Mikrocontrollertechnik

MODUL C

Copyright ©

Inhaltsverzeichnis MODUL C

C0 Wiederholung

Die Grundlagen der Assembler-Programmierung aus dem Modul B sollen noch einmal kurz in Erinnerung gerufen werden.

Kurze Zusammenfassung Modul B

- **Mechanische Schalter** an Eingängen ohne hardwaremäßige Entprellung müssen softwaremäßig mit Hilfe einer Zeitschleife **entprellt** werden (Pull-Up-Widerstände nicht vergessen; mit PIN und nicht mit PORT einlesen!).
- Bei der Benutzung von Unterprogrammen und Interrupt-Behandlungsroutinen (ISR) muss der **Stapel initialisiert** werden! Am Besten die 4 Zeilen zur Initialisierung im Template nie löschen.
- Auf die Arbeitsregister kann üblicherweise im gesamten Programm zugegriffen werden. Man spricht dann von **globalen Variablen**. Variablen die aber nur innerhalb eines Unterprogramms benötigt werden, sollten kein wertvolles Arbeitsregister blockieren. Innerhalb von Unterprogrammen und Interruptroutinen soll man mit **lokalen Variablen** arbeiten. Um Arbeitsregister für lokale Variablen zu befreien wird ihr Inhalt auf dem Stapel mit dem Befehl **`push`** zwischengespeichert. Vor dem Verlassen des Unterprogramms bzw. der Routine wird der Inhalt dann wieder ins Arbeitsregister zurückgespeichert (**`pop`**).
- Globale Variablen, die zur Übergabe von Daten an Unterprogramme dienen werden **Parameter** genannt. Am besten benutzt man immer die gleichen Arbeitsregister für die Parameterübergabe (in diesem Kurs das Doppelregister W (`r24`:`r25` bzw. `WL`:`WH`)). Diese Arbeitsregister dürfen dann nicht auf den Stapel gerettet werden. Bei mehreren Parametern (z.B. Tabellen) sollte auf Speicherzellen im SRAM zurückgegriffen werden.
- Interrupts müssen einzeln für die lokale Baugruppe und global (**`sei`**) freigeschaltet werden. Zusätzlich muss der Interruptvektor mit einem Sprungbefehl zur ISR initialisiert werden.
- In einer Interruptroutine müssen zwingend alle Register die in der ISR verwendet werden, sowie auch das Statusregister **`SREG`**, auf dem Stapel gesichert (**`push`**) und von diesem wiederhergestellt (**`pop`**) werden. Die ISR schließt mit einem **`reti`** ab!
- Es existieren beim ATmega32A drei externe Interrupts (**`INT0`**, **`INT1`**, **`INT2`**) an **`PD2`**, **`PD3`** und **`PB2`**. Im SF-Register **`MCUCR`** (bzw. **`MCUCSR`** für **`INT2`**) wird festgelegt, ob der Interrupt auf eine positive Flanke, eine negative Flanke oder eine beliebige Flanke bzw. einen Pegel (nur **`INT0`** und **`INT1`**) reagiert. Erstaunlicherweise reicht es das **`MCUCR`**-Register (bzw. **`MCUCSR`**) zu initialisieren, damit die Interruptflags bei auftretender Flanke bereits gesetzt werden, sogar wenn der externe Interrupt noch gar nicht freigeschaltet ist!
- Wartet der Controller in einer Warteschleife auf ein Ereignis, so bezeichnet man das als **Polling**. Polling hat den Vorteil einer sequentiellen Programmierung, ist aber nicht sehr effektiv, da der Controller blockiert wird. Wenn möglich sollten deshalb **Interrupts** eingesetzt werden. Mit Interrupts ist Multitasking möglich, da einzelne Baugruppen unabhängig von Prozessor arbeiten können und beim Auftreten eines wichtigen Ereignisses dieses durch eine Unterbrechung mitteilen. Die Programmierung mittels Interrupts erhöht allerdings die Komplexität der Programme.

⌂ **C000** Mit Hilfe der Interrupt-Eingänge INT0 und INT1 soll ein Lauflicht beeinflusst werden. Die LEDs für das Lauflicht sind mit Port D verbunden.
Eine positive Flanke an INT0 dreht die Richtung des Lauflichtes um, während an INT1 eine positive Flanke die Geschwindigkeit des Lauflichtes von schnell nach langsam oder umgedreht wechselt.

a) Zeichne das Flussdiagramm zum Unterprogramm.
b) Schreibe ein kommentiertes Assemblerprogramm.
Gib dem Programm den Namen "**C000_INT_driven_running_light.asm**".
c) Verbinde die beiden Interrupt-Eingänge miteinander und löse beide Interrupts *gleichzeitig* aus. Erkläre Deine Beobachtungen!

⌂ **C001** Ändere die vorige Aufgabe um, so dass man einen Schrittmotor ansteuern kann. `INT0` ist wieder für den Richtungswechsel und `INT1` für eine Änderung der Geschwindigkeit verantwortlich.
Gib dem Programm den Namen "**C001_INT_stepper_motor.asm**".

C1 Timer

Einführung

Die drei Timer des ATmega32 bieten sehr viele unterschiedliche Betriebsarten. Im folgenden Kapitel sollen nur einige der Betriebsarten kennen gelernt werden.

Was ist ein Timer?

Ein Timerbaustein oder eine Timereinheit ist ein dualer Aufwärtszähler mit der Schrittweite Eins. Er ist zu jeder Zeit lese- und schreib-bar und kann vielfältig als Zeitgeber (*timer*) oder Zähler (*counter*) eingesetzt werden.
Als **Timer** wird der Systemtakt zum Zählen verwendet. Mittels Vorteiler kann dieser, falls nötig verringert werden.
Als **Counter** erhöhen externe Flanken am Timereingang (z.B. **T0**) den Zählstand.

Der Timer[1] ist eigenständig und läuft immer unabhängig vom Controllerkern und Programm. Nur mit ihr kann man zum Beispiel zeitliche Abstände präzise bestimmen oder mittels Interrupt in genauen Zeitabständen Ereignisse auslösen.

Im ATmega32 bzw. ATmega8 sind drei unabhängige Timereinheiten mit teils unterschiedlichen Eigenschaften verfügbar.

Was kann man mit Timern tun?

Timer kann man einsetzen, um Zeitverzögerungen zu bewirken (anstatt von Zeitschleifen), als Frequenzgenerator oder -zähler (-messer), als Ereigniszähler für externe Signale, um Zeitabstände externer Signale zu messen, als Zeitgeber (Uhr), oder zur Pulsweitenmodulation zum Beispiel zum Ansteuern von Motoren,

Der ATmega32 enthält drei Universal-Timer mit folgenden Eigenschaften:

Timer/Counter 0: 8 Bit (0-255)

- Aufwärtszähler mit externem oder internem Takt mit Vorteiler (10 Bit)
- Überlauf-Interrupt (**T0V0**) und Vergleichswertinterrupt (**OCF0**)
- Frequenzgenerator
- Externer Ereigniszähler
- Pulsweitenmodulation PWM (*fast* PWM und *phase correct* PWM)
- Vergleichswertauswertung mit der Möglichkeit den Timer zurückzusetzen (*clear timer on compare match*, CTC-Modus)

Timer/Counter 1: 16 Bit (0-65535)
zusätzlich zu Timer 0:

- Zwei unabhängige Vergleichswertauswertungen

1 Der Einfachheit halber wird im Text nicht immer zwischen Timer und Counter unterschieden. Die Bezeichnung "Timer" wird allgemein für die Timereinheit verwendet.

- Input-Capture-Einheit (mit Rauschunterdrückung). Sie ermöglicht die Zeitmessung externer Signale (Stoppuhr)
- 4 Interruptquellen (Überlauf (**T0V1**), 2 mal Vergleichswert (**OCF1A** und **OCF1B**) und Input-Capture (**ICF1**))

Timer/Counter 2: 8 Bit (0-255)
ähnlich Timer 0, allerdings ohne externen Ereigniszähler!
zusätzlich zu Timer 0:
- Timer 2 kann als Echtzeituhr (*real time clock*, RTC) verwendet werden.

Grundsätzliche Funktionsweise des Timers

Ein Timer (Zeitgeber) oder Counter (Zähler) wird gesetzt (initialisiert) und gestartet. Danach läuft[2] (zählt) er unabhängig und unbeeinflusst vom Controllerkern und der anderen Peripherie bis man ihn stoppt. Hat der Timer seinen Höchstwert (255 bzw. 65535 für 8 und 16 Bit) erreicht, so beginnt er wieder bei null. Bei diesem Überlauf, oder bei einem anderen vorgegebenem Wert wird ein Flag gesetzt, das durch Polling abgefragt werden kann, oder es wird, was meist sinnvoller ist, ein Interrupt ausgelöst.

Man kann den Timer auch beim Erreichen eines Überlaufs oder eines Vergleichswertes ein Signal an einem Pin ausgeben lassen oder zum Beispiel die Zählrichtung umkehren.

Als Takt kann der AVR-Takt verwendet werden. Ein Vorteiler ermöglicht die Änderung der Taktfrequenz. Es kann aber auch ein externer Takt verwendet werden.

Zum Kennenlernen wird in den folgenden Kapiteln der 8-Bit Timer 0 verwendet. Alle Beispiele lassen sich auch auf die beiden anderen Timer übertragen, indem man die Bezeichnung der Register anpasst.

Der Timer als Zeitgeber

Überlauf mit dem Timer 0

Nach der Aktivierung des Timers beginnt dieser gleich mit dem Zählen. Bei jedem Überlauf des Timers wird automatisch das Flag **T0V0** im SF-Register **TIFR** gesetzt. Bei Timer 0 also jedes Mal wenn dieser von 0 (Default-Startwert nach Reset) bis 255 gezählt hat (nach 256 Zählschritten).

Der Timer 0 soll nun bei jedem Überlauf periodisch einen Interrupt auslösen (Timer 0 Overflow Interrupt).

Initialisierung des Timer 0 für den Überlauf-Interrupt

Zuerst muss geklärt werden mit welcher Taktrate gezählt werden soll. Diese Einstellung erfolgt mit den Bits 0 bis 2 (**CS00-CS02**) des Timer/Counter Controll Registers **TCCR0**. Folgende Tabelle bietet einen Überblick:

2 Der Timer wird automatisch inkrementiert oder dekrementiert.

Bit 2	Bit 1	Bit 0	Taktquelle
CS02	CS01	CS00	
0	0	0	Timer Stopp (verbraucht keinen Strom)
0	0	1	Systemtakt (:1)
0	1	0	Systemtakt / 8
0	1	1	Systemtakt / 64
1	0	0	Systemtakt / 256
1	0	1	Systemtakt / 1024
1	1	0	externer Takt: fallende Flanke an T0 (PB0)
1	1	1	externer Takt: steigende Flanke an T0 (PB0)

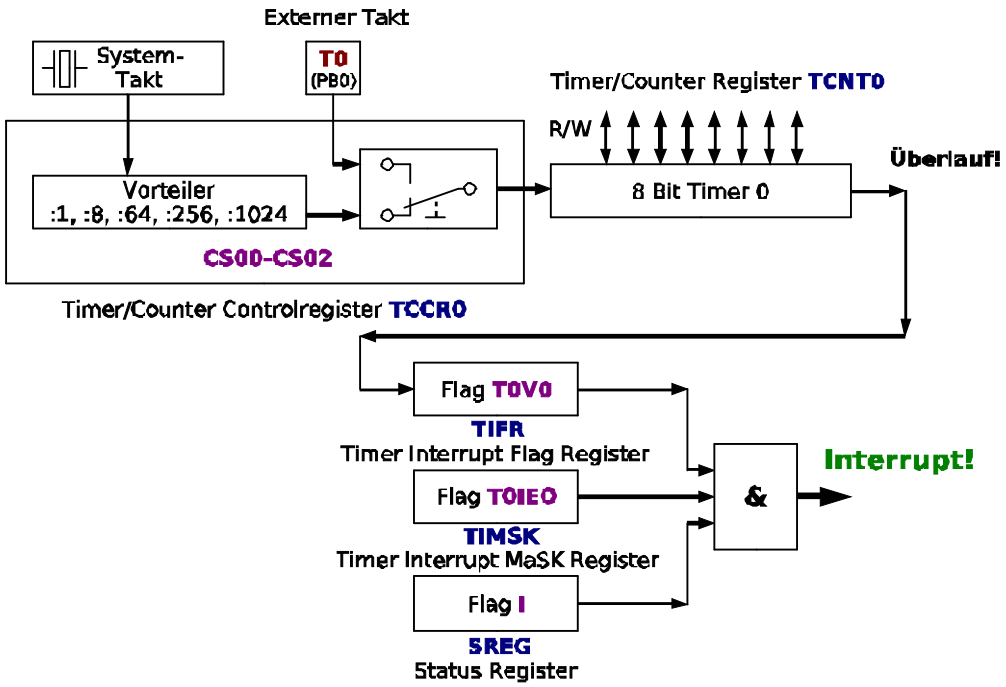
Bemerkung: Durch Setzen der drei Bits **CS00-CS02** auf einen Wert größer als Null wird der Timerbaustein eingeschaltet. Er beginnt nach dieser Initialisierung gleich mit dem Zählen ab Null, wenn kein Wert voreingestellt wurde.

Die Frequenz errechnet sich mit:

$$f = \frac{\text{Systemtakt}}{\text{Vorteiler} \cdot \text{Zählschritte}}$$

Beispiel: Als Vorteiler wurde 1024 gewählt bei einer Quarzfrequenz von 16 MHz. Damit ergibt sich eine Zählfrequenz des Timers von 16 MHz / 1024 = 15625 Hz. Da jedes Interrupt nach 256 Zählschritten erfolgt, wird der Interrupt also mit einer Frequenz von 15625 / 256 = 61,03515625 Hz aufgerufen.

Hinweis: Die obige Tabelle gilt ähnlich für Timer 1. Für Timer 2 ist sie allerdings nicht gültig da dieser keine externen Signale verarbeiten kann. Dafür kann aber zusätzlich ein Vorteiler von 32 und 128 eingestellt werden (siehe Datenblatt).



Damit das Interrupt ausgelöst wird müssen drei Bedingungen erfüllt sein:

1. Das Flag **T0V0** im Timer Interrupt Flag Register **TIFR** muss Eins sein (**T0V0** = 1). Dies wird **automatisch gesetzt** wenn ein Überlauf erfolgt!
2. Der **T0**-Interrupt muss erlaubt sein (**TOIE0** = 1 im Register **TIMSK**).
3. Interrupts müssen global freigegeben sein (**I** in **SREG** = 1 mit "**sei**")

Um den Timer als Zeitgeber im Interrupt-Betrieb zu benutzen, müssen außer der üblichen Initialisierung also noch folgende Schritte erledigt werden:

- Sprungadresse für den Interrupt organisieren (aus der Definitionsdatei: **OVF0addr = 0x0016**) und der Interruptroutine einen Namen zuweisen (z. B. **INTT0**).

```
;-----------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
;-----------------------------------------------------------------------------
       ;Vektortabelle (im Flash-Speicher)
.ORG    OVF0addr                ;Vektor Nr 12 (Adresse 0x16) fuer
        rjmp INTT0              ;INTT0 (Timer0-Overflow Interrupt)

;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
```

- Initialisierung des Stapels nicht vergessen!
- Die Timereinheit mit den Flags **CS00-CS02** im SF-Register **TCCR0** einschalten. Gleichzeitig wird der Vorteiler festgelegt. Da die oberen fünf Bit dieses Registers hier Null sind (normaler Modus, **OC0** normales Port Pin), muss hier nicht zwingend eine Maskierung eingesetzt werden:

```
        ;Timer einschalten und Vorteiler festlegen
        ldi     Tmp1,0x01       ;TCCR0 = 0b00000 001 (Systemtakt/1)
        out     TCCR0,Tmp1      ;
```

- **T0IE0** im SF-Register **TIMSK** mittels Maskierung setzen:

```
        ;Timer Interrupt erlauben
        in      Tmp1,TIMSK      ;
        ori     Tmp1,0x01       ;Mit einer ODER-Maskierung (00000001b)
                                ;Bit 0 (TOIE0) im TIMSK-Register auf eins setzen
        out     TIMSK,Tmp1      ;Timer 0-Overflow-Interrupt ein
```

- Interrupts global freigeben (sind bei Reset des Controllers per *default* gesperrt):

```
        ;Interrupts freigeben
        sei                     ;Interrupts global freigeben (I-Bit im SREG)
```

- Interruptroutine schreiben. Innerhalb der Routine müssen die Flags (**SREG**) und alle!, innerhalb der Routine benutzten Register, auf den Stapel gerettet werden.

⌂ **C100** Es soll am Pin **PA0** mittels Timer 0-Interrupt ein Rechtecksignal durch Toggeln des Ausgangs erzeugt werden. Es wird ein Frequenzzähler oder Oszilloskop an das Pin angeschlossen. Um zu zeigen, dass die CPU (das Hauptprogramm) dadurch nicht belastet wird, soll das Hauptprogramm im Sekundentakt von null an aufwärts zählen und den Zählerstand am Sieben-Segment-Display ausgeben. Der Code für das Hauptprogramm ist vorgegeben (Unterprogrammbibliotheken für Display und Zeitschleifen müssen eingebunden werden):

```
        rcall   D7SINI
;-------------------------------------------------------------------------------
;       Hauptprogramm
;-------------------------------------------------------------------------------
MAIN:   rcall   D7SHEX          ;Darstellungsunterprogramm aufrufen
        adiw    WL,1            ;Zaehler inkrementieren
        rjmp    MAIN            ;Weiter
```

a) Errechne die zu erwartende Frequenz, wenn bei der Initialisierung des Timers der Vorteiler auf 256 eingestellt wird (Quarzfrequenz 16 MHz).
b) Zeichne das Flussdiagramm und erstelle das Programm.
Nenne das Programm "**C100_8bit_timer0_ISR_OVF1.asm**"
c) Überprüfe die Frequenz mittels Zähler oder Oszilloskop.

Überlauf mit dem Timer 0 mit Voreinstellung

Das Timerzählregister **TCNT0** kann nicht nur ausgelesen werden, sondern auch beschrieben werden. Dies ermöglicht es die Zählschritte bis zum Überlauf zu verringern. Wird das Register zum Beispiel mit dem Startwert 200 voreingestellt, so zählt der Timer von 200 bis 255, also nur 56 Schritte.

Soll jetzt, um zum Beispiel eine höhere Frequenz zu erzeugen, der Timer nach jedem Überlauf mit einem festen Startwert versehen werden, so muss das gleich am Anfang der Interruptroutine (vor den Push-Befehlen!) geschehen um die Genauigkeit des Timers nicht zu gefährden. Am einfachsten wird der Startwert in einer globalen Variablen bei der Initialisierung festgelegt.

⌂ **C101** Ändere das Programm **C100** so um, dass der Timer 0 immer mit dem Startwert 150 beginnt.
Nenne das neue Programm "**C101_8bit_timer0_ISR_OVF2.asm**".
a) Ermittle rechnerisch und per Zähler (Oszilloskop) die erzeugte Frequenz.
b) Errechne den Startwert, der benötigt wird um eine Frequenz von 440 Hz (Kammereton A (LA)) zu erzeugen. Überprüfe das Ergebnis.

⌂ **C102** Ändere das Programm **C101** so um, dass im Hauptprogramm im Sekundenrythmus der Startwert des Timers ab Null jeweils um 5 erhöht wird. Nenne das neue Programm "**C102_8bit_timer0_ISR_OVF3.asm**".

Interrupt durch Vergleich mit dem Timer 0 (CTC)

Neben dem Zählregister existiert ein Vergleichregister **OCR0**, das eine einfachere Einstellung der Zählschritte erlaubt. Statt des Überlauf-Interrupts wird der Vergleich-Interrupt im **TIMSK** Register eingeschaltet und natürlich auch die entsprechende Sprungadresse initialisiert (**.ORG OCR0addr**).

Im **TCCR0**-Register wird wie gehabt der Teiler festgelegt. Es wird der **CTC-Modus** (*clear timer on compare match*) eingeschaltet mit den beiden *Waveform Generation Mode* Bits **WGM0=10**. Die beiden *Compare Match Output Mode* Bits bleiben auf null **COM0=00. OC0 (PB3)** bleibt abgeschaltet (normales Port-Pin; siehe weiter unten).

Im Programm wird das Vergleichsregister mit den Zählschritten initialisiert um eine bestimmte Frequenz der Interruptroutine zu erhalten.

⌂ **C103** Programmiere den Timer 0 im Interrupt-CTC-Modus, so dass eine Frequenz von 440 Hz (Kammereton A (LA)) am Pin PC7 erzeugt wird. Überprüfe das Ergebnis. Nenne das neue Programm "**C103_8bit_timer0_ISR_CTC.asm**".

Vergleichsmodus (CTC) ohne Interrupt (Timer 0)

Der Timer 0 hat auch eine Funktion um Frequenzen ohne Interrupt zu erzeugen (Frequenzgenerator)! Ein fest zugeordnetes Pin **OC0** (**PB3**) wird bei entsprechender Initialisierung getoggelt, sobald ein im Output Compare Register **OCR0** eingetragener Wert erreicht wurde (siehe auch PWM).

Dies beiden *Compare Match Output Mode* Bit im **TCCR0**-SF-Register entscheiden über den Zustand des Ausgangspin.

COM01 COM00 2^1 2^0	Verhalten im normalen Modus:
00	**OC0 abgeschaltet (normales Port-Pin)**
01	**Toggele OC0 bei Vergleichsübereinstimmung**
10	**Lösche OC0 bei Vergleichsübereinstimmung**
11	**Setze OC0 bei Vergleichsübereinstimmung**

⌂ **C104** Programmiere den Timer 0 im CTC-Modus ohne Interrupt, so dass er folgendes Lied abspielt. Der zweite Wert in der Tabelle entspricht der Pausendauer (1 = 200ms, 2 = 100ms, 4 = 50ms usw.). Zum Schluss jeder Note wird eine kurze Pause (100µs) eingelegt. 0xFF bedeutet, dass das Lied vorbei ist.
Nenne das neue Programm "**C104_8bit_timer0_CTC.asm**".

```
; Zaehlschritte fuer Timer0 (CTC) mit Vorteiler 64
.EQU     Si2    = 254
```

```
.EQU     Do3    = 239
.EQU     Do3d   = 226
.EQU     Re3    = 213
.EQU     Re3d   = 201
.EQU     Mi3    = 190
.EQU     Fa3    = 179
.EQU     Fa3d   = 169
.EQU     Sol3   = 160
.EQU     Sol3d  = 151
.EQU     La3    = 143
.EQU     La3d   = 135
.EQU     Si3    = 127
.EQU     Do4    = 120
.EQU     Do4d   = 113
.EQU     Re4    = 107
.EQU     Re4d   = 101
.EQU     Mi4    = 95
.EQU     Fa4    = 90
.EQU     Fa4d   = 58
.EQU     Sol4   = 80
.EQU     Sol4d  = 76
.EQU     La4    = 72
.EQU     La4d   = 68
.EQU     Si4    = 64
.EQU     Do5    = 60
.EQU     Do5d   = 57
.EQU     Re5    = 54
.EQU     Re5d   = 51
.EQU     Mi5    = 48
.EQU     Fa5    = 45
.EQU     Fa5d   = 43
.EQU     Sol5   = 40
.EQU     Sol5d  = 38
.EQU     La5    = 36
.EQU     La5d   = 34

; Zeitdauer 1 = ganze Note, 2 = halbe Note, 4 = Viertelnote etc
NOTET1:  .DB     Fa3,4,Sol3d,6,Fa3,8,Fa3,16,La3d,8,Fa3,8,Re3d,8
         .DB     Fa3,4,Do4,6,Fa3,8,Fa3,16,Do4d,8,Do4,8,Sol3d,8
         .DB     Fa3,8,Do4,8,Fa4,8,Fa3,16,Re3d,8,Re3d,16,Do3,8,Sol3,8,Fa3,8
         .DB     Fa3,2,0,2
         .DB     Fa3,4,Sol3d,6,Fa3,8,Fa3,16,La3d,8,Fa3,8,Re3d,8
         .DB     Fa3,4,Do4,6,Fa3,8,Fa3,16,Do4d,8,Do4,8,Sol3d,8
         .DB     Fa3,8,Do4,8,Fa4,8,Fa3,16,Re3d,8,Re3d,16,Do3,8,Sol3,8,Fa3,8
         .DB     Fa3,2,0,2
         .DB     0xFF,0xFF
```

Der Timer als Zähler (Counter)

Über ein reserviertes Pin kann die fallende oder die steigende Flanke eines externen Signals dazu verwendet werden den Zähler zu erhöhen. So ist es möglich den Timer als Zähler externer Ereignisse einzusetzen.

Timer 0 als Zähler (Counter)

Um den Timer als Zähler einzusetzen, reicht es, über die Bits **CS00-CS02** im Timer/Counter Control Register **TCCR0** statt einem internen Takt einen externen Takt auszuwählen. Dabei kann die steigende Flanke oder die fallende Flanke des externen Signals zum Zählen genutzt werden. Es ist sinnvoll, da meist mit Pull-Up Widerständen gearbeitet wird (negative Logik), die fallende Flanke zum Zählen zu verwenden. Das externe Signal muss an Pin **T0** (**PB0**) angelegt werden.

⌂ **C105** Ein nicht entprellter Taster soll als Signalquelle für den Zähler dienen (**T0** als Eingang initialisieren und Pull-Up einschalten!). Der Zählstand (**TCNT0**) soll im Hauptprogramm im Sekundentakt angezeigt werden. Der Zähler ist auf den Wert 246 voreingestellt, und beim Überlauf soll eine LED (**PA0**) umgeschaltet werden.
Nenne das neue Programm "**C105_8bit_counter0_1.asm**".

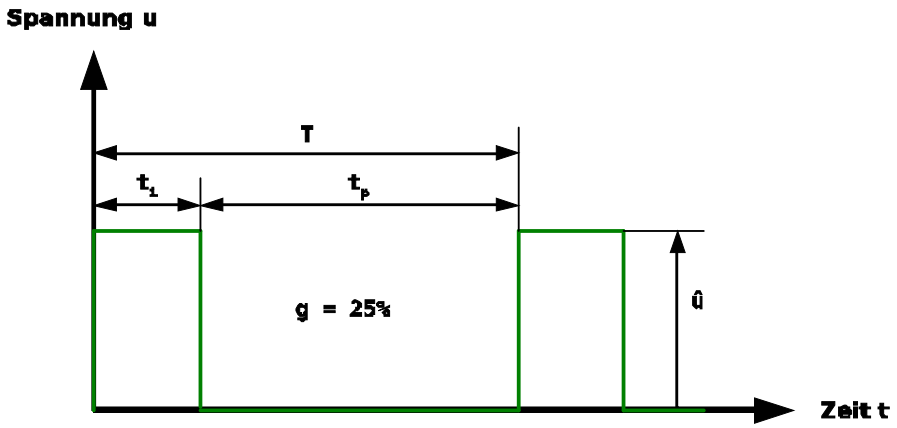
Pulsweitenmodulation mit dem Timer

Die Leistung kann bei vielen elektrischen Geräten durch Veränderung der Spannung gesteuert werden. Es gibt aber auch Geräte wo dies nicht zufriedenstellend funktioniert, da diese Geräte zum Beispiel eine Mindestspannung zum Arbeiten benötigen (LED, Motor). Die Pulsweitenmodulation bietet hier Abhilfe und lässt sich (anders als eine variable Spannung) sehr leicht mit einem Mikrocontroller realisieren.

Bei der Pulsweitenmodulation wird eine Rechteckspannung mit fester Frequenz verwendet, bei der der Tastgrad[3] (*duty cycle*) verändert wird. Die Impulsweite im Verhältnis zur Periodendauer wird meist in Prozent ausgedrückt.

Impulsdauer t_i Pausendauer t_p

Tastgrad $g = \frac{t_i}{T}$ Periodendauer $T = t_i + t_p$



Pulsweitenmodulation mit Timer 0

Die Pulsweitenmodulation lässt mit dem Timer relativ einfach softwaremäßig realisieren. Bei dieser Variante mit Überlauf- und Vergleichswert-Interrupt kann dann jedes beliebige Ausgangspin genutzt werden.

3 Oft findet man auch den Begriff "Tastverhältnis". Dieser Begriff ist allerdings nicht einheitlich definiert. Oft findet man ihn als Kehrwert des Tastgrads, manchmal ist er gleich wie der Tastgrad definiert, oder auch als Verhältnis zwischen Impuls- und Pausendauer.

Noch einfacher geht es allerdings, wenn der Timer alle Aufgaben selber übernimmt. Dieser von ATMEL® genannte "Fast PWM"-Modus benötigt keine Interruptroutinen und schont somit die Ressourcen des Controllers. Als Ausgang muss für Timer 0 der **OC0**-Pin (**PB3**) genutzt werden.

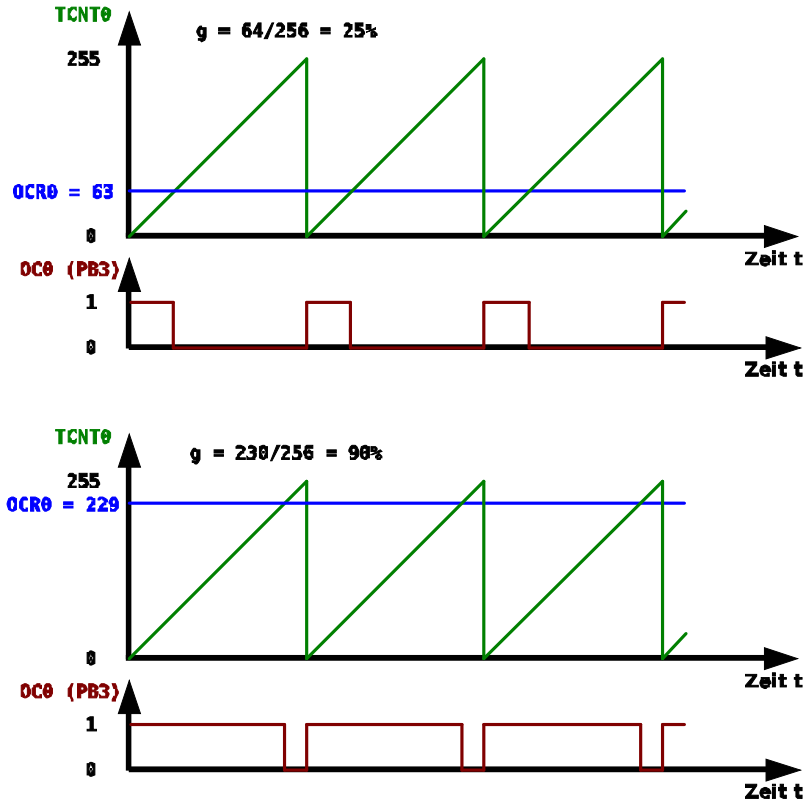
Funktionsweise des "Fast PWM"-Modus:

Der Timer 0 zählt durchgehend von 0 bis 255 und beginnt dann wieder bei 0 (Zählregister **TCNT0**). Ein gesamter Durchlauf entspricht der Periodendauer des PWM-Signals. Die Frequenz wurde durch den Systemtakt und den Vorteiler bestimmt.

$$f = \frac{\text{Systemtakt}}{\text{Vorteiler} \cdot 256}$$

Am Ausgangspin **OC0** liegt High-Pegel an (nicht-invertierender Modus).

In einem zweiten Register, dem sogenannten Vergleichsregister **OCR0**, wird ein Vergleichswert festgelegt. Erreicht der Zählwert diesen Vergleichswert, so kippt der Pegel am Ausgangspin **OC0** auf Low-Pegel. Der Zähler zählt ungerührt weiter, um beim Überlauf das Ausgangspin wieder auf High-Pegel zu setzen. Der Vergleichswert bestimmt den Tastgrad!



Initialisierung des Timer 0 für den "Fast-PWM"-Modus

Um den "Fast PWM"-Modus einzuschalten müssen die Bits **WGM00** und **WGM01** im Kontrollregister **TCCR0** beide auf Eins gesetzt werden. In diesem Modus wird mit **COM01** = 1 und **COM00** = 0 (beide ebenfalls im Kontrollregister **TCCR0**) der nicht invertierende PWM-Modus ausgewählt (invertierender Modus: Modus: **COM01** = 1 und **COM00** = 1). Über die Bits **CS00-CS02** wird über den Vorteiler die Frequenz eingestellt. Das letzte freie Bit (**FOC0**) im Register **TCCR0** muss in diesem Modus auf null gesetzt werden.

Zusätzlich zur Initialisierung des Kontrollregisters **TCCR0** muss nur noch das Pin **OC0** (**PB3**) als Ausgang initialisiert werden und ein Vergleichswert muss ins Vergleichsregister **OCR0** Register geschrieben werden.

Die ganze Initialisierung besteht zum Beispiel aus folgenden Zeilen:

```
;viertes Pin von Port B (PB3) als Ausgang initialisieren
sbi     DDRB,3          ;Ausgang

;Timer einschalten, Vorteiler festlegen und Fast PWM waehlen
ldi     Tmp1,0x6C       ;TCCR0 = 0b01101100 (Fast PWM,nicht inv., 1/256)
out     TCCR0,Tmp1      ;
;Vergleichswert fuer PWM festlegen
ldi     Tmp1,63         ;Vergleichswert festlegen
out     OCR0,Tmp1       ;
```

⌂ **C106** Die Helligkeit einer LED soll im Sekundenrhythmus erhöht werden. Dazu soll im Hauptprogramm einmal pro Sekunde der Vergleichswert ab 0 um 5 erhöht werden. Die Siebensegmentanzeige gibt gleichzeitig den Vergleichswert aus.

a) Schreibe das Programm und nenne es "**C106_8bit_timer0_pwm_1.asm**".
b) Betrachte die Spannung mit einem Oszilloskop. Interessant ist auch die Ausgabe mittels Summer.
c) Welchen Einfluss hat die Frequenz?

⌂ **C107** Mit Hilfe einer PWM soll ein Funktionsgenerator (Rechteck, Sinus, Dreieck und Sägezahn) mit einer festen Frequenz von 440 Hz programmiert werden. Die PWM arbeitet mit der höchsten Frequenz (Vorteiler = 1). Ein Tiefpass am Ausgang (C = 47 nF, R = 1,2 kΩ) siebt den hochfrequenten Teil der PWM aus. Die Wellenformen sollen sich in Tabellen (8 Bit, 256 Werte) befinden(siehe Kapitel DA Wandler).

a) Berechne die Grenzfrequenz des Tiefpasses.
b) Schreibe das Programm und nenne es "**C107_PWM_wavegen.asm**".
c) Betrachte die Spannung mit einem Oszilloskop. Interessant ist auch die Ausgabe mittels Summer.

Bemerkung: Im normalen Modus kann mit Hilfe des Vergleichswertregisters ein zusätzlicher Interrupt bei Übereinstimmung des Zählwertes mit dem Vergleichswert ausgelöst werden (*Output Compare Match Interrupt*). Im sogenannten CTC-Modus (*Clear Timer on Compare Match Mode)* kann der Timer bei Erreichen des Vergleichswertes auch zurückgesetzt werden.

Die SF-Register des Timer 0

Das Timer/Counter Control Register TCCR0

Das **Timer/Counter Control Register 0** befindet sich auf der SRAM-Adresse **0x0053** (SF-Register-Adresse **0x33**) und wird mit der Abkürzung "**TCCR0**" angesprochen (Definitionsdatei). Die Befehle **sbi**, **cbi**, **sbic** und **sbis** können nicht verwendet werden.

TCCR0 = Timer/Counter Control Register 0

Bit	7	6	5	4	3	2	1	0
TCCR0 0x33	**FOC0**	**WGM00**	**COM01**	**COM00**	**WGM01**	**CS02**	**CS01**	**CS00**
Startwert	0	0	0	0	0	0	0	0
Read/Write	W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

WGM0n ***Waveform Generation Mode*** ***WGM01, WGM00***

Mit diesen zwei Bit wird der Operationsmodus des Timer festgelegt:

WGM01 WGM00 2^1 2^0	Modus:
00	Normaler Modus
01	Phasenkorrekter PWM-Modus
10	CTC-Modus (clear timer on compare match)
11	Fast PWM-Modus

COM0n ***Compare Match Output Mode*** ***COM01, COM00***

Mit diesen zwei Bit wird das Verhalten des Ausgangspin **OC0** festgelegt.
Je nach Modus ändert das Verhalten:

COM01 COM00 2^1 2^0	Verhalten im normalen Modus:
00	**OC0** abgeschaltet (normales Port-Pin)
01	Toggele **OC0** bei Vergleichsübereinstimmung
10	Lösche **OC0** bei Vergleichsübereinstimmung
11	Setze **OC0** bei Vergleichsübereinstimmung
COM01 COM00 2^1 2^0	**Verhalten im Fast-PWM Modus:**
00	**OC0** abgeschaltet (normales Port-Pin)
01	Reserviert
10	nicht-invertierender Fast-PWM-Modus
11	invertierender Fast-PWM-Modus

CS0n ***Clock Select Timer 0*** ***CS02, CS01, CS00***

Mit diesen drei Bit wird die Taktquelle für Timer 0 festgelegt.

CS02 CS01 CS00 2^2 2^1 2^0	Taktquelle:
000	Timer Stopp (verbraucht keinen Strom, default nach RESET!)
001	Systemtakt (:1)

010	**Systemtakt / 8**
011	**Systemtakt / 64**
100	**Systemtakt / 256**
101	**Systemtakt / 1024**
110	**externer Takt: fallende Flanke an `T0` (`PB0`)**
111	**externer Takt: steigende Flanke an `T0` (`PB0`)**

Weitere Informationen findet man im Datenblatt.

Das Timer/Counter Interrupt Mask Register TIMSK

Das **Timer/Counter Interrupt Mask Register** befindet sich auf der SRAM-Adresse **`0x0059`** (SF-Register-Adresse **`0x39`**) und wird mit der Abkürzung "**`TIMSK`**" angesprochen (Definitionsdatei). Die Befehle **`sbi`**, **`cbi`**, **`sbic`** und **`sbis`** können nicht verwendet werden.

TIMSK = Timer/Counter Interrupt Mask Register

Bit	7	6	5	4	3	2	1	0
`TIMSK` **`0x39`**	**`OCIE2`**	**`TOIE2`**	**`TICIE1`**	**`OCIE1A`**	**`OCIE1B`**	**`TOIE1`**	**`OCIE0`**	**`TOIE0`**
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

`OCIE0` ***Timer/Counter 0 Output Compare Match Interrupt Enable***

- **0** kein **`OCF0`**-Interrupt erlaubt.
- **1** Das Setzen dieses Bit **ermöglicht das Auslösen eines Interrupts** in dem Moment, wo das **`OCF0`**-Flag (**`TIFR`**) gesetzt wird, also eine Übereinstimmung zwischen dem Zählregister **`TCNT0`** und dem Vergleichsregister **`OCR0`** stattgefunden hat, oder wenn das Flag manuell gesetzt wurde. Interrupts müssen dazu global frei gegeben sein (**`I`**=1 im **`SREG`** mit "**`sei`**").

`TOIE0` ***Timer/Counter 0 Overflow Interrupt Enable***

- **0** kein **`TOV0`**-Interrupt erlaubt.
- **1** Das Setzen dieses Bit **ermöglicht das Auslösen eines Interrupts** in dem Moment, wo das **`TOV0`**-Flag (**`TIFR`**) gesetzt wird, also ein Überlauf auftrat oder wenn das Flag manuell gesetzt wurde. Interrupts müssen dazu global frei gegeben sein (**`I`**=1 im **`SREG`** mit "**`sei`**").

Das Timer/Counter Interrupt Flag Register TIFR

Das **Timer/Counter Interrupt Flag Register** befindet sich auf der SRAM-Adresse **`0x0058`** (SF-Register-Adresse **`0x38`**) und wird mit der Abkürzung "**`TIFR`**" angesprochen (Definitionsdatei). Die Befehle **`sbi`**, **`cbi`**, **`sbic`** und **`sbis`** können nicht verwendet werden.

TIFR = Timer/Counter Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0
`TIFR` **`0x38`**	**`OCF2`**	**`TOV2`**	**`ICF1`**	**`OCF1A`**	**`OCF1B`**	**`TOV1`**	**`OCF0`**	**`TOV0`**
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

OCF0 ***Output Compare Flag 0***

0 Keine Übereinstimmung des Inhalts des Zählregister mit dem Inhalt des Vergleichsregisters.

1 Bei Übereinstimmung (*compare match*) des Wertes im Zählregister **TCNT0** mit dem Wert im Vergleichsregister **OCR0** wird das Flag gesetzt. Das Flag wird automatisch gelöscht, wenn der **OC**-Interrupt ausgeführt wird. Manuell kann das Flag durch das **Schreiben einer Eins!** gelöscht werden.

TOV0 ***Timer/Counter 0 Overflow Flag***

0 Keine Überlauf eingetreten.

1 Tritt ein Überlauf des Zählregisters **TCNT0** auf so wird das Flag gesetzt. Das Flag wird automatisch gelöscht, wenn der **OV**-Interrupt ausgeführt wird. Manuell kann das Flag durch das **Schreiben einer Eins!** gelöscht werden.

Das Timer/Counter Register 0 TCNT0

Das **Timer/Counter Register 0** befindet sich auf der SRAM-Adresse **0x0052** (SF-Register-Adresse **0x32**) und wird mit der Abkürzung "**TCNT0**" angesprochen (Definitionsdatei). Die Befehle **sbi**, **cbi**, **sbic** und **sbis** können nicht verwendet werden. Das Register ist les- und schreibbar.

TCNT0 = Timer/Counter Register 0

Bit	7	6	5	4	3	2	1	0
TCNT0 0x32	**TCNT07**	**TCNT06**	**TCNT05**	**TCNT04**	**TCNT03**	**TCNT02**	**TCNT01**	**TCNT00**
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

Das Output Compare Register 0 OCR0

Das **Output Compare Register 0** befindet sich auf der SRAM-Adresse **0x005C** (SF-Register-Adresse **0x3C**) und wird mit der Abkürzung "**OCR0**" angesprochen (Definitionsdatei). Die Befehle **sbi**, **cbi**, **sbic** und **sbis** können nicht verwendet werden. Das Register ist les- und schreibbar.

OCR0 = Output Compare Register 0

Bit	7	6	5	4	3	2	1	0
OCR0 0x3C	**OCR07**	**OCR06**	**OCR05**	**OCR04**	**OCR03**	**OCR02**	**OCR01**	**OCR00**
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

Weitere Aufgaben

⌂ **C108** Ändere das erste Programm dieses Kapitels so um, dass es mit dem 16-Bit-Timer 1 arbeitet. Es soll die gleiche Frequenz erzeugt werden! Ermittle die für Timer 1 benötigten Register (Datenblatt).
Nenne das neue Programm "**`C108_16bit_timer1_1.asm`**".

(Achtung!! Wenn mit 16 Bit (SF Doppelregister) gearbeitet wird, soll bei beiden **`in`** oder **`out`** Befehlen **`cli`** vor und sei hinten angestellt werden damit Interrupts das Lesen nicht beeinflussen können!!)

C2 Serielle Schnittstelle

Hardware-Schnittstellen

Die Schnittstelle oder Interface ist ein Teil des Computers und dient der Kommunikation (Datenaustausch) mit peripheren Geräten.

Periphere Geräte müssen dabei nicht extern sein (Beispiel: SATA-Festplatte im PC). Eine Schnittstelle erkennt man oft am Steckersystem, mit welchem das periphere Gerät angeschlossen wird. Neben Anpassungsschaltkreisen und der Steckervorrichtung ist für eine Hardware-Schnittstelle meist auch ein steuernder Prozessor (inkl. Software) nötig.

Industrienormen definieren standardisierte Schnittstellen und ermöglichen so, dass Komponenten verschiedener Hersteller miteinander kommunizieren und gegeneinander ausgetauscht werden können. Die Komponenten sind dann zueinander kompatibel.

Unterscheiden kann man Hardware-Schnittstellen nach folgenden Kriterien:

- **Digital - Analog**
- **Seriell - Parallel**
- **Synchron - Asynchron**
- **Mechanische Eigenschaften**

Bemerkung: Durch Veränderungen und Verbesserungen in der Datenverarbeitung sind viele neue Schnittstellen digital, seriell und synchron.

Beispiele von digitalen Hardware-Schnittstellen:

- **Parallel:** PCI, AGP, SCSI, ATA/ATAPI, IEC-625, HP-IB, IEEE-488, IEEE 1284 (früher Centronics)
- **Seriell:** USB, Firewire, EIA-232 (früher RS-232 oder V24), EIA-485 (früher RS-485), I2C, SPI, S-ATA, PCI-Express, HDMI, IrDA, Bluetooth, MIDI, S/P-DIF, CAN-Bus, S0-Bus (ISDN), Ethernet, WLAN.

Bei Mikrocontrollern eingesetzte Hardware-Schnittstellen

EIA-232 Die klassische serielle asynchrone Schnittstelle wird wegen ihrer Einfachheit auch heute noch gerne bei Mikrocontrollern eingesetzt. Die meisten Mikrocontroller besitzen einen internen Baustein (UART oder USART; siehe später) für diese Schnittstelle.

I2C Die von Philipps entwickelte synchrone Master-Slave Schnittstelle, (*Inter-Integrated Circuit;* gesprochen I-Quadrat-C) zur Kommunikation zwischen ICs, benötigt nur drei Leitungen (Halbduplex) und ist in der Unterhaltungselektronik weit verbreitet (viele ansteuerbare Spezial-ICs). Die Taktrate und die überbrückbare Distanz sind recht gering. Bei einigen AVR-Controllern wird die Schnittstelle hardwaremäßig unterstützt und heißt TWI-Schnittstelle (*Two Wire Bus*).

SPI SPI (*Serial Periphal Interface*) ist ein Bus-System (Master-Slave, 5 Leitungen), das von Motorola für die Kommunikation zwischen Mikrocontrollern entwickelt wurde (keine großen Distanzen). Es arbeitet synchron in Vollduplex mit hoher Taktgeschwindigkeit (bis zu mehreren 10 MHz). Die SPI-Schnittstelle wird bei der AVR-Familie zur *In-System*-Programmierung genutzt und hardwaremäßig unterstützt.

USB Die USB-Schnittstelle verdrängt immer mehr die serielle Schnittstelle bei den PCs. Möchte man den Controller an der USB-Schnittstelle betreiben, so ist dies mit entsprechenden USB-EIA-232 Wandlerbausteinen (z. B. FT232BM) möglich, oder mit USB-EIA-232-Adapter-Verbindungskabeln. Auch existieren Mikrocontroller mit integrierter USB-Schnittstelle (z. B. ATmega32U4). Es ist sogar gelungen die komplexe USB-Schnittstelle in ihrer Version 1.1 softwaremäßig auf einem ATmega8A zu implementieren.

Ethernet Auch für die Ethernet-Schnittstelle gibt es Wandlerbausteine nach EIA-232 (z. B. Lantronix X-Port), die allerdings noch recht teuer sind. Softwaremäßige Lösungen (Webserver mit ATmega32A und Ethernetchip (z. B. ENC28J60)) existieren ebenso.

Die Betriebsarten der Datenübertragung

Man kann bei der Datenübertragung folgende Betriebsarten unterscheiden:

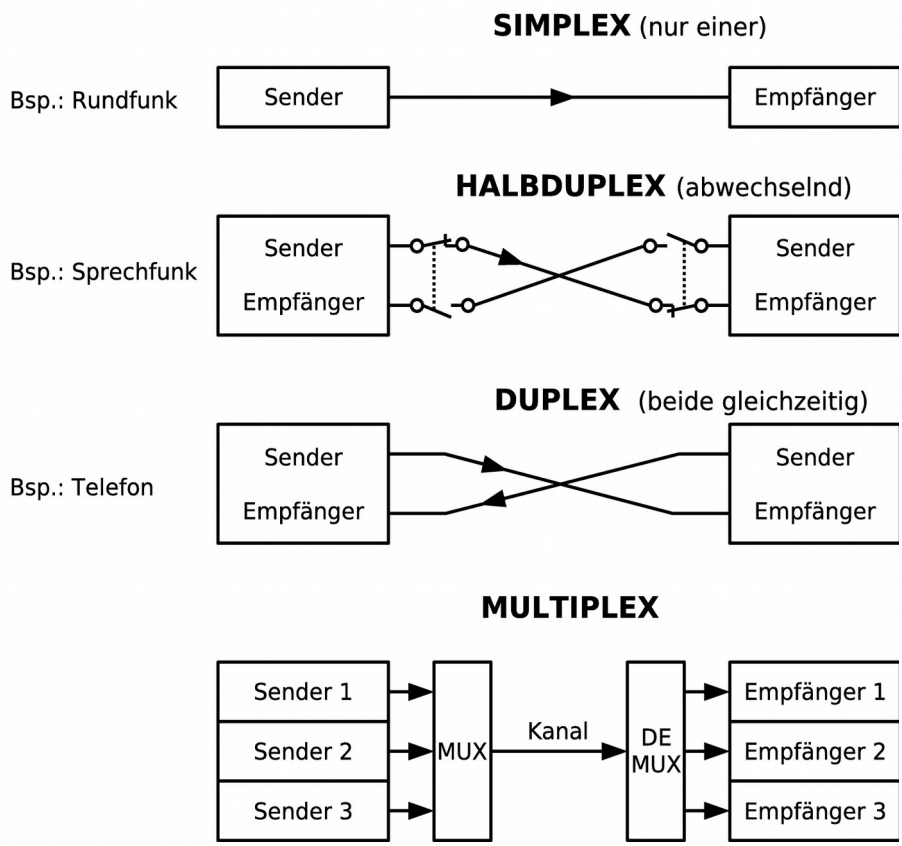
- **Seriell – Parallel** (siehe nächstes Kapitel)
- **Synchron – Asynchron** (siehe nächstes Kapitel)
- **Simplex - Halbduplex - Vollduplex - Multiplex**

Beim **Simplexbetrieb** (Richtungsbetrieb) erfolgt die Übertragung nur in eine Richtung. Es kann kein Feedback (z. B. Fehlermitteilung) erfolgen. Der Simplexbetrieb wird in der Verteilerkommunikation (z. B. Rundfunk) verwendet.

Der **Halbduplexbetrieb** (Semiduplex, Wechselbetrieb) erfolgt die Übertragung zeitlich abwechselnd. Beide Partner benötigen Sende- und Empfangseinrichtungen, die aber nicht gleichzeitig betreibbar sein müssen. Auch die Übertragungsstrecke muss nicht unbedingt doppelt ausgeführt sein (Beispiel Sprechfunk).

Beim **Vollduplexbetrieb** (Duplex, Gegenbetrieb) findet die Kommunikation gleichzeitig in beide Richtungen statt. Auch die Strecke muss Vollduplex zulassen und ist meist doppelt ausgeführt. Neue Techniken wie die Echokompensation (ISDN) erlauben allerdings auch Vollduplex auf nur einer Leitung.

Eine Sonderform der drei Betriebsarten ist der **Multiplexbetrieb** (MUX, MPX). Die Anzahl von Übertragungskanälen ist nicht beliebig erweiterbar. Eine bessere Ausnutzung der Kanalkapazität ermöglicht die gleichzeitige mehrfache Nutzung eines Kanals mit Hilfe von Multiplexern (Schaltung zum Multiplexen) und Demultiplexern. Man unterscheidet als Verfahren: **Raummultiplex**, **Zeitmultiplex**, **Frequenzmultiplex** und **Codemultiplex**.



Serielle Datenübertragung

Allgemeines

Bei der seriellen Datenübertragung wird der Datenaustausch zwischen Datensender und Datenempfänger **bitweise nacheinander** über nur eine Leitung vollzogen. Da die Daten oft in paralleler Form vorliegen, muss der Sender über ein Schieberegister eine **parallel-seriell Wandlung** realisieren. Die Taktung übernimmt meist der Sender (Sendetakt). Auf der Empfängerseite, wo die Daten wieder in paralleler Form weiterverarbeitet werden sollen, ist eine **seriell-parallel Wandlung** nötig:

Bei der Parallel-Seriell-Umwandlung werden die einzelnen Zeichenbits mit Hilfe eines Zeittakts auf die Leitung gegeben. Der Zeittakt bestimmt die **Übertragungsgeschwindigkeit** der einzelnen Bits. Man drückt diese Geschwindigkeit in **Bit/s** (**bps**, bits per second) aus. Übliche Übertragungsraten liegen heutzutage im Bereich zwischen 9600 Bit/s bis einige hundert Millionen Bit/s.

Die serielle Datenübertragung wird immer dann angewendet, wenn das Übertragungsmedium begrenzt ist, oder einen Kostenfaktor darstellt. Die serielle Datenübertragung bietet eine geringere Übertragungskapazität als die parallele Datenübertragung. Die Fortschritte in der Halbleitertechnik führten aber zu sehr schnellen kostengünstigen Seriell-Parallel-Wandlern, so dass die parallele Datenübertragung in vielen Bereichen verdrängt wurde.

Vorteile der seriellen Datenübertragung gegenüber der parallelen Datenübertragung:

- Geringere Kosten da weniger Übertragungsleitungen,
- Größere Reichweite,
- Kein Übersprechen auf benachbarte Leitungen und kein Zeitversatz (*clock skew*).

Nachteil:

- Geringere Übertragungskapazität als bei paralleler Verarbeitung.

Bei der seriellen Übertragung muss der Zeitpunkt, zu dem die ankommenden Daten vom Empfänger übernommen werden sollen, präzise definiert sein. Der Empfänger muss wissen zu welchem Zeitpunkt er abtasten soll. Die Signalerkennung muss also gleichlaufend zur Signalerzeugung erfolgen.

Man unterscheidet zwischen synchroner und asynchroner serieller Übertragung.

Das synchrone Verfahren

Beim synchronen Verfahren besteht ein festes Zeitraster, in dem die Binärzeichen eingebettet sind. Sender und Empfänger arbeiten mit dem gleichen Schritttakt und stehen ab Beginn des ersten Zeichens in einer festen Beziehung zueinander.

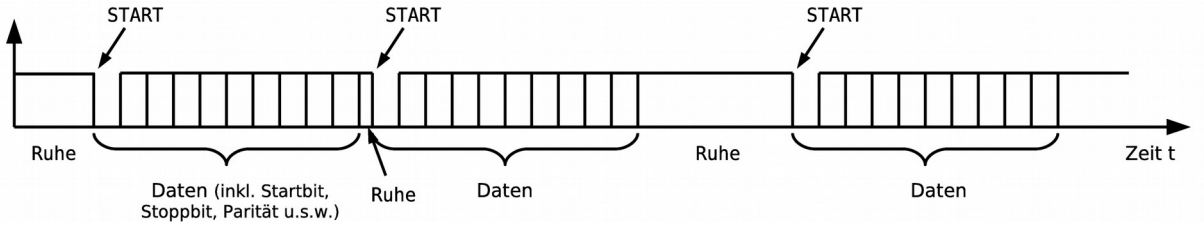
Die Synchronisation erfolgt am Anfang eines Datenblocks und bleibt während der gesamten Übertragung eines Blocks erhalten (Blocksynchronisation). Die Taktinformation kann auf einem separaten Kanal bereit gestellt werden (3 Leitungen) oder aber aus selbst taktenden Leitungscodes (Codes welche Nutzinformation und Taktinformation enthalten) abgeleitet werden. Die Blöcke selbst können asynchron versendet werden.

Die Übertragungsgeschwindigkeit ist eventuell veränderbar und kann dem Kanal angepasst werden. Das synchrone Verfahren verlangt, dass die Daten vor der Übertragung bereits vollständig vorhanden sind. Sie müssen also vor dem Versenden in einem Puffer zwischengespeichert werden.

Das asynchrone Verfahren

Bei der asynchronen Datenübertragung kann die Informationsübertragung zeichenweise zu einem beliebigen Zeitpunkt erfolgen.

Ein an einem PC arbeitender Benutzer sendet z. B. diese Zeichen in zufälligen unvorhersehbaren Intervallen. Sender (Tastatur) und Empfänger (PC) sind daher nur während der Übermittlung eines einzelnen Zeichens synchronisiert. Die Synchronisation ist durch eine festgelegte Bitrate, ein festgelegtes Datenformat sowie die Verwendung von Start- und Stoppbits möglich (keine Taktleitung).



Die asynchronen Übertragung beginnt mit einem Startbit und endet mit einem oder mehreren Stoppbits. Zusätzlich zum Zeichen können noch Kontrollbits (z. B. Parität) eingefügt werden.

Das asynchrone und synchrone Verfahren im Vergleich

Asynchrones Verfahren:

Das Übertragen eines jeden Zeichens mit einem Start- und Stoppbit ist eine einfache, aber robuste und wirkungsvolle Synchronisationsmethode. Sie erfordert weder eine Pufferung der Zeichen, noch einen ständigen Gleichlauf zwischen Sender und Empfänger. Der Aufwand für Hardware ist gering, daher sind Datenendgeräte, die mit diesem Verfahren arbeiten, kostengünstig.

Synchrones Verfahren:

Das synchrone Verfahren ist gegenüber der asynchronen Übertragung effektiver, weil innerhalb eines Datenblocks nur reine Daten (ohne Steuerbits) übertragen werden. Es wird immer dann eingesetzt, wenn es auf optimale Ausnutzung der Übertragungszeit ankommt.

Größere Entfernungen

Um digitale Signale über größere Entfernungen zu übertragen (z.B. Telefonleitung), benötigt man **Mod**em**s** (**Mod**ulator-**Dem**odulator) an der Sendeseite und an der Empfangsseite. Das Modem realisiert durch eine Modulation die Umsetzung in Signale, die an den Übertragungsweg angepasst sind. Durch zusätzliche Datencodierung kann die Übertragung auch effizienter gemacht werden, da

pro Übertragungsschritt mehrere Bits gleichzeitig ausgesendet werden. Auf der Empfangsseite muss ebenfalls ein Modem für die Rückgewinnung der ausgesendeten Daten sorgen (Demodulation und Decodierung).

Folgende Bezeichnungen werden verwendet (deutsche Bezeichnungen in Klammern):

DTE: ***Data Terminal Equipment***
ein Gerät, welches am Ende einer Übertragungsleitung angeschlossen ist, z. B. Computer, Terminal, Drucker oder Datalogger.
(**DEE** steht für **Daten-End-Einrichtung**)

DCE: ***Data Communication Equipment***
Bezeichnet das Zusatzgerät, welches für die Fernübertragung benötigt wird, das Modem.
(**DÜE** steht für **Daten-Übertragungs-Einrichtung**).

EIA-232

Die **EIA-232**-Schnittstelle ist seit fast 50 Jahren standardisiert. Die Norm wurde mehrmals überarbeitet. Die aktuelle amerikanische Version der Norm heißt offiziell ANSI/EIA/TIA-232-F-1997 und ist aus dem Jahr 1997. Ebenso häufig wie die aktuelle Bezeichnung EIA-232 (EIA für *Electronic Industries Alliance*) findet man die alte Bezeichnung **RS-232** (RS für *Radio Sector* bzw. *Recommended Standard*) oder die Bezeichnung **V24**. Diese rührt von Empfehlungen des früheren CCITT (*Comité consultatif international télégraphique et téléphonique*), heute **ITU** (*International Telecommunication Union*) die für die Norm verwendet wurden (CCITT V.28 für elektrische Eigenschaften, CCITT V.24 für Protokoll“-Eigenschaften).

EIA-232 definiert an sich die Verbindung zwischen einem Terminal (DTE) und einem Modem (DCE), was Timing, Spannungspegel, Protokoll und Stecker betrifft.

Mikrocontroller bedienen sich oft der EIA-232-Schnittstelle, um mit externen Geräten in Kontakt zu treten. Auch wenn diese Schnittstelle schon viele Jahre besteht, reicht ihre Geschwindigkeit für übliche Anwendungen oft aus. Sie ist äußerst robust und erlaubt auch größere Kabellängen. USB-EIA-232-Wandler ermöglichen die Verbindung mit dem PC, falls keine serielle EIA-232-Schnittstelle mehr verfügbar ist.

Der Zeichenrahmen (SDU, Serial Data Unit)

Jedes einzelne Zeichen wird innerhalb eines Zeichenrahmens (*frame*, **SDU**, *Serial Data Unit*) zwischen Steuerbits eingefasst. Im inaktiven Zustand (Ruhezustand, es wird kein Zeichen übertragen) wird die Übertragungsleitung auf logisch 1 (*Mark*) gehalten.

Der Beginn der Datenübertragung und damit auch die Synchronisation erfolgt mit Hilfe eines **Startbits** (logisch 0, ***Space***), das an den Anfang eines jeden Zeichens gesetzt wird.

Anschließend werden die Datenbits ausgesendet. Je nach gewähltem Code können dies 5, 6, 7 oder 8 Bits sein. Am häufigsten ist die Übertragung mit 8 Bit.

Man beachte, dass das niederwertigste Datenbit (LSB, *Least Significant Bit*, `D0`) zuerst übertragen wird!

Nach den Daten wird ein **Paritätsbit** und ein, anderthalb oder zwei **Stoppbits** (logisch 1) übertragen.

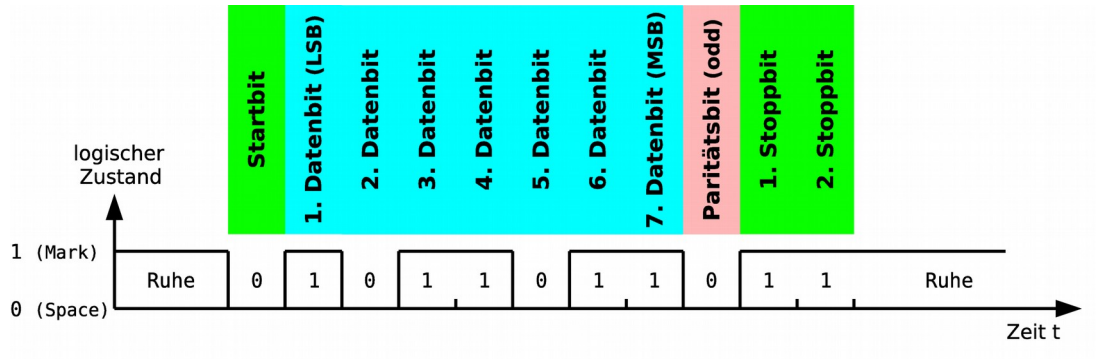
Das Paritätsbit ist ein Kontrollbit, dient der Fehlererkennung, und bezieht sich nur auf die Datenbits. Der Vorteil dieses sehr einfachen aber wenig effektiven Schutzes gegen Übertragungsfehler ist, dass er durch die Hardware ausgeführt wird (im Baustein integriert) und somit keine Rechenzeit in Anspruch nimmt. Man unterscheidet:

- **gerade Parität:** (**E**, ***even*** parity) Das Paritätsbit wird so gesetzt, dass die Summe der Einsen aus Datenbits und Paritätsbit gerade ist.
- **ungerade Parität:** (**O**, ***odd*** parity) Das Paritätsbit wird so gesetzt, dass die Summe der Einsen aus Datenbits und Paritätsbit ungerade ist.
- **keine Parität:** (**N**, ***no*** parity) Die Parität wird nicht berücksichtigt

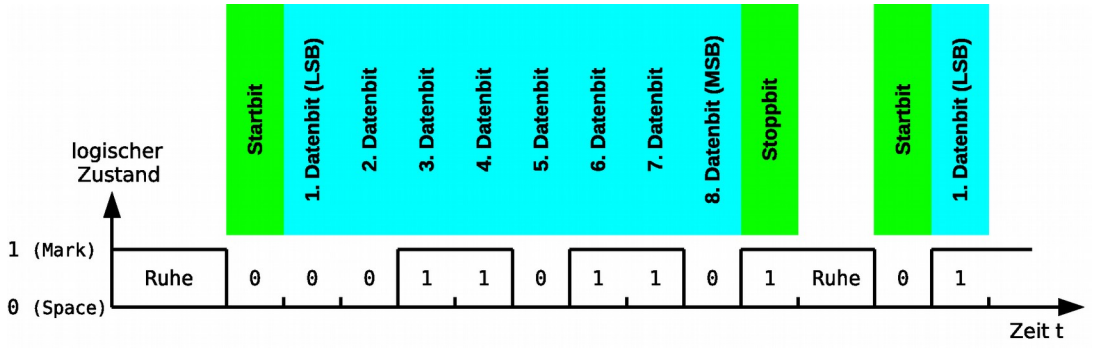
Das Paritätsbit nimmt den Wert an, der erforderlich ist, um die gewählte Parität für die Anzahl aller Eins-Werte zu erhalten.

Die nächsten beiden Bilder zeigen jeweils ein Beispiel für die logischen Zustände der Datenleitung bei der Übertragung eines Zeichenrahmens.

Beispiel 1: 7 Datenbit (0b1101101 = 'm'), Ungerade Parität, 2 Stoppbit,
Kurzschreibweise: 7O2



Beispiel 2: 8 Datenbit (0b01101100 = 'l'), Ohne Parität, 1 Stoppbit
Kurzschreibweise: 8N1
Das nachfolgende Datenbyte wird zu einem beliebigen Zeitpunkt verschickt.



Die Übertragungsgeschwindigkeit der EIA-232 Schnittstelle liegt zwischen 300 bit/s und 115200 bit/s.

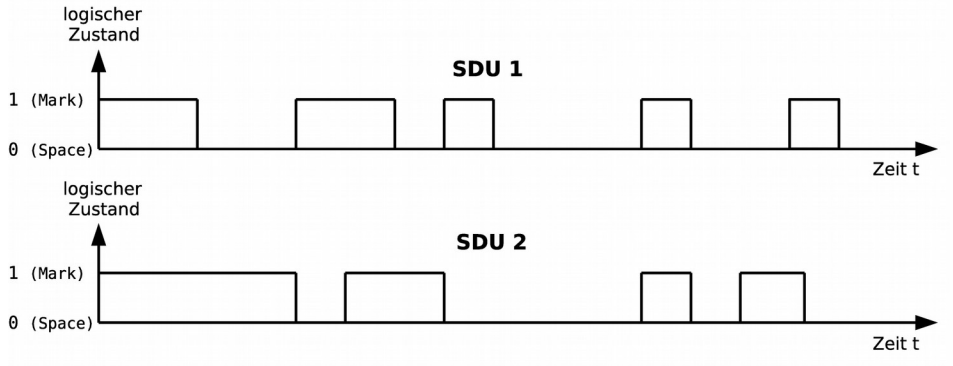
Bei jeder Signaländerung wird nur ein Bit übertragen. Die Übertragungsgeschwindigkeit ist somit der Baudrate (Signaländerung/Sekunde) bei EIA-232 gleichzusetzen[4].

Heute werden durchweg höhere Bitraten als früher eingesetzt. Häufige Bitraten sind 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600 und 115200 bit/s (bzw. Baud (Bd)). Höhere Bitraten verringern die maximale Kabellänge (siehe: Die Reichweite von EIA-232)!

Empfänger und Sender müssen auf die gleiche Geschwindigkeit (Bit- bzw. Baudrate) und das gleiche Datenformat eingestellt werden (unter Datenformat versteht man die Zahl der Datenbits und Stoppbits sowie die Parität, Bsp.: 8N1).

⌂ **C200** Bestimme aus den beiden folgenden Zeitaufnahmen einer Übertragung das mögliche Datenformat, die Anzahl der Sendeschritte (Bits) sowie das übermittelte ASCII-Zeichen, wenn mit 7 Datenbit gearbeitet wurde. Sind mehrere Möglichkeiten vorhanden, so gib sie alle an.

4 Achtung! Bei fast allen anderen Schnittstellen gilt dies nicht, da dort meist mehrere Bits pro Signaländerung übertragen werden.



⌂ **C201**

a) Zeichne einen Zeichenrahmen für die Übertragung eines ‘Z’ (8 Datenbit, ASCII-Code: 90(10)) mit gerader Parität und 2 Stoppbit.
b) Berechne die Zeichenrate (Zeichengeschwindigkeit in Zeichen/s), wenn die Übertragungsgeschwindigkeit 115200 bit/s beträgt.
c) Wie lange braucht ein Zeichen um übertragen zu werden?
d) Wie lange braucht eine Textdatei um übertragen zu werden, wenn sie 64 kByte an Daten enthält?

Die EIA-232-Schnittstellensignale

Die wichtigsten Schnittstellensignale:

25	9	Signal	Beschreibung englisch	Beschreibung	Art	Richt. PC M.	Null-aktiv
2	3	**TxD (SD)**	*Transmitter Data*	Sendedaten	Datenleit.	→	
3	2	**RxD**	*Receiver Data*	Empfangsdaten	Datenleit.	←	
4	7	**RTS**	*Request to Send*	Sendeteil eingeschaltet	Steuerleit.	→	ja
5	8	**CTS (CS)**	*Clear to Send*	Sendebereitschaft	Meldeleit.	←	ja
6	6	**DSR**	*Data Set Ready*	Betriebsbereitschaft	Meldeleit.	←	ja
7	5	**GND**	*(Signal) Ground*	Betriebserde	Erdleitung		
8	1	**DCD (CD)**	*Data Carrier Detect (Data Channel Received)*	Empfangssignalpegel	Meldeleit.	←	ja
20	4	**DTR (TR)**	*Data Terminal Ready*	Endgerät bereit	Steuerleit.	→	ja
22	9	**RI**	*Ring Indicator*	Ankommender Ruf	Meldeleit.	←	ja

2 Datenleitungen
1 Masseleitung
2 Steuerleitungen
4 Meldeleitungen

Die Richtungsangabe gilt für eine Verbindung PC - Modem. Nullaktiv bezieht sich auf die Anschlüsse am Schnittstellenbaustein bzw. Port des Mikrocontroller (TTL-Pegel[5]). Die Melde- und Steuerleitungen werden also durch Ziehen des Anschlusses auf Nullpegel (*Low*, 0V) eingeschaltet.

Man erkennt, dass alle Signale aus der Sicht des steuernden Gerätes (PC, DTE) bezeichnet wurden. Da PC und Modem (DCE) bei bidirektionaler Betriebsweise senden und empfangen können, kann die Bezeichnung manchmal zur Verwirrung führen. Besonders, da die Bezeichnungen am Modem denen des PC entsprechen (Verbindung 1:1). Verbindet man allerdings zwei PCs miteinander so müssen die Leitungen gekreuzt werden (siehe Null-Modem-Verbindung).

Die allerwichtigsten Schnittstellensignale sind die beiden Datenleitungen "***Transmitter Data***" (**TxD**) und "***Receiver Data***" (**RxD**) sowie die Masseleitung (GND). Mit diesen drei Leitungen ist eine bidirektionale Datenübertragung möglich (unidirektional bereits mit zwei Leitungen).

EIA-232 wurde erschaffen um mit einem schnellen Terminal Daten an ein ein langsames Modem zu senden. Da das Modem oft die Daten nicht ausreichend schnell entgegennehmen konnte war eine Steuerung des Datenflusses nötig. Eine softwaremäßige Datenflusssteuerung hätte die Datenübertragung verlangsamt, so dass die Steuerung hardwaremäßig mit 2 Steuer – und 4 Meldeleitungen vorgenommen wurden.

Die Steuerleitung "***Data Terminal Ready***" (**DTR**) und die Meldeleitung "***Data Set Ready***" (**DSR**) dienen dazu zu signalisieren, dass das jeweilige Gerät (*Data Terminal* ≡ PC; *Data Set* ≡ Modem) eingeschaltet ist. Die Leitungen sind während der gesamten Übertragung aktiviert. Bei modernen Geräten werden die Leitungen manchmal, entgegen ihrer ursprünglichen Bestimmung, für eine erweiterte Datenflusskontrolle eingesetzt.

Die eigentliche hardwaremäßige Flusskontrolle geschieht mittels der Steuerleitung "***Request To Send***" (**RTS**; "darf ich Senden?") mit der, der PC beim Modem anfragt, ob es bereit ist. Eine Aktivierung der Meldeleitung "***Clear To Send***" (**CTS**; "bin bereit!") leitet dann das Senden ein. Kann das Modem die eingehenden Daten nicht schnell genug verarbeiten, so deaktiviert es CTS. Der PC wartet dann so lange, bis CTS wieder aktiviert wird. In umgekehrter Richtung ist so eine Steuerung nicht vorgesehen (schnelles Modem, langsamer PC).

"*Data Carrier Detect*" (DCD, Träger auf der Telefonleitung vorhanden) und "*Ring Indicator*" (RI, ankommender Anruf) werden nur bei der Übertragung PC - Modem (DTE-DCE) benötigt. Sie werden heute oft nicht mehr verwendet!

Die serielle Schnittstelle wird sehr vielfältig bei sehr unterschiedlichen Geräten eingesetzt. Die Steuer und Meldeleitungen werden dabei öfter zweckentfremdet.

Bevor man zwei Geräte verbindet sollte man sich genaustes Informieren wie die Schnittstelle eingesetzt wird.

5 Die Transistor-Transistor-Logik (TTL) ist eine Schaltungstechnik und bezeichnet eine Schaltkreisfamilie bei der bipolare Transistoren zur Realisierung von logischen Gattern verwendet wurden. Die TTL-Schaltkreisfamilie arbeitet mit einer Betriebsspannung von 5 Volt (±5 %). High-Pegel am Eingang ist dabei eine Spannung > 2 V (meist 5 V). Low-Pegel eine Spannung < 0,8 V (meist 0 V).

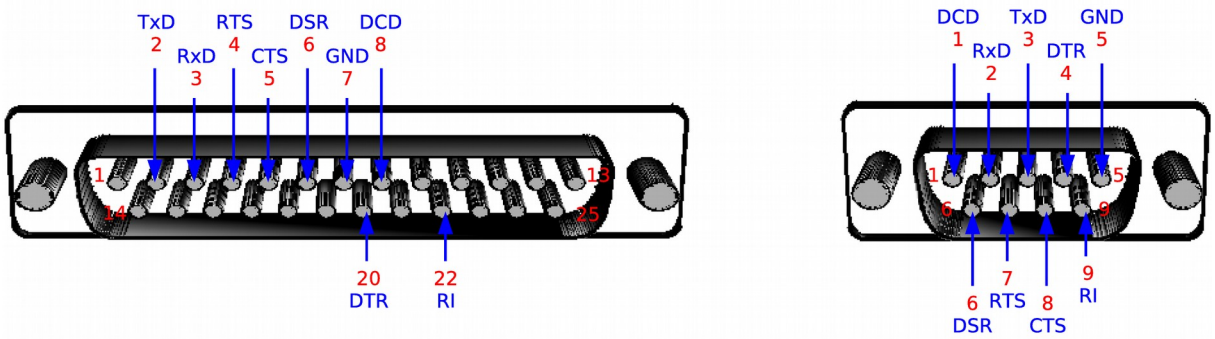
Die EIA-232-Verbindungen

Für die EIA-232-Schnittstelle werden 9- oder 25-polige D-Sub-Buchsen oder Stecker (Das D steht wegen der Ähnlichkeit der Buchse mit dem Großbuchstaben D) verwendet, wobei die 25-polige Ausführung immer weniger eingesetzt wird.

Es gibt viele verschiedene Anschlussmöglichkeiten der Leitungen, um die einzelnen seriellen Geräten miteinander zu Verbinden. Welche Verbindungsmöglichkeit genutzt wird, hängt vom Geräte und der benötigten Übertragungssicherheit ab.

Meist wird beim PC ein 9-poliger **Stecker** (männlich, Stifte) benutzt. Am Modem wird eine Aufnahme-**Buchse** (weiblich, auch meist 9-polig) verwendet. Bei DTE-DCE-Verbindungskabel benötigt man also eine Buchse für den PC und einen Stecker für das Modem.

Steckerbelegung Draufsicht (männlich):

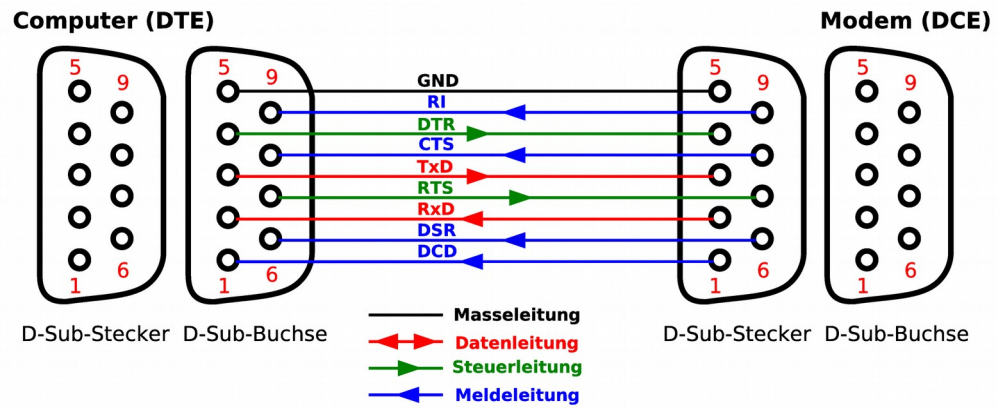


Ändert man die Perspektive (Lötperspektive anstatt Draufsicht) so ändert auch die Nummerierung! Das gleiche gilt, wenn man eine Buchse statt einem Stecker in der gleichen Ansicht betrachtet.

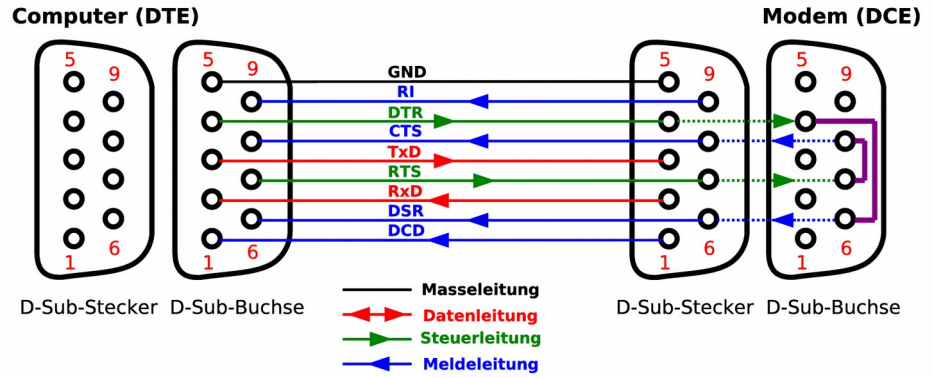
Verbindung PC (DTE) – Modem (DCE)

Wie oben beschrieben erfolgt hier die Verbindungen 1:1. Ein solches Kabel hat auf DTE-Seite eine Buchse und auf DCE-Seite einen Stecker. Es sind alle Leitungen verbunden. Es werden allerdings auch Kabel verkauft, wo aus Kostengründen nicht alle Leitungen vorhanden sind.

Im Zweifelsfall soll man das Kabel immer durchmessen.



Die hardwaremäßige Datenflusskontrolle (Handshake) für die serielle Schnittstelle kann am PC ein- bzw. ausgeschaltet werden. Auch wenn ein Gerät keine hardwaremäßige Datenflusskontrolle benötigt, sollte man damit rechnen, dass am PC die hardwaremäßige Datenflusskontrolle eingeschaltet ist. Mit Brücken zwischen RTS und CTS bzw. DSR und DTR kann man den PC überlisten. Sein eigenes Steuersignal gaukelt ihm eine positive Antwort auf der Meldeleitung vor.



Die Verbindung PC (DTE) – PC (DTE) (Nullmodemkabel)

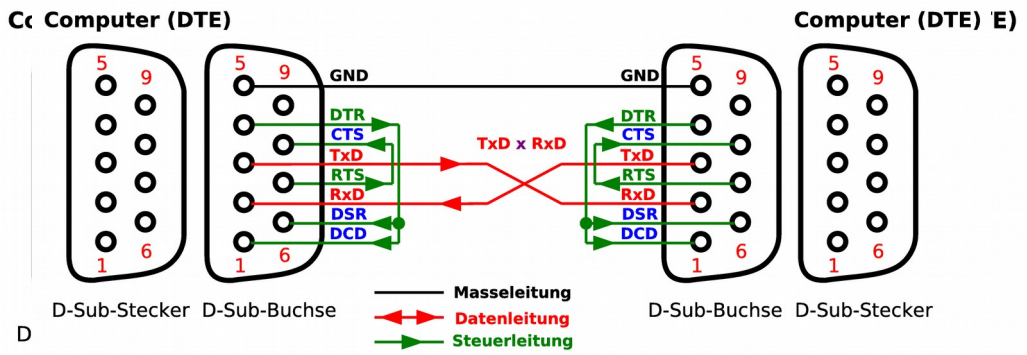
Es ist heute üblich auch zwei steuernde Geräte, zum Beispiel zwei PCs, miteinander zu verbinden. In diesem Fall ist die Sendeleitung des einen Gerätes die Empfangsleitung des anderen Gerätes und umgekehrt. Die Leitungen sind also zu kreuzen.

Dies gilt auch für die Steuer und Meldeleitungen, denn jeder Computer muss seinem Gegenüber ein Modem vortäuschen. Die Flusskontrolle funktioniert in beide Richtungen.

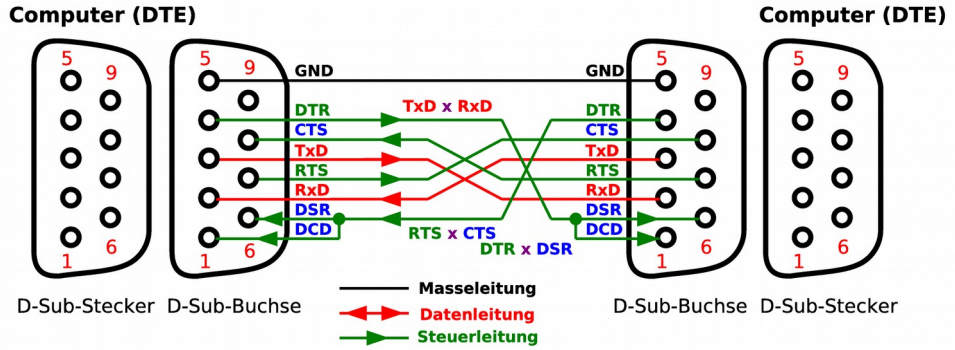
Kabel mit intern gekreuzten Leitungen nennt man **Null-Modem-Kabel**. Sie enthalten meist an beiden Enden Buchsen und sollten speziell gekennzeichnet werden um sie von anderen Kabeln unterscheiden zu können.

Für eine minimale Verbindung mit reinem Software-Handshake reicht die Datenleitungen zu kreuzen und somit drei Leitungen zu verwenden:

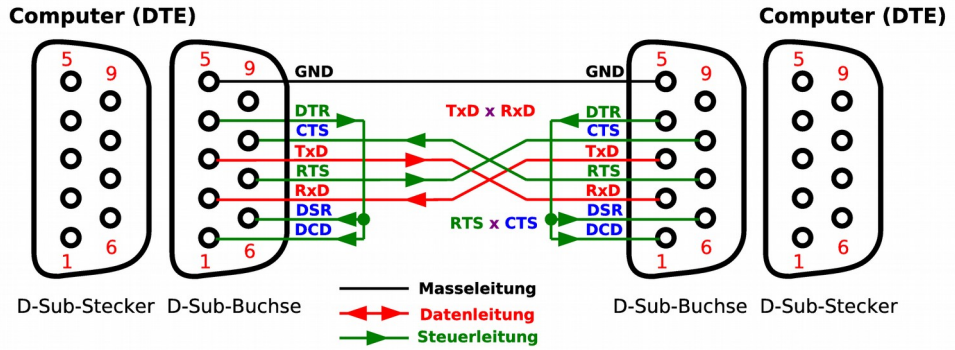
Um Probleme zu vermeiden, wenn eines der Geräte doch Meldeleitungen abfragt, kann die gebrückte Version verwendet werden:



Für ein vollständiges Hardware-Protokoll müssen die Datenleitungen und vier Steuerleitungen gekreuzt werden. Hierbei kann dann, falls erwünscht, auch noch DCD mitversorgt werden. Die Meldeleitung RI entfällt, da ja kein Modem versorgt wird.



Um die Dicke des Kabels zu verringern oder wenn kein vollständiges Handshake implementiert ist, kann die abgespeckte Versionen verwendet werden.



Es existieren auch Null-Modem-Adapter die man auf 1:1-Kabel aufstecken kann, um ein Null-Modem-Kabel zu simulieren. Besonders praktisch sind Adapter die es erlauben mit Steckbrücken die Signalleitungen beliebig zu verdrahten. *Gender-Changer* ermöglichen zusätzlich den Wechsel von Buchse zu Stecker und umgekehrt. Andere Adapter ermöglichen auch die Änderung der Polzahl der Verbindungsstecker bzw. Buchsen.

Mit einer so genannten *Loopback*-Buchse kann man die Schnittstelle testen. Mit Brücken (TxD-RxD, RTS-CLS, DTR-DSR) werden alle Signale eines Gerätes direkt zum Empfangsteil des gleichen Gerätes zurückgeführt.

Die Pegel der EIA-232-Schnittstellensignale

EIA-232 ist eine Schnittstelle, bei der die Information mittels Spannungspegel übertragen wird. (Spannungsschnittstelle im Gegensatz zu einer Stromschnittstelle wie z. B. MIDI).

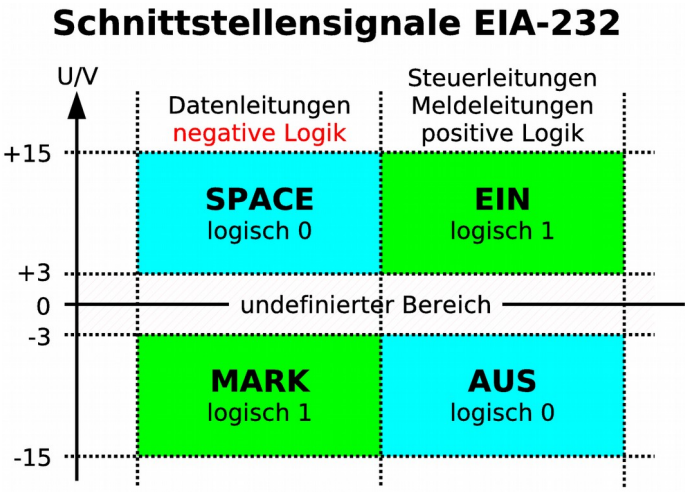
Es wird bei der EIA-232-Schnittstelle keine Leistungsanpassung verwendet, da dabei die Spannungsverluste zu hoch wären. Die Leitung wird also nicht, wie bei vielen anderen schnelleren Schnittstellen üblich, mit einem Wellenwiderstand abgeschlossen. Dadurch treten aber Reflexionen auf der Leitung auf, die neben der Dämpfung der Leitung die Reichweite verringern.

Um akzeptable Reichweiten zu erhalten, muss die Spannungsdifferenz höher als bei TTL ausfallen. Der Pegel am Empfänger liegt zwischen +15 V und -15 V. Spannungen geringer als +3 V bzw. -3 V gelten als undefiniert. Umso höher die Spannungsdifferenz, umso größer die Reichweite. Sender müssen mindestens ±5 V an einer Last von 3 bis 7 kΩ liefern. PCs arbeiten üblicherweise mit ±12 V, Notebooks mit ±7 bis 8 V.

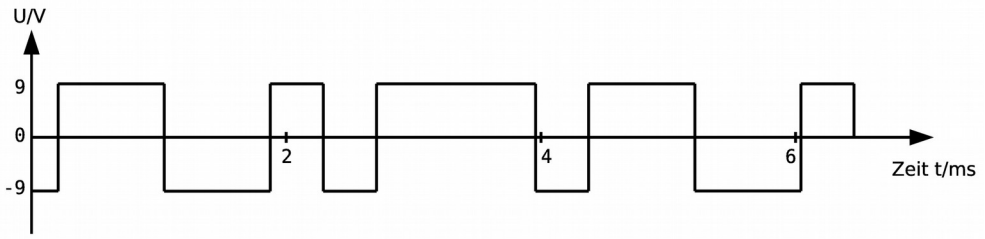
Die Datenleitungen arbeiten mit negativer Logik, die Steuer und Meldeleitungen mit positiver Logik! Alle Leitungen des Schnittstellenbausteins bzw. des Ausgabeports des Controllers werden also invertiert!

Aus der negativen Logik der (nullaktiven) Steuer- und Meldeleitungen wird eine positive Logik. High-Pegel auf der Datenleitungen werden durch eine negative Spannung dargestellt.

Da 0 V als Spannungspegel nicht möglich ist, kann eine Unterbrechung der Leitung immer sofort erkannt werden.



⌂ **C202** Bestimme beim folgenden von einem Oszilloskopschirm abgezeichneten Bild das verwendete Datenformat (8 Datenbit), die Anzahl der Sendeschritte (Bits), das übermittelte ASCII-Zeichen sowie die Übertragungsrate.



Die Reichweite von EIA-232

Mit Standardkabeln ist bei einer Baudrate von 19200 Baud eine Reichweite von um die 15 m möglich[6]. Mit Kabeln, welche eine besonders niedriger Kapazität aufweisen (z. B. nicht geschirmtes Netzwerkkabel UTP CAT-5), lassen auch 45 m erreichen. Folgende Tabelle zeigt Erfahrungswerte der Firma Texas Instruments (Quelle: wikipedia):

Baudrate	max. Länge
2.400	900 m
4.800	300 m
9.600	152 m
19.200	15 m
57.600	5 m
115.200	<2 m

Der Pegelwandlerbaustein 232

Eine digitale Schaltung mit TTL-Pegeln darf nicht ohne Pegelwandler an die serielle Schnittstelle angeschlossen werden, da sie sonst zerstört wird!

Will man mit TTL-Bausteinen die serielle Schnittstelle benutzen, so muss eine TTL/EIA-232-Pegelanpassung vorgenommen werden. Die Halbleiterindustrie bietet solche Pegelwandler als integrierte Schaltkreise an. Sie werden als „232“-ICs bezeichnet. Je nach Hersteller sind verschiedene Buchstaben voran gesetzt.

Diese Bausteine benötigen als Spannungsversorgung meist nur eine positive Spannung von 5 V. Mit zwei externen Kondensatoren wird die Spannung verdoppelt. Zwei weitere externen Kondensatoren helfen die Spannung zu invertieren. Inverter in der Sende- und in der Empfangsstufe sorgen für die Umwandlung der positiven in die negative Logik und umgekehrt. Im Anhang ist die Beschaltung eines solchen ICs zu sehen.

6 Die Lastkapazität, die hauptsächlich durch die Kabelkapazität bestimmt wird, soll laut Norm 2500 pF nicht überschreiten.

Die Datenflusskontrolle (bei EIA-232)

Bei der asynchronen Datenübertragung wird eine Datenflusssteuerung oder Datenflusskontrolle (*data flow control*) benötigt, wenn die Gefahr besteht, dass der Sender Daten zu schnell zum Empfänger sendet, so dass eine kontinuierliche Datenübermittlung ohne Verluste nicht möglich ist.

Der langsamere Empfänger muss also die Möglichkeit erhalten die Datenübertragung zeitweise zu unterbrechen, da er sonst mit Daten überlastet wird, und diese dadurch verloren gehen könnten.

Die Steuerung dieser Unterbrechungen ist Aufgabe der Datenflusskontrolle. Es gibt hierzu zwei Möglichkeiten:

- ***Software-Handshake*** mit im Datenstrom eingebauten Steuerinformationen.
- ***Hardware-Handshake*** mit zusätzlichen Leitungen (Steuer- und Meldeleitungen).

Die Software-Flusskontrolle

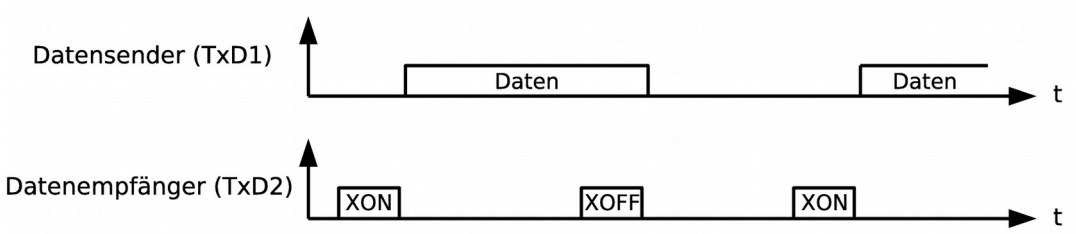
Bei der Software-Flusskontrolle (*Software-Handshake*, Software-Protokoll) geschieht die Kontrolle (Steuerung) mit im seriellen Datenstrom eingebetteten Steuerzeichen. Zwei bekannte Formen des Software-Handshake sind das

- **XON/XOFF-Protokoll** und das
- **ETX/ACK-Protokoll**.

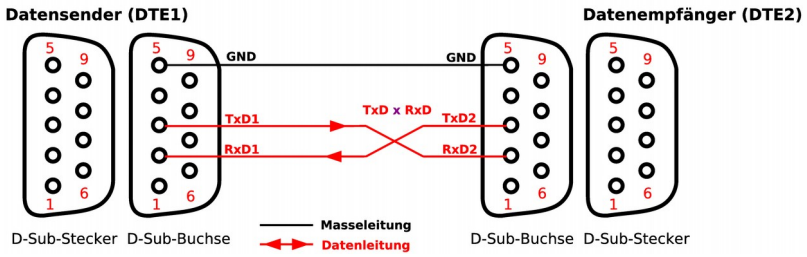
Das XON/XOFF-Protokoll

Zur Kontrolle werden die beiden Steuerzeichen **XON** (T*ransmission ON*) und **XOFF** (T*ransmission OFF*) verwendet. Die Codierung der Steuerzeichen **XON** und **XOFF** kann unterschiedlich sein, meistens wird für **XON** das ASCII-Steuerzeichen „**DC1**“ (11h) und für **XOFF** das ASCII-Steuerzeichen „**DC3**“ (13h) verwendet.

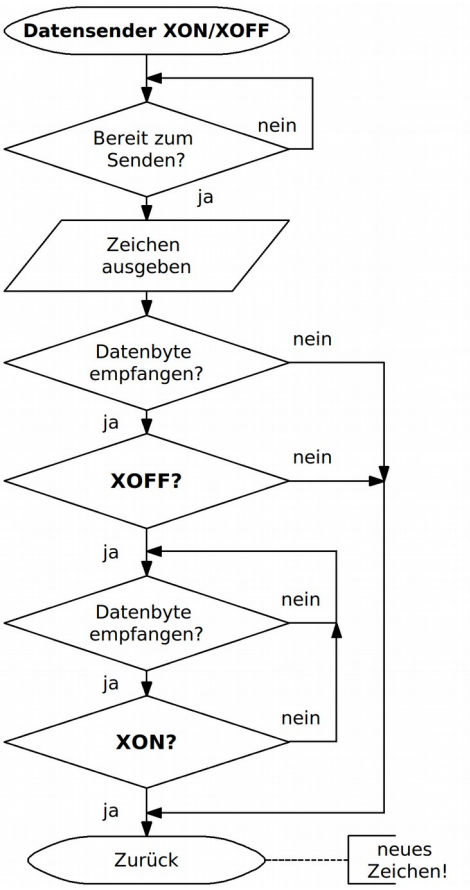
Der Empfänger bestimmt die Flusskontrolle. Er sendet bei Empfangsbereitschaft einen **XON**-Zeichen an den Sender. Sobald der Sender dieses Steuerzeichen erkannt hat, beginnt er mit dem Senden von Daten. Kann der Empfänger keine weiteren Daten mehr aufzunehmen, weil zum Beispiel sein Datenpuffer voll ist, so gibt er dem Sender mit einem **XOFF**-Zeichen zu verstehen, dass dieser die Datensendung aussetzen soll. Der Sender arbeitet erst nach dem Empfang eines erneuten **XON**-Zeichens weiter.



Für das **XON**/**XOFF** - Protokoll werden nur drei Leitungen benötigt (2 gekreuzte Datenleitungen und Betriebserde). Da gleichzeitig Daten übertragen werden können, handelt es sich um eine Vollduplex-Verbindung.

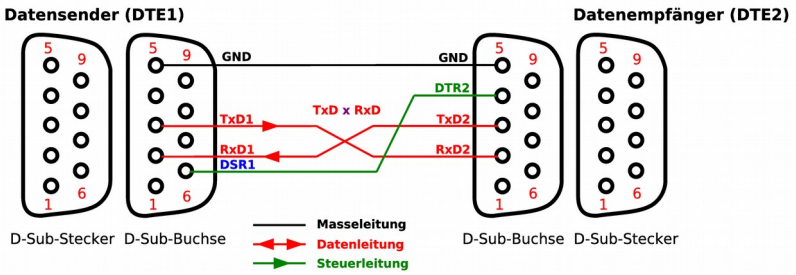


Mögliches Flussdiagramm für ein Unterprogramm zum Senden:



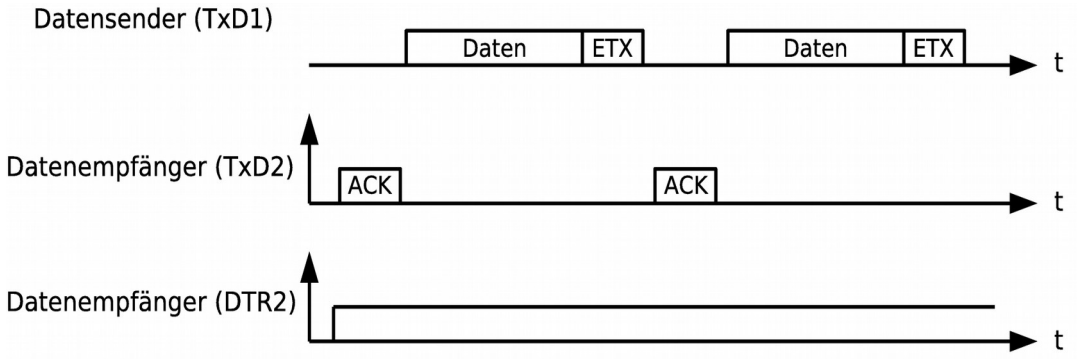
Das ETX/ACK-Protokoll

Beim **ETX**/**ACK**-Protokoll wird eine **zusätzliche Steuerleitung** (**DTR** (Empfänger) nach **DSR**) eingesetzt. Es ist also kein reines Software-Protokoll.



Es werden **Datenpakete** übertragen, wobei die Länge der Pakete von der Pufferkapazität des Empfängers abhängt (oft 128 Byte). Jedes Paket schließt mit dem ASCII-Steuerzeichen **ETX** (03h, *End of Text*) ab. Die Paketlänge muss so bestimmt sein, dass es zu keinem Überlauf des Empfangspuffers kommt! Die Betriebsbereitschaft des Empfängers wird mit der Steuerleitung **DTR** signalisiert (High-Pegel). Anschließend sendet der Empfänger das ASCII-Steuerzeichen **ACK** (06h, *Acknowledge*). Der Sender übermittelt daraufhin das Datenpaket, das mit dem **ETX** abgeschlossen ist. Ist der Empfangspuffer nach dem Empfang der Daten noch nicht voll und zur weiteren Aufnahme von Daten bereit, sendet der Empfänger erneut ein **ACK** an den Sender. Der Sender kann nun das nächste Datenpaket übermitteln.

Es handelt sich hierbei um eine Halbduplex-Verbindung, da abwechselnd Daten übertragen werden.

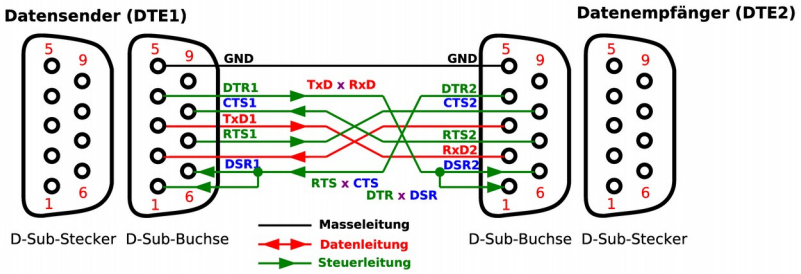


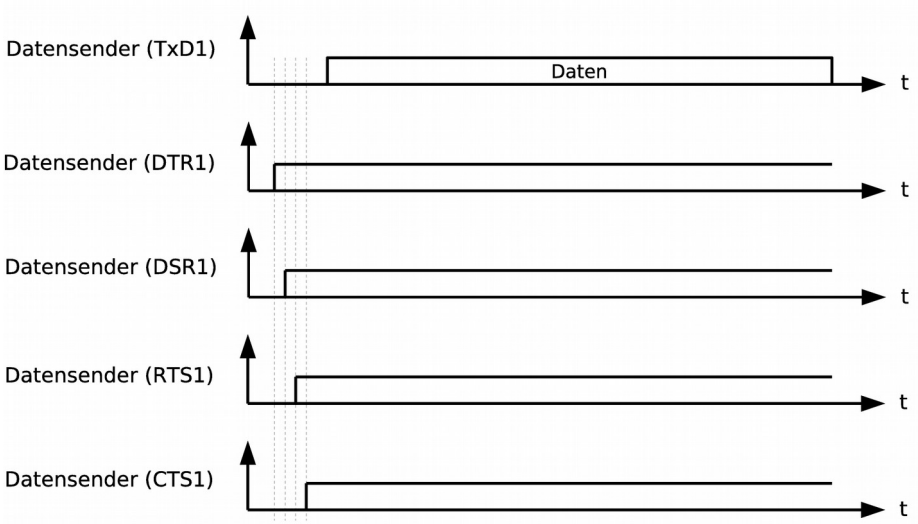
Nachteile der Software-Flusskontrolle:

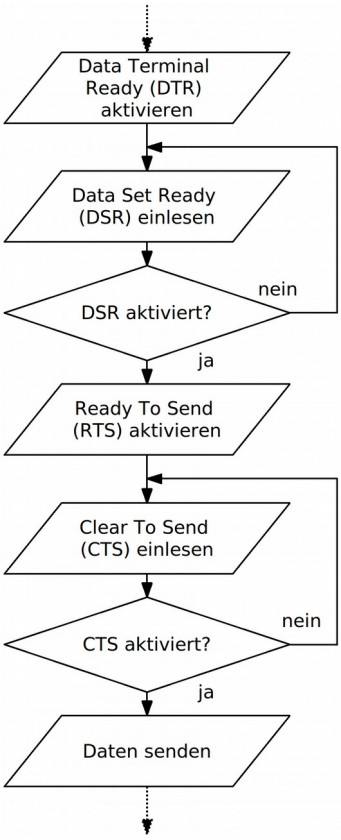
Die Software-Flusskontrolle ist durch die benötigten Steuerzeichen weniger effektiv als die Hardware-Flusskontrolle. Auch können durch die Steuerzeichen nicht alle 256 Zeichen eines Byte genutzt werden, was die Übertragung binärer Daten erschwert, da diese umcodiert werden müssen. Die Software-Flusskontrolle sollte nur genutzt werden, wenn es keine Alternative gibt.

Die Hardware-Flusskontrolle

Die Funktionsweise der Hardware-Flusskontrolle (*Hardware-Handshake*, Hardware-Protokoll) bei der asynchronen seriellen Schnittstelle wurde schon beim Kennenlernen der Steuerleitungen besprochen (PC-Modem). Im Folgenden soll nochmal das Handshake beim Aufbau einer Verbindung zwischen zwei Daten-End-Einrichtungen (PC oder Mikrocontroller) vom sendenden PC (Controller) aus mit Hilfe eines Zeitdiagramms und eines Flussdiagramms betrachtet werden.







Bemerkung: Die EIA-232-Schnittstelle sowie ihr Hardware-Handshake wurde konzipiert und genormt für eine Datenverbindung PC-Modem. Heute wird sie oft zur Realisierung von Datenverbindung ohne Verwendung eines Modems eingesetzt. Viele Gerätehersteller haben aus Kostengründen aber nur Fragmente der Schnittstelle implementiert. Die Verbindung mit Hardware-Protokoll zwischen unterschiedlichen so genannten kompatiblen Geräten ist daher oft schwierig und zeitraubend.

Der USART des ATmega32A

Viele Mikrocontroller besitzen eine so genannte **UART** (*Universal Asynchronous Receiver/ Transmitter*), also einen universelle Sender- und Empfänger-Baustein, der **asynchron** Daten übertragen kann. Seine Aufgabe ist es beim Senden mit Hilfe eines Schieberegisters einen seriellen digitalen Datenstrom mit einem fixen Rahmen aufzubauen. Beim Empfang muss eine Synchronisation über Start- und Stopp-Bit und eine bekannte Bitrate erfolgen, da kein Taktsignal vorhanden ist. Dazu wird das empfangene Signal mit einer mehrfachen Taktfrequenz abgetastet (siehe später).

Der ATmega32A besitzt, genau wie der ATmega8A und ATmega16A einen **USART** (*Universal Synchronous/Asynchronous Receiver/ Transmitter*), welche zusätzlich zu den Eigenschaften des UART auch noch synchron übertragen kann[7].

Die USART des ATmega32A hat getrennte Sende- und Empfangspuffer und ermöglicht so einen Vollduplexbetrieb. Er kann drei verschiedene Interrupts auslösen, so dass eine effektive Datenübertragung mit Interruptsteuerung möglich ist. Rahmen-, Paritäts- und Überlauffehler werden erkannt.

Weitere hier nicht genutzte Eigenschaften sind die Multiprozessor-Datenübertragung oder das Arbeiten mit doppelter Übertragungsgeschwindigkeit.

Bemerkung: Eine EIA-232-Schnittstelle kann mit niedriger Bitrate und entsprechendem Aufwand natürlich auch ohne UART bzw. USART softwaremäßig realisiert werden.

Die Initialisierung der USART

Damit eine Kommunikation über eine serielle Schnittstelle möglich wird, müssen folgende Parameter bei Sender und Empfänger richtig initialisiert werden:

1. Parameter welche die Schnittstelle betreffen (Sender/Empfänger einschalten, asynchron/synchron, Polling oder Interrupt)
2. Das Datenformat.
3. Die Baudrate.

Bevor diese Parameter umgestellt werden sollte im Zweifelsfall geklärt werden ob alle vorigen Kommunikationen abgeschlossen sind (TXC- und RXC-Flags überprüfen!).

Zur **Initialisierung und Statusabfrage** des USART dienen die drei Register:

- **UCSRA**, **UCSRB** und **UCSRC** (**U**SART **C**ontrol and **S**tatus **R**egister A, B, C).

Die **Baudrate** wird über das Doppelregister:

- **UBRR** (**U**SART **B**aud **R**ate **R**egister) eingestellt.
  Es besteht aus **UBRRH**, und **UBRRL**.

7 Es wird dann zusätzlich zu den Datenleitungen auch noch eine Taktleitung benötigt.

Daten werden über das Register:

- **UDR** (**U**SART I/O **D**ata **R**egister) ausgegeben bzw. eingelesen.

Wie aus dem Registersatz (Anhang) ersichtlich ist das SF-Register an der Adresse **0x20** doppelt belegt mit dem Register **UCSRC** und **UBRRH**. Dies war leider aus Platzgründen[8] nötig. Das höherwertige Bit **URSEL** (Bit 7) entscheidet, welches Register gerade angesprochen wird.

- **URSEL** (Bit 7) = 1 Register wird als **UCSRC** angesprochen.
- **URSEL** (Bit 7) = 0 Register wird als **UBRRH** angesprochen.

Beim Auslesen muss man Bit 7 auswerten, um das Register zu erkennen.

Daten werden am Pin **PD0** (**RxD**) eingelesen und am Pin **PD1** (**TxD**) ausgegeben. Wurde der Sende- bzw. Empfängerbaustein eingeschaltet so ist das entsprechende Pin automatisch richtig als Eingang oder Ausgang initialisiert.

Im Folgenden sollen die wichtigsten Bits für eine Datenübertragung nach EIA-232 besprochen werden.

Die USART Control und Status Register UCSRA

UCSRA-Register: SF-Register-Adresse **0x0B** (SRAM-Adresse **0x002B**)
Befehle: **in**, **out**, **sbi**, **cbi**, **sbic**, **sbis**
UCSRA beinhaltet 6 wichtige Flags zum Polling und zur Fehlerkontrolle.

UCSRA = USART Control and Status Register A

Bit	7	6	5	4	3	2	1	0
UCSRA 0x0B	**RXC**	**TXC**	**UDRE**	**FE**	**DOR**	**PE**	**U2X**	**MPCM**
Startwert	0	0	1	0	0	0	0	0
Read/Write	R	R/W	R	R	R	R	R/W	R/W

RXC ***USART Receive Complete***

0 Wenn sich keine Daten mehr im Empfangspuffer befinden oder wenn der Empfänger ausgeschaltet ist (Startwert).

1 Wird auf Eins gesetzt, wenn sich Daten im Empfangspuffer befinden. Zeigt also an, dass die Daten ausgelesen werden können.
Kann zusätzlich einen Interrupt auslösen (siehe **UCSRB**).

TXC ***USART Transmit Complete***

0 Es sind noch Daten im Schieberegister oder Datenregister vorhanden. Es kann noch kein neues Zeichen gesendet werden (Startwert).

1 Wird auf Eins gesetzt, wenn alle Daten (gesamter Rahmen) aus dem Schieberegister ausgegeben wurden und keine neuen Daten im **UDR**-Datenregister (Sendepuffer) vorliegen. Muss manuell mit einer Eins! vor der Ausgabe gelöscht werden!
Kann zusätzlich einen Interrupt auslösen (siehe **UCSRB**).

UDRE ***USART Data Register Empty***

0 Datenregister **UDR** besetzt.

1 Wird auf Eins gesetzt, wenn das Datenregister **UDR** (Sendepuffer) leer ist und der USART somit

8 Mit 6 Bit zur Registercodierung im Befehl können nur 64 SF-Register direkt adressiert werden.

bereit
: ist neue Daten anzunehmen (Startwert nach **RESET**!). Kann zusätzlich einen Interrupt auslösen (siehe **UCSRB**).

FE ***Frame Error***

0 kein Rahmenfehler.

1 **Rahmenfehler**, kein gültiges Stoppbit erkannt.

DOR ***Data OverRun***

0 kein Überlauffehler.

1 **Überlauffehler**; tritt auf, wenn beide Pufferregister (**RXB** bzw. **UDR**) und das Schieberegister belegt sind, und dann ein gültiges Startbit erkannt wird.

PE ***Parity Error***

0 kein Paritätsfehler.

1 **Paritätsfehler.**

Die Bedeutung der anderen beiden Bit kann im Datenblatt nachgelesen werden.

Die USART Control und Status Register UCSRB

UCSRB-Register: SF-Register-Adresse **0x0A** (SRAM-Adresse **0x002A**)
Befehle: **in**, **out**, **sbi**, **cbi**, **sbic**, **sbis**
UCSRB wird benötigt um Sender, Empfänger und die Interrupts ein- bzw. auszuschalten.

UCSRB = USART Control and Status Register B

Bit	7	6	5	4	3	2	1	0
UCSRB 0x0A	**RXCIE**	**TXCIE**	**UDRIE**	**RXEN**	**TXEN**	**UCSZ2**	**RXB8**	**TXB8**
Startwert	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W

RXCIE ***RXC Interrupt Enable***

0 kein **RXC**-Interrupt erlaubt.

1 Das Setzen dieses Bit **ermöglicht das Auslösen eines Interrupts** in dem Moment, wo das **RXC**-lag (**UCSRA**) gesetzt wird, also neue Daten im Empfangspuffer vorhanden sind. Interrupts müssen dazu global frei gegeben sein (**I**=1 im **SREG** mit "**sei**").

TXCIE ***TXC Interrupt Enable***

0 kein **TXC**-Interrupt erlaubt.

1 Das Setzen dieses Bit **ermöglicht das Auslösen eines Interrupts** in dem Moment, wo das **TXC**-Flag (**UCSRA**) gesetzt wird, also Schieberegister und **UDR**-Register (Sendepuffer) leer sind. Interrupts müssen dazu global frei gegeben sein (**I**=1 im **SREG** mit "**sei**").

UDRIE ***UDRE Interrupt Enable***

0 kein **TXC**-Interrupt erlaubt.

1 Das Setzen dieses Bit **ermöglicht das Auslösen eines Interrupts** in dem Moment, wo das **UDRE**-Flag (**UCSRA**) gesetzt wird, also das **UDR**-Datenregister (Sendepuffer) leer ist. Interrupts müssen dazu global frei gegeben sein (**I**=1 im **SREG** mit "**sei**").

RXEN ***Receiver ENnable***

0 schaltet den Empfänger aus.

1 **schaltet den Empfänger ein**. Pin **RxD** (**PORTD0**) ist dann als Eingang reserviert (muss nicht extra initialisiert werden) und ist für andere Aktionen nicht mehr zugänglich.

TXEN ***Transmitter ENnable***

0 schaltet den Sender aus.

1 **schaltet den Sender ein**. Pin **TxD** (**PORTD1**) ist als Ausgang reserviert (muss nicht extra initialisiert werden) und ist für andere Aktionen nicht mehr zugänglich.

UCSZ2 ***USART Character Size 2***

siehe nächstes Register **UCSRC**

Die Bedeutung der anderen 2 Bit kann im Datenblatt nachgelesen werden.

Die USART Control und Status Register UCSRC

UCSRC-Register: SF-Register-Adresse **0x20** (SRAM-Adresse **0x0040**)
Befehle: **in**, **out** (**sbi**, **cbi**, **sbic**, **sbis** nicht da Adresse > 32 (0x1F)!)
Im **UCSRC**-Register wird das Datenformat (z.B.: 8N1) festgelegt.

UCSRC = USART Control and Status Register C

Bit	7	6	5	4	3	2	1	0
UCSRC 0x20	**URSEL**	**UMSEL**	**UPM1**	**UPM0**	**USBS**	**UCSZ1**	**UCSZ0**	**UCPOL**
Startwert	1	0	0	0	0	1	1	0
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W

URSEL ***USART Register SELect***

Der Speicherplatz ist mit zwei Registern belegt.

0 das Register **UBRRH** **wird ausgewählt**. Beim Lesen von **UBRRH** ist das Bit Null.

1 das Register **UCSRC** **wird ausgewählt** (default). Beim Lesen von **UCSRC** ist das Bit Eins.

UMSEL ***USART Mode SELect***

0 **Asynchroner Modus**

1 **Synchroner Modus**

UPMn ***USART Parity Mode UPM1, UPM0***

Ist die Parität eingeschaltet (gerade oder ungerade Parität) wird hardwaremäßig die Parität generiert und in den Zeichenrahmen mit integriert. Als Empfänger kontrolliert der USART die Parität. Ist ein Fehler aufgetreten, so wird das **PE**-Flag (Parity Error) in **UCSRA** gesetzt.

00 **keine Parität**

01 reserviert

10 **gerade Parität** eingeschaltet

11 **ungerade Parität** eingeschaltet.

USBS ***USART Stop Bit Select***

0 **1 Stoppbit**

1 **2 Stoppbit**

UCSZn ***USART Character Size*** ***UCSZ2, UCSZ1, UCSZ0***

Mit diesen drei Bit wird die **Anzahl der Datenbits (Zeichengröße) eingestellt**. **UCSZ2** befindet sich auf Bit 2 im **UCSRB**-Register und wird nur benötigt wenn 9 Bit benötigt werden. Sonst kann dieses Bit unberücksichtigt bleiben, da sein Default-Wert Null ist.

UCSZ2 UCSZ1 UCSZ0 2^2 2^1 2^0	Anzahl der Datenbits
000	5
001	6
010	7

011	8
100	reserviert
101	reserviert
110	reserviert
111	9

UCPOL ***USART Clock POLarity***
Wird nur bei synchroner Datenübertragung benutzt
0 bei **asynchroner** Übertragung

⌂ **C203** Welcher Wert muss nach **UCSRC** geschrieben werden um eine asynchrone Übertragung mit **8N1** zu initialisieren?

Das USART Baud Rate (Doppel-) Register UBRR

Um die Baudrate einzustellen wird ein 12 Bit großer Teiler benötigt. Die Hochwertigen vier Bit befinden sich im Register **U**SART **B**aud **R**ate **R**egister **H**igh **UBRRH**. Das niederwertige Byte des Teilers befindet sich im Register **U**SART **B**aud **R**ate **R**egister **L**ow **UBRRL**.

Die Register dürfen nur beschrieben werden wenn keine Datenübertragung stattfindet, da eine Änderung der Baudrate sofort wirksam wird und so zu einem Fehler führen würde.

UBRRH-Register: SF-Register-Adresse **0x20** (SRAM-Adresse **0x0040**)
Befehle: **in**, **out** (**sbi**, **cbi**, **sbic**, **sbis** nicht da Adresse > 32 (0x1F)!)

UBRRH = USART Baud Rate Register High

Bit	7	6	5	4	3	2	1	0
UBRRH 0x20	**URSEL**	**-**	**-**	**-**	**UBRR 11**	**UBRR 10**	**UBRR 9**	**UBRR 8**
Startwert	0	0	0	0	0	0	0	0
Read/Write	R/W	R	R	R	R/W	R/W	R/W	R/W

UCSRB-Register: SF-Register-Adresse **0x09** (SRAM-Adresse **0x0029**)
Befehle: **in**, **out**, **sbi**, **cbi**, **sbic**, **sbis**

UBRRL = USART Baud Rate Register Low

Bit	7	6	5	4	3	2	1	0
UBRRL 0x09	**UBRR 7**	**UBRR 6**	**UBRR 5**	**UBRR 4**	**UBRR 3**	**UBRR 2**	**UBRR 1**	**UBRR 0**
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

URSEL ***USART Register SELect***
Der Speicherplatz ist mit zwei Registern belegt.
0 das Register **UBRRH wird ausgewählt**. Beim Lesen von **UBRRH** ist das Bit Null.
1 das Register **UCSRC wird ausgewählt** (Startwert). Beim Lesen von **UCSRC** ist das Bit Eins.

UBRRn ***USART Baud Rate Register Bit n***
In diesen 12 Bit befindet sich der Teiler aus dem sich die Baudrate berechnen lässt. Die Abtastung des

Eingangsignals **RxD** erfolgt mit dem 16-fachen Übertragungstakt (Baudrate). Die Formel lautet:

$$Baudrate = \frac{Systemtakt}{16 \cdot (Teiler + 1)}$$

Umgekehrt lässt sich natürlich auch der Teiler bei erwünschter Baudrate errechnen:

$$Teiler = \frac{Systemtakt}{16 \cdot Baudrate} - 1$$

Die mit dem Teiler erreichte Baudrate entspricht nicht immer dem genauen Standardwert. Der Fehler (in Prozent) errechnet sich mit:

$$Fehler[\%] = \left(\frac{Baudrate}{Standardbaudrate} - 1 \right) \cdot 100\%$$

Fehler unter 0,5 % sollen angestrebt werden, da sonst die Fehlerrate bei der Übertragung zunimmt. Die kann zum Beispiel durch das Auswechseln des Quarzes erfolgen oder durch Verdoppeln der Baudrate mit **U2X** in **UCSRA** (siehe Datenblatt).

⌂ **C204** Ergänze die folgende Tabelle für den 16 MHz-Quarz. Markiere die ungünstigen Baudraten (Fehler > 0,5%) mit roter Farbe.

Standard-Baudrate [bps]	UBRR	Baudrate [bps]	Fehler [%]
2400			
4800			
9600			
14400			
19200			
28800			
38400			
57600			
115200			
230400			
250000			
500000			
1000000			

Das USART Daten Register UDR

Werden Daten zum Senden in das Datenregister **UDR** geschrieben, so landen sie Pufferspeicher des Senders **TXB** (*Transmit Data Buffer Register*)[9]. Beim Lesen des **UDR** werden die Daten in den

9 Der Sendepuffer ist nur schreibbar.

Pufferspeicher des Empfängers **RXB** (*Receive Data Buffer Register*)[10] geladen. Diese getrennten Pufferspeicher ermöglichen Voll-Duplex-Operationen.

Wird mit weniger als 8 Datenbit gearbeitet, so werden die nicht benötigten Bits ignoriert oder auf Null gesetzt.

UDR-Register: SF-Register-Adresse **0x0C** (SRAM-Adresse **0x002C**)
Befehle: **in**, **out**, **sbi**, **cbi**, **sbic**, **sbis**

UDR = USART I/O Data Register

Bit	7	6	5	4	3	2	1	0
UDR r/w 0x0C	**RXB7 TXB7**	**RXB6 TXB6**	**RXB5 TXB5**	**RXB4 TXB4**	**RXB3 TXB3**	**RXB2 TXB2**	**RXB1 TXB1**	**RXB0 TXB0**
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

TXBn ***Transmit Data Buffer Register Bit n***
Sendepuffer.

RXBn ***Receive Data Buffer Register Bit n***
Empfangspuffer.

Beispiel für eine Initialisierung

```
        ;USART initialisieren
        ;Sender und Empfaenger einschalten und Datenformat festlegen
        sbi     UCSRB,RXEN      ;Empfaenger einschalten (RxEnable)
        sbi     UCSRB,TXEN      ;Sender einschalten (TxEnable)
        ldi     Tmp1,0x86       ;UCSRC = 0b10000110
        out     UCSRC,Tmp1      ;Waehle Asynchr. 8N1
                                ;B7  URSEL  =1   UCSRC ausgewaehlt
                                ;B6  UMSEL  =0   Asynchron
                                ;B45 UPM    =00  keine Paritaet
                                ;B3  USBS   =0   1 Stoppbit
                                ;B21 UCSZ   =11  8 Datenbit
                                ;B0  UCPOL  =0   bei Asyn.
        ;Baudrate initialisieren
        clr     Tmp1            ;Teiler fuer Baudrate festlegen
        out     UBRRH,Tmp1      ;HByte = 0, da Teiler < 255
        ldi     Tmp1,103        ;Teile durch dezimal 103 fuer 9600 Baud (16MHz)
        out     UBRRL,Tmp1      ;
```

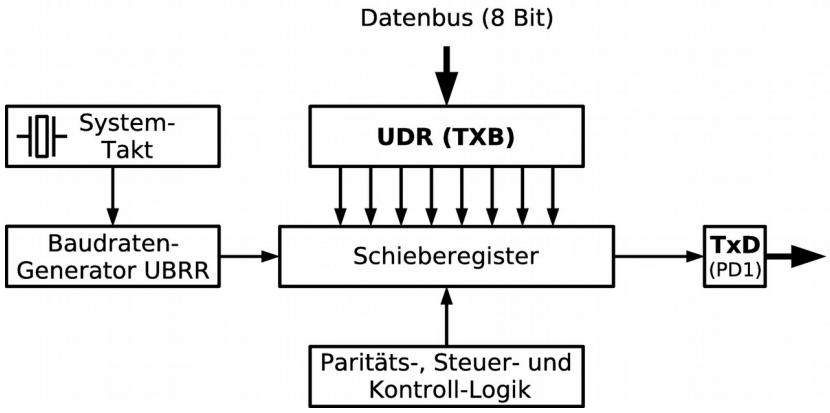
EIA-232-Sender und -Empfänger

Der USART als EIA-232-Sender (ohne Interrupt)

Falls der Sender eingeschaltet ist (**TXEN** in **UCSRB**) wird die Datenübertragung durch ein Schreiben von Daten in das Datenregister **UDR** ausgelöst (z.B.: **out UDR,r16**). Dies ist allerdings nur möglich, wenn das **UDRE**-Bit (**UDR** *Empty*) im **UCSRA**-Register gesetzt (Eins) ist, das Datenregister **UDR** also leer ist.

10 Der Lesepuffer ist zweistufig ausgeführt und nur lesbar.

Beim Schreiben werden die Daten in den Sendepuffer **TXB** geladen. Der Sendepuffer gibt die Daten gleich weiter an ein Schieberegister (nachdem die letzten Daten ausgegeben wurden und somit das Schieberegister leer ist). Die Daten im Schieberegister werden um die nötigen Zusatzbits (Start-, Paritäts- und Stoppbit) zum Zeichenrahmen ergänzt und dann seriell im Takt des Baudraten-Generator am Pin **TxD** (**PD1**) rausgeschoben.



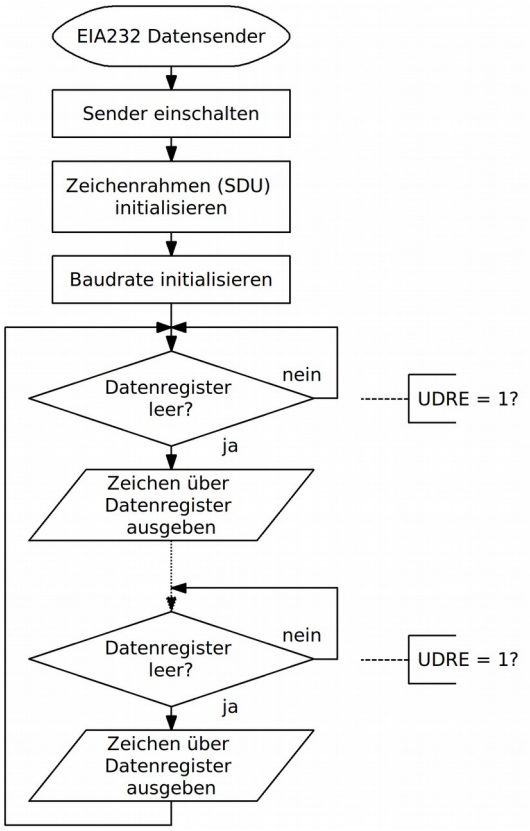
Wenn alle Daten (gesamter Rahmen) aus dem Schieberegister ausgegeben wurden und keine neuen Daten im **UDR**-Datenregister vorliegen, dann wird das **TXC**-Flag im **UCSRA** gesetzt.

UDRE ist solange gesetzt (Eins), wie das Datenregister **UDR** (**TXB**) leer ist, also Daten angenommen werden können. Es wird automatisch gelöscht, sobald Daten nach **UDR** geschrieben werden. Es ist also wichtig, nur dann Daten nach **UDR** zu schreiben wenn dieses Bit gesetzt ist. Daten die ankommen, wenn der Sendepuffer noch nicht frei ist, werden einfach ignoriert und gehen dadurch verloren.

Durch ständige Abfrage (Polling) muss im Programm sichergestellt werden, dass UDR frei ist (UDRE = 1).

Eine andere Möglichkeit wäre das **TXC**-Flag in **UCSRA** zu überwachen. Allerdings wird dieses Bit nicht automatisch rückgesetzt. Dies muss manuell durch Schreiben einer Eins! in das **TXC**-Bit erfolgen bevor Daten nach **UDR** geschrieben werden. Die Kontrolle erfolgt dann nach der Ausgabe.

Prinzipielles Flussdiagramm:



Programmbeispiel:

```
;-------------------------------------------------------------------------------
;       Hauptprogramm
;-------------------------------------------------------------------------------

MAIN:   sbis    UCSRA,UDRE      ;Warte bis UDREmpty gesetzt ist, da der Schreib-
        rjmp    MAIN            ;versuch sonst ignoriert wird
        ldi     Tmp1,'T'        ;Lade einen Buchstaben in das Ausgaberegister
        out     UDR,Tmp1        ;und gib das Zeichen aus

WDRES1: sbis    UCSRA,UDRE      ;Warte bis UDREmpty gesetzt ist, da der Schreib-
        rjmp    WDRES1          ;versuch sonst ignoriert wird
        ldi     Tmp1,'3'        ;naechstes Zeichen
        out     UDR,Tmp1        ;

WDRES2: sbis    UCSRA,UDRE      ;Kontrolliere UDRE
        rjmp    WDRES2          ;
        ldi     Tmp1,'E'        ;naechstes Zeichen
        out     UDR,Tmp1        ;

WDRES3: sbis    UCSRA,UDRE      ;Kontrolliere UDRE
        rjmp    WDRES3          ;
        ldi     Tmp1,'C'        ;naechstes Zeichen
        out     UDR,Tmp1        ;

WDRES4: sbis    UCSRA,UDRE      ;Kontrolliere UDRE
        rjmp    WDRES4          ;
        ldi     Tmp1,0x0D       ;naechstes Zeichen hier das Steuerzeichen
        out     UDR,Tmp1        ;"carriage return" (neue Zeile)

        rjmp    MAIN            ;Endlosschleife
```

⌂ **C205** a) Schreibe ein Assemblerprogramm mit Kommentaren, das nur die zwei Zeichen 'd' und 'a' hintereinander in einer Endlosschleife ausgibt. Das Senden des Zeichens soll in einem Unterprogramm mit dem Namen **SNDCHR** erfolgen. Die Übergabe

erfolgt mit der globalen Variablen **r24**.
Die Baudrate soll 19200 Baud betragen; als Datenformat wird **702** gewählt.
Speichere das Programm als "**C205_EIA232_sender_char_1.asm**".

b) Schließe einen PC an die serielle Schnittstelle an. Konfiguriere ein Terminal programm (z. B.: HyperTerminal unter Windows, Cutecom oder Minicom unter Linux) mit den richtigen Parametern (Datenformat, Baudrate, richtige Schnittstelle, keine Flusskontrolle, ASCII-Text …) und betrachte das Ergebnis auf dem Bildschirm.

c) Schließe einen Oszilloskopen zuerst an das Ausgangspin (**TxD**, **PD1**) des Controllers an und danach an das Ausgangspin des Pegelwandlers (TxD am EIA-232-Wandler). Zeichne beide SDUs ab. Beschrifte die einzelnen Bits. Bestimme die Dauer eines Bits.

d) Lösche die Zeilen, die für das Polling von **UDRE** zuständig sind und starte das Programm erneut. Betrachte das Ergebnis im Terminalprogramm und auf dem Oszilloskop. Was stellst du fest?

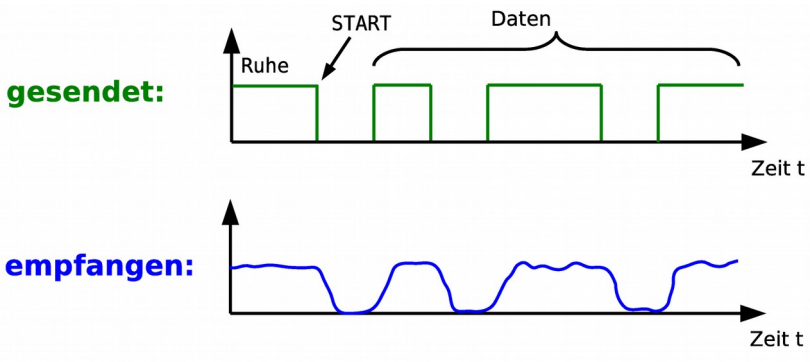
e) Für Fleißige:
Teste das Polling mit **TXC** statt **UDRE**. Achte darauf, dass vor dem Schreiben nach **UDR** das Bit mit einer Eins gelöscht werden muss. Auch muss vor dem ersten Polling nach **UDR** geschrieben werden da sonnst, **TXC** nie gesetzt wird. Betrachte das Ergebnis im Terminalprogramm und auf dem Oszilloskop. Ist der zeitliche Unterschied zu der Methode mit **UDRE** erkennbar?
Speichere das Programm als "**C205_EIA232_sender_char_2.asm**"

⌂ **C206** Meist möchte man natürlich mehr als ein paar Zeichen senden. Es soll ein Assemblerprogramm mit Kommentaren geschrieben werden, der einen ganzen Satz aus einer Tabelle ausgibt. Der Satz schließt mit dem Steuerzeichen Wagenrücklauf (*Carriage Return*, **CR**, **0x0D**) ab. Das Ende der Tabelle wird durch das Nullbyte (**0x00**) markiert. Die Daten sollen mit 38400 Baud 8E1 übertragen werden.
Speichere das Programm als "**C206_EIA232_sender_string_1.asm**".

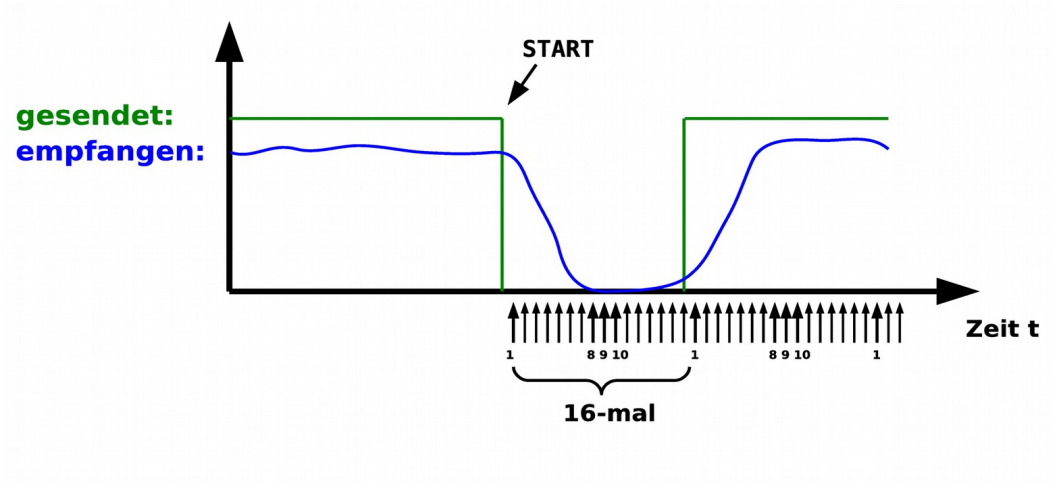
Der USART als EIA-232-Empfänger (ohne Interrupt)

Falls der Empfänger eingeschaltet ist (**RXEN** in **UCSRB**) besteht die Möglichkeit eingegangene Daten über das Datenregister **UDR** einzulesen.

Durch die begrenzte Bandbreite der Datenübertragungswege werden die Flanken der gesendeten rechteckförmigen Signale abgeflacht. Die Dämpfung vermindert zudem den Signalpegel.

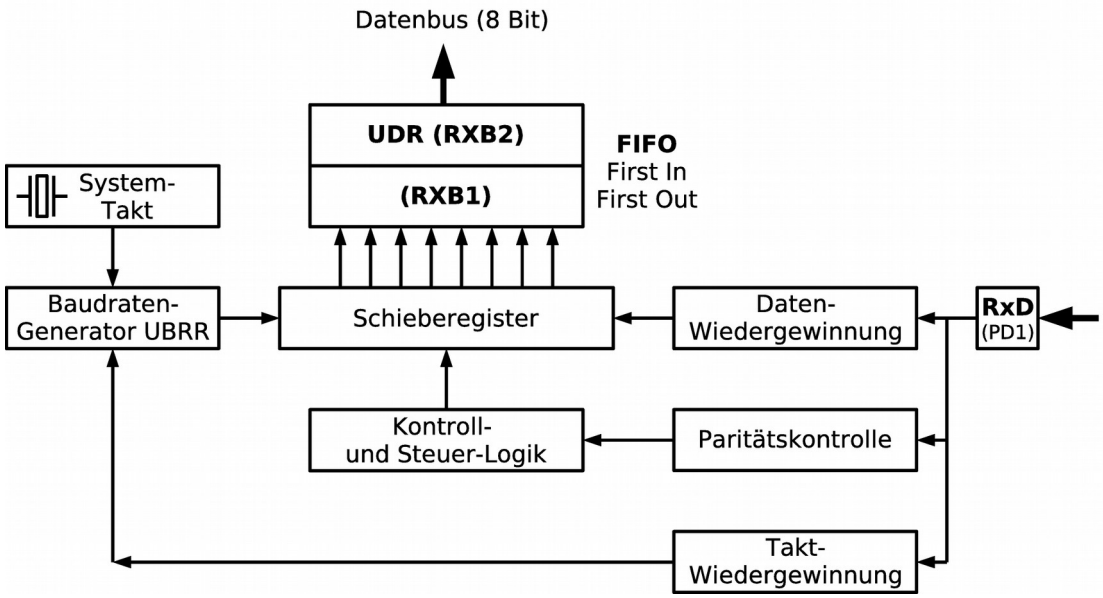


Der Empfangsteil des USART muss durch Abtastung aus diesem asynchronen verzerrten Signal die richtigen logischen Pegel wiedergewinnen. Damit dies gelingt, wird das Signal mit der 16-fachen Frequenz der Baudrate abgetastet. Sobald eine fallende Flanke erkannt wurde, wird 16-mal abgetastet. Entsprechen die Abtastwerte 8, 9 und 10 einer Null, so wird das Bit als gültige Null gewertet.



Ist ein gültiges Startbit erkannt worden, so werden die restlichen Daten abgetastet, eingelesen und seriell in das Empfangs-Schieberegister übertragen. Wurde beim Empfang ein Fehler entdeckt (falsche Parität, Datenüberlauf, Rahmenfehler) so wird das entsprechende Bit (**PE**, **DOR**, **FE**) im **USCRA** gesetzt.

Die Fehlerflags sollten in Programmen immer ausgewertet werden, um Übertragungsfehler zu erkennen. Dies muss, da sie mit den Daten im Sendepuffer gespeichert werden, vor! dem Auslesen der Daten erfolgen. Anderweitig geht die Information der Fehlerflags verloren.



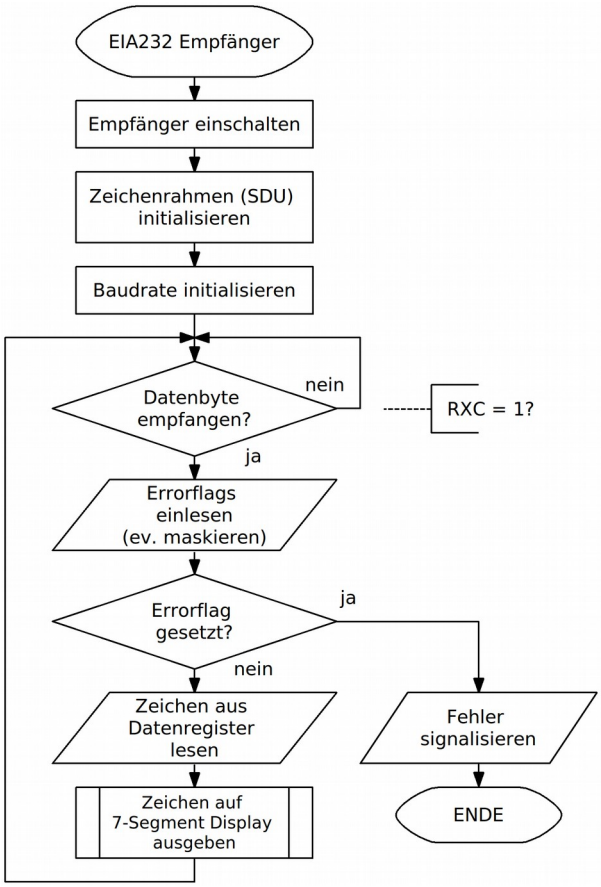
Wurde ein Zeichen vollständig empfangen, so wird das Flag **RXC** (*Receive Complete*) im **UCSRA** gesetzt. Sobald dieses Bit gesetzt ist, kann das Datenbyte aus dem Empfangspuffer **RXB** ausgelesen werden. Dies geschieht über das Register **UDR** (z. B. `in r16,UDR`). Nach dem Auslesen wird das **RXC**-Flag wieder automatisch gelöscht (Null). Ein doppeltes Auslesen wird so verhindert.

Interessant ist, dass der Empfangspuffer zweistufig ausgeführt ist. Die zwei Empfangspuffer-Register arbeiten als FIFO-Puffer (*First In First Out*). Sind beide Register voll, kann sogar das Schieberegister noch als drittes Pufferregister aushelfen. Durch den Puffer kann das Programm auch mal länger brauchen als die maximale Verarbeitungszeit, ohne dass es zu einem Überlauf kommt (keine Echtzeit-Verarbeitung mehr nötig). In Mittel muss die maximale Verarbeitungszeit bei pausenloser Versendung von Zeichen eingehalten werden.

Ein solches FIFO kann man sich als Rutschbahn vorstellen. Oben kommen die Bytes rein und unten werden sie entnommen. Die mehrstufige Ausführung ermöglicht es dem Controller mehr Zeit zu lassen bei der Bearbeitung anderer Aufgaben, da er nicht sofort nach dem Empfang das Byte abholen muss. Der Anwender muss sich nicht um das FIFO-Puffer kümmern, da es hardwaremäßig verwaltet wird.

⌂ **C207** Errechne die Anzahl der Taktzyklen (Zeit), innerhalb derer ein mit 16 Mhz getakteter Controller, die Daten in Mittel abholen muss (**8N1**, 115200 Baud), damit kein Überlauf entsteht. Wie lange könnte es kurzzeitig beim ATmega32A durch den FIFO-Puffer dauern?

⌂ **C208** Das folgende Flussdiagramm ist gegeben. Schreibe das entsprechende Assemblerprogramm mit Kommentaren. Die Datenübertragung soll mit 19200 Baud (**702**) erfolgen. Die Ausgabe des vom PC mittels Modemprogramm (z. B. Cutecom, Minicom, Hyperterminal) gesendeten Zeichen erfolgt auf der 7-Segmentanzeige (mit Hilfe des Unterprogramms **SR_NUMDISPLAY.asm**). Es wird kein Handshake verwendet. Speichere das Programm als **C208_EIA232_receiver_char_1.asm.**



⌂ **C209**

a) Zeichne das Flussdiagramm des folgenden Programms (**C209_EIA232_receiver_char_2.asm).**
b) Beschreibe präzise die Aufgabe des Programms.
c) Ergänze das Programm mit sinnvollen kurzen Kommentaren.
d) Teste das Programm.

```
;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
INIT:
.DEF    Tmp1 = r16      ;Register 16 dient als erster Zwischenspeicher
.EQU    LEDPort = PORTC
.EQU    LEDPin  = PINC

        ldi     Tmp1,HIGH(RAMEND)
        out     SPH,Tmp1
        ldi     Tmp1,LOW(RAMEND)
        out     SPL,Tmp1

        sbi     UCSRB,RXEN

        ldi     Tmp1,0x00
        out     UBRRH,Tmp1
        ldi     Tmp1,0x33
        out     UBRRL,Tmp1

        ldi     Tmp1,0x87
        out     DDRC,Tmp1

;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
```

```
MAIN:   sbis    UCSRA,RXC
        rjmp    MAIN
        in      Tmp1,UCSRA
        andi    Tmp1,0b00011100
        brne    ERROR
        in      Tmp1,UDR
        cpi     Tmp1,'L'
        breq    CHRL
        cpi     Tmp1,'E'
        breq    CHRE
        cpi     Tmp1,'D'
        breq    CHRD
        rjmp    MAIN
CHRL:   sbi     LEDPort,0
        rjmp    MAIN
CHRE:   sbic    LEDPIN,0
        sbi     LEDPort,1
        rjmp    MAIN
CHRD:   sbis    LEDPin,0
        rjmp    MAIN
        sbic    LEDPin,1
        sbi     LEDPort,2
        rjmp    MAIN
ERROR:  sbi     LEDPort,7
END:    rjmp    END

;-------------------------------------------------------------------------------
.EXIT                           ;Ende des Quelltextes
```

Polling oder Interrupts?

Der Pollingbetrieb gestaltet sich einfacher und ist immer dann angebracht, wenn der Controller sowieso nur eine Aufgabe erledigen soll, oder die Aufgaben sich gut nacheinander lösen lassen.

Soll aber zum Beispiel die Datenübertragung gleichzeitig zu den Aufgaben des Hauptprogramms im Hintergrund ablaufen, so kann diese auch mit Hilfe von Interrupts abgewickelt werden. Dadurch ist eine effektivere Datenübertragung möglich.

Hierbei muss allerdings immer bedacht werden, dass ein Interrupt zu jedem beliebigen Zeitpunkt auftreten kann. Es dürfen zum Beispiel vom Hauptprogramm oder von Unterprogrammen verwendete globale Variablen (z. B.: Tabellenzeiger **Z**) von der Interruptroutine nicht verändert werden!

USART-Sender mit Interrupts

Wenn der Sendepuffer leer ist wird ein Flag gesetzt (**UDRE**), das dann im Pollingbetrieb abgefragt werden kann. Man kann dem Controller allerdings auch erlauben bei dieser Aktion einen Interrupt auszulösen, indem man eine Eins in Bit 5 (**UDRIE**, *UDR Empty Interrupt Enable*) von **UCSRB** schreibt.

Damit das Interrupt ausgelöst wird müssen drei Bedingungen erfüllt sein:

1. Das **UDRE**-Interrupt muss erlaubt sein (**UDRIE** = 1).
2. Interrupts müssen global freigegeben sein (**I** in **SREG** = 1 mit **sei**)
3. Das Flag **UDRE** muss gesetzt sein (**UDRE** = 1).

Um den USART als Sender im Interruptbetrieb zu benutzen müssen außer der üblichen Initialisierung also noch folgende Schritte erledigt werden:

- Sprungadresse für den Interrupt organisieren (aus der Definitionsdatei: **UDREaddr = 0x001C**) und der Interruptroutine einen Namen zuweisen (z.B. **TXINT**).

```
;-----------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
;-----------------------------------------------------------------------------
.ORG    UDREaddr                ;Vektor Nr 15 (Adresse 0x1C) fuer TXINT
        rjmp TXINT

;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
```

- Der Stapel muss initialisiert werden!
- **UDRIE** im **UCSRB** setzen

```
        sbi     UCSRB,UDRIE     ;Interrupt UDRIE freigeben
```

- Interrupts global freigeben (sind bei Reset des Controllers per default gesperrt)

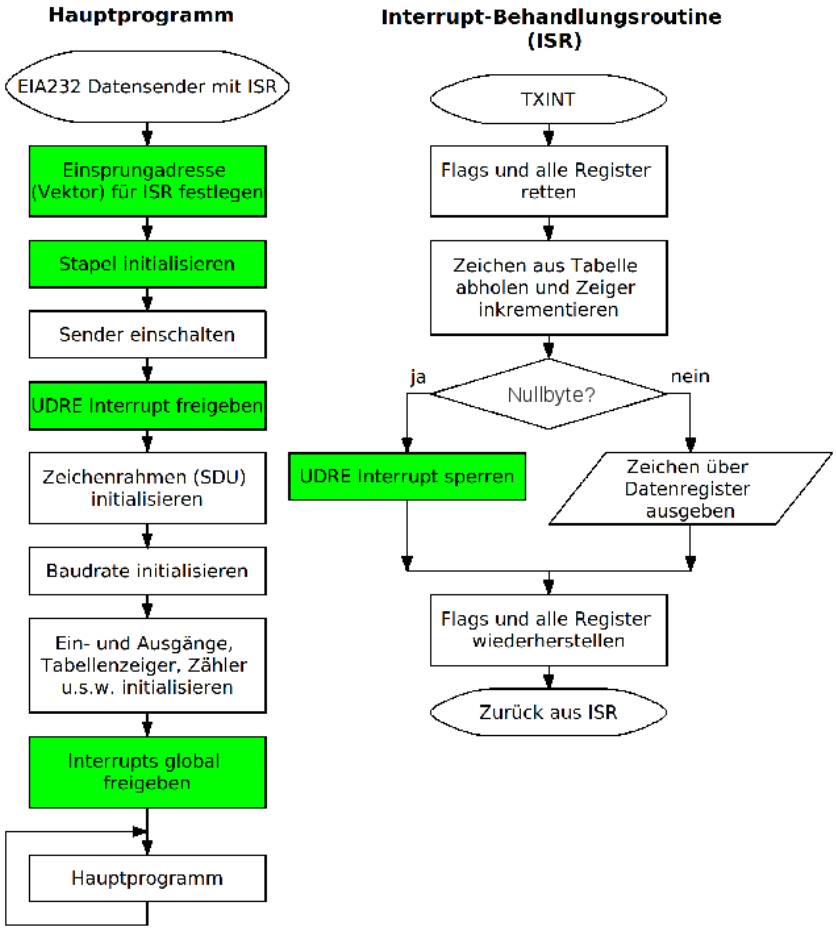
```
        ;Interrupt einschalten und freigeben
        sei                     ;Interrupts global freigeben (I-Bit im SREG)
```

- Interruptroutine schreiben.

Innerhalb der Routine müssen folgende Aufgaben erledigt werden:

- Die Flags (**SREG**-Register) und alle!, innerhalb der Routine benutzten Register, auf den Stapel retten.
- **UDRE** wird immer dann automatisch gelöscht wenn in das Datenregister **UDR** geschrieben wird (**out UDR,Tmp1**). Ist dies der Fall in der Interruptroutine, so ist nach Verlassen der Routine **UDRE** wieder Null und es wird kein neuer Interrupt ausgelöst bis **UDR** wieder leer ist. Wird allerdings in der Routine **UDR** nicht beschrieben, so muss die Routine das Interruptflag **UDRIE** in **UCSRB** löschen, da sonst immer weiter neue Interrupts ausgelöst werden!!

Flussdiagramm (Beispiel):



⌂ **C20A** a) Erstelle ein kommentiertes Assemblerprogramm mit dem Namen **"C20A_EIA232_sender_string_ISR_1.asm"** zum oberen Flussdiagramm.

Das Hauptprogramm tut nichts (Endlosschleife).
Es soll der Text "Hallo T3EC!!" einmalig mit nachfolgendem Wagenrücklauf ("carriage return", **CR**) aus einer Tabelle mit 38400 Baud (7N1) gesendet werden. Die Tabelle im Flash, schließt mit dem Nullbyte (**0x00**) ab. Die Ausgabe wird per Interruptroutine erledigt. Beobachte das Resultat mit einem Terminalprogramm.

Tipp: Der Adresszeiger **Z** dient als globale Variable. Er darf also nicht auf den Stapel gerettet werden!

b) Das Hauptprogramm soll nun Arbeit verrichten, und zwar soll es genau die gleiche Aufgabe erfüllen wie das Programm "**B202_numdisplay_4.asm**". Es zeigt als 16-Bit-Zähler seine Zustände im Sekundentakt auf dem 7-Segment display an. Die Interruptroutine kümmert sich weiter um die Textausgabe wie in Punkt b).
Speichere das Programm als **"C20A_EIA232_sender_string_ISR_2.asm"**

Tipp: Die Interruptroutine kann zu einem beliebigen Zeitpunkt aktiv werden!, also auch während des Display-Unterprogramms. Dieses arbeitet allerdings

ebenfalls mit dem Adresszeiger **Z** als globale Variable. Es ist daher nötig, dass die Interruptroutine den Adresszeiger **Z** auf den Stapel rettet und ihren eigenen Adresszeiger in sonst nicht benutzten Arbeitsregistern oder im SRAM zwischenspeichert und beim nächsten Aufruf wiederherstellt. Benutze z. B. die Arbeitsregister **r3** und **r4**.

USART-Empfänger mit Interrupts

Der Empfang von Daten mittels Interrupt erfolgt ähnlich wie beim Senden.

Wurde ein Zeichen vollständig empfangen, so wird das Bit **RXC** (*Receive Complete*) im **UCSRA** gesetzt. Sobald dieses Bit gesetzt ist, kann das Datenbyte mit "**in r16,UDR**" ausgelesen werden. Damit zusätzlich ein Interrupt ausgelöst werden kann, muss man eine Eins in Bit 7 (**RXCIE**, *RXC Interrupt Enable*) von **UCSRB** schreiben. Dies erlaubt das Auslösen eines Interrupts in dem Moment, wo das **RXC**-Flag (**UCSRA**) gesetzt wird, also neue Daten im Empfangspuffer vorhanden sind.

Damit das Interrupt ausgelöst wird müssen drei Bedingungen erfüllt sein:

1. Das Flag **RXC** muss gesetzt sein (**RXC** = 1).
2. Das **RXC**-Interrupt muss erlaubt sein (**RXCIE** = 1).
3. Interrupts müssen global freigegeben sein (**I** in **SREG** = 1 mit "**sei**")

Um den USART als Empfänger im Interruptbetrieb zu benutzen müssen außer der üblichen Initialisierung also noch folgende Schritte erledigt werden:

- Sprungadresse für den Interrupt organisieren (aus der Definitionsdatei: **URXCaddr = 0x001A**) und der Interruptroutine einen Namen zuweisen (z.B. **RXINT**).

```
;-------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
;-------------------------------------------------------------------------
.ORG    URXCaddr                ;Vektor Nr 14 (Adresse 0x1A) fuer RXINT
        rjmp RXINT
```

- Initialisierung des Stapels nicht vergessen!
- **RXCIE** im **UCSRB** setzen

```
        sbi     UCSRB,RXCIE     ;Interrupt RXCIE freigeben
```

- Interrupts global freigeben (sind bei Reset des Controllers per default gesperrt)

```
        ;Interrupt einschalten und freigeben
        sei                     ;Interrupts global freigeben (I-Bit im SREG)
```

- Interruptroutine schreiben.

Innerhalb der Routine müssen folgende Aufgaben erledigt werden:

- Die Flags (**SREG**-Register) und alle!, innerhalb der Routine benutzten Register (außer globale Variablen, die verändert werden sollen), auf den Stapel retten.
- **RXC** wird immer dann automatisch gelöscht, wenn das Datenregister **UDR** gelesen wird. Ist dies der Fall in der Interruptroutine, so ist nach Verlassen der Routine **RXC** wieder Null

und es wird kein neues Interrupt ausgelöst, bis ein neues Datenbyte angekommen ist. Wird allerdings in der Routine **UDR** nicht gelesen, so muss die Routine das Interruptflag **RXCIE** in **UCSRB** löschen, da sonst immer weiter neue Interrupts ausgelöst werden!

⌂ **C20B** Der Controller soll interruptgesteuert Daten vom PC empfangen und diese auch gleich als Echo wieder an den PC zurückschicken. Das Hauptprogramm soll währenddessen eine LED mit 5 Hz am Blinken halten. Der PC sendet mit 38400 Baud (7N1).

a) Zeichne die entsprechenden Flussdiagramme.
b) Erstelle ein kommentiertes Assemblerprogramm.
Gib ihm den Namen **"C20B_EIA232_receiver_char_ISR_1.asm".**
c) Beobachte das Resultat mit einem Terminalprogramm.

Weitere Aufgaben

⌂ **C20C** Zeichne ein Flussdiagramm und erstelle ein kommentiertes Assemblerprogramm mit dem Namen "**C20C_EIA232_coder_1.asm**" zur folgenden Aufgabenstellung:

Der PC sendet mit 9600 Baud (8N1) ASCII-Zeichen. Der Mikrocontroller codiert die Zeichen so, dass aus dem Buchstaben A ein Z wird, aus dem Buchstaben B ein Y usw.. Die Ziffern werden ebenfalls verdreht (0 wird 9 etc.). Alle Zeichen außer Ziffern, Groß- und Kleinbuchstaben werden ignoriert. Das codierte Zeichen wird dem PC augenblicklich nach der Codierung zurückgesendet. Es erfolgt eine Fehlerkontrolle der empfangenen Zeichen.

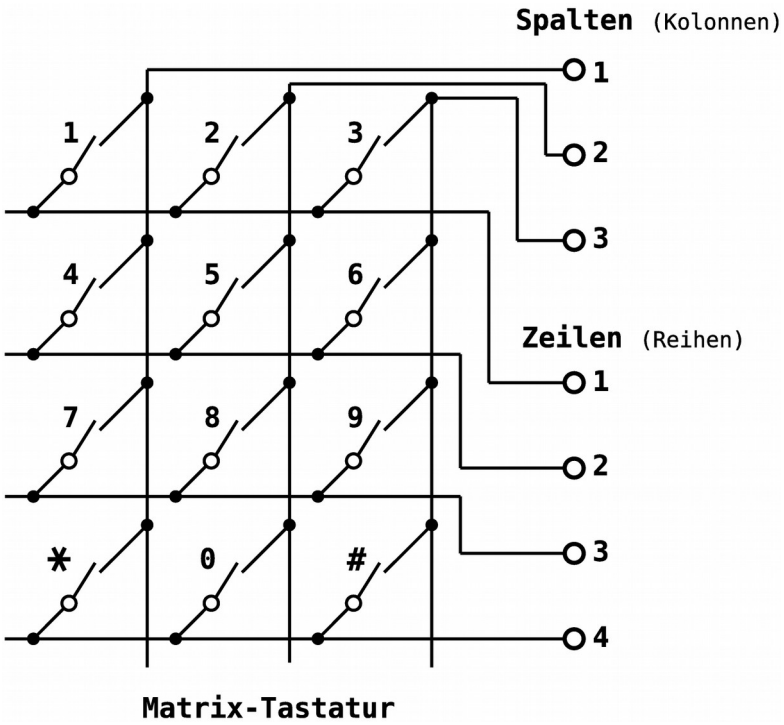
⌂ **C20D** Mit Hilfe einer Matrixtastatur soll es möglich sein 12 verschiedene Zeichen (Ziffern 0-9, * und #) an den PC zu senden (19200 Baud 7N1). Es existiert eine Unterprogrammbibliothek ("**SR_KEYPAD_3x4.asm**"), um die Tastatur abzufragen. Das Unterprogramm **KEYPAD** liefert mit dem Parameter **r24** eine Zahl von 0-12 zurück. Bei Null wurde keine Taste gedrückt. 1-9 entspricht den Ziffern 1-9, 10 entspricht dem Stern, 11 der Null und 12 der Raute. Das Programm soll dem PC den richtigen ASCII-Code zur betätigten Taste zusenden.

a) Untersuche das Unterprogramm **KEYPAD** indem du das Flussdiagramm erstellst.
b) Erstelle das Flussdiagramm und das kommentierte Programm zur obigen Aufgabe. Nenne das Programm "**C20D_EIA232_sender_keypad_1.asm**".

```
;***************************************************************************
;*
;*      Titel:  Unterprogramm zur 12-Tasten-Matrix-Tastatur (SR_3x4_KEYPAD.asm)
;*
;*      Datum:  23/12/07        Version:        0.3 (12/01/13)
;*      Autor:  WEIGU
;*
;*      Informationen zur Beschaltung:
;*      Prozessor:      ATmega32        Quarzfrequenz:  16MHz
;*      Eingaenge:      3 Spalten (columns) sind Eingaenge
;*      Ausgaenge:      4 Zeilen (rows) sind Ausgaenge am Port
;*
;*      Informationen zur Funktionsweise:
;*
```

```
;*      Eine Zeitschleifenbibliothek (UP W1ms) muss im Hauptprogramm eingebunden
;*      sein. Bei den Zuweisungen wird festgelegt, welche Ports und Pins
;*      verwendet werden sollen (3 Spalten (Eingang) 4 Zeilen (Ausgang).
;*      Es kann bei den Zuweisungen auch festgelegt werden wie viele
;*      Millisekunden entprellt werden soll (0-255).
;*      Das Unterprogramm KEYINI muss im Hauptprogramm in der Initialisierungs-
;*      phase aufgerufen werden (Initialisiert Portpins).
;*
;*      Beim Aufruf von KEYPAD wird der Tastenwert mit dem Parameter r24
;*      uebergeben (10 entspricht dem Stern, 11 der Null und 12 der Raute).
;*      Bei 0 wurde keine Taste gedrueckt!
;*
;*      Unterprogramme:
;*
;*      KEYINI          Initialisiert die Matrixtastatur (siehe Zuweisungen)
;*                      Muss im Hauptprogramm in der Initialisierungsphase
;*                      aufgerufen werden
;*      KEYPAD          Fragt die Matrixtastaur ab
;*                      Tastenwert in r24 (10 = *, 11 = 0, 12 = #)
;*                      Bei r24 = 0 wurde keine Taste gedrueckt!
;*
;*      Copyright (c) 2011   Guy WEILER         weigu[at]weigu[dot]lu
;*
;*      Hiermit wird unentgeltlich, jeder Person, die eine Kopie der Software
;*      und der zugehoerigen Dokumentationen (die "Software") erhaelt, die
;*      Erlaubnis erteilt, uneingeschraenkt zu benutzen, inklusive und ohne
;*      Ausnahme, dem Recht, sie zu verwenden, kopieren, aendern, fusionieren,
;*      verlegen, verbreiten, unterlizenzieren und/oder zu verkaufen, und
;*      Personen, die diese Software erhalten, diese Rechte zu geben, unter den
;*      folgenden Bedingungen:
;*
;*      ;Der obige Urheberrechtsvermerk und dieser Erlaubnisvermerk sind in alle
;*      ;Kopien oder Teilkopien der Software beizulegen.
;*      ;
;*      ;DIE SOFTWARE WIRD OHNE JEDE AUSDRUECKLICHE ODER IMPLIZIERTE GARANTIE
;*      ;BEREITGESTELLT, EINSCHLIESSLICH DER GARANTIE ZUR BENUTZUNG FUER DEN
;*      ;VORGESEHENEN ODER EINEM BESTIMMTEN ZWECK SOWIE JEGLICHER RECHTS-
;*      ;VERLETZUNG, JEDOCH NICHT DARAUF BESCHRAENKT. IN KEINEM FALL SIND DIE
;*      ;AUTOREN ODER COPYRIGHTINHABER FUER JEGLICHEN SCHADEN ODER SONSTIGE
;*      ;ANSPRUECHE HAFTBAR ZU MACHEN, OB INFOLGE DER ERFUELLUNG EINES
;*      ;VERTRAGES, EINES DELIKTES ODER ANDERS IM ZUSAMMENHANG MIT DER SOFTWARE
;*      ;ODER SONSTIGER VERWENDUNG DER SOFTWARE ENTSTANDEN.
;*
;*****************************************************************************

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       Zuweisungen
;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
.EQU    KeyPort = PORTA
.EQU    KeyPin  = PINA
.EQU    KeyDDR  = DDRA

.EQU    KP_C1  = 0      ;Tastatur 1 Spalte (147*) Eingang
.EQU    KP_C2  = 1      ;Tastatur 2 Spalte (2580) Eingang
.EQU    KP_C3  = 2      ;Tastatur 3 Spalte (369#) Eingang
.EQU    KP_R1  = 3      ;Tastatur 1. Zeile (123) Ausgang
.EQU    KP_R2  = 4      ;Tastatur 2. Zeile (456) Ausgang
.EQU    KP_R3  = 5      ;Tastatur 3. Zeile (789) Ausgang
.EQU    KP_R4  = 6      ;Tastatur 4. Zeile (*0#) Ausgang

.EQU    KeyDBC = 10     ;DebounceCounter * 1ms, gibt an wie viele Millisekunden
.                       ;entprellt werden soll (0-255!); kann auch null sein!

;----------------------------------------------------------------------------
;       Unterprogramm Keypad initialisieren
;----------------------------------------------------------------------------
KEYINI: cbi     KeyDDR,KP_C1    ;1 Spalte Eingang
        cbi     KeyDDR,KP_C2    ;2 Spalte Eingang
        cbi     KeyDDR,KP_C3    ;3 Spalte Eingang
        sbi     KeyPORT,KP_C1   ;1 Spalte Pull-Up
        sbi     KeyPORT,KP_C2   ;2 Spalte Pull-Up
```

```
        sbi     KeyPORT,KP_C3   ;3 Spalte Pull-Up
        sbi     KeyDDR,KP_R1    ;1 Zeile Ausgang
        sbi     KeyDDR,KP_R2    ;2 Zeile Ausgang
        sbi     KeyDDR,KP_R3    ;3 Zeile Ausgang
        sbi     KeyDDR,KP_R4    ;4 Zeile Ausgang
        ret

;-------------------------------------------------------------------------------
;       Unterprogramm Matrixtastatur 3*4
;       Tastenwert in r24 (10 = *, 11 = 0, 12 = #)
;       Bei r24 = 0 wurde keine Taste gedrueckt!
;-------------------------------------------------------------------------------
KEYPAD: push    r16             ;r16 retten

        ldi     r16,KeyDBC      ;r16 = DebounceCounter
KEYP1:  ldi     r24,12          ;Initialisiere Tastenzaehler r24 mit 12

        ;Ausgangszustand initialisieren (4. Zeile aktivieren)
        sbi     KeyPort,KP_R1   ;1. Zeile auf High schalten (inaktiv)
        sbi     KeyPort,KP_R2   ;2. Zeile auf High schalten (inaktiv)
        sbi     KeyPort,KP_R3   ;3. Zeile auf High schalten (inaktiv)
        cbi     KeyPort,KP_R4   ;4. Zeile aktivieren (Low)

        ;drei Spalten abfragen
KEYP2:  sbis    KeyPin,KP_C3    ;3. Spalte abfragen; Weiter bei High
        rjmp    KEYP5           ;(not pressed) sonst KEYP5
        dec     r24             ;erniedrige Tastenzaehler r24
        sbis    KeyPin,KP_C2    ;2. Spalte abfragen; Weiter bei High
        rjmp    KEYP5           ;(not pressed) sonst KEYP5
        dec     r24             ;erniedrige Tastenzaehler r24
        sbis    KeyPin,KP_C1    ;1. Spalte abfragen; Weiter bei High
        rjmp    KEYP5           ;(not pressed) sonst KEYP5
        dec     r24             ;erniedrige Tastenzaehler r24

        ;3. Zeile aktivieren
        sbis    KeyPin,KP_R3    ;Falls 3. Zeile nicht auf Low dann aktivieren
        rjmp    KEYP3           ;Falls schon auf Low dann weiter mit KEYP3
        cbi     KeyPort,KP_R3   ;3. Zeile auf Low
        rjmp    KEYP2           ;Spalten testen

        ;2. Zeile aktivieren
KEYP3:  sbis    KeyPin,KP_R2    ;Falls 2. Zeile nicht auf Low dann aktivieren
        rjmp    KEYP4           ;Falls schon auf Low dann weiter mit KEYP4
        cbi     KeyPort,KP_R2   ;2. Zeile auf Low
        rjmp    KEYP2           ;Spalten testen

        ;1. Zeile aktivieren
KEYP4:  sbis    KeyPin,KP_R1    ;Falls 1. Zeile nicht auf Low dann aktivieren
        rjmp    KEYP6           ;Falls schon auf Low dann fertig mit UP
        cbi     KeyPort,KP_R1   ;1. Zeile auf Low
        rjmp    KEYP2           ;Spalten testen

        ;Pruefen ob schon entprellt wurde
KEYP5:  tst     r16             ;Falls gleich Null, bereits entprellt UP beenden
        breq    KEYP6           ;
        rcall   W1ms            ;Falls nicht gleich, 1ms warten (entprellen)
        dec     r16             ;Debouncecounter dekrementieren
        rjmp    KEYP1           ;Tasten noch einmal abfragen

KEYP6:  pop     r16             ;r16 wiederherstellen
        ret                     ;fertig mit UP
```



Matrix-Tastatur

C3 A/D- und D/A-Wandler

Analog-Digital-Wandler[11] (*Analog-to-Digital Converter (ADC)*) setzen analoge Signale in digitale Daten um.

Digital-Analog-Wandler[12] (*Digital-to-Analog Converter (DAC)*) tun genau das Gegenteil.

Die Theorie zu A/D-Wandlern füllt ganze Bücher. Es soll hier nur auf die in den ATmega-Mikrocontrollern integrierten A/D-Wandler eingegangen werden.

A/D-Wandler

A/D-Wandler wandeln eine analoge Spannung in digitale Zahlenwerte um. Wichtige Kriterien sind hierbei die **Wandlungszeit**, die **Auflösung** in Bit und die **Genauigkeit**.

A/D-Wandler mit sehr hoher Auflösung benötigen viel Wandlungszeit. Ein solcher Wandler nach dem Dual-Slope-Verfahren (Zählverfahren) kann relativ einfach mit dem internen Komparator des ATmega32A und ein wenig externer Beschaltung aufgebaut werden.

Schnelle A/D-Wandler, welche Signale mit hohen Frequenzen wandeln können, haben eine relativ geringe Auflösung. Einen guten Kompromiss ermöglichen die in den AVR-Controllern eingesetzten A/D-Wandler, welche nach dem Wägeverfahren[13] arbeiten.

Die Genauigkeit eines A/D-Wandlers wird durch Quantisierungsfehler, Nullpunktfehler, Verstärkungsfehler, Nichtlinearität, Monotoniefehler und dynamische Fehler bestimmt. Im Datenblatt des ATmega32A findet man Angaben zur Genauigkeit des integrierten Wandlers. Durch ein eingebautes Abtast- und -Halte-Glied ist der dynamischen Fehler des integrierten Wandlers klein. Es können sogar Wandlungen, während sich der Controller in einem Schlafmodus befindet durchgeführt werden, um das eingestreute Rauschen zu minimieren. Wichtig für eine hohe Genauigkeit ist aber auch das Layout und die Beschaltung beim Anschluss an den A/D-Wandler (siehe Datenblatt).

Der im ATmega32A integrierte A/D-Wandler besitzt viele unterschiedliche Konfigurations-möglichkeiten und ist deshalb flexibel einsetzbar. Er eine **Auflösung von 10 Bit** und eine schnellste Wandlungszeit nach Datenblatt von 13 µs (bei geringerer Genauigkeit), womit Frequenzen von einigen kHz verarbeitet werden können. Ein **Multiplexer** am Eingang ermöglicht es 8 verschiedene Eingangsspannungen zu wandeln. Es können auch mit Hilfe von je zwei Eingängen **Differenzsignale** verarbeitet werden. Ein **Vorverstärker** kann die Eingangsspannung um den Faktor 10 verstärken (Faktor 200 möglich bei SMD-Bauformen und entsprechendem Layout). Alle Eingänge des A/D-Wandlers werden über die Pins von Port A eingelesen.

Als Referenzspannung kann die Betriebsspannung des Controllers verwendet werden, eine externe Referenzspannung oder eine interne Referenzquelle von 2,56 V. Mit einer 10 Bit-Auflösung und

11 A/D-Wandler, oder auch noch Analog-Digital-Umsetzer (ADU)

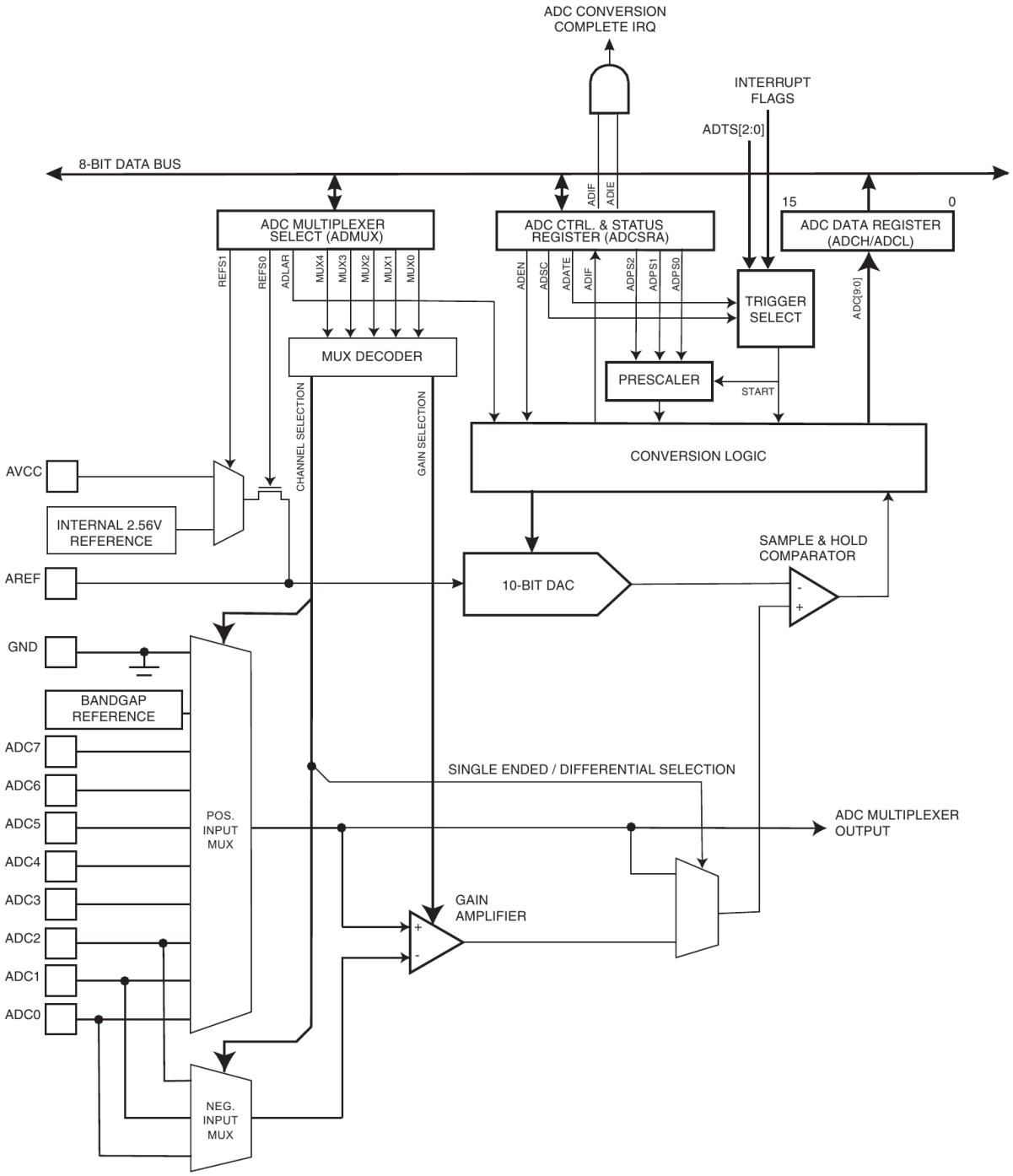
12 D/A-Wandler, oder auch noch Digital-Analog-Umsetzer (DAU)

13 Beim Wägeverfahren handelt es sich um ein Kompensationsverfahren. Die Annäherung an die Eingangsspannung erfolgt in gewichteten Schritten (Verfahren der sukzessiven Approximation).

einer Referenzspannung von zum Beispiel 5 V ist dann die feinste Auflösungsstufe 5 V / 2^{10} = 4,883 mV.

In der folgenden Abbildung aus dem Datenblatt des ATmega32A sieht man die vollständige A/D-Wandler-Baugruppe im Blockschaltbild.

Über die SF-Register **ADMUX**, **ADCSRA** und **SFIOR** wird der Wandler konfiguriert. Im 16-Bit Datenregistern **ADC** (**ADCL** und **ADCH**) wird das Ergebnis der Wandlung abgelegt.



Die Initialisierung des A/D-Wandlers

Zur **Initialisierung und Statusabfrage** des A/D-Wandler dienen die drei Register:

- **ADMUX** In diesem Register werden die Referenzspannungsquelle, die Anordnung der 10 Datenbit im Datenregister (links- oder rechtsbündig) und die zu verwendeten Eingangspins mit ihrer Betriebsart und Verstärkung festgelegt.
- **ADCSRA** Das Kontroll- und Statusregister schaltet auf Wunsch den Wandler, den ADC-Interrupt und den Auto Trigger ein, kann eine neue Wandlung einleiten, initialisiert den Vorteiler und enthält das Interrupt-Flag.
- **SFIOR** In diesem Spezialregister werden nur drei Bit für den A/D-Wandler benötigt. Sie entscheiden, welche Interrupt-Quelle den Auto-Trigger auslösen soll.

Daten stehen nach der Wandlung im Doppelregister:

- **ADC** (**ADCH**, **ADCL**) zur Verfügung.

Die Referenzspannungsquelle (ADMUX)

Es kann zwischen mittels zweier Bit (**REFS1**, **REFS0**) im Register **ADMUX** zwischen drei Referenzspannungsquellen gewählt werden:

1. Es wird die i**nterne Referenzspannungsquelle von 2,56 V** benutzt.
2. Es wird eine **externe Referenzspannung** am Pin **AREF** des Chips angelegt.
3. Die **Betriebsspannung des Wandlers** am Pin **AVCC** wird als Referenz benutzt.

Bemerkung: Beim MICES-Board sind **AVCC** und **AREF** mit der Betriebsspannung des Controllers verbunden. Für die folgenden Versuche wird die Spannung des Wandlers (5 V) am Pin **AVCC** als Referenz benutzt.

Anordnung im Datenregister (ADMUX)

Das Bit **ADLAR** (*ADC Left Adjust Result*) entscheidet über die Ausrichtung des 10-Bit-Ergebnisses der A/D-Wandlung.

Werden alle 10 Bit des Ergebnisses benötigt, so macht nur eine rechtsbündige Anordnung im 16-Bit-Datenregister **ADC** Sinn. Das Ergebnis ist eine 16-Bit-Dualzahl.

ADLAR = 0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADC	0	0	0	0	0	0	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0
	ADCH								ADCL							

Beim Einlesen des Resultats muss ADCL vor ADCH eingelesen werden!!

Wird keine so hohe Präzision benötigt oder soll die Wandlung mit möglichst hoher Geschwindigkeit erfolgen, so kann mit nur 8-Bit gewandelt werden. Die beiden niederwertigsten Bits werden verworfen. Dies vereinfacht auch die Verarbeitung der Daten, da dann nur mit dem 8-Bit-SF-Register **ADCH** gearbeitet wird.

Die Anordnung erfolgt linksbündig.

ADLAR = 1

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADC	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	0	0	0	0	0	0
	ADCH								ADCL							

Es wird nur **ADCH** eingelesen.

Der Kanalmultiplexer (ADMUX)

Bei eingeschalteter A/D-Wandler-Baugruppe sind die ausgewählten Pins des Port A keine digitalen Ein-bzw. Ausgangspins mehr. Es können nun analoge Signale eingelesen werden. Ein Multiplexer ermöglicht es aus den 8 Eingangspins auszuwählen.

Oft wird ein analoger Eingang unsymmetrisch betrieben. Die analoge Spannung liegt gegen die analoge Masse **AGND** des Chips an. Die Auswahl des Pins geschieht durch Einschreiben der dem Pin entsprechenden Dualzahl in die 5 **MUX**-Bits des **ADMUX**-Register (**MUX0** - **MUX4**). Soll zum Beispiel PIN **PA4** ausgewählt werden, so ist **0b00100** zu initialisieren (siehe Tabelle des **ADMUX**-Registers im Kapitel „Die SF-Register des A/D-Wandlers“).

Soll eine symmetrische Spannung eingelesen werden, so werden zwei Eingangspins benötigt. Die vielen Kombinationsmöglichkeiten des Anschlusses ($2^5 = 32$) zeigt die oben erwähnte Tabelle. Interessant ist auch die Möglichkeit den A/D-Wandler-Eingang zur Fehlersuche oder zur Kalibrierung mit einer Referenzspannungsquelle oder mit Masse zu verbinden. Das Messen einer Referenzspannung ermöglicht zum Beispiel die Bestimmung der Betriebsspannung des Chips. Besonders interessant ist dies bei mobilen Geräten mit Akku.

Der Vorteiler (ADCSRA)

Wie im großen Blockschaltbild ersichtlich benötigt der A/D-Wandler eine Taktfrequenz, welche von einem Vorteiler (*prescaler*) geliefert wird. Der Vorteiler teilt den Systemtakt und muss laut Datenblatt eine Taktfrequenz zwischen 50 kHz und 200 kHz liefern, damit der A/D-Wandler seine maximale Auflösung erreicht. Die Taktfrequenz darf allerdings auch höher sein, wenn nur mit 8 Bit gearbeitet wird. Der Vorteiler wird mit 3 Bit (**ADPS0** - **ADPS2**) im Kontrollregister **ADCSRA** eingestellt. Die beiden ersten Kombinationen liefern den gleichen Teilungsfaktor. Um zum Beispiel, mit einem 16 MHz Quarz bei voller Auflösung zu arbeiten, bleibt nur die Teilung durch 128 (16 MHz/128 = 125 kHz).

Das Taktsignal wird nur geliefert, wenn der A/D-Wandler-Baustein eingeschaltet ist.

ADPS2 ADPS1 ADPS0 2^2 2^1 2^0	Teilungs-faktor
000	2
001	2
010	4
011	8
100	16
101	32
110	64
111	128

⌂ **C300**

a) Eine normale Wandlung (*free running mode*) benötigt 13 Taktzyklen des A/D-Wandlertakts. Berechne die Wandlungszeit, falls bei 16 MHz der Vorteiler auf 128 eingestellt wird.

b) Im Datenblatt ist eine minimale Wandlungszeit von 13 µs angegeben. Mit welcher höchsten Taktfrequenz kann der Wandler noch arbeiten (wenn auch nicht mit voller Auflösung)?

c) Berechne die maximale Signalfrequenz bei einer A/D-Wandlungszeit von 13 µs nach Nyquist[14].

Weitere Einstellungen

Wie bei allen anderen Baugruppen ist der A/D-Wandler nach einem **RESET** abgeschaltet, um Strom zu sparen. Das Einschalten der Baugruppe geschieht mit dem Bit **ADEN** (**ADCSRA**).

Das Einleiten einer Wandlung geschieht entweder softwaremäßig durch das Setzen des Bit **ADSC** (*start conversion*) im Register **ADCSRA** oder aber hardwaremäßig durch einen Interrupt.

14 Das Nyquist-Shannonsche Abtasttheorem besagt, dass ein kontinuierliches, bandbegrenztes Signal, mit einer Minimalfrequenz von 0 Hz und einer Maximalfrequenz f_{max}, mit einer Frequenz größer als 2 f_{max} abgetastet werden muss, damit man aus dem so erhaltenen zeitdiskreten Signal das Ursprungssignal ohne Informationsverlust (aber mit unendlich großem Aufwand) exakt rekonstruieren und (mit endlichem Aufwand) beliebig genau approximieren kann (Quelle Wikipedia).

Polling

Beim **Polling** (ständige Abfrage durch die Software) kann das Bit **ADSC** oder das Flag **ADIF** benutzt werden, um das Ende der Wandlung festzustellen. **ADSC** bleibt während der Wandlung auf Eins und wird gleich nach der Wandlung durch die Hardware gelöscht, **ADIF** muss nach der Wandlung manuell durch Setzen einer Eins gelöscht werden.

Man unterscheidet den ***Single Conversion Mode***, bei dem jede einzelne Wandlung durch erneutes Schreiben des **ADSC**-Bits eingeleitet wird und den ***Free Running Mode*** (Freilaufmodus), bei dem jedes Ende einer Wandlung automatisch eine neue Wandlung auslöst.

Für den ***Free Running Mode*** muss der **Auto-Trigger** mit dem Bit **ADATE** (**ADCSRA**) eingeschaltet werden. Durch (einmaliges) Setzen des **ADSC**-Bits wird die erste Wandlung eingeleitet. Diese erste Wandlung dauert etwas länger (25 statt 13 A/D-Taktzyklen). Der Auto-Trigger leitet automatisch nach der ersten erfolgten Wandlung eine neue Wandlung ein, wenn eine positive Flanke einer Trigger-Quelle entdeckt wird[15]. Die Trigger-Quelle wird mit drei Bit (**ADTS0** - **ADTS2**) im SF-Register **SFIOR** festgelegt. Im *Free Running Mode* (**ADTS0** = **ADTS1** = **ADTS2** = 0) ist die Trigger-Quelle der A/D-Wandler selbst. Jedes Ende einer Wandlung triggert also eine erneute Wandlung.

Der **Auto-Trigger** funktioniert im *Free Running Mode* mit und ohne freigeschaltete Interrupts! Auch bei gesperrten Interrupts wird das Interrupt-Flag **ADIF** gesetzt. Um weitere Trigger zu ermöglichen, und damit Wandlungen auszulösen, muss das Flag dann aber softwaremäßig gelöscht werden.

Interrupts

Außer dem A/D-Wandler selbst kommen als Trigger-Quellen noch unterschiedliche Interrupt-Quellen in Frage. Besonders interessant ist das Auslösen eines Interrupts durch ein externes Signal (externer Interrupt) oder durch einen der Timer. Damit ist das Wandeln in festen Zeitabständen zum Beispiel fürs Datalogging möglich.

Das interruptgesteuerte Wandeln ermöglicht es den Mikrocontroller, während der Wandlungszeit des A/D-Wandlers, für andere Aufgaben zu nutzen und sollte bevorzugt verwendet werden.

15 Tritt diese Flanke während einer laufenden Wandlung auf, so wird sie wird sie ignoriert.

Beispiel für eine Initialisierung (Polling, Single Conversion Mode)

```
;ADC initialisieren
ldi     Tmp1,0b01100000 ;REFS = 01 AVCC als Referenzspannung
out     ADMUX,Tmp1      ;ADLAR = 1 linksbuendig (obere 8 Bit in ADCH)
                        ;MUX = 00000 unsymetrisch PA0 (ADC0)
ldi     Tmp1,0b10000111 ;ADEN = 1 (ADC einschalten)
out     ADCSRA,Tmp1     ;ADPS = 111 (16MHz/128 = 125kHz)
                        ;andere Bits per default = 0
```

Die SF-Register des A/D-Wandlers

Das ADMUX Register

Das **ADMUX**-Register befindet sich auf der SRAM-Adresse **0x0027** (SF-Register-Adresse **0x07**). Die Befehle **sbi**, **cbi**, **sbic** und **sbis** dürfen verwendet werden, da es sich um eines der unteren 32 Register handelt.

ADMUX = ADC Multiplexer Selection Register

Bit	7	6	5	4	3	2	1	0
ADMUX 0x07	**REFS1**	**REFS0**	**ADLAR**	**MUX4**	**MUX3**	**MUX2**	**MUX1**	**MUX0**
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

REFSn ***Reference Selection Bits*** ***REFS1, REFS0***

Mit diesen beiden Bit wird die **Quelle für die Referenzspannung ausgewählt** (3 Möglichkeiten). Änderungen an diesen Bits werden erst nach einer eventuell stattfindenden Wandlung berücksichtigt. Liegt eine externe Referenzspannung an Pin **AREF**, so soll nicht softwaremäßig auf interne Spannungsquelle geschaltet werden!

00 **Externe Referenzspannung am Pin AREF des Chip (zwischen 2 V und VCC, interne Spannungsquelle VREF abgeschaltet)**
01 **Die Spannung des Wandlers am Pin AVCC wird als Referenz benutzt*.**
10 **reserviert**
11 **Eine interne Referenzspannung VREF von 2,56 V wird benutzt*.**

***** Der offene Eingang **AREF** soll zur Störunterdrückung mit einem Kondensator (z.B. 100 nF) gegen Masse geschaltet werden. Weitere Informationen zur Störunterdrückung im Datenblatt unter "*Analog Noise Canceling Techniques*".

ADLAR ***ADC Left Adjust Result***

0 Das 10-Bit-Ergebnis der A/D-Wandlung wird **rechtsbündig** im Doppelregister ADC abgelegt (siehe **ADCL** und **ADCH**).
1 Das 10-Bit-Ergebnis der A/D-Wandlung wird **linksbündig** im Doppelregister ADC abgelegt (siehe **ADCL** und **ADCH**).

MUXn ***Analog Channel and Gain Selection Bits MUX0-MUX4***

Mit diesen fünf Bit wird der bzw. die **Eingangspins PAx** am Port A (Eingangskanal), die **Betriebsart** (unsymmetrisch (unipolar, Spannung gegen Masse) oder symmetrisch (differenziell, bipolar, Spannung zwischen zwei Pins)) und der **Verstärkungsfaktor**[16] nach der folgenden Tabelle **ausgewählt**.

16 Die 200-fache Verstärkung ist nicht für PDIP-Gehäuseformen getestet.

Änderungen an diesen Bits werden erst nach einer eventuell stattfindenden Wandlung berücksichtigt.

MUXn	UNSYMETRISCH	SYMETRISCH		
43 210 $2^4 2^3 2^2 2^1 2^0$	(Pin gegen Masse)	Positiver (nicht-invert.) Eingang	Negativer (invert.) Eingang	Verstärkung
00 000	PA0 (ADC0)			
00 001	PA1 (ADC1)			
00 010	PA2 (ADC2)			
00 011	PA3 (ADC3)			
00 100	PA4 (ADC4)			
00 101	PA5 (ADC5)			
00 110	PA6 (ADC6)			
00 111	PA7 (ADC7)			
01 000		PA0 (ADC0)	PA0 (ADC0)	10x
01 001		PA1 (ADC1)	PA0 (ADC0)	10x
01 010		PA0 (ADC0)	PA0 (ADC0)	200x
01 011		PA1 (ADC1)	PA0 (ADC0)	200x
01 100		PA2 (ADC2)	PA2 (ADC2)	10x
01 101		PA3 (ADC3)	PA2 (ADC2)	10x
01 110		PA2 (ADC2)	PA2 (ADC2)	200x
01 111		PA3 (ADC3)	PA2 (ADC2)	200x
10 000		PA0 (ADC0)	PA1 (ADC1)	1x
10 001		PA1 (ADC1)	PA1 (ADC1)	1x
10 010		PA2 (ADC2)	PA1 (ADC1)	1x
10 011		PA3 (ADC3)	PA1 (ADC1)	1x
10 100		PA4 (ADC4)	PA1 (ADC1)	1x
10 101		PA5 (ADC5)	PA1 (ADC1)	1x
10 110		PA6 (ADC6)	PA1 (ADC1)	1x
10 111		PA7 (ADC7)	PA1 (ADC1)	1x
11 000		PA0 (ADC0)	PA2 (ADC2)	1x
11 001		PA1 (ADC1)	PA2 (ADC2)	1x
11 010		PA2 (ADC2)	PA2 (ADC2)	1x
11 011		PA3 (ADC3)	PA2 (ADC2)	1x
11 100		PA4 (ADC4)	PA2 (ADC2)	1x
11 101		PA5 (ADC5)	PA2 (ADC2)	1x
11 110	1,22 V (Bandgap-Referenz V_{BG})			
11 111	0 V (GND)			

Das ADC Kontroll- und Statusregister A

Das **ADCSRA**-Register befindet sich auf der SRAM-Adresse **0x0026** (SF-Register-Adresse **0x06**). Die Befehle **sbi**, **cbi**, **sbic** und **sbis** dürfen verwendet werden, da es sich um eines der unteren 32 Register handelt.

ADCSRA = ADC Control and Status Register A

Bit	7	6	5	4	3	2	1	0
ADCSRA 0x06	**ADEN**	**ADSC**	**ADATE**	**ADIF**	**ADIE**	**ADPS2**	**ADPS1**	**ADPS0**
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

ADEN *ADC Enable*

0 schaltet den A/D-Wandler aus (Baugruppe verbraucht keinen Strom!). Geschieht dies während einer Wandlung, so wird diese abgebrochen.

1 **schaltet den A/D-Wandler ein**.

ADSC *ADC Start Conversion*

0 wird nach einer Wandlung automatisch wieder auf Null gesetzt. Das Löschen des Bit während der Wandlung hat keinen Einfluss.

1 **startet die A/D-Wandlung** und kann dazu benutzt werden festzustellen, ob momentan eine Wandlung durchgeführt wird.
Single Conversion Mode: Jede einzelne Wandlung wird durch erneutes Schreiben des Bits eingeleitet.
Free Running Mode: Durch (einmaliges) Setzen des Bit wird die erste Wandlung eingeleitet.
Die erste Wandlung dauert länger (25 statt 13 A/D-Taktzyklen), da ein zusätzlicher Umwandlungszyklus vorangestellt wird, der zur Initialisierung des Wandlers dient.
ADSC und **ADEN** können gleichzeitig gesetzt werden.

ADATE *ADC Auto Trigger Enable*

0 beendet den Auto-Trigger.

1 **startet den Auto-Trigger**.
Der Auto-Trigger kann auf eine von acht Interrupt-Quellen (Trigger-Quellen) reagieren. Der A/D-Wandler startet eine Wandlung, wenn eine **positive Flanke** der Interrupt-Quelle erkannt wird. Die Interrupt-Quelle wird mit drei Bit (**ADTS0** - **ADTS2**) im **SFIOR**-Register ausgewählt.

ADIF *ADC Interrupt Flag*

0 keine aktuellen Daten im Datenregister **ADC**

1 wird **Eins**, sobald der **Wandlungszyklus beendet** ist und die Daten im Doppelregister **ADC** (**ADCL**, **ADCH**) aktualisiert wurden. Gleichzeitig wird ein „*ADC Complete Interrupt*“ ausgelöst, wenn Interrupts global erlaubt sind (**I** im **SREG**) und das **ADIE**-Bit gesetzt wurde. Wurde ein Interrupt ausgelöst, so wird das Bit hardwaremäßig nach dem Ausführen des Interrupt gelöscht.
Im Polling-Betrieb kann das Bit manuell durch Schreiben einer Eins gelöscht werden!

ADIE *ADC Interrupt Enable*

0 der A/D-Wandler Interrupt ist gesperrt.

1 Das Setzen dieses Bit **ermöglicht das Auslösen eines Interrupts** in dem Moment, wo die A/D-Wandlung beendet ist und neue Daten anliegen (**ADI**-Flag im **ADCSRA**). Interrupts müssen dazu global frei gegeben sein (**I**=1 im **SREG** mit "**sei**").

ADPSn *ADC Prescaler Select Bits ADPS2, ADPS1, ADPS0*

Mit diesen drei Bit wird der Teilungsfaktor des Vorteilers ausgewählt. Der kleinste Teilungsfaktor ist 2.
Die Taktfrequenz des A/D-Wandlers soll bei voller Auflösung (10 Bit) zwischen 50 und 200 kHz liegen! Bei 8 Bit kann sie bis zu 1 MHz betragen. Sie errechnet sich mit:

$$Wandlungstakt = \frac{Systemtakt}{Teilungsfaktor}$$

ADPS2 ADPS1 ADPS0 2^2 2^1 2^0	Teilungs-faktor
000	2
001	2
010	4
011	8
100	16
101	32
110	64
111	128

Das Sonderfunktionsregister SFIOR

Das **SFIOR**-Register befindet sich auf der SRAM-Adresse **0x0050** (SF-Register-Adresse **0x30**). Die Befehle **sbi**, **cbi**, **sbic** und **sbis** dürfen **nicht verwendet** werden, da es sich um eines der oberen 32 Register handelt.

SFIOR = Special Function IO Register

Bit	7	6	5	4	3	2	1	0
SFIOR 0x30	**ADTS2**	**ADTS1**	**ADTS0**	**-**	**ACME**	**PUD**	**PSR2**	**PSR10**
Startwert	0	0	0	0	0	0	0	0
Read/Write	R/W	R/W	R/W	R	R/W	R/W	R/W	R/W

ADTSn ***ADC Auto Trigger Source*** ***ADTS2, ADTS1, ADTS0***

Falls der Auto Trigger Modus eingeschaltet wurde (**ADATE** in **ADCSRA**) werden diese drei Bit im **SFIOR**-Register benötigt, um die Trigger-Quelle auszuwählen. Eine A/D-Wandlung wird durch die steigende Flanke der ausgewählten Interrupt-Quelle ausgelöst.

ADTS2 ADTS1 ADTS0 2^2 2^1 2^0	Trigger-Quelle
000	Free Running Modus
001	Analoger Komparator
010	Externer Interrupt INT0
011	Timer 0 Compare
100	Timer 0 Overflow
101	Timer 1 Compare B
110	Timer 1 Overflow
111	Timer 1 Capture Event

Das 16-Bit ADC Datenregister

Die beiden 8 Bit-Register **ADCH** und **ADCL** befindet sich auf den SRAM-Adressen **0x0025** und **0x0024** (SF-Register-Adressen **0x05** und **0x04**). Die Befehle **sbi**, **cbi**, **sbic** und **sbis** dürfen verwendet werden.

ADCH = ADC Data Register High

Bit	7	6	5	4	3	2	1	0
ADCH 0x05	- ADC9	- ADC8	- ADC7	- ADC6	- ADC5	- ADC4	ADC9 ADC3	ADC8 ADC2
Startwert	0	0	0	0	0	0	0	0
Read/Write	R	R	R	R	R	R	R	R

ADCL = ADC Data Register Low

Bit	7	6	5	4	3	2	1	0
ADCL 0x04	ADC7 ADC1	ADC6 ADC0	ADC5 -	ADC4 -	ADC3 -	ADC2 -	ADC1 -	ADC0 -
Startwert	0	0	0	0	0	0	0	0
Read/Write	R	R	R	R	R	R	R	R

ADCn ***ADC Data Register Bit n***

Das **Resultat einer A/D-Wandlung** befindet sich in diesem 16-Bit Doppelregister. Werden differentielle (symmetrische) Eingänge benutzt, so liegt das Resultat in Zweierkomplementform vor!

ADLAR = 0: Ist das **ADLAR**-Bit im Register **ADMUX** gelöscht, so sind die 10 Bits rechtsbündig angeordnet (obere Zeile). Diese Einstellung ist günstig, wenn alle 10 Bit benötigt werden. Die Wertigkeit der Bits entspricht dann einer 16-Bit Dualzahl. **Das niederwertige Register muss zuerst gelesen werden und dann as hochwertige Register!** Erst nach dem Lesen des hochwertigen Register werden neue Daten in **ADC** abgelegt.

ADLAR = 1: Ist das **ADLAR**-Bit im Register **ADMUX** gesetzt, so sind die 10 Bits linksbündig angeordnet (untere Zeile). Diese Einstellung ist günstig, wenn die zwei niederwertigsten Bit vernachlässigt werden können, man also nur mit 8 Bit eiterarbeitet. Es reicht dann nur **ADCH** auszulesen.

ADLAR = 0

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADC	0	0	0	0	0	0	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0
							ADCH						ADCL			

ADLAR = 1

Bit	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ADC	ADC9	ADC8	ADC7	ADC6	ADC5	ADC4	ADC3	ADC2	ADC1	ADC0	0	0	0	0	0	0
	ADCH								ADCL							

A/D-Wandlung mittels Polling

Single Conversion Mode

Im ***Single Conversion Mode*** (Einzelwandlungsmodus) muss jede Wandlung manuell neu gestartet werden, da das Bit **ADSC** (starte Wandlung, **ADCSRA**) nach der Wandlung automatisch von der Steuerung wieder zurückgesetzt wird. Dieses Bit kann auch für das Polling zur Kontrolle, ob die Wandlung angeschlossen ist, verwendet werden.

Bemerkung: Eine Variante ist die Überprüfung des **ADIF**-Flag. Dieses Flag muss allerdings manuell, durch Schreiben einer Eins, zurückgesetzt werden.

Programmbeispiel:

Der ATmega32A-Controller soll eine Spannung zwischen 0 V und 5 V an **PA0** (**ADC0**) in eine 8-Bit-Zahl wandeln. **PA0** soll als unsymmetrischer Eingang geschaltet sein. Der Spannungswert soll binär über LEDs an Port C angezeigt werden.
Das Ende der Wandlungszeit wird mittels Polling ermittelt.

```
;***************************************************************************
;*
;*      Titel:  AD-Wandler Polling (single conversion) mit ADSC
;*              (C301_adc_single_conversion_1.asm)
;*      Datum:  15/11/10        Version:        0.2
;*      Autor:  WEIGU
;*
;*      Informationen zur Beschaltung:
;*
;*      Prozessor:      ATmega32A               Quarzfrequenz: 16MHz
;*      Eingaenge:      Analoge Spannung an PA0 (ADC0)
;*      Ausgaenge:      8 LEDs an PORTC
;*
;*      Informationen zur Funktionsweise:
;*      Die Spannung an PA0 (ADC0) wird eingelesen und als 8 Bit-Wert binaer am
;*      PORTC an 8 LEDs ausgegeben sobald die Wandlung abgeschlossen ist.
;*
;***************************************************************************

;---------------------------------------------------------------------------
;       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    Zero = r15              ;Register 1 wird zum Rechnen benoetigt
        clr     r15             ;und mit Null belegt
.DEF    Tmp1 = r16              ;Register 16 dient als erster Zwischenspeicher
.DEF    Tmp2 = r17              ;Register 17 dient als zweiter Zwischenspeicher
.DEF    Cnt1 = r18              ;Register 18 dient als Zaehler
```

```
.DEF    WL = r24                ;Register 24 und 25 dienen als universelles
.DEF    WH = r25                ;Doppelregister W und zur Parameteruebergabe

        ;ADC initialisieren
        ldi     Tmp1,0b01100000 ;REFS = 01 AVCC als Referenzspannung
        out     ADMUX,Tmp1      ;ADLAR = 1 linksbuendig (obere 8 Bit in ADCH)
                                ;MUX = 00000 unsymetrisch PA0 (ADC0)
        ldi     Tmp1,0b10000111 ;ADEN = 1 (ADC einschalten)
        out     ADCSRA,Tmp1     ;ADPS = 111 (16MHz/128 = 125kHz)
                                ;andere Bits per default = 0

        ;PORTC (8 Bit) als Ausgang
        ser     Tmp1            ;DDRC = 0b11111111
        out     DDRC,Tmp1

;-------------------------------------------------------------------------------
;       Hauptprogramm
;-------------------------------------------------------------------------------
MAIN:   sbi     ADCSRA,ADSC     ;eine AD-Wandlung ausloesen
LOOP:   sbic    ADCSRA,ADSC     ;Polling bis Wandlung fertig
        rjmp    LOOP
        in      Tmp1,ADCH       ;oberste 8 Bit einlesen und ausgeben
        out     PORTC,Tmp1
        rjmp    MAIN            ;Endlosschleife

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

⌂ **C301**

a) Zeichne das dem Programmbeispiel entsprechende Flussdiagramm.
b) Teste das Programm mit einer variablen Spannungsquelle (0-5 V).
Speichere das Programm als "**C301_adc_single_conversion_1.asm**".
c) Teste eine Variante mit der Abfrage des **ADIF**-Flag.
Speichere das Programm als "**C301_adc_single_conversion_2.asm**".

Free Running Mode

Der ***Free Running Mode*** unterscheidet sich zum einfachen Polling mit **ADIF** dadurch, dass die Wandlung nur einmalig eingeleitet wird. Das Ende einer Wandlung startet automatisch die nächste Wandlung. Dadurch wird um ein Takt schneller gewandelt, da der Vorteiler nicht zurückgesetzt werden muss (siehe Zeitdiagramme im Datenblatt).

Bei der Initialisierung muss das **ADATE**-Bit (*ADc Auto Trigger Enable*) gesetzt werden und der ***Free Running Mode*** im **SFIOR** Register gewählt werden.

Programmbeispiel (Auszug):

```
        ;ADC initialisieren
        ldi     Tmp1,0b01100000 ;REFS = 01 AVCC als Referenzspannung
        out     ADMUX,Tmp1      ;ADLAR = 1 linksb. (obere 8 Bit in ADCH)
                                ;MUX = 00000 unsymetrisch PA0 (ADC0)
        ldi     Tmp1,0b10100111 ;ADEN = 1 (ADC einschalten)
        out     ADCSRA,Tmp1     ;ADATE = 1 (Auto Trigger Modus ein)
                                ;ADPS = 111 (16MHz/128 = 125kHz)
                                ;andere Bits per default = 0
        in      Tmp1,SFIOR      ;ADTS = 000 free running mode
        andi    Tmp1,0b00011111
        out     SFIOR,Tmp1

        ;PORTC (8 Bit) als Ausgang
        ser     Tmp1            ;DDRC = 0b11111111
        out     DDRC,Tmp1
```

```
;-------------------------------------------------------------------------------
;       Hauptprogramm
;-------------------------------------------------------------------------------
MAIN:   sbi     ADCSRA,ADSC     ;AD-Wandlung ausloesen fuer free running mode
                                ;einmaliger Vorgang
LOOP:   sbis    ADCSRA,ADIF     ;Polling ob Wandlung fertig
        rjmp    LOOP
        in      Tmp1,ADCH       ;oberste 8 Bit einlesen und ausgeben
        sbi     ADCSRA,ADIF     ;ADC Interrupt Flag manuell loeschen (mit 1!)
        out     PORTC,Tmp1
        rjmp    LOOP            ;Endlosschleife
```

⌂ **C302**

a) Teste das obige Programm im ***Free Running Mode***.
Speichere das Programm als "**C302_adc_free_running_1.asm**".

b) Es soll mit Hilfe des Oszilloskop die Wandlungszeit bei allen drei Programmen ermittelt werden. Dazu wird die analoge Eingangsspannung auf Minimum (0 V) eingestellt und das Oszilloskop an Pin **PC0** angeschlossen. Hinter der Ausgabe in den Programmen wird folgende Zeile eingefügt: **sbi PORTC,0**. Bestimme jetzt aus dem Abstand der Impulse die Wandlungszeiten. Notiere und kommentiere die Ergebnisse.

A/D-Wandlung mittels Interrupt (Auto Trigger)

Bei eingeschaltetem ***Auto Trigger***, kann eine von acht Interrupt-Quellen (Trigger-Quellen) eine Wandlung auslösen. Der A/D-Wandler startet eine Wandlung, wenn eine positive Flanke der Interrupt-Quelle erkannt wird. Die Interrupt-Quelle wird mit drei Bit (**ADTS0** - **ADTS2**) im **SFIOR**-Register ausgewählt.

Zusätzlich zu den Initialisierungen beim Polling muss der Interruptvektor initialisiert werden (**ADCCaddr**), der Auto Trigger Modus muss eingeschaltet werden (**ADATE** im SF-Register **ADCSRA**) und die Interrupt-Quelle muss im **SFIOR** ausgewählt werden (**ADTS0** - **ADTS2**).
Der A/D-Wandler-Interrupt muss erlaubt sein (**ADIE** im SF-Register **ADCSRA**) und Interrupts müssen global zugelassen sein (**I** im Statusregister **SREG**).

Free Running Mode

Ein Spezialfall ist der **Free Running Mode**. Hierbei wird kontinuierlich gewandelt. Wurde das Resultat der Wandlung im Datenregister abgelegt, so wird gleich die nächste Wandlung gestartet. Dies geschieht durch die positive Flanke des Interrupt des A/D-Wandlers selbst. Die Wandlung muss nur einmalig mit dem Bit **ADSC** gestartet werden.

Der **Free Running Mode** wird im **SFIOR** aktiviert mit **ADTS = 0b000**.

Die gleiche Aufgabe wie oben soll nun im Free Running-Mode mit Hilfe eines Interrupts programmiert werden.

Programmbeispiel (Auszug):

```
;-------------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
;-------------------------------------------------------------------------------
        ;Vektortabelle (im Flash-Speicher)
.ORG    ADCCaddr                ;interner Vektor fuer ADCC (alt.: .ORG 0x0020)
                                ;A/D-Wandlung vollstaendig ADC complete
```

```
        rjmp    ISR_AD          ;Springe zur ISR von ADCC

;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Zero = r15              ;Register 1 wird zum Rechnen benoetigt
        clr     r15             ;und mit Null belegt
.DEF    Tmp1 = r16              ;Register 16 dient als erster Zwischenspeicher
.DEF    Tmp2 = r17              ;Register 17 dient als zweiter Zwischenspeicher
.DEF    Cnt1 = r18              ;Register 18 dient als Zaehler
.DEF    WL = r24                ;Register 24 und 25 dienen als universelles
.DEF    WH = r25                ;Doppelregister W und zur Parameteruebergabe
...
        ;Stapel initialisieren (fuer Unterprogramme bzw. Interrupts)
        ldi     Tmp1,HIGH(RAMEND)       ;RAMEND (SRAM) ist in der Definitions-
        out     SPH,Tmp1                ;datei festgelegt
        ldi     Tmp1,LOW(RAMEND)
        out     SPL,Tmp1

        ;ADC initialisieren
        ldi     Tmp1,0b01100000 ;REFS = 01 AVCC als Referenzspannung
        out     ADMUX,Tmp1      ;ADLAR = 1 linksb. (obere 8 Bit in ADCH)
                                ;MUX = 00000 unsymetrisch PA0 (ADC0)
        ldi     Tmp1,0b10101111 ;ADEN = 1 (ADC einschalten)
        out     ADCSRA,Tmp1     ;ADATE = 1 (Auto Trigger Modus ein)
                                ;ADIE = 1 (ADC interrupt enable)
                                ;ADPS = 111 (16MHz/128 = 125kHz)
                                ;andere Bits per default = 0
        in      Tmp1,SFIOR      ;ADTS = 000 free running mode
        andi    Tmp1,0b00011111
        out     SFIOR,Tmp1

        ;PORTC (8 Bit) als Ausgang
        ser     Tmp1            ;DDRC = 0b11111111
        out     DDRC,Tmp1

        ;A/D-Wandlung ausloesen (einmalig) Interrupts erlauben
        sbi     ADCSRA,ADSC     ;A/D-Wandlung ausloesen fuer free running mode
        sei                     ;globales Interrupt-Flag setzen (Int. erlauben)

;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:   rjmp      MAIN

;-----------------------------------------------------------------------------
;       Unterprogramme und Interrupt-Behandlungsroutinen
;-----------------------------------------------------------------------------
; Interrupt-Behandlungsroutine ADC
ISR_AD: push    r16             ;benutzte Reg. retten (r16 = Zwischenspeicher)
        in      r16,SREG        ;Statusregister einlesen
        push    r16             ;Statusregister retten

        in      r16,ADCH        ;oberste 8 Bit einlesen und ausgeben
        out     PORTC,r16
        rcall   W1s             ;Anzeige 1* pro Sekunde

        pop     r16             ;Werte der geretteten Register wieder-
        out     SREG,r16        ;herstellen
        pop     r16
        reti                    ;Ruecksprung ins Hauptprogramm aus einer
                                ;Interrupt-Behandlungsroutine
```

⌂ **C303** a) Zeichne das dem Programmbeispiel entsprechende Flussdiagramm.
b) Teste das Programm mit einer variablen Spannungsquelle (0-5 V).
Speichere das Programm als "**C303_adc_int_free_running_1.asm**".

Wandlung auslösen mit dem externen Interrupt 0

Bei diesem Fall handelt es sich auch um einen Spezialfall, da gegenüber den anderen sechs Fällen hier kein echter zweiter Interrupt generiert werden muss. Bereits die Initialisierung des SF-Register **MCUCR** bewirkt, dass das Interrupt-Flag (**INTF0**, **INTF1** oder **INTF2**) beim Auftreten einer Flanke am entsprechenden Pin gesetzt wird. Das externe Interrupt-Flag löst dann eine ADC-Wandlung aus.

Wurde der ADC-Interrupt erlaubt, wird dann nach der Wandlung ein ADC-Interrupt ausgelöst.

!! Zum Nutzen des externen Interrupt 0 als Triggerquelle muss nur im SF-Register **MCUCR** festgelegt werden mit welcher Flanke gearbeitet wird!

Die Triggerquelle **INT0** wird im **SFIOR** aktiviert mit **ADTS = 0b010**.

Da hier kein realer externer Interrupt auftritt, wird auch das Flag **INTF0** nicht automatisch gelöscht. Dies muss manuell durch Schreiben einer Eins in der ADC-ISR geschehen.

Eine steigende Flanke am Pin **INT0** (**PD2**) soll eine Wandlung auslösen. Die Spannung am unsymmetrischen Eingang **PA0** wird als 8 Bit-Wert über Port C ausgegeben.

Programmbeispiel (Auszug):

```
;-----------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
;-----------------------------------------------------------------------------
        ;Vektortabelle (im Flash-Speicher)
.ORG    ADCCaddr                ;interner Vektor fuer ADCC (alt.: .ORG 0x0020)
                                ;AD-Wandlung vollstaendig ADC complete
        rjmp    ISR_AD          ;Springe zur ISR von ADCC

;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:
.DEF    Zero = r15              ;Register 1 wird zum Rechnen benoetigt
        clr     r15             ;und mit Null belegt
.DEF    Tmp1 = r16              ;Register 16 dient als erster Zwischenspeicher
.DEF    Tmp2 = r17              ;Register 17 dient als zweiter Zwischenspeicher
.DEF    Cnt1 = r18              ;Register 18 dient als Zaehler
.DEF    WL = r24                ;Register 24 und 25 dienen als universelles
.DEF    WH = r25                ;Doppelregister W und zur Parameteruebergabe

        ;Stapel initialisieren (fuer Unterprogramme bzw. Interrupts)
        ldi     Tmp1,LOW(RAMEND)        ;RAMEND (SRAM) ist in der Definitions-
        out     SPL,Tmp1                ;datei festgelegt
        ldi     Tmp1,HIGH(RAMEND)
        out     SPH,Tmp1

        ;ADC initialisieren
        ldi     Tmp1,0b01100000 ;REFS = 01 AVCC als Referenzspannung
        out     ADMUX,Tmp1      ;ADLAR = 1 linksbuendig (obere 8 Bit in ADCH)
                                ;MUX = 00000 unsymetrisch PA0 (ADC0)
        ldi     Tmp1,0b10101111 ;ADEN = 1 (ADC einschalten)
        out     ADCSRA,Tmp1     ;ADATE = 1 (Auto Trigger Modus aktivieren)
                                ;ADIE = 1       (ADC interrupt enable)
                                ;ADPS = 111 (16MHz/128 = 125kHz)
                                ;andere Bits per default = 0
        in      Tmp1,SFIOR      ;ADTS = 010 external interrupt request 0
        andi    Tmp1,0b01011111
        ori     Tmp1,0b01000000
```

```
        out     SFIOR,Tmp1

        ;INT0 steigende Flanke initialisieren
        ;(INT0 muss nicht gesondert ueber GICR aktiviert werden!!)
        in      Tmp1,MCUCR      ;steigende Flanke von INT0 initialisieren
        ori     Tmp1,0b00000011
        out     MCUCR,Tmp1

        ;PORTC (8 Bit) als Ausgang
        ser     Tmp1            ;DDRC = 0b11111111
        out     DDRC,Tmp1

         ;Interrupts global erlauben
         sei                     ;globales Interrupt-Flag setzen (Int. erlauben)

;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:   rjmp    MAIN            ;Endlosschleife

;-----------------------------------------------------------------------------
;       Unterprogramme und Interrupt-Behandlungsroutinen
;-----------------------------------------------------------------------------
; Interrupt-Behandlungsroutine ADC
ISR_AD: push    r16             ;benutzte Reg. retten (r16 = Zwischenspeicher)
        in      r16,SREG        ;Statusregister einlesen
        push    r16             ;Statusregister retten

        in      r16,ADCH        ;oberste 8 Bit einlesen und ausgeben
        out     PORTC,r16

        in      r16,GIFR        ;INTF0-Bit im GIFR manuell loeschen, (mit 1!)
        ori     r16,0b01000000 ;damit erneutes Interrupt erkannt wird!
        out     GIFR,r16

        pop     r16             ;Werte der geretteten Register wieder-
        out     SREG,r16        ;herstellen
        pop     r16
        reti                    ;Ruecksprung ins Hauptprogramm aus einer
                                ;Interrupt-Behandlungsroutine
```

⌂ **C304** a) Zeichne das dem Programmbeispiel entsprechende Flussdiagramm.
b) Teste das Programm mit einer variablen Spannungsquelle (0-5 V) .
Speichere das Programm als "**C304_adc_int_int0.asm**".

Wandlung auslösen mit dem Timer 0

Hier soll als Beispiel noch das Auslösen eines Interrupt im Millisekundenrythmus mit dem Timer 0 demonstriert werden. Auf dieses Beispiel kann man zurückkommen, wenn man das Kapitel zum Timer durchgearbeitet hat.

Es werden dazu Timer 0 und Timer 2 im CTC-Modus verwendet. Mit einem Quarz von 16 MHz und einem Teiler von 64 benötigt man 250 Zählschritte um eine Interruptfrequenz von 1000 Hz (t = 1 ms) zu erhalten.

Weitere Aufgaben

⌂ **C305** Erweitere das erste Programm um eine Dual-BCD Wandlung. Dazu kann das unten stehende Unterprogramm zum Teilen durch 100 und durch 10 verwendet werden. Der Spannungswert soll als Vielfaches von 10 mV an den drei untersten Stellen des

Siebensegment-Displays ausgegeben werden.
Der Einfachheit halber wird der gewandelte 8-Bit-Wert (0-255) mit Zwei multipliziert (Befehl **lsl**). Der dabei gemachte Fehler (0-510 statt 0-500) liegt nur bei zwei Prozent.
Teste das Programm mit einer variablen Spannungsquelle (0-5 V). Speichere das Programm als "**C305_adc_voltmeter.asm**".

```
;-----------------------------------------------------------------------------
;       Ganzzahldivision 2Byte / 1Byte
;-----------------------------------------------------------------------------

;Dividend:       r1,r0 (LSB)
;Divisor         r2
;Schleifenzähler r16
;Resultat Rest in r1 + Quotient in r0
;Achtung Ueberlauf wenn obere n Bit des Dividenden groesser gleich n Bit Divisor

DIV21:  push    r16
        in      r16,SREG
        push    r16

        ldi     r16,8           ;Schleifenzaehler init. (Bitzahl Quotient)

DIV21A: lsl     r0              ;Rotiere Dividend 1 Stelle nach links
        rol     r1              ;rechts wird Null reingeschoben
        brcs    DIV21B          ;Falls Carry muss Subtraktion erfolgen!
        cp      r1,r2           ;Subtraktion testen
        brlo    DIV21C          ;Ueberspringe Subtraktion, wenn kleiner

DIV21B: sub     r1,r2           ;Subtraktion durchfuehren
        inc     r0              ;Bit 0 im Dividenden setzen

DIV21C: dec     r16             ;Schleifenzaehler dekrementieren
        brne    DIV21A

        pop     r16
        out     SREG,r16
        pop     r16
        ret
```

Bemerkung: Zum Erhöhen der Genauigkeit kann noch die oberste Stelle (0-5) mit 2 multipliziert werden (Befehl **lsl**) und dann vom gesamten Wert subtrahiert werden.

⌂ **C306** Konfiguriere und programmiere den ATmega32A so, dass sein A/D-Wandler im *Free Running Mode* mit Interrupt arbeitet. Das Assemblerprogramm soll so aufgebaut sein, dass die Spannung an **PA0** (ADC0) gemessen und ihre Höhe über eine Bargraph-Anzeige (Balkenanzeige) ausgegeben wird. Die Bargraph-Anzeige soll mit Hilfe von den 8 LEDs an Port C gebildet werden. Dazu ist es notwendig die Spannung in einen 3 Bit-Wert umzuwandeln (einfach die hochwertigsten drei Bits verwenden!). Benutze zur Programmierung der Bargraph-Anzeige eine Tabelle.
Schließe an **PA0** ein Potentiometer an und überprüfe die Funktionsweise deines Programms. Bei minimaler Spannung (linker Anschlag) darf nur eine LED leuchten, bei maximaler Spannung (rechter Anschlag) sollen alle LEDs leuchten. Speichere das Programm als "**C306_adc_bargraph.asm**".

⌂ **C307** Entwickle für den ATmega32A ein Assemblerprogramm, das an **PA2** (ADC2) die Spannung misst, wenn an **INT0** eine fallende Signalflanke erscheint. Der umgewandelte Spannungswert (10 Bit) soll mit Hilfe eines Unterprogramms in ein String (mit ASCII-Zeichen) umgewandelt werden und dann über die serielle

Schnittstelle mit 19200 Baud (8N1) zu einem PC gesendet werden. Das Unterprogramms mit den symbolischen Namen **B16_STR** wandelt eine 16 Bit-Zahl in einen String mit einer festen Länge von 5 ASCII-Zeichen (00000-65536). Speichere das Programm als "**C307_adc_to_EIA232.asm**".

⌂ **C308** Ähnliche Aufgabe wie die vorige Aufgabe, nun soll der String allerdings in mittels Interrupt über die serielle Schnittstelle versendet werden. Speichere das Programm als "**C308_adc_to_EIA232_2.asm**".

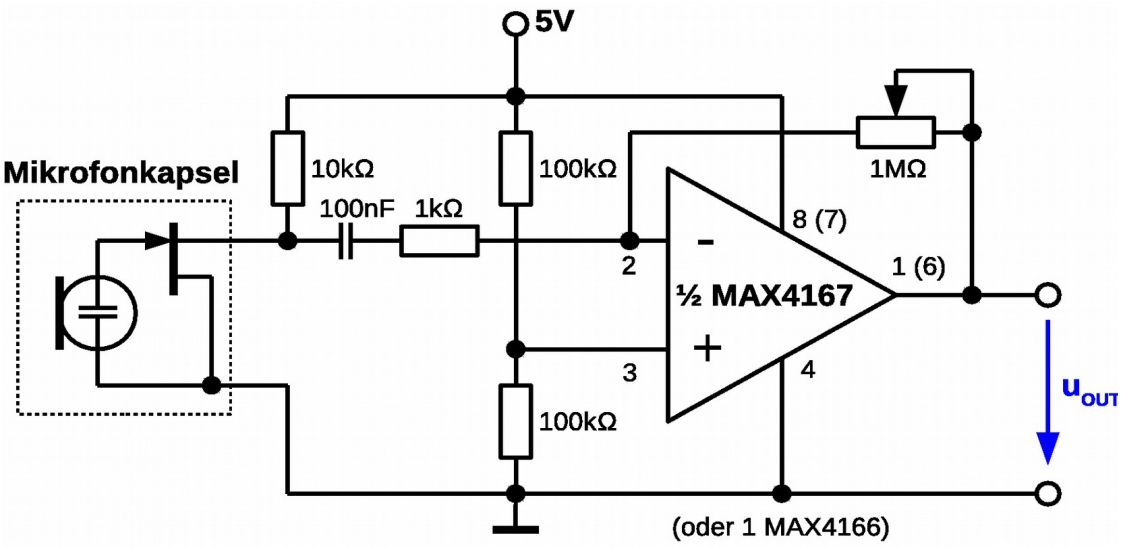
⌂ **C309** Im Single Conversion Mode soll abwechselnd die Spannung an **PA0** (ADC0) und an **PA1** (ADC1) in einen 8 Bit-Wert umgewandelt und über Port C (für ADC0) bzw. Port B (für ADC1) ausgegeben werden. Entwickle das hierzu notwendige Assemblerprogramm für den ATmega32A. Speichere das Programm als "**C309_adc_dual_conversion.asm**".

⌂ **C30A**

a) Baue das folgende Modul um Sprache zu digitalisieren. Das Signal eines Elektret-Kondensatormikrofons[17] wird mittels eines OPV als Differenzverstärker in eine Spannung zwischen 0 und 5 V umgesetzt. Es wird ein sogenannter *rail-to-rail* OPV verwendet, der den gesamten Bereich seiner Versorgungsspannung nutzen kann.

b) Beschreibe ausführlich die Funktionsweise der Schaltung (nutze die Formeln des OPV als Differenzverstärker!)

c) Teste die Schaltung ausgiebig mit den obigen Programmen.



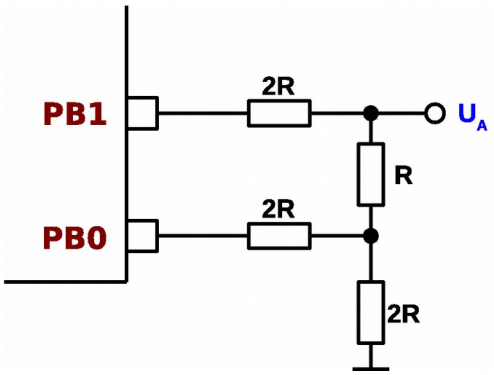
17 Das Elektretmikrofon ist ein Kondensatormikrofon mit einer Elektretfolie. Meist befindet sich in der Mikrofonkapsel noch ein Mikrofonverstärker (FET). Zum Betrieb reicht eine Spannung von ungefähr 1,5 Volt bei 1 mA. Das Elektretmikrofon hat Kugelcharakteristik und deckt den Frequenzbereich von 20 Hz bis 20 kHz ab. Wegen seiner kompakten Bauweise wird er praktisch in allen mobilen Geräten eingesetzt.

D/A-Wandler

Damit digitale Informationen mittels D/A-Wandlern umgesetzt werden können, müssen sie in einem gewichteten Code vorliegen. Der Dualcode (binäres Zahlensystem) ist ein solch gewichteter Code (2^0 = 1, 2^1 = 2, 2^2= 4, 2^3 = 8 ...). Umso mehr Bits verwendet werden umso höher ist die Auflösung des D/A-Wandlers. Mit 8 Bits kann ein analoges Signal mit 256 unterschiedlichen Spannungswerten erzeugt werden. Umso höher die Auflösung, umso kleiner wird die Treppenstufung des analogen Signals.

Das R-2R-Netzwerk

Ein D/A-Wandler lässt sich sehr einfach mit einem sogenannten R-2R-Netzwerk realisieren. Ein solches Netzwerk besitzt nur Widerstände mit den Werten R und 2R. Im folgenden soll ein solches Netzwerk für zwei Stellen des Dualcodes untersucht werden:



1 Fall:

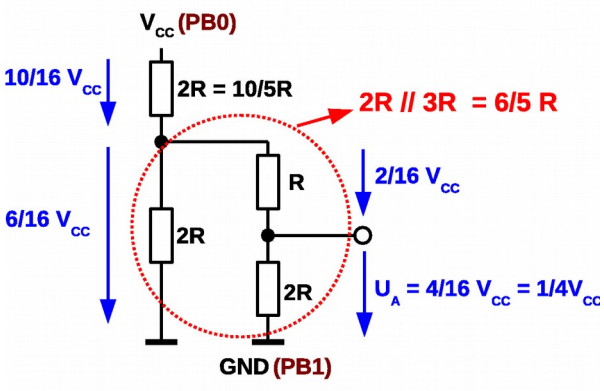
PB1 = 0 und **PB0 = 0**

Die Ausgangsspannung wird in diesem ersten Fall 0 V sein, da keine Spannung anliegt.

2 Fall:

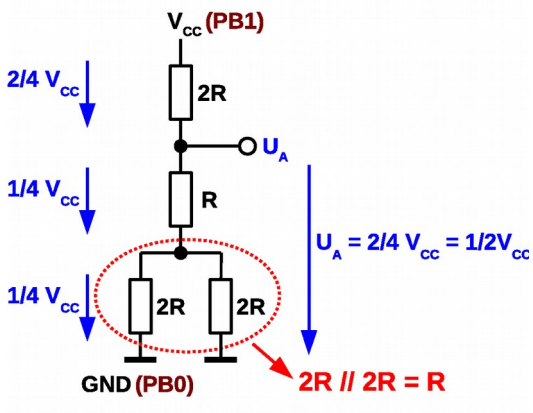
PB1 = 0 und **PB0 = 1**

An **PB0** = 1 liegt die Betriebsspannung des Controllers an. Es ergibt sich folgende Schaltung:



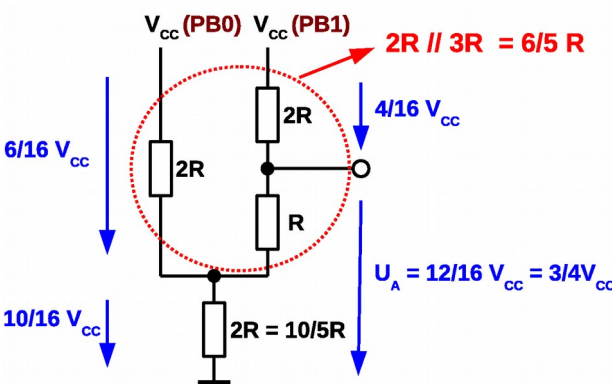
Die Parallelschaltung von 3R (R+2R) mit 2R ergibt einen Gesamtwiderstand von 6/5 R. Auf diesen fallen in der Reihenschaltung mit 2R = 10/5 R also 6 Anteile der Spannung von 16. Davon tragen dann 2/3 zur Ausgangsspannung bei. Die Ausgangsspannung beträgt ¼ der Gesamtspannung.

3 Fall:
PB1 = 1 und **PB0 = 0**



Die Spannung wird geteilt und die Hälfte der Betriebsspannung liegt am Ausgang.

4 Fall:
PB1 = 1 und **PB0 = 1**



Die Spannung teilt sich ähnlich wie im 2. Fall auf. Es liegen dann ¾ der Betriebsspannung am Ausgang an.

Damit ergibt sich folgende Tabelle:

Fall	Dezimalwert	PB1 (2^1)	PB0 (2^0)	U_A
1	0	0	0	0
2	1	0	1	¼ V_{CC}
3	2	1	0	½ V_{CC}
4	3	1	1	¾ V_{CC}

Jedes Bit trägt seinen Teil zur resultierenden Ausgangsspannung bei! Die höchste Stelle PB1 (2^1) steuert die Hälfte der Betriebsspannung bei. Die niedrigere Stelle ein Viertel.

Erweitert man das Netzwerk zum Beispiel auf drei Bit, so ergibt sich folgende Tabelle:

Fall	Dezimalwert	PB2 (2^2)	PB1 (2^1)	PB0 (2^0)	U_A
1	0	0	0	0	0
2	1	0	0	1	1/8 V_{CC}
3	2	0	1	0	1/4 V_{CC}
4	3	0	1	1	3/8 V_{CC}
5	4	1	0	0	1/2 V_{CC}
6	5	1	0	1	5/8 V_{CC}
7	6	1	1	0	3/4 V_{CC}
8	7	1	1	1	7/8 V_{CC}

Eine dritte Stelle wird also ein Achtel der Betriebsspannung beisteuern. Ein solches Netzwerk lässt sich beliebig ausbauen.

Die **kleinste Änderung** der Ausgangsspannung lässt sich dann mit Hilfe der folgenden Gleichung berechnen :

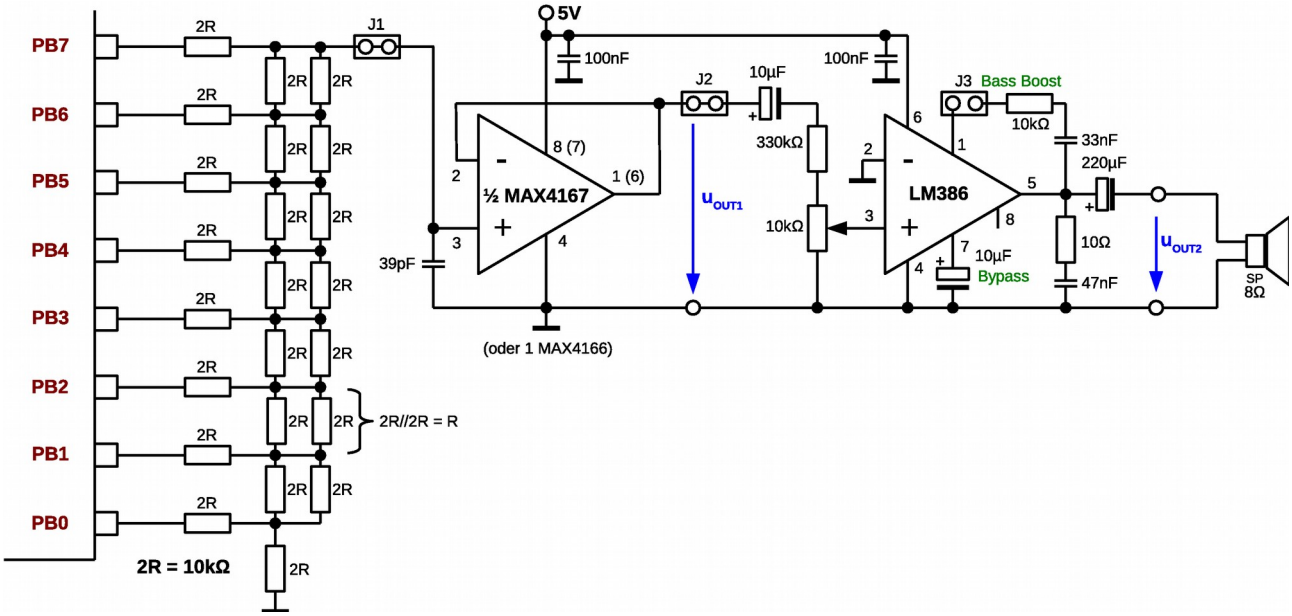
$$\Delta U_{Amin} = \frac{1}{2^n} \cdot V_{CC}$$

Ein solches R-2R-Netzwerkes hat einen konstanten Innenwiderstand $R_i = R$, egal für wie viele Stellen es ausgelegt wurde. Eine Belastung des Ausgangs bewirkt also keine Verfälschung des Resultats. Nur die maximal abgreifbare Spannung sinkt.

Um in der Praxis ein R-2R-Netzwerk zu bauen braucht man präzise Widerstandswerte (E96-Reihe, 1 %). Mit einer Parallelschaltung kann der Widerstandswert halbiert werden. Um die Ausgänge des Controllers nicht zu sehr zu belasten, sollen die Widerstandswerte nicht kleiner als 10 kΩ sein.

Ein nachgeschalteter Operationsverstärker minimiert die Belastung des Netzwerks und liefert den nötigen Strom. Um den ganzen Spannungsbereich zu nutzen (GND – VCC) wird ein „r*ail to rail*“

OPV einzusetzen, der bis zu den Grenzen seiner Betriebsspannung arbeiten kann. Der nachgeschaltete Verstärker (LM386) erlaubt es einen Lautsprecher anzuschließen.



Bemerkungen: Natürlich gibt es Digital-Analog-Wandler mit R-2R-Netzwerke auch in integrierter Form.

Eine andere und einfachere Möglichkeit ein analoges Signal zu erzeugen ist die Pulsweitenmodulation (siehe Timer).

Weitere Aufgaben

⌂ **C30B** Berechne die kleinste Spannungsänderung für den obigen 8-Bit-D/A-Wandler. In welcher Schaltung wird der Operationsverstärker betrieben? Ermittle die Verstärkung und die Funktion Bass Boost (J3) aus dem Datenblatt des LM386.

⌂ **C30C** Programmiere einen kleinen Funktionsgenerator. Über zwei Schalter S0 und S1 soll die Wellenform ausgewählt werden können (Sinus, Dreieck, Sägezahn und Rechteck). Der Sinus wird mit der folgenden Tabelle erzeugt (Datei auf weigu.lu/a/asm). Sinus, Dreieck und Sägezahn werden mit der höchstmöglichen Frequenz ausgegeben. Beim Rechteck kann eine beliebige Zeitschleife eingefügt werden.

a) Zeichne das dem Programmbeispiel entsprechende Flussdiagramm.
b) Schreibe das Assemblerprogramm.
   Speichere es unter dem Namen **"C30C_DAC_wavegen.asm"**.
c) Ermittle mit Hilfe eines Oszilloskops oder eines Frequenzzählers die maximalen Frequenzen für Sinus, Dreieck und Sägezahn.
d) Die Ausgabefrequenz aller Wellenformen soll nun auf 440 Hz verringert werden. Bestimme die nötigen Zeitverzögerungen zwischen zwei Ausgaben und erweitere entsprechend das Programm.

Sinustabelle:

```
.DB   128, 131, 134, 137, 140, 143, 146, 149
.DB   152, 155, 158, 162, 165, 167, 170, 173
.DB   176, 179, 182, 185, 188, 190, 193, 196
.DB   198, 201, 203, 206, 208, 211, 213, 215
.DB   218, 220, 222, 224, 226, 228, 230, 232
.DB   234, 235, 237, 238, 240, 241, 243, 244
.DB   245, 246, 248, 249, 250, 250, 251, 252
.DB   253, 253, 254, 254, 254, 255, 255, 255
.DB   255, 255, 255, 255, 254, 254, 254, 253
.DB   253, 252, 251, 250, 250, 249, 248, 246
.DB   245, 244, 243, 241, 240, 238, 237, 235
.DB   234, 232, 230, 228, 226, 224, 222, 220
.DB   218, 215, 213, 211, 208, 206, 203, 201
.DB   198, 196, 193, 190, 188, 185, 182, 179
.DB   176, 173, 170, 167, 165, 162, 158, 155
.DB   152, 149, 146, 143, 140, 137, 134, 131
.DB   128, 124, 121, 118, 115, 112, 109, 106
.DB   103, 100, 97, 93, 90, 88, 85, 82
.DB   79, 76, 73, 70, 67, 65, 62, 59
.DB   57, 54, 52, 49, 47, 44, 42, 40
.DB   37, 35, 33, 31, 29, 27, 25, 23
.DB   21, 20, 18, 17, 15, 14, 12, 11
.DB   10, 9, 7, 6, 5, 5, 4, 3
.DB   2, 2, 1, 1, 1, 0, 0, 0
.DB   0, 0, 0, 0, 1, 1, 1, 2
.DB   2, 3, 4, 5, 5, 6, 7, 9
.DB   10, 11, 12, 14, 15, 17, 18, 20
.DB   21, 23, 25, 27, 29, 31, 33, 35
.DB   37, 40, 42, 44, 47, 49, 52, 54
.DB   57, 59, 62, 65, 67, 70, 73, 76
.DB   79, 82, 85, 88, 90, 93, 97, 100
.DB   103, 106, 109, 112, 115, 118, 121, 124
```

⌂ **C30D** Für Fortgeschrittene:
Die zwölf Tasten einer Matrixtastatur sollen dazu dienen Musik zu erzeugen. (siehe Tabelle). Erweitere das obige Programm, so dass die erwünschten Töne in einer beliebigen Wellenform anhand der Tasten erzeugt werden können. Zur Abfrage der Tastatur kann das Unterprogramm "**SR_3x4_KEYPAD.asm**" verwendet werden (siehe Aufgabe B50D).
Schreibe das Assemblerprogramm und speichere es unter dem Namen **"C30D_DAC_tonegen.asm"**.

	f (Hz)	f (Hz)
Do (C)	131	262
Re (D)	147	294
Mi (E)	165	330
Fa (F)	175	349
Sol (G)	196	392
La (A)	220	
Si (H, B)	247	

Tipps: Die Ausgabe aller Wellenform aus 256 Byte großen Flash-Tabellen erleichtert die Aufgabe.

Es muss ein Lautsprecher verwendet werden, der diese tiefen Tön (Sinus hat keine Oberwellen) auch ausgeben kann!

⌂ **C30E**

a) Mit Hilfe des Electret-Mikrofons und des A/D-Wandlers sollen jetzt Signale digitalisiert werden und dann gleich wieder mit dem D/A/Wandler über den Lautsprecher ausgegeben werden.
Schreibe das Assemblerprogramm.
Speichere es unter dem Namen **"C30E_ADC_DAC.asm"**.

b) Vervollständige das folgende Programm mit den entsprechenden Initialisierungen und beschreibe ausführlich seine Funktionsweise.
Speichere es unter dem Namen **"C30E_ADC_DAC_delay.asm"**.

```
;---------------------------------------------------------------------------
;       Organisation des Datenspeichers (SRAM)
;---------------------------------------------------------------------------
.DSEG                           ;was ab hier folgt kommt in den SRAM-Speicher
TAB:   .BYTE  1800              ;1800 Byte grosse Tabelle im Datensegment
```

```
MAIN:   sbi     ADCSRA,ADSC     ;AD-Wandlung ausloesen fuer free running mode
                                ;einmaliger Vorgang
LOOP1:  ldi     XL,LOW(TAB)
        ldi     XH,HIGH(TAB)
        ldi     WL,LOW(1800)
        ldi     WH,HIGH(1800)

LOOP2:  sbis    ADCSRA,ADIF     ;Polling bis Wandlung fertig
        rjmp    LOOP2
        in      Tmp1,ADCH       ;oberste 8 Bit einlesen und ausgeben
        sbi     ADCSRA,ADIF     ;ADC Interrupt Flag manuell loeschen (mit 1!)
        st      X+,Tmp1

        sbiw    WL,1
        breq    LOOP1

        ld      Tmp1,X
        out     PORTB,Tmp1
        rjmp    LOOP2            ;Endlosschleife
```

C4 DDS

Kurze Einführung

Bei der Direkten Digitalen Synthese (*Direct Digital Synthesis*) wird ein periodisches bandbegrenztes, analoges Signal (zum Beispiel ein Sinussignal) mit Hilfe eines D/A-Wandlers oder einer PWM aus einem zeitlich änderndem digitalen Signal erzeugt.

Der große Vorteil des DDS-Verfahrens ist die praktisch beliebig feine Frequenzauflösung. Weitere Vorteile sind das schnelle Umschalten zwischen den Frequenzen, die große erreichbare Bandbreite und hohe Frequenzstabilität.

Die weite Verbreitung von ICs, die die komplette Hardware eines Synthesizers nach dem DDS-Verfahren realisieren, hat wesentlich zum Erfolg des Verfahrens beigetragen.

Warum DDS?

Die Berechnung zum Beispiel einer Sinusschwingung (8 Bit: u=255*sin(α)) mit dem Mikrocontroller würde viel zu lange dauern. Es ist einfacher die Amplitudenwerte einer Schwingung in einer Tabelle abzulegen und dann auf diese zuzugreifen. Die Frequenz ist dann allerdings durch die Zugriffsgeschwindigkeit des Programms festgelegt.

Beispiel: Sinustabelle mit 256 Werten, Timerinterrupt, der die Routine zur Ausgabe der Werte mit 1 MHz aufruft: Maximale Frequenz: 1 MHz/256 = 3,9 kHz. Die Änderung der Frequenz ist nur durch die Änderung der Timerinterrupt-Frequenz möglich.

Mit einem Trick lassen sich auch höhere Frequenzen ausgeben. Gibt man nur jeden zweiten Tabellenwert aus, so verdoppelt sich die Frequenz. Bei jedem dritten Wert verdreifacht die Frequenz usw..

Pro Periode müssen mindestens noch 2 Werte übertragen werden (Nyquist-Kriterium). Man ist also auch nach oben hin in der Frequenz begrenzt.

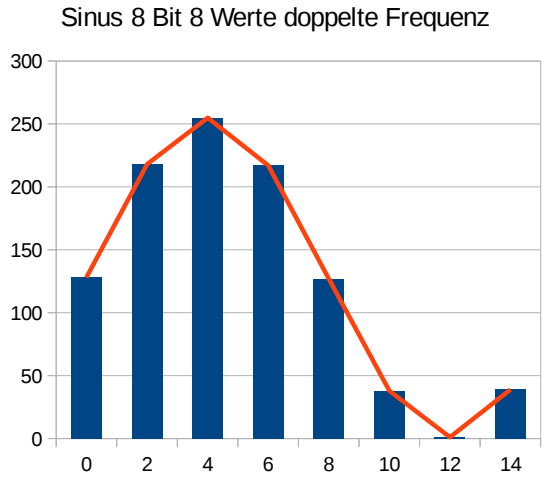
Beispiel: Sinustabelle mit 256 Werten, Timerinterrupt, der die Routine zur Ausgabe der Werte mit 36,363 kHz aufruft:

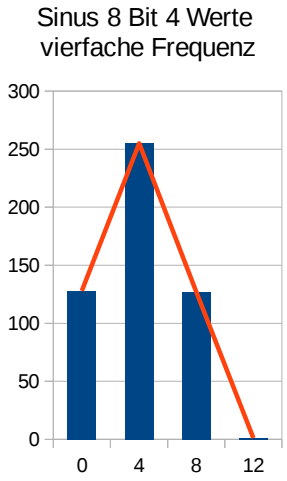
Grundfrequenz:	36,36 kHz/256 =	140 Hz
Jeder zweite Wert:	36,36 kHz/128 =	280 Hz
Jeder dritte Wert:	36,36 kHz/85 =	426 Hz
Jeder vierte Wert:	36,36 kHz/64 =	568 Hz

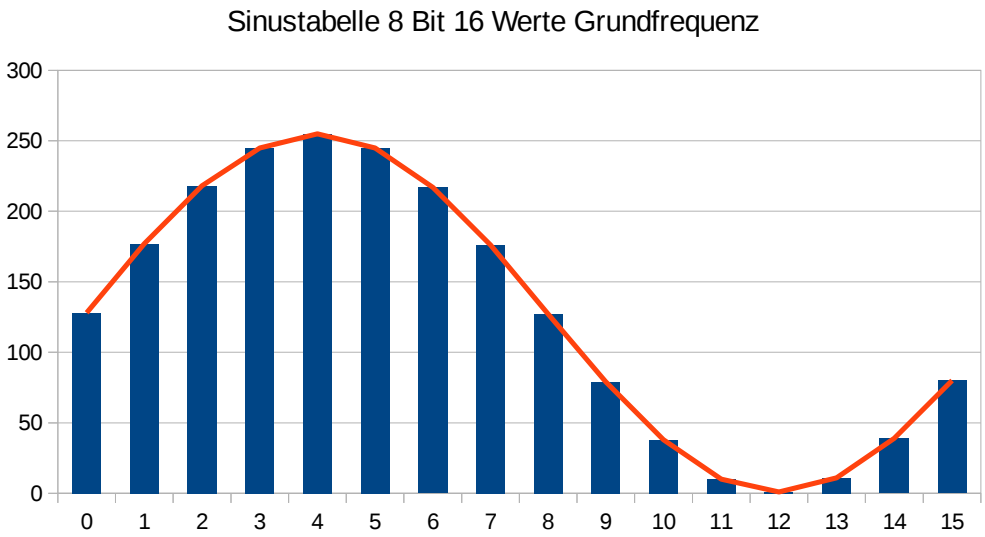
...

Maximale Frequenz: 36,36 kHz/2 = 18,18k Hz.

Beispiel: Tabelle (Wavetable, B = 8 Bit, Amplitude 0-255) mit 16 Werten (Adresszeiger P = 4 Bit):







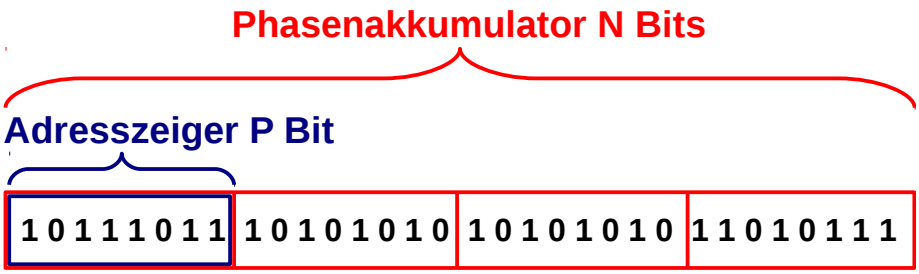
Nachteilig ist die bei diesem Verfahren die immer gröbere Abstufung des Signals. Außerdem sind so nur ganzzahlige Vielfache der Grundfrequenz erreichbar.

Mit Hilfe der DDS lassen sich diese beiden Nachteile umgehen.

DDS mit dem Mikrocontroller

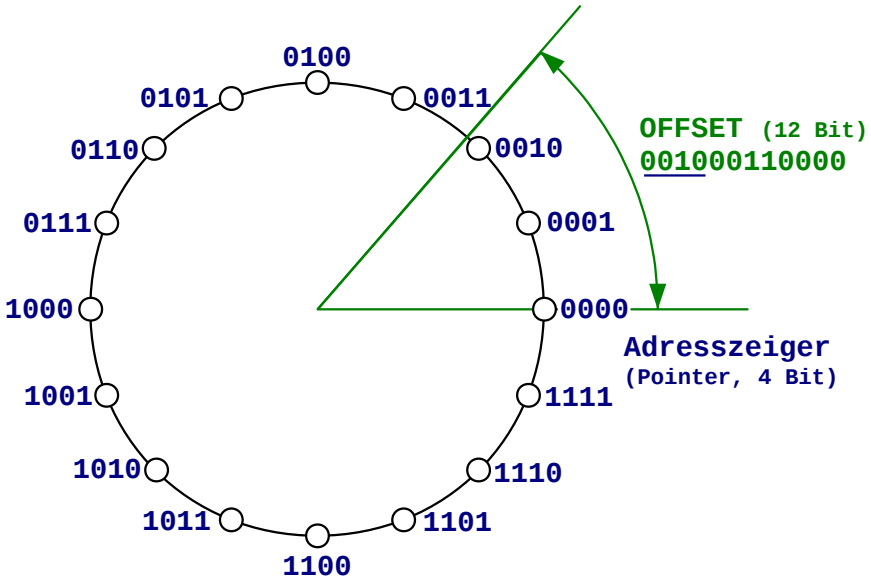
Statt eines einfachen binären Zählers, der die Tabellenwerte der Reihe nach aufruft, verwenden wir einen sogenannten Phasenakkumulator mit N Bit. Dieses Register ist viel größer als, zum Adressieren der Tabelle notwendig wäre. Meist werden Register mit 4 oder 6 Byte (N = 32 bzw. 48 Bit) verwendet. Nur die oberen P-Bit (zum Beispiel 8 Bit für eine Tabelle mit 256 Werten) des Phasenregister werden dann zur Adressierung der Tabelle verwendet.

Beispiel: Phasenakku mit 4 Byte

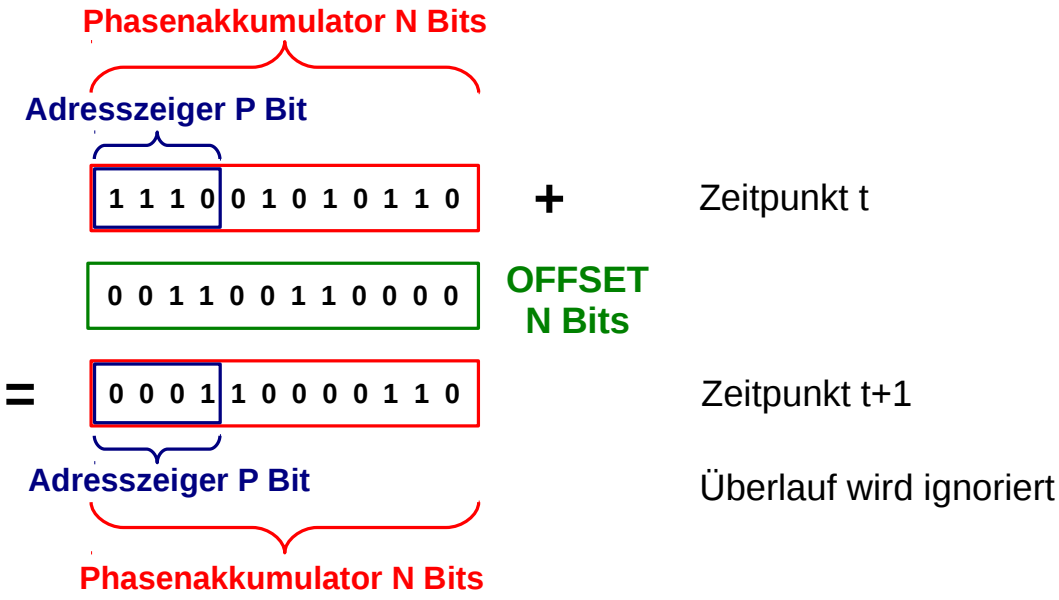


Das Phasenregister wird um einen festen Phasenoffset (Phaseninkrement), hier Offset genannt erhöht. Einsteht dabei ein Überlauf, so wird dieser ignoriert. Dieses Phasenoffset-Register hat dieselbe Größe wie das Phasenregister.

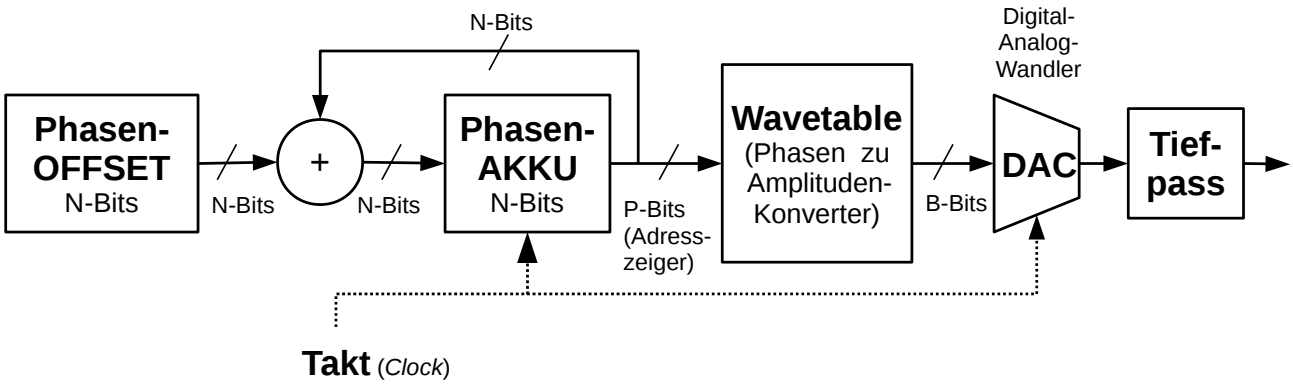
Für die folgenden Beispiele wird ein Phasenakku und ein Offset mit je N = 12 Bit verwendet. Der Adresszeiger hat P = 4 Bit um eine Tabelle mit 16 Werten zu adressieren. Die Amplitude beträgt B = 8 Bit (0-255).



Jetzt sind nicht mehr ganzzahlige Vielfache der Frequenz möglich, sondern auch Zwischenwerte (Der Effekt ist wie wenn, man mit Kommastellen arbeiten würde, also zum Beispiel auch jeden 1,3ten Wert nehmen könnte).



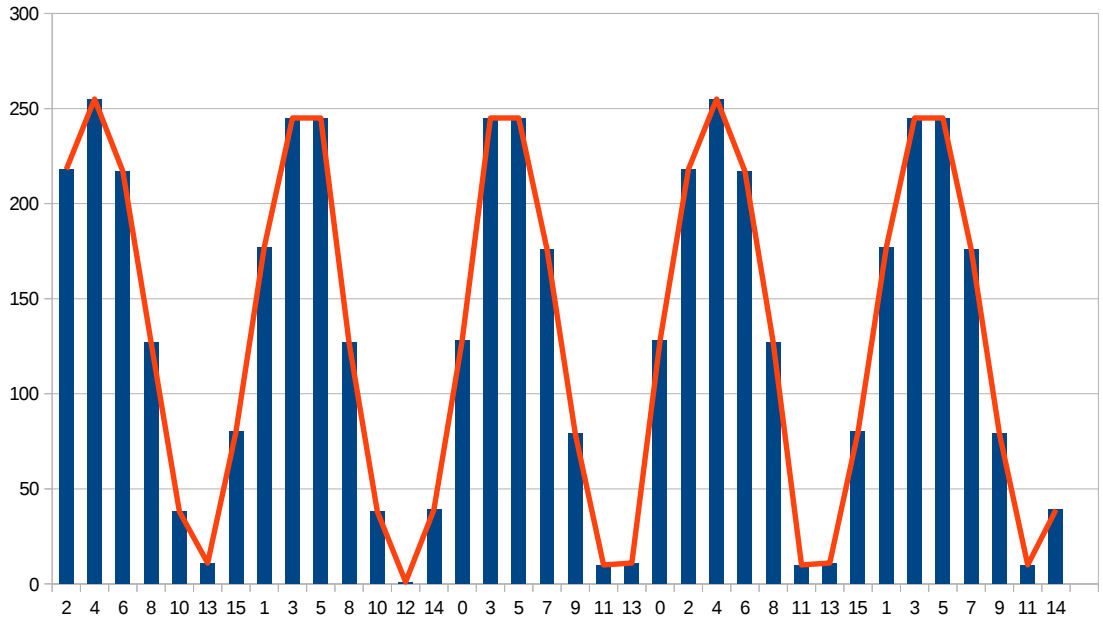
Blockschaltbild der DDS:



Beispiel: N = 12 Bit → Phasenakku und Offset haben 12 Bit-Breite
Für den Adresszeiger (*Pointer*) werden nur die oberen 4 Bit verwendet (P = 4Bit)

Der Überlauf bei der Addition wird nicht berücksichtigt (Modulo 2^N).

	Phasenakku		**Offset**	**Adresszeiger**
	0000 00000000	+	001000110000	0000 = 0
=	0010 00110000	+	001000110000	0010 = 2
=	0100 01100000	+	001000110000	0100 = 4
=	0110 10010000	+	001000110000	0110 = 6
=	1000 11000000	+	001000110000	1000 = 8
=	1010 11110000	+	001000110000	1010 = 10
=	1101 00100000	+	001000110000	1101 = 13
=	1111 01010000	+	001000110000	1111 = 15
=	0001 10000000	+	001000110000	0001 = 1
=	0011 10110000	+	001000110000	0011 = 3
=	0101 11100000	+	001000110000	0101 = 5
=	1000 00010000	+	001000110000	1000 = 8
	usw.			



Die Frequenz Ausgangsfrequenz f_{out} ergibt sich durch folgende Formel:

$$f_{out} = \frac{Offset \cdot f_{CLK}}{2^N}$$

Die Auflösung beträgt dabei: $f_{CLK}/2^N$.

f_{CLK} ist die Frequenz mit der die Werte an den D/A-Wandler bzw. mit der PWM ausgegeben werden. Um sie zu erzeugen, kann man zum Beispiel einen Timerinterrupt verwenden.
Die Formel lässt erkennen, dass man umso höhere Frequenzen erzeugen kann, je höher die Frequenz f_{CLK} ist. Ist sie beim Mikrocontroller jedoch zu hoch, bleibt dem Hauptprogramm keine Zeit mehr. Beim AVR-Syntesizer meeblib (meeblip.com) wurde zum Beispiel eine Frequenz von 36,36 kHz gewählt. So verbleiben beim verwendeten 16 MHz Quarz rund 440 Taktzyklen zum Bearbeiten der Daten (ISR + Hauptprogramm).

Je höher die zu erzeugende Frequenz f_{OUT} im Vergleich zu f_{CLK} wird, umso mehr machen sich „Unschönheiten“ des DDS-Prinzips durch Jitter, Rauschen und Nebenwellen im Spektrum bemerkbar. Mit einer Taktrate von 36,36 kHz kommt man nicht höher als rund 3 kHz in der Programmiersprache C. Mit Assembler kann man durch effizientere Programmierung viel Zeit einsparen und so höherer Frequenzen erreichen.

Beispiel: Nehmen wir in unserem obigen Beispiel eine Taktfrequenz f_{CLK} von 36363 Hz an, so erhalten wir:

$$f_{out} = \frac{OFFSET \cdot f_{CLK}}{2^N} = \frac{560 \cdot 36363}{4096} = 4971{,}5\,Hz \qquad (0b0010001100 = 560)$$

Die kleinste Frequenz (Grundfrequenz der Tabelle, alle 16 Werte) ohne DDS ergibt sich mit $f_{OUTGF} = f_{CLK}/16 = 2272{,}7$ Hz.

Die Frequenz wurde also mit diesem Offset um den Faktor $f_{OUT}/f_{OUTGF} = 4971{,}5/2272{,}7 = 2{,}187$ erhöht.
Wird der Offset um 1 erhöht, so steigt die Frequenz auf 4980,38Hz. Die Differenz von 8,878 Hz errechnet sich aus $f_{CLK}/2^N$.

Die Grundfrequenz wird übrigens mit dem Offset 0b000100000000 = 256 erreicht. Es sind mit der DDS auch kleinere Frequenzen als die Grundfrequenz möglich!

Hier einige Beispiele, aus denen man ersieht, wie sich die Auflösung mit Hilfe der Verbreiterung des Phasenakkus erhöhen lässt:

Breite des Phasenakku N	12 Bit	16 Bit	16 Bit	24 Bit	32 Bit
Taktfrequenz f_{CLK}	36363 Hz	36363 Hz	100kHz	36363 Hz	36363 Hz
maximaler Offset	4096	65636	65636	16777216	4294967296
OFFSET	700	11200	11200	2867200	734003200
Ausgangsfrequenz f_{OUT}	6,21 kHz	6,21 kHz	17,089 kHz	6,21 kHz	6,21 kHz
Auflösung	8,877Hz	0,554 Hz	1,525 Hz	0,002167 Hz	8,466µHz!

Der Offset für eine beliebige Frequenz errechnet sich aus:

$$Offset = \frac{f_{out} \cdot 2^N}{f_{CLK}}$$

Assembler-Beispielcode für ein Programm, das ein Sinussignal mit 1 kHz ausgibt:

F_{CLK} = 36363 Hz, N = 24 Bit, Offset = 1000Hz*2^{24}/36363Hz = 461373 = 0x070A3D

```
;***************************************************************************
;*
;*      Titel:  D1_dds_1kHz.asm (DDS feste Frequenz)
;*      Datum:  05/07/14        Version: 0.2 (18/12/14)
;*      Autor:  WEIGU
;*
;*      Informationen zur Beschaltung:
;*
;*      Prozessor:      ATmega32A           Quarzfrequenz: 16MHz
;*      Eingaenge:
;*      Ausgaenge:      PORTB verbunden mit einem R2R DAC +
;*                      RC-Tiefpass 47nF, 1,2k
;*
;*      Informationen zur Funktionsweise:
;*
;*      Frequenzgenerator mit DDS (f = 1kHz)
;*
;*      Eine Interruptroutine wird mit 36364 Hz aufgerufen (Routine kurz halten
;*      da nur 440 Taktzyklen fuer Routine + Hauptprogramm zur Verfuegung
;*      stehen).
;*      Innerhalb dieser Routine wird ein Sinussignalsignal mit Hilfe der
;*      direkten Frequenz Synthese (DDS = Direct Digital Synthesis) erzeugt.
;*      Dazu wird ein 24-Bit Phasenakku mit einem Wert inkrementiert (OFFSET)
;*      der proportional zur erwuenschten Frequenz ist. Die oberen 8 Bit des
;*      Adresszaehler bilden dann die Adresse fuer eine eine ROM Tabelle mit 256
;*      Werten (Lockup-Tabelle, Wavetable).
;*
;*      OFFSET = 2 ^ 24 * Freq / SamplingFreq (36364Hz)
;*      OFFSET = 461,37344*f
;*      fuer f = 1kHz: OFFSET = 461373 = 0x070A3D
;*
;***************************************************************************

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

;---------------------------------------------------------------------------
;       Organisation des Datenspeichers (SRAM)
;---------------------------------------------------------------------------
.DSEG                          ;was ab hier folgt kommt in den SRAM-Speicher

; Phaseninkrement OFFSET des DCO
OFFSET0: .byte 1
OFFSET1: .byte 1
OFFSET2: .byte 1

;+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
;       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)

;-----------------------------------------------------------------------------
;       Sprungadressen fuer die Interrupts organisieren (ISR VECTORS)
;-----------------------------------------------------------------------------
        ;Vektortabelle (im Flash-Speicher)
.ORG    OC2addr                 ;Timer2 Compare
        rjmp    ISRTC

;-----------------------------------------------------------------------------
;       Initialisierungen und eigene Definitionen
;-----------------------------------------------------------------------------
.ORG    INT_VECTORS_SIZE        ;Platz fuer ISR Vektoren lassen
INIT:

.DEF    PHASE0 = r2             ;Phasenakku des DCO
.DEF    PHASE1 = r3
.DEF    PHASE2 = r4
.DEF    Zero    = r15           ;Register 1 wird zum Rechnen benoetigt
        clr     r15             ;und mit Null belegt
.DEF    Tmp1  = r16             ;Register 16 dient als erster Zwischenspeicher

        ;Stapel initialisieren (fuer Unterprogramme bzw. Interrupts)
        ldi     Tmp1,LOW(RAMEND) ;RAMEND (SRAM) ist in der Definitions-
        out     SPL,Tmp1         ;datei festgelegt
        ldi     Tmp1,HIGH(RAMEND)
        out     SPH,Tmp1

        ;Initialisiere den Phasenakku
        clr     PHASE0
        clr     PHASE1
        clr     PHASE2

        ;OFFSET fuer 1 kHz initialisieren (0x070A3D)
        ldi     Tmp1,0x3D
        sts     OFFSET0,Tmp1
        ldi     Tmp1,0x0A
        sts     OFFSET1,Tmp1
        ldi     Tmp1,0x07
        sts     OFFSET2,Tmp1

        ;Initialisiere Timer2 (CTC Interrupt mit 36,36 KHz):
        ldi     r16, 54         ;OCR2 = 54 gives 16MHz/8/55 = 36363.63636 Hz
        out     OCR2, r16
        ldi     r16, 0x0A       ;CTC mode (OC2 off), prescaler = CK/8
        out     TCCR2, r16
        ldi     r16, 0x80       ;OCIE2=1
        out     TIMSK, r16

        ser     Tmp1            ;PortC Ausgang (R2R DAC)
        out     DDRB,Tmp1
        sei                     ;Interrupts global erlauben
;-----------------------------------------------------------------------------
;       Hauptprogramm
;-----------------------------------------------------------------------------
MAIN:   rjmp    MAIN    ;Endlosschleife

;-----------------------------------------------------------------------------
;       Unterprogramme und Interrupt-Behandlungsroutinen
;-----------------------------------------------------------------------------
ISRTC:  push    r16
        in      r16,SREG
        push    r16
        push    ZL
        push    ZH

        ldi     ZL,LOW(SINET*2)         ;Adressiere Wavetable
        ldi     ZH,HIGH(SINET*2)
        add     ZL,PHASE2
        adc     ZH,Zero
```

```
        lpm     r16,Z
        out     PORTB,r16               ;Ausgabe Sample (Wavetable)

        ;Erhoehe die Phase des DCO um Delta
        lds     r16,OFFSET0
        add     PHASE0,r16
        lds     r16,OFFSET1
        adc     PHASE1,r16
        lds     r16,OFFSET2
        adc     PHASE2,r16

ISRTCR: pop     ZH
        pop     ZL
        pop     r16
        out     SREG,r16
        pop     r16
        reti

;-----------------------------------------------------------------------------
;       Tabellen im Programmspeicher (Flash)
;-----------------------------------------------------------------------------
;Sinustabelle mit 256 Werten
SINET:  .DB     128, 131, 134, 137, 140, 143, 146, 149
        .DB     152, 155, 158, 162, 165, 167, 170, 173
        .DB     176, 179, 182, 185, 188, 190, 193, 196
        .DB     198, 201, 203, 206, 208, 211, 213, 215
        .DB     218, 220, 222, 224, 226, 228, 230, 232
        .DB     234, 235, 237, 238, 240, 241, 243, 244
        .DB     245, 246, 248, 249, 250, 250, 251, 252
        .DB     253, 253, 254, 254, 254, 255, 255, 255
        .DB     255, 255, 255, 255, 254, 254, 254, 253
        .DB     253, 252, 251, 250, 250, 249, 248, 246
        .DB     245, 244, 243, 241, 240, 238, 237, 235
        .DB     234, 232, 230, 228, 226, 224, 222, 220
        .DB     218, 215, 213, 211, 208, 206, 203, 201
        .DB     198, 196, 193, 190, 188, 185, 182, 179
        .DB     176, 173, 170, 167, 165, 162, 158, 155
        .DB     152, 149, 146, 143, 140, 137, 134, 131
        .DB     128, 124, 121, 118, 115, 112, 109, 106
        .DB     103, 100, 97, 93, 90, 88, 85, 82
        .DB     79, 76, 73, 70, 67, 65, 62, 59
        .DB     57, 54, 52, 49, 47, 44, 42, 40
        .DB     37, 35, 33, 31, 29, 27, 25, 23
        .DB     21, 20, 18, 17, 15, 14, 12, 11
        .DB     10, 9, 7, 6, 5, 5, 4, 3
        .DB     2, 2, 1, 1, 1, 0, 0, 0
        .DB     0, 0, 0, 0, 1, 1, 1, 2
        .DB     2, 3, 4, 5, 5, 6, 7, 9
        .DB     10, 11, 12, 14, 15, 17, 18, 20
        .DB     21, 23, 25, 27, 29, 31, 33, 35
        .DB     37, 40, 42, 44, 47, 49, 52, 54
        .DB     57, 59, 62, 65, 67, 70, 73, 76
        .DB     79, 82, 85, 88, 90, 93, 97, 100
        .DB     103, 106, 109, 112, 115, 118, 121, 124

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

Bascom-Beispielcode von Jean Daubenfeld für ein Programm, das ein Sinussignal mit 1 kHz ausgibt:

FCLK = 36363 Hz, N = 32 Bit, Offset = 1000Hz*2^{32}/36363Hz = 118111601 = 0x070A3D71

!Das Bascom Programm arbeitet mit 4 Byte (32 Bit).

```
' 1kHz Sinus DDS (PortB verbunden mit R2R DAC + RC-Tiefpass 47nF, 1,2k
' Offset = fout*2^N/fCLK = 1000Hz*2^32/36363Hz = 118111601 = &H070A3D71
' Jean Daubenfeld / WEIGU

$regfile = "m32def.dat"
$crystal = 16000000

$hwstack = 50
$swstack = 50
$framesize = 60

Config Portb = Output


Enable Timer2
Config Timer2 = Timer , Prescale = 8 , Clear_timer = 1
On Compare2 Tim2_isr
Enable Compare2
Compare2 = 54                    'OCR2 = 54 gives 16MHz/8/55 = 36,36kHz
Enable Interrupts

Dim Offset As Dword
Dim Phasenakku As Dword
Offset = &H070A3D71
Phasenakku = &H00000000

Dim Adresszeiger As Dword
Dim Sin_wert As Byte

'------------            Hauptprogramm:          ------------------------------
Do
Loop
End

'------------            Unterprogramme:          ------------------------------
Tim2_isr:
   Phasenakku = Phasenakku + Offset
   Adresszeiger = Phasenakku
   Shift Adresszeiger , Right , 24
Sin_wert = Lookup(Adresszeiger , Sinus)
Portb = Sin_wert
Return

'------------            Sinustabelle:          ------------------------------
Sinus:
Data 128 , 131 , 134 , 137 , 140 , 143 , 146 , 149
Data 152 , 155 , 158 , 162 , 165 , 167 , 170 , 173
Data 176 , 179 , 182 , 185 , 188 , 190 , 193 , 196
Data 198 , 201 , 203 , 206 , 208 , 211 , 213 , 215
Data 218 , 220 , 222 , 224 , 226 , 228 , 230 , 232
Data 234 , 235 , 237 , 238 , 240 , 241 , 243 , 244
Data 245 , 246 , 248 , 249 , 250 , 250 , 251 , 252
Data 253 , 253 , 254 , 254 , 254 , 255 , 255 , 255
Data 255 , 255 , 255 , 255 , 254 , 254 , 254 , 253
Data 253 , 252 , 251 , 250 , 250 , 249 , 248 , 246
Data 245 , 244 , 243 , 241 , 240 , 238 , 237 , 235
Data 234 , 232 , 230 , 228 , 226 , 224 , 222 , 220
Data 218 , 215 , 213 , 211 , 208 , 206 , 203 , 201
Data 198 , 196 , 193 , 190 , 188 , 185 , 182 , 179
Data 176 , 173 , 170 , 167 , 165 , 162 , 158 , 155
Data 152 , 149 , 146 , 143 , 140 , 137 , 134 , 131
```

```
Data 128 , 124 , 121 , 118 , 115 , 112 , 109 , 106
Data 103 , 100 , 97 , 93 , 90 , 88 , 85 , 82
Data 79 , 76 , 73 , 70 , 67 , 65 , 62 , 59
Data 57 , 54 , 52 , 49 , 47 , 44 , 42 , 40
Data 37 , 35 , 33 , 31 , 29 , 27 , 25 , 23
Data 21 , 20 , 18 , 17 , 15 , 14 , 12 , 11
Data 10 , 9 , 7 , 6 , 5 , 5 , 4 , 3
Data 2 , 2 , 1 , 1 , 1 , 0 , 0 , 0
Data 0 , 0 , 0 , 0 , 1 , 1 , 1 , 2
Data 2 , 3 , 4 , 5 , 5 , 6 , 7 , 9
Data 10 , 11 , 12 , 14 , 15 , 17 , 18 , 20
Data 21 , 23 , 25 , 27 , 29 , 31 , 33 , 35
Data 37 , 40 , 42 , 44 , 47 , 49 , 52 , 54
Data 57 , 59 , 62 , 65 , 67 , 70 , 73 , 76
Data 79 , 82 , 85 , 88 , 90 , 93 , 97 , 100
Data 103 , 106 , 109 , 112 , 115 , 118 , 121 , 124
```