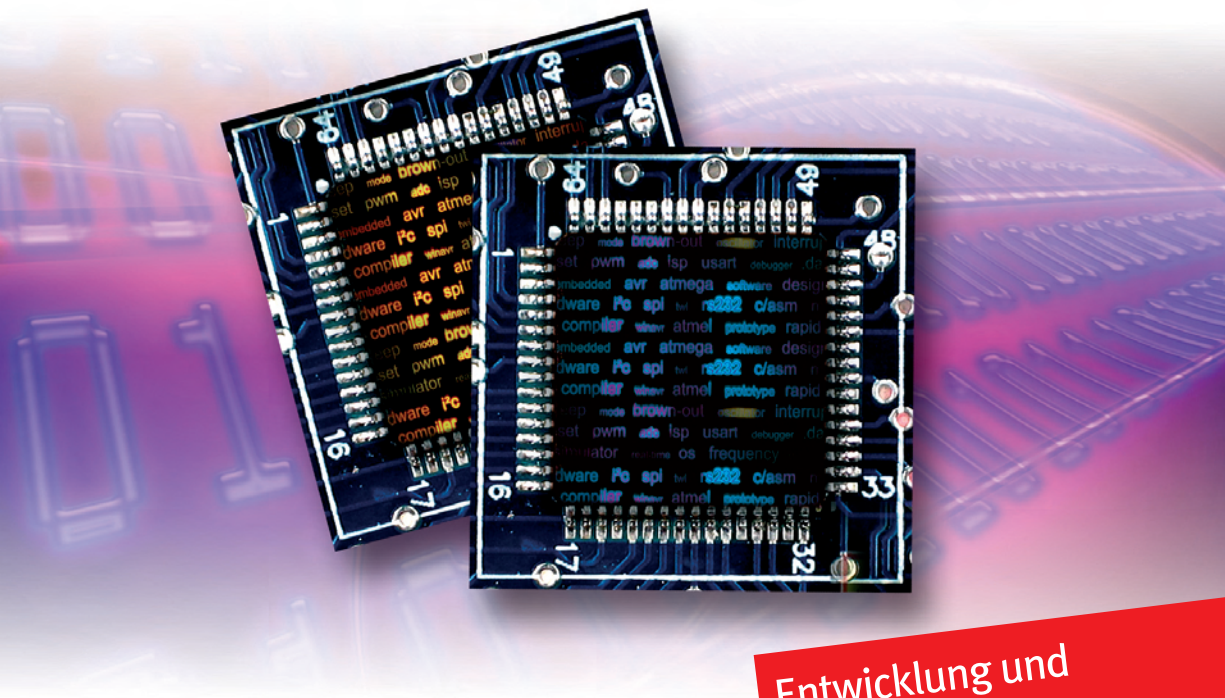
FRANZIS
PC & ELEKTRONIK

Roman Mittermayr



Entwicklung und Programmierung in Assembler und C

AVR-RISC

Embedded Software selbst entwickeln

Auf CD-ROM:

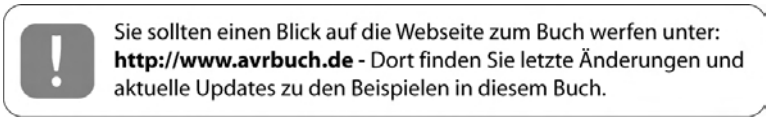
- AVR Studio 4.13 und WinAVR
- Praxisbeispiele
- Schaltungen

Vorwort

Ganz gleich, ob Sie das Wort *Mikrocontroller* heute zum ersten Mal hören oder seit fünf Jahren „Embedded Software“ schreiben: Dieses Buch hilft Ihnen beim Erlernen, Umsteigen und Vertiefen Ihres Wissens zu diesem Thema. Es wird prägnant erklärt, worauf es bei der Softwareentwicklung mit AVR-Prozessoren wirklich ankommt.

Die 8-Bit-Mikrocontroller-Serie von ATMEL erlaubt Ihnen eine rasche Verwirklichung verschiedenster Ideen und Projekte. Die schnelle RISC-Architektur, der einheitliche Aufbau der kompletten Prozessorfamilie und nicht zuletzt die attraktiven Preisangebote für Hobbyentwickler lassen diesen Controller die richtige Wahl sein.

Ich hoffe, dass Ihnen dieses Buch nicht nur Wissen verschafft, sondern auch Freude bereitet. Um Software zu entwickeln, braucht man einen klaren Kopf. Gönnen Sie sich eine Pause, wenn Sie einmal nicht vorankommen sollten. Ich wünsche Ihnen eine Vielzahl an erfolgreichen und spannenden Projekten und stehe Ihnen für Fragen zu diesem Buch gerne über meine Webseite http://www.avrbuch.de zur Verfügung.



Alle Preise in diesem Buch sind ohne Gewähr. Bitte beachten Sie, dass es sich bei vielen Produktnamen und Unternehmen um eingetragene Warenzeichen handelt.

Mein Dank gebührt einer Reihe von Menschen, ohne deren Hilfe dieses Buch nur eine Idee geblieben wäre. Dazu gehören Alexander Huwaldt von der Firma Laser & Co. Solutions GmbH (www.myAVR.de) und Benedikt Sauter (www.embedded-projects.net) für die großartige Unterstützung. Großer Dank gilt auch den wahren Helden der AVR-Community: Jörg Wunsch und Eric Weddington. Ohne die beiden wäre die Softwareentwicklung am AVR nicht mit dem Komfort möglich, der heute gegeben ist. Sehr hilfreich war die Arbeit von Maria Mittermayr, die mich beim Verfassen des Buchs unterstützt hat. Anke Mündler hat das Projekt mit wesentlichen

Verbesserungsvorschlägen und Änderungen bereichert. Die Schaltungen stammen von Walter Mittermayr. Dank in diese Richtung für die großartige technische Unterstützung beim Aufbau und der Entwicklung der Praxisanwendungen.

Roman Mittermayr
Oktober 2007, Wien

Inhalt

4 Eine Einführung in C

4.1 Was ist C?

Die Programmiersprache C wurde 1972 in den Bell Labors von Dennis Ritchie entwickelt. Ursprünglich für den Entwurf des Betriebssystems UNIX gedacht, entstanden im Laufe der Zeit weitere unterschiedliche Versionen von C.

Aus diesem Grund hat sich ANSI (American National Standards Institute) 1983 dazu entschieden, einen weltweiten Standard für C festzulegen. Dieser wurde folglich auch als *ANSI C* bekannt. Die meisten modernen C-Compiler halten sich an diesen Standard.

Einer der wesentlichsten Vorteile von C besteht in der großen Verbreitung von C-Compilern auf verschiedensten Systemen, geeignet für eine Vielzahl von Prozessoren. Dies ermöglicht, (meist) ein und denselben Code ohne große Anpassungen auf ein neues System zu portieren. Der Vorteil liegt auf der Hand: Frei erhältlicher Quellcode, gebunden an meist kostenlose, „faire" Lizenzmodelle (GPL, BSD, Beer-Ware, ...). Vor allem in Bereichen, in denen Open-Source eine große Bedeutung darstellt, stehen eine Fülle von Quellcodes zur Verfügung.

In langer Arbeit hatten Entwickler die Treibersoftware für einen weit verbreiteten Ethernet Netzwerk-Chip unter Linux programmiert. Jahre später wurde dieser Treiber in nur wenigen Stunden auf die ATMEL-Plattform portiert. Dies war großteils durch die gute Portierbarkeit des C-Quelltextes so schnell möglich.

Mit Assembler „sprechen" Sie direkt mit dem Prozessor, während Sie in C lediglich eine „Hochsprache" sprechen, die dann von einem Übersetzer (Compiler) für den gewünschten Prozessor übersetzt wird. Um mit anderen Prozessoren zu sprechen, müssen Sie dazu nur den verwendeten Compiler auswechseln und einen anderen Compiler (der jeweils gewünschten Plattform) mit Ihrem Quellcode füttern.

Dies klingt in der Praxis so revolutionär wie die Erfindung des Telefons, in Wirklichkeit gibt es natürlich einige Tücken auf dem Weg zum „portablen" Code. Mehr dazu finden Sie auf den folgenden Seiten.

Kapitel 1 beinhaltet detaillierte Informationen zur Installation des Compilersystems.

4.2 Das Hauptprogramm

Zu diesem Zeitpunkt gehen wir davon aus, dass Ihr Compilersystem installiert ist und Ihr Entwicklungsboard (oder der Simulator) erfolgreich verbunden und getestet wurde.

Leider müssen wir hier mit der Tradition brechen und beginnen daher nicht, wie üblich, mit dem populären *Hello-World*-Programm. Stattdessen beschränken wir uns auf die einfache Schaltung eines ganzen Ports, also *Spannung aus*, *Spannung an*.

```
1:   #include <avr/io.h>
2:   int main(void)
3:   {
4:         DDRB = 0xFF;
5:         PORTB = 0xFF;
6:         PORTB = 0x00;
7:         return 0;
8:   }
```

Nachdem Sie diesen Quelltext kompiliert haben, müssen Sie anschließend die erzeugte HEX-Datei in den Mikrocontroller (oder den Simulator) flashen. Informationen dazu finden Sie in Kapitel 1. Nach einem Reset startet das Programm und Sie werden am Entwicklungsboard kaum eine Veränderung bemerken.

Sollten Sie eine LED-Reihe an PORT B angeschlossen haben, wird diese nach Ausführung des Programms leuchten (bei einer Pull-Up-Konfiguration, ansonsten wird die LED-Reihe nur kurz aufblinken). Im Simulator sehen Sie schrittweise die Veränderung des Portzustands von *0xFF = alle Pins hochohmig* auf *0x00 = alle Pins schalten durch*. Soviel zur Funktion dieses Beispiels. Nun zur näheren Erläuterung des Quelltextes:

In Zeile 1 finden Sie die Anweisung *#include*. Diese deutet dem Compiler an, dass er zusätzliche Informationen zum „Verständnis" des Quelltextes benötigt. Diese Informationen findet er in der Datei *io.h* im Verzeichnis *avr*, das sich wiederum im Standardverzeichnis für Include-Dateien befindet (kann für den Compiler beliebig geändert werden).

In der *io.h* findet der Compiler Informationen über die I/O-Möglichkeiten (Input/Output, also Eingabe/Ausgabe) des verwendeten Mikroprozessors. Die Prozessoren haben unterschiedliche Pin-Belegungen.

Beispiel: An welcher Adresse liegt PORTB bei einem ATmega16 und an welcher Adresse bei einem ATmega128? Genau diese Pin-Belegungen werden in den Include-Dateien definiert.

Sie werden diese Anweisung in den meisten Ihrer Programme benötigen. Sie muss auch immer am Anfang des Programms stehen, zumindest bevor Ihre Funktionen (wie etwa: main) beginnen.

Das Hauptprogramm beginnt in Zeile 2. Egal, an welchem Projekt Sie arbeiten, jedes Ihrer C-Programme wird immer mit demselben „Skelett" beginnen. Dieses Skelett werden Sie nach einigen Startversuchen relativ schnell auswendig können. So sieht das Skelett eines jeden Programms aus:

```
1:   #include <avr/io.h>
2:   int main(void)
3:   {
4:
5:         return 0;
6:   }
```

Bis auf drei Zeilen besteht Ihr erstes Programm (siehe oben) im Wesentlichen nur aus dem Grundskelett. Ich empfehle Ihnen, das „Warum" und „Wie" hinter diesem Skelett vorerst nicht zu hinterfragen. Das zeigt sich nach einer Weile ganz von selbst.

Es gibt in Ihrem Programm also nur drei wirkliche Anweisungen. Diese finden Sie in den Zeilen 4, 5 und 6.

C arbeitet auch nicht „parallel". Jeder Befehl wird nur sequenziell, also einer nach dem anderen, ausgeführt. Das bedeutet in unserem Fall, dass das Datenrichtungsregister (DDRB) für den PORTB erst auf *AUSGABE (0xFF)* geschaltet (Zeile 4) und anschließend der PORTB mit Werten beschrieben wird (Zeile 5 und 6). Dieser wird hier bewusst mit zwei verschiedenen Werten (0x00 und 0xFF) versorgt, um das Ergebnis im Simulator deutlich darzustellen.

Dieses Testprogramm „am Mikrocontroller" wird Ihnen zunächst noch wenig Freude schenken. Die Zustandsänderung passiert einmalig und dies in nur wenigen Taktzyklen, ist also kaum messbar.

Sie haben hiermit also bereits Ihr erstes C-Programm geschrieben. Um die Einführung in diese Sprache möglichst kurz zu halten, folgt eine Zusammenfassung der wichtigsten Punkte, die Sie dabei beachten sollen:

- Befehle werden in C immer mit einem Semikolon abgeschlossen. Beispiel: DDRB = 0xFF; PORTB = 0x00;
- Um eine hexadezimale Zahl zu verwenden, setzt man in C immer die Notierung *0x* vor die Zahl. Dezimale Zahlen werden ganz normal, also ohne diesem 0x-Präfix angegeben. Beispiele: Hexadezimal: DDRB = 0x5A oder Dezimal: DDRB = 90;

- Es ist gleichgültig, ob man *DDRB=0x00;* oder *DDRB = 0x00;* schreibt. Diese Leerzeichen werden ignoriert. Aber Vorsicht: Die einzelnen Argumente bzw. Werte dürfen Sie nicht auftrennen. *DD RB = 0x 00;* wäre nicht erlaubt.
- Jedes Ihrer Programme beginnt mit folgendem Skelett-Code:

```
1:   #include <avr/io.h>
2:   int main(void)
3:   {
4:
5:         return 0;
6:   }
```

4.3 Variablen

Die Sprache C besitzt einen wesentlichen Vorteil gegenüber Assembler: Anstatt Registern, stehen Ihnen (wie in den meisten „Hochsprachen“) beliebig definierbare Variablen zur Verfügung. Beispiel:

```
1:   #include <avr/io.h>
2:   int main(void)
3:   {
4:         char led_status;
5:         char rotationsgeschwindigkeit;
6:         int geheimcode;
7:         long schluessel;
8:
9:         led_status = 0x00;
10:        rotationsgeschwindigkeit = 92;
11:        geheimcode = 3456;
12:        schluessel = 1234564345;
13:
14:        return 0;
15:  }
```

Sie erkennen nun das bereits geläufige Standardskelett. Das Hauptprogramm ist nun schon etwas größer ausgefallen als vorhin.

In den Zeilen 4 bis 7 finden Sie sogenannte *Variablen-Deklarationen*. In den Zeilen 9 bis 12 die entsprechende „Zuweisung von Werten zu den Variablen“.

Einfacher ausgedrückt: Man erklärt dem Compiler, welche Art von Variablen man in seinem Programm verwenden möchte und welchen Namen diese tragen sollen. Die „Zuweisung der Werte“ füllt diese Variablen mit einem Inhalt.

Die korrekte Syntax zur Deklarierung einer Variable sieht folgendermaßen aus:

DATENTYP VARIABLENNAME;

Wir haben im angeführten Quelltext-Beispiel also drei verschiedene Datentypen verwendet, nämlich *char, int* und *long*.

Besonders in der Programmierung von 8-Bit-Mikrocontrollern steht Ihnen meist nur ein begrenzter Speicher für Ihr Programm zu Verfügung. Bei der Verwendung von Variablen muss der Compiler diesen Speicher vorab reservieren. Je mehr Variablen Sie verwenden, umso weniger Speicherplatz bleibt für Ihr eigentliches Programm. Es gibt zum einen Variablen, die mehr Speicher belegen (und größere Inhalte speichern können) und solche, die nur ein einziges Byte speichern können (also eine einzige Zahl zwischen 0 und 255).

Ihre Aufgabe während der Entwicklung Ihrer Software besteht nun darin, den richtigen Datentyp für eine Variable auszuwählen. Um Ihnen diese Entscheidung zu vereinfachen, gibt es nur eine sehr geringe Auswahl an Datentypen:

Bezeichnung	Schlüsselwort	Bytes	Wertebereich
character	char	1	-128 .. 127
unsigned character	unsigned char	1	0 .. 255
integer	int	2	-32 768 .. 32767
short integer	short	2	-32 768 .. 32767
long integer	long	4	-2147483648 .. 2147483647
unsigned integer	unsigned int	2	0 .. 65535
unsigned short integer	unsigned short	2	0 .. 65535
unsigned long integer	unsigned long	4	0 .. 4294967295
single-precision floating-point (7 Stellen)	float	4	1.17E-38 .. 3.4E38
double-precision floating-point (19 Stellen)	double	8	2.2E-308 .. 1.8E308

Abb. 4.1: Datentypen

So bietet sich für unsere Variable *led_status*, die nur 1 oder 0 enthalten wird, natürlich der Datentyp *char* an, da dieser die kleinstmögliche Einheit darstellt. Größere Wertebereiche würden hier keinen Sinn machen.

Wichtig: Stellen Sie sicher, dass Ihren Variablen vor der Verwendung (also vor dem „Auslesen“) ein Wert zugewiesen wurde. Ist das nicht der Fall, steht meist ein rein zufälliger Wert in der Variable und dies führt nicht selten zu einer langen Fehlersuche. Aus diesem Grund wurde in C auch die direkte Zuweisung von Werten bereits bei der Deklarierung der Variable eingeführt. Dies kann beispielsweise so aussehen:

```
1:   #include <avr/io.h>
2:   int main(void)
3:   {
4:         char led_status = 0x00;
5:         char rotationsgeschwindigkeit = 92;
6:         int geheimcode = 3456;
7:         long schluessel = 1234564345;
8:         return 0;
9:   }
```

Außerdem hat sich durch diese Änderung auch die Länge des Programms wesentlich verkürzt.

Richtig interessant werden Variablen natürlich erst dann, wenn Sie mit den Ausgabeports in Verbindung gebracht werden:

```
1:   #include <avr/io.h>
2:   int main(void)
3:   {
4:         char led_status = 0xFF;
5:         DDRB = 0xFF;
6:         PORTB = led_status;
7:         led_status = 0x00;
8:         PORTB = led_status;
9:         return 0;
10:  }
```

Am besten erklärt sich das Beispiel wieder Zeile für Zeile: In Zeile 4 wird eine Variable mit der Bezeichnung *led_status* vom Typ *char* erstellt und erhält den Wert 0xFF (also hexadezimal FF). Binär würde das der Zahl „11111111“ entsprechen.

Zeile 5 ist bereits aus dem Einführungsbeispiel bekannt. Hier wird lediglich die Datenrichtung von PORTB auf *AUSGANG* geschaltet.

In Zeile 6 wird auf PORTB der Inhalt der Variablen *led_status* geschrieben, in unserem Fall also *0xFF*. Das wiederum bedeutet, dass alle Pins von PORTB auf 1 geschaltet werden. In einer 8-teiligen LED-Reihe würde jetzt kein Licht leuchten (bei Pull-Up-Schaltung, ansonsten natürlich schon).

In Zeile 7 wird der Inhalt der Variable *led_status* auf 0x00 (Binär: 00000000) geändert. An der Schaltung der LEDs passiert hier noch gar nichts. Wir haben lediglich den neuen Inhalt der Variable *led_status* festgelegt. Darum wird in Zeile 8 dieser neue Wert dem PORTB zugewiesen, was bei einer LED-Reihe alle acht Lichter erhellen würde.

Gehen wir einen Schritt weiter:

```
1:   #include <avr/io.h>
2:   int main(void)
3:   {
4:         char led_status = 0x00;
5:         DDRA = 0xFF; // PORT A auf Ausgang schalten
6:         DDRB = 0xFF; /* PORT B auf Ausgang schalten */
7:         PORTA = 0xAA;
8:         PORTB = 0x00;
9:         DDRA = 0x00;
10:        led_status = PINA;
11:        PORTB = led_status;
12:        return 0;
13:  }
```

Sie sehen im angeführten Beispiel zwei verschiedene Möglichkeiten zur Kommentierung von Quelltext. Zum Einen den doppelten Schrägstrich (//) und zum Anderen den Schrägstrich mit einem Start- (/*) und einem Stoppzeichen (*/). Es ist ganz Ihnen überlassen, für welche Art der Kommentierung Sie sich entscheiden.

// bedeutet, dass für den Rest der aktuellen Zeile nur mehr Kommentar folgt. Bei der Verwendung von /* und */ definieren Sie, im Gegensatz zur //-Kommentierung, selbst, wo Ihr Kommentar beginnt und wo er endet. Sie können hiermit auch über mehrere Zeilen hinweg kommentieren. (Weitere Beispiele dazu finden Sie in einigen der noch folgenden Quelltexte.)

In den Zeilen 5 und 6 werden die Ports A und B jeweils auf *Ausgang* geschaltet. In den darauffolgenden zwei Zeilen wird dabei auf Port A der Wert *Hex: AA* und auf Port B der Wert *Hex: 00* ausgegeben. Anschließend wird die Datenrichtung von Port A zurück auf *Eingang* gesetzt. Besonderes interessant wird es in Zeile 10. Sie erkennen bereits die Verwendung der Variable *led_status*. In diesem Fall wird ihr der *Eingangswert* von Port A zugewiesen.

Würden Sie anstatt PINA die Bezeichnung PORTA verwenden, weiß der Mikrocontroller nicht, dass Sie einen Wert einlesen wollen und er liest lediglich den *Ausgabebuffer* von PORTA, ohne dazu auf die Pins des Mikrocontrollers selbst zuzugreifen. Wenn Sie gerade den Simulator auf Ihrem Rechner laufen haben, werfen Sie einmal einen Blick auf das Fenster *I/O View* (Achtung: Nur sichtbar, wenn Sie sich im Debugmodus befinden). Öffnen Sie dort den Eintrag *I/O Port A* und Sie werden erkennen, dass der Simulator tatsächlich zwischen *Data Register* und *Input Pins* unterscheidet. Beachten Sie daher bitte: Wenn Sie den Zustand eines Ports wissen wollen, müssen Sie immer *PINA*, *PINB* usw. verwenden. Wollen Sie den Zustand eines Ports ändern, verwenden Sie *PORTA*, *PORTB*, usw.

Zurück zu Zeile 10. Wir lesen also den Wert von Port A (an den Pins) ein und schreiben diesen Zustand in die Variable *led_status*. Am Ende des Beispiels, in Zeile 11, wird der eben in der Variable *led_status* gespeicherte Wert nun auf Port B geschrieben. Diesen Vorgang können Sie ebenfalls im Simulator verfolgen.

In der folgenden Tabelle können Sie die Änderungen der Zustände genau mitverfolgen:

Zeile	Wert PORTA/(PINA)	Datenrichtung PORTA	Wert PORTB/(PINB)	Datenrichtung PORTB
4	?	?	?	?
5	?	Ausgang	?	?
6	?	Ausgang	?	Ausgang
7	Hex: AA	Ausgang	?	Ausgang
8	Hex: AA	Ausgang	Hex: 00	Ausgang
9	Hex: AA	Eingang	Hex: 00	Ausgang
11	Hex: AA	Eingang	Hex: AA	Ausgang

Achtung: Dieses Beispiel eignet sich hauptsächlich für die Veranschaulichung der Wertezuweisungen bei Variablen und Ports mittels Simulator. In einem hardwarebasierten Test ist nicht sichergestellt, dass der Wert von Port A ab Zeile 7 am Port selbst gespeichert wird. So könnte beim Einlesen des Zustands in Zeile 10 bereits ein gänzlich unterschiedlicher Wert am Port A anliegen. Wundern Sie sich also nicht, wenn daher auf Port B am Ende nicht *Hex AA* anliegt.

Ein weiteres Beispiel zur arithmetischen Verwendung von Variablen:

```
1:   #include <avr/io.h>
2:   int main(void)
3:   {
4:         /* Noch eine kleine Demonstration
5:         von einem Kommentar, dass über einige Zeilen
6:         hinweg verläuft. */
7:
8:         char led_status = 0x00;
9:
10:        DDRA = 0xFF;
11:        PORTA = led_status;
12:
13:        led_status = led_status + 1;
14:        PORTA = led_status;
15:
16:        led_status++;
17:        PORTA = led_status;
18:
```

```
19:         led_status+=2;
20:         PORTA = led_status;
21:
22:         return 0;
23:   }
```

Hier sehen Sie drei verschiedene Möglichkeiten, um den Wert einer Variable zu erhöhen bzw. eine Addition durchzuführen. Zeile 13 erhöht den Wert von *led_status* auf *Hex: 01*. Die Funktionsweise dabei ist relativ einfach zu verstehen: „Neuer Wert = Alter Wert + Zahl." Zeile 16 bietet schon eine wesentlich komfortablere Möglichkeit, den Wert von *led_status* um die Zahl 1 zu erhöhen. Zeile 13 erlaubt das Erhöhen um beliebige Werte. Zeile 16 bedeutet jedoch: „Erhöhe den Wert von *led_status* um genau eine Einheit". Mehrere „+" ändern daran nichts. Da aber sehr häufig die Addition von *1* benötigt wird, wurde diese ++-Anweisung eingeführt. Zeile 19 zeigt schließlich die dritte Möglichkeit zur Addition. Es handelt sich dabei im Wesentlichen nur um eine „kürzere Schreibweise" von Zeile 13. Sie können daher auch hier beliebige Werte addieren. Der Wert *Hex 04* befindet sich nun auf Port A am Ende des Beispiels.

Sie sind bei C natürlich nicht nur auf das Addieren beschränkt, sondern können alle gängigen Grundrechenarten durchführen. Hier einige Beispiele:

```
1:          led_status = 0;
2:          led_status = led_status + 1;
3:          led_status *= 2;
4:          led_status--;
5:          led_status += 10;
6:          led_status++;
7:          led_status /= 2;
8:          led_status = 10 - led_status;
9:          led_status = led_status * 2;
```

Dieser Auszug ist schon etwas anspruchsvoller. Am Ende enthält die Variable *led_status* den Wert *Hex 08*.

4.3.1 Schlüsselwörter und Einschränkungen

Sie können bei der Deklarierung Ihrer Variablen alle möglichen Kombinationen von Wörtern und Zahlen verwenden, es gibt dabei jedoch ein paar Einschränkungen:

- Der Name der Variable darf nicht mit einer Zahl beginnen, jedoch mit einer – oder mehreren Zahlen – enden. (Nicht erlaubt: *int 12variable*; erlaubt: *int variable12*)

- Die Variablen dürfen keine arithmetischen oder C-spezifischen Operatoren enthalten, wie etwa *!*, /, \, +, -, #, usw. Erlaubt sind Zeichen von a-z, A-Z, 0–9 und der Unterstrich (_). Umlaute und Satzzeichen sind nicht erlaubt.
- Die Länge der Bezeichnung Ihrer Variablen sollte nicht aus mehr als 32 Zeichen bestehen, denn C unterscheidet Variablen nur anhand dieser ersten 32 Zeichen.
- **Achtung:** C unterscheidet zwischen Groß- und Kleinschreibung! Die Variablen *int test* und *int Test* sind zwei komplett verschiedene Instanzen.
- Es gibt eine Reihe von Schlüsselwörtern, die für die Programmierung in C reserviert sind und daher nicht als Variablenbezeichnung verwendet werden können:

```
auto      break     case      char      const
continue  default   do        double    else
enum      extern    float     for       goto
if        inline    int       long      register
restrict  return    short     signed    sizeof
static    struct    switch    typedef   union
unsigned  void      volatile  while     _Bool
_Complex  _Imaginary
```

Abb. 4.2: Schlüsselwörter in C

4.4 Anweisungen und Funktionen

Sie haben anhand der bisherigen Beispiele bereits einige Anweisungen kennengelernt. In diesem Abschnitt widmen wir uns den sogenannten *Funktionen*.

Beispiel:

```
1:   #include <avr/io.h>
2:
3:   void ledon()
4:   {
5:         PORTB = 0x00;
6:   }
7:
8:   void ledoff()
9:   {
10:        PORTB = 0xFF;
11:  }
12:
13:  int main(void)
14:  {
15:        DDRB = 0xFF; // PortB auf Ausgang
16:
17:        ledon();
```

```
18:         ledoff();
19:
20:         return 0;
21:   }
```

Eine *Funktion* ist eine „Programmeinheit", die erst dann ausgeführt wird, wenn Sie dazu (im Quellcode) den Befehl geben. So könnten Sie z. B. eine Reihe von Funktionen für die Kommunikation mit der seriellen Schnittstelle schreiben. Diese Funktionen können Sie dann in mehreren Projekten wiederverwenden.

Der Aufbau einer Funktion sieht immer gleich aus:

```
DATENTYP DES RÜCKGABEWERTS FUNKTIONSNAME ( OPTIONAL: PARAMETER )
{
  INHALT DER FUNKTION
}
```

In Zeile 3 beginnt bereits die erste Funktion mit dem Namen *ledon*. Außerdem kennen Sie den Rückgabewert dieser Funktion, *void*. Da sich zwischen den beiden Klammern in Zeile 3 kein Inhalt befindet, erwartet diese Funktion keine Übergabeparameter (mehr dazu etwas später). Der Inhalt der Funktion, Port B auf 0x00 zu schalten, ist denkbar einfach.

Eine Funktion könnte z. B. eine komplexe Berechnung enthalten. Eine Berechnung liefert bekanntlich auch ein „Ergebnis". Dieses ist der sogenannte *Rückgabewert* einer Funktion. Wie überall in C müssen Sie sich auch hier überlegen, welcher Datentyp sich für diesen Rückgabewert am besten eignet.

Angenommen, Ihre Funktion liefert nur *0* oder *1* zurück. Welcher Datentyp würde sich dafür anbieten? Natürlich der mit dem kleinsten Wertebereich, also *char*. Der Funktionskopf würde dann etwa so aussehen:

```
1:   char wahrOderFalsch()
2:   {
3:
4:   }
```

Was bedeutet aber nun der Datentyp *void*, wie er sich im Eingangsbeispiel findet? Das Wort *void* drückt so viel aus wie „Leere" oder „nichts". Und genau das wollen wir dort ja auch: keinen Rückgabewert. Unsere Funktion aus dem Beispiel schaltet lediglich den Port B auf einen neuen Zustand. Wir führen keine Berechnungen aus, und wenn wir diese Funktion aufrufen, erwarten wir auch kein Ergebnis (abgesehen von der Zustandsänderung).

Wichtig: Wenn Ihre Funktion also gar keinen Rückgabewert liefert, müssen Sie trotzdem die Bezeichnung *void* angeben.

Zurück zum Beispiel: Dort folgt mit *ledoff()* eine weitere, sehr ähnliche Funktion. Richtig zur Sache geht es dann im Hauptprogramm. Dort finden Sie den eigentlichen „Aufruf" von den beiden zuvor definierten Funktionen. Um eine Funktion zu starten, gehen Sie nach folgendem Schema vor:

FUNKTIONSNAME (OPTIONAL: PARAMETER) ;

Den Aufruf einer Funktion nennt man auch eine *Anweisung*. Sie finden zwei dieser Anweisungen in den Zeilen 17 und 18. Wir übergeben den Funktionen hier keine Parameter und erwarten auch keinen Rückgabewert. Das folgende Beispiel veranschaulicht nun die Verwendung eines Rückgabewertes:

```
1:   #include <avr/io.h>
2:
3:   char getPortBValue()
4:   {
5:         DDRB = 0x00; // Port B als Eingang schalten
6:         return PINB;
7:   }
8:
9:   void setPortBValue(char newvalue)
10:  {
11:        DDRB = 0xFF; // Port B als Ausgang schalten
12:        PORTB = newvalue;
13:  }
14:
15:  int main(void)
16:  {
17:        char led_status=0;
18:        DDRB = 0xFF; // PortB auf Ausgang
19:
20:        led_status = getPortBValue();
21:        led_status++;
22:        setPortBValue(led_status);
23:
24:        return 0;
25:  }
```

Dieses Beispiel ist schon etwas anspruchsvoller. Zerlegen wir es in seine Bestandteile:

Es gibt zwei Funktionen: *getPortBValue()* liefert Ihnen den aktuellen Zustand an Port B als Rückgabewert zurück. Die Funktion *setPortBValue()* erlaubt es Ihnen, Port B einen neuen Wert zuzuweisen.

Drei wichtige Grundlagen sind:

1. Die Definition von Funktionen, die Parameter erwarten und einen Rückgabewert liefern.

2. Das Zwischenspeichern des Rückgabewertes in eine Hilfsvariable.
3. Der Aufruf von Funktionen (mit Parameterübergabe und Rückgabewert).

In Zeile 3 beginnt die Funktion *getPortBValue()* und Sie erkennen sofort: Es wird nicht *void* als Rückgabewert verwendet, sondern *char*. Das bedeutet, die Funktion muss tatsächlich einen Wert zurückgeben. Dies passiert direkt in Zeile 6. Dazu verwenden wir die Anweisung *return*. Nach dem Schlüsselwort *return* können Sie auf verschiedene Weise einen Wert zurückgeben. Das könnte z. B. so aussehen:

```
1:   return 0;
2:   return -1;
3:   return (1+1+2+3);
4:   return 128;
5:   return led_status; // wenn led_status eine char Variable ist
6:   return;
```

Es gibt also einige Möglichkeiten, um aus einer Funktion einen Wert zurückzuliefern. Hierbei ist es jedoch wichtig, einige wesentliche Punkte zu beachten:

1. Nachdem Sie die Anweisung *return* verwenden, wird kein weiterer Code in der Funktion mehr ausgeführt und die Funktion sofort beendet. Sie können somit eine Funktion auch „vorzeitig" abbrechen.
2. Der in Zeile 6 (in den verschiedenen *return*-Beispielen oberhalb) angeführte parameterlose Aufruf wird nur in Funktionen akzeptiert, die keinen Rückgabewert abliefern müssen (also *void*-Funktionen sind). Dies wird in solchen Funktionen ausschließlich als *vorzeitiger Abbruch* verwendet und macht nur Sinn in Verbindung mit Kontrollabfragen wie z. B. der *IF/ELSE*-Abfrage. Mehr dazu etwas später.
3. Der Datentyp der Daten, die Sie als Parameter für die Anweisung *return* verwenden, muss dem Rückgabewert-Datentyp entsprechen. Sie können also z. B. nicht eine Zahl wie „123456532" bei einem Rückgabe-Datentyp *char* übergeben.

Zurück zum Beispiel. In Zeile 20 finden Sie den Aufruf der Funktion *getPortBValue()*. Diesmal findet sich aber die Variable *led_status* noch vor dem Funktionsnamen. Um in einer Variablen einen neuen Wert zu speichern, müssen Sie diesen Wert „zuweisen". Das funktioniert folgendermaßen:

VARIABLENNAME = NEUER WERT ;

Wobei der „neue Wert" natürlich selbst eine Variable oder gar eine komplexe Berechnung sein kann. Der Compiler setzt dann automatisch den berechneten Wert oder den Inhalt der angegebenen Variable als „neuen Wert" ein. Außerdem kann der „neue Wert" auch das Resultat einer Funktion sein. Und genau diese Art der Zuweisung verwenden wir beim Aufruf in Zeile 20. Der Compiler führt zuerst die Funk-

tion *getPortBValue()* aus und setzt anschließend das Ergebnis dieser Funktion (Rückgabewert) als neuen Wert ein. Dieser neue Wert wird in unserem Beispiel dann direkt in der Variable *led_status* gespeichert. Anschließend wird in Zeile 21 der Wert von *led_status* um eine Einheit erhöht.

Wieder interessant wird es in Zeile 22. Hier sehen Sie den Aufruf der Funktion *setPortBValue()*. In diesem Fall erwarten wir keinen Rückgabewert, jedoch erwartet die Funktion einen Übergabeparameter. Dieser Parameter muss vom Datentyp *char* sein (siehe Zeile 9). In der Definition der Funktion (Zeile 9) wurde im Klammerausdruck die Variable *char newvalue* definiert. Diese Variable steht der Funktion zur Verfügung und enthält den Wert, der beim Aufruf übergeben wurde. Hier einige Beispiele:

```
1:   setPortBValue(10);
2:   setPortBValue(0xFF);
3:   setPortBValue(19+29);
4:   setPortBValue(led_status); // led_status ist eine Variable
```

Auch wenn die aufgerufene Funktion keine Parameter erwartet, müssen Sie auf jeden Fall immer die beiden Klammern „(“ und „)“ angeben:

```
1:   irgendeineFunktion();
```

Die Variable *led_status* im Ausgangsbeispiel enthält einen bestimmten Wert (Eingangswert von Port B um eine Einheit erhöht). Dieser wird der Funktion übergeben. Der Compiler springt dann auf Zeile 9 und setzt diesen Übergabewert in die lokale Variable *newvalue* ein.

Im weiteren Verlauf dieser Funktion (Zeile 12) kann diese lokale Variable (nur „sichtbar“ innerhalb der Funktion) verwendet werden. In unserem Beispiel wird der Wert der Variable auf den Port B geschrieben.

Wichtig: Sie benötigen in dieser Funktion kein *return*, da die Funktion keinen Rückgabeparameter erwartet. Die Funktion „läuft einfach aus“ und springt dann automatisch an den Ursprung (Zeile 22) zurück.

Wie würde das Beispielszenario bei einer Übergabe von mehreren Parametern aussehen?

```
1:   #include <avr/io.h>
2:
3:   int addiere(int a, int b)
4:   {
5:         return a+b;
6:   }
7:
```

```
8:   int main(void)
9:   {
10:        DDRB = 0xFF;
11:        PORTB = addiere(5,15);
12:        return 0;
13:  }
```

In Zeile 3 finden Sie den entscheidenden Schritt dazu. Im Funktionskopf werden mehrere Variablen einfach mit einem Beistrich voneinander getrennt. Sie können hier eine ganze Reihe von Variablen übergeben. Achten Sie jedoch darauf: Für jede Variable, die von der Funktion erwartet wird, müssen Sie beim Aufruf auch einen Wert übergeben (siehe Zeile 11). Im oben angeführten Beispiel wird schlussendlich der Wert *Dez: 20* auf Port B ausgegeben.

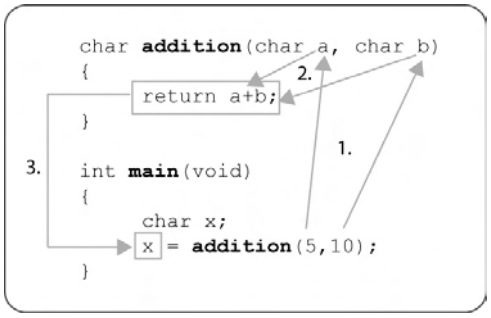


Abb. 4.3: Übergabe von Funktionsparametern

4.5 Abfragen und Schleifen

In diesem Kapitel lernen Sie zwei wichtige Techniken der Programmierung kennen: Entscheidungen und Schleifen. Diese werden Ihnen eine Fülle von Möglichkeiten eröffnen und einen weiteren großen Vorteil von C gegenüber Assembler deutlich machen.

4.5.1 If-Abfrage

Um einen Ausdruck auszuwerten und je nach Ergebnis weitere Schritte zu setzen, verwendet man in C die Anweisung *If*. Grundsätzlich ist diese Anweisung relativ beschränkt, denn Sie erlaubt nur die Unterscheidung, ob ein Ausdruck *wahr* oder *falsch* ist.

Erstes Beispiel:

```
1:   #include <avr/io.h>
2:
3:   int main(void)
4:   {
5:         DDRB = 0x00; // Port B auf Eingang schalten
6:         DDRA = 0xFF; // Port A auf Ausgang schalten
7:
8:         if (PINB == 0xFF)
9:         {
10:              PORTA = 0xAA;
11:        }
12:
13:        return 0;
14:  }
```

Port B wird auf Eingang geschaltet, Port A auf Ausgang. Anschließend wird überprüft, ob der aktuelle Wert an Port B der Zahl *Hex: FF* entspricht. Trifft dies zu, wird auf Port A der Wert *Hex: AA* ausgegeben, ansonsten nicht. In Zeile 8 finden Sie unsere Erweiterung. Es handelt sich um eine sogenannte *If*-Abfrage. In der darauf folgenden Klammer (in Zeile 8) sehen Sie den zu überprüfenden Ausdruck.

Die *If-Abfrage* kann nur überprüfen, ob ein Ausdruck *wahr* oder *falsch* ist. Genau das passiert auch hier. Die Abfrage überprüft, ob an *PINB* gerade der Wert *0xFF* anliegt. Entscheidend hier ist das doppelte Gleichheitszeichen (==). Es signalisiert keine Zuweisung (wie das normale Gleichheitszeichen in C, =), sondern eine Überprüfung: „Ist der erste Ausdruck gleich dem Zweiten?“ Sollte dieser Ausdruck wahr sein, wird der Programmblock darunter ausgeführt. Dieser Block reicht von Zeile 9 bis Zeile 11 und ist durch die geschwungenen Klammern „{“ und „}“ gekennzeichnet. Ist der Ausdruck nicht wahr, wird dieser Block übersprungen. Somit haben Sie einen bestimmten Programmteil, der „auf Bedingungen“ reagiert. Würde an Port B eine Taste oder ein Knopf geschaltet sein, könnte man damit nun auf eine Zustandsänderung reagieren. Ist die Taste gedrückt, wird ein spezieller Programmblock ausgeführt, ist die Taste nicht gedrückt, wird dieser Abschnitt übersprungen.

Auch das Gegenteil der *Gleichheitsprüfung* ist möglich, die Prüfung auf *Ungleichheit*:

```
1:   #include <avr/io.h>
2:
3:   int main(void)
4:   {
5:         DDRB = 0x00; // Port B auf Eingang schalten
6:         DDRA = 0xFF; // Port A auf Ausgang schalten
7:
8:         if (PINB != 0xFF)
9:         {
```

```
10:               PORTA = 0xAA;
11:         }
12:         else
13:         {
14:               PORTA = 0x00;
15:         }
16:
17:         return 0;
18:   }
```

In diesem Programm sehen Sie zum einen die angekündigte *Ungleichheitsprüfung* (Zeile 8) und zum anderen eine neue Anweisung, nämlich die *else*-Anweisung (Zeile 12). In Zeile 8 wird nun also überprüft, ob der aktuelle Wert an Port B nicht *Hex: FF* entspricht. Das kann bei allen Werten von *0x00* bis *0xFE* der Fall sein. Die *else*-Anweisung erlaubt es Ihnen, einen speziellen Programmblock zu definieren, der nur dann ausgeführt wird, wenn die *If*-Abfrage nicht „wahr" oder „nicht erfüllt" ist. In unserem Fall würde das bedeuten: Ist der Wert an Port B nicht *0xFF*, wird der Programmblock von Zeile 13 bis Zeile 15 ausgeführt, nicht jedoch der Block von Zeile 9 bis Zeile 11. Würde, im umgekehrten Fall, an Port B der Wert *0xFF* anliegen, würde der erste Programmblock ausgeführt (weil der Ausdruck *wahr* ist), der zweite jedoch übersprungen.

Wir können diese Technik noch um eine letzte Möglichkeit erweitern:

```
1:   #include <avr/io.h>
2:
3:   int main(void)
4:   {
5:         DDRB = 0x00; // Port B auf Eingang schalten
6:         DDRA = 0xFF; // Port A auf Ausgang schalten
7:
8:         if (PINB == 0xFF)
9:         {
10:              PORTA = 0xFF;
11:        }
12:        else if (PINB == 0x00)
13:        {
14:              PORTA = 0x00;
15:        }
16:        else if (PINB == 0x05)
17:        {
18:              PORTA = 0x50;
19:        }
20:        else
21:        {
22:              PORTA = 0x11;
23:        }
24:        return 0;
25:  }
```

In diesem Beispiel befindet sich eine *Else-If*-Anweisung in Zeile 12. Sie können auf diese Weise mehrere Ausdrücke überprüfen, dabei kann jedoch nur ein einziger *wahr* sein. Alternativ dazu könnten Sie natürlich auch einzelne *If*-Abfragen verwenden. Dies würde jedoch unter Umständen mehrere Lösungen ergeben und somit mehrere Programmblöcke ausführen.

Im oben angeführten Beispiel bekommt Port A einen bestimmten Wert zugewiesen, je nachdem welcher Wert an Port B gerade anliegt. Trifft keine der Bedingungen zu, wird der Programmcode in Zeile 22, also dem *else*-Block, ausgeführt.

Natürlich kann in einem *If*-Programmblock auch eine weitere *If*-Abfrage liegen. Das könnte dann so aussehen:

```
1:   if (PINB == 0x50)
2:   {
3:         if (PIND == 0x50)
4:         {
5:               PORTB = 0x00;
6:         }
7:   }
```

In diesem Fall würde Zeile 5 nur dann erreicht werden, wenn an Port B und Port D jeweils den Wert *Hex: 50* anliegt.

4.5.2 switch-Abfrage

Die *If*-Abfrage ist ein brauchbares Hilfsmittel bei der Entwicklung interaktiver Applikationen. Ein kleiner Nachteil dieser Abfrage macht sich dann bemerkbar, wenn eine einzige Variable auf verschiedene Werte geprüft werden soll.

Nehmen wir an, Sie verwenden in Ihrer Anwendung einen Drehschalter, der 20 verschiedene Zustände erlaubt. Je nach Zustand liegt ein entsprechender Wert an Port B an. Die Abfrage dieser Zustände würde ein unübersichtliches *If-Else*-Konstrukt benötigen.

Dies lässt sich aber mit einer ähnlichen Art von *Abfrage*, nämlich der *switch*-Abfrage einfacher lösen:

```
1:   #include <avr/io.h>
2:
3:   int main(void)
4:   {
5:         DDRB = 0x00; // Port B auf Eingang schalten
6:         DDRA = 0xFF; // Port A auf Ausgang schalten
7:
8:         switch (PINB)
```

```
9:          {
10:               case 0x00:
11:                     PORTA = 0xFF;
12:                     break;
13:
14:               case 0x01:
15:                     PORTA = 0xAA;
16:                     break;
17:
18:               case 0x02:
19:                     PORTA = 0x55;
20:                     break;
21:
22:               case 0x03:
23:                     PORTA = 0x88;
24:                     break;
25:
26:               default:
27:                     PORTA = 0x00;
28:                     break;
29:         }
30:         return 0;
31:   }
```

Vor allem für State-Machines ist diese *switch*-Abfrage sehr hilfreich. State-Machines beschreiben eine besondere Art der Programmierung, wobei die Software von einem Zustand in einen anderen Zustand wechselt und jeder Zustand eine eigene Aufgabe ausführt.

In Zeile 8 befindet sich der Start der Anweisung. In den runden Klammern, die nach dem Schlüsselwort *switch* folgen, findet sich immer die zu überprüfende Variable (bzw. ein Port). Wir wollen den aktuellen Zustand von Port B näher untersuchen. Ferner gibt es dann verschiedene Möglichkeiten (eng.: cases), die auf den Zustand von Port B zutreffen.

In unseren Fall prüfen wir Port B auf die Werte: 0, 1, 2 und 3. Sollte an Port B der Wert 0x02 anliegen, springt die *switch*-Abfrage direkt zu Zeile 18. Dort wird anschließend der Wert 0x55 auf Port A geschrieben. Das anschließende *break* ist hier besonders wichtig.

Break bedeutet „Abbruch“, und genau das wollen wir hier auch. Wir haben unseren Wert gefunden und wollen, dass der Controller die Überprüfung abbricht. Würde man das *break* nicht verwenden, würde der ganze Code, der diesem *case* noch folgt, automatisch ausgeführt. Dies ist eine der wenigen, jedoch typischen Eigenheiten der Sprache C.

Daher: Das Schlüsselwort *break* bricht an der aktuellen Stelle die weitere Ausführung der Abfrage ab und kehrt zum Programm zurück. Wird dieses *break* nicht am Ende eines jeden *case* verwendet, werden bei einer Übereinstimmung mit einem Abfragewert auch alle anderen Abfragen als „richtig" erkannt und der komplette, noch folgende Code ausgeführt.

Würde in unserem Beispiel oben kein einziges *break* vorhanden sein und an Port B der Wert 0x00 anliegen, würden nacheinander die Zeilen 11, 15, 19, 23 und 27 ausgeführt werden. Das ist sicherlich nicht erwünscht, daher sollte man niemals das *break* vergessen.

In Zeile 26 findet sich mit *default* aber noch ein weiteres Schlüsselwort. Nehmen wir an, Port B trüge den Wert 0x20. Dann würde keine der oben angeführten *case*-Möglichkeiten zutreffen. Genau darum kümmert sich das Schlüsselwort *default*. Es bedeutet in erster Linie: „Nur wenn wirklich keine der Möglichkeiten zutrifft, mach Folgendes ..." Sobald eine Möglichkeit „wahr" ist, wird das *default* auch schon verworfen. Wäre der Wert im Beispiel an Port B also 0x20, würden wir an Port A den Wert 0x00 ausgeben.

Diese Art der Abfrage sieht bei einer Reihe von Anwendungen doch etwas übersichtlicher aus als eine reine *If-Abfrage*. Es sei jedoch angemerkt, dass man eigentlich immer auch eine normale *If/Else*-Abfrage statt einer *switch*-Abfrage verwenden kann.

Sie haben zu diesem Zeitpunkt bereits die wichtigsten Methoden der Softwareentwicklung in C kennengelernt. Das Einzige, was Ihnen zur erfolgreichen Entwicklung noch fehlt, sind Schleifen.

4.5.3 while-Schleife

Eine Schleife ist, wie der Name schon erahnen lässt, etwas, was sich „im Kreis" bewegt. Genau diese Eigenschaft wird Ihnen bei Ihren Programmen eine große Hilfe sein. Was, denken Sie, passiert, nachdem Ihr Programm zu Ende ist, also die letzte Zeile von *main()* erreicht wurde? Darauf gibt es wahrscheinlich keine hundertprozentig richtige Antwort. Denn je nach Compiler und Einstellungen wird Ihr Programm sich anschließend entweder aufhängen, neu starten oder verrückte, unbestimmte Dinge tun.

Der Programmcode wird im Flash des Mikrocontrollers gespeichert. Wenn nach Ihrem Programmcode noch Zeichen im Speicher liegen (alles außer 0x00), wird der Controller dies als Maschinencode zu interpretieren versuchen. Das kann zu unerwünschten Resultaten führen. Normalerweise hängt sich der Controller aber auf und benötigt anschließend einen Reset.

Die gängigste Anwendung einer *Schleife* beginnt bereits hier. Das Programm könnte sich z. B. immer wiederholen. Es gibt hierbei zwei Möglichkeiten, wie Sie Ihre Software entwickeln können:

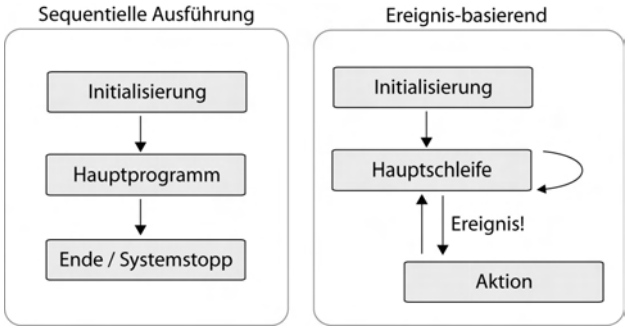


Abb. 4.4: Programmablauf

Ein Auszug, der „ewig" den Port B mit wechselnden Werten beschreibt:

```
1:   #include <avr/io.h>
2:
3:   int main(void)
4:   {
5:         DDRB = 0xFF; // Port B auf Ausgang schalten
6:         while (1)
7:         {
8:               PORTB = 0x00;
9:               PORTB = 0xFF;
10:        }
11:        return 0;
12:  }
```

Die neue Funktion trägt den Namen *while*, was so viel bedeutet wie „während". Um genauer zu sein: „Während der Ausdruck in der Klammer wahr ist, führe den untergeordneten Programmblock aus".

Warum haben wir dann eine *1* in die Klammer geschrieben? In C (und generell in der Softwareentwicklung) entsprechen 0 und alle negativen Zahlen automatisch dem Zustand „nicht wahr", oder „falsch". Zahlen wie die *1* und darüber entsprechen dann dem Zustand „wahr". In unserem Fall würde das bedeuten, dass die Klammerbedingung immer „wahr" ist (weil sich die *1* dort nicht mehr verändert). Das macht durchaus Sinn, denn das ist genau unser Anliegen: bedingungslose Wiederholung des Programmblocks. Man nennt dies auch eine *Endlosschleife*. Der Controller wird in diesem Beispiel aus der Schleife nicht mehr herauskommen – zumindest solange nicht, bis Sie ihm einen Reset genehmigen.

Um sicherzugehen, dass sich Ihr Programm nach der Ausführung nicht in einen unbestimmten Zustand begibt, empfiehlt es sich auch, diese *Endlosschleife* am Ende jedes Programms zu verwenden. Vor allem natürlich bei solchen, die nur einen Durchlauf pro Reset benötigen (Beispiel: Einschaltmechanismen, bei denen der Mikrocontroller mit einer Spannungsversorgung eingeschaltet wird und dann eine kurze Routine ausführt, um anschließend „nichts mehr" zu tun).

Auch beim Testen Ihrer Software wird sich diese Schleifenmethodik als hilfreich erweisen. Sie können das Ganze auch in einer *Hang-up*-Anweisung (in dieser Zeile bleibt der Controller sicher bis zum Reset hängen) zusammenfassen:

```
1:   #include <avr/io.h>
2:
3:   int main(void)
4:   {
5:         DDRA = 0xFF; // Port A auf Ausgang schalten
6:         DDRB = 0xFF; // Port B auf Ausgang schalten
7:
8:         PORTB = 0x00;
9:         PORTA = 0xA3; // Beispiel: Einschaltcode schicken
10:        PORTB = 0xFF;
11:
12:        while(1); // Endlosschleife: Hang-up
13:
14:        return 0;
15:  }
```

Erreicht der Mikrocontroller Zeile 12, wird er sich dort bis zum nächsten Reset festhängen. Alles darunter (in unserem Fall nur *return 0*) wird niemals ausgeführt.

Die *while*-Schleife ist aber eigentlich für höhere Aufgaben bestimmt. Das könnte z. B. so aussehen:

```
1:   #include <avr/io.h>
2:
3:   int main(void)
4:   {
5:         DDRA = 0x00; // Port A (=Taste) auf Eingang schalten
6:         DDRB = 0xFF; // Port B auf Ausgang schalten
7:
8:         while (PINA != 0x00) ;
9:
10:        PORTB = 0xAA;
11:
12:        while(1); // Endlosschleife: Hang-up
13:
14:        return 0;
15:  }
```

Die Magie passiert hier in Zeile 8. Im Wesentlichen bedeutet diese Anweisung „Solange an Port A nicht der Wert Hex: 00 anliegt, warte in einer Schleife und mache nichts“. Nehmen wir an, es liegt plötzlich der Wert *Hex: 00* an Port A an (Beispiel: eine Taste, die alle 8 Bit auf *0* schaltet). Die Schleifenbedingung ist somit nicht mehr erfüllt und der Controller springt weiter zur nächsten Anweisung in Zeile 10. Das Programm läuft schlussendlich in die Endlosschleife ab Zeile 12.

Das wäre bereits ein gutes Beispiel für *Interaktivität*, basierend auf externen Zustandsänderungen. Gehen wir einen Schritt weiter:

```
1:   #include <avr/io.h>
2:
3:   int main(void)
4:   {
5:         char a=0; // Zählvariable a mit 0 initialisieren
6:         DDRA = 0x00; // Port A (=Taste) auf Eingang schalten
7:         DDRB = 0xFF; // Port B auf Ausgang schalten
8:
9:         while (PINA != 0x00)
10:        {
11:              a++;
12:              PORTB = a;
13:        }
14:
15:        while(1); // Endlosschleife: Hang-up
16:
17:        return 0;
18:  }
```

Der Kern dieses Beispiels findet sich in den Zeilen 9 bis 13. Wir verwenden eine Variable *a* vom Typ *char* (Wertebereich: 0 bis 255, 8 Bit). Die *while*-Schleife hat als Bedingung, dass der Zustand an Port A nicht *Hex: 00* annimmt (wir gehen also davon aus, dass der Zustand zu Beginn *Hex: FF* ist). In der Schleife selbst wird der Wert von *a* immer um eine Einheit erhöht und anschließend der neue Wert auf Port B geschrieben.

Wenn wir davon ausgehen, dass an Port A eine Taste oder ein Schalter liegt, der zu Beginn den Zustand *Hex: FF* besitzt, würde bis zu einem Tastendruck (also einer Änderung von Port A) ständig eine Zahlenreihe an Port B ausgegeben werden.

Sie können dieses Beispiel anschaulich im Simulator verfolgen. Was wird passieren, wenn die Variable *a* den Wert *255* erreicht? Sie wird bei der nächsten Erhöhung wieder bei *0* beginnen. Man nennt dies einen „Überlauf“. Somit würde am Port eine Zahlenreihe, ähnlich dieser ausgegeben werden: „1, 2, 3, 4, 5 ... 253, 254, 255, 0, 1, 2, 3, 4 .....“

Sachverzeichnis

FRANZIS
PC & ELEKTRONIK

Roman Mittermayr

AVR-RISC

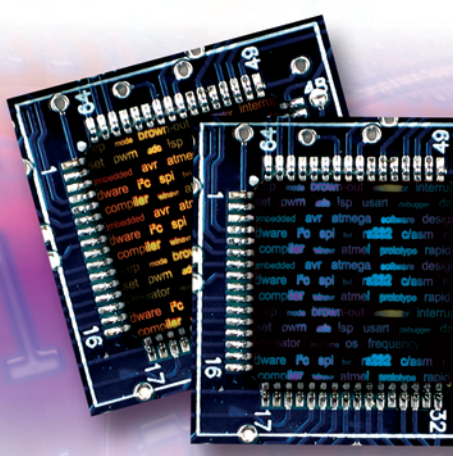
Embedded Software selbst entwickeln

Ganz gleich, ob Sie das Wort Mikrocontroller zum ersten Mal hören, seit einigen Jahren „Embedded Software“ schreiben oder Informatik unterrichten: Dieses Buch hilft Ihnen beim Erlernen, Umsteigen und Vertiefen Ihres Wissens. Sie bekommen eine kurze Einführung in die Sprache Assembler und lernen außerdem die wichtigsten Grundlagen der C-Programmierung kennen. Für den Praktiker wird prägnant erklärt, worauf es bei der Software-entwicklung auf AVR-Prozessoren wirklich ankommt.

Die 8-Bit-Mikrocontroller-Serie von ATMEL erlaubt Ihnen eine unglaublich rasche Verwirklichung verschiedenster Ideen und Projekte. Die schnelle RISC-Architektur, der einheitliche Aufbau der kompletten Prozessorfamilie und nicht zuletzt die attraktiven Preisangebote für Hobbyentwickler machen diesen Controller zur richtigen Wahl.
Der Autor hat sich darauf konzentriert, das Buch verständlich und praktisch zu gestalten. Dieses Werk wird den Arbeitstisch kaum verlassen. Ein Nachschlagewerk für Ihre Projekte: Häufig benötigte Tabellen, Routinen und bekannte „Fallen“ werden klar und einfach erklärt.
Eine umfangreiche Onlineplattform unterstützt den Leser und beantwortet Fragen, die über den Umfang des Buchs hinausgehen. Zahlreiche Praxisbeispiele und Schaltungen stehen darüber hinaus frei zum Download zur Verfügung.

Aus dem Inhalt:

- Serielle Kommunikation
- LCD-Steuerung
- EEPROM
- A/D-Wandler
- GPS Way-Tracker
- Klingelton
- LED-Matrix-Segment
- Frequenzzähler



ISBN 978-3-7723-4107-6

Euro 39,95 [D]