Rechnerarchitektur, Foliensatz 3
Schnittstellen und Zusatzwerke

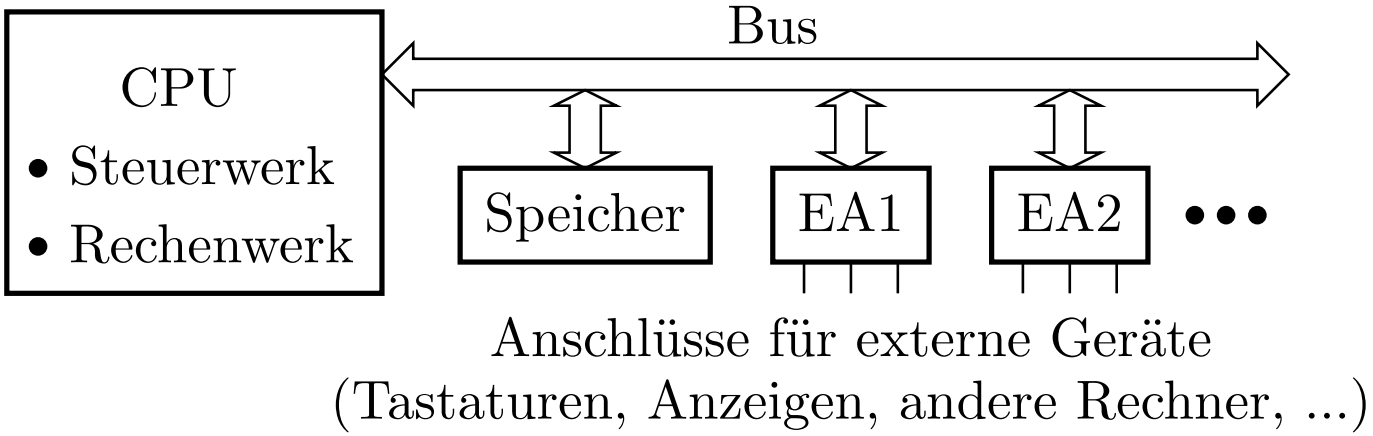
G. Kemnitz

Institut für Informatik, Technische Universität Clausthal

29. Januar 2015

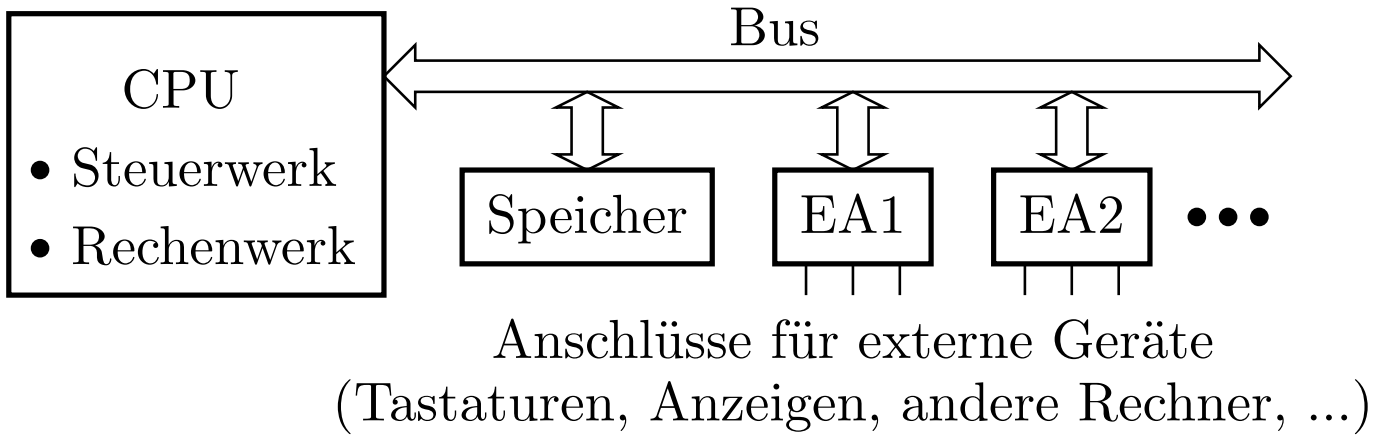
Inhalt des Foliensatzes

Ein- und Ausgabeschnittstellen



Schnittstellen zur Umgebung:

- Digitale Ein- und Ausgänge zur Abfrage von Schaltern, Steuerung von Anzeigen und Aktoren, ...
- Serielle Schnittstellen für den Datenaustausch mit anderen Schaltkreisen und Rechnern.
- Analoge Ein- und Ausgabe z.B. für Sensorwerte.
- Pulsweitenmessung und Pulsweitenmodulation, z.B. für die stufenlose Steuerung von Motoren.



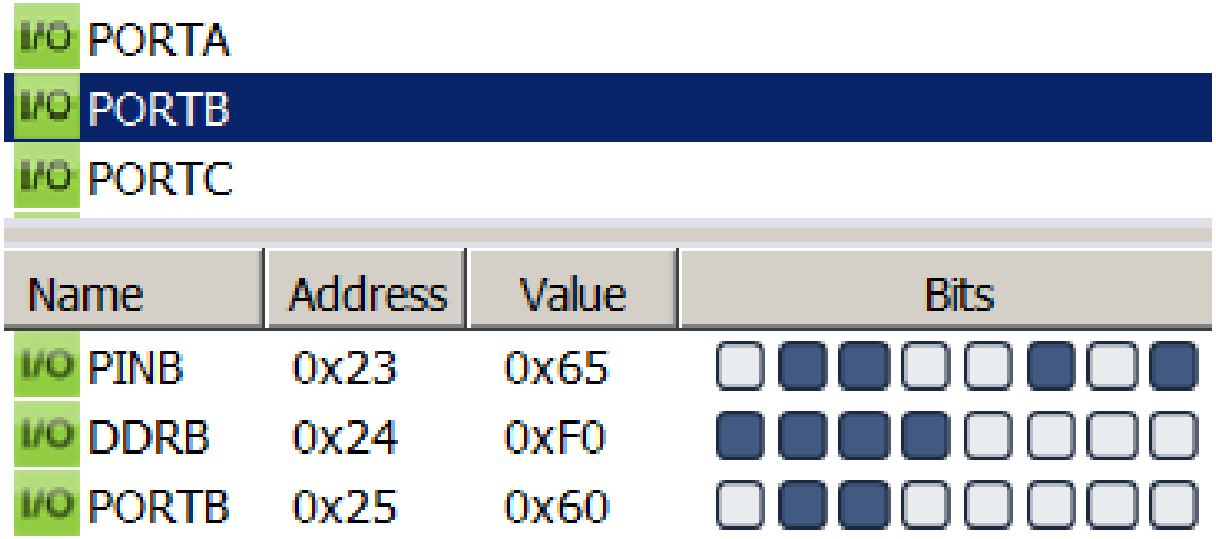
Auf diese Schnittstellen greift der Rechner über EA-Register wie auf den Speicher zu. Über EA-Register hat der Prozessor auch auf weitere Funktionseinheiten Zugriff:

- Timer: Schaltung zur Erzeugung von Zeitereignissen.
- Watchdog: Zeitüberwachungsschaltung zur automatischen Neuinitialisierung bei Programmabsturz.
- Programmierschnittstellen für den Programmspeicher (Flash), den Daten-EEPROM und die Fuse-Register.

Ein Treiber für EA-Schnittstellen muss die richtigen Bits der EA-Register in richtiger Reihenfolge mit richtigen Werten laden.

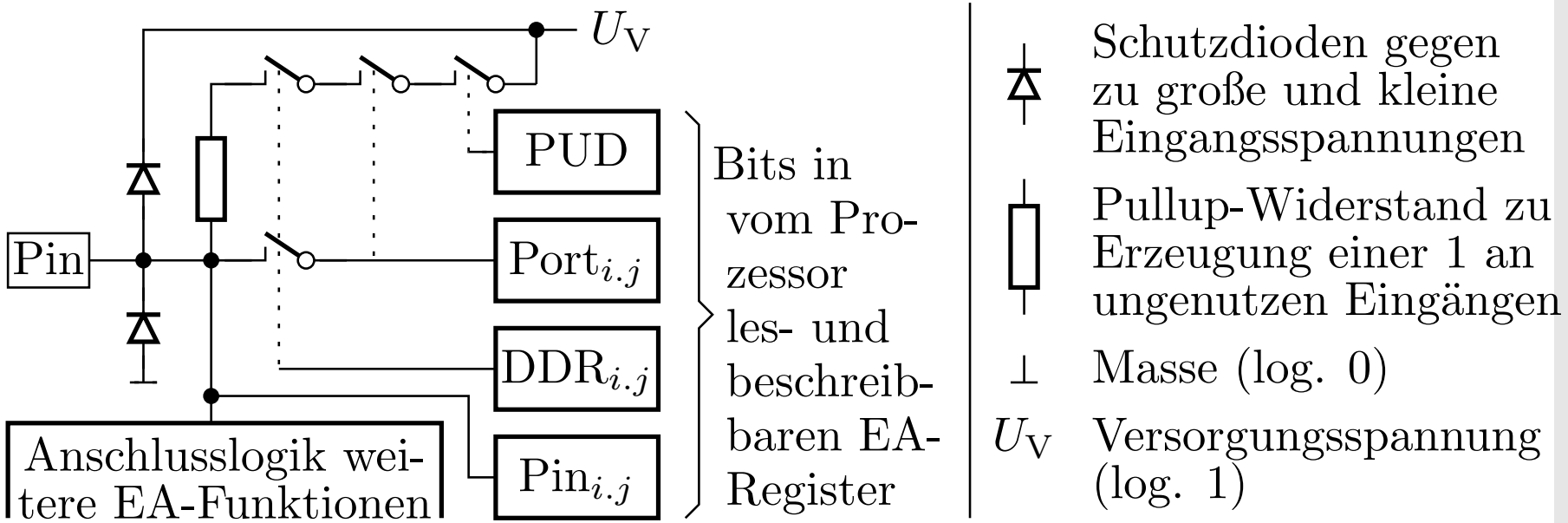
Parallele Schnittstellen

Parallel Schnittstellen (Ports)



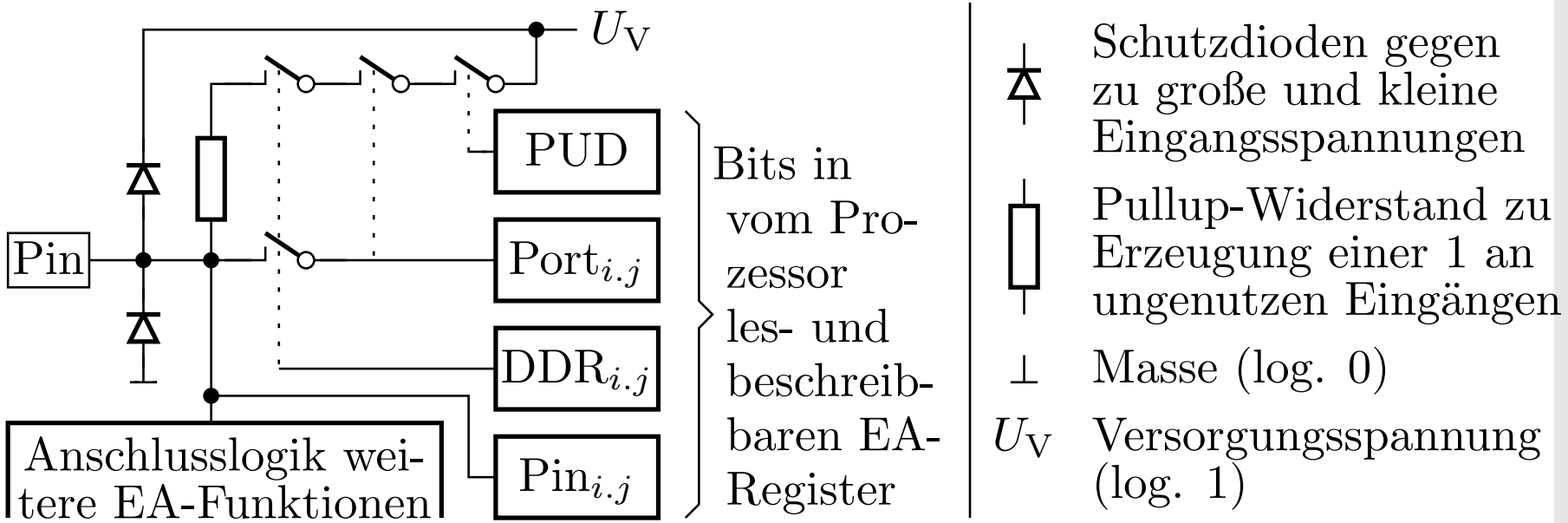
Eine parallele Schnittstelle (Port) fasst je nach Verarbeitungsbreite des Prozessors 8, 16 oder 32 Prozessoranschlüsse zu einer gemeinsam les- und schreibbaren EA-Einheit zusammen. Unser Beispielprozessor hat die 12 Ports A bis L, überwiegen 8 Bit breit. Jeder Port hat drei vom Prozessor les- und beschreibbare Register: Eingaberegister (PINx), Richtungsregister (DDRx) und Ausgaberegister (PORTx).

Schaltung eines einzelnen Anschlusses



Nutzung als Ausgang:

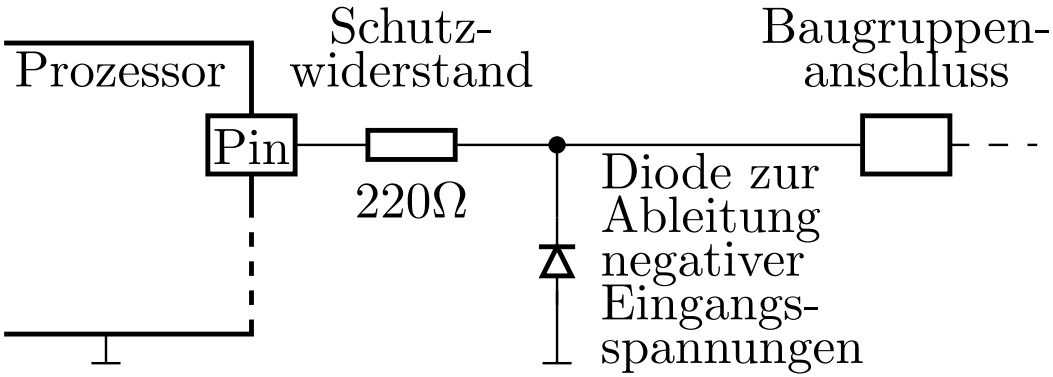
- Richtungsbit eins setzen: $DDR_{i.j} \leftarrow 1$:
- Ausgabe eines Wertes: $PORT_{i.j} \leftarrow$Rd
- Rücklesen des Ausgabewertes: Rd$\leftarrow PORT_{i.j}$
- Lesen des Werts am Pin: Rd$\leftarrow PIN_{i.j}$. $PIN_{i.j} \neq PORT_{i.j}$ ist möglich und deutet auf Programmier- oder Schaltungsfehler.



Nutzung als Eingang:

- Richtungsbit null setzen: $DDR_{i.j} \leftarrow 0$:
- Werte zwischen 0 und 1, z.B. bei ungenutzten Anschlüssen verursachen erhöhte Stromaufnahme.
- Ausgabewert eins ($PORT_{i.j} \leftarrow 1$) und SFR-Bit »PUD« nicht gesetzt, zieht ungenutzte Eingänge über einen Widerstand auf eins. Zu empfehlen für alle ungenutzte Eingänge.
- Bei externer Signalquelle und vor allem für analoge Eingänge Pullup-Widerstand mit ($PORT_{i.j} \leftarrow 0$) deaktivieren.

Port-Beschaltung auf der Versuchsbaugruppe



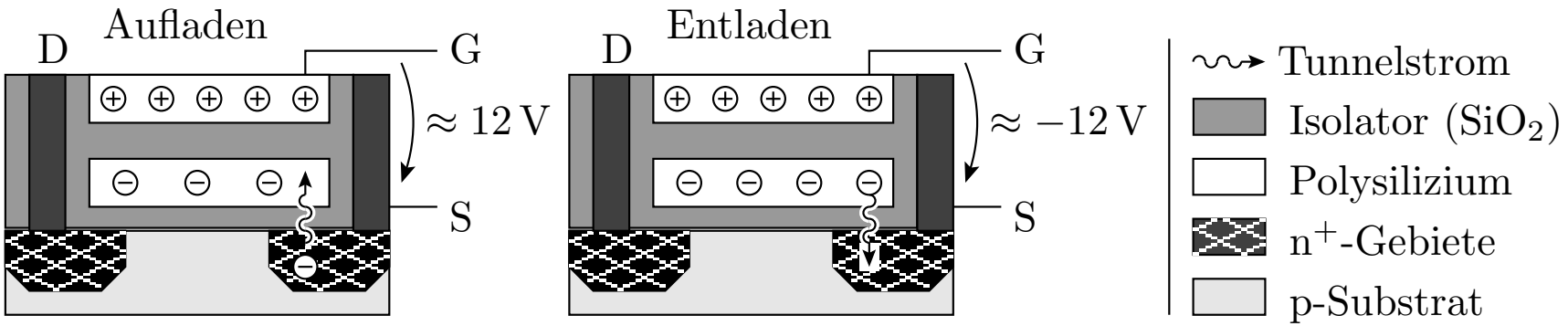
- Empfänger (Ausgabe)
  $DDR_{i.j}$=1; $Port_{i.j}$ ∈{0,1}
- Signalquelle (Eingang)
  $DDR_{i.j}$=0; $Port_{i.j}$=0
- nichts
  $DDR_{i.j}$=1 oder $Port_{i.j}$=1

- Die Baugruppe hat an allen herausgeführten Ports Schutzschaltungen vor Zerstörung bei falscher Port-Initialisierung.
- Für ungenutzte Ports ist die Beibehaltung der Initialwerte $DDR_{i.j} = 0$ und $Port_{i.j} = 0$ wie in den bisherigen Übungen nicht empfehlenswert.

EEPROM

EEPROM

- Der Datenspeicher verliert ohne Spannung seine Information.
- Die Werte für Konstanten stehen im Befehlsspeicher und werden vom Startup-Code in den RAM kopiert.
- Wo werden während des Betriebs anfallende Daten gespeichert, die nach Abschalten nicht verloren gehen dürfen? ⇒ EEPROM (electrically erasable programmable read-only memory).
- Programmierung über Tunnelströme. Schreibzeit ca. 30.000 Prozessortakte.



EEPROM-Ansteuerung

EEPROM-Adressregister	–	–	–	–	Bit 11	Bit 10	Bit 9	Bit 8	EEARH (0x42)
	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	EEARL (0x41)
Datenregister	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	EEDR (0x40)
Kontrollregister	–	–	EEPM1	EEPM0	EERIE	EEMPE	EEPE	EERE	EECR (0x40)

löschen+schreiben	0	0
nur löschen	0	1
nur schreiben	1	0

Bits zur Aktivierung und Steueung der Programmierung

Ablauf zum Schreiben eines Bytes in den EEPROM:

1. Warte bis EEPE null ist[1].
2. Warte bis SPMEN in SPMCSR null ist.
3. Schreibe die Adresse in EEAR und die Daten in EEDR.
4. Schreibe EEMPE← 1 und EEPE← 0.
5. Innerhalb der nächsten 4 Zyklen schreibe auch EEPE← 1.

[1] Warte, bis die vorherige EEPROM-Schreiboperation abgeschlossen ist. Das ist ca. 3.3 ms nach letzten Setzen des Bits EEPE.

Schreiboperation als C-Funktion:

```
void EEPROM_write(uint16_t uiAddress, uint8_t ucData)
  // Warte auf Abschluss letzte Schreiboperation
  while(EECR & (1<<EEPE));
  EEAR = uiAddress;  // Adressübergabe
  EEDR = ucData;     // Datenübergabe
  EECR |= (1<<EEMPE);
  EECR |= (1<<EEPE); // Start der Schreiboperation
}
```

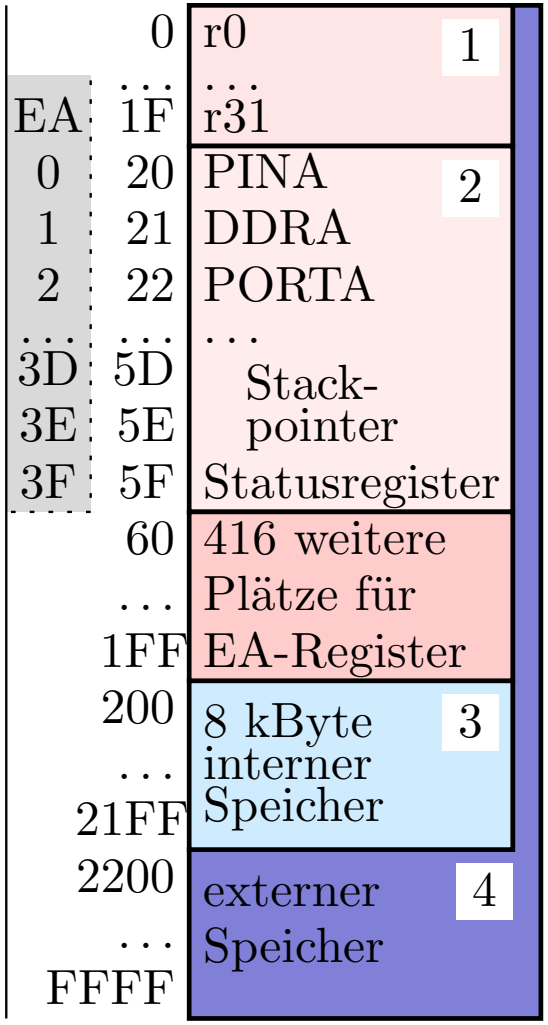
Die Leseoperation muss auf den Abschluss der Schreiboperation warten und ist einen Takt nach Start der Leseoperation fertig:

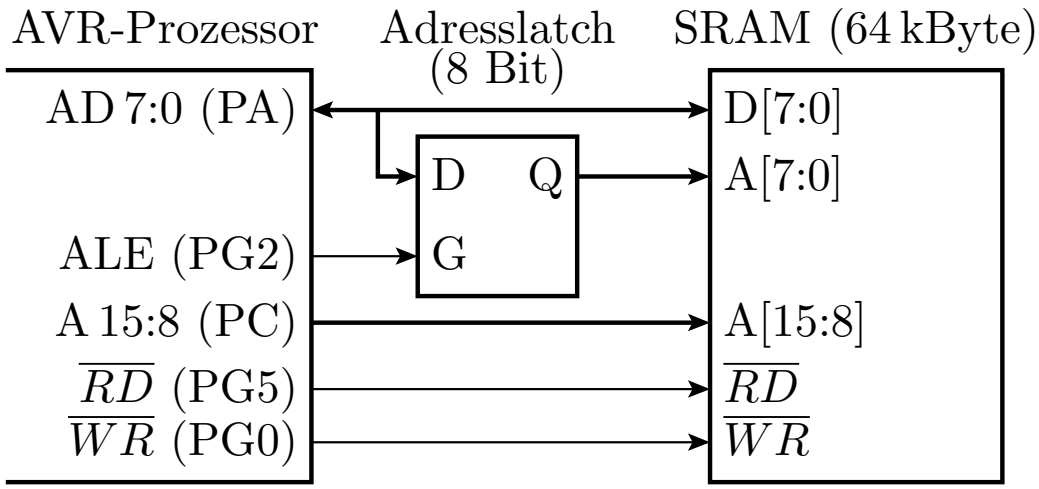
```
unsigned char EEPROM_read(uint16_t uiAddress) {
 // Warte auf Abschluss letzte Schreiboperation
  while(EECR & (1<<EEPE));
  EEAR = uiAddress;  // Adressübergabe
  EECR |= (1<<EERE); // Start der Leseoperation
  return EEDR;       // Ergebnisrückgabe
}
```

Externer Speicher

Externer Speicher

- Zur Erweiterung des Datenspeichers kann ein max. 64 kByte großer externe Speicher an den Prozessor angeschlossen werden.
- Anschluss an die Ports A, C und G.
- Damit Lade- und Speicherbefehle auf den externen Speicher zugreifen, sind in SFR-Register XMCRA das Bit SRE (Bit 7) und gegebenenfalls weitere Bits in XMCRA und XMCRB zu setzen.





- Port A liefert das niederwertige Adressbyte und übernimmt einen Schritt später die Lesedaten bzw. gibt die Schreibdaten aus.
- Port C liefert das höherwertige Adressbyte. Ein Teil von Port G liefert die Steuersignale.
- Wenn der externe, auf der Versuchsbaugruppe vorhandene Speicher nicht genutzt wird (SRE=0), sind auf Port G5 und Port G0 jeweils eine »1« auszugeben[2].

[2] Durch falsche Ansteuerung sind bereits auf mehreren Versuchsbaugruppen im Übungsraum die Speicher-Chips kaputt gegangen.

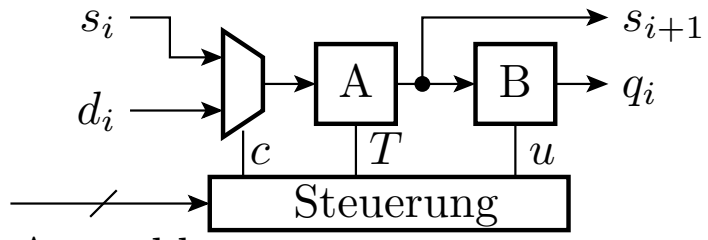
Serielle Schnittstellen

Serieller Datenaustausch

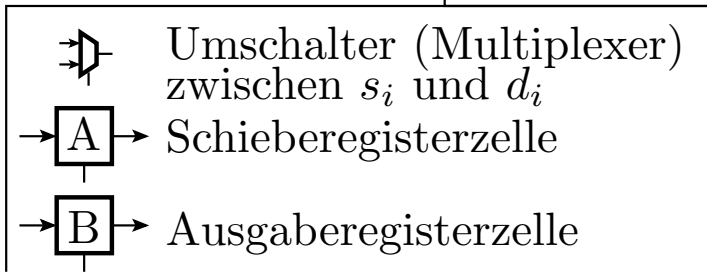
Der Datenaustausch zwischen Rechnern erfolgt in der Regel seriell[3]. Grundbaustein Schieberegister mit den Funktionen

- parallele Übernahme der zu übertragenden Daten,
- serielle Übertragung und
- paralleler Übergabe.

Schaltung einer Schieberegisterzelle:



Steueraufgabe	c	T	u	Operation
Schieben	0	$\int$	0	$s_{i+1} \leftarrow s_i$
Übernahme	1	$\int$	0	$s_{i+1} \leftarrow d_i$
Übergabe	–	0	$\int$	$q_i \leftarrow s_{i+1}$
keine	–	0	0	——

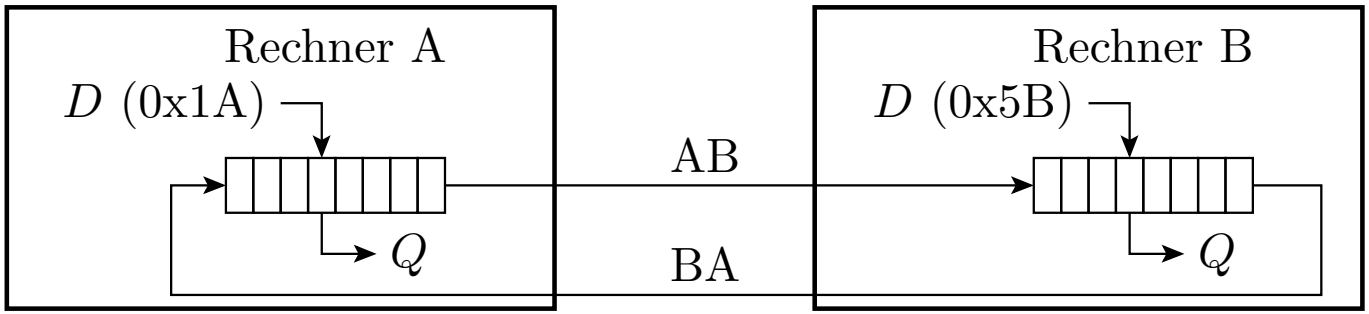


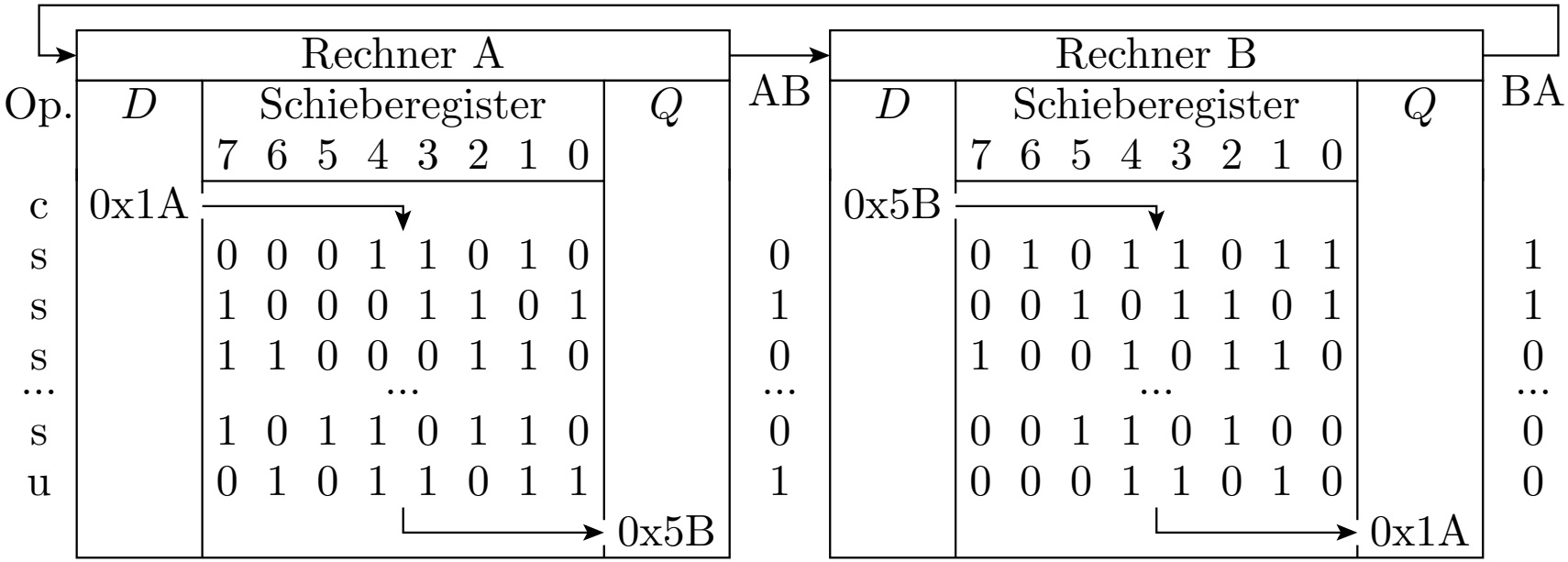
interne Takt- und Steuersignale:

u Update
c Capture
T Takt

[3] Seriell, d.h. hintereinander über eine, statt parallel über viele Leitungen.

Bidirektionale Kopplung zweier Rechner



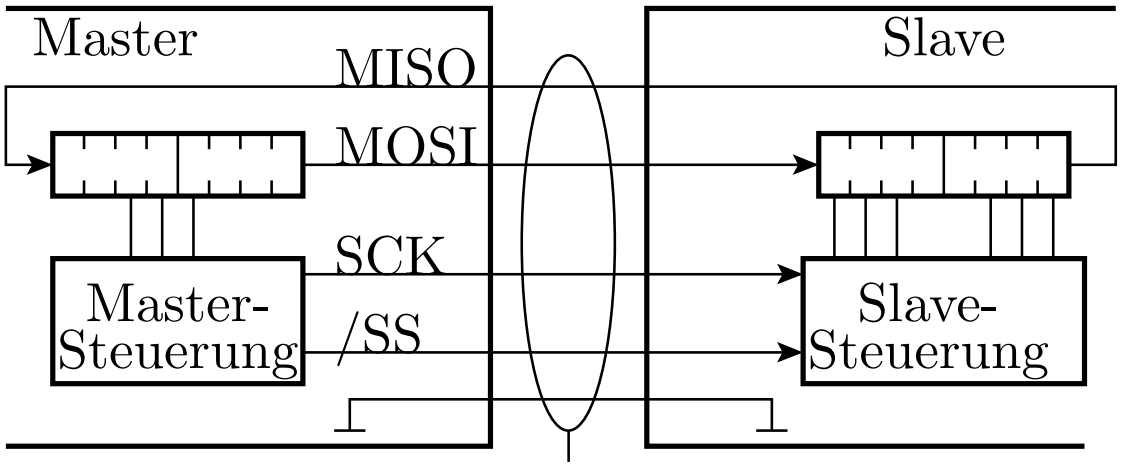


c Übernahme (Capture) s Schieben (Shift) u Übergabe (Update)

SPI-Bus

SPI-Bus

Serieller Bus zur Vernetzung von Schaltkreisen.

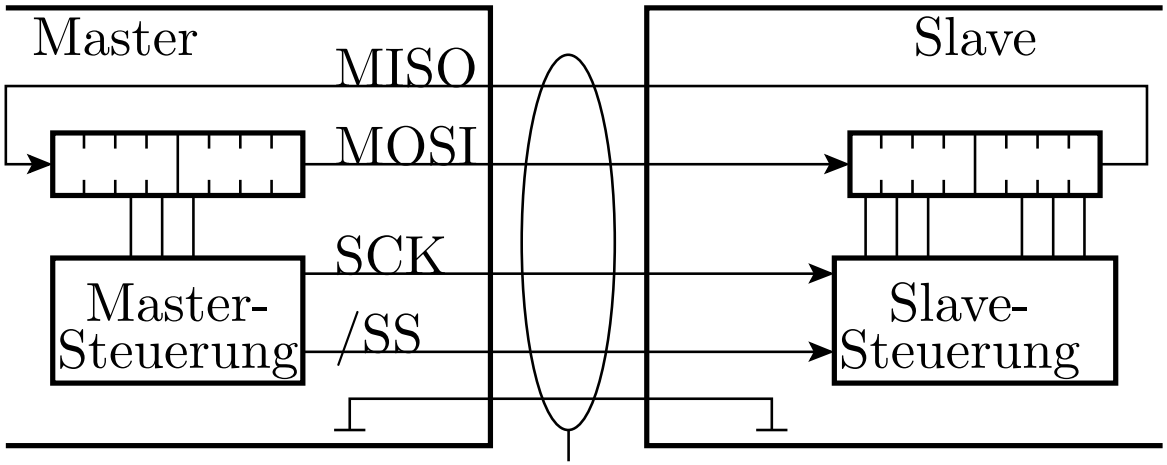


Leitungen zwischen den Schaltkreisen

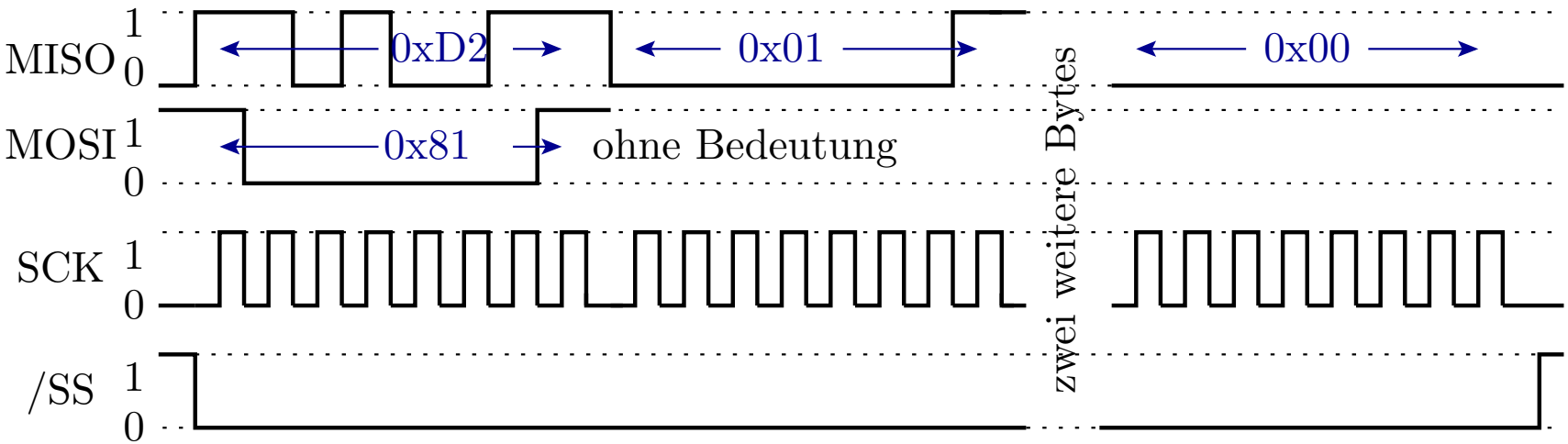
MOSI	Master Out Slave In
MISO	Master In Slave Out
SCK	Schiebetakt
/SS	Slave-Auswahl

Aktion	/SS	SCK
Übernahme	⊓	
Schieben	0	⊓
Ausgabe	⊓	
keine	sonst	

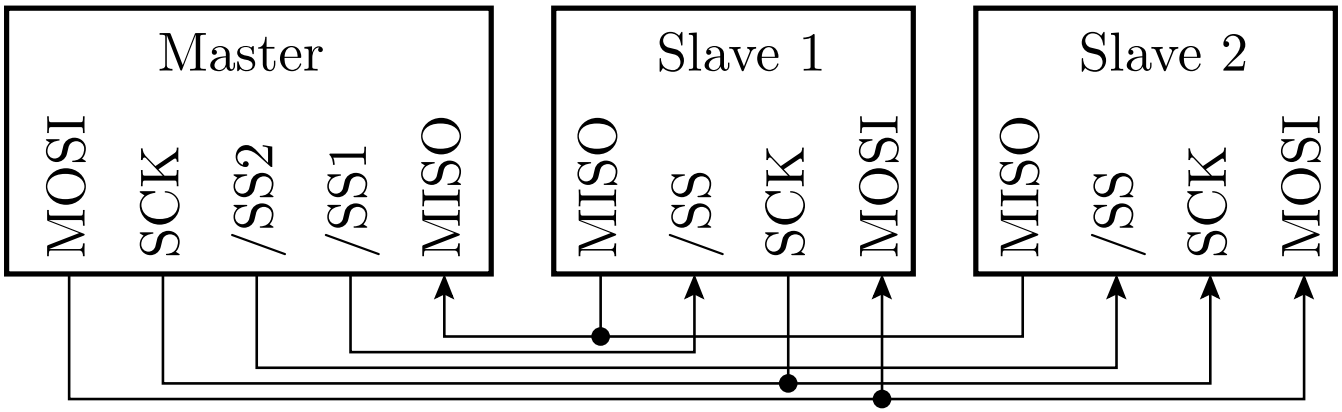
- Ein Schaltkreis ist der Master, der den Takt SCK und die Slave-Auswahlsignale erzeugt, die anderen sind Slaves, die diese Signale vom Master erhalten.



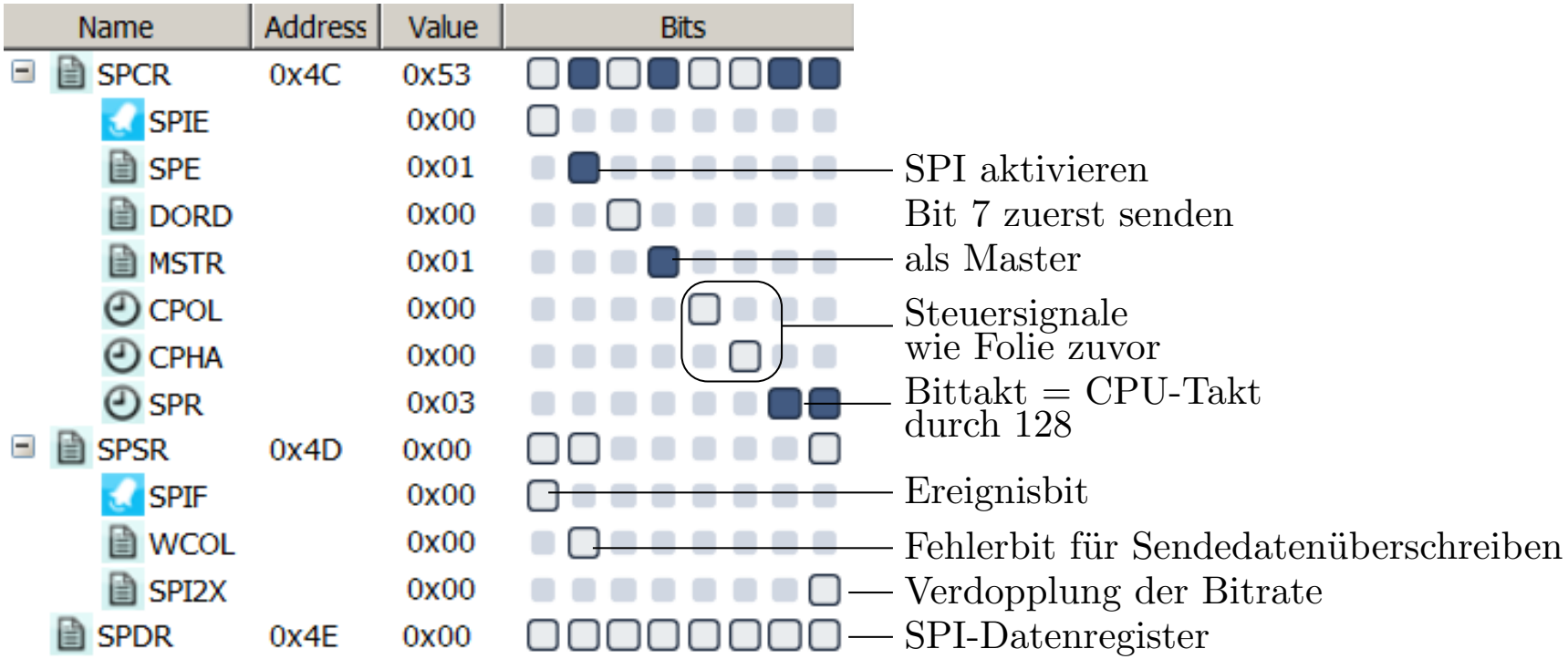
Beispiel für die Übertragung 0x81 vom Master zum Slave und von 0xD201...00 vom Slave zum Master:



- Eine Übertragung beginnt mit Aktivierung von /SS=0 (Übernahme), gefolgt von n Schiebetakten und endet mit Deaktivierung /SS=1.
- Das /SS-Signal des Masters wird über einen Ausgang einer parallelen Schnittstelle gesteuert.
- An einen Master können parallel mehrere Slaves mit unterschiedlichen /SS-Signalen angesteuert werden.



Konfiguration des SPI-Controllers



- Einschalten als Master oder Slave.
- Festlegen der Bitrate und Protokollparameter.
- Pins für /SS Signale konfigurieren, beim Master als Ausgänge mit Wert eins, beim Slave als Eingänge.

Algorithmus für den Datenaustausch

Name	Address	Value	Bits
SPSR	0x4D	0x00	
SPIF		0x00	
SPDR	0x4E	0x00	

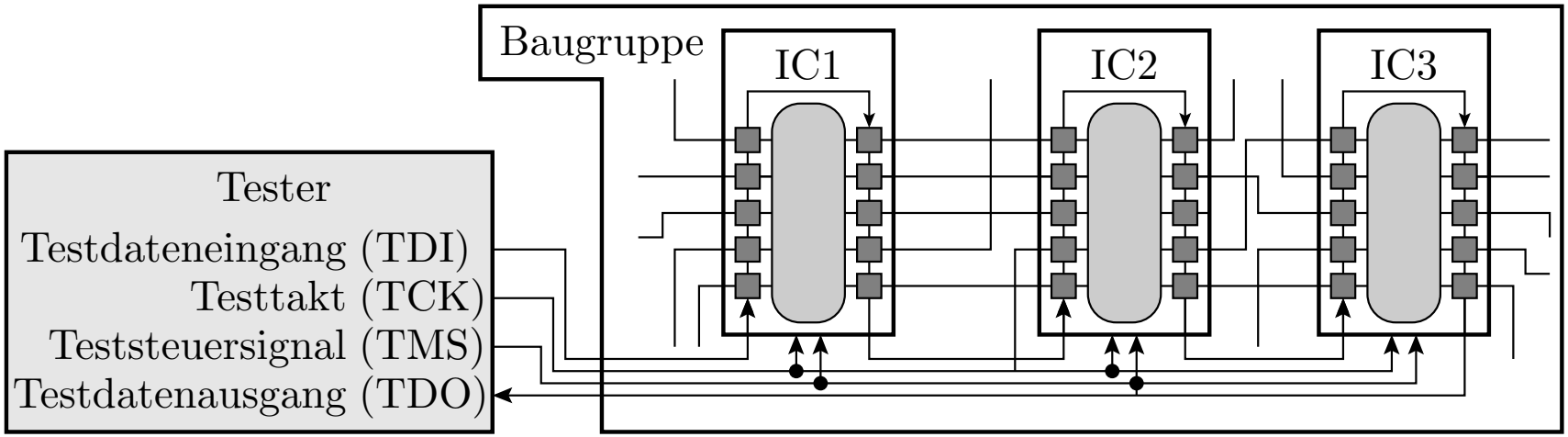
Für jede n-Byte-Übertragung

- Aktiviere das Slave-Auswahlsignal /SS (Master) bzw. warte auf /SS=0 (Slave).
- Für jedes Byte
  - Schreibe Sendewert in das Datenregister
  - warte bis Ereignisbit SPIF eins ist
  - Lese empfangenes Byte aus und schreibe nächstes zu sendende Byte in das SPI-Datenregister SPDR.
- Deaktiviere das Slave-Auswahlsignal.

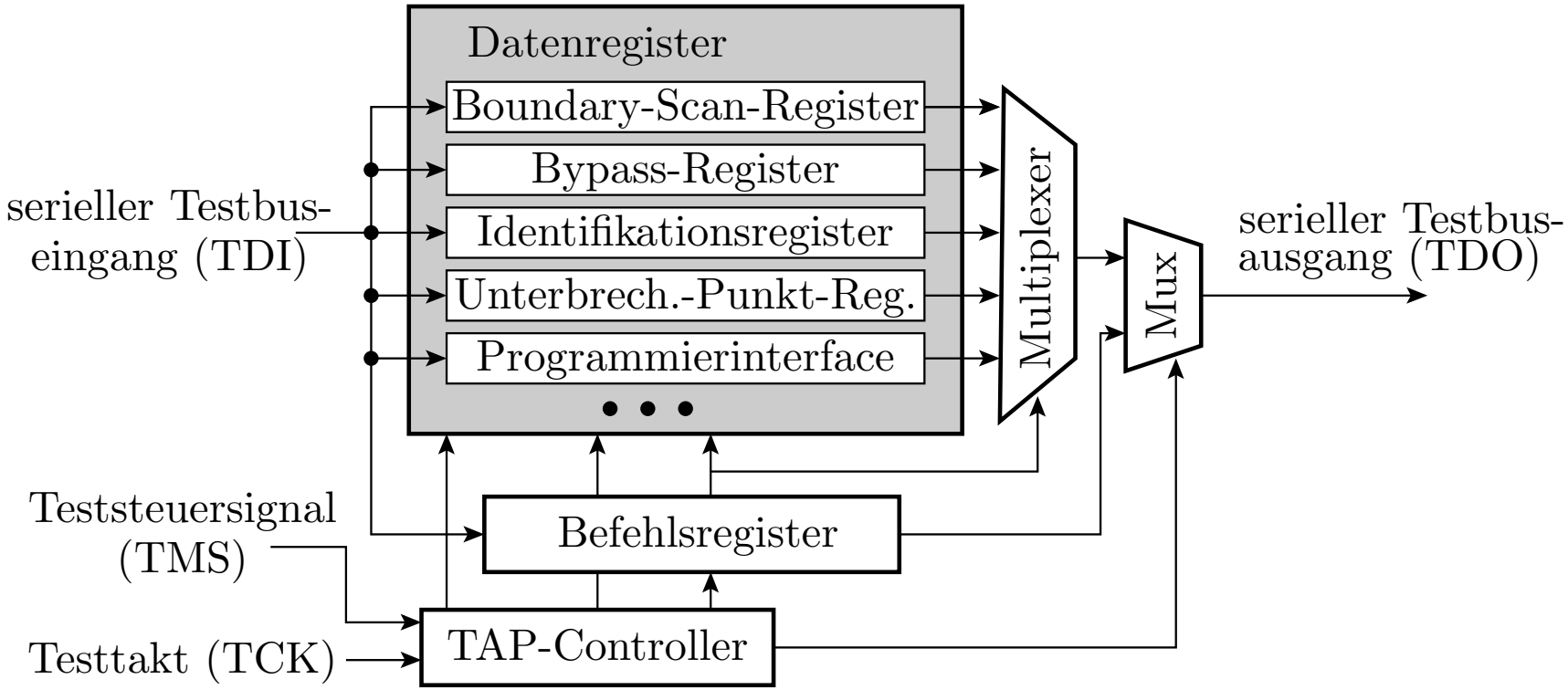
JTAG (Testbus)

JTAG

Test-, Diagnose-, Debug- und Programmierbus.

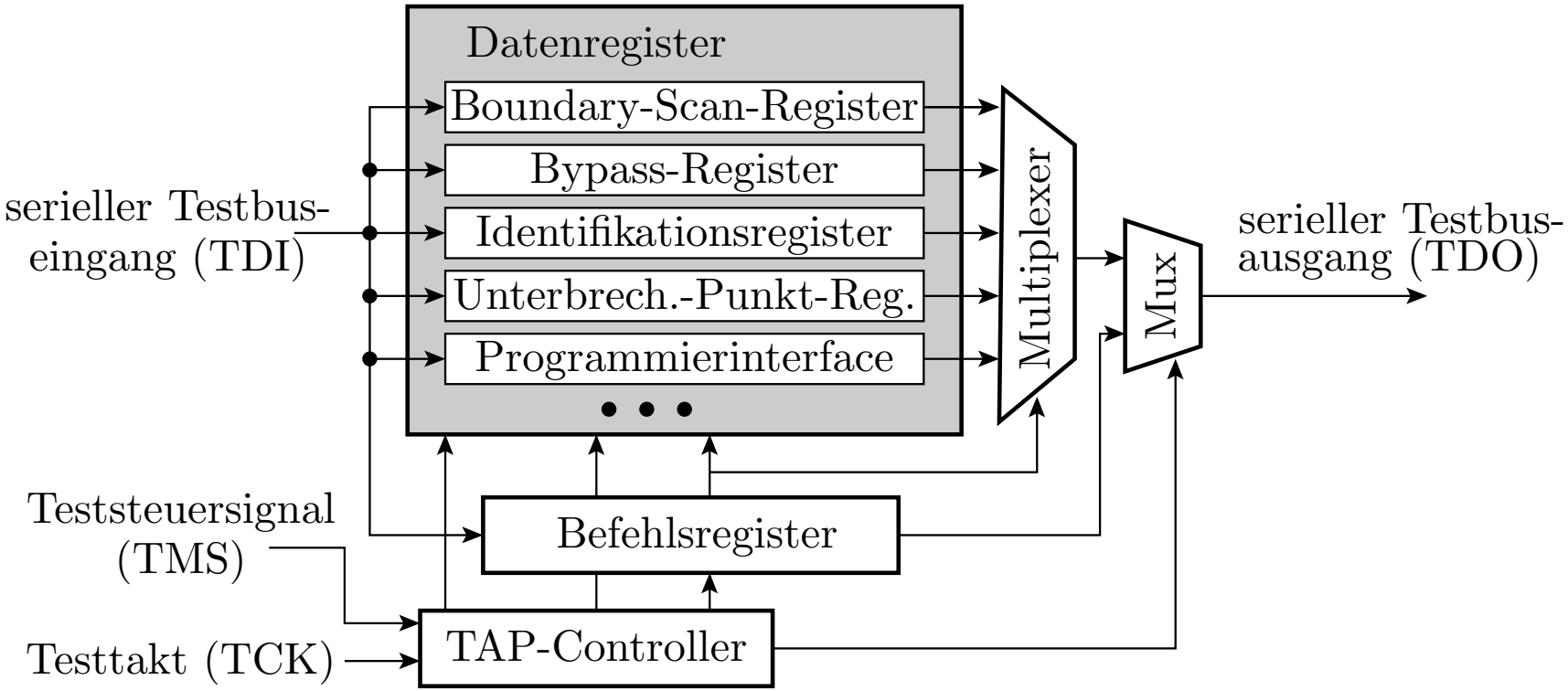


- Verschaltung aller Schaltkreise zu einer Kette.
- In der Ursprungsidee (Boundary-Scan) waren die Schieberegister mit den Funktionen Übernahme, Schieben und Übergabe am Schaltkreisrand angeordnet, um die Verbindungen zwischen den Schaltkreisen ohne mechanische Kontaktierung testen zu können.



Ein Schaltkreis mit JTAG-Bus hat mehrere über ein Befehlswort auswählbare Datenregister:

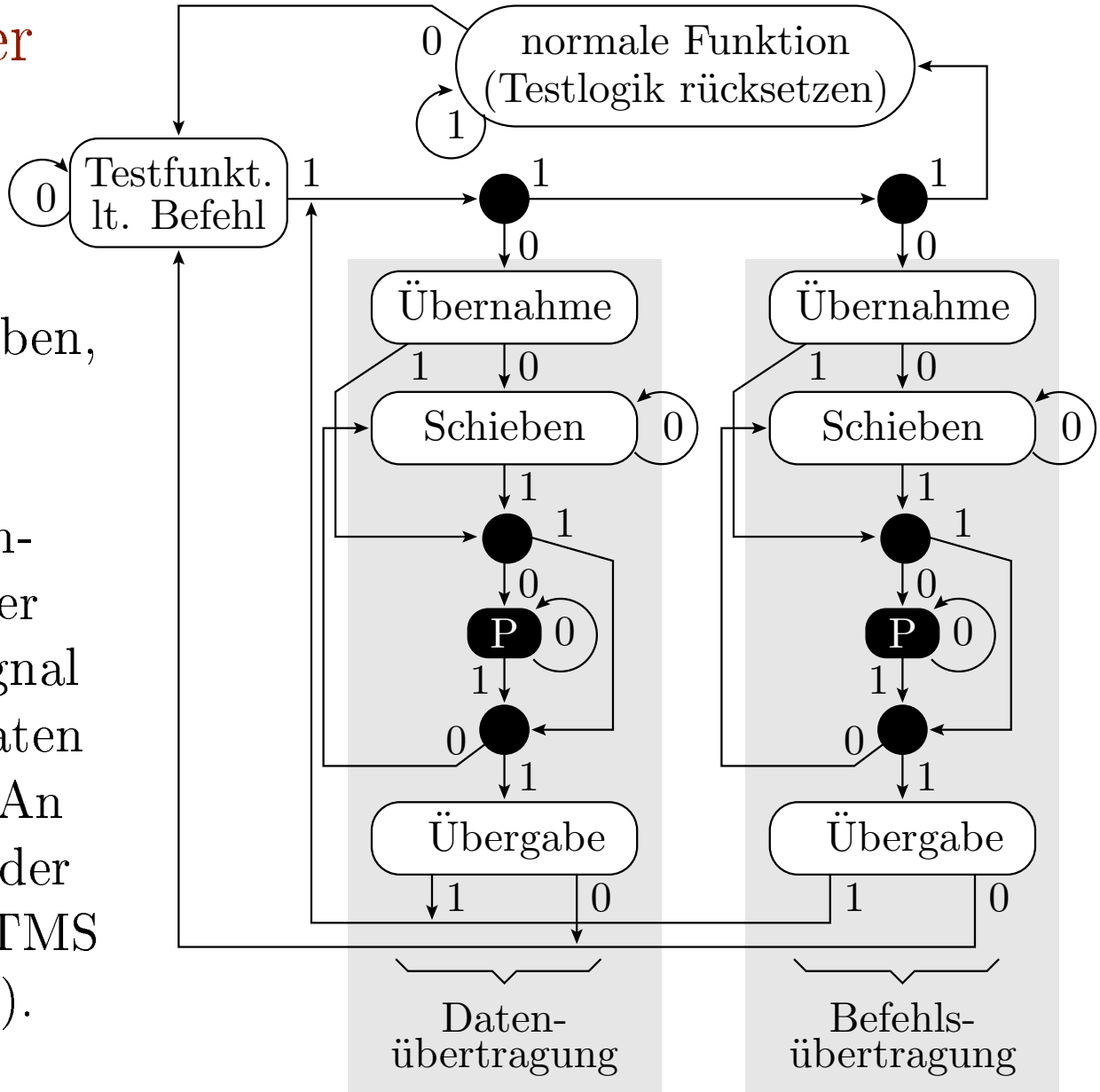
- Bypass-Register zur Verkürzung der Länge des Schieberegisters durch den Schaltkreis auf 1 Bit.
- Identifikationsregister mit Hersteller- und Bauteilnummer.



- Programmier-Interface: Schnittstelle zum Lesen und Schreiben des Befehls-Flashes, des Daten-EEPROMs und der Fuse-Register.
- Schnittstellenregister zum OCD (On-Chip Debugger), ...

TAP-Controller

Die Auswahl der 6 Busaktionen: Übernahme, Schieben, Übergabe für des Befehls- und das ausgewählte Datenregister erfolgt über ein 1-Bit-Steuersignal und einen Automaten mit 8 Zuständen. An den Kanten steht der Wert des Signals TMS (Test Mode Select).

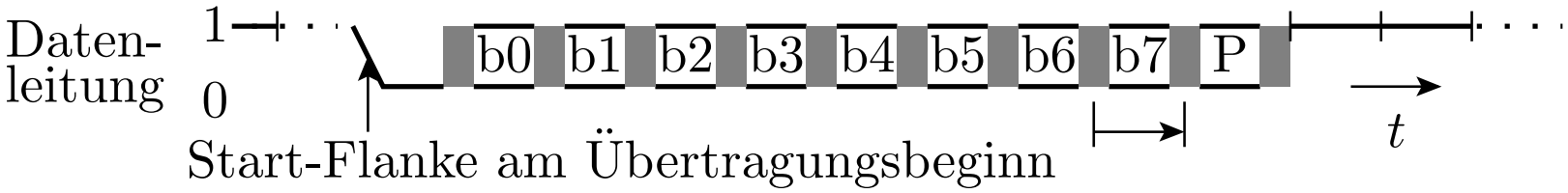


Von der JTAG-Implementierung in unserem Prozessor sind nur die standardisierten Testfunktionen, die für den Bestückungstest von Baugruppen vorgesehen sind, öffentlich zugänglich dokumentiert. Die Befehle für die Programmierung und den OCD (On-Chip Debugger) sind in den Dokumentationen nicht beschrieben.

USART

USART (Universal Asynchronous Receiver Transmitter)

Übertragung ohne Takt und Steuersignale.



$b \in \{0,1\}$	Datenbits	$\cdots$	Übertragungspause
$P \in \{0,1\}$	Paritätsbit	$\longmapsto$	Bitzeit, z.B. $t_{\text{Bit}} \approx 0{,}1\,\text{ms}$
$\vdash\!\!\dashv$	Stoppbit (Wert 1)		Übertragungsdauer: $12 \cdot t_{\text{Bit}}$

Der Empfänger erkennt den Übertragungsbeginn an der Stopp-/Start-Flanke und übernimmt die Werte nach 1,5, 2,5 etc. Bitzeiten. Voraussetzung: Gleich eingestellte Bitzeit, Bitanzahl, Stoppbitanzahl und Parität bei Sender und Empfänger. Die Baudrate b als Kehrwert der Bitzeit t_{Bit} wird mit einem Teiler aus dem Prozessortakt gebildet.

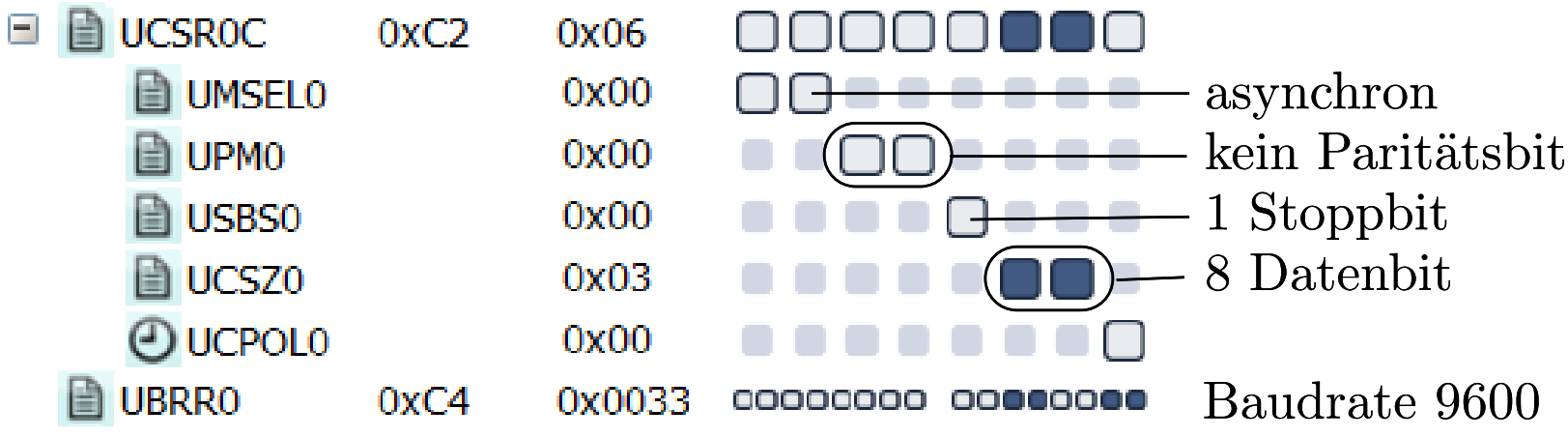
Baudrate b als Funktion der Prozessortaktfrequenz f_{Proz} und dem programmierbaren Teilerfaktor T:

$$b = \frac{f_{\text{Proz}}}{16 \cdot (T + 1)}$$

Teilerfaktor für $f_{\text{Proz}} = 8\,\text{MHz}$ und $b = 9600\,\text{Baud}$:

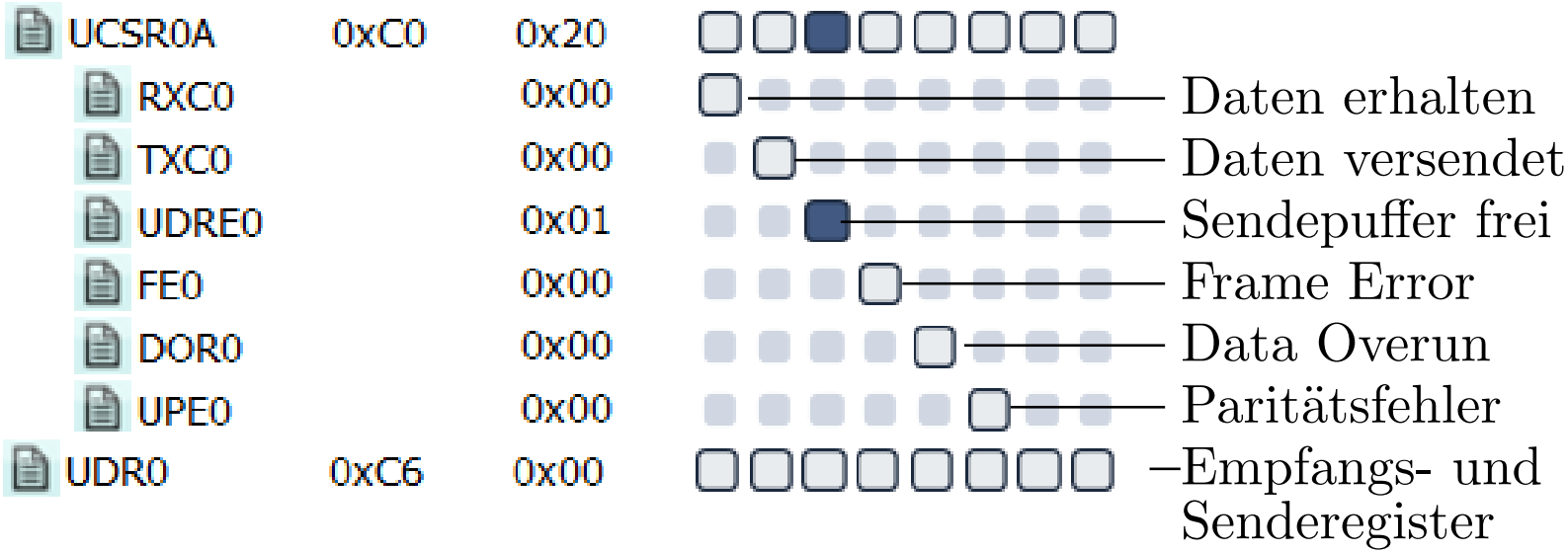
$$T = \frac{8\,\text{MHz}}{16 \cdot 9600\,\text{Hz}} - 1 = 51 = 0\text{x}33$$

Initialisierung: 9600 Baud, 8 Daten-, 1 Stoppbit, keine Parität:




```
UBRR0  = 0x33;          Teilerwert für 9600 baud
UCSR0B = 0b00011000;    Sender und Empfänger einschalten
UCSR0C = 0b00000110;    Datenformat 8n1
```

Sende- und Empfang



Funktionen für den Empfang und das Versenden eines Bytes

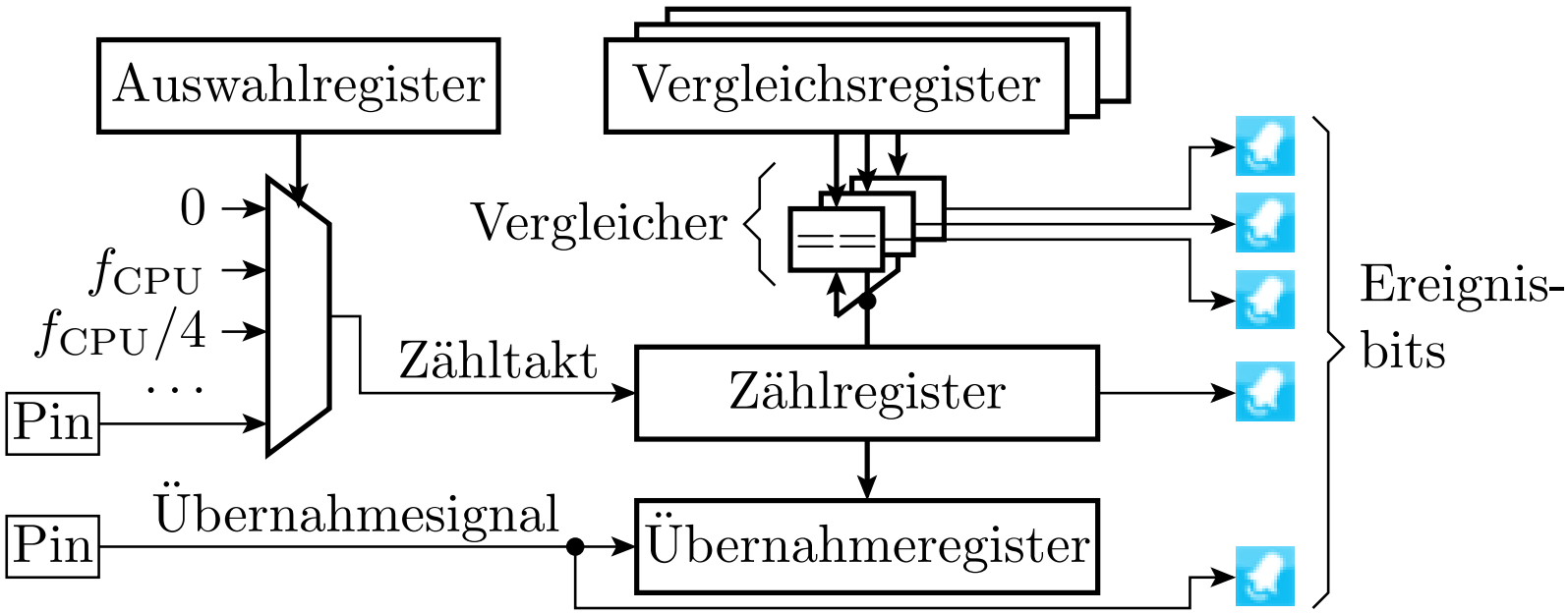
```
uint8_t getChar(){
  while (!(UCSR1A & (1<<RXC0))){}; // Warte auf Empfang
  return UDR0;                     // Rückgabe Empfangsbyte
}

void putChar(uint8_t c){
  while (!(UCSR1A & (1<<UDRE0))){};// Warte bis Sendepuffer
  UDR0 = c;                        // frei. Byte versenden
}
```

Zähler und Zeitgeber

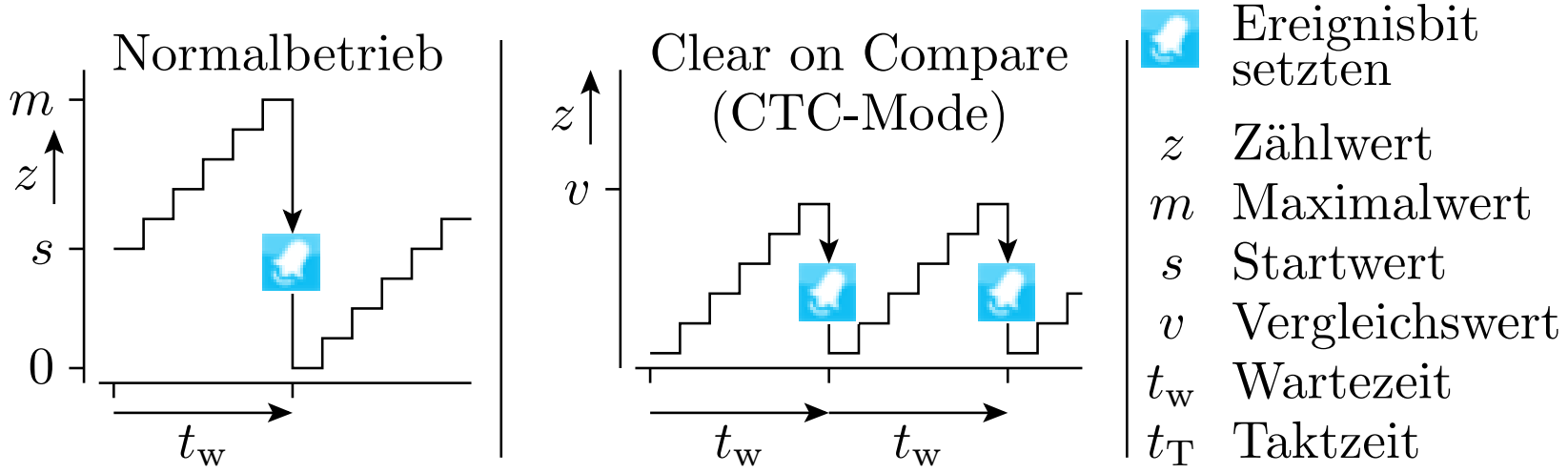
Zähler-/Zeitgeber

Multifunktionsbaustein mit einem Zähler, wählbarer Quelle für zu zählende Impulse, Vergleichsregistern, Registern für die Übernahme von Zählwerten, ...



Normal- und CTC-Betrieb

Betriebsarten als Zeitgeber



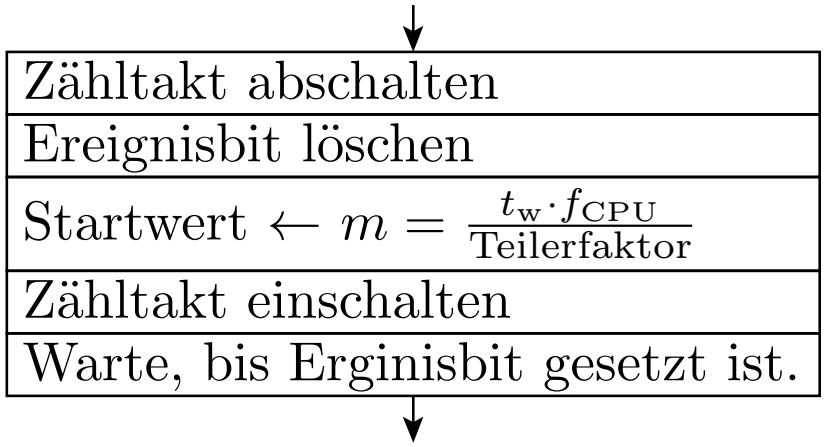
- Normalbetrieb: Initialisierung mit einem Startwert. Zähltakt einschalten. Warten, bis Ereignisbit gesetzt ist. Wartezeit:

$$t_w = (m - s) \cdot t_T$$

- Erzeugung periodischer Zeitsignale: Vergleichswert einstellen und Zähler rücksetzen. Zähltakt einschalten. Wiederhole, wenn Ereignisbit gesetzt wird. Ereignisperiode:

$$t_w = v \cdot t_T$$

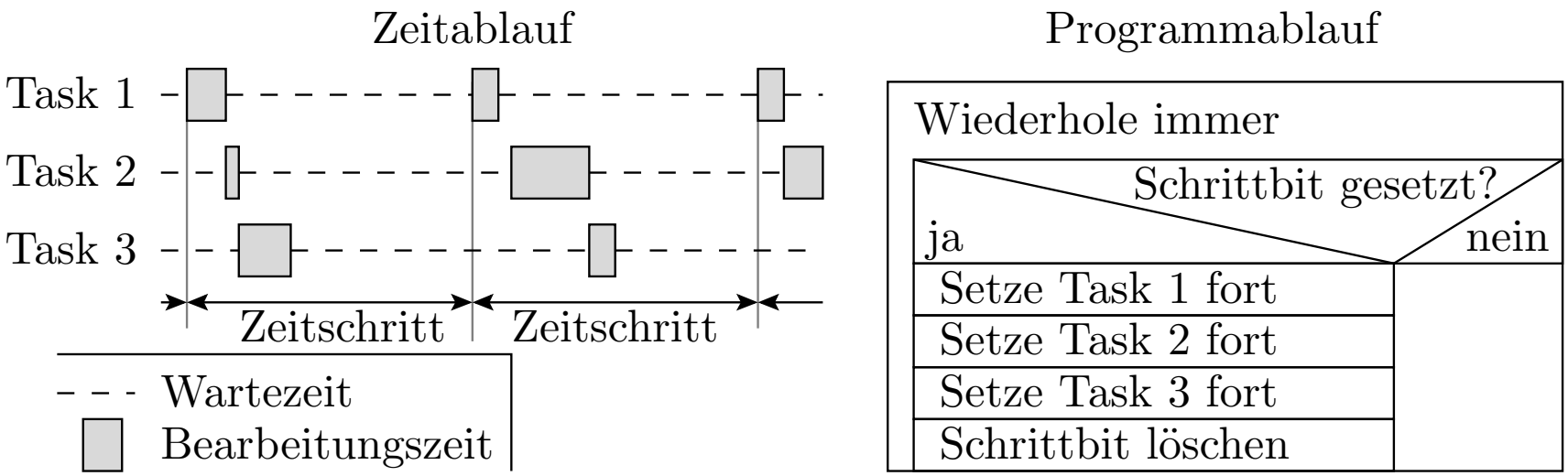
Wartefunktion mit einem Timer



Vorteile gegenüber einer einfachen Programmwarteschleife:

- Insbesondere in C einfacher zu programmieren.
- Wird vom Compiler nicht wegoptimiert.
- Wartezeit ändert sich nicht beim Übersetzrn mit einem anderen Compilern oder einer anderen Compileroptimierung, ...

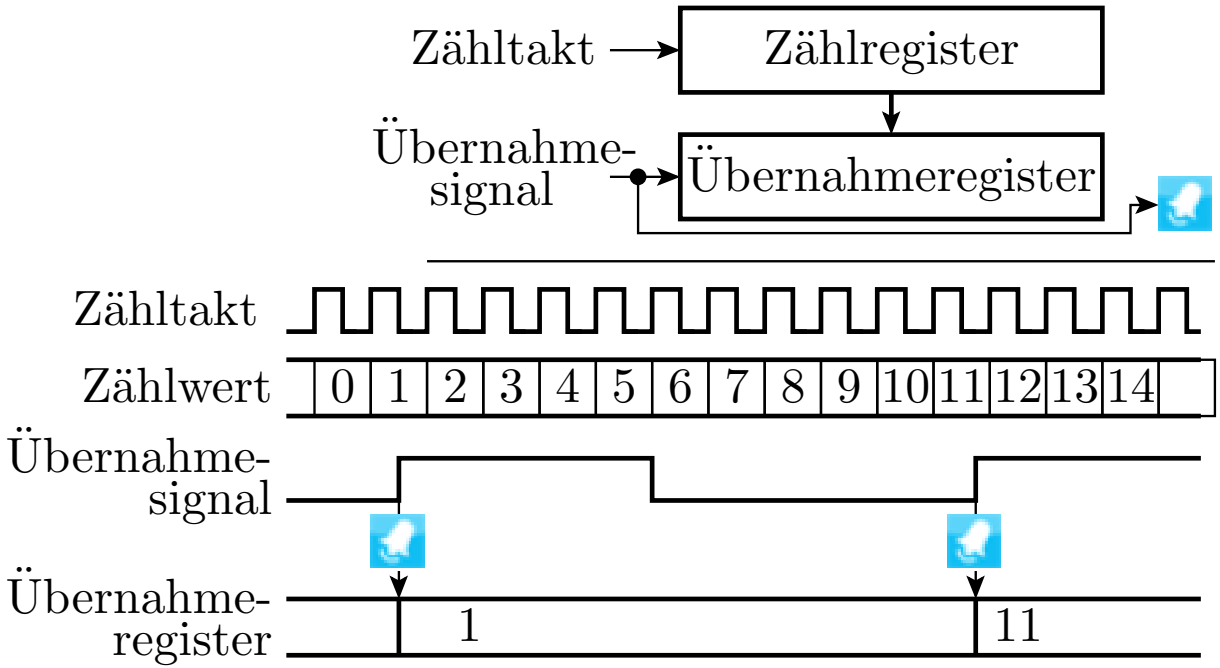
Periodische Dienste



Ein Mikrorechnerprogramm besteht in der Regel aus mehreren Tasks, wie Sensorwerte und andere Eingaben lesen, Berechnungen ausführen und Aktoren ansteuern. Programmierung mit einem Timer im CTC-Modus, der zyklisch mit einer Periodendauer t_w ein Ereignisbit setzt. Das Programm wartet auf dieses Ereignisbit, führt die Schrittfunktionen aus, z.B. das Weiterschalten einer Uhr, und setzt das Ereignisbit zurück.

Zeitmessung

Zeitmessung



Initialisierung

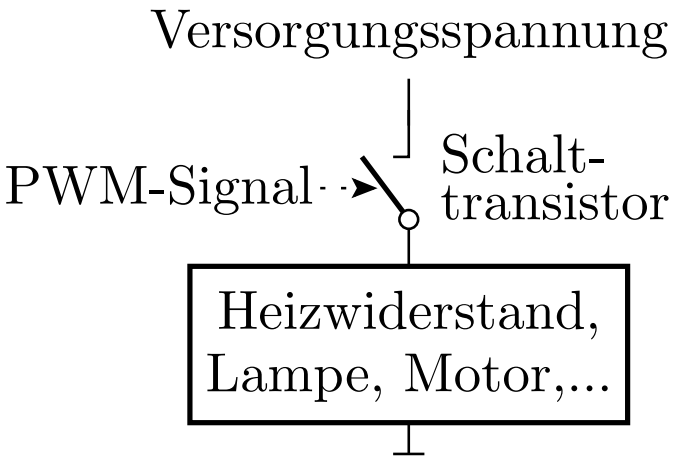
- Auswahl des Zähltakts
- Auswahl der Übernahmebedingung (steigende, fallende, beide Flanken des Übernahmesignals)

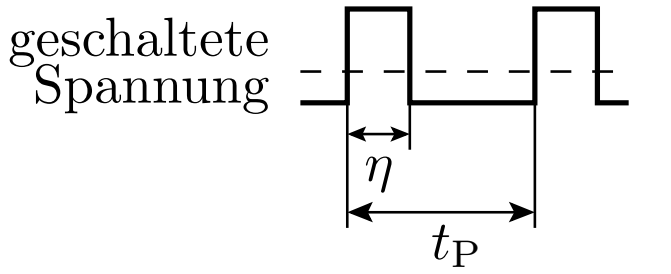
Messung von Zeitdifferenzen:

- Warte auf Ereignisbit »Übernahme«
- Lese Übernahmeregister und Subtrahiere den Übernahmewert vom Ereignis zuvor.

PWM

PWM-Signale



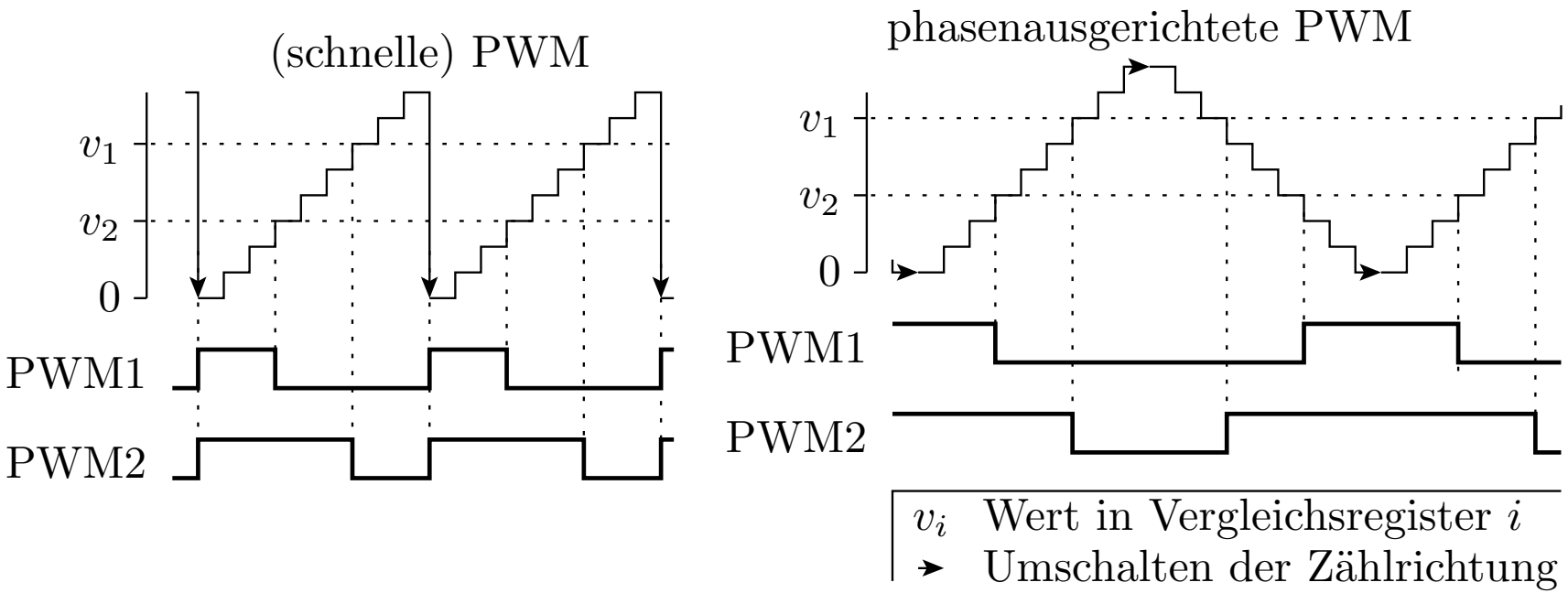


η	relative Pulsbreite
t_P	Periodendauer
- - -	mittlere Spannung, Strom oder Leistung

PWM- (PulsWeitenModulierte) Signale dienen zur

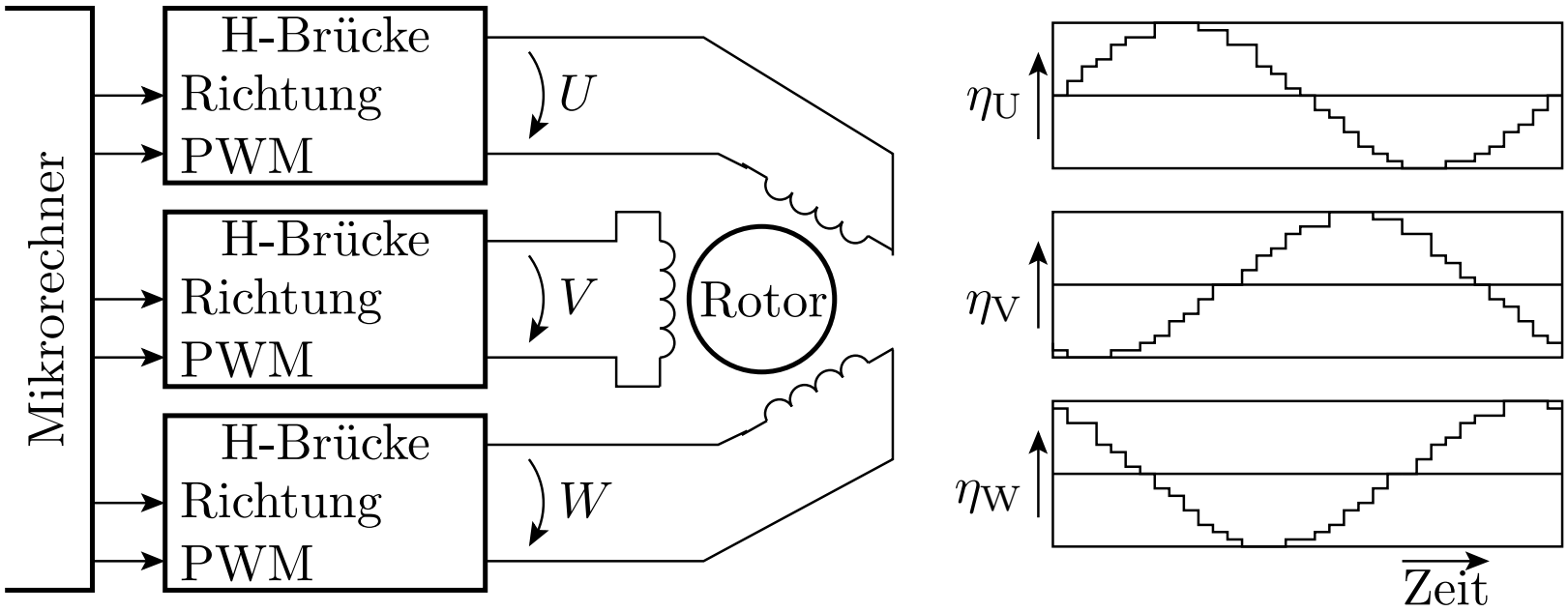
- feingestuften Steuerung von Aktoren (Heizleistung, Helligkeit, Motordrehzahl) und
- zur Informationsübertragung (Sender sendet PWM-Signale und der Empfänger misst die Pulsbreite).

Erzeugen von PWM-Signalen



- Ein- und Ausschalten der PWM-Ausgänge ohne Programmunterstützung.
- Ausgabe der PWM-Signale auch negiert möglich.
- Die Periode wird entweder durch den Zählerüberlauf oder einen weiteren Vergleichswert in einem Register bestimmt.

Typische Motoransteuerung



- Die Erzeugung von 3 sinusförmigen Mittelwertverläufen erfordert eine PWM-Einheit mit drei Vergleichsregistern.
- Ansteuerung über H-Brücken.
- Erlaubt eine stufenlose Positions-, Geschwindigkeits- und Drehmomentsteuerung sogar für Asynchronmaschinen.

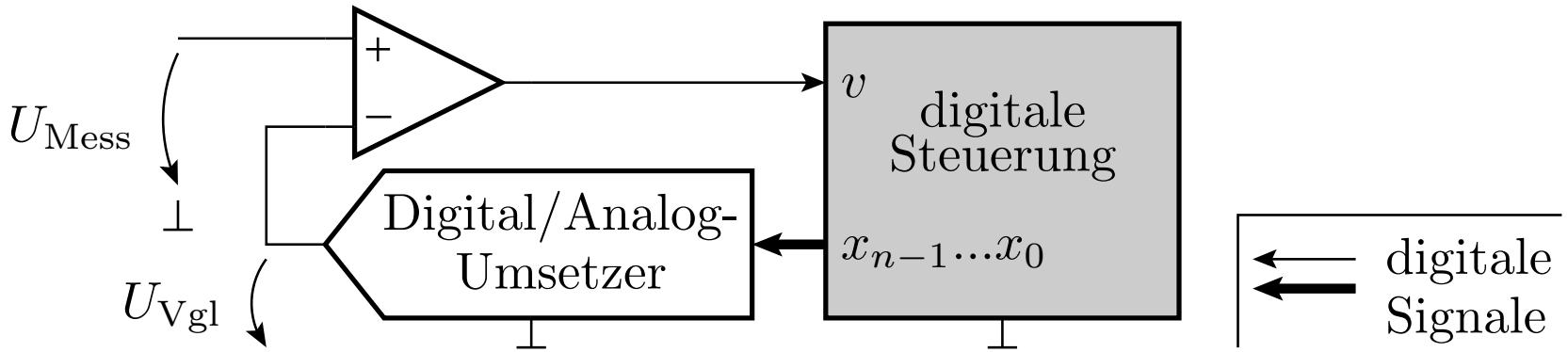
Watchdog

Watchdog

Jedes größere Programm enthält statisch gesehen Fehler, die unter anderem auch dazu führen können, dass das Programm abstürzt. Bei Absturz erreicht das Programm einen Zustand, den es nicht wieder von selbst verlassen kann, z.B. eine endlose Warteschleife.
Ein Watchdog ist ein Zeitzähler, der vom Programm in einem bestimmten Zeitabstand rückgesetzt werden muss. Bei Überlauf initialisiert er den Rechner neu. Begrenzt die Dauer der Nichtverfügbarkeit von Rechnern durch Ausfälle auf wenige ms bis s.
Zur Verhinderung, dass Programmierfehler den Watchdog deaktivieren, gibt es nur einen Befehl zum Einschalten des Watchdogs nach Programmstart und zum Rücksetzten des Zählers, aber keinen zum Ausschalten.

Analog-Digital-Wandler

Mehrschrittwandler



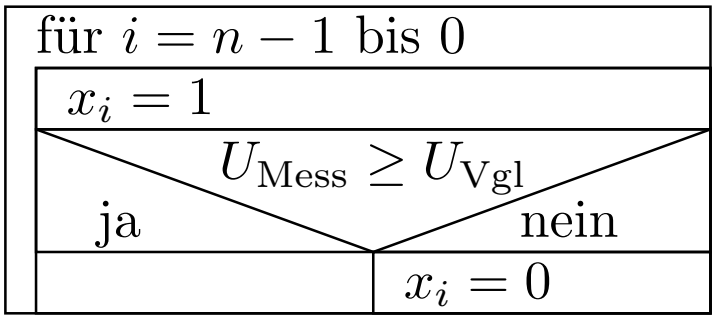
$$v = \begin{cases} 0 & \text{wenn } U_{\text{Mess}} < U_{\text{Vgl}} \\ 1 & \text{sonst} \end{cases}$$

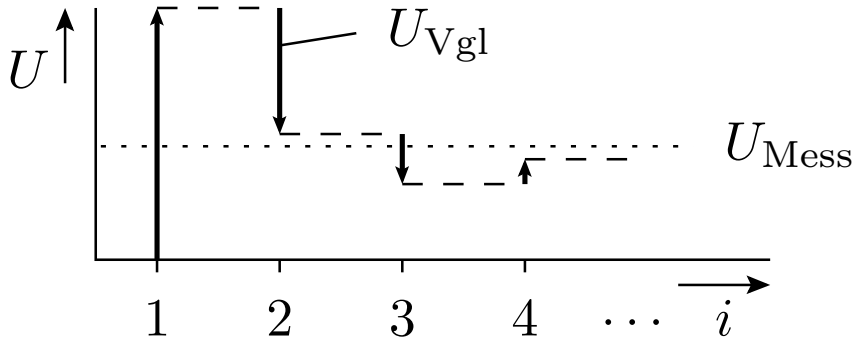
- Die Vergleichsspannung, die der DAU (Digital-Analog-Wandler) ausgibt:

$$U_{\text{Vgl}} = U_{\text{ref}} \cdot \frac{x}{2^n}$$

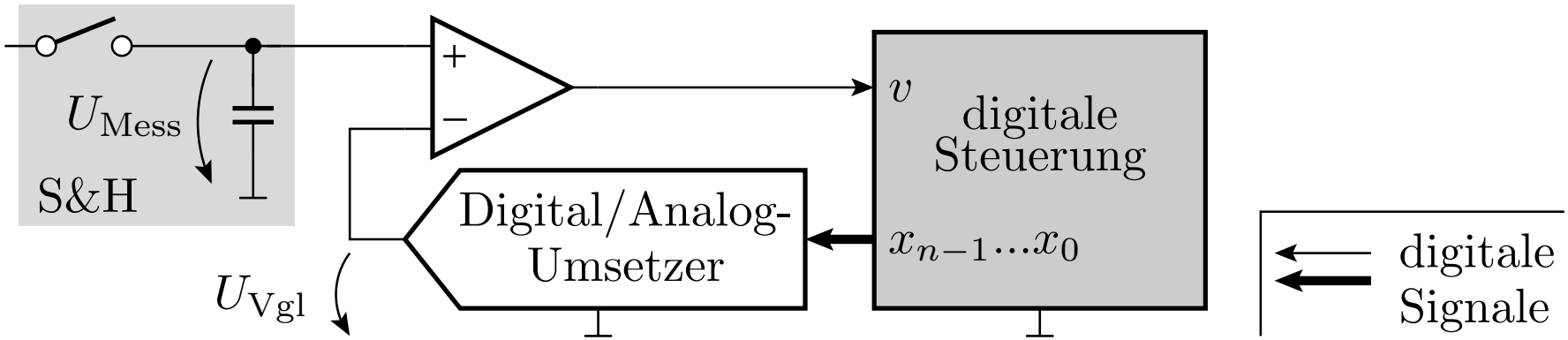
(U_{ref} – Referenzspannung; x – Ausgabewert (Bitvektor); n – Bitanzahl des Ausgabewerts).

Sukzessive Approximation





- Ein Vergleich je Bit. Schnellster serielle Wandlung.
- Der Messwert darf sich während der Wandlung nicht ändern.
  Erweiterung um eine Sample-and-Hold-Schaltung:



ADC des ATmega 2560

- 10 Bit-Wandler.
- Eine normale Wandlung benötigt 13 und die erste mit Initialisierung der Analogschaltung 25 10µs-Schritte.
- Mögliche Messeingänge sind die Anschlüsse ADCi (PF, PK)
- Programmierbar sind einmalige und kontinuierliche Messungen. Max. $\approx 10^4$ Messungen pro s.
- Bei mehreren Messeingängen erfolgen die Messungen nacheinander.
- Zur Minderung der Messfehler kann der Prozessor während der Wandlung in den Sleep-Modus versetzt werden (Abschalten des Prozessortakts).
- ...

Die Konfiguration erfolgt über die SFR-Register ADMUX, ADCSRA, ADCSRB, ...

Beispiel zur ADC-Ansteuerung

ADC-Initialisierung:

```
11  void initADC(){
12    ADMUX = (3<<MUX0); // Kanal 3 auswählen
13    // Einschalten mit Wandlungstaktteiler 64
14    ADCSRA = (1<<ADEN)|(0b110<<ADPS0);
15    DDRE &= ~0x80;  // Sensoreingang als Eingang
16    PORTE &=~0x80;  // Ausgabewert 0 (hochohmig)
17  }
```

- Der Sensor ist an ADC3 (PF3) (Kanal 3 auswählen).
- Der Wandlertakt als CPU-Takt durch Teilerwert

$$f_{\mathrm{ADC}} = \frac{f_{\mathrm{CPU}}}{64} \approx 117\,\mathrm{kHz}$$

- Um den Analogwert nicht zur verfälschen ist PF3 als Eingang mit Ausgabewert 0 (Pull-up aus) zu konfigurieren.

Funktion zur Rückgabe eines Analogwerts

```
19  uint32_t getADC(){
20    uint16_t wert;
21    ADCSRA |= (1<<ADSC); //Wandlung starten
22    // Warten bis "fertig"-Bit gesetzt ist
23    while(!(ADCSRA & (1<<ADIF))){}
24    // "fertig"-Bit durch Schreiben einer
25    ADCSRA |= (1<<ADIF); // 1 löschen
26    wert = ADC;
27    return wert;
28  }
```

- Wandlungsstart durch Setzen von ADSC in ADCSRA.
- Bei Wandlungsabschluss setzt der Prozessor ADIF=1.
- ADIF wird durch Schreiben einer Eins gelöscht.

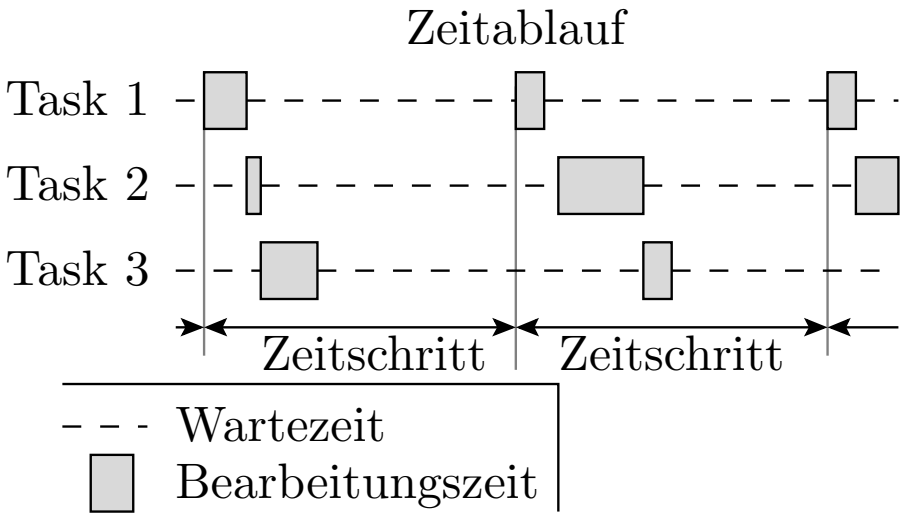
Interrupts

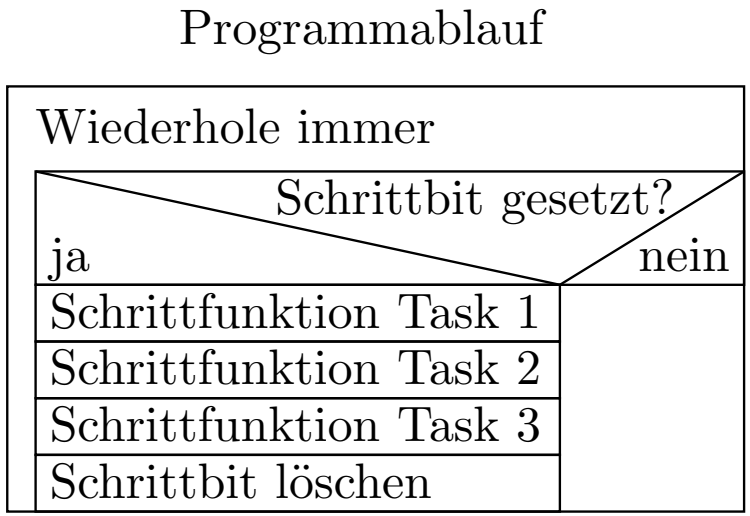
Quasizeitgleiche Abarbeitung mehrerer Tasks

Systeme mit Mikrocontrollern müssen oft mehrere Aufgaben (Tasks) quasi zeitgleich ausführen:

- eine LED blinken lassen,
- auf Zeichen warten, Sensorwerte abfragen, ...

Ein möglicher Programmablauf ist eine Endlosscheife, die zyklisch auf ein Timer-Bit wartet, dann für jeden Task einen Schritt abarbeitet und das Timer-Bit löscht.

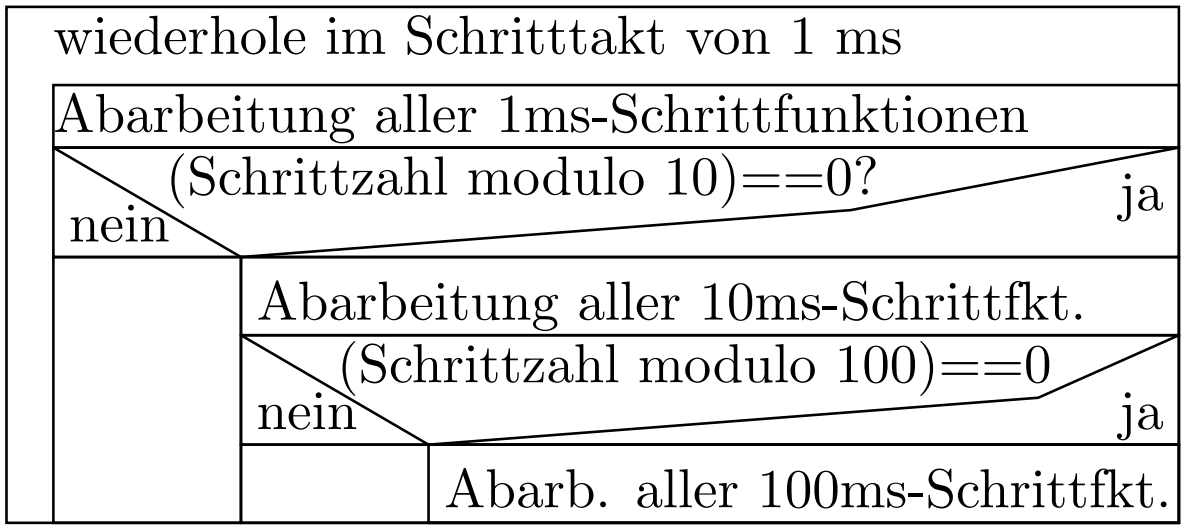




Aufgaben mit unterschiedlicher Schrittzeit

- Aktualisierung der Ausgabe: 100ms bis mehrere Sekunden.
- Reaktion auf Sensordaten: typ. 10 ms
- Abfrage, ob die USART ein Byte empfangen hat: $< 1\,\text{ms}$.

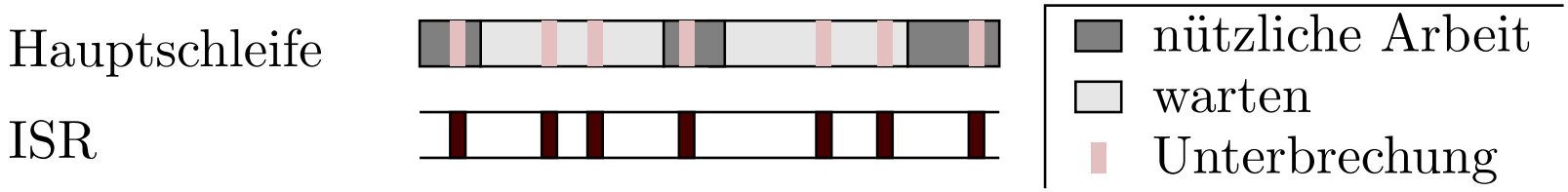
Programmtechnische Lösung:



In Summe dürfen die Bearbeitungszeiten aller Schrittfunktionen zusammen nicht länger als die kürzeste Schrittzeit sein.

Interrupts

Hardware-Funktionen zur ereignisgesteuerten Programmunterbrechung und Einfügung von Tasks zur Ereignisbehandlung:



Ein Interrupt-Behandlung (Unterbrechung):

- verlangt eine ISR (Interrupt-Service-Routine) auf einer hardware-codierten Programmadresse und
- muss explizit freigeschaltet werden.
- Die ISR wird durch Aktivierung eines Ereignisbits aufgerufen,
- endet mit einem Rücksprung zum unterbrochenen Programm
- und ist in der Regel selbst nicht unterbrechbar.

Interrupts ATmega2560

57 verschiedene Interrupts. Jeder hat eine Startadresse, ein Ereignisbit und ein Aktivierungsbit:

- (ausgewählte) Port-Eingänge: Pegeländerungen, ...
- Timer: Überlauf, Gleichheit mit Vergleichsregister, Datenübernehme im Zählmodus, ...
- USART: Senden fertig , Sendepuffer leer, Daten empfangen.
- SPI: Transferende. ADC: Wandlung beendet,
- EEPROM: Schreiboperation abgeschlossen, ...

Zur Nutzung von Interrupts ist

- der Header avr/interrupt.h einzubinden
- eine ISR zu definieren[4]:

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

[4] Eine ISR ist keine normale Funktion, sondern eine, die mit goto <interrupt-Vektor> aufgerufen wird, keine Übergabeparameter und keinen Rückgabewert hat, alle genutzten Register rettet, mit »reti« endet.

Beispiel für eine ISR

```
15  ISR(TIMER3_COMPA_vect){//ISR Timer 3 Vergleichs-
16    // interrupt B, Schrittbit OCF3A wird bei ISR-
17    // aufruf automatisch gelöscht
18    LED_Ct++;
19    if (LED_Ct>=500){ // alle 500 Schritte
20      PORTE ^= 0x10;  // LD4 invertieren
21      LED_Ct = 0;     // Zähler rücksetzen
      }                 // erzeugt 1Hz-Blinksignal
    }
```

- Timer 3 sei so eingestellt, dass er zyklisch jede Millisekunde das Ereignisbit OCF3A setzt.
- Die dann aufgerufene ISR invertiert in jedem 500sten Aufruf LD4 (Blinkausgabe 1 Hz).

- Das Hauptprogramm initialisiert den Ausgabeport, den Timer etc. schaltet Interrupts lokal und global ein und geht in eine Endlosschleife.

```
37  int main(void) {
38     DDRE = 0xF0;          // LD1 bis LD4 als Ausgänge
39    // Timer 3: Clear on Compare Match mit OCR3A,
40    TCCR3B = (1<<WGM32)|(0b001<<CS30); // Vorteiler 1
41    OCR3A  = 7580;        // 1 ms Aufrufperiode
42    // beim Zählen von 1 bis 7580 soll auch mit OCR3B
43    OCR3B  = 1000;        // einmal Gleichheit auftreten
44    LED_Ct = 0;
45    PORTE ^= 0x20;        // LD3 umschalten
46    ETIMSK = 1<<OCIE3A; // Timer3-Comp.B-Interrupt ein
47    sei();                // Interrupts global freigeben
48    while(1){
49  //  cli();              // Interrupts sperren
50  //  PORTE ^= 0x80;      // LD1 umschalten
51  //  sei();              // Interrupts wieder freigeben
      };
  }
```

Die Tücken bei der Nutzung von Interrupts

- Was passiert, wenn versehentlich ein Interrupt freigeschaltet und ausgelöst wird, für den keine ISR programmiert ist?
  - Was schreibt dann der Compiler auf die ISR-Startadresse für Befehle? ...
- Was passiert, wenn eine ISR gleiche Daten wie das unterbrochene Programm bearbeitet?
  - Wer ist schneller?
  - Wer überschreibt wessen Ergebnis? ...
- Was passiert, wenn eine ISR während ihrer Abarbeitung unterbrechbar ist?
  - Durch andere Ereignisse?
  - Durch dasselbe Ereignis?
- ...

- »Einkommentieren« Hauptprogrammzeile 50:

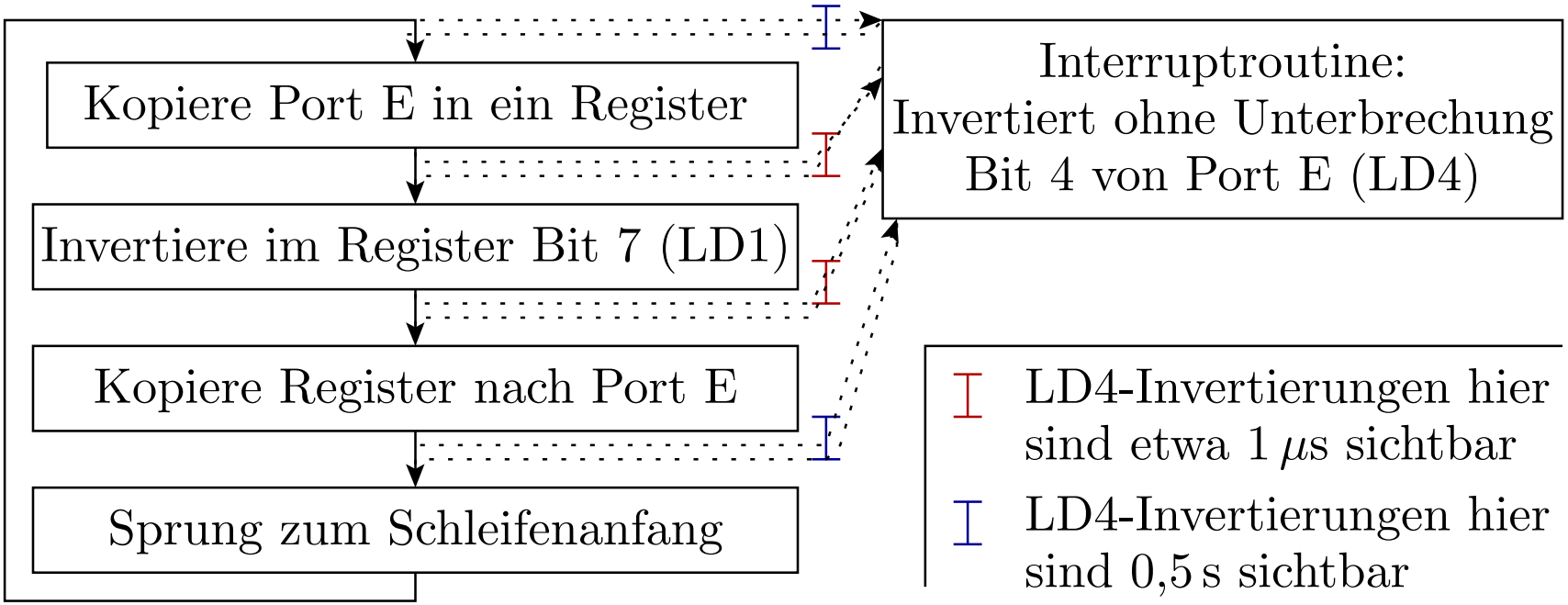
```
37  int main(void) {
    ...
48   while(1){
49  //  cli();              // Interrupts sperren
50      PORTE ^= 0x10;      // LD1 umschalten
51  //  sei();              // Interrupts wieder freigeben
    };
43  }
```

- Hauptprogramm und ISR ändern Port E. Was passiert?

```
15  ISR(TIMER3_COMPA_vect){
16    LED_Ct++;
17    if (LED_Ct>=500){ // alle 500 Schritte
18      PORTE ^= 0x10;  // LD4 invertieren
19      LED_Ct = 0;     // Zähler rücksetzen
      }
20  }                   // erzeugt 1Hz-Blinksignal
```

Das Blinken von LD4 wird ungleichmäßig.

Das Hauptprogramm und die ISR ändern beide den Wert von Port E. Die c-Anweisung zum invertieren von LD1 besteht aus drei Maschinenanweisungen. Plus Sprung zum Schleifenanfang umfasst der Schleifenkörper vier Anweisungen:



An 50% der Unterbrechungsmöglichkeiten wird die Invertierung von LD4 in der ISR vom Rückschreibwert für die Invertierung von LD1 überschrieben.

Unterbrechungsfreie Sequenzen

Sperren der Interrupts während der Invertierung von LD1:

```
48    while(1){
49      cli();              // Interrupts sperren
50      PORTE ^= 0x80;      // LD1 umschalten
51      sei();              // Interrupts wieder freigeben
52    };
```

Damit kann der Prozessor das Programm vom Lesen von Port E bis zum Schreiben des für LD1 invertierten Wertes nicht unterbrechen. Unterbrechungen vor und nach dem Sprung zum Schleifenbeginn verursachen keine Fehlfunktionen.

Fakt 1

Bei Interrupts sollte man wissen, was man tut, gewissenhaft programmieren und die Dokumentation gründlich lesen!