B Data Register – PORTB

Bit	7	6	5	4	3	2	1	0	
	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	PORTB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

if문을 시행하면 위의 PORTB에 대한 소스코드가 있다. PORTB(Data Register)는 출력용 데이터 값을 위한 레지스터이다. D0(첫 번째다이오드)는 PB0이고, 출력으로 HIGH(1)를 내보낸다. D1(두 번째 다이오드) 또한 PB1이고, 출력으로 HIGH(1)를 내보낸다. |(OR) 연산자를 시행하여, 각각의 PB0, PB1 비트의 OR 연산을 PORTB에 OR마스크를 씌워 True(1)가 되게 한다. 결국 출력으로 HIGH(1)가 나와, LED가 ON된다. 아래의 나머지 소스 코드들도 마찬가지로 시행된다.

```
        else // 첫번째 스위치를 누르지 않으면
                PORTB &= ~(1<<D0 | 1<<D1); // 첫번째, 두번째 다이오드를 AND MASK를 씌워 OFF 시킨다.
```

```
        if(! (PINB &(1<<S1))) // S1에 풀업 저항으로 인해 스위치를 누르면, LOW가 되고, PINB에 입력되면
                PORTB |= 1<<D2; // PB2의 PIN에 PORTB 레지스터로 HIGH를 내보낸다.
                                //세번째 다이오드가 켜진다.
        else // 두번째 스위치를 누르지 않으면
                PORTB &= ~(1<<D2); // 세번째 다이오드를 AND MASK를 씌워 OFF 시킨다.

        if(! (PINB &(1<<S2))) // S2에 풀업 저항으로 인해 스위치를 누르면, LOW가 되고, PINB에 입력되면
                PORTB |= 1<<D3 | 1<<D4; // PB3과 PB4의 PIN에 PORTB 레지스터로 HIGH를 내보낸다.
                                        //네번째, 다섯번째 다이오드가 켜진다.

        else // 세번째 스위치를 누르지 않으면
                PORTB &=~(1<<D3 | 1<<D4); // 네번째, 다섯번째 다이오드를 AND MASK를 씌워 OFF 시킨
다.
        }

        return 0; // main함수가 int형이기 때문에 0을 반환하면서 종료
}
```

2. 추가실험

인터럽트를 사용한 LED 켜기

2-0. 인터럽트에 관하여

실험에 앞서 기본적으로 알아야할 인터럽트 개념에 대해 알아본다.

(1) 인터럽트?

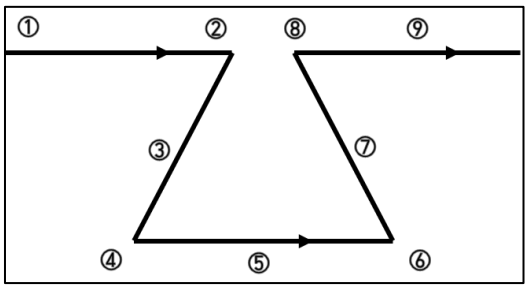
마이크로 프로세서가 프로그램을 마이크로프로세서(CPU)가 프로그램 실행하고 있을 때, 예외상황이 발생하여 처리가 필요할 경우에 마이크로프로세서에게 알려 우선적으로 처리할 수 있도록 하는 것

(2) 인터럽트 처리과정

① 주 프로그램 실행

② 인터럽트 발생

인터럽트가 검출되면 외부 인터럽트 플랙 레지스터의 플랙이 SET(1) 된다. 해당 인터럽트의 벡터가 인출되어 인터럽트 서비스 루틴의 실행이 일어나면 자동으로 0으로 클리어 된다.



[인터럽트 처리과정]

External Interrupt Flag Register – EIFR

Bit	7	6	5	4	3	2	1	0	
	INTF7	INTF6	INTF5	INTF4	INTF3	INTF2	INTF1	IINTF0	EIFR
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

그리고 각 인터럽트에 해당하는 인터럽트 마스크 레지스터와 SREG 레지스터의 글로벌 인터럽트 허용 비트(최상위 비트, I)를 보고 인터럽트 허용 여부를 판단한다.

External Interrupt Mask Register – EIMSK

Bit	7	6	5	4	3	2	1	0	
	INT7	INT6	INT5	INT4	INT3	INT2	INT1	IINT0	EIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

The AVR status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

각 인터럽트 별로 인터럽트 벡터의 번지가 미리 정해져 있기 때문에 그것에 따라 인터럽트 서비스 루틴의 시작 번지나 그곳으로의 점프 명령을 저장해 두어야 한다. 아래의 (4) 인터럽트 우선순위 첨부 사진의 Address를 참고하면 된다.

③ 복귀주소 저장

인터럽트 서비스 루틴을 종료하고 되돌아 올 수 있도록 복귀주소(복귀할 Program Counter값)을 스택에 저장해 놓는다.

④ 인터럽트 벡터로 점프

해당 인터럽트 서비스 루틴으로 점프하여 프로그램을 실행한다.

⑤ 인터럽트 처리

⑥ 인터럽트 처리완료

⑦ 복귀주소 로드

인터럽트 서비스 루틴의 실행 중에 리턴 명령을 만나면 스택에서 복귀 주소를 되찾아 프로그램 카운터에 로드함으로써 원래의 위치로 찾아온다. RETI 명령을 만나면 해당 인터럽트를 종료하고 주프로그램으로 복귀한다.

⑧ 마지막에 실행되던 주소로 점프

⑨ 주 프로그램 실행

[인터럽트 종류]

Interrupt Vectors in ATmega128

Table 23. Reset and Interrupt Vectors

Vector No.	Program Address(2)	Source	Interrupt Definition
1	$0000(1)	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, and JTAG AVR Reset
2	$0002	INT0	External Interrupt Request 0
3	$0004	INT1	External Interrupt Request 1
4	$0006	INT2	External Interrupt Request 2
5	$0008	INT3	External Interrupt Request 3
6	$000A	INT4	External Interrupt Request 4
7	$000C	INT5	External Interrupt Request 5
8	$000E	INT6	External Interrupt Request 6
9	$0010	INT7	External Interrupt Request 7
10	$0012	TIMER2 COMP	Timer/Counter2 Compare Match
11	$0014	TIMER2 OVF	Timer/Counter2 Overflow
12	$0016	TIMER1 CAPT	Timer/Counter1 Capture Event
13	$0018	TIMER1 COMPA	Timer/Counter1 Compare Match A
14	$001A	TIMER1 COMPB	Timer/Counter1 Compare Match B
15	$001C	TIMER1 OVF	Timer/Counter1 Overflow
16	$001E	TIMER0 COMP	Timer/Counter0 Compare Match
17	$0020	TIMER0 OVF	Timer/Counter0 Overflow
18	$0022	SPI, STC	SPI Serial Transfer Complete
19	$0024	USART0, RX	USART0, Rx Complete
20	$0026	USART0, UDRE	USART0 Data Register Empty
21	$0028	USART0, TX	USART0, Tx Complete
22	$002A	ADC	ADC Conversion Complete
23	$002C	EE READY	EEPROM Ready
24	$002E	ANALOG COMP	Analog Comparator
25	$0030(3)	TIMER1 COMPC	Timer/Countre1 Compare Match C
26	$0032(3)	TIMER3 CAPT	Timer/Counter3 Capture Event
27	$0034(3)	TIMER3 COMPA	Timer/Counter3 Compare Match A
28	$0036(3)	TIMER3 COMPB	Timer/Counter3 Compare Match B
29	$0038(3)	TIMER3 COMPC	Timer/Counter3 Compare Match C
30	$003A(3)	TIMER3 OVF	Timer/Counter3 Overflow

[인터럽트 우선순위]

(3) 인터럽트 종류

ATmega128에는 내부 인터럽트, 차단불가능 인터럽트 NMI가 없다. 조사형 인터럽트는 사용하지 않는다. 위 왼쪽 그림의 빨간색 네모가 실험에 사용했던 해당하는 사항이다.

(4) 인터럽트 우선순위

인터럽트의 요청이 중복될 수도 있기 때문에 각 인터럽트 사이에 우선순위가 정해져 있다. 동시에 인터럽트가 발생할 경우, 우선 순위가 높은 인터럽트가 먼저 실행된다. 위 오른쪽 그림은 우선 순위에 따라 나열한 표이다.

(5) 인터럽트 방식

인터럽트를 사용하기 전에, 우선적으로 인터럽트가 어떻게 걸릴지 방식을 설정해놓아야 한다.

External Interrupt Control Register A – EICRA

Bit	7	6	5	4	3	2	1	0	
	ISC31	ISC30	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

External Interrupt Control Register B – EICRB

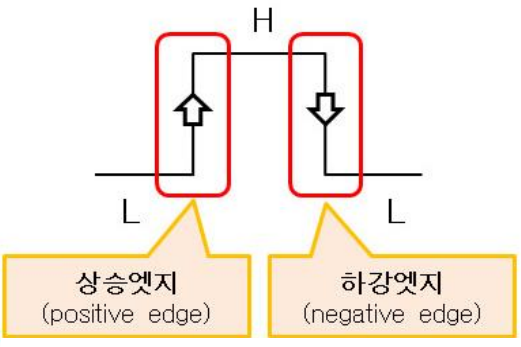
Bit	7	6	5	4	3	2	1	0	
	ISC71	ISC70	ISC61	ISC60	ISC51	ISC50	ISC41	ISC40	EICRB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

EICRA 는 외부 인터럽트 중 0~3번 인터럽트까지의 트리거 방식을 LOW/상승 엣지/ 하강 엣지중에 선택한다. EICRB는 외부 인터럽트 중 4~7번까지의 트리거 방식을 LOW/ 상승 엣지/ 하강 엣지중에 선택한다. 다음의 표와 같다.

EICRA			EICRB		
ISCn1	ISCn0	인터럽트 트리거 방식	ISCn1	ISCn0	인터럽트 트리거 방식
0	0	Low Level 이 인터럽트 트리거	0	0	Low Level 이 인터럽트 트리거
0	1	RESERVED	0	1	Falling or Rising Edge 가 인터럽트 트리거
1	0	Falling Edge 가 인터럽트 트리거	1	0	Falling Edge 가 인터럽트 트리거
1	1	Rising Edge 가 인터럽트 트리거	1	1	Rising Edge 가 인터럽트 트리거

인터럽트를 사용할 때는 이러한 방식 설정이 대단히 중요하다. 예를 들어 스위치를 풀업 저항으로 사용할 경우, 스위치를 누르면 LOW 레벨이 된다. 하강 엣지가 될 때 스위치를 누름과 같다. 풀업 저항으로 연결하여 스위치를 사용할 경우에는 인터럽트 트리거 방식에서 하강 엣지를 사용하는 것이 프로그램과 사용자가 이해하기에 좋은 프로그램이 된다.

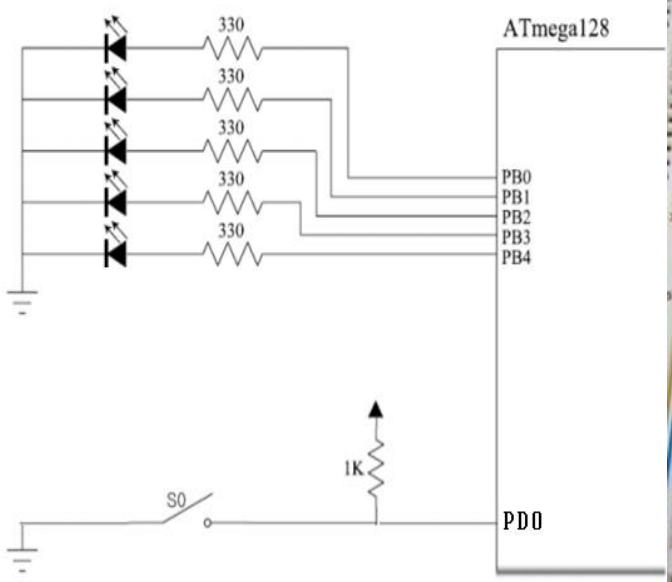
INT0을 사용할 경우, EICRA를 통해 방식을 설정한다. 위 표에서 ISC01, ISC00이라 생각하면 된다. 위 비트를 1과 0으로 SET할 시에 원하는 Falling Edge 인터럽트가 트리거 된다.

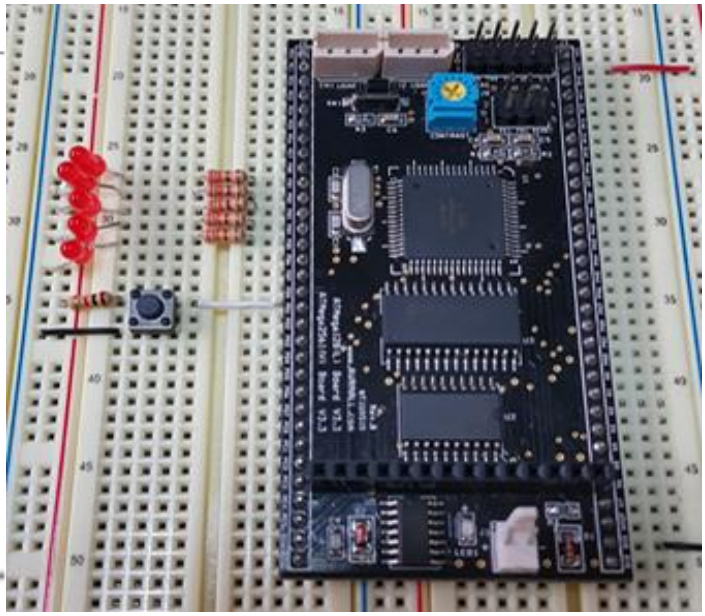


[인터럽트 방식]

2-1. 실험 회로

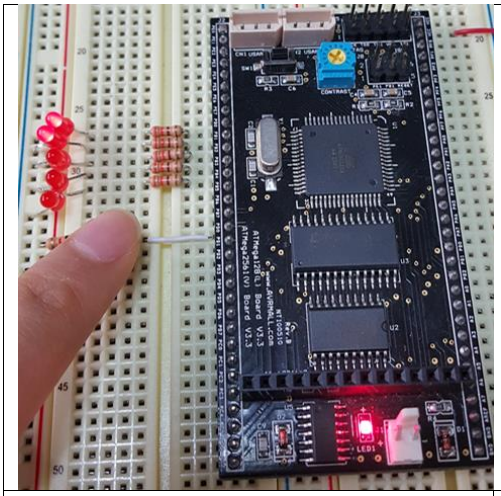
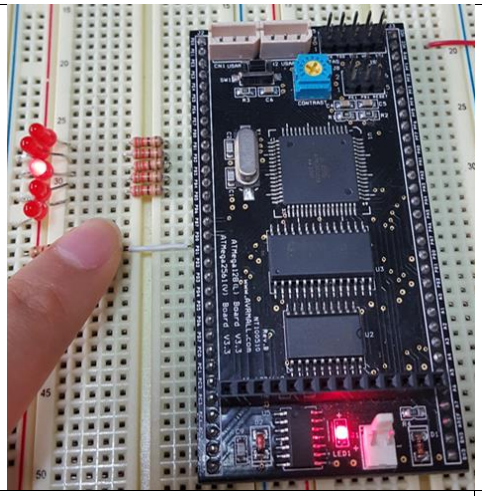
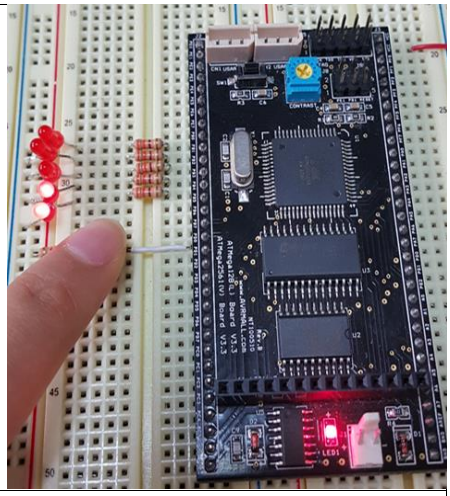
아래의 실험 회로는 스위치를 누를 때마다 인터럽트가 걸려 인터럽트 서비스 루틴에 따라 LED가 순서대로 켜지는 회로이다.





그림에서는 풀업 저항으로 1K가 설정되었지만, 실험에서는 330옴 저항을 사용.
회로의 나머지는 동일하며, 풀업 저항의 스위치가 사용된 포트는 PORT D로 변경하였다.

2-2. 실험 결과

		
처음 스위치를 누르면 1, 2번째 LED ON	다시 스위치를 누르면 3번째 LED ON	다시 스위치를 누르면 4, 5번째 LED ON

스위치를 반복적으로 눌러주면 LED가 1, 2번째 → 3번째 → 4, 5번째 순으로 켜졌다 꺼졌다 반복된다.

2-3. AVR 소스 코드

레이블 정의 및 함수 정의

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

#define S0 PD0

#define D0 PB0
#define D1 PB1
#define D2 PB2
#define D3 PB3
#define D4 PB4

volatile int state = 0;

ISR(INT0_vect)
{
    state++;
    if(state>3) state=1;
}

void INIT_PORT(void)
{
 DDRB = 1<<D0 | 1<<D1 | 1<<D2 | 1<<D3 | 1<<D4;

 DDRD = 0<<S0;
 PORTD = 1<<S0;
}

void INIT_INT0(void)
{
    EIMSK |= (1<<INT0);
    EICRA |= (1<<ISC01);
    sei();
}
```

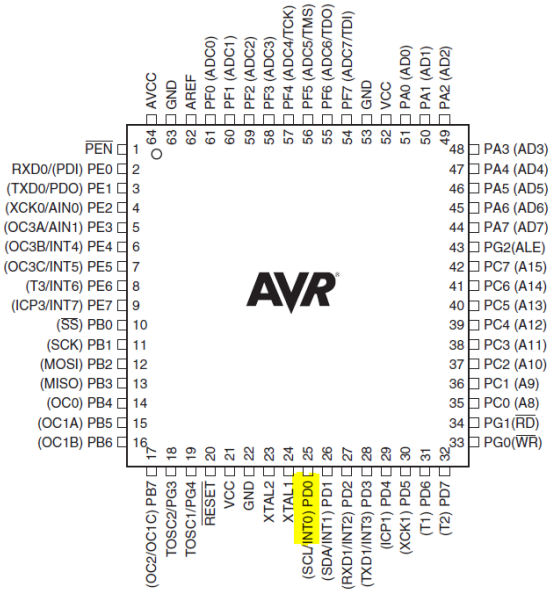
main 함수

```
int main(void)
{
    INIT_PORT();
    INIT_INT0();

    while(1){

        if(state==1)    {
            PORTB &=~(1<<D3 | 1<<D4);
            PORTB |= 1<<D0 | 1<<D1;
}
        else if(state==2){
            PORTB &=~(1<<D0 | 1<<D1);
            PORTB |= 1<<D2;
}
        else {
            PORTB &=~(1<<D2);
            PORTB |= 1<<D3 | 1<<D4;
        }
    }
    return 0;
}
```

2-4. 프로그램 분석

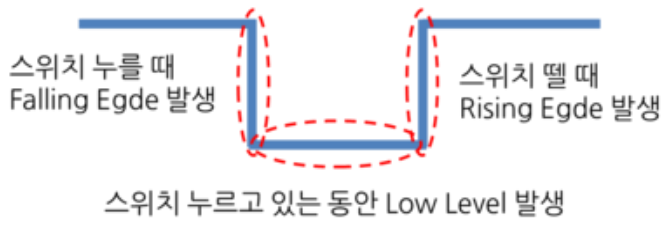
#include <avr/io.h> //AVR 기본 입출력 관련 헤더파일
#include <avr/interrupt.h> // 인터럽트 관련 헤더파일

#define S0 PD0
// S는 Switch, PD0 핀을 풀업 저항이 달린 스위치로 사용
 //인터럽트 사용 핀

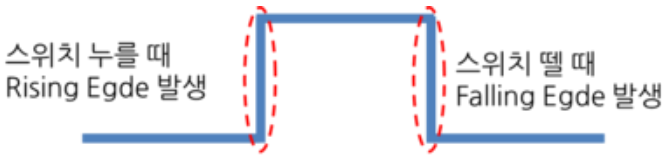


인터럽트를 사용하기 위해 INT0 PIN이 있는 PD0를 사용하였다.

[풀 업 방식의 스위치]



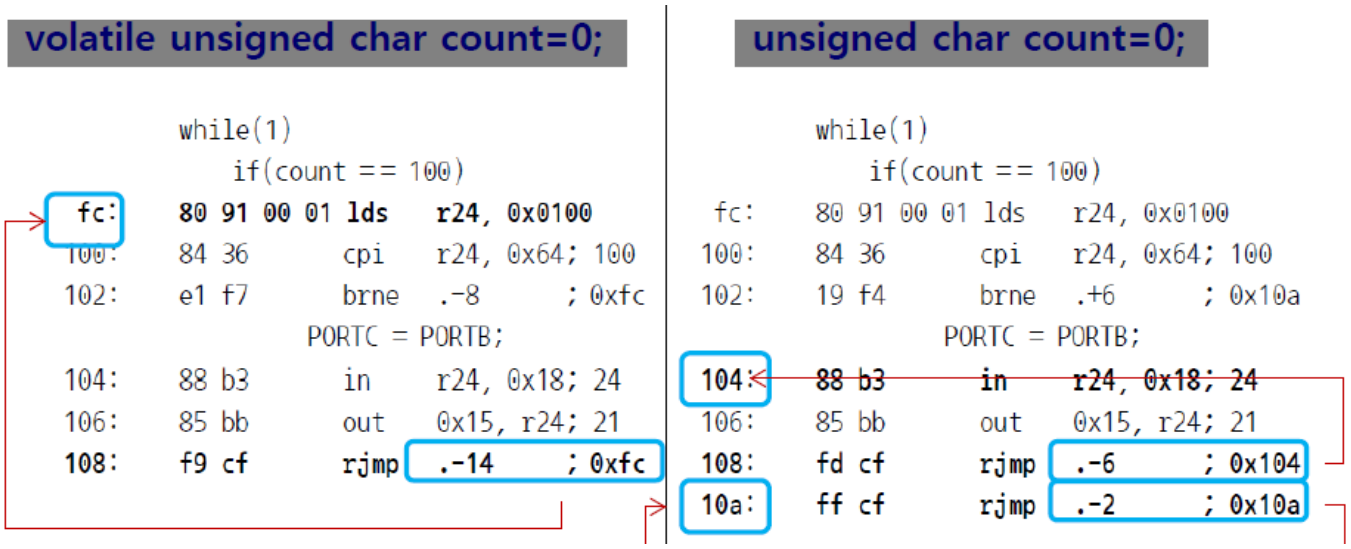
[풀 다운 방식의 스위치]



인터럽트 방식(상승 엣지, 하강 엣지, LOW 트리거 설정)을 결정할 때, 스위치가 풀업 저항을 사용하였는가, 풀다운 저항을 사용하였는 가는 아주 중요한 사항이다.

```
#define D0 PB0 // D는 Diode, PB0 핀을 330옴 저항이 달린 LED로 사용
#define D1 PB1
#define D2 PB2
#define D3 PB3
#define D4 PB4

volatile int state = 0; // 인터럽트 사용을 위한 Volatile 변수
```

인터럽트를 사용할 때는 프로그램 최적화를 피해서 하여야 한다. 일반 변수들은 프로그램 최적화로 적절한 주소값을 못 찾아갈 수 있기 때문이다. 다음의 강의 자료로 참고 자료를 함께 첨부한다.




```
ISR(INT0_vect) //Interrupt Service Routine: 인터럽트가 호출되었을 시 실행되는 루틴
{
        state++; //state로 어떤 LED를 켤지 결정
        if(state>3) state=1; //state가 3이상이면 다시 1로  SET
}

void INIT_PORT(void) //PORT 초기화 함수
{
//PORT B의 0~4번 PIN을 다이오드로 사용하기 위한 출력으로 설정
 DDRB = 1<<D0 | 1<<D1 | 1<<D2 | 1<<D3 | 1<<D4;
```

```
 DDRD = 0<<S0; //PORT D를 인터럽트를 사용한 스위치로, PD0 PIN을 입력으로 설정
 PORTD = 1<<S0; // PD0 PIN의 풀업 저항 사용
}
```

위의 포트에 관한 초기화 부분은 앞의 본 실험에서 설명해 놓았기 때문에 생략한다.

```
void INIT_INT0(void) //인터럽트 초기화 함수
{
        EIMSK |= (1<<INT0); //INT0 사용, 인터럽트 마스크 레지스터 : 인터럽트 활성화
        EICRA |= (1<<ISC01); //인터럽트 컨트롤 레지스터 : 하강 엣지에서 인터럽트 발생
        sei(); // 전역적으로 인터럽트 허용, SREG |= 0x80;
}
```

External Interrupt Mask Register – EIMSK

Bit	7	6	5	4	3	2	1	0	
	INT7	INT6	INT5	INT4	INT3	INT2	INT1	IINT0	EIMSK
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

인터럽트0(INT0)을 사용하였기 때문에 외부 인터럽트 마스크 레지스터의 0번 비트 INT0가 SET(1)로 된다.

External Interrupt Control Register A – EICRA

Bit	7	6	5	4	3	2	1	0	
	ISC31	ISC30	ISC21	ISC20	ISC11	ISC10	ISC01	ISC00	EICRA
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

다음 EICRA의 0번, 1번 PIN은 인터럽트0(INT0)의 인터럽트가 걸리는 방식을 설정한다. ISC01을 1로, ISC00을 0으로 SET했기 때문에 Falling Edge에서 인터럽트가 트리거된다.

EICRA		
ISCn1	ISCn0	인터럽트 트리거 방식
0	0	Low Level 이 인터럽트 트리거
0	1	RESERVED
1	0	Falling Edge 가 인터럽트 트리거
1	1	Rising Edge 가 인터럽트 트리거

즉 풀업 저항이 달린 스위치를 사용하였기 때문에, 스위치를 누르면 LOW가 됨을 감안하였다. LOW가 될 때를 하강 엣지로 하여, 스위치를 누르면 인터럽트가 작동하게 만들었다.

인터럽트 초기화 함수를 설정할 때, 풀 업 저항으로 사용하는 것을 꼭 알려줘야 할까? PORT D가 입력이 바뀌면 인터럽트가 일어나게 하면 되지 않을까? 이에 대한 답은 아니다.

인터럽트일수록 더욱 많은 정보를 알려줘야 한다. 인터럽트는 rising edge와 falling edge와 같이 신호 변화로 작동한다. 이에 따라 풀 업, 풀 다운에 따라서 작동 방법이 바뀌기 때문에 더욱 중요한 정보가 된다. 이는 컴퓨터에게도, 우리가 알도록 하는 것에도 둘다 중요하다.

The AVR status Register – SREG – is defined as:

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

마지막으로 인터럽트 초기화 함수에는 sei(); 가 들어간다. sei()는 Set Interrupt Flag로 위의 SREG의 최상위 비트, I비트를 1로 SET하여 전체 인터럽트를 허용한다.

```
int main(void) //main함수. void는 parameters(매개변수)가 갖지 않음을 의미
{
	INIT_PORT(); //포트 설정
	INIT_INT0(); //INT0 인터럽트 설정

	while(1){
```

```
ISR(INT0_vect)
{
    state++;
    if(state>3) state=1;
}
```

[추가 실험 소스 코드 중 일부분, 인터럽트 서비스 루틴 소스]

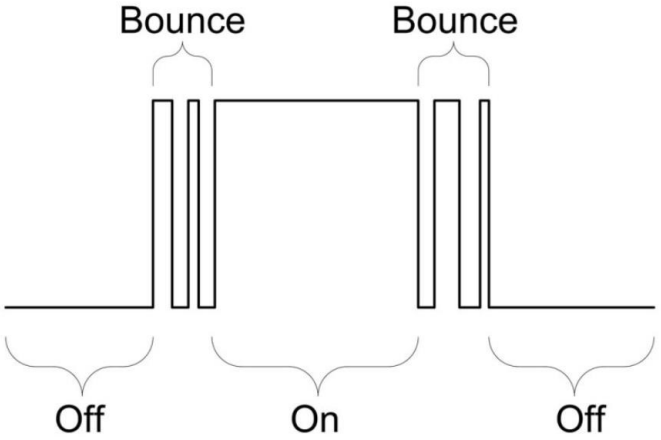
인터럽트 서비스 루틴에서 스위치를 누를 때마다 state가 증가하게 만들었다. state가 3 이상이 되면, 다시 1로 되어 3번마다 스위치를 돌아가면서 작동하도록 하였다.
스위치를 첫번째로 누르면 첫번째 if문이,
스위치를 두번째로 누르면 두번째 else if문이,
스위치를 세번째로 누르면 세번째 else문이 작동한다.
돌아가면서 작동하면서 다음 LED가 켜질 때, 이전의 LED가 꺼지게끔 소스를 작성하였다.
다음의 소스에서 동작이 어떻게 이루어지는지는 이전의 본 실험에서 자세히 설명을 해놓았기에 생략한다. 동작은 주석처리로 간단히 표시해보았다.

```
		if(state==1){
			PORTB &=~(1<<D3 | 1<<D4); // 4, 5번째 LED 끄기
			PORTB |= 1<<D0 | 1<<D1; // 1, 2번째 LED 켜기
		}
		else if(state==2){
			PORTB &=~(1<<D0 | 1<<D1); // 1, 2번째 LED 끄기
			PORTB |= 1<<D2; // 3번째 LED 켜기
		}
		else {
			PORTB &=~(1<<D2); //3번째 LED 끄기
				PORTB |= 1<<D3 | 1<<D4; // 4, 5번째 LED 켜기
		}

	}
	return 0; // main함수가 int형이기 때문에 0을 반환하면서 종료
}
```

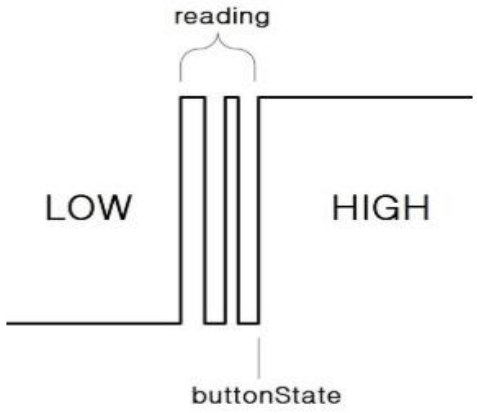
스위치를 풀업 저항으로 만들어 놓았지만, 채터링 현상이 발생하여, 한번 누를 때 다음 단계를 건너뛰고 그다음 LED로 켜지는 현상이 있기도 하였다.

-채터링[chattering]이란?
전자 회로 내의 스위치 접점이 닫히거나 열리는 순간에 기계적인 진동에 의해 매우 짧은 시간안에 스위치가 붙었다가 떨어지는 것을 반복하는 현상

이를 해결하는 과정을 **디바운스[debounce]**라고 한다. 디바운스는 짧은 시간에 여러 번 스위치 상태를 확인하는 방법을 의미한다.
이는 소프트웨어적으로 프로그램을 짤 때, <util/delay.h> 딜레이와 관련한 헤더파일을 추가하여, 딜레이를 이용하면 쉽게 해결할 수 있다. 예를 들어 50ms 동안 스위치의 동작 상태 변화가 없다면 현재 스위치 상태를 저장하는 방법으로 해결할 수 있다. 채터링 현상이 심하다면, 50ms의 검사시간을 더욱 늘려서 테스트해본다.
하드웨어적으로 채터링 현상을 완화하는 방법은 i/o 핀과 GND 핀에 커패시터를 끼워줌으로써 해결할 수 있다. 가장 큰 해결책은 스위치를 토글 스위치로 바꾸는 것이다.



[스위치가 붙을 때와 떨어질 때의 채터링 현상]



[디바운스 과정]