

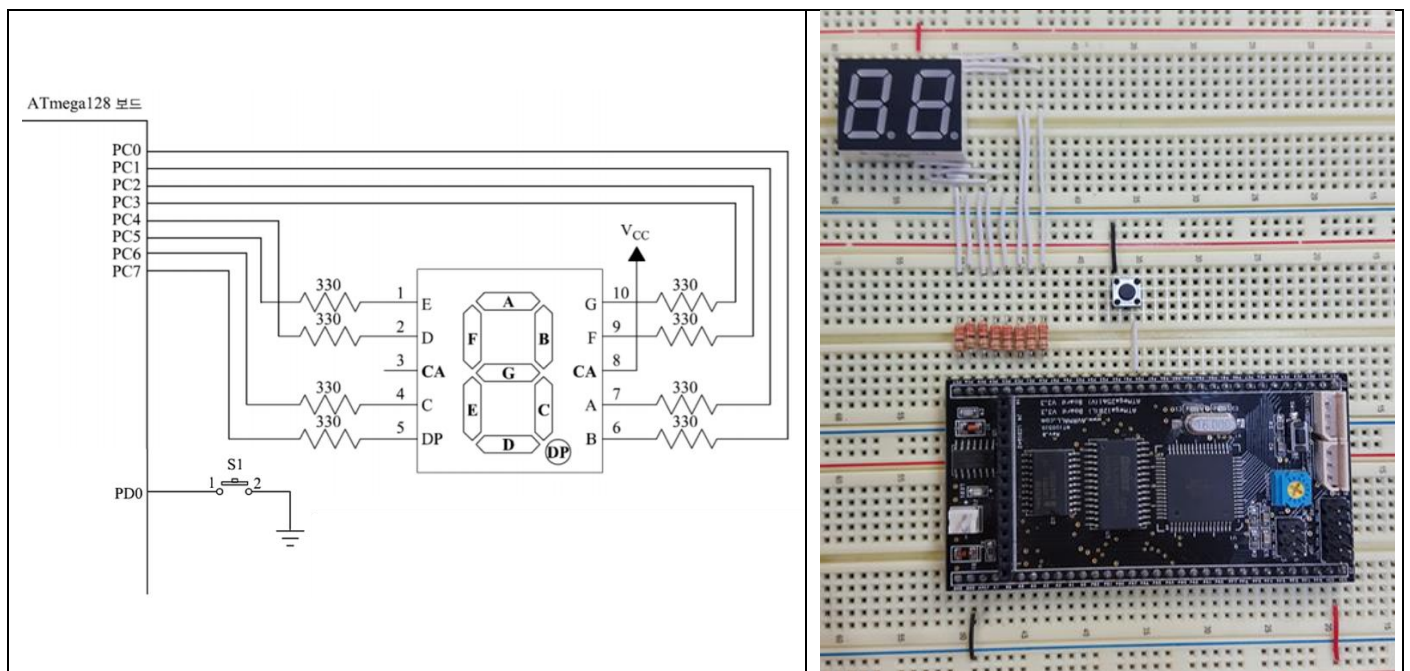
실험 2. 스위치 디바운싱

전자공학과 21611591 김난희

실험 목적

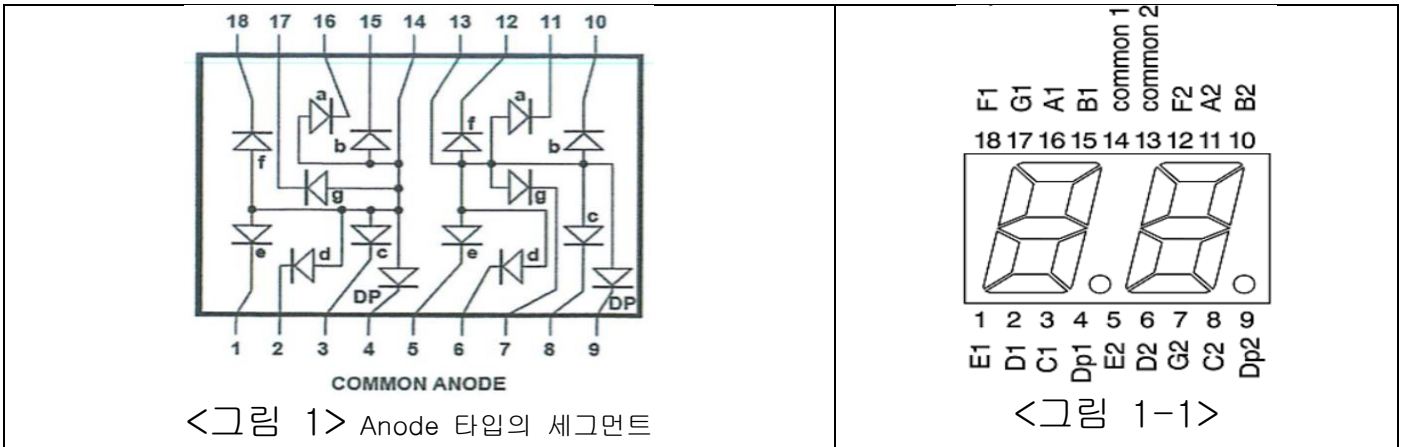
0. 세그먼트의 동작, 내부 풀업 스위치의 동작을 이해한다.
1. 스위치 바운싱 현상과 디바운싱 현상에 대해 이해한다.
2. 시간 지연 함수를 이용하여 스위치 바운싱 현상에 의한 오동작을 확인한다.
3. 시간 지연을 이용하여 스위치 디바운싱 현상을 확인한다.
4. 채터링 현상이 실제 어느정도의 시간동안 발생하는지 오실로스코프로 관찰해본다.
5. 본 실험의 추가 실험으로, 다양한 소스 코드 변화로 디바운싱 현상이 잘 동작하는지 확인한다.
6. 하드웨어적 디바운싱 회로를 꾸려, 잘 동작하는지 확인한다.
7. 내부 풀업 스위치를 잘 이해하고 있는지 확인한다.

1. 실험 회로



Common Anode 타입의 7-Segment를 사용하였기 때문에 공통 단자는 Vcc를 연결해준다. PD0 핀에 연결된 스위치는 내부 풀업 저항으로 프로그램 소스 코드로 설정해준다.

1-1. 세그먼트 회로 분석



<그림 1>과 같이 7세그먼트에는 하나의 숫자를 표시하기 위해 A,B,C,D,E,F,G,DP 총 8개의 led를 사용했다. 실험에는 2개가 묶어진 7세그먼트를 사용하였다. 각각에 공통 단자에 5V를 공급하고 LED 소자가 타지않게 하기 위해 저항 330옴을 사용하였다. 이는 실험 1에서 LED를 스위치로 ON, OFF하는 실험의 LED가 여러 개 합쳐진 회로와 같다. 실험 1에서 저항이 왜 330옴이 달리는지 자세하게 분석했기 때문에 이 부분은 생략한다.

1. ELECTRO—OPTICAL CHARACTERISTICS 光电特性：

SYMBOL 符 号	PARAMETER 项 目	TEST CONDITION 测 试 条 件	TYP 标准值	MAX 最大值	UNIT 单 位
V _F	Forward Voltage, Per Segment 正向压降	I _F =20mA	1.8	2.2	V
I _R	Reverse Current, Per Segment 反向漏电流	V _R =5V		50	uA

<그림 2> S-5263ASR1 datasheet

2. ABSOLUTE MAXIMUM RATING 其他参数：(Ta=25℃)

SYMBOL 符号	PARAMETER 项目	RED 高亮红	UNIT 单位
P _{AD}	Power Dissipation Per Segment 功 耗	100	mw
V _R	Reverse Voltage Per Segment 反 向 耐 压	5	V
I _{AF}	Continuous Forward Current Per Segment 最大使用电流	30	mA
I _{PE}	Peak Forward Current Per Segment (Duty=0.1, 1KHz) 峰 值 电 流	200	mA

<그림 2-1> S-5263ASR1 datasheet

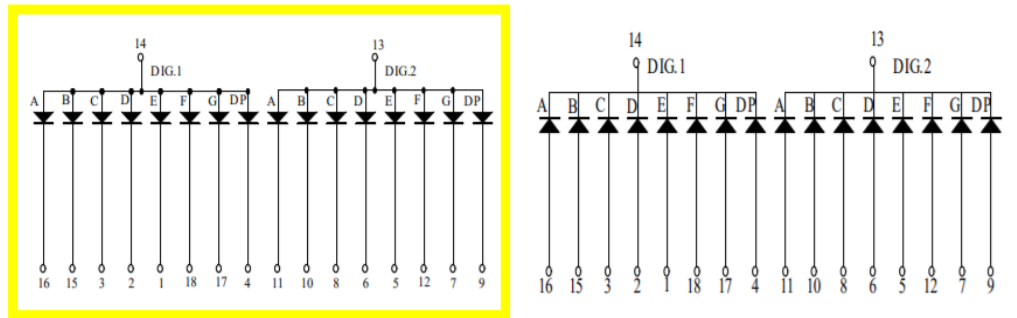
<그림 2>와 <그림 2-1>는 실험에서 사용한 S-5263ASR1(7세그먼트)의 data sheet 이다. <그림 2>는 7세그먼트의 전기적 특성이다. 20mA 전류를 테스트로 흘려주었을 때, 최대 2.2V까지 동작한다. 보통 1.8V가 적절하다. <그림 2-1>은 최대 동작을 확인할 수 있는 특성으로, 한 세그먼트당 파워소모는 최대 100mW이다.



- 색상:Red(빨강)
- 타입:Anode
- 사이즈: 25mm x 19mm

<그림 3>

S-5263ASR1/S-5263CSR1



<그림 3-1> S-5263ASR1 datasheet

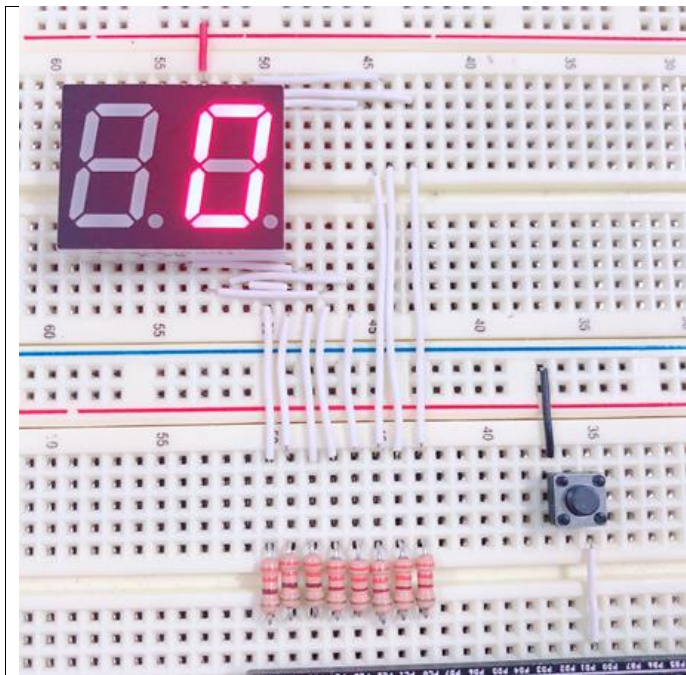
소자에 적힌 글자 중 A가 적힌 것을 보아 Anode 형의 7 segment 임을 확인했다. <그림3-1>의 S-5263ASR1 datasheet를 보아서도 쉽게 확인할 수 있었다.

(보충 공부)

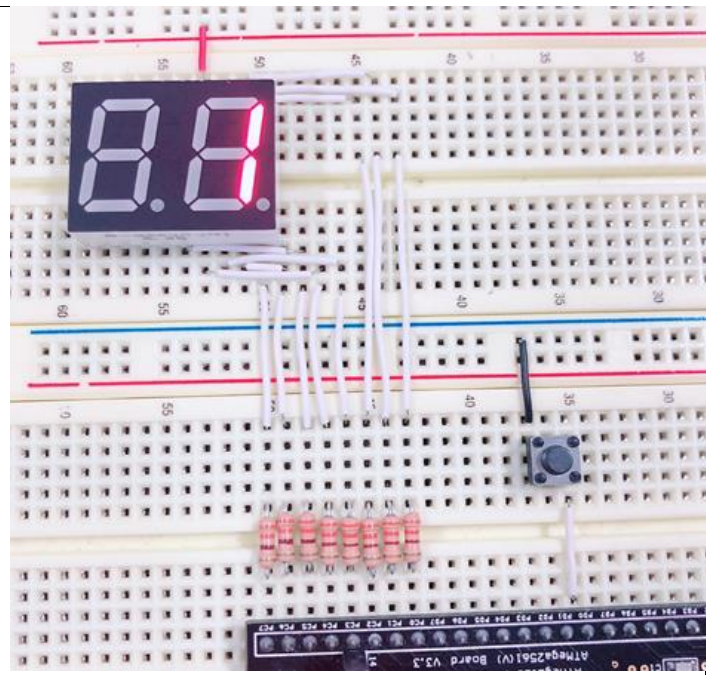
7 세그먼트는 숫자나 문자를 나타낼 수 있는 표시장치로 FND(Flexible Numeric Display)라고도 한다. 이러한 세그먼트를 여러 개 켜기 위해서는 dynamic 구동과 Static 구동 방식이 있다.

Dynamic 구동 방식은 세그먼트 전체를 한번에 켜지 않고, 순차적으로 매우 빠르게 점등한다. Led의 잔상 효과로 인해 우리 눈에는 모두 한꺼번에 켜져 있는 것처럼 보인다. Static 구동 방식은 한번에 전체를 켜는 방식이다. dynamic 구동 방식은 static 구동 방식이 비해 회로가 조금 더 복잡하지만, 소비 전력이 더욱 적고 수명도 길어지는 장점이 있다.

1-2. 실험 결과

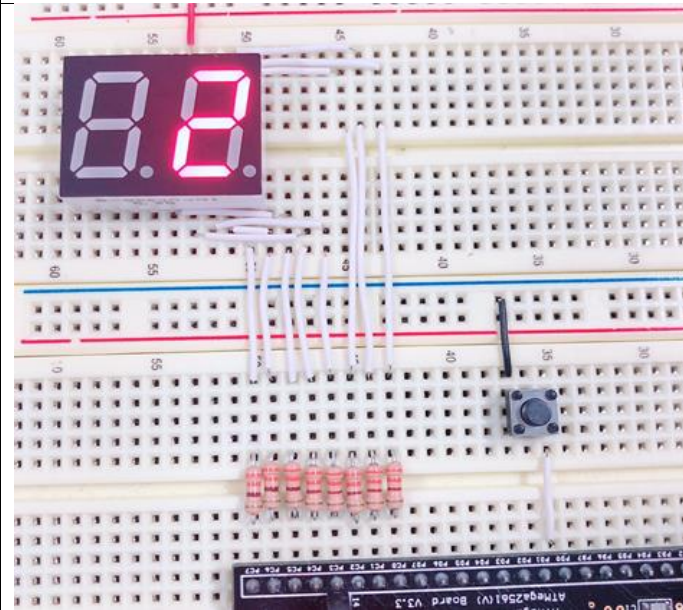


처음 시작한 상태,

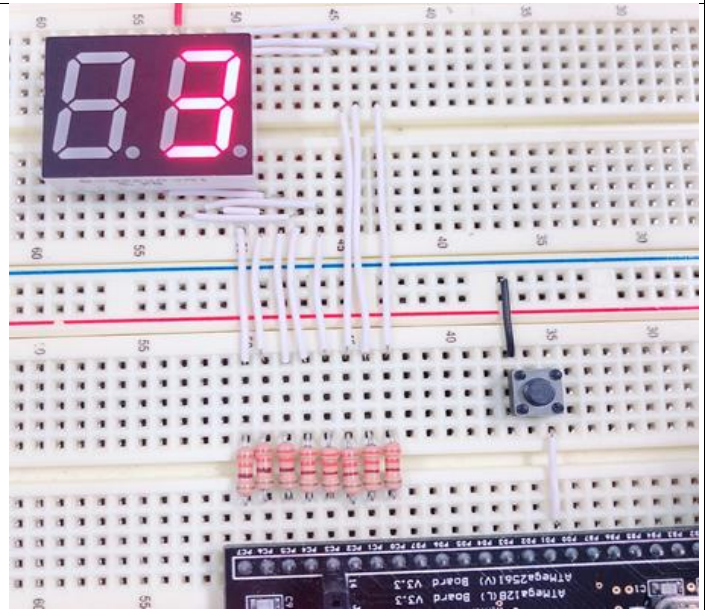


스위치를 1번째 누르면

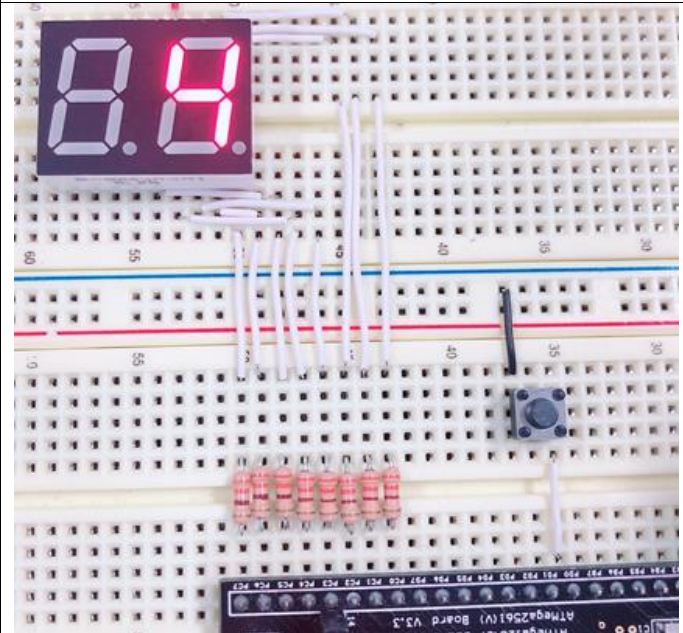
스위치를 누르지 않았을 때



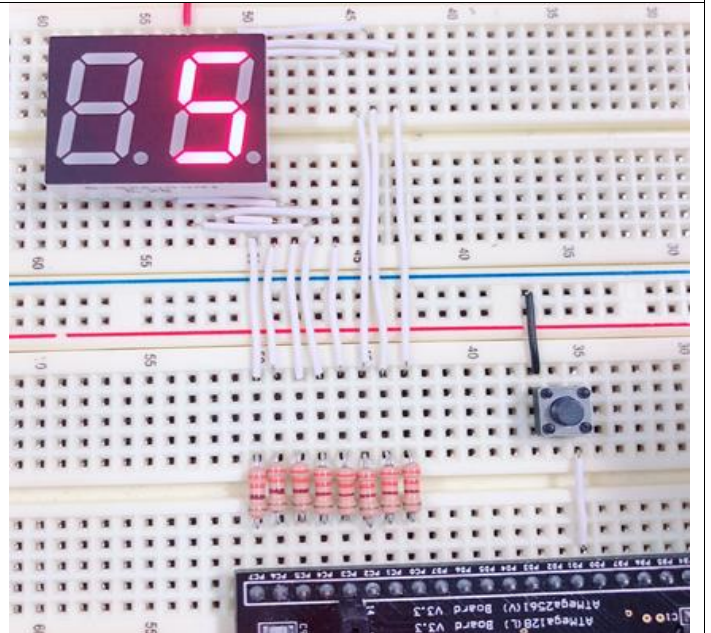
7 segment에 숫자 1이 표시된다.



스위치를 2번째 누르면
7 segment에 숫자 2가 표시된다.

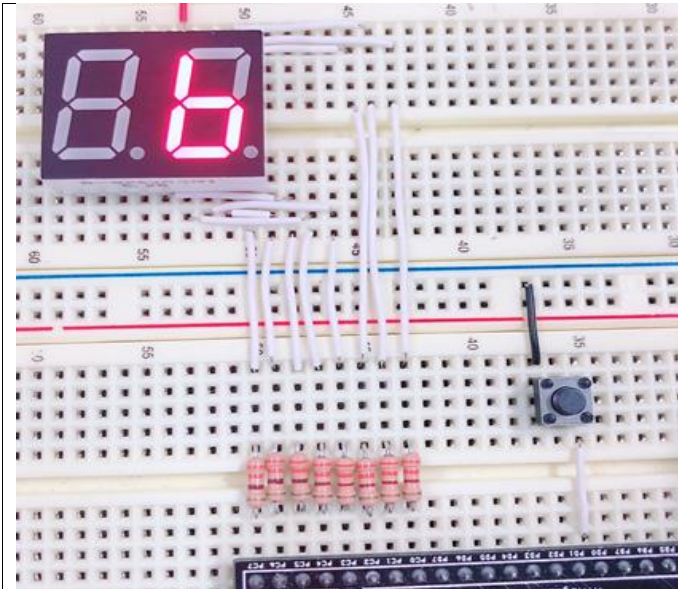


스위치를 3번째 누르면
7 segment에 숫자 3이 표시된다.

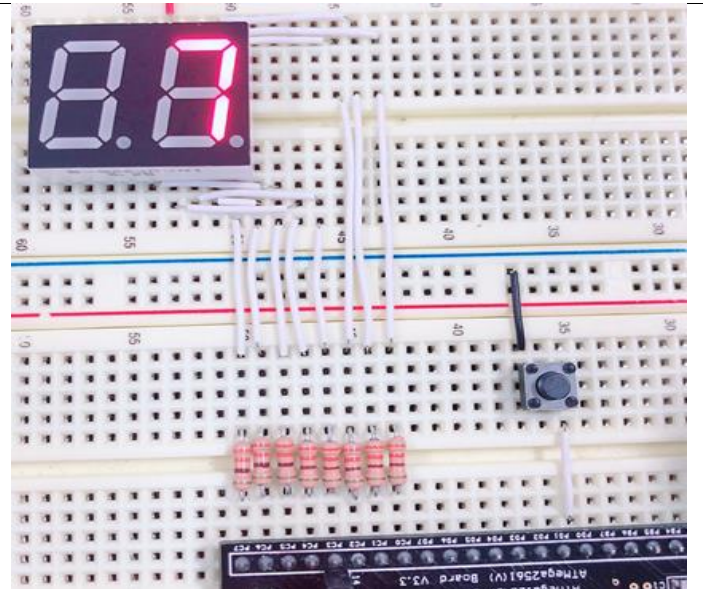


스위치를 4번째 누르면
7 segment에 숫자 4가 표시된다.

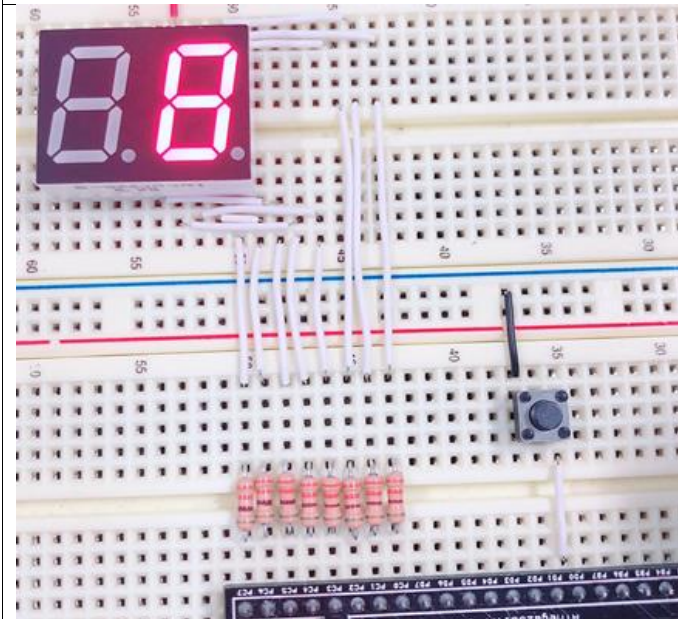
스위치를 5번째 누르면
7 segment에 숫자 5가 표시된다.



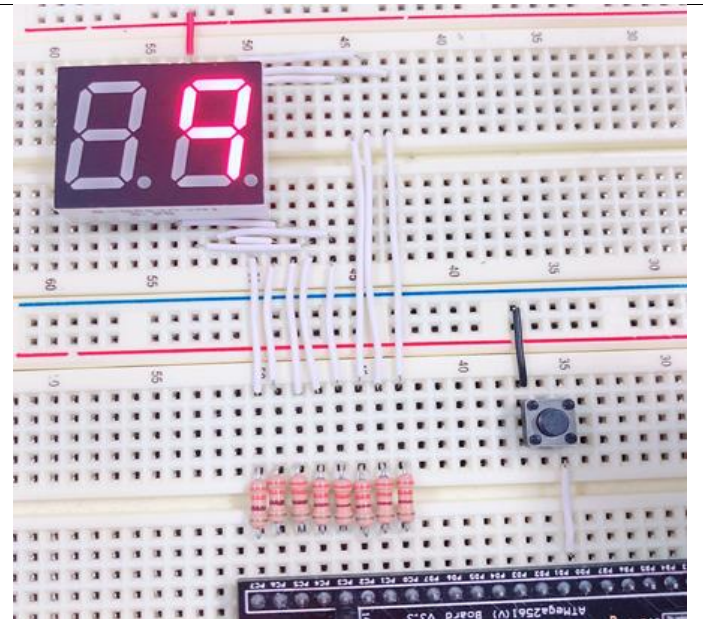
스위치를 6번째 누르면
7 segment에 숫자 6이 표시된다.



스위치를 7번째 누르면
7 segment에 숫자 7이 표시된다.



스위치를 8번째 누르면
7 segment에 숫자 8이 표시된다.



스위치를 9번째 누르면
7 segment에 숫자 9가 표시된다.

스위치를 계속 누르면 위의 과정이 반복된다. 위 실험 결과는 아래의 <1-3>, <1-3-1>의 스위치의 바운싱 현상을 제거하지 못한 프로그램과 <1-4>, <1-4-1>의 바운싱 현상을 제거한 프로그램의 사진처럼 보이는 모습은 같다. 다만, 두 실험의 차이점이 있다면 다음과 같다.

<1-3>, <1-3-1>

바운싱 현상을 **제거하지 못한** 프로그램

<1-4>, <1-4-1>

바운싱 현상을 **제거한** 프로그램

스위치를 누를 때마다 세그먼트의 숫자가 순서대로 하나씩 증가하는 것이 아니라, 여러 번 건너 뛰는 현상 이 발생함.	스위치를 누를 때마다 세그먼트의 숫자가 순서대로 하나씩 증가함.
바운싱 현상 확인	시간 지연 함수를 이용한 디바운싱 이용

1-3. AVR 소스 코드 - 바운싱 현상 확인

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

#define PRESSED 1
#define RELEASED 0

unsigned char digit[] = {0x88, 0xBE, 0xC4, 0xA4, 0xB2, 0xA1, 0x83, 0xBC, 0x80, 0xB0};

void display_7segled(unsigned char led[], unsigned int number)
{ PORTC = led[number]; }

int main(void)
{
    int number;
    int before;

    DDRC = 0xFF;
    DDRD = DDRD & ~(1<<PDD); // PDD를 입력핀으로 설정
    PORTD = PORTD | 1<<PDD; // 입력핀 PDD를 내부 풀업저항으로 연결

    number = 0;
    before = RELEASED;

    while(1){
        display_7segled(digit, number % 10);

        if( before==RELEASED && !(PIND & 1<<PDD) ){ // 전에 눌리지 않은 상태에서 처음으로 눌림
            number++;
            before = PRESSED;
        }else if( before==PRESSED && (PIND & 1<<PDD) ){ // 전에 눌린 상태에서 처음으로 떨어짐
            before = RELEASED;
        }
    }
    return 0;
}
```

1-3-1. 프로그램 분석

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

#define PRESSED 1 //PRESSED를 1로 전처리. const와 비슷한 역할을 하지만, const는 메모리 공간을 차지하는 데 비해 #define은 프로그램을 읽으며 같은 이름으로 정의된 것을 숫자로 바꾸어 읽음.
#define RELEASED 0
```

<#define과 const의 차이 정리>

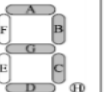
(1) 메모리 공간 차지	<pre>const int PI = 3.141592; // PI라는 전역상수로 선언하여 실제 메모리 공간에 생성 #define PI 3.141592 // 프로그램 컴파일전, 컴파일러의 전처리에 의해 // 뒤의 값(3.141592)으로 전부 변환 area = x * x * PI; area = x * x * 3.141592;</pre>
(2) type 기록	<pre>const 자료형 상수명 = 값 //자료형을 기록함 #define 상수명 값</pre>
(3) 디버깅시 수월함	const로 선언된 것이 값을 찾아보기 수월

`unsigned char digit[] = {0x88, 0xBE, 0xC4, 0xA4, 0xB2, 0xA1, 0x83, 0xBC, 0x80, 0xB0};`

실험에 사용된 세그먼트가 애노드 타입이므로, 다음과 같이 16진수 값을 적어주었다. 구하는 과정은 다음과 같았다. 애노드는 캐소드의 각 포트 출력값이 반대로 된다. Common Anode는 Vcc로 묶어주기 때문이다.

7 SEGMENT			DISPLAY NUMBER									
			0	1	2	3	4	5	6	7	8	9
포트 출력 값	PC7	DP	1	1	1	1	1	1	1	1	1	1
	PC6	C	0	0	1	0	0	0	0	0	0	0
	PC5	E	0	1	0	1	1	1	0	1	0	1
	PC4	D	0	1	0	0	1	0	0	1	0	1
	PC3	G	1	1	0	0	0	0	0	1	0	0
	PC2	F	0	1	1	1	0	0	0	1	0	0
	PC1	A	0	1	0	0	1	0	1	0	0	0
	PC0	B	0	0	0	0	0	1	1	0	0	1
16진수 값			88	BE	C4	A4	B2	A1	83	BC	80	B0

* 참고 : 세그먼트가 캐소드 타입일 경우

LED 위치			디스플레이 숫자									
			0	1	2	3	4	5	6	7	8	9
												
포트 출력 값	PC7	DP	0	0	0	0	0	0	0	0	0	0
	PC6	C	1	1	0	1	1	1	1	1	1	1
	PC5	E	1	0	1	0	0	0	1	0	1	0
	PC4	D	1	0	1	1	0	1	1	0	1	0
	PC3	G	0	0	1	1	1	1	1	0	1	1
	PC2	F	1	0	0	0	1	1	1	0	1	1
	PC1	A	1	0	1	1	0	1	0	1	1	1
	PC0	B	1	1	1	1	1	0	0	1	1	
16진수값			77	41	3B	5B	4D	5E	7C	43	7F	4F

```
void display_7segled(unsigned char led[], unsigned int number)
{
    PORTC = led[number]; }
//위에서 설정한 digit[]를 받아와 원하는 숫자(number)를 출력하는 함수이다.
```

```
int main(void) //main함수. void는 parameter를 갖지 않음을 의미.
{
    int number; // 차례대로 증가시키기 위해 현재 표시할 숫자를 의미한다.
    int before; // 이전의 상태를 저장하기 위한 변수
```

아래 DDR과 PORT에 대한 레지스터 설명, |(or)과 &(amp) 설명은 실험 1에서 자세하게 하였으므로 생략한다.

```
DDRC = 0xFF; // 포트 C를 출력으로 지정한다.
DDRD = DDRC & ~(1<<PD0); // PD0를 입력핀으로 설정
```

0번 PIN을 1로 한 것을 ~(NOT) 취했으니 0으로 &(AND) 한것으로 포트 D 입력핀 설정

```
PORTD = PORTD | 1<<PD0; // 입력핀 PD0를 내부 풀업저항으로 연결
```

회로에서도 스위치에는 저항을 연결해주지 않았다.

```
number = 0; // 숫자는 0부터 표시하기 위해 0으로 초기화 한다.
before = RELEASED; // 처음 시작할 때 이전의 상태는 누르지 않은 상태이다.
```

```
while(1){
    display_7segled(digit, number % 10); //위에서 설정한 숫자를 넣어준다.
    // % 연산자는 나머지를 구하는 연산이다. 예를 들어 number가 5일 경우 number % 10은 5.
    if( before==RELEASED && !(PIND & 1<<PD0) ){ // 전에 눌리지 않은 상태에서 처음으로 눌림
```

before==RELEASED : 전에 눌리지 않은 상태 : 0 참이면 1(TRUE)
1<<PD0 는 스위치를 누르면 LOW(0), PIND와 &연산자로 0을 입력. !는 비트 반전이므로 1이 된다.

```
        number++; // 스위치를 누르면 카운트(number)가 증가한다.
        before = PRESSED; // flag 같은 역할, 스위치의 전 상태가 눌러짐을 표현.
    }else if( before==PRESSED && (PIND & 1<<PD0) ){ // 전에 눌린 상태에서 처음으로 떨어짐
```

before==PRSSSED : 전에 눌린 상태 : 1
1<<PD0 는 스위치가 떨어지면 HIGH(1), PIND와 &연산자로 1을 입력. 결국 참이 되므로 실행.

```
        before = RELEASED; //스위치의 전 상태가 떨어짐을 표현
    }
}
return 0; // main함수가 int형이기 때문에 0을 반환하면서 종료
}
```

위와 같은 소스로 실험을 하면 스위치의 바운스 현상(채터링)이 일어난다.

스위치에서 채터링 현상이 발생하여, 한번 누를 때 다음 단계를 건너뛰고 그 다음(혹은 경충 뛰어감) LED로 켜지는 현상이 있기도 하였다.

-채터링[chattering]이란?

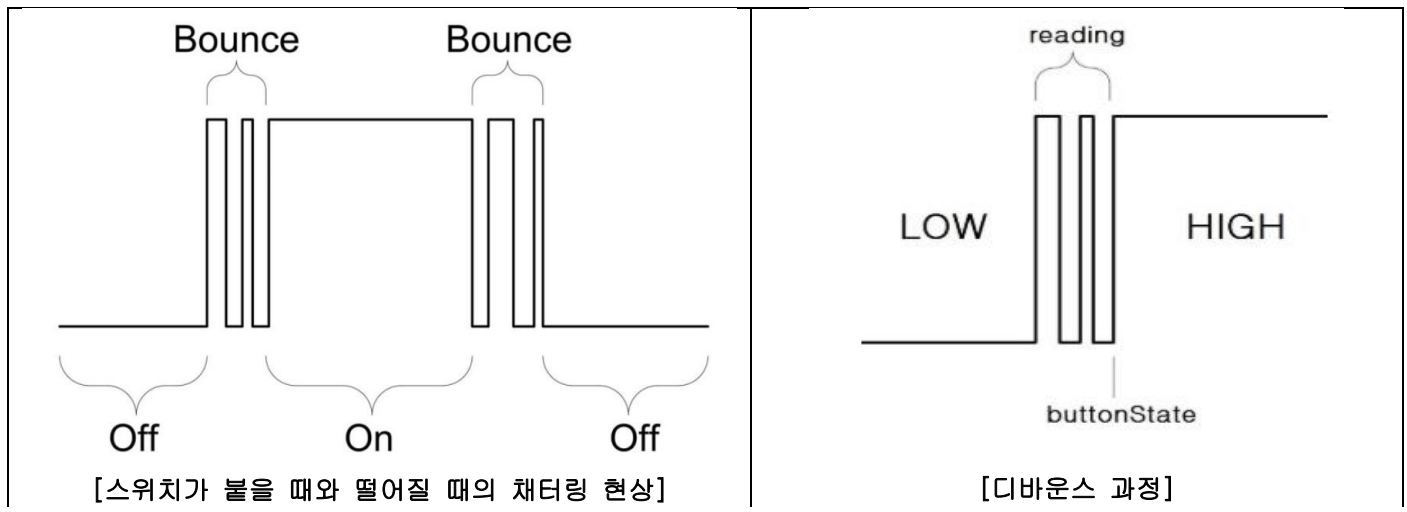
전자 회로 내의 스위치 접점이 닫히거나 열리는 순간에 기계적인 진동에 의해 매우 짧은 시간안에 스위치가 불

있다가 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. 다음 실험을 살펴보자.

하드웨어적으로 채터링 현상을 완화하는 방법은 추가 실험에서 진행하였다.



1-4. AVR 소스 코드 - 디바운싱 이용

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

#define PRESSED 1
#define RELEASED 0
#define DEBOUNCE_MS 100

unsigned char digit[] = {0x88, 0xBE, 0xC4, 0xA4, 0xB2, 0xA1, 0x83, 0xBC, 0x80, 0xB0};

void display_7segled(unsigned char led[], unsigned int number)
{
    PORTC = led[number];
}

int main(void)
{
    int number;
    int before;

    DDRC = 0xFF;
    DDRD = DDRD & ~(1<<PDO); // PDO를 입력핀으로 설정
    PORTD = PORTD | 1<<PDO; // 입력핀 PDO를 내부 풀업저항으로 연결

    number = 0;
    before = RELEASED;

    while(1){
        display_7segled(digit, number % 10);

        if( before==RELEASED && !(PIND & 1<<PDO) ){ // 전에 눌리지 않은 상태에서 처음 눌림
            _delay_ms(DEBOUNCE_MS);
            number++;
            before = PRESSED;
        }else if( before==PRESSED && (PIND & 1<<PDO) ){ // 전에 눌렸으나 처음으로 떨어짐
            _delay_ms(DEBOUNCE_MS);
            before = RELEASED;
        }
    }
    return 0;
}
```

1-4-1. 프로그램 분석 → 1-3-1에서 겹치지 않는 부분만 설명.

`#define F_CPU 16000000UL` // AVR이 16MHz 이므로 설정해주는 대목, 따로 설정해주었다면 해줄 필요 없음

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

```
#define PRESSED 1
#define RELEASED 0
```

`#define DEBOUNCE_MS 100` // 시간 지연을 주기 위해 전처리지시자를 이용한 선언

```
unsigned char digit[] = {0x88, 0xBE, 0xC4, 0xA4, 0xB2, 0xA1, 0x83, 0xBC, 0x80, 0xB0};
```

```
void display_7segled(unsigned char led[], unsigned int number)
{
    PORTC = led[number];
}
```

```
int main(void)
{
```

```
    int number;
    int before;
```

```
    DDRC = 0xFF;
    DDRD = DDRD & ~(1<<PD0); // PD0를 입력핀으로 설정
    PORTD = PORTD | 1<<PD0; // 입력핀 PD0를 내부 풀업저항으로 연결
```

```
    number = 0;
    before = RELEASED;
```

```
    while(1){
        display_7segled(digit, number % 10);
```

```
        if( before==RELEASED && !(PIND & 1<<PD0) ){ // 전에 눌리지 않은 상태에서 처음 눌림
            _delay_ms(DEBOUNCE_MS);
```

// 시간 지연을 주어 바운스 현상이 발생하는 시간 동안을 기다림.

스위치 변화 후 일정시간 강제 시간 지연을 갖게 함. // 이곳이 바운스 현상을 해결해주는 대목.--> 디바운싱

```
            number++;
```

```
            before = PRESSED;
```

```
        }else if( before==PRESSED && (PIND & 1<<PD0) ){ // 전에 눌렀으나 처음으로 떨어짐
```

```
            _delay_ms(DEBOUNCE_MS);
```

```
            before = RELEASED;
```

```
        }
```

```
    }
    return 0;
```

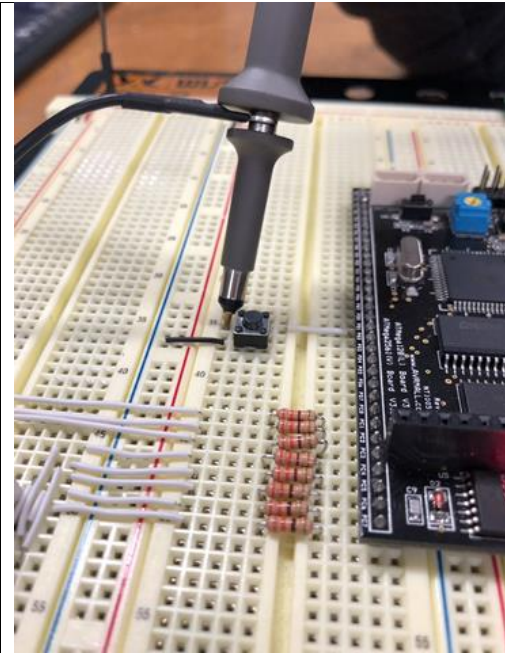
```
}
```

2. 추가실험

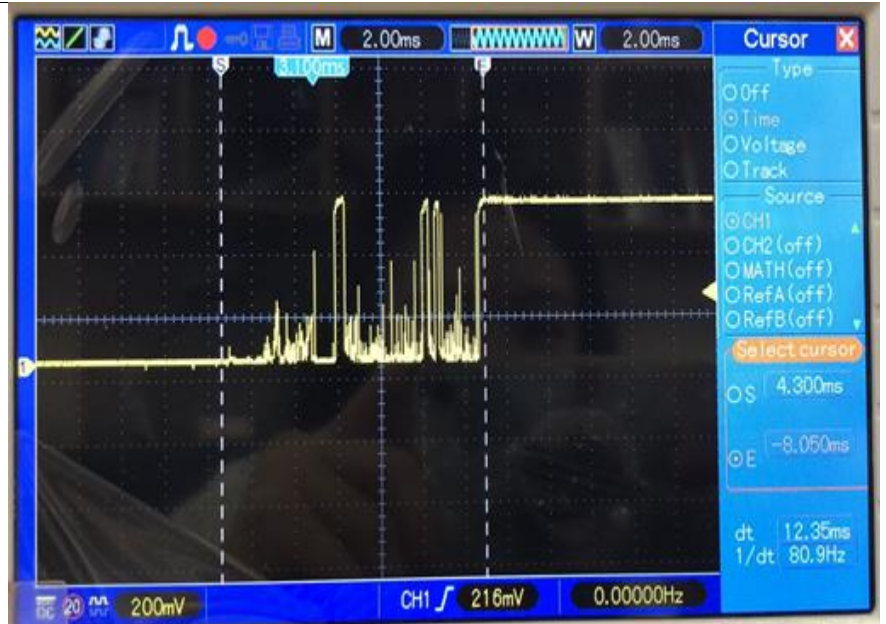
1. 채터링 현상 오실로스코프로 관찰

(1) 실험 환경 - 바운스 현상이 일어나는 실험회로 및 프로그램과 같다.

(1)의 실험 결과

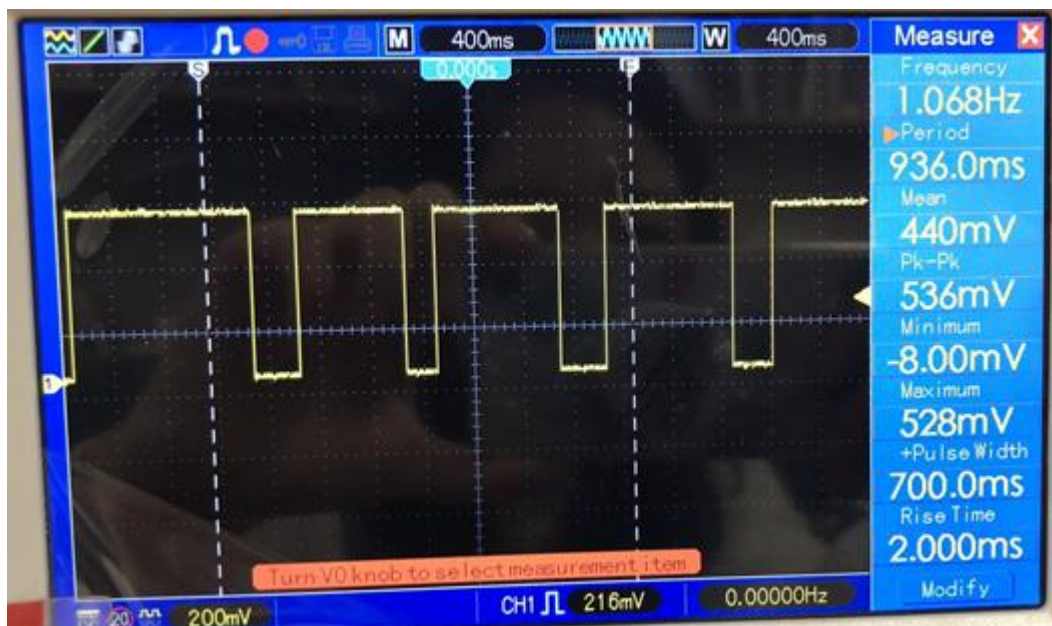


실험 모습



스위치를 누르고 떨어지는 시점에서 바운스 현상이 일어남을 관찰할 수 있다. 바운스 현상이 일어난 시간은 약 12.35ms 동안이다. 스위치를 누르면 LOW가 된다. 스위치가 누를 때의 바운스 현상은 잘 관찰할 수 없어서 의문이 든다.

아래는 바운스 현상이 일어나지 않을 때의 모습이다. 연속적으로 스위치를 눌러주었고 위 표와 다르게 깔끔하게 스위치가 on/off 된다.



2. 다양한 소스 코드 변화

아래의 실험은 본 실험의 while문만 수정하여 진행하였습니다.

(1) number의 위치에 따라

Case1. 바운싱 현상 일어나지 않음	Case2. 바운싱 현상 일어나지 않음
<pre>while(1){ display_7segled(digit, number % 10); if(before==RELEASED && !(PIND & 1<<PDD)){ _delay_ms(DEBOUNCE_MS); number++; before = PRESSED; }else if(before==PRESSED && (PIND & 1<<PDD)){ _delay_ms(DEBOUNCE_MS); before = RELEASED; } }</pre>	<pre>while(1){ display_7segled(digit, number % 10); if(before==RELEASED && !(PIND & 1<<PDD)){ _delay_ms(DEBOUNCE_MS); before = PRESSED; number++; }else if(before==PRESSED && (PIND & 1<<PDD)){ _delay_ms(DEBOUNCE_MS); before = RELEASED; } }</pre>
Case3. 바운싱 현상 일어나지 않음	Case4. 바운싱 현상 일어나지 않음
<pre>while(1){ display_7segled(digit, number % 10); if(before==RELEASED && !(PIND & 1<<PDD)){ _delay_ms(DEBOUNCE_MS); before = PRESSED; }else if(before==PRESSED && (PIND & 1<<PDD)){ number++; _delay_ms(DEBOUNCE_MS); before = RELEASED; } }</pre>	<pre>while(1){ display_7segled(digit, number % 10); if(before==RELEASED && !(PIND & 1<<PDD)){ _delay_ms(DEBOUNCE_MS); before = PRESSED; }else if(before==PRESSED && (PIND & 1<<PDD)){ _delay_ms(DEBOUNCE_MS); number++; before = RELEASED; } }</pre>

4가지 case 모드 바운싱 현상이 일어나지 않았다. 다만, case 3과 4는 number가 else if문에 있기 때문에, 스위치를 누르고 떨어질 때 숫자가 증가하는 모습을 보였다.

(2) _delay_ms의 위치에 따라

Case1. 채터링 多	Case2. 채터링 少
<pre>while(1){ display_7segled(digit, number % 10); if(before==RELEASED && !(PIND & 1<<PDD)){ number++; // _delay_ms(DEBOUNCE_MS); before = PRESSED; }else if(before==PRESSED && (PIND & 1<<PDD)){ _delay_ms(DEBOUNCE_MS); before = RELEASED; } }</pre>	<pre>while(1){ display_7segled(digit, number % 10); if(before==RELEASED && !(PIND & 1<<PDD)){ number++; _delay_ms(DEBOUNCE_MS); before = PRESSED; }else if(before==PRESSED && (PIND & 1<<PDD)){ // _delay_ms(DEBOUNCE_MS); before = RELEASED; } }</pre>

if문에서 delay_ms를 사용하지 않을 때는 else if문에서 delay_ms를 사용하지 않을 때보다 채터링 현상이 더 많이 관찰되었다. 이는 숫자를 증가시키는 역할인 number의 변수의 위치가 if문에 적용되었기 때문에 생겨난 원인이라 생각된다.

(2) _delay_ms의 시간에 따라

- 프로그램 소스는 시간지연을 이용한 디바운스 현상 관찰 소스와 같음. 다만 아래의 시간을 변경해줌.

Case1. 채터링 현상 발견 못함	Case2. 채터링 현상 아주 조금씩 확인	Case3. 채터링 多
#define DEBOUNCE_MS 100	#define DEBOUNCE_MS 10	#define DEBOUNCE_MS 5

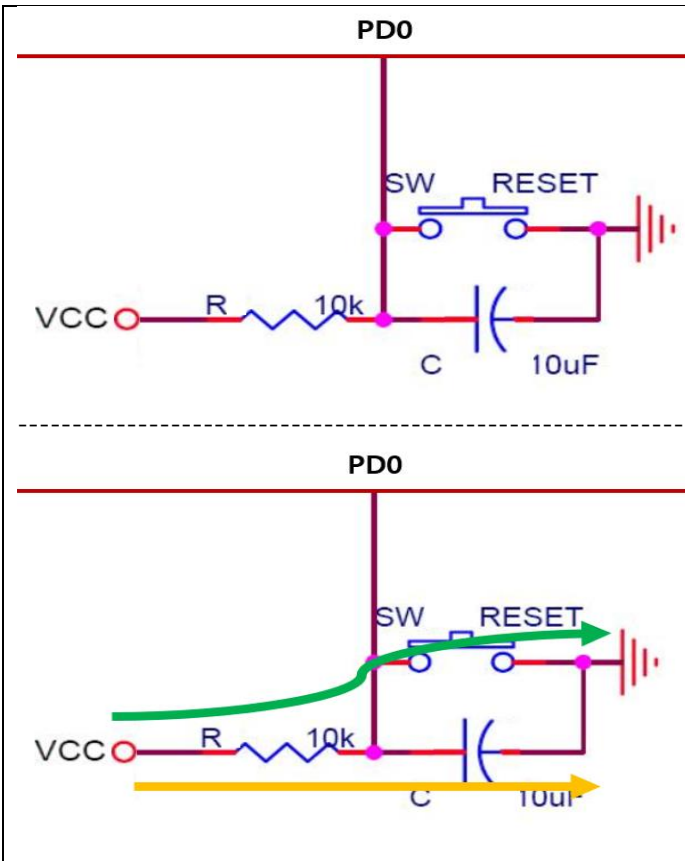
딜레이 시간을 조정해줌에 따라 채터링 현상이 어느 정도 발생하는지 확인해보았다.

10ms를 주었을 때는 채터링 현상이 20번에 2번 꼴로 관찰되었다. 이는 앞에서 오실로스코프로 실험하였을 때의 결과와 관련이 있다. 추가 실험 1의 결과는 12.35ms이다. 실험 당시 가장 많이 채터링 현상이 나온 결과물을 첨부하였다. 그렇기 때문에 위의 10ms로 준 딜레이는 어느 정도 충분한 지연시간이 된다. 그러므로 채터링 현상을 잘 관찰할 수 없었다. 그에 반해 5ms로 주었을 때는 거의 딜레이 지연시간이 없다고 느낄 정도로 채터링 현상이 많이 관찰되었다.

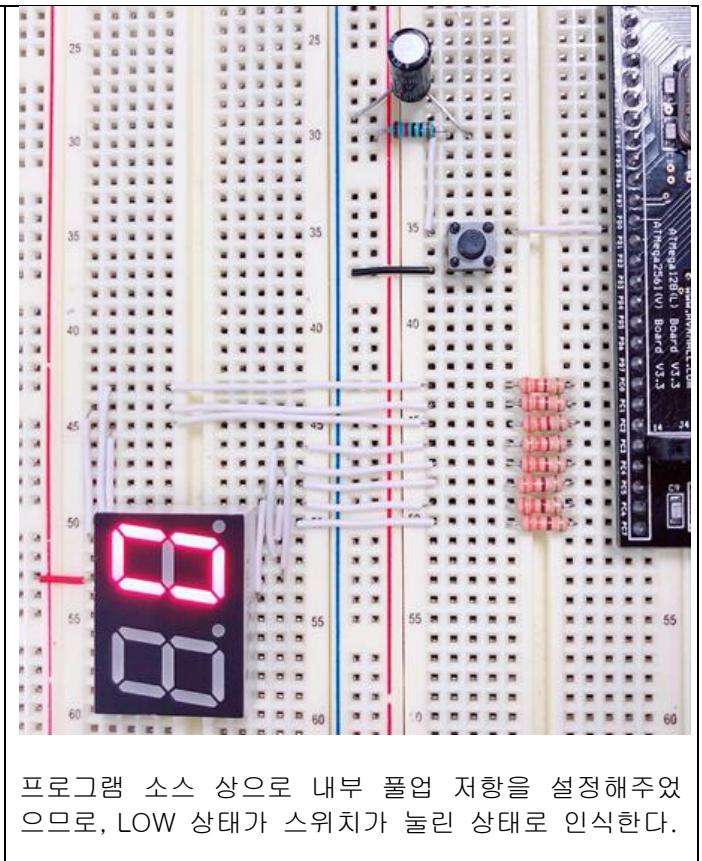
3. 하드웨어적 바운스 현상 해결 - 하드웨어적 디바운싱

* 실험 환경 - 바운스 현상이 일어나는 프로그램소스와 같다.

(1) 스위치를 누를 때 숫자가 바뀐



The diagram illustrates a hardware debouncing circuit. A switch (SW) is connected to a pull-up resistor (R, 10k) and a capacitor (C, 10uF). The resistor is connected to VCC, and the capacitor is connected to ground. The switch is also connected to a RESET pin. The output of the circuit is labeled PD0.



The photograph shows the physical implementation of the circuit on a breadboard. A red LED display shows the number '00'. The circuit components, including the switch, resistor, and capacitor, are visible on the breadboard.

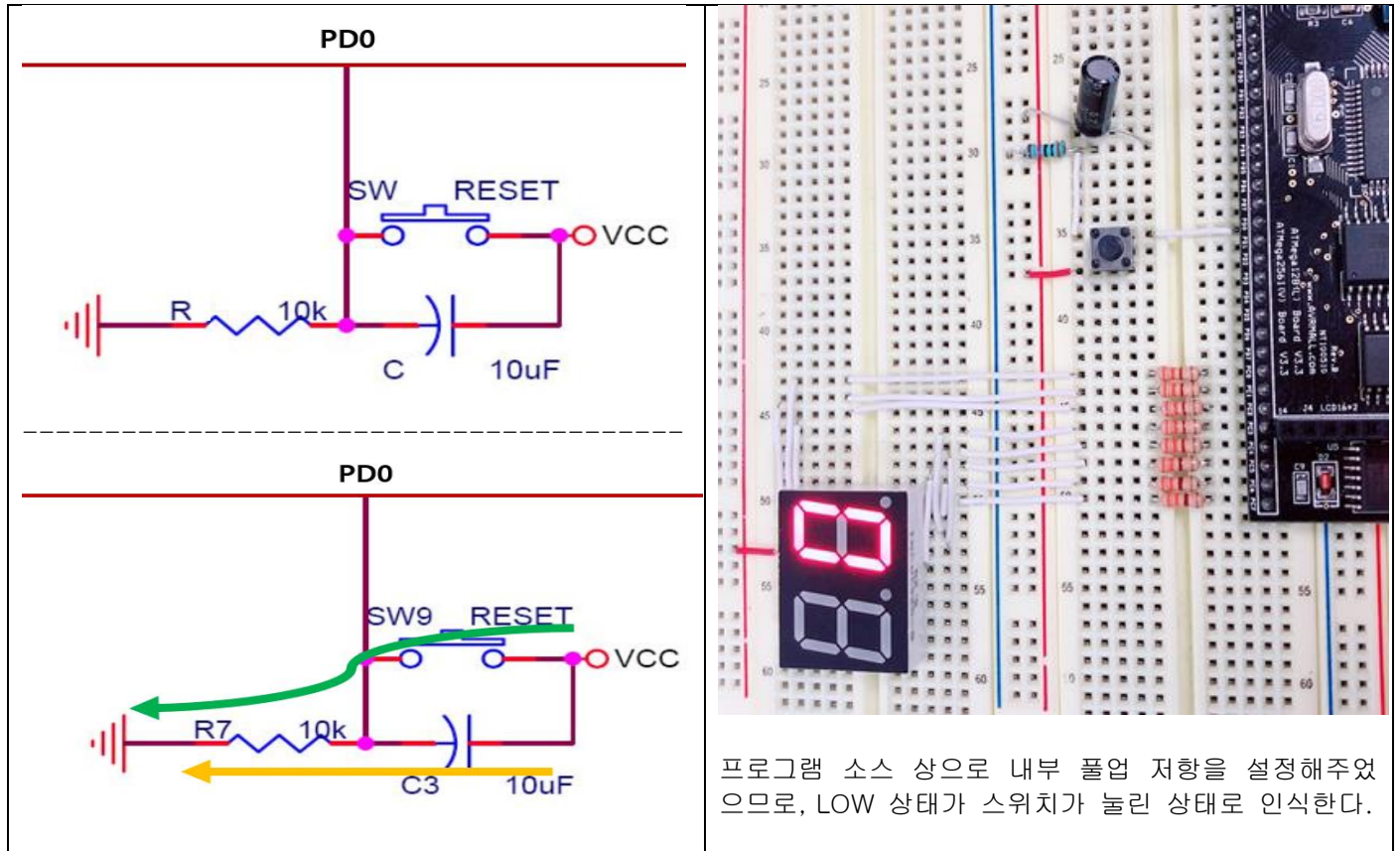
프로그램 소스 상으로 내부 풀업 저항을 설정해주었으므로, LOW 상태가 스위치가 눌린 상태로 인식한다.

회로에서 스위치를 누르면, 초록색 선과 같이 전류가 흐르고, 저항에서 소모가 일어나므로, PD0는 LOW로 인식한다. PD0는 LOW를 스위치가 눌린 상태 인식하므로(내부 풀업 저항)로, 숫자가 바뀐다.

반대로 스위치가 떨어지면 PD0는 HIGH로 인식하기 때문에 스위치가 떨어진 것으로 인식해 숫자는 바뀌지 않

는다. 여기서 커패시터를 사용했기 때문에 소프트웨어적으로 지연시간을 주는 것과 같은 역할을 한다. 커패시터는 시간에 대해 천천히 변화하기 때문이다.

(2) 스위치가 눌렸다가 떨어질 때 숫자가 바뀜



다음 회로도 마찬가지로이다.

회로에서 스위치를 누르면, 초록색 선과 같이 전류가 흐르고, VCC와 연결되었기 때문에, PD0는 HIGH로 인식한다. PD0는 LOW를 스위치가 눌린 상태로 인식하므로(내부 풀업 저항), 숫자가 바뀌지 않는다.

반대로 스위치가 떨어지면 PD0는 LOW로 인식하기 때문에 PD0는 스위치가 붙어진 것으로 인식해 숫자가 바뀐다. 이 경우는 스위치가 눌려진 다음 떨어질 때 숫자가 증가하는 것을 볼 수 있다.

여기서 커패시터를 사용했기 때문에 소프트웨어적으로 지연시간을 주는 것과 같은 역할을 한다. 커패시터는 시간에 대해 천천히 변화하기 때문이다.

결과적으로 위의 두 회로는 바운싱 현상을 관찰할 수 없었다. 그러나 스위치를 아주 빠른 속도로 눌러줄 경우에도 세그먼트의 숫자가 변화하지 않는다는 점이 관찰되었다. 이 점은 커패시터 값을 변화시키면서 좀 더 보완해야 할 과정이라고 생각이 든다.