

# Systemnahe Programmierung

Technische Informatik II

Roman Wetenkamp

12. Dezember 2021



# Inhaltsverzeichnis

<b>I. Einfache Ein-/Ausgabe-Prozesse</b>	<b>4</b>
1. Einführung	4
2. Hardware	5
3. Grundlegendes zur Programmierung	7
3.1. Register und Ports . . . . .	8
3.2. Eingangssignale . . . . .	9
4. Programmbeispiele	9
4.1. Blinken . . . . .	10
4.2. Tastendruck auslesen . . . . .	10
<b>II. Interrupts</b>	<b>12</b>
5. Interrupts	12
6. Programmbeispiel	13
6.1. Lichtschalter . . . . .	13
<b>III. Timer / Counter</b>	<b>15</b>
7. Timer und Counter	15
8. PWM (Pulsweiten-Modulation)	16
9. Programmbeispiel	17
9.1. LED dimmen über PWM-Timer . . . . .	17

## Vorwort

Liebe Mitstudierende,

Systemnah Programmieren – Klingt komplizierter als es ist. Vielleicht habt ihr euch schon einmal einen Elektronik-Adventskalender oder -baukasten schenken lassen, um damit herumzuspielen, einen Feuchtigkeitssensor für Blumenerde zu entwickeln oder die intelligente Wäscheklammer? In diesem Modul widmen wir uns genau diesem Gebiet, dem hardwarenahen Herumtüfteln an Mikrocomputern.

In diesem Skript notieren wir alles Relevante aus der Vorlesung, ergänzen es um ein paar Aufgaben und Anmerkungen und bereiten uns so auf die Klausur vor. Dieses Skript basiert auf der Vorlesung von Joachim Wagner an der DHBW Mannheim im Studiengang Informatik – Cyber Security, die Passgenauigkeit für andere Dozierende oder Jahrgänge kann ich nicht beurteilen.

*Viel Erfolg!*

Roman Wetenkamp  
Mannheim, den 12. Dezember 2021

**Fehlerfinden** Mein Dank gilt folgenden Personen, die Fehler gefunden, korrigiert und so dieses Skript verbessert haben:

- Daniel Riebel

**Warnung** Das Studium an einer Dualen Hochschule unterscheidet sich von dem Studium an Universitäten oder regulären Fachhochschulen insbesondere dadurch, dass aufgrund der Dualität von Theorie und Praxis meist nur die Hälfte der Zeit zur Vermittlung des Stoffes zur Verfügung steht (wenn dann auch intensiver). Daher gehen Sie bitte nicht davon aus, dass Sie dieses Skript ausreichend auf Klausuren in regulären Vollzeitstudiengängen vorbereitet!

**Hinweis** Dieses Dokument ist kein Vorlesungsmaterial, hat nicht den Anspruch auf Vollständigkeit und enthält mit Sicherheit Fehler. Desweiteren ist es noch lange nicht vollendet (es ist infrage zu stellen, ob es das je sein wird), und doch möchte ich Sie ermutigen, beizutragen! Jegliche Fehler, Probleme oder Anmerkungen können Sie mir gerne über das dazugehörige GitHub-Repository unter der URL <https://github.com/RWetenkamp/sysprog> zukommen lassen. Danke!

## Teil I.

# Einfache Ein-/Ausgabe-Prozesse

## 1. Einführung

Systemnah zu programmieren bedeutet zuallererst, viele Ebenen der Abstraktion, die Betriebssysteme oder Virtualisierungen hinter sich zu lassen und zurückzukehren zu dem grundlegendsten Konzept der Computertechnik, dem Bit.

**Definition 1.** *Ein Bit ist die kleinste Speichereinheit eines Computersystems. Der Wert eines Bits ist entweder 0 für **aus** oder 1 für **ein**. Sprechen wir von einem 8-Bit-Register, so bezeichnen wir damit acht einzeln adressierbare Werte, die jeweils auf 0 oder 1 gesetzt werden können.*

$$8 \text{ Bit} \cong 1 \text{ Byte}$$

Sie werden in Ihren Programmen verschiedene Bitoperationen implementieren müssen, da die zugrundliegende Hardware zu großen Teilen aus Registern besteht, die sie auf diese Weise ansprechen können.

Bezeichnung	Symbol	C-Operatorsymbol	<i>a</i>	<i>b</i>	Ergebnis
Konjunktion	$\wedge$	$\&$	0	0	0
			0	1	0
			1	0	0
			1	1	1
Disjunktion	$\vee$	$ $	0	0	0
			0	1	1
			1	0	1
			1	1	1
XOR	$\oplus$	$\wedge$	0	0	0
			0	1	1
			1	0	1
			1	1	0

Tabelle 1: Übersicht über binäre Bitoperationen

Bezeichnung	Symbol	C-Operatorsymbol	<i>a</i>	Ergebnis
Negation	$\neg$		0	1
			1	0
Linksverschiebung		$\ll$	0000	1 $\ll$ 1: 0001, 1 $\ll$ n: 00...1...0
Rechtsverschiebung		$\gg$	1000	1 $\gg$ 1: 0100, 1 $\ll$ n: 00...1...0

Tabelle 2: Übersicht über unäre Bitoperationen

Im folgenden Kapitel betrachten wir die verwendete Hardware und beginnen damit, die einzelnen Bestandteile und ihre Aufgaben zu erläutern. Wir werden jeweils anhand von

Beispielprogrammen zeigen, wie die einzelnen Komponenten angesprochen und programmiert werden können. Die Programmiersprache hierfür ist C, in der Vorlesung wurde das „Microchip Studio“ als IDE verwendet. Sie können es hier beziehen: [Microchip Studio](#)

## 2. Hardware

Grundlegend ist Ihnen der Aufbau eines Computersystems sicherlich vertraut.

- Die **CPU** (Central Processing Unit), also der Prozessor, ist die zentrale Komponente, die die Befehle ausführt.
- Während der Ausführung speichert die CPU Programmdaten im Arbeitsspeicher, dem **RAM** (Random Accessible Memory). Dieser Speicher wird nur zur Laufzeit genutzt und ist daher nicht persistent.
- Ebenso gibt es einen Massenspeicher, eine Festplatte, auf der Daten persistent gespeichert werden und bei Bedarf gelesen/geschrieben werden.

Diese Grundstruktur gleicht im Wesentlichen dem VON-NEUMANN-PRINZIP und ist sowohl auf klassische PCs anwendbar als auch auf Microcontroller, denen wir uns im Rahmen dieser Vorlesung widmen wollen.

**Bemerkung 1.** *Ein **Microcontroller** ist ein Minicomputer auf einem Chip. Er enthält einen Microprozessor und Speicher.*

Ein solcher Microcontroller findet sich häufig auf Entwicklungsboards, wie z. B. der Arduino-Produktfamilie, die bereits aufgelötete Ports für weitere Komponenten enthalten. Derartige Boards gibt es zuhauf im Markt. Häufig sind die Unterschiede gering, da Arduino-Boards unter einer Open-Source-Lizenz stehen und somit von anderen ohne Weiteres kopiert werden dürfen.

Im Rahmen dieser Vorlesung arbeiten wir mit Arduino-Microcontrollern oder zu Arduino kompatiblen Äquivalenten. Die von uns verwendeten Boards enthalten allesamt folgenden Microprozessor:

### **ATMEL ATmega 328P**

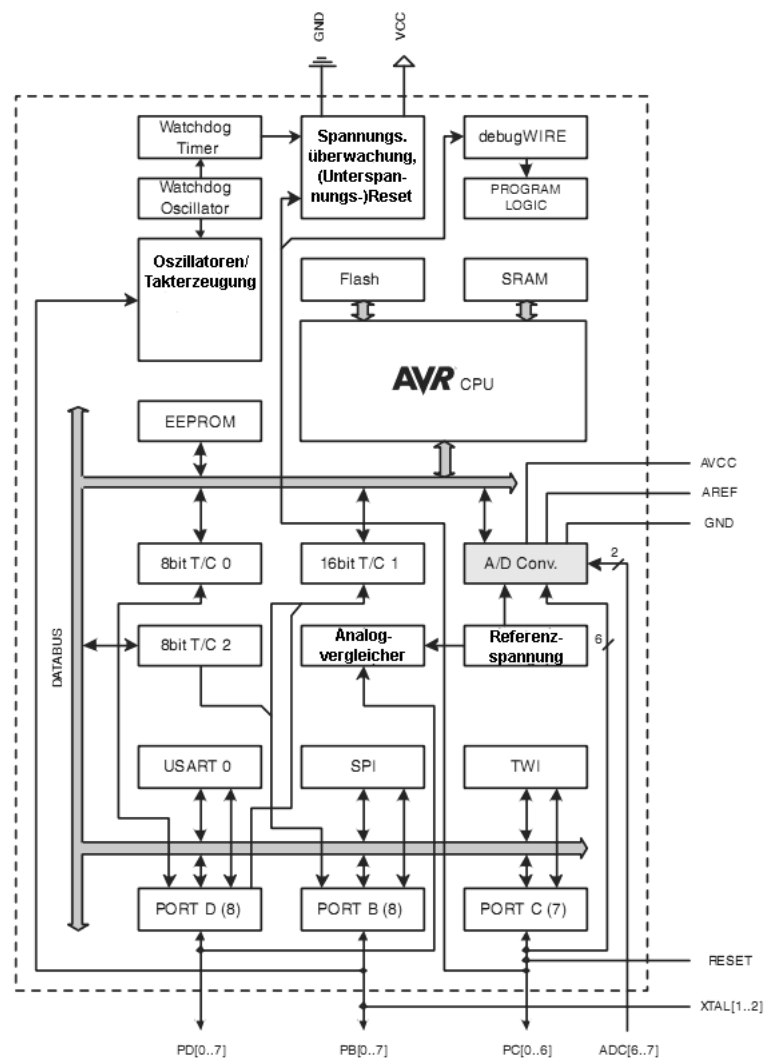
Die Kenntnis des Microcontrollers ist unabdingbar für den weiteren Verlauf der Vorlesung. Die Microcontroller Arduino Uno und Arduino Nano enthalten genannten Prozessor. Wir werden als Referenz jeweils den Arduino Uno verwenden.

**Aufbau** Dieser Microcontroller enthält einige relevante Komponenten, die über einen Bus mit der CPU verbunden sind. Wir entnehmen diese dem folgenden Blockschaltbild.

- **Flash** – Der Flash-Speicher ist neben SRAM und EEPROM einer der drei Speichertypen. Der Flash-Speicher ist unveränderlich und kann lediglich von außen gebrannt werden. Hier wird das aktuelle Skript / Programm gespeichert.
- **SRAM** – Hierbei handelt es sich um einen winzig kleinen Arbeitsspeicher. Dieser Speicher ist nicht persistent und wird von der CPU während der Laufzeit verwaltet und verwendet.



Abbildung 1: Arduino Uno mit ATMEL ATmega328P



- **EEPROM** – Dieser Speicher ist persistent und fungiert als „Festplatte“.
- **Watchdog** – Sollte der Chip nach einer gewissen Zeit kein Lebenssignal mehr sen-

den, setzt der Watchdog den Microcontroller automatisch zurück.

- **A/D Converter** – Die Prozessoren der ATmega-Reihe können analoge Signale in digitale umwandeln. Dafür ist dieser Baustein zuständig.
- **8/16bit T/C** – Diese Bauteile sind Timer/Counter, die zur Prozesssteuerung genutzt werden können. Wir widmen uns diesen Bauteilen später.
- **USART** – Eine serielle Schnittstelle für diversen Datenverkehr ist der USART-Baustein.
- **TWI** – Die TWI-Schnittstelle kann für die Verknüpfung mehrerer Microcontroller untereinander genutzt werden.

Im Allgemeinen wird die Kommunikation mit jeder Hardware/CPU über I/O-Ports abgewickelt.

**Bemerkung 2.** *Gelegentlich kann es zu Verwirrung kommen, da sowohl internen I/O-Ports des Microcontrollers als auch die Anschlusspins für elektronische Bauteile als **Ports** bezeichnet werden. Im Kontext dieses Kurses bezeichnen wir letztere als „Beinchen“.*

**System** Der hier gezeigte Controller folgt der **RISC**-Architektur. RISC steht für „Reduced Instruction Set Controller“. Es hat sich gezeigt, dass mit einem im Vergleich zu Assembler reduzierten Befehlssatz ähnlich effizient, jedoch viel leichter dekodierbar und in der Regel schneller, gearbeitet werden kann. Der Microcontroller enthält von sich aus kein Betriebssystem. Wollen wir ein Programm ausführen, so müssen wir dieses speziell für diesen Microprozessor in einer hardwarenahen Programmiersprache wie Java oder C implementieren.

### 3. Grundlegendes zur Programmierung

Wir werden in dieser Vorlesung – und damit auch in diesem Skript – die Programmiersprache C verwenden. Um die benötigten Bibliotheken nutzen zu können, müssen die passenden Treiber installiert werden. Am einfachsten lässt sich dies durch die Arduino IDE und das zuvor erwähnte Microchip Studio realisieren. Das **Flashen**, also das Aufspielen der Programme auf das Board, erfolgt über USB.

**Programmaufbau** Wie Sie es von C gewohnt sein werden, unterscheiden wir **Programmzeilen** und **Präprozessordirektiven**. Die Befehle, die ausgeführt werden sollen, müssen Bestandteil der `main()`-Funktion des Programms sein.

**Bemerkung 3.** *Jedes Programm, dass Sie flashen wollen, muss eine Endlosschleife enthalten! Andernfalls wird die Operation lediglich ein einziges Mal ausgeführt.*

Angewandt bedeutet dies:

```
#define F_CPU 16000000UL
#include <avr/io.h>

int main() {
    // Initialisierungen
    while(1) {
```

```

    }
    // Hier koennen Sie Ihren Code platzieren
}

```

Sie erkennen die Definition der Konstante `F_CPU` in der ersten Zeile. Hier legen wir fest, dass der Prozessor mit einem Takt von 16 MHz arbeitet.

### 3.1. Register und Ports

Unser Ziel ist es, verschiedene elektronische Bauteile an den Ports des Microcontrollers anzuschließen und mit diesem interagieren zu lassen. Dafür veranschaulichen wir uns zunächst die Pinbelegung des Boardes. Von Relevanz für das nun folgende programma-

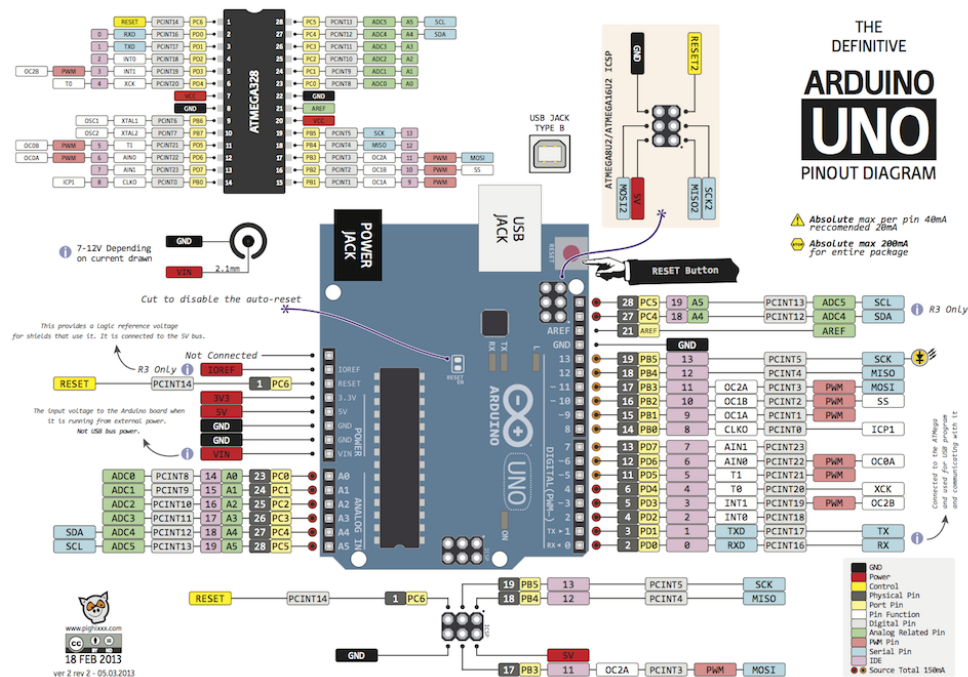


Abbildung 2: Pinbelegungen des Arduino Unos

tische Ansprechen der Pins sind die hier gelb hinterlegten „Port Pin“-Belegungen, also beispielsweise der Port **PB5** an Pin 19 bzw. Onboard-IDE 13. Wir erkennen, dass es drei mal acht dieser Ports gibt, die Register PB (für Port B), PC und PD. Diese Ports nutzen wir – wie angesprochen – um zusätzliche Bauteile anzusprechen.

Für jeden dieser 24 Pins müssen wir festlegen, ob es sich um eine Input- oder Output-Port handelt. Diese Informationen halten wir im **DDR (Data direction register)** fest.

#### Programmzeilen

```
DDRB |= (1 << DDB5); // (1)
```

```
DDRD |= (1 << DDD3) | (1 << DDD4) | (1 << DDD5); // (2)
```

```
DDRD &= ~(1 << DDD2); // (3)
```



- (1) Im Data Direction Register B verschieben wir eine 1 an die Stelle des fünften Bits, um Pin B5 als Ausgang zu definieren. Wir verwenden die Disjunktion, um andere, möglicherweise bestehende Eingaben, nicht zu überschreiben. Alternativ dazu ließe sich das Register selbststredend auch durch eine Angabe der Art `0b00100000` initialisieren, falls man beabsichtigt, nur das Bit 5 als Ausgang zu definieren.
- (2) In dieser Zeile gehen wir ähnlich vor und initialisieren das DDR D. Nun soll jedoch nicht bloß ein einzelnes Bit initialisiert, sondern drei gleichzeitig, die wir gegenseitig disjunktiv verknüpfen.
- (3) Nun bleibt noch, einen Pin wie hier als Eingang zu definieren. Da sich Nullen binär nicht schieben lassen, arbeiten wir hier mit der Negation der 1-Shift-Operation.

Im nächsten Kapitel betrachten wir die Möglichkeiten des Einlesens genauer, insbesondere behandeln wir dort Interrupts.

Mit der Definition des DDRs für die entsprechenden Bits ist die Hälfte getan. Wollen wir nun die Systemspannung von 5 V an einen spezifizierten Port anlegen, gehen wir analog vor:

```
PORTB |= (1 << PORTB5);           // einschalten
PORTB &= ~(1 << PORTB5);         // ausschalten
```

### 3.2. Eingangssignale

In vielen Schaltungen verwenden wir Taster und Schalter, um auf Benutzereingaben zu reagieren. Diese Bauelemente beeinflussen die Ströme, in dem eine Spannung erst bei Tastendruck anliegt oder dann nicht mehr. Der Microcontroller liest diese Spannungen nun aus und interpretiert sie entsprechend mit 0/1 (an/aus). Für dieses Verfahren gibt es zwei Varianten:

- **Polling** – Hierbei wertet das Programm intervallbasiert die anliegende Spannung am Pin des Bauelements aus. Dieses Vorgehen ist ressourcenintensiv und daher nach Möglichkeit zu vermeiden.
- **Interrupts** – Effizienter hingegen ist es, wenn das Programm durch einen vordefinierten Interrupt unterbrochen wird. Dieses Verfahren wird im nächsten Kapitel ausführlicher behandelt.

Eine allgemeine Schwierigkeit beim Einlesen der Daten liegt aufgrund des **Prellens** vor. So verläuft der Spannungswechsel zwischen 0 und 5 V längst nicht so sauber, wie es wünschenswert ist. Stattdessen kann es bis zu 10 Sekunden dauern, bis ein klares Signal vorliegt, da elektromagnetische Spannungen und Wellen die gemessene Spannung verändern. Um diesem Phänomen zu begegnen, werden sogenannte **Pull-Up-Widerstände** eingesetzt, die dafür sorgen, dass die Spannung im nicht gedrückten Zustand „hochgehoben“ wird und im Falle des Tastendrucks spürbar abfällt. Der Widerstand hat hierbei häufig den Wert 4.7 kΩ.

## 4. Programmbeispiele

An dieser Stelle will ich die relevantesten Programme aus der Vorlesung abbilden, um sie reproduzierbar zu machen.



Abbildung 3: Prellen eines Tasters

## 4.1. Blinken

Ein Standard-Beispiel, um zu testen, ob ein Arduino-Board funktionsfähig ist, ist das **Blink**-Beispiel. Wir wollen die Onboard-LED des Boards in einem vordefinierten Intervall zum Blinken bringen.

**Schaltung** Für dieses Beispiel müssen keine weiteren Komponenten an das Board angeschlossen werden. Wir verwenden lediglich die bereits fest verlötete Onboard-LED.

### Programm

```
/*
 * Blinken
 *
 */

#define F_CPU 16000000UL           // Prozessortaktkonstante
#include <avr/io.h>                 // Systembibliothek
#include <util/delay.h>             // Systembibliothek

int main(void)
{
    DDRB |= (1 << DDB5);           // initialisiert B5 als Ausgangspin
    PORTB |= (1 << PORTB5);         // legt Spannung an Port B5 an

    while (1)
    {
        PORTB &= ~(1 << PORTB5);   // entzieht B5 die Spannung
        _delay_ms(500);             // Wartet 500 Millisekunden
        PORTB |= (1 << PORTB5);     // setzt B5 unter Spannung
        _delay_ms(500);
    }
}
```

## 4.2. Tastendruck auslesen

Nun wollen wir zwei Taster mittels Polling auslesen und bei die Onboard-LED bei Tastendruck ein- bzw. ausschalten.

**Schaltung** Zusätzlich zum Entwicklungsboard benötigen wir nun zwei Taster, den wir auf einem Steckbrett befestigen und mit Port D2 und D3 des Boards verbinden. Der zweite Anschluss der Taster wird mit GND (Masse) verbunden. Bezogen auf obige Grafik handelt es sich dabei um die Pins 16, 17 und 20 (GND).

### Programm

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

int main(void)
{
    DDRD &= ~(1 << DDD2);    // D2 ist Eingabe
    DDRD &= ~(1 << DDD3);    // D3 ist Eingabe

    // Pull up
    PORTD |= (1 << DDD2) | (1 << DDD3);

    DDRB |= (1 << PORTB5);    // Onboard-LED ist Ausgabe
    PORTB &= ~(1 << PORTB5); // Onboard-LED ist initial aus

    while (1)
    {
        // Auslesen der Taster
        if (! (PIND & (1 << DDD2))) {
            // Einschalten der LED
            PORTB |= (1 << PORTB5);
        }

        if (! (PIND & (1 << DDD3))) {
            // Ausschalten der LED
            PORTB &= ~(1 << PORTB5);
        }
    }
}
```

### Aufgaben

1. Nennen Sie die drei unterschiedlichen Arten von Speicher. Erläutern Sie die Unterschiede.
2. Erläutern Sie das Vorgehen, um eine weitere LED vom Board aus unter Spannung zu setzen.
3. Schließen Sie eine LED an Port B3 und einen Taster an Port D2 an. Entwickeln Sie ein Programm, das beim Druck des Tasters zwischen der Onboard-LED und der externen LED umschaltet.

# Teil II.

## Interrupts

### 5. Interrupts

Die zweite Variante, um mit Eingaben umzugehen, sind Interrupts.

**Definition 2.** *Unter einem Interrupt verstehen wir eine kurzfristige und unmittelbare Unterbrechung der Programmausführung, um einen kurzen, zeitlich kritischen Vorgang abzuarbeiten.*

**Motivation** Dem Kontext von Eingaben nähern wir uns durch folgende Betrachtungsweise: Während wir beim Polling fortwährend überprüfen, ob eine Bedingung eingetreten ist – wie z. B. ein Tastendruck – und dann auf dieses Ereignis reagieren, operieren wir bei Interrupts anders. Wir melden an, dass wir bei einem bestimmten Ereignis unterbrochen werden wollen. Tritt dieses Ereignis nun ein, wird unsere Programmausführung unterbrochen und die Behandlung des Ereignisses wird ausgeführt. Auf diese Weise besteht keine Notwendigkeit mehr, ressourcenintensiv die Bedingung fortwährend zu überprüfen, sondern viel mehr ergeben sich Möglichkeiten der Nebenläufigkeit.

**Umsetzung** Wenn wir erneut die Abbildung 3.1 betrachten, so finden wir dort den Ports PD2 und PD3 die Bezeichnungen **INT0** und **INT1**. Diese Pins sind die einzigen direkt geschalteten Interrupt-Ports. Eine Änderung der Spannung an diesen Pins triggert die Interrupt-Routinen, die wir im Folgenden implementieren werden. Neben diesen beiden Interrupt-Pins gibt es eine Zahl weiterer, die über ein Multiplexing-Verfahren adressiert werden können. Dies war jedoch nicht Teil der Vorlesung und wird daher an dieser Stelle ausgespart.

**Interrupttabelle** Für jeden möglichen Interrupt muss zentral gespeichert sein, was nach Eintritt des selbigen abläuft. Dazu steht am Anfang des Flash-Speichers die **Interrupttabelle**. So befindet sich beispielsweise für den Interrupt INT0 ein Eintrag in der Tabelle, der einen Sprungbefehl der Form `JMP 1000` oder `RETI` beinhaltet. Letzteres steht für „Return from Interrupt“ und referenziert die Rücksprungadresse zurück in das Hauptprogramm.

**ISR (Interrupt Service Routine)** Wir implementieren in unserem Programm sogenannte ISRs, in denen der Code steht, den wir im Falle eines Interrupts an einem spezifizierten Pin ausführen.

```
ISR (INT0_vect) {  
    // Programmzeilen  
}
```

Der Code in den ISRs sollte so klein wie möglich sein, da die Abarbeitung per se kritisch ist und keine Interrupts ineinander ausgelöst werden sollten. Daraus folgt auch, dass keine teuren Operationen in ISRs zu erfolgen haben.

**Konfiguration** Um Interrupts im Programm nutzen zu können und auf Eingaben zu reagieren, müssen wir – auf gewohnte Art und Weise – wieder bestimmte Registereinträge ändern.

- **EICRA** (External Interrupt Control Register A) – Die Bits in diesem Register spezifizieren genauer, welche Änderungen der Spannung am Interrupt-Pin die ISR auslösen. Die genauen Einstellungen hierfür sind den Datenblättern der Microcontroller zu entnehmen. Möglich ist beispielsweise, dass nur bei steigenden bzw. fallenden Flanken reagiert wird oder generell bei jedem Pegelwechsel. Dabei spezifizieren

7	6	5	4	3	2	1	0
–	–	–	–	ISC11	ISC10	ISC01	ISC00

Tabelle 3: EICRA

ISC11 und ISC10 zusammen INT1 und ISC01 und ISC00 INT0.

- **EIMSK** (External Interrupt Mask Register) – In diesem Register werden die Interrupts „scharf geschaltet“. Nur wenn das Bit für INT0 hier auf 1 gesetzt wurde, reagiert das Programm tatsächlich auf einen Interrupt an Pin 1.

7	6	5	4	3	2	1	0
–	–	–	–	–	–	INT1	INT0

Tabelle 4: EIMSK

## 6. Programmbeispiel

### 6.1. Lichtschalter

Wir wollen Interrupts nutzen, um die Onboard-LED des Entwicklungsboards über zwei Schalter ein- und auszuschalten.

**Schaltung** Wir benötigen erneut zwei Schalter, die wir auf einem Entwicklungsboard platzieren. Jeweils einen Kontakt der Schalter verbinden wir mit GND und einen der anderen mit INT0/INT1, also PD2 und PD3.

#### Programm

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

ISR (INT0_vect) {                  // ISR fuer INT0
    PORTB = 0x00;                 // Schaltet die Ports an B aus
}

ISR (INT1_vect) {                  // ISR fuer INT1
    PORTB = 0xff;                 // Schaltet die Ports an B ein
}
```

```

int main(void)
{
    DDRB = 0xFF;           // Alle B-Ports werden Ausgaenge
    PORTB = 0xFF;          // Schaltet alle B-Ports ein

    DDRD &= ~(1 << DDD2);  // D2 wird Eingang
    PORTD |= (1 << PORTD2); // An D2 wird Spannung angelegt
    DDRD &= ~(1 << DDD3);  // D3 wird Eingang
    PORTD |= (1 << PORTD3); // An D3 wird Spannung angelegt

    EICRA |= (1 << ISC00);  // INT0 triggert bei Pegelwechsel
    EICRA |= (1 << ISC11);  // INT1 triggert bei Pegelwechsel
    EIMSK |= (1 << INT0);   // INT0 wird scharf
    EIMSK |= (1 << INT1);   // INT1 wird scharf

    sei();                  // set enable interrupt

    while (1)
    {
    }
}

```

## Aufgaben

1. Erläutern Sie, inwiefern der Einsatz von Interrupts gegenüber Polling vorteilsbehaftet ist.
2. Nennen Sie Anforderungen an ISRs.
3. Bauen Sie eine Schaltung mit zwei Tastern an INT0 und INT1 und einem Summer an einem Port aus B nach Wahl auf. Entwickeln Sie ein Programm, dass die Frequenz des Summers bei Tastendruck verdoppelt (starten Sie mit einer niedrigen Frequenz) und dem Summer sofort die Spannung entzieht, sollte der andere Taster gedrückt werden (ein Not-Aus-Schalter). Verwenden Sie dafür Interrupts.

# Teil III.

## Timer / Counter

### 7. Timer und Counter

Das nächste Konstrukt, dem wir uns widmen wollen, sind Timer. Im Blockschaltbild haben wir bereits die zwei 8-Bit- und den einen 16-Bit-Timer entdeckt und nun wollen wir mit diesen arbeiten und die Theorie dahinter verstehen.

**Allgemeine Funktionsweise** Ein Timer ermöglicht es uns, Aussagen zur Zeit, insbesondere der Laufzeit, zu treffen und in Abhängigkeit dazu Aktionen zu realisieren. Allgemein orientiert sich das Zählen immer am Prozessortakt. Über die verschiedenen Modi, die wir später noch genauer kennenlernen werden, lässt sich jedoch z. B. festlegen, dass nur jeder achte Prozessortakt gezählt werden soll. Läuft der Zähler über, beginnt er anschließend wieder bei 0.

**Normal- und CTC-Mode** In welchem Modus ein Timer operiert, wird in Registern festgehalten. Für jeden Timer gibt es drei **Timer/Counter Control Register (TCCRnA, TCCRnB, TCCRnC)**. Betrachten wir nun die verschiedenen Modi (mittlere Tabelle).

	7 bit	6 bit	5 bit	4 bit	3 bit	2 bit	1 bit	0 bit
<b>TCCR0A</b>	COM0A1	COM0A0	COM0B1	COM0B0	-	-	WGM01	WGM00

*Timer/Counter Control Register 0 A*

	7 bit	6 bit	5 bit	4 bit	3 bit	2 bit	1 bit	0 bit
<b>TCCR0B</b>	FOC0A	FOC0B	-	-	WGM02	CS02	CS01	CS00

*Timer/Counter Control Register 0 B*

MODE	WGM02	WGM01	WGM00	DESCRIPTION	TOP
0	0	0	0	Normal	0xFF
1	0	0	1	PWM, Phase Corrected	0xFF
2	0	1	0	CTC	OCR0A
3	0	1	1	Fast PWM	0xFF
4	1	0	0	Reserved	-
5	1	0	1	Fast PWM, Phase Corrected	OCR0A
6	1	1	0	Reserved	-
7	1	1	1	Fast PWM	OCR0A

*Waveform Generator Mode bits*

CS02	CS01	CS00	DESCRIPTION
0	0	0	Timer/Counter0 Disabled
0	0	1	No Prescaling
0	1	0	Clock / 8
0	1	1	Clock / 64
1	0	0	Clock / 256
1	0	1	Clock / 1024
1	1	0	External clock source on T0 pin, Clock on Falling edge
1	1	1	External clock source on T0 pin, Clock on rising edge

*CS bits*

Abbildung 4: Registerübersicht

- **Normal-Mode** – In diesem Modus zählt der Timer die Prozessortakte bis 0xFF, also 255. Anschließend wird er zurückgesetzt und beginnt wieder bei 0. Dieser Modus wird selten eingesetzt.
- **CTC-Mode (Clear Timer on Compare Match)** – Ist man hingegen daran interessiert, weniger weit als 255 zu zählen, lässt sich in diesem Modus eine Zahl definieren

(OCR0A), bis der gezählt wird und ab der der Counter wieder zurückgesetzt wird. Dieser Modus ist verbreiteter als der Normalmode.

Zusätzlich gibt es für beide Modi die Möglichkeit, einen Prescaler einzusetzen. Jener, ebenfalls in Registern festgelegte, Wert teilt die Frequenz des Prozessortaktes und bewirkt folglich, dass der Timer gewisse Takte ignoriert und entsprechend länger zählt. Dafür setzen wir der unteren Tabelle entsprechend die CSn-Bits.

Nun bleibt noch die Frage, wie wir auf die Ergebnisse/Werte des Timers reagieren. Für Timer im Normal- oder CTC-Mode verwenden wir hierfür Interrupts und deklarieren entsprechende ISR-Routinen. Dafür gehen wir vor wie im vorherigen Kapitel und setzen nun statt des EIMSK-Register das **TIMSKn**-Register.

```
ISR (TIMER0_COMPA_vect) {  
    // code  
}
```

## 8. PWM (Pulsweiten-Modulation)

Weitere, signifikant andere Modi für Timer/Counter sind die der Pulsweiten-Modulation. Häufiges Einsatzfeld hierfür ist das Dimmen von LEDs.<sup>1</sup>

**Funktionsweise von PWM im Allgemeinen** Stellen wir uns eine LED vor. Schließen wir sie ohne Timer/Counter an die Spannungsversorgung unseres Arduinos an, leuchtet sie konstant in der vorgesehenen und über den Vorwiderstand gesteuerten Helligkeit. Wir können nun einen Schalter einsetzen, der die LED alle halbe Sekunde ein- und wieder ausschaltet. Dann blinkt sie. Wenn wir nun nicht mehr von Sekunden reden, sondern sehr viel schneller ein- und wieder ausschalten, wirkt die LED dunkler und unser menschliches Auge erkennt das Schalten nicht mehr. Das nutzen wir aus. Wenn wir eine LED also nun dimmen wollen, beeinflussen wir folglich, wie groß die Intervalle zwischen Ein- und Ausschalten sind. Diese Technik ist besagte Pulsweitenmodulation. In der obigen Grafik

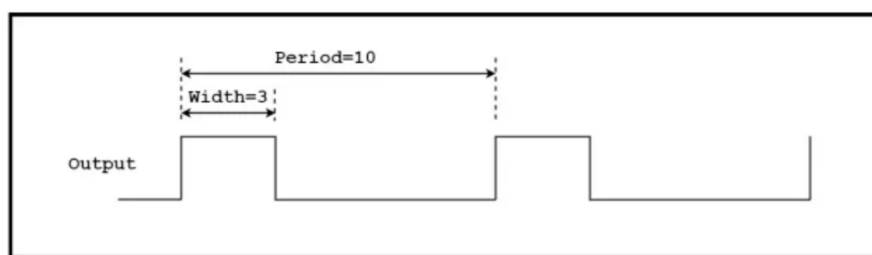


Abbildung 5: Darstellung von Fast PWM

wird die LED mit einer Pulsweite von 3 in einer Periode von 10 mit Spannung versorgt. Das entspricht einem **Duty-Cycle** von 30 %. Beträgt die Breite in diesem Beispiel 5, ergäbe sich ein Duty-Cycle von 50 %. Die Angabe erfolgt üblicherweise in Prozent.

---

<sup>1</sup>Basierend auf <https://www.allaboutcircuits.com/technical-articles/introduction-to-microcontroller-timers-pwm-timers/>



**Timer im PWM-Modus** Nun sind wir aber selbstredend noch daran interessiert, zu erfahren, wie denn nun Timer auf dieser Technik aufbauen. Für PWM gibt es verschiedene Modi (siehe Tabelle), in denen auch wieder festgelegt ist, ob bis 255 oder dem OCR0-Wert (Top-Wert) gezählt wird. Dieser Wert stellt die **Periode** dar. Im einfachsten Fall, dem Modus Fast PWM, funktioniert die Technik so: Zu dem Zeitpunkt, wo der Timer den Top-Wert erreicht, werden die Ausgabe-Pins der PWM-Timer unter Spannung gesetzt. Die Spannungsversorgung endet, wenn der Wert für die Pulsweite, der durch die Registerwerte für OCRA (im Fall Top=255) und OCRB (im Fall Top=OCRA) definiert ist, erreicht wird. Dann zählt der Timer bis zum Ende der Periode weiter.

Wie angedeutet, existieren zwei PWM-Modi mit leichten Unterschieden:

- **Fast PWM** – Der Timer zählt von 0 bis z. B. 255, setzt nun die Pins unter Spannung, bis Compare Match eingetreten ist (OCRA-Wert). Nun wird weitergezählt, bis erneut 255 erreicht ist.
- **Phase-corrected PWM** – Der Timer zählt von 0 bis z. B. 255 und entzieht den Pins auf dem „Hinweg“ die Spannung, wenn der entsprechende Wert erreicht wurde. Ist TOP (also 255) erreicht, wird nicht wieder bei 0 begonnen, sondern es wird jetzt rückwärts gezählt und beim Vergleichswert werden die Pins wieder mit Spannung versorgt. So ergibt sich eine symmetrischere Spannungskurve.

**Signalausgabe** In bekanntem Schaltbild 3.1 können wir eine Reihe an Pins entnehmen, die mit dem Zusatz **PWM** gekennzeichnet sind. Diese Pins sind in der Lage, ein PWM-Signal auszugeben. Für den ersten Timer, den wir nutzen, ist dies beispielsweise an Pin D6 der Fall.

## 9. Programmbeispiel

### 9.1. LED dimmen über PWM-Timer

**Schaltung** Wir schließen die Anode einer LED mit Vorwiderstand an den Port D6 und die Kathode an GND an. Der passende Vorwiderstand ist abhängig von der Spezifikation der LED.

#### Programm

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>

int main(void)
{
    DDRD |= (1 << DDD6);
    PORTD |= (1 << PORTD6);

    OCR0A = 0;

    // Counter is now non-inverting
    TCCR0A |= (1 << COM0A1);
```

```

// Set counter in PWM mode (fast PWM)
TCCR0A |= (1 << WGM02) | (1 << WGM01) | (1 << WGM00);

// Set prescaler = 8
TCCR0B |= (1 << CS01);

while (1)
{
    while (1) {
        // Increment OCR0A and wait 25 ms
        OCR0A += 1;
        _delay_ms(25);
    }
}

```

## Aufgaben

1. Erläutern Sie die Möglichkeiten, auf Timer-Ereignisse programmatisch zu reagieren. Gehen Sie dabei auf die unterschiedlichen Modi ein.
2. Berechnen Sie die OCRA-Werte für folgende Duty-Cycles und TOP-Werte und erläutern Sie die Bedeutung der einzelnen Werte.
  - 45 % Duty-Cycle bei einem TOP-Wert von 0xFF
  - 70 % Duty-Cycle bei einem TOP-Wert von 128
  - 10 % Duty-Cycle bei einem TOP-Wert von 64
3. Erweitern Sie das dargestellte Programm so, dass nun zwei LEDs invers zueinander hoch- bzw. heruntergedimmt werden. Die eine LED soll folglich genau dann am stärksten leuchten, wenn die andere nicht leuchtet. Begründen Sie Ihre Ansätze.