## Speicher, Digitale Ein- und Ausgabe

*DDRx* (Data Direction Register):
Bit auf **1 für Ausgang**, oder **0 für Eingang**
*PORTx* (Port Register):
Wenn Pin auf Ausgang, dann 1 = 5V und 0 = 0V
*PINx* (Port Input Register):
Wenn Pin auf Eingang, dann 1 = HIGH liegt an und 0 = LOW liegt an

```
pinMode(13, OUTPUT);
digitalWrite(13, HIGH);
digitalRead(13); … if (digitalRead(13) == HIGH)…


Serial.begin(9600);

Serial.println("Eingabe ist: " + Serial.readString());

Serial.available() :
Anz. Bytes, die zum Lesen verfügbar sind. Evtl. in while Schleife.
```

```
void setup() {
  DDRA |= (1 << DDA4);
  //pinMode(pin, OUTPUT);
  DDRA &= ~(1 << DDA0);
  //pinMode(sw, INPUT);

  PORTA &= ~(1 << PA4);
  //digitalWrite(pin, LOW) }
}
```

```
void loop() {
  if (PINA & (1 << PINA0)) {
  //if (digitalRead(sw)) {
    PORTA |= (1 << PA4);
    //digitalWrite(pin, HIGH);
    delay(2000);
    PORTA &= ~(1 << PA4);
    //digitalWrite(pin, LOW);
  }
```

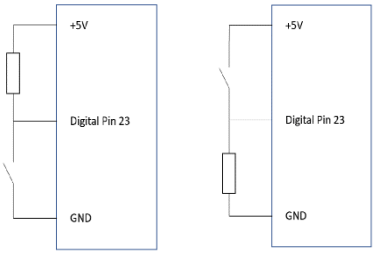
**Mikrocontroller Bestandteile**: MicroProzessor, Timer, Schnittstellen, Speicher, AD-Wandler
**Entwicklerboard**: Arduino Mega
**Mikroprozessor**: Atmega2560
**Cross-Compilation**: Programm wird nicht auf Zielplattform (Mikrocontroller), sondern auf anderer Plattform übersetzt
**Flashen**: hex-File von PC an Entwicklerboard senden

**Harvard-Architektur**: Daten- und Instruktionsspeicher getrennt
Instruktionsspeicher: Flash Speicher
Daten: SRAM

**Nichtflüchtige Speicher:**
ROM, OTPROM
EEPROM: Begrenzte Anzahl an Schreib/Lesezyklen.
Konfigurationsdaten, Kalibrierungsdaten
Flash: Programm- Daten/Code
**Flüchtige Speicher**: SRAM(Arbeitsspeicher, Register, Stack usw.) und DRAM

*Einzelne Bits setzen:*
x |= (1 << Bitnummer);
*Einzelne Bits löschen:*
x &= ~((1 << Bitnummer) | (1 << Bitnummer2));
*Testen ob Bit auf 1:*
If (DDRC & (1 << Bitnummer)) { }
*Testen ob Bit auf 0:*
If ( ! (DDRC & (1 << Bitnummer))) { }
*Alle Bits umdrehen:*
x = 0xFF ^ x
*LED togglen:*
PORTA ^= (1 << PORTA2);
PINA |= (1 << PINA2);
**Vorwiderstand** berechnen:

$$R_v = \frac{u_{ges} - u_F}{I_F}$$




**Pull-Up** / Active Low
Bei offenem Taster wird Spannung am Pin auf HIGH gezogen. Auch intern vorhanden.

**Pull-Down** / Active High
Bei offenem Taster wird Spannung am Eingang Auf LOW gezogen.

**Entprellung**
Einmaliges betätigen eines Schalters führt evtl zu mechanischen Vibrationen.
SW-Lsg: Künstliche Wartezeit nach Zustandswechsel – Bis Schalter eingeschwungen.

## Interrupts

sei(): Interrupts global aktivieren (oder: SREG |= 128)
cli(): Interrupts global deaktivieren

*SREG* (AVR Status Register):
Bit 7 auf 1 = sei();

*EIMSK* (External Interrupt Mask Register):
Speziellen Interrupt de-/aktivieren

*EICRA* (External Interrupt Control Register A)
*EICRB* (External Interrupt Control Register B):
ISCn0 und ISCn1. Falling oder Rising Edge.
n ist die Interrupt Nummer.

*EIFR* (External Interrupt Flag Register):
Wenn Interrupt ausgelöst: Bit ist 1

*ISR* (INT0_vect) { }

attachInterrupt(digitalPinToInterrupt(21), count, RISING);

```
volatile int counter = 0;
volatile unsigned long time = 0;

void setup() {
  Serial.begin(9600);

  sei();
  EIMSK |= (1 << INT2); //pin 19
  EICRA |= (1 << ISC20) | (1 << ISC21); // rising edge
}

void loop() {
  Serial.println("Zählerstand: ");
  Serial.println(counter);
  delay(3000);
}

ISR (INT2_vect) {
  if (millis() - time >= 250) {
    time = millis();        //Entprellung
    counter++;
  }
}
```

**Busy Waiting:** while (DDRC & (1 << DDC3));
**Polling**: periodisches Abfragen, ob Ereignis eingetreten
**Interrupt**: Kurze Unterbrechung des laufenden Programms um einen anderen zeitkritischen, kurzen Vorgang zu bearbeiten. Hardware prüft dauernd parallel, ob Ereignis eingetreten ist.
Wenn auf ein seltenes Ereignis schnell reagiert werden muss.
**Trap**: Art von Interrupt, die aber synchron und reproduzierbar ist. z.B. System Call, Div durch 0 …

**Interrupt Request:**
Interruptereignis – [InterruptController] – über IRQ Eingang Unterbrechungsanforderung an CPU – CPU unterbricht Programm und startet Unterbrechungsroutine
**Interrupt Vector Table:**
Welches Interruptereignis gehört zu welcher ISR?
Jede Vectornummer hat eine zugehörige Programmadresse.
ISR ist selbst nicht unterbrechbar (I Bit SREG)

**Externe Interrupts**
Controller tastet zu Beginn jedes Taktzyklus ab. Falls Interrupt aktiviert, Aufruf der ISR.
Probleme: Leichte Verzögerung, „Prellung“
**Interne Interrupts**
Timer, A/D-Wandler, Watchdog
Bei Ablauf eines Timers unterbricht HW Ausführung der normalen Software

**Volatile**
Variable wird vor jedem Lesen aus SRAM gelesen und nach jedem Schreiben in SRAM geschrieben
!!Globale Variablen die in ISR vorkommen **immer volatile**!!!

**!!!volatile!!!**


## Timer

**n**: Timer 1-5
*TCCRnA* (Timer/Counter n Control Register A):
PWM
*TCCRnB* (Timer/Counter n Control Register B):
Prescaler; Starten des Timers; Input Capture, CTC
Beide TCCRn erst auf 0x00 setzen.
Auch wenn keinen Prescaler will, muss man setzen
*TCNTn* (Timer Counter n, 16 Bit):
Aktueller Zählerstand. Anfangs auf 0 setzen.
*OCRnA, OCRnB, OCRnC* (Output Compare Register, 16 Bit):
Wert gegen den Zählerstand verglichen werden kann
*ICRn* (Input Capture Register):
Bei Input Capture erfasster Wert wird gespeichert
*TIMSKn*:
Aktivieren/Deaktivieren der Timer Interrupts
*TIFRn*:
Timer bezogene Interrupt Flags
**CTC Beispiel**: TCCR4B |= (1 << WGM42);
ISR(TIMER4_OVF_vect) {}:
Interrupt bei Timer 4 Overflow
ISR(TIMER4_COMPA_vect) {}: Timer 4 compare A

```
// counts number of overflows since last second
volatile unsigned int overflow_counter = 0;

void setup()
{
  DDRA |= (1 << DDA2);

  // initialize timer4
  TCCR4A = 0x00;
  TCCR4B = 0x00;
  TCNT4  = 0x00;
  // activate clock, but don't use a prescaler p157
  TCCR4B |= (1 << CS40);
  // enable timer overflow interrupt
  TIMSK4 |= (1 << TOIE4);

  sei();// enable all interrupts
}

// ISR, called when timer overflow occurs
ISR(TIMER4_OVF_vect)
{
  // 16 MHz / 2^16 = 244,1 ->
  // during 1 second approx. 244 overflow events
  if (overflow_counter == 244) {
    overflow_counter = 0;
    PINA |= (1 << PINA2);       // toggle LED
  }
  overflow_counter++;
}
```

16Bit Timer => Timer läuft nach $\frac{2^{16}-1}{16\ MHz} = 4ms$ über
**Prescaler**
Fallende Flanke des Prescalers an Bit $Q_n$ triggert Counter.
Vorteil großer Prescaler: Messen langer Zeiten möglich.

Kleinstes messbares Zeitintervall **ohne Prescaler**:

$$\frac{1}{16MHz} = 62ns$$

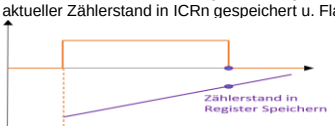
**mit f/1024 Prescaler**: $\frac{1}{16MHz/1024} = 64\mu s$ (1Tick)

Nachteil: Schlechtere Auflösung.
⇨ immer kleinstmöglichen Prescaler!

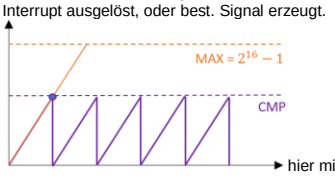
Welchen Prescaler für 3s Intervalle mit 16 Bit Timer?
Takt: 1 MHz (Bei 16MHz bis 16*3*1000000)

$$\frac{3000000}{x} = 2^{16} - 1 \Leftrightarrow x = 45{,}8 \Rightarrow 64\text{Prescaler}$$

**Input Capture**
Bei externen oder internen Signalen/Ereignissen wird aktueller Zählerstand in ICRn gespeichert u. Flag ges




**Output Compare**
Bei erreichen eines konfigurierten Zählerstandes wird Interrupt ausgelöst, oder best. Signal erzeugt.




**CTC Mode (Clear Timer on Compare Match)**
TOP Wert in OCRnA oder ICRn konfiguriert.
Zähler bei erreichen des Zählerstandes automatisch auf 0.

## Pulsweitenmodulation

**n**: Nummer des Timers

TCCRnA
Compare Output Mode
Fast PWM, Clear on compare match, usw.

TCCRnB
Fast PWM; Prescaler

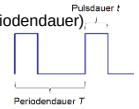
OCnA, OCnB, OCnC (Output Compare Pins):
PWM-Ausgang
Inverting oder non-Inverting Mode
Output Compare Pins müssen **als Ausgang im DDR** Register konfiguriert sein!

OCRnX (Output Compare Register):
Vergleichswert (Schwellwert) muss gesetzt werden, der jeweils PWM-Ausgang OCnX beeinflusst.

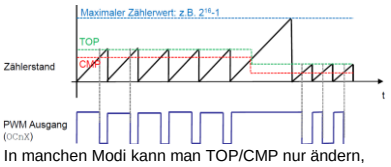
```
void setup() {
// set pin PH5 (pin5 of port H) to output; this is the PWM pin
// (alternative function: alternative function of PH5: OC4C)
DDRH |= (1 << DDH5);

// to be safe: initialize counter control registers to zero
TCCR4A = 0x00;
TCCR4B = 0x00;

// Fast PWM mode, counter TOP value taken from ICR,
TCCR4A |= (1<<WGM41);       // WGM bits: "1110", manual p145
TCCR4B |= (1<<WGM43) | (1<<WGM42);

// non-inverting mode; clear on compare match; manual p155, Table 17-4
TCCR4A |= (1<<COM4C1);

// good choice: use clk/8 prescaler -> 16 MHz / 8 = 2 MHz
// counter counts up from 0 to 39999 within 20 ms
TCCR4B |= (1<<CS41);

ICR4 = 40000;// configure period of PWM, i.e. TOP value in ICR register

// initial pulse width / duty cycle 1,25 ms
// --> (1,25 ms / 20 ms ) * 40000
OCR4C = 2500;

delay(3000);
}
```

```
void loop() {
  // Max Wert: 2,4 us:  4800
  // Min-Wert: 544 us: 1088

  OCR4C = 1088;   // duty cycle: min value
  delay(3000);

  OCR4C = 4800;   // duty cycle: max value
  delay(3000);
}
```

Signal mit konstanter Periode, aber variabler Pulsdauer wird erzeugt.
**Duty Cycle**: t/T (= Pulsdauer / Periodendauer)
Nur Duty Cycle wird ausgewertet, nicht Periodendauer.




TOP: ICRn Register (oder andere siehe S145 Tabelle)
CMP: OCRnX Register




In manchen Modi kann man TOP/CMP nur ändern, wenn Zähler gerade auf BOTTOM/TOP ist. („*Update of OCRnX at...*“ in Tabelle)

**Inverting u. Non-Inverting Mode**
Non-Inverting: siehe Links.
Inverting: PWM Ausgang genau andersrum TCCRnX Register

**Up-Down-Counter**
doppelte Periodendauer, geringere Auflösung
BOTTOM u. TOP immer genau in der Mitte

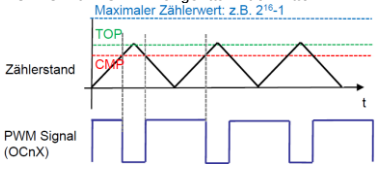



Tabelle S145 TCCRnA u TCCRnB Fast PWM(Up-Counter), PWM(Up-Down Counter)

## Analoge Ein-/Ausgabe

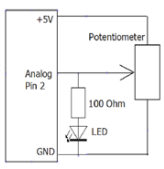
ADMUX
Referenzspannung wählen
Analoge Eingangspins für A/D Umsetzung wählen

ADCSRA
Aktivieren und Starten der A/D Umsetzung
(**ADSC** für manuelles triggern, wenn fertig wieder 0)
(**ADATE** für auto trigger z.b. bei Free running u. timer ovf (z.B. beim Sensor) im Free Running Mode nur einmal ADSC triggern)
**Prescaler** (ADPS!! 50-200kHz bei int. AD-Wandler)
Interrupts

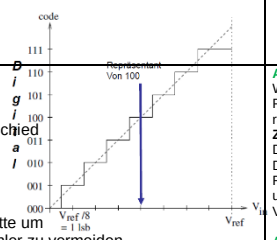
ADCSRB
Analoge Eingangspins für A/D Umsetzung wählen
Single Ended oder Differential Conversion
Free Running Mode oder manuelles Triggern

ADCL, ADCH
Speichert Ergebnis der A/D Umsetzung
Erst ADCL, dann ADCH lesen (atomarer Zugriff)
ADCL + 256 * ADCH

```
void setup()
{
  // activate serial console
  Serial.begin(9600);

  // enable ADC functionality
  ADCSRA |= (1 << ADEN);

  // use /128 prescaler (ADC requires 50 kHz to 200 kHz, see manual p271,
  // but system clock is 16 MHz)
  ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);

  // select ADC2 as input pin (there is one AD converter for the 16 AD:
  ADMUX |= (1 << MUX1);

  // use reference voltage 5V (Note: AVCC is AREF), manual, p281
  ADMUX |= (1 << REFS0);
}

void loop()
{
  // trigger ADC conversion
  ADCSRA |= (1 << ADSC);

  // wait until conversion is finished, see
  while (ADCSRA & (1 << ADSC));

  // read analog value, first LOW then HIGH
  unsigned int read = ADCL + 256 * ADCH;

  double result = 5.0 * read / 1024.0;
  Serial.print(result);
  Serial.println(" Volt");
}
```




**A/D Wandlung**
**Auflösung**:
Wie viel Spannungsunterschied pro Stufe?
$V_{ref} / 2^r$
**Repräsentant** eines Intervalls liegt in Intervallmitte um Quantisierungsfehler zu vermeiden.
Erstes Intervall: Repr. Von 000 Stufenbreite ½ LSB
Letztes Intervall: Stufenbreite 1 ½ LSB
r = 10 bei ATmega




**Fehlerquellen**
Quantisierungsrauschen
Umsetzungszeit
Änderung des Eingangs während der Umsetzung

**Ersatzwiederstand** von zwei parallelen Widerständen

$$R_{1,2} = \frac{R_1 \cdot R_2}{R_1 + R_2}$$

**Ansätze zur A/D Wandlung**
Wird mithilfe von D/A Umsetzung gemacht. Kompratr, Parallelverfahren, **Wägeverfahren**:
r=3, Vin=(1011)V -> 1000 – 1100 – 1010 – 1011
**Zählverfahren**: A/D mit D/A:
Digitaler Zähler speichert Schätzung des digitalen Wertes.
DAC wandelt Schätzwert in analoge Spannung um.
Rückkoppling: Komparator vergleicht gegen Eingangsspannung und anschließendes Erhöhen/Erniedrigen des Zählers.
Vorteil: Einfache Realisierung, Nachteil: lange Einschwingzeit

**A/D Umsetzung beim Atmega**
Nur **1 interner A/D Umsetzer**, aber 16 analoge Eingangspins können an A/D Umwandler weitergeleitet werden.
Konfigurierbar, welcher Eingang an A/D Um. Weitergeleitet wird

**Trigger**:
Manuelles Auslösen: durch Codeanweisung
Free Running Mode: Endlosschleife
Auto Trigger: angestoßen durch Timeroverflow, Komparatorausgang etc

**Erkennen, dass A/D Umsetzung beendet wurde:**
Auswerten eines speziellen Flags(ADSC in ADCSRA), oder durch speziellen Interrupt.

**Wertebereich:**
Single-Ended Conversion: [0,$V_{ref}$]
Differential Conversion: [-$V_{ref}$/2, $V_{ref}$/2]

Watchdog, Energiesparmodus, Reset

WDTCSR
Watchdog Modul Konfiguration
WDP's: Prescaler für Watchdog Zeit
Achtung!!: einmal nur "="
**Spezielles vorgehen zum Beschreiben des Registers! (Damit nicht ausversehen)**

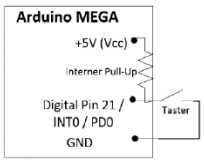
MCUSR
Informationen über Ursache des Resets (nach Neustart abrufbar)

wdt_reset() (in C) (Assembler: WDT)
Watchdog Timer zurücksetzen

SMCR
Energiesparmodus wählen

sleep_mode()
(Assembler: SE-Bit in SMCR setzen, dann SLEEP-Instruktion) sleep_mode() macht das automatisch!!
Energiesparmodus aktivieren

```
void setup(){
  Serial.begin(9600);
  Serial.println("System restart");
  // start watchdog
  cli();
  wdt_reset();
  // preparation for configuration: write logic
  // one to WDCE and WDE, manual p61
  WDTCSR |= (1<<WDCE) | (1<<WDE);
  // immediately afterwards (within 4 clock cycles):
  // set timeout to 4 seconds and start watchdog
  // Hint: WDCE bit must be cleared
  WDTCSR = (1<<WDP3) | (1<<WDE);

  // configure interrupt (alternative: use Arduino commands)
  DDRD &= ~(1 << DDD0);    // configure PD0 as input
  PORTD |= (1 << PORTD0);  // pull up (manual p68)
  EIMSK |= (1 << INT0);    // turn on INT0
  EICRA |= (1 << ISC01);   // set INT0 to trigger on falling edge
  sei();
}

void loop(){}

ISR (INT0_vect){
    wdt_reset();
    Serial.println("ResetWDT");
}
```




**Watchdog**
Timer, der hoch oder runterzählt. Muss vor Überlauf zurückgesetzt werden. Sonst: Interrupt oder Reset.
**Aufgaben:**
Überprüfung: Codestellen in vorgegebener Zeit erreicht? SW noch aktiv und nicht abgestürzt?
Bei Timeout: Überführen in wohldefinierten Zustand. Neustart oder Interrupt auslösen.
Erkennt Probleme, löst sie aber nicht!
**Prescaler:** Beeinflusst Zeit bis Watchdog Timeout

**Energiesparmodus**
Energieverbrauch verringern durch: Systemtakt verlangsamen, Betriebsspannung verringern, abschalten nicht benötigter Module (Energiesparmodi (ESM))
**ESM unterscheiden sich bzgl.** Abgeschalteter Komponenten und aufweckender Ereignisse (Ext Interrupts, Watchdog Interrupt, Speicherzugriff beendet, Timer, Anlegen einer (leeren) ISR und Aktivieren des Interrupts genügt).
Aufwachen kann verzögert passieren
**Energiesparmodi beim Atmega2560:**
Idle Mode, ADC Noise Reduction Mode, Power Save Mode, Power Down Mode, Standby Mode

**Reset**
System von wohldefiniertem Zustand starten.
Init. aller Register u. I/O Ports auf Default Werte, künstl. Delay, damit sich Spannungswerte stabilisieren, erste Instruktion an Adresse 0x0000 ausführen, wo im Normalfall JMP zur Reset-Routine ist, Reset Routine initialisiert stack pointer u. letzte Anweisung ist JMP in Main-Routine (setup)

**Sensordaten**
In bestimmten Bereich linearer Zusammenhang zw. Messgröße (z.B. °C) u. Ausgangsspannung.
**Beispiel TMP36**: -40°C – 125°C
750mV bei 25°C. Output *Scale Factor* 10mV/°C
Min Ausgangsspannung: 100 mV
Max Ausgangsspannung: 1750 mV
Max Ausgangsspannung sollte möglichst knapp unter Referenzspannung liegen.
**Binäre Zahl** (bei $V_{ref}$ = 2,56V):
Max: (1,750V / 2,56V) * 2^10 = 700
Min: (0,100V / 2,56V) * 2^10 = 40

**Binäre Zahl in Messgröße: y=mx + t**
y: Messgröße, x: binäre Zahl
-40°C = m*40 + t [°C]  125°C = m*700 + t [°C]

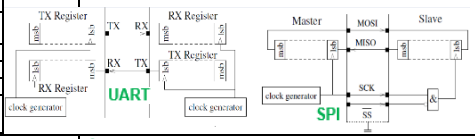
Kommunikationsschnittstellen

**USART Register (n: welches der 4 UART Module):**
UDRn (char a = UDR2, oder UDR2 = a)
Zu sendendes, oder empfangenes Byte
UCSRnA
Übertragungsinfos, z.B. Übertragung erfolgr beendet?
UCSRnB
USART-bezogene Interrupts,
Empfänger / Receiver aktivieren
UCSRnC
Modus wählen (synchron oder asynchron)
Datenformat(Stoppbit, Parität)
UBRRnL (8Bit) / UBRRnH (4 MS Bits)
Baudrate einstellen
**SPI Register:**
Mstr: MOSI, SCK als Ausgang, Slv: MISO als Ausgng
SPCR
Konfiguration: Aktivierung, Interrupts, Master/Slave? Daten bei steigender oder Fallender Flanke lesen?
SPSR Infos, z.B. trat SPI Interrupt auf?
SPDR (SPDR = 7 oder char tmp = SPDR)
Nach einer Taktperiode sind 8 Bits aus Register gesendet worden u. 8 empfangene Bits stehen jetzt im Register

```
// USART BEISPIEL
#define MYUBBR 103//calculated. table S.202
void setup(){
  // set BAUD RATE
  UBRR0L = (unsigned char) MYUBBR;
  UBRR0H = (unsigned char) (MYUBBR >> 8);
  UCSR0B = (1<<TXEN0); // ENABLE transmitter
  // Set FRAME FORMAT: 8 data, 1 stop bit
  UCSR0C =  (1<<UCSZ01) | (1<<UCSZ00);
}
void uart_putchar(char c) {
   // Wait until data register empty.
  while (!(UCSR0A & (1 << UDRE0)));
  UDR0 = c;
}
void loop(){
  char text[] = "Hallo";
  int i = 0;
  for (i = 0; i < sizeof(text); i++)  {
    uart_putchar(text[i]);
  }
  delay(2000);
}
```

**Klassifizierung**
**Seriell** vs **parallel** | vs |||
**Synchron** (eigener Takt für Datenleitung) vs **Asynchron**(Empfänger muss Takt d. Senders kennen)
**Bus** (Mehr als zwei Geräte verbunden, erfordert Adressierung) vs **Point-to-Point**
**Volldduplex**(Datenübertragung in beide Richtungen gleichzeitig möglich, separate Leitungen für Senden u. Empfangen vs **Halbduplex**
**Peer-toPeer** vs **Master-Slave** (Nur Master darf Kommunikation starten)
**Differential**(Spannungsunterschied zw. 2 Leitungen trägt Information vs **Single-Ended**(Gemeinsame GND Leitung für alle Datenleitungen)

| | UART | SPI | I²C |
|---|---|---|---|
| Seriell | Ja | Ja | Ja |
| Duplex | Ja | Ja | Nein |
| Synchron | Nein<br>Baud config | Ja | Ja |
| Bus | Nein | Jein | Ja |
| Anz.Leitungen | 3 | 5 | 3 |
| Differential | Nein | Nein | Nein |
| Datenrate ATmega2560 | BAUD = fosc / (16(UBRRn+1)) | $f_{osc}/128$ – $f_{osc}/2$ | Max 400 kbit/s |




**UART (oder SCI) (Arduino hat 4 USARTs)**
2 Datenleitungen: TxD und RxD
Sender u. Empfänger müssen Baudrate kennen
Übertragung von UART-Frames D{E|O|N}S
Bsp: 8E1: 1Startbit, 8Datenbits, even parity, 1 Stopbit

**SPI (hohe Geschwindigkeit, kein Overhead)**
Master-Slave. 4 Datenleitungen:
**MOSI**: Master Out, Slave IN (8 Bit Schieberegister)
**MISO**: Master In, Slave Out (8 Bit Schieberegister
**SCK**: System Clock, **$\overline{SS}$**: Slave Select (aktiver Slave)

**I²C, TWI (Viele Geräte)**
Bus mit 7 Bit Adressierung
**SCL**: Serial Clock Line, **SDA**: Serial Data Line
*Startbedingung*: (Fall. Flanke SDA) + (SCL == HIGH)
Adresse anlegen ->R/$\overline{w}$: Master spezifiziert, ob Lese oder Schriebzugriff -> Slave: ACK -> Datentransfer
*Stoppbedingung*: (Steigende Fl. SDA)+(SCL==HIGH)

```
//SLAVE
void setup()
{
 Serial.begin(9600);
  // data direction: set MIS==DDB3 to output, all other input
  DDRB = (1<<DDB3);
  // Slave Init: Enable SPI,
  SPCR = (1<<SPE); //not setting MSTR means that it is a slave
}
// send and receive data
unsigned char spi_transceive(unsigned char data) {
  // start transmission by putting data into buffer
  SPDR = data;
  // wait until transmission completes
  while(!(SPSR & (1<<SPIF)));
  // return received data
  return (SPDR);
}


void loop()
{
  char text[] = "Hallo Maste!";
  for (int i = 0; i < sizeof(text); i++)  {
    char received = spi_transceive(text[i]);
    Serial.print(received);
  }
  delay(1000);
}
```

Peripherie

```
// Function Set, 4-bit, 2 line mode, 5x8 dots
writeInstruction(0x28); // 2 line mode (DB3 = 1)
//  Return Home, set cursor to beginning
writeInstruction(0x02);
delayMicroseconds(2500);
//Entry Mode Set, increment cursor, no display shift
writeInstruction(0x06);
writeInstruction(0x01);// clear display: 0x01
delayMicroseconds(300);
}
// write an instruction, indicated by RS == LOW
void writeInstruction(char instr) {
  digitalWrite(RS, LOW); // when data => HIGH
  PORTC = instr;
  enablePulse(); // commit command, short pulse on E pin
  PORTC = (instr << 4);
  enablePulse();
}
void enablePulse() {
  digitalWrite(E, LOW);
  delayMicroseconds(3);
  digitalWrite(E, HIGH);
  delayMicroseconds(3); // enable pulse must be > 450 ns, see p49 of HD44780 manu
  digitalWrite(E, LOW);
  delayMicroseconds(200);// commands need > 37 us to settle
}
void loop()
{
  // set cursor to beginning of first line:  command 0x80
  writeInstruction(0x80); //(DDRAM address 0x00, see p11 of manual)
  char line1[] = "Embedded Systems  ";
  for (int i = 0; i < strlen(line1); i++) {
    writeData(line1[i]);
  }
  // set cursor to beginning of 2nd line line: command 0xC0
  writeInstruction(0xC0);// (DDRAM address 0x40, see p11 of manual)
  char line2[] = "macht Spass";
  for (int i = 0; i < strlen(line2); i++) {
    writeData(line2[i]);
```

**Ablauf Display schreiben**
Mikrocontroller -> ASCII an DDRAM -> Steuereinheit schlägt Muster im CGRAM nach u. blendet es auf Display ein.

**CGRAM (definiert Aussehen von Schriftzeichen)**
ROM: Standardzeichen, a-z, A-Z, ...
RAM: Benutzerdefinierte Zeichen

| CGRAM-Adresse | 16 | 8 | 4 | 2 | 1 | Byte dezimal | Byte hexadezimal |
|---|---|---|---|---|---|---|---|
| 0x00 | | | | | | 16+1 = 17 | 0x11 |
| 0x01 | | | | | | 16+1 = 17 | 0x11 |
| 0x02 | | | | | | 16+2+1 = 19 | 0x13 |
| 0x03 | | | | | | 16+4+ | |
| 0x04 | | | | | | | |
| 0x05 | | | | | | 16+ | |
| 0x06 | | | | | | 16+1 | |
| 0x07 | | | | | | | |

| Upper 4 Bits / Lower 4 Bits | 0000 | 0001 | 0010 | 0011 |
|---|---|---|---|---|
| xxxx0000 | CG RAM (1) | | | 0 |
| xxxx0001 | (2) | | ! | 1 |

**DDRAM (welche Zeichen zeigt Display aktuell und an welcher Stelle?)**
Speicheradresse:
1 Zeilenmodus: 0x00 – 0x4F
2 Zeilenmodus:
0x00 – 0x27 (1. Zeile), 0x40 – 0x68 (2. Zeile)

**Cursor**
Display kann nicht alle Zeichen des DDRAM anzeigen
Shift-Operationen zum Verschieben des sichtbaren Bereichs
Cursor zeigt auf Zeichen, das User verändert, wenn er WriteCommand zum LCD Didplay sendet
Cursor: **Zeiger auf Adresse im DDRAM**

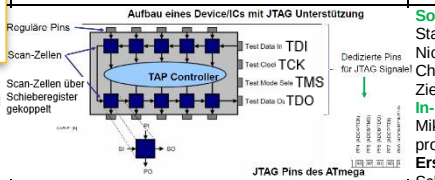

**4-Bit Modus (4 statt 8 Datenleitungen)**
D4-D7 statt D0-D7
Sende **erst höherwertige** Nibble, dann das niedrigwertige Nibble => 2 statt 1 Schreibzyklus
Zum Aktivieren des 4 Bit Modus muss eine spezielle Initialisierungssequenz durchlaufen werden.
0x18 in DDRAM: erst 0001, dann 1000

```
//MASTER
void setup(){
  Serial.begin(9600);
  // data direction: set MOSI==DDB2 and SCK==DDB1 and SS=DDB0 to output
  DDRB = (1<<DDB2) | (1<<DDB1) | (1<<DDB0);
  // Master init: Enable SPI, set as master, set clock rate to fck/128
  SPCR = (1<<SPE)|(1<<MSTR)|(1<<SPR0)|(1<<SPR1);
}
// send and receive data
unsigned char spi_transceive(unsigned char data) {
  // set SS to low, activating slave, synchronization
  PORTB &= ~(1<<DDB0);
  // start transmission by putting data into buffer
  SPDR = data;
  // wait until transmission completes
  while(!(SPSR & (1<<SPIF)));
  // return received data
  char result = SPDR;
  // set SS to high, deactivate slave
  PORTB |= (1<<DDB0);
  return result;
}
```

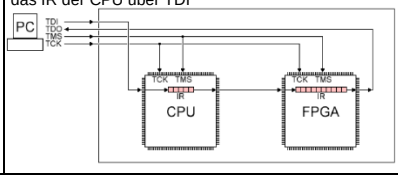
SW-Download / Debugging
- Jeder Atmel hat eindeutigen Signaturcode. Wird bei ISP überprüft.
- Aktuell geladenes Programm kann aus Flash des uC's als HEX-Datei auf PC geladen werden.
- Bootloader reparieren: Bootloader (Hex-File) ins Flash laden.

**JTAG (Boundary Scan, Debugging, In-System Programming)**
Erreichbarkeit aller virtuellen Testpunkte über eine einzige, einheitliche Schnittstelle prüfen.
JTAG Chain: alle DR-Register werden in Chain eingebunden. Kreisförm. weiterschiben in Schiebereg

**Bestandteile JTAG Kompatible ICs**
TAP Controller: Zustandsautomat, der Zustandslogik steuert. Gesteuert durch TMS Eingang
Zustand Shift-IR: Bits an TDI werden als Inst. ausgef.
Zustand Shift-DR: Bits in TDI/TDO Chain werden als zu schreibende/lesende Daten interpretiert.




**Beispiel Schreiben der IR-Register**
Beide Controller müssen in SHIFT-IR Zustand versetzt werden (TMS: 01100). Dann senden der 10 Bits für IR des FPGAs über TDI und dann 5 Bits für das IR der CPU über TDI




**Software Download (Flashing)**
Standardformat: Intel Hex-Format.
Nicht nur Sequenz von Opcodes, sondern auch Checksumme, Info über Programmgröße und Zielspeicherort

**In-System Programming**
Mikrocontroller direkt im Einsatzsystem programmieren.
**Erste Möglichkeit**: Programmierung mit seriellen Schnittstellen: SPI, oder JTAG
Benötigt Zusatzhardware, die zuvor erstelltes Programm/Daten in internen nichtflüchtigen Speicher (EEPROM, Flash) schreibt.
uC erkennt Programmierung durch spezielle Signalfolgen, Timing usw. **zweite Möglichkeit**:

**Bootloader (keine Zusatzhardware nötig)**
Programm für das laden von Programmen + USB Kommunikation
Lausch nach Reset, ob neues Programm über USB hochgeladen werden soll.
Falls nein: Bereits vorhandenes Programm wird gestartet.
Separater Speicherbereich für Bootloader

**Debugging**
**HW Breakpoint**: Spezielles HW-Modul überwacht Adressbus und wartet auf Holen einer Instruktion von einer bestimmten Adresse.
**SW Breakpoint**: Opcode am Ort des Breakpoints wird vorübergehend mit einer speziellen „Halte"-Instruktion ersetzt.

**Hilfsmethoden zum Debuggen:**
LEDs, Taster und Schalter, UART (sout)

**Simulation**
Target Controller wird auf Host System simuliert.

**HW-Debugging**
Debugging direkt auf Ziel-Hardware – HW-Breakpoints auf Mikrocontroller
Schnittstelle: **JTAG**
Kommunikation von PC zu Mikrocontroller über Zusatz-HW / JTAG-Adapter.
Häufig werden nur Register, die für Debugging wichtig sind in die JTAG Chain eingebunden.

Automaten

```
int microPin = 24;
int ledPin = 21;
// states                    // events
typedef enum {               typedef enum {
  WAIT_FOR_CLAP_ONE,           NO_EVENT,
  ONE_CLAP_DETECTED,           MICRO_PIN_HIGH,
  WAIT_FOR_CLAP_TWO            TIMER100_EXPIRED,
} state_t;                     TIMER300_EXPIRED
                             } event_t;
// global variables
state_t state;       // current state
// timestamp for 100 ms timer, 0 means inactive
unsigned long timer100 = 0;
// timerstamp for 300 ms timer, 0 means inactive
unsigned long timer300 = 0;

// transition functions
void stay() {
}
void enterOneClapDetected() {
  state = ONE_CLAP_DETECTED;
  timer100 = millis();  // start timer for 100 ms
}
void enterWaitForClapTwo() {
  state = WAIT_FOR_CLAP_TWO;
  timer300 = millis(); // start timer for 300 ms
}
void enterWaitForClapOneTimer300() {
   state = WAIT_FOR_CLAP_ONE;
}
void enterWaitForClapOneMicroPin() {
   state = WAIT_FOR_CLAP_ONE;
   digitalWrite(ledPin, !digitalRead(ledPin)); // toggle LED
}
```

```
// transition table
state_t (*state_table[3][4]) (void) = {
  //                          NO_EVENT    MICRO_PIN_HIGH
  /*WAIT_FOR_CLAP_ONE*/      {stay,      enterOneClapDetected,
  /*ONE_CLAP_DETECTED*/      {stay,      stay,
  /*WAIT_FOR_CLAP_TWO*/      {stay,      enterWaitForClapOneMicroPin,
};

void setup() {
  pinMode(microPin, INPUT);
  pinMode(ledPin, OUTPUT);
  state = WAIT_FOR_CLAP_ONE;   // initial state
  Serial.begin(9600);
}

void loop() {

  // detect events
  event_t event = NO_EVENT;
  if (digitalRead(microPin) == HIGH) {
    event = MICRO_PIN_HIGH;
  } // timer valid and 100 ms expired
  else if (timer100 && millis() - timer100 > 100) {
    event = TIMER100_EXPIRED;
    timer100 = 0;      // reset timer;
  }
  else if (timer300 && millis() - timer300 > 300) {
    event = TIMER300_EXPIRED;
    timer300 = 0;
  }

  // use transition table to switch state
  state_table[state][event]();
}
```

```
      TIMER100_EXPIRED      TIMER300_EXPIRED
      stay,                 stay},
      enterWaitForClapTwo,  stay },
                            enterWaitForClapOneTimer300}
```

**Ereignisse**
Ereignisse in der Praxis asynchron, da der Zustandsautomat nie blockieren darf.
delay kann dazu führen, dass andere Ereignisse ignoriert werden.

2 Ansätze:
Event-Driven: Ereignisse werden vorwiegend über HW-Interrupts erkannt. Zustandsübergang in ISR.
Problem: Gleichzeitige Interrupts (priorisieren?)

Polling: Prüfe in jedem Schleifendurchlauf, ob neues Ereignis vorliegt. -> Zustandübergang

**Arduino als ISP Programmierer**
Programm auf Arduino laden, damit er als ISP fungiert. Und ATtiny85 per SPI programmieren kann.
Bootloader brennen
Programm auf anderen uC laden

# Register

**Digital IO:**

- DDRx (Data Direction Register):
  - Entsprechendes Bit auf 1 für Ausgang, oder 0 für Eingang
- PORTx (Port Register):
  - Wenn Pin auf Ausgang, dann 1 = 5V und 0 = 0V
- PINx (Port Input Register):
  - Wenn Pin auf Eingang, dann 1 = HIGH liegt an und 0 = LOW liegt an

**Timer:**

- TCCRnA (Timer/Counter n Control Register A):
- TCCRnB (Timer/Counter n Control Register B):
  - Prescaler
  - Starten des Timers
  - Input Capture
- TCNTn (Timer Counter n, 16 Bit):
  - Aktueller Zählerstand
- OCRnA (Output Compare Register A, 16 Bit):
  - Wert gegen den Zählerstand verglichen werden kann
- OCRnB (Output Compare Register B, 16 Bit):
  - Wert gegen den Zählerstand verglichen werden kann
- ICRn (Input Capture Register):
  - Bei Input Capture erfasster Wert wird gespeichert
- TIMSKn:
  - Aktivieren/Deaktivieren der Timer Interrupts
- TIFRn:
  - Timer bezogene Interrupt Flags

**Pulsweitenmodulation:**

- OCnA:
- OCnB:
- OCnC (Output Compare Pins):
  - Inverting oder non-Inverting Mode
  - Output Compare Pins müssen als Ausgang im DDR Reigster konfigueriert sein!
- OCRnX (Output Compare Register):
  - Vergleichswert muss gesetzt werden

**Interrupts:**

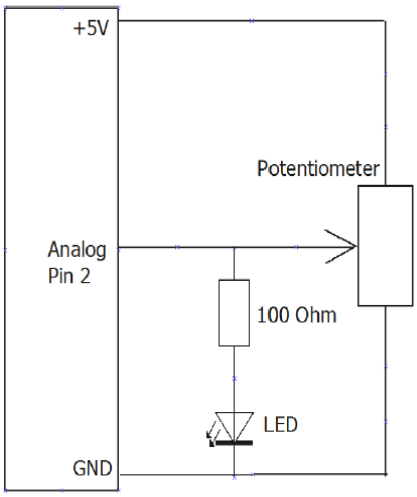
- sei() (Set Enable Interrupt):
  - Interrupts global aktivieren
- SREG:
  - I Bit hier setzen statt sei() möglich
- EIMSK:
  - De/aktivieren von speziellen Interrupts
- EIFR:
  - Interrupt Flags
- EICRA:
- EICRB:
  - Steigende/fallende Flanke?

**Analoge IO:**

- ADMUX:
  - Referenzspannung wählen
  - Analoge Eingangspins für A/D Umsetzung wählen
- ADCSRB:
  - Analoge Eingangspins für A/D Umsetzung wählen
  - Single Ended oder Differential Conversion
  - Free Running Mode oder manuelles Triggern
- ADCSRA:
  - Aktivieren und Starten der A/D Umsetzung
  - Prescaler
  - Interrupts
- ADCL u. ADCH:
  - Speichert Ergebnis der A/D Umsetzung
  - Erst ADCL, dann ADCH lesen (atomarer Zugriff)

TODO:

- Übung 7 vielleicht noch mehr Beispiele




-
- Reset arten

```ino
void setup(){
  Serial.begin(9600);
  // enable ADC functionality
  ADCSRA |= (1 << ADEN);
  // use /128 prescaler, see manual p271
  ADCSRA |= (1 << ADPS2) | (1 << ADPS1) | (1 << ADPS0);
  // select autotrigger, to use free-running mode
  ADCSRA |= (1 << ADATE);
  // use free-running mode
  ADCSRB &= ~((1 << ADTS2) | (1 << ADTS1) | (1 << ADTS0));
  // select ADC2 as input pin
  ADMUX |= (1 << MUX1);
  // use reference voltage 2,56 V, manual p281
  ADMUX |= (1 << REFS1) | (1 << REFS0);
  // start conversion
  ADCSRA |= (1 << ADSC);
}

void loop()
{
  // note: conversion is automatically triggered in free running mode
  // read analog value, first LOW then HIGH register
  unsigned int read = ADCL + 256 * ADCH;

  // convert integer value into temperature
  double temperature = 0.25 * read - 50;
  Serial.println(temperature);
  delay(1000);
}
```

- Sensor Beispiel:
- I^2C Datenübertragung: