AVR - HW5

임베디드스쿨1기

Lv1과정

2020. 10. 16

박하늘

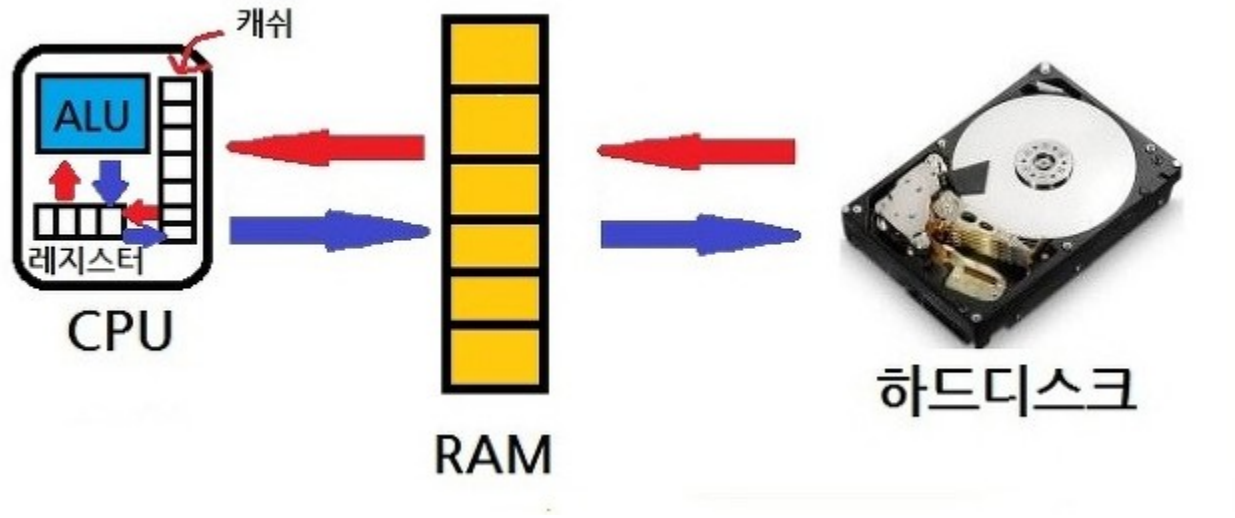
1. volatile 변수

1) 정의

volatile의 기능적 의미는 캐시 사용 안 함(No-Cache)이다. 보통 프로그램이 실행될 때 속도를 위해 필요한 데이터를 메모리에서 직접 읽어오지 않고 Cache로부터 읽어온다. 하지만, 하드웨어에 의해서 변경되는 값들은 Cache에 즉각적으로 반영되지 않으므로 데이터를 Cache 로부터 읽어오지 말고 주 메모리 에서 직접 읽어오도록 해야 한다. 이러한 특성 때문에 하드웨어가 사용하는 메모리 는 volatile로 선언해야 하드웨어에 의해 변경된 값들이 프로그램에 제대로 반영 된다.
volatile은 Embedded Software에서 자주 사용하는 타입 한정자이다.

2) 필요한 경우

1) Optimize Option을 켜기 전까지는 코드가 잘 동작한다.
2) 어떤 인터럽트를 disable 시킨 동안에는 코드가 잘 동작한다.
3) RTOS가 탑재된 Mutitasking System에서 어떤 Task가 enable 되기 전까지는 Task가 잘 동작한다.



1. volatile 변수

3) 사용하는 대상 및 Example

3-1) Memory-mapped peripheral registers

:주변 디바이스(Peripheral)들은 프로그램 흐름과 비동기적으로 값들이 변하는 레지스터들을 가지고 있는 경우가 대부분임

```
INT8U *ptr = (INT8U *)0X1234;
while(*ptr == 0);
```

→ 0x1234 Address에 위치한 8bit Status Register가 0이 아닌 값을 가질 때까지 매번 메모리에 Access하여 Polling한다.

3-2) 인터럽트 서비스 루틴

```
int ETXRcvd = FALSE;
void main(void)
{
        while(!ETXRcvd)
        {
                //something
        }
}
Interrupt void RXISR(void)
{
        if(rx_char = ETX)
        {
                ETXRcvd = TRUE;
        }
}
```

→ Optimize Option을 켜놓게 되면 ETXRcvd의 값을 Register에 두고 연산을 처리하게 되어 인터럽트 서비스 루틴에 의해서 값이 바뀌었을 경우 이를 알수 없는 경우가 생긴다. 이를 방지하기 위해 volatile로 값의 Read / Write를 메모리에 직접 접근하여 수행하게끔 한다.

3-3) Multitasking 또는 Multithread

```
int strawberry(int arg)
{
        int *dummy;
        dummy = &arg;

        *dummy = 1;
        *dummy = 2;
        *dummy = 3;
        *dummy = 4;
        *dummy = 5;
        return *dummy;
}
```

→ dummy 변수를 메모리에 저장하여 사용하지 않고 레지스터에 올려놓고 사용할 경우 다른 스레드에서는 공유 변수의 변경을 알 수가 없게 되어 잘못된 Communication으로 오류가 발생할 수 있다.

1. volatile 변수

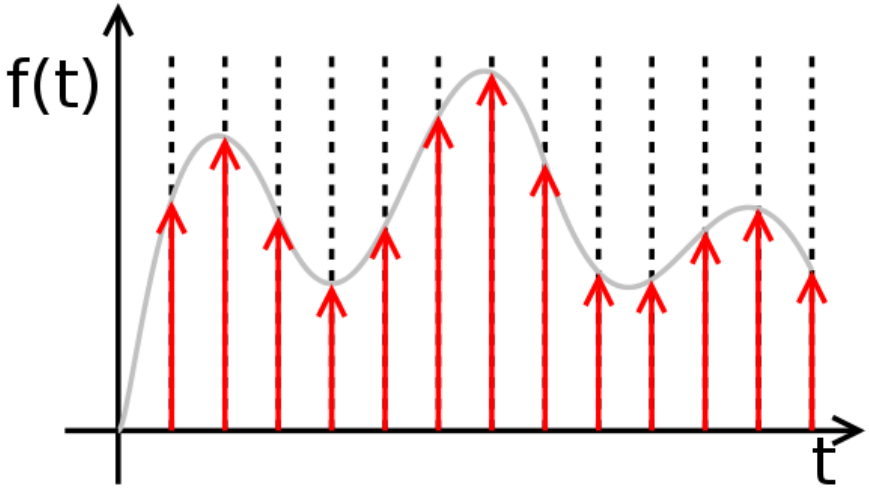
4) 사용시 고려사항

1) 최적화를 막게 되어 프로그램 수행 속도가 떨어짐
2) 어떤 장치의 한 영역을 나타내는 변수가 있고 이 변수를 통하여 그 장치의 상황을 받아들여 특정한 일을 처리함에 있어서 자주 값이 변경되는 경우인지 확인한다.
3) volatile 선언은 디버그 모드에서는 작동하지 않고 보통 Release Mode에서 최적화 시 발생

2. [Review] ADC 개념

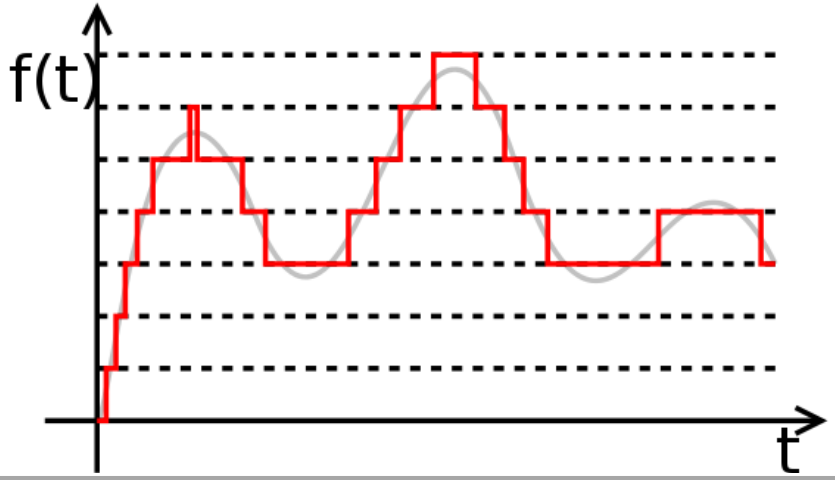
1) 표본화

: 아날로그 신호를 일정한 시간 간격으로 구간을 나눈 후 그 시간간격에 해당하는 아날로그 신호값을 채취
(연속된 시간의 신호를 이산 시간 신호로 변환)



2) 양자화

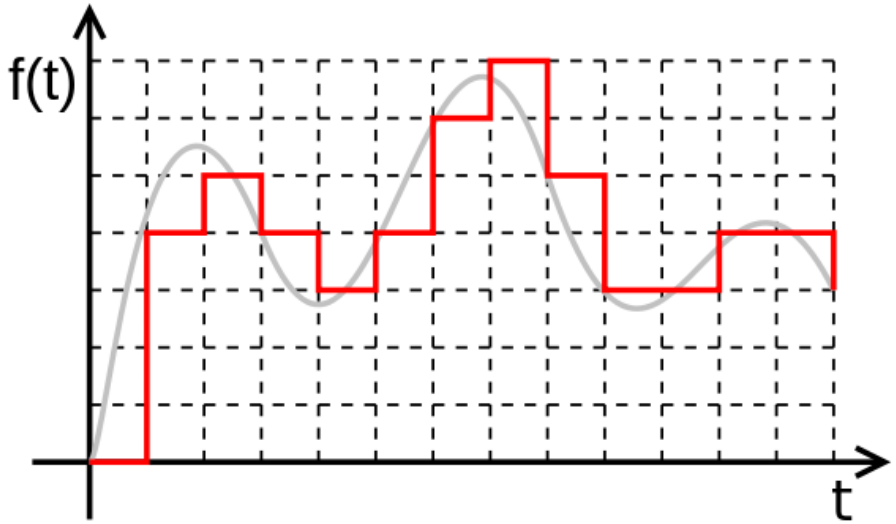
: 표본화 과정으로 아날로그 신호를 일정한 전압 레벨의 구간을 나눈 영역에 강제적으로 대응 시키는 과정
(샘플링된 신호의 진폭을 제한된 디지털 레벨로 변환)



2. [Review] ADC 개념

3) 부호화

:표본화와 양자화를 포개어 실제적으로 0,1의 디지털로 처리할 수 있도록 구성



- 이론적으로 단순하게 두 배 이상의 주파수로 샘플링 한 후 양자화를 하면 디지털 데이터가 나온다고 하지만 회로 구성 까다롭고,A/D변환 방법이 다양하여 속도와 정확성에 차이가 있다.
따라서, 적당한 경제적인 범위 내에서 적절한 성능을 택하도록 한다. (나이퀴스트 정리)

4) 나이퀴스트 이론

샘플링 법칙에 따르면 샘플링 주파수 fs는 신호의 최대 주파수 f의 두 배 이상이 되어야 함.
이보다 낮을 경우 앨리어싱(Aliasing)이 발생하므로 원래 신호가 왜곡된다. 이때, fs/2를 중첩주파수 or 나이퀴스트 주파수라고 한다.

2. [Review] ADC 용어

1)AVCC

:아날로그 전원 (Vcc에서 끌어온 다음 필터 처리하여 노이즈를 제거한 후 사용한다)

2)AVREF(Analog Reference Voltage)

: 변환작업에 사용되는 기준 전압값

전원전압 VCC를 초과할 수 없다.(VREF=VGND~VCC, 내부기준전압 2.56V)

3)분해능

외부에서 아날로그 신호가 들어 오게 되면 sin파가 입력되게 됨.

그런데 예를들어 이 아날로그 신호가 어떻게 변화가 되는지를 1초에 한번씩 체크하는 것과 0.1초마다 체크하는 것, 그리고 0.01 초마다 체크 하는 것은 다름.

얼마나 세밀하게 체크를 하는가가 얼마나 "분해"를 하는 것인가와 의미가 같음.

좀더 세밀하게 체크할 수 있다면 보다 정확한 데이터를 구할 수가 있음. 이것이 분해할 수 있는 능력이라고 함

2. [Review] ADC Code

```c
#define F_CPU 16000000UL

#include <avr/io.h>
#include <util/delay.h>
#include <stdlib.h>
#include <string.h>

#define sbi(PORTX, BitX) (PORTX |= (1 << BitX)) //비트 set
#define cbi(PORTX, BitX) (PORTX &= ~(1 << BitX))//비트 clear

uint16_t adcValue = 0;

#define ADC_REG 0x78

struct adc  //구조체 선언
{
    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;// Volatile 변수선언.
/*mutax, semaphore를 사용하기 위함. 최적화 금지하고 FM대로 메모리에 접근

void adcInit(void){

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

1. 사용자 구조체 선언

: uint_8t 총 5개로 이루어진 구조체 선언 (40bit)

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
(0x7C)	ADMUX	REFS1	REFS0	ADLAR	–	MUX3	MUX2	MUX1	MUX0	217
(0x7B)	ADCSRB	–	ACME	–	–	–	ADTS2	ADTS1	ADTS0	220
(0x7A)	ADCSRA	ADEN	ADSC	ADATE	ADIF	ADIE	ADPS2	ADPS1	ADPS0	218
(0x79)	ADCH	ADC data register high byte								219
(0x78)	ADCL	ADC data register low byte								219

2. volatile

: 컴파일러는 해당 변수를 최적화에서 제외하여 항상 메모리에 접근가능
즉, 0x78~0x7C가 씹히지 않고 모두 메모리에 접근토록 volatile변수 선언

3. *const

: 변수의 초기값을 변경할 수 없는 변수를 const 상수

4. (void*)

: #define ADC_REG 0X78이 상수이므로,
상수를 주소연산이 가능하게 끔 형 변환 시켜줌

5. adcInit(void)

:set bit로 해당 레지스터 활성화 → volatile로 변수 선언하였기 때문에 모두 메모리에 접근가능하게 됨

2. [Review] ADC Code

```c
uint16_t readADC(uint8_t channel)
{

    adc->admux &= 0XF0;          //핀 초기화
    adc->admux |= channel;       // 사용할 ADC 셋팅 설정

    sbi(adc->adcsr_a, ADSC);     //ADC Start conversion ACC 변환 시작
    while(adc->adcsr_a & (1<<ADSC));     //ADC 변환 완료까지 대기

    return adc->adc;
}
```

```c
void UART_INIT(void)
{
    sbi(UCSR0A, U2X0);  //U2X0 = 1 -> Baudrate 9600 = 207 , 속도 2배

    UBRR0H = 0x00;
    UBRR0L = 207;       //Baudrate 9600

    UCSR0C |=0X06;

    sbi(UCSR0B , RXEN0);
    sbi(UCSR0B, TXEN0);     //송수신 Enable
}
```

Control

```c
unsigned char UART_receive(void)
{

                        <<RXC0)));  //UCSR0A의 RXC0값이 1인지 확인 작업


}
unsigned char UART_tranmit(unsigned char data)
{
    while(!(UCSR0A & (1<<UDRE0)));  //UCSR0A의 UDRE0값이 1인지 확인 작업
    UDR0 = data;
}
```

Status, Data

6. readADC

: ADC Status, Data 부분

사용할 ADC셋팅을 설정하고, ADC 변환 시작 및 상태 체크

7. UART_INIT(void)/ UART_receive(void) / UART_tranmit(unsigned char data)

: 1. 통신 속도 설정

	f_{osc} = 16.0000MHz			
	U2Xn = 0		U2Xn = 1	
Baud Rate (bps)	UBRRn	Error	UBRRn	Error
2400	416	–0.1%	832	0.0%
4800	207	0.2%	416	–0.1%
9600	103	0.2%	207	0.2%

2. 데이터 설정

Bit	7	6	5	4	3	2	1	0	
	UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn	UCSRnC
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	1	1	0	

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

1) UCSR0C = 0x06 // 0000 0110 data 8bit 설정
2) UCSR0A & (1 << RXC0) // 수신버퍼가 비어 있다면 0으로 클리어, 1이 되면 데이터 수신 받았다는 것
3) UCSR0A & (1<<UDRE0) // 전송버퍼가 비어 있다면 1 값을 가지고 새로운 데이터를 받아서 전송할 준비 완료된 것

2. [Review] ADC Code

```c
void UART_string_transmit(char *string)      //*string 문자열의 시작주소
{

    while(*string != '\0')                   //문자열 맨 마지막 문자("\0") 확인
    {

        UART_tranmit( *string);
        string++;
    }
}
int main(void)
{
    /* Replace with your application code */

    //unsigned char data;

    char str;
    char a[3] = {'\0',};//unsigned char이 3byte이므로, char a[3] = {'\n',}; --
    UART_INIT();        //UART초기화
    adcInit();

    while (1)
    {
        adcValue = readADC(0);

        itoa(adcValue, a, 10);
        UART_string_transmit(a);
        UART_string_transmit("\n");
        _delay_ms(100);

    }
    return 0;
}
```

8. UART_string_transmit(char* string)

: 문자열을 받아 송신을 보냄

마지막 문자가 일치 할때까지 순환

9. itoa(adcvalue,a,10);

ADC값(adcvalue)의 integer 자료형을 문자열 자료형으로 변환

a: 배열의 주소값이 저장되어 있음

a에 저장된 문자열을 UART_string_transmit으로 문자열을 체크하여 마지막 문자 상태 확인

■ itoa()란?

:정수형을 문자열로 변환

(리눅스에서는 itoa함수가 인식되지 않으므로 sprintf 함수로 대체)

→ sprintf(버퍼, :형식지정자",값)

```c
#ifdef __DOXYGEN__
extern char *itoa(int val, char *s, int radix);
#else
extern __inline__ __ATTR_GNU_INLINE__
char *itoa (int __val, char *__s, int __radix)
{
    if (!__builtin_constant_p (__radix)) {
    extern char *__itoa (int, char *, int);
    return __itoa (__val, __s, __radix);
    } else if (__radix < 2 || __radix > 36) {
    *__s = 0;
    return __s;
    } else {
    extern char *__itoa_ncheck (int, char *, unsigned char);
    return __itoa_ncheck (__val, __s, __radix);
    }
}
#endif
```

1. 매개변수
- val: 변환할 정수값 명시, 이 값을 기준으로 변환
- string: 변환된 문자열이 저장될 배열, 포인터를 명시
- radix: 변환할때 사용할 진법을 명시

2. 함수의 반환값
- 자신이 넘겨준 배열 or 포인터의 시작 주소값이 반환됨 (string변수에 명시한 주소 값이 그대로 반환)

3. 해더
#include "stdlib.h"

감사합니다.