Informatikwerkstatt, Foliensatz 6
Zeitabläufe und Schrittketten

G. Kemnitz

Institut für Informatik, Technische Universität Clausthal

12. Dezember 2013

Inhalt des Foliensatzes

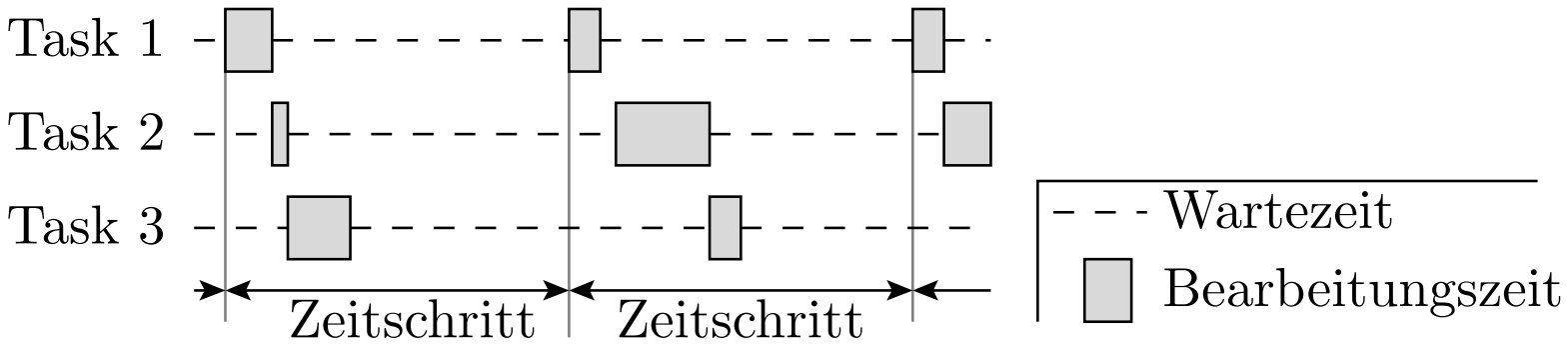
Mehrere Tasks

Mehr-Task-Programme

In der Steuer- und Automatisierungstechnik, auch bei unserem Fahrzeug, muss der Mikrocontroller mehrere Aufgaben (Tasks) quasi zeitgleich ausführen:

- eine LED blinken lassen
- auf Zeichen warten
- Sensorwerte abfragen, ...

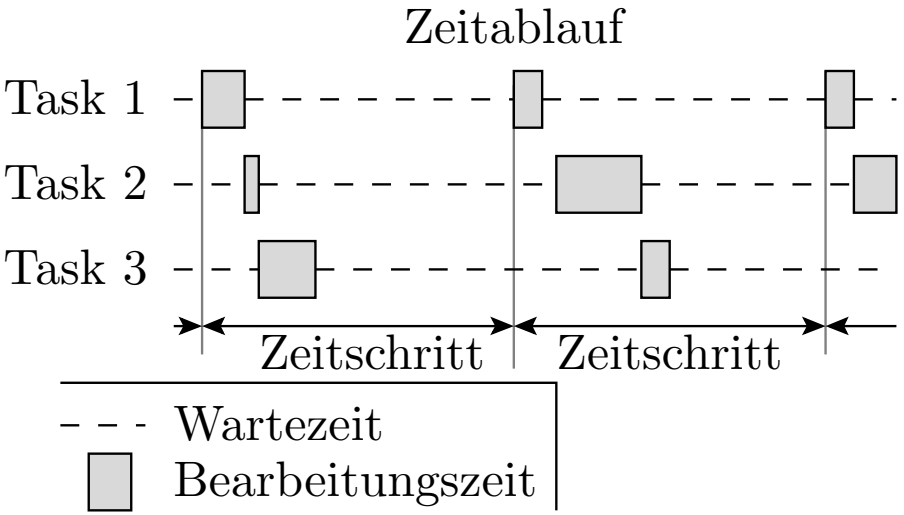
Bei jeder dieser Aufgaben wird die meiste Zeit gewartet. Die nachfolgende Abbildung zeigt ein einfaches Taskscheduling[1]:

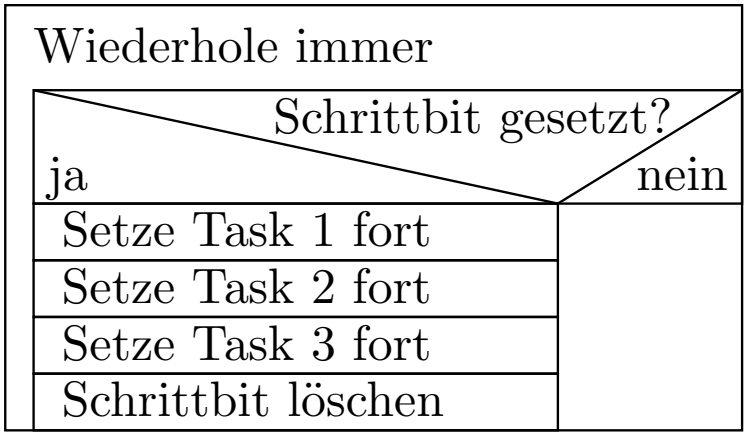


[1] Zyklische, zeitschrittgesteuerte Abarbeitung von als Schrittketten programmierten Tasks.

Programmtechnische Umsetzung

- Timer, der immer nach der Schrittzeit ein Bit setzt.
- Als Schrittketten[2] programmierte Tasks.
- Endlosschleife, die auf das Schrittbit wartet.
- Wenn gesetzt, sequentielle Abarbeitung des nächsten Schritts aller Tasks und Schrittbit löschen.



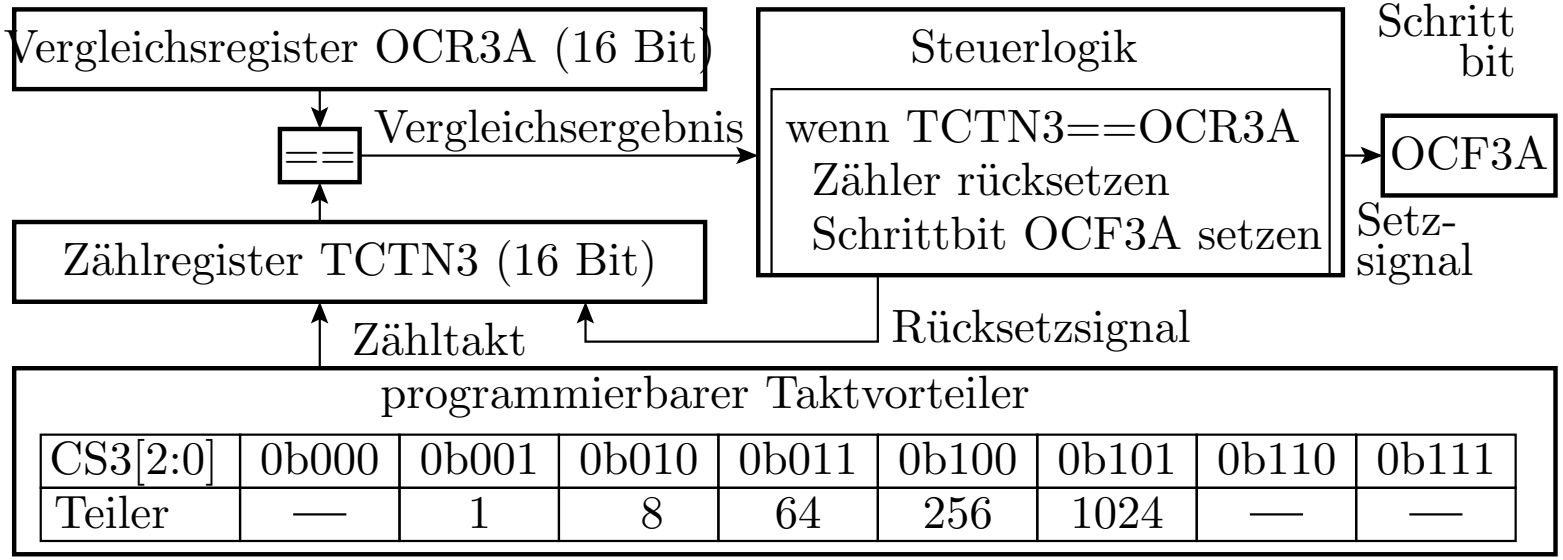


[2] Funktionen mit einer Zustandsvariablen, die bei Aufruf einen Schritt abarbeiten, den Zustand weiterschalten und den Kontrollfluss zurück geben.

Timer einrichten

Timer

Ein Timer ist eine Hardware-Einheit aus mehreren Zähl-, Vergleichs- und Konfigurationsregistern. Als Schrittkettenzeitgeber soll Timer 3 im CTC- (clear timer on compare) Modus mit OCR3A als Vergleichsregister genutzt werden (WGM0[3:0]=0b0100):



CS3[2:0]	0b000	0b001	0b010	0b011	0b100	0b101	0b110	0b111
Teiler	—	1	8	64	256	1024	—	—

- Vergleichswert:

$$\mathrm{OCR3} = t_\mathrm{s} \cdot f_\mathrm{CPU}/vt$$

(t_s – Schrittzeit; $f_\mathrm{CPU} \approx 7{,}58\,\mathrm{MHz}$ (experimentell bestimmt).

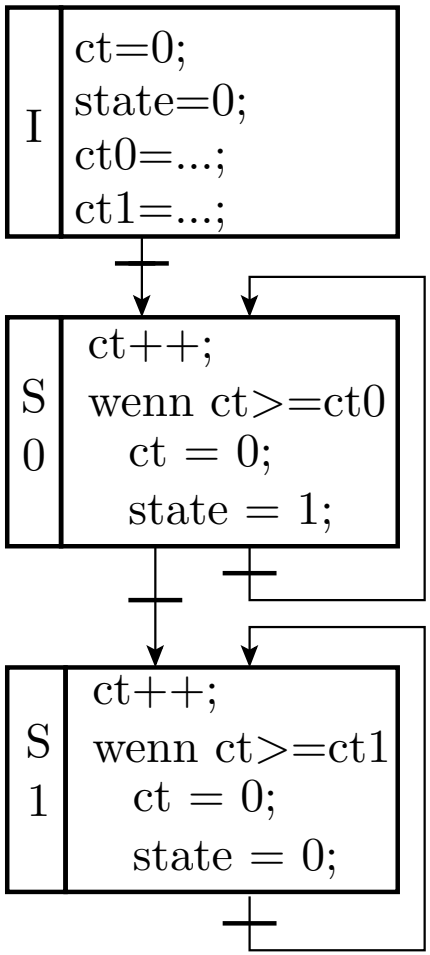
2. Timer einrichten

- Das Schrittbit OCF3 ist nach Abschluss aller Task-Schritte durch Schreiben einer Eins zu löschen.

```
void initTmr3(uint32_t ts){
  ts=ts*750;                  // Takte je Schrittzeit
  uint8_t cs;                 // Vorteilereinstellung
  if (ts>0x3FFFFFF){          // wenn vT=1024 nicht reicht
  ts=0xFFFF;  cs = 0b101;}// max. Schrittzeit ca. 9s
  else if (ts>0xFFFFFF){  // sonst wenn vt=256 nicht reicht
  ts=ts>>10; cs = 0b101;} // Vorteiler: vt=1024
  else if (ts>0x3FFFFF){  // sonst wenn vt=64 nicht reicht
  ts=ts>>8; cs = 0b100;}  // Vorteiler: vt=256
  else if (ts>0x7FFFF){   // sonst wenn vt=8 nicht reicht
  ts=ts>>6; cs = 0b011;}  // Vorteiler: vt=64
  else if (ts>0xFFFF){    // sonst wenn vt=1 nicht reicht
  ts=ts>>3; cs = 0b010;}  // Vorteiler: vt=8
  else       cs = 0b001;  // sonst Vorteiler: vt=1
  TCCR3B = (1<<WGM32)|(cs<<CS30);
  OCR3A  = ts;                // Vergleichswert einstellen
} ETIFR = 1<< OCF3A;          // Schrittbit löschen
```

Nebenläufig blinkende LEDs

Schrittkettenbeschreibung einer blinkenden LED



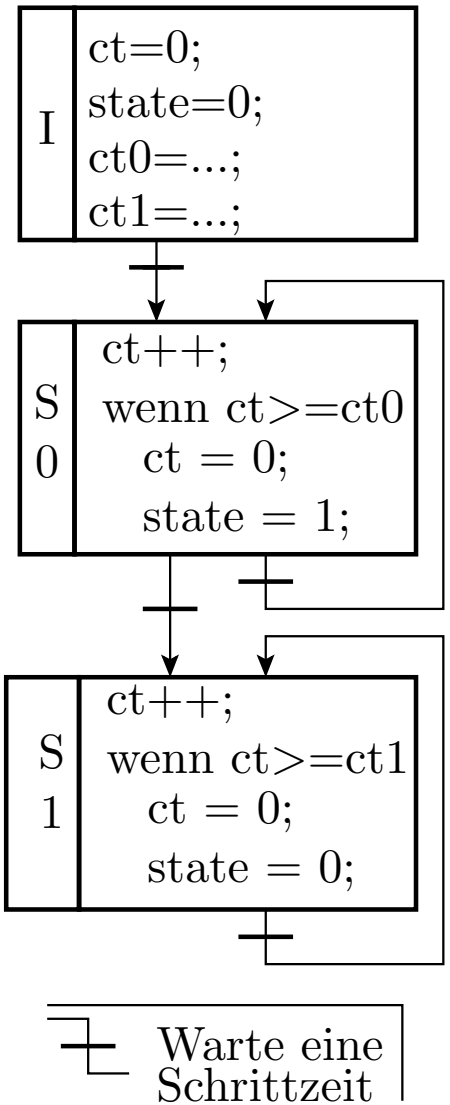
Vereinbarungen in der Header-Datei LED_Task.h

```
// Datenstruktur für ein Task-Objekt
struct LED_Task_t{
    uint8_t state;      // Zustand des Tasks
    uint8_t ct;         // Zähler (counter)
    uint8_t ct0, ct1;   // Aus- und Anzeit
};                      // in Schritten

// Initialisierungsfunktion
void LED_Task_init(uint8_t initCt0,
     uint8_t initCt1, struct LED_Task_t* tp);

// Schrittfunktion
uint8_t LED_Task_Step(struct LED_Task_t *tp);
```

Warte eine Schrittzeit

- Die Variablen »state« zeigt auf den auszuführenden Schritt.

Programmieren der Schrittfunktion



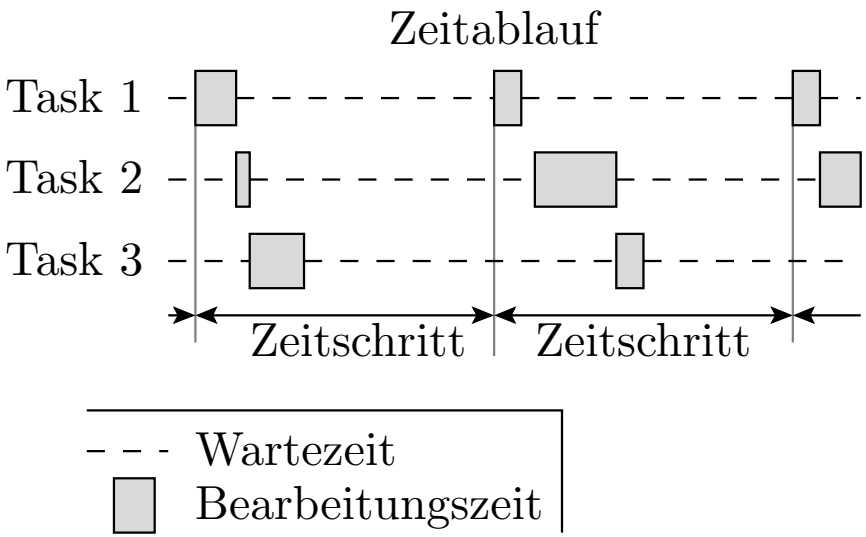

```
void LED_Task_init(uint8_t initCt0,
     uint8_t initCt1, struct LED_Task_t* tp){
    tp->state=0;
    tp->ct =0;
    tp->ct0=initCt0;
    tp->ct1=initCt1;
}

uint8_t LED_Task_Step(struct LED_Task_t *tp){
    tp->ct++;
    switch (tp->state){
        case 0: if (tp->ct >= tp->ct0){
            tp->state = 1; tp->ct = 0;
            }
            break;
        case 1: if (tp->ct >= tp->ct1){
            tp->state = 0; tp->ct = 0;
            }
    }
    return tp->state;
}
```

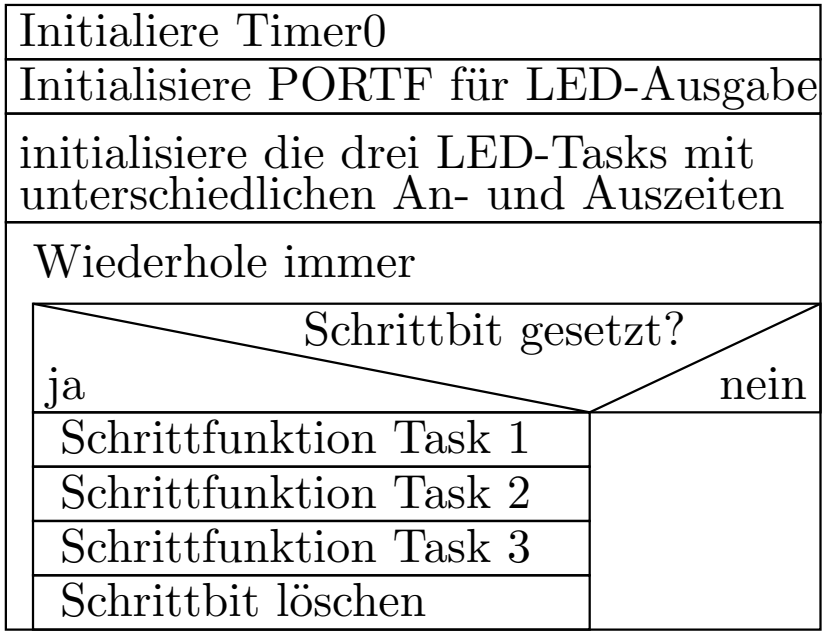
- Die Init-Funktion eines Tasks weist den Task-Variablen ihre Anfangswerte zu.
- Die Step-Funktion hat immer als äußeren Programmrahmen eine Fallunterscheidung nach dem nächsten auszuführenden Schritt.
- Jeder »case« außer dem letzten muss mit einer Break-Anweisung enden. Sonst werden die Befehle des nächsten »case« mit ausgeführt.
- Rückgabewert ist im Beispiel der Zustand 0 oder 1, der hier gleichzeitig der an die LED auszugebende Wert ist.

Neue Programmierelemente:

...(struct LED_Task_t *tp)	Übergabeparameter ist die Adresse des Task-Objekts, bestehend aus Zustand, Zähler, ...
tp->state	Auswahl eines Elements des Task-Objekts, hier des Zustands.

Das Hauptprogramm





Das Hauptprogramm umfasst Vereinbarungen der allgemeinen Variablen und der Task-Objekte, die Initialisierungen und eine Endlosschleife, in der in jedem Zeitschritt die Schrittfunktionen der Tasks aufgerufen werden.

Das Hauptprogramm:

```
struct LED_Task_t a, b, c;
int main(void) {
  initTmr3(100);              // Schrittzeit 10 ms
  DDRE |= 0b01110000;         // Port E[6:4] als LED-Ausgabe
  uint8_t tmp;
  // Initialisiere 3 Tasks
  LED_Task_init( 50,  50, &a); // Task 1 initialisieren
  LED_Task_init(150,  75, &b); // Task 2 initialisieren
  LED_Task_init(120, 150, &c); // Task 3 initialisieren
  while(1){
    if (ETIFR & (1<< OCF3A)) {  //wenn Schrittbit gesetzt
      tmp  = LED_Task_Step(&a)<<4;// Task 1 abarbeiten
      tmp |= LED_Task_Step(&b)<<5;// Task 2 abarbeiten
      tmp |= LED_Task_Step(&c)<<6;// Task 3 abarbeiten
      PORTE = tmp;
      ETIFR = 1<< OCF3A;          // Schrittbit löschen
    }
  }
}
```

&a – Übergabe der Adresse des Datenobjekts a

Aufgabe 6.1: Beispiel testen

- Legen Sie ein neues Projekt »LED_Task« an und binden Sie die drei Dateien LED_Task.h, LED_Task.c und LED_Schrittkette.c von der Web-Seite in das neue Projekt ein.
- Übersetzen mit Compiler-Optimierung auf »O0« und Test im Debugger. Untersuchen Sie, welche Programmstellen günstig für Unterbrechungspunkte sind.

Aufgabe 6.2: Zeitablauf tabellarisch erfassen

Bestimmen Sie für jeden Schaltschritt, in dem sich ein Zustand ändert (eine LED schaltet), die Schrittnummer sowie Zählerstand und Zustand aller drei Tasks. Fortführung nachfolgender Tabelle[3]:

Schritt	a.state	a.Ct	b.State	b.Ct	c.state	c.Ct
0	0	0	0	0	0	0
50	1	50	0	50	0	50
100	0	50	0	99	0	99
120	0	20	0	120	1	120

[3] In der Tabelle stehen die Sollwerte der ersten Schaltschritte für a.Ct0=50, a.Ct1=50, c.Ct0=120, ...

Lösungsempfehlungen:

- Vereinbaren Sie im Programm eine zusätzliche globale Zählvariable »dbgCt« für die Schritte seit Programmstart, die mit null initialisiert und in der Hauptschleife in jedem Zeitschritt um eins erhöht wird.
- Setzen Sie beim Debuggen in der Schrittfunktion nach »tp->state=0« / »tp->state=1« und vor »tp->ct=0;« je einen Unterbrechungspunkt.
- Richten Sie die Debug-Fenster wie auf der Folgefolie ein[4]:
  - Watch 1 soll alle Zähler und Zustände anzeigten.
  - Local soll den Wert von tp anzeigen, um den Task, der bearbeitet wird, zu erkennen.

[4] Bildschirmfoto nach 120 Hauptschleifendurchläufen am Haltepunkt nach »pt->state=1« im Unterprogramm LED_Task_step(), aufgerufen mit pt=0x104 (Task c).

3. Nebenläufig blinkende LEDs

Watch 1

Name	Value	Type
a	{struct LED_Task_t{data}@0x010a}	struct LED_Task_t{data}@0x010a
state	0	uint8_t{data}@0x010a
ct	20	uint8_t{data}@0x010b
ct0	50	uint8_t{data}@0x010c
ct1	50	uint8_t{data}@0x010d
b	{struct LED_Task_t{data}@0x0100}	struct LED_Task_t{data}@0x0100
state	0	uint8_t{data}@0x0100
ct	120	uint8_t{data}@0x0101
ct0	150	uint8_t{data}@0x0102
ct1	75	uint8_t{data}@0x0103
c	{struct LED_Task_t{data}@0x0104}	struct LED_Task_t{data}@0x0104
state	1	uint8_t{data}@0x0104
ct	120	uint8_t{data}@0x0105
ct0	120	uint8_t{data}@0x0106
ct1	150	uint8_t{data}@0x0107
dbgCt	120	uint16_t{data}@0x0108

Locals

Name	Value	Type
tp	0x0104	struct LED_Task_t*{data}@0x10f5 ([R28]+1)

Aufgabe 6.3: Prüffreundlich programmieren

Schreiben Sie das Programm nach dem Schema auf der nachfolgenden Folie so um, dass

- jeder LED-Task seine eigene Init- und Step-Funktion hat
- damit keine Parameter zu übergeben sind
- das Programm ohne Zeiger auskommt und
- und die Unterbrechungspunkte für die Ablaufkontrolle taskweise gesetzt werden können.

Wiederholen Sie die Tests aus Aufgabe 6.2 und schätzen Sie ein, wie sich der Debug-Aufwand für diese Programmversion zu der mit gemeinsam genutzten Unterprogrammen für alle Tasks in Aufgabe 6.2 verhält.

- Vereinbarung eines Variablensatzen für jeden Task:

```
uint8_t a_state, b_state, c_state;
uint8_t a_Ct0, b_Ct0, c_Ct0; ..._
```

- Definition der An- und Auszeit innerhalb der Init-Funktionen:

```
LED_TaskA_init(){
 a_state =0; a_Ct0=50; ...
}
```

- Task-zugeordnete Unterbrechungspunkte

```
LED_TaskA_step(){
 a.Ct++;
 switch (a.state){
  case 0: if (a_Ct>=a_Ct0) {
    a_state=1;
(*) a_Ct=0; // (*) Unterbrechungspunkt 01-Wechsel LED
    }
    break;
  ...
```

Aufgabe 6.4: Blinkzeichenausgaben

Elektronische Geräte signalisieren oft Zustände, vor allem Fehlerzustände, mit Blink- oder Piepzeichen. Entwickeln Sie ein Programm, das auf einen Bytewert von der seriellen Schnittstelle wartet und ihn als Blinksequenz auf LD1 (Port E, Bit 7) ausgibt:

- Beginn mit dem niedrigsten Bit
- Ausgabe einer Eins: 0,3 s aus und 0,9 s an
- Ausgabe einer Null: 0,3 s aus und 0,3 s an.

Ein- und Ausgabe als Task

Das Echo-Programm von Foliensatz 2

- Serielle Schnittstelle an JD mit 9600 Baud, 8 Daten-, 1 Stoppbit und keine Parität initialisieren:

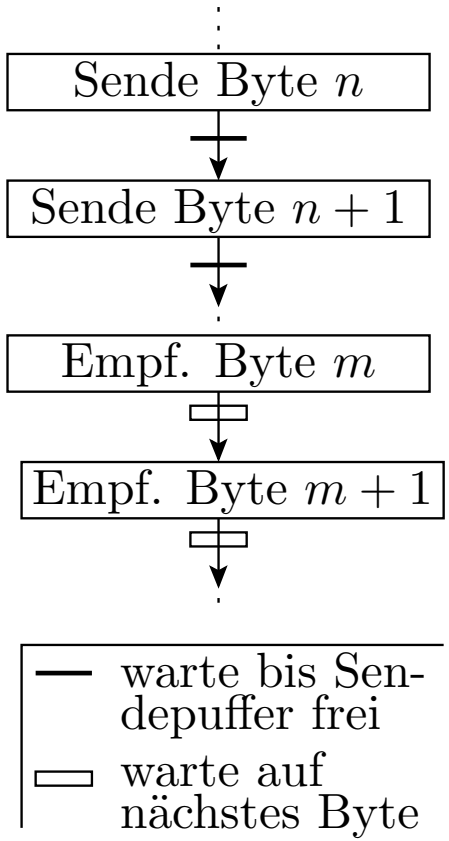
```
UBRR0L = 49;           UBRR0H = 0;
UCSR0B = 0b00011000; UCSR0C=0b0011100;
```

- Hauptschleife:

```
while (1){
 while (!(UCSR0A & (1<<RXC00))){} // warte auf Empfang
 daten = UDR0;                    // Daten übernehmen
 while (!(UCSR0A & (1<<UDRE0))){} // warte bis
                                  // Sendepuffer frei
 UDR0 = daten;                    // Daten senden
}
```

Für die Programmierung als Tasks ist das Warten auf Empfang und »Sendepuffer frei« durch Schrittübergänge nachzubilden.

Ein- und Ausgabe als Schrittkette



Randbedingungen:

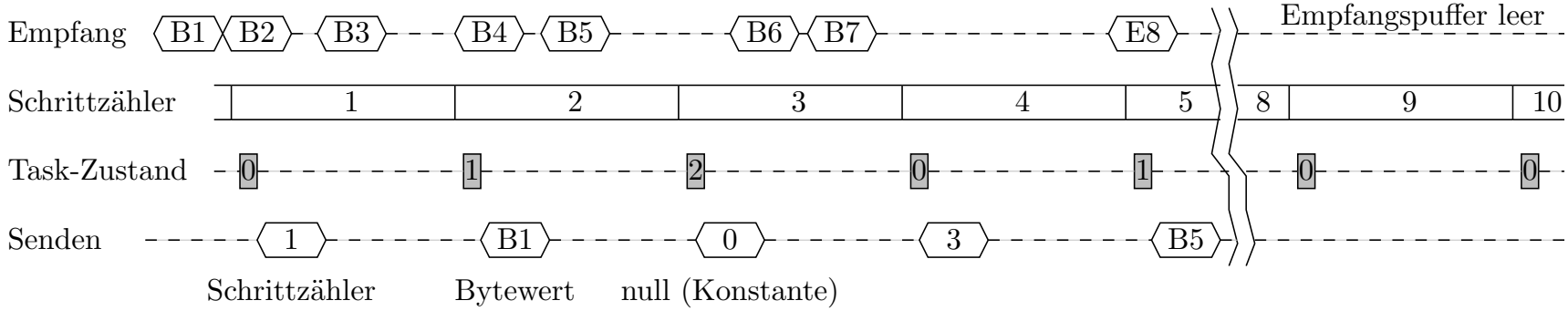
- Dauer einer Byte-Übertragung bei 9600 Baud, 8 Daten-, 1 oder 2 Stopp- und 0 oder 1 Paritätsbit:

$$t_{\text{byte}} = \frac{10 \ldots 12\,\text{Bit}}{9600\,\frac{\text{Bit}}{\text{s}}} = 1{,}04 \ldots 1{,}25\,\text{ms}$$

- Bei Schrittzeit $> 1{,}25\,\text{ms}$ ist der Sendepuffer in jedem Schritt frei, Empfangsbytes können verloren gehen.
- Bei Schrittzeit $< 1{,}04\,\text{ms}$ gehen keine Empfangsbytes verloren, können aber möglicherweise nicht alle empfangenen Bytes im selben Schritt zurückgesendet werden.
- Auch im ungünstigsten Fall müssen die Schrittoperationen aller Tasks innerhalb der Schrittzeit abgeschlossen werden.

Ein Experiment

Für jedes empfangene Byte sollen drei Bytes gesendet werden, Schrittzählerwert, Bytewert, Abschlussnull.



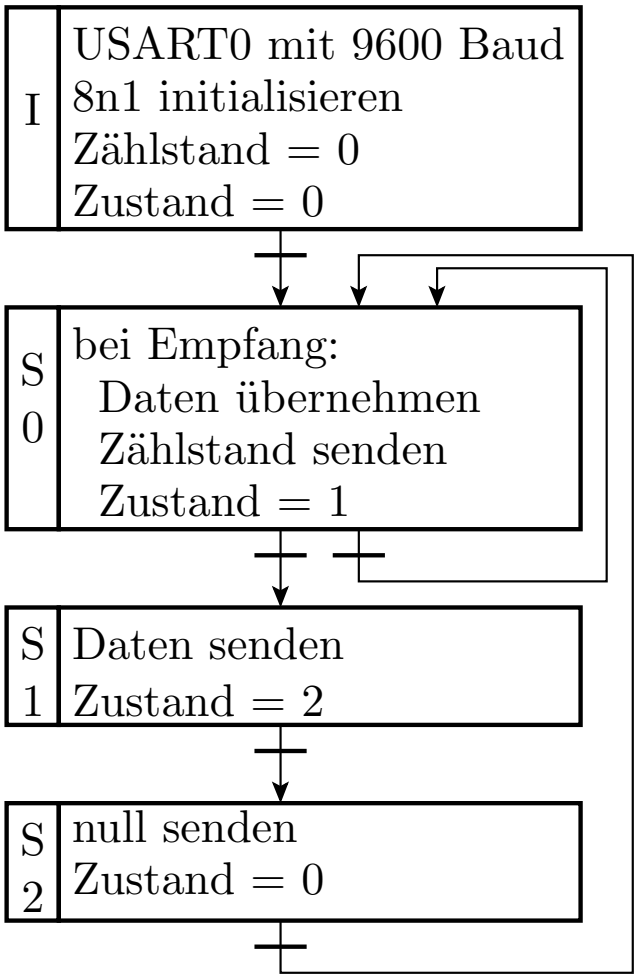
- Auf welche der vom PC gesendeten Bytes wartet der Mikrokontroller in Abhängigkeit von der Schrittzeit?
- Bis zu wie viele Bytes können zwischen zwei empfangenen Bytes verloren gehen?
- Wie lange muss der PC zwischen dem Versenden von Bytes mindestens warten, damit garantiert alle Bytes empfangen werden?

Aufgabe 6.5: Experiment vorbereiten

- Legen Sie ein neues Projekt »Echo_Schrittkette« an. Binden Sie die Dateien TaskFkt.h, TaskFkt.c und Echo_Schrittkette.c von der Web-Seite ein, übersetzten Sie mit Compiler-Optimierung »O0«.
- Anstecken des USART-USB-Modul an Stecker JD. (Spannung erst danach einschalten!)
- HTerm mit 9600 Baud 8n1 starten. Programm im Debugger starten.
- Einzelbyte eingeben und kontrollieren, dass drei Bytes zurückgesendet werden (Schrittzähler, Bytewert, Abschlussnull) zurückgegeben werden.

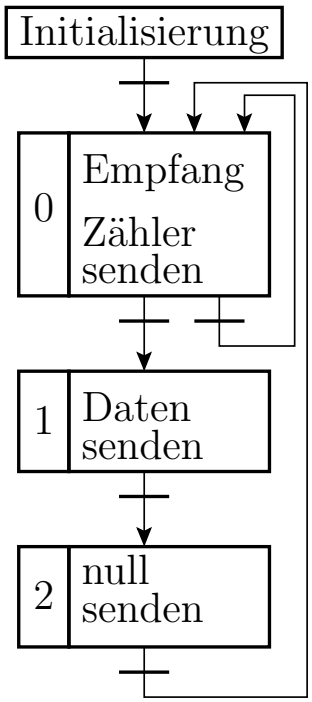
Besprechen des Programms

- Daten des Tasks und USART-Initialisierung.




```
// Daten des USART-Tasks
uint8_t USART_Daten;
uint8_t USART_state;
uint8_t USART_Ct;

// USART0 an JE 9600 Baud 8n1
void initUSART0(){
    UBRR0H=0; UBRR0L=49;
    UCSR0B = 0b00011000;
    UCSR0C = 0b00000110;
    USART_state=0;
    USART_Ct=0;
}
```

Schrittfunktion des Empfangs-und Sende-Task:




```
void USART_Task_step(){
    USART_Ct++;
    switch (USART_state){
        case 0:
            if (UCSR0A & (1<<RXC0)) { // bei Empfang
                USART_Daten = UDR0;
                UDR0 = USART_Ct;       // Zähler senden
                USART_state = 1;
            }
            return;
        case 1:
            UDR0 = USART_Daten;       // Daten senden
            USART_state = 2;
            return;
        case 2:
            UDR0        = 0;          // null senden
    }
    USART_state = 0;
}
```

- Eine blinkende LED lässt erkennen, ob das Programm arbeitet.

```
73  uint8_t LED_state;  // Zustand des Tasks
74  uint8_t LED_Ct;     // Zähler (counter)
75  uint8_t maxCt;      // maximaler Zählerwert
78  // ts: Schrittzeit in 0.1ms-Schritten
79  void LED_Task_init(uint32_t ts){
80    LED_state = 0;
81    LED_Ct = 0;
82    maxCt = (uint8_t)(5000/ts); // 5000*0.1ms
    }
86  uint8_t LED_Task_step(){// 1 Hz Blinksignal
87    LED_Ct++;
88    if (LED_Ct>=maxCt){   // alle 0,5 Sekunden
89      LED_state ^= 0x10;  // Bit 4 invertieren
90      LED_Ct = 0;         // Zähler rücksetzen
      }
92    return LED_state;
    }
```

- Die über das Header-File exportierten Funktionen:

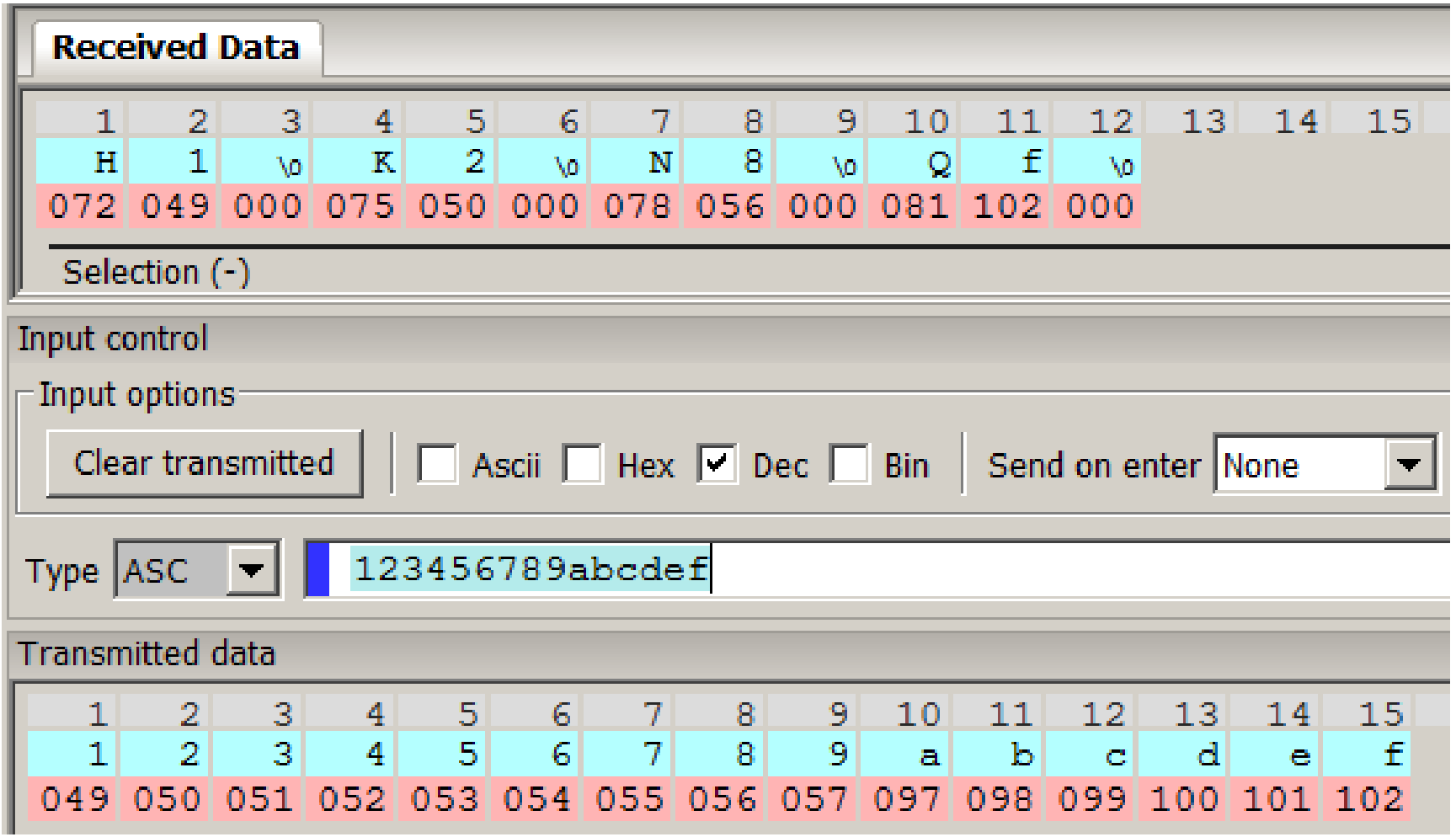
```
12  // Initialisierung USART0 an JE 9600 Baud 8n1
13  void initUSART0();

15  // USART0-Schrittfunktion
16  void USART_Task_step();

18  // Initialisieren von Timer 3
19  // ts: Schrittzeit in 0.1ms-Schritten
20  void initTmr3(uint32_t ts);

22  // Initialisierung LED-Task
23  // ts: Schrittzeit in 0.1ms-Schritten
24  void LED_Task_init(uint32_t ts);

26  // Schrittfunktion LED-Task
27  uint8_t LED_Task_step();
```

- Init-Funktion für den Timer wie im Projekt davor.

- Das Hauptprogramm

```
 9  #include <avr/io.h>
10  #include "TaskFkt.h"
11
12  int main(void) {
13    DDRE = 0x10;                    // LD4 als Ausgang
14    initUSART0();
15    initTmr3(100);                  // Initialisierung
16    LED_Task_init(100);             // für 10ms Schritte
17    while(1) {
18      if (ETIFR & (1<< OCF3A)) {// wenn Schrittzeit um
19        USART_Task_step();          // Tasks einen
20        PORTE = LED_Task_step();// Schritt weiter
21        ETIFR = 1<< OCF3A;          // Schrittbit löschen
22      }
23    }
24  }
```

Testbeispiel mit dem HTerm



Aufgabe 6.6: Experimentiervorschläge

- Verwenden Sie die Terminaleinstellungen und Eingaben der Folie zuvor. Erschließen Sie sich die Funktionsweise durch stückweise Abarbeitung mit Unterbrechungspunkten.
- Untersuchen Sie, wie viele gesendete Bytes im Mittel ignoriert werden, wenn
  1. 30 Bytes als Block gesendet werden
  2. 4 Bytes mit Wartezeit 10 ms, 20 ms und 30 ms

  vom PC gesendet werden.
- In welchem Zeitabstand werden die Bytes empfangen[5]?
- Wiederholen Sie die Experimente mit einer geänderten Schrittzeit von 1 ms.

[5] Nach Empfang rechts neben »Save Output« »Hex« auswählen und bei »Timestamp« Haken setzen. Mit »Save Output« in eine Datei schreiben und Datei mit Editor ansehen.

EA-Task für Bytefolgen

Anforderungen an die zu entwickelnden Bausteine

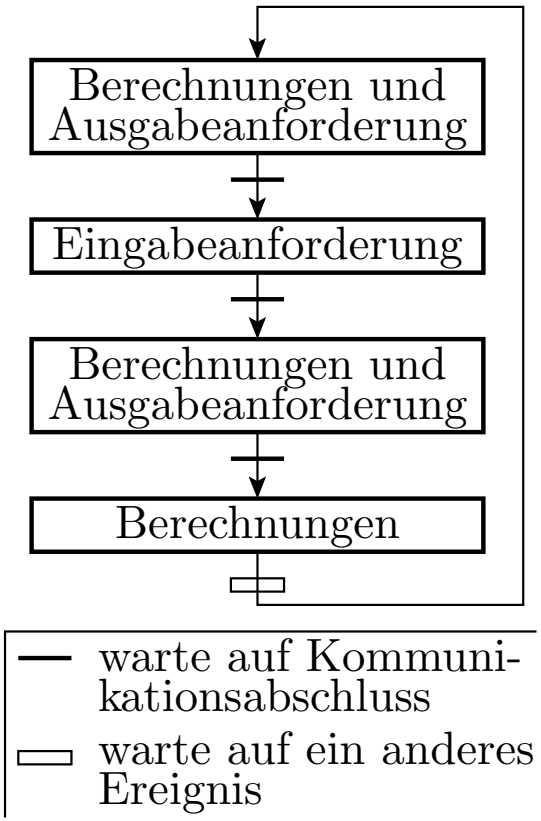
Das Programm für die Ein- und Ausgabe soll in der Lage sein:

- variable Texte ohne Zeichenverlust an den PC zu senden
- auf einen Text variabler Länge vom PC zu warten und diesen verlustfrei einzulesen.

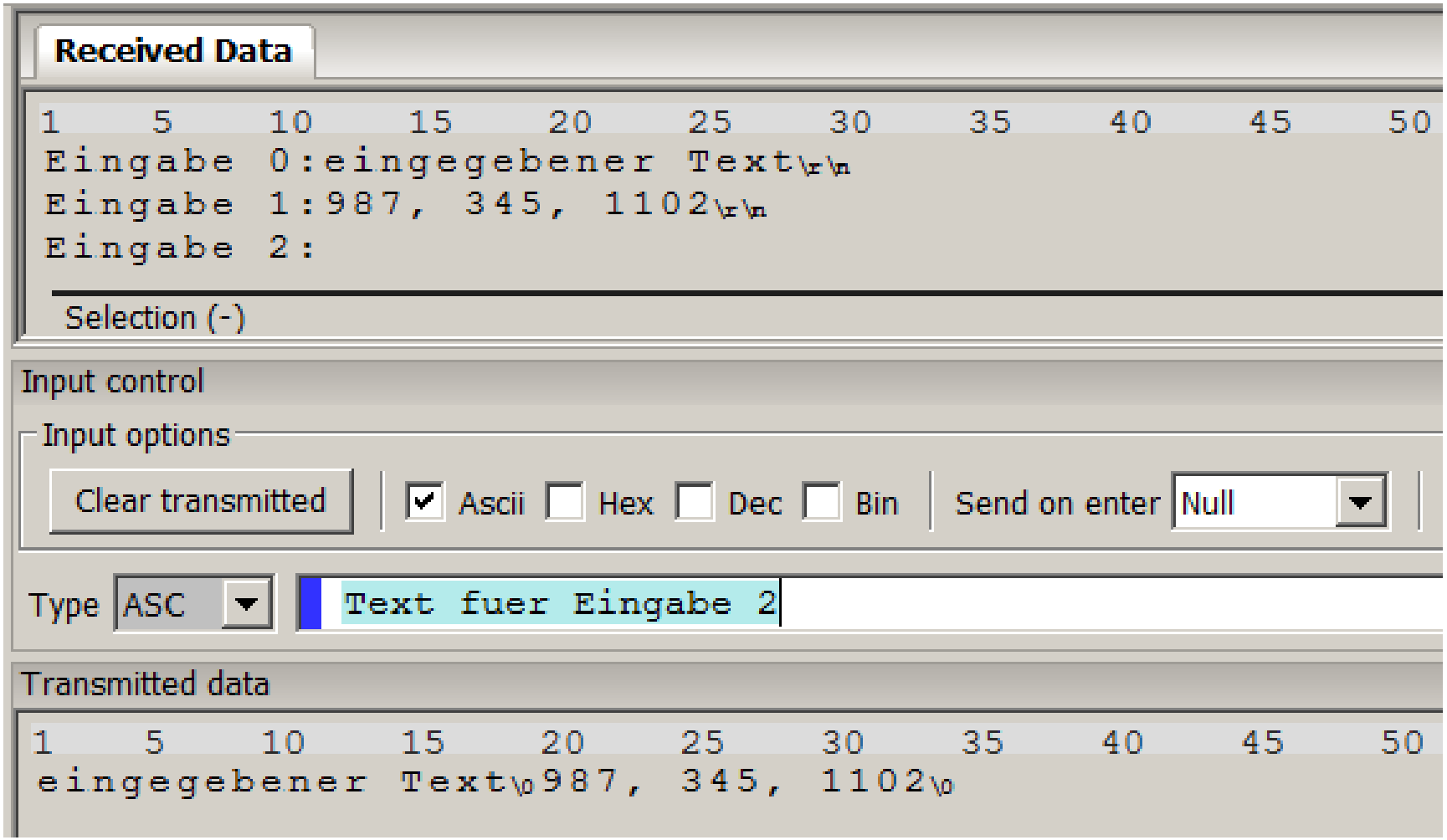
Das erfordert drei Schritttypen:

- Berechnung + Warten
- Berechnung + Ausgabe + Warten
- Eingabeanforderung + Warten

Nach Ein- und Ausgabe ist immer auf Abschluss der Kommunikation zu warten. Zur Ausgabe ist ein gefüllter Zeichenpuffer mit zu sendendem Inhalt und für die Eingabe ein leerer Puffer für die empfangenen Zeichen zu übergeben.



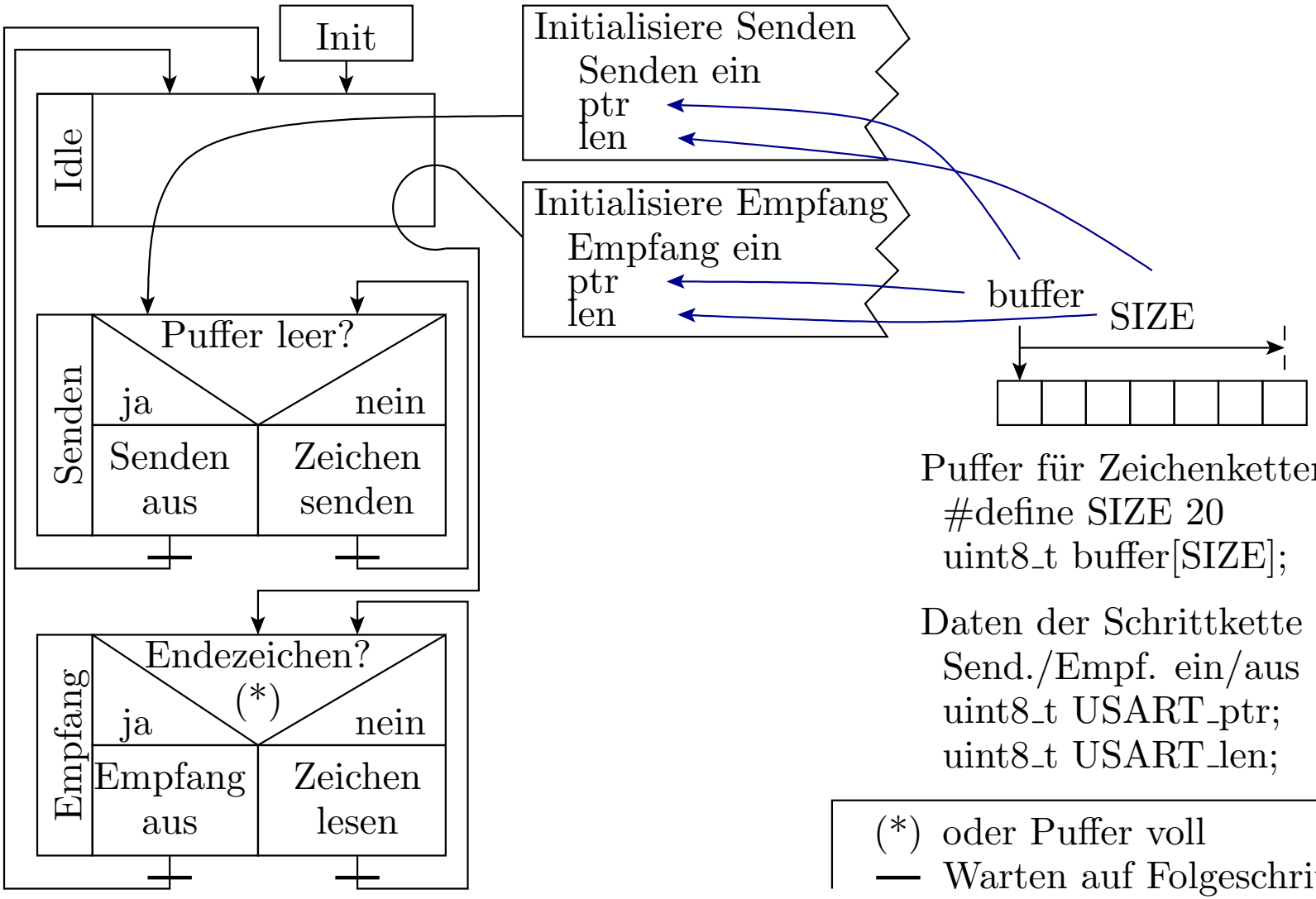
Gewünschter Ein-/Ausgabe-Dialog



Aufgabe 6.7: Beispiel vorbereiten

- Legen Sie ein neues Projekt »TestTaskIOStep« an und binden Sie die Dateien TaskIO.h, TaskIO.c, TestTaskIO.c, strg.c und strg.h von der Web-Seite in das neue Projekt ein.
- Starten Sie HTerm und nehmen Sie folgende Einstellungen vor:
  - 9600 Baud, 8 Datenbit, 1 Stoppbit, keine Parität, Show Errors
  - Newline at: CR+LF
  - Send on Enter: Null
  - Sende und Empfangsanzeige: nur Ascii
- Programm übersetzen. Im Debugger ohne Unterbrechungspunkte starten. Nach Eingabeanforderung Text eingeben + Enter. Soll-Ausgabe siehe Folie 36.

Entwurf der Schrittkette für die Kommunikation



Puffer für Zeichenketten

```
#define SIZE 20
uint8_t buffer[SIZE];
```

Daten der Schrittkette

```
Send./Empf. ein/aus
uint8_t USART_ptr;
uint8_t USART_len;
```

(*) oder Puffer voll

— Warten auf Folgeschritt

In TaskIO.h definierte Funktionen

```
15 // Funktionen für USART0 (PC-Kommunikation)
16
17 // Initialisiere USART0 an JE mit 9600 Baud 8n1
18 void initUSART0();
19
20 // Start einer Sendeoperation
21 void sendUSART0(uint8_t *ptr, uint8_t len);
22
23 // Schrittfunktion Senden
24 void stepSendUSART0();
25
26 // Start einer Empfangsoperation
27 void getUSART0(uint8_t *ptr, uint8_t len);
28
29 // Schrittfunktion Empfang
30 void stepGetUSART0();
```

```
33  // Funktionen zur Ablaufkontrolle
34
35  // Initialisieren von Timer 3
36  // ts: Schrittzeit in 0.1ms-Schritten
37  void initTmr3(uint32_t ts);
38
39  // Funktion zu Signalisierung eines neuen Zeitschritts
40  uint8_t NeuerZeitschritt();
41
42  // Initialisierung LED-Task
43  // ts: Schrittzeit in 0.1ms-Schritten
44  void LED_Task_init(uint32_t ts);
45
46  // Schrittfkt. zur Erzeugung eines 1 Hz Blinksignals
47  void LED_Task_step();
48
49  // Test, ob alle IO-Operationen abgeschlossen sind
50  uint8_t IO_ready();
```

- Die Initialisierungsfunktion für Timer 3 (InitTmr3()), die privaten Daten und die Initialisierung des LED-Tasks sind identisch mit denen aus dem Projekt zuvor.

```
130  // Schrittfkt. zur Erzeugung eines 1 Hz Blinksignals
131  void LED_Task_step(){
132    ETIFR = 1<< OCF3A;          // Schrittbit löschen
133    LED_Ct++;
134    if (LED_Ct>=maxCt){         // alle 0,5 Sekunden
135      PORTE ^= 0x10;            // LD4 invertieren
136      LED_Ct = 0;               // Zähler rücksetzen
137    }
138  }
```

- Die Schrittfunktion des Blink-Tasks setzt zusätzlich das Schrittbit zurück.
- Die Abfrage des Schrittbits wurde in eine Funktion gekapselt.

```
114  // Funktion zu Signalisierung eines neuen Zeitschritts
115  uint8_t NeuerZeitschritt(){return ETIFR & (1<< OCF3A);
```

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

- Das Einschalten der Sende und Empfangsfunktion erfolgt erst in den Initiierungsfunktionen für Senden und Empfang.

```
19  // Daten für USART0
20  uint8_t *USART0_ptr;  // Zeiger auf den nächte
21                        // Pufferplatz
22  uint8_t USART0_len;   // Länge bis Pufferende
```

- Auch die privaten globalen Daten werden erst in den Initiierungsfunktionen initialisiert.

```
25  // Start einer Sendeoperation
26  void sendUSART0(uint8_t *ptr, uint8_t len){
27    if ((*ptr==0)||(len==0)) return; // Puffer leer
28    if (UCSR0A & (1<<UDRE0)){ // wenn Puffer frei
29      UDR0 = *ptr;              // 1. Zeichen versenden
30      USART0_ptr   = ptr+1; // Zeiger auf 2. Zeichen
31      USART0_len   = len-1; // Länge eins kürzer
32    }
33    else {
34      USART0_ptr   = ptr;   // Zeiger 1. Zeichen
35      USART0_len   = len;   // gesamte Pufferlänge
36    }
37    UCSR0B = 1<<TXEN0;      // Sender einschalten
38  }
```

- Wenn der Sendepuffer leer ist, wird das erste Byte sofort verschickt. Die übrigen Bytes verschickt die Schrittfunktion.

```
41  // Schrittfunktion Senden
42  void stepSendUSART0(){
43    // Wenn Senden aktiv und Sendepuffer frei
44    if ((UCSR0B & 1<<TXEN0) && (UCSR0A & (1<<UDRE0))){
45      uint8_t c =  *USART0_ptr; // nächsten Sendezeichen
46      USART0_ptr++;   // Zeiger erhöhen
47      USART0_len--;   // Länge bis Pufferende verringern
48    // Wenn das Zeichen oder Länge bis Pufferende null
49      if ((c==0) || (USART0_len==0))
50        UCSR0B = 0; // Sender (und Empfänge) ausschalten
51      else UDR0 = c;          // sonst Zeichen senden
52    }
53  }
```

- Wenn der Sender eingeschaltet und der Puffer leer ist, wird das nächste Byte versendet.
- Wenn das nächste Byte null oder das Pufferende erreicht ist, wird der Sender ausgeschaltet.

```
56  // Start einer Empfangsoperation
57  void getUSART0(uint8_t *ptr, uint8_t len){
58    USART0_ptr   = ptr;
59    USART0_len   = len;
60    UCSR0B = 1<<RXEN0;  // Empfänger einschalten
61  }
```

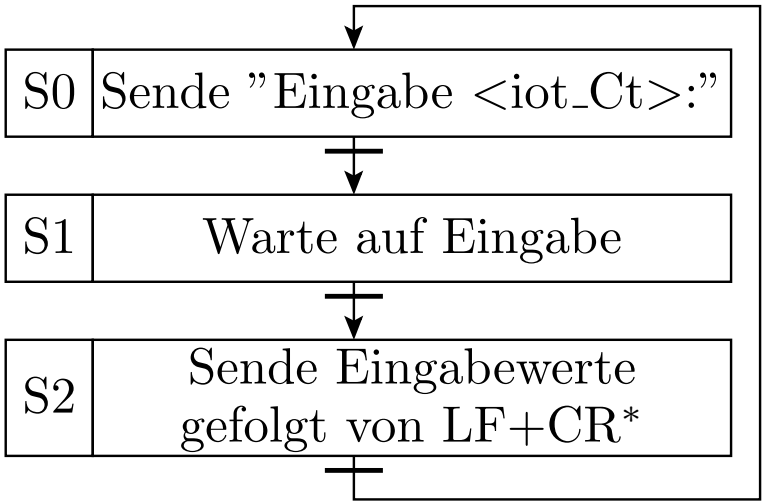
- Beim Start einer Empfangsoperation wird nur der Zeiger auf den Puffer und die Puffergröße lokal gespeichert und der Empfänger eingeschaltet.

```
// Schrittfunktion Empfang
void stepGetUSART0(){
    // Wenn Empfang aktiv und neues Byte empfangen
    if ((UCSR0B & 1<<RXEN0) && (UCSR0A & (1<<RXC0))){
        uint8_t c = UDR0;
        *USART0_ptr = c;
        USART0_ptr++;                   // Zeiger erhöhen
        USART0_len--;                   // Länge bis Pufferende
                                        // verringern
    // Wenn empfangenes Zeichen oder Länge bis Pufferende null
        if ((USART0_len==0) || (c==0))
            UCSR0B = 0; // Empfänger (und Sender) ausschalten
    }
}
```

```
80  // Test, ob alle IO-Operationen abgeschlossen sind
81  uint8_t IO_ready(){
82    // wenn Sender und Empfänger von USART0
83    // ausgeschaltet sind
84    return (UCSR0B & (1<<RXEN0 | 1 << TXEN0))==0;
85  }
```

- Der Abschluss aller IO-Operationen ist daran zu erkennen, dass Sender und Empfänger ausgeschaltet sind.

Schrittkette für die Ein- und Ausgabe




```
// Daten des IO-Tasks
uint16_t iot_Ct;
uint8_t iot_state;
#define SIZE 20
uint8_t iot_buffer[SIZE];
```

— Warte auf Zeitschritt und Abschluss aller IO-Operationen

- Der IO-Task umfasst drei Schritte, in denen hauptsächlich Zeichenkettenverarbeitung erfolgt.
- Die Zeichenkettenfunktionen strgcpy(), strgcat(), ... stammen aus strg.h /strg.c vom Foliensatz zuvor.
- Variablen sind der Zustand, ein Zähler und ein Puffer für Zeichenketten. »#define SIZE 20« legt die Puffergröße fest.

5. EA-Task für Bytefolgen

```
20 void IO_task(){
21   if (IO_ready()){
22     switch (iot_state){
23       case 0: //  "Nr.: <N>:" senden
24         strgcpy(iot_buffer, (uint8_t*)"Eingabe ", SIZE);
25         strgcat(iot_buffer, uint2str(iot_Ct), SIZE);
26         strgcat(iot_buffer, (uint8_t*)":", SIZE);
27         iot_Ct++;
28         sendUSART0(iot_buffer, SIZE);
29         iot_state=1;
30         break;
31       case 1: // Eingabe anfordern
32         getUSART0(iot_buffer, SIZE);
33         iot_state=2;
34         break;
35       case 2: // Eingabe aus buffer ausgeben
36         strgcat(iot_buffer, (uint8_t*)"\r\n", SIZE);
37         sendUSART0(iot_buffer, SIZE);
38         iot_state=0;
     }
   }
}
```

Das Hauptprogramm

```
 9   #include <avr/io.h>
10   #include "strg.h"
11   #include "TaskIO.h"

     ...
43   int main(void){
45     initTmr3(10);     // Timer und LED-Task
46     LED_Task_init(10);// Schrittzeit 1 ms
47     initUSART0();
49     while(1){
50       if(NeuerZeitschritt()){
51         IO_task();
52         stepSendUSART0();
53         stepGetUSART0();
54         LED_Task_step();
       }
     }
   }
```

- Initialisierung und Aufruf aller Schrittfunktionen

*Aufgabe 6.8: Erweiterung um eine LCD-Ausgabe[6]

Für die LCD-Ausgabe muss

- USART1 nach dem Vorbild von USART0 initialisiert
- eine SendUSART1-Funktion und eine StepSendUSART1-Funktion geschrieben werden.
- Die IO_Ready-Funktion darf nur wahr sein, wenn auch USART1 nicht sendet.
- Schrittfunktion in der Hauptschleife in main() aufrufen.
- Zum testen im IO-Task eine LCD-Ausgabe einfügen z.B.

```
SendUSART1("\0x1b[i LCD Ausgabetext",20);
```

[6] In den mit * gekennzeichneten Aufgaben werden Funktionsbausteine für das finale Projekt entwickelt. Die hier entwickelten Lösungen gut dokumentieren und gründlich testen.

*Aufgabe 6.9: Erweiterung um eine Sleep-Funktion

Der IO-Task wird später der Steueralgorithmus für das Fahrzeug. Für Aufgaben der Form

```
Fahre bis ..., warte 1s, ...
```

wird eine Funktion

```
sleep(uint16_t w); // w -- Wartezeit in ms
```

benötigt. Lösungsvorschlag:

- Die sleep-Funktion kopiert den Übergabewert geteilt durch die Schrittzeit in eine globale Variable:

  ```
  int16_t wait_Ct;
  ```

- Diese wird, wenn sie ungleich null ist, im Blink-Task heruntergezählt.
- Solange sie ungleich null ist, soll IO_Ready() den Wert null zurückgeben.

*Aufgabe 6.10: Ein- und Ausgabe von Zahlenwerten

Zur Steuerung und Überwachung des Fahrzeugs ist es oft günstiger, Zahlen statt Texte mit dem PC auszutauschen. Ein Struktur von Zahlenwerten besteht aus eine festen Anzahl von Bytes statt einer null-terminierten Bytefolge. Benötigt werden Funktionen vom Typ:

```
sendZahlen(uint16_t a, int8_t b); // das sind 3 Byte
getZahlen(int32_t x, uint16_t y); // das sind 6 Byte
```

Man könnte diese Bytes in den Puffer mit einer Abschlussnull übergeben. Dann würden bei einem Bytewert null die Bytes danach verloren gehen.
Wie könnten solche Funktionen fehlerfrei implementiert werden?

Lösungsvorschlag:

- Vereinbarung eines globalen privaten Byte-Puffers in der Datei TaskIO.c.
- Die Sendefunktion für Zahlen kopiert die Werte in Bytes aufgeteilt in diesen Puffer, setzt den Zeiger auf den Pufferanfang und die Länge gleich der zu übertragenden Byteanzahl.
- Die Get-Funktion für Zahlen setzt nur Zeiger und Länge. Nach dem Empfang werden auch noch Funktionen zum Lesen der empfangenen Daten aus dem Puffer benötigt.
- Die Schrittfunktionen unterbinden den Abbruch bei Bytewert null, wenn der Zeiger auf eine Adresse im privaten Puffer zeigt.
- Entwickeln Sie je eine Beispielfunktion für das Senden und Anfordern von Zahlenwerten.
- Passen Sie den Rest des Programms an.
- Entwickeln Sie einen Testrahmen für diese Funktionen.

Aufgabe 6.11: Experimente mit der Schrittzeit

Im Beispielprogramm ist die Schrittzeit auf 1 ms eingestellt, damit keine Empfangsbyts verloren gehen.

- Übersetzen Sie das Programm mit 10 ms Schrittzeit. Treten Fehlfunktionen auf und was für welche?
- Stellen Sie die Baudrate für diese Schrittzeit so niedrig ein, dass die Fehlfunktionen verschwinden.
- Fügen Sie bei einer eingestellten Schrittzeit von 1 ms in eine der Schrittfunktion eine Warteschleife von etwa 2 ms Zeit ein. Treten jetzt Fehlfunktionen auf und was für welche?