INSTUTIONEN FÖR
INFORMATIONSTEKNOLOGI

PROJEKTRAPPORT

DIGITALT OSCILLOSKOP PÅ PC VIA PARALLELLPORTEN

VT 2004

August Bering e98abe@efd.lth.se
Andreas Nevalainen e99an@efd.lth.se

Abstract

This document describes a PC controlled oscilloscope. The oscilloscope is a simple and low cost implementation for the home electrician. It only contains two circuits, one AVR ATmega 16 processor and one ADC0820 AD-converter. The oscilloscope is controlled through a program window in the PC, the signal is also presented in this window. The controlles available is setting the trigg level and if the scope should trigg on a rising or falling flank. The code for the user interface is programmed in C++ and all the control is programmed in C.

1. Inledning

Målet var att implementera ett så enkelt och billigt digitalt oscilloskop som möjligt innehållande de vanligaste funktionerna. Oscilloskopet visar sina data i ett Windowsprogram på en vanlig hemma PC. Insamplingen skulle vara så enkel och flexibel som möjligt, därför används en enchipsdator direkt kopplad till en AD-omvandlare. Detta ger en måttlig samplingshastighet, men en mycket enkel design.

1.1. Kravspecifikation

Oscilloskopet skall uppfylla följande krav:

- Endast en ingångskanal
- 400kHz samplignshastighet
- Variabel samplingshastighet från 500Hz till maxhastigheten i 2 potenser
- 8 bit/sampel i upplösning
- Variabel triggning på både stigande och fallande flank

1.2. Funktion

Triggning:
AVR-processorn läser kontinuerligt in värden från AD-omvandlaren i högsta möjliga hastighet. Värdena jämförs sedan med triggvärdet och det kontrolleras om det är en stigande eller fallande flank.
Sampling:
När triggpuls fås börjar insamplingen till minnet (buffern är 512 bytes stor). Eventuellt delay för inställning av samplingshastighet sker med nop-instruktioner. När minnet är fullt överförs hela buffern till datorn via SPI.
PC-delen:
Vågformen ritas upp i ett fönster i Windows, i fönstret kan även triggningen ställas.

1.3. Eventuella utökningar:

I ett kommersiellt oscilloskop finns många finesser som inte är möjliga att implementera med den enkla lösning som används här. Dels på grund av att minnet endast är 512 byte, dels att det inte finns något riktigt ingångssteg. Det finns dock en hel del funktioner som är möjliga att utöka implementationen med, till exempel:

- Kontinuerlig överföring av data vid låga samplingshastigheter
- Logikanalysatorfunktion för analys av digitala kretsar, där varje sample sparas som 1 bit direkt från ett ingångsben

- Delayinställning, där samplingen är förskjuten från triggpulsen
- 2 eller 4 kanaler
- Automatisk delay-shiftning så att hela perioden på snabba repetetiva förlopp kan överföras över flera perioder
- Överföring via USB istället för parallellporten
- Externt minne för sampling av långa förlopp som kan frysas för detaljstudier

2. Hårdvara

2.1. Processor, AVR ATmega 16

Eftersom en enkel konstruktion eftersträvades valdes Amtels AVR ATmega 16 som är en 8-bitars enchipsdator med en maximal beräkningskapacitet av 16 MIPS vid 16 MHz. Nackdelen är att den enbart har 1Kb minne, vilket är i minsta laget för ett oscilloskop. Fördelen är att det inte behövs några andra kretsar än en AD-omvandlare.

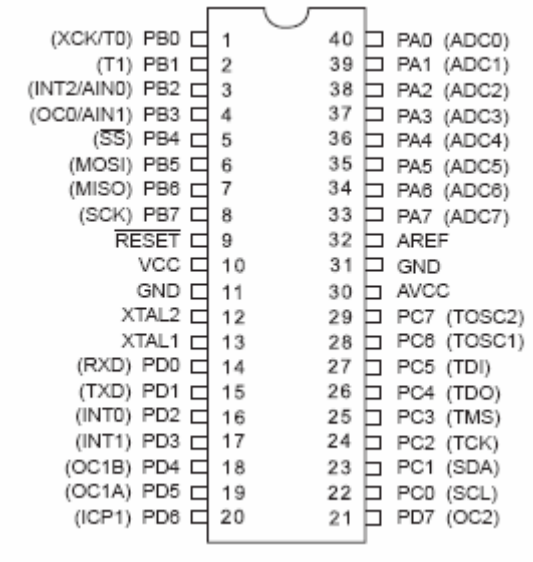

Fig. 1 Pin-konfiguration, AVR ATmega16

2.2. AD-omvandlare, ADC0820

AD-omvandlaren är en viktig del i ett oscillosop, då det är där de största kraven på noggrannhet och hastighet ligger. Att hitta en optimal AD-omvandlare visade sig vara svårt, då de flesta var alldeles för avancerade och dyra. Valet föll på National Semiconductors ADC0820. AD0820 är en 8-bitars omvandlare och har en omvandlingstid på 1.5µs.

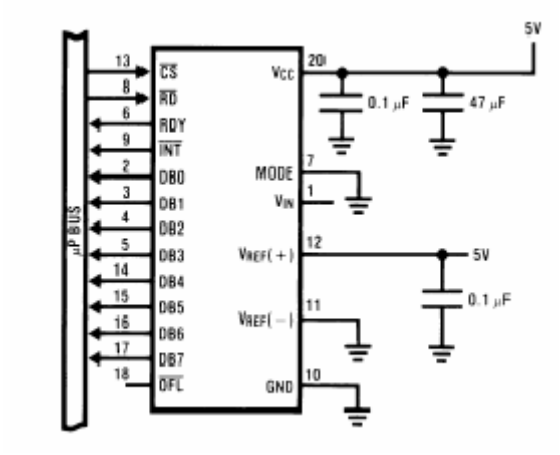

Fig. 2 ADC0820 med avkopplingskondensatorer

2.3. PC-gränssnitt

För kommunikation mellan PC:n och oscilloskopet används parallellporten. Den är ansluten till AVR-processorn via SPI, Serial Peripheral Interface. Detta är ett seriellt protokoll och skälet till att det inte använder ett parallellt är att det inte stöds av hårdvara i AVR:n. Det gör däremot SPI och simuleras i mjukvara under Windows. Till SPI–interfacet behövs 4 kablar, SCK (SPI-clock), MISO (master in, slave out), MOSI (master out, slave in) och GND. SCK är klocksignalen som styr överföringen, varje gång den går hög shiftas en bit på MOSI från PC:n, och en bit på MISO till PC:n.
På parallellporten används följande ben för SPI-interfacet:

Ben	Funktion
10	MISO
16	SCK
17	MOSI
25	GND

Tabell 1 Benkonfiguration för parallellporten

3. Mjukvara

Den stora arbetsuppgiften i detta projekt låg i att programmeringen och att få den att fungera korrekt. Programkoden är till största del skriven i C, det är enbart visningsfönstret i Windows som är skrivet i C++.

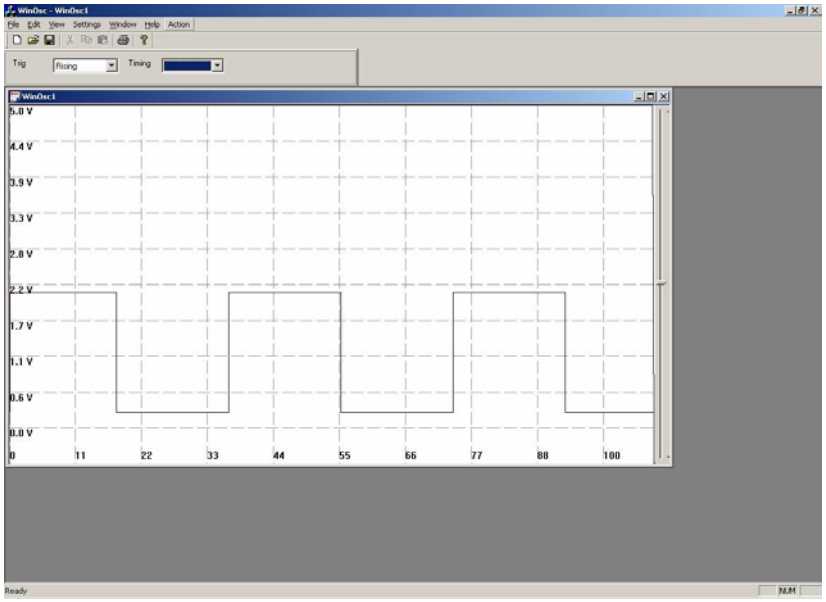

Fig. 3 Screenshot från PC-programmet

3.1. PC-gränssnitt

Den största uppgiften i programmeringen är kommunikationen mellan PC:n och AVR-processorn. I SPI-interfacet finns det en Master och en Slave. Det finns egentligen ingen möjlighet för enheten som är Slave att skicka data på eget initiativ, eftersom Mastern driver klockan. Därför måste Mastern fråga efter nya data när den vill ha det. Protokoll för kommunikationen mellan PC:n och AVR-processorn utformades enligt nedanstående:

```
enum OscCommand
{
        OSC_NOP=0x00,//no command
        OSC_BREAK,

        OSC_SEND_DATA,//message from avr when it's sending a data page.
        OSC_SET_OSC_MODE,//next: OscMode
        OSC_ENABLE_INPUT,//next: 0=ADC0, 1=ADC1, 20=ADC0820 connected to PORTD
        OSC_GET_BUFFER,
        OSC_SET_DELAY, // delay= 2^<next> clock cycles.

        OSC_ERR=0x55,
        OSC_ACK=0x56,
        OSC_BUSY,
        OSC_BUFFER_EMPTY,
        OSC_BUFFER_FULL
};
```

Funktionen är följande, när PC:n vill ha en ny buffer skickas först ett OSC_SET_OSC_MODE, därefter ett OSC_GET_BUFFER som följs av ett antal OSC_NOP. Dessa OSC_NOP används för att kunna ta emot den data som AVR:n skickar till PC:n eftersom det enligt definitionen av SPI-interfacet enbarat är möjligt för PC:n att ta emot data samtidigt som den skickar.

3.1.1. AVR-sidan

AVR:n har ingen möjlighet att tala om att den har fyllt sin buffer med sampels eftersom den är Slave, utan den väntar tills den får en förfrågan. Kommunikation i AVR:n är interrupt styrd, och varje gång ett byte tas emot via SPI genereras ett hårdvaruinterrupt. Interruptrutinen lägger dessa bytes i en buffer som sedan accessas av huvudprogrammet. Ett ungefärligt utseende av huvudloopen i interruptrutinen kan ses nedan:

```
while (1)
{
        char command = SPI_Read();
          switch (command)
          {
          case OSC_BREAK:
            SPI_Write(OSC_ACK);
            break;

          case OSC_GET_BUFFER:
            SPI_Write(OSC_ACK);
            SendDataBuffer();
            break;

          case OSC_SET_OSC_MODE:

            SPI_Write(OSC_ACK);
            char mode=SPI_Read();
            SetMode(mode);
            break;
        ….
          }
}
```

SPI_Read() och SPI_Write() är blockerande, det vill säga de returnerar inte förens det finns data i buffern, respektive att det finns plats i buffern.

3.1.2. PC-sidan

Under Windows finns ingen möjlighet att direkt läsa och skriva till parallporten, därför används en särskild driver som på ett ungefär simulerar en parallellport under DOS, den heter input32 och kan laddas ner från http://www.lvr.com/parport.htm.
För att läsa och skriva till och från porten används följande funktion:

```
#define PORT          0x378   /* Parallel port address */
#define SCK           0x04    /* Control port; pin 16 (PINIT) */
#define MOSI          0x08    /* Control port; pin 17 (!SLIN) */
#define NRESET        0x20    /* Status port; pin 12 (PEND) */
#define MISO          0x40    /* Status port; pin 10 (NACK) */
#define GET(PIN)      ((inportb(PORT+1) & PIN) != 0)
#define MOSI0         {outportb(PORT+2,(spishadow |=  MOSI));}
#define MOSI1         {outportb(PORT+2,(spishadow &= ~MOSI));}
#define SCK1          {outportb(PORT+2,(spishadow |=  SCK)); DELAY(SHORTDELAY);}
#define SCK0          {outportb(PORT+2,(spishadow &= ~SCK)); DELAY(SHORTDELAY);}

unsigned char spiwr(unsigned char cin)
{
        unsigned char i, cout;
        for (i = 0x80; i > 0; i >>= 1) { //shiftar ut bit för bit, och klockar varje med SCK.
          SCK0;
          if ((cin & i) != 0) {MOSI1;} else {MOSI0;}
           SCK1;
           cout = (cout << 1) | GET(MISO);
        }
        return cout;
}
```

SHORTDELAY anpassas till parallellportens hastighet, men ligger typiskt på 100uS.

3.2. AD-omvandlingen.

AD-omvandlingen är mycket enkelt programmerad.

1. Först sätts PORTB = 0 för att starta omvandlingen
2. Därefter sätts PB0 = 1 för att latcha resultatet
3. Sedan läses resultatet från PIND och placeras i den 512 bytes stora databuffer. När buffern är full slutar inläsningen och AVR:n väntar på ett OSC_GET_BUFFER kommando

När data läses in till databuffern sker detta i en loop. Tyvärr måste programmet använda jump-instruktioner till loopen och dessutom måste PB0 växla mellan 0 och 1 för att konvertera respektive latcha. Totalt blir det ca. 20 klockcykler mellan varje sampling.

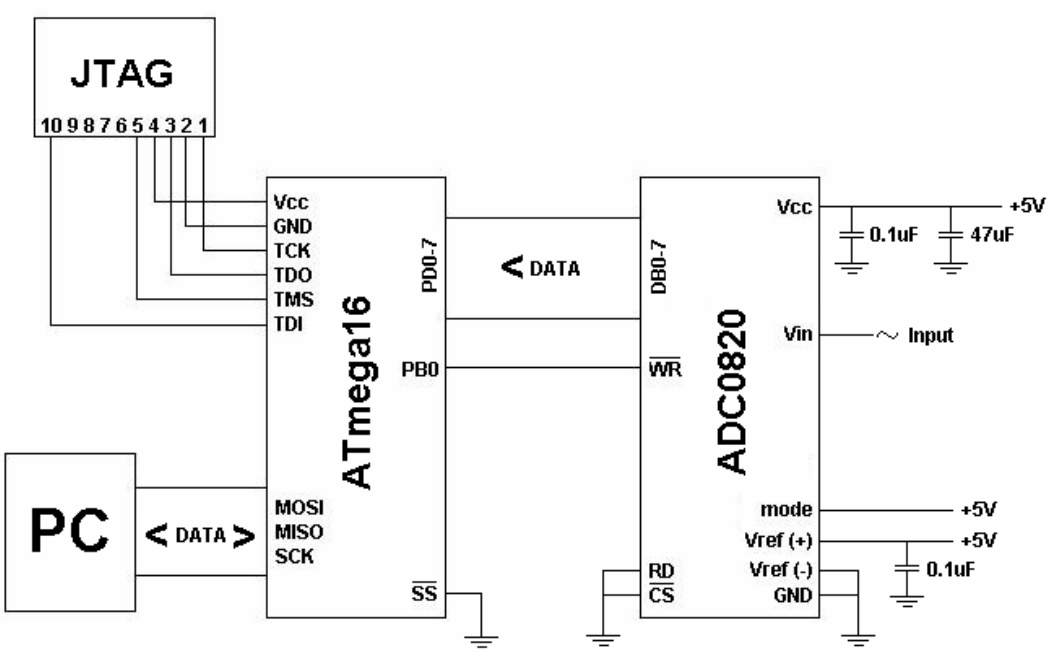

Fig. 3 Kopplingsschema

4. Resultat

Att försöka implementera ett enkelt oscilloskop visade bli ett lagom stort projekt för en sju veckors kurs. Eftersom det är en väldigt enkel konstruktion i vår implementation, endast två kretsar, låg den största arbetsbördan på programmeringssidan. Trots enkelheten i konstruktionen gick kopplandet förvånansvärt smärtfritt och bortsett från vissa smärre felkopplingar fungerade konstruktionen nästan omedelbart. Det färdiga oscilloskopet fungerar tillfredsställande och vi har lyckats uppfylla alla kraven bortsett från hastighetsinställningen. Den kör även bara i 200K sample/s. Detta går dock att lösa enkelt genom att ansluta en yttre kristall till AVR:n och klocka den i 16MHz. Den körs nu bara på 8MHz eftersom den interna klockan inte går snabbare, och samplingshastigheten blir därför enbart 200K sample/s.
Oscilloskopet kan utökas med ganska många fler funktioner och finesser, med förhoppningsvis inte allt för mycket arbete men de finns tyvärr inte med i denna implementation.

5. Bilagor

5.1 Programkod

5.1.1. ocs.c

```
#include <inttypes.h>
#include <avr/io.h>
//#include <avr/interrupt.h>
#include <avr/signal.h>
#include "../osc_messages.h"
#include "global_defines.h"
#include "adc0820.h"

#define START_CONVERSION()  ADCSRA|=BV(ADSC)//start conversion
#define WAIT_FOR_CONVERSION() loop_until_bit_is_set(ADCSRA,ADIF)
#define GET_CONV_VALUE() (ADCH)

int8_t SPI_Read(void);
void SPI_Write(int8_t byte);
void SPI_Init(void);
void SetPWMValue(char val);
void PWM_Init(void);
unsigned char DelayExponent=0;//delay between ad-conversion.
int8_t Data[DATA_SIZE];
int8_t Flags,TrigLevel=10,TrigType=0;
char Inport=20;//0=ADC0,20=ADC0820 on PORTD

void SampleSingleValuesADC0(void)
{
        while (SPI_Read()!=OSC_BREAK)
        {
          START_CONVERSION();
                        WAIT_FOR_CONVERSION();

                        SPI_Write(GET_CONV_VALUE());

                        //loop_until_bit_is_clear(Flags,FLAGS_OpCompleteBit);
                        //ReturnValue=GET_CONV_VALUE();

                        //Flags|=_BV(FLAGS_OpCompleteBit);

          //val=ADCH;
        }
        SPI_Write(OSC_ACK);
}

void ADC0_Init (void)
```

```
{
        ADCSRA=_BV(ADEN)|_BV(ADPS0);//enable ad
        ADMUX=_BV(REFS0)|//turn on Vcc voltage ref
          _BV(ADLAR);//and use left adjusted result
}

//as fast as possible
void ADC0_SampleData(void)
{
        START_CONVERSION();
        for (int16_t i=0;i<DATA_SIZE;i++)
        {
          WAIT_FOR_CONVERSION();
          Data[i]=GET_CONV_VALUE();
          START_CONVERSION();
        }
        SPI_Write(OSC_ACK);
}

void SendDataBuffer(void)
{
  for (int i=0;i<DATA_SIZE;i++)
        {
          SPI_Write(Data[i]);
          if (SPI_Read()==OSC_BREAK)
                        break;
        }
        SPI_Write(OSC_ACK);
}

void SetMode(char mode)
{
            switch (mode)
            {
                        case OSCMODE_SINGLE_VALUE:
                        SPI_Write(OSC_ACK);
                        SampleSingleValuesADC0();
                        break;
                        case OSCMODE_PAGE:
                        switch (Inport)
                        {
                                    case 0://adc0
                                    SPI_Write(OSC_ACK);
                                    ADC0_SampleData();
                                    break;
                                    #ifdef ADC0820_H
                                    case 20:
```

```
                                        SPI_Write(OSC_ACK);
                                        //Disable spi interrupts
                                        SPCR=_BV(SPE);
                                        ADC0820_SampleData(Data);
                                        SPCR=_BV(SPE)|_BV(SPIE);//enable SPI
                                        break;
                                        #endif

                                        default:
                                        SPI_Write(OSC_ERR);
                                        break;
                                      }
                                break;

                                default://urecognize
                                SPI_Write(OSC_ERR);
                              }
}

int main (void)
{
   ADC0_Init ();
        SPI_Init();
        //PWM_Init();
        #ifdef ADC0820_H
        ADC0820_Init();
        #endif
        while (1)
  {
          char command=SPI_Read();
          switch (command)
          {
            case OSC_BREAK:
            SPI_Write(OSC_ACK);
            break;

            case OSC_GET_BUFFER:
            SPI_Write(OSC_ACK);
            SendDataBuffer();
            break;

            case OSC_SET_OSC_MODE:

            SPI_Write(OSC_ACK);
            char mode=SPI_Read();
            SetMode(mode);
            break;
```

```
            case OSC_SET_DELAY:
            SPI_Write(OSC_ACK);
            DelayExponent=SPI_Read();
            SPI_Write(OSC_ACK);
            break;

            case OSC_SET_TRIG_TYPE:
            SPI_Write(OSC_ACK);
            TrigType=SPI_Read();
            SPI_Write(OSC_ACK);
            break;

            case OSC_NOP:
            break;
            /*
            case OSC_SET_PWM_VAL:
            SPI_Write(OSC_ACK);

                        SetPWMValue(SPI_Read());
                        SPI_Write(OSC_ACK);
                        break;
                        */
            default:
            SPI_Write(OSC_ERR);
            break;
          }
        }
}
```

5.1.2. adc0820.c

```
#define DATA_SIZE 512
#include <inttypes.h>
#include <avr/io.h>
#include "global_defines.h"

extern int8_t TrigType,TrigLevel;
extern unsigned char DelayExponent;

static void Delay(char delay_exp)//delay=2^delay_exp
{
        if (delay_exp==0)
          return;
        else
        {
          Delay(delay_exp-1);
          Delay(delay_exp-1);
```

```
        }
}

void ADC0820_Init(void)
{
        //setup port D up for input with
        DDRD=0x00;
        //And disable pullup
        PORTD=0x00;

        //PB0 as output (WR to ADC0820)
        DDRB=DDRB|_BV(PB0);
}

//as fast as possible
void ADC0820_SampleData(int8_t *data)
{
        unsigned char prevVal,val;
        //wait for trigger
        switch (TrigType)//if (Flags & _BV(TrigUpBit))
        {
          case 0:
          //rising
          val=255;
          do
          {
                        //start conversion
                        PORTB=0;
                        prevVal=val;
                        PORTB=_BV(PB0);// 1 out to PB0 (latch the result)
                        if (DelayExponent>0)
                                        Delay(DelayExponent);
                        //read the result
                        val=PIND;
          }
          //wait until change from <triglevel to >triglevel. Or break from SPI.
          while ((SPDR!=1) & (val<TrigLevel || prevVal>TrigLevel));
        break;
        case 1://falling
        val=0;
          do
          {
                        //start conversion
                        PORTB=0;
                        prevVal=val;
                        PORTB=_BV(PB0);// 1 out to PB0 (latch the result)
                        if (DelayExponent>0)
```

```
                                        Delay(DelayExponent);
                        //read the result
                        val=PIND;
          }
          //wait until change from <triglevel to >triglevel.
          while (val>TrigLevel || prevVal<TrigLevel);
        break;
        }

        //START_CONVERSION();
        for (int16_t i=0;i<DATA_SIZE;i++)
        {
          //start conversion
                        PORTB=0;

                        if (DelayExponent>0)
                                Delay(DelayExponent);

                        PORTB=_BV(PB0);// 1 out to PB0 (latch the result)
                        //read the result

                        data[i]=PIND;
        }
}
```

5.1.3. SPIDriver.c

```
#define Buffsize 10
//#define __AVR_ATmega8__
#include <inttypes.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>
#include "../osc_messages.h"

#if defined (__AVR_ATmega8__)
#define DDR_SPI DDRB
#define DD_MISO DDB4
#elif defined (__AVR_ATmega16__)
#define DDR_SPI DDRB
#define DD_MISO DDB6
#endif

volatile int8_t InQPos=0;
volatile int8_t OutQPos=0;
volatile int8_t InQueue[Buffsize];
volatile int8_t OutQueue[Buffsize];
```

```
int8_t SPI_GetReadableBytes(void)
{
        return InQPos;
}

int8_t SPI_Read(void)
{
        while (InQPos==0);
          return InQueue[--InQPos];
}

void SPI_Write(int8_t byte)
{
        /*if (OutQPos<Buffsize)
        {
          OutQueue[OutQPos++]=byte;
          return OSC_ACK;
        }
        else
        */
        while (OutQPos>Buffsize-1);
          OutQueue[OutQPos++]=byte;
}

SIGNAL(SIG_SPI)
{
        if (InQPos>=Buffsize)
        {
          SPDR=OSC_BUFFER_FULL;
          return;
        }
        InQueue[InQPos++]=SPDR;
        if (OutQPos<1)
          SPDR=OSC_BUFFER_EMPTY;
        else
          SPDR=OutQueue[--OutQPos];
}

void SPI_Init(void)
{
//init SPI
        DDR_SPI=_BV(DD_MISO); //Set data direction to out
        SPCR=_BV(SPE)|_BV(SPIE);//enable SPI
        //enable interrupts..
        SREG=_BV(SREG_I);
}
```