```
/*****************************************************
This program was created by the CodeWizardAVR V3.31
Automatic Program Generator
© Copyright 1998-2017 Pavel Haiduc, HP InfoTech s.r.l.
http://www.hpinfotech.com

Project : Simulator NexStar Basic
Version : 1.0
Date    : 10/1/2022
Author  : Tristian Botez
Company : Home
Comments:
simulator Basic de controler
NexStar cu ATmega 328P


Chip type               : ATmega328P
Program type            : Application
AVR Core Clock frequency: 4.000000 MHz
Memory model            : Small
External RAM size       : 0
Data Stack size         : 512
*****************************************************/

#include <io.h>
#include <stdio.h>

// Declare your global variables here
bit cancel=0;

flash char locatie [10]={0x2E ,0x2E, 0x11, 0x30, 0x17,0x23,0x2D,0x30,0x23,0};

unsigned char viteza = 0,  viteza_dec =0, ;
unsigned long int pasigoto=0, pasi=0, pasigoto_dec=0,  pasi_dec=0;

#define DATA_REGISTER_EMPTY (1<<UDRE0)
#define RX_COMPLETE (1<<RXC0)
#define FRAMING_ERROR (1<<FE0)
#define PARITY_ERROR (1<<UPE0)
#define DATA_OVERRUN (1<<DOR0)

// USART Receiver buffer
#define RX_BUFFER_SIZE0 64
char rx_buffer0[RX_BUFFER_SIZE0];

#if RX_BUFFER_SIZE0 <= 256
volatile unsigned char rx_wr_index0=0,rx_rd_index0=0,rd_index=0;
#else
volatile unsigned int rx_wr_index0=0,rx_rd_index0=0;
#endif

#if RX_BUFFER_SIZE0 < 256
volatile unsigned char rx_counter0=0;
```

```
#else
volatile unsigned int rx_counter0=0;
#endif

// This flag is set on USART Receiver buffer overflow
bit rx_buffer_overflow0;

// USART Receiver interrupt service routine
interrupt [USART_RXC] void usart_rx_isr(void)
{
unsigned char status;
char data;
status=UCSR0A;
data=UDR0;
if ((status & (FRAMING_ERROR | PARITY_ERROR | DATA_OVERRUN))==0)
   {
   rx_buffer0[rx_wr_index0++]=data;
#if RX_BUFFER_SIZE0 == 256
   // special case for receiver buffer size=256
   if (++rx_counter0 == 0) rx_buffer_overflow0=1;
#else
   if (rx_wr_index0 == RX_BUFFER_SIZE0) rx_wr_index0=0;
   if (++rx_counter0 == RX_BUFFER_SIZE0)
      {
      rx_counter0=0;
      rx_buffer_overflow0=1;
      }
#endif
   }
}

// Get a character from the USART Receiver buffer
#define _ALTERNATE_GETCHAR_
#pragma used+
char getchar(void)
{
char data;
while (rx_counter0==0);
data=rx_buffer0[rx_rd_index0++];
#if RX_BUFFER_SIZE0 != 256
if (rx_rd_index0 == RX_BUFFER_SIZE0) rx_rd_index0=0;
#endif
#asm("cli")
--rx_counter0;
#asm("sei")
return data;
}
#pragma used-



void main(void)
{
// Declare your local variables here
```

```
bit   dest_hex=0;
char x=0;
unsigned  int k=0;
char model [8]={0,0,0,0,0,0,0,0};

char Crt_hex [19]={0,0,0,0,0,0,0,0,',',0,0,0,0,0,0,0,0,'#',0};

char R_dest_hex [8]={0,0,0,0,0,0,0,0};
char D_dest_hex [8]={0,0,0,0,0,0,0,0};

char R_crt_hex [8]={0,0,0,0,0,0,0,0};
char D_crt_hex [8]={0,0,0,0,0,0,0,0};


// Crystal Oscillator division factor: 4
#pragma optsize-
CLKPR=(1<<CLKPCE);
CLKPR=(0<<CLKPCE) | (0<<CLKPS3) | (0<<CLKPS2) | (1<<CLKPS1) | (0<<CLKPS0);
#ifdef _OPTIMIZE_SIZE_
#pragma optsize+
#endif


// Input/Output Ports initialization
// Port B initialization
// Function: Bit7=Out Bit6=Out Bit5=Out Bit4=Out Bit3=Out Bit2=Out Bit1=Out Bit0=Out
DDRB=(1<<DDB7) | (1<<DDB6) | (1<<DDB5) | (1<<DDB4) | (1<<DDB3) | (1<<DDB2) | (1<<DDB1) | (1<<DDB0);
// State: Bit7=0 Bit6=0 Bit5=0 Bit4=0 Bit3=0 Bit2=0 Bit1=0 Bit0=0
PORTB=(0<<PORTB7) | (0<<PORTB6) | (0<<PORTB5) | (0<<PORTB4) | (0<<PORTB3) | (0<<PORTB2) |
(0<<PORTB1) | (0<<PORTB0);

// Port C initialization
// Function: Bit6=In Bit5=Out Bit4=Out Bit3=Out Bit2=Out Bit1=Out Bit0=Out
DDRC=(0<<DDC6) | (1<<DDC5) | (1<<DDC4) | (1<<DDC3) | (1<<DDC2) | (1<<DDC1) | (1<<DDC0);
// State: Bit6=T Bit5=0 Bit4=0 Bit3=0 Bit2=0 Bit1=0 Bit0=0
PORTC=(0<<PORTC6) | (0<<PORTC5) | (0<<PORTC4) | (0<<PORTC3) | (0<<PORTC2) | (0<<PORTC1) |
(0<<PORTC0);

// Port D initialization
// Function: Bit7=Out Bit6=Out Bit5=Out Bit4=Out Bit3=Out Bit2=In Bit1=Out Bit0=In
DDRD=(1<<DDD7) | (1<<DDD6) | (1<<DDD5) | (1<<DDD4) | (1<<DDD3) | (0<<DDD2) | (1<<DDD1) | (0<<DDD0);
// State: Bit7=0 Bit6=0 Bit5=0 Bit4=0 Bit3=0 Bit2=P Bit1=0 Bit0=T
PORTD=(0<<PORTD7) | (0<<PORTD6) | (0<<PORTD5) | (0<<PORTD4) | (0<<PORTD3) | (1<<PORTD2) |
(0<<PORTD1) | (0<<PORTD0);


// Timer/Counter 0 initialization
// Clock source: System Clock
// Clock value: 15.625 kHz
// Mode: Normal top=0xFF
// OC0A output: Disconnected
// OC0B output: Disconnected
// Timer Period: 16 ms
TCCR0A=(0<<COM0A1) | (0<<COM0A0) | (0<<COM0B1) | (0<<COM0B0) | (0<<WGM01) | (0<<WGM00);
```

```
TCCR0B=(0<<WGM02) | (1<<CS02) | (0<<CS01) | (0<<CS00);
TCNT0=0x06;
OCR0A=0x00;
OCR0B=0x00;


// Timer/Counter 1 initialization
// Clock source: System Clock
// Clock value: 62.500 kHz
// Mode: Normal top=0xFFFF
// OC1A output: Disconnected
// OC1B output: Disconnected
// Noise Canceler: Off
// Input Capture on Falling Edge
// Timer Period: 1.0256 s
// Timer1 Overflow Interrupt: On
// Input Capture Interrupt: Off
// Compare A Match Interrupt: Off
// Compare B Match Interrupt: Off
TCCR1A=(0<<COM1A1) | (0<<COM1A0) | (0<<COM1B1) | (0<<COM1B0) | (0<<WGM11) | (0<<WGM10);
TCCR1B=(0<<ICNC1) | (0<<ICES1) | (0<<WGM13) | (0<<WGM12) | (0<<CS12) | (1<<CS11) | (1<<CS10);
TCNT1H=0x05;
TCNT1L=0x9C;
ICR1H=0x00;
ICR1L=0x00;
OCR1AH=0x00;
OCR1AL=0x00;
OCR1BH=0x00;
OCR1BL=0x00;

// Timer/Counter 2 initialization
// Clock source: System Clock
// Clock value: Timer2 Stopped
// Mode: Normal top=0xFF
// OC2A output: Disconnected
// OC2B output: Disconnected
ASSR=(0<<EXCLK) | (0<<AS2);
TCCR2A=(0<<COM2A1) | (0<<COM2A0) | (0<<COM2B1) | (0<<COM2B0) | (0<<WGM21) | (0<<WGM20);
TCCR2B=(0<<WGM22) | (0<<CS22) | (0<<CS21) | (0<<CS20);
TCNT2=0x00;
OCR2A=0x00;
OCR2B=0x00;

// Timer/Counter 0 Interrupt(s) initialization
TIMSK0=(0<<OCIE0B) | (0<<OCIE0A) | (1<<TOIE0);

// Timer/Counter 1 Interrupt(s) initialization
TIMSK1=(0<<ICIE1) | (0<<OCIE1B) | (0<<OCIE1A) | (1<<TOIE1);

// Timer/Counter 2 Interrupt(s) initialization
TIMSK2=(0<<OCIE2B) | (0<<OCIE2A) | (0<<TOIE2);

// External Interrupt(s) initialization
// INT0: Off
```

```
// INT1: Off
// Interrupt on any change on pins PCINT0-7: Off
// Interrupt on any change on pins PCINT8-14: Off
// Interrupt on any change on pins PCINT16-23: Off
EICRA=(0<<ISC11) | (0<<ISC10) | (0<<ISC01) | (0<<ISC00);
EIMSK=(0<<INT1) | (0<<INT0);
PCICR=(0<<PCIE2) | (0<<PCIE1) | (0<<PCIE0);

// USART initialization
// Communication Parameters: 8 Data, 1 Stop, No Parity
// USART Receiver: On
// USART Transmitter: On
// USART Mode: Asynchronous
// USART Baud Rate: 9600
UCSR0A=(0<<RXC0) | (0<<TXC0) | (0<<UDRE0) | (0<<FE0) | (0<<DOR0) | (0<<UPE0) | (0<<U2X0) |
(0<<MPCM0);
UCSR0B=(1<<RXCIE0) | (0<<TXCIE0) | (0<<UDRIE0) | (1<<RXEN0) | (1<<TXEN0) | (0<<UCSZ02) | (0<<RXB80)
| (0<<TXB80);
UCSR0C=(0<<UMSEL01) | (0<<UMSEL00) | (0<<UPM01) | (0<<UPM00) | (0<<USBS0) | (1<<UCSZ01) |
(1<<UCSZ00) | (0<<UCPOL0);
UBRR0H=0x00;
UBRR0L=0x19;

// Analog Comparator initialization
// Analog Comparator: Off
// The Analog Comparator's positive input is
// connected to the AIN0 pin
// The Analog Comparator's negative input is
// connected to the AIN1 pin
ACSR=(1<<ACD) | (0<<ACBG) | (0<<ACO) | (0<<ACI) | (0<<ACIE) | (0<<ACIC) | (0<<ACIS1) | (0<<ACIS0);
ADCSRB=(0<<ACME);
// Digital input buffer on AIN0: On
// Digital input buffer on AIN1: On
DIDR1=(0<<AIN0D) | (0<<AIN1D);

// ADC initialization
// ADC disabled
ADCSRA=(0<<ADEN) | (0<<ADSC) | (0<<ADATE) | (0<<ADIF) | (0<<ADIE) | (0<<ADPS2) | (0<<ADPS1) |
(0<<ADPS0);

// SPI initialization
// SPI disabled
SPCR=(0<<SPIE) | (0<<SPE) | (0<<DORD) | (0<<MSTR) | (0<<CPOL) | (0<<CPHA) | (0<<SPR1) | (0<<SPR0);

// TWI initialization
// TWI disabled
TWCR=(0<<TWEA) | (0<<TWSTA) | (0<<TWSTO) | (0<<TWEN) | (0<<TWIE);


// Globally enable interrupts
#asm("sei")
```

```
while (1)
    if (dest_hex !=0)
      {   // tratam RA
          #asm("cli")
           pasigoto = 500; // ca si exemplu e data o constanta in realitate ea se calculeza functie de comanda motorului si raportul monturii
           pasi=0;
           viteza = 10;
           #asm("sei")
           for (k=0;k<8;k++)                //  destinatia devine pozitie curenta
            { R_crt_hex [k] = R_dest_hex [k];
              Crt_hex [k] = R_dest_hex [k];
            }

           while (pasi != pasigoto)
           if (rx_counter0 !=0 )       // vedem daca avem comenzi sosite pe seriala
               switch (rx_buffer0[rd_index])
                     {
                      case 'M' :        // cancel goto
                         printf ( "#");
                         cancel=1;
                         rx_counter0 = rx_counter0 -1;    // am citit un caracter
                         rd_index ++ ;
                         if (rd_index == RX_BUFFER_SIZE0)
                         rd_index = 0 ;

                       break;

                       case 'L' :        // goto in progres ?
                         printf ( "1#");
                         rx_counter0 = rx_counter0 -1;    // am citit un caracter
                         rd_index ++ ;
                         if (rd_index == RX_BUFFER_SIZE0)
                         rd_index = 0 ;
                       break;


                       default:             // daca nu recunoastem inceput de comanda resetam
                         rx_counter0=0;
                         rd_index =0  ;
                         rx_wr_index0 = 0;
                      }


                // tratam DEC


                  #asm("cli")
                    pasigoto_dec = 500;// constanta ca si exemplu in realitate ea se calculeaza dupa comanda motorului si raportul monturii
                    pasi_dec=0;
                    viteza_dec=1;
                    #asm("sei")
                    for (k=0;k<8;k++)
```

```
                    {D_crt_hex [k] = D_dest_hex [k];  //  destinatia devine pozitie curenta
                     Crt_hex [k+9] = D_dest_hex [k];
                    }


          while (  pasi_dec != pasigoto_dec)
            { if (rx_counter0 !=0 )       // vedem daca avem comenzi sosite pe seriala
                switch (rx_buffer0[rd_index])
                    {
                     case 'M' :        // cancel goto
                      printf ( "#");
                      cancel=1;
                      rx_counter0 = rx_counter0 -1;    // am citit un caracter
                      rd_index ++ ;
                      if (rd_index == RX_BUFFER_SIZE0)
                      rd_index = 0 ;

                      break;

                      case 'L' :        // goto in progres ?
                      printf ( "1#");
                      rx_counter0 = rx_counter0 -1;    // am citit un caracter
                      rd_index ++ ;
                      if (rd_index == RX_BUFFER_SIZE0)
                      rd_index = 0 ;
                       break;


                       default:              // daca nu recunoastem inceput de comanda resetam
                      rx_counter0=0;
                      rd_index =0  ;
                      rx_wr_index0 = 0;

                      }

                }
           dest_hex =0;
           putchar ('#');

      }

    else
       {if (rx_counter0 !=0 )       // vedem daca avem comenzi sosite pe seriala    ; rx_counter0 e actualizat de rutina
USART
          {  switch (rx_buffer0[rd_index])
             {
              case 'e' :  //  get precise RA/DEC
                for (x=0;x<18;x++)
                 {  putchar (Crt_hex [x]);
                   for (k=0; k<400; k++);
                 }
                    rx_counter0 = rx_counter0 -1;    // am citit un caracter
                    rd_index++;
                    if (rd_index == RX_BUFFER_SIZE0)
```

```
    rd_index = 0 ;


 break;

case 'K' :        // cmda echo
   if (rx_counter0 > 1)   // avem minim doua caractere ?
      {
         if (rd_index == (RX_BUFFER_SIZE0-1))    // aflam caracterul 2
         { x = rx_buffer0[0];
          rd_index=1;
         }
         else
        { x = rx_buffer0[rd_index +1];
          rd_index =  rd_index + 2;
         }
        printf ( "%c%c", x,'#' );
        rx_counter0 = rx_counter0 -2;
      }

break;

case 't' :        // get traking mode
   printf ( "2#");
   rx_counter0 = rx_counter0 -1;    // am citit un caracter
   rd_index ++ ;
 if (rd_index == RX_BUFFER_SIZE0)
   rd_index = 0 ;

break;

case 'L' :        // goto in progres ?
   printf ( "0#");
   rx_counter0 = rx_counter0 -1;    // am citit un caracter
   rd_index ++ ;
 if (rd_index == RX_BUFFER_SIZE0)
    rd_index = 0 ;

 break;

case 'r' :        // goto precis
   if (rx_counter0 > 17 )   // avem minim 18 caractere ?
    {  for (k=0 ;k < 8; k++)
          {   rd_index ++ ;
             if (rd_index == RX_BUFFER_SIZE0)
             rd_index = 0 ;
             R_dest_hex [k] = rx_buffer0[rd_index];
          }

          rd_index ++ ;                    // virgula
           if (rd_index == RX_BUFFER_SIZE0)
           rd_index = 0 ;
```

```
      for (k=0 ;k < 8; k++)
       {  rd_index ++ ;
          if (rd_index == RX_BUFFER_SIZE0)
          rd_index = 0 ;
          D_dest_hex [k] = rx_buffer0[rd_index];
        }

          rd_index ++ ;
          if (rd_index == RX_BUFFER_SIZE0)
          rd_index = 0 ;
       rx_counter0 = rx_counter0 -18;


      dest_hex=1;// semnalizam ca avem destinatii noi in hexa


      }

break;


case 's' :        // sync precis
   if (rx_counter0 > 17 )   // avem minim 18 caractere ?
    {  for (k=0 ;k < 8; k++)
          {   rd_index ++ ;
             if (rd_index == RX_BUFFER_SIZE0)
             rd_index = 0 ;
             R_crt_hex [k] = rx_buffer0[rd_index];
             Crt_hex [k] = rx_buffer0[rd_index];
          }

          rd_index ++ ;                  // virgula
           if (rd_index == RX_BUFFER_SIZE0)
           rd_index = 0 ;


      for (k=0 ;k < 8; k++)
       {  rd_index ++ ;
          if (rd_index == RX_BUFFER_SIZE0)
          rd_index = 0 ;
          D_crt_hex [k] = rx_buffer0[rd_index];
          Crt_hex [k+9] = rx_buffer0[rd_index];
        }

          rd_index ++ ;
          if (rd_index == RX_BUFFER_SIZE0)
          rd_index = 0 ;
       rx_counter0 = rx_counter0 -18;
      }

      printf ( "#");

break;
```

```
case 'J' :       // aligned  ?

          printf ( "1#");
          rx_counter0 = rx_counter0 -1;    // am citit un caracter
          rd_index ++ ;
          if (rd_index == RX_BUFFER_SIZE0)
           rd_index = 0 ;

break;

case 'V' :       // version ?
          printf ( "16#");
          rx_counter0 = rx_counter0 -1;    // am citit un caracter
          rd_index ++ ;
          if (rd_index == RX_BUFFER_SIZE0)
          rd_index = 0 ;

break;

case 'm' :       // model  ?

          printf ( "10#");  //   model GT
          rx_counter0 = rx_counter0 -1;    // am citit un caracter
          rd_index ++ ;
          if (rd_index == RX_BUFFER_SIZE0)
          rd_index = 0 ;

break;

case 'M' :       // cancel goto
          printf ( "#");
          // cancel=1;
          rx_counter0 = rx_counter0 -1;    // am citit un caracter
          rd_index ++ ;
          if (rd_index == RX_BUFFER_SIZE0)
          rd_index = 0 ;

break;


case 'P' :       //     model extins
   if (rx_counter0 > 7 )   // avem minim 8 caractere ?
    {  for (k=0 ;k < 7; k++)
          {  rd_index ++ ;
           if (rd_index == RX_BUFFER_SIZE0)
           rd_index = 0 ;
           model [k] = rx_buffer0[rd_index];
          }

          rd_index ++ ;
          if (rd_index == RX_BUFFER_SIZE0)
          rd_index = 0 ;
```

```
                rx_counter0 = rx_counter0 -8;

               putchar (model [2]);
               printf ( "#");

             }

    break;

    case 'w' :        // locatia  ?

       printf (locatie, "%s");  //   cluj
       rx_counter0 = rx_counter0 -1;    // am citit un caracter
       rd_index ++ ;
       if (rd_index == RX_BUFFER_SIZE0)
       rd_index = 0 ;

    break;

    default:              // daca nu recunoastem inceput de comanda resetam
           rx_counter0=0;
           rd_index =0  ;
           rx_wr_index0 = 0;

  }


   }









 }

}
```