Informatikwerkstatt, Foliensatz 5
Timer und Interrupts

G. Kemnitz

Institut für Informatik, TU Clausthal (IW-F5)

14. Dezember 2015

Inhalt Foliensatz 5

Wiederholung

Wiederholungsaufgabe 5.1

Bei den Treibern vom vorherigen Foliensatz

1. Wozu dienen die Initialisierungsfunktionen?
2. Wozu dienen die Schrittfunktionen und an welcher Stelle im Verarbeitungsfluss sind sie einzubinden?
3. Welche Aufgaben haben die Send- und die Get-Funktionen?
4. Was bedeutet blockierungsfreies Lesen oder Schreiben.
5. Warum müssen in einem Mehr-Task-System alle IO-Lese- und Schreibfunktionen blockierungsfrei arbeiten?

Wiederholungsaufgabe 5.2

```
void sendByte(uint8_t dat){
  while (!(UCSR2A & (1<<UDRE2)));// warte Puffer frei
  UDR2 = dat;                    // Byte uebergeben
}
```

1. Wozu dient und wie funktioniert diese Funktion?
2. Vervollständigen Sie die nachfolgende Funktion zum Versenden einer vorzeichenfreien 2-Byte-Zahl.

```
#define SENDBUF_FREE (UCSR2A & (1<<UDRE2)
void send2Byte(uint16_t dat){
 while (!SENDBUF_FREE));          // warte bis Puffer frei




 return;
}
```

Timer

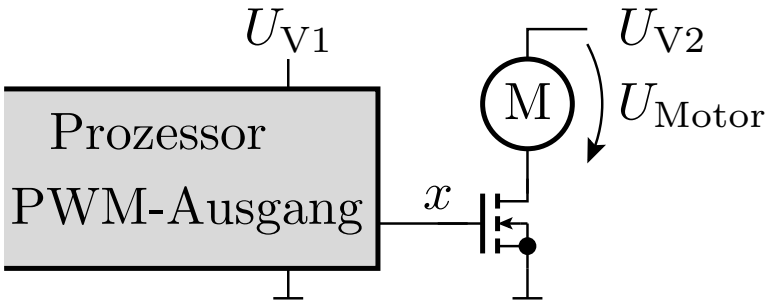
Timer

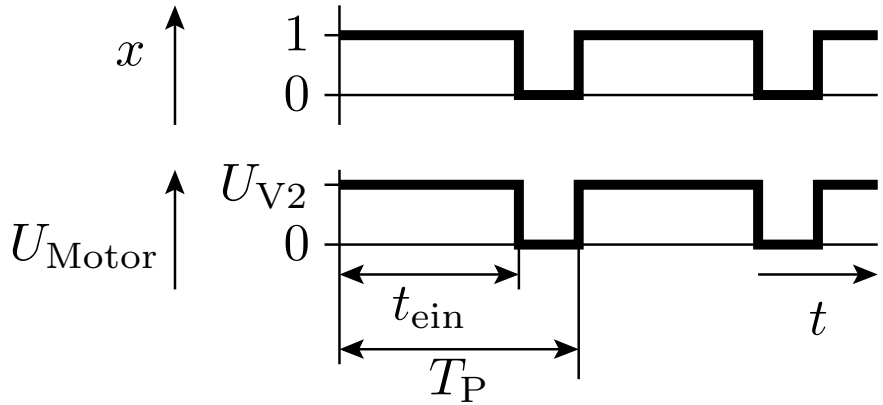
Ein Timer ist eine Hardware-Einheit aus Zähl-, Vergleichs-, Konfigurationsregistern, ... zur

- Erzeugung von Wartezeiten,
- zeitgesteuerten Ereignisabarbeitung,
- Erzeugung pulsweitenmodulierter (PWM-) Signale und
- Pulsweitenmessung.

PWM-Signale dienen

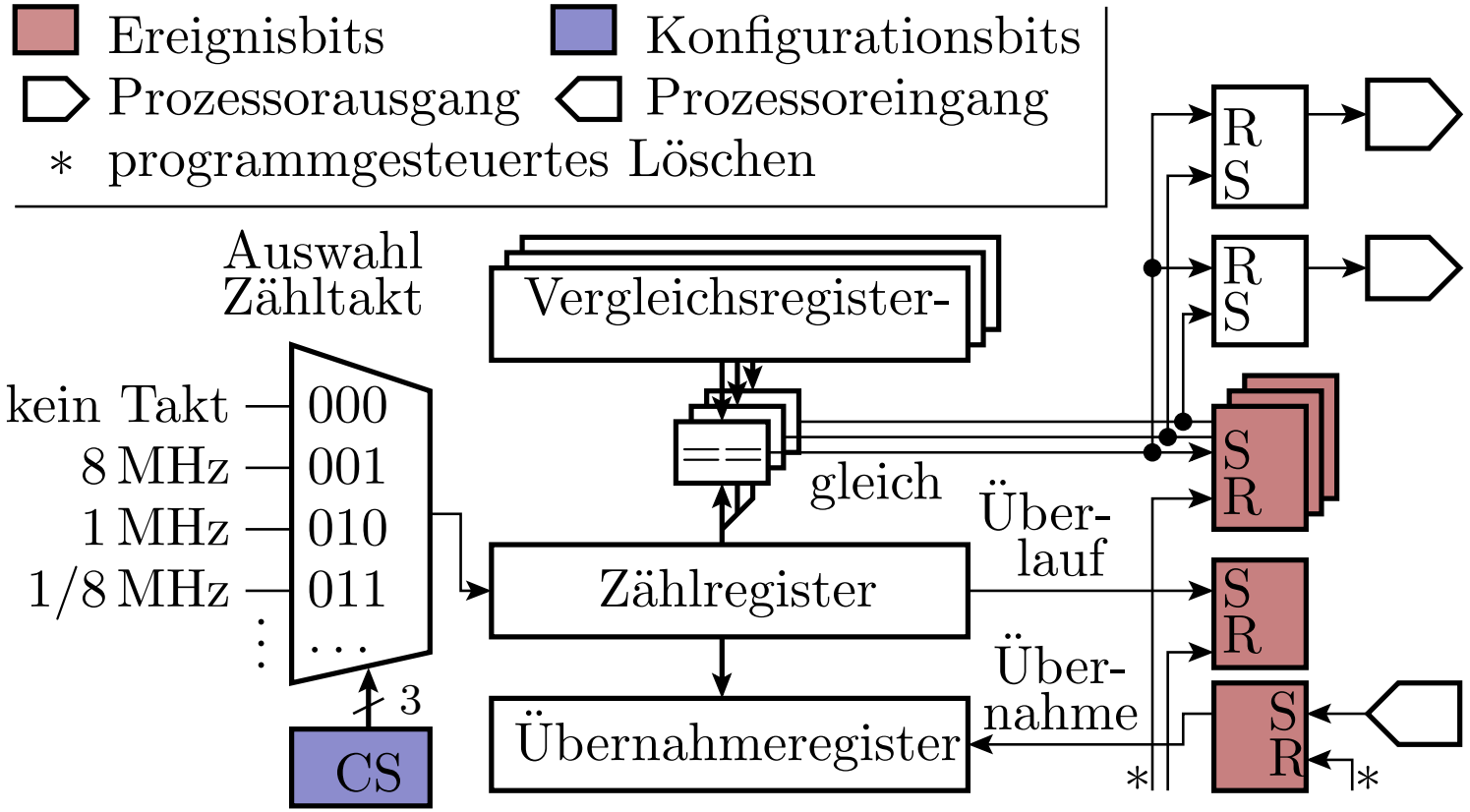
- zur Informationsübertragung z.B. an Modellbauservos und
- zur stufenlosen Leistungssteuerung, z.B. unserer Motoren.





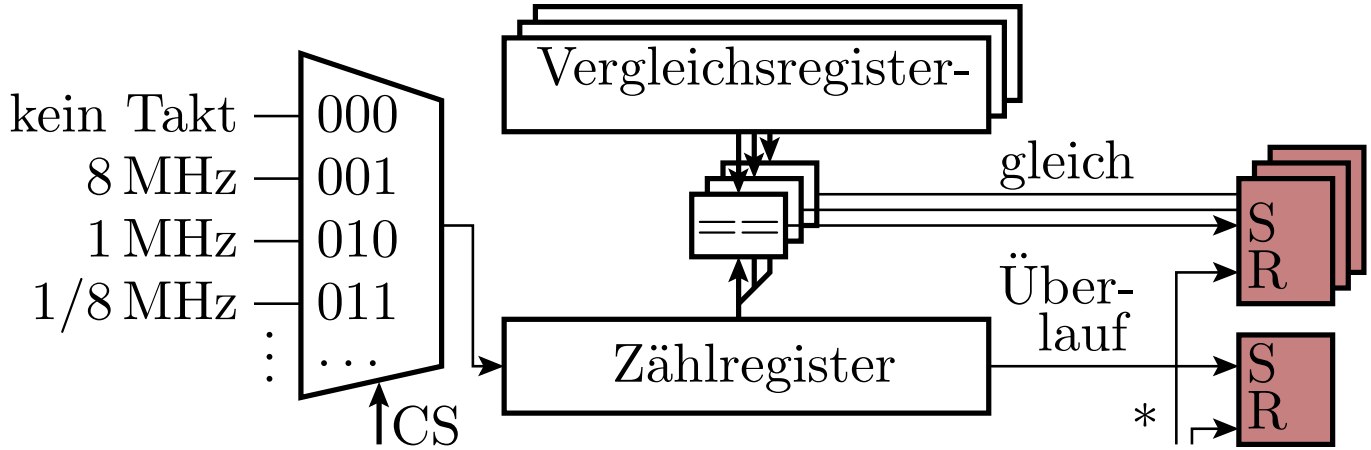
Funktionsweise

Aufbau und Funktionsweise eines Timers



- Kern eines Timers ist ein Zählregister mit einem vom Programm zuschaltbaren programmierbaren Takt.
- Die Ereignisbits (Überlauf, Gleichheit, externe Flanke) sind vom Programm les- und löschbar.

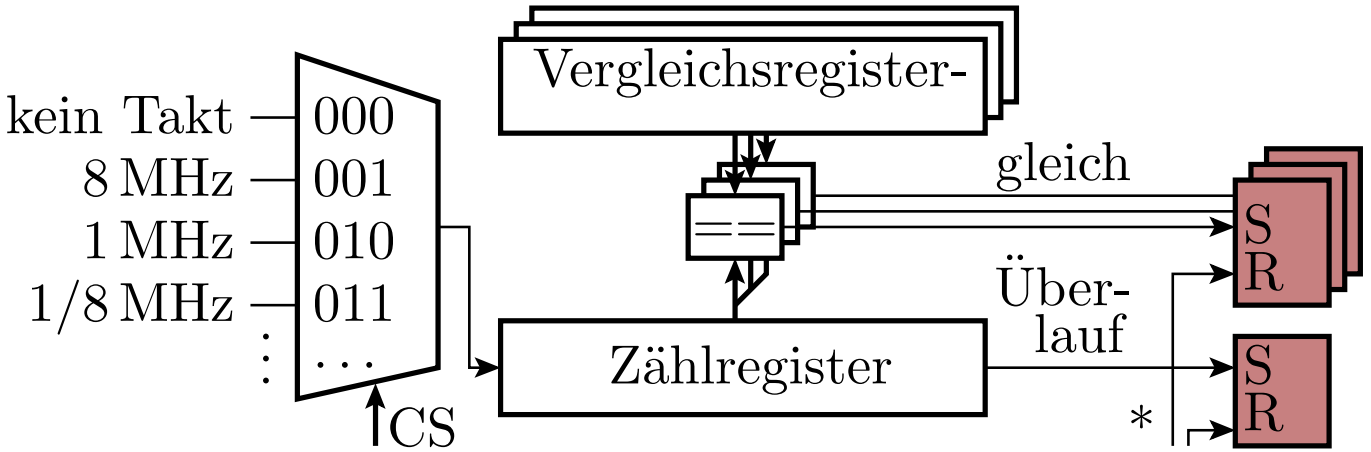
Normalmodus



- Zählregister zählt zyklisch bis zum Überlauf.
- Beim Überlauf wir eine Überlaufbit und bei Gleichheit mit einem Vergleichsregister ein Vergleichsbit gesetzt.
- Beispiel Wartefunktion:

```
void wait(int32_t tw){
 <berechne und setze Taktauswahl und Vergleichswert>
 <Lösche Zähler und Vergleichsereignisbit>
 <warte bis Ereignisbit wieder gesetzt ist>
 <schalte Zähltakt aus> }
```

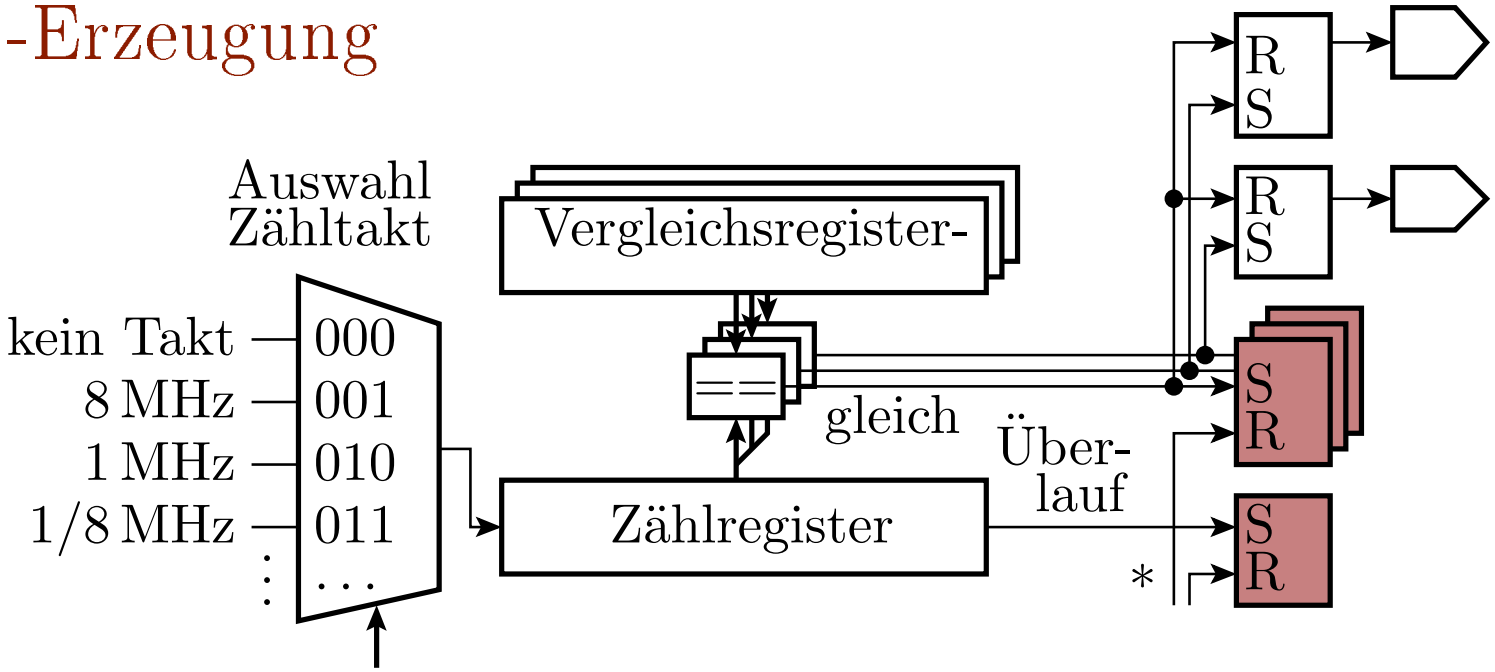
CTC- (Clear on Compare) Modus



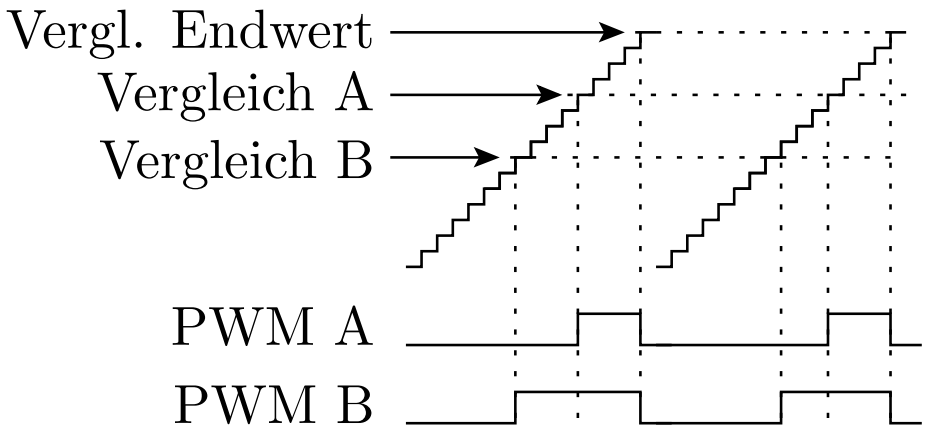
- Zähler wird bei Gleichheit mit einem der Vergleichsregister rückgesetzt.
- Auslösung zyklischer Ereignisse, z.B. Uhrenprozess:

```
void Schrittfunktion Uhr(){
  if (<Vergleichs-Rücksetz-Ereignis>)
   <lösche Ereignisbit(s) und schalte Uhr weiter>
}
```

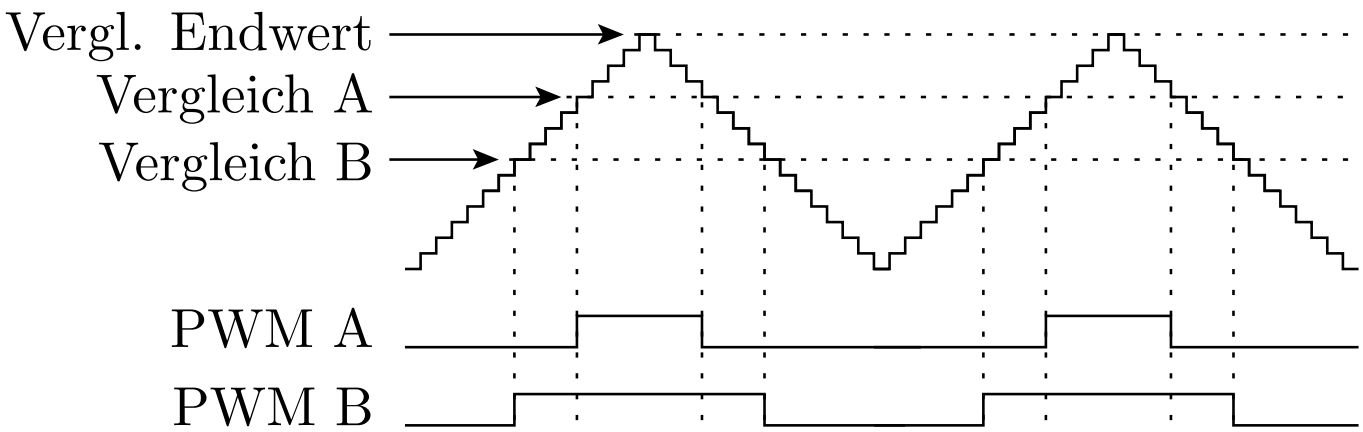
PWM-Erzeugung



- Vergleichsereignis setzt Zählerrücksetzereignis (Überlauf oder CTC) löscht Ausgabe.
- Pulsgenerierung z.B. zur Motoransteuerung ohne Schrittfunktion.



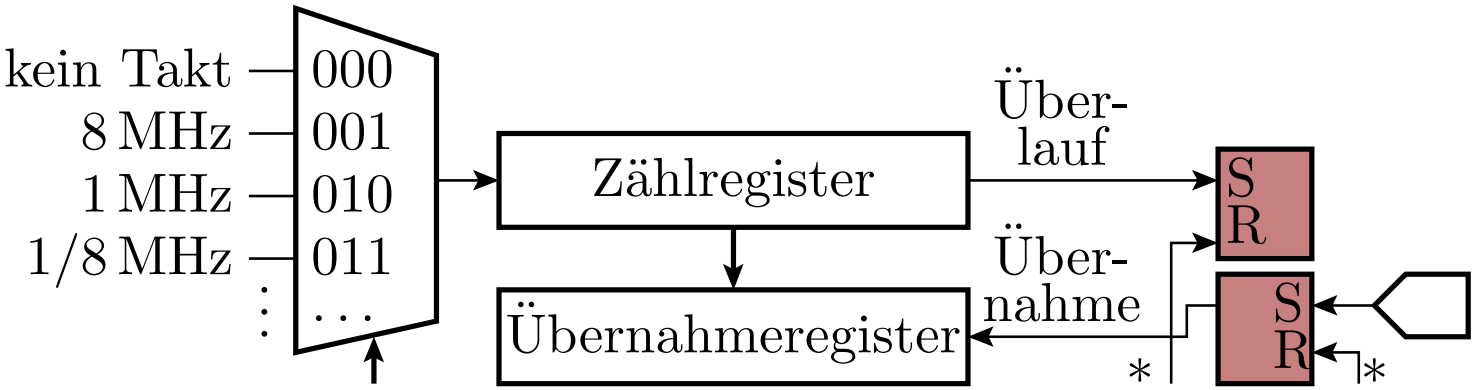
Symmetrische PWM[1]



- Endvergleichswert schaltet die Zählrichtung um.
- Bei Gleichheit und Hochzählen wird die Ausgabe ein- und bei Gleichheit und Abwärtszählen ausgeschaltet.
- Bei dieser und der vorherigen PWM kann auch eine invertiert Ausgabe programmiert werden, so dass der Vergleichswert statt der Ausschalt-, die Einschaltzeit festlegt.

[1] Im Datenblatt unseres Prozessors ist das die phasenrichtige und die vorhergehende normale PWM die schnelle (Fast-) PWM.

Pulsweitenmessung



- Externes Ereignis (Schaltflanke) bewirkt Übernahme des Zählwerts in das Übernahmeregister.
- Programmgesteuerte Differenzbildung der Übernahmewerte zwischen den Übernahmeereignissen.

Der Zeitmessmodus von Timern wird in dieser Veranstaltung nicht genutzt.

Timer des ATMega2560

- Zwei 8-Bit Timer (0 und 2).
- Vier 16-Bit-Timer (1, 3, 4 und 5).

Die Bit-Anzahl beschreibt die Größe der Zähl- und Vergleichsregister.

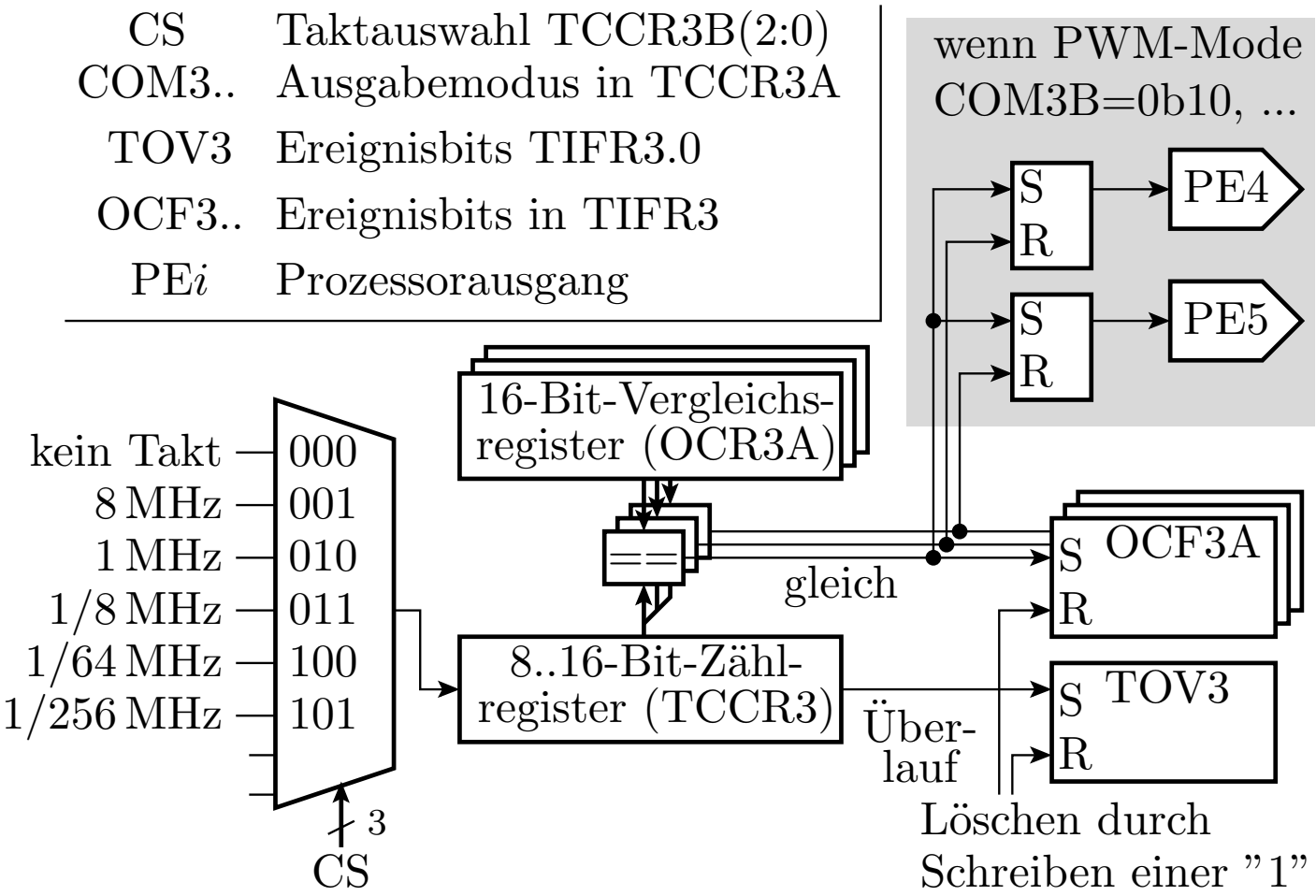
Nutzung der Timer in den Beispielprojekten:

- Timer 0: Treiber »wegmess« Abtastintervall.
- Timer 1: Treiber »comir_tmr« Programmuhr und Wartezeitzähler.
- Timer 3:
  - Timer- und Interrupt-Experimente.
  - Treiber »comir_PC« Empfangs-Timeout.
- Timer 5: Treiber »pwm« Motor-PWM.

Die übrigen Timer können für eigene Programme, z.B. als Timeout-Zähler für den Bluetooth-Empfang genutzt werden.

Programmbeispiele mit Timer 3

Timer 3: 16-Bit, Normal-, CTC-, PWM-Mode



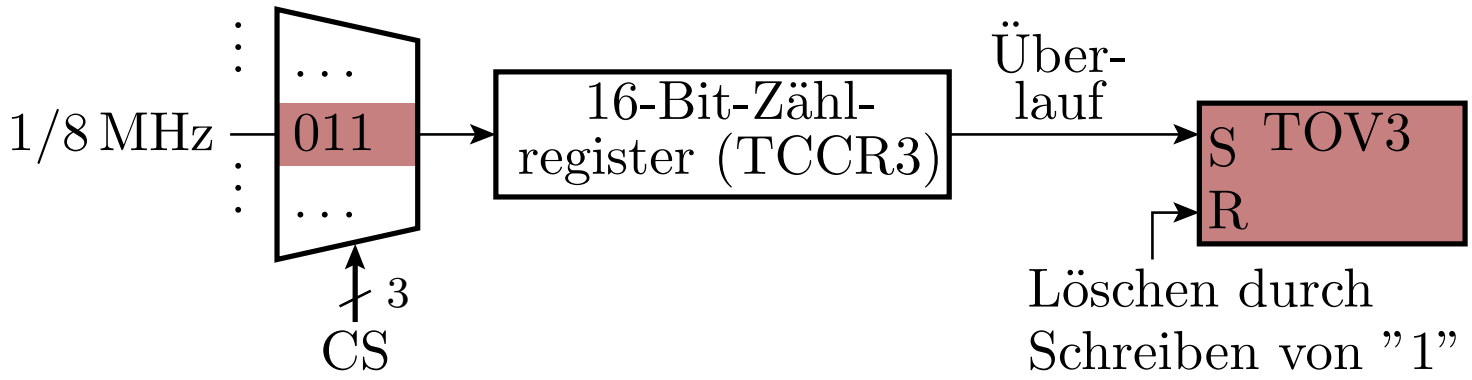
- Modusauswahl: WGM(3:0) in TCCR3A und TCCR3B.

Betriebsarten (Auswahl)

WGM	Betriebsart	max. Zählwert	OCR-Übernahme	Setzen von TOV
0b0000	normal	0xFFFF	sofort	0xFFFF
0b0100	CTC	OCR3A	sofort	OCR3A
0b0001	sym. PWM$^{(*1)}$, 8 Bit	0x00FF	0x00FF	0x0000
0b0011	sym. PWM$^{(*1)}$, 10 Bit	0x03FF	0x03FF	0x0000
0b1011	sym. PWM$^{(*1)}$, OCR	OCR3A	OCR3A	0x0000
0b0101	fast PWM$^{(*2)}$ 8 Bit	0x00FF	0x0000	0x00FF
0b0111	fast PWM$^{(*2)}$ 10 Bit	0x03FF	0x0000	0x03FF
0b1111	fast PWM$^{(*2)}$, OCR	OCR3A	0x0000	OCR3A

$^{(*1)}$ symmetrische oder phasenausgerichtete PWM.
$^{(*2)}$ Schnelle oder normale PWM.

LED mit Timer hochzählen



- Timer im Normalmodus (WGM(3:0)=0) und Zähltakt:

$$f_{\text{count}} = \frac{f_{\text{Proc}}}{64} = \frac{1}{8}\,\text{MHz}$$

- Bei jedem Überlauf des Zählregisters nach 2^{16} Zählschritten, Überlaufsereignisbit löschen und LED-Ausgabe weiterzählen. LED-Zählfrequenz:

$$f_{\text{LED}} = \frac{f_{\text{cont}}}{2^{16}} = 1{,}9\,\text{Hz}$$

```
#include <avr/io.h>
int main(void){
  TCCR3A = 0;          // WGM3[1:0] = 0
  TCCR3B = 0b011;      // WGM3[3:2] = 0, CS3=0b011
  DDRJ = 0xFF;
  while(1){
    if (TIFR3 & (1<<TOV3)){ // Warte auf Überlauf
      PORTJ++;              // Erhöhe Led-Ausgabe
      TIFR3 = (1<<TOV3);    // Lösche Überlaufbit
    }
  }
}
```

- Projekt F5-1_test_time\test_timer öffnen.
- Alle außer erste Main-Funktion auskommentiert lassen.
- Übersetzen. Start im Debugger . Continue .
- LED-Zählfrequenz kontrollieren.
- Anhalten . Unterbrechungspunkt wie im Bild setzen.
- Continue bis .

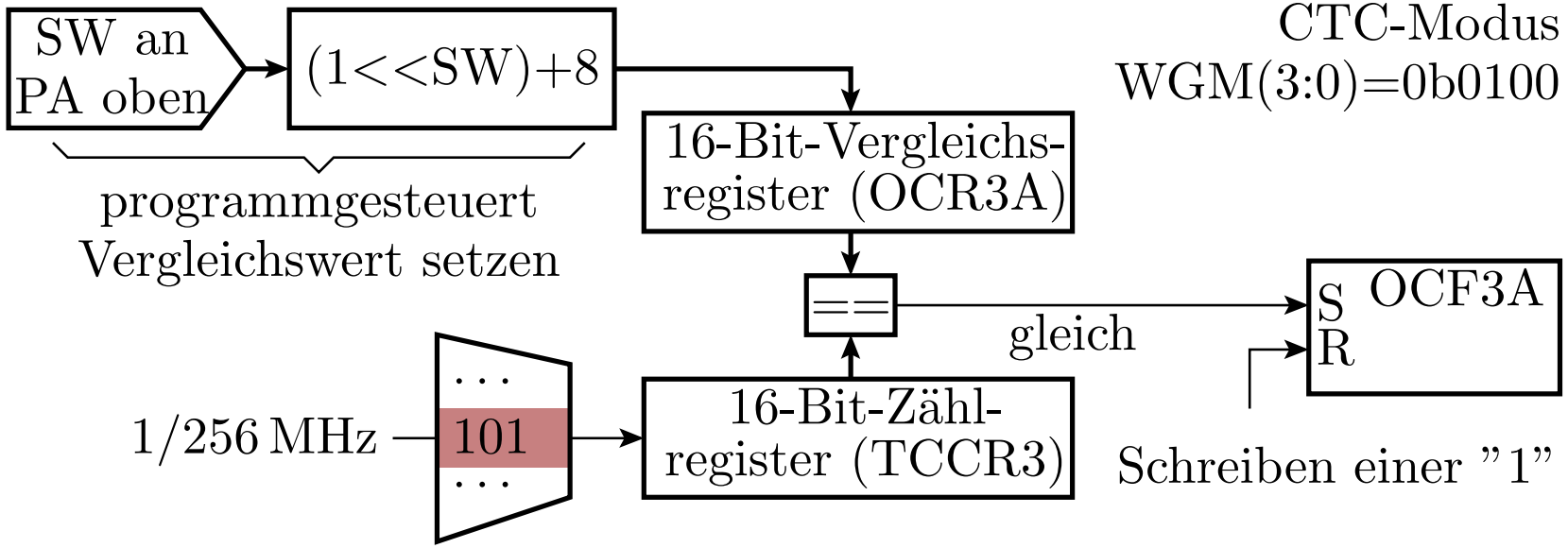
IO-View am Unterbrechungspunkt

TIMER_COUNTER_3

Name	Address	Value	Bits
TIFR3	0x38	0x0F	
OCF3A		0x01	
TOV3		0x01	
TIMSK3	0x71	0x00	
TCCR3A	0x90	0x00	
TCCR3B	0x91	0x03	
WGM3		0x00	
CS3		0x03	
TCNT3	0x94	0x0000	
OCR3A	0x98	0x0000	

»TOV3« gesetzt. Zähler »TCNT3« null, warum? »OCF3A« ist auch gesetzt, da »OCR3A==0« in jedem Zählzyklus erreicht wird und »OCF3A« nie gelöscht wird.

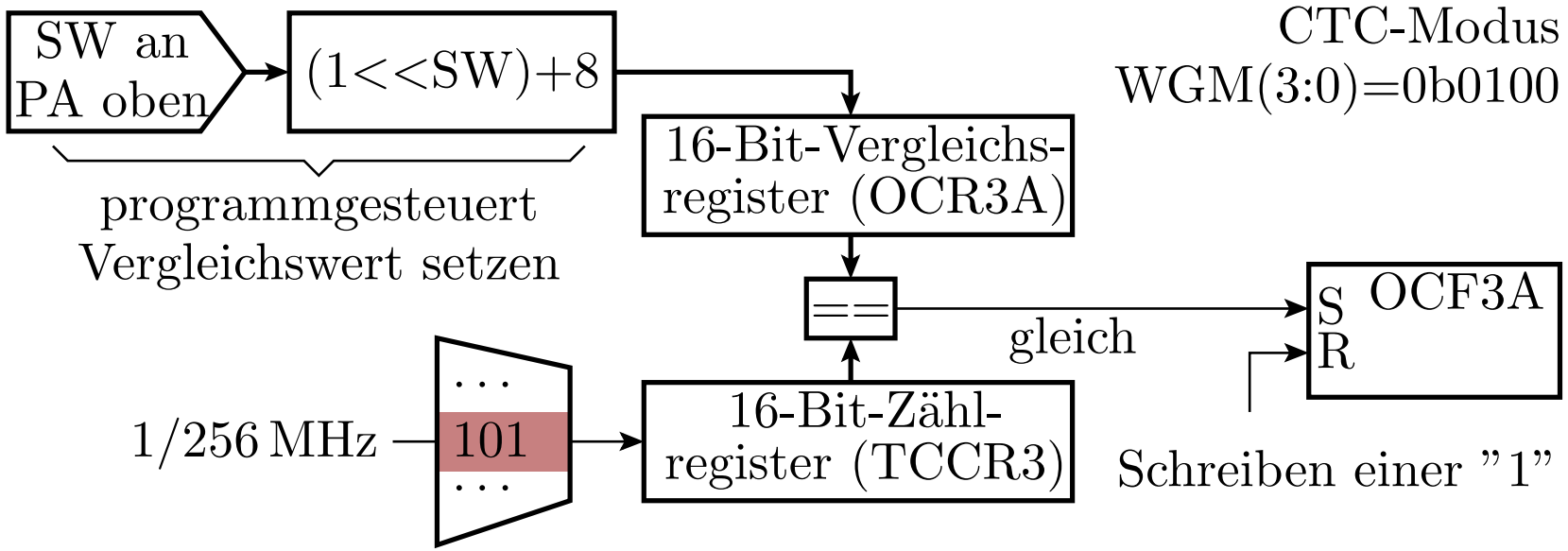
CTC-Modus und umschaltbare Zähltaktperiode



LED-Zähltakt:

$$f_{\mathrm{LED}} = \frac{f_{\mathrm{Proc}} = 8\,\mathrm{MHz}}{1024 \cdot w_{\mathrm{OCR3A}}}$$

sw	0000	0010	0100	1000	1001	1010	1011	1100	1101
$\frac{f_{\mathrm{LED}}}{\mathrm{Hz}}$	868	651	326	30	15	7,6	3,8	1,9	0,95



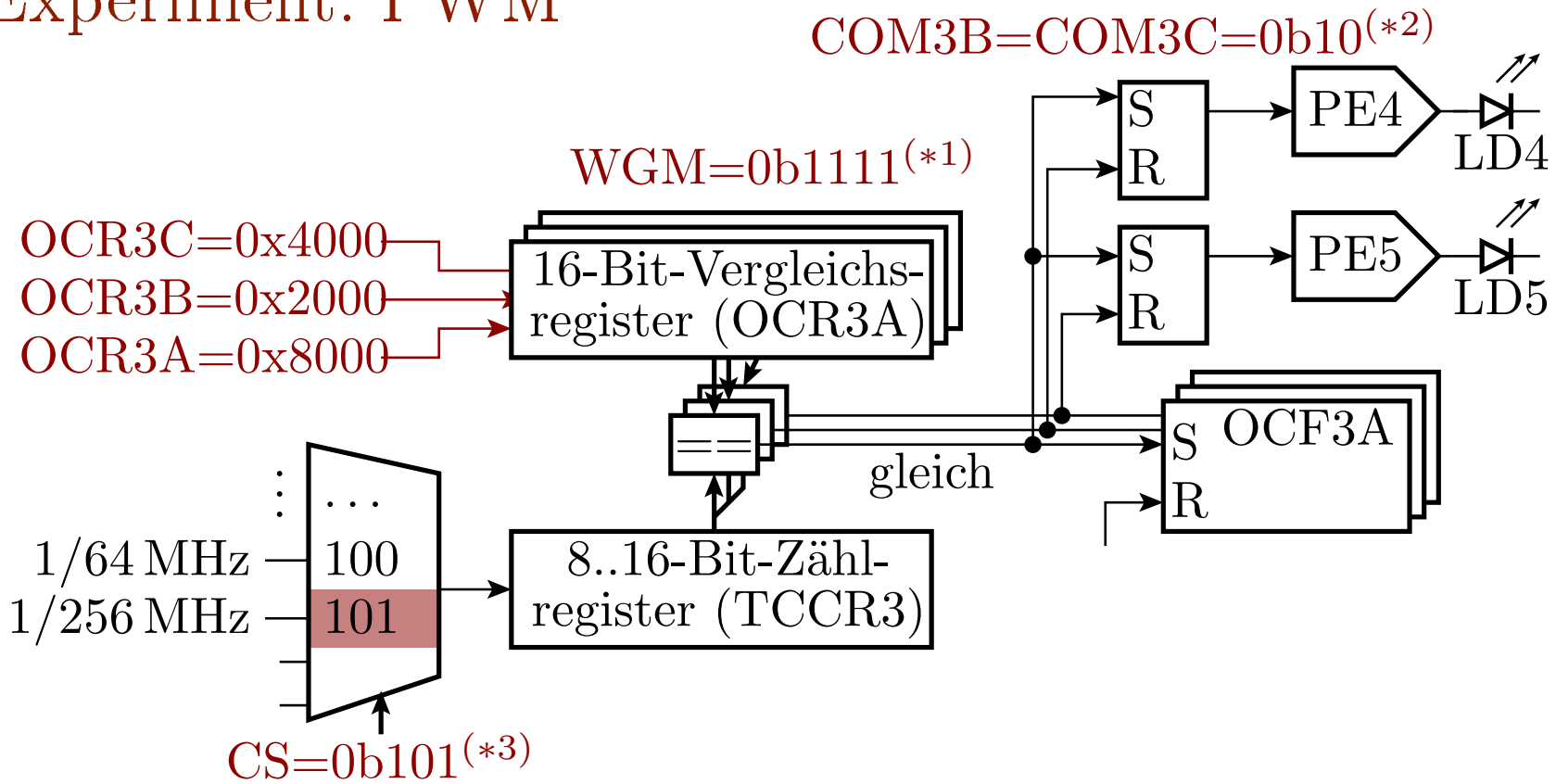
Testprogramm:

- Timer und LED-Ausgabe initialisieren.
- Wiederhole immer
  - Warte, bis Vergleichsbit »OCF3A« gesetzt.
  - LED-Ausgabe weiterzählern.
  - Vergleichsbit »OCF3A« löschen.
  - neuen Vergleichswert aus der Schaltereingabe bestimmen und in OCR3A schreiben.

```
#include <avr/io.h>
int main(void){          // Schaltermodul an JA oben
  TCCR3A = 0;            // WGM3[1:0] = 0
  TCCR3B = 0b1101;       // WGM3[3:2] = 1, CS3=0b101
  DDRJ = 0xFF;
  OCR3A = (1<<(PINA&0xF))+8; // Vergleichswert
  while(1){
    if (TIFR3 & (1<<OCF3A)){// Warte auf Gleichheit
      PORTJ++;                  // Erhöhe Led-Ausgabe
      TIFR3 = (1<<OCF3A);       // Lösche Vergleichsbit
      OCR3A = (1<<(PINA&0xF))+8;// neuer Vgl.-Wert
    }
  }
}
```

- Im Projekt F5-1_test_time\test_timer alle außer zweite Main-Funktion auskommentieren.
- Schaltermodul an Port A oben anstecken. SW(4:1)=1011.
- Übersetzen. Start ( , ). Kontrolle Zähltakt 1 Hz.
- Schalterwert erhöhen/verringern und Frequenz kontrollieren.

Experiment: PWM



- (*1) schnelle PWM mit OCR3A als Periodenregister.
- (*2) Zählperiode 256 µs. PWM-Periode: $256\,\mu s \cdot 0x8000 \approx 8{,}4\,s$
- (*3) PE4 und PE5: PWM-Ausgänge invertiert.
- LD4, LD5: LEDs auf PMOD8LD an JE

Testprogramm:

- Timer initialisieren.
- Endlosschleife, die nichts tun muss.

- LED-Modul »PMOD8LD« an JE stecken.
- Im Projekt F5-1_test_time\test_timer alle außer dritte Main-Funktion auskommentieren.
- Übersetzen. Start im Debugger  . Continue  .
- Kontrolle:
  - Blinkperiode ca. 8 s,
  - Einschaltzeit LED4 25% (ca. 2 s) und
  - Einschaltzeit LED5 50% (ca. 4 s).
- Anhalten  . Unterbrechungspunkt siehe nächste Folie setzen. Continue  bis  und Kontrolle der SFR-Werte.
- Ausprobieren mit anderen Haltepunkten, Pulsbreiten, CS-Werten, ..., ohne Anweisungen in der Endlosschleife.

```
#include <avr/io.h>
int main(void){
  // Aktivierung der PWM-Ausgänge
  TCCR3A = 0b10<<COM3B0 | 0b10<<COM3C0 | 0b11;
  TCCR3B = 0b11101;     // WGM3[3:2] = 0b11, CS3=0b101
  OCR3A  = 0x8000;      // Zählperiode
  OCR3B  = 0x2000;      // PE4 ein nach 25% Periode
  OCR3C  = 0x4000;      // PE5 ein nach 50% Periode
  DDRE = 0xFF;
  while(1){
    if (TIFR3 & (1<<OCF3A))
      TIFR3 = (1<<OCF3A);
    if (TIFR3 & (1<<OCF3B))
      TIFR3 = (1<<OCF3B);
    if (TIFR3 & (1<<OCF3C))
      TIFR3 = (1<<OCF3C);
  }
}
```

Die Anweisungen in der Endlosschleife dienen nur zum Setzen von Unterbrechungspunkten.

- Werte der Timer-Register am Haltepunkt:

TIMER_COUNTER_3

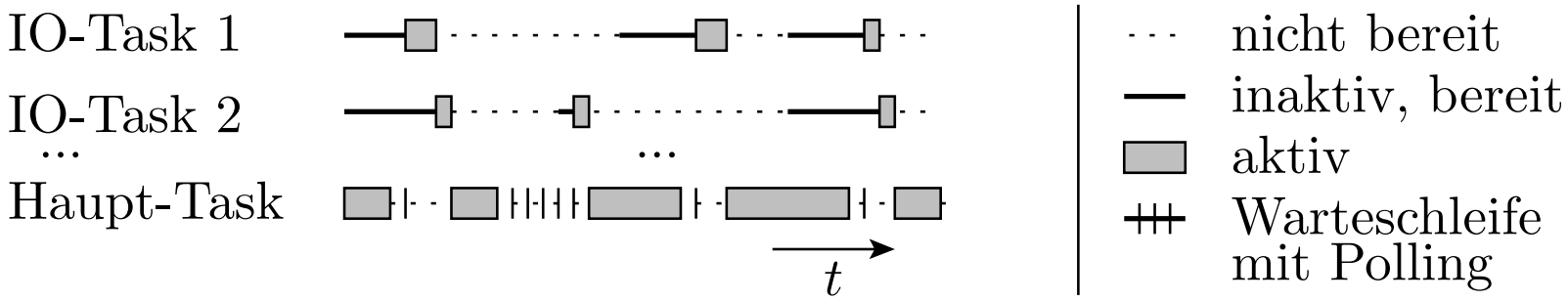
Name	Address	Value	Bits
TIFR3	0x38	0x03	
OCF3C		0x00	
OCF3B		0x00	
OCF3A		0x01	
TCCR3A	0x90	0x2B	
COM3A		0x00	
COM3B		0x02	
COM3C		0x02	
TCCR3B	0x91	0x1D	
TCNT3	0x94	0x0000	
OCR3A	0x98	0x8000	
OCR3B	0x9A	0x2000	
OCR3C	0x9C	0x4000	

- Verringern Sie den CS-Wert im Debugger am Unterbrechungspunkt auf CS=0b100. Unterbrechungspunkt löschen und Continue  . Wie ändert sich die Pulsperiode und die relative Pulsbreite?
- Schlagen Sie im Prozessordatenblatt nach, was mit COM3B und COM3C eingestellt wird. Programmänderung, so dass die LED-Ausgaben an PE4 und PE5 gegenüber dem Vorgabeprogramm invertiert werden.
- Die »OCR...« Werte lassen sich nicht im Debugger ändern, bzw. beim nächsten Debugger-Stopp steht wieder der alte Wert in den Registern. Workaround: Wertezuweisung aus einer Variablen in der Hauptschleife und Änderung der Variablenwerte im Debugger.
- Eine PWM mit einer Taktperiode im Millisekundenbereich wird später zur Steuerung der Motorgeschwindigkeit genutzt.

Interrupts

Task-Scheduling mit ISRs

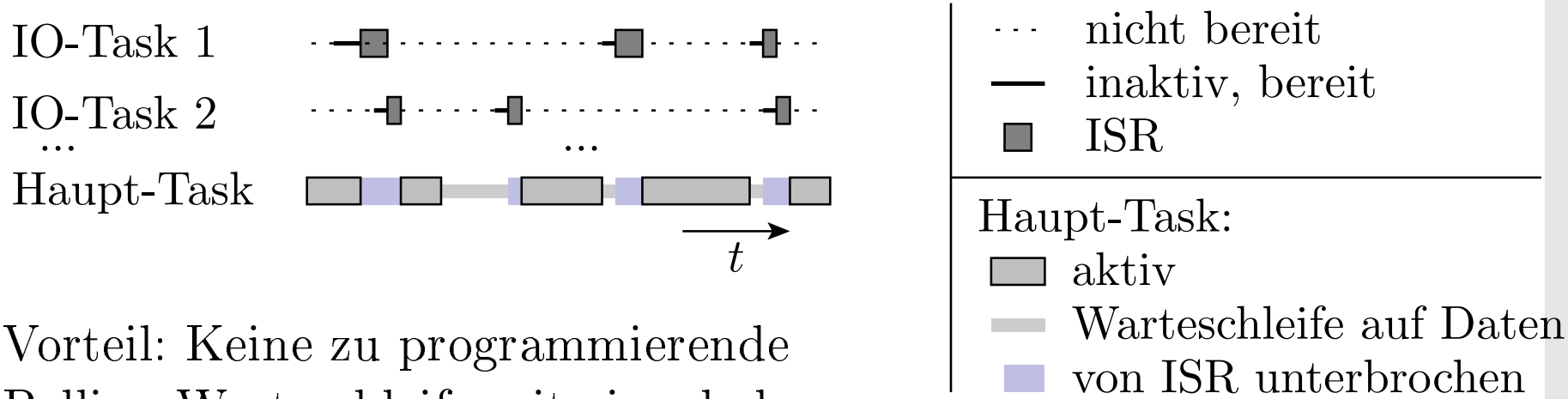
Task-Scheduling nach Foliensatz F4



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

Interrupts

- Beim Setzen eines Ereignisbits (UART, Timer, ...) Unterbrechung des aktuellen Verarbeitungsfluss und startet eine ISR (Interrupt-Service-Routine) von fester Adresse.
- Nach Abarbeitung der ISR Programmfortsetzung.



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

Nachteil: Programmunterbrechung an einer zufälligen Stellen.

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

- 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[2]:

```
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:

```
ETIMSK |=1<<OCIE3A;
```

- Interrupts global freigeben / sperren:

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

[2]Die Definition einer ISR kann an einer beliebigen Stelle in einer beliebigen c-Datei des Projekts erfolgen. Eine ISR ist keine normale Funktion, sondern eine, die mit goto <interrupt-Vektor> aufgerufen wird, keine Übergabeparameter und keinen Rückgabewert hat.

Tips und Tricks

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

```
ISR(BADISR_vect);
```

Zu Vereinfachung der Fehlersuche bei vergessenen ISR, falschem Interrupt-Vektor, falsche Interruptfreigabe eine »BAD-ISR« mit einer markannten 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 des Freigabebits wiederherstellen>
```

ISRs kurz halten:

- max. einige 100 abzuarbeitende Maschinenbefehle,
- keine Warteschleifen, möglichst nur Datenübernahme oder Ausgabe.
- Aufwändigere Verarbeitungen in die get- und put-Funktionen des Treibers verlagern, die der Haupt-Task aufruft.

Experimente

Experiment »F5-2_test_interrupt\test_interrupt«

Die nachfolgende Timer-ISR soll die Led-Ausgabe je Sekunde 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.

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

Port J wird mit 0 initialisiert und bei jedem 500sten ISR-Aufruf incrementiert. Die LEDs an Port C zählen die Neustarts:

```
PORTJ = 0;
PORTC++;
```

TIMER3_COMPA-Interrupt alle 1 ms:

```
TCCR3B = (1<<WGM32)|(0b001<<CS30); //Vorteiler 1
OCR3A  = 8000;       //1 ms Aufrufperiode
TIMSK3 = 1<<OCIE3A; //Timer3-Comp.A-Interrupt ein
```

TIMER1-OVF- (Überlauf) nach ca. alle 8 s:

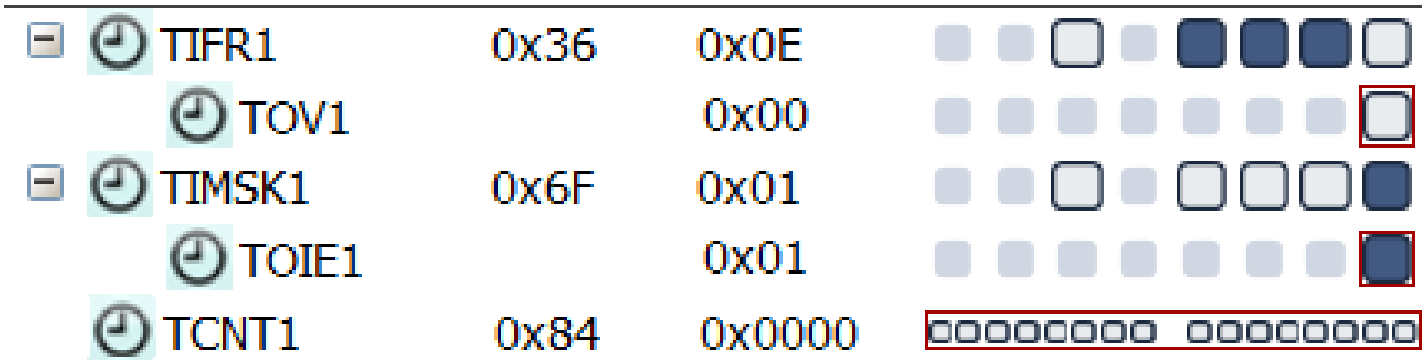
```
TCCR1B = 0b101;      //Nomalmodus, Vorteiler 1024
TIMSK1 = 0;          //Freigabe bleibt vorerst aus
sei();               //Interrupts global freigeben
while(1){            //in der Hauptschleife passiert
  //...              //vorerst nichts
}
}
```

Experiment 1: Test der TIMER3_COMPA-ISR

- Projekt »F5-2_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_TOV-Interrupt

- Anhalten. IO-View von Timer 1: Überlaufbit TOV1 löschen, Int.-Freigabe TOI1 setzen, Zähler CTNT1 löschen:



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

Experiment 3: Periodischer TIMER3_TOV-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 JC zählen jetzt ca. aller 8 s weiter und löscht dabei die LED-Ausgabe an PJ.
- 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-Ausgabe an Port C hochzaehlen
}
```

- Übersetzen. Start im Debugger . Continue .
- Die LEDs an JC zählen ca. aller 8 s weiter, aber der LED-Zählstand an PJ wird dabei nicht gelöscht.

Keine Neuinitialisierung mehr.

Experiment 5: ISR-Datenzugriff ohne ISR-Deaktivierung

- Debugger stoppen. Nur Invertierung von PJ.7 in der Endlosschleife einkommentieren:

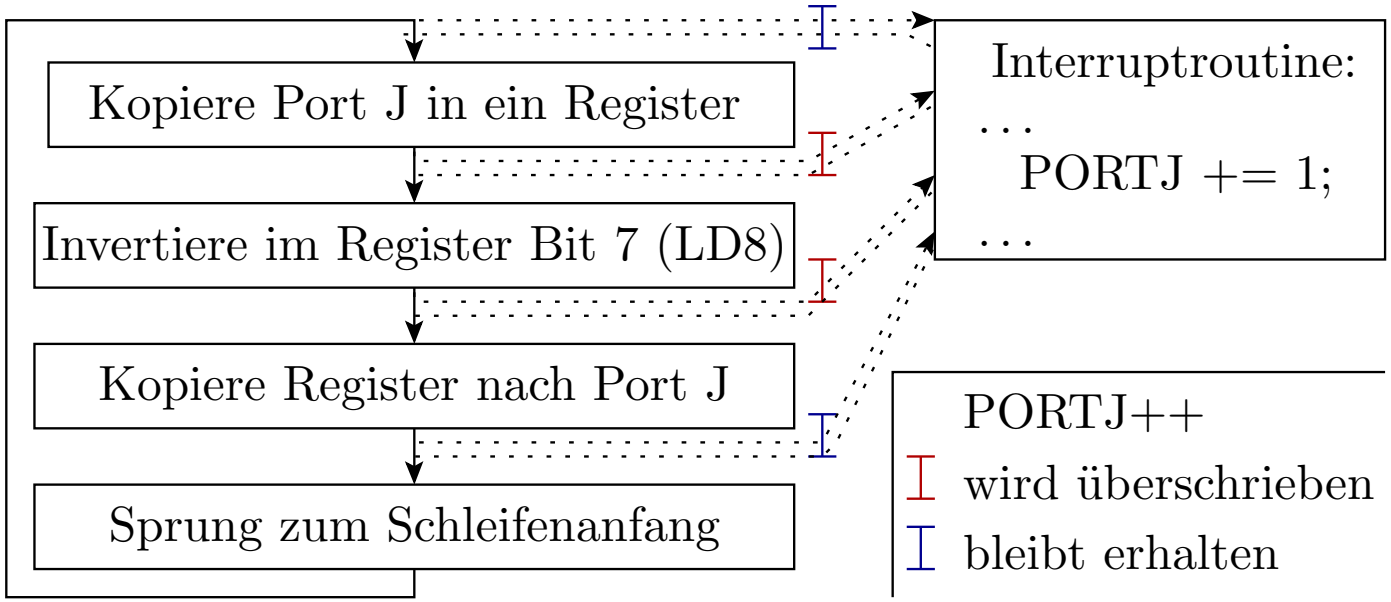
```
while(1){                //Hauptschleife
   PORTJ ^= 0x80;        //LD8 umschalten
  }
```

- 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 LD8 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 PJ.7 überschrieben.

Unterbrechungsfreie Sequenzen

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

- Interruptfreigabe speichern,
- Freigabebit löschen,
- nicht unterbrechbare Sequenz,
- Wiederherstellen der Interruptfreigabe.

Erweiterung der Anweisungsfolge in der Endlosschleife:

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

Nach Neuübersetzung und Neustart sind keine Zählanormalitäten mehr beobachtbar.

ISR-Treiber

Timer (comir_tmr)

Der Treiber »comir_tmr«

Das übergeordnete Programm der zu entwickelnden Fahrzeugsteuerung wird zeit- und ereignisgesteuerte Abläufe beschreiben, der Form:

- Fahre maximal 10 s geradeaus,
- wenn der Sensor 0,1 s einen Wert größer 0x25 erkennt, breche Geradeausfahrt ab, ...

Der Treiber »comir_tmr« nutzt Timer 1 und stellt über den Header comir_tmr.h folgende Funktionen bereit:

```
void tmr_init();      //Treiber initialisieren
uint32_t tmr_get();   //Zeitzähler lesen
```

Für vier unabhängige Timer-Kanäle:

```
void tmr_start(uint16_t tw, uint8_t nr);//Start
uint8_t tmr_restzeit(uint8_t nr);       //Restzeit lesen
```

(nr$\in \{0, 1, 2, 3\}$ – Timer-Kanal; Zeitwert in in 0,1 s-Schritten).

Private Daten und Initialisierungsfunktion

Private Daten:

```
uint32_t tmr_ct;        //Aufwärtszähler Programmzeit
uint16_t tmr_array[4]; //Abwärtszähler für 4 Kanäle
```

Initialisierung: Timer 1, CTC-Modus mit OCR1A als Vergleichregister, Zähltakt $\frac{1}{32}$ MHz:

```
#define WGM_CTC   0b0100   //Clear Timer on Compare
#define CS256     0b100    //Vorteiler 256
void tmr_init(){
  TCCR1A = WGM_CTC & 0b11; //Betriebsart & Zähltakt
  TCCR1B = (WGM_CTC & 0b1100)<<1 | (CS256 & 0b111);
  OCR1A  = 3125;           //Vergleichswert für 100 ms
  tmr_ct = 0;              //Laufzeitzähler löschen
  TIMSK1 |= 1<<OCIE1A;     //Vergleichs-Int. A ein
}
```

Ereignisperiode: $\frac{32 \cdot 3125}{1\,\mathrm{MHz}} = 0{,}1\,\mathrm{s}$

Interruptroutine

Die ISR incrementiert alle 100 ms den Programmzeitzähler und decrementiert die Wartezeitzähler der Kanäle, die nicht null sind:

```
ISR(TIMER1_COMPA_vect){      //Vergleichs-Interrupt
  uint8_t idx;               //alle 100 ms
  tmr_ct++;                  //Programmzeit zählen
  for (idx=0; idx<4; idx++) //für alle 4 Wartezähler
    if (tmr_array[idx])      //wenn ungleich null
      tmr_array[idx]--;      //abwärts zählen
}
```

Zugriffsmethoden für das Anwenderprogramm

```
uint32_t tmr_get(){                    //Zeitzähler lesen
  return tmr_ct;
}
```

Für vier unabhängige Timer-Kanäle:

```
void tmr_start(uint16_t tw, uint8_t nr){//Start
  tmr_array[nr & 0b11] = tw;           //Wartezeit schreiben
}

uint8_t tmr_restzeit(uint8_t nr){ //Lesen der
  return tmr_array[nr & 0b11];     //Restzeit
}
```

Was ist an den Zugriffsmethoden falsch programmiert?

Werden möglicherweise Daten bearbeitet, die die ISR verwendet?

- Zugriff mit unterbrechungsfreien Sequenzen:

```
void tmr_start(uint16_t tw, uint8_t nr){
  uint8_t  tmp = TIMSK1;     //Int.-Zustand sichern
  TIMSK1 &= ~(1<<OCIE1A);    //Vergleichs-Interrupt A aus
  tmr_array[nr & 0b11] = tw;//Wartezeit schreiben
  TIMSK1 = tmp;              //Int.-Zustand wiederherst.
}
uint8_t tmr_restzeit(uint8_t nr){ //Lese Restzeit
  uint8_t  tmp = TIMSK1;     //Int.-Zustand sichern
  TIMSK1 &= ~(1<<OCIE1A);    //Vergleichs-Interrupt A aus
  uint16_t z = tmr_array[nr & 0b11];//Restzeit lesen
  TIMSK1 = tmp;              //Int.-Zustand wiederherst.
  return z;
}
uint32_t tmr_get(){          //Zeitzähler lesen
  uint8_t  tmp = TIMSK1;     //Int.-Zustand sichern
  TIMSK1 &= ~(1<<OCIE1A);    //Vergleichs-Interrupt A aus
  uint16_t z = tmr_ct;       //Uhrzeit zurückgeben
  TIMSK1 = tmp;              //Int.-Zustand wiederherst.
  return z;
}
```

Testbeispiel für der Treiber

- Timer-Kanal 0 soll die LED an PJ.7 aller 0,7 s und Timer-Kanal 1 soll die LED an PJ.6 aller 1,2 s invertieren.
- Auf den LEDs an PJ[4:0] soll der Zeitwert in Sekunden ausgegeben werden.

- Projekt »F5-3_test_comir_tmr\test_comir_tmr« öffnen.
- Übersetzen. Start im Debugger  . Continue  .
- LED-Ausgaben kontrollieren.

Das Testprogramm:

```
int main(){
  tmr_init();          //Timer-Treiber initialisieren
  DDRJ = 0xFF;         //Port J LED-Ausgabe
  sei();               //Interrupts global ein
  while(1){            //Beginn Endlosschleife
```

- Ablauf in der Endlosschleife:

```
if (!tmr_restzeit(0)){ //wenn Kanal 0 abgelaufen
 PORTJ ^=0x40;         //LD6 invertieren
 tmr_start(12, 0);     //Kanal 0 mit 1,2 s init.
}
if (!tmr_restzeit(1)){ //wenn Kanal 1 abgelaufen
 PORTJ ^=0x80;         //LD7 invertieren
 tmr_start(07, 1);     //Kanal 1 mit 0,7 s init.
}
                       //Zeit in s auf LD[4:0] ausgeben
uint8_t tmp = (tmr_get()/10) & 0x1F;
PORTJ = (PORTJ & ~0x1F) | tmp;
 }
}
```

Anregungen zum Experimentieren:

- Die anderen Timer-Kanäle mitnutzen.
- Komplexere Blinksequenzen erzeugen.
- Schalter und LED-Module mit einbeziehen, ...

LCD (comir_lcd)

Der Treiber »comir_lcd«

Umstellung des Treibers »comsf_lcd« auf Interrupts:

- Gleichfalls Sendeeinheit von UART1 (JD).
- Bei Initialisierung Sendepuffer-frei-Interrupt-Freigabe.
- ISR statt Schrittfunktion.
- BADISR mit dem letzten Anzeigezeichen als Fehlerzähler.

Gleiche Textbearbeitungfunktionen für die LCD-Ausgabe:

```
//Fehlerzähler erhöhen
void lcd_incErr(uint8_t pos);

//Einzelzeichenausgabe
void lcd_disp_chr(uint8_t c, uint8_t pos);

//Ausgabe von »len« Zeichen
void lcd_disp_str(uint8_t *str, uint8_t pos, uint8_t len);
```

```
// Ausgabe eines Zahlenwertes
void lcd_disp_val(uint32_t val, uint8_t pos, uint8_t len);
```

(pos – Schreibposition; len – Anz. der zu schreibenden Zeichen).

Unterbrechungssperren nicht erforderlich. Wenn das Schreiben auf den Puffer unterbrochen wird, erscheint im ungünstigsten Fall für wenige ms ein halb geänderter Text auf dem LCD.

Die Funktionen zur Textmanipulation können deshalb unverändert vom Treiber »comsf_lcd« übernommen werden. Beschreibung auf Foliensatz F4.

Private Daten und Initialisierung

Die privaten Daten sind dieselben wie beim Treiber comsf_lcd:

```
uint8_t LCD_dat[32];  //Ausgabetext
uint8_t lcd_idx;      //Indexvariable
```

Die Initialisierungsfunktion aktiviert nur zusätzlich den Sendepuffer-frei-Interrupt:

```
void lcd_init(uint8_t *text){ //LCD-Treiber init.
  ... //genau wie in comsf UART1 initialisieren
  ... //LC-Display initialisieren
  ... //Initialisierung des Hintergrundtextes
  UCSR1B |= (1<<UDRIE1); //Puffer-frei-Interrupt ein
}
```

Die Interruptroutine und die BADISR

ISR: Zirkulares versenden des nächsten Zeichens, sobald der Sendepuffer frei ist:

```
ISR(USART1_UDRE_vect){              //Puffer-frei ISR
  UDR1 = LCD_dat[lcd_idx];          //schicke nächstes
  lcd_idx++;                        //Zeichen
  //nach dem letzten folgt das erste Zeichen
  if (lcd_idx>=32) lcd_idx = 0;
}
```

BADISR: Erhöhung des letzten Anzeigezeichens (unten rechts) als Fehlerzähler bei jedem ungewollten Interrupt:

```
ISR(BADISR_vect){                   //Fehlerzähler (letztes
  lcd_incErr(31);                   //Zeichen) hochzählen
}
```

Testbeispiel

Definition des Anzeigeformats für das nachfolgende Testbeispiel:

```
// Ausgabetext
#define INITSTR "W0:.. x W1:.. x Zeit:....s E:..."

// Zeichenpositionen für Ausgaben
#define LCP_W0T   3 // Restzeit Timer-Kanal 0
#define LCP_W0Z   6 // Zustand Timer-Kanal 0
#define LCP_W1T  11 // Restzeit Timer-Kanal 1
#define LCP_W1Z  14 // Zustand Timer-Kanal 1
#define LCP_ZEIT 21 // Zeit seit Programmstart in s

// Fehlerzähler BAD_ISR hat LCD-Position 31
```

Initialisierungsteil des Testbeispiels

```
int main(void){
  uint8_t z0='0', z1='0'; //Ausgabezustand
  tmr_init();             //Treiber initialisieren
  lcd_init((uint8_t*)INITSTR);

  //nicht behandelter Interrupt ca. alle 8 s
  //für den Test des Fehlerzählers
  TCCR3B = 0b101;      //Timer 3, Normalmodus, Vorteiler
  TIMSK3 = 1<<TOIE3;   //1024, Überlaufinterrupt ein

  sei();               //Interrupts global ein
```

Endlosschleife zum Testbeispiel

```
while(1){
  if (!tmr_restzeit(0)){ //wenn Kanal 0 abgelaufen
    tmr_start(31, 0);    //Kanal 0 mit 3,1 s init.
    lcd_disp_chr(z0, LCP_W0Z);
    z0 ^=1;              //'0' (0x30) <=> '1' (0x31)
  }
  if (!tmr_restzeit(1)){ //wenn Kanal 1 abgelaufen
    tmr_start(17, 1);    //Kanal 1 mit 1,7 s init.
    lcd_disp_chr(z1, LCP_W1Z);
    z1 ^=1;              //'0' (0x30) <=> '1' (0x31)
  }
  //Zeitwerte immer aktualisieren
  lcd_disp_val(tmr_restzeit(0),LCP_W0T, 2);
  lcd_disp_val(tmr_restzeit(1),LCP_W1T, 2);
  lcd_disp_val(tmr_get()/10, LCP_ZEIT, 4);
 }
}
```

- LCD-Modul mit Doppelkabel an JD (Kabel A oben) anstecken. JDX »gekreuzt (=)«.
- LCD-Anschluss und LCD-Jumper-Stellungen siehe Bild.
- Projekt »F5-4_test_comir_tmr\test_comir_tmr« öffnen.
- Übersetzen. Start im Debugger  . Continue  .
- Ausgabe kontrollieren.

PC-Kommunikation (comir_pc)

Änderungen gegenüber dem Treiber »comsf_pc«

- Ersatz der Schrittfunktion für Senden und Empfangen durch je eine ISR für Empfang und Senden.
- Zusätzliche Empfangs-Timeout-ISR mit Timer 3. Löscht im Empfangspuffer Bytes halb empfangener Nachrichten nach 0,1 s. Fehlertoleranz gegenüber Fehlansteuerungen vom PC.
- Drei Interrupt-Freigaben am Ende der Initialisierung.
- Interrupt-Sperren für Sequenzen zum Lesen und Schreiben von IO-Daten in der Send- und Get-Funktion.
- Zusätzlicher Zähler für Sende- und Empfangsfehler:

```
uint8_t com_pc_err_ct; //private globale Variable
```

Zählbefehle nutzbar für Unterbrechungspunkte.

Fehlerzähler-Programmierschnittstelle der Treiber-API :

```
uint8_t com_pc_err();
```

  - Kein Fehler, Rückgabewert eins,
  - sonst Rückgabewert null und Fehlerzähler löschen.

Private Daten und Initialisierung

```
uint8_t rmsg[COM_PC_RMSG_LEN];  //Empfangspuffer
uint8_t smsg[COM_PC_SMSG_LEN];  //Sendepuffer
uint8_t sidx, ridx, last_byte;  //Pufferzeiger, letztes ...
uint8_t com_pc_err_ct;          //Fehlerzähler
```

Die Puffergrößen sind im Header definiert, in diesem Projekt 2× vier. Für andere Projekte im Header anzupassen.

Erweiterung der Initialisierungsfunktion:

```
void com_pc_init(){
  ... //Initialisierung und Einschalten UART2
  ... //Empfangs- und Sendepuffer leer
  UCSR2B |= (1<<RXCIE2);     //Empfangs-Interrupt ein
  TCNT3  = 0;                //Zähler 3 Rücksetzen
  TCCR3B = 0;                //Zähltakt aus
  OCR3A  = 12500;            //Empfangs-Timeout 100 ms
  TIMSK3|= 1<<OCIE3A;        //Vergleichs-Interrupt A ein
}
```

ISR für Empfang und Empfangs-Timeout

```
ISR(USART2_RX_vect){                    // ISR Empfang
  last_byte = UDR2;                     // Byte lesen
  if (ridx < COM_PC_RMSG_LEN){          // wenn Platz
    rmsg[ridx] = last_byte;             // in Puffer übernehmen
    ridx++;
    TCNT3  = 0;                         // Rücksetzen Zähler 3
    TCCR3B = 0b11;                      // Zähltakt clk/64 ein
  }
}

ISR(TIMER3_COMPA_vect){                 // ISR Empfangs-Timeout
      // wenn der Empfangspuffer noch nicht gefüllt ist
  if ((ridx>0) && (ridx <COM_PC_RMSG_LEN)){
    ridx = 0;                           // Puffer löschen
    com_pc_err_ct++;                    // Fehlerzähler erhöhen
  }
  TCCR3B = 0;                           // Zähltakt aus
}
```

ISR für das Senden

```
ISR(USART2_UDRE_vect){                  //Sendepuffer-frei-ISR
  if (sidx < COM_PC_SMSG_LEN){          //wenn Sendepuffer frei
    UDR2 = smsg[sidx];                  //und Senddaten, diese
    sidx++;                             //versenden und Index
  }                                     //erhöhen
  else
    UCSR2B &= ~(1<<UDRIE2);             //Puffer-frei-Int. aus
}
```

Wenn keine zu versendenden Daten mehr anstehen, MUSS der Puffer-Frei-Interrupt deaktiviert werden. Sonst wird nach jedem Maschinenbefehl des Hauptprogramm die ISR eingeschoben. Erhebliche Reduzierung der nutzbaren Verarbeitungsleistung.

Get-Funktion

Unterbrechungsfreies Kopieren der gesamten Nachricht aus dem Empfangspuffer in einen Nachrichtenpuffer des Programms.

Vom Test, ob Daten da sind, bis Abschluss der Datenübergabe, sind Empfangs-Interrupts gesperrt.

```
uint8_t com_pc_get(uint8_t *msg){
  uint8_t tmp = UCSR2B;    // Int.-Freigabe speichern
  UCSR2B &= ~(1<<RXCIE2); // Empfangs-Interrupt aus
  if (ridx < COM_PC_RMSG_LEN){// wenn der Empf. nicht voll
    UCSR2B = tmp;          // Int.-Freigabe wiederherstell.
    return 0;              // voll, Rücksp. mit "falsch"
  }
  for (ridx=0; ridx<COM_PC_RMSG_LEN;ridx++) // sonst Empf.-
    msg[ridx] = rmsg[ridx]; // nachricht kopieren
  ridx = 0;                // Empfangspuffer leeren
  UCSR2B = tmp;            // Int.-Freigabe wiederherstell.
  return 1;                // Rücksprung mit "wahr"
}
```

Send-Funktion

Unterbrechungsfrei gesamte Nachricht aus dem Nachrichtenpuffer des Programms in den Sendepuffer des Treibers kopieren.

```
uint8_t com_pc_send(uint8_t *msg){
  uint8_t tmp = UCSR2B;          //Int.-Freigabe speichern
  UCSR2B &= ~(1<<UDRIE2);        //Puffer-frei-Interrupt aus
  if (sidx < COM_PC_SMSG_LEN){//wenn der S-Puf. nicht voll
    UCSR2B = tmp;                //Int.-Freigabe wiederherst.
    return 0;                    //leer, Rückgabe "falsch"
  }
  for (sidx=0; sidx<COM_PC_SMSG_LEN;sidx++)
    smsg[sidx] = msg[sidx];      //sonst Nachricht übergeben
//und Zeiger auf Nachrichtenanfang
  sidx = 0;                      //Sendepuffer voll
  UCSR2B |= 1<<UDRIE2;           //Interrupt-Freigabe ein
  return 1;                      //Rücksprung mit "wahr"
}
```

Interrupt-Sperre analog Get-Funktion.

Letztes Byte und Fehlerbehandlung

```
uint8_t com_pc_last_byte(){   //Rückgabe letztes Byte
  return last_byte;
}
```

Gedacht für Sofort-Nachrichten wie »Fahrzeughalt«, wenn im Empfangspuffer noch unabgearbeitete Steuernachrichten stehen.

```
uint8_t com_pc_err(){      //Fehlerabfrage
  if (com_pc_err_ct){      //wenn Fehler aufgetreten waren
    com_pc_err_ct = 0;     //Fehlerzähler löschen
    return 1;              //Rückkehr mit 1 (wahr)
  }                        //sonst
  return 0;                //Rückkehr mit 0 (falsch)
}
```

In beiden Funktionen keine Unterbrechungssperre, weil die kritischen Sequenzen genau ein Byte kopieren, d.h. je nur aus einem nicht unterbrechbaren Maschinenbefehl bestehen.

Ultraschallsensor (comir_sonar)

Der Treiber »comir_sonar«

Bereitstellung Sonar-Abstandswerte. Gegenüber »comsf_sonar«

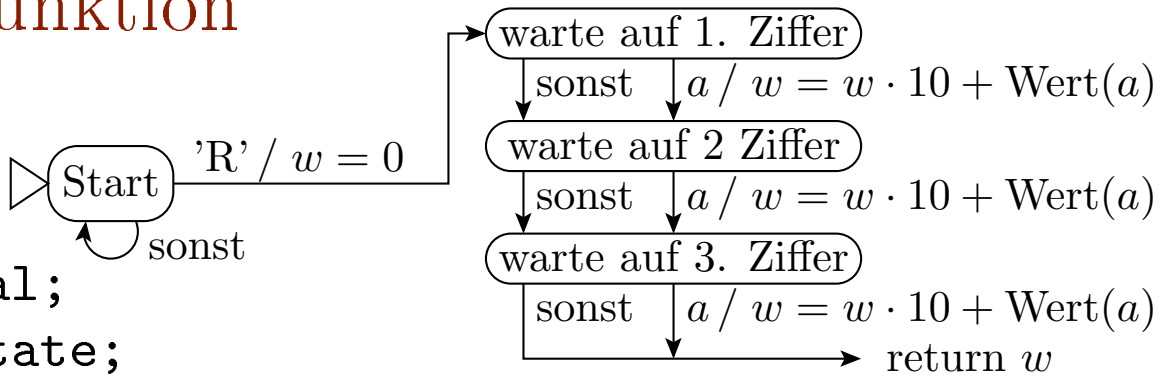
- schaltet die Init-Funktion den Empfangs-Interrupt frei.
- ISR statt Schrittfunktion.
- Get-Funktion mit nicht unterbrechbarer kritischer Sequenz.

Erweiterung der Init-Funktion:

```
void sonar_init(){
  ... //USART0 Initialisieren 8N1, 9600 Baud, ...
  PORTE |= 1<<PE1;              //Sonar einschalten
  snr_state =0;                 //Empfangsautomat init.
  UCSR0B |= (1<<RXCIE0);        //Empfangsinterrupt ein
}
```

Achtung: Der Sendeteil von UART0 wird von dem Teiber für das LCD genutzt. Beim Einschalten von Empfänger und Empfangs-Interrupts nicht Sender und Sende-Interrupt ausschalten.

Aus der Schrittfunktion wird die ISR




```
uint16_t snr_val;
uint8_t  snr_state;
ISR(USART0_RX_vect){ //Empfangs-ISR
  int8_t dat = UDR0; //Zeichen lesen
                     //Empfangsautomat weiterschalten
  if (dat=='R'){     //wenn 'R', Zustand
    snr_state = 1;   //"warte 1. Hexziffer'
    snr_val   = 0;   //Wert mit 0 initialisieren
    return;
  }  //wenn »warte auf Ziffer« Wert * 10 + Ziffernwerk
  if (snr_state>0 && snr_state<4){
    if (dat>='0' && dat<='9')
      snr_val = (snr_val*10) + (dat-'0');
    snr_state++;
  }
}
```

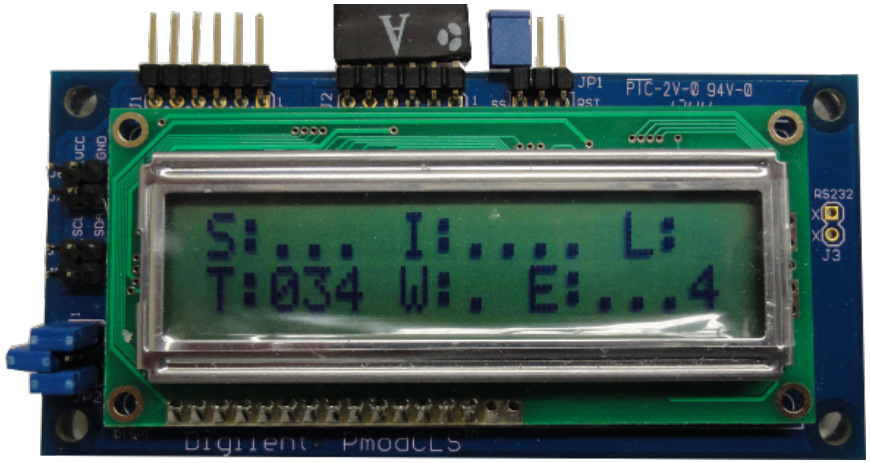
Get-Funktion

Die Get-Funktion muss während der Auswertung und Änderung des Automatenzustands und dem Lesen des Abstandswertes die Interrupt-Annahme verbieten:

```
uint8_t sonar_get(uint16_t *sptr){
  uint8_t tmp = UCSR0B;  // Int.-Freigabe speichern
  UCSR0B &= ~(1<<RXCIE0);// Empfangs-Interrupt aus
  if (snr_state>=4) {    // wenn neuer Wert da,
    *sptr = snr_val;     // ausgeben
    snr_state = 0;       // Zustand rücksetzen
    UCSR0B = tmp;        // Int.-Freigabe wiederherstellen
    return 1;            // Rücksprung mit "wahr"
  }
    UCSR0B = tmp;        // Int.-Freigabe wiederherstellen
    return 0;            // Rücksprung mit "falsch"
}
```

Testbeispiel mit allen Treibern

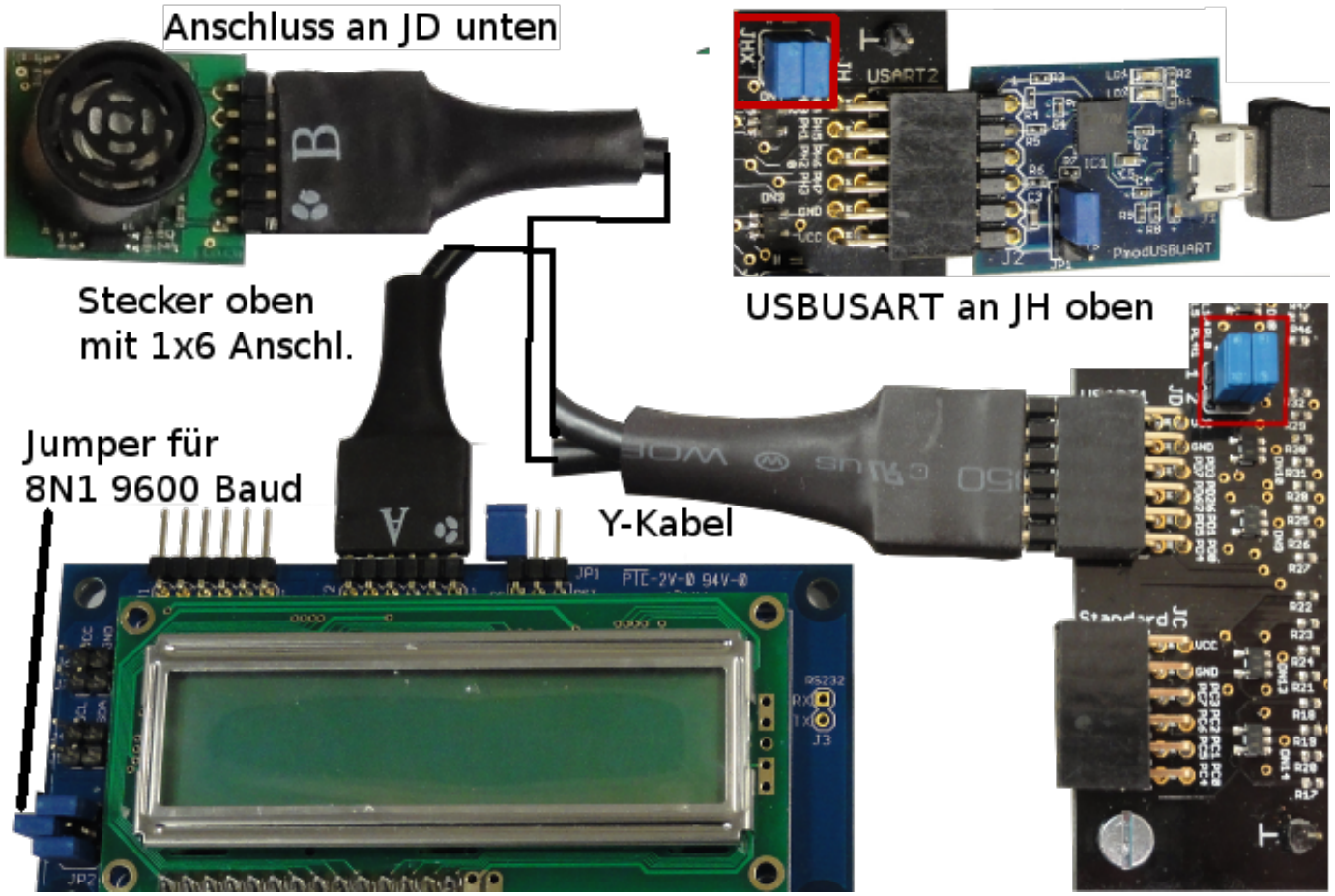
Testbeispiel mit allen Treibern



Ausgabe nach Programmstart:

S: Sonar-Abstandswert (Treiber »comir_sonar«)
I: 4-Zeichen-Nachricht vom PC (Treiber »comir_pc«)
L: letztes vom PC empfangenes Byte (Treiber »comir_pc«)
T: Zeit seit Programmstart in s (Treiber »comir_timer)
W: Wartezeit bis zur Ausgabe (»comir_timer).
E: Fehlerzähler (Treiber »comir_lcd«)

- Programm wartet auf eine mindests 4 Bytes lange Nachricht.
- Wartet dann 8 s mit Count-Down-Anzeige.
- Schickt dann den Sonar- und zwei Zählwerte zurück.



- »PmodMAXSONAR« und »PModPTH2« wie im Bild über Y-Kabel an JD (LCD oben, Sonar unten).
- PModUSBUSART an JH oben und über USBl zum PC.
- Jumper JHK und JDX »gekreuzt (=)«.

- Projekt »F5-5_comir\comir« öffnen, übersetzen und Programm starten.
- HTerm starten, 8N1 9600 Baud, Connect.
- Zeichenfolge »asdf« schicken.
- Kontrolle Nachrichtanzeige LCD.

- Weitere Einzelzeichen schicken und Kontrolle »letztes empfangenes Byte«.
- Countdown von 8 auf 0 s und HTerm-Zeichenempfang abwarten. Kontrolle des im HTerm empfangenen 2-Byte-Sonarwerts (in Zoll), des 1-Byte-Sonar-Zählerwerts und des 1-Byte Nachrichtenzählers auf Plausibilität.

Fehlerzähler:

1. Versendefehler (nicht künstlich erzeugt).
2. Empfangs-Timout: Increment bei Nachrichenlänge $\neq 4$
3. Testzähler: Increment bei jedem Nachrichtenempfang.
4. BADIR-Zähler: Increment ca. alle 8 s (Tmr4-Überlauf).

Programmbeschreibung

Konstanten für die Anzeige:

```
#define INITSTR "S:... I:.... L:.T:... W:. E:...."
#define LCP_SONAR      2 //Sonarwert
#define LCP_RMSG       8 //Eingabedaten
#define LCP_LBYTE     15 //letztes empf. Byte
#define LCP_TIME      18 //Zeitanzeige
#define LCP_WAIT      24 //Wartezeit
//Anzeigepositionen für Fehlerzähler
#define ERR_SEND      28 //Sendeversagen
#define ERR_ETO       29 //Empfangs-Timout
#define ERR_TEST      30 //zählt Empfangsnachrichten
//Zeichen 31 ist der Zähler falsche Interrupts
```

Initialisierung

```
int main(void){
  uint8_t state='E';     //Programmzustand E∈{E,V,A}
  uint16_t snrval;       //Sonarwert
  uint8_t sn_ct, msg_ct;//Sonarwert und Nachrichtenz.
  sonar_init();          //alle Hintergrundprozesse
  com_pc_init();         //initialisieren
  lcd_init((uint8_t*)INITSTR);
  tmr_init();
  //nicht behandelter Interrupt ca. alle 8 s
  TCCR4B = 0b101;        //Timer 4, Nomalmodus, Vorteiler
  TIMSK4 = 1<<TOIE4;     //1024, Überlaufs-Interrupt ein
  sei();                 //Interrupts freigeben
  while(1) {
  ... //
  }
}
```

```
if (sonar_get(&snrval)){   //wenn neue Sonardaten
   //Sonarwert auf LCD ausgeben
   lcd_disp_val(snrval, LCP_SONAR, 3);
    sn_ct++;                //Sonarwerte zählen
 }
if (state=='E'){           //wenn Programmzust. "Eingabe"
  if (com_pc_get(mrmsg)){  //Wenn neue PC-Nachricht
    //diese übernehmen, auf LCD ausgeben
    lcd_disp_str(mrmsg, LCP_RMSG, COM_PC_RMSG_LEN);
    lcd_incErr(LCP_TESTERR); //Testfehlerzäehler erhöhen
    msg_ct++;
    tmr_start(80, 0);      //Tmr-Kanal 0 für 8s starten
    state = 'V';           //Programmzust. "Verarbeitung"
  }
}
else if (state=='V'){
  //Ausgabe Wartezeit bis zur nächsten EA-Operation
  lcd_disp_val(tmr_restzeit(0)/10, LCP_WAIT, 1);
    if (!tmr_restzeit(0))
      state = 'A';         //Programmzustand "Ausgabe"
 }
```

```
else if (state=='V'){
  //Ausgabe Wartezeit bis zur nächsten EA-Operation
  lcd_disp_val(tmr_restzeit(0)/10, LCP_WAIT, 1);
  if (!tmr_restzeit(0))
    state = 'A';              //Programmzustand "Ausgabe"
}
else{                         //wenn Programmz. "Ausgabe"
  msmsg[0] = snrval >> 8;     //Sensor- und Zählwerte
  msmsg[1] = snrval & 0xFF;//byteweise in die Send-
  msmsg[2] = sn_ct;           //nachricht schreiben
  msmsg[3] = msg_ct;
  if (!com_pc_send(msmsg)) //"string" versenden, wenn
    lcd_incErr(ERR_SEND);  //erfolglos, Fehlerz. ++
  state = 'E';                //Programmzustand "Eingabe"
}
```

```
    //immer letztes empf. Byte auf LCD schreiben
    lcd_disp_chr(com_pc_last_byte(), LCP_LBYTE);
    //immer Zeit seit Programmstart in s ausgeben
    lcd_disp_val((tmr_get()/10) % 1000, LCP_TIME, 3);
    if (com_pc_err())              //Wenn Empfangs-Timeout
      lcd_incErr(ERR_ETO);         //Fehlerzähler erhöhen
  //Ende der Hauptschleife
```

Kontrollfragen zum Testprogramm:

- Beschreiben Sie den Zustandsablauf des Testprogramms durch einen Automatengraphen.
- Was passiert, wenn eine zweite 4-Byte-Nachricht gesendet wird, bevor der 8-s-Timeout abgelaufen ist?
- Wird die Timout-Ausgabe im Programmzustand 'A' (Ausgabe) gelöscht?
- Wie könnte man zum Testen einen Sendefehler provozieren?

Aufgabe 5.1: Timer-Experimente

- Die Warteschleife im Projekt »F1-3_bit_io3\bit_io3« durch einen Timer ersetzen.
- Im PWM-Modus Periodendauer auf 1 ms verringern und Signalverlauf mit USB-Logi sichtbar machen.

Aufgabe 5.2: Test Timer-Treiber

- Anderen Timer-Kanäle mitnutzen.
- Komplexere Blinksequenzen erzeugen.
- Schalter und LED-Module mit einbeziehen, ...
- Mit dem Tmr-Treiber periodische Ausgabe einer Morse-Folge, z.B. »kurz-kurz-lang-kurz-lang-lang-kurz-...-Pause«.
- Anderes Testbeispiel mit den Treibern, z.B. wenn der Hindernisabstand zum Sonar-Sensor kleiner 8 inch ist, soll LD1 mit 1 Hz blinken.

Aufgabe 5.3: Interrupt-Experimente

Im Projekt F3-1_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 5.4: Test mit Logikanalysator

Visualisierung der Zeitverläufe im Testprogramm »test_comier« (ab Folie 78) mit dem USB-LOGI.

- Anschluss Ch0 bis Ch7 des USB-Logi an JA (PA.0 bis PA.7).
- Testprogramm und Treiber so ändern, dass jede ISR und jedes Unterprogramm einer Nummer ≥ 2 ausgibt und in »main« für den Aufzeichnungsbeginn das aufzuzeichnende Byte von 0b0000 0000 nach 0b0000 0001 wechselt.
- Aufzeichnungstaktperiode ca. 1 µs (5 bis 10 Maschinenbefehle).

Aufgabe 5.5: Benutzung der Comir-Treiber

Die besprochenen Treiber können Sie in ihrem eigenen Projekt nutzen.

- Denken Sie sich ein Steuerprogramm für ihr Fahrzeug aus. Stellen Sie die geplanten Motoraktivitäten auf dem LC-Display dar. Simulieren Sie externe Ereignisse (Weg abgefahren, Hindernis erkannt, ...) mit Timern.
- Entwickeln Sie in Anlehnung an »comir_pc« einen Treiber für die Kommunikation über Bluetooth mit dem PC (oder ihrem Handy).

Es werden weitere Treiber besprochen:

- Foliensatz F6: Motorensteuerung und Wegemessung.
- Foliensatz F7: IR-Abstandssensor und Joystick.