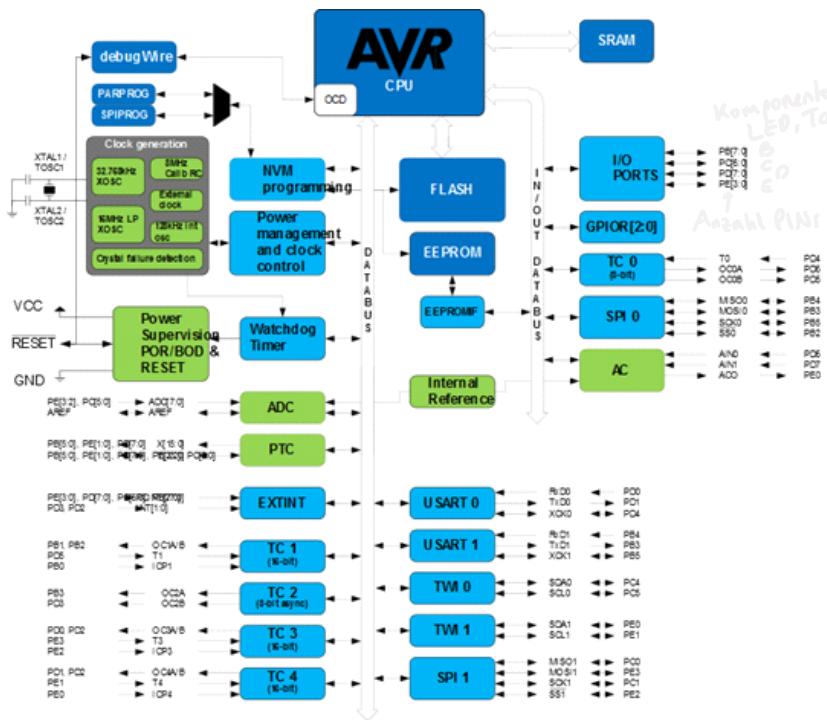


Theorie

Mittwoch, 30. November 2022 19:14

Block Diagram

Mittwoch, 21. September 2022 09:49



{ Port B

MC

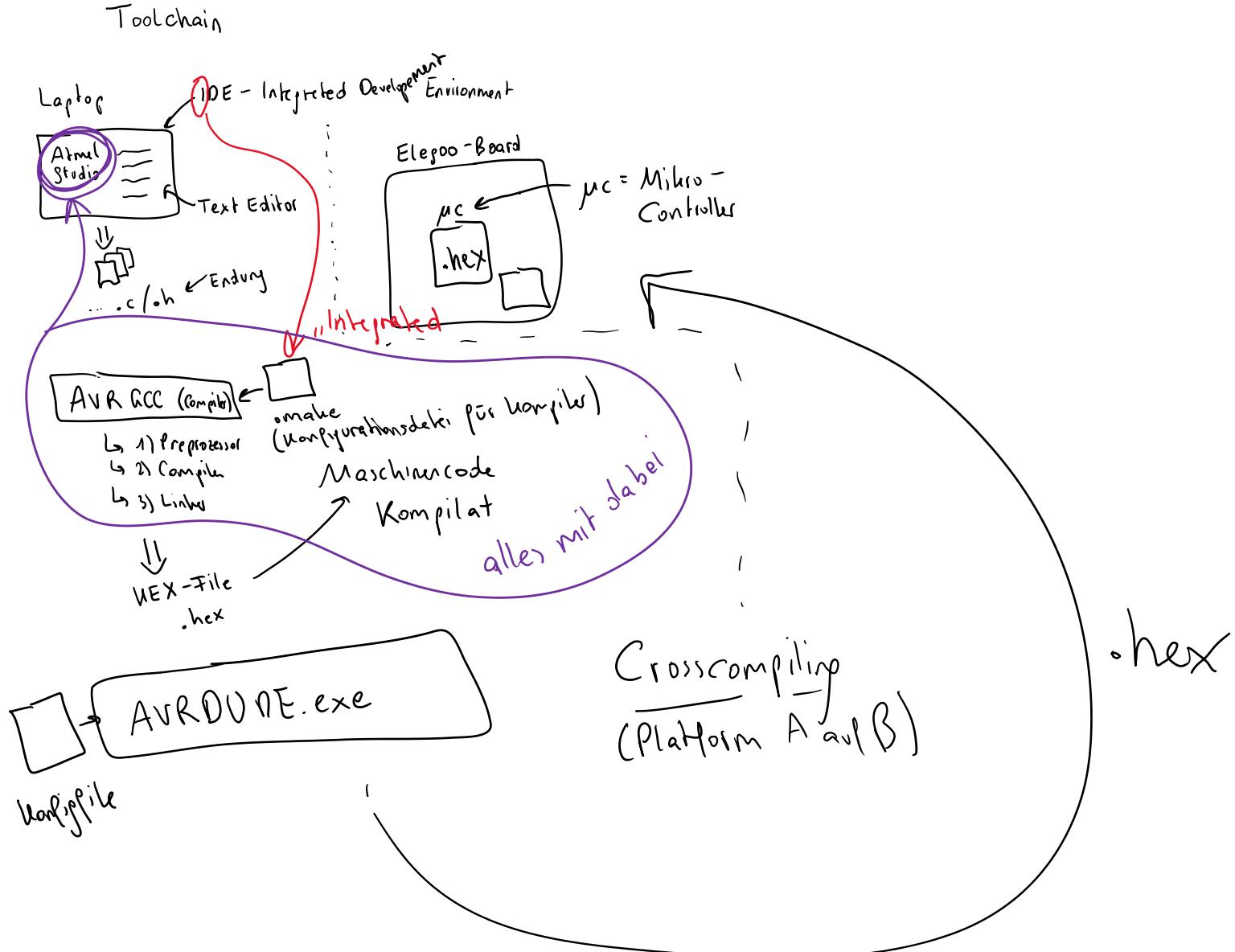
$\beta[7:0]$
Σ 8 pins

Elegoo B.

Verfügbarkeit d.
PINs ist ungleich

Toolchain

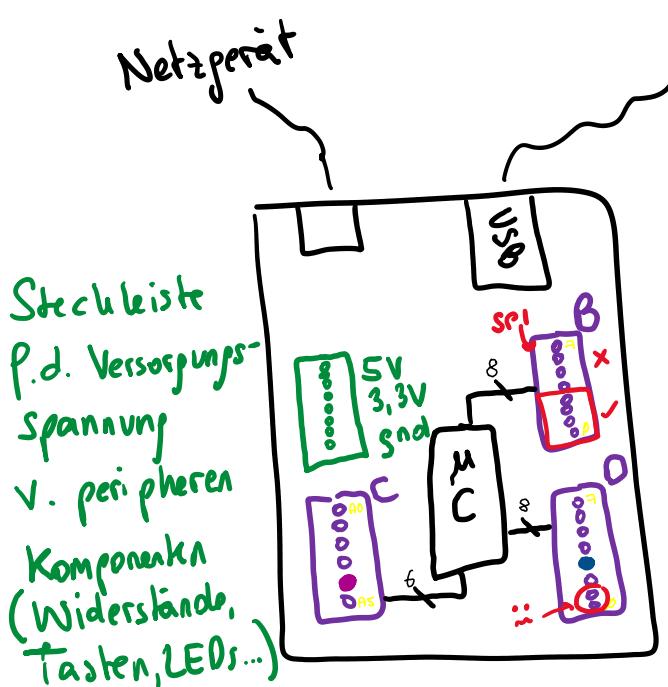
Freitag, 9. September 2022 14:09



Funktionsweise des Elegoo UNO R3 - Board

Mittwoch, 14. September 2022 09:53

Funktionsweise des Elegoo UNO R3 - BOARD



→ AVR-Familie: 8-Bit Architektur

ATmega 328P:

- + verfügt über 3 Ports (B, C, D)
- + PORT: dient als Schnittstelle, um Peripherie-Komponenten anzuschalten.

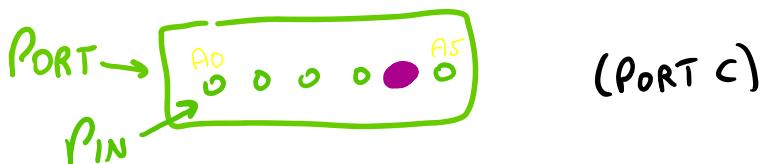
+ digitale / analoge Ports (B, C, D)
(C)

↓ ↓

I/O I/X
"nur Input"

5V-Logik!

PIN: einzelner Ein-/Ausgang
PORT: Gruppe von PINs

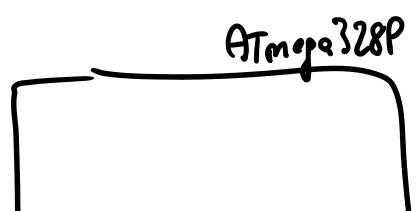


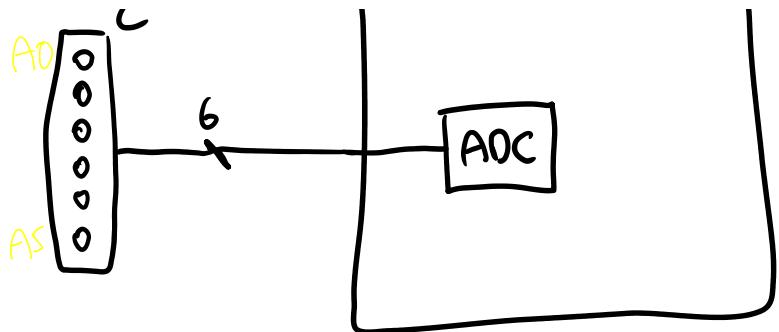
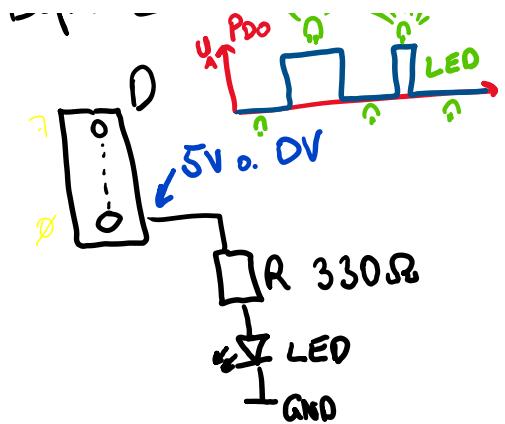
Bsp.: PC4 => Temp. Sensor
PD3 → PORT D

Bsp. LED:

D LED

↑
PDD

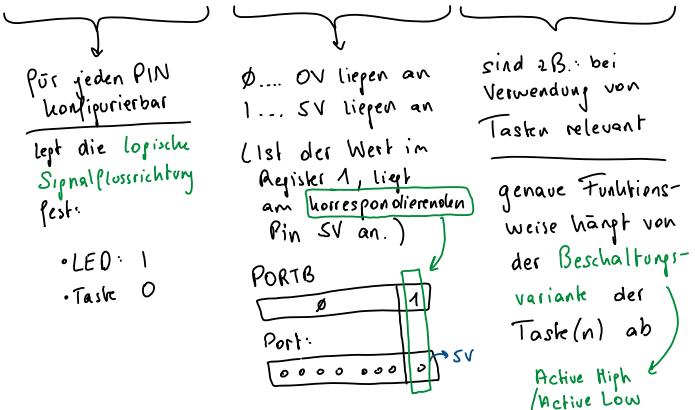




Wichtige Register (SFR) für digital I/O

Mittwoch, 28. September 2022 09:56

Registers Ports	Data Direction Register	Ausgangsregister	Eingangsregister
B	DDRB	PORTB	PINB
C	DDRC	PORTC	PINC
D	DDRD	PORTD	PIND
Anmerkung:	0...Eingang / 1...Ausgang		\cong Dataregister



=> ALLE REGISTER SIND 1 BYTE "BREIT"

=> Die Portx- und DDRx- Register werden default mit "Ø" initialisiert

=> C ist "Case Sensitive" => SFRs sind UPPER CASE

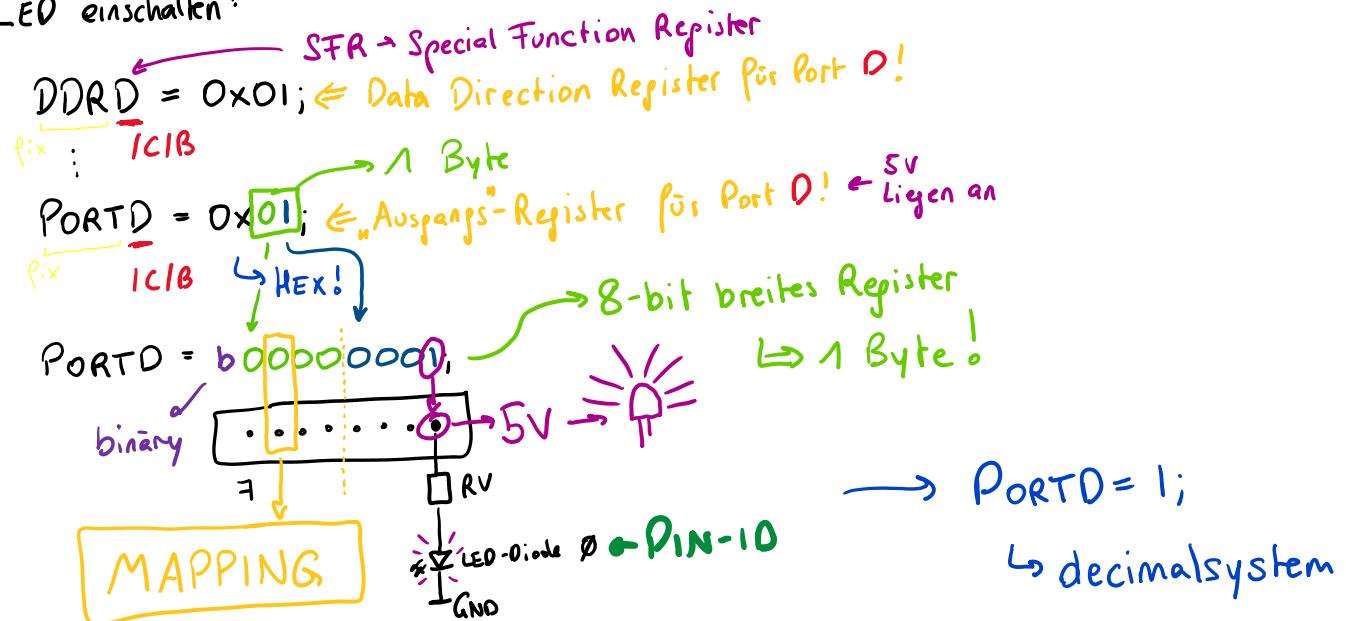
↓
io.h
(iom328p.h)
Header File

Funktionsweise eines Ports/Pins

Mittwoch, 21. September 2022 09:55

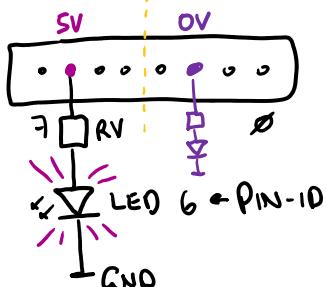
⇒ Digital Output!

LED einschalten:



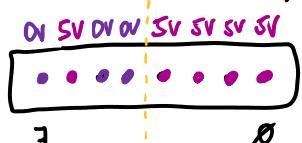
$$\text{PORTD} = 0x40;$$

$$\text{PORTD} = 0b01000000;$$



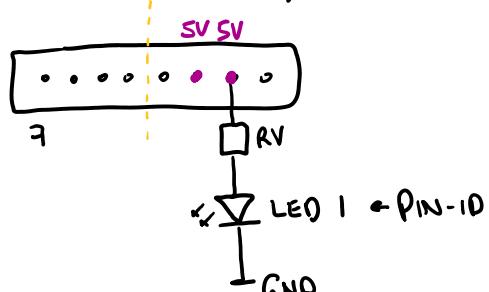
$$\text{PORTD} = 0x4F; \rightarrow \text{PORTD} = 79;$$

$$\text{PORTD} = 0b01001111;$$



$$\text{PORTD} = 0x06;$$

$$\text{PORTD} = 0b00000110;$$



Es Leuchten LED

0, 1, 2, 3 & 6

$$\text{PORTD} = 0x00;$$

→ Alle LEDs aus

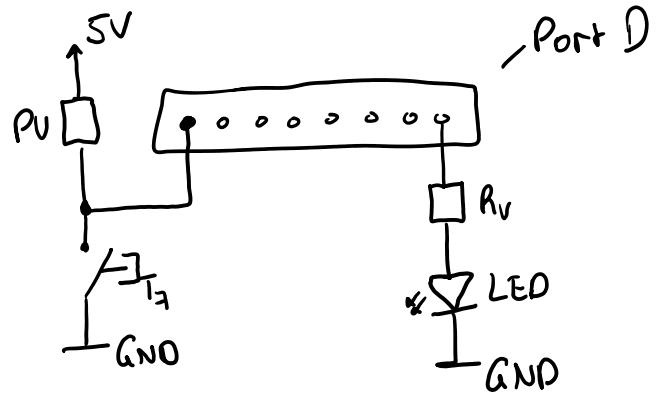
$\text{PORTD} = 0x00;$ → Alle LEDs aus

$\text{DDR D} = 0x01;$

„Data Direction Register“

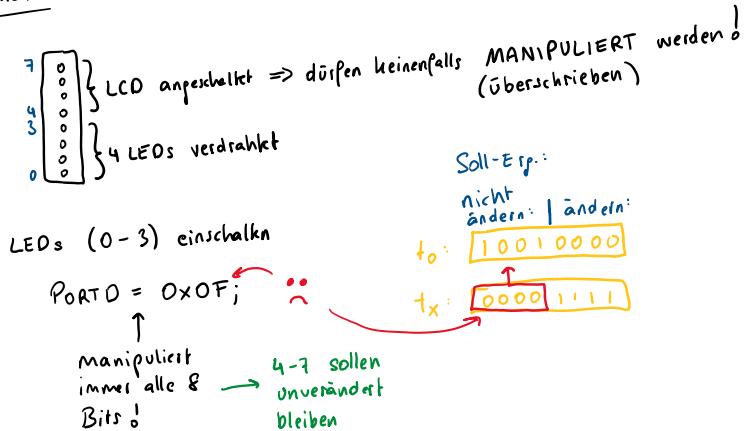
↓

Logische Signalflussrichtung



Tastensymbol:



Annahme:

Lösung: bitweise Registerzugriff!

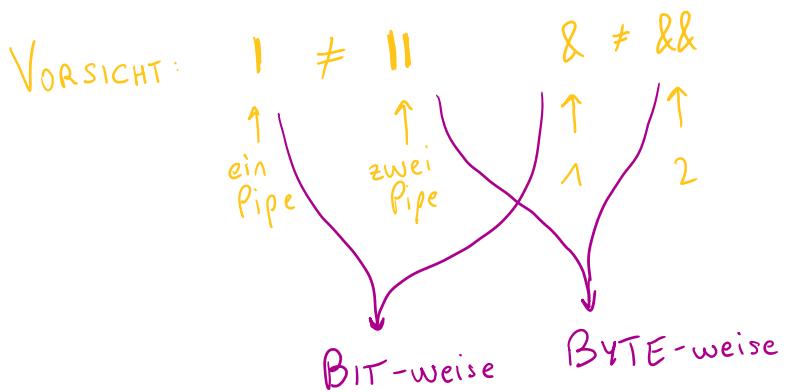
Bit-Operatoren:

„|“ Pipe $\hat{=}$ ODER-Verknüpfung und dient zum Setzen eines oder mehrerer Bits

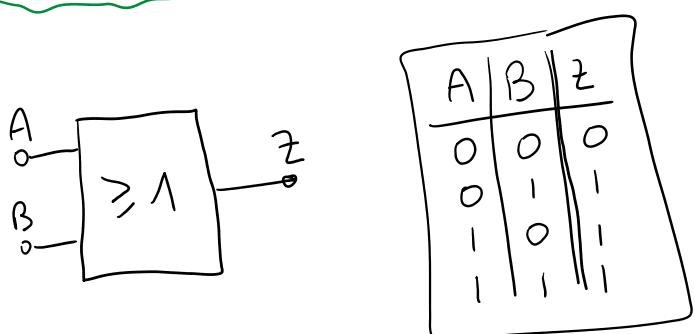
→ Übergang von 0 auf 1.

„&“ Ampersand $\hat{=}$ UND-Verknüpfung und dient zum Löschen eines oder mehrerer Bits

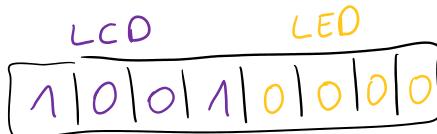
→ Übergang von 1 auf 0.



ODER-Verknüpfung



t_0 : PORTD

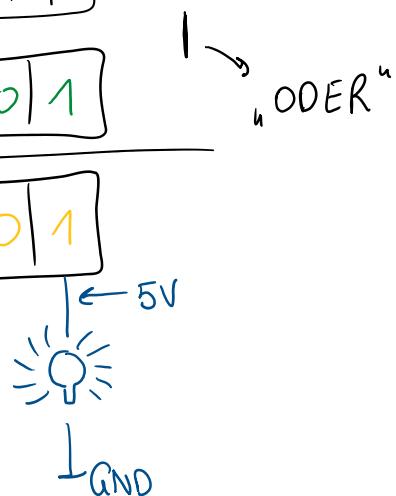
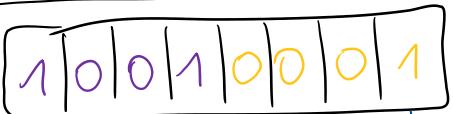


Ziel: LED0 einschalten
ohne LCD zu manipulieren

Maske:



t_1 :



CODE:

Lange Form: $PORTD = PORTD | 0x01;$

↑ ↑
 t_1 t_0

Maske
↓

Kurze Form: $PORTD |= 0x01;$

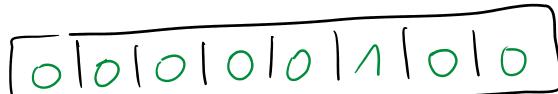
Beim Setzen eines Bits ist an jener Stelle eine "1" in der Maske vorzuschen, die auch im Ergebnis (Register) 1 sein soll!

Annahme: LED 2 u. 3 ein:

t_1 : PORTD



Maske LED2:



Maske LED3:

0|0|0|0|1|0|0|0

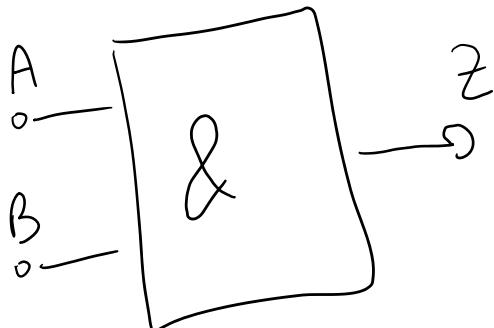
t_1 : PORTD

1|0|0|1|1|1|0|1

PORTD = PORTD | 0x04 | 0x08;

PORTD |= (0x04 | 0x08);

UND-Verknüpfung



A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

t_1 : PORTD

1|0|0|1|0|0|0|1

Maske:

1|1|1|1|1|1|1|0

t_2 : PORTD

1|0|0|1|0|0|0|0

CODE:

PORTD = PORTD & 0xFE;

Beim Löschen eines Bits ist an jener Stelle eine „0“ in der Maske vorzusehen, die auch im Ergebnis (Register) „0“ sein soll!

Toggle „ \wedge “ ... zirkumflex = XOR Operator

Bitweise Negation: \sim Tilde

Schiebeoperationen

Mittwoch, 5. Oktober 2022 10:02

- Schiebeoperator direkt an Port D
(z.B.: Lauflicht) angewandt:

<< um nach links zu verschieben

>> ... um nach rechts zu verschieben

// LED0 ein
PORTD = 0x01;
PORTD = PORTD << 1.
LED1 leuchtet
LED0 aus

MSB LSB → Least Significant Bit

alles im Byte um 1 Stelle (Bit)
nach links verschieben

Merkzettel: Bei der Verwendung des Schieboperators wird immer das ganze Byte manipuliert.

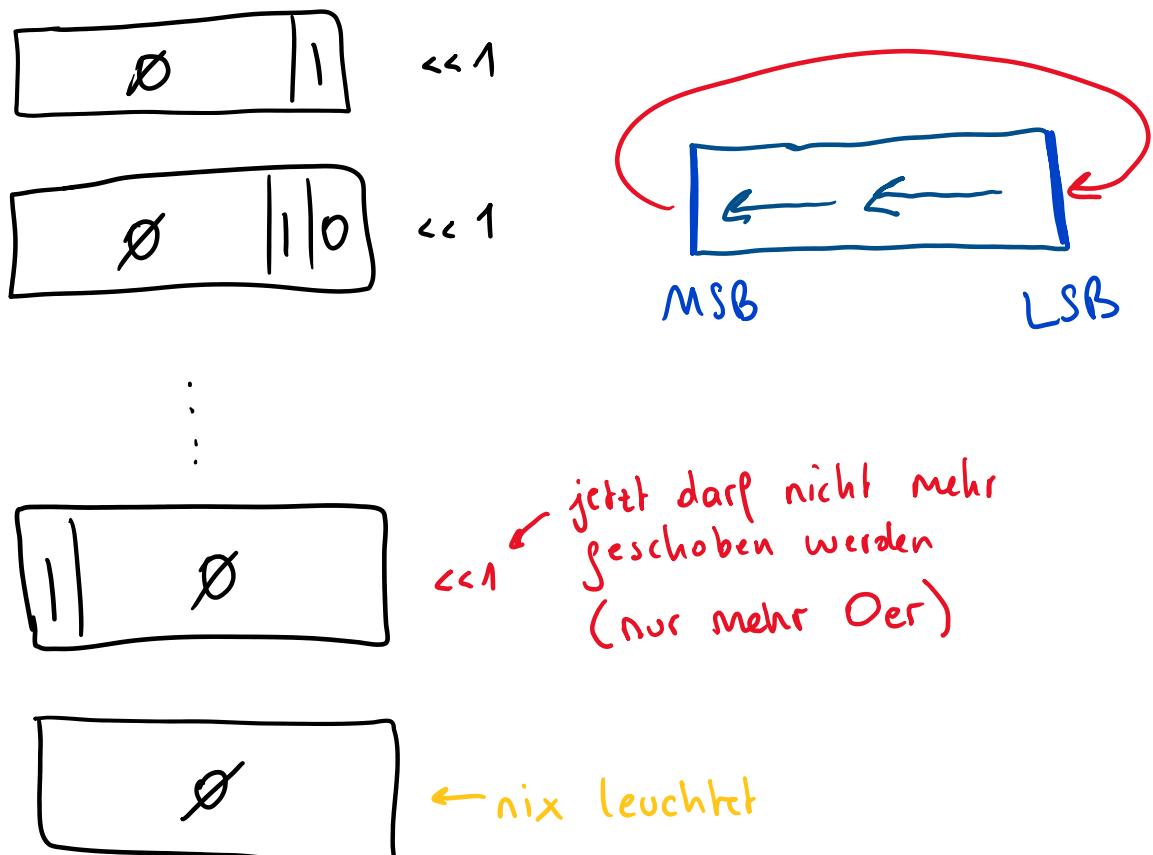
Merkzettel: Beim Nach-links-Schieben fällt das MSB raus, LSB wird mit 0 aufgefüllt, beim Nach-Recht-Schieben vice versa.

Bsp.:  ← kein SO verwenden!

(Lauflicht mit 4 LEDs)

Lauflicht mit 8 LEDs : SO kann angewandt werden

Port ist exklusiv zur Verfügung ↪



VORSICHT BEI DEN RÄNDERN!
(LSB & MSB)

```
if (PORTD == 0x80)  
    PORTD = 0x01;
```

- Schiebeoperator bei der Maskenerstellung (einschalten)

`PORTD |= 0x01; // LED0 einschalten`

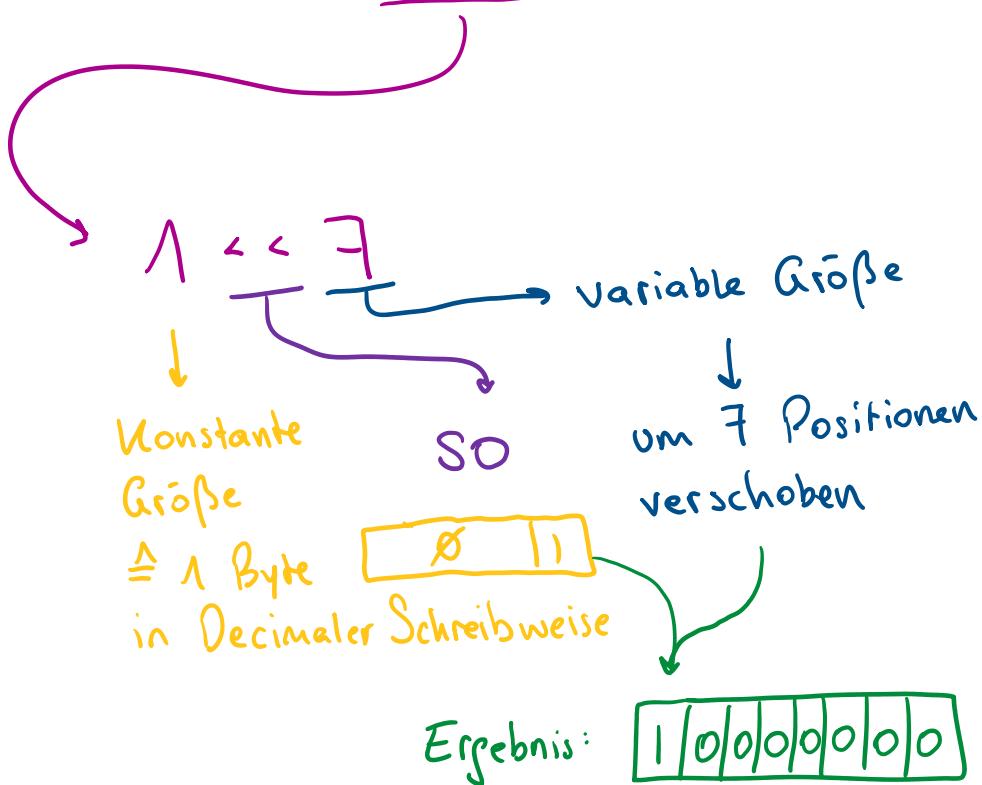
↓
Maske

`PORTD |= (1<<0); // LED0 einschalten`

`PORTD |= (1<<1); // LED1 einschalten`

⋮

`PORTD |= (1<<7); // LED7 einschalten`



allgemein: $(1 \ll n)$

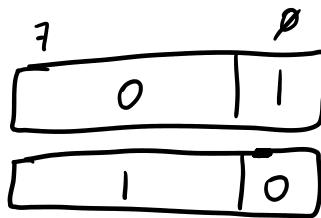
$n \dots$ Anzahl d. Positionen, die ich nach links verschieben will
(kann natürlich auch rechts sein)

- Schieberepler bei Maskenerstellung (löschen)

- Schieberegler bei Maskenerstellung (löschen)

LED Ø sehen \Rightarrow Maske

LED Ø löschen \Rightarrow Maske



Inverse
(invertiert)

$\text{PORTD} |= (\sim 1 \ll 0);$ // setzen

$\text{PORTD} \&= \sim(\sim 1 \ll 0);$ // löschen

invertierte
Maske

(bitweises
Invertieren
 \sim)

$\begin{array}{ccccccc} 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{array} 1$

$\begin{array}{ccccccc} 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array} 0$ $\Rightarrow 0x\text{FE};$

Merke: Prinzip der Maskenerstellung
bleibt gleich, Maske ist
jedoch bitweise zu invertieren

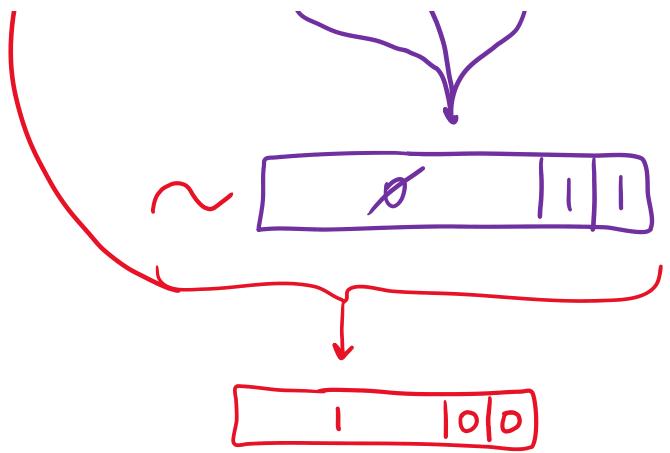
SONDERFALL:

Annahme: LED 0 und LED 1 ausschalten

$\text{PORTD} \&= \sim((1 \ll 0) | (1 \ll 1))$

$\begin{array}{cc} \emptyset & 1 \end{array}$

$\begin{array}{cc} \emptyset & 1 \mid 0 \end{array}$



(PORTD &= 0xFC;

LED ein-/auszuschalten

Mittwoch, 14. September 2022 10:27

Um meine LED ein-/auszuschalten:

1) Das PIN als Ausgang konfigurieren!!!

einmalig
n-Mal

DORD |= (1 << PORTD0)

2) LED einschalten

Arbeitsschleife

3) LED ausschalten

↳ Endlosloop

PORTD |= (1 << PORTD0);

PORTD &= ~(1 << PORTD0);

white(1) {
:
:
3}

Bitweises Abfragen/Auswerten von Digitalen Eingängen (Taste)

Freitag, 14. Oktober 2022 14:38

Bitweises Abfragen/Auswerten von Digitalen Eingängen (Taste)

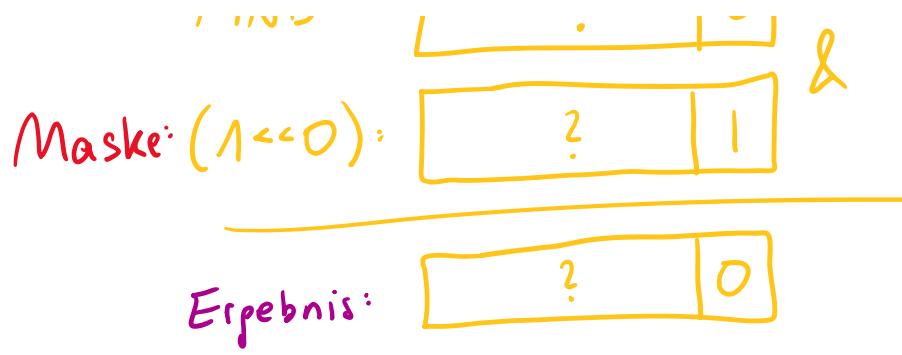
```
if ( PIND & (1 << PIND)) {  
    // do something  
}
```

≤1 wenn Taste gedrückt
0 wenn Taste nicht gedrückt

A	B	Z
0	0	0
0	1	0
1	0	0
1	1	1

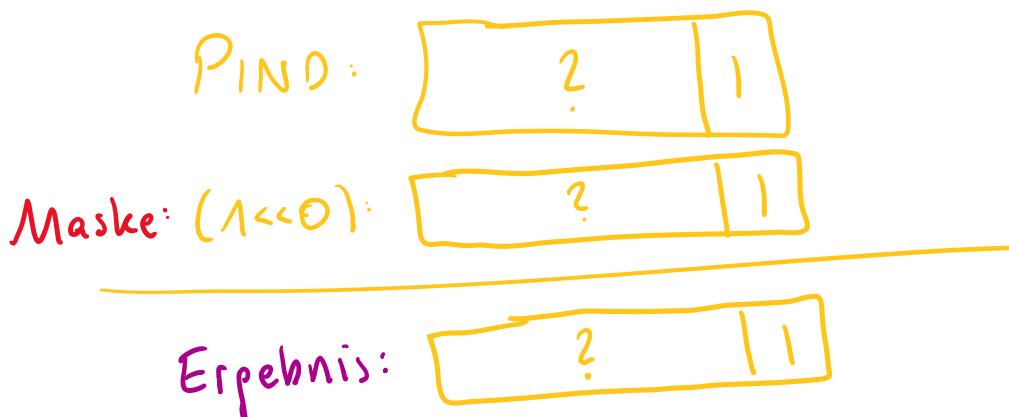
Taste nicht betätigt:

PIND: 



\hookrightarrow if nicht erfüllt! $\therefore (ip(0))$

Task befüllt:



\hookrightarrow if erfüllt! $\therefore (ip(1))$

Digital Input mit Tasten

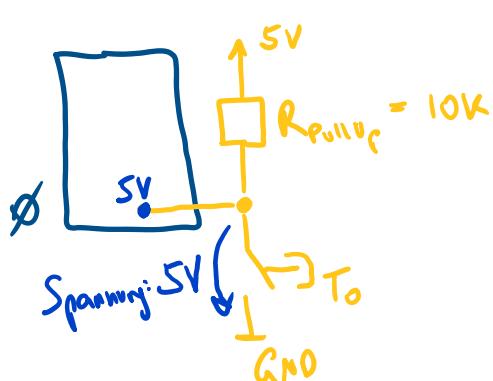
Mittwoch, 12. Oktober 2022 10:15

Man unterscheidet 2 Beschaltungsvarianten:

- Pull-Up-Widerstand \Rightarrow Active-Low-Beschaltung
- Pull-Down-Widerstand \Rightarrow Active-High-Beschaltung

PULL-UP - BESCHREIBUNG

PORTD (eigentlich nicht geeignet)



high-potential (5V)

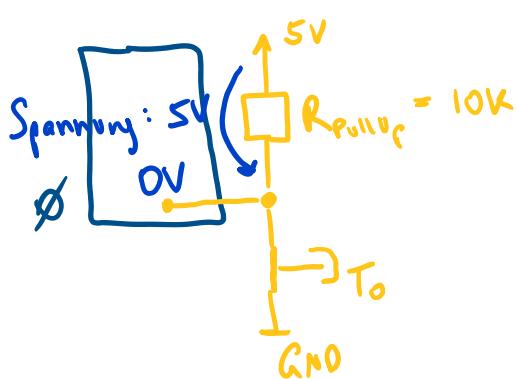


T_0 nicht betätigt:
5V liegen am
Eingang PD∅ an

Schalsymbol



PORTD (eigentlich nicht geeignet)



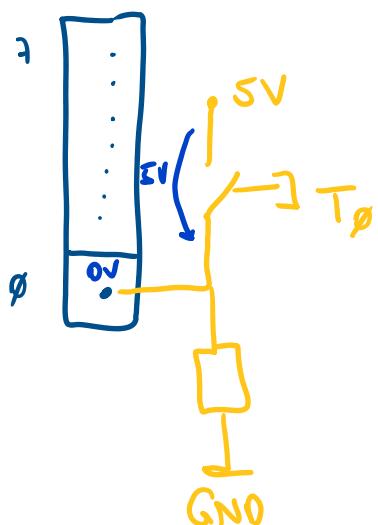
low-potential (0V)

T_0 betätigt:
0V liegen am
Eingang PD∅ an

Ist die Task betätigt, dann liegt das Low-Potenzial am Eingang an (\rightarrow Active-Low-Beschaltung)

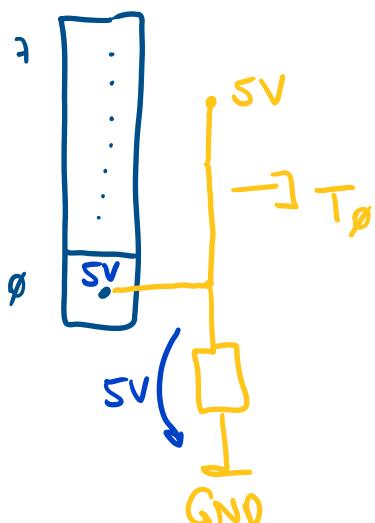
PULL-DOWN - BESCHREIBUNG

PORTD



high-potential
(5V)

T_Ø nicht betätigt:
OV liegen am
Eingang PDØ an

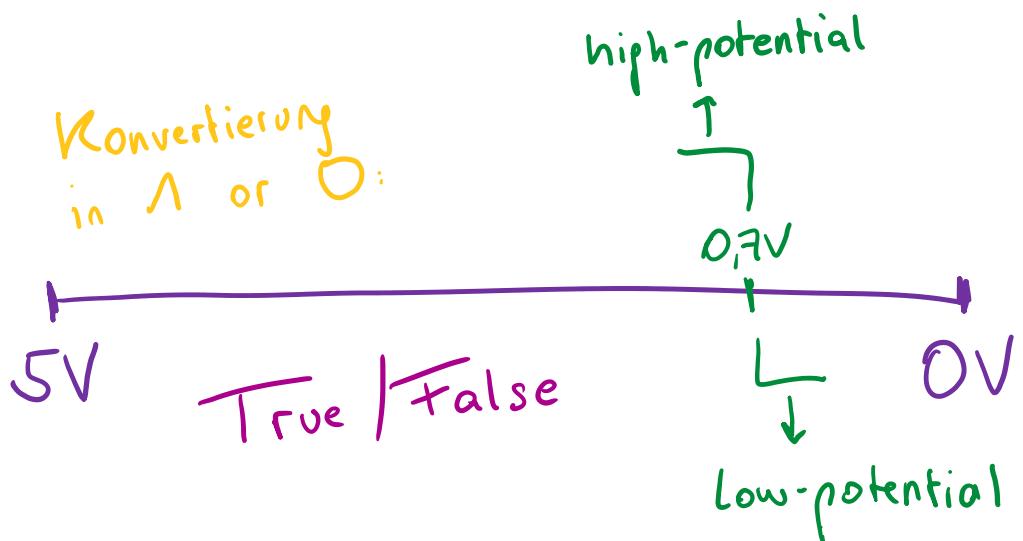


low-potential
(OV)

T_Ø betätigt:
SV liegen am
Eingang PDØ an.

Ist die Taste betätigt, dann liegt das
- - - PDØ - - -

Ist die Taste betätigt, dann liegt aus
high-Potential am Eingang P00 an
(Active-High-Beschaltung)



Floating Pin: Kurtschluss:



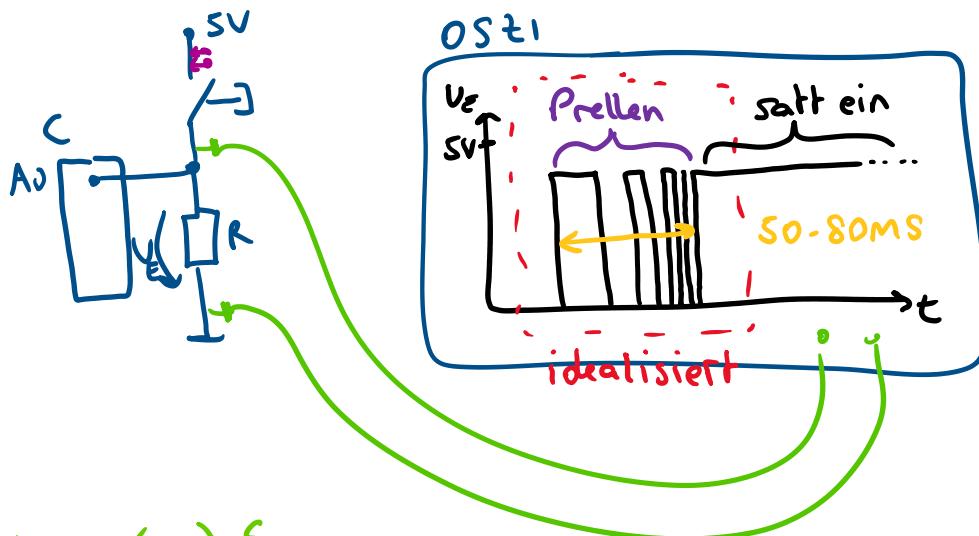
Tasten prellen (Bouncing)



Taste „schwingt“ ganz kurz
hin/her + auf/zu aufrund
↓ I oder → Pausen



hin/wie → **PRELLEN**
der Feder. → PREllen



while (1) {

 if (PINC & (1<<PINCO)) {

 _delay_ms(70);

$$P = \frac{1}{T} \quad T = \frac{1}{P} = \frac{1}{16\text{MHz}}$$

sehr
opt

}

Lösung:

hardware:

entprellte
Taster

/ Kondensator
+ Schmitttrigger

software

_delay_ms(70);

Verarbeitung einer Taste auf CODE-Ebene

Freitag, 14. Oktober 2022 14:26

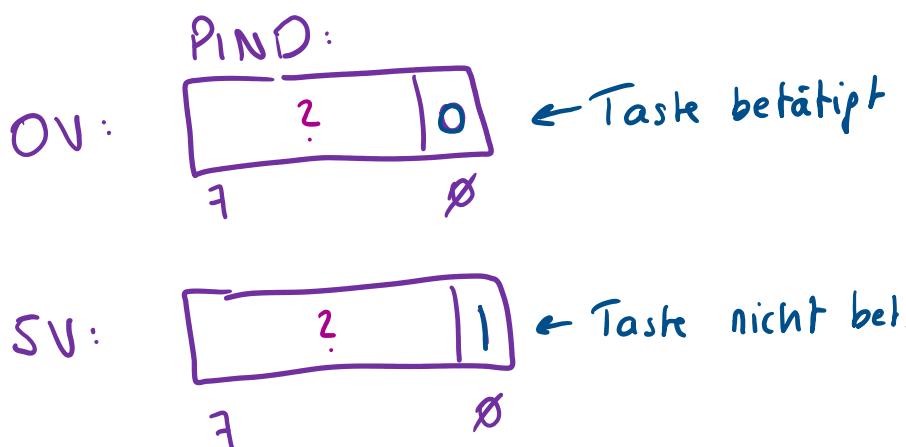
Verarbeitung einer Taste auf CODE-Ebene:

PIN_x - Register $x = \{B_i; C_i; D\}$

Input

Das jeweilige PIN als Eingang konfigurieren (DDR_x)
↳ default: 0 (Input)

ACTIVE-HIGH-BESCHALTUNG



if ($PIN0 \& (1 \ll PIN0)$) { Tastendruck: 200ms

; //do something

} else {} }

$$T = \frac{1}{P} = \frac{1}{16\text{MHz}}$$

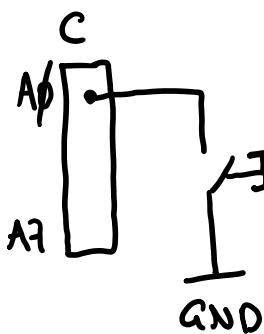


Active Low mit dem internen Pull-Up Widerständen

Freitag, 4. November 2022 14:16

Active Low mit dem internen Pull-Up Widerständen

⇒ Beschaltung reduziert sich auf die Taste



Wichtig: Du Widerstände
sind zu aktivieren!
PIN für PIN

Hierzu dienen die PORTx-Register
(z.B.: PORTC)

PIN als Eingang konfiguriert sein.

↓ (default)

Bsp.:

$\boxed{\begin{array}{l} \text{DDRC \&} \\ \sim(1 \ll \text{DDCO}); \\ \\ \text{PORTC I = } +1 \\ (1 \ll \text{PORTC}\phi); \end{array}}$

↑
PULL UP für PC ϕ aktivieren!

//ToDo:

- Code Ebene
 - + 3x Pull up WS aktivieren (PC ϕ , PC1, PC2)
 - + Kontrollstruktur umbauen

→ falls es nur einen Wert gibt, kann man das weglassen

+ Kontrollstruktur umbauen

Bsp.: $PC \neq \emptyset \Rightarrow T \rightarrow PINC$ 

betätigt nicht betätigt
↓ if ($!(1)$) {}

$T \rightarrow PINC$  if ($!(PINC \&= (1 \ll PINC \neq))$) {}
if ($!(0)$) {}

oder:

if ($((PINC \&= (1 \ll PINC \neq)) == \emptyset)$) {}

• Hardware Ebene

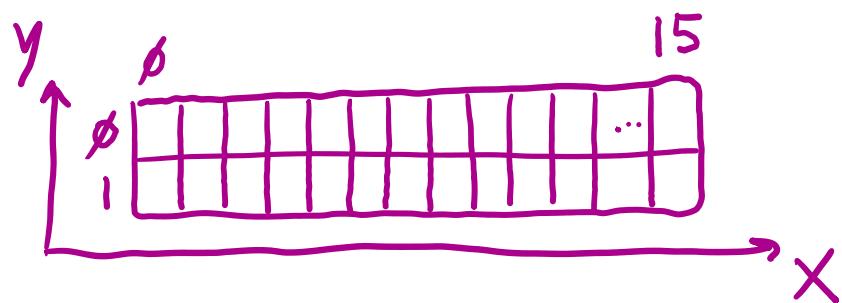
+ Widerstand ausbauen

+ SV Versorgungsspannung entfernen

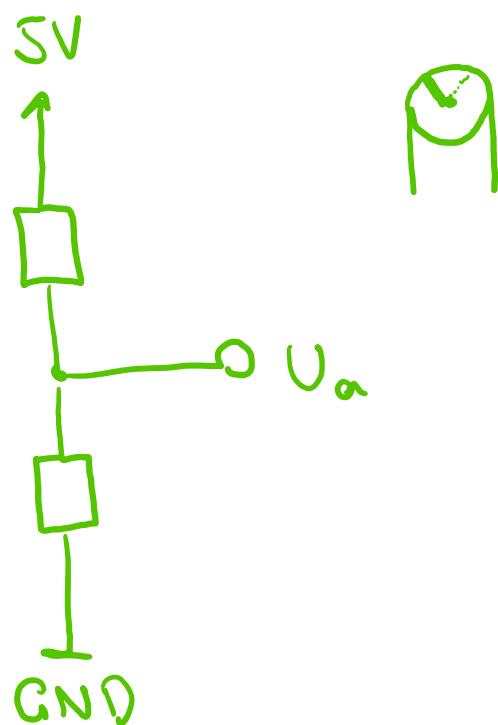
Liquid Crystal Display

Freitag, 18. November 2022 14:09

Liquid Crystal Display



POTI:



Externe Interrupts (Hardware-Lösung)

void main() {

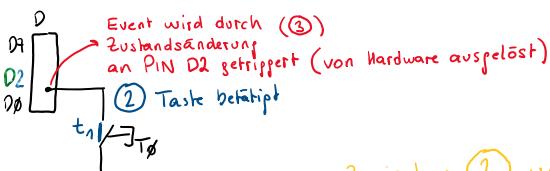
while(1) {

}

}

{ Hauptpfad to

① Loop



zwischen ② und ③
wird das Interrupt-
Event gepeuert

⑥ Controller kehrt in
Hauptpfad zurück
⑤ ... und führt
diese aus

Vektor ist variabel

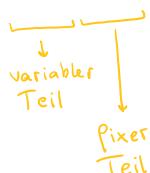
ISR (...) {
 // Interrupt Service Routine
 // „„ bzw. vergleichbar mit Funktion
 // Bezeichnung ISR = konstant

zwischen ③ und ④
speichert Context
(damit weiß, wohin
zurück) in RAM

⑦ Lädt Context
und setzt
Ausführung des
Hauptpfades
fort.

Vectorbezeichnung:

D2 → INT0-vect



HTL Krems - SYTI

ATmega328P & External Interrupts – How to

Interrupts unterbrechen die "normale" Programmausführung, um eine andere, meist zeitkritische Verarbeitung durchzuführen. Als Sprungziel dient eine sog. Interrupt Service Routine (ISR) mit entsprechendem Interrupt-Vector (=fixe Adresse im Speicher) als Parameter.

Die externen Interrupts des ATmega328P:

• INT0 und INT1 → direkte ext. Interrupts → 1:1-Zuordnung	→ 8/16/8: 1 Zuordnung (B/C/D)
• PCINT0...PCINT3	gruppenbasierter Ext. Interrupts

Pinout ATmega328P:

PCINT14 RESET PB0	PCINT15 SREG PB1	PCINT16 ADC0 PB2	PCINT17 ADC1 PB3	PCINT18 ADC2 PB4	PCINT19 ADC3 PB5	PCINT20 ADC4 PB6	PCINT21 ADC5 PB7	PCINT22 ADC6 PB8	PCINT23 ADC7 PB9
PCINT15 T0CKI PB0	PCINT16 T0CKI PB1	PCINT17 T0CKI PB2	PCINT18 T0CKI PB3	PCINT19 T0CKI PB4	PCINT20 T0CKI PB5	PCINT21 T0CKI PB6	PCINT22 T0CKI PB7	PCINT23 T0CKI PB8	
PCINT24 T0OFI PB0	PCINT25 T0OFI PB1	PCINT26 T0OFI PB2	PCINT27 T0OFI PB3	PCINT28 T0OFI PB4	PCINT29 T0OFI PB5	PCINT30 T0OFI PB6	PCINT31 T0OFI PB7	PCINT32 T0OFI PB8	
PCINT24 CKE PB0	PCINT25 CKE PB1	PCINT26 CKE PB2	PCINT27 CKE PB3	PCINT28 CKE PB4	PCINT29 CKE PB5	PCINT30 CKE PB6	PCINT31 CKE PB7	PCINT32 CKE PB8	
PCINT24 ICP0 PB0	PCINT25 ICP0 PB1	PCINT26 ICP0 PB2	PCINT27 ICP0 PB3	PCINT28 ICP0 PB4	PCINT29 ICP0 PB5	PCINT30 ICP0 PB6	PCINT31 ICP0 PB7	PCINT32 ICP0 PB8	

INT0 und INT1 im Detail

Möchte man die externen Interrupts INT0 und INT1 verwenden, gilt es folgende Register zu manipulieren:

EIMSK – External Interrupt Mask Register

Dieses Register dient zur Aktivierung des gewünschten Interrupts:

bit	7	6	5	4	3	2	1	0	INT0	INT1	EIMSK
ResetValue	0	0	0	0	0	0	0	0	0	0	0

Es gilt: 1 = aktiviert

Voraussetzung: Interrupts müssen global aktiviert sein (siehe I-Flag des Registers SREG, Datenblatt S. 11).

Pin: PD2 ⇒ ISR(INT0-vect)



ISR(INT0-vect) {
 // Weiß, dass Zustandsänderung
 // an PD2, weil INT0-vekt
 // 1:1 Zuordnung

 // Weiß, dass PD2

} verschiedene
ISRs

Konfiguration von INT0 & INT1:

main() {
 EIMSK |= (1 << INT0) | (1 << INT1);

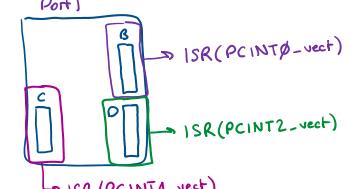
 ↑ aktivieren
 & INT0

 ↑ aktivieren
 Bit 1

sei(); // Interrupt "global" aktivieren
 // Makro rgl. Funktion

 // SREG-Register Bit 7 setzen!

while(1) { ... }

8/6/8 : 1-Zuordnung
(PINs v Port)

Alle PINs eines Ports werden einer ISR
zuordnet. Beim Ausführen d. ISR
muss man prüfen, welches PIN des
Ports eine Zustandsänderung erfahren hat.

ISR(PCINT1-vekt) {
 if (!Taste an PIN0) {}
 else if (!Taste an PIN1) {}

EICRA – External Interrupt Control Register A
Mit diesem Register werden die „Auslösebedingungen“ festgelegt (siehe Datenblatt, S.73).

Bit	7	6	5	4	INT1	INT0
Initial Value	0	0	0	0	0000	0000
Reset Value	0	0	0	0	0000	0000
Reading	0	0	0	0	0000	0000

Auslösebedingungen für INT0:

ISCO1	ISCO0	Description
0	0	The low level of INT0 generates an interrupt request.
0	1	Any logical change on INT0 generates an interrupt request.
1	0	The falling edge of INT0 generates an interrupt request.
1	1	The rising edge of INT0 generates an interrupt request.

Implementierung der ISR

```
Arbeitsblatt: exercise_01
Arbeitsblatt: exercise_01.c

/*
Interrupt-Vektor: http://www.nexys4arm.com/learn/group__avr__interrupts.html#def02099a2d422cf39e2ba3056289f

int main(void)
{
    sei(); // Makro
    while(1)
    {
        /* Aktionen, die nach dem Setzen */
    }
}

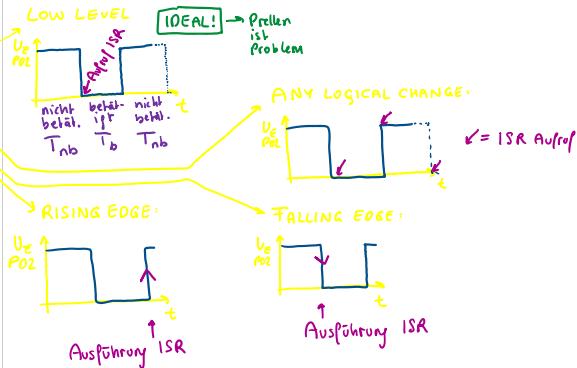
ISR(INT0_vect)
{
    /* Interrupt-Code für INT0 */
}

ISR(INT1_vect)
{
    /* Interrupt-Code für INT1 */
}
```

Sei(); // Interrupt „global“ aktivieren
// Makro rgl. Funktion → SRef-Register Bit 7 setzen!

while(1) { ... }
}ISR(INT0_vect) { ... }

PULL UPS aktivieren!

**PCINT[0:23] Im Detail**

Im Gegensatz zu INT0 und INT1 gibt es keine Möglichkeit, um die Auslösebedingungen zu definieren. Es gilt das Prinzip „join change“. Eine eindeutige ISR-Zuordnung ist ebenfalls nicht gegeben. Diesbezüglich gilt: ↗ PC (keine 1:1-Zuordnung)

Pin	ISR-Vector
B	PCINT[7..0] ISR ((INT0_vect))
C	PCINT[14..8] ISR ((PCINT1_vect))
D	PCINT[23..16] ISR ((PCINT2_vect))

Tabelle 1: Zuordnung „Pin - ISR-Vector“

Im Folgenden wird die Funktionsweise des PCINT[0:23]-Registers übersichtsartig dargelegt.

PCICR – Pin Change Interrupt Control Register

Mit dem PCICR kann eine beliebige Interrupt-Gruppe (vgl. Tabelle 1) aktiviert werden.

Bit	7	6	5	4	3	2	1	0	PCIE0	PCIE1	PCIE2	PCIR
Initial Value	-	-	-	-	-	-	-	-	PCIE0	PCIE1	PCIE2	PCIR
Reset Value	0	0	0	0	0	0	0	0	0	0	0	0

Es gilt:

- 1 = aktiviert

Weiteres gilt:

- PCIE0 = PCINT[7..0]
- PCIE1 = PCINT[14..8]
- PCIE2 = PCINT[23..16]

PCMSKx – Pin Change Mask Register x [0:2]

Um einen Eingang/Ausgang (=Pin) zu aktivieren, dienen die Register PCMSK0, PCMSK1 und PCMSK2.

Bit	7	6	5	4	3	2	1	0	PCMSK0
Initial Value	0	0	0	0	0	0	0	0	0
Reset Value	0	0	0	0	0	0	0	0	0

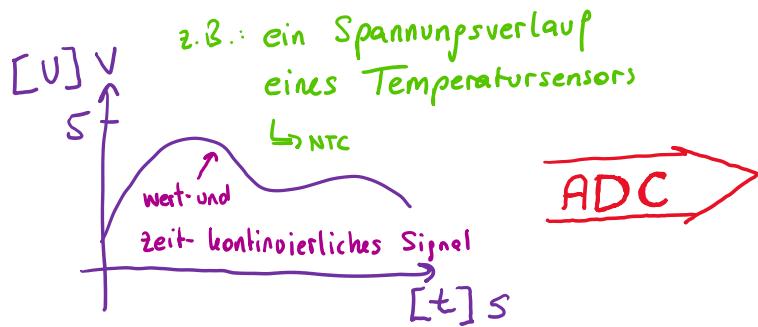
Es gilt:

- PCMSK0 = PCINT[7..0]
- PCMSK1 = PCINT[14..8]
- PCMSK2 = PCINT[23..16]

→ Für weiterführende Informationen siehe Datenblatt, Kapitel 13.

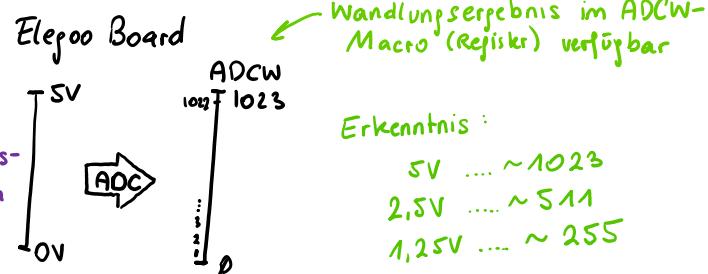
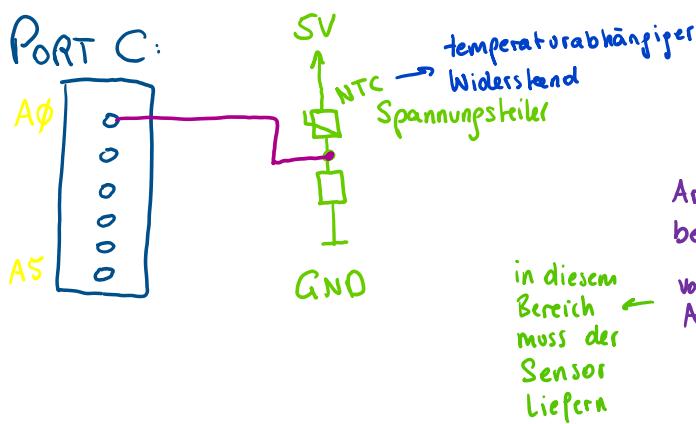
ADC - Analog Digital Converter

Zweck: Wandelt ein analoges Signal (Größe) in einen digitalen Wert um.



0 Binärzahl,
1 die beim
ATmega 328P
10 Bit umfasst

Auflösung
Resolution

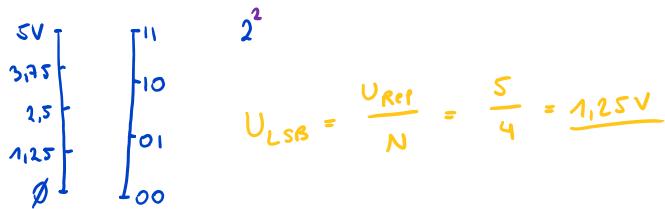


Erkenntnis:
5V ~1023
2,5V ~511
1,25V ~255

$$U_{LSB} = \frac{U_{ref}}{N} = \frac{5V}{1024} \approx 4,88mV$$

VERSTÄNDNIS:

Annahme: 2 bit ADC

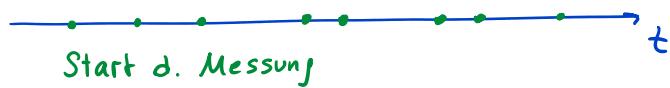


Betriebsarten

Single Conversion Mode

→ einzelne Messungen sind auf Programmebene zu starten

↳ wiederholte, zyklische Messungen möglich!



ADSC - Bit → Start Conversion

Ablauf / Prinzip:

1. Messung starten
 2. warten bis Ergebnis vorliegt
 3. Ergebnis auslesen
1. Messung starten (kann auch mit ISR iteriert werden)

⋮

Free Running Mode

→ zyklische Messungen werden
automatisch ausgeführt

→ Messzyklen (Intervalle)
können nicht beeinflusst werden



Messungen starten

↳ Δt ist gleich

Ablauf / Prinzip:

1. erste Messung muss
auf Programmeebene
gestartet werden
 2. nachfolgende Messungen
werden automatisch
- ⋮

2. nachfolgende Messungen
werden automatisch
gestartet

↳ so schnell, wie der
ADC kann

3. nach Kanalwechsel

↳ erste Messung weg

↳ oder Neustart

richtig einstellen:

ADCSRA: ADATE setzen

Code:

ADCSRA |= (1 << ADATE);

im ADCSRB ist nichts zu setzen, weil
für Free Running Mode AOTSO - AOTS1
0 sein müssen.

externer Trigger Mode

→ zyklische und azyklische
Messungen möglich

Verzahnung v. Timer und ADC



der Timer startet eine Messung

Timer Event startet Messung
im bestimmten Zeitintervall Δt
ist auf Timerebene konfigurierbar

Ablauf / Prinzip:

1. Konfiguration:

$T_{\text{----}}$

1. Konfiguration:
 - a. Timer
 - b. ADCSRA
 - c. ADCSRB
2. Messwerte verarbeiten
[in der ISR(ADC_vect) {}]



ADC II - Grundprinzipien der AD-Wandlung

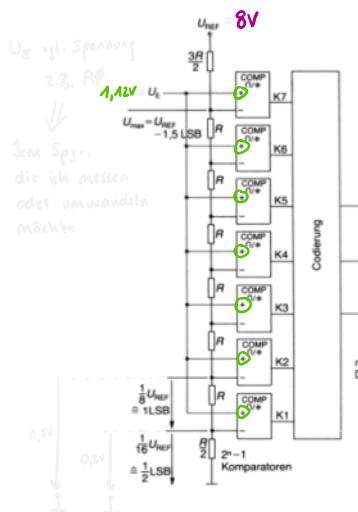
Grundsätzlich unterscheidet man zwei Klassen von Wandlungsverfahren:

- nicht integrierende und \Rightarrow direktes Verfahren
 - z.B. Parallel- oder Wägeverfahren
 - integrierende Verfahren. \Rightarrow indirekter Vergleich
 - z.B. Single- oder Dual Slope

der Vergleich mit der zu messenden Größe / Spannung
Ermittlung d. Spannung erfolgt über die Zeit

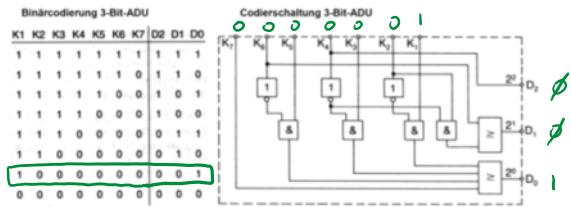
Parallelverfahren

Ein Vertreter des Parallelverfahrens ist der *Flash Converter*. Dieses Verfahren ist besonders schnell, da die Wandlung in einem Arbeitsschritt erfolgt. Daher auch die Bezeichnung *Flash*. Nachteilig wirkt sich der Hardwareaufwand aus, da es n-viele „innere“ Referenzgrößen (und somit Komparatoren) braucht.



Nebenstehende Abbildung zeigt einen **3 Bit-Flash** Converter. Mit einer Spannungsteilerkette werden von der Referenzspannung U_{REF} ausgehend alle erforderlichen Spannungsstufen abgeteilt und den negativen Eingängen der Komparatoren zugeführt. Die Eingangsspannung U_E – also die zu wandelnde Spannung – wird an die positiven Eingänge geschaltet. Ist nun die Spannung am positiven Eingang eines Komparators größer als am negativen, liefert dieser „logisch“ 1 am Ausgang, sonst „logisch“ 0. Die Auswertung der Komparatoren erfolgt gleichzeitig (vgl. Füllstandsanzeige). Der Dual-Code wird von einer nachfolgenden Codierschaltung gebildet.

Wie viele Komparatoren würde der ADC des ATmega328P bei dieser Bauweise benötigen?

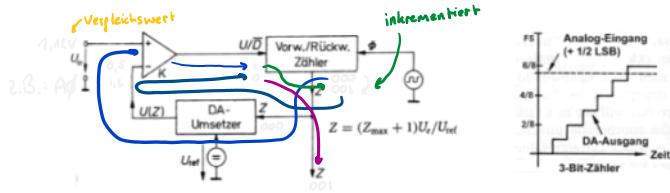


Wägeverfahren

Das einfachste Prinzip dieses Verfahrens ist mit jenem einer Balkenwaage vergleichbar. Es werden solange entsprechende Gewichte aufgelegt, bis die Wage ins Gleichgewicht bzw. diesem nahe kommt.

Nachlaufverfahren

Die Idee dieses Verfahrens beruht auf der schrittweisen Erhöhung einer „Vergleichsspannung“ (siehe U(Z)), welche kontinuierlich mit der Eingangsspannung U_e verglichen wird. Ist $U_e > U(Z)$, wird der Zählerstand (Zahl Z) mit dem nächsten Takt um 1 erhöht, und zwar solange, bis $U(Z)$ erstmalig größer als U_e wird. Als Folge darauf liefert der Komparator als Vergleichsergebnis 0, da die Spannung am negativen Eingang größer ist als am positiven. Konsequenz: der Zähler stoppt, und mit dem aktuellen Z-Wert steht das Ergebnis als Digitalzahl bereit. Der Vollständigkeit halber: Der DA (Digital Analog)-Wandler wandelt die jeweilige Zahl Z in eine entsprechende Spannung ($U(Z)$) um.

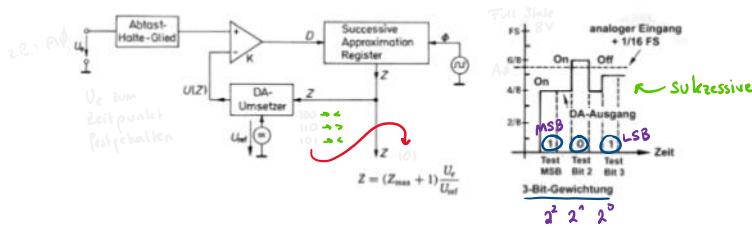


Als suboptimal ist die unterschiedliche Wandlungsdauer zu werten, die von der Größe der zu wandelnden Spannung (U_e) abhängt.

Sukzessive Approximation

Bei der Sukzessiven Approximation vergleicht der Komparator K ebenfalls die vom DA-Wandler bereitgestellte Ausgangsspannung $U(Z)$ mit der Eingangsspannung U_e . Jedoch wird $U(Z)$ nicht durch einfaches Inkrementieren eines Zählers generiert, sondern durch – welch' Überraschung – **Approximation**. Im ersten Schritt wird das höchste Bit (MSB) gesetzt und geprüft, ob $U_e > U(Z)$ ist. Trifft das zu, bleibt das MSB

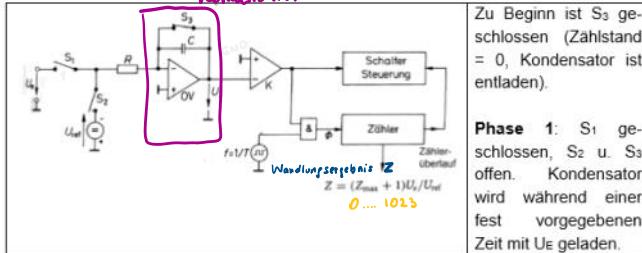
gesetzt. Andernfalls wird es wieder gelöscht. Dieser Wägevorgang wird anschließend für alle weiteren Bits wiederholt, und zwar bis zum LSB. $MSB \rightarrow LSB$



Dual Slope Verfahren

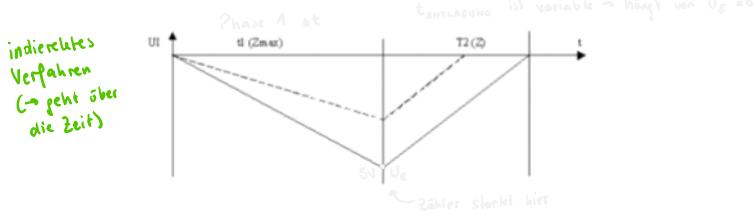
Beim Dual Slope-Verfahren wird nicht die Eingangsspannung selbst gemessen oder verglichen – was übrigens für alle Vertreter dieser Klasse zählt – sondern eine zur Eingangsspannung proportionale Größe. Dieses Verfahren wird bspw. bei Digitalvoltmetern (Multimeter) eingesetzt.

Kondensator



Dabei entsteht eine negative, linear abfallende Spannung U_i . Die vorgegebene Zeit entspricht i.d.R. dem Maximalzählwert des Zählers (Z_{max}).

Mit Überlauf ($Z_{max} + 1$) beginnt **Phase 2**, die Entladung des Kondensators durch entgegengesetzte gepolte Referenzspannung U_{ref} (S_1 : u. S_2 offen, S_3 geschlossen). Die Integratorspannung U_i steigt wieder an, und zwar bis 0 erreicht ist. Phase 2 ist somit abgeschlossen, der Komparator (K) geht auf 0 zurück und stoppt damit die Zählimpulse. Das Ergebnis steht mit der Anzahl der Zählimpulse, die während der Zeit T_2 generiert wurden, bereit. Die Zählimpulse sind proportional zur Eingangsspannung U_e .



Genaugigkeit und Fehler

Durch die Quantisierung entsteht ein Informationsverlust, der nicht mehr rückgängig gemacht werden kann. In diesem Zusammenhang spricht man vom sog. Quantisierungsfehler!

Wichtig: Die Auflösung ist in der Tat eine wichtige Kenngröße eines ADC, sagt aber nichts über die eigentliche Genaugkeit aus. Denn die Genaugkeit eines ADC wird durch die Auflösung und die SUMME aller Fehler (Quantisierungs-, Offset-, Verstärkungs-, Linearitäts- und Umsetzfehler sowie die differentielle Nichtlinearität) bestimmt.

Abtasttheorem oder Nyquist-Shannon-Theorem

Das Abtast-Theorem besagt, dass die Abtastfrequenz beim Digitalisieren **mehr** als doppelt so hoch sein muss, wie die **höchste im kontinuierlichen Signal enthaltene Frequenz**.

Literatur

Tietze, U; Schenk, Ch.: Halbleiterschaltungstechnik. Berlin. Springer Verlag.
Beuth, K.: Digitaltechnik – Elektronik 4. Würzburg. Vogel Verlag.

ziel: Signal digitalisieren \Rightarrow ADC

Mitko

Form /
Signal
sollte
erhalten
bleiben

bei rosa Punkten tastet man ab \rightarrow relativ penes Ergebnis
bei grünen Punkten tastet man ab \rightarrow äußerst unpenes Ergebnis
 \hookrightarrow zu geringe Frequenz (\hookrightarrow Verlureng Theorems)

Timer Controller / Counter

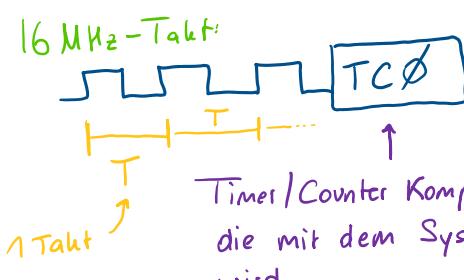
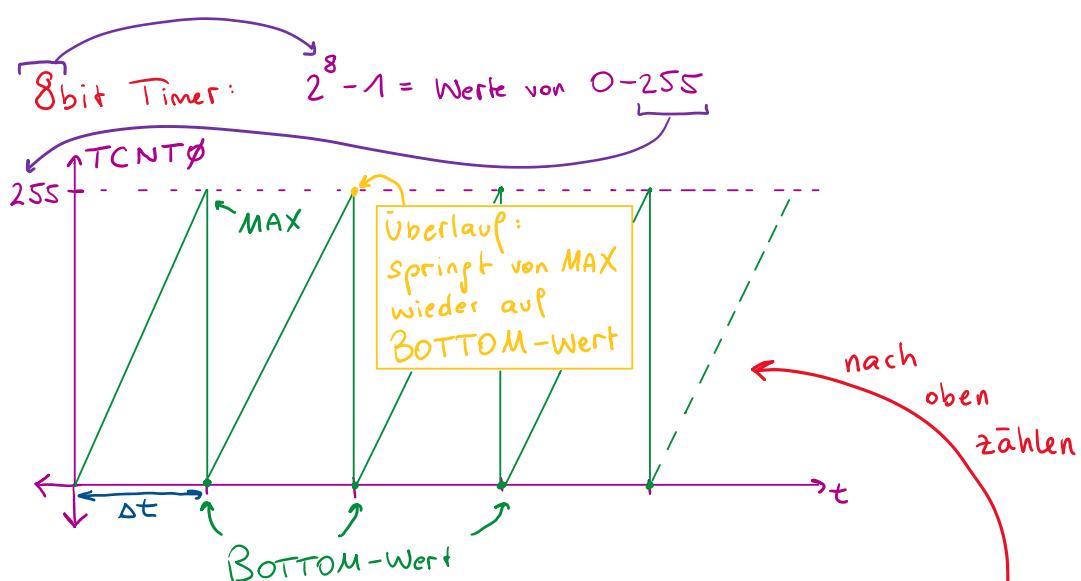
TCO → 8 bit
 TC1 → 16 bit
 TC2 → 8 bit async

wichtige Kennzahl → z.B.:
 d. Timers bei zeitbasierten Vorgängen → als delay-Ersatz
 → ADC zu triggern
 → DC-Motor drehzahlgesteuert zu betreiben

Jeder Timer verfügt über ein "Zählregister":

TC0: TCNT0	=> 8bit	1 Byte	
TC1: TCNT1	=> 16bit	1 Byte 1 Byte	Σ 2 Byte
TC2: TCNT2	=> 8bit	1 Byte	

Siehe io.h → CASE SENSITIVE!



Nach dem Aktivieren und der entsprechenden Konfiguration beginnt TC0 mit dem Arbeiten/Zählen ...

1 Takt → die mit dem Systemtakt versorgt wird

Wie groß ist jetzt Δt ?

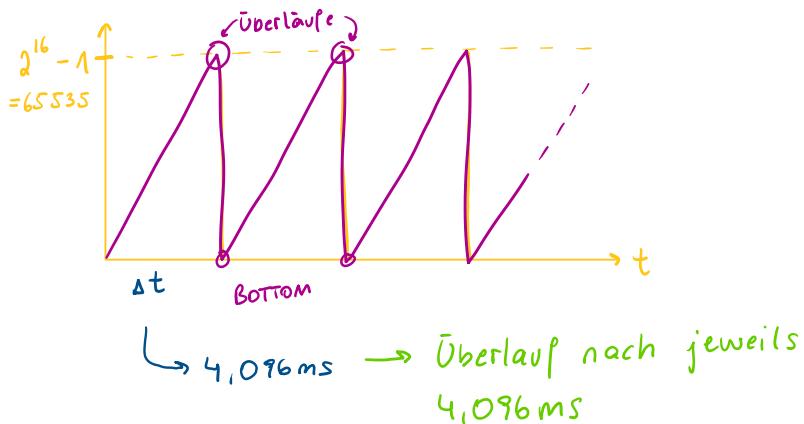
$$T = \frac{1}{f} = \frac{1}{16\text{MHz}} = 0,0000625\text{ms}$$

↑
default der ungeteilte Systemtakt

$$\Delta t = T \cdot 256 = 0,000016\text{s} \stackrel{!}{=} 0,016\text{ms}$$

(2^8)

Timer 1 (TCNT1):



Zusammenfassung:

$$\left. \begin{array}{l} \text{TCNT0: } 0,016\text{ms} \\ \text{TCNT1: } 4,096\text{ms} \end{array} \right\} \rightarrow \text{Überlauf}$$

Realisierung eines 1-Sekunden-Intervall
Möglichkeiten, um ein 1-Sekunden-Intervall umzusetzen

1000ms:

8-bit-Timer:

$$\underline{1000\text{ms}} = 62500 \text{ Überläufe} \stackrel{!}{=} 1 \text{ Sekunde}$$

0-bit-Timer:

$$\frac{1000 \text{ ms}}{0,016 \text{ ms}} = 62500 \text{ Überläufe} \triangleq 1 \text{ Sekunde}$$

16-bit-Timer:

$$\frac{1000 \text{ ms}}{4,096 \text{ ms}} = 244 \text{ Überläufe} \triangleq 1 \text{ Sekunde}$$

@code: (naiver Ansatz)

```
while(1) {
    if(TCNT0 == 255) { // 1 Überlauf
        cntOverflow++;
        if(cntOverflow == 62500) {
            // do something (alle 1 Sekunden)
        }
    }
}
```

Verwendung des Overflow Interrupts:

ISR(TIMER0_OVF_vect)

natürlich für jeden Timer verfügbar
(Timer 1, Timer 2, Timer 0)

@code: (etwas weniger naiver Ansatz)

```
ISR ( TIMER1_OVF_vect ) {  
    countOverflows++;  
    if ( countOverflow == 62500 ) {  
        // do something  
    }  
}
```

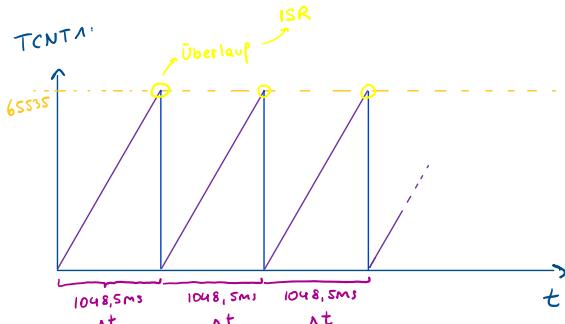
Verlangsamung des Zähltaktes
durch Einsatz des sogenannten Prescalers

Prescaler	Takt neu	T_{neu}	8-Bit \rightarrow 256	16 \rightarrow 65536
8	2 MHz	0,0005ms	0,1275ms	32,76ms
64	250 kHz	0,004ms	1,02ms	262,14ms
256	62,5 kHz	0,016ms	4,08ms	
1024	15,625 kHz	0,064ms	16,32ms	
				1048,5ms 4194,24ms
		$T = \frac{1}{f}$	Δt bis zum jeweiligen Überlauf	
mit diesem verlangsamten Takt wird Timer-Zählwerk gespeist				

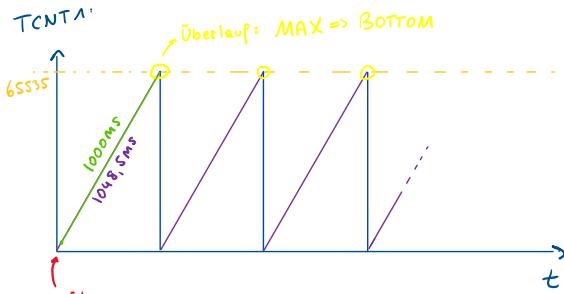
Vorladen des Timers

Ziel:

genau 1s Intervall erzeugen



Lösung: Prescaler 256
 $\hookrightarrow \Delta t = 1048,5\text{ms}$
 bis zum Überlauf
 wird ausgeführt
 ISR(TIMER1_OVF-vect) { }
 \hookrightarrow entspricht nicht
 100% der Aufgabenstellung!



\hookrightarrow VORLADE-WERT
 \downarrow
 bietet neuen Startpunkt
 für Hochzählen bis zum
 Überlauf. Die restliche
 Zeit entspricht 1 Sekunde.

ISR(TIMER1_OVF-vect) {
 // 1 Sekunde Intervall Code
 }

$$\text{TCNT1} = 65535 - 62505; // \text{Vorladewert: } 3030$$

Diese Lösung ist OKAY...

$$\begin{array}{r} 65536 \dots\dots 1048,5\text{ms} \\ \times \dots\dots 1000\text{ms} \\ \hline \end{array}$$

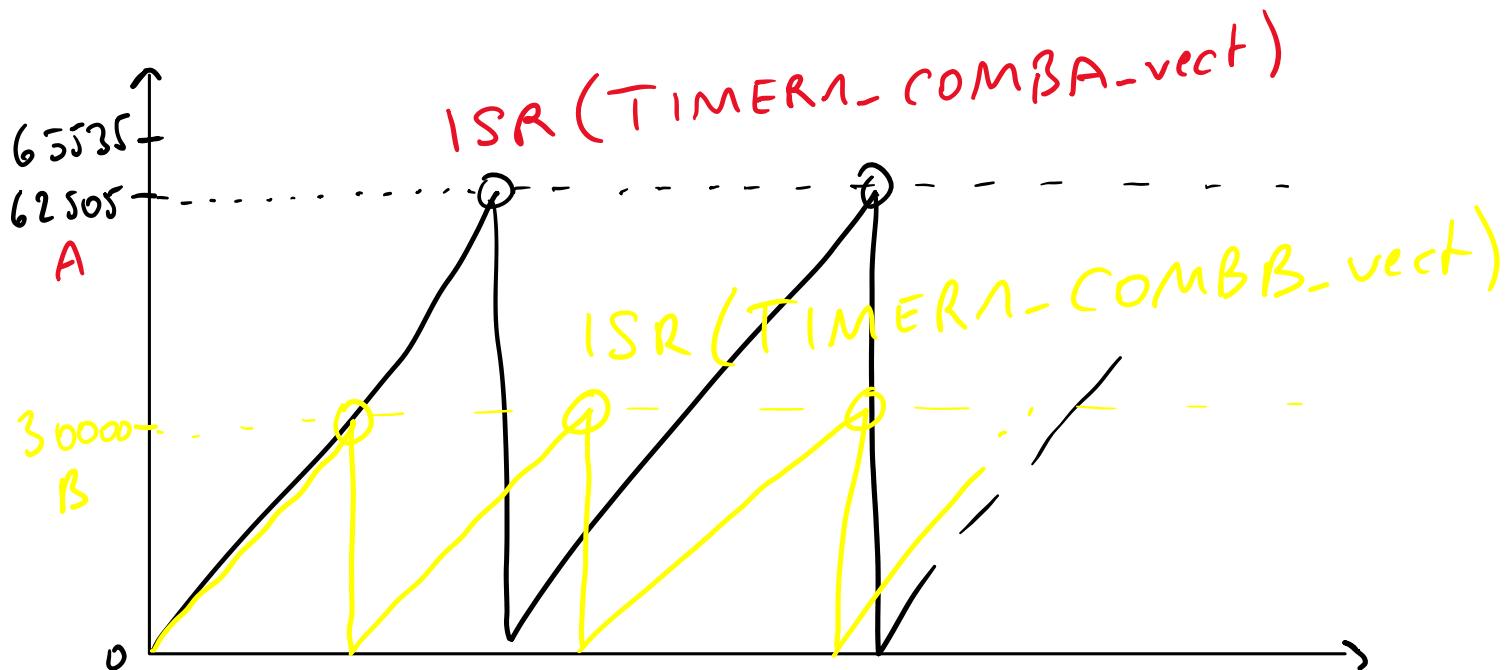
$$\frac{1000\text{ms}}{1048,5\text{ms}} \cdot 65536 = 62500 \text{ Zählschritte}$$

\hookrightarrow genau 1 Sekunde

CTC - Clear Timer on Compare

Friday, March 24, 2023 2:10 PM

CTC - Clear Timer on Compare



// CTC aktivieren

TCCR1B |= (1<<WGM12);

// Output Compare A Interrupt

TIMSKn |= (1<<OCIE1A);



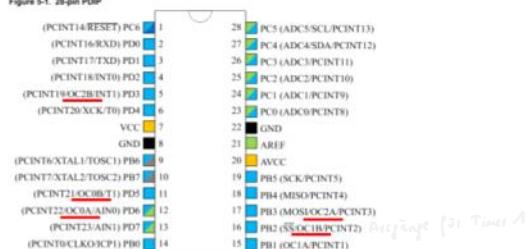
HTL Krems – SYTI4

Timer/Counter Compare Unit

CTC – Wave Form Generation

Der CTC Mode kann nicht nur als „Ersatz“ für das Vorladen, sondern auch zur sog. **Wave Form Generation** (dt. Signalerzeugung) herangezogen werden. Soll heißen: jeder Compare Match-Interrupt bewirkt einen Zustandswechsel (0V => 5V und vice versa) an einem bestimmten Pin des eingesetzten Timers. Somit lässt sich bspw. eine blinkende LED (vgl. Toggle-Funktion) relativ einfach (ohne ISR) umsetzen.

Pin-out
Figure 5-1. 28-pin PDIP



Konfigurationsbeispiel für Timer 1, CTC für Kanal A mit 256 Prescaler:

```

// LED am PDA anschalten
DDRB |= (1 << DDB1); // Port B Ausgang setzen
TCCR1B |= (1 << WGM12) | (1 << CS12); // 256 Prescaler aktivieren
// Toggle Mode für OC1A pin aktivieren
TCCR1A |= (1 << COM1A0); //

// Match Wert für 1 Sekunde setzen
OCR1A = 62505;

```

Hinweis: Die Aktivierung von CTC unterscheidet sich nicht von der „anderen“ CTC-Variante (als Ersatz für das Vorladen). Die **COM1A[1:0]** Bits überschreiben die „normale“ Port-Konfiguration. Es braucht dann keine ISR(TIMERx_COMPA_vect).

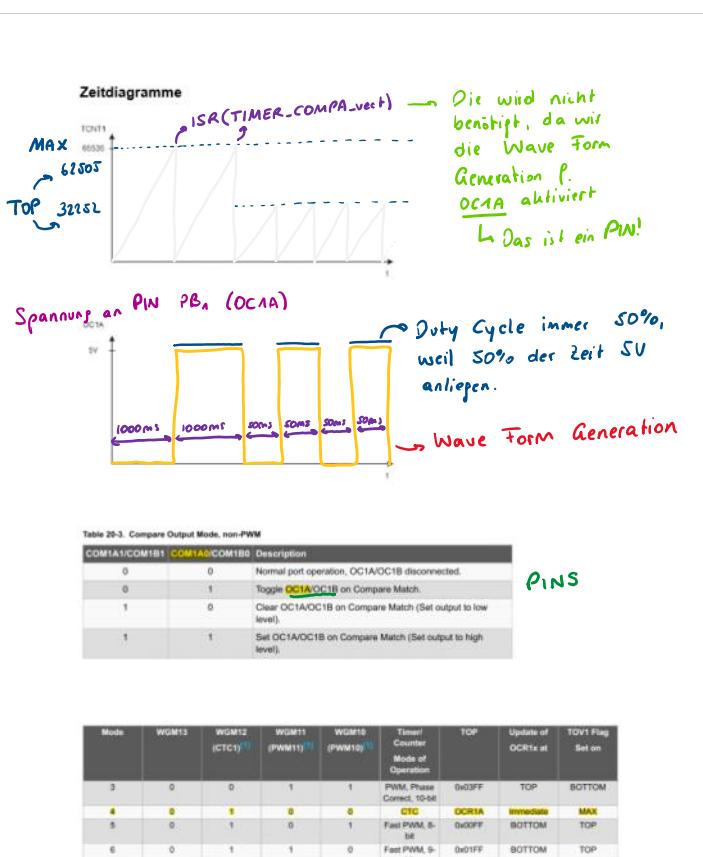
Timer Betriebsarten:

1: normal mode

2a: CTC (Software) ISR(TIMERx_COMPA_vect) {}

2b: CTC (Hardware)

3: PWM



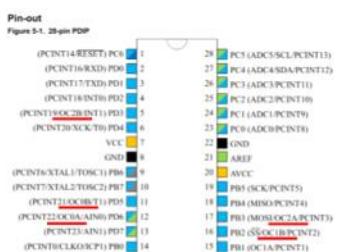
Operation								
3	0	0	1	1	PWM_Phase Config_10-bit	0x03FF	TOP	BOTTOM
4	0	0	0	0	CIC	0x01FA	Immediate	MAX
5	0	1	0	1	Fast PWM_5- bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM_5- bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM_10- bit	0x02FF	BOTTOM	TOP



HTL Krems – SYTI4

Timer – Pulse Width Modulation

Mithilfe der *Pulse Width Modulation* (PWM) kann man bspw. die Drehzahl eines DC-Motors oder die Leuchtkraft einer LED bestimmen.

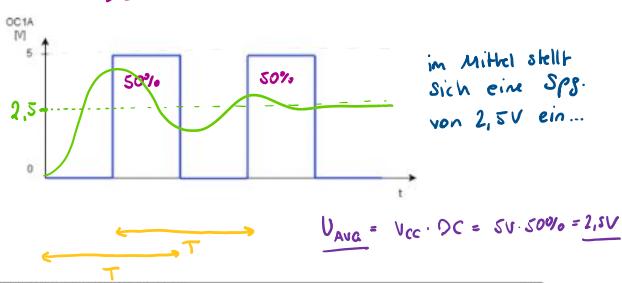


Ann: Ein DC-Motor hat folgende Kenndaten: 5V/300 Umdrehungen (rpm) im Leerlauf

- Drehzahl bei 2,5V: 150rpm
 - Drehzahl bei 1,25V: 75rpm

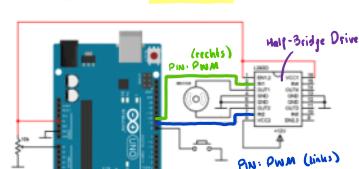
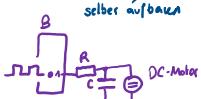
Wie verhält sich der Motor bzw. dessen Drehzahl, wenn folgender Spannungsverlauf über die Zeit t anliegt?

$$DC = 50\% \rightarrow \phi_{2,SV} \quad (\text{Frequenz richtig einstellen})$$



Filter (RC-Glied) am Ausgang (OC_{ox}) oder gleich | 293D Motor Driver:

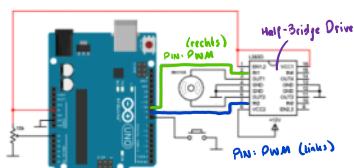
Filter (RC-Glied)



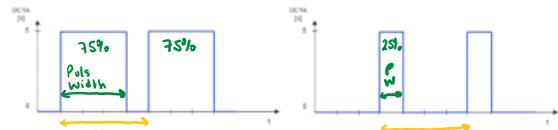
75% Duty Cycle

25M Duty Cycle

Filter (RC-Glied) am Ausgang (OCnx) oder gleich L293D Motor Driver.



75% Duty Cycle



Ermittlung des Duty Cycles

$$\text{Duty Cycle} = \frac{T_{on}}{T_{on} + T_{off}} * 100\%$$

wie viel Prozent der Periode T high sind

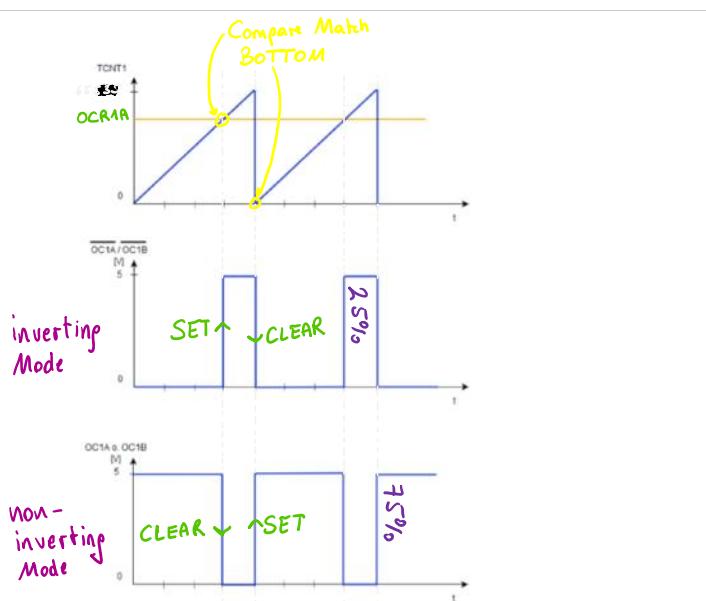
Fast PWM am Bsp. Timer 1

Fast PWM kann entweder invertiert oder nicht-invertiert betrieben werden. Die Konfiguration erfolgt über die COM-Bits des TCCR1A Registers. Die Bedeutung der Bits hängt von der Betriebsart ab.

Table 20-4. Compare Output Mode, Fast PWM

COM1A1/ COM1B1	COM1A0/ COM1B0	Description
1	0	Clear OC1A/OC1B on Compare Match, set OC1A/OC1B at BOTTOM (non-inverting mode)
1	1	Set OC1A/OC1B on Compare Match, clear OC1A/OC1B at BOTTOM (inverting mode)

Im Grunde legt man fest, wann der PWM-Ausgang *high* sein soll - also entweder nach dem Erreichen oder vor dem Erreichen des *Match*-Wertes. In jedem Fall zählt der Timer von BOTTOM auf TOP und beginnt dann wieder bei BOTTOM. Dieser Verlauf entspricht einem Sägezahn.



Die Frequenz des Fast PWM-Signals kann für Timer 1 wie folgt berechnet werden:

$$f_{PWM} = \frac{f_{clk}}{N * (1 + TOP)}$$

PWM-Signalfrequenzen bei 16 MHz System Clock:

kein Prescaler	8-Bit (Top = 255)	9-Bit (Top=511)	10-Bit (Top = 1023)
System Clk / 1	62500 Hz	31250 Hz	15625 Hz
System Clk / 8	7812,5 Hz	3906,3 Hz	1953,1 Hz
System Clk / 64	976,6 Hz	488,3 Hz	244 Hz
System Clk / 256	244,1 Hz	122,1 Hz	61 Hz
System Clk / 1024	61 Hz	30,5 Hz	15,3 Hz

WGM-Bits kann man die PWM Betriebsart konfigurieren:

Table 20-6. Waveform Generation Mode Bit Description

Mode	WGM13 (CTC1) ^④	WGM12 (CTC1) ^⑤	WGM11 (PWM11) ^⑥	WGM10 (PWM10) ^⑦	Timer/ Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 1:4	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-bit	0x1FFF	TOP	BOTTOM
Mode	WGM13	WGM12 (CTC1) ^④	WGM11 (PWM11) ^⑥	WGM10 (PWM10) ^⑦	Timer/ Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag Set on
3	0	0	1	1	PWM, Phase Correct, 10:8	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8- bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 8- bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10- 14	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase and Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase and Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	Reserved	-	-	-
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

- Fast PWM
- Phase Correct
- Phase & Frequency Correct

TCCR1A non-inverting vs. inverting

variable

TCCR1A

Bit	7	6	5	4	3	2	1	0
Access	COM1	COM1	COM1	COM1			WGM11	WGM10
Reset	0	R/W	0	R/W	0		R/W	0

Timer
PWM
Betrieb

TCCR1B

Bit	7	6	5	4	3	2	1	0
Access	IHC1	ICES1		WGM13	WGM12	CS12	CS11	CS10
Reset	0	R/W	0	R/W	R/W	R/W	R/W	R/W

Minimalkonfiguration für 8-Bit Fast PWM mit Timer 1 im Non-inverted Mode:

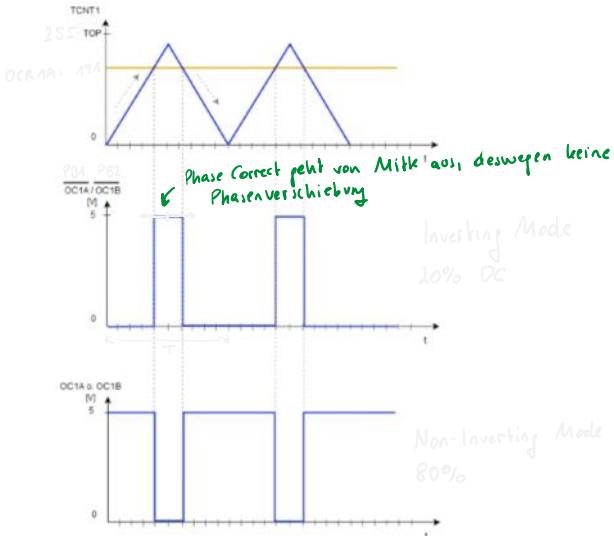
↳ siehe Übung LED / DC-Motor



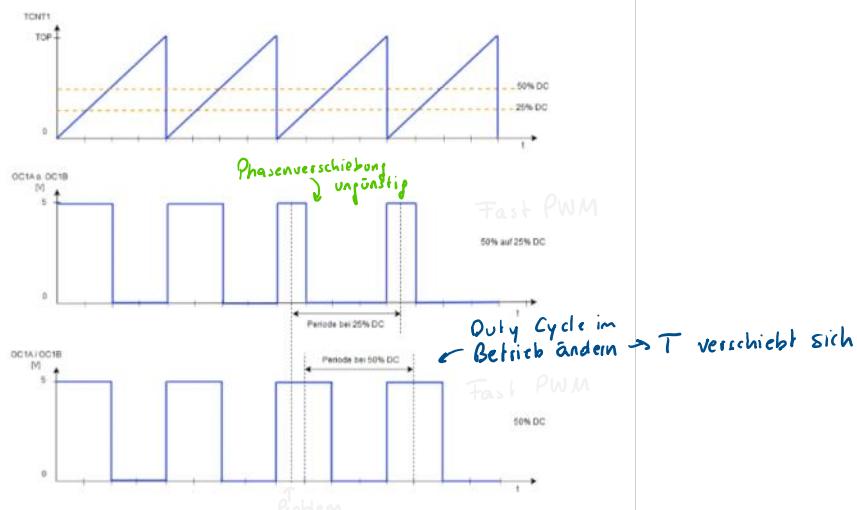
Timer – Pulse Width Modulation II

Phase Correct PWM am Bsp. Timer 1

Bei der Phase Correct PWM zählt der Timer rau (von BOTTOM auf TOP) **und** wieder runter (von TOP auf BOTTOM). Dadurch ergibt sich ein dreieckförmiger Signalverlauf, wie nachfolgende Abbildung illustriert. Das Ausgangssignal an den Pins OC1A oder OC1B ist grundsätzlich das gleiche wie bei der Fast PWM.



Wie unterscheidet sich nun die Fast PWM von der Phase Correct PWM? Die Fast PWM ist zwar „schneller“ als die Phase Correct PWM, weist aber einen Nachteil auf; und zwar dann, wenn man das DC während des Betriebes ändert. Es kommt zur **Phasenverschiebung**, da die Grundschwingung des Rechteckausgangssignals nicht mehr von Mitte zu Mitte der Periode reicht, wie nachfolgende Abbildung zeigt.



Die Frequenz des Phase Correct PWM-Signals kann (für Timer 1) wie folgt berechnet werden:

$$f_{\text{Phase Correct PWM}} = \frac{f_{\text{clk}}}{N * 2 + TOP}$$

rauf- & runter-zählen, deswegen durch 2
PWM-Signalfrequenzen bei 16 MHz System Clock:

Fast PWM
immer schneller, weil
Phase Correct zuerst
unterzählt

	8-Bit (Top = 255)	9-Bit (Top=511)	10-Bit (Top = 1023)
System Clk / 1	21.200	118.625	7812,5
System Clk / 8	3.906,25	1153,1	976,6
System Clk / 64	613,75	174,7	123,1
System Clk / 256	24,5	61	30,5
System Clk / 1024	5,0	15,125	7,6



USART I

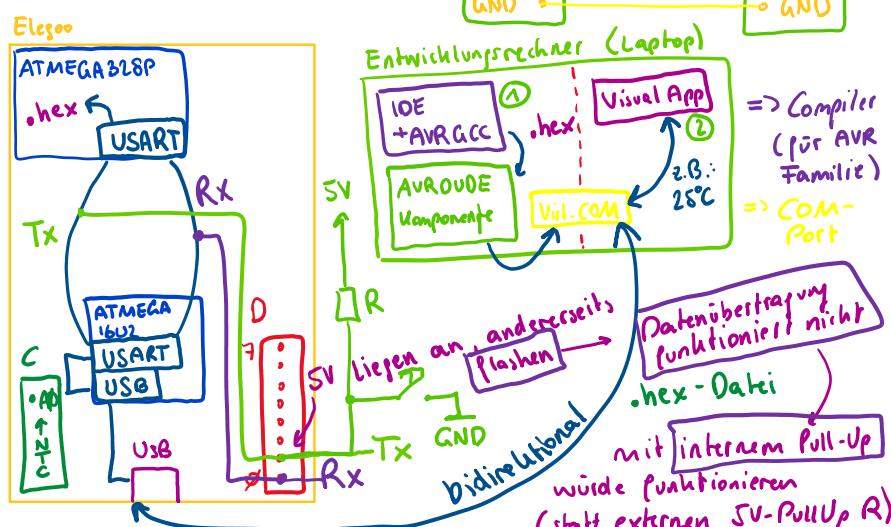
Die µC-Komponente **USART** (*Universal Synchronous and Asynchronous serial Receiver and Transmitter*) realisiert eine digitale Schnittstelle, die eine serielle Datenübertragung ermöglicht. Mögliche Anwendungsbereiche sind:

- Flashen (vgl. Übung)
- Übertragen von (gespeicherten) Werten z.B.: Temperatur übertragen
- Mensch-Maschine-Schnittstelle (z.B. Konfiguration eines Embedded-SW-Systems)
- ...

Die einfachste Realisierungsform ist die asynchrone Variante, welche ausschließlich – neben GND – zwei Datenleitungen benötigt:

- TXD (Tranceive Data → Daten senden)
- RXD (Receive Data → Daten empfangen)

Realisierung auf dem Arduino/Elegoo-Board:



Der Betrieb der Schnittstelle setzt beim Sender als auch Empfänger entsprechende Einstellungen voraus, die bei beiden Systemen identisch sein müssen:

- Geschwindigkeit (**baud rate** = Anzahl der Symbole/Zeiteinheit)
- Anzahl der **Datenbits** (i. d. R. 8)
- **Stop-Bit** (Ja/Nein)
- **Parity** (Ja/Nein)
- **Flow Control**

Der ATmega328P verfügt über einen USART: **USART0** → siehe Datenblatt, S.178

- ① USART nur beim Flashen in Verwendung
 - ② USART während Programm ausführung in Verwendung
- ↓
- PDI und PDO dürfen NICHT anderweitig verwendet werden

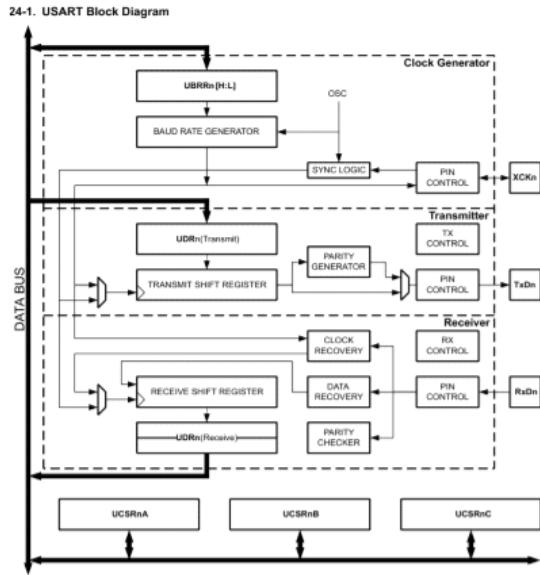


Table 24-1. Equations for Calculating Baud Rate Register Setting

Operating Mode	Equation for Calculating Baud Rate ⁽¹⁾	Equation for Calculating UBRRn Value	Baud Rate $f_{osc} = 16.0000MHz$				
			U2Xn = 0		U2Xn = 1		
		UBRRn	Error	UBRRn	Error		
Asynchronous Normal mode (U2Xn = 0)	$BAUD = \frac{f_{osc}}{16(UBRRn + 1)}$	$UBRRn = \frac{f_{osc}}{16BAUD} - 1$					
Asynchronous Double Speed mode (U2Xn = 1)	$BAUD = \frac{f_{osc}}{8(UBRRn + 1)}$	$UBRRn = \frac{f_{osc}}{8BAUD} - 1$	2400	416	-0.1%	832	0.0%
Synchronous Master mode	$BAUD = \frac{f_{osc}}{2(UBRRn + 1)}$	$UBRRn = \frac{f_{osc}}{2BAUD} - 1$	4800	207	0.2%	416	-0.1%
			9600	103	0.2%	207	0.2%
			14.4k	68	0.6%	138	-0.1%
			19.2k	51	0.2%	103	0.2%
			28.8k	34	-0.8%	68	0.6%
			38.4k	25	0.2%	51	0.2%
			57.6k	16	2.1%	34	-0.8%
			76.8k	12	0.2%	25	0.2%
			115.2k	8	-3.5%	16	2.1%
			230.4k	3	8.5%	8	-3.5%
			250k	3	0.0%	7	0.0%
			0.5M	1	0.0%	3	0.0%
			1M	0	0.0%	1	0.0%

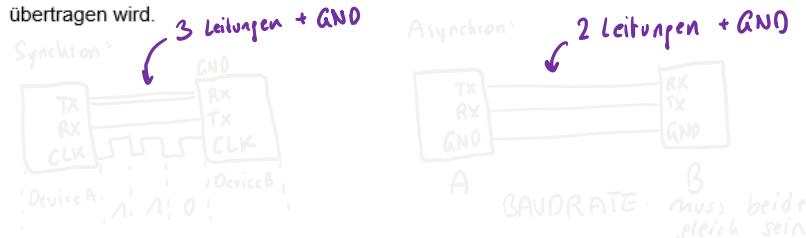


USART II

*UART → kann nur
asynchron*
kein S

Synchrone vs. asynchrone Übertragung

Bei der synchronen Übertragung wird zusätzlich zu den Daten (Rx u. Tx) ein Taktsignal übertragen. Der Empfänger kann am Takt signal erkennen, wann das nächste Bit übertragen wird.

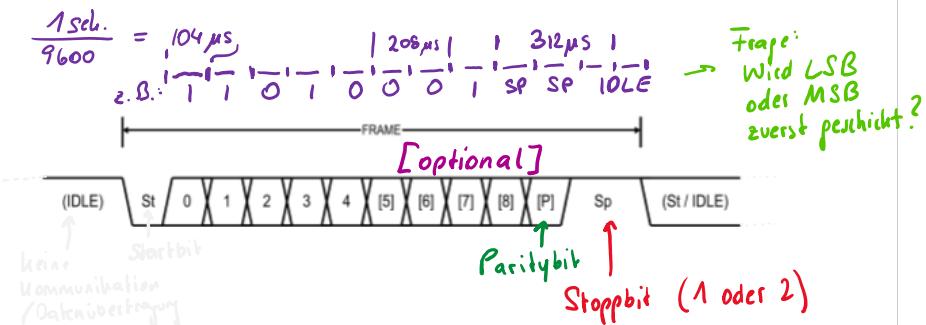


Bei der asynchronen Variante wird kein Takt signal übertragen. Sender und Empfänger verwenden jeder für sich einen eigenen Takt, wobei beide auf die gleiche Frequenz eingestellt sein müssen. Diese Vereinbarung wird mit der sog. Schrittgeschwindigkeit (Baudrate) definiert.

$$\text{UART: BAUDRATE} = \text{BITRATE}$$

Aufbau eines Frames

Im Zeitintervall zw. Start- und Stopbit arbeiten Sender und Empfänger mit gleicher Schrittgeschwindigkeit. Hierzu definiert der Standard verschiedene **Baudraten** (300, 600, 1200, ..., 9600, ..., 115200).



Jede Übertragung beginnt mit einem **Startbit** (St, low), dann folgen 5, 6, 7, 8 oder 9 Datenbits. In der Regel sind es 8.

→ Prüfsumme

Dann kann ein sog. **Paritybit** (P) folgen. Dieses ist optional, und dient zur Erkennung von 1-Bit-Fehlern. Je nach Variante erfolgt die Ergänzung auf eine **gerade Anzahl** von „1“ (**even parity**) oder **ungerade Anzahl** von „1“ (**odd parity**). Die Kalkulation erfolgt mittels XOR-Verknüpfung der Datenbits.

1, 3 o. 5 Bit-Fehler

$$P_{\text{even}} = d_{n+1} \oplus \dots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 0$$

XOR: E1 E2 A		
0	0	0
0	1	1
1	0	1
1	1	0

$$P_{\text{odd}} = d_{n+1} \oplus \dots \oplus d_3 \oplus d_2 \oplus d_1 \oplus d_0 \oplus 1$$

Kalkulation am Bsp. der Zahl 5 mit dem ASCII-Code: 0011 0101

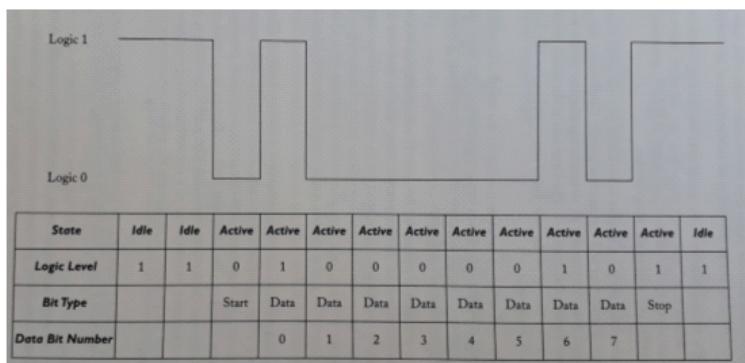
Ergebnis even parity: 00110101 → PB: 0 ⇒ 001101010

Ergebnis odd parity: 00110101 → PB: 1 ⇒ 001101011

Sendereihenfolge:
 MSB: 00110101
 LSB: 11010110

Abschluss der Übertragung eines Frames erfolgt mittels **einem oder zwei Stop-Bits**.

An einem Oszi wurde folgender Signalverlauf aufgezeichnet und in weiterer Folge ansprechend aufbereitet. Welches Zeichen wird übertragen?



MSB first: 1000 0010 ⇒ 1

LSB first: 0100 0001 ⇒ A

SYTI_Pointer-v1.0

Wednesday, May 3, 2023 9:49 AM



Pointer in C

Pointer (Zeiger) sind ein überaus probates Mittel, um nicht zuletzt performanten Programmcode zu entwickeln. Aber: Mit Pointern kann man auch die **undurchschaubarsten Fehler** produzieren – echt!

Bei **Pointern** handelt es sich um Variablen, die **Speicheradressen** aufnehmen könnten. Das lässt sich folgendermaßen realisieren:

```

1 //Variable val erstellen und Wert zuweisen
2 char val = 12;
3 //Pointer-Variablen vom Typ char mit * erstellen
4 char *ptrToVal = NULL;
5 //Mit dem &-Operator (Kaufmanns-Und) wird der Pointer-
//Variable die Adresse der Variable val zugewiesen
6 ptrToVal = &val; → Addressoperator &
7 //Die Ausgabe der Adresse von ,val' über Pointer ,ptrToVal'
8 printf("Adresse von val: %p \n", ptrToVal);

```

Möglicheres Ergebnis auf einem Windows-Rechner¹:

```

Administrator: C:\Windows\System32\cmd
D:\HTML\Lehre\Lehrinhalte\SYTI\UE\Pointer>pointerDemo.exe
00000008 Adresse von val

```

Im Arbeitsspeicher ergibt sich nun Folgendes:

`char val = 12;`
`char *ptrToVal = NULL;`

`ptrToVal = &val;`

Adresse	Wert
0000 0008	12
...	
0000 0012	0

val
ptrToVal

Adresse	Wert
0000 0008	12
...	
0000 0012	0000 0008

val
ptrToVal

¹ Am einfachsten mit Visual Studio ein C-Konsolenprogramm erstellen und ausführen. Unter Unix (oder Cygwin) ist der gcc standardmäßig verfügbar.

Wie gelangt man nun an den Wert, auf welchen die Pointer-Variable zeigt? Ganz einfach, wie nachfolgende Erweiterung des Beispiels zeigt.

```

9 //Dereferenzierung mithilfe des *-Operators
10 char tmp = *ptrToVal;
11 //Die Ausgabe des Wertes von ,val' über Pointer ,ptrToVal'
12 printf("Wert von val: %d \n", tmp);
13 //...oder ohne ,tmp'-Variable
14 printf("Wert von val: %d \n", *ptrToVal);

```

Zusammenfassend ist bzgl. der eingesetzten Operatoren festzuhalten:

- &: Mit dem **Adressoperator** weist man einem Pointer die Adresse einer Variable zu.
- *: Mit dem „**Dereferenzierungsoperator**“ erhält man jenen Wert, der in der referenzierten Speicheradresse steht.

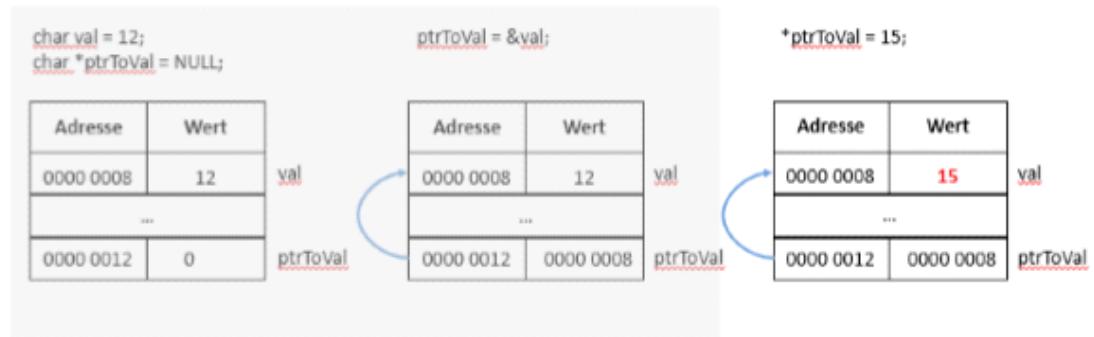
Mithilfe des *-Operators ist es auch möglich, den Wert der referenzierten Variable zu manipulieren, wie nachfolgender Code zeigt.

```

15 //Manipulation von ,var' mithilfe des *-Operators
16 *ptrToVal = 15;
17 //Die Ausgabe des Wertes von ,val'
18 printf("Wert von val: %d \n", val);

```

Im Arbeitsspeicher ergibt sich nun nachfolgendes Bild:



Pointer und Arrays

Pointer und Arrays sind sehr eng miteinander verwandt. Denn ein Array ist eigentlich nichts anderes als ein Pointer auf das erste Element (an der Stelle 0) des Arrays. Logisch, oder?

```

1 //Deklaration eines Arrays
2 char values[5] = {22, 23, 22, 21, 23};
3 //Pointer-Variable erstellen und auf ,values'
4 char *ptrToValues = &values;
      ↑
      und result;

```

Zeile 4: Bei der Zuweisung von `values` ist der Adressoperator – also `&values` – nicht erforderlich!

Beispielhaftes Abbild des Arbeitsspeichers:

`*ptrToValues = values;`

Adresse	Wert	
0000 00F4	22	values
0000 00F5	23	
0000 00F6	22	
0000 00F7	21	
0000 00F8	23	
...		
0000 00FC	0000 00F4	ptrToValues

Um das erste Feld von `values` auszulesen, kann wie gewohnt dereferenziert werden:

```
5 printf("Wert von values[0]: %d \n", *ptrToValues);
```

Um den nächsten Wert auszulesen, ist der Zeiger auf die darauffolgende Adresse zu setzen. Das geht ganz einfach:

```

6 //Zeiger um 1 inkrementieren
7 ptrToValues++;
8 printf("Wert von values[1]: %d \n", *ptrToValues);
→ Pointer zeigt auf nächste Adresse

```

Ist die Ausgabe sämtlicher Elemente von `values` gewünscht, ist wie folgt vorzugehen:

```
9  while(*ptrToValues){  
10    printf("%d \n", *ptrToValues);  
11    ptrToValues++;  
12 }
```

Pointer finden sehr oft bei Funktionen Anwendung (Stichwort *call by reference*). Ein Beispiel liefert nachfolgender Codeauszug:

```
1  #include <stdio.h>  
2  
3  void printMsg(char *data){  
4      while(*data)  
5          printf("%c", *data++);  
6  }  
7  
8  int main(){  
9      char msg1[10] = "Msg 1\n";  
10     char msg2[10] = "Msg 2\n";  
11  
12     char *ptrToMsg = NULL;  
13     ptrToMsg = msg1;  
14     printMsg(ptrToMsg);  
15  
16     ptrToMsg = msg2;  
17     printMsg(ptrToMsg);  
18     printMsg(msg1);  
19     return 0;  
20 }
```

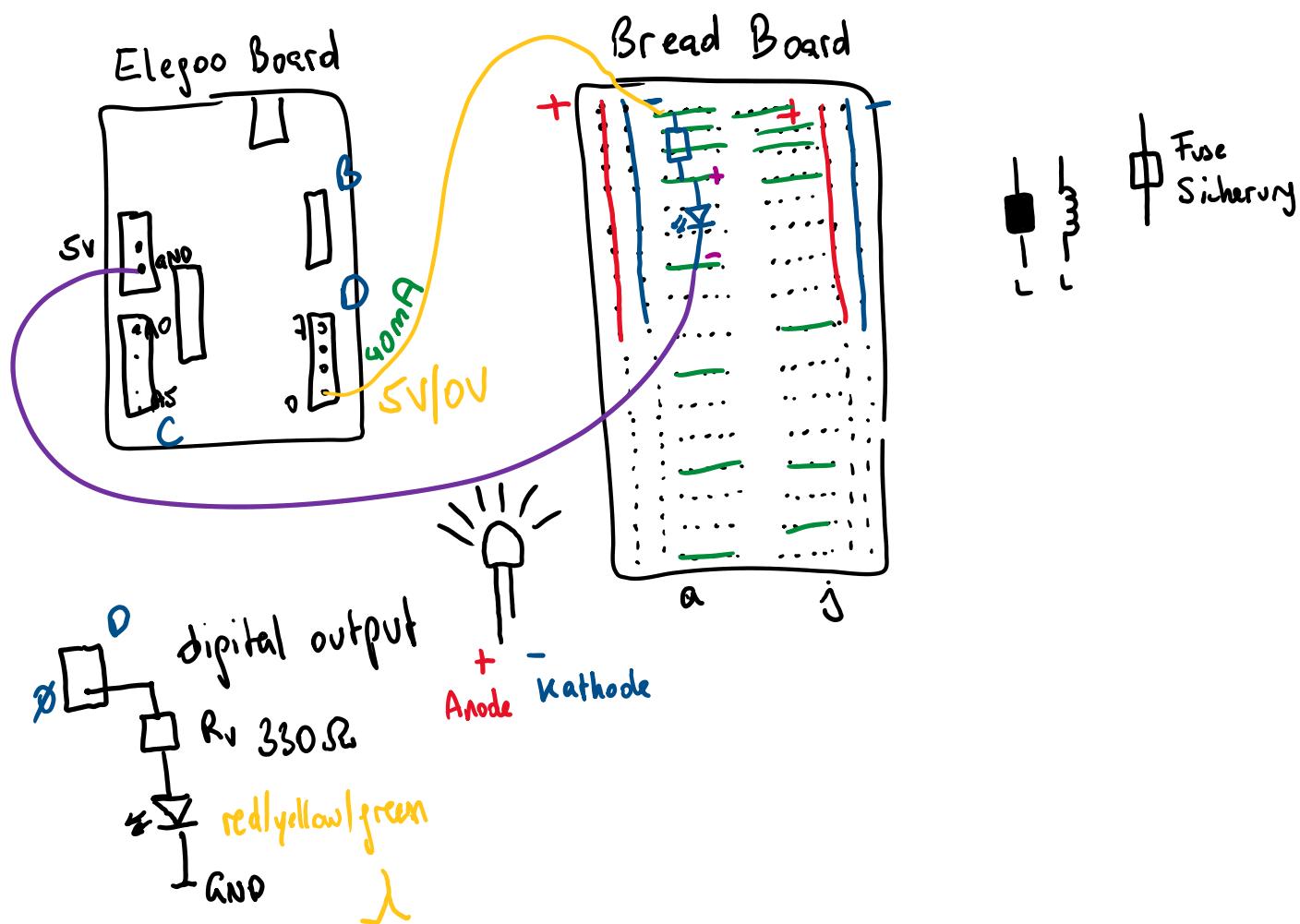
Zeile 18: Der unmittelbare Aufruf (der Funktion) mit einem Array ist natürlich auch möglich!

Aufbau

Mittwoch, 30. November 2022 19:13

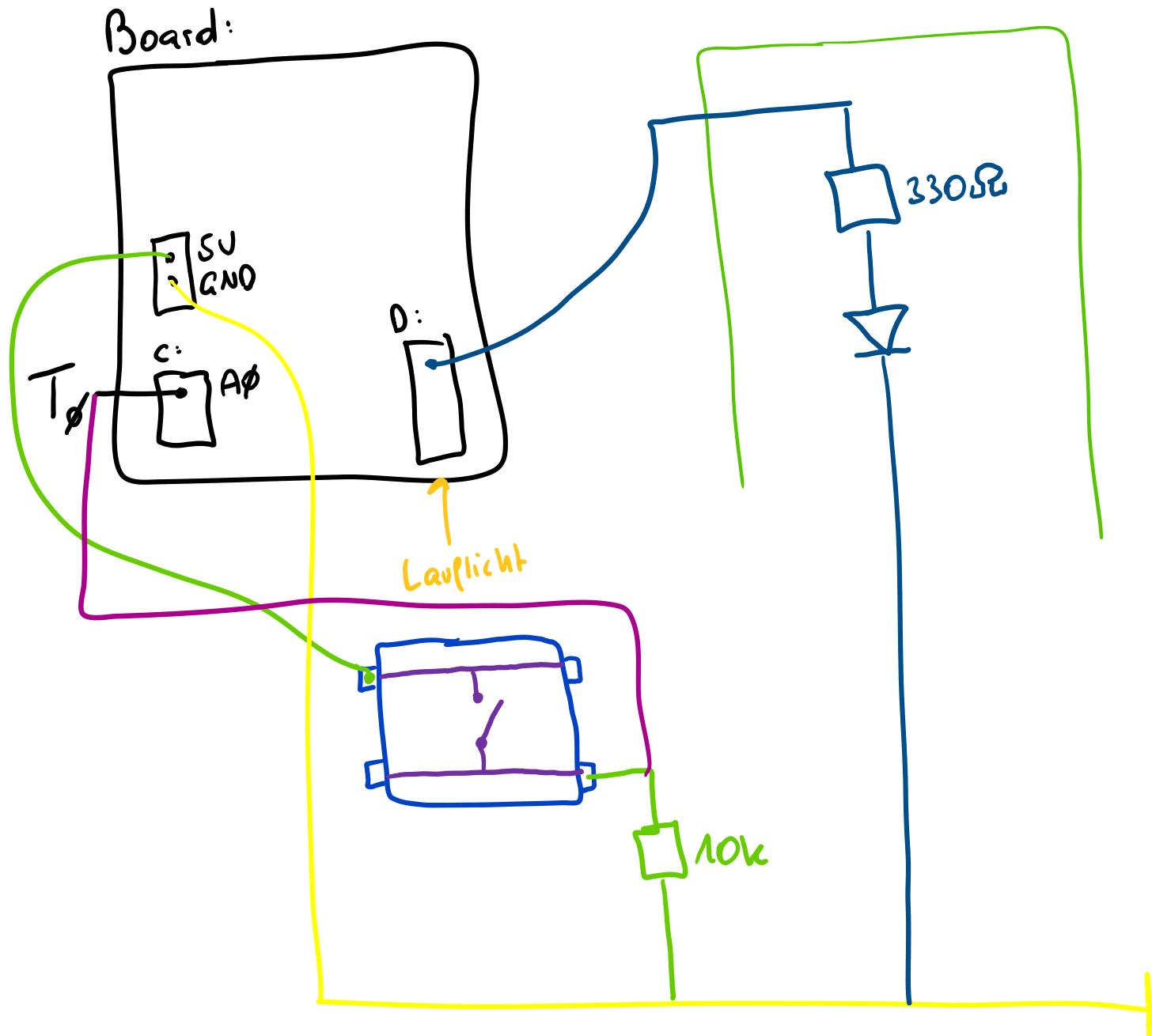
Elegoo Board and Bread Board 1 LED

Freitag, 16. September 2022 14:08



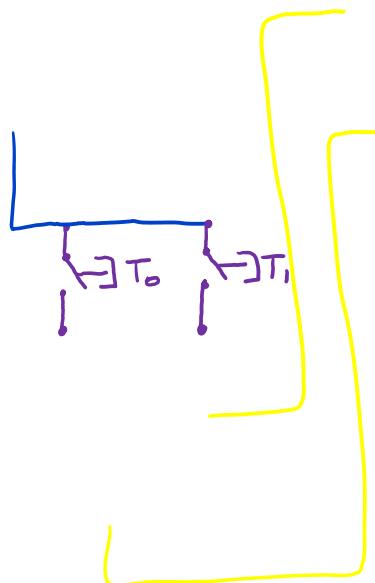
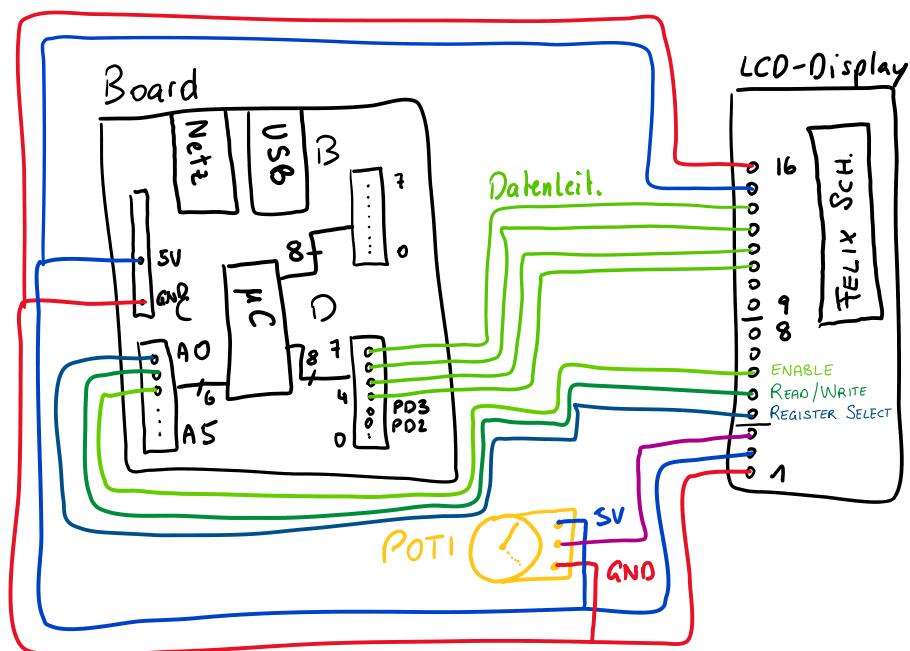
Tasten Board Aufbau

Freitag, 14. Oktober 2022 14:50



LCD-Display

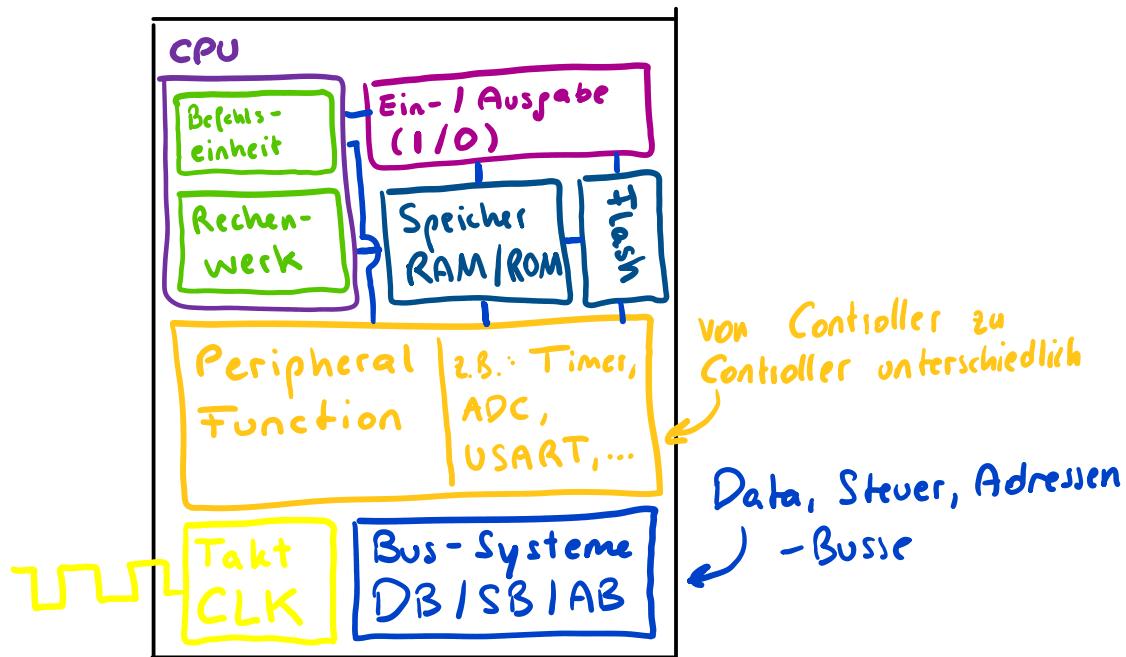
Tuesday, January 10, 2023 8:21 PM



Typischer Aufbau eines μC

Mittwoch, 24. Mai 2023 10:01

Typischer Aufbau eines μC



- μC = Einchip - Computer

Code

Sunday, March 12, 2023 5:40 PM

All includes and defines

Sunday, March 12, 2023 5:55 PM

```
#define F_CPU 16000000

#include <avr/io.h>
#include <string.h>
#include <avr/interrupt.h>
#include <stdbool.h>
#include <stdio.h>
#include "lcd.h"

#include "dht.h"

#define __DELAY_BACKWARD_COMPATIBLE__
#include <util/delay.h>
```

Max Min

Friday, April 21, 2023 2:31 PM

```
#define MAX(x, y) (((x) > (y)) ? (x) : (y))
#define MIN(x, y) (((x) < (y)) ? (x) : (y))
```

LED / Bitmanipulation

Sunday, March 12, 2023 5:41 PM

```
DDRD = 0xFF;
```

```
PORTD = 0x00;
```

```
// LED 0
```

```
PORTD |= (1<<PORTD0);
```

```
PORTD &= ~(1<<PORTD0);
```

Tasten

Sunday, March 12, 2023 5:40 PM

PullUp mit Interrupts:

```
PORTD |= (1<<PORTD2);
```

```
EIMSK |= (1<<INT0);
```

Check what taste:

```
if (!(PINC & (1<<PINCO)))
```

LCD

Sunday, March 12, 2023 5:40 PM

```
#include "lcd.h"

lcd_init(LCD_DISP_ON);

lcd_command(LCD_MOVE_DISP_LEFT); // nach left moven
lcd_command(LCD_ENTRY_INC_SHIFT); // ltr mode

lcd_clrscr();
lcd_gotoxy(16, 0);

lcd_puts(buffer);
```

Interrupts

Sunday, March 12, 2023 5:40 PM

```
#include <avr/interrupt.h>

// direkt externe Interrupts
EIMSK |= (1<<INT0) | (1<<INT1);

// gruppenbasierte externe Interrupts
PCICR |= (1<<PCIE2);
PCMSK2 |= (1<<PCINT18) | (1<<PCINT19);

sei();

ISR (INT0_vect) {
}

volatile
```

ADC

Sunday, March 12, 2023 5:40 PM

```
// REFS0: Aufgrund der Beschaltung des ADCs.  
// A3: Analoges Signal an PC3 => MUX0 | MUX1  
ADMUX |= (1<<REFS0) | (1<<MUX0) | (1<<MUX1);  
  
// ADEN => Enables ADC  
// ADPSx => Division Factor to get between 50kHz and 200kHz with our 60MHz Elegoo.  
ADCSRA |= (1<<ADEN) | (1<<ADPS0) | (1<<ADPS1) | (1<<ADPS2);  
  
// ADSC => Start Conversion  
ADCSRA |= (1<<ADSC);  
while(ADCSRA & (1<<ADSC));  
  
#define U_REF 5.0  
#define ADC_RES 1024.0  
float ConvertADCtoVolt(float adc) {  
    return (adc / ADC_RES) * U_REF;  
}  
  
ADCSRA |= (1<<ADIE);  
ISR(ADC_vect) {  
  
    // ADSC => Start Conversion  
    ADCSRA |= (1<<ADSC);  
}
```

Timer

Sunday, March 12, 2023 5:41 PM

```
// activate timer (TCNT1) -> set clock select
TCCR1B |= (1<<CS10);

// activate timer 1 overflow interrupt
TIMSK1 |= (1<<TOIE1);
sei();

ISR(TIMER1_OVF_vect) {

}
```

Übung / Sonstiges

Mittwoch, 30. November 2022 19:16

Übung HÜ 1 (Bitmanipulation)

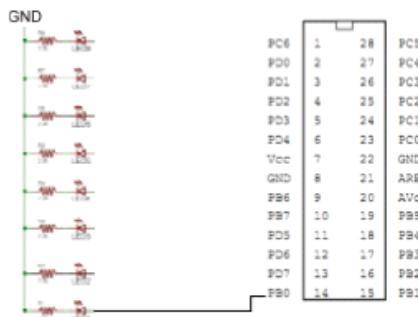
Mittwoch, 5. Oktober 2022 09:52



SYTI

Übungsblatt „Bit-Operationen 1“

1 LED am Port B (PORTB):



Pin 1 bis 7 werden anderweitig verwendet und dürfen keinesfalls überschrieben werden!

LED 0 ein:

PORTB \leftarrow

1	1	1	0	0	0	0	0
---	---	---	---	---	---	---	---

Maske: 00000001

Port-Zuweisung PORTB |= 0x01;

LED 0 aus:

PORTB \leftarrow

1	1	1	0	0	0	0	1
---	---	---	---	---	---	---	---

Maske: 11111110

Port-Zuweisung PORTB &= 0xFE;

Ergebnis:

PORTB \leftarrow

1	1	1	0	0	0	0	0
---	---	---	---	---	---	---	---

4 LEDs am Port B:



Pins 2-5 werden anderweitig verwendet und dürfen keinesfalls überschrieben werden!

LED 0 u. 1 ein:

PORTB \leftarrow

0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---

Maske: 00000011

Port-Zuweisung PORTB |= 0x03;

LED 6 u. 7 ein:

PORTB \leftarrow

0	0	0	0	0	0	1	1
---	---	---	---	---	---	---	---

Maske: 11000000

Port-Zuweisung PORTB |= 0xC0;

LED 0, 1, 6 u. 7 aus:

PORTB \leftarrow

1	1	0	0	0	0	1	1
---	---	---	---	---	---	---	---

Maske: 00111100

Port-Zuweisung PORTB &= 0x3C;

Ergebnis: 00000000

Schriftliche Mitarbeitsüberprüfung

Name: Felix Schneider

Beurteilung: -

Viel Erfolg!

1) Welche Aussage(n) ist/sind richtig?

- Der ATmega328P verfügt über die Ports B, C und D, wobei ausschließlich Port C digital Output ermöglicht. (falsch, weil auch analog)
- Zum Flashen eines Programmes ist das Atmel- bzw. Microchip-Studio unabdingbar. (wahrscheinlich auch andere IDEs)
- Bei der bitweisen Manipulation eines Registers (z.B. PORTD) werden immer 8 Bits mit der Maske verrechnet, auch dann, wenn nur ein einziges gesetzt/gelöscht wird.
- Beim ATmega328P handelt es sich um einen 8 Byte Controller mit 32 kB Flash-Speicher. (X)
- Keine ist richtig

2) Analysieren Sie nachfolgenden Code und wählen Sie eine der Antwort-Optionen.

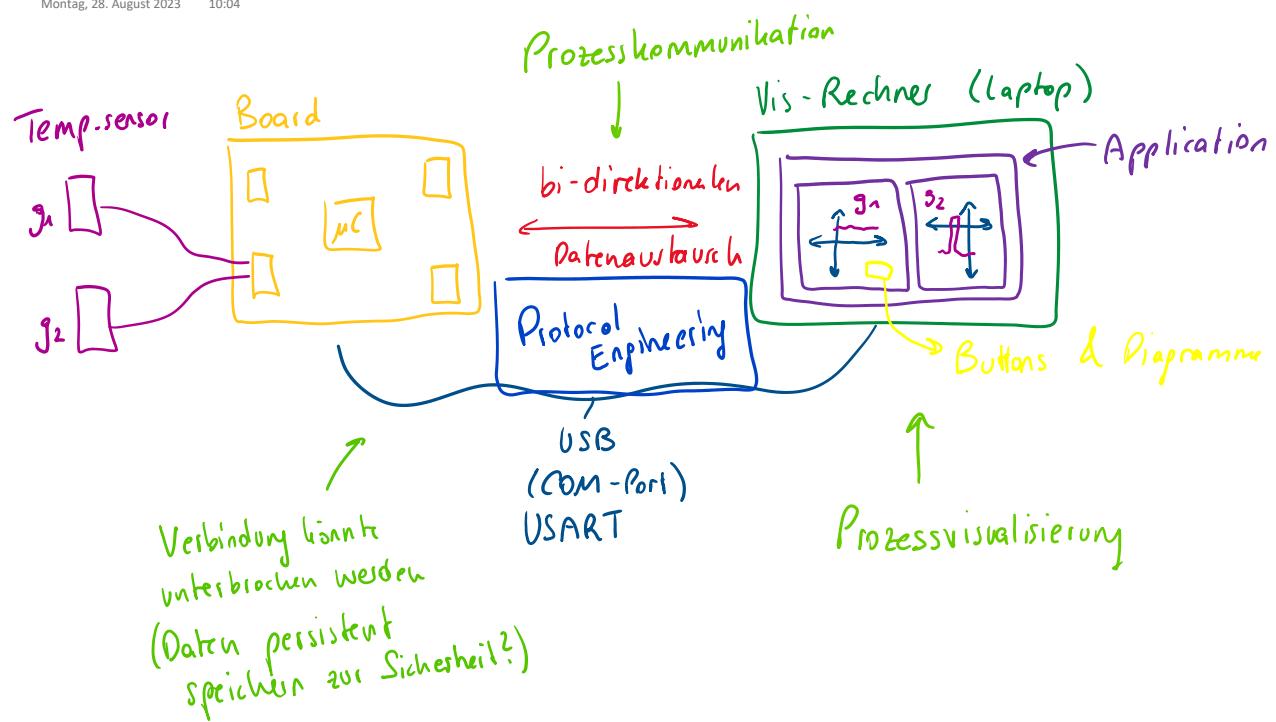
DDRD=0xFF;

```
PORTD=0x01; // LED 0 leuchtet  
  
PORTD |= (1<<1); // LED 1 einschalten  
PORTD <= 2; // LED 3, 2 leuchten  
PORTD &= ((1<<3) | (1<<4)); // LED 4 und 3 ausschalten  
PORTD |= 0x01; // LED 1 einschalten
```

Analyse-Ergebnis wählen:

→ 1 & 2 leuchten

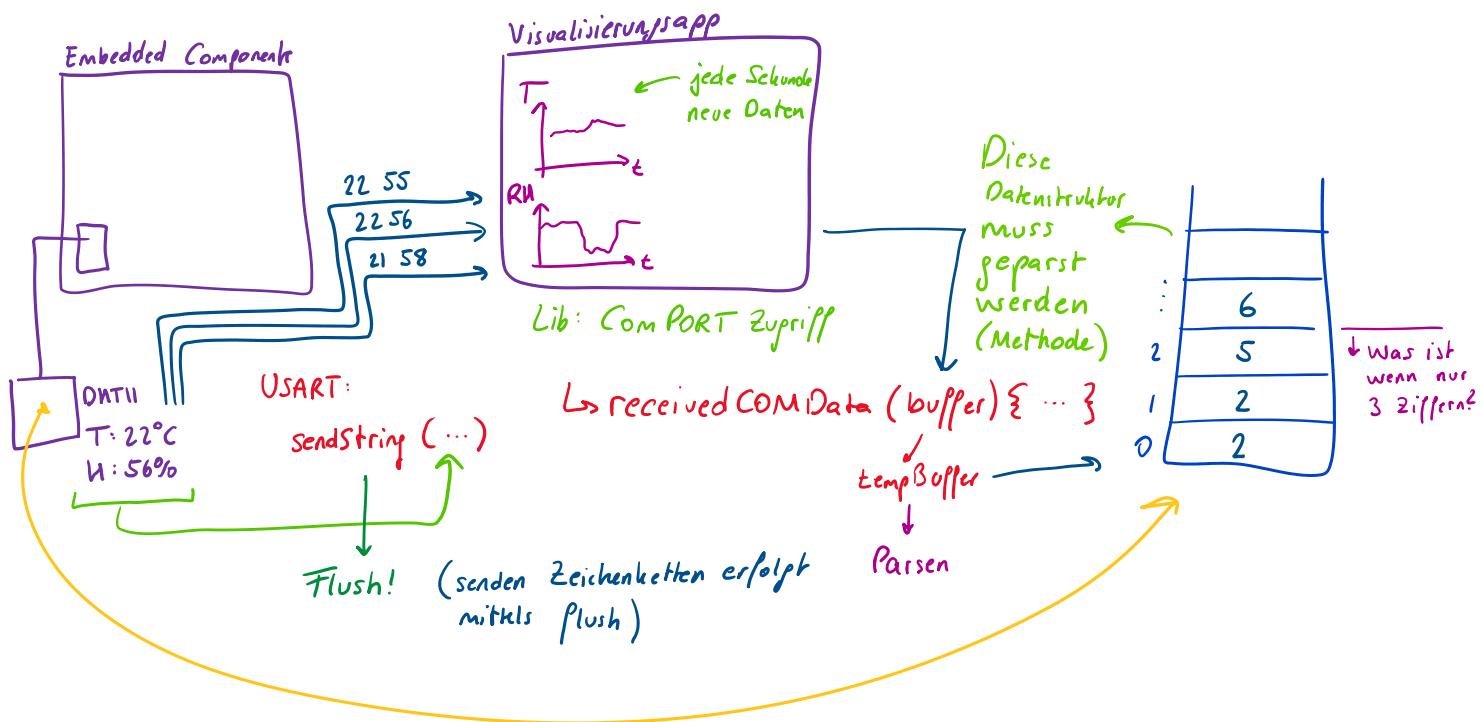
- LED 1 leuchtet, alle anderen nicht.
- LED1 und LED0 leuchten, alle anderen nicht.
- LED3 und LED4 leuchten, alle anderen nicht.
- LED0 leuchtet, alle anderen nicht.



- Ethernet Chip / Shield → W5100
OSI-Layer-4: TCP/IP realisieren
- RTOS - Real Time Operating System → freeRTOS



Protokoll engineering



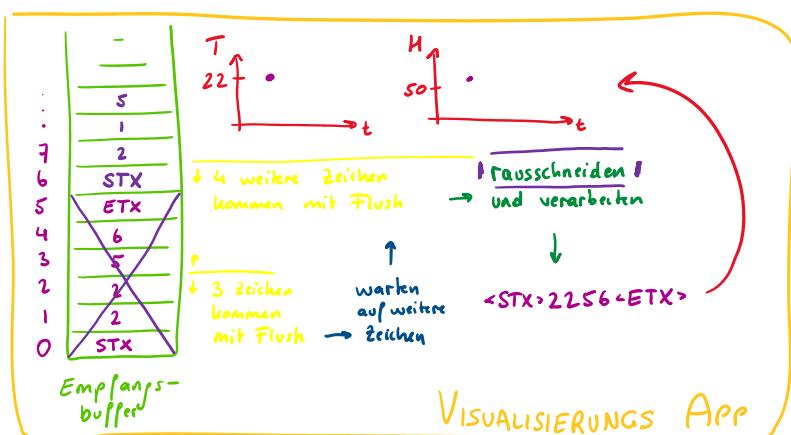
Protokoll Definieren:

- Definition v. Nachrichten (Messages, MSG)

- Framing der definierten Nachrichten

↳ STX] Start of Text → 0x02
 ↳ ETX] ASCII - Steuerzeichen
 ↳ End of Text → 0x03

`<STX> 2256 <ETX>`



↓↓↓↓↓:

`<STX> 2256 <ETX>`

Annahme: <STX> 2256<ETX>

Aber was ist, wenn nur Temp oder Humidity?

<STX> 22<ETX> ? <STX> 56<ETX>

Solution: noch ein Overhead zum deklarieren der Regeln.

nämlich in Form von Nachrichtenname und Parametername
↓ ↓
Parametername → Message
↓ → Param

Idee für obiges Beispiel

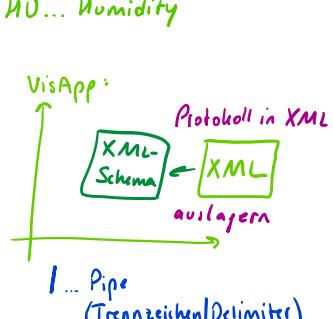
<STX> DATE22|HUS6|<ETX>
 ↑
 Pipe

- Nachrichtenname: 2 Zeichen
- Parametername: 2 Zeichen
- Parameterwert: n-viele Zeichen
wird durch Delimiter beendet

Für die Temp:

<STX> DATE23|<ETX>

DA ... Data
TE ... Temperatur
HU ... Humidity

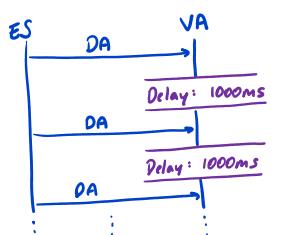


Für die Humi:

<STX> DAHUS6|<ETX>

STEP 3:

MSC für Msg DA
(Message Sequence Chart)

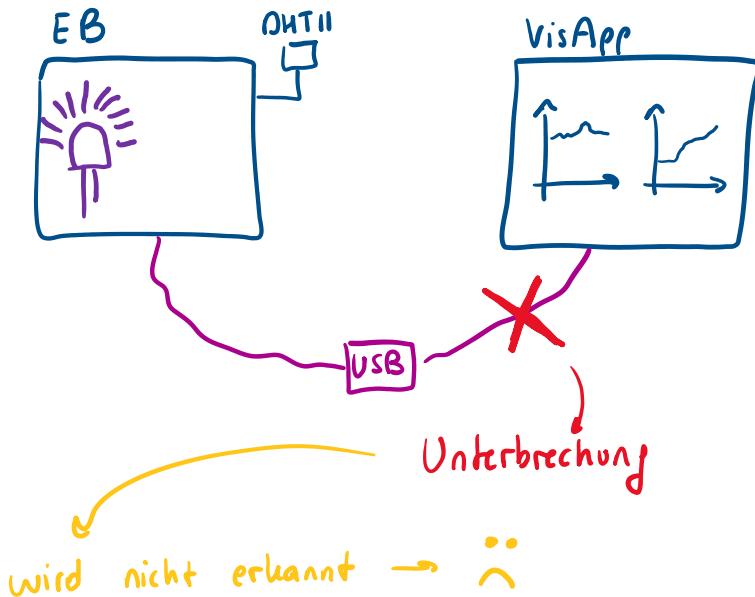


ES... Embedded System
VA... VisApp
DA... Data

Hängt vom eingestellten Modus ab. Dieser kann via Taste oder USART eingestellt werden.

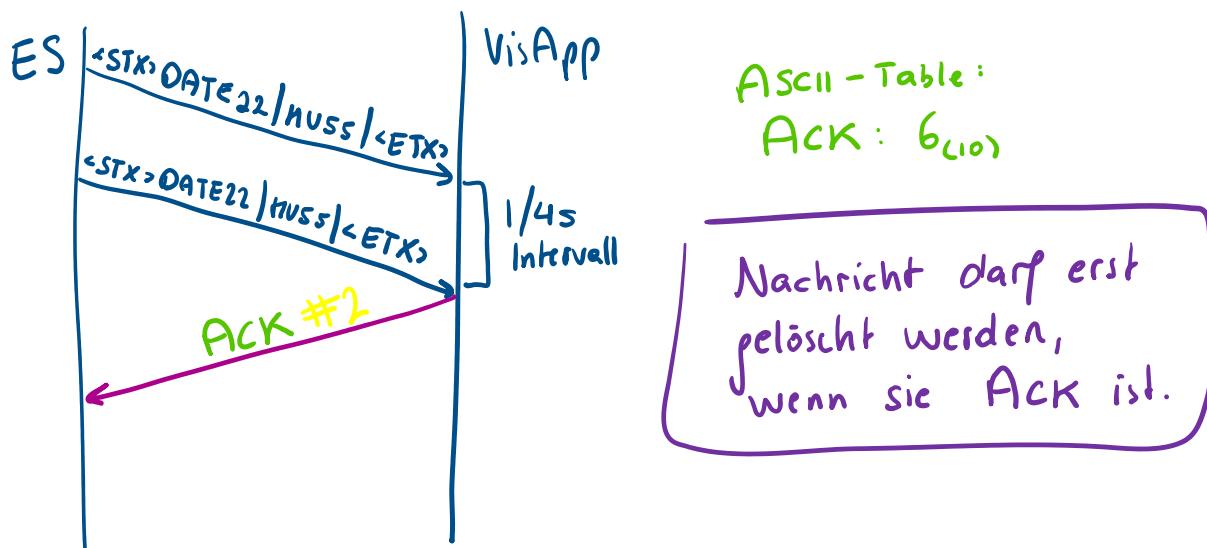
MSG DA entweder im 1000ms oder 4000ms Intervall senden.

Verbindungsabbruch



Wunschscenario:

LED am ES leuchtet, wenn Verbindungsabbruch



Fehlerfall #1

ES bekommt kein ACK

ES Verhalten

- Nachricht 3x im gegebenen Intervall nochmal senden
- Wenn 3x kein ACK → ES LED leuchten lassen
(LinkError)
- andere Messungen laufen im Hintergrund weiter
- Daten temporär persistieren (EEPROM)
- 2250 → 1 Byte Half-Byte: 0-127
- Programm wieder korrekt reinitialisieren

Sequence Number (SN)

<STX> DATE 22 | HUSS | SN1 | <ETX>
<STX> DATE 22 | HUSS | SN2 | <ETX>
<STX> DATE 22 | HUSS | SN3 | <ETX>
<STX> DATE 22 | HUSS | SN4 | <ETX>
<STX> DATE 22 | HUSS | SN5 | <ETX>
<STX> DATE 22 | HUSS | SN6 | <ETX>
<STX> DATE 22 | HUSS | SN7 | <ETX>

<STX> DATE 22 | HUSS | SN1 | <ETX>

<STX> DATE 22 | HUSS | SN8 | <ETX>

1 = 1 ? → „
“

Mit der Sequence Number ist fast eine eindeutige Zuordnung der Antwort ACK möglich.

ISR (USART_RX-vect) {

UDR0; ← 'ACK'

}

ISR (USART_RX-vect) {

UDR0; ← '1'

}

ACK1 Führt zu mehreren ISR aufrufen...

↳ nochmal Rahmen

→ Set Time Intervall auch Pramen

