www.mikrocontroller.net

AVR-GCC-Tutorial

Inhaltsverzeichnis

Voraussetzungen

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

Vorwort

Dieses Tutorial soll den Einstieg in die Programmierung von Atmel AVR-Mikrocontrollern in der Programmiersprache C mit dem freien C-Compiler AVR-GCC aus der GNU Compiler Collection erleichtern.

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

Der Compiler und die Standardbibliothek avr-libc werden stetig weiterentwickelt. Einige Unterschiede, die sich im Verlauf der Entwicklung ergeben haben, werden im Haupttext und Anhang zwar erläutert, Anfängern sei jedoch empfohlen, die aktuellen Versionen zu nutzen (für MS-Windows: aktuelle Version des WinAVR-Pakets; für Linux: siehe Artikel AVR und Linux).

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

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

Siehe auch

Um diese riesige Seite etwas überschaubarer zu gestalten, wurden einige Kapitel ausgelagert, die nicht unmittelbar mit den Grundlagen von avr-gcc in Verbindung stehen. All diese Seiten gehören zur Kategorie:avr-gcc Tutorial.

- → Der UART

- → Der Watchdog
- → Die Timer und Zähler des AVR
- → Exkurs Makefiles
- → LCD-Ansteuerung

Benötigte Werkzeuge

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

- Platine oder Versuchsaufbau für die Aufnahme eines AVR Controllers, der vom avr-gcc Compiler unterstützt wird (alle ATmegas und die meisten AT90, siehe Dokumentation der avr-libc für unterstützte Typen). Dieses Testboard kann durchaus auch selbst gelötet oder auf einem Steckbrett aufgebaut werden. Einige Registerbeschreibungen dieses Tutorials beziehen sich auf den inzwischen veralteten AT90S2313. Der weitaus größte Teil des Textes ist aber für alle Controller der AVR-Familie gültig. Brauchbare Testplattformen sind auch das STK500 und der AVR Butterfly von Atmel. Weitere Infos findet man in den Artikeln AVR Starterkits und AVR-Tutorial: Equipment.

- 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 und im Artikel AVR und Linux.

- 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 Artikel Exkurs: Makefiles).

- Wer unter Windows und Linux gleichermassen entwickeln will, der sollte sich die IDE Eclipse for C/C++ Developers und das AVR-Eclipse Plugin ansehen, beide sind unter Windows und Linux einfach zu installieren. Hier gibt es auch einen Artikel AVR Eclipse in dieser Wiki. Ebenfalls unter Linux und Windows verfügbar ist die Entwicklungsumgebung Code::Blocks (aktuelle, stabile Versionen sind als Nightly Builds regelmäßig im Forum verfügbar). Innerhalb dieser Entwicklungsumgebung können ohne die Installation zusätzlicher Plugins "AVR-Projekte" angelegt werden. Für Linux gibt es auch noch das KontrollerLab.

Was tun, wenn's nicht "klappt"?

- Herausfinden, ob es tatsächlich ein avr(-gcc) spezifisches Problem ist oder nur die eigenen C-Kenntnisse einer Auffrischung bedürfen. Allgemeine C-Fragen kann man eventuell "beim freundlichen Programmierer zwei Büro-, Zimmer- oder Haustüren weiter" loswerden. Ansonsten: C-Buch (gibt's auch "gratis" online) lesen.

- Die AVR Checkliste durcharbeiten.

- Die **Dokumentation der avr-libc** lesen, vor allem (aber nicht nur) den Abschnitt Related Pages/**Frequently Asked Questions** = Oft gestellte Fragen (und Antworten dazu). Z.Zt leider nur in englischer Sprache verfügbar.

- Den Artikel AVR-GCC in diesem Wiki lesen.

- Das GCC-Forum auf www.mikrocontroller.net nach vergleichbaren Problemen absuchen.
- Das avr-gcc-Forum bei AVRfreaks nach vergleichbaren Problemen absuchen.
- Das Archiv der avr-gcc Mailing-Liste nach vergleichbaren Problemen absuchen.
- Nach Beispielcode suchen. Vor allem im *Projects*-Bereich von AVRfreaks (anmelden).
- Google oder yahoo befragen schadet nie.
- Bei Problemen mit der Ansteuerung interner AVR-Funktionen mit C-Code: das Datenblatt des Controllers lesen (ganz und am Besten zweimal). Datenblätter sind auf den Atmel Webseiten als pdf-Dateien verfügbar. Das komplette Datenblatt (complete) und nicht die Kurzfassung (summary) verwenden.
- Die Beispieleprogramme im AVR-Tutorial sind zwar in AVR-Assembler verfasst, Erläuterungen und Vorgehensweisen sind aber auch auf C-Programme übertragbar.
- Einen Beitrag in eines der Foren oder eine Mail an die Mailing-Liste schreiben. Dabei möglichst viel Information geben: Controller, Compilerversion, genutzte Bibliotheken, Ausschnitte aus dem Quellcode oder besser ein Testprojekt mit allen notwendigen Dateien, um das Problem nachzuvollziehen, sowie genaue Fehlermeldungen bzw. Beschreibung des Fehlverhaltens. Bei Ansteuerung externer Geräte die Beschaltung beschreiben oder skizzieren (z. B. mit Andys ASCII Circuit). Siehe dazu auch: **"Wie man Fragen richtig stellt"**.

Erzeugen von Maschinencode

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

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

Unterverzeichnis des Projektverzeichnisses).

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

Einführungsbeispiel

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

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

Ein kurzes Wort zur Hardware: Bei diesem Programm werden alle Pins von PORTB auf Ausgang gesetzt, und einige davon werden auf HIGH andere auf LOW gesetzt. Das kann je nach angeschlossener Hardware an diesen Pins kritisch sein. Am ungefährlichsten ist es, wenn nichts an den Pins angeschlossen ist und man die Funktion des Programmes durch eine Spannungsmessung mit einem Multimeter kontrolliert. Die Spannung wird dabei zwischen GND-Pin und den einzelnen Pins von PORTB gemessen.

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

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

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

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

int main (void) {            // (2)

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

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

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

- In der mit (1) markierten Zeile wird eine sogenannte Header-Datei eingebunden. In io.h sind die Registernamen definiert, die im späteren Verlauf genutzt werden.
- Bei (2) beginnt das eigentliche Programm. Jedes C-Programm beginnt mit den Anweisungen in der Funktion main.
- (3) Die Anschlüsse eines AVR ("Beinchen") werden zu Blöcken zusammengefasst, einen solchen Block

bezeichnet man als Port. Beim ATmega16 hat jeder Port 8 Anschlüsse, bei kleineren AVRs können einem Port auch weniger als 8 Anschlüsse zugeordnet sein. Da per Definition (Datenblatt) alle gesetzten Bits in einem Datenrichtungsregister den entsprechenden Anschluss auf Ausgang schalten, werden mit DDRB=0xff alle Anschlüsse des Ports B als Ausgänge eingestellt.

- (4) stellt die Werte der Ausgänge ein. Die den ersten beiden Bits des Ports zugeordneten Anschlüsse (PB0 und PB1) werden 1, alle anderen Anschlüsse des Ports B (PB2-PB7) zu 0. Aktivierte Ausgänge (logisch 1 oder "high") liegen auf Betriebsspannung (VCC, meist 5 Volt), nicht aktivierte Ausgänge führen 0 Volt (GND, Bezugspotential). Es ist sinnvoll, sich möglichst frühzeitig eine alternative Schreibweise beizubringen, die wegen der leichteren Überprüfbarkeit und Portierbarkeit oft im weiteren Tutorial und in Forenbeiträgen benutzt wird. Die Zuordnung sieht in diesem Fall so aus:

```
PORTB = (1<<PB1) | (1<<PB0);
```

  Näheres dazu im Artikel Bitmanipulation.

- (5) ist die sogenannte Hauptschleife (main-loop). Dies ist eine Programmschleife, welche kontinuierlich wiederkehrende Befehle enthält. In diesem Beispiel ist sie leer. Der Controller durchläuft die Schleife immer wieder, ohne dass etwas passiert (außer das Strom verbraucht wird). Eine solche Schleife ist notwendig, da es auf dem Controller kein Betriebssystem gibt, das nach Beendigung des Programmes die Kontrolle übernehmen könnte. Ohne diese Schleife wäre der Zustand des Controllers nach dem Programmende undefiniert.

- (6) wäre das Programmende. Die Zeile ist nur aus Gründen der C-Kompatibilität enthalten: int main(void) besagt, dass die Funktion einen Wert zurückgibt. Die Anweisung wird aber nicht erreicht, da das Programm die Hauptschleife nie verlässt.

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

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

 Verzeichnis von D:\tmp\gcc_tut\quickstart

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

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

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

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

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

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

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

[...]
```

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

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

Ganzzahlige (Integer) Datentypen

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

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

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

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

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

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

typedef short              int16_t;
typedef unsigned short     uint16_t;

typedef long               int32_t;
typedef unsigned long      uint32_t;

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

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

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

Bitfelder

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

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

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

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

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

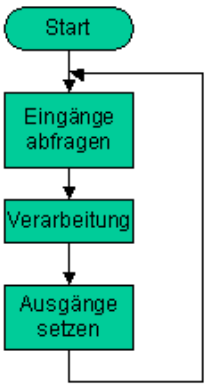
Bitfelder sparen Platz im RAM, zu Lasten von Platz im Flash, verschlechtern aber unter Umständen die Les- und Wartbarkeit des Codes. Anfängern wird deshalb geraten, ein "ganzes" Byte (uint8_t) zu nutzen, auch wenn nur ein Bitwert gespeichert werden soll.

Grundsätzlicher Programmaufbau eines µC-Programms

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

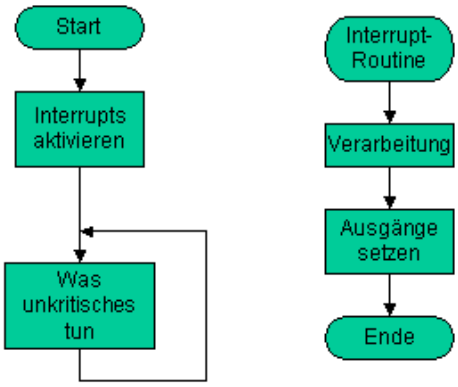
Sequentieller Programmablauf

Bei dieser Programmiertechnik wird eine Endlosschleife programmiert, welche im Wesentlichen immer den gleichen Aufbau hat. Es wird hier nach dem sogenannten EVA-Prinzip gehandelt. EVA steht für "Eingabe, Verarbeitung, Ausgabe":



Interruptgesteuerter Programmablauf

Bei dieser Methode werden beim Programmstart zuerst die gewünschten Interruptquellen aktiviert und dann in eine Endlosschleife gegangen, in welcher Dinge erledigt werden können, welche nicht zeitkritisch sind. Wenn ein Interrupt ausgelöst wird, so wird automatisch die zugeordnete Interruptfunktion ausgeführt.



Zugriff auf Register

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

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

Einzelne Register sind bei allen AVRs vorhanden, andere wiederum nur bei bestimmten Typen. So sind beispielsweise die Register, welche für den Zugriff auf den UART notwendig sind, selbstverständlich nur bei denjenigen Modellen vorhanden, welche über einen integrierten Hardware UART bzw. USART verfügen.

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

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

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

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

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

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

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

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

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

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

Schreiben in Register

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

Beispiel:

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

...

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

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

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

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

Die ausführliche Schreibweise sollte bevorzugt verwendet werden, da dadurch die Zuweisungen selbsterklärend sind und somit der Code leichter nachvollzogen werden kann. Atmel verwendet sie auch bei Beispielen in Datenblättern und in den allermeisten Quellcodes zu Application-Notes. Mehr zu der Schreibweise mit "|" und "<<" findet man im Artikel Bitmanipulation.

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

Verändern von Registerinhalten

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

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

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

bleibt erhalten.

Beispiel:

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

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

Beispiel:

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

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

Siehe auch:

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

Lesen aus Registern

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

Beispiel:

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

uint8_t foo;

//...

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

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

```
#define MEINBIT0 0
#define MEINBIT2 2

uint8_t i;

extern test1();

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

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

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

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

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

bit_is_set (<Register>,<Bitnummer>)

Die Funktion *bit_is_set* prüft, ob ein Bit gesetzt ist. Wenn das Bit gesetzt ist, wird ein Wert ungleich 0 zurückgegeben. Genau genommen ist der Rückgabewert 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 (eigentlich Makros) bit_is_clear bzw. bit_is_set sind nicht erforderlich, man kann und sollte C-Syntax verwenden, die universell verwendbar und portabel ist. Siehe auch Bitmanipulation.

Warten auf einen bestimmten Zustand

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

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

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

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

#define WARTEPIN PINA
#define WARTEBIT PA2

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

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

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

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

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

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

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

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

Siehe auch:

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

16-Bit Register (ADC, ICR1, OCR1x, TCNT1, UBRR)

Einige der Portregister in den AVR-Controllern sind 16 Bit breit. Im Datenblatt sind diese Register üblicherweise mit dem Suffix "L" (Low-Byte) und "H" (High-Byte) versehen. Die avr-libc definiert zusätzlich die meisten dieser Variablen die Bezeichnung ohne "L" oder "H". Auf diese Register kann dann direkt zugegriffen werden. Die ist zum Beispiel der Fall für Register wie ADC oder TCNT1.

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

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

Bei anderen Registern, wie zum Beispiel Baudraten-Register, liegen High- und Low-Teil nicht direkt nebeneinander im SFR-Bereich, so dass ein 16-Bit Zugriff nicht möglich ist und der Zugriff zusammengebastelt werden muss:

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

#ifndef F_CPU
#define F_CPU 3686400
#endif
#define UART_BAUD_RATE 9600

...
   uint16_t baud = F_CPU / (UART_BAUD_RATE * 16L) -1;

   UBRRH = (uint8_t) (baud >> 8);
   UBRRL = (uint8_t) baud;
...
```

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

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

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

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

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

Das bedeutet für die Reihenfolge:

- Lesezugriff: Erst Low-Byte, dann High-Byte
- Schreibzugriff: Erst High-Byte, dann Low-Byte

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

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

Nach einem Zugriff auf ADCL muss grundsätzlich ADCH gelesen werden!

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

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

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

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

Im Umgang mit 16-Bit Registern siehe auch:

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

IO-Register als Parameter und Variablen

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

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

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

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

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

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

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

/* Beispiel für einen Funktionsaufruf: */

//...
uint8_t i;
//...
i = key_pressed( &PINB, PB1 );
//...
```

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

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

Zugriff auf IO-Ports

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

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

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

Die Verarbeitung von analogen Eingangswerten und die Ausgabe von Analogwerten wird in Kapitel Analoge Ein- und Ausgabe behandelt.

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

	Datenrichtungsregister für Port**x**.

DDRx	**x** entspricht **A**, **B**, **C**, **D** usw. (abhängig von der Anzahl der Ports des verwendeten AVR). Bit im Register gesetzt (1) für Ausgang, Bit gelöscht (0) für Eingang.
PINx	Eingangsadresse für Port**x**. 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 Port**x**. Dieses Register wird verwendet, um die Ausgänge eines Ports anzusteuern. Bei Pins, die mittels DDRx auf Eingang geschaltet wurden, können über PORTx die internen Pull-Up Widerstände aktiviert oder deaktiviert werden (1 = aktiv).

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

Datenrichtung bestimmen

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

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

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

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

In C liest sich das dann so:

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

...

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

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

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

...
```

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

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

Vordefinierte Bitnummern für I/O-Register

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

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

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

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

Digitale Signale

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

Ausgänge

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

Mit dem Befehl

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

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

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

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

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

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

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

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

Siehe auch Bitmanipulation

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

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

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

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

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

war:

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

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

Eingänge (Wie kommen Signale in den µC)

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

Signalkopplung

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

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

Die Masse der beiden Schaltungen muss selbstverständlich miteinander verbunden werden. Der Software selber ist es natürlich letztendlich egal, wie das Signal eingespeist wird. Wir können ja ohnehin lediglich prüfen, ob an einem Pin unseres Controllers eine logische 1 (Spannung größer ca. 0,7*Vcc) oder eine logische 0 (Spannung kleiner ca. 0,2*Vcc) 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.

Spannungstabelle
(ca. Grenzwerte)

	Low	High
bei 5 V	1 V	3,5 V
bei 3,3 V	0,66 V	2,31 V
bei 1,8 V	0,36 V	1,26 V

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

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

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

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

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

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

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

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

Siehe auch Bitmanipulation#Bits_prüfen

Interne Pull-Up Widerstände

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

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

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

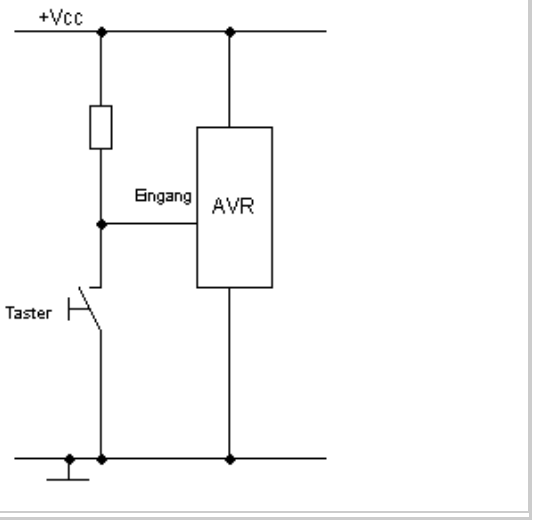
Im Beispiel werden alle Pins des Ports D als Eingänge geschaltet und alle Pull-Up Widerstände aktiviert. Weiterhin wird Pin PC7 als Eingang geschaltet und dessen interner Pull-Up Widerstand aktiviert, ohne die Einstellungen für die anderen Portpins (PC0-PC6) zu verändern.

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

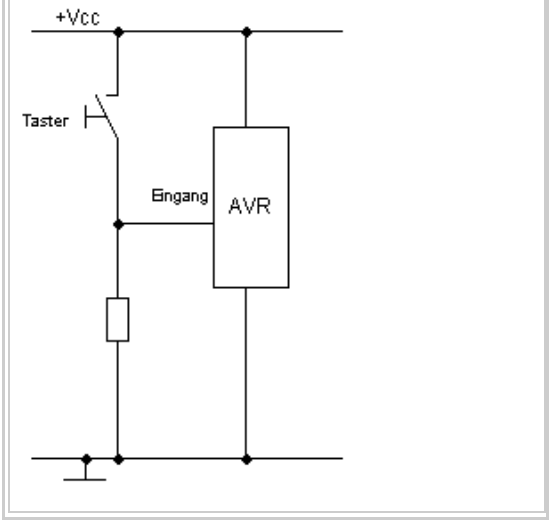
Tasten und Schalter

Der Anschluss mechanischer Kontakte an den Mikrocontroller, ist zwischen zwei unterschiedliche Methoden zu unterscheiden: *Active Low* und *Active High*.

Anschluss mechanischer Kontakte an einen µC



Active Low: Bei dieser Methode wird der Kontakt zwischen den Eingangspin des Controllers und Masse geschaltet. Damit bei offenem Schalter der Controller kein undefiniertes Signal bekommt, wird zwischen die Versorgungsspannung und den Eingangspin ein sogenannter **Pull-Up** Widerstand geschaltet. Dieser dient dazu, den Pegel bei geöffnetem Schalter auf logisch 1 zu ziehen.



Active High: Hier wird der Kontakt zwischen die Versorgungsspannung und den Eingangspin geschaltet. Damit bei offener Schalterstellung kein undefiniertes Signal am Controller ansteht, wird zwischen den Eingangspin und die Masse ein **Pull-Down** Widerstand geschaltet. Dieser dient dazu, den Pegel bei geöffneter Schalterstellung auf logisch 0 zu halten.

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

(Tasten-)Entprellung

Alle mechanischen Kontakte, sei es von Schaltern, Tastern oder auch von Relais, haben die unangenehme Eigenschaft zu prellen. Dies bedeutet, dass beim Schalten eines Kontaktes derselbe nicht direkt den endgültigen Zustand aufweist, sondern zwischenzeitlich möglicherweise mehrfach ein- und ausschaltet.

Soll nun mit einem Mikrocontroller gezählt werden, wie oft ein solcher Kontakt geschaltet wird, muss das Prellen des Kontakts berücksichtigt werden, da sonst pro Schaltvorgang möglicherweise mehrfache Impulse gezählt werden. Diesem Phänomen muss beim Schreiben des Programms unbedingt Rechnung getragen werden.

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

```
#include <avr/io.h>
#include <inttypes.h>
#ifndef F_CPU
#warning "F_CPU war noch nicht definiert, wird nun mit 3686400 definiert"
#define F_CPU 3686400UL     /* Quarz mit 3.6864 Mhz  */
```

```
#endif
#include <util/delay.h>     /* bei alter avr-libc: #include <avr/delay.h> */

/* Einfache Funktion zum Entprellen eines Tasters */
inline uint8_t debounce(volatile uint8_t *port, uint8_t pin)
{
    if ( !(*port & (1 << pin)) )
    {
        /* Pin wurde auf Masse gezogen, 100ms warten   */
        _delay_ms(50);   // Maximalwert des Parameters an _delay_ms
        _delay_ms(50);   // beachten, vgl. Dokumentation der avr-libc
        if ( *port & (1 << pin) )
        {
            /* Anwender Zeit zum Loslassen des Tasters geben */
            _delay_ms(50);
            _delay_ms(50);
            return 1;
        }
    }
    return 0;
}

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

Die obige Routine hat leider mehrere Nachteile:

- sie detektiert nur das Loslassen (unergonomisch)
- sie verzögert die Mainloop immer um 100ms bei gedrückter Taste
- sie verliert Tastendrücke, je mehr die Mainloop zu tun hat.

Eine ähnlich einfach zu benutzende Routine, aber ohne all diese Nachteile findet sich im Forenthread Entprellung für Anfänger. und weiteres zum Thema entprellen im Artikel Entprellung.

Analoge Ein- und Ausgabe

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

Es gibt innerhalb der ATMega- und ATTiny-AVR Reihe keine Typen mit eingebautem Digital-Analog-Konverter (DAC) - diese Funktion ist erst ab der XMEGA-Reihe der AVR-Familie verfügbar, die aber wegen ihrer vielen Unterschiede im Umfang dieses Tutorials nicht behandelt wird. Die Umsetzung zu einer analogen Spannung muss daher durch externe Komponenten vorgenommen werden. Das kann z. B. durch PWM und deren Filterung zu (fast) DC, oder einem sogenannten R2R-Netzwerk erfolgen.

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

AVR anzusteuern.

AC (Analog Comparator)

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

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

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

Das Steuer- bzw. Statusregister ist wie folgt aufgebaut:

ACSR - Analog Comparator Status Register

Bit	7	6	5	4	3	2	1	0
Name	**ACD**	**ACBG**	**ACO**	**ACI**	**ACIE**	**ACIC**	**ACIS1**	**ACIS0**
R/W	R/W	R/W	R	R/W	R/W	R/W	R/W	R/W
Initialwert	0	0	n/a	0	0	0	0	0

Bit 7 ACD
: Analog Comparator Disable: 0 = Comparator ein, 1 = Comparator aus. Wird dieses Bit geändert kann ein Interrupt ausgelöst werden. Soll dies vermieden werden muss das Bit 3 ACIE ggf. abgeschaltet werden.

Bit 6 ACBG
: Analog Comparator Bandgap Select: Ermöglicht das umschalten zwischen interner und externer Referenzspannung. 1 = interne (~1,3 Volt), 0 = externe Referenzspannung (an Pin AIN0)

Bit 5 ACO
: Analog Comparator Output: Hier wird das Ergebnis des Vergleichs angezeigt. Es liegt typischerweise nach 1-2 Taktzyklen vor.

  IST < SOLL → 1
  IST > SOLL → 0

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

Bit 3 ACIE
: Analog Comparator Interrupt Enable: Ist das Bit auf 1 gesetzt wird immer ein Interrupt ausgelöst wenn das Ereignis das in Bit 1 und 0 definiert ist eintritt.

Bit 2 ACIC
: Analog Comparator Input Capture Enable: Wird das Bit gesetzt wird der Comparatorausgang intern mit dem Counter 1 verbunden. Es könnten damit z.b. die Anzahl der Vergleiche im Counter1 gezählt werden. Um den Comparator an den Timer1 Input Capture Interrupt zu verbinden muss im Timerregister das TICIE1 Bit auf 1 gesetzt werden. Der Trigger wird immer dann ausgelöst wenn das in Bit 1 und 0 definierte Ereignis eintritt.

Bit 1,0 ACIS1,ACIS0
: Analog Comparator Interrupt select: Hier wird definiert welche Ereignisse einen Interrupt auslösen sollen:

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

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

ADC (Analog Digital Converter)

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

Ein ADC mit 8 Bit Auflösung kann somit das analoge Signal mit einer Genauigkeit von 1/256 des Maximalwertes darstellen. Wenn wir nun mal annehmen, wir hätten eine Spannung zwischen 0 und 5 Volt und eine Auflösung von 3 Bit, dann könnten die Werte 0V, 0.625V, 1.25, 1.875V, 2.5V, 3.125V, 3.75, 4.375, 5V daherkommen, siehe dazu folgende Tabelle:

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

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

Der interne ADC im AVR

Wenn es einmal etwas genauer sein soll, dann müssen wir auf einen AVR mit eingebautem Analog-Digital-Wandler (ADC) zurückgreifen, die über mehrere Kanäle verfügen. Kanäle heißt in diesem Zusammenhang, dass zwar bis zu zehn analoge Eingänge am AVR verfügbar sind, aber nur ein "echter" Analog-Digital-

Wandler zur Verfügung steht, vor der eigentlichen Messung ist also einzustellen, welcher Kanal ("Pin") mit dem Wandler verbunden und gemessen wird.

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

Das Ergebnis der Analog-Digital-Wandlung wird auf eine Referenzspannung bezogen. Aktuelle AVRs bieten 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.
- Es kann die Spannung AVcc als Referenzspannung herangezogen werden

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

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

Einfache Wandlung (Single Conversion)
: In dieser Betriebsart wird der Wandler bei Bedarf vom Programm angestoßen für jeweils eine Messung.

Frei laufend (Free Running)
: In dieser Betriebsart erfasst der Wandler permanent die anliegende Spannung und schreibt diese in das **ADC Data Register**.

Die Register des ADC

Der **ADC** verfügt über eigene Register. Im Folgenden die Registerbeschreibung eines ATMega16, welcher über 8 ADC-Kanäle verfügt. Die Register unterscheiden sich jedoch nicht erheblich von denen anderer AVRs (vgl. Datenblatt).

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

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

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

ADEN (**AD**C **En**able)

Dieses Bit muss gesetzt werden, um den **ADC** überhaupt zu aktivieren. Wenn das Bit

nicht gesetzt ist, können die Pins wie normale I/O-Pins verwendet werden.

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

Mit diesem Bit wird ein Messvorgang gestartet. In der frei laufenden Betriebsart muss das Bit gesetzt werden, um die kontinuierliche Messung zu aktivieren.
Wenn das Bit nach dem Setzen des **ADEN**-Bits zum ersten Mal gesetzt wird, führt der Controller zuerst eine zusätzliche Wandlung und erst dann die eigentliche Wandlung aus. Diese zusätzliche Wandlung wird zu Initialisierungszwecken durchgeführt.
Das Bit bleibt nun so lange auf 1, bis die Umwandlung abgeschlossen ist, im Initialisierungsfall entsprechend bis die zweite Umwandlung erfolgt ist und geht danach auf 0.

ADFR (**AD**C **F**ree **R**un select)

Mit diesem Bit wird die Betriebsart eingestellt.
Ist das Bit auf 1 gesetzt arbeitet der ADC im "Free Running"-Modus. Dabei wird das Datenregister permanent aktualisiert. Ist das Bit hingegen auf 0 gesetzt, macht der ADC nur eine "Single Conversion".

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

ADCSRA

Dieses Bit wird vom **ADC** gesetzt, sobald eine Umwandlung erfolgt ist und das **ADC Data Register** aktualisiert wurde. Das Bit wird bei lesendem Zugriff auf **ADC(L,H)** automatisch (d.h. durch die Hardware) gelöscht.
Wenn das **ADIE** Bit sowie das **I-Bit** im AVR **Statusregister** gesetzt ist, wird der **ADC Interrupt** ausgelöst und die Interrupt-Behandlungsroutine aufgerufen.
Das Bit wird automatisch gelöscht, wenn die Interrupt-Behandlungsroutine aufgerufen wird. Es kann jedoch auch gelöscht werden, indem eine logische 1 in das Register geschrieben wird.

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

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

ADPS2...ADPS0 (**AD**C **P**rescaler **S**elect Bits)

Diese Bits bestimmen den Teilungsfaktor zwischen der Taktfrequenz und dem Eingangstakt des **ADC**.
Der **ADC** benötigt einen eigenen Takt, welchen er sich selber aus der CPU-Taktfreqenz erzeugt. Der **ADC**-Takt sollte zwischen 50 und 200kHz sein.
Der Vorteiler muss also so eingestellt werden, dass die CPU-Taktfrequenz dividiert durch den Teilungsfaktor einen Wert zwischen 50-200kHz ergibt.
Bei einer CPU-Taktfrequenz von 4MHz beispielsweise rechnen wir

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

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

Somit kann hier der Teilungsfaktor 32 oder 64 verwendet werden. Im Interesse der schnelleren Wandlungszeit werden wir hier den Faktor 32 einstellen.

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

ADCL

ADCH

ADC Data Register

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

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

oder

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

ADC **Mu**ltiple**x**er Select Register

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

Bit	7	6	5	4	3	2	1	0
Name	REFS1	REFS0	ADLAR	MUX4	MUX3	MUX2	MUX1	MUX0
R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W
Initialwert	0	0	0	0	0	0	0	0

REFS1...REFS0 (**Ref**erence**S**election Bits)

Mit diesen Bits kann die Referenzspannung eingestellt werden. Bei der Umstellung sind Wartezeiten zu beachten, bis die ADC-Hardware einsatzfähig ist (Datenblatt und [1]):

REFS1	REFS0	Referenzspanung
0	0	Externes AREF

ADMUX

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

ADLAR (**ADC L**eft **A**djust **R**esult)

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

MUX4...MUX0

Mit diesen 5 Bits wird der zu messende Kanal bestimmt. Wenn man einen einfachen 1-kanaligen ADC verwendet wird einfach die entsprechende Pinnummer des Ports in die Bits 0...2 eingeschrieben.
Wenn das Register beschrieben wird, während dem eine Umwandlung läuft, so wird zuerst die aktuelle Umwandlung auf dem bisherigen Kanal beendet. Dies ist vor allem beim frei laufenden Betrieb zu berücksichtigen.

Eine Empfehlung ist deswegen diese, dass der frei laufende Betrieb nur bei einem einzelnen zu verwendenden Analogeingang verwendet werden sollte, wenn man sich Probleme bei der Umschalterei ersparen will.

Nutzung des ADC

Um den **ADC** zu aktivieren, müssen wir das **ADEN**-Bit im **ADCSR**-Register setzen. Im gleichen Schritt legen wir auch die Betriebsart fest.

Ein kleines Beispiel für den "single conversion"-Mode bei einem ATmega169 und Nutzung der internen Referenzspannung (beim '169 1,1V bei anderen AVRs auch 2,56V). D.h. das Eingangssignal darf diese Spannung nicht überschreiten, gegebenenfalls muss es mit einem Spannungsteiler verringert werden. Das Ergebnis der Routine ist der ADC-Wert, also 0 für 0-Volt und 1023 für V_ref-Volt.

In der praktischen Anwendung wird man zum Programmstart den ADC erst einmal grundlegend konfigurieren und dann auf verschiedenen Kanälen messen. Diese beiden Dinge sollte man meist trennen, denn das Einschalten des ADC und vor allem der Referenzspannung dauert ein paar Dutzend Mikrosekunden. Außerdem ist das erste Ergebnis nach dem Einschalten ungültig und muss verworfen werden.

```
/* ADC initialisieren */
void ADC_Init(void) {

  uint16_t result;

//  ADMUX = (0<<REFS1) | (1<<REFS0);      // AVcc als Referenz benutzen
  ADMUX = (1<<REFS1) | (1<<REFS0);      // interne Referenzspannung nutzen
  ADCSRA = (1<<ADPS1) | (1<<ADPS0);     // Frequenzvorteiler
  ADCSRA |= (1<<ADEN);                  // ADC aktivieren

  /* nach Aktivieren des ADC wird ein "Dummy-Readout" empfohlen, man liest
     also einen Wert und verwirft diesen, um den ADC "warmlaufen zu lassen" */
```

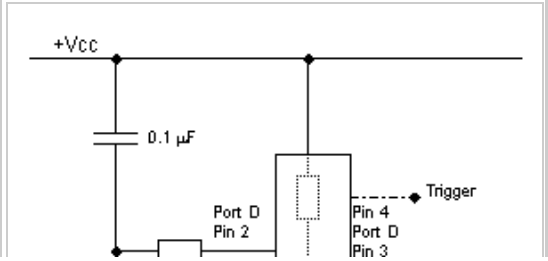
```
  ADCSRA |= (1<<ADSC);                  // eine ADC-Wandlung
  while (ADCSRA & (1<<ADSC) ) {}        // auf Abschluss der Konvertierung warten
  /* ADCW muss einmal gelesen werden, sonst wird Ergebnis der nächsten
     Wandlung nicht übernommen. */
  result = ADCW;
}

/* ADC Einzelmessung */
uint16_t ADC_Read( uint8_t channel )
{
  // Kanal waehlen, ohne andere Bits zu beeinflußen
  ADMUX = (ADMUX & ~(0x1F)) | (channel & 0x1F);
  ADCSRA |= (1<<ADSC);            // eine Wandlung "single conversion"
  while (ADCSRA & (1<<ADSC) ) {}  // auf Abschluss der Konvertierung warten
  return ADCW;                    // ADC auslesen und zurückgeben
}

/* ADC Mehrfachmessung mit Mittelwertbbildung */
uint16_t ADC_Read_Avg( uint8_t channel, uint8_t average )
{
  uint32_t result = 0;

  for (uint8_t i = 0; i < average; ++i )
    result += ADC_Read( channel );

  return (uint16_t)( result / average );
}

...

/* Beispielaufrufe: */

int main()
{
  uint16_t adcval;
  ADC_Init();

  while( 1 ) {
    adcval = ADC_Read(0);  // Kanal 0
    // mach was mit adcval

    adcval = ADC_Read_Avg(2, 4);  // Kanal 2, Mittelwert aus 4 Messungen
    // mach was mit adcval
  }
}
```

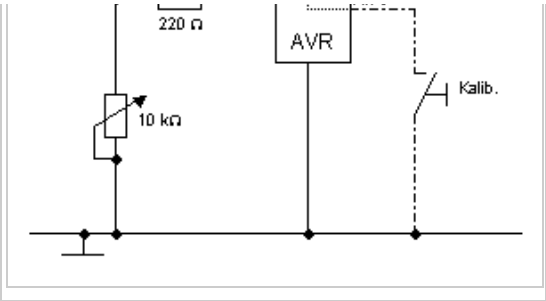
Im Beispiel läuft der ADC ständig. Wenn man diesen Strom sparen will, z.B. bei Verwendung des Sleep Modes, muss man den ADC nach jeder Messung abschalten und vor der nächsten Messung wieder einschalten, wobei auch dann wieder eine kleine Pause und Anfangswandlung nötig sind.

Analog-Digital-Wandlung ohne internen ADC

Messen eines Widerstandes

Analoge Werte lassen sich ohne Analog-Digital-Wandler auch indirekt ermitteln. Im Folgenden wird die Messung des an einem Potentiometer eingestellten Widerstands anhand der Ladekurve eines Kondensators erläutert. Bei dieser Methode wird nur ein Portpin benötigt, ein Analog-Digital-Wandler oder Analog-Comparator ist nicht erforderlich. Es wird dazu ein Kondensator und der Widerstand (das Potentiometer) in Reihe zwischen Vorsorgungsspannung und Masse/GND geschaltet (sogen. RC-





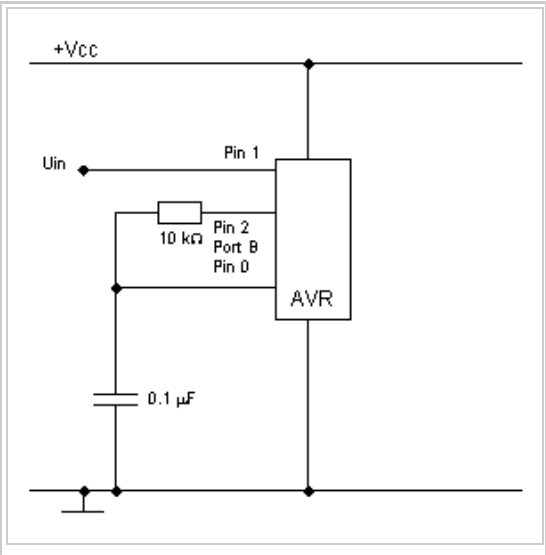
Netzwerk). Zusätzlich wird eine Verbindung der Leitung zwischen Kondensator und Potentiometer zu einem Portpin des Controllers hergestellt. Die folgende Abbildung verdeutlicht die erforderliche Schaltung.

Wird der Portpin des Controllers auf Ausgang konfiguriert (im Beispiel *DDRD |= (1<<PD2)*) und dieser Ausgang auf Logisch 1 ("High", *PORTD |= (1<<PD2)*) geschaltet, liegt an beiden "Platten" des Kondensators das gleiche Potential **VCC** an und der Kondensator somit 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).

Nach einer gewissen Zeit ist der Kondensator entladen und der Portpin wird als Eingang konfiguriert (*DDRD &= ~(1<<PD2); PORTD &= ~(1<<PD2)*), wodurch dieser hochohmig wird. Der Status des Eingangspin (in PIND) ist Logisch 1 (High). Der Kondensator lädt sich jetzt über das Poti auf, dabei steigt der Spannungsabfall über dem Kondensator und derjenige über dem Poti sinkt. Fällt nun der Spannungsabfall über dem Poti unter die Threshold-Spannung des Eingangspins (2/5 Vcc, also ca. 2V), wird das Eingangssignal als LOW erkannt (Bit in PIND wird 0). Die Zeitspanne zwischen der Umschaltung von Entladung auf Aufladung und dem Wechsel des Eingangssignals von High auf Low ist ein Maß für den am Potentiometer eingestellten Widerstand. Zur Zeitmessung kann einer der im Controller vorhandenen Timer genutzt werden. Der 220 Ohm Widerstand dient dem Schutz des Controllers. Es würde sonst bei Maximaleinstellung des Potentionmeters (hier 0 Ohm) ein zu hoher Strom fließen, der die Ausgangsstufe des Controllers zerstört.

Mit einem weiteren 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.

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.

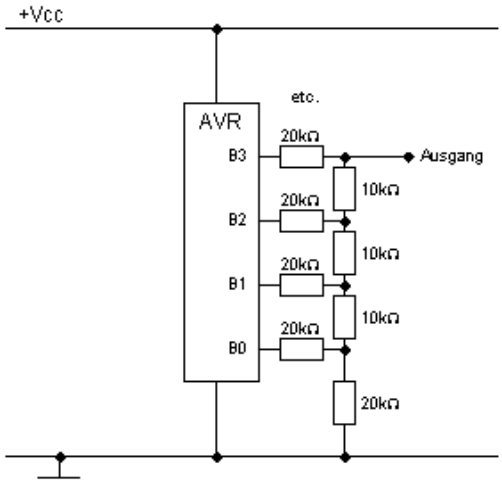
Der Vorteil dieser Schaltung liegt allerdings darin, dass damit direkt Spannungen gemessen werden können.

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:

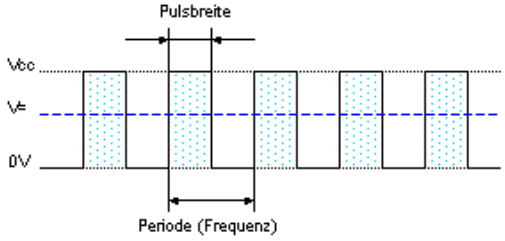


Es sollten selbstverständlich möglichst genaue Widerstände verwendet werden, also nicht unbedingt solche mit einer Toleranz von 10% oder mehr. Weiterhin empfiehlt es sich, je nach Anwendung den Ausgangsstrom über einen Operationsverstärker zu verstärken.

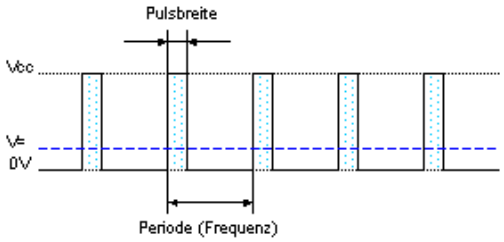
PWM (Pulsweitenmodulation)

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



Diese Rechteckspannung hat nun einen arithmetischen Mittelwert, der je nach Pulsbreite kleiner oder größer ist.



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

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

Hinweis

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

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

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

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

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

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

COM1A1	COM1A0	Bedeutung
0	0	Keine Wirkung, Pin wird nicht geschaltet.
0	1	Keine Wirkung, Pin wird nicht geschaltet.
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:

alte Schreibweise (PWMxx wird nicht mehr akzeptiert)

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

neue Schreibweise

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

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

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

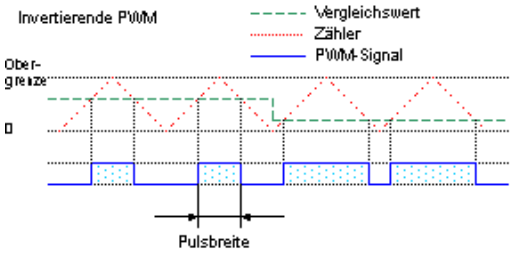
Also um einen Takt von CK / 1024 zu generieren, verwenden wir folgenden Befehl:

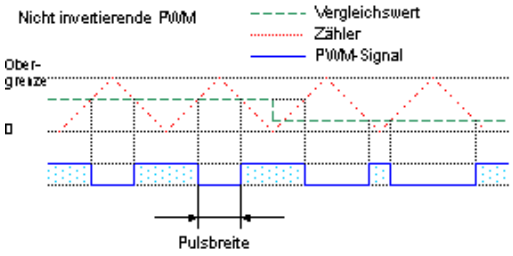
```
TCCR1B = (1<<CS12) | (1<<CS10);
```

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

```
OCR1A = xxx;
```

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





Ach ja, fast hätte ich's vergessen. Das generierte **PWM**-Signal wird am Output Compare Pin **OC1** des Timers ausgegeben und leider können wir deshalb auch beim AT90S2313 nur ein einzelnes **PWM**-Signal mit dieser Methode generieren. Andere AVR-Typen verfügen über bis zu vier PWM-Ausgänge. Zu beachten ist außerdem, das wenn der OC Pin aktiviert ist, er nichtmehr wie üblich funktioniert und z. B. nicht einfach über PINx ausgelesen werden kann.

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

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

int main()
{
  // OC1A auf Ausgang
  DDRB = (1 << PB1 );  //ATMega8
  // DDRD = (1 << PD5 ); //ATMega16
  //
  // Timer 1 einstellen
  //
  // Modus 14:
  //    Fast PWM, Top von ICR1
  //
  //    WGM13     WGM12    WGM11     WGM10
  //      1         1        1         0
  //
  //    Timer Vorteiler: 1
  //     CS12      CS11     CS10
  //       0         0        1
  //
  //  Steuerung des Ausgangsport: Set at BOTTOM, Clear at match
  //     COM1A1    COM1A0
  //       1         0

  TCCR1A = (1<<COM1A1) | (1<<WGM11);
  TCCR1B = (1<<WGM13) | (1<<WGM12) | (1<<CS10);

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

  ICR1 = 0x6FFF;

  // der Compare Wert
  // Wenn der Zähler diesen Wert erreicht, wird mit
  // obiger Konfiguration der OC1A Ausgang abgeschaltet
  // Sobald der Zähler wieder bei 0 startet, wird der
  // Ausgang wieder auf 1 gesetzt
```

```
  //
  // Durch Verändern dieses Wertes, werden die unterschiedlichen
  // PWM Werte eingestellt.

  OCR1A = 0x3FFF;

  while (1) {}
}
```

PWM-Mode Tabelle aus dem Datenblatt des ATmega8515

Mode	WGM13	WGM12	WGM11	WGM10	Timer/Counter Mode of Operation	TOP	Update of OCR1x at	TOV1 Flag set on
0	0	0	0	0	Normal	0xFFFF	Immediate	MAX
1	0	0	0	1	PWM, Phase Correct, 8-Bit	0x00FF	TOP	BOTTOM
2	0	0	1	0	PWM, Phase Correct, 9-Bit	0x01FF	TOP	BOTTOM
3	0	0	1	1	PWM, Phase Correct, 10-Bit	0x03FF	TOP	BOTTOM
4	0	1	0	0	CTC	OCR1A	Immediate	MAX
5	0	1	0	1	Fast PWM, 8-Bit	0x00FF	BOTTOM	TOP
6	0	1	1	0	Fast PWM, 9-Bit	0x01FF	BOTTOM	TOP
7	0	1	1	1	Fast PWM, 10-Bit	0x03FF	BOTTOM	TOP
8	1	0	0	0	PWM, Phase an Frequency Correct	ICR1	BOTTOM	BOTTOM
9	1	0	0	1	PWM, Phase an Frequency Correct	OCR1A	BOTTOM	BOTTOM
10	1	0	1	0	PWM, Phase Correct	ICR1	TOP	BOTTOM
11	1	0	1	1	PWM, Phase an Frequency Correct	OCR1A	TOP	BOTTOM
12	1	1	0	0	CTC	ICR1	Immediate	MAX
13	1	1	0	1	Reserved	-	-	-
14	1	1	1	0	Fast PWM	ICR1	BOTTOM	TOP
15	1	1	1	1	Fast PWM	OCR1A	BOTTOM	TOP

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

Warteschleifen (delay.h)

Der Programmablauf kann verschiedene Arten von Wartefunktionen erfordern:

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

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

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

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

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

avr-libc Versionen kleiner 1.6

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

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

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

/*
 lange, variable Verzögerungszeit, Einheit in Millisekunden

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

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

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

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

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

    return 0;
}
```

avr-libc Versionen ab 1.6

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

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

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

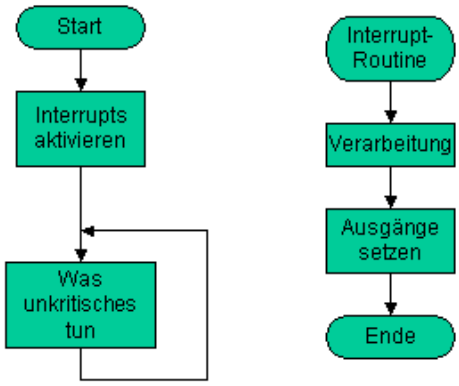
```
#include <avr/io.h>
#ifndef F_CPU
/* Definiere F_CPU, wenn F_CPU nicht bereits vorher definiert
   (z. B. durch Übergabe als Parameter zum Compiler innerhalb
   des Makefiles). Zusätzlich Ausgabe einer Warnung, die auf die
   "nachträgliche" Definition hinweist */
#warning "F_CPU war noch nicht definiert,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

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

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

    }
    return 0;
}
```

Die _delay_ms() und die _delay_us aus **avr-libc 1.7.0** sind fehlerhaft. _delay_ms () läuft 4x schneller als erwartet. Abhilfe ist eine korrigierte Includedatei: [2]

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.

Siehe auch

- Ausführlicher Thread im Forum
- Artikel Interrupt
- Artikel Multitasking

Anforderungen an Interrupt-Routinen

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

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

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

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

Interrupt-Quellen

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

- Reset

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

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

Register

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

GIMSK

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

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

INT1 (External **Int**errupt Request **1** Enable)

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

INT0 (External **Int**errupt Request **0** Enable)

Wenn dieses Bit gesetzt ist, wird ein Interrupt ausgelöst, wenn am **INT0**-Pin eine steigende oder fallende (je nach Konfiguration im **MCUCR**) Flanke erkannt wird.
Das Global Enable Interrupt Flag muss selbstverständlich auch gesetzt sein.
Der Interrupt wird auch ausgelöst, wenn der Pin als Ausgang geschaltet ist. Auf diese Weise bietet sich die Möglichkeit, Software-Interrupts zu realisieren.

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

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

INTF1 (External **Int**errupt Flag **1**)

GIFR

Dieses Bit wird gesetzt, wenn am **INT1**-Pin eine Interrupt-Kondition, entsprechend der Konfiguration, erkannt wird. Wenn das Global Enable Interrupt Flag gesetzt ist, wird die Interruptroutine angesprungen.
Das Flag wird automatisch gelöscht, wenn die Interruptroutine beendet ist. Alternativ kann das Flag gelöscht werden, indem der Wert **1(!)** eingeschrieben wird.

INTF0 (External **Int**errupt Flag **0**)

Dieses Bit wird gesetzt, wenn am **INT0**-Pin eine Interrupt-Kondition, entsprechend der Konfiguration, erkannt wird. Wenn das Global Enable Interrupt Flag gesetzt ist, wird die Interruptroutine angesprungen.
Das Flag wird automatisch gelöscht, wenn die Interruptroutine beendet ist. Alternativ kann das Flag gelöscht werden, indem der Wert **1(!)** eingeschrieben wird.

MCUCR

MCU Control **R**egister.

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

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

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

Dieses Bit muss gesetzt sein, um den Controller mit dem **SLEEP**-Befehl in den Schlafzustand versetzen zu können.
Um den Schlafmodus nicht irrtümlich einzuschalten, wird empfohlen, das Bit erst unmittelbar vor Ausführung des **SLEEP**-Befehls zu setzen.

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

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

ISC11, **ISC10** (**I**nterrupt **S**ense **C**ontrol **1** Bits)

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

ISC11	**ISC10**	**Bedeutung**
0	0	Low Level an **INT1** erzeugt einen Interrupt. Der Interrupt wird getriggert, solange der Pin auf 0 bleibt.
0	1	Reserviert
1	0	Die fallende Flanke an **INT1** erzeugt einen Interrupt.
1	1	Die steigende Flanke an **INT1** erzeugt einen Interrupt.

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

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

ISC01	ISC00	Bedeutung
0	0	Low Level an **INT0** erzeugt einen Interrupt. Der Interrupt wird getriggert, solange der Pin auf 0 bleibt.
0	1	Reserviert
1	0	Die fallende Flanke an **INT0** erzeugt einen Interrupt.
1	1	Die steigende Flanke an **INT0** erzeugt einen Interrupt.

Allgemeines über die Interrupt-Abarbeitung

Wenn ein Interrupt eintrifft, wird automatisch das **Global Interrupt Enable** Bit im Status Register **SREG** gelöscht und alle weiteren Interrupts unterbunden. Obwohl es möglich ist, zu diesem Zeitpunkt bereits wieder das GIE-bit zu setzen, wird dringend davon abgeraten. Dieses wird nämlich automatisch gesetzt, wenn die Interruptroutine beendet wird. Wenn in der Zwischenzeit weitere Interrupts eintreffen, werden die zugehörigen Interrupt-Bits gesetzt und die Interrupts bei Beendigung der laufenden Interrupt-Routine in der Reihenfolge ihrer Priorität ausgeführt. Dies kann eigentlich nur dann zu Problemen führen, wenn ein hoch priorisierter Interrupt ständig und in kurzer Folge auftritt. Dieser sperrt dann möglicherweise alle anderen Interrupts mit niedrigerer Priorität. Dies ist einer der Gründe, weshalb die Interrupt-Routinen sehr kurz gehalten werden sollen.

Interrupts mit dem AVR GCC Compiler (WinAVR)

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

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

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

```
    sei();
```

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

```
    cli();
```

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

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

//...

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

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

   /* hier "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!
    //...

}
```

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

ISR

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

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

Mit *ISR* wird eine Funktion für die Bearbeitung eines Interrupts eingeleitet. Als Argument muss dabei die Benennung des entsprechenden Interruptvektors angegeben werden. Diese sind in den jeweiligen Includedateien IOxxxx.h zu finden. Die Bezeichnung entspricht dem Namen aus dem Datenblatt, bei dem die Leerzeichen durch Unterstriche ersetzt sind und ein *_vect* angehängt ist.

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

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

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

/* Interrupt vectors */

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

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

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

/* Timer/Counter2 Overflow */
```

```
#define TIMER2_OVF_vect                 _VECTOR(4)
#define SIG_OVERFLOW2                   _VECTOR(4)

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

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

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

//...
```

Mögliche Funktionsrümpfe für Interruptfunktionen sind zum Beispiel:

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

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

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

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

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

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

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

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

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

Unterbrechbare Interruptroutinen

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

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

ISR(XXX,ISR_NOBLOCK) /* veraltet: INTERRUPT(SIG_OVERFLOW0) */
{
    /* Interrupt-Code */
}
```

Hierbei steht XXX für den oben beschriebenen Namen des Vektors (also z. B. *TIMER0_OVF_vect*). Der Unterschied im Vergleich zu einer herkömmlichen ISR ist, dass hier beim Aufrufen der Funktion das **Global Enable Interrupt** Bit durch Einfügen einer SEI-Anweisung direkt wieder gesetzt und somit alle Interrupts zugelassen werden – auch XXX-Interrupts.

Bei unsachgemässer Handhabung kann dies zu erheblichen Problemen wie einem Stack-Overflow oder anderen unerwarteten Effekten führen und sollte wirklich nur dann eingesetzt werden, wenn man sich sicher ist, das Ganze auch im Griff zu haben.

Insbesondere sollte möglichst am ISR-Anfang die auslösende IRQ-Quelle deaktiviert und erst am Ende der ISR wieder aktiviert werden. Robuster als die Verwendung einer NOBLOCK-ISR ist daher folgender ISR-Aufbau:

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

ISR (XXX)
{
    // Implementiere die ISR ohne zunaechst weitere IRQs zuzulassen

    <<Deaktiviere die XXX-IRQ>>

    // Erlaube alle Interrupts (ausser XXX)
    sei();

    //... Code ...

    // IRQs global deaktivieren um die XXX-IRQ wieder gefahrlos
    // aktivieren zu koennen
    cli();

    <<Aktiviere die XXX-IRQ>>
}
```

Auf diese Weise kann sich die XXX-IRQ nicht selbst unterbrechen, was zu einer Art Endlosschleife führen würde.

Siehe auch: Hinweise in AVR-GCC

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

Datenaustausch mit Interrupt-Routinen

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

der Änderung woanders mitbekommen.

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

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

// Schwellwerte
// Entprellung:
#define CNTDEBOUNCE 10
// "lange 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
ISR(TIMER1_COMPA_vect)
{
   // hier wird gKeyCounter veraendert. Die übrigen
   // Programmteile müssen diese Aenderung "sehen":
   // volatile -> aktuellen Wert immer in den Speicher schreiben
   if ( !(KEY_PIN & (1<<KEY_PINNO)) ) {
      if (gKeyCounter < CNTREPEAT) gKeyCounter++;
   }
   else {
      gKeyCounter = 0;
   }
}

//...

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

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

sich so optimieren:

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

   tmp_kc = gKeyCounter; // Uebernahme in lokale Arbeitsvariable

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

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

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

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

```
ISR(TIMER1_COMPA_vect)
{
    86a:       1f 92           push    r1
    86c:       0f 92           push    r0
    86e:       0f b6           in      r0, 0x3f        ; 63
    870:       0f 92           push    r0
```

```
    872:       11 24           eor     r1, r1
    874:       8f 93           push    r24
  uint8_t tmp_kc;

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

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

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

volatile und Pointer

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

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

uint8_t *volatile a;   // a selbst ist volatile
```

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

Variablen größer 1 Byte

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

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

```
//...
}

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


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

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

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

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

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

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

// oder:

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

//...
```

- siehe auch Dokumentation der avr-libc zu atomic.h

Interrupt-Routinen und Registerzugriffe

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

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

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

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

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

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

```
...
        PORTA |= (1<<PA0);
  d2:   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 Unterbrechungen durch Interrupts. Die Anweisung zum Setzen von drei Bits wird jedoch in drei abhängige Assembler-Anweisungen übersetzt und bietet damit zwei "Angriffspunkte" für Unterbrechungen. 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-verknüft (c) und das Ergebnis 0b00011101 in PortA geschrieben (d). Obwohl zwischenzeitlich Bit 0 gelöscht wurde, ist es nach (d) wieder gesetzt.

Lösungsmöglichkeiten:

- Register ohne besondere Vorkehrungen nicht in Interruptroutinen *und* im Hauptprogramm verändern.
- Interrupts vor Veränderungen in Registern, die auch in ISRs verändert werden, deaktivieren ("cli").
- Bits einzeln löschen oder setzen. sbi und cbi können nicht unterbrochen werden. Vorsicht: nur Register im unteren Speicherbereich sind mittels sbi/cbi ansprechbar. Der Compiler kann nur für diese sbi/cbi-Anweisungen generieren. Für Register außerhalb dieses Adressbereichs ("Memory-Mapped"-Register) werden auch zur Manipulation einzelner Bits abhängige Anweisungen erzeugt (lds,...,sts).

- siehe auch: Dokumentation der avr-libc Frequently asked Questions/Fragen Nr. 1 und 8. (Stand: avr-

libc Vers. 1.0.4)

Interruptflags löschen

Beim Löschen von Interruptflags haben AVRs eine Besonderheit, die auch im Datenblatt beschrieben ist: Es wird zum Löschen eine 1 in das betreffende Bit geschrieben.

Hinweis: Dazu **nicht** übliche bitweise VerODERung nehmen, sondern eine direkte Zuweisung machen (Erklärung).

Was macht das Hauptprogramm?

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

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

- siehe auch: Dokumentation der avr-libc Abschnitt Modules/Interrupts and Signals

Ansteuerung eines LCD

Siehe: AVR-GCC-Tutorial/LCD-Ansteuerung

Timer

Siehe: AVR-GCC-Tutorial/Die Timer und Zähler des AVR

UART

Siehe: AVR-GCC-Tutorial/Der UART

Sleep-Modes

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

set_sleep_mode (uint8_t mode)
: Setzt den Schlafmodus, der bei Aufruf von sleep() aktiviert wird. In sleep.h sind einige Konstanten definiert (z. B. SLEEP_MODE_PWR_DOWN). Die definierten Modi werden jedoch nicht alle von sämtlichten

AVR-Controllern unterstützt.

sleep_enable()
: Aktiviert den gesetzten Schlafmodus, versetzt den Controller aber noch nicht in den Schlafmodus

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

sleep_disable()
: Deaktiviert den gesetzten Schlafmodus

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

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

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

int main(void)
{
...

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

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

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

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

Sleep Modi

Zu beachten ist, dass unterschiedliche Prozessoren aus der AVR Familie unterschiedliche Sleep-Modi unterstützen oder nicht unterstützen. Auskunft über die tatsächlichen Gegebenheiten gibt, wie immer, das zum Prozessor gehörende Datenblatt. Die unterschiedlichen Modi unterscheiden sich dadurch, welche

Bereiche des Prozessors abgeschaltet werden. Damit korrespondiert unmittelbar welche Möglichkeiten es gibt, den Prozessor aus den jeweiligen Sleep Modus wieder aufzuwecken.

Idle Mode (SLEEP_MODE_IDLE)
: Die CPU kann durch SPI, USART, Analog Comperator, ADC, TWI, Timer, Watchdog und irgendeinen anderen Interrupt wieder aufgeweckt werden.

ADC Noise Reduction Mode (SLEEP_MODE_ADC)
: In diesem Modus liegt das Hauptaugenmerk darauf, die CPU soweit stillzulegen, dass der ADC möglichst keine Störungen aus dem inneren der CPU auffangen kann. Aufwachen aus diesem Modus kann ausgelöst werden durch den ADC, externe Interrupts, TWI, Timer und Watchdog.

Power-Down Mode (SLEEP_MODE_PWR_DOWN)
: In diesem Modus wird ein externer Oszillator (Quarz, Quarzoszillator) gestoppt. Geweckt werden kann die CPU durch einen externen Level Interrupt, TWI, Watchdog, Brown-Out-Reset

Power-Save-Mode (SLEEP_MODE_PWR_SAVE)
: Power-Save ist identisch zu Power-Down mit einer Ausnahme: Ist der Timer 2 auf die Verwendung eines externen Taktes konfiguriert, so läuft dieser Timer auch im Power-Save weiter und kann die CPU mit einem Interrupt aufwecken.

Standby-Mode (SLEEP_MODE_STANDBY, SLEEP_MODE_EXT_STANDBY)
: Voraussetzung für den Standby-Modus ist die Verwendung eines Quarzes oder eines Quarzoszillators (also einer externen Taktquelle). Ansonsten ist dieser Modus identisch zum Power-Down Modus. Vorteil dieses Modus ist eine kürzere Aufwachzeit.

Siehe auch:

- Dokumentation der avr-libc Abschnitt Modules/Power Management and Sleep-Modes
- Forenbeitrag zur "Nichtverwendung" von sleep_mode in ISRs.

Zeiger

Zeiger (engl. /pointer/), d.h. Variablen, die die Adresse von Daten oder Funktionen enthalten, belegen 16 bits. Das hängt mit dem addressierbaren Speicherbereich zusammen, und gcc reserviert dann den entsprechenden Platz. Ggf. ist es also günstiger, Indizes auf Arrays (Listen) zu verwenden, so dass der GCC für die Zeigerarithmetik den erforderlichen RAM nur temporär benötigt.

Siehe auch: Zeiger

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

Siehe auch:

- Binäre Daten zum Programm hinzufügen

RAM

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

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

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

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

Beispiel zur Verwendung von malloc():

```
#include <stdlib.h>

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

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

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

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

siehe auch:

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

Programmspeicher (Flash)

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

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

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

//...

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

/* Wort */
const uint16_t pgmFooWort PROGMEM = 12345;

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

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

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

 // ...

}
```

Byte lesen

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

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


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

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

```
    //...

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

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

Wort lesen

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

```
const uint16_t pgmFooWort PROGMEM = 12345;

    uint16_t myWord;

    myWord = pgm_read_word(&pgmFooWort);
```

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

```
    uint8_t *ptrToArray;

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

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

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

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

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

Strings lesen

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

```
```

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

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

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

  ...

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

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

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

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

int main()
{
  char string[40];

  ...

  strcpy_P( string, pgmString );
}
```

Float lesen

Auch um floats zu lesen gibt es ein Makro in avr/pgmspace.h. Ein Beispiel:

```
/* Beispiel float aus Flash */

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

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 für structs und pointer aus Flash auf struct im Flash (menues, state-machines etc.). Eine kleine Einleitung insbesondere auch in Bezug auf die auftretenden Schwierigkeiten liefert [3].

Array aus Strings im Flash-Speicher

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

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

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

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

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

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

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

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


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

        while (1) { ; }
}
```

Siehe dazu auch die avr-libc FAQ: "How do I put an array of strings completely in ROM?" (http://www.nongnu.org/avr-libc/user-manual/FAQ.html#faq_rom_array)

Vereinfachung für Zeichenketten (Strings) im Flash

Zeichenketten können innerhalb des Quellcodes als "Flash-Konstanten" ausgewiesen werden. Dazu dient das

Makro PSTR aus pgmspace.h. Dies erspart die getrennte Deklaration mit PROGMEM-Attribut.

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

#define MAXLEN 30

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

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

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

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

Aber Vorsicht: Ersetzt man zum Beispiel

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

durch

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

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

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

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

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

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

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

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

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

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

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

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

Flash in der Anwendung schreiben

Bei AVRs mit "self-programming"-Option (auch bekannt als Bootloader-Support) können Teile des Flash-Speichers auch vom Anwendungsprogramm selbst beschrieben werden. Dies ist nur möglich, wenn die Schreibfunktionen in einem besonderen Speicherbereich (boot-section) des Programmspeichers/Flash abgelegt sind. Bei wenigen "kleinen" AVRs 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.

Siehe auch:

- Forumsbeitrag Daten in Programmspeicher speichern

Warum so kompliziert?

Zu dem Thema, warum die Verabeitung von Werten aus dem Flash-Speicher so "kompliziert" ist, sei hier nur kurz erläutert: Die Harvard-Architektur des AVR weist getrennte Adressräume für Programm(Flash)- und Datenspeicher(RAM) auf. Der C-Standard und der gcc-Compiler sehen keine unterschiedlichen Adressräume vor. Hat man zum Beispiel eine Funktion string_an_uart(const char* s) und übergibt an diese Funktion die Adresse einer Zeichenkette (einen Pointer, z. B. 0x01fe), "weiß" die Funktion nicht, ob die Adresse auf den

Flash-Speicher oder den/das RAM zeigt. Allein aus dem Pointer-Wert (der Zahl) kann nicht geschlossen werden, ob ein "einfaches" zeichen_an_uart(s[i]) oder zeichen_an_uart(pgm_read_byte(&s[i]) genutzt werden muss, um das i-te Zeichen auszugeben.

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

- siehe auch: Dokumentation der avr-libc 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 bestimme Befehlsabfolgen vorgegeben sind, die innerhalb weniger Taktzyklen aufeinanderfolgen müssen ("timed sequence"). Auch dies muss bei Nutzung der Funktionen aus der avr-libc/eeprom.h-Datei nicht selbst implementiert werden. Innerhalb der Funktionen werden Interrupts vor der "EEPROM-Sequenz" global deaktiviert und im Anschluss, falls vorher auch schon eingeschaltet, wieder aktiviert.

Bei der Deklaration einer Variable im EEPROM, ist das Attribut für die Section ".eeprom" zu ergänzen. Diese Art der Deklaration funktioniert übrigens nur global, also z.B. nicht innerhalb der Main-Funktion. Siehe dazu folgendes Beispiel:

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

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

//...

/* Byte */
uint8_t eeFooByte EEMEM = 123;

/* Wort */
uint16_t eeFooWord EEMEM = 12345;

/* float */
float eeFooFloat EEMEM;
```

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

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

Bytes lesen/schreiben

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

```
//...
    uint8_t myByte;

    myByte = eeprom_read_byte(&eeFooByte); // lesen
    // myByte hat nun den Wert 123
//...
    myByte = 99;
    eeprom_write_byte(&eeFooByte, myByte); // schreiben
    // der Wert 99 wird im EEPROM an die Adresse der
    // 'Variablen' eeFooByte geschrieben
//...
    myByte = eeprom_read_byte(&eeFooByteArray1[1]);
    // myByte hat nun den Wert 3
//...

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

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

Wort lesen/schreiben

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

```
//...
    uint16_t myWord;

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

Block lesen/schreiben

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

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

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

    /* Datenblock aus EEPROM LESEN  */

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

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

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

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

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

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

```
//...
   float myFloat = 12.34;

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

   u.r = myFloat;

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

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

//...
```

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

```
//...
```

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

#define MAXSENSORS 3
tMyStruct eeMyStruct[MAXSENSORS] EEMEM;

//...

void egal(void)
{
   tMyStruct work;

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

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

EEPROM-Speicherabbild in .eep-Datei

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

EEPROM-Variable auf feste Adressen legen

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

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

Im folgenden nun zwei Sourcelistings mit einem Beispiel:

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

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

#define   EESIZE   512      // Maximale Größe des EEPROMS

#define   EE_DUMMY   0x000  // Dummyelement (Adresse 0 sollte nicht genutzt werden)
#define   EE_VALUE1  0x001  // Eine Bytevariable
#define   EE_WORD1L  0x002  // Eine Wordvariable (Lowbyte)
#define   EE_WORD1H  0x003  // Eine Wordvariable (Highbyte)
#define   EE_VALUE2  0x004  // Eine weitere Bytevariable
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
uint8_t   temp1;
uint16_t  temp2;

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

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

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

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

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

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

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

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

Kleine Schlussbemerkung:

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

Direkter Zugriff auf EEPROM-Adressen

Will man direkt auf bestimmte EEPROM Adressen zugreifen, dann sind folgende Funktionen hilfreich, um sich die Typecasts zu ersparen:

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

// Byte aus dem EEPROM lesen
uint8_t EEPReadByte(uint16_t addr)
{
  return eeprom_read_byte((uint8_t *)addr);
}

// Byte in das EEPROM schreiben
void EEPWriteByte(uint16_t addr, uint8_t val)
{
  eeprom_write_byte((uint8_t *)addr, val);
}
```

oder als Makro:

```
#define   EEPReadByte(addr)         eeprom_read_byte((uint8_t *)addr)
#define   EEPWriteByte(addr, val)   eeprom_write_byte((uint8_t *)addr, val)
```

Verwendung:

```
EEPWriteByte(0x20, 128);   // Byte an die Adresse 0x20 schreiben
...
Val=EEPReadByte(0x20);     // EEPROM-Wert von Adresse 0x20 lesen
```

Bekannte Probleme bei den EEPROM-Funktionen

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

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

- siehe auch: Dokumentation der avr-libc Abschnitt Modules/EEPROM handling

EEPROM Register

Um das EEPROM anzusteuern, sind drei Register von Bedeutung:

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

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

EECR
: Ist das Kontrollregister für das EEPROM

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

Aufbau des EECR-Registers

Bit	7	6	5	4	3	2	1	0
Name	-	-	-	-	EERIE	EEMWE	EEWE	EERE
Read/Write	R	R	R	R	R/W	R/W	R/W	R/W
Init Value	0	0	0	0	0	0	0	0

Bedeutung der Bits

Bit 4-7
: nicht belegt

Bit 3 (EERIE)
: *EEPROM Ready Interrupt Enable*: Wenn das Bit gesetzt ist und globale Interrupts erlaubt sind in Register SREG (Bit 7), wird ein Interrupt ausgelöst nach Beendigung des Schreibzyklus (EEPROM Ready Interrupt). Ist einer der beiden Bits 0, wird kein Interrupt ausgelöst.

Bit 2 EEMWE)
: *EEPROM Master Write Enable*: Dieses Bit bestimmt, dass, wenn EEWE = 1 gesetzt wird (innerhalb von 4 Taktzyklen), das EEPROM beschrieben wird mit den Daten in EEDR bei Adresse EEAR. Wenn EEMWE = 0 ist und EEWE = 1 gesetzt wird, hat das keine Auswirkungen. Der Schreibvorgang wird dann nicht ausgelöst. Nach 4 Taktzyklen wird das Bit EEMWE automatisch wieder auf 0 gesetzt. Dieses Bit löst den Schreibvorgang nicht aus, es dient sozusagen als Sicherungsbit für EEWE.

Bit 1 (EEWE)
: *EEPROM Write Enable*: Dieses Bit löst den Schreibvorgang aus, wenn es auf 1 gesetzt wird, sofern vorher EEMWE gesetzt wurde und seitdem nicht mehr als 4 Taktzyklen vergangen sind. Wenn der Schreibvorgang abgeschlossen ist, wird dieses Bit automatisch wieder auf 0 gesetzt und, sofern EERIE gesetzt ist, ein Interrupt ausgelöst. Ein Schreibvorgang sieht typischerweise wie folgt aus:

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

Bit 0 EERE
: *EEPROM Read Enable*: Wird dieses Bit auf 1 gesetzt wird das EEPROM an der Adresse in EEAR ausgelesen und die Daten in EEDR gespeichert. Das EEPROM kann nicht ausgelesen werden, wenn

bereits eine Schreiboperation gestartet wurde. Es ist daher zu empfehlen, die Bereitschaft vorher zu prüfen. Das EEPROM ist lesebereit, wenn das Bit EEWE=0 ist. Ist der Lesevorgang abgeschlossen, wird das Bit wieder auf 0 gesetzt, und das EEPROM ist für neue Lese- und Schreibbefehle wieder bereit. Ein typischer Lesevorgang kann wie folgt aufgebaut sein:

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

Die Nutzung von sprintf und printf

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

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

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

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

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

uint16_t counter;

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

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

int main()
{
  counter = 5;

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

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

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

void uart_init(void);

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

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

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

    loop_until_bit_is_set( UCSRA, UDRE );
    UDR = c;
    return 0;
}

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

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

#ifndef F_CPU
#define F_CPU 3686400
#endif
#define UART_BAUD_RATE 9600

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

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

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

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

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

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

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

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

Assembler und Inline-Assembler

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

Die "gnu-Toolchain" bietet dazu zwei Möglichkeiten:

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

Inline-Assembler

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

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

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

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

```
   ...
   uint16_t i;

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

   ...

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

   ...

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

```
    ;
```

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

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

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

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

Das Resultat zeigt ein Blick in die Assembler-Datei, die der Compiler mit der option `-save-temps` nicht löscht:

```
        cp  r24, __zero_reg__    ;  count
        cpc r25, __zero_reg__    ;  count
        breq 2f
        1:
        sbiw r24,1               ;  count
        brne 1b
        2:
```

Detaillierte Ausführungen zum Thema Inline-Assembler finden sich in der Dokumentation der avr-libc im Abschnitt Related Pages/Inline Asm.

Siehe auch:

- AVR Assembler-Anweisungsliste
- Deutsche Einführung in Inline-Assembler

Assembler-Dateien

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

Wenn man mit dem AVR Studio arbeitet werden ".S"-Dateien im "Source Files"-Projektordner automatisch

mit übersetzt und gelinkt (ohne Umweg über externes Makefile).

Möchte man den Umweg über externes Makefile gehen, muss alternativ auch das standardmäßig erstellte Makefile bearbeitet und folgende Zeilen eingefügt werden:

```
## Objects that must be built in order to link
OBJECTS = (alte Dateien...) useful.o

## Compile
## Hier folgt eine Liste der gelinkten Dateien, darunter einfügen:
useful.o: ../useful.S
	$(CC) $(INCLUDES) $(ASMFLAGS) -c  $<
```

Das war es schon. Allerdings gilt es zu beachten, dass das makefile über "Project -> Configuration options" selbst einzubinden ist, sonst wird es natürlich wieder überschrieben.

Im Beispiel eine Funktion *superFunc*, die alle Pins des Ports D auf "Ausgang" schaltet, eine Funktion *ultraFunc*, die die Ausgänge entsprechend des übergebenen Parameters schaltet, eine Funktion *gigaFunc*, die den Status von Port A zurückgibt und eine Funktion *addFunc*, die zwei Bytes zu einem 16-bit-Wort addiert. Die Zuweisungen im C-Code (PORTx = ...) verhindern, dass der Compiler die Aufrufe wegoptimiert und dienen nur zur Veranschaulichung der Parameterübergaben.

Zuerst der Assembler-Code. Der Dateiname sei useful.S:

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

//; Arbeitsregister (ohne "r")
workreg  = 16
workreg2 = 17

//; Konstante:
ALLOUT = 0xff

//; ** Setze alle Pins von PortD auf Ausgang **
//; keine Parameter, keine Rückgabe
.global superFunc
.func superFunc
superFunc:
   push workreg
   ldi workreg, ALLOUT
   out  _SFR_IO_ADDR(DDRD), workreg  // beachte: _SFR_IO_ADDR()
   pop workreg
   ret
.endfunc


//; ** Setze PORTD auf übergebenen Wert **
//; Parameter in r24 (LSB immer bei "graden" Nummern)
.global ultraFunc
.func ultraFunc
ultraFunc:
   out  _SFR_IO_ADDR(PORTD), 24
   ret
.endfunc


//; ** Zustand von PINA zurückgeben **
//; Rückgabewerte in r24:r25 (LSB:MSB), hier nur LSB genutzt
.global gigaFunc
.func gigaFunc
gigaFunc:
   in 24, _SFR_IO_ADDR(PINA)
   ret
.endfunc
```

```
//; ** Zwei Bytes addieren und 16-bit-Wort zurückgeben **
//; Parameter in r24 (Summand1) und r22 (Summand2) -
//;  Parameter sind Word-"aligned" d.h. LSB immer auf "graden"
//;  Registernummern. Bei 8-Bit und 16-Bit Paramtern somit
//;  beginnend bei r24 dann r22 dann r20 etc.
//; Rückgabewert in r24:r25
.global addFunc
.func addFunc
addFunc:
   push workreg
   push workreg2
   clr workreg2
   mov workreg, 22
   add workreg, 24
   adc workreg2, 1    // r1 - assumed to be always zero ...
   movw r24, workreg
   pop workreg2
   pop workreg
   ret
.endfunc

//; oh je - sorry - Mein AVR-Assembler ist eingerostet, hoffe das stimmt so...

.end
```

Im Makefile ist der Name der Assembler-Quellcodedatei einzutragen:

```
ASRC = useful.S
```

Der Aufruf erfolgt dann im C-Code so:

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

extern void superFunc(void);
extern void ultraFunc(uint8_t setVal);
extern uint8_t gigaFunc(void);
extern uint16_t addFunc(uint8_t w1, uint8_t w2);

int main(void)
{
[...]
  superFunc();

  ultraFunc(0x55);

  PORTD = gigaFunc();

  PORTA = (addFunc(0xF0, 0x11) & 0xff);
  PORTB = (addFunc(0xF0, 0x11) >> 8);
[...]
}
```

Das Ergebnis wird wieder in der lss-Datei ersichtlich:

```
[...]
   superFunc();
 148:   0e 94 f6 00     call    0x1ec

  ultraFunc(0x55);
```

```
 14c:   85 e5           ldi     r24, 0x55       ; 85
 14e:   0e 94 fb 00     call    0x1f6

  PORTD = gigaFunc();
 152:   0e 94 fd 00     call    0x1fa
 156:   82 bb           out     0x12, r24       ; 18

  PORTA = (addFunc(0xF0, 0x11) & 0xff);
 158:   61 e1           ldi     r22, 0x11       ; 17
 15a:   80 ef           ldi     r24, 0xF0       ; 240
 15c:   0e 94 ff 00     call    0x1fe
 160:   8b bb           out     0x1b, r24       ; 27
  PORTB = (addFunc(0xF0, 0x11) >> 8);
 162:   61 e1           ldi     r22, 0x11       ; 17
 164:   80 ef           ldi     r24, 0xF0       ; 240
 166:   0e 94 fc 00     call    0x1f8
 16a:   89 2f           mov     r24, r25
 16c:   99 27           eor     r25, r25
 16e:   88 bb           out     0x18, r24       ; 24

[...]
000001ec <superFunc>:
// setze alle Pins von PortD auf Ausgang
.global superFunc
.func superFunc
superFunc:
   push workreg
 1ec:   0f 93           push    r16
   ldi workreg, ALLOUT
 1ee:   0f ef           ldi     r16, 0xFF       ; 255
   out  _SFR_IO_ADDR(DDRD), workreg
 1f0:   01 bb           out     0x11, r16       ; 17
   pop workreg
 1f2:   0f 91           pop     r16
   ret
 1f4:   08 95           ret

000001f6 <ultraFunc>:
.endfunc


// setze PORTD auf übergebenen Wert
.global ultraFunc
.func ultraFunc
ultraFunc:
   out  _SFR_IO_ADDR(PORTD), 24
 1f6:   82 bb           out     0x12, r24       ; 18
   ret
 1f8:   08 95           ret

000001fa <gigaFunc>:
.endfunc


// Zustand von PINA zurückgeben
.global gigaFunc
.func gigaFunc
gigaFunc:
   in 24, _SFR_IO_ADDR(PINA)
 1fa:   89 b3           in      r24, 0x19       ; 25
   ret
 1fc:   08 95           ret

000001fe <addFunc>:
.endfunc


// zwei Bytes addieren und 16-bit-Wort zurückgeben
.global addFunc
.func addFunc
```

```
addFunc:
   push workreg
 1fe:   0f 93           push    r16
   push workreg2
 200:   1f 93           push    r17
   clr workreg2
 202:   11 27           eor     r17, r17
   mov workreg, 22
 204:   06 2f           mov     r16, r22
   add workreg, 24
 206:   08 0f           add     r16, r24
   adc workreg2, 1    // r1 - assumed to be always zero ...
 208:   11 1d           adc     r17, r1
   movw r24, workreg
 20a:   c8 01           movw    r24, r16
   pop workreg2
 20c:   1f 91           pop     r17
   pop workreg
 20e:   0f 91           pop     r16
   ret
 210:   08 95           ret

[...]
```

Die Zuweisung von Registern zu Parameternummer und die Register für die Rückgabewerte sind in den "Register Usage Guidelines" der avr-libc-Dokumentation erläutert.

Siehe auch:

- avr-libc-Dokumentation: Related Pages/avr-libc and assembler programs
- avr-libc-Dokumentation: Related Pages/FAQ/"What registers are used by the C compiler?"

Globale Variablen für Datenaustausch

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

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

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

volatile uint8_t zaehler;

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

}
```

Im folgenden Assembler-Beispiel wird der Externe Interrupt0 verwendet, um den Zähler hochzuzählen. Es fehlen die Initialisierungen des Interrupts und die Interrupt-Freigabe, so richtig sinnvoll ist der Code auch nicht, aber er zeigt (hoffentlich) wie es geht.

Im Umgang mit Interrupt-Vektoren gilt beim GCC-Assembler das Gleiche, wie bei C: Man muss die exakte Schreibweise beachten, ansonsten wird nicht der Interrupt-Vektor angelegt, sondern eine neue Funktion - und man wundert sich, dass nichts funktionert (vgl. das AVR-GCC-Handbuch).

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

```
temp = 16

.extern zaehler

.global INT0_vect
INT0_vect:

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

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

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

.end
```

Globale Variablen im Assemblerfile anlegen

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

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

zaehl_asm.S

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

temp = 16

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

.global INT0_vect
INT0_vect:

...
```

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

zaehl_asm.h

```
#ifndef ZAEHL_ASM_H
#define ZAEHL_ASM_H

extern volatile uint8_t zaehler;

#endif
```

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

Variablen größer als 1 Byte

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

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

Die dazugehörige Deklaration im Headerfile wäre dann:

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

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

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

temp = 16

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

.global INT0_vect
INT0_vect:

    push temp                       // 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

Besonderheiten bei der Anpassung bestehenden Quellcodes

Einige Funktionen aus früheren Versionen der avr-libc werden inzwischen als veraltet angesehen. Sie sind nicht mehr vorhanden oder als *deprecated* (missbilligt) ausgewiesen und Definitionen in <compat/deprecated.h> verschoben. Es empfiehlt sich, vorhandenen Code zu portieren und die alten Funktionen nicht mehr zu nutzen, auch wenn diese noch zur Verfügung stehen.

Veraltete Funktionen zur Deklaration von Interrupt-Routinen

Die Funktionen (eigentlich Makros) *SIGNAL* und *INTERRUPT* zur Deklaration von Interruptroutinen sollten nicht mehr genutzt werden.

In aktuellen Versionen der avr-libc (z. B. avr-libc 1.4.3 aus WinAVR 20060125) werden Interruptroutinen, die **nicht** durch andere Interrupts **unterbrechbar** sind, mit ISR deklariert (siehe Abschnitt im Hauptteil). Auch die Benennung wurden vereinheitlicht und an die üblichen Bezeichnungen in den AVR Datenblättern angepasst. In der Dokumentation der avr-libc sind alte und neue Bezeichnungen in der Tabelle gegenübergestellt. Die erforderlichen Schritte zur Portierung:

- #include von avr/signal.h entfernen
- SIGNAL durch ISR ersetzen
- Name des Interrupt-Vektors anpassen (SIG_* durch entsprechendes *_vect)

Als Beispiel für die Anpassung zuerst ein "alter" Code:

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

...

/* Timer2 Output Compare bei einem ATmega8 */
SIGNAL(SIG_OUTPUT_COMPARE2)
{
   ...
}
```

Im Datenblatt wird der Vektor mit TIMER2 COMP bezeichnet. Die Bezeichnung in der avr-libc entspricht dem Namen im Datenblatt, Leerzeichen werden durch Unterstriche (_) ersetzt und ein _vect angehängt.

Der neue Code sieht dann so aus:

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

```
ISR(TIMER2_COMP_vect)
{
   ...
}
```

Bei Unklarheiten bezüglich der neuen Vektorlabels hilft (noch) ein Blick in die Headerdatei des entsprechenden Controllers. Für das vorherige Beispiel also der Blick in die Datei iom8.h für den ATmega8, dort findet man die veraltete Bezeichnung unterhalb der aktuellen.

```
...
/* $Id: iom8.h,v 1.13 2005/10/30 22:11:23 joerg_wunsch Exp $ */
/* avr/iom8.h - definitions for ATmega8 */

...

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

Für **unterbrechbare** Interruptroutinen, die mittels *INTERRUPT* deklariert sind, gibt es keinen direkten Ersatz in Form eines Makros. Solche Routinen sind laut Dokumentation der avr-libc in folgender Form zu deklarieren:

```
void XXX_vect(void) __attribute__((interrupt));
void XXX_vect(void) {
  ...
}
```

Beispiel:

```
/* ** alt ** */
#include <avr/io.h>
#include <avr/interrupt.h>

//...

INTERRUPT(SIG_OVERFLOW0)
{
   ...
}

/* ** neu: ** */
#include <avr/io.h>

//...

void TIMER0_OVF_vect(void) __attribute__((interrupt));
void TIMER0_OVF_vect(void)
{
   ...
}
```

Will oder kann man den Code nicht portieren, ist zur weiteren Verwendung von *INTERRUPT* die Header-Datei *compat/deprecated.h* einzubinden. Man sollte bei dieser Gelegenheit jedoch nochmals überprüfen, ob die Funktionalität von *INTERRUPT* tatsächlich gewollt ist. In vielen Fällen wurde *INTERRUPT* dort genutzt, wo

eigentlich *SIGNAL* (nunmehr *ISR*) hätte genutzt werden sollen.

Veraltete Funktionen zum Portzugriff

inp und *outp* zum Einlesen bzw. Schreiben von Registern sind nicht mehr erforderlich, der Compiler unterstützt dies ohne diesen Umweg.

```
unsigned char i, j;

// alt:
  i = inp(PINA);
  j = 0xff;
  outp(PORTB, j);

// neu (nicht mehr wirklich neu...):
  i = PINA
  j = 0xff;
  PORTB = j;
```

Will oder kann man den Code nicht portieren, ist zur weiteren Verwendung von inp und outp die Header-Datei **compat/deprecated.h** einzubinden.

Veraltete Funktionen zum Zugriff auf Bits in Registern

cbi und *sbi* zum Löschen und Setzen von Bits sind nicht mehr erforderlich, der Compiler unterstützt dies ohne diesen Umweg. Die Bezeichnung ist ohnehin irreführend da die Funktionen nur für Register mit Adressen im unteren Speicherbereich tatsächlich in die Assembleranweisungen cbi und sbi übersetzt werden.

```
// alt:
  sbi(PORTB, PB2);
  cbi(PORTC, PC1);

// neu (auch nicht mehr wirklich neu...):
  PORTB |=  (1<<PB2);
  PORTC &= ~(1<<PC1);
```

Will oder kann man den Code nicht portieren, ist zur weiteren Verwendung von sbi und cbi die Header-Datei **compat/deprecated.h** einzubinden. Wer unbedingt will, kann sich natürlich eigene Makros mit aussagekräftigeren Namen definieren. Zum Beispiel:

```
#define SET_BIT(PORT, BITNUM)    ((PORT) |=  (1<<(BITNUM)))
#define CLEAR_BIT(PORT, BITNUM)  ((PORT) &= ~(1<<(BITNUM)))
#define TOGGLE_BIT(PORT, BITNUM) ((PORT) ^=  (1<<(BITNUM)))
```

Selbstdefinierte (nicht-standardisierte) ganzzahlige Datentypen

Bei den im Folgenden genannten Typdefinitionen ist zu beachten, dass die Bezeichnungen für "Worte" teilweise je nach Prozessorplattform unterschiedlich verwendet werden. Die angegebenen Definitionen beziehen sich auf die im Zusammenhang mit AVR/8-bit-Controllern üblichen "Bit-Breiten" (In Erläuterungen zum ARM7TDMI z. B. werden oft 32-bit Integer mit "Wort" ohne weitere Ergänzung bezeichnet). Es empfiehlt sich, bei der Überarbeitung von altem Code die im Abschnitt *standardisierten ganzzahligen Datentypen* beschriebenen Datentypen zu nutzen (stdint.h) und damit "Missverständnissen" vorzubeugen, die z. B. bei der Portierung von C-Code zwischen verschiedenen Plattformen auftreten können.

```
typedef unsigned char      BYTE;       // besser: uint8_t  aus <stdint.h>
typedef unsigned short     WORD;       // besser: uint16_t aus <stdint.h>
typedef unsigned long      DWORD;      // besser: uint32_t aus <stdint.h>
typedef unsigned long long QWORD;      // besser: uint64_t aus <stdint.h>
```

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

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

DWORD
: Der Datentyp DWORD (gesprochen: Double-Word) definiert eine Variable mit 32 Bit Breite zur Darstellung von ganzen Zahlen im Bereich zwischen 0 ... 4294967295.

QWORD
: Der Datentyp QWORD (gesprochen: Quad-Word) definiert eine Variable mit 64 Bit Breite zur Darstellung von ganzen Zahlen im Bereich zwischen 0 ... 18446744073709551615.

Zusätzliche Funktionen im Makefile

Bibliotheken (Libraries/.a-Dateien) hinzufügen

Um Funktionen aus Bibliotheken ("echte" Libraries, *.a-Dateien) zu nutzen, sind dem Linker die Namen der Bibliotheken als Parameter zu übergeben. Dazu ist die Option -l (kleines L) vorgesehen, an die der Name der Library angehängt wird.

Dabei ist zu beachten, dass der Name der Library und der Dateiname der Library nicht identisch sind. Der hinter -l angegebene Name entspricht dem Dateinamen der Library ohne die Zeichenfolge *lib* am Anfang des Dateinamens und ohne die Endung *.a*. Sollen z. B. Funktionen aus einer Library mit dem Dateinamen *libefsl.a* eingebunden (gelinkt) werden, lautet der entsprechende Parameter -lefsl (vergl. auch -lm zum Anbinden von libm.a).

In Makefiles wird traditonell eine make-Variable LDLIBS genutzt, in die "l-Parameter" abgelegt werden. Die WinAVR-makefile-Vorlage enthält diese Variable zwar nicht, dies stellt jedoch keine Einschränkung dar, da alle in der make-Variable LDFLAGS abgelegten Parameter an den Linker weitergereicht werden.

Beispiele:

```
# Einbinden von Funktionen aus einer Library efsl (Dateiname libefsl.a)
LDFLAGS += -lefsl
# Einbinden von Funktionen aus einer Library xyz (Dateiname libxyz.a)
LDFLAGS += -lxyz
```

Liegen die Library-Dateien nicht im Standard Library-Suchpfad, sind die Pfade mittels Parameter *-L* ebenfalls anzugeben. (Der vordefinierte Suchpfad kann mittels *avr-gcc --print-search-dirs* angezeigt werden.)

Als Beispiel ein Projekt ("superapp2"), in dem der Quellcode von zwei Libraries (efsl und xyz) und der Quellcode der eigentlichen Anwendung in verschiedenen Verzeichnissen mit der folgenden "Baumstruktur" abgelegt sind:

```
superapp2
|
+----- efslsource (darin libefsl.a)
|
+----- xyzsource (darin libxyz.a)
|
+----- firmware (darin Anwendungs-Quellcode und Makefile)
```

Daraus folgt, dass im Makefile die Verzeichnis efslsource und xyzsource in den Library-Suchpfad aufzunehmen sind:

```
LDFLAGS += -L../efslsource/ -L../xyzsource/
```

Fuse-Bits

Zur Berechnung der Fuse-Bits bietet sich neben dem Studium des Datenblattes auch der AVR Fuse Calculator an. Gewarnt werden muss vor der Benutzung von PonyProg, weil dort durch die negierte Darstellung gern Fehler gemacht werden.

Soll die Programmierung von Fuse- und Lockbits automatisiert werden, kann man dies ebenfalls durch Einträge im Makefile vornehmen, die beim Aufruf von "make program" an die genutzte Programmiersoftware übergeben werden. In der makefile-Vorlage von WinAVR (und mfile) gibt es dafuer jedoch keine "Ausfüllhilfe" (Stand 9/2006). Die folgenden Ausführungen gelten für die Programmiersoftware AVRDUDE (Standard in der WinAVR-Vorlage), können jedoch sinngemäß auf andere Programmiersoftware übertragen werden, die die Angabe der Fuse- und Lockbits-Einstellungen per Kommandozeilenparameter unterstützt (z. B. stk500.exe). Im einfachsten Fall ergänzt man im Makefile einige Variablen, deren Werte natürlich vom verwendeten Controller und den gewünschten Einstellungen abhängen (vgl. Datenblatt Fuse-/Lockbits):

```
#---------------- Programming Options (avrdude) ----------------

#...
#Beispiel! f. ATmega16 - nicht einfach uebernehmen! Zahlenwerte anhand
#--------- des Datenblatts nachvollziehen und gegebenenfalls aendern.
#
AVRDUDE_WRITE_LFUSE = -U lfuse:w:0xff:m
AVRDUDE_WRITE_HFUSE = -U hfuse:w:0xd8:m
AVRDUDE_WRITE_LOCK  = -U lock:w:0x2f:m
#...
```

Damit diese Variablen auch genutzt werden, ist der Aufruf von avrdude im Makefile entsprechend zu ergänzen:

```
# Program the device.
program: $(TARGET).hex $(TARGET).eep
# ohne Fuse-/Lock-Einstellungen (nach WinAVR Vorlage Stand 4/2006)
#       $(AVRDUDE) $(AVRDUDE_FLAGS) $(AVRDUDE_WRITE_FLASH) \
#        $(AVRDUDE_WRITE_EEPROM)
# mit Fuse-/Lock-Einstellungen
        $(AVRDUDE) $(AVRDUDE_FLAGS) $(AVRDUDE_WRITE_LFUSE) \
        $(AVRDUDE_WRITE_HFUSE) $(AVRDUDE_WRITE_FLASH) \
        $(AVRDUDE_WRITE_EEPROM) $(AVRDUDE_WRITE_LOCK)
```

Eine weitere Möglichkeit besteht darin, die Fuse- und Lockbit-Einstellungen vom Preprozessor/Compiler generieren zu lassen. Die Fuse-Bits werden dann bei Verwendung von AVRDUDE in eigene Hex-Files geschrieben. Hierzu kann man z. B. folgendes Konstrukt verwenden:

In eine der C-Sourcen wird eine Variable je Fuse-Byte vom Typ *unsigned char* deklariert und in eine extra Section gepackt. Dies kann entweder in einem vorhandenen File passieren oder in ein neues (z. B. fuses.c) geschrieben werden. Das File muss im Makefile aber auf jeden Fall mit kompiliert und gelinkt werden.

```
// tiny 2313 fuses low byte
#define CKDIV8  7
#define CKOUT   6
#define SUT1    5
#define SUT0    4
#define CKSEL3  3
#define CKSEL2  2
#define CKSEL1  1
#define CKSEL0  0

// tiny2313 fuses high byte
#define DWEN       7
#define EESAVE     6
#define SPIEN      5
#define WDTON      4
#define BODLEVEL2  3
#define BODLEVEL1  2
#define BODLEVEL0  1
#define RSTDISBL   0

// tiny2313 fuses extended byte
#define SELFPRGEN  0

#define LFUSE         __attribute__ ((section ("lfuses")))
#define HFUSE         __attribute__ ((section ("hfuses")))
#define EFUSE         __attribute__ ((section ("efuses")))


// select ext crystal 3-8Mhz
unsigned char lfuse LFUSE =
    ( (1<<CKDIV8) | (1<<CKOUT) | (1<<CKSEL3) | (1<<CKSEL2) |
      (0<<CKSEL1) | (1<<CKSEL0) | (0<<SUT1) | (1<<SUT0) );
unsigned char hfuse HFUSE =
    ( (1<<DWEN) | (1<<EESAVE) | (0<<SPIEN) | (1<<WDTON) |
      (1<<BODLEVEL2) | (1<<BODLEVEL1) | (0<<BODLEVEL0) | (1<<RSTDISBL) );
unsigned char efuse EFUSE =
    ((0<<SELFPRGEN));
```

ACHTUNG: Die Bitpositionen wurden nicht vollständig getestet!

Eine "1" bedeutet hier, dass das Fuse-Bit *nicht* programmiert wird - die Funktion also i.A. nicht aktiviert ist. Eine "0" hingegen aktiviert die meisten Funktionen. Dies ist wie im Datenblatt (1 = unprogrammed, 0 = programmed).

Das Makefile muss nun noch um folgende Targets erweitert werden (mit Tabulator einrücken - nicht mit Leerzeichen):

```
lfuses: build
        -$(OBJCOPY) -j lfuses --change-section-address lfuses=0 \
          -O ihex $(TARGET).elf $(TARGET)-lfuse.hex
        @if [ -f $(TARGET)-lfuse.hex ]; then \
         $(AVRDUDE) $(AVRDUDE_FLAGS) -U lfuse:w:$(TARGET)-lfuse.hex; \
        fi;

hfuses: build
        -$(OBJCOPY) -j hfuses --change-section-address hfuses=0 \
          -O ihex $(TARGET).elf $(TARGET)-hfuse.hex
        @if [ -f $(TARGET)-hfuse.hex ]; then \
         $(AVRDUDE) $(AVRDUDE_FLAGS) -U hfuse:w:$(TARGET)-hfuse.hex; \
        fi;
```

```
efuses: build
        -$(OBJCOPY) -j efuses --change-section-address efuses=0 \
         -O ihex $(TARGET).elf $(TARGET)-efuse.hex
        @if [ -f $(TARGET)-efuse.hex ]; then \
         $(AVRDUDE) $(AVRDUDE_FLAGS) -U efuse:w:$(TARGET)-efuse.hex;
        fi;
```

Das Target "clean" muss noch um die Zeilen

```
        $(REMOVE) $(TARGET)-lfuse.hex
        $(REMOVE) $(TARGET)-hfuse.hex
        $(REMOVE) $(TARGET)-efuse.hex
```

erweitert werden, wenn auch die Fuse-Dateien gelöscht werden sollen.

Um nun die Fusebits des angeschlossenen Controllers zu programmieren muss lediglichein "make lfuses", "make hfuses" oder "make efuses" gestartet werden. Bei den Fuse-Bits ist besondere Vorsicht geboten, da diese das Programmieren des Controllers unmöglich machen können. Also erst programmieren, wenn man einen HV-Programmierer hat oder ein paar Reserve-AVRs zur Hand ;-)

Um weiterhin den "normalen" Flash beschreiben zu können, ist es wichtig, für das Target "*.hex" im Makefile nicht nur "-R .eeprom" als Parameter zu übergeben sondern zusätzlich noch "-R lfuses -R efuses -R hfuses". Sonst bekommt AVRDUDE Probleme diese Sections in den Flash (wo sie ja nicht hingehören) zu schreiben.

Siehe auch: Vergleich der Fuses bei verschiedenen Programmen

Externe Referenzspannung des internen Analog-Digital-Wandlers

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

Watchdog

Siehe AVR-GCC-Tutorial/Der Watchdog

TODO

- Aktualisierung Register- und Bitbeschreibungen an aktuelle AVR
- stdio.h, malloc()
- "naked"-Funktionen

- Übersicht zu den C bzw. GCC-predefined Makros (__DATE__, __TIME__,...)
- Bootloader => erl. AVR Bootloader in C - eine einfache Anleitung
- Mixing C and assembly language programs Copyright © 2007 William Barnekow (PDF).

Softwareentwicklung

"Wild drauflos" zu programmieren kann nach einiger Zeit frustrieren, da mehr Zeit erforderlich wird, das Programm neuen Anforderungen anzupassen. Wer erst etwas Zeit darauf verwendet, ein offenes Konzept (erweiterbare Programmstruktur, Algorithmen) zu entwickeln (ggf. in Ruhe mit Papier und Bleistift), wird später schneller ans Ziel gelangen.

http://de.wikipedia.org/wiki/Softwareentwicklung

Programmierstil

Damit ein größeres Programm (nach längerer Zeit) überschaubar bleibt, sollte man sich bei der Gliederung, Namensgebung, Formatierung und Kommentierung an bewährten, begründeten Konzepten orientieren.

- Include-Files (C)
- www.avrfreaks.net Modularizing C Code
- www.elektroniknet.de: Der Programmierstandard misra
- Wikipedia:Programmierstil
- http://www.mikrocontroller.net/topic/130218
- http://www.mikrocontroller.net/topic/132304

Von „https://www.mikrocontroller.net/articles/AVR-GCC-Tutorial“
Kategorien: Avr-gcc Tutorial | AVR | Avr-gcc