

REPORT

임베디드 코딩 및 실습

- Final Project -



담당 교수:

김한솔 교수님

학부(과) / 전공:

인공지능공학부 지능제어시스템공학

20181304 김기태

보고서 작성자 학번 / 이름:

20181312 류재후

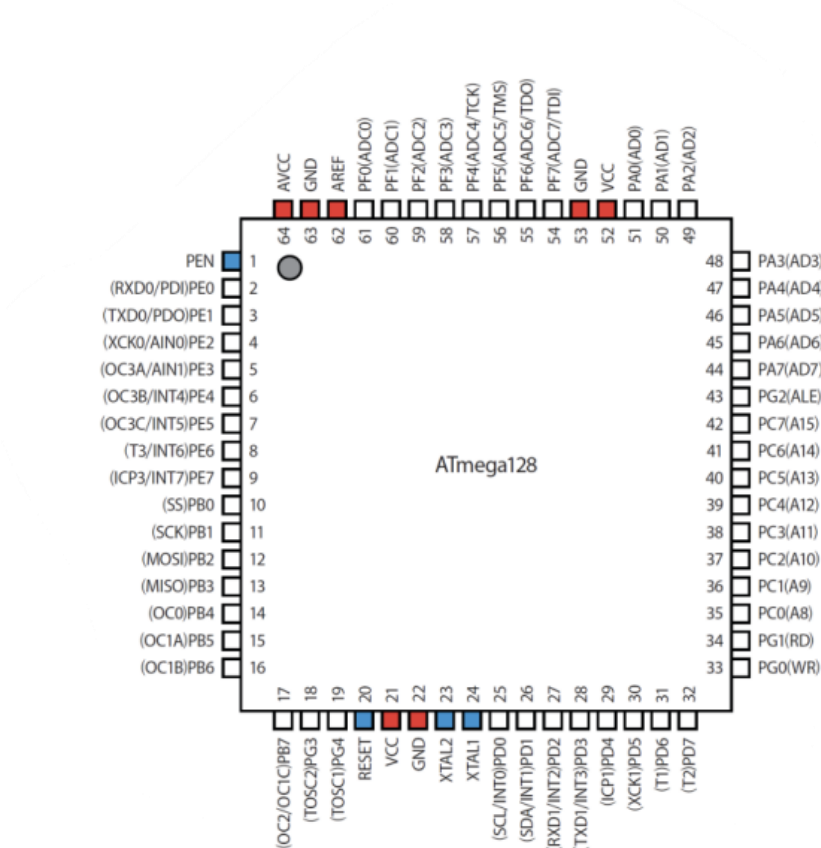
20181329 신승훈

1. 사용한 모듈의 기능 및 레지스터 설명

[Atmega 128]

GPIO

ATMega128은 알맞은 전원을 공급할 수 있는 회로가 필요한 레귤레이터(Regulator)회로, 일정한 속도로 동작하기 위한 클럭 발진 회로인 크리스탈 발진 회로, 리셋 명령을 내릴 수 있는 회로인 리셋 스위치 회로, 프로그램 다운로드하기 위한 회로인 ISP 커넥터 회로, 전원 입력 확인과 테스트용 LED 회로, 기준전압을 입력하는 부분을 위한 AD Converter 회로, ATMega128 칩을 구동하기 위해 필요한 Main MCU 회로도 순으로 나열되어 있다.



VCC, GND, 전원회로: MUC 동작을 위한 전원 연결

레귤레이터 회로: 알맞은 전압의 전원을 공급할 수 있는 LDO, Switching regulator, Boost convert IC회로

크리스탈 발진 회로: 일정한 주기로 클럭이 입력되어야 마이크로컨트롤러가 타이밍 작업을 할 수 있는 회로

RESET Switch 회로: 마이크로컨트롤러가 비정상 작동을 하거나 임의로 초기화를 하고 싶은 경우 리셋을 동작한다. active HIGH 상티앨 때 칩이 정상 동작하며, 리셋 스위치를 누르면 20번핀이 active LOW 상태가

되어 리셋이 된다.

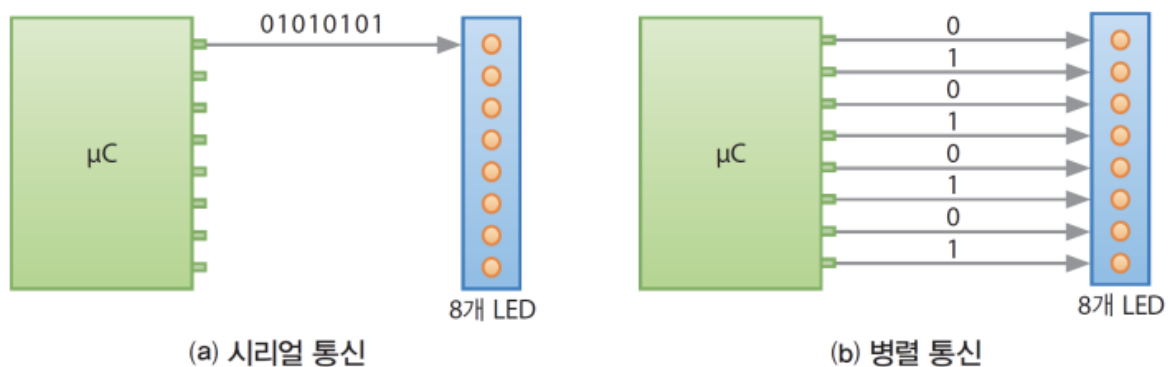
ISP 커넥터 회로: ATmega 칩에 프로그램을 다운로드하기 위한 ISP 다운로더 보드와 aTmega 칩을 연결하는 회로

AD Converter: AVCC, AREF 핀을 통해 기준전압을 입력하는 부분을 위한 AD Converter 회로

[UART 시리얼 통신]

시리얼 통신

마이크로컨트롤러는 비트 단위의 데이터를 핀 단위로 전송한다. 바이트 단위 데이터 전송을 위한 방법으로는 병렬 전송과 직렬 전송이 있다. 병렬 전송은 8개의 핀을 통해 1번에 1바이트 데이터를 전송한다. 연결이 복잡하고 핀수가 제한된 마이크로컨트롤러에서는 핀 부족으로 연결이 불가능할 수 있다. UART 통신은 시리얼/직렬 통신의 한 종류로 직렬 전송은 1개의 핀을 8번에 나누어 1바이트 데이터 전송한다.



UART 통신은 비동기식 통신이고 데이터를 위한 별도의 클록을 사용하지 않으므로 약속된 송수신을 수행한다.

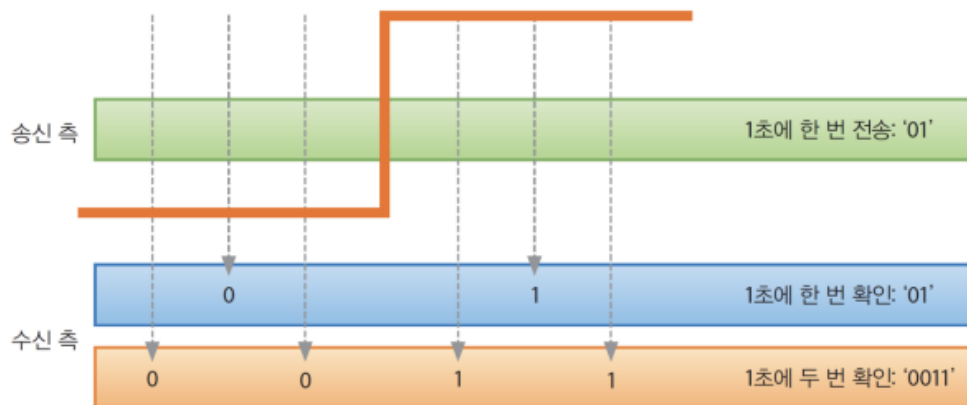


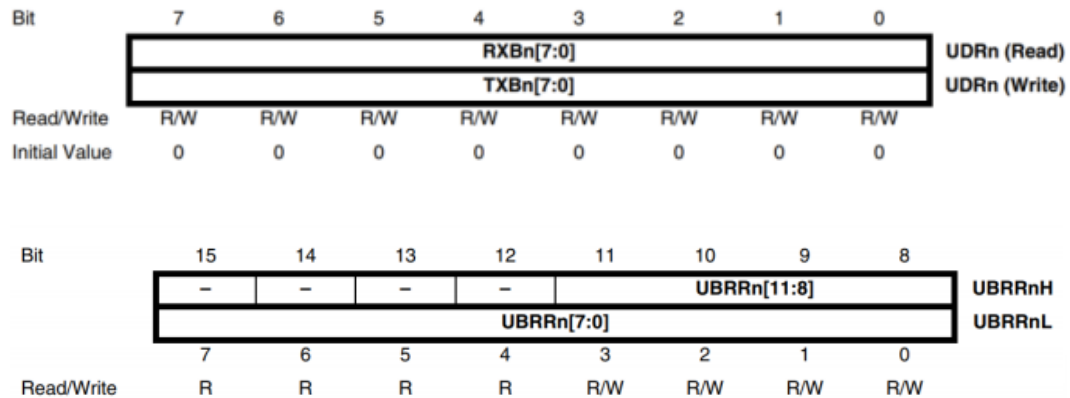
그림 9-2 송신 속도와 수신 속도 차이에 의한 전송 데이터 차이

레지스터

- UDRn: Data Register
- UCSRnA: Control and Status Register A
- UCSRnB: Control and Status Register B
- UCSRnC: Control and Status Register C
- UBRRnH: Baud Rate Register High
- UBRRnL: Baud Rate Register Low

레지스터는 UART의 설정과 사용을 위해 위의 그림과 같이 6개의 레지스터가 주어진다.

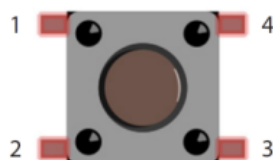
- UDRn은 송수신 데이터를 임시로 저장하는 1byte의 변수 레지스터, UCSRnA, UCSRnB, UCSRnC는 UART의 기능을 설정하거나 각종 현재 상태 등을 모니터링하는 용도로 사용되는 레지스터 묶음, UBRRnH, UBRRnL은 Baud rate를 설정하는 레지스터이다.



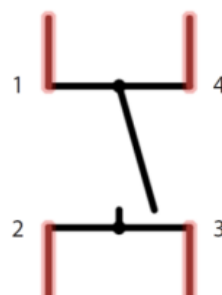
[디지털 데이터 입력]

스위치 입력

- 버튼이 눌려진 경우 입력(GPIO) 핀에는 5V (HIGH)가 가해짐
- 버튼이 눌러지지 않은 경우 입력 핀에 가해지는 값은 알 수 없음
- 회로가 개방된 경우 플로팅(floating)되어 있다고 이야기함
- 회로가 개방(open circuit)되는 경우는 피해야 함



(a) 4핀 푸시 버튼

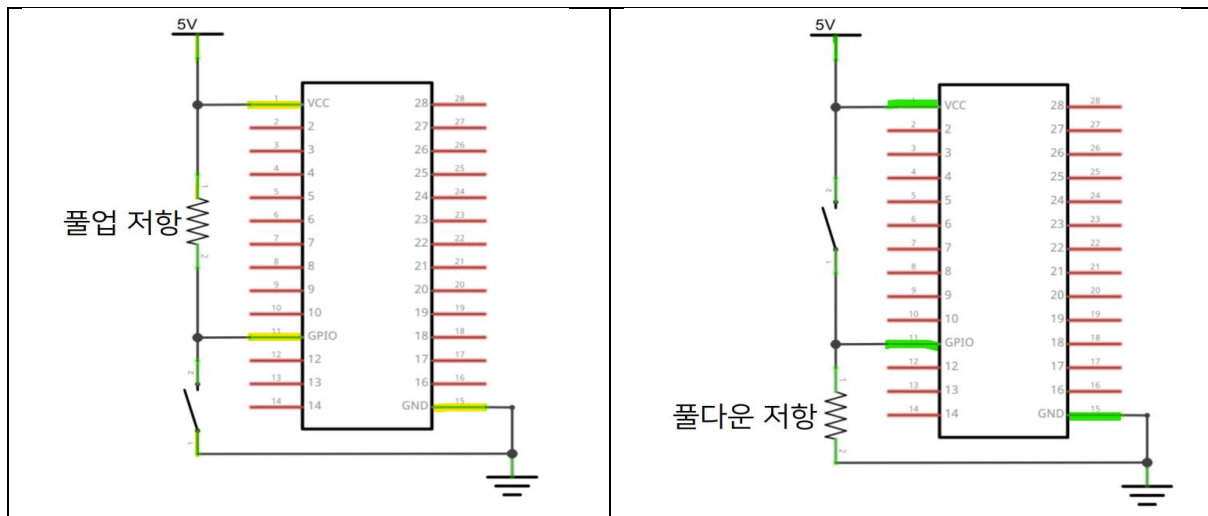


(b) 4핀 푸시 버튼 회로

- ➔ 스위치의 경우 2번과 3번을 연결하게 되면 항상 스위치가 연결되어 있는 상태이기 때문에 스위치를 누르면 켜지고 꺼지게 하는 동작을 수행할 수 없게 된다. 따라서 1과 4, 2 와 3을 연결하지 않아야 한다.

풀업, 풀다운 저항

- 풀업 : 버튼이 눌러지지 않은 경우 입력 핀에 5V(HIGH)가 가해지도록 보장
- 풀다운 : 버튼이 눌러지지 않은 경우 입력 핀에 GND(LOW)가 가해지도록 보장



- 위에 보이는 풀업 저항과 풀다운 저항은 큰 값을 써야한다. 너무 작은 저항을 쓰게 되면 소모되는 전력이 커지게 되어 제품의 효율이 떨어지게 된다.

레지스터

- DDR 핀의 입출력 방향 선택 ▪ 0: 입력, 1: 출력
- 초기값이 0의 의미는 전압이 인가되면 모두 입력으로 설정되어 있는 걸 의미한다.

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

(a) DDRA~DDRF

비트	7	6	5	4	3	2	1	0
비트 이름	-	-	-	DDG4	DDG3	DDG2	DDG1	DDG0
읽기/쓰기	R	R	R	R/W	R/W	R/W	R/W	R/W
초깃값	0	0	0	0	0	0	0	0

(b) DDRG

그림 8-6 DDRx 레지스터의 구조(1은 출력, 0은 입력 상태)

- PINx: 핀으로 입력 받은 데이터 저장

0: LOW, 1: HIGH

- ➔ PINx5에 GPIO에 5V의 신호가 입력이 되면 저절로 1로 바뀌게 된다. 따라서 우리가 원하는 레지스터의 비트를 읽으면 어떤 값인지 알게 된다.

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

(a) PINA~PINF

비트	7	6	5	4	3	2	1	0
비트 이름	PING7	PING6	PING5	PING4	PING3	PING2	PING1	PING0
읽기/쓰기	R	R	R	R	R	R	R	R
초깃값	0	0	0	N/A	N/A	N/A	N/A	N/A

(b) PING

그림 8-7 PINx 레지스터의 구조

- 디지털 데이터 입출력을 위한 레지스터

- LED를 켜고 싶으면 DDR을 출력으로 설정하고 PORT에 값을 쓰면 LED를 켜고 끄고 할 수 있다. 스위치를 입력 받고 싶다면 DDR을 0으로 설정하고 레지스터 PINx 값을 확인을 하면 된다.

내부 풀 업, 외부 풀 업/다운

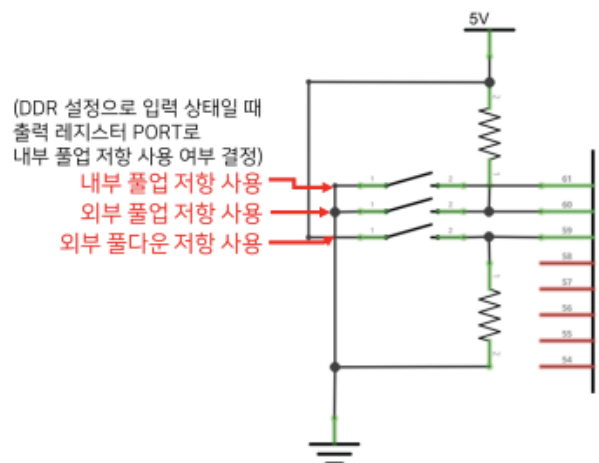


그림 8-9 버튼 3개의 연결 회로도

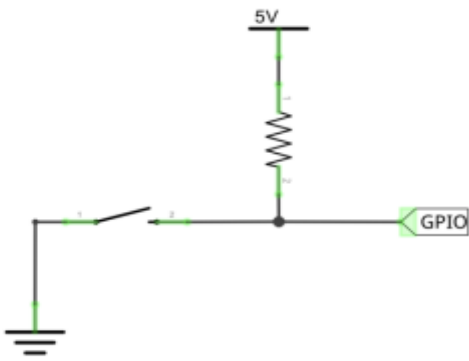
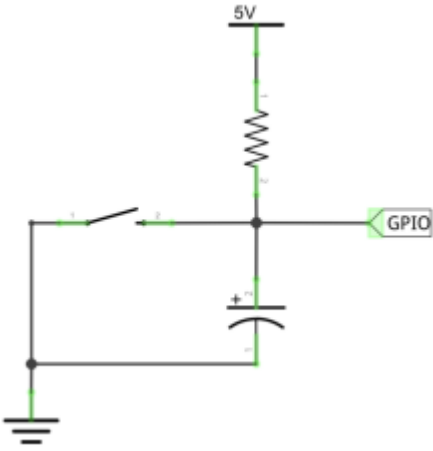
- ATmega에는 내부 풀업 저항이 내장 되어있다. 따라서 굳이 외부 풀업을 달지 않아도 된다.
- 내부 풀업을 사용하고 싶을 때 DDR을 0으로 설정하고 PORT에 1을 설정한다. 값을 읽을 땐 PIN레지스터를 사용한다.

SFIOR 레지스터의 구조								
비트	7	6	5	4	3	2	1	0
비트 이름	TSM	-	-	-	ACME	PUD	PSR0	PSR321
읽기/쓰기	R/W	R	R	R	R/W	R/W	R/W	R/W
초깃값	0	0	0	0	0	0	0	0

PUD(pull-up-disable)비트가 1이면 내부 풀업을 사용할 수 없다.

채터링과 디바운스

- 채터링 : 버튼의 기계적인 진동에 의해 버튼이 완전히 눌러질 때까지 버튼 상태가 빠르게 변하는 현상
- 디바운스: 채터링 현상을 제거하는 방법
- 소프트웨어에 의한 방법: 시간 지연을 통해 버튼의 빠른 상태 변화 무시
- 하드웨어에 의한 방법: 커패시터를 통해 버튼의 빠른 상태 변화 무시
- 디바운스 회로를 포함하는 스위치 입력 회로

풀업 저항을 사용한 버튼 회로	디바운스 회로를 포함하는 버튼 회로1
	

논리값과 전압

- 입력되는 전압의 크기는 0V와 5V의 디지털 값이 아니다.
- 노이즈를 포함하는 이러한 입력값을 0과 1의 디지털 값으로 변환하기 위해 아래의 조건을 적용한다.
- VCC의 값이 크면 클수록 노이즈에 강하다.

[ADC]

아날로그-디지털 변환

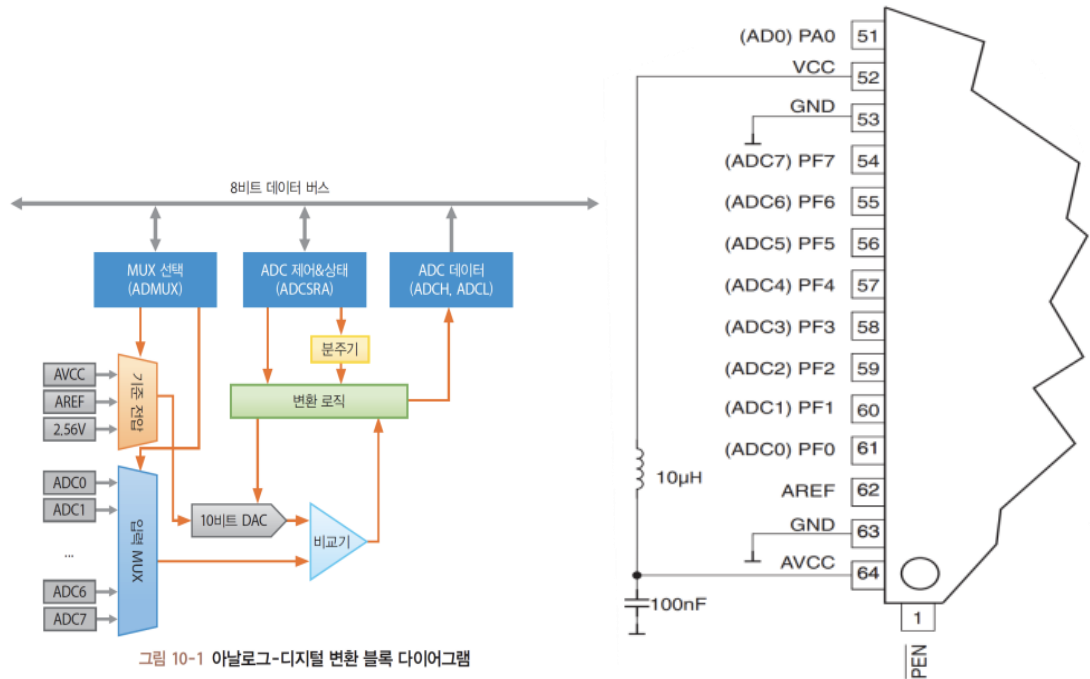
아날로그-디지털 변환이 필요한 이유는 주변의 모든 데이터는 아날로그 데이터이기 때문이며, 마이크로컨트롤러가 처리할 수 있는 데이터는 디지털 데이터이다.

- ATmega128의 아날로그-디지털 변환기(ADC)
 - 10비트 해상도 : 아날로그 전압을 0~1,023 사이 디지털 값으로 변환
 - 1,023으로 변환되는 기준 전압은 AVCC, AREF, 내부 2.56V 중 선택 사용

AVCC는 ADC장치구동전압이다. 아날로그 신호는 신호의 값 자체를 측정하기 때문에 노이즈 타면 성능이 떨어지게 된다. 그래서 별도의 공급전압을 넣어주는 역할을 한다. 이 전압은 5v 또는 3.3v

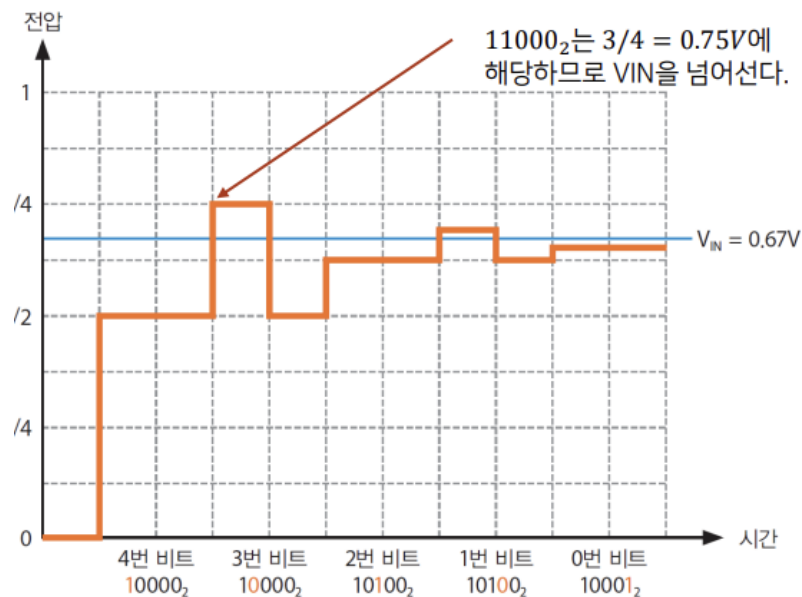
가 된다. 내가 측정하고자 하는 최대 전압값을 정하고자 한다면 그 정한 값을 AREF핀에 넣어주면 된다.

- 8개 채널 제공 : 포트 F, MUX를 통해 연결되고 한 번에 하나씩만 사용 하고 포트F 8개의 핀이 ADC로 연결 되어있다.



아날로그 전압을 디지털 전압으로 바꾸는 방식은 축차 비교 방식 (Successive approximation)이 있다. 축차 비교 방식은 변환될 디지털 값의 모든 비트를 0으로 설정한 후 MSB에서부터 값을 1과 0으로 바꿔가며 입력된 아날로그 전압과 비교하여 값을 결정하는 방식이다.

- 예) 기준전압 1V이고, $V_{IN}=0.67V$ 인 경우 digital 신호는 $10001_2 = 17$ 변환이 된다.



ADC의 성능 지표

- 분해능: 디지털 값 1에 해당하는 아날로그 전압의 차이만큼 측정 가능함.
- 참조 전압이 5V인 경우 $= 5/1024 \approx 0.00488V = 4.89mV$ 따라서 4.89mV보다 작은 값은 무시가 된다.

분해능을 높이는 방법에는 2가지의 방법이 있다.

첫번째 방법은 해상도를 높인다. (별도의 고해상도 ADC 칩이 필요). 이 방법은 가장 좋은 방법이지만 가격이 엄청나게 올라가므로 어렵다. 2번째 방법은 참조 전압을 줄이는 것이다(입력 전압이 작은 범위일 경우 가능)

▪ 변환 속도

- ADC 변환은 순간적으로 이뤄지는 것이 아니고 시간이 필요함.
- 따라서 1초당 최대로 변환할 수 있는 샘플의 수가 정해져 있으므로, 측정하고자 하는 신호의 주파수 제한이 있음. (Nyquist frequency)
- 측정을 세밀하게 빠른 속도로 할 수 있다면 정확하게 측정할 수 있다.

▪ 오차

- ADC의 성능상 오차
- 전원 안정성
- PCB / 도선에 유도된 노이즈
- 양자화 오차

- 10-bit Resolution
- 0.5LSB Integral Non-linearity
- ± 2 LSB Absolute Accuracy
- 13 - 260 μ s Conversion Time
- Up to 76.9kSPS (Up to 15 kSPS at Maximum Resolution)
- 8 Multiplexed Single Ended Input Channels
- 7 Differential Input Channels
- 2 Differential Input Channels with Optional Gain of 10x and 200x
- Optional Left Adjustment for ADC Result Readout
- 0 - VCC ADC Input Voltage Range
- Selectable 2.56V ADC Reference Voltage
- Free Running or Single Conversion Mode
- Interrupt on ADC Conversion Complete
- Sleep Mode Noise Canceler

4

➔ (한 번 측정하는데 13 - 260us가 걸린다.) 이러한 이유는 측차비교 방식을 사용하기 있기 때문이다. (가변적)

해상도가 클수록 변환하는데 시간이 걸리기 때문에 최소 최대 샘플수가 다르다.

▪ 오차 보정

- ADC 회로의 성능 제약으로 인해 여러 오차가 발생하는데, 이를 보정하면 더욱 정확한 아날로그 전압을 측정할 수 있다.

- Offset error

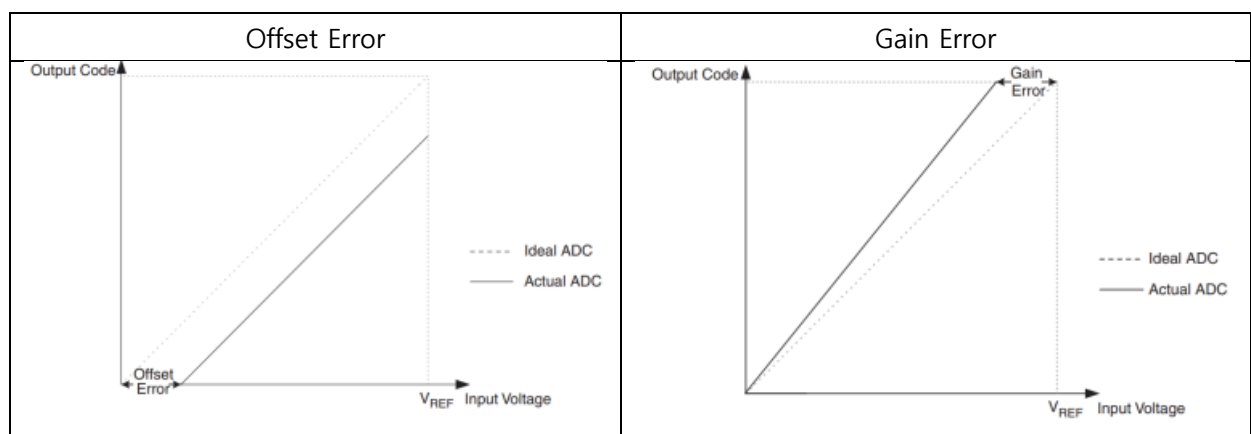
- 매우 작은 입력 전압에 대해 측정값이 발생하지 않는 오차 (x절편에 해당)

- Gain error

- 입력 전압과 디지털 변환값 사이의 기울기 오차

- 입력 전압을 변화시키며 ADC 값을 측정하여 offset error를 보정한 후, gain error를 보정한다

- Offset Error는 낮은 전압 Offset Error와 Gain Error를 개선해주면 성능이 좋아진다.



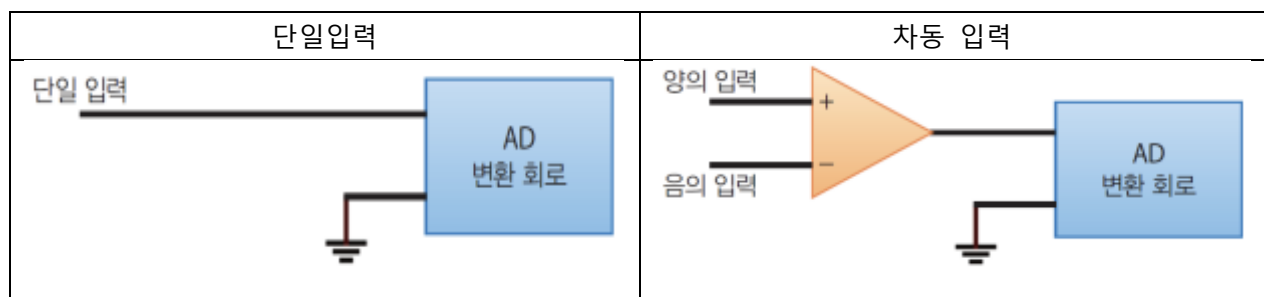
- 단일 입력(single ended input)과 차동 입력(differential input)

- 단일 입력: 1개의 입력 신호의 GND 대비 전위차를 측정

$$ADC = V_{in} \cdot 1024 / V_{ref}$$

- 차동 입력: 2개의 입력 신호의 전위차를 측정

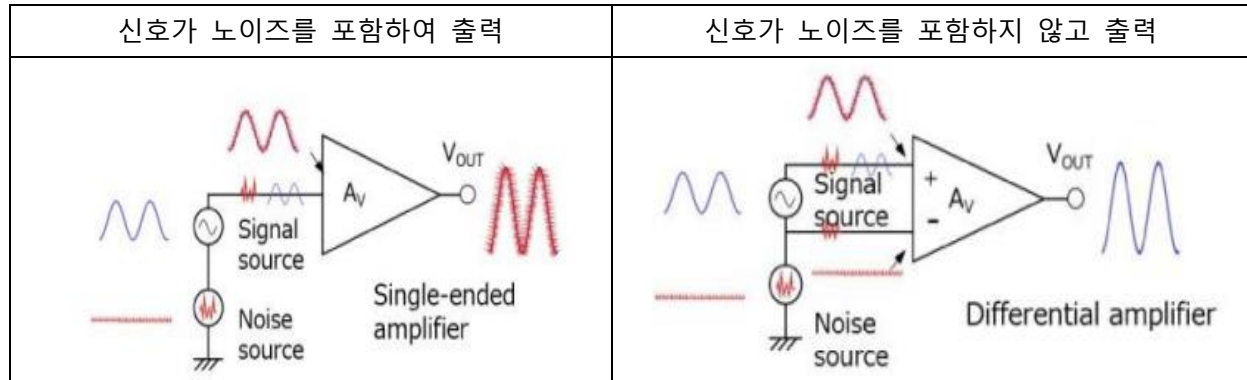
$$ADC = (V_{pos} - V_{neg}) \cdot GAIN \cdot 512 / V_{ref}$$



- 디지털 회로에서 GND는 여러 고주파 신호들로 인해 노이즈를 포함할 가능성이 높다.

- 차동 입력을 통해 공통 모드 노이즈를 효과적으로 저감할 수 있다

- 따라서 노이즈가 심한 환경에서는 차동 입력이 훨씬 유리하다.



레지스터

- ADC의 설정과 상태 모니터링, 변환된 값 측정을 위해 다음의 레지스터가 사용된다.
 - ADMUX: ADC Multiplexer Selection Register
 - ADCSRA: SDC Control and Status Register A
 - ADCH and ADCL: ADC Data Register
- ADCH and ADCL
 - ADC 변환 값은 10bit를 이용해 정수값을 저장하는데 이 저장되는 곳이 ADCH와 ADCL이다. ATmega의 레지스터는 8bit이므로 두 개의 레지스터가 필요하다.
 - ADMUX의 ADLAR bit 설정에 따라 10bit를 정렬하는 순서가 다르다.

ADLAR = 0:

Bit	15	14	13	12	11	10	9	8	
	—	—	—	—	—	—	ADC9	ADC8	ADCH
	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	ADCL
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

ADLAR = 1:

Bit	15	14	13	12	11	10	9	8	
	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADCH
	ADC1	ADC0	—	—	—	—	—	—	ADCL
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

- ADMUX에 5번째 비트에서 ADLAR = 0으로 설정하면 오른쪽 정렬이 되고 ADLAR =1로 설정하면 왼쪽 정렬한 것과 같다.
- ADMUX : ADC의 기준 전압과 입력 채널 등을 선택한다. 그래서 REFS1과 REFS0을 어떤 숫자로 조합하는 지에 따라 기준전압 AVCC, AREF, 2.56 3가지중 하나를 선택할 수 있다.

REFS1	REFS0	Voltage Reference Selection
0	0	AREF, Internal Vref turned off
0	1	AVCC with external capacitor at AREF pin
1	0	Reserved
1	1	Internal 2.56V Voltage Reference with external capacitor at AREF pin

Table 98. Input Channel and Gain Selections

MUX4..0	Single Ended Input	Positive Differential Input	Negative Differential Input	Gain
00000	ADC0	N/A		
00001	ADC1			
00010	ADC2			
00011	ADC3			
00100	ADC4			
00101	ADC5			
00110	ADC6			
00111	ADC7			
01000 ⁽¹⁾	N/A	ADC0	ADC0	10x
01001		ADC1	ADC0	10x
01010 ⁽¹⁾		ADC0	ADC0	200x
01011		ADC1	ADC0	200x
01100		ADC2	ADC2	10x
01101		ADC3	ADC2	10x
01110		ADC2	ADC2	200x
01111		ADC3	ADC2	200x
10000		ADC0	ADC1	1x
10001		ADC1	ADC1	1x
10010		ADC2	ADC1	1x
10011		ADC3	ADC1	1x
10100		ADC4	ADC1	1x
10101		ADC5	ADC1	1x
10110		ADC6	ADC1	1x
10111		ADC7	ADC1	1x
11000		ADC0	ADC2	1x
11001		ADC1	ADC2	1x
11010		ADC2	ADC2	1x
11011		ADC3	ADC2	1x
11100		ADC4	ADC2	1x
11101		ADC5	ADC2	1x
11110	1.23V (V _{BG})	N/A		
11111	0V (GND)			

- 00000~00111은 단일입력으로 몇 번째 핀으로 전압측정을 할 것인지를 나타낸다.
- 01000~11101은 차동 입력으로 +와 -핀을 고를 수 있다.
- 두 신호를 뺀을 때 너무 작을 때 200배 증폭을 해서 ADC변환을 수행한다.
- 11110은 무조건 1.23V를 측정한다. 11111은 무조건 0V를 측정한다.
- 차동 입력 시에는 입력 전압에 상수배하여 작은 신호를 크게 측정하는 기능이 선택 가능함.

- ADCSRA: ADC 상태 표시 및 ADC 제어

Bit	7	6	5	4	3	2	1	0	
	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0	ADCSRA
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	

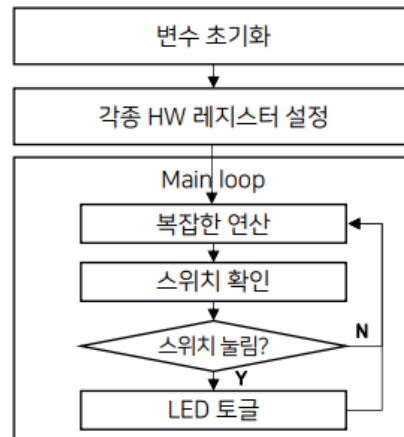
- ADEN: ADC 활성화, 디폴트값은 비활성화 상태이고 ADEN을 1로 설정하면 켜지게 된다.
- ADSC: 변환 시작
 - 단일 변환 모드: AD 변환 시작
 - 프리러닝 모드: 첫 번째 변환 시작
- ADFR: 단일 변환 모드와 프리러닝 모드가 있는데 ADFR을 1로 설정후에 ADSC를 1로 설정하면 계속 변환을 한다.
 - 단일 변환 모드(0): AD 변환 시작 비트가 세트되면 1번 AD 변환 후 종료
 - 프리러닝 모드(1): AD 변환이 시작된 이후, 이전 AD 변환이 끝나면 다음 AD 변환을 자동으로 시작
- ADIE: AD 변환이 종료될 때 인터럽트 발생 허용
- ADPSn: AD 변환을 위한 분주비 설정
 - ADC를 위한 주파수는 50~200KHz가 추천됨

ADPS2	ADPS1	ADPS0	Division Factor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

- ➔ 클럭을 최대 2에서 128까지 나눌 수 있다는 것을 알 수 있다. 예를 들어 ADPS2 ADPS1 ADPS0을 각 1로 설정해서 분주비가 128인 경우 ADC에 입력되는 클럭이 $16\text{MHz} / 128 = 125\text{KHz}$ 입력이 될 것이다. 약 13클럭 당 1 샘플을 측정할 수 있으므로, 초당 $125000 / 13 \approx 9615\text{SPS}$ 이다. 즉 분주비가 낮을 수록 더 빨리 측정이 가능하다. 하지만 노이즈에 취약하다.

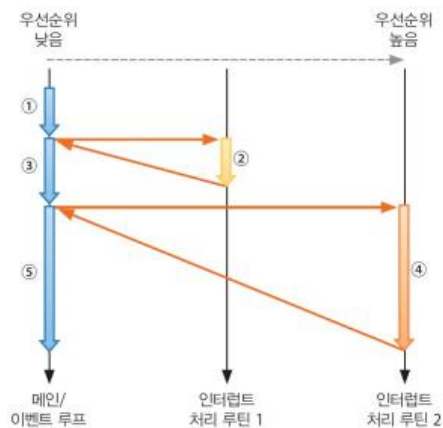
[인터럽트]

폴링



위의 그림과 같이 프로세서의 코드는 순차적으로 시행되는 것을 폴링(polling)이라고 정의한다. 위의 예에서 복잡한 연산이 오래 걸리면, 그 시간만큼 스위치가 눌렸다는 것을 알지 못한다. 다른 작업이 얼마나 걸리든지 스위치가 눌린 즉시 어떤 동작을 하게 하고 싶다면 인터럽트(interrupt) 사용하면 된다.

인터럽트



위의 그림과 같이 동시에 여러 개의 인터럽트가 발생하면 우선 순위가 높은 인터럽트를 우선 처리하게 해 주는 것을 인터럽트(interrupt)라고 정의한다. 마이크로컨트롤러가 특정 작업을 즉시 처리하도록 요구하는 특정한 사건으로 우선 순위가 낮은 프로세서는 멈추고 우선 순위가 높은 인터럽트 처리 루틴이 실행되어 인터럽트를 우선 처리한다. 즉, 인터럽트가 발생하면 현재 실행 중인 코드를 정지하고 인터럽트 서비스 루틴(ISR)으로 즉시 이동하여 인터럽트를 먼저 처리해주는 것이다. 하드웨어 의해 호출되는 함수로 이해할 수 있다. 최대 35개의 인터럽트가 사용 가능하다.

- 인터럽트 종류와 인터럽트 벡터 테이블

표 12-1 ATmega128의 인터럽트 벡터 테이블

벡터 번호	프로그램 주소	소스	인터럽트 정의	벡터 번호	프로그램 주소	소스	인터럽트 정의
1	0x0000	RESET	External Pin, Power-on Reset, Brown-out Reset, Watchdog Reset, JTAG AVR Reset	17	0x0020	TIMER0 OVF	Timer/Counter0 Overflow
2	0x0002	INT0	External Interrupt Request 0	18	0x0022	SPI STC	SPI Serial Transfer Complete
3	0x0004	INT1	External Interrupt Request 1	19	0x0024	USART0 RX	USART0 Rx Complete
4	0x0006	INT2	External Interrupt Request 2	20	0x0026	USART0 UDRE	USART0 Data Register Empty
5	0x0008	INT3	External Interrupt Request 3	21	0x0028	USART0 TX	USART0 Tx Complete
6	0x000A	INT4	External Interrupt Request 4	22	0x002A	ADC	ADC Conversion Complete
7	0x000C	INT5	External Interrupt Request 5	23	0x002C	EE READY	EEPROM READY
8	0x000E	INT6	External Interrupt Request 6	24	0x002E	ANALOG COMP	Analog Comparator
9	0x0010	INT7	External Interrupt Request 7	25	0x0030	TIMER1 COMP	Timer/Counter1 Compare Match C
10	0x0012	TIMER2 COMP	Timer/Counter2 Compare Match	26	0x0032	TIMER3 CAPT	Timer/Counter3 Capture Event
11	0x0014	TIMER2 OVF	Timer/Counter2 Overflow	27	0x0034	TIMER3 COMPA	Timer/Counter3 Compare Match A
12	0x0016	TIMER1 CAPT	Timer/Counter1 Capture Event	28	0x0036	TIMER3 COMPB	Timer/Counter3 Compare Match B
13	0x0018	TIMER1 COMPA	Timer/Counter1 Compare Match A	29	0x0038	TIMER3 COMPC	Timer/Counter3 Compare Match C
14	0x001A	TIMER1 COMPB	Timer/Counter1 Compare Match B	30	0x003A	TIMER3 OVF	Timer/Counter3 Overflow
15	0x001C	TIMER1 OVF	Timer/Counter1 Overflow	31	0x003C	USART1 RX	USART1 Rx Complete
16	0x001E	TIMER0 COMP	Timer/Counter0 Compare Match	32	0x003E	USART1 UDRE	USART1 Data Register Empty
				33	0x0040	USART1 TX	USART1 Tx Complete
				34	0x0042	TWI	Two-wire Serial Interface
				35	0x0044	SPM READY	Store Program Memory Ready

위의 그림은 인터럽트 종류와 벡터 테이블로 인터럽트가 발생하였을 때 처리가 옮겨질 해당 ISR의 Interrupt, service, routine 함수의 메모리 주소이다. 35개의 인터럽트에 대한 인터럽트 벡터를 모아 놓은 테이블로 플래시 메모리의 0x0000 번지에서 0x0045 번지까지 기록되어 있다. ATmega128에서 자주 사용되는 프로그램 주소 0x002~0x0010은 INT0~INT7의 소스이고, 0x0024~28, 0x003C~O은 USART 소스이다.

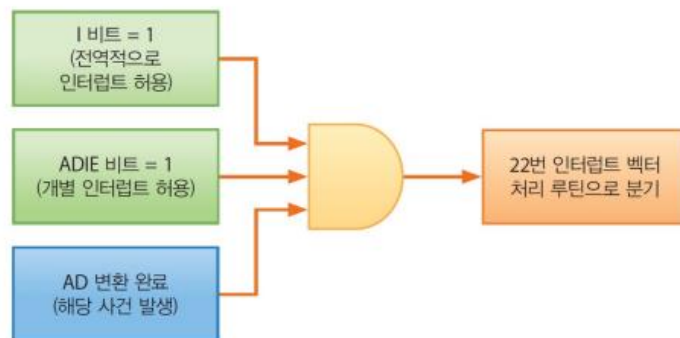


그림 12-3 AD 변환 완료 인터럽트 처리

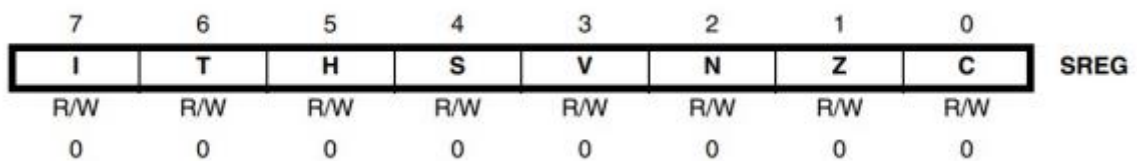
위의 그림은 인터럽트가 처리되기 위한 조건으로 전역적 인터럽트 비트 세트, 개별 인터럽트 활성화 비트 세트, 인터럽트 발생 조건 충족이 되어야 한다. 상태 레지스터 SERG의 7번 'I' 비트 세트는 1로, 세트를 위해 sei() 함수를 사용하고, 개별 인터럽트 허용하여 AD 변환 완료가 되도록 인터럽트 처리를 해준다. 인터럽트가 발생하였을 때 처리 루틴으로 ISR 함수를 이용하여 하드웨어에 의해 호출을 해준다. ISR은 코드 내에서 호출되는 부분이 없으므로 ISR은 메인 코드와 무관한 코드로 간주될 수 있기에 최적화 방지의 필요성으로 ISR에서 값을 변경하는 변수는 'volatile' 키워드를 사용하여 준다.

ISR

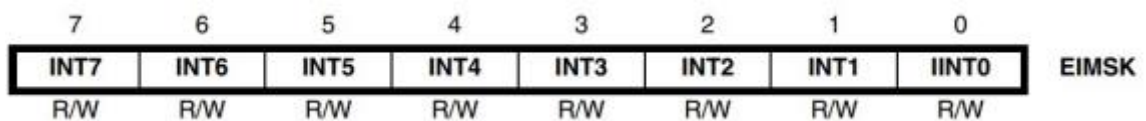
ISR(Interrupt Service Routine)은 인터럽트가 발생하였을 때 처리 루틴이며 모든 ISR은 동일한 이름을 가진다. 처리할 인터럽트의 종류를 인터럽트 벡터 이름인 매개변수로 구분하며 우리가 사용한 인터럽트 벡터 이름은 다음과 같다.

USART1_RX_vect	USART1 Tx complete
TIMER1_COMPA_vect	Timer/counter1 compare Match
TIMER0_OVF_vect	Timer/Counter0 Overflow

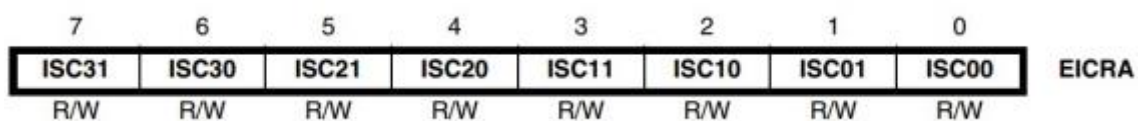
레지스터



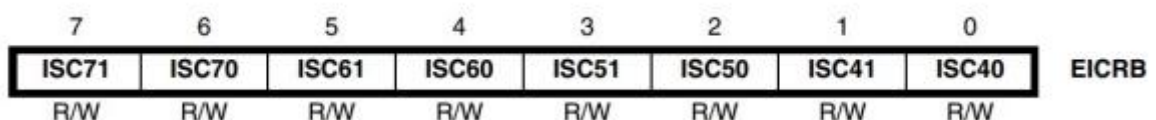
- ➔ 외부 인터럽트를 사용하기 위해서는 SERG, EIMSK, EICRA, EICRB 레지스터를 설정해야 한다. 전역 인터럽트 활성을 위해 SERG 레지스터에서 7번째 비트 'I' 비트 세트는 1로 설정해준다.



- ➔ 어떤 핀에 입력되는 인터럽트를 활성화할 것인가를 설정하는 레지스터로, 1으로 set된 pin은 외부 인터럽트 동작이 활성화되어 EIMSK 레지스터의 0번째 비트 'INT0' 비트 세트는 1으로 설정해준다.



- ➔ 각 핀에 어떤 신호가 입력되었을 때 인터럽트가 걸릴 지를 설정하는 레지스터로, 각 핀 별로 2개의 bit가 필요하여 총 2byte의 레지스터가 요구되므로 EICRA와 EICRB로 나누어졌다. EICRA 레지스터의 3번째 비트의 'ISC11' 비트 세트는 1로 설정해준다.

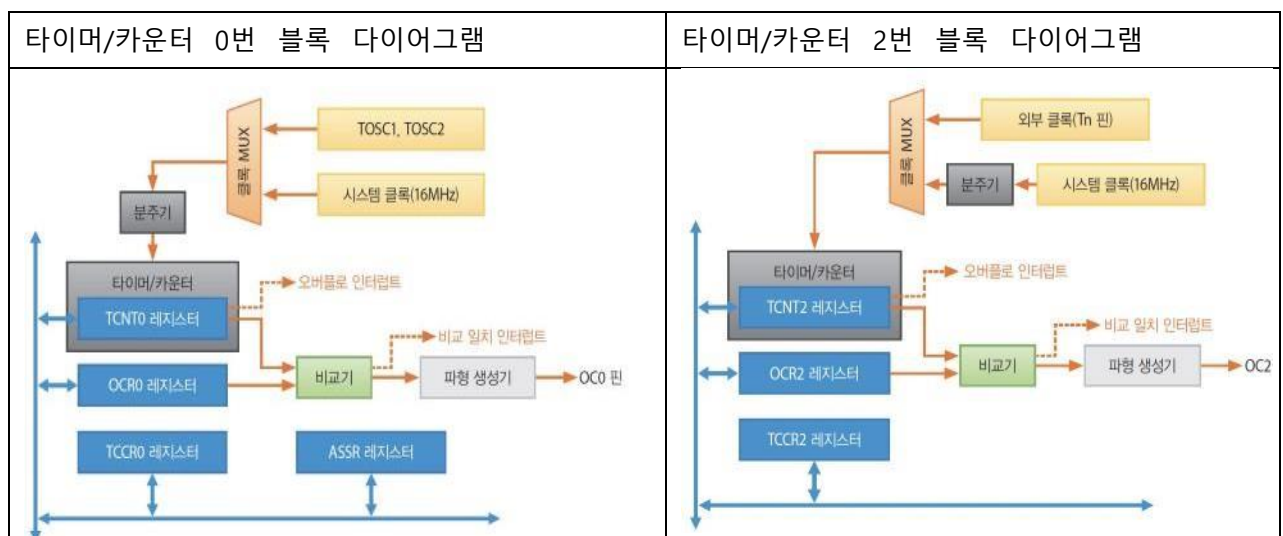


- ➔ 위의 레지스터는 EICRA 레지스터와 마찬가지로 각 핀에 어떤 신호가 입력되었을 때 인터럽트가 걸릴 지를 설정하는 레지스터이다. EICRB 레지스터의 7번째 비트의 'ISC71' 비트 세트는 1로 설정해준다. 세트를 위해 인터럽트 허용할 때 sei() 함수를 사용해주고, 클리어를 위해 인터럽트를 금지할 때 cli() 함수를 사용해주다.

[8비트 카운터/타이머]

타이머/카운터 모듈

- 타이머 /카운터 모듈은 어떤 동작을 수행한 시간을 측정해서 일정한 주기마다 혹은 특정 시점에 동작 (알람, 제어 시스템)등을 한다.
 - 타이머/카운터가 시간을 측정하는 방법은 프로세서에 일정하게 입력되는 클럭 펄스수를 카운트한다. 즉 일정한 주기의 펄스를 샘플로써 측정, 즉 타이머 역할 수행이 가능하다.
 - Atmega128 은 4개의 타이머/카운터를 제공하며 그 종류는 0번과 2번의 8비트 타이머/카운터, 1번과 3번의 16비트 타이머/카운터이다.
 - 8비트 타이머/카운터는 0~255까지 (8bit) 세기를 반복하고 256번째는 0이 된다. 16MHz 시스템 클럭을 사용하는 경우 256개 클럭의 발생 시간은 0.016ms이지만 우리는 1초에 한 번씩 기능을 구현해야 하므로 분주기를 통해 클럭의 속도를 늦춤으로써 보다 긴 시간 측정 가능하다.
- ➔ ATmega128은 두 개의 8bit 타이머/카운터 모듈 (0번과 2번)이 있는데 기능상으로 거의 비슷하지만, 약간의 차이가 있다. 즉 펄스를 재는 종류가 다르다. 0번은 Mux를 통해 하나의 클럭을 결정하고 분주기를 통해 클럭 수를 줄여서 TCNT0 레지스터에 센 값을 저장해서 인터럽트를 발생시킨다. 2번은 분주기가 MUX 안에 들어있고 시스템 클럭과 연결되어 있다. 별도의 크리스탈로 만든 펄스를 측정할 수 있으며, 외부 클럭(Tn핀)을 통해 스위치를 몇 번 눌렀는지를 카운터 해주는 모듈이다. 즉 펄스가 몇 번 들어왔는지 측정해 주는 것이다.



- 동작 원리

1. 클럭의 주파수를 분주기를 이용해 줄인다.
2. 분주기를 거친 한 클럭 펄스마다 TCNTn 레지스터의 값이 1씩 증가한다.
3. TCNTn 레지스터와 값과 사전에 정한 OCRn 레지스터 값을 비교한다.
4. 두 값이 같으면 비교 일치 인터럽트(Compare match interrupt)를 발생시키는데 설정된 동작에 따라 OCn 핀의 출력값을 토글 시켜서 일정한 펄스를 내보낼 수 있다. 또한 설정된 동작에 따라 TCNTn 레지스터의 값을 0으로 초기화할 수도 있다.
5. TCNTn 레지스터의 값이 255를 넘게 되면, TCNTn 레지스터의 값을 0으로 초기화하고 오버플로 인터럽트(Overflow interrupt)를 발생시킨다. 따라서 OCn 값을 설정한 것과 같아질 때 인터럽트가 걸리고 255 넘어서 0으로 떨어질 때 인터럽트가 걸려서 총 2번이 걸리게 된다.

입력 클럭의 주파수와 분주기 설정값에 의해 TCNTn이 증가하는 속도가 일정하다.

OCRn의 값은 0~255 중에 원하는 값으로 정할 수 있기 때문에 overflow interrupt나 compare match interrupt가 걸리는 주기를 결정할 수 있다. 그래서 interrupt가 걸린 횟수를 측정하면 얼마든지 원하는 주기 마다 동작하는 기능을 구현할 수 있다. 또한 OCn 핀이 일정 주기마다 변하므로 일정 주기 마다 외부에 신호를 줄 수도 있다.

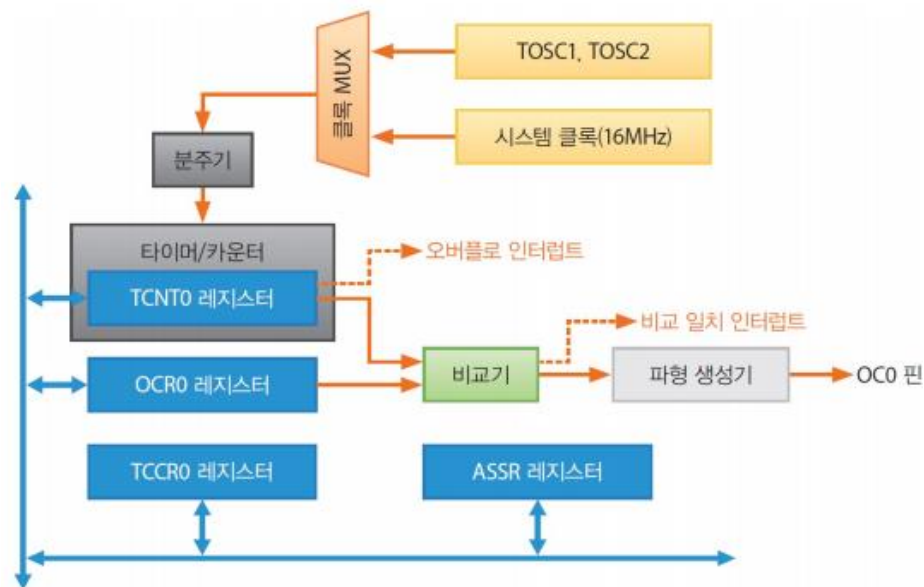
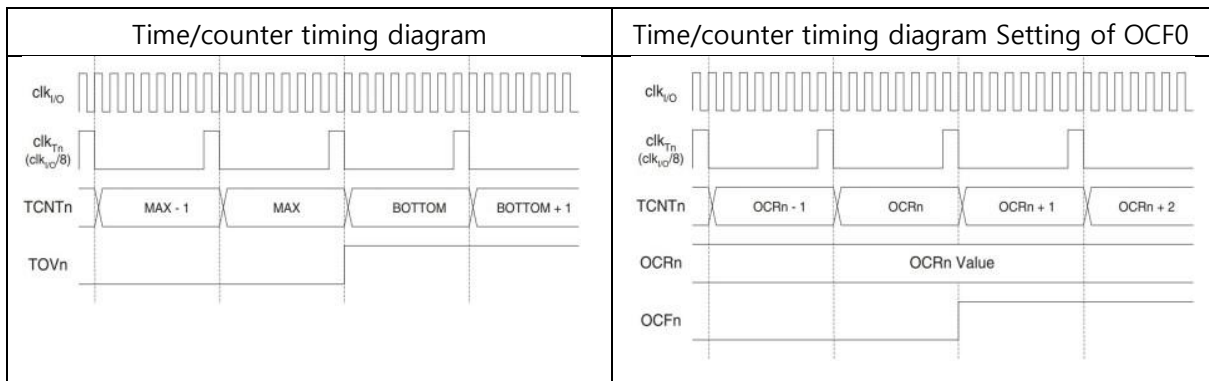


그림 13-1 타이머/카운터 0번 블록 다이어그램

인터럽트

현재까지의 펄스 수는 TCNTn 레지스터에 저장됨

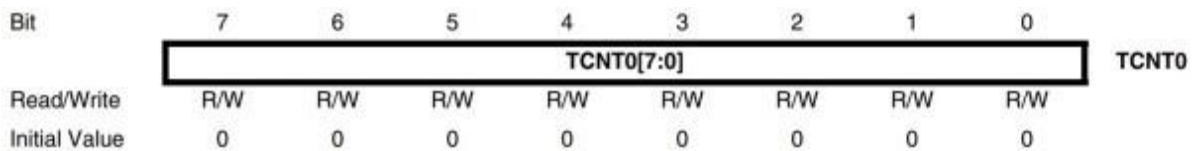
- Overflow interrupt : 최대로 셀 수 있는 펄스의 수를 넘어설 때 TCNTn 레지스터가 0으로 바뀌면서 발생
- Compare match interrupt : TCNTn의 값이 미리 설정된 OCRn 레지스터의 값과 일치하는 경우 발생하며 비교 일치 인터럽트가 발생할 때 OCFn 핀을 통해 파형 출력 가능



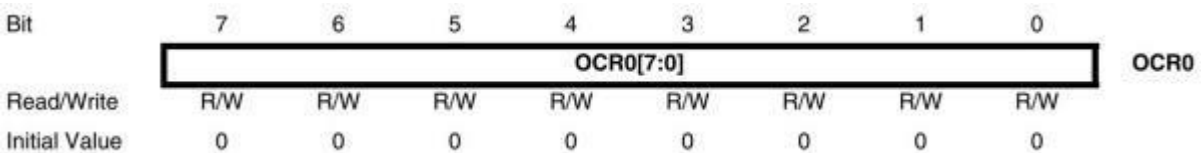
레지스터

8bit 타이머/카운터 사용에는 다음의 레지스터가 사용된다. (N=0,2)

- TCNTn : 입력되는 클럭 마다 타이머/카운터 모듈이 자동으로 1씩 증가시키는 1byte 레지스터



- OCRn : Compare match interrupt를 위해 사용자가 설정하는 레지스터로 0~255 사이의 값을 할당가능하고 TCNTn 레지스터의 값과 비교되어 일치 여부를 판단한다.



- TCCRn : 타이머/카운터 모듈의 동작을 설정하는 레지스터

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	

- FOCn: Force output compare의 용도로 사용되지만 특별히 사용되는 경우가 없으므로 0으로 설정한다.
- WGM : Waveform generation mode로 타이머/카운터를 어떤 모드로 동작을 시킬것인지 결정하는 레지스터이다.

Mode	WGM01 ⁽¹⁾ (CTC0)	WGM00 ⁽¹⁾ (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

- ➔ 타이머/카운터 용도로는 compare match가 됐을 때 0으로 초기화를 시키지 않는 normal 모드와 compare match가 됐을 때 0으로 초기화 시키는 CTC(Clear Timer on Compare match) 모드가 사용된다. 즉 Normal 모드에서 TCNTn은 0에서 255까지 증가하고 256째에 overflow interrupt와 같이 0으로 초기화 된다.

CTC 모드에서 TCNTn은 OCRn에 설정된 값까지 증가한 후 그 다음 번째에 0으로 초기화 된다.

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	

- ➔ COM(compare match output Mode): Compare match가 일어났을 때 OCn핀의 출력을 어떻게 설정할 것인지를 지정한다. 즉 토글을 시키거나 0으로 값을 떨어뜨리거나 1로 올릴 수 있다.

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

Fast PWM Mode		
---------------	--	--

COM01	COM00	Description
0	0	Normal port operation, OC0 disconnected.
0	1	Reserved
1	0	Clear OC0 on compare match, set OC0 at BOTTOM, (non-inverting mode)
1	1	Set OC0 on compare match, clear OC0 at BOTTOM, (inverting mode)

→ CS(clock Select) : 분주비를 설정한다. 아래 그림을 보게되면 CS02 CS01 CS00에 값을 상황에 따라넣어주면 분주를 하지 않거나 8~1024에서 정해진 값을 사용할 수 있다.

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

Clock select bit description▪

- TIMSK : 어떤 interrupt를 활성화할 것인지를 설정한다.

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	

->

표 13-2 타이머/카운터 0번 및 2번과 관련된 인터럽트

벡터 번호	인터럽트	벡터 이름	인터럽트 허용 비트	
10	비교 일치 인터럽트	TIMER2_COMP_vect	OCIE2	Timer/Counter 2 Output Compare Match Interrupt Enable
11	오버플로 인터럽트	TIMER2_OVF_vect	TOIE2	Timer/Counter 2 Overflow Interrupt Enable
16	비교 일치 인터럽트	TIMER0_COMP_vect	OCIE0	Timer/Counter 0 Output Compare Match Interrupt Enable
17	오버플로 인터럽트	TIMER0_OVF_vect	TOIE0	Timer/Counter 0 Overflow Interrupt Enable

➔ 위의 그림을 보면 인터럽트 허용 비트 이름이 보인다. OCIE0은 compare match interrupt를 활성화 시키고 TOIE0은 over flow interrupt를 활성화시킨다.

- ASSR : 타이머/카운터 모듈 0의 입력으로 외부 저주파 크리스탈을 사용할 수 있도록 허용한다.

Bit	7	6	5	4	3	2	1	0	
	-	-	-	-	AS0	TCN0UB	OCR0UB	TCR0UB	ASSR
Read/Write	R	R	R	R	R/W	R	R	R	
Initial Value	0	0	0	0	0	0	0	0	

➔ 시스템 클럭이나 저주파 크리스탈을 MUX로 선택하기 위한 레지스터 AS0비트가 0으로 입력될 때, 타이머/카운터 모듈 0은 I/O clock, MAIN 클럭 clk(I/O)로부터 클럭되고 1이면TOSC1에 연결된 저주파 크리스탈로 클럭이 들어간다. 저주파 크리스탈을 이용하는 이유는 정확하게 1초를 만들지 못하기 때문이다. 32.768KHz인 이유는 32768을 128으로 분주할 경우 정확하게 1초 마다 over flow interrupt가 걸리기 때문이다. 하지만 주변온도, 입 력전압에 따라 클럭주기가 바뀌기 때문에 완벽한 타이머를 맞추는데 한계가 있다.

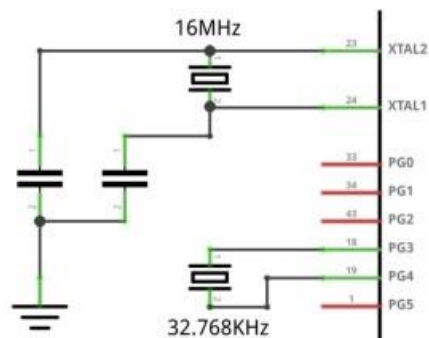


그림 13-9 시스템 클럭과 0번 타이머/카운터를 위한 크리스탈 연결

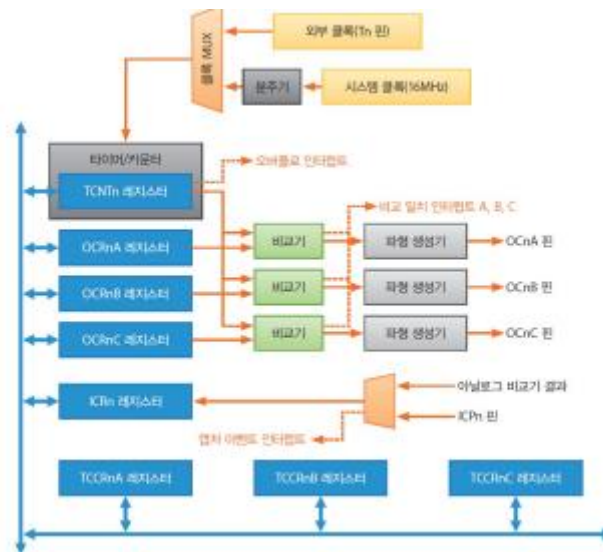
$32768 = 2^{15}$ 이므로 128으로 분주할 경우 정확하게 1초 마다 OVF interrupt가 걸릴 수 있다.

[16비트 카운터/타이머]

타이머/카운터 모듈

8bit 타이머/카운터 모듈인 0번과 2번 모듈은 TCNT 레지스터가 8Bit이므로 인터럽트 주기가 매우 짧다는 한계를 가진다. 하지만 1번과 3번 타이머/카운터 모듈은 16bit 타이머/카운터 모듈이다. 즉, TCNT 레지스터가 16bit로 구성된다. 8bit에 비해서 단순히 bit수가 두 배라고 생각 할 수 있지만 그렇지 않으며, 256배 큰 값으로 2^{16} 까지 카운트가 가능하다. 그러므로 인터럽트의 주기를 훨씬 길게 가져갈 수 있다. 또한, 타이머 하나에 최대 세개의 OCR 값을 지정하여 서로 다른 주기의 compare match interrupt를 발생시킬 수 있다.

오버플로 인터럽트는 $0 \sim 2^{16} - 1 = 65535$ 까지 카운터가 가능하며 세 개의 비교 일치 인터럽트 사용이 가능하다. 세 개의 서로 다른 비교 일치 값 설정을 위한 OCRnX 레지스터가 있으며 세 개의 파형 출력 핀이 존재한다. 입력 캡처는 특정 사건이 발생한 경우 현재 카운터 값인 TCNTn 레지스터 값을 저장하는 역할을 한다. 아래는 16비트 타이머/카운터 블록 다이어그램을 나타낸 것이다.



레지스터

- 16bit 타이머/카운터 사용에는 다음의 레지스터가 사용된다. (N=1,3)
- TCNT1n; 16bit 타이머/카운터 모듈이므로 2byte 크기를 가진다. 다만, 프로세서가 8bit로 동작하므로, 두개의 1byte 레지스터 (RCNTnH와 TCNTnL)을 묶어서 TCNTn으로 관리한다.

Bit	7	6	5	4	3	2	1	0	
	TCNT1[15:8]								TCNT1H
	TCNT1[7:0]								TCNT1L
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	

- OCRnA~C : 총 세 개의 OCR 레지스터가 있어서, TCNTn이 각 OCRnX과 같을 때 각자의 compare match interrupt가 발생한다. TCNT와 비교하기 위해 OCR 역시 16bit로 구성되며, 2byte 레지스터 두개를 묶어서 관리한다. OCRnA, OCRnB, OCRnC가 존재한다.

Bit	7	6	5	4	3	2	1	0	
	OCR1A[15:8]								OCR1AH
	OCR1A[7:0]								OCR1AL
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	
Bit	7	6	5	4	3	2	1	0	
	OCR1B[15:8]								OCR1BH
	OCR1B[7:0]								OCR1BL
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	
Bit	7	6	5	4	3	2	1	0	
	OCR1C[15:8]								OCR1CH
	OCR1C[7:0]								OCR1CL
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	

- TIMSK와 ETIMSK : 특정 interrupt를 허용하기 위한 설정 mask이며, 16bit 타이머/카운터는 크게 세 가지 interrupt가 가능하다. 세 가지 interrupt는 아래와 같다.

➔ Overflow interrupt는 TCNT가 overflow될 때 발생하며, compare match interrupt는 TCNT가 OCRx (x=A,B,C)중 하나와 일치할 때 발생, input capture interrupt는 특정 조건이 만족되었을 때 발생하는데 이때 특정 조건이란 ICPn(n=0,1)핀에 상승/하강 edge가 감지 되었을 대나 아날로그 비교기가 특정 조건을 만족했을 때를 말한다.

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	
Bit	7	6	5	4	3	2	1	0	
	-	-	TICIE3	OCIE3A	OCIE3B	TOIE3	OCIE3C	OCIE1C	ETIMSK
Read/Write	R	R	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	



Table 61. Waveform Generation Mode Bit Description

Mode	WGMn3	WGMn2 (CTCn)	WGMn1 (PWMn1)	WGMn0 (PWMn0)	Timer/Counter Mode of Operation ⁽¹⁾	TOP	Update of OCRnx at	TOVn Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCRnA	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICRn	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCRnA	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICRn	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCRnA	TOP	BOTTOM
12	1	1	0	0	CTC	ICRn	Immediate	MAX
13	1	1	0	1	(Reserved)	–	–	–
14	1	1	1	0	Fast PWM	ICRn	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCRnA	BOTTOM	TOP

Table 62. Clock Select Bit Description

CSn2	CSn1	CSn0	Description
0	0	0	No clock source. (Timer/Counter stopped)
0	0	1	clk _{ICU} /1 (No prescaling)
0	1	0	clk _{ICU} /8 (From prescaler)
0	1	1	clk _{ICU} /64 (From prescaler)
1	0	0	clk _{ICU} /256 (From prescaler)
1	0	1	clk _{ICU} /1024 (From prescaler)
1	1	0	External clock source on Tn pin. Clock on falling edge
1	1	1	External clock source on Tn pin. Clock on rising edge

8

[Character LCD]

Character Lcd

- Character (문자) LCD 또는 Text LCD 라고 불린다.
- 3개의 제어 신호와 8개의 데이터 신호를 이용해 조작할 수 있다. 데이터 신호는 8개 전부(8bit 모드) 또는 상위 4개만 사용(4bit 모드) 가능하다.
- 텍스트 LCD RS, R/W, EN핀은 출력할 문자에 해당하는지 결정하고 보내고자 하는 데이터를 설정하고 데이터를 LCD에게 명령하는 제어 핀들이다.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	40	
1행	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	10	...	27
2행	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F	50	...	67

- Character LCD는 자체적으로 화면에 표시할 문자열을 저장할 메모리를 가지고 있다.
 - ➔ 16*2 LCD (한 줄 당 16개 문자, 2줄 표현) 인 경우 실제로 표현되는 문자의 개수는 32개이다. 그러나 내부에는 총 80개의 문자를 지정할 수 있는 메모리 공간이 있다. 따라서 1행 1열의 메모리 주소는 0x00이지만, 2행 1열의 메모리 주소는 0x40이다.
 - 텍스트 LCD 모듈의 명령어

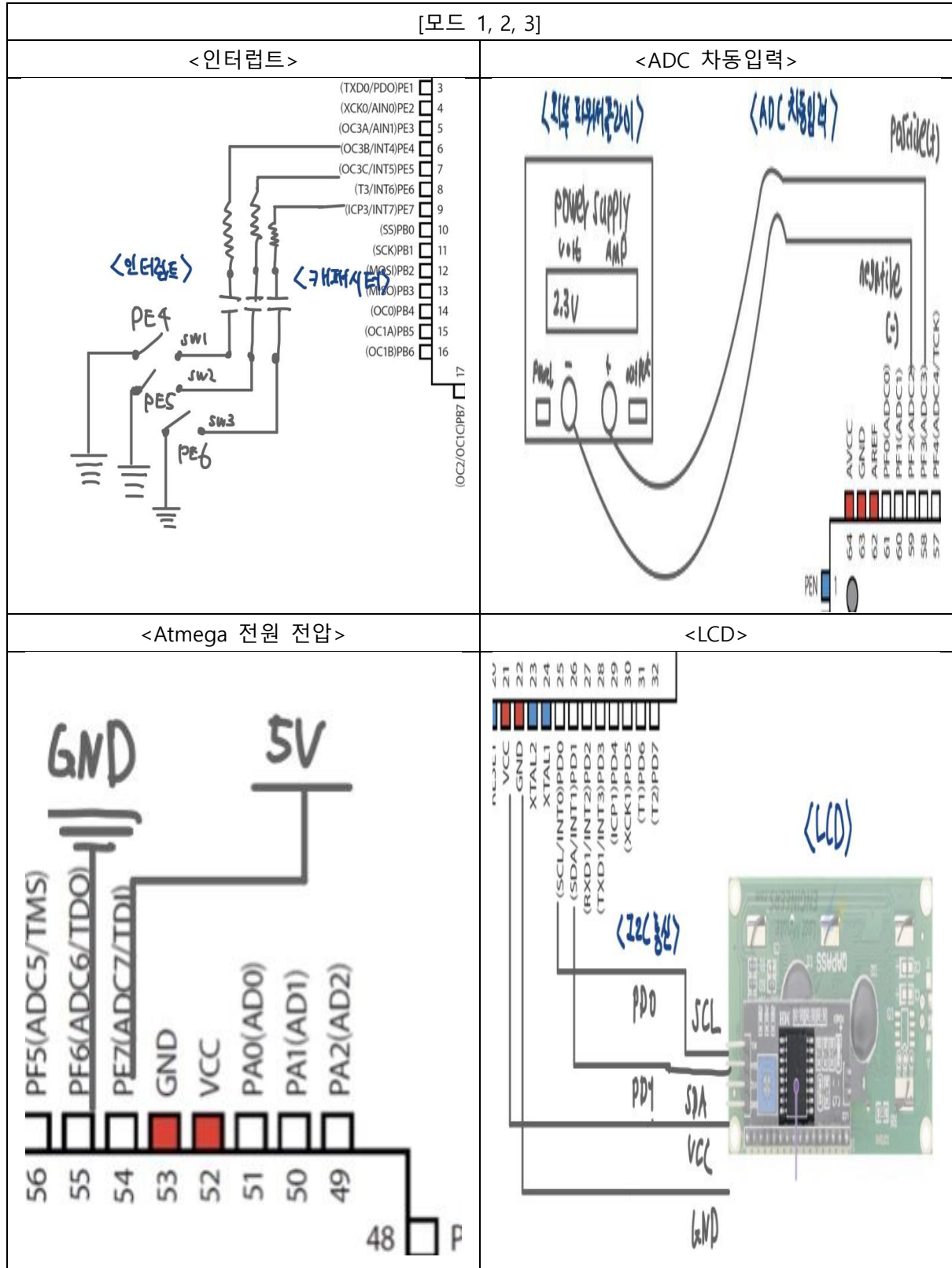
명령	명령 코드									
	RS	R/W	DB7	DB6	DB5	DB4	DB3	DB2	DB1	DB0

Clear Display	0	0	0	0	0	0	0	0	0	1
Retrun Home	0	0	0	0	0	0	0	0	1	0
Entry Mode Set	0	0	0	0	0	0	0	1	I/D	S
Display on/off control	0	0	0	0	0	0	1	D	C	B
Cursor or Display shift	0	0	0	0	0	1	S/C	R/S	-	-
Function Set	0	0	0	0	1	DL	N	F	-	-
Set CGRAM	0	0	0	1	AC5	AC4	AC3	AC2	AC1	AC0
Set DDRAM Address	0	0	1	AC6	AC5	AC4	AC3	AC2	AC1	AC0

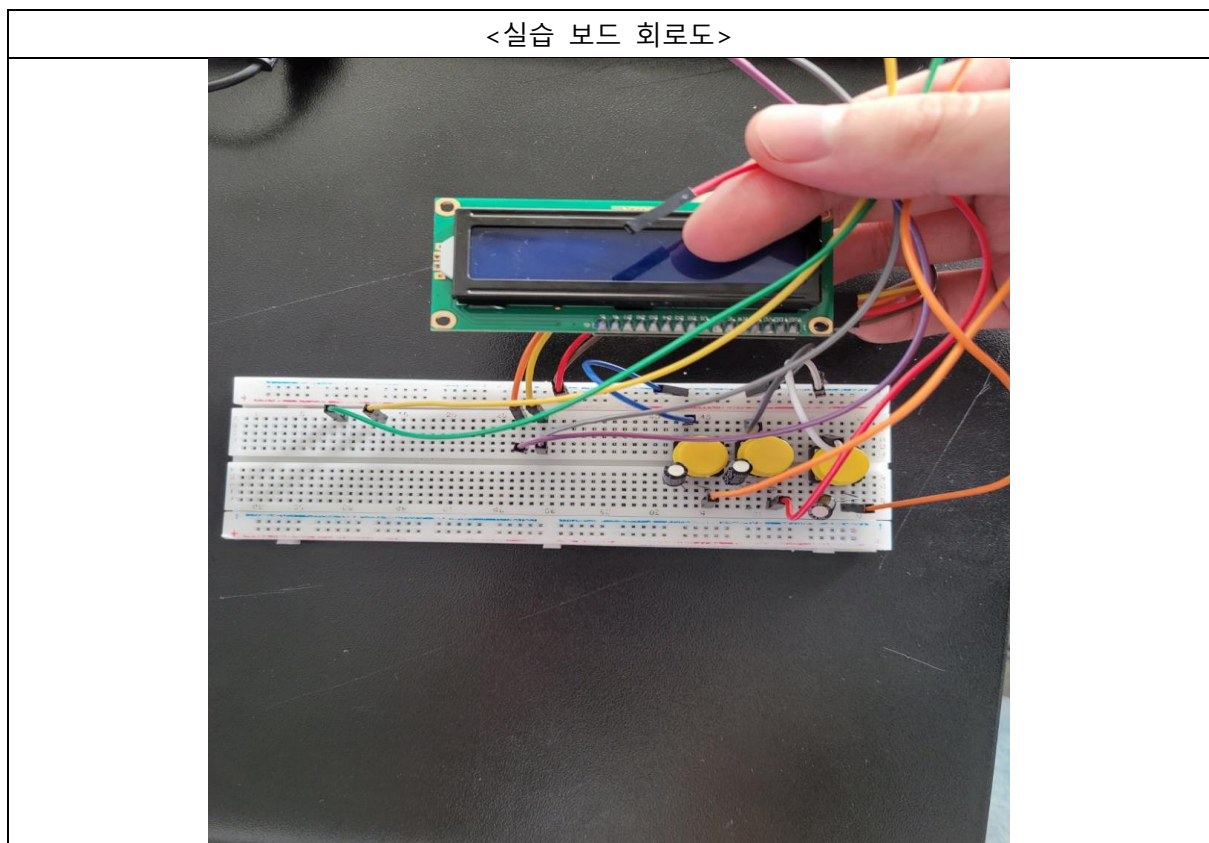
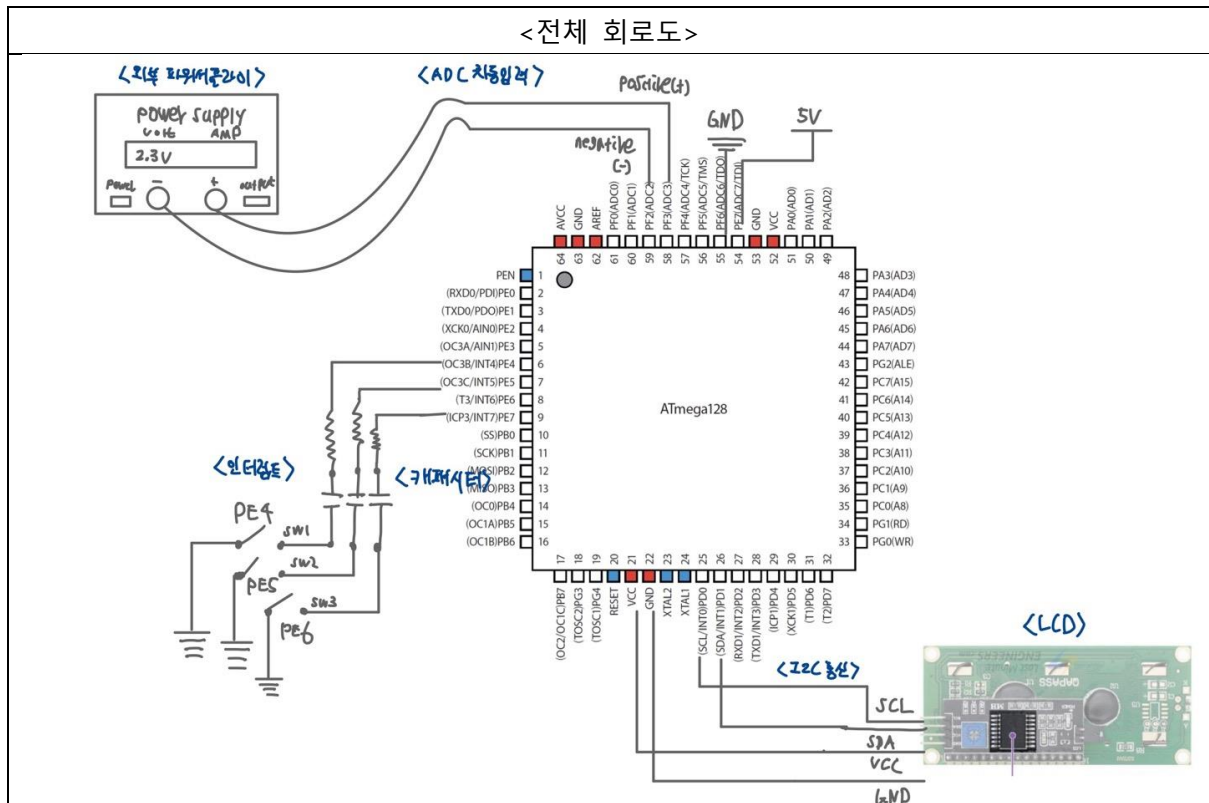
- Clear Display : 공백문자(코드 0x20)로 화면을 지우고 커서를 홈 위치(주소 0번)로 이동시킨다.
 - Return Home : 커서를 홈 위치로 이동시키고, 표시 영역이 이동된 경우 초기 위치로 이동시킨다. 화면에 출력된 내용(DDRAM의 값)은 변하지 않는다.
 - Entry Mode Set : 데이터 읽기 또는 쓰기 후 메모리의 증가 또는 감소 방향을 지정한다. DDRAM 에서 I/D =1 이면 커서를 오른쪽으로, I/D = 0 이면 왼쪽으로 옮긴다. S=1이면 I/D값에 따라 디스플레이를 왼쪽 또는 오른쪽으로 옮기며 , 이때 커서는 고정된 위치에 나타난다.
 - Display on/off control : 디스플레이(D), 커서(C), 커서 깜빡임(B)의 ON/OFF를 설정한다.
 - Cursor or Display Shift : 화면에 출력된 내용의 변경 없이 커서와 화면을 이동시킨다.
 - Function Set : 데이터 비트 크기(DL =1 이면 8비트, DL = 0 이면 4비트),
 - 디스플레이 행 수(N), 폰트 크기(F)를 설정한다.
 - Set CGRAM Address : 주소 카운터에 CGRAM 주소를 설정한다.
 - Set DDRAM Address : 주소 카운터에 DDRAM 주소를 설정한다.
- ➔ I2C 통신을 통해 한 번에 1 byte를 보낼 수 있으며, 각 bit는 서로 다른 역할의 의미를 가진다. I2C 통신에서는 장치의 주소가 필요하며, 실습 보드에서는 0x27로 설정되어 있다. Character LCD 조작에 필요한 모든 데이터는 8bit 단위이므로, 8bit 데이터를 상위, 하위 4bit 씩 잘라서 두 번에 나누어 전송한다. LCD에 전달하는 데이터는 설정 데이터와 문자 데이터로 나뉜다. RS=1 인 데이터를 Character LCD에 전달하면, 전달 받은 데이터를 ASCII 문자로 인식하고, 이를 현재 Cursor 위치에 저장한다.

2. 전체 회로도

[모드별 회로 구성]



[전체 회로 구성]



3. 각 모듈 별 초기화와 사용법에 대한

코드 및 이에 대한 설명

- 사용한 함수 선언 초기화

```
void TWI_init(void);
void TWI_OneByteWrite(uint8_t SLA_address, uint8_t byte_data);
void LCD_write_4bit(uint8_t data, uint8_t rs_bit); //4bit data 전달 함수
void LCD_write_8bit(uint8_t data, uint8_t rs_bit);
void LCD_write_data(char data);
void LCD_write_command(uint8_t command);
void LCD_clear_display();
void LCD_return_home();
void LCD_entry_mode_set(uint8_t ID, uint8_t S);
void LCD_display_onoff_control(uint8_t D, uint8_t C, uint8_t B);
void LCD_cursor_or_display_shift(uint8_t SC, uint8_t RL);
void LCD_function_set(uint8_t DL, uint8_t N, uint8_t F);
void LCD_goto_XY(uint8_t row, uint8_t col);
void LCD_init();
void LCD_write_string(char* string);
```

- 인터럽트 변수 선언 초기화

```
volatile uint8_t tmr1_cmpa_count = 0; // ISR에서 값을 변경하는 변수 tmr1_cmpa_count 선언
volatile uint8_t tmr1_one_second_flag = 0;
volatile uint16_t adc_value = 0;
volatile char qwer[32];
volatile int qwer_next = 0;
volatile int qwer1 = 0;
volatile int SW = 0;
volatile int SW3 = 0;
volatile int UART_SW = 0;
volatile int ms = 0;
volatile int int6_SW = 0;
```

- 메인문 초기화 함수

```
TWI_init(); // I2C 초기화
LCD_init(); // LCD 초기화
```

```
DDRE |= ~0x70; // PE4, 5, 6 입력
PORTE |= 0x70; // 내부 풀업 저항 활성화
EIMSK |= (1<< INT6)|(1<< INT5)|(1<< INT4); // 외부 인터럽트 핀 지정
EICRB |= (1<< ISC61)|(1<< ISC51)|(1<< ISC41); // Falling edgy
sei(); // 전역 인터럽트 활성화
```


▪ DDRE 레지스터를 설정을 통해 외부 인터럽트를 INT4, INT5, IN6을 사용하기 위해 포트 PE4, PE5, PE6을 설정하였다. PORTE 레지스터를 이용해 내부 풀업 저항을 활성화 하였고 EIMSK 레지스터를 통해 INT6, IN5, INT4핀에 입력되는 인터럽트를 활성화 되도록 설정하였다.

- 모드1 초기화 함수

```
tmr1_one_second_flag = 0; // 타이머 플래그 초기화
UCSR1A = 0x00; //현재 상태를 모니터링 하는 용도
UCSR1B |= (1<<RXCIE1) | (1<<RXEN1) | (1<<TXEN1); // UART의 기능을 설정
UCSR1C = 0x06; //UART의 기능을 설정
OCR1A = 4000; // OCR값 4000
    - 분주비를 8로 설정하고 OCR1A가 5번걸리게 되면 0.01초가 흐른다.
TCCR1B = 0x00; // 일시중지
분주비가 0일때는 타이머가 돌아가지 않는다.

TIMSK |= (1 << OCIE1A); // 1번 타이머/카운터 Compare match interrupt 설정

char s[17]; // Char 형 배열 길이 설정
```

▪ tmr1_one_second_flag 변수를 이용해서 타이머 플래그를 초기화 하고 UCSR1B |= (1<<RXCIE1) | (1<<RXEN1) | (1<<TXEN1) 코드는 RXCIE1 비트는 RX Complete interrupt Enable로 수신완료 인터럽트 발생을 허용, RXEN1은 RX Enable로 UART 수신기의 수신 기능을 활성화, TXEN1은 TX Enable로 송신기의 송신 기능을 활성화 한다. UCSR1C = 0x06로 설정을 해 Character Size의 크기를 정하였다. OCR1A값이 4000이고 compare match가 5번걸릴때 0.01초가 나오게 설정하였다. TCCR1B를 0으로 설정한 이유는 분주비를 0으로 설정을 해서 타이머가 돌아가지 않게끔 설정해 받아온 문자가 마지막 문자임을 알린다. TIMSK 레지스터를 이용해 1번 타이머/카운터 Compare match interrupt 설정하였다.

- 모드2 초기화 함수

```
uint16_t seconds = 0; // seconds 변수
TIMSK |= (1 << OCIE1A); // 1번 타이머/카운터 Compare match A Interrupt 설정
TCCR1B = 0x0A; // 1번 타이머/카운터 분주비 8
OCR1A = 0x0FA0; //TOP값

sei(); //전역변수 활성화
```

`TCCR1B = 0x0A` //분주비 8로 설정

- `uint16_t seconds = 0` seconds 변수를 초기화 해준다. `TIMSK |= (1 << OCIE1A)` 를 통해 1번 타이머/카운터 compare match A interrupt가 활성화 되도록 설정해준다. `TCCR1B = 0x0A`는 분주비를 8로 설정하였을 때와 타이머/카운터 모드의 동작을 CTC로 설정하였을 때 레지스터 설정 결과이다. `OCR1A = 0x0FA0`는 Top값이 OCRnA값이기 때문이다.

- 모드3 초기화 함수

```
ADMUX |= 0x5B;          // 차등입력 ADC3->양, ADC2->음
ADCSRA |= (1<<ADEN);    // ADC 활성화
ADCSRA |= (1<<ADFR);    // 프리러닝모드
ADCSRA |= (1<<ADSC);    // AD변환시작
ADCSRA |= 0x07;         // 분주비 128
```

`char asd[15];` // asd 배열 만들

- ADMUX 레지스터 값을 0x5B로 설정한 이유는 차등입력을 주기 위해서 MUXn레지스터 값을 11011로 설정하였고 REFS0와 REFS1을 각각 1 0으로 설정하였는데 이는 외부 AVCC를 사용하기 위해서이다.

- 모드3 초기화 함수

```
float V1; // V1
float V2; //V2
float adc1 = 51; //adc1
float adc2 = 87; //adc2
sei(); //전역변수 활성화
```

- 파워서플라이의 출력 전압을 각각 1V, 2V로 설정하였을 때, 이 때의 ADC 값을 변수 선언 초기화

4. 각 동작 모드 별 전체 코드 및 이에 대한 설명

[모드 1] 광고문 출력 코드

```
while(1){  
    if(SW==0){ // 광고문 출력 모드  
        tmr1_one_second_flag = 0; // 타이머 플레그 초기화  
        UCSR1A = 0x00; //현재 상태를 모니터링 하는 용도  
        UCSR1B |= (1<<RXIE1) | (1<<RXEN1) | (1<<TXEN1); // UART 기능  
        설정  
  
        UCSR1C = 0x06; // UART의 기능을 설정  
        OCR1A = 4000; // OCR값 4000  
        TCCR1B = 0x00; // 중지  
        TIMSK |= (1 << OCIE1A); // 1번 타이머/카운터 C match interrupt  
        설정  
  
        char s[17]; // Char 형 배열 길이 설정  
  
        LCD_clear_display(); // 화면을 지우고 커서를 home으로 이동  
        TWI_init();// I2C 초기화  
        LCD_init();// LCD 초기화  
  
        if (UART_SW == 0){ // 통신을 하지 않았을 때  
            LCD_goto_XY(0,0); // LCD 0,0 위치에  
            LCD_write_string("Embedded coding"); // Embedded coding  
            출력  
  
            LCD_goto_XY(1,0); // LCD 1,0 위치에  
            LCD_write_string("Team #5"); // Team #5 출력  
        }  
        else{ // 모드 변경 후 다시 1로 왔을 때 문자 유지  
            LCD_clear_display();// 화면을 지우고 커서를 home으로 이동  
            LCD_goto_XY(0,0); // LCD 0,0 위치에  
            strncpy(s,qwer,(qwer1<17) ? qwer1 : 16); //문자열 받음  
            s[qwer1]=NULL; // 문자열의 마지막은 NULL  
            LCD_write_string(s); // 받은 문자열 출력  
  
            if (qwer1 > 16) // 길이가 16보다 클 때  
            {  
                LCD_goto_XY(1,0); // LCD 1,0 자리에  
                strncpy(s,qwer+16,qwer1-16); //문자열 받음  
                s[qwer1]=NULL; // 문자열의 마지막은 NULL  
                LCD_write_string(s); // 받은 문자열 출력  
            }  
        }  
    }  
}
```

- ➔ 처음 컴퓨터에 atmega128을 연결 하고 시리얼 통신이 연결 되어 있지 않으면 LCD 화면 (0, 0) 자리에는 Embedded coding, (1,0) 자리에는 Team #5가 출력된다. 시리얼 통신을 통하여 문자를 입력하였을 때 문자열을 너무 빠르게 받아 못 받는 문자가 생기는 것을 예방하기 위하여 if (tmr1_one_second_flag - ms > 30)를 이용한다. 또한 UART 통신을 통해 광고문을 입력하면 해당 광고문의 길이에 따라 줄을 맞추기 위해 광고문의 길이가 16보다 작을 경우와 클 경우를 나누어서 생각하고 만약 광고문의 길이가 16보다 작다면 LCD의 (0,0) 자리에 출력하고 16보다 클 때에는 다음 줄로 넘겨서 작성하도록 한다. 또한 광고문 출력 모드에서 1번 스위치를 누르게 되면 디지털 전압계 모드로 이동하고 2번 스위치를 누르면 스탑워치 모드로 이동한다. 또한 if문을 사용하여 3번 스위치를 눌렀을 때 화면을 키고 끄도록 제어한다.

[모드 2] 스탑워치 모드

```

if (SW==1){ // 스톱워치 모드
    TWI_init();
    LCD_init();
    uint16_t seconds = 0; // seconds 변수
    TIMSK |= (1 << OCIE1A); // 1번 타이머/카운터 Compare match A

Interrupt 설정

    TCCR1B = 0x0A; // 1번 타이머/카운터 분주비 8
    OCR1A = 0x0FA0;
    sei();

    LCD_goto_XY(0,0); //LCD화면
(0,0)으로 커서 이동

    LCD_write_string("00:00:00.00"); //초기값 출력

    int h = 0; // 시
    int m = 0; // 분
    int s = 0; // 초

    while(SW == 1){ // 스톱워치 모드를 실행할 동안
        if(SW3 % 3 == 1){ // 스위치3을 누른 횟수 / 3의 나머지가
1일 때

            if(tmr1_one_second_flag){ // 0.01초 흐름
                tmr1_one_second_flag = 0; // 초기화
                if(seconds < 10)LCD_goto_XY(0,10);
//seconds가 10보다 작으면 0,10 자리에

                else LCD_goto_XY(0,9); // 아니라면 0,9
자리에

                sprintf(asd,"%d", seconds); // 문자열

```

받은

출력

s가 10보다 작으면 0,7 자리에

아니라면 0,6 자리에

받은

자리에

10보다 작으면 0,4 자리에

아니라면 0,3 자리에

출력

자리에

10보다 작으면 0,1자리에

아니라면 0,0 자리에

받은

출력

```
LCD_write_string(asd); // 받은 문자열
```

```
if(seconds >= 98){ // 1초 흐름  
    s++;  
    seconds=0; // 초기화  
    if(s<10)LCD_goto_XY(0, 7); //  
  
    else LCD_goto_XY(0,6); //
```

```
sprintf(asd,"%d", s); // 초
```

```
LCD_write_string(asd); // 출력
```

```
}  
if (s == 60) { // 1분 흐름  
    LCD_goto_XY(0,6); // LCD 0,6
```

```
LCD_write_string("0"); // 0 출력  
m++;  
s=0; // 초기화  
if(m<10)LCD_goto_XY(0,4); // m이
```

```
else LCD_goto_XY(0,3); //
```

```
sprintf(asd,"%d", m); // 분 받은  
LCD_write_string(asd); // 문자열
```

```
}
```

```
if (m == 60){ // 1시간 흐름  
    LCD_goto_XY(0,3); // LCD 0,3
```

```
LCD_write_string("0"); // 0 출력  
h++;  
m=0; // 초기화  
if(h<10)LCD_goto_XY(0,1); // h가
```

```
else LCD_goto_XY(0,0); //
```

```
sprintf(asd,"%d", h); // 시간
```

```
LCD_write_string(asd); // 문자열
```

```
}
```

```

        printf("%d:%d:%d:%d\r\n", h, m, s,
++seconds); // 시리얼 통신으로 확인

        sprintf(asd,"%d:%d:%d:%d",h,m,s,++seconds); // 시간, 분, 초 받음
        LCD_write_string(asd);
    }

}

else if(SW3 % 3 == 2){ // SW3 2번 클릭, 스톱워치가 멈춘다
    printf("%d:%d:%d:%d\r\n", h, m, s, seconds);
}

else if(SW3 == 3){ // SW3 3번 클릭 // 초기화 후 다시
    스톱워치 실행

    h = 0; // 시
    m = 0; // 분
    s = 0; // 초
    seconds = 0; // seconds 초기화
    SW3 = 1;
    LCD_goto_XY(0,0); // LCD 광고문 출력 모드
    LCD_write_string("00:00:00:00");
}
}
}

```

- ➔ 타이머 인터럽트를 사용하여 스톱워치 기능을 만드는데 처음 스톱워치 모드로 들어갔을 때에는 LCD 화면에 초기값 00:00:00:00을 출력한다. 스톱워치 모드가 실행 되고 있을 때에는 if문의 조건 문을 3번 스위치를 누른 횟수를 이용하여 만들고 1번 눌렀을 경우에 타이머 인터럽트를 이용하여 0.01초씩 증가하도록 만든다. 또한 10의 단위로 올라갔을 경우를 대비하여 if문을 하나 더 만들어 자릿수를 정리한다. 0.01초의 단위와 초, 분, 시간 단위를 따로 설정하여 각 자리에 맞는 시간과 분, 초가 나오도록 제어한다. else if(SW3 % 3 == 2)를 이용하여 스톱워치를 멈추는 기능을 만들고 else if(SW3 == 3)을 이용하여 멈춘 상태에서 다시 한 번 눌렀을 때 초기화하여 다시 스톱워치가 실행 되도록 한다. 이때 1번 스위치를 누른다면 광고문 출력 모드로 2번 스위치를 누른다면 디지털 전압계 모드로 이동한다.

[모드 3] 디지털 전압계 모드

```

if(SW==2){ // 디지털 전압계 모드
    TWI_init(); // I2C 초기화
    LCD_init(); // LCD 초기화
}

```

```

ADMUX |= 0x5B;           // 차등입력 AD0->양, AD1->음
ADCSRA |= (1<<ADEN); // ADC 활성화
ADCSRA |= (1<<ADFR); // 프리러닝모드
ADCSRA |= (1<<ADSC); // AD변환시작
ADCSRA |= 0x07;         // 분주비 128

LCD_goto_XY(0, 0);      // LCD화면 (0,0)으로 커서 이동
LCD_write_string("Voltage: "); // Voltage: 문자열 출력

float V1; // V1
float V2; // V2
float adc1 = 51; // adc1 정의
float adc2 = 87; // adc2 정의
sei(); // 전역 인터럽트
while (SW == 2) // 디지털 전압계 모드
{
    adc_value = ADC; // adc_value 정의

    //V1 = (adc_value*(5.0/512.0)); // 차등입력 값을 구하는
    식
    V2 = (1/(adc2-adc1)*adc_value)+ ((adc2 - 2*adc1)/(adc2
- adc1)); // V2 식
    printf("ADC : %d, voltage : %.2fV\r\n", adc_value, V2);
    // 시리얼 통신으로 확인을 위한 값 출력
    _delay_ms(100); // 딜레이 0.1초

    LCD_goto_XY(0,0); // LCD 0,0 자리에
    sprintf(asd,"Voltage:%.2fV", V2); // V2값 받음
    LCD_write_string(asd); // 문자열 출력

    if(int6_SW == 1){ // 스위치 3이 작동하면
        int6_SW=0; // 초기화
        LCD_goto_XY(1,0); // LCD 1,0 자리에
        sprintf(asd,"Voltage:%.2fV", V2); // V2값 받음
        LCD_write_string(asd); // 문자열 출력
    }
    _delay_ms(100); //0.1초 마다 출력
}
}
}
}

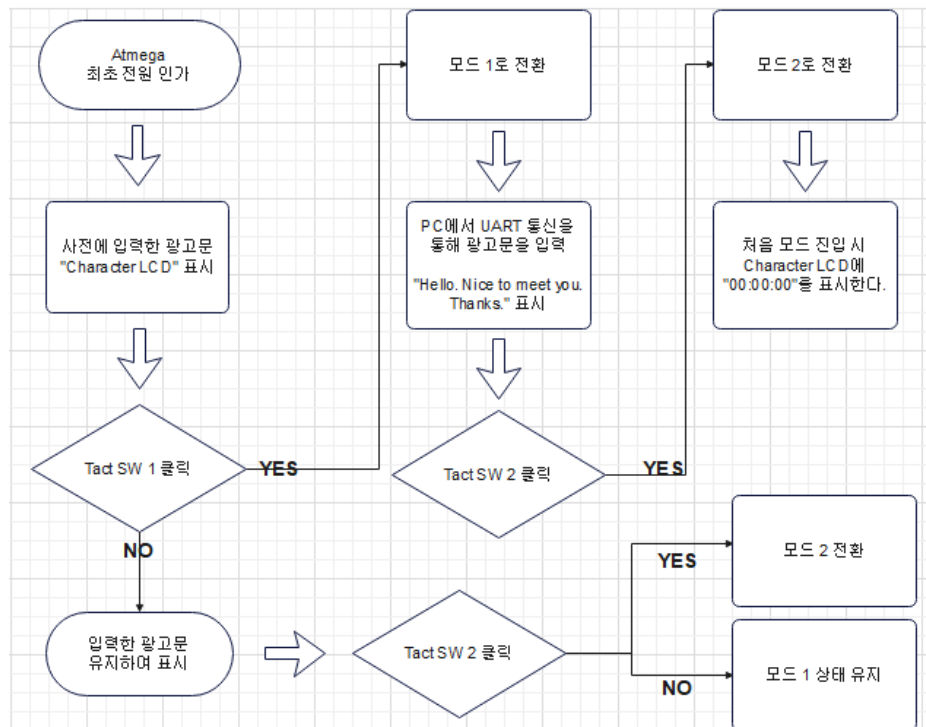
```

- ➔ ADC0 핀과 그라운드의 전위차를 ADC를 통하여 측정하고 측정된 값을 LCD에 표시한다.
두 개의 ADC값을 구하기 위해서 $V1 = (adc_value * (5.0/512.0))$ 의 식을 이용하고 전위차를

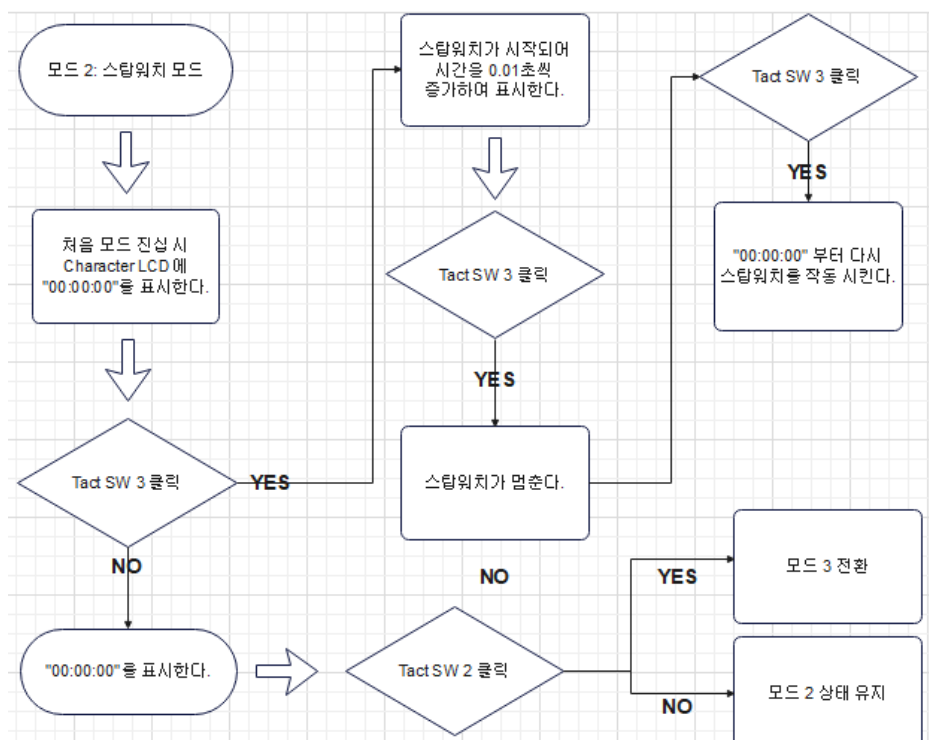
구하기 위해서 $V2 = (1/(adc2-adc1)*adc_value) + ((adc2 - 2*adc1)/(adc2 - adc1));$ 식을 사용한다. 0.01V 단위로 측정해야 하기 때문에 %.2f를 사용한다. if(int6_SW == 1)를 이용하여 3번 스위치를 눌렀을 때 LCD의 두 번째 줄에 스위치를 누를 당시의 전압을 표시한다. 이때 1번 스위치를 누르게 된다면 스탑워치 모드로 2번 스위치를 누른다면 광고문 출력 모드로 이동한다.

5. 전체 코드의 순서도

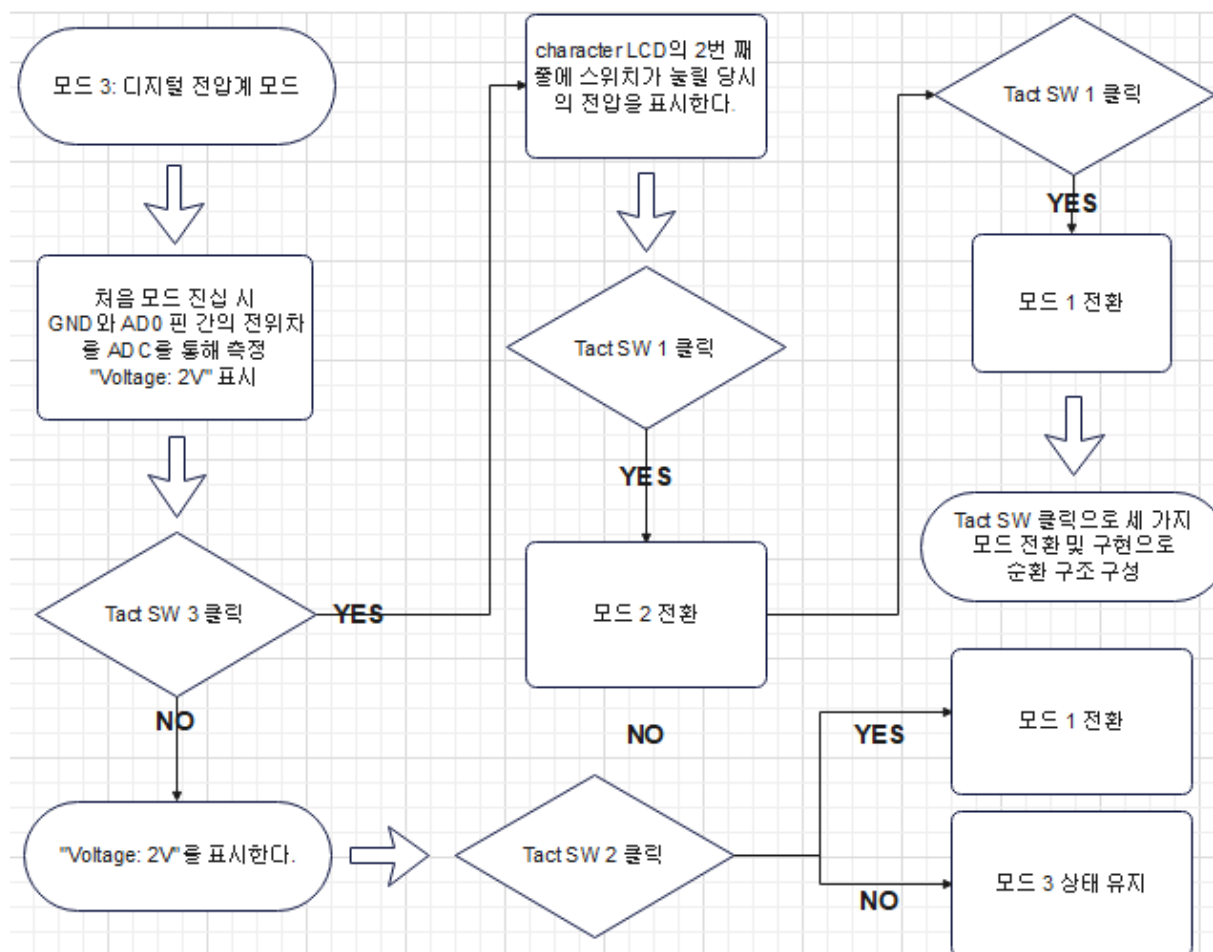
[모드 1] 광고문 출력 모드 순서도



[모드 2] 스탑워치 모드 순서도



[모드 3] 디지털 전압계 모드 순서도



6. 전체 코드 (+주석)

```
#define F_CPU 16000000L
#include <avr/io.h>
#include <stdio.h>
#include <string.h>
#include <util/delay.h>
#include <util/twi.h>
#include <avr/interrupt.h>
#define LCD_ADDR 0x27
#define RS_COMMAND 0
#define RS_DATA 1
#define BACKLIGHT (1 << 3)
#define PIN_EN (1 << 2)
#define SCL_FREQ 100 // kHz
#define CPU_CLOCK 16000 // kHz
// #define START_TRANSMITTED 0x08
// #define MT_SLA_ACK 0x18
// #define MT_DATA_ACK 0x28

// 사용한 함수 선언
void TWI_init(void);
void TWI_OneByteWrite(uint8_t SLA_address, uint8_t byte_data);
void LCD_write_4bit(uint8_t data, uint8_t rs_bit); // 4bit data 전달 함수
void LCD_write_8bit(uint8_t data, uint8_t rs_bit);
void LCD_write_data(char data);
void LCD_write_command(uint8_t command);
void LCD_clear_display();
void LCD_return_home();
void LCD_entry_mode_set(uint8_t ID, uint8_t S);
void LCD_display_onoff_control(uint8_t D, uint8_t C, uint8_t B);
void LCD_cursor_or_display_shift(uint8_t SC, uint8_t RL);
void LCD_function_set(uint8_t DL, uint8_t N, uint8_t F);
void LCD_goto_XY(uint8_t row, uint8_t col);
void LCD_init();
void LCD_write_string(char* string);

// 4비트 데이터 출력 함수
void LCD_write_4bit(uint8_t data, uint8_t rs_bit) {
    uint8_t lcd_data, msd_data;
    msd_data = data & 0xF0;

    lcd_data = msd_data | BACKLIGHT | PIN_EN | rs_bit; // D7-4 BL EN RW RS

    TWI_OneByteWrite(LCD_ADDR, lcd_data);
    lcd_data &= (~PIN_EN);
    TWI_OneByteWrite(LCD_ADDR, lcd_data);

    _delay_us(40);
}

// 8비트 데이터 출력 함수
void LCD_write_8bit(uint8_t data, uint8_t rs_bit){
    uint8_t lcd_data, msd_data;

    msd_data = data & 0xF0;
    lcd_data = (data << 4) & 0xF0;

    LCD_write_4bit(msd_data, rs_bit);
```

```

        LCD_write_4bit(lcd_data, rs_bit);
    }
    // LCD 문자 데이터 출력 함수
    void LCD_write_data(char data){
        LCD_write_8bit(data, RS_DATA);
    }
    // LCD 레지스터 설정 함수
    void LCD_write_command(uint8_t command){
        LCD_write_8bit(command, RS_COMMAND);
    }
    // LCD 화면 지우고 커서 초기화 함수
    void LCD_clear_display(){
        LCD_write_command(0x01);
        _delay_ms(2);
    }
    // LCD 커서 초기화 함수
    void LCD_return_home(){
        LCD_write_command(0x02);
        _delay_ms(2);
    }
    // LCD 화면 이동 및 커서위치 증감 설정 함수
    void LCD_entry_mode_set(uint8_t ID, uint8_t S){
        LCD_write_command(0x04 | (ID<<1) | (S<<0));
        _delay_ms(40);
    }
    // LCD 화면, 커서 설정 함수
    void LCD_display_onoff_control(uint8_t D, uint8_t C, uint8_t B){
        LCD_write_command(0x08 | (D<<2) | (C<<1) | (B<<0));
        _delay_ms(40);
    }
    // LCD 화면이나 커서 이동 방향 설정 함수
    void LCD_cursor_or_display_shift(uint8_t SC, uint8_t RL){
        LCD_write_command(0x10 | (SC<<3) | (RL<<2));
        _delay_ms(40);
    }
    // LCD 인터페이스 길이, 화면 표시 행수, 폰트 크기 설정 함수
    void LCD_function_set(uint8_t DL, uint8_t N, uint8_t F){
        LCD_write_command(0x20 | (DL<<4) | (N<<3) | (F<<2));
        _delay_ms(40);
    }
    // LCD 커서 이동 함수
    void LCD_goto_XY(uint8_t row, uint8_t col){
        col %= 16;
        row %= 2;

        uint8_t address = (0x40 * row) + col;
        uint8_t command = 0x80 + address;
        LCD_write_command(command);
    }
    // LCD 초기화 함수
    void LCD_init(){
        _delay_ms(25);

        LCD_write_4bit(0x30, RS_COMMAND);
        _delay_ms(2);
        LCD_write_4bit(0x30, RS_COMMAND);
    }

```

```

        _delay_ms(2);
        LCD_write_4bit(0x30, RS_COMMAND);
        _delay_ms(2);
        LCD_write_4bit(0x20, RS_COMMAND);
        _delay_ms(2);

        LCD_function_set(0, 1, 0);
        LCD_display_onoff_control(1, 0, 0);
        LCD_entry_mode_set(1, 0);
        _delay_ms(2);
        LCD_clear_display();
    }
    // LCD 문자열 출력 함수
    void LCD_write_string(char* string){
        uint8_t i;
        for (i=0; string[i]; i++){
            LCD_write_data(string[i]);
        }
    }
    // I2C 초기화 함수
    void TWI_init(void) {
        TWSR &= ~((1 << TWPS1) | (1 << TWPS0));
        TWBR = ((CPU_CLOCK /SCL_FREQ) - 16) / 2;
    }
    // I2C 함수
    void TWI_OneByteWrite(uint8_t SLA_address, uint8_t byte_data) {
        uint8_t SLA_W;
        SLA_W = (SLA_address << 1);

        TWCR = (1<<TWINT)|(1<<TWSTA)|(1<<TWEN);
        while (!(TWCR & (1<<TWINT)));
        TWDR = SLA_W;
        TWCR = (1<<TWINT) | (1<<TWEN);
        while (!(TWCR & (1<<TWINT)));
        TWDR = byte_data;
        TWCR = (1<<TWINT) | (1<<TWEN);
        while (!(TWCR & (1<<TWINT)));
        TWCR = (1<<TWINT)|(1<<TWEN)| (1<<TWSTO);
        _delay_us(10);
    }

    // 초기화, 송신, 수신 기능을 함수로 작성
    void UART1_init(uint16_t baud){
        UCSR1B |= (1<<RXEN1) | (1<<TXEN1); // RXEN1과 TXEN1을 1로 초기화
        UBRR1H = baud >> 8;
        UBRR1L = baud & 0x00FF;
    }
    void UART1_transmit(char data){ // 송신 함수
        while(!(UCSR1A & (1<<UDRE1))); // 송신 버퍼가 비어 있을 때까지 무한 반복
        UDR1=data; // data값을 UDR1에 저장
    }
    uint8_t UART1_receive(void){ // 수신 함수
        while (!(UCSR1A & (1<<RXC1))); // 수신 버퍼에 읽을 문자가 있을 때까지 무한 반복
        return UDR1; // UDR1을 반환
    }

```

```

// printf와 scanf 구현
FILE OUTPUT = FDEV_SETUP_STREAM(UART1_transmit, NULL, _FDEV_SETUP_WRITE);
FILE INPUT = FDEV_SETUP_STREAM(NULL, UART1_receive, _FDEV_SETUP_READ);

// 인터럽트 변수 선언
volatile uint8_t tmr1_cmpa_count = 0; // ISR에서 값을 변경하는 변수 tmr1_cmpa_count 선언
volatile uint8_t tmr1_one_second_flag = 0;
volatile uint16_t adc_value = 0;
volatile char qwer[32];
volatile int qwer_next = 0;
volatile int qwer1 = 0;
volatile int SW = 0;
volatile int SW3 = 0;
volatile int UART_SW = 0;
volatile int ms = 0;
volatile int int6_SW = 0;

// 반환값이 없고 매개변수의 데이터 형이 없는 인터럽트 함수 선언
ISR(INT4_vect){ // PE4 스위치 이전 모드로 가도록 동작
    SW = (SW+2) % 3;
}

ISR(INT5_vect){ // PE5 스위치 다음 모드로 가도록 동작
    SW = (SW+1) % 3;
}

ISR(INT6_vect){ // PE6 스위치 모드 별 동작 실행
    SW3++;
    int6_SW=1;
}

ISR(TIMER1_COMPA_vect) { // 1번 타이머/카운터의 compare match 인터럽트
    tmr1_cmpa_count++;
    if (tmr1_cmpa_count == 5) {
        tmr1_cmpa_count = 0;
        tmr1_one_second_flag++;
    }
}

ISR(USART1_RX_vect){ // UART 수신 인터럽트
    TCCR1B = 0x0A; // CTC모드
    ms = tmr1_one_second_flag;

    UART_SW = 1;
    qwer[qwer_next]=UDR1;
    qwer_next++;
    qwer1 = qwer_next;
}

int main(void)
{
    stdout = &OUTPUT;
    stdin = &INPUT;
    UART1_init(103); // 9600
    TWI_init(); // I2C 초기화
}

```

```

LCD_init(); // LCD 초기화
char asd[32]; // 문자열 생성
DDRE |= ~0x70; // PE4, 5, 6 입력
PORTE |= 0x70; // 내부 풀업 저항 활성화
EIMSK |= (1<< INT6)|(1<< INT5)|(1<< INT4); // 외부 인터럽트 핀 지정
EICRB |= (1<< ISC61)|(1<< ISC51)|(1<< ISC41); // Falling edgy

sei(); // 인터럽트 활성화

while(1){
    if(SW==0){ // 광고문 출력 모드
        tmr1_one_second_flag = 0; // 타이머 플레그 초기화
        UCSR1A = 0x00; //현재 상태를 모니터링 하는 용도
        UCSR1B |= (1<<RXIE1) | (1<<RXEN1) | (1<<TXEN1); // UART 기능

        UCSR1C = 0x06; // UART의 기능을 설정
        OCR1A = 4000; // OCR값 4000
        TCCR1B = 0x00; // 중지
        TIMSK |= (1 << OCIE1A); // 1번 타이머/카운터 C match interrupt

        char s[17]; // Char 형 배열 길이 설정

        LCD_clear_display(); // 화면을 지우고 커서를 home으로 이동
        TWI_init();// I2C 초기화
        LCD_init();// LCD 초기화

        if (UART_SW == 0){ // 통신을 하지 않았을 때
            LCD_goto_XY(0,0); // LCD 0,0 위치에
            LCD_write_string("Embedded coding"); // Embedded coding

            LCD_goto_XY(1,0); // LCD 1,0 위치에
            LCD_write_string("Team #5"); // Team #5 출력
        }
        else{ // 모드 변경 후 다시 1로 왔을 때 문자 유지
            LCD_clear_display();// 화면을 지우고 커서를 home으로 이동
            LCD_goto_XY(0,0); // LCD 0,0 위치에
            strncpy(s,qwer,(qwer1<17) ? qwer1 : 16); //문자열 받음
            s[qwer1]=NULL; // 문자열의 마지막은 NULL
            LCD_write_string(s); // 받은 문자열 출력

            if (qwer1 > 16) // 길이가 16보다 클 때
            {
                LCD_goto_XY(1,0); // LCD 1,0 자리에
                strncpy(s,qwer+16,qwer1-16); //문자열 받음
                s[qwer1]=NULL; // 문자열의 마지막은 NULL
            }
        }
    }
}

```

설정

설정

출력

```

        LCD_write_string(s); // 받은 문자열 출력
    }
}

while (SW == 0) // 광고문 출력 모드가 실행되는 동안
{
    if (tmr1_one_second_flag - ms > 30){ // 0.30초 뒤에
        tmr1_one_second_flag=0; // 타이머 flag초기화
        TCCR1B = 0x00; // 중지
        LCD_clear_display();// 화면을 지우고 커서를

home으로 이동

        LCD_goto_XY(0,0); // LCD 0, 0 자리에
        strncpy(s,qwer,(qwer_next<17) ? qwer_next : 16);

// 문자열 받음

        s[qwer_next]=NULL; // 문자열의

마지막은 NULL

        LCD_write_string(s); // 받은 문자열 출력

        if (qwer_next > 16) // 길이가 16보다 클 때
        {
            LCD_goto_XY(1,0); //

LCD 1,0 자리에

            strncpy(s,qwer+16,qwer_next-16); //

문자열 복사

            s[qwer_next-16]=NULL; //

문자열의 마지막은 NULL

            LCD_write_string(s); // 받은 문자열

출력

        }
        qwer_next = 0;

// 0으로 초기화

    }
    if(SW3 == 1){ // LCD 화면 끄고
        LCD_display_onoff_control(0,0,0);
    }
    if(SW3 >= 2){ // LCD 화면 키고
        LCD_display_onoff_control(1,0,0);
        SW3=0;
    }
}

}
if (SW==1){ // 스톱워치 모드
    TWI_init();
    LCD_init();
    uint16_t seconds = 0; // seconds 변수
    TIMSK |= (1 << OCIE1A); // 1번 타이머/카운터 Compare match A

Interrupt 설정

```

```

TCCR1B = 0x0A;           // 1번 타이머/카운터 분주비 8
OCR1A = 0x0FA0;
sei();

LCD_goto_XY(0,0);           //LCD화면
(0,0)으로 커서 이동

LCD_write_string("00:00:00.00"); //초기값 출력

int h = 0; // 시
int m = 0; // 분
int s = 0; // 초

while(SW == 1){ // 스톱위치 모드를 실행할 동안
    if(SW3 % 3 == 1){ // 스위치3을 누른 횟수 / 3의 나머지가
1일 때
        if(tmr1_one_second_flag){ // 0.01초 흐름
            tmr1_one_second_flag = 0; // 초기화
            if(seconds < 10)LCD_goto_XY(0,10);
//seconds가 10보다 작으면 0,10 자리에
            else LCD_goto_XY(0,9); // 아니라면 0,9
자리에
            sprintf(asd,"%d", seconds); // 문자열
받은
            LCD_write_string(asd); // 받은 문자열
출력

            if(seconds >= 98){ // 1초 흐름
                s++;
                seconds=0; // 초기화
                if(s<10)LCD_goto_XY(0, 7); //
s가 10보다 작으면 0,7 자리에
                else LCD_goto_XY(0,6); //
아니라면 0,6 자리에
                sprintf(asd,"%d", s); // 초
받은
                LCD_write_string(asd); // 출력
            }
            if (s == 60) { // 1분 흐름
                LCD_goto_XY(0,6); // LCD 0,6
자리에
                LCD_write_string("0"); // 0 출력
                m++;
                s=0; // 초기화
                if(m<10)LCD_goto_XY(0,4); // m이
10보다 작으면 0,4 자리에

```

아니라면 0,3 자리에

출력

자리에

10보다 작으면 0,1자리에

아니라면 0,0 자리에

받음

출력

++seconds); // 시리얼 통신으로 확인

스탑워치 실행

```
else LCD_goto_XY(0,3); //

sprintf(asd,"%d", m); // 분 받음
LCD_write_string(asd); // 문자열

}

if (m == 60){ // 1시간 흐름
    LCD_goto_XY(0,3); // LCD 0,3

    LCD_write_string("0"); // 0 출력
    h++;
    m=0; // 초기화
    if(h<10)LCD_goto_XY(0,1); // h가

else LCD_goto_XY(0,0); //

    sprintf(asd,"%d", h); // 시간

    LCD_write_string(asd); // 문자열

}
printf("%d:%d:%d:%d\r\n", h, m, s,
++seconds); // 시리얼 통신으로 확인

    sprintf(asd,"%d:%d:%d:%d",h,m,s,++seconds); // 시간, 분, 초 받음
    LCD_write_string(asd);
}

}
else if(SW3 % 3 == 2){ // SW3 2번 클릭, 스탑워치가 멈춘다
    printf("%d:%d:%d:%d\r\n", h, m, s, seconds);
}

else if(SW3 == 3){ // SW3 3번 클릭 // 초기화 후 다시

    h = 0; // 시
    m = 0; // 분
    s = 0; // 초
    seconds = 0; // seconds 초기화
    SW3 = 1;
    LCD_goto_XY(0,0); // LCD 광고문 출력 모드
    LCD_write_string("00:00:00:00");
}

}

}

if(SW==2){ // 디지털 전압계 모드
```



```

TWI_init();           // I2C 초기화
LCD_init();           // LCD 초기화

ADMUX |= 0x5B;        // 차등입력 AD0->양, AD1->음
ADCSRA |= (1<<ADEN); // ADC 활성화
ADCSRA |= (1<<ADFR); // 프리러닝모드
ADCSRA |= (1<<ADSC); // AD변환시작
ADCSRA |= 0x07;       // 분주비 128

LCD_goto_XY(0, 0);    // LCD화면 (0,0)으로 커서 이동
LCD_write_string("Voltage: "); // Voltage: 문자열 출력

float V1; // V1
float V2; // V2
float adc1 = 51; // adc1 정의
float adc2 = 87; // adc2 정의
sei(); // 전역 인터럽트
while (SW == 2) // 디지털 전압계 모드
{
    adc_value = ADC; // adc_value 정의

    //V1 = (adc_value*(5.0/512.0)); // 차등입력 값을 구하는
    식

    V2 = (1/(adc2-adc1)*adc_value)+ ((adc2 - 2*adc1)/(adc2
- adc1)); // V2 식

    printf("ADC : %d, voltage : %.2fV\r\n", adc_value, V2);
// 시리얼 통신으로 확인을 위한 값 출력

    _delay_ms(100); // 딜레이 0.1초

    LCD_goto_XY(0,0); // LCD 0,0 자리에
    sprintf(asd,"Voltage:%.2fV", V2); // V2값 받음
    LCD_write_string(asd); // 문자열 출력

    if(int6_SW == 1){ // 스위치 3이 작동하면
        int6_SW=0; // 초기화
        LCD_goto_XY(1,0); // LCD 1,0 자리에
        sprintf(asd,"Voltage:%.2fV", V2); // V2값 받음
        LCD_write_string(asd); // 문자열 출력
    }
    _delay_ms(100); //0.1초 마다 출력
}
}
}


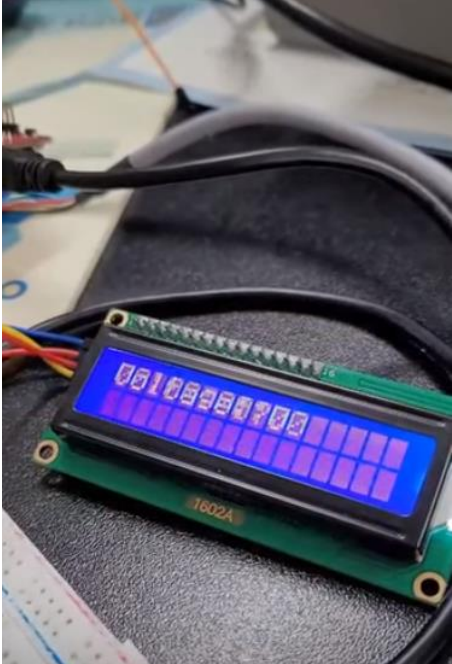
```

7. 동작 시연 영상 유튜브 링크 주소



[모드 1]

<p>사전에 입력한 광고문 모드 전환</p> <p>Tact SW 1, 2, 모드 전환 및 Tact SW 3 모드별 기능 구현</p>	<p>UART 통신을 통해 광고문 모드 전환</p> <p>Tact SW 1, 2, 모드 전환 및 Tact SW 3 모드별 기능 구현</p>
<p>Youtube link: https://www.youtube.com/shorts/l2BW_H47xqU</p>	<p>Youtbe link: https://www.youtube.com/shorts/iwet-cKbGPo</p>
	

[모드 2]

<p>Tact SW 3 모드별 기능</p>	<p>Tact SW 1, 2 클릭 모드 이동</p>
<p>Youtube link: https://www.youtube.com/shorts/iwet-cKbGPo</p>	<p>Youtube. link: https://www.youtube.com/shorts/d43mvMgFCoQ</p>
	

[모드 3]

Tact SW 3 모드별 기능	Tact SW 1, 2 클릭 모드 이동
Youtube link: https://www.youtube.com/watch?v=9tD5ZAuoXoo	Youtube link: https://www.youtube.com/shorts/LmFoRTxT1Dg
	

[전체]

전체 모드 동작 시연 영상
Youtube link: https://www.youtube.com/shorts/-6nKFW2DCrQ


8. 소감

20181304 김기태 : 한 학기 동안 배운 내용을 가지고 기말 프로젝트를 진행하였습니다. 모드 1,2,3 을 만들면서 우리가 배운 내용이 빠짐없이 들어가는 게 정말 신기했고 그 중에서도 스톱워치를 CLCD로 나타내어 시간이 흘러가는 걸 구현했을 때 뿌듯하였습니다. 만드는 게 굉장히 어려웠다. 교수님이 수업을 잘 따라간 사람이라면 하루면 된다고 했는데 나는 열심히 했을까 라는 생각이 또 듭니다. 가장 어려웠던 점은 모드1을 구현하는 것이었고, 두 번째는 각각의 모드들을 합쳤을 때 원하는 모드로 이동하지 않거나 화면이 꺼지는 등 현상이었습니다. 하지만 조원들과의 토의를 통해 해결할 수 있었다. 프로젝트를 진행하면서 각각의 레지스터들을 다시 한 번 보게 되니 복습이 동시에 되고 스위치를 누름과 동시에 내가 어떠한 동작을 제어할 수 있게 되었다는 사실에 뿌듯함이 몰려왔습니다. 이번 학기를 통해 ATmega에 대해 알게 되었고 다양한 코드들도 알게 되어서 확실한 성장을 했다는 게 느껴졌습니다.

20181312 류재후 : 드디어 종강이라는 생각에 우선 홀가분한 마음이 듭니다. 한 학기 동안 임베디드 강의를 들으면서 atmega128이라는 것을 알게 되었으며 많은 실습들을 통하여 다양한 기능들을 구현해 보았습니다. 실험 실습을 진행할 때에 우여곡절이 많을 때도 있고, 이해를 잘 되지 않아 실습을 가장 늦게 끝낼 때도 있었지만 조원들과 함께 하면서 문제를 해결하는 과정을 통해 더욱 돈독해지고 실력이 향상된 것 같아서 기쁩니다. 이번 마지막 기말 프로젝트를 진행하면서 지금까지 배운 것들을 조합하여 만들었는데 따로 실습을 진행했을 때에는 문제없이 되는 것들도 하나로 모아 만들 때에는 잘 되지 않았습니다. 그럴 때 마다 교수님께서 잘 가르쳐 주신 강의를 다시 생각하고 관련 이론과 레지스터들을 다시 공부하는 과정과 모든 조원이 포기하지 않고 끝까지 붙잡고 있어서 결국 성공할 수 있었던 것 같습니다. 기말 프로젝트로 임베디드 강의를 잘 마무리 지은 것 같아 기쁩니다. 2학기때도 교수님 강의를 수강하고 싶습니다!

20181329 신승훈 : 이번 임베디드 코딩 및 실습을 기말 프로젝트를 마무리하면서 이때까지 배운 것들을 되새기면서 많은 것들을 알고 배우는 시간을 가졌습니다. 지금까지 수업에서 배운 Atmega128A와 각 장치들의 이론과 실습 및 실험을 하면서 다양한 기능들과 동작들을 구현할 수 있었습니다. 지난 아두이노 수업에 이어 이렇게 임베디드 실습을 김한솔 교수님 강의를 들을 수 있음에 기대하였고 역시 기대를 저버리지 않고 훌륭한 강의를 해주신 덕분에 많은 것을 배워가고 알아가는 시간이었습니다. 조원들과 조별 실습 및 결과 보고서를 매주 거둬가면서 실습에서 어려운 일이 닥쳐도 조원들과 머리를 맞대고 협동하여 문제를 잘 해결해 나갈 수 있어 좋았습니다. 이번 기말 프로젝트를 마무리하면서 하드웨어 및 소프트웨어 적으로 많은 시행착오를 겪어 한 단계 더 발전한 모습으로 성장한 것 같아 기쁘고 뿌듯합니다. 최종 결과물을 만들고 성공하였을 때의 기분은 이루어 말할 수 없을 정도로 감격스럽고 그동안의 노력이 결실을 맺어 나 자신

테 고생했다고 다시 말하고 싶습니다. 힘들고 어려울 때 마다 교수님의 친절한 설명과 강의가 있었기에 여기까지 올 수 있었던 것 같습니다. 다시 한번 부족함 없는 훌륭한 강의를 하신 교수님께 감사합니다. 더 준비된 상태에서 다음 학기 교수님 수업을 기다리고 기대가 되는 것 같습니다. 한 학기 동안 수고 많으셨습니다. 감사합니다. 😊👍