Rechnerarchitektur, Foliensatz 5
Parallele und serielle Schnittstellen

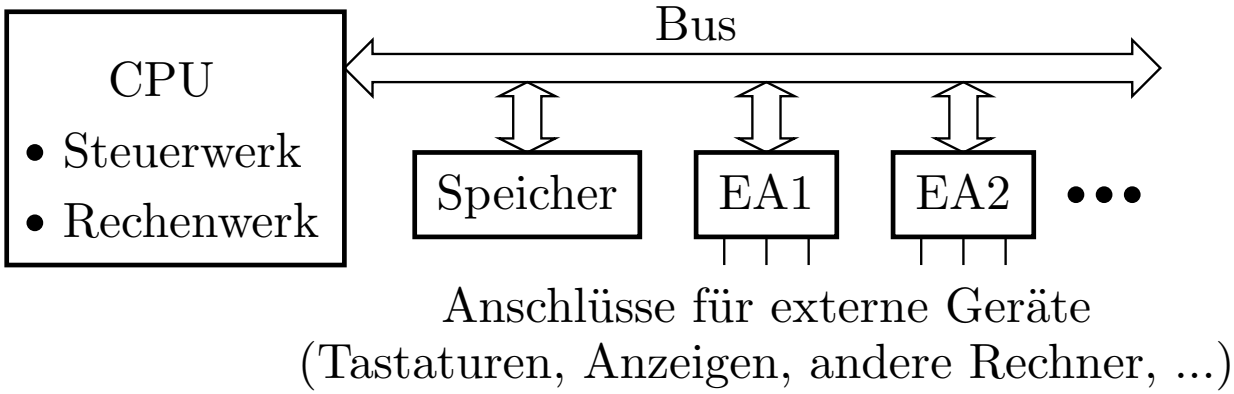
G. Kemnitz

Institut für Informatik, TU Clausthal (RA-F5.pdf)

30. Januar 2020

Ports

Prinzip der Ein- und Ausgabe



Ein Prozessor kommuniziert mit seiner Umgebung

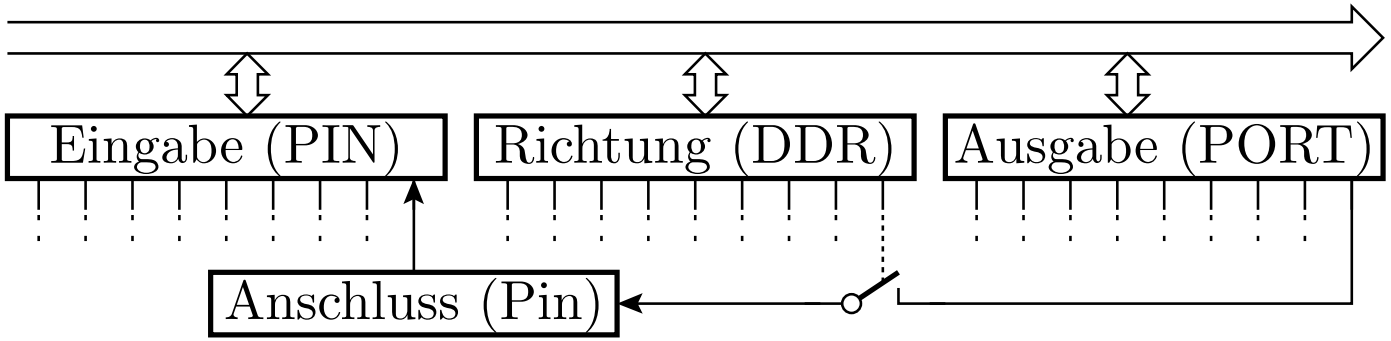
- Benutzer, Sensoren, Aktoren,
- getrennten Werken (Timer, Watchdog, ...),
- anderen Rechnern, ...

über EA-Register.

Im einfachten Fall sind die EA-Register Parallele Schnittstellen, d.h. binäre Ausgabesignale z.B. zum Schalten von Anzeigen, Motoren und Eingabesignale, z.B. zum Lesen von Schalter und Sensorwerte.

Ports des ATmega2560

Ports des ATmega2560



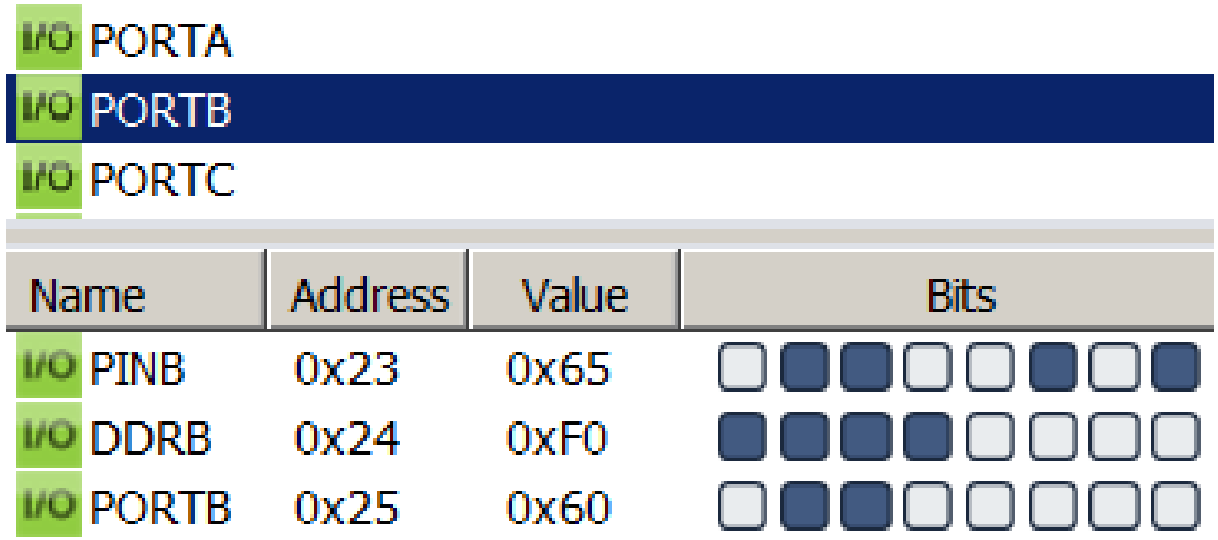
- Ports (Parallele Schnittstellen) sind 8-Bit-, bei größeren Prozessoren auch 16- oder 32-Bit-IO-Register mit anschließbaren Leitungen z.B. für Schalter und LEDs.
- Universelle Ports können bitweise als Eingänge, Ausgänge oder mit umschaltbarer Übertragungsrichtung konfiguriert werden.
- Bei AVR-Prozessoren gehören zu jedem Port 3 Register mit aufeinanderfolgenden Adressen, PINx für die Eingabe, DDRx für die Übertragungsrichtung und PORTx für die Ausgabe.

Port-Adressen und Darstellung im Debugger

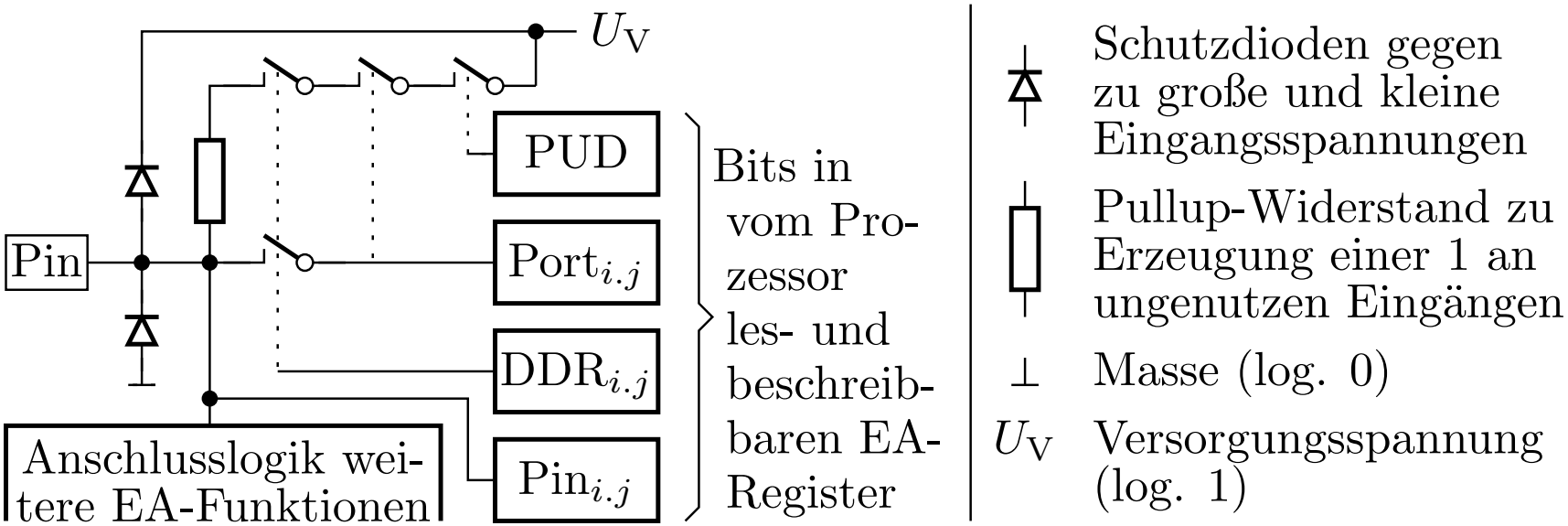
Der ATmega2560 hat 12 Ports mit je Eingaberegister PINx, Richtungsregister DDRx und Ausgaberegister PORTx ($x \in \{\mathrm{A}, \mathrm{B}, ..., \mathrm{L}\}$; 0/0x20 – IO-Adresse / Datenspeicheradresse).

x	A	B	C	D	E	•
PINx	0 / 0x20	3 / 0x23	6 / 0x26	9 / 0x29	0xC / 0x2C	•
DDRx	1 / 0x21	4 / 0x24	7 / 0x27	0xA / 0x2A	0xD / 0x2D	•
PORTx	2 / 0x22	5 / 0x25	8 / 0x28	0xB / 0x2B	0xE / 0x2E	•

Debug-Ansicht:

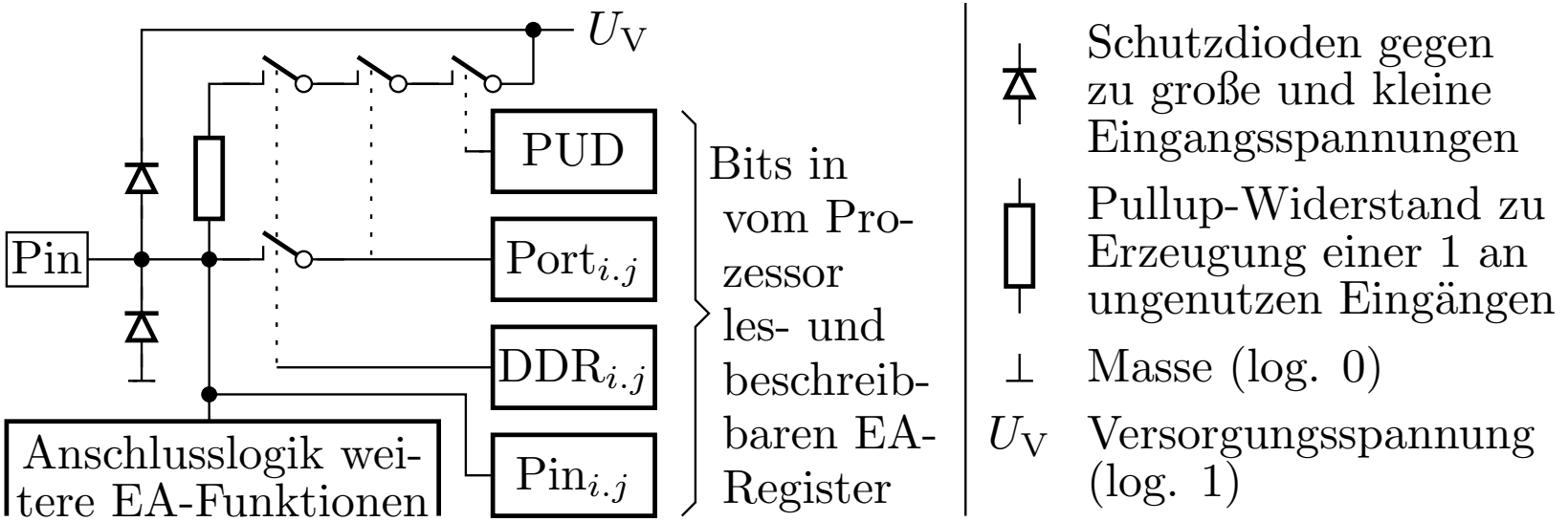


Beschaltung und Nutzung eines einzelnen IO-Pins



Nutzung als Ausgang:

- Richtungsbit eins setzen: $\mathrm{DDR}_{i.j} := 1$.
- Ausgabe: $\mathrm{PORT}_{i.j} :=$ Rd (Rd – Arbeitsregister)
- Rücklesen des Ausgabewertes: $\mathrm{Rd} := \mathrm{PORT}_{i.j}$
- Eingabe: $\mathrm{Rd} := \mathrm{PIN}_{i.j}$. $\mathrm{PIN}_{i.j} \neq \mathrm{PORT}_{i.j}$ ist möglich und deutet auf Programmier- oder Schaltungsfehler.



Nutzung als Eingang:

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

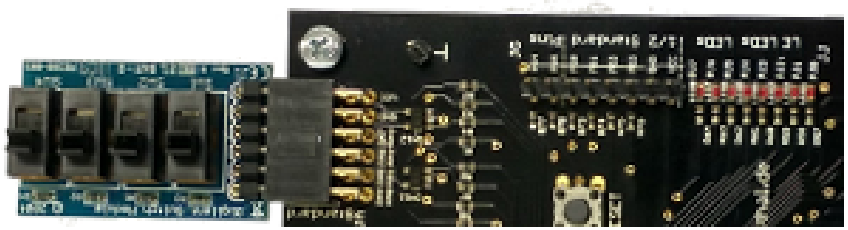
Polling

Polling und Interrupt

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

- Polling: Zyklische Abfrage aller EA-Geräte durch den Rechner, ob Datenübergabe angefordert oder zur Übernahme bereit. Wenn ja, Verzweigung zum Programmbaustein für den Datenaustausch (Software-Funktion).
- Interrupt: Gerät fordert Datenaustausch an. Rechner ruft, sobald er dafür bereit ist, eine **I**nterrupt-**S**ervice-**R**outine (ISR) auf. Erfordert Hardware-Unterstützung, siehe Foliensatz RA-F6.pdf).

Warten auf sequentielle Ereignisse



```
void main(){
 DDRA = 0xFF;                    // Tastereingänge
 DDRJ = 0;                       // LED-Ausgabe
 while (1){                      // Wiederhole immer
  while (!(PINA&1));PORTJ|=1;//warte bis SW1 ein
  while (PINA&1); PORTJ&=~1;//warte bis SW1 aus
  while (!(PINA&2));PORTJ|=2;//warte bis SW2 ein
  while (PINA&2); PORTJ&=~2;//warte bis SW2 aus
 }
}
```

Programm wartet immer auf ein Ereignis nach dem anderen.

Warten auf nebenläufige Ereignisse

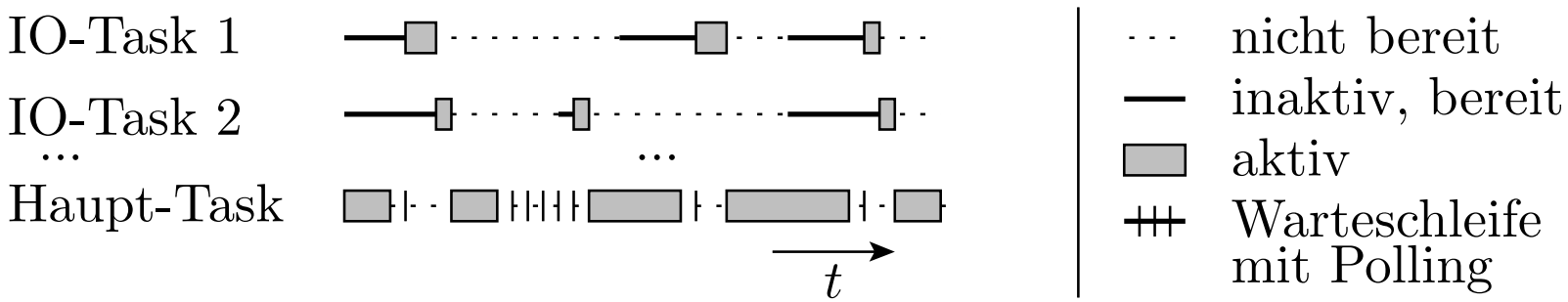
```
void main(){
 DDRA = 0xFF;                  // Tastereingänge
 DDRJ = 0;                     // LED-Ausgabe
 while (1){                    // Wiederhole immer
  if    (PINA&1) PORTJ|= 1; //wenn SW1 ein: LD1 ein
  if (!(PINA&1))PORTJ&=~1; //wenn SW1 aus: LD1 aus
  if    (PINA&2) PORTJ|= 2; //wenn SW2 ein: LD2 ein
  if (!(PINA&2))PORTJ&=~2; //wenn SW2 aus: LD2 aus
 }
}
```

Reihumabfrage der Ereignisbits. Wenn Ereignis (Schalterwert) eingetreten, zugeordnete Aktion ausführen (hier LED ein- oder ausschalten).

Bedienung mehrer EA-Geräte

Ein EA-Gerät verlangt meist erst nach vielen Tausend Takten die nächste Aktion in Form eines Datenaustauschens von wenigen Bytes.

Software-Ablauf mit einem Haupt-Task und mehreren IO-Tasks zur Bedienung von EA-Einheiten über Polling:



Wiederhole immer

- Haupt-Task arbeitet einige tausend Befehle ab.
- Abfrage aller genutzten EA-Geräte auf Bereitschaft
- für Geräte die bereit sind, Einschieben eines IO-Tasks.

Programmstruktur für EA mit Polling

```
int main(){
  <Initialisierung, Variablen, ...>
  while(1){                    //Endlosschleife
    if (<Haupt-Task bereit>)
      {<Haupt-Task weiter abarbeiten>}
    if (<IO-Task 1 bereit>)
      {<IO-Task 1 abarbeiten>}
    if (<IO-Task 2 bereit>)
      {<IO-Task 2 abarbeiten>}
    ...
  }
}
```

- Der Haupt-Task muss sich nach hinreichend kurzer Zeit für mindesten einen IO-Abfragezyklus unterbrechen.
- IO-Tasks max. wenige hundert abzuarbeitende Befehle.
- Keine Warteschleifen außer der Endlosschleife.

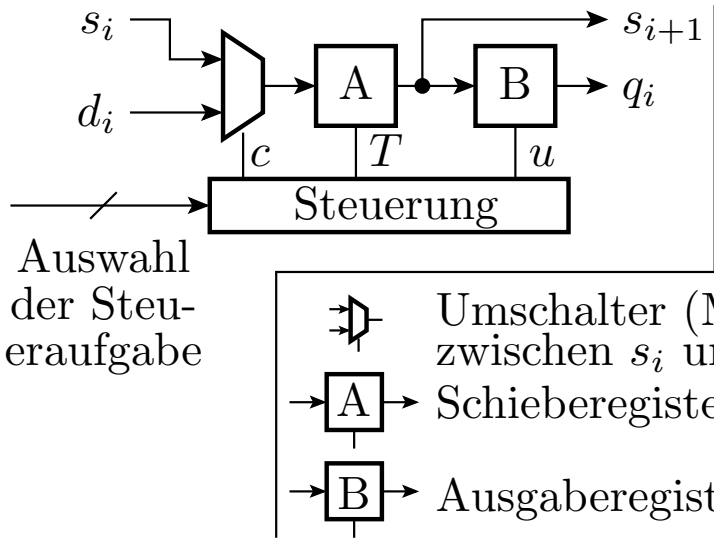
Serielle Schnittstellen

Serieller Datenaustausch

Der Datenaustausch zwischen Rechnern erfolgt in der Regel seriell[1].
Grundbaustein Schieberegister mit den Funktionen

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

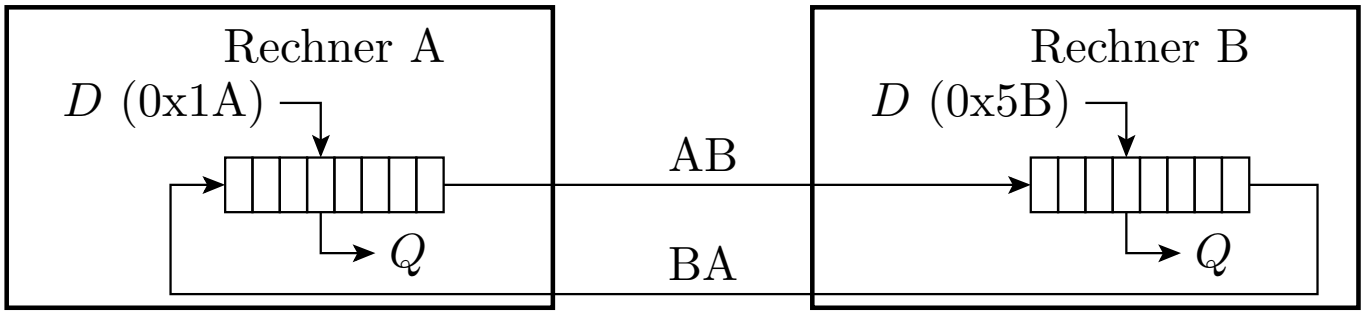
Schaltung einer Schieberegisterzelle:

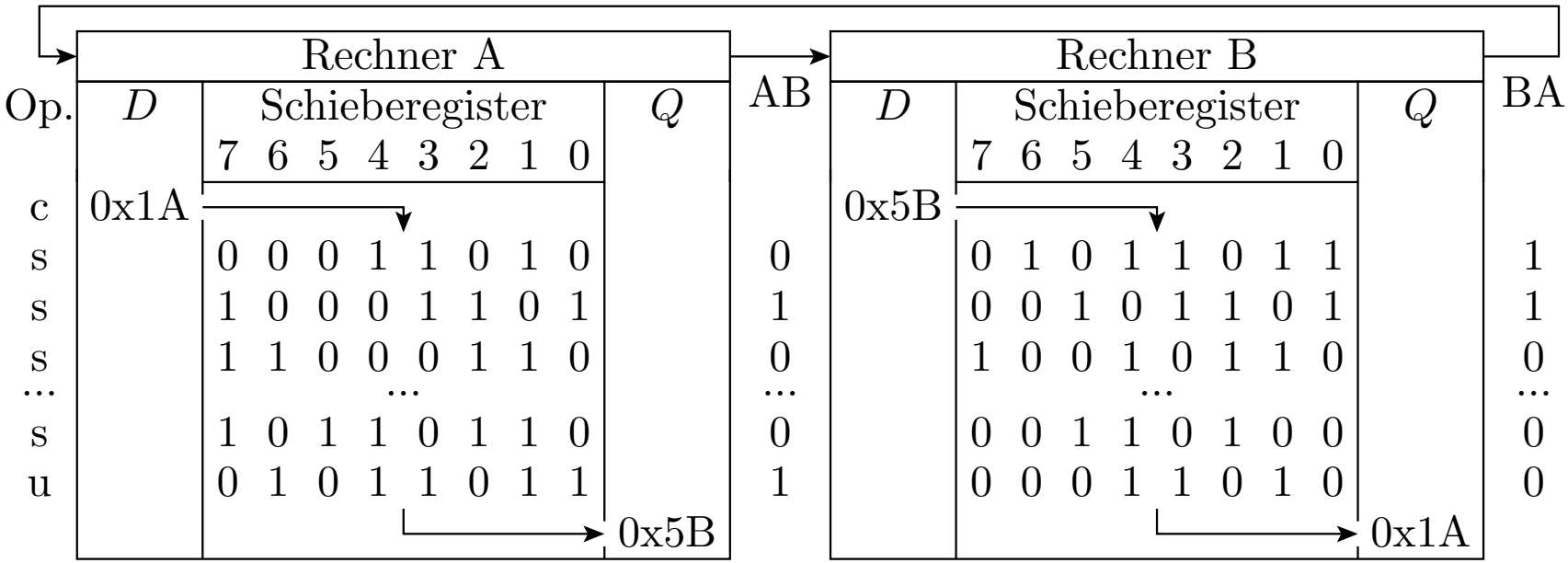


Steueraufgabe	c	T	u	Operation
Schieben	0	$\int$	0	$s_{i+1} \leftarrow s_i$
Übernahme	1	$\int$	0	$s_{i+1} \leftarrow d_i$
Übergabe	–	0	$\int$	$q_i \leftarrow s_{i+1}$
keine	–	0	0	——

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

Bidirektionale Kopplung zweier Rechner



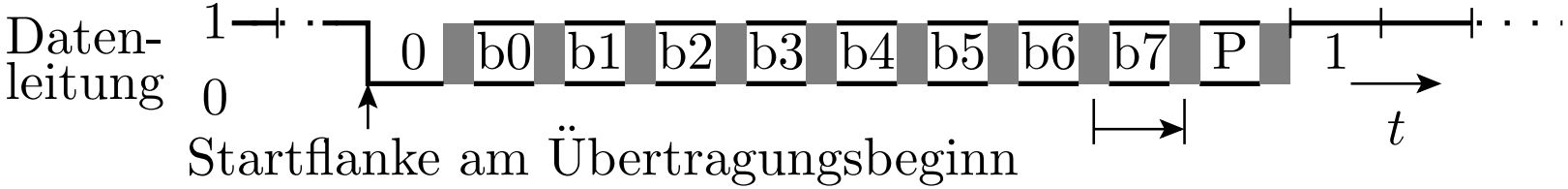


c Übernahme (Capture) s Schieben (Shift) u Übergabe (Update)

USART

USART (**U**niversal **S**ynchronous or **A**synchronous **R**eceiver and **T**ransmitter)

Übertragung ohne Takt und Steuersignale.



$b \in \{0,1\}$	Datenbits	1	Stoppbit, Übertragungspause
$P \in \{0,1\}$	Paritätsbit	$\longmapsto$	Bitzeit, z.B. $t_{\mathrm{Bit}} \approx 0{,}1\,\mathrm{ms}$
0	Stoppbit		Übertragungsdauer: $12 \cdot t_{\mathrm{Bit}}$

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.

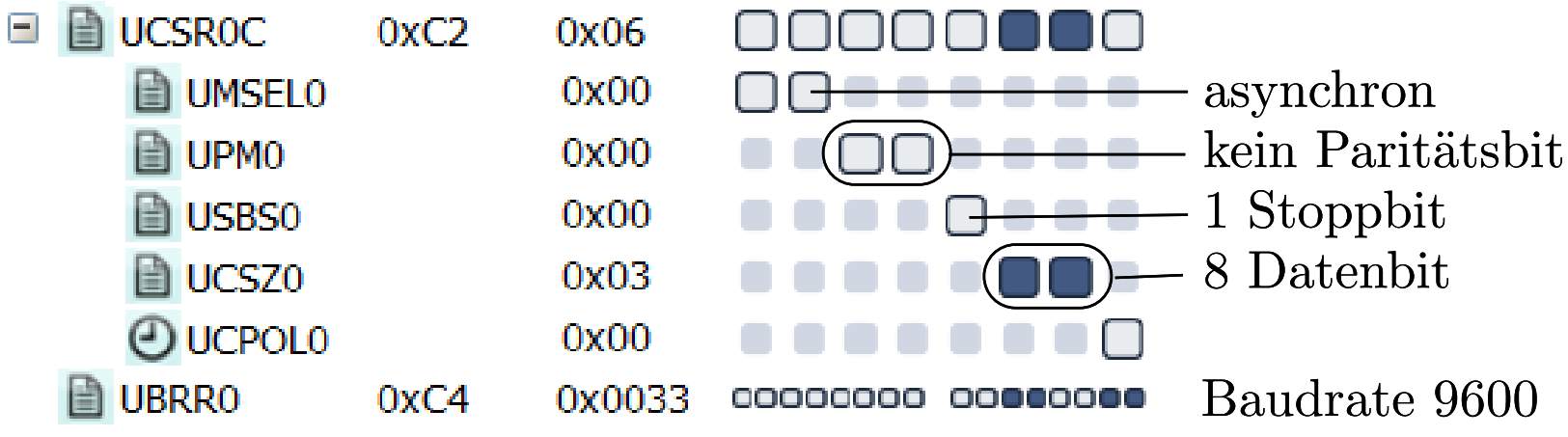
Baudrate b, Masseinheit Bd (Baud), ergibt sich aus der Prozessortaktfrequenz f_{Proz} und einem programmierten Teilerfaktor T:

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

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

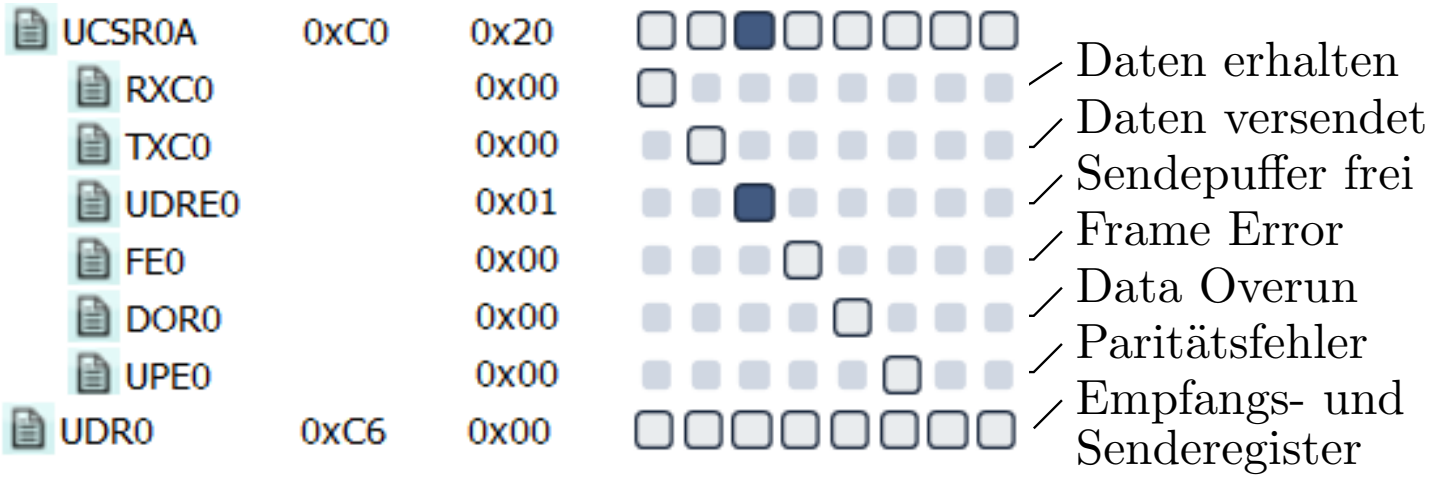
$$T = \frac{8\,\mathrm{MHz}}{16 \cdot 9600\,\mathrm{Bd}} - 1 = 51 = 0\mathrm{x}33$$

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




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

Senden und Empfang



Funktionen für den Empfang und das Versenden eines Bytes:

```
uint8_t get_byte(){
 while(!(UCSR0A&(1<<RXC0))); //warte auf Empfang
 return UDR0;                //Rückgabe Empfangsbyte
}

void send_byte(uint8_t dat){
 while(!(UCSR0A&(1<<UDRE0)));//warte bis Sendepuffer
 UDR0 = dat;                 //frei. Byte versenden
}
```

Senden und Empfang blockierungsfrei

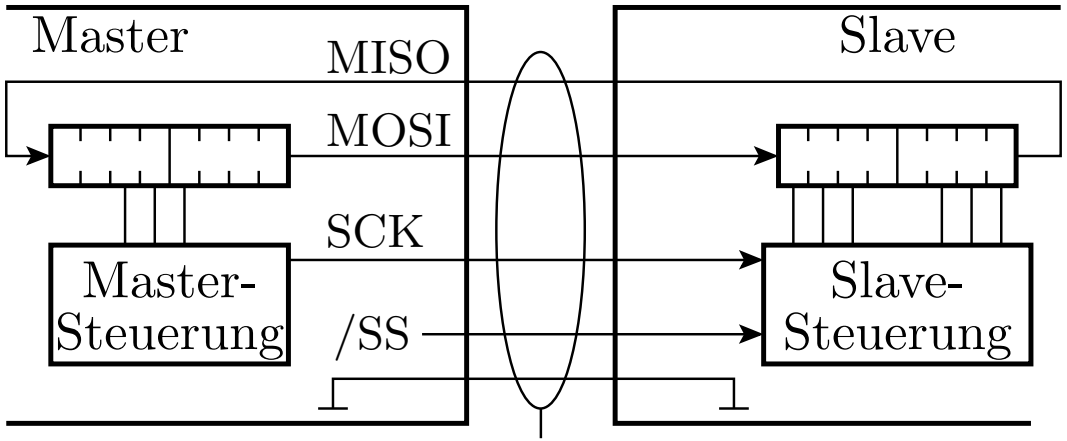
```
uint8_t get_byte_nb(uint8_t *dat){
 if (UCSR0A & (1<<RXC0)){ //wenn Byte empfangen
  *dat = UDR0;            //speichere Empfangsbyte
  return 1;               //Rückgabewert "wahr"
 }
 else return 0;           //sonst Rückgabe "falsch"
}

uint8_t send_byte_nb(uint8_t dat){
 if (UCSR0A & (1<<UDRE0)){//wenn Sendepuffer frei
  UDR0 = dat;             //Daten senden
  return 1;               //Rückgabewert "wahr"
 }
 else return 0;           //sonst Rückgabe "falsch"
}
```

- Funktionsergebnis: Operation ausgeführt wahr/falsch.
- Datenrückgabe über Zeiger.

SPI-Bus

SPI-Bus

Serieller Bus zur Vernetzung von Schaltkreisen.

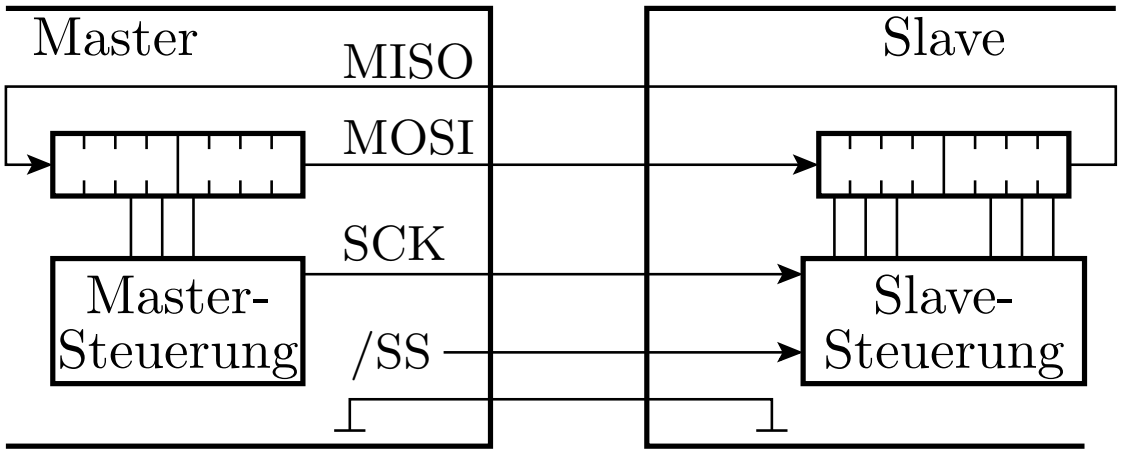


Leitungen zwischen den Schaltkreisen

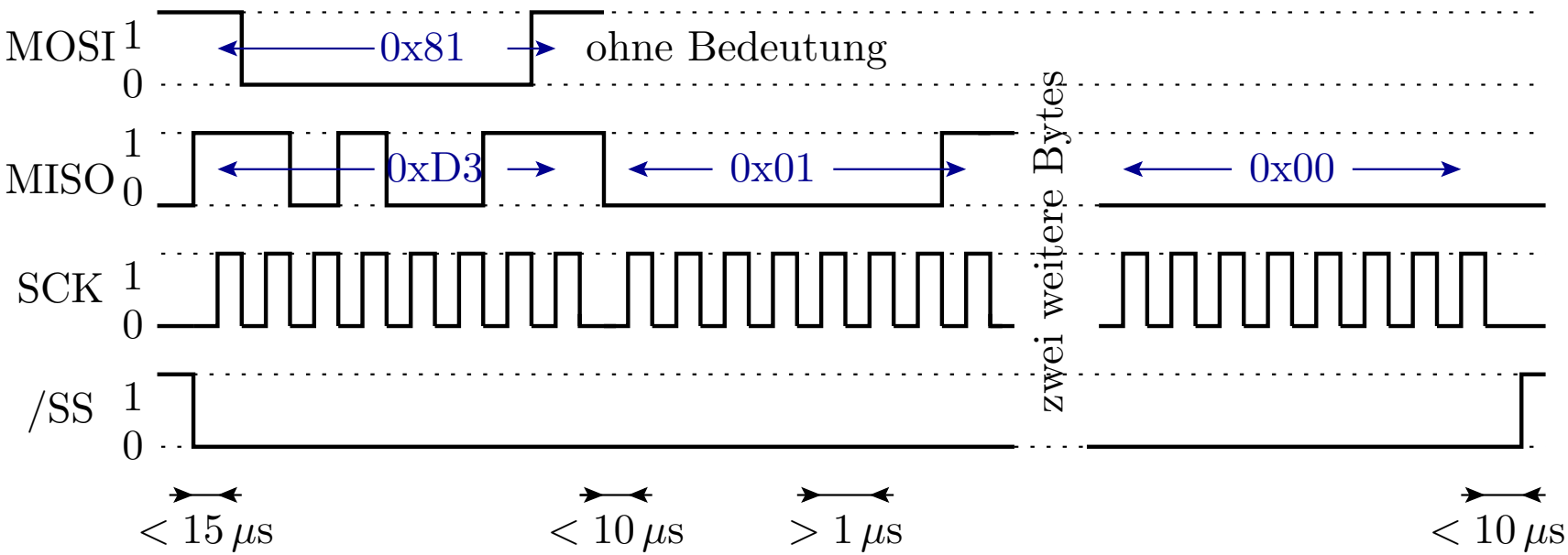
MOSI **M**aster **O**ut **S**lave **I**n
MISO **M**aster **I**n **S**lave **O**ut
SCK Schiebetakt
/SS Slave-Auswahl

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

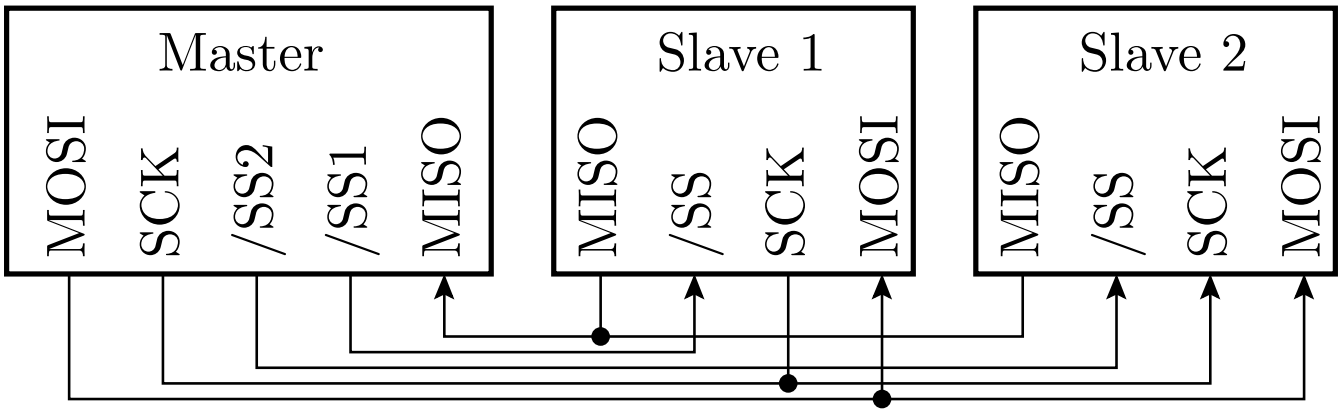
- Ein Schaltkreis ist der Master, der den Takt SCK und die Slave-Auswahlsignale erzeugt, die anderen sind Slaves, die diese Signale vom Master erhalten.



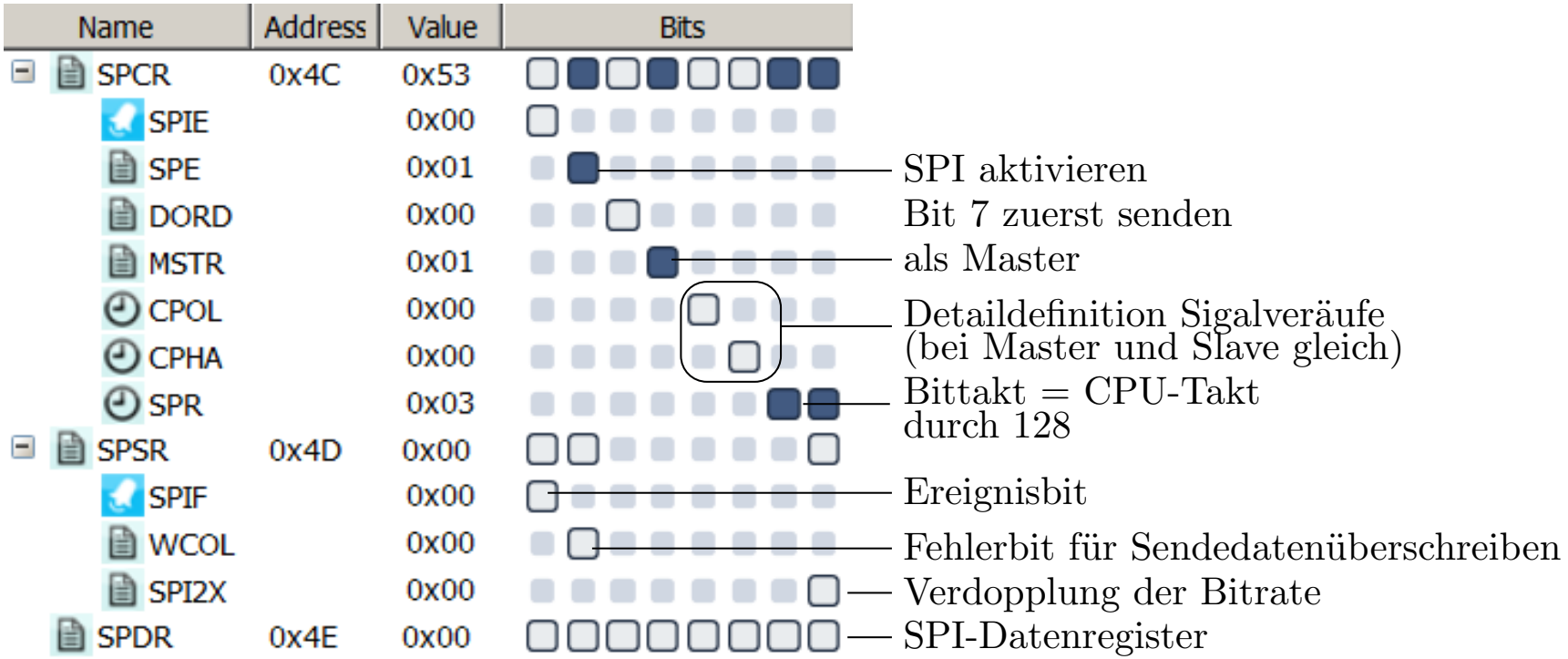
Beispiel für die Übertragung 0x81 vom Master zum Slave und von 0xD301...00 vom Slave zum Master:



- Eine Übertragung beginnt mit Aktivierung von /SS=0 (Übernahme), gefolgt von n Schiebetakten und endet mit Deaktivierung /SS=1.
- Die /SS-Signale des Masters werden über Ausgänge paralleler Schnittstellen ausgegeben.
- Ein Master kann mehrere Slaves mit unterschiedlichen /SS-Signalen angesteuern.



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.

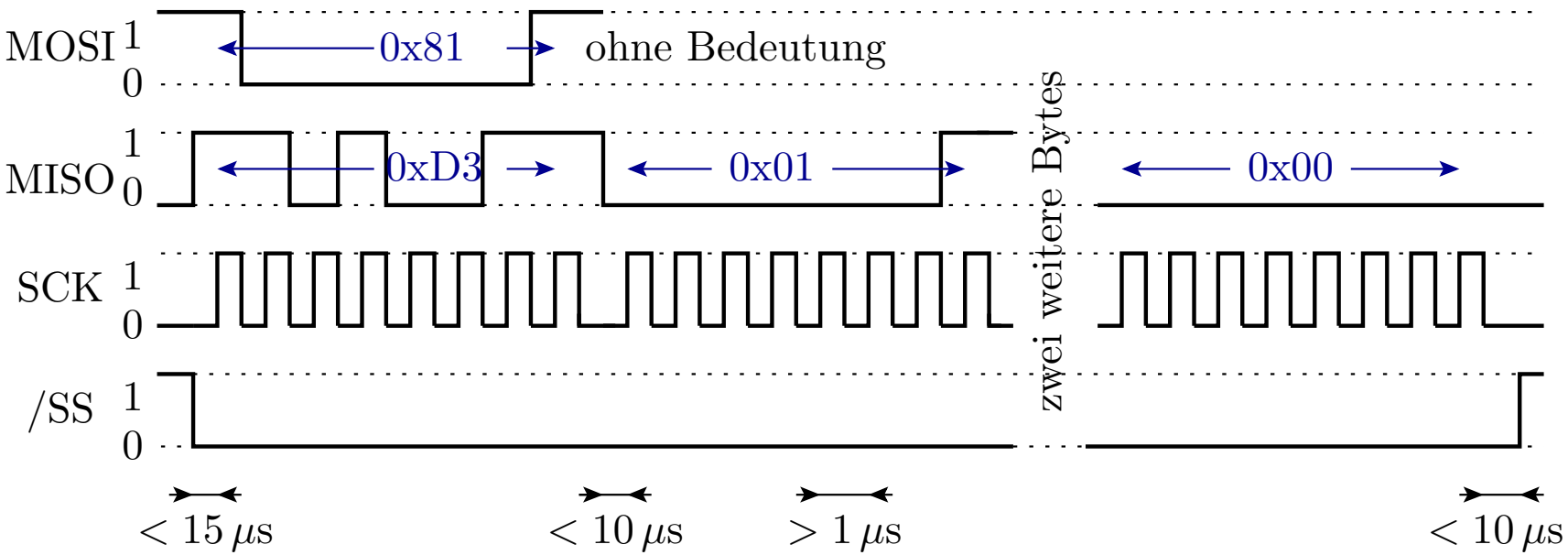
SPI-Datenaustausch blockierend

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

Funktionen für den blockierenden Austausch einer 4 Byte-Nachricht mit »/SS« an Port B Anschluss 0:

```
void spi_send_get_4byte(uint8_t *dat){
 PORTB &= ~1;                    // /SS = 0
 SPDR = dat[0];                  //1. Byte Senden
 for (idx=0;idx<4;idx++){        //wiederhole 4x
  while (!(SPSR&1<<SPIF));       //Warte Senden fertig
  dat[idx] = SPDR;               //Empf-dat. speichern
  if (idx<4) SPDR = dat[idx+1];
 }                                //erste 3x Folgebyte senden
 PORTB |= 1;
}
```

Schnittstellensignalverläufe:



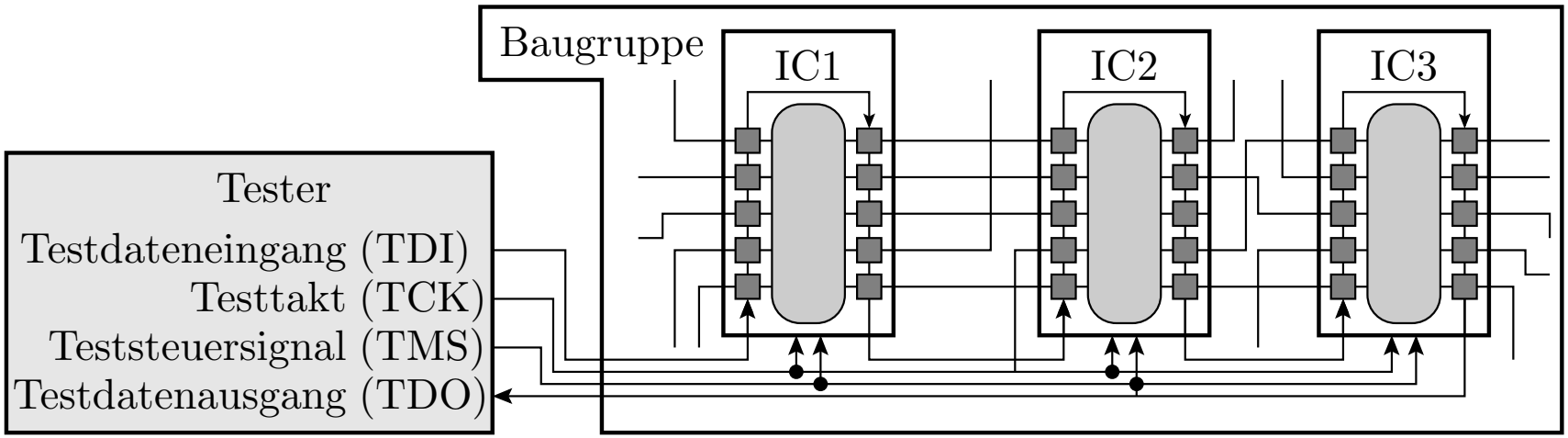
Eine nicht blockierender Datenaustausch verlangt eine Schrittfunktion:

- die im Zeitabstand kleiner 10 µs aufzurufen ist,
- »SPIF« (Bytetransfer fertig) abfragt,
- wenn »fertig«, das Empfangsbyte in einen Puffer schreibt und das nächste Sendebyte verschickt bzw »/SS« deaktiviert.

JTAG (Testbus)

JTAG (Boundary Scan[3])

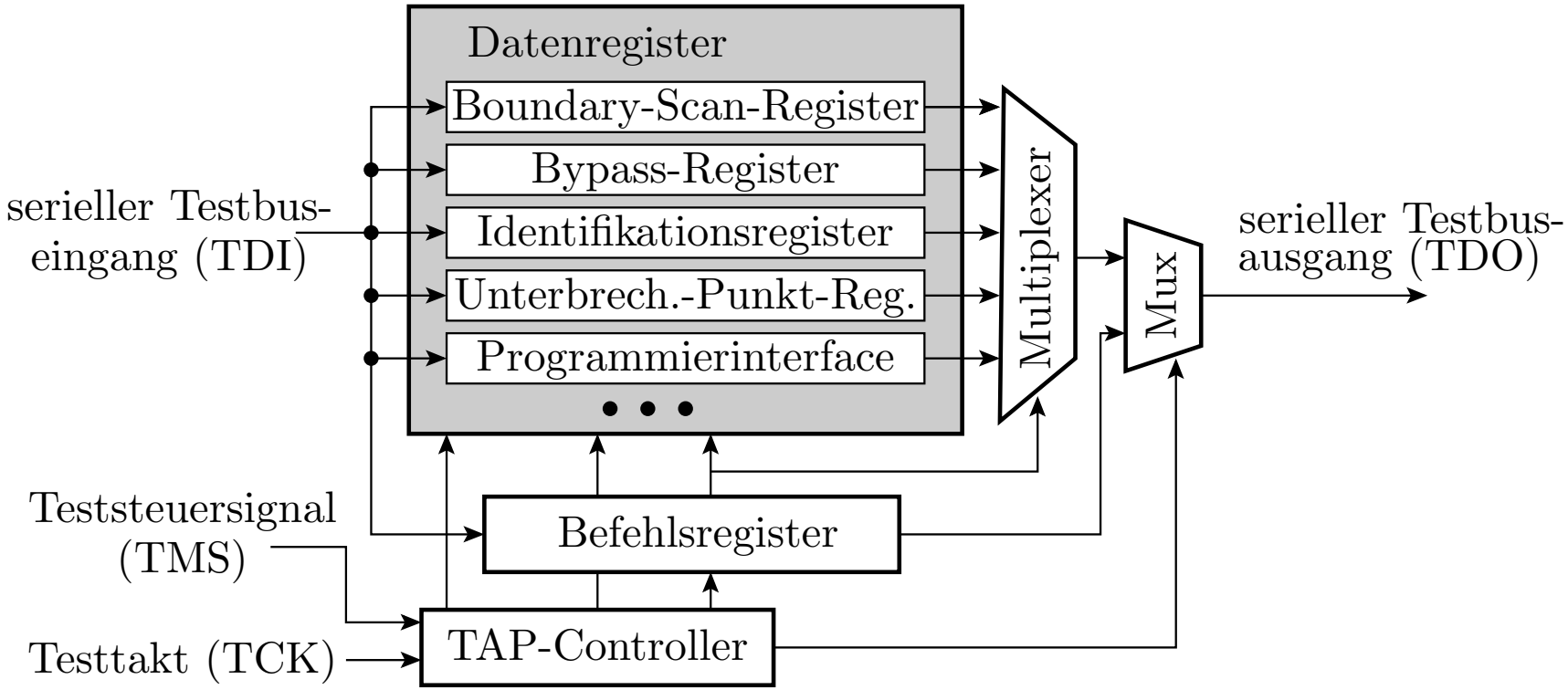
Test-, Diagnose-, Debug- und Programmierbus.



- Verschaltung aller Schaltkreise zu einer Kette.
- Serieller Datenaustausch mit einem Steuergerät[2].

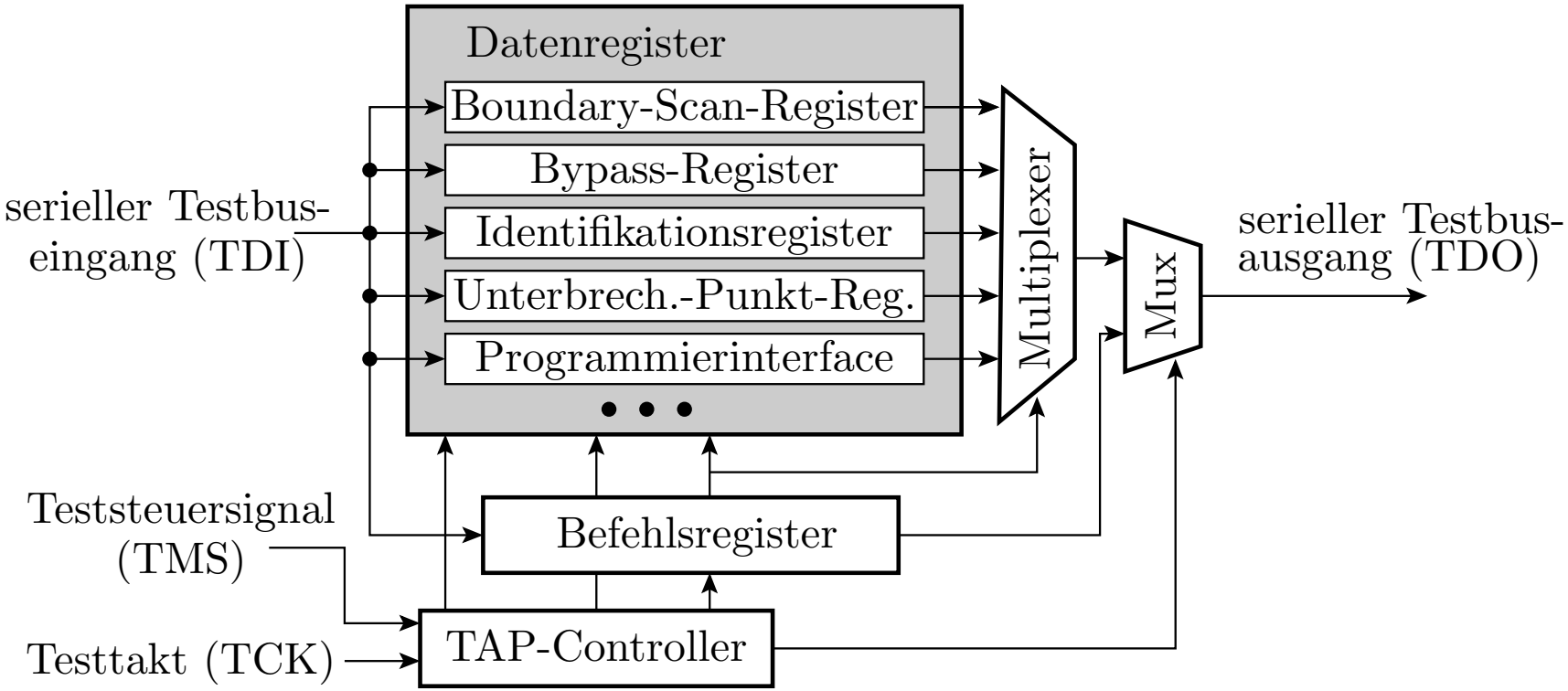
[2] In den Laborübungen der über USB mit dem PC verbundene »Dragon«.

[3] Ursprungsidee: Schieberegisterring mit den Funktionen Übernahme, Schieben und Übergabe am Schaltkreisrand zum Verbindungstest zwischen Schaltkreisen ohne mechanische Kontaktierung. Heute auch genutzt zum Lesen und Schreiben beliebiger Daten zum Programmieren, Debuggen, ...).



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

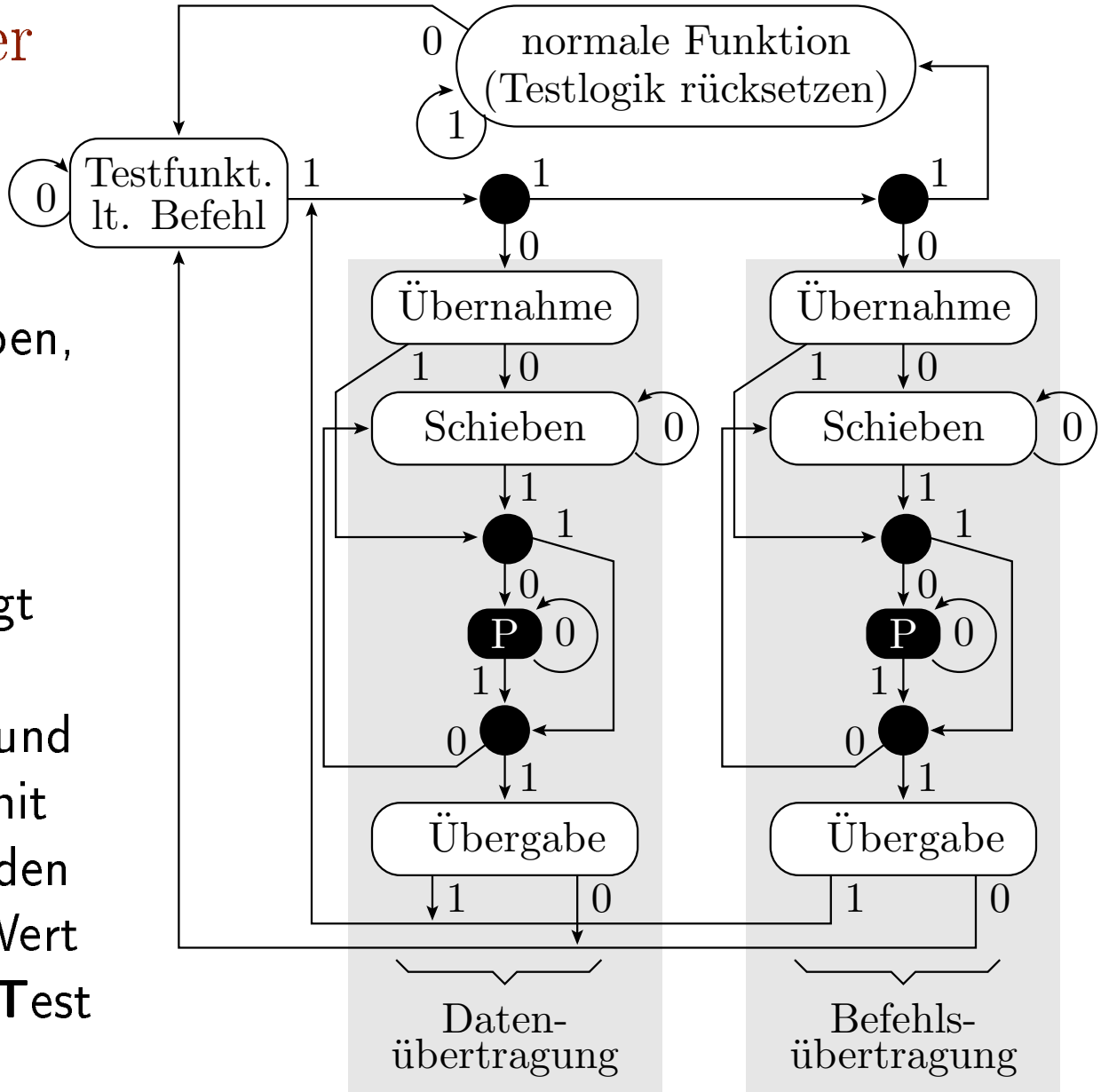
- Bypass-Register zur Verkürzung der Länge des Schieberegisters durch den Schaltkreis auf 1 Bit,
- Identifikationsregister mit Hersteller- und Bauteilnummer,



- Programmier-Interface: Schnittstelle zum Lesen und Schreiben des Befehls-Flashes, des Daten-EEPROMs und der Fuse-Register.
- Schnittstellenregister zum OCD (**O**n-**C**hip **D**ebugger), ...

TAP-Controller

Die Auswahl der 6 Busaktionen: Übernahme, Schieben, Übergabe für das Befehls- und das ausgewählte Datenregister erfolgt über ein 1-Bit-Steuersignal und einen Automaten mit 16 Zuständen. An den Kanten steht der Wert des Signals TMS (**T**est **M**ode **S**elect).



Von der JTAG-Implementierung in unserem Prozessor sind nur die standardisierten Testfunktionen, die für den Bestückungstest von Baugruppen vorgesehen sind, veröffentlicht. Die Befehle für die Programmierung und den OCD (On-Chip Debugger) fehlen.

Analoge Eingabe

Messung und Überwachung von analogen Werten

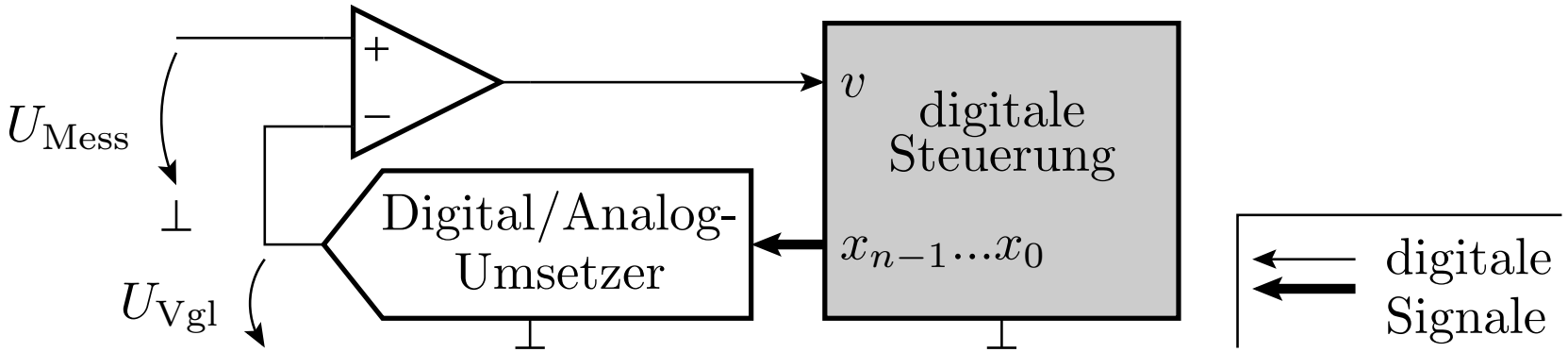
Analog-Digital-Wandler:

- Umwandlung einer analogen Eingangsspannung oder Eingangsspannungsdifferenz in einen Bitvektor (beim ATmega 2560 10 Bit).
- Wandlungsdauer 13 bis 25 Takte mit einer Taktperiode $t_{\mathrm{AClk}} \geq 1\mu\mathrm{s}$. Gesamte Wandlungsdauer $\geq 13 \ldots 25\mu\mathrm{s}$.
- Über einen programmierbaren Eingabemultiplexer kann zwischen unterschiedlichen Signalquellen ausgewählt werden.

Analog-Komparator:

- Vergleich zweier analoger Eingangsspannungen.
- Das 1-Bit-Vergleichergebnis kann programmgesteuert ausgewertet werden (Polling) oder Interrupts auslösen (siehe später Foliensatz RA-F6.pdf).

Prinzip eines seriellen Analog-Digital-Wandlers



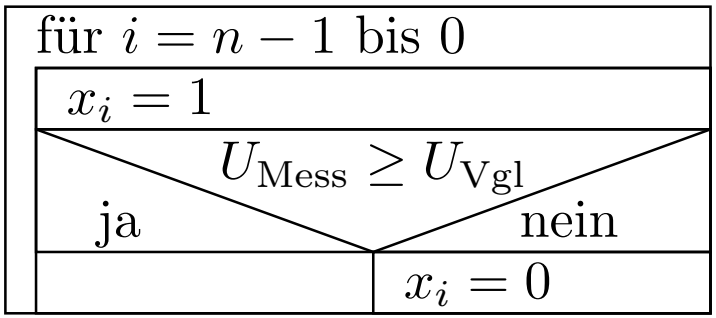
$$v = \begin{cases} 0 & \text{wenn } U_{\text{Mess}} < U_{\text{Vgl}} \\ 1 & \text{sonst} \end{cases}$$

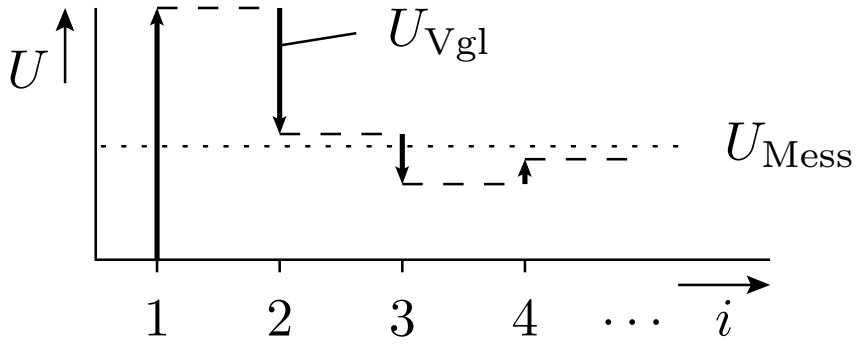
Die Vergleichsspannung, die der DAU (**D**igital-**A**nalog-**U**msetzer) ausgibt:

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

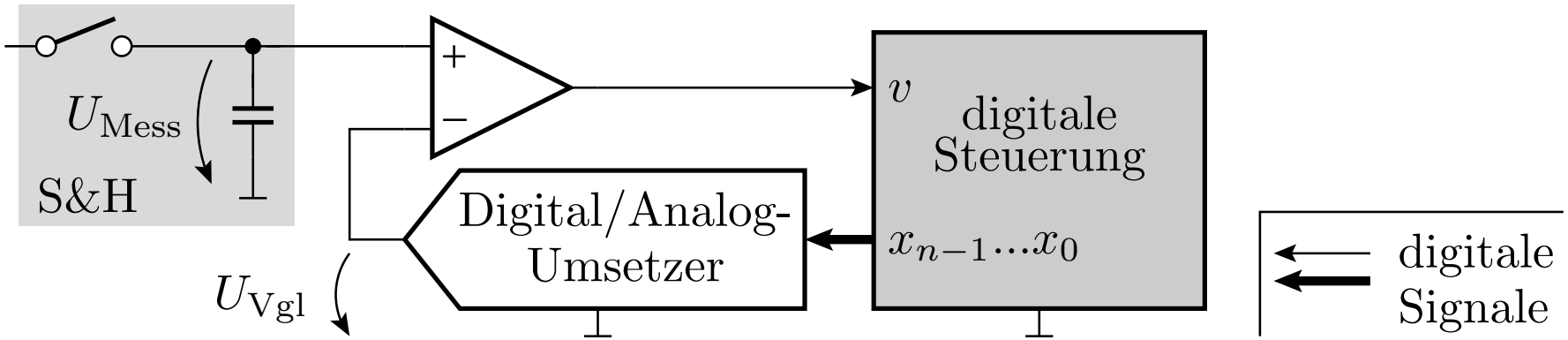
(U_{ref} – Referenzspannung; $\mathbf{x}$ – Ausgabewert (Bitvektor); n – Bitanzahl des Ausgabewerts).

Sukzessive Approximation





Ein Vergleich je Bit. Der Messwert darf sich während der Wandlung nicht ändern. Deshalb wird U_{Mess} mit einer Sample-and-Hold-Schaltung (S&H) abgetastet und während der Messung gespeichert.



Beispiel zur ADC-Initialisierung

```
void adc_init(){
  ADMUX = 0b01000000;// ADC0 (PF0) mit AREF=AVCC
  // Einschalten mit Wandlungstaktteiler 64
  ADCSRA = (1<<ADEN)|(0b110<<ADPS0);
  DDRF &= ~0x01;      // Sensoreingang als Eingang
  PORTF &=~0x01;      // Ausgabewert 0 (hochohmig)
}
```

- Der Sensor ist an ADC0 (PF0) (Kanal 0 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 PF0 als Eingang mit Ausgabewert 0 (Pullup aus) zu konfigurieren.

Blockierende Funktion zur Messung eines Analogwerts

```
uint16_t getADC(){
 uint16_t wert;
 ADCSRA |= (1<<ADSC);                  //Wandlung starten
 while(!(ADCSRA & (1<<ADIF)));//auf ADIF warten
 ADCSRA |= (1<<ADIF);                  //ADIF löschen
 wert = ADC;                           //Ergebnisrückgabe
 return wert;
}
```

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

Aufgaben

Aufgabe 5.1: Zahlenschloss

An Port A soll ein Modul mit vier Tastern an PA.0 bis PA.3 mit den Nummern 0 bis 3 stecken. In einer Endlosschleife soll

- Taster 0 die Schaltung rücksetzen (Startzustand, alle LEDs aus.)
- Aus dem rückgesetzten Zustand soll jede Tastereingabe die LED6:LED0 hochzählen.
- Die richtige Eingabefolge 2, 4, 8, 4, 2 soll zusätzlich LED an PJ.0 einschalten.

1. Wie ist das Polling zu organisieren, nacheinander je auf ein oder nebenläufig auf mehrere Ereignisse warten?
2. Programm entwickeln.

Lösung

1. Taste 0 immer abfragen. Für die anderen Tasten genügt die alternierende Abfragereihenfolge »keine Taste« und »mindestens eine Taste« gedrückt.
2. Programm: Automat mit PORTJ gleichzeitig als Zustand und Ausgabe.

```
#include <avr/io.h>
void main(){
 DDRB = 0; DDRJ = 0xFF;
 uint8_t x;          // aktuelle Eingabe
 uint8_t x_del=0;    // letzte Eingabe
 uint8_t ok;         // bisherige Eingabe richtig
 uint8_t dat[] = {2, 4, 8, 4, 2};//richtige Eing
 while(1){           // Endlosschleife
  <Schrittfunktion Automat>
 }
}
```

In einer Endlosschleife soll

- Taster 0 rücksetzen (Startzustand Z0, alle LEDs aus.)
- Von Z0 aus soll jede Tastereingabe die LED6:LED0 hochzählen.
- Die richtige Eingabefolge 2, 4, 8, 4, 2 soll zusätzlich LED an PJ.0 einschalten.

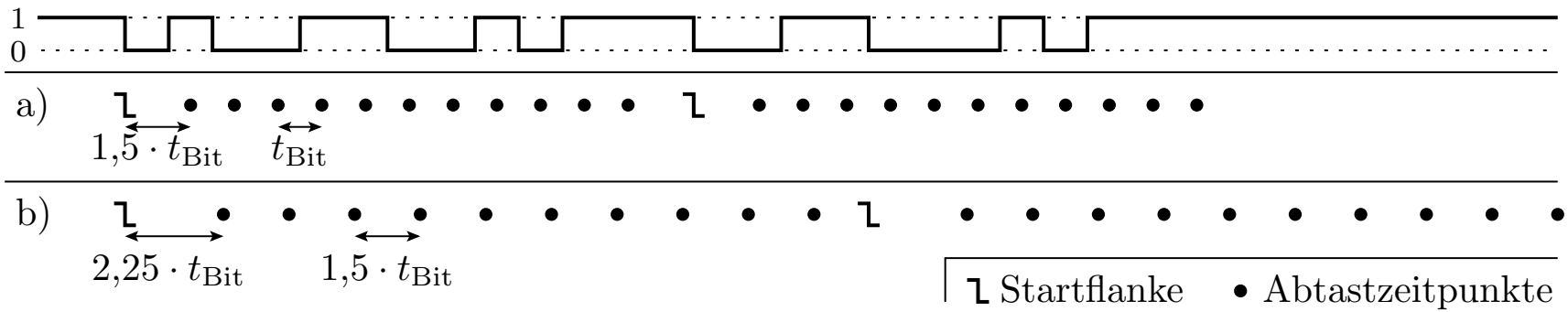
```
x = PINB;                          //Eingabe lesen
if (x & 1){ok=1; PORTJ=0;}         //wenn Rücksetztaste ...
else if (!x_del && x){             //sonst wenn Taste
                                   //und vorher keine
 if(PORTJ < sizeof(dat)){          //wenn Eingaben fehlen
  if(x != dat[PORTJ]) ok=0;        //bei falscher Zahl "ok"
 }                                 //löschen
 PORTJ++;                          //Zustand weiterzähler
 if ((PORTJ==sizeof(dat))          //alle Zahlen eingegeben
                  && ok){          //und alle richtig
 PORTJ |= (1<<7);}                 //LED an PJ.7 ein
}
x_del = x;                         //vorherige Eingabe
```

Aufgabe 5.2: UARD-Daten

Gegeben ist der nachfolgende von einer UART mit dem Protokoll 8E1 (8 Datenbit, gerade Parität, ein Stoppbit) generierte Signalverlauf. Welche Werte werden für die 8 Datenbits, das Paritätsbit und das Stoppbit empfangen, wenn im Empfänger dasselbe Protokoll und

1. dieselbe Baudrate
2. die 1,5-fache Baudrate

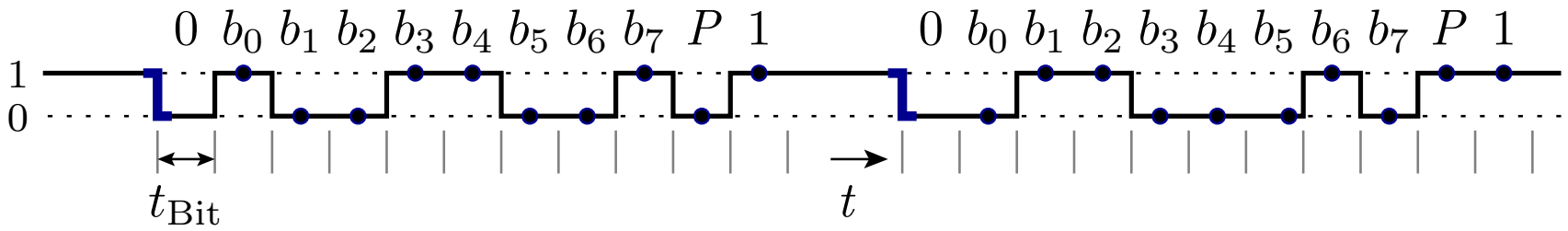
wie im Sender eingestellt ist.



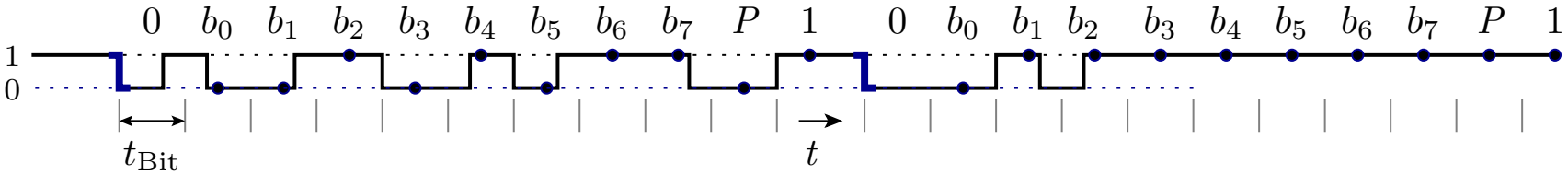
Sind die Empfangsdaten zulässig?

Lösung

1. Dieselbe Baudraten:



1. Byte: $\mathbf{b} = 0\mathrm{b}10011001$, $\mathrm{P} = 0\surd$, Stoppbit $= 1\surd$
2. Byte: $\mathbf{b} = 0\mathrm{b}01000110$, $\mathrm{P} = 01\surd$, Stoppbit $= 1\surd$



1. Byte: $\mathbf{b} = 0\mathrm{b}11010100$, $\mathrm{P} = 0\surd$, Stoppbit $= 1\surd$
2. Byte: $\mathbf{b} = 0\mathrm{b}11111110$, $\mathrm{P} = 1\surd$, Stoppbit $= 1\surd$