

Programmierung der AVR-Microcontroller mit C

Dieses Tutorial soll dem Einsteiger helfen, mit der Programmiersprache C die Microcontroller der Atmel AVR-Reihe zu programmieren.

Es gibt einige Gründe, weshalb eine höhere Programmiersprache der reinen Assembler-Programmierung vorgezogen werden kann. So kann mit C das Programm wesentlich lesbarer geschrieben werden als in Assembler.

Es soll allerdings auch nicht verheimlicht werden, dass in Assembler in der Regel schnellerer Programmcode entwickelt werden kann als mit C. Meiner Meinung nach jedoch können wir 99 Prozent aller Aufgaben problemlos mit C meistern und für die ganz krassen Fälle kann sogar Assembler-Code direkt in ein C-Programm eingebunden werden.

Der Autor hat sich alle Mühe gegeben, sein Wissen hier fehlerfrei wiederzugeben. Fehler können jedoch nicht ausgeschlossen werden.

Fehlermeldungen bitte an christian.schifferle@bluewin.ch

Der Autor übernimmt keinerlei Haftung für etwaige Schäden wie z.B. durchgeknallte Sicherungen, welche durch die Verwendung dieses Dokuments entstehen könnten.

Die deutschen Leser mögen mir verzeihen, dass ich, wie es in der Schweiz üblich ist, kein scharfes ß verwendet habe.

Die vorliegende Dokumentation ist und bleibt geistiges Machwerk von Christian Schifferle und darf nicht als Eigenproduktion angepriesen werden.

Änderungen an diesem Dokument sind nur mit schriftlicher Genehmigung des Autors gestattet.

Das Dokument darf nach Belieben an Dritte weiter gegeben werden, sofern der Copyright-Hinweis auf den Autor nicht verändert oder gelöscht wird.

Der Autor, nämlich ich, das ist:

Christian Schifferle

Risweg 7

CH-4624 Härkingen

E-Mail: christian.schifferle@bluewin.ch

Achtung:

Der gesamte Kurs als ZIP-Datei kann [hier](#) herunter geladen werden. Die ZIP-Datei muss dann auf dem eigenen Rechner in ein beliebiges Verzeichnis entpackt werden. Die Startdatei heisst dann Index.htm. Für die Freunde des PDF-Formats gibt es den Kurs [hier](#) auch als PDF-Datei. Für diejenigen, welche neu in die Programmiersprache C einsteigen, empfiehlt es sich, zuvor die ebenfalls [hier](#) erhältliche Einführung in die Programmiersprache C durchzuarbeiten.

Inhaltsverzeichnis

1	Benötigte Werkzeuge	4
2	Definition einiger Datentypen.....	4
2.1	BYTE	4
2.2	WORD	4
2.3	Bitfelder.....	5
3	Grundsätzlicher Programmaufbau eines μ C-Programms.....	5
3.1	Sequentieller Programmablauf	5
3.2	Interruptgesteuerter Programmablauf.....	6
4	Allgemeiner Zugriff auf Register	7
4.1	I/O-Register.....	7
4.1.1	Lesen eines I/O-Registers	7
4.1.2	Schreiben eines I/O-Registers	8
4.1.3	Warten auf einen bestimmten Zustand	8
4.1.4	Speicherbezogener Portzugriff	8
5	Zugriff auf Ports.....	10
5.1	Datenrichtung bestimmen	10
5.1.1	Ganze Ports	10
5.2	Digitale Signale	10
5.3	Ausgänge.....	10
5.4	Eingänge (Wie kommen Signale in den μ C).....	11
5.4.2	Pull-Up Widerstände aktivieren.....	12
5.5	Analog.....	13
5.5.1	16-Bit Portregister (ADC, ICR1, OCR1, TCNT1).....	13
6	Übung 1, Tastenprellen	13
6.1	Benötigte Bauteile	13
6.2	Die Schaltung.....	14
6.3	Aufgabe	14
6.4	Die Lösung.....	14
7	Übung 2, Tasten-Entprellung.....	14
7.1	Die bessere Lösung	15
8	Der UART, Teil 1	16
8.1	Allgemeines zum UART	16
8.2	Die Hardware	18
8.3	Senden mit dem UART	19
8.3.1	Schreiben einer Zeichenkette (String).....	19
9	Übung 2, Senden mit dem UART	20
9.1	Benötigte Bauteile	20
9.2	Die Schaltung.....	20
9.2.1	Aufgabe.....	21
9.2.2	Hilfestellung.....	21
10	Der UART, Teil 2	22
10.1	Empfangen mit dem UART	22
10.2	AVR's ohne UART	23
10.3	Zusammenfassung.....	23
11	Übung 2, Empfangen mit dem UART	23
11.1	Benötigte Bauteile	23
11.2	Die Schaltung.....	24
11.2.1	Aufgabe.....	24
11.3	Weitere Übungen	24
12	Analoge Ein- und Ausgabe	25
12.1	ADC (Analog Digital Converter).....	25
12.1.1	Messen eines Widerstandes.....	25
12.1.2	ADC über Komparator	27

12.1.3	Der ADC im AVR.....	27
12.2	DAC (Digital Analog Converter).....	30
12.2.1	DAC über mehrere digitale Ausgänge	30
12.2.2	PWM (Pulsweitenmodulation).....	31
13	Übung 5, DAC über Widerstandsnetzwerk	33
14	Übung 6, PWM.....	34
15	Die Timer/Counter des AVR	35
15.1.1	Der Vorzähler (Prescaler)	35
15.2	8-Bit Timer/Counter.....	35
15.3	16-Bit Timer/Counter.....	36
15.3.1	Die PWM-Betriebsart.....	40
15.3.2	Vergleichswert-Überprüfung.....	40
15.3.3	Einfangen eines Eingangssignals (Input Capturing).....	41
15.4	Gemeinsame Register	41
16	Übung 7, Mehrkanal-Software-PWM	42
17	Übung 8, 16-Bit-Timer/Counter.....	43
18	Der Watchdog	46
18.1	Wie funktioniert nun der Watchdog	46
18.2	Ein paar grundsätzliche Gedanken	47
19	Programmieren mit Interrupts.....	49
19.1	Anforderungen an die Interrupt-Routine	49
19.2	Interrupt-Quellen	49
19.3	Register	50
19.4	Allgemeines über die Interrupt-Abarbeitung	52
19.4.1	Das Status-Register	52
19.5	Interrupts mit dem AVR GCC Compiler (WinAVR)	52
19.6	Was tut das Hauptprogramm	54
20	Übung 9, Interrupts.....	54
21	Schlusswort.....	56

1 BENÖTIGTE WERKZEUGE

Um die Übungen in diesem Tutorial nachvollziehen zu können benötigen wir folgende Hard- und Software:

- Testboard für die Aufnahme eines AVR Controllers Ihrer Wahl. Dieses Testboard kann durchaus auch selber zusammen gelötet werden. So arbeite ich mit einem Testboard, welches ich mir auf einer Veroboard-Platine zusammen gestellt habe. Für die Versuche bzw. Übungen in diesem Tutorial habe ich jeweils einen AT90S2313 verwendet. Dieser Controller weist meiner Meinung nach das beste Preis-/Leistungsverhältnis auf.
- AVRGCC-Compiler
Ich beschränke mich hier auf den GCC Compiler, weil ich diesen selber verwende und weil er kostenlos zu haben und weit verbreitet ist.
- Programmiersoftware, z.B. PonyProg oder die von Atmel bereitgestellte Software, und natürlich ein passendes Kabel, um die Programme auf den Atmel übertragen zu können.

2 DEFINITION EINIGER DATENTYPEN

Für die Programmierung von Microcontrollern ist es sinnvoll, dass wir uns vorerst einige Datentypen definieren, welche den Zugriff auf die verschiedenen Komponenten des Controllers vereinfachen oder wenigstens das Programm lesbarer machen können.

```
typedef unsigned char  BYTE;  
typedef unsigned short WORD;
```

2.1 BYTE

Der Datentyp BYTE definiert eine Variable mit 8 Bit Breite zur Darstellung von ganzen Zahlen im Bereich zwischen 0 ... 255.

2.2 WORD

Der Datentyp WORD definiert eine Variable mit 16 Bit Breite zur Darstellung von ganzen Zahlen im Bereich zwischen 0 ... 65535.

Wir können aber auch die von der AVR-Umgebung des Compiler in der Headerdatei <inttypes.h> definierten Datentypen verwenden. Ich persönlich finde dieselben allerdings nicht sehr leserlich.

```
typedef signed char  int8_t;  
typedef unsigned char uint8_t;  
  
typedef int int16_t;  
typedef unsigned int uint16_t;  
  
typedef long int32_t;  
typedef unsigned long uint32_t;  
  
typedef long long int64_t;  
typedef unsigned long long uint64_t;  
  
typedef int16_t intptr_t;  
typedef uint16_t uintptr_t;
```

2.3 Bitfelder

Beim Programmieren von Microcontrollern muss auf jedes Byte oder sogar auf jedes Bit geachtet werden. Oft müssen wir in einer Variablen lediglich den Zustand 0 oder 1 speichern. Wenn wir nun zur Speicherung eines einzelnen Wertes den kleinsten bekannten Datentypen, nämlich **unsigned char**, nehmen, dann verschwenden wir 7 Bits, da ein **unsigned char** ja 8 Bit breit ist.

Hier bietet uns die Programmiersprache C ein mächtiges Werkzeug an, mit dessen Hilfe wir 8 Bits in einer einzelnen Bytevariable zusammen fassen und (fast) wie 8 einzelne Variablen ansprechen können.

Die Rede ist von sogenannten Bitfeldern. Diese werden als Strukturelemente definiert. Sehen wir uns dazu doch am besten gleich ein Beispiel an:

```
struct {  
    unsigned char bStatus_1:1;    // 1 Bit für bStatus_1  
    unsigned char bStatus_2:1;    // 1 Bit für bStatus_2  
    unsigned char bNochNBit:1;    // Und hier noch mal ein Bit  
    unsigned char b2Bits:2;       // Dieses Feld ist 2 Bits breit  
    // All das hat in einer einzigen Byte-Variable Platz.  
    // die 3 verbleibenden Bits bleiben ungenutzt  
} x;
```

Der Zugriff auf ein solches Feld erfolgt nun wie beim Strukturzugriff bekannt über den Punkt- oder den Dereferenzierungs-Operator:

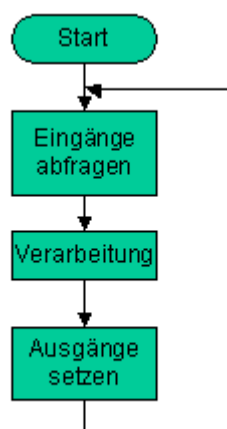
```
x.bStatus_1 = 1;  
x.bStatus_2 = 0;  
x.b2Bits    = 3;
```

3 GRUNDSÄTZLICHER PROGRAMMAUFBAU EINES μ C-PROGRAMMS

Wir unterscheiden zwischen 2 verschiedenen Methoden, um ein Microcontroller-Programm zu schreiben, und zwar völlig unabhängig davon, in welcher Programmiersprache das Programm geschrieben wird.

3.1 Sequentieller Programmablauf

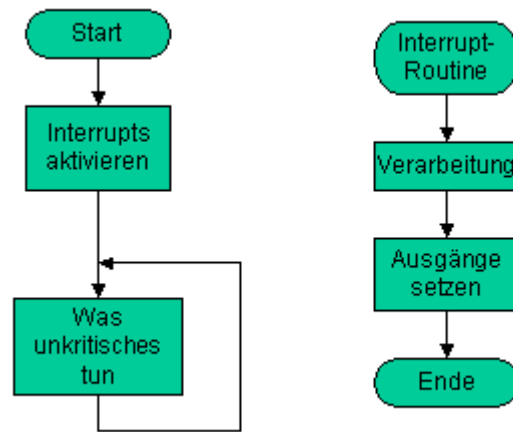
Bei dieser Programmiermethode wird eine Endlosschleife programmiert, welche im Wesentlichen immer den gleichen Aufbau hat:



3.2 Interruptgesteuerter Programmablauf

Bei dieser Methode werden beim Programmstart zuerst die gewünschten Interruptquellen aktiviert und dann in eine Endlosschleife gegangen, in welcher Dinge erledigt werden können, welche nicht zeitkritisch sind..

Wenn ein Interrupt ausgelöst wird so wird automatisch die zugeordnete Interruptfunktion ausgeführt.



4 ALLGEMEINER ZUGRIFF AUF REGISTER

Die AVR-Controller verfügen über eine Vielzahl von Registern. Die meisten davon sind sogenannte Schreib-/Leseregister. Das heisst, das Programm kann die Inhalte der Register auslesen und beschreiben.

Einige Register haben spezielle Funktionen, andere wiederum könne für allgemeine Zwecke (Speichern von Datenwerten) verwendet werden.

Einzelne Register sind bei allen AVR's vorhanden, andere wiederum nur bei bestimmten Typen. So sind beispielsweise die Register, welche für den Zugriff auf den UART notwendig sind selbstverständlich nur bei denjenigen Modellen vorhanden, welche über einen UART verfügen.

Die Namen der Register sind in den Headerdateien zu den entsprechenden AVR-Typen definiert. Wenn im Makefile der MCU-Typ definiert ist so bindet das System automatisch die richtige Headerdatei ein.

Man kann der MCU-Typ aber selbstverständlich auch noch in der C-Quelldatei definieren, wenn man Freude daran hat.

4.1 I/O-Register

Die I/O-Register haben einen besonderen Stellenwert bei den AVR Controllern. Sie dienen dem Zugriff auf die Ports und die Schnittstellen des Controllers.

Wir unterscheiden zwischen 8-Bit und 16-Bit Registern. Vorerst behandeln wir mal die 8-Bit Register.

Hinweis: Die folgenden Funktionen erwarten als Argument für das jeweilige Portregister konstante Werte. Am besten verwenden wir die entsprechenden defines aus der Headerdatei <io.h>. Wenn die Portadresse über eine 8-Bit Variable übergeben quitiert der Inline Assembler dies jeweils mit 2 Fehlermeldungen folgender Form:

```
<Dateiname>:<Zeilennummer>: warning: asm operand 0 probably doesn't match constraints
```

```
<Dateiname>:<Zeilennummer>: warning: asm operand 1 probably doesn't match constraints
```

Offensichtlich läuft das Programm so auch tatsächlich nicht korrekt ab. Wenn wir also Ports über Variablen ansprechen wollen müssen wir auf die Low Level-Funktion [mmio](#) ausweichen.

4.1.1 LESEN EINES I/O-REGISTERS

Der gesamte Inhalt eines Registers kann mit dem Befehl

```
inp (<register>);
```

ausgelesen werden.

4.1.1.1 Lesen eines Bits

Die AVR-Bibliothek stellt auch Funktionen zur Abfrage eines einzelnen Bits eines Registers zur Verfügung:

```
bit_is_set (<port>, <pin>);
```

Die Funktion **bit_is_set** prüft, ob ein Bit gesetzt ist. Wenn das Bit gesetzt ist wird ein Wert ungleich 0 zurückgegeben. Genau genommen ist es die Wertigkeit des abgefragten Bits, also 1 für Bit0, 2 für Bit1, 4 für Bit2 etc.

```
bit_is_clear (<port>, <pin>);
```

Die Funktion **bit_is_clear** prüft, ob ein Bit gelöscht ist. Wenn das Bit gelöscht ist, also auf 0 ist, wird ein Wert ungleich 0 zurückgegeben.

4.1.2 SCHREIBEN EINES I/O-REGISTERS

Zum Beschreiben eines I/O-Registers verwendet man allgemein den Befehl

```
outp (<wert>, <register>);
```

Als Wert muss die Bitmaske mit den Werten aller Pins angegeben werden.

4.1.2.1 Schreiben eines Bits

Auch für das Setzen bzw. Löschen eines einzelnen Bits eines I/O-Registers stellt die AVR-Bibliothek entsprechende Funktionen zur Verfügung.

```
sbi (<register>, <bitnummer>);
```

Die Funktion **sbi** setzt ein beliebiges Bit eines Registers, das heisst, es wird der logische Wert 1 in das Bit geschrieben. Die anderen Bits des Ports werden nicht verändert. Das niederwertigste Bit hat die Bitnummer 0.

```
cbi (<register>, <bitnummer>);
```

Die Funktion **cbi** löscht ein beliebiges Bit eines Registers, das heisst, es wird der logische Wert 0 in das Bit geschrieben. Die anderen Bits des Ports werden nicht verändert. Das niederwertigste Bit hat die Bitnummer 0.

4.1.3 WARTEN AUF EINEN BESTIMMTEN ZUSTAND

Es gibt in der Bibliothek sogar Funktionen, die Warten, bis ein bestimmter Zustand auf einem Bit erreicht ist.

Es ist allerdings normalerweise eine eher unschöne Programmieretechnik.

```
loop_until_bit_is_set(<register>, <bitnummer>);
```

Die Funktion **loop_until_bit_is_set** wartet in einer Schleife, bis das definierte Bit gesetzt ist. Wenn das Bit beim Aufruf der Funktion bereits gesetzt ist wird die Funktion sofort wieder verlassen. Das niederwertigste Bit hat die Bitnummer 0.

```
loop_until_bit_is_clear(<register>, <bitnummer>);
```

Die Funktion **loop_until_bit_is_clear** wartet in einer Schleife, bis das definierte Bit gelöscht ist. Wenn das Bit beim Aufruf der Funktion bereits gelöscht ist wird die Funktion sofort wieder verlassen.

Das niederwertigste Bit hat die Bitnummer 0.

4.1.4 SPEICHERBEZOGENER PORTZUGRIFF

In der Regel sind die weiter oben erwähnten Funktionen zu bevorzugen. Es kann aber auch sein, dass wird mal eine Stufe tiefer einsteigen müssen. Dann verwenden wir für den Portzugriff das Synonym **__mmio**.

```
__mmio(<port>)
```

Diese Funktion dient dem speicherbasierten Zugriff auf die Ports (Memory Mapped I/O).

Die Funktion kann sowohl zum Lesen als auch zum Schreiben eines Ports verwendet werden.

Um ein einzelnes Bit in einem Port zu setzen bzw. zu löschen kann also ein der folgenden Befehlszeilen verwendet werden:

```
__mmio (<port>) = __mmio (<port>) | (1 << <bitnummer>) // Setzt ein Bit  
__mmio (<port>) = __mmio (<port>) & ~(1 << <bitnummer>) // Löscht ein Bit
```

Die anderen Bits des Ports bleiben unverändert.

5 ZUGRIFF AUF PORTS

Alle Ports der AVR-Controller werden über Register gesteuert. Dazu sind jedem Port 3 Register zugeordnet:

DDRx	Datenrichtungsregister für Port x (x entspricht A , B , C oder D).
PORTx	Datenregister für Port x (x entspricht A , B , C oder D). Dieses Register wird verwendet, um die Ausgänge eines Ports anzusteuern. Wird ein Port als Eingang geschaltet, so können mit diesem Register die internen Pull-Up Widerstände aktiviert oder deaktiviert werden (1 = aktiv).
PINx	Eingangadresse für Port x (x entspricht A , B , C oder D). Dies ist kein eigentliches Register, sondern definiert lediglich eine Adresse, in welcher der aktuelle Zustand der Eingangspins eines Ports vom Controller abgelegt werden. Nichtsdestotrotz erfolgt der Zugriff auf den Zustand der Pins genau so, wie wenn PINx ein normales Register wäre. Die Adresse kann nur gelesen und nicht beschrieben werden.

5.1 Datenrichtung bestimmen

Zuerst muss die Datenrichtung der verwendeten Pins bestimmt werden.

Um dies zu erreichen wird das Datenrichtungsregister des entsprechenden Ports beschrieben.

```
outp (<bits>, <port>);
```

Für jeden Pin, der als Ausgang verwendet werden soll, muss dabei das entsprechende Bit auf dem Port gesetzt werden. Soll der Pin als Eingang verwendet werden muss das entsprechende Bit gelöscht sein.

Wollen wir also beispielsweise Pin 0 bis 4 von Port B als Ausgänge definieren so schreiben wir folgende Zeile:

```
outp (0x1F, DDRB);  
// Binär 00011111 = Hexadezimal 1F
```

Die Pins 5 bis 7 werden als Eingänge geschaltet.

5.1.1 GANZE PORTS

Um einen ganzen Port als Ausgang zu definieren, kann der folgende Befehl verwendet werden:

```
outp (0xFF, DDRB);
```

Im Beispiel wird der Port B als Ganzes als Ausgang geschaltet.

5.2 Digitale Signale

Am einfachsten ist es, digitale Signale mit dem Microcontroller zu erfassen bzw. auszugeben.

5.3 Ausgänge

Wir wollen nun einen als Ausgang definierten Pin auf Logisch 1 setzen.

Dazu schreiben wir den entsprechenden Wert in das Portregister des entsprechenden Ports.

Mit dem Befehl

```
outp (0x04, PORTB);
```

wird also der Ausgang an Pin 2 gesetzt (Beachte, dass die Bits immer von 0 an gezählt werden, das niederwertigste Bit ist also Bit 0 und nicht etwa Bit 1).

Man beachte bitte, dass bei der Verwendung der **outp**-Funktion immer alle Pins gleichzeitig angegeben werden. Man sollte also zuerst den aktuellen Wert des Ports einlesen und das Bit des gewünschten Ports in diesen Wert einfließen lassen.

Es gibt jedoch in der AVRGCC-Bibliothek Funktionen, welche dies selbständig machen. Die Funktionen lassen sich wie folgt verwenden:

```
sbi (PORTB, 2);    // Setzt Pin 2 auf Logisch 1 (EIN)  
cbi (PORTB, 2);    // Setzt Pin 2 auf Logisch 0 (AUS)
```

5.4 Eingänge (Wie kommen Signale in den μ C)

Die digitalen Eingangssignale können auf verschiedene Arten zu unserer Logik gelangen.

5.4.1.1 Signalkopplung

Am einfachsten ist es, wenn die Signale direkt aus einer anderen digitalen Schaltung übernommen werden können. Hat der Ausgang der entsprechenden Schaltung TTL-Pegel dann können wir sogar direkt den Ausgang der Schaltung mit einem Eingangspin von unserem Controller verbinden. Hat der Ausgang der anderen Schaltung keinen TTL-Pegel so müssen wir den Pegel über entsprechende Hardware, z.B. einen Optokoppler, anpassen. Die Masse der beiden Schaltungen muss selbstverständlich miteinander verbunden werden.

Der Software selber ist es natürlich letztendlich egal, wie das Signal eingespeist wird. Wir können ja ohnehin lediglich prüfen, ob an einem Pin unserer Controllers eine logische 1 (Vcc) oder eine logische 0 (Masse) anliegt.

Die Abfrage der Zustände der Portpins erfolgt über den Befehl

```
inp (<port>)
```

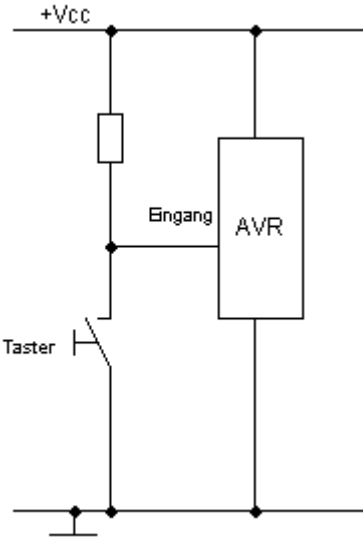
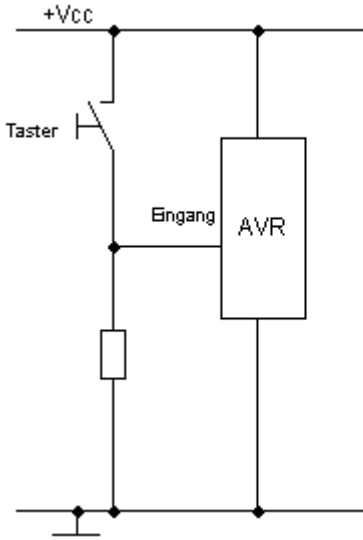
Wobei es hier sehr wichtig ist, als Portadresse nicht etwa das Portregister **PORTx zu verwenden, sondern die Porteingangsadresse **PINx**. Dies ist ein oft gemachter Fehler!**

Wollen wir also die aktuellen Signalzustände von Port D abfragen und in einer Variable namens bPortD abspeichern so schreiben wir dazu folgende Befehlszeile:

```
bPortD = inp (PIND);
```

5.4.1.2 Tasten und Schalter

Der Anschluss mechanischer Kontakte an den Microcontroller gestaltet sich ebenfalls ganz einfach, wobei wir zwei unterschiedliche Methoden unterscheiden müssen (*Active Low* und *Active High*):

Active Low	Active High
	
<p>Bei dieser Methode wird der Kontakt zwischen den Eingangspin des Controllers und Masse geschaltet.</p> <p>Damit bei offenem Schalter der Controller kein undefiniertes Signal bekommt wird zwischen die Versorgungsspannung und den Eingangspin ein sogenannter Pull-Up Widerstand geschaltet. Dieser dient dazu, den Pegel bei geöffnetem Schalter auf logisch 1 zu ziehen.</p> <p>Der Widerstandswert des Pull-Up Widerstands ist an sich nicht kritisch. Es muss jedoch beachtet werden, dass über den Widerstand ein Strom in den Eingang fließt, also sollte er nicht zu klein gewählt werden um den Controller nicht zu zerstören. Wird er allerdings zu hoch gewählt ist die Wirkung eventuell nicht gegeben. Als üblicher Wert haben sich 10 Kiloohm eingebürgert.</p> <p>Die AVR's haben sogar an den meisten Pins softwaremässig zuschaltbare interne Pull-Up Widerstände, welche wir natürlich auch verwenden können.</p>	<p>Hier wird der Kontakt zwischen die Versorgungsspannung und Masse geschaltet. Damit bei offener Schalterstellung kein undefiniertes Signal am Controller ansteht wird zwischen den Eingangspin und die Masse ein Pull-Down Widerstand geschaltet. Dieser dient dazu, den Pegel bei geöffneter Schalterstellung auf logisch 0 zu halten,</p>

5.4.2 PULL-UP WIDERSTÄNDE AKTIVIEREN

Die internen Pull-Up Widerstände von Vcc zu den einzelnen Portpins werden über das Register **PORTx** aktiviert bzw. deaktiviert, wenn ein Pin als **Eingang** geschaltet ist.

Wird der Wert des entsprechenden Portpins auf 1 gesetzt so ist der Pull-Up Widerstand aktiviert. Bei einem Wert von 0 ist der Pull-Up Widerstand nicht aktiv.

Man sollte jeweils entweder den internen oder einen externen Pull-Up Widerstand verwenden, aber nicht beide zusammen.

```
outp (0x00, DDRD);      // Port D als Eingang schalten
outp (0x00, PORTD);    // Interne Pull-Up Widerstände aus
```

Im Beispiel wird der gesamte Port D als Eingang geschaltet und alle Pull-Up Widerstände deaktiviert.

5.4.2.1.1 TASTENENTPRELLUNG

Nun haben alle mechanischen Kontakte, sei es von Schaltern, Tastern oder auch von Relais, die unangenehme Eigenschaft zu prellen. Dies bedeutet, dass beim Schliessen des Kontaktes derselbe nicht direkt Kontakt herstellt, sondern mehrfach ein- und ausschaltet bis zum endgültigen Herstellen des Kontaktes.

Soll nun mit einem schnellen Microcontroller gezählt werden, wie oft ein solcher Kontakt geschaltet wird, dann haben wir ein Problem, weil das Prellen als mehrfache Impulse gezählt wird. Diesem Phänomen muss beim Schreiben des Programms unbedingt Rechnung getragen werden.

5.5 Analog

Leider können wir mit unseren Controllern keine analogen Werte direkt ausgeben oder einlesen. Dazu müssen wir Umwege gehen, doch dies wird in einem späteren Kapitel behandelt.

5.5.1 16-BIT PORTREGISTER (ADC, ICR1, OCR1, TCNT1)

Einige der Portregister in den AVR-Controllern sind 16 Bit breit, Man spricht dann von einem **Wort**. Für den Zugriff auf diese Register stellt die Funktionsbibliothek zwei spezielle Befehle zur Verfügung:

```
__inw (<port>);
```

Liest den aktuellen Wert eines 16-Bit Portregisters ein.

```
__outw (<wert>, <port>);
```

Schreibt einen Wert in ein 16-Bit Portregister.

6 ÜBUNG 1, TASTENPRELLEN

Alle mechanischen Kontakte, sei es von Schaltern, Tastern oder auch von Relais, haben die unangenehme Eigenschaft zu prellen. Dies bedeutet, dass beim Schliessen des Kontaktes derselbe nicht direkt Kontakt herstellt, sondern mehrfach ein- und ausschaltet bis zum endgültigen Herstellen des Kontaktes.

Soll nun mit einem schnellen Microcontroller gezählt werden, wie oft ein solcher Kontakt geschaltet wird, dann haben wir ein Problem, weil das Prellen als mehrfache Impulse gezählt wird.

Diese Übung soll dazu helfen, das Prellen von mechanischen Tastern zu untersuchen.

6.1 Benötigte Bauteile

Für den Aufbau der Testschaltung benötigen wir nebst dem Experimentierboard mit eingesetztem AVR Controller folgende Bauteile:

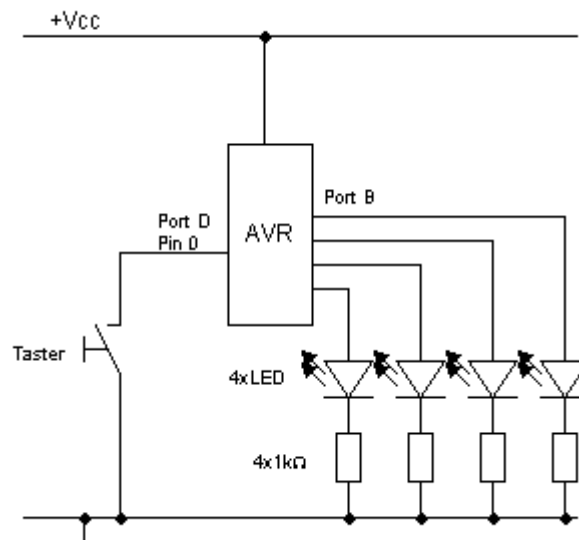
- 1 Taster, zur Not tut's auch ein Schalter
- 4 bis 8 LED's, Farbe egal
- 4 bis 8 Widerstände 1 kOhm

6.2 Die Schaltung

Der Taster wird von Pin 0 von Port D nach Masse angeschlossen. Durch die Verwendung des internen Pull-Up Widerstandes brauchen wir nicht mal einen extern Widerstand.

Die LED's werden an Port B angeschlossen, von Pin 0 an aufwärts bis je nach Anzahl LED's.

- Achtung: Bitte darauf achten, dass nicht zu viel Strom aus dem AVR gezogen wird. Deswegen auch die hohen Widerstandswerte für die Vorwiderstände der LED's. Bei 1 kOhm werden pro leuchtende LED ca. 3 mA gezogen.



6.3 Aufgabe

Es soll nun ein Programm entwickelt werden, welches die Anzahl Tastendrucke zählt und das Ergebnis in binärer Form auf die Leuchtdioden ausgibt. Versuche, das Programm vorerst alleine zu entwickeln.

6.4 Die Lösung

Hier ist das [Makefile](#) und die zugehörige [C-Quelldatei](#).

7 ÜBUNG 2, TASTEN-ENTPRELLUNG

Ausgehend von der Übung 1 soll jetzt ein Weg gefunden werden, das Pellen der Kontakte auszutricksen und unser Programm so zu gestalten, dass ein Druck auf eine Taste auch tatsächlich nur als ein einzelner Druck erkannt wird.

Der Versuchsaufbau von Übung 1 kann auch für diese Übung verwendet werden.

Ein Lösungsansatz besteht darin, dass der Zustand *Taste gedrückt* während einer bestimmten Zeitdauer anstehen muss bevor der Tastendruck registriert wird.

Im einfachsten Fall wird dies dadurch realisiert, dass beim ersten Erkennen des Zustands eine Warteschleife mit einer definierten Anzahl Loops durchlaufen wird. Wenn danach der Status der Taste immer noch gedrückt ist können wir diesen Zustand als gegeben hinnehmen.

Diese Lösung hat den Nachteil, dass wir während dem Warten nicht mehr auf andere Ereignisse reagieren können. Wenn wir das etwas besser Lösen wollen dann müssen wir uns etwas einfallen

lassen.

Diejenigen, die jetzt selber versuchen wollen eine bessere Lösung zu finden sollten jetzt aufhören zu lesen und sich an die Arbeit machen. Die unter euch, die sich noch zu unsicher fühlen können noch etwas weiter lesen.

7.1 Die bessere Lösung

Wir wollen nun folgendes versuchen, ausgehend von unserer Hauptprogrammschleife, welche immer durchlaufen werden soll.

Innerhalb unserer Programmschleife lesen wir den Zustand des entsprechenden Pins ein.

Ist der Pin Logisch 1, merken wir uns diesen Zustand.

Erkennen wir nun die negative Flanke, d.h. den Wechsel von 1 auf 0, dann initialisieren wir einen Zähler.

Solange der 0-Pegel ansteht inkrementieren wir den Zähler bei jedem Durchlauf unserer Hauptprogrammschleife. Hat der Zähler einen bestimmten Wert erreicht, den wir durch Versuche ermitteln müssen, dann wird der Tastendruck gewertet.

Damit jetzt nicht beim nächsten Durchlauf der Programmschleife der Tastenzähler weiter hochgezählt wird müssen wir uns merken, dass momentan die Taste gedrückt ist. Dazu verwenden wir eine entsprechende Statusvariable.

Dies kann eine eigene Variable sein, oder aber wir definieren eine Bitfeldvariable, in welcher wir die beiden Signale für den Portzustand und des Tastenstatus in einer einzigen BYTE-Variablen zusammenfassen.

So, nun solltet ihr aber wirklich versuchen, die Lösung selbständig zu erarbeiten.

Wenn es gar nicht klappen sollte, dann habt ihr hier den [C-Quellcode](#) und das [Makefile](#) meiner Musterlösung.

8 DER UART, TEIL 1

Wir werden in zukünftigen Übungen in die Situation kommen, dass wir Informationen vom Controller auswerten bzw. anzeigen müssen wobei es vielleicht dann nicht mehr reicht, ein paar Leuchtdioden anzusteuern.

Somit brauchen wir eine Verbindung vom AVR zu unserem PC, und dies am besten über die serielle Schnittstelle.

8.1 Allgemeines zum UART

Einige AVR-Controller haben einen vollduplexfähigen UART (**U**niversal **A**synchronous **R**eceiver and **T**ransmitter) schon eingebaut. Dies ist eine wirklich feine Sache, haben wir doch damit schon das nötige Werkzeug, um mit anderen Geräten, welche über eine serielle Schnittstelle verfügen (insbesondere mit unserem PC), zu kommunizieren.

Übrigens: Vollduplex heisst nichts anderes, als dass der Baustein gleichzeitig senden und empfangen kann.

Der UART wird über vier separate Register angesprochen:

UCR	UART Control Register.								
	In diesem Register stellen wir ein, wie wir den UART verwenden möchten. Das Register ist wie folgt aufgebaut:								
	Bit	7	6	5	4	3	2	1	0
	Name	RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHR9	RXB8	TXB8
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	W
	Initialwert	0	0	0	0	0	0	1	0
	RXCIE	RX Complete Interrupt Enable Wenn dieses Bit gesetzt ist wird ein UART RX Complete Interrupt ausgelöst wenn ein Zeichen vom UART empfangen wurde. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein.							
	TXCIE	TX Complete Interrupt Enable Wenn dieses Bit gesetzt ist wird ein UART TX Complete Interrupt ausgelöst wenn ein Zeichen vom UART gesendet wurde. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein.							
	UDRIE	UART Data Register Empty Interrupt Enable Wenn dieses Bit gesetzt ist wird ein UART Datenregister Leer Interrupt ausgelöst, wenn der UART wieder bereit ist um ein neues zu sendendes Zeichen zu übernehmen. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein.							
	RXEN	Receiver Enable Nur wenn dieses Bit gesetzt ist arbeitet der Empfänger des UART überhaupt. Wenn das Bit nicht gesetzt ist kann der entsprechende Pin des AVR als normaler I/O-Pin verwendet werden.							
TXEN	Transmitter Enable Nur wenn dieses Bit gesetzt ist arbeitet der Sender des UART überhaupt. Wenn das Bit nicht gesetzt ist kann der entsprechende Pin des AVR als normaler I/O-Pin verwendet werden.								
CHR9	9 Bit Characters Wenn dieses Bit gesetzt ist können 9 Bit lange Zeichen übertragen und empfangen werden. Das 9. Bit kann bei Bedarf als zusätzliches Stopbit oder als Paritätsbit verwendet werden. Man spricht dann von einem 11-Bit Zeichenrahmen: 1 Startbit + 8 Datenbits + 1 Stopbit + 1 Paritätsbit = 11 Bits								
RXB8	Receive Data Bit 8 Wenn das vorher erwähnte CHR9-Bit gesetzt ist, dann enthält dieses Bit das 9. Datenbit eines empfangenen Zeichens.								
TXB8	Transmit Data Bit 8 Wenn das vorher erwähnte CHR9-Bit gesetzt ist, dann muss in dieses Bit das 9. Bit des zu sendenden Zeichens eingeschrieben werden bevor das eigentliche Datenbyte in das Datenregister geschrieben wird.								
USR	UART Status Register.								
	Hier teilt uns der UART mit, was er gerade so macht.								
	Bit	7	6	5	4	3	2	1	0
	Name	RXC	TXC	UDRE	FE	OR	-	-	-
	R/W	R	R/W	R	R	R	R	R	R
Initialwert	0	0	1	0	0	0	0	0	
RXC	UART Receive Complete Dieses Bit wird vom AVR gesetzt, wenn ein empfangenes Zeichen vom Empfangs-Schieberegister in das Empfangs-Datenregister transferiert								

	<p>wurde. Das Zeichen muss nun schnellstmöglich aus dem Datenregister ausgelesen werden. Falls dies nicht erfolgt bevor ein weiteres Zeichen komplett empfangen wurde wird eine Überlauf-Fehlersituation eintreffen. Mit dem Auslesen des Datenregisters wird das Bit automatisch gelöscht.</p> <p>TXC UART Transmit Complete Dieses Bit wird vom AVR gesetzt, wenn das im Sende-Schieberegister befindliche Zeichen vollständig ausgegeben wurde und kein weiteres Zeichen im Sendedatenregister ansteht. Dies bedeutet also, wenn die Kommunikation vollumfänglich abgeschlossen ist. Dieses Bit ist wichtig bei Halbduplex-Verbindungen, wenn das Programm nach dem Senden von Daten auf Empfang schalten muss. Im Vollduplexbetrieb brauchen wir dieses Bit nicht zu beachten. Das Bit nur dann automatisch gelöscht, wenn der entsprechende Interrupthandler aufgerufen wird, ansonsten müssen wir das Bit selber löschen.</p> <p>UDRE UART Data Register Empty Dieses Bit wird vom AVR gesetzt, wenn ein Zeichen vom Sendedatenregister in das Send-Schieberegister übernommen wurde und der UART nun wieder bereit ist, ein neues Zeichen zum Senden aufzunehmen. Das Bit wird automatisch gelöscht, wenn ein Zeichen in das Sendedatenregister geschrieben wird.</p> <p>FE Framing Error Dieses Bit wird vom AVR gesetzt, wenn der UART einen Zeichenrahmenfehler detektiert, d.h. wenn das Stopbit eines empfangenen Zeichens 0 ist. Das Bit wird automatisch gelöscht, wenn das Stopbit des empfangenen Zeichens 1 ist.</p> <p>OR OverRun Dieses Bit wird vom AVR gesetzt, wenn unser Programm das im Empfangsdatenregister bereit liegende Zeichen nicht abholt bevor das nachfolgende Zeichen komplett empfangen wurde. Das nachfolgende Zeichen wird verworfen. Das Bit wird automatisch gelöscht, wenn das empfangene Zeichen in das Empfangsdatenregister transferiert werden konnte.</p>
UDR	<p>UART Data Register. Hier werden Daten zwischen UART und CPU übertragen. Da der UART im Vollduplexbetrieb gleichzeitig empfangen und senden kann handelt es sich hier physikalisch um 2 Register, die aber über die gleiche I/O-Adresse angesprochen werden. Je nachdem, ob ein Lese- oder ein Schreibzugriff auf den UART erfolgt wird automatisch das richtige UDR angesprochen.</p>
UBRR	<p>UART Baud Rate Register. In diesem Register müssen wir dem UART mitteilen, wie schnell wir gerne kommunizieren möchten. Der Wert, der in dieses Register geschrieben werden muss, errechnet sich nach folgender Formel:</p> $UBRR = \text{Taktfrequenz} / (\text{Baudrate} * 16) - 1$ <p>Es sind Baudraten bis zu 115200 Baud und höher möglich.</p>

8.2 Die Hardware

Der UART basiert auf normalem TTL-Pegel mit 0V (LOW) und 5V (HIGH). Die Schnittstellenspezifikation für RS232 definiert jedoch -3V ... -12V (LOW) und +3 ... +12V (HIGH). Zudem muss der Signalaustausch zwischen AVR und Partnergerät invertiert werden. Für die

Anpassung der Pegel und das Invertieren der Signale gibt es fertige Schnittstellenbausteine. Der bekannteste davon ist wohl der MAX232. Allerdings kostet der auch wieder Geld und benötigt zusätzlich immerhin 4 externe Elkos.

Die in den PC eingebauten Schnittstellen vertragen ohne Klagen auch den TTL-Pegel vom AVR. Allerdings müssen wir immer noch die Signale invertieren. Im einfachsten Fall verwenden wir dazu jeweils einen einfachen NPN-Transistor und 2 Widerstände. Näheres dazu erfahrt ihr in den folgenden Übungen.

8.3 Senden mit dem UART

Wir wollen nun Daten mit dem UART auf die serielle Schnittstelle ausgeben.

Dazu müssen wir den UART zuerst mal initialisieren. Dazu setzen wir je nach gewünschter Funktionsweise die benötigten Bits im **UART Control Register**.

Da wir vorerst nur senden möchten und (noch) keine Interrupts auswerten wollen gestaltet sich die Initialisierung wirklich sehr einfach, da wir lediglich das **Transmitter Enable** Bit setzen müssen:

```
outp ((1 << TXEN), UCR);
```

Nun müssen wir noch die Baudrate festlegen. Gemäss unserer Formel brauchen wir dazu die Taktfrequenz des angeschlossenen Oszillators bzw. Quarz in die Formel einzufügen und das Resultat der Berechnung in das Baudratenregister des UART einzuschreiben:

```
#define F_CPU 4000000 // Zum Beispiel 4Mhz-Quarz
#define UART_BAUD_RATE 9600 // Wir versuchen mal mit 9600 Baud
```

```
outp (F_CPU / (UART_BAUD_RATE * 16L) - 1, UBRR);
```

Um nun ein Zeichen auf die Schnittstelle auszugeben müssen wir dasselbe lediglich in das **UART Daten Register** schreiben.

```
outp ('x', UDR); // Schreibt das Zeichen x auf die
Schnittstelle
```

8.3.1 SCHREIBEN EINER ZEICHENKETTE (STRING)

Wenn wir nun mehrere, aufeinanderfolgende Zeichen auf die Schnittstelle ausgeben wollen, dann müssen wir mit dem nächsten Zeichen warten, bis das vorhergehende jeweils aus dem Datenregister in das Sende-Schieberegister übernommen wurde.

Zu diesem Zweck können wir uns für's Erste eine einfache Funktion basteln:

```
void SendeString (char *szBuf)
{
    while (*szBuf++) {
        outp (*szBuf, UDR);
        loop_until_bit_is_set (USR, UDRE);
    }
}
```

Das ist jetzt insofern etwas kritisch, dass wir während dem Senden des Strings nicht mehr auf andere Ereignisse reagieren können. Dies wird aber ohnehin erst dann ein Thema, wenn wir mit unserem Programm gleichzeitig Senden und Empfangen wollen.

Wir werden zu einem späteren Zeitpunkt Lösung für dieses Problem finden.

9 ÜBUNG 2, SENDEN MIT DEM UART

Wir wollen nun ein Programm entwickeln, welches einen Zähler permanent inkrementiert und den jeweiligen Zählerstand immer auf die serielle Schnittstelle ausgibt.

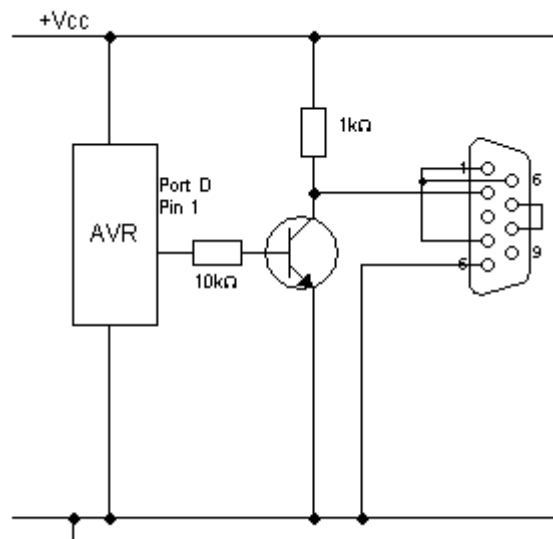
9.1 Benötigte Bauteile

Für den Aufbau der Testschaltung benötigen wir nebst dem Experimentierboard mit eingesetztem AVR Controller folgende Bauteile:

- 1 Transistor, NPN-Allerweltstyp
- 1 Widerstand 10 k Ω
- 1 Widerstand 1 k Ω
- 1 Sub-D Buchse 9pol W
- ca. 2m Signalkabel 2-Adrig

9.2 Die Schaltung

Da wir uns den Einsatz eines Pegelwandlers mit Inverters (MAX232 o.ä.) sparen wollen müssen wir das Signal des UART Ausgangspins invertieren. Dazu dient der Transistor, dessen Basis über den 10k Ω Vorwiderstand angesteuert wird. Am Kollektor greifen wir das invertierte Signal ab und leiten es zum Eingang der seriellen Schnittstelle am PC. Die meisten (eigentlich alle) PC's kommen mit 0 und 5V auf der RS232-Schnittstelle problemlos zurecht. Wer es ganz genau haben will kann aber natürlich auch einen integrierten Baustein wie den MAX232 einsetzen.



Die Drahtbrücken in der Sub-D Buchse dienen der Simulation der Hardware-Handshake Signale.

9.2.1 AUFGABE

Das zu entwickelnde Programm soll eine Variable einfach immer um den Wert 1 hochzählen und dann den aktuellen Zählerwert als Ganzzahl mit und ohne Vorzeichen, als Hexzahl und als Oktale Zahl in einer Zeile ausgeben.

Die Bildschirmausgabe auf dem Terminalprogramm am PC könnte dann z.B. so aussehen.

```
Dez:      1 Uns:      1 Hex:      1 Oct:      1
Dez:      2 Uns:      2 Hex:      2 Oct:      2
Dez:      3 Uns:      3 Hex:      3 Oct:      3
Dez:      4 Uns:      4 Hex:      4 Oct:      4
...
...
Dez:-25536 Uns:40000 Hex:9c40 Oct:116100
Dez:-25535 Uns:40001 Hex:9c41 Oct:116101
usw.
```

Die Baudrate sollte auf 9600 Baud eingestellt werden. Die Einstellungen am Terminalprogramm sind 8 Datenbits, 1 Stopbit, Keine Parität.

9.2.2 HILFESTELLUNG

Für die Ausgabe von formatierten Zahlenwerten habe ich euch eine kleine Funktion `UartPrintF` geschrieben, deren [Code](#) und [Headerdatei](#) hier zu finden sind.

So, und jetzt bitte schön an die Arbeit.

Wenn es gar nicht klappen sollte, dann habt ihr hier den [C-Quellcode](#) und das [Makefile](#) meiner Musterlösung.

10 DER UART, TEIL 2

Nun habt ihr wahrscheinlich alle schon Daten vom Controller zum PC geschickt. Es gibt aber durchaus auch Anwendungen, bei denen wir Informationen vom PC oder anderen Geräten über die serielle Schnittstelle an den Controller schicken können.

10.1 Empfangen mit dem UART

Bevor wir Daten über den UART einlesen können müssen wir diesen natürlich auch wieder initialisieren. Ich erinnere dazu auch an den [UART, Teil 1](#), in welchem das Senden besprochen wurde.

Um grundsätzlich mal Empfangen zu können, muss das **Receiver Enable** Bit im **UART Control Register** gesetzt werden:

```
outp ((1 << RXEN), UCR);
```

Wenn wir senden und empfangen wollen muss zusätzlich das **Transmitter Enable** Bit gesetzt werden. Dann sieht der Befehl so aus:

```
outp ((1 << RXEN) | (1 << TXEN), UCR);
```

Und denkt daran, dass wir auch noch die Baudrate festlegen müssen.

```
#define F_CPU 4000000 // Zum Beispiel 4Mhz-Quarz
#define UART_BAUD_RATE 9600 // Wir versuchen mal mit 9600 Baud
```

```
outp (F_CPU / (UART_BAUD_RATE * 16L) - 1, UBRR);
```

Sodele, jetzt müssen wir in unserer Programmschleife bloss noch prüfen, ob eventuell ein Zeichen empfangen wurde. Dies wird vom **UART** im **Status Register** angezeigt, indem das **RXC** Bit gesetzt wird. Nun müssen wir möglichst schnell das **UART Daten Register** auslesen, damit das nächste Zeichen empfangen werden kann.

```
char ch; // Variable zur Speicherung des gelesenen Zeichens
```

```
for (;;) { // Endlosschleife
    if (inp (USR) & (1 << RXC)) { // Bit RXC im USR gesetzt ?
        ch = inp (UDR); // Datenregister auslesen
    }
    ...
    ...
}
```

Wenn wir sicher sein wollen, dass wir auch alle Zeichen empfangen haben müssen wir jeweils noch das **OverRun** Bit testen. Wenn wir nämlich zu lange warten mit dem Auslesen des Datenregisters kann es passieren, dass bereits wieder ein Zeichen in das Empfangs-Schieberegister eingelesen wurde. Da aber das Datenregister noch belegt ist schmeisst der UART das neue Zeichen weg und setzt das **OR** Bit im Status Register.

```

char ch;    // Variable zur Speicherung des gelesenen Zeichens

for (;;) {
    if (inp (USR) & (1 << RXC)) {    // Endlosschleife
        ch = inp (UDR);                // Bit RXC im USR gesetzt ?
        if (inp (USR) & (1 << OR)) { // Datenregister auslesen
            // OverRun Bit gesetzt ?
            // Autsch, jetzt ist
            // was verloren gegangen
            ...
        }
    }
    ...
    ...
}

```

10.2 AVR's ohne UART

Wer jetzt glaubt, eine serielle Schnittstelle könne nur mit denjenigen AVR's realisiert werden welche auch einen UART an Bord haben, der hat sich geschnitten.

Selbstverständlich können wir mit einer entsprechenden Software eine serielle Schnittstelle auf 2 beliebigen Pin's des Controllers realisieren. Allerdings ist der Aufwand für eine entsprechende Software schon erheblich. Beispiele dafür findet ihr auf den einschlägigen Seiten im Internet. Man spricht in diesem Fall von einem Software-UART.

Mit dieser Methode ist es auch möglich, einen AVR mit mehreren seriellen Schnittstellen zu versehen.

10.3 Zusammenfassung

Wir haben jetzt gesehen, wie wir mit dem UART ohne Verwendung der Interrupts Daten senden und empfangen können.

Wenn wir die Interrupts zur Hilfe nehmen geht das ganze noch viel komfortabler, aber dazu später mehr.

11 ÜBUNG 2, EMPFANGEN MIT DEM UART

In dieser Übung wollen wir Daten mit Hilfe des UART empfangen.

11.1 Benötigte Bauteile

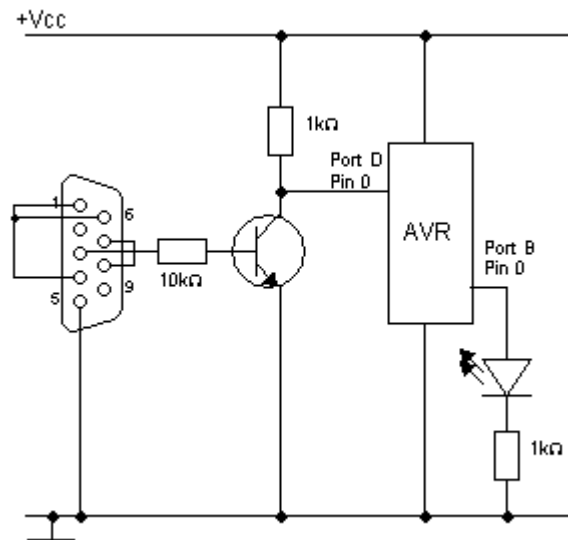
Für den Aufbau der Testschaltung benötigen wir nebst dem Experimentierboard mit eingesetztem AVR Controller folgende Bauteile:

- 1 Transistor, NPN-Allerweltstyp
- 1 Widerstand 10 k Ω
- 2 Widerstände 1 k Ω
- 1 LED
- 1 Sub-D Buchse 9pol W
- ca. 2m Signalkabel 2-Adrig

11.2 Die Schaltung

Da wir uns den Einsatz eines Pegelwandlers mit Inverters (MAX232 o.ä.) sparen wollen müssen wir das Signal vom Partnergerät invertieren bevor wir es an den Eingang des UART weiterleiten. Dazu dient der Transistor, dessen Basis über den $10\text{k}\Omega$ Vorwiderstand angesteuert wird. Am Kollektor greifen wir das invertierte Signal ab und leiten es zum Eingang des UART.

Wer es ganz genau haben will kann aber natürlich auch einen integrierten Baustein wie den MAX232 einsetzen.



Die Drahtbrücken in der Sub-D Buchse dienen der Simulation der Hardware-Handshake Signale.

11.2.1 AUFGABE

Die LED wird über die Tastatur am PC gesteuert. Dazu wird ein Terminalprogramm am PC gestartet. Wird die Taste **1** gedrückt so soll die LED leuchten. Wenn die Taste **0** gedrückt wird erlischt die LED.

Die Baudrate sollte auf 9600 Baud eingestellt werden. Die Einstellungen am Terminalprogramm sind 8 Datenbits, 1 Stopbit, Keine Parität.

So, und jetzt bitte schön an die Arbeit.

Wenn es gar nicht klappen sollte, dann habt ihr hier den [C-Quellcode](#) und das [Makefile](#) meiner Musterlösung.

11.3 Weitere Übungen

Wenn euch das Fieber gepackt hat könnt ihr euch auch eigene Übungen ausdenken.

Wie wäre es zum Beispiel mit einem Schnittstellenwandler von Parallel nach Seriell?

Die Musiker unter euch könnten eine MIDI-gesteuerte Lichtorgel entwerfen etc., etc. pp.

12 ANALOGE EIN- UND AUSGABE

Leider können wir mit unseren Controllern keine analogen Werte direkt verarbeiten. Dazu bedarf es jeweils einer Umwandlung des analogen Signals in einen digitalen Zahlenwert. Je nachdem, ob wir Daten einlesen oder ausgeben wollen reden wir dabei von **DAC** oder **ADC**.

12.1 ADC (Analog Digital Converter)

Der **ADC** wandelt analoge Signale in digitale Werte um welche vom Controller interpretiert werden können. Einige AVR-Typen haben bereits einen oder mehrere solcher **ADC**'s eingebaut, bei den kleineren AVR's müssen wir uns anders aus der Affäre ziehen. Die Genauigkeit, mit welcher ein analoges Signal aufgelöst werden kann wird durch die Auflösung des **ADC** in Anzahl Bits angegeben, man hört bzw. liest jeweils von 8-Bit **ADC** oder 10-Bit **ADC** oder noch höher. Ein **ADC** mit 8 Bit Auflösung kann somit das analoge Signal mit einer Genauigkeit von 1/255 des Maximalwertes darstellen. Wenn wir nun mal annehmen, wir hätten eine Spannung zwischen 0 und 5 Volt und eine Auflösung von 3 Bit, dann könnten die Werte 0V, 0.625V, 1.25, 1.875V, 2.5V, 3.125V, 3.75, 4.375, 5V daherkommen, siehe dazu folgende Tabelle:

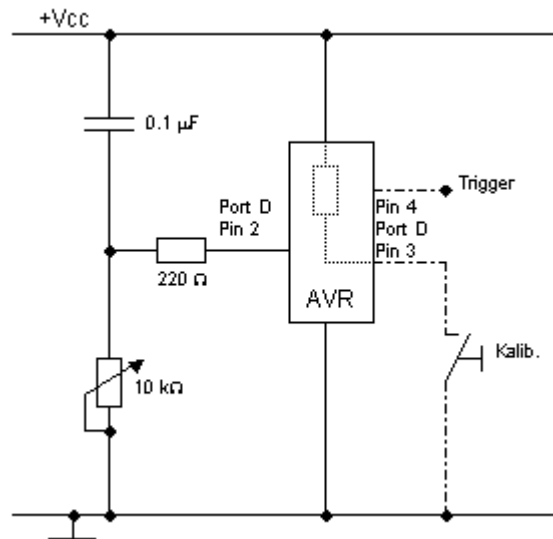
Eingangsspannung am ADC	Entsprechender Messwert
0...0.625V	0
0.625...1.25V	1
1.25...1.875V	2
1.875...2.5V	3
2.5...3.125V	4
3.125...3.75V	5
3.75...4.375V	6
4.375...5V	7
5V	8

Die Angaben sind natürlich nur ungefähr. Je höher nun die Auflösung des **ADC** ist, also je mehr Bits er hat, um so genauer kann der Wert erfasst werden.

12.1.1 MESSEN EINES WIDERSTANDES

Wir wollen hier einmal die wohl einfachste Methode zur Erfassung eines analogen Wertes realisieren und zwar das Messen eines veränderlichen Widerstandes wie z.B. eines Potentiometers.

Man stelle sich vor, wir schalten einen Kondensator in Reihe zu einem Widerstand zwischen die Versorgungsspannung und Masse und dazwischen nehmen wir das Signal ab und führen es auf einen der Pins an unserem Controller, genau so wie es in folgender Grafik dargestellt ist.



Wenn wir nun den Pin des AVR als Ausgang schalten und auf Logisch 1 (HIGH) legen, dann liegt an beiden Platten des Kondensators **Vcc** an und dieser wird entladen (Klingt komisch, mit **Vcc** entladen, ist aber so).

Nachdem nun der Kondensator genügend entladen ist schalten wir einfach den Pin als Eingang wodurch dieser hochohmig wird. Der Kondensator lädt sich jetzt über das Poti auf, dabei steigt der Spannungsabfall über dem Kondensator und derjenige über dem Poti sinkt. Fällt nun der Spannungsabfall über dem Poti unter die Thresholdspannung des Eingangspins ($2/5 V_{cc}$, also ca. 2V), dann schaltet der Eingang von HIGH auf LOW um. Wenn wir nun messen (zählen), wie lange es dauert, bis der Kondensator so weit geladen ist, dann haben wir einen ungefähren Wert der Potentiometerstellung.

Der 220 Ohm Widerstand dient dem Schutz des Controllers. Wenn nämlich sonst die Potentiometerstellung auf Maximum steht (0 Ohm), dann würde in den Eingang des Controllers ein viel zu hoher Strom fließen und der AVR würde in Rauch aufgehen.

Dies ist meines Wissens die einzige Schaltung zur Erfassung von Analogwerten, welche mit nur einem einzigen Pin auskommt.

Mit einem weiteren Eingangspin und ein wenig Software können wir auch eine Kalibrierung realisieren, um den Messwert in einen vernünftigen Bereich (z.B: 0...100 % oder so) umzurechnen.

Wer Lust hat, sich selber mal an ein solches Programm heranzuwagen, der sollte das jetzt tun. Für diejenigen, die es gern schnell mögen, hier das Beispielprogramm, welches den UART-Printf aus den vorangegangenen Kapiteln benötigt, incl. Makefile:

- [Poti.c](#) Hauptprogramm.
- [Pot.c](#) Separate Routine zur Ermittlung des Messwertes.
- [Pot.h](#) Zugehörige Headerdatei.
- [UartPrintF.c](#) Für die Debugausgabe auf den UART.
- [UartPrintF.h](#) Zugehörige Headerdatei.
- [Makefile](#) Makefile.

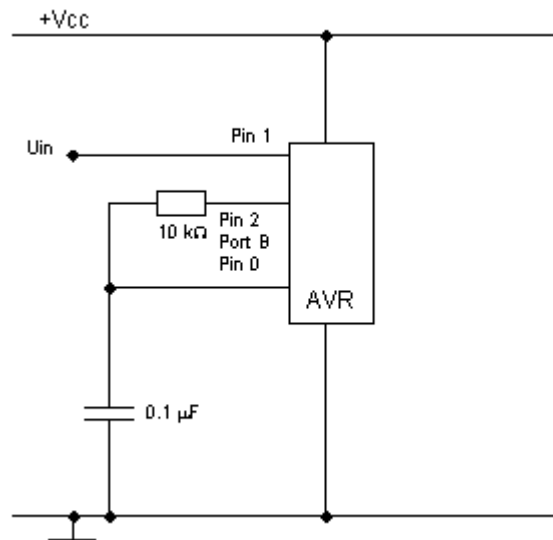
Nachdem das Programm auf den AVR geladen wurde muss dieser kalibriert werden. Dazu wird der Kalibrierungsschalter geschlossen und das Poti einige Male zwischen minimaler und maximaler Stellung hin und her gedreht. Dabei werden die jeweiligen Maximalwerte bestimmt. Wenn der Kalibrierschalter wieder geöffnet wird werden die Kalibrierungsdaten in's EEPROM des AVR geschrieben, damit die Prozedur nicht nach jedem Reset wiederholt werden muss.

Auf Pin 4 habe ich noch ein Triggersignal gelegt, welches auf HIGH geht wenn die Messung beginnt und auf LOW, wenn der Messvorgang beendet wird. Mit Hilfe dieses Signals kann der Vorgang wunderschön auf einem Oszillographen dargestellt werden.

12.1.2 ADC ÜBER KOMPARATOR

Es gibt einen weiteren Weg, eine analoge Spannung mit Hilfe des Komparators, welcher in fast jedem AVR integriert ist, zu messen. Siehe dazu auch die Application Note AVR400 von Atmel.

Dabei wird das zu messende Signal auf den invertierenden Eingang des Komparators geführt. Zusätzlich wird ein Referenzsignal an den nicht invertierenden Eingang des Komparators angeschlossen. Das Referenzsignal wird hier auch wieder über ein RC-Glied erzeugt, allerdings mit festen Werten für R und C.



Das Prinzip der Messung ist nun dem vorhergehenden recht ähnlich. Durch Anlegen eines LOW-Pegels an Pin 2 wird der Kondensator zuerst einmal entladen. Auch hier muss darauf geachtet werden, dass der Entladevorgang genügend lang dauert.

Nun wird Pin 2 auf HIGH gelegt. Der Kondensator wird geladen. Wenn die Spannung über dem Kondensator die am Eingangspin anliegende Spannung erreicht hat schaltet der Komparator durch. Die Zeit, welche benötigt wird, um den Kondensator zu laden kann nun auch wieder als Mass für die Spannung an Pin 1 herangezogen werden.

Ich habe es mir gespart, diese Schaltung auch aufzubauen und zwar aus mehreren Gründen:

1. 3 Pins notwendig.
2. Genauigkeit vergleichbar mit einfacherer Lösung.
3. War einfach zu faul.

Der Vorteil dieser Schaltung liegt allerdings darin, dass damit direkt Spannungen gemessen werden können.

12.1.3 DER ADC IM AVR

Wenn es einmal etwas genauer sein soll, dann müssen wir auf einen AVR mit eingebautem(n) ADC-Wandler zurückgreifen. Wir wollen hier einmal den AT90S8535 besprechen, welcher über 8 ADC-Kanäle verfügt.

Die Umwandlung innerhalb des AVR basiert auf der Schrittweisen Näherung. Beim AVR müssen die Pins **AGND** und **AVCC** beschaltet werden mit **0V** bzw. **Vcc**. Warum der das nicht intern macht ist mir eigentlich auch nicht so ganz klar. Zusätzlich muss eine externe Referenzspannung im Bereich von **2V** bis **Vcc** an **AREF** angelegt werden. Die zu messende Spannung muss nun im Bereich zwischen **AGND** und **AREF** liegen.

Der **ADC** kann in zwei verschiedenen Betriebsarten verwendet werden.

12.1.3.1 Einfache Wandlung (Single Conversion)

In dieser Betriebsart wird der Wandler bei Bedarf vom Programm angestossen für jeweils eine Messung.

12.1.3.2 Frei laufend (Free Running)

In dieser Betriebsart erfasst der Wandler permanent die anliegende Spannung und schreibt diese in das **ADC Data Register**.

12.1.3.3 Die Register des ADC

Der **ADC** verfügt über eigene Register, welche hier aufgelistet werden:

ADCSR	ADC Control and Status Register.								
	In diesem Register stellen wir ein, wie wir den ADC verwenden möchten.								
	Das Register ist wie folgt aufgebaut:								
	Bit	7	6	5	4	3	2	1	0
	Name	ADEN	ADSC	ADFR	ADIF	ADIE	ADPS2	ADPS1	ADPS0
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	W
Initialwert	0	0	0	0	0	0	1	0	
ADEN	ADC Enable Dieses Bit muss gesetzt werden, um den ADC überhaupt zu aktivieren. Wenn das Bit nicht gesetzt ist können die Pin's wie normale I/O-Pins verwendet werden.								
ADSC	ADC Start Conversion Mit diesem Bit wird ein Messvorgang gestartet. In der frei laufenden Betriebsart muss das Bit gesetzt werden, um die kontinuierliche Messung zu aktivieren. Wenn das Bit nach dem Setzen des ADEN -Bits zum ersten Mal gesetzt wird führt der Controller zuerst eine zusätzliche Wandlung und erst dann die eigentliche Wandlung aus. Diese zusätzliche Wandlung wird zu Initialisierungszwecken durchgeführt. Das Bit bleibt nun so lange auf 1, bis die Umwandlung abgeschlossen ist, im Initialisierungsfall entsprechend bis die zweite Umwandlung erfolgt ist und geht danach auf 0.								
ADFR	ADC Free Running Select Mit diesem Bit wird die Betriebsart eingestellt. Eine logische 1 aktiviert den frei laufenden Modus. Der ADC misst nun ständig den ausgewählten Kanal und schreibt den gemessenen Wert in das ADC Data Register .								
ADIF	ADC Interrupt Flag Dieses Bit wird vom ADC gesetzt wenn eine Umwandlung erfolgt und das ADC Data Register aktualisiert ist. Wenn das ADIE Bit sowie das I-Bit im AVR Statusregister gesetzt ist wird der ADC Interrupt ausgelöst und die Interrupt-Behandlungsroutine aufgerufen.. Das Bit wird automatisch gelöscht wenn die Interrupt-Behandlungsroutine aufgerufen wird. Es kann jedoch auch gelöscht werden, indem ein logisches 1 in das Register geschrieben wird (So steht's in der AVR-Doku).								
ADIE	ADC Interrupt Enable Wenn dieses Bit gesetzt ist und ebenso das I-Bit im Statusregister SREG ,								

dann wird der **ADC-Interrupt** aktiviert.

ADPS2 **ADC Prescaler Select Bits**
 ...
ADPS0 Diese Bits bestimmen den Teilungsfaktor zwischen der Taktfrequenz und dem Eingangstakt des **ADC**.
 Der **ADC** benötigt einen eigenen Takt, welchen er sich selber aus der CPU-Taktfrequenz erzeugt. Der **ADC**-Takt sollte zwischen 50 und 200kHz sein.
 Der Vorteiler muss also so eingestellt werden, dass die CPU-Taktfrequenz dividiert durch den Teilungsfaktor einen Wert zwischen 50-200kHz ergibt. Bei einer CPU-Taktfrequenz von 4MHz beispielsweise rechnen wir
 $TF_{min} = CLK / 200kHz = 4000000 / 200000 = 20$
 $TF_{max} = CLK / 50kHz = 4000000 / 50000 = 80$
 Somit kann hier der Teilungsfaktor 32 oder 64 verwendet werden. Im Interesse der schnelleren Wandlungszeit werden wir hier den Faktor 32 einstellen.

ADPS2	ADPS1	ADPS0	Teilungsfaktor
0	0	0	2
0	0	1	2
0	1	0	4
0	1	1	8
1	0	0	16
1	0	1	32
1	1	0	64
1	1	1	128

ADCL
ADCH

ADC Data Register
 Wenn eine Umwandlung abgeschlossen ist befindet sich der gemessene Wert in diesen beiden Registern.
 Von **ADCH** werden nur die beiden niederwertigsten Bits verwendet.
 Es müssen immer beide Register ausgelesen werden und zwar immer in der Reihenfolge **ADCL**, **ADCH**. Der effektive Messwert ergibt sich dann zu
 $x = ADCL + ADCH * 256$

ADMUX

ADC Multiplexer Select Register
 Mit diesem Register wird der zu messende Kanal ausgewählt. Beim 90S8535 kann jeder Pin von Port A als **ADC**-Eingang verwendet werden (=8 Kanäle).
 Das Register ist wie folgt aufgebaut:

Bit	7	6	5	4	3	2	1	0
Name	-	-	-	-	-	MUX2	MUX1	MUX0
R/W	R	R	R	R	R	R/W	R/W	R/W
Initialwert	0	0	0	0	0	0	0	0

MUX2 Mit diesem 3 Bits wird der zu messende Kanal bestimmt. Es wird einfach die entsprechende Pinnummer des Ports eingeschrieben.
 ...
MUX0 Wenn das Register beschrieben wird, während dem eine Umwandlung läuft, so wird zuerst die aktuelle Umwandlung auf dem bisherigen Kanal beendet. Dies ist vor allem beim frei laufenden Betrieb zu berücksichtigen.
 Meine Empfehlung ist deswegen klar diese, dass der frei laufende Betrieb nur bei einem einzelnen zu verwendenden Analogeingang verwendet werden sollte, wenn man sich Probleme bei der Umschalterei ersparen will.

12.1.3.4 Aktivieren des ADC

Um den **ADC** zu aktivieren müssen wir das **ADEN**-Bit im **ADCSR**-Register setzen. Im gleichen Schritt legen wir auch gleich die Betriebsart fest. Wenn wir frei laufend arbeiten wollen setzen wir zusätzlich das **ADFR**-Bit. Bei Bedarf wird auch gleich der Teilungsfaktor eingestellt.

```
// Teilungsfaktor auf 8 und ADC aktivieren
// Nicht frei laufend
outp ((1<<ADEN) & 3, ADCSR);
```

Um nun eine einzelne Messung, sagen wir mal an Pin 3, durchzuführen schalten wir den Kanal ein, starten die Wandlung und warten, bis der **ADC** die Beendigung meldet:

```
outp ((1<<PINB3), ADMUX);           // Kanal an Pin 3 auswählen
sbi (ADCSR, ADSC);
while (bit_is_set (ADCSR, ADSC)) /* Nur warten */;
x = __inw (ADCL);                   // Wert auslesen
```

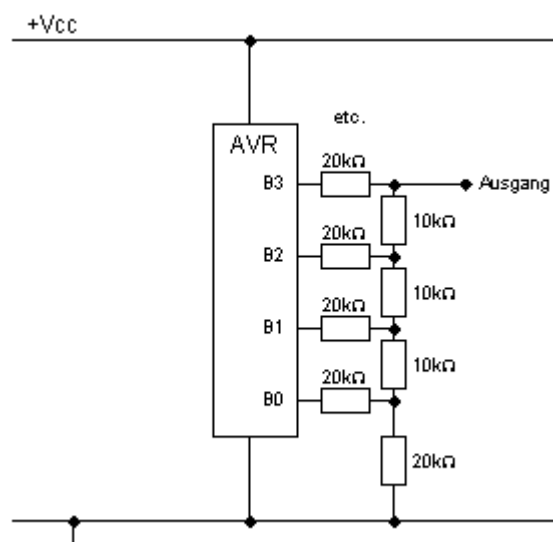
Da ich leider bisher noch kein Experimentierboard für den 8535 gebastelt habe kann ich euch auch keine erprobte Übung anbieten.

12.2 DAC (Digital Analog Converter)

Mit Hilfe eines **DAC** können wir nun auch Analogsignale ausgeben. Es gibt hier mehrere Verfahren. Wenn wir beim **ADC** die Möglichkeit haben, mit externen Komponenten zu operieren müssen wir bei der **DAC**-Wandlung mit dem auskommen, was der Controller selber zu bieten hat.

12.2.1 DAC ÜBER MEHRERE DIGITALE AUSGÄNGE

Wenn wir an den Ausgängen des Controllers ein entsprechendes Widerstandsnetzwerk aufbauen haben wir die Möglichkeit, durch die Ansteuerung der Ausgänge über den Widerständen einen Addierer aufzubauen, mit dessen Hilfe wir eine dem Zahlenwert proportionale Spannung erzeugen können. Das Schaltbild dazu kann etwa so aussehen:



Es sollten selbstverständlich möglichst genaue Widerstände verwendet werden, also nicht unbedingt solche mit einer Toleranz von 10% oder mehr. Weiterhin empfiehlt es sich, je nach Anwendung den Ausgangsstrom über einen Operationsverstärker zu verstärken.

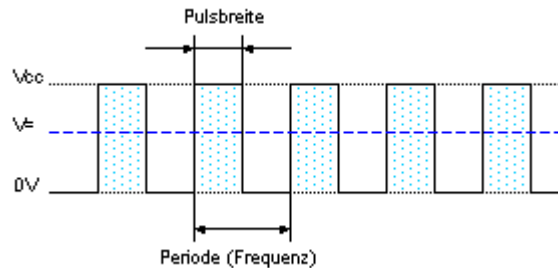
12.2.2 PWM (PULSWEITENMODULATION)

Wir kommen nun zu einem Thema, welches in aller Munde ist, aber viele Anwender verstehen nicht ganz, wie **PWM** eigentlich funktioniert.

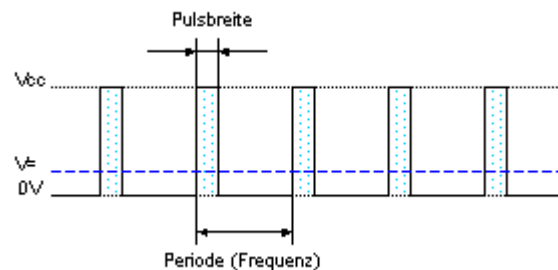
Wie wir alle wissen ist ein Microcontroller ein rein digitales Bauteil. Definieren wir einen Pin als Ausgang, dann können wir diesen Ausgang entweder auf HIGH setzen worauf am Ausgang die Versorgungsspannung **V_{cc}** anliegt, oder aber wir setzen den Ausgang auf LOW wonach dann **0V** am Ausgang liegt.

Was passiert aber nun, wenn wir periodisch mit einer festen Frequenz zwischen HIGH und LOW umschalten?

Richtig, wir erhalten eine Rechteckspannung, wie die folgende Abbildung zeigt:



Diese Rechteckspannung hat nun einen geometrischen Mittelwert, der je nach Pulsbreite kleiner oder grösser ist.



Wenn wir nun diese pulsierende Ausgangsspannung noch über ein RC-Glied filtern dann haben wir schon eine entsprechende Gleichspannung erzeugt.

Mit den AVR's können wir direkt **PWM**-Signale erzeugen. Dazu dient der 16-Bit Zähler, welcher im sogenannten **PWM**-Modus betrieben werden kann.

In den folgenden Überlegungen wird als Controller der 90S2313 vorausgesetzt. Die Hinweis: Theorie ist allerdings bei anderen AVR-Controllern vergleichbar, die Pinbelegung allerdings nicht unbedingt.

Um den **PWM**-Modus zu aktivieren müssen im Timer/Counter1 Control Register A **TCCR1A** die Pulsweiten-Modulatorbits **PWM10** bzw. **PWM11** entsprechend nachfolgender Tabelle gesetzt werden:

PWM11	PWM10	Bedeutung
0	0	PWM-Modus des Timers ist nicht aktiv.
0	1	8-Bit PWM.
1	0	9-Bit PWM.
1	1	10-Bit PWM.

Der Timer/Counter zählt nun permanent von 0 bis zur Obergrenze und wieder zurück, er wird also als sogenannter Auf-/Ab Zähler betrieben. Die Obergrenze hängt davon ab, ob wir mit 8, 9 oder 10-Bit PWM arbeiten wollen:

Auflösung	Obergrenze	Frequenz
8	255	$f_{TC1} / 510$
9	511	$f_{TC1} / 1022$
10	1023	$f_{TC1} / 2046$

Zusätzlich muss mit den Bits **COM1A1** und **COM1A0** desselben Registers die gewünschte Ausgabeart des Signals definiert werden:

COM1A1	COM1A0	Bedeutung
0	0	Keine Wirkung, Pin wird nicht geschaltet.
0	1	Keine Wirkung, Pin wird nicht geschaltet.
1	0	Nicht invertierende PWM. Der Ausgangspin wird gelöscht beim Hochzählen und gesetzt beim Herunterzählen.
1	1	Invertierende PWM. Der Ausgangspin wird gelöscht beim Herunterzählen und gesetzt beim Hochzählen.

Der entsprechende Befehl um beispielsweise den Timer/Counter als nicht invertierenden 10-Bit PWM zu verwenden heisst dann:

```
outp ((1<<PWM11) | (1<<PWM10) | (1<<COM1A1), TCCR1A);
```

Damit der Timer/Counter überhaupt läuft müssen wir im Control Register B **TCCR1B** noch den gewünschten Takt (Vorzähler) einstellen, und somit auch die Frequenz des **PWM**-Signals bestimmen.

CS12	CS11	CS10	Bedeutung
0	0	0	Stop. Der Timer/Counter wird gestoppt.
0	0	1	CK
0	1	0	CK / 8
0	1	1	CK / 64
1	0	0	CK / 256
1	0	1	CK / 1024
1	1	0	Externer Pin 1, negative Flanke
1	1	1	Externer Pin 1, positive Flanke

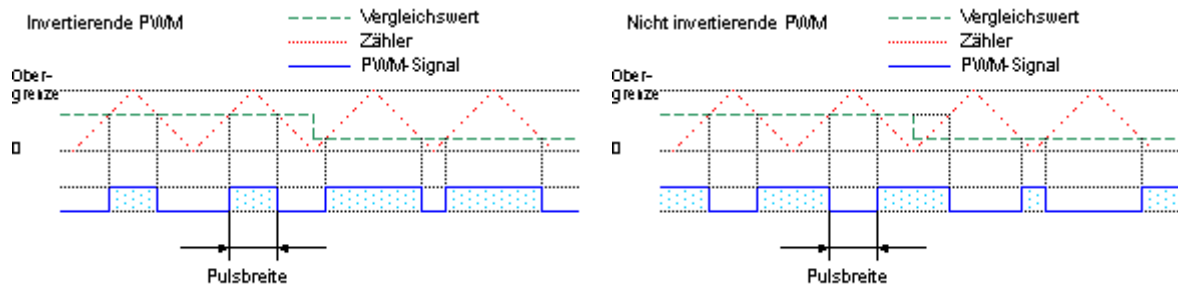
Also, um einen Takt von CK / 1024 zu generieren verwenden wir folgenden Befehl:

```
outp ((1<<CS12) | (1<<CS10), TCCR1B);
```


Jetzt muss nur noch der Vergleichswert festgelegt werden. Diesen schreiben wir in das 16-Bit Timer/Counter Output Compare Register **OCR1A**. Wir können dazu den von der Bibliothek zur Verfügung gestellten Befehl zum Schreiben von 16-Bit Registern verwenden, als Portadresse wird das Low-Byte des Ports verwendet.

```
__outw (xxx, OCR1AL);
```

Die folgende Grafik soll den Zusammenhang zwischen dem Vergleichswert und dem generierten **PWM-Signal** aufzeigen.

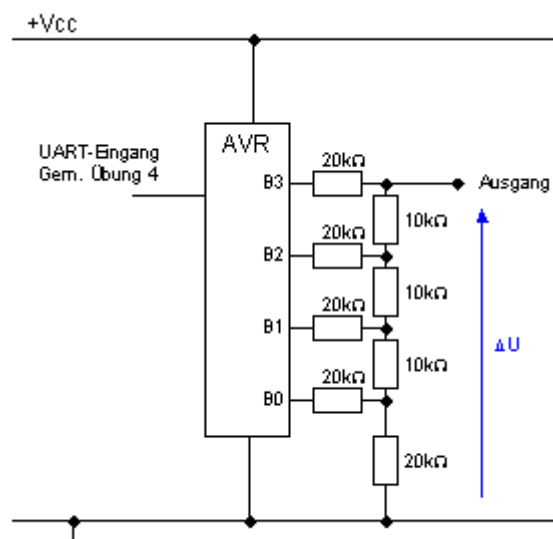


Ach ja, fast hätte ich's vergessen. Das generierte **PWM-Signal** wird am Output Compare Pin **OC1** des Timers ausgegeben und leider können wir deshalb auch nur ein einzelnes **PWM-Signal** mit dieser Methode generieren.

13 ÜBUNG 5, DAC ÜBER WIDERSTANDSNETZWERK

Es soll eine analoge Spannung auf Port B erzeugt werden, der einem Zahlenwert entspricht, welcher dem AVR über den UART zugeführt wird.

Der auszugebende Wert soll auf dem PC in einem Terminalprogramm (z.B. HyperTerminal) eingegeben werden als hexadezimale Ziffer '0' bis 'F'. Damit können also 16 Stufen ausgegeben werden. Wir benötigen also eine 4-Bit Wandlung. Der entsprechende Schaltplan sieht wie folgt aus:



Wird eine der gewünschten Ziffern auf dem UART empfangen, so soll der entsprechende Zahlenwert auf Port B ausgegeben werden.

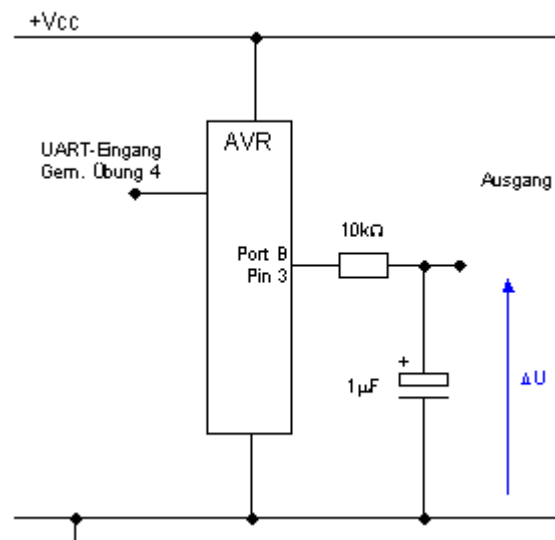
Hinweis Die maximale Spannung, die am Ausgang erzeugt werden kann ist etwas kleiner als V_{cc} .

Versuche das Programm möglichst kompakt zu schreiben. Hier ist die [Musterlösung](#) mit entsprechendem [Makefile](#).

Versuche, über das Terminalprogramm eine Textdatei an den AVR zu senden, um eine Sägezahnspannung oder eine Dreiecksspannung am Ausgang zu erzeugen. In HyperTerminal kann dazu die Funktion *Übertragung->Textdatei senden* verwendet werden.

14 ÜBUNG 6, PWM

Die Funktionalität soll dieselbe sein wie bei Übung 5. Allerdings wollen wir diesmal das Analogsignal mittels Pulsweitenmodulation erzeugen und mit einem RC-Glied filtern.



Die Werte für den Widerstand und den Kondensator hängen davon ab, wie viel Strom mit dem PWM-Signal geliefert werden soll. Es ist empfehlenswert, den Strom durch Nachschalten eines Operationsverstärkers zu erhöhen.

Ebenso hängt der Wert des Kondensators natürlich auf von der eingestellten Zählfrequenz (Vorteiler) ab.

Der Vergleichswert für den Timer/Counter muss natürlich entsprechend der PWM-Auflösung hochgerechnet werden.

Es gilt zu beachten, dass wir mit dieser Methode keine schnellen Änderungen der Ausgangsspannung erzeugen können, da der Kondensator jeweils eine bestimmte Zeit braucht, bis er sich wieder eingepegelt hat. Ganz einfach gesagt gilt:

Je höher die Kapazität des Kondensators, um so linearer, jedoch um so träger reagiert die Ausgangsspannung.

Auch hier gibt es wieder eine [Musterlösung](#) mit [Makefile](#).

15 DIE TIMER/COUNTER DES AVR

Die heutigen Microcontroller und insbesondere die RISC-AVR's sind für viele Steuerungsaufgaben natürlich viel zu schnell. Wenn wir beispielsweise eine LED oder Lampe blinken lassen wollen können wir selbstverständlich nicht die CPU-Frequenz verwenden da ja dann nichts mehr vom Blinken zu bemerken wäre.

Wir brauchen also eine Möglichkeit, die Taktfrequenz auf vernünftige Werte herunter zu brechen. Selbstverständlich sollte die resultierende Frequenz dann auch noch einigermaßen genau und stabil sein.

Hier kommen die im AVR vorhandenen Timer/Counter zum Einsatz.

Ein anderes Anwendungsgebiet ist die Zählung von Signalen, welche über einen I/O-Pin zugeführt werden können.

Die folgenden Ausführungen beziehen sich auch den AT90S2313. Für andere Modelltypen müsst ihr euch die allenfalls notwendigen Anpassungen aus den Datenblättern der entsprechenden Controller raus lesen.

Wir unterscheiden grundsätzlich zwischen 8-Bit Timern, welche eine Auflösung von 256 aufweisen und 16-Bit Timern mit (logischerweise) einer Auflösung von 65536.

Als Eingangstakt für die Timer/Counter kann entweder die CPU-Taktfrequenz, der Vorteiler-Ausgang oder ein an einen I/O-Pin angelegtes Signal verwendet werden. Wenn ein externes Signal verwendet wird, so darf dessen Frequenz nicht höher sein als die Hälfte des CPU-Taktes.

15.1.1 DER VORZÄHLER (PRESCALER)

Beide Timer/Counter werden im Timerbetrieb über den gleichen Vorzähler versorgt. Der Vorzähler dient dazu, den CPU-Takt vorerst mal runter zu brechen auf eine wählbare Teilung. Die so geteilte Frequenz wird den Eingängen der Timer zugeführt. Wenn wir mit einer einem CPU-Takt von 4 MHz arbeiten und den Vorteiler auf 1024 einstellen wird also der Timer mit einer Frequenz von $4 \text{ MHz} / 1024$, also mit ca. 4 kHz versorgt. Wenn also der Timer läuft, so wird das Daten- bzw. Zählregister mit dieser Frequenz inkrementiert.

15.2 8-Bit Timer/Counter

Alle AVR-Modelle weisen mindestens einen, teilweise sogar zwei 8-Bit Timer auf.

Der 8-Bit Timer wird über folgende Register angesprochen:

TCCR0	Timer/Counter Control Register Timer 0								
	In diesem Register stellen wir ein, wie wir den Timer/Counter verwenden möchten. Das Register ist wie folgt aufgebaut:								
	Bit	7	6	5	4	3	2	1	0
	Name	-	-	-	-	-	CS02	CS01	CS00
	R/W	R	R	R	R	R	R/W	R/W	R/W
	Initialwert	0	0	0	0	0	0	0	0
	CS02	Clock Select Bits							
	CS01	Diese 3 Bits bestimmen die Quelle für den Timer/Counter:							
	CS00	CS02	CS01	CS00	Resultat				
		0	0	0	Stopp, Der Timer/Counter wird angehalten.				
	0	0	1	CPU-Takt					
	0	1	0	CPU-Takt / 8					
	0	1	1	CPU-Takt / 64					
	1	0	0	CPU-Takt / 256					
	1	0	1	CPU-Takt / 1024					
	1	1	0	Externer Pin TO , fallende Flanke					
	1	1	1	Externer Pin TO , steigende Flanke					
	Wenn als Quelle der externe Pin TO verwendet wird, so wird ein Flankenwechsel auch erkannt, wenn der Pin TO als Ausgang geschaltet ist.								
TCNT0	Timer/Counter Daten Register Timer 0								
	Dieses ist als 8-Bit Aufwärtszähler mit Schreib- und Lesezugriff realisiert. Wenn der Zähler den Wert 255 erreicht hat beginnt er beim nächsten Zyklus wieder bei 0.								
	Bit	7	6	5	4	3	2	1	0
	Name	MSB							LSB
	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
	Initialwert	0	0	0	0	0	0	0	0

Um nun also den Timer0 in Betrieb zu setzen und ihn mit einer Frequenz von 1/64-tel des CPU-Taktes zählen zu lassen schreiben wir die folgende Befehlszeile:

```
outp ((1<<CS01) | (1<<CS00), TCCR0);
```

Der Zähler zählt nun von 0 an aufwärts bis 255 um dann wieder bei 0 zu beginnen. Bei jedem Überlauf von 255 auf 0 wird das Timer Overflow Flag **TOV0** im Timer Interrupt Flag **TIFR** Register gesetzt und, falls so konfiguriert, ein entsprechender Interrupt ausgelöst.

15.3 16-Bit Timer/Counter

Viele AVR-Modelle besitzen ausser den 8-Bit Timers auch einen oder sogar zwei (einige ATMega-Modelle) 16-Bit Timer.

Die 16-Bit Timer/Counter sind wesentlich komplexer aufgebaut als die 8-Bit Timer/Counter, bieten dafür aber auch viel mehr Möglichkeiten, als da sind:

- [Erzeugung eines pulswidenmodulierten Ausgangssignals \(PWM\)](#).

- [Vergleichswert-Überprüfung](#) mit Erzeugung eines Ausgangssignals (Output Compare Match).
- [Einfangen eines Eingangssignals](#) mit Speicherung des aktuellen Zählerwertes (Input Capturing), mit zuschaltbarer Rauschunterdrückung (Noise Filtering).

Folgende Register sind dem Timer/Counter 1 zugeordnet:

TCCR1A	Timer/Counter Control Register A Timer 1								
	In diesem und dem folgenden Register stellen wir ein, wie wir den Timer/Counter verwenden möchten.								
	Das Register ist wie folgt aufgebaut:								
	Bit	7	6	5	4	3	2	1	0
	Name	COM1A1	COM1A0	-	-	-	-	PWM11	PWM10
	R/W	R/W	R/W	R	R	R	R	R/W	R/W
	Initialwert	0	0	0	0	0	0	0	0
	COM1A1	Compare Match Control Bits							
	COM1A0	Diese 2 Bits bestimmen die Aktion, welche am Output-Pin OC1 ausgeführt werden soll, wenn der Wert des Datenregisters des Timer/Counter 1 den Wert des Vergleichsregisters erreicht, also ein so genannter Compare Match auftritt.							
		Der Pin OC1 (PB3 beim 2313) muss mit dem Datenrichtungsregister als Ausgang konfiguriert werden.							
	COM1A1	COM1A0	Resultat						
	0	0	Output-Pin OC1 wird nicht angesteuert.						
	0	1	Das Signal am Pin OC1 wird invertiert (Toggle).						
	1	0	Der Output Pin OC1 wird auf 0 gesetzt.						
	1	1	Der Output Pin OC1 wird auf 1 gesetzt.						
	In der PWM-Betriebsart haben diese Bits eine andere Funktion.								
	COM1A1	COM1A0	Resultat						
	0	0	Output-Pin OC1 wird nicht angesteuert.						
	0	1	Output-Pin OC1 wird nicht angesteuert.						
	1	0	Wird beim Hochzählen der Wert im Vergleichsregister erreicht so wird der Pin OC1 auf 0 gesetzt. Wird beim Herunterzählen der Wert im Vergleichsregister erreicht so wird der Pin auf 1 gesetzt. Man nennt dies nicht invertierende PWM.						
	1	1	Wird beim Hochzählen der Wert im Vergleichsregister erreicht so wird der Pin OC1 auf 1 gesetzt. Wird beim Herunterzählen der Wert im Vergleichsregister erreicht so wird der Pin auf 0 gesetzt. Man nennt dies invertierende PWM.						
PWM11	PWM Mode Select Bits								
PWM10	Mit diesen 2 Bits wird die PWM-Betriebsart des Timer/Counter 1 gesteuert.								
	PWM11	PWM10	Resultat						
	0	0	Die PWM-Betriebsart ist nicht aktiviert. Timer/Counter 1 arbeitet als normaler Timer bzw. Zähler.						
	0	1	8-Bit PWM Betriebsart aktivieren.						
	1	0	9-Bit PWM Betriebsart aktivieren.						

		1	1	10-Bit PWM Betriebsart aktivieren.																																								
TCCR1B	Timer/Counter Control Register B Timer 1																																											
Bit	7	6	5	4	3	2	1	0																																				
Name	ICNC1	ICES1	-	-	CTC1	CS12	CS11	CS10																																				
R/W	R/W	R/W	R	R	R/W	R/W	R/W	R/W																																				
Initialwert	0	0	0	0	0	0	0	0																																				
ICNC1	<p>Input Capture Noise Canceler (4 CKs) Timer/Counter 1 oder auf Deutsch Rauschunterdrückung des Eingangssignals. Wenn dieses Bit gesetzt ist und mit dem Input Capture Signal gearbeitet wird so werden nach der Triggerung des Signals mit der entsprechenden Flanke (steigend oder fallend) am Input Capture Pin ICP jeweils 4 Messungen mit der CPU-Frequenz des Eingangssignals abgefragt. Nur dann, wenn alle 4 Messungen den gleichen Zustand aufweisen gilt das Signal als erkannt.</p>																																											
ICES1	<p>Input Capture Edge Select Timer/Counter 1 Mit diesem Bit wird bestimmt, ob die steigende (ICES1=1) oder fallende (ICES1=0) Flanke zur Auswertung des Input Capture Signals an Pin ICP heran gezogen wird.</p>																																											
CTC1	<p>Clear Timer/Counter on Compare Match Timer/Counter 1 Wenn dieses Bit gesetzt ist so wird nach einer Übereinstimmung des Datenregisters TCNT1H/TCNT1L mit dem Vergleichswert in OCR1H/OCR1L das Datenregister TCNT1H/TCNT1L auf 0 gesetzt. Da die Übereinstimmung im Takt nach dem Vergleich behandelt wird ergibt sich je nach eingestelltem Vorzähler ein etwas anderes Zählverhalten: Wenn der Vorteiler auf 1 gestellt ist und C der jeweilige Zählerwert ist, dann nimmt das Datenregister, im CPU-Takt betrachtet, folgende Werte an: ... C-2 C-1 C 0 1 ... Wenn der Vorteiler z.B. auf 8 eingestellt ist, dann nimmt das Datenregister folgende Werte an: ... C-2, C-2, C-2, C-2, C-2, C-2, C-2, C-2 C-1, C-1, C-1, C-1, C-1, C-1, C-1, C-1 C, 0, 0, 0, 0, 0, 0, 0 ... In der PWM-Betriebsart hat dieses Bit keine Funktion.</p>																																											
CS12	Clock Select Bits																																											
CS11	Diese 3 Bits bestimmen die Quelle für den Timer/Counter:																																											
CS10	<table border="1"> <thead> <tr> <th>CS12</th> <th>CS11</th> <th>CS10</th> <th>Resultat</th> </tr> </thead> <tbody> <tr> <td>0</td> <td>0</td> <td>0</td> <td>Stopp, Der Timer/Counter wird angehalten.</td> </tr> <tr> <td>0</td> <td>0</td> <td>1</td> <td>CPU-Takt</td> </tr> <tr> <td>0</td> <td>1</td> <td>0</td> <td>CPU-Takt / 8</td> </tr> <tr> <td>0</td> <td>1</td> <td>1</td> <td>CPU-Takt / 64</td> </tr> <tr> <td>1</td> <td>0</td> <td>0</td> <td>CPU-Takt / 256</td> </tr> <tr> <td>1</td> <td>0</td> <td>1</td> <td>CPU-Takt / 1024</td> </tr> <tr> <td>1</td> <td>1</td> <td>0</td> <td>Externer Pin T0, fallende Flanke</td> </tr> <tr> <td>1</td> <td>1</td> <td>1</td> <td>Externer Pin T0, steigende Flanke</td> </tr> </tbody> </table>								CS12	CS11	CS10	Resultat	0	0	0	Stopp, Der Timer/Counter wird angehalten.	0	0	1	CPU-Takt	0	1	0	CPU-Takt / 8	0	1	1	CPU-Takt / 64	1	0	0	CPU-Takt / 256	1	0	1	CPU-Takt / 1024	1	1	0	Externer Pin T0, fallende Flanke	1	1	1	Externer Pin T0, steigende Flanke
CS12	CS11	CS10	Resultat																																									
0	0	0	Stopp, Der Timer/Counter wird angehalten.																																									
0	0	1	CPU-Takt																																									
0	1	0	CPU-Takt / 8																																									
0	1	1	CPU-Takt / 64																																									
1	0	0	CPU-Takt / 256																																									
1	0	1	CPU-Takt / 1024																																									
1	1	0	Externer Pin T0, fallende Flanke																																									
1	1	1	Externer Pin T0, steigende Flanke																																									
	<p>Wenn als Quelle der externe Pin T0 verwendet wird, so wird ein Flankenwechsel auch erkannt, wenn der Pin T0 als Ausgang geschaltet ist.</p>																																											
TCNT1H TCNT1L	Timer/Counter Daten Register Timer/Counter 1																																											
	Dieses ist als 16-Bit Aufwärtszähler mit Schreib- und Lesezugriff realisiert. Wenn der Zähler den Wert 65535 erreicht hat beginnt er beim nächsten Zyklus wieder bei 0.																																											
Bit	7	6	5	4	3	2	1	0																																				
Name	MSB							TCNT1H																																				
Name							LSB	TCNT1L																																				

Name								LSB	TCNT1L
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initialwert	0	0	0	0	0	0	0	0	

In der PWM-Betriebsart wird das Register als Auf/Ab-Zähler verwendet, d.h. der Wert steigt zuerst von 0 bis er den Überlauf von 65535 auf 0 erreicht hat. Dann zählt das Register rückwärts wiederum bis 0.

Zum Auslesen des Registers wird von der CPU ein internes TEMP-Register verwendet. Das gleiche Register wird auch verwendet, wenn auf **OCR1** oder **ICR1** zugegriffen wird.

Deshalb müssen vor dem Zugriff auf eines dieser Register alle Interrupts gesperrt werden, weil sonst die Möglichkeit des gleichzeitigen Zugriffs auf das Temporärregister gegeben ist, was natürlich zu fehlerhaftem Verhalten des Programms führt.. Zudem muss zuerst **TCNT1L** und erst danach **TCNT1H** ausgelesen werden.

Wenn in das Register geschrieben werden soll müssen ebenfalls alle Interrupts gesperrt werden. Dann muss zuerst das **TCNT1H**-Register und erst danach das **TCNT1L**-Register geschrieben werden, also genau die umgekehrte Reihenfolge wie beim Lesen des Registers.

OCR1H
OCR1L

Timer/Counter Output Compare Register Timer/Counter 1

Bit	7	6	5	4	3	2	1	0	
Name	MSB								OCR1H
Name								LSB	OCR1L
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initialwert	0	0	0	0	0	0	0	0	

Der Wert im Output Compare Register wird ständig mit dem aktuellen Wert im Datenregister TCNT1H/TCNT1L verglichen. Stimmen die beiden Werte überein so wird ein sogenannter Output Compare Match ausgelöst. Die entsprechenden Aktionen werden über die Timer/Counter 1 Control und Status Register eingestellt..

Zum Auslesen des Registers wird von der CPU ein internes TEMP-Register verwendet. Das gleiche Register wird auch verwendet, wenn auf **OCR1** oder **ICR1** zugegriffen wird.

Deshalb müssen vor dem Zugriff auf eines dieser Register alle Interrupts gesperrt werden, weil sonst die Möglichkeit des gleichzeitigen Zugriffs auf das Temporärregister gegeben ist, was natürlich zu fehlerhaftem Verhalten des Programms führt.. Zudem muss zuerst **TCNT1L** und erst danach **TCNT1H** ausgelesen werden.

Wenn in das Register geschrieben werden soll müssen ebenfalls alle Interrupts gesperrt werden. Dann muss zuerst das **TCNT1H**-Register und erst danach das **TCNT1L**-Register geschrieben werden, also genau die umgekehrte Reihenfolge wie beim Lesen des Registers.

ICR1H
ICR1L

Timer/Counter Input Capture Register Timer/Counter 1

Bit	7	6	5	4	3	2	1	0	
Name	MSB								ICR1H
Name								LSB	ICR1L
R/W	R	R	R	R	R	R	R	R	
Initialwert	0	0	0	0	0	0	0	0	

Das Input Capture Register ist ein 16-Bit Register mit Lesezugriff. Es kann nicht

beschrieben werden.

Wenn am Input Capture Pin **ICP** die gemäss Einstellungen im **TCCR1B** definierte Flanke erkannt wird so wird der aktuelle Inhalt des Datenregisters **TCNT1H/TCNT1L** sofort in dieses Register kopiert und das Input Capture Flag **ICF1** im Timer Interrupt Flag Register **TIFR** gesetzt.

Wie bereits oben erwähnt müssen vor dem Zugriff auf dieses Register alle Interrupts gesperrt werden. Zudem müssen Low- und Highbyte des Registers in der richtigen Reihenfolge bearbeitet werden:

Lesen: **ICR1L -> ICR1H**

Schreiben: **ICR1H -> ICR1L**

15.3.1 DIE PWM-BETRIEBSART

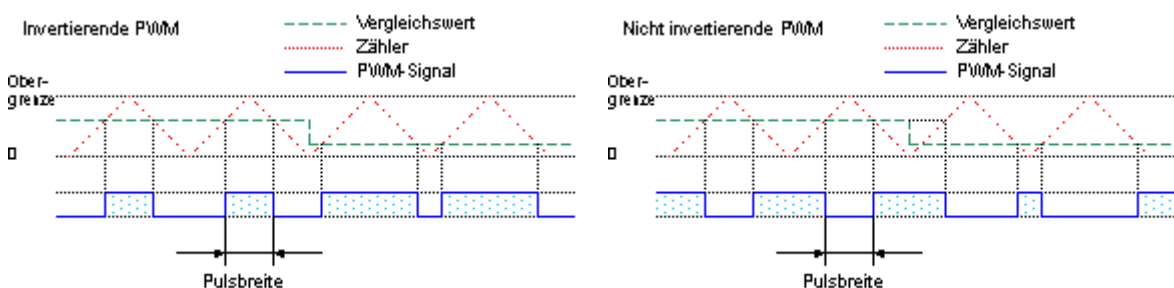
Wenn der Timer/Counter 1 in der PWM-Betriebsart betrieben wird so bilden das Datenregister **TCNT1H/TCNT1L** und das Vergleichsregister **OCR1H/OCR1L** einen 8-, 9- oder 10-Bit, frei laufenden PWM-Modulator, welcher als PWM-Signal am **OC1**-Pin (**PB3** beim 2313) abgegriffen werden kann. Das Datenregister **TCNT1H/TCNT1L** wird dabei als Auf-/Ab-Zähler betrieben, welcher von 0 an aufwärts zählt bis zur Obergrenze und danach wieder zurück auf 0.

Die Obergrenze ergibt sich daraus, ob 8- 9- oder 10-Bit PWM verwendet wird und zwar gemäss folgender Tabelle:

Auflösung	Obergrenze	Frequenz
8	255	$f_{TC1} / 510$
9	511	$f_{TC1} / 1022$
10	1023	$f_{TC1} / 2046$

Wenn nun der Zählerwert im Datenregister den in **OCR1H/OCR1L** gespeicherten Wert erreicht wird der Ausgangsbein **OC1** gesetzt bzw. gelöscht, je nach Einstellung von **COM1A1** und **COM1A0** im **TCCR1A**-Register.

Ich habe versucht, die entsprechenden Signale in der folgenden Grafik zusammenzufassen



15.3.2 VERGLEICHSWERT-ÜBERPRÜFUNG

Hier wird in ein spezielles Vergleichswertregister (**OCR1H/OCR1L**) ein Wert eingeschrieben, welcher ständig mit dem aktuellen Zählerwert verglichen wird.

Erreicht der Zähler den in diesem Register eingetragenen Wert so kann ein Signal (0 oder 1) am Pin **OC1** erzeugt und/oder ein Interrupt ausgelöst werden.

15.3.3 EINFANGEN EINES EINGANGSSIGNALS (INPUT CAPTURING)

Bei dieser Betriebsart wird an den Input Capturing Pin (ICP) des Controllers eine Signalquelle angeschlossen.

Nun kann je nach Konfiguration entweder ein Signalwechsel von 0 nach 1 (steigende Flanke) oder von 1 nach 0 (fallende Flanke) erkannt werden und der zu diesem Zeitpunkt aktuelle Zählerstand in ein spezielles Register abgelegt werden. Gleichzeitig kann auch ein entsprechender Interrupt ausgelöst werden.

Wenn die Signalquelle ein starkes Rauschen beinhaltet kann die Rauschunterdrückung eingeschaltet werden. Dann wird beim Erkennen der konfigurierten Flanke über 4 Taktzyklen das Signal überwacht und nur dann, wenn alle 4 Messungen gleich sind wird die entsprechende Aktion ausgelöst.

15.4 Gemeinsame Register

Verschiedene Register beinhalten Zustände und Einstellungen, welche sowohl für den 8-Bit, als auch für den 16-Bit Timer/Counter in ein und demselben Register zu finden sind.

TIMSK	Timer/Counter Interrupt Mask Register								
	Bit	7	6	5	4	3	2	1	0
	Name	TOIE1	OCIE1A	-	-	TICIE	-	TOIE0	-
	R/W	R/W	R/W	R	R	R/W	R	R/W	R
	Initialwert	0	0	0	0	0	0	0	0
	TOIE1	Timer/Counter Overflow Interrupt Enable Timer/Counter 1 Wenn dieses Bit gesetzt ist wird bei einem Überlauf des Datenregisters des Timer/Counter 1 ein Timer Overflow 1 Interrupt ausgelöst. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein.							
	OCIE1A	Output Compare Match Interrupt Enable Timer/Counter 1 Beim Timer/Counter 1 kann zusätzlich zum Überlauf ein Vergleichswert definiert werden. Wenn dieses Bit gesetzt ist wird beim Erreichen des Vergleichswertes ein Compare Match Interrupt ausgelöst. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein.							
	TICIE	Timer/Counter Input Capture Interrupt Enable Wenn dieses Bit gesetzt ist wird ein Capture Event Interrupt ausgelöst, wenn ein entsprechendes Signalereignis am Pin PD6(ICP) auftritt. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein, wenn auch ein entsprechender Interrupt ausgelöst werden soll.							
	TOIE0	Timer/Counter Overflow Interrupt Enable Timer/Counter 0 Wenn dieses Bit gesetzt ist wird bei einem Überlauf des Datenregisters des Timer/Counter 0 ein Timer Overflow 0 Interrupt ausgelöst. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein.							
TIFR	Timer/Counter Interrupt Flag Register								
	Bit	7	6	5	4	3	2	1	0
	Name	TOV1	OCF1A	-	-	ICF1	-	TOV0	-
	R/W	R/W	R/W	R	R	R/W	R	R/W	R
	Initialwert	0	0	0	0	0	0	0	0
	TOV1	Timer/Counter Overflow Flag Timer/Counter 1 Dieses Bit wird vom Controller gesetzt, wenn beim Timer 1 ein Überlauf des Datenregisters stattfindet.							

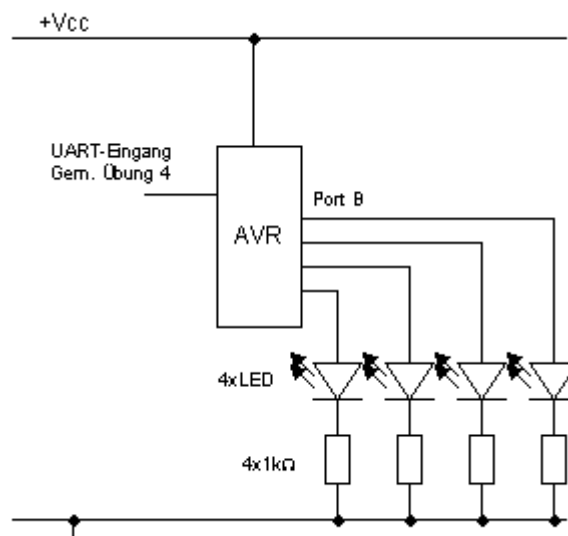
	<p>In der PWM-Betriebsart wird das Bit gesetzt, wenn die Zählrichtung von auf- zu abwärts und umgekehrt geändert wird (Zählerwert = 0). Das Flag wird automatisch gelöscht, wenn der zugehörige Interrupt-Vektor aufgerufen wird. Es kann jedoch auch gelöscht werden, indem eine logische 1 (!) in das entsprechende Bit geschrieben wird.</p>
OCF1A	<p>Output Compare Flag Timer/Counter 1 Dieses Bit wird gesetzt, wenn der aktuelle Wert des Datenregisters von Timer/Counter 1 mit demjenigen im Vergleichsregister OCR1 übereinstimmt. Das Flag wird automatisch gelöscht, wenn der zugehörige Interrupt-Vektor aufgerufen wird. Es kann jedoch auch gelöscht werden, indem eine logische 1 (!) in das entsprechende Bit geschrieben wird.</p>
ICF1	<p>Input Capture Flag Timer/Counter 1 Dieses Bit wird gesetzt, wenn ein Capture-Ereignis aufgetreten ist, welches anzeigt, dass der Wert des Datenregisters des Timer/Counter 1 in das Input Capture Register ICR1 übertragen wurde. Das Flag wird automatisch gelöscht, wenn der zugehörige Interrupt-Vektor aufgerufen wird. Es kann jedoch auch gelöscht werden, indem eine logische 1 (!) in das entsprechende Bit geschrieben wird.</p>
TOV0	<p>Timer/Counter Overflow Flag Timer/Counter 0 Dieses Bit wird vom Controller gesetzt, wenn beim Timer 0 ein Überlauf des Datenregisters stattfindet. Das Flag wird automatisch gelöscht, wenn der zugehörige Interrupt-Vektor aufgerufen wird. Es kann jedoch auch gelöscht werden, indem eine logische 1 (!) in das entsprechende Bit geschrieben wird.</p>

16 ÜBUNG 7, MEHRKANAL-SOFTWARE-PWM

Wie wir bereits wissen können wir mit dem 16-Bit Timer direkt ein PWM-Signal erzeugen. Allerdings eben nur eines.

Wir wollen nun versuchen, unter Zuhilfenahme des 8-Bit-Timers eine Softwaregesteuerte PWM mit mehreren Kanälen zu realisieren. Wir lassen es mal bei 4 Kanälen bewenden obwohl natürlich auch mehr möglich wären.

Als Anzeige dienen uns Leuchtdioden, welche wir wie gewohnt an Pin 0 bis Pin 3 von Port B anschliessen wollen, für jeden PWM-Kanal eine separate LED. Je nach Pulsweite des Signals wird nun die entsprechende LED mehr oder weniger hell leuchten.



Die Steuerung der Kanäle erfolgt über die serielle Schnittstelle direkt vom PC aus. Dazu muss auf dem PC jeweils eine Befehlszeile eingegeben und mit der Enter-Taste abgeschlossen werden. Die Befehlszeile ist wie folgt aufgebaut:

<Kanalnummer (1..4)>:<PWM-Wert (0...255)<Enter>

Beispiel:

1:50

2:120

3:40

4:200

Tipp:

Die vom PC empfangenen Zeichen müssen im AVR RAM gepuffert werden bis ein CR (Hex 0x0d) empfangen wird. Das Terminalprogramm auf dem PC muss so eingestellt sein, dass das CR alleine übertragen wird und kein LF angefügt wird.

Nach dem Empfang des CR wird der Puffer ausgewertet und der gewünschte Kanal angesteuert.

Denk bitte daran, die Kanalnummer im Puffer zu prüfen, oder anders gesagt, es darf nichts passieren, wenn der Benutzer versucht, den Kanal 65 anzusprechen.

Es wäre durchaus auch denkbar, bei fehlerhaften Befehlszeilen eine LED anzusteuern um den Benutzer darauf aufmerksam zu machen, dass er einen Quatsch eingegeben hat.

Die auszugebenden PWM-Werte werden nun für jeden Kanal gespeichert.

In der Hauptschleife des Programms wird jetzt einfach für jeden Kanal geprüft, ob der aktuelle Zählerwert grösser ist als der PWM-Wert für den entsprechenden Kanal. Falls ja wird das zugeordnete Bit des Port B gesetzt.

Versuch doch einfach einmal, das Programm selbst zu entwickeln.

Wenn es überhaupt nicht gelingen sollte dann geht's hier zur [Musterlösung](#) mit zugehörigem [Makefile](#).

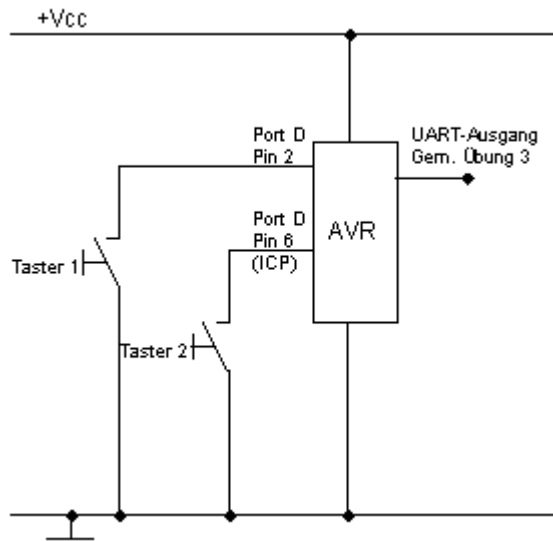
17 ÜBUNG 8, 16-BIT-TIMER/COUNTER

Wir wollen eine Zeitmessung zwischen 2 Ereignissen realisieren so wie sie z.B. bei Geschwindigkeitsmessungen verwendet wird.

Dazu schliessen wir am ICP ein Eingangssignal an. In der realen Welt könnte dies z.B. eine Lichtschranke sein. Wir nehmen hier der Einfachheit halber einen Taster.

Zusätzlich benötigen wir ein Triggersignal, welches die Zeitmessung startet. Auch hier könnte eine Lichtschranke verwendet werden. Wir schliessen hier aber einfach einen weiteren Taster an.

Wenn nun also das Triggersignal erkannt wird soll der Timer/Counter1 gestartet und auf 0 gesetzt werden. Wenn dann zu einem späteren Zeitpunkt die Flanke am ICP erkannt wird lesen wir den Wert des ICR1-Registers aus und berechnen anhand des Abstands der beiden virtuellen Lichtschranken die Geschwindigkeit. Das Resultat soll über den UART auf den PC übertragen und dort angezeigt werden (Terminal-Programm).



Tipps:

Für die Berechnung der Geschwindigkeit sollte keine Fließkommaarithmetik verwendet werden da sonst der Speicher des 2313 im Null Komma Nichts voll ist wenn die entsprechende Bibliothek vom GCC dazu gelinkt wird.

Dies bedeutet, dass wir die einmal erworbenen Mathematik- und Algebrakenntnisse wieder hervorkramen müssen um die entsprechenden Formeln zu erhalten.

Verwendete Formelzeichen:

s	Wegstrecke bzw. Abstand zwischen den beiden Signalgebern. Ich habe mit einem Abstand von 20 cm gerechnet, was 1/5 m entspricht.
t	Zeit, welche benötigt wird um die Strecke s zu durchfahren, in Sekunden. Bei einer CPU-Taktrate von 10 MHz und einem Vorteiler von 8 ergibt sich eine Taktzeit am Timerausgang von 0.0000008 bzw. 1/1250000 Sekunden.
v	Geschwindigkeit in m/s. Wenn wir diese noch mal 3600 und dann geteilt durch 1000 rechnen erhalten wir die Geschwindigkeit in km/h.
x	Anzahl Zählimpulse während der Durchfahrt. Diese müssen dann mit t multipliziert werden. Wenn wir auch langsamere Geschwindigkeiten erfassen wollen müssen wir den Überlauf des Timers ebenfalls auswerten und die Anzahl Durchläufe mit einberechnen.

Die Grundformel für die Geschwindigkeit lautet: $v=s/t$

Mit $s=1/5$ und $t=x * 1/1250000$: $v=(1/5) / (x*1/1250000)$

Wenn wir nun noch wissen, dass man Brüche dividiert, indem man mit dem Kehrwert des zweiten Bruchs multipliziert, dann wird daraus: $v=(1/5) * (1250000/x)$

Das ganze dann noch ein wenig gekürzt: $v=250000/x$

Und jetzt noch in km/h umrechnen: $v=(250000*3600)/(x*1000)$

Und nochmal kürzen: $v=(250*3600)/x$

ergibt schlussendlich: $v=900000/x$

Da wir hier mit grossen Zahlen rechnen müssen entsprechend **long**-Variablen verwendet werden. Desweiteren sollte immer zuerst multipliziert und erst dann dividiert werden, da sonst durch die Integerarithmetik zu viel weggeschnitten wird.

Nehmen wir dazu mal folgendes einfaches Beispiel:

$$x = 15 / 4 * 8$$

Wenn wir zuerst die Division durchführen erhalten wir $15 / 4 = 3.75$. Dies wird aber auf 3 geschnitten. Diese 3 multipliziert mit 8 ergibt $3*8=24$.

Führen wir aber zuerst die Multiplikation aus so ergibt sich $15 * 8 = 120$ und diese $120 / 4 = 30$.

Versuch doch einfach einmal, das Programm selbst zu entwickeln.

Wenn es überhaupt nicht gelingen sollte dann geht's hier zur [Musterlösung](#) mit zugehörigem [Makefile](#).

Zur Ausgabe der Resultate kann die bereits früher verwendete Funktion [UartPrintF](#) verwendet werden. Hier ist die zugehörige [Headerdatei](#).

Ich habe versuchsweise einmal anstelle der Taster 2 Reed-Kontakte angeschlossen und bin mit einem Dauermagneten darüber gefahren. Auf diese Art und Weise könnte beispielsweise eine Geschwindigkeitsmessung bei einer Modelleisenbahn realisiert werden. Die Reedkontakte werden einfach in die Schienen montiert und unter der Lok oder einem Wagen wird ein Magnet befestigt. Die Anzeige sollte dann allerdings nicht unbedingt in km/h erfolgen oder wenn, dann im entsprechenden Massstab reduziert.

18 DER WATCHDOG

Und hier kommt das ultimative Mittel gegen die Unvollkommenheit von uns Programmierern, der Watchdog.

So sehr wir uns auch anstrengen, es wird uns kaum je gelingen, das absolut perfekte und fehlerfreie Programm zu entwickeln.

Der Watchdog kann uns zwar auch nicht zu besseren Programmen verhelfen aber er kann dafür sorgen, dass unser Programm, wenn es sich wieder mal in's Nirwana verabschiedet hat, neu gestartet wird, indem ein Reset des Controllers ausgelöst wird.

Betrachten wir doch einmal folgende Codesequenz:

```
unsigned char x;

x = 10;

while (x >= 0) {
    // tu was
    x--;
}
```

Wenn wir die Schleife mal genau anschauen sollte uns auffallen, dass dieselbe niemals beendet wird. Warum nicht? Ganz einfach, weil eine als *unsigned* deklarierte Variable niemals kleiner als Null werden kann.

Das Programm würde sich also hier aufhängen und auf ewig in der Schleife drehen.

Und hier genau kommt der Watchdog zum Zug.

18.1 Wie funktioniert nun der Watchdog

Der Watchdog enthält einen separaten Timer/Counter, welcher mit einem intern erzeugten Takt von 1 MHz bei 5V Vcc getaktet wird. Nachdem der Watchdog aktiviert und der gewünschte Vorteiler eingestellt wurde beginnt der Counter von 0 an hochzuzählen. Wenn nun die je nach Vorteiler eingestellte Anzahl Zyklen erreicht wurde löst der Watchdog einen Reset aus. Um nun also im Normalbetrieb den Reset zu verhindern müssen wir den Watchdog regelmässig wieder neu starten bzw. Rücksetzen (Watchdog Reset). Dies sollte innerehalb unserer Hauptschleife passieren.

Um ein unbeabsichtigtes Ausschalten des Watchdogs zu verhindern muss ein spezielles Prozedere verwendet werden um den WD auszuschalten und zwar müssen zuerst die beiden Bits WDTOE und WDE in einer einzelnen Operation (also nicht mit sbi) auf 1 gesetzt werden. Dann muss innerhalb der nächsten 4 Taktzyklen das Bit WDE auf 0 gesetzt werden.

Das Watchdog Control Register:

WDTCR	Watchdog Timer Control Register In diesem Register stellen wir ein, wie wir den Watchdog verwenden möchten. Das Register ist wie folgt aufgebaut:																																				
	<table border="1"><thead><tr><th>Bit</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr></thead><tbody><tr><td>Name</td><td>-</td><td>-</td><td>-</td><td>WDTOE</td><td>WDE</td><td>WDP2</td><td>WDP1</td><td>WDP0</td></tr><tr><td>R/W</td><td>R</td><td>R</td><td>R</td><td>R/W</td><td>R/W</td><td>R/W</td><td>R/W</td><td>R/W</td></tr><tr><td>Initialwert</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></tbody></table>	Bit	7	6	5	4	3	2	1	0	Name	-	-	-	WDTOE	WDE	WDP2	WDP1	WDP0	R/W	R	R	R	R/W	R/W	R/W	R/W	R/W	Initialwert	0	0	0	0	0	0	0	0
	Bit	7	6	5	4	3	2	1	0																												
	Name	-	-	-	WDTOE	WDE	WDP2	WDP1	WDP0																												
	R/W	R	R	R	R/W	R/W	R/W	R/W	R/W																												
Initialwert	0	0	0	0	0	0	0	0																													
WDTOE Watchdog Turn Off Enable Dieses Bit muss gesetzt sein, wenn das Bit WDE gelöscht wird, andernfalls wird der Watchdog nicht ausgeschaltet.																																					

			Wenn das Bit einmal gesetzt ist wird es von der Hardware nach 4 Taktzyklen automatisch wieder gelöscht.			
WDE	Watchdog Enable		Wenn dieses Bit gesetzt wird so wird der Watchdog aktiviert. Das Bit kann nur gelöscht werden solange das Bit WDTOE auf 1 steht.			
WDP2	Watchdog Timer Prescaler Bits		Diese 3 Bits bestimmen die Anzahl Oszillatorzyklen für den Watchdog, also, wie lange es dauert, bis ein Reset ausgelöst wird:			
WDP1						
WDP0						
	WDP2	WDP1	WDP0	Anzahl Zyklen	Typ. Timeoutzeit bei Vcc = 3V	Typ. Timeoutzeit bei Vcc = 5V
	0	0	0	16K	47ms	15ms
	0	0	1	32K	94ms	30ms
	0	1	0	64K	0.19s	60ms
	0	1	1	128K	0.38s	0.12s
	1	0	0	256K	0.75s	0.24s
	1	0	1	512K	1.5s	0.49s
	1	1	0	1024K	3s	0.97s
	1	1	1	2048K	6s	1.9s

Um den Watchdog mit dem AVR-GCC Compiler zu verwenden muss die Headerdatei wdt.h in die Quelldatei eingebunden werden. Dadurch wird auch der Startup-Code entsprechend angepasst, so dass der Watchdog nach einem Reset automatisch gestartet wird. Das WDTCR-Register wird dabei mit dem Wert 0 beschrieben. Falls ein anderer Wert gewünscht ist so kann dies im Makfile in den Linker-Optionen eingetragen werden. Dazu muss in der Zeile LDFLAGS folgende Option angefügt werden:

```
--defsym __init_wdcr__=0x1f
```

wenn beispielsweise der Wert des Registers auf 0x1f gestellt werden soll.

Danach können die folgenden Funktionen verwendet werden:

```
wdt_disable();
```

Mit dieser Funktion kann der Watchdog ausgeschaltet werden. Dabei wird das notwendige Prozedere, wie oben beschrieben, automatisch ausgeführt.

```
wdt_enable(uint8_t timeout);
```

Aktiviert den Watchdog und stellt den Vorteiler auf den gewünschten Wert ein bzw. der in timeout übergebene Wert wird in das WDTCR-Register eingetragen.

```
wdt_reset();
```

Dies ist wohl die wichtigste der Watchdog-Funktionen. Sie erzeugt einen Watchdog-Reset, welcher periodisch, und zwar vor Ablauf der Timeoutzeit, ausgeführt werden muss, damit der Watchdog nicht etwa unbeabsichtigt den AVR zurücksetzt.

Selbstverständlich kann das **WDTCR**-Register auch mit den uns bereits bekannten Funktionen für den Zugriff auf Register programmiert werden.

18.2 Ein paar grundsätzliche Gedanken

Ob nun so ein Watchdog überhaupt verwendet werden soll ist wohl eher eine philosophische Frage und hängt vom Geschmack jedes einzelnen Entwicklers ab.

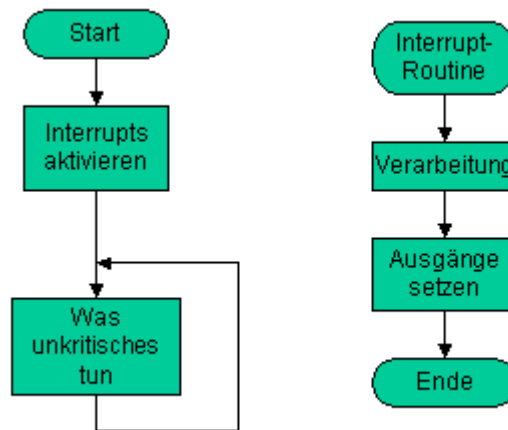
Ich persönlich habe es lieber, wenn ich auch merke, dass in meine Programm etwas noch nicht in Ordnung ist, als dass mir ein Watchdog einfach die CPU immer wieder zurücksetzt, denn immerhin

kann es so passieren, dass ein Programm über Jahre hinweg so einigermaßen läuft, obwohl noch ein voll krasser Bock drin liegt.

19 PROGRAMMIEREN MIT INTERRUPTS

Nachdem wir nun alles Wissenswerte für die serielle Programmerstellung gelernt haben nehmen wir jetzt ein völlig anderes Thema in Angriff, nämlich die Programmierung unter Zuhilfenahme der Interrupts des AVR.

Als erstes wollen wir uns noch einmal den allgemeinen Programmablauf bei der Interrupt-Programmierung zu Gemüte führen.



Man sieht, dass die Interruptroutine quasi parallel zum Hauptprogramm abläuft. Da wir nur eine CPU haben ist es natürlich keine echte Parallelität, sondern das Hauptprogramm wird beim Eintreffen eines Interrupts unterbrochen, die Interruptroutine wird ausgeführt und danach erst wieder zum Hauptprogramm zurückgekehrt.

19.1 Anforderungen an die Interrupt-Routine

Um unliebsamen Überraschungen vorzubeugen sollten einige Grundregeln bei der Gestaltung der Interruptroutinen beachtet werden.

- Die Interruptroutine soll möglichst kurz und schnell abarbeitbar sein, daraus folgt:
- Keine umfangreichen Berechnungen innerhalb der Interruptroutine.
- Keine endlos langen Programmschleifen.
- Obschon es möglich ist, während der Abarbeitung einer Interruptroutine andere oder sogar den gleichen Interrupt wieder zuzulassen rate ich dringend von solchen Spielen ab.

19.2 Interrupt-Quellen

Die folgenden Ereignisse können einen Interrupt auf dem AVR auslösen, wobei die Reihenfolge der Auflistung auch die Priorität der Interrupts aufzeigt.

- Reset
- Externer Interrupt 0
- Externer Interrupt 1
- Timer/Counter 1 Capture Ereignis
- Timer/Counter 1 Compare Match

- Timer/Counter 1 Überlauf
- Timer/Counter 0 Überlauf
- UART Zeichen empfangen
- UART Datenregister leer
- UART Zeichen gesendet
- Analoger Komparator

19.3 Register

Der AT90S2313 verfügt über 2 Register welche mit den Interrupts zusammen hängen.

GIMSK	General Interrupt Mask Register.								
	Bit	7	6	5	4	3	2	1	0
	Name	INT1	INT0	-	-	-	-	-	-
	R/W	R/W	R/W	R	R	R	R	R	R
	Initialwert	0	0	0	0	0	0	0	0
	INT1	<p>External Interrupt Request 1 Enable Wenn dieses Bit gesetzt ist wird ein Interrupt ausgelöst wenn am INT1-Pin eine steigende oder fallende (je nach Konfiguration im MCUCR) Flanke erkannt wird. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein. Der Interrupt wird auch ausgelöst, wenn der Pin als Ausgang geschaltet ist. Auf diese Weise bietet sich die Möglichkeit, Software-Interrupts zu realisieren.</p>							
	INT0	<p>External Interrupt Request 0 Enable Wenn dieses Bit gesetzt ist wird ein Interrupt ausgelöst wenn am INT0-Pin eine steigende oder fallende (je nach Konfiguration im MCUCR) Flanke erkannt wird. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein. Der Interrupt wird auch ausgelöst, wenn der Pin als Ausgang geschaltet ist. Auf diese Weise bietet sich die Möglichkeit, Software-Interrupts zu realisieren.</p>							

GIFR	General Interrupt Flag Register.								
	Bit	7	6	5	4	3	2	1	0
	Name	INTF1	INTF0	-	-	-	-	-	-
	R/W	R/W	R/W	R	R	R	R	R	R
	Initialwert	0	0	0	0	0	0	0	0
	INTF1	<p>External Interrupt Flag 1 Dieses Bit wird gesetzt, wenn am INT1-Pin eine Interrupt-Kondition, entsprechend der Konfiguration, erkannt wird. Wenn das Global Enable</p>							
	INTF0	<p>External Interrupt Flag 0 Dieses Bit wird gesetzt, wenn am INT0-Pin eine Interrupt-Kondition,</p>							

Das Flag wird automatisch gelöscht, wenn die Interruptroutine beendet ist. Alternativ kann das Flag gelöscht werden, indem der Wert **1(!)** eingeschrieben wird.

MCUCCR	MCU Control Register.								
	Das MCU Control Register enthält Kontrollbits für allgemeine MCU-Funktionen.								
	Bit	7	6	5	4	3	2	1	0
	Name	-	-	SE	SM	ISC11	ISC10	ISC01	ISC00
	R/W	R	R	R/W	R/W	R/W	R/W	R/W	R
Initialwert	0	0	0	0	0	0	0	0	
	SE	Sleep Enable Dieses Bit muss gesetzt sein, um den Controller mit dem SLEEP -Befehl in den Schlafzustand versetzen zu können. Um den Schlafmodus nicht irrtümlich einzuschalten wird empfohlen, das Bit erst unmittelbar vor Ausführung des SLEEP -Befehls zu setzen.							
	SM	Sleep Mode Dieses Bit bestimmt der Schlafmodus. Ist das Bit gelöscht so wird der Idle -Modus ausgeführt. Ist das Bit gesetzt so wird der Power-Down -Modus ausgeführt.							
	ISC11 ISC10	Interrupt Sense Control 1 Bits Diese beiden Bits bestimmen, ob die steigende oder die fallende Flanke für die Interrupterkennung am INT1 -Pin ausgewertet wird.							
		ISC11	ISC10	Bedeutung					
		0	0	Low Level an INT1 erzeugt einen Interrupt. In der Beschreibung heisst es, der Interrupt wird getriggert, solange der Pin auf 0 bleibt, also eigentlich unbrauchbar.					
		0	1	Reserviert					
		1	0	Die fallende Flank an INT1 erzeugt einen Interrupt.					
		1	1	Die steigende Flanke an INT1 erzeugt einen Interrupt.					
	ISC01 ISC00	Interrupt Sense Control 0 Bits Diese beiden Bits bestimmen, ob die steigende oder die fallende Flanke für die Interrupterkennung am INT0 -Pin ausgewertet wird.							
		ISC01	ISC00	Bedeutung					
		0	0	Low Level an INT0 erzeugt einen Interrupt. In der Beschreibung heisst es, der Interrupt wird getriggert, solange der Pin auf 0 bleibt, also eigentlich unbrauchbar.					
		0	1	Reserviert					
		1	0	Die fallende Flank an INT0 erzeugt einen Interrupt.					
		1	1	Die steigende Flanke an INT0 erzeugt einen Interrupt.					

19.4 Allgemeines über die Interrupt-Abarbeitung

Wenn ein Interrupt eintrifft wird automatisch das **Global Interrupt Enable** Bit im Status Register **SREG** gelöscht und alle weiteren Interrupts unterbunden. Obwohl es möglich ist, zu diesem Zeitpunkt bereits wieder das I-bit zu setzen rate ich dringend davon ab. Dieses wird nämlich

automatisch gesetzt, wenn die Interruptroutine beendet wird. Wenn in der Zwischenzeit weitere Interrupts eintreffen werden die zugehörigen Interrupt-Bits gesetzt und die Interrupts bei Beendigung der laufenden Interrupt-Routine in der Reihenfolge ihrer Priorität ausgeführt. Dies kann eigentlich nur dann zu Problemen führen, wenn ein hoch priorisierter Interrupt ständig und in kurzer Folge auftritt. Dieser sperrt dann womöglich alle anderen Interrupts mit niedrigerer Priorität. Dies ist einer der Gründe, weshalb die Interrupt-Routinen sehr kurz gehalten werden sollen.

19.4.1 DAS STATUS-REGISTER

Es gilt auch zu beachten, dass das Status-Register während der Abarbeitung einer Interruptroutine nicht automatisch gesichert wird. Falls notwendig muss dies vom Programmierer selber vorgesehen werden.

19.5 Interrupts mit dem AVR GCC Compiler (WinAVR)

Selbstverständlich können alle Interruptspezifischen Registerzugriffe wie gewohnt über I/O-Adressierung vorgenommen werden. Etwas einfacher geht es jedoch, wenn wir die vom Compiler zur Verfügung gestellten Mittel einsetzen.

Damit diese Mittel zur Verfügung stehen müssen wir die Includedatei interrupt.h einbinden mittels

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

Und dann kann's losgehen

```
sei ();
```

Das Makro **sei()** schaltet die Interrupts ein. Eigentlich wird nichts anderes gemacht als das **Global Interrupt Enable** Bit im Status Register gesetzt.

```
cli ();
```

Das Makro **cli()** schaltet die Interrupts aus oder anders gesagt, das **Global Interrupt Enable** Bit im Status Register wird gelöscht.

```
timer_enable_int (unsigned char ints);
```

Schaltet Timerbezogene Interrupts ein bzw. aus.

Wenn als Argument **ints** der Wert 0 übergeben wird so werden alle Timerinterrupts ausgeschaltet, ansonsten muss in **ints** angegeben werden, welche Interrupts zu aktivieren sind. Dabei müssen einfach die entsprechend zu setzenden Bits definiert werden.

```
Beispiel: timer_enable_int (1 << TOIE1);
```

Achtung: Wenn ein Timerinterrupt eingeschaltet wird während ein anderer Timerinterrupt bereits läuft, dann müssen beide Bits angegeben werden sonst wird der andere Timerinterrupt versehentlich ausgeschaltet.

```
enable_external_int (unsigned char ints);
```

Schaltet die externen Interrupts ein bzw. aus.

Wenn als Argument **ints** der Wert 0 übergeben wird so werden alle externen Interrupts ausgeschaltet, ansonsten muss in **ints** angegeben werden, welche Interrupts zu aktivieren sind. Dabei müssen einfach die entsprechend zu setzenden Bits definiert werden.

```
Beispiel: enable_external_int ((1<<INT0) | (1<<INT1));
```

Schaltet die externen Interrupts 0 und 1 ein.

Nachdem nun die Interrupts aktiviert sind braucht es selbstverständlich noch den auszuführenden Code, der ablaufen soll wenn ein Interrupt eintrifft.

Dazu gibt es zwei Definitionen welche allerdings AVR-GCC spezifisch sind und bei anderen Compilern womöglich anders heissen können.

```
SIGNAL(signame);
```

Mit SIGNAL wird eine Funktion für die Bearbeitung eines Interrupts eingeleitet. Als Argument muss dabei die Benennung des entsprechenden Interruptvektoren angegeben werden. Diese sind in den jeweiligen Includedateien IOxxxx.h zu finden. Mögliche Funktionsrumpfe für solche Interruptfunktionen sind zum Beispiel:

```
SIGNAL (SIG_INTERRUPT0)
{
    // Hier kommt der Code hin
}

SIGNAL (SIG_OVERFLOW1)
{
    // Und hier kommt auch Code hin
}

SIGNAL (SIG_UART_RECV)
{
    // rate mal was hier hin kommt
}

// und so weiter und so fort...
```

Während der Ausführung des Funktion sind alle weiteren Interrupts automatisch gesperrt. Beim Verlassen der Funktion werden die Interrupts wieder zugelassen.

Sollte während der Abarbeitung der Interruptroutine ein weiterer Interrupt (gleiche oder andere Interruptquelle) auftreten so wird das entsprechende Bit im zugeordneten Interrupt Flag Register gesetzt und die entsprechende Interruptroutine automatisch nach dem Beenden der aktuellen Funktion aufgerufen.

Ein Problem ergibt sich eigentlich nur dann, wenn während der Abarbeitung der aktuellen Interruptroutine mehrere gleichartige Interrupts auftreten. Die entsprechende Interruptroutine wird im Nachhinein zwar aufgerufen jedoch wissen wir nicht, ob nun der entsprechende Interrupt einmal, zweimal oder gar noch öfter aufgetreten ist. Deshalb soll hier noch einmal betont werden, dass Interruptroutinen so schnell wie nur irgend möglich wieder verlassen werden sollten.

```
INTERRUPT (signame);
```

Mit INTERRUPT wird genau gleich gearbeitet wie mit SIGNAL. Der Unterschied ist derjenige, dass bei INTERRUPT beim Aufrufen der Funktion das **Global Enable Interrupt** Bit automatisch wieder gesetzt und somit weitere Interrupts zugelassen werden. Dies kann zu nicht unerheblichen Problemen von im einfachsten Fall einem Stack overflow bis zu sonstigen unerwarteten Effekten führen und sollte wirklich nur dann angewendet werden wenn man sich absolut sicher ist, das ganze auch im Griff zu haben.

19.6 Was tut das Hauptprogramm

In einfachsten Fall gar nichts mehr.

Es ist also durchaus denkbar, ein Programm zu schreiben, welches in der main-Funktion lediglich noch die Interrupts aktiviert und dann in eine Endlosschleife folgender Art verzweigt:

```
for (;;) {
```

Normalerweise wird man allerdings in den Interruptroutinen die Interrupts erfassen und im Hauptprogramm dann gemütlich auswerten.

Wie wir im bisherigen Kursverlauf gesehen haben ist es ohnehin mit so schnellen Controllern meistens gar nicht unbedingt notwendig mit Interruptfunktionen zu arbeiten.

Es ist allerdings auch zu bemerken, dass mit den Interruptroutinen ein Programm sehr schön strukturiert werden kann, wenn man es richtig macht.

20 ÜBUNG 9, INTERRUPTS

Zum Abschluss dieses Tutorials wollen wir eine interruptgesteuerte Uhr realisieren, welche die aktuelle Stunde und Minute in binärer Form auf Leuchtdioden anzeigt.

Die Leuchtdioden für die Anzeige der Minute werden an Port B, Pin 0...5 angeschlossen.

Die Leuchtdioden für die Anzeige der Stunden werden an Port D, Pin 2...6 angeschlossen.

An Pin 7 von Port B soll ein Taster angeschlossen werden. Wenn dieser Taster gedrückt wird kann die Zeit eingestellt werden. Dabei wird bei einem kurzen Tastendruck die Minute einmal hochgezählt. Wenn der Taster gedrückt bleibt werden die Minuten im Schnelldurchlauf hochgezählt (ca. 2 bis 3 Impulse pro Sekunden).

Einige Tipps

Die globalen Variablen (z.B. zur Speicherung der Uhrzeit) sollten alle mit dem Schlüsselwort `volatile` deklariert werden da der Compiler dieselben sonst möglicherweise wegoptimiert.

Ich habe einen Quarz mit einer Frequenz von 386400 Hz eingesetzt. Wenn nun der Timer 1 mit dem Vorzähler 1 gestartet wird so läuft dieser jede Minute genau 3375 mal durch. Wir brauchen also lediglich eine Interruptroutine in den Timer 1 Overflow einzubinden und bei jedem Aufruf einen Zähler zu inkrementieren. Erreicht dieser Zähler den Wert 3375 so ist eine Minute verstrichen.

Für die Tastaturabfrage kann der Zählerwert Modulo 25 genommen werden. Wenn diese Berechnung 0 ergibt kann der Tastenzustand überprüft werden. Damit haben wir automatisch eine Entprellung und auch die Dauerfunktion der Taste ist dann eigentlich schon dabei.

Ich persönlich würde allerdings innerhalb der Timeroutine lediglich ein globales Flag setzen und die eigentliche Auswertung der Taste in's Hauptprogramm verlegen (Ihr wisst schon: kurze Interruptroutine).

Genügend grosse Vorwiderstände für die LED's vorsehen, da wir ja keinen AVR verbraten wollen.

An den unbenutzten Pins können jetzt nach Lust und Laune etwelche Verbraucher zeitlich gesteuert werden und dies auf eine Minute genau.

Auch hier habe ich eine Musterlösung vorbereitet. Diesmal ist der [Quellcode](#) und das [Makefile](#) für WinAVR geschrieben. Es lässt sich aber mit dem bisherigen Wissen problemlos auf die ältere Compilerversion anpassen.

21 SCHLUSSWORT

Wir sind nun am Ende dieses Tutorials angelangt.

Ich hoffe, ich konnte mit meinen Ausführungen dem/der einen oder anderen zukünftigen Programmierer(in) etwas mit auf den Weg geben.

Solltet ihr Fragen oder Anregungen haben, welche direkt dieses Tutorial betreffen, so setzt euch direkt mit mir in Verbindungen.

Alle Fragen allgemeiner Art (Programmierung etc.) bitte ich jedoch im Forum (www.mikrocontroller.net) zu stellen.

Ich wünsche allen von euch viel Erfolg bei der Arbeit mit diesen leistungsfähigen Mikrocontrollern.

Christian Schifferle