Informatikwerkstatt, Foliensatz 7
Interrupts

G. Kemnitz

Institut für Informatik, TU Clausthal (IW7)

1. Dezember 2016

Inhalt:

- Wiederholung
- Task-Scheduling mit Interrupts
- Experimente
- Aufgaben

Interaktive Übungen:

1. Experimente mit Interrupts (test_interrupt)

Wiederholung

Wiederholungsaufgabe 7.1: Timer

- Was sind die Bestandteile und die grundlegenden Funktionen eines Timers?
- Wie wird mit einem Timer eine Wartefunktion programmiert?
- Was unterscheidet den CTC-Modus vom normalen Modus?

Lösung

1. Ein Timer besteht aus einem Zählregister mit programmierbarem Zähltakt, Vergleichsregistern, ..., zählt Zeitschritte oder externe Impulse und setzt bei Überlauf, Gleichheit von Zähl- und Vergleichswerten, ... Ereignisbits, die vom Programm ausgewertet werden können.
2. Wartefunktion:
   - Zählregister mit einem Startwert beschreiben,
   - Überlaufsbit löschen,
   - Zähltakt einschalten,
   - bist zum Überlauf warten,
   - Zähltakt ausschalten und Rücksprung.
3. Im Normalmodus zählt der Zähler zyklisch von null bis zum Maximalwert $2^{N_{\mathrm{Bit}}}$ (N_{Bit} – Bitanzahl des Zählregisters). Im CTC- (**C**lear **T**imer on **C**ompare) Modus wird der Zähler bei Erreichen des Vergleichswerts zurückgesetzt.

Wiederholungsaufgabe 7.2: Globale und lokale Variablen

1. Was sind die Unterschiede zwischen globalen und lokalen Variablen?
2. Wie werden globale und lokale Variablen in C vereinbart?
3. Welche Adressen vergibt unser Compiler für globale und lokale Variablen?
4. Sind Daten, die von unterschiedlichen Treiberfunktionen bearbeitet werden, global oder lokal zu vereinbaren? Begründung?

Lösung

1. Globale Variablen existieren während der gesamten Programmlaufzeit und haben feste Adressen. Lokale Variablen existieren nur innerhalb des mit {...} eingeramten Programmblocks, in dem sie definiert sind, und erhalten Adressen auf dem Stack.
2. Globale Variablen werden außerhalb der Funktion und lokale innerhalb des Blocks, in dem sie existieren, vereinbart.
3. Unser Compiler vergibt für globale Variablen die Datenspeicheradressen ab 0x200 aufwärts und für lokale Variablen Stackadressen ab 0x21FF abwärts.
4. Daten, die von unterschiedlichen Treiberfunktionen bearbeitet werden, sind global zu vereinbaren, damit die Werte nach Verlassen einer Bearbeitungsfunktionen erhalten bleiben.

Wiederholungsaufgabe 7.3: Treiber

1. Wie kann man im SFR-Register TCCR1B die Bits 0 bis 2 auf den Wert 0b101 setzen, ohne die übrigen Bits zu verändern?
2. Welchen offensichtlichen Nachteile hat das bisherige Schrittfunktionskonzept unter dem Blickwinkel, dass EA-Ereignisse in sehr unterschiedlichen zeitlichen Abständen eintreten?
3. Welche zusätzliche Funktionalität wäre erforderlich, damit die Schrittfunktionen für EA-Ereignisse genau einmal je Ereignis aufgerufen werden?

Lösung

1. SFR-Register TCCR1B[2:0]=0b101:

```
TCCR1B = (TCCR1B & 0b11111000) | 0b101;
```

2. Der zeitliche Abstand zwischen dem Aufrufen einer Schrittfunktion muss, damit alle EA-Ereignisse bearbeitet werden, kürzer als der minimale Zeitabstand zwischen EA-Ereignissen sein. Bei größeren Ereignisabständen hat man viele überflüssige Aufrufe.
3. Damit die Schrittfunktionen für jedes EA-Ereignis genau einmal aufgerufen werden, müsste die Hardware eine Möglichkeit bereitstellen, dass beim Setzen eines Ereignisbits automatisch ein Unterprogramm von einer dem Ereignisbit zugeordneten Adresse aufgerufen wird.

Task-Scheduling mit Interrupts

Polling und Interrupt

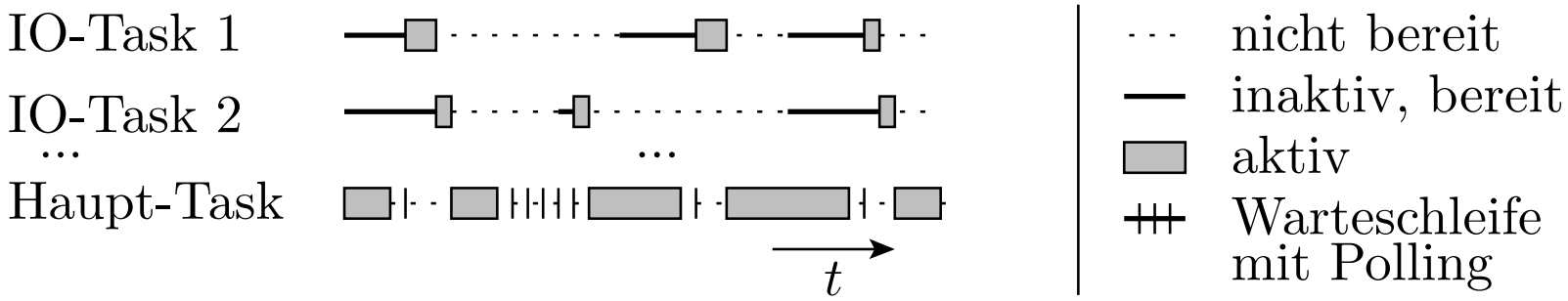
Zur Abstimmung der Ein- und Ausgabezeitpunkte muss ein EA-Gerät warten, bis der Rechner und der Rechner bis das EA-Gerät bereit ist. Dafür gibt es zwei Prinzipien:

- Polling: Zyklische Abfrage aller EA-Geräte durch den Rechner, ob Datenaustausch angefordert. Wenn ja, Verzweigung zum Programmbaustein für den Datenaustausch (bisher genutztes Prinzip).
- Interrupt: Gerät fordert Interrupt an. Rechner verzweigt sobald bereit zu einer Interrupt-Routine auf einer festen Adresse.

Unterbrechungen sind im Programm global und lokal (für jede Interrupt-Quelle einzeln) freizugeben.

Nach Freigabe kann das externe Gerät das Rechnerprogramm nach jedem Maschinenbefehl unterbrechen.

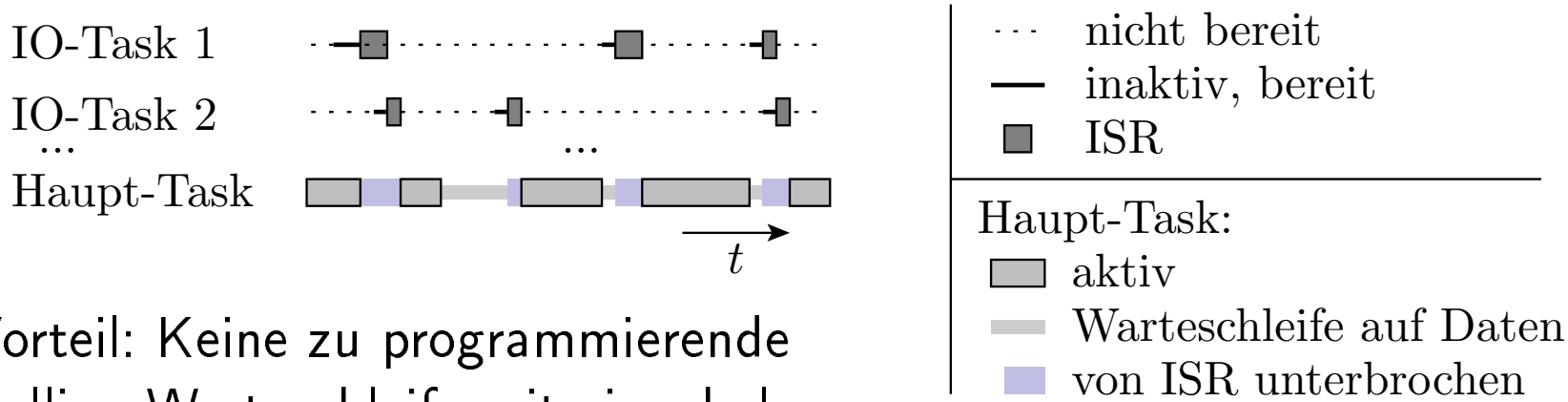
Bisheriges Task-Scheduling mit Polling



- Wenn der Haupt-Task keine Arbeit mehr hat, fragt er reihum die EA-Tasks ab, ob sie bereit sind. Wenn ja, Abarbeitung des EA-Tasks bis zum Start der nächsten Ein- oder Ausgabe.
- Falls kein EA-Task bereit, wiederholt der Haupt-Task die Abfrage zyklisch.
- Nach Abarbeitung aller bereiten EA-Tasks hat der Haupt-Task möglicherweise wieder Daten für seine Fortsetzung.
- Voraussetzung kurze Schrittfunktionen und hinreichend kurze Abfrageintervalle.

Task-Scheduling mit Interrupts

- Beim Setzen eines Ereignisbits (USART, Timer, ...) Unterbrechung des aktuellen Verarbeitung und Aufruf einer ISR (Interrupt-Service-Routine) auf einer festen Adresse.
- Nach Abarbeitung der ISR Programmfortsetzung.



Vorteil: Keine zu programmierende Polling-Warteschleife mit einzuhaltenden Obergrenzen für das Abfrageintervall.

Nachteil: Programmunterbrechung an einer zufälligen Stelle.

»Programmunterbrechung an einer zufälligen Stelle« verlangt Sonderbehandlungen bei der Programmierung, zum Teil vom Programmierer und zum Teil vom Compiler:

- Datenübergabe an ISR nur über globale Variablen möglich (Programmierer).
- ISR und unterbrochene Programmsequenz dürfen nicht dieselben Daten bearbeiten (Programmierer).
- Für jedes interrupt-auslösende Ereignis muss eine ISR ab der zugehörigen Befehlsspeicheradresse stehen (Programmierer).
- ISR dürfen nicht durch sich selbst und nur eingeschränkt durch andere ISR unterbrechbar sein (Compiler, Programmierer muss ISR als solche kennzeichnen und kann Ausnahmen erlauben).
- Alle von der ISR veränderten Register vorher sichern und vor dem Rücksprung zurücksetzen (Compiler).

Nutzung von Interrupts

- Einbindung des Headers avr/interrupt.h.
- Definition einer ISR[1]:

```
ISR(<Interrupt-Vektor>){...}
```

- Zur lokalen Freigabe eines einzelnen Interrupts ist jeweils ein bestimmtes Bit in einem SFR zu setzen, z.B. für den Vergleichsinterrupt A von Timer 3:

```
TIMSK3 |=1<<OCIE3A;
```

[1] ISR lassen sich an beliebigen Stellen in einer C-Datei des Projekts definieren und werden in Funktionen mit der Startadresse <interrupt-Vektor> ohne Übergabeparameter und Rückgabewert übersetzt.

- Interrupts global freigeben / sperren:

```
sei(); //Interrupts global freigeben
cli(); //alle Interrupts sperren
```

- Beim Aufruf einer ISR werden Interrupts automatisch global gesperrt, d.h. ISRs sind nicht unterbrechbar.
- Prinzipielle können ISRs unterbrechbar programmiert werden, indem zu Beginn sei() und am Ende cli() aufgerufen wird. Dann muss aber mindestens der zur ISR gehörende Interrupt lokal gesperrt sein, sonst rekursiver Selbstaufruf (Programmabsturz).

Tips und Tricks

Für ungenutzte Interrupts programmiert der Compiler einen Systemneustart. Vermeidbar durch Definition einer ISR für ungenutzte Interrupts:

```
ISR(BADISR_vect);
```

Zur Vereinfachung der Fehlersuche bei vergessenen ISR, falschem Interrupt-Vektor, falsche Interruptfreigabe eine »BAD-ISR« mit einer markanten Ausgabe programmieren.

Für Programmsequenzen, die in einer ISR bearbeitete Daten verwenden, z.B. Ein- und Ausgaben lesen oder schreiben, betreffende Interrupts mit folgender Sequenz sperren:

```
<Freigabebit speichern und löschen>
<unterbrechungsfreie Befehlsfolge>
<Werte der Freigabebits wiederherstellen>
```

ISRs kurz halten:

- max. einige 100 abzuarbeitende Maschinenbefehle,
- keine Warteschleifen, möglichst nur Datenübernahme oder Ausgabe.
- Aufwändigere Verarbeitungen in die von den anderen Tasks aufzurufenden Funktionen des Treibers, in denen die Datenübergabe erfolgt, auslagern.

Experimente

Experiment »F7-test_interrupt\test_interrupt«

Die nachfolgende Timer-ISR soll an Port J die LED-Ausgabe alle 0,5 s um eins vergrößern:

```
uint16_t LED_Ct;    //private Daten der ISR
```

ISR Timer 3 Vergleichsinterrupt A, Aufruf alle 1 ms

```
ISR(TIMER3_COMPA_vect){
 LED_Ct++;
 if (LED_Ct>=500){//alle 500 ms
  PORTJ++;          //Led-Ausgabe um eins erhöhen
  LED_Ct = 0;       //Zähler rücksetzen
 }
}
```

Das Hauptprogramm initialisiert Port J und C als Ausgänge. Port J wird mit 0 initialisiert und bei jedem 500sten ISR-Aufruf inkrementiert. Die LEDs an Port C zählen die Neustarts:

```
int main(void) {
 DDRJ  = 0xFF;  //LEDs an Port J als Ausgänge
 DDRC  = 0xFF;  //LED-Modul an Port C Ausgänge
 PORTJ = 0;
 PORTC++;
```

Tmr3 CTC-Modus, TIMER3_COMPA-Interrupt alle 1 ms, Tmr1 Normalmodus, Überlauf (OVF-) Ereignis nach ca. alle 8 s:

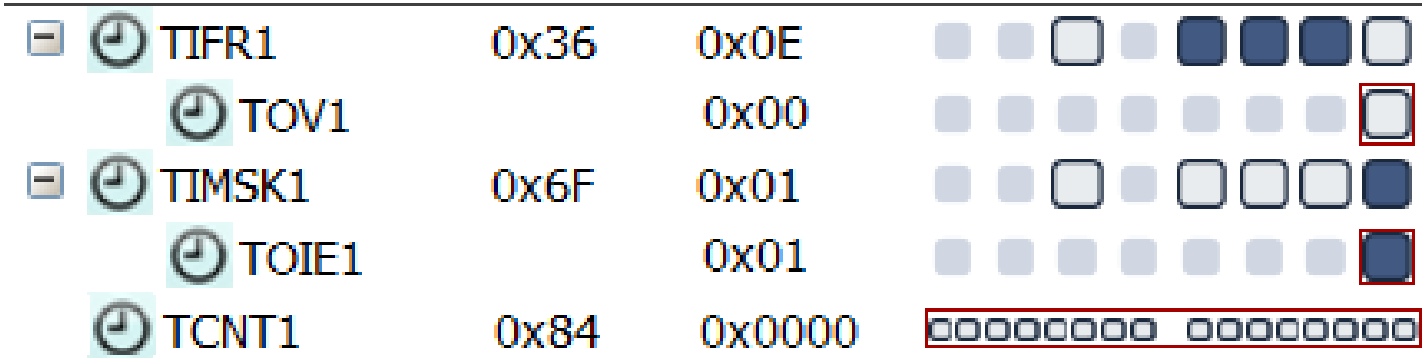
```
 TCCR3B = (1<<WGM32)|(0b001<<CS30);//Vorteiler 1
 OCR3A  = 8000;       //1 ms Aufrufperiode
 TIMSK3 = 1<<OCIE3A;//Tmr3-Comp.A-Interrupt ein
 TCCR1B = 0b101;      //Normalmod., Vorteiler 1024
 TIMSK1 = 0;          //Keine Freigabe Tmr1-Int.
 sei();               //Interruptfreig. global
 while(1);            //leere Hauptschleife, alle
}                     //Anweisungen auskommentiert
```

Experiment 1: Test der TIMER3_COMPA-ISR

- Projekt »F7-test_interrupt\test_interrupt« öffnen. Übersetzen.
- Start im Debugger. Continue. Port J zählt 2x je s hoch. Port C erhöht sich beim Start auf eins.

Experiment 2: Einmaliger TIMER3_OVF-Interrupt

- Anhalten. IO-View von Timer 1: Überlaufbit TOV1 löschen, Int.-Freigabe TOI1 setzen, Zähler TCNT1 löschen:



- Continue. Nach ca. 8 s: Inkrement der LEDs an Port J auf 0b10 und Rücksetzen der Onboard-LEDs, d.h. Neustart.
- Warum genau einen Neustart?

Experiment 3: Periodischer TIMER3_OVF-Interrupt

- Debugger stoppen.. Im Initialisierungsteil von Timer 1 Überlauf-Interrupt aktivieren. Ersatz »TIMSK1=0« durch:

```
TIMSK1=1<<TOIE1; //Ueberlauf-Interrupt ein
```

- Übersetzen. Start im Debugger  . Continue  .
- Die LEDs an Port C zählen jetzt ca. alle 8 s weiter und die LED-Ausgabe an Port J werden dabei gelöscht.
- Warum jetzt periodischer Neustart?

Timer1-Überlauf-Interrupt wird beim Neustart nicht mehr deaktiviert, sondern aktiviert.

Experiment 4: Ergänzung einer BADISR

- Debugger stoppen . BADISR einkommentieren:

```
ISR(BADISR_vect){
 PORTC++;  //Led-Ausg. Port C hochzaehlen
}
```

- Übersetzen. Start im Debugger . Continue .
- Die LEDs an Port C zählen ca. alle 8 s weiter, aber der LED-Zählstand an Port J wird dabei nicht gelöscht.

Keine Neuinitialisierung mehr.

Experiment 5: ISR-Datenzugriff ohne ISR-Deaktivierung

- Debugger stoppen. Nur Invertierung von PJ7 in der Endlosschleife einkommentieren:

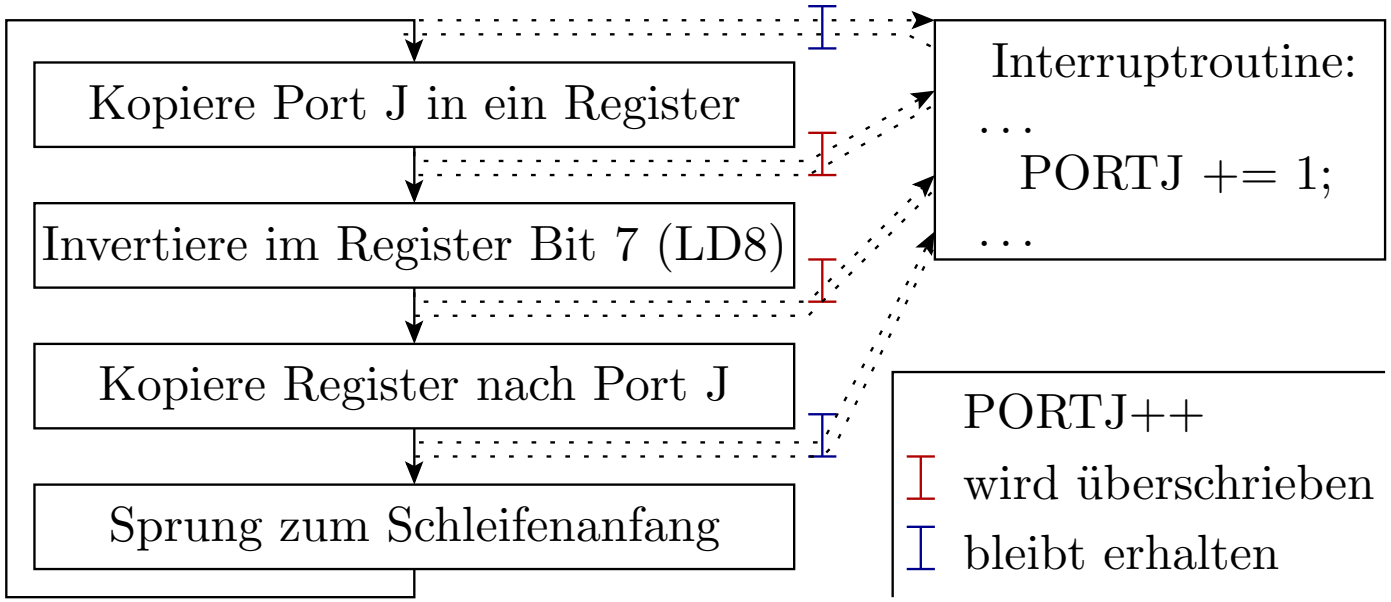
```
while(1){              //Hauptschleife
 PORTJ ^= 0x80;        //LED8 umschalten
}
```

- LD8 invertiert schnell (dunkleres Dauerleuchten). Die Dauer zwischen den Schaltvorgängen der anderen LEDs von Port J wächselt zufällig zwischen 0,5 s und 1 s. Warum?

Das Hauptprogramm und die ISR ändern beide den Wert von Port J. Die C-Anweisung zum Invertieren von LED 8 besteht aus mehreren Maschinenanweisungen. Zwischen bestimmten Anweisungen ist der Werte der Variablen Port J bereits gelesen, aber der neue Wert noch nicht geschrieben. Wenn die ISR die Operation »PORTJ++« dort einfügt, wird das Ergebnis durch den alten weiterverarbeiteten Wert überschrieben.

Der Schleifenkörper besteht mindestens aus den Schritten:

- Port J lesen,
- Wert bearbeiten,
- Wert schreiben und
- Sprung zum Schleifenbeginn:



An 50% der Unterbrechungsmöglichkeiten wird die Erhöhung von PJ in der ISR vom Rückschreibwert für die Invertierung von PJ7 überschrieben.

Unterbrechungsfreie Sequenzen

Zu Vermeidung, dass Hauptprogramm und ISR nebenläufig gleiche Daten bearbeiten, ist bei der Verwendung von Daten, die auch eine ISR nutzt, die ISR-Freigabe abzuschalten. Beschreibung einer »nicht unterbrechbaren Sequenz«:

```
<Freigabebit speichern und löschen>
<unterbrechungsfreie Befehlsfolge>
<Werte der Freigabebits wiederherstellen>
```

Erweiterung der Anweisungsfolge in der Endlosschleife:

```
uint8_t INTZ = TIMSK3;   //Int.-Freigabe sichern
TIMSK3 &= ~(1<<OCIE3A); //Int.-Freigabe löschen
PORTJ ^= 0x80;           //LD8 umschalten
TIMSK3 = INTZ;           //Int.-F. wiederherst.
```

Nach Neuübersetzung und Neustart keine Zählanormalitäten mehr.

Aufgaben

Aufgabe 7.1: Interrupt-Experimente

Im Projekt F3-echo\echo« mit dem Debugger im Register UCSR2B das Interrupt-Freigabebit RXCIE2 setzen. Was passiert?

- Kontrollieren Sie, ob das Programm bei Datenempfang neu startet.
- Wird die Abfrage von »UCSR2A & (1<<RXC2)« jemals wahr.
- Warum (nicht)?

Aufgabe 7.2: LCD-Interrupt-Ausgabe

- Schreiben Sie die Schrittfunktion des Treibers comsf_lcd in eine ISR für den Puffer-Frei-Interrupt von USART1 um:

```
ISR(USART1_UDRE_vect){ //Puffer-frei ISR
   ...
}
```

- Passen Sie die Initialisierungsfunktion und den Testrahmen an.

Hinweise:

- Lokale Interruptfreigabe:

```
UCSR1B |= (1<<UDRIE1);
```

- Im Testrahmen entfällt der Aufruf der Schrittfunktion. Rest unverändert.

Aufgabe 7.3: Sonar-ISR

- Ändern Sie die Schrittfunktion für den Sonartreiber aus dem Projekt »F5-comsf« in der Datei »comsf_sonar« in eine ISR für den Empfangsinterrupt von USART1 um:

```
ISR(USART1_RX_vect){...}
```

- Passen Sie die Initialisierungsfunktion und den Testrahmen an. Lokale Interruptfreigabe:

```
UCSR1B |= (1<<RXCIE1);
```

Aufgabe 7.4: gepufferter PC-Treiber

Für den Treiber für die PC-Kommunikation wäre es günstig, wenn übergebene Daten in einen Puffer abgelegt und vom Programm nach dem FIFO-Prinzip (**F**irst **I**n **F**irst **O**ut) abgeholt werden. Ein FIFO wird als Ringpuffer programmiert. Ein Ringpuffer ist ein Feld mit einem Schreib- und einem Lese-Zeiger. Geschrieben wird auf die Adresse des Schreibzeigers. Danach der Schreibzeiger erhöhen. Beim Lesen wird von der Adresse des Lesezeigers gelesen und der Lesezeiger erhöht. Wenn ein Zeiger über das Feldende hinauszeigt, wird er auf den Anfang rückgesetzt. Für Ringpuffer leer und voll sind Sonderbehandlungen zu programmieren.

Entwickeln Sie für die PC-Kommunikation (JH, USART2) einen ISR-basierten Treiber mit je einem 8-Byte-Ringpuffer für empfangene und zu sendende Daten.