

```

/*****
***
***      eeeee eeeee eeeee eeeee eeeee eeeee
***      8      8 8      8      8      8 8 8      8
***      8e     8 8      8      8      8      8 8      8
***      88     8 8eee  8eee  8e     8      8 8e     8
***      88     8 8e    8e    88     8      8 88     8
***      88     e 88     88     88     8      e 88     8
***      88eee8 88eee 88     88eee 88eee8 88     8
***      -----
***              defcon-cc.dyndns.org
***
*****/
***      Programm: laptimer          ***
***      Autor: Patrick Langosch    ***
***      erstellt: 11.04.14         ***
***      Version: 1.0              ***
*****/
/*****
***
***      Anschlußbelegung ATMEGA8
***
***      Pin      Description      Connected with
***      -----
***      1        PC6 (RESET)      RC-Network
***      2        PD0 (RXD)        Connector (Bootloader)
***      3        PD1 (TXD)        Connector (Bootloader)
***      4        PD2 (INT0)       Drehzahl
***      5        PD3 (INT1)       Geschwindigkeit
***      6        PD4 (XCK/T0)     -
***      7        VCC              +5V
***      8        GND              GND
***      9        PB6 (XTAL1)      4Mhz Quarz
***      10       PB7 (XTAL2)      4Mhz Quarz
***      11       PD5 (T1)         -
***      12       PD6 (AIN0)       -
***      13       PD7 (AIN1)       Bestzeit-LED
***      14       PB0 (ICP)        Rundenzeit (Reed-Kontakt)
***      15       PB1 (OC1A)       Transistor (LCD_BACKGRND)
***      16       PB2 (SS)         Taster "Hoch"
***      17       PB3 (MOSI)       Taster "Runter"
***      18       PB4 (MISO)       Taster "OK/EXIT"
***      19       PB5 (SCK)        -
***      20       AVCC             +5V
***      21       AREF             -
***      22       AGND            GND
***      23       PC0 (ADC0)       LCD (DB7)
***      24       PC1 (ADC1)       LCD (DB6)
***      25       PC2 (ADC2)       LCD (DB5)
***      26       PC3 (ADC3)       LCD (DB4)
***      27       PC4 (ADC4)       LCD (Enable)
***      28       PC5 (ADC5)       LCD (RS)
***
*****/
/*****
***
***      Changelog:
***
***      - ver 1.0 adding interim support,
***        store besttime in eeprom, adding menu
***        fixed point arithmetic, usart
***        4-stroke-engine supported (RPM),
***        floating average
***
***      - ver 0.9 switched to Atmega8 (more Flash)
***        adding velocity + rpm
***

```

```
*** - ver 0.8 reducing code size
***
*** - ver 0.7 no RS-FF needed,
***   debounce by Software, LCD con. changed
***
*** - ver 0.6 using new µC ATTiny2313
***   complete workaround of time calculation
***
*****/
```

```
#include <math.h>
#include <avr/io.h>
#include <stdlib.h>
#include <string.h>
#include <avr/wdt.h>
#include <avr/sleep.h>
#include <avr/eeprom.h>
#include <util/delay.h>
#include <util/atomic.h>
#include <avr/interrupt.h>
```

```
// GENERAL
```

```
#define F_CPU 4000000UL // Systemtakt in Hz
#define TRUE 1
#define FALSE 0
#define ERROR -1
#define ON TRUE
#define OFF FALSE
```

```
// SWITCHES
```

```
#define ACCESS_EEPROM // Soll der EEPROM genutzt werden?
```

```
// PARAMETERS
```

```
#define N_STROKE_ENGINE 2 // 2-Takt oder 4-Takt Motor? Zwei oder Vier eintragen!
#define SEC_LOCK_ICP 1 // Zeit die der ICP nach auslösen gesperrt bleibt, in Sekunden.
#define STANDBY_AFTER_MIN 6 // Zurücksetzen der Rundenzeitmessung und Standby (Runde nach x Minuten noch nicht beendet)
#define MAX_INTERIM_TICKS 12000 // Schwelle für Differenz zu Best-Zwischenzeiten (ganze Minuten!!)
#define AVG_N_ORDER 6 // Ordnung der gleitenden Mittelwertberechnung
#define AMOUNT_MAGNETIC_LOOPS 3 // Anzahl der Magnetschleifen auf der Rennstrecke
#define KPH_TO_MPS 3.6 // Umrechnung km/h <-> m/s
#define PERIMETER_TYRE 0.86 // Reifenumfang in Meter
#define FRONT_SPROCKET 10 // Zähnezah Ritzel
#define REAR_SPROCKET 86 // Zähnezah Kettenrad
#define SEC_SHOW_LAPTIME 5 // Zeit in Sekunden, bei der die alte Rundenzeit angezeigt wird
```

```
// LIMITS
```

```
#define LIMIT_MIN_TICKS_RPM 6000 // 100 U/min
#define LIMIT_MAX_TICKS_RPM 25 // 24000 U/min
#define LIMIT_MIN_TICKS_KPH 30960 // 1 km/h
#define LIMIT_MAX_TICKS_KPH 129 // 240 km/h
```

```
// TIMEOUTS
```

```
#define TIMEOUT_RPM LIMIT_MIN_TICKS_RPM
#define TIMEOUT_KPH LIMIT_MIN_TICKS_KPH
```

```
// BUTTONS
```

```
#define BUTTON_UP PB2
#define BUTTON_DOWN PB3
#define BUTTON_OK_EXIT PB4
```

```
// USART
```

```
// USART
#define BAUD 19200UL // Baudrate für serielle Kommunikation
#define USART_TAKT F_CPU/16/BAUD-1 // Takt für den USART
```

```
// LED
#define LED_PORT PORTD
#define LED_BESTTIME PD7

// CURSOR
#define CURSOR_UP -1
#define CURSOR_DOWN +1

// MENU pages
#define NUM_PAGES 1 // number of menu pages
#define PAGE_MAIN 0

// LCD
#define LCD_LINES 4 // number of display lines
#define LCD_DISP_LENGTH 20 // number of character in one line
#define CLEAR_DISPLAY 0x01 // Display Inhalt komplett mit Null überschreiben
#define CURSOR_HOME 0x02 // Cursor an Anfangsposition setzen
#define LCD_CURSOR_ON 0x0F
#define LCD_CURSOR_OFF 0x0C
#define LCD_PORT PORTC // Port an dem das LCD angeschlossen ist
#define LCD_DDR DDRC // Datenpins an denen das LCD angeschlossen ist
#define LCD_RS PC5 // Ausgang an dem der RS-Pin des LCD hängt
#define LCD_EN PC4 // Ausgang an dem der EN-Pin des LCD hängt

// EEPROM
#define EEPROM_BYTE_DEF 0xFF // EEPROM-Zelle leer
#define EEPROM_WORD_DEF 0xFFFF

// SOME CHECKS
#ifndef AMOUNT_MAGNETIC_LOOPS
    #error Anzahl der Magnetschleifen nicht definiert!
#endif

#ifdef AMOUNT_MAGNETIC_LOOPS
    #if AMOUNT_MAGNETIC_LOOPS < 1
        #error Anzahl der Magnetschleifen kann nicht negativ sein!
    #elif AMOUNT_MAGNETIC_LOOPS > 6
        #error Es werden max. sech Magnetschleifen unterstützt!
    #else
        #if AMOUNT_MAGNETIC_LOOPS == 1
            #define INIT_BLOCK 0
        #elif AMOUNT_MAGNETIC_LOOPS == 2
            #define INIT_BLOCK 0, 0
        #elif AMOUNT_MAGNETIC_LOOPS == 3
            #define INIT_BLOCK 0, 0, 0
        #elif AMOUNT_MAGNETIC_LOOPS == 4
            #define INIT_BLOCK 0, 0, 0, 0
        #elif AMOUNT_MAGNETIC_LOOPS == 5
            #define INIT_BLOCK 0, 0, 0, 0, 0
        #elif AMOUNT_MAGNETIC_LOOPS == 6
            #define INIT_BLOCK 0, 0, 0, 0, 0, 0
        #endif
    #endif
#endif

// Struktur für die diversen Zeiten
struct myTime
{
    uint8_t min;
    uint8_t sec;
    uint8_t csec;
    uint16_t ticks;
    uint16_t interim[AMOUNT_MAGNETIC_LOOPS];
};

// Struktur für die Einstellungen
```

```
struct conf
{
    uint8_t amount_magnetic_loops;
    uint8_t perimeter_tyre;
    uint8_t front_sprocket;
    uint8_t rear_sprocket;
};

// Struktur für den gleitenden Mittelwert
struct floating_avg
{
    uint16_t data[AVG_N_ORDER];
    uint8_t index;
};

char buffer[12];

volatile struct myTime lap; // die letzte Rundenzeit
volatile struct myTime act; // aktuell mitlaufende Zeit
struct myTime show; // die angezeigte Zeit
struct myTime best; // Bestzeit
struct myTime interim; // Zwischenzeit
struct conf config; // Konfiguration zur Laufzeit

// EEPROM
uint8_t eeprom_best_min EEMEM = 0;
uint8_t eeprom_best_sec EEMEM = 0;
uint8_t eeprom_best_csec EEMEM = 0;
uint16_t eeprom_best_ticks EEMEM = 0;
uint16_t eeprom_best_interim[AMOUNT_MAGNETIC_LOOPS] EEMEM = {INIT_BLOCK};

uint8_t eeprom_amount_magnetic_loops EEMEM = 0;
uint8_t eeprom_perimeter_tyre EEMEM = 0.0;
uint8_t eeprom_front_sprocket EEMEM = 0;
uint8_t eeprom_rear_sprocket EEMEM = 0;

uint8_t flag_besttime = 0; // Flag ob Bestzeit gefahren wurde
uint8_t brightness = 0; // Helligkeit der LCD-Hintergrundbeleuchtung
volatile uint8_t initial_start = 1; // Wird zurückgesetzt wenn Magnetschleife das aller
erste mal belegt wird
volatile uint8_t temp = 0; // Zwischenspeicher für Sekunden
volatile uint8_t loop_crossed = 0; // Flag signalisiert das soeben eine Magnetschleife
überfahren wurde
volatile uint8_t run_over_magnetic_loops = 0; // überfahrene Magnetschleifen
volatile uint8_t lap_complete = 0; // Flag Runde beendet
volatile uint8_t timeout_rpm = 0; // Flag Timeout Ereignis Geschwindigkeit
volatile uint8_t timeout_kph = 0; // Flag Timeout Ereignis Drehzahl
volatile uint16_t temp_ticks = 0; // Ticks zwischenspeicher für Zwischenzeiten
volatile uint16_t overflow_ticks_10ms = 0; // Zähler der Overflows von TCNT1, max: 6553
6 = 10min bis Overflow
volatile uint16_t overflow_ticks_100us = 0; // Zähler der Overflows von TCNT1, max: 655
36 = 6sec bis Overflow
volatile uint16_t retired_rpm_ticks_100us = 0; // Alter Wert aus vorgehenden Aufruf
volatile uint16_t retired_kph_ticks_100us = 0; // Alter Wert aus vorgehenden Aufruf
volatile uint16_t diff_rpm_ticks_100us = 0; // Differenz aus aktellen und vorherigen We
rt
volatile uint16_t diff_kph_ticks_100us = 0; // Differenz aus aktellen und vorherigen We
rt
volatile uint16_t timeout_ticks_rpm = 0; // Timeout-Ticks Zähler ab dem Drehzahl Null a
ngezeigt wird
volatile uint16_t timeout_ticks_kph = 0; // Timeout-Ticks Zähler ab dem Drehzahl Null a
ngezeigt wird

// Prototypendeklaration
void initTimer0(void);
void initTimer1(void);
void initExtInt(void);
void initTimer2(void);
```

```

void initStruct(struct myTime*);
void LCD_init(void);
void LCD_clear(void);
void LCD_home(void);
void LCD_data(unsigned char);
void LCD_string(const char*);
void LCD_string_fixed_point(char*, uint8_t, uint8_t, uint8_t);
void LCD_set_cursor(uint8_t, uint8_t);
void LCD_showLaptime(void);
void LCD_showStaticText(void);
void LCD_backgrnd(uint8_t);
void LCD_backgrnd_fade(uint8_t);
void LCD_showKPH(uint8_t);
void LCD_showRPM(uint8_t);
void LCD_showLaps(uint8_t);
void LCD_showInterim(int16_t);
void LCD_indicateWrite(void);
void USART_init(uint8_t);
void USART_send(char *data);
void AVG_init(struct floating_avg*, uint8_t);
void AVG_addValue(struct floating_avg*, uint8_t);
uint16_t AVG_getAVG(struct floating_avg*);
void LED_besttime(uint8_t);
void LCD_showMenu(struct conf*);
void splitTicks(uint16_t, volatile struct myTime*);
uint8_t calcRPM(uint16_t);
uint8_t calcKPH(uint16_t);
unsigned char swap_nibble(unsigned char);
void EEPROM_initCells(void);
void EEPROM_readBesttime(struct myTime*);
void EEPROM_readConfig(struct conf*);
void EEPROM_writeConfig(struct conf*);
void EEPROM_writeBesttime(struct myTime*);
void MENU_control_init(void);
void MENU_control(void);
void MENU_editValue(uint8_t, uint8_t, uint8_t, uint8_t, uint8_t, uint8_t*, uint8_t);
uint8_t MENU_debounce_input(volatile uint8_t*, uint8_t);
uint8_t MENU_direction(int8_t);
char *my_utoa(uint8_t, char*);
uint8_t timeButtonPressed(volatile uint8_t*, uint8_t, uint8_t);

/***** ISR des Externen Interrupts 0 (Drehzahl) *****/
ISR(INT0_vect)
{
    timeout_ticks_rpm = 0;

    // Zeitliche Differenz zwischen Aufruf dieser ISR ergibt Drehzahl
    if(overflow_ticks_100us > retired_rpm_ticks_100us)
    {
        diff_rpm_ticks_100us = overflow_ticks_100us - retired_rpm_ticks_100us; // Zwisc
henzeitlich kein Überlauf!
    }

    else
    {
        diff_rpm_ticks_100us = (65536 - retired_rpm_ticks_100us) + overflow_ticks_100us
; // Überlauf der Variablen overflow_ticks_100us
    }

    retired_rpm_ticks_100us = overflow_ticks_100us; // Wert zwischenspeichern für nächs
ten Aufruf der ISR
}

/***** ISR des Externen Interrupts 1 (Geschwindigkeit) *****/
ISR(INT1_vect)
{
    timeout_ticks_kph = 0;

```

```

// Geschwindigkeit gesondert erfassen
if(overflow_ticks_100us > retired_kph_ticks_100us)
{
    diff_kph_ticks_100us = overflow_ticks_100us - retired_kph_ticks_100us; // Zwisc
henzeitlich kein Überlauf!
}

else
{
    diff_kph_ticks_100us = (65536 - retired_kph_ticks_100us) + overflow_ticks_100us
; // Überlauf der Variablen overflow_ticks_100us
}

retired_kph_ticks_100us = overflow_ticks_100us; // Wert zwischenspeichern für nächs
ten Aufruf der ISR
}

/***** ISR des Overflow von Timer0 *****/
ISR(TIMER0_OVF_vect)
{
    TCNT0 = -50; // Timer Register wird mit einem Wert vorbelegt, 2^8 - 206 = 50 als
Rest
// somit ist mit jedem Aufruf der ISR 0.1ms vergangen

    overflow_ticks_100us++; // Mit jedem Aufruf der ISR sind 100µs vergangen
}

/***** ISR des Input Capture (Rundenzeit) (Timer1 / 16Bit) *****/
ISR(TIMER1_CAPT_vect) // Aufruf bei steigender Flanke an ICP
{
    TIMSK &= ~(1 << TICIE1); // Input Capture Interrupt deaktivieren
    TIFR &= (1 << ICF1); // Zur Sicherheit das IC-Flag löschen, durch schreiben einer e
ins

    if(initial_start == FALSE) // Kein Erststart wenn = 0
    {
        loop_crossed = TRUE; // Flag signalisiert das soeben eine Magnetschleife überfa
hren wurde
        act.interim[run_over_magnetic_loops] = overflow_ticks_10ms; // Zwischenzeit spe
ichern

        run_over_magnetic_loops++; // Anzahl der überfahrenen Magnetschleifen hochzähle
n
    }

    else // Wenn Magnetschleife das erste mal überfahren wird = 1
    {
        ICR1 = TCNT1 - ICR1; // Wert für Input Capture Register berechnen, die Takte di
e zwischen aufruf der ISR und dem setzen von ICR1 vergangen sind
        TCNT1 = -5000 + (ICR1 + 6); // Takte die zwischen vorherigen Befehl ablaufen we
rden dazuaddiert

        initial_start = FALSE;
    }

    // Wenn alle Magnetschleifen auf der Rennstrecke überfahren wurden
    // d.h. eine Runde abgeschlossen wurde
    if(run_over_magnetic_loops == AMOUNT_MAGNETIC_LOOPS)
    {
        run_over_magnetic_loops = 0; // Überfahrene Magnetschleifen zurücksetzen

        ICR1 = TCNT1 - ICR1; // Wert für Input Capture Register berechnen, die Takte di
e zwischen aufruf der ISR und dem setzen von ICR1 vergangen sind
        TCNT1 = -5000 + (ICR1 + 2); // Takte die zwischen vorherigen Befehl ablaufen we
rden dazuaddiert
        lap_complete = TRUE; // Flag setzen wenn Runde beendet
    }
}

```

```

        lap = act; // Werte übertragen
        overflow_ticks_10ms = 0; // Overflow Zähler auf Null setzen
    }

    // Sorgt dafür das Pegeländerungen welche durch Prellen des Reed-Kontaktes
    // hervorgerufen werden nicht an den ICP1 Pin gelangen können
    temp = act.sec;
}

/***** ISR des Overflow von Timer1 *****/
ISR(TIMER1_OVF_vect)
{
    TCNT1 = -5000; // Timer Register wird mit einem Wert vorbelegt, 2^16 - 60536 = 5000
    als Rest
                // somit ist mit jedem Aufruf der ISR 10^-2 Sekunden (0.01) vergange
n

    if(initial_start == FALSE) // Erstüberfahrt einer Magnetschleife
    {
        // d.h jedes mal wenn overflow_ticks_10ms um eins inkrementiert wird,
        // sind 10ms vergangen
        overflow_ticks_10ms++; // max. 65536 da 16Bit Variable
    }

    // Timeout Zähler der Drehzahl und Geschwindigkeit
    if(timeout_ticks_rpm < (TIMEOUT_RPM / 100))
    {
        timeout_ticks_rpm++;
        timeout_rpm = FALSE;
    }

    else timeout_rpm = TRUE;

    if(timeout_ticks_kph < (TIMEOUT_KPH / 100))
    {
        timeout_ticks_kph++;
        timeout_kph = FALSE;
    }

    else timeout_kph = TRUE;
}

/***** ISR des Compare Match von Timer2 *****/
ISR(TIMER2_COMP_vect)
{
    PORTB &= ~(1 << PB1); // LCD-Hintergrundbeleuchtung (PWM)
}

/***** ISR des Overflow von Timer2 *****/
ISR(TIMER2_OVF_vect)
{
    PORTB |= (1 << PB1); // LCD-Hintergrundbeleuchtung (PWM)
}

/***** MAIN *****/
int main(void)
{
    DDRB = 0x02; // Setze Datenrichtungsregister DDRB
    DDRD = 0x80; // Setze Datenrichtungsregister DDRD

    struct floating_avg avg_rpm; // gleitender Mittelwert (Drehzahl)
    struct floating_avg avg_velocity; // gleitender Mittelwert (Geschwindigkeit)

    uint8_t nLaps = 0;
    uint8_t standby = FALSE;
    uint8_t pre_rpm = 0;
    uint8_t kph = 0;
    int16_t diff_ticks = 0;

```

```

cli(); // Globale Interrupts ausschalten

ACSR = (1 << ACD); // Analog Komparator Deaktivieren
WDTCSR &= ~(1 << WDE); // Disable Watchdog

// Alle Member der Instanzen mit Null initialisieren
initStruct((struct myTime*) &act);
initStruct((struct myTime*) &show);
initStruct((struct myTime*) &lap);
initStruct((struct myTime*) &best);
initStruct((struct myTime*) &interim);

// Init. peripherie
initExtInt();
initTimer0();
initTimer1();
initTimer2();

AVG_init(&avg_rpm, 0); // Mittelwert-Strukturen initialisieren
AVG_init(&avg_velocity, 0); // Mittelwert-Strukturen initialisieren
LCD_init(); // LCD Anzeige initialisieren
USART_init(USART_TAKT); // USART initialisieren
MENU_control_init(); // Menüsteuerung initialisieren

LCD_backgrnd_fade(TRUE); // Hintergrundbeleuchtung einschalten
LED_besttime(OFF); // LED für Bestzeit gefahren aus

#ifdef ACCESS_EEPROM
// ***** OK/EXIT *****
if(MENU_debounce_input(&PINB, BUTTON_OK_EXIT))
{
// Initialisiert die EEPROM-Speicherzellen
EEPROM_initCells();
LCD_string("Init. EEPROM!");
LCD_indicateWrite();
}

EEPROM_readConfig(&config);
EEPROM_readBesttime(&best);
#endif

LCD_clear(); // LCD Anzeige komplett leeren
LCD_showStaticText(); // spart Zeit in der Endlosschleife

sei(); // Globale Interrupts einschalten

/***** LOOP *****/
/
while(TRUE)
{
// DEBUG
//LED_PORT ^= (1 << LED_BESTTIME);

// Zwischenzeit sichern sobald Magnetscheleife überfahren wurde
if(loop_crossed == TRUE && nLaps != 0)
{
diff_ticks = act.interim[run_over_magnetic_loops-1] - best.interim[run_over_magnetic_loops-1];

// Es werden nur Differenzen zu den Best-Zwischenzeiten angezeigt die unter
einer // definierten Schwelle liegen. Sind die Best-Zwischenzeiten null (Passiert
wenn // Bestzeit aus EEPROM geladen wird) diese nicht anzeigen.
if(abs(diff_ticks) < MAX_INTERIM_TICKS && best.interim[run_over_magnetic_loops-1] != 0)
{

```



```
        splitTicks((uint16_t)(abs(diff_ticks)), &interim);
    }

    loop_crossed = FALSE;
}

if(initial_start == FALSE) // Wenn Rundenzeit-Messung am laufen, kein Erststart
{
    splitTicks(overflow_ticks_10ms, &act); // Zeiten ermitteln

// Wurde die Runde noch nicht beendet wird die aktuell mitlaufende Zeit ang
zeigt
    if(lap_complete == FALSE)
    {
        show = act; // Aktuelle Zeit anzeigen
    }
}

// Wenn die Runde beendet wurde d.h. alle Magnetschleifen überfahren wurden
if(lap_complete == TRUE)
{
    if(act.sec < SEC_SHOW_LAPTIME) // LCD zeigt die alte Rundenzeit an wenn tru
e
    {
        show = lap; // Die vorherige Rundenzeit anzeigen
    }

    else // Nach ablauf der Zeit wird dann die aktuell mitlaufende Zeit angezei
gt
    {
        show = act; // Aktuelle Rundenzeit anzeigen
        lap_complete = 0; // Es beginnt eine neue Runde

        nLaps++;
    }

    if(best.min == 0 && best.sec == 0 && best.csec == 0) // Wenn noch keine Bes
tzeit eingetragen ist, erste Rundenzeit zur Bestzeit machen
    {
        best = lap; // Die vorherige Rundenzeit wird zu Bestzeit gemacht
        LED_besttime(ON); // LED für Bestzeit gefahren ein
    }

    if(best.min != 0 || best.sec != 0 || best.csec != 0) // Wenn schon eine Bes
tzeit eingetragen ist, muss die jetzige verglichen werden
    {
        if(best.ticks > lap.ticks) // Wenn Rundenzeit schneller als Bestzeit ->
Rundenzeit wird zur Bestzeit
        {
            best = lap; // Rundenzeit zur Bestzeit machen
            LED_besttime(ON); // LED für Bestzeit gefahren an

            #ifdef ACCESS_EEPROM
                EEPROM_writeBesttime(&best); // Bestzeit im EEPROM sichern
            #endif
        }

        else temp = 0;
    }
}

if(act.min >= STANDBY_AFTER_MIN) // Standby wenn Runde nicht in gewisser Zeit b
eendet wird
{
    // creates a block of code that is guaranteed
    // to be executed atomically
    ATOMIC_BLOCK(ATOMIC_FORCEON)
    {
```

```
// Diverse variablen zurücksetzen
initial_start = TRUE;
loop_crossed = FALSE;
overflow_ticks_10ms = 0;
run_over_magnetic_loops = 0;

initStruct((struct myTime*) &act);
initStruct((struct myTime*) &lap);
initStruct((struct myTime*) &show);

LCD_showLaptime();
standby = TRUE;

LCD_backgrnd_fade(FALSE); // Stromverbrauch senken
set_sleep_mode(SLEEP_MODE_IDLE); // µc in Idle setzen
sleep_mode();
}
}

if(initial_start == FALSE && standby == TRUE) // Aufgewacht aus dem Standby
{
    standby = FALSE;
    LCD_backgrnd_fade(TRUE); // Hintergrundbeleuchtung wieder einschalten
}

// Es wird ein gewisse Zeit gewartet bis wieder neue Flanken
// am ICP1 Pin durchkommen
if((act.sec - temp) >= SEC_LOCK_ICP)
{
    temp = 0; // Immer neu initialisieren

    TIFR &= (1 << ICF1); // Zur Sicherheit das IC-Flag löschen, durch schreiben
einer eins
    TIMSK |= (1 << TICIE1); // Input Capture Interrupt wieder aktivieren
}

if(act.sec > SEC_LOCK_ICP && flag_besttime == TRUE) // Nach 1sek die LED (Bestzeit) wieder ausschalten
{
    LED_besttime(OFF); // LED für Bestzeit gefahren aus
    flag_besttime = FALSE; // Flag wieder zurücksetzen

    temp = 0; // Immer neu initialisieren

    TIFR &= (1 << ICF1); // Zur Sicherheit das IC-Flag löschen, durch schreiben
einer eins
    TIMSK |= (1 << TICIE1); // Input Capture Interrupt wieder aktivieren
}

LCD_showLaptime(); // Rundenzeiten per LCD anzeigen
LCD_showLaps(nLaps); // Rundenzahl anzeigen

if(diff_ticks > MAX_INTERIM_TICKS)
{
    interim.min = MAX_INTERIM_TICKS / 6000;
    interim.sec = 0;
    interim.csec = 0;

    LCD_showInterim(diff_ticks); // Zwischenzeit anzeigen

    LCD_set_cursor(12, 3);
    LCD_data('>');
}

else
{
    LCD_showInterim(diff_ticks); // Zwischenzeiten + Rundenzahl
}
}
```

```

/***** Drehzahl *****/
if(timeout_rpm) pre_rpm = 0; // Bei Timeout die Drehzahl auf Null setzen
else pre_rpm = calcRPM(diff_rpm_ticks_100us); // Drehzahl ermitteln

// Gleitender Mittelwert
AVG_addValue(&avg_rpm, pre_rpm);
LCD_showRPM(AVG_getAVG(&avg_rpm)); // Drehzahl anzeigen
//LCD_showRPM(pre_rpm); // Drehzahl anzeigen

/***** Geschwindigkeit *****/
if(config.front_sprocket != 0 && config.rear_sprocket != 0)
{
    //kph = pre_rpm * (float)((100 * FRONT_SPROCKET * PERIMETER_TYRE * KPH_TO_M
PS) / (REAR_SPROCKET * 60)); // Berechne Geschwindigkeit aus Drehzahl
    //kph = avg_pre_rpm * (float)((100 * FRONT_SPROCKET * PERIMETER_TYRE * KPH_
TO_MPS) / (REAR_SPROCKET * 60)); // Berechne Geschwindigkeit aus Drehzahl
    kph = AVG_getAVG(&avg_rpm) * (float)((100 * FRONT_SPROCKET * PERIMETER_TYRE
* KPH_TO_MPS) / (REAR_SPROCKET * 60)); // Berechne Geschwindigkeit aus Drehzahl
    LCD_showKPH(kph); // Geschwindigkeit anzeigen
}

else
{
    if(timeout_kph) kph = 0; // Bei Timeout die Geschwindigkeit auf Null setzen
    else kph = calcKPH(diff_kph_ticks_100us); // Geschwindigkeit ermitteln

    // Gleitender Mittelwert
    AVG_addValue(&avg_velocity, kph);
    LCD_showKPH(AVG_getAVG(&avg_velocity)); // Geschwindigkeit anzeigen
    //LCD_showKPH(kph); // Geschwindigkeit anzeigen
}

// Debug
USART_send(utoa(pre_rpm, buffer, 10));
USART_send(",");
USART_send(utoa(kph, buffer, 10));
USART_send("\n");

MENU_control(); // Menüsteuerung
}

return 0; // Wird niemals erreicht
}

/***** EEPROM *****/
/***** Initialisiert den die benutzten EEPROM-Speicherzellen *****/
void EEPROM_initCells(void)
{
    struct conf confInit;
    struct myTime myTimeInit;

    initStruct((struct myTime*) &myTimeInit);

    confInit.amount_magnetic_loops = AMOUNT_MAGNETIC_LOOPS;
    confInit.perimeter_tyre = PERIMETER_TYRE * 100;
    confInit.front_sprocket = FRONT_SPROCKET;
    confInit.rear_sprocket = REAR_SPROCKET;

    EEPROM_writeBesttime(&myTimeInit);
    EEPROM_writeConfig(&confInit);
}

/***** Schreibt die Bestzeit ins EEPROM *****/
void EEPROM_writeBesttime(struct myTime* besttime)
{

```

```

    // Nur schreiben wenn Wert in Speicherzelle neu ist
    eeprom_update_byte(&eeprom_best_min, besttime->min);
    eeprom_update_byte(&eeprom_best_sec, besttime->sec);
    eeprom_update_byte(&eeprom_best_csec, besttime->csec);
    eeprom_update_word(&eeprom_best_ticks, besttime->ticks);
    eeprom_update_block(besttime->interim, eeprom_best_interim, sizeof(besttime->interi
m));
}

/***** Schreibt die Einstellungen ins EEPROM *****/
void EEPROM_writeConfig(struct conf* config)
{
    // Nur schreiben wenn Wert in Speicherzelle neu ist
    eeprom_update_byte(&eeprom_amount_magnetic_loops, config->amount_magnetic_loops);
    eeprom_update_byte(&eeprom_perimeter_tyre, config->perimeter_tyre);
    eeprom_update_byte(&eeprom_front_sprocket, config->front_sprocket);
    eeprom_update_byte(&eeprom_rear_sprocket, config->rear_sprocket);
}

/***** Ließt die Bestzeit aus dem EEPROM *****/
void EEPROM_readBesttime(struct myTime* besttime)
{
    uint8_t index = 0;

    uint8_t temp_byte = 0;
    uint16_t temp_word = 0;
    uint16_t temp_block[AMOUNT_MAGNETIC_LOOPS];

    temp_byte = eeprom_read_byte(&eeprom_best_min); // Minuten

    // EEPROM leer --> Defaultwert übernehmen
    if(temp_byte == EEPROM_BYTE_DEF) besttime->min = 0;
    else besttime->min = temp_byte;

    temp_byte = eeprom_read_byte(&eeprom_best_sec); // Sekunden

    // EEPROM leer --> Defaultwert übernehmen
    if(temp_byte == EEPROM_BYTE_DEF) besttime->sec = 0;
    else besttime->sec = temp_byte;

    temp_byte = eeprom_read_byte(&eeprom_best_csec); // Centisekunden

    // EEPROM leer --> Defaultwert übernehmen
    if(temp_byte == EEPROM_BYTE_DEF) besttime->csec = 0;
    else besttime->csec = temp_byte;

    temp_word = eeprom_read_word(&eeprom_best_ticks); // Ticks

    // EEPROM leer --> Defaultwert übernehmen
    if(temp_word == EEPROM_WORD_DEF) besttime->ticks = EEPROM_WORD_DEF;
    else besttime->ticks = temp_word;

    eeprom_read_block(temp_block, eeprom_best_interim, sizeof(temp_block));

    // EEPROM leer --> Defaultwert übernehmen
    for(index = 0; index < AMOUNT_MAGNETIC_LOOPS; index++)
    {
        if(temp_block[index] == EEPROM_WORD_DEF) besttime->interim[index] = EEPROM_WORD
_DEF;
        else besttime->interim[index] = temp_block[index];
    }
}

/***** Ließt die Bestzeit aus dem EEPROM *****/
void EEPROM_readConfig(struct conf* config)
{
    uint8_t temp_byte = 0;

```

```

temp_byte = eeprom_read_byte(&eeprom_amount_magnetic_loops); // Anzahl Magnetschleifen
fen

// EEPROM leer oder ungültig --> Defaultwert übernehmen
// Max Anzahl an Magnetschleifen = 6
if(temp_byte > 6) config->amount_magnetic_loops = AMOUNT_MAGNETIC_LOOPS;
else config->amount_magnetic_loops = temp_byte;

temp_byte = eeprom_read_byte(&eeprom_perimeter_tyre); // Durchmesser Reifen

// EEPROM leer --> Defaultwert übernehmen
if(temp_byte == EEPROM_BYTE_DEF) config->perimeter_tyre = PERIMETER_TYRE;
else config->perimeter_tyre = temp_byte;

temp_byte = eeprom_read_byte(&eeprom_front_sprocket); // Ritzel

// EEPROM leer --> Defaultwert übernehmen
if(temp_byte == EEPROM_BYTE_DEF) config->front_sprocket = FRONT_SPROCKET;
else config->front_sprocket = temp_byte;

temp_byte = eeprom_read_byte(&eeprom_rear_sprocket); // Kettenrad

// EEPROM leer --> Defaultwert übernehmen
if(temp_byte == EEPROM_BYTE_DEF) config->rear_sprocket = REAR_SPROCKET;
else config->rear_sprocket = temp_byte;
}

/***** USART *****/
/***** USART initialisieren *****/
void USART_init(uint8_t ubrr)
{
    UBRRH = (unsigned char)(ubrr >> 8);
    UBRRL = (unsigned char)(ubrr & 0xFF);

    UCSRB |= (1 << TXEN) | (1 << RXEN); // TX + RX einschalten
    UCSRC |= (1 << URSEL) | (1 << UCSZ1) | (1 << UCSZ0); // Asynchron 8N1
}

/***** Daten über den USART senden *****/
void USART_send(char *data)
{
    while(*data != '\0') // solange kein String-Endzeichen kommt
    {
        while(!(UCSRA & (1 << UDRE))); // Warten bis Senden möglich ist

        UDR = *data;
        data++; // Nächsten Character schnappen
    }
}

/***** AVERAGE *****/
/***** Mittelwert-Struktur initialisieren *****/
void AVG_init(struct floating_avg* avg, uint8_t default_value)
{
    uint8_t index = 0;

    for(index = 0; index < AVG_N_ORDER; ++index)
    {
        avg->data[index] = default_value;
    }

    avg->index = 0;
}

/***** Wert zur Mittelwertberechnung hinzufügen *****/
void AVG_addValue(struct floating_avg* avg, uint8_t new_value)

```

```

{
    avg->data[avg->index] = new_value; // Neuen Wert in den Buffer schreiben
    avg->index++; // Positions-Variable zeigt auf die nächste freie Stelle
    avg->index %= AVG_N_ORDER; // Ende erreicht? Wieder vorne anfangen!
}

/***** Aktuellen Mittelwert zurückgeben *****/
uint16_t AVG_getAVG(struct floating_avg* avg)
{
    uint8_t index = 0;
    uint32_t sum = 0;

    for(index = 0; index < AVG_N_ORDER; ++index)
    {
        sum += avg->data[index];
    }

    return((uint16_t)(sum / AVG_N_ORDER));
}

/***** LCD *****/
/***** erzeugt den Enable-Puls *****/
void LCD_enable(void)
{
    LCD_PORT |= (1 << LCD_EN); // LCD_ENABLED = 1 an LCD Port senden
    _delay_us(2); // kurze Pause zum verarbeiten des Befehls
    LCD_PORT &= ~(1 << LCD_EN); // LCD_ENABLED = 0 an LCD Port senden
}

/***** Befehl an das LCD senden *****/
void LCD_command(unsigned char temp1)
{
    unsigned char temp2 = temp1;

    LCD_PORT &= ~(1 << LCD_RS); // Register Select auf "Transferring Instruction Data" s
etzen

    temp1 = (temp1 >> 4) & 0x0F; // Oberes Nibble holen + Maskieren
    LCD_PORT = (LCD_PORT & 0xF0) | swap_nibble(temp1); // Daten Bits setzen
    LCD_enable();

    temp2 = temp2 & 0x0F; // unteres Nibble holen und maskieren
    LCD_PORT = (LCD_PORT & 0xF0) | swap_nibble(temp2); // Daten Bits setzen
    LCD_enable();

    _delay_us(42);
}

/***** LCD löschen *****/
void LCD_clear(void)
{
    LCD_command(CLEAR_DISPLAY);
    _delay_ms(2);
}

/***** LCD Cursor Home *****/
void LCD_home(void)
{
    LCD_command(CURSOR_HOME);
    _delay_ms(2);
}

/***** Datenbyte an das LCD senden *****/
void LCD_data(unsigned char temp1)
{
    unsigned char temp2 = temp1;

```

```

LCD_PORT |= (1 << LCD_RS); // Register Select auf "Transferring Display Data" setze
n

temp1 = temp1 >> 4;
temp1 = temp1 & 0x0F;
LCD_PORT = (LCD_PORT & 0xF0) | swap_nibble(temp1); // Daten Bits setzen
LCD_enable();

temp2 = temp2 & 0x0F;
LCD_PORT = (LCD_PORT & 0xF0) | swap_nibble(temp2); // Daten Bits setzen

LCD_enable();
_delay_us(42);
}

/***** LCD initialisieren *****/
void LCD_init(void)
{
    LCD_DDR = LCD_DDR | 0x0F | (1 << LCD_RS) | (1 << LCD_EN); // Port auf Ausgang schal
ten

    _delay_ms(15);
    LCD_PORT = (LCD_PORT & 0xF0) | 0x0C;

    LCD_PORT &= ~(1 << LCD_RS); // Register Select auf "Transferring Instruction Data"
setzen
    LCD_enable();

    _delay_ms(5);
    LCD_enable();

    _delay_ms(1);
    LCD_enable();
    _delay_ms(1);

    // 4 Bit Modus aktivieren
    LCD_PORT = (LCD_PORT & 0xF0) | 0x04;
    LCD_enable();
    _delay_ms(1);

    LCD_command(0x28); // 4Bit / 4 Zeilen / 5x7
    LCD_command(0x0C); // Display ein / Cursor aus / kein Blinken
    LCD_command(0x06); // inkrement / kein Scrollen

    LCD_clear();
}

/***** setzt Cursor in Zeile y (1..4) Spalte x (0..19) *****/
void LCD_set_cursor(uint8_t x, uint8_t y)
{
    uint8_t tmp;

    switch(y)
    {
        case 1: // 1. Zeile
            tmp = 0x80 + 0x00 + x;
            break;

        case 2: // 2. Zeile
            tmp = 0x80 + 0x40 + x;
            break;

        case 3: // 3. Zeile
            tmp = 0x80 + 0x14 + x;
            break;

        case 4: // 4. Zeile

```

```
        tmp = 0x80 + 0x54 + x;
        break;

        default:
            return; // für den Fall einer falschen Zeile
    }

    LCD_command(tmp);
}

/***** schreibt String auf LCD *****/
void LCD_string(const char *data)
{
    while(*data != '\0')
    {
        LCD_data(*data++);
    }
}

/***** schreibt String (Festkommaarithmetik) auf LCD *****/
void LCD_string_fixed_point(char* data, uint8_t start, uint8_t comma, uint8_t frac)
{
    uint8_t index = 0;
    uint8_t flag = FALSE; // Flag für führende Nullen

    // Vorkommastellen mit führender Nullen ausgeben
    for(index = start; index < comma; index++)
    {
        if(flag == 1 || data[index] != '0')
        {
            LCD_data(data[index]);
            flag = TRUE;
        }

        else
        {
            LCD_data('0'); // Leerzeichen
        }
    }

    LCD_data('.'); // Komma ausgeben

    // Nachkommastellen ausgeben
    for(; index < (comma+frac); index++)
    {
        LCD_data(data[index]);
    }
}

/***** zeigt die Zeiten auf dem LCD an *****/
void LCD_showTime(struct myTime* time)
{
    LCD_string(utoa(time->min, buffer, 10)); // Minuten aus der struktur time anzeigen
    LCD_string(":");

    if(time->sec < 10) LCD_string("0");
    LCD_string(utoa(time->sec, buffer, 10)); // Sekunden aus der struktur time anzeigen
    LCD_string(".");

    if(time->csec < 10) LCD_string("0");
    LCD_string(utoa(time->csec, buffer, 10)); // Millisekunden aus der struktur time an
    zeigen
}

/***** zeigt statischen Text auf dem LCD an *****/
void LCD_showStaticText(void)
{
    LCD_home();
}
```



```
LCD_string("AKT:");

LCD_set_cursor(0, 2);
LCD_string("BEST:");

LCD_set_cursor(0, 3);
LCD_string("LAP:");

LCD_set_cursor(0, 4);
LCD_string("RPM:");

LCD_set_cursor(13, 4);
LCD_string("V:");
}

/***** Rundenzahl anzeigen *****/
void LCD_showLaps(uint8_t nLaps)
{
    LCD_set_cursor(8, 3);

    if(nLaps != 0)
    {
        if(nLaps < 100) LCD_data('0');
        if(nLaps < 10) LCD_data('0');

        utoa(nLaps, buffer, 10);
        LCD_string(buffer);
    }

    else
    {
        LCD_string("000");
    }
}

/***** Zwischenzeit anzeigen *****/
void LCD_showInterim(int16_t diff_ticks)
{
    LCD_set_cursor(12, 3);

    if(diff_ticks == 0) LCD_data('=');
    else if(diff_ticks > 0) LCD_data('+');
    else LCD_data('-');

    LCD_showTime(&interim);
}

/***** Rundenzeit anzeigen *****/
void LCD_showLaptime(void)
{
    LCD_set_cursor(13, 1);
    LCD_showTime(&show);

    LCD_set_cursor(13, 2);
    LCD_showTime(&best);
}

/***** Drehzahl anzeigen *****/
void LCD_showRPM(uint8_t pre_rpm)
{
    if(pre_rpm != 0)
    {
        LCD_set_cursor(6, 4);

        if(pre_rpm < 100) LCD_data(' ');
        if(pre_rpm < 10) LCD_data(' ');

        utoa(pre_rpm, buffer, 10);
    }
}
```

```

        LCD_string(buffer);
        LCD_string("00");
    }

    else
    {
        LCD_set_cursor(6, 4);
        LCD_string("  0");
    }
}

/***** Geschwindigkeit anzeigen *****/
void LCD_showKPH(uint8_t kph)
{
    if(kph != 0)
    {
        LCD_set_cursor(17, 4);

        if(kph < 100) LCD_data(' ');
        if(kph < 10) LCD_data(' ');

        utoa(kph, buffer, 10);
        LCD_string(buffer);
    }

    else
    {
        LCD_set_cursor(17, 4);
        LCD_string(" 0");
    }
}

/***** Menü anzeigen *****/
void LCD_showMenu(struct conf* config)
{
    LCD_set_cursor(1, 1);
    LCD_string("Magnetschleifen");
    LCD_set_cursor(19, 1);
    LCD_string(utoa(config->amount_magnetic_loops, buffer, 10));

    LCD_set_cursor(1, 2);
    LCD_string("Reifenumfang");
    LCD_set_cursor(16, 2);
    LCD_string_fixed_point(my_utoa(config->perimeter_tyre, buffer), 0, 1, 2);

    LCD_set_cursor(1, 3);
    LCD_string("Ritzel");
    LCD_set_cursor(17, 3);
    if(config->front_sprocket < 10) LCD_string(" ");
    else if(config->front_sprocket < 100) LCD_data(' ');
    LCD_string(utoa(config->front_sprocket, buffer, 10));

    LCD_set_cursor(1, 4);
    LCD_string("Kettenrad");
    LCD_set_cursor(17, 4);
    if(config->rear_sprocket < 10) LCD_string(" ");
    else if(config->rear_sprocket < 100) LCD_data(' ');
    LCD_string(utoa(config->rear_sprocket, buffer, 10));
}

/***** Hintergrundbeleuchtung dimmen *****/
void LCD_backgrnd(uint8_t brightness)
{
    // value = 0    Hintergrundbeleuchtung aus
    // value = 255 Hintergrundbeleuchtung komplett an
    if(brightness == 0)
    {
        TCCR2 &= ~(1 << CS21); // Timer stoppen
    }
}

```

```
    PORTB &= ~(1 << PB1); // Hintergrundbeleuchtung ausschalten
}

else if(brightness == 255)
{
    TCCR2 &= ~(1 << CS21); // Timer stoppen
    PORTB |= (1 << PB1); // Hintergrundbeleuchtung einschalten
}

else
{
    TCCR2 &= ~(1 << CS21); // Timer stoppen

    // creates a block of code that is guaranteed
    // to be executed atomically
    ATOMIC_BLOCK(ATOMIC_FORCEON)
    {
        OCR2 = brightness; // Hintergrundbeleuchtung dimmen
    }

    TCCR2 |= (1 << CS21); // Timer wieder starten (Prescaler = 8)
}
}

/***** Hintergrundbeleuchtung dimmen *****/
void LCD_backgrnd_fade(uint8_t in_out)
{
    // in_out = 1  Hintergrundbeleuchtung reinfaden
    // in_out = 0  Hintergrundbeleuchtung rausfaden
    if(in_out == TRUE)
    {
        for(uint8_t n = 0; n < 255; n++)
        {
            LCD_backgrnd(n);
            _delay_ms(5);
        }

        LCD_backgrnd(255);
    }

    else
    {
        for(uint8_t n = 255; n > 0; n--)
        {
            LCD_backgrnd(n);
            _delay_ms(5);
        }

        LCD_backgrnd(0);
    }
}

/***** Schreiben auf EEPROM kenntlich machen *****/
void LCD_indicateWrite(void)
{
    for(uint8_t index = 0; index < 3; index++)
    {
        LED_besttime(ON);
        _delay_ms(500);
        LED_besttime(OFF);
        _delay_ms(500);
        LCD_data('.');
    }
}

/***** Initialisiert die Member der Struktur *****/
void initStruct(struct myTime* temp)
{
```

```

// Alles mit Null Initialisieren
temp->csec = 0;
temp->sec = 0;
temp->min = 0;
temp->ticks = 0;
for(uint8_t index = 0; index < AMOUNT_MAGNETIC_LOOPS; index++) temp->interim[index]
= 0;
}

/***** Zeit aufteilen *****/
void splitTicks(uint16_t ticks, volatile struct myTime* result)
{
    uint16_t centiSec = 0;
    uint16_t seconds = 0;

    centiSec = ticks; // Zenti-Sekunden (10ms)
    seconds = centiSec / 100; // Sekunden berechnen

    result->ticks = ticks;
    result->csec = centiSec % 100; // Rest bei der Berechnung der Sekunden --> Zenti-Se
kunden
    result->min = seconds / 60; // Minuten berechnen
    result->sec = seconds % 60; // Rest bei der Berechnung der Minuten -- > Sekunden
}

/***** Zeitdifferenz in Drehzahl (U/min) umrechnen *****/
uint8_t calcRPM(uint16_t diff_ticks)
{
    // Drehzahl von 100 bis 24000 U/min begrenzt
    // auf 8-bit (1...240)
    uint8_t pre_rpm = 0;

    // Auflösung: 100 U/min
    if(diff_ticks > LIMIT_MIN_TICKS_RPM || diff_ticks == 0) // Drehzahl < 100 U/min
    {
        return(0); // nicht relevant
    }

    else if(diff_ticks < LIMIT_MAX_TICKS_RPM) // Drehzahl > 24000 U/min
    {
        return(240); // Maximum ausgeben: 24000 U/min
    }

    else
    {
        pre_rpm = (6000 / diff_ticks) * (N_STROKE_ENGINE / 2);
    }

    return(pre_rpm);
}

/***** Zeitdifferenz in Geschwindigkeit (km/h) umrechnen *****/
uint8_t calcKPH(uint16_t diff_ticks)
{
    uint8_t kph = 0;

    // Auflösung: 1 km/h
    if(diff_ticks > LIMIT_MIN_TICKS_KPH || diff_ticks == 0) // Geschwindigkeit < 1 km/h
oder undefiniert
    {
        return(0); // nicht relevant
    }

    else if(diff_ticks < LIMIT_MAX_TICKS_KPH) // Geschwindigkeit > 240 km/h
    {
        return(240); // Maximum ausgeben: 240 km/h
    }
}

```

```

else
{
    kph = (float)((KPH_TO_MPS * PERIMETER_TYRE * 10000) / diff_ticks);
}

return(kph);
}

/***** Bestzeit gefahren -> LED an *****/
void LED_besttime(uint8_t on_off)
{
    // on_off = 1 LED für Bestzeit einschalten
    // on_off = 0 LED für Bestzeit ausschalten
    if(on_off == 1)
    {
        flag_besttime = 1; // Flag für Bestzeit gefahren setzen
        LED_PORT &= ~(1 << LED_BESTTIME); // LED für Bestzeit einschalten
    }
    else
    {
        LED_PORT |= (1 << LED_BESTTIME); // LED für Bestzeit ausschalten
    }
}

/***** Timer 0-2 *****/
/***** Initialisiert den Timer0 *****/
void initTimer0(void)
{
    TCCR0 = (1 << CS01); // Vorteiler 8 -> 4MHz / 8 = 500kHz Timer Frequenz
    TIMSK |= (1 << TOIE0); // Overflow Interrupt einschalten
    TCNT0 = -50; // Timer Register wird mit einem Wert vorbelegt, 2^8 - 206 = 50
}

/***** Initialisiert den Timer1 *****/
void initTimer1(void)
{
    TCCR1A = 0; // No Compare Output Mode or Waveform Generation
    TCCR1B = (1 << ICNC1) | (1 << CS11); // Noise Canceler an, Fallende Flanke, Vorteil
er 8 -> 4MHz / 8 = 500kHz Timer Frequenz
    TIMSK |= (1 << TICIE1) | (1 << TOIE1); // Input Capture Interrupt einschalten, Over
flow Interrupt einschalten
    TCNT1 = -5000; // Timer Register wird mit einem Wert vorbelegt, 2^16 - 60536 = 5000
}

/***** Initialisiert den Timer2 *****/
void initTimer2(void)
{
    TCCR2 = (1 << WGM21) | (1 << WGM20) | (1 << CS21); // Fast-PWM mode, OC2 disconnect
ed, prescaler = 8
    TIMSK |= (1 << TOIE2) | (1 << OCIE2); // Overflow + Compare match interrupt enable
    ASSR &= ~(1 << AS2); // asynchronon mode deactivated
    TCNT2 = 0;
}

/***** Initialisiert die externen Interrupts *****/
void initExtInt(void)
{
    MCUCR &= ~(1 << ISC10) | (1 << ISC00);
    MCUCR |= (1 << ISC11) | (1 << ISC01); // INTO + INT1: Aufruf der ISR bei fallender
Flanke
    GICR |= (1 << INT0) | (1 << INT1); // Interrupt (INT0 + INT1) aktivieren
}

/***** vier Bit vertauschen für LCD_data *****/
unsigned char swap_nibble(unsigned char in)

```

```

{
    return((in & (1 << 0)) << 3) | ((in & (1 << 1)) << 1) | ((in & (1 << 2)) >> 1) | ((
in & (1 << 3)) >> 3);
}

/***** Menu *****/
/***** init. menu buttons *****/
void MENU_control_init(void)
{
    DDRB |= 0x02; // define all pins with buttons as input
    PORTB |= 0x1C; // activate internal pull-ups
}

/***** get the state of the menu buttons *****/
void MENU_control(void)
{
    uint8_t pastTime = 0;

    // ***** OK/EXIT *****
    if(MENU_debounce_input(&PINB, BUTTON_OK_EXIT))
    {
        uint8_t entry = 1;

        LCD_clear();
        LCD_showMenu(&config);

        LCD_set_cursor(0, entry);
        LCD_data('>');

        while(TRUE)
        {
            // ***** UP *****
            if(MENU_debounce_input(&PINB, BUTTON_UP))
            {
                entry = MENU_direction(CURSOR_UP);
            }

            // ***** DOWN *****
            else if(MENU_debounce_input(&PINB, BUTTON_DOWN))
            {
                entry = MENU_direction(CURSOR_DOWN);
            }

            pastTime = timeButtonPressed(&PINB, BUTTON_OK_EXIT, 60);

            if(pastTime != 0 && pastTime < 10)
            {
                switch(entry)
                {
                    case 1: // Magnetschleife
                        MENU_editValue(19, 1, 1, 6, 1, &config.amount_magnetic_loops, F
ALSE);
                        break;

                    case 2: // Reifenumfang
                        MENU_editValue(16, 2, 1, 250, 3, &config.perimeter_tyre, TRUE);
                        break;

                    case 3: // Ritzel
                        MENU_editValue(18, 3, 0, 25, 2, &config.front_sprocket, FALSE);
                        break;

                    case 4: // Kettenrad
                        MENU_editValue(17, 4, 0, 100, 3, &config.rear_sprocket, FALSE);
                        break;

                    default:

```

```

        break;
    }
}

else if(pastTime >= 60)
{
    break;
}

_delay_ms(25);
}

// Daten ins EEPROM schreiben
LCD_clear();
LCD_home();
LCD_string("Save to EEPROM");
LCD_indicateWrite();

#ifdef ACCESS_EEPROM
    EEPROM_writeConfig(&config);
#endif

// Standardanzeige anzeigen
LCD_clear();
LCD_showStaticText();
}

// Helligkeit der LCD-Hintergrundbeleuchtung einstellen
// ***** UP *****
else if(MENU_debounce_input(&PINB, BUTTON_UP))
{
    brightness = OCR2;

    if(brightness < 235)
    {
        brightness = brightness + 20;
        LCD_backgrnd(brightness);
    }
}

// ***** DOWN *****
else if(MENU_debounce_input(&PINB, BUTTON_DOWN))
{
    brightness = OCR2;

    if(brightness > 20)
    {
        brightness = brightness - 20;
        LCD_backgrnd(brightness);
    }
}
}

/***** edit value at given position *****/
void MENU_editValue(uint8_t posX, uint8_t posY, uint8_t minValue, uint8_t maxValue, uint8_t digits, uint8_t* value, uint8_t isFloat)
{
    while(!MENU_debounce_input(&PINB, BUTTON_OK_EXIT))
    {
        LCD_set_cursor(posX, posY);

        if(isFloat != TRUE)
        {
            if((digits > 1) && (*value < 10))
            {
                LCD_data(' ');
            }
        }
    }
}

```

```
        if((digits > 2) && (*value < 100))
        {
            LCD_data(' ');
        }

        LCD_string(utoa(*value, buffer, 10));
    }

else
{
    LCD_string_fixed_point(my_utoa(*value, buffer), 0, 1, 2);
}

LCD_set_cursor(posX+digits+isFloat-1, posY);
LCD_command(LCD_CURSOR_ON);

// ***** UP *****
if(MENU_debounce_input(&PINB, BUTTON_UP))
{
    if(*value < maxValue)
    {
        *value = *value + 1;
    }
}

// ***** DOWN *****
else if(MENU_debounce_input(&PINB, BUTTON_DOWN))
{
    if(*value > minValue)
    {
        *value = *value - 1;
    }
}

    _delay_ms(25);
}

LCD_command(LCD_CURSOR_OFF);
}

/***** debounce menu buttons input *****/
uint8_t MENU_debounce_input(volatile uint8_t *port, uint8_t pin)
{
    static uint8_t flag = 0;
    uint8_t i = 0;

    if(flag)
    {
        // check for key release
        while(TRUE)
        {
            // until key pressed
            if(!(*port & (1 << pin)))
            {
                i = 0; // 0 = bounce
                break;
            }

            _delay_us(500); // 128ms

            if(--i == 0) // until key >128ms released
            {
                flag = 0; // clear press flag
                i = 0; // 0 = key release debounced
                break;
            }
        }
    }
}
```



```
else
{
    // else check for key press
    while(TRUE)
    {
        if((*port & (1 << pin)))
        {
            // until key released
            i = 0; // 0 = bounce
            break;
        }

        _delay_us(500); // 128ms

        if(--i == 0)
        {
            // until key >128ms pressed
            flag = 1; // set press flag
            i = 1; // 1 = key press debounced
            break;
        }
    }

    return(i);
}

/***** scroll through the menu *****/
uint8_t MENU_direction(int8_t direction)
{
    static uint8_t index = 1;

    // clear cursor symbol at the old position
    LCD_set_cursor(0, index);
    LCD_data(' ');

    // over the bottom?
    if(direction == CURSOR_DOWN && index < 4)
    {
        index++;
    }

    // over the top?
    else if(direction == CURSOR_UP && index > 1)
    {
        index--;
    }

    // set cursor at the new position
    LCD_set_cursor(0, index);
    LCD_data('>');

    return(index);
}

/***** Umwandlung uint <> string mit führenden Nullen *****/
char *my_utoa(uint8_t zahl, char *string)
{
    int8_t index = 0;

    string[3] = '\0'; // String Terminator

    for(index = 2; index >= 0; index--)
    {
        string[index] = (zahl % 10) + '0'; // Modulo rechnen, dann den ASCII-Code von '
0' addieren
        zahl /= 10;
    }
}
```

```
    }

    return(string);
}

/***** Langes drücken des Tasters erkennen *****/
uint8_t timeButtonPressed(volatile uint8_t *port, uint8_t pin, uint8_t limit)
{
    uint8_t counter = 0;

    if(!(*port & (1 << pin))) // pressed
    {
        while(TRUE)
        {
            _delay_ms(50);

            // released
            if(*port & (1 << pin)) break;
            else counter++;
            if(counter > limit) break;
        }
    }

    return(counter);
}
```