
# AVR - HW5

임베디드스쿨1기
Lv1과정
2020. 10. 13
손표훈

# 1. ADC

```c
#define ADC_REG 0x78

uint16_t adcValue = 0;

struct adc
{
    union
    {
        struct
        {
            uint8_t adc_l;
            uint8_t adc_h;
        };
        uint16_t adc;
    };
        uint8_t adcsr_a;
        uint8_t adcsr_b;
        uint8_t admux;
};

volatile struct adc *const adc = (void *)ADC_REG;
```

ADC결과를
저정하기 위한 변수

### 23.9.3 ADCL and ADCH – The ADC Data Register

#### 23.9.3.1 ADLAR = 0

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x79) | - | - | - | - | - | - | ADC9 | ADC8 | ADCH |
| (0x78) | ADC7 | ADC6 | ADC5 | ADC4 | ADC3 | ADC2 | ADC1 | ADC0 | ADCL |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R | R | R | R | R | R | R | R | |
| | R | R | R | R | R | R | R | R | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

#### 23.9.3.2 ADLAR = 1

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x79) | ADC9 | ADC8 | ADC7 | ADC6 | ADC5 | ADC4 | ADC3 | ADC2 | ADCH |
| (0x78) | ADC1 | ADC0 | - | - | - | - | - | - | ADCL |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R | R | R | R | R | R | R | R | |
| | R | R | R | R | R | R | R | R | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

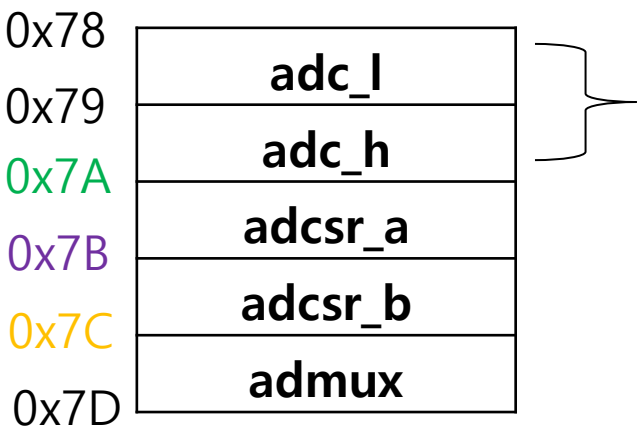
# 1. ADC

```
#define ADC_REG 0x78

uint16_t adcValue = 0;

struct adc
{
    union
    {
        struct
        {
            uint8_t adc_l;
            uint8_t adc_h;
        };
        uint16_t adc;
    };
        uint8_t adcsr_a;
        uint8_t adcsr_b;
        uint8_t admux;
};

volatile struct adc *const adc = (void *)ADC_REG;
```




**23.9.2 ADCSRA – ADC Control and Status Register A**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x7A) | ADEN | ADSC | ADATE | 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 | |

**ADCSRB – ADC Control and Status Register B**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x7B) | – | ACME | – | – | – | ADTS2 | ADTS1 | ADTS0 | ADCSRB |
| Read/Write | R | R/W | R | R | R | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

**23.9.1 ADMUX – ADC Multiplexer Selection Register**

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x7C) | REFS1 | REFS0 | ADLAR | – | MUX3 | MUX2 | MUX1 | MUX0 | ADMUX |
| Read/Write | R/W | R/W | R/W | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

구조체 첫 번째 멤버의 주소가 ADC DATA Register의 ADCL 주소로
고정된 adc구조체 포인터변수 선언으로 ADC 관련 레지스터에 접근

# 1. ADC

```
void adcInit(void)
{
    /*sbi(ADMUX, REFS0);
    sbi(ADCSRA, ADPS0);
    sbi(ADCSRA, ADPS1);
    sbi(ADCSRA, ADPS2);
    sbi(ADCSRA, ADEN);*/

    sbi(adc->admux, REFS0);
    sbi(adc->adcsr_a, ADPS0);
    sbi(adc->adcsr_a, ADPS1);
    sbi(adc->adcsr_a, ADPS2);
    sbi(adc->adcsr_a, ADEN);
}
```

```
#define ADMUX _SFR_MEM8(0x7C)
#define MUX0 0
#define MUX1 1
#define MUX2 2
#define MUX3 3
#define ADLAR 5
#define REFS0 6
#define REFS1 7
```

```
#define ADCSRA _SFR_MEM8(0x7A)
#define ADPS0 0
#define ADPS1 1
#define ADPS2 2
#define ADIE 3
#define ADIF 4
#define ADATE 5
#define ADSC 6
#define ADEN 7
```

### 23.9.1 ADMUX – ADC Multiplexer Selection Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x7C) | REFS1 | REFS0 | ADLAR | – | MUX3 | MUX2 | MUX1 | MUX0 | ADMUX |
| Read/Write | R/W | R/W | R/W | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | |

**Table 23-3. Voltage Reference Selections for ADC**

| REFS1 | REFS0 | Voltage Reference Selection |
|---|---|---|
| 0 | 0 | AREF, internal $V_{REF}$ turned off |
| 0 | 1 | $AV_{CC}$ with external capacitor at AREF pin |
| 1 | 0 | Reserved |
| 1 | 1 | Internal 1.1V voltage reference with external capacitor at AREF pin |

# 1. ADC

```
void adcInit(void)
{
    /*sbi(ADMUX, REFS0);
    sbi(ADCSRA, ADPS0);
    sbi(ADCSRA, ADPS1);
    sbi(ADCSRA, ADPS2);
    sbi(ADCSRA, ADEN);*/

    sbi(adc->admux, REFS0);
    sbi(adc->adcsr_a, ADPS0);
    sbi(adc->adcsr_a, ADPS1);
    sbi(adc->adcsr_a, ADPS2);
    sbi(adc->adcsr_a, ADEN);
}
```

```
#define ADMUX _SFR_MEM8(0x7C)
#define MUX0 0
#define MUX1 1
#define MUX2 2
#define MUX3 3
#define ADLAR 5
#define REFS0 6
#define REFS1 7
```

```
#define ADCSRA _SFR_MEM8(0x7A)
#define ADPS0 0
#define ADPS1 1
#define ADPS2 2
#define ADIE 3
#define ADIF 4
#define ADATE 5
#define ADSC 6
#define ADEN 7
```

### 23.9.2 ADCSRA – ADC Control and Status Register A

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x7A) | ADEN | ADSC | ADATE | 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 | |

- **Bits 2:0 – ADPS2:0: ADC Prescaler Select Bits**

These bits determine the division factor between the system clock frequency and the input clock to the ADC.

**Table 23-5. ADC Prescaler Selections**

| 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 |

ADC clock = system clock / 128

# 1. ADC

```
uint16_t readADC(uint8_t channel)
{
    /*ADMUX &= 0xF0;
    ADMUX |= channel;

    sbi(ADCSRA, ADSC);
    while(ADCSRA & (1 << ADSC));

    return ADC;*/

                    * 내부 1.1V 사용
    adc->admux &= 0xF0;
    adc->admux |= channel;

    sbi(adc->adcsr_a, ADSC);
    while(adc->adcsr_a & (1 << ADSC));

    return adc->adc;
}
```

### 23.9.1 ADMUX – ADC Multiplexer Selection Register

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x7C) | REFS1 | REFS0 | ADLAR | – | MUX3 | MUX2 | MUX1 | MUX0 | ADMUX |
| Read/Write | R/W | R/W | R/W | R | R/W | R/W | R/W | R/W | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

- **Bit 7:6 – REFS1:0: Reference Selection Bits**

These bits select the voltage reference for the ADC, as shown in Table 23-3. If these bits are changed during a conversion, the change will not go in effect until this conversion is complete (ADIF in ADCSRA is set). The internal voltage reference options may not be used if an external reference voltage is being applied to the AREF pin.

**Table 23-3. Voltage Reference Selections for ADC**

| REFS1 | REFS0 | Voltage Reference Selection |
|---|---|---|
| 0 | 0 | AREF, internal $V_{REF}$ turned off |
| 0 | 1 | $AV_{CC}$ with external capacitor at AREF pin |
| 1 | 0 | Reserved |
| 1 | 1 | Internal 1.1V voltage reference with external capacitor at AREF pin |

- **Bit 5 – ADLAR: ADC Left Adjust Result**

The ADLAR bit affects the presentation of the ADC conversion result in the ADC data register. Write one to ADLAR to left adjust the result. Otherwise, the result is right adjusted. Changing the ADLAR bit will affect the ADC data register immediately, regardless of any ongoing conversions. For a complete description of this bit, see Section 23.9.3 "ADCL and ADCH – The ADC Data Register" on page 219.

- **Bit 4 – Res: Reserved Bit**

This bit is an unused bit in the Atmel® ATmega328P, and will always read as zero.

### 23.9.3.2 ADLAR = 1

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x79) | ADC9 | ADC8 | ADC7 | ADC6 | ADC5 | ADC4 | ADC3 | ADC2 | ADCH |
| (0x78) | ADC1 | ADC0 | – | – | – | – | – | – | ADCL |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R | R | R | R | R | R | R | R | |
| | R | R | R | R | R | R | R | R | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

# 1. ADC

```
uint16_t readADC(uint8_t channel)
{
    /*ADMUX &= 0xF0;
    ADMUX |= channel;

    sbi(ADCSRA, ADSC);
    while(ADCSRA & (1 << ADSC));

    return ADC;*/

    adc->admux &= 0xF0;
    adc->admux |= channel;

    sbi(adc->adcsr_a, ADSC);
    while(adc->adcsr_a & (1 << ADSC));

    return adc->adc;
}
```

**Table 23-4. Input Channel Selections**

| MUX3..0 | Single Ended Input |
|---|---|
| 0000 | ADC0 |
| 0001 | ADC1 |
| 0010 | ADC2 |
| 0011 | ADC3 |
| 0100 | ADC4 |
| 0101 | ADC5 |
| 0110 | ADC6 |
| 0111 | ADC7 |
| 1000 | ADC8(1) |
| 1001 | (reserved) |
| 1010 | (reserved) |
| 1011 | (reserved) |
| 1100 | (reserved) |
| 1101 | (reserved) |
| 1110 | 1.1V ($V_{BG}$) |
| 1111 | 0V (GND) |

Note: 1. For temperature sensor.

# 1. ADC

```
uint16_t readADC(uint8_t channel)
{
    /*ADMUX &= 0xF0;
    ADMUX |= channel;

    sbi(ADCSRA, ADSC);
    while(ADCSRA & (1 << ADSC));

    return ADC;*/

    adc->admux &= 0xF0;
    adc->admux |= channel;

    sbi(adc->adcsr_a, ADSC);
    while(adc->adcsr_a & (1 << ADSC));

    return adc->adc;
}
```

### 23.9.2 ADCSRA – ADC Control and Status Register A

| Bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x7A) | ADEN | ADSC | ADATE | 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 | |

- **Bit 7 – ADEN: ADC Enable**

Writing this bit to one enables the ADC. By writing it to zero, the ADC is turned off. Turning the ADC off while a conversion is in progress, will terminate this conversion.

- **Bit 6 – ADSC: ADC Start Conversion**

In single conversion mode, write this bit to one to start each conversion. In free running mode, write this bit to one to start the first conversion. The first conversion after ADSC has been written after the ADC has been enabled, or if ADSC is written at the same time as the ADC is enabled, will take 25 ADC clock cycles instead of the normal 13. This first conversion performs initialization of the ADC.

ADSC will read as one as long as a conversion is in progress. When the conversion is complete, it returns to zero. Writing zero to this bit has no effect.

- **Bit 5 – ADATE: ADC Auto Trigger Enable**

* ADSC는 AD변환이 완료되면 자동으로 0으로 clear된다

* AD변환이 완료될 때까지 while문에서 대기

# 1. ADC

```
uint16_t readADC(uint8_t channel)
{
    /*ADMUX &= 0xF0;
    ADMUX |= channel;

    sbi(ADCSRA, ADSC);
    while(ADCSRA & (1 << ADSC));

    return ADC;*/

    adc->admux &= 0xF0;
    adc->admux |= channel;

    sbi(adc->adcsr_a, ADSC);
    while(adc->adcsr_a & (1 << ADSC));

    return adc->adc;
}
```

**23.9.3.2 ADLAR = 1**

| Bit | 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | |
|---|---|---|---|---|---|---|---|---|---|
| (0x79) | ADC9 | ADC8 | ADC7 | ADC6 | ADC5 | ADC4 | ADC3 | ADC2 | ADCH |
| (0x78) | ADC1 | ADC0 | – | – | – | – | – | – | ADCL |
| | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 | |
| Read/Write | R | R | R | R | R | R | R | R | |
| | R | R | R | R | R | R | R | R | |
| Initial Value | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |

ADC Data Register의 결과를 Reteurn함

여기서 ADC0~9는 채널이 아님!!
Bit임

# 2. Volatile 키워드

(1) Volatile 키워드를 사용하는 이유?

- 컴파일러의 최적화 옵션을 피하기 위해 사용한다.
- 컴파일러의 최적화 옵션이란?(아래 그림1을 참고)
  : adc라는 구조체를 선언하고 adc 구조체 변수를 이용해 adc관련 레지스터에 접근한다.
  이 때 구조체 변수 adc를 volatil로 선언하지 않으면 adcInit함수에서 컴파일러의 최적화에 중복된 코드 사용을 줄여 마지막줄의 ADEN만 설정이 된다.

```
#define ADC_REG 0x78

uint16_t adcValue = 0;

struct adc
{
    union
    {
        struct
        {
            uint8_t adc_l;
            uint8_t adc_h;
        };
        uint16_t adc;
    };
        uint8_t adcsr_a;
        uint8_t adcsr_b;
        uint8_t admux;
};

volatile struct adc *const adc = (void *)ADC_REG;
```

```
void adcInit(void)
{
    /*sbi(ADMUX, REFS0);
    sbi(ADCSRA, ADPS0);
    sbi(ADCSRA, ADPS1);
    sbi(ADCSRA, ADPS2);
    sbi(ADCSRA, ADEN);*/

    sbi(adc->admux, REFS0);
    sbi(adc->adcsr_a, ADPS0);
    sbi(adc->adcsr_a, ADPS1);
    sbi(adc->adcsr_a, ADPS2);
    sbi(adc->adcsr_a, ADEN);
}
```

# 2. Volatile 키워드

(1) 임베디드 Volatile 키워드를 사용하는 이유? (Memory Map I/O (Register) 설정시)

- 컴파일러의 최적화 옵션을 피하기 위해 사용한다.
- 컴파일러의 최적화 옵션이란?(아래 그림1을 참고)
  : adc라는 구조체를 선언하고 adc 구조체 변수를 이용해 adc관련 레지스터에 접근한다.
  이 때 구조체 변수 adc를 volatil로 선언하지 않으면 adcInit함수에서 컴파일러의 최적화에 중복된 코드 사용을 줄여 마지막줄의 ADEN만 설정이 된다.

```
#define ADC_REG 0x78

uint16_t adcValue = 0;

struct adc
{
    union
    {
        struct
        {
            uint8_t adc_l;
            uint8_t adc_h;
        };
        uint16_t adc;
    };
        uint8_t adcsr_a;
        uint8_t adcsr_b;
        uint8_t admux;
};

volatile struct adc *const adc = (void *)ADC_REG;
```

```
void adcInit(void)
{
    /*sbi(ADMUX, REFS0);
    sbi(ADCSRA, ADPS0);
    sbi(ADCSRA, ADPS1);
    sbi(ADCSRA, ADPS2);
    sbi(ADCSRA, ADEN);*/

    sbi(adc->admux, REFS0);
    sbi(adc->adcsr_a, ADPS0);
    sbi(adc->adcsr_a, ADPS1);
    sbi(adc->adcsr_a, ADPS2);
    sbi(adc->adcsr_a, ADEN);
}
```

# 2. Volatile 키워드

(1) 임베디드 Volatile 키워드를 사용하는 이유?

- 인터럽트 핸들의 경우..
- 메인문과 인터럽트에서 공유하는 전역변수가 있을 경우..
- 아래 코드에서 Ala와 같이 상수?로 선언된 변수가 vaoltaile이 아니라면 컴파일러는 실행속도를 높이기 위해 레지스터에 Ala의 값을 3으로 저장한다.
- 인터럽트 ISR(context switching)에 의해 Ala는 1증가 하지만 Pepe로 돌아왔을 때 itmp의 return값은 100이 된다..(Ala가 Hardware Register에 3으로 최적화 되어 저장되어 있기때

- Volatile키워드를 사용하면 최적화에서 제외시킨다!(결국 최적화에서 제외가 목적?)

```
volatile int Ala;

int Pepe(void)
{
  int itmp;
  Ala = 3;
  if (Ala == 3)
  {
    itmp = 100;
  }
  else
  {
    itmp = 1234;
  }
  return itmp;
}
```

인터럽트 발생!

```
void interrupt Timer1_Output_Compare_1A(void)
{
   Ala ++;
}
```

추가 공부내용 : 예제 코드를 어셈으로 확인해보는 방법을 찾아보자..
반복되는 변수접근시 컴파일러 최적화에 의해
로드/스토어 명령어가 줄어든다고 한다..
ISR이 포함된 코드를 어셈으로 확인이 필요.