Informatikwerkstatt, Foliensatz 8 EA-Funktionen, Kommandointerpreter, PWM und PI-Regler

G. Kemnitz

Institut für Informatik, Technische Universität Clausthal

10. Dezember 2014

Inhalt des Foliensatzes

EA-Funktionen und Kommandointerpreter

Kommandointerpreter

Für die Fahrzeugsteuerung genügt die Kommandosprache:

```
Kommando ::= c {b}
```

(Kommandobyte c gefolgt von $n \geq 0$ Bytes b). Programmstrukt.:

```
void main(){
<Initialisierungen, Interrupts einschalten>
while (1) { // Hauptschleife mit Kommandointerpreter
  getCommand();
  switch(getState(Eingabezustand)){
   case <1. Kommandobuchstabe>:
    <Daten anfordern und Kommando ausführen>
    break;
   case <2. Kommandobuchstabe>:
    ...
  }
  <kurz zur Anzeigezeitverlängerung warten>
 }
```

Interrupt-gesteuerte Ein- und Ausgabe

Für den seriellen Datenaustausch mit dem PC empfiehlt sich eine ähnliche Funktionsaufteilung wie für die LCD-Ausgabe:

- Initialisierungsfunktion
- Startfunktion zur Vorbereitung der Ausgabe
- Schrittfunktion für das byte-weise Senden und Empfangen.

Änderungen und Funktionserweiterungen:

- Statt Anschlussfunktion Testfunktion auf Übertragungsabschluss bzw. Mode-Bit für blockierende Übertragung.
- Mode-Bit zur Umschaltung zwischen der Übertragung von null-terminierten Zeichenketten variabler Länge und Byte-Blöcken fester Länge.
- Spezielle Empfangsfunktion für Kommandobytes mit Zustandsanzeige des Eingabeinterpreters.
- sleep() zum Einfügen von Wartezeiten.

Funktionsdefinitionen in comPC.h

```
14 // Initialisiere USART0 an JE mit 9600 Baud 8n1
15 void initUSART0();
17 // Mode-Konstanten zur Steuerung der Übertragung
18 #define BLOCKING    1   // Bit 0 gesetzt
19 #define NONBLOCKING 0   // Bit 0 nicht gesetzt
20 #define FIXSIZE     2   // Bit 1 gesetzt
22 // Start einer Sendeoperation
23 uint8_t sendUSART0(uint8_t *ptr, uint8_t len, uint8_t mode);
25 // Test, ob Sendeoperation in Arbeit
26 uint8_t UA0SendBusy();
```

- Die Startfunktion bekommen einen Zeiger auf eine Byte-Feld, die (max.) Anzahl der zu übergebenden Bytes und den Übertragungsmodus BLOCKING, FIXSIZE, ... übergeben.

```
28  // Start einer Empfangsoperation
29  uint8_t getUSART0(uint8_t *ptr, uint8_t len, uint8_t mode);

31  // Test, ob Empfangsoperation in Arbeit
32  uint8_t UA0EmpfangBusy();

34  // Auf Kommandobyte warten
35  uint8_t getCommand();

37  // Warte t*100 ms
38  void sleep(uint16_t t);
```

- Die Funktion getCommand() blockiert bis Eingabeabschluss
  - zeigt bis zum Byte-Empfang als Eingabezustand '?'
  - ab Byte-Empfang das empfangene Zeichen an und
  - gibt das empfangende Zeichen zurück.
- Die Funktion sleep() dient zur Programmierung von Zeitabläufen, um Eingaben länger sichtbar zu machen, das Fahrzeug eine bestimmte Zeit anzuhalten, ...

Aufgabe 8.1: Experiment vorbereiten

- Legen Sie ein neues Projekt »TestComIR« an, binden Sie die Dateien für dieses Projekt von der Webseite ein.
- Belassen Sie das LC-Display an Port JE, den Abstandssensor an Port F und schließen Sie zusätzlich das USB-Modul an Stecker JD sowie an den PC an. Bitte während der Umbauarbeiten an der Hardware immer Spannung ausschalten.
- Übersetzen Sie das Programm, starten Sie es im Debugger und starten Sie HTerm. Die LCD-Anzeige fast wie im Vorprojekt. Zusätzliche Anzeige '?' als Kommunikationszustand für die Eingabeaufforderung.

USART-Initialisierung und private Daten

```
12  // Initialisiere USART0 an JE mit 9600 Baud 8n1
13  void initUSART0(){
14    UBRR0H=0; UBRR0L=49;  // 9600 Baud
15    UCSR0B = 0b00000000;  // Senden und Empfang aus
16    UCSR0C = 0b00000110;  // 8n1
    }

19  // Daten der Sendefunktion
20  uint8_t *UA0S_ptr;  // Zeiger auf den nächten Pufferplatz
21  uint8_t UA0S_len;   // Pufferlänge / Byteanzahl
22  uint8_t UA0S_mode;  // Sendemodus

24  // Daten der Empfangsfunktion
25  uint8_t *UA0E_ptr;  // Zeiger auf den nächten Pufferplatz
26  uint8_t UA0E_len;   // Pufferlänge / Byteanzahl
27  uint8_t UA0E_mode;  // Empfangsmodus
```

Senden, Empfang haben eigene Daten, können zeitgleich erfolgen.

Start einer Sendeoperation

```
30  uint8_t sendUSART0( uint8_t *ptr, uint8_t len, uint8_t mode){
31    if (UA0SendBusy()){   // Wenn vorherige Sendeoperation
32      incErr(UA0_SByCt);  // Sender-besetzt-Fehlerzähler
33      return 1;           // erhöhen, dann Fehlerabbruch.
      }
35    if ((len==0)||((*ptr==0)&&(mode&FIXSIZE)==0))// Wenn
36      return 0;           // nichts zu senden, Normalabbruch.
38    UA0S_ptr   = ptr;     // Zeiger auf 1. Zeichen,
39    UA0S_len   = len;     // Pufferlänge / Byteanzahl und
40    UA0S_mode  = mode;    // Sendemodus kopieren.
41    UCSR0B |= (1<<TXEN0); // Sender einschalten,
42              (1<<UDRIE0);// Puffer-frei-Interrupt ein.
44    if (mode & BLOCKING){ // Wenn blockierendes Senden
45      while (UA0SendBusy()){} // warte bis alle Bytes
      }                     // versendet sind.
47    return 0;             // Normalabbruch
    }
```

Schrittfunktion für das Senden und Test, ob fertig

```
51 ISR(USART0_UDRE_vect){
52   uint8_t c=*UA0S_ptr;// Nächsten Byte aus Puffer lesen
53   UA0S_ptr++;          // Zeiger erhöhen und Länge bis
54   // Bei FIXSIZE, wenn alle Bytes gesendet, oder sonst,
55   // wenn Zeichen null oder Länge bis Pufferende null
56   if (((c==0)&&((UA0S_mode&FIXSIZE)==0))||(UA0S_len==0))
57     // Sender und Puffer-frei-Interrupt ausschalten,
58     UCSR0B &= ~((1<<TXEN0)|(1<<UDRIE0));
59   else {          // sonst
60     UDR0 = c;     // Zeichen senden und Restbytezahl bzw.
61     UA0S_len--;  // Länge zum Pufferende verringern.
   }
}

65 // Test, ob Sendeoperation in Arbeit
66 uint8_t UA0SendBusy(){return UCSR0B & (1<<TXEN0);}
```

Start einer Empfangsoperation

```
69  uint8_t getUSART0(uint8_t *ptr, uint8_t len, uint8_t mode){
70    if (UA0EmpfangBusy()){// Wenn vorherige Empfangsoperation
71      incErr(UA0_EByCt);  // nicht fertig, Fehlerzähler
72      return 1;           // erhöhen, dann Fehlerabbruch.
      }
74    if (len==0)           // Wenn auf keine Bytes gewartet
75      return 0;           // wird, Normalabbruch.
77    UA0E_ptr    = ptr;    // Zeiger auf Pufferanfang,
78    UA0E_len    = len;    // Pufferlänge / Byteanzahl
79    UA0E_mode   = mode;   // und Empfangsmodus speichern.
80    UCSR0B = (1<<RXEN0)|  // Empfänger und Empfangs-
81           (1<<RXCIE0);   // Interrupt einschalten.
83    if (mode & BLOCKING){ // Wenn blockierend
84      while (UA0EmpfangBusy()){}// warte bis alle
      }                     // Bytes empfangen sind.
86    return 0;             // Normalabbruch
    }
```

Schrittfunktion für den Empfang

```
 90 ISR(USART0_RX_vect){
 91   uint8_t c = UDR0;      // Empfangenes Byte lesen
 92   *UA0E_ptr = c;         // in den Puffer kopieren
 93   UA0E_ptr++;            // Zeiger erhöhen und Länge
 94   UA0E_len--;            // bis Pufferende verringern.
 95   if (UCSR0A & (1<< FE0)) // Wenn Frame Error
 96     incErr(UA0_FeCt);     // Fehlerzähler erhöhen.
 97   if (UCSR0A & (1<< DOR0))// Wenn Empfangspufferüber-
 98     incErr(UA0_DorCt);    // lauf, Fehlerzähler erhöhen.
 99   if (UCSR0A & (1<< UPE0))// Wenn Paritätsfehler,
100     incErr(UA0_ParCt);    // Fehlerzähler erhöhen

102   // Bei FIXSIZE, wenn alle Bytes empfangen, oder sonst
103   // wenn Zeichen null oder Länge bis Pufferende null
104   if (((c==0)&&((UA0E_mode&FIXSIZE)==0))||(UA0E_len==0))
105   // Empfänger und Empfangsinterrupt ausschalten.
106     UCSR0B &= ~((1<<RXEN0)|(1<<RXC0));
   }
```

Empfangsfunktion für Kommandobytes

```
109   // Test, ob Empfangsoperation in Arbeit
110   uint8_t UA0EmpfangBusy(){return UCSR0B & (1<<RXEN0);}
111
112     // Auf Kommandobyte warten
113   uint8_t getCommand(){
114     setState('?', Eingabezustand);// Warte auf Kommando
115
116     UCSR0B |= (1<<RXEN0);   // Empfänger einschalten.
117     while (!(UCSR0A&(1<<RXC0))){} // auf Byte warten
118     uint8_t w = UDR0;       // Byte lesen
119     UCSR0B &= !(1<<RXEN0);  // Empfänger ausschalten.
120     setState(w, Eingabezustand); // Zustands setzen
121     return w;               // Empfangswertrückgabe
122   }
```

- Hier ist ein gefährlicher Fehler drin. Wer sieht ihn?

Warte-Funktion

```
124  // Warte für 100 ms
125 void sleep100ms(){
126    ETIFR = (1<<TOV3);       // Überlaufbit löschen
127    TCNT3  = 53692;          // 11844 Zählschr. bis Überl.
128    TCCR3B = (0b11<<CS30);   // Zähle CPU-Takte/64
129    while (!(ETIFR & (1<<TOV3))){};// warte bis Überlauf
130  } TCCR3B = 0;              // Zähler aus

133  // Warte t*100 ms
134 void sleep(uint16_t t){
135    uint16_t Ct;
136    for (Ct=0; Ct<t; Ct++)   // wiederhole t-mal
137  }   sleep100ms();          // warte 100ms
```

- Für das Warten von 100ms wird Timer 3 mit dem CPU-Takt von 7,58 MHz geteilt durch 64 von 11844 bis 2^{16} gezählt und auf das Überlaufbit gewartet.

Kommandointerpreter

Für die Fahrzeugsteuerung genügt die Kommandosprache:

```
Kommando ::= c {b}
```

(Kommandobyte c gefolgt von $n \geq 0$ Bytes b). Programmstrukt.:

```
void main(){
<Initialisierungen, Interrupts einschalten>
while (1) { // Hauptschleife mit Kommandointerpreter
  getCommand();
  switch(getState(Eingabezustand)){
   case <1. Kommandobuchstabe>:
    <Daten anfordern und Kommando ausführen>
    break;
   case <2. Kommandobuchstabe>:
    ...
  }
  <kurz zur Anzeigezeitverlängerung warten>
 }
```

In der Datei TestComIR.c

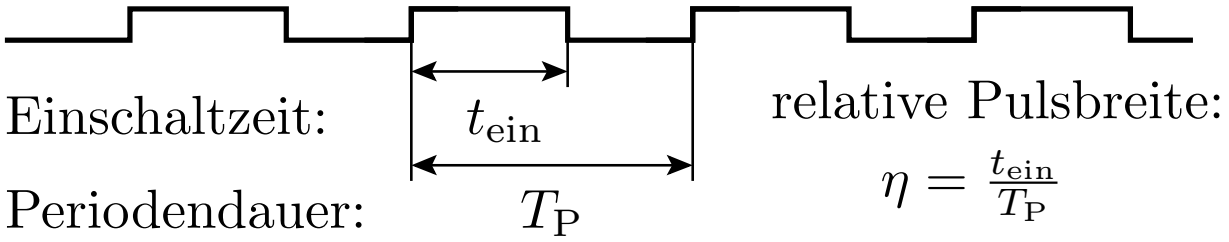
- sind die Steuerschrittfunktion und die Initialisierungen in main() aus dem Vorprojekt übernommen: Der Abstand wird gemessen, alle 2s Steuerzähler A weiter geschaltet, ...
- Zusätzliche Initialisierung von USART0 für die Verbindung zum PC.
- Für Kommando 'a' erfolgt derselbe Dialog, wie beim Test der Texteingaben und -ausgaben mit Schrittketten
  - Ausgabe des Texts »Eingabe <Nr>:
  - Einlesen eines null-terminierten Textes
  - Anhängen des Texts und eines Zeilenumbruchs an die Eingabezeile.
- Für Kommando 'b' werden
  - genau vier Bytes gelesen,
  - als zwei 16-Bit-Zahlen gespeichert,
  - die 16-Bit-Zahlen addiert und
  - das Ergebnis als 2 Bytes an den PC gesendet.

```
44 int main(void){
45   initUSART0();
46   // -- LCD- und ADC-Initialisierungen wie Vorprojekt --
...
54   sei();
55   while(1){
56     getCommand();// Kommandobyte lesen und als Eingabezust. speichern
57     switch (getState(Eingabezustand)){  // Fallunterscheidung
58       case 'a':                         // Zeichenketten blockierend
59         // Textein- und Ausgabe in Schrittkettenbeschreibung
...
72       break;
73       case 'b':                       // feste Byteanzahl, blockierend
74         getUSART0(buff, 4, FIXSIZE|BLOCKING);// auf 4 Bytes warten
75         uint16_t A = (buff[0]<<8) + buff[1];// in zwei 16-Bit
76         uint16_t B = (buff[2]<<8) + buff[3];// Zahlen umwandeln
77         A += B;                           // Zahlen addieren
78         buff[0] = A>>8;                   // Summe in Bytes
79         buff[1] = A & 0xFF;               // aufteilen und
80         sendUSART0(buff, 2, FIXSIZE|BLOCKING);// zurückschicken
81         break;
       }
83     sleep(10);                        // 10*100ms=1s warten
   }
}
```

Motorsteuerung mit Pulsweitenmodulation

PWM-Erzeugung

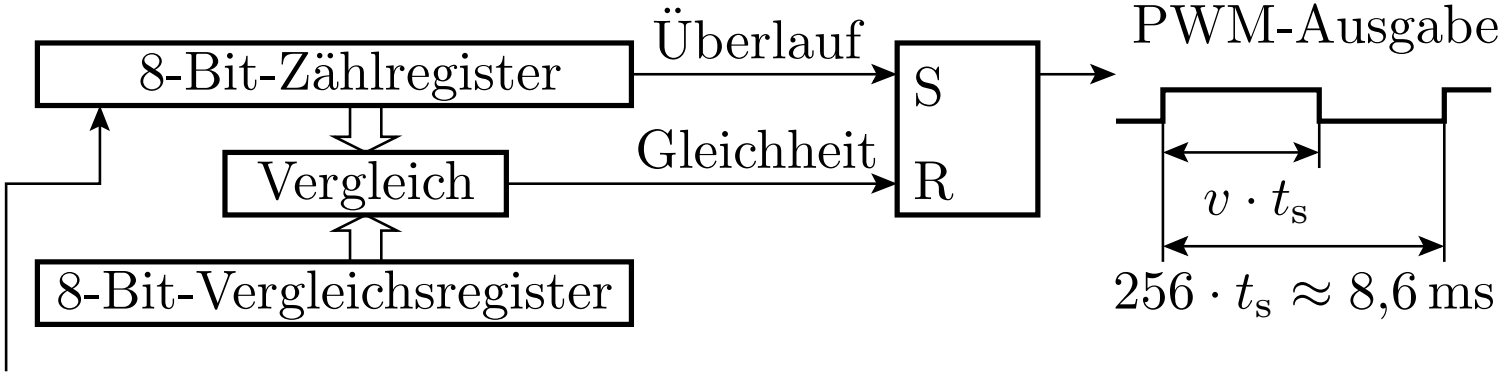
Ein PWM- (PulseWeitenModulation) Signal hat eine konstante Periode und eine variable Einschaltzeit.



Der Antriebswert ist proportional zur relativen Pulsbreite.

Die Timer des Mikrorechners können so konfiguriert werden, dass an den Enable-Eingängen der H-Brücken PWM-Signale ohne Programmunterstützung erzeugt werden:

- Motor A: Timer 0, Ausgang OC0
- Motor B: Timer 1, Ausgang OC1A



Takt mit der Periode:

$$t_s = \frac{256}{f_{CPU}} \approx 34\mu s$$

	Motor A	Motor B
Zählregister	TCNT0	TCNT1
Vergleichsregister	OCR0	OCR1A
PWM-Ausgabe	PB4 (ENA)	PB5 (ENB)

Einstellungen für Timer 0:

- Fast PWM (WGM0[0:1] = 0b11
- OC0 bei Gleichheit löschen (COM0[1:0]=0b10).
- Vorteiler 256 ($T_P \approx 100\,\text{Hz}$, CS0[2:0=110)

Einstellungen für Timer 1:

- Fast PWM, 8-Bit (WGM1[3:0]=0b0001)
- OC1A bei Gleichheit löschen (COM1A[1:0]=0b10).
- Vorteiler 256 ($T_P \approx 100\,\text{Hz}$, CS1[2:0=100) .

Aufgabe 8.2: Experiment vorbereiten

- Legen Sie ein neues Projekt »TestPwmIR« an. Binden Sie die Dateien für dieses Projekt von der Webseite ein.
- Belassen Sie das LC-Display an Stecker JE, den Abstandssensor an Stecker JF und das USB-Modul an Stecker JD. Schließen Sie zusätzlich die Motoren über die H-Brücken an die Stecker JA und JB und das Bodensensormodul an Stecker JC an. Bitte während der Umbauarbeiten an der Hardware Spannung aus.
- Übersetzen Sie das Programm, starten Sie es im Debugger und starten Sie HTerm. Verfügbare Kommandos:
  - 'a': PWM anschalten.
  - 'b' PWM ausschalten.
  - 'c': Steuerprogramm aktivieren.
  - 'd'<A1><A0><B1><B0: Motorwerte (relative Pulsbreiten) A und B setzen (Wertebereich 0xFF01 bis 0x00FF).

Initialisieren der PWM-Steuerung

```
15 void startPWM(){
16  // Motorsteuersignale als Ausgänge initialisieren
17    DDRB = 0b00110000; //EnA(PB4), EnB(PB5)
18    DDRD = 0b11000000; //DirA(PD7), DirB(PD6)
20    // Einstellungen für PWM an PB4 (Timer 0)
21    // Fast PWM:         WGM0[1:0] = 11
22    // Vorteiler 256:    CS0[2:0]  =110 (/256, ca. 100 Hz)
23    // nicht inv Aufg.: COM0[1:0] = 10
24    TCCR0 = (1<<WGM00)|(1<<WGM01)|(1<<COM01)|(0b110<<CS00);
25    OCR0 = 0;           // relative Pulsbreite null
27    // Einstellungen für PWM an PB5 (Timer 1)
28    // Fast PWM 8 Bit:  WGM0[3:0] = 0101
29    // Vorteiler 256:    CS1[2:0]  =100 (/256, ca. 100 Hz)
30    // nicht inv Aufg.: COM1A[1:0] = 10
31    TCCR1A |= (0b10<<COM1A0)|(0b01<<WGM10)|(0b10<<COM1A0);
32    TCCR1B |= (0b01<<WGM12) |(0b100<<CS10);
33    OCR1A = 0;          // relative Pulsbreite null
  }
```

Schreiben der relativen Pulsbreite

```
37  // A, B: Motorsteuerwerte, WB: -255 bis 255
38  void setPWM(int16_t A, int16_t B){
39    if (A>=0)             // wenn Steuerwert positiv
40      SetDirA_pos;        // setze Richtung auf positiv
41    else {                // sonst
42      A=-A;               // Bilde Betrag
43      SetDirA_neg;        // setze Richtung auf negativ
      }
45    if (A>255) A=255; // Betragsbegrenzung, rela-
46    OCR0 = A;           // tive Pulsbreite einstellen
47    if (B>=0){SetDirB_pos;} // dasselbe für Motor B
48    else {B=-B; SetDirB_neg;}
49    if (B>255) B=255;
50    OCR1A = B;
  }
```

Die SetDir-Anweisungen sind Makros, mit denen die Richtungsbits gesetzt und gelöscht werden:

```
 9   #define SetDirA_pos    PORTD&=~(1<<7)
10   #define SetDirA_neg    PORTD|=(1<<7)
11   #define SetDirB_pos    PORTD&=~(1<<6)
12   #define SetDirB_neg    PORTD|=(1<<6)
```

PWM Ausschalten:

```
52 void stopPWM(){
53    TCCR0 = 0;
54    TCCR1A &= ~((0b10<<COM1A0)|(0b01<<WGM10)|(0b10<<COM1A0));
55    TCCR1B &= ~((0b01<<WGM12) |(0b100<<CS10));
56    PORTB  &=~ 0b00110000; //EnA(PB4)=0, EnB(PB5)=0
   }
```

- Anhalten der PWM und Trennen der Timer-PWM-Ausgänge von den Port-Anschlüssen.
- ENA und ENB null setzen.

Die Steuerschrittfunktion

```
20   #define BSens1 (PINE&(1<<6))
21   #define BSens2 (PINE&(1<<4))
22   #define BSens3 (PINE&(1<<7))
23   #define BSens4 (PINE&(1<<5))
24
25   void Steuerschrittfunktion(uint16_t VAbst){
26     if (getState(Steuerzustand_A)=='A'){
27       if (VAbst < 1500)      // wenn Spannung Abstands
...        <nächste Folie>
37     } else setPWM(0, 0);     // sonst Halt
39   startLCD();                // LCD-Ausgabe starten
     }
```

- Nur wenn kein Hindernis, steuere eine Bewegung.
- Sonst relative Pulsbreite null für beide Motoren.

Wenn kein Hindernis:

```
28  if (BSens1&&BSens4) // Linie zwischen Außensensoren
29    if (BSens2&&BSens3)// Linie zwischen Innensensoren
30      setPWM(70, -70);// Fahrt gerade
31    else if (BSens2)  // Linie nicht unter rechtem Sensor
32      setPWM(70, 0);  // Fahrt nach links
33    else if (BSens3)  // Linie nicht unter linkem Sensor
34      setPWM(0, 70);  // Fahrt nach rechts
35    else setPWM(0,0); // sonst Halt
36  else setPWM(0, 0);  // sonst Halt
```

- Einfacher Algorithmus zur Linienverfolgung. Beim Ausprobieren stellt man fest, dass er nicht in allen praktisch auftretenden Situationen zufriedenstellend funktioniert.

Das Hauptprogramm

```
42  uint8_t buff[4];

44  int main(void){
45    initUSART0();
46    // LCD- und ADC-Initialisierungen wie Vorprojekt
47    // Das ADC-Objekt bekommt einen Zeiger auf die Steuer-
48    initADC(&Steuerschrittfunktion); // schrittfunktion
49    // die als letztes die nächste LCD-Ausgabe startet.
50    // Das LCD-Objekt bekommt einen Zeiger auf die
51    initLCD(&startADC);// Funktion zum Start der Messungen
52    startADC();      // Start der ersten Messung
53    sei();           // globale Interruptfreigabe
55    while(1) {
        <Kommandointerpreter in der Hauptschleife>
      }
    }
```

```
56      getCommand();
57      switch (getState(Eingabezustand)){
58        case 'a':   // PWM anschalten
59          setState('B', Steuerzustand_A);
60          startPWM();
61          break;
62        case 'b':   // PWM ausschalten
63          stopPWM();
64          setState('-', Steuerzustand_A);
65          break;
66        case 'c':   // Steuerprogramm aktivieren
67          setState('A', Steuerzustand_A);
68          break;
69        case 'd':   // Motorwerte setzen
70          getUSART0(buff, 4, FIXSIZE|BLOCKING);
71          uint16_t A = (buff[0]<<8) + buff[1];
72          uint16_t B = (buff[2]<<8) + buff[3];
73          setState('F', Steuerzustand_A);
74          setPWM(A, B);
75          break;
        }
77      sleep(10);
```

PI-Regler

Vorüberlegungen

PI-Regler

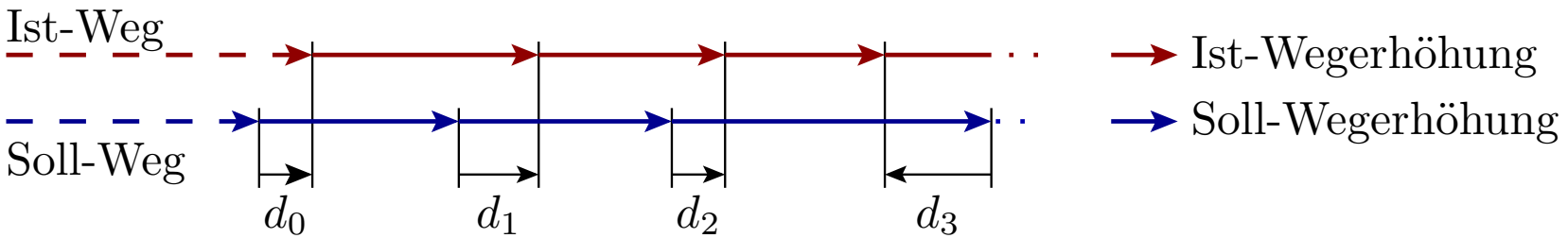
Zu regeln ist die Radposition. Ist- und Sollwert ergeben sich jeweils aus den aktuellen Werten und einem Inkrement:

- für den Soll-Wert die Soll-Wegerhöhung pro Schrittzeit und
- für den Ist-Wert die Ist-Wegerhöhung pro Schrittzeit.

Die Soll/Ist-Abweichung d erhöht sich in jedem Schritt um die Soll-Wegerhöhung minus der Ist-Wegerhöhung. Ein PI-Regler berechnet aus d den Stellwert s für den Motor:

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

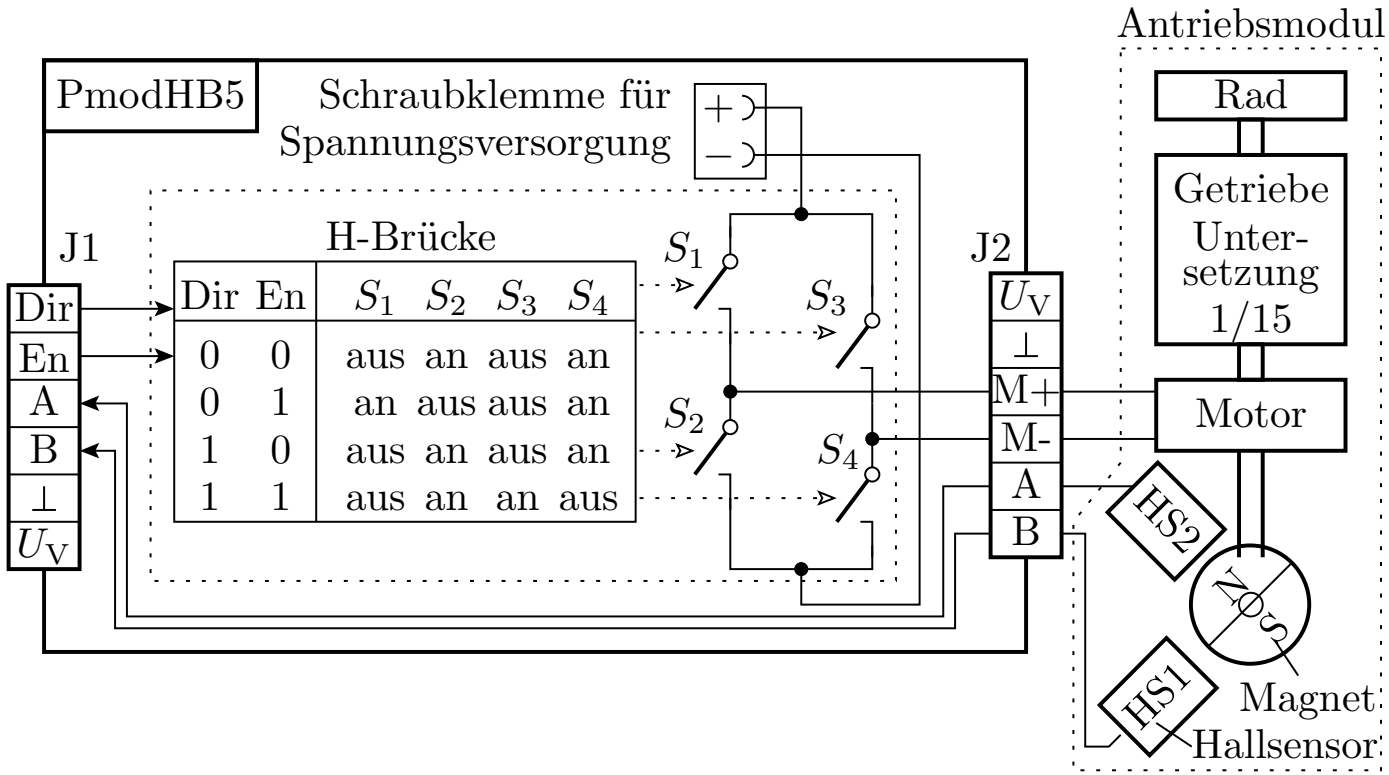
(k_p, k_i – empirisch bestimmbare Reglerparameter, I – Integralteil.



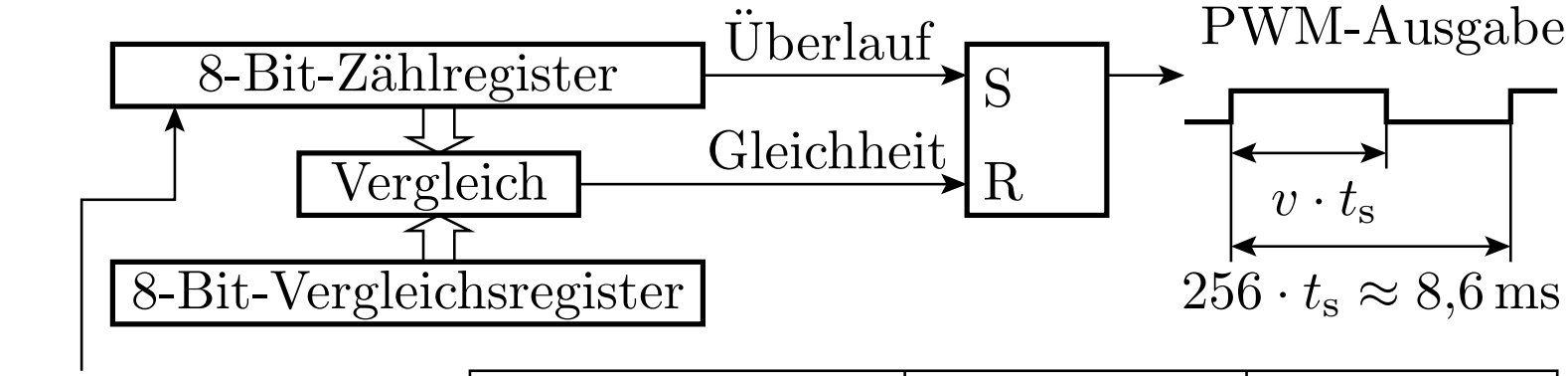
PWM-Steuerung

Port-Zuordnung Mikrocontroller

Motor an JB	Motor an JA
PD6	PD7
PB5	PB4
PD4	PD5
PB6	PB7



Die Drehrichtung wird mit dem Dir-Bit und die Drehgeschwindigkeit mit der relativen Pulsbreite am EN- (Enable) Eingang gesteuert.



	Motor A	Motor B
Vergleichsregister	OCR0	OCR1A
PWM-Ausgabe	PB4 (ENA)	PB5 (ENB)

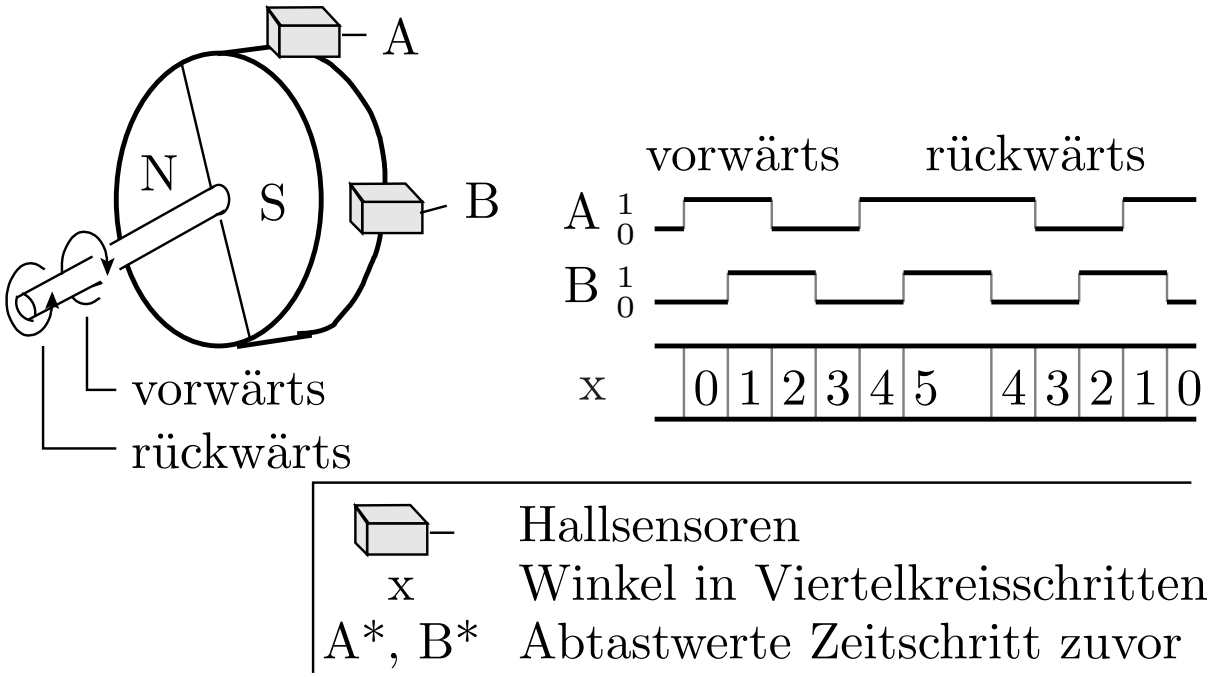
Die PWM-Signale sollen wie im Vorprojekt mit den Timern 0 und 1 erzeugt werden. Der Header pwm.h stellt bereit:

```
14 // Initialisierung der PWM-Steuerung für die Motoren
15 void startPWM();

17 // Schreiben der relativen Pulsbreite
18 // A, B: vorzeichenbehaftet, 7 Nachkommastellen
19 void setPWM(int16_t A, int16_t B);

21 // PWM ausschalten
22 void stopPWM();
```

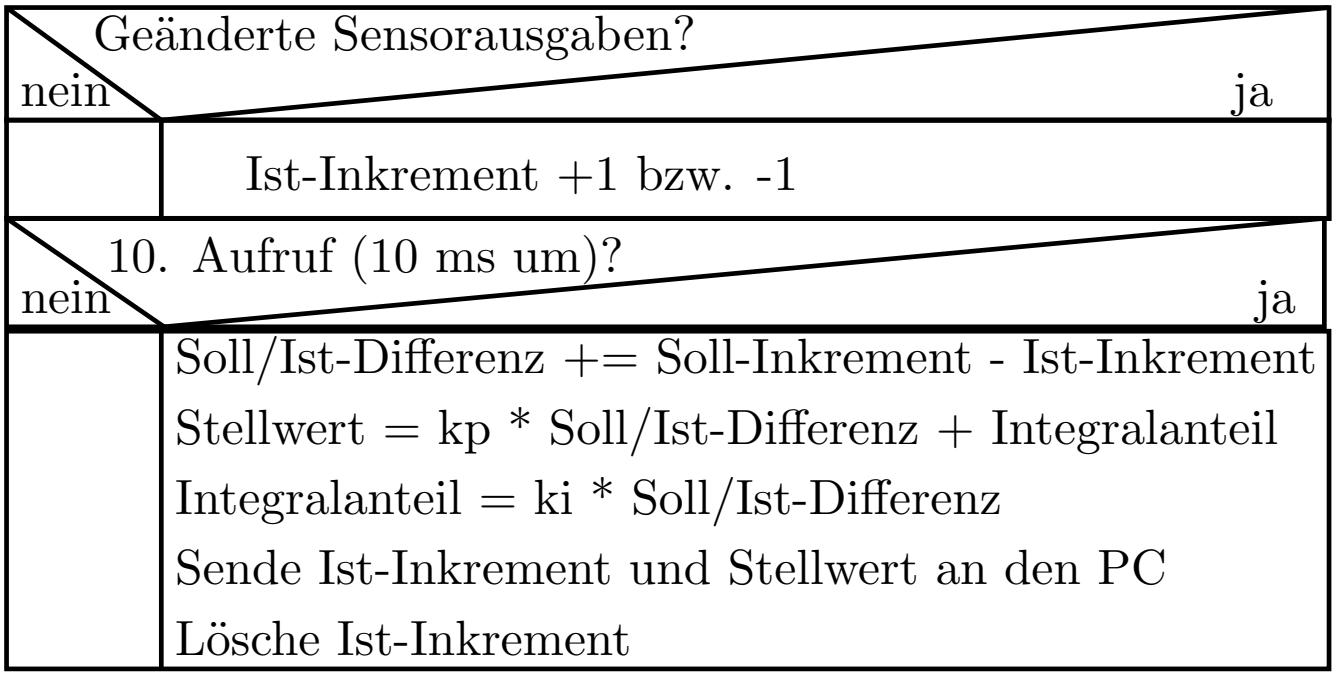
Positionsmessung



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

- Die Messung erfolgt mit den Hallsensoren A und B. Es sind 4 Winkelstellungen pro Motorumdrehung und 60 pro Radumdrehung unterscheidbar.
- Abtastung und Auswertung der Sensorsignale 800 bis 1000 mal pro Sekunde.

Mess- und Reglerablauf



- Die Sensorwerte müssen mit 1 ms abgetastet werden. Für den Regler genügt eine Periode von 10ms.
- Abzufangende Fehlerfälle:
  - Wertebereichsüberläufe der Differenz und des Integralteils,
  - Unterabtastung der Sensorsignale, ...

Zahlenformate und Wertebereich

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

Größe	Datentyp	Wertebereich
Differenz d	int32_t	-0x100,00 bis + 0x100,00
Prop.-Koeff. k_p	uint16_t	0x0,00 bis 0x7F,FF
Integralanteil I	int32_t	-0x4000,0000 bis + 0x4000,0000
Integ.-Koeff. k_i	uint16_t	0x0,00 bis 0x3F,FF
Stellwert s	int16_t	-0xFF bis 0xFF

Zur Verhinderung von Wertebereichsüberläufen werden Zwischenergebnisse außerhalb des Wertebereichs auf die überschrittene Wertebereichsgrenze beschränkt (Sättigungsarthmetik).

Fehlerbehandlung

Der Stellwert während der Regelung soll in 5 Stufen auf dem LCD angezeigt werden:

- Überschreitung Maximalwerts (max. vorwärts): '^'
- zulässiger positive Stellwert: '+'
- Stellwert null: '0'
- zulässiger negative Stellwert: '-' und
- Unterschreitung Minimalwert (max. rückwärts): 'v'

Wenn die Regelung für 0,5 s max. Geschwindigkeit vorgibt, ohne das sich der Motor bewegt[1], soll der Stellwert zum Schutz des Motors auf null und die Zustandsanzeige auf '!' gesetzt werden.

Bei Sensorabtastfehlern (gleichzeitige Änderung beider Sensorwerte) wird Fehlerzähler 5 (WM_Abtast) hochgezählt.

[1] Das passiert z.B., wenn der Motor nicht angeschlossen oder mechanisch blockiert ist.

Aufgabe 8.3: Experiment vorbereiten

- Legen Sie ein neues Projekt »TestPI_Regler« an. Binden Sie die Dateien für dieses Projekt von der Webseite ein.
- Belassen Sie das LC-Display an Stecker JE, den Abstandssensor an Stecker JF, das USB-Modul an Stecker JD und die H-Brücken an den Steckern JA und JB. Bitte während der Umbauarbeiten an der Hardware Spannung ausschalten.
- Übersetzen Sie das Programm, starten Sie es im Debugger und starten Sie HTerm. Bei Eingabe:
  - 'a' 0x02 0x00 0xFE 0x00 0x01 0x00: Geregelte Bewegung mit etwa einem viertel der Maximalgeschwindigkeit, Motor A vorwärts, Motor B rückwärts für ca. 2,5 s.
  - 's': Bewegung stoppen.

  Alternativ stoppt die Bewegung auch bei einem Hindernis vor dem Abstandssensor (Abstandsspannung > 2V).

Private Daten

Private Daten der Regelung

```
15 // Datenstruktur für die PI-Regeler der Motoren
16 struct PI_struc {
17   uint8_t SState; // aktuelle und vorherige Sensorwerte
18   int8_t istInc;  // Wegzählererhöhung in 10 ms, -8..+8
19   int32_t diff;   // Soll-Ist-Differenz mit 8 Nachkommabits
20   int32_t integ;  // Integralanteil mit 8 Nachkommabits
21   int16_t sollInc;// Sollwerterhöhung mit 8 Nachkommabits
22   int16_t stellw; // Stellwert für den Motor -0xFF..+0xFF
23   uint8_t mbCt;   // Zeitzähler für Motorblockierung
24   uint8_t MState; // Zustandsnummer des Motors
25 };
```

Von dieser Datenstruktur wird für jeden Motor eine Instanz angelegt und mit einem Zeiger an die Bearbeitungsfunktionen übergeben.

- Initialisierungsfunktion für die Regler-Datenstruktur:

```
30 void clearPiStruc(struct PI_struc *m, uint8_t MState){
31   m->istInc = 0;  m->diff     = 0;
32   m->integ  = 0;  m->sollInc  = 0;
33   m->stellw = 0;  m->mbCt     = 0;
34   m->MState = MState; // Zustansnr. auf dem LCD:
35 }                     // 1 für Motor A, 3 für Motor B
```

- Die privaten Datenobjekte des Software-Moduls:

```
38 struct PI_struc PI_MA, PI_MB;//Daten der Motorregelungen
39 uint8_t PI_Ct_ms;         // Zähler für ms-Schritte
40 int16_t PI_Restzeit10ms;// Zähler für 10ms-Schritte
41 uint8_t PI_buf[4];        // Puffer für die Testausgabe
42 uint16_t PI_kp = 500;    // Reglerparameter
43 uint16_t PI_ki = 800;    // ki<0x4000 und kp<0x7FFF
```

Regler und Wegmessung

Der Regler

```
47 void Regler(struct PI_struc *m){
48    // Soll/Ist-Abweichung bestimmen und begrenzen
49    m->diff += (m->sollInc - ((int16_t)m->istInc*256));
50    if (m->diff> 0x10000) m->diff =  0x10000;
51    if (m->diff<-0x10000) m->diff = -0x10000;
52    // Stellw = (Integ/256 + diff * kp)>>16;
53    int32_t tmp = (m->integ/256) + m->diff * PI_kp;
54    m->stellw = tmp >>16;
55    // neuen Integralteil berechnen und begrenzen
56    m->integ += (m->diff) * PI_ki;
57    if (m->integ> 0x40000000) m->integ =  0x40000000;
58    if (m->integ<-0x40000000) m->integ = -0x40000000;
```

- tmp>>16 – Abschneiden der 16 Nachkommastellen.

```
60    // Reaktion auf Schleppfehler und Motorblockierung
61    if (m->stellw >  0xFF){      // wenn Stellwertüberlauf
62      m->stellw =  0xFF;         // begrenze Stellwert
63      setState('^',m->MState);   // Motorzustandsanzeige: ^
64    } m->mbCt++;                 // Überlastzähler erhöhen
66    else if (m->stellw < -0xFF){// wenn Stellwertunterlauf
67       m->stellw = -0xFF;        // begrenze Stellwert
68       setState('v',m->MState); // Motorzustandsanzeige:
69       m->mbCt ++;               // Überlastzähler erhöhen
70    }   // normale Motorzustände: + vorw., - rückw., 0 halt
71    else if (m->stellw > 0) setState('+' ,m->MState);
72    else if (m->stellw < 0) setState('-' ,m->MState);
73    else setState('0', m->MState);
74    if (m->istInc)               // wenn sich der Motor bewegt
75      m->mbCt = 0;               // Überlastzähler rücksetzen
76    if (m->mbCt>50) {            // Motor blockiert >0,5 s
77      m->stellw = 0;             // Motor aus
78    } setState('!', m->MState); // Motorzustandsanzeige: !
}
```

Geht der Motor nach Nothalt wieder an?

- Kontrolldaten für Matlab in 2 Bytes je Motor verpacken:

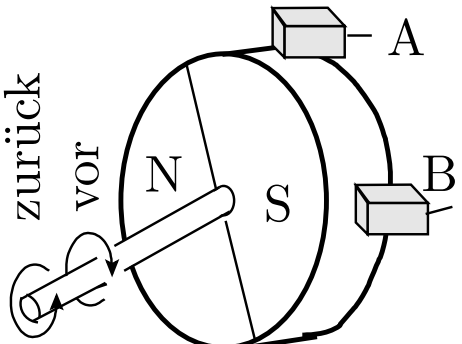
```
85 void ErzeugeAusgabe(struct PI_struc *m, uint8_t *buff){
86   uint8_t istInc = (uint8_t)m->istInc;
87   uint16_t stellw = (uint16_t)m->stellw;
88   *buff = (stellw >> 1);// Stellwertbits 8..1
89                  // Stellwertbit 0, istInc-Bits 6..0
90   *(buff+1) = ((stellw & 1)<<7) |(istInc & 0x7F);
91   m->istInc = 0;          // löschen des Ist-Increments
92 }
```

- Lesen der Sensorwerte

```
94   // Lesen der Sensorwerte Rad A als 2-Bit-Vektor
95 uint8_t getSWA(){
96   return ((PIND>>4)&0b10)|((PINB>>7)&0b01);
   }
99   // Lesen der Sensorwerte Rad B als 2-Bit-Vektor
100 uint8_t getSWB(){
101  return ((PIND>>3)&0b10)|((PINB>>6)&0b01);
   }
```

- Wegmessung

```
107 void QuadEnc(struct PI_struc *m, uint8_t SW){
108   m->SState = ((m->SState<<2)&0b1100)|(SW&0b11);
109   switch (m->SState){
110     case 0b0010:
111     case 0b0100:
112     case 0b1011:
113     case 0b1101:
114     m->istInc++;
115     break;
116     case 0b0001:
117     case 0b0111:
118     case 0b1000:
119     case 0b1110:
120     m->istInc--;
121     break;
122     case 0b0011:
123     case 0b0110:
124     case 0b1100:
125     case 0b1001:
126     incErr(WM_Abstast);
    }
  }
```



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

Steuerung von Bewegungen

Bewegung starten

```
133 void startBewegung(int16_t incA, int16_t incB, uint16_t Zeit){
134   if (!(TIMSK & (1<<OCIE2))){ // Wenn Motorregelung aus
        <Neuinitialisierung des Timers und der privaten Daten>
142   }                           // sonst, wenn der Motor noch läuft
143   else TIMSK&=  ~(1<<OCIE2);// Timer2-Compare-Interrupt aus,
144                               // aber keine Neuinitialisierung
145   PI_Restzeit10ms = Zeit;     // Übergabe der Bewegungsdauer
146   PI_MA.sollInc = incA;       // und der Soll-Increments in
147   PI_MB.sollInc = incB;       // 10 ms Schritten / für 10 ms
148   startPWM();                 // PWM einschalten
149   TIMSK |= 1<<OCIE2;          // Timer2-Compare-Interrupt ein
    }
```

- Während der Bearbeitung von Daten einer ISR, hier der Restzeit, immer Interrupt-Freigabe für diese ISR sperren!

Wenn die Motorregelung aus ist

```
134  if (!(TIMSK & (1<<OCIE2))){ // Wenn Motorregelung aus
135    TCCR2 = 1<<WGM21 | 0b011<<CS20; // Timer (neu-) initia-
136    OCR2 = 118;                     // lisieren: CTC, 1ms
137    clearPiStruc(&PI_MA, Motorzustand_A); // Motordaten neu
138    clearPiStruc(&PI_MB, Motorzustand_B); // initialisieren
139    PI_MA.SState = getSWA();
140    PI_MB.SState = getSWB();
141  } PI_Ct_ms = 0;
```

- Neuinitialisierung des Timer und aller Datenstrukturen, auch des Integralteils.
- Geschwindigkeitswechsel ohne Zwischenstopp erfordert Funktionsaufruf vor Ablauf der Restzeit:

```
if (getRestzeit()<=1)
    startBewegung(...);
```

Timer-ISR

```
153 ISR(TIMER2_COMP_vect){
154   PORTF ^=1;
155   PI_Ct_ms++;                       // Zähler erhöhen
156   QuadEnc(&PI_MA, getSWA());  //Wegmessung Rad A
157   QuadEnc(&PI_MB, getSWB());  //Wegmessung Rad B
158   if (PI_Ct_ms==10){                // alle 10 ms
159     Regler(&PI_MA);                 // Schrittfunktionen
160     Regler(&PI_MB);                 // der Regeler ausführen
161     ErzeugeAusgabe(&PI_MA, PI_buf);   // Testausgabe
162     ErzeugeAusgabe(&PI_MB, PI_buf+2); // erzeugen
163     sendUSART0(PI_buf, 4, FIXSIZE|NONBLOCKING);
164     setPWM(PI_MA.stellw, PI_MB.stellw);// PWM einstellen
165     PI_Ct_ms = 0;                   // Zähler rücksetzen
166     if (PI_Restzeit10ms>0)          // wenn noch Restzeit
167       PI_Restzeit10ms--;            // Restzeitzähler verringern
168     else                            // sonst
169       stopMotorregelung();          // Motorregelung ausschalten
  }
}
```

Hilfsfunktionen

```
173   // Ändern der Reglerparameter
174 - void setReglerparameter(uint16_t kp, uint16_t ki){
175     PI_kp = kp;
176     PI_ki = ki;
177   }
```

Suche geeigneter Parameterwerte für den Regler:

- ki = 0 setzen und kp soweit erhöhen, dass die Regelung schwingt.
- kp auf 60% bis 80% davon verringern.
- ki soweit hochsetzen, dass spätestens eine Sekunde nach Start kein Schleppfehler mehr da ist.

Weitere Hinweise finden Sie mit google unter dem Stichwort »empirischer Reglerentwurf«.

```
180 int16_t getRestzeit(){
181   TIMSK &= ~(1<<OCIE2);// Timer2-Compare-Interrupt aus
182   uint16_t t = PI_Restzeit10ms;
183   TIMSK |= (1<<OCIE2);// Timer2-Compare-Interrupt an
184   return t;
185 }
```

- Während des Lesens der Restzeit Interrupt-Freigabe aus, sonst kann der Rückgabewert sehr merkwürdig sein.

```
187  //Motorregelung anhalten
188 void stopMotorregelung(){
189   TIMSK &= ~(1<<OCIE2);// Timer2-Compare-Interrupt aus
190   stopPWM();           // PWM ausschalten
191   setState('.', Motorzustand_A);
192   setState('.', Motorzustand_B);
193   TCCR2  = 0;   // Timer aus; Stelle für den Unterbre-
194 }               // chungspunkt für den Reglertest
```

- Beim Debuggen des Reglers empfiehlt es sich, die Motoren vor Programmunterbrechung mit dieser Funktion anzuhalten.

Steuerschrittfkt, Kommandointerpreter

Steuerschrittfunktion

```
20 void Steuerschrittfunktion(uint16_t VAbst){
21   // Bewegungsabbruch bei Hindernis
22   if (VAbst>2000){
23     stopMotorregelung();
24     setState('A', Motorzustand_A);
25     setState('A', Motorzustand_B);
26   }
27   startLCD();
28 }
```

Die Steuerschrittfunktion stoppt bei einem Hindernis vor dem Abstandssensor die Motorregelung und setzt die Motorzustände auf 'A'.

Das Hauptprogramm

```
33 int main(void){
34    initUSART0();
35    // LCD- und ADC-Initialisierungen wie Vorprojekt
36    // Das ADC-Objekt bekommt einen Zeiger auf die Steuer-
37    initADC(&Steuerschrittfunktion); // schrittfunktion
38    // die als letztes die nächste LCD-Ausgabe startet.
39    // Das LCD-Objekt bekommt einen Zeiger auf die
40    initLCD(&startADC); // Funktion zum Start der Messungen
41    startADC();      // Start der ersten Messung
42    sei();           // globale Interruptfreigabe
43    while(1) {
44      getCommand();
          <Kommandointerpreter>
61    }
62    sleep(10);  // Verlängert Kommandoanzeige um 1 s
48 }
```

Kommandointerpreter in der Hauptschleife

```
44  getCommand();
45  switch (getState(Eingabezustand)){
46    case 'a':         // Bewegung starten
47      stopMotorregelung();
48      getUSART0(buff, 6, FIXSIZE | BLOCKING);
49      uint16_t A = (buff[0]<<8) + buff[1];
50      uint16_t B = (buff[2]<<8) + buff[3];
51      uint16_t T = (buff[4]<<8) + buff[5];
52      startBewegung((int16_t)A, (int16_t)B, T);
53      break;
54    case 'k':         // Reglerparameter übergeben
55      getUSART0(buff, 4, FIXSIZE | BLOCKING);
56      setReglerparameter((buff[0]<<8) + buff[1],// kp
57          (buff[2]<<8) + buff[3]);
58      break;
59    case 's':         // Bewegung stoppen
60    stopMotorregelung();
```

Regler mit Matlab testen

Aufgabe 8.4: Matlab starten

- Schließen Sie im HTerm die Verbindung. Laden Sie die m-Scripte zum Projekt von der Web-Seite.
- Starten Sie Matlab und in Matlab TestPI_Regler.m.
- Es sollte eine Bewegung ausgeführt und diese tabellarisch und graphisch angezeigt werden.
- Ändern Sie mit dem Skript SetzeRegelerparameter.m die Regelerparameter und schauen Sie anhand der Graphik wie sich das Reglerverhalten ändert.
- Optimieren Sie nach den Hinweisen auf Folie 52 die Reglereigenschaften.

Programm zur Änderung der Regelerparameter

```
 1 kp = 500;       % Initialwert im Mikrokontroller
 2 ki = 800;       % kp=500, ki = 800
 3
 4 s = serial('COM9','BAUD',9600);
 5 set(s,'Timeout',1000); % Timeout 1000s
 6 fopen(s);
 7 % wenn der Prozessor im Debugger hält SIO bei Absturz
 8 % freigeben. Alternative Neustart von Matlab nach
 9 closeFID = onCleanup(@() fclose(s)); % jedem Fehler
10
11 fwrite(s, 'k');                % Kommado Reglerparameter
12 fwrite(s, floor(kp/256));     % Sende Byte 1 von kp
13 fwrite(s, bitand(kp, 255)); % Sende Byte 0 von kp
14 fwrite(s, floor(ki/256));     % Sende Byte 1 von ki
15 fwrite(s, bitand(ki, 255)); % Sende Byte 0 von ki
```

Untersuchung des Reglerverhaltens mit Matlab

```
 4 % Bewegungsdauer und Geschwindigkeiten
 5 tz = uint16(400);% Bewegungsdauer mal 10 ms
 6 % Geschwindigkeit in 256stel Sensorschritten pro
 7 vA =  500;       % 10 ms, der Motor schafft von
 8 vB = -500;       % -2000 bis 2000 256stel Sensor-
 9                                    schritte/10ms
10 s = serial('COM9','BAUD',9600);
11 set(s,'Timeout',1000); % Timeout 1000s
12 fopen(s);                % serielle Verbindung herstellen
13 % wenn der Prozessor im Debugger hält SIO bei Absturz
14 % freigeben. Alternative Neustart von Matlab nach
15 closeFID = onCleanup(@() fclose(s)); % jedem Fehler
```

```
17 %Geschwindigkeits- und Zeitwert versenden
18 fwrite(s, 'a');              % Kommandobyte
19
20 if vA<0 vAu16=uint16(2^16+vA);   % 2er-Kompl.
21 else    vAu16=uint16(vA); end;   % Umformung
22 fwrite(s, bitshift(vAu16, -8)); % va Byte 1
23 fwrite(s, bitand(vAu16, 255));  % va Byte 0
24
25 if vB<0 vBu16=uint16(2^16+vB);   % 2er-Kompl.
26 else    vBu16=uint16(vB); end;   % Umformung
27 fwrite(s, bitshift(vBu16, -8)); % vb Byte 1
28 fwrite(s, bitand(vBu16, 255));  % vb Byte 0
29
30 fwrite(s, bitshift(tz, -8));     % tz Byte 1
31 fwrite(s, bitand(tz, 255));      % tz Byte 0
```

```
33  istA(1) = 0;                % Vektoren, zur Speicherung
34  sollA(1)= 0;                % der Ist- und Sollwerte
35  istB(1) = 0;                % der Motorpositionen
36  sollB(1)= 0;                % in Sensorschritten
37  pwmMA(1)= 0;                % PWM-Werte der
38  pwmMB(1)= 0;                % Motoren
39
40  fprintf('\n-----------------------------------
41  fprintf('Zeit| incA istA  sollA pwmA| incB istB  sollB
42  for t =2:tz
43      dat=fread(s,4);    % 4 Byte vom Mikrorechner lesen
44      if length(dat)<4, ... % Falls keine Daten
45              return, end % Abbruch
```

```
47 % Aufspalten der Bytes in seinen Bestandteile
48 incA    = bitand(dat(2), 63); % Ist-Schritte Motor A
49 if bitget(dat(2),7) incA=incA-64; end;
50 pwmA = dat(1)*2;                % PWM-Wert Motor A
51 if bitget(dat(2),8) pwmA=pwmA+1; end;
52 if bitget(dat(1),8) pwmA=pwmA-512; end;
53 %fprintf('incA=%x, pwmA=%x\n', incA, pwmA);
54
55 incB    = bitand(dat(4), 63); % Ist-Schritte Motor B
56 if bitget(dat(4),7) incB=incB-64; end;
57 pwmB = dat(3)*2;                % PWM-Wert Motor B
58 if bitget(dat(4),8) pwmB=pwmB+1; end;
59 if bitget(dat(3),8) pwmB=pwmB-512; end;
```

```
61    istA(t) = istA(t-1) + single(incA);    % Ist-Pos. A
62    sollA(t)= sollA(t-1)+ single(vA)/256; % Soll-Pos. A
63    pwmMA(t)= pwmA;                        % PWM Motor A
64    istB(t) = istB(t-1) + single(incB);    % Ist-Pos. B
65    sollB(t)= sollB(t-1)+ single(vB)/256; % Soll-Pos. B
66    pwmMB(t)= pwmB;                        % PWM Motor B
67
68    % tabellarische Ausgabe
69    fprintf('%4i | %4i %4i %7.2f %4i ', t, ...
70        incA, istA(t), sollA(t), pwmA);
71    fprintf('|%4i %4i %7.2f %4i|\n', incB, ...
72        istB(t), sollB(t), pwmB);
```

```
73 end;
74 tv = ((1:tz)-1)*10;            % Vektor der Zeitwerte in ms
75 % graphische Ausgabe
76 plot(tv, sollA, 'r', tv, istA, 'g', tv, pwmMA, 'b', ...
77     tv,sollB, 'k', tv, istB, 'm', tv, pwmMB, 'c');
78
79
80 xlabel('Zeit in Millisekunden')
81 ylabel('Weg in Sensorschritten');
82 title('rot: SollA, grün: IstA, blau: pwmA, gelb: sollB, r
83 grid on;
84
85 fclose(s);          % serielle Schnittstelle schließen
```

Graphische Programmausgabe