AVR-GCC-Tutorial

Inhaltsverzeichnis

Vorwort

Dieses Tutorial soll Einsteigern helfen, mit der Programmiersprache **C** die Mikrocontroller der Atmel AVR-Reihe zu programmieren.

Die ursprüngliche Version stammt von Christian Schifferle, die meisten aktuellen Anpassungen von Benutzer:Mthomas. Viele der im Original-Dokument verwendeten Funktionen sind in aktuellen Versionen des avr-gcc C-Compilers und der avr-libc zwar noch enthalten, aber als überholt (*deprecated*) ausgewiesen. D.h. diese Funktionen sollten nicht mehr verwendet werden und existieren in zukünftigen Versionen des Compilers/der Library nicht mehr (z.B. sbi(), cbi(), outp(), inp()). Der Artikel wurde auf die neuen Funktionen/Methoden angepasst.

In diesem Text wird häufig auf die Standardbibliothek für den avr-gcc-Compiler, die avr-libc, verwiesen. Die **Dokumentation der avr-libc** findet sich hier (*http://www.nongnu.org/avr-libc/user-manual/index.html*). Sollte die Dokumentation zeitweise nicht online verfügbar sein, kann diese **hier als PDF-Datei** (Stand Version 1.0.4) heruntergeladen werden. Bei WinAVR gehört diese Dokumentation zum Lieferumfang und wird mitinstalliert.

Benötigte Werkzeuge

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

- Testboard für die Aufnahme eines AVR Controllers, der vom gcc Compiler unterstützt wird (alle ATmegas und die meisten AT90). Dieses Testboard kann durchaus auch selber zusammen gelötet werden. Einige Registerbeschreibungen dieses Tutorials beziehen sich auf den inzwischen veralteten AT90S2313. Der weitaus größte Teil des Textes gilt aber für alle Controller der AVR-Familie. Brauchbare Testplattform sind auch das STK500 und der AVR Butterfly von Atmel.
- AVRGCC-Compiler. Kostenlos erhältlich für nahezu alle Plattformen/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, AVR-Studio).
- Nicht unbedingt erforderlich aber zur Simulation und zum Debuggen unter MS-Windows recht nützlich: AVR-Studio (siehe Abschnitt Exkurs: makefiles).

Was tun, wenn's nicht "klappt"?

- Herausfinden, ob es tatsächlich ein avr(-gcc) spezifisches Problem ist oder die 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 Dokumentation der avr-libc (*http://www.nongnu.org/avr-libc/user-manual/index.html*) 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 (*http://www.avrfreaks.net*) nach vergleichbaren Problemen absuchen.
- Das Archiv der avr-gcc Mailing-Liste (*http://www.avr1.org/pipermail/avr-gcc-list/*) nach vergleichbaren Problemen absuchen.
- Nach Beispielcode suchen. Vor allem in der Academy von AVRFREAKS (*http://www.avrfreaks.net*) (anmelden).
- Google oder alltheweb befragen schadet nie.
- Bei Problemen mit der Ansteuerung interner AVR-Funktionen mit C-Code: das Datenblatt des Controllers lesen (ganz und am Besten zweimal)
- einen Beitrag in eines der Foren oder eine Mail an die Mailing-Liste schreiben. Dabei mgl. viel Information geben: Controller, Compilerversion, genutzte Bibliotheken, Ausschnitte aus dem Quellcode, genaue Fehlermeldungen bzw. Beschreibung des Fehlverhaltens. Bei Ansteuerung externer Geräte die Beschaltung beschreiben oder skizzieren (z.B. mit Andys ASCII Circuit (*http://www.tech-chat.de/*)). Siehe dazu auch: "Wie man Fragen stellt" (*http://www.lugbz.org/documents/smart-questions_de.html*).

Exkurs: makefiles

Wenn man bisher gewohnt ist, mit integrierten Entwicklungsumgebung à la Visual-C Programme zu erstellen, wirkt das makefile-Konzept auf den ersten Blick etwas kryptisch. Nach kurzer Einarbeitung ist diese Vorgehensweise jedoch sehr praktisch. Diese Dateien (üblicher Name: 'Makefile' ohne Dateiendung) dienen der Ablaufsteuerung des Programms make, das auf allen Unix/Linux-Systemen installiert sein sollte, und in einer Fassung fuer MS-Windows auch in WinAVR (Unterverzeichnis utils/bin) enthalten ist.

Im Unterverzeichnis *sample* einer WinAVR-Installation findet man eine sehr brauchbare Vorlage, die sich einfach an das eigene Projekt anpassen lässt (lokale Kopie Stand Sept. 2004). Wahlweise kann man auch mfile (*http://www.sax.de/~joerg/mfile/*) von Jörg Wunsch nutzen. mfile erzeugt ein makefile nach Einstellungen in einer grafischen Nutzeroberfläche, wird bei WinAVR mitinstalliert, ist aber als TCL/TK-Programm auf nahezu allen Platformen lauffähig.

Die folgenden Ausführungen beziehen sich auf das WinAVR Beispiel-Makefile.

Ist im Makefile alles richtig eingestellt genügt es, sich drei Parameter zu merken, die über die shell bzw. die Windows-Kommandozeile (cmd.exe/command.com) als Parameter an "make" übergeben werden. Das Programm make sucht sich "automatisch" das makefile im aktuellen Arbeitsverzeichnis und führt die darin definierten Operationen für den entsprechenden Aufrufparameter durch.

make all	Erstellt aus den in im makefile angegebenen Quellcodes eine *hex*-Datei (und ggf. auch *eep*-Datei).
make program	Überträgt die hex-Datei (und wahlweise auch die eep-Datei für den EEPROM) zum AVR.
make clean	löscht alle temporären Dateien, also auch die hex-Datei

Diese Aufrufe können in die allermeisten Editoren in "Tool-Menüs" eingebunden werden. Dies erspart den Kontakt mit der Kommandozeile. (Bei WinAVR sind die Aufrufe bereits im Tools-Menü des mitgelieferten Editors Programmers-Notepad eingefügt.)

Üblicherweise sind folgende Daten im makefile anzupassen:

- Controllertyp
- Quellcode-Dateien (c-Dateien)
- Typ und Anschluss des Programmiergeräts

seltener sind folgende Einstellungen durchzuführen:

- Grad der Optimierung
- Methode zur Erzeugung der Debug-Symbole (Debug-Format)
- Assembler-Quellcode-Dateien (S-Dateien)

Die in den folgenden Unterabschnitten gezeigten makefile-Ausschnitte sind für ein Programm, das auf einem ATmega8 ausgeführt werden soll. Der Quellcode besteht aus den c-Dateien superprog.c (darin main()), uart.c, lcd.c und 1wire.c. Im Quellcodeverzeichnis befinden sich diese Dateien: superprog.c, uart.h, uart.c, lcd.h, lcd.c, 1wire.h, 1wire.c und das makefile (die angepasste Kopie des WinAVR-Beispiels).

Der Controller wird mittels AVRDUDE über ein STK200-Programmierdongle an der Schnittstelle lpt1 (bzw. /dev/lp0) programmiert. Im Quellcode sind auch Daten für die *section .eeprom* definiert (siehe Abschnitt Speicherzugriffe TODO: nach unten verlinken), diese sollen beim Programmieren gleich mit ins EEPROM geschrieben werden.

Controllertyp setzen

Dazu wird die "make-Variable" MCU entsprechend dem Namen des verwendeten Controllers gesetzt. Ein Liste der von avr-gcc und der avr-libc untersützten Typen findet sich in der Dokumentation der avr-libc (*http://www.nongnu.org/avr-libc/user-manual/index.html*).

```
# Kommentare in Makefiles beginnen mit einem Doppelkreuz
...

# ATmega8 at work
MCU = atmega8
# oder MCU = atmega16
# oder MCU = at90s8535
# oder ...
...
```

Quellcode-Dateien einstellen

Den Namen der Quellcodedatei welche die Funktion main enthält, wird hinter TARGET eingetragen. Dies jedoch ohne die Endung *.c*.

```
...
TARGET = superprog
...
```

Besteht das Projekt wie im Beispiel aus mehr als einer Quellcodedatei, sind die weiteren c-Dateien (nicht die Header-Dateien, vgl. Include-Files (C)) durch Leerzeichen getrennt bei SRC einzutragen. Die bei TARGET definierte Datei ist schon in der SRC-Liste enthalten. Diesen Eintrag nicht löschen!

```
...
SRC = $(TARGET).c uart.c lcd.c 1wire.c
...
```

Alternativ kann man die Liste der Quellcodedateien auch mit dem Operator += erweitern.

```
SRC = $(TARGET).c uart.c 1wire.c
# lcd-Code fuer Controller xyz123 (auskommentiert)
# SRC += lcd_xyz.c
# lcd-Code fuer "Standard-Controller" (genutzt)
SRC += lcd.c
```

Programmiergerät einstellen

Die Vorlagen sind auf die Programmiersoftware AVRDUDE angepasst, jedoch lässt sich auch andere Programmiersoftware einbinden, sofern diese über Kommandozeile gesteuert werden kann (z.B. stk500.exe, uisp, sp12).

```
...
# Einstellung fuer STK500 an com1 (auskommentiert)
# AVRDUDE_PROGRAMMER = stk500
# com1 = serial port. Use lpt1 to connect to parallel port.
# AVRDUDE_PORT = com1    # programmer connected to serial device

# Einstellung fuer STK200-Dongle an lpt1
AVRDUDE_PROGRAMMER = stk200
AVRDUDE_PORT = lpt1
...
```

Sollen Flash(=.hex) und EEPROM(=.eep) zusammen auf den Controller programmiert werden, ist das Kommentarzeichen vor AVRDUDE_WRITE_EEPROM zu löschen.

```
...
#Auskommentiert: EERPOM-Inhalt wird nicht mitgeschrieben
#AVRDUDE_WRITE_EEPROM = -U eeprom:w:$(TARGET).eep

#Nich-auskommentiert EERPOM-Inhalt wird mitgeschrieben
AVRDUDE_WRITE_EEPROM = -U eeprom:w:$(TARGET).eep
...
```

Anwendung

Die Eingabe von *make all* im Arbeitsverzeichnis mit dem makefile und den Quellcodedateien erzeugt (unter anderem) die Dateien superprog.hex und superprog.eep. Abhängigkeiten zwischen den einzelnen c-Dateien werden dabei automatisch berücksichtigt. Die *superprog.hex* und *superprog.eep* werden mit *make program* zum Controller übertragen. Mit *make clean* werden alle temporären Dateien gelöscht (="aufgeräumt").

Sonstige Einstellungen

Optimierungsgrad

Der gcc-Compiler kennt verschiedene Stufen der Optimierung. Nur zu Testzwecken sollte die Optimierung ganz deaktiviert werden (*OPT = 0*). Die weiteren möglichen Optionen weisen den Compiler an, möglichst kompakten oder möglichst schnellen Code zu erzeugen. In den weitaus meisten Fällen ist *OPT = s* die optimale (sic) Einstellung, damit wird kompakter und oft auch der "schnellste" Maschinencode erzeugt.

Debug-Format

Unterstützt werden die Formate stabs und dwarf-2. Das Format wir hinter *DEBUG =* eingestellt. Siehe dazu Abschnitt *Eingabedateien zur Simulation*.

Assembler-Dateien

Die im Projekt genutzten Assembler-Dateien werden hinter ASRC durch Leerzeichen getrennt aufgelistet. Assembler-Dateien haben immer die Endung .S (grosses S). Ist zum Beispiel der Assembler-Quellcode eines Software-UARTs in einer Datei softuart.S enthalten lautet die Zeile: *ASRC = softuart.S*

Eingabedateien zur Simulation in AVR-Studio

Mit älteren AVR-Studio-Versionen kann man nur auf Grundlage sogenannter *coff*-Dateien simulieren. Neuere Versionen von AVR-Studio (ab 4.10.356) unterstützen zudem das

modernere aber noch experimentelle dwarf-2-Format, das ab WinAVR 20040722 (avr-gcc 3.4.1/Binutils inkl. Atmel add-ons) "direkt" vom Compiler erzeugt wird.

Vorgehensweise bei extcoff:

- im Makefile bei DEBUG:

```
DEBUG=stabs
```

- *make extcoff* (evtl. vorher *make clean*)
- die erzeugte *cof*-Datei (im Beispiel oben *superprog.cof*) in AVR-Studio laden
- AVR-Simulator und zu simulierenden Controller wählen, "Finish"
- weiteres siehe AVR-Studio Online-Hilfe

Vorgehensweise bei dwarf-2:

- im Makefile bei DEBUG:

```
DEBUG=dwarf-2
```

- *make all* (evtl. vorher *make clean*)
- die erzeugte *elf*-Datei (im Beispiel oben *superprog.elf*) in AVR-Studio laden
- AVR-Simulator und zu simulierenden Controller wählen, "Finish"
- weiteres siehe AVR-Studio Online-Hilfe

Beim Simulieren scheinen oft "Variablen zu fehlen". Ursache dafür ist, dass der Compiler die "Variablen" direkt Registern zuweist. Dies kann vermieden werden, indem die Optimierung abgeschaltet wird (im makefile). Man simuliert dann jedoch ein vom optimierten Code stark abweichendes Programm. Das Abschalten der Optimierung wird nicht empfohlen.

Statt des Software-Simulators kann das AVR-Studio auch genutzt werden, um mit dem ATMEL JTAGICE, ein Nachbau davon (BootICE, Evertool o.ä.) oder dem ATMEL JTAGICE MKII "im System" zu debuggen. Dazu sind keine speziellen Einstellungen im makefile erforderlich. Debugging bzw. "In-System-Emulation" mit dem JTAGICE und JTAGICE MKII sind in der AVR-Studio Online-Hilfe beschrieben.

Ganzzahlige (Integer) Datentypen

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

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 daher, die im

nachfolgenden Abschnitt beschriebenen *standardisierten ganzzahligen Datentypen* zu nutzen 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 <inttypes.h>
typedef unsigned short     WORD;       // besser: uint16_t aus <inttypes.h>
typedef unsigned long      DWORD;      // besser: uint32_t aus <inttypes.h>
typedef unsigned long long QWORD;      // besser: uint64_t aus <inttypes.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.

Standardisierte Integer(Ganzzahl)-Typen

Standardisierte Datentypen werden in der Header-Datei inttypes.h definiert. In neueren Versionen der avr-libc (ab V1.2.0) heisst die Datei C99-standardkonform stdint.h. inttypes.h existiert aus Kompatibiltätsgründen weiterhin und enthält ein *#include <stdint.h>*. Zur Nutzung der standardisierten Typen bindet man die "Definitionsdatei" wie folgt ein:

```
#include <inttypes.h>

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

Ein Vierfach-Wort (64Bit) entspricht beim AVR *uint64_t*, ein Doppel-Wort (32bit) entspricht *uint32_t*, ein Wort (16bit) entspricht *uint16_t* und ein Byte (8bit) entspricht *uint8_t*. Die Typen ohne vorangestelltes *u* können auch vorzeichenbehaftete Zahlen speichern. *int8_t* geht also von -128 bis 127, *uint8_t* dagegen geht von 0 bis 255.

- siehe auch: Dokumentation der avr-libc (*http://www.nongnu.org/avr-libc/user-manual/index.html*) 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 Bit breit ist.
Hier bietet uns die Programmiersprache C ein mächtiges Werkzeug an, mit dessen Hilfe wir 8 Bits in einer einzelnen Bytevariable zusammen fassen und (fast) wie 8 einzelne Variablen ansprechen können.
Die Rede ist von sogenannten Bitfeldern. Diese werden als Strukturelemente definiert. Sehen wir uns dazu doch am besten gleich ein Beispiel an:

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

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

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

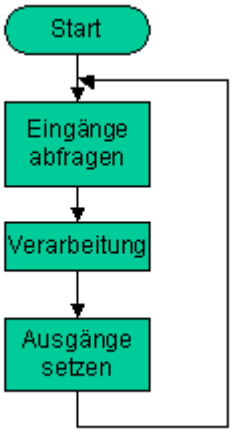
(mt Empfehlung: Bitfelder sparen zwar Platz, verschlechtern aber unter Umständen die Les- und Wartbarkeit des Codes. Anfängern wird geraten, ein "ganzes" Byte (uint8_t) zu nutzen auch wenn nur ein Bitwert gespeichert werden soll.)

Grundsätzlicher Programmaufbau eines µC-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.

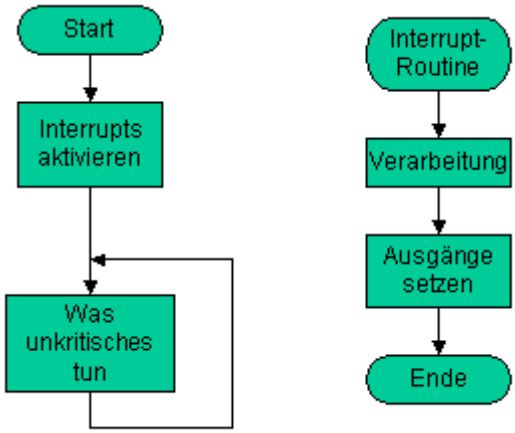
Sequentieller Programmablauf

Bei dieser Programmiertechnik 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.



Allgemeiner Zugriff auf Register

Die AVR-Controller verfügen über eine Vielzahl von Registern. Die meisten davon sind sogenannte Schreib-/Leseregister. Das heisst, das Programm kann die Inhalte der Register auslesen und beschreiben.
Einige Register haben spezielle Funktionen, andere wiederum könne für allgemeine Zwecke (Speichern von Datenwerten) verwendet werden.

Einzelne Register sind bei allen 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.
Wenn im Makefile der MCU-Typ definiert ist, wird vom System automatisch die zum Typen passende Definitionsdatei genutzt, sobald man im Code die allgemeine "io.h" Header-Datei einbinden.

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

Ist im Makefile der MCU Type z.B. mit dem Inhalt atmega8 definiert, wird beim einlesen der io.h-Datei implizit ("automatisch") auch die iom8.h-Datei mit den Register-Definitionen für den ATmega8 eingelesen.

I/O-Register

Die I/O-Register haben einen besonderen Stellenwert bei den AVR Controllern. Sie dienen dem Zugriff auf die Ports und die Schnittstellen des Controllers.
Wir unterscheiden zwischen 8-Bit und 16-Bit Registern. Vorerst behandeln wir mal die 8-Bit Register.

Lesen eines I/O-Registers

Zum Lesen kann man auf I/O Register einfach wie auf eine Variable zugreifen. Die spezielle Funktion inp() ist nicht mehr notwendig und veraltet.

Beispiel:

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

uint8_t foo;

...

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

Lesen eines Bits

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

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, 4 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 bit_is_clear bzw. bit_is_set sind nicht unbedingt erfoderlich, man kann auch "einfache" C-Syntax verwenden. Der universell verwendbar ist. *bit_is_set* entspricht dabei z.B. (Variable & (1 << Bitnummer)). Das Ergebnis ist <>0 wenn das Bit gesetzt und 0 wenn nicht gesetzt ist.

- siehe auch Bitmanipulation

Schreiben eines I/O-Registers

Zum Schreiben kann man I/O Register einfach wie eine Variable setzen. Die spezielle Funktion outp() ist nicht mehr notwendig, veraltet und sollte nicht mehr genutzt werden.

Beispiel:

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

...

int
main(void)
{
    DDRA = 0xff;    /* Setzt das Richtungsregister des Ports A auf 0xff
(alle Pins als Ausgang) */
    PORTA = 0x03;   /* Setzt PortA auf 0x03, Bit 0 und 1 "high", restliche
"low" */
    ...
}
```

Schreiben eines Bits

Einzelne Bits setzt man "Standard-C-Konform" mittels logischer (bit-) Operationen.

mit dem Ausdruck:

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

Die Funktionen sbi und cbi sind dazu nicht notwendig, veraltet und sollten nicht mehr genutzt werden.

Beispiel:

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

- siehe auch:

Bitmanipulation

Dokumentation der avr-libc (Abschnitt Modules/Special Function Registers)

Warten auf einen bestimmten Zustand

Es gibt in der Bibliothek sogar Funktionen, die warten, bis ein bestimmter Zustand auf einem Bit erreicht ist.
Es ist allerdings normalerweise eine eher unschöne Programmiertechnik, da in diesen Funktionen "blockierend gewartet" wird. D.h., der Programmablauf bleibt 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.

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.

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

#define WARTEPIN PINA
#define WARTEBIT 2

// 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.
Das niederwertigste Bit hat die Bitnummer 0.

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

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

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

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

siehe auch:

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

Zugriff auf Ports

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

DDRx	Datenrichtungsregister für Port**x**. **x** entspricht **A**, **B**, **C**, **D** usw. (abhängig von der Anzahl der Ports des verwendeten AVR). Bit im Register gesetzt (1) für Ausgang, Bit gelöscht (0) für Eingang.
PORTx	Datenregister für Port**x**. Dieses Register wird verwendet, um die Ausgänge eines Ports anzusteuern. Wird ein Port als Eingang geschaltet, so können mit diesem Register die internen Pull-Up Widerstände aktiviert oder deaktiviert werden (1 = aktiv).
PINx	Eingangadresse für Port**x**. Zustand des Ports. Die Bits in PINx entsprechen dem Zustand der Portspins. Bit gesetzt (1) wenn Pin "high/an", Bit gelöscht (0) wenn Portpin "low/aus".

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.

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

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

...

// Setzen der Bits 0,1,2,3 und 4
// Binär 00011111 = Hexadezimal 1F

DDRB = 0x1F;    /* direkte Zuweisung - unuebersichtlich */

/* mehr Tipparbeit aber uebersichtlicher: */
DDRB = (1 << DDB0) | (1 << DDB1) | (1 << DDB2) | (1 << DDB3) | (1 << DDB4);
...
```

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

Ganze Ports

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

```
DDRB = 0xff;
```

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

Digitale Signale

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

Ausgänge

Wir wollen nun einen als Ausgang definierten Pin auf Logisch 1 setzen. Dazu schreiben wir den entsprechenden Wert in das Portregister des entsprechenden Ports.

Mit dem Befehl

```
PORTB=0x04; /* besser PORTB=(1<<DDB2) */
```

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

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

```
PORTB = PORTB | 0x04; /* besser: PORTB = PORTB | ( 1<<DDB2 ) */
/* vereinfacht durch Nutzung des |= Operators : */
PORTB |= (1<<DDB2)
```

Die Funktionen cbi und sbi sind dazu nicht notwendig, veraltet und sollten nicht mehr genutzt werden.

Eingänge (Wie kommen Signale in den µC)

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 wird den Pegel über entsprechende Hardware (z.B. Optokoppler, Spannungsteiler "Levelshifter") anpassen.

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

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

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

```
#include <avr/io.h>
...
bPortD = PIND;
...
```

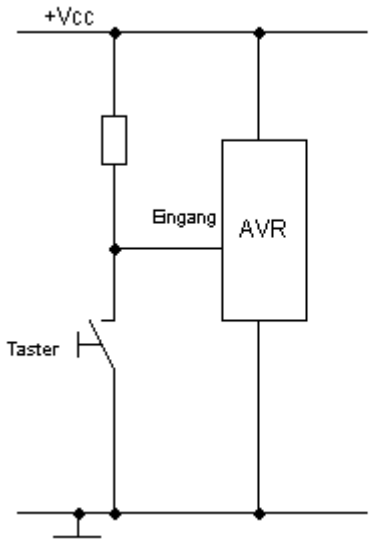
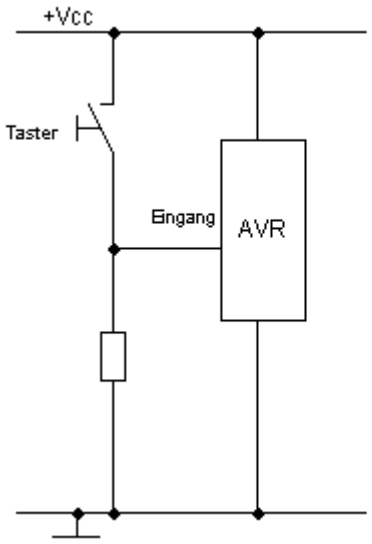
Mit den C-Bitoperationen kann man den Status einzelner Bits abfragen.

```
#include <avr/io.h>
...
/* Fuehre Aktion aus, wenn Bit1 in PINC gesetzt (1) */
if ( PINC & (1<<1) ) {
  /* Aktion */
}

/* Fuehre Aktion aus, wenn Bit 2 in PINB geloescht (0) */
if ( !(PINB & (1<<2)) ) {
  /* Aktion */
}
...
```

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*):

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

Pull-Up Widerstände aktivieren

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

Wird der Wert des entsprechenden Portpins auf 1 gesetzt so ist der Pull-Up Widerstand aktiviert. Bei einem Wert von 0 ist der Pull-Up Widerstand nicht aktiv.
Man sollte jeweils entweder den internen oder einen externen Pull-Up Widerstand verwenden, aber nicht beide zusammen.

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

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

Tastenentprellung

Nun haben alle mechanischen Kontakte, sei es von Schaltern, Tastern oder auch von Relais, die unangenehme Eigenschaft zu prellen.
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.

```
#include <avr/io.h>
#include <inttypes.h>
#define F_CPU 3686400UL                 /* Quarz mit 3.6854 Mhz  */
#include <avr/delay.h>                  /* definiert _delay_ms() */

/* 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(100);
                if ( ! (*port & (1 << pin)) )
                {
                        /* Anwender Zeit zum Loslassen des Tasters geben */
                        _delay_ms(100);
                        return 1;
                }
        }
        return 0;
}
int main(void)
{
        DDRB &= ~( 1 << PB0 )             /* PIN PB0 auf Eingang Taster)*/
        PORTB |= ( 1 << PB0 9)            /* Pullup-Widerstand Aktivieren*/
        ...
        if (debounce(&PINB, PB0))    /* Falls Taster an PIN PB0 gedrueckt*/
                PORTD = PIND ^ ( 1 << PD7 );  /*LED an Port PD7 an-
                                                bzw. ausschalten */
        }
```

Bei diesem Beispiel ist zu beachten, dass der AVR im Falle eines Tastendrucks 200ms wartet, also brach liegt. Zeitkritische Anwendungen sollten ein anderes Verfahren wählen.

Analog

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

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

Einige der Portregister in den AVR-Controllern sind 16 Bit breit, Man spricht dann von einem **Wort**.

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>
...
uint16_t foo;

foo=ADC; /* setzt die Wort-Variable 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.

```
#include <avr/io.h>
#define F_CPU 3686400
#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;
...
```

Bei einigen AVR-Typen (z.B. ATmega8) 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*.

- Siehe auch: Dokumentation der avr-libc (*http://www.nongnu.org/avr-libc/user-manual/index.html*) Abschnitt Related Pages/Frequently Asked Questions/Nr. 8

Der UART, Teil 1

Wir werden in zukünftigen Übungen in die Situation kommen, dass wir Informationen vom Controller auswerten bzw. anzeigen müssen wobei es vielleicht dann nicht mehr reicht, ein paar Leuchtdioden anzusteuern.
Somit brauchen wir eine Verbindung vom AVR zu unserem PC, und dies am besten über die serielle Schnittstelle.

Allgemeines zum UART

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

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

Neuere AVRs (ATmega) verfügen über einen oder zwei U**S**ART(s), dieser unterschiedet sich vom UART häuptsächlich 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 Konfigurations/Datenregister. Das Datenblatt gibt darüber Auskunft. Die Folgende Tabelle gibt nur die Register für die (veralteten) UARTs wieder.

UCR

UART **C**ontrol **R**egister.

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 C**omplete **I**nterrupt **E**nable)

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 C**omplete **I**nterrupt **E**nable) 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** (**U**ART **D**ata **R**egister **E**mpty **I**nterrupt **E**nable) Wenn dieses Bit gesetzt ist, wird ein UART Datenregister Leer Interrupt ausgelöst, wenn der UART wieder bereit ist um ein neues zu sendendes Zeichen zu übernehmen. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein. **RXEN** (**R**eceiver **E**nable) 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** (**T**ransmitter **E**nable) Nur wenn dieses Bit gesetzt ist, arbeitet der Sender des UART überhaupt. Wenn das Bit nicht gesetzt ist, kann der entsprechende Pin des AVR als normaler I/O-Pin verwendet werden. **CHR9** (9 Bit Characters) Wenn dieses Bit gesetzt ist, können 9 Bit lange Zeichen übertragen und empfangen werden. Das 9. Bit kann bei Bedarf als zusätzliches Stopbit oder als Paritätsbit verwendet werden. Man spricht dann von einem 11-Bit Zeichenrahmen: 1 Startbit + 8 Datenbits + 1 Stopbit + 1 Paritätsbit = 11 Bits **RXB8** (Receive Data Bit 8) Wenn das vorher erwähnte CHR9-Bit gesetzt ist, dann enthält dieses Bit das 9. Datenbit eines empfangenen Zeichens. **TXB8** (Transmit Data Bit 8) Wenn das vorher erwähnte CHR9-Bit gesetzt ist, dann muss in dieses Bit das 9. Bit des zu sendenden Zeichens eingeschrieben werden bevor das eigentliche Datenbyte in das Datenregister geschrieben wird.
USR	**U**ART **S**tatus **R**egister. 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.
Das Bit wird nur dann automatisch gelöscht, wenn der entsprechende Interrupthandler aufgerufen wird, ansonsten müssen wir das Bit selber löschen.

UDRE (**U**ART **D**ata **R**egister **E**mpty)

Dieses Bit wird vom AVR gesetzt, wenn ein Zeichen vom Sendedatenregister in das Send-Schieberegister übernommen wurde und der UART nun wieder bereit ist, ein neues Zeichen zum Senden aufzunehmen.
Das Bit wird automatisch gelöscht, wenn ein Zeichen in das Sendedatenregister geschrieben wird.

FE (**F**raming **E**rror)

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.

OR (**O**ver**R**un)

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.
UDR	**U**ART **D**ata **R**egister. 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.
UBRR	**U**ART **B**aud **R**ate **R**egister. 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: $UBRR = \frac{Taktfrequenz}{Baudrate*16} - 1$ Es sind Baudraten bis zu 115200 Baud und höher möglich.

Die Hardware

Der UART basiert auf normalem TTL-Pegel mit 0V (LOW) und 5V (HIGH). Die Schnittstellenspezifikation für RS232 definiert jedoch -3V ... -12V (LOW) und +3 ... +12V (HIGH). Zudem muss der Signalaustausch zwischen AVR und Partnergerät invertiert werden. Für die Anpassung der Pegel und das Invertieren der Signale gibt es fertige Schnittstellenbausteine. Der bekannteste davon ist wohl der MAX232.

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 empfielt sich auch immer ein Blick in die AVR_Checkliste.

Senden mit dem UART

Wir wollen nun Daten mit dem UART auf die serielle Schnittstelle ausgeben.
Dazu müssen wir den UART zuerst mal initialisieren. Dazu setzen wir je nach gewünschter Funktionsweise die benötigten Bits im **U**ART **C**ontrol **R**egister.
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);
```

Der Atmega16 hat mehrere Konfigurationsregister für USART und erfordert eine etwas andere Konfiguration

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

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

```
/* UART-Init beim AT90S2313 */
#define F_CPU 4000000;       // Zum Beispiel 4Mhz-Quarz
#define UART_BAUD_RATE 9600  // Wir versuchen mal mit 9600 Baud
UBRR = F_CPU / (UART_BAUD_RATE * 16L) - 1;
```

Wieder für den Mega16 mit einem 16bit-Register eine andere Programmierung

```
  /* USART-Init beim ATmegaXX */
  #define F_OSC 3686400           /* Oszillator-Frequenz in Hz */
  #define UART_BAUD_RATE 9600
  #define UART_BAUD_CALC(UART_BAUD_RATE,F_OSC)
((F_OSC)/((UART_BAUD_RATE)*16)-1)

  UBRRH=(uint8_t)(UART_BAUD_CALC(UART_BAUD_RATE,F_OSC)>>8);
  UBRRL=(uint8_t)UART_BAUD_CALC(UART_BAUD_RATE,F_OSC);

  /* alternativ bei der avr-libc "direkt 16bit" : */
  UBRR=UART_BAUD_CALC(UART_BAUD_RATE,F_OSC);
```

Senden einzelner Zeichen

Um nun ein Zeichen auf die Schnittstelle auszugeben, müssen wir dasselbe lediglich in das **U**ART **D**ata **R**egister schreiben.

```
    while (!(USR & (1<<UDRE))); /* warten bis Senden moeglich
*/
    UDR = 'x';                  /* Schreibt das Zeichen x auf die
Schnittstelle */
```

Schreiben einer Zeichenkette (String)

Es ist darauf zu achten, dass vor dem Senden geprüft wird, ob der UART bereit ist den "Sendeauftrag" entgegenzunehmen. Zu diesem Zweck können wir uns für's Erste eine einfache Funktion basteln:

```
void
uart_putc(unsigned char c)
{
    while(!(USR & (1 << UDRE)))
    ;   /* warte, bis UDR bereit */

    UDR = c;     /* sende Zeichen */
}

void
uart_puts (char *s)
{
    while (*s)
    {   /* so lange *s != NULL */
        uart_putc(*s);
        s++;
    }
}
```

Warteschleifen 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. emfangenen Zeichen/Bytes zwischengespeichert und mittels Interruptroutinen an den U(S)ART weitergebgen bzw. 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.

- siehe auch: Weiterführende Informationen inkl. Beispielen für die Nutzung von stdio-Funktionen (prinf etc.) im AVR-Tutorial - UART.
- Peter Fleurys (*http://homepage.sunrise.ch/mysunrise/peterfleury/*) UART-Bibiliothek fuer avr-gcc/avr-libc

Analoge Ein- und Ausgabe

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

ADC (Analog Digital Converter)

Der **ADC** wandelt analoge Signale in digitale Werte um, welche vom Controller interpretiert werden können. Einige AVR-Typen haben bereits einen oder mehrere solcher **ADC**'s eingebaut, bei den kleineren AVR's müssen wir uns anders aus der Affäre ziehen. Die Genauigkeit, mit welcher ein analoges Signal aufgelöst werden kann, wird durch die

Auflösung des **ADC** in Anzahl Bits angegeben, man hört bzw. liest jeweils von 8-Bit **ADC** oder 10-Bit **ADC** oder noch höher.
Ein **ADC** mit 8 Bit Auflösung kann somit das analoge Signal mit einer Genauigkeit von 1/256 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:

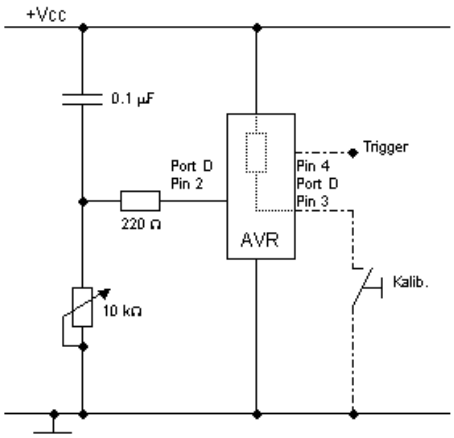
Eingangsspannung am ADC	Entsprechender Messwert
0...<0.625V	0
0.625...<1.25V	1
1.25...<1.875V	2
1.875...<2.5V	3
2.5...<3.125V	4
3.125...<3.75V	5
3.75...<4.375V	6
4.375...5V	7

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

Messen eines Widerstandes

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

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



Wenn wir nun den Pin des AVR als Ausgang schalten und auf Logisch 1 (HIGH) legen, dann liegt an beiden Platten des Kondensators **Vcc** an und dieser wird entladen (Klingt komisch, mit **Vcc** entladen, ist aber so, da an beiden Seiten des Kondensators das gleiche Potential anliegt und somit eine Potentialdifferenz von 0V besteht => Kondensator ist entladen). Nachdem nun der Kondensator genügend entladen ist schalten wir einfach den Pin als Eingang wodurch dieser hochohmig wird. Der Kondensator lädt sich jetzt über das Poti auf, dabei steigt der Spannungsabfall über dem Kondensator und derjenige über dem Poti sinkt. Fällt nun der Spannungsabfall über dem Poti unter die Thresholdspannung des Eingangspins (2/5 Vcc, also ca. 2V), dann schaltet der Eingang von HIGH auf LOW um. Wenn wir nun messen (zählen), wie lange es dauert, bis der Kondensator so weit geladen ist, dann haben wir einen ungefähren Wert der Potentiometerstellung.
Der 220 Ohm Widerstand dient dem Schutz des Controllers. Wenn nämlich sonst die Potentiometerstellung auf Maximum steht (0 Ohm), dann würde in den Eingang des Controllers ein viel zu hoher Strom fliessen und der AVR würde in Rauch aufgehen.

Dies ist meines Wissens die einzige Schaltung zur Erfassung von Analogwerten, welche mit nur einem einzigen Pin auskommt.
Mit einem weiteren Eingangspin und ein wenig Software können wir auch eine Kalibrierung realisieren, um den Messwert in einen vernünftigen Bereich (z.B: 0...100 % oder so) umzurechnen.

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

Poti.c (*http://www.mypage.bluewin.ch/ch_schifferle/AtmelC/Uebungen/Poti/Poti.c*)	Hauptprogramm.
Pot.c (*http://www.mypage.bluewin.ch/ch_schifferle/AtmelC/Uebungen/Poti/Pot.c*)	Separate Routine zur Ermittlung des Messwertes.
Pot.h (*http://www.mypage.bluewin.ch/ch_schifferle/AtmelC/Uebungen/Poti/Pot.h*)	Zugehörige Headerdatei.
UartPrintF.c (*http://www.mypage.bluewin.ch/ch_schifferle/AtmelC/Uebungen/Allgemein/UartPrintF.c*)	Für die Debugausgabe auf den UART.
UartPrintF.h (*http://www.mypage.bluewin.ch/ch_schifferle/AtmelC/Uebungen/Allgemein/UartPrintF.h*)	Zugehörige Headerdatei.
Makefile (*http://www.mypage.bluewin.ch/ch_schifferle/AtmelC/Uebungen/Poti/makefile*)	Makefile.

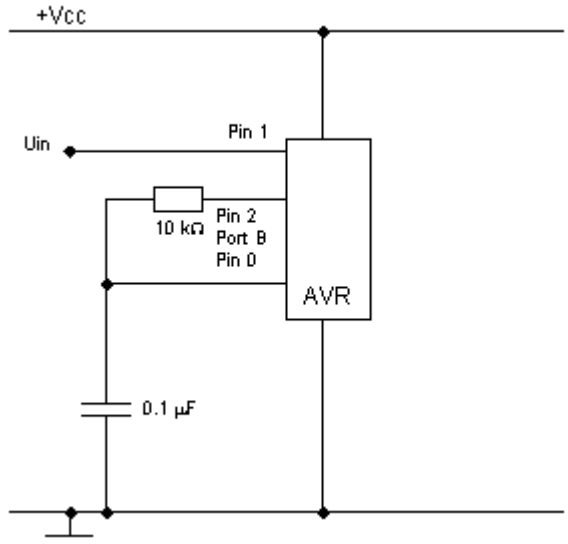
Nachdem das Programm auf den AVR geladen wurde, muss dieser kalibriert werden. Dazu wird der Kalibrierungsschalter geschlossen und das Poti einige Male zwischen minimaler und maximaler Stellung hin und her gedreht. Dabei werden die jeweiligen Maximalwerte bestimmt. Wenn der Kalibrierschalter wieder geöffnet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

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

ADC über Komparator

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

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



Das Prinzip der Messung ist nun dem vorhergehenden recht ähnlich. Durch Anlegen eines LOW-Pegels an Pin 2 wird der Kondensator zuerst einmal entladen. Auch hier muss darauf geachtet werden, dass der Entladevorgang genügend lang dauert.
Nun wird Pin 2 auf HIGH gelegt. Der Kondensator wird geladen. Wenn die Spannung über dem Kondensator die am Eingangspin anliegende Spannung erreicht hat schaltet der Komparator durch. Die Zeit, welche benötigt wird, um den Kondensator zu laden kann nun auch wieder als Maß für die Spannung an Pin 1 herangezogen werden.

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

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

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

Der ADC im AVR

Wenn es einmal etwas genauer sein soll, dann müssen wir auf einen AVR mit eingebautem **ADC-Wandler** zurückgreifen. Wir wollen hier einmal den AT90S8535 besprechen, welcher über 8 ADC-Kanäle verfügt. (TODO: nur ein Wandler, Mulitiplexbetrieb, nur eine Pin zur gleichen Zeit)

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.

Verfügt der AVR über eine interne Referenzspannung (Datenblatt typisch 2,56 oder 1,1V je nach AVR), bleibt der *Anschluss AREF* unbeschaltet und man stellt die ADC-Register so ein, dass die interne Referenzspannung als "AREF" genutzt wird. Ansonsten ist eine externe Referenzspannung von maximal **Vcc** an den Abschluss **AREF** anzulegen. 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, welche hier aufgelistet werden:

ADCSR

ADC Control and **S**tatus **R**egister.

In diesem Register stellen wir ein, wie wir den **ADC** verwenden möchten.
Das Register ist wie folgt aufgebaut:

Bit	7	6	5	4	3	2	1	0
Name	**ADEN**	**ADSC**	**ADFR**	**ADIF**	**ADIE**	**ADPS2**	**ADPS1**	**ADPS0**
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	W
Initialwert	0	0	0	0	0	0	1	0

ADEN (**ADC En**able)

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 (**AD**C **S**tart **C**onversion)

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 (**AD**C **Fr**ee Running Select)

Mit diesem Bit wird die Betriebsart eingestellt.
Eine logische 1 aktiviert den frei laufenden Modus. Der **ADC** misst nun ständig den ausgewählten Kanal und schreibt den gemessenen Wert in das **ADC Data Register**.

ADIF (**AD**C **I**nterrupt **F**lag)

Dieses Bit wird vom **ADC** gesetzt, wenn eine Umwandlung erfolgt und das **ADC Data Register** aktualisiert ist.
Wenn das **ADIE** Bit sowie das **I-Bit** im AVR **Statusregister** gesetzt ist, wird der **ADC Interrupt** ausgelöst und die Interrupt-Behandlungsroutine aufgerufen.
Das Bit wird automatisch gelöscht, wenn die Interrupt-Behandlungsroutine aufgerufen wird. Es kann jedoch auch gelöscht werden, indem eine logische 1 in das Register geschrieben wird (So steht es in der AVR-Dokumentation).

ADIE (**AD**C **I**nterrupt **E**nable)

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

ADPS2...ADPS0 (**AD**C **P**rescaler **S**elect 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-Taktfreqenz 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

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

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

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

ADCL

ADCH

ADC Data Register

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

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

oder

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

ADMUX

ADC Multiple**x**er Select Register

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

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

	MUX2...MUX0 Mit diesem 3 Bits wird der zu messende Kanal bestimmt. Es wird einfach die entsprechende Pinnummer des Ports 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. Meine Empfehlung ist deswegen klar diese, dass der frei laufende Betrieb nur bei einem einzelnen zu verwendenden Analogeingang verwendet werden sollte, wenn man sich Probleme bei der Umschalterei ersparen will.

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 AVRs auch 2,56V). D.h. das Eingangssignal darf diese Spannung nicht überschreiten, gegebenenfalls mit Spannungsteiler konditionieren. Ergebnis der Routine ist der ADC-Wert, also 0 für 0-Volt und 1024 für V_ref-Volt.

```
#define CHANNELOFFSET 4

uint16_t ReadChannel(uint8_t channel)
{
  uint8_t i;
  uint16_t result;

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

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

  /* 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<<ADIF)));     // auf Abschluss der Konvertierung
warten (ADIF-bit)

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

  ADCSRA &= ~(1<<ADEN);             // ADC deaktivieren (2)

  result >>= 2;             // Summe durch vier teilen = arithm. Mittelwert
```

```
  return result;
}
```

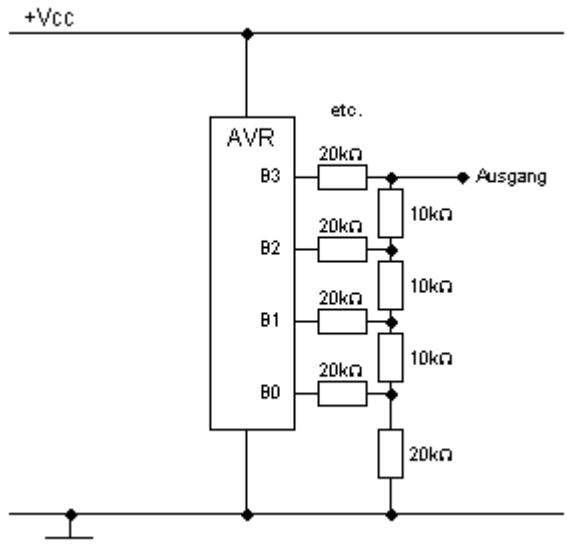
Im Beispiel wird bei jedem Aufruf der ADC aktiviert und nach der Wandlung wieder abgeschaltet, das spart Strom. Will man dies nicht, verschiebt man die mit (1) gekenntzeichneten Zeilen in eine Funktion adc_init() o.ä. und löscht die mit (2) markierten Zeilen.

DAC (Digital Analog Converter)

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

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:

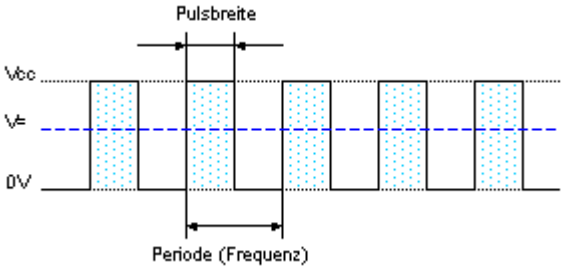


Es sollten selbstverständlich möglichst genaue Widerstände verwendet werden, also nicht unbedingt solche mit einer Toleranz von 10% oder mehr. Weiterhin empfiehlt es sich, je nach Anwendung den Ausgangsstrom über einen Operationsverstärker zu verstärken.

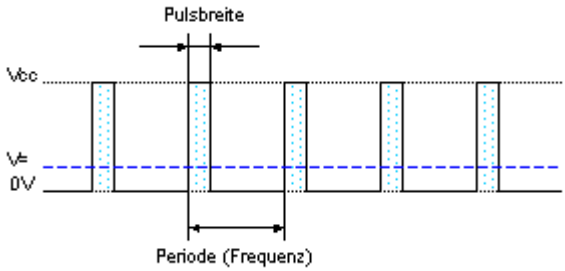
PWM (Pulsweitenmodulation)

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 **Vcc** anliegt, oder aber wir setzen den Ausgang auf LOW, wonach dann **0V** am Ausgang liegt.
Was passiert aber nun, wenn wir periodisch mit einer festen Frequenz zwischen HIGH und LOW umschalten?
Richtig, wir erhalten eine Rechteckspannung, wie die folgende Abbildung zeigt:



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



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

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

Hinweis:

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

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

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

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

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

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

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

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

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

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

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

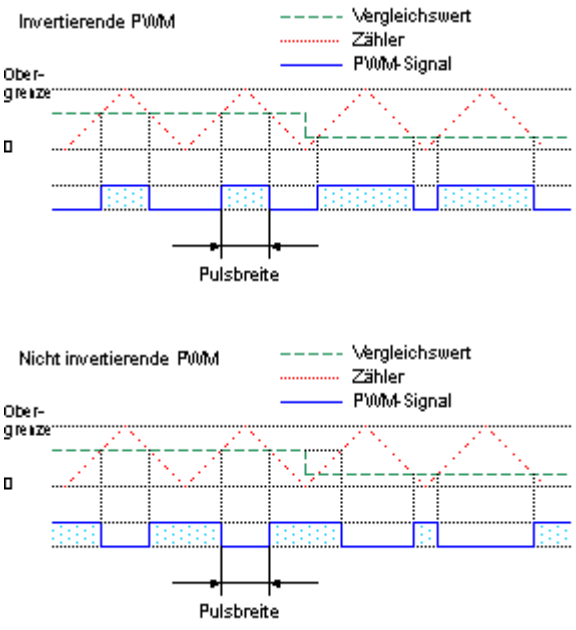
Also um einen Takt von CK / 1024 zu generieren, verwenden wir folgenden Befehl:

```
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 nur ein einzelnes **PWM**-Signal mit dieser Methode generieren.

Die Timer/Counter des AVR

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

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

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

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

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

Beide Timer/Counter werden im Timerbetrieb über den gleichen Vorzähler versorgt.

Der Vorzähler dient dazu, den CPU-Takt vorerst mal runter zu brechen auf eine wählbare Teilung. Die so geteilte Frequenz wird den Eingängen der Timer zugeführt.

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

8-Bit Timer/Counter

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

Der 8-Bit Timer wird über folgende Register angesprochen:

TCCR0

Timer/**C**ounter **C**ontrol **R**egister

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 (**C**lock **S**elect Bits)

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.

TCNT0

Timer/**Coun**ter Daten Register Timer **0**

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

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

Um nun also den Timer0 in Betrieb zu setzen und ihn mit einer Frequenz von 1/1024-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.

16-Bit Timer/Counter

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

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

- Die PWM-Betriebsart Erzeugung eines pulsweitenmodulierten 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:

TCCR1A

Timer/**C**ounter **C**ontrol **R**egister **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** (**Co**mpare **M**atch 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.

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

PWM11, **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.

TCCR1B

Timer/**C**ounter **C**ontrol **R**egister **B** Timer **1**

Bit	7	6	5	4	3	2	1	0
Name	ICNC1	ICES1	-	-	CTC1	CS12	CS11	CS10
R/W	R/W	R/W	R	R	R/W	R/W	R/W	R/W
Initialwert	0	0	0	0	0	0	0	0

ICNC1 (**I**nput **C**apture **N**oise **C**anceler (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 (**I**nput **C**apture **E**dge **S**elect 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 (**C**lear **T**imer/**C**ounter on Compare Match Timer/Counter **1**)

Wenn dieses Bit gesetzt ist, so wird nach einer Übereinstimmung des Datenregisters **TCNT1H/TCNT1L** mit dem Vergleichswert in **OCR1H/OCR1L** das Datenregister **TCNT1H/TCNT1L** auf 0 gesetzt.
Da die Übereinstimmung im Takt nach dem Vergleich behandelt wird, ergibt sich je nach eingestelltem Vorzähler ein etwas anderes Zählverhalten:
Wenn der Vorteiler auf 1 gestellt ist und C der jeweilige Zählerwert ist, dann nimmt das Datenregister, im CPU-Takt betrachtet, folgende Werte an:
... | C-2 | C-1 | C | 0 | 1 |...
Wenn der Vorteiler z.B. auf 8 eingestellt ist, dann nimmt das Datenregister folgende Werte an:
... | C-2, C-2, C-2, C-2, C-2, C-2, C-2, C-2 | C-1, C-1, C-1, C-1, C-1, C-1, C-1, C-1 | C, 0, 0, 0, 0, 0, 0, 0 |...
In der PWM-Betriebsart hat dieses Bit keine Funktion.

CS12, **CS11**, **CS10** (**C**lock **S**elect 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 TO, fallende Flanke
1	1	1	Externer Pin T0, steigende Flanke

Wenn als Quelle der externe Pin T0 verwendet wird, so wird ein Flankenwechsel auch erkannt, wenn der Pin T0 als Ausgang geschaltet ist.

TCNT1H
TCNT1L

Timer/**C**ou**nt**er 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	

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 Interrrupts gesperrt werden. Dann muss zuerst das **TCNT1H**-Register und erst danach das **TCNT1L**-Register geschrieben werden, also genau die umgekehrte Reihenfolge wie beim Lesen des Registers.

OCR1H
OCR1L

Timer/Counter **O**utput **C**ompare **R**egister Timer/Counter **1**

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

	Der Wert im Output Compare Register wird ständig mit dem aktuellen Wert im Datenregister TCNT1H/TCNT1L verglichen. Stimmen die beiden Werte überein, so wird ein sogenannter Output Compare Match ausgelöst. Die entsprechenden Aktionen werden über die Timer/Counter 1 Control und Status Register eingestellt. Zum Auslesen des Registers wird von der CPU ein internes TEMP-Register verwendet. Das gleiche Register wird auch verwendet, wenn auf **OCR1** oder **ICR1** zugegriffen wird. Deshalb müssen vor dem Zugriff auf eines dieser Register alle Interrupts gesperrt werden, weil sonst die Möglichkeit des gleichzeitigen Zugriffs auf das Temporärregister gegeben ist, was natürlich zu fehlerhaftem Verhalten des Programms führt.. Zudem muss zuerst **TCNT1L** und erst danach **TCNT1H** ausgelesen werden. Wenn in das Register geschrieben werden soll, müssen ebenfalls alle Interrupts gesperrt werden. Dann muss zuerst das **TCNT1H**-Register und erst danach das **TCNT1L**-Register geschrieben werden, also genau die umgekehrte Reihenfolge wie beim Lesen des Registers.
ICR1H **ICR1L**	Timer/Counter **I**nput **C**apture **R**egister Timer/Counter **1** **Bit** 7 6 5 4 3 2 1 0 **Name** **MSB** **ICR1H** **Name** **LSB** **ICR1L** **R/W** R R R R R R R R **Initialwert** 0 0 0 0 0 0 0 0 Das Input Capture Register ist ein 16-Bit Register mit Lesezugriff. Es kann nicht beschrieben werden. Wenn am Input Capture Pin **ICP** die gemäß Einstellungen im **TCCR1B** definierte Flanke erkannt wird, so wird der aktuelle Inhalt des Datenregisters **TCNT1H/TCNT1L** sofort in dieses Register kopiert und das Input Capture Flag **ICF1** im Timer Interrupt Flag Register **TIFR** gesetzt. Wie bereits oben erwähnt, müssen vor dem Zugriff auf dieses Register alle Interrupts gesperrt werden. Zudem müssen Low- und Highbyte des Registers in der richtigen Reihenfolge bearbeitet werden: Lesen: **ICR1L -> ICR1H** Schreiben: **ICR1H -> ICR1L**

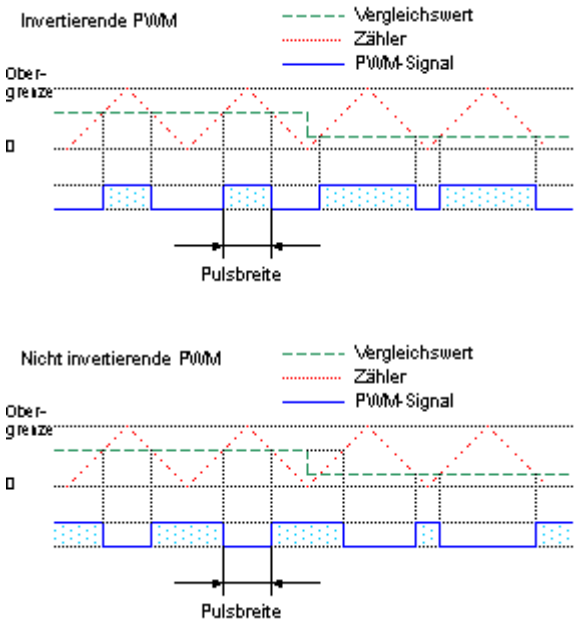
Die PWM-Betriebsart

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 Ausgabepin **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

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.

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.

TIMSK

Timer/Counter **I**nterrupt **M**a**sk**

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 (**T**imer/Counter **O**verflow **I**nterrupt **E**nable Timer/Counter **1**)

Wenn dieses Bit gesetzt ist, wird bei einem Überlauf des Datenregisters des Timer/Counter 1 ein Timer Overflow 1 Interrupt ausgelöst. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein.

OCIE1A (**O**utput **C**ompare Match **I**nterrupt **E**nable 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 (**T**imer/Counter **I**nput **C**apture **I**nterrupt **E**nable)

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 (**T**imer/Counter **O**verflow **I**nterrupt **E**nable Timer/Counter **0**)

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

TIFR

Timer/Counter **I**nterrupt **F**lag **R**egister

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 (**T**imer/Counter **O**verflow 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 (**O**utput **C**ompare **F**lag Timer/Counter **1**)

Dieses Bit wird gesetzt, wenn der aktuelle Wert des Datenregisters von Timer/Counter 1 mit demjenigen im Vergleichsregister **OCR1** übereinstimmt.
Das Flag wird automatisch gelöscht, wenn der zugehörige Interrupt-Vektor aufgerufen wird. Es kann jedoch auch gelöscht werden, indem eine logische 1 (!) in das entsprechende Bit geschrieben wird.

ICF1 (**I**nput **C**apture **F**lag 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 (**T**imer/Counter **O**verflow 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.

Sleep-Modes

AVR Controller verfügen über eine Reihe von sog. *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-Comperators negativ beeinflussen. Der Controller wird durch Interrupts aus dem Schlaf geweckt. Welche Interrups den jeweiligen Schlafmodus beenden, ist einer Tabelle im Datenblatt des jeweiligen Controllers zu entnehmen. Die Funktionen 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 sämtlichten AVR-Controllern unterstützt.

- **sleep_mode()**

Versetzt den Controller in den mit set_sleep_mode gewählten Schlafmodus.

```
#include <avr/io.h>
#include <avr/sleep.h>
...
  set_sleep_mode(SLEEP_MODE_PWR_DOWN);
   sleep_mode();

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

Vorsicht: Nicht alle AVR-Controller (v.a. nicht die "Brandneuen") werden von den avr-libc sleep-Funktionen richtig angesteuert. Bei nicht-untersützten Typen erreicht man die gewollte Funktionalität durch direkte "Bitmanipulation" der ensprechenden Register (vgl. Datenblatt) und Aufruf des Sleep-Befehls via Inline-Assembler:

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

- siehe auch: Dokumentation der avr-libc (*http://www.nongnu.org/avr-libc/user-manual/index.html*) Abschnitt Modules/Power Management and Sleep-Modes

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 ensprechende 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. Nachdem der Watchdog aktiviert und der gewünschte Vorteiler eingestellt wurde beginnt der Counter von 0 an hochzuzählen. Wenn nun die je nach Vorteiler eingestellte Anzahl Zyklen erreicht wurde, löst der Watchdog einen Reset aus. Um nun also im Normalbetrieb den Reset zu verhindern müssen wir den Watchdog regelmässig wieder neu starten bzw. Rücksetzen (Watchdog Reset). Dies sollte innerehalb unserer Hauptschleife passieren.

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

Das Watchdog Control Register:

WDTCR

Watchog **T**imer **C**ontrol **R**egister

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 (**W**atch**d**og **T**urn **O**ff **E**nable)

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 (**W**atch**d**og **E**nable)

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** (**W**atch**d**og Timer **P**rescaler 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 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 (z.B. WDTO_30MS).

- **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 Wachdog das geeignete Mittel dies zu verhindern. Weiterhin kann bei geschickter Programmierung der Watchdog dazu genutzt werden, bestimmte Stromsparfunktionen zu implementieren. 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.

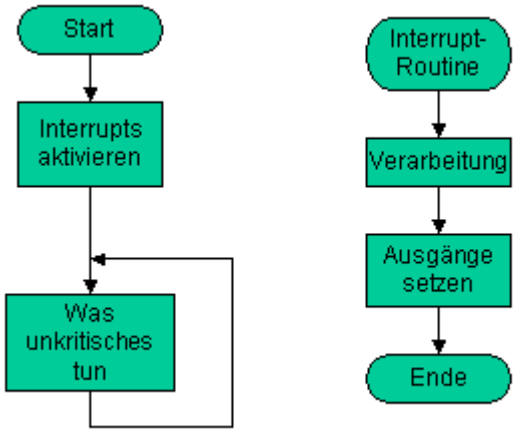
Es besteht die Möglichkeit herauszufinden, ob ein Reset durch den Watchdog ausgelöst wurde (beim ATmega16 z.B. Bit WDRF in MCUCSR). Diese Information sollte auch genutzt werden, falls ein WD-Reset in der Anwendung nicht planmässig implementiert wurde. Zum Beispiel kann man eine LED an einen freien Pin 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).

- siehe auch: Dokumentation der avr-libc (*http://www.nongnu.org/avr-libc/user-manual/index.html*) Abschnitt Modules/Watchdog timer handling

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.

Anforderungen an die Interrupt-Routine

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

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

Interrupt-Quellen

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

- Reset
- Externer Interrupt 0
- Externer Interrupt 1
- Timer/Counter 1 Capture Ereignis
- Timer/Counter 1 Compare Match
- Timer/Counter 1 Überlauf
- Timer/Counter 0 Überlauf
- UART Zeichen empfangen
- UART Datenregister leer
- UART Zeichen gesendet
- Analoger Komparator

Register

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

GIMSK

General **I**nterrupt **M**ask **R**egister.

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 **Int**errupt Request **1** Enable)

Wenn dieses Bit gesetzt ist, wird ein Interrupt ausgelöst, wenn am **INT1**-Pin eine steigende oder fallende (je nach Konfiguration im **MCUCR**) Flanke erkannt wird.
Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein.
Der Interrupt wird auch ausgelöst, wenn der Pin als Ausgang geschaltet ist. Auf diese Weise bietet sich die Möglichkeit, Software-Interrupts zu realisieren.

INT0 (External **Int**errupt 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.

GIFR

General **I**nterrupt **F**lag **R**egister.

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

INTF0 (External **Int**errupt 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.

MCUCR

MCU Control **R**egister.

Das MCU Control Register enthält Kontrollbits für allgemeine MCU-Funktionen.

Bit	7	6	5	4	3	2	1	0
Name	**-**	**-**	**SE**	**SM**	**ISC11**	**ISC10**	**ISC01**	**ISC00**
R/W	R	R	R/W	R/W	R/W	R/W	R/W	R
Initialwert	0	0	0	0	0	0	0	0

SE (**S**leep **E**nable)

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 (**S**leep **M**ode)

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** (**I**nterrupt **S**ense **C**ontrol **1** Bits)

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

ISC11	**ISC10**	**Bedeutung**
0	0	Low Level an **INT1** erzeugt einen Interrupt. In der Beschreibung heisst es, der Interrupt wird getriggert, solange der Pin auf 0 bleibt, also eigentlich unbrauchbar.
0	1	Reserviert
1	0	Die fallende Flanke an **INT1** erzeugt einen Interrupt.
1	1	Die steigende Flanke an **INT1** erzeugt einen Interrupt.

ISC01, **ISC00** (**I**nterrupt **S**ense **C**ontrol **0** Bits)

Diese beiden Bits bestimmen, ob die steigende oder die fallende Flanke für die Interrupterkennung am **INT0**-Pin ausgewertet wird.

ISC01	**ISC00**	**Bedeutung**
0	0	Low Level an **INT0** erzeugt einen Interrupt. In der Beschreibung heisst es, der Interrupt wird getriggert, solange der Pin auf 0 bleibt, also eigentlich unbrauchbar.
0	1	Reserviert
1	0	Die fallende Flank an **INT0** erzeugt einen Interrupt.
1	1	Die steigende Flanke an **INT0** erzeugt einen Interrupt.

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

Das Status-Register

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

Interrupts mit dem AVR GCC Compiler (WinAVR)

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

Damit diese Mittel zur Verfügung stehen, müssen wir die Includedatei *interrupt.h* und evtl. *signal.h* einbinden mittels:

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

// fuer SIGNAL() auch:
#include <avr/signal.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 Interrups 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 "unterbrechnungsfreier" 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 "unterbrechnungsfreier" 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

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

}
```

Nachdem nun die Interrupts aktiviert sind, braucht es selbstverständlich noch den auszuführenden Code, der ablaufen soll, wenn ein Interrupt eintrifft. Dazu gibt es zwei Dfinitionen: **SIGNAL** und **INTERRUPT**, welche allerdings AVR-GCC spezifisch sind und bei anderen Compilern womöglich anders heissen können.

SIGNAL

```
#include <avr/signal.h>
...
SIGNAL (siglabel)
{
    /* Interrupt Code */
}
```

Mit *SIGNAL* wird eine Funktion für die Bearbeitung eines Interrupts eingeleitet. Als Argument muss dabei die Benennung des entsprechenden Interruptvektoren angegeben werden. Diese sind in den jeweiligen Includedateien IOxxxx.h zu finden. Auf die korrekte Schreibweise ist zu achten, der Compiler prüft diese nicht (vgl. AVR-GCC). Als Beispiel ein Ausschnitt aus der Datei für den ATmega8 (bei WinAVR Standardinstallation in C:\WinAVR\avr\include\avr\iom8.h):

```
[...]
/* $Id: iom8.h,v 1.8 2003/02/17 09:57:28 marekm Exp $ */

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

/* Interrupt vectors */

#define SIG_INTERRUPT0         _VECTOR(1)
#define SIG_INTERRUPT1         _VECTOR(2)
#define SIG_OUTPUT_COMPARE2    _VECTOR(3)
#define SIG_OVERFLOW2          _VECTOR(4)
#define SIG_INPUT_CAPTURE1     _VECTOR(5)
#define SIG_OUTPUT_COMPARE1A   _VECTOR(6)
#define SIG_OUTPUT_COMPARE1B   _VECTOR(7)
#define SIG_OVERFLOW1          _VECTOR(8)
#define SIG_OVERFLOW0          _VECTOR(9)
#define SIG_SPI                _VECTOR(10)
#define SIG_UART_RECV          _VECTOR(11)
#define SIG_UART_DATA          _VECTOR(12)
#define SIG_UART_TRANS         _VECTOR(13)
#define SIG_ADC                _VECTOR(14)
#define SIG_EEPROM_READY       _VECTOR(15)
#define SIG_COMPARATOR         _VECTOR(16)
#define SIG_2WIRE_SERIAL       _VECTOR(17)
#define SIG_SPM_READY          _VECTOR(18)
```

Vor Nutzung von SIGNAL muss ebenfalls die Header-Datei signal.h eingebunden werden. Mögliche Funktionsrümpfe für solche Interruptfunktionen sind zum Beispiel:

```
#include <avr/interrupt.h>
#include <avr/signal.h>

SIGNAL (SIG_INTERRUPT0)
{
    /* Interrupt Code */
}

SIGNAL (SIG_OVERFLOW1)
{
    /* Interrupt Code */
}

SIGNAL (SIG_UART_RECV)
{
    /* Interrupt Code */
}

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

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.

INTERRUPT

```
#include <avr/interrupt.h>
...
INTERRUPT (signame)
{
    /* Interrupt Code */
}
```

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

Interruptroutinen (ISRs) sollten 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.

siehe auch: Hinweise in AVR-GCC

Datenaustausch mit Interrupt-Routinen

Variablen auf 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 wird und nach jedem Schreibzugriff in den Speicher geschrieben wird. Ansonsten könnte die Code-Optimierung "greifen" und der Wert der Variablen nur in Prozessorregistern zwischenspeichert werden, 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 <avr/signal.h>
#include <inttypes.h>
...

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

// hier z.B. Taste an Pin2 PortA "active low" = 0 wenn gedrueckt
```

```
#define KEY_PIN  PINA
#define KEY_PINNO PA2

// beachte: volatile!
volatile uint8_t gKeyCounter;

// Timer-Compare Interrupt z.B. alle 10ms
SIGNAL(SIG_OUTPUT_COMPARE1A)
{
   // 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 (gKeyCouter < 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 geschrieben 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 gedrueckt"
       }
       else {
          /* hier: Code fuer "Taste kurz gedrueckt"
       }
...
}
```

Bei Variablen größer ein Byte muss darauf geachtet werden, dass die Zugriffe auf die einzelnen Bytes ausserhalb der ISR nicht durch einen Interrupt unterbrochen werden. (Allgemeinplatz: AVRs sind 8-bit Controller). Zur Veranschaulichung ein Codefragment:

```
...
volatile uint16_t gMyCounter16bit
...
SIGNAL(...)
{
...
   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 Kopieren des zweiten Bytes
   // ein Interrupt auftritt der den Inhalt von MyCounter veraendert
   tmpCnt = gMyCounter16bit;
```

```
   // besser: Aenderungen "ausserhalb" verhindern, alle "Teilbytes"
   // bleiben konsistent
   cli();  // Interupts deaktivieren
   tmpCnt = gMyCounter16Bit;
   sei();  // wieder aktivieren
...
}
```

- siehe auch: Dokumentation der avr-libc (*http://www.nongnu.org/avr-libc/user-anual/index.html*) Frequently asked Questions/Fragen Nr. 1 und 8 (Stand: avr-libc ers. 1.0.4)

Was macht das Hauptprogramm

Im einfachsten Fall gar nichts mehr.

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

```
...
    for (;;) {}
...
```

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

Wie wir im bisherigen Kursverlauf gesehen haben ist es ohnehin mit so schnellen Controller meistens gar nicht unbedingt notwendig mit Interruptfunktionen zu arbeiten.
Es ist allerdings auch zu bemerken, dass mit den Interruptroutinen ein Programm sehr schön strukturiert werden kann, wenn man es richtig macht.

- siehe auch: Dokumentation der avr-libc (*http://www.nongnu.org/avr-libc/user-anual/index.html*) Abschnitt Modules/Interrupts and Signals

Speicherzugriffe

Atmel AVR-Controller verfügen typisch über drei Speicher:

- RAM: Im RAM (eigentlich SRAM) wird vom gcc-Compiler Platz für Variablen eserviert. Auch der Stack befindet sich im RAM. Dieser Speicher ist "flüchtig", d.h. er Inhalt der Variablen geht beim Ausschalten oder einem Zusammenbruch der pannungsversorgung verloren.

- Programmspeicher: Ausgeführt als FLASH-Speicher, seitenweise wiederbeschreibbar. arin ist das Anwendungsprogramm abgelegt.

- EEPROM: Nichtflüchtiger Speicher, d.h. der einmal geschriebene Inhalt bleibt auch hne Stromversorgung erhalten. Byte-weise schreib/lesbar. Im EEPROM werden ypischerweise gerätespezifische Werte wie z.B. Kalibrierungswerte von Sensoren bgelegt.

Einige AVRs besitzen keinen RAM-Speicher, lediglich die Register können als "Arbeitsvariablen" genutzt werden. Da die Anwendung des avr-gcc auf solch "kleinen" Controllern ohnehin wenig Sinn macht 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.

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.

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). 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 geschrieben und 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,
pgmFooByteArray1 };
...

void foo(void)
{
  static /*const*/ uint8_t pgmTestByteLocal PROGMEM = 0x55;
  static /*const*/ char pgmTestStringLocal[] PROGMEM = "im Flash";
```

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.

```
    // 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.

```
    uint16_t myWord;

    myWord = pgm_read_word(&pgmFooWord);
```

Zeiger auf Werte im Flash sind ebenfalls 16 Bits "gross". (Die avr-libc pgmspace-Funktionen unterstützen nur die unteren 64kB Flash bei Controllern mit mehr als 64kB.) Pointer müssen gegebenenfalls "gecastet" werden.

```
    uint8_t *ptrToArray;

    ptrToArray = (uint8_t*)(pgm_read_word(&pgmPointerToArray1));
    // ptrToArray zeigt nun auf das erste Element des Byte-Arrays
pgmPointerToArray1

    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 zeigt nun auf das erste Element des Byte-Arrays
pgmPointerToArray2
    // da im zweiteb Element des Pointer-Arrays pgmPointerArray die Adresse
    // von pgmPointerToArray2 abgelegt ist

    for (i=0;i<3;i++) {
        myByte = pgm_read_byte(ptrToArray+i);
        // mach' was mit myByte... (30, 7, 79)
    }
```

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-Addresse 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.)

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 (!strncmp_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 "textInFlashProblem" 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 weiters 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 abglegt 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.

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 (wieauchimmmer.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 AVRs 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. 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 Verabeitung 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_an_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_an_uart(s[i]) oder zeichen_an_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 (*http://www.nongnu.org/avr-libc/user-anual/index.html*) Abschnitte Modules/Program Space String Utilities und Abschnitt odules/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 bestimme 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. Die Nutzung einer C-Präprozessor-Ersetzung bringt etwas Bequemlichkeit. Siehe dazu folgendes Beispiel.

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

// alle Textstellen EEPROM im Quellcode durch __attribute__ ... ersetzen
#define EEPROM  __attribute__ ((section (".eeprom")))

...

/* Byte */
uint8_t eeFooByte EEPROM = 123;

/* Wort */
uint16_t eeFooWort EEPROM = 12345;

/* float */
float eeFooFloat EEPROM;

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

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

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
...
    myByte = eeprom_read_byte(&eeFooByteArray1[1]);
    // myByte hat nun den Wert 3
...
```

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-Addresse und die Länge des Datenblocks in Bytes (size_t).

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

```
...
    unit8_t  myByteBuffer[3];
    uint16_t myWordBuffer[4];

    /* Datenblock aus EEPROM LESEN  */

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

    /* dito etwas anschaulicher aber "unnuetze Tipparbeit": */
```

```
    eeprom_read_block(&myByteBuffer[0],&eeFooByteArray[0],3);

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

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

    /* Datenlock in EEPROM SCHREIBEN */
    eeprom_write_block(myByteBuffer,eeFooByteArray1,sizeof(myByteBuffer));
    eeprom_write_block(myWordBuffer,eeFooWordArray,sizeof(myWordBuffer));
...
```

"Nicht-Integer"-Datentypen wie z.B. Fliesskommazahlen lassen sich recht praktisch ueber 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 n[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];
    unin8_t   rom_code[8];
} tMyStruct;

#define MAXSENSORS 3
tMyStruct eeMyStruct[MAXSENSORS] EEPROM;

...

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

Eine besondere Funktion des avr-gcc ist, dass mit einem entsprechenden makefile aus den Initialisierungswerten der Variablen im Quellcode eine Datei erzeugt werden kann, die man auf den Controller programmieren kann (.eep-Datei). Damit können sehr elegant Standardwerte für den EEPROM-Inhalt im Quellcode definiert werden. Die Vorgehensweise wird aus dem WinAVR-Beispielmakefile ersichtlich. Siehe dazu die Erläuterungen im Abschnitt Exkurs: Makefiles.

Bekannte Probleme bei den EEPROM-Funktionen

Vorsicht: Nicht alle neuen AVR Controller werden von avr-libc/eeprom.h untersützt (Stand Version 1.0.4). Insbesondere beim ATMega169 funktionieren die Funktionen nicht korrekt (Ursache: unterschiedliche Speicheradressen der EEPROM-Register) Etwas ältere Typen, oder zu den "etablierten" Controllern kompatible, bereiten jedoch hier keine Proleme. Im Zweifel hilft ein Blick in den vom Compiler erzeugten Assembler-Code (lst/lss-Dateien).

In jedem Datenblatt zu AVR-Controllern mit EEPROM sind kurze Beispielecodes für den Schreib- und Lesezugriff enthalten. Der dort gezeigt Code kann direkt auch mit dem avr-gcc (ohne avr-libc/eeprom.h) genutzt werden ("copy/paste", gegebenfalls Schutz vor Unterbrechnung/Interrupt ergänzen *uint8_t sreg; sreg=SREG; [EEPROM-Code] ; SREG=sreg; return;*, siehe Abschnitt Interrupts).

- siehe auch: Dokumentation der avr-libc (*http://www.nongnu.org/avr-libc/user-anual/index.html*) Abschnitt Modules/EEPROM handling

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 "gcc-Toolchain" bietet dazu zwei Möglichkeiten:

Inline-Assembler:

Die Assembleranweisungen werden in direkt in den C-Code integriert. Eine Quellcode-Datei enhält somit C- und Assembleranweisungen

Assembler-Dateien:

Der Assembler-Codecode befindet sich in eigenen Quellcodedateien. Dieser 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**-O**P**eration. Dieser Assembler-Befehl benötigt genau einen Taktzyklus, ansonsten "tut sich nichts". Sinnvolle Anwendungen für NOP sind genaue Delay(=warte)-Funktionen.

```
...
/* Verzoegern der weiteren Programmausfuehrung 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 vermeindlich 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 nützliche "Assembler-Einzeiler" ist der Auruf von sleep (*asm volatile ("sleep"::);*), da hierzu keine eigene Funktion in der avr-libc existiert.

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 des 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 L_Exit_%=       \n\t"  \
                        "L_LOOP_%=:           \n\t"  \
                        "sbiw %0,1            \n\t"  \
                        "brne L_LOOP_%=       \n\t"  \
                        "L_Exit_%=:           \n\t"  \
                        : "=w" (count)
                       : "0"  (count)
                     );
}
```

- Jede Anweisung wird mit **\n\t** abgeschlossen, alle Zeilen mit **** zu einer "langen eichenkette" verbunden.
- Sprung-Marken (labels) werden mit einem Prozentzeichen abgeschlossen, der räprozessor (Assembler?) setzt an dieser Stelle eine laufende Nummer ein, die oppelbezeichnungen bei mehrmaliger Verwendung wird somit verhindert.

Das Resultat zeigt ein Blick in die .lss-Datei. Aus delayloop16(20); erzeugt der Preprozessor/Assembler/Linker z.B.:

```
 13a:   84 e1           ldi     r24, 0x14       ; 20
 13c:   90 e0           ldi     r25, 0x00       ; 0
 13e:   81 15           cp      r24, r1
 140:   91 05           cpc     r25, r1
 142:   11 f0           breq    .+4             ; 0x148
00000144 <L_LOOP_121>:
 144:   01 97           sbiw    r24, 0x01       ; 1
 146:   f1 f7           brne    .-4             ; 0x144
00000148 <L_Exit_121>:
```

Detaillierte Ausführungen zum Thema Inline-Assembler finden sich in der Dokumentation der avr-libc.

- siehe auch: Dokumentation der avr-libc/Related Pages/Inline Asm
- AVR Assembler-Anweisungsliste (*http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf*)

Assembler-Dateien

Assembler-Dateien erhalten die Endung .S (*grosses* S) und werden im makefile nach WinAVR/mfile-Vorlage hinter *ASRC*= durch Leerzeichen getrennt aufgelistet.

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 Rueckgabe
.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 uebergebenen 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 zurueckgeben **
// Rueckgabewerte 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 zurueckgeben **
// 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.
// Rueckgabewert 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:

```
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 uebergebenen 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 zurueckgeben
.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 zurueckgeben
.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 (*http://www.nongnu.org/avr-libc/user-manual/assembler.html*)
- avr-libc-Dokumentation: Related Pages/FAQ/"What registers are used by the C compiler?" (*http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq_reg_usage*)