

AVR-GCC-Tutorial

Inhaltsverzeichnis

[[Verbergen](#)]

- [1 Vorwort](#)
- [2 Benötigte Werkzeuge](#)
- [3 Was tun, wenn's nicht "klappt"?](#)
- [4 Erzeugen von Maschinencode](#)
- [5 Einführungsbeispiel](#)
- [6 Exkurs: Makefiles](#)
- [7 Ganzzahlige \(Integer\) Datentypen](#)
- [8 Bitfelder](#)
- [9 Grundsätzlicher Programmaufbau eines \$\mu\$ C-Programms](#)
 - [9.1 Sequentieller Programmablauf](#)
 - [9.2 Interruptgesteuerter Programmablauf](#)
- [10 Zugriff auf Register](#)
 - [10.1 Schreiben in Register](#)
 - [10.2 Verändern von Registerinhalten](#)
 - [10.3 Lesen aus Registern](#)
 - [10.4 Warten auf einen bestimmten Zustand](#)
 - [10.5 16-Bit Register \(ADC, ICR1, OCR1, TCNT1, UBRR\)](#)
 - [10.6 IO-Register als Parameter und Variablen](#)
- [11 Zugriff auf IO-Ports](#)
 - [11.1 Datenrichtung bestimmen](#)
 - [11.2 Vordefinierte Bitnummern für I/O-Register](#)
 - [11.3 Digitale Signale](#)
 - [11.4 Ausgänge](#)
 - [11.5 Eingänge \(Wie kommen Signale in den \$\mu\$ C\)](#)
 - [11.5.1 Signalkopplung](#)
 - [11.5.2 Interne Pull-Up Widerstände](#)
 - [11.5.3 Tasten und Schalter](#)
 - [11.5.3.1 \(Tasten-\)Entprellung](#)
- [12 Der UART](#)
 - [12.1 Allgemeines zum UART](#)
 - [12.2 Die Hardware](#)
 - [12.3 UART initialisieren](#)
 - [12.4 Senden mit dem UART](#)
 - [12.4.1 Senden einzelner Zeichen](#)
 - [12.4.2 Schreiben einer Zeichenkette \(String\)](#)
 - [12.4.3 Schreiben von Variableninhalten](#)
 - [12.5 Empfangen](#)
 - [12.5.1 einzelne Zeichen empfangen](#)
 - [12.5.2 Empfang von Zeichenketten \(Strings\)](#)
 - [12.6 Interruptbetrieb](#)
 - [12.7 Software-UART](#)
 - [12.8 Handshaking](#)
 - [12.8.1 Hardwarehandshake \(RTS/CTS\)](#)
 - [12.8.2 Softwarehandshake \(XON/XOFF\)](#)
 - [12.9 Fehlersuche](#)
 - [12.10 Links](#)
- [13 Analoge Ein- und Ausgabe](#)
 - [13.1 AC \(Analog Comparator\)](#)
 - [13.2 ADC \(Analog Digital Converter\)](#)
 - [13.2.1 Der interne ADC im AVR](#)
 - [13.2.1.1 Die Register des ADC](#)
 - [13.2.1.2 Aktivieren des ADC](#)
 - [13.2.2 Analog-Digital-Wandlung ohne internen ADC](#)
 - [13.2.2.1 Messen eines Widerstandes](#)
 - [13.2.2.2 ADC über Komparator](#)
 - [13.3 DAC \(Digital Analog Converter\)](#)
 - [13.3.1 DAC über mehrere digitale Ausgänge](#)
 - [13.3.2 PWM \(Pulsweitenmodulation\)](#)
- [14 LCD-Ansteuerung](#)
 - [14.1 Das LCD und sein Controller](#)
 - [14.2 Anschluss an den Controller](#)
 - [14.3 Programmierung](#)

- [15 Die Timer/Counter des AVR](#)
 - [15.1 Der Vorteiler \(Prescaler\)](#)
 - [15.2 8-Bit Timer/Counter](#)
 - [15.3 Timer-Bitzahlen verschiedener AVR](#)
 - [15.4 16-Bit Timer/Counter](#)
 - [15.4.1 Die PWM-Betriebsart](#)
 - [15.4.2 Vergleichswert-Überprüfung \(Compare Match\)](#)
 - [15.4.3 CTC-Betriebsart \(Clear Timer on Compare Match\)](#)
 - [15.4.4 Einfangen eines Eingangssignals \(Input Capturing\)](#)
 - [15.5 Gemeinsame Register](#)
- [16 Warteschleifen \(delay.h\)](#)
 - [16.1 avr-libc Versionen kleiner 1.6](#)
 - [16.2 avr-libc Versionen ab 1.6](#)
- [17 Der Watchdog](#)
 - [17.1 Wie funktioniert nun der Watchdog?](#)
 - [17.2 Watchdog-Anwendungshinweise](#)
- [18 Programmieren mit Interrupts](#)
 - [18.1 Anforderungen an Interrupt-Routinen](#)
 - [18.2 Interrupt-Quellen](#)
 - [18.3 Register](#)
 - [18.4 Allgemeines über die Interrupt-Abarbeitung](#)
 - [18.5 Interrupts mit dem AVR GCC Compiler \(WinAVR\)](#)
 - [18.5.1 ISR](#)
 - [18.5.2 Unterbrechbare Interruptroutinen](#)
 - [18.6 Datenaustausch mit Interrupt-Routinen](#)
 - [18.6.1 volatile und Pointer](#)
 - [18.6.2 Variablen größer 1 Byte](#)
 - [18.7 Interrupt-Routinen und Registerzugriffe](#)
 - [18.8 Was macht das Hauptprogramm?](#)
- [19 Sleep-Modes](#)
- [20 Speicherzugriffe](#)
 - [20.1 RAM](#)
 - [20.2 Programmspeicher \(Flash\)](#)
 - [20.2.1 Byte lesen](#)
 - [20.2.2 Wort lesen](#)
 - [20.2.3 Strings lesen](#)
 - [20.2.4 Floats und Structs lesen](#)
 - [20.2.5 Array aus Strings im Flash-Speicher](#)
 - [20.2.6 Vereinfachung für Zeichenketten \(Strings\) im Flash](#)
 - [20.2.7 Flash in der Anwendung schreiben](#)
 - [20.2.8 Warum so kompliziert?](#)
 - [20.3 EEPROM](#)
 - [20.3.1 Bytes lesen/schreiben](#)
 - [20.3.2 Wort lesen/schreiben](#)
 - [20.3.3 Block lesen/schreiben](#)
 - [20.3.4 EEPROM-Speicherabbild in .eep-Datei](#)
 - [20.3.5 EEPROM-Variable auf feste Adressen legen](#)
 - [20.3.6 Bekannte Probleme bei den EEPROM-Funktionen](#)
 - [20.3.7 EEPROM Register](#)
- [21 Die Nutzung von sprintf und printf](#)
- [22 Assembler und Inline-Assembler](#)
 - [22.1 Inline-Assembler](#)
 - [22.2 Assembler-Dateien](#)
 - [22.3 Globale Variablen für Datenaustausch](#)
 - [22.3.1 Globale Variablen im Assemblerfile anlegen](#)
 - [22.3.2 Variablen größer als 1 Byte](#)
- [23 Anhang](#)
 - [23.1 Besonderheiten bei der Anpassung bestehenden Quellcodes](#)
 - [23.1.1 Veraltete Funktionen zur Deklaration von Interrupt-Routinen](#)
 - [23.1.2 Veraltete Funktionen zum Portzugriff](#)
 - [23.1.3 Veraltete Funktionen zum Zugriff auf Bits in Registern](#)
 - [23.1.4 Selbstdefinierte \(nicht-standardisierte\) ganzzahlige Datentypen](#)
 - [23.2 Zusätzliche Funktionen im Makefile](#)
 - [23.2.1 Bibliotheken \(Libraries/.a-Dateien\) hinzufügen](#)
 - [23.2.2 Fuse-Bits](#)
 - [23.3 Externe Referenzspannung des internen Analog-Digital-Wandlers](#)
- [24 TODO](#)
- [25 Softwareentwicklung](#)
- [26 Programmierstil](#)

Vorwort

Dieses Tutorial soll den Einstieg in die Programmierung von Atmel [AVR](#)-Mikrocontrollern in der Programmiersprache [C](#) mit dem freien C-Compiler [AVR-](#)

[GCC](#) aus der [GNU Compiler Collection](#) erleichtern.

Vorausgesetzt werden Grundkenntnisse der Programmiersprache C. Diese Kenntnisse kann man sich online erarbeiten, z. B. mit dem [C Tutorial von Helmut Schellong](#). Nicht erforderlich sind Vorkenntnisse in der Programmierung von Mikrocontrollern, weder in Assembler noch in einer anderen Sprache.

In diesem Text wird häufig auf die Standardbibliothek avr-libc verwiesen, für die es eine [Online-Dokumentation](#) gibt, in der sich auch viele nützliche Informationen zum Compiler und zur Programmierung von AVR Controllern finden. Bei WinAVR gehört die avr-libc Dokumentation zum Lieferumfang und wird mitinstalliert.

Der Compiler und die Standardbibliothek avr-libc werden stetig weiterentwickelt. Erläuterungen und Beispiele beziehen sich auf den C-Compiler avr-gcc ab Version 3.4 und die avr-libc ab Version 1.4.3. Die Unterschiede zu älteren Versionen werden im Haupttext und Anhang zwar erläutert, Anfängern sei jedoch empfohlen, die aktuellen Versionen zu nutzen (für MS-Windows: aktuelle Version des [WinAVR](#)-Pakets; für Linux gibt es CDK4AVR: <http://cdk4avr.sf.net> oder auch fertige Pakete bei verschiedenen Distributionen.).

Das ursprüngliche Tutorial stammt von Christian Schifferle, viele neue Abschnitte und aktuelle Anpassungen von Martin Thomas.

Dieses Tutorial ist in PDF-Form hier erhältlich (nicht immer auf aktuellem Stand): [Media:AVR-GCC-Tutorial.pdf](#)

Benötigte Werkzeuge

Um eigene Programme für AVRs mittels avr-gcc/avr-libc zu erstellen und zu testen, wird folgende Hard- und Software benötigt:

- Platine oder Versuchsaufbau für die Aufnahme eines AVR Controllers, der vom avr-gcc Compiler unterstützt wird (alle ATmegas und die meisten AT90, siehe Dokumentation der avr-libc für unterstützte Typen). Dieses Testboard kann durchaus auch selbst gelötet oder auf einem Steckbrett aufgebaut werden. Einige Registerbeschreibungen dieses Tutorials beziehen sich auf den inzwischen veralteten AT90S2313. Der weitaus größte Teil des Textes ist aber für alle Controller der AVR-Familie gültig. Brauchbare Testplattformen sind auch das [STK500](#) und der [AVR Butterfly](#) von Atmel. Weitere Infos findet man [hier](#).
- Der avr-gcc Compiler und die avr-libc. Kostenlos erhältlich für nahezu alle Plattformen und Betriebssysteme. Für MS-Windows im Paket [WinAVR](#); für Unix/Linux siehe auch Hinweise im Artikel [AVR-GCC](#).
- Programmiersoftware und [-hardware](#) z. B. PonyProg (siehe auch: [Pony-Prog Tutorial](#)) oder [AVRDUDE](#) mit [STK200](#)-Dongle oder die von Atmel verfügbare Hard- und Software ([STK500](#), Atmel AVRISP, [AVR-Studio](#)).
- Nicht unbedingt erforderlich, aber zur Simulation und zum Debuggen unter MS-Windows recht nützlich: [AVR-Studio](#) (siehe Abschnitt [Exkurs: Makefiles](#)).
- Wer unter Windows und Linux gleichermaßen entwickeln will, der sollte sich die [IDE Eclipse for C/C++ Developers](#) und das [AVR-Eclipse Plugin](#) ansehen, beide sind unter Windows und Linux einfach zu installieren. Ebenfalls unter Linux und Windows verfügbar ist die Entwicklungsumgebung [Code::Blocks](#) (aktuelle, stabile Versionen sind als Nightly Builds regelmäßig im [Forum](#) verfügbar). Innerhalb dieser Entwicklungsumgebung können ohne die Installation zusätzlicher Plugins "AVR-Projekte" angelegt werden. Für Linux gibt es auch noch das [KontrollerLab](#).

Was tun, wenn 's nicht "klappt"?

- Herausfinden, ob es tatsächlich ein avr(-gcc) spezifisches Problem ist oder nur die eigenen C-Kenntnisse einer Auffrischung bedürfen. Allgemeine C-Fragen kann man eventuell "beim freundlichen Programmierer zwei Büro-, Zimmer- oder Haustüren weiter" loswerden. Ansonsten: [C-Buch](#) (gibt's auch "gratis" online) lesen.
- Die [AVR Checkliste](#) durcharbeiten.
- Die [Dokumentation der avr-libc](#) lesen, vor allem (aber nicht nur) den Abschnitt Related Pages/**Frequently Asked Questions** = Oft gestellte Fragen (und Antworten dazu). Z.Zt leider nur in englischer Sprache verfügbar.
- Den Artikel [AVR-GCC](#) in diesem Wiki lesen.
- Das [GCC-Forum auf www.mikrocontroller.net](#) nach vergleichbaren Problemen absuchen.
- Das avr-gcc-Forum bei [AVRfreaks](#) nach vergleichbaren Problemen absuchen.
- Das [Archiv der avr-gcc Mailing-Liste](#) nach vergleichbaren Problemen absuchen.
- Nach Beispielcode suchen. Vor allem im *Projects*-Bereich von [AVRfreaks](#) (anmelden).
- Google oder yahoo befragen schadet nie.
- Bei Problemen mit der Ansteuerung interner AVR-Funktionen mit C-Code: das Datenblatt des Controllers lesen (ganz und am Besten zweimal). Datenblätter sind auf den [Atmel Webseiten](#) als pdf-Dateien verfügbar. Das komplette Datenblatt (complete) und nicht die Kurzfassung (summary) verwenden.
- Die Beispieleprogramme im [AVR-Tutorial](#) sind zwar in AVR-Assembler verfasst, Erläuterungen und Vorgehensweisen sind aber auch auf C-Programme übertragbar.
- Einen Beitrag in eines der Foren oder eine Mail an die Mailing-Liste schreiben. Dabei möglichst viel Information geben: Controller, Compilerversion, genutzte Bibliotheken, Ausschnitte aus dem Quellcode oder besser ein [Testprojekt](#) mit allen notwendigen Dateien, um das Problem nachzuvollziehen, sowie genaue Fehlermeldungen bzw. Beschreibung des Fehlverhaltens. Bei Ansteuerung externer Geräte die Beschaltung beschreiben oder skizzieren

(z. B. mit [Andys ASCII Circuit](#)). Siehe dazu auch: "[Wie man Fragen richtig stellt](#)".

Erzeugen von Maschinencode

Aus dem C-Quellcode erzeugt der avr-gcc Compiler (zusammen mit Hilfsprogrammen wie z. B. Präprozessor, Assembler und Linker) Maschinencode für den AVR-Controller. Üblicherweise liegt dieser Code dann im Intel Hex-Format vor ("Hex-Datei"). Die Programmiersoftware (z. B. [AVRDUDE](#), PonyProg oder AVRStudio/STK500-plugin) liest diese Datei ein und überträgt die enthaltene Information (den Maschinencode) in den Speicher des Controllers. Im Prinzip sind also "nur" der avr-gcc-Compiler (und wenige Hilfsprogramme) mit den "richtigen" Optionen aufzurufen, um aus C-Code eine "Hex-Datei" zu erzeugen. Grundsätzlich stehen dazu zwei verschiedene Ansätze zur Verfügung:

- Die Verwendung einer integrierten Entwicklungsumgebung (IDE = Integrated Development Environment), bei der alle Einstellungen z. B. in Dialogboxen durchgeführt werden können. Unter Anderem kann AVRStudio ab Version 4.12 (kostenlos auf [atmel.com](#)) zusammen mit WinAVR als integrierte Entwicklungsumgebung für den Compiler avr-gcc genutzt werden (dazu müssen AVRStudio und WinAVR auf dem Rechner installiert sein). Weitere IDEs (ohne Anspruch auf Vollständigkeit): [Eclipse for C/C++ Developers](#) (d.h. inkl. CDT) und das [AVR-Eclipse Plugin](#) (für diverse Plattformen, u.a. Linux und MS Windows, IDE und Plugin kostenlos), [KontrollerLab](#) (Linux/KDE, kostenlos), [AtmanAvr](#) (MS Windows, relativ günstig), KamAVR (MS-Windows, kostenlos, wird augenscheinlich nicht mehr weiterentwickelt), [VMLab](#) (MS Windows, ab Version 3.12 ebenfalls kostenlos). Integrierte Entwicklungsumgebungen unterscheiden sich stark in Ihrer Bedienung und stehen auch nicht für alle Plattformen zur Verfügung, auf denen der Compiler ausführbar ist (z. B. AVRStudio nur für MS-Windows). Zur Anwendung des avr-gcc Compilers mit IDEs sei hier auf deren Dokumentation verwiesen.
- Die Nutzung des Programms make mit passenden Makefiles. In den folgenden Abschnitten wird die Generierung von Maschinencode für einen AVR ("hex-Datei") aus C-Quellcode ("c-Dateien") anhand von "make" und den "Makefiles" näher erläutert. Viele der darin beschriebenen Optionen findet man auch im Konfigurationsdialog des avr-gcc-Plugins von AVRStudio (AVRStudio generiert ein makefile in einem Unterverzeichnis des Projektverzeichnisses).

Beim Wechsel vom makefile-Ansatz nach WinAVR-Vorlage zu AVRStudio ist darauf zu achten, dass AVRStudio (Stand: AVRStudio Version 4.13) bei einem neuen Projekt die Optimierungsoption (vgl. Abschnitt [Exkurs: Makefiles](#), typisch: -Os) nicht einstellt und die mathematische Bibliothek der avr-libc (libm.a, Linker-Option -lm) nicht einbindet. (Hinweis: Bei Version 4.16 wird beides bereits gesetzt). Beides ist Standard bei Verwendung von makefiles nach WinAVR-Vorlage und sollte daher auch im Konfigurationsdialog des avr-gcc-Plugins von AVRStudio "manuell" eingestellt werden, um auch mit AVRStudio kompakten Code zu erzeugen.

Einführungsbeispiel

Zum Einstieg ein kleines Beispiel, an dem die Nutzung des Compilers und der Hilfsprogramme (der sogenannten *Toolchain*) demonstriert wird. Detaillierte Erläuterungen folgen in den weiteren Abschnitten dieses Tutorials.

Das Programm soll auf einem AVR Mikrocontroller einige Ausgänge ein- und andere ausschalten. Das Beispiel ist für einen ATmega16 programmiert ([Datenblatt](#)), kann aber sinngemäß für andere Controller der AVR-Familie modifiziert werden.

Zunächst der Quellcode der Anwendung, der in einer Text-Datei mit dem Namen *main.c* abgespeichert wird.

```

/* Alle Zeichen zwischen Schrägstrich-Stern
   und Stern-Schrägstrich sind lediglich Kommentare */

// Zeilenkommentare sind ebenfalls möglich
// alle auf die beiden Schrägstriche folgenden
// Zeichen einer Zeile sind Kommentar

#include <avr/io.h>           // (1)

int main (void) {           // (2)

    DDRB = 0xff;            // (3)
    PORTB = 0x03;          // (4)

    while(1) {              // (5a)
        /* "leere" Schleife*/ // (5b)
    }                        // (5c)

    /* wird nie erreicht */
    return 0;               // (6)
}

```

- In der mit (1) markierten Zeile wird eine so genannte Header-Datei eingebunden. In io.h sind die Registernamen definiert, die im späteren Verlauf genutzt werden.
- Bei (2) beginnt das eigentliche Programm. Jedes C-Programm beginnt mit den Anweisungen in der Funktion main.
- Die Anschlüsse eines AVR ("Beinchen") werden zu Blöcken zusammengefasst, einen solchen Block bezeichnet man als Port. Beim ATmega16 hat jeder Port 8 Anschlüsse, bei kleineren AVR's können einem Port auch weniger als 8 Anschlüsse zugeordnet sein. Da per Definition ([Datenblatt](#)) alle gesetzten Bits in einem Richtungsregister den entsprechenden Anschluss auf Ausgang schalten, werden mit DDRB=0xff alle Anschlüsse des Ports B zu Ausgängen.
- (4) stellt die Werte der Ausgänge ein. Die den ersten beiden Bits des Ports zugeordneten Anschlüsse (PB0 und PB1) werden 1, alle anderen Anschlüsse des Ports B (PB2-PB7) zu 0. Aktivierte Ausgänge (logisch 1 oder "high") liegen auf Betriebsspannung (VCC, meist 5 Volt), nicht aktivierte Ausgänge

führen 0 Volt (GND, Bezugspotential).

- (5) ist die so genannte Hauptschleife (main-loop). Dies ist eine Programmschleife, welche kontinuierlich wiederkehrende Befehle enthält. In diesem Beispiel ist sie leer. Der Controller durchläuft die Schleife immer wieder, ohne dass etwas passiert (außer das Strom verbraucht wird). Eine solche Schleife ist notwendig, da es auf dem Controller kein Betriebssystem gibt, das nach Beendigung des Programmes die Kontrolle übernehmen könnte. Ohne diese Schleife wäre der Zustand des Controllers nach dem Programmende undefiniert.
- (6) wäre das Programmende. Die Zeile ist nur aus Gründen der C-Kompatibilität enthalten: `int main(void)` besagt, dass die Funktion einen Wert zurückgibt. Die Anweisung wird aber nicht erreicht, da das Programm die Hauptschleife nie verlässt.

Um diesen Quellcode in ein auf dem Controller lauffähiges Programm zu übersetzen, wird hier ein Makefile genutzt. Das verwendete Makefile findet sich auf der Seite [Beispiel Makefile](#) und basiert auf der Vorlage, die in WinAVR mitgeliefert wird und wurde bereits angepasst (Controllertyp ATmega16). Man kann das Makefile bearbeiten und an andere Controller anpassen oder sich mit dem Programm MFile menügesteuert ein Makefile "zusammenklicken". Das Makefile speichert man unter dem Namen Makefile (ohne Endung) im selben Verzeichnis, in dem auch die Datei `main.c` mit dem Programmcode abgelegt ist. Detailliertere Erklärungen zur Funktion von Makefiles finden sich im folgenden Abschnitt [Exkurs: Makefiles](#).

```
D:\tmp\gcc_tut\quickstart>dir
```

```
Verzeichnis von D:\tmp\gcc_tut\quickstart
```

```
28.11.2006  22:53  <DIR>          .
28.11.2006  22:53  <DIR>          ..
28.11.2006  20:06                118 main.c
28.11.2006  20:03           16.810 Makefile
                2 Datei(en)           16.928 Bytes
```

Nun gibt man `make all` ein. Falls das mit WinAVR installierte Programmiers Notepad genutzt wird, gibt es dazu einen Menüpunkt im Tools Menü. Sind alle Einstellungen korrekt, entsteht eine Datei `main.hex`, in der der Code für den AVR enthalten ist.

```
D:\tmp\gcc_tut\quickstart>make all
```

```
----- begin -----
avr-gcc (GCC) 3.4.6
Copyright (C) 2006 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

```
Compiling C: main.c
avr-gcc -c -mmcu=atmega16 -I. -gdwarf-2 -DF_CPU=1000000UL -Os -funsigned-char -f
unsigned-bitfields -fpack-struct -fshort-enums -Wall -Wstrict-prototypes -Wundef
-Wa,-adhlns=obj/main.lst -std=gnu99 -Wundef -MD -MP -MF .dep/main.o.d main.c -
o obj/main.o
```

```
Linking: main.elf
avr-gcc -mmcu=atmega16 -I. -gdwarf-2 -DF_CPU=1000000UL -Os -funsigned-char -funs
igned-bitfields -fpack-struct -fshort-enums -Wall -Wstrict-prototypes -Wundef -W
a,-adhlns=obj/main.o -std=gnu99 -Wundef -MD -MP -MF .dep/main.elf.d obj/main.o
--output main.elf -Wl,-Map=main.map,--cref -lm
```

```
Creating load file for Flash: main.hex
avr-objcopy -O ihex -R .eeprom main.elf main.hex
```

```
[...]
```

Der Inhalt der hex-Datei kann nun zum Controller übertragen werden. Dies kann z. B. über In-System-Programming (ISP) erfolgen, das im [AVR-Tutorial: Equipment](#) beschrieben ist. Makefiles nach der WinAVR/MFile-Vorlage sind für die Nutzung des Programms [AVRDUDE](#) vorbereitet. Wenn man den Typ und Anschluss des Programmiergerätes richtig eingestellt hat, kann mit `make program` die Übertragung mittels AVRDUDE gestartet werden. Jede andere Software, die hex-Dateien lesen und zu einem AVR übertragen kann (z. B. [Ponyprog](#), yapp, AVRStudio), kann natürlich ebenfalls genutzt werden.

Startet man nun den Controller (Reset-Taster oder Stromzufuhr aus/an), werden vom Programm die Anschlüsse PB0 und PB1 auf 1 gesetzt. Man kann mit einem Messgerät nun an diesem Anschluss die Betriebsspannung messen oder eine LED leuchten lassen (Anode an den Pin, Vorwiderstand nicht vergessen). An den Anschlüssen PB2-PB7 misst man 0 Volt. Eine mit der Anode mit einem dieser Anschlüsse verbundene LED leuchtet nicht.

Exkurs: Makefiles

→ siehe [AVR-GCC-Tutorial/Exkurs Makefiles](#)

Ganzzahlige (Integer) Datentypen

Bei der Programmierung von Mikrocontrollern ist die Definition einiger ganzzahliger Datentypen sinnvoll, an denen eindeutig die Bit-Länge abgelesen werden kann.

Standardisierte Datentypen werden in der Header-Datei `stdint.h` definiert. Zur Nutzung der standardisierten Typen bindet man die "Definitionsdatei" wie folgt ein:

```
// ab avr-libc Version 1.2.0 möglich und empfohlen:
#include <stdint.h>
```

```
// veraltet: #include <inttypes.h>
```

Einige der dort definierten Typen (avr-libc Version 1.0.4):

```
typedef signed char      int8_t;
typedef unsigned char    uint8_t;

typedef short            int16_t;
typedef unsigned short   uint16_t;

typedef long             int32_t;
typedef unsigned long    uint32_t;

typedef long long        int64_t;
typedef unsigned long long uint64_t;
```

- int8_t steht für einen 8-Bit Integer mit einem Wertebereich -128 bis +127.
- uint8_t steht für einen 8-Bit Integer ohne Vorzeichen (unsigned int) mit einem Wertebereich von 0 bis 255
- int16_t steht für einen 16-Bit Integer mit einem Wertebereich -32768 bis +32767.
- uint16_t steht für einen 16-Bit Integer ohne Vorzeichen (unsigned int) mit einem Wertebereich von 0 bis 65535.

Die Typen ohne vorangestelltes *u* werden als vorzeichenbehaftete Zahlen abgespeichert. Typen mit vorgestelltem *u* dienen der Ablage von positiven Zahlen (inkl. 0). Siehe dazu auch: [Dokumentation der avr-libc](#) Abschnitt Modules/(Standard) Integer Types.

Bitfelder

Beim Programmieren von Mikrocontrollern 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 Bits breit ist.

Hier bietet uns die Programmiersprache C ein mächtiges Werkzeug an, mit dessen Hilfe wir 8 Bits in eine einzelne Bytevariable zusammenfassen und (fast) wie 8 einzelne Variablen ansprechen können. Die Rede ist von so genannten Bitfeldern. Diese werden als Strukturelemente definiert. Sehen wir uns dazu doch am besten gleich ein Beispiel an:

```
struct {
    unsigned bStatus_1:1; // 1 Bit für bStatus_1
    unsigned bStatus_2:1; // 1 Bit für bStatus_2
    unsigned bNochNBit:1; // Und hier noch mal ein Bit
    unsigned 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;
```

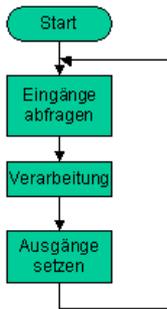
Bitfelder sparen Platz im RAM, zu Lasten von Platz im Flash, verschlechtern aber unter Umständen die Les- und Wartbarkeit des Codes. Anfängern wird deshalb geraten, ein "ganzes" Byte (uint8_t) zu nutzen, auch wenn nur ein Bitwert gespeichert werden soll.

Grundsätzlicher Programmaufbau eines μ C-Programms

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

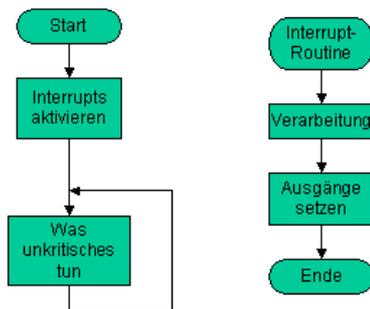
Sequentieller Programmablauf

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



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.



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 sowohl auslesen als auch beschreiben.

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.

Einzelne Register sind bei allen AVRs 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 integrierten Hardware UART bzw. USART verfügen.

Die Namen der Register sind in den Headerdateien zu den entsprechenden AVR-Typen definiert. Dazu muss man den Namen der controllerspezifischen Headerdatei nicht kennen. Es reicht aus, die allgemeine Headerdatei `avr/io.h` einzubinden:

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

Ist im Makefile der MCU-Typ z. B. mit dem Inhalt `atmega8` definiert (und wird somit per `-mmcu=atmega8` an den Compiler übergeben), wird beim Einlesen der `io.h`-Datei implizit ("automatisch") auch die `iom8.h`-Datei mit den Register-Definitionen für den ATmega8 eingelesen.

Intern wird diese "Automatik" wie folgt realisiert: Der Controllertyp wird dem Compiler als Parameter übergeben (vgl. `avr-gcc -c -mmcu=atmega16 [...]` im Einführungsbeispiel). Wird ein Makefile nach der WinAVR/mfile-Vorlage verwendet, setzt man die Variable `MCU`, der Inhalt dieser Variable wird dann an passender Stelle für die Compilerparameter verwendet. Der Compiler definiert intern eine dem `mmcu`-Parameter zugeordnete "Variable" (genauer: ein Makro) mit dem Namen des Controllers, vorangestelltem `__AVR__` und angehängten Unterstrichen (z.B. wird bei `-mmcu=atmega16` das Makro `__AVR_ATmega16__` definiert). Beim Einbinden der Header-Datei `avr/io.h` wird geprüft, ob das jeweilige Makro definiert ist und die zum Controller passende Definitionsdatei eingelesen. Zur Veranschaulichung einige Ausschnitte aus einem Makefile:

```
[...]
# MCU Type ("name") setzen:
MCU = atmega16
[...]

[...]
## Verwendung des Inhalts von MCU (hier atmega16) fuer die
## Compiler- und Assembler-Parameter
ALL_CFLAGS = -mmcu=$(MCU) -I. $(CFLAGS) $(GENDEPFLAGS)
ALL_CPPFLAGS = -mmcu=$(MCU) -I. -x c++ $(CPPFLAGS) $(GENDEPFLAGS)
ALL_ASFLAGS = -mmcu=$(MCU) -I. -x assembler-with-cpp $(ASFLAGS)
[...]
```

```
[...]
## Aufruf des Compilers:
## mit den Parametern ($(ALL_CFLAGS) ist -mmcu=$(MCU)[...] = -mmcu=atmega16[...]
$(OBJDIR)/%.o : %.c
    @echo
    @echo $(MSG_COMPILING) $<
    $(CC) -c $(ALL_CFLAGS) $< -o $@
[...]
```

Da `--mmcu=atmega16` übergeben wurde, wird `__AVR_ATmega16__` definiert und kann in `avr/io.h` zur Fallunterscheidung genutzt werden:

```
// avr/io.h
// (bei WinAVR-Standardinstallation in C:\WinAVR\avr\include\avr)
[...]
```

```
#if defined (__AVR_AT94K__)
# include <avr/ioat94k.h>
// [...]
#elif defined (__AVR_ATmega16__)
// da __AVR_ATmega16__ definiert ist, wird avr/iom16.h eingebunden:
# include <avr/iom16.h>
// [...]
#else
# if !defined(__COMPILING_AVR_LIBC__)
#   warning "device type not defined"
# endif
#endif
```

Die Beispiele in den folgenden Abschnitten demonstrieren den Zugriff auf Register anhand der Register für I/O-Ports (PORTx, DDRx, PINx), die Vorgehensweise ist jedoch für alle Register (z.B. die des UART, ADC, SPI) analog.

Schreiben in Register

Zum Schreiben kann man Register einfach wie eine Variable setzen. In Quellcodes, die für ältere Versionen des `avr-gcc`/der `avr-libc` entwickelt wurden, erfolgt der Schreibzugriff über die Funktion `outp()`. Aktuelle Versionen des Compilers unterstützen den Zugriff nun direkt und `outp()` ist nicht mehr erforderlich.

Beispiel:

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

...

int main(void)
{
    /* Setzt das Richtungsregister des Ports A auf 0xff
       (alle Pins als Ausgang, vgl. Abschnitt Zugriff auf Ports): */
    DDRA = 0xff;

    /* Setzt PortA auf 0x03, Bit 0 und 1 "high", restliche "low": */
    PORTA = 0x03;
    ...

    // Setzen der Bits 0,1,2,3 und 4
    // Binär 00011111 = Hexadezimal 1F
    DDRB = 0x1F;    /* direkte Zuweisung - unübersichtlich */

    /* Ausführliche Schreibweise: identische Funktionalität, mehr Tipparbeit
       aber übersichtlicher und selbsterklärend: */
    DDRB = (1 << DDB0) | (1 << DDB1) | (1 << DDB2) | (1 << DDB3) | (1 << DDB4);
    // (Hinweis: für Einsteiger (wie mich) nicht selbsterklärend. Was macht |, was macht <<, gibt es auch DDB5-7?)
}
```

Die ausführliche Schreibweise sollte bevorzugt verwendet werden, da dadurch die Zuweisungen selbsterklärend sind und somit der Code leichter nachvollzogen werden kann. Atmel verwendet sie auch bei Beispielen in Datenblättern und in den allermeisten Quellcodes zu Application-Notes.

Der gcc C-Compiler (genauer der Präprozessor) unterstützt ab Version 4.3.0 Konstanten im Binärformat, z.B. `DDRB = 0b00011111` (für WinAVR wurden schon ältere Versionen des gcc entsprechend angepasst). Diese Schreibweise ist jedoch nicht standardkonform und man sollte sie daher insbesondere dann nicht verwenden, wenn Code mit anderen ausgetauscht oder mit anderen Compilern bzw. älteren Versionen des gcc genutzt werden soll.

Verändern von Registerinhalten

Einzelne Bits setzt und löscht man "Standard-C-konform" mittels logischer (Bit-) Operationen.

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

Es wird jeweils nur der Zustand des angegebenen Bits geändert, der vorherige Zustand der anderen Bits bleibt erhalten.

Beispiel:

```
#include <avr/io.h>
...
#define MEINBIT 2
...
PORTA |= (1 << MEINBIT);    /* setzt Bit 2 an PortA auf 1 */
PORTA &= ~(1 << MEINBIT);  /* loescht Bit 2 an PortA */
```

Mit dieser Methode lassen sich auch mehrere Bits eines Registers gleichzeitig setzen und löschen.

Beispiel:

```
#include <avr/io.h>
...
DDRA &= ~( (1<<PA0) | (1<<PA3) ); /* PA0 und PA3 als Eingaenge */
PORTA |= (1<<PA0) | (1<<PA3);    /* Interne Pull-Up fuer beide einschalten */
```

In Quellcodes, die für ältere Versionen des avr-gcc/der avr-libc entwickelt wurden, werden einzelne Bits mittels der Funktionen sbi und cbi gesetzt bzw. gelöscht. Beide Funktionen sind nicht mehr erforderlich.

Siehe auch:

- [Bitmanipulation](#)
- [Dokumentation der avr-libc](#) Abschnitt Modules/Special Function Registers

Lesen aus Registern

Zum Lesen kann man auf Register einfach wie auf eine Variable zugreifen. In Quellcodes, die für ältere Versionen des avr-gcc/der avr-libc entwickelt wurden, erfolgt der Lesezugriff über die Funktion inp(). Aktuelle Versionen des Compilers unterstützen den Zugriff nun direkt und inp() ist nicht mehr erforderlich.

Beispiel:

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

uint8_t foo;

//...

int main(void)
{
    /* kopiert den Status der Eingabepins an PortB
       in die Variable foo: */
    foo = PINB;
    //...
}
```

Die Abfrage der Zustände von Bits erfolgt durch Einlesen des gesamten Registerinhalts und ausblenden der Bits deren Zustand nicht von Interesse ist. Einige Beispiele zum Prüfen ob Bits gesetzt oder gelöscht sind:

```
#define MEINBIT0 0
#define MEINBIT2 2

uint8_t i;

extern test1();

// Funktion test1 aufrufen, wenn Bit 0 in Register PINA gesetzt (1) ist
i = PINA;    // Inhalt in Arbeitsvariable
i = i & 0x01; // alle Bits bis auf Bit 0 ausblenden (logisches und)
// falls das Bit gesetzt war, hat i den Inhalt 1
if ( i != 0 ) { // Ergebnis ungleich 0 (wahr)?
    test1();    // dann muss Bit 0 in i gesetzt sein -> Funktion aufrufen
}
// verkürzt:
if ( ( PINA & 0x01 ) != 0 ) {
    test1();
}
// nochmals verkürzt:
if ( PINA & 0x01 ) {
    test1();
}
// mit definierter Bitnummer:
if ( PINA & ( 1 << MEINBIT0 ) ) {
    test1();
}
```

```

// Funktion aufrufen, wenn Bit 0 oder Bit 2 gesetzt ist. (Bit 0 und 2 also Wert 5)
// (Bedenke: Bit 0 hat Wert 1, Bit 1 hat Wert 2 und Bit 2 hat Wert 4)
if ( PINA & 0x05 ) {
    test1(); // Vergleich <> 0 (wahr), also muss Bit 0 oder 2 gesetzt sein
}
// mit definierten Bitnummern:
if ( PINA & ( ( 1 << MEINBIT0 ) | ( 1 << MEINBIT2 ) ) ) {
    test1();
}

// Funktion aufrufen, wenn Bit 0 und Bit 2 gesetzt sind
if ( ( PINA & 0x05 ) == 0x05 ) { // nur wahr, wenn beide Bits gesetzt
    test1();
}

// Funktion test2() aufrufen, wenn Bit 0 gelöscht (0) ist
i = PINA; // einlesen in temporäre Variable
i = i & 0x01; // maskieren von Bit 0
if ( i == 0 ) { // Vergleich ist wahr, wenn Bit 0 nicht gesetzt ist
    test2();
}
// analog mit !-Operator (not)
if ( !i ) {
    test2();
}
// nochmals verkürzt:
if ( !( PINA & 0x01 ) ) {
    test2();
}
}

```

Die AVR-Bibliothek (avr-libc) stellt auch Funktionen (Makros) zur Abfrage eines einzelnen Bits eines Registers zur Verfügung, diese sind bei anderen Compilern meist nicht verfügbar (können aber dann einfach durch Makros "nachgerüstet" werden).

`bit_is_set (<Register>,<Bitnummer>)`

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, 3 für Bit2 etc.

`bit_is_clear (<Register>,<Bitnummer>)`

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.

Die Funktionen (eigentlich Makros) `bit_is_clear` bzw. `bit_is_set` sind nicht erforderlich, man kann und sollte C-Syntax verwenden, die universell verwendbar und portabel ist. Siehe auch [Bitmanipulation](#).

Warten auf einen bestimmten Zustand

Es gibt in der Bibliothek avr-libc Funktionen, die warten, bis ein bestimmter Zustand eines Bits erreicht ist. Es ist allerdings normalerweise eine eher unschöne Programmieretechnik, da in diesen Funktionen "blockierend" gewartet wird. Der Programmablauf bleibt also an dieser Stelle stehen, bis das maskierte Ereignis erfolgt ist. Setzt man den Watchdog ein, muss man darauf achten, dass dieser auch noch getriggert wird (Zurücksetzen des Watchdogtimers).

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.

```

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

/* Warten bis Bit Nr. 2 (das dritte Bit) in Register PINA gesetzt (1) ist */

#define WARTEPIN PINA
#define WARTEBIT PA2

// mit der avr-libc Funktion:
loop_until_bit_is_set(WARTEPIN, WARTEBIT);

// dito in "C-Standard":
// Durchlaufe die (leere) Schleife solange das WARTEBIT in Register WARTEPIN
// _nicht_ ungleich 0 (also 0) ist.
while ( !(WARTEPIN & (1 << WARTEBIT)) );
...

```

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.

```

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

/* Warten bis Bit Nr. 4 (das fuenfte Bit) in Register PINB geloescht (0) ist */
#define WARTEPIN PINB
#define WARTEBIT PB4

```

```
// avr-libc-Funktion:
loop_until_bit_is_clear(WARTEPIN, WARTEBIT);

// dito in "C-Standard":
// Durchlaufe die (leere) Schleife solange das WARTEBIT in Register WARTEPIN
// gesetzt (1) ist
while ( WARTEPIN & (1<<WARTEBIT) );
...

```

Universeller und auch auf andere Plattformen besser übertragbar ist die Verwendung von C-Standardoperationen.

Siehe auch:

- [Dokumentation der avr-libc](#) Abschnitt Modules/Special Function Registers
- [Bitmanipulation](#)

16-Bit Register (ADC, ICR1, OCR1, TCNT1, UBRR)

Einige der Portregister in den AVR-Controllern sind 16 Bit breit. Im Datenblatt sind diese Register üblicherweise mit dem Suffix "L" (LSB) und "H" (MSB) versehen. Die avr-libc definiert zusätzlich die meisten dieser Variablen die Bezeichnung ohne "L" oder "H". Auf diese kann direkt zugewiesen bzw. zugegriffen werden. Die Konvertierung von 16-bit Wort nach 2*8-bit Byte erfolgt intern.

```
#include <avr/io.h>
#include <stdint.h>
...
uint16_t foo;

foo=ADC; /* setzt die Wort-Variablen foo auf den Wert der letzten AD-Wandlung */

```

Falls benötigt, kann eine 16-Bit Variable auch recht einfach manuell in ihre zwei 8-Bit Bestandteile zerlegt werden. Folgendes Beispiel demonstriert dies anhand des pseudo- 16-Bit Registers UBRR.

```
/* Diese Variante ist normal am effizientesten */
#include <avr/io.h>
#include <stdint.h>
#ifndef F_CPU
#define F_CPU 3686400
#endif
#define UART_BAUD_RATE 9600

typedef union {
    uint16_t i16;
    struct {
        uint8_t i8l;
        uint8_t i8h;
    };
} convert16to8;

...
convert16to8 baud;
baud.i16 = F_CPU / (UART_BAUD_RATE * 16L) -1;
UBRRH = baud.i8h;
UBRRL = baud.i8l;
...

/* Alternative 1:*/
#include <inttypes.h>

uint16_t wFoo16;
uint8_t bFooLow, bFooHigh;

...
wFoo16 = 0xAA55; /* zu "zerlegende" 16Bit-Integer */
bFooHigh = (uint8_t)(wFoo16 >> 8); /* MS-Byte */
bFooLow = (uint8_t)(wFoo16); /* LS-Byte */

/* Alternative 2:*/

#define us0(Data) (*((unsigned char *)&Data))
#define us1(Data) (*((unsigned char *)&Data+1))
bFooHigh = us1(wFoo16);
bFooLow = us0(wFoo16);
...

```

Bei einigen AVR-Typen (z.B. ATmega8, ATmega16) teilen sich UBRRH und UCSRC die gleiche Memory-Adresse. Damit der AVR trotzdem zwischen den beiden Registern unterscheiden kann, bestimmt das Bit7 (URSEL) welches Register tatsächlich beschrieben werden soll. `1000 0011` (0x83) adressiert demnach UCSRC und übergibt den Wert 3 und `0000 0011` (0x3) adressiert UBRRH und übergibt ebenfalls den Wert 3.

Speziell bei den **16-Bit-Timern** und auch beim **ADC** ist es bei allen Zugriffen auf Datenregister erforderlich, dass diese Daten synchronisiert sind. Wenn z.B. bei einem 16-Bit-Timer das High-Byte des Zählregisters gelesen wurde und vor dem Lesezugriff auf das Low-Byte ein Überlauf des Low-Bytes stattfindet, erhält man einen völlig unsinnigen Wert. Auch die Compare-Register müssen synchron geschrieben werden, da es ansonsten zu unerwünschten Compare-Ereignissen kommen kann.

Beim ADC besteht das Problem darin, dass zwischen den Zugriffen auf die beiden Teilregister eine Wandlung beendet werden kann und der ADC ein neues Ergebnis in ADCL und ADCH schreiben will, wodurch High- und Low-Byte nicht zusammenpassen.

Um diese Datenmüllproduktion zu verhindern, gibt es in beiden Fällen eine Synchronisation, die jeweils durch den Zugriff auf das Low-Byte ausgelöst wird:

- Bei den Timer-Registern (das gilt für alle TCNT-, OCR- und ICR-Register bei den 16-Bit-Timern) wird bei einem **Lesezugriff** auf das Low-Byte automatisch das High-Byte in ein temporäres Register, das ansonsten nach außen nicht sichtbar ist, geschoben. Greift man nun *anschließend* auf das High-Byte zu, dann wird eben dieses temporäre Register gelesen.
- Bei einem **Schreibzugriff** auf eines der genannten Register wird das High-Byte in besagtem temporären Register zwischengespeichert und erst beim Schreiben des Low-Bytes werden *beide* gleichzeitig in das eigentliche Register übernommen.

Das bedeutet für die Reihenfolge beim **Lesezugriff: Erst Low-Byte, dann High-Byte** und für den **Schreibzugriff: Erst High-Byte, dann Low-Byte**.

Des Weiteren ist zu beachten, dass es für all diese 16-Bit-Register nur ein einziges temporäres Register gibt, so dass das Auftreten eines Interrupts, in dessen Handler ein solches Register manipuliert wird, bei einem durch ihn unterbrochenen Zugriff i.d.R. zu Datenmüll führt. **16-Bit-Zugriffe sind generell nicht atomar!** Wenn mit **Interrupts** gearbeitet wird, kann es erforderlich sein, vor einem solchen Zugriff auf ein 16-Bit-Register die Interrupt-Bearbeitung zu deaktivieren.

Beim ADC-Datenregister ADCH/ADCL ist die Synchronisierung anders gelöst. Hier wird beim Lesezugriff (ADCH/ADCL sind logischerweise Read-only) auf das Low-Byte ADCL beide Teilregister für Zugriffe seitens des ADC so lange gesperrt, bis das High-Byte ADCH ausgelesen wurde. Dadurch kann der ADC nach einem Zugriff auf ADCL keinen neuen Wert in ADCH/ADCL ablegen, bis ADCH gelesen wurde. Ergebnisse von Wandlungen, die zwischen einem Zugriff auf ADCL und ADCH beendet werden, gehen verloren!

Nach einem Zugriff auf ADCL muss grundsätzlich ADCH gelesen werden!

In beiden Fällen (also sowohl bei den Timern als auch beim ADC) werden vom C-Compiler 16-Bit-Pseudo-Register zur Verfügung gestellt (z.B. TCNT1H/TCNT1L -> TCNT1, ADCH/ADCL -> ADC bzw. ADCW), bei deren Verwendung der Compiler automatisch die richtige Zugriffsreihenfolge regelt. **In C-Programmen sollten grundsätzlich diese 16-Bit-Register verwendet werden.** Sollte trotzdem ein Zugriff auf ein Teilregister erforderlich sein, sind obige Angaben zu berücksichtigen.

Es ist darauf zu achten, dass auch ein Zugriff auf die 16-Bit-Register vom Compiler in zwei 8-Bit-Zugriffe aufgeteilt wird und dementsprechend genauso nicht-atomar ist wie die Einzelzugriffe. Auch hier gilt, dass u.U. die Interrupt-Bearbeitung gesperrt werden muss, um Datenmüll zu vermeiden.

Beim ADC gibt es für den Fall, dass eine Auflösung von 8 Bit ausreicht, die Möglichkeit, das Ergebnis "linksbündig" in ADCH/ADCL auszurichten, so dass die relevanten 8 MSB in ADCH stehen. In diesem Fall muss bzw. sollte nur ADCH ausgelesen werden.

ADC und ADCW sind unterschiedliche Bezeichner für das selbe Registerpaar. Üblicherweise kann man in C-Programmen ADC verwenden, was analog zu den anderen 16-Bit-Registern benannt ist. ADCW (ADC Word) existiert nur deshalb, weil die Headerdateien auch für Assembler vorgesehen sind und es bereits einen Assembler-Befehl namens *adc* gibt.

Im Umgang mit 16-Bit Registern siehe auch:

- [Dokumentation der avr-libc](#) Abschnitt Related Pages/Frequently Asked Questions/Nr. 8
- Datenblatt Abschnitt *Accessing 16-bit Registers*

IO-Register als Parameter und Variablen

Um Register als Parameter für eigene Funktionen übergeben zu können, muss man sie als einen volatile uint8_t Pointer übergeben. Zum Beispiel:

```
uint8_t key_pressed(const volatile uint8_t *inputreg, uint8_t inputbit)
{
    static uint8_t last_state = 0;

    if ( last_state == ( *inputreg & (1<<inputbit) ) ) {
        return 0; /* keine Änderung */
    }

    /* Wenn doch, warten bis etwaiges Prellen vorbei ist: */
    _delay_ms(20);

    /* Zustand für nächsten Aufruf merken: */
    last_state = ( *inputreg & (1<<inputbit) );

    /* und den entprellten Tastendruck zurückgeben: */
    return ( *inputreg & (1<<inputbit) );
}

/* Beispiel für einen Funktionsaufruf: */

#include <avr/io.h>

uint8_t i;
```

```
//...
i = key_pressed( &PINB, PB1 );
//...
```

Ein Aufruf der Funktion mit call by value würde Folgendes bewirken: Beim Funktionseintritt wird nur eine Kopie des momentanen Portzustandes angefertigt, die sich unabhängig vom tatsächlichen Zustand des Ports nicht mehr ändert, womit die Funktion wirkungslos wäre. Die Übergabe eines Zeigers wäre die Lösung, wenn der Compiler nicht optimieren würde. Denn dadurch wird im Programm nicht von der Hardware gelesen, sondern wieder nur von einem Abbild im Speicher. Das Ergebnis wäre das gleiche wie oben. Mit dem Schlüsselwort volatile sagt man nun dem Compiler, dass die entsprechende Variable entweder durch andere Software-Routinen (Interrupts) oder durch die Hardware verändert werden kann.

Im Übrigen können mit volatile gekennzeichnete Variablen auch als const deklariert werden, um sicherzustellen, dass sie nur noch von der Hardware änderbar sind.

Zugriff auf IO-Ports

Jeder AVR implementiert eine unterschiedliche Menge an GPIO-Registern (GPIO - General Purpose Input/Output). Diese Register dienen dazu:

- einzustellen welche der Anschlüsse ("Beinchen") des Controllers als Ein- oder Ausgänge dienen
- bei Ausgängen deren Zustand festzulegen
- bei Eingängen deren Zustand zu erfassen

Mittels GPIO werden digitale Zustände gesetzt und erfasst, d.h. die Spannung an einem Ausgang wird ein- oder ausgeschaltet und an einem Eingang wird erfasst, ob die anliegende Spannung über oder unter einem bestimmten Schwellwert liegt. Im Datenblatt Abschnitt Electrical Characteristics/DC Characteristics finden sich die Spannungswerte (V_OL, V_OH für Ausgänge, V_IL, V_IH für Eingänge).

Die Verarbeitung von analogen Eingangswerten und die Ausgabe von Analogwerten wird in Kapitel [Analoge Ein- und Ausgabe](#) behandelt.

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

Datenrichtungsregister für Portx.

DDRx x entspricht A, B, C, D usw. (abhängig von der Anzahl der Ports des verwendeten AVR). Bit im Register gesetzt (1) für Ausgang, Bit gelöscht (0) für Eingang.

Eingangsadresse für Portx.

PINx Zustand des Ports. Die Bits in PINx entsprechen dem Zustand der als Eingang definierten Portpins. Bit 1 wenn Pin "high", Bit 0 wenn Portpin low.

Datenregister für Portx.

PORTx Dieses Register wird verwendet, um die Ausgänge eines Ports anzusteuern. Bei Pins, die mittels DDRx auf Eingang geschaltet wurden, können über PORTx die internen Pull-Up Widerstände aktiviert oder deaktiviert werden (1 = aktiv).

Die folgenden Beispiele gehen von einem AVR aus, der sowohl Port A als auch Port B besitzt. Sie müssen für andere AVR (zum Beispiel ATmega8/48/88/168) entsprechend angepasst werden.

Datenrichtung bestimmen

Zuerst muss die Datenrichtung der verwendeten Pins bestimmt werden. Um dies zu erreichen, wird das Datenrichtungsregister des entsprechenden Ports beschrieben.

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.

Beispiel: Angenommen am Port B sollen die Pins 0 bis 4 als Ausgänge definiert werden, die noch verbleibenden Pins 5 bis 7 sollen als Eingänge fungieren. Dazu ist es daher notwendig, im für das Port B zuständigen Datenrichtungsregister DDRB folgende Bitkonfiguration einzutragen

```
+---+---+---+---+---+---+---+
| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
+---+---+---+---+---+---+---+
  7   6   5   4   3   2   1   0
```

In C liest sich das dann so:

```
// in io.h wird u.a. DDRB definiert:
#include <avr/io.h>

...

// Setzen der Bits 0,1,2,3 und 4
// Binär 00011111 = Hexadezimal 1F
DDRB = 0x1F; /* direkte Zuweisung - unübersichtlich */

// übersichtliche Alternative - Binärschreibweise
DDRB = 0b00011111; /* direkte Zuweisung - übersichtlich */
```

```

/* Ausführliche Schreibweise: identische Funktionalität, mehr Tipparbeit
aber übersichtlicher und selbsterklärend: */
DDRB = (1 << DDB0) | (1 << DDB1) | (1 << DDB2) | (1 << DDB3) | (1 << DDB4);

...

```

Die Pins 5 bis 7 werden (da 0) als Eingänge geschaltet. Weitere Beispiele:

```

// Alle Pins des Ports B als Ausgang definieren:
DDRB = 0xff;
// Pin0 wieder auf Eingang und andere im ursprünglichen Zustand belassen:
DDRB &= ~(1 << DDB0);
// Pin 3 und 4 auf Eingang und andere im ursprünglichen Zustand belassen:
DDRB &= ~( (1 << DDB3) | (1 << DDB4) );
// Pin 0 und 3 wieder auf Ausgang und andere im ursprünglichen Zustand belassen:
DDRB |= (1 << DDB0) | (1 << DDB3);
// Alle Pins auf Eingang:
DDRB = 0x00;

```

Vordefinierte Bitnummern für I/O-Register

Die Bitnummern (z.B. PCx, PINCx und DDCx für den Port C) sind in den io*.h-Dateien der avr-libc definiert und dienen lediglich der besseren Lesbarkeit. Man muss diese Definitionen nicht verwenden oder kann auch einfach "immer" PAX, PBx, PCx usw. nutzen, auch wenn der Zugriff auf Bits in DDRx- oder PINx-Registern erfolgt. Für den Compiler sind die Ausdrücke (1<<PC7), (1<<DDC7) und (1<<PINC7) identisch zu (1<<7) (genauer: der Präprozessor ersetzt die Ausdrücke (1<<PC7),... zu (1<<7)). Ein Ausschnitt der Definitionen für Port C eines ATmega32 aus der iom32.h-Datei zur Verdeutlichung (analog für die weiteren Ports):

```

...
/* PORTC */
#define PC7 7
#define PC6 6
#define PC5 5
#define PC4 4
#define PC3 3
#define PC2 2
#define PC1 1
#define PC0 0

/* DDRC */
#define DDC7 7
#define DDC6 6
#define DDC5 5
#define DDC4 4
#define DDC3 3
#define DDC2 2
#define DDC1 1
#define DDC0 0

/* PINC */
#define PINC7 7
#define PINC6 6
#define PINC5 5
#define PINC4 4
#define PINC3 3
#define PINC2 2
#define PINC1 1
#define PINC0 0

```

Digitale Signale

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

Ausgänge

Will man als Ausgang definierte Pins (entsprechende DDRx-Bits = 1) auf Logisch 1 setzen, setzt man die entsprechenden Bits im Portregister.

Mit dem Befehl

```

#include <avr/io.h>
...
PORTB = 0x04; /* besser PORTB=(1<<PB2) */

// übersichtliche Alternative – Binärschreibweise
PORTB = 0b00000100; /* direkte Zuweisung – übersichtlich */

```

wird also der Ausgang an Pin PB2 gesetzt (Beachte, dass die Bits immer *von 0 an* gezählt werden, das niederwertigste Bit ist also Bitnummer 0 und nicht etwa Bitnummer 1).

Man beachte, dass bei der Zuweisung mittels `=` immer alle Pins gleichzeitig angegeben werden. Man sollte also, wenn nur bestimmte Ausgänge geschaltet werden sollen, zuerst den aktuellen Wert des Ports einlesen und das Bit des gewünschten Ports in diesen Wert einfließen lassen. Will man also nur den dritten Pin (Bit Nr. 2) an Port B auf "high" setzen und den Status der anderen Ausgänge unverändert lassen, nutze man diese Form:

```
#include <avr/io.h>
...
PORTB = PORTB | 0x04; /* besser: PORTB = PORTB | ( 1<<PB2 ) */
/* vereinfacht durch Nutzung des |= Operators : */
PORTB |= (1<<PB2);

/* auch mehrere "gleichzeitig": */
PORTB |= (1<<PB4) | (1<<PB5); /* Pins PB4 und PB5 "high" */
```

"Ausschalten", also Ausgänge auf "low" setzen, erfolgt analog:

```
#include <avr/io.h>
...
PORTB &= ~(1<<PB2); /* löscht Bit 2 in PORTB und setzt damit Pin PB2 auf low */
PORTB &= ~( (1<<PB4) | (1<<PB5) ); /* Pin PB4 und Pin PB5 "low" */
```

In Quellcodes, die für ältere Versionen des `avr-gcc`/der `avr-libc` entwickelt wurden, werden einzelne Bits mittels der Funktionen `sbi` und `cbi` gesetzt bzw. gelöscht. Beide Funktionen sind in aktuellen Versionen der `avr-libc` nicht mehr enthalten und auch nicht mehr erforderlich.

Falls der Anfangszustand von Ausgängen kritisch ist, muss die Reihenfolge beachtet werden, mit der die Datenrichtung (`DDRx`) eingestellt und der Ausgabewert (`PORTx`) gesetzt wird:

Für Ausgangspins, die mit Anfangswert "high" initialisiert werden sollen:

- zuerst die Bits im `PORTx`-Register setzen
- anschließend die Datenrichtung auf Ausgang stellen

Daraus ergibt sich die Abfolge für einen Pin, der bisher als Eingang mit abgeschaltetem Pull-Up konfiguriert war:

- setze `PORTx`: interner Pull-Up aktiv
- setze `DDRx`: Ausgang ("high")

Bei der Reihenfolge erst `DDRx` und dann `PORTx`, kann es zu einem kurzen "low-Puls" kommen, der auch externe Pull-Up-Widerstände "überstimmt". Die (ungünstige) Abfolge: Eingang -> setze `DDRx`: Ausgang (auf "low", da `PORTx` nach Reset 0) -> setze `PORTx`: Ausgang auf high. Vergleiche dazu auch das Datenblatt Abschnitt *Configuring the Pin*.

Eingänge (Wie kommen Signale in den μ C)

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

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. Optokoppler, [Spannungsteiler](#), "Levelshifter" aka [Pegelwandler](#)) 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 unseres Controllers eine logische 1 (Spannung größer ca. $0,7 \cdot V_{cc}$) oder eine logische 0 (Spannung kleiner ca. $0,2 \cdot V_{cc}$) anliegt. Detaillierte Informationen darüber, ab welcher Spannung ein Eingang als 0 ("low") bzw. 1 ("high") erkannt wird, liefert die Tabelle DC Characteristics im Datenblatt des genutzten Controllers.

Die Abfrage der Zustände der Portpins erfolgt direkt über den Registernamen.

Dabei ist wichtig, zur Abfrage der Eingänge *nicht* etwa Portregister `PORTx` zu verwenden, sondern Eingangsregister `PINx`. Ansonsten liest man nicht den Zustand der Eingänge, sondern den Status der internen Pull-Up-Widerstände. Die Abfrage der Pinzustände über `PORTx` statt `PINx` ist ein häufiger Fehler beim AVR-"Erstkontakt".

Will man also die aktuellen Signalzustände von Port D abfragen und in eine Variable namens `bPortD` abspeichern, schreibt man folgende Befehlszeilen:

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

```

...
uint8_t bPortD;
...
bPortD = PIND;
...

```

Mit den C-Bitoperationen kann man den Status der Bits abfragen.

```

#include <avr/io.h>
...
/* Fuehre Aktion aus, wenn Bit Nr. 1 (das "zweite" Bit) in PINC gesetzt (1) ist */
if ( PINC & (1<<PINC1) ) {
  /* Aktion */
}

/* Fuehre Aktion aus, wenn Bit Nr. 2 (das "dritte" Bit) in PINB geloescht (0) ist */
if ( !(PINB & (1<<PINB2)) ) {
  /* Aktion */
}
...

```

Interne Pull-Up Widerstände

Portpins für Ein- und Ausgänge (GPIO) eines AVR verfügen über zuschaltbare interne Pull-Up Widerstände (nominal mehrere 10kOhm, z.B. ATmega16 20-50kOhm). Diese können in vielen Fällen statt externer Widerstände genutzt werden.

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.

Im Beispiel werden alle Pins des Ports D als Eingänge geschaltet und alle Pull-Up Widerstände aktiviert. Weiterhin wird Pin PC7 als Eingang geschaltet und dessen interner Pull-Up Widerstand aktiviert, ohne die Einstellungen für die anderen Portpins (PC0-PC6) zu verändern.

```

#include <avr/io.h>
...
DDRD = 0x00; /* alle Pins von Port D als Eingang */
PORTD = 0xff; /* interne Pull-Ups an allen Port-Pins aktivieren */

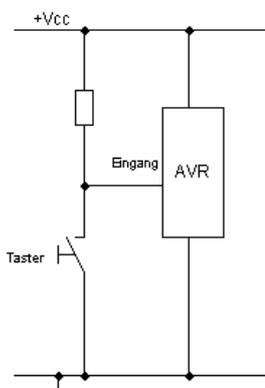
...
DDRC |= ~(1<<DDC7); /* Pin PC7 als Eingang */
PORTC |= (1<<PC7); /* internen Pull-Up an PC7 aktivieren */

```

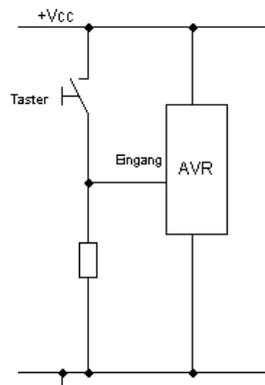
Tasten und Schalter

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

Anschluss mechanischer Kontakte an einen μC



Active Low: Bei dieser Methode wird der Kontakt zwischen den Eingangspin des Controllers und Masse geschaltet. Damit bei offenem Schalter der Controller kein undefiniertes Signal bekommt wird zwischen die Versorgungsspannung und den Eingangspin ein



Active High: Hier wird der Kontakt zwischen die Versorgungsspannung und den Eingangspin geschaltet. Damit bei offener Schalterstellung kein undefiniertes Signal am Controller ansteht, wird zwischen den Eingangspin und die Masse ein Pull Down Widerstand

sogenannter Pull-Up Widerstand geschaltet. Dieser dient dazu, den Pegel bei geöffnetem Schalter auf logisch 1 zu ziehen. Eingangspin und die Masse ein Pull-Down Widerstand geschaltet. Dieser dient dazu, den Pegel bei geöffneter Schalterstellung auf logisch 0 zu halten.

Der Widerstandswert von Pull-Up- und Pull-Down-Widerständen ist an sich nicht kritisch. Wird er allerdings zu hoch gewählt, ist die Wirkung eventuell nicht gegeben. Als üblicher Wert haben sich 10 kOhm eingebürgert. Die AVRs verfügen an den meisten Pins softwaremäßig über zuschaltbare interne Pull-Up Widerstände (vgl. Abschnitt [Interne Pull-Up Widerstände](#)), welche insbesondere wie hier bei Tastern und ähnlichen Bauteilen (z.B. Drehgebern) statt externer Bauteile verwenden werden können.

(Tasten-)Entprellung

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 Mikrocontroller 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.

Beim folgenden einfachen Beispiel für eine Entprellung ist zu beachten, dass der AVR im Falle eines Tastendrucks 200ms wartet, also brach liegt. Bei zeitkritische Anwendungen sollte man ein anderes Verfahren nutzen (z.B. Abfrage der Tastenzustände in einer Timer-Interrupt-Service-Routine).

```
#include <avr/io.h>
#include <inttypes.h>
#ifdef F_CPU
#warning "F_CPU war noch nicht definiert, wird nun mit 3686400 definiert"
#define F_CPU 3686400UL /* Quarz mit 3.6864 Mhz */
#endif
#include <util/delay.h> /* bei alter avr-libc: #include <avr/delay.h> */

/* Einfache Funktion zum Entprellen eines Tasters */
inline uint8_t debounce(volatile uint8_t *port, uint8_t pin)
{
    if ( ! (*port & (1 << pin)) )
    {
        /* Pin wurde auf Masse gezogen, 100ms warten */
        _delay_ms(50); // max. 262.1 ms / F_CPU in MHz
        _delay_ms(50);
        if ( *port & (1 << pin) )
        {
            /* Anwender Zeit zum Loslassen des Tasters geben */
            _delay_ms(50);
            _delay_ms(50);
            return 1;
        }
    }
    return 0;
}

int main(void)
{
    DDRB &= ~( 1 << PB0 ); /* PIN PB0 auf Eingang (Taster) */
    PORTB |= ( 1 << PB0 ); /* Pullup-Widerstand aktivieren */
    ...
    if (debounce(&PINB, PB0)) /* Falls Taster an PIN PB0 gedrueckt.. */
        PORTD = PIND ^ ( 1 << PD7 ); /* ..LED an Port PD7 an-
        bzw. ausschalten */
    ...
}
```

Zum Thema Entprellen siehe auch:

- Artikel [Entprellung](#)

Der UART

Allgemeines zum UART

Über den [UART](#) kann ein AVR leicht mit einer [RS-232](#)-Schnittstelle eines PC oder sonstiger Geräte mit "serieller Schnittstelle" verbunden werden.

Mögliche Anwendungen des UART:

Debug-Schnittstelle

z.B. zur Anzeige von Zwischenergebnissen ("printf-debugging" - hier besser "UART-debugging") auf einem PC. Auf dem Rechner reicht dazu ein [Terminalprogramm](#) (MS-Windows: Hyperterm oder besser [Bray-Terminal](#), [HTerm](#); Unix/Linux z.B. minicom). Ein direkter Anschluss ist aufgrund unterschiedlicher Pegel nicht möglich, jedoch sind entsprechende Schnittstellen-ICs wie z.B. ein MAX232 günstig und leicht zu integrieren. Rechner ohne serielle Schnittstelle können über fertige USB-seriell-Adapter angeschlossen werden.

Mensch-Maschine Schnittstelle

z.B. Konfiguration und Statusabfrage über eine "Kommandozeile" oder Menüs (siehe z.B. Forumsbeitrag [Auswertung RS232-Befehle](#) und Artikel

Tinykon

Übertragen von gespeicherten Werten

z.B. bei einem Datenlogger

Anschluss von Geräten

mit serieller Schnittstelle (z.B. (Funk-)Modems, Mobiltelefone, Drucker, Sensoren, "intelligente" LC-Displays, GPS-Empfänger).

"Feldbusse"

auf RS485/RS422-Basis mittels entsprechenden Bustreiberbausteinen (z.B. MAX485)

DMX, Midi

etc.

LIN-Bus

Local Interconnect Network: Preiswerte Sensoren/Aktoren in der Automobiltechnik und darüber hinaus

Einige AVR-Controller haben ein bis zwei vollduplexfähigen UART (**U**niversal **A**synchronous **R**eceiver and **T**ransmitter) schon eingebaut ("Hardware-UART"). Übrigens: Vollduplex heißt nichts anderes, als dass der Baustein gleichzeitig senden und empfangen kann.

Neuere AVR's (ATmega, ATtiny) verfügen über einen oder zwei USART(s), dieser unterscheidet sich vom UART hauptsächlich durch interne FIFO-Puffer für Ein- und Ausgabe und erweiterte Konfigurationsmöglichkeiten. Die Puffergröße ist allerdings nur 1 Byte.

Der UART wird über vier separate Register angesprochen. USARTs der ATMEGAs verfügen über mehrere zusätzliche Konfigurationsregister. Das Datenblatt gibt darüber Auskunft. Die Folgende Tabelle gibt nur die Register für die (veralteten) UARTs wieder.

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.

UCR**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.

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 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.

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.

USR

Das Bit wird nur dann automatisch gelöscht, wenn der entsprechende Interrupthandler aufgerufen wird, ansonsten müssen wir das Bit selber löschen.

UDRE (UART Data Register Empty)

Dieses Bit zeigt an, ob der Sendepuffer bereit ist, um ein zu sendendes Zeichen aufzunehmen. Das Bit wird vom AVR gesetzt (1), wenn der Sendepuffer leer ist. Es wird gelöscht (0), wenn ein Zeichen im Sendedatenregister vorhanden ist und noch nicht in das Sendedatenregister übernommen wurde. Atmel empfiehlt aus Kompatibilitätsgründen mit kommenden μC , UDRE auf 0 zu setzen, wenn das UCSRA Register beschrieben wird.

Das Bit wird automatisch gelöscht, wenn ein Zeichen in das Sendedatenregister geschrieben wird.

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.

DOR (DataOverRun)

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.

UART Data Register.

UDR

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.

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:

UBRR

$$UBRR = \frac{\text{Taktfrequenz}}{\text{Baudrate} \cdot 16} - 1$$

Es sind Baudraten bis zu 115200 Baud und höher möglich.

Die Hardware

Der UART basiert auf normalem TTL-Pegel mit 0V (logisch 0) und 5V (logisch 1). Die Schnittstellenspezifikation für RS-232 definiert jedoch -3V ... -12V (logisch 1) und +3 ... +12V (logisch 0). Daher 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.

Streikt die Kommunikation per UART, so ist oft eine fehlerhafte Einstellung der Baudrate die Ursache. Die Konfiguration auf eine bestimmte Baudrate ist abhängig von der Taktfrequenz des Controllers. Gerade bei neu aufgebauten Schaltungen (bzw. neu gekauften Controllern) sollte man sich daher noch einmal vergewissern, dass der Controller auch tatsächlich mit der vermuteten Taktrate arbeitet und nicht z.B. den bei einigen Modellen werksseitig eingestellten internen [Oszillator](#) statt eines externen Quarzes nutzt. Die Werte der verschiedenen fuse-bits im Fehlerfall also beispielsweise mit [AVRDUDE](#) kontrollieren und falls nötig anpassen. Grundsätzlich empfiehlt sich auch immer ein Blick in die [AVR Checkliste](#).

UART initialisieren

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:

```
UCR |= (1<<TXEN);
```

Neuere AVR's mit USART haben mehrere Konfigurationsregister und erfordern eine etwas andere Konfiguration. Für einen ATmega16 z.B.:

```
UCSRB |= (1<<TXEN); // UART TX einschalten
UCSRC |= (1<<URSEL)|(1 << UCSZ1)|(1 << UCSZ0); // Asynchron 8N1
```

Nun ist noch das Baudratenregister **UBRR** einzustellen. Bei neueren AVR's besteht es aus zwei Registern **UBRRL** und **UBRRH**. Der Wert dafür ergibt sich aus der angegebenen Formel durch Einsetzen der Taktfrequenz und der gewünschten Übertragungsrate. Das Berechnen der Formel wird dem [Präprozessor](#) überlassen.

```
/* UART-Init beim AT90S2313 */
#ifdef F_CPU
/* In neueren Version der WinAVR/Mfile Makefile-Vorlage kann
F_CPU im Makefile definiert werden, eine nochmalige Definition
hier wuerde zu einer Compilerwarnung fuehren. Daher "Schutz" durch
#ifdef/#endif

Dieser "Schutz" kann zu Debugsessions fuehren, wenn AVRStudio
verwendet wird und dort eine andere, nicht zur Hardware passende
Taktrate eingestellt ist: Dann wird die folgende Definition
nicht verwendet, sondern stattdessen der Defaultwert (8 MHz?)
von AVRStudio – daher Ausgabe einer Warnung falls F_CPU
noch nicht definiert: */
#warning "F_CPU war noch nicht definiert, wird nun nachgeholt mit 4000000"
#define F_CPU 4000000UL // Systemtakt in Hz – Definition als unsigned long beachten >> Ohne ergeben Fehler in der Berechnung
#endif

#define BAUD 9600UL // Baudrate

// Berechnungen
#define UBRR_VAL ((F_CPU+BAUD*8)/(BAUD*16)-1) // clever runden
#define BAUD_REAL (F_CPU/(16*(UBRR_VAL+1))) // Reale Baudrate
#define BAUD_ERROR ((BAUD_REAL*1000)/BAUD) // Fehler in Promille, 1000 = kein Fehler.

#if ((BAUD_ERROR<990) || (BAUD_ERROR>1010))
#error Systematischer Fehler der Baudrate grösser 1% und damit zu hoch!
#endif

int main(void)
{
    UCR |= (1<<TXEN);
    UBRR = UBRR_VAL;
    //...
}
```

Wieder für den Mega16 mit zwei Registern für die Baudrateneinstellung eine etwas andere Programmierung. Wichtig ist, dass **UBRRH vor UBRRL** geschrieben wird.

```
/* USART-Init beim ATmega16 */
int main(void)
{
    UCSRB |= (1<<TXEN); // UART TX einschalten
    UCSRC |= (1<<URSEL)|(3<<UCSZ0); // Asynchron 8N1

    UBRRH = UBRR_VAL >> 8;
    UBRRL = UBRR_VAL & 0xFF;
}
```

Für einige AVR (z.B. ATmega169, ATmega48/88/168, AT90CAN jedoch nicht für z.B. ATmega16/32, ATmega128, ATtiny2313) wird durch die Registerdefinitionen der avr-libc (io*.h) auch für Controller mit zwei UBRR-Registern (UBRRL/UBRRH) ein UBRR bzw. UBRR0 als "16-bit-Register" definiert und man kann auch Werte direkt per `UBRR = UBRR_VAL` zuweisen. Intern werden dann zwei Zuweisungen für UBRRH und UBRRL generiert. Dies ist nicht bei allen Controllern möglich, da die beiden Register nicht bei allen aufeinanderfolgende Adressen aufweisen. Die getrennte Zuweisung an UBRRH und UBRRL wie im Beispiel gezeigt ist universeller und portabler und daher vorzuziehen.

Die Makros sind sehr praktisch, da sie sowohl automatisch den Wert für UBRR als auch den Fehler in der generierten Baudrate berechnen und im Falle einer Überschreitung (+/-1%) einen Fehler und somit Abbruch im Compilerablauf generieren. Damit können viele Probleme mit "UART sendet komische Zeichen" vermieden werden. Ausserdem kann man mühelos die Einstellung an eine neue Taktfrequenz bzw. Baudrate anpassen, ohne selber rechnen oder in Tabellen nachschlagen zu müssen.

Inzwischen gibt es in der avr-libc Makros für obige Berechnung der UBRR Registerwerte aus Taktrate F_CPU und Baudrate BAUD in der Includedatei <util/setbaud.h> ([\(11\)](#)).

Siehe auch:

- [WormFood's AVR Baud Rate Calculator](#) online.
- [AVR Baudraten-Rechner](#) in JavaScript

Senden mit dem UART

Senden einzelner Zeichen

Um nun ein Zeichen auf die Schnittstelle auszugeben, müssen wir dasselbe lediglich in das UART Data Register schreiben. Vorher ist zu prüfen, ob das UART-Modul bereit ist, das zu sendende Zeichen entgegenzunehmen. Die Bezeichnungen des/der Statusregisters mit dem Bit UDRE ist abhängig vom Controllertypen (vgl. Datenblatt).

```
// bei AVR mit einem UART ("classic AVR" z.B. AT90S8515)
while (!(USR & (1<<UDRE))) /* warten bis Senden moeglich */
{
}

UDR = 'x'; /* schreibt das Zeichen x auf die Schnittstelle */

/** ODER */

// bei neueren AVR's steht der Status in UCSRA/UCSR0A/UCSR1A, hier z.B. fuer ATmega16:
while (!(UCSRA & (1<<UDRE))) /* warten bis Senden moeglich */
{
}

UDR = 'x'; /* schreibt das Zeichen x auf die Schnittstelle */
```

Schreiben einer Zeichenkette (String)

Die Aufgabe "String senden" wird durch zwei Funktionen abgearbeitet. Die universelle/controllerunabhängige Funktion `uart_puts` übergibt jeweils ein Zeichen der Zeichenkette an eine Funktion `uart_putc`, die abhängig von der vorhandenen Hardware implementiert werden muss. In der Funktion zum Senden eines Zeichens ist darauf zu achten, dass vor dem Senden geprüft wird, ob der UART bereit ist den "Sendeauftrag" entgegenzunehmen.

```
// putc fuer AVR mit einem UART (z.B. AT90S8515)
int uart_putc(unsigned char c)
{
    while(!(USR & (1 << UDRE))) /* warte, bis UDR bereit */
    {
    }

    UDR = c; /* sende Zeichen */
    return 0;
}

/** ODER */

// bei neueren AVR's andere Bezeichnung fuer die Statusregister, hier ATmega16:
int uart_putc(unsigned char c)
{
    while (!(UCSRA & (1<<UDRE))) /* warten bis Senden moeglich */
    {
    }

    UDR = c; /* sende Zeichen */
    return 0;
}

/* puts ist unabhængig vom Controllertyp */
void uart_puts (char *s)
{
    while (*s)
    { /* so lange *s != '\0' also ungleich dem "String-Endezeichen" */
        uart_putc(*s);
        s++;
    }
}
```

Die in `uart_putc` verwendeten Schleifen, in denen gewartet wird bis die UART-Hardware zum senden bereit ist, sind insofern etwas kritisch, da während des Sendens eines Strings nicht mehr auf andere Ereignisse reagieren werden kann. Universeller ist die Nutzung von FIFO(first-in first-out)-Puffern, in denen die zu sendenden bzw. empfangenen Zeichen/Bytes zwischengespeichert und in Interruptroutinen an die U(S)ART-Hardware weitergegeben bzw. von ihr ausgelesen werden. Dazu existieren fertige Komponenten (Bibliotheken, Libraries), die man recht einfach in eigene Entwicklungen integrieren kann. Es empfiehlt sich, diese Komponenten zu nutzen und das Rad nicht neu zu erfinden.

Schreiben von Variableninhalten

Sollen Inhalte von Variablen (Ganzzahlen, Fließkomma) in "menschlesbarer" Form gesendet werden, ist vor dem Transfer eine Umwandlung in Zeichen ("ASCII") erforderlich. Bei nur einer Ziffer ist diese Umwandlung relativ einfach: man addiert den ASCII-Wert von Null zur Ziffer und kann diesen Wert direkt senden.

```
//...
// hier uart_putc (s.o.)
// Ausgabe von 0123456789
char c;
for (uint8_t i=0; i<=9; ++i) {
    c = i + '0';
    uart_putc( c );
    // verkuerzt: uart_putc( i + '0' );
}
```

Soll mehr als eine Ziffer ausgegeben werden, bedient man sich zweckmäßigerweise vorhandener Funktionen zur Umwandlung von Zahlen in Zeichenketten/Strings. Die Funktion der avr-libc zur Umwandlung von vorzeichenbehafteten 16bit-Ganzzahlen (int16_t) in Zeichenketten heißt *itoa* (Integer to ASCII). Man muss der Funktion einen Speicherbereich zur Verarbeitung (buffer) mit Platz für alle Ziffern, das String-Endezeichen ('\0') und evtl. das Vorzeichen bereitstellen.

```
#include <stdlib.h>
//...
// hier uart_init, uart_putc, uart_puts (s.o.)
int main(void)
{
    char s[7];
    int16_t i = -12345;

    uart_init();

    itoa( i, s, 10 ); // 10 fuer radix -> Dezimalsystem
    uart_puts( s );

    // da itoa einen Zeiger auf den Beginn von s zurueckgibt verkuerzt auch:
    uart_puts( itoa( i, s, 10 ) );

    while (1) {
        ;
    }

    return 0; // never reached
}
```

Für vorzeichenlose 16bit-Ganzzahlen (uint16_t) existiert *utoa*. Die Funktionen für 32bit-Ganzzahlen (int32_t und uint32_t) heißen *ltoa* bzw. *ultoa*. Da 32bit-Ganzzahlen mehr Stellen aufweisen können, ist ein entsprechend größerer Pufferspeicher vorzusehen.

Auch Fließkommazahlen (float/double) können mit bereits vorhandenen Funktionen in Zeichenfolgen umgewandelt werden, dazu existieren die Funktionen *dtostrf* und *dtostrf*. *dtostrf* nutzt Exponential Schreibweise ("engineering"-Format). (Hinweis: z.Zt. existiert im avr-gcc kein "echtes" double, intern wird immer mit "einfacher Genauigkeit", entsprechend float, gerechnet.)

dtostrf und *dtostrf* benötigen die libm.a der avr-libc. Bei Nutzung von Makefiles ist der Parameter -lm in in LDFLAGS anzugeben (Standard in den WinAVR/mfile-Makefilevorlagen). Nutzt man AVRStudio als IDE für den GNU-Compiler (gcc-Plugin) ist die libm.a unter Libraries auszuwählen: Project -> Configurations Options -> Libraries -> libm.a mit dem Pfeil nach rechts einbinden. Siehe auch die [FAQ](#)

```
#include <stdlib.h>
//...
// hier uart_init, uart_putc, uart_puts (s.o.)
/* lt. avr-libc Dokumentation:
char* dtostrf(
    double __val,
    char __width,
    char __prec,
    char * __s
)
*/
int main(void)
{
    // Pufferspeicher ausreichend groß
    // evtl. Vorzeichen + width + Endezeichen:
    char s[8];
```

```

float f = -12.345;

uart_init();

dtostrf( f, 6, 3, s );
uart_puts( s );

// verkürzt: uart_puts( dtostrf( f, 7, 3, s ) );

while (1) {
    ;
}

return 0; // never reached
}

```

Empfangen

einzelne Zeichen empfangen

Zum Empfang von Zeichen muss der Empfangsteil des UART bei der Initialisierung aktiviert werden, indem das RXEN-Bit im jeweiligen Konfigurationsregister (UCSRB bzw UCSROB/UCSR1B) gesetzt wird. Im einfachsten Fall wird solange gewartet, bis ein Zeichen empfangen wurde, dieses steht dann im UART-Datenregister (UDR bzw. UDR0 und UDR1 bei AVR mit 2 UARTS) zur Verfügung (sogen. "Polling-Betrieb"). Ein Beispiel für den ATmega16:

```

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

/* Zusätzlich zur Baudrateneinstellung und der weiteren Initialisierung: */
void Usart_EnableRX(void)
{
    UCSRB |= ( 1 << RXEN );
}

/* Zeichen empfangen */
uint8_t uart_getc(void)
{
    while (!(UCSRA & (1<<RXC))) // warten bis Zeichen verfuegbar
        ;
    return UDR; // Zeichen aus UDR an Aufrufer zurueckgeben
}

```

Diese Funktion blockiert den Programmablauf. Alternativ kann das RXC-Bit in einer Programmschleife abgefragt werden und dann nur bei gesetztem RXC-Bit UDR ausgelesen werden. Eleganter und in den meisten Anwendungsfällen "stabiler" ist die Vorgehensweise, die empfangenen Zeichen in einer Interrupt-Routine einzulesen und zur späteren Verarbeitung in einem Eingangsbuffer (FIFO-Buffer) zwischenspeichern. Dazu existieren fertige und gut getestete [Bibliotheken](#) und Quellcodekomponenten (z.B. UART-Library von P. Fleury, procyon-avr-lib und einige in der "Academy" von avrfreaks.net).

siehe auch:

- [Dokumentation der avr-libc/stdlib.h](#)
- [Die Nutzung von printf](#)
- [Peter Fleurys](#) UART-Bibliothek fuer avr-gcc/avr-libc

TODO: 9bit

Empfang von Zeichenketten (Strings)

Beim Empfang von Zeichenketten, muß man sich zunächst darüber im klaren sein, daß es ein Kriterium geben muß, an dem der μC erkennen kann, wann ein String zu Ende ist. Sehr oft wird dazu das Zeichen 'Return' benutzt, um das Ende eines Strings zu markieren. Dies ist vom Benutzer einfach eingebbar und er ist auch daran gewöhnt, daß er eine Eingabezeile mit einem Druck auf die Return Taste abgeschlossen wird.

Prinzipiell gibt es jedoch keine Einschränkung bezüglich dieses speziellen Zeichens. Es muß nur sichergestellt werden, daß dieses spezielle 'Ende eines Strings' - Zeichen nicht mit einem im Text vorkommenden Zeichen verwechselt werden kann. Wenn also im zu übertragenden Text beispielsweise kein ';' vorkommt, dann spricht nichts dagegen, einen String mit einem ';' abschließen zu lassen.

Im Folgenden wird die durchaus übliche Annahme getroffen, daß eine Stringübertragung identisch ist mit der Übertragung einer Textzeile und daher mit einem Return ('\n') abgeschlossen wird.

Das Problem der Übertragung eines Strings reduziert sich damit auf die Aufgabenstellung: Empfange und Sammele Zeichen in einem char Array, bis entweder das Array voll ist oder das 'String Ende Zeichen' empfangen wurde. Danach wird der empfangene Text noch mit einem '\0' Zeichen abgeschlossen um einen Standard C-String daraus zu machen, mit dem dann weiter gearbeitet werden kann.

```

/* Zeichen empfangen */
uint8_t uart_getc(void)
{
    while (!(UCSRA & (1<<RXC))) // warten bis Zeichen verfuegbar
        ;
}

```

```

    return UDR;                // Zeichen aus UDR an Aufrufer zurueckgeben
}

void uart_gets( char* Buffer, uint8_t MaxLen )
{
    uint8_t NextChar;
    uint8_t StringLen = 0;

    NextChar = uart_getc();    // Warte auf und empfang das nächste Zeichen

                                // Sammle solange Zeichen, bis:
                                // * entweder das String Ende Zeichen kam
                                // * oder das aufnehmende Array voll ist
    while( NextChar != '\n' && StringLen < MaxLen - 1 ) {
        *Buffer++ = NextChar;
        StringLen++;
        NextChar = uart_getc();
    }

                                // Noch ein '\0' anhängen um einen Standard
                                // C-String daraus zu machen

    *Buffer = '\0';
}

```

Beim Aufruf ist darauf zu achten, dass das empfangende Array auch mit einer vernünftigen Größe definiert wird.

```

char Line[40];    // String mit maximal 39 zeichen

uart_gets( Line, sizeof( Line ) );

```

Bei der Benutzung von sizeof() ist allerdings zu beachten, dass sizeof() nicht die Anzahl der Elemente des Arrays liefert, sondern die Länge in Byte. Da ein char nur ein Byte lang ist, passt der Aufruf 'uart_gets(Line, sizeof(Line));' in diesem Fall. Falls man - aus welchen Gründen auch immer - andere Datentypen benutzen möchte, sollte man zur korrekten Angabe der Array-Länge folgende Vorgehensweise bevorzugen:

```

int Line[40];    // Array vom Typ int

uart_gets( Line, sizeof( Line ) / sizeof( Line[0] ) );

```

Interruptbetrieb

Beim ATMEGA8 muss das RXCIE Bit im Register UCSRB gesetzt werden, damit ein Interrupt ausgelöst werden kann. Der Interrupt wird immer ausgelöst, wenn Daten erfolgreich empfangen wurden.

Zusätzlich braucht man die Routine:

```

ISR (USART_RXC_vect) {
    //irgendein Code
}

```

natürlich muss "Global Interrupt Enable" Aktiviert sein. !! Nur getestet beim ATMEGA8 !!

[BAUSTELLE! Aus Lerngründen eventuell als eigenen UART-Interrupt-Block hinter den grundlegenden Interrupt-Teil im Tutorial und hier eine kurze Einführung und einen Verweis darauf anbieten.]

- Unterschied Polling-Betrieb (bisher, oben) und Interrupt-Betrieb
- Empfangen (Receive)
 - Verändertes UART-Init, ISR (RXC), ggf. Fallstricke ([UDR in der ISR lesen!](#)), Philosophie einer ISR (kurz und schmerzlos), Datenaustausch ISR zu Restprogramm (volatile)
 - Einfachstbeispiel ([Echo](#) (noch buggy beim Datenzugriff, siehe Lit. 2+3!)), ggf. LED zur ISR-Empfangsanzeige oder Overflow-Anzeige
- FIFO-Puffer, Ringpuffer
- Senden (Transmit)
 - Variante "UART Data Register Empty" (UDRE)
 - Variante "UART Transmit Complete" (TXC)
- Fertige UART-Bibliotheken (->Fleury, ->Procyon)
- Literatur:
 - 1/ [avrfreaks.net Tutorial](#) inkl. Diskussion (engl.)
 - 2/ avr-libc FAQ: [Why do some 16-bit timer registers sometimes get trashed?](#)
 - 3/ [Interrupt](#) und atomarer Datenzugriff

Software-UART

Falls die Zahl der vorhandenen Hardware-UARTs nicht ausreicht, können weitere Schnittstellen über sogenannte Software-UARTs ergänzt werden. Es gibt dazu (mindestens) zwei Ansätze:

- Der bei AVR's üblichste Ansatz basiert auf dem Prinzip, dass ein externer Interrupt-Pin für den Empfang ("RX") genutzt wird. Das Startbit löst den Interrupt aus, in der Interrupt-Routine (ISR) wird der externe Interrupt deaktiviert und ein Timer aktiviert. In der Interrupt-Routine des Timers wird der Zustand des Empfangs-Pins entsprechend der Baudrate abgetastet. Nach Empfang des Stop-Bits wird der externe Interrupt wieder aktiviert. Senden kann über einen beliebigen Pin ("TX") erfolgen, der entsprechend der Baudrate und dem zu sendenden Zeichen auf 0 oder 1 gesetzt wird. Die Implementierung ist nicht ganz einfach, es existieren dazu aber fertige Bibliotheken (z.B. bei [avrfreaks](#) oder in der [Procyon avrlib](#)).
- Ein weiterer Ansatz erfordert keinen Pin mit "Interrupt-Funktion" aber benötigt mehr Rechenzeit. Jeder Input-Pin kann als Empfangspin (RX) dienen. Über einen Timer wird der Zustand des RX-Pins mit einem vielfachen der Baudrate abgetastet (dreifach scheint üblich) und High- bzw. Lowbits anhand einer Mindestanzahl identifiziert. (Beispiel: "Generic Software Uart" Application-Note von IAR)

Neuere AVR's (z.B. ATtiny26 oder ATmega48,88,168,169) verfügen über ein Universal Serial Interface (USI), das teilweise UART-Funktion übernehmen kann. Atmel stellt eine Application-Note bereit, in der die Nutzung des USI als UART erläutert wird (im Prinzip "Hardware-unterstützter Software-UART").

Handshaking

Wenn der Sender ständig sendet, wird irgendwann der Fall eintreten, daß der Empfänger nicht bereit ist, neue Zeichen zu empfangen. In diesem Fall muß durch ein **Handshake-Verfahren** die Situation bereinigt werden. Handshake bedeutet nichts anderes, als daß der Empfänger dem Sender mitteilt, daß er zur Zeit keine Daten annehmen kann und der Sender die Übertragung der nächsten Zeichen solange einstellen soll, bis der Empfänger signalisiert, daß er wieder Zeichen aufnehmen kann.

Hardwarehandshake (RTS/CTS)

Beim Hardwarehandshake werden zusätzlich zu den beiden Daten-Übertragungsleitungen noch 2 weitere Leitungen benötigt: **RTS (Request To Send)** und **CTS (Clear To Send)**. Jeder der beiden Kommunikationspartner ist verpflichtet, bevor ein Zeichen gesendet wird, den Zustand der **RTS** Leitung zu überprüfen. Nur wenn die Gegenstelle darauf Empfangsbereitschaft signalisiert, darf das Zeichen gesendet werden. Um der Gegenstelle zu signalisieren, daß sie zur Zeit keine Zeichen schicken soll, wird die Leitung **CTS** benutzt.

Softwarehandshake (XON/XOFF)

Beim Softwarehandshake sind keine speziellen Leitungen notwendig. Statt dessen werden besondere ASCII-Zeichen benutzt, die der Gegenstelle signalisieren, daß Senden einzustellen bzw. wieder aufzunehmen.

- **XOFF** Aufforderung das Senden einzustellen
- **XON** Gegenstelle darf wieder senden

Nachteilig bei einem Softwarehandshake ist es, dass dadurch keine direkte binäre Datenübertragung mehr möglich ist. Von den möglichen 256 Bytewerten werden ja 2 (nämlich **XON** und **XOFF**) für besondere Zwecke benutzt und fallen daher aus.

Fehlersuche

Erstaunlich oft wird im Forum der Hilferuf laut: "Meine UART funktioniert nicht, was mache ich falsch". In der überwiegenden Mehrzahl der Fälle stellt sich dann heraus, daß es sich um ein Hardwareproblem handelt, wobei da wiederum der Löwenanteil auf das Konto einer nicht korrekt eingestellten Taktrate geht: Der μ C benutzt nicht einen angeschlossenen Quarz, so wie er auch im Programm eingetragen ist, sondern läuft immer noch mit dem internen RC-Takt. Daraus resultiert aber auch, daß der Baudraten Konfigurationswert falsch berechnet wird.

Eine Checkliste zum Aufspüren solcher Fehler findet sich [hier](#).

Links

FAQ zur Verarbeitung von Strings: <http://www.mikrocontroller.net/articles/FAQ>

Analoge Ein- und Ausgabe

Analoge Eingangswerte werden in der Regel über den AVR Analog-Digital-Converter (AD-Wandler, ADC) eingelesen, der in vielen Typen verfügbar ist (typisch 10bit Auflösung). Durch diesen werden analoge Signale (Spannungen) in digitale Zahlenwerte gewandelt. Bei AVR's, die über keinen internen AD-Wandler verfügen (z.B. ATmega162), kann durch externe Beschaltung (R/C-Netzwerk und "Zeitmessung") die Funktion des AD-Wandlers "emuliert" werden.

Es existieren keine AVR's mit eingebautem Digital-Analog-Konverter (DAC). Diese Funktion muss durch externe Komponenten nachgebildet werden (z.B. PWM und "Glättung").

Unabhängig davon besteht natürlich immer die Möglichkeit, spezielle Bausteine zur Analog-Digital- bzw. Digital-Analog-Wandlung zu nutzen und diese über eine digitale Schnittstelle (z.b. SPI oder I2C) mit einem AVR anzusteuern.

AC (Analog Comparator)

Der Comparator vergleicht 2 Spannungen an den Pins AIN0 und AIN1 und gibt einen Status aus welche der beiden Spannungen größer ist. AIN0 Dient dabei als Referenzspannung (Sollwert) und AIN1 als Vergleichsspannung (Istwert). Als Referenzspannung kann auch alternativ eine interne Referenzspannung ausgewählt werden.

Liegt die Vergleichsspannung (IST) unter der der Referenzspannung (SOLL) gibt der Comperator eine logische 1 aus. Ist die Vergleichsspannung hingegen größer als die Referenzspannung wird eine logische 0 ausgegeben.

Der Comparator arbeitet völlig autark bzw. parallel zum Prozessor. Für mobile Anwendungen empfiehlt es sich ihn abzuschalten sofern er nicht benötigt wird, da er ansonsten Strom benötigt. Der Comparator kann Interruptgesteuert abgefragt werden oder im Pollingbetrieb.

Das Steuer- bzw. Statusregister ist wie folgt aufgebaut:

ACSR - Analog Comparator Status Register								
Bit	7	6	5	4	3	2	1	0
Name	ACD	ACBG	ACO	ACI	ACIE	ACIC	ACIS1	ACIS0
R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
Initialwert	0	0	n/a	0	0	0	0	0

Bit 7 ACD

Analog Comparator Disable: 0 = Comparator ein, 1 = Comparator aus. Wird dieses Bit geändert kann ein Interrupt ausgelöst werden. Soll dies vermieden werden muß das Bit 3 ACIE ggf. abgeschaltet werden.

Bit 6 ACBG

Analog Comparator Bandgap Select: Ermöglicht das Umschalten zwischen interner und externer Referenzspannung. 1 = interne (~1,3 Volt), 0 = externe Referenzspannung (an Pin AIN0)

Bit 5 ACO

Analog Comparator Output: Hier wird das Ergebnis des Vergleichs angezeigt. Es liegt typischerweise nach 1-2 Taktzyklen vor.

IST < SOLL → 1
IST > SOLL → 0

Bit 4 ACI

Analog Comparator Interrupt Flag: Dieses Bit wird von der Hardware gesetzt wenn ein Interruptereignis das in Bit 0 und 1 definiert ist eintritt. Dieses Bit löst noch keinen Interrupt aus! Die Interruptroutine wird nur dann ausgeführt wenn das Bit 3 ACIE gesetzt ist und global Interrupts erlaubt sind (I-Bit in SREG=1). Das Bit 4 ACI wird wieder gelöscht wenn die Interruptroutine ausgeführt wurde oder wenn manuell das Bit auf 1 gesetzt wird. Das Bit kann für Abfragen genutzt werden, steuert oder konfiguriert aber nicht den Comparator.

Bit 3 ACIE

Analog Comparator Interrupt Enable: Ist das Bit auf 1 gesetzt wird immer ein Interrupt ausgelöst wenn das Ereignis das in Bit 1 und 0 definiert ist eintritt.

Bit 2 ACIC

Analog Comparator Input Capture Enable: Wird das Bit gesetzt wird der Comparatorausgang intern mit dem Counter 1 verbunden. Es könnten damit z.B. die Anzahl der Vergleiche im Counter1 gezählt werden. Um den Comparator an den Timer1 Input Capture Interrupt zu verbinden muß im Timerregister das TICIE1 Bit auf 1 gesetzt werden. Der Trigger wird immer dann ausgelöst wenn das in Bit 1 und 0 definierte Ereignis eintritt.

Bit 1,0 ACIS1,ACIS0

Analog Comparator Interrupt select: Hier wird definiert welche Ereignisse einen Interrupt auslösen sollen:

- 00 = Interrupt auslösen bei jedem Flankenwechsel
- 10 = Interrupt auslösen bei fallender Flanke
- 11 = Interrupt auslösen bei steigender Flanke

Werden diese Bit geändert kann ein Interrupt ausgelöst werden. Soll dies vermieden werden muß das Bit 3 gelöscht werden.

ADC (Analog Digital Converter)

Der Analog-Digital-Konverter (ADC) wandelt analoge Signale in digitale Werte um, welche vom Controller interpretiert werden können. Einige AVR-Typen haben bereits einen mehrkanaligen Analog-Digital-Konverter eingebaut. 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. ADCs die in AVR's enthalten sind haben zur Zeit eine maximale Auflösung von 10-Bit.

Ein ADC mit 8 Bit Auflösung kann somit das analoge Signal mit einer Genauigkeit von 1/256 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 [V]	Entsprechender Messwert
0–0.625	0
0.625–1.25	1
1.25–1.875	2
1.875–2.5	3
2.5–3.125	4
3.125–3.75	5
3.75–4.375	6
4.375–5	7

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

Der interne ADC im AVR

Wenn es einmal etwas genauer sein soll, dann müssen wir auf einen AVR mit eingebautem Analog-Digital-Wandler (ADC) zurückgreifen, die über mehrere Kanäle verfügen. Kanäle heißt in diesem Zusammenhang, dass zwar bis zu zehn analoge Eingänge am AVR verfügbar sind, aber nur ein "echter" Analog-Digital-Wandler zur Verfügung steht, vor der eigentlichen Messung ist also einzustellen, welcher Kanal ("Pin") mit dem Wandler verbunden und gemessen wird.

Die Umwandlung innerhalb des AVR basiert auf der schrittweisen Näherung. Beim AVR müssen die Pins **AGND** und **AVCC** beschaltet werden. Für genaue Messungen sollte **AVCC** über ein L-C Netzwerk mit **VCC** verbunden werden, um Spannungsspitzen und -einbrüche vom Analog-Digital-Wandler fernzuhalten. Im Datenblatt findet sich dazu eine Schaltung, die 10uH und 100nF vorsieht.

Das Ergebnis der Analog-Digital-Wandlung wird auf eine Referenzspannung bezogen. Aktuelle AVRs bieten 3 Möglichkeiten zur Wahl dieser Spannung:

- Eine externe Referenzspannung von maximal **Vcc** am Anschlusspin **AREF**. Die minimale (externe) Referenzspannung darf jedoch nicht beliebig niedrig sein, vgl. dazu das (aktuellste) Datenblatt des verwendeten Controllers.
- Verfügt der AVR über eine interne Referenzspannung, kann diese genutzt werden. Alle aktuellen AVRs mit internem AD-Wandler sollten damit ausgestattet sein (vgl. Datenblatt: 2,56V oder 1,1V je nach Typ). Das Datenblatt gibt auch über die Genauigkeit dieser Spannung Auskunft.
- Es kann die Spannung **AVcc** als Referenzspannung herangezogen werden

Bei Nutzung von **AVcc** oder der internen Referenz wird empfohlen, einen Kondensator zwischen dem **AREF**-Pin und **GND** anzuordnen. Die Festlegung, welche Spannungsreferenz genutzt wird, erfolgt z.B. beim ATmega16 mit den Bits **REFS1/REFS0** im **ADMUX**-Register. Die zu messende Spannung muss im Bereich zwischen **AGND** und **AREF** (egal ob intern oder extern) liegen.

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

Einfache Wandlung (Single Conversion)

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

Frei laufend (Free Running)

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

Die Register des ADC

Der **ADC** verfügt über eigene Register. Im Folgenden die Registerbeschreibung eines ATmega16, welcher über 8 ADC-Kanäle verfügt. Die Register unterscheiden sich jedoch nicht erheblich von denen anderer AVRs (vgl. Datenblatt).

ADC Control and Status Register A.

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						
Initialwert	0	0	0	0	0	0	0	0

ADEN (ADC Enable)

Dieses Bit muss gesetzt werden, um den **ADC** überhaupt zu aktivieren. Wenn das Bit nicht gesetzt ist, können die Pins 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 Run select)

Mit diesem Bit wird die Betriebsart eingestellt.

Ist das Bit auf 1 gesetzt arbeitet der **ADC** im Freerunning Modus. Dabei wird das Datenregister permanent aktualisiert. Ist das Bit hingegen auf 0 gesetzt macht der **ADC** nur eine Single Conversion.

ADIF (ADC Interrupt Flag)

Dieses Bit wird vom **ADC** gesetzt, sobald eine Umwandlung erfolgt ist und das **ADC Data Register** aktualisiert wurde. Das Bit wird bei lesendem Zugriff auf **ADC(L,H)** automatisch (d.h. durch die Hardware) gelöscht.

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

ADCSRA

eine logische 1 in das Register geschrieben wird.

ADIE (ADC Interrupt Enable)

Wenn dieses Bit gesetzt ist und ebenso das **I-Bit** im Statusregister **SREG**, dann wird der **ADC-Interrupt** aktiviert.

ADPS2...ADPS0 (ADC Prescaler Select Bits)

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

$$TF_{min} = \frac{CLK}{200\text{kHz}} = \frac{4000000}{200000} = 20$$

$$TF_{max} = \frac{CLK}{50\text{kHz}} = \frac{4000000}{50000} = 80$$

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

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:

ADCL

```
x = ADCL; // mit uint16_t x
x += (ADCH<<8); // in zwei Zeilen (LSB/MSB-Reihenfolge und
// C-Operatorpriorität sichergestellt)
```

ADCH

oder

```
x = ADCW; // je nach AVR auch x = ADC (siehe avr/ioxxx.h)
```

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	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
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

REFS1...REFS0 (ReferenceSelection Bits)

Mit diesen Bits kann die Referenzspannung eingestellt werden:

REFS1 REFS0 Referenzspannung

0	0	Externes AREF
0	1	AVCC als Referenz
1	0	Reserviert
1	1	Interne 2,56 Volt

ADMUX

ADLAR (ADC Left Adjust Result)

Das ADLAR Bit verändert das Aussehen des Ergebnisses der AD-Wandlung. Bei einer logischen 1 wird das Ergebnis linksbündig ausgegeben, bei einer 0 rechtsbündig. Eine Änderung in diesem Bit beeinflusst das Ergebnis sofort, ganz egal ob bereits eine Wandlung läuft.

MUX4...MUX0

Mit diesen 5 Bits wird der zu messende Kanal bestimmt. Wenn man einen einfachen 1-kanaligen ADC verwendet wird einfach die entsprechende Pinnummer des Ports in die Bits 0...2 eingeschrieben.

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.

Eine Empfehlung ist deswegen 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.

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.

Ein kleines Beispiel für den "single conversion"-Mode bei einem ATmega169 und Nutzung der internen Referenzspannung (beim '169 1,1V bei anderen AVR's auch 2,56V). D.h. das Eingangssignal darf diese Spannung nicht überschreiten, gegebenenfalls mit Spannungsteiler einstellen. Ergebnis der Routine ist der ADC-Wert, also 0 für 0-Volt und 1023 für V_{ref}-Volt.

```
uint16_t ReadChannel(uint8_t mux)
{
    uint8_t i;
    uint16_t result;

    ADMUX = mux; // Kanal waehlen
    ADMUX |= (1<<REFS1) | (1<<REFS0); // interne Referenzspannung nutzen

    ADCSRA = (1<<ADEN) | (1<<ADPS1) | (1<<ADPS0); // Frequenzvorteiler
    // setzen auf 8 (1) und ADC aktivieren (1)

    /* nach Aktivieren des ADC wird ein "Dummy-Readout" empfohlen, man liest
       also einen Wert und verwirft diesen, um den ADC "warmlaufen zu lassen" */
    ADCSRA |= (1<<ADSC); // eine ADC-Wandlung
    while ( ADCSRA & (1<<ADSC) ) {
        ; // auf Abschluss der Konvertierung warten
    }
    result = ADCW; // ADCW muss einmal gelesen werden,
    // sonst wird Ergebnis der nächsten Wandlung
    // nicht übernommen.

    /* Eigentliche Messung - Mittelwert aus 4 aufeinanderfolgenden Wandlungen */
    result = 0;
    for( i=0; i<4; i++ )
    {
        ADCSRA |= (1<<ADSC); // eine Wandlung "single conversion"
        while ( ADCSRA & (1<<ADSC) ) {
            ; // auf Abschluss der Konvertierung warten
        }
        result += ADCW; // Wandlungsergebnisse aufaddieren
    }
    ADCSRA &= ~(1<<ADEN); // ADC deaktivieren (2)

    result /= 4; // Summe durch vier teilen = arithm. Mittelwert
    return result;
}

...

/* Beispielaufrufe: */

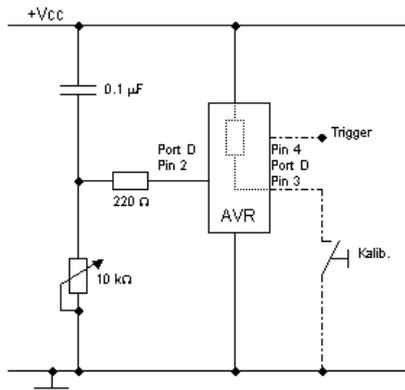
void foo(void)
{
    uint16_t adcval;

    adcval = ReadChannel(0); /* MUX-Bits auf 0b0000 -> Channel 0 */
    ...
    adcval = ReadChannel(2); /* MUX-Bits auf 0b0010 -> Channel 2 */
    ...
}
```

Im Beispiel wird bei jedem Aufruf der ADC aktiviert und nach der Wandlung wieder abgeschaltet, das spart Strom. Will man dies nicht, verschiebt man die mit (1) gekennzeichneten Zeilen in eine Funktion adc_init() o.ä. und löscht die mit (2) markierten Zeilen.

Analog-Digital-Wandlung ohne internen ADC

Messen eines Widerstandes



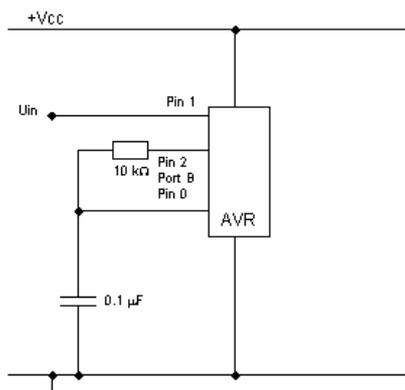
Analoge Werte lassen sich ohne Analog-Digital-Wandler auch indirekt ermitteln. Im Folgenden wird die Messung des an einem Potentiometer eingestellten Widerstands anhand der Ladekurve eines Kondensators erläutert. Bei dieser Methode wird nur ein Portpin benötigt, ein Analog-Digital-Wandler oder Analog-Comparator ist nicht erforderlich. Es wird dazu ein Kondensator und der Widerstand (das Potentiometer) in Reihe zwischen Versorgungsspannung und Masse/GND geschaltet (sogen. RC-Netzwerk). Zusätzlich wird eine Verbindung der Leitung zwischen Kondensator und Potentiometer zu einem Portpin des Controllers hergestellt. Die folgende Abbildung verdeutlicht die erforderliche Schaltung.

Wird der Portpin des Controllers auf Ausgang konfiguriert (im Beispiel $DDRD \text{ |= } (1 \ll PD2)$) und dieser Ausgang auf Logisch 1 ("High", $PORTD \text{ |= } (1 \ll PD2)$) geschaltet, liegt an beiden "Platten" des Kondensators das gleiche Potential V_{CC} an und der Kondensator somit entladen. (Klingt komisch, mit V_{CC} entladen, ist aber so, da an beiden Seiten des Kondensators das gleiche Potential anliegt und somit eine Potentialdifferenz von 0V besteht => Kondensator ist entladen).

Nach einer gewissen Zeit ist der Kondensator entladen und der Portpin wird als Eingang konfiguriert ($DDRD \text{ \&= } \sim(1 \ll PD2)$; $PORTD \text{ \&= } \sim(1 \ll PD2)$), wodurch dieser hochohmig wird. Der Status des Eingangspins (in PIND) ist Logisch 1 (High). 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), wird das Eingangssignal als LOW erkannt (Bit in PIND wird 0). Die Zeitspanne zwischen der Umschaltung von Entladung auf Aufladung und dem Wechsel des Eingangssignals von High auf Low ist ein Maß für den am Potentiometer eingestellten Widerstand. Zur Zeitmessung kann einer der im Controller vorhandenen Timer genutzt werden. Der 220 Ohm Widerstand dient dem Schutz des Controllers. Es würde sonst bei Maximaleinstellung des Potentionmeters (hier 0 Ohm) ein zu hoher Strom fließen, der die Ausgangsstufe des Controllers zerstört.

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

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 Eingangspins 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.

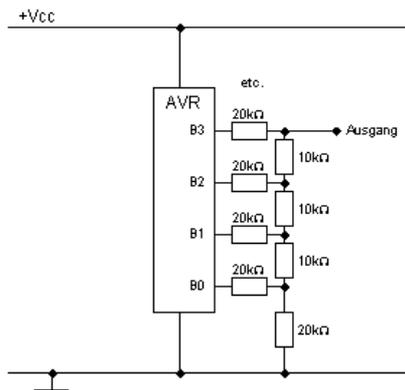
Der Vorteil dieser Schaltung liegt allerdings darin, dass damit direkt Spannungen gemessen werden können.

DAC (Digital Analog Converter)

Mit Hilfe eines Digital-Analog-Konverters (DAC) können wir nun auch Analogsignale ausgeben. Es gibt hier mehrere Verfahren.

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:

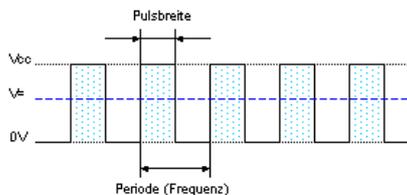


Es sollten selbstverständlich möglichst genaue Widerstände verwendet werden, also nicht unbedingt solche mit einer Toleranz von 10% oder mehr. Weiterhin empfiehlt es sich, je nach Anwendung den Ausgangsstrom über einen Operationsverstärker zu verstärken.

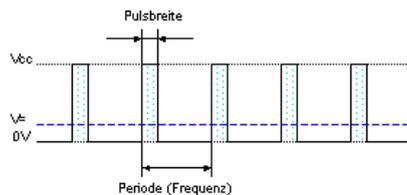
PWM (Pulsweitenmodulation)

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 Mikrocontroller 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 arithmetischen Mittelwert, der je nach Pulsbreite kleiner oder größer ist.



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

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

Hinweis

In den folgenden Überlegungen wird als Controller der 90S2313 vorausgesetzt. Die Theorie ist bei anderen AVR-Controllern vergleichbar, die Pinbelegung allerdings nicht unbedingt, weshalb ein Blick ins entsprechende Datenblatt dringend angeraten wird.

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. Nicht invertierende PWM.
1	0	Der Ausgangspin wird gelöscht beim Hochzählen und gesetzt beim Herunterzählen. Invertierende PWM.
1	1	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:

alte Schreibweise (PWMxx wird nicht mehr akzeptiert)

```
TCCR1A = (1<<PWM11) | (1<<PWM10) | (1<<COM1A1);
```

neue Schreibweise

```
TCCR1A = (1<<WGM11) | (1<<WGM10) | (1<<COM1A1);
```

Damit der Timer/Counter überhaupt läuft, müssen wir im Control Register B **TCCR1B** noch den gewünschten Takt (Vorteiler) 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

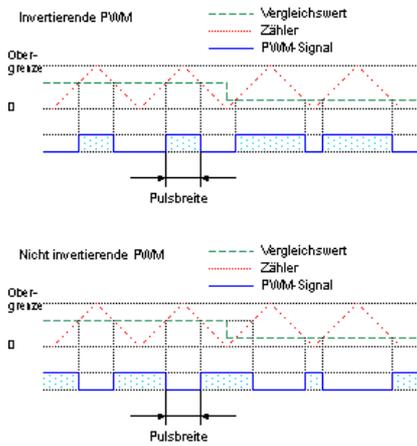
Also um einen Takt von CK / 1024 zu generieren, verwenden wir folgenden Befehl:

```
TCCR1B = (1<<CS12) | (1<<CS10);
```

Jetzt muss nur noch der Vergleichswert festgelegt werden. Diesen schreiben wir in das 16-Bit Timer/Counter Output Compare Register **OCR1A**.

```
OCR1A = xxx;
```

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 beim AT90S2313 nur ein einzelnes **PWM-Signal** mit dieser Methode generieren. Andere AVR-Typen verfügen über bis zu vier PWM-Ausgänge. Zu beachten ist außerdem, dass wenn der OC Pin aktiviert ist, er nichtmehr wie üblich funktioniert und z.B. nicht einfach über PINx ausgelesen werden kann.

Ein Programm, welches an einem ATmega8 den Fast-PWM Modus verwendet, den Modus 14, könnte so aussehen

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

int main()
{
    // OC1A auf Ausgang
    DDRB = (1 << PB1); //ATmega8
    // DDRD = (1 << PD5); //ATmega16
    //
    // Timer 1 einstellen
    //
    // Modus 14:
    //   Fast PWM, Top von ICR1
    //
    //   WGM13   WGM12   WGM11   WGM10
    //   1       1       1       0
    //
    //   Timer Verteiler: 1
    //   CS12    CS11    CS10
    //   0       0       1
    //
    // Steuerung des Ausgangsport: Set at BOTTOM, Clear at match
    //   COM1A1  COM1A0
    //   1       0
    //
    TCCR1A = (1<<COM1A1) | (1<<WGM11);
    TCCR1B = (1<<WGM13) | (1<<WGM12) | (1<<CS10);

    // den Endwert (TOP) für den Zähler setzen
    // der Zähler zählt bis zu diesem Wert

    ICR1 = 0x6FFF;

    // der Compare Wert
    // Wenn der Zähler diesen Wert erreicht, wird mit
    // obiger Konfiguration der OC1A Ausgang abgeschaltet
    // Sobald der Zähler wieder bei 0 startet, wird der
    // Ausgang wieder auf 1 gesetzt
    //
    // Durch Verändern dieses Wertes, werden die unterschiedlichen
    // PWM Werte eingestellt.

    OCR1A = 0x3FFF;

    while( 1 )
    ;
}
```

PWM-Mode Tabelle aus dem Datenblatt des ATmega8515

Mode	WGM13	WGM12	WGM11	WGM10	Timer/Counter	Mode of Operation	TOP	Update of OCR1x at TOV1	Flag set on
0	0	0	0	0		Normal	0xFFFF	Immediate	MAX
1	0	0	0	1		PWM, Phase Correct, 8-Bit	0x00FF	TOP	BOTTOM
2	0	0	1	0		PWM, Phase Correct, 9-Bit	0x01FF	TOP	BOTTOM
3	0	0	1	1		PWM, Phase Correct, 10-Bit	0x03FF	TOP	BOTTOM
4	0	1	0	0		CTC	OCR1A	Immediate	MAX
5	0	1	0	1		Fast PWM, 8-Bit	0x00FF	BOTTOM	TOP
6	0	1	1	0		Fast PWM, 9-Bit	0x01FF	BOTTOM	TOP

7	0	1	1	1	Fast PWM, 10-Bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase an Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase an Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase an Frequency Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	Reserved	-	-	-
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

Für Details der PWM Möglichkeiten, muß immer das jeweilige Datenblatt des Prozessors konsultiert werden, da sich die unterschiedlichen Prozessoren in ihren Möglichkeiten doch stark unterscheiden. Auch muß man aufpassen, welches zu setzende Bit in welchem Register sind. Auch hier kann es sein, dass gleichnamige Konfigurationsbits in unterschiedlichen Konfigurationsregistern (je nach konkretem Prozessortyp) sitzen.

LCD-Ansteuerung

Das LCD und sein Controller

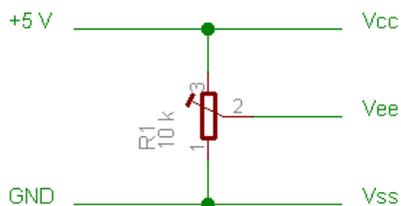
Die meisten Text-LCDs verwenden den Controller [HD44780](#) oder einen kompatiblen (z.B. KS0070) und haben 14 oder 16 Pins. Die Pinbelegung an der LCD-Controller-Platine ist praktisch immer gleich. Trotzdem lohnt sich ein Blick in das Datenblatt des Displays, da es gelegentlich Ausnahmen gibt. Die normale Pinbelegung sieht wie folgt aus:

Pin #	Bezeichnung	Funktion
1	Vss	GND
2	Vcc	5V
3	Vee	Kontrastspannung (0V bis 5V)
4	RS	Register Select (Befehle/Daten)
5	RW	Read/Write
6	E	Enable
7	DB0	Datenbit 0
8	DB1	Datenbit 1
9	DB2	Datenbit 2
10	DB3	Datenbit 3
11	DB4	Datenbit 4
12	DB5	Datenbit 5
13	DB6	Datenbit 6
14	DB7	Datenbit 7
15	A	LED-Beleuchtung, Anode
16	K	LED-Beleuchtung, Kathode

Achtung: Unbedingt von der richtigen Seite zu zählen anfangen! Meistens ist neben Pin 1 eine kleine 1 auf der LCD-Platine, ansonsten im Datenblatt nachschauen.

Bei LCDs mit 16-poligem Anschluss sind die beiden letzten Pins für die Hintergrundbeleuchtung reserviert. Hier unbedingt das Datenblatt zu Rate ziehen, die beiden Anschlüsse sind je nach Hersteller verdreht beschaltet. Falls kein Datenblatt vorliegt, kann man mit einem Durchgangsprüfer feststellen, welcher Anschluss mit Masse (GND) verbunden ist.

Vss wird ganz einfach an GND angeschlossen und Vcc an 5V. Vee kann man testweise auch an GND legen. Wenn das LCD dann zu dunkel sein sollte muss man ein 10k-Potentiometer zwischen GND und 5V schalten, mit dem Schleifer an Vee:



Es gibt zwei verschiedene Möglichkeiten zur Ansteuerung eines solchen Displays: den **8-bit-** und den **4-bit-**Modus.

- Für den **8-bit-Modus** werden (wie der Name schon sagt) alle acht Datenleitungen zur Ansteuerung verwendet, somit kann durch einen Zugriff immer ein ganzes Byte übertragen werden.
- Der **4-bit-Modus** verwendet nur die oberen vier Datenleitungen (**DB4-DB7**). Um ein Byte zu übertragen braucht man somit zwei Zugriffe, wobei zuerst das höherwertige "**Nibble**" (= 4 Bits), also Bit 4 bis Bit 7 übertragen wird und dann das niederwertige, also Bit 0 bis Bit 3. Die unteren Datenleitungen des LCDs, die beim Lesezyklus Ausgänge sind, lässt man offen (siehe Datasheets, z.B. vom KS0070).

Der 4-bit-Modus hat den Vorteil, dass man 4 IO-Pins weniger benötigt als beim 8-bit-Modus, weshalb ich mich hier für eine Ansteuerung mit 4bit entschieden habe.

Neben den vier Datenleitungen (DB4, DB5, DB6 und DB7) werden noch die Anschlüsse **RS**, **RW** und **E** (ist in manchen Unterlagen auch **EN** für *Enable* abgekürzt) benötigt.

- Über **RS** wird ausgewählt, ob man einen Befehl oder ein Datenbyte an das LCD schicken möchte. Ist RS Low, dann wird das ankommende Byte als Befehl interpretiert, ist RS high, dann wird das Byte auf dem LCD angezeigt.
- **RW** legt fest, ob geschrieben oder gelesen werden soll. High bedeutet lesen, low bedeutet schreiben. Wenn man RW auf lesen einstellt und RS auf Befehl, dann kann man das **Busy-Flag** an DB7 lesen, das anzeigt, ob das LCD den vorhergehenden Befehl fertig verarbeitet hat (diese Methode u.a. in der LCD-Library von Peter Fleury verwendet). Ist RS auf Daten eingestellt, dann kann man z.B. den Inhalt des Displays lesen - was jedoch nur in den wenigsten Fällen Sinn macht. Deshalb kann man RW dauerhaft auf low lassen (= an GND anschließen), so dass man noch ein IO-Pin am Controller einspart. Der Nachteil ist, dass man dann das Busy-Flag nicht lesen kann, weswegen man nach jedem Befehl vorsichtshalber ein paar Mikrosekunden warten sollte, um dem LCD Zeit zum Ausführen des Befehls zu geben. Dummerweise schwankt die Ausführungszeit von Display zu Display und ist auch von der Betriebsspannung abhängig. Für professionellere Sachen also lieber den IO-Pin opfern und Busy abfragen.
- Der **E** Anschluss schließlich signalisiert dem LCD, dass die übrigen Datenleitungen jetzt korrekte Pegel angenommen haben und es die gewünschten Daten von den Datenleitungen bzw. Kommandos von den Datenleitungen übernehmen kann.

Anschluss an den Controller

Jetzt da wir wissen, welche Anschlüsse das LCDs benötigt, können wir das LCD mit dem Mikrocontroller verbinden:

Pin #	LCD Bezeichnung	LCD Pin- μ C
1	Vss	GND
2	Vcc	5V
3	Vee	GND oder Poti (siehe oben)
4	RS	PD4 am AVR
5	RW	GND
6	E	PD5 am AVR
7	DB0	offen
8	DB1	offen
9	DB2	offen
10	DB3	offen
11	DB4	PD0 am AVR
12	DB5	PD1 am AVR
13	DB6	PD2 am AVR
14	DB7	PD3 am AVR

Wenn man die Steuerleitungen EN und RS auf Pins an einem anderen Port legen möchte, kann man so wie in diesem [Forumsbeitrag](#) vorgehen.

Ok, alles ist verbunden, wenn man jetzt den Strom einschaltet sollten ein oder zwei schwarze Balken auf dem Display angezeigt werden. Doch wie bekommt man jetzt die Befehle und Daten in das Display?

Programmierung

Datei `lcd-routines.h`

```
// Ansteuerung eines HD44780 kompatiblen LCD im 4-Bit-Interfacemodus
// http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial
//
void lcd_data(unsigned char temp1);
void lcd_string(char *data);
void lcd_command(unsigned char temp1);
void lcd_enable(void);
void lcd_init(void);
void lcd_home(void);
void lcd_clear(void);
void set_cursor(uint8_t x, uint8_t y);

// Hier die verwendete Taktfrequenz in Hz eintragen, wichtig!
#define F_CPU 8000000

// LCD Befehle
#define CLEAR_DISPLAY 0x01
#define CURSOR_HOME 0x02

// Pinbelegung für das LCD, an verwendete Pins anpassen

#define LCD_PORT PORTD
#define LCD_DDR DDRD
#define LCD_RS PD4
#define LCD_EN PD5
```

```
// DB4 bis DB7 des LCD sind mit PD0 bis PD3 des AVR verbunden
```

Datei `lcd-routines.c`:

```
// Ansteuerung eines HD44780 kompatiblen LCD im 4-Bit-Interfacemodus
// http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial
//
// Die Pinbelegung ist über defines in lcd-routines.h einstellbar

#include <avr/io.h>
#include "lcd-routines.h"
#include <util/delay.h>

// sendet ein Datenbyte an das LCD
void lcd_data(unsigned char temp1)
{
    unsigned char temp2 = temp1;

    LCD_PORT |= (1<<LCD_RS);          // RS auf 1 setzen

    temp1 = temp1 >> 4;
    temp1 = temp1 & 0x0F;
    LCD_PORT &= 0xF0;
    LCD_PORT |= temp1;                // setzen
    lcd_enable();

    temp2 = temp2 & 0x0F;
    LCD_PORT &= 0xF0;
    LCD_PORT |= temp2;                // setzen
    lcd_enable();

    _delay_us(42);
}

// sendet einen Befehl an das LCD
void lcd_command(unsigned char temp1)
{
    unsigned char temp2 = temp1;

    LCD_PORT &= ~(1<<LCD_RS);        // RS auf 0 setzen

    temp1 = temp1 >> 4;                // oberes Nibble holen
    temp1 = temp1 & 0x0F;              // maskieren
    LCD_PORT &= 0xF0;
    LCD_PORT |= temp1;                // setzen
    lcd_enable();

    temp2 = temp2 & 0x0F;              // unteres Nibble holen und maskieren
    LCD_PORT &= 0xF0;
    LCD_PORT |= temp2;                // setzen
    lcd_enable();

    _delay_us(42);
}

// erzeugt den Enable-Puls
void lcd_enable(void)
{
    // Bei Problemen ggf. Pause gemäß Datenblatt des LCD Controllers einfügen
    // http://www.mikrocontroller.net/topic/81974#685882
    LCD_PORT |= (1<<LCD_EN);
    _delay_us(1);                      // kurze Pause
    // Bei Problemen ggf. Pause gemäß Datenblatt des LCD Controllers verlängern
    // http://www.mikrocontroller.net/topic/80900
    LCD_PORT &= ~(1<<LCD_EN);
}

// Initialisierung:
// Muss ganz am Anfang des Programms aufgerufen werden.
void lcd_init(void)
{
    LCD_DDR = LCD_DDR | 0x0F | (1<<LCD_RS) | (1<<LCD_EN); // Port auf Ausgang schalten

    // muss 3mal hintereinander gesendet werden zur Initialisierung

    _delay_ms(15);
    LCD_PORT &= 0xF0;
    LCD_PORT |= 0x03;
    LCD_PORT &= ~(1<<LCD_RS);          // RS auf 0
    lcd_enable();

    _delay_ms(5);
    lcd_enable();

    _delay_ms(1);
}
```

```

    lcd_enable();
    _delay_ms(1);

    // 4 Bit Modus aktivieren
    LCD_PORT &= 0xF0;
    LCD_PORT |= 0x02;
    lcd_enable();
    _delay_ms(1);

    // 4Bit / 2 Zeilen / 5x7
    lcd_command(0x28);

    // Display ein / Cursor aus / kein Blinken
    lcd_command(0x0C);

    // inkrement / kein Scrollen
    lcd_command(0x06);

    lcd_clear();
}

// Sendet den Befehl zur Löschung des Displays
void lcd_clear(void)
{
    lcd_command(CLEAR_DISPLAY);
    _delay_ms(5);
}

// Sendet den Befehl: Cursor Home
void lcd_home(void)
{
    lcd_command(CURSOR_HOME);
    _delay_ms(5);
}

// setzt den Cursor in Zeile y (1..4) Spalte x (0..15)
void set_cursor(uint8_t x, uint8_t y)
{
    uint8_t tmp;

    switch (y) {
        case 1: tmp=0x80+0x00+x; break; // 1. Zeile
        case 2: tmp=0x80+0x40+x; break; // 2. Zeile
        case 3: tmp=0x80+0x10+x; break; // 3. Zeile
        case 4: tmp=0x80+0x50+x; break; // 4. Zeile
        default: return; // für den Fall einer falschen Zeile
    }
    lcd_command(tmp);
}

// Schreibt einen String auf das LCD
void lcd_string(char *data)
{
    while(*data) {
        lcd_data(*data);
        data++;
    }
}

```

Ein Hauptprogramm, welches die Funktionen benutzt, sieht zb. so aus:

```

//
// Anpassungen im makefile:
//   ATmega8 => MCU=atmega8 im makefile einstellen
//   lcd-routines.c in SRC = ... Zeile anhängen
//
#include <avr/io.h>
#include "lcd-routines.h"

int main(void)
{
    lcd_init();

    lcd_data('T');
    lcd_data('e');
    lcd_data('s');
    lcd_data('t');

    set_cursor(0,2);

    lcd_string("Hello World!");

    while(1)
    {

```

```

    }
    return 0;
}

```

Wichtig ist dabei, dass die Optimierung bei der Compilierung eingeschaltet ist, sonst stimmen die Zeiten der Funktionen `_delay_us()` und `_delay_ms()` nicht und der Code wird wesentlich länger (Siehe Dokumentation der `libc` im WinAVR).

Ein Hauptprogramm, welches eine Variable ausgibt, sieht zb. so aus. Mittels der `itoa()` Funktion (`itoa` = **I**n**T**eger **T**o **A**scii) wird von einem Zahlenwert eine textuelle Repräsentierung ermittelt (sprich: ein String erzeugt) und dieser String mit der bereits vorhandenen Funktion `lcd_string` ausgegeben:

```

//
// Anpassungen im makefile:
// ATmega8 => MCU=atmega8 im makefile einstellen
// lcd-routines.c in SRC = ... Zeile anhängen
//
#include <avr/io.h>
#include <stdlib.h>
#include "lcd-routines.h"

// Beispiel
int variable = 42;

int main(void)
{
    lcd_init();

    // Ausgabe des Zeichens dessen ASCII-Code gleich dem Variablenwert ist
    // (Im Beispiel entspricht der ASCII-Code 42 dem Zeichen *)
    // http://www.code-knacker.de/ascii.htm
    lcd_data(variable);

    set_cursor(0,2);

    // Ausgabe der Variable als Text in dezimaler Schreibweise
    {
        // ... umwandeln siehe FAQ Artikel bei www.mikrocontroller.net
        // WinAVR hat eine itoa()-Funktion, das erfordert obiges #include <stdlib.h>
        char Buffer[20]; // in diesem {} lokal
        itoa( variable, Buffer, 10 );

        // ... ausgeben
        lcd_string( Buffer );
    }

    while(1)
    {
    }

    return 0;
}

```

Beim Einrichten eines Projekts muss man zu der Datei mit dem Hauptprogramm auch die Datei `lcd-routines.c` in das Projekt aufnehmen. Dies geschieht beim AVR Studio unter Source Files im Fenster AVR GCC oder bei WinAVR im Makefile (z.B. durch `SRC += lcd-routines.c`).

Die Timer/Counter des AVR

Die heutigen Mikrocontroller und insbesondere die RISC-AVRs sind für viele Steuerungsaufgaben 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, Vorgänge in Zeitabständen durchzuführen, die geringer als die Taktfrequenz des Controllers sind. Selbstverständlich sollte die resultierende Frequenz auch noch möglichst genau und stabil sein.

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

Ein Timer ist ganz einfach ein bestimmtes Register im μC , das völlig ohne Zutun des Programms, also per Hardware, hochgezählt wird. Das alleine wäre noch nicht allzu nützlich, wenn nicht dieses Hardwareregister bei bestimmten Zählerständen einen Interrupt auslösen könnte. Ein solches Ereignis ist der Overflow: Da die Bitbreite des Registers beschränkt ist, kommt es natürlich auch vor, dass der Zähler so hoch zählt, dass der nächste Zählerstand mit dieser Bitbreite nicht mehr darstellbar ist und der Zähler wieder auf 0 zurückgesetzt wird. Dieses Ereignis nennt man den Overflow und es ist möglich an dieses Ereignis einen Interrupt zu koppeln.

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

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

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.

Der Vorteiler (Prescaler)

Der Vorteiler dient dazu, den CPU-Takt vorerst um einen einstellbaren Faktor zu reduzieren. Die so geteilte Frequenz wird den Eingängen der Timer zugeführt.

Wenn wir mit einem CPU-Takt von 4 MHz arbeiten und den Vorteiler auf 1024 einstellen, wird also der Timer mit einer Frequenz von 4 MHz / 1024, also mit ca. 4 kHz versorgt. Wenn also der Timer läuft, so wird das Daten- bzw. Zählregister (TCNTx) mit dieser Frequenz inkrementiert.

8-Bit Timer/Counter

Alle AVR-Modelle verfügen über mindestens einen, teilweise sogar zwei, 8-Bit Timer.

Der 8-Bit Timer wird z.B. bei AT90S2313 über folgende Register angesprochen (bei anderen Typen weitestgehend analog):

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, CS01, CS00 (Clock Select Bits)

TCCR0 Diese 3 Bits bestimmen die Quelle für den Timer/Counter:

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.

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.

TCNT0	Bit	7	6	5	4	3	2	1	0	
	Name	MSB							LSB	
	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 1/1024-tel des CPU-Taktes zählen zu lassen, schreiben wir die folgende Befehlszeile:

```
TCCR0 |= (1<<CS00)|(1<<CS02);
```

Der Zähler zählt nun aufwärts bis 255, um dann wieder bei 0 zu beginnen. Der aktuelle Zählerstand steht in TCNT0. 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 Timer-Overflow-Interrupt ausgelöst und die daran gebundene Interrupt-Routine abgearbeitet. Das TOV Flag lässt sich durch das Hineinschreiben einer 1 und nicht wie erwartet einer 0 wieder zurücksetzen.

Beispiel für Compare Match Mode:

```
/*Hinweis: Der Timer ist für die 1Mhz vom interne Takt des ATMEGAs*/
#include <avr/io.h>
#include <avr/interrupt.h>
```

```

//Variablen für die Zeit
volatile unsigned int millisekunden=0;
volatile unsigned int sekunde=0;
volatile unsigned int minute=0;
volatile unsigned int stunde=0;
main()
{
    //Timer 0 konfigurieren

    TCCR0 =(1<<WGM01) |(1<<CS01);
    OCR0=125;

    //Compare Interrupt aktivieren
    TIMSK|=(1<<OCIE0);
    //Globale Interrupts aktivieren
    sei();
    while(1)
    {
        /*Hier kann die aktuelle Zeit ausgegeben werden*/
    }
}

// Der Compare Interrupt Handler
// Wird aufgerufen wenn TCNT0 = 125
ISR (TIMER0_COMP_vect)
{
    millisekunden++;
    if(millisekunden==1000)
    {
        sekunde++;
        millisekunden=0;
        if(sekunde==60)
        {
            minute++;
            sekunde=0;
        }
        if(minute ==60)
        {
            stunde++;
            minute=0;
        }
    }
}

```

Timer-Bitzahlen verschiedener AVR

AVR-Typ	Timer/Counter0	Timer/Counter1	Timer/Counter2
ATtiny2313	8	16	-
ATmega8	8	16	8
ATmega88	8	16	8

16-Bit Timer/Counter

Viele AVR-Modelle besitzen außer den 8-Bit Timern auch 16-Bit Timer. Die 16-Bit Timer/Counter sind etwas komplexer aufgebaut als die 8-Bit Timer/Counter, bieten dafür aber auch viel mehr Möglichkeiten, als da sind:

- Die **PWM**-Betriebsart Erzeugung eines pulswertenmodulierten Ausgangssignals.
- 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:

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, COM1A0 (Compare Match Control Bits)

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.

TCCR1A	COM1A1	COM1A0	Resultat
	0	0	Output-Pin OC1 wird nicht angesteuert.
	0	1	Output-Pin OC1 wird nicht angesteuert. Wird beim Hochzählen der Wert im Vergleichsregister erreicht, so wird der Pin OC1 auf 0 gesetzt.
	1	0	Wird beim Herunterzählen der Wert im Vergleichsregister erreicht, so wird der Pin auf 1 gesetzt. Man nennt dies <i>nicht invertierende PWM</i> . Wird beim Hochzählen der Wert im Vergleichsregister erreicht, so wird der Pin OC1 auf 1 gesetzt.
	1	1	Wird beim Herunterzählen der Wert im Vergleichsregister erreicht, so wird der Pin auf 0 gesetzt. Man nennt dies <i>invertierende PWM</i> .

PWM11, PWM10 (PWM Mode Select Bits)

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.

Timer/Counter Control Register B Timer 1

Bit	7	6	5	4	3	2	1	0
Name	ICNC1	ICES1	- WGM13 WGM12 (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 (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.

ICES1 (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.

CTC1 (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 Vorteiler ein etwas anderes Zählverhalten:

TCCR1B

Wenn der Vorteiler auf 1 gestellt, und C der voreingestellte Vergleichswert ist, dann nimmt das Datenregister, im CPU-Takt betrachtet, folgende Werte an:

... | C-2 | C-1 | C | 0 | 1 | 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-1, C-1, C-1, C-1, C-1, C-1, C-1 | C, 0, 0, 0, 0, 0, 0 | ...

In der PWM-Betriebsart hat dieses Bit keine Funktion.

CS12, CS11, CS10 (Clock Select Bits)

Diese 3 Bits bestimmen die Quelle für den Timer/Counter:

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 T1, fallende Flanke
1	1	1	Externer Pin T1, steigende Flanke

Wenn als Quelle der externe Pin T1 verwendet wird, so wird ein Flankenwechsel auch erkannt, wenn der Pin T1 als Ausgang geschaltet ist.

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	
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	

TCNT1H
TCNT1L

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.

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	

OCR1H
OCR1L

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 **TCNT1** 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 **OCR1L** und erst danach **OCR1H** ausgelesen werden.

Wenn in das Register geschrieben werden soll, müssen ebenfalls alle Interrupts gesperrt werden. Dann muss zuerst das **OCR1H**-Register und erst danach das **OCR1L**-Register geschrieben werden, also genau die umgekehrte Reihenfolge wie beim Lesen des Registers.

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.

ICR1H
ICR1L

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

Bei Verwendung des Compilers wird der Zugriff in der korrekten Reihenfolge ausgeführt, wenn man das 16-Bit register ICR1 verwendet anstatt zwei Zugriffe auf 8-Bit Register.

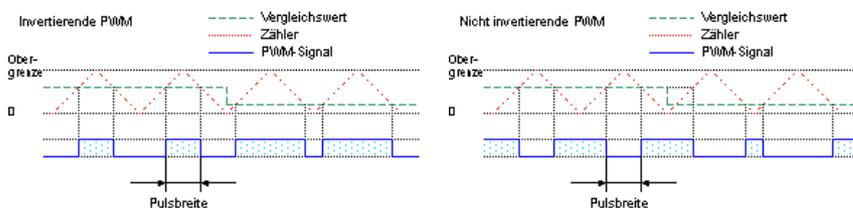
Die PWM-Betriebsart

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 Ausgangspin **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



Vergleichswert-Überprüfung (Compare Match)

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.

Zu erwähnen ist in dem Zusammenhang, dass das zur Compare-Einheit gehörende Interrupt-Flag erst beim auf die Übereinstimmung der Werte folgenden Timertakt gesetzt wird. Das ist v.a. deshalb wichtig, da es sonst bei $OCR_{nx} = 0$ einen undefinierten Zustand gäbe.

Möchte man ein Compare-Ereignis 100 Takte nach dem Timerüberlauf auslösen, dann muss in das betreffende Compare-Register eine 99 geschrieben werden.

CTC-Betriebsart (Clear Timer on Compare Match)

Das sogenannte **Compare Match-Ereignis** kann auch dazu verwendet werden, um den Timer automatisch zurückzusetzen (d.h. das **TCNT**-Register wird zu Null gesetzt). Diese Betriebsart heißt "Clear Timer on Compare Match", also auf deutsch "Lösche Timer bei Vergleichsübereinstimmung".

Mit dieser Funktionalität ist es möglich, sehr präzise Taktsignale zu erzeugen, ohne dabei programmtechnisch eingreifen zu müssen. **Diese Funktion ersetzt das bei anderen Controllern und Timern ohne Compare-Einheit erforderliche Timer Reload** (also das Nachladen des Zählregisters mit "Überlaufwert minus gewünschte Taktzahl bis zum Überlauf", v.a. verbreitet bei 8051er- μ Cs).

Zur Erzeugung eines Taktes per Hardware muss lediglich eine der CTC-Betriebsarten ausgewählt werden und einer der **OC_{nx}**-Pins so gesetzt werden, dass er bei Auftreten des Compare Match getoggelt wird (über die **COM**-Bits).

Die Frequenz des Taktes am entsprechenden Ausgang ist dann

$$f_{OC} = \frac{f_{CPU}}{\text{Prescaler} \cdot (OCR_{nx} + 1)}$$

Diese Betriebsart macht das Timer-Nachladen, das bei AVR's, die ja im Unterschied zu 8051-Derivaten keine Auto-Reload-Funktion haben, immer mit Ungenauigkeiten und programmtechnischen Klammzügen verbunden ist, überflüssig. Ist das **OCR_{nx}** einmal gesetzt, dann wird das Signal am Ausgang kontinuierlich ausgegeben, ohne dass die Anwendersoftware eingreifen muss (es sei denn, die Frequenz soll geändert werden).

Beim ATmega8 hat der 8-Bit-Timer 0 keine Compare-Einheit, so dass dort CTC und auch sonstige automatische Vergleichsoperationen nicht möglich sind. Bei Timer 1 und Timer 2 ist das jedoch möglich. Bei den neueren AVR's besitzen i.d.R. *alle* Timer eine oder mehrere Compare-Einheiten, so dass dort eine größere Flexibilität gegeben ist.

Im Unterschied zu den PWM-Betriebsarten wird die Registeraktualisierung bei CTC nicht automatisch synchronisiert. Schreibt man einen neuen Compare-

Wert, dann wird dieser sofort übernommen, was zu Fehlfunktionen führen kann, wenn der neue Compare-Wert höher ist, als der aktuelle Stand von TCNTx. In den PWM-Betriebsarten wird hingegen der TOP-Wert synchron bei Erreichen von TOP oder BOTTOM aktualisiert.

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.

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.

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.

TIMSK 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.

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.

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.

OCF1A (Output Compare Flag Timer/Counter 1)

TIFR

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.

ICF1 (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.

TOV0 (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.

Warteschleifen (delay.h)

Der Programmablauf kann verschiedene Arten von Wartefunktionen erfordern:

- Warten im Sinn von Zeitvertrödeln
- Warten auf einen bestimmten Zustand an den I/O-Pins
- Warten auf einen bestimmten Zeitpunkt (siehe Timer)
- Warten auf einen bestimmten Zählerstand (siehe Counter)

Der einfachste Fall, das Zeitvertrödeln, kann in vielen Fällen und mit großer Genauigkeit anhand der avr-libc Bibliotheksfunktionen `_delay_ms()` und `_delay_us()` erledigt werden. Die Bibliotheksfunktionen sind einfachen Zählschleifen (Warteschleifen) vorzuziehen, da leere Zählschleifen ohne besondere Vorkehrungen sonst bei eingeschalteter Optimierung vom avr-gcc-Compiler wegoptimiert werden. Weiterhin sind die Bibliotheksfunktionen bereits darauf vorbereitet, die in `F_CPU` definierte Taktfrequenz zu verwenden. Ausserdem sind die Funktionen der Bibliothek wirklich getestet.

Einfach!? Schon, aber während gewartet wird, macht der μ C nichts anderes mehr. Die Wartefunktion blockiert den Programmablauf. Möchte man einerseits warten, um z.B. eine LED blinken zu lassen und gleichzeitig andere Aktionen ausführen z.B. weitere LED bedienen, sollten die Timer/Counter des AVR verwendet werden.

Die Bibliotheksfunktionen funktionieren allerdings nur dann korrekt, wenn sie mit zur Übersetzungszeit (beim Compilieren) bekannten konstanten Werten aufgerufen werden. Der Quellcode muss mit eingeschalteter Optimierung übersetzt werden, sonst wird sehr viel Maschinencode erzeugt und die Wartezeiten stimmen nicht mehr mit dem Parameter überein.

Abhängig von der Version der Bibliothek verhalten sich die Bibliotheksfunktionen etwas unterschiedlich.

avr-libc Versionen kleiner 1.6

Die Wartezeit der Funktion `_delay_ms()` ist auf $262,14\text{ms}/F_{\text{CPU}}$ (in MHz) begrenzt, d.h. bei 20 MHz kann man nur max. 13,1ms warten. Die Wartezeit der Funktion `_delay_us()` ist auf $768\text{us}/F_{\text{CPU}}$ (in MHz) begrenzt, d.h. bei 20 MHz kann man nur max. 38,4us warten. Längere Wartezeiten müssen dann über einen mehrfachen Aufruf in einer Schleife gelöst werden.

Beispiel: Blinken einer LED an PORTB Pin PB0 im ca. 1s Rhythmus

```
#include <avr/io.h>
#ifndef F_CPU
/* Definiere F_CPU, wenn F_CPU nicht bereits vorher definiert
   (z.B. durch Übergabe als Parameter zum Compiler innerhalb
   des Makefiles). Zusätzlich Ausgabe einer Warnung, die auf die
   "nachträgliche" Definition hinweist */
#warning "F_CPU war noch nicht definiert, wird nun mit 3686400 definiert"
#define F_CPU 3686400UL /* Quarz mit 3.6864 Mhz */
#endif
#include <util/delay.h> /* in älteren avr-libc Versionen <avr/delay.h> */

/*
   lange, variable Verzögerungszeit, Einheit in Millisekunden

Die maximale Zeit pro Funktionsaufruf ist begrenzt auf
262.14 ms / F_CPU in MHz (im Beispiel:
262.1 / 3.6864 = max. 71 ms)

Daher wird die kleine Warteschleife mehrfach aufgerufen,
um auf eine längere Wartezeit zu kommen. Die zusätzliche
Prüfung der Schleifenbedingung lässt die Wartezeit geringfügig
ungenau werden (macht hier vielleicht 2-3ms aus).
*/

void long_delay(uint16_t ms) {
    for(; ms>0; ms--) _delay_ms(1);
}

int main( void )
{
    DDRB = ( 1 << PB0 ); // PB0 an PORTB als Ausgang setzen

    while( 1 ) { // Endlosschleife
        PORTB ^= ( 1 << PB0 ); // Toggle PB0 z.B. angeschlossene LED
        long_delay(1000); // Eine Sekunde warten...
    }
}
```

```

    return 0;
}

```

avr-libc Versionen ab 1.6

`_delay_ms()` kann mit einem Argument bis 6553,5 ms (= 6,5535 Sekunden) benutzt werden. Wird die früher gültige Grenze von 262,14 ms/F_CPU (in MHz) überschritten, so arbeitet `_delay_ms()` einfach etwas ungenauer und zählt nur noch mit einer Auflösung von 1/10 ms. Eine Verzögerung von 1000,10 ms ließe sich nicht mehr von einer von 1000,19 ms unterscheiden. Ein Verlust, der sich im Allgemeinen verschmerzen lässt. Dem Programmierer wird keine Rückmeldung gegeben, dass die Funktion ggf. gröber arbeitet, d.h. wenn es darauf ankommt, bitte den Parameter wie bisher geschickt wählen.

Die Funktion `_delay_us()` wurde ebenfalls erweitert. Wenn deren maximal als genau behandelbares Argument überschritten wird, benutzt diese intern `_delay_ms()`. Damit gelten in diesem Fall die `_delay_ms()` Einschränkungen.

Beispiel: Blinken einer LED an PORTB Pin PB0 im ca. 1s Rhythmus, avr-libc ab Version 1.6

```

#include <avr/io.h>
#ifndef F_CPU
/* Definiere F_CPU, wenn F_CPU nicht bereits vorher definiert
   (z.B. durch Übergabe als Parameter zum Compiler innerhalb
   des Makefiles). Zusätzlich Ausgabe einer Warnung, die auf die
   "nachträgliche" Definition hinweist */
#warning "F_CPU war noch nicht definiert, wird nun mit 3686400 definiert"
#define F_CPU 3686400UL /* Quarz mit 3.6864 Mhz */
#endif
#include <util/delay.h>

int main( void )
{
    DDRB = ( 1 << PB0 ); // PB0 an PORTB als Ausgang setzen

    while( 1 ) { // Endlosschleife
        PORTB ^= ( 1 << PB0 ); // Toggle PB0 z.B. angeschlossene LED
        _delay_ms(1000); // Eine Sekunde +/-1/10000 Sekunde warten...
                          // funktioniert nicht mit Bibliotheken vor 1.6
    }
    return 0;
}

```

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:

```

uint8_t 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 (der Compiler sollte jedoch eine entsprechende Warnung ausgeben). Das Programm würde sich also hier aufhängen und auf ewig in der Schleife drehen. Und hier genau kommt der Watchdog zum Zug.

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. Einige Controller haben einen eigenen Watchdog Oszillator, z.B. der Tiny2313 mit 128kHz. 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 innerhalb unserer Hauptschleife passieren.

Um ein unbeabsichtigtes Ausschalten des Watchdogs zu verhindern, muss ein spezielles Prozedere verwendet werden, um den WD auszuschalten. Es 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:

WatchDog Timer Control Register

In diesem Register stellen wir ein, wie wir den Watchdog verwenden möchten.

Das Register ist wie folgt aufgebaut:

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)

WDTCR Wenn dieses Bit gesetzt wird, so wird der Watchdog aktiviert. Das Bit kann nur gelöscht werden, solange das Bit **WDTOE** auf 1 steht.

WDP2, WDP1, WDP0 (Watchdog Timer Prescaler Bits)

Diese 3 Bits bestimmen die Anzahl Oszillatorzyklen für den Watchdog, also, wie lange es dauert, bis ein Reset ausgelöst wird:

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` (`#include <avr/wdt.h>`) in die Quelldatei eingebunden werden. Danach können die folgenden Funktionen verwendet werden:

- **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. Einige Timeout-Werte sind als Konstanten vordefiniert
Mögliche Timeoutwerte:

Konstante	Wert	TimeOut
<code>WDTO_15MS</code>	0	15 ms
<code>WDTO_30MS</code>	1	30 ms
<code>WDTO_60MS</code>	2	60 ms
<code>WDTO_120MS</code>	3	120 ms
<code>WDTO_250MS</code>	4	250 ms
<code>WDTO_500MS</code>	5	500 ms
<code>WDTO_1S</code>	6	1 s
<code>WDTO_2S</code>	2	2 s

`wdt_disable()`

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

`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 den AVR zurücksetzt.

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

Watchdog-Anwendungshinweise

Ob nun der Watchdog als Schutzfunktion überhaupt verwendet werden soll, hängt stark von der Anwendung, der genutzten Peripherie und dem Umfang und

der Qualitätssicherung des Codes ab. Will man sicher gehen, dass ein Programm sich nicht in einer Endlosschleife verfängt, ist der Watchdog das geeignete Mittel dies zu verhindern. Weiterhin kann bei geschickter Programmierung der Watchdog dazu genutzt werden, bestimmte Stromsparfunktionen zu implementieren. Bei einigen neueren AVR's (z.B. dem ATtiny13) kann der Watchdog auch direkt als Timer genutzt werden, der den Controller aus einem Schlafmodus aufweckt. Auch dies kann im **WDTCR**-Register eingestellt werden. Außerdem bietet der WD die einzige Möglichkeit einen beabsichtigten System-Reset (ein "richtiger Reset", kein "jmp 0x0000") ohne externe Beschaltung auszulösen, was z.B. bei der Implementierung eines Bootloaders nützlich ist. Bei bestimmten Anwendungen kann die Nutzung des WD als "ultimative Deadlock-Sicherung für nicht bedachte Zustände" natürlich immer als zusätzliche Sicherung dienen.

Es besteht die Möglichkeit herauszufinden, ob ein Reset durch den Watchdog ausgelöst wurde (beim ATmega16 z.B. Bit WDRF in MCUCSR). Diese Information sollte auch genutzt werden, falls ein WD-Reset in der Anwendung nicht planmäßig implementiert wurde. Zum Beispiel kann man eine LED an einen freien Pin hängen, die nur bei einem Reset durch den WD aufleuchtet oder aber das "Ereignis WD-Reset" im internen EEPROM des AVR absichern, um die Information später z.B. über UART oder ein Display auszugeben (oder einfach den EEPROM-Inhalt über die ISP/JTAG-Schnittstelle auslesen).

Bei neueren AVR-Typen bleibt der Watchdog auch nach einem Reset durch den Watchdog aktiviert. Wenn ein Programm nach dem Neustart bis zur erstmaligen Rückstellung des Watchdogs länger braucht, als die im Watchdog eingestellte Zeit, sollte man den Watchdog explizit möglichst früh deaktivieren. Ansonsten resetet der Watchdog den Controller immerfort von Neuem. Die frühe Deaktivierung sollte durch eine Funktion erfolgen, die noch vor allen anderen Operationen (insbesondere vor dem mglw. länger andauernden internen Initialisierungen vor dem Sprung zu main()) ausgeführt wird. Näheres zur Implementierung mit avr-gcc/avr-libc findet sich in der Dokumentation der avr-libc (Suchbegriffe: attribut, section, init).

Siehe auch:

- [Dokumentation der avr-libc](#) Abschnitt Modules/Watchdog timer handling
- [Bug in ATtiny2313?](#)

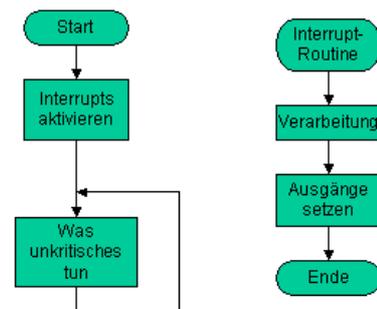
Programmieren mit Interrupts

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.

[Ausführlicher Thread im Forum](#)



Anforderungen an Interrupt-Routinen

Um unliebsamen Überraschungen vorzubeugen, sollten einige Grundregeln bei der Implementierung der Interruptroutinen beachtet werden. Interruptroutinen soll möglichst kurz und schnell abarbeitbar sein, daraus folgt:

- Keine umfangreichen Berechnungen innerhalb der Interruptroutine. (*)
- Keine langen Programmschleifen.
- Obwohl es möglich ist, während der Abarbeitung einer Interruptroutine andere oder sogar den gleichen Interrupt wieder zuzulassen, wird davon ohne genaue Kenntnis der internen Abläufe dringend abgeraten.

Interruptroutinen (ISRs) sollten also möglichst kurz sein und keine Schleifen mit vielen Durchläufen enthalten. Längere Operationen können meist in einen "Interrupt-Teil" in einer ISR und einen "Arbeitsteil" im Hauptprogramm aufgetrennt werden. Z.B. Speichern des Zustands aller Eingänge im EEPROM in bestimmten Zeitabständen: ISR-Teil: Zeitvergleich (Timer,RTC) mit Logzeit/-intervall. Bei Übereinstimmung ein globales Flag setzen (volatile bei Flag-Deklaration nicht vergessen, s.u.). Dann im Hauptprogramm prüfen, ob das Flag gesetzt ist. Wenn ja: die Daten im EEPROM ablegen und Flag löschen.

(*) Hinweis: Es gibt allerdings die seltene Situation, dass man gerade eingelese ADC-Werte sofort verarbeiten muss. Besonders dann, wenn man mehrere Werte sehr schnell hintereinander bekommt. Dann bleibt einem nichts anderes übrig, als die Werte noch in der ISR zu verarbeiten. Kommt aber sehr selten vor und sollte durch geeignete Wahl des Systemtaktes bzw. Auswahl des Controllers vermieden werden!

Interrupt-Quellen

Die folgenden Ereignisse können einen Interrupt auf einem AVR AT90S2313 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

Die Anzahl der möglichen Interruptquellen variiert zwischen den verschiedenen Typen. Im Zweifel hilft ein Blick ins Datenblatt ("Interrupt Vectors").

Register

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

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 (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.

GIMSK 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.

INT0 (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.

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 (External Interrupt Flag 1)

Dieses Bit wird gesetzt, wenn am **INT1**-Pin eine Interrupt-Kondition, entsprechend der Konfiguration, erkannt wird. Wenn das Global Enable Interrupt Flag gesetzt ist, wird die Interruptroutine angesprungen.

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

INTF0 (External Interrupt Flag 0)

Dieses Bit wird gesetzt, wenn am **INT0**-Pin eine Interrupt-Kondition, entsprechend der Konfiguration, erkannt wird. Wenn das Global Enable Interrupt Flag gesetzt ist, wird die Interruptroutine angesprungen.

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

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/W
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. (für andere AVR Controller siehe Abschnitt "Sleep-Mode")

ISC11, ISC10 (Interrupt Sense Control 1 Bits)

MCUCR

Diese beiden Bits bestimmen, ob die steigende oder die fallende Flanke für die Interrupterkennung am **INT1**-Pin ausgewertet wird.

ISC11	ISC10	Bedeutung
		Low Level an INT1 erzeugt einen Interrupt.
0	0	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 Flanke 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
		Low Level an INT0 erzeugt einen Interrupt.
0	0	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 Flanke an INT0 erzeugt einen Interrupt.
1	1	Die steigende Flanke an INT0 erzeugt einen Interrupt.

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 GIE-bit zu setzen, wird dringend davon abgeraten. 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 möglicherweise alle anderen Interrupts mit niedrigerer Priorität. Dies ist einer der Gründe, weshalb die Interrupt-Routinen sehr kurz gehalten werden sollen.

Interrupts mit dem AVR GCC Compiler (WinAVR)

Funktionen zur Interrupt-Verarbeitung werden in den Includedateien *interrupt.h* der avr-libc zur Verfügung gestellt (bei älterem Quellcode zusätzlich *signal.h*).

```
// fuer sei(), cli() und ISR():
#include <avr/interrupt.h>
```

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

```
sei();
```

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

```
cli();
```

Oft steht man vor der Aufgabe, dass eine Codesequenz nicht unterbrochen werden darf. Es liegt dann nahe, zu Beginn dieser Sequenz ein **cli()** und am Ende ein **sei()** einzufügen. Dies ist jedoch ungünstig, wenn die Interrupts vor Aufruf der Sequenz deaktiviert waren und danach auch weiterhin deaktiviert bleiben sollen. Ein **sei()** würde ungeachtet des vorherigen Zustands die Interrupts aktivieren, was zu unerwünschten Seiteneffekten führen kann. Die aus dem folgenden Beispiel ersichtliche Vorgehensweise ist in solchen Fällen vorzuziehen:

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

//...
```

```

void NichtUnterbrechenBitte(void)
{
    uint8_t tmp_sreg; // temporaerer Speicher fuer das Statusregister

    tmp_sreg = SREG; // Statusregister (also auch das I-Flag darin) sichern
    cli();          // Interrupts global deaktivieren

    /* hier "unterbrechungsfreier" Code */

    /* Beispiel Anfang
    JTAG-Interface eines ATmega16 per Software deaktivieren
    und damit die JTAG-Pins an PORTC für "general I/O" nutzbar machen
    ohne die JTAG-Fuse-Bit zu aendern. Dazu ist eine "timed sequence"
    einzuhalten (vgl Datenblatt ATmega16, Stand 10/04, S. 229):
    Das JTD-Bit muss zweimal innerhalb von 4 Taktzyklen geschrieben
    werden. Ein Interrupt zwischen den beiden Schreibzugriffen wuerde
    die erforderliche Sequenz "brechen", das JTAG-Interface bliebe
    weiterhin aktiv und die IO-Pins weiterhin für JTAG reserviert. */

    MCUCSR |= (1<<JTD);
    MCUCSR |= (1<<JTD); // 2 mal in Folge ,vgl. Datenblatt fuer mehr Information

    /* Beispiel Ende */

    SREG = tmp_sreg; // Status-Register wieder herstellen
                    // somit auch das I-Flag auf gesicherten Zustand setzen
}

void NichtSoGut(void)
{
    cli();

    /* hier "unterbrechungsfreier" Code */

    sei();
}

int main(void)
{
    //...

    cli();
    // Interrupts global deaktiviert

    NichtUnterbrechenBitte();
    // auch nach Aufruf der Funktion deaktiviert

    sei();
    // Interrupts global aktiviert

    NichtUnterbrechenBitte();
    // weiterhin aktiviert
    //...

    /* Verdeutlichung der unguenstigen Vorgehensweise mit cli/sei: */
    cli();
    // Interrupts jetzt global deaktiviert

    NichtSoGut();
    // nach Aufruf der Funktion sind Interrupts global aktiviert
    // dies ist mglw. ungewollt!
    //...
}

```

Zu den aktivierten Interrupts ist eine Funktion zu programmieren, deren Code aufgerufen wird, wenn der betreffende Interrupt auftritt (Interrupt-Handler, Interrupt-Service-Routine). Dazu existiert die Definition (ein Makro) **ISR**.

ISR

(*ISR()* ersetzt bei neueren Versionen der avr-libc *SIGNAL()*). *SIGNAL* sollte nicht mehr genutzt werden, zur Portierung von *SIGNAL* nach *ISR* siehe den [Anhang](#).

```

#include <avr/interrupt.h>
//...
ISR(Vectorname) /* vormalis: SIGNAL(siglabel) dabei Vectorname != siglabel ! */
{
    /* Interrupt Code */
}

```

Mit *ISR* wird eine Funktion für die Bearbeitung eines Interrupts eingeleitet. Als Argument muss dabei die Benennung des entsprechenden Interruptvektors angegeben werden. Diese sind in den jeweiligen Includedateien *IOxxx.h* zu finden. Die Bezeichnung entspricht dem Namen aus dem Datenblatt, bei dem

die Leerzeichen durch Unterstriche ersetzt sind und ein `_vect` angehängt ist.

Als Beispiel ein Ausschnitt aus der Datei für den ATmega8 (bei WinAVR Standardinstallation in `C:\WinAVR\avr\include\avr\iom8.h`) in der neben den aktuellen Namen für `ISR` (`*_vect`) noch die Bezeichnungen für das inzwischen nicht mehr aktuelle `SIGNAL` (`SIG_*`) enthalten sind.

```
//...
/* $Id: iom8.h,v 1.13 2005/10/30 22:11:23 joerg_wunsch Exp $ */

/* avr/iom8.h - definitions for ATmega8 */
//...

/* Interrupt vectors */

/* External Interrupt Request 0 */
#define INT0_vect          _VECTOR(1)
#define SIG_INTERRUPT0    _VECTOR(1)

/* External Interrupt Request 1 */
#define INT1_vect          _VECTOR(2)
#define SIG_INTERRUPT1    _VECTOR(2)

/* Timer/Counter2 Compare Match */
#define TIMER2_COMP_vect  _VECTOR(3)
#define SIG_OUTPUT_COMPARE2 _VECTOR(3)

/* Timer/Counter2 Overflow */
#define TIMER2_OVF_vect   _VECTOR(4)
#define SIG_OVERFLOW2    _VECTOR(4)

/* Timer/Counter1 Capture Event */
#define TIMER1_CAPT_vect  _VECTOR(5)
#define SIG_INPUT_CAPTURE1 _VECTOR(5)

/* Timer/Counter1 Compare Match A */
#define TIMER1_COMPA_vect _VECTOR(6)
#define SIG_OUTPUT_COMPARE1A _VECTOR(6)

/* Timer/Counter1 Compare Match B */
#define TIMER1_COMPB_vect _VECTOR(7)
#define SIG_OUTPUT_COMPARE1B _VECTOR(7)

//...
```

Mögliche Funktionsrumpfe für Interruptfunktionen sind zum Beispiel:

```
#include <avr/interrupt.h>
/* veraltet: #include <avr/signal.h> */

ISR(INT0_vect) /* veraltet: SIGNAL(SIG_INTERRUPT0) */
{
    /* Interrupt Code */
}

ISR(TIMER0_OVF_vect) /* veraltet: SIGNAL(SIG_OVERFLOW0) */
{
    /* Interrupt Code */
}

ISR(USART_RXC_vect) /* veraltet: SIGNAL(SIG_UART_RECV) */
{
    /* Interrupt Code */
}

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

Auf die korrekte Schreibweise der Vektorbezeichnung ist zu achten. Der gcc-Compiler prüft erst ab Version 4.x, ob ein Signal/Interrupt der angegebenen Bezeichnung tatsächlich in der Includedatei definiert ist und gibt andernfalls eine Warnung aus. Bei WinAVR (ab 2/2005) wurde die Überprüfung auch in den mitgelieferten Compiler der Version 3.x integriert. Aus dem gcc-Quellcode Version 3.x selbst erstellte Compiler enthalten die Prüfung nicht (vgl. [AVR-GCC](#)).

Während der Ausführung der 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.

Unterbrechbare Interruptroutinen

"Faustregel": im Zweifel **ISR**. Die nachfolgend beschriebene Methode nur dann verwenden, wenn man sich über die unterschiedliche Funktionsweise im Klaren ist.

```
#include <avr/interrupt.h>
//...
void XXX_vect(void) __attribute__((interrupt));
void XXX_vect(void) {
    //...
}
```

Hierbei steht XXX für den oben beschriebenen Namen des Vektors (also z.B. `void TIMER0_OVF_vect(void)`...). Der Unterschied im Vergleich zu ISR ist, dass hier 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.

siehe auch: Hinweise in [AVR-GCC](#)

siehe dazu: http://www.nongnu.org/avr-libc/user-manual/group_avr_interrupts.html

Datenaustausch mit Interrupt-Routinen

Variablen die sowohl in Interrupt-Routinen (ISR = Interrupt Service Routine(s)), als auch vom übrigen Programmcode geschrieben oder gelesen werden, müssen mit einem **volatile** deklariert werden. Damit wird dem Compiler mitgeteilt, dass der Inhalt der Variablen vor jedem Lesezugriff aus dem Speicher gelesen und nach jedem Schreibzugriff in den Speicher geschrieben wird. Ansonsten könnte der Compiler den Code so optimieren, dass der Wert der Variablen nur in Prozessorregistern zwischengespeichert wird, die nichts von der Änderung woanders mitbekommen.

Zur Veranschaulichung ein Codefragment für eine Tastenentprellung mit Erkennung einer "lange gedrückten" Taste.

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

// Schwellwerte
// Entprellung:
#define CNTDEBOUNCE 10
// "lange gedreuekt:"
#define CNTREPEAT 200

// hier z.B. Taste an Pin2 PortA "active low" = 0 wenn gedreuekt
#define KEY_PIN PINA
#define KEY_PINNO PA2

// beachte: volatile!
volatile uint8_t gKeyCounter;

// Timer-Compare Interrupt ISR, wird z.B. alle 10ms ausgefuehrt
ISR(TIMER1_COMPA_vect)
{
    // hier wird gKeyCounter veraendert. Die uebrigen
    // Programmteile muessen diese Aenderung "sehen":
    // volatile -> aktuellen Wert immer in den Speicher schreiben
    if ( !(KEY_PIN & (1<<KEY_PINNO)) ) {
        if (gKeyCounter < CNTREPEAT) gKeyCounter++;
    }
    else {
        gKeyCounter = 0;
    }
}

//...

int main(void)
{
    //...
    /* hier: Initialisierung der Ports und des Timer-Interrupts */
    //...
    // hier wird auf gKeyCounter zugegriffen. Dazu muss der in der
    // ISR geschriebene Wert bekannt sein:
    // volatile -> aktuellen Wert immer aus dem Speicher lesen
    if ( gKeyCounter > CNTDEBOUNCE ) { // Taste mind. 10*10 ms "prellfrei"
        if (gKeyCounter == CNTREPEAT) {
            /* hier: Code fuer "Taste lange gedreuekt" */
        }
        else {
            /* hier: Code fuer "Taste kurz gedreuekt" */
        }
    }
}
//...
```

}

Wird innerhalb einer ISR mehrfach auf eine mit volatile deklarierte Variable zugegriffen, wirkt sich dies ungünstig auf die Verarbeitungsgeschwindigkeit aus, da bei jedem Zugriff mit dem Speicherinhalt abgeglichen wird. Da bei AVR-Controllern *innerhalb* einer ISR keine Unterbrechungen zu erwarten sind, bietet es sich an, einen Zwischenspeicher in Form einer lokalen Variable zu verwenden, deren Inhalt zu Beginn und am Ende mit dem der volatile Variable synchronisiert wird. Lokale Variable werden bei eingeschalteter Optimierung mit hoher Wahrscheinlichkeit in Prozessorregistern verwaltet und der Zugriff darauf ist daher nur mit wenigen internen Operationen verbunden. Die ISR aus dem vorherigen Beispiel lässt sich so optimieren:

```
//...
ISR(TIMER1_COMPA_vect)
{
    uint8_t tmp_kc;

    tmp_kc = gKeyCounter; // Uebernahme in lokale Arbeitsvariable

    if ( !(KEY_PIN & (1<<KEY_PINNO)) ) {
        if (tmp_kc < CNTREPEAT) {
            tmp_kc++;
        }
    }
    else {
        tmp_kc = 0;
    }

    gKeyCounter = tmp_kc; // Zurueckschreiben
}
//...
```

Zum Vergleich die Disassemblies (Ausschnitte der "lss-Dateien", kompiliert für ATmega162) im Anschluss. Man erkennt den viermaligen Zugriff auf die Speicheradresse von *gKeyCounter* (hier 0x032A) in der ISR ohne "Cache"-Variable und den zweimaligen Zugriff in der Variante mit Zwischenspeicher. Im Beispiel ist der Vorteil gering, bei komplexeren Routinen kann die Zwischenspeicherung in lokalen Variablen jedoch zu deutlicheren Verbesserungen führen.

```
ISR(TIMER1_COMPA_vect)
{
    86a:    1f 92    push    r1
    86c:    0f 92    push    r0
    86e:    0f b6    in     r0, 0x3f    ; 63
    870:    0f 92    push    r0
    872:    11 24    eor    r1, r1
    874:    8f 93    push    r24
    if ( !(KEY_PIN & (1<<KEY_PINNO)) ) {
    876:    ca 99    sbic   0x19, 2 ; 25
    878:    0a c0    rjmp   .+20      ; 0x88e <__vector_13+0x24>
        if (gKeyCounter < CNTREPEAT) gKeyCounter++;
    87a:    80 91 2a 03    lds    r24, 0x032A
    87e:    88 3c    cpi    r24, 0xC8    ; 200
    880:    40 f4    brcc   .+16      ; 0x892 <__vector_13+0x28>
    882:    80 91 2a 03    lds    r24, 0x032A
    886:    8f 5f    subi   r24, 0xFF    ; 255
    888:    80 93 2a 03    sts    0x032A, r24
    88c:    02 c0    rjmp   .+4      ; 0x892 <__vector_13+0x28>
    }
    else {
        gKeyCounter = 0;
    88e:    10 92 2a 03    sts    0x032A, r1
    892:    8f 91    pop    r24
    894:    0f 90    pop    r0
    896:    0f be    out    0x3f, r0    ; 63
    898:    0f 90    pop    r0
    89a:    1f 90    pop    r1
    89c:    18 95    reti

ISR(TIMER1_COMPA_vect)
{
    86a:    1f 92    push    r1
    86c:    0f 92    push    r0
    86e:    0f b6    in     r0, 0x3f    ; 63
    870:    0f 92    push    r0
    872:    11 24    eor    r1, r1
    874:    8f 93    push    r24
    uint8_t tmp_kc;

    tmp_kc = gKeyCounter;
    876:    80 91 2a 03    lds    r24, 0x032A

    if ( !(KEY_PIN & (1<<KEY_PINNO)) ) {
    87a:    ca 9b    sbis   0x19, 2 ; 25
    87c:    02 c0    rjmp   .+4      ; 0x882 <__vector_13+0x18>
    87e:    80 e0    ldi    r24, 0x00    ; 0
    880:    03 c0    rjmp   .+6      ; 0x888 <__vector_13+0x1e>
        if (tmp_kc < CNTREPEAT) {
    882:    88 3c    cpi    r24, 0xC8    ; 200
    884:    08 f4    brcc   .+2      ; 0x888 <__vector_13+0x1e>
            tmp_kc++;
    886:    8f 5f    subi   r24, 0xFF    ; 255
```

```

    }
}
else {
    tmp_kc = 0;
}

gKeyCounter = tmp_kc;
888:    80 93 2a 03    sts    0x032A, r24
88c:    8f 91          pop    r24
88e:    0f 90          pop    r0
890:    0f be          out    0x3f, r0      ; 63
892:    0f 90          pop    r0
894:    1f 90          pop    r1
896:    18 95          reti

```

volatile und Pointer

Bei **volatile** in Verbindung mit Pointern ist zu beachten, ob der Pointer selbst oder die Variable auf die der Pointer zeigt **volatile** ist.

```

volatile uint8_t *a; // das Ziel von a ist volatile
uint8_t *volatile a; // a selbst ist volatile

```

Falls der Pointer volatile ist (zweiter Fall im Beispiel), ist zu beachten, dass der Wert des Pointers, also eine Speicheradresse, intern in mehr als einem Byte verwaltet wird. Lese- und Schreibzugriffe im Hauptprogramm (ausserhalb von Interrupt-Routinen) sind daher so zu implementieren, dass alle Teilbytes der Adresse konsistent bleiben, vgl. dazu den folgenden Abschnitt.

Variablen größer 1 Byte

Bei Variablen größer ein Byte, auf die in Interrupt-Routinen und im Hauptprogramm zugegriffen wird, muss darauf geachtet werden, dass die Zugriffe auf die einzelnen Bytes außerhalb der ISR nicht durch einen Interrupt unterbrochen werden. (Allgemeinplatz: AVR's sind 8-bit Controller). Zur Veranschaulichung ein Codefragment:

```

//...
volatile uint16_t gMyCounter16bit;
//...
ISR(...)
{
//...
    gMyCounter16Bit++;
//...
}

int main(void)
{
    uint16_t tmpCnt;
//...
    // nicht gut: Mglw. hier ein Fehler, wenn ein Byte von MyCounter
    // schon in tmpCnt kopiert ist aber vor dem Kopieren des zweiten Bytes
    // ein Interrupt auftritt, der den Inhalt von MyCounter verändert.
    tmpCnt = gMyCounter16bit;

    // besser: Änderungen "außerhalb" verhindern -> alle "Teilbytes"
    // bleiben konsistent
    cli(); // Interrupts deaktivieren
    tmpCnt = gMyCounter16Bit;
    sei(); // wieder aktivieren

    // oder: vorheriger Status des globalen Interrupt-Flags bleibt erhalten
    uint8_t sreg_tmp;
    sreg_tmp = SREG; /* Sichern */
    cli()
    tmpCnt = gMyCounter16Bit;
    SREG = sreg_tmp; /* Wiederherstellen */

    // oder: mehrfach lesen, bis man konsistente Daten hat
    uint16_t count1 = gMyCounter16Bit;
    uint16_t count2 = gMyCounter16Bit;
    while (count1 != count2) {
        count1 = count2;
        count2 = gMyCounter16Bit;
    }
    tmpCnt = count1;
//...
}

```

Die avr-libc bietet ab Version 1.6.0(?) einige Hilfsfunktionen/Makros, mit der im Beispiel oben gezeigten Funktionalität, die zusätzlich auch so genannte [memory barriers](#) beinhalten. Diese stehen nach `#include <util/atomic.h>` zur Verfügung.

```

//...

```

```
#include <util/atomic.h>
//...

// analog zu cli, Zugriff, sei:
ATOMIC_BLOCK(ATOMIC_FORCEON) {
    tmpCnt = gMyCounter16Bit;
}

// oder:

// analog zu Sicherung des SREG, cli, Zugriff und Zurückschreiben des SREG:
ATOMIC_BLOCK(ATOMIC_RESTORESTATE) {
    tmpCnt = gMyCounter16Bit;
}

//...
```

- siehe auch [Dokumentation der avr-libc zu atomic.h](#)

Interrupt-Routinen und Registerzugriffe

Falls Register sowohl im Hauptprogramm als auch in Interrupt-Routinen verändert werden, ist darauf zu achten, dass diese Zugriffe sich nicht überlappen. Nur wenige Anweisungen lassen sich in sogenannte "atomare" Zugriffe übersetzen, die nicht von Interrupt-Routinen unterbrochen werden können.

Zur Veranschaulichung eine Anweisung, bei der ein Bit und im Anschluss drei Bits in einem Register gesetzt werden:

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

int main(void)
{
//...
    PORTA |= (1<<PA0);

    PORTA |= (1<<PA2) | (1<<PA3) | (1<<PA4);
//...
}
```

Der Compiler übersetzt diese Anweisungen für einen ATmega128 bei Optimierungsstufe "S" nach:

```
...
d2:  PORTA |= (1<<PA0);
      d8 9a          sbi    0x1b, 0 ; 27 (a)

      PORTA |= (1<<PA2) | (1<<PA3) | (1<<PA4);
d4:  8b b3          in     r24, 0x1b      ; 27 (b)
d6:  8c 61          ori    r24, 0x1C      ; 28 (c)
d8:  8b bb          out   0x1b, r24      ; 27 (d)
...
```

Das Setzen des einzelnen Bits wird bei eingeschalteter Optimierung für Register im unteren Speicherbereich in eine einzige Assembler-Anweisung (sbi) übersetzt und ist nicht anfällig für Unterbrechnungen durch Interrupts. Die Anweisung zum Setzen von drei Bits wird jedoch in drei abhängige Assembler-Anweisungen übersetzt und bietet damit zwei "Angriffspunkte" für Unterbrechnungen. Eine Interrupt-Routine könnte nach dem Laden des Ausgangszustands in den Zwischenspeicher (hier Register 24) den Wert des Registers ändern, z.B. ein Bit löschen. Damit würde der Zwischenspeicher nicht mehr mit dem tatsächlichen Zustand übereinstimmen aber dennoch nach der Bitoperation (hier ori) in das Register zurückgeschrieben.

Beispiel: PORTA sei anfangs 0b00000000. Die erste Anweisung (a) setzt Bit 0, PORTA ist danach 0b00000001. Nun wird im ersten Teil der zweiten Anweisung der Portzustand in ein Register eingelesen (b). Unmittelbar darauf (vor (c)) "feuert" ein Interrupt, in dessen Interrupt-Routine Bit 0 von PORTA gelöscht wird. Nach Verlassen der Interrupt-Routine hat PORTA den Wert 0b00000000. In den beiden noch folgenden Anweisungen des Hauptprogramms wird nun der zwischengespeicherte "alte" Zustand 0b00000001 mit 0b00011100 logisch-oder-verknüpft (c) und das Ergebnis 0b00011101 in PortA geschrieben (d). Obwohl zwischenzeitlich Bit 0 gelöscht wurde, ist es nach (d) wieder gesetzt.

Lösungsmöglichkeiten:

- Register ohne besondere Vorkehrungen nicht in Interruptroutinen *und* im Hauptprogramm verändern.
- Interrupts vor Veränderungen in Registern, die auch in ISRs verändert werden, deaktivieren ("cli").
- Bits einzeln löschen oder setzen. sbi und cbi können nicht unterbrochen werden. Vorsicht: nur Register im unteren Speicherbereich sind mittels sbi/cbi ansprechbar. Der Compiler kann nur für diese sbi/cbi-Anweisungen generieren. Für Register außerhalb dieses Adressbereichs ("Memory-Mapped"-Register) werden auch zur Manipulation einzelner Bits abhängige Anweisungen erzeugt (lds, ..., sts).
- siehe auch: [Dokumentation der avr-libc](#) Frequently asked Questions/Fragen Nr. 1 und 8. (Stand: avr-libc Vers. 1.0.4)

Was macht das Hauptprogramm?

Im einfachsten (Ausnahme-)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 einer Endlosschleife verharrt. Sämtliche Funktionen werden dann in den ISRs abgearbeitet. Diese Vorgehensweise ist jedoch bei den meisten Anwendungen schlecht: man verschenkt eine Verarbeitungsebene und hat außerdem möglicherweise Probleme durch Interruptroutinen, die zu viel Verarbeitungszeit benötigen.

Normalerweise wird man in den Interruptroutinen nur die bei Auftreten des jeweiligen Interruptereignisses unbedingt notwendigen Operationen ausführen lassen. Alle weniger kritischen Aufgaben werden dann im Hauptprogramm abgearbeitet.

- siehe auch: [Dokumentation der avr-libc](#) Abschnitt Modules/Interrupts and Signals

Sleep-Modes

AVR Controller verfügen über eine Reihe von sogenannten *Sleep-Modes* ("Schlaf-Modi"). Diese ermöglichen es, Teile des Controllers abzuschalten. Zum Einen kann damit besonders bei Batteriebetrieb Strom gespart werden, zum Anderen können Komponenten des Controllers deaktiviert werden, die die Genauigkeit des Analog-Digital-Wandlers bzw. des Analog-Comparators negativ beeinflussen. Der Controller wird durch Interrupts aus dem Schlaf geweckt. Welche Interrupts den jeweiligen Schlafmodus beenden, ist einer Tabelle im Datenblatt des jeweiligen Controllers zu entnehmen. Die Funktionen (eigentlich Makros) der avr-libc stehen nach Einbinden der header-Datei *sleep.h* zur Verfügung.

- **set_sleep_mode(uint8_t mode)**

Setzt den Schlafmodus, der bei Aufruf von sleep() aktiviert wird. In sleep.h sind einige Konstanten definiert (z.B. SLEEP_MODE_PWR_DOWN). Die definierten Modi werden jedoch nicht alle von sämtlichen AVR-Controllern unterstützt.

- **sleep_enable()**

aktiviert den gesetzten Schlafmodus, versetzt den Controller aber noch nicht in den Schlafmodus

- **sleep_cpu()**

Versetzt den Controller in den Schlafmodus (sleep_cpu wird im Prinzip durch die Assembler-Anweisung sleep ersetzt)

- **sleep_disable()**

deaktiviert den gesetzten Schlafmodus

- **sleep_mode()**

Versetzt den Controller in den mit set_sleep_mode gewählten Schlafmodus. Das Makro entspricht sleep_enable()+sleep_cpu()+sleep_disable(), beinhaltet also nicht die Aktivierung von Interrupts (besser nicht benutzen).

Bei Anwendung von sleep_cpu() müssen Interrupts also bereits freigegeben sein (sei()), da der Controller sonst nicht mehr "aufwachen" kann. sleep_mode() ist nicht geeignet für die Verwendung in ISR Interrupt-Service-Routinen, da bei deren Abarbeitung Interrupts global deaktiviert sind und somit auch die möglichen "Aufwachinterrupts". Abhilfe: stattdessen sleep_enable(), sei(), sleep_cpu(), sleep_disable() und evtl. cli() verwenden (vgl. Dokumentation der avr-libc).

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

int main(void)
{
    ...

    while (1) {
    ...
        set_sleep_mode(SLEEP_MODE_PWR_DOWN);
        sleep_mode();

        // Code hier wird erst nach Auftreten eines entsprechenden
        // "Aufwach-Interrupts" verarbeitet
    ...
    }
}
```

In älteren Versionen der avr-libc wurden nicht alle AVR-Controller durch die sleep-Funktionen richtig angesteuert. Mit avr-libc 1.2.0 wurde die Anzahl der unterstützten Typen jedoch deutlich erweitert. Bei nicht-unterstützten Typen erreicht man die gewünschte Funktionalität durch direkte [Bitmanipulation](#) der entsprechenden Register (vgl. Datenblatt) und Aufruf des Sleep-Befehls via Inline-Assembler oder sleep_cpu():

```
#include <avr/io.h>
...
// Sleep-Mode "Power-Save" beim ATmega169 "manuell" aktivieren
SMCR = (3<<SM0) | (1<<SE);
asm volatile ("sleep::"); // alternativ sleep_cpu() aus sleep.h
...
```

- siehe auch: [Dokumentation der avr-libc](#) Abschnitt Modules/Power Management and Sleep-Modes
- [Forenbeitrag](#) zur "Nichtverwendung" von sleep_mode in ISRs.

Speicherzugriffe

Atmel AVR-Controller verfügen typisch über drei Speicher:

- **RAM:** Im RAM (genauer statisches RAM/SRAM) wird vom gcc-Compiler Platz für Variablen reserviert. Auch der Stack befindet sich im RAM. Dieser Speicher ist "flüchtig", d.h. der Inhalt der Variablen geht beim Ausschalten oder einem Zusammenbruch der Spannungsversorgung verloren.
- Programmspeicher: Ausgeführt als FLASH-Speicher, seitenweise wiederbeschreibbar. Darin ist das Anwendungsprogramm abgelegt.
- **EEPROM:** Nichtflüchtiger Speicher, d.h. der einmal geschriebene Inhalt bleibt auch ohne Stromversorgung erhalten. Byte-weise schreib/lesbar. Im EEPROM werden typischerweise gerätespezifische Werte wie z.B. Kalibrierungswerte von Sensoren abgelegt.

Einige AVR's besitzen keinen RAM-Speicher, lediglich die Register können als "Arbeitsvariablen" genutzt werden. Da die Anwendung des avr-gcc auf solch "kleinen" Controllern ohnehin selten sinnvoll ist und auch nur bei einigen RAM-losen Typen nach "[Bastelarbeiten](#)" möglich ist, werden diese Controller hier nicht weiter berücksichtigt. Auch EEPROM-Speicher ist nicht auf allen Typen verfügbar. Generell sollten die nachfolgenden Erläuterungen auf alle ATmega-Controller und die größeren AT90-Typen übertragbar sein. Für die Typen ATtiny2313, ATtiny26 und viele weitere der "ATtiny-Reihe" gelten die Ausführungen ebenfalls.

RAM

Die Verwaltung des RAM-Speichers erfolgt durch den Compiler, im Regelfall ist beim Zugriff auf Variablen im RAM nichts Besonderes zu beachten. Die Erläuterungen in jedem brauchbaren C-Buch gelten auch für den vom avr-gcc-Compiler erzeugten Code.

Um Speicher dynamisch (während der Laufzeit) zu reservieren, kann `malloc()` verwendet werden. `malloc(size)` "alloziert" (~reserviert) einen gewissen Speicherblock mit `size` Bytes. Ist kein Platz für den neuen Block, wird `NULL` (0) zurückgegeben.

Wird der angelegte Block zu klein (groß), kann die Größe mit `realloc()` verändert werden. Den allozierten Speicherbereich kann man mit `free()` wieder freigeben. Wenn das Freigeben eines Blocks vergessen wird spricht man von einem "Speicherleck" (memory leak).

`malloc()` legt Speicherblöcke im **Heap** an, belegt man zuviel Platz, dann wächst der Heap zu weit nach oben und überschreibt den Stack, und der Controller kommt in Teufels Küche. Das kann leider nicht nur passieren wenn man insgesamt zu viel Speicher anfordert, sondern auch wenn man Blöcke unterschiedlicher Größe in ungünstiger Reihenfolge alloziert/freigibt (siehe Artikel [Heap-Fragmentierung](#)). Aus diesem Grund sollte man `malloc()` auf Mikrocontrollern sehr sparsam (am besten gar nicht) verwenden.

Beispiel zur Verwendung von `malloc()`:

```
#include <stdlib.h>

void foo(void) {
    // neuen speicherbereich anlegen,
    // platz für 10 uint16
    uint16_t* pBuffer = malloc(10 * sizeof(uint16_t));

    // darauf zugreifen, als wärs ein gewohnter Buffer
    pBuffer[2] = 5;

    // Speicher (unbedingt!) wieder freigeben
    free(pBuffer);
}
```

Wenn (wie in obigem Beispiel) dynamischer Speicher nur für die Dauer einer Funktion benötigt und am Ende wieder freigegeben wird, bietet es sich an, statt `malloc()` `alloca()` zu verwenden. Der Unterschied zu `malloc()` ist, dass der Speicher auf dem Stack reserviert wird, und beim Verlassen der Funktion automatisch wieder freigegeben wird. Es kann somit kein Speicherleck und keine Fragmentierung entstehen.

siehe auch:

- <http://www.nongnu.org/avr-libc/user-manual/malloc.html>

Programmspeicher (Flash)

Ein Zugriff auf Konstanten im Programmspeicher ist mittels avr-gcc nicht "transparent" möglich. D.h. es sind besondere Zugriffsfunktionen erforderlich, um Daten aus diesem Speicher zu lesen. Grundsätzlich basieren alle Zugriffsfunktionen auf der Assembler-Anweisung `lpm` (load program memory, bei AVR Controllern mit mehr als 64kB Flash auch `elpm`). Die Standard-Laufzeitbibliothek des avr-gcc (die avr-libc) stellt diese Funktionen nach Einbinden der Header-Datei `pgmspace.h` zur Verfügung. Mit diesen Funktionen können einzelne Bytes, Datenworte (16bit) und Datenblöcke gelesen werden.

Deklarationen von Variablen im Flash-Speicher werden durch das "Attribut" `PROGMEM` ergänzt. Lokale Variablen (eigentlich Konstanten) innerhalb von Funktionen können ebenfalls im Programmspeicher abgelegt werden. Dazu ist bei der Definition jedoch ein `static` voranzustellen, da solche "Variablen" nicht auf dem Stack bzw. (bei Optimierung) in Registern verwaltet werden können. Der Compiler "wirft" eine Warnung falls `static` fehlt.

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

//...

/* Byte */
const uint8_t pgmFooByte PROGMEM = 123;

/* Wort */
```

```

const uint16_t pgmFooWort PROGMEM = 12345;

/* Byte-Feld */
const uint8_t pgmFooByteArray1[] PROGMEM = { 18, 3, 70 };
const uint8_t pgmFooByteArray2[] PROGMEM = { 30, 7, 79 };

/* Zeiger */
const uint8_t *pgmPointerToArray1 PROGMEM = pgmFooByteArray1;
const uint8_t *pgmPointerArray[] PROGMEM = { pgmFooByteArray1, pgmFooByteArray2 };
//...

void foo(void)
{
    static /*const*/ uint8_t pgmTestByteLocal PROGMEM = 0x55;
    static /*const*/ char pgmTestStringLocal[] PROGMEM = "im Flash";
    // so nicht (static fehlt): char pgmTestStringLocalFalsch [] PROGMEM = "so nicht";

    // ...
}

```

Byte lesen

Mit der Funktion `pgm_read_byte` aus `pgmspace.h` erfolgt der Zugriff auf die Daten. Parameter der Funktion ist die Adresse des Bytes im Flash-Speicher.

```

const uint8_t pgmFooByte PROGMEM = 123;
const uint8_t pgmFooByteArray1[] PROGMEM = { 18, 3, 70 };

// Wert der Ram-Variablen myByte auf den Wert von pgmFooByte setzen:
uint8_t myByte;

myByte = pgm_read_byte(&pgmFooByte);
// myByte hat nun den Wert 123

//...

// Schleife ueber ein Array aus Byte-Werten im Flash
uint8_t i;

for (i=0;i<3;i++) {
    myByte = pgm_read_byte(&pgmFooByteArray1[i]);
    // mach' was mit myByte....
}

```

Wort lesen

Für "einfache" 16-bit breite Variablen erfolgt der Zugriff analog zum Byte-Beispiel, jedoch mit der Funktion `pgm_read_word`.

```

const uint16_t pgmFooWort PROGMEM = 12345;

uint16_t myWord;

myWord = pgm_read_word(&pgmFooWort);

```

Zeiger auf Werte im Flash sind ebenfalls 16 Bits "groß" (Stand `avr-gcc 3.4.x`). Damit ist der mögliche Speicherbereich für "Flash-Konstanten" auf 64kB begrenzt.

```

uint8_t *ptrToArray;

ptrToArray = (uint8_t*)(pgm_read_word(&pgmPointerToArray1));
// ptrToArray enthält nun die Startadresse des Byte-Arrays pgmFooByteArray1
// Allerdings würde ein direkter Zugriff mit diesem Pointer (z.B. temp=*ptrToArray)
// ''nicht'' den Inhalt von pgmFooByteArray1[0] liefern, sondern von einer Speicherstelle
// im ''RAM'', die die gleiche Adresse hat wie pgmFooByteArray1[0]
// Daher muss nun die Funktion pgm_read_byte() benutzt werden, die die in ptrToArray
// enthaltene Adresse benutzt und auf das Flash zugreift.

for (i=0;i<3;i++) {
    myByte = pgm_read_byte(ptrToArray+i);
    // mach' was mit myByte... (18, 3, 70)
}

ptrToArray = (uint8_t*)(pgm_read_word(&pgmPointerArray[1]));

// ptrToArray enthält nun die Adresse des ersten Elements des Byte-Arrays pgmFooByteArray2
// da im zweiten Element des Pointer-Arrays pgmPointerArray die Adresse
// von pgmFooByteArray2 abgelegt ist

for (i=0;i<3;i++) {
    myByte = pgm_read_byte(ptrToArray+i);
}

```

```

    } // mach' was mit myByte... (30, 7, 79)
}

```

Strings lesen

Strings sind in C ja nichts anderes als eine Abfolge von Zeichen. Der prinzipielle Weg ist daher identisch zu "Bytes lesen" wobei allerdings auf die [Besonderheiten von Strings](#) (0-Terminierung) geachtet werden muss, bzw. diese zur Steuerung einer Schleife über die Zeichen im String ausgenutzt werden kann

```

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

const char pgmString[] PROGMEM = "Hallo world";

int main()
{
    char c;
    const char* addr;

    ...

    addr = &pgmString;
    while( ( c = pgm_read_byte( addr++ ) != '\0' ) {
        // mach was mit c
    }
}

```

Zur Unterstützung des Programmierers steht das Repertoire der str... Funktionen auch in jeweils eine Variante zur Verfügung, die mit dem Flash Speicher arbeiten kann. Die Funktionsnamen wurden dabei um ein '_P' ergänzt.

```

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

const char pgmString[] PROGMEM = "Hallo world";

int main()
{
    char string[40];

    ...

    strcpy_P( string, pgmString );
}

```

Floats und Structs lesen

Um komplexe Datentypen (structs), nicht-integer Datentypen (floats) aus dem Flash auszulesen, sind Hilfsfunktionen erforderlich. Einige Beispiele:

```

/* Beispiel float aus Flash */
float pgmFloatArray[3] PROGMEM = {1.1, 2.2, 3.3};
//...

/* liest float von Flash-Adresse addr und gibt diese als return-value zurueck */
inline float pgm_read_float(const float *addr)
{
    union
    {
        uint16_t i[2]; // 2 16-bit-Worte
        float f;
    } u;

    u.i[0]=pgm_read_word((PGM_P)addr);
    u.i[1]=pgm_read_word((PGM_P)addr+2);

    return u.f;
}
...

void egal(void)
{
    int i;
    float f;

    for (i=0;i<3;i++) {
        f = pgm_read_float(&pgmFloatArray[i]); // entspr. "f = pgmFloatArray[i];"
        // mach' was mit f
    }
}

```

```
}

```

TODO: Beispiele fuer structs und pointer aus flash auf struct im flash (menues, state-machines etc.). Eine kleine Einleitung insbesondere auch in Bezug auf die auftretenden Schwierigkeiten liefert [\[2\]](#).

Array aus Strings im Flash-Speicher

Arrays aus Strings im Flash-Speicher werden in zwei Schritten angelegt: Zuerst die einzelnen Elemente des Arrays und im Anschluss ein Array, in dem die Startadressen der Strings abgelegt werden. Zum Auslesen wird zuerst die Adresse des i-ten Elements aus dem Array im Flash-Speicher gelesen, die im Anschluss dazu genutzt wird, auf das Element (den String) selbst zuzugreifen.

```
#include <stdint.h>
#include <avr/pgmspace.h>

const char str1[] PROGMEM = "first_A";
const char str2[] PROGMEM = "second_A";
const char str3[] PROGMEM = "third_A";

const char *strarray1[] PROGMEM = {
    str1,
    str2,
    str3
};

int main(void)
{
    int i, j, l;
    const char *pstrflash;
    char work[20], work2[20];
    // fuer Simulation: per volatile Optimierung verhindern,
    // da c nicht genutzt
    volatile char c;

    for ( i = 0; i < (sizeof(strarray1)/sizeof(strarray1[0])) ; i++ ) {

        // setze Pointer auf die Adresse des i-ten Elements des
        // "Flash-Arrays" (str1, str2, ...)
        pstrflash = (const char*)( pgm_read_word( &(strarray1[i]) ) );

        // kopiere den Inhalt der Zeichenkette von der
        // in pstrflash abgelegten Adresse in das work-Array
        // analog zu strcpy( work, strarray1[i]) wenn alles im RAM
        strcpy_P( work, pstrflash );
        // verkuerzt:
        strcpy_P( work2, (const char*)( pgm_read_word( &(strarray1[i]) ) ) );

        // Zeichen-fuer-Zeichen
        l = strlen_P( pstrflash );
        for ( j=0; j < l; j++ ) {
            // analog zu c=strarray[i][j] wenn alles im RAM
            c = (char)( pgm_read_byte( pstrflash++ ) );
        }

        while (1) { ; }
    }
}
```

Siehe dazu auch die avr-libc FAQ: "How do I put an array of strings completely in ROM?"

Vereinfachung für Zeichenketten (Strings) im Flash

Zeichenketten können innerhalb des Quellcodes als "Flash-Konstanten" ausgewiesen werden. Dazu dient das Makro PSTR aus pgmspace.h. Dies erspart die getrennte Deklaration mit PROGMEM-Attribut.

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

#define MAXLEN 30

char StringImFlash[] PROGMEM = "Erwin Lindemann"; // im "Flash"
char StringImRam[MAXLEN];

//...
strcpy(StringImRam, "Mueller-Luedenscheidt");

if (!strncmp_P(StringImRam, StringImFlash, 5)) {
    // mach' was, wenn die ersten 5 Zeichen identisch – hier nicht
}
else {
```

```

    // der Code hier wuerde ausgefuehrt
}

if (!strcmp_P(StringImRam, PSTR("Mueller-Schmitt"), 5)) {
    // der Code hier wuerde ausgefuehrt, die ersten 5 Zeichen stimmen ueberein
}
else {
    // wuerde bei nicht-Uebereinstimmung ausgefuehrt
}
//...

```

Aber Vorsicht: Ersetzt man zum Beispiel

```

const char textImFlashOK[] PROGMEM = "mit[]";
// = Daten im "Flash", textImFlashOK* zeigt auf Flashadresse

```

durch

```

const char* textImFlashProblem PROGMEM = "mit*";
// Konflikt: Daten im BSS (lies: RAM), textImFlashFAIL* zeigt auf Flashadresse

```

dann kann es zu Problemen mit AVR-GCC kommen. Zu erkennen daran, dass der Initialisierungsstring von "textImFlashProblem" zu den Konstanten ans Ende des Programmcodes gelegt wird (BSS), von dem aus er zur Benutzung eigentlich ins RAM kopiert werden sollte (und wird). Da der lesende Code (mittels `pgm_read*`) trotzdem an einer Stelle vorne im Flash sucht, wird Unsinn gelesen. Dies scheint ein weiteres Problem des AVR-GCC (gesehen bei `avr-gcc 3.4.1` und `3.4.2`) bei der Anpassung an die Harvard-Architektur zu sein (konstanter Pointer auf variable Daten?!). Abhilfe ("Workaround"): Initialisierung bei Zeichenketten mit `[]` oder gleich im Code `PSTR("...")` nutzen.

Übergibt man Zeichenketten (genauer: die Adresse des ersten Zeichens), die im Flash abgelegt sind an eine Funktion, muss diese entsprechend programmiert sein. Die Funktion selbst hat keine Möglichkeit zu unterscheiden, ob es sich um eine Adresse im Flash oder im RAM handelt. Die `avr-libc` und viele andere `avr-gcc`-Bibliotheken halten sich an die Konvention, dass Namen von Funktionen die Flash-Adressen erwarten mit dem Suffix `_p` (oder `_P`) versehen sind.

Eine Funktion, die einen im Flash abgelegten String z.B. an eine UART ausgibt, würde dann so aussehen:

```

void uart_puts_p(const char *text)
{
    char Zeichen;

    while (Zeichen = pgm_read_byte(text))
    { /* so lange, wie mittels pgm_read_byte ein Zeichen vom Flash gelesen
       werden konnte, welches nicht das "String-Endezeichen" darstellt */

        /* Das gelesene Zeichen über die normalen Kanäle verschicken */
        uart_putc(Zeichen);
        text++;
    }
}

```

Von einigen Bibliotheken werden Makros definiert, die "automatisch" ein `PSTR` bei Verwendung einer Funktion einfügen. Ein Blick in den Header-File der Bibliothek zeigt, ob dies der Fall ist. Ein Beispiel aus P. Fleurys `lcd-Library`:

```

// Ausschnitt aus dem Header-File lcd.h der "Fleury-LCD-Lib."
//...
extern void lcd_puts_p(const char *progmem_s);
#define lcd_puts_P(__s) lcd_puts_p(PSTR(__s))
//...

// in einer Anwendung (wieauchimmer.c)
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <string.h>
#include "lcd.h"

char StringImFlash[] PROGMEM = "Erwin Lindemann"; // im "Flash"

//...
lcd_puts_p(StringImFlash);
lcd_puts_P("Dr. Kloebner");
// daraus wird wg. #define lcd_put_P...: lcd_puts_p( PSTR("Dr. Kloebner") );
//...

```

Flash in der Anwendung schreiben

Bei AVR's mit "self-programming"-Option (auch bekannt als Bootloader-Support) können Teile des Flash-Speichers auch vom Anwendungsprogramm selbst beschrieben werden. Dies ist nur möglich, wenn die Schreibfunktionen in einem besonderen Speicherbereich (boot-section) des Programmspeichers/Flash abgelegt sind. Bei wenigen "kleinen" AVR's gibt es keine gesonderte Boot-Section, bei diesen kann der Flashspeicher von jeder Stelle des Programms geschrieben werden. Für Details sei hier auf das jeweilige Controller-Datenblatt und die Erläuterungen zum Modul `boot.h` der `avr-libc` verwiesen. Es

existieren auch Application-Notes dazu bei atmel.com, die auf avr-gcc-Code übertragbar sind.

Warum so kompliziert?

Zu dem Thema, warum die Verarbeitung von Werten aus dem Flash-Speicher so "kompliziert" ist, sei hier nur kurz erläutert: Die Harvard-Architektur des AVR weist getrennte Adressräume für Programm(Flash)- und Datenspeicher(RAM) auf. Der C-Standard und der gcc-Compiler sehen keine unterschiedlichen Adressräume vor. Hat man zum Beispiel eine Funktion `string_uart(const char* s)` und übergibt an diese Funktion die Adresse einer Zeichenkette (einen Pointer, z.B. `0x01fe`), "weiß" die Funktion nicht, ob die Adresse auf den Flash-Speicher oder den/das RAM zeigt. Allein aus dem Pointer-Wert (der Zahl) kann nicht geschlossen werden, ob ein "einfaches" `zeichen_uart(s[i])` oder `zeichen_uart(pgm_read_byte(&s[i]))` genutzt werden muss, um das i-te Zeichen auszugeben.

Einige AVR-Compiler "tricksen" etwas, in dem sie für einen Pointer nicht nur die Adresse anlegen, sondern zusätzlich zu jedem Pointer den Ablageort (Flash oder RAM) intern sichern. Bei Aufruf einer Funktion wird dann bei Pointer-Parametern neben der Adresse auch der Speicherbereich, auf den der Pointer zeigt, übergeben. Dies hat jedoch nicht nur Vorteile; Erläuterungen warum dies so ist, führen an dieser Stelle zu weit.

- siehe auch: [Dokumentation der avr-libc](#) Abschnitte Modules/Program Space String Utilities und Abschnitt Modules/Bootloader Support Utilities

EEPROM

Man beachte, dass der EEPROM-Speicher nur eine begrenzte Anzahl von Schreibzugriffen zulässt. Beschreibt man eine EEPROM-Zelle öfter als die im Datenblatt zugesicherte Anzahl (typisch 100.000), wird die Funktion der Zelle nicht mehr garantiert. Dies gilt für jede einzelne Zelle. Bei geschickter Programmierung (z.B. Ring-Puffer), bei der die zu beschreibenden Zellen regelmäßig gewechselt werden, kann man eine deutlich höhere Anzahl an Schreibzugriffen, bezogen auf den Gesamtspeicher, erreichen.

Schreib- und Lesezugriffe auf den EEPROM-Speicher erfolgen über die im Modul `eeprom.h` definierten Funktionen. Mit diesen Funktionen können einzelne Bytes, Datenworte (16bit) und Datenblöcke geschrieben und gelesen werden.

Bei Nutzung des EEPROMs ist zu beachten, dass vor dem Zugriff auf diesen Speicher abgefragt wird, ob der Controller die vorherige EEPROM-Operation abgeschlossen hat. Die `avr-libc`-Funktionen beinhalten diese Prüfung, man muss sie nicht selbst implementieren. Man sollte auch verhindern, dass der Zugriff durch die Abarbeitung einer Interrupt-Routine unterbrochen wird, da bestimmte Befehlsabfolgen vorgegeben sind, die innerhalb weniger Taktzyklen aufeinanderfolgen müssen ("timed sequence"). Auch dies muss bei Nutzung der Funktionen aus der `avr-libc/eeprom.h`-Datei nicht selbst implementiert werden. Innerhalb der Funktionen werden Interrupts vor der "EEPROM-Sequenz" global deaktiviert und im Anschluss, falls vorher auch schon eingeschaltet, wieder aktiviert.

Bei der Deklaration einer Variable im EEPROM, ist das Attribut für die Section `".eeprom"` zu ergänzen. Siehe dazu folgendes Beispiel:

```
#include <avr/io.h>
#include <avr/eeprom.h>
#include <avr/interrupt.h>
#include <inttypes.h> // wird in aktuellen Versionen der avr-lib mit xx.h eingebunden

// EEMEM wird bei aktuellen Versionen der avr-lib in eeprom.h definiert
// hier: definiere falls noch nicht bekannt ("alte" avr-libc)
#ifndef EEMEM
// alle Textstellen EEMEM im Quellcode durch __attribute__ ... ersetzen
#define EEMEM __attribute__((section(".eeprom")))
#endif

//...

/* Byte */
uint8_t eeFooByte EEMEM = 123;

/* Wort */
uint16_t eeFooWord EEMEM = 12345;

/* float */
float eeFooFloat EEMEM;

/* Byte-Feld */
uint8_t eeFooByteArray1[] EEMEM = { 18, 3, 70 };
uint8_t eeFooByteArray2[] EEMEM = { 30, 7, 79 };

/* 16-bit unsigned short feld */
uint16_t eeFooWordArray1[4] EEMEM;
//...
```

Bytes lesen/schreiben

Die `avr-libc` Funktion zum Lesen eines Bytes heißt `eeprom_read_byte`. Parameter ist die Adresse des Bytes im EEPROM. Geschrieben wird über die Funktion `eeprom_write_byte` mit den Parametern Adresse und Inhalt. Anwendungsbeispiel:

```
//...
uint8_t myByte;

myByte = eeprom_read_byte(&eeFooByte); // lesen
// myByte hat nun den Wert 123
//...
```

```

myByte = 99;
eeprom_write_byte(&eeFooByte, myByte); // schreiben
// der Wert 99 wird im EEPROM an die Adresse der
// 'Variablen' eeFooByte geschrieben
//...
myByte = eeprom_read_byte(&eeFooByteArray1[1]);
// myByte hat nun den Wert 3
//...

// Beispiel zur "Sicherung" gegen leeres EEPROM nach "Chip Erase"
// (z.B. wenn die .eep-Datei nach Programmierung einer neuen Version
// des Programms nicht in den EEPROM uebertragen wurde und EESAVE
// deaktiviert ist (unprogrammed/1)
//
// Vorsicht: wenn EESAVE "programmed" ist, hilft diese Sicherung nicht
// weiter, da die Speicheradressen in einem neuen/erweiterten Programm
// moeglicherweise verschoben wurden. An der Stelle &eeFooByte steht
// dann u.U. der Wert einer anderen Variable aus einer "alten" Version.

#define EEPROM_DEF 0xFF
uint8_t fooByteDefault = 222;
if ( ( myByte = eeprom_read_byte(&eeFooByte) ) == EEPROM_DEF ) {
    myByte = fooByteDefault;
}

```

Wort lesen/schreiben

Schreiben und Lesen von Datenworten erfolgt analog zur Vorgehensweise bei Bytes:

```

//...
uint16_t myWord;

myWord = eeprom_read_word(&eeFooWord); // lesen
// myWord hat nun den Wert 12345
//...
myWord = 2222;
eeprom_write_word(&eeFooWord, myWord); // schreiben
//...

```

Block lesen/schreiben

Lesen und Schreiben von Datenblöcken erfolgt über die Funktionen `eeprom_read_block()` bzw. `eeprom_write_block()`. Die Funktionen erwarten drei Parameter: die Adresse der Quell- bzw. Zieldaten im RAM, die EEPROM-Adresse und die Länge des Datenblocks in Bytes (`size_t`).

TODO: Vorsicht! die folgenden Beispiele sind noch nicht geprüft, erstmal nur als Hinweis auf "das Prinzip". Evtl. fehlen "casts" und möglicherweise noch mehr.

```

//...
uint8_t myByteBuffer[3];
uint16_t myWordBuffer[4];

/* Datenblock aus EEPROM LESEN */

/* liest 3 Bytes ab der von eeFooByteArray1 definierten EEPROM-Adresse
in das RAM-Array myByteBuffer */
eeprom_read_block(myByteBuffer, eeFooByteArray1, 3);

/* dito etwas anschaulicher aber "unnütze Tipparbeit": */
eeprom_read_block(&myByteBuffer[0], &eeFooByteArray1[0], 3);

/* dito mit etwas Absicherung betr. der Länge */
eeprom_read_block(myByteBuffer, eeFooByteArray1, sizeof(myByteBuffer));

/* und nun mit "16bit" */
eeprom_read_block(myWordBuffer, eeFooWordArray1, sizeof(myWordBuffer));

/* Datenblock in EEPROM SCHREIBEN */
eeprom_write_block(myByteBuffer, eeFooByteArray1, sizeof(myByteBuffer));
eeprom_write_block(myWordBuffer, eeFooWordArray1, sizeof(myWordBuffer));
//...

```

"Nicht-Integer"-Datentypen wie z.B. Fließkommazahlen lassen sich recht praktisch über eine `union` in "Byte-Arrays" konvertieren und wieder "zurückwandeln". Dies erweist sich hier (aber nicht nur hier) als nützlich.

```

//...
float myFloat = 12.34;

union {
    float r;
    uint8_t i[sizeof(float)];
}

```

```

} u;

u.r = myFloat;

/* float in EEPROM */
eeprom_write_block(&(u.i), &eeFooFloat, sizeof(float));

/* float aus EEPROM */
eeprom_read_block(&(u.i), &eeFooFloat, sizeof(float));
/* u.r wieder 12.34 */

//...

```

Auch zusammengesetzte Typen lassen sich mit den Block-Routinen verarbeiten.

```

//...
typedef struct {
    uint8_t  label[8];
    uint8_t  rom_code[8];
} tMyStruct;

#define MAXSENSORS 3
tMyStruct eeMyStruct[MAXSENSORS] EEMEM;

//...

void egal(void)
{
    tMyStruct work;

    strcpy(work.label, "Flur");
    GetRomCode(work.rom_code);    // Dummy zur Veranschaulichung – setzt rom-code

    /* Sichern von "work" im EEPROM */
    eeprom_write_block(&work, &eeMyStruct[0], sizeof(tMyStruct)); // f. Index 0
    strcpy(work.label, "Bad");
    GetRomCode(work.rom_code);
    eeprom_write_block(&work, &eeMyStruct[1], sizeof(tMyStruct)); // f. Index 1
//...
    /* Lesen der Daten EEPROM Index 0 in "work" */
    eeprom_read_block(&work, &eeMyStruct[0], sizeof(tMyStruct));
    // work.label hat nun den Inhalt "Flur"
//...
}
//...

```

EEPROM-Speicherabbild in .eep-Datei

Mit den zum Compiler gehörenden Werkzeugen kann der aus den Variablendeklarationen abgeleitete EEPROM-Inhalt in eine Datei geschrieben werden (übliche Dateiendung: .eep, Daten im Intel Hex-Format). Damit können recht elegant Standardwerte für den EEPROM-Inhalt im Quellcode definiert werden. Makefiles nach WinAVR/MFile-Vorlage enthalten bereits die notwendigen Einstellungen (siehe dazu die Erläuterungen im Abschnitt Exkurs: Makefiles). Der Inhalt der eep-Datei muss ebenfalls zum Mikrocontroller übertragen werden (Write EEPROM), wenn die Initialisierungswerte aus der Deklaration vom Programm erwartet werden. Ansonsten enthält der EEPROM-Speicher nach der Übertragung des Programms mittels ISP abhängig von der Einstellung der EESAVE-Fuse (vgl. Datenblatt Abschnitt Fuse Bits) die vorherigen Daten (EESAVE programmed = 0), deren Position möglicherweise nicht mehr mit der Belegung im aktuellen Programm übereinstimmt oder den Standardwert nach "Chip Erase": 0xFF (EESAVE unprogrammed = 1). Als Sicherung kann man im Programm nochmals die Standardwerte vorhalten, beim Lesen auf 0xFF prüfen und gegebenenfalls einen Standardwert nutzen.

EEPROM-Variable auf feste Adressen legen

Gleich zu Beginn möchte ich darauf hinweisen, dass dieses Verfahren nur ein Workaround ist, mit dem man das Problem der anscheinend "zufälligen" Verteilung der EEPROM-Variablen durch den Compiler etwas in den Griff bekommen kann.

Hilfreich kann dies vor allem dann sein, wenn man z.B. über einen Kommandointerpreter (o.ä. Funktionen) direkt bestimmte EEPROM-Adressen manipulieren möchte. Auch wenn man über einen JTAG-Adapter (mk I oder mkII) den Programmablauf manipulieren möchte, indem man die EEPROM-Werte direkt ändert, kann diese Technik hilfreich sein.

Im folgenden nun zwei Sourcelistings mit einem Beispiel:

```

////////////////////////////////////
// Datei "eeprom.h" eines eigenen Projektes
////////////////////////////////////

#include <avr/eeprom.h>    // Die EEPROM-Definitionen/Macros der avr-libc einbinden

#define  EESIZE  512      // Maximale Größe des EEPROMS

#define  EE_DUMMY  0x000 // Dummyelement (Adresse 0 sollte nicht genutzt werden)
#define  EE_VALUE1 0x001 // Eine Bytevariable
#define  EE_WORD1L  0x002 // Eine Wordvariable (Lowbyte)

```

```
#define EE_WORD1H 0x003 // Eine Wordvariable (Highbyte)
#define EE_VALUE2 0x004 // Eine weitere Bytevariable
```

Mit den Macros **#define EE_VALUE1** legt man den Namen und die Adresse der 'Variablen' fest.

WICHTIG:Die Adressen sollten fortlaufend, zumindest aber aufsteigend sortiert sein! Ansonsten besteht die Gefahr, daß man sehr schnell ein Durcheinander im EEPROM Speicher veranstaltet.

WICHTIG:Für den Compiler sind das lediglich Speicher-Adressen, über die auf das EEPROM zugegriffen wird. Der Compiler sieht nichts davon als eine echte Variable an und stößt sich daher auch nicht daran, wenn 2 Makros mit der gleichen Speicheradresse, bzw. überlappenden Speicherbereichen definiert werden. Es liegt einzig und alleine in der Hand des Programmierers, hier keinen Fehler zu machen.

```
////////////////////////////////////
// Datei "eeprom.c" eines eigenen Projektes
////////////////////////////////////

#include "eeprom.h" // Eigene EEPROM-Headerdatei einbinden

uint8_t ee_mem[EESIZE] EEMEM =
{
    [EE_DUMMY] = 0x00,
    [EE_VALUE1] = 0x05,
    [EE_WORD1L] = 0x01,
    [EE_WORD1H] = 0x00,
    [EE_VALUE2] = 0xFF
};
```

Durch die Verwendung eines Array, welches das gesamte EEPROM umfasst, bleibt dem Compiler nicht anderes übrig, als das Array so zu platzieren, dass Element 0 des Arrays der Adresse 0 des EEPROMs entspricht. (*Ich hoffe nur, dass die Compilerbauer daran nichts ändern!*)

Wie man in dem obigen Codelisting auch sehen kann, hat das Verfahren einen kleinen Haken. Variablen die größer sind als 1 Byte, müssen etwas umständlicher definiert werden. Benötigt man keine Initialisierung durch das Programm (was der Normalfall sein dürfte), dann kann man das auch so machen:

Möchte man im EEPROM hintereinander beispielsweise Variablen, mit den Namen **Wert**, **Anzahl**, **Name** und **Wertigkeit** definieren, wobei Wert und Wertigkeit 1 Byte belegen sollen, Anzahl als 1 Wort (also 2 Bytes) und Name mit 10 Bytes reserviert werden soll, so geht auch folgendes:

```
#define EE_DUMMY 0x000
#define EE_WERT ( 0x000 + sizeof( uint8_t ) )
#define EE_ANZAHL ( EE_WERT + sizeof( uint8_t ) )
#define EE_NAME ( EE_ANZAHL + sizeof( uint16_t ) )
#define EE_WERTIGKEIT ( EE_NAME + 10 * sizeof( uint8_t ) )
#define EE_LAST ( EE_WERTIGKEIT + sizeof( uint8_t ) )
```

Jedes Makro definiert also seine Startadresse durch die Startadresse der unmittelbar vorhergehende 'Variablen' plus der Anzahl der Bytes die von der vorhergehenden 'Variablen' verbraucht werden. Dadurch ist man zumindest etwas auf der sicheren Seite, dass keine 2 'Variablen' im EEPROM überlappend definiert werden. Möchte man eine weitere 'Variable' hinzufügen, so wird deren Name, einfach anstelle der EE_LAST eingesetzt und eine neue Zeile für EE_LAST eingefügt, in der dann die Größe der 'Variablen' festgelegt wird. Zb.

```
#define EE_DUMMY 0x000
#define EE_WERT ( 0x000 + sizeof( uint8_t ) )
#define EE_ANZAHL ( EE_WERT + sizeof( uint8_t ) )
#define EE_NAME ( EE_ANZAHL + sizeof( uint16_t ) )
#define EE_WERTIGKEIT ( EE_NAME + 10 * sizeof( uint8_t ) )
#define EE_PROZENT ( EE_WERTIGKEIT + sizeof( uint8_t ) )
#define EE_LAST ( EE_PROZENT + sizeof( double ) )
```

EE_PROZENT legt die Startadresse für eine neue 'Variable' des Datentyps double fest.

Der Zugriff auf die EEPROM Werte kann dann z.B.so erfolgen:

```
uint8_t temp1;
uint16_t temp2;

temp1 = eeprom_read_byte(EE_VALUE1);
temp2 = eeprom_read_word(EE_WORD1L);
```

Ob die in der avr-libc vorhandenen Funktionen dafür verwendet werden können, weiß ich nicht. Aber in einigen Fällen muss man sich sowieso eigene Funktionen bauen, welche die spezifischen Anforderungen (Interrupt - Atom Problem, etc.) erfüllen.

Die oben beschriebene Möglichkeit ist nur eine Möglichkeit, wie man dies realisieren kann. Sie bietet einem eine relativ einfache Art die EEPROM-Werte auf beliebige Adressen zu legen oder Adressen zu ändern. Die Andere Möglichkeit besteht darin, die EEPROM-Werte wie folgt zu belegen:

```
////////////////////////////////////
```

```
// Datei "eeprom.c" eines eigenen Projektes
///////////////////////////////////////////////////////////////////

#include "eeprom.h" // Eigene EEPROM-Headerdatei einbinden

uint8_t ee_mem[EESIZE] EEMEM =
{
    0x00, // ee_dummy
    0x05, // ee_value1
    0x01, // ee_word1L
    0x00, // (ee_word1H)
    0xFF // ee_value2
};
```

Hierbei kann man Variablen, die größer sind als 1 Byte einfacher definieren und man muss nur die Highbyte- oder Lowbyte-Adresse in der "eeprom.h" definieren. Allerdings muss man hier höllisch aufpassen, dass man nicht um eine oder mehrere Positionen verrutscht!

Welche der beiden Möglichkeiten man einsetzt, hängt vor allem davon ab, wieviele Byte, Word und sonstige Variablen man benutzt. Gewöhnen sollte man sich an beide Varianten können :)

Kleine Schlussbemerkung:

- Der avr-gcc unterstützt die Variante 1 und die Variante 2
- Der icc-avr Compiler unterstützt nur die Variante 2!

Bekannte Probleme bei den EEPROM-Funktionen

Vorsicht: Bei alten Versionen der avr-libc wurden nicht alle AVR Controller unterstützt. Z.B. bei der avr-libc Version 1.2.3 insbesondere bei AVR's "der neuen Generation" (ATmega48/88/168/169) funktionieren die Funktionen nicht korrekt (Ursache: unterschiedliche Speicheradressen der EEPROM-Register). In neueren Versionen (z.B. avr-libc 1.4.3 aus WinAVR 20050125) wurde die Zahl der unterstützten Controller deutlich erweitert und eine Methode zur leichten Anpassung an zukünftige Controller eingeführt.

In jedem Datenblatt zu AVR-Controllern mit EEPROM sind kurze Beispielercodes für den Schreib- und Lesezugriff enthalten. Will oder kann man nicht auf die neue Version aktualisieren, kann der dort gezeigte Code auch mit dem avr-gcc (ohne avr-libc/eeprom.h) genutzt werden ("copy/paste", gegebenfalls Schutz vor Unterbrechung/Interrupt ergänzen `uint8_t sreg; sreg=SREG; cli(); [EEPROM-Code]; SREG=sreg; return;`, siehe Abschnitt Interrupts). Im Zweifel hilft ein Blick in den vom Compiler erzeugten Assembler-Code (Ist/Iss-Dateien).

- siehe auch: [Dokumentation der avr-libc](#) Abschnitt Modules/EEPROM handling

EEPROM Register

Um das EEPROM anzusteuern sind drei Register von Bedeutung:

EEAR

Hier werden die Adressen eingetragen zum Schreiben oder Lesen. Dieses Register unterteilt sich nochmal in EEARH und EEARL da in einem 8 Bit Register keine 512 Adressen adressiert werden können

EEDR

Hier werden die Daten eingetragen die geschrieben werden sollen bzw. es enthält die gelesenen Daten

EECR

Ist das Kontrollregister für das EEPROM

Das EECR steuert den Zugriff auf das EEPROM und ist wie folgt aufgebaut:

Aufbau des EECR-Registers								
Bit	7	6	5	4	3	2	1	0
Name	-	-	-	-	EERIE	EEMWE	EWE	EERE
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W
Init Value	0	0	0	0	0	0	0	0

Bedeutung der Bits

Bit 4-7

nicht belegt

Bit 3 (EERIE)

EEPROM Ready Interrupt Enable: Wenn das Bit gesetzt ist und globale Interrupts erlaubt sind in Register SREG (Bit 7) wird ein Interrupt ausgelöst nach Beendigung des Schreibzyklus (EEPROM Ready Interrupt). Ist einer der beiden Bits 0 wird kein Interrupt ausgelöst.

Bit 2 (EEMWE)

EEPROM Master Write Enable: Dieses Bit bestimmt, daß wenn EWE = 1 gesetzt wird (innerhalb von 4 Taktzyklen), das EEPROM beschrieben wird mit den Daten in EEDR bei Adresse EEAR. Wenn EEMWE = 0 ist und EWE = 1 gesetzt wird hat das keine Auswirkungen. Der Schreibvorgang wird dann nicht ausgelöst. Nach 4 Taktzyklen wird das Bit EEMWE automatisch wieder auf 0 gesetzt. Dieses Bit löst den Schreibvorgang nicht aus, es dient sozusagen als Sicherungsbit für EWE.

Bit 1 (EWE)

EEPROM Write Enable: Dieses Bit löst den Schreibvorgang aus wenn es auf 1 gesetzt wird, sofern vorher EEMWE gesetzt wurde und seitdem nicht mehr als 4 Taktzyklen vergangen sind. Wenn der Schreibvorgang abgeschlossen ist wird dieses Bit automatisch wieder auf 0 gesetzt und sofern EERIE

gesetzt ist ein Interrupt ausgelöst. Ein Schreibvorgang sieht typischerweise wie folgt aus:

1. EEPROM Bereitschaft abwarten (EWE=0)
2. Adresse übergeben an EEAR
3. Daten übergeben an EEDR
4. Schreibvorgang auslösen in EECR mit Bit EEMWE=1 und EWE=1
5. (Optional) Warten bis Schreibvorgang abgeschlossen ist

Bei 0 EERE

EEPROM Read Enable: Wird dieses Bit auf 1 gesetzt wird das EEPROM an der Adresse in EEAR ausgelesen und die Daten in EEDR gespeichert. Das EEPROM kann nicht ausgelesen werden wenn bereits eine Schreiboperation gestartet wurde. Es ist daher zu empfehlen die Bereitschaft vorher zu prüfen. Das EEPROM ist lesebereit wenn das Bit EWE=0 ist. Ist der Lesevorgang abgeschlossen wird das Bit wieder auf 0 gesetzt und das EEPROM ist für neue Lese/Schreibbefehle wieder bereit. Ein typischer Lesevorgang kann wie folgt aufgebaut sein:

1. Bereitschaft zum lesen prüfen (EWE=0)
2. Adresse übergeben an EEAR
3. Lesezyklus auslösen mit EERE = 1
4. Warten bis Lesevorgang abgeschlossen EERE = 0
5. Daten abholen aus EEDR

Die Nutzung von sprintf und printf

Um komfortabel, d.h. formatiert, Ausgaben auf ein Display oder die serielle Schnittstelle zu tätigen, bieten sich **sprintf** oder **printf** an.

Alle *printf-Varianten sind jedoch ziemlich speicherintensiv und der Einsatz in einem Mikrocontroller mit knappem Speicher muss sorgsam abgewogen werden.

Bei **sprintf** wird die Ausgabe zunächst in einem Puffer vorbereitet und anschliessend mit einfachen Funktionen zeichenweise ausgegeben. Es liegt in der Verantwortung des Programmierers genügend Platz im Puffer für die erwarteten Zeichen bereitzuhalten.

```
#include <stdio.h>
#include <stdint.h>

// ...
// nicht dargestellt: Implementierung von uart_puts (vgl. Abschnitt UART)
// ...

uint16_t counter;

// Ausgabe eines unsigned Integerwertes
void uart_puti( uint16_t value )
{
    uint8_t puffer[20];

    sprintf( puffer, "Zählerstand: %u", value );
    uart_puts( puffer );
}

int main()
{
    counter = 5;

    uart_puti( counter );
    uart_puti( 42 );
}
```

Eine weitere elegante Möglichkeit besteht darin, den STREAM stdout (Standardausgabe) auf eine eigene Ausgabefunktion umzuleiten. Dazu wird dem Ausgabemechanismus der C-Bibliothek eine neue Ausgabefunktion bekannt gemacht, deren Aufgabe es ist, ein einzelnes Zeichen auszugeben. Wohin die Ausgabe dann tatsächlich stattfindet, ist Sache der Ausgabefunktion. Im Beispiel unten wird auf UART ausgegeben. Alle anderen, höheren Funktionen wie z.B. **printf** greifen letztendlich auf diese primitive Ausgabefunktion zurück.

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

void uart_init(void);

// a. Deklaration der primitiven Ausgabefunktion
int uart_putchar(char c, FILE *stream);

// b. Umleiten der Standardausgabe stdout (Teil 1)
static FILE mystdout = FDEV_SETUP_STREAM( uart_putchar, NULL, _FDEV_SETUP_WRITE );

// c. Definition der Ausgabefunktion
int uart_putchar( char c, FILE *stream )
{
    if( c == '\n' )
        uart_putchar( '\r', stream );

    loop_until_bit_is_set( UCSRA, UDRE );
}
```

```

    UDR = c;
    return 0;
}

void uart_init(void)
{
    /* hier µC spezifischen Code zur Initialisierung */
    /* des UART einfügen... s.o. im AVR-GCC-Tutorial */

    // Beispiel:
    //
    // myAVR Board 1.5 mit externem Quarz Q1 3,6864 MHz
    // 9600 Baud 8N1

#ifdef F_CPU
#define F_CPU 3686400
#endif
#define UART_BAUD_RATE 9600

// Hilfsmakro zur UBRR-Berechnung ("Formel" laut Datenblatt)
#define UART_UBRR_CALC(BAUD_,FREQ_) ((FREQ_)/((BAUD_)*16L)-1)

    UCSRB |= (1<<TXEN) | (1<<RXEN);    // UART TX und RX einschalten
    UCSRC |= (1<<URSEL)|(3<<UCSZ0);    // Asynchron 8N1

    UBRRH = (uint8_t)( UART_UBRR_CALC( UART_BAUD_RATE, F_CPU ) >> 8 );
    UBRRL = (uint8_t)UART_UBRR_CALC( UART_BAUD_RATE, F_CPU );
}

int main(void)
{
    int16_t antwort = 42;
    uart_init();

    // b. Umleiten der Standardausgabe stdout (Teil 2)
    stdout = &mystdout;

    // Anwendung
    printf( "Die Antwort ist %d.\n", antwort );
    return 0;
}

// Quelle: avr-libc-user-manual-1.4.3.pdf, S.74
//         + Ergänzungen

```

Sollen Fließkommazahlen ausgegeben werden, muss im Makefile eine andere (größere) Version der [printflib](#) eingebunden werden.

Assembler und Inline-Assembler

Gelegentlich erweist es sich als nützlich, C- und Assembler-Code in einer Anwendung zu nutzen. Typischerweise wird das Hauptprogramm in C verfasst und wenige, extrem zeitkritische oder hardwarenahe Operationen in Assembler.

Die "gnu-Toolchain" bietet dazu zwei Möglichkeiten:

- **Inline-Assembler:** Die Assembleranweisungen werden direkt in den C-Code integriert. Eine Quellcode-Datei enthält somit C- und Assembleranweisungen
- **Assembler-Dateien:** Der Assemblercode befindet sich in eigenen Quellcodedateien. Diese werden vom gnu-Assembler (avr-as) zu Object-Dateien assembliert ("compiliert") und mit den aus dem C-Code erstellten Object-Dateien zusammengebunden (gelinkt).

Inline-Assembler

Inline-Assembler bietet sich an, wenn nur wenig Assembleranweisungen benötigt werden. Typische Anwendung sind kurze Codesequenzen für zeitkritische Operationen in Interrupt-Routinen oder sehr präzise Warteschleifen (z.B. 1-Wire). Inline-Assembler wird mit **asm volatile** eingeleitet, die Assembler-Anweisungen werden in einer Zeichenkette zusammengefasst, die als "Parameter" übergeben wird. Durch Doppelpunkte getrennt werden die Ein- und Ausgaben sowie die "Clobber-Liste" angegeben.

Ein einfaches Beispiel für Inline-Assembler ist das Einfügen einer NOP-Anweisung (NOP steht für No Operation). Dieser Assembler-Befehl benötigt genau einen Taktzyklus, ansonsten "tut sich nichts". Sinnvolle Anwendungen für NOP sind genaue Delay(=Warte)-Funktionen.

```

...
/* Verzögern der weiteren Programmausführung um
   genau 3 Taktzyklen */
asm volatile ("nop");
asm volatile ("nop");
asm volatile ("nop");
...

```

Weiterhin kann mit einem NOP verhindert werden, dass leere Schleifen, die als Warteschleifen gedacht sind, wegoptimiert werden. Der Compiler erkennt ansonsten die vermeintlich nutzlose Schleife und erzeugt dafür keinen Code im ausführbaren Programm.

```

...
uint16_t i;

/* leere Schleife - wird bei eingeschalteter Compiler-Optimierung wegoptimiert */
for (i = 0; i < 1000; i++)
;

...

/* Schleife erzwingen (keine Optimierung): "NOP-Methode" */
for (i = 0; i < 1000; i++)
asm volatile("NOP");

...

/* alternative Methode (keine Optimierung): */
volatile uint16_t j;
for (j = 0; j < 1000; j++)
;

```

Ein weiterer nützlicher "Assembler-Einzeiler" ist der Aufruf von `sleep` (`asm volatile ("sleep");`); da hierzu in älteren Versionen der `avr-libc` keine eigene Funktion existiert (in neueren Versionen `sleep_cpu()` aus `sleep.h`).

Als Beispiel für mehrzeiligen Inline-Assembler eine präzise Delay-Funktion. Die Funktion erhält ein 16-bit Wort als Parameter, prüft den Parameter auf 0 und beendet die Funktion in diesem Fall oder durchläuft die folgende Schleife sooft wie im Wert des Parameters angegeben. Inline-Assembler hat hier den Vorteil, dass die Laufzeit unabhängig von der Optimierungsstufe (Parameter `-O`, vgl. `makefile`) und der Compiler-Version ist.

```

static inline void delayloop16 (uint16_t count)
{
asm volatile ("cp %A0, __zero_reg__ \n\t"
             "cpc %B0, __zero_reg__ \n\t"
             "breq 2f \n\t"
             "1: \n\t"
             "sbiw %0,1 \n\t"
             "brne 1b \n\t"
             "2: \n\t"
             : "=w" (count)
             : "0" (count)
);
}

```

- Jede Anweisung wird mit `\n\t` abgeschlossen. Der Zeilenumbruch teilt dem Assembler mit, dass ein neuer Befehl beginnt.
- Als Sprung-Marken (Labels) werden Ziffern verwendet. Diese speziellen Labels sind mehrfach im Code verwendbar. Gesprungen wird jeweils zurück (b) oder vorwärts (f) zum nächsten auffindbaren Label.

Das Resultat zeigt ein Blick in die Assembler-Datei, die der Compiler mit der option `-save-temps` nicht löscht:

```

cp r24, __zero_reg__ ; count
cpc r25, __zero_reg__ ; count
breq 2f
1:
sbiw r24,1 ; count
brne 1b
2:

```

Detaillierte Ausführungen zum Thema Inline-Assembler finden sich in der Dokumentation der `avr-libc` im Abschnitt [Related Pages/Inline Asm](#).

Siehe auch:

- [AVR Assembler-Anweisungsliste](#)
- [Deutsche Einführung in Inline-Assembler](#)

Assembler-Dateien

Assembler-Dateien erhalten die Endung `.S` (*grosses S*) und werden im `makefile` nach `WinAVR/mfile`-Vorlage hinter `ASRC=` durch Leerzeichen getrennt aufgelistet.

Wenn man mit dem AVR Studio arbeitet, kann alternativ auch das standardmäßig erstellte Makefile bearbeitet und folgende Zeilen eingefügt werden:

```

## Objects that must be built in order to link
OBJECTS = (alte Dateien...) useful.o

## Compile
## Hier folgt eine Liste der gelinkten Dateien, darunter einfügen:
useful.o: ../useful.S
$(CC) $(INCLUDES) $(ASMFLAGS) -c $<

```

Das war es schon. Allerdings gilt es zu beachten, dass das makefile über "Project -> Configuration options" selbst einzubinden ist, sonst wird es natürlich wieder überschrieben.

Im Beispiel eine Funktion *superFunc*, die alle Pins des Ports D auf "Ausgang" schaltet, eine Funktion *ultraFunc*, die die Ausgänge entsprechend des übergebenen Parameters schaltet, eine Funktion *gigaFunc*, die den Status von Port A zurückgibt und eine Funktion *addFunc*, die zwei Bytes zu einem 16-bit-Wort addiert. Die Zuweisungen im C-Code (PORTx = ...) verhindern, dass der Compiler die Aufrufe wegoptimiert und dienen nur zur Veranschaulichung der Parameterübergaben.

Zuerst der Assembler-Code. Der Dateiname sei useful.S:

```
#include "avr/io.h"

//; Arbeitsregister (ohne "r")
workreg = 16
workreg2 = 17

//; Konstante:
ALLOUT = 0xff

//; ** Setze alle Pins von PortD auf Ausgang **
//; keine Parameter, keine Rückgabe
.global superFunc
.func superFunc
superFunc:
    push workreg
    ldi workreg, ALLOUT
    out _SFR_IO_ADDR(DDRD), workreg // beachte: _SFR_IO_ADDR()
    pop workreg
    ret
.endfunc

//; ** Setze PORTD auf übergebenen Wert **
//; Parameter in r24 (LSB immer bei "graden" Nummern)
.global ultraFunc
.func ultraFunc
ultraFunc:
    out _SFR_IO_ADDR(PORTD), 24
    ret
.endfunc

//; ** Zustand von PINA zurückgeben **
//; Rückgabewerte in r24:r25 (LSB:MSB), hier nur LSB genutzt
.global gigaFunc
.func gigaFunc
gigaFunc:
    in 24, _SFR_IO_ADDR(PINA)
    ret
.endfunc

//; ** Zwei Bytes addieren und 16-bit-Wort zurückgeben **
//; Parameter in r24 (Summand1) und r22 (Summand2) -
//; Parameter sind Word-"aligned" d.h. LSB immer auf "graden"
//; Registernummern. Bei 8-Bit und 16-Bit Paramtern somit
//; beginnend bei r24 dann r22 dann r20 etc.
//; Rückgabewert in r24:r25
.global addFunc
.func addFunc
addFunc:
    push workreg
    push workreg2
    clr workreg2
    mov workreg, 22
    add workreg, 24
    adc workreg2, 1 // r1 - assumed to be always zero ...
    movw r24, workreg
    pop workreg2
    pop workreg
    ret
.endfunc

//; oh je - sorry - Mein AVR-Assembler ist eingerostet, hoffe das stimmt so...

.end
```

Im Makefile ist der Name der Assembler-Quellcodedatei einzutragen:

```
ASRC = useful.S
```

Der Aufruf erfolgt dann im C-Code so:

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

```

extern void superFunc(void);
extern void ultraFunc(uint8_t setVal);
extern uint8_t gigaFunc(void);
extern uint16_t addFunc(uint8_t w1, uint8_t w2);

int main(void)
{
[...]
    superFunc();

    ultraFunc(0x55);

    PORTD = gigaFunc();

    PORTA = (addFunc(0xF0, 0x11) & 0xff);
    PORTB = (addFunc(0xF0, 0x11) >> 8);
[...]}

```

Das Ergebnis wird wieder in der lss-Datei ersichtlich:

```

[...]
    superFunc();
148: 0e 94 f6 00    call    0x1ec

    ultraFunc(0x55);
14c: 85 e5          ldi    r24, 0x55        ; 85
14e: 0e 94 fb 00    call    0x1f6

    PORTD = gigaFunc();
152: 0e 94 fd 00    call    0x1fa
156: 82 bb          out    0x12, r24        ; 18

    PORTA = (addFunc(0xF0, 0x11) & 0xff);
158: 61 e1          ldi    r22, 0x11        ; 17
15a: 80 ef          ldi    r24, 0xF0        ; 240
15c: 0e 94 ff 00    call    0x1fe
160: 8b bb          out    0x1b, r24        ; 27
    PORTB = (addFunc(0xF0, 0x11) >> 8);
162: 61 e1          ldi    r22, 0x11        ; 17
164: 80 ef          ldi    r24, 0xF0        ; 240
166: 0e 94 fc 00    call    0x1f8
16a: 89 2f          mov    r24, r25
16c: 99 27          eor    r25, r25
16e: 88 bb          out    0x18, r24        ; 24

[...]
000001ec <superFunc>:
// setze alle Pins von PortD auf Ausgang
.global superFunc
.func superFunc
superFunc:
    push workreg
1ec: 0f 93          push   r16
        ldi workreg, ALLOUT
1ee: 0f ef          ldi    r16, 0xFF        ; 255
        out _SFR_IO_ADDR(DDRD), workreg
1f0: 01 bb          out    0x11, r16        ; 17
    pop workreg
1f2: 0f 91          pop    r16
    ret
1f4: 08 95          ret

000001f6 <ultraFunc>:
.endfunc

// setze PORTD auf übergebenen Wert
.global ultraFunc
.func ultraFunc
ultraFunc:
    out _SFR_IO_ADDR(PORTD), 24
1f6: 82 bb          out    0x12, r24        ; 18
    ret
1f8: 08 95          ret

000001fa <gigaFunc>:
.endfunc

// Zustand von PINA zurückgeben
.global gigaFunc
.func gigaFunc
gigaFunc:
    in 24, _SFR_IO_ADDR(PINA)
1fa: 89 b3          in     r24, 0x19        ; 25
    ret
1fc: 08 95          ret

```

```

000001fe <addFunc>:
.endfunc

// zwei Bytes addieren und 16-bit-Wort zurückgeben
.global addFunc
.func addFunc
addFunc:
    push workreg
1fe:  0f 93          push    r16
    push workreg2
200:  1f 93          push    r17
    clr workreg2
202:  11 27          eor     r17, r17
    mov workreg, 22
204:  06 2f          mov     r16, r22
    add workreg, 24
206:  08 0f          add     r16, r24
    adc workreg2, 1 // r1 - assumed to be always zero ...
208:  11 1d          adc     r17, r1
    movw r24, workreg
20a:  c8 01          movw   r24, r16
    pop workreg2
20c:  1f 91          pop     r17
    pop workreg
20e:  0f 91          pop     r16
    ret
210:  08 95          ret

[...]

```

Die Zuweisung von Registern zu Parameternummer und die Register für die Rückgabewerte sind in den "Register Usage Guidelines" der avr-libc-Dokumentation erläutert.

Siehe auch:

- [avr-libc-Dokumentation: Related Pages/avr-libc and assembler programs](#)
- [avr-libc-Dokumentation: Related Pages/FAQ/"What registers are used by the C compiler?"](#)

Globale Variablen für Datenaustausch

Oftmals kommt man um globale Variablen nicht herum, z.B. um den Datenaustausch zwischen Hauptprogramm und Interrupt-Routinen zu realisieren. Hierzu muss man im Assembler wissen, wo genau die Variable vom C-Compiler abgespeichert wird.

Hierzu muss die Variable, hier "zaehler" genannt, zuerst im C-Code als Global definiert werden, z.B. so:

```

#include <avr/io.h>

volatile uint8_t zaehler;

int16_t main (void)
{
    // irgendein Code, in dem zaehler benutzt werden kann
}

```

Im folgenden Assembler-Beispiel wird der Externe Interrupt0 verwendet, um den Zähler hochzuzählen. Es fehlen die Initialisierungen des Interrupts und die Interrupt-Freigabe, so richtig sinnvoll ist der Code auch nicht, aber er zeigt (hoffentlich) wie es geht.

Im Umgang mit Interrupt-Vektoren gilt beim GCC-Assembler das Gleiche, wie bei C: Man muss die exakte Schreibweise beachten, ansonsten wird nicht der Interrupt-Vektor angelegt, sondern eine neue Funktion - und man wundert sich, dass nichts funktioniert (vgl. das AVR-GCC-Handbuch).

```

#include "avr/io.h"

temp = 16

.extern zaehler

.global INT0_vect
INT0_vect:

    push temp                //; wichtig: Benutzte Register und das
    in temp, _SFR_IO_ADDR(SREG) //; Status-Register (SREG) sichern!
    push temp

    lds temp, zaehler        //; Wert aus dem Speicher lesen
    inc temp                 //; bearbeiten
    sts zaehler, temp        //; und wieder zurückschreiben

    pop temp                 //; die benutzten Register wiederherstellen
    out _SFR_IO_ADDR(SREG), temp
    pop temp

```

```
    reti
.end
```

Globale Variablen im Assemblerfile anlegen

Alternativ können Variablen aber auch im Assemblerfile angelegt werden. Dadurch kann auf eine .c-Datei verzichtet werden. Für das obige Beispiel könnte der Quelltext dann die Dateien zaehl_asm.S und zaehl_asm.h abgelegt werden, so dass nur noch zaehl_asm.S mit kompiliert werden müsste.

Anstatt im Assemblerfile über das Schlüsselwort *.extern* auf eine vorhandene Variable zu verweisen, wird dazu mit dem Schlüsselwort *.comm* die benötigte Anzahl von Bytes für eine Variable reserviert.

zaehl_asm.S

```
#include "avr/io.h"
temp = 16

//; 1 Byte im RAM für den Zähler reservieren
.comm zaehler, 1

.global INT0_vect
INT0_vect:

...
```

In der Headerdatei wird dann auf die Variable nur noch verwiesen (Schlüsselwort *extern*):

zaehl_asm.h

```
#ifndef ZAEHL_ASM_H
#define ZAEHL_ASM_H

extern volatile uint8_t zaehler;

#endif
```

Im Gegensatz zu globalen Variablen in C werden so angelegte Variablen nicht automatisch mit dem Wert 0 initialisiert.

Variablen größer als 1 Byte

Variablen, die größer als **ein** Byte sind, können in Assembler auf ähnliche Art angesprochen werden. Hierzu müssen nur genug Bytes angefordert werden, um die Variable aufzunehmen. Soll z.B. für den Zähler eine Variable vom Typ *unsigned long*, also *uint32_t* verwendet werden, so müssen 4 Bytes reserviert werden:

```
...
// 4 Byte im RAM für den Zähler reservieren
.comm zaehler, 4
...
```

Die dazugehörige Deklaration im Headerfile wäre dann:

```
...
extern volatile uint32_t zaehler;
...
```

Bei Variablen, die größer als ein Byte sind, werden die Werte beginnend mit dem niederwertigsten Byte im RAM abgelegt. Das folgende Codeschnippel zeigt, wie unter Assembler auf die einzelnen Bytes zugegriffen werden kann. Dazu wird im Interrupt nun ein 32-Bit Zähler erhöht:

```
#include "avr/io.h"
temp = 16

// 4 Byte im RAM für den Zähler reservieren
.comm zaehler, 4

.global INT0_vect
INT0_vect:

    push temp
    in temp, _SFR_IO_ADDR(SREG) // wichtig: Benutzte Register und das
    push temp // Status-Register (SREG) sichern !
```

```

// 32-Bit-Zähler incrementieren
lds temp, (zaehler + 0) // 0. Byte (niederwertigstes Byte)
inc temp
sts (zaehler + 0), temp
brne RAUS

lds temp, (zaehler + 1) // 1. Byte
inc temp
sts (zaehler + 1), temp
brne RAUS

lds temp, (zaehler + 2) // 2. Byte
inc temp
sts (zaehler + 2), temp
brne RAUS

lds temp, (zaehler + 3) // 3. Byte (höchstwertigstes Byte)
inc temp
sts (zaehler + 3), temp
brne RAUS

RAUS:
pop temp // die benutzten Register wiederherstellen
out _SFR_IO_ADDR(SREG), temp
pop temp
reti

.end

```

TODO: 16-Bit / 32-Bit Variablen, Zugriff auf Arrays (Strings)

Anhang

Besonderheiten bei der Anpassung bestehenden Quellcodes

Einige Funktionen aus früheren Versionen der avr-libc werden inzwischen als veraltet angesehen. Sie sind nicht mehr vorhanden oder als *deprecated* (missbilligt) ausgewiesen und Definitionen in `<compat/deprecated.h>` verschoben. Es empfiehlt sich, vorhandenen Code zu portieren und die alten Funktionen nicht mehr zu nutzen, auch wenn diese noch zur Verfügung stehen.

Veraltete Funktionen zur Deklaration von Interrupt-Routinen

Die Funktionen (eigentlich Makros) *SIGNAL* und *INTERRUPT* zur Deklaration von Interruptroutinen sollten nicht mehr genutzt werden.

In aktuellen Versionen der avr-libc (z.B. avr-libc 1.4.3 aus WinAVR 20060125) werden Interruptroutinen, die **nicht** durch andere Interrupts **unterbrechbar** sind, mit `ISR` deklariert (siehe Abschnitt im Hauptteil). Auch die Benennung wurden vereinheitlicht und an die üblichen Bezeichnungen in den AVR Datenblättern angepasst. In der Dokumentation der avr-libc sind alte und neue Bezeichnungen in der Tabelle gegenübergestellt. Die erforderlichen Schritte zur Portierung:

- `#include` von `avr/signal.h` entfernen
- `SIGNAL` durch `ISR` ersetzen
- Name des Interrupt-Vektors anpassen (`SIG_*` durch entsprechendes `*_vect`)

Als Beispiel für die Anpassung zuerst ein "alter" Code:

```

#include <avr/interrupt.h>
#include <avr/signal.h>

...

/* Timer2 Output Compare bei einem ATmega8 */
SIGNAL(SIG_OUTPUT_COMPARE2)
{
    ...
}

```

Im Datenblatt wird der Vektor mit `TIMER2 COMP` bezeichnet. Die Bezeichnung in der avr-libc entspricht dem Namen im Datenblatt, Leerzeichen werden durch Unterstriche (`_`) ersetzt und ein `_vect` angehängt.

Der neue Code sieht dann so aus:

```

#include <avr/interrupt.h>
/* signal.h entfällt */

ISR(TIMER2_COMP_vect)
{
    ...
}

```

Bei Unklarheiten bezüglich der neuen Vektorlabels hilft (noch) ein Blick in die Headerdatei des entsprechenden Controllers. Für das vorherige Beispiel also der Blick in die Datei `iom8.h` für den ATmega8, dort findet man die veraltete Bezeichnung unterhalb der aktuellen.

```
...
/* $Id: iom8.h,v 1.13 2005/10/30 22:11:23 joerg_wunsch Exp $ */
/* avr/iom8.h - definitions for ATmega8 */

...

/* Timer/Counter2 Compare Match */
#define TIMER2_COMP_vect      _VECTOR(3)
#define SIG_OUTPUT_COMPARE2  _VECTOR(3)
...

```

Für **unterbrechbare** Interruptroutinen, die mittels `INTERRUPT` deklariert sind, gibt es keinen direkten Ersatz in Form eines Makros. Solche Routinen sind laut Dokumentation der `avr-libc` in folgender Form zu deklarieren:

```
void XXX_vect(void) __attribute__((interrupt));
void XXX_vect(void) {
    ...
}
```

Beispiel:

```
/* ** alt ** */
#include <avr/io.h>
#include <avr/interrupt.h>

//...

INTERRUPT(SIG_OVERFLOW0)
{
    ...
}

/* ** neu: ** */
#include <avr/io.h>

//...

void TIMER0_OVF_vect(void) __attribute__((interrupt));
void TIMER0_OVF_vect(void)
{
    ...
}
```

Will oder kann man den Code nicht portieren, ist zur weiteren Verwendung von `INTERRUPT` die Header-Datei `compat/deprecated.h` einzubinden. Man sollte bei dieser Gelegenheit jedoch nochmals überprüfen, ob die Funktionalität von `INTERRUPT` tatsächlich gewünscht ist. In vielen Fällen wurde `INTERRUPT` dort genutzt, wo eigentlich `SIGNAL` (nunmehr `ISR`) hätte genutzt werden sollen.

Veraltete Funktionen zum Portzugriff

`inp` und `outp` zum Einlesen bzw. Schreiben von Registern sind nicht mehr erforderlich, der Compiler unterstützt dies ohne diesen Umweg.

```
unsigned char i, j;

// alt:
i = inp(PINA);
j = 0xff;
outp(PORTB, j);

// neu (nicht mehr wirklich neu...):
i = PINA;
j = 0xff;
PORTB = j;
```

Will oder kann man den Code nicht portieren, ist zur weiteren Verwendung von `inp` und `outp` die Header-Datei `compat/deprecated.h` einzubinden.

Veraltete Funktionen zum Zugriff auf Bits in Registern

`cbi` und `sbi` zum Löschen und Setzen von Bits sind nicht mehr erforderlich, der Compiler unterstützt dies ohne diesen Umweg. Die Bezeichnung ist ohnehin irreführend da die Funktionen nur für Register mit Adressen im unteren Speicherbereich tatsächlich in die Assembleranweisungen `cbi` und `sbi` übersetzt werden.

```
// alt:
sbi(PORTB, PB2);
cbi(PORTC, PC1);

// neu (auch nicht mehr wirklich neu...):
PORTB |= (1<<PB2);
PORTC &= ~(1<<PC1);
```

Will oder kann man den Code nicht portieren, ist zur weiteren Verwendung von sbi und cbi die Header-Datei **compat/deprecated.h** einzubinden. Wer unbedingt will, kann sich natürlich eigene Makros mit aussagekräftigeren Namen definieren. Zum Beispiel:

```
#define SET_BIT(PORT, BITNUM) ((PORT) |= (1<<(BITNUM)))
#define CLEAR_BIT(PORT, BITNUM) ((PORT) &= ~(1<<(BITNUM)))
#define TOGGLE_BIT(PORT, BITNUM) ((PORT) ^= (1<<(BITNUM)))
```

Selbstdefinierte (nicht-standardisierte) ganzzahlige Datentypen

Bei den im Folgenden genannten Typdefinitionen ist zu beachten, dass die Bezeichnungen für "Worte" teilweise je nach Prozessorplattform unterschiedlich verwendet werden. Die angegebenen Definitionen beziehen sich auf die im Zusammenhang mit AVR/8-bit-Controllern üblichen "Bit-Breiten" (In Erläuterungen zum ARM7TDMI z.B. werden oft 32-bit Integer mit "Wort" ohne weitere Ergänzung bezeichnet). Es empfiehlt sich, bei der Überarbeitung von altem Code die im Abschnitt *standardisierten ganzzahligen Datentypen* beschriebenen Datentypen zu nutzen (stdint.h) und damit "Missverständnissen" vorzubeugen, die z.B. bei der Portierung von C-Code zwischen verschiedenen Plattformen auftreten können.

```
typedef unsigned char    BYTE;        // besser: uint8_t aus <stdint.h>
typedef unsigned short  WORD;        // besser: uint16_t aus <stdint.h>
typedef unsigned long   DWORD;       // besser: uint32_t aus <stdint.h>
typedef unsigned long long QWORD;    // besser: uint64_t aus <stdint.h>
```

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

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

DWORD
Der Datentyp DWORD (gesprochen: Double-Word) definiert eine Variable mit 32 Bit Breite zur Darstellung von ganzen Zahlen im Bereich zwischen 0 ... 4294967295.

QWORD
Der Datentyp QWORD (gesprochen: Quad-Word) definiert eine Variable mit 64 Bit Breite zur Darstellung von ganzen Zahlen im Bereich zwischen 0 ... 18446744073709551615.

Zusätzliche Funktionen im Makefile

Bibliotheken (Libraries/.a-Dateien) hinzufügen

Um Funktionen aus Bibliotheken ("echte" Libraries, *.a-Dateien) zu nutzen, sind dem Linker die Namen der Bibliotheken als Parameter zu übergeben. Dazu ist die Option -l (kleines L) vorgesehen, an die der Name der Library angehängt wird.

Dabei ist zu beachten, dass der Name der Library und der Dateiname der Library nicht identisch sind. Der hinter -l angegebene Name entspricht dem Dateinamen der Library ohne die Zeichenfolge *lib* am Anfang des Dateinamens und ohne die Endung *.a*. Sollen z.B. Funktionen aus einer Library mit dem Dateinamen *libefsl.a* eingebunden (gelinkt) werden, lautet der entsprechende Parameter -lefs (vergl. auch -lm zum Anbinden von libm.a).

In Makefiles wird traditionell eine make-Variable LDLIBS genutzt, in die "-l-Parameter" abgelegt werden. Die WinAVR-makefile-Vorlage enthält diese Variable zwar nicht, dies stellt jedoch keine Einschränkung dar, da alle in der make-Variable LDFLAGS abgelegten Parameter an den Linker weitergereicht werden.

Beispiele:

```
# Einbinden von Funktionen aus einer Library efsl (Dateiname libefsl.a)
LDFLAGS += -lefs
# Einbinden von Funktionen aus einer Library xyz (Dateiname libxyz.a)
LDFLAGS += -lxyz
```

Liegen die Library-Dateien nicht im Standard Library-Suchpfad, sind die Pfade mittels Parameter *-L* ebenfalls anzugeben. (Der vordefinierte Suchpfad kann mittels *avr-gcc --print-search-dirs* angezeigt werden.)

Als Beispiel ein Projekt ("superapp2"), in dem der Quellcode von zwei Libraries (efsl und xyz) und der Quellcode der eigentlichen Anwendung in verschiedenen Verzeichnissen mit der folgenden "Baumstruktur" abgelegt sind:

```
superapp2
|
+----- efslsource (darin libefsl.a)
|
+----- xyzsource (darin libxyz.a)
```

```
|
+----- firmware (darin Anwendungs-Quellcode und Makefile)
```

Daraus folgt, dass im Makefile die Verzeichnisse `efslsource` und `xyzsource` in den Library-Suchpfad aufzunehmen sind:

```
LDFLAGS += -L../efslsource/ -L../xyzsource/
```

Fuse-Bits

Zur Berechnung der Fuse-Bits bietet sich neben dem Studium des Datenblattes auch der [AVR Fuse Calculator](#) an. Gewarnt werden muss vor der Benutzung von PonyProg, weil dort durch die negierte Darstellung gern Fehler gemacht werden.

Soll die Programmierung von Fuse- und Lockbits automatisiert werden, kann man dies ebenfalls durch Einträge im Makefile vornehmen, die beim Aufruf von "make program" an die genutzte Programmiersoftware übergeben werden. In der makefile-Vorlage von WinAVR (und mfile) gibt es dafür jedoch keine "Ausfüllhilfe" (Stand 9/2006). Die folgenden Ausführungen gelten für die Programmiersoftware [AVRDUDE](#) (Standard in der WinAVR-Vorlage), können jedoch sinngemäß auf andere Programmiersoftware übertragen werden, die die Angabe der Fuse- und Lockbits-Einstellungen per Kommandozeilenparameter unterstützt (z.B. `stk500.exe`). Im einfachsten Fall ergänzt man im Makefile einige Variablen, deren Werte natürlich vom verwendeten Controller und den gewünschten Einstellungen abhängen (vgl. Datenblatt Fuse-/Lockbits):

```
#----- Programming Options (avrdude) -----
#...
#Beispiel! f. ATmega16 – nicht einfach uebernehmen! Zahlenwerte anhand
#----- des Datenblatts nachvollziehen und gegebenenfalls aendern.
#
AVRDUDE_WRITE_LFUSE = -U lfuse:w:0xff:m
AVRDUDE_WRITE_HFUSE = -U hfuse:w:0xd8:m
AVRDUDE_WRITE_LOCK = -U lock:w:0x2f:m
#...
```

Damit diese Variablen auch genutzt werden, ist der Aufruf von `avrdude` im Makefile entsprechend zu ergänzen:

```
# Program the device.
program: $(TARGET).hex $(TARGET).eep
# ohne Fuse-/Lock-Einstellungen (nach WinAVR Vorlage Stand 4/2006)
# $(AVRDUDE) $(AVRDUDE_FLAGS) $(AVRDUDE_WRITE_FLASH) \
# $(AVRDUDE_WRITE_EEPROM)
# mit Fuse-/Lock-Einstellungen
$(AVRDUDE) $(AVRDUDE_FLAGS) $(AVRDUDE_WRITE_LFUSE) \
$(AVRDUDE_WRITE_HFUSE) $(AVRDUDE_WRITE_FLASH) \
$(AVRDUDE_WRITE_EEPROM) $(AVRDUDE_WRITE_LOCK)
```

Eine weitere Möglichkeit besteht darin, die Fuse- und Lockbit-Einstellungen vom Preprozessor/Compiler generieren zu lassen. Die Fuse-Bits werden dann bei Verwendung von `AVRDUDE` in eigene Hex-Files geschrieben. Hierzu kann man z.B. folgendes Konstrukt verwenden:

In eine der C-Sourcen wird eine Variable je Fuse-Byte vom Typ *unsigned char* deklariert und in eine extra Section gepackt. Dies kann entweder in einem vorhandenen File passieren oder in ein neues (z.B. `fuses.c`) geschrieben werden. Das File muss im Makefile aber auf jeden Fall mit kompiliert und gelinkt werden.

```
// tiny 2313 fuses low byte
#define CKDIV8 7
#define CKOUT 6
#define SUT1 5
#define SUT0 4
#define CKSEL3 3
#define CKSEL2 2
#define CKSEL1 1
#define CKSEL0 0

// tiny2313 fuses high byte
#define DWEN 7
#define EESAVE 6
#define SPIEN 5
#define WDTON 4
#define BODLEVEL2 3
#define BODLEVEL1 2
#define BODLEVEL0 1
#define RSTDISBL 0

// tiny2313 fuses extended byte
#define SELFPRGEN 0

#define LFUSE __attribute__((section("lfuses")))
#define HFUSE __attribute__((section("hfuses")))
#define EFUSE __attribute__((section("efuses")))

// select ext crystal 3-8Mhz
unsigned char lfuse LFUSE =
( (1<<CKDIV8) | (1<<CKOUT) | (1<<CKSEL3) | (1<<CKSEL2) |
(0<<CKSEL1) | (1<<CKSEL0) | (0<<SUT1) | (1<<SUT0) );
```

```

unsigned char hfuse HFUSE =
    ( (1<<DWN) | (1<<EESAVE) | (0<<SPIEN) | (1<<WDTON) |
      (1<<BODLEVEL2) | (1<<BODLEVEL1) | (0<<BODLEVEL0) | (1<<RSTDISBL) );
unsigned char efuse EFUSE =
    ((0<<SELFPGEN));

```

ACHTUNG: Die Bitpositionen wurden nicht vollständig getestet!

Eine "1" bedeutet hier, dass das Fuse-Bit *nicht* programmiert wird - die Funktion also i.A. nicht aktiviert ist. Eine "0" hingegen aktiviert die meisten Funktionen. Dies ist wie im Datenblatt (1 = unprogrammed, 0 = programmed).

Das Makefile muss nun noch um folgende Targets erweitert werden (mit Tabulator einrücken - nicht mit Leerzeichen):

```

lfuses: build
    -$(OBJCOPY) -j lfuses --change-section-address lfuses=0 \
    -O ihex $(TARGET).elf $(TARGET)-lfuse.hex
    @if [ -f $(TARGET)-lfuse.hex ]; then \
    $(AVRDUDE) $(AVRDUDE_FLAGS) -U lfuse:w:$(TARGET)-lfuse.hex; \
    fi;

hfuses: build
    -$(OBJCOPY) -j hfuses --change-section-address hfuses=0 \
    -O ihex $(TARGET).elf $(TARGET)-hfuse.hex
    @if [ -f $(TARGET)-hfuse.hex ]; then \
    $(AVRDUDE) $(AVRDUDE_FLAGS) -U hfuse:w:$(TARGET)-hfuse.hex; \
    fi;

efuses: build
    -$(OBJCOPY) -j efuses --change-section-address efuses=0 \
    -O ihex $(TARGET).elf $(TARGET)-efuse.hex
    @if [ -f $(TARGET)-efuse.hex ]; then \
    $(AVRDUDE) $(AVRDUDE_FLAGS) -U efuse:w:$(TARGET)-efuse.hex; \
    fi;

```

Das Target "clean" muss noch um die Zeilen

```

$(REMOVE) $(TARGET)-lfuse.hex
$(REMOVE) $(TARGET)-hfuse.hex
$(REMOVE) $(TARGET)-efuse.hex

```

erweitert werden, wenn auch die Fuse-Dateien gelöscht werden sollen.

Um nun die Fusebits des angeschlossenen Controllers zu programmieren muss lediglich "make lfuses", "make hfuses" oder "make efuses" gestartet werden. Bei den Fuse-Bits ist besondere Vorsicht geboten, da diese das Programmieren des Controllers unmöglich machen können. Also erst programmieren, wenn man einen HV-Programmierer hat oder ein paar Reserve-AVRs zur Hand ;-)

Um weiterhin den "normalen" Flash beschreiben zu können, ist es wichtig, für das Target "*.hex" im Makefile nicht nur "-R .eeprom" als Parameter zu übergeben sondern zusätzlich noch "-R lfuses -R efuses -R hfuses". Sonst bekommt AVRDUDE Probleme diese Sections in den Flash (wo sie ja nicht hingehören) zu schreiben.

Siehe auch: [Vergleich der Fuses bei verschiedenen Programmen](#)

Externe Referenzspannung des internen Analog-Digital-Wandlers

Die minimale (externe) Referenzspannung des ADC darf nicht beliebig niedrig sein, vgl. dazu das (aktuellste) Datenblatt des verwendeten Controllers. z.B. beim ATMEGA8 darf sie laut Datenblatt (S.245, Tabelle 103, Zeile "VREF") 2,0V nicht unterschreiten. HINWEIS: diese Information findet sich erst in der letzten Revision (Rev. 24860-10/04) des Datenblatts.

Meiner eigenen Erfahrung nach kann man aber (auf eigene Gefahr und natürlich nicht für Seriengeräte) durchaus noch ein klein wenig weiter heruntergehen, bei dem von mir unter die Lupe genommenen ATMEGA8L (also die Low-Voltage-Variante) funktioniert der ADC bei 5V Betriebsspannung mit bis zu VREF=1,15V hinunter korrekt, ab 1,1V und darunter digitalisiert er jedoch nur noch Blödsinn). Ich würde sicherheitshalber nicht unter 1,5V gehen und bei niedrigeren Betriebsspannungen mag sich die Untergrenze für VREF am Pin AREF ggf. nach oben(!) verschieben.

In der letzten Revision des Datenblatts ist außerdem korrigiert, dass ADC4 und ADC5 sehr wohl 10 Bit Genauigkeit bieten (und nicht bloß 8 Bit, wie in älteren Revisionen irrtümlich angegeben.)

TODO

- Aktualisierung Register- und Bitbeschreibungen an aktuelle AVR
- stdio.h, malloc()
- Code-Optimierungen ("tricks"), siehe auch Application Note [AVR035: Efficient C Coding for AVR](#)
- "naked"-Funktionen
- SPI siehe [SPI Bus mit Atmel AVR](#)
- I²C / TWI Bus [\[3\]](#)
- Bootloader (bez. auf boot.h)
- CAN-Bus
- Einsatz von einfachen Betriebssystemen auf dem AVR
- Übersicht zu den C bzw. GCC-predefined Makros (__DATE__, __TIME__,...)

- ADC ;
- Timer
- USB ; Steuerung mit USB
- Multiplexen Siebensegment
- Sichere vs. effiziente Serialisierung von Datentypen
- Zustandsautomaten

Softwareentwicklung

"Wild drauflos" zu programmieren kann nach einiger Zeit frustrieren, da mehr Zeit erforderlich wird, das Programm neuen Anforderungen anzupassen. Wer erst etwas Zeit darauf verwendet, ein offenes Konzept (erweiterbare Programmstruktur, Algorithmen) zu entwickeln (ggf. in Ruhe mit Papier und Bleistift), wird später schneller ans Ziel gelangen.

<http://de.wikipedia.org/wiki/Softwareentwicklung>

Programmierstil

Damit ein größeres Programm (nach längerer Zeit) überschaubar bleibt, sollte man sich bei der Gliederung, Namensgebung, Formatierung und Kommentierung an bewährten, begründeten Konzepten orientieren.

- <http://www.avrfreaks.net/index.php?name=PNphpBB2&file=viewtopic&t=48535&postdays=0&postorder=asc&sid=9098c3a3c0cdfcdafbd8187255136d26> Modularizing C Code: Managing large projects
- [http://www.mikrocontroller.net/articles/Include-Files_\(C\)](http://www.mikrocontroller.net/articles/Include-Files_(C))
- <http://www.mikrocontroller.net/topic/130218>
- <http://www.elektroniknet.de/home/embeddedsystems/fachwissen/uebersicht/software/entwicklungssoftware/der-programmierstandard-misra/>
- <http://www.mikrocontroller.net/topic/132304>
- <http://de.wikipedia.org/wiki/Programmierstil>

Von „<http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial>“

[Kategorien: Avr-gcc Tutorial](#) | [AVR](#)