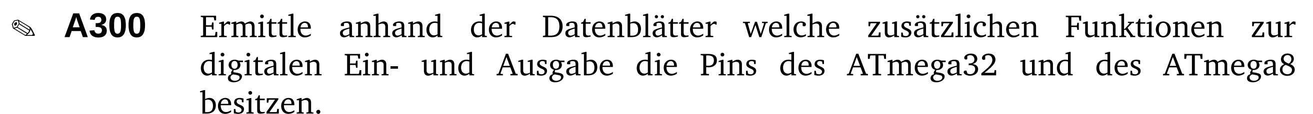
A3 Digitale Ein- und Ausgabe

In diesem Kapitel wird die Grundfunktion der I/O-Anschlussstifte (I/O-Pins), die digitale Ein- und Ausgabe, behandelt.

Bemerkungen: Für die folgenden Programme wird meist die parallele 8-Bit-Schnittstelle Port D verwendet, da sie sowohl beim ATmega8 wie auch beim ATmega32 zur Verfügung steht.

Fast alle Pins der Ausgangsports sind doppelt belegt. Zum Beispiel werden beim ATmega32 vier Pins des Port B für die ISP-Schnittstelle (*In System Programming*) benötigt. Benutzt man Port B jetzt zum Beispiel für die Ansteuerung eines LCD-Displays, so können bei der Programmierung mit angeschlossenem Display Probleme auftreten. Das Display muss zur Programmierung abgeklemmt werden. Man muss also immer genau überlegen welche Pins man benutzen will. Dies ist natürlich auch vom Chip abhängig.

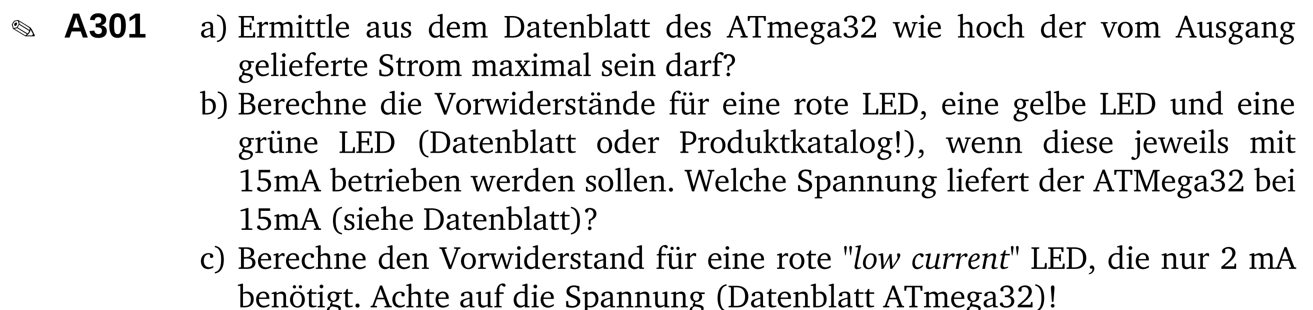


✎ **A300** Ermittle anhand der Datenblätter welche zusätzlichen Funktionen zur digitalen Ein- und Ausgabe die Pins des ATmega32 und des ATmega8 besitzen.

Digitale Daten ausgeben

Das erste Programm `(A303_dig_out_8bit.asm)`

Das erste Programm soll acht LEDs gleichzeitig einschalten. Dazu sind acht LEDs (mit entsprechenden Vorwiderständen) mit den acht Pins von Port D zu verbinden.



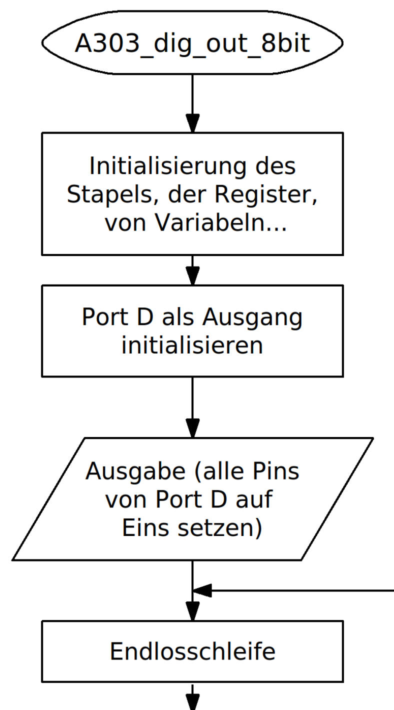
✎ **A301** a) Ermittle aus dem Datenblatt des ATmega32 wie hoch der vom Ausgang gelieferte Strom maximal sein darf?

b) Berechne die Vorwiderstände für eine rote LED, eine gelbe LED und eine grüne LED (Datenblatt oder Produktkatalog!), wenn diese jeweils mit 15mA betrieben werden sollen. Welche Spannung liefert der ATMega32 bei 15mA (siehe Datenblatt)?

c) Berechne den Vorwiderstand für eine rote "*low current*" LED, die nur 2 mA benötigt. Achte auf die Spannung (Datenblatt ATmega32)!

Bemerkung: Die Befehle **`ldi`**, **`out`** und **`rjmp`** wurden schon im vorigen Kapitel besprochen.

Das entsprechende Flussdiagramm sieht folgendermaßen aus:



Folgende Programmzeilen müssen im Initialisierungsteil in der Vorlage hinzugefügt werden:

```
        ;alle Pins von PortD als Ausgang initialisieren
        ldi     Tmp1,0xFF       ;alle Bits im Zwischenspeicher auf Eins
        out     DDRD,Tmp1       ;DDRD = 0b11111111; alle Pins Ausgang
```

Im Datenrichtungsregister für Port D (**DDRD**, ***D****ata* ***D****irection* ***R****egister* ***D***) wird festgelegt ob die einzelnen Pins als Ausgänge oder als Eingänge funktionieren. Eine Eins bedeutet Ausgang, eine Null Eingang.

Das Hauptprogramm sieht folgendermaßen aus:

```
;-------------------------------------------------------------------------------
;       Hauptprogramm
;-------------------------------------------------------------------------------
MAIN:   ser     Tmp1            ;alle Bits im Zwischenspeicher auf Eins
        out     PORTD,Tmp1      ;PORTD = 0b11111111
                                ;Alle PortD-Pins auf High setzen (8 LEDs ein)
END:    rjmp    END             ;Endlosschleife
```

Im Hauptprogramm werden die Ausgangspins von Port D auf High-Pegel (+5V) gezogen. Dazu müssen alle Bits von Port D auf Eins gesetzt werden.

Der "**ser**"-Befehl (*set register*) setzt das Arbeitsregister auf **0xFF**. Mit dem "**out**"-Befehl wird der Inhalt des Arbeitsregisters dann an das SF-Register **PORTD** gesendet.

ser Rd

Setze alle Bits des Register Rd auf Eins (*set all bits in register*).

1 1 1 0	1 1 1 1	d d d d	1 1 1 1

Entspricht dem Befehl **ldi Rd,0xFF**.
Beeinflusste Flags: keine **Taktzyklen: 1**

✎ **A302** Wird die Codezeile mit dem "**ser**"-Befehl zwingend benötigt? Erkläre!?

✎ **A303** a) Erstelle das Programm indem du die Vorlage mit den obigen Zeilen ergänzt.
b) Speichere das Programm unter dem Namen "**A303_dig_out_8bit.asm**" ab. Teste dann das Programm einmal mit deiner Hardware und simuliere das Programm ebenfalls im Studio 4 (Anhang "Das erste Programm"!).
c) Miss alle Spannungen und den Strom an einer Diode!

Bemerkungen: An oberster Stelle soll immer die Übersichtlichkeit und Flexibilität eines Programms stehen. Hier wurde der "**ser**"-Befehl eingeführt, da auch Ziel dieses Kurses ist einen größtmöglichen Teil der Befehle kennen zu lernen. Es kann aber aussagekräftiger und flexibler sein den Befehl **ldi Rd,0xFF** zu verwenden.

Die Codezeile mit dem erneuten Setzen der Bits im Zwischenspeicher (A302) vor der Ausgabe könnte man natürlich weg rationalisieren. Dies ist aber nicht anzuraten, da es die Übersichtlichkeit verschlechtert und schon bei leichten Änderung des Programms (nochmalige Verwendung des Zwischenspeichers für andere Aufgaben) zu schwer auffindbaren Fehlern führen kann.

Im Kurs wird oft die hexadezimale Darstellung gewählt um diese durch häufige Anwendung zu trainieren. Natürlich kann auch die übersichtlichere binäre Darstellung (z.B. **ldi Rd,0b11111111**) verwendet werden.

Das gesamte Programm könnte folgendermaßen aussehen:

```
;*******************************************************************************
;*
;*      Titel:  Mein erstes Programm (A303_dig_out_8bit.asm)
;*      Datum:  20/02/08        Version:        0.2
;*      Autor:  WEIGU
;*
;*      Informationen zur Beschaltung:
;*      Prozessor:      ATmega32        Quarzfrequenz:  intern 1MHz
;*      Eingaenge:      keine
;*      Ausgaenge:      Es sollen acht LEDs mit den acht Pins des PORTD
;*                      verbunden werden (Vorwiderstaende!).
;*
;*      Informationen zur Funktionsweise:
;*      PORTD als Ausgang. Alle LEDs sollen gleichzeitig eingeschaltet werden.
;*
```

```
;************************************************************************************

;------------------------------------------------------------------------------------
;       Einbinden der controllerspezifischen Definitionsdatei
;------------------------------------------------------------------------------------
.NOLIST                         ;List-Output ausschalten
.INCLUDE "m32def.inc"           ;AVR-Definitionsdatei einbinden
.LIST                           ;List-Output wieder einschalten

;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       Programmspeicher (FLASH)     Programmstart nach RESET ab Adr. 0x0000
;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.CSEG                           ;was ab hier folgt kommt in den FLASH-Speicher
.ORG    0x0000                  ;Programm beginnt an der FLASH-Adresse 0x0000
RESET1:  rjmp    INIT           ;springe nach INIT (ueberspringe ISR Vektoren)

;------------------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;------------------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Tmp1 = r16              ;Register 16 dient als erster Zwischenspeicher

        ;alle Pins von PortD als Ausgang initialisieren
        ldi     Tmp1,0xFF       ;alle Bits im Zwischenspeicher auf Eins
                                ;(alternativer Befehl: ser Tmp1)
        out     DDRD,Tmp1       ;DDRD = 0b11111111; alle Pins Ausgang

;------------------------------------------------------------------------------------
;       Hauptprogramm
;------------------------------------------------------------------------------------
MAIN:   ser     Tmp1            ;alle Bits im Zwischenspeicher auf Eins
        out     PORTD,Tmp1      ;PORTD = 0b11111111
                                ;Alle PortD-Pins auf High setzen (8 LEDs ein)
END:    rjmp    END             ;Endlosschleife

;++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.EXIT                           ;Ende des Quelltextes
```

✎ **A304** Schreibe das Programm so um, dass nur die LED an Pin 5 aufleuchtet (neuer Befehl "**sbi**"). Dabei soll auch die Initialisierung so verändert werden, dass alle nicht benötigten Pins als Eingänge konfiguriert sind.
Nenne das Programm "**A304_dig_out_1bit.asm**".

sbi P,b

Setze Bit b im SF-Register auf Eins (*s*et *b*it *i*n *SF-register*).

1	0	0	1	1	0	1	0	P	P	P	P	P	b	b	b

0≤b≤7. Leider können **nur die unteren 32 SF-Register** angesprochen werden (5 Bit). Bei den oberen 32 Registern muss eine Maskierung eingesetzt werden.
Beeinflusste Flags: keine **Taktzyklen: 2**

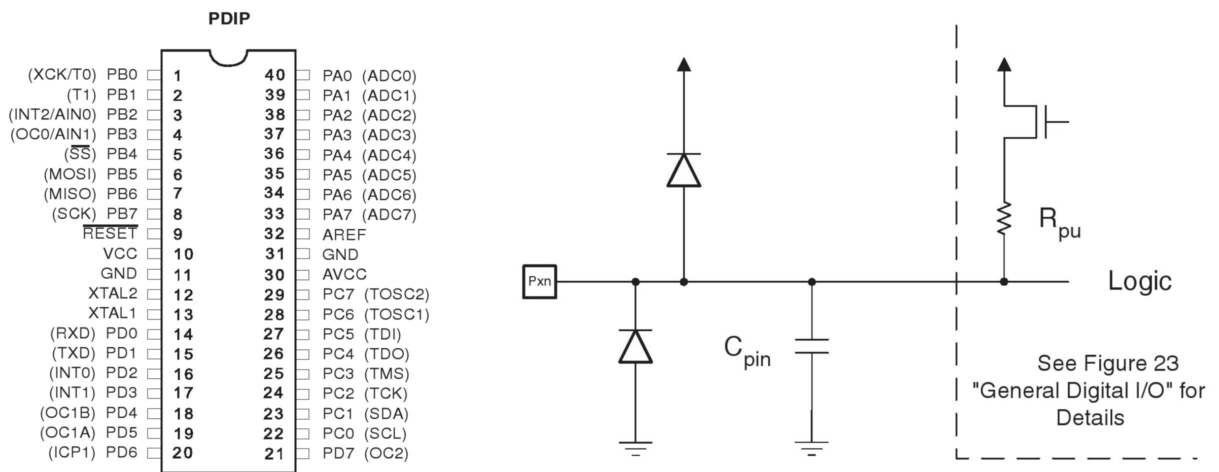
Der ATmega32 besitzt 4 Ein-/Ausgabeports. Das macht insgesamt $4 \cdot 8 = 32$ **individuell ansteuerbare** Pins. Jeder dieser Pins kann als Eingang oder Ausgang konfiguriert werden. Als Eingang kann ein interner Pull-Up-Widerstand dazugeschaltet werden. Jeder als Ausgang geschaltete Pin kann genug Strom liefern um eine LED direkt anzusteuern. Die Pins sind mit zwei Dioden gegen Masse und VCC geschützt. Pins werden mit **PORTxn** oder **Pxn** bezeichnet, wobei **x** der Portname und **n** die Pinnummer ist (beginnend mit Null!). Das sechste Pin von Port D heißt also **PORTD5** (oder **PD5**)!

cbi P,b

Lösche Bit b im SF-Register (clear *bit in SF-register*).

1 0 0 1	1 0 0 0	P P P P	P b b b

0≤b≤7. Setzt Bit auf Null. Leider können **nur die unteren 32 SF-Register** angesprochen werden (5 Bit).
Beeinflusste Flags: keine **Taktzyklen: 2**



Quellen: Atmel Datenblatt ATmega32

Für die Programmierung eines Pins stehen drei Bits in drei unterschiedlichen SF-Registern zur Verfügung:

- Im **Datenrichtungsregister DDRx** (***D****ata* ***D****irection* ***R****egister*) wird (meist im Initialisierungsteil) festgelegt ob ein Pin als Eingang oder Ausgang funktionieren soll. Das Register ist lese- und schreibbar. Das erwünschte Bit (Pin) wird mit **DDxn** bezeichnet.
  DDxn = 1: Ausgang
  DDxn = 0: Eingang
  Nicht beschaltete Pins sind immer als Eingang zu initialisieren.

  Beispiele:

```
ldi     r16,0xF0        ;DDRD = 11110000b
out     DDRD,Tmp1       ;obere 4 Bit als Ausgang, untere 4 Bit unbenutzt
sbi     DDRA,3          ;Pin 3 (PORTA3,PA3) = Ausgang
cbi     DDRA,4          ;Pin 4 (PORTA4,PA4) = Eingang
```

- Über das **Datenausgaberegister PORTx** (***PORTx*** *data register*) werden die Daten an die einzelnen Pins ausgegeben. Es ist les- und schreibbar.
  Ist ein Pin als Eingang initialisiert kann besitzt dieses Register eine andere Funktion!

Mit einer Eins dann dann ein interner Pull-Up-Widerstand zum Eingangspin zugeschaltet werden.

Beispiele:

```
        ldi     Tmp1,'A'        ;ASCII-Code von 'A' ueber die LEDs ausgeben
        out     PORTD,Tmp1      ;
        sbi     PORTB,7         ;oberste LED PB7 einschalten

        cbi     DDRD,0          ;PD0 = Eingang
        sbi     PORTDD,0        ;internen Pull-Up-Widerstand zuschalten
```

- Über das Dateneingaberegister **PINx** (***P****ortx* ***IN****put pins adress*) werden Daten eingelesen. Es ist nur lesbar (an sich ist **PINx** kein klassisches Register, denn die Bits folgen augenblicklich dem Zustand der Pins).

  Beispiele:

```
        in      Tmp1,PINA       ;PINA (8 Bit) einlesen
        andi    Tmp1,0x80       ;maskiere oberstes Bit
        breq    SW_ON           ;falls Null, dann springe nach SW_ON
        ....
        ....
SW_ON:  ....

        sbis    PINA,7          ;ueberspringe naechste Zeile falls PA7=1
        ....
SW_OFF: ....
```

Zusätzlich besteht die Möglichkeit über das Flag (Bit) **PUD** (***P****ull-****U****p* ***D****isable*) im **SFIOR** (***S****pecial* ***F****unction* ***I/O*** ***R****egister*) alle Pull-Up-Widerstände für alle Pins abzuschalten.

Damit erhalten wir folgende Tabelle für **PORTD5**:

	DDD5	PORTD5	Pull-UP	Bemerkung
Eingang	0	0	Nein	Hochohmiger Eingang (Tri-State, Hi-Z). Daten in **PIND5**.
	0	1	Ja	Der Pull-Up-Widerstand ist nur eingeschaltet, wenn **PUD** im **SFIOR** Null ist (*default*). Pull-Up-Widerstände ziehen Strom. Nur Einschalten wenn nötig! Daten in **PIND5**.
Ausgang	1	0	Nein	Gegentakt-Ausgang (Push-Pull). Unterer Transistor legt den Ausgang auf Low-Potential (0).
	1	1	Nein	Gegentakt-Ausgang (Push-Pull). Oberer Transistor legt den Ausgang auf High-Potential (1)

DDRx = Data Direction Register PORTx

Bit	7	6	5	4	3	2	1	0
DDRx	DDx7	DDx6	DDx5	DDx4	DDx3	DDx2	DDx1	DDx0
Startwert	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

DDxn ***Data Direction Bit n (Port x)***

0 Das betreffende Pin als Eingang aktiv. Um Kurzschlüsse zu vermeiden sollten alle nicht benötigten Pins auf Eingang geschaltet werden (Startwert).

1 Das betreffende Pin ist als Ausgang beschaltet.

PORTx PORTx Data Register

Bit	7	6	5	4	3	2	1	0
PORTx	PORTx7	PORTx6	PORTx5	PORTx4	PORTx3	PORTx2	PORTx1	PORTx0
Startwert	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

PORTxn ***PORTx Data Pin n***

a) **Das Pin ist als Ausgang konfiguriert DDRxn = 1**

0 Das Pin liegt auf Masse. Der Strom kann zum Pin fließen (Stromsenke).

1 Das Pin liegt auf VCC und kann als Quelle einen Strom liefern.

b) **Das Pin ist als Eingang konfiguriert DDRxn = 1**

0 Das Pin liegt ist hochohmig (Tri-State).

1 Ist das **PUD**-Bit (*pull-up disable*)im Register **SFIOR** gelöscht (Startwert) so wird intern ein Pull-Up-Widerstand gegen **VCC** geschaltet. Dadurch ist es möglich auf externe Pull-Up-Widerstände zu verzichten, wenn das Pin als Eingang benutzt wird. Bei gesetztem **PUD**-Bit ist der Ausgang hochohmig.

PINx = PORTx INput Pins Adress

Bit	7	6	5	4	3	2	1	0
PINx	PINx7	PINx6	PINx5	PINx4	PINx3	PINx2	PINx1	PINx0
Startwert	-	-	-	-	-	-	-	-
Read/Write	R	R	R	R	R	R	R	R

PINxn ***PORTx INput Pin n***

Unabhängig vom Datenrichtungsregister kann über die **PINx** Adresse der Zustand eines Pins eingelesen werden. Das Pin sollte dabei immer einen definierten Zustand haben (Pull-Up oder Pull-Down-Widerstand zuschalten!).

Bemerkungen: Um mögliche Kurzschlüsse zu vermeiden und Energie zu sparen sollen nicht benutzte Pins immer als Eingang initialisiert werden.

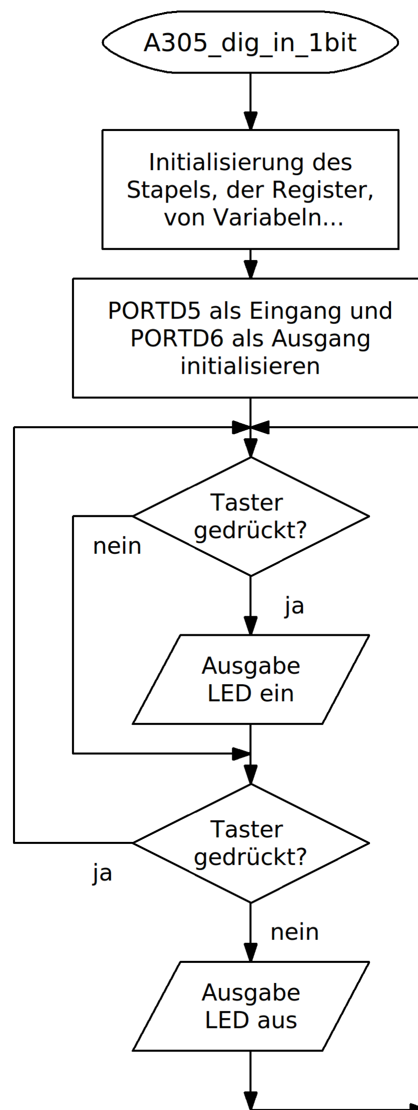
Es sollen immer nur die wirklich benötigten Pins oder Bits angesprochen werden und nicht das ganze Port global, um Fehler wie zum Beispiel das unbeabsichtigte Abschalten eines Pull-Up-Widerstandes zu vermeiden. Zur individuellen Ansteuerung sind die Befehle "**sbi**" und "**cbi**" bzw. "**sbis**" und "**sbic**" besonders gut geeignet.

Digitale Daten einlesen

Das dritte Programm (A305_dig_in_1bit.asm)

In unserem nächsten Programm soll mit Hilfe eines Tasters (nicht entprellter Taster an **PORTD5** gegen Masse![19]) eine LED (an **PORTD6**) eingeschaltet werden Die LED soll nur so lange leuchten wie der Taster gedrückt wurde.

Das entsprechende Flussdiagramm sieht folgendermaßen aus:



19 Da bei den ATmega-Controllern interne Pull-Up-Widerstände zur Verfügung stehen wird meist mit der negativen Logik gearbeitet! Ein betätigter Taster gibt also eine Null zurück!

Eine mögliche Version des Programms kann so aussehen:

```
.EQU    PButt = 5               ;Taster (Push-Button) an PORTD5
.EQU    LED   = 6               ;LED an PORTD6

        ;nur Pin6 von PortD als Ausgang initialisieren
        ldi     Tmp1,0x40       ;DDRD = 0b01000000
        out     DDRD,Tmp1       ;PORTD6 auf 1 (Ausgang) alle anderen 0 (Eingang)

;------------------------------------------------------------------------------
;       Hauptprogramm
;------------------------------------------------------------------------------
MAIN:   sbis    PIND,PButt      ;Ueberspringe naechste Zeile, wenn Bit gesetzt
                                ;Taster nicht gedrueckt (Pin PORTD5 = 1)
        sbi     PORTD,LED       ;Schalte LED ein (PORTD6 = 1)
        sbic    PIND,PButt      ;Ueberspringe naechste Zeile, wenn Bit geloescht
                                ;Taster gedrueckt (Pin PORTD5 = 0)
        cbi     PORTD,LED       ;Schalte LED aus (PORTD6 = 0)
        rjmp    Main            ;Endlosschleife
```

Diese Zeilen werden in die Vorlage eingefügt. Das Programm arbeitet mit einigen neuen sehr praktischen Befehlen um mit SF-Registern (z.B. **`PORTx`** oder. **`PINx`**)zu arbeiten: **`sbis`** (*skip if bit set*) und **`sbic`** (*skip if bit cleared*) ermöglichen es ein einzelnes Bit in einem Ein-/Ausgaberegister zu überprüfen. Wenn die Bedingung erfüllt ist wird die nächste Zeile übersprungen. Mit **`sbi`** (*set bit*) und **`cbi`** (*clear bit*) können einzelne Bits in Ein-Ausgaberegistern gesetzt oder gelöscht werden.

Mit klassischen Befehlen hätte dieses Programmstück wegen der notwendigen Maskierung (siehe nächstes Kapitel) mehr Zeilen benötigt und wäre komplizierter ausgefallen:

`sbis    P,b`

Überspringe nächste Zeile, wenn Bit b im SF-Register gesetzt (Eins) ist (*skip if bit in SF-register is set*).

1 0 0 1	1 0 1 1	P P P P	P b b b

$0 \leq b \leq 7$. Leider können **nur die unteren 32 SF-Register** angesprochen werden (5 Bit).
Beeinflusste Flags: keine Taktzyklen:
1 (kein Sprung), 2 oder 3 (Sprung 1 oder 2 Worte)

`sbic    P,b`

Überspringe nächste Zeile, wenn Bit b im SF-Register gelöscht (Null) ist (*skip if bit in SF-register is cleared*).

1 0 0 1	1 0 0 1	P P P P	P b b b

$0 \leq b \leq 7$. Leider können **nur die unteren 32 SF-Register** angesprochen werden (5 Bit).
Beeinflusste Flags: keine Taktzyklen:
1 (kein Sprung), 2 oder 3 (Sprung 1 oder 2 Worte)

```
(MAIN:  in      Tmp1,PIND       ;PIND einlesen
        andi    Tmp1,0x20       ;Maskierung von PD5 (0b00100000) mit AND
        breq    ON              ;Springe falls Null (PD5 = 0) nach ON
        in      Tmp1,PORTD      ;PORTD einlesen
        andi    Tmp1,0xBF       ;AND-Maske mit 0b10111111
        out     PORTD,Tmp1      ;LED mittels AND-Maskierung ausschalten, PD6 = 0
        rjmp    MAIN            ;Verbleibe in der Endlosschleife
ON:     in      Tmp1,PORTD      ;PORTD einlesen
        ori     Tmp1,0x40       ;ODER-Maske mit 0b01000000
        out     PORTD,Tmp1      ;LED mittels ODER-Maskierung einschalten, PD6 = 1
        rjmp    MAIN            ;Verbleibe in der Endlosschleife )
```

Die Befehle sbis, sbic, sbi und cbi können nur bei den untersten 32 SF-Registern eingesetzt werden!.

Soll das ganze Port eingelesen werden, so geschieht dies mit dem "**in**"-Befehl und dem Dateneingaberegister **PINx**. Beispielsweise zum Einlesen von Port B:

```
in      Tmp1,PINB      ;ganzes PORTB (8 Bit) einlesen
```

Achtung!
Man muss zur Abfrage der Eingänge das Eingangsregister PINx benutzen!

Irrt man sich und verwendet **PORTx** statt **PINx** so liest man den Zustand der Pull-Up-Widerstände ein und nicht den Zustand des Eingangssignals (siehe später).

in Rd,P

Lade den Inhalt (8 Bit) eines SF-Register in das Arbeitsreg. Rd (load *SF-register to register*).

1 0 1 1	0 P P d	d d d d	P P P P

Beeinflusste Flags: keine Taktzyklen: 1

✎ **A305** Erstelle das Programm indem du die Vorlage mit den obigen Zeilen ergänzt. Speichere das Programm unter dem Namen "**A305_dig_in_1bit.asm**" ab. Teste dann das Programm einmal mit deiner Hardware und simuliere das Programm ebenfalls im Studio 4 (Anhang "Das erste Programm!).

Bemerkung: Leider funktioniert das erstellte Programm nicht einwandfrei! Der digitale Eingang hängt, wenn der Taster nicht gedrückt wird, in der Luft. Der Eingang ist nicht definiert und wirkt so als Antenne. Die Helligkeit der LED ändert sich, wenn man mit der Hand in die Nähe des Tasters kommt. Je nach Störpegel kann die LED sogar dauernd eingeschaltet bleiben.

Pull-Up Pull-Down

Damit Aufgabe 305 zufriedenstellend funktionieren kann wird ein Pull-Up-Widerstand (da der Taster gegen Masse liegt!) benötigt. Man kann dazu den internen Pull-Up-Widerstand einschalten. Dazu wird im Ausgaberegister der dem Taster entsprechende Pin auf Eins gesetzt:

```
sbi     PORTD,PButt     ;internen Pull-Up-Widerstand aktivieren
```

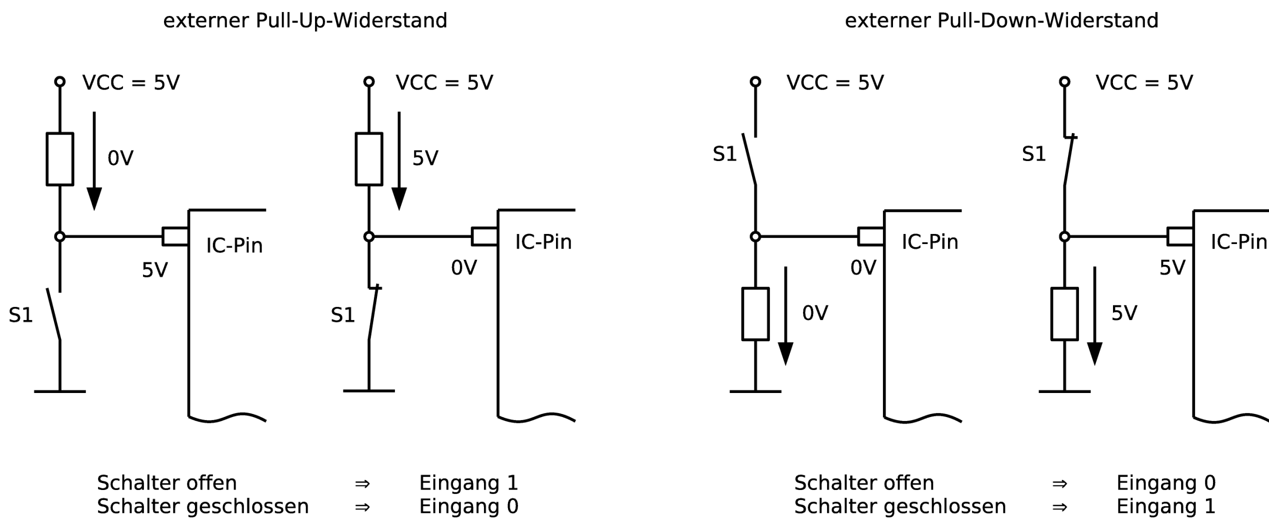
Es besteht natürlich auch die Möglichkeit einen externen Widerstand zu verwenden.

Hier eine kleine Wiederholung wie Taster an einen Eingang angeschlossen werden. Taster gegen Masse benötigen Pull-Up-Widerstände. Taster gegen Betriebsspannung benötigen Pull-Down-Widerstände.

Achtung:
Die Logik ändert sich je nachdem welche Schaltungsvariante benutzt wird.

Bei Pull-Up-Widerständen entsteht eine negative Logik (Taster betätigt entspricht 0V).

Mit Pull-Down- Widerständen entsteht eine positive Logik (Taster betätigt entspricht 5V)!



Schalter offen	⇒	Eingang 1
Schalter geschlossen	⇒	Eingang 0

Schalter offen	⇒	Eingang 0
Schalter geschlossen	⇒	Eingang 1

Pull-Up- und Pull-Down-Widerstände werden benötigt um zu verhindern, dass ein digitaler Eingang einen undefinierten Zustand annehmen kann!

Bemerkungen: Mit einer Invertierung der Eingänge nach dem Einlesen kann natürlich ohne weiteres die Logik verändert werden. Dabei kann mit dem "**com**"-Befehl (Einerkomplement) entweder das ganze Register invertiert werden oder besser mit einer Exklusiv-Oder-Verknüpfung nur das benötigte Pin (Maskierung siehe nächstes Kapitel).

com Rd

Einerkomplement Reg. Rd (*one's complement*).

1 0 0 1	0 1 0 d	d d d d	0 0 0 0

Invertiert gesamtes Register Rd.
Beeinflusste Flags: S, V(0), N, Z, C(1) Taktzyklen: 1

Um die Spannungsquelle des Controller nicht unnötig zu belasten sollen nur die wirklich benötigten Pull-Up-Widerstände aktiviert werden!

✎ **A306** a) Ändere das Programm so um, dass es den internen Pull-Up-Widerstand verwendet und nenne es "**A306_dig_out_1bit_pull_up.asm**".
b) Miss an **PORTD4** den Strom der von Masse in die Schaltung fließt. Stimmt dieser Wert mit dem Datenblatt überein? Welcher Wert hat dann der interne Pull-Up-Widerstand?

✎ **A307** Ändere das Programm jetzt so um, dass der Zustand der LED bei jedem Drücken des Tasters verändert wird (dies nennt man auch noch "*toggeln*" des Zustandes).
a) Erstelle zuerst! das entsprechende Flussdiagramm.
b) Speichere das Programm als "**A307_dig_in_1bit_toggle.asm**" ab.

Tipp:
Mit Hilfe einer Exklusiv-Oder-Funktion lässt sich der Zustand eines einzelnen Bits bzw. Pins invertieren (Befehl **eor**). Bei Schwierigkeiten kann man zuerst das nächste Unterkapitel zur Maskierung durchlesen!

Bemerkung: Wenn ein normaler Taster benutzt wird, wird das Programm noch immer nicht ganz richtig funktionieren, da der Taster prellt. In einem nächsten Kapitel (Modul B) wird erklärt wie man dieses Prellen softwaremäßig beseitigen kann. Wenn die Möglichkeit besteht, dann soll das Programm auch einmal mit einem hardwaremäßig entprellten Taster oder Schalter getestet werden.

Die Maskierung von Daten

Bei den unteren 32 SF-Registern bietet uns die ATmega-Serie bequeme Befehle um sogar einzelne Bits zu adressieren. Auf andere Register (zum Beispiel die oberen 32 SF-Register) trifft dies allerdings nicht zu. Hier muss klassisch mit einer Maskierung des Datenbytes, zwecks Löschen, Setzen oder Toggeln eines oder mehrerer Bits, gearbeitet werden.

Unter der Maskierung von Bits versteht man das zwangsweise Setzen eines Bits auf logisch Null (0) oder logisch Eins (1) durch Verwendung einer geeigneten Bitmaske und einer Booleschen Grundverknüpfung (logische Funktion: AND, OR, NOT), ohne die anderen Bits zu verändern.

Die AND-Verknüpfung:

Eine AND-Verknüpfung mit logisch 0 setzt den Wert einer Bitstelle zwangsweise auf logisch 0!

alter Wert des Bits		Maske		neuer Wert der Bits
0	AND	0	=	0
1	AND	0	=	0

Eine AND-Verknüpfung mit logisch 1 ändert den Wert eines Bit nicht!

alter Wert des Bits		Maske		neuer Wert der Bits
0	AND	1	=	0
1	AND	1	=	1

Die zwangsweise Ausmaskierung einer Bitstelle auf den Wert logisch 0 (Löschen eines Bits) erfolgt durch eine Bitmaske in der an der entsprechenden Stelle eine logische 0 steht und an allen anderen Stellen eine logische 1. Natürlich können auch mehrere Bits auf einmal ausmaskiert werden.

AND: **Null** in der Maske **löscht** das Bit (logisch 0)
Eins in der Maske hat keinen Einfluss

Beispiel: In Register **r20** wird mit Hilfe von Zustandsbits (Flags) der Zustand von acht Werkzeugmaschinen protokolliert. Bei einer Null ist die Maschine aus, bei einer Eins ist sie eingeschaltet.
Es soll überprüft werden ob die dritte und die sechste Maschine wirklich ausgeschaltet sind. Ermittle die zu verwendende Maske.

Bit 2^2 und Bit 2^5 sollen unverändert eingelesen werden. Die Zustände der anderen Bits (Maschinen) interessieren uns nicht, und sie werden durch die Maske gelöscht. Wir benötigen also folgende Maske:

Maschine	8	7	6	5	4	3	2	1
Bit	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
Maske	0	0	1	0	0	1	0	0

Ein Programmcode könnte dann beispielsweise so aussehen:

```
mov     Tmp1,r20       ;Register 20 in den Zwischenspeicher
                       ;kopieren. Noetig, da Register sonst durch
                       ;die logische Operation veraendert wird!
andi    Tmp1,0x24      ;Maskierung von Bit 2^2 und 2^5(00x00x00b)
                       ;mit AND
brne    WARNING        ;Springe falls eine der beiden Maschinen
                       ;nicht aus (oder beide; Z=1) zum Label WARNING
```

mov Rd,Rr

Kopiere den Inhalt von Arbeitsregister Rr (Quelle, *source*) zum Arbeitsregister Rd (Ziel, *destination*) (*copy register*).

0 0 1 0	1 1 r d	d d d d	r r r r

Klassischer Transferbefehl zum Kopieren von **Arbeitsregistern**. Der Inhalt wird nicht verschoben sondern kopiert. Die Quelle Rr (*source*) bleibt unverändert.
Beeinflusste Flags: keine Taktzyklen: 1

andi Rd,K

Log. UND-Verknüpfung des Registers Rd mit der Konstanten K (*logical and with immediate*).

0 1 1 1	K K K K	d d d d	K K K K

Unmittelbare Verknüpfung. Resultat in Rd. Nur Reg. r16-r31.
Beeinflusste Flags: S, V(0), N, Z Taktzyklen: 1

and Rd,Rr

Logische UND-Verknüpfung des Arbeitsregisters Rd mit dem Arbeitsregister Rr (*logical and*).

0 0 1 0	0 0 r d	d d d d	r r r r

Resultat in Rd. Bei Maskierung Maske in Rr.
Beeinflusste Flags: S, V(0), N, Z Taktzyklen: 1

brne k

Bedingter relativer Sprung falls nicht gleich (*branch if not equal*).(k = Adresskonstante)

1 1 1 1	0 1 k k	k k k k	k 0 0 1

Sprünge die an eine Bedingung geknüpft sind werden als Verzweigung bezeichnet (*branch*). Der Befehl wird meist gleich nach einer arithmetischen oder logischen Operation eingesetzt. **Der Sprung erfolgt falls das Resultat der Operation nicht Null wurde** (**Z**-Flag = 0). War diese Operation ein Vergleich (Subtraktion, siehe *compare*-Befehle) so waren beide Operanden nicht gleich.
Beeinflusste Flags: keine
Taktzyklen: 1 (kein Sprung), 2 (Sprung)

Die OR-Verknüpfung:

Eine OR-Verknüpfung mit logisch 0 ändert den Wert eines Bit nicht!

alter Wert des Bits		Maske		neuer Wert der Bits
0	OR	0	=	0
1	OR	0	=	1

Eine OR-Verknüpfung mit logisch 1 setzt den Wert einer Bitstelle zwangsweise auf logisch 1!

alter Wert des Bits		Maske		neuer Wert der Bits
0	OR	1	=	1
1	OR	1	=	1

Das zwangsweise Setzen einer Bitstelle auf den Wert logisch 1 erfolgt durch eine Bitmaske in der, an der entsprechenden Stelle eine logische 1 steht, und an allen anderen Stellen eine logische 0. Natürlich können auch mehrere Bits auf einmal gesetzt werden.

OR: **Eins** in der Maske **setzt** das Bit (logisch '1')
Null in der Maske hat keinen Einfluss

✎ **A308** Die Werkzeugmaschinen 8, 7 und 2 aus dem vorigen Beispiel wurden eingeschaltet. Dies soll jetzt im Register **r20** dokumentiert werden. Berechne die Maske und schreibe die entsprechenden Programmzeilen.

ori Rd,K

Log. ODER-Verknüpfung des Registers Rd mit der Konstanten K (*logical or with immediate*).

0 1 1 0	K K K K	d d d d	K K K K

Unmittelbare Verknüpfung. Resultat in Rd. Nur Reg. r16-r31.
Beeinflusste Flags: S, V(0), N, Z Taktzyklen: 1

or Rd,Rr

Log. ODER-Verknüpfung des Arbeitsregisters Rd mit dem Arbeitsregister Rr (*logical or*).

0 0 1 0	1 0 r d	d d d d	r r r r

Resultat in Rd. Bei Maskierung Maske in Rr.
Beeinflusste Flags: S, V(0), N, Z Taktzyklen: 1

Die XOR-Verknüpfung:

Neben dem Löschen und Setzen einzelner Bits ist es manchmal nötig einzelne Bits zu invertieren (umzuschalten, zu toggeln). Hierzu eignet sich die XOR-Verknüpfung.

Bemerkung: XOR steht für Exklusiv-Oder (*exclusive or*). Andere Bezeichnung sind EOR (siehe Befehl), Antivalenz (Kontravalenz), und ENTWEDER-ODER. Es handelt sich auch um eine Modulo-2-Addition (siehe Halbaddierer)

Eine XOR-Verknüpfung mit logisch 0 ändert den Wert eines Bit nicht!

alter Wert des Bits		Maske		neuer Wert der Bits
0	XOR	0	=	0
1	XOR	0	=	1

Eine XOR-Verknüpfung mit logisch 1 invertiert den Wert einer Bitstelle!

alter Wert des Bits		Maske		neuer Wert der Bits
0	XOR	1	=	1
1	XOR	1	=	0

Das zwangsweise Invertieren einer Bitstelle erfolgt durch eine Bitmaske in der an der entsprechenden Stelle eine logische 1 steht und an allen anderen Stellen eine logische 0. Natürlich können auch mehrere Bits auf einmal invertiert werden.

XOR: **Eins** in der Maske **invertiert** das Bit.
Null in der Maske hat keinen Einfluss.

✎ **A309** Der Inhalt von Register `r21` soll mit der **XOR**-Funktion invertiert werden. Schreibe die dazu nötigen Programmzeilen. Welchen Befehl könnte man hier noch verwenden?

`eor Rd,Rr`

Logische EXKLUSIV-ODER-Verknüpfung des Arbeitsregisters Rd mit dem Arbeitsregister Rr (*exclusive or*).

0 0 1 0	0 1 r d	d d d d	r r r r

Resultat in Rd. Bei Maskierung Maske in Rr. Es existiert kein unmittelbarer Befehl für die EOR-Verknüpfung.
Beeinflusste Flags: S,V(0),N,Z Taktzyklen: 1