Informatikwerkstatt, Foliensatz 9
Interrupts

G. Kemnitz

Institut für Informatik, TU Clausthal (IW-F9)

26. November 2024

Inhalt:

- Task-Scheduling mit Interrupts
- Experimente
- Aufgaben

Interaktive Übungen:

1. Experimente mit Interrupts (test_interrupt)

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:

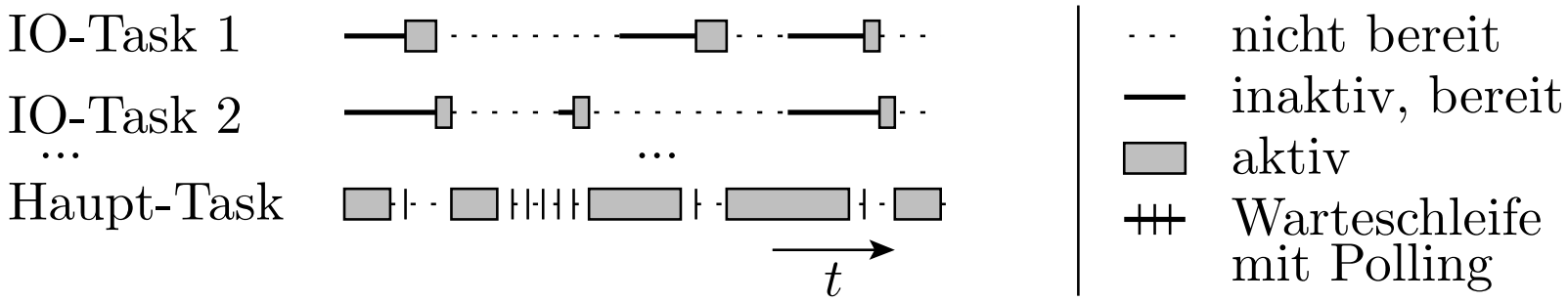
- Polling: Zyklische Abfrage aller EA-Geräte durch den Rechner, ob Datenaustausch angefordert. Wenn ja, Verzweigung zum Programmbaustein für den Datenaustausch (bisher genutztes Prinzip).
- Interrupt: Gerät fordert Interrupt an. Rechner startet, sobald bereit, eine Interrupt-Service-Routine (ISR) von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

Programm

- erwartet die ISR an vorgesehener Adresse und
- muss Unterbrechungen lokal (für jede Interrupt-Quelle einzeln)
- und global (für alle Interrupt-Quellen) freizugeben.

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

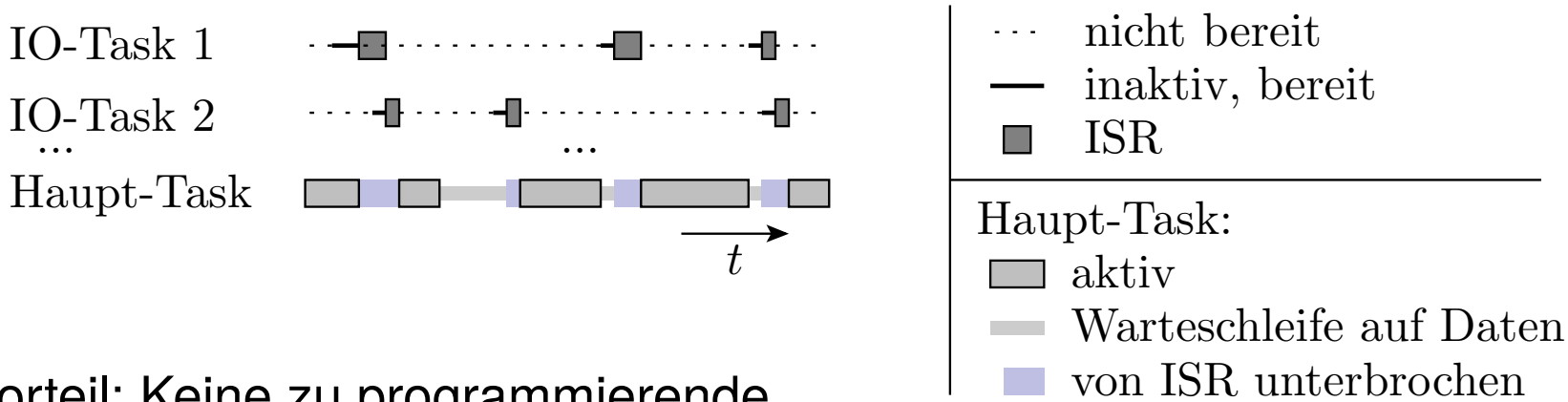
Bisheriges Task-Scheduling mit Polling



- 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 EA-Tasks, die bereit sind, 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 der 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 Abfrageintervallen.

Nachteil: Programmunterbrechung an zufälligen Stellen.

1. Task-Scheduling mit Interrupts

Programmunterbrechung an zufälligen Stellen verlangt besondere Maßnahmen bei der Programmgestaltung:

- Datenaustausch mit ISR nur über globale Variablen.
- ISR und unterbrochene Programmsequenz dürfen nicht zeitgleich gleiche Daten bearbeiten.
- Für jedes interrupt-auslösende Ereignis muss eine ISR ab der zugehörigen Befehlsspeicheradresse stehen.
- ISR dürfen nicht durch sich selbst und nur eingeschränkt durch andere ISR unterbrechbar sein.
- Registerwerte, die die ISR verändert, sind zu Beginn zu sichern und vor dem Rücksprung wiederherzustellen.

Nutzung von Interrupts

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

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

- Zur Freigabe eines Interrupts sind
  - sein Freigabebit zu setzen, z.B. für den Vergleichsinterrupt A von Timer 3:

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

  - und Interrupts sind global freizugeben.

[1] ISR lassen sich an beliebigen Stellen in einer C-Datei des Projekts definieren. Der Compiler erzeugt daraus eine Funktionen mit der Startadresse <interrupt-Vektor> ohne Übergabeparameter und ohne Rückgabewert. Beim Start einer ISR wird automatisch das globale ISR-Freigabebit gelöscht und beim Rücksprung wieder gesetzt.

- Interrupts global freigeben und sperren:

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

- Prinzipiell können ISRs unterbrechbar programmiert werden, indem zu Beginn sei() und am Ende cli() aufgerufen wird.
- Während das globale Freigabebit in einer ISR gesetzt ist, muss das lokale Freigabebit der ISR »0« sein. Sonst Gefahr, dass sich die ISR in einer Endlos-Rekursion immer wieder selbst unterbricht (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>
<Ursprungswert Freigabebit wiederherstellen>
```

ISRs kurz halten:

- nicht blockierend, ohne Warteschleifen,
- möglichst nur Datenaustausch,
- max. einige 100 abzuarbeitende Maschinenbefehle,
- Zustandsautomat (fallabhängige Abarbeitungssequenz).

Aufwändigere Verarbeitungen besser in die öffentlichen Funktionen der Treiber oder regelmäßig auszuführende Schrittfunktionen auslagern.

Experimente

Experiment »F9-test_interrupt\test_interrupt«

1. Test eines Programms mit einer Timer-3-Comp-A-ISR, die alle 0,5 s die LED-Ausgabe an Port J hochzählt.
2. Wirkung eines einmaligen unerwarteten (unbehandelten) Interrupts, hier der Freigabe des Timer-1-Überlauf-Interrupt mit dem Debugger.
3. Wirkung eines dauerhaft aktivierten unbehandelten Interrupts, hier des Timer-1-Überlauf-Interrupts alle 8 s.
4. Bad-ISR zum Abfangen der unbehandelten Interrupts, hier für den periodischen Timer-1-Überlauf-Interrupt.
5. Beispiel für die Fehlerwirkung, wenn ISR-Daten im Hauptprogramm ohne Sperren der ISR bearbeitet werden.
6. Fehlerbeseitigung mit Unterbrechungssperren.

Vorbereitung: PmodLED an Port C stecken.

Timer-3-Comp-A-ISR

Die nachfolgende Timer-ISR wird von dem im CTC-Modus mit Periode 1 ms initialisierten Timer 3 beim Setzen des Vergleichsbits OCF3A gestartet:

```
uint16_t LED_Ct;   //private Daten der ISR
...
ISR(TIMER3_COMPA_vect){
 LED_Ct++;
 if (LED_Ct>=500){//alle 500 ms
  PORTJ++;         //Led-Ausgabe um eins erhöhen
  LED_Ct = 0;      //Zähler rücksetzen
 }
}
```

Bei jedem 500sten Aufruf (alle $500 \cdot 1$ ms) erhöht sich die LED-Ausgabe an Port J um eins.

Hauptprogramm

```
1  int main(void) {
2   DDRJ  = 0xFF;       //LEDs an Port J als Ausgänge
3   DDRC  = 0xFF;       //LED-Modul an Port C Ausgänge
4   PORTJ = 0;          //Port J zählt ISR-Aufrufe
5   PORTC++;            //Port C zählt die Neustarts
6   TCCR3B = (1<<WGM32)|(0b001<<CS30);//Vorteiler 1
7   OCR3A  = 8000;       //1 ms Aufrufperiode
8   TIMSK3 = 1<<OCIE3A;//Tmr3-Comp.A-Interrupt ein
9   TCCR1B = 0b101;     //Normalmod., Vorteiler 1024
10  TIMSK1 = 0;         //Keine Freigabe Tmr1-Int.
11  sei();              //Interruptfreig. global
12  while(1);           //leere Hauptschleife, alle
13 }                    //Anweisungen auskommentiert
```

- Zeile 9 und 10 sind für Experiment 2
- In der Hauptschleife passiert nichts.

Experiment 1: Test eines Programms mit ISR

- LED-Modul an Port C.
- Projekt »F9-test_interrupt\test_interrupt« öffnen.
- Übersetzen. Start im Debugger . Continue .

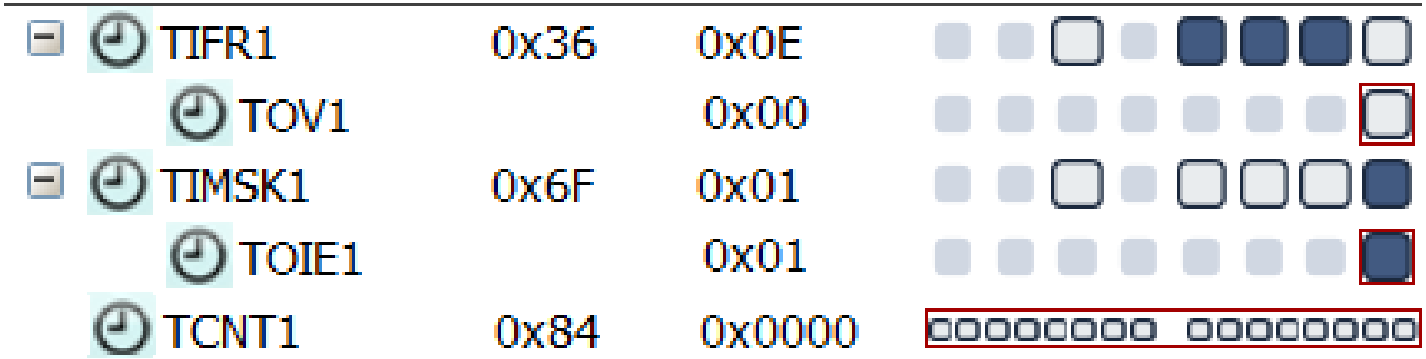
- Port C erhöht sich beim Start auf eins.
- die LED-Ausgabe an Port J erhöht sich alle 0,5 s um eins.

```
int main(void) {
 ...
 PORTC++;              //zählen der Neustarts
 ...
 while(1);             //leere Hauptschleife
}

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

Experiment 2: Einmalige Bad-ISR

- 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. Rücksetzen der LEDs an Port J, d.h. Neustart.

```
int main(void) {
 ...
 PORTJ = 0;          // Port J zählt ISR-Aufrufe
 PORTC++;            // Port C zählt die Neustarts
 ...
 TIMSK1 = 0;         // Keine Freigabe Tmr1-Int.
 sei();              // Interrupt-Freigabe
 while(1);           // leere Hauptschleife
}
```

2. Experimente

- Warum ein Neustart?
- Warum nicht mehrere Neustarts?

```
int main(void) {
 ...
 PORTJ = 0;      // Port J zählt ISR-Aufrufe
 PORTC++;        // Port C zählt die Neustarts
 ...
 TIMSK3 = 1<<OCIE3A;//Tmr3-Comp.A-Interrupt ein
 TIMSK1 = 0;     // Keine Freigabe Tmr1-Int.
 sei();          // Int.-Freig.,
 while(1);       // leere Hauptschleife
}

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

Experiment 3: aktivierter unbehandelter Interrupt

- Debugger stoppen. Im Initialisierungsteil von Timer 1 Überlauf-Interrupt aktivieren:

```
TIMSK1=1<<TOIE1; // statt TIMSK1=0
```

- Übersetzen. Start im Debugger . Continue .

Die LEDs an Port C zählen jetzt ca. alle 8 s weiter. Gleichzeitig wird die LED-Ausgabe an Port J gelöscht.

- Offenbar periodischer Neustart. Warum?

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

Experiment 4: Ergänzung einer Bad-ISR

- Debugger stoppen . BADISR einkommentieren:

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

- Übersetzen. Start im Debugger . Continue .

- Die LEDs an Port C zählen beim Start und dann ca. alle 8 s weiter.
- Der LED-Zählstand an Port J wird dabei nicht gelöscht.
- Statt im Start-Up-Code Ausführung von »PORT C++« in der Bad-ISR.

Exp. 5: ISR-Datenzugriff ohne ISR-Sperrung

- 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 zufällig 0,5 s oder 1 s?

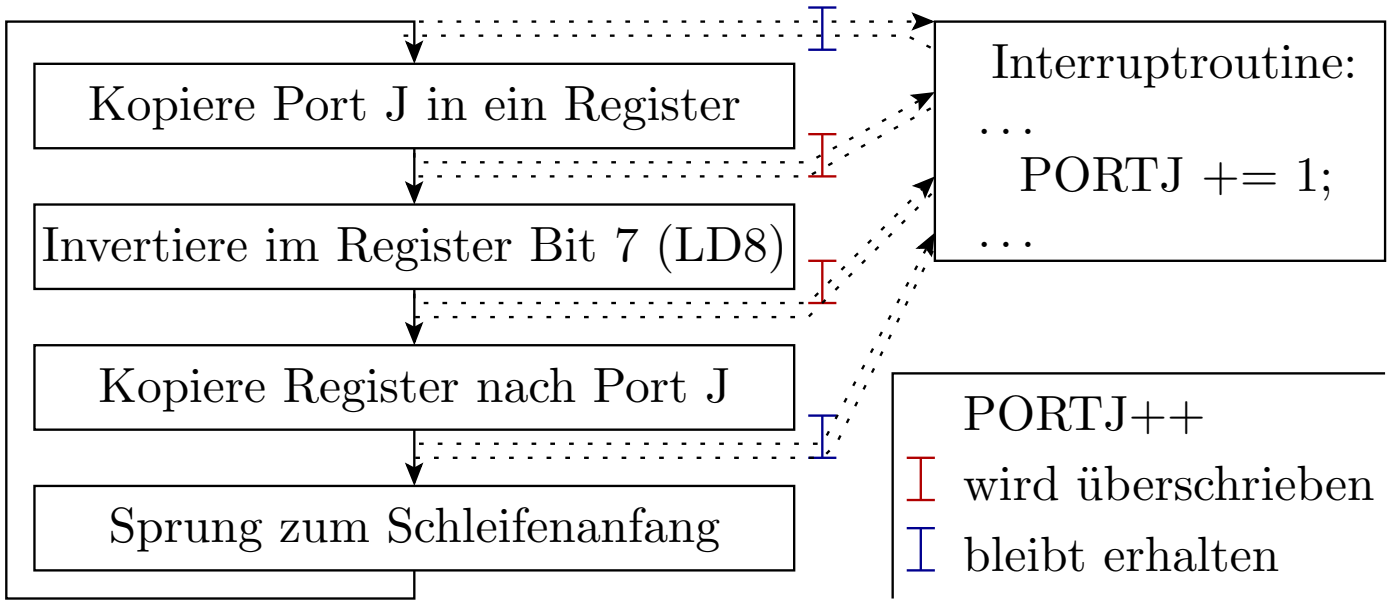
- Weil Port J zu den privaten Daten der ISR gehört und
- der Schleifenkörper von

```
while(1) PORTJ ^= 0x80;
```

aus mehreren Maschienenbefehlen besteht.

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 Sequenz

Zur 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 Hauptschleife:

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

Nach Neuübersetzung und Neustart keine Zählanormalitäten mehr.

Aufgaben

Aufgabe 9.1: Interrupt-Experimente aus der Vorlesung

Arbeiten Sie die Experimente ab Folie 16 ab.

Aufgabe 9.2: Interrupt-Experimente mit Echo-Programm

1. Starten Sie das Projekt P04\F4-echo\echo« im Debugger. Testen Sie mit HTerm, dass gesendete Zeichen zurückkommen.
2. Halten Sie das Programm an. Setzen Sie im Register UCSR2B das Interrupt-Freigabebit RXCIE2. Starten Sie das Programm wieder. Warum empfängt HTerm ein Zeichen weniger, als es sendet und welches Zeichen kommt nicht zurück?
3. Kontrollieren Sie mit einem Unterbrechungspunkt vor der Hauptschleife, dass das Programm nach Setzen von RXCIE2 im Debugger beim ersten Datenempfang neu startet.
4. Verhindern Sie den Neustart mit einer Bad-ISR. Kontrollieren Sie im Debugger mit einem Unterbrechungspunkt, dass nach Setzen von RXCIE2 nach jedem Zeichenempfang die Bad-ISR ausgeführt wird.

5. Wird die Bad-ISR nach jedem Zeichenempfang einmalig oder immer wieder ausgeführt, sprich wird bei Ausführung in der Bad-ISR das Ereignisbit »RXC2« im Register »UCSR2A« gelöscht. Wenn nicht, ergänzen Sie den Befehl zum Löschen von »RXC2« in der Bad-ISR.
6. Warum wird, wenn in der Bad-ISR des Empfangsbit »RXC2« gelöscht wird, aus der Warteschleife

```
//warte auf den Empfang eines Bytes
while(!(UCSR2A &(1<<RXC2)));
```

eine Endlosschleife.

Aufgabe 9.3: LCD-Interrupt-Ausgabe

1. 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
  ...
}
```

2. Passen Sie die Initialisierungsfunktion und den Testrahmen an.

Hinweise:

- Lokale Interruptfreigabe:

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

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

Aufgabe 9.4: Sonar-ISR

1. Ändern Sie die Schrittfunktion für den Sonartreiber aus dem Projekt »F7-comsf« in der Datei »comsf_sonar« in eine ISR für den Empfangsinterrupt von USART1 um:

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

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

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

Aufgabe 9.5: 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 Schreibzeiger erhöhen. Beim Lesen wird von der Adresse des Lesezeigers gelesen und der Lesezeiger erhöht. Wenn ein Zeiger über das Feldende hinaus zeigt, wird er auf den Anfang rückgesetzt. Des weiteren benötigen Sie eine Variable für den Füllstand, die beim Schreiben erhöht, beim Lesen verringert und für die Sonderfälle Puffer voll/leer ausgewertet wird.

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