Informatikwerkstatt, Foliensatz 6
Zeitabläufe und Schrittketten

G. Kemnitz

Institut für Informatik, Technische Universität Clausthal

15. Dezember 2014

Inhalt des Foliensatzes

Mehr-Task-Programme

Timer einrichten

Nebenläufig blinkende LEDs

Task mit Ein- und Ausgabe

Seperater Ein-/Ausgabe-Tasks

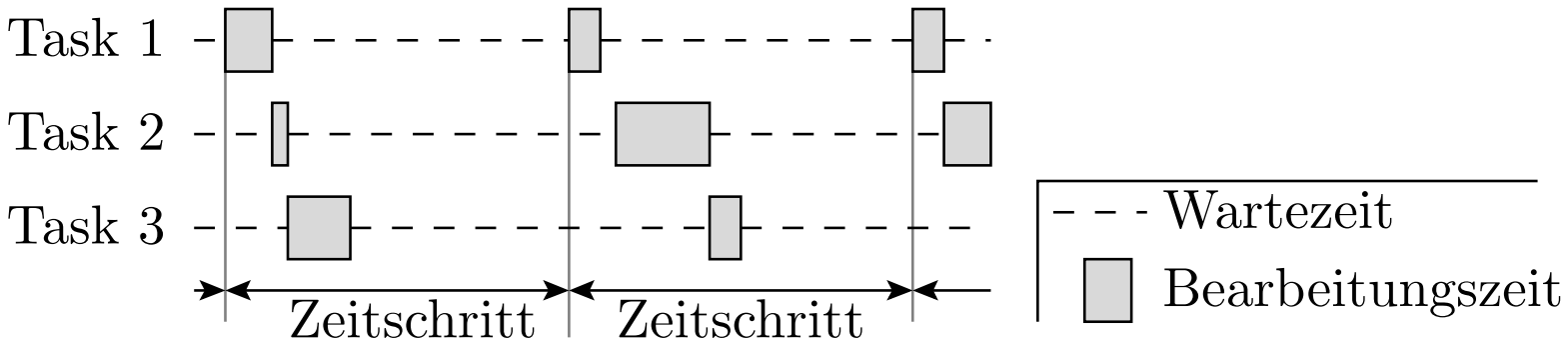
Mehr-Task-Programme

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, ...

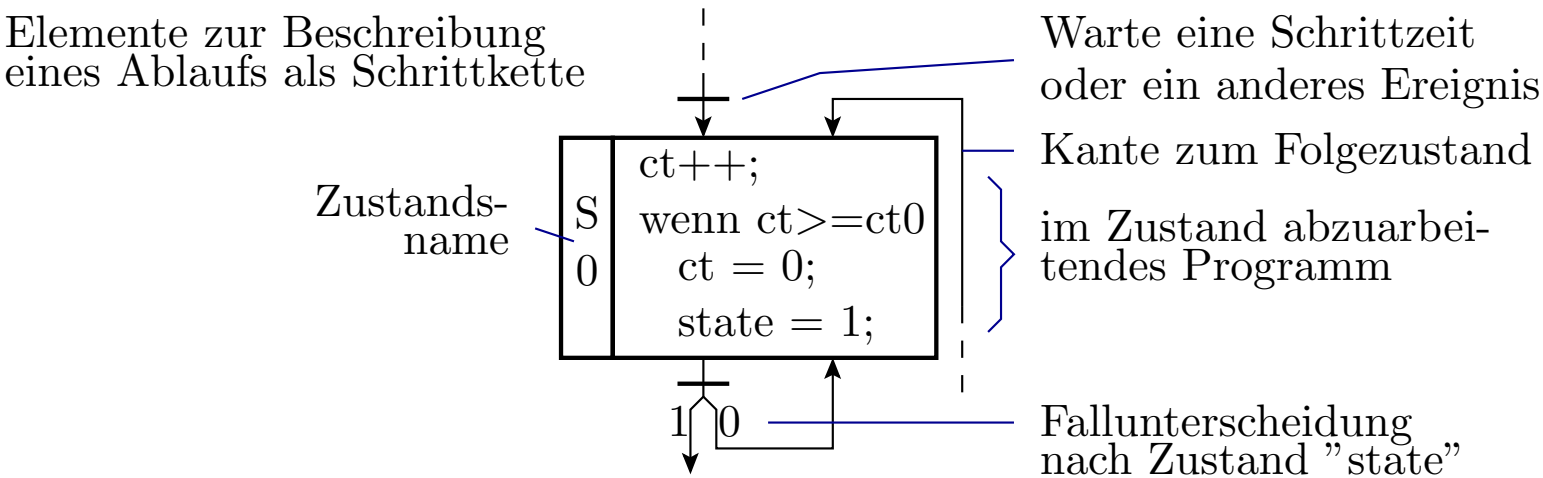
Bei jeder dieser Aufgaben wird die meiste Zeit gewartet. Die nachfolgende Abbildung zeigt ein einfaches Taskscheduling[1]:



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

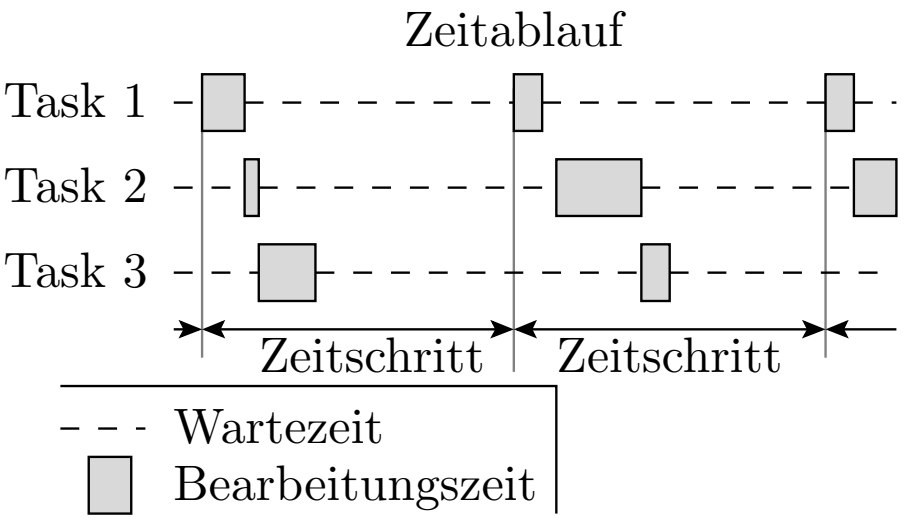
Programmtechnische Umsetzung

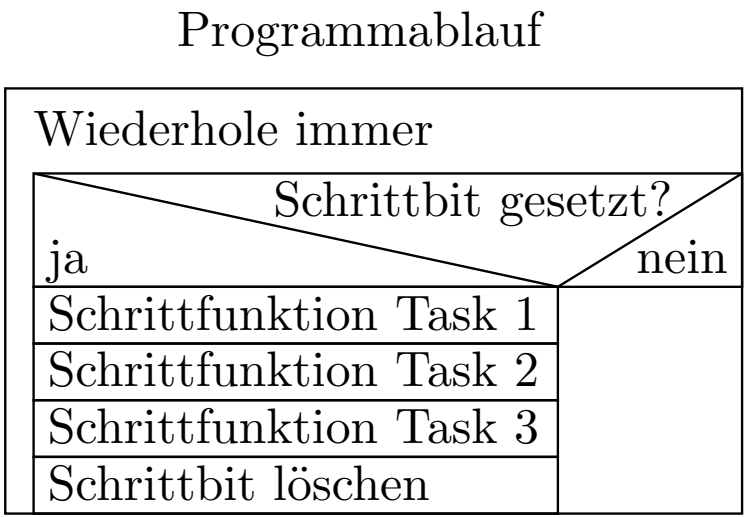
- Programmieren der Tasks als Schrittketten (Funktionen mit einer Zustandsvariablen, die bei Aufruf einen Schritt abarbeiten, den Zustand weiterschalten und den Kontrollfluss zurück geben),



- Timer, der immer nach der Schrittzeit ein Bit setzt (oder eine EA-Schnittstelle, die ein Ereignisbit setzt).

- Endlosschleife, die auf das Schritt- oder Ereignisbit wartet.
- Wenn gesetzt, sequentielle Abarbeitung des nächsten Schritts aller Tasks und Schrittbit löschen.

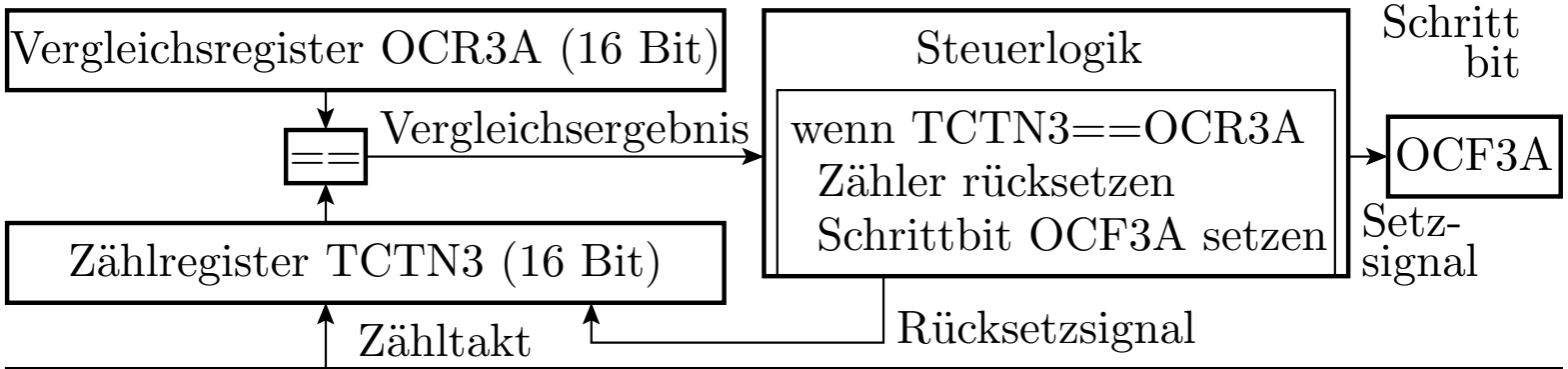




Timer einrichten

Funktionsweise eines Timers im CTC-Mode

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	—	—

Im CTC-Mode wird der Timer bei Erreichen des Vergleichswertes rückgesetzt. Schrittzeit: $t_s = \frac{\mathrm{OCR3} \cdot vt}{f_{\mathrm{CPU}}}$

Timer-Initialisierung

- Übergabe der Schrittzeit
$$t_\mathrm{s} = \frac{\mathrm{OCR3} \cdot vt}{f_\mathrm{CPU}}$$
($f_\mathrm{CPU} \approx 7{,}58\,\mathrm{MHz}$ – Prozessortakt[2]; vt – Vorteilerwert; OCR3 – Vergleichswert) in 100µs-Schritten.
- Maximal einstellbare Schrittzeit:
$$t_\mathrm{s.max} = \frac{\mathrm{OCR3_{max}} \cdot \mathrm{vt_{max}}}{f_\mathrm{CPU}} = \frac{\mathrm{0xFFFF} \cdot 1024}{7{,}58\,\mathrm{MHz}} = 8{,}85\,\mathrm{s}\ (88500)$$
- Wahl des Vorteilers so, dass OCR3 $\leq$0xFFFF und mit dem nächst kleinere Vorteilerwerts $>$0xFFFF ist.
- In TCCR3B CTC-Modus und vt eingestellt.
- Das Schrittbit OCF3 ist nach der Initialisierung und auch nach Abschluss aller Task-Schritte durch Schreiben einer Eins zu löschen.

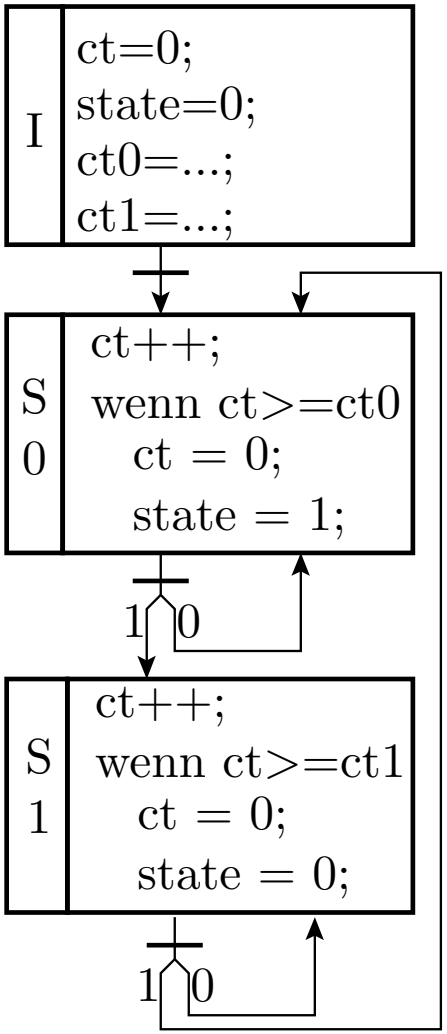
[2] Experimentell bestimmt.

2. Timer einrichten

```
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
```

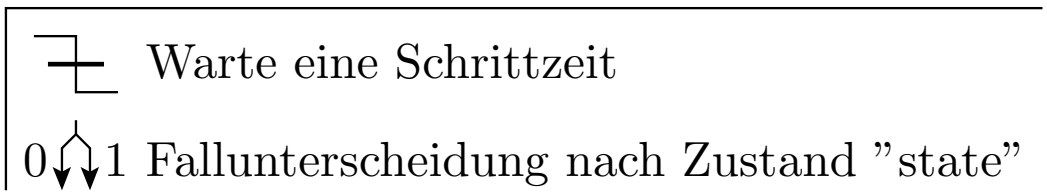
Nebenläufig blinkende LEDs

Schrittkettenbeschreibung einer blinkenden LED

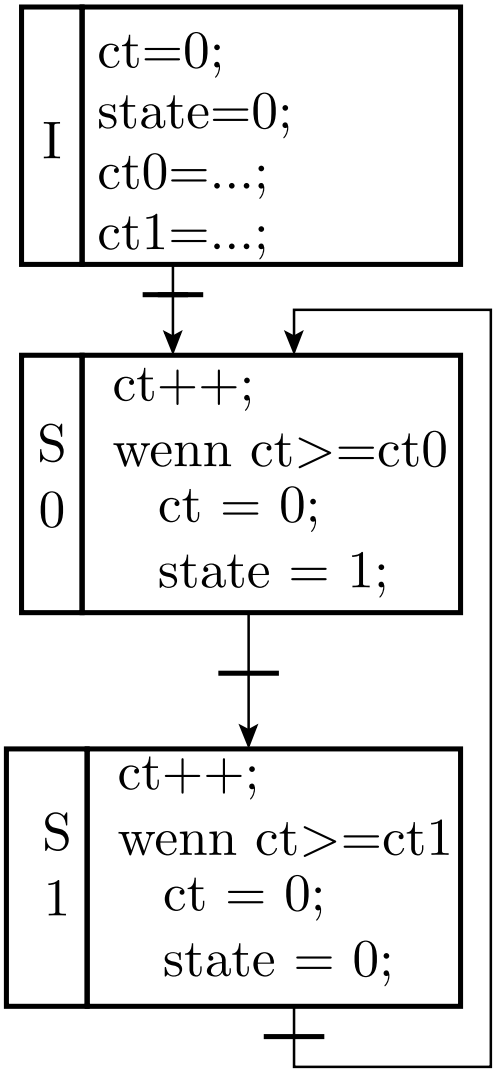


Vereinbarungen in der Header-Datei LED_Task.h

```
// 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);
```



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

- Die Variablen der einzelnen Tasks müssen global vereinbart werden[3]. Warum?
- Zu Beginn sind der Timer, jeder Task und die Led-Anschlüsse als Ausgänge zu intialisieren.
- In der Endlosschleife wird
  - die meiste Zeit (fast die gesamte Schrittzeit) auf die Aktivierung des Schrittbits gewartet.
  - die Schrittfunktion jedes Tasks ausgeführt,
  - aus den Rückgabewerten (Zuständen) der Schrittfunktionen die Led-Ausgabe gebildet und
  - ausgegeben.

[3] In Modulen (seperaten C-Dateien) definierte globale Variablen sind nur für Funktionen des Moduls zugänglich, aber nicht für die der anderen Module.

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[4]:

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

[4] 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[5]:
  - Watch 1 soll alle Zähler und Zustände anzeigten.
  - Local soll den Wert von tp anzeigen, um den Task, der bearbeitet wird, zu erkennen.

[5] 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.

Task mit Ein- und Ausgabe

Das Echo-Programm von Foliensatz 2

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

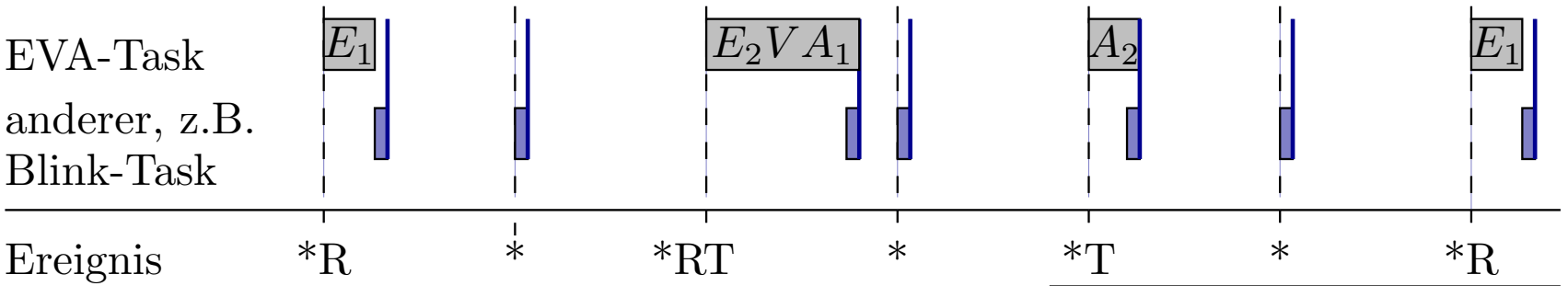
```
UBRR0L = 49;           UBRR0H = 0;
UCSR0B = 0b00011000; UCSR0C=0b0011100;
```

- 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
}
```

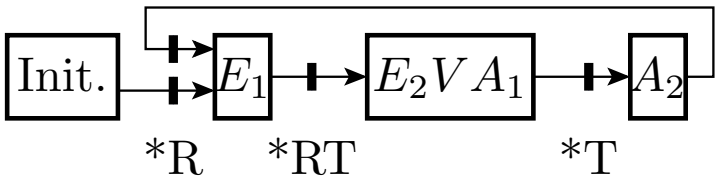
In Mehr-Task-Programmen ist die Folge »Eingabe – Verarbeitung – Ausgabe« mit Schrittketten statt »Warten« nachzubilden.

EVA-Task als Schrittkette

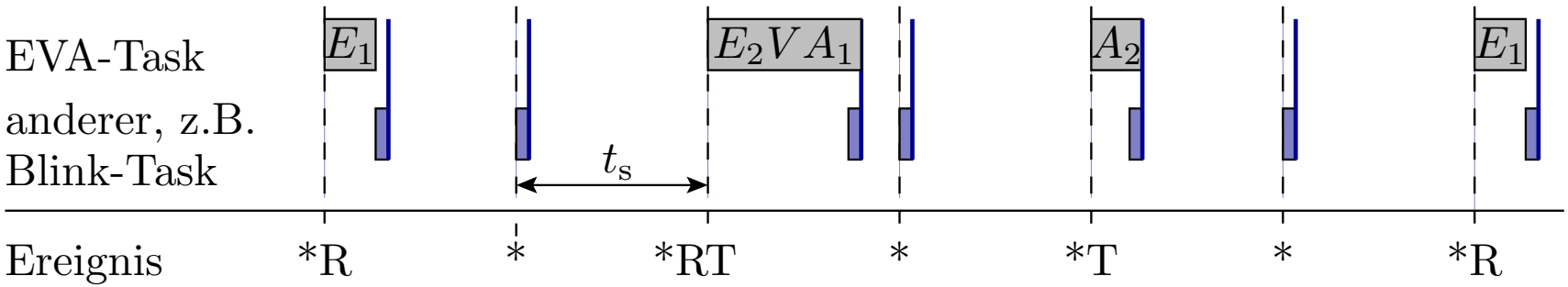


E_i	Eingabe Byte i
V	Verarbeitung
A_i	Ausgabe Byte i
*	Schrittbit gesetzt
R	Empfangsbit gesetzt
T	Sendepuffer frei
|	Schrittbit löschen

EVA-Task als Schrittkette:



- Der Task benötigt für jedes Eingabebyte einen Schritt.
- Die letzte Eingabe, die Verarbeitung und die ersten Ausgabe können gegebenenfalls in einem Zeitschritt erfolgen.
- Die zul. Eingabegeschwindigkeit hängt von der Schrittzeit ab.



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

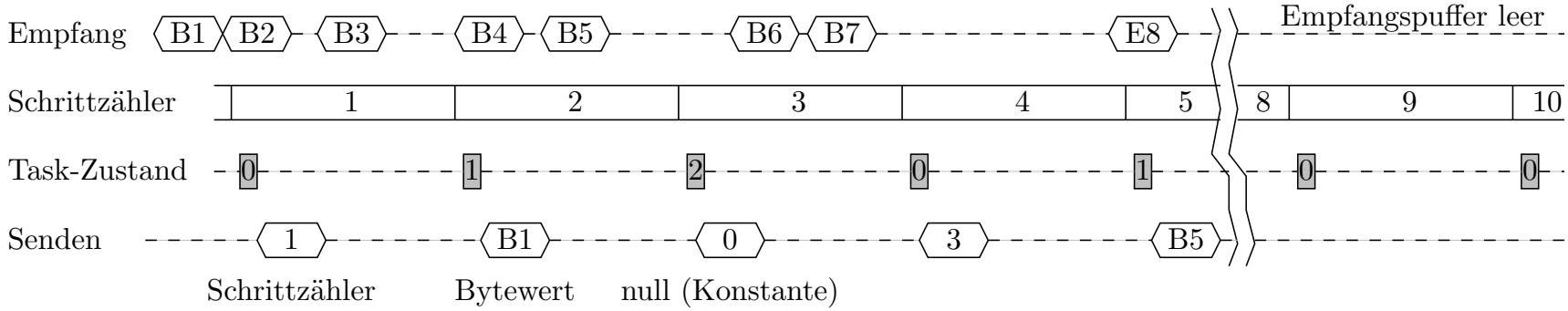
$$t_{\text{byte}} = \frac{10 \ldots 12\,\text{Bit}}{9600\,\frac{\text{Bit}}{\text{s}}} = 1{,}04 \ldots 1{,}25\,\text{ms}$$

- Um mehrere Bytes hintereinander lückenlos zu empfangen, Schrittzeit $t_s < t_{\text{byte}}$.
- Damit auch alle anderen Tasks korrekt arbeiten, Schrittzeit größer als max. Summe der Verarbeitungszeiten aller Tasks.

- Wie viele Schrittzeiten dauert max. ein EVA-Zyklus?
- Alle wie viel Schrittzeiten dürfen Service-Anforderungen gestellt werden?

Experiment

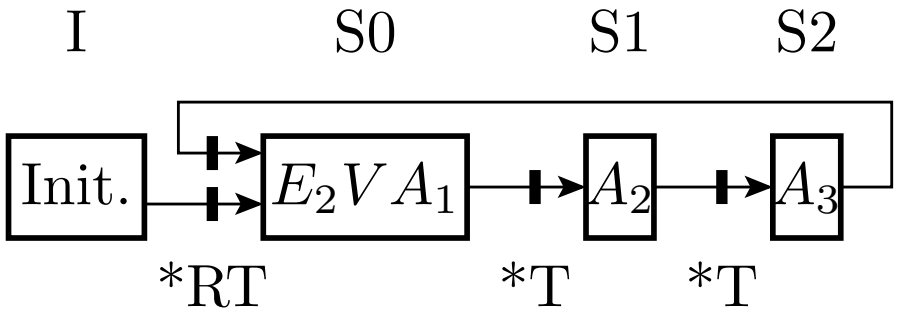
Der EVA-Task soll auf ein Byte warten und für jedes empfangene Byte drei Bytes senden, einen Schrittzähler, den Bytewert und eine Abschlussnull. Parallel soll ein Blink-Task laufen. Zu untersuchen ist, auf welche Eingaben der EVA-Task reagiert und welche er ignoriert.



Mit der Beispieleingabe reagiert er auf »B1« und »B5« und die anderen nicht. Welches Sende-Timing muss HTerm einhalten und wie kann man das sicherstellen?

Schrittkettenbeschreibung des EVA-Tasks:

Zustände:



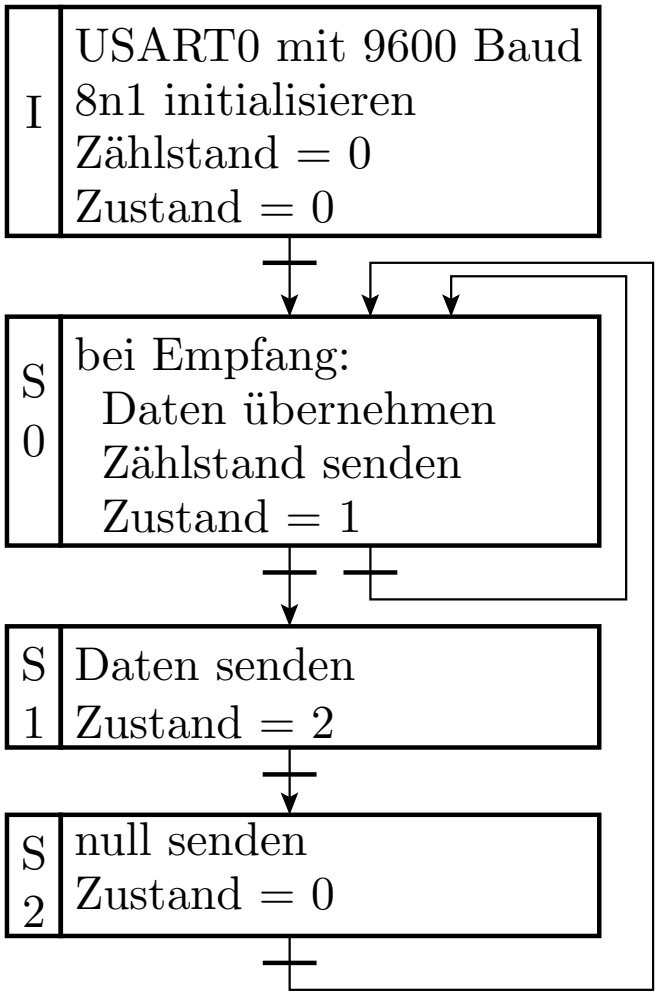
E_i	Eingabe Byte i
V	Verarbeitung
A_i	Ausgabe Byte i
*	Schrittbit gesetzt
R	Empfangsbit gesetzt
T	Sendepuffer frei
|	Schrittbit löschen

Experiment vorbereiten:

- 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 und Abschlussnull).

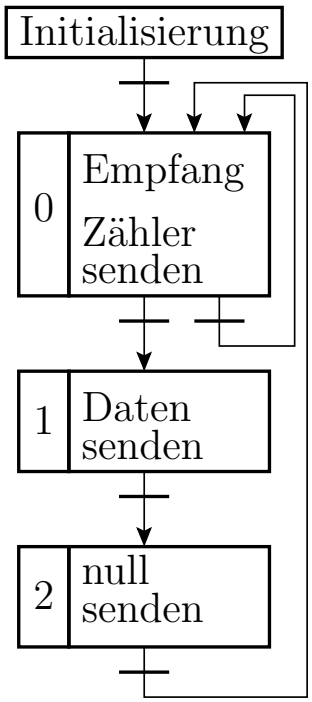
Besprechen des Programms

- 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;
}
```

- Der zusätzliche Blink-Task für die 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;
    }
```

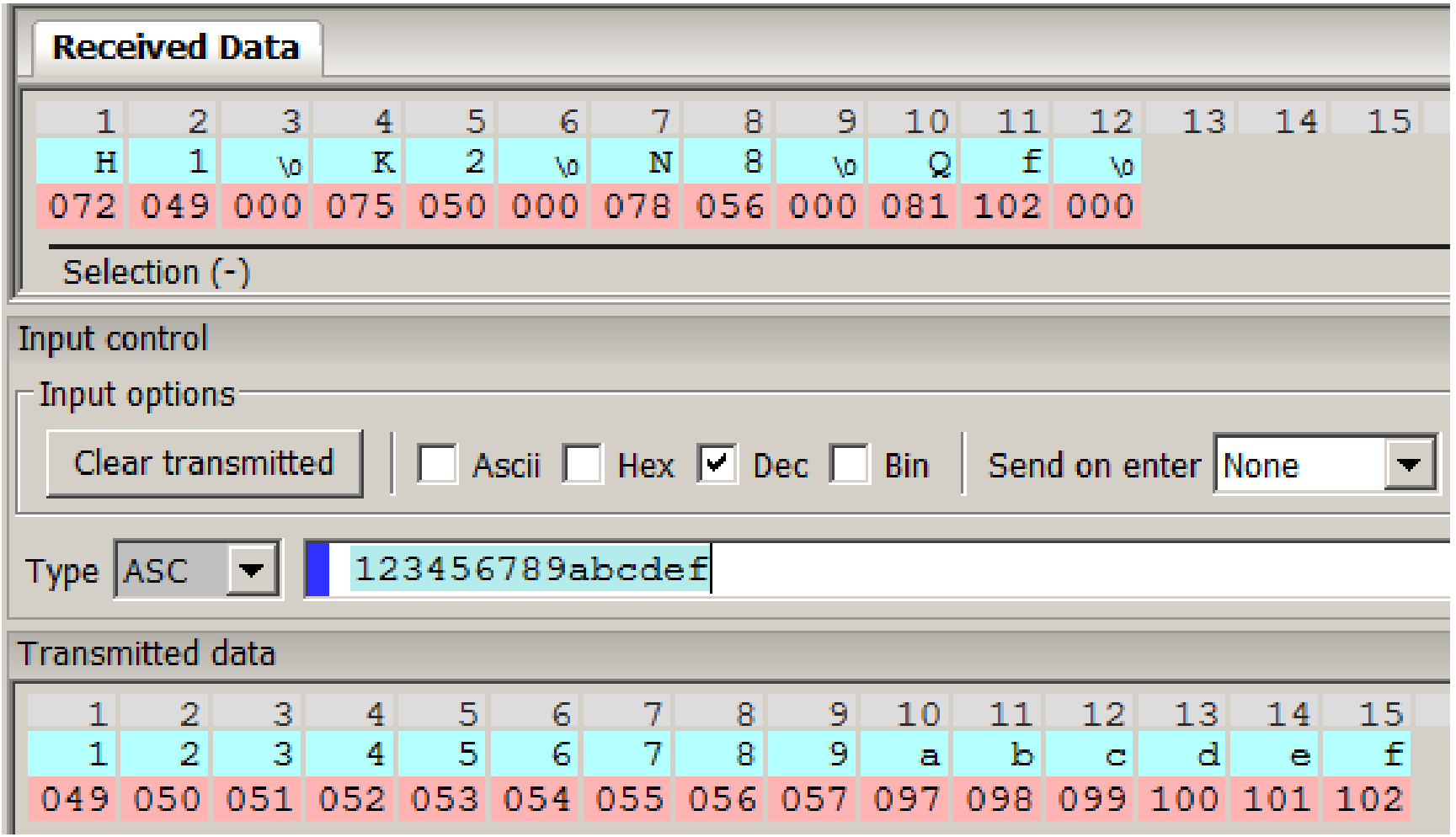
- Die bis hierher beschrieben Funktionen stehen in »TaskFkt.c«. Das Hautprogramm übernimmt die Definitionen aus dem Header »TaskFkt.h«:

```
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();
```

- 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 6.5: Experimentiervorschläge

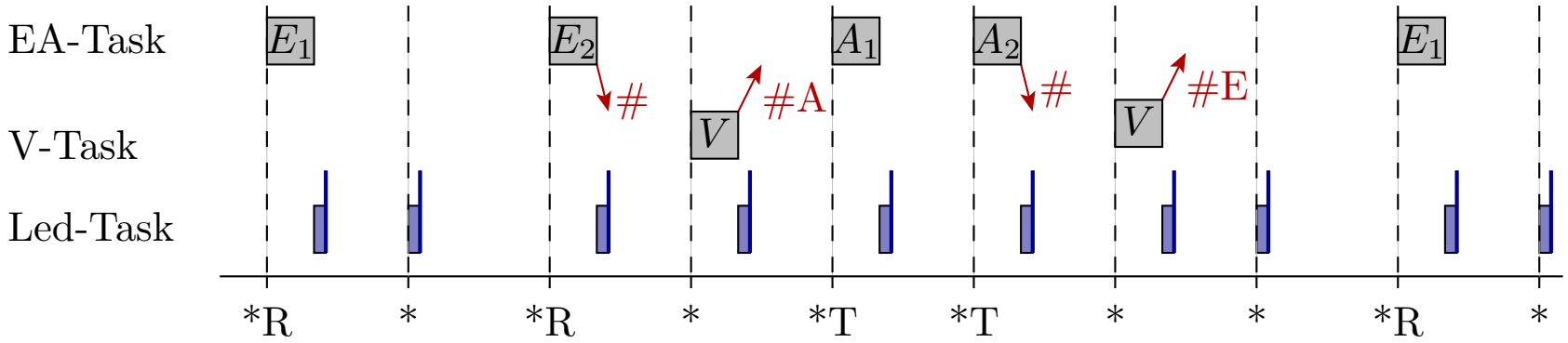
- Verwenden Sie die Terminaleinstellungen und Eingaben der Folie zuvor. Erschließen Sie sich die Funktionsweise durch stückweise Abarbeitung mit Unterbrechungspunkten.
- Untersuchen Sie, wie viele gesendete Bytes im Mittel ignoriert werden, wenn
  1. 30 Bytes als Block gesendet werden
  2. 4 Bytes mit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

  vom PC gesendet werden.
- In welchem Zeitabstand werden die Bytes empfangen[6]?
- Wiederholen Sie die Experimente mit einer geänderten Schrittzeit von 1 ms.

[6] 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.

Seperater Ein-/Ausgabe-Tasks

Ablauf mit seperatem EA-Task



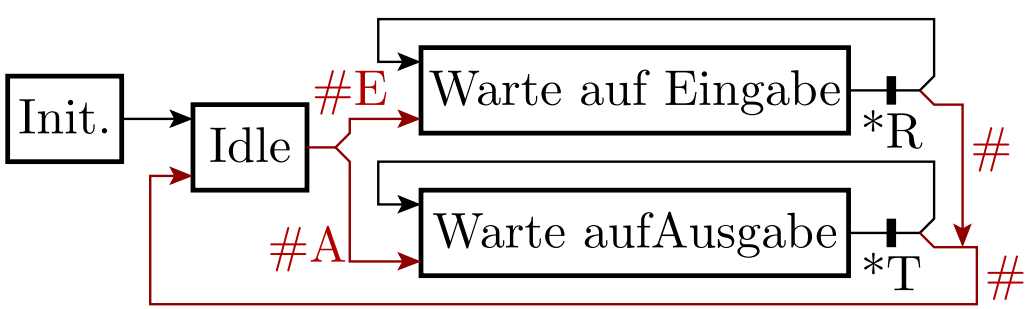
#E Eingabeanforderung
#A Ausgabeanforderung
\# EA abgeschlosen (Ready)

E_i Eingabe Byte i
V Verarbeitung
A_i Ausgabe Byte i

* Schrittbit gesetzt
R Empfangsbit gesetzt
T Sendepuffer frei
| Schrittbit löschen

- Trennung des EVA-Ablaufs in einen EA- und einen V-Task.
- Während der Ein- und Ausgabe ruht die Verarbeitung.
- In einem Verarbeitungsschritt werden aus mehreren Eingabebytes mehrere Ergebnisbytes berechnet und eine Ein- oder Ausgabeoperation initialisiert.
- Led-Blink-Task zur Kontrolle, dass die Schrittkette arbeitet.

EA-Task



*	Schrittbit gesetzt
R	Empfangsbit gesetzt
T	Sendepuffer frei
#E	Eingabeanforderung
#A	Ausgabeanforderung
#	EA abgeschlosen (Ready)

- Der EA-Task hat die Zustände Idle (keine Aufgabe), auf Eingabe- oder auf Ausgabe warten. Unterscheidbar an den SFR-Einschaltbits »TXEN« und »RXEN« der UART.
- Die Ein- und Ausgabeanforderungen sollen durch Initialisierungsfunktionen für Sende- und Empfangsanforderungen realisiert werden.
- Die Bestätigung, dass keine EA-Operation läuft, kann auch mit »EA-Taskzustand==Idle« erfolgen.
- Übergabe der EA-Zeichenfolge mit einem Puffer.

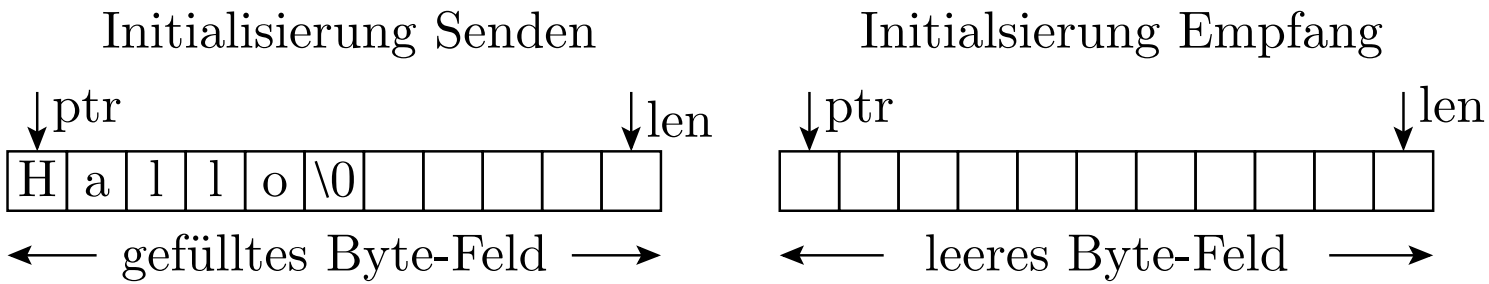
Initialisierung der UART und des EA-Tasks

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

- Mit »Senden und Empfang aus« (TXEN0=RXEN0=0) wird der Empfangsautomat mit Zustand »Idle« initialisiert.
- Die Sendeschrittfunktion soll alle Zeichen bis zur ersten »\0«, aber max. »len« Zeichen versenden und dann den Sender deaktivieren.
- Die Empfangsschrittfunktion soll alle Zeichen bis zur ersten »\0«, aber max. »len« Zeichen empfangen und dann den Empfänger deaktivieren.

Übergabe der Sende- und Empfangsbytes

Die Anforderungsfunktionen für die Ein- und Ausgabe bekommt einen Zeiger »ptr« auf den Pufferanfang und eine Obergrenze »len« für die Zeichenanzahl übergeben:



Zeiger und Restlänge werden in privaten globalen Variablen des IO-Tasks gespeichert:

```
uint8_t *USART0_ptr; // Zeiger auf den nächten Pufferplatz
uint8_t USART0_len;  // Länge bis Pufferende
```

Das Byte-Feld wird eine Variable des aufrufenden Programms sein.

Start einer Sendeoperation:

```
void sendUSART0(uint8_t *ptr, uint8_t len){
  if ((*ptr==0)||(len==0)) return; // Puffer leer
  if (UCSR0A & (1<<UDRE0)){ // wenn Puffer frei
    UDR0 = *ptr;            // 1. Zeichen versenden
    USART0_ptr   = ptr+1; // Zeiger auf 2. Zeichen
    USART0_len   = len-1; // Länge eins kürzer
  }
  else {
    USART0_ptr   = ptr;   // Zeiger 1. Zeichen
    USART0_len   = len;   // gesamte Pufferlänge
  }
  UCSR0B = 1<<TXEN0;      // Sender einschalten
}
```

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

Schrittfunktion Senden:

```
void stepSendUSART0(){
  // Wenn Senden aktiv und Sendepuffer frei
  if ((UCSR0B & 1<<TXEN0) && (UCSR0A & (1<<UDRE0))){
    uint8_t c =  *USART0_ptr; // nächsten Sendezeichen
    USART0_ptr++;   // Zeiger erhöhen
    USART0_len--;   // Länge bis Pufferende verringern
  // Wenn das Zeichen oder Länge bis Pufferende null
    if ((c==0) || (USART0_len==0))
      UCSR0B = 0; // Sender (und Empfänge) ausschalten
    else UDR0 = c;          // sonst Zeichen senden
  }
}
```

- 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.

Weitere Hilfsfunktionen

Start einer Empfangsoperation:

```
void getUSART0(uint8_t *ptr, uint8_t len){
  USART0_ptr    = ptr;
  USART0_len    = len;
  UCSR0B = 1<<RXEN0;  // Empfänger einschalten
}
```

- Beim Start einer Empfangsoperation wird nur der Zeiger auf den Puffer und die Puffergröße in den privaten globalen Variablen des EA-Tasks 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) aus
    }
}
```

- Wenn der Empfänger eingeschaltet ist und ein Byte empfange wurde, Byte in den Puffer speichern.
- Bei Empfang von »\0« Empfänger (und Sender) ausschalten.

Test, ob alle IO-Operationen abgeschlossen sind:

```
uint8_t IO_ready(){
  // wenn Sender und Empfänger von USART0
  // ausgeschaltet sind
  return (UCSR0B & (1<<RXEN0 | 1 << TXEN0))==0;
```

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

Die Initialisierungsfunktion für Timer 3 (InitTmr3()):

- Die privaten Daten und die Initialisierung des LED-Tasks sind identisch mit denen aus dem Projekt zuvor.

Schrittfunktion des Led-Tasks

```
void LED_Task_step(){
  ETIFR = 1<< OCF3A;          // Schrittbit löschen
  LED_Ct++;
  if (LED_Ct>=maxCt){         // alle 0,5 Sekunden
    PORTE ^= 0x10;            // LD4 invertieren
    LED_Ct = 0;               // Zähler rücksetzen
  }
}
```

- erzeugt ein 1Hz-Blinksignal,
- setzt zusätzlich das Schrittbit zurück.

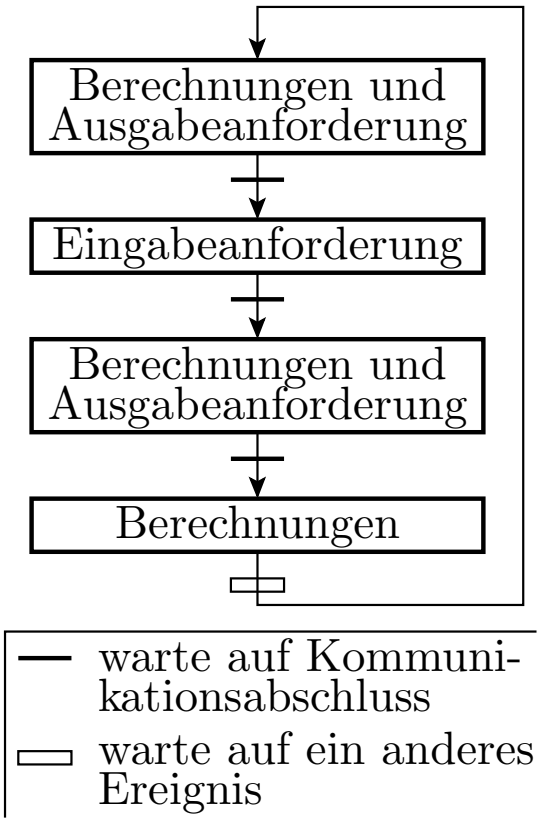
Kapselung der Schrittbitabfrage in eine Funktion:

```
uint8_t NeuerZeitschritt(){return ETIFR & (1<< OCF3A);}
```

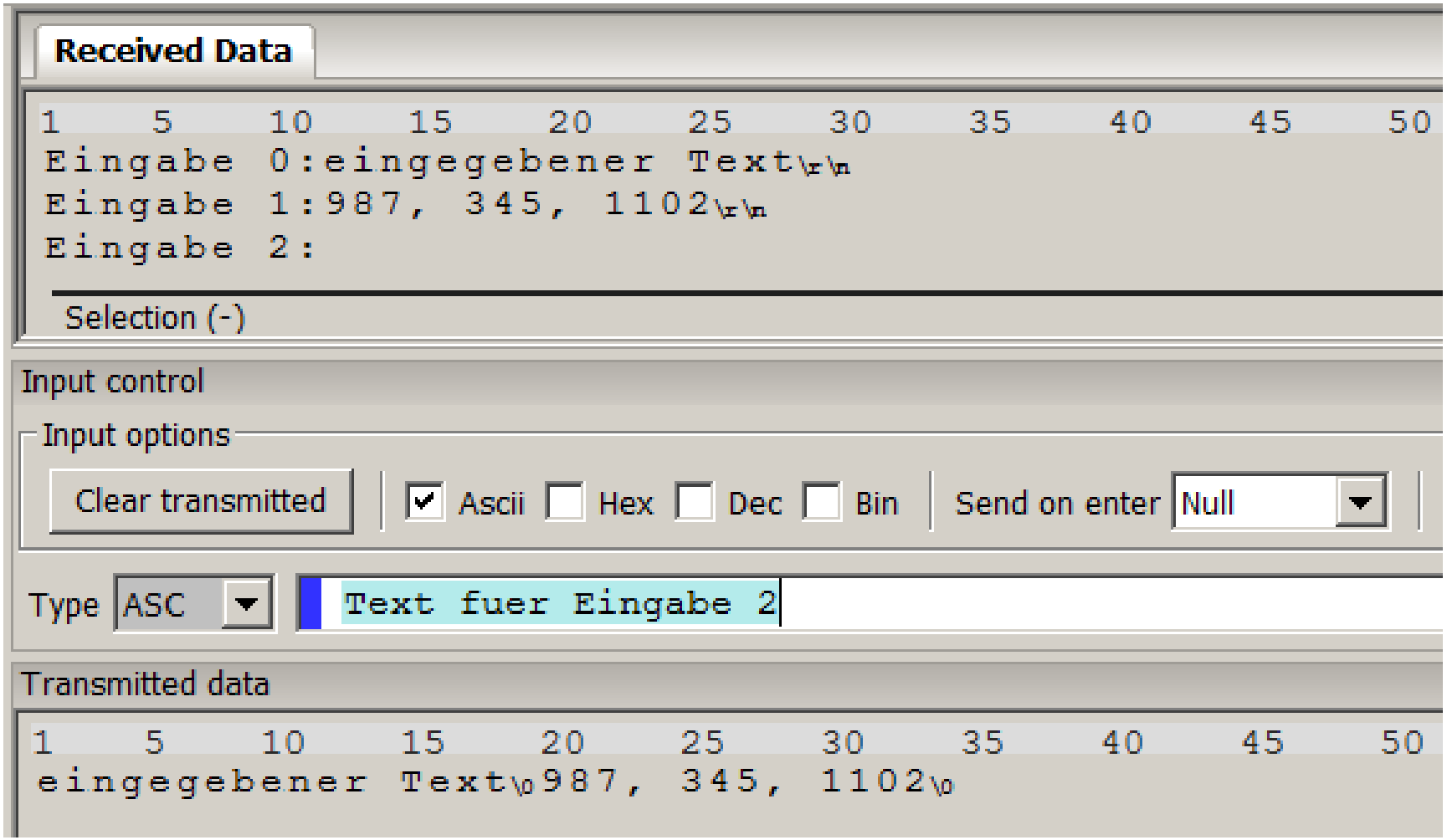
Der übergeordnete Zeitablauf

Der übliche Zeitablauf eines EVA-Programmes, z.B. der übergeordneten Fahrzeugsteuerung, besteht darin, Eingaben zu empfangen, diese zu verarbeiten und Ausgaben zurückzusenden.

Nach Initialisierung einer Ein- oder 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



Die bereits definierte Funktionen (TaskIO.h)

- Funktionen zur Kommunikation über die UART:

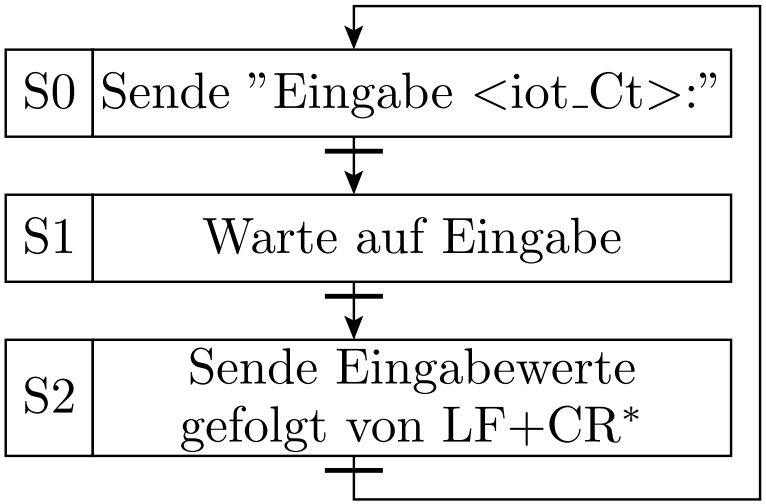
```
// Initialisiere USART0 an JE mit 9600 Baud 8n1
void initUSART0();

// Start einer Sendeoperation
void sendUSART0(uint8_t *ptr, uint8_t len);

// Schrittfunktion Senden
void stepSendUSART0();

// Start einer Empfangsoperation
void getUSART0(uint8_t *ptr, uint8_t len);

// Schrittfunktion Empfang
void stepGetUSART0();
```

- Funktionen zur Ablaufkontrolle

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

// Funktion zu Signalisierung eines neuen Zeitschritts
uint8_t NeuerZeitschritt();

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

// Schrittfkt. zur Erzeugung eines 1 Hz Blinksignals
void LED_Task_step();

// Test, ob alle IO-Operationen abgeschlossen sind
uint8_t IO_ready();
```

- Zeichenkettenverarbeitungsfunktionen »strgcat« und »strgcpy« aus strg.h /strg.c von Foliensatz 4.

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.
- Private globale Daten des Tasks: Zustand »iot_state«, Zähler »iot_Ct«, 20-Byte-Puffer »iot_buffer« für Zeichenketten.

```
void IO_task(){
  if (IO_ready()){
    switch (iot_state){
      case 0: //  "Nr.: <N>:" senden
        strgcpy(iot_buffer, (uint8_t*)"Eingabe ", SIZE);
        strgcat(iot_buffer, uint2str(iot_Ct), SIZE);
        strgcat(iot_buffer, (uint8_t*)":", SIZE);
        iot_Ct++;
        sendUSART0(iot_buffer, SIZE);
        iot_state=1;
        break;
      case 1: // Eingabe anfordern
        getUSART0(iot_buffer, SIZE);
        iot_state=2;
        break;
      case 2: // Eingabe aus buffer ausgeben
        strgcat(iot_buffer, (uint8_t*)"\r\n", SIZE);
        sendUSART0(iot_buffer, SIZE);
        iot_state=0;
    }
  }
}
```

Das Hauptprogramm

```
#include <avr/io.h>
#include "strg.h"
#include "TaskIO.h"
    ...
int main(void){
  initTmr3(10);      // Timer und LED-Task
  LED_Task_init(10);// Schrittzeit 1 ms
  initUSART0();
  while(1){
    if(NeuerZeitschritt()){
      IO_task();
      stepSendUSART0();
      stepGetUSART0();
      LED_Task_step();
    }
  }
}
```

*Aufgabe 6.6: Erweiterung um eine LCD-Ausgabe[7]

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);
```

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

*Aufgabe 6.7: 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 6.8: 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 6.9: 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?