

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 1 von 27

## **GRUNDLAGEN MIKROCONTROLLER (Atmel AVR 8 Bit RISC Controller)**

Arbeiten Sie folgende Fragen aus. Diese Kenntnisse werden bei Mitarbeitskontrollen abgefragt!

**Was sind Mikroprozessoren?**

**Was sind Mikrocontroller?**

**Nenne Funktionen von Mikrocontrollern.**

**Nenne und beschreibe die Speicherarten in Mikrocontrollern (4).**

**Was sind Register?**

**Beschreibe die Funktion der Register DDR, PIN und PORT?**

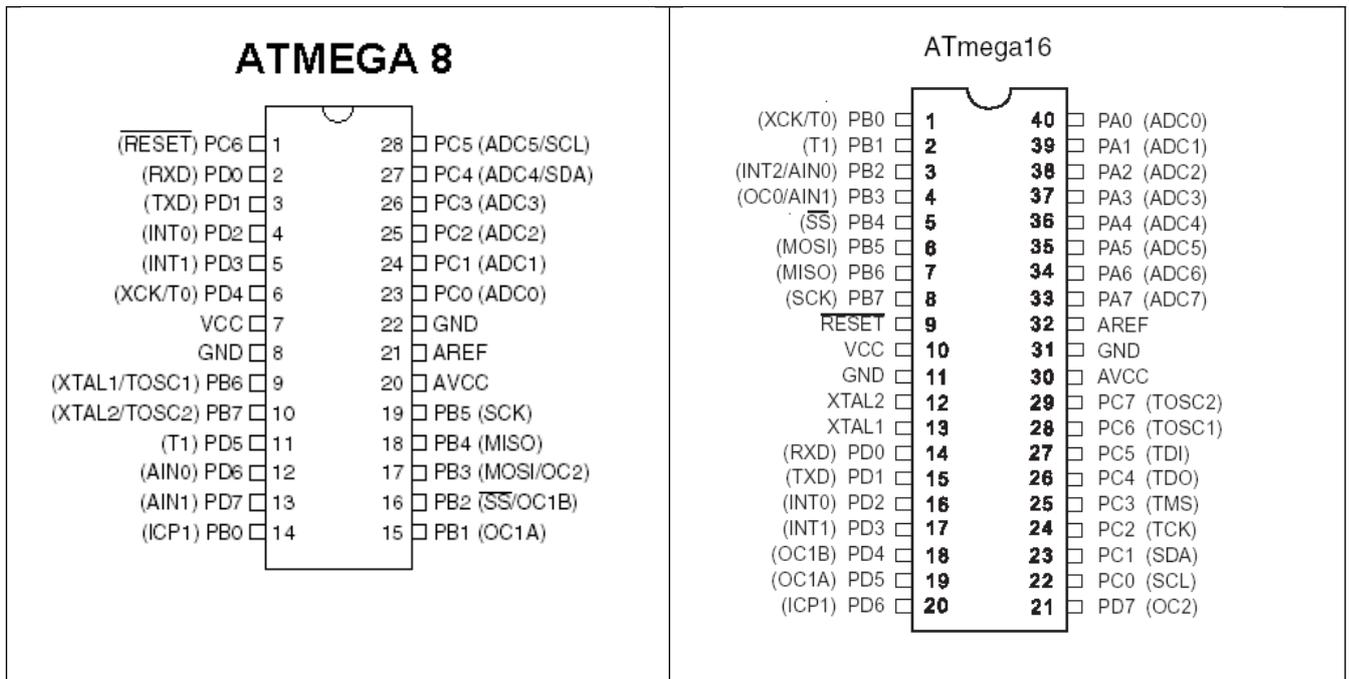
**Wozu werden Interrupts benötigt?**

**Welche Einstellungen benötigt der A/D-Wandler? (4)**

**Welche minimale Beschaltung ist für den Betrieb eines Mikrocontrollers notwendig (Grundschtaltung)?**

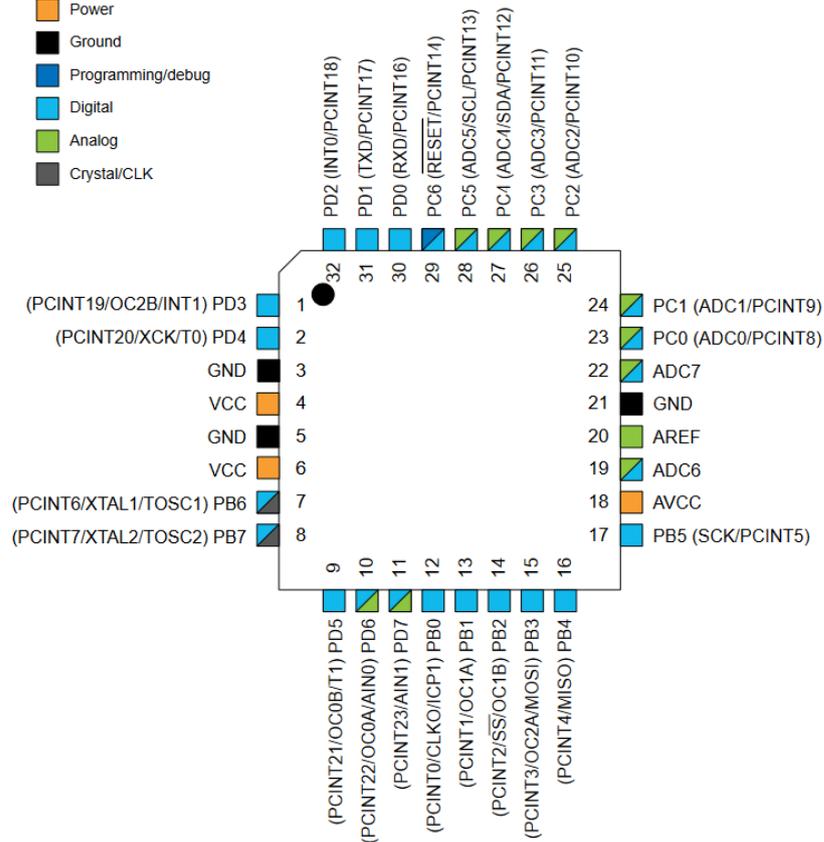
<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 2 von 27

## 1. PINBELEGUNG: Controller und Arduino Platinen

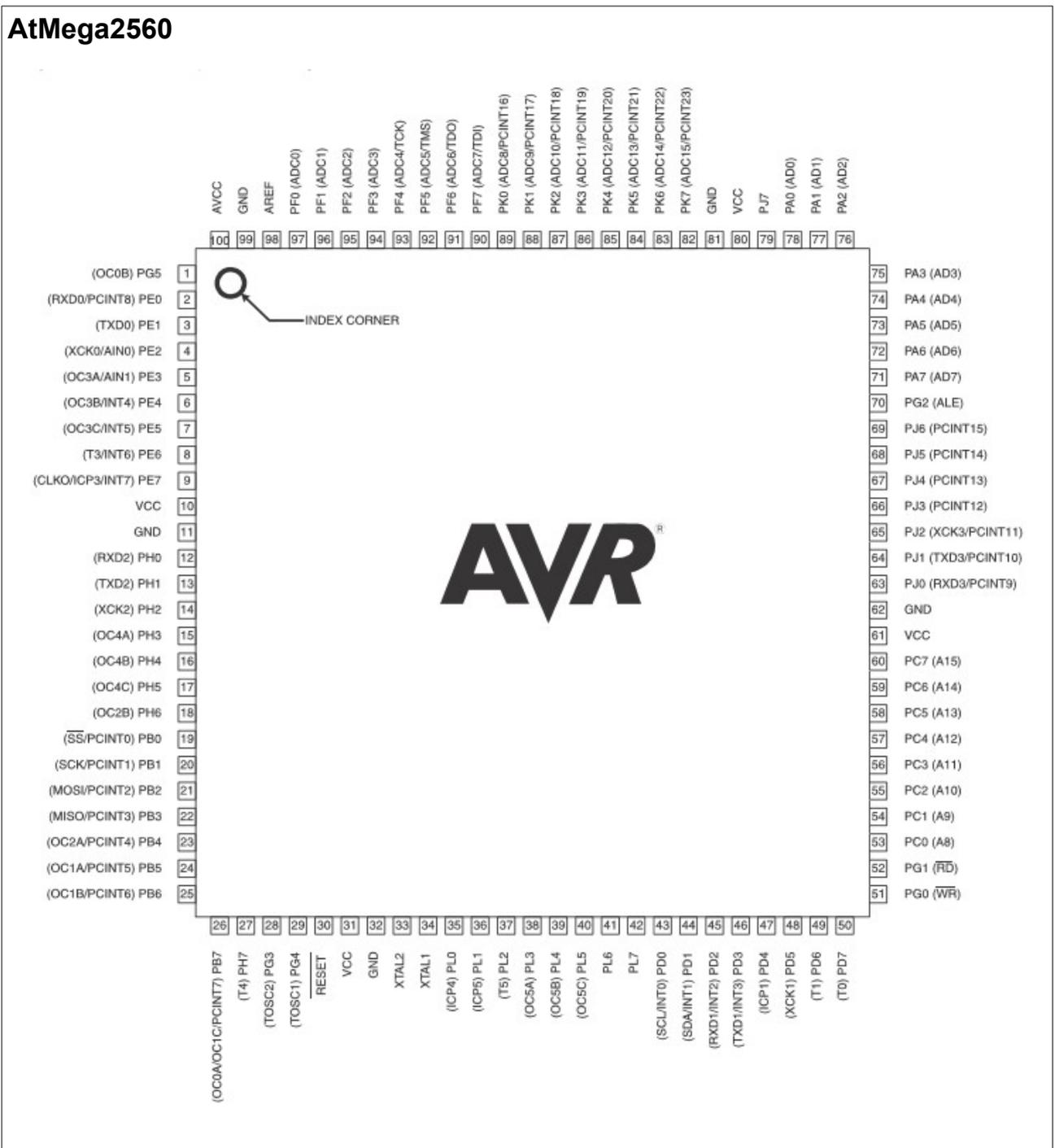


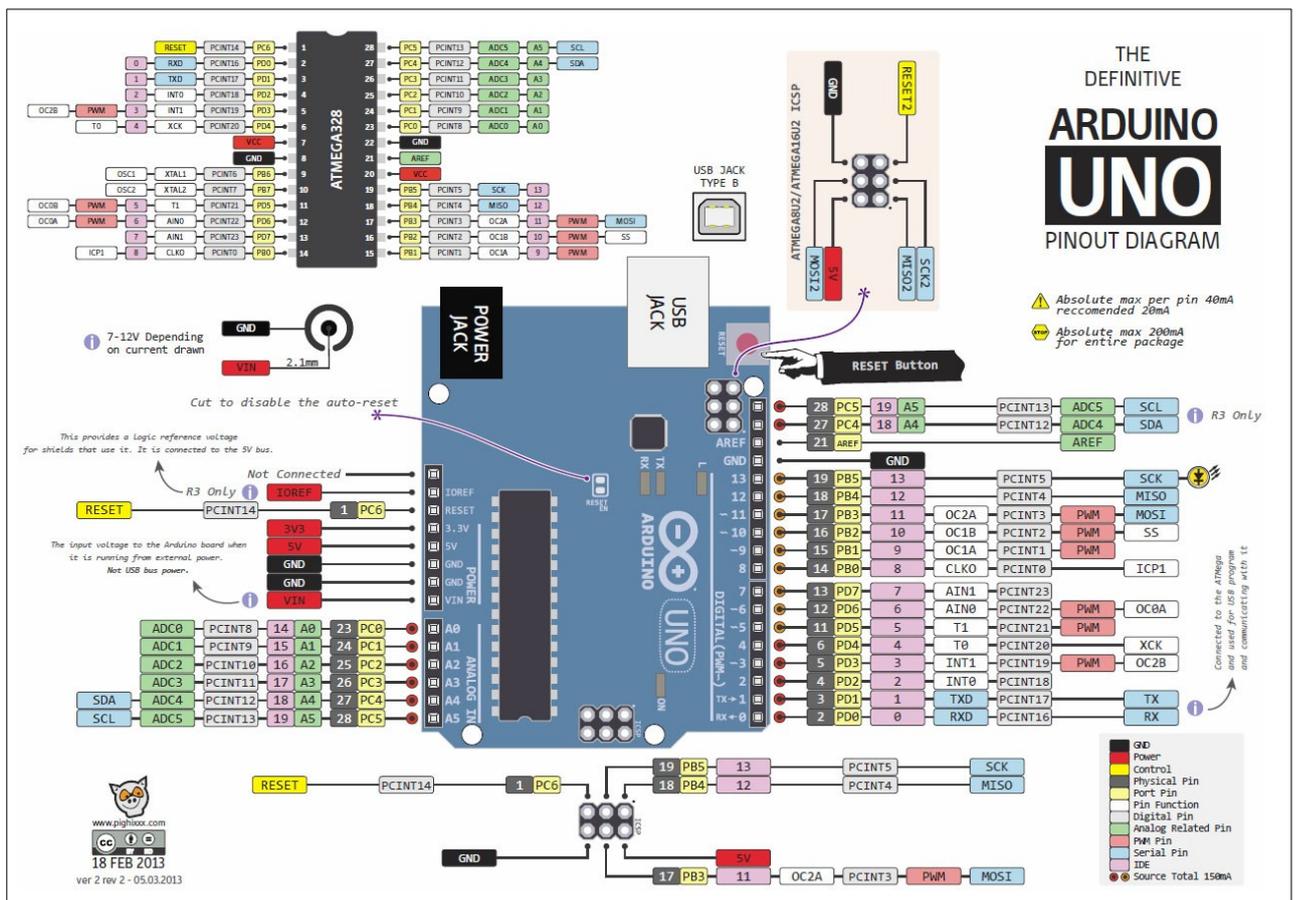
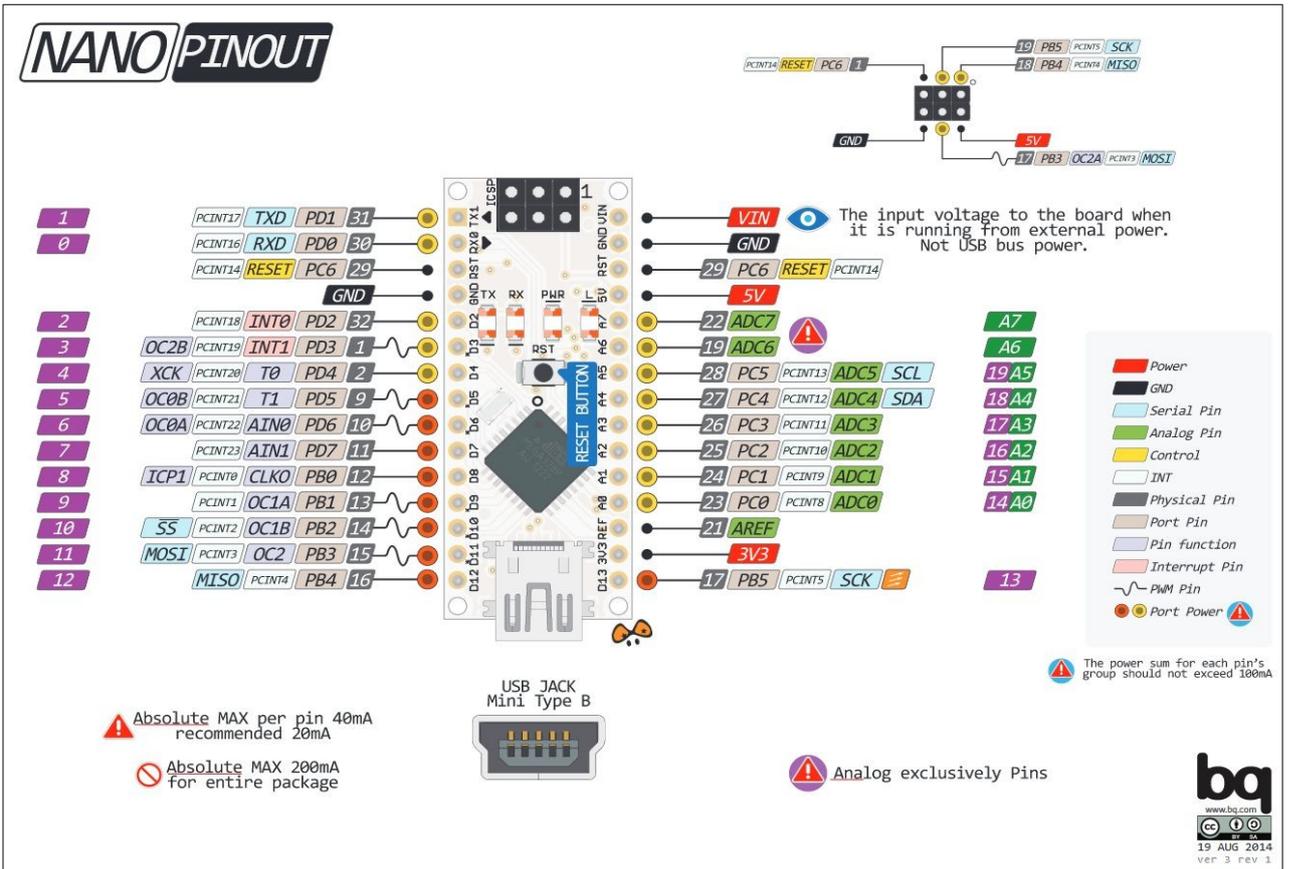
### Atmega328P

- Power
- Ground
- Programming/debug
- Digital
- Analog
- Crystal/CLK



AtMega2560







<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 6 von 27

## 2. Grundlagen der uC-Programmerstellung

### C IDE: Aufbau eines uC-Programms:

#include	<i>Einfügen fertiger C oder H Dateien</i>
Deklaration globaler Variablen	
Funktionsprototypen	<i>Kopfzeile von Unterprogrammen</i>
ev. Unterprogramme	<i>z.b. Einbinden der seriellen Schnittstelle</i>
<b>Hauptprogramm main</b> mit folgenden Teilen:	
Variablendeklaration	<i>Variablen mit Gültigkeit in der main</i>
Vorbereiten der Hardware Ein und Ausgänge festlegen	<i>(DDR, PORT, PIN)</i>
<b>Endlos-Schleife</b> mit dem Programmablauf	
ev. Unterprogramme	<i>z.b. Interrupts, usw</i>

### Arduino IDE: Aufbau eines uC-Programms:

#include	<i>Einfügen fertiger C oder H Dateien</i>
Deklaration globaler Variablen	
Funktionsprototypen	<i>Kopfzeile von Unterprogrammen</i>
ev. Unterprogramme	
<b>Unterprogramm setup()</b>	<i>Wird 1x aufgerufen</i>
Vorbereiten der Hardware Ein und Ausgänge festlegen	<i>Inputs, Outputs, Grundeinstellungen z.b. Einbinden der seriellen Schnittstelle</i>
<b>Unterprogramm loop()</b>	<i>Wird automatisch zyklisch aufgerufen</i>
Variablendeklaration	<i>Variablen mit Gültigkeit in der loop()</i>
Programmablauf	
ev. Unterprogramme	

### Überprüfen der Hardware:

Vor dem Erstellen der Software muss die Funktion der Hardware geprüft werden.  
Daher VOR JEDEM Arbeiten mit der Hardware (Controllerplatine, Roboter) ein fertiges Musterprogramm an die Hardware senden.  
Dadurch werden die Programmierhardware, der Controller und Teile der Peripherie geprüft.

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 7 von 27

### 3. VS Code – Installation, Einrichtung und Anwendung

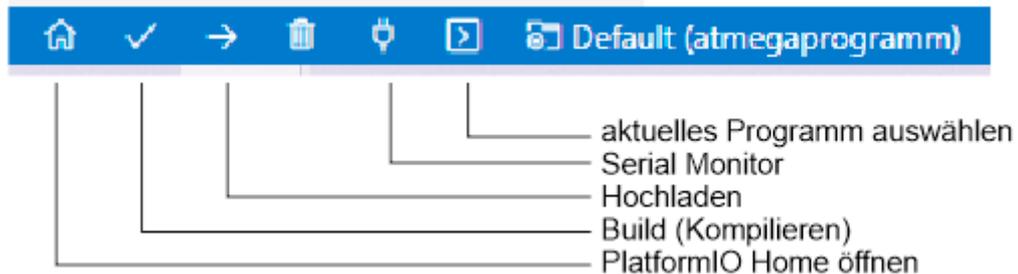
#### Anmerkung zur Installation:

- Download der Entwicklungsumgebung von <https://code.visualstudio.com/download>
- Erweiterung **PlatformIO** installieren

#### Erstellen einer ATMEGA-Anwendung:

Überprüfen sie zuerst die Funktion und USB Verbindung der Controllerplatine! (Geräte-Manager, COM-Port)

1. **PlatformIO Home** auswählen, Button **New Project**
2. Programmnamen eingeben
3. Platine auswählen: „Arduino Nano Atmega328“ oder „Arduino Mega Atmega2560“
4. Framework Arduino auswählen
5. Use default location NICHT auswählen, danach gewünschtes Verzeichnis auswählen
6. Button **Finish**
7. Programmvorlagen aus a.oswald\Atmel\Programmvorlage in den Programmordner kopieren:  
**code\_atmega.c** in den Ordner **src** kopieren  
Datei **main.cpp** aus dem Ordner src **löschen**  
**AODEF.h** in den Ordner **include** kopieren
8. Aktuelles Programm als Arbeitsbereich auswählen (Menüpunkt in der Fußzeile)
9. Kompilieren
10. Programm auf den AtMega Hochladen



#### Achtung:

Arduino Nano-Platine: Atmega328 auswählen, NICHT Variante New Bootloader

Arduino Mega-Platine: Atmega2560 auswählen

Immer aktuelles Programm in der PlatformIO Fußzeile prüfen

#### Arduino IDE:

Die Verwendete Hardware wird im Menü **Werkzeuge**, Punkt **Board**, eingestellt

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 8 von 27

### Platformio.ini Einstellungen:

Vorlage für Platine ArduinoNano:

```
[env:nanoatmega328]
```

```
platform = atmelavr
```

```
board = nanoatmega328
```

```
framework =          oder framework = arduino (Für Arduino-C Befehle)
```

Vorlage für Platine ArduinoMega:

```
[env:megaatmega2560]
```

```
platform = atmelavr
```

```
board = megaatmega2560
```

```
framework =          oder framework = arduino (Für Arduino-C Befehle)
```

Standard bei mehreren Varianten:

```
[platformio]
```

```
default_envs = nanoatmega328
```

Weitere Optionen:

```
; Serial Monitor options
```

```
monitor_speed = 115200
```

```
; Minimalistic printf version
```

```
build_flags = -Wl,-u,vfprintf -lprintf_min
```

```
;Floating point printf version (requires MATH_LIB = -lm below)
```

```
build_flags = -Wl,-u,vfprintf -lprintf_flt -lm
```

Setzen von Fuse Bits: (Beispiel Mega328 interner Oszillator 8Mhz)

```
board_fuses.efuse = 0xFF
```

```
board_fuses.hfuse = 0xD9
```

```
board_fuses.lfuse = 0xE2
```

Verwenden alternativer Programmer: (Beispiel Programmer USBASP)

```
upload_flags =
```

```
-Pusb
```

```
-e
```

```
-E reset, novc
```

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 9 von 27

## 4. Mikrocontroller-Programmierung mit C

### Links und Quellverweis:

<http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial>  
<http://www.kreatives-chaos.com/artikel/avr>  
<http://www.roboternetz.de/wissen/index.php/Avr-gcc>  
<http://www.pronix.de/pronix-4.html>  
<http://www.atmel.com/products/AVR/overview.asp>

### avr-libc Standard C library for AVR-GCC

<https://www.nongnu.org/avr-libc/user-manual/index.html>

### Datentypen (<inttypes.h>)

Typ	Länge	Bereich	Typ	Länge	Bereich
uint8_t	8 Bit	0 ... 255	int8_t	8 Bit	-128 ... 127
uint16_t	16 Bit	0 ... 65.535	int16_t	16 Bit	-32.768 ... 32.767
uint32_t	32 Bit	0 ... 4.294.967.295	int32_t	32 Bit	-2.147.483.648 ... 2.147.483.647
			float	32 Bit	Reelle Zahlen mit ca. 7 Kommastellen

### Schreiben von Bits

Ein ganzes Byte kann durch Befüllen mit einer **Dezimal-, Binär- oder HEX-Zahl** gesetzt werden:

```

x = 7; // Bitmuster 0000 1001 gesetzt (dezimal)
x = 0x22; // Bitmuster 0010 0010 gesetzt (hexadezimal)
x = 0b01000100; // Bitmuster 0100 0100 gesetzt (binär)

```

Einzelne Bits setzt man mittels logischer (Bit-) Operationen:

```

x |= (1 << Bitnummer); // wird ein Bit in x gesetzt (1)
x &= ~(1 << Bitnummer); // wird ein Bit in x gelöscht (0)

```

Setzen mehrerer Bits:

```

x |= (1<<1) | (1<<4); // BIT 1 und 4 in x gesetzt (1)
x &= ~( (1<<0) | (1<<3) ); // BIT 0 und 3 in x gelöscht (0)

```

Bit Toggeln (invertieren):

```

PORTB ^= (1<<0); /* toggle Bit 0, Ex-OR mit Standardschreibweise */

```

Anwendungsbeispiel: Setzen von Registern:

```

DDRB |= (1 << 2); // Port B.2 auf OUTPUT (1)
PORTB &= ~(1 << 2); // Port B.2 auf LOW (0)

```

### Bitsstatus abfragen

```

if ( x & (1<<5) ) /* Fuehre Aktion aus, wenn Bit Nr. 5 in x gesetzt (1) ist */
{
    /* Aktion */
}
if ( !(x & (1<<6)) ) /* Fuehre Aktion aus, wenn Bit Nr. 6 in x gelöscht ist */
{
    /* Aktion */
}

```

Anwendungsbeispiel: Port C, Bit 3 auf Dateneingang schalten und PullUp aktivieren:

```

DDRC &= ~(1 << 3); // Port C.3 auf INPUT (0)
PORTC |= (1 << 3); // PullUp bei Port C.3 aktivieren (1)
if ( PINC & (1<<3) ) /* Port C.3 abfragen */
{
    /* Aktion */
}

```

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 10 von 27

## Zugriff auf Digital-Ports

Alle Anschlüsse des Controllers (Ports) werden über Register gesteuert.  
Dazu sind jedem Anschluss 3 Register zugeordnet:

DDRx	Datenrichtungsregister für Anschluss Portx. x entspricht A, B, C, D usw. Bit im Register gesetzt <b>(1) für Ausgang</b> , Bit gelöscht <b>(0) für Eingang</b> .
PINx	Der Register PIN dient zum Abfragen von Eingängen.
PORTx	Dieses Register wird verwendet, um die Ausgänge anzusteuern. Bei Eingangspins, werden die internen Pull-Up Widerstände aktiviert (1) oder deaktiviert (0).



## C-Makros

Zur Vereinfachung der Schreibweise können Befehle durch C-Makros umbenannt werden:

```
// Setzen und Löschen von Bits als Macro
#define SETBIT1(ADRESS,BIT) (ADRESS |= (1<<BIT))
#define SETBIT0(ADRESS,BIT) (ADRESS &= ~(1<<BIT))
#define SETBITT(ADRESS,BIT) (ADRESS ^= (1<<BIT))
//Macro zur BIT-Abfrage
#define CHECKBIT1(ADRESS,BIT) (ADRESS & (1<<BIT))
#define CHECKBIT0(ADRESS,BIT) (!(ADRESS & (1<<BIT)))
```

Diese C-Makros stehen in der Vorlage AODEF.h zur Verfügung.

Um diese Makros zu verwenden muss im Kopf der C-Datei folgender include stehen:

```
#include "AODEF.h"
```

## Beispiele:

Port B, Bit 2 auf **Datenausgang** schalten und auf LOW setzen:

```
SETBIT1(DDRB, 2); // Port B.2 auf OUTPUT (1)
SETBIT0(PORTB, 2); // Port B.2 auf LOW (0)
```

Port C, Bit 3 auf **Dateneingang** schalten und PullUp aktivieren:

```
SETBIT0(DDRC, 3); // Port C.3 auf INPUT (0)
SETBIT1(PORTC, 3); // PullUp bei Port C.3 aktivieren (1)
```

Port B, Bit 7 **abfragen**:

```
/* Port B.7 abfragen */
if (CHECKBIT0(PINB,7) )
{
    /* Aktion */
}
```

## Arduino IDE:

Für das Festlegen von Ein- und Ausgänge sind Makros vordefiniert:

PinNr 2 auf **Datenausgang** schalten und auf LOW setzen:

```
pinMode(2, OUTPUT);
digitalWrite(2, LOW);
```

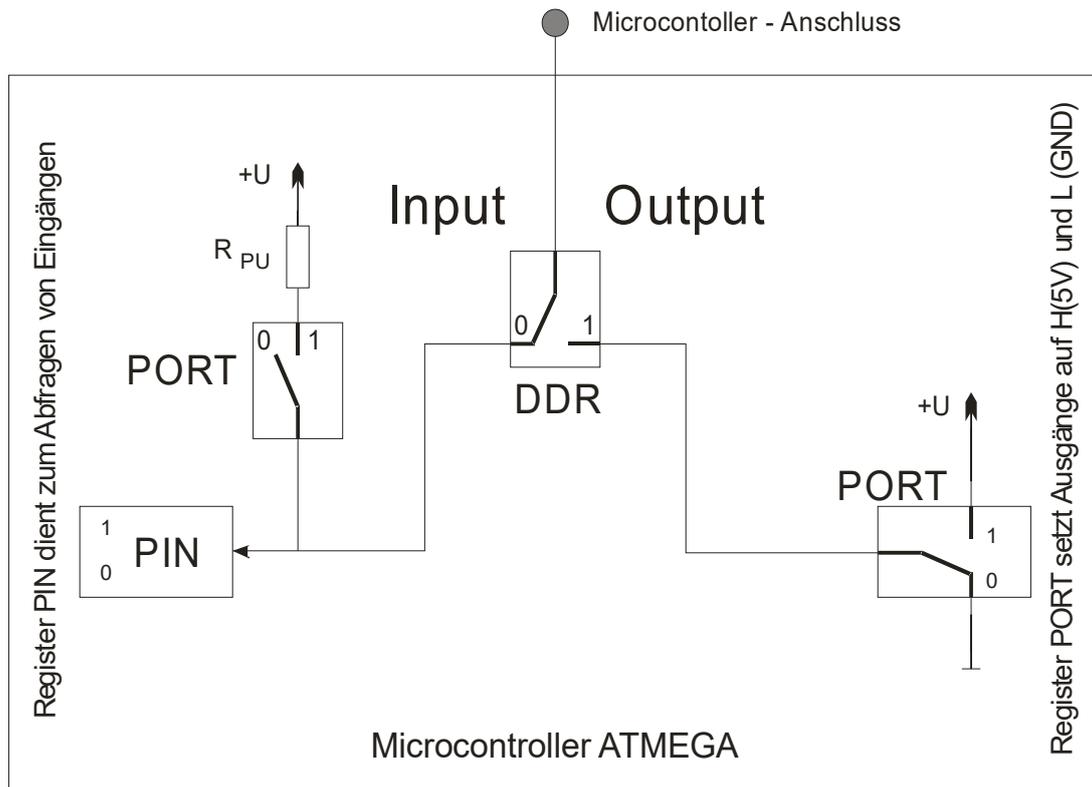
PinNr 4 auf **Dateneingang** schalten und PullUp aktivieren:

```
pinMode(PinNr, INPUT_PULLUP);

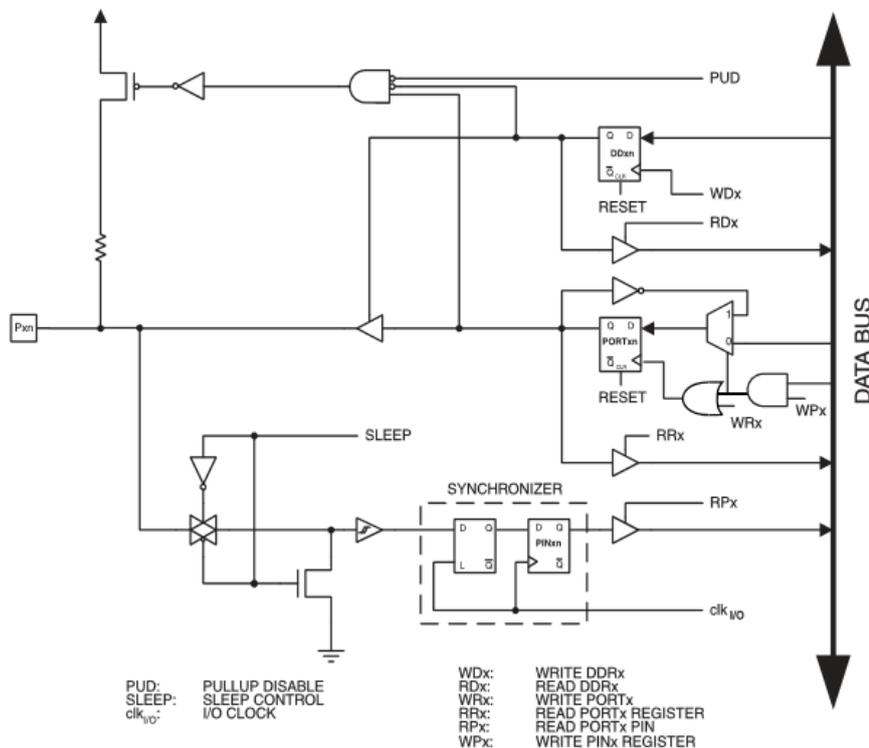
if (digitalRead(PinNr) == LOW)
{
    /* Aktion */
}
```

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 11 von 27

## Registerbeschreibung zum Zugriff auf Digital-Ports



Eingänge / Inputs	Ausgänge / Outputs
Dienen zum Abfragen	Werden vom Programm gesetzt
Für Taster, Sensoren ...	Für LEDs, Transistoren, ...
Pull-UP Widerstand zum Verhindern offener Eingänge möglich (ca 30kΩ)	Max DC Current per I/O Pin: 40.0mA Gesamter erlaubter Strom aller Anschlüsse: 200mA



<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 12 von 27

## Verwenden der seriellen Schnittstelle

Für die Kommunikation zwischen PC und Mikrocontroller und besonders zur Fehlersuche in Programmen kann die serielle Schnittstelle verwendet werden:

### 1.) Übertragungsgeschwindigkeit einstellen und starten

```
void uart_init(void)
int uart_putchar(char c, FILE *stream)
int uart_getchar(FILE *stream)
```

### 2.) Schnittstelle festlegen

```
uart_init();
fdevopen(uart_putchar, uart_getchar); // Standard-IO festlegen
```

### 3.) Senden vom Controller an den PC

```
printf("PROGRAMMSTART");
```

### 4.) Daten Empfangen

```
if ( CHECKBIT1(UCSR0A, RXC0) ) // Ist etwas angekommen?
{
    Zeichen = UDR0; // Das empfangene Zeichen ist im Register UDR
    printf("Zeichen %c empfangen!", Zeichen);
}
```

- Einstellungen der Programmvorlage:  
Übertragung 9600 Bit/s, 8 Datenbits, 1 Stopbit, keine Parität, keine Flusststeuerung
- Ausgabe einer 8 oder 16 bit Zahl ohne Vorzeichen: `printf("Zahl: %u ", Variable);`
- Ausgabe einer 32 bit Zahl: `printf("Zahl: %ld ", Variable);`
- Die Ausgabe von Float-Zahlen durch:
  - Umwandlung der Float-Variable in einen Text `dtostrf (fZahl, 4, 2, TextZahl);`
  - Aktivieren der erweiterten printf-Variante in der PlatformIO.ini

## Arduino IDE:

Für das Festlegen von Seriellen Kommunikation sind Makros vordefiniert:

### 1.) Übertragungsgeschwindigkeit einstellen und starten

```
Serial.begin(9600);
```

### 2.) Senden vom Controller an den PC

```
Serial.print("Hallo Welt");
Serial.println("Hallo Welt");
Serial.write("Hallo Welt");
```

### 4.) Daten Empfangen

```
if (Serial.available())
{
    char Zeichen = (char)Serial.read();
    Serial.print(">>"); Serial.println(Zeichen);

    if (Zeichen=='e')
    { Serial.println("Zeichen e empfangen"); }
}
```

<b>HTBLA Leoding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 13 von 27

## Verwenden externer Interrupts

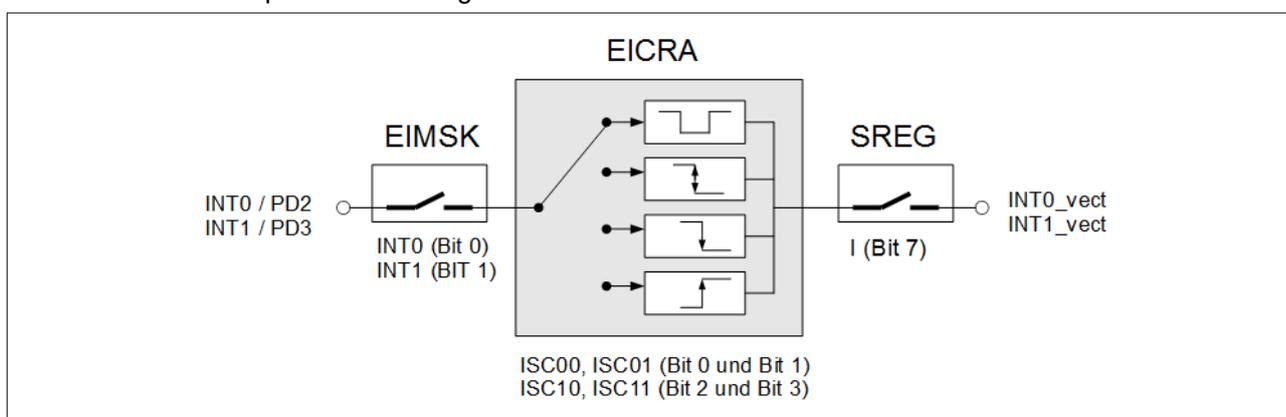
Bei Steuerungen zur Automatisierung von Maschinen und Prozessen erfolgt die Programmbearbeitung laufend über Abfragen der Eingänge. Die Reaktionszeit auf externe Signale beträgt also mindestens eine Zykluszeit und liegt damit im Millisekundenbereich. Bei dieser Art der Programmierung wird auf Eingangssignale oder innere Zustände erst reagiert, wenn sie durch das Programm abgefragt werden. Diese Methode nennt man Polling.

Interrupts sind nützlich, um den uC nicht mit solchen Aufgaben zu belasten. Der Mikrocontroller kann über einen Interrupt auf Ereignisse reagieren, wenn sie eintreten. Wird ein Interrupt ausgelöst, dann wird das aktuelle Programm unterbrochen und zu einer festen Speicherposition (Unterprogramm in C) verzweigt, die von der jeweiligen Interruptquelle abhängt.

Im Resetzustand, nach dem Einschalten der Spannungsversorgung, sind alle Interrupts abgeschaltet. Zum Einschalten eines bestimmten Interrupts müssen erst entsprechende Register gesetzt werden. Die Funktionen zur Interrupt-Verarbeitung stehen in der Includedatei **interrupt.h** der avr-libc zur Verfügung.

Der Atmega328 hat 2 **externe** Interruptquellen an den Pins PD2 und PD3. (Mega 2560 PD0 und PD1)  
Diese Interrupts werden durch RegisterEinstellungen aktiviert bzw. deaktiviert:

1. PORT auf Eingang und ev. PULL UP schalten
2. Signalfanke einstellen: Register EICRA
3. Externer Interrupt aktivieren : Register EIMSK
4. Global Interrupt aktivieren: Register SREG



- External Interrupt Mask Register – **EIMSK** (ATMEGA328 z.B. auf Arduino Platinen)
- External Interrupt Control Register – **EICRA** (ATMEGA328 z.B. auf Arduino Platinen)

Bit	7	6	5	4	3	2	1	0
Name	SE	SM2	SM1	SM0	<b>ISC11</b>	<b>ISC10</b>	<b>ISC01</b>	<b>ISC00</b>
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initialwert	0	0	0	0	0	0	0	0

### ISCxx : Interrupt Sense Control

Diese beiden Bits bestimmen die Flanke für die Interrupterkennung am **INT0**-Pin / **INT1**-Pin

Interrupt 1		Interrupt 0		
ISC11(3)	ISC10 (2)	ISC01 (1)	ISC00 (0)	
0	0	0	0	Low Level erzeugt einen Interrupt
0	1	0	1	Flankenänderung erzeugt Interrupt
1	0	1	0	Fallende Flanke erzeugt Interrupt
1	1	1	1	Steigende Flanke erzeugt Interrupt

- **AVR Status Register – SREG, Bit 7 – I: Global Interrupt Enable**

Um Interrupts zu ermöglichen muss SREG, BIT 7 gesetzt sein.

(Setzen und Löschen dieses Bits kann auch mit den Funktionen `sli()` und `cli()` gemacht werden.)

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 14 von 27

## Interrupt-Unterprogramm

Nachdem die Interrupts aktiviert sind, braucht es noch den auszuführenden Code, der ablaufen soll, wenn ein Interrupt eintrifft. Dazu existiert die Definition (ein Makro) **ISR**.

Als Parameter wird der entsprechende **Interruptvektor** übergeben.

Die Pins PD2 und PD3 müssen nicht abgefragt werden!!

```
#include <avr/io.h>
#include <avr/interrupt.h>          // NICHT VERGESSEN!
#include "AODEF.h"

void main(void)
{
    // Grundeinstellungen
    SETBIT0(DDRD, 2);           // Port D2 auf Dateneingang
    SETBIT1(PORTD, 2);         // PullUp aktivieren

    SETBIT0(EICRA, 0);         // Flanke bei INT0
    SETBIT1(EICRA, 1);         // auf fallend setzen

    SETBIT1(EIMSK, 0);         // Ext. Int0 ein

    SETBIT1(SREG, 7);          // Global Interrupt Enable

    while(1)
    {
        /* MAIN Code */
    }
} // ENDE DER MAIN

ISR(INT0_vect) // Dieses Unterprogramm wird automatisch bei INT0 ausgeführt
{
    /* Interrupt Code */
}
```

**Arduino IDE:** Für das Festlegen der Interrupts sind Makros vordefiniert:

**attachInterrupt( digitalPinToInterrupt( pin ), ISR, mode);**

Parameter:    pin: Arduino pin number  
                     Mögliche Pins bei Uno, Nano, Mini: 2, 3  
                     Mögliche Pins bei Mega2560: 2, 3, 18, 19, 20, 21  
                     ISR: Name des Unterprogramms, das aufgerufen werden soll  
                     mode: Signalfanke zum Auslösen des Interrupt: LOW / CHANGE / RISING / FALLING

```
void setup() {
    Serial.begin(9600);
    Serial.println("Start");

    const byte interruptPin = 2;
    pinMode( interruptPin, INPUT_PULLUP );
    attachInterrupt( digitalPinToInterrupt( interruptPin ), ISR_Pin2, FALLING);
}

void ISR_Pin2()
{
    Serial.println("Interrupt");
}

void loop()
{
    delay(500);
    Serial.write(".");
}
```

Interrupt deaktivieren:

**detachInterrupt( digitalPinToInterrupt( pin ) );**

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 15 von 27

## Interrupt-Unterprogramm für die Serielle Schnittstelle

- Aktivierung des Interrupts für den Empfang
- Aktivierung „Global interrupt Enable“
- ISR-Unterprogramm mit entsprechenden Interruptvektor:

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include "AODEF.h"

void main(void)
{
    //Variante AtMega328p, AtMega2560
    SETBIT1(UCSR0B , RXCIE0); //UART0 Receive Complete Interrupt Enable

    SETBIT1(SREG, 7); // Global Interrupt Enable

    while(1)
    {
        /* MAIN Code */
    }
}

ISR(USART_RX_vect) // ISR für AtMega328P: Arduino UNO/NANO
bzw
ISR(USART0_RX_vect) // ISR für AtMega2560: Arduino MEGA
{
    char Zeichen;

    Zeichen = UDR0; // Daten auslesen, Interruptflag wird gelöscht
    printf("UART-Interrupt : %c ", Zeichen);
}

```

### Arduino IDE:

Für die serielle Schnittstelle existieren keine Interrupts-Makros, es ist nur die Abfrage in der loop möglich:

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

void loop() {

    Serial.println(".");

    while (Serial.available())
    {
        char Zeichen = (char)Serial.read();
        Serial.print(">>"); Serial.println(Zeichen);

        if (Zeichen=='e')
        { Serial.println("Zeichen e empfangen"); }
    }
}

```

<b>HTBLA Leoding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 16 von 27

## 8 BIT - Timer/Counter

Mikrocontroller besitzen eine unterschiedliche Anzahl an programmierbaren Timern. Bei den ATmegas sind das 8-Bit Timer und 16-Bit Timer.

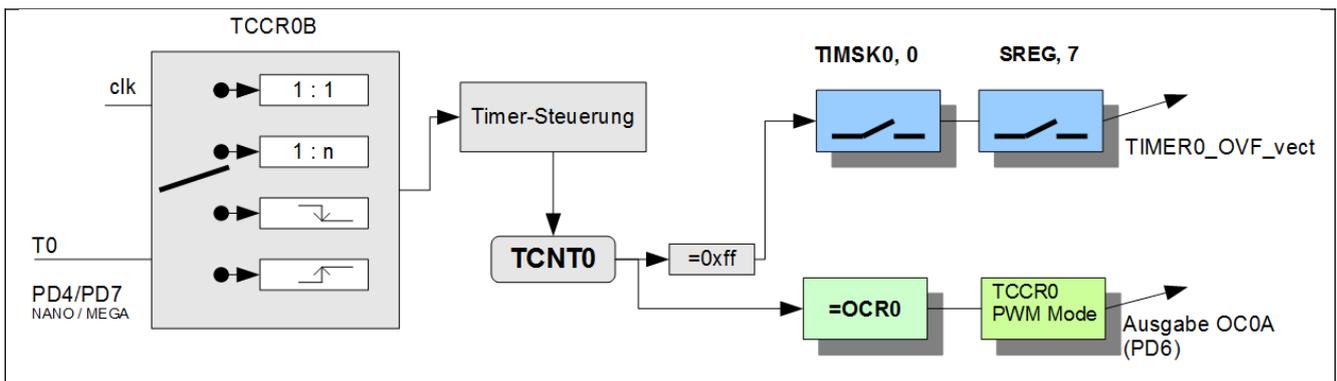
Timer funktionieren nach dem allgemeinen Prinzip, dass eine Zahl (im weiteren als Zähler bezeichnet) je nach Betriebsmodus um 1 erhöht oder um 1 verkleinert wird. Als Taktquelle dafür können externe Pins, der CPU Takt oder der CPU Takt mit Vorteiler (Prescaler) gewählt werden.

Wird eine interne Taktquelle (clk) verwendet, spricht man von einem Timer. Werden externe Pins als Takt genommen, spricht man von einem Zähler (Counter).

Dazu den entsprechenden Pin auf Eingang und ev. Pull Up schalten!

Der Zähler/Timer-Wert ist im Register TCNT0 gespeichert und kann durch Auslesen im Programm verwendet werden. Der Zählerstand kann durch Setzen des Registers auch jederzeit festgelegt werden.

Weiters besteht die Möglichkeit, dass der Timer bei Überlauf einen Interrupt auslösen.



### ➤ Timer/Counter Register – TCNT0

Dieses ist als 8-Bit Aufwärtszähler mit Schreib- und Lesezugriff realisiert. Wenn der Zähler den Wert 255 (0xff) erreicht hat, beginnt er beim nächsten Zyklus wieder bei 0.

### ➤ Timer/Counter Control Register – TCCR0B

Bit	7	6	5	4	3	2	1	0
Name	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00
R/W	R	R	R	R	R/W	R/W	R/W	R/W
Initialwert	0	0	0	0	0	0	0	0

**CS02-CS00: Clock Select**, diese drei Bits bestimmen die Ansteuerung des Timer 0

CS02	CS01	CS00	
0	0	0	Der Timer/Counter wird angehalten
0	0	1	CPU-Takt
0	1	0	CPU-Takt / 8
0	1	1	CPU-Takt / 64
1	0	0	CPU-Takt / 256
1	0	1	CPU-Takt / 1024
1	1	0	Externer Pin TO, fallende Flanke
1	1	1	Externer Pin TO, steigende Flanke

```
//Timer ein, CLOCK/1024
SETBIT1(TCCR0B , CS02);
SETBIT0(TCCR0B , CS01);
SETBIT1(TCCR0B , CS00);
```

```
while(1)
{
    printf("Aktueller Wert des Timers: %u"CR, TCNT0);
}
```

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 17 von 27

## 8 BIT - Timer/Counter Interruptsteuerung

### ➤ Timer/Counter Interrupt Mask Register – TIMSK0

Bit	7	6	5	4	3	2	1	0
Name	Xx	Xx	Xx	Xx	Xx	<b>OCIE0B</b>	<b>OCIE0A</b>	<b>TOIE0</b>
R/W	R	R	R	R	R	R/W	R/W	R/W
Initialwert	0	0	0	0	0	0	0	0

**TOIE0** (Timer/Counter Overflow Interrupt Enable Timer/Counter 0)

Wenn dieses Bit gesetzt ist, wird bei einem Überlauf des Datenregisters des Timer/Counter 0 ein Timer Overflow 0 Interrupt ausgelöst. Das Global Enable Interrupt Flag muss gesetzt sein.

### ➤ AVR Status Register – SREG, Bit 7 – I: Global Interrupt Enable

Um Interrupts zu ermöglichen muss SREG, BIT 7 gesetzt sein.  
(Setzen und Löschen dieses Bits kann auch mit den Funktionen sli() und cli() gemacht werden.)

### Interrupt-Unterprogramm

Nachdem die Interrupts aktiviert sind, braucht es noch den auszuführenden Code, der ablaufen soll, wenn ein Interrupt eintritt. Dazu existiert die Definition (ein Makro) **ISR** mit dem **Interruptvektor** als Parameter.

```
#include <avr/io.h>
#include <avr/interrupt.h>          // NICHT VERGESSEN!
#include "AODEF.h"

void int_init(void)
{
    //Timer ein, CLOCK/1024
    SETBIT1(TCCR0B , CS02);
    SETBIT0(TCCR0B , CS01);
    SETBIT1(TCCR0B , CS00);

    SETBIT1(TIMSK0, TOIE0); // Interrupt Timer 0 Enable
    SETBIT1(SREG, 7);      // Global Interrupt Enable
}

ISR(TIMER0_OVF_vect) // Unterprogramm wird automatisch bei Timerüberlauf ausgeführt
{
    /* Interrupt Code */
}

.... HIER FOLGT DIE main() ...
void main(void)
{
    int_init(); // Aufrufen der Grundeinstellungen

    while(1)
    { /* MAIN Code */ }
}
```

**Arduino IDE:** Timer sind nicht vorgesehen, Zeitsteuerung kann mit der Millis-Funktion gelöst werden

```
unsigned long startmillis = 0;
unsigned long dauer = 1000;

void setup() {
    Serial.begin(9600);
    startmillis = millis();
}

void loop()
{
    if (millis() > (startmillis+dauer))
    {
        startmillis = millis();
        Serial.println("1s");
    }
}
```

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 18 von 27

}

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 19 von 27

## 16 BIT - Timer/Counter

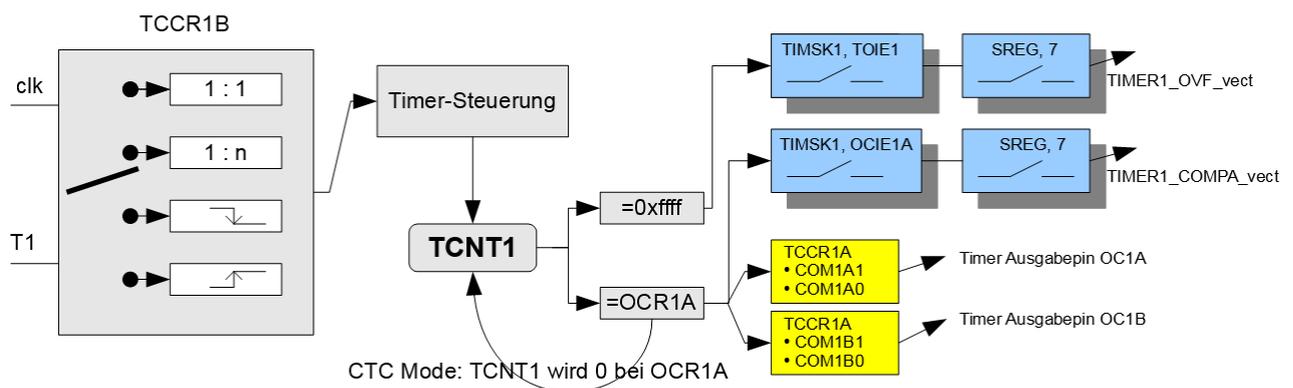
16 Bit-Timer haben verschiedene Betriebsmodi. Der Zähler/Timer-Wert ist im Register **TCNT1** gespeichert und kann durch Auslesen im Programm verwendet werden. Der Zählerstand kann durch Setzen des Registers auch jederzeit festgelegt werden (z.B. als Startwert).

Abhängig von der Taktquelle erhöht sich der Zähler/Timer-Wert ist im Register **TCNT1** durch den CPU-Takt oder durch einen externen Signal..

Der Timer-Modus bestimmt das Verhalten des Timers.

Im „Normal-Modus“ zählt der Timer bis zum Überlauf, dabei kann ein Interrupt ausgelöst werden.

Beim **CTC-Modus** zählt der Timer bis zum Vergleichswert OCR1A und beginnt danach wieder bei 0. Bei Erreichen des Vergleichswertes kann auch ein Interrupt ausgelöst werden oder automatisch ein Ausgang des Controllers angesteuert werden.



### 16 Bit Timer Modi: REGISTER TCCR1A und TCCR1B (!!!)

TCCR1B	TCCR1B	TCCR1A	TCCR1A	Register
WGM13	WGM12	WGM11	WGM10	Bit Betriebsart:
0	0	0	0	Normal-Modus mit Überlauf bei 0xffff (65536)
0	1	0	0	CTC Modus mit Vergleichswert (=Überlauf) bei OCR1A
1	1	0	0	CTC Modus mit Vergleichswert (=Überlauf) bei ICR1

### 16 Bit Timer TAKTQUELLE: REGISTER TCCR1B

CS12	CS11	CS10	
0	0	0	Der Timer/Counter wird angehalten
0	0	1	CPU-Takt
0	1	0	CPU-Takt / 8
0	1	1	CPU-Takt / 64
1	0	0	CPU-Takt / 256
1	0	1	CPU-Takt / 1024
1	1	0	Externer Pin T1, fallende Flanke
1	1	1	Externer Pin T1, steigende Flanke

### 16 Bit Timer INTERRUPT bei Überlauf

```

SETBIT1(TIMSK1, TOIE1); //Aktiviert Interrupt bei Timer 1 Überlauf
SETBIT1(SREG, 7); // Global Interrupt Enable

ISR(TIMER1_OVF_vect) //Aufruf der Interrupt-Routine

```

### 16 Bit Timer INTERRUPT bei Vergleichswert

```

SETBIT1(TIMSK1, OCIE1A); //Aktiviert den OUTPUT COMPARE INTERRUPT
SETBIT1(SREG, 7); // Global Interrupt Enable

ISR(TIMER1_COMPA_vect) //Aufruf der Interrupt-Routine

```

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 20 von 27

### 16 Bit Timer Ausgabepin OC1A und OC1B: REGISTER TCCR1A

COM1A1 COM1B1	COM1A0 COM1B0	
0	0	Normale Port-Funktion für OC1A und OC1B Anschlüsse
0	1	Toggle OC1A / OC1B bei Timer=Vergleichswert
1	0	OC1A / OC1B auf „0“ bei Timer=Vergleichswert
1	1	OC1A / OC1B auf „1“ bei Timer=Vergleichswert

### Programmbeispiel - Sekundentakt

Mit dem 16-Bit Timer wird ein Sekundentakt erzeugt und über die Serielle Schnittstelle ausgegeben. Eine LED am Anschluss OC1A blinkt im Sekundentakt.

```
void timer16init(void)
{
    // Timer 1 Vorteiler auf 1024
    SETBIT1(TCCR1B, CS10);
    SETBIT0(TCCR1B, CS11);
    SETBIT1(TCCR1B, CS12);

    // Timer1 auf Modus CTC (mode 4)
    SETBIT0(TCCR1B, WGM13);
    SETBIT1(TCCR1B, WGM12);
    SETBIT0(TCCR1A, WGM11);
    SETBIT0(TCCR1A, WGM10);

    OCR1A = 15625; // CPU Takt 16000000/1024=15625 Compare Match OCR1A stellen

    // Aktiviert TOGGLE auf OC1A-Anschluss
    SETBIT1(DDRB, 1); // OC1A-Anschluss auf Output (B1 bei Atmega328)
    SETBIT1(TCCR1A, COM1A0); // OC1A-Anschluss TOGGLE
    SETBIT0(TCCR1A, COM1A1); // wenn TCNT1 = OCR1A

    // Aktiviert den OUTPUT COMPARE INTERRUPT, wenn TCNT1=OCR1A
    SETBIT1(TIMSK1, OCIE1A);
    SETBIT1(SREG, 7); //Global Interrupt ein
}

ISR(TIMER1_COMPA_vect)
{
    /* Interrupt Code */
    printf (CR"Eine Sekunde ist um");
}

```

**Arduino IDE:** Timer sind nicht vorgesehen, Zeitsteuerung kann mit der Millis-Funktion gelöst werden

```
unsigned long startmillis = 0;
unsigned long dauer = 1000;

void setup() {
    Serial.begin(9600);
    startmillis = millis();
}

void loop()
{
    if (millis() > (startmillis+dauer))
    {
        startmillis = millis();
        Serial.println("Eine Sekunde ist um");
    }
}

```

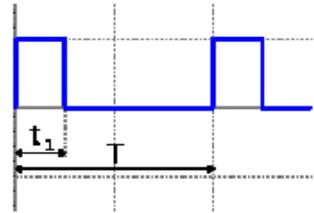
<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 21 von 27

## Pulsweitenmodulation PWM (engl: pulse-width modulation)

Bei der Pulsweitenmodulation (PWM) wird eine elektrische Spannung abwechselnd Ein/Aus-geschaltet. Dabei wird bei konstanter Frequenz ( $T$ ) die Einschaltdauer ( $t_1$ ) des Signales verändert.

- Bei  $t_1=0$  ist das Signal immer ausgeschaltet,
- bei  $t_1=T$  ist das Signal immer eingeschaltet.

Das PWM Signal wird an den Controller-Anschlüssen OCx ausgegeben.



### PWM mit **Timer 0** ausgeben (Programmvorlage für AtMega328)

Der Timer 0 (8 Bit) hat mehrere PWM Betriebsmodi. Die Periodendauer  $T$  hat den Wert 255 (0xff), die Einschaltdauer  $t_1$  kann über den Register OCR0A (OCR0B) festgelegt werden. Nach aktivieren des PWM Mode gibt der Controller an dem Anschluss OC0A (OC0B) das PWM Signal aus.

Vorgehensweise:

- Timer 0 Vorteiler aktivieren ( $T$  im Diagramm) (**REGISTER TCCR0B**) (Siehe Seite 16)
- PWM Modus einstellen (**REGISTER TCCR0A, BIT WGM01 und WGM00**)
- Anschluss OC0 auf Ausgang schalten (Register **DDR**)
- Wert für Ein/Aus Verhältnis ( $t_1$  im Diagramm) festlegen (Register **OCR0A**)
- PWM Ausgabe für Anschluss OCR0 aktivieren (Register **TCCR0A, BIT COM0A1, BIT COM0A0**)

#### Timer 0 PWM Modi: REGISTER TCCR0

TCCR0A	TCCR0A	Register
WGM01	WGM00	Bit Betriebsart:
0	1	8 BIT Phase Correct PWM (0 - 0xff)
1	1	8 BIT Fast PWM (0 - 0xff)

#### Timer 0 Output-Modi: REGISTER TCCR0

TCCR0A	TCCR0A	Register
COM0A1	COM0A0	Bit Betriebsart:
1	0	Normal-Mode mit Ausgang 1 bei während $t_1$
1	1	Invert-Mode mit Ausgang 0 bei während $t_1$

### Beispiel Timer 0 PWM Signal an Ausgang OC0 mit Tastverhältnis 1:3

```

SETBIT0(TCCR0B , CS02); //Takt Vorteiler setzten
SETBIT1(TCCR0B , CS01);
SETBIT0(TCCR0B , CS00);

//PWM Mode 8 BIT Phase Correct PWM
SETBIT0(TCCR0A , WGM01);
SETBIT1(TCCR0A , WGM00);

//Output Normal-Mode mit Ausgang H während t1 und 0 nach Compare
SETBIT1(TCCR0A , COM0A1);
SETBIT0(TCCR0A , COM0A0);

SETBIT1(DDRD, 6); //Anschluss OC0A (D6 bei Mega328) auf Ausgang setzten
OCR0A = 85; //t1 auf T/3 setzten

//Output Normal-Mode mit Ausgang H während t1 und 0 nach Compare
SETBIT1(TCCR0A , COM0B1);
SETBIT0(TCCR0A , COM0B0);

SETBIT1(DDRD, 5); //Anschluss OC0B (D5 bei Mega328) auf Ausgang setzten
OCR0B = 64; //t1 auf T/4 setzten

```

<b>HTBLA Leonding</b>		<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :			Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>		Seite : 22 von 27

## PWM mit Timer 1 ausgeben (Programmvorlage für AtMega328)

Der Timer 1 (16 Bit) hat mehrere PWM Betriebsmodi. Nach aktivieren des PWM Modi gibt der Controller an den Anschlüssen OCR1A bzw ORC1B andauernd das PWM Signal aus.

- Timer 1 Vorteiler aktivieren (T im Diagramm) (**REGISTER TCCR1B**) (Siehe Seite 18)
- PWM Modus einstellen (**REGISTER TCCR1A und TCCR1B**)
- Anschluss OCR1A bzw OCR1B auf Ausgang schalten (Register **DDR**)
- Wert für Ein/Aus Verhältnis (t1 im Diagramm) festlegen (Register **OCR1A** bzw **OCR1B**)
- PWM Ausgabe für Anschluss OCR1A bzw OCR1B aktivieren (Register **TCCR1A**)

### Timer 1 PWM Modi: REGISTER TCCR1A und TCCR1B (!!!)

TCCR1B	TCCR1B	TCCR1A	TCCR1A	Register
WGM13	WGM12	WGM11	WGM10	Bit Betriebsart:
0	0	0	1	8 BIT Phase Correct PWM (0 - 0xff)
0	0	1	0	9 BIT Phase Correct PWM (0 - 0x01ff)
0	0	1	1	10 BIT Phase Correct PWM (0 – 0x03ff)
0	1	0	1	8 BIT Fast PWM (0 - 0xff)
0	1	1	0	9 BIT Fast PWM (0 - 0x1ff)
0	1	1	1	10 BIT Fast PWM (0 - 0x3ff)

### Timer 1 Output-Modi: REGISTER TCCR1A

Anschluss OCR1A		Anschluss OCR1B		Register
TCCR1A	TCCR1A	TCCR1A	TCCR1A	Bit Betriebsart:
COM1A1	COM1A0	COM1B1	COM1B0	Bit Betriebsart:
1	0	1	0	Normal-Mode mit Ausgang 1 bei während t1
1	1	1	1	Invert-Mode mit Ausgang 0 bei während t1

### Taktverhältnis t1 zu T festlegen: REGISTER OCR1A bzw OCR1B

PWM-Art	T	t1
8 Bit	255 (0xff)	0 bis 255
9 Bit	511 (0x01ff)	0 bis 511
10 Bit	1023 (0x03ff)	0 bis 1023

## Programmbeispiel Timer 1 PWM Signal an Ausgang OCR1A und ORR1B

```

SETBIT1(TCCR1B, CS10); //prescaler
SETBIT1(TCCR1B, CS11); //prescaler
SETBIT0(TCCR1B, CS12); //prescaler

SETBIT1(TCCR1A, WGM10); // 10 BIT
SETBIT1(TCCR1A, WGM11); // Phase
SETBIT0(TCCR1B, WGM12); // Correct
SETBIT0(TCCR1B, WGM13); // PWM

SETBIT1(DDRB, 1); //Anschluss OC1A (B1) auf Ausgang setzten
OCR1A = 255; //Vergleichswert auf 255: 1/4 von T
SETBIT0(TCCR1A, COM1A0); //Output Mode:
SETBIT1(TCCR1A, COM1A1); //Normal-Mode mit Ausgang H bei während t1

SETBIT1(DDRB, 2); //Anschluss OC1B (B2) auf Ausgang setzten
OCR1B = 512; ///Vergleichswert auf 512: 1/2 von T
SETBIT0(TCCR1A, COM1B0); //Output Mode:
SETBIT1(TCCR1A, COM1B1); //Normal-Mode mit Ausgang H bei während t1

```

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 23 von 27

## Arduino IDE:

Die PWM Erzeugung wird mit der Funktion **analogWrite(pin, value)** gestartet.

Parameter pin: Arduino-Pin auf den geschrieben werden soll

value: Zykluszeit zwischen 0 (aus) und 255 (immer an)

PWM-Pins und Frequenz:

Uno, Nano, Mini	3, 5, 6, 9, 10, 11	490 Hz (Pins 5 und 6: 980 Hz)
Mega	2 - 13, 44 - 46	490 Hz (Pins 4 und 13: 980 Hz)

```
void setup() {
  analogWrite(3, 128); //PWM Signal an PIN 3, 50% ein
}
```

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 24 von 27

## 5. Mikrocontroller-Programmvorlagen

### Einbinden einer LCD-Anzeige

- Neues Projekt laut Beschreibung erstellen
- Folgende Dateien den Ordner **src** kopieren: code\_atmega.c, lcd.c
- Folgende Dateien den Ordner **include** kopieren: AODEF.h, lcd.h
- h-Datei einbinden: #include "lcd.h"
- Anschlussleitungen des Displays in der Datei lcd.h eintragen
- Befehle für die Vorbereitung des Displays eintragen:  

```
lcd_init(LCD_DISP_ON);  
lcd_clrscr();
```
- Ausgabe an Display: lcd\_puts("Hallo Welt");
- Kompilieren und an Controller senden

Weitere Möglichkeiten:

- Cursor-Position angeben: lcd\_gotoxy(Zeichennummer, Zeilennummer);
- Ausgabe von Variablen:  
Zahlen müssen zur Ausgabe in Text umgewandelt werden:  

```
itoa(i,s,10); (integer to ascii), utoa(i,s,10); (unsigned to ascii),  
ultoa(i,s,10); (unsigned long to a.), dtostrf (f, 6, 3, s); (float to a.)
```

```
#include <stdlib.h>
```

```
uint16_t Zahl=0;  
float fZahl=0;  
char TextZahl[7];
```

```
Zahl=4711;  
utoa (Zahl, TextZahl, 10); // Umwandlung Ganzzahl in einen Text  
lcd_puts(TextZahl);
```

```
fZahl=12.34;  
dtostrf (fZahl, 4, 2, TextZahl); // Umwandlung Kommazahl in einen Text  
lcd_puts(TextZahl);
```



<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 26 von 27

## Arduino IDE:

Die Funktion **analogRead(pinNr)** liest den Wert vom angegebenen analogen Pin ein (10 Bit, 0 bis 1023)

Parameter **pinNr**: Mini, Nano - A0 to A7  
Mega, Mega2560 - A0 to A14

```
void loop() {
  int ADCval = 0;
  ADCval = analogRead( A3 ); // Pin einlesen
  Serial.println(ADCval); // Wert ausgeben
}
```

Die Funktion **analogReference(type)** legt die Referenzspannung fest, die Voreinstellung ist Aref=AVcc (5V)

Parameter **type**: DEFAULT, INTERNAL2V56, oder EXTERNAL

## Verwenden des EEPROM

- Folgende Dateien in einen neuen Ordner kopieren: makefile, code\_arduino3, 4, 6, 8, 10.c, AODEF.h,
- Neues Projekt laut Beschreibung erstellen

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

//fixe Speicherplätze für das Abspeichern festlegen
#define EE_DUMMY 0x000
#define EE_WERT1 ( EE_DUMMY + sizeof( uint16_t ) )
#define EE_WERT2 ( EE_WERT1 + sizeof( uint16_t ) )
#define EE_WERT3 ( EE_WERT2 + sizeof( uint16_t ) )
#define EE_WERT4 ( EE_WERT3 + sizeof( uint16_t ) )
#define EE_WERT5 ( EE_WERT4 + sizeof( uint16_t ) )

void set_eeprom (void)
{
  while(!eeprom_is_ready());
  eeprom_write_word(EE_WERT1, 1313);
}

void get_eeprom (void)
{
  x1 = 4711; //zum Test mit Dummy-Wert befüllen
  while(!eeprom_is_ready());
  x1 = eeprom_read_word(EE_WERT1);
}
```

### *Anmerkungen zum Anwenden der EEPROM Funktionen:*

Der EEPROM-Speicher lässt nur eine begrenzte Anzahl von Schreibzugriffen zu (ca. 100.000), danach wird die Funktion der Zelle nicht mehr garantiert. Bei geschickter Programmierung, wobei die zu beschreibenden Zellen regelmäßig gewechselt werden, kann man eine deutlich höhere Anzahl an Zugriffen erreichen.

Bei Nutzung des EEPROMs ist zu beachten, dass vor dem Zugriff auf diesen Speicher abgefragt wird, ob der Controller die vorherige EEPROM-Operation abgeschlossen hat. Man sollte auch verhindern, dass der Zugriff durch die Abarbeitung einer Interrupt-Routine unterbrochen wird.

Bei Nutzung der Funktionen aus der avr-libc/eeprom.h-Datei ist das gegeben.

Der Inhalt des EEPROMs kann auch mittels Programmieradapter ausgelesen und beschrieben werden. Diese Funktionen könne über das Makefile eingestellt werden.

Der EEPROM-Speicher enthält nach der Übertragung des Programms mittels ISP abhängig von der Einstellung der EESAVE-Fuse die **vorherigen** Daten (programmed = 0) oder den **Standardwert** 0xFF (unprogrammed = 1). Zur Kontrolle sollten Programme immer Standardwerte enthalten und beim Lesen im EEPROM auf 0xFF prüfen!

<b>HTBLA Leonding</b>	<b>Mikrocontrollertechnik</b>	_Atmel_Programieranleitung.odt V1.2j23
Klasse, Name :		Datum :
Übung :	<b>Programmieren von ATMEL AVR 8 Bit RISC Controller</b>	Seite : 27 von 27

## Arduino IDE:

Die Funktion **EEPROM.write(address, value)** schreibt in das EEPROM.  
 Parameter address: Speicheradresse im EEPROM beginnend mit 0  
 value: Wert der abgespeichert wird, zwischen 0 und 255 (byte)

Die Funktion **EEPROM.read(address)** liest aus dem EEPROM aus.  
 Parameter address: Leseadresse im EEPROM

```
#include <EEPROM.h>

int addr = 0;
int value;

void setup()
{
  Serial.begin(9600);
  EEPROM.write(addr, 123); //Schreibt 123 in die Adresse
}

void loop()
{
  value = EEPROM.read(addr);
  Serial.print("Adresse : ");
  Serial.print(addr);
  Serial.print(", Wert: ");
  Serial.print(value);
  Serial.println();
}
```

Komplexere Inhalte und Datentypen können mit den Funktionen:

EEPROM.put()  
 EEPROM.get()  
 EEPROM.update()

bearbeitet werden. ( siehe <https://www.arduino.cc/en/Reference/EEPROM> )

<http://creativecommons.org/licenses/by-sa/3.0/deed.de>

Alle Inhalte dieses Dokuments sind unter einer Creative Commons 2.0 License veröffentlicht.:



- Sie dürfen den Inhalt vervielfältigen, verbreiten und öffentlich aufführen.
- Sie dürfen Bearbeitungen anfertigen.
- Sie müssen den Namen des Autors/Rechtsinhabers nennen.
- Wenn Sie diesen Inhalt bearbeiten oder in anderer Weise umgestalten, verändern oder als Grundlage für einen anderen Inhalt verwenden, dann dürfen Sie den neu entstandenen Inhalt nur unter Verwendung identischer Lizenzbedingungen weitergeben.