Informatikwerkstatt, Foliensatz 6
Timer

G. Kemnitz

Institut für Informatik, TU Clausthal (IW6)

15. November 2016

Inhalt:

Interaktive Übungen:

1. Normalmodus (F6-test_timer/test_timer).
2. CTC-Modus (F6-test_timer/test_timer).
3. PWM (F6-test_timer/test_timer).

Wiederholung

Wiederholungsaufgabe 6.1: Treiber

1. Wozu dienen die Initialisierungsfunktionen?
2. Was tun 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 Lese- und Schreibfunktionen blockierungsfrei arbeiten?

Lösung

1. Initialisierungsfunktionen konfigurieren die zugeordnete Hardware, legen private Daten an und initialisieren diese.
2. Schrittfunktionen testen auf EA-Ereignisse und führen für eingetretene EA-Ereignisse die dann erforderlichen Funktionen aus. Bei uns werden sie immer am Anfang der Endlosschleife aufgerufen.
3. Die Send- und Get-Funktionen dienen in den Programmteilen, die die Treiber nutzen, zur Datenübergabe und Übernahme.
4. Blockierungsfrei bedeutet hier, dass die Funktion, wenn der Treiber noch nicht zur Datenübernahme bzw. Übergabe bereit ist, nicht wartet, sondern mit dem Status »Datenaustausch noch nicht möglich« zurückkehrt.
5. Blockierungsfrei, damit der Haupttask und mehrere EA-Tasks nebenläufig arbeiten können, d.h. dass jeder Task abgearbeitet wird, sobald er bereit und der Rechner frei ist.

Wiederholungsaufgabe 6.2: Sendefunktion

```
#define SENDBUF_FREE (UCSR2A & (1<<UDRE2))
void sendByte(uint8_t dat){
  while (!SENDBUF_FREE);
  UDR2 = dat;
}
```

1. Warum wird diese Funktion als blockierend bezeichnet?
2. Die Funktion auf der nächsten Folie soll, wenn der Sendepuffer frei ist, das übergebene Byte versenden und sich mit Rückgabewert 1 und sonst ohne Versenden mit Rückgabewert 0 beenden.

Bitte vervollständigen:

```
#define SENDBUF_FREE (UCSR2A & (1<<UDRE2))

uint8_t sendByte_nb(uint8_t dat){




}
```

Lösung

1. Die Funktion ist blockierend, weil sie wartet, bis der Sendepuffer frei ist.
2. Nicht blockierende Sendefunktion:

```
#define SENDBUF_FREE (UCSR2A & (1<<UDRE2))

uint8_t sendByte_nb(uint8_t dat){
 if (SENDBUF_FREE){
  UDR2 = dat;
  return 1;
 }
 else {
  return 0;
 }
}
```

Timer

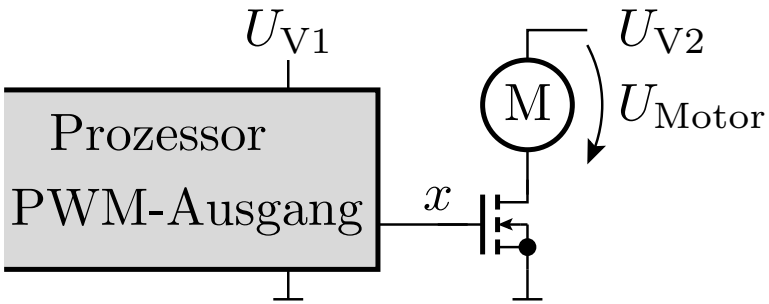
Timer

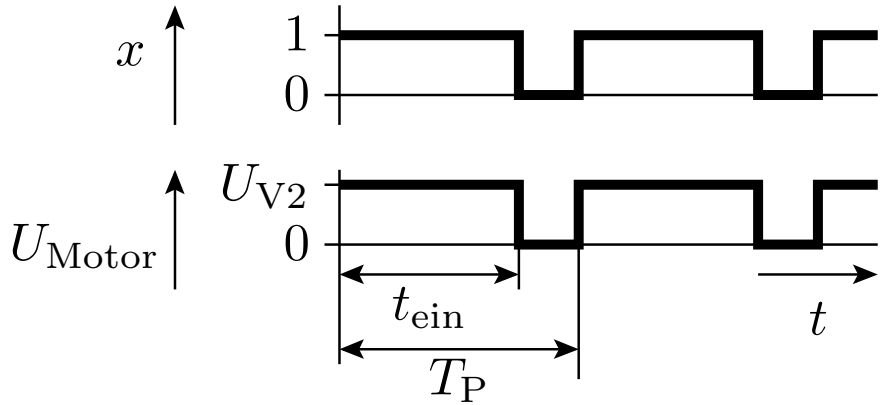
Ein Timer ist eine Hardware-Einheit aus Zähl-, Vergleichs-, Konfigurationsregistern, ... zur

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

PWM-Signale dienen

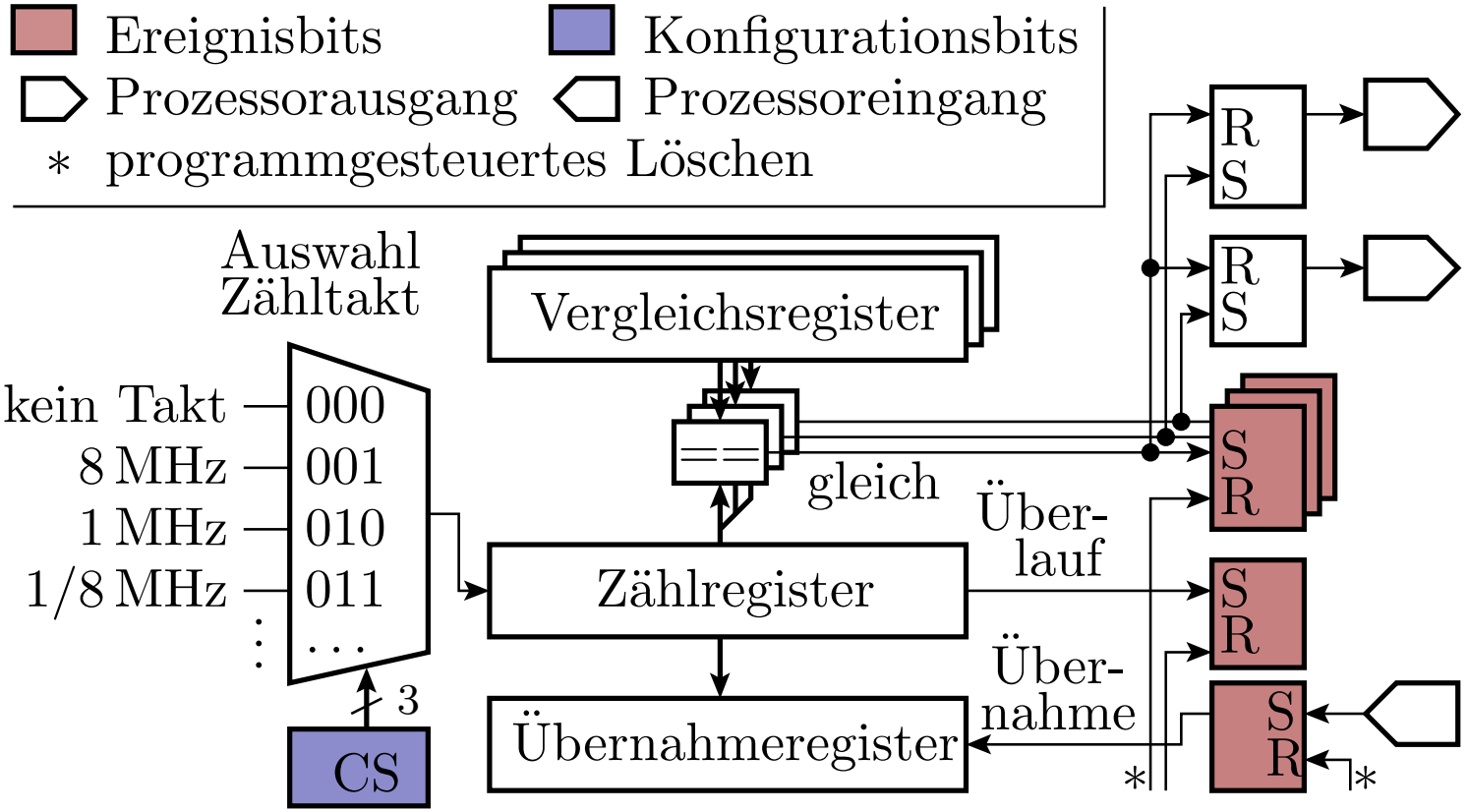
- zur Informationsübertragung z.B. an Modellbauservos und
- zur stufenlosen Leistungssteuerung, z.B. unserer Motoren.





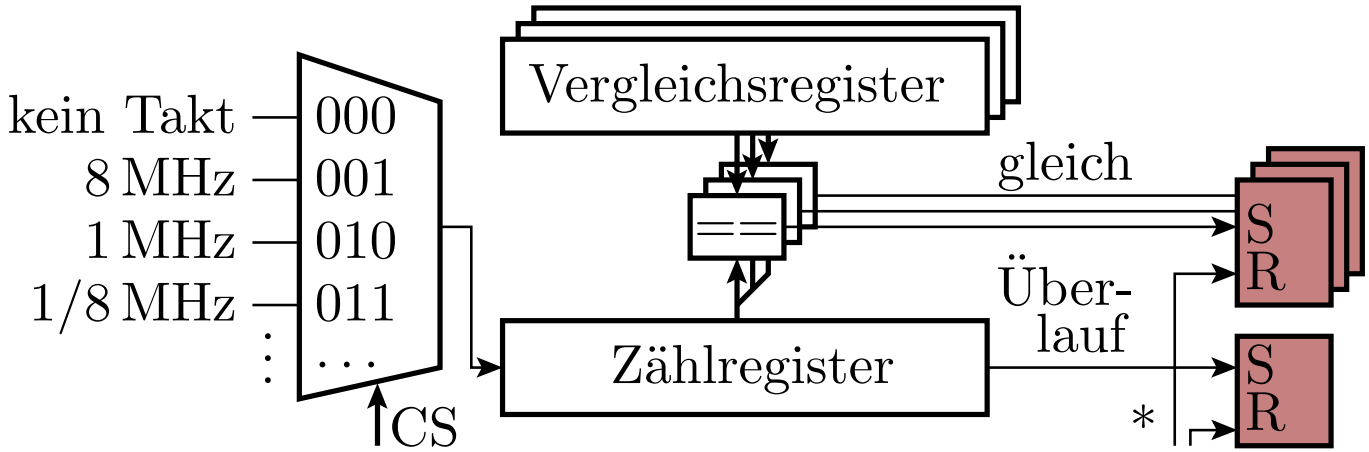
Funktionsweise

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.

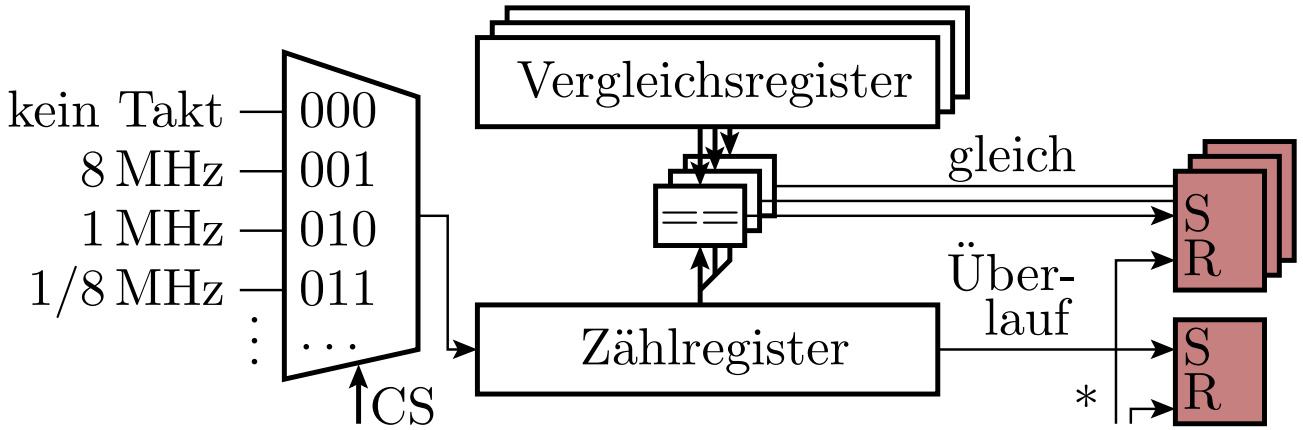
Normalmodus



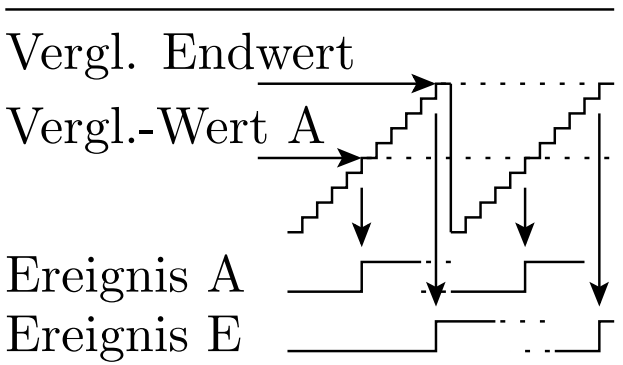
- Zählregister zählt zyklisch bis zum Überlauf.
- Beim Überlauf wird ein Überlaufbit und bei Gleichheit mit einem Vergleichsregister ein Vergleichsereignisbit gesetzt.
- Beispiel Wartefunktion:

```
void wait(uint32_t tw){
 <berechne und setze Takt und Vergleichswert>
 <Lösche Zähler und Vergleichsereignisbit>
 <warte bis Vergleichsereignisbit==1>
 <schalte Zähltakt aus> }
```

CTC- (Clear on Compare) Modus

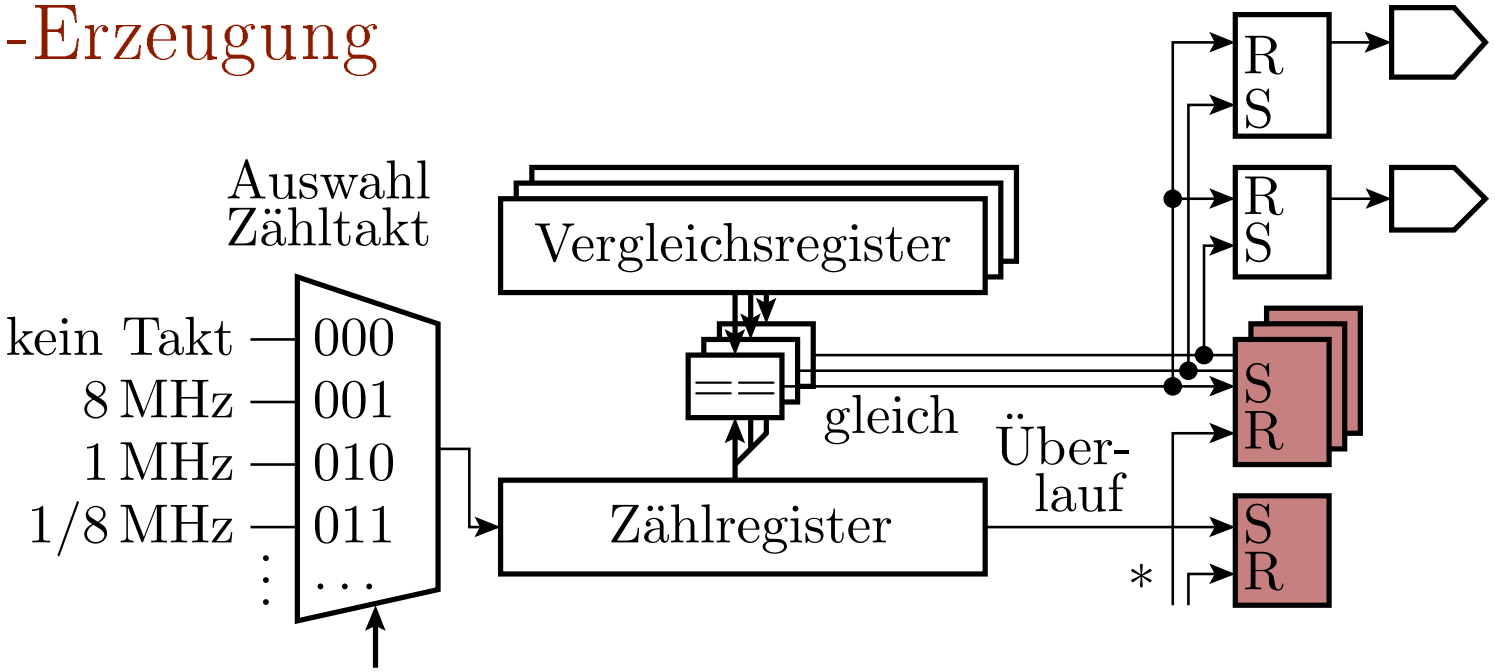


- Zähler wird bei Gleichheit mit einem der Vergleichsregister rückgesetzt.
- Auslösung zyklischer Ereignisse, z.B. Uhrenprozess:

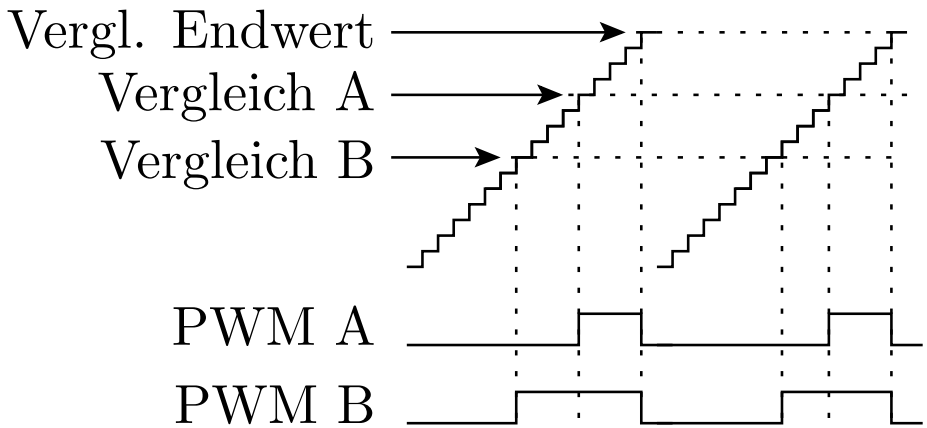



```
void Schrittfunktion Uhr(){
  if (<Vergleichs-Rücksetz-Ereignis>)
   <lösche Ereignisbit, schalte Uhr weiter>
}
```

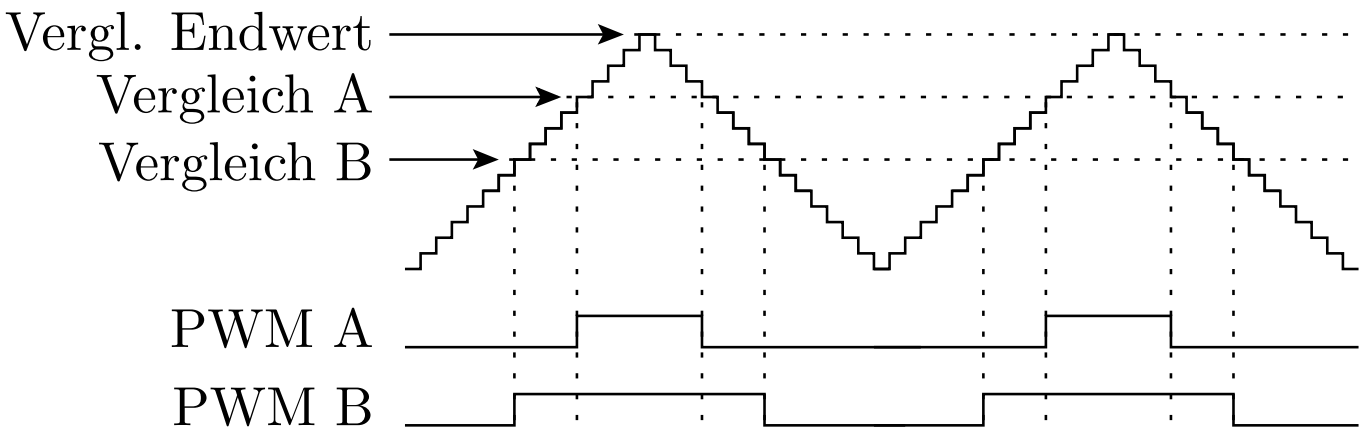
PWM-Erzeugung



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



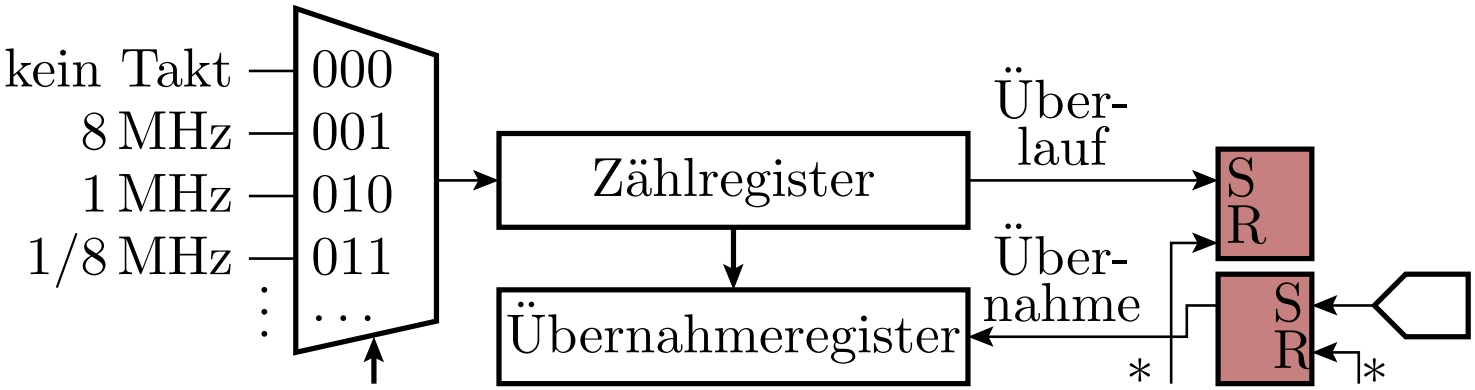
Symmetrische PWM[1]



- 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 invertierte 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.

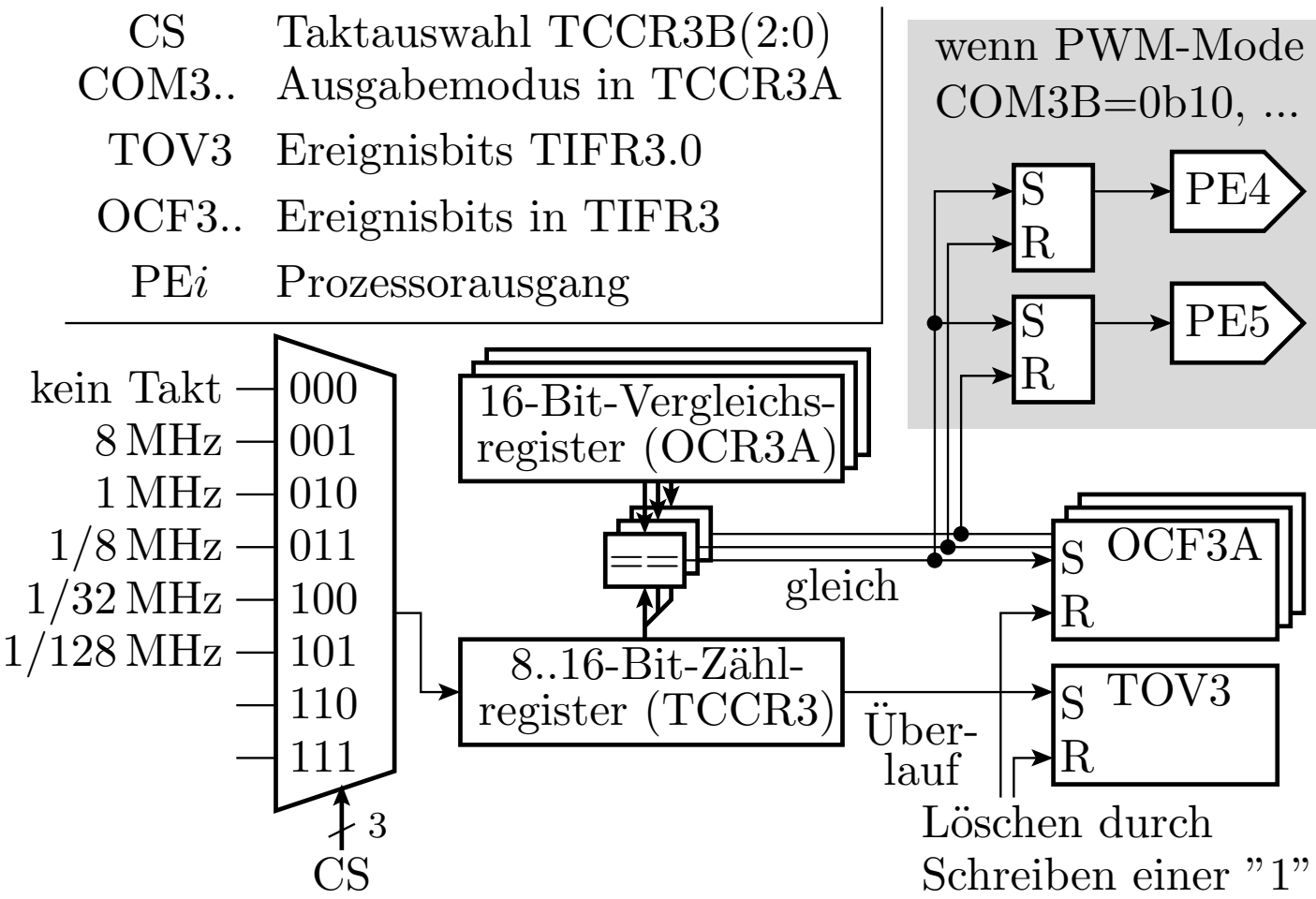
Nutzung der Timer in den Beispielprojekten:

- 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 ungenutzten Timer 2 und 4 sind noch frei für andere Aufgaben, z.B. als Timeout-Zähler für den Bluetooth-Empfang.

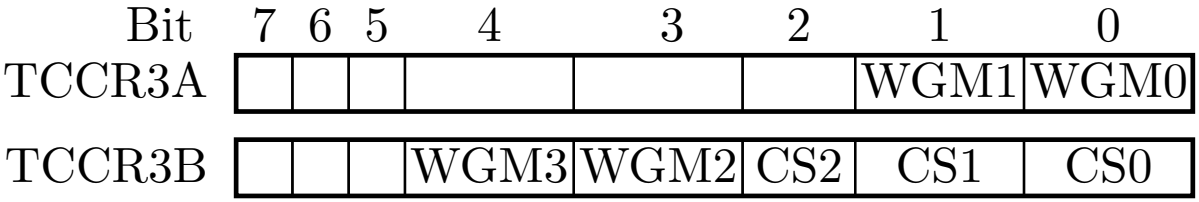
Timer 3

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

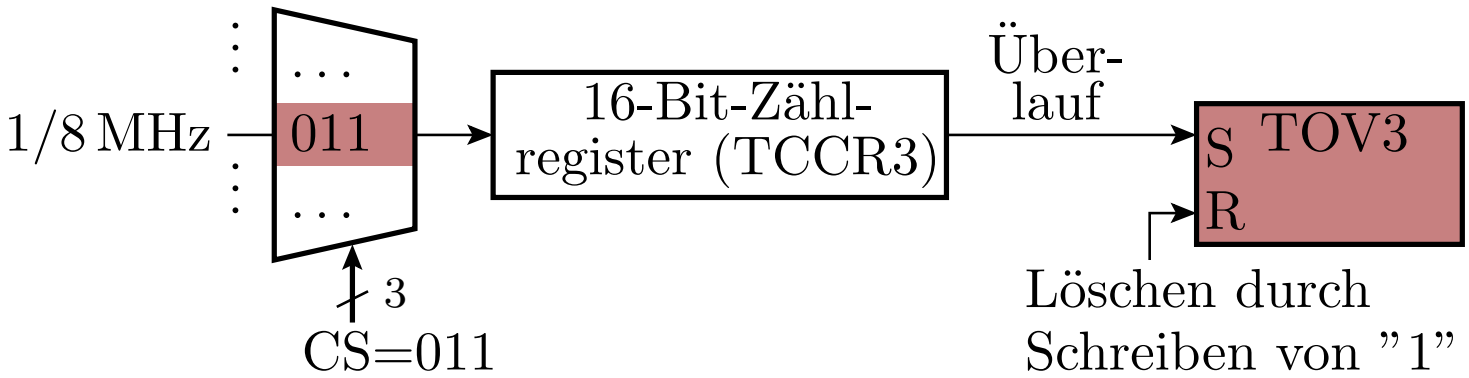
$^{(*1)}$ symmetrische oder phasenausgerichtete PWM.

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

Experiment: Normalmodus

LED mit Timer hochzählen

- Timer im Normalmodus (WGM(3:0)=0) und CS=011:



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

$$f_{\mathrm{LED}} = \frac{\frac{1}{8}\,\mathrm{MHz}}{2^{16}} = 1{,}9\,\mathrm{Hz}$$

```
#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 F6-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

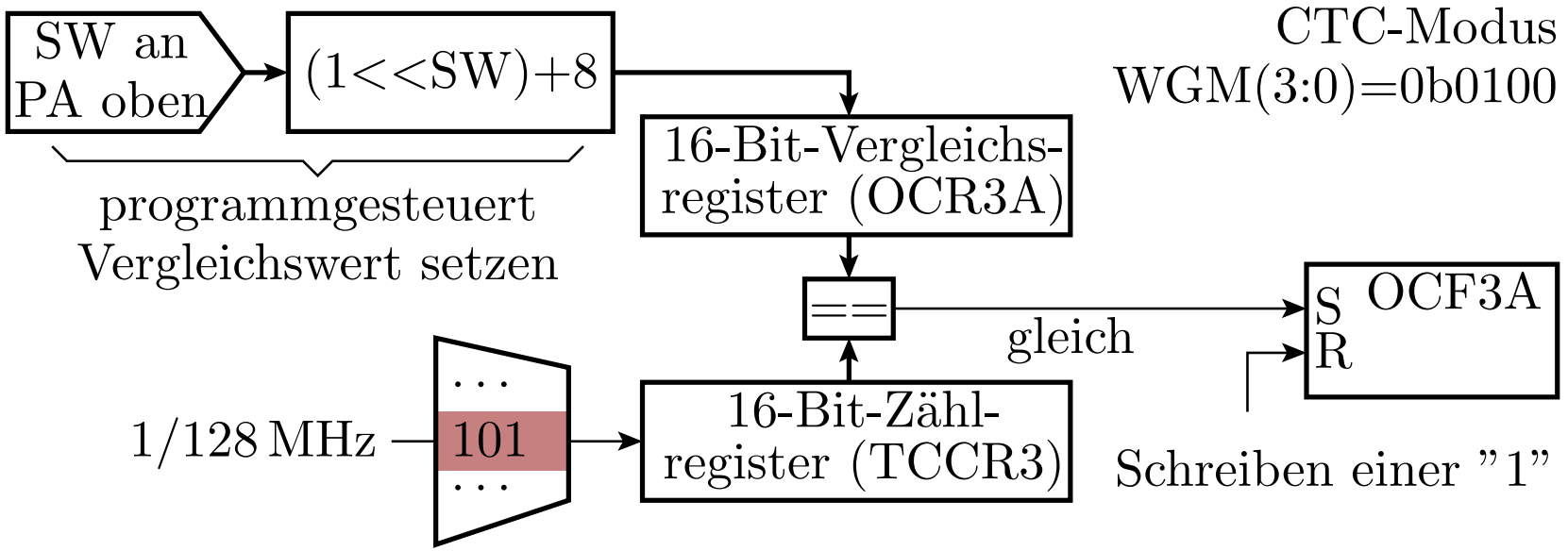
TIMER_COUNTER_3

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.

Experiment: CTC-Modus

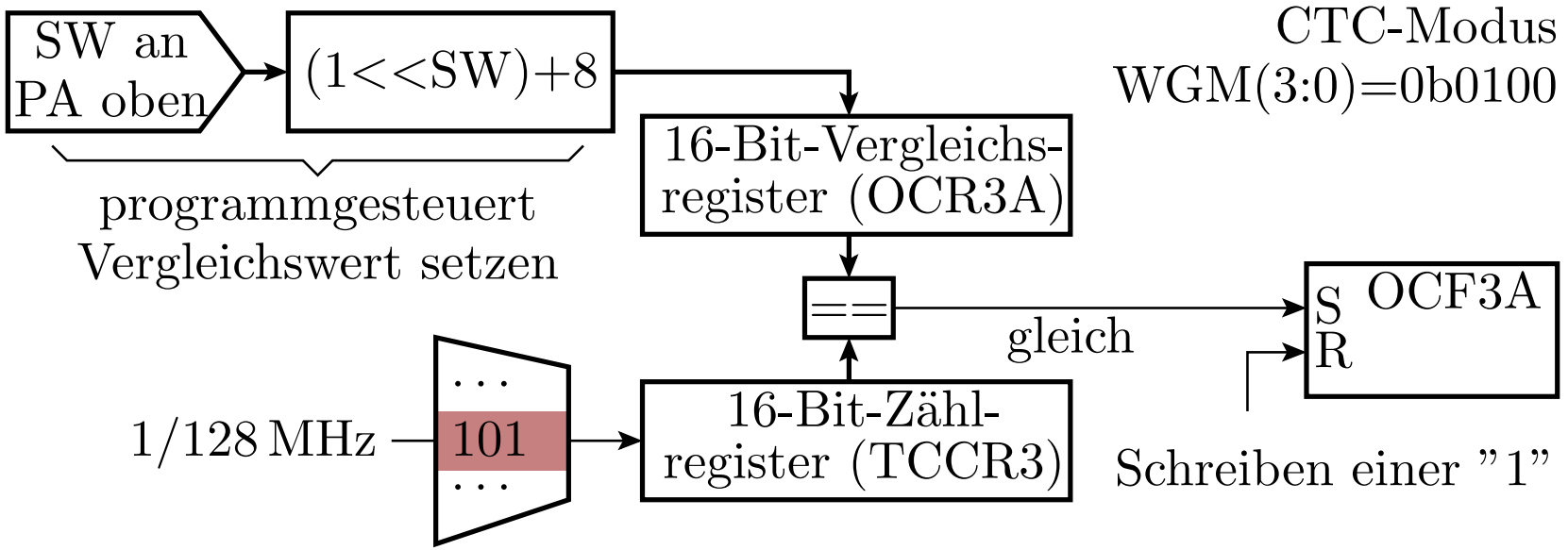
CTC-Modus und umschaltbare Zähltaktperiode



LED-Zähltakt:

$$f_{\text{LED}} = \frac{\frac{1}{128}\,\text{MHz}}{\text{OCR3A}} \text{ mit OCR3A=8+(1 << } sw\text{)}$$

sw	0000	0010	0100	1000	1001	1010	1011	1100
OCR3A	1+8	4+8	16+8	264	520	1034	2056+	4104
$\frac{f_{\text{LED}}}{\text{Hz}}$	868	651	326	30	15	7,6	3,8	1,9



Testprogramm:

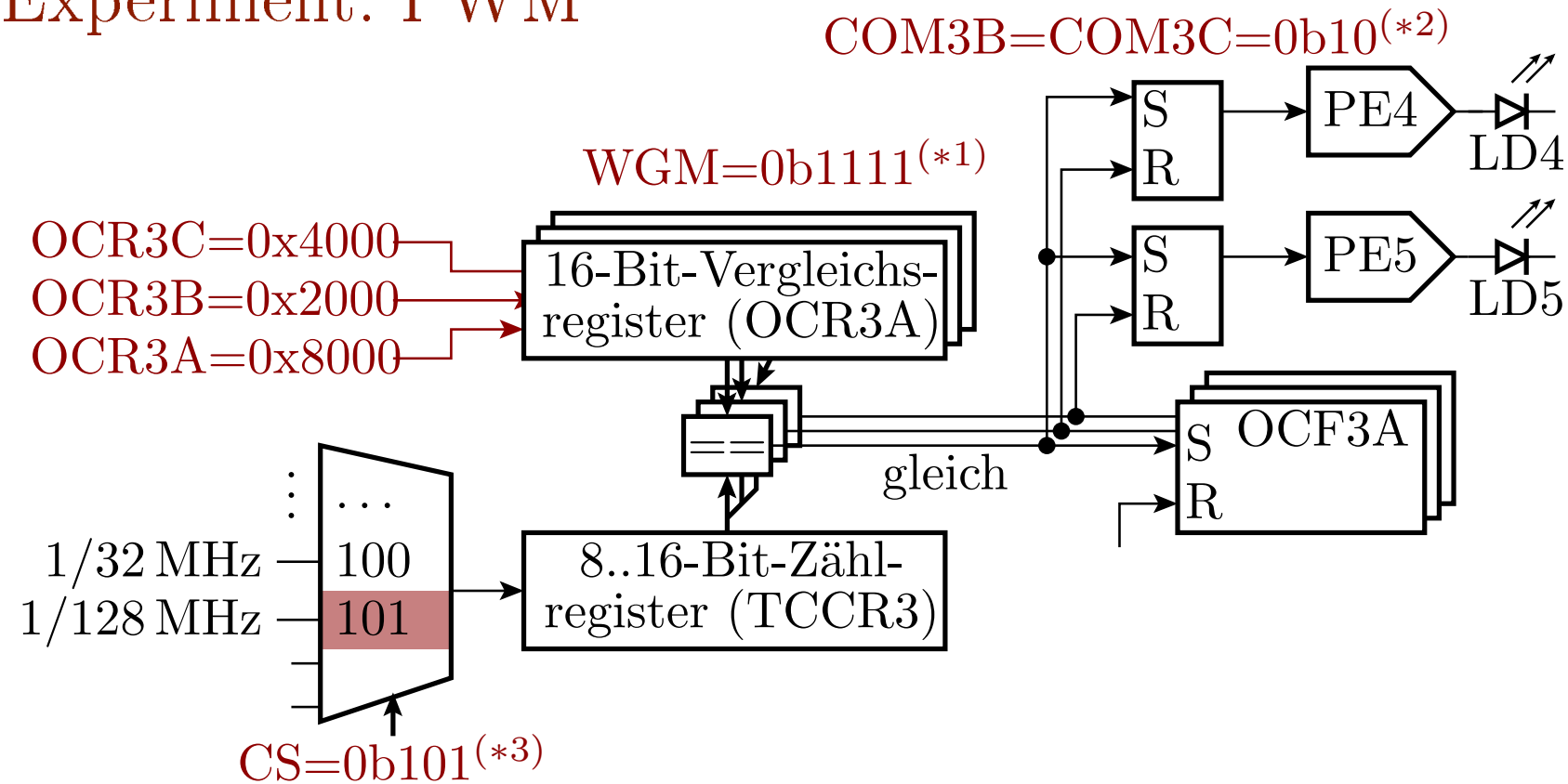
- Timer und LED-Ausgabe initialisieren.
- Wiederhole immer
  - Warte, bis Vergleichsbit »OCF3A« gesetzt.
  - LED-Ausgabe weiterzählern.
  - Vergleichsbit »OCF3A« löschen.
  - neuen Vergleichswert aus der Schaltereingabe bestimmen und in OCR3A schreiben.

```
#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 F6-test_time\test_timer alle außer zweite Main-Funktion auskommentieren.
- Schaltermodul an Port A oben anstecken. SW(4:1)=1100.
- Übersetzen. Start (,). Kontrolle Zähltakt ≈2 Hz.
- Schalterwert erhöhen/verringern und Frequenz kontrollieren.

Experiment: PWM

Experiment: PWM



- (*1) schnelle PWM mit OCR3A als Periodenregister.
- (*3) PE4 und PE5: PWM-Ausgänge invertiert.
- (*2) Zählperiode 128 µs. PWM-Periode: $256\,\mu s \cdot 0x8000 \approx 4{,}2\,s$
- LD4, LD5: LEDs auf PMOD8LD an JE ⇑ stimmt das?

Testprogramm:

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

- LED-Modul »PMOD8LD« an JE stecken.
- Im Projekt F6-test_timer\test_timer alle außer dritte Main-Funktion auskommentieren.
- Übersetzen. Start im Debugger  . Continue  .
- Kontrolle:
  - Blinkperiode ca. 4 s,
  - Ausschaltzeit LED4 25% (ca. 1 s) und
  - Ausschaltzeit LED5 50% (ca. 2 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.

Aufgaben

Aufgabe 6.1: Warteschleife mit Timer

- Ersetzen Sie im Projekt »bit_io3_mod«, Foliensatz 2 in »Warte_1s()« in »myfunc.c« die Wartezählschleife durch eine Wartefunktion mit Timer 3 (Normalmodus).
- Testen Sie bei dem Originalprogramm, wie stark die Wartezeit bei Übersetzung mit »-O0«, »-O1« und »-O2« vom Sollwert 1 s abweicht.
- Wiederholen Sie die Tests mit dem modifizierten Programm.

Hinweise:

- Festlegen eines geeigneten Vorteiler- und Timer-Startwerts.
- Programmstruktur der Wartefunktion:

```
void Warte_1s(){
 <Timer initialisieren und starten>
 while (!<Timerüberlauf>);
 <Timer anhalten>
}
```

Aufgabe 6.2: PWM-Helligkeitssteuerung

Ändern Sie im Experiment PWM ab Folie 31 die Einstellungen von Timer 3 so, dass mit einer Periode von 1 ms

- am Ausgang PE4 eine PWM-Signal mit 10% Einschaltzeit und
- am Ausgang PE5 eine PWM-Signal mit 75% Einschaltzeit

ausgegeben wird. Kontrollieren Sie die PWM-Signale

1. mit einem LED-Modul an JE (kein flimmern, 10% bzw. 75% Helligkeit) und
2. mit dem Logikanalysator (Anstecken der LA-Anschlüsse für Masse, PE4 und PE5 über Doppelstecker an JE, XML-File anpassen, ..., Signalverläufe kontrollieren).

Aufgabe 6.3: Timer-Treiber

Einwickeln Sie einen Timer-Treiber, mit Initialisierungsfunktion, Schrittfunktion und Get-Funktion, die die Zeit seit Programmstart in Millisekunden in einer 32-Bit-Variablen zurückgibt. Header:

```
#ifndef TMRSF_H_              //Dateinname: tmrsf.h
#define TMRSF_H_
#include <avr/io.h>
  void tmrsf_init();          //Initialisierungsfkt.
  void tmrsf_step();          //Schrittfunktion
  uint32_t tmrsf_get();       //Get-Funktion
#endif /* TMRSF_H_ */
```

Schließen Sie für den Test an JA ein »Button Modul« an. Schreiben Sie einen Testrahmen, der die Zeitdifferenz vom Drücken von BTN1 bis zum Drücken von BTN3 auf den LEDs an Port J in 100 ms-Schritten anzeigt.