

AVR-GCC-Tutorial

Inhaltsverzeichnis [[Verbergen](#)]

[1 Vorwort](#)

[2 Benötigte Werkzeuge](#)

[3 Was tun, wenn's nicht "klappt"?](#)

[4 Exkurs: makefiles](#)

[4.1 Controllertyp setzen](#)

[4.2 Quellcode-Dateien einstellen](#)

[4.3 Programmiergerät einstellen](#)

[4.4 Anwendung](#)

[4.5 Sonstige Einstellungen](#)

[4.5.1 Optimierungsgrad](#)

[4.5.2 Debug-Format](#)

[4.5.3 Assembler-Dateien](#)

[4.5.4 Taktfrequenz](#)

[4.6 Eingabedateien zur Simulation in AVR-Studio](#)

[5 Ganzzahlige \(Integer\) Datentypen](#)

[5.1 Selbstdefinierte \(nicht-standardisierte\) ganzzahlige Datentypen](#)

[5.2 Standardisierte Integer\(Ganzzahl\)-Typen](#)

[6 Bitfelder](#)

[7 Grundsätzlicher Programmaufbau eines \$\mu\$ C-Programms](#)

[7.1 Sequentieller Programmablauf](#)

[7.2 Interruptgesteuerter Programmablauf](#)

[8 Allgemeiner Zugriff auf Register](#)

[8.1 I/O-Register](#)

[8.1.1 Lesen eines I/O-Registers](#)

[8.1.1.1 Lesen eines Bits](#)

[8.1.2 Schreiben eines I/O-Registers](#)

[8.1.2.1 Schreiben von Bits](#)

[8.1.3 Warten auf einen bestimmten Zustand](#)

[9 Zugriff auf Ports](#)

[9.1 Datenrichtung bestimmen](#)

[9.1.1 Ganze Ports](#)

[9.2 Vordefinierte Bitnummern für I/O-Register](#)[9.3 Digitale Signale](#)[9.4 Ausgänge](#)[9.5 Eingänge \(Wie kommen Signale in den \$\mu\$ C\)](#)[9.5.1 Signalkopplung](#)[9.5.2 Tasten und Schalter](#)[9.5.2.1 Pull-Up Widerstände aktivieren](#)[9.5.2.2 \(Tasten-\)Entprellung](#)[9.6 Analog](#)[9.7 16-Bit Portregister \(ADC, ICR1, OCR1, TCNT1, UBRR\)](#)[10 Der UART](#)[10.1 Allgemeines zum UART](#)[10.2 Die Hardware](#)[10.3 Senden mit dem UART](#)[10.3.1 Senden einzelner Zeichen](#)[10.3.2 Schreiben einer Zeichenkette \(String\)](#)[10.4 Zeichen Empfangen](#)[10.5 Software-UART](#)[11 Analoge Ein- und Ausgabe](#)[11.1 ADC \(Analog Digital Converter\)](#)[11.1.1 Der interne ADC im AVR](#)[11.1.1.1 Die Register des ADC](#)[11.1.1.2 Aktivieren des ADC](#)[11.1.2 Analog-Digital-Wandlung ohne internen ADC](#)[11.1.2.1 Messen eines Widerstandes](#)[11.1.2.2 ADC über Komparator](#)[11.2 DAC \(Digital Analog Converter\)](#)[11.2.1 DAC über mehrere digitale Ausgänge](#)[11.2.2 PWM \(Pulsweitenmodulation\)](#)[12 Die Timer/Counter des AVR](#)[12.1 Der Vorteiler \(Prescaler\)](#)[12.2 8-Bit Timer/Counter](#)[12.3 16-Bit Timer/Counter](#)[12.3.1 Die PWM-Betriebsart](#)[12.3.2 Vergleichswert-Überprüfung](#)[12.3.3 Einfangen eines Eingangssignals \(Input Capturing\)](#)[12.4 Gemeinsame Register](#)[13 Sleep-Modes](#)[14 Der Watchdog](#)

[14.1 Wie funktioniert nun der Watchdog](#)

[14.2 Watchdog-Anwendungshinweise](#)

[15 Programmieren mit Interrupts](#)

[15.1 Anforderungen an Interrupt-Routinen](#)

[15.2 Interrupt-Quellen](#)

[15.3 Register](#)

[15.4 Allgemeines über die Interrupt-Abarbeitung](#)

[15.5 Interrupts mit dem AVR GCC Compiler \(WinAVR\)](#)

[15.5.1 SIGNAL](#)

[15.5.2 INTERRUPT](#)

[15.6 Datenaustausch mit Interrupt-Routinen](#)

[15.7 Interrupt-Routinen und Registerzugriffe](#)

[15.8 Was macht das Hauptprogramm?](#)

[16 Speicherzugriffe](#)

[16.1 RAM](#)

[16.2 Programmspeicher \(Flash\)](#)

[16.2.1 Byte lesen](#)

[16.2.2 Wort lesen](#)

[16.2.3 Floats und Structs lesen](#)

[16.2.4 Vereinfachung für Zeichenketten \(Strings\) im Flash](#)

[16.2.5 Flash in der Anwendung schreiben](#)

[16.2.6 Warum so kompliziert?](#)

[16.3 EEPROM](#)

[16.3.1 Bytes lesen/schreiben](#)

[16.3.2 Wort lesen/schreiben](#)

[16.3.3 Block lesen/schreiben](#)

[16.3.4 EEPROM-Speicherabbild in .eep-Datei](#)

[16.3.5 Bekannte Probleme bei den EEPROM-Funktionen](#)

[17 Assembler und Inline-Assembler](#)

[17.1 Inline-Assembler](#)

[17.2 Assembler-Dateien](#)

[17.3 Globale Variablen für Datenaustausch](#)

[17.3.1 Globale Variablen im Assemblerfile anlegen](#)

[17.3.2 Variablen größer als 1 Byte](#)

[18 Anhang](#)

[18.1 Anhang A - Externe Referenzspannung des internen Analog-Digital-Wandlers](#)

[19 TODO](#)

Vorwort

Dieses Tutorial soll den Einstieg in die Programmierung von Atmel [AVR](#)-Mikrocontroller in der Programmiersprache [C](#) mit dem freien ("kostenlosen") C-Compiler avr-gcc erleichtern.

Die ursprüngliche Version stammt von Christian Schifferle, viele neue Abschnitte und aktuelle Anpassungen von [Benutzer:Mthomas](#). Viele der im Original-Dokument verwendeten Funktionen (z.B. `sbi()`, `cbi()`, `outp()`, `inp()`) sind in aktuellen Versionen des [avr-gcc](#) C-Compilers und der Laufzeitbibliothek `avr-libc` nicht mehr enthalten. Dieses Tutorial wurde an die neuen Funktionen/Methoden angepasst.

Eine Version als PDF zum Ausdrucken ist hier erhältlich: [http://www.siwawi.arubi.uni-kl.de/avr_projects/AVR-GCC-Tutorial - www mikrocontroller net.pdf](http://www.siwawi.arubi.uni-kl.de/avr_projects/AVR-GCC-Tutorial_-_www_mikrocontroller_net.pdf)

In diesem Text wird häufig auf die Standardbibliothek für den `avr-gcc`-Compiler, die `avr-libc`, verwiesen. Die online-Version der **avr-libc Dokumentation** findet sich [hier](http://www.nongnu.org/avr-libc/user-manual/index.html) (<http://www.nongnu.org/avr-libc/user-manual/index.html>). Bei [WinAVR](#) gehört die Dokumentation zum Lieferumfang und wird mitinstalliert.

Benötigte Werkzeuge

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

- Testboard für die Aufnahme eines AVR Controllers, der vom `avr-gcc` Compiler unterstützt wird (alle ATmegas und die meisten AT90, siehe Dokumentation der `avr-libc` für unterstützte Typen). Dieses Testboard kann durchaus auch selbst gelötet oder auf einem Steckbrett aufgebaut werden. Einige Registerbeschreibungen dieses Tutorials beziehen sich auf den inzwischen veralteten AT90S2313. Der weitaus größte Teil des Textes ist aber für alle Controller der AVR-Familie gültig. Brauchbare Testplattform sind auch das [STK500](#) und der [AVR Butterfly](#) von Atmel.
- Der `avr-gcc` Compiler und die `avr-libc`. Kostenlos erhältlich für nahezu alle Plattformen und Betriebssysteme. Für MS-Windows im Paket [WinAVR](#); für Unix/Linux siehe auch Hinweise im Artikel [AVR-GCC](#).
- Programmiersoftware und -hardware z.B. PonyProg (siehe auch: [Pony-Prog Tutorial](#)) oder [AVRDUDE](#) mit [STK200](#)-Dongle oder die von Atmel verfügbare Hard- und Software ([STK500](#), Atmel AVRISP, [AVR-Studio](#)).
- Nicht unbedingt erforderlich, aber zur Simulation und zum Debuggen unter MS-Windows recht nützlich: [AVR-Studio](#) (siehe Abschnitt [Exkurs: makefiles](#)).

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) (<http://www.avrfreaks.net>) nach vergleichbaren Problemen absuchen.
- Das [Archiv der avr-gcc Mailing-Liste](#) (<http://lists.gnu.org/archive/html/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 möglichst 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 integrierter 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 Plattformen 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.

<code>make all</code>	Erstellt aus den im Makefile angegebenen Quellcodes eine <i>hex</i> -Datei (und ggf. auch <i>eep</i> -Datei).
<code>make program</code>	Überträgt die <i>hex</i> -Datei (und wahlweise auch die <i>eep</i> -Datei für den EEPROM) zum AVR.
<code>make clean</code>	löscht alle temporären Dateien, also auch die <i>hex</i> -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](#)), 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` unterstützten 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 lwire.c
...
```

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

```
SRC = $(TARGET).c uart.c lwire.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

# nicht 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 wird 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*

Taktfrequenz

Neuere Versionen der WinAVR/Mfile Vorlage für Makefiles beinhalten die Definition einer Variablen *F_CPU* (WinAVR 2/2005). Darin wird die Taktfrequenz des Controllers in Hertz eingetragen. Die Definition steht dann im gesamten Projekt ebenfalls unter der Bezeichnung *F_CPU* zur Verfügung (z.B. um daraus UART-, SPI- oder ADC-Frequenzeinstellungen abzuleiten).

Die Angabe hat rein "informativen" Charakter, die tatsächliche Taktrate wird über den externen Takt (z.B. Quarz) bzw. die Einstellung des internen R/C-Oszillators bestimmt. Die Nutzung von *F_CPU* macht also nur Sinn, wenn die Angabe mit dem tatsächlichen Takt übereinstimmt.

Innerhalb der *avr-libc* ab Version 1.2 (in WinAVR ab 2/2005 enthalten) wird die Definition der Taktfrequenz (*F_CPU*) zur Berechnung der Wartefunktionen in *avr/delay.h* genutzt. Diese funktionieren nur dann korrekt, wenn *F_CPU* mit der tatsächlichen Taktfrequenz übereinstimmt. *F_CPU* muss dazu jedoch nicht unbedingt im makefile definiert werden. Es reicht aus, wird aber bei mehrfacher Anwendung unübersichtlich, vor *#include <avr/delay.h>* ein *#define F_CPU [hier Takt in Hz]UL* einzufügen. Vgl. dazu den [entsprechenden Abschnitt der Dokumentation](http://www.nongnu.org/avr-libc/user-manual/group__avr__delay.html) (http://www.nongnu.org/avr-libc/user-manual/group__avr__delay.html).

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 Mikrocontrollern 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 Version der `avr-libc` (ab V1.2.0) heisst die Datei entsprechend aktuellem Standard `stdint.h`. `inttypes.h` existiert aus Kompatibilitätsgründen 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) (<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 eine einzelne 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;
```

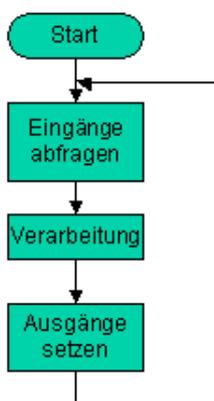
Bitfelder sparen zwar Platz, verschlechtern aber unter Umständen die Les- und Wartbarkeit des Codes. Anfängern wird geraten, ein "ganzes" Byte (uint8_t) zu nutzen auch wenn nur ein Bitwert gespeichert werden soll.

Grundsätzlicher Programmaufbau eines μ C-Programms

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

Sequentieller Programmablauf

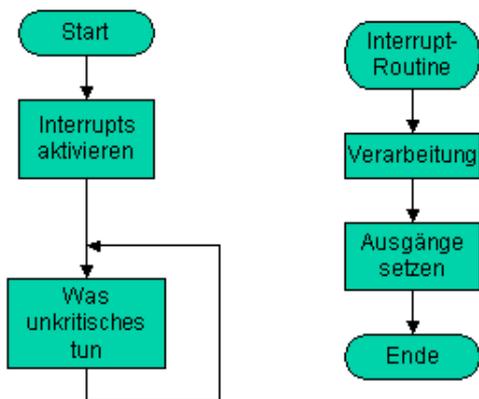
Bei dieser Programmiermethode wird eine Endlosschleife programmiert, welche im Wesentlichen immer den gleichen Aufbau hat:



Interruptgesteuerter Programmablauf

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

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



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 sowohl auslesen als auch beschreiben.

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

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

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

Ist im Makefile der MCU-Typ 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. In Quellcodes, die für ältere Versionen des avr-gcc/der avr-libc entwickelt wurden, erfolgt der Lesezugriff über die Funktion inp(). inp() ist in aktuellen Versionen der avr-libc nicht mehr enthalten und auch nicht mehr erforderlich.

Beispiel:

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

uint8_t foo;

...

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

Lesen eines Bits

Die AVR-Bibliothek (avr-libc) 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 erforderlich*, man kann auch "einfache" C-Syntax verwenden, die universell verwendbar ist. *bit_is_set* entspricht dabei z.B. (Registername & (1 << Bitnummer)). Das Ergebnis ist <>0

("wahr") wenn das Bit gesetzt und 0 ("falsch") wenn es nicht gesetzt ist. Die Negierung des Ausdrucks, also `!(Registername & (1 << Bitnummer))`, entspricht `bit_is_clear` und gibt einen Wert `<>0` ("wahr") zurück, falls das Bit nicht gesetzt ist.

- siehe auch [Bitmanipulation](#)

Schreiben eines I/O-Registers

Zum Schreiben kann man I/O-Register einfach wie eine Variable setzen. In Quellcodes, die für ältere Versionen des `avr-gcc/der avr-libc` entwickelt wurden, erfolgt der Schreibzugriff über die Funktion `outp()`. `outp()` ist in aktuellen Versionen der `avr-libc` nicht mehr enthalten und auch nicht mehr erforderlich.

Beispiel:

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

...

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

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

Schreiben von 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 gelöscht
```

Das niederwertigste Bit (für 1) eines Bytes hat die Bitnummer 0, das "höchstwertige" (für 128) die Nummer 7.

Beispiel:

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

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

Beispiel:

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

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

Siehe auch:

- [Bitmanipulation](#)
- [Dokumentation der avr-libc](http://www.nongnu.org/avr-libc/user-manual/index.html) (<http://www.nongnu.org/avr-libc/user-manual/index.html>) 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 Programmieretechnik, 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 (zurücksetzen des Watchdogtimers).

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

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

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

#define WARTEPIN PINA
#define WARTEBIT 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.

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

/* 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
// gesetzt (1) ist
while ( WARTEPIN & (1<<WARTEBIT) ) ;
...
```

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

siehe auch:

- [Dokumentation der avr-libc](http://www.nongnu.org/avr-libc/user-manual/index.html) (<http://www.nongnu.org/avr-libc/user-manual/index.html>) 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 Portx. 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.
PINx	Eingangsadresse für Portx. Zustand des Ports. Die Bits in PINx entsprechen dem Zustand der als Eingang definierten Portpins. Bit 1 wenn Pin "high", Bit 0 wenn Portpin low.
PORTx	Datenregister für Portx. Dieses Register wird verwendet, um die Ausgänge eines Ports anzusteuern. Bei Pins, die mittels DD Rx auf Eingang geschaltet wurden, koenen ueber PORTx die internen Pull-Up Widerstände aktiviert oder deaktiviert werden (1 = aktiv).

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. Dazu muss die Header-Datei avr/io.h eingebunden sein (darin wird u.a. DDRB definiert).

Vordefinierte Bitnummern für I/O-Register

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

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

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

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

Digitale Signale

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

Ausgänge

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

Mit dem Befehl

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

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

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

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

```
PORTB |= (1<<PB2);

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

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

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

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

Falls der Anfangszustand von Ausgängen kritisch ist, muss die Reihenfolge beachtet werden, mit der die Datenrichtung (DDRx) eingestellt und der Ausgabewert (PORTx) gesetzt wird. Für Ausgangspins, die mit Ausgangswert "high" initialisiert werden, zuerst die Bits im PORTx-Register setzen und anschliessend die Datenrichtung auf Ausgang stellen. Daraus ergibt sich die Abfolge Eingang -> setze PORTx: interner Pull-Up aktiv -> setze DDRx: Ausgang "high". Bei der Reihenfolge erst DDRx and PORTx, kann es zu einem kurzen "low-Puls" kommen, der auch externe Pull-Up-Widerstände "überstimmt". Die (ungünstige) Abfolge: Eingang -> setze DDRx: Ausgang (auf "low", da PORTx nach Reset 0) -> setze PORTx: Ausgang auf high. Vergleiche dazu auch das Datenblatt Abschnitt *Configuring the Pin*.

Eingänge (Wie kommen Signale in den μ C)

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

Signalkopplung

Am einfachsten ist es, wenn die Signale direkt aus einer anderen digitalen Schaltung übernommen werden können. Hat der Ausgang der entsprechenden Schaltung TTL-Pegel dann können wir sogar direkt den Ausgang der Schaltung mit einem Eingangspin von unserem Controller verbinden.

Hat der Ausgang der anderen Schaltung keinen TTL-Pegel so müssen wir den Pegel über entsprechende Hardware (z.B. Optokoppler, Spannungsteiler "Levelshifter") anpassen.

Die Masse der beiden Schaltungen muss selbstverständlich miteinander verbunden werden. Der Software selber ist es natürlich letztendlich egal, wie das Signal eingespeist wird. Wir können ja ohnehin lediglich prüfen, ob an einem Pin unseres Controllers eine logische 1 (Spannung größer ca. $0,7 \cdot V_{cc}$) oder eine logische 0 (Spannung kleiner ca. $0,2 \cdot V_{cc}$) anliegt. Detaillierte Informationen darüber, ab welcher Spannung ein Eingang als 0 ("low") bzw. 1 ("high") erkannt wird, liefert die Tabelle DC Characteristics im Datenblatt des genutzten Controllers.

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

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

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

```
#include <avr/io.h>
#include <stdint.h>
...
uint8_t bPortD;
...
bPortD = PIND;
...

```

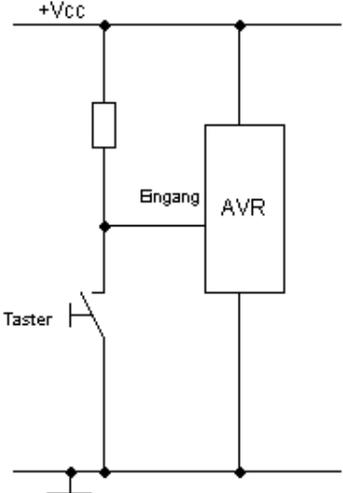
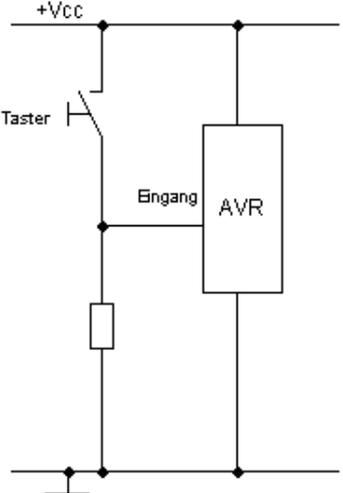
Mit den C-Bitoperationen kann man den Status der Bits abfragen.

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

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

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

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.

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

```
#include <avr/io.h>
...
DDRD = 0x00; /* alle Pins von Port D als Eingang */
PORTD = 0xff; /* interne Pull-Ups an allen Port-Pins aktivieren */
...
DDRC  &= ~(1<<DDC7); /* Pin PC7 als Eingang */
PORTC |= (1<<PC7);   /* internen Pull-Up an PC7 aktivieren */
```

(Tasten-)Entprellung

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

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

```
#include <avr/io.h>
#include <inttypes.h>
#ifndef F_CPU
#define F_CPU 3686400UL /* Quarz mit 3.6864 Mhz */
#endif
#include <avr/delay.h> /* definiert _delay_ms() ab avr-libc Version 1.2.0 */

/* 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 ); /* 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.

Zum Thema Entprellen siehe auch:

- Artikel [Entprellung](#)

Analog

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

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

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

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

foo=ADC; /* setzt die Wort-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>
#include <stdint.h>
#ifndef F_CPU
#define F_CPU 3686400
#endif
#define UART_BAUD_RATE 9600

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

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

/*alternativ:*/
#include <inttypes.h>

uint16_t wFoo16;
uint8_t bFooLow, bFooHigh;

...

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

...

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.

Bei einigen 16-bit Registern (insbesondere die der 16-bit Timer) erfolgen Schreibzugriffe über das sogenannte *temporary Register*. Die Reihenfolge der Zugriffe bestimmt, wann der Wert tatsächlich ins Register geschrieben wird. Typisch wird erst das High-Byte beschrieben (xxxH) und intern im temporary Register zwischengespeichert. Nachdem das Low-Byte (xxxL) geschrieben wurde, setzt der Controller mit diesem und dem im temporary Register zwischengespeicherten Wert für das High-Byte das 16-bit Register. Dabei ist zu beachten, dass intern nur ein temporary Register verfügbar ist, welches in Interruptroutinen mglw. mit einem anderen Wert überschrieben wird, wenn dort ebenfalls 16-bit Register beschrieben werden. `avr-gcc/avr-libc` berücksichtigen die korrekte Reihenfolge automatisch, wenn die Register mit ihrem "16-bit Label" (ohne H bzw. L) angesprochen werden, dabei ist der Schutz des temporary Registers vor Überschreiben durch Interruptroutinen dennoch zu beachten (im Zweifel beim Schreibzugriff die Interrupts kurzzeitig global deaktivieren).

Im Umgang mit 16-Bit Registern siehe auch:

- [Dokumentation der avr-libc](http://www.nongnu.org/avr-libc/user-manual/index.html) (<http://www.nongnu.org/avr-libc/user-manual/index.html>) Abschnitt Related Pages/Frequently Asked Questions/Nr. 8
- Datenblatt Abschnitt *Accessing 16-bit Registers*

Der UART

Allgemeines zum UART

Über den [UART](#) kann ein AVR leicht mit einer [RS232](#)-Schnittstelle eines PC oder sonstiger Geräte mit "serieller Schnittstelle" verbunden werden.

Mögliche Anwendungen des UART:

- Debug-Schnittstelle: z.B. zur Anzeige von Zwischenergebnissen ("printf-debugging" - hier besser "UART-debugging") auf einem PC. Auf dem Rechner reicht dazu eine Terminal-Software (MS-Windows: Hyperterm oder besser Bray-Terminal). Ein direkter Anschluss ist aufgrund unterschiedlicher Pegel nicht möglich, jedoch sind entsprechende Schnittstellen-ICs wie z.B. ein MAX232 günstig und leicht zu integrieren. Rechner ohne serielle Schnittstelle können über fertige USB-seriell-Adapter angeschlossen werden.
- "Mensch-Maschine Schnittstelle": z.B. Konfiguration und Statusabfrage über eine "Kommandozeile" oder Menüs
- Übertragen von gespeicherten Werten: z.B. bei einem Datenlogger
- Anschluss von Geräten mit serieller Schnittstelle (z.B. (Funk-)Modems, Mobiltelefone, Drucker, Sensoren, "intelligente" LC-Displays, GPS-Empfänger).
- "Feldbusse" auf RS485/RS422-Basis mittels entsprechenden Bustreiberbausteinen (z.B. MAX485)
- DMX, Midi etc.
- LIN-Bus (Local Interconnect Network): Preiswerte Sensoren/Aktoren in der Automobiltechnik und darüber hinaus

Einige AVR-Controller haben ein bis zwei vollduplexfähigen UART (Universal Asynchronous Receiver and Transmitter) schon eingebaut ("Hardware-UART"). Übrigens: Voll duplex heisst nichts anderes, als dass der Baustein gleichzeitig senden und empfangen kann.

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

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

UART Control Register.

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

Bit	7	6	5	4	3	2	1	0
Name	RXCIE	TXCIE	UDRIE	RXEN	TXEN	CHR9	RXB8	TXB8
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R	W
Initialwert	0	0	0	0	0	0	1	0

RXCIE (RX Complete Interrupt Enable)

Wenn dieses Bit gesetzt ist, wird ein UART RX Complete Interrupt ausgelöst, wenn ein Zeichen vom UART empfangen wurde. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein.

TXCIE (TX Complete Interrupt Enable)

Wenn dieses Bit gesetzt ist, wird ein UART TX Complete Interrupt ausgelöst, wenn ein Zeichen vom UART gesendet wurde. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein.

UDRIE (UART Data Register Empty Interrupt Enable)

Wenn dieses Bit gesetzt ist, wird ein UART Datenregister Leer Interrupt ausgelöst, wenn der UART wieder bereit ist um ein neues zu sendendes Zeichen zu übernehmen. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein.

UCR

RXEN (Receiver Enable)

Nur wenn dieses Bit gesetzt ist, arbeitet der Empfänger des UART überhaupt. Wenn das Bit nicht gesetzt ist, kann der entsprechende Pin des AVR als normaler I/O-Pin verwendet werden.

TXEN (Transmitter Enable)

Nur wenn dieses Bit gesetzt ist, arbeitet der Sender des UART überhaupt. Wenn das Bit nicht gesetzt ist, kann der entsprechende Pin des AVR als normaler I/O-Pin verwendet werden.

CHR9 (9 Bit Characters)

Wenn dieses Bit gesetzt ist, können 9 Bit lange Zeichen übertragen und empfangen werden. Das 9. Bit kann bei Bedarf als zusätzliches Stopbit oder als Paritätsbit verwendet werden. Man spricht dann von einem 11-Bit Zeichenrahmen:

1 Startbit + 8 Datenbits + 1 Stopbit + 1 Paritätsbit = 11 Bits

RXB8 (Receive Data Bit 8)

Wenn das vorher erwähnte CHR9-Bit gesetzt ist, dann enthält dieses Bit das 9. Datenbit eines empfangenen Zeichens.

TXB8 (Transmit Data Bit 8)

Wenn das vorher erwähnte CHR9-Bit gesetzt ist, dann muss in dieses Bit das 9. Bit des zu sendenden Zeichens eingeschrieben werden bevor das eigentliche Datenbyte in das Datenregister geschrieben wird.

UART Status Register.

Hier teilt uns der UART mit, was er gerade so macht.

Bit	7	6	5	4	3	2	1	0
Name	RXC	TXC	UDRE	FE	OR	-	-	-
R/W	R	R/W	R	R	R	R	R	R
Initialwert	0	0	1	0	0	0	0	0

RXC (UART Receive Complete)

Dieses Bit wird vom AVR gesetzt, wenn ein empfangenes Zeichen vom Empfangs-Schieberegister in das Empfangs-Datenregister transferiert wurde.

Das Zeichen muss nun schnellstmöglich aus dem Datenregister ausgelesen werden. Falls dies nicht erfolgt bevor ein weiteres Zeichen komplett empfangen wurde wird eine Überlauf-Fehlersituation eintreffen. Mit dem Auslesen des Datenregisters wird das Bit automatisch gelöscht.

TXC (UART Transmit Complete)

Dieses Bit wird vom AVR gesetzt, wenn das im Sende-Schieberegister befindliche Zeichen vollständig ausgegeben wurde und kein weiteres Zeichen im Sendedatenregister ansteht. Dies bedeutet also, wenn die Kommunikation vollumfänglich abgeschlossen ist.

Dieses Bit ist wichtig bei Halbduplex-Verbindungen, wenn das Programm nach dem Senden von Daten auf Empfang schalten muss. Im Vollduplexbetrieb brauchen wir dieses Bit nicht zu beachten.

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

UDRE (UART Data Register Empty)

Dieses Bit 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 (Framing Error)

Dieses Bit wird vom AVR gesetzt, wenn der UART einen Zeichenrahmenfehler detektiert, d.h. wenn das Stopbit eines empfangenen Zeichens 0 ist.

Das Bit wird automatisch gelöscht, wenn das Stopbit des empfangenen Zeichens 1 ist.

OR (OverRun)

Dieses Bit wird vom AVR gesetzt, wenn unser Programm das im Empfangsdatenregister bereit liegende Zeichen nicht abholt bevor das nachfolgende Zeichen komplett empfangen wurde.

Das nachfolgende Zeichen wird verworfen.

Das Bit wird automatisch gelöscht, wenn das empfangene Zeichen in das Empfangsdatenregister transferiert werden konnte.

USR

UART Data Register.

Hier werden Daten zwischen UART und CPU übertragen. Da der UART im Vollduplexbetrieb gleichzeitig empfangen und senden kann, handelt es sich hier physikalisch um 2 Register, die aber über die gleiche I/O-Adresse angesprochen werden. Je nachdem, ob ein Lese- oder ein Schreibzugriff auf den UART erfolgt wird automatisch das richtige UDR angesprochen.

UDR

UART Baud Rate Register.

In diesem Register müssen wir dem UART mitteilen, wie schnell wir gerne kommunizieren möchten. Der Wert, der in dieses Register geschrieben werden muss, errechnet sich nach folgender Formel:

UBRR

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

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 empfiehlt 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 **UART Control Register**.

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

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

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

```
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 */

#ifndef F_CPU
/* In neueren Version der WinAVR/Mfile Makefile-Vorlage kann
   F_CPU im Makefile definiert werden, eine nochmalige Definition
   hier wuerde zu einer Compilerwarnung fuehren. Daher "Schutz" durch
   #ifndef/#endif */
/* Zum Beispiel 4Mhz-Quarz (falls nicht schon im Makefile definiert): */
#define F_CPU 4000000;
#endif
#define UART_BAUD_RATE 9600 // 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 */
#ifndef F_CPU
#define F_CPU 3686400 /* Oszillator-Frequenz in Hz */
#endif
#define UART_BAUD_RATE 9600
#define UART_BAUD_CALC(UART_BAUD_RATE,F_OSC) ((F_CPU)/((UART_BAUD_RATE)*16L)-1)

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

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

Teilweise kann man im Datenblatt der jeweiligen CPU's auch den entsprechenden Wert nachschlagen. Ein Beispiel für einen ATmega32 bei 16MHz und für 19200 Baud: im Datenblatt des ATmega32 Tabelle "Examples of UBRR Settings" liest man für diese Vorgaben den Wert 51 ab. Dieser wird nun auf die Register UBRRH und UBRRL verteilt.

Die Initialisierung für den USART könnte dann so aussehen:

```
/* USART-Init 19200 Baud bei 16MHz für Mega32 */
UCSRB |= ( 1 << TXEN ); /* UART TX einschalten
UCSRC |= ( 1 << URSEL ) | ( 3 << UCSZ0 ); /* Asynchron 8N1
UBRRH = 0; /* Highbyte ist 0
UBRRL = 51; /* Lowbyte ist 51 ( dezimal )
```

Der obige Code ist gegenüber den Versionen mit '#define ...' nicht so portabel, dafür aber etwas übersichtlicher.

Senden einzelner Zeichen

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

```
// bei AVR mit einem UART ("classic AVR" z.B. AT90S8515)
while (!(USR & (1<<UDRE))); /* warten bis Senden moeglich */
UDR = 'x'; /* schreibt das Zeichen x auf die Schnittstelle */

/** ODER **/

// bei neueren AVR's steht der Status in UCSRA/UCSR0A/UCSR1A, hier z.B. fuer ATmega16:
while (!(UCSRA & (1<<UDRE))); /* warten bis Senden moeglich */
UDR = 'x'; /* schreibt das Zeichen x auf die Schnittstelle */
```

Schreiben einer Zeichenkette (String)

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

```
// putc fuer AVR mit einem UART (z.B. AT90S8515)
int uart_putc(unsigned char c)
{
    while(!(USR & (1 << UDRE))); /* warte, bis UDR bereit */
    UDR = c; /* sende Zeichen */
    return 0;
}

/** ODER **/
```

```
// bei neueren AVR's andere Bezeichnung fuer die Statusregister, hier ATmega16:
int uart_putc(unsigned char c)
{
    while (!(UCSRA & (1<<UDRE))); /* warten bis Senden moeglich */
    UDR = c;                       /* sende Zeichen */
    return 0;
}

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

/* verwenden der printf-lib: */
int
main(void)
{
    /* initialisieren der UART */
    uart_init();

    /* stream öffnen */
    fdevopen(uart_putc, NULL, 0);

    /* printf... */
    printf("%s: stdout erfolgreich auf UART umgeleitet\n", __FUNCTION__);

    /* exit... */
    return 0;
}
```

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. empfangenen Zeichen/Bytes zwischengespeichert und mittels Interruptroutinen an den U(S)ART weitergegeben 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.

Zeichen Empfangen

Zum Empfang von Zeichen muss der Empfangsteil des UART bei der Initialisierung aktiviert werden, indem das RXEN-Bit im jeweiligen Konfigurationsregister (UCSRB bzw UCSR0B/UCSR1B) gesetzt wird. Im einfachsten Fall wird solange gewartet, bis ein Zeichen empfangen wurde, dieses steht dann im UART-Datenregister (UDR bzw. UDR0 und UDR1 bei AVR's mit 2 UARTS) zur Verfügung (sogen. "Polling-Betrieb"). Ein Beispiel für den ATmega16:

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

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

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

Diese Funktion blockiert den Programmablauf. Alternativ kann das RXC-Bit in einer Programmschleife abgefragt

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

TODO: 9bit

Software-UART

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

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

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

- siehe auch: Weiterführende Informationen inkl. Beispielen für die Nutzung von stdio-Funktionen (printf etc.) im [AVR-Tutorial - UART](#).
- [Peter Fleury's](http://homepage.sunrise.ch/mysunrise/peterfleury/) (<http://homepage.sunrise.ch/mysunrise/peterfleury/>) UART-Bibliothek fuer avr-gcc/avr-libc

Analoge Ein- und Ausgabe

Analoge Eingangswerte werden in der Regel über den AVR Analog-Digital-Converter (AD-Wandler, ADC) eingelesen, der in vielen Typen verfügbar ist (typisch 10bit Auflösung). Durch diesen werden analogen Signale (Spannungen) in digitale Zahlenwerte gewandelt. Bei AVR's, die über keinen internen AD-Wandler verfügen (z.B. ATmega162, ATtiny2313), kann durch externe Beschaltung (R/C-Netzwerk und "Zeitmessung") die Funktion des AD-Wandlers "emuliert" werden.

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

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

ADC (Analog Digital Converter)

Der Analog-Digital-Konverter (ADC) wandelt analoge Signale in digitale Werte um, welche vom Controller interpretiert werden können. Einige AVR-Typen haben bereits einen mehrkanaligen Analog-Digital-Konverter eingebaut. Die Genauigkeit, mit welcher ein analoges Signal aufgelöst werden kann, wird durch die Auflösung des ADC in Anzahl Bits angegeben, man hört bzw. liest jeweils von 8-Bit-ADC oder 10-Bit-ADC oder noch höher.

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 Analog-Digital-Konverters ist, also je mehr Bits er hat, um so genauer kann der Wert erfasst werden.

Der interne ADC im AVR

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

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

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

- Eine externe Referenzspannung von maximal **Vcc** am Anschlusspin **AREF**. Die minimale (externe) Referenzspannung darf jedoch nicht beliebig niedrig sein, vgl. dazu das (aktuellste) Datenblatt des verwendeten Controllers.
- Verfügt der AVR über eine interne Referenzspannung, kann diese genutzt werden. Alle aktuellen AVRs mit internem AD-Wandler sollten damit ausgestattet sein (vgl. Datenblatt: 2,56V oder 1,1V je nach Typ). Das Datenblatt gibt auch über die Genauigkeit dieser Spannung Auskunft.
- Die **interne** Referenzspannung wird auf **Vcc** bezogen, eine externe Referenzspannung auf **GND (Masse)**. Davon unabhängig werden digitalisierte Spannungen **immer** auf GND bezogen. Beim ATMEGA8 z.B. ist der Pin AREF über 32kOhm mit GND verbunden, d.h. man muss diese doch extrem niedrige Eingangsimpedanz mit in die Berechnung für einen Spannungsteiler einbeziehen, bzw. kann diesen Widerstand als R2 gleich mit benutzen. Formel für Spannungsteiler: $U_{div} = U / ((R1 + R2) / R2)$

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

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

Einfache Wandlung (Single Conversion)

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

Frei laufend (Free Running)

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

Die Register des ADC

Der **ADC** verfügt über eigene Register. Im Folgenden die Registerbeschreibung eines (inzwischen veralteten) AT90S8535, welcher über 8 ADC-Kanäle verfügt. Die Register unterscheiden sich jedoch nicht erheblich von denen aktueller AVR's (vgl. Datenblatt).

ADC Control and Status Register.

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

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

ADEN (ADC Enable)

Dieses Bit muss gesetzt werden, um den **ADC** überhaupt zu aktivieren. Wenn das Bit nicht gesetzt, ist können die Pins wie normale I/O-Pins verwendet werden.

ADSC (ADC Start Conversion)

Mit diesem Bit wird ein Messvorgang gestartet. In der frei laufenden Betriebsart muss das Bit gesetzt werden, um die kontinuierliche Messung zu aktivieren.

Wenn das Bit nach dem Setzen des **ADEN**-Bits zum ersten Mal gesetzt wird, führt der Controller zuerst eine zusätzliche Wandlung und erst dann die eigentliche Wandlung aus. Diese zusätzliche Wandlung wird zu Initialisierungszwecken durchgeführt.

Das Bit bleibt nun so lange auf 1, bis die Umwandlung abgeschlossen ist, im Initialisierungsfall entsprechend bis die zweite Umwandlung erfolgt ist und geht danach auf 0.

ADFR (ADC Free Running Select)

Mit diesem Bit wird die Betriebsart eingestellt.

Eine logische 1 aktiviert den frei laufenden Modus. Der **ADC** misst nun ständig den ausgewählten Kanal und schreibt den gemessenen Wert in das **ADC Data Register**.

ADIF (ADC Interrupt Flag)

Dieses Bit wird vom **ADC** gesetzt, wenn eine Umwandlung erfolgt und das **ADC Data Register** aktualisiert ist.

Wenn das **ADIE** Bit sowie das **I-Bit** im AVR **Statusregister** gesetzt ist, wird der **ADC Interrupt** ausgelöst und die Interrupt-Behandlungsroutine aufgerufen.

Das Bit wird automatisch gelöscht, wenn die Interrupt-Behandlungsroutine aufgerufen wird. Es kann jedoch auch gelöscht werden, indem eine logische 1 in das Register geschrieben wird (So steht es in der AVR-Dokumentation).

ADCSR

ADIE (ADC Interrupt Enable)

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

ADPS2...ADPS0 (ADC Prescaler Select Bits)

Diese Bits bestimmen den Teilungsfaktor zwischen der Taktfrequenz und dem Eingangstakt des **ADC**.

Der **ADC** benötigt einen eigenen Takt, welchen er sich selber aus der CPU-Taktfrequenz erzeugt. Der **ADC**-Takt sollte zwischen 50 und 200kHz sein.

Der Vorteiler muss also so eingestellt werden, dass die CPU-Taktfrequenz dividiert durch den Teilungsfaktor einen Wert zwischen 50-200kHz ergibt.

Bei einer CPU-Taktfrequenz von 4MHz beispielsweise rechnen wir

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

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

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

ADC Data Register

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

ADCL**ADCH**

```
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)
```

ADC Multiplexer Select Register

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

Das Register ist wie folgt aufgebaut:

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

ADMUX

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 AVR's 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 mux (1<<MUX2)

uint16_t ReadChannel(uint8_t mux)
{
    uint8_t i;
    uint16_t result = 0;           //Initialisieren wichtig, da lokale Variablen
                                   //nicht automatisch initialisiert werden und
                                   //zufällige Werte haben. Sonst kann Quatsch rauskommen
    ADCSRA = (1<<ADEN) | (1<<ADPS1) | (1<<ADPS0); // Frequenzvorteiler
                                   // setzen auf 8 (1) und ADC aktivieren (1)

    ADMUX = mux;                   // Kanal waehlen (MUX2 ist ausgewählt)
    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)
    ADCSRA = (1<<ADIF);           //ADIF löschen (siehe DB, macht sonst die ISR),
                                   //da sonst Schleife im nächsten Durchgang nicht
                                   //wartet bis Wandlung fertig. Nicht ADCSRA |= (1<<ADIF);
                                   //siehe FAQ Nr. 24 der avr-libc
                                   // ** ACHTUNG **
                                   //das funktioniert nur, wenn in dem
                                   //Register nur(!) Interrupt-Flags stehen
                                   //ansonsten ADCSRA |= 1(<<ADIF) verwenden

    /* 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 += ADCW;                 // Wandlungsergebnisse aufaddieren
    ADCSRA = (1<<ADIF);            //ADIF löschen
}
ADCSRA &= ~(1<<ADEN);             // ADC deaktivieren (2)

result /= 4;                       // 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) gekennzeichneten Zeilen in eine Funktion `adc_init()` o.ä. und löscht die mit (2) markierten Zeilen. Das Löschen des ADIF-Flags sollte, **entgegen** der [FAQ](http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq_intbits) (http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq_intbits), mit

```

...
ADCSRA |= (1<<ADIF);
...

```

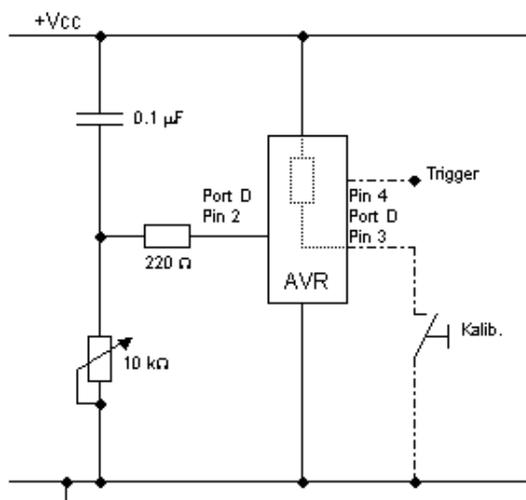
erfolgen. Die Methode in der FAQ eignet sich nur für Register in denen **nur** Interrupt-Flags stehen.

Analog-Digital-Wandlung ohne internen ADC

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.

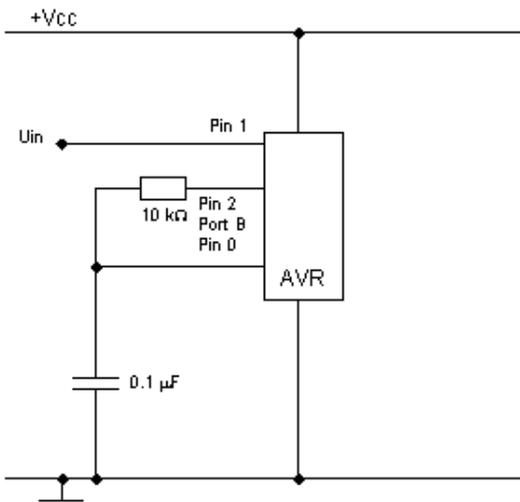
Nachdem das Programm auf den AVR geladen wurde, muss dieser kalibriert werden. Dazu wird der Kalibrierungsschalter geschlossen und das Poti einige Male zwischen minimaler und maximaler Stellung hin und her gedreht. Dabei werden die jeweiligen Maximalwerte bestimmt. Wenn der Kalibrierschalter wieder geöffnet 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.

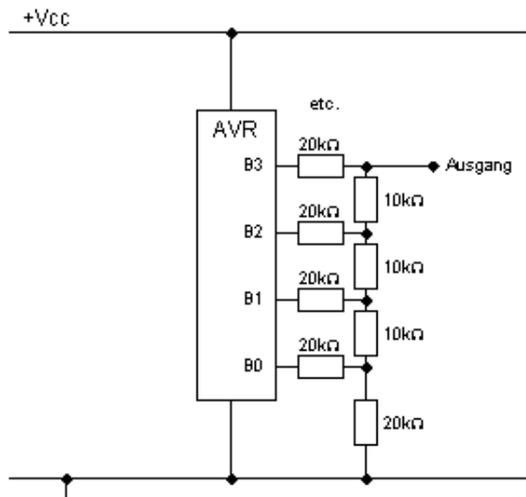
Der Vorteil dieser Schaltung liegt allerdings darin, dass damit direkt Spannungen gemessen werden können.

DAC (Digital Analog Converter)

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

DAC über mehrere digitale Ausgänge

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

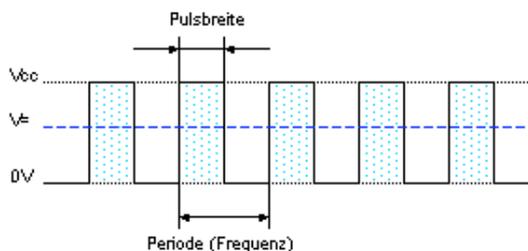


Es sollten selbstverständlich möglichst genaue Widerstände verwendet werden, also nicht unbedingt solche mit einer Toleranz von 10% oder mehr. Weiterhin empfiehlt es sich, je nach Anwendung den Ausgangsstrom über einen Operationsverstärker zu verstärken.

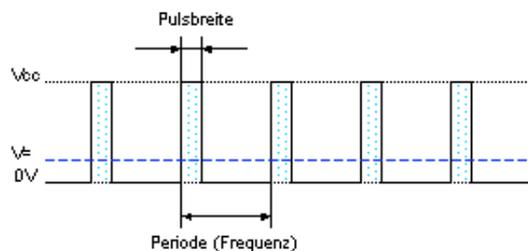
PWM (Pulsweitenmodulation)

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

Wie wir alle wissen ist ein Mikrocontroller ein rein digitales Bauteil. Definieren wir einen Pin als Ausgang, dann können wir diesen Ausgang entweder auf HIGH setzen, worauf am Ausgang die Versorgungsspannung V_{cc} anliegt, oder aber wir setzen den Ausgang auf LOW, wonach dann $0V$ am Ausgang liegt. Was passiert aber nun, wenn wir periodisch mit einer festen Frequenz zwischen HIGH und LOW umschalten? - Richtig, wir erhalten eine Rechteckspannung, wie die folgende Abbildung zeigt:



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



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

Mit den 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 bei anderen AVR-Controllern vergleichbar, die Pinbelegung allerdings nicht unbedingt, weshalb ein Blick ins entsprechende Datenblatt dringend angeraten wird.

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

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

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

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

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

COM1A1	COM1A0	Bedeutung
0	0	Keine Wirkung, Pin wird nicht geschaltet.
0	1	Keine Wirkung, Pin wird nicht geschaltet.
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 (Vorteiler) einstellen, und somit auch die Frequenz des **PWM-Signals** bestimmen.

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

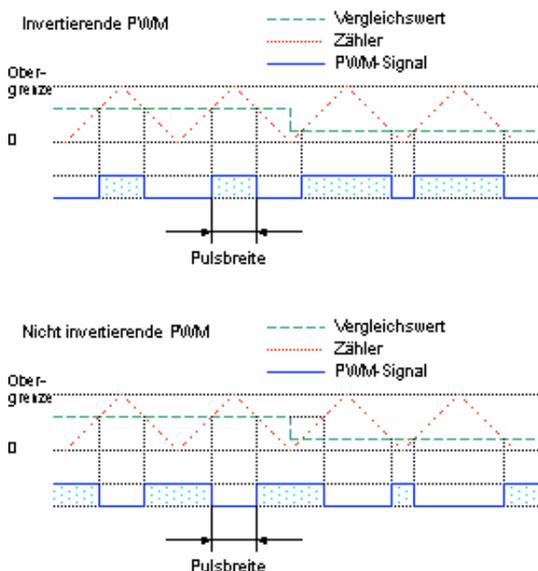
Also um einen Takt von CK / 1024 zu generieren, verwenden wir folgenden Befehl:

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

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

```
OCR1A = xxx;
```

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



Ach ja, fast hätte ich's vergessen. Das generierte **PWM**-Signal wird am Output Compare Pin **OC1** des Timers ausgegeben und leider können wir deshalb auch beim AT90S2313 nur ein einzelnes **PWM**-Signal mit dieser Methode generieren. Andere AVR-Typen verfügen über bis zu vier PWM-Ausgänge.

Die Timer/Counter des AVR

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

Wir brauchen also eine Möglichkeit, Vorgänge in Zeitabständen durchzuführen, die geringer als die Taktfrequenz des

Controllern sind. Selbstverständlich sollte die resultierende Frequenz auch noch möglichst 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 auf den AT90S2313. Für andere Modelltypen müsst ihr euch die allenfalls notwendigen Anpassungen aus den Datenblättern der entsprechenden Controller herauslesen.

Wir unterscheiden grundsätzlich zwischen 8-Bit Timern, welche eine Auflösung von 256 aufweisen und 16-Bit Timern mit (logischerweise) einer Auflösung von 65536. Als Eingangstakt für die Timer/Counter kann entweder die CPU-Taktfrequenz, der Vorteiler-Ausgang oder ein an einen I/O-Pin angelegtes Signal verwendet werden. Wenn ein externes Signal verwendet wird, so darf dessen Frequenz nicht höher sein als die Hälfte des CPU-Taktes.

Der Vorteiler (Prescaler)

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

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

8-Bit Timer/Counter

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

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

Timer/Counter Control Register

Timer 0

In diesem Register stellen wir ein, wie wir den Timer/Counter verwenden möchten.

Das Register ist wie folgt aufgebaut:

Bit	7	6	5	4	3	2	1	0
Name	-	-	-	-	-	CS02	CS01	CS00
R/W	R	R	R	R	R	R/W	R/W	R/W
Initialwert	0	0	0	0	0	0	0	0

CS02, CS01, CS00 (Clock Select Bits)

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

TCCR0

0	1	0	CPU-Takt / 8
0	1	1	CPU-Takt / 64
1	0	0	CPU-Takt / 256
1	0	1	CPU-Takt / 1024
1	1	0	Externer Pin TO , fallende Flanke
1	1	1	Externer Pin TO , steigende Flanke

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

Timer/Counter Daten Register Timer 0

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

TCNT0

Bit	7	6	5	4	3	2	1	0
Name	MSB							LSB
R/W	R/W	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 und die daran gebundene Interrupt-Routine abgearbeitet.

16-Bit Timer/Counter

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

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

Folgende Register sind dem Timer/Counter 1 zugeordnet:

Timer/Counter Control Register A Timer 1

In diesem und dem folgenden Register stellen wir ein, wie wir den Timer/Counter verwenden möchten. Das Register ist wie folgt aufgebaut:

Bit	7	6	5	4	3	2	1	0
------------	---	---	---	---	---	---	---	---

Name	COM1A1	COM1A0	-	-	-	-	PWM11	PWM10
R/W	R/W	R/W	R	R	R	R	R/W	R/W
Initialwert	0	0	0	0	0	0	0	0

COM1A1, COM1A0 (Compare Match Control Bits)

Diese 2 Bits bestimmen die Aktion, welche am Output-Pin **OC1** ausgeführt werden soll, wenn der Wert des Datenregisters des Timer/Counter 1 den Wert des Vergleichsregisters erreicht, also ein so genannter Compare Match auftritt.

Der Pin **OC1** (**PB3** beim 2313) muss mit dem Datenrichtungsregister als Ausgang konfiguriert werden.

COM1A1	COM1A0	Resultat
0	0	Output-Pin OC1 wird nicht angesteuert.
0	1	Das Signal am Pin OC1 wird invertiert (Toggle).
1	0	Der Output Pin OC1 wird auf 0 gesetzt.
1	1	Der Output Pin OC1 wird auf 1 gesetzt.

In der PWM-Betriebsart haben diese Bits eine andere Funktion.

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 <i>nicht invertierende PWM</i> .
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 <i>invertierende PWM</i> .

PWM11, PWM10 (PWM Mode Select Bits)

Mit diesen 2 Bits wird die PWM-Betriebsart des Timer/Counter 1 gesteuert.

PWM11	PWM10	Resultat
0	0	Die PWM-Betriebsart ist nicht aktiviert. Timer/Counter 1 arbeitet als normaler Timer bzw. Zähler.
0	1	8-Bit PWM Betriebsart aktivieren.
1	0	9-Bit PWM Betriebsart aktivieren.
1	1	10-Bit PWM Betriebsart aktivieren.

TCCR1A

Timer/Counter Control Register B Timer 1

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

ICNC1 (Input Capture Noise Canceler (4 CKs) Timer/Counter 1

oder auf Deutsch Rauschunterdrückung des Eingangssignals.

Wenn dieses Bit gesetzt ist und mit dem Input Capture Signal gearbeitet wird so werden nach der Triggerung des Signals mit der entsprechenden Flanke (steigend oder fallend) am Input Capture Pin **ICP** jeweils 4 Messungen mit der CPU-Frequenz des Eingangssignals abgefragt. Nur dann, wenn alle 4 Messungen den gleichen Zustand aufweisen gilt das Signal als erkannt.

ICES1 (Input Capture Edge Select Timer/Counter 1)

Mit diesem Bit wird bestimmt, ob die steigende (**ICES1=1**) oder fallende (**ICES1=0**) Flanke zur Auswertung des Input Capture Signals an Pin **ICP** heran gezogen wird.

CTC1 (Clear Timer/Counter on Compare Match Timer/Counter 1)

Wenn dieses Bit gesetzt ist, so wird nach einer Übereinstimmung des Datenregisters **TCNT1H/TCNT1L** mit dem Vergleichswert in **OCR1H/OCR1L** das Datenregister **TCNT1H/TCNT1L** auf 0 gesetzt.

TCCR1B

Da die Übereinstimmung im Takt nach dem Vergleich behandelt wird, ergibt sich je nach eingestelltem Vorteiler 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 (Clock Select Bits)

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

CS12	CS11	CS10	Resultat
0	0	0	Stopp, Der Timer/Counter wird angehalten.
0	0	1	CPU-Takt
0	1	0	CPU-Takt / 8
0	1	1	CPU-Takt / 64
1	0	0	CPU-Takt / 256
1	0	1	CPU-Takt / 1024
1	1	0	Externer Pin T0, 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.

Timer/Counter Daten Register Timer/Counter 1

Dieses ist als 16-Bit Aufwärtszähler mit Schreib- und Lesezugriff realisiert. Wenn der Zähler den Wert 65535 erreicht hat, beginnt er beim nächsten Zyklus wieder bei 0.

Bit	7	6	5	4	3	2	1	0	
Name	MSB								TCNT1H
Name								LSB	TCNT1L
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initialwert	0	0	0	0	0	0	0	0	

TCNT1H
TCNT1L

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

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

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

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

Timer/Counter Output Compare Register Timer/Counter 1

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

OCR1H
OCR1L

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

Zum Auslesen des Registers wird von der CPU ein internes TEMP-Register verwendet. Das gleiche Register wird auch verwendet, wenn auf **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 **OCR1L** und erst danach **OCR1H** ausgelesen werden.

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

Timer/Counter Input Capture Register Timer/Counter 1

Bit	7	6	5	4	3	2	1	0	
------------	---	---	---	---	---	---	---	---	--

Name	MSB									ICR1'H'
Name									LSB	ICR1'L'
R/W	R	R	R	R	R	R	R	R		
Initialwert	0	0	0	0	0	0	0	0		

ICR1H
ICR1L

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**

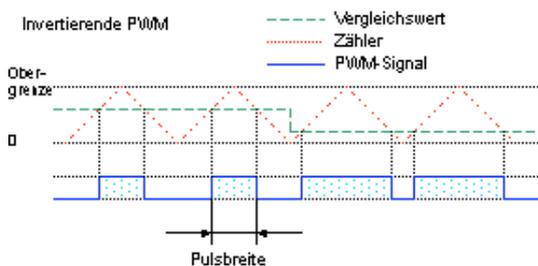
Die PWM-Betriebsart

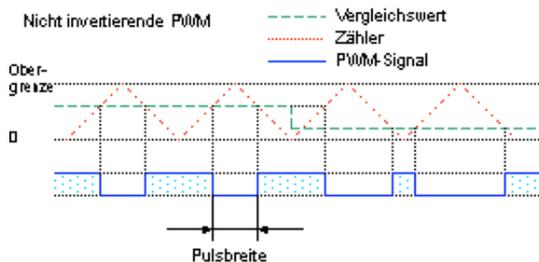
Wenn der Timer/Counter 1 in der PWM-Betriebsart betrieben wird, so bilden das Datenregister **TCNT1H/TCNT1L** und das Vergleichsregister **OCR1H/OCR1L** einen 8-, 9- oder 10-Bit, frei laufenden PWM-Modulator, welcher als PWM-Signal am **OC1**-Pin (**PB3** beim 2313) abgegriffen werden kann. Das Datenregister **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 Ausgangspin **OC1** gesetzt bzw. gelöscht, je nach Einstellung von **COM1A1** und **COM1A0** im **TCCR1A**-Register.

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





Vergleichswert-Überprüfung

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.

Timer/Counter Interrupt Mask

Register

Bit	7	6	5	4	3	2	1	0
Name	TOIE1	OCIE1A	-	-	TICIE	-	TOIE0	-
R/W	R/W	R/W	R	R	R/W	R	R/W	R
Initialwert	0	0	0	0	0	0	0	0

TOIE1 (Timer/Counter Overflow Interrupt Enable Timer/Counter 1)

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

TIMSK

OCIE1A (Output Compare Match Interrupt Enable Timer/Counter 1)

Beim Timer/Counter 1 kann zusätzlich zum Überlauf ein Vergleichswert definiert werden.

Wenn dieses Bit gesetzt ist, wird beim Erreichen des Vergleichswertes ein Compare Match Interrupt ausgelöst. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein.

TICIE (Timer/Counter Input Capture Interrupt Enable)

Wenn dieses Bit gesetzt ist, wird ein Capture Event Interrupt ausgelöst, wenn ein entsprechendes Signalereignis am Pin PD6(ICP) auftritt. Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein, wenn auch ein entsprechender Interrupt ausgelöst werden soll.

TOIE0 (Timer/Counter Overflow Interrupt Enable Timer/Counter 0)

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

Timer/Counter Interrupt Flag Register

Bit	7	6	5	4	3	2	1	0
Name	TOV1	OCF1A	-	-	ICF1	-	TOV0	-
R/W	R/W	R/W	R	R	R/W	R	R/W	R
Initialwert	0	0	0	0	0	0	0	0

TOV1 (Timer/Counter Overflow Flag Timer/Counter 1)

Dieses Bit wird vom Controller gesetzt, wenn beim Timer 1 ein Überlauf des Datenregisters stattfindet.

In der PWM-Betriebsart wird das Bit gesetzt, wenn die Zählrichtung von auf- zu abwärts und umgekehrt geändert wird (Zählerwert = 0).

Das Flag wird automatisch gelöscht, wenn der zugehörige Interrupt-Vektor aufgerufen wird. Es kann jedoch auch gelöscht werden, indem eine logische 1 (!) in das entsprechende Bit geschrieben wird.

TIFR OCF1A (Output Compare Flag Timer/Counter 1)

Dieses Bit wird gesetzt, wenn der aktuelle Wert des Datenregisters von Timer/Counter 1 mit demjenigen im Vergleichsregister **OCR1** übereinstimmt.

Das Flag wird automatisch gelöscht, wenn der zugehörige Interrupt-Vektor aufgerufen wird. Es kann jedoch auch gelöscht werden, indem eine logische 1 (!) in das entsprechende Bit geschrieben wird.

ICF1 (Input Capture Flag Timer/Counter 1)

Dieses Bit wird gesetzt, wenn ein Capture-Ereignis aufgetreten ist, welches anzeigt, dass der Wert des Datenregisters des Timer/Counter 1 in das Input Capture Register ICR1 übertragen wurde.

Das Flag wird automatisch gelöscht, wenn der zugehörige Interrupt-Vektor aufgerufen wird. Es kann jedoch auch gelöscht werden, indem eine logische 1 (!) in das entsprechende Bit geschrieben wird.

TOV0 (Timer/Counter Overflow Flag Timer/Counter 0)

Dieses Bit wird vom Controller gesetzt, wenn beim Timer 0 ein Überlauf des Datenregisters stattfindet.

Das Flag wird automatisch gelöscht, wenn der zugehörige Interrupt-Vektor aufgerufen wird. Es kann jedoch auch gelöscht werden, indem eine logische 1 (!) in das entsprechende Bit geschrieben wird.

Sleep-Modes

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

- **set_sleep_mode(uint8_t mode)**

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

- **sleep_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
...
```

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

```
#include <avr/io.h>
...
// Sleep-Mode "Power-Save" beim ATmega169 "manuell" 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) (<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 entsprechende Warnung ausgeben). Das Programm würde sich also hier aufhängen und auf ewig in der Schleife drehen. Und hier genau kommt der Watchdog zum Zug.

Wie funktioniert nun der Watchdog

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

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

Watchdog Timer Control Register

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

Das Register ist wie folgt aufgebaut:

Bit	7	6	5	4	3	2	1	0
Name	-	-	-	WDTOE	WDE	WDP2	WDP1	WDP0
R/W	R	R	R	R/W	R/W	R/W	R/W	R/W
Initialwert	0	0	0	0	0	0	0	0

WDTOE (Watchdog Turn Off Enable)

Dieses Bit muss gesetzt sein, wenn das Bit **WDE** gelöscht wird, andernfalls wird der Watchdog nicht ausgeschaltet.

Wenn das Bit einmal gesetzt ist wird es von der Hardware nach 4 Taktzyklen automatisch wieder gelöscht.

WDTCR WDE (Watchdog Enable)

Wenn dieses Bit gesetzt wird, so wird der Watchdog aktiviert.

Das Bit kann nur gelöscht werden, solange das Bit **WDTOE** auf 1 steht.

WDP2, WDP1, WDP0 (Watchdog Timer Prescaler Bits)

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

WDP2	WDP1	WDP0	Anzahl Zyklen	Typ. Timeoutzeit bei Vcc = 3V	Typ. Timeoutzeit bei Vcc = 5V
0	0	0	16K	47ms	15ms
0	0	1	32K	94ms	30ms
0	1	0	64K	0.19s	60ms
0	1	1	128K	0.38s	0.12s
1	0	0	256K	0.75s	0.24s
1	0	1	512K	1.5s	0.49s
1	1	0	1024K	3s	0.97s
1	1	1	2048K	6s	1.9s

Um den Watchdog mit dem AVR-GCC Compiler zu verwenden, muss die Headerdatei *wdt.h* (`#include <avr/wdt.h>`) in die Quelldatei eingebunden werden. Danach können die folgenden Funktionen verwendet werden:

- **wdt_enable(uint8_t timeout)**

Aktiviert den Watchdog und stellt den Vorteiler auf den gewünschten Wert ein bzw. der in timeout übergebene Wert wird in das WDTCR-Register eingetragen. Einige Timeout-Werte sind als Konstanten vordefiniert
Mögliche Timeoutwerte:

Konstante	Wert	TimeOut
WDTO_15MS	0	15 ms
WDTO_30MS	1	30 ms
WDTO_60MS	2	60 ms
WDTO_120MS	3	120 ms
WDTO_250MS	4	250 ms
WDTO_500MS	5	500 ms
WDTO_1S	6	1 S
WDTO_2S	7	2s

- **wdt_disable()**

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

- **wdt_reset()**

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

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

Watchdog-Anwendungshinweise

Ob nun der Watchdog als Schutzfunktion überhaupt verwendet werden soll, hängt stark von der Anwendung, der genutzten Peripherie und dem Umfang und der Qualitätssicherung des Codes ab. Will man sicher gehen, dass ein

Programm sich nicht in einer Endlosschleife verfängt, ist der Watchdog das geeignete Mittel dies zu verhindern. Weiterhin kann bei geschickter Programmierung der Watchdog dazu genutzt werden, bestimmte Stromsparfunktionen zu implementieren. 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.

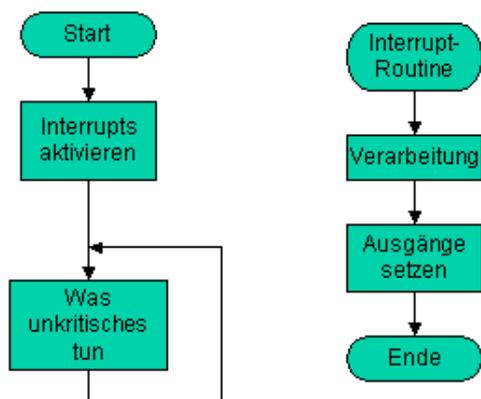
Es besteht die Möglichkeit herauszufinden, ob ein Reset durch den Watchdog ausgelöst wurde (beim ATmega16 z.B. Bit WDRF in MCUCSR). Diese Information sollte auch genutzt werden, falls ein WD-Reset in der Anwendung nicht planmässig implementiert wurde. Zum Beispiel kann man eine LED an einen freien Pin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

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

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

Interruptroutinen (ISRs) sollten also möglichst kurz sein und keine Schleifen mit vielen Durchläufen enthalten. Längere Operationen können meist in einen "Interrupt-Teil" in einer ISR und einen "Arbeitsteil" im Hauptprogramm aufgetrennt

werden. Z.B. Speichern des Zustands aller Eingänge im EEPROM in bestimmten Zeitabständen: ISR-Teil: Zeitvergleich (Timer,RTC) mit Logzeit/-intervall. Bei Übereinstimmung ein globales Flag setzen (volatile bei Flag-Deklaration nicht vergessen, s.u.). Dann im Hauptprogramm prüfen, ob das Flag gesetzt ist. Wenn ja: die Daten im EEPROM ablegen und Flag löschen.

Interrupt-Quellen

Die folgenden Ereignisse können einen Interrupt auf einem AVR AT90S2313 auslösen, wobei die Reihenfolge der Auflistung auch die Priorität der Interrupts aufzeigt.

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

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

Register

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

General Interrupt Mask Register.

Bit	7	6	5	4	3	2	1	0
Name	INT1	INT0	-	-	-	-	-	-
R/W	R/W	R/W	R	R	R	R	R	R
Initialwert	0	0	0	0	0	0	0	0

INT1 (External **I**nterrupt Request **1** Enable)

GIMSK

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 **I**nterrupt Request **0** Enable)

Wenn dieses Bit gesetzt ist, wird ein Interrupt ausgelöst, wenn am **INT0**-Pin eine steigende oder fallende (je nach Konfiguration im **MCUCR**) Flanke erkannt wird.

Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein.

Der Interrupt wird auch ausgelöst, wenn der Pin als Ausgang geschaltet ist. Auf diese Weise bietet sich die Möglichkeit, Software-Interrupts zu realisieren.

General Interrupt Flag Register.

Bit	7	6	5	4	3	2	1	0
Name	INTF1	INTF0	-	-	-	-	-	-
R/W	R/W	R/W	R	R	R	R	R	R
Initialwert	0	0	0	0	0	0	0	0

INTF1 (External Interrupt Flag 1)

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

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

INTF0 (External Interrupt Flag 0)

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

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

GIFR**MCU Control Register.**

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

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

SE (Sleep Enable)

Dieses Bit muss gesetzt sein, um den Controller mit dem **SLEEP**-Befehl in den Schlafzustand versetzen zu können.

Um den Schlafmodus nicht irrtümlich einzuschalten, wird empfohlen, das Bit erst unmittelbar vor Ausführung des **SLEEP**-Befehls zu setzen.

SM (Sleep Mode)

Dieses Bit bestimmt der Schlafmodus.

Ist das Bit gelöscht, so wird der **Idle**-Modus ausgeführt. Ist das Bit gesetzt, so wird der **Power-Down**-Modus ausgeführt. (für andere AVR Controller siehe Abschnitt "Sleep-Mode")

ISC11, ISC10 (Interrupt Sense Control 1 Bits)

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

MCUCR

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 (Interrupt Sense Control 0 Bits)

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

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

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.

Interrupts mit dem AVR GCC Compiler (WinAVR)

Funktionen zur Interrupt-Verarbeitung werden in den Includedateien *interrupt.h* und *signal.h* der avr-libc zur Verfügung gestellt.

```
// 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 Interrupts aktivieren, was zu unerwünschten Seiteneffekten führen kann. Die aus dem folgenden Beispiel ersichtliche Vorgehensweise ist in solchen Fällen vorzuziehen:

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

...

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

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

    /* hier "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

NichtSoGut();
// 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 Definitionen: **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 Interruptvektors angegeben werden. Diese sind in den jeweiligen Includedateien IOxxxx.h zu finden.

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 Funktionsrumpfe 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...
```

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

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

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

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

INTERRUPT

"Faustregel": im Zweifel **SIGNAL**. **INTERRUPT** nur dann, wenn man sich über die unterschiedliche Funktionsweise im Klaren ist.

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

Mit **INTERRUPT** wird fast genauso 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.

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

Datenaustausch mit Interrupt-Routinen

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

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

```
#include <avr/io.h>
#include <avr/interrupt.h>
#include <avr/signal.h>
#include <stdint.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 ISR, wird z.B. alle 10ms ausgefuehrt
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 geschriebene Wert bekannt sein:
    // volatile -> aktuellen Wert immer aus dem Speicher lesen
    if ( gKeyCounter > CNTDEBOUNCE ) { // Taste mind. 10*10 ms "prellfrei"
        if (gKeyCounter == CNTREPEAT) {
            /* hier: Code fuer "Taste lange gedrueckt" */
        }
        else {
            /* hier: Code fuer "Taste kurz gedrueckt" */
        }
    }
    ...
}
```

Bei Variablen größer ein Byte, auf die in Interrupt-Routinen und im Hauptprogramm zugegriffen wird, muss darauf geachtet werden, dass die Zugriffe auf die einzelnen Bytes ausserhalb der ISR nicht durch einen Interrupt unterbrochen werden. (Allgemeinplatz: 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 dem 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

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

```

Interrupt-Routinen und Registerzugriffe

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

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

```

#include <avr/io.h>

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

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

```

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

```

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

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

```

Das Setzen des einzelnen Bits wird bei eingeschalteter Optimierung für Register im unteren Speicherbereich in eine einzige Assembler-Anweisung (sbi) übersetzt und ist nicht anfällig für Unterbrechnungen durch Interrupts. Die

Anweisung zum Setzen von drei Bits wird jedoch in drei abhängige Assembler-Anweisungen übersetzt und bietet damit zwei "Angriffspunkte" für Unterbrechnungen. Eine Interrupt-Routine könnte nach dem Laden des Ausgangszustands in den Zwischenspeicher (hier Register 24) den Wert des Registers ändern, z.B. ein Bit löschen. Damit würde der Zwischenspeicher nicht mehr mit dem tatsächlichen Zustand übereinstimmen aber dennoch nach der Bitoperation (hier `ori`) in das Register zurückgeschrieben.

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

Lösungsmöglichkeiten:

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

Was macht das Hauptprogramm?

Im einfachsten (Ausnahme-)Fall gar nichts mehr. Es ist also durchaus denkbar, ein Programm zu schreiben, welches in der `main`-Funktion lediglich noch die Interrupts aktiviert und dann in einer Endlosschleife verharrt. Sämtliche Funktionen werden dann in den ISRs abgearbeitet. Diese Vorgehensweise ist jedoch bei den meisten Anwendungen schlecht: man verschenkt eine Verarbeitungsebene und hat ausserdem möglicherweise Probleme durch Interruptroutinen, die zu viel Verarbeitungszeit benötigen.

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

- siehe auch: [Dokumentation der avr-libc](http://www.nongnu.org/avr-libc/user-manual/index.html) (<http://www.nongnu.org/avr-libc/user-manual/index.html>) Abschnitt Modules/Interrupts and Signals

Speicherzugriffe

Atmel AVR-Controller verfügen typisch über drei Speicher:

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

Einige AVR's besitzen keinen RAM-Speicher, lediglich die Register können als "Arbeitsvariablen" genutzt werden. Da die Anwendung des avr-gcc auf solch "kleinen" Controllern ohnehin selten sinnvoll ist und auch nur bei einigen RAM-

losen Typen nach "[Bastelarbeiten](http://lightner.net/avr/ATtinyAvrGcc.html)" (<http://lightner.net/avr/ATtinyAvrGcc.html>) möglich ist, werden diese Controller hier nicht weiter berücksichtigt. Auch EEPROM-Speicher ist nicht auf allen Typen verfügbar. Generell sollten die nachfolgenden Erläuterungen auf alle ATmega-Controller und die größeren AT90-Typen übertragbar sein. Für die Typen ATtiny2313 und ATtiny26 der "tiny-Reihe" gelten die Ausführungen ebenfalls.

RAM

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

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 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 *pgmPointerToArray2[] PROGMEM = { pgmFooByteArray1, pgmFooByteArray2 };
...

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" (Stand avr-gcc 3.4.x). Damit ist der mögliche Speicherbereich für "Flash-Konstanten" auf 64kB begrenzt.

```
uint8_t *ptrToArray;

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

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

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

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

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

```

Floats und Structs lesen

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

```
/* Beispiel float aus Flash */

float pgmFloatArray[3] PROGMEM = {1.1, 2.2, 3.3};
...

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

```

```

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

    return u.f;
}
...

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

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

```

TODO: Beispiele fuer structs und pointer aus flash auf struct im flash (menues, state-machines etc.)

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 (!strcmp_P(StringImRam, StringImFlash, 5) {
    // mach' was, wenn die ersten 5 Zeichen identisch - hier nicht
}
else {
    // der Code hier wuerde ausgefuehrt
}

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

```

Aber Vorsicht: Ersetzt man zum Beispiel

```

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

```

durch

```

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

```

dann kann es zu Problemen mit AVR-GCC kommen. Zu erkennen daran, dass der Initialisierungsstring von "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 weiteres Problem des AVR-GCC (gesehen bei

avr-gcc 3.4.1 und 3.4.2) bei der Anpassung an die Harvard-Architektur zu sein (konstanter Pointer auf variable Daten? !). Abhilfe ("Workaround"): Initialisierung bei Zeichenketten mit [] oder gleich im Code PSTR(...) nutzen.

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

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

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

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

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

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

```

Flash in der Anwendung schreiben

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

Warum so kompliziert?

Zu dem Thema, warum die Verarbeitung von Werten aus dem Flash-Speicher so "kompliziert" ist, sei hier nur kurz erläutert: Die Harvard-Architektur des AVR weist getrennte Adressräume für Programm(Flash)- und Datenspeicher(RAM) auf. Der C-Standard und der gcc-Compiler sehen keine unterschiedlichen Adressräume vor. Hat man zum Beispiel eine Funktion string_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-manual/index.html) (<http://www.nongnu.org/avr-libc/user-manual/index.html>) Abschnitte Modules/Program Space String Utilities und Abschnitt Modules/Bootloader Support Utilities

EEPROM

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

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

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

Bei der Deklaration einer Variable im EEPROM, ist das Attribut für die Section ".eeprom" zu ergänzen. 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 EEMEM im Quellcode durch __attribute__ ... ersetzen

// EEMEM ist mglw. schon in in eeprom.h definiert
// (neuere Versionen der avr-libc)
// hier: definiere falls noch nicht bekannt ("alte" avr-libc)
#ifndef EEMEM
#define EEMEM __attribute__((section(".eeprom")))
#endif

...

/* Byte */
uint8_t eeFooByte EEMEM = 123;

/* Wort */
uint16_t eeFooWord EEMEM = 12345;

/* float */
float eeFooFloat EEMEM;

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

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

Bytes lesen/schreiben

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

```
...
uint8_t myByte;

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

```

...
myByte = 99;
eeprom_write_byte(&eeFooByte, myByte); // schreiben
...
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-Adresse und die Länge des Datenblocks in Bytes (*size_t*).

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

```

...
uint8_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], &eeFooByteArray1[0], 3);

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

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

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

```

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

```

...
float myFloat = 12.34;

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

```

```

u.r = myFloat;

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

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

```

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

```

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

#define MAXSENSORS 3
tMyStruct eeMyStruct[MAXSENSORS] 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 entsprechenden Optionen im 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 unterstützt (Stand Version 1.2.3). Insbesondere beim den AVR's "der neuen Generation" (ATmega48/88/168/169) funktionieren die Funktionen nicht korrekt (Ursache: unterschiedliche Speicheradressen der EEPROM-Register, detaillierte Auflistung der nicht-unterstützten Typen in der avr-libc-Dokumentation) Etwas ältere Typen, oder zu den "etablierten" Controllern kompatible, bereiten jedoch hier keine Probleme. 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 Beispiellecodes für den Schreib- und Lesezugriff enthalten. Der dort gezeigte Code kann direkt auch mit dem avr-gcc (ohne avr-libc/eeprom.h) genutzt werden ("copy/paste", gegebenenfalls Schutz vor Unterbrechnung/Interrupt ergänzen `uint8_t sreg; sreg=SREG; cli(); [EEPROM-Code]; SREG=sreg; return;`, siehe Abschnitt Interrupts). Alternativ kann aus dem Quellcode der eeprom-Funktionen der avr-libc eine "passende" EEprom-Library erstellt werden, in der die EEPROM-Registeradressen zum Controller

"passen".

Vorabinformation: Unterstützung für "neue" AVR's wurde in der avr-libc Version 1.2.5 integriert. Abschnitt wird überarbeitet, sobald WinAVR mit avr-libc 1.2.5+ verfügbar. (mt 18.8.05)

- siehe auch: [Dokumentation der avr-libc](http://www.nongnu.org/avr-libc/user-manual/index.html) (<http://www.nongnu.org/avr-libc/user-manual/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 direkt in den C-Code integriert. Eine Quellcode-Datei enthält somit C- und Assembleranweisungen

Assembler-Dateien

Der Assembler-Code befindet sich in eigenen Quellcode-Dateien. Diese werden vom gnu-Assembler (avr-as) zu Object-Dateien assembliert ("compiliert") und mit den aus dem C-Code erstellten Object-Dateien zusammengebunden (gelinkt).

Inline-Assembler

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

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

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

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

```
...
uint16_t i;

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

...

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

...
```

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

Ein weiterer nützlicher "Assembler-Einzeiler" ist der Aufruf von `sleep (asm volatile ("sleep"::));`, da hierzu 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, das 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 Zeichenkette" verbunden.
- Sprung-Marken (Labels) werden mit einem Prozentzeichen abgeschlossen, der Präprozessor (Assembler?) setzt an dieser Stelle eine laufende Nummer ein, die Doppelbezeichnungen 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` im Abschnitt [Related Pages/Inline Asm](http://www.nongnu.org/avr-libc/user-manual/inline_asm.html) (http://www.nongnu.org/avr-libc/user-manual/inline_asm.html).

Siehe auch: [AVR Assembler-Anweisungsliste](http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf) (http://www.atmel.com/dyn/resources/prod_documents/doc0856.pdf)

Assembler-Dateien

Assembler-Dateien erhalten die Endung `.S` (groses 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/assembly.html) (<http://www.nongnu.org/avr-libc/user-manual/assembly.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) (http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq_reg_usage)

Globale Variablen für Datenaustausch

Oftmals kommt man um globale Variablen nicht herum, z.B. um den Datenaustausch zwischen Hauptprogramm und Interrupt-Routinen zu realisieren. Hierzu muß man im Assembler wissen, wo genau die Variable vom C-Compiler abgespeichert wird.

Hierzu muß die Variable, hier "zaehler" genannt, zuerst im C-Code als Global definiert werden, z.B. so:

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

volatile uint8_t zaehler;

int16_t main (void)
{
    // irgendein Code, in dem zaehler benutzt werden kann
}
```

Im folgenden Assembler-Beispiel wird der Externe Interrupt0 verwendet, um den Zähler hochzuzählen. Es fehlen die Initialisierungen des Interrupts und die Interrupt-Freigabe, so richtig sinnvoll ist der Code auch nicht, aber er zeigt (hoffentlich) wie es geht. Im Umgang mit Interrupt-Vektoren gilt beim GCC-Assembler das Gleiche, wie bei C: Man muß die exakte Schreibweise beachten, ansonsten wird nicht der Interrupt-Vektor angelegt, sondern eine neue Funktion - und man wundert sich, daß nichts funktioniert (vgl. das AVR-GCC Handbuch).

```
#include "avr/io.h"

temp = 16

.extern zaehler

.global SIG_INTERRUPT0
SIG_INTERRUPT0:

    push temp                // wichtig: Benutzte Register und das
    in temp,_SFR_IO_ADDR(SREG) // Status-Register (SREG) sichern !
    push temp

    lds temp,zaehler        // Wert aus dem Speicher lesen
    inc temp                // bearbeiten
    sts zaehler,temp        // und wieder zurückschreiben

    pop temp                // die benutzten Register wiederherstellen
    out _SFR_IO_ADDR(SREG),temp
    pop temp
    reti

.end
```

Globale Variablen im Assemblerfile anlegen

Alternativ können Variablen aber auch im Assemblerfile angelegt werden. Dadurch kann auf eine .c-Datei verzichtet werden. Für das obige Beispiel könnte der Quelltext dann die Dateien zaehl_asm.S und zaehl_asm.h abgelegt werden, so dass nur noch zaehl_asm.S mit kompiliert werden müsste.

Anstatt im Assemblerfile über das Schlüsselwort *.extern* auf eine vorhandene Variable zu verweisen, wird dazu mit dem Schlüsselwort *.comm* die benötigte Anzahl von Bytes für eine Variable reserviert.

zaehl_asm.S

```
#include "avr/io.h"

temp = 16

// 1 Byte im RAM für den Zähler reservieren
.comm zaehler, 1

.global SIG_INTERRUPT0
SIG_INTERRUPT0:

...
```

In der Headerdatei wird dann auf die Variable nur noch verwiesen (Schlüsselwort *extern*):

zaehl_asm.h

```
#ifndef ZAEHL_ASM_H
#define ZAEHL_ASM_H

extern volatile uint8_t zaehler;

#endif
```

Im Gegensatz zu globalen Variablen in C werden so angelegte Variablen nicht automatisch mit dem Wert 0 initialisiert.

Variablen größer als 1 Byte

Variablen, die größer als ein Byte sind, können in Assembler auf ähnliche Art angesprochen werden. Hierzu müssen nur genug Bytes angefordert werden, um die Variable aufzunehmen. Soll z.B. für den Zähler eine Variable vom Typ *unsigned long*, also *uint32_t* verwendet werden, so müssen 4 Bytes reserviert werden:

```
...
// 4 Byte im RAM für den Zähler reservieren
.comm zaehler, 4
...
```

Die dazugehörige Deklaration im Headerfile wäre dann:

```
...
extern volatile uint32_t zaehler;
...
```

Bei Variablen, die größer als ein Byte sind, werden die Werte beginnend mit dem niederwertigsten Byte im RAM abgelegt. Das folgende Codeschnippel zeigt, wie unter Assembler auf die einzelnen Bytes zugegriffen werden kann. Dazu wird im Interrupt nun ein 32-Bit Zähler erhöht:

```
#include "avr/io.h"

temp = 16

// 4 Byte im RAM für den Zähler reservieren
.comm zaehler, 4

.global SIG_INTERRUPT0
SIG_INTERRUPT0:
```

```

push temp                // wichtig: Benutzte Register und das
in temp, _SFR_IO_ADDR(SREG) // Status-Register (SREG) sichern !
push temp

// 32-Bit-Zähler incrementieren
lds temp, (zaehler + 0) // 0. Byte (niederwertigstes Byte)
inc temp
sts (zaehler + 0), temp
brne RAUS

lds temp, (zaehler + 1) // 1. Byte
inc temp
sts (zaehler + 1), temp
brne RAUS

lds temp, (zaehler + 2) // 2. Byte
inc temp
sts (zaehler + 2), temp
brne RAUS

lds temp, (zaehler + 3) // 3. Byte (höchstwertigstes Byte)
inc temp
sts (zaehler + 3), temp
brne RAUS

RAUS:
pop temp                // die benutzten Register wiederherstellen
out _SFR_IO_ADDR(SREG), temp
pop temp
reti

.end

```

TODO: 16-Bit / 32-Bit Variablen, Zugriff auf Arrays (Strings)

Anhang

Anhang A - Externe Referenzspannung des internen Analog-Digital-Wandlers

Die minimale (externe) Referenzspannung des ADC darf nicht beliebig niedrig sein, vlg. dazu das (aktuellste) Datenblatt des verwendeten Controllers. z.B. beim ATMEGA8 darf sie lt. Datenblatt (S.245, Tabelle 103, Zeile "VREF") 2,0V nicht unterschreiten. HINWEIS: diese Information findet sich erst in der letzten Revision (Rev. 2486O-10/04) des Datenblatts.

Meiner eigenen Erfahrung nach kann man aber (auf eigene Gefahr und natürlich nicht für Seriengeräte) durchaus noch ein klein wenig weiter heruntergehen, bei dem von mir unter die Lupe genommenen ATMEGA8L (also die Low-Voltage-Variante) funktioniert der ADC bei 5V Betriebsspannung mit bis zu VREF=1,15V hinunter korrekt, ab 1,1V und darunter digitalisiert er jedoch nur noch Blödsinn). Ich würde sicherheitshalber nicht unter 1,5V gehen und bei niedrigeren Betriebsspannungen mag sich die Untergrenze für VREF am Pin AREF ggf. nach oben(!) verschieben. In der letzten Revision des Datenblatts ist außerdem korrigiert, dass ADC4 und ADC5 sehr wohl 10 Bit Genauigkeit bieten (und nicht bloß 8 Bit, wie in älteren Revisionen irrtümlich angegeben.)

TODO

stdio.h, malloc() ???, Code-Optimierungen ("tricks"), "naked"-Functionen ??, IO-Register als Parameter/"Variablen" (volatile uint8_t *mybusport; mybusport=&PORTB; void sendbus(uint8_t *parm)...) SPI ? JTAG ? I²C Bus

Von "<http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial>"

Seitenkategorien: [AVR](#)

Impressum: Andreas Schwarz - Seßlacher Weg 4 - 96450 Coburg - webmaster(at)mikrocontroller(dot)net