```
/*
 * MAC_sla.c
 *
 * Created: 29APR2018
 *  Author: David K. Watson
 *
 *
 *
    Copyright (C) 2018 David K. Watson

    MAC_sla is free software: you can redistribute it and/or modify
    it under the terms of the GNU Lesser General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    MAC_sla is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU Lesser General Public License for more details.

    You should have received a copy of the GNU Lesser General Public License
    along with MAC_sla.  If not, see <http://www.gnu.org/licenses/>.
*/
// compiles to 2654 with avr-gcc 7.3
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5====== Includes and directives
// Includes and directives
//

//#include  <stdio.h>
//#include  <stdlib.h>
#include    <stdint.h>
#include    <string.h>
#include    <avr/io.h>
#include    <avr/interrupt.h>
#include    <avr/sleep.h>
#include    <util/delay.h>

#ifndef     byte
    #define byte    uint8_t
#endif

#if defined (__AVR_ATmega328P__) || defined (__AVR_ATmega328__)
    #define m328p
    #define LED_PORT        PORTB
    #define LED_PIN         PB5
    char BOARD[]            = {"m328p"};
#elif defined (__AVR_ATmega2560__)
    #define m2560
    #define USART_RX_vect   USART0_RX_vect
    #define USART_UDRE_vect USART0_UDRE_vect
    #define LED_PORT        PORTB
    #define LED_PIN         PB7
    char BOARD[]            = {"m2560"};
#else
    #warning "device type not defined"
#endif

#ifndef F_CPU
    #define F_CPU    16000000UL
#endif
```

```
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5======== Constants and defines
// Constants and defines
//

const char      VERSION[]   = {"MAC_sla"};
const byte      DEVICE_ID   = 28;
const uint32_t  PULSE_DELAY = 300000;   // delay in us per pulse for 1 RPM

#define     STX         2
#define     ETX         3
#define     ACK         6
#define     TAB         9
#define     LF          10
#define     CR          13
#define     NAK         21
#define     ESC         27
#define     SPACE       32
#define     AST         42
#define     COMMA       44
#define     DOT         46
#define     DIGIT       48 ... 57
#define     COLON       58
#define     SEMI_COLON  59
#define     PROMPT      62
#define     PRINTABLE   32 ... 126

#define     BUFLGTH     64
#define     BIT_RATE    38400
#define     BYTE_RATE   (11000000/BIT_RATE)

#define     SLEEP_MODE_IDLE         (0x00<<1)
#define     SLEEP_MODE_ADC          (0x01<<1)
#define     SLEEP_MODE_PWR_DOWN     (0x02<<1)
#define     SLEEP_MODE_PWR_SAVE     (0x03<<1)
#define     SLEEP_MODE_STANDBY      (0x06<<1)
#define     SLEEP_MODE_EXT_STANDBY  (0x07<<1)

#define     CLOCKWISE   0
#define     PULSE       10

enum dir {cw = CLOCKWISE, ccw = !cw} __attribute__ ((__packed__)) dir;
enum mode {CRUISE, STEP} __attribute__ ((__packed__)) mode;
enum ustep {ONE, TWO, FOUR, EIGHT, SIXTEEN}  __attribute__ ((__packed__)) ustep;
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5========= typedefs and structs
// typedefs and structs
//
typedef char * string;
typedef byte * bstring;
typedef struct
{
    byte data[BUFLGTH];
    byte index;
} stack;

typedef struct
{
    byte data[BUFLGTH];
    byte next;
    byte first;
} FIFO;
```

```
typedef struct
{
    byte    bit_0: 1;
    byte    bit_1: 1;
    byte    bit_2: 1;
    byte    bit_3: 1;
    byte    bit_4: 1;
    byte    bit_5: 1;
    byte    bit_6: 1;
    byte    bit_7: 1;
} bits;

typedef struct
{
    uint16_t    old;
    uint16_t    current;
    uint16_t    rover;
    int         counter;
    int         velocity;
    enum dir    dir;
} bezl;

bezl bezel = {.old = 0, .current = 0, .rover = 100, .counter = 0, .velocity = 0,
 .dir = cw};
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6              Macros
// Macros
//

#define nop()       __asm__ __volatile__ ("nop \n")
#define wdr()       __asm__ __volatile__ ("wdr \n")
#define sbi(sfr, bit) (_SFR_BYTE(sfr) |= _BV(bit))
#define cbi(sfr, bit) (_SFR_BYTE(sfr) &= ~_BV(bit))

#define Set(x)      (x = 1)
#define Clear(x)    (x = 0)
#define getBit(x,y) ((x >> y) & 1)

#define tab()       prtC(TAB)
#define cr()        prtC(CR)
#define lf()        prtC(LF)
#define space()     prtC(SPACE)
#define comma()     prtC(COMMA)
#define colon()     prtC(COLON)
#define prompt()    prtC(62)
#define crlf()      cr();lf()
#define sprint(x)   prtS((char*)(x))
#define sprintln(x) (prtS((char*)(x)));lf()

#define bitRead(value, bit)     (((value) >> (bit)) & 0x01)
#define bitSet(value, bit)      ((value) |= (1UL << (bit)))
#define bitClear(value, bit)    ((value) &= ~(1UL << (bit)))
#define bitWrite(value, bit, b) (b? bitSet(value, bit): \
                bitClear(value, bit))

#define clockCyclesPerMicrosecond()   (F_CPU / 1000000L)
#define microsecondsToClockCycles(a)  ((a) * clockCyclesPerMicrosecond())
#define clockCyclesToMicroseconds(a)  ((a) / clockCyclesPerMicrosecond())

#define MICROSECONDS_PER_TIMER_OVERFLOW    (clockCyclesToMicroseconds(64 * 256))
#define MILLIS_INC                      (MICROSECONDS_PER_TIMER_OVERFLOW / 1000)
#define FRACT_INC                ((MICROSECONDS_PER_TIMER_OVERFLOW % 1000) >> 3)
#define FRACT_MAX                           (1000 >> 3)
```

```
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6 Function prototypes
// Function prototypes
//
void        initTimer0();
void        timer0on();
void        timer0off();

void        enableU0();
void        disableU0();
void        clearBuffer(uint8_t*);
void        binC(byte);
void        push(byte);
byte        pop();
void        flash_led();

byte        getBufLen(FIFO*);
byte        isFull(FIFO*);
byte        isEmpty(FIFO*);
void        clearBuf(FIFO*);
byte        peekBuf(FIFO*, byte);
byte        getCh(FIFO*);
void        addCh(FIFO*, byte);

void        usartInit(uint32_t);

ISR         (PORT_INT_VECT);
void        setupPCI();
void        reset_encoder();
void        sendPulse();
void        setup_motor_controls();

ISR         (USART_RX_vect);
ISR         (USART_UDRE_vect);

int         main ();

void        process_bezel_change();
void        process_msg();
void        process_message(char*);

void        prtC(char);
void        prtS(char*);
void        prtB(byte);
void        prtI(int);
void        prtL(long);
void        prtD(double);

void        lasc(uint32_t, char*);
void        iasc(int, char*);
void        basc(byte, char*);

uint32_t    get_long(char*, byte*);

void        wait(uint32_t us);
uint32_t    millis();
uint32_t    micros();
void        reset_timer0();
ISR         (TIMER0_OVF_vect);
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5======== variable declarations
// variable declarations
```

```
FIFO uartRx;
FIFO uartTx;
stack stk;

char msg_buffer[BUFLGTH];

bits flag1;
#define message_waiting     flag1.bit_0
#define ISR_T1_OCAflag      flag1.bit_1
#define ISR_T1_OCBflag      flag1.bit_2
#define bezel_has_changed   flag1.bit_3
#define manual_off          flag1.bit_4
//#define flag1.bit_5
//#define flag1.bit_6
//#define flag1.bit_7

static byte             timer0_fract            = 0;
volatile unsigned long  timer0_overflow_count   = 0;
volatile unsigned long  timer0_millis           = 0;

byte     msg_length     = 0;
byte     full_buffer    = (BUFLGTH - 1);
byte     buff_safety    = (BUFLGTH - 5);

uint16_t recv_count     = 0;
uint16_t print_count    = 0;

uint32_t old_micros     = 0;
uint32_t cycle          = 0;
uint32_t pulse_counter  = 0;
uint32_t dir_counter[2] = {};

//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6    inline functions
// inline functions
//
inline void     initTimer0()    {TCCR0B = 0B00000011;}
inline void     timer0on()      {TIMSK0 = 1;}
inline void     timer0off()     {TIMSK0 = 0;}

inline void     enableU0()      {UCSR0B |= (1<<UDRIE0);}
inline void     disableU0()     {UCSR0B &= ~(1<<UDRIE0);}
inline void     clearBuffer(uint8_t *s) {for(byte i = 0; i < BUFLGTH; i++)
                                    s[i] = 0;}
inline void     binC(byte c)    {for(byte i = 0; i < 8; i++)
                                    prtC(getBit(c, (7 - i)) + 48);}
inline void     push(byte a)    {if(stk.index != BUFLGTH)
                                    (stk.data[stk.index++] = a);}
inline byte     pop()           {return (stk.index == 0)?0:
                                    (stk.data[--stk.index]);}
inline void     flash_led()     {LED_PORT |= (1 << LED_PIN); _delay_ms(30);
                                    LED_PORT &= ~(1 << LED_PIN);}

//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6                FIFO
// FIFO
//
inline byte     getBufLen(FIFO *buffer)
                    {return (buffer->next >= buffer->first)?
                    (buffer->next - buffer->first):
                    ((BUFLGTH - buffer->first) + buffer->next);}
inline byte     isFull(FIFO *buffer)
```

```
                    {return (getBufLen(buffer) == BUFLGTH)?1:0;}
inline byte     isEmpty(FIFO *buffer)
                    {return (buffer->next == buffer->first)?1:0;}
inline void     clearBuf(FIFO *buffer)
                    {buffer->next = buffer->first = 0;}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6        FIFO peekBuf
//
byte peekBuf(FIFO *buffer, byte index)
{
    index = ((buffer->first + index) % BUFLGTH);
    return buffer->data[index];
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6          FIFO getCh
//
byte getCh(FIFO *buffer)
{
    byte byte = buffer->data[buffer->first];
    buffer->first = (buffer->first + 1) % BUFLGTH;
    return byte;
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6          FIFO addCh
//
void addCh(FIFO *buffer, byte ch)
{
    byte _temp = (buffer->next + 1) % BUFLGTH;
    buffer->data[buffer->next] = ch;
    buffer->next = _temp;
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6           usartInit
//
void usartInit(uint32_t brate)
{
    UCSR0B = (1 << RXCIE0 ) | (1 << RXEN0 ) | (1 << TXEN0 );
    UCSR0C = (1 << UCSZ00 ) | (1 << UCSZ01 );
    UBRR0 = ((F_CPU / 16) / brate) - 1;
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6         motor stuff
//
#define     PORT_CLK    PORTD
#define     PIN_CLK     PD7
#define     DDR_CLK     DDRD

#define     PORT_DIR    PORTD
#define     PIN_DIR     PD6
#define     DDR_DIR     DDRD

#define     PORT_MODE   PORTB
#define     DDR_MODE    DDRB
#define     PIN_MODE    PB4
#define     INP_MODE    PINB

//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6       encoder stuff
//
// The following establish which port the encoder pins are attached to
// It is important that both encoder pins are on the same port,
// so that only a single port interrupt vector is used
// un-comment the port in use
```

```
#define PORT_B
//#define PORT_C
//#define PORT_D
#ifdef PORT_B
    #define     PORT_INT_VECT   PCINT0_vect // PORTB
    #define     PORT_EN PINB    // port tag to read pin state
    #define     DDR     DDRB    // the data direction port - set in main
    #define     PORT    PORTB   // the port data address - set in main
    #define     PCI     PCIE0   // port interrupt enabler
    #define     MASK    PCMSK0  // port interrupt mask
#endif
#ifdef PORT_C
    #define     PORT_INT_VECT   PCINT1_vect // PORTB
    #define     PORT_EN PINC    // port tag to read pin state
    #define     DDR     DDRC    // the data direction port - set in main
    #define     PORT    PORTC   // the port data address - set in main
    #define     PCI     PCIE1   // port interrupt enabler
    #define     MASK    PCMSK1  // port interrupt mask
#endif
#ifdef PORT_D
    #define     PORT_INT_VECT   PCINT2_vect // PORTB
    #define     PORT_EN PIND    // port tag to read pin state
    #define     DDR     DDRD    // the data direction port - set in main
    #define     PORT    PORTD   // the port data address - set in main
    #define     PCI     PCIE2   // port interrupt enabler
    #define     MASK    PCMSK3  // port interrupt mask
#endif

// All settings relative to the encoder are in the next 7/9 statements
// check pin DDR and pullups in main()
// change next 4 for pins - currently set A8 B9
#define     PIN_A   PB0     // pin tag for pinA
#define     PIN_B   PB1     // pin tag for pinB
#define     INTA    PCINT0  // pinA interrupt enable
#define     INTB    PCINT1  // pinB interrupt enable
// all encoder related statements from here thru the ISR and setupPCI
// need not be touched as the all feed from references defined above

#define     MASKA   (1 << PIN_A)
#define     MASKB   (1 << PIN_B)

byte        detente     = MASKA | MASKB;

uint16_t    a_first     = ((MASKA << 8) + MASKB);
uint16_t    b_first     = ((MASKB << 8) + MASKA);

#define     enable_pci()    bitSet(PCICR, PCI)
#define     disable_pci()   bitClear(PCICR, PCI)
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=======  ENCODER PORT_INT_VECT
//
ISR(PORT_INT_VECT)
{
    sei();
    disable_pci();

    static uint16_t code = 0;
    byte _state = PORT_EN & detente;
    bezel.old = bezel.current;

    if (_state == detente)
    {
        if(code == a_first)
        {
             bezel.current = (((bezel.current + bezel.rover) - 1) % bezel.rover);
             bezel.dir = ccw;
             bezel.counter--;
        }

        if(code == b_first)
        {
            bezel.current = ((bezel.current + 1) % bezel.rover);
            bezel.dir = cw;
            bezel.counter++;
        }

        if(mode == CRUISE)
        {
            if (bezel.counter < 0)
            {
                bezel.velocity = -bezel.counter;
                bezel.dir = ccw;
            }
            else
            {
                bezel.velocity = bezel.counter;
                bezel.dir = cw;
            }
        }

        if(bezel.current != bezel.old) Set(bezel_has_changed);
        code = 0;
    }
    else code = ((code << 4) + _state);

    enable_pci();
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6            setupPCI
//
void setupPCI()
{
    bitSet(MASK, INTA);                 // enable interrupt on pinA
    bitSet(MASK, INTB);                 // enable interrupt on pinB
    enable_pci();                       // enable PCI
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6       reset_encoder
//
void reset_encoder()
{
    bezel.old       = 0;
    bezel.current   = 0;
    bezel.counter   = 0;
    bezel.velocity  = 0;
    if(mode == CRUISE) timer0off();;
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6          send_pulse
//
void send_pulse()
{
    bitWrite(PORT_DIR, PIN_DIR, bezel.dir);
    sbi(PORT_CLK, PIN_CLK);
    if(mode == CRUISE) reset_timer0();
    _delay_us(PULSE);
    cbi(PORT_CLK, PIN_CLK);
    pulse_counter++;
    dir_counter[dir]++;
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5========  setup_motor_controls
//
void setup_motor_controls()
{
    bitSet(DDR_CLK, PIN_CLK);       // set clock pin as output
    bitSet(DDR_DIR, PIN_DIR);       // set direction pin as output
    bitClear(PORT_CLK, PIN_CLK);    // initialize clock at 0
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6       USART_RX_vect
//
ISR(USART_RX_vect)
{
    byte ch = UDR0;
    if (ch == 13) return;
    if (ch == 10)
    {
        msg_length = 0;
        Set(message_waiting);
    }
    else
    {
        if(msg_length != full_buffer)
        {
            addCh(&uartRx, ch);
            msg_length++;
            recv_count++;
        }
    }
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6     USART_UDRE_vect
//
ISR(USART_UDRE_vect)
{
    if (isEmpty(&uartTx)) disableU0();
    else
    {
        UDR0 = getCh(&uartTx);
        print_count++;
    }
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6                main
//
int main(void)
{
    DDR   = 0B00100000;    // make sure pins are input
    PORT  = 0B11011111;    // ensure pullups
    usartInit(BIT_RATE);
    sei ();
    set_sleep_mode(SLEEP_MODE_IDLE);
    initTimer0();
    timer0on();
    process_message((char*)"-v");
    setupPCI();             // activates encoder interrupts

    ustep = TWO;
    //mode = STEP;
    mode = CRUISE;

    while (1)
    {
        if((micros() > cycle) && (mode == CRUISE)) send_pulse();
        if(bezel_has_changed) process_bezel_change();
        if(message_waiting) process_msg();
        //sleep_mode();
        //flash_led();
    }
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5========  process_bezel_change
//
void process_bezel_change()
{
    Clear(bezel_has_changed);
    if(mode == STEP) send_pulse();
    if(mode == CRUISE)
    {
        if(bezel.velocity == 0)
        {
            timer0off();
            sprintln("all stop");
            flash_led();
        }
        else
        {
            cycle = ((PULSE_DELAY/bezel.velocity) >> ustep);
            if(!manual_off) timer0on();
            //prtI(bezel.velocity);tab();
            //prtL(cycle);lf();
        }
    }
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6         process_msg
//
void process_msg()
{
    Clear(message_waiting);
    //clearBuffer(msg_buffer);
    byte index = 0;
    while(!isEmpty(&uartRx)) msg_buffer[index++] = getCh(&uartRx);
    msg_buffer[index] = 0;
    process_message(msg_buffer);
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6      processMessage
//
void process_message(char *msg)
{
    char _ch = (char)msg[0];
    char error_flag = 0;
    byte ndex = 0;

    switch (_ch)
    {
        /*
        case 32:    // <space>

            break;
```

```
case 36:    // '$'

    break;
case 43:    // '+'

    return;*/
case 45:    // '-'
    switch(msg[1])
    {
        case 118:
            sprint(VERSION);colon();
            sprint(BOARD);colon();
            prtB(DEVICE_ID);lf();
            break;
        case 77:
        case 109:
            if(mode == STEP)
            {
                mode = CRUISE;
                sprintln("Now in Cruise mode");
            }
            else
            {
                mode = STEP;
                sprintln("Now in Step mode");
            }
            reset_encoder();
            break;
        case 112:
            //print_test();
            break;
        default:
            Set(error_flag);
            break;
    }
    break;

case 47:    // '/'
    sprint("Execute command ");
    prtI(bezel.current);lf();
    break;
/*case 46:    // '.'

    break;
case 48 ... 57:

    break;
case 63:    // '?'

    break;
case 64:    // '@'
    break;

case 65:    // 'A'
case 97:    // 'a'

    break;
case 66:    // 'B'
case 98:    // 'b'

    break;
case 67:    // 'C'
case 99:    // 'c'
```

```
        break;

    case 68:    // 'D'
    case 100:   // 'd'

        break;

    case 71:    // 'G'
    case 103:   // 'g'

        break;
    case 73:    // 'I'
    case 105:   // 'i'

        break;
    case 76:    // 'L'
    case 108:   // 'l'

        break;

    case 77:    // 'M'
    case 109:   // 'm'

        break;

    case 78:    // 'N'
    case 110:   // 'n'
        break;

    case 80:    // 'P'
    case 112:   // 'p'

        break;

    case 81:    // 'Q'
    case 113:   // 'q'

        break;
        */
    case 82:    // 'R'
    case 114:   // 'r'
        if(msg[1] == 0) msg[1] = 48;
        bezel.rover = get_long(msg, &ndex);
        sprint("Rollover set to ");
        prtI(bezel.rover);lf();
        break;

    //case 83:    // 'S'
    //case 115:   // 's'

        //break;

    case 84:    // 'T'
        //if((_ch2 == 82) || (_ch2 == 114)) resetTimer0();
        timer0on();
        sprint("Timer on");lf();
        Clear(manual_off);
        Set(bezel_has_changed);
        break;
    case 116:   // 't'
        timer0off();
        sprint("Timer off");lf();
```

```
            Set(manual_off);
            Set(bezel_has_changed);
            break;
            /*
        case 85:    // 'U'
        case 117:   // 'u'

            break;

        case 86:    // 'V'
        case 118:   // 'v'

            break;
        case 87:    // 'W'
        case 119:   // 'w'

            break;
        case 88:    // 'X'
        case 120:   // 'x'

            break;
        case 89:    // 'Y'
        case 121:   // 'y'

            break;
            */
        case 90:    // 'Z'
        case 122:   // 'z'
            if(msg[1] == 0)
            {
                reset_encoder();
                if(mode == CRUISE) break;
            }
            else bezel.current = get_long(msg, &ndex);
            //Set(bezel_has_changed);
            sprint("Encoder set to ");
            prtI(bezel.current);lf();
            break;

        default:
            prtS(msg);lf();
            break;
    }

    if (error_flag) {sprint("Invalid Command"); lf();}
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6                prtC
//
void prtC(char ch)
{
    addCh(&uartTx, ch);
    enableU0();
    if(getBufLen(&uartTx) > buff_safety) _delay_us(BYTE_RATE);
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6                prtS
//
void prtS(char *s)
{
    byte i = 0;
    while(s[i]) prtC(s[i++]);
}
```

```
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6                prtB
//
void prtB(byte b)
{
    char s[4] = {};
    basc(b, s);
    prtS(s);
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6                prtI
//
void prtI(int i)
{
    char s[6];
    iasc(i, s);
    prtS(s);
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6                prtL
//
void prtL(long i)
{
    char s[11];
    lasc(i, s);
    prtS(s);
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6                prtD
//
void prtD(double d)
{
    long l = (long)d;
    prtL(l);
    prtC(DOT);
    int f = (((d + .0005) - l) * 1000);
    if(f < 100) prtC(48);
    if(f < 10)  prtC(48);
    prtI(f);
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6                lasc
//
void lasc(uint32_t b, char *c)
{   // these functions are slower the the gcc lib but lighter
    switch (b)
    {
        case 1000000000 ... 4294967295:
            *c++ = ((b / 1000000000) + 48);
            b %= 1000000000;
        case 100000000 ... 999999999:
            *c++ = ((b / 100000000) + 48);
            b %= 100000000;
        case 10000000 ... 99999999:
            *c++ = ((b / 10000000) + 48);
            b %= 10000000;
        case 1000000 ... 9999999:
            *c++ = ((b / 1000000) + 48);
            b %= 1000000;
        case 100000 ... 999999:
            *c++ = ((b / 100000) + 48);
            b %= 100000;
        case 10000 ... 99999:
            *c++ = ((b / 10000) + 48);
            b %= 10000;
        case 1000 ... 9999:
            *c++ = ((b / 1000) + 48);
            b %= 1000;
        case 100 ... 999:
            *c++ = ((b / 100) + 48);
            b %= 100;
        case 10 ... 99:
            *c++ = ((b / 10) + 48);
            b %= 10;
        default:
            *c++ = (b + 48);
            *c = 0;
            break;
    }
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6                iasc
//
void iasc(int b, char *c)
{
    if(b < 0) {*c++ = 45; b = -b;}
    switch (b)
    {
        case 10000 ... 32767:
            *c++ = ((b / 10000) + 48);
            b %= 10000;
        case 1000 ... 9999:
            *c++ = ((b / 1000) + 48);
            b %= 1000;
        case 100 ... 999:
            *c++ = ((b / 100) + 48);
            b %= 100;
        case 10 ... 99:
            *c++ = ((b / 10) + 48);
            b %= 10;
        default:
            *c++ = (b + 48);
            *c = 0;
            break;
    }
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6                basc
//
void basc(byte b, char *c)
{
    switch (b)
    {
        case 100 ... 255:
            *c++ = ((b / 100) + 48);
            b %= 100;
        case 10 ... 99:
            *c++ = ((b / 10) + 48);
            b %= 10;
        default:
            *c++ = (b + 48);
            *c = 0;
            break;
    }
}
//=======1=========2=========3=========4=========5=========6=========7=========8
```

```
//=======1=========2=========3=========4=========5=========6            get_long
//
uint32_t get_long(char *s, byte *b)
{   // returns the first number found starting from position b
    // non-numerics until the first occurence of a numeric are ignored.
    // b is passed in by reference and its return value indicates the first
    // array position following the end of the previous number and so may be
    // used to extract the next in a string array
    #define digit(c) ((c >= 48 && c <= 57)?1:0)
    long number = 0;
    while (!digit(s[*b])) (*b)++;
    while (digit(s[*b])) (number = (number * 10) + (s[(*b)++] - 48));
    return number;
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6                wait
//
void wait(uint32_t us)
{
    uint32_t start_wait = micros();
    while ((micros() - start_wait) < us);
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6              millis
//
uint32_t millis()
{
    unsigned long m;
    uint8_t oldSREG = SREG;

    cli();
    m = timer0_millis;
    SREG = oldSREG;

    return m;
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6              micros
//
uint32_t micros()
{
    unsigned long m;
    uint8_t oldSREG = SREG, t;

    cli();
    m = timer0_overflow_count;
    t = TCNT0;

    if ((TIFR0 & _BV(TOV0)) && (t < 255)) m++;

    SREG = oldSREG;

    return ((m << 8) + t) * (64 / clockCyclesPerMicrosecond());
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6        reset_timer0
//
void reset_timer0()
{
    timer0_fract = 0;
    timer0_millis = 0;
    timer0_overflow_count = 0;
    old_micros = 0;
}
//=======1=========2=========3=========4=========5=========6=========7=========8
//=======1=========2=========3=========4=========5=========6     TIMER0_OVF_vect
//
ISR(TIMER0_OVF_vect)
{
    unsigned long m = timer0_millis;
    byte f = timer0_fract;

    m += MILLIS_INC;
    f += FRACT_INC;
    if (f >= FRACT_MAX)
    {
        f -= FRACT_MAX;
        m += 1;
    }

    timer0_fract = f;
    timer0_millis = m;
    timer0_overflow_count++;
}
//=======1=========2=========3=========4=========5=========6      end of program
```