

## 실험 3. 7-Segment LED 디스플레이

전자공학과 21611591 김 난 희

### 실험 목적

1. Dynamic Display 방식을 이해한다.
2. 트랜지스터 동작을 이해하고 사용해본다.
3. Timer/Counter 0을 이해하고 동작해본다.
4. 잦은 Overflow Interrupt 현상에 의해 동작이 정지될 수 있음을 확인한다.
5. Timer/Counter 0을 이용하여 60초 시계를 만들어 본다.

### 1. 실험 회로

ATmega128 보드

공통 애노드 7-세그먼트 LED

강의 자료와 소자의 pin 연결이 달라 조금 수정해주었다. 저기서 세그먼트의 pin 이 왼쪽 위부터 1로 되는데 사실은 다르게 배열되어있다. 실험에는 다리 개수가 18개인 세그먼트 사용.

### 1-1. 세그먼트 회로 분석

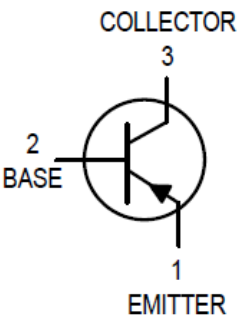
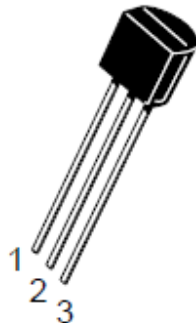
저번 실험 때 자세히 하였으므로, 중요한 것만 기술한다. 왜 330옴 저항을 사용하였는지, 선을 왜 그렇게 연결했는지, 잘 언급하였다. 실험에는 Common Anode 타입의 7-Segment를 사용하였다. 공통 단자를 Vcc에 연결해 주어야 한다.

Anode 타입의 세그먼트

소자명 : S-5263ASR1

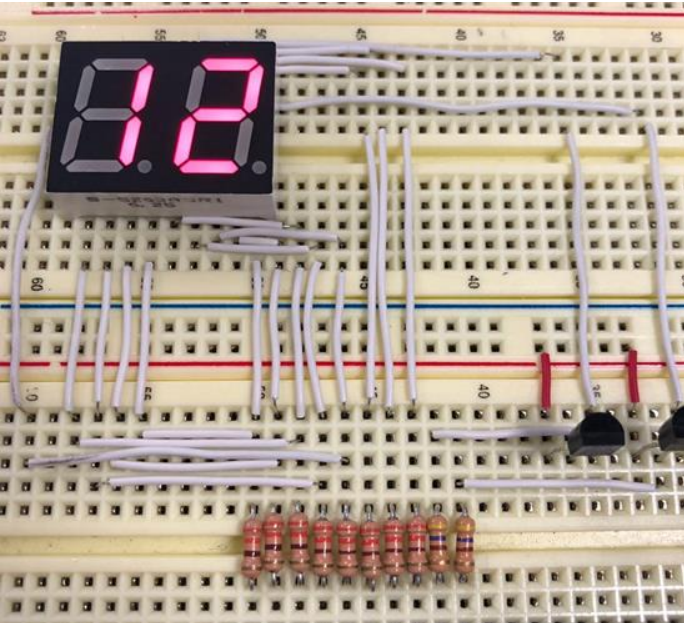
소자명 : S-5263ASR1

## 1-2. 트랜지스터 회로 분석

		<p><b>MAXIMUM RATINGS</b></p> <table><tr><th>Rating</th><th>Symbol</th><th>MPS2907</th><th>MPS2907A</th><th>Unit</th></tr><tr><td>Collector–Emitter Voltage</td><td><math>V_{CEO}</math></td><td>–40</td><td>–60</td><td>Vdc</td></tr><tr><td>Collector–Base Voltage</td><td><math>V_{CBO}</math></td><td colspan="2">–60</td><td>Vdc</td></tr><tr><td>Emitter–Base Voltage</td><td><math>V_{EBO}</math></td><td colspan="2">–5.0</td><td>Vdc</td></tr><tr><td>Collector Current — Continuous</td><td><math>I_C</math></td><td colspan="2">–600</td><td>mAdc</td></tr></table> <p>소자명 : MPS2907A</p>	Rating	Symbol	MPS2907	MPS2907A	Unit	Collector–Emitter Voltage	$V_{CEO}$	–40	–60	Vdc	Collector–Base Voltage	$V_{CBO}$	–60		Vdc	Emitter–Base Voltage	$V_{EBO}$	–5.0		Vdc	Collector Current — Continuous	$I_C$	–600		mAdc
Rating	Symbol	MPS2907	MPS2907A	Unit																							
Collector–Emitter Voltage	$V_{CEO}$	–40	–60	Vdc																							
Collector–Base Voltage	$V_{CBO}$	–60		Vdc																							
Emitter–Base Voltage	$V_{EBO}$	–5.0		Vdc																							
Collector Current — Continuous	$I_C$	–600		mAdc																							

위 트랜지스터는 Emitter에서 Base로 순방향 전류가 흐르므로, pnp형 트랜지스터이다. PD6과 PD7번 PIN에서 LOW가 SET되어 나오면, 각각의 트랜지스터 Base에서 통로를 열어준다. Emitter와 연결해 놓은 Vcc에서 전류가 통해 흘러 결국 세그먼트의 Select pin에 도달하게 된다. 470옴을 base에 연결해준 것은 순방향 전류가 세그먼트 까지 잘 도달하기 위함이다. 또한 정확한 low신호를 주기 위함이다.

## 2. 실험 결과

	<p>왼쪽 사진보다 불이 밝게 들어온다.</p> <p>실제로는 두 세그먼트가 번갈아서 빠르게 켜졌다 꺼졌다를 반복한다. 눈으로는 깜빡이는 것이 보이지 않는다. 휴대폰의 슬로우 모션 기능을 이용하거나, 회로를 흔들어보면 꺼졌다 켜지는 것이 보이기도 한다.</p> <p>이에 따라 분주를 달리 설정해주면 정지해 보이거나, 오히려 서로 깜빡이며 반복하는 모습이 눈에 보이기도 한다. 이는 추가 실험에서 설명할 것이다.</p>
--	--

### 3. 프로그램 코드

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

volatile unsigned long timer0; // 오버플로마다 1씩 증가될 변수
volatile unsigned int number; // 증가되어 7-세그먼트 LED에 디스플레이될 숫자

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

ISR(TIMERO_OVF_vect) // 타이머/카운터0 인터럽트 서비스 루틴
{
    timer0++; // 오버플로마다 1씩 증가

    // 오버플로 횟수가 4의 배수일 때 10자리 혹은 2의 배수일 때 1자리 디스플레이
    if(timer0 % 2 == 0){
        PORTC = (timer0%4 == 0) ? led[(number%100) / 10] : led[number%10];
        PORTD = (PORTD | 0xC0) & ~(1<<((timer0 % 4 == 0) ? PD7 : PD6));
    }
}

int main(void)
{
    DDRC = 0xFF;
    DDRD |= 1<<PD7 | 1<<PD6; // 번갈아가며 7-세그먼트 LED를 켜기위한 출력

    TCCR0 |= 1<<CS02 | 1<<CS01; // 256 프리스케일 설정
    TIMSK |= 1<<TOIE0; // 타이머/카운터0 인터럽트 활성화

    timer0 = 0; //타이머 오버플로마다 갱신되는 변수
    sei();

    number = 12; // 디스플레이할 숫자
    while(1);

    return 0;
}
```

#### 3-1. 프로그램 분석

```
#include <avr/io.h> //AVR 기본 입출력 관련 헤더파일
#include <avr/interrupt.h> // 인터럽트 관련 헤더파일 //타이머 인터럽트 사용을 위해 사용됨
```

```
volatile unsigned long timer0; // 오버플로마다 1씩 증가될 변수
volatile unsigned int number; // 증가되어 7-세그먼트 LED에 디스플레이될 숫자
// 타이머 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.

**volatile unsigned char count=0;**

```
while(1)
    if(count == 100)
        fc: 80 91 00 01 lds    r24, 0x0100
            100: 84 36      cpi     r24, 0x64; 100
            102: e1 f7      brne  .-8      ; 0xfc
                PORTC = PORTB;
            104: 88 b3      in     r24, 0x18; 24
            106: 85 bb      out    0x15, r24; 21
            108: f9 cf      rjmp  .-14      ; 0xfc
```

**unsigned char count=0;**

```
while(1)
    if(count == 100)
        fc: 80 91 00 01 lds    r24, 0x0100
            100: 84 36      cpi     r24, 0x64; 100
            102: 19 f4      brne  .+6      ; 0x10a
                PORTC = PORTB;
            104: 88 b3      in     r24, 0x18; 24
            106: 85 bb      out    0x15, r24; 21
            108: fd cf      rjmp  .-6      ; 0x104
            10a: ff cf      rjmp  .-2      ; 0x10a
```

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

실험에 사용된 세그먼트는 Anode 타입이다. 다음과 같이 16진수값을 설정해주었다.

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

강의자료의 소스코드가 이 코드와 다른 이유는, 강의자료의 세그먼트는 다리 PIN의 개수가 10개가 있는 세그먼트를 사용했기 때문이다. 이러한 소스코드는 세그먼트 종류에 따라(Anode, Cathode), PIN의 모습에 따라(실험에 사용한 코드와 강의자료) 달라진다.

ISR(TIMER0\_OVF\_vect)// 타이머/카운터0 인터럽트 서비스 루틴

Interrupt Service Routine: 인터럽트가 호출되었을 시 실행되는 루틴으로, 여기서는 타이머 인터럽트가 호출되었을 시 실행된다.

```
{
    timer0++;           // 오버플로마다 1씩 증가

    // 오버플로 횟수가 4의 배수일 때 10자리 혹은 2의 배수일 때 1자리 디스플레이
    if(timer0 % 2 == 0){ //2의 배수일 때
        PORTC = (timer0%4 == 0) ? led[(number%100) / 10] : led[number%10];
        4의 배수일 때 참이면 led[(number%100) / 10] 실행 : 10자리 디스플레이
        4의 배수일 때 거짓이면 led[number%10] 실행 : 1자리 디스플레이
        아래에서 number = 12 이므로, 12를 디스플레이 한다.
        %는 나머지 연산자로, [(number%100) / 10] = [(12%100)/10] = 12/10 = 1
        [number%10] = 12%10 = 2
```

```
    PORTD = (PORTD | 0xC0) & ~(1 << ((timer0 % 4 == 0) ? PD7 : PD6));
```

4의 배수일 때 참이면 PD7 PIN을 1로 SET하고, 거짓이면 PD6 PIN을 1로 SET한다.

이것이 ~연산자로 not되므로, 0(LOW)이 된다.

PORTD 에서 0b11000000이 |(or)연산자로 되어있으므로, PORTD는 0b11000000로 SET된다.

PORTD	1	1	0	0	0	0	0	0
-------	---	---	---	---	---	---	---	---

다시 위의 ~ 연산자의 결과인 0과 &를 취하므로, PD7과 PD6자리에는 4의 배수인지에 따라 번갈아가며 LOW가 된다. LOW가 아니면 HIGH가 SET 된다.

4의 배수 일경우 : PD7 = LOW → 10의 자리 디스플레이

0	1	0	0	0	0	0	0
---	---	---	---	---	---	---	---

4의 배수 아닐 경우 : PD6 = LOW → 1의 자리 디스플레이

1	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

이것으로 부터 두 PIN에서 LOW가 나오면 트랜지스터는 PNP형이기 때문에 ON이 된다. 트랜지스터의 Emitter에서 공급되는 전류가 세그먼트로 흐르면서 불이 켜지게 된다.

즉, timer0 변수가 4의 배수일 경우 10의 자리가, 아닐 경우 1의 자리가 켜지게 된다.

여기서 세그먼트에서 Common Anode 단자가 digital select 역할을 하게 됨을 알 수 있다.

}

}

int main(void) //main함수. void는 parameters(매개변수)가 갖지 않음을 의미

{

DDRC = 0xFF; //전체 출력으로 지정

우리는 Anode 타입 세그먼트를 사용하였기 때문에 세그먼트의 해당하는 각각의 LED를 켜기 위해서는 LOW설정을 해주어야 한다. 이는 위의 unsigned char led[] 소스 부분의 설정을 참고한다.

DDRD |= 1<<PD7 | 1<<PD6; // 번갈아가며 7-세그먼트 LED를 켜기 위한 출력

TCCR0 |= 1<<CS02 | 1<<CS01; // 256 프리스케일 설정

Timer/Counter Control  
Register – TCCR0

Bit	7	6	5	4	3	2	1	0	
	FOC0	WGM00	COM01	COM00	WGM01	CS02	CS01	CS00	TCCR0
Read/Write	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	

위 레지스터에서 CS02와 CS01만 1로 SET하였기 때문에 다음과 같이 설정이 된다.

TCCR0	0	0	0	0	0	1	1	0
-------	---	---	---	---	---	---	---	---

타이머/카운터 0번을 사용하기 때문에 각 비트의 처음오는 숫자는 모두 0으로 되어있다.

최상위 7번 FOC0 비트는 NON-PWM 모드일 때 1로 SET 된다. 이때 Waveform Generation Unit으로 동작한다.

아래는 WGM00과 WGM01비트를 0으로 설정해주었기 때문에 Waveform Generation Mode를 Normal 모드로 설정한다. 최고점은 8비트이므로 0xFF가 된다 OCR0는 중간에서 업데이트되며, 오버플로 플래그는 최고점에서 SET 됨을 보여준다.

Table 52. Waveform Generation Mode Bit Description

Mode	WGM01 <sup>(1)</sup> (CTC0)	WGM00 <sup>(1)</sup> (PWM0)	Timer/Counter Mode of Operation	TOP	Update of OCR0 at	TOV0 Flag Set on
0	0	0	Normal	0xFF	Immediate	MAX
1	0	1	PWM, Phase Correct	0xFF	TOP	BOTTOM
2	1	0	CTC	OCR0	Immediate	MAX
3	1	1	Fast PWM	0xFF	BOTTOM	MAX



또한 COM00과 COM01비트를 0으로 SET해놓았으므로, OC0를 사용하지 않음과 같다.

**Table 53. Compare Output Mode, non-PWM Mode**

COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected.
0	1	Toggle OC0 on compare match
1	0	Clear OC0 on compare match
1	1	Set OC0 on compare match

마지막으로 CS00, CS01, CS02 비트는 분주를 결정한다.

CS02, CS01, CS00을 1,1,0으로 설정했으므로, 다음 clock select를 설정한다.

**Table 56. Clock Select Bit Description**

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	clk <sub>T0S</sub> /(No prescaling)
0	1	0	clk <sub>T0S</sub> /8 (From prescaler)
0	1	1	clk <sub>T0S</sub> /32 (From prescaler)
1	0	0	clk <sub>T0S</sub> /64 (From prescaler)
1	0	1	clk <sub>T0S</sub> /128 (From prescaler)
1	1	0	clk <sub>T0S</sub> /256 (From prescaler)
1	1	1	clk <sub>T0S</sub> /1024 (From prescaler)

Normal mode일 때, 위 설정에 따른 주기를 계산해본다.

$$f = \frac{f_{clk}}{N \times 256}$$

N은 위에서 256으로 설정,  $f_{clk}$ 은 16MHz이므로,

$$f = \frac{16 \times 10^6}{256 \times 256}$$

주기 T는 주파수  $f$ 의 역수이므로,

$$T = \frac{256 \times 256}{16 \times 10^6} = 0.004096sec = 4.096msec$$

위의 소스 코드에서 다음과 같이 설정해주었으므로,

```
if(timer0 % 2 == 0){
    PORTC = (timer0%4 == 0) ? led[(number%100) / 10] : led[number%10];
    PORTD = (PORTD | 0xC0) & ~(1 << ((timer0 % 4 == 0) ? PD7 : PD6));
```

2T(8.192msec)가 한 번 켜져있거나 꺼져있는 주기이다. 전체 켜졌다 꺼지는 주기는 4T가 된다.

TIMSK |= 1<<TOIE0;      // 타이머/카운터0 인터럽트 활성화

아래의 타이머/카운터0을 사용하므로, TIMSK에서 다음 주황색 형관펜으로 칠해진 부분만 보면 된다. OC 관련해서는 사용하지 않기 때문에 제외하고 본다. TIMSK의 TOIE0(Timer/counter0 Overflow Interrupt Enable)가 1로 SET되면 Overflow 인터럽트를 Enable하는 것이다.

아래의 TIFR(Timer/Counter Interrupt Flag Register)의 TOV0가 1로 SET되면, 인터럽트를 걸 수 있다.

### Timer/Counter Interrupt Mask Register – TIMSK

Bit	7	6	5	4	3	2	1	0	
	OCIE2	TOIE2	TICIE1	OCIE1A	OCIE1B	TOIE1	OCIE0	TOIE0	TIMSK
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	

### Timer/Counter Interrupt Flag Register – TIFR

Bit	7	6	5	4	3	2	1	0	
	OCF2	TOV2	ICF1	OCF1A	OCF1B	TOV1	OCF0	TOV0	TIFR
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	

timer0 = 0; //타이머 오버플로마다 갱신되는 변수  
처음 시작 전에 0으로 초기화 해놓는다.

sei(); //Global interrupt Enable bit

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	

위에서 인터럽트 허용 비트를 Enable 해주고 다음 sei()를 해주면, 전체 인터럽트를 허용하여 최종적으로 인터럽트를 사용 가능하게 한다. sei()는 Set Interrupt Flag로 위의 SREG의 최상위 비트, I비트를 1로 SET하여 전체 인터럽트를 허용 여부를 판단한다.

number = 12; // 디스플레이할 숫자

while(1); //무한 반복

무한 루프를 통해 아무 것도 안하는 것을 수행하고 있다. 어셈블리어로 말하면 NOP를 수행하고 있다. 이 while문을 반복 수행하면서 인터럽트가 걸리면 ISR(인터럽트 서비스 루틴)을 수행하는 형태로 되어 있다.

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









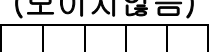
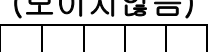
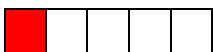


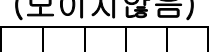
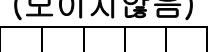
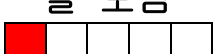

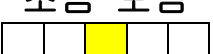
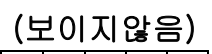
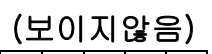
}

## 2. 추가 실험

### (1) 분주 바꾸기

2-1. 실험 회로 → 본 실험과 회로가 동일합니다.

#### 2-2. 프로그램 분석 및 결과

Prescalar	$clk_{TOS}/1024$	$clk_{TOS}/256$	$clk_{TOS}/128$	$clk_{TOS}/64$	$clk_{TOS}/8$
변경된 소스 (Clock Select 부분)	TCCR0  = 1<<CS02   1<<CS01   1<<CS00;	TCCR0  = 1<<CS02   1<<CS01;	TCCR0  = 1<<CS02   1<<CS00;	TCCR0  = 1<<CS02;	TCCR0  = 1<<CS01;
눈에 보이는 7-seg의 밝기	중간 	밝음 	아주 밝음 	아주 밝음 	아주 밝음 
깜빡임 (불이 꺼졌다 켜지는 정도)	심함 	조금 	아주 조금 	없음 (보이지않음) 	없음 (보이지않음) 
영상으로 본 깜빡임 (슬로우 모션 동영상)	심함 	조금 	조금 	없음 (보이지않음) 	없음 (보이지않음) 
흔들림에 의한 깜빡임 (회로를 흔들 때 깜빡임이 보이는 정도)	잘 보임 	보임 	조금 보임 	없음 (보이지않음) 	없음 (보이지않음) 

분주비가 8에서 1024로 커질수록 전체 주기가 커지므로, 깜빡이는 주기가 길어짐을 알 수 있다. 주기가 길어지므로 눈에 깜빡임이 더 잘 보인다. 흔들림에 의한 깜빡임은 회로를 가만히 두었을 때는 깜빡임이 보이지 않지만, 좌우 또는 상하로 흔들어주면 깜빡이는 정도가 보이게 된다. dynamic display의 특징이다.

$$F_{CPU} = 16\text{Mhz} \rightarrow 1/16000000\text{Hz} = 0.0000000625\text{s} = 0.0000625\text{ms}$$

$$16\text{Mhz} / 8 = 16,000,000/8 = 2,000,000\text{Hz} \rightarrow (\text{주파수의 역수는 주기}) 0.0000005 \text{ s} = 0.0005 \text{ ms}$$

$$16\text{Mhz} / 64 = 16,000,000/64 = 250,000\text{Hz} \rightarrow 0.000004 \text{ s} = 0.004 \text{ ms}$$

$$16\text{Mhz} / 128 = 16,000,000/128 = 125000\text{Hz} \rightarrow 0.000008 \text{ s} = 0.008 \text{ ms}$$

$$16\text{Mhz} / 256 = 16,000,000/256 = 62500\text{Hz} \rightarrow 0.000016 \text{ s} = 0.016 \text{ ms}$$

$$16\text{Mhz} / 1024 = 16,000,000/1024 = 15625 \text{ Hz} \rightarrow 0.000064 \text{ s} = 0.064 \text{ ms}$$



## 2. 추가 실험

### (2) 타이머를 이용하여 60초 시계 만들기

2-1. 실험 회로 → 본 실험과 회로가 동일합니다.

#### 2-2. 프로그램 분석 및 결과

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

volatile unsigned long timer0; // 오버플로마다 1씩 증가될 변수

volatile unsigned int number=0; // 증가되어 7-세그먼트 LED에 디스플레이될 숫자
unsigned char led[] = {0x88, 0xBE, 0xC4, 0xA4, 0xB2, 0xA1, 0x83, 0xBC, 0x80, 0xB0};

ISR(TIMER0_OVF_vect) // 타이머/카운터0 인터럽트 서비스 루틴
{
    timer0++; // 오버플로마다 1씩 증가

    if(timer0 % 8 == 0){ // 오버플로 횟수가 16의 배수일 때 10자리 혹은 의 8배수일 때 1자리 디스플레이
        PORTC = (timer0%16 == 0) ? led[(number%100) / 10] : led[number%10];
        PORTD = (PORTD | 0xC0) & ~(1<<((timer0 % 16 == 0) ? PD7 : PD6));
    }

    if(timer0 >= 62500){ //256/16M *62500 = 1 => timer0가 62500이 되는 시점은 1초가 되는 시점
        number++; //1초마다 number 증가
        timer0 = 0; //1초후 다시 헤아리기 위해 갱신
        if(number==61) number=1; //60초가 된 이후 1초부터 시작
    }
}

int main(void)
{
    DDRC = 0xFF;
    DDRD |= 1<<PD7 | 1<<PD6; // 번갈아가며 7-세그먼트 LED를 켜기위한 출력

    TCCR0 |= 1<<CS00; // NO 프리스케일 설정(분주비 1)
    TIMSK |= 1<<TOIE0; // 타이머/카운터0 인터럽트 활성화

    timer0 = 0; //타이머 오버플로마다 갱신되는 변수 //시작 전에 0으로 초기화
    sei(); // 글로벌 인터럽트 허용

    while(1);

    return 0;
}
```

본 실험 소스 코드에서 추가하거나 수정된 부분만 추가로 설명한다. 간단하게 delay를 이용하여 1초를 계산할 수 있지만, 정확한 1초를 구현하지는 못할 것이라 생각이 들었다. 프로그램 동작시간을 고려해서이다. 또한, 추가 실험에서는 타이머/카운터 외에 다른 기능을 사용하고 있지 않기 때문에 delay 함수를 사용하여도 괜찮긴 하지만, 만약 MCU가 다른 일들을 수행하면서 타이머/카운터를 사용하여야 한다면 타이머를 이용하여 delay함수를 대체하여야 한다. PWM 기능을 사용하게 된다면, PWM 신호를 꺼내 쓰게 되므로, 타이머를 이용한 시간 계산은 더욱더 중요하게 된다.

```
if(timer0 % 8 == 0){
    // 오버플로 횟수가 16의 배수일 때 10자리 혹은 의 8배수일 때 1자리 디스플레이
    PORTC = (timer0%16 == 0) ? led[(number%100) / 10] : led[number%10];
    PORTD = (PORTD | 0xC0) & ~(1<<((timer0 % 16 == 0) ? PD7 : PD6));
}
```

처음에는 본 실험처럼 4의 배수일 때와 2의 배수일 때로 하였을 때는, 숫자가 너무 자주 바뀌어서 정확한 세그먼트의 숫자 모양이 보이지 않았다. 바뀌는 지점의 횟수를 늘려서 딱 16배, 8배가 되었을 때, 숫자가 진하게 잘 보였다. 1의 자리와 10의 자리 세그먼트에서 서로의 숫자가 비춰 보였던 것을 잘 해결하였다.

```
if(timer0 >= 62500){
```

//256/16M \*62500 = 1 => timer0가 62500이 되는 시점은 1초가 되는 시점

Normal mode일 때, 위 설정에 따른 주기를 계산해본다.

$$f = \frac{f_{clk}}{N \times 256}$$

N은 위에서 1(NO 프리스케일)로 설정,  $f_{clk}$ 은 16MHz이므로,

$$f = \frac{16 \times 10^6}{1 \times 256}$$

주기 T는 주파수  $f$  의 역수이므로,

$$T = \frac{1 \times 256}{16 \times 10^6} = 0.000016sec = 0.016msec$$

$$T \times 62500 = 1sec$$

```
number++; //1초마다 number 증가
```

세그먼트에 1초마다 숫자가 바뀌면서 표시하기 위함

```
timer0 = 0; //1초후 다시 헤아리기 위해 갱신
```

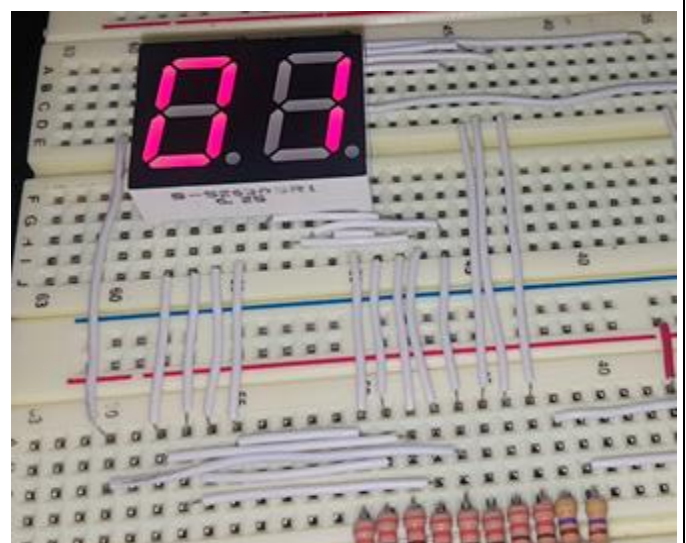
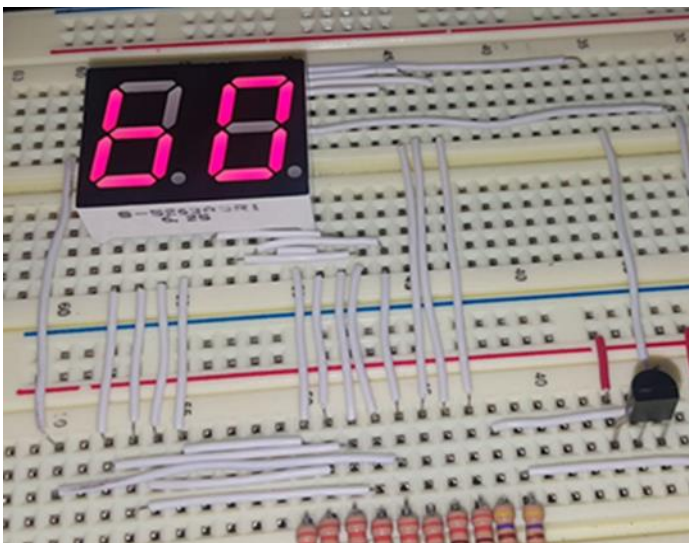
```
if(number==61) number=1; //60초가 된 이후 1초부터 시작
```

```
}
```

만약 이 부분을 넣어주지 않으면 숫자 99가 표시된 이후 00이 된다.

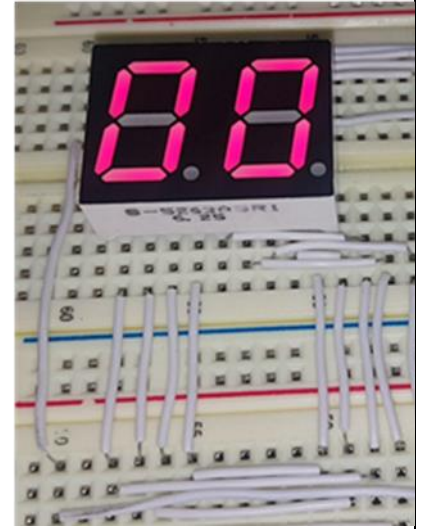
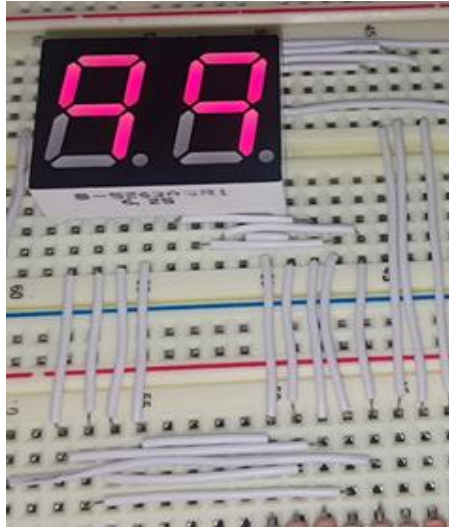
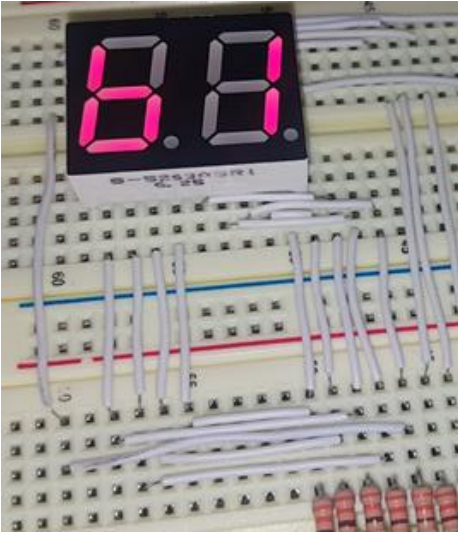
**if(number==61) number=1;** 를 넣은 경우

60이후 바로 1이 된다.



`// if(number==61) number=1;` 를 넣은 경우

60이후 61이 되며, 숫자가 계속 증가해 99이후 00으로 된다.



`TCCR0 |= 1<<CS00;` // NO 프리스케일 설정(분주비 1)

나머지는 0이 됨과 같다. 나머지 비트는 본 실험과 같으므로, 설명을 생략한다.

**Table 56.** Clock Select Bit Description

CS02	CS01	CS00	Description
0	0	0	No clock source (Timer/Counter stopped)
0	0	1	clk <sub>T0S</sub> /(No prescaling)
0	1	0	clk <sub>T0S</sub> /8 (From prescaler)
0	1	1	clk <sub>T0S</sub> /32 (From prescaler)
1	0	0	clk <sub>T0S</sub> /64 (From prescaler)
1	0	1	clk <sub>T0S</sub> /128 (From prescaler)
1	1	0	clk <sub>T0S</sub> /256 (From prescaler)
1	1	1	clk <sub>T0S</sub> /1024 (From prescaler)

본 실험에서 사용한 타이머/카운터0을 이용하여, 분주비와 주기 계산을 통해 1초가 되는 시점을 만들고, 직접 확인해볼 수 있어, 분주비 설정을 공부하는데 많은 도움이 된 것 같다. 1초를 만들기 위해 여러가지 방법으로 시도해볼 수 있음을 생각해보게 되었다. 추가하여 interrupt 벡터가 저장되는 주소를 확인해볼 생각이다.