mikrocontroller.net

AVR-GCC-Tutorial

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 (GCC) erleichtern.

Vorausgesetzt werden Grundkenntnisse der Programmiersprache 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.

Inhaltsverzeichnis

Vorwort

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 Paket WinAVR gehört die avr-libc Dokumentation zum Lieferumfang und wird mitinstalliert.

Der Compiler und die Standardbibliothek avr-libc werden ständig weiterentwickelt. Einige Unterschiede, die sich im Verlauf der Entwicklung ergeben haben, werden hier und im Artikel Alte Quellen zwar angesprochen, Anfängern und Umsteigern sei jedoch empfohlen, eine aktuelle Versionen zu nutzen.

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

Dieses Tutorial ist in PDF-Form erhältlich (zur Zeit nur eine sehr veraltete Version).

Weiterführende Kapitel

Um dieses riesige Tutorial 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.

UART
→ Hauptartikel: *Der UART*

ADC
→ Hauptartikel: *Analoge Ein- und Ausgabe (ADC)*

Timer
→ Hauptartikel: *Die Timer und Zähler des AVR*

LCD
→ Hauptartikel: *LCD-Ansteuerung*

Watchdog
→ Hauptartikel: *Der Watchdog*

Assembler
→ Hauptartikel: *Assembler und Inline-Assembler*

alte Quellen anpassen
→ Hauptartikel: *Alte Quellen anpassen*

Makefiles
→ Hauptartikel: *Exkurs Makefiles* sowie als Alternative für sehr kleine Projekte → Hauptartikel: *C ohne Makefile*

Benötigte Werkzeuge

Um eigene Programme für AVRs mittels einer AVR-Toolchain zu erstellen wird folgende Hard- und Software benötigt:

- Eine AVR-Toolchain bestehend aus avr-gcc, den avr-Binutils (Assembler, Linker, etc) und einer Standard-C Bibliothek. Üblich ist die AVR-LibC, die auch quasi in allen avr-gcc Distributionen enthalten ist.

Hardware wird keine benötigt – bis auf einen PC natürlich, auf dem der Compiler ablaufen kann. Selbst ohne AVR-Hardware kann man also bereits C-Programme für AVRs schreiben, compiliern und sich das Look-and-Feel von avr-gcc sowie von IDEs wie Atmel Studio, Eclipse oder leichtgewichtigeren Entwicklungsumbgebungen anschauen. Selbst das Debuggen und Simulieren ist mithilfe entsprechender Tools wie Debugger und Simulator in gewissen Grenzen möglich.

Um Programme für AVRs mittels einer AVR-Toolchain zu testen, wird folgende Hard- und Software benötigt:

- Platine oder Versuchsaufbau für die Aufnahme eines AVR-Controllers, der vom avr-gcc Compiler unterstützt wird.[1] 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.

- 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.

- 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, siehe auch AVR Eclipse. Ebenfalls unter Linux und Windows verfügbar ist die Entwicklungsumgebung Code::Blocks[2]. 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 Beispielprogramme 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 drei 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).

- Das Generieren des Programms ohne IDE und ohne Makefile. In diesem Fall muss die Quellcodedatei durch eine vorgefertigte Kommandofolge an den Compiler übergeben werden. Der Artikel C ohne Makefile zeigt, wie das funktioniert. Diese Vorgehensweise empfiehlt sich jedoch nur für kleine Programme, die nicht auf verschiedene Quellcodedateien verteilt sind.

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 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) {                // (5)
     /* "leere" Schleife*/   // (6)
   }                         // (7)

   /* wird nie erreicht */
   return 0;                 // (8)
}
```

1. In dieser Zeile wird eine sogenannte Header-Datei eingebunden. In `avr/io.h` sind die Registernamen definiert, die im späteren Verlauf genutzt werden. Auch unter Windows wird ein `/` zur Kennzeichnung von Unterverzeichnissen in Include-Dateinamen verwendet und kein `\`.
2. Hier beginnt das eigentliche Programm. Jedes C-Programm beginnt mit den Anweisungen in der Funktion `main`.
3. Die Anschlüsse eines AVR (Pins) 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. 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, Näheres dazu im Artikel Bitmanipulation:

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

5. ist der Beginn der sogenannte *Hauptschleife* (main-loop). Dies ist eine Endlosschleife, welche kontinuierlich wiederkehrende Befehle enthält.
6. In diesem Beispiel ist die Hauptschleife leer. Der Controller durchläuft die Schleife immer wieder, ohne dass etwas passiert. 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 kehrt das Programm aus `main` zurück, alle Interrupts werden deaktiviert und eine Endlosschleife betreten.
7. Ende der Hauptschleife und Sprung zur passenden, öffnenden Klammer, also zu 5.

8. ist das Programmende. Die Zeile ist nur aus Gründen der C-Kompatibilität enthalten:

```
int main(void)
```

besagt, dass die Funktion einen int-Wert zurückgibt. Die Anweisung wird aber nicht erreicht, da das Programm die Hauptschleife nie verlässt.

Um diesen Quellcode in ein 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:\beispiel>dir

 Verzeichnis von D:\beispiel

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 welcher der Code für den AVR enthalten ist.

```
D:\beispiel>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[3], 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 Datentypen (Integer)

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, die folgendermaßen eingebunden werden kann:

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

int-Typen aus `stdint.h` (C99)

Vorzeichenbehaftete int-Typen				
Typname	**Bit-Breite**	**Wertebereich**		**C-Entsprechung** (avr-gcc)
`int8_t`	8	−128 ⋯ 127	$-2^7 \cdots 2^7-1$	`signed char`
`int16_t`	16	−32768 ⋯ 32767	$-2^{15} \cdots 2^{15}-1$	`signed short`, `signed int`
`int32_t`	32	−2147483648 ⋯ 2147483647	$-2^{31} \cdots 2^{31}-1$	`signed long`
`int64_t`	64	−9223372036854775808 ⋯ 9223372036854775807	$-2^{63} \cdots 2^{63}-1$	`signed long long`
Vorzeichenlose int-Typen				
Typname	**Bit-Breite**	**Wertebereich**		**C-Entsprechung** (avr-gcc)
`uint8_t`	8	0 ⋯ 255	$0 \cdots 2^8-1$	`unsigned char`
`uint16_t`	16	0 ⋯ 65535	$0 \cdots 2^{16}-1$	`unsigned short`, `unsigned int`
`uint32_t`	32	0 ⋯ 4294967295	$0 \cdots 2^{32}-1$	`unsigned long`
`uint64_t`	64	0 ⋯ 18446744073709551615	$0 \cdots 2^{64}-1$	`unsigned long long`

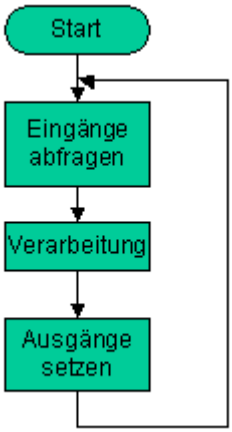
Neben den Typen gibt es auch Makros für die Bereichsgrenzen wie `INT8_MIN` oder `UINT16_MAX`. Siehe dazu auch: Dokumentation der avr-libc: Standard Integer Types.

Grundsätzlicher Programmaufbau eines µC-Programms

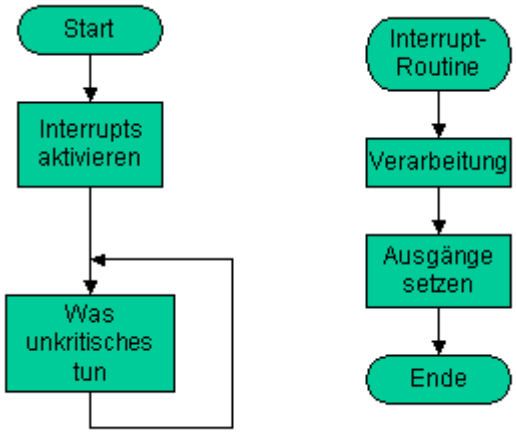
Wir unterscheiden zwischen 2 verschiedenen Methoden, um ein Mikrocontroller-Programm zu schreiben, und zwar völlig unabhängig davon, in welcher Programmiersprache das Programm geschrieben wird.

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.[4]

Beispiel:

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

int main()
```

```
{
    /* 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);

    while (1);
}
```

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 unter Bitmanipulation.

Der gcc C-Compiler unterstützt ab Version 4.3.0 Konstanten im Binärformat, z. B. DDRB = 0b00011111. Diese Schreibweise ist jedoch nur in GNU-C verfügbar und nicht in ISO-C definiert. Man sollte sie daher 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 */
```

Bei bestimmten AVR Registern mit Bits, die durch Beschreiben mit einer logischen 1 gelöscht werden, muss eine absolute Zuweisung benutzt werden. Ein ODER löscht in diesen Registern ALLE gesetzten Bits!

Beispiel:

```
#include <avr/io.h>
...
TIFR2 = (1<<OCF2A); // Nur Bit OCF2A löschen
```

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 (bitweise 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 not-Operator
if ( !i ) {
  test2();
}
// nochmals verkürzt:
if ( !( PINA & 0x01 ) ) {
  test2();
}
```

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. Dies 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 (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: */

void foo (void)
{
   uint8_t 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.

Siehe auch: avr-libc FAQ: "How do I pass an IO port as a parameter to a function?"

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.

Die physischen Ein- und Ausgänge werden bei AVR-Controllern zu logischen Ports gruppiert.

Alle Ports werden über Register gesteuert. Dazu sind jedem Port 3 Register zugeordnet:

DDRx	Datenrichtungsregister für Port**x**. **x** entspricht **A**, **B**, **C**, **D** usw. (abhängig von der Anzahl der Ports des verwendeten AVR). Bit im Register gesetzt (1) für Ausgang, Bit gelöscht (0) für Eingang.
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
  // direkte Zuweisung - standardkonform */
  DDRB = 0x1F;    /*

  // übersichtliche Alternative - Binärschreibweise, aber kein ISO-C
  DDRB = 0b00011111;

  // 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.

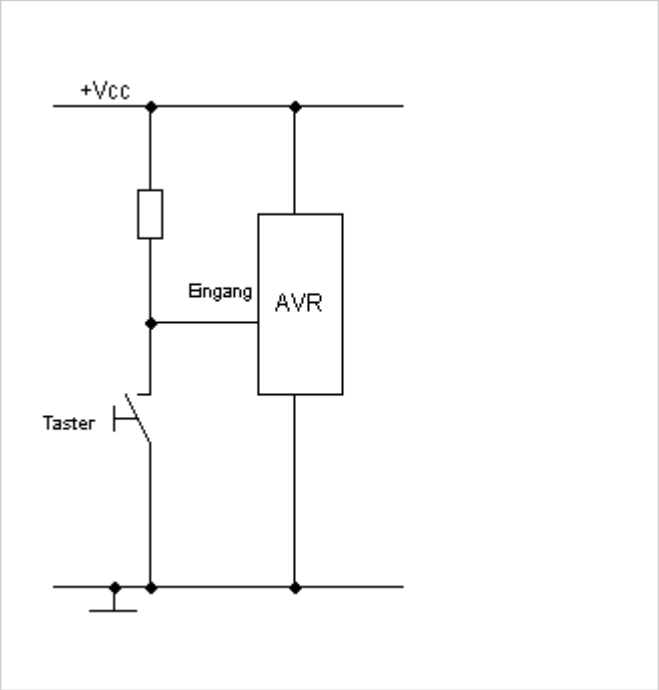
Im Beispiel werden alle Pins des Ports D als Eingänge geschaltet und alle Pull-Up Widerstände aktiviert. Weiterhin wird Pin PC7 als Eingang geschaltet und dessen interner Pull-Up Widerstand aktiviert, ohne die Einstellungen für die anderen Portpins (PC0-PC6) zu verändern.

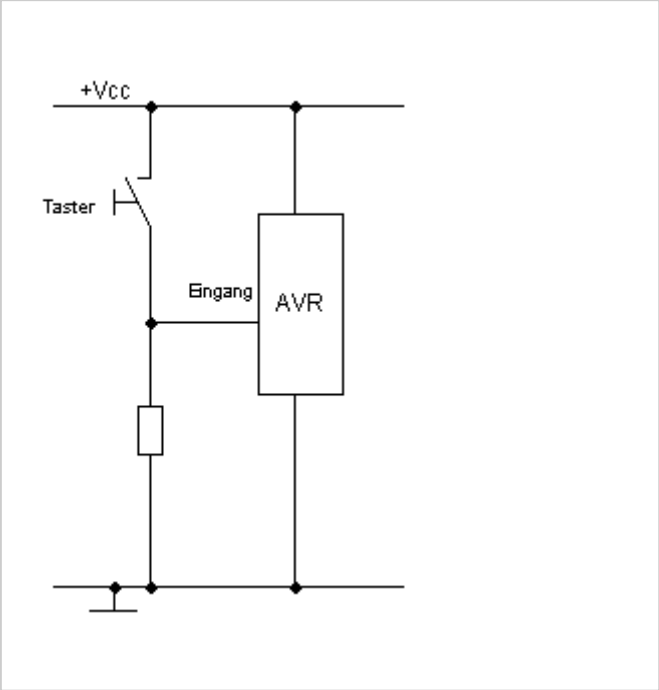
```
#include <avr/io.h>
...
DDRD  = 0x00; /* alle Pins von Port D als Eingang */
PORTD = 0xff; /* interne Pull-Ups an allen Port-Pins aktivieren */
...
DDRC  &= ~(1<<PC7);  /* Pin PC7 als Eingang */
PORTC |= (1<<PC7);    /* internen Pull-Up an PC7 aktivieren */
```

Taster 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.

Taster entprellen

Siehe: *Entprellung: Warteschleifen-Verfahren*

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 (abgesehen von möglicherweise auftretenden Interrupts, falls welche aktiviert sind). 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, siehe Artikel Multitasking.

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.

Eine weitere Einschränkung liegt darin, daß sie möglicherweise länger warten, als erwartet, nämlich in dem Fall, daß Interrupts auftreten und die _delay...()-Funktion unterbrechen. Genau genommen warten diese nämlich nicht eine bestimmte Zeit, sondern verbrauchen eine bestimmte Anzahl von Prozessortakten. Die wiederum ist so bemessen, daß ohne Unterbrechung durch Interrupts die gewünschte Wartezeit erreicht wird. Wird das Warten aber durch eine oder mehrere ISR unterbrochen, die zusammen 1% Prozessorzeit verbrauchen, dann dauert das Warten etwa 1% länger. Bei 50% Last durch die ISR dauert das Warten doppelt solange wie gewünscht, bei 90% zehnmal solange...

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

avr-libc Versionen bis 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. 38,4µs warten. Längere Wartezeiten müssen dann über einen mehrfachen Aufruf in einer Schleife gelöst werden.

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

```
#include <avr/io.h>
#ifndef F_CPU
/* Definiere F_CPU, wenn F_CPU nicht bereits vorher definiert
   (z. B. durch Übergabe als Parameter zum Compiler innerhalb
   des Makefiles). Zusätzlich Ausgabe einer Warnung, die auf die
   "nachträgliche" Definition hinweist */
```

```
#warning "F_CPU war noch nicht definiert, wird nun mit 3686400 definiert"
#define F_CPU 3686400UL     /* Quarz mit 3.6864 Mhz */
#endif
#include <util/delay.h>     /* in älteren avr-libc Versionen <avr/delay.h> */

/*
 lange, variable Verzögerungszeit, Einheit in Millisekunden

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

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

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

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

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

    return 0;
}
```

avr-libc Versionen ab 1.7

_delay_ms() kann mit einem Argument bis 6553,5 ms (= 6,5535 Sekunden) benutzt werden. Es ist nicht möglich, eine Variable als Argument zu übergeben. 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
```

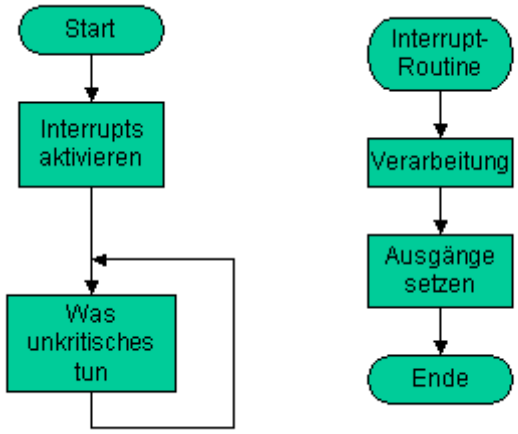
```
    while( 1 ) {                // Endlosschleife
        PORTB ^= ( 1 << PB0 );  // Toggle PB0 z.B. angeschlossene LED
        _delay_ms(1000);        // Eine Sekunde +/-1/10000 Sekunde warten...
                                // funktioniert nicht mit Bibliotheken vor 1.6

    }
    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: [1]

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.



Tritt ein Interrupt auf, unterbricht (engl. interrupts) der Controller die Verarbeitung des Hauptprogramms und verzweigt zu einer Interruptroutine. Das Hauptprogramm wird also beim Eintreffen eines Interrupts unterbrochen, die Interruptroutine ausgeführt und danach erst wieder das Hauptprogramm an der Unterbrechungsstelle fortgesetzt (vgl. die Abbildung).

Um Interrupts verarbeiten zu können, ist folgendes zu beachten:

- Für jede aktivierte Interruptquelle ist eine Funktion zu programmieren, in der die beim Auftreten des jeweiligen Interrupts erforderlichen Verarbeitungsschritte enthalten sind. Für diese Funktion existieren verschiedene Bezeichnungen. Üblich sind die englischen Begriffe Interrupt-Handler oder Interrupt-Service-Routinen (ISR), man findet aber auch die Bezeichnungen Interruptverarbeitungs- oder -behandlungsroutine oder auch kurz Interruptroutine. Zum Beispiel wird üblicherweise in der ISR zur Verarbeitung des Empfangsinterrupts eines UARTs (UART-RX Interrupt) das empfangene Zeichen in einen Zwischenspeicher (FIFO-Buffer) kopiert, dessen Inhalt später von anderen Programmteilen geleert wird. Sofern der Zwischenspeicher ausreichend groß ist, geht also kein Zeichen verloren, auch wenn im Hauptprogramm zeitintensive Operationen durchgeführt werden.
- Die benötigten Interrupts sind in den jeweiligen Funktionsbausteinen einzuschalten. Dies erfolgt über das jeweilige Aktivierungsbit (Interrupt Enable) in einem der Hardwareregister (z.B. RX(Complete)Interrupt Enable eines UARTs)
- Sämtliche Interrupts werden über einen weiteren globalen Schalter aktiviert und deaktiviert. Zur Verarbeitung der Interrupts ist dieser Schalter zu aktivieren (sei(), siehe unten).

Alle Punkte sind zu beachten. Fehlt z.B. die globale Aktivierung, werden Interruptroutinen auch dann nicht aufgerufen, wenn sie im Funktionsbaustein eingeschaltet sind und eine Behandlungsroutine verhanden ist.

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 zusammenhängen.

GIMSK

General **I**nterrupt **M**a**sk** Register.

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

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

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

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

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

GIFR

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

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

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

Dieses Bit wird gesetzt, wenn am **INT1**-Pin eine Interrupt-Bedingung, entsprechend der Konfiguration, als eingetreten 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-Bedingung, entsprechend der Konfiguration, als eingetreten 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 über den 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. Dieses wird automatisch wieder 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. Es ist möglich das GIE-Bit in der ISR zu setzen und so schon wieder weitere Interrupts zuzulassen - allerdings sollte man damit vorsichtig sein und genau wissen was man damit macht. Kritisch wird es vor allem wenn der gleiche Interrupt noch einmal kommt, bevor die ISR abgearbeitet ist.

Interrupts mit avr-gcc

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 durch Rekursion 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<<PA4);
//...
}
```

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 auf **1**, PORTA ist danach 0b0000000**1**. 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:
Bei Registern mit mehreren Interrupt-Flag-Bits (wie die Timer Interrupt Flag Register) **nicht** die übliche bitweise VerODERung nehmen, sondern eine direkte Zuweisung machen. Da sonst weitere Flags, als nur das gewünschte, ebenfalls gelöscht werden könnten.
(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

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

Die vielen Prozessoren aus der AVR-Familie unterstützen unterschiedliche Sleep-Modi, gefächert nach Vorhandensein von Funktionsblöcken im Controller. Konkrete und verläßliche Auskunft über die tatsächlichen Gegebenheiten finden sich wie immer in den jeweiligen Datenblättern. Die Modi unterscheiden sich darin, welche Funktionsbereiche zum Energiesparen abgeschaltet werden. Davon hängt auch ab, mit welchen Mitteln der Prozessor aus der jeweiligen Schlaftiefe wieder aufgeweckt werden kann.

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, die das Meßergebnis negativ beeinflussen können. Das 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), wenn vorhanden. 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.

Abschalten des Brownout Detect (BOD) während der Sleep-Phase (nur P-Typen)
: Zur Stromersparnis bieten die P-Typen die Möglichkeit den BOD während der Sleep-Phase abzuschalten. Bei einem Atmega88PA beispielsweise, kann dadurch der Stromverbrauch im

SLEEP_MODE_PWR_SAVE mit Timer2 im Asynchronmodus mit Uhrenquarz und periodischer Selbstaufweckung um ca. 50% gesenkt werden.

Das Einschalten dieser Funktion geschieht in einer Timed Sequence.

```
unsigned char temp0 = MCUCR;
unsigned char temp1 = MCUCR;
temp0 |= (1 << BODS) | (1 << BODSE);
temp1 |= (1 << BODS);
MCUCR = temp0;
MCUCR = temp1;
sleep_cpu();
```

Hierbei ist unbedingt zu beachten, dass das BODS-Bit 3 Takte nach dem Setzen wieder gelöscht wird. Daher muss der Aufruf des Sleep unmittelbar nach dem Setzen erfolgen und das BODS-Bit muss jedes Mal vor einem Sleep Aufruf erneut gesetzt werden.

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*) sind Variablen, die die Adresse von Daten oder Funktionen enthalten und belegen 16 Bits. Die Größe hängt mit dem adressierbaren Speicherbereich zusammen und der 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

Flash mit PROGMEM und pgm_read

→ avr-libc: Doku zu avr/pgmspace.h

Ein Zugriff auf Konstanten im Programmspeicher ist mittels avr-gcc erst ab Version 4.7 "transparent" möglich. Um Daten aus dem Flash zu lesen, muss die AVR-Instruktion LPM (*Load from Program Memory*) erzeugt werden, bei Controllern mit mehr als 64kiB Flash auch ELMP.

Dazu gibt es das AVR-spezifische GCC-Attribut `progmem`, mit dem eine Variablendeklaration im *static storage*[5] markiert werden kann:

```
const int value __attribute__((progmem)) = 1;
```

Effekt ist, dass die so markierte Variable nicht im RAM sondern im Flash angelegt wird. Wird durch "normalen" C-Code auf solch eine Variable zugegriffen, wird jedoch aus dem RAM gelesen und nicht aus dem Flash!

Zum Lesen aus dem Flash stellt die avr-libc daher zahlreiche Makros zur Verfügung. Zudem wird das Makro `PROGMEM` definiert, das etwas Tipparbeit spart:

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

const int value PROGMEM = 1;
```

progmem funktioniert im Wesentlichen wie ein Section-Attribut, das die Daten in der Section .progmem.data ablegt. Im Gegensatz zum Section-Attribut werden jedoch noch weitere Prüfungen unternommen, ab avr-gcc 4.6 etwa muss die entsprechende Variable const sein.

Integer und float

Zum Lesen von Skalaren stellt die avr-libc folgende Makros zu Verfügung, die jeweils ein Argument erhalten: Die 16-Bit Adresse des zu lesenden Wertes[6]

Übersicht der pgm_read Funktionen aus dem Header avr/pgmspace.h der avr-libc

Gelesener Wert	pgm_read_xxx	Anzahl Bytes
uint8_t	pgm_read_byte	1
uint16_t	pgm_read_word	2
uint32_t	pgm_read_dword	4
float	pgm_read_float[7]	4

Soll ein Zeiger gelesen werden, so verwendet man pgm_read_word und castet das Ergebnis zum gewünschten Zeiger-Typ.

Beispiele

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

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

/* int-Array */
const int anArray[] PROGMEM = { 18, 3 ,70 };

void foo (void)
{
  /* Zeiger */
  static const uint8_t* const aPointer PROGMEM = &aByte;

  uint8_t a        = pgm_read_byte (&aByte);
  int a2           = (int) pgm_read_word (&anArray[2]);
  const uint8_t* p = (const uint8_t*) pgm_read_word (&aPointer);
}
```

Blöcke

In den Flash-Funktionen der avr-libc sind keine der pgm_read_xxxx Nomenklatur folgenden Funktionen, die Speicherblöcke auslesen oder vergleichen. Die enstprechende Funktionen sind Varianten von memcpy, memcmp und heißt memcpy_P, memcmp_P, usw. Für weitere Funktionen und deren Prototypen siehe die Dokumentation der avr-libc.

Strings

Strings sind in C nichts anderes als eine Abfolge von Zeichen und einem '\0' als Stringende. Der prinzipielle Weg ist daher identisch zum Lesen von Bytes, wobei auf die Besonderheiten von Strings wie 0-Terminierung geachtet werden muss.

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

size_t my_string_length (const char* addr)
{
    size_t length = 0;

    while (pgm_read_byte (addr++))
    {
        len++;
    }
    return length;
}
```

Zur Unterstützung des Programmierers steht das Repertoire der str-Funktionen auch in jeweils eine Variante zur Verfügung, die mit dem Flash-Speicher arbeiten kann. Die Funktionsnamen tragen den Suffix _P. Darüber hinaus gibt es das Makro PSTR, das ein String-Literal im Flash-Speicher ablegt und die Adresse des Strings liefert:

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

/* Liefert true zurück, wenn string_im_ram gleich "Hallo Welt" ist. */
int foo (const char* string_im_ram)
{
    return strcmp_P (string_im_ram, PSTR ("Hallo Welt"));
}
```

Zu beachten ist, dass PSTR nur innerhalb von Funktionen verwendet werden kann.

Array aus Strings

Arrays aus Strings im Flash-Speicher werden in zwei Schritten angelegt:

1. Zuerst die einzelnen Elemente des Arrays und
2. im Anschluss ein Array, in dem die Startaddressen der Strings abgelegt werden.

Zum Auslesen wird zuerst die Adresse des gewünschten Elements aus dem Array im Flash-Speicher gelesen, die im Anschluss dazu genutzt wird, um auf das Element (den String) selbst zuzugreifen.

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

static const char str1[] PROGMEM = "Hund";
static const char str2[] PROGMEM = "Katze";
static const char str3[] PROGMEM = "Maus";

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

// Liest den i-ten String von array[] und kopiert ihn ins RAM nach buf[]
void read_string (char* buf, size_t i)
{
    // Lese die Adresse des i-ten Strings aus array[]
    const char *parray = (const char*) pgm_read_word (&array[i]);

    // Kopiere den Inhalt der Zeichenkette vom Flash ins RAM
    strcpy_P (buf, parray);
}
```

Eine weitere Möglichkeit ist, die Strings in einem 2-dimensionalen char-Array abzulegen anstatt deren Adresse in einem 1-dimensionalen Adress-Array zu speichern.

Vorteil ist, dass der Code einfacher wird. Nachteil ist, dass bei unterschiedlich langen Strings Speicherplatz verschwendet wird, weil sich die Array-Dimension and der Länge des längsten Strings orientieret. Bei in etwa gleich langen Strings kann es aber sogar Speicherplatz sparen, denn es die Adressen der einzelnen Strings müssen nicht abgespeichert werden.[8]

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

// Die "6" ist 1 plus die Länge des längsten Strings ("Katze")
const char array[][6] PROGMEM =
{
   "Hund", "Katze", "Maus"
};

// Liest den i-ten String von array[] und kopiert ihn ins RAM nach buf[]
void read_string (char* buf, size_t i)
{
    // Kopiere den Inhalt der i-ten Zeichenkette vom Flash ins RAM
    strcpy_P (buf, array[i]);
}
```

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

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 sieht 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, dann weiß die Funktion nicht, ob die Adresse in den Flash-Speicher oder das RAM zeigt. Weder aus dem Pointer-Wert, also dem Zahlenwert, noch aus dem "const" kann auf den Ort der Ablage geschlossen werden.

Einige AVR-Compiler bilden die Harvard-Architektur ab, indem sie in einen Pointer nicht nur die Adresse speichern, sondern auch den Ablageort wie *Flash* oder *RAM*. In einem Aufruf einer Funktion wird dann bei Pointer-Parametern neben der Adresse auch der Speicherbereich, auf den der Pointer zeigt, übergeben.

Dies hat jedoch auch Nachteile, denn bei jedem Zugriff über einen Zeiger muss zur *Laufzeit* entschieden werden, wie der Zugriff auszuführen ist und entsprechend länglicher und langsamer wird der erzeugte Code.

Siehe auch:

- Dokumentation der avr-libc Abschnitte Modules/Program Space String Utilities und Abschnitt Modules/Bootloader Support Utilities

Variablenzugriff >64kB

Die Zeiger beim avr-gcc sind nur 16 Bit breit, können somit also nur 64kiB Datenspeicher adressieren. Als Funktionspointer können sie beim AVR bis zu 128 kiB Programmspeicher adressieren, weil Funktionsadressen immer 16-Bit Worte adressieren und nicht Bytes. Um Flashzugriff jenseits von 64KiB zu bewerkstelligen gibt es mehrere Möglichkeiten:

- Address-Spaces wie `__flash1` oder `__memx`, siehe Abschnitt "Jenseits von __flash".
- Die Funktionen bzw. Makros `pgm_read_xxx_far` der AVR-Libc, wie im folgenden beschrieben.

Unverständlicherweise gibt es in der AVR-Libc keine Funktion, um 32-Bit Pointer zu erhalten. Hier schafft ein eigenes GNU-C99 Makro Abhilfe:

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

//============================================================
// Macro to access strings defined in PROGMEM above 64kB
```

```
//-------------------------------------------------------------------
#define FAR(var)                    \
({ uint_farptr_t tmp;               \
   __asm__ (                        \
       "ldi    %A0, lo8(%1)"  "\n\t" \
       "ldi    %B0, hi8(%1)"  "\n\t" \
       "ldi    %C0, hh8(%1)"        \
       : "=d" (tmp)                 \
       : "i"  (&(var)));            \
   tmp;                             \
})
//-------------------------------------------------------------------

//===================================================================
// Define a section above 64kiB (FAR_SECTION)
// and add the required linker argument below
// -Wl,--section-start=.far_section=0x10000
//-------------------------------------------------------------------
#define FAR_SECTION   __attribute__((__section__(".far_section")))
//-------------------------------------------------------------------

//===================================================================
// Just a Sample
//-------------------------------------------------------------------

const char MyString[] FAR_SECTION = "Hier liegt mein FAR-Teststring!";
const char MyBmp64[]  FAR_SECTION = {0xAA,0xBB,0xCC,0xDD,0xEE,0xFF,0x00};

int main(void)
{
  char MyChar;
  DDRC = 0xFF;
  do
  {
    MyChar = pgm_read_byte_far(FAR(MyBmp64));
    PORTC  = MyChar;
  }
  while(MyChar);
}
```

D.h. man muss

- Das Makro `FAR` im Quellcode einfügen
- Die Definition der neuen Section `FAR_SECTION` einfügen
- Die Variablen mit dieser Section kennzeichnen
- Dem Linker mittels Kommandozeilenoption die Startadrese dieser Section mitteilen

Der Zugriff auf diese Variablen kann nur mittels direkter Pointerarithmetik erfolgen, eine Indizierung von Arrays mit variablem Index ist nicht möglich.

```
int n=3;
MyChar = pgm_read_byte_far(FAR(MyBmp64)+n);
```

Flash mit __flash und Embedded-C

Ab Version 4.7 unterstützt avr-gcc *Adress-Spaces* gemäß dem Embedded-C Dokument ISO/IEC TR18037. Der geläufigste Adress-Space ist `__flash`, der im Gegensatz zu `progmem` kein GCC-Attribut ist, sondern ein Qualifier und damit syntaktisch ähnlich verwendet wird wie `const` oder `volatile`.

GCC kennt keine eigene Option zum Aktivieren von Embedded-C, es wird als GNU-C Erweiterung behandelt. Daher müssen C-Module, die Address-Spaces verwenden, mit `-std=gnu99` o.ä. compiliert werden.

```
static const __flash int value = 10;
```

```
int get_value (void)
{
  return value;
}
```

1. Im Gegensatz zu `progmem` sind keine speziellen Bibliotheksfunktionen oder -makros für den Zugriff mehr notwendig: Der Code zum Lesen der Variable ist "normales" C.
2. Die Variable wird im richtigen Speicherbereich (Flash) angelegt.
3. `__flash` ist nur zusammen mit read-only Objekten oder Zeigern, d.h. nur zusammen mit `const`, erlaubt.
4. Zugriffe wie im obigen Beispiel können (weg)optimiert werden. Das Beispiel entspricht einem "`return 10`". Es besteht keine Notwendigkeit, für `value` überhaupt Flash-Speicher zu reservieren.

Auch Zeiger-Indirektionen sind problemlos möglich. Zu beachten ist, dass `__flash` auf der richtigen Seite des "`*`" in der Zeigerdeklaration bzw. -definition steht:

- **Rechts vom `*`:** Der Zeiger selbst liegt im Flash
- **Links vom `*`:** Der Zeiger enthält eine Flash-Adresse

```
// val ist eine Variable im Flash
const __flash int val = 42;

// pval liegt auch im Flash und enthält die Adresse von val
const __flash int* const __flash pval = &val;

char get_val (void)
{
  // liest den Wert von val über die in pval abgelegte Adresse
  return *pval;
}
```

Blöcke

Um Speicherbereiche vom Flash in den RAM zu kopieren, gibt es zwei Möglichkeiten: Zum einen können wie bei `progmem` beschreiben die Funktionen der avr-libc wie `memcpy_P`, `memcmp_P`, `movmem_P`, etc. verwendet werden:

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

// Eine Datenstruktur
typedef struct
{
    int id;
    char buf[10];
} data_t;

extern void uart_send (const void*, size_t);

void send_data (const __flash data_t *pdata)
{
    // buf wird auf dem Stack angelegt
    data_t buf;

    // Kopiere Daten vom Flash nach buf ins RAM
    memcpy_P (&buf, pdata, sizeof (data_t));

    // Sende die Daten in buf
    uart_send (&buf, sizeof (data_t));
}
```

Zum anderen kann eine Struktur auch über direktes Kopieren ins RAM geladen werden:

```
#include <stdlib.h>
```

```
// Eine Datenstruktur
typedef struct
{
    int id;
    char buf[10];
} data_t;

extern void uart_send (const void*, size_t);

void send_data (const __flash data_t *pdata)
{
    // Kopiere Daten ins RAM.  buf wird auf dem Stack angelegt
    const data_t buf = *pdata;

    // Verwendet die Daten in buf
    uart_send (&buf, sizeof (data_t));
}
```

Strings

Natürlich können auch Strings im Flash abgelegt werden und auch mit Funktionen wie `strcpy_P` aus der avr-libc verarbeitet werden. Zudem ist es möglich, Flash-Zeiger mit der Adresse eines String-Literals zu initialisieren:

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

#define FSTR(X) ((const __flash char[]) { X } )

const __flash char * const __flash array[] =
{
    FSTR ("Hund"), FSTR ("Katze"), FSTR ("Maus")
};

size_t get_len (uint8_t tier)
{
    return strlen_P (array[tier]);
}
```

Leider sieht der Embedded-C Draft nicht vor, String-Literale direkt in einem anderen Adress-Space als *generic* anzulegen, so dass hier der Umweg über `FSTR` genommen werden muss. Dieses Konstrukt ist nur ausserhalb von Funktionen möglich und kann daher nicht als Ersatz für `PSTR` aus der avr-libc dienen.

Soll `array` ein 2-dimensonales Array sein anstatt ein 1-dimensionales Array von Zeigern, dann geht das ohne große Verrenkungen:

```
// Die 6 ergibt sich aus 1 plus der Länge des längsten Strings "Katze"
const __flash char array[][6] =
{
    "Hund", "Katze", "Maus"
};
```

Weiters besteht die Möglichkeit, `array` analog anzulegen, wie man es mit `PROGMEM` machen würde: Jeder String wird explizit angelegt und seine Adresse bei der Initialisierung von `array` verwendet. Dies entspricht dem ersten Beispiel eines 1-dimensionalen Zeigerarrays:

```
static const __flash char strHund[]  = "Hund";
static const __flash char strKatze[] = "Katze";
static const __flash char strMaus[]  = "Maus";

const __flash char * const __flash array[] =
{
    strHund, strKatze, strMaus
};
```

Casts

Embedded C fordert, dass zwei Adress-Spaces entweder disjunkt sind – d.h. sie enthalten keine gemeinsamen Adressen – oder aber ein Space komplett im anderen enthalten ist, also eine Teilmengen-Beziehung besteht. Die Adress-Spaces von avr-gcc sind so implementiert, dass jeder Space Teilmenge jedes anderes ist. Zwar haben Spaces wie RAM und Flash physikalisch keinen Speicherbereich gemein, allerdings ermöglicht diese Implementierung das Casten von Zeigern zu unterschiedlichen Adress-Spaces[9]:

```
#include <stdbool.h>

char read_char (const char *address, bool data_in_flash)
{
    if (data_in_flash)
        return *(const __flash char*) address;
    else
        return *address;
}
```

Der Cast selbst erzeugt keinen zusätzlichen Code, da eine RAM-Adresse und eine Flash-Adresse die gleiche Binärdarstellung haben. Allerdings wird über den nach `__flash` gecasteten Zeiger anders zugegriffen, nämlich per LPM.

Jenseits von __flash

Ausser `__flash` gibt es auch folgende Address-Spaces:

`__flash`*N*

Mit *N* = 1..5 sind fünf weitere Spaces, die analog zu `__flash` funktionieren und deren Zeiger ebenfalls 16 Bit breit sind. avr-gcc erwartet, dass die zugehörigen Daten, welche in die Section `.progmem`*N*`.data` abgelegt werden, so lokatiert sind, dass das high-Byte der Adresse (Bits 16..23) gerade *N* ist. Dies wird in Binutils noch nicht unterstützt[10] (Stand Binutils 2.24) Die Unterstützung kann durch ein eigenes Linker-Skript erreicht werden, welches diese Sections wie vom Compiler erwartet lokatiert.

`__memx`

Dieser Address-Space implementiert 3-Byte Zeiger und unterstützt Lesen über 64KiB-Segmentgrenzen hinweg. Das MSB (Bit 23) gibt dabei an, ob der `__memx`-Zeiger eine Flash-Adresse enthält (Bit23 = 0) oder eine RAM-Adresse (Bit23 = 1), was folgenden Code erlaubt:

```
const __memx int a_flash = 42;
const        int a_ram   = 100;

static int get_a (const __memx int* pa)
{
    return *pa;
}

int main (void)
{
    return get_a (&a_flash) + get_a (&a_ram);
}
```

Dies bedeutet, dass erst zur *Laufzeit* entschieden werden kann, ob aus dem RAM oder aus dem Flash gelesen werden soll, was `__memx` im Vergleich zu den anderen Address-Spaces langsamer macht. Ausserdem ist zu beachten, dass `__memx`-Zeiger zwar 24-Bit Zeiger sind, die zugrundeliegende Adress-Arithmetik jedoch gemäß dem C-Standard erfolgt, also als 16-Bit Arithmetik. Bestehende Funktion der avr libc wie z.B. printf_P funktionieren damit ebensowenig wie printf! Wenn man __memx verwenden will, braucht man dafür eigene Funktionen.

__flash, progmem und Portierbarkeit

Da ab er aktuellen Compilerversion 4.7 sowohl `__flash` als auch `PROGMEM` und die `pgm_read`-Funktionen zur Verfügung stehen, ergibt sich die Frage, welche Variante "besser" ist und wie zwischen ihnen hin- und her zu portieren ist.

Zunächst sei erwähnt, dass `__flash` kein Ersatz für `PROGMEM` ist, sondern lediglich eine Alternative dazu. Das "alte" progmem wird weiterhin mir gleicher Semantik unterstützt, so dass alter Code ohne Änderungen mit den neueren Compilerversionen übersetzbar bleibt.

Von der Codegüte her dürften sich keine großen Unterschiede ergeben. Es ist nicht zu erwarten, dass die eine oder die andere Variante wesentlich besseren oder schlechteren Code erzeugt — von einer Ausnahme abgesehen: Der Wert beim Zugriff ist zur Compilezeit bekannt und kann daher eliminiert werden.

```
static const __flash char x[] = { 'A', 'V', 'R' };

char foo (void)
{
    return x[2];
}
```

Dies wird übersetzt wie "`return 'R';`", und das Array `x[]` kann komplett wegoptimiert werden und entfallen.

progmem → __flash

Portierung in diese Richtung bedeutet, alten Code anzupassen. Zwingend ist die Portierung nicht, da `progmem` weiterhin unterstützt wird. Allerdings ist eine Quelle mit `__flash` besser lesbar, denn der Code wird von den `pgm_read`-Funktionen befreit, die vor allem bei Mehrfach-Indirektion den Code ziemlich verunstalten und unleserlich machen können. Weiterer Vorteil von `_flash` ist, daß eine striktere Typprüfung erfolgen kann.

Eine Portierung wird man in zwei Schritten vornehmen:

1. Definitionen von Flash-Variablen werden angepasst

Vorher:

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

static const char hund[]  PROGMEM = "Hund";
static const char katze[] PROGMEM = "Katze";
static const char maus[]  PROGMEM = "Maus";

const char * const tier[] PROGMEM =
{
   hund, katze, maus
};
```

Nachher:

```
static const __flash char hund[]  = "Hund";
static const __flash char katze[] = "Katze";
static const __flash char maus[]  = "Maus";

const __flash char * const __flash tier[] =
{
   hund, katze, maus
};
```

Der Header `avr/pgmspace.h` wird nicht mehr benötigt. Im Gegensatz zu `progmem` müssen Qualifier immer links von der definierten Variablen stehen; bei Attributen wie `progmem` ist das mehr oder weniger egal.

Nachdem diese Anpassung erfolgreich abgeschlossen ist, folgt Schritt

2. Der Code wird von `pgm_read`-Aufrufen bereinigt

Vorher:

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

extern const char *tier[];

char first_letter (uint8_t i)
{
    const char* ptier = (const char*) pgm_read_word (&tier[i]);
    return (char) pgm_read_byte (&ptier[0]);
}
```

Nachher:

```
#include <stdint.h>

extern const __flash char * const __flash tier[];

char first_letter (uint8_t i)
{
    return tier[i][0];
}
```

Dateien direkt im Flash einbinden

Wenn man größere Dateien direkt im Programm einbinden will, ohne sie vorher in C Quelltext umzuwandeln, muss man das mit dem Linker machen. Wie das geht steht hier.

- Atmel, avr gcc Dokumentation
- Nongnu avr gcc Dokumentation

Wie man das dann praktisch umsetzt, sieht man in diesem Beitrag.

- Forumsbeitrag: Binärdateien mittels Linker einbinden
- Forumsbeitrag: Ein kleines Tool zum Umwandeln von Binärdateien in C-Quelltext.

Flash in der Anwendung schreiben

Bei AVRs mit "self-programming"-Option – auch bekannt als Bootloader-Support – können Teile des Flash-Speichers vom Anwendungsprogramm beschrieben werden. Dies ist nur möglich, wenn die Schreibfunktion in einem besonderen Speicherbereich, der Boot-Section des Programmspeichers/Flash, abgelegt ist.

Bei einigen 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

EEPROM

Möchte man Werte aus einem Programm heraus so speichern, dass sie auch nach dem Abschalten der Versorgungsspannung noch erhalten bleiben und nach dem Wiederherstellen der Versorgungsspannung bei erneutem Programmstart wieder zur Verfügung stehen, dann benutzt man das EEPROM.

Schreib- und Lesezugriffe auf den EEPROM-Speicher erfolgen über die im Modul eeprom.h der avr-libc definierten Funktionen. Mit diesen Funktionen können einzelne Bytes, Datenworte (16 Bit), Fließkommawerte (32 Bit, single-precision, float) und Datenblöcke geschrieben und gelesen werden.

Diese Funktionen kümmern sich auch um diverse Details, die bei der Benutzung des EEPROMs normalerweise notwendig sind:

- EEPROM-Operationen sind im Vergleich relativ langsam. Man muss daher darauf achten, dass eine vorhergehende Operation abgeschlossen ist, ehe die nächste Operation mit dem EEPROM gestartet wird. Die in der avr-libc implementierten Funktionen aus eeprom.h berücksichtigten dies. Soll beim Aufruf einer EEPROM-Funktion sichergestellt werden, dass diese nicht intern in einer Warteschleife auf den Abschluss der vorherigen Operation wartet, kann vorher per eeprom_is_ready testen, ob der Zugriff auf den EEPROM-Speicher sofort möglich ist.
- Es ist darauf zu achten, dass die EEPROM-Funktionen nicht durch einen Interrupt unterbrochen werden. Einige Phasen des Zugriffs sind zeitkritisch und müssen in einer definierten bzw. begrenzten Anzahl von Takten durchgeführt werden. Durch einen unterbrechenden Interrupt würde diese Restriktion nicht mehr eingehalten. Auch dieses Detail wird von den avr-libc Funktionen berücksichtigt, so dass man sich als C-Programmierer nicht darum kümmern muss. Innerhalb der Funktionen werden Interrupts vor der "EEPROM-Sequenz" global deaktiviert und im Anschluss, falls vorher auch schon eingeschaltet, wieder aktiviert.

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 gesamten EEPROM-Speicher, erreichen. Auf jeden Fall sollte man aber eine Abschätzung über die zu erwartende Lebensdauer des EEPROM durchführen. Wird ein Wert im EEPROM im Durchschnitt nur einmal pro Woche verändert, wird die garantierte Anzahl der Schreibzyklen innerhalb der voraussichtlichen Verwendungszeit des Controllers sicherlich nicht erreicht werden. Welcher Controller ist schon 100000 / 52 = 1923 Jahre im Einsatz? In diesem Fall lohnt es sich daher nicht, erweiterte Programmfunktionen zu implementieren, mit denen die Anzahl der Schreibzugriffe minimiert wird.

Eine weitere Möglichkeit, Schreibzyklen einzusparen, besteht in der Vorabprüfung, ob der zu speichernde Wert im EEPROM bereits enthalten ist und nur veränderte Werte zu schreiben. In aktuelleren Versionen der avr-libc sind bereits Funktionen enthalten, die solche Prüfungen enthalten (eeprom_update_*).

Eine dritte Möglichkeit speichert alle Daten zunächst im RAM, wo sie beliebig oft beschrieben werden können. Nur beim Ausschalten oder beim Ausfall der Stromversorgung werden die Daten in den EEPROM geschrieben. Wie man das richtig macht sieht man im Artikel Speicher.

Lesezugriffe können beliebig oft durchgeführt werden. Sie unterliegen keinen Einschränkungen in Bezug auf deren Anzahl.

EEMEM

Um eine Variable im EEPROM anzulegen, stellt die avr-libc das Makro EEMEM zur Verfügung:

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

/* Byte */
uint8_t eeFooByte EEMEM = 123;

/* Wort */
uint16_t eeFooWord EEMEM = 12345;

/* float */
```

```
float eeFooFloat EEMEM;

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

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

Die grundsätzliche Vorgehensweise ist identisch zur Verwendung von PROGMEM. Auch hier erzeugt man sich spezielle attributierte Variablen (EEMEM erledigt das), die vom Compiler/Linker nicht wie normale Variablen behandelt werden. Compiler/Linker kümmern sich zwar darum, dass diesen Variablen eine Adresse zugewiesen wird, diese Adresse ist dann aber die Adresse der 'Variablen' im EEPROM. Um die dort gespeicherten Werte zu lesen bzw. zu schreiben, übergibt man diese Adresse an spezielle Funktionen, die die entsprechenden Werte aus dem EEPROM holen bzw. das EEPROM neu beschreiben.

Die mittels EEMEM erzeugten 'Variablen' sind also mehr als Platzhalter zu verstehen, denn als echte Variablen. Es geht nur darum, im C-Programm symbolische Namen zur Verfügung zu haben, anstatt mit echten EEPROM-Adressen hantieren zu müssen, etwas, das grundsätzlich aber auch genauso gut möglich ist. Nur muss man sich in diesem Fall dann selbst darum kümmern, dass mehrere 'Variablen' ohne Überschneidung im EEPROM angeordnet werden.

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:

```
#define EEPROM_DEF 0xFF

void eeprom_example (void)
{
    uint8_t myByte;

    // myByte lesen (Wert = 123)
    myByte = eeprom_read_byte (&eeFooByte);

    // der Wert 99 wird im EEPROM an die Adresse der
    // Variablen eeFooByte geschrieben
    myByte = 99;
    eeprom_write_byte(&eeFooByte, myByte); // schreiben

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

    // Beispiel fuer eeprom_update_byte: die EEPROM-Zelle wird nur
    // dann beschrieben, wenn deren Inhalt sich vom Parameterwert
    // unterscheidet. In diesem Beispiel erfolgt also kein Schreib-
    // zugriff, da die Werte gleich sind.
    eeprom_update_byte(&eeFooByte, myByte);


    // 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.

    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:

```
    // lesen
    uint16_t myWord = eeprom_read_word (&eeFooWord);

    // schreiben
    eeprom_write_word (&eeFooWord, 2222);
```

Block lesen/schreiben

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

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

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

    /* und nun mit 16-Bit Array */
    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));
}
```

Fließkommawerte lesen/schreiben

In der avr-libc stehen auch EEPROM-Funktionen für Variablen des Typs float (Fließkommazahlen mit "einfacher" Genauigkeit) zur Verfügung.

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

float eeFloat EEMEM = 12.34f;

float void eeprom_float_example(float value)
{
   /* float in EEPROM schreiben */
   eeprom_write_float(&eeFloat, value);

   /* float aus EEPROM lesen */
   return  eeprom_read_float(&eeFloat);
}
```

EEPROM-Speicherabbild in .eep-Datei

Mit den zum Compiler gehörenden Werkzeugen kann der aus den Variablendeklarationen abgeleitete EEPROM-Inhalt in eine Datei geschrieben werden. Die übliche Dateiendung ist .eep, Daten im Intel Hex-Format. Damit können Standardwerte für den EEPROM-Inhalt im Quellcode definiert werden.

Makefiles nach WinAVR/MFile-Vorlage enthalten bereits die notwendigen Einstellungen, siehe dazu die Erläuterungen im Exkurs Makefiles.

Der Inhalt der eep-Datei muss ebenfalls zum Mikrocontroller übertragen werden, 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[11] nicht die korrekten Werte:

EESAVE = 0 (programmed)
: Die Daten im EEPROM bleiben erhalten. Werden sie nicht neu geschrieben, so enthält das EEPROM evtl. Daten, die nicht mehr zum Programm passen.

EESAVE = 1 (unprogrammed)
: Beim Programmieren werden die Daten im EEPROM gelöscht, also auf 0xff gesetzt.

Als Sicherung kann man im Programm nochmals die Standardwerte vorhalten, beim Lesen auf 0xFF prüfen und gegebenenfalls einen Standardwert nutzen. Das geht natürlich nur, wenn 0xFF selbst nicht als Datenwert vorkommen kann.

```
#define DUTY_CYCLE_DEFAULT 0x80

uint8_t eeDutyCycle EEMEM;   // Platzhalter für EEPROM
uint8_t DutyCycle;           // die echte Variable


int main(void)
{
  DutyCycle = eeprom_read_byte( &eeDutyCycle );
  if( DutyCycle == 0xFF )                    // das allererste mal. Im EEPROM steht noch kein gültiger Wert
  {
    DutyCycle = DUTY_CYCLE_DEFAULT;
    eeprom_writeByte( &eeDutyCycle, DutyCycle );
  }

  ...
```

Direkter Zugriff auf EEPROM-Adressen

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

Hinweis! Die hier nachfolgend gezeigten Funktionen und Zugriffe auf absolute Adressen sind in Normalfall nicht nötig und nur auf sehr wenige, spezielle Fälle beschränkt! Im Normalfall sollte man auf absolute Adressen möglichst nicht zugreifen und den Compiler seine Arbeit machen lassen, der verwaltet die Variablen und deren Adressen meist besser als der Programmierer. Der Zugriff auf Variablen im EEPROM sollte immer über ihren Namen erfolgen.

```
#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
```

Was steckt dahinter? - EEPROM-Register

Auch wenn es normalerweise keinen Grund gibt, in C selbst an den Steuerregistern herumzuschrauben - die eeprom Funktionen erledigen das alles zuverlässig - der Vollständigkeit halber der registermässige technische Unterbau. 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.

Anmerkungen

1. Für eine Liste der unterstützten COntroller siehe die Dokumentation des Compilers oder AVR-Libc: Supported Devices.
2. Aktuelle, stabile Versionen sind als Nightly Builds regelmäßig im Forum verfügbar.
3. z. B. Ponyprog, yapp, AVRStudio
4. 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.
5. Variablen der Speicherklasse *static storage* haben eine unbegrenzte Lebensdauer. Beispiel für solche Variablen sind globale Variablen, aber auch static-Variablen innerhalb einer Funktion gehören dazu. Beispiele für Variablen, die nicht *static storage* sind: auto-Variablen ("normale" lokale Variablen), register-Variablen, durch malloc geschaffene Objekte, etc.
6. Damit ist der mögliche Speicherbereich für Flash-Konstanten auf 64kiB begrenzt. Einige pgmspace-Funktionen ermöglichen den Lesezugriff auf den gesamten Flash-Speicher, intern via Assembler-Anweisung ELPM. Die Initialisierungswerte des Speicherinhalts jenseits der 64kiB-Marke müssen dann jedoch auf anderem Weg angelegt werden, d.h. nicht per PROGMEM. Evtl. eigene Section und Linker-Optionen. Alt und nicht ganz korrekt: Die avr-libc pgmspace-Funktionen unterstützen nur die unteren 64kiB Flash bei Controllern mit mehr als 64kiB.
7. ab avr-libc 1.7.0
8. In unserem Hund-Katze-Maus Beispiel belegt die erste Variante 22 Bytes Daten und 18 Bytes Code, die zweite Variante mit 2-dimensionalem Array belegt 18 Bytes Daten und 20 Bytes Code. Gemessen wurde mit avr-gcc 4.8 -Os für ATmega8.
9. Im Gegensatz zu einem Attribute wie `progmem` ist ein (Adress Space) Qualifier Teil des Zeiger-Typs.
10. Binutils PR14406: Support .progmem<N>.data sections to work with GCC's PR49868.
11. vgl. Datenblatt Abschnitt Fuse Bits

TODO

- Aktualisierung Register- und Bitbeschreibungen an aktuelle AVR
- "naked"-Funktionen [2][3]

Kategorie: Avr-gcc Tutorial