Informatikwerkstatt, Foliensatz 6
Motorsteuerung

G. Kemnitz

Institut für Informatik, TU Clausthal (IW-F6)

4. Januar 2016

Inhalt Foliensatz 6: Motoransteuerung

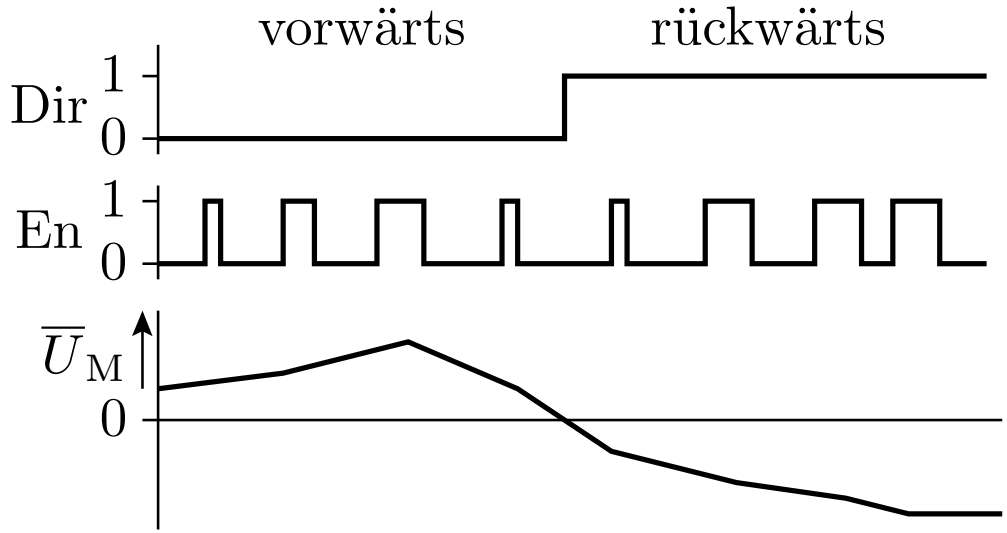
Drehzahlsteuerung

Prinzip und Motortest

Drehzahlsteuerung durch Pulsweitenmodulation

Pulsweitenmodulation (PWM) schaltet die Motoren schnell ein und aus. Drehzahlsteuerung über die relative Einschaltzeit.

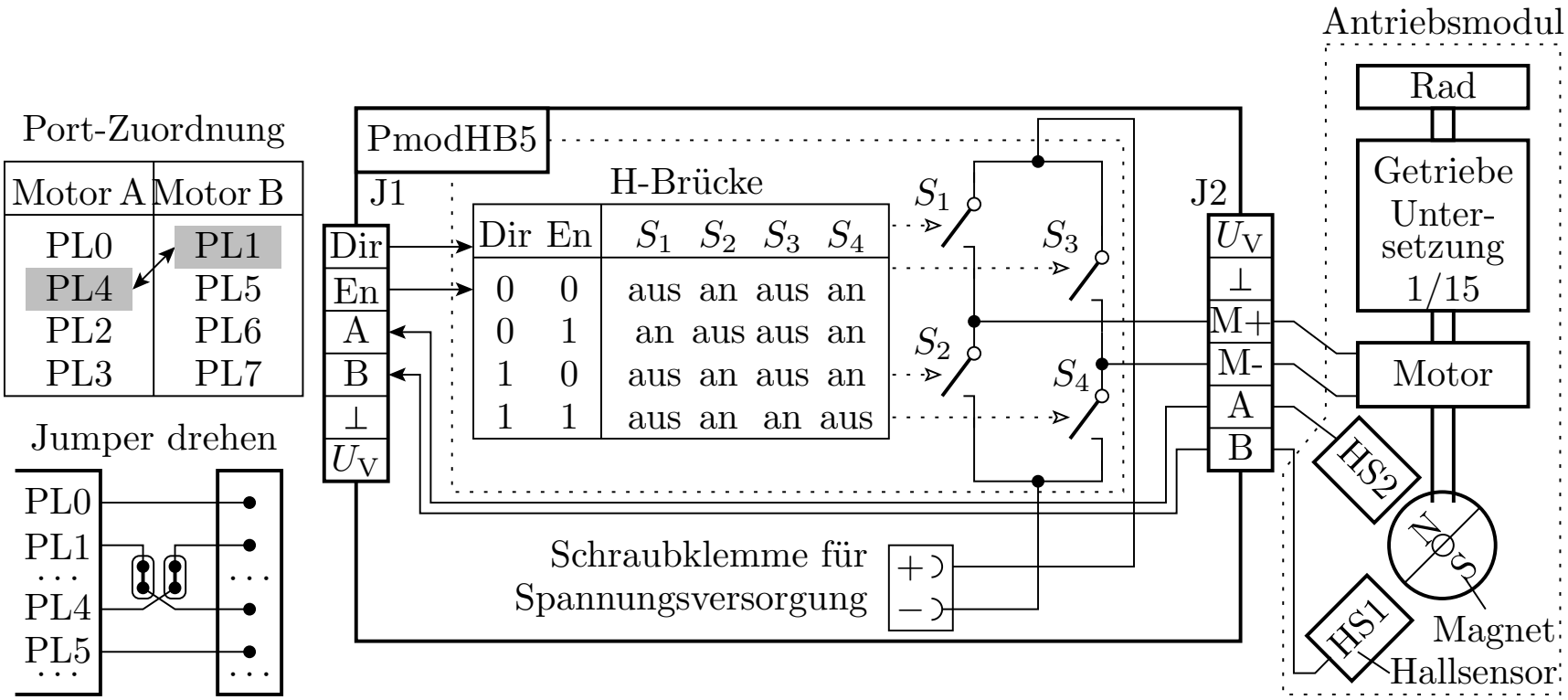
En	1	0	0	1
Dir	0	0	1	1
	+ M −	M	− M +	



An den Antriebsbaugruppen erfolgt die

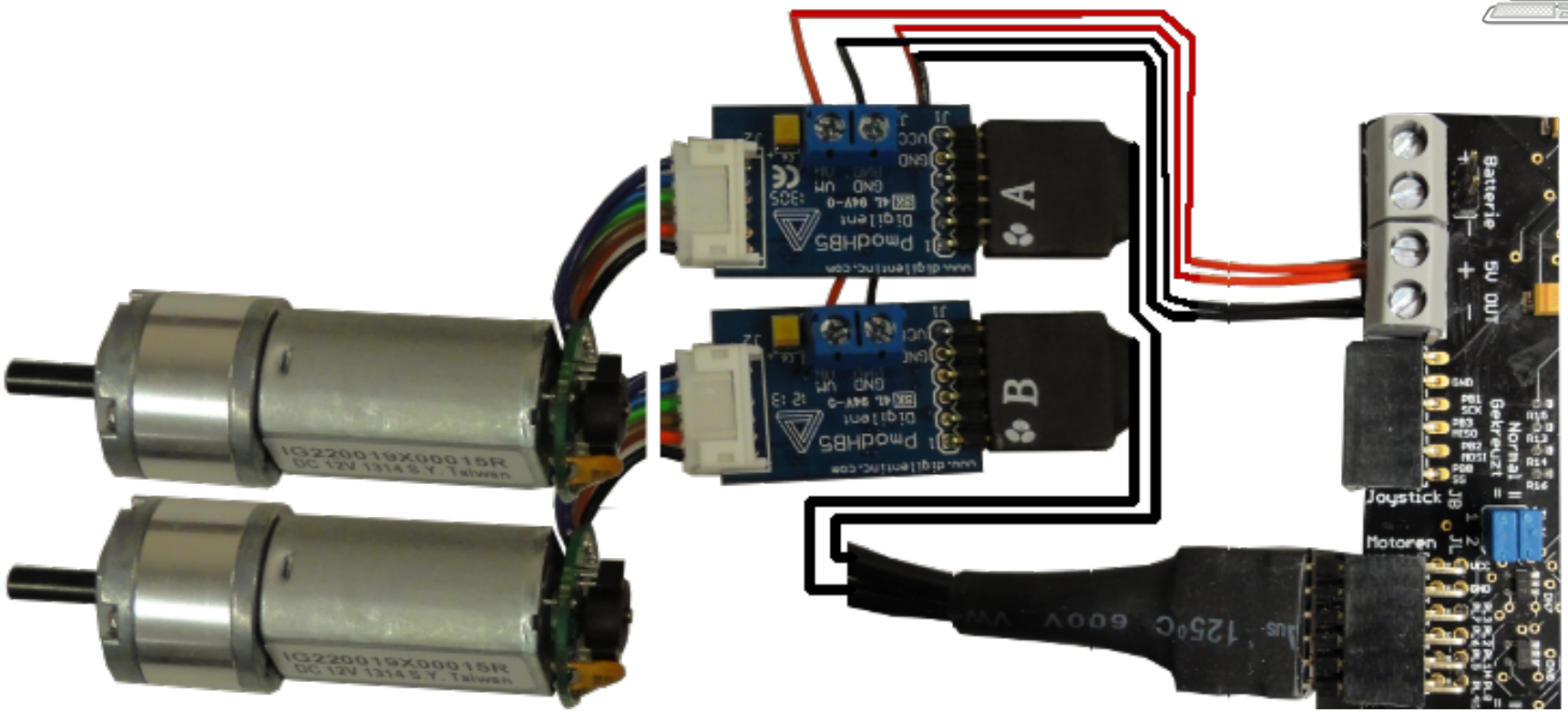
- Einstellung der Drehrichtung über ein Richtungsbit Dir und
- die Einstellung der relative Pulsbreite mit dem En- (Enable-) Signal.

Anschluss der Motoren an den Mikrorechner



- Antriebsmodule: Motor, Untersetzungsgetriebe, rotierender Magnet + Hallsensoren zum Zählen der Winkelschritte.
- PmodHB5: H-Brücke, angesteuert über Dir und En + Rückgabe der Hallsensorsignal an den Mikrorechner.

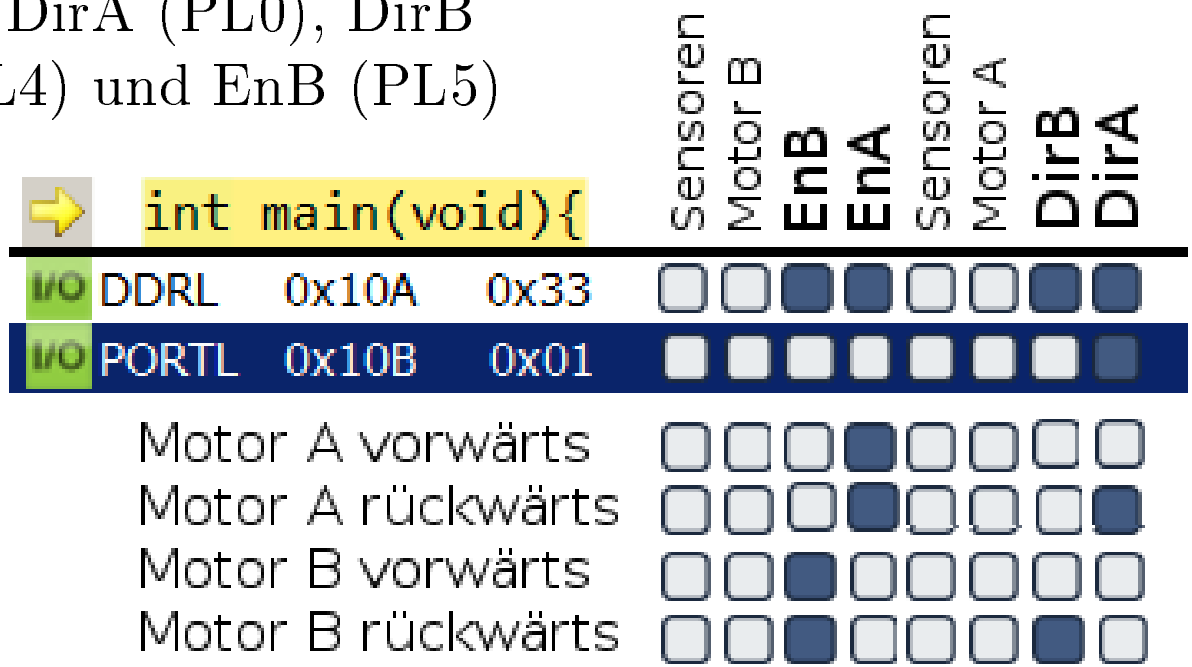
Praktischer Aufbau



- 2×H-Brücke PmodHB5 über Y-Kabel an JL,
- Motoren an die H-Brücken stecken,
- JLX »gekreuzt (=)« (Pin-Tausch PL0 und PL4),
- Spannungsversorgungdrähte zuschneiden und anschrauben.

Motoren ausprobieren

- Beliebiges Projekt im Debugger starten  . Anhalten.
- I/O > PORTL aufklappen.
- Zum Motortest DirA (PL0), DirB (PL1), EnA (PL4) und EnB (PL5) auf Ausgang und Ausgabewerte setzen.

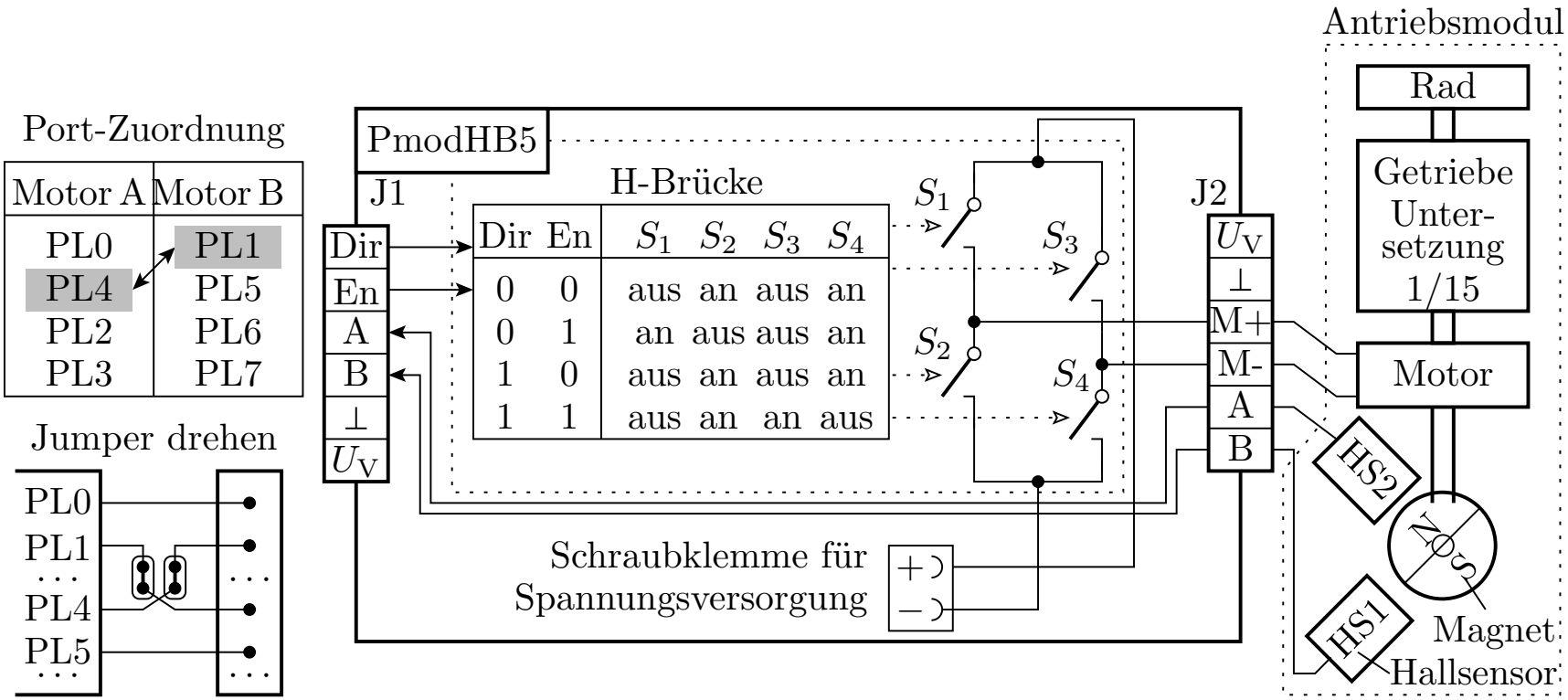


- Motoren vor- und rückwärts drehen lassen.
- Kontrolle der Sensorausgaben mit Multimeter[1].

[1] Die Anzeige von »PINA« wird nur bei Programm-Start-Stop aktualisiert.

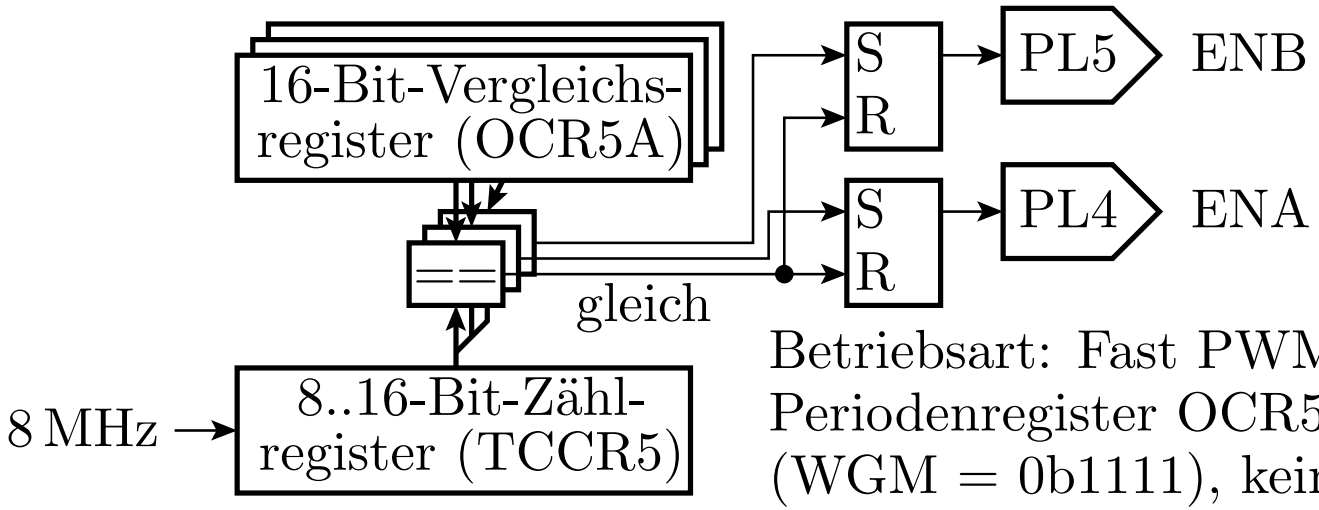
Treiber »pwm«

Treiber »pwm« für die Drehzahlsteuerung

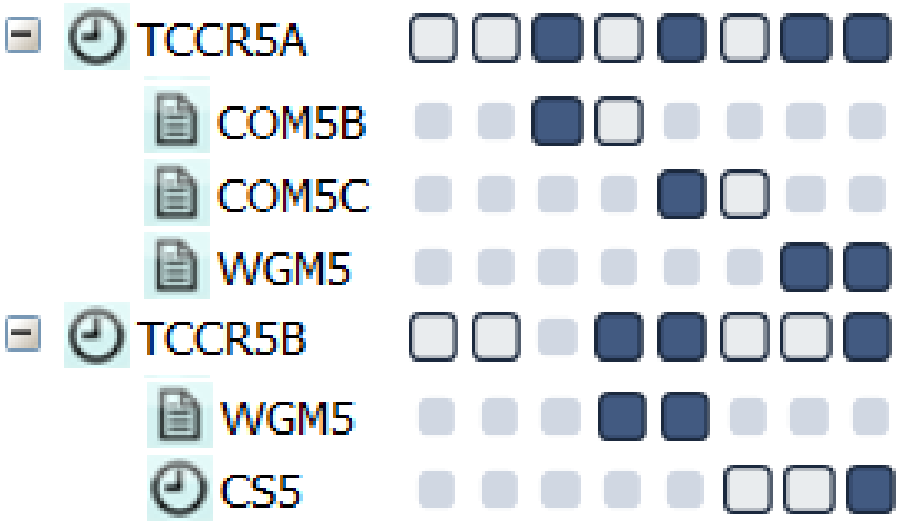


- Der Treiber erwartet die dargestellte Hardware und erzeugt die Dir- und En-Signale für beide Motoren.
- Die gepulsten En-Signale generiert Timer 5 im PWM-Mode ohne ISR an PL40 und PL5.

Timer-Einstellung für die Enable-Signale



Betriebsart: Fast PWM
Periodenregister OCR5A
(WGM = 0b1111), kein
Taktvorteiler (CS=0b001)
PWM-Ausgabe ”Clear on
Compare” (COM5B/C=0b10)



- PWM-Treiber initialisieren:

```
void pwm_init(){
  DDRL  =0b00110011;   //En und Dir als Ausgänge
  pwm_stop();          //Zähltakt und PWM aus ...
  TCCR5C = 0b00000000; //Zählregister löschen
  OCR5A = 0x2000;      //Periodenregister (ca. 1 ms)
  OCR5B = 0;           //Motor A: Pulsbreite 0
} OCR5C = 0;           //Motor B: Pulsbreite 0
```

- Zähltakt und PWM-Ausgabe aus:

```
void pwm_stop(){
  TCCR5A = 0;   //PWM ausschalten
  TCCR5B = 0;   //Zähltakt aus
} PORTL  = 0;   //Enable (Motoren) ausschalten
```

- Zähltakt und PWM-Ausgabe ein:

```
void pwm_start(){
  TCCR5A = 0b00101011; //COM5B/C=0b10 (PWM-Ausg. ein)
} TCCR5B = 0b00011001; //WGM=0b1111 CS=0b001 (Takt ein)
```

Pulsbreite vorgeben

```
void pwm_set_A(int16_t pwm){
  if (pwm>=0){
    OCR5B =pwm;
    PORTL |=1;          //Dir-Bit (PL0) setzen
  }
  else{
    OCR5B = -pwm;
    PORTL &= ~1;        //Dir-Bit (PL0) löschen
  }
}
```

- Der Geschwindigkeitswert ist 16-Bit vorzeichenbehaftet.
- Bei Betragswerten größer Periodenwert bleibt das Freigabesignal dauerhaft an.
- In der Funktion für Motor B

```
void pwm_set_B(int16_t pwm);
```

Ersatz »OCR5B« durch »OCR5C« und »PL0« durch »PL1.

Treibertest

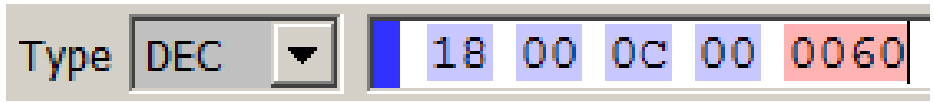
Treiber »pwm« ausprobieren

- PmodUSBUART an JH oben und USB-Verbindung zum PC.
- JHX und JLX auf »gekreuzt (=)«.
- Projekt »F6-1_test_pwm\test_pwm« Übersetzen. Starten.
- HTerm starten. 8N1 9600 Baud. Verbinden.

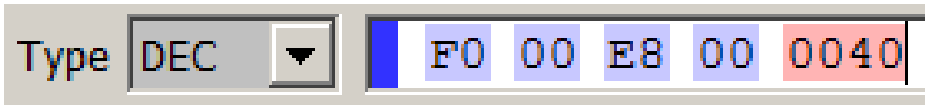
Testbeispiele:

- Motoren A und B mit 50% für 3 s vorwärts:

- Motor A mit 75% und Motor B mit 3/8 für 6 s vorwärts:



- Motor A mit 50% und Motor B 75% rückwärts für 4 s.



Erstellung weiterer Testbeispiele

- Die Motoren werden mit 6-Byte-Nachrichten $B_0 B_1 \ldots B_5$ (B_i – Byte i) angesteuert.
  - Byte B_0 und B_1 definieren die reltive Pulsbreite Motor A:

$$\eta_{\mathrm{A}} = \begin{cases} 1 & B_0 \geq 0\mathrm{x}20 \\ \frac{|16 \cdot B_0 + B_1|}{0\mathrm{x}2000} & B_0 < 0\mathrm{x}20 \end{cases}$$

  - Byte B_2 und B_3 definieren die reltive Pulsbreite Motor B:

$$\eta_{\mathrm{B}} = \begin{cases} 1 & B_2 \geq 0\mathrm{x}20 \\ \frac{|16 \cdot B_2 + B_3|}{0\mathrm{x}2000} & B_2 < 0\mathrm{x}20 \end{cases}$$

  - Byte B_4 und B_5, auch zusammen als Dezimalzahl eingebbar, definieren die Bewegungsdauer:

$$t = \frac{16 \cdot B_3 + B_4}{10}\,\mathrm{s}$$

Das Testprogramm

Das Testbeispiel nutzt außer »pwm.h« folgende Treiber:

```
#include "comir_pc.h"  //PC-Eingabe
#include "comir_tmr.h" //Bewegungsdauer
```

In »comir_pc.h« sind die Puffergrößen geändert auf:

```
//Empfangs- und Sendepuffergröße in »comir_pc.h«
#define COM_PC_RMSG_LEN  6 //Empfangsnachricht 6 Byte
#define COM_PC_SMSG_LEN  0 //keine Sendenachricht
```

Das Hauptprogramm beginnt mit Variablenvereinbarungen, Initialisierung der Treiber und globaler Interruptfreigabe:

```
uint8_t msg[COM_PC_RMSG_LEN];
int main(void){
  int16_t pwm; uint16_t time;
  com_pc_init();           //PC-Kommunikations-, PWM- und
  pwm_init(); tmr_init(); //Timer-Treiber initialisieren
  sei();                   //Interrupts global ein
```

- In der Enlosschleife wird auf eine 6-Byte-Nachricht gewartet.
- Wenn sie eintrifft, wird der PWM-Werte gesetzt, der Timer und die Bewegung gestartet.

```
while(1){
 if (com_pc_get(msg)){ //wenn neue Nachricht
   pwm = msg[0]<<8 | msg[1];
   pwm_set_A(pwm);      //PWM-Wert für Motor A
   pwm = msg[2]<<8 | msg[3];
   pwm_set_B(pwm);      //PWM-Wert für Motor B
   time = msg[4]<<8 | msg[5];
   tmr_start(time, 0); //Timer Kanal 0 starten
   pwm_start();         //PWM (Motoren) starten
  }
```

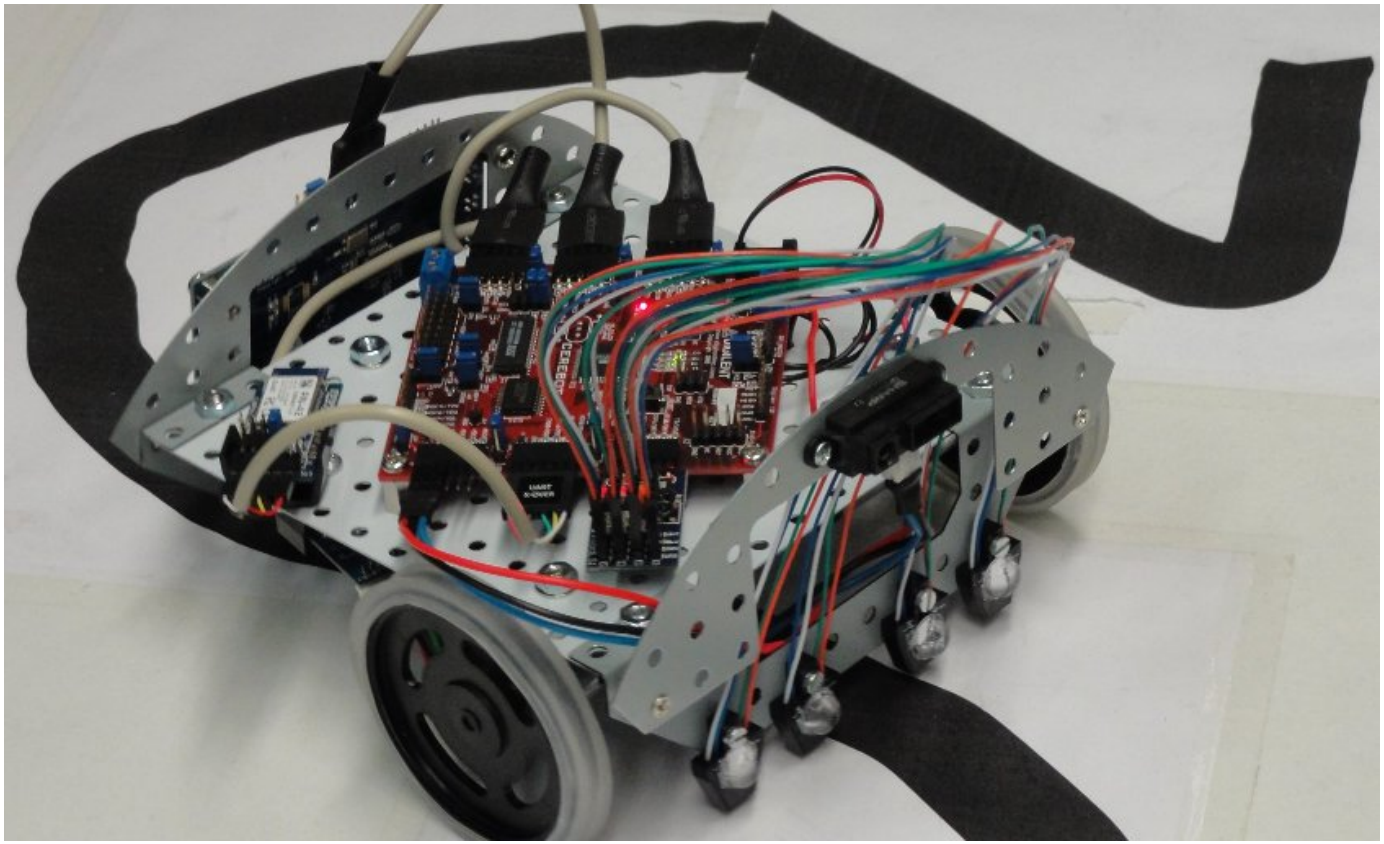
Wenn Timer abgelaufen, PWM aus, Motoren angehalten:

```
   if (!tmr_restzeit(0))//wenn Timer abgelaufen
    pwm_stop();         //PWM und Motoren aus
 }
}
```

Winkelmessung

Messprinzip

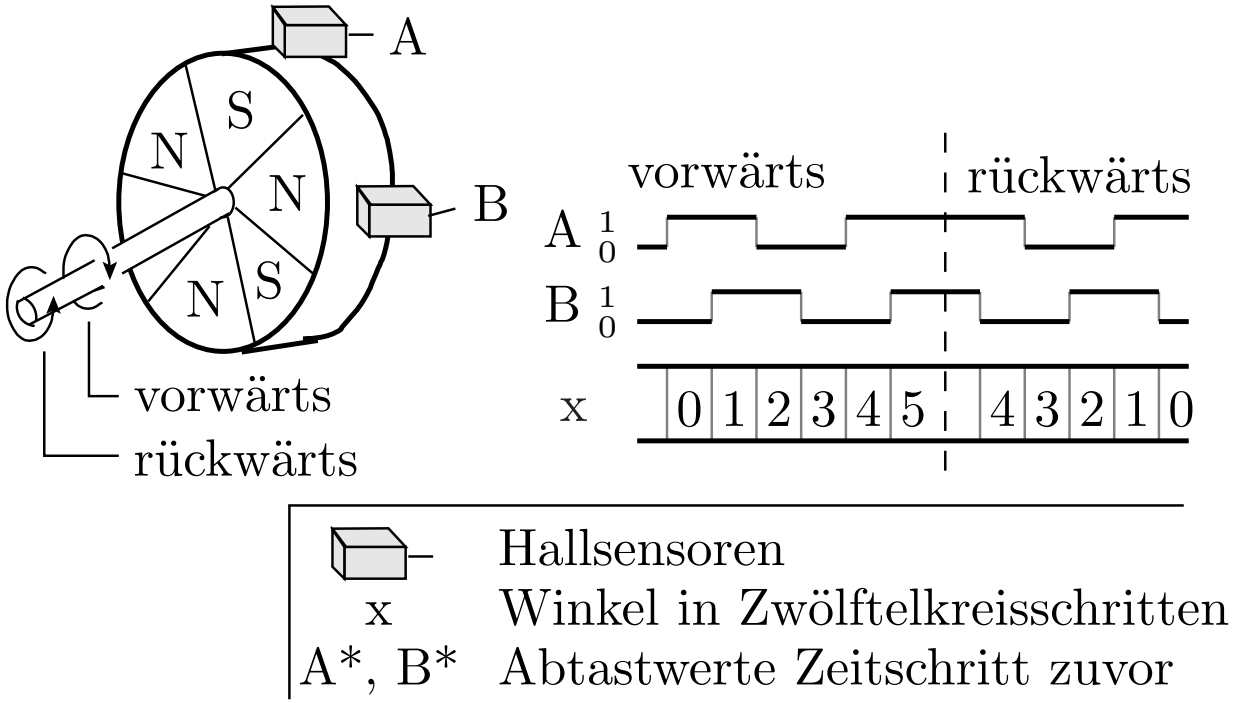
Drehwinkel und Fahrzeugposition



Zählen der Winkelschritte an beiden Antriebsrädern. Erweiterbar zu einer 2D+R-Positionsbestimmung[2].

[2] »2D-Position und Richtung relativ zur Startposition und Ausrichtung.

Winkelschrittzähler



A*	B*	A	B	x
0	0	0	0	—
0	0	0	1	−1
0	0	1	0	+1
0	1	0	1	—
0	1	0	0	+1
0	1	1	1	−1
1	0	1	0	—
1	0	1	1	+1
1	0	0	0	−1
1	1	1	1	—
1	1	1	0	−1
1	1	0	1	+1

- Auflösung 1/12 Motorumdrehungen. Eine Radumdrehung sind 20 Motorumdrehungen. Max. 5 Radumdrehungen / s.
- Die Sensorbitwerte müssen mindestens einmal je Winkelschritt gelesen und verarbeitet werden ($\geq 1200\,\mathrm{s}^{-1}$).

Geschwindigkeit- und 2D-R-Positionsbestimmung

Geschwindigkeit: Winkelschritte für eine bestimmte Zeit zählen.

2D+R-Position: Für jeden Auswerteschritt der Sensorbits 9 Fälle unterscheiden:

Δw_R	0	-1	+1	0	-1	+1	0	-1	+1
Δw_L	0	0	0	-1	-1	-1	+1	+1	+1
Δs	0	$-s_0$	s_0	$-s_0$	$-2s_0$	0	s_0	0	$2s_0$
$\Delta\alpha$	0	α_0	$-\alpha_0$	$-\alpha_0$	0	$-2\alpha_0$	α_0	$2\alpha_0$	0

α – Bewegungsrichtung; Δs – Schrittweite in Richtung α;
$s_0 = \frac{\pi \cdot d}{24}$ – Basisschrittweite und $\alpha_0 = \tan\left(\frac{s}{a}\right)$ – Winkelschritt
Auto für ein Rad-Winkel-Increment. ...

$\Rightarrow$ Aufgabe für Programmier- und Mathe-Experten.

Treiber »rotmess«

Der Treiber »rotmess«

Bestimmt die Anzahl der Winkelschritte für ein Zeitintervall, hier 1 s, für später einen Regelerschritt (20 ms).

Zählen der Winkelschritte für beide Räder in einer zyklisch aller 0,5 ms gestarteten ISR (Timer 1):

- Increment eines Zeitzählers.
- Einlesen der Sensorbitwerte A und B für beide Räder.
- Aus diesen und den vorhergehenden Sensorbitwerten Berechnung der Drehwinkeländerungen $\Delta w \in \{-1, 0, +1\}$.
- Summierung der Δw für je 1 s (2.000 Schritte[3] zu je 0,5 ms).
- Danach werden ein Ereignisbit gesetzt, die Schrittzähler gelöscht und die Zählwerte gesichert.

Eine get- Funktion liest und löscht die gesicherten Zählwerte.

[3] Im Header »rotmess.h« einstellbar. Später Schrittzeit des Reglers.

Private Daten und Initialisierung

```
int16_t Ct_T;        //Zeitzähler
int16_t Ct_A, Ct_B; //Geschwindigkeitszähler
int16_t speed_A, speed_B;//Geschwindigkeitswerte
int8_t sens_A, sens_B;   //Bit(3:2) neue und Bit(1:0)
                         //alte Sensorwerte
int8_t rotmess_err_ct;   //Fehlerzähler, nur Debug
int8_t new_dat;      //0 keine neuen Daten, 1 neue Daten
```

Initialisierungsfunktion:

- Timer 0, CTC-Mode, 0,5 ms Periode, OCR0A-Interrupt[4]:

```
void rotmess_init(){
//Timer 0 für OCR0A-Interupts alle 0.5 ms einrichten
  TCCR0A  = 0b10;      //WGM = 0b010 (CTC Mode mit OCR0A)
  TCCR0B  = 0b011;     //CS  = 0b011 (Vorteiler 64)
  OCR0A   = 62;        //OCR = (0,5 ms*8MHz)/(2*8)-1
  TIMSK0 |= 1<<OCIE0A;//Vergleichs-Interrupt freigeben
```

[4] 8-Bit-Timer mit weniger Konfigurationsmöglichkeiten.

- Sensorzustand initialisieren. Zähler löschen:

```
  sens_A = (PINL>>4) & 0b1100;// Startwerte der Hall-
  sens_B =  PINL     & 0b1100;// sensoren lesen
  clear_counter();    // Zähler löschen
}
```

Löschfunktion für die Zähler:

```
void clear_counter(){
  Ct_A = 0; Ct_B  = 0; // Winkelschritt- und
  Ct_T = 0;            // Zeitzähler löschen
}
```

Die ISR setzt für beide Motoren die aktuellen und vorherigen Sensorwerte zu einem 4-Bit-Vektor zusammen, ...

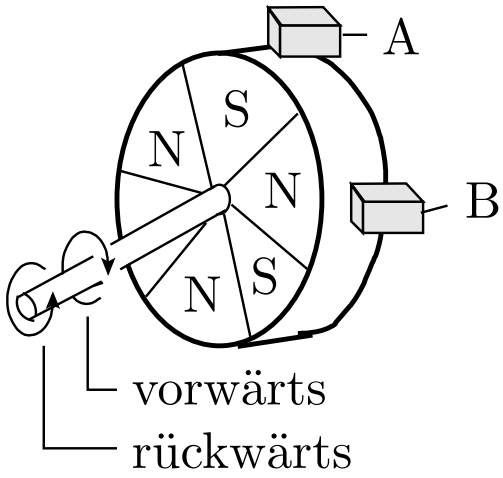
```
ISR(TIMER0_COMPA_vect){
  sens_A = (sens_A>>2) | ((PINL>>4) & 0b1100);
  sens_B = (sens_B>>2) |  (PINL & 0b1100);
```

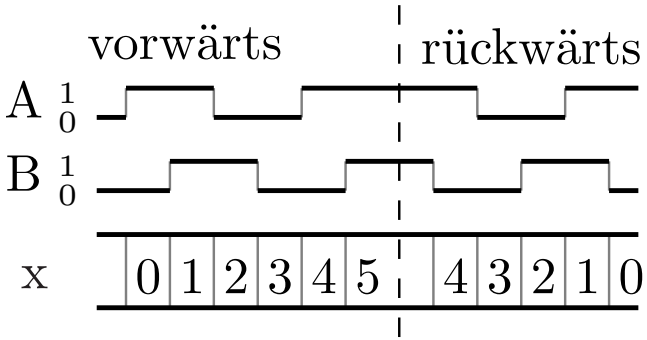
bestimmt mit einer Funktion QuadEnc() den Winkel-Increment (WB: $-1, 0, +1$), zählt die Zeit weiter, ...

```
    Ct_A += QuadEnc(sens_A);
    Ct_B += QuadEnc(sens_B);
    Ct_T++;
    if (Ct_T>=ABTASTSCHRITTE){
      speed_A = Ct_A;
      speed_B = Ct_B;
      err_out = err_ct;
      new_dat=1;
      clear_counter();
    }
}
```

Nach einer als Konstante definierten Anzahl von Abtastschritten werden die Zählwerte gespeichert, ein Flag »neue Daten« gesetzt« und die Zähler gelöscht.

Bestimmung der Winkelbewegung





A*	B*	A	B	x
0	0	0	0	—
0	0	0	1	−1
0	0	1	0	+1
0	1	0	1	—
0	1	0	0	+1
0	1	1	1	−1
1	0	1	0	—
1	0	1	1	+1
1	0	0	0	−1
1	1	1	1	—
1	1	1	0	−1
1	1	0	1	+1

```
int8_t  QuadEnc(uint8_t nsdat){
 switch (sensdat){
  case  0b0010:
  case  0b0100:
  case  0b1011:
  case  0b1101:
  return  -1;

  case  0b0001:
  case  0b0111:
  case  0b1000:
  case  0b1110:
  return  1;

  case  0b0011:
  case  0b0110:
  case  0b1100:
  case  0b1001:
  if  (err_ct<0xFF)
    err_ct++;
  }
  return  0;
}
```

- Funktion zum Lesen der gemessen Winkelschritte:

```
uint8_t rotmess_get(int16_t *spA, int16_t *spB,
                    uint8_t *CtE){
  if (new_dat){                  //wenn neue Daten
    uint8_t tmp = TIMSK0;        //ISR, die dieselben Daten
    TIMSK0 &= ~(1<<OCIE0A); //bearbeitet sperren
    *spA = speed_A;              //Ergebnisse kopieren
    *spB = speed_B;
    *CtE = err_out;
    TIMSK0 = tmp;          //Interrupt wieder freigeben
    new_dat = 0;           //neue-Daten-Flag löschen
    return 1;              //Rückkehr mit "neue Daten"
  }                        //sonst
  return 0;                //Rückkehr ohne neue Daten
}
```

Zur Fehlerbehandlung im übergeordneten Modul gibt es nur eine Abfragefunktion, ob Abtastfehler im Abfrageintervall aufgetreten sind. Der interne Fehlerzähler ist nur im Debug-Modus zugänglich:

```
uint8_t rotmess_err(){  //Fehlerabfrage
  if (rotmess_err_ct){  //wenn Fehler aufgetreten sind
    rotmess_err_ct=0;   //Fehlerzähler löschen
    return 1;           //Rückkehr mit 1 (wahr)
  }                     //sonst
  return 0;             //Rückkehr mit 0 (falsch)
}
```

Das Testprogramm »test_rotmess«

Das Testprogramm bindet außer »rotmess« folgende Treiber ein:

```
#include "comir_pc.h" //PC-Eingabe und -ausgabe
#include "pwm.h"      //Geschwindigkeitssteuerung
```

Vom PC wird auf ein 6-Byte-Datenpaket gewartet, die Motoren bewegt und ein 8-Byte-Paket zurückgesendet (in »comir_pc.h«):

```
#define COM_PC_RMSG_LEN  6
#define COM_PC_SMSG_LEN  8
```

Sendedaten:

- Byte 1 und 2: Pulslänge Motor A (OCR5B),
- Byte 3 und 4: Pulslänge Motor B (OCR5C),
- Byte 1 und 2: Pulsperiode Motor A und B (OCR5A).

Zurückgesendete Bytes:

- Byte 1, 2, 5 und 6: empfangene Bytes 1 bis 4,
- Byte 3 und 4: Winkelschritt pro s Motor A,
- Byte 7 und 8: Winkelschritt pro s Motor B.

Variablen des Hauptprogramms:

```
uint8_t rmsg[COM_PC_RMSG_LEN];
uint8_t smsg[COM_PC_SMSG_LEN];
int main(){
  int16_t speed_A, speed_B, pwm;
  uint8_t state=0;  //Programmzustand
```

Treiberinitialisierung, globale Interrupt-Freigabe:

```
  rotmess_init();   //initialisieren aller Treiber
  com_pc_init();
  pwm_init();
  sei();            //Interrupts einschalten
  while(1){ ...  }
```

Das Hauptprogramm ist ein Zustandsautomat:

- Zustand 0: Warte auf 6-Byte-Nachricht vom PC,
- Zustand 1 und 2: Bewegung ohne Messung,
- Zustand 3: Bewegung mit Messung der Winkelschritte,
- Zustand 4: Messergebnisse zum PC senden.

Im Zustand 0 wird auf eine 6-Byte-Nachricht vom PC gewartet. Falls keine da ist, wird die PWM angehalten:

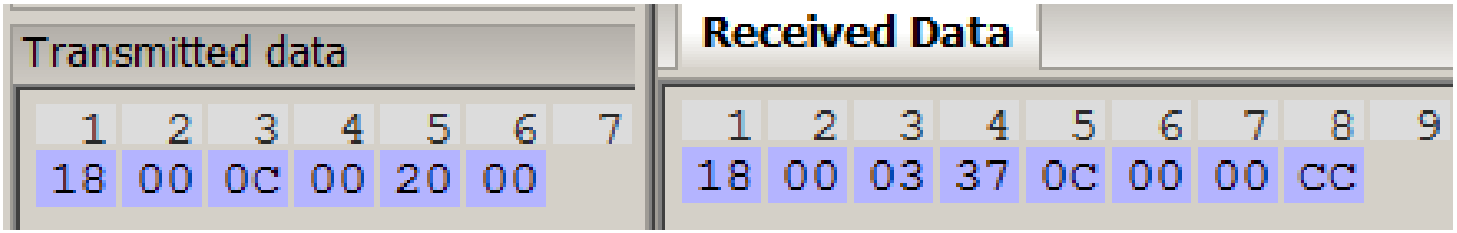
```
if (state == 0){
  if (com_pc_get(rmsg)){// wenn neue Nachricht
    pwm = rmsg[0]<<8 | rmsg[1];
    pwm_set_A(pwm);   // Wert für Motor A einstellen
    pwm = rmsg[2]<<8 | rmsg[3];
    pwm_set_B(pwm);   // Wert für Motor B einstellen
    OCR5A = rmsg[4]<<8 | rmsg[5];
    pwm_start();          state = 1;
  }
  else
    pwm_stop();
}
```

In den Zuständen 1 bis 3 passiert nur etwas, wenn neue Winkelmessdaten bereit sind, d.h. alle 1 s. In Zustand 2 und 3 soll sich eine konstanten Geschwindigkeit einstellen. Im Zustand 4 werden die PWM-Vorgaben und Zählwerte zum PC gesendet und der Zustand auf null zurückgesetzt:

```
if (state && rotmess_get(&speed_A, &speed_B, &ErrCt)){
    state++;                  //nach jeder Messung Zustand++
    if (state>3){             //3. Messergebnis zum PC senden
      smsg[0] = rmsg[0];    smsg[1] = rmsg[1];
      smsg[2] = speed_A>>8; smsg[3] = speed_A & 0xff;
      smsg[4] = rmsg[2];    smsg[5] = rmsg[3];
      smsg[6] = speed_B>>8; smsg[7] = speed_B & 0xff;
      com_pc_send(smsg);
      state=0;
    }
  }
```

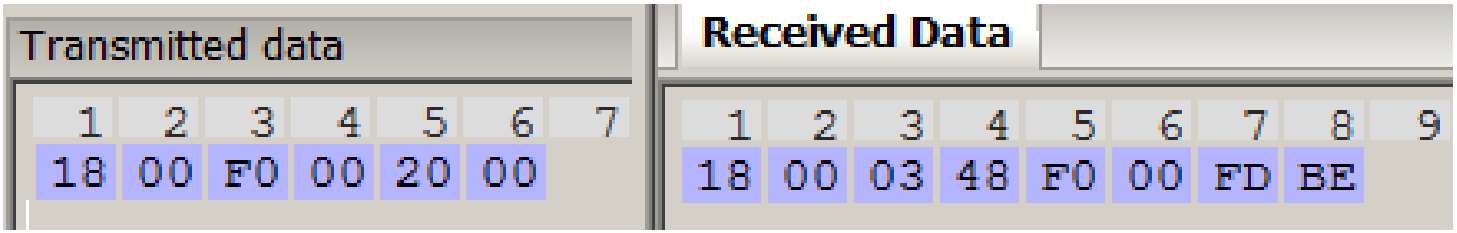
Testbeispiele mit HTerm

Beispiel1:



PWM_A	speed_A	PWM_B	speed_B
$\frac{0x1800}{0x2000} = 75\%$	$\frac{0x337}{240} = 3{,}43\,\frac{U}{s}$	$\frac{0x0C00}{0x2000} = 37{,}5\%$	$\frac{0x0CC}{240} = 0{,}85\,\frac{U}{s}$

Beispiel 2:



PWM_A	speed_A	PWM_B	speed_B
$\frac{0x1800}{0x2000} = 75\%$	$\frac{0x348}{240} = 3{,}63\,\frac{U}{s}$	$\frac{-0x1000}{0x2000} = -50\%$	$\frac{-0x242}{240} = -2{,}41\,\frac{U}{s}$

Absolute Pulsweite in den Beispielen :$0x2000/8\,\text{MHz} \approx 1\,\text{ms}$

Motorkennlinie

Bestimmung der Motorkennlinien

Für die Konzeption der Fahrzeugsteuerung wird die Funktion

$$v = f(\eta, \ldots)$$

(v – Fahrzeuggeschwindigkeit; η – relative Pulsweite; ... – weitere Einflüsse wie Pulsperiode, Versorgungsspannung, ...) benötigt. Bestimmbar mit HTerm und vielen Einzelmessungen.

Alternative: Programmgesteuert mit Python-Programm.

Mit den Funktionen write() und read() aus dem dem Python-Modul »serial« lassen sich nur Zeichenketten senden und empfangen. Auf den nächsten Folien wird deshalb zuerst ein Python-Modul mit Konvertierfunktionen Zahlen und Zeichenketten entwickelt.

Konvertierfunktionen zwischen Zahlen und Zeichenketten »conf_bstring.py«

```
#Umrechnung eines Integers in eine Bytefolge
def int2bstr(wert, byteanz):
  bstr = '';                #leere Zeichenkette
  for i in range(byteanz):
    bstr = chr(wert & 0xFF)+bstr;
    wert = wert>>8;
  return bstr;

#Umrechnung einer Bytefolge in ein Integer
def bstr2int(bstr):
  wert = 0
  for c in bstr:
    wert = (wert<<8) + ord(c);
  if (wert>0) and ord(bstr[0])>0x7F:
    wert=wert- (1<<(8*len(bstr)))
  return wert
```

Vorteile der Auslagerung in ein seperates Modul:

- in unterschieliche Python-Programme importierbar,
- separat testbar.

In Python kann man den Testrahmen unter die Funktionen schreiben und dort lassen[5]:

```
if __name__ == '__main__':
 for (w, a) in ((34,2),(-3456,2), (-18567,3), (48000,2)):
  bs = int2bstr(w, a)
  print 'w:%6i'%w,'a:%i'%a,'| w==bstr2int(bs):',w==bstr2int(bs
```

Im Beispiel wird getestet, dass die in einen Bytestring umgewandelten Zahlen wieder in dieselbe Zahl zurückgewandelt werden.

Ausgabe:

```
H: ...\Python>conf_bstring.py
w:    34 a:2 | w==bstr2int(bs): True
w: -3456 a:2 | w==bstr2int(bs): True
w:-18567 a:3 | w==bstr2int(bs): True
w: 48000 a:2 | w==bstr2int(bs): False
```

Letzten Beispiel Wertebereichsüberlauf ($48000 > 2^{15} - 1$).

[5]Der Test-Code wird nur ausgeführt, wenn das Modul selbst gestartet, statt impotiert wird.

PC-Programm »rotmess.py«

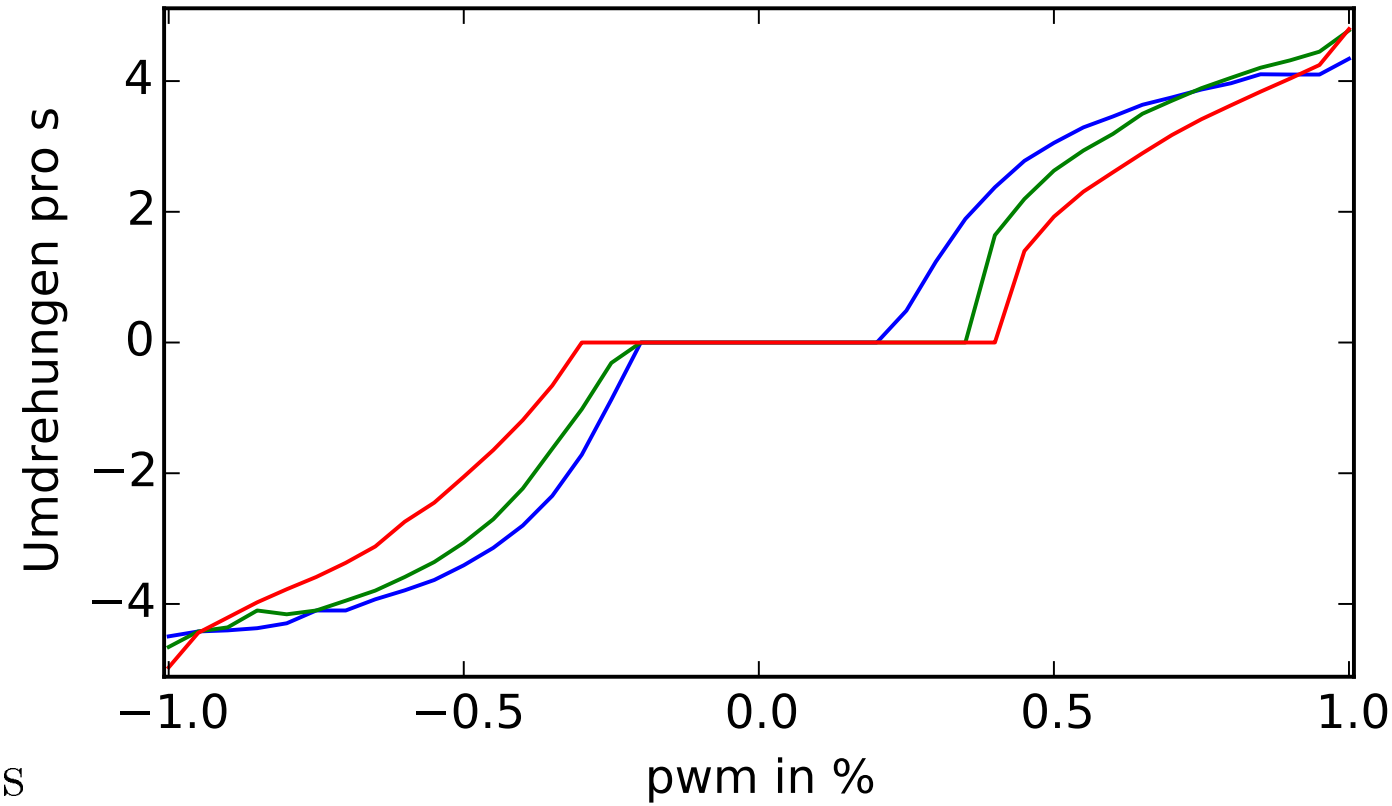
```
wiederhole für PWM-Periode ∈ {2 ms, 1 ms, 0,5 ms}
  wiederhole für pwm=-100% bis 100% in 5%-Schritten
    bestimme Motordrehzahl
    Ausgabe der Werte als Tabelle
    Sammeln der Werte von Motor A für eine Graphik
```

```
from conf_bstring import *       #Konv.-fkt. importieren
import serial                    #Modul serial importieren
ser = serial.Serial("COM9")      #COM anpassen!
import matplotlib.pyplot as plt #Plotfunktion importieren
for Periode in (0x4000,0x2000,0x1000): #Periodenwerte
  pwm_list=[]                    #leere Listen für die
  speed_list=[]                  #graphisch darzust. Werte
  pwm=-1.0
  while pwm<=1.01:               #für pwm = -1 bis 1
```

```
        ocr = int(pwm*Periode)  #%-Zahl => OCR-Wert
        sstr = 2 * int2bstr(ocr,2) + int2bstr(Periode, 2)
        ser.write(sstr)         #senden von 6 Bytes
        rstr=ser.read(8)        #auf 8 Bytes warten
        #Bytes in Ergebnisswerte umrechnen und ausgeben
        pwmA = float(bstr2int(rstr[0:2]))/Periode * 100
        spA  = float(bstr2int(rstr[2:4]))/240
        pwmB = float(bstr2int(rstr[4:6]))/Periode * 100
        spB  = float(bstr2int(rstr[6:8]))/240
        print 'Periode = %4.2fms'%(Periode/8E3),
        print 'pwmA =%4.1f'%pwmA + '%', 'spA = %4.2f U/s'%spA
        #PWM-Wert und Ergebnisse Motor A an Listen hängen
        pwm_list.append(pwm)
        speed_list.append(spA)
        pwm +=0.05              #Erhöhung um 5% Schritten
    #Liste als xy-Graph ausgeben, einen je Periode
    plt.plot(pwm_list, speed_list)
ser.close()                     #serielle Verb. schliessen
plt.xlabel('pwm in %')          #Achsen beschriften
plt.ylabel('Umdrehungen pro s') plt.show()
```

Ergebnis

```
Periode = 2.05ms pwmA =-100.0% spA = -4.85 U/s
Periode = 2.05ms pwmA = -95.0% spA = -4.70 U/s
Periode = 2.05ms pwmA = -90.0% spA = -4.63 U/s
```



Periode:
rot 2 ms
grün 1 ms
blau 0,5 ms

- Die Kennlinie ist nichtlinear mit einem Totbereich zwischen ca. -25% bis +25%.
- Vernünftig steuern lässt sich der Motor nur im Betragsbereich von 2 bis 4 Radumdrehungen pro Sekunde.
- Langsame und genaue Bewegungsvorgaben verlangen eine Regelung.
- PWM-Periode ca. 1 ms ist ein vernüftiger Wert.

Motorregelung

Motorregelung

Regelung:

- Subtraktion des Ist-Werts (Position, Geschwindigkeit, ...) vom Soll-Wert.
- Die Differenz ist die Regelerabweichung.
- Berechnung der neuen Stellgröße so, dass die Regelerabweichung gegen null strebt.

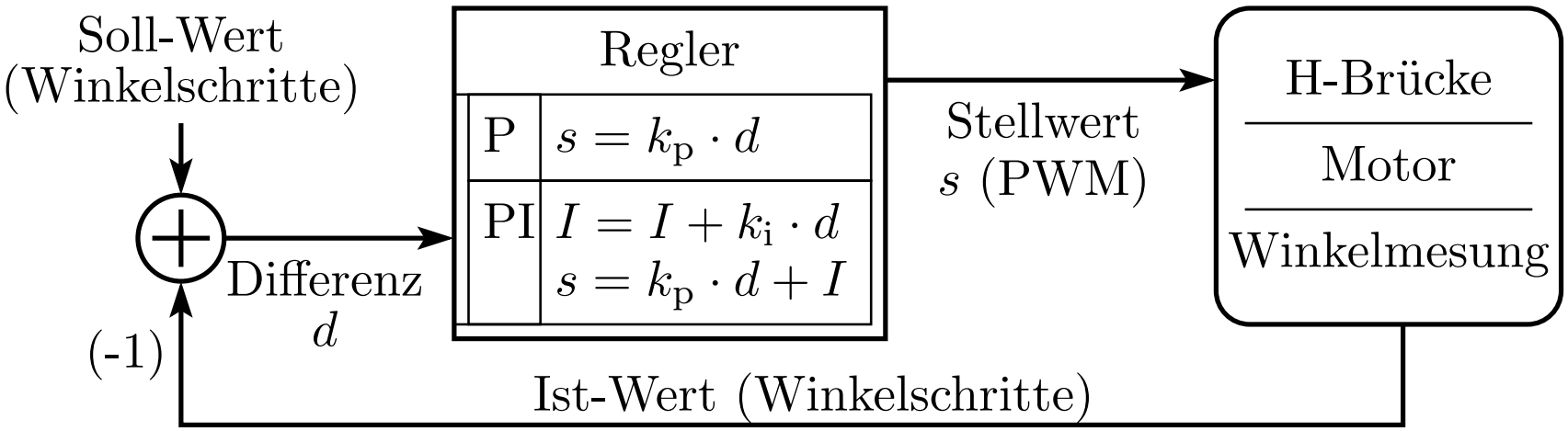
Für das Fahrzeug wird

- ein PI-Positionsregler,
- mit einem aus der Soll-Geschwindigkeit berechneten sich stetig ändernden Soll-Wert
- empirisch (durch geschicktes Probieren) entworfen.

WEB-Suchbegriff:»PI-Regler empirisch einstellen«

PI-Regler

Funktion von P- und PI-Reglern



- Bestimmung des PWM-Stellwerts aus der Soll-/Ist-Differenz der Winkelschritte.
- Je kleiner k_p desto größer die Soll-/Ist-Abweichung.
- Bei zu großem k_p schwingt die Regelung.
- Mit einem zusätzlichen Integralanteil I und passendem k_i verringert sich die Soll-/Ist-Abweichung auf nahe null.
- k_p und k_i experimentell bestimmbar.

Die beiden Reglergleichungen:

$$\begin{aligned} I &= I + k_\mathrm{I} \cdot d \\ s &= k_\mathrm{P} \cdot d + I \end{aligned}$$

- sind mit 8 NKB (Nachkommabits) zu programmieren:

```
uint16_t kp, ki;          //Reglerkoeffizient, 8 NKB
int32_t diffA, diffB;     //Schrittdifferenz, 8 NKB
int32_t integA, integB;   //Integralanteile, 8 NKB
int32_t pwmA, pwmB;       //Stellwerte, 8 NKB
int16_t speedA, speedB;   //Sollgeschwindigkeit, 8 NKB
```

- Ausschluss Überlauf: 0x7FFFFF.FF $\Leftrightarrow -$0x100000:

```
void limit(int32_t *val, int32_t max){
  if (*val >  max)
    *val = max;
  else if (*val < -max)
    *val = -max;
}
```

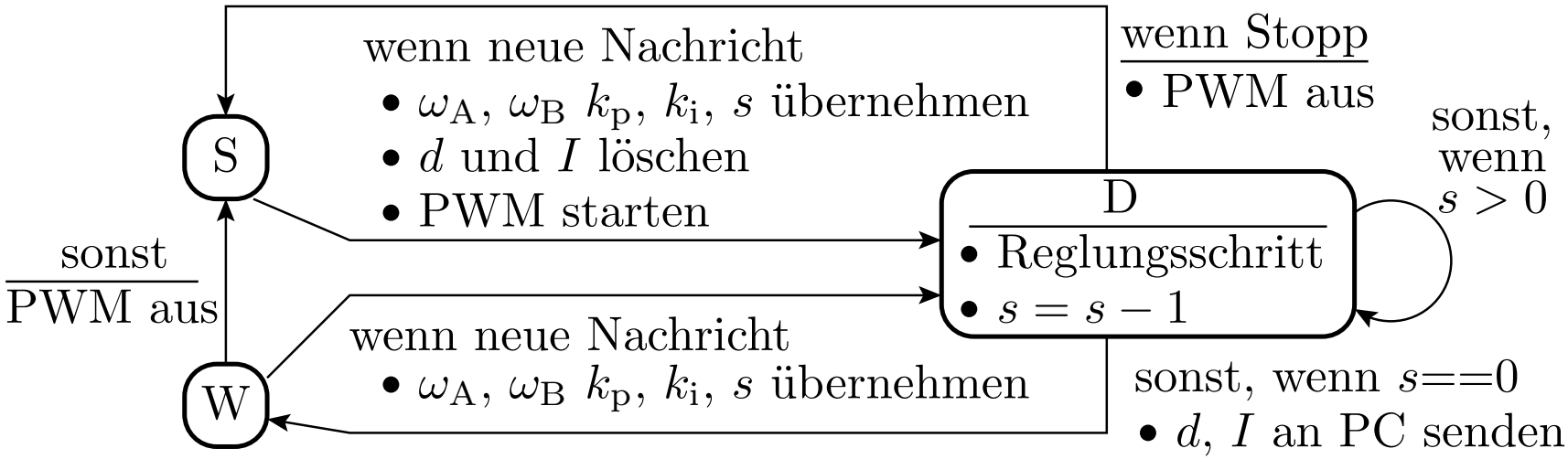
- Kommastellenkorrektur für Produkte mit Shift-Operationen.

Programm für einen Regelungsschrittes

```
if (rotmess_get(&sa, &sb)){ //wenn neuer Messwert
  if (rotmess_err())        //bei Fehler im Treiber
    lcd_incErr(ERR_RMESS);  //Fehlerzähler erhöhen
  //Ausführung eines PID-Reglerschritts je Motor
  diffA  = diffA + speedA - (sa << 8);
  limit(&diffA, 0x100000);  //Begrenzung der Differenz
  integA = integA + ((ki * diffA)>>8);
  limit(&integA, 0x400000); //Begrenzung des Integralteils
  pwmA = (integA>>8) + ((kp * diffA)>>12);
  limit(&pwmA, 0x4000);     //Begrenzung der Stellgrösse
  pwm_set_A(pwmA);
  ... //dasselbe für Motor B
  ... //Ausgabe Regelabweichungen und Integralteile
  ... //zur Kontrolle auf das LC-Display
}
```

Regelungsprogramm

Antriebssteuerung als Automat

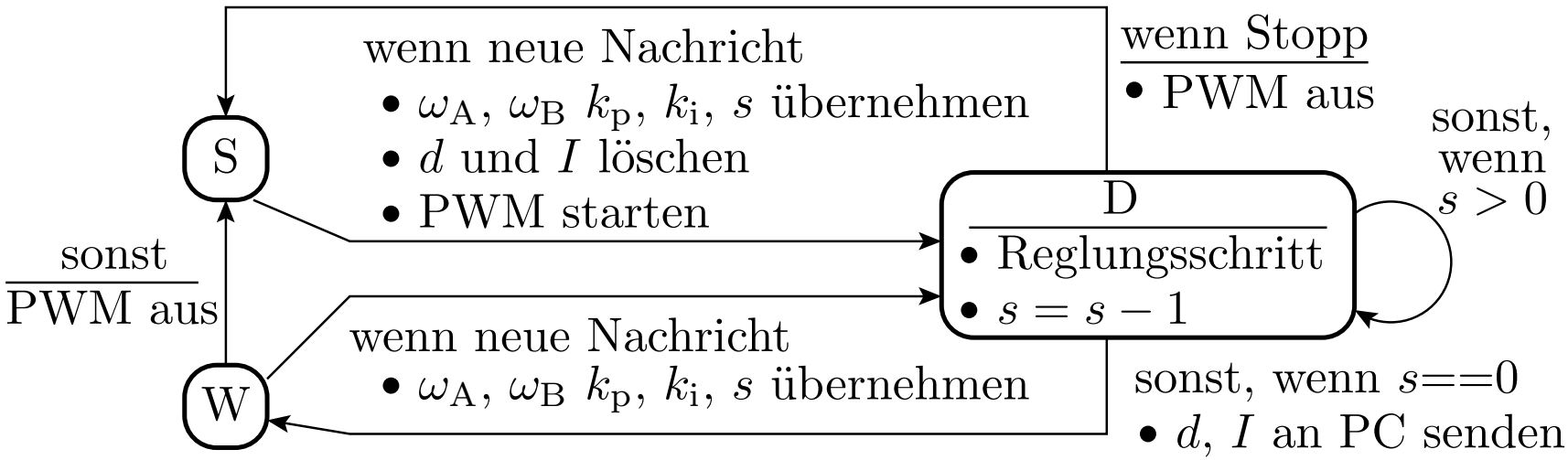


Automatenzustände:

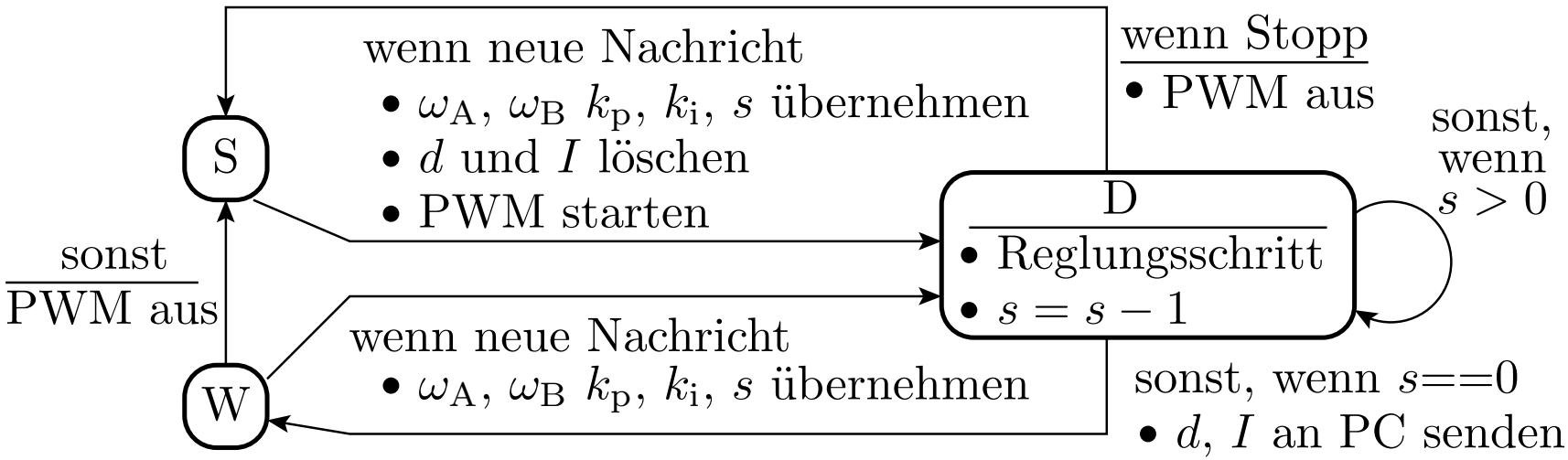
S	Antriebe gestoppt
D	Abarbeitung einer Teilbewegung
W	Übergang zur nächsten Teilbewegung ohne Stopp

Nachrichtenbestandteile / Beschreibung einer Teilbewegung:

ω_A, ω_B	Soll- Winkelgeschwindigkeit
k_p, k_i	Reglerparamerter
d, I, s	Differenzen (Regelabweichung), Integralteile, Schrittzahl



- Wenn nach Ablauf der Schrittzahl eine neue Nachricht da ist, wird die Bewegung ohne Zwischenhalt fortgesetzt.
- Sonst oder, wenn die Stopptaste betätigt wird, bricht die Bewegung ab und starte mit der nächsten Nachricht neu.
- Nach Abarbeitung einer Teilbewegung werden die Differenzen und und Intgralteile an den PC gesendet.
- Ist-Positionen und -Geschwindigkeiten für die graphische Darstellung ergeben sich aus den Sollwerten und Differenzen.



- Für eine flüssige Bewegung ohne »Stopp« muss der PC die Folgenachricht vor der Anwort auf die aktuelle Nachricht absenden.

Ausgabe auf dem LCD-Monitor:

- aktuelle Soll/Ist-Abweichungen, Integralanteile,
- Zähler für Bewegungsstopps,
- Automatenzustand und
- Fehlerzähler (Empfangs-Timeout, Sendversagen, Winkelmessfehler, »falsche Interrupts«).

Konstanten zur Definition der LCD-Ausgabe


```
#define INITSTR "A:xxx xxx T:xx .B:xxx xxx E:...."
#define LCP_DIFFA    2      //Differenz Motor A
#define LCP_INTEGA   6      //Integralteil Motor A
#define LCP_STPCT   12      //Zähler "Bewegungsstopps"
#define LCP_STATE   15      //Zustand des Testprogramms
#define LCP_DIFFB   18      //Differenz Motor B
#define LCP_INTEGB  22      //Integralteil Motor B
#define LCP_ERR     28      //Beginn Fehlerzähler
#define ERR_SEND    28      //Zähler Sendversagen
#define ERR_ETO     29      //Zähler Empfangs-Timeout
#define ERR_WMESS   30      //FZ Winkelmessung
//Zeichen 31 ist der Zähler falsche Interrupts
```

Hardware-Konfiguration und Treiber

Hardware-Konfiguration:

- Taster- oder Schalter-Modul an JA (Not-Aus)
- LC-Display an JD (LCD Monitor), JDX »gekreuzt (=)«.
- H-Brücken mit Motoren an JL, JLX »gekreuzt (=)«.
- PModUSBUSART an JH und PC, JHX »gekreuzt (=)«.

Treiber:

```
#include "pwm.h"        //Motorsteuerung ueber PWM
#include "rotmess.h"    //Messung der Rotationsschritte
#include "comir_pc.h"   //Kommunikation mit dem PC
#include "comir_lcd.h" //LCD-Kontrollmonitor
```

Einstellung im Headern »rotmess.h« 20 ms Messzeit:

```
#define ABTASTSCHRITTE 40   //40*0,5ms = 20ms
```

Einstellung im Header »comir_pc.h« :

```
#define COM_PC_RMSG_LEN 10  //Empfangsbytes
#define COM_PC_SMSG_LEN  8  //Sendebytes
```

Private globale Daten:

```
uint8_t stop_ct=1;           //Zähler Bewegungsstopps
uint16_t step_ct;            //Schrittzähler
uint16_t kp, ki;             //Reglerkoeffizienten
int32_t diffA, diffB;        //Schrittdifferenz
int32_t integA, integB;      //Integralanteile
int32_t pwmA, pwmB;          //Stellwerte
int16_t speedA, speedB;      //Soll-Geschwindigkeit
uint8_t mrmsg[COM_PC_RMSG_LEN]; //Empfangsnachricht
uint8_t msmsg[COM_PC_SMSG_LEN]; //Sendnachricht
```

(Re-)Initialisierungsfunktion der reglerinternen Größen:

```
void regelung_reset(){       //Regelung initialisieren
  diffA = 0;                 //Regelungsabweichungen
  diffB = 0;                 //löschen
  integA = 0;                //Integralanteile löschen
  integB = 0;
}
```

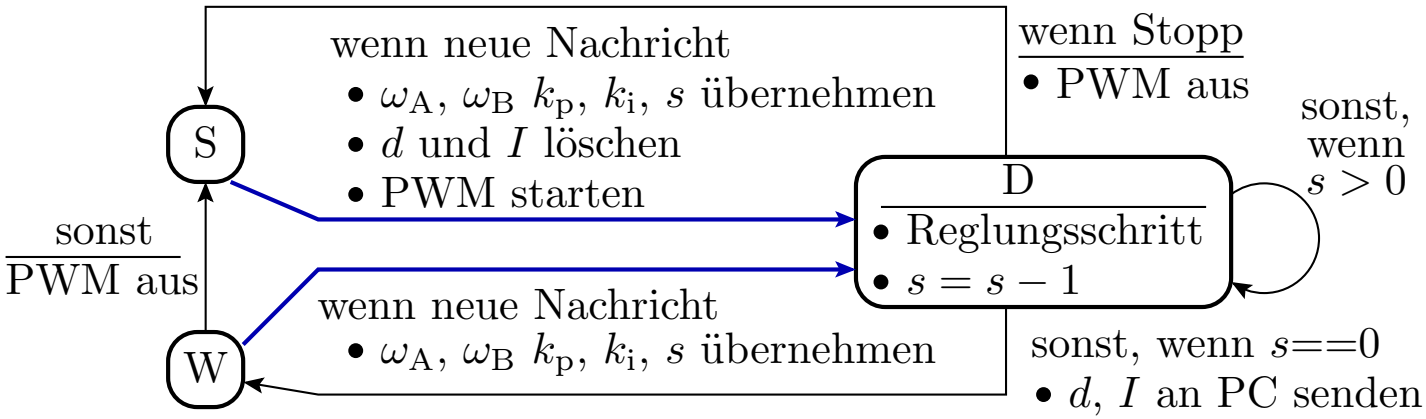
Programmrahmen mit Initialisierung:

```
int main(){
  uint8_t state = 'S';      //Anfangszustand gestoppt
  com_pc_init();            //Treiber initialisieren
  rotmess_init();
  pwm_init();
  lcd_init((uint8_t*)INITSTR);
  DDRA = 0;                 //für Tastereingabe
  sei();                    //Interrupts ein
  while(1){ ... }           //Endlosschleife
}
```

Verarbeitung einer Eingabenachricht

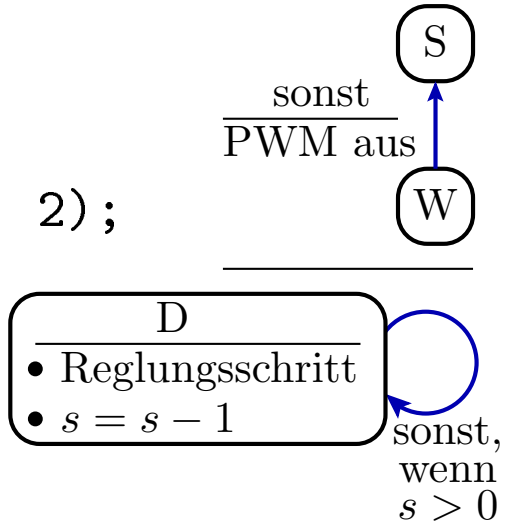
```
speedA = ((int16_t)mrmsg[0] <<8) + mrmsg[1];
speedB = ((int16_t)mrmsg[2] <<8) + mrmsg[3];
step_ct= ((uint16_t)mrmsg[4]<<8) + mrmsg[5];
kp =     ((uint16_t)mrmsg[6]<<8) + mrmsg[7];
ki =     ((uint16_t)mrmsg[8]<<8) + mrmsg[9];
```

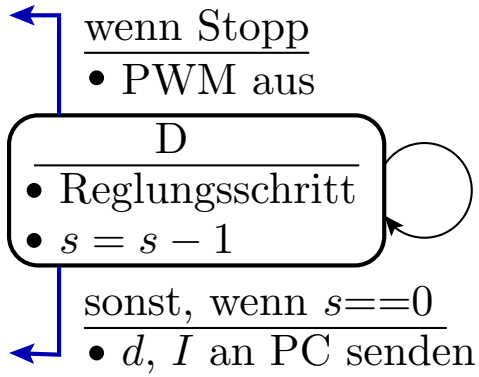
Ablauf in der Hauptschleife:




```
lcd_disp_chr(state, LCP_STATE); //Zustand anzeigen
  if (state=='W'||state=='S'){
    if (com_pc_get(mrmsg)){ //wenn neue Nachricht
      ...                   //ωA, ωB, ... übernehmen
      if (state=='S'){
          reglung_reset();
          pwm_start();
      }
      state = 'D';          //neuer Zustand "Bewegung"
    }
```

```
// wenn im Zustand «Weiter« und noch keine neue Eingabe
  else if (state == 'W'){
    pwm_stop();     // Motoren anhalten
    state = 'S';    // Zustand "Stop"
    lcd_disp_val(++stop_ct, LCP_STPCT, 2);
  }
}
if ((step_ct>0) && state =='D'){
  int16_t sa, sb;
  if (rotmess_get(&sa, &sb)){
    if (rotmess_err())        // bei Fehler in "rotmess"
      lcd_incErr(ERR_WMESS); // Fehlerzähler erhöhen
    ... // Ausführung eines PID-Regelschritts je Motor
    step_ct--;
    lcd_disp_val(abs(diffA)>>8, LCP_DIFFA, 3);
    lcd_disp_val(abs(integA)>>8, LCP_INTEGA, 3);
    lcd_disp_val(abs(diffB)>>8, LCP_DIFFB, 3);
    lcd_disp_val(abs(integB)>>8, LCP_INTEGB, 3);
  }
}
```






```
if ((step_ct<=0) && state =='D' ){
  //bei Bewegungsende oder
  //"Pause" Daten zum PC
  msmsg[0] =  diffA>>16;
  msmsg[1] =  diffA>>8 & 0xFF;
  msmsg[2] = integA>>16; msmsg[3] = integA>>8 & 0xFF;
  msmsg[4] =  diffB>>16; msmsg[5] =  diffB>>8 & 0xFF;
  msmsg[6] = integB>>16; msmsg[7] = integB>>8 & 0xFF;
  if (!com_pc_send(msmsg))  //wenn Senden versagt
    lcd_incErr(ERR_SEND);   //Fehlerzähler erhöhen
  state='W';                //Zustand => "Weiter"
}
if (PINA){                  //bei Tastendruck an Port A
  pwm_stop();               //Motoren anhalten
  state = 'S';              //Anfangszustand herstellen
  lcd_disp_str((uint8_t*)"xx", LCP_STPCT, 2);
  lcd_disp_str((uint8_t*)"....", LCP_ERR, 4);
  stop_ct = 0;
}
```

Python-Steuerprogramm

Reglungstest mit Python

Genutzte Module:

```
from conf_bstring import *     #Konv. Zahl<=>string
import serial                  #serielle Schnittstelle
import matplotlib.pyplot as plt #Plotfunktion
from sys import exit           #Programmabbruch
```

Grundeinstellung für die Reglung und Kommunikation:

```
kp = 1000; ki=500;             #Regelungskoeffizienten
ts = 5                         #Regelschr. je Nachricht
```

Wenn der PC die Nachrichten nicht schnell genug bereitstellt, wird die PWM gestoppt. Wenn auf dem LCD der Zähler für Bewegungsstopps sich um mehr als eins pro Bewegung erhöht (bzw. die Reglung ruckt), ts hochsetzen[6].

[6] Für Messungen im kürzeren Zeitabstand ist das Programm so umzuschreiben, dass die Zeittoleranzen und Datenpakete größer sind.

Beschreibung der Bewegung als Liste von Sollgeschwindigkeit-Dauer-Tupeln:

```
#Bewegungsablauftupen (speed, count)
#speed: Sollgeschwindigkeit in WS*256 je 20ms
#count Anzahl von Schritten der Dauer ts*20ms
trajList = [(2000, 20), (1000, 20), (-3000, 20)]
```

Funktion zur Erzeugung der Sendenachricht:

```
def sstr(v):
  msg = 2*int2bstr(v, 2) + int2bstr(ts, 2)
  msg += int2bstr(kp, 2) + int2bstr(ki, 2)
  return msg
```

Serielle Schnittstelle öffnen. COM anpassen. Timeout so setzen, dass Leseoperationen nach etwa der doppelten Zeit, in der die µP geantwortet haben muss, mit weniger gelesenen Bytes abbrechen:

```
serial.Serial("COM9", timeout=ts*0.04)
```

Anfangspunkt für die graphische Ausgabe:

```
t =[0]; dA=[0]; dB=[0]; s=[0]
```

Tabellenkopf der Textausgabe:

```
print ' t |speed|diff_A|intg_A|diff_B|intg_B|'
```

Damit der µP nach Abschluss jeder Teilbewegung die nächste Nachricht hat, müssen zum Bewegungsbeginn vor dem Warten auf die ersten Antwort zwei Nachrichten gesendet werden:

```
ser.write(sstr(trajList[0][0]))
```

Wiederhole für jedes Tupel der Trajektorliste »count« mal:

```
for (speed, count) in trajList:
  for idx in range(count):
    ser.write(sstr(speed)) #Nachricht senden
    rstr = ser.read(8)     #auf 8 Antwortbytes warten
    if len(rstr)<8:        #werden weniger empfangen[7],
      ser.close();         #Schnittstelle schliessen
      exit()               #Script beenden
```

[7] Das passiert, wenn eine Taste am Versuchboard gedrückt wird.

Sonst die 8 Bytes in ihre Bestandteile aufspalten. Zeit, Sollgschwindigkeit, ... tabellarisch ausgeben und für graphische Ausgabe an Listen hängen:

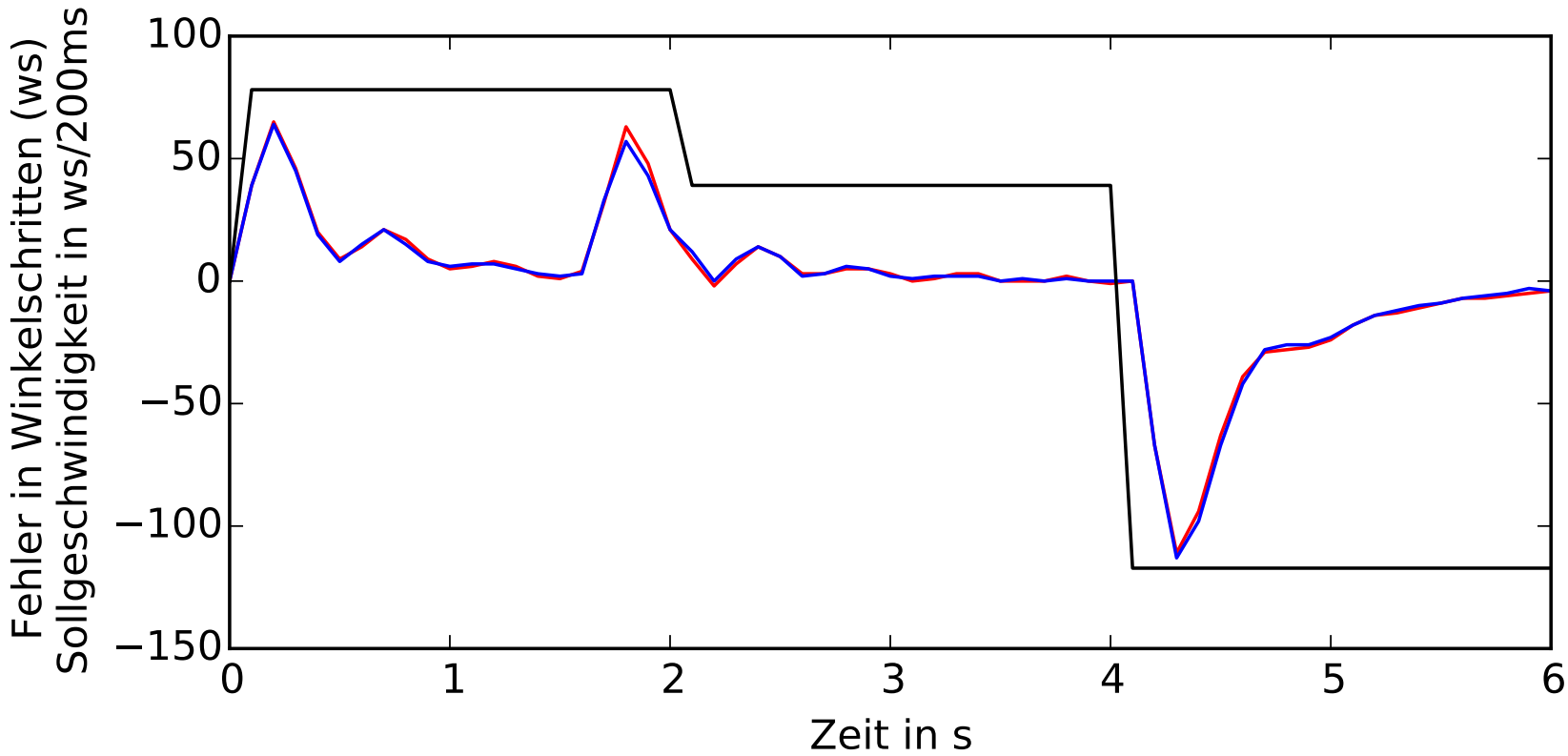
```
diff_A = bstr2int(rstr[0:2])
intg_A = bstr2int(rstr[2:4])
diff_B = bstr2int(rstr[4:6])
intg_B = bstr2int(rstr[6:8])
print '%3.1f|'%t[-1] + '%5i|'%speed, '%5i|'%diff_A,
print '%5i|'%intg_A, '%5i|'%diff_B, '%5i|'%intg_B
t  += [t[-1]+0.1]; s  += [speed*(10.0/256)]
dA += [diff_A];    dB += [diff_B]
```

Erzeugte Textausgabe:

```
 t |speed|diff_A|intg_A|diff_B|intg_B|
0.0| 2000|    39|   228|    39|   228|
0.1| 2000|    65|   796|    68|   809|
0.2| 2000|    48|  1354|    48|  1381|
0.3| 2000|    23|  1675|    20|  1682|
```

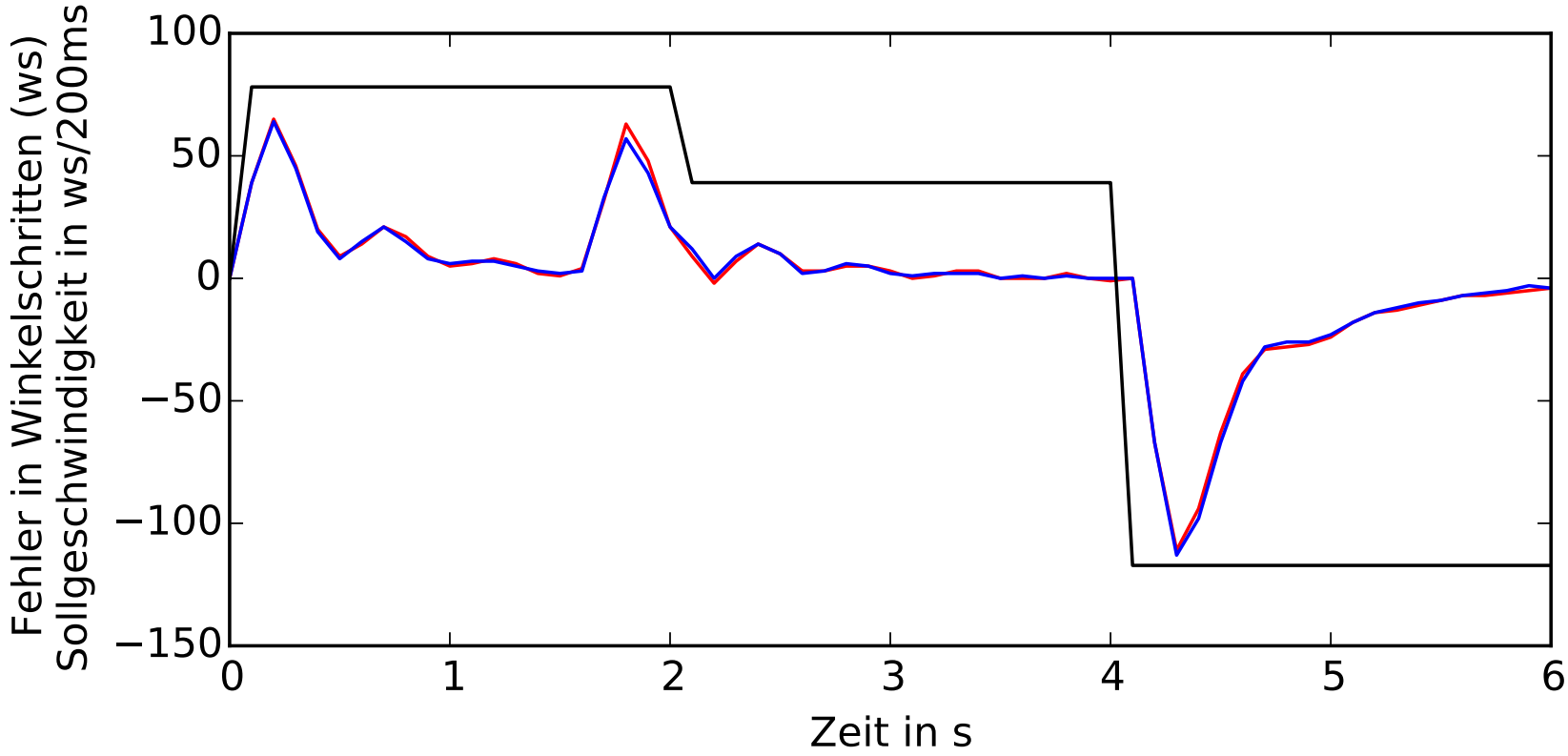
Serielle Schnittstelle schließen und Graphik erzeugen:

```
ser.close()
plt.plot(t, dA, 'r', t, dB, 'b', t, s, 'k')
plt.xlabel('Zeit in s')
txt = 'Fehler in Winkelschritten (ws)\n'
plt.ylabel(txt + 'Sollgeschwindigkeit in ws/200ms')
plt.show()
```



Ergebnisdiskussion

```
kp = 1000; ki=500;              #Regelungskoeffizienten
ts = 5                          #Regelschritte je Nachricht
trajList = [(2000, 20), (1000, 20), (-3000, 20)]
```



Bei jedem Geschwindigkeitssprung schwingt die Soll/Ist-Abweichung, bevor sie gegen null strebt. Das Schwingen lässt sich unterbinden,

- indem die Soll-Geschwindigkeit in kleinen Schritten oder stetig geändert wird.
- Durch bessere Wahl von k_p und k_i.

Experimentelle Bestimmung von k_p und k_i:

- $k_\mathrm{i} = 0$ setzen k_p so lange erhöhen, bis die Reglung schwingt. Davon auf 60% reduzieren.
- k_i soweit erhöhen, dass die Reglung schwingt und davon auch auf 60% reduzieren.