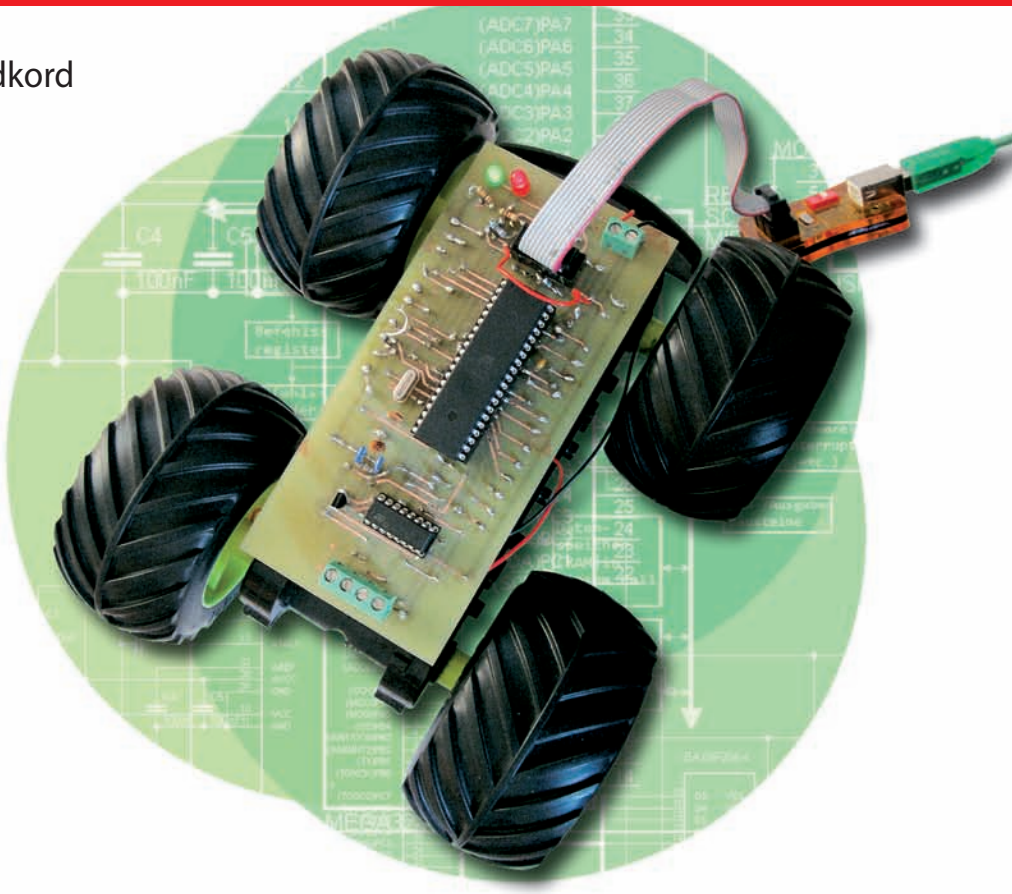
FRANZIS
PC & ELEKTRONIK

Sven Feldkord



Einführung in die C-Programmierung mit dem ATmega32

Aufbau und Programmierung

Vorwort

Immer häufiger begegnen wir im Alltag eingebetteten Systemen: z. B. beim Auto, das elektronische Unterstützung durch einen Bordcomputer erhält, bei einer einfachen digitalen Zeitanzeige oder einer Stoppuhr an einem Backofen.

Sie alle sind elektronisch gesteuert und eine Art kleiner Computer. Wie ein solcher bestehen sie aus einer Hardware und einer Software, die von der Hardware ausgeführt wird. Dieses Buch ermöglicht Ihnen den Einstieg in die Welt der eingebetteten Systeme anhand eines Mikrocontrollers der Atmel-AVR-Reihe. Am Beispiel des Mikrocontrollers ATMega32 werden zuerst die Grundlagen der Funktionsweise von Mikrocontrollern und anschließend Grundkenntnisse der Programmiersprache C vermittelt.

Nach Abschluss der Grundlagen werden gleichzeitig weiterführende Funktionen des Mikrocontrollers und deren Ansteuerung in einem Programm erläutert.

Nach Ausführungen zu Funktionalitäten und Programmierung wird das erworbene Wissen durch Beispielschaltungen und -programme vertieft. Das beginnt mit einer einfachen Blinkschaltung und führt bis zum Speichern einer Textdatei auf einen USB-Stick.

Voraussetzung zum Verständnis des Buchs sind grundlegende Kenntnisse von elektronischen Schaltungen im Allgemeinen (ohne Mikrocontroller). Kenntnisse über Aufbau und Funktionsweise eines normalen Rechners sind teilweise hilfreich (z. B. um den Aufbau des CPU-Kerns des Mikrocontrollers nachvollziehen zu können). Sie sind aber nicht Voraussetzung zum Verständnis des gesamten Werks.

Weiterhin sind viele der Code-Beispiele auch in Assembler vorhanden. Ein Verständnis der Assembler-Sprache ist jedoch nicht erforderlich, um die Beispiele nachvollziehen zu können. Es ist lediglich eine Hilfe für jene, die bereits in Assembler programmiert haben. Das ermöglicht zudem einen tieferen Einblick in die Funktions- und Arbeitsweise des Mikrocontrollers, weil in C-Programmen ein Befehl meist den Inhalt sehr vieler Befehle in Assembler umfasst. Da es jedoch hauptsächlich um die Programmierung in C und die technischen Grundlagen geht, wird es keine Einführung in die Programmierung in Assembler geben. Lediglich ausgewählte Instruktionen und ihre Auswirkungen werden erläutert.

Das erleichtert all jenen den Umstieg, die bereits auf anderen Systemen in Assembler programmiert haben.

Inhaltsverzeichnis

4 RAM-Nutzung

Das RAM wird lediglich beim Laden oder Speichern eines Bytes angesprochen. Zur Ausführung der entsprechenden Befehle werden drei Register benötigt: ein Datenregister, aus dem das gelesene Byte ausgelesen oder das zu schreibende Byte geschrieben werden soll, und ein Adressregister, in das die Adresse der Speicherzelle geschrieben wird, aus der gelesen oder in die geschrieben werden soll.

Das Adressregister besitzt eine Breite von 16 Bit. Es ist also kein einzelnes Register, sondern wird aus zwei Registern zusammengesetzt, von denen ein Register das niederwertige und das andere das höherwertige Byte der Adresse aufnimmt. Die Adressen der beiden Register liegen dabei hintereinander, sodass eine direkte Übernahme der 16 Bit in den Adressbus ohne Sprung möglich ist.

Von diesen Registerpaaren werden drei bereitgestellt und sind als X-, Y- und Z-Register benannt. Dabei bilden die Register *r26* und *r27* das X-Register, *r28* und *r29* das Y-Register und *r30* und *r31* das Z-Register. Das Register auf der niederen Adresse/der kleineren Nummer enthält dabei das niederwertige Byte (das Low-Byte des X-Registers ist somit in r26 enthalten).

Ein weiterer Aspekt des RAM ist, dass sich der Adressbereich innerhalb des gleichen Bereichs, in dem auch die Register und IO-Ports liegen, befindet. Er beginnt an der Adresse 0x60/96 (dezimal). Folglich muss dieser Offset zu logischen Adressen addiert werden, um die physikalischen zu erhalten.

Beispiel: Byte 0 liegt an der logischen Adresse 0, die physikalische entspricht aber 0x60, weil der Adressbereich des RAM erst dort beginnt.

Beim Kompilieren eines C-Programms werden die Code-Sequenzen, die den Inhalt deklarierter Variablen (dazu später mehr) in das RAM auslagern, automatisch erzeugt. Soll die Speicherbelegung manuell gesteuert werden, kann dies nur in Assembler erfolgen.

Das folgende Beispielprogramm wird 8 Bit im RAM nacheinander mit den Werten von 0 bis 255 belegen.

Programmcode (Assembler):

```
;ram_test.asm

   .include "m32def.inc"

   .def          count      =     r16
   .def          tmp        =     r17

   rjmp          _main
_main:
   ldi           count,           0x00
   ldi           tmp,             0x00
   ;Adresse in Adressregister laden
   ldi           XL,              0x60
   ldi           XH,              0x00
   ; Startwert (0) einspeichern
   st            X+,              tmp
_loop:
   inc           count      ;Zähler erhöhen
   ;Wert aus RAM laden
   ld            tmp,             X+
   inc           tmp        ;Wert erhöhen
   ;neuen Wert einspeichern
   st            X+,              tmp
   ;Vergleichen, ob der Grenzwert (255) erreicht
   ;wurde. Wenn ja, Schleife abbrechen
   cpi           count,           0xff
   breq          _endloop
   rjmp          _loop      ; Rücksprung
_endloop:
   rjmp          _endloop
```

Im Folgenden konzentriert sich dieses Buch auf die Programmierung in C. Ergänzend sind aber in fast allen Fällen die Beispiele auch in Assembler beigefügt.

5 C-Programmierung

Um die folgenden Programmbeispiele in C nachvollziehen zu können, wird es an dieser Stelle notwendig, grundlegende Kenntnisse der Programmierung eines Mikrocontrollers in C zu vermitteln.

Falls diese Kenntnisse schon vorhanden sind, können Sie dieses Kapitel überspringen. Hier werden nur grundlegende Mechanismen erläutert. Spezielle Möglichkeiten der Ansteuerung verschiedener Funktionen wie die Verwendung von Interrupts oder Analog-Digital-Wandlern erfolgen in den entsprechenden Kapiteln synchron zur Erläuterung der Funktionen.

5.1 Kompilieren und Linken

Damit ein Programm ausgeführt werden kann, muss es zuerst in Maschinencode übersetzt werden.

Dies geschieht in zwei Schritten. Zuerst wird eine Quelldatei in eine Objektdatei umgewandelt. Diese Objektdatei wird anschließend in ausführbaren Maschinencode umgewandelt.

Die Bedeutung der Objektdatei liegt in der Auflösung symbolischer Adressen. In fast jedem Programm benutzt man Unterprogramme oder Funktionen, häufig auch aus anderen Dateien. Anstelle der Befehle, die beim Aufruf dieser Funktionen ausgeführt werden, wird lediglich ein Funktionsaufruf in den Quellcode geschrieben.

Beispiel: Es gibt eine Funktion *f*, die eine übergebene Zahl um 1 erhöht. In unserem Programm würden wir dann die Zahl *i* nicht direkt um 1 erhöhen, sondern die Funktion *f* mit dem Parameter *i* aufrufen.

Beim sogenannten *Linken* werden die Objektdateien zu einer ausführbaren Datei zusammengefügt und symbolische Adressen aufgelöst oder durch logische ersetzt.

Eine symbolische Adresse wäre der Name *f* unserer Funktion, die wir aufrufen. Im Maschinencode brauchen wir allerdings die Speicheradresse, an der die

Funktion liegt. Somit wird der Name *f* durch die Adresse der Funktion in Relation zum Programmbeginn ersetzt.

Die Adresse der Funktion ist allerdings nicht die Adresse, zu der letztlich während der Ausführung gesprungen wird. Deshalb heißt sie *logische* Adresse, da sie sich auf die Speicheradresse, an der das Programm liegt, bezieht. Folglich muss zum Ansteuern der physikalischen Adresse ein Offset addiert werden.

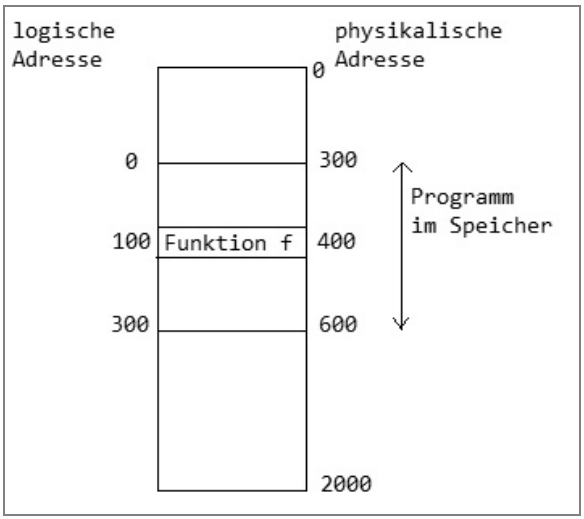


Abb. 5.1: Logische und physikalische Adresse

In diesem Beispiel (Abb.5.1) haben wir einen fiktiven 2.000 Byte großen Speicher. Unser Programm ist 300 Byte groß und beginnt an der Speicherstelle 300 im Speicher. Folglich wird auf jede logische Adresse 300 addiert, um das entsprechende Byte im Speicher zu adressieren.

Die beim Linken ermittelte logische Adresse der Funktion *f* ist 100. Weil das Programm jedoch erst an der Adresse 300 beginnt, ist die physikalische Adresse, zu der beim Ausführen des Programms gesprungen wird, 400.

Im Allgemeinen ist es jedoch nicht der Fall, dass sämtliche Referenzen zu logischen Adressen aufgelöst werden. In C gibt es zusätzlich zu der direkten Einbindung anderer Dateien auch die Möglichkeit, *shared libraries* zu benutzen.

Diese Programmbibliotheken werden nur einmal zur Laufzeit in den Speicher geladen und können von mehreren Programmen gleichzeitig benutzt werden.

Dies erspart viel Speicherplatz, wenn mehrere Programme dieselbe Funktion benutzen. Die Standard-C-Bibliothek ist eine solche shared library.

Die Funktionsweise dieser Einbindung während der Laufzeit ist jedoch zu komplex, um sie hier zu erläutern. Außerdem ist sie für die Mikrocontroller-Programmierung irrelevant. Auf dem Mikrocontroller ist schließlich kein Betriebssystem oder anderes Programm vorhanden. Der einzige Befehlscode, der ausgeführt wird, ist der, den wir schreiben.

5.2 Header- und Source-Dateien

Bei der Programmierung in C werden zwei Typen von Dateien verwendet: die Header- und die Source-Dateien.

Header-Dateien enthalten lediglich Prototypen von Funktionen und Definitionen. Sie werden beim Schreiben von anderen Programmen eingebunden und stellen für den Compiler sicher, dass diese Funktionen mit den angegebenen Parametertypen existieren. Wie diese Funktionen selbst aufgebaut sind und funktionieren, ist in den Header-Dateien allerdings nicht definiert.

Die dazugehörige Source-Datei implementiert die Funktionen, die in der Header-Datei definiert sind. Die Source-Dateien sind, im Gegensatz zu den Header-Dateien, oft für externe nicht frei, sondern lediglich als kompilierte Objektdateien verfügbar. Der fremde Programmierer wird also in seinem Programm die Header-Datei einbinden und beim Linken die Objektdatei mit einbeziehen.

Jede Source-Datei sollte ihre eigene Header-Datei einbinden.

5.3 Der Präprozessor

Ein weiterer Faktor, der das Kompilieren und Linken beeinflusst, sind Präprozessor-Direktiven.

Hier werden wir folgende Anweisungen behandeln:

- include
- Symbole und bedingte Einbindung
- Makrodefinitionen
- die Operatoren # und ##

Im Allgemeinen beginnen Präprozessor-Anweisungen mit einer Raute.

Die *include*-Anweisung wird verwendet, um Header-Dateien in den Programmtext einzubinden.

Meist stehen die *include*-Anweisungen am Beginn einer Source-Datei. Den Namen der einzubindenden Datei kann man dabei auf zwei Arten angeben:

```
#include "mycode.h"
```

oder

```
#include <mycode.h>
```

Diese beiden Versionen unterscheiden sich nur geringfügig – jedoch darin, wo nach der Datei gesucht wird. Normalerweise übergibt man dem Compiler eine Liste von Verzeichnissen, in denen nach den eingebundenen Dateien gesucht wird. Im oberen Fall wird zusätzlich noch im aktuellen Verzeichnis gesucht, also in dem, in dem sich die Datei selbst befindet. Im unteren Fall hingegen wird nur in den übergebenen *include*-Pfaden gesucht.

Beim Kompilieren werden die Header-Dateien in den Programmtext (die Source-Datei) eingefügt und daraus die Objektdatei erzeugt (→ eine Objektdatei entsteht niemals nur aus einer Header-Datei).

Zudem ist es mithilfe des Präprozessors möglich, die Einbindung eines bestimmten Teils mit einer Bedingung zu verknüpfen. Dafür gibt es unter anderem die Anweisungen *ifdef*, *ifndef* und *else* in Verbindung mit einer der beiden vorhergegangenen.

Hierbei wird überprüft, ob ein Symbol definiert wurde (*ifdef*) oder ob es nicht definiert wurde (*ifndef*). Falls die Bedingung nicht eintrifft, wird in den *else*-Teil (optional) gesprungen:

```
#ifdef SYMBOL
//Wird eingefügt, wenn SYMBOL definiert wurde
#else
//Wird eingefügt, wenn SYMBOL nicht definiert wurde
#endif
```

Das »#endif« schließt den bedingten Teil ab.

```
#ifndef SYMBOL
//Wird eingefügt, wenn SYMBOL nicht definiert wurde
#else
//Wird eingefügt, wenn SYMBOL definiert wurde
#endif
```

Ein Symbol wird über den *define*-Befehl definiert:

```
#define SYMBOL
```

Eine weitere Möglichkeit, die Einbindung mit einer Bedingung zu verknüpfen, ist die *if*-Anweisung:

```
#if AUSDRUCK
```

Ein gültiger Ausdruck ist dabei ein boolescher Wahrheitswert. Ist der Wert, der sich aus dem Ausdruck ergibt, »0«, ist die Bedingung nicht erfüllt, ansonsten ist sie erfüllt. Dies könnte z. B. ein Vergleich von zwei Werten sein. Bei der Abfrage, ob sie gleich sind, käme im Fall einer Gleichheit der Wert *1* (true) heraus, andernfalls »0« (false).

Bisher war ein Symbol einfach nur definiert. Es ist aber auch möglich, dem Symbol bei der Definition einen Wert zuzuweisen:

```
#define SYMBOL   WERT
```

Wenn einem Symbol ein solcher Wert zugewiesen wurde, kann es dennoch weiterhin wie oben beschrieben zur bedingten Einbindung verwendet werden. Gleichzeitig ist es aber auch möglich, das Symbol innerhalb des Programmtexts zu verwenden. Während des Kompilierens wird im Programmtext das Symbol durch den Wert ersetzt. Dies kann nützlich sein, wenn bestimmte Zahlen eine Bedeutung haben. Anstelle der Zahl kann dann zur besseren Lesbarkeit ein Symbol, das die Bedeutung der Zahl im Namen trägt, mit dem Wert der Zahl definiert werden.

Beispiel: In C gibt es keinen eigenen Variablentyp für boolesche Wahrheitswerte. Die Implementation könnte daher folgendermaßen aussehen:

```
#define TRUE 1
#define FALSE 0
```

Der Wert des Symbols muss allerdings keine Zahl sein. Es ist möglich, beliebige Zeichenketten einfügen zu lassen. So könnte das Symbol auch ein Ausgabebefehl

oder der Vergleich von zwei Variablen sein. Der Fantasie sind dabei kaum Grenzen gesetzt.

Es ist weiterhin möglich, die Symbole zu parametrisieren und somit komplette Funktionen oder Code-Abschnitte als Symbol zu definieren.

Beispiel:

```
#define EQUALS(VAR1,VAR2)   (VAR1==VAR2)
```

Selbst mehrzeilige Anweisungsblöcke sind möglich:

```
#define SWAP(VAR1,VAR2)     int swap_tmp = VAR1; \
   VAR1=VAR2; \
   VAR2 = swap_tmp;
```

5.4 Variablen und Typen

In einem Programm gibt es viele Stellen, an denen es notwendig ist, Informationen kurzfristig abzuspeichern oder zu ändern. Dies geschieht in sogenannten *Variablen*.

Variablen haben eine symbolische Adresse (den Namen der Variablen), den der Programmierer definieren muss. Weiterhin muss er festlegen, was für einen Variablentyp er benutzen möchte. Die grundlegenden Variablentypen unterscheiden sich durch die Größe, Verwendungsart und Vorzeichenbehaftung.

Die Vorzeichenbehaftung sagt aus, ob die Variable auch negative Werte annehmen kann oder nicht. Mit *Größe* ist die Anzahl der Bytes, die die Variable umfasst, gemeint. Jede Variable kann nur eine Zahl ganzer Bytes belegen. Die Anzahl Bytes entspricht einer Potenz von 2. Die tatsächliche Größe der Variablen ist plattformabhängig. Deshalb werden im Folgenden standardisierte Typen vorgestellt, in denen Größe und Vorzeichenbehaftung im Namen ablesbar sind.

Die Verwendungsart ist gleichbedeutend mit dem Unterschied zwischen ganzen Zahlen und Gleitkomma-Zahlen.

Ganzzahlige Datentypen:

Typ	*Größe*	*Vorzeichenbehaftet*	*Wertebereich*
int8_t | signed char	1 Byte	Ja	–128 bis 127
uint8_t | unsigned char	1 Byte	Nein	0 bis 255
int16_t | signed short	2 Byte	Ja	-2^{15} bis $2^{15}-1$
uint16_t | unsigned short	2 Byte	Nein	0 bis $2^{16}-1$
int32_t | signed long	4 Byte	Ja	-2^{31} bis $2^{31}-1$
uint32_t | unsigned long	4 Byte	Nein	0 bis $2^{32}-1$

Diese Datentypen sind in der Header-Datei *stdint.h* definiert. Ein int8_t entspricht dem signed char, ein uint8_t, dem unsigned char. Nach dem Muster entsprechen die 2 Byte großen Typen *unsigned* und *signed short* und die 4 Byte großen *unsigned* und *signed long*.

Die Größe einer *int*-Variablen ist plattformabhängig. Unter Verwendung von AVR-Studio und dem mitgelieferten Compiler ist die Größe einer *int*-Variable jedoch äquivalent zu der Größe einer *long*-Variablen.

Wenn man bei der Definition einer Variablen den Zusatz *signed/unsigned* weglässt, wird die Variable automatisch als *signed* definiert. Eine Ausnahme davon bildet der Typ *char*. Dieser Variablentyp ist ohne Zusatz zur Aufnahme eines einzelnen ASCII-Zeichens. Eine Menge von *char*-Variablen dient folglich der Darstellung einer Zeichenkette (String).

Die Gleitkommatypen sind immer vorzeichenbehaftet. Dabei wird zwischen *float* und *double* unterschieden.

Eine Variable des Typs *float* ist 4 Byte lang und deckt einen Wertebereich von $-3{,}4*10^{38}$ bis $3{,}4*10^{38}$ ab. Eine *double*-Variable ist doppelt so lang (8 Byte) und deckt hingegen einen Bereich von $-1{,}8*10^{308}$ bis $1{,}8*10^{308}$ ab.

Eine Definition einer Variable sieht nun folgendermaßen aus:

```
VAR-TYP VAR-NAME;
```

Beispiel:

```
int16_t result;
```

In diesem Beispiel haben wir eine 2 Byte große ganzzahlige Variable mit dem Namen *result* definiert.

Eine Wertzuweisung ist folgendermaßen möglich:

```
VAR-NAME = WERT;
→ result = 0;
```

Zudem ist es möglich, einer Variablen bei der Definition einen Startwert zuzuweisen:

```
VAR-TYP VAR-NAME = WERT;
→ int16_t result = 0;
```

Abgesehen von dem Variablentyp kann man allerdings auch noch optionale Präfixe bei der Definition verwenden.

Das Präfix *const* bedeutet, dass der Wert der Variablen durch eine Wertzuweisung im Programmtext nicht geändert werden kann.

volatile sollte als Präfix vor Variablen stehen, die in Interrupt-Routinen verwendet werden. Es stellt sicher, dass Variablen vor jedem Zugriff aus dem Speicher gelesen und nach jeder Änderung direkt wieder in den Speicher zurückgeschrieben werden.

Falls dies nicht der Fall wäre, könnte eine Variable im Hauptprogramm geändert worden sein. Ein Interrupt tritt auf, die Routine, die den Interrupt verarbeitet, findet jedoch noch den alten Wert der Variablen im Speicher und reagiert möglicherweise falsch.

Die Definition einer Variablen kann somit auch wie folgt aussehen:

```
PRÄFIX VAR-TYP VAR-NAME = WERT;
→ volatile int16_t result = 0;
```

5.5 Schreibweisen von Konstanten

In C gibt es verschiedene Möglichkeiten, Konstanten aufzuschreiben. Welche man wählt, liegt beim Programmierer und ist für das Programm selbst nicht relevant.

Bei Gleitkommazahlen wird häufig die gewohnte Schreibweise in Dezimalzahlen bevorzugt. Ganze Zahlen werden jedoch zumeist im Hexadezimal- oder dem Binärsystem codiert.

Die binäre Schreibweise wird nur äußerst selten verwendet. Sie ist allerdings hilfreich, wenn man in einem Port vom Mikrocontroller Bits setzt. Auf diese Weise sieht man sofort, welche der Bits gesetzt werden und welche nicht:

```
53 (dezimal) = 0b00101011 (binär)
```

Wie zu sehen ist, wird eine binär codierte Zahl von dem Präfix »0b« eingeleitet.

Die hexadezimale Schreibweise wird allerdings häufiger eingesetzt. Aufgrund der 16 Symbole (im binären zwei) sind die Zahlen wesentlich kürzer zu schreiben. Genau genommen fassen sie immer vier Ziffern einer Binärzahl zusammen, sodass sich immer noch die Werte der einzelnen Bits ablesen lassen. Zudem kann man anhand der Länge der Zahl feststellen, wie viele Bytes sie belegt (1 Ziffer hexadezimal = 4 Ziffern binär => 2 Ziffern hexadezimal = 1 Byte Speicherplatz).

Die Schreibweise erfolgt analog zum Binärsystem, allerdings mit einem »0x« als Präfix:

```
53 (dezimal) = 0x2B (hexadezimal)
```

Auch wenn man eine Weile benötigt, um mit Zahlen des Hexadezimalsystems auch im Kopf gut rechnen zu können, sind sie vorteilhaft, weil sie näher an der Maschine sind. Dies liegt daran, dass die Basis eine Potenz von 2 ist, was dazu führt, dass die Zahl der Ziffern sofortige Rückschlüsse auf den Speicherbedarf ermöglicht.

Zuletzt ist es selbstverständlich auch möglich, Werte als Dezimalzahlen zu verwenden. Dies ist bei sehr kleinen (0, 1, –1 etc.) und einzelnen feststehenden Werten manchmal kürzer in der Schreibweise. Letztendlich muss aber jeder selbst wissen, wie er die Werte am besten lesen kann. Doch wenn man gezwungen ist, schon häufig in binär und hexadezimal zu denken, gewöhnt man es sich auch an, die Werte konstant als Hexadezimalzahlen aufzuschreiben, um nicht durch unterschiedliche Schreibweisen zu verwirren.

5.6 Arrays

Bisher haben Sie nur erfahren, wie man einzelne Variablen definiert. Um mehrere Variablen eines Typs gleichzeitig zu definieren, gibt es mehrere Möglichkeiten. Die erste einfache Methode ist, mehrere Variablennamen bei der Definition hintereinander und durch ein Komma getrennt zu schreiben:

```
vartyp var1, var2, var3,..., varn;
```

beispielsweise `int16_t result, count;`

Auch die direkte Initialisierung ist hierbei möglich:

```
int16_t result = 0, count = 0;
```

Diese Methode erspart jedoch nur wenig Schreibarbeit und ist nur bei einer geringen Anzahl von Variablen nützlich. Wenn man eine Liste von 100 Zahlen verarbeiten möchte, ist diese Methode sehr aufwendig. Als Lösung gibt es *Arrays*.

Arrays sind Listen eines Variablentyps, die unter einem gemeinsamen Variablennamen mithilfe eines Index angesprochen werden können. Bei der Definition eines Arrays wird die Größe mit angegeben. Der Index eines Arrays beginnt bei 0 und endet mit der Zahl der Elemente minus 1.

Beispiel: Wir wollen das Alter von 100 Personen in einer Liste/einem Array erfassen.

Definition:

```
uint8_t ages[100];
```

Wertzuweisung:

```
ages[0] = 13;
ages[1] = 42;
      …
ages[99] = 23;
```

Anstelle einer Konstante kann selbstverständlich, wie bei der Wertzuweisung selbst, auch der Index eine Variable sein:

```
uint8_t index = 0,wert = 21;
ages[index] = wert;
```

Doch auch mehrdimensionale Felder sind durchaus möglich. Dies bedeutet, dass der Zugriff auf ein Element zwei oder mehr Werte erfordert (einen Wert pro Dimension).

Anschaulicher könnte man sich die Erweiterung erst von einer Strecke, auf der die Elemente liegen, zu einem rechteckigen Gitter vorstellen.

Statt der Position auf der Strecke braucht man nun eine Spalte und eine Zeile, um die Position des Elements zu bestimmen. Nach diesem Muster lässt sich eine Erweiterung auf beliebig viele Dimensionen vorstellen.

Eine Definition eines solchen mehrdimensionalen Arrays könnte wie folgt aussehen:

```
uint8_t array[2][2][2][2][2];
```

und die dazugehörige Wertzuweisung beispielsweise:

```
array[1][0][0][1][0] = 23;
```

Die Organisation eines Arrays im Speicher ist linear. Das bedeutet, dass die Elemente in einem Block hintereinanderliegen. Bei mehrdimensionalen Arrays ist dies ebenfalls der Fall. Dabei wechselt der letzte Index am schnellsten und der vorderste am langsamsten (Abb.5.2).

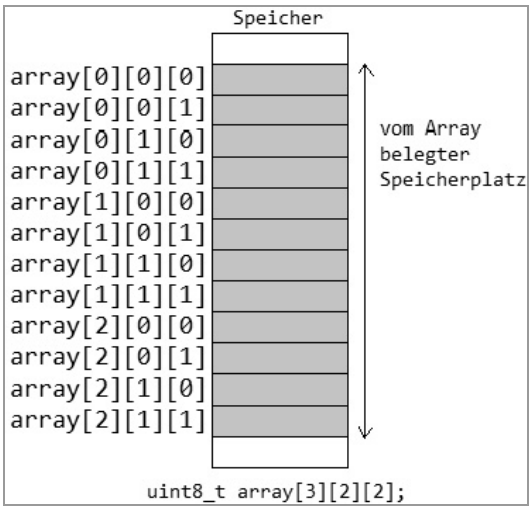


Abb. 5.2: Organisation eines mehrdimensionalen Arrays im Speicher

5.7 Pointer

Bisher hatten wir als Variablentypen immer die Grunddatentypen selbst. Dabei wurde jedoch außer Acht gelassen, dass der Inhalt der Variablen an einer bestimmten Stelle im Speicher steht und diese Stelle im Speicher über eine bestimmte Adresse ansprechbar ist.

Die Adressierung von Speicherplätzen ist das Thema dieses Kapitels.

Mithilfe sogenannter *Pointer* kann man die Startadresse einer Variablen im Speicher verwenden. Jede Pointer-Variable hat die gleiche Länge (im Fall der hier verwendeten CPU 16 Bit).

Weil der Inhalt dieser Pointer-Variablen (Adresse des Speicherplatzes, auf den der Pointer »zeigt«) selbst ebenfalls im Speicher steht, ist es möglich, einen Pointer auf einen Pointer zeigen zu lassen.

Die Deklaration eines Pointers auf eine Variable eines Typs unterscheidet sich von der bisherigen Deklaration dadurch, dass ein * vor den Namen der Variablen gesetzt wird:

```
int16_t *result;
```

Dies ist die Deklaration des Pointers *result*, der auf eine 16-Bit-Integer-Variable zeigt.

Um den Inhalt der Pointer zu ändern oder zu verwenden, benutzt man ihn wie eine normale Variable mit seinem Namen. Wir haben zwei Pointer desselben Typs: *result_old* und *result_new*.

```
result_new = result_old;  //sieheAbb.5.3
```

Bei diesem Befehl wird der Inhalt von *result_new* mit dem Inhalt von *result_old* überschrieben. Somit zeigen nun beide Pointer auf denselben Speicherplatz. An dem Inhalt des Speicherplatzes, auf den *result_new* gezeigt hat, ändert dies nichts. Wenn die Adresse aber sonst an keiner Stelle abgespeichert war, kann auf diesen Speicherplatz nicht mehr zugegriffen werden. Falls er nicht vorher freigegeben wurde, kann er bis zum Neustart des Systems nicht mehr verwendet werden.

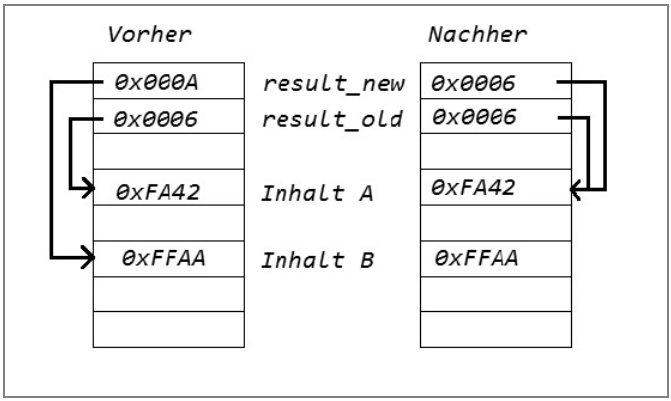


Abb. 5.3: Pointer und Variablen im Speicher

Diese Grafik zeigt, dass die Pointer ebenso wie normale Variablen im Speicher liegen. Bei einer Änderung des Pointers ändert sich an dem Inhalt des Speicherplatzes, auf den er zeigt, nichts.

Beim Zugriff auf den Speicherplatz, auf den der Pointer zeigt, gibt es folgende Schreibweise:

```
int16_t *ptr;
int16_t zahl = 0;
int16_t zahl2;
ptr = &zahl;
//Durch das vorgezogene "&" bekommen
//wir die Adresse des Speicherplatzes in
//dem der Inhalt von "zahl" gespeichert ist
*ptr = 20;
//durch das vorgezogene "*" wird der Wert
//20 in die Speicherstelle von "zahl"
//geschrieben
zahl2 = *ptr;
//die 20 wird an die Speicherstelle von
//"zahl2" kopiert. Dieser Befehl wäre
//äquivalent zu "zahl2 = zahl";
```

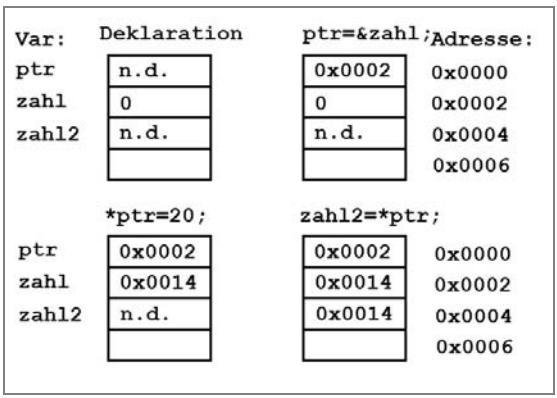


Abb. 5.4: Variablenzugriff mit Pointern

Da man theoretisch jeden beliebigen Speicherplatz adressieren kann, also auch jede Variable jeden beliebigen Typs, kann man auch einen Pointer auf einen Pointer zeigen lassen. Auch die Deklaration dieses Pointers auf einen Pointer erfolgt analog zur Deklaration eines Pointers auf eine Variable eines Grundtyps:

Vor den Namen der Variablen wird ein zweites * gesetzt. Somit hat man einen Pointer, der auf einen Pointer zeigt, der wiederum auf eine Variable eines bestimmten Typs zeigt.

```
int16_t **ptrptr;
int16_t *ptr;
int16_t zahl;
ptr = &zahl;
ptrptr = &ptr;
```

Jetzt ist der Befehl `**ptrptr=0x14;` äquivalent zu `zahl=0x14;` oder `*ptr=0x14;`.

Man kann auch einen Pointer auf einen Pointer zeigen lassen, der selbst wiederum auf einen Pointer zeigt usw. Man möchte aber nicht immer für jede Pointer-Variable auch eine Variable des Typs anlegen, um den Pointer darauf zeigen zu lassen. Deshalb gibt es für viele Plattformen die Möglichkeit, direkt Speicherplatz einer beliebigen Größe von der Speicherverwaltung anzufordern und die Startadresse dieses freien Speicherplatzes dem Pointer zuzuweisen. Dies ist jedoch nur mit einer MMU möglich, die in AVR-Controllern nicht vorhanden ist.

Zugleich bietet diese Speicherallokation eine weitere Möglichkeit, ein Feld von Variablen zu verwenden. Um das zu verstehen, muss man nochmals zurückdenken, wie ein Array im Speicher liegt. Gehen wir erst einmal von einem eindimensionalen Array aus.

Es wurde festgestellt, dass die Elemente des Arrays linear im Speicher abgelegt werden. Das Array zeigt also auf die Startadresse des belegten Speicherblocks. Der Index besagt nur, wie oft man die Variablenlänge auf diese Adresse addieren muss, um zu der Adresse des gesuchten Elements zu gelangen.

Mit dieser Darstellung sind wir auch schon fast bei der Ansteuerung mithilfe eines Pointers angelangt. Der Pointer zeigt auf die Startadresse des Arrays. Wenn man ihn dann um die Länge des Variablentyps erhöht, zeigt er auf das nächste Element des Arrays.

Wenn man kein bestehendes Array hat, kann man der Pointer-Variablen über die oben erwähnte Möglichkeit einen Speicherblock zuweisen. Dabei wird die Größe des Speicherblocks nicht nach der Größe einer einzelnen Variablen des Typs gewählt. Die Größe einer einzelnen Variable des Typs wird mit der Anzahl der Variablen, die das Feld enthalten soll, multipliziert. Wie dies im Code aussieht, wird an dieser Stelle nicht erläutert, da die Allokation eine Aufgabe der MMU (Memory Management Unit) und diese in AVR-Controllern nicht vorhanden ist.

Beispiel: Normalerweise würden wir für einen Pointer auf eine *int16_t*-Variable 2 Byte reservieren. Wenn das Feld allerdings fünf dieser Variablen hintereinander enthalten soll, reservieren wir 10 Byte, auf deren Startadresse der Pointer zeigen soll.

Wenn hingegen ein bestehendes Array vorhanden ist, lässt sich die Startadresse folgendermaßen übermitteln:

```
int16_t arrray[10];
int16_t *ptr;
ptr = array;
```

Nun wäre der Befehl

```
*ptr = 0x17;
```

äquivalent zu

```
array[0] = 0x17;
```

5.8 Strukturen, Unions, Enumerations

Bis jetzt haben wir immer nur grundlegende Datentypen verwendet. Bei komplexeren Programmen wird es jedoch unabdingbar, selbst neue Typen zu definieren, die unter anderem auch mehrere Variablen verschiedener Typen zusammenfassen können. Manchmal möchte man auch Statusabfragen realisieren. Der Status

Stichwortverzeichnis

FRANZIS
PC & ELEKTRONIK

Sven Feldkord

Einführung in die C-Programmierung mit dem ATmega32

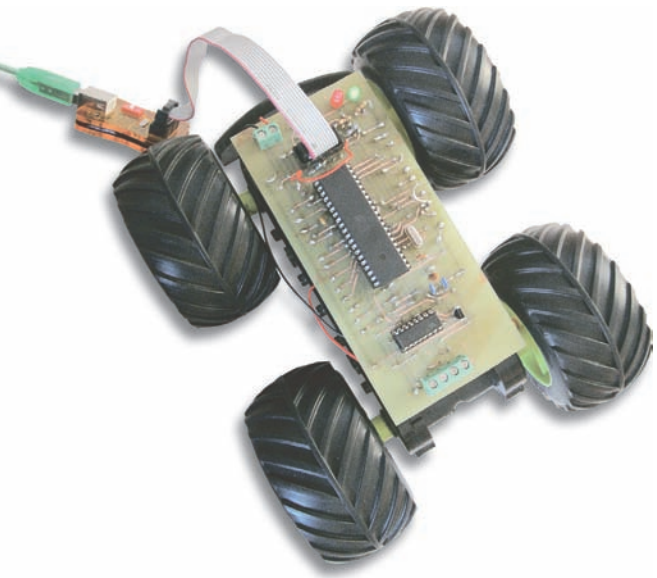
Immer häufiger begegnen wir im Alltag eingebetteten Systemen: dem Auto, das elektronische Unterstützung durch einen Bordcomputer erhält, oder der digitalen Zeitanzeige bzw. Stoppuhr an einem Backofen oder einer Mikrowelle. Alle diese Dinge werden elektronisch gesteuert und sind daher so etwas wie kleine Computer. Denn wie ein Computer bestehen diese Dinge aus einer Hardware und einer Software, die von der Hardware ausgeführt wird.

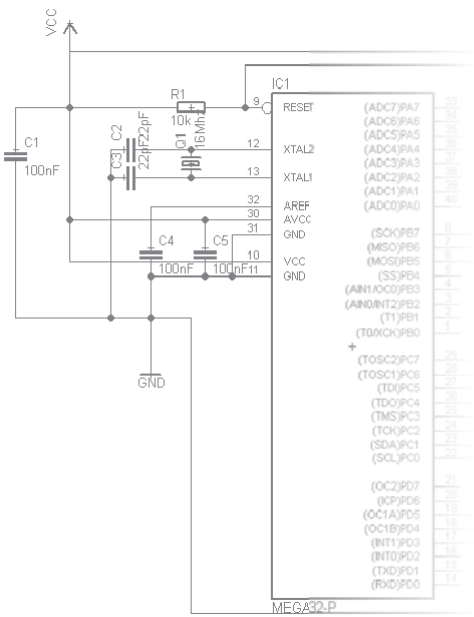
Dieses Buch ermöglicht Ihnen den Einstieg in die Welt der eingebetteten Systeme anhand eines Mikrocontrollers der Reihe Atmel-AVR. Am Beispiel des Mikrocontrollers ATmega32 werden die Grundlagen der Funktionsweise von Mikrocontrollern und Grundkenntnisse der Programmiersprache C vermittelt und anhand praktischer Anwendungen verdeutlicht.

Hardwareunterstützte Übertragungsprotokolle, Analog-Digital-Wandler oder interne Speichereinheiten – Schritt für Schritt werden Sie an die verschiedensten Funktionen herangeführt und durch einige abschließende Programmbeispiele zu eigenen Projekten angeregt.

Aus dem Inhalt:

- Mikrocontroller und Programmierung
- Grundlagen der Programmierung
- C-Programmierung
- Aufbau eines C-Programms
- Praxisanwendungen und Beispiele





ISBN 978-3-645-65060-1

Euro **19,95** [D]