# Einführung in die Mikrocontroller-Programmierung am Beispiel der ATMEL-AVR-Familie

**V. 20080430-791 Dipl.-Ing. J. Menzel**

# Inhaltsverzeichnis

# 0 Einführung

Das vorliegende Dokument soll Grundlagen der Embedded-C-Programmierung am Beispiel der AVR-Mikrocontroller der Firma ATMEL vermitteln und den Einstieg in die Anwendung von Mikrocontrollern erleichtern, indem es Informationen, die man sich sonst aus den unterschiedlichsten Quellen zusammensuchen müsste, kompakt und für Einsteiger verständlich darstellt. Dabei wird zunächst auf die Hardware des Controllers eingegangen. Aufbau und Funktion der für den Einstieg relevanten Baugruppen werden (weitgehend Programmiersprachen-unabhängig) erläutert, soweit es für das Verständnis erforderlich ist. Ziel ist es, die grundlegende Funktionsweise der Controller-Peripherie zu verstehen.

Der zweite Teil geht auf die Besonderheiten der Mikrocontroller-Programmierung mit der Programmiersprache C ein, sofern diese Controller-spezifisch und nicht vom ANSI-Standard abgedeckt sind. Kenntnisse der Programmiersprache C sind Voraussetzung und können im Rahmen dieser Einführung nicht vermittelt werden.

# 1 Microcontroller-Hardware

Grundsätzlich ist ein Mikrocontroller ($\mu$C, Mikrocomputer, Mikrorechner) ein Baustein, der im Unterschied zu einem Mikroprozessor ($\mu$P) neben einer CPU noch Speicher und diverse Peripheriekomponenten wie Schnittstellen, Timer, Analog-Digital-Wandler enthält, wodurch zum Betrieb in einer Anwendung nur sehr wenige zusätzliche Komponenten benötigt werden. Es existieren aktuell sehr viele sehr unterschiedliche Mikrocontroller-Familien von vielen Herstellern, die sich neben der Breite des internen Datenbus (8, 16 oder 32 Bit), der Rechenleistung (Befehle pro Sekunde, MIPS, MOPS bzw. MFLOPS) in ihrer Architektur (Harvard- oder von-Neumann-Architektur, z.T. auch Kombinationen aus beiden) stark unterscheiden können.

Die Auswahl eines Bausteins für eine spezielle Anwendung ist aufgrund der großen Vielfalt der erhältlichen Typen oft schwierig. Da es für jede $\mu$C-Familie unterschiedliche Programmier- und Entwicklungstools gibt, ist es meist sinnvoll, eine Familie auszuwählen, die ein möglichst breites Spektrum an Bausteinen für unterschiedlich komplexe Anwendungen bietet. Die in dieser Einführung beschriebene 8-bit-AVR-Familie von Atmel bietet bei sehr günstigen Preisen für Komponenten und Tools eine mittlerweile sehr große Auswahl an Bausteinen (u.a. auch USB-, CAN- und spezielle PWM-Controller) mit im Vergleich zu anderen 8-bit-$\mu$Cs hoher Rechenleistung (u.a. aufgrund der verwendeten Harvard-RISC-Architektur), wodurch sie auch für Hobbybastler sehr interessant ist.

## 1.1 Grundlegendes zur ATMega16-Hardware

Der in dieser Dokumentation beschriebene Baustein ATMega16 kann bis zu 16 Millionen Assembler-Befehle pro Sekunde (MIPS) bei einer Taktfrequenz von 16 MHz verarbeiten. Er besitzt einen Flash-Programmspeicher von 16 KiB[1], einen SRAM-Datenspeicher von 1 KiB und 512 Bytes EEPROM. Zur Peripherie-Ausstattung gehören 3 Timer/Zähler (ein 16-Bit, zwei 8-Bit), ein 10-Bit Analog-Digital-Wandler mit bis zu 8 Kanälen, ein Analog-Komparator, diverse serielle Schnittstellen sowie 32 I/O-Pins. Die I/O-Pins sind in vier 8-Bit-Ports eingeteilt und prinzipiell gleichwertig. Je nach verwendeter Peripherie bekommen einige von ihnen jedoch Sonderfunktionen zugeteilt, z.B. als Interrupt- oder Analog-Eingänge, Timer-Ein- und Ausgänge, serielle Interfaces usw.. Wie alle aktuellen AVRs besitzt der ATMega16 einen internen RC-Oszillator, der in Fällen, in denen keine großen Anforderungen an Frequenzbereich, -Genauigkeit und Temperaturstabilität gestellt werden, als Taktgeber verwendbar ist und einen externen Oszillator (Quarz) entbehrlich machen kann.

Allen Peripherie-Einheiten sind spezielle I/O-Register im SRAM-Speicher des Controllers zugeordnet, über die die Komponenten konfiguriert werden können bzw. über die

[1] Da Halbleiterspeicher binär adressiert werden, ergeben sich für die Speichergröße bei einem n Bit breiten Adressbus grundsätzlich $2^n$ Adressen (bei einem Byte-organisierten Speicher ist das gleichzeitig die Anzahl der Bytes). Für Potenzen von 2 existieren jedoch, anders als bei Zehnerpotenzen (Kilo [k], Mega [M], Giga [G], Tera [T]...) keine Faktoren im SI-Standard, weshalb empfohlen wird, die in der IEC 60027-2 vorgeschlagenen Kürzel zu verwenden, um Verwechslungen zu vermeiden. Die binären Faktoren sind nach Potenzen von $2^{10}$ (1024) gestaffelt. 1024 Bytes sind danach 1 KiBi-Byte ('**Ki**lo **Bi**nary-**Byte**', kurz KiB), $1024^2$ ($2^{20}$) Bytes 1 MeBi-Byte (MiB) usw.. Das KiB (1024 Bytes) wurde früher i.d.R. als KB abgekürzt (man achte auf das große 'K' zur Unterscheidung vom 'normalen' Kilo, das mit einem kleinen 'k' abgekürzt wird), was aber zu Verwechslungen führen kann und deshalb vermieden werden sollte.

der Datenaustausch mit der CPU stattfindet. Um auf bestimmte asynchron auftretende Hardware-Ereignisse schnellstmöglich reagieren zu können, besitzt der Controller eine Anzahl von Interrupt-Quellen, die bei entsprechender Konfiguration dafür sorgen, dass das ablaufende Programm sofort unterbrochen und ein Unterprogramm (sog. Interrupt Handler oder Interrupt Subroutine, ISR) aufgerufen wird.

Im Folgenden werden die wichtigsten Komponenten des ATMega16 kurz vorgestellt. Weiterführende Informationen sind der Literatur zu entnehmen. In erster Linie sollen diese Abschnitte dazu dienen, Inhalte des Controller-Handbuchs leicht verständlich darzustellen, um die Einarbeitung zu erleichtern.

## 1.2 Speicher und Programmausführung

Der ATMega16 enthält vier unterschiedliche Arten von Speicher:

- Programmspeicher (Flash-EPROM)
- flüchtigen Datenspeicher (SRAM)
- nichtflüchtigen Datenspeicher (EEPROM)
- Rechenregistersatz (flüchtig)

### 1.2.1 Flash-Speicher

Der Flash-Speicher ist zur Speicherung des Programmcodes und von (konstanten) Daten vorgesehen. Der im Flash untergebrachte Programmcode wird durch einen Befehls- oder Programmzähler (Program Counter, PC) adressiert. Dieser Zähler wird normalerweise nach jedem Befehl um die Länge des betreffenden Befehls inkrementiert, um den jeweils nächsten Befehl zu laden. Er kann jedoch durch das Programm einen anderen Wert zugewiesen bekommen, z.B. durch einen Sprungbefehl. Dadurch sind bedingte Verarbeitung und Unterprogrammaufrufe möglich. Am Anfang des Programmspeichers (Adresse 0) steht der Reset-Vektor, gefolgt von den Interrupt-Vektoren[2]. Wird der Controller aktiviert oder durch einen Hardware-Reset in den Ausgangszustand versetzt, so hat der Programmzähler zunächst den Wert '0', so dass der im Reset-Vektor stehende Befehl ausgeführt wird. Dieser ist i.A. ein Sprungbefehl auf den Beginn des eigentlichen Programms. Die Interrupt-Vektoren enthalten die Sprungadressen zu den entsprechenden Interrupt-Handlern. Für jeden Interrupt existiert ein solcher Vektor. Aufgrund der Tatsache, dass die Befehle des AVR 16 oder 32 Bit breit sind, ist der Flash-Speicher 'Wort-organisiert', d.h. eine Adresse spricht immer zwei Bytes an. Zu beachten ist, dass bei den AVR-Controllern aufgrund der Harvard-Architektur, deren hauptsächlicher Unterschied zur von-Neumann-Architektur (die z.B. bei der 8051er-Controller-Familie Anwendung findet) in den getrennten Bussen für Programm (Befehle) und Daten besteht, der Programmspeicher nicht erweiterbar ist, da der Befehls-Bus nicht von außen zugänglich ist. Der Datenbus hingegen ist bei einigen Vertretern der AVR-Familie (nicht jedoch beim ATMega16) zugänglich und dementsprechend ist es möglich, extern zusätzlichen Datenspeicher vorzusehen, dessen Größe lediglich durch den Adressraum des Controllers auf max. 64 KiB limitiert ist.

[2]Es ist zu beachten, dass die Interrupt-Vektoren bei den AVR-Controllern mit bis zu 8 KiB Flash-Speicher 16 Bit (1 Wort) breit sind, während bei AVRs mit mehr als 8 KiB Flash (also auch beim ATMega16) aufgrund des erweiterten Adressbereiches 32 Bit (2 Worte) pro Vektor vorgesehen sind.

### 1.2.2 Rechenregistersatz

Der Controller besitzt einen Satz von 32 (fast) gleichberechtigten Rechenregistern[3], deren Adressraum in die 32 niedrigsten Adressen des SRAM gemappt ist. Die für die eigentliche Rechenarbeit zuständige Arithmetical Logical Unit (ALU) kann auf diese Register direkt zugreifen, was dazu führt, dass Rechenoperationen in einem Zyklus bzw. mit einem einzigen Befehl durchgeführt werden können. Diese Architektur ist einer der Gründe für die große Flexibilität und den hohen Befehlsdurchsatz der AVR-Controller. Andere Controller (z.B. die weit verbreiteten 8051er) besitzen lediglich einen sogenannten Akkumulator als einziges echtes Rechenregister, wodurch sich in den seltensten Fällen selbst einfache Rechenoperationen in einem Befehl durchführen lassen. Bei Hochsprachen-Programmierung versucht der Linker i.d.R., bei der Allokierung von Variablen, möglichst viele von ihnen im Registersatz unterzubringen, um die Bearbeitungsgeschwindigkeit zu maximieren.

### 1.2.3 SRAM

Das SRAM dient der Speicherung von Daten (Variablen), die während der Ausführung des Programms anfallen. Außerdem befindet sich im SRAM (i.d.R. am Ende des Speichers) der Stack, der in erster Linie der Speicherung von Rücksprungadressen beim Aufruf von Unterprogrammen dient. Der Stack ist ein LIFO- (Last In First Out-) oder Keller-Speicher, d.h. es kann nur auf die Komponente zugegriffen werden, die 'oben' liegt. Bei einem Unterprogrammaufruf durch das Programm oder durch einen Interrupt wird die Adresse des gerade ausgeführten Befehls (also der Stand des Programmzählers) auf dem Stack abgelegt ('push'). Bei Beendigung des Unterprogramms erhält der Programmzähler die auf dem Stack liegende Adresse zugewiesen ('pop') und wird anschließend um eins erhöht. Dies ermöglicht auch geschachtelte Unterprogrammaufrufe, wobei allerdings zu beachten ist, dass der Stack mit der Schachtelungstiefe dynamisch von hinten in das SRAM 'hineinwächst'. Dabei nimmt er keine Rücksicht auf eventuell dort gespeicherte Variablen. Die Schachtelungstiefe von Unterprogrammen, v.a. bei der Interrupt-Bearbeitung (s. Abschnitt 1.9), ist also möglichst zu begrenzen, zumal es bei Interrupt-Unterprogrammen im Unterschied zu normalen Unterprogrammen keine Warnungen des Compilers bzw. Linkers im Falle eines Stack-Überlaufes gibt, da Interrupts Hardware-Ereignisse sind, die sich der Kontrolle durch die Programmiersoftware entziehen. Neben der Speicherung von Rücksprungadressen wird der Stack auch vom Compiler zum Ablegen lokaler Variablen genutzt.

### 1.2.4 I/O-Register

Die nach den Rechenregistern niedrigsten Adressen des SRAM sind mit den I/O-Registern belegt. Diese Register dienen der Kommunikation der CPU mit den Peripheriekomponenten. Dabei existieren einerseits Steuerregister, die Konfigurationsdaten enthalten, und andererseits Datenregister. Die 32 I/O-Register mit den niedrigsten Adressen sind bitadressierbar, d.h. mit bestimmten (Assembler-) Befehlen lassen sich die Bits einzeln ansprechen.

Eines dieser I/O-Register, das Statusregister `SREG`, ist durch seine Aufgaben besonders hervorzuheben. Es enthält eine Reihe von sogenannten Status-Flags, also Bits, die zum

[3]Es gibt einige Unterschiede, was Zugriffe für bestimmte Befehle angeht. Außerdem besitzen einige der Rechenregister Sonderfunktionen z.B. für Zeiger-Arithmetik.

Anzeigen und Bearbeiten bestimmter Vorgänge bzw. Ereignisse auf Hardware-Ebene benötigt werden. Das Carry- ('Übertrag'-) Flag `C` zeigt z.B. an, ob die letzte Rechenoperation einen Überlauf zur Folge hatte, während das Zero-Flag `Z` gesetzt wird, wenn das Ergebnis der letzten Operation Null war. Diese Flags sind elementar wichtig für die Mehrzahl der arithmetischen Operationen, da nur durch sie eine bedingte Verarbeitung (Verzweigungen, Entscheidungen) sowie die Verarbeitung von Daten, die mehr als ein Register in Anspruch nehmen, möglich ist. Ein anderes wichtiges Bit in diesem Register ist das Interrupt-Enable-(`I`-) Bit, das dazu dient, die Bearbeitung von Interrupts (siehe Abschnitt 1.9 auf Seite 16) global freizugeben bzw. zu sperren.

Einige Peripherie-Einheiten (16-Bit-Timer, ADC) arbeiten mit Daten, die nicht in einem einzelnen Register Platz finden. Diese Daten sind dann entsprechend auf zwei Register aufgeteilt (Low- und High-Byte). Da Zugriffe auf mehrere Register auch immer mehrere Zyklen in Anspruch nehmen, kann es in solchen Fällen geschehen, dass sich der Inhalt des Doppelregisters während des Zugriffs ändert (z.B. Timer-Zählregister wird inkrementiert oder ADC beendet nächste Wandlung) und so Low- und High-Byte nicht zusammenpassen. Um sicherzustellen, dass Low- und High-Byte auch wirklich zusammengehören, besitzen die meisten dieser Doppelregister (das trifft auf die Zähl-, Compare- und Capture-Register des 16-Bit-Timers zu) eine Pufferung. Dabei wird der eigentliche Schreib- oder Lesevorgang jeweils durch den Zugriff auf das Low-Byte getriggert, während das High-Byte in einem temporären 'Schatten'-Register (also einem Register, auf das ansonsten kein direkter Zugriff besteht) abgelegt wird. Es ergeben sich dadurch folgende Abläufe:

- **Schreibzugriff:** Ein Schreibzugriff auf das High-Byte führt dazu, dass der Wert zunächst im temporären Register abgelegt wird. Das Schreiben des Low-Bytes führt zur synchronen Übernahme beider Werte in die Register.
- **Lesezugriff:** Beim Lesezugriff auf das Low-Byte wird zeitgleich das High-Byte in das temporäre Register übernommen. Ein darauf folgender Lesezugriff auf das High-Byte liefert den Wert des temporären Registers.

Die sich ergebende Reihenfolge der Zugriffe auf die 16-Bit-Doppelregister ist unbedingt zu beachten. **Regel: Beim Schreiben zuerst das High-Byte, beim Lesen zuerst das Low-Byte!**

Konflikte können sich allerdings trotz der Pufferung ergeben, da für alle 16-Bit-Register nur ein einziges temporäres Register existiert. Tritt zwischen den Zugriffen auf Low- und High-Byte ein Interrupt auf, in dessen Handler ebenfalls ein Doppelregister manipuliert wird, dann kommt es zu Datenverlust. Um das zu verhindern, sollte während des Zugriffs auf die Doppelregister die Interrupt-Bearbeitung gesperrt sein.

Bei den Datenregistern des Analog-Digital-Wandlers (`ADCL` und `ADCH`) existiert kein temporäres Register. Hier geschieht die Synchronisierung des Zugriffs dadurch, dass mit dem Lesezugriff auf `ADCL` beide Register so lange gegen einen Schreibzugriff durch die Hardware gesperrt werden, bis `ADCH` gelesen wurde. Beendet der ADC eine Wandlung nach einem Zugriff auf `ADCL`, aber vor dem entsprechenden Zugriff auf `ADCH`, dann geht das Ergebnis dieser Wandlung verloren, da es nicht gespeichert werden kann.

**Anmerkung:** Bei der Programmierung in Hochsprachen (z.B. C) sind für diese Doppelregister i.d.R. in den entsprechenden Bibliotheken 16-Bit-Register definiert, so dass der Programmierer sich keine Gedanken um die Zugriffsreihenfolge machen muss (s.a. Abschnitt 2.7 auf Seite 26).

### 1.2.5 EEPROM

Zur Speicherung von Daten, die nur selten verändert werden und die auch nach dem Abschalten der Spannungsversorgung erhalten bleiben sollen (z.B. Konfigurationsparameter o.ä.), steht beim ATMega16 ein 512 Bytes umfassendes EEPROM zur Verfügung. Da das EEPROM, ähnlich wie der Flash-Speicher, nur eine begrenzte Anzahl von Lösch-Schreib-Zyklen zulässt[4], ist bei der Benutzung darauf zu achten, dass Änderungen an gespeicherten Daten nur dann vorgenommen werden, wenn es nötig ist.

## 1.3 Betriebsarten des Controllers

Neben dem Standard-Betrieb (Ausführung eines Programms) existiert eine Reihe von 'Sleep'-Modi, in denen stufenweise Peripherie-Komponenten und schließlich die CPU selbst außer Betrieb genommen werden können, um die Stromaufnahme des $\mu$C zu senken, wenn gerade keine Aktionen auszuführen sind, was v.a. für Anwendungen mit batteriegestützter Spannungsversorgung interessant ist. Je nach eingestelltem Sleep-Modus existieren unterschiedliche Möglichkeiten, den Controller wieder 'aufzuwecken', und zwar umso eingeschränktere, je mehr Peripherie deaktiviert wurde. Aus dem Powerdown-Modus, bei dem auch der Hauptoszillator abgeschaltet ist, ist es z.B. nur über einen externen Pegel- (Level-) Interrupt möglich, mit der Programmausführung fortzufahren. In diesem Modus ist allerdings auch die Stromaufnahme am geringsten. In allen Sleep-Modi bleiben die Werte in den flüchtigen Speichern (das gilt selbstverständlich auch für die I/O-Register und den Programmzähler) erhalten.

Zum Aktivieren eines Sleep-Modus muss das Steuerregister `MCUCR` entsprechend konfiguriert und anschließend der Assembler-Befehl `sleep` ausgeführt werden. Nach dem 'Aufwachen' wird das Programm bei dem dem `sleep` folgenden Befehl fortgesetzt, nachdem der Handler des zum Aufwecken verwendeten Interrupt abgearbeitet ist. Ein 'Schlafenlegen' des Controllers ist also prinzipiell an jeder Stelle im Programm möglich, wobei allerdings selbstverständlich an der betreffenden Stelle der zum Aufwecken vorgesehene Interrupt lokal `und` global freigegeben sein muss.

## 1.4 I/O-Ports

Die in vier 8-Bit-Ports aufgeteilten I/O-Pins des ATMega16 besitzen symmetrische Treiberstufen, die dafür sorgen, dass ein als Ausgang geschalteter Pin sowohl High als auch Low einen Strom von mehr als 20 mA treiben kann (kurzzeitig max. 40 mA). Dadurch ist es möglich, Verbraucher wie z.B. LEDs direkt am Portpin zu betreiben. Dabei sind allerdings die im Datenblatt angegebenen Grenzwerte für die Strombelastung der einzelnen Ports und der Spannungsversorgungsanschlüsse einzuhalten. Ein als Eingang programmierter Pin kann hochohmig (undefiniertes Potenzial bei offenem Pin) oder mit internem Pull-Up-Widerstand (HIGH-Pegel bei offenem Pin) arbeiten, was das Anschließen z.B. von Tastern vereinfacht. Die Ports B-D sind im internen Aufbau weitestgehend identisch, während Port A durch die Sonderfunktion als Analog-Eingangsport anders verschaltet ist. Die Ausgangstreiber von Port A besitzen zusammen mit dem A/D-Wandler eine separate Spannungsversorgung AVCC. Ist diese korrekt angeschlossen, so ist der Port für die

[4] min. 100.000 nach Hersteller-Angaben

Programmierung den anderen gleichwertig. Es ist aber darauf zu achten, dass bei Benutzung eines oder mehrerer Analog-Eingänge möglichst kein Pin an diesem Port als digitaler Ausgang benutzt wird, da ein Umschalten während einer A/D-Wandlung das Wandlungsergebnis verfälschen kann.

| DDRx.n | PORTx.n | Funktion |
|---|---|---|
| 0 | 0 | Eingang, 'Tristate' (hochohmig) |
| 0 | 1 | Eingang mit Pull-Up |
| 1 | 0 | Ausgang, niederohmig gegen GND |
| 1 | 1 | Ausgang, niederohmig gegen VCC |

**Tabelle 1:** Initialisierung der I/O-Ports

Jedem Port sind zwei Steuerregister zugeordnet, ein Portregister `PORTx` (wobei x für den Namen des Ports steht, also A, B, C oder D), das direkt auf die Ausgangstreiber zugreift, und ein Datenrichtungsregister `DDRx`, dessen Wert für jeden einzelnen Pin entscheidet, ob er als Eingang oder als Ausgang behandelt wird. Tabelle 1 zeigt die Einstellungen für die Ports. Zusätzlich existiert noch ein Register `PINx`, über das die aktuellen Zustände an den Pins von Port x abgefragt werden können. Bei der Abfrage von Eingangszuständen ist es **wichtig**, dass man `PINx` einliest und **nicht** `PORTx`, da letzteres nur den Schaltzustand der Ausgangsstufe enthält, der von außen nicht beeinflussbar ist! Abgesehen davon ist zu beachten, dass durch Setzen des Bits `PUD` (Pull Up Disable) im Register `SFIOR` die internen Pull-Ups an allen I/O-Ports deaktiviert werden.

Grundsätzlich können alle Portpins als allgemeine digitale Ein- bzw. Ausgänge genutzt werden. Bei der Verwendung von Peripherie-Komponenten oder bestimmten Controller-Funktionen erhalten viele Pins Sonderfunktionen. In einigen Fällen wird durch die Aktivierung einer Sonderfunktion der direkte Zugriff auf den Pin durch die Software über die Portregister blockiert (z.B. bei den Daten-Aus- und Eingängen von Schnittstellen), bei anderen ist dies jedoch nicht der Fall (z.B. bei Timer-Compare-Ausgängen). Im zweiten Fall kann es bei Schreibzugriffen auf das betreffende Portregister zu unerwünschten Effekten kommen, deren mögliche Folgen im Voraus abgeschätzt werden sollten. Eine Minimierung der Gefahr asynchroner Änderungen lässt sich mit vertretbarem Aufwand erreichen, indem man unmittelbar vor dem Schreibzugriff auf den Port den Zustand der betreffenden Ausgangspins einliest und diese dann beim Schreiben entsprechend ausmaskiert. Jedoch ist auch diese Vorgehensweise nicht sicher, da die Hardware, die die Kontrolle über den Pin haben soll (also z.B. ein Timer), zwischen Lese- und Schreibzugriff den Pinzustand ändern kann. Die Wahrscheinlichkeit dafür wächst mit der Häufigkeit der Zugriffe sowohl seitens der Hardware (z.B. Timer-Frequenz) als auch seitens der Software. Die Belegung der einzelnen Pins ist dem Controller-Handbuch zu entnehmen.

## 1.5 Timer / Zähler

Ein Timer/Zähler besteht prinzipiell aus einem Zählregister und einer Taktquelle, wobei letztere je nach Anwendungsfall ein Taktsignal mit konstanter Frequenz (Betriebsart Timer) oder ein asynchrones Signal (Betriebsart Ereigniszähler) zur Verfügung stellt, deren Flanken das Zählregister inkrementieren. Die Timer/Zähler des ATMega16 besitzen als Taktquelle einerseits Vorteiler (Prescaler), die die CPU-Frequenz in unterschiedlichen

Verhältnissen heruntergeteilt zur Verfügung stellen, andererseits kann ein Takt über einen entsprechenden I/O-Pin (Tn) angelegt werden. Der 8-Bit-Timer 2 besitzt die Möglichkeit der Taktung über einen zusätzlichen Quarz mit geringerer Frequenz (ausgelegt für einen 'Uhrenquarz' mit 32,768 kHz), um eine asynchrone 'Echtzeituhr' zu implementieren. Bei einem Überlauf des Zählregisters (also beim Übergang vom Maximalwert zu Null, wobei der Maximalwert bei einem 8-Bit-Timer $FF_h$ und bei einem 16-Bit-Timer $FFFF_h$ beträgt) wird das zugehörige Overflow-Flag gesetzt, das entweder per Software abgefragt oder zum Auslösen eines Interrupt genutzt werden kann.

Die Timer/Zähler sind 'autonome' Blöcke aus Logik-Komponenten auf dem Chip und können nach entsprechender Initialisierung durch die Software unabhängig arbeiten. Dadurch ergibt sich die Möglichkeit, sehr präzise Signale zu erzeugen bzw. zu erfassen. Die Kommunikation zwischen CPU und Timer erfolgt über die entsprechenden Steuer- und Datenregister im I/O-Bereich bzw. über Interrupt-Flags.

### 1.5.1 Compare-Einheiten und PWM

Zur Erzeugung unterschiedlicher Timing-Sequenzen bzw. Signalformen besitzen die Timer Compare-Einheiten (beim ATMega16 eine pro 8-Bit-Timer und zwei beim 16-Bit Timer), die aus jeweils einem Compare-Register und einem Digitalkomparator bestehen. Der im Zählregister des Timers befindliche Wert wird permanent mit dem Vergleichswert im Compare-Register verglichen. Erreicht das Zählregister diesen Vergleichswert, wird beim nächsten Timer-Takt ein Bit (Flag) im entsprechenden Steuerregister gesetzt. Dieses Flag kann zum Auslösen eines Interrupts genutzt werden. Es ist aber auch möglich, das Flag per Software abzufragen. Zusätzlich kann (vom Programmablauf völlig unabhängig) der Zustand eines Portpins geändert werden, sobald ein Compare-Ereignis eintritt.

Mit den Compare-Einheiten lassen sich neben dem 'normalen' Betrieb noch zwei Grund-Betriebsarten realisieren:

- **CTC (Clear Timer on Compare Match)**: In dieser Betriebsart wird das Zählregister des Timers bei einem Compare-Ereignis zurückgesetzt. Diese Betriebsart ist v.a. zur Erzeugung von Takten mit präzise einstellbarer Frequenz nutzbar.
- **PWM (Pulse Width Modulation)**: Die PWM-Betriebsarten dienen der Generierung von pulsweitenmodulierten Signalen, also Signalen mit sich änderndem Tastverhältnis bei i.d.R. konstanter Frequenz. Dabei wird grundsätzlich zwischen zwei Verfahren unterschieden:
  - *Fast PWM* (schnelle PWM, entspricht einer analogen PWM mit einem Sägezahn als Vergleichsspannung): Der Timer zählt aufwärts; beim Compare Match wird der dazugehörige Output Compare-Pin (OCnx) umgeschaltet und bei Erreichen des Maximalwertes (TOP), ähnlich wie in der CTC-Betriebsart, der Timer zurückgesetzt und der Output Compare-Pin wieder in den ursprünglichen Zustand versetzt.
  - *Phase Correct PWM* (phasenrichtige PWM, entspricht einer analogen PWM mit einer dreieckförmigen Vergleichsspannung): Der Timer zählt zunächst aufwärts; beim Compare-Match wird der dazugehörige Output Compare-Pin umgeschaltet; erreicht der Zählerstand TOP, wird die Zählrichtung umgekehrt;

wird beim Abwärtszählen wieder der Compare-Wert erreicht, schaltet der entsprechende OCnx-Pin wieder in den Ausgangszustand.

Bei gleichen TOP-Werten besitzt die Fast PWM gegenüber der Phase Correct PWM die doppelte Frequenz. Die Phase Correct PWM kommt in erster Linie dort zur Anwendung, wo Schalttotzeiten zu generieren sind. Bei Benutzung beider PWM-Kanäle des 16-Bit-Timers z.B. zur Ansteuerung einer Halbbrücke ist es durch Wahl unterschiedlicher Compare-Werte möglich, den Zeitraum zwischen dem Abschalten des einen und dem Einschalten des jeweils anderen Transistors einzustellen.

In den PWM-Betriebsarten werden Schreibzugriffe auf die Compare-Register von der Hardware über Pufferregister synchronisiert. Dadurch wird verhindert, dass ein Compare-Ereignis ausbleibt, wenn der neue Compare-Wert beim aufwärtszählenden Timer kleiner (bzw. beim abwärtszählenden Timer größer) ist als der aktuelle Zählerstand, was zu Unregelmäßigkeiten in der Signalform führen würde. Um dies zu vermeiden, findet die Aktualisierung der Compare-Werte immer erst beim nächsten Erreichen von TOP oder BOTTOM (je nach Betriebsart) statt.

In den nicht-PWM-Betriebsarten ist diese Synchronisierung deaktiviert, so dass es dort erforderlich ist, per Software für ein synchronisiertes Schreiben der Compare-Werte zu sorgen. Insbesondere in der CTC-Betriebsart kann ein fehlendes Compare Match unangenehme Folgen haben, da der Timer in solch einem Fall bis zum Überlauf weiterzählt und erst im nächsten Durchgang wieder ein Compare-Ereignis auftreten kann. Eine Möglichkeit, solche Probleme zu vermeiden ist eine Abfrage beim Setzen des neuen Compare-Wertes, ob `TCNTx` kleiner ist als der neue Compare-Wert. Ist dies nicht der Fall, dann ist `TCNTx` auf einen Wert zu setzen, der mindestens um eins kleiner ist als der neue Compare-Wert[5].

In den PWM-Modi bestehen je nach verwendetem Timer mehrere Möglichkeiten zur Festlegung des TOP-Wertes, der entweder ein fest eingestellter Wert sein kann (8-, 9- oder 10-Bit PWM mit `0xFF`, `0x1FF` oder `0x3FF` als TOP, wobei die beiden letztgenannten selbstverständlich nur bei 16-Bit-Timern auswählbar sind) oder ein frei definierbarer Wert in einem Compare-Register bzw. (bei Timern mit Input Capture Unit) dem Input Capture-Register. Die Betriebsarten, in denen TOP nicht durch eines der Compare-Register definiert wird, eignen sich bei Timern mit mehreren Compare-Einheiten zur Erzeugung mehrerer synchroner PWM-Signale.

Die Frequenz des erzeugten Signals ergibt sich in den CTC-Betriebsarten zu $f_{Comp} = \frac{f_{CPU}}{Prescaler \cdot (OCRnX+1)}$ mit `OCRnx` als Wert des betreffenden Compare-Registers. In den PWM-Betriebsarten ist `OCRnx` durch den entsprechenden TOP-Wert zu ersetzen. Der Zusammenhang zwischen Compare- bzw. TOP-Wert und Frequenz ist also reziprok und damit stark nichtlinear, was bei vielen Anwendungen zu beachten ist. Das '`OCRnX + 1`' in der Formel ergibt sich aus der Tatsache, dass, wenn die Gleichheit der Werte in `TCNTn` und `OCRnX` festgestellt ist, erst im darauf folgenden Timertakt die entsprechenden Aktionen durchgeführt werden (z.B. Interrupt-Flag setzen, Portpin umschalten). Wäre das nicht der Fall, dann würde ein Compare-Wert von Null einen undefinierten Zustand verursachen.

Die CTC-Betriebsart eignet sich u.a. zur Implementierung eines programmierbaren Frequenzteilers. Zu diesem Zweck ist der zur betreffenden Compare-Einheit gehörende Ausgangspin (OCnx) so zu konfigurieren, dass er bei einem Compare-Ereignis umgeschaltet

[5] Setzt man `TCNTx` exakt auf den Compare-Wert, dann tritt das selbe Problem auf, da ein Schreibzugriff auf `TCNTx` die Compare-Unit für den jeweils folgenden Timer-Takt sperrt.

('getoggelt') wird. Da eine Periode eines Rechteck-Signals aus zwei Umschaltvorgängen (Flanken) besteht, halbiert sich die Frequenz allerdings zusätzlich. Die höchste Frequenz, die mit einem solchen Frequenzteiler ausgegeben werden kann, ist also $f_{CT}/2$, wobei $f_{CT}$ die Taktfrequenz des Timers ist, die entweder vom CPU-Takt abgeleitet ist (evtl. über den Prescaler) oder aus einer externen Quelle stammen kann.

### 1.5.2 Input Capture

Zur Erfassung der Dauer externer Ereignisse stellt der 16-Bit Timer 1 eine Input-Capture-Einheit zur Verfügung. Diese besitzt ein spezielles Register, in das beim Auftreten einer Flanke am Input-Capture-Pin (oder bei entsprechender Konfiguration am Ausgang des Analog-Komparators, s. Abschnitt 1.7 auf Seite 14) der Stand des Timer-Zählregisters verzögerungsfrei übernommen werden kann. Dadurch sind z.B. hochgenaue Frequenz-, Periodendauer- oder Impulslängenmessungen möglich. Auch auf ein Capture-Ereignis kann ein Interrupt ausgelöst werden.

Bei Zeitmessungen mittels Input Capture ist es üblich, den Timer kontinuierlich laufen zu lassen und die Zeit zwischen den Capture-Ereignissen durch Bildung der Differenz zweier aufeinander folgender Capture-Werte zu ermitteln. Durch geeignete Wahl der Capture-Flanken ist es möglich, eine Periodendauer (Zeit zwischen zwei gleichen Flanken), eine Impulsdauer (Zeit zwischen zwei unterschiedlichen Flanken) oder beides kombiniert zu erfassen. Die Auswertung der Capture-Werte geschieht sinnvollerweise interruptgesteuert, wobei aufgrund der verzögerungsfreien Übernahme des Zählerstandes in das Capture-Register ICR1 (bzw. ICR1H und ICR1L) der Zeitverzug bis zur Auswertung im Interrupt-Handler keine Rolle spielt. Es muss lediglich gewährleistet sein, dass der Capture-Wert bis zum nächsten Capture-Ereignis gesichert wird.

## 1.6 Analog-Digital-Wandler

Der 10-Bit-Analog-Digital-Wandler des ATMega16 arbeitet nach dem 'Successive Approximation'-Prinzip (Wägeverfahren). Je nach Konfiguration können bis zu 15000 Abtastungen pro Sekunde bei voller Auflösung erreicht werden. Falls nur 8 Bit Auflösung erforderlich sind, können erheblich höhere Samplingraten erreicht werden. Bis zu 8 Eingänge (Kanäle) können über einen Multiplexer nacheinander eingelesen werden[6]. Es besteht die Möglichkeit, einige der Kanäle paarweise zum Einlesen von Differenz-Werten zu konfigurieren. Die Wandlungen können zyklisch (automatisch), durch eine der acht einstellbaren Triggerquellen (ADC-Auto-Trigger bzw. Free-Running-Modus) oder per Software einzeln gestartet werden. Der A/D-Wandler besitzt ein Interrupt-Flag, das gesetzt wird, sobald eine Wandlung abgeschlossen ist.

Als Referenzspannung für die Wandlung stehen unterschiedliche Quellen zur Verfügung, die je nach Anwendungsfall durch Setzen der `REFSn`-Bits im Register `ADMUX` selektierbar sind. Zur Auswahl stehen die Betriebsspannung des Analogteils (AVCC), eine intern erzeugte Referenzspannung von 2,56 V (die allerdings mit einer angegebenen Toleranz von mehr als ±10 % für viele Messaufgaben zu ungenau ist) sowie eine externe, an den Pin AREF anschließbare Referenz. Es ist darauf zu achten, dass im Falle der Verwendung

[6]Die betreffenden Portpins (beim ATMega16 die Pins von PORTA) sind zu diesem Zweck als hochohmige Eingänge zu konfigurieren.

von interner 2,56 V-Referenz oder AVCC der AREF-Pin bis auf einen Kondensator zur Störunterdrückung unbeschaltet bleiben sollte. Eine andere Beschaltung des Pins sollte ausschließlich dann erfolgen, wenn eine externe Referenzspannung zum Einsatz kommen soll, die der Controller intern nicht zur Verfügung stellt. Auf jeden Fall sollte AREF nie mit AVCC oder VCC verbunden werden! Die Referenzspannung muss beim ATMega16/32 mindestens 2 V betragen. Bei einigen neueren AVRs können auch geringere Spannungen verwendet werden.

Grundsätzlich wird eine Wandlung durch das Setzen des Bits `ADSC` (**AD**C **S**tart **C**onversion) im Steuerregister `ADCSRA` gestartet, was im Auto-Trigger- bzw. Free-Running-Modus durch die Hardware geschieht (außer bei der jeweils ersten Wandlung). Dieses Bit bleibt bis zum Ende der Wandlung gesetzt und wird erst bei Vorliegen des Ergebnisses gelöscht. Durch Abfragen von `ADSC` ist es möglich, das Ende der Wandlung in einer Schleife abzuwarten (Polling) und anschließend das Ergebnis auszuwerten. In den meisten Fällen wird man jedoch den Interrupt benutzen, damit das Programm während der Wandlung, die i.d.R. einige 10 $\mu$s dauert[7], weiterlaufen kann.

Nach der Wandlung liegt das Ergebnis in den Registern `ADCL` und `ADCH` vor. Beim Auslesen der Ergebnisregister `ADCL` und `ADCH` muss unbedingt erst das Low- und dann das High-Byte ausgelesen werden, da bei einem Lesezugriff auf `ADCL` beide Register so lange für die Aktualisierung durch die Hardware gesperrt werden, bis `ADCH` gelesen wurde. Benötigt man nur 8 Bit Auflösung für das Wandlungsergebnis, dann kann man das Bit `ADLAR` (ADC Left Adjust Result) im `ADMUX`-Register setzen. Dieses sorgt dafür, dass das Ergebnis der Wandlung 'linksbündig' ausgerichtet wird und die 8 MSB im `ADCH` gespeichert werden. Dann genügt es, nach einer Wandlung nur das `ADCH` zu lesen. `ADCL` kann dann unberührt bleiben.

ADC-Datenregister `ADCH` und `ADCL`, `ADLAR` nicht gesetzt:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **ADCH** | - | - | - | - | - | - | ADC9 | ADC8 |
| **ADCL** | ADC7 | ADC6 | ADC5 | ADC4 | ADC3 | ADC2 | ADC1 | ADC0 |

ADC-Datenregister `ADCH` und `ADCL`, `ADLAR` gesetzt:

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **ADCH** | ADC9 | ADC8 | ADC7 | ADC6 | ADC5 | ADC4 | ADC3 | ADC2 |
| **ADCL** | ADC1 | ADC0 | - | - | - | - | - | - |

Außerdem ist zu beachten, dass nach dem Aktivieren des ADC durch Setzen des Bits `ADEN` in `ADCSRA` die erste Wandlung u.U. kein brauchbares Ergebnis liefert. Es ist deshalb sinnvoll, den ADC zu Beginn des Programms eine 'Dummy'-Wandlung durchführen zu lassen und das Ergebnis dieser Wandlung zu verwerfen. Um bessere Ergebnisse zu erzielen, bietet es sich bei Anwendungen, die nicht auf eine hohe Abtastrate angewiesen sind, an, eine Mittelwertbildung über mehrere Wandlungen durchzuführen. Dadurch ist es möglich, Störungen wie Spannungsspitzen weitgehend zu eliminieren. Die Anzahl der Einzelmessungen für die Mittelwertbildung sollte eine Potenz von 2 sein, um die abschließende Division zu vereinfachen (zur Erläuterung siehe Abschnitt 2.9 auf Seite 31).

Wie die oben beschriebenen Timer ist auch der A/D-Wandler eine unabhängige Peripherie-Komponente, die nur über die zugeordneten I/O-Register bzw. den Interrupt mit der CPU kommuniziert.

[7] Eine normale Wandlung benötigt 13 ADC-Zyklen, also bei einer ADC-Taktfrequenz von 125 kHz 104 $\mu$s, die jeweils erste Wandlung nach dem Aktivieren des ADC dauert 25 Taktzyklen.

## 1.7 Analog-Komparator

Für einfachere Auswertungen analoger Eingangssignale (z.B. Schwellwertüberwachung) besitzt der ATMega16 einen Analog-Komparator. Diesem sind zwei Eingangspins zugeordnet, die mit dem nichtinvertierenden (AIN0) bzw. dem invertierenden (AIN1) Eingang des Komparators verbunden werden können. Wie die Eingangspins des A/D-Wandlers sind auch AIN0 und AIN1 über die Portregister als Eingänge ohne Pull-Up zu konfigurieren, um sie als Analog-Eingänge nutzen zu können.

Alternativ zur Standardkonfiguration, bei der beide Eingänge des Komparators mit externen Signalen versorgt werden, ist es auch möglich, die vom Controller intern zur Verfügung gestellte Bandgap-Referenz (1,23 V) mit dem nichtinvertierenden Eingang zu verbinden. Außerdem ist es möglich, den invertierenden Eingang mit dem Ausgang des Kanalmultiplexers des A/D-Wandlers zu verbinden, vorausgesetzt allerdings, dass der ADC nicht aktiv ist. So ist es möglich, über das `ADMUX`-Register des ADC alle 8 Analog-Eingänge an Port A mit dem Komparator zu verbinden.

Der Zustand des Komparator-Ausgangs ist (durch Synchronisation um 1-2 CPU-Takte verzögert) in dem Bit `ACO` im Steuerregister `ACSR` abrufbar. Der Komparator besitzt einen eigenen Interrupt, der derart konfigurierbar ist, dass er entweder auf eine positive oder eine negative Flanke des Komparator-Ausgangs oder auf jeden Wechsel reagiert. Zusätzlich besteht die Möglichkeit, mit dem Pegelwechsel ein Input Capture des 16-Bit-Timers 1 auszulösen (s. dazu Abschnitt 1.5.2 auf Seite 12).

## 1.8 Serielle Schnittstellen

Der ATMega16 verfügt zur Kommunikation mit anderen Komponenten über eine Reihe unterschiedlicher Interfaces. Alle seriellen Interfaces stellen unabhängige Hardware-Einheiten dar, die die eigentlichen Kommunikationsvorgänge selbständig durchführen und über Interrupts verfügen, um der CPU mitzuteilen, wenn ein entsprechendes Ereignis aufgetreten ist.

### 1.8.1 SPI

Die **SPI-Schnittstelle** (**S**erial **P**eripheral **I**nterface), deren Hardware auch von der In-System-Programmierschnittstelle des Controllers genutzt wird, ist zur Nahbereichs-Kommunikation mit anderen Mikrocontrollern im selben System oder z.B. mit Speicherbausteinen und ähnlicher Peripherie nutzbar. Die Hardware der SPI-Schnittstelle wird auch zur in-System-Programmierung (AVR-ISP) des Controllers verwendet.

Die Sende- und Empfangseinheit des SPI besteht im Prinzip aus einem Schieberegister, an dessen beiden Enden die Datenleitungen MOSI (**M**aster **O**ut **S**lave **I**n) und MISO (**M**aster **I**n **S**lave **O**ut) angeschlossen sind[8]. Das vom SPI-Master generierte Taktsignal SCK sorgt dafür, dass der Inhalt des Schieberegisters bei jeder fallenden Flanke um ein Bit nach 'hinten' (also zum LSB hin) geschoben wird. Dadurch wird zeitgleich vom 'höherwertigen' Ende das empfangene Byte hereingeschoben und am anderen Ende das zu sendende Byte

[8] Welche der Datenleitungen mit welchem 'Ende' des Schieberegisters verbunden wird, hängt davon ab, ob das Interface als Master oder als Slave konfiguriert ist.

ausgegeben. Nach 8 Takten ist der Übertragungsvorgang beendet und das empfangene Byte kann aus dem Schieberegister ausgelesen werden.

Eine Datenübertragung findet bei SPI grundsätzlich bidirektional statt, d.h. es steht für jede Datenrichtung eine Leitung zur Verfügung (MOSI und MISO). Zusätzlich wird das Taktsignal (SCK) übertragen, das vom Master generiert wird. Wenn der Master auf MOSI ein Byte sendet, wird synchron ein Byte vom ausgewählten Slave über MISO an den Master übertragen. Zur Auswahl der Slaves muss der Master zusätzlich die Slave-Select-Signale generieren (zumindest dann, wenn mehr als ein Slave vorhanden ist), was zu einem relativ hohen Verdrahtungsaufwand bei der Verwendung von SPI führt. Durch die bidirektionale Übertragung und die direkte Adressierung über Slave-Select-Leitungen (direkt oder gemultiplext) sowie die möglichen Taktfrequenzen ist SPI jedoch im Vergleich sehr schnell. Aufgrund des Aufbaus der Schnittstelle eignet sich SPI sehr gut zur Ansteuerung von Schieberegistern, die z.B. zur Porterweiterung dienen können. Dabei können, bedingt durch die getrennten Datenleitungen, synchron Daten eingelesen und ausgegeben werden.

Bei der Verwendung des SPI im Master-Betrieb ist unbedingt darauf zu achten, dass der Slave Select ($\overline{\text{SS}}$)-Pin des Controllers, der im Master-Betrieb eigentlich keine Funktion hat, ausschließlich als Ausgang verwendet werden darf. Ist er als Eingang konfiguriert und liegt ein Low-Pegel an, dann schaltet das Interface automatisch in den Slave-Betrieb um. Dies dient dazu, den Aufbau eines Multi-Master-Systems zu ermöglichen, was aufgrund der Bustopologie, die prinzipiell nur einen Master gleichzeitig zulässt, dazu führt, dass zumindest der vom gerade sendenden Master angesprochene Teilnehmer sofort in den Slave-Betrieb umschalten muss.

### 1.8.2 TWI

Einen ähnlichen Anwendungsbereich wie SPI besitzt das **TWI** (**T**wo **W**ire **I**nterface), das dem von anderen Herstellern als $I^2C$ bezeichneten System entspricht. Bei TWI existieren nur zwei Leitungen, eine Daten- und eine Taktleitung, was den Verdrahtungsaufwand im Vergleich zu SPI erheblich reduziert und die Flexibilität erhöht. Bei TWI muss allerdings die Slave-Adresse ebenfalls seriell übertragen werden, was zusätzlich zur unidirektionalen Kommunikation zu Performance-Nachteilen gegenüber SPI führt. Durch die serielle Übertragung der Adresse ist außerdem die maximale Anzahl der Busteilnehmer auf 128 begrenzt.

### 1.8.3 USART

Des weiteren steht ein **USART**-(**U**niversal **S**ynchronous/**A**synchronous **R**eceiver/**T**ransmitter) Interface zur Verfügung. Dieses ähnelt in der Funktionsweise den beiden oben erwähnten seriellen Bus-Interfaces SPI und TWI. In einigen Details existieren jedoch Unterschiede. Das USART besitzt, wie die Bezeichnung bereits andeutet, zwei Betriebsarten, eine synchrone und eine asynchrone. Der Unterschied liegt darin, dass bei einer synchronen Übertragung ein Taktsignal übermittelt wird, auf das der Datenstrom synchronisiert ist. Diese Betriebsart ist dem zuvor erwähnten SPI sehr ähnlich, zumal es dabei ebenfalls Master- und Slave-Betrieb gibt, je nachdem, ob der Controller den Takt generiert oder ob er einen externen Übertragungstakt erhält.

Die am häufigsten verwendete Betriebsart des USART ist jedoch die asynchrone. Dabei wird kein Taktsignal mit übertragen, sondern die Synchronisation von Sender und Empfänger, die jeweils einen eigenen Takt erzeugen, über die Flanken der Bitübertragung durchgeführt. Dabei ist es essentiell wichtig, dass die Taktgeneratoren von Sender und Empfänger mit möglichst exakt der selben Frequenz arbeiten. Die Abweichungen der Takte müssen zumindest ausreichend gering sein, um zu gewährleisten, dass innerhalb eines übertragenen Frames kein Pegel falsch erfasst wird. I.d.R. kann man davon ausgehen, dass Abweichungen unter 2% eine sichere Datenübertragung garantieren.

Ähnlich wie bei den anderen erwähnten seriellen Interfaces erfolgt die Datenübertragung beim USART durch Schieberegister, allerdings existieren im Unterschied z.B. zum SPI zwei dieser Register, je eines für den Sender und den Empfänger, wodurch sich eine bessere Pufferung der ankommenden Daten ergibt und Sender und Empfänger unabhängiger voneinander arbeiten können. Durch diesen Aufbau eignet sich das USART besonders zur Punkt-zu-Punkt-Kommunikation zweier (gleichberechtigter) Teilnehmer, es lassen sich aber auch Bus-Topologien realisieren.

Mit der asynchronen Betriebsart des USART lässt sich relativ einfach eine Kommunikation mit einem PC über die EIA-232-Schnittstelle (vormals RS-232, COM-Port des PC) herstellen[9]. Die reine Datenübertragung des EIA-232-Protokolls wird von der Hardware des ATMega16-USART unterstützt, wobei allerdings die Flusssteuerung, wenn sie erforderlich ist, in Software durchzuführen ist (Erzeugung und Auswertung der Statussignale bei Hardware-Flusssteuerung oder Implementierung eines XON/XOFF-Protokolls). Für einfache Datenübertragungsanwendungen genügt jedoch meist eine Kommunikation ohne Flusssteuerung, also nur über die beiden Datenleitungen TxD und RxD ('Transmit Data' und 'Receive Data').

Der EIA-232-Standard sieht eine Reihe von Baudraten vor, die sich i.d.R. als ganzzahlige Vielfache von 300 Baud ergeben (meist $600 * 2^n$ oder $900 * 2^n$ Bd). Um diese wirklich exakt zu erzeugen, kann man den Controller mit einem sogenannten Baudratenquarz betreiben, dessen Frequenz ein ganzzahliges Vielfaches der üblichen Baudraten ist (z.B. 7,3728 oder 11,0592 MHz), so dass sich die gewünschten Takte durch die Teiler im Controller präzise generieren lassen. Allerdings lassen sich auch mit 'normalen' Quarzfrequenzen viele der gebräuchlichen Baudraten mit deutlich weniger als 2% Abweichung generieren, so dass Baudratenquarze eigentlich nur in Sonderfällen wirklich erforderlich sind (z.B. dann, wenn eine große Bandbreite unterschiedlicher Baudraten einstellbar sein soll).

Für das Senden und Empfangen von Daten über das USART stehen insgesamt drei Interrupts zur Verfügung. Zusätzlich besteht bei entsprechender Konfiguration die Möglichkeit, durch entsprechende Flags bestimmte Fehler bei der Übertragung zu erkennen (Parity Error, Framing Error und Empfangspuffer-Überlauf).

## 1.9 Hardware-Interrupts

Das Interrupt-Verfahren stellt i.d.R. die schnellste Methode dar, um auf Ereignisse zu reagieren, die in Bezug auf den Programmablauf asynchron auftreten. Im Unterschied

[9] Zur Übertragung ist i.d.R. eine Pegelanpassung erforderlich, da der RS232-Standard -5...-15 V für eine logische '1' und +5...15 V für eine '0' (invertierte Übertragung) vorschreibt (Pegel jeweils für die Senderseite, auf der Empfangsseite sind die Pegel -3...-15 V bzw. +3...+15 V). Schnittstellen-Treiber wie der weit verbreitete MAX232 enthalten neben den Pegelwandlern auch Ladungspumpenschaltungen zur Erzeugung der symmetrischen Spannungen, so dass lediglich eine 5 V-Spannungsquelle erforderlich ist.

zum Polling- (Abfrage-) Verfahren werden unnötige Wartezeiten (Busy-Wait) vermieden. Der ATMega16 stellt abgesehen vom Reset-Vektor insgesamt 20 Interrupt-Vektoren zur schnellen Bearbeitung von Hardware-Ereignissen zur Verfügung (s. Tabelle 2). Die Interrupts sind je nach Bedarf über Enable-Bits in den entsprechenden I/O-Registern einzeln (de-) aktivierbar. Durch ein globales Enable (I-Bit im Statusregister `SREG`) können alle aktiven Interrupts gleichzeitig zur Bearbeitung freigegeben oder gesperrt werden. Dies ist v.a. bei bestimmten Vorgängen von Bedeutung, die nicht unterbrochen werden dürfen.

| Nr. | Adresse | Vektorname | Beschreibung |
|---|---|---|---|
| 1 | 0x000 | RESET | Reset-Vektor |
| 2 | 0x002 | INT0 | Externer Interrupt 0 |
| 3 | 0x004 | INT1 | Externer Interrupt 1 |
| 4 | 0x006 | TIMER2_COMP | Timer 2 Compare Match |
| 5 | 0x008 | TIMER2_OVF | Timer 2 Overflow |
| 6 | 0x00A | TIMER1_CAPT | Timer 1 Capture |
| 7 | 0x00C | TIMER1_COMPA | Timer 1 Compare Match A |
| 8 | 0x00E | TIMER1_COMPB | Timer 1 Compare Match B |
| 9 | 0x010 | TIMER1_OVF | Timer 1 Overflow |
| 10 | 0x012 | TIMER0_OVF | Timer 0 Overflow |
| 11 | 0x014 | SPI_STC | SPI Serial Transfer Complete |
| 12 | 0x016 | USART_RXC | USART Receive Complete |
| 13 | 0x018 | USART_UDRE | USART Data Register Empty |
| 14 | 0x01A | USART_TXC | USART Transmit Complete |
| 15 | 0x01C | ADC | ADC Conversion Complete |
| 16 | 0x01E | EE_RDY | EEPROM Ready |
| 17 | 0x020 | ANA_COMP | Analog Comparator |
| 18 | 0x022 | TWI | Two Wire Serial Interface |
| 19 | 0x024 | INT2 | Externer Interrupt 2 |
| 20 | 0x026 | TIMER0_COMP | Timer 0 Compare Match |
| 21 | 0x028 | SPM_RDY | Store Program Memory Ready |

**Tabelle 2:** Interrupt-Vektoren im ATMega16

Das Setzen des Interrupt-Flags einer lokal und global freigegebenen Interruptquelle durch Auftreten eines entsprechenden Hardware-Ereignisses hat zur Folge, dass der gerade in Arbeit befindliche Befehl zu Ende ausgeführt wird. Anschließend wird der Stand des Programmzählers auf den Stack gerettet (s. Abschnitt 1.2) und das zum Interrupt gehörende Unterprogramm (Interrupt-Handler oder -Subroutine, ISR) aufgerufen. Beim Sprung in die ISR wird das auslösende Flag automatisch gelöscht[10]. Ist die ISR abgearbeitet, so wird der Programmzähler mit dem auf dem Stack befindlichen Wert geladen und um eins erhöht, so dass das zuvor unterbrochene Programm fortgesetzt werden kann. Nach dem Rücksprung wird zunächst mindestens ein Befehl im Programm ausgeführt, bevor weitere mittlerweile aufgetretene Interrupts abgearbeitet werden können.

Es ist bei der Programmierung mit Interrupts darauf zu achten, dass bei Einsprung in die ISR das globale Interrupt Enable Bit (I) im Statusregister `SREG` automatisch gelöscht

[10] Ausnahme: Das Flag des USART Receive Interrupts wird erst gelöscht, wenn nach einem Lesezugriff auf das USART-Datenregister der Empfangspuffer leer ist.

und erst wieder gesetzt wird, wenn die ISR verlassen wird! Das führt dazu, dass die ISR selbst nicht unterbrochen werden kann, es sei denn, man setzt das I-Bit in der ISR per Software auf '1'. Unterbrechbare Interrupt Handler sollten jedoch nur in Ausnahmefällen eingesetzt werden, wenn man sich über die Folgen ihrer Verwendung im Klaren ist. Weiterhin muss beachtet werden, dass auch bei gelöschtem I-Bit (alle Interrupts gesperrt) auftretende Hardware-Ereignisse die jeweiligen Interrupt-Flags setzen. Ein erneutes Setzen des I-Bits führt dann dazu, dass alle inzwischen aufgelaufenen Interrupts entsprechend ihrer Priorität[11] abgearbeitet werden (Je niedriger die Adresse des Interrupt-Vektors, desto höher die Priorität; RESET hat die höchste Priorität, dann folgt der externe Interrupt 0 usw., s. Tabelle 2 auf der vorherigen Seite). Ist dies nicht erwünscht, so sollte man **vor** der (erneuten) globalen Interrupt-Freigabe die entsprechenden Interrupt-Flags löschen.

Drei der Interrupts sind direkt mit entsprechenden Portpins verknüpft und so konfigurierbar, dass sie entweder auf eine fallende oder steigende Flanke, auf beide Flanken oder auf einen LOW-Pegel am Pin reagieren. Diese sogenannten externen Interrupts können z.B. zur Auswertung bestimmter externer Ereignisse genutzt werden. Das Anschließen von Tastern oder anderen mechanischen Kontakten an externe Interrupts ist i.d.R. nicht sinnvoll, da mechanische Kontakte immer prellen (d.h. der Kontakt schließt und öffnet mehrfach kurz hintereinander), was, bedingt durch die hohe Verarbeitungsgeschwindigkeit des Controllers, dazu führen kann, dass ein entsprechender Interrupt mehrfach direkt hintereinander ausgelöst wird. Zur zuverlässigen und eindeutigen Auswertung wären deshalb aufwändige Entprell-Maßnahmen erforderlich (s. Abschnitt 3.1.2). Die einzige sinnvolle Anwendung für Schaltkontakte an externen Interrupts ist das Aufwecken des $\mu$C aus einem Sleep-Modus (s. Abschnitt 1.3).

Zu einem freigegebenen Interrupt muss grundsätzlich ein entsprechender Interrupt-Handler vorhanden sein, da ansonsten bei Auftreten des Interrupt-Ereignisses ein Absturz des Programms erfolgen kann. Hochsprachen-Compiler erzeugen für solche Fälle einen 'Spurious Interrupt Handler', der i.d.R. standardmäßig einen Reset zur Folge hat. Alle Interrupt-Vektoren, zu denen kein Handler existiert, werden mit einem Sprungbefehl zu diesem Spurious Interrupt Handler versehen, womit jeder versehentlich freigegebene Interrupt bei seinem Auftreten einen Software-Reset des Controllers verursacht, also einen Sprung an den Beginn des Programms durchführt. Alternativ (bei der Programmierung in Assembler) können auch direkt in die nicht verwendeten Interrupt-Vektoren Sprungbefehle zur Adresse des Programmbeginns geschrieben werden. Möchte man bei solchen sporadischen Ereignissen keinen Reset des Controllers, kann in die entsprechenden Vektoren auch ein direktes 'return from interrupt' geschrieben werden, was dazu führt, dass das Programm einfach weiterläuft.

Bei der Arbeit mit Interrupts ist zu bedenken, dass diese, sofern freigegeben, zu jedem Zeitpunkt während der Abarbeitung des Programms auftreten können. Dadurch können bestimmte Prozesse behindert oder sogar empfindlich gestört werden. Die Ursache für solche Störungen liegt in den meisten Fällen in der Behandlung des Statusregisters `SREG`[12]. Da die meisten arithmetischen und logischen Operationen die Flags im `SREG` beeinflussen, ist es nachvollziehbar, dass es zu Problemen kommt, wenn ein Interrupt zwischen einer

[11] Es handelt sich hierbei allerdings um keine 'echten' Prioritäten (d.h. die Interrupts können nicht, wie z.B. bei der 8051er-Familie, auf unterschiedliche Vorrangstufen gesetzt werden, so dass ein Interrupt höherer Priorität die Bearbeitung eines anderen Interrupts niederer Priorität unterbrechen kann), sondern der Begriff bezieht sich hier ausschließlich auf die Abarbeitungsreihenfolge bei **zeitgleich** anstehenden Ereignissen.

[12] Zur Beschreibung des `SREG` siehe Abschnitt 1.2.4 auf Seite 6.

Operation und der anschließenden Auswertung oder Weiterverarbeitung der Status-Flags auftritt und innerhalb des Interrupt-Handlers weitere Operationen durchgeführt werden, die die Status-Flags beeinflussen. Um Störungen des Programmablaufs zu vermeiden, ist es i.d.R. unbedingt erforderlich, nach dem Einsprung in den Interrupt-Handler das `SREG` zu sichern und am Ende der Interrupt-Bearbeitung wiederherzustellen. Dies geschieht meist durch ein Ablegen des `SREG` auf dem Stack. Je nach Anwendung müssen auch einige der Rechenregister gesichert werden. Das gilt v.a. für diejenigen Register, die Sonderfunktionen in Zusammenhang mit Zeiger-Arithmetik ausüben. Bei einer Hochsprachen-Programmierung erfolgen all diese Sicherungsmaßnahmen durch den Compiler, so dass der Programmierer lediglich wissen muss, *dass* etwas gesichert wird. Das ist insbesondere bei zeitkritischen Anwendungen von Bedeutung, da das Register-Sichern durchaus einige zusätzliche Prozessortakte dauern kann, abhängig davon, wie viele Register zusätzlich zum `SREG` zu sichern sind.

Des weiteren ist zu beachten, dass auch eine Interrupt-Bearbeitung keinesfalls verzögerungsfrei ist. Das Sichern des Programmzählers, das Löschen von `I` und der Sprung in den Interrupt-Vektor benötigt 4 CPU-Takte. Dazu kommen oft 1-2 Takte für das vorherige zu-Ende-Bearbeiten des gerade ausgeführten Befehls. Der Sprung aus dem Interrupt-Vektor in den Interrupt Handler benötigt je nach Art des Sprunges 2-3 Takte (bei $\mu$Cs mit bis zu 8 KiB Flash reicht ein relativer Sprung, der nur zwei Takte benötigt, während bei AVRs mit 16 KiB Flash und mehr i.d.R. ein absoluter Sprung erforderlich ist, der 3 Takte benötigt). Das Sichern von `SREG` dauert 3 Takte, zusätzliche Sicherungen zwei Takte pro Rechenregister. Erst nach diesen Aktionen wird der eigentliche Programmcode des Interrupt Handlers ausgeführt. Das Rückschreiben der gesicherten Register am Ende des Interrupt Handlers dauert noch einmal genauso lange wie das Sichern, und der eigentliche Rücksprung inklusive `I` wieder setzen und Programmzähler wiederherstellen wiederum 4 Takte. Es ergibt sich je nach Art der Programmierung ein Overhead von *minimal* 8 CPU-Takten, im Normalfall (und insbesondere bei Hochsprachen-Programmierung) deutlich mehr. Diese Aspekte sind bei der Programmierung zeitkritischer Vorgänge in Betracht zu ziehen.

# 2 Mikrocontroller-Programmierung in C

Ein Mikrocontroller 'versteht' grundsätzlich nur binäre Instruktionen. Jeder Controller bzw. jede Controller-Familie besitzt einen spezifischen Befehlssatz, der zur besseren Les- und Programmierbarkeit durch mnemonics, also Befehle aus Textzeichen, und Operanden dargestellt wird, wobei ein mnemonic grundsätzlich genau einem Maschinenbefehl entspricht. Sogenannte Assembler erzeugen aus diesen Befehlen die entsprechenden Binärcodes. Assembler-Programmierung ist zwar der beste Weg, den $\mu$C richtig kennenzulernen, jedoch sind solche Programme bei höherer Komplexität nicht mehr gut lesbar. Aus diesem Grund existieren für die meisten $\mu$Cs Hochsprachen-Compiler, wobei sich C als geeignetste Programmiersprache erwiesen und demzufolge auch durchgesetzt hat.

Zur Programmierung des ATMega16 und aller anderen Controller der AVR-Familie stehen diverse Tools zur Verfügung. Zur Programmierung in C benötigt man

- Editor (vorzugsweise mit Farb-Syntax-Darstellung)
- C-Compiler, der Assembler-Code erzeugt
- Assembler und Linker
- Programmiergerät oder -Adapter

Das im Editor erstellte C-Programm wird vom Compiler in Assembler-Befehle übersetzt. Der Assembler erzeugt daraus den Maschinen- (Hexadezimal- bzw. Binär-) Code. Der Linker fügt die Programm- und Datensegmente (Objects) zusammen und allokiert sie im zur Verfügung stehenden Speicher. Das Ergebnis dieser Verarbeitung ist eine Hexadezimal-Datei (Intel-Hex-Format), deren Inhalt nach Konvertierung durch ein Programmiergerät oder einen Adapter mit entsprechender PC-Software in den Speicher des Controllers geschrieben wird.

Mit der Software AVRStudio, die von Atmel kostenlos erhältlich ist, steht eine komplette Entwicklungsumgebung (IDE) zur Verfügung, in die der ebenfalls frei erhältliche C-Compiler AVR-GCC als Plugin eingebunden werden kann, was bei Installation der WINAVR-Distribution, die diesen Compiler enthält, automatisch geschieht. Diese Integrationsmöglichkeit hat u.a. den Vorteil, dass der Entwickler alle o.g. Schritte (Compilieren, Assemblieren, Linken) mit nur einem Knopfdruck ausführen kann. WINAVR enthält neben dem AVR-GCC-C-Compiler auch die AVR-libc, die Funktionsbibliotheken zur Verfügung stellt, die speziell auf die AVR-Controller zugeschnitten sind. Darunter sind einerseits die C-Standardbibliotheken, die jeder ANSI-C-Compiler mitbringt, und zusätzliche, $\mu$C-spezifische. Auch die Device-Header-Dateien, die die Definitionen der Register, Interrupt-Vektoren usw. enthalten, sind Bestandteil der AVR-libc.

Der GCC-C-Compiler ist ein ANSI-C-Compiler, der im Unterschied zu kommerziellen Embedded-C-Compilern wie CodeVision oder Keil keine $\mu$C-spezifischen Erweiterungen, v.a. in Hinblick auf Bitmanipulationen, enthält. Dies führt dazu, dass z.B. der Zugriff auf einzelne Bits in der ANSI-C-konformen Syntax erfolgen muss (s. Abschnitt 2.8). Der GCC-C-Compiler erfüllt den gegenüber dem ursprünglichen ANSI-C (C89, [2]) erweiterten C99-Standard[13], der z.B. einige Restriktionen der ersten standardisierten Version lockert und einige an C++ angelehnte Konstrukte zulässt (Kommentare bis zum Zeilenende mit '//', Variablendeklarationen an beliebiger Stelle im Programm usw.).

[13] Bei entsprechender Einstellung, s. Abschnitt 4

## 2.1 Allgemeine Hinweise zur Mikrocontroller-Programmierung

Die Programmierung von Embedded Systems bzw. Mikrocontrollern unterscheidet sich v.a. aufgrund der im Vergleich sehr eingeschränkten Hardware-Ressourcen (Speicher, Rechenleistung) von der PC-Programmierung. Die Verwendung bestimmter Funktionen (z.B. Fließkomma-Arithmetik, bestimmte mathematische Bibliotheksfunktionen) kann dazu führen, dass der zur Verfügung stehende Speicher nicht ausreicht. Hinzu kommt, dass man 'hardwarenah' programmiert, d.h. dass die physikalischen Eigenschaften der Hardware zu beachten sind (Schaltfrequenzen, Verzögerungszeiten...). Die folgenden Abschnitte erläutern vom ANSI-Standard nicht abgedeckte Besonderheiten der $\mu$C-Programmierung sowie Vorgehensweisen, die zwar im Standard vorgesehen sind, jedoch in der PC-Programmierung praktisch keine Rolle spielen.

## 2.2 Aufbau eines C-Programms

Grundsätzlich gilt auch für Mikrocontroller-Programme die ANSI-C-Syntax (C89/90- oder C99-Standard), unabhängig vom Compiler. Lediglich bei $\mu$C-spezifischen Erweiterungen (und das gilt auch für die Bibliotheken), die nicht im ANSI-Standard enthalten sind, können Abweichungen und auch erhebliche Unterschiede zwischen verschiedenen Compilern auftreten, wodurch sich z.T. erhebliche Probleme mit der Portabilität von Code ergeben. Ein in CodeVision geschriebenes Programm lässt sich z.B. nicht unter AVR-GCC compilieren.

So wie es eine Vielzahl (Software-Hersteller-) spezifischer Lösungen für die Syntax der Compiler gibt, existieren auch eine ganze Reihe Philosophien, wie ein Programm aufzubauen ist. Man sollte sich auf eine bestimmte Gliederung festlegen, um Programmteile immer schnell wiederzufinden. Eine sinnvolle Reihenfolge der durchzuführenden Aktionen könnte (muss aber nicht) wie folgt aussehen:

1. Einbinden der benötigten Bibliotheks- bzw. Header-Dateien (`#include...`)
2. Definition von Makros (dient der Übersicht; kann prinzipiell auch im Programmverlauf vor dem ersten Aufruf geschehen)
3. Typdefinitionen (falls erforderlich)
4. Deklaration globaler Variablen
5. Deklaration von Funktionsprototypen (falls erforderlich)
6. Definition der Interrupt-Handler (ISR)
7. Definition von Funktionen (Unterprogrammen)
8. Hauptprogramm 'main':
   (a) Deklaration lokaler Variablen
   (b) Initialisierung der I/O-Register
   (c) globale Interrupt-Freigabe
   (d) eigentliches Hauptprogramm (in Endlosschleife, also z.B. `while(1){}`)

Die folgenden Abschnitte gehen auf die genannten Punkte näher ein, sofern sie nicht dem ANSI-Standard entsprechen. Allgemeine Hinweise zu Variablendeklarationen, Funktionsaufrufen usw. finden deshalb an dieser Stelle keine Erwähnung.

## 2.3 Bibliotheken

Die zum Compiler-Paket gehörende Bibliothek (AVR-libc) enthält neben den (ggf. an die Bedürfnisse der AVR-Plattform angepassten) Standard-Bibliotheken, die im ANSI-Standard enthalten sind, eine Reihe $\mu$C-spezifischer Bibliotheken. Diese enthalten Definitionen, Makros und Funktionen, die dem Programmierer die Arbeit erleichtern und ihm helfen, Programme übersichtlich und nachvollziehbar zu gestalten.

Eine sehr wichtige Grundlage für jedes Mikrocontroller-C-Programm bilden die Headerdateien mit den Definitionen für die I/O-Register, Interrupt-Vektoren, Speicheraufteilung usw., die für jeden AVR-Typ in einer 'io[Typkürzel].h'-Datei abgelegt sind. Für den ATMega16 heißt die betreffende Headerdatei z.B. iom16.h. Allerdings werden nicht diese Header direkt ins Programm eingebunden, sondern eine allgemeine Datei 'io.h'. Über die Compiler-Konfiguration, die durch ein sog. Makefile geschieht, wird der verwendete $\mu$C eingestellt (bei der Verwendung von AVRStudio geht das über die Configuration Options unter dem Menüpunkt 'Project') und dementsprechend automatisch die richtige Headerdatei eingebunden. Durch die Verwendung dieser Headerdateien ist es z.B. möglich, direkt die Register- und Bitnamen aus den Datenblättern zu verwenden.

## 2.4 Datentypen und Variablen

***Allgemeiner Hinweis:*** *Bei der Definition von Makros (mit* `#define`*) und Variablen ist darauf zu achten, dass die Schreibweise eindeutig ist. Makronamen sollten immer vollständig aus* `GROSSBUCHSTABEN` *bestehen, während Variablennamen* `klein` *zu schreiben sind. Ein Leser kann dann immer sofort erkennen, um was es sich handelt. Des weiteren ist zu beachten, dass Bezeichner, die mit Unterstrichen ('_') beginnen, für das System (Compiler und Bibliotheken) reserviert sind und vom Programmierer nicht vergeben werden dürfen. Diese Vereinbarung dient der Vermeidung von Kollisionen bei der Namensvergabe.*

Da es sich beim verwendeten ATMega16 um einen 8-Bit-Controller handelt, kann dieser grundsätzlich auch nur mit 8-Bit-Variablen rechnen. Das Berechnen größerer Variablen (int, long int, float) erfordert mehrere Schritte. In C existieren zur Darstellung ganzzahliger (Integer-) Werte die Typen `char` (Character, 8 Bit), `int` (Integer, 16 Bit), `long (int)` (long Integer, 32 Bit) und `long long (int)` (64 Bit). Der Typ `char` ist ursprünglich zur Speicherung von ASCII-Zeichen vorgesehen, wo auch der Name 'Character' ('Textzeichen') herrührt. Da `char` aber der einzige von C zur Verfügung gestellte 8-Bit-Datentyp ist, wird er bei der Mikrocontroller-Programmierung als Rechenvariable 'missbraucht'[14]. Sowohl `char` als auch die unterschiedlichen `int`-Typen können vorzeichenbehaftet (`signed`) oder nicht vorzeichenbehaftet (`unsigned`) verwendet werden. Die

[14]Es ist zu beachten, dass der Compiler nach dem ANSI-Standard im Prinzip alle Rechenoperationen **mindestens** in 16 Bit ausführen muss, auch wenn alle an einer Operation beteiligten Werte in 8 Bit darstellbar sind. In vielen Fällen greift allerdings die Optimierung und reduziert Operationen bei entsprechenden Werten auf 8 Bit. Die Optimierung ist jedoch (noch) nicht für alle Operationen vollständig implementiert.

Header-Datei `stdint.h` enthält eine Reihe vordefinierte Integer-Typen (`int8_t`, `int16_t`, `int32_t`, `int64_t`, `uint8_t`...), deren Verwendung aufgrund der integrierten Längenangabe zur besseren Lesbarkeit des Programmcodes führen kann. Dabei entspricht `uint8_t` einem `(unsigned) char`, `int32_t` einem `(signed) long (int)`.

Die Angabe von Integer-Zahlenwerten kann in AVR-GCC-Syntax neben den in ANSI-C gebräuchlichen Hexadezimal-(`0x1a`), Dezimal-(`26`) und Oktal-(`032`) Darstellungen auch in Binär-Ausdrücken (`0b00011010`) erfolgen. Die Binärform kann bei Zugriffen auf einzelne Bits (Maskierung, s. Abschnitt 2.8 auf Seite 27) von Vorteil sein.

Berechnungen mit Gleitkomma-Zahlen sind auf Festkomma-CPUs immer mit recht großem Ressourcen-Aufwand verbunden und sollten wenn möglich vermieden werden (siehe auch Abschnitt 2.9 auf Seite 31). Für eine Darstellung nicht-ganzzahliger Werte genügt jedoch meist eine Festkomma-Darstellung, die durch mehrere ganzzahlige Variablen oder durch eine Multiplikation und (bei der Ausgabe) einer Division mit anschließendem Einfügen des Dezimaltrennzeichens an der gewünschten Stelle implementiert werden kann (Beispiel: Ein Spannungswert soll in Volt mit zwei Nachkommastellen ausgegeben werden. Dazu rechnet man intern in hundertstel Volt ganzzahlig. Bei der Ausgabe (z.B. im ASCII-Format) wird dann einfach das Dezimaltrennzeichen an der entsprechenden Stelle eingefügt).

Sollten Gleitkomma-Ausdrücke unvermeidbar sein, so ist zu beachten, dass der AVR-GCC-C-Compiler derzeit nur 32-Bit-Gleitkommazahlen unterstützt. Daher wird der Typ `double`, der auf den meisten Systemen die doppelte Auflösung von `float` hat (64 Bit), im AVR-GCC-C genauso behandelt wie ein `float`[15]. Viele Funktionen aus den C-Standard-Bibliotheken erwarten als Parameter `double`-Werte. Übergibt man einer solchen Funktion `float`-Werte, erhält man Warnmeldungen, die man durch explizite Typkonversionen (Casts) vermeiden kann. Verwendet man solche Funktionen häufiger, kann es u.U. sinnvoll sein, alle Gleitkommavariablen direkt als `double` zu deklarieren, da dadurch kein zusätzlicher Speicher- oder Rechenzeitbedarf entsteht.

Wie oben bereits erwähnt ist es in C99 auch möglich, Variablen an beliebiger Stelle im Programm zu deklarieren[16]. Es ist jedoch u.U. sinnvoll, davon keinen Gebrauch zu machen, sondern der Übersicht halber Variablen nach wie vor am Anfang des Blocks, in dem sie verwendet werden zu deklarieren. **Globale** und **lokale statische Variablen** sind möglichst zu **vermeiden**, da sie permanent Speicherplatz belegen. Einzige Ausnahme sollten Variablen sein, über die Interrupt-Handler (ISRs) Daten mit dem laufenden Programm austauschen. **Variablen sollten immer mit dem kleinstmöglichen Gültigkeitsbereich** (Scope) **deklariert werden**.

Globale Variablen, die sowohl aus dem laufenden Programm heraus als auch in ISRs veränderbar sein sollen, sind bei der Deklaration mit dem Typqualifizierer `volatile` zu versehen. Das ist erforderlich, weil ein Interrupt-Handler aus Compiler-Sicht nie aufgerufen wird und deshalb der Optimizer die Freiheit hat, Zugriffe auf Variablen 'wegzuoptimieren', wenn diese sich während des normalen Programmablaufs offensichtlich gar nicht ändern kann. Das hätte zur Folge, dass sich eine global deklarierte Variable, die in einer ISR verändert wird, aus Sicht des Hauptprogrammes, in dem sie verarbeitet werden soll, u.U. gar nicht ändert. Soll z.B. in einem Interrupt Handler eine Variable hochgezählt werden

[15] Das ist nach ANSI auch zulässig, da im Standard lediglich vorgeschrieben ist, dass `double` mindestens dieselbe Auflösung haben muss wie `float`

[16] In WINAVR-C ist dies allerdings im Prinzip auch ohne die Einstellungen nach Abschnitt 4 auf Seite 39 möglich, ausgenommen jedoch die Deklaration einer Variable in der Startwertangabe einer `for`-Schleife, was nur in C99 zulässig ist.

(Schleifenzähler) und im Hauptprogramm bei Erreichen eines bestimmten Wertes dieses Zählers eine Aktion durchgeführt werden, dann kann es passieren, dass die Variable im Hauptprogramm nur einmal gelesen wird und somit den erwarteten Wert nie erreicht. Der Typqualifizierer `volatile` sorgt dafür, dass der Compiler weiß, dass sich diese Variable außerhalb des eigentlichen Programms ändern kann und somit kein Zugriff wegoptimiert werden darf.

Ein Beispiel für die Verwendung einer solchen globalen Variable ist das Auslesen des Ergebnisregisters des Analog-Digital-Wandlers. Die Deklaration der entsprechenden Variable, die die Daten, die in der ADC-ISR bearbeitet werden, an das Hauptprogramm übergibt, sollte folgendermaßen aussehen:

```
volatile unsigned int adc_result;
```

Anstelle von `unsigned int` kann selbstverständlich auch `uint16_t` verwendet werden.

Es ist zu beachten, dass ein Zugriff auf eine Variable oder ein Register mit mehr als 8 Bit in einem 8-Bit-System grundsätzlich mehrere Zyklen in Anspruch nimmt. Vor einem Zugriff auf globale Variablen mit einer Länge von mehr als 8 Bit, die auch von Interrupt-Subroutines geändert werden können, sind zumindest diejenigen Interrupts, die auf diese Variablen zugreifen, zu deaktivieren (s. Abschnitt 2.6 auf der nächsten Seite), um Datenverlust oder -Verfälschung zu verhindern. Ansonsten besteht die Möglichkeit, dass zwischen den einzelnen Verarbeitungsschritten der betreffende Interrupt auftritt und die Variable, die letztendlich verarbeitet wird, einen eventuell völlig unsinnigen Wert enthält. Deaktiviert man den Interrupt für die Dauer des Zugriffs auf die Variable, so lassen sich derartige Fehler vermeiden[17].

## 2.5 Daten im Flash und im EEPROM

Werte, die im Programm benötigt, jedoch dabei nicht verändert werden, können im Programmspeicher (Flash) abgelegt werden. Zum Lese-Zugriff[18] auf das Flash sind die Definitionen und Funktionen aus der Headerdatei `pgmspace.h` erforderlich, die sich im Unterverzeichnis `avr` des Standard-Include-Pfades befindet. Die Definition einer Konstanten im Flash erfolgt mit Hilfe des Makros `PROGMEM`, das eine Kurzschreibweise für das entsprechende Speicherklassen-Attribut darstellt. Beispiel:

```
const char PROGMEM string[] = "Hallo";
```

[17] Es ist dabei allerdings zu beachten, dass durch einfaches Sperren und anschließendes Wiederfreigeben einzelner oder aller Interrupts (im letztgenannten Fall über das I-Bit im Statusregister `SREG`, siehe Abschnitt 2.6) u.U. Programmfehler auftreten können, wenn das betreffende Steuerbit vor dem Sperren bereits gelöscht war. Wenn die Möglichkeit besteht, dass Interrupts, die von anderen Programmteilen gesperrt wurden, unbeabsichtigt freigegeben werden, sollten die Zustände der entsprechenden Freigabebits vor der Sperrung gesichert und am Ende anstelle einer einfachen Freigabe wieder zurückgeschrieben werden.

[18] Der Vollständigkeit halber: Schreibzugriffe auf den Flash-Speicher aus dem laufenden Programm heraus sind unter normalen Bedingungen nicht möglich. Es besteht jedoch die Möglichkeit, einen Teil des Flash als Boot-Section zu konfigurieren. Code, der sich in dieser Boot-Section befindet, kann dann in Speicher außerhalb der Boot-Section schreiben. Dieser Mechanismus ermöglicht den Betrieb eines Bootloader-Programms in der Boot-Section, das in der Lage ist, selbständig Updates für das eigentliche $\mu$C-Programm durchzuführen.

```
const uint16_t PROGMEM konstante1 = 12345;
const int8_t PROGMEM konstante2 = -120;
```

Für den Zugriff auf die so angelegten Konstanten stehen diverse Funktionen zur Verfügung, deren Verwendung vom Typ der Konstanten abhängt. Ein einzelnes Byte (`char` oder `(u)int8_t`) kann über die Funktion `pgm_read_byte()` ausgelesen werden. Soll z.B. die oben im Beispiel definierte `konstante2` in die Variable `dummy` kopiert werden, dann sieht das folgendermaßen aus:

```
dummy = pgm_read_byte(&konstante2);
```

Der Adressoperator vor der Konstante ist erforderlich, da die Funktionen die Adresse des Bytes im Flash übergeben bekommen.

Für Zugriffe auf 16-Bit-Konstanten existiert die Funktion `pgm_read_word()` und für 32-Bit-Konstanten `pgm_read_dword()`, deren Anwendung derjenigen von `pgm_read_byte()` entspricht. Auf einen String im Flash wird üblicherweise mit einem Zeiger zugegriffen, wobei der Zeiger selbstverständlich auch mit dem Attribut `PROGMEM` zu versehen ist.

Ganz ähnlich wie beim Flash funktionieren Zugriffe auf das im Controller integrierte EEPROM (zumindest auf C-Ebene, auf Hardware-Ebene unterscheiden sich die Zugriffsverfahren erheblich). Die Headerdatei `eeprom.h` enthält die entsprechenden Funktionen (zur ausführlichen Beschreibung der Funktionen aus der AVR-libc siehe [4]). In diesem Falle ist bei der Deklaration von Variablen anstelle von `PROGMEM` das Attribut `EEMEM` zu verwenden. Ein wichtiger Unterschied zum Flash besteht darin, dass das EEPROM auch während des normalen Programmablaufes geschrieben werden kann. Dadurch lassen sich wichtige Variablen, die nach dem Abschalten der Spannungsversorgung erhalten bleiben sollen, sichern. Dabei ist allerdings die begrenzte Lebensdauer des EEPROM zu beachten. Schreibvorgänge sollten auf das Nötigste beschränkt werden und es ist immer sinnvoll, vor dem Schreiben einer Variable ins EEPROM zu überprüfen, ob diese sich überhaupt geändert hat. Lesevorgänge haben allerdings selbstverständlich keinen Einfluss auf die Lebensdauer des Speichers.

## 2.6 Interrupts

Die Definition von Interrupt Subroutines geschieht in AVR-GCC mit Hilfe von Makros. Dazu ist es erforderlich, die Header-Datei `interrupt.h` einzubinden, die sich im Unterverzeichnis `avr` des Include-Pfades befindet. Die Definition einer ISR erfolgt dann mit dem Schlüsselwort `ISR` und dem Bezeichner für den gewünschten Interrupt-Vektor (nach Tabelle 2 auf Seite 17, jeweils dem angegebenen Vektornamen ein `_vect` anhängen, also z.B. `INT0_vect`):

```
ISR(VEKTORNAME_vect)
{
//Programmcode ISR
}
```

Bis auf die Einleitung mit `ISR` ist die Struktur der ISR gleich der eines normalen Unterprogramms bzw. einer Funktion. Allerdings können ISRs weder aus dem Programm heraus

aufgerufen werden, noch können sie einen Wert zurückgeben (was u.a. dazu führt, dass eine ISR ausschließlich über globale Variablen Daten mit dem Hauptprogramm austauschen kann).

Aufgrund der in Abschnitt 1.9 beschriebenen Eigenschaften der Interrupts ist darauf zu achten, dass ISRs nicht zu viel Code enthalten, da während der Ausführung das komplette System blockiert ist. Sinnvoll ist die Definition von Software- oder Job-Flags, also z.B. einzelnen Bits in einer (globalen) Variablen, die in der ISR gesetzt und nach dem Rücksprung ins Hauptprogramm abgefragt werden. Grundsätzlich sollte eine ISR möglichst nur Aktionen direkt durchführen, die zeitkritisch sind oder die durch andere Vorgänge im Programm behindert werden könnten. Auch der Aufruf von Unterprogrammen aus einer ISR ist zu vermeiden.

Zum Zugriff auf das I-Bit im Statusregister `SREG`, das die Interrupts global freigibt bzw. sperrt, stellt WINAVR zwei Funktionen zur Verfügung, die die entsprechenden Assemblerbefehle 'inline' in das Programm einfügen, d.h. es handelt sich nicht um echte Funktionsaufrufe, sondern der Compiler fügt an jeder Stelle, an der diese 'Pseudo-Funktionen' aufgerufen werden, den jeweiligen Assembler-Befehl direkt ein. Die Funktion `sei()` (**se**t **I**) setzt das I-Bit, wodurch die Interrupts freigegeben werden, während `cli()` (**cl**ear **I**) das Bit löscht und so alle Interrupts sperrt.

Wie in Abschnitt 1.9 bereits erwähnt, erzeugt der Compiler einen Default-Handler für 'bad interrupts', also Interrupts, die freigegeben sind, für die aber kein Handler existiert. Dieser Standard-Handler führt einen Sprung in den Reset-Vektor durch, wodurch das Programm von vorne beginnt. Ist dies nicht erwünscht, dann kann man (z.B. für Debugging-Zwecke) den Default-Handler anpassen, indem man mit

```
ISR(__vector_default)
{
    //Alternativer Code
}
```

den gewünschten Code einbaut.

## 2.7 Initialisierung der Peripherie-Komponenten

Vor der Benutzung von Peripherie-Komponenten des Controllers müssen diese durch entsprechende Belegung der zugehörigen Steuer- und Datenregister initialisiert werden. Die Bezeichnungen der Datenregister sind in der Controller-spezifischen Headerdatei enthalten und entsprechen i.d.R. den im Controller-Handbuch angegebenen Namen. Da nach dem Einschalten oder einem Reset die meisten I/O-Register mit dem Wert '0' initialisiert sind, ist es i.A. ausreichend, lediglich Register zu ändern, die zu in der Anwendung benutzten Peripherie-Komponenten gehören. Einzige Ausnahme ist der integrierte Analog-Komparator, dessen Steuerregister `ACSR` das Bit `ACD` (Analog Comparator Disable) enthält, das '1' gesetzt werden muss, um den Komparator zu deaktivieren, wenn er nicht benötigt wird. Dies kann bei bestimmten Anwendungen genutzt werden, um die Stromaufnahme des $\mu$C zu reduzieren.

Einige Komponenten, z.B. der 16-Bit-Timer 1 und der ADC arbeiten mit Daten, die länger als 8 Bit sind. Diese 16-Bit-Daten sind jeweils in zwei im I/O-Bereich hintereinander

angeordneten 8-Bit-Registern abgelegt. Dadurch können sich Probleme mit dem Zugriff ergeben, da sich z.B. während eines Lesezugriffs, der ja aus zwei separaten 8-Bit-Operationen besteht (8-Bit-Controller!), der Inhalt des Registers ändern kann, so dass die ausgelesenen Daten falsch sind. Der Controller verfügt zur Lösung dieses Problems für einige dieser Doppelregister über temporäre Register, in denen das jeweilige High-Byte (also die 8 höherwertigen Bits) zwischengespeichert wird, so dass gewährleistet ist, dass beide Bytes aus dem selben Zyklus stammen. Für die Programmierung bedeutet das jedoch, dass bei Zugriffen auf diese Register eine bestimmte Reihenfolge einzuhalten ist, da der Registerinhalt nur bei Zugriff auf das Low-Byte aktualisiert wird. Bei der Programmierung in C kann der Compiler den Zugriff jedoch automatisch korrekt ausführen. Dazu sind in den Header-Dateien 16-Bit-Register definiert, z.B. '`TCNT1`' für '`TCNT1H`' und '`TCNT1L`'. Bei der Programmierung sind also auf jeden Fall die 16-Bit-Register zu verwenden. Es ist jedoch zu beachten, dass ein solcher Zugriff nur scheinbar eine 16-Bit-Operation ist! Der Compiler macht daraus wieder den Doppelzugriff, da der Controller an sich nur 8 Bit auf einmal verarbeiten kann. Vor einer Änderung von 16-Bit-Registern (aber auch von globalen Variablen, die länger als 8 Bit sind!), auf die auch in ISRs zugegriffen wird, sollten die Interrupts mit `cli()` deaktiviert werden, um zu verhindern, dass der Zugriff unterbrochen wird[19].

## 2.8 Bitmanipulation und Maskierung

Bei der Arbeit mit Mikrocontrollern kommt es sehr häufig vor, dass einzelne Bits zu ändern oder abzufragen sind. Zur Bearbeitung einzelner Bits in einer Variable bzw. einem Register sieht ANSI-C, und damit auch der AVR-GCC, jedoch keine speziellen Befehle vor.

### 2.8.1 Bitzugriff mit logischen Operatoren

Möchte man z.B. in einem Steuerregister das Bit 4 auf '1' setzen, so könnte man schreiben `REGISTER = 0b00010000;` oder `REGISTER = 0x10;`, was allerdings dazu führen würde, dass alle anderen Bits in `REGISTER` auf '0' gesetzt werden. Dies ist in den meisten Fällen jedoch nicht erwünscht. Man behilft sich also in diesem Fall mit einer 'Maskierung' der Bits im Register durch eine logische Verknüpfung. Da ANSI-C logische Bitoperatoren zur Verfügung stellt, kann man das o.a. Problem lösen, indem man schreibt

```
REGISTER |= 0b00010000;
```

Diese bitweise ODER-Verknüpfung sorgt dafür, dass das Bit Nr. 4 den Wert '1' erhält, unabhängig davon, welchen Wert es vorher hatte, und dass die Zustände aller anderen Bits in `REGISTER` unverändert bleiben. Mit einer solchen Maske ist es selbstverständlich auch möglich, mehrere Bits gleichzeitig zu setzen.

Im umgekehrten Fall, wenn einzelne Bits zu löschen (also auf '0' zu setzen) sind, wird eine UND-Verknüpfung und eine komplementäre Maske verwendet, also für das obige Beispiel mit Bit 4:

```
REGISTER &= ~0b00010000;
```

[19] Zur Sperrung der Interrupts s.a. Abschnitt 2.4 auf Seite 22

Dadurch wird `REGISTER` mit dem Bitkomplement von `0b00010000` bitweise UND-verknüpft. Der Bitkomplement-Operator invertiert jedes einzelne Bit im Operanden, d.h.

$$\sim 00010000_{\mathrm{b}} \Rightarrow 11101111_{\mathrm{b}}$$

Bit Nr. 4 wird also in jedem Fall zu '0', während alle anderen Bits unverändert bleiben.

Manchmal ist es auch erforderlich, einzelne Bits zu invertieren ('toggeln'), ohne dabei zu wissen, welchen Zustand das Bit gerade besitzt, d.h. wenn ein Bit den Wert '0' hat, soll es auf '1' gesetzt werden und wenn es '1' ist auf '0'. Dies kann durch eine Exklusiv-ODER-Verknüpfung mit der Maske geschehen, z.B.

```
REGISTER ^= 0b00010000;
```

Dadurch wird Bit Nr. 4 in `REGISTER` mit '1' Exklusiv-ODER-verknüpft, was dazu führt, dass es unabhängig von seinem aktuellen Wert invertiert wird. Aufgrund ihrer Eigenschaft, nur dann eine '1' zu liefern, wenn sich die Zustände der miteinander verknüpften Bits unterscheiden, ist die Exklusiv-ODER-Verknüpfung auch sehr gut als Vergleichsoperation geeignet, wenn es darum geht, Änderungen zu verfolgen.

Mit den o.g. Methoden ist es jetzt zwar möglich, einzelne Bits in einem Register anzusprechen, jedoch ist ein Programm mit solchen Bitmasken nicht gut lesbar, weil man immer mühsam im Controller-Handbuch nachsehen muss, welches Bit welche Nummer hat und zusätzlich die Gefahr besteht, dass man sich in der Bitstelle verzählt. Sinnvoll wäre es, eine Darstellung zu finden, bei der man die im Datenblatt angegebenen Namen für die einzelnen Bits verwenden kann. Die mit dem Compiler gelieferten Bibliotheksdateien aus der AVR-libc für die unterschiedlichen Controller-Typen enthalten bereits Definitionen für die einzelnen Bits in der Art

```
#define TCCR1B  _SFR_IO8(0x2E) //Definition des Registers
#define CS10    0              //Definition einzelner Bits
#define CS11    1
#define CS12    2
#define WGM12   3
#define WGM13   4
#define ICES1   6
#define ICNC1   7
```

Das Beispiel ist ein Auszug aus der Bibliotheksdatei für den ATMega16. Der Name des Registers (`TCCR1B`) wird mit seiner absoluten Speicheradresse (0x2E) verknüpft[20]. Jedes einzelne Bit erhält als Namen seine 'Nummer' entsprechend der Wertigkeit im Register (bei einem 8-Bit-Register also die Nummern 0 – 7). Das Setzen z.B. von Bit `CS12` in `TCCR1B` sieht dann so aus:

```
TCCR1B |= 1 << CS12;
```

[20] Das '`_SFR_IO8(0x2E)`' teilt dem Compiler mit, dass es sich um ein Special Function Register handelt, genauer gesagt um ein 8-Bit-I/O-Register an der Adresse 0x2E; der Compiler benötigt diese Information, um bei Zugriffen auf dieses Register die korrekten Assembler-Befehle einsetzen zu können, die sich von denen zum Zugriff auf 'normale' SRAM-Variablen unterscheiden.

Im Klartext bedeutet dies, dass die Zahl '1', in Binärdarstellung '0b00000001', um 'CS12', also um zwei Stellen (dem Bezeichner 'CS12' war ja oben der Wert '2' zugewiesen worden) nach links verschoben wird. Dadurch erhält man die Maske '0b00000100'. Die '1' steht jetzt also an genau der Stelle, an der in TCCR1B das Bit CS12 steht. Möchte man mehrere Bits in einem Register gleichzeitig setzen, so kann man die einzelnen Bitmasken untereinander verodern:

```
TCCR1B |= (1 << CS10) | (1 << CS11) | (1 << CS12);
```

Der Ausdruck rechts vom '=' liefert die Maske '0b00000111', so dass in TCCR1B die drei letzten Bits gesetzt werden. Für das Löschen und Invertieren von Bits gilt Entsprechendes. Möchte man CS10 und CS12 löschen, so ergibt sich

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

(auf korrekte Klammerung achten!)

Die Maskierung ermöglicht auch das **Lesen der Zustände** einzelner Bits. Eine Warteschleife der Art

```
while(!(TIFR & (1 << OCF0)));
```

wird z.B. ausgeführt, so lange das Bit OCF0 (Timer 0 Compare Flag) im Register TIFR nicht gesetzt ist, also bis der Wert von !(TIFR & (1 « OCF0)) '0' ergibt. In Abfragen ist jedoch ganz besonders auf die Logik der Verknüpfungen zu achten, v.a. dann, wenn mehrere Bits abzufragen sind. Soll z.B. eine Anweisung nur dann ausgeführt werden, wenn zwei einzelne Bits in einem Register gesetzt sind, so lautet der Ausdruck dafür:

```
if((TIFR & (1 << OCF0)) && (TIFR & (1 << TOV2)))
{ /*Anweisungen*/ }
```

Es wird also überprüft, ob beide Ausdrücke TIFR & (1 « OCF0) und TIFR & (1 « TOV0) wahr, also von '0' verschieden sind. Alternativ könnte man auch schreiben

```
if((TIFR & ((1 << OCF0) | (1 << TOV2))) == ((1 << OCF0) | (1 << TOV2)))
{ /*Anweisungen*/ }
```

Diese Schreibweisen erleichtern die Programmierung und das Lesen der Programme erheblich. Man kann jetzt für jedes einzelne Steuerbit den Namen aus dem Handbuch übernehmen und das Programm so auch für andere besser lesbar machen. Der erzeugte Code wird dadurch nicht größer, da alle Operationen mit Konstanten zur Laufzeit des Compilers durchgeführt werden.

Eine **Besonderheit** ergibt sich für einige spezielle Bits in I/O-Registern, namentlich für Interrupt-Flags (s. auch Abschnitt 1.9 auf Seite 16). Um ein solches Bit zu löschen, muss man eine '1' hineinschreiben. Das Schreiben einer '0' hat hier keinen Effekt! Bei Registern, die Interrupt-Flags enthalten, ergibt sich ein besonderes Problem mit sogenannten 'Read-Modify-Write'-Operationen, also Zugriffen, bei denen der aktuelle Register-Inhalt

eingelesen, einige Bits (durch die oben angesprochenen Operationen) geändert werden und der neue Wert wieder in das betreffende Register zurückgeschrieben wird.

Das Problem besteht darin, dass z.B. aus der C-Anweisung '`TIFR |= 1 « OCF0;`' zum löschen des `OCF0`-Flags in `TIFR` in Assembler drei einzelne Befehle werden, von denen der erste den aktuellen Inhalt von `TIFR` in ein Rechenregister lädt. Der zweite Befehl führt die ODER-Verknüpfung mit `1 « OCF0` durch, der dritte schreibt den Inhalt des Rechenregisters wieder in `TIFR`. Bei diesem letzten Vorgang werden alle Bits, die vorher gesetzt waren, mit einer '1' beschrieben, was zu einem Rücksetzen aller vorher gesetzten Interrupt-Flags in dem Register führt. Register wie das `TIFR`, die nur Interrupt-Flags enthalten, schreibt man daher direkt, d.h. ohne die ODER-Verknüpfung, da das Schreiben einer '0' sowieso keinen Effekt hat[21]. Das Löschen von `OCF0` funktioniert also nur korrekt mit '`TIFR = 1 « OCF0;`'.

### 2.8.2 Bitfelder

Eine alternative Möglichkeit des Bitzugriffs besteht bei Variablen über sogenannte Bitfelder. Dazu wird die Variable als Struktur einzelner Bitfelder mit vorgegebener Länge angelegt:

```
struct{
    unsigned char bit1 : 1;
    unsigned char bit2 : 1;
    unsigned char bit3 : 1;
    unsigned char zweibits : 2;
    //...usw.
    } foo;
```

Dabei wird den Bezeichnern `bit1`, `bit2` und `bit3` je ein Bit einer `unsigned char`-Variable zugeordnet. Mit '`foo.bit1 = 1;`' kann `bit1` nun in einem Zugriff gesetzt werden. Dem Bezeichner `zweibits` sind zwei Bits zugewiesen, auf die ebenfalls wie auf eine normale Variable zugegriffen werden kann, z.B. mit '`foo.zweibits = 3`'.

Im Gegensatz zum C-Standard, der für die einzelnen Feldelemente den Typ `int` und damit für die `struct` eine Länge von 16 Bit vorschreibt, lässt der AVR-GCC-C-Compiler auch `unsigned char`-Felder zu, so dass nicht jeder Zugriff eine 16-Bit-Operation sein muss. Problematisch ist ein Zugriff auf mehrere Feldelemente gleichzeitig. Da nach dem C-Standard nicht definiert ist, in welcher Reihenfolge der Compiler die Elemente ablegt, ist es beim Zugriff z.B. mit einer `union`[22] erforderlich, vorher die Bitreihenfolge zu ermitteln. Beim AVR-GCC-C-Compiler liegt z.B. das erste Feldelement am niedrigstwertigen 'Ende' des Bytes. Ein Zugriff auf das Feld als Ganzes könnte dann so aussehen:

[21] Bei Registern, die neben Interrupt-Flags 'normale' Steuerbits enthalten, wie z.B. dem A/D-Wandler-Steuerregister `ADCSRA`, ist zu beachten, dass jeder Zugriff auf eines der anderen Bits die Interrupt-Flags löscht, es sei denn, man maskiert sie mit '0'. Bei den meisten Registern, auf die das zutrifft, ist es aber gewollt, dass das Interrupt-Flag bei einer Änderung der Einstellungen gelöscht wird.

[22] Generell ist zu beachten, dass der Zugriff auf Feldelemente in der beschriebenen Art über eine union nach dem C-Standard genaugenommen unzulässig ist, da das Ergebnis des Zugriffs nur dann definiert ist, wenn auf das zuletzt geschriebene Element zugegriffen wird. Insbesondere bei Strukturen, die mehr als ein Byte belegen, ist die Speicherstruktur des Systems zu beachten: Die AVR-Controller sind Little-Endian (sie verwenden das Intel-Format), d.h., das jeweils niedrigstwertige Byte wird zuerst (an der niedrigeren Adresse) abgelegt.

```
union{
    unsigned char bar;
    struct{
        unsigned char dreibits : 3; //Bits 2-0 (LSB)
        unsigned char bit1 : 1; //Bit 3
        unsigned char bit2 : 1; //Bit 4
        unsigned char bit3 : 1; //Bit 5
        unsigned char zweibits : 2; //Bits 7 und 6
    } foo;
} foobar;

foobar.bar = 0xEE;
```

Nach der Zuweisung liegen in den Feldelementen folgende Werte vor:

```
dreibits = 6
bit1 = 1
bit2 = 0
bit3 = 1
zweibits = 3
```

Die Abfrage eines einzelnen Elements wird dann schon etwas unübersichtlicher:

```
if(foobar.foo.bit1)
{
    //Anweisungen
}
```

Mit den im vorhergehenden Abschnitt genannten Bitschiebeoperationen ist es hingegen problemlos möglich, durch VerODERn der Bitmasken auf beliebig viele Bits in einer einzigen Anweisung zuzugreifen, ohne dass der Code zu unübersichtlich wird. Daher ist die Methode mit den logischen Operatoren (u.a. in Hinblick auf Portabilität des Codes) vorzuziehen, auch wenn sie in vielen Fällen zu mehr Schreibarbeit führt.

## 2.9 Arithmetik

Wie oben bereits angedeutet, stellen besonders für Mikrocontroller im unteren Leistungsbereich gewisse Berechnungen große Anforderungen an Rechenzeit und Ressourcen. Viele komplexe arithmetische Operationen (was die Grundrechenarten betrifft sind das v.a. Divisionen) lassen sich jedoch vermeiden oder, falls doch erforderlich, durch einfachere Operationen ersetzen. Letzteres funktioniert allerdings u.U. nur unter ganz bestimmten Voraussetzungen, also mit bestimmten Parametern.

Compiler besitzen einen Optimizer, der (falls aktiviert) bereits versucht, zu vereinfachen, wo es möglich ist. Einige Vereinfachungen kann der Optimizer allerdings nicht durchführen,

da sie seine Kompetenzen überschreiten würden[23]. Ein einfaches und häufig anzutreffendes Beispiel für einen solchen Fall ist die **Berechnung von Mittelwerten**, z.B. aus Messwertreihen. Zur Ermittlung des arithmetischen Mittelwertes einer Anzahl von Ausgangswerten ist es erforderlich, diese Werte aufzusummieren und anschließend durch die Anzahl der Werte zu dividieren.

Soll nun aus 10 Werten der Mittelwert ermittelt werden, so führt das zu einer Division durch 10, die im Mikrocontroller durch einen speziellen Algorithmus nachgebildet werden muss. Oft ist jedoch die genaue Anzahl der Werte für den Mittelwert unkritisch (z.B. bei einer einfachen Glättungsfunktion), so dass man auch eine andere Anzahl nehmen könnte. Eine Division durch 8 lässt sich z.B. durch ein Verschieben des Binärwertes um drei Stellen nach rechts nachbilden. Da jeder $\mu$C Bitschiebe-Operationen durchführen kann, ist die Division durch 8 also in sehr wenigen Taktzyklen durchführbar.

Diese Vereinfachung lässt sich für alle Divisionen durch $2^n$ durchführen, was allerdings bei Verschiebungen um größeren Stellenzahlen durch Kombinationen aus Bitschiebe-, Swap- und anderen Registeroperationen realisiert wird. Eine Division eines 8-Bit-Wertes durch 16 ließe sich z.B. rechenzeitsparend implementieren, indem man die Nibbles, also die 'Halbbytes', vertauscht (Ein 'swap'-Befehl gehört zum Standard-Befehlssatz) und anschließend das High-Nibble auf Null setzt. Noch einfacher gestaltet sich eine Division durch 256. In diesem Falle ist lediglich das Low-Byte des Ausgangswertes zu verwerfen und das High-Byte stellt das Ergebnis dar. In diesen Fällen ist leicht einzusehen, dass es Aufgabe des Programmierers ist, Vereinfachungen in diesem Sinne vorzunehmen, sofern das möglich ist. Sache des Compilers ist es dann, zu erkennen, dass eine Division durch eine Zweierpotenz durchgeführt werden soll und dementsprechend nicht die normalen Divisionsroutinen einzubauen, sondern die alternativen Algorithmen.

Entsprechend vereinfachen lassen sich auch **Modulo-Operationen mit Zweierpotenzen**, die sogar i.a. mit noch weniger Code auskommen. Dazu sollte man sich überlegen, was der Modulo-Operator eigentlich macht: Er gibt den Rest der ganzzahligen Division zurück. Bei einer Division durch $2^n$, also einem Verschieben des Binärwertes um $n$ Stellen nach rechts, ist der Rest aber genau das, was beim Schieben rechts 'herausfällt'. Und das lässt sich durch eine einfache bitweise UND-Verknüpfung des Ausgangswertes mit einer Bitmaske, in der die letzten $n$ Bits gesetzt sind, ermitteln. Eine Bitmaske, deren letzte $n$ Bits gesetzt sind, besitzt den Wert $2^n - 1$. Allgemein ist die Operation '`int_variable %` <$2^n$>' also durch '`int_variable &` <$2^n - 1$>' ersetzbar. Die bitweisen logischen Operatoren gehören wie die Schiebe-Operatoren zum Standard-Befehlssatz eines jeden Controllers, wodurch solche Berechnungen sehr effizient durchführbar sind.

Als Beispiel zur Veranschaulichung soll die Zahl 94 (in Binärdarstellung `0101 1110`) durch 16 ($2^4$) dividiert und der Rest der Division ermittelt werden. Zunächst erfolgt die Ermittlung des Restes. Bei der Division durch $2^4$ besteht der Rest aus den 4 niederwertigsten Bits. Das führt zur Maskierung mit `0b00001111` (`0x0f`), also (in Pseudo-C-Syntax) '`0b01011110 & 0b00001111 == 0b00001110`'. `0b00001110` entspricht `0x0e` oder 14 (dezimal). Die eigentliche Division erfolgt jetzt durch die Schiebe-Operation: `0b01011110 >> 4 == 0b00000101`, wobei `0b00000101` der Zahl 5 entspricht. Das Ergebnis der Rechnung ist also korrekt: $94/16 = 5$ Rest 14.

[23]Der Optimizer darf das Programm zum Zweck der Optimierung nur so weit verändern, dass diese Änderungen keine Auswirkung auf den durch den Code vorgegebenen Programmablauf haben. Der Programmierer kann also durchaus durch schlechten Programmierstil dem Optimizer die Möglichkeit nehmen, bestimmte Dinge zu vereinfachen.

In den meisten Fällen (z.B. bei einer Mittelwertbildung) ist der Divisionsrest an sich uninteressant, das Ergebnis sollte jedoch korrekt gerundet sein. Bei Berechnungen in Gleitkomma-Arithmetik kann bei einer Typkonversion in einen ganzzahligen Wert die Rundung durch Hinzuaddieren von 0,5 (vor der Konversion) erreicht werden. Bei ganzzahligen Rechenoperationen ist das jedoch nicht möglich. Hier muss das Runden bereits vor der Division durchgeführt werden, indem man die Hälfte des Divisors zum Ausgangswert hinzuaddiert, was einer Addition von 0,5 nach der Division entspricht. Im obigen Beispiel sollte bei der Division von 94 durch 16 eigentlich 6 herauskommen. Addiert man jetzt vor der Rechnung 8 zur 94 hinzu, dann erhält man das korrekt gerundete Ergebnis 6. Diese Methode funktioniert allerdings nur mit geraden Divisoren korrekt. Bei ungeraden Divisoren kann man sich u.U. behelfen, indem man die gesamte Rechnung mit zwei erweitert (also Dividend und Divisor mit 2 multipliziert). Das funktioniert allerdings auch nur dann, wenn der Wertebereich der verwendeten Datentypen ausreicht, um die Zwischenwerte darzustellen.

Oft ist es erforderlich, Binärwerte auf einen dezimalen Wertebereich zu **skalieren**, z.B. um eine Ausgabe in Prozent o.ä. zu erhalten. Dies scheint zunächst nicht ohne größeren Aufwand mit ausreichender Genauigkeit durchführbar. Allerdings besteht die Möglichkeit, auch solche Operationen zu vereinfachen und gleichzeitig eine sehr gute Genauigkeit zu erreichen, sofern die bei der Umrechnung entstehenden Zwischenwerte den Wertebereich des verwendeten Datentyps nicht überschreiten. In manchen Fällen kann jedoch selbst eine Erweiterung auf den nächstgrößeren Datentyp noch sinnvoll sein und Rechenzeit sowie Ressourcen sparen.

Als Beispiel soll hier die Umrechnung eines 10-Bit-Zahlenwertes (z.B. das Wandlungsergebnis eines A/D-Wandlers) in einen auf den Maximalwert skalierten Prozentwert umgerechnet werden. Der höchste mit 10 Bit darstellbare Dezimalwert ist 1023. Eine direkte Umrechnung in Prozent würde eine Division durch 10,23 erfordern. Das allerdings hätte wiederum Gleitkommaoperationen zur Folge. Da der Wertebereich des Datentyps `int`, in dem die Berechnung standardmäßig durchgeführt wird, jedoch genügend Platz lässt, kann die Division durch 10,23 durch eine Multiplikation mit 25 und anschließende Division durch 256 ersetzt werden. Der effektive Divisor ergibt sich damit zu 10,24 ($\frac{256}{25}$), was im Vergleich zum ursprünglichen Wert von 10,23 eine Abweichung von weniger als 0,1 % ergibt. Da die AVR-$\mu$Cs aus der ATMega-Serie allesamt über einen Hardware-Multiplier verfügen, stellt die Multiplikation mit 25 kein Problem dar. Die Division durch 256 ist, wie oben bereits erläutert, besonders einfach zu implementieren, indem einfach das High-Byte des Ausgangswertes als Ergebnis übernommen und das Low-Byte verworfen wird. Der maximale Zwischenwert bei der Rechnung beträgt $1023 \cdot 25 = 25575$ und ist damit in `int` darstellbar.

Allgemein sollte eine Division in einer aus mehreren Schritten bestehenden Berechnung immer die letzte Operation sein, da sie die Genauigkeit des Ergebnisses maßgeblich bestimmt. Gegebenenfalls sind andere Operationen und Zahlenwerte entsprechend zu erweitern, was am Beispiel der Rundung und der Skalierung oben ja bereits erläutert wurde.

## 2.10 Kommentare

Generell ist ein Programm (und das gilt nicht nur bei Mikrocontrollern) präzise zu **kommentieren**. Erfahrungen zeigen, dass selbst gestandene Programmierer bereits nach kurzer Zeit u.U. nicht mehr in der Lage sind, ein selbst verfasstes, nicht oder mangelhaft

kommentiertes Programm zu interpretieren. Kommentare werden in ANSI-C durch `/*` und `*/` eingeschlossen, d.h. alles, was zwischen diesen Zeichen steht, wird vom Compiler ignoriert. C-Compiler, die den C99-Standard erfüllen, also auch der GCC-C-Compiler, bieten auch die von C++ bekannte Methode, einzeilige Kommentare mit `//` einzuleiten. Der Kommentar geht dann nur bis zum Zeilenende. Kommentarzeichen können auch eingesetzt werden, um zu Testzwecken bestimmte Programmteile 'auszublenden'.

Nachteilig am 'Auskommentieren' von mehrzeiligen Codeblöcken mit `/*` und `*/` ist, dass sich einerseits innerhalb des Kommentars keine weiteren mehrzeiligen Kommentare befinden dürfen und andererseits das Syntax-Highlighting des Editors dafür sorgt, dass jeglicher Text innerhalb des betreffenden Blocks als Kommentar gekennzeichnet wird. Daher kann es sinnvoll sein, mehrzeilige Auskommentierungen mit den Präprozessor-Anweisungen `#if 0` und `#endif` einzuschließen. Da Code nach einem `#if` nur dann vom Compiler berücksichtigt wird, wenn die Bedingung der `#if`-Anweisung wahr ist, sorgt `#if 0` stets dafür, dass der jeweilige Code dem Compiler verborgen bleibt, während alle Funktionen des Syntax-Highlightings unbeeinflusst bleiben und auch mehrzeilige Kommentare innerhalb des Blocks unproblematisch sind.

# 3 Ein- und Ausgabe mit dem Mikrocontroller

In vielen Fällen ist es erforderlich, dass der Controller mit einem Bediener kommunizieren kann, also einerseits von außen Anweisungen entgegennehmen (z.B. über Eingabetasten) und andererseits dem Bediener Informationen zukommen lassen kann (z.B. über optische Anzeigen). Einige Möglichkeiten für ein 'Mensch-Maschine-Interface' werden in den folgenden Abschnitten erläutert.

## 3.1 Eingabe mit Tastern oder Schaltern

Um von außen Einfluss auf das Programm nehmen zu können kann man Taster bzw. Schalter an den Controller anschließen. Dabei sollten einige Punkte beachtet werden, die ansonsten zu Fehlfunktionen und unerwünschten Effekten führen können.

### 3.1.1 Anschließen mechanischer Kontakte an den $\mu$C

Ein Taster oder anderer Schaltkontakt an einem Mikrocontroller muss in der Lage sein, vom Controller auswertbare Pegel zur Verfügung zu stellen. Bei den meisten Tastern handelt es sich um einfache Schließer-Kontakte, für deren Anschluss an einen Controller es prinzipiell zwei Möglichkeiten gibt, nämlich Low-Side (also zwischen $\mu$C-Pin und GND) mit Pull-Up-Widerstand oder High-Side (zwischen $\mu$C-Pin und positiver Versorgungsspannung) mit Pull-Down-Widerstand. Die Widerstände sind erforderlich, um im unbetätigten Zustand des Tasters für einen definierten Pegel am (hochohmigen) Eingangspin zu sorgen. Da die AVR-Controller an allen I/O-Ports über einzeln aktivierbare Pull-Up-Widerstände verfügen und die erstgenannte Methode demnach ohne zusätzliche externe Komponenten realisierbar ist, wird diese auch in praktisch allen Fällen zum Einsatz kommen. Ein betätigter Taster führt dann zu einem Low-Pegel am entsprechenden Pin.

### 3.1.2 Entprellung

Praktisch alle mechanischen Kontakte besitzen die unangenehme Eigenschaft, dass sie **prellen**, also beim Schaltvorgang nicht sauber von einem Zustand in den anderen wechseln. Da eine digitale Schaltung, die die Zustände der Kontakte auswerten soll, definierte Pegel benötigt, um den Zustand eindeutig zu erkennen, führt diese Übergangsphase meist zu Problemen bei der Auswertung. Um dies zu verhindern müssen Taster entprellt werden, was durch Hard- und/oder Software geschehen kann. Z.B. kann ein Kondensator parallel zum Schaltkontakt genutzt werden, der die relativ hochfrequenten Schwingvorgänge abblockt. Eine weitere, in diskreten Logikschaltungen häufig verwendete Methode der Hardware-Entprellung beruht auf der Verwendung von Flip-Flops.

Bei Mikrocontroller-Systemen ist es hingegen meist nicht erforderlich, eine Hardware-Entprellung vorzusehen, da man diese auch per Software mit vergleichsweise geringem Aufwand implementieren kann. Bedenkt man die Tatsache, dass mechanische Kontakte und (im Falle von Tastern) deren Bediener recht träge sind und es wenig Sinn macht, einen Taster oder Schalter permanent zu überwachen und sofort (nach Controller-Maßstäben, also innerhalb von $\mu$s) auf eine Änderung zu reagieren, dann leuchtet es ein, dass eine Abfrage alle paar 10 ms völlig ausreicht. Die Prellvorgänge liegen i.d.R. im Bereich unter

1 ms. Eine sinnvolle Methode zum Erfassen der Tasterzustände ist das Einlesen in einer zyklisch aufgerufenen Funktion, die z.B. auf einem Timertakt basiert.

Da für die Auswertung von Tastern meist nur die Änderung des Zustandes gegenüber der letzten Abfrage interessiert, genügt es i.d.R., den jeweils aktuellen Zustand mit einem Exklusiv-ODER mit dem vorherigen zu verknüpfen. Ist es darüber hinaus nicht erforderlich, ein Loslassen der Taster zu detektieren, kann man über eine UND-Verknüpfung (bei Low-Side angeschlossenen Tastern mit dem Bitkomplement des aktuellen Zustandes) diejenigen Taster ermitteln, die seit der letzten Abfrage betätigt wurden. Es ist leicht nachzuvollziehen, dass sich dadurch ohne großen Aufwand und mit kompaktem Code die gesamte Prell-Problematik erübrigt.

In Anwendungsfällen, bei denen so viele Taster benötigt werden, dass nicht mehr genügend I/O-Pins am Controller zu Verfügung stehen, um jeden Kontakt einzeln anzuschließen, ist es sinnvoll, die Taster entweder in einer Matrix anzuordnen oder sie an Schieberegister anzuschließen, die dann seriell eingelesen werden können. Eine dritte Möglichkeit ist das Einlesen der Taster mit dem Analog-Digital-Wandler, wobei man die Taster mit Widerständen in einem R2R-Netzwerk, wie es von Digital-Analog-Wandlern bekannt ist, verschaltet. Bei dieser Variante ist es auch problemlos möglich, bei Betätigung mehrerer Kontakte gleichzeitig den Zustand korrekt einzulesen. Auch eine einfache Reihenschaltung von Widerständen, von der ein Teil jeweils von einem Schaltkontakt gebrückt wird, stellt eine einfache Möglichkeit dar, Taster über den Analog-Digital-Wandler einzulesen, wobei hier jedoch Mehrfachbetätigungen u.U. nicht mehr problemlos erkennbar sind.

## 3.2 Anzeige von Zuständen und Zahlenwerten

### 3.2.1 Ausgabe über LEDs

Die einfachste Möglichkeit, einem Anwender Informationen über Vorgänge im System zu geben ist eine Statusanzeige über entsprechende Leuchtmittel. Als Leuchtmittel kommen in der Elektronik i.d.R. LEDs zum Einsatz, da diese bei entsprechend niedrigen Spannungen arbeiten und vergleichsweise wenig Strom benötigen. Die I/O-Pins der AVR-Controller sind als Ausgänge in der Lage, Standard-LEDs direkt zu treiben. Die Push-Pull-Ausgangstreiber der Portpins sind weitgehend symmetrisch und können den selben Strom 'sourcen' und 'sinken', es kann also sowohl Strom 'aus dem Pin heraus' fließen (bei High-Pegel am Ausgang) oder 'in den Pin hinein' (bei Low-Pegel)[24]. Beim Anschluss von Lasten an Controller-Pins ist aber darauf zu achten, dass ein einzelner Pin zwar (lt. Spezifikationen) bis zu 40 mA treiben kann[25], es jedoch auch Grenzwerte für jeden Port an sich und für die VCC- und GND-Anschlüsse am $\mu$C gibt. Schließlich muss ein Strom, der durch einen I/O-Pin in den Controller hineinfließt, über den GND-Anschluss wieder hinausfließen können. Umgekehrt muss ein Strom, der aus einem Pin hinausfließen soll, über den VCC-Anschluss hineingelangen können. Außerdem besitzen die Transistoren der

[24] Andere $\mu$C-Familien (z.B. die meisten 'älteren' 8051er), die keine Push-Pull-Stufen an den Ausgängen besitzen, sondern nur Open-Collector-Ausgänge, können nur Strom 'sinken', d.h. Lasten müssen extern gegen die positive Versorgungsspannung angeschlossen werden und ein Strom kann nur fließen, wenn der Ausgang Low-Pegel hat, also 'in den Pin hinein'.

[25] Allerdings sind 40 mA schon der absolute Maximalwert (im Datenblatt unter 'Absolute Maximum Ratings' nachzulesen), der zu keiner Zeit überschritten werden sollte. Das Datenblatt sagt dazu, dass eine längere Belastung mit diesem Maximalwert 'die Zuverlässigkeit des Bausteins beeinträchtigen kann'.

Ausgangstreiber auch im aufgesteuerten Zustand einen Widerstand, der Verlustleistung und damit Wärme verursacht, wenn ein Strom über ihn fließt.

Möchte man zu Anzeigezwecken so viele LEDs anschließen, dass die Spezifikationen der Porttreiber aus einem der o.g. Gründe nicht eingehalten werden können, kann man entweder auf Low-Current-LEDs ausweichen (die allerdings erheblich lichtschwächer sind als Standard-LEDs), die Last anders verteilen oder zusätzliche externe Treiber einsetzen. Bei einer Überschreitung der Maximalströme von GND- oder VCC-Anschlüssen kann man sich mit einem Trick behelfen, indem man z.B. einen Teil der LEDs High-Side (also gegen VCC) anschließt und den anderen Teil Low-Side (also gegen GND), so dass sich die Source- und Sink-Ströme aufteilen. Der eine Teil ist dann gegenüber dem anderen invertiert anzusteuern.

### 3.2.2 Ausgabe von Zahlenwerten

Zur Ausgabe von Zahlenwerten existiert eine Vielzahl von Möglichkeiten. Zahlen können z.B. direkt in Binärdarstellung über LEDs ausgegeben werden oder, weniger 'abstrakt', über eine LED-Balkenanzeige. Eine präzise und leicht ablesbare Ausgabe auch größerer Zahlenwerte ist allerdings nur über eine entsprechende numerische Anzeige möglich. Dabei gibt es wiederum eine Vielzahl von Varianten, von der 'einfachen', direkt angesteuerten 7-Segment-Anzeige über Punktmatrix-Displays bis hin zu alphanumerischen LC-Displays (mit ASCII-Textzeichendarstellung), die i.d.R. über einen eigenen Controller verfügen und über serielle oder parallele Interfaces mit dem Controller kommunizieren können. Letztere benutzen bestimmte Übertragungsprotokolle, deren Beschreibung den entsprechenden Datenblättern zu entnehmen ist.

Die Ansteuerung von 7-Segment-LED-Anzeigen ist oft direkt über den Controller möglich. Für mehrstellige Displays bieten sich im Prinzip zwei Verfahren an:

1. Ansteuerung über externe Treiber-Bausteine mit integrierten Latches und BCD-Decodern
2. Multiplex-Steuerung direkt durch den Controller

Dabei ist das erste Verfahren auf der Software-Seite am einfachsten zu realisieren, was allerdings einen größeren Aufwand an externer Zusatzhardware mit sich bringt (für jede Display-Stelle ist ein entsprechender Treiberbaustein, z.B. CMOS 4543, und für jedes Segment ein Vorwiderstand erforderlich), während beim Multiplex-Verfahren der Hardware-Aufwand vergleichsweise gering ist (i.d.R. nur ein Satz Vorwiderstände und für jede Stelle ein Treibertransistor für den gemeinsamen Anschluss), jedoch die Ansteuerung um einiges komplexer.

Zur **Ansteuerung über externe Treiberbausteine** (z.B. den o.g. 4543) gibt der Controller auf vier Ausgängen die darzustellende Ziffer als BCD- (Binary Coded Decimal-) Wert aus und erzeugt am Latch-Eingang des Treibers einen kurzen Übernahme-Impuls[26].

[26]Beim 4543 ist der betreffende Eingang mit 'Latch Disable' bezeichnet. Liegt dort ein High-Pegel vor, dann ist das Latch deaktiviert und eine Änderung an den Dateneingängen führt zu einer sofortigen Änderung an den Segment-Ausgängen, während bei Low-Pegel die Segment-Ausgänge 'eingefroren' sind. Der Pin ist also auf Low zu halten und nur bei einer Aktualisierung für kurze Zeit High zu schalten, nachdem die Dateneingänge entsprechend gesetzt sind.

Dabei wird der eingegebene Wert decodiert an die Segment-Ausgänge des Treibers weitergegeben und bleibt bis zum nächsten Übernahme-Impuls 'eingefroren'. Anschließend können die anderen Stellen ausgegeben werden. Am Controller werden dafür vier Ausgänge für die Datenbits und ein Übernahme-Ausgang für jede Stelle benötigt, bei einem dreistelligen Display also 7 Portpins. Eine Aktualisierung des Display-Wertes ist, bedingt durch die in den Treibern vorhandenen Latches, nur im Falle einer Änderung erforderlich.

Möchte man führende Nullen bei einer Zahlendarstellung unterdrücken, so genügt es bei einigen Treibern (z.B. beim oben erwähnten 4543), einen nicht als Dezimalziffer darstellbaren Wert (also eine Binärzahl größer als 9) auszugeben, um die Display-Stelle 'blank' zu schalten. Eine separate Ansteuerung des 'Blanking Input' des 4543, die den gleichen Effekt hätte, ist also nicht erforderlich. Einige Display-Treiber können allerdings auch die Zeichen A, b, C, d, E und F ausgeben und sind daher nicht für dieses Verfahren geeignet.

Beim **Multiplex-Verfahren** werden die Segmente direkt oder (bei höherem Strombedarf) über entsprechende Treiber von Controller-Ausgängen angesteuert. Dabei ist immer nur eine Display-Stelle eingeschaltet (über den entsprechenden gemeinsamen Anschluss der Segmente der betreffenden Stelle), während an den Segment-Anschlüssen die auszugebenden Bitmuster anliegen. Die Display-Stellen werden nun der Reihe nach ein- und wieder ausgeschaltet, jeweils mit dem dazugehörigen Bitmuster an den Segmentanschlüssen. Die Frequenz, mit der die Ausgabe geschieht, muss so hoch sein, dass die Anzeige nicht flimmert (also mindestens im Bereich einiger hundert Hz, wo für das menschliche Auge kein Flimmern mehr wahrnehmbar ist).

Dieses Steuerverfahren führt dazu, dass z.B. bei einer dreistelligen Anzeige jede einzelne Stelle nur ein Drittel der Zeit eingeschaltet und den Rest der Zeit nicht aktiv ist. Dadurch sinkt bei gleichem Strom durch die LEDs (also bei gleichem Vorwiderstand) die scheinbare Helligkeit der Anzeige entsprechend, was u.U. nicht mehr ausreicht, um die Anzeige gut ablesen zu können. Meist erhöht man dann (durch die Wahl kleinerer Vorwiderstände) den Betriebsstrom, um im Mittel wieder den ursprünglichen Strom und damit auch die entsprechende Helligkeit zu erreichen. Da der höhere Strom jeweils nur kurzzeitig fließt, stellt dies für die LEDs auch kein Problem dar, da diese kurzzeitig einen erheblich über dem zulässigen Dauerstrom liegenden Strom vertragen und der Mittelwert nach wie vor im 'normalen' Bereich liegt.

Es ist leicht nachvollziehbar, dass die Ansteuerung mittels Multiplex-Verfahren viel höhere Anforderungen an die Software und den Controller stellt, da das Display ständig aktualisiert werden muss, auch wenn kein neuer Anzeigewert vorliegt. Außerdem erfordert ein dreistelliges Display in diesem Fall bereits 10 Controller-Pins[27]. Hinzu kommt noch, dass die maximale Strombelastung der Controller-Pins bei mehr als zwei Display-Stellen (bei Verwendung normaler, also nicht-low-Current-Displays) i.d.R. bereits erreicht ist und externe Treiber (z.B. Transistoren) erforderlich sind (wodurch der Vorteil des geringeren Hardware-Aufwandes gegenüber der o.g. Variante mit Treiberbaustein hinfällig ist). Auch die maximale Impulsstrombelastbarkeit der LEDs setzt Multiplex-Displays höherer Stellenzahl Grenzen.

[27]...wobei es im Prinzip möglich wäre, die gemeinsamen Anschlüsse der Stellen über einen Multiplexer anzusprechen und so Controller-Pins zu sparen, was allerdings wieder zu einem größeren Aufwand an externer Hardware führt.

# 4 Installation und Anwendung der Software-Tools

Bei einer Neuinstallation der Software-Tools sind einige Punkte zu beachten, um ein fehlerfreies Funktionieren der Software zu gewährleisten. Zunächst sollte AVRStudio und (falls vorhanden) das jeweils aktuelle Service-Pack installiert werden. Die jeweils aktuellste Version ist unter der im Abschnitt 4.5 aufgeführten Adresse verfügbar. Installiert man nun WINAVR, so wird der GNU-C-Compiler von AVRStudio automatisch als Plugin erkannt.

## 4.1 Erstellen und Compilieren des Programms mit AVRStudio und WINAVR

Die Programmierung des ATMega16 mit AVRStudio und WINAVR geschieht in mehreren Schritten. Zunächst ist in AVRStudio ein Projekt zu erstellen. Beim Starten der IDE erscheint das 'Welcome'-Fenster. Der Button 'New Project' führt in einen Dialog, in dem zunächst die Art des Projekts anzugeben ist. Falls WINAVR korrekt installiert ist, erscheint dort der Eintrag 'AVR-GCC'. Nach dessen Anwahl ist rechts der Projektname einzugeben. Falls bereits eine Quelldatei vorliegt, ist das Häkchen im Feld 'Create initial file' zu entfernen, da ansonsten eine Datei <Projektname>.c angelegt wird. Ein Ordner, in dem die Projektdatei und die dazugehörigen Dateien abgespeichert werden sollen, ist unter 'Location' anzugeben. Es ist sinnvoll, einen eigenen Ordner für das Projekt anzulegen. Anschließend ist der Button 'Finish' zu betätigen. Bereits vorhandene Quelldateien sollten nun in den Projektordner kopiert werden.

Nach diesen Anfangsschritten erscheint das Hauptfenster der IDE. Auf der linken Seite befindet sich der Project Explorer (Tab 'AVR-GCC'), in dem alle zum Projekt gehörenden Dateien dargestellt sind. Ein Klick mit der rechten Maustaste auf den Ordner 'Source Files' öffnet das Kontext-Menü. Mit 'Add Existing Source File(s)...' können nun bereits vorhandene Quelldateien zum Projekt hinzugefügt werden. Im 'Project'-Menü unter 'Configuration Options' sind noch einige Einstellungen vorzunehmen. Im Unterpunkt 'General' ist im Drop-Down-Menü 'Device' der ATMega16 auszuwählen[28] und die Taktfrequenz von 8 MHz einzustellen. Das Optimierungslevel sollte '-Os' sein, um einen möglichst platzsparenden Code zu erzeugen. Die anderen Levels ('-O1' bis '-O3') sind Kompromisse zwischen Codegröße und Geschwindigkeit, wobei '-O3' den schnellsten Code erzeugt. -O0 bedeutet, dass keine Optimierungen seitens des Compilers vorgenommen werden, was zu erheblich aufgeblähtem Code führen kann. Zusätzlich muss das Häkchen 'Generate Hex File' unten im Fenster gesetzt sein, da sonst keine .hex-Datei zur Programmierung des Controllers erzeugt wird.

Ebenfalls im Dialog 'Configuration Options' im Unterpunkt 'Custom Options' sollte noch die Option '-std=gnu99' hinzugefügt werden, um den C99-Standard für den Compiler zu aktivieren. Dazu im linken Fenster '[All files]' auswählen, in die Eingabezeile '-std=gnu99' eingeben und mit 'Add' zur Liste hinzufügen. Bei Verwendung von Gleitkommaberechnungen (dazu gehören auch die zur Erzeugung von Wartezeiten nutzbaren _delay_XX-Funktionen aus der util.h!) im Programm ist zusätzlich unter '[Linker Options]' die Option '-lm' einzutragen, damit die korrekte mathematische Bibliothek dazugelinkt wird.

[28] ...falls bei der Erstellung des Projektes noch kein Controllertyp in der Debugger-Auswahl angegeben wurde...

Ein Doppelklick auf die zuvor eingefügte Quelldatei im Project Explorer öffnet diese im rechts befindlichen Editor-Fenster. Das Programm kann nun ergänzt werden. Betätigt man die Taste F7 oder wählt den Befehl 'Build (active configuration)' im Menü 'Build' oder in der Befehlsleiste oberhalb des Editor-Fensters aus, werden automatisch die zum Projekt gehörenden Quelldateien compiliert und assembliert/gelinkt. Fehlermeldungen erscheinen im 'Build'-Fenster unterhalb des Editors.

Die 'Syntax Highlighting'-Funktion des Editors hebt Schlüsselwörter (Datentypen, Anweisungen) farblich hervor, um das Programm besser lesbar zu machen. Benutzt man allerdings z.B. Datentypen aus der stdint.h, werden diese nicht hervorgehoben, da sie im C-Standard nicht als Schlüsselwörter bekannt sind. Ist eine Hervorhebung jedoch erwünscht, dann kann man diese der entsprechenden .ini-Datei hinzufügen. Die betreffende Datei 'AvrStudio_c.ini', die für die C-Syntax zuständig ist, befindet sich im Installationsverzeichnis unter '/AVR Tools/AvrStudio4/edit'. Die Liste der Schlüsselwörter befindet sich am Ende der Datei. Fügt man z.B. '`uint8_t = Keyword`' hinzu, so wird `uint8_t` im Editor ebenfalls blau dargestellt. Es kann sinnvoll sein, vor dem Ändern der .ini-Datei eine Kopie der Originaldatei unter anderem Namen zu sichern.

Es ist beim Compilieren darauf zu achten, dass nicht alle Fehler auch wirklich entsprechende Fehlermeldungen verursachen. Nur bei wirklich fatalen Fehlern erscheinen im Build-Fenster die entsprechenden Fehlermeldungen (mit roten Punkten gekennzeichnet) und die Meldung 'Build failed with X errors and Y warnings... '. Es gibt jedoch einige Fehler, die lediglich eine Warnmeldung des Compilers oder Linkers hervorrufen. Diese Fehler, die für die Funktion des Programmes u.U. durchaus fatal sein können, erzeugen lediglich die entsprechenden Meldungen (mit gelben Punkten gekennzeichnet), haben jedoch keine Auswirkungen auf den Build-Prozess, so dass am Ende die Meldung 'Build succeeded with Y warnings...' erscheint. Da in diesem Falle jedoch die Information über die Speicherauslastung ausgegeben wird, verschwindet die Liste der Warnungen je nach Einstellungen der Fensteraufteilung über dem oberen Rand des Build-Fensters. Um keine Warnungen zu übersehen, sollte man auf jeden Fall auf die Angabe in der letzten Zeile achten.

Eine weitere Unschönheit ist, dass auch bei Überschreitung der verfügbaren Speicherressourcen derzeit keine Warnungen oder Fehlermeldungen ausgegeben werden. Vergisst man z.B., die Optimierung zu aktivieren, so kann es (v.a. bei Programmen mit komplexeren Funktionen) passieren, dass mehr Code erzeugt wird, als in den Speicher des betreffenden Controllers hineinpasst. Im Build-Fenster steht dann lediglich eine Meldung wie z.B.

```
Program:     17,894 bytes (113% Full)
```

## 4.2 Programmieren des Controllers mit PonyProg

Das Programmieren des Controllers übernimmt die Programmiersoftware 'PonyProg'. Diese benutzt die serielle PC-Schnittstelle (RS232), um die SPI-Schnittstelle, über die die AVR-Controller ihre Programmdaten austauschen, zu 'emulieren'. Auf dem Mikrocontrollerboard ist zu diesem Zweck ein Pegelwandler vorhanden, der die Signalpegel der RS232 an die 5 V auf dem Board anpasst. Nach dem ersten Aufruf von PonyProg nach der Installation wird der Benutzer aufgefordert, zunächst das Setup und die Kalibrierung durchzuführen. Die Setup-Konfiguration ist 'serial Port' (freie Schnittstelle auswählen) und 'SI-Prog I/O' oder 'SI-Prog API', wobei letzteres schneller ist, allerdings nicht auf

jedem Rechner funktioniert. Nach dem Setup und der Auswahl des Controllers ATMega16 in der Befehlsleiste oder im Menü 'Device' kann die von AVRStudio erzeugte Datei 'motorsteuerung.hex', die sich im Unterordner 'default' des Projektordners befindet, über 'Open Program Memory (FLASH) File' geladen werden. Vor dem eigentlichen Programmiervorgang ist das Schnittstellenkabel an das Mikrocontroller-Board anzuschließen und die Versorgungsspannung ist einzuschalten. Mit 'Write Program Memory (FLASH)' wird die .hex-Datei zum Controller übertragen. Anschließend verifiziert PonyProg den Inhalt des Programmspeichers und gibt eine Meldung aus (im besten Fall 'Write successful', ansonsten 'Write failed').

Vor der Inbetriebnahme der Schaltung zur Ausführung des Programms sollte das Schnittstellenkabel entfernt werden, um Störungen zu vermeiden.

## 4.3 Sonstige Tools

Zur Suche von logischen Fehlern im Programm und zum Testen ohne Hardware enthält AVRStudio einen Debugger/Simulator, der benutzt werden kann, um das Projekt zu simulieren. Der Simulator hat allerdings einige Fehler, auch solche die in der Hilfe unter 'Known issues' nicht aufgeführt sind, und ist somit mit Vorsicht anzuwenden. Hat man das Projekt mit 'Build' kompiliert, so kann man mit dem 'Run'-Button oder über das Menü 'Debug' den Simulator starten. Details zur Benutzung des Debugger/Simulators sind der Hilfe zu entnehmen.

## 4.4 Weitere Informationen zur AVR-Familie von ATMEL

Der in dieser Dokumentation beschriebene ATMega16 ist nur ein Beispiel aus einer mittlerweile sehr vielseitigen Familie von Mikrocontrollern. Es existiert ein breites Spektrum in Sachen Größe und Leistungsfähigkeit. Grundsätzlich lassen sich die AVR-Controller grob in drei Klassen einteilen:

- ATTiny-AVR (Typ-Bezeichnung: ATTiny<xxxx>): Die 'kleinen' General-Purpose-AVRs. Programmspeicher von 1 KiB bis 8 KiB, kein Hardware-Multiplier, 6 bis 18 I/O-Pins
- ATMega-AVR (ATMega<xxxx>): Die 'großen' General-Purpose-AVRs. 4 bis 256 KiB Programmspeicher, Hardware-Multiplier, bis zu 86 I/O-Pins
- AVRs mit Sonderfunktionen:
  - USB-AVR (AT90USB<xxxx>): AVRs mit integriertem USB-Host-Controller
  - CAN-AVR (AT90CAN<xxxx>): AVRs mit CAN-Interface
  - Lighting AVR (AT90PWM<xxx>): AVRs zur Ansteuerung von Vorschaltgeräten für Beleuchtungszwecke sowie für Motorsteuerungen, ausgestattet mit sog. Power Stage Controllers, die spezielle Timer mit jeweils mehreren PWM-Kanälen besitzen
  - LCD AVR (ATMega<xxxx>): AVRs aus der ATMega-Reihe mit integrierten LCD-Segmenttreibern

– Smart Battery AVR (ATMega<xxxx>): AVR zum Aufbau von 'intelligenten' Batterieladegeräten

Einige Mega-, Tiny- und CAN-AVRs sind auch in einer 'Automotive'-Version mit erweitertem Betriebstemperaturbereich für den Einsatz in der Kfz-Elektronik erhältlich. Die 'klassischen' AVRs der AT90S-Reihe (nicht zu verwechseln mit den anderen AT90-Typen) sind mittlerweile abgekündigt. Für die meisten Typen aus dieser Reihe existieren pinkompatible Nachfolger in den Reihen ATTiny und ATMega.

Die ein- bis vierstellige Nummer in der Typbezeichnung der AVRs enthält auch die Größe des Programmspeichers (in KiB) in Gestalt der ersten 1 bis 3 Ziffern. Da die Speichergrößenangabe immer eine Zweierpotenz ist, kann man sie eindeutig ablesen, indem man die ersten Ziffern der Nummer zu einer Zahl zusammenfasst, bis sich eine Potenz von zwei ergibt. Ergibt sich dann durch Hinzufügen einer weiteren Ziffer eine Zahl, die keine Zweierpotenz ist, dann ist die zuletzt hinzugefügte Ziffer kein Bestandteil der Größenangabe mehr. Beim ATMega3250 z.B. bilden die ersten beiden Ziffern die Zahl 32, also eine Zweierpotenz. 325 ist jedoch keine Zweierpotenz mehr, wodurch sich die Größe des Programmspeichers des ATMega3250 zu 32 KiB ergibt, während z.B. der ATTiny15 1 KiB Flash besitzt.

## 4.5 Bezugsquellen

- WINAVR-Compiler-System (enthält GNU GCC Compiler) und Tools http://sourceforge.net/projects/winavr/
- AVRStudio Software (Entwicklungsumgebung mit Assembler und Programmiersoftware): http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2725
- Programmiersoftware PonyProg: http://www.lancos.com/prog.html
- Handbuch Mikrocontroller ATMega 16 (8-bit RISC-Controller mit 16 kB Flash Memory): http://www.atmel.com/dyn/products/product_card.asp?part_id=2010

Unter http://www.mikrocontroller.net/articles/AVR-GCC-Tutorial befindet sich ein detailliertes Tutorial zur Programmierung von ATMEL AVR-Mikrocontrollern mit WINAVR bzw. AVR-GCC sowie diverse Foren und Artikelsammlungen. Diese können herangezogen werden, wenn das vorliegende Dokument Fragen offen lässt.

# Literatur


[1] Datenblatt Mikrocontroller ATMega16, ATMEL Corp., 08.2007

[2] Brian W. Kernighan, Dennis M. Ritchie: Programmieren in C, 2. Ausgabe, Hanser Fachbuch, 1990, ISBN 3-446-15497-3

[3] Die Programmiersprache C – ein Nachschlagwerk, RRZN Hannover

[4] Dokumentation der AVR-libc, im WINAVR-Bundle enthalten