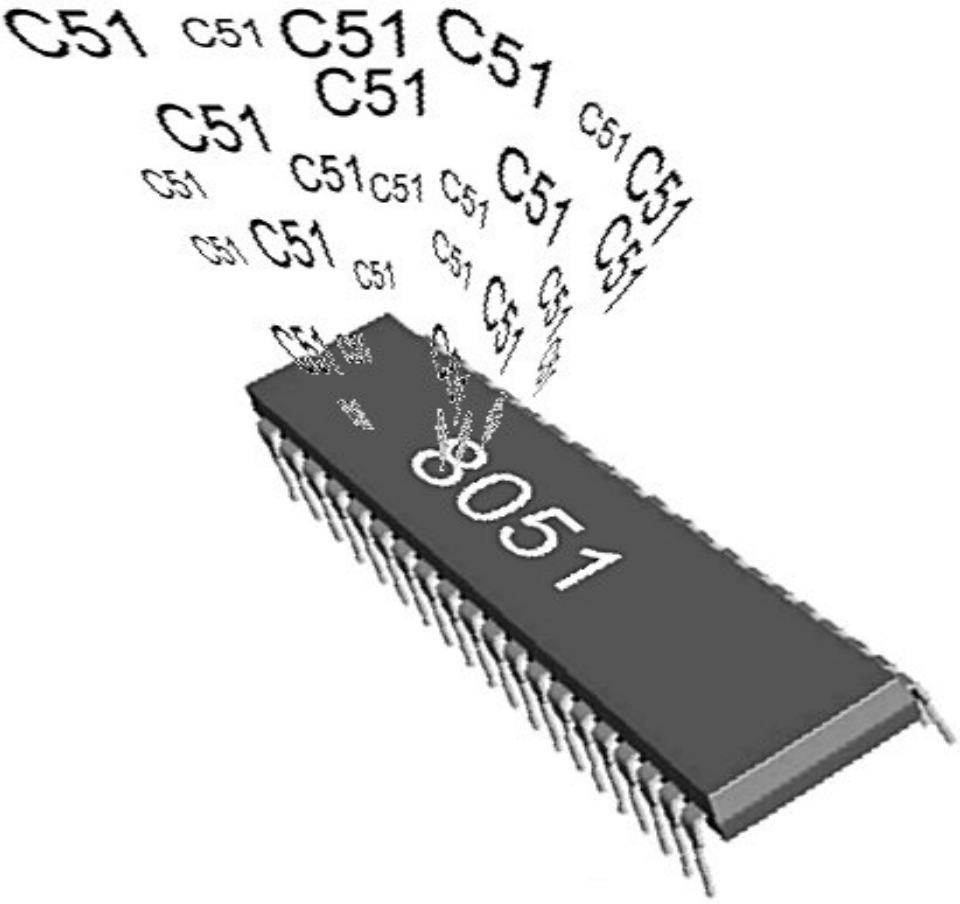


Den 8051 in

C51

programmiert

Peter Erwin

Den 8051 in C51 programmiert

Programmierung von Mikrocontrollern der 8051-Familie in der Hochsprache C

Dipl.-Ing. (BA) Peter Erwin

1. Auflage

23. September 2003

Dieses Buch ist ein Kompendium zur Programmiersprache C, mit den Sprach-erweiterungen für die 8051-Controller-Familie. Es zeigt praxisnahe Beispiele im Umgang mit dem C51-Compiler der Fa. KEIL-Elektronik für alle 8051-Derivate. Es dient zu Lehrzwecken und als Nachschlagewerk.

Peter Erwin

Vorwort

C ist eine allgemein einsetzbare Programmiersprache, die Code-Effizienz, strukturiertes Programmieren, komfortable Datenstrukturen und einen reichhaltigen Satz von Operationen vereinigt. C ist eine höhere Programmiersprache, die in keiner Weise auf bestimmte Anwendungen spezialisiert ist. Trotzdem lassen sich viele Anwendungen auf Grund der Allgemeinheit der Sprache komfortabler und effizienter lösen als mit spezialisierten Programmiersprachen.

Dieses Buch zeigt die Elemente der Programmiersprache C, wobei hier auf die Spracherweiterungen für die 8051-Controller-Familie eingegangen und deshalb von "C51" gesprochen wird. Im Handel sind zur Programmiersprache C verschiedene Bücher erschienen. Das wohl bekannteste Standardwerk sei nachfolgend insbesondere wegen seiner Vollständigkeit des gesamten ANSI-C-Sprachumfangs erwähnt:

Programmieren In C
Zweite Ausgabe ANSI C in deutscher Sprache
Kernighan, Ritchie
Hanser Verlag
ISBN 3-446-15497-3-1

Der C51-Compiler der Firma KEIL-Elektronik ist ein Cross-Compiler, der speziell für die 8051 Familie entwickelt wurde. Er ist ein erweiterter ANSI-C-Compiler, der vollen Zugriff auf alle Ressourcen eines 8051 Systems zuläßt. Der generierte Code ist von der Ausführungsgeschwindigkeit und der Effizienz insbesondere bei steigender Versionsnummer mit der in Assembler geschriebenen Programmen vergleichbar.

Mit diesem Werk ist mir daran gelegen ein Gefühl dafür zu bekommen, nach den Maßstäben der Informatik ein vernünftiges Programm in der Sprache C51 zu schreiben. Es soll auch ein Nachschlagewerk sein.
Es ist nicht Ziel die Programmiersprache C mit all seinen Sprachelementen vollständig wiederzugeben. Auch geht es nicht darum das Handbuch des KEIL-C51-Compilers sowie das des Linkers und anderer Werkzeuge zu zitieren. Daher werden vom C51-Compiler und den weiteren zur Erstellung eines Programms benötigten Werkzeuge nur die wichtigen Elemente jedoch nicht eine komplette Entwicklungsumgebung (z.B. unter Windows) gezeigt. Ausführlichere und IDE-bezogene Aufrufmöglichkeiten sind den zugehörigen Handbüchern zu entnehmen.
Neben der Behandlung von Sprachelementen enthält diese Ausgabe Richtlinien im Umgang mit der Sprache C zur Erstellung eines *qualitativ wertvollen* Programms. Die Schreibweisen der Bezeichner in allen Beispielen basieren auf diesen Richtlinien. Ergänzend werde ich wichtige Dinge aus meiner 10-jährigen Programmiererfahrung der 8051-Controller-Familie in der Trickkiste zusammenfassen.
Schlüsselworter und Wörter ähnlicher Bedeutung werden ***kursiv*** gedruckt, C-Beispiele und Bezüge hierauf werden in der Schriftart `Courier` wiedergegeben.

Dank an:
- die Herren Brian W. Kernighan und Dennis M. Ritchie, die mit "C" eine Programmiersprache geschaffen haben, die wohl die universellste und flexibelste Hochsprache ist, und die besonders auch das Spektrum von maschinennah bis maschinenfern abdeckt, sowie dem Hanser-Verlag, bei dem deren Buch dazu erschienen ist.
- den Markt&Technik Verlag (Rainer G. Haselier, Quick-C 2.0: Schnellübersicht)
- die Fa. KEIL-Elektronik (C51-Compiler V3.20, V3.40, V4.06, V5.10 und V7.01)
- Oliver Seebach, dessen Informatik-Kenntnisse wegweisend für meine Denkrichtung in Bezug auf "vernünftiges Programmieren" waren.

P. Erwin

1 C(51) und der ANSI-Standard

Im Gegensatz zu anderen Hochsprachen, wie beispielsweise Pascal oder Basic, die durch Universitätsgremien als Lehrsprachen konzipiert wurden, ist C eine Sprache, die aus dem "wirklichen" Leben entstanden ist und für konkrete Probleme der Praxis entworfen wurde. Die beiden geistigen Väter von C, Brian W. Kernighan und Dennis M. Ritchie, benötigten eine Sprache für die Systemprogrammierung, die darum sehr schnellen und kompakten Code erzeugen, und darüber hinaus möglichst portabel sein sollte. Die Sprache sollte Fähigkeiten einer echten Hochsprache haben, da sich herausgestellt hatte, daß Sprachen auf einem sehr maschinennahen Niveau (Assembler) bei der Entwicklung von großen Software-Projekten schnell zu unübersichtlichen und unportablen Programmen führen. Diese Zielsetzung wurde in einer Art und Weise umgesetzt, die als außerordentlich gelungen bezeichnet werden kann. Seitdem C-Compiler auch für Mikro-Computer verfügbar sind, läßt sich der Siegeszug von C kaum noch aufhalten. C51 ist ein C-Derivat für Mikrocontroller, das aus der C-Urform entstanden ist.

1.1 Der ANSI-Standard

Die hinter der Programmiersprache C stehende Philosophie der Portierbar- und Erweiterbarkeit hat im Laufe der Zeit zu einer Vielzahl von neuen Bibliotheksfunktionen geführt, die die Portierbarkeit wieder schwieriger machten. Darüber hinaus haben verschiedene Compiler-Hersteller die Sprache erweitert und ihren Produkten neue Features hinzugefügt, die von anderen Compilern aber nicht unterstützt wurden. Um diesem Problem auf den Leib zu rücken, wurde im Jahre 1983 im American National Standards Institute (ANSI) das Komitee X3J11 gegründet, das einen Standard für die Programmiersprache C erarbeiten sollte. Das Komitee bestand aus 130 Mitgliedern, die sich aus Vertretern von Compiler-Herstellern, aus Software-Entwicklern, sowie aus Hardware-Herstellern zusammensetzten. Das ANSI-C-Komitee hat sich seit seiner Gründung ungefähr viermal pro Jahr getroffen. KEIL-C51 ist ein C Übersetzer, der dem ANSI-Standard entspricht.

1.1.1 Richtlinien des ANSI-Komitees

Der große Erfolg des ANSI-Komitees und die weite Akzeptanz, die der Standard bereits in seiner vorläufigen Fassung hatte, ist nicht zuletzt den Prinzipien zu verdanken, nach denen das Komitee seine Arbeit ausrichtete.

"Existierender Code ist wichtig, existierende Implementierungen nicht."

Wichtig bei der Festlegung des Standards war die Zielsetzung, soviel existierende Codes wie möglich unverändert lauffähig zu halten. Sollte dies nur um den Preis möglich sein, daß bestehende Compiler-Implementierungen verändert werden müssen, soll dieser Preis gezahlt werden.

"C-Code ist portabel."

Der vom Komitee beschriebene ANSI-Standard gibt dem Programmierer eine reelle Chance, daß, hält er sich an die Vorgaben von ANSI-C, sein Programm ohne Änderungen auf anderen Betriebssystemumgebungen portiert werden kann.

"C-Code kann auch nicht portabel sein."

Der Programmierer soll durch den Standard nicht in seiner Freiheit eingeschränkt werden, auch Programmcodes zu schreiben, die nicht portabel und damit an eine bestimmte Hardware-Umgebung gebunden sind.

"Vermeide stillschweigende Änderungen."

Mit diesem Leitsatz sollte vermieden werden, daß sich ein korrekt geschriebenes Programm, wenn es mit einem ANSI-kompatiblen Compiler übersetzt wird, anders verhält, als wenn es durch einen Compiler aus der Vor-ANSI-Zeit geschickt wird. Sollte das Programm sich anders verhalten, dann soll der Programmierer durch eine Warnung darauf aufmerksam gemacht werden.

"Der Standard ist ein Übereinkommen zwischen dem Implementator und dem Programmierern"
Mit diesem Leitsatz sollten beide Seiten (die Compiler-Hersteller und die Compiler-Nutzer) zu ihrem Recht kommen. Bei Änderungen, die vereinbart wurden, sollten die Anforderungen des Implementators und des Programmierers zu einem Kompromiß verknüpft werden.

1.1.2 Der Geist von C

Darüber hinaus sollte der Geist von C, wie er von Kernighan & Ritchie beschrieben wurde, beibehalten werden.

- "Vertraue dem Programmierer"
- "Hindere den Programmierer nicht daran, das zu tun, was getan werden muß."
- "Die Sprache soll klein und einfach bleiben."
- "Stelle nur einen Weg zur Verfügung, um eine Operation durchzuführen."
- "Mache es schnell, auch wenn die Garantie der Portabilität dabei verlorengeht."

1.1.3 Die wichtigsten Neuerungen

Neben der Beibehaltung der grundlegenden Prinzipien erarbeitete das Komitee auch einige Änderungen, von denen die wichtigsten nachfolgend kurz beschrieben werden.

1.1.3.1 Funktions-Prototypen

Die Aufnahme der Funktions-Prototypen in den ANSI-Standard war wohl die wichtigste Neuerung, die C durch das Komitee widerfuhr. Bei Funktions-Prototypen handelt es sich um Deklarationen von Funktionen. Bei diesen Deklarationen kann nun auch der Typ der Funktionsargumente und der Rückgabewert der Funktion angegeben werden. Durch diese Prototypen wird der Compiler in die Lage versetzt, an allen Stellen, an denen eine Funktion aufgerufen wird, und an der Stelle, an der die Funktion definiert wird, zu überprüfen, ob die aktuellen Argumente mit den bei der Deklaration festgelegten übereinstimmen. Damit kann der Programmierer bereits bei der Kompilierung des Programms auf mögliche Fehlerquellen hingewiesen werden.

1.1.3.2 Schlüsselworte

Die Schlüsselworte *const*, *signed*, *void* und *volatile* wurden offizieller Bestandteil der Sprache. Das alte Schlüsselwort *entry*, das aus der Sprachdefinition von K&R stammte, wurde entfernt.

1.1.3.3 *void** als genereller Zeiger

Das Schlüsselwort *void*, das bereits von vielen Compilern unterstützt wird, ist jetzt offizieller Bestandteil der Sprache. Der Typ *void**. wird jetzt als allgemeiner Zeiger benutzt und nimmt die Rolle von c*har** ein. Auf diese Weise wird der Zeiger auf ein Zeichen eindeutig.

1.1.3.4 Funktionsbibliothek

Im Gegensatz zu K&R definiert der ANSI-C-Standard die Laufzeitbibliothek detaillierter. Insgesamt 127 Funktionen wurden in die Bibliothek aufgenommen, mit denen die meisten der alltäglichen Programmaufgaben erledigt werden können. Die Verwendung dieser Funktionen erweckt das Prinzip der leichten Portierbarkeit wieder mit Leben, da diese Funktionen bei allen ANSI-C-Compilern, unabhängig vom benutzten Betriebssystem, verfügbar sein sollen.

1.1.3.5 Grenzwerte

ANSI C legt für alle arithmetischen Datentypen Grenzwerte (Minimum und Maximum) fest, die in den Header-Dateien *float.h* und *limits.h* definiert sind.

1.2 Die Struktur eines C-Programms

C-Programme bestehen aus einer unbegrenzten Anzahl von Funktionen, von denen eine den Namen *main()* haben muß. Dieser Funktion wird nach dem Programmstart die Kontrolle übergeben. Sie ist also die erste, die nach der Initialisierungsphase aufgerufen wird. An einem einfachen und viel zitierten Beispiel kann man die Grundelemente von C-Programmen verdeutlichen. Dieses Programm gibt, einmal gestartet, den Text "`Hello world.`" aus.

```
#include <stdio.h>

main()
{
printf("Hello world.");
}
```

1.2.1 Anweisungsblöcke

Im Beispiel sieht man nach dem Namen *main()* eine öffnende geschweifte Klammer ´{´ und am Ende des Programms eine schließende geschweifte Klammer ´}´. Diese Klammern umgeben eine Gruppe von Befehlen (Anweisungen), die auch Anweisungsblock genannt werden. Die geschweiften Klammern haben die gleiche Funktion wie die BEGIN- und END-Anweisungen in Pascal-Programmen. Gleichzeitig verdeutlichen sie die Philosophie der syntaktischen Prägnanz: Warum mehrere Zeichen verwenden, wenn ein Zeichen schon ausreicht.

1.2.2 Anweisungsbegrenzer

Jede Programmanweisung muß mit einem Semikolon abgeschlossen werden. Darum heißt das Semikolon Anweisungsbegrenzer.

1.2.3 Kommentare

Kommentare beginnen in C mit der Zeichenkombination Schrägstrich-Stern "/*". Alle Zeichen, die nach diesen Zeichen stehen, werden vom Compiler als Kommentar behandelt, den er nicht weiter auszuwerten braucht. Zwischen diesen beiden Zeichen darf kein Leerzeichen stehen. Mit der Zeichenkombination Stern-Schrägstrich "*/" wird der Kommentar beendet. Kommentare dürfen sich in C auch über mehrere Zeilen erstrecken. Sie dürfen jedoch nach dem ANSI-Standard nicht verschachtelt werden und nicht innerhalb von Zeichenkonstanten oder konstanten Zeichenketten auftreten.
Eine weitere Art des Kommentars ist der Zeilenendkommentar. Am Ende einer Programmzeile beginnt er mit zwei aufeinanderfolgenden Schrägstrichen "//" und endet automatisch mit dem Ende der Zeile. Bei C51-V3.20 kann es jedoch zu Compilerproblemen kommen, wenn im Kommentartext Umlaute oder sonstige Sonderzeichen enthalten sind, bei der Version V5.10 sind mir keine Probleme bekannt.

1.2.4 Ausgabe

Im Unterschied zu Sprachen wie Fortran oder Basic besitzt die Sprache C keine Möglichkeiten der Ausgabe. Der Grund ist, daß eine Programmiersprache die Möglichkeiten der Ausgabe eigentlich nicht kennen kann, da nicht von vorne herein bekannt ist, auf welches I/O-System die Ausgabe stattfinden soll. Dies bedeutet nicht, daß mit C-Programmen keine Ausgabe gemacht werden kann. Sondern die Ausgabe erfolgt mit Funktionen, die sich in einer besonderen Bibliothek befinden. *printf()* ist eine dieser Funktionen, mit denen eine formatierte Ausgabe ermöglicht wird. Da *printf()* kein integrales Sprachelement von C ist, wird zu Anfang des Programms mit der Anweisung *#include* eine Datei eingebunden, die den Prototyp der Funktion *printf()* enthält. Der Compiler kann somit überprüfen, ob die im Programm benutzten Argumente mit denen der Deklaration übereinstimmen.

1.2.5 Erlaubte Zeichen in Bezeichnern

Für die Festlegung der Namen von Bezeichnern, das sind Variablen-, Funktions- und Makronamen, dürfen folgende Zeichen benutzt werden:

- alle Buchstaben des englischen Alphabets (a bis z, A bis Z),
- die Ziffern 0 bis 9,
- der Unterstrich ´_´.

Der Name eines Bezeichners muß mit einem Buchstaben oder dem Unterstrich beginnen. Die Verwendung einer Ziffer als erster Buchstabe eines Bezeichners ist nicht erlaubt. Die deutschen Umlaute und andere Sonderzeichen dürfen also nur in Kommentaren und in Zeichenketten vorkommen. Die Sprache C unterscheidet zwischen Groß- und Kleinschreibung. Die Bezeichner `var1` und `Var1` können also für zwei verschiedene Variablen benutzt werden. Die maximale Länge der Bezeichner ist nicht vorgeschrieben, jedoch müssen nach ANSI C mindestens die ersten 31 Zeichen zur Unterscheidung herangezogen werden.
Es sei hier gleich vermerkt, daß die Namen von Bezeichnern möglichst sinnvoll sein sollten, damit ein Programm für den Programmierer selbst und auch für andere Menschen lesbar ist. Namen wie `A1`, `A2` oder `a`, `b` oder `c` sollten vermieden werden, da sie wenig aussagekräftig sind. Mehr dazu im Kapitel 7.

2 Variablen und Datentypen

Ganz egal, welche Art von Programm man schreibt: Die Hauptaufgabe besteht immer darin, Daten zu verarbeiten. Programmtechnisch werden diese Daten an verschiedenen Stellen im Arbeitsspeicher abgelegt. Um auf die Daten zugreifen zu können, ohne dafür die Adressen dieser Daten angeben zu müssen, werden relokatible Variablen benutzt. Über den Varaiblennamen kann auf den Speicherbereich zugegriffen werden, um die dort befindenden Daten eines Datentyps zu lesen oder Daten dorthin zu schreiben. Je nach Art der Daten, die aufgenommen werden sollen, können für die Variablen verschiedene Datentypen verwendet werden. Die Datentypen haben Einfluß auf die interne Darstellung der Variablen im Rechner, den Wertebereich, der in ihnen dargestellt werden kann, und den Speicherplatz, der von den Variablen belegt wird.

2.1 Definition und Deklaration von Variablen

Die beiden Begriffe Definition von Variablen und Deklaration von Variablen werden meist synonym benutzt, auch wenn sie eigentlich Unterschiedliches meinen. Diese Unterscheidung ist vor allem wichtig, wenn globale Variablen benutzt werden sollen. Mit der Deklaration einer Variablen wird ihr Typ, die Speicherklasse und Sichtbarkeit des Bezeichners festgelegt. Durch die Deklaration einer Variablen wird sie dem Compiler bekannt gemacht, jedoch kein Speicherbereich für die Variable zur Verfügung gestellt. Erst durch die Definition stellt der Compiler Speicherplatz zur Verfügung. Die Deklaration einer Variablen kann mehrfach, die Definition hingegen nur einfach erfolgen. Erfolgt die Definition gar nicht oder mehrfach, dann wird der Linker den Fehler melden.

2.2 Einfache Variablen

Die Programmiersprache C kennt die vier grundlegende Datentypen: *char*, *int, float* und *double*. Bei C51 kommt zusätzlich der Typ *bit* hinzu, der Typ *double* entfällt bzw. ist gleichbedeutend mit dem Typ *float*. Die beim 8051-Controller vorhandenen Special-Function-Register werden über die Datentypen *sfr*, *sfr16* und *sbit* definiert. Zeiger sind keine "einfachen Variablen" und werden in Kapitel 2.12 behandelt. Unter C51 werden die einfachen Variablen im big-endian-Format gespeichert, mit Ausnahme mit dem Datentyp *float*, der bis zur Version V4.x im little-endian-Format gespeichert und erst ab der Version V5.x angepaßt wurde. Wegen der Problematiken der unterschiedlichen Speichersyntax sei auf Kapitel 7.6 hingeweisen.

2.2.1 Binärer Datentyp bit

Der Datentyp *bit* belegt im Speicher nur ein einziges Bit. Dieser Datentyp ist in den ANSI-C-Konventionen nicht vorgesehen. Er ist für die 8051-Bausteinfamilie eingeführt worden, da diese Familie einen Booleschen Prozessor und bitadressierbare Speicherplätze enthält, in denen entsprechend deklarierte Variablen platziert werden. Der "Akku" des Booleschen Prozessors ist das Carry-Flag C. Der Datentyp *bit* kann nur die Werte 0 oder 1 annehmen, eine Vorzeichenbehaftung erübrigt sich.

2.2.2 Alphanumerischer Datentyp char

Der Datentyp *char* (Zeichen) belegt 1 Byte Speicherplatz und ist für die Aufnahme eines Zeichens aus dem ASCII-Zeichensatz vorgesehen. 1 Byte kann 256 verschiedene Werte annehmen, da 1 Byte mit 8 Bit 2^8 = 256 ist. Den Datentyp *char* gibt es in zwei Varianten , und zwar mit und ohne Vorzeichen. Als *signed char* oder einfach nur *char* (mit Vorzeichen) kann der Variablen ein Wert im Bereich von -128 bis 127 zugewiesen werden. Wenn man eine Variable mit dem Typbezeichner *char* definiert, geht C51 davon aus, daß man diese Variable mit Vorzeichen benutzen will. Will man den vorzeichenlosen Typ benutzen, muß man die Variable explizit mit dem Typbezeichner *unsigned char* definieren. Der Wertebereich läuft

dann von 0 bis 255. Landläufig wird auch die Bezeichnung *byte* für *unsigned char* verwendet, was jedoch eine entsprechende Datentypendefinition voraussetzt.

2.2.3 Ganzzahltypen int und long

Der Typbezeichner *int* weist einer Variablen den Typ integer zu. Dieser Datentyp repräsentiert ganzzahlige Werte. Der Wertebereich für Variablen vom Typ *int* ist abhängig von der jeweiligen Hardware sowie der Implementierung des Compilers, und wird nicht durch ANSI C definiert. Für 8- und 16-Bit-Rechner wird für eine *int*-Variable üblicherweise 16 Bit (2 Byte) verwendet. Bei 32-Bit-Rechnern belegt eine Ganzahl-Variable 32 Bit (4 Byte). Vom Speicherplatz, der für die Variable zur Verfügung gestellt wird, hängt der Wertebereich ab, der dieser Variablen zugewiesen werden kann. Durch Voranstellen weiterer Schlüsselwörter kann der Wertebereich (wie auch beim Typ *char*) verändert werden.
In C51 ist der Datentyp *int* mit 16 Bit und vorzeichenbehaftet (signed) implementiert, der darstellbare Wertebereich beträgt -32768 bis 32767. Wird der Variablen der Typbezeichner *unsigned int* (landläufig auch *word*) vorangestellt, verschiebt sich der Wertebereich auf die Zahlen von 0 bis 65535.
Der Datentyp *unsigned long int* (kurz: *unsigned long* oder landläufig auch *dword*) belegt 4 Bytes, wodurch sein Wertebereich von 0 bis 4294967295 (4 Giga) läuft. Wird dieser Typ mit dem Vorzeichenattribut versehen, heißt er *signed long int* (kurz: *long*) und umfaßt den Wertebereich von -2147483648 bis 2147483647 (±2 Giga).

2.2.4 Gleitkommatypen float und double

Um noch größere Zahlen benutzen zu können, und gleichzeitig ökonomisch mit dem Speicherplatz umzugehen, existiert in C die Möglichkeit, Variablen als Gleitkommazahlen zu definieren. Gleitkommazahlen unterscheiden sich von den ganzen Zahlen u.a. dadurch, wie sie im Speicher repräsentiert werden. Ganze Zahlen werden als ein Element abgelegt, wobei der Wert an der Speicherstelle dem Wert der Variablen entspricht. Gleitkommazahlen hingegen werden in zwei Elementen abgelegt, wobei der eine Teil *Mantisse*, der andere *Exponent* genannt wird. In dieser sogenannten Exponential-Schreibweise wird z.B. die Zahl 54123 folgendermaßen aussehen: 5,4123E4. Die Zahl 5,4123 in unserem Beispiel ist die Mantisse, die Zahl 4 der Exponent. Diese Schreibweise bedeutet nichts anderes als:

$5{,}4123 * 10^4$

Die Zahl 4 hinter dem E gibt den Exponent zur Basis 10 an, mit der die Mantisse multipliziert werden muß, um die Dezimalzahl zu erhalten. Der Typbezeichner, der bei der Definition dieser Variablen verwendet werden muß, lautet *float*. Variablen vom Typ *float* werden in C51 in 4 Byte nach dem IEEE-754-Format abgelegt (Institute of Electrical and Electronic Engineers, Inc.). Ein bestimmter Teil einer *float*-Variable dient zur Aufnahme der Mantisse, ein weiterer zur Aufnahme des Exponenten. Variablen vom Typ *float* können Werte im Bereich von 1,176E-38 bis 3,40E+38 annehmen. Neben der einfachen Gleitkommazahl gibt es bei 16-Bit- und größeren Rechnern noch den Datentyp *double*, der in 8 Byte abgelegt wird, wobei hier der Wertebereich von 1,7E-308 bis 1,7E+308 reicht und mit doppelter Genauigkeit gerechnet werden kann. Der grundsätzliche Nachteil bei der Verwendung von *float*-Variablen besteht in der zunehmenden Ungenauigkeit des Ergebnisses, wenn zwei Operatoren mit stark unterschiedlichem Exponenten verarbeitet werden.
In C51 ist der Datentyp *double* syntaktisch zwar implementiert, sonst jedoch mit dem Datentypen *float* identisch.

Das Format für *float* entsprechend dem IEEE-754-Standard-Format (32-Bit) mit einer Genauigkeit von 24 Bit (entsprechend der Mantisse) wird nachfolgend beschrieben.

Das Vorzeichen ist das höchstwertigste der 32 Bits, die Mantisse wird durch die 23 niederwertigen Bits repräsentiert.

Die Bytes werden unter C51-V3.20 im Speicher des 8051 im little-endian-Format abgelegt:

Adresse	+0	+1	+2	+3
Inhalt	MMMMMMMM	MMMMMMMM	E MMM MMMM	S EEE EEEE

Ab C51-V5.10 ist die Reihenfolge umgekehrt (big-endian-Format):

Adresse	+0	+1	+2	+3
Inhalt	S EEE EEEE	E MMM MMMM	MMMMMMMM	MMMMMMMM

Die unterschiedliche Speichersyntax ist besonders zu berücksichtigen, wenn z.B. eine *float*-Zahl speicherresident von einem mit C51-V3.20 übersetzten Programm abgelegt worden war und von einem mit C51-V5.10 übersetzten Programm gelesen wird.

In den Darstellungen bedeutet:

S Vorzeichen-Bit, 1=negativ, 0=positiv.
E Exponent (2er Komplement) mit Offset 127 dezimal.
M 23-Bit normalisierte Mantisse. Das höchstwertigste Bit ist gedanklich immer '1'. Es wird daher nicht gespeichert, woraus sich die 24 Bit Genauigkeit ergeben.

Die Umwandlung einer *float*-Zahl in die entsprechende Speicherdarstellung und umgekehrt ist nicht sonderlich schwierig, wie folgendes Beispiel zeigen soll.

Der *float*-Wert -12,5 hat die Hexdarstellung 0xC1480000, was der Binärdarstellung 1100'0001'0100'1000'0000'0000'0000'0000 entspricht.

Aufgeschlüsselt erhalten
- das Vorzeichenbit eine 1, einer negativen Zahl entsprechend.
- der Exponent die Zahl 10000010 binär bzw. 130 dezimal. Abzüglich 127 bleiben noch 3 für den eigentlichen Exponenten.
- die Mantisse den Rest als 100'1000'0000'0000'0000'0000 binär.

Wird vor die Mantisse die gedankliche eine 1 mit Dezimalpunkt gesetzt, ergibt sich letztlich eine 1,100'1000'0000'0000'0000'0000 binär.
Nun wird diese Mantisse an den Exponenten angepaßt, wobei ein positiver Exponent den Dezimalpunkt nach rechts bewegt. Ein negativer Exponent bewegt den Dezimalpunkt nach links, wobei äquvalent zum Exponenten eine Anzahl von Nullen von links angefügt wird.
In unserem Beispiel wird der Dezimalpunkt um 3 Stellen nach rechts bewegt, so daß sich eine 1100,1000'0000'0000'0000'0000 binär ergibt.
Binärziffern, die jetzt links vom Dezimalpunkt stehen, entsprechen positiven Potenzen zur Zahl 2 gemäß ihrer Position. Hier ergibt sich also $1x2^3+1x2^2+0x2^1+0x2^0 = 12$.
Binärziffern, die rechts vom Dezimalpunkt stehen, entsprechen negativen Potenzen zur Zahl 2 ebenfalls gemäß ihrer Position. Hier ergibt sich also $1x2^{-1}+0x2^{-2}+0x2^{-3}....= 0,5$.
Die beiden Teilergebnisse addieren sich zu 12,5.
Da das Vorzeichenbit gesetzt war, wird hier noch das Minuszeichen hinzugefügt.
Somit entsteht aus der Hexdarstellung 0xC1480000 die Zahl -12,5

2.2.5 Special-Function-Typen sfr, sfr16 und sbit

Die 8051 Bausteinfamilie ist mit Registern ausgestattet, die mit speziellen Funktionen versehen sind. Dabei handelt es sich um Steuerregister für Timer, Ports usw. Um den direkten Zugriff auch auf diese Register zu ermöglichen, stellt C51 eine eigenständige Art der Definition zur Verfügung. Dies ist erforderlich, weil diese Definitionen nicht mit denen der Sprache C nach ANSI-Standard vereinbar sind.
Die Anzahl der Register ist innerhalb der 8051-Bausteinfamilie stark unterschiedlich, so daß sich die Zusammenfassung der spezifischen in einer Include-Datei empfiehlt. Üblicherweise sind im Lieferumfang des Compilers solche Deklarationsdateien für einige der Mitglieder der 8051-Bausteinfamilie enthalten. Weitere Definitionsdateien können leicht selbst erstellt werden. Der Compiler selbst hat keine Special-Function-Register implementiert. Wenn solche nicht definiert sind, dann wird bei Benutzung eine entsprechende Fehlermeldung ausgegeben.
Zur Deklaration von Special-Function-Registern wurde das reservierte Wort *sfr* eingeführt. Es hat eine Breite von 8 Bit, ggf. entsprechend einem Wertebereich von 0 bis 255.
Häufig (vor allem bei neueren 8051 Derivaten) sind zwei Special-Function-Register funktionsmäßig zu einem 16-Bit Wert zusammengefaßt. Um einen effizienten Zugriff auch auf diese Special-Function-Register zu gestatten, kann mit der Definition *sfr16* gearbeitet werden. Der Zugriff auf 16-Bit Special-Function-Register ist möglich, wenn im Speicher der höherwertige Teil des SFR Register direkt dem niederwertigen Teil folgt. Hier sei angemerkt, daß die Speichersyntax solcher *sfr16* dem little-endian-Format entspricht, also umgekehrt dem einer 16-Bit-Variablen. Bei Zuweisungen von *sfr16* zu Variablen oder umgekehrt wird die Richtigstellung vom Compiler vorgenommen.
In typischen 8051-Applikationen ist es oftmals notwendig auf einzelne Bits innerhalb der Special-Function-Register zuzugreifen. Diesem Umstand wird durch eine Erweiterung der Deklarationsmöglichkeiten um das reservierte Wort *sbit* Rechnung getragen. Hierbei sei erwähnt, daß nur solche Special-Function-Register bitadressierbar sind, deren Adresse ohne Rest durch 8 teilbar ist.

2.3 Syntax

Bevor sie benutzt werden, müssen alle Variablen vereinbart (deklariert) werden, d.h. ihnen wird ein Typ und ein Name zugewiesen, bei gleichzeitiger Vergabe eines Speicherplatzes spricht man von einer Definition der Variablen.
<Typbezeichner> <Variablenname1>[,<Variablenname2>,...];

Bei der Vereinbarung wird zuerst der Datentyp durch Verwendung des entsprechenden Typbezeichners gemäß den Regeln in 1.2.5 angegeben. Durch mindestens ein Leerzeichen getrennt folgt dann der Name der Variablen.

```
int nMonat; char cZeichen;
```

Möchte man mehrere Variablen gleichen Typs definieren, kann man dies auch in einer einzigen Programmzeile tun.

```
int nTag, nMonat, nJahr;
```

In diesem Fall werden die Variablennamen durch Kommata getrennt. Da es sich bei der Definition von Variablen um Anweisungen handelt, müssen die Vereinbarungen mit einem Semikolon abgeschlossen werden. Es ist möglich, bei der Definition von einfachen Variablen eine Initialisierung (gleichzeitige Wertzuweisung) vorzunehmen:

```
int nMonat = 3;
```

Hinter dem Variablennamen steht dann ein Gleichheitszeichen. Der Wert auf der rechten Seite wird der Variablen auf der linken zugewiesen.

Allerdings weicht bei gleicher Syntax der Special-Function-Variablen die Bedeutung von dieser Definition ab.
Es ist zu beachten, daß die Angabe hinter dem Namen keine Zuweisung, sondern eine Deklaration oder noch eher eine Platzierung darstellt. Damit werden im folgenden Beispiel die Namen `P0` und `P1` (Port0, Port1) als Special-Function-Register deklariert und mit der jeweiligen absoluten Adresse versehen. Die Namen sind frei wählbar, innerhalb des Compilers sind keinerlei SFR-Namen vordefiniert oder bekannt.

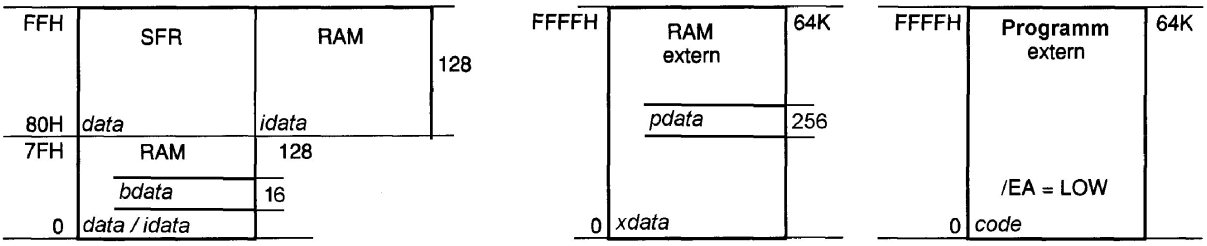
```
sfr P0 = 0x80; /* Port0, Adresse 80h */
sfr P1 = 0x90; /* Port1, Adresse 90h */
sfr16 T2 = 0xCC; /* Timer 2: T2L 0CCh, T2H 0CDh */
sfr PSW = 0xD0; /* ProzessorStatusWort 0D0h (bitadressierbar) */
```

Es ist also nicht so, daß `P0` (Port 0) der Wert 80H zugewiesen wird, sondern mit `P0` wird eine Bezeichnung für den Port 0 deklariert, der an der Adresse 80h liegt.
Die Adreßangabe nach dem Zeichen '=' muß eine Konstante sein, Ausdrücke mit Operatoren sind nicht zulässig. Die Konstante muß innerhalb des Bereichs >= 0x80 und <= 0xFF sein.
Eine gleichzeitige Wertzuweisung ist in diesem Fall nicht möglich.
Bei dem Typen *sbit* sind verschiedene Deklarationsschreibweisen möglich:

```
sbit OV = PSW^2; /* Overflowflag ist das Bit2 im PSW */
```
oder
```
sbit OV = 0xD0^2;
```
oder
```
sbit OV = 0xD2;
```

2.4 Speichertypen

Bedingt durch die Harvard-Architektur der 8051-Controller und die interne Speicheraufteilung gibt es fünf Adressierungsmöglichkeiten.



Entsprechend der im Bild gezeigten Speicherbereiche kann man Variablen jeweils dort platzieren. Damit der Compiler jedoch weiß, in welchem Speicherbereich eine Variable zu liegen hat, muß man entweder den Speichertyp angeben, oder die Variable wird platziert, wie es durch das Speichermodell vorgegeben ist.

Der direkt adressierbare interne Datenspeicher *data* mit einer Größe von 128 Byte ermöglicht schnellste Zugriffe auf Variablen mit dem Maschinenbefehl *MOV* über den Akku. Wird *MOV* auf die oberen 128 Byte angewendet, dann erfolgt ein Zugriff auf die Special-Function-Register. Der bitadressierbare interne Datenspeicher *bdata* (16 Bytes ab 20h) ermöglicht gemischten Bit- und Byte-Zugriff ebenfalls über *MOV*, wobei bei Bit-Zugriffen als Zwischenregister nicht der Akku A sondern das Carry-Flag C verwendet wird.

Der indirekt adressierbare interne Datenspeicher *idata* ermöglicht den Zugriff auf den vollen internen Adreßbereich von 256 Byte über den Befehl *MOV @Ri* wobei Ri für die Register R0 und R1 steht.

Auf die 64kByte externen Datenspeicher *xdata* kann nur indirekt mit dem Befehl *MOVX @DPTR* zugegriffen werden. Dieser Zugriff ist recht langsam, da zunächst immer der 16-Bit-DPTR geladen werden muß.

Ein alternativer Zugriff zum DPTR-Zugriff auf *xdata* bietet *pdata* (paged data). Dies ist ein 256-Byte-Segment des xdata-Bereichs, auf den mit dem Befehl *MOVX @Ri* zugegriffen werden kann. Der höherwertige Adressteil, der die Seite auswählt, wird durch das Beschreiben des Registers P2 bestimmt. Allerdings sei hier angemerkt, daß nicht alle 8051-Derivate den pdata-Zugriff richtig beherrschen. So können beispielsweise die 80C251-Controller von INTEL und TEMIC entgegen dem Datenblatt mit dem Beschreiben von P2 nicht viel anfangen und greifen immer auf die Seite 0 (also die ersten 256 Bytes des xdata-Bereichs) zu. Man sollte das verwendete Derivat entsprechend testen.

Als letzter Speichertyp sei der 64 KByte große Programm-Speicher *code* genannt auf den mit dem Befehl *MOVC @A+DPTR* zugegriffen werden kann. Diese Adressierungsart ist die langsamste aller Zugriffsarten weil nicht nur der DPTR, sondern auch der Akku initialisiert werden muß. Diese Zugriffsart eignet sich am ehesten für Tabellenzugriffe, bei denen im DPTR die Basisadresse der Tabelle und im Akku der Laufindex steht.

2.5 Speichermodell

Das Speichermodell bestimmt den 'default'-Speichertyp, der verwendet wird:
- für Variablen-Deklarationen ohne expliziten Speichertyp,
- für Parameter-Variablen, die nicht mehr in die Register passen.

Der C51-Compiler hat drei Speichermodelle implementiert:
- SMALL-Modell: Die Variablen werden standardmäßig im *data*-Bereich platziert.
- COMPACT-Modell: Die Variablen werden standardmäßig im *pdata*-Bereich platziert.
- LARGE-Modell: Die Variablen werden standardmäßig im *xdata*-Bereich platziert

Da die Rechenleistung der 8051-Derivate seine Grenzen hat, ist meines Erachtens immer das SMALL-Modell sinnvoll, weil hier die Zugriffe auf die Variablen am schnellsten sind. Ein weiterer Grund hierfür wird sich später bei der Overlaytechnik zeigen. Allerdings sollte man sich darüber bewußt sein, daß beim Anlegen von statischen Variablen im *data*-Bereich dieser schnell überlaufen kann.
Bei den anderen Speichermodellen wird immer über eines der Register R0 oder R1 bzw. über den Datenpointer zugegriffen.
Ist eine andere Platzierung als die Standardplatzierung gewünscht, dann kann der Speichertyp explizit angegeben werden.
Eine Variable, die beispielsweise im *idata*-Bereich zu liegen kommen soll, wird so deklariert:

```
int idata nMonat;
```
bzw. `idata int nMonat;` zur Kompatibilität früherer Compilerversionen.

Ein Zeiger, der auf eine Variable zeigt, die explizit im *xdata*-Bereich liegt, sieht z.B. so aus:

```
int xdata *pnMonat;
```

Die Lage des Zeigers selbst ist hierbei wiederum durch das Speichermodell vorgegeben.
Möchte man zusätzlich die Lage des Zeigers festlegen (z.B. in den *data*-Bereich), dann sieht eine Definition so aus:

```
int xdata * data pnMonat;
```
bzw. `data int xdata *pnMonat;` zur Kompatibilität früherer Compilerversionen.

Möchte man Tabellen im Code ablegen (hier beispielsweise die Anfangsbuchstaben der Wochentage), so nimmt man folgende Definition vor:

```
code char acTage[] = {´-´,´M´,´D´,´M´,´D´,´F´,´S´,´S´};
```

In diesem Beispiel sind die Tage 1 bis 7 angenommen, der Tag 0 wird den ´-´ ausweisen. Somit entspricht die Position des Wochentags unserer gewohnten Denkweise.

2.6 Sichtbarkeit und Lebensdauer

C erkennt Variablen nur während bestimmter Abschnitte eines Programms. Wo der Compiler die Variablen erkennen kann, hängt zum einen mit der Speicherklasse der Variablen zusammen, wie sie bei der Definition angegeben wird. Desweiteren ist der Ort im Quellcode, an der die Variable definiert wird, von entscheidender Bedeutung.

2.6.1 Speicherklassen

Grundsätzlich kennt C zwei Arten von Speicherklassen. Die Speicherklasse bestimmt die Lebensdauer der Variablen.

- Variablen mit lokaler Lebensdauer werden innerhalb einer Funktion oder eines Blocks (geschweifte Klammern {}) definiert, und existieren nur solange, wie der Block abgearbeitet wird. Der Compiler erzeugt diese Variablen dynamisch. Nach der Beendigung des Blocks oder der Funktion wird der Speicherplatz der lokalen Variablen freigegeben. Damit ist auch ihr Wert als verloren zu betrachten.
- Variablen mit globaler oder statischer Lebensdauer existieren von Anbeginn des Programms bis zu seinem Ende (oder Untergang). Der Wert dieser Variablen kann somit während des Programms verändert und auch wieder abgefragt und verwendet werden.

2.6.2 Interne und externe Deklaration

Die Wirkung der Angabe der Speicherklasse hängt auch davon ab, wo die Variable definiert ist. Hier wird zwischen der internen und der externen Definition bzw. Deklaration unterschieden.

- Eine interne Definition bzw. Deklaration steht innerhalb einer Funktion oder innerhalb eines Anweisungsblocks. Mit der Definition wird eine neue Variable angelegt, durch die Deklaration ein Verweis auf eine Variable erzeugt, die an einer anderen Stelle definiert ist.
- Eine externe Deklaration befindet sich außerhalb aller Funktionen an einer beliebigen Stelle der Quelldatei.

2.6.3 Speicherklassenangaben

Bei der Definition oder Deklaration einer Variablen kann die Typangabe durch die Angabe einer Speicherklasse ergänzt werden. Die Speicherklasse bestimmt in Wechselwirkung mit dem Ort der Definition oder Deklaration die Sichtbarkeit und die Lebensdauer der Variablen.
Ohne Speicherklassenangabe Variablen, die ohne Angabe einer Speicherklasse definiert werden, müssen nach dem Ort der Definition unterschieden werden.

- Steht die Definition innerhalb einer Funktion, hat die Variable lokale Lebensdauer und ist nur innerhalb des Blocks sichtbar, in der sie definiert ist. Dies entspricht der Speicherklassenangabe *auto*.
- Steht die Definition auf der externen Ebene, also außerhalb einer Funktion, hat die Variable globale Lebensdauer. Außerdem ist sie während eine Compilerlaufs allen Funktionen bekannt, die nach der Variablen-Definition definiert werden. Durch Verwendung des Schlüsselwortes *extern* bei der Deklaration kann die Variable außerdem anderen Dateien oder Funktionen in anderen Dateien bekannt gemacht werden.

2.6.4 Speicherklasse auto

Das Schlüsselwort *auto* weist einer Variablen explizit die automatische Speicherklasse zu. Diese Speicherklassenangabe kann nur auf der internen Ebene, d.h. innerhalb von Funktionen und Blöcken verwendet werden. Lokale Variablen innerhalb einer Funktion haben automatisch die automatische Speicherklasse (daher der Name). Der Speicherplatz für automatische Variablen wird innerhalb des Blocks, in dem sie deklariert werden, erzeugt, und beim Verlassen des Blocks wieder freigegeben. Wenn aus dieser Funktion heraus weitere Funktionen aufgerufen werden, bleibt der Wert der Variablen zwar erhalten, es kann jedoch auf sie nicht innerhalb der aufgerufenen Funktionen zugegriffen werden. Variablen der Speicherklasse *auto* werden nicht automatisch initialisiert.

2.6.5 Speicherklasse static

Variablen, die die Speicherklasse *static* besitzen, bleiben während des gesamten Programmablaufs erhalten. Ist ihre Definition mit einer Initialisierung verknüpft, wird ihr zu Beginn des Programms (noch vor dem Aufruf von *main()*) der gewünschte Wert zugewiesen. Findet keine Initialisierung statt, erhält sie den Null-Wert des entsprechenden Datentyps. Statische Variablen können innerhalb eines Blocks oder außerhalb einer Funktion definiert werden. Dies beeinflußt die Sichtbarkeit der Variablen. Wurde sie innerhalb eines Blocks definiert, ist sie nur innerhalb des Blocks bekannt. Wird die Variable außerhalb einer Funktion definiert, ist sie ab der Stelle ihrer Definition bis zum Ende der Datei sichtbar. Statische Variablen werden meist dann verwendet, wenn ihre Sichtbarkeit auf eine Datei oder eine Funktion beschränkt ist, jedoch der Wert der Variablen zwischen Funktionsaufrufen nicht verändert werden soll.

2.6.6 Speicherklasse extern

Die Speicherklasse *extern* kann auf der internen Ebene (innerhalb von Funktionen) und auf der externen Ebene (außerhalb von Funktionen) benutzt werden. Steht bei einem Variablennamen das Schlüsselwort *extern*, handelt es sich immer um eine Deklaration, nie um eine Definition. Mit *extern* wird ein Verweis auf eine Variable eingeleitet, die an einer anderen Stelle definiert worden ist. Die Sichtbarkeit in der anderen Quelldatei ist davon abhängig, wo die Deklaration stattfindet:

- Findet die Deklaration auf der externen Ebene statt, ist die Variable allen Funktionen, die nach dieser Deklaration definiert werden, bekannt.
- Findet die Deklaration auf der internen Ebene statt, dann ist die externe Variable nur innerhalb des Blocks bekannt.

Die Definition der Variablen, auf die mit *extern* Bezug genommen wird, muß ebenfalls auf der externen Ebene erfolgen.

2.6.7 Schlüsselwort register

Mit dem Schlüsselwort *register* wird festgelegt, daß auf die so definierte Variable so schnell wie möglich zugegriffen werden soll. Diese Speicherklassenangabe kann nur auf der internen Ebene, d.h. innerhalb von Funktionen verwendet werden oder bei der Angabe des Typs eines Funktionsarguments. Für den Rechner bedeutet das, daß die mit diesem Bezeichner deklarierten Variablen in die Register des Prozessors geladen werden sollen. Der Compiler braucht dies jedoch nicht zu tun. Die Verwendung von *register* ist also eher eine Bitte des Programmierers an den Compiler, in der Hoffnung, so effizienteren Code zu erzeugen. *register* wird häufig in Zusammenhang mit Schleifenvariablen eingesetzt. Die Adresse eines Objekts, die mit *register* deklariert worden ist, kann nicht mehr ermittelt werden, gleichgültig, ob dieses Objekt sich in einem Prozessorregister befindet oder nicht. Die Anzahl der verfügbaren Register ist maschinenabhängig. In C51 ist *register* zwar syntaktisch implementiert, hat semantisch jedoch keine Bedeutung, weil nach Möglichkeit sowieso alle lokalen Variablen in den Registern abgelegt werden.

2.7 Aufzählungstypen

Aufzählungstypen sind Datentypen, deren mögliche Werte durch eine Reihe von Konstanten festgelegt sind. Die Syntax für Aufzählungstypen an einem Beispiel:

```
enum FARBE {SCHWARZ, BLAU=2, ROT=5, GELB};
```

Einer Variablen dieses Typs kann nun der Wert eines der Mitglieder zugewiesen werden. Die Wertzuweisung ist möglich, da der Compiler jedem Element einen Wert zuweist. Er beginnt beim ersten Element, weist ihm den Wert 0 zu und inkrementiert den Wert für die nachfolgenden Elemente. Dem Element `SCHWARZ` wird demnach der Wert 0 zugewiesen. Den Namen der Elemente `BLAU` und `ROT` folgt das Gleichheitszeichen, wodurch diesen Elementen jeweils der Wert der rechten Seite zugewiesen wird. `BLAU` hat also den Wert 2 und `ROT` den Wert 5. Für das Element `GELB` gelten wieder die alten Regeln. Der Wert des vorigen Elements wird inkrementiert, und `GELB` erhält den Wert 6. Nachdem der Aufzählungstyp deklariert ist, kann eine Variable dieses Typs definiert und ihr der Wert eines der Elemente zugewiesen werden.

```
enum FARBE nHintergrund = SCHWARZ;
```

Nachdem ihr ein Wert zugewiesen wurde, kann diese Variable überall dort benutzt werden, wo eine Variable vom Typ *int* verwendet werden kann. Achtet man darauf, daß die Werte der Elemente den Bereich eines anderen Datentyps (z.B. *byte*) nicht überschreitet, dann kann ein Element eines Aufzählungstypen auch einer Variablen entsprechend anderen Typs zugewiesen werden.

Das folgende Beispiel verwendet den Aufzählungstyp, um Konstanten zu definieren, die Fehlernummern symbolisieren. Diese Konstanten können dann an die Funktion `Fehlermeldung()` als Parameter vom Typ *int* übergeben werden.

```
enum FEHLER_NUMMER
    {
    ERROR_NO_MEM = 100,     /* Speichermangel */
    ERROR_LESEN,            /* Lesefehler */
    ERROR_SCHREIBEN         /* Schreibfehler */
    };

void Fehlermeldung(byte byFehlernummer)
{
printf("Der Fehler Nr. %u ist aufgetreten.\n", (word)byFehlernummer);
}

main()
{
Fehlermeldung(ERROR_NO_MEM);
}
```

Die Verwendung des enumerativen Datentyps kann anstelle der Definition von symbolischen Konstanten mit der Präprozessordirektive #define erfolgen. Die Konstanten hätten auch folgendermaßen definiert werden können:

```
#define ERROR_NO_MEM     100
#define ERROR_LESEN      101
#define ERROR_SCHREIBEN  102
```

Der Vorteil bei der Verwendung der Aufzählungstypen besteht darin, daß zusammengehörende Konstanten besser als Gruppe deutlich gemacht werden können.

2.8 Strukturen

Strukturen fassen mehrere Daten, die einen unterschiedlichen Typ besitzen können, zu einer Variablen zusammen. Wenn man beispielsweise die Adresse einer Person bearbeiten will, kann man folgende Variablen definieren, die die einzelnen Daten aufnehmen sollen.

```
char acName[20];
long lPlz;
char acOrt[20];
char acStrasse[20];
```

Wenn jedoch mehrere Personen in dieser Kartei vorkommen sollen, ist es sinnvoll, diese zusammengehörenden Daten auch zusammenzufassen. Damit kann sowohl die Definition der Variablen verkürzt, als auch der Zugriff auf die Daten vereinfacht werden. Für das obige Beispiel sieht die Deklaration der Struktur folgendermaßen aus.

```
struct typtPerson
    {
    char acName[20];
    long lPlz;
    char acOrt[20];
    char acStrasse[20];
    };
```

2.8.1 Deklaration

Man beachte, daß mit der Deklaration kein Speicherplatz bereitgestellt wird, sondern lediglich der Compiler über diesen Aggregat-Typ informiert wird. Speicherplatz wird erst dann bereitgestellt, wenn eine Variable dieses Strukturmusters definiert wird.
Die Deklaration beginnt mit dem Schlüsselwort *struct*, dem ein optionaler Bezeichner (als Name für das Strukturmuster) folgen kann. Wenn der Name des Strukturmusters nicht angegeben wird, muß die Definition aller Variablen dieses Strukturmusters bei seiner Deklaration erfolgen.
In geschweifte Klammern eingeschlossen folgt eine Liste der Strukturelemente Die Beschreibung der einzelnen Strukturelemente besitzt das gleiche Format wie die Definition einer Variablen. Strukturelemente können einfache Variablen, Arrays, andere Strukturen oder Zeiger sein. Der Datentyp *bit* (C51) kann nicht Element einer Struktur sein, weil für Bits der bitadressierbare Speicherbereich vorgesehen ist wird. Es ist nicht möglich, die Struktur, die deklariert wird, zu einem Element der Struktur selbst zu machen. Wird bei der Deklaration der Name eines Strukturmusters benutzt, kann jedoch ein (oder auch mehrere) Element(e) als Zeiger auf dieses Strukturmuster deklariert werden.

2.8.2 Definition

Für die Definition einer Strukturvariablen gelten die gleichen Regeln wie für die Definition von Variablen der Basistypen. Außerdem kann die Definition einer Strukturvariablen um die Speicherklassenangaben erweitert werden. Mit der Definition wird der Speicherplatz für die Struktur bereitgestellt. Die Elemente einer Struktur werden in der gleichen Reihenfolge im Speicher abgelegt, wie sie deklariert sind. Beispiele:

```
struct typtPerson tKunde;         /* definiert eine Variable vom Typ
                                  struct typtPerson.*/
static struct typtPerson tKunde; /* definiert eine statische Variable vom
                                  Typ struct typtPerson. */
struct typtPerson atKunde[40];    /* definiert ein Array mit 40 Elemente vom
                                  Typ struct typtPerson.*/
struct typtPerson *ptKunde;       /* definiert einen Zeiger auf eine Struk-
                                  turvariable vom Typ struct typtPerson */
```

2.8.3 Initialisierung

Auch Strukturvariablen können bei ihrer Definition initialisiert werden. Nach der Definition der Variablen folgt das Gleichheitszeichen. In geschweifte Klammern eingeschlossen, werden die einzelnen Elemente, die initialisiert werden sollen, durch Kommata getrennt aufgelistet.

```
struct typtPerson
    {
    char acName[20];
    long lPlz;
    char acOrt[20];
    char acStrasse[20];
    };

struct typtPerson tKunde =
    {"Klaus Müller", 76152, "Entenhausen", "Duckstr."};
```

Auch wenn es in diesem Beispiel so gemacht ist: Es ist nicht erforderlich, alle Elemente der Struktur zu initialisieren, die letzten Elemente können entfallen.

2.8.4 Zugriff auf einzelne Strukturelemente

Es gibt zwei Möglichkeiten, auf die Elemente einer Struktur zuzugreifen (Kap. 3.3.3 und 3.3.4). Handelt es sich bei der Variablen um eine Struktur, dann wird der Punkt-Operator "**.**". verwendet, bei einem Strukturzeiger wird der Strukturverweisoperator "->" benutzt. Im folgenden Beispiel wird eine Variable vom Typ `struct typtPerson` definiert, die den Namen `tKunde` hat und einen Zeiger auf diesen Strukturtyp mit dem Namen `ptKunde` .
Durch eine Anweisung erhält `ptKunde` die Adresse der Variablen `tKunde`.

```
#include <stdio.h>

struct typtPerson
    {
    char acName[20];
    long lPlz;
    char acOrt[20];
    };

main()
{
struct typtPerson tKunde;
struct typtPerson *ptKunde;

ptKunde = &tKunde;
tKunde.lPlz = 76135;
strcpy(tKunde.acName, "Otter");
strcpy(ptKunde->acOrt, "Karlsruhe");
printf("Kunde: %s, PLZ: %ld Ort: %s",
    tKunde.acName, tKunde.lPlz, tKunde.acOrt);
printf("Kunde: %s, PLZ: %ld Ort: %s",
    ptKunde->acName, ptKunde->lPlz, ptKunde->acOrt);
}
```

Die folgenden Ausdrücke haben die gleiche Bedeutung:

```
strcpy(ptKunde->acOrt, "Karlsruhe");
strcpy((*ptKunde).acOrt, "Karlsruhe");
```

2.9 Unions

Eine Union ist ein Datentyp, dessen Elemente überlagert sind, d.h. den gleichen Speicherplatz beanspruchen. Unions werden meist dann eingesetzt, wenn unter bestimmten Umständen immer nur ein Element benutzt wird. Darin besteht auch der Unterschied zu einer Struktur, bei der für jedes Element Speicherplatz bereitgestellt wird. Die Größe einer Union entspricht der Größe des größten Elements. Die Deklaration einer Union hat Ähnlichkeit mit der einer Struktur. Nur wird hier die Deklaration durch das Schlüsselwort *union* eingeleitet.

```
union uMuster
    {
    int nZahl;
    float fKommaZahl;
    char *pcZeichen;
    };
```

Die Definition einer Union-Variablen und der Zugriff auf einzelne Elemente folgt den gleichen Prinzipien, wie sie bei Strukturen erläutert wurden. Wenn bei der Definition die Variable initialisiert werden soll, darf jedoch immer nur das erste Element initialisiert werden.

2.10 Bitfelder

Bitfelder stellen eine Sonderform der Strukturen dar. Bitfelder werden benutzt, wenn Daten gespeichert werden sollen, die weniger als 1 Byte Speicherplatz benötigen. Hierzu gehören zum Beispiel 1 Bit große Flags, die ausreichen, um Wahrheitswerte darzustellen. Bis auf die Beschreibung der Strukturelemente ist die Deklaration eines Bitfeldes mit der einer Struktur identisch. Ein einzelnes Bitfeld muß nach ANSI C immer den Datentyp int, signed int oder unsigned int besitzen. Nach der Typangabe und dem Namen des Elements folgt ein Doppelpunkt. Daran schließt sich ein ganzzahliger Wert an, der bestimmt, wieviel Bits diesem Element zur Verfügung gestellt werden sollen:

```
struct tbStatus
    {
    unsigned bSchatten  :1;
    unsigned bRahmenEin :1;
    unsigned bPalette   :1;
    unsigned bHzeileEin :1;
    unsigned bFlAktiv   :1;
    }
```

Fast alles bei Bitfeldern ist compilerabhängig. Ob ein Bitfeld eine Wortgrenze überschreiten kann, hängt vom Compiler ab. Bitfelder müssen nicht unbedingt benannt werden; für Zwischenräume können unbenannte Bitfelder verwendet werden, die dann nur aus Doppelpunkt und Angabe zur Breite bestehen. Mit der besonderen Breite 0 kann man die Ausrichtung auf die nächste Wortgrenze verlangen.
Bitfelder werden bei manchen Maschinen von links nach rechts und bei anderen von rechts nach links angeordnet. Andere wiederum ordnen die Bitfelder in der Reihenfolge 0,2,1,3 an. Bitfelder sind deshalb zwar nützlich, um intern definierte Datenstrukturen zu repräsentieren; bildet man jedoch extern definierte Objekte ab, so muß man sorgfältig überlegen, in welcher Reihenfolge die Bitfelder angelegt werden. Programme, die von solchen Dingen abhängen, sind nicht portabel. Bitfelder dürfen auch nur als int vereinbart werden; zur Portabilität sollte jedoch explizit unsigned angegeben werden. Es gibt keine Arrays von Bitfeldern und auch keine Zeiger. Bitfelder haben keine Adressen, so daß der Adreß-Operator & auf sie nicht angewendet werden kann.
C51 nutzt bei Bitfeldern weder den bdata-Bereich noch den Booleschen Prozessor aus, auch dann nicht, wenn das Bitfeld im bdata-Bereich deklariert ist.

2.11 Arrays

Ein Array, auch Datenfeld oder auch einfach Feld genannt, ist die Zusammenfassung von mehreren Daten des gleichen Typs zu einer Variablen. Neben den einfachen Typen können Arrays auch aus zusammengesetzten Typen bestehen.

2.11.1 Definition eines Arrays

Die Definition eines eindimensionalen Arrays hat folgende allgemeine Form:

Typ Name[Größe];

Die Typangabe legt fest, aus welchen Elementen das Array gebildet wird, und liefert zusammen mit der Dimensionierung Informationen über den Speicherplatzbedarf. Für die Namensgebung eines Arrays gelten die in 2.3 beschriebenen Regeln.
Die Größe legt die Anzahl der Elemente fest, aus denen das Array besteht. Beispiele:

```
int anMonatsUmsatz[12];
char acBuchstaben[256];
```

2.11.2 Initialisierung

Es ist möglich, bei der Definition eines Arrays Initialisierungen vorzunehmen. Voraussetzung dafür ist jedoch, daß es sich um ein Array der Speicherklasse *static* handelt, oder das Array auf der externen Ebene definiert wird. Dem Namen des Arrays folgt dann das Gleichheitszeichen. Die einzelnen Werte, die zugewiesen werden sollen, werden in geschweifte Klammern eingeschlossen und durch Kommata getrennt.

```
static float afWerte[3] = { 2.34, 5.76, 4.62 };
```

Die Werte in den einzelnen Elementen sind:
afWerte[0]: 2.34
afWerte[1]: 5.76
afWerte[2]: 4.62

Es ist nicht erforderlich, alle Elemente zu initialisieren. Elemente, die nicht initialisiert worden sind, erhalten automatisch den Wert 0.

```
static int anZahlen[3] = { 2, 5 };
```

Die Werte in den einzelnen Elementen sind:

anZahlen[0]: 2
anZahlen[1]: 5
anZahlen[2]: 0

Wenn bei einem initialisierten Array die Größe nicht angegeben wird, ermittelt der Compiler die Größe des Arrays anhand der Anzahl der initialisierten Elemente.

```
static int anZahlen[] = { 2, 5, 7, 9 };
```

Damit erhält das Array die Größe von 4 Elementen.

2.11.3 Zugriff auf ein Array

Auf die einzelnen Elemente des Arrays wird über einen Index zugegriffen, der in eckigen Klammern steht. Das erste Element eines Arrays hat immer den Index 0. Das letzte Element hat den Index, der sich aus der Formel Größe -1 berechnet. Der Zugriff auf ein Array ist mit größter Vorsicht zu handhaben, weil C grundsätzlich keine Möglichkeit bietet, den Zugriff auf seine Gültigkeit zu prüfen. Der Programmierer muß selbst darauf achten, daß der Zugriff nicht außerhalb des definierten Bereichs stattfindet. Der unerlaubte Schreibzugriff (speziell bei Laufindizes und als Parameter übergebene Indizes) ist die häufigste Ursache für einen schwer ermittelbaren Absturz eines Programms oder des gesamten Rechners. Daher sollte in Schleifen der Laufindex über eine mit #define festgelegte Größe oder mit Hilfe des sizeof-Operators begrenzt bzw. überprüft werden (s. Kap. 8.2.6).

```
#define ARRAY_GROESSE 20
int anArray[ARRAY_GROESSE];
byte byI;
for (byI=0; byI<ARRAY_GROESSE; byI++)
    anArray[byI] = byI;
```

Alternativ hierzu kann durch Verwendung des in Kap. 8.2.6 gezeigten Makros das #define entfallen:

```
int anArray[20];
byte byI;
for (byI=0; byI<ELEMENTE(anArray,int); byI++)
    anArray[byI] = byI;
```

2.11.4 Zeichen-Arrays und Strings

Zeichenketten oder Strings stellen eine Sonderform der Arrays dar. Sie können vom Typ *char*, *signed char* oder *unsigned char* sein. ANSI C verwendet die sogenannten ASCIIZ-Strings, also Zeichenketten, die durch das Zeichen ASCII 0 (Zero) abgeschlossen werden. Bei der Initialisierung von Zeichenketten-Konstanten wird die abschließende '\0' automatisch durch den Compiler angehängt. Die Initialisierung eines String-Arrays kann verschiedene Formen annehmen:

```
static char acVar1[] = ( 'e', 'i', 'n', 's' };
static char acVar2[] = ("zwei"};
```

Im ersten Beispiel werden die einzelnen Elemente des Arrays `acVar1` einzeln initialisiert. Da der Compiler nicht erkennen kann, daß es sich um eine Zeichenkette handelt, besteht das Array aus vier Elementen. Beim zweiten Beispiel wird das Array mit einer Zeichenketten-Konstanten initialisiert. Der Compiler erkennt, daß es sich um eine Zeichenkette handelt und ergänzt das abschließende Null-Zeichen. `acVar2` besteht aus fünf Elementen.

2.11.5 Mehrdimensionale Arrays

Es ist auch möglich, zwei- oder mehrdimensionale Arrays zu verwenden. Die Definition enthält so viele Paare eckiger Klammern, wie das Array Dimensionen besitzt.

```
int anMatrix[3][2];
```

Mit dieser Anweisung wird ein zweidimensionales Array aus sechs Elementen definiert. Auch mehrdimensionale Arrays werden nacheinander im Speicher abgelegt. Das Prinzip funktioniert so, daß der letzte Index sich am schnellsten verändert. Die Reihenfolge des oben definierten Arrays im Speicher:

anMatrix[0] [0]
anMatrix[0] [1]
anMatrix[1] [0]
anMatrix[1] [1]
anMatrix[2] [0]
anMatrix[2] [1]

Bei mehrdimensionalen Arrays kann die erste eckige Klammer in folgenden Fällen leer bleiben:

- Bei Definitionen eines Arrays auf der externen Ebene, bei der gleichzeitig eine Initialisierung stattfindet.
- Bei der Definition eines Arrays, das die Speicherklasse *static* besitzt, und bei der die Definition initialisiert wird.
- Bei der Deklaration eines Arrays mit der Speicherklassenangabe *extern*.
- Bei der Deklaration einer Funktion, die ein Array als Parameter erhält.

Mehrdimensionale Arrays können auf zwei Arten initialisiert werden. Entweder wird für jede Komponente eine eigene Initialisierungsliste benutzt, die in geschweifte Klammern eingeschlossen wird

```
int anMatrix[3][2] =
    {
        { 0, 0 },
        { 1, 1 },
        { 2, 2 },
    };
```

oder jeder Komponente des untergeordneten Aggregattyps die Werte einfach der Reihe nach zugewiesen.

```
int anMatrix[3][2] = ( 0, 0, 1, 1, 2, 2 };
```

Beide Schreibweisen sind von der Wirkung her identisch. Die erste hat den Vorteil, daß sie den Charakter des zweidimensionalen Arrays besser hervorhebt, auch wenn sie etwas mehr Platz für die Schreibweise benötigt.

Es sei angemerkt, daß C garantiert, daß die Elemente des Arrays fortlaufend im Speicher abgelegt werden. Der Compiler überprüft nicht, ob mit einem Index die Bereichsgrenzen des Arrays überschritten werden, wohl aber bemerkt er, wenn der Versuch unternommen wird, mehr Elemente bei der Definition zu initialisieren als deklariert sind. Bei der Übergabe eines Arrays an eine Funktion wird die Adresse des ersten Elements übergeben.

2.12 Zeiger

Ein Zeiger ist eine Variable, in der sich die Adresse eines Datenobjekts oder einer Funktion befindet. Ein Zeiger kann auf jedes beliebige Datenobjekt zeigen, bis auf wenige Ausnahmen: Eine Variable des Datentyps *bit*, eine Variable, die die Speicherklasse *register* besitzt, und auf ein Bitfeld.

2.12.1 Definition einer Zeigervariablen

Die Definition einer Zeigervariablen hat die Form:

```
int *pnZeigerAufInt;
struct person *ptKunde;
```

Zwischen der Typangabe und dem Namen der Variablen wird ein Stern eingefügt. Daran erkennt der Compiler, daß es sich um einen Zeiger handelt. Man beachte,

- daß im Kontext einer Variablendefinition der Stern nicht mit dem Indirektionsoperator oder dem Zeichen für Multiplikation verwechselt werden darf;
- daß der Compiler bei der Definition von Zeigervariablen nur den Speicherplatz für die Adresse, nicht den Platz für das Datenobjekt selbst bereitstellt;
- daß der Zeiger den gleichen Typ besitzen muß wie das Datenobiekt auf den er zeigt.

2.12.2 Initialisierung

Die Initialisierung eines Zeigers erfolgt durch eine Wertzuweisung. Das folgende Beispiel definiert die Variable `nVar` vom Typ *int* und die Variable `pnVar` als Zeiger auf *int*. Gleichzeitig wird `pnVar` die Adresse von `nVar` zugewiesen.

```
int nVar;
int *pnVar = &nVar;
```

Die Zuweisung erfolgt, nachdem mit dem Adreßoperator die Adresse der Variablen `nVar` ermittelt worden ist.

Bei der Zuweisung der Adresse eines Arrays übersetzt der Compiler den Arraynamen in die Adresse des ersten Elements. Die folgenden beiden Anweisungen können alternativ verwendet werden.

```
int anArray[10];
int *pnArray = anAarray; /* entspricht int *pnArray = &anArray[0]; */
```

Eine weitere mögliche Konstruktion ist ein Array, dessen Elemente Zeiger sind. Folgendes Beispiel initialisiert ein Array mit Zeigern auf *char*. Jeder Zeiger zeigt auf eine andere Zeichenkette.

```
static char *aszMonate[] =
    {
    NULL, "Januar", "Februar", "März", "April", "Mai", "Juni",
    "Juli", "August", "September", "Oktober", "November", "Dezember",
    };
```

Das Element `aszMonate[4]` zeigt beispielsweise auf den Text "April". Man beachte, daß das erste Element des Arrays als NULL-Zeiger initialisiert wurde. Somit entspricht die Position des Monatsnamens unserer gewohnten Denkweise.

2.12.3 Zeiger in C51

Die vorgenannte Beschreibung der Zeiger entspricht dem ANSI-Standard. Auch C51 genügt dem ANSI-Standard mit dem *generic pointer*. Dieser Zeiger ist so aufgebaut, daß er alle Speicherbereiche adressieren kann (vgl. hierzu Kap. 2.4). Er ist 3 Bytes groß, wobei im ersten Byte der Speicherbereich codiert ist, in den der Zeiger zeigt:

Speicherbereich /-typ	Code beim Compiler V3.20	Code ab C51 V5.10	Bytes beim mem. spec. pointer
idata	1	0	1
xdata	2	1	2
pdata	3	FEh	1
data	4	0	1
code	5	FFh	2

In die anderen beiden Bytes des generischen Zeigers wird die Adresse eingetragen:

Zeiger-Adresse	Zeiger-Adresse+1	Zeiger-Adresse+2
Speicherbereichcode	Hi-Adresse	Lo-Adresse

Da der *generic pointer* zur Laufzeit interpretiert wird (Zugriff über Bibliotheksfunktion wegen Fallunterscheidung des Speicherbereichcodes), ist er auch entsprechend langsam. Für schnelle Applikationen ist es in jedem Fall empfehlenswert den *memory specific pointer* zu verwenden. Dieser ist erkennbar an der Angabe des Speichertyps bei der Deklaration, die bei der Programmierung festlegt, in welchen Speicherbereich der Zeiger zeigt (Kap. 2.4). Auf diese Weise kann der Compiler gleich den richtigen Maschinencode der CPU für die Adressierung einsetzen, das Interpretieren zur Laufzeit entfällt.
Ein Zeiger, der auf eine Variable zeigt, die explizit z.B. im *xdata*-Bereich liegt, sieht so aus:

```
int xdata *pnMonat;     /* Zeiger auf eine Integer-Variable in xdata */
```

Die Lage des Zeigers selbst ist in diesem Fall durch das Speichermodell vorgegeben.
Weitere Deklarationsbeispiele sind in Kap. 2.5 zu finden.

2.12.4 Zeigerarithmetik

Mit Zeigern kann auch gerechnet werden. Die bei einfachen Datentypen mögliche In- und Dekrementierung kann auch mit Zeigern vorgenommen werden, wenn die Datenobjekte in der Form eines Arrays abgelegt sind. Denn nur dort ist garantiert, daß die einzelnen Elemente lückenlos im Speicher abgelegt sind. Wird der Wert eines Zeigers verändert, verändert sich die Adresse. Aus diesem Grunde wird beim Inkrementieren nicht einfach 1 addiert, sondern die Adresse um die einfache Größe des Datenobjekts erhöht, auf das der Zeiger zeigt. Das gleiche gilt auch für das Dekrementieren. Es ist bei Zeigern auch möglich, einen Ganzzahlwert zu addieren bzw. zu subtrahieren. Der Compiler multipliziert die Größe des Datenobjekts mit diesem Ganzzahlwert, um die korrekte Adresse zu bestimmen.

2.12.5 Zeiger auf void

Ein *void** (Zeiger auf void) ist ein universeller Zeiger. Er kann genauso benutzt werden, wie in frühen Sprachdefinitionen der *char** (Zeiger auf *char*). Dieser erst in ANSI C aufgenommene Zeigertyp hebt die Verwirrung zwischen einem universellen Zeiger und einem Zeiger auf *char* auf. Alle Funktionen, mit denen dynamisch Speicher angefordert werden kann, geben die Adresse des zugeordneten Speicherbereichs in Form dieses typlosen Zeigers zurück. Durch eine explizite Typumwandlung muß der typlose Zeiger in den gewünschten Zeigertyp umgewandelt werden, bevor auf die Werte, auf die der Zeiger zeigt, zugegriffen oder mit ihm gerechnet werden kann.

2.12.6 Zeiger auf Funktionen

Zeiger auf Funktionen tauchen z.B. dann auf, wenn Funktionen indirekt aufgerufen oder Interruptvektoren verbogen und dann wieder restauriert werden sollen.
C wird gelegentlich wegen der Syntax seiner Vereinbarungen kritisiert, wenn Zeiger und Adressoperatoren ins Spiel kommen, besonders aber, wenn es sich dabei um Zeiger auf Funktionen handelt. Um ein wenig Klarheit zu bekommen sollen folgende leicht unterschiedliche Deklarationen dienlich sein:

```
int * nWert;            /*Zeiger auf eine Variable vom typ int (bekannt)*/
int * Func(void);       /*Prototyp einer Funktion, die einen Zeiger auf int liefert (bekannt)*/
int (*pfnFunc)(void);   /* Zeiger auf eine Funktion, die int liefert (neu)*/
int * (*pfnFunc)(void);/* Zeiger auf eine Funktion, die einen Zeiger auf int liefert (neu)*/
```

Beispiel eines kleinen Programms, welches maximal 1000 Zeichen ausgibt:

```
struct
    {
    byte *abyData;
    word wAnz;
    void (*pfnFunc)(word wWert);
   }t_SER_DATA={NULL,0,NULL};
word wANZAHL=1000;

void BeiFunc1(word wWert)
{
if(wANZAHL>wWert)
    wANZAHL-=wWert;
}

void main(void)
{
while(wANZAHL && (!t_SER_DATA.pfnFunc))
    {
    t_SER_DATA.abyData="Hallo";
    t_SER_DATA.wAnz=strlen(t_SER_DATA.abyData);
    t_SER_DATA.pfnFunc=BeiFunc1;
    TI=1;
    }
}

void Serial_Int(void) interrupt 4 using 1
{
static word wGesendet=0;
if(t_SER_DATA.wAnz)
    {
    t_SER_DATA.wAnz--;
    SBUF=*t_SER_DATA.abyData++;
    wGesendet++;
    }
else if (t_SER_DATA.pfnFunc!=NULL)
    {
    t_SER_DATA.pfnFunc(wGesendet);
    wGesendet=0;
    t_SER_DATA.pfnFunc=NULL;
    }
}
```

Hier sei angemerkt, daß C51 erst ab der Version V3.40 indirekte Funktionsaufrufe mit Parametern zuläßt.

2.13 Weitere Kennzeichner

2.13.1 Eigene Datentypen erzeugen: typedef

Manchmal reichen die zur Verfügung gestellten Typen nicht aus, oder man möchte andere Namen für die Standard-Datentypen vereinbaren. Dies geht spielend mit der *typedef*-Anweisung. Eine Situation, in der häufig die *typedef*-Anweisung benutzt wird, ist die Schaffung von vorzeichenlosen Datentypen, die ja bekanntlich in C mit dem Typvorsatz *unsigned* umschrieben werden müssen. Um ein *unsigned int* als *word* bezeichnen zu können, läßt sich das zweite Beispiel einsetzen.

```
typedef unsigned char byte;
typedef unsigned int word;
typedef unsigned long dword;
typedef struct
    {
    byte byAv;
    word wAvSum;
    byte byFiltertiefe;
    }typTiefpass;
```

In dieser Ausgabe werden die Datentypen *byte*, *word* und *dword* ohne weiteren Kommentar als die entsprechenden *unsigned* Datentypen zu *char*, *int* und *long* verwendet.

2.13.2 Nicht veränderbare Daten: const

Mit dem Datentyp-Kennzeichner *const* wird ein Objekt als unveränderbar gekennzeichnet. Ein Objekt, das als Konstante deklariert wurde, kann nicht auf der linken Seite einer Zuweisung stehen (vgl. Kap. 3.2.1), sein Wert kann nicht in- oder dekrementiert werden. *const* wird beispielsweise beim Prototyp der Funktion *printf()* benutzt. Das erste Argument, also die Formatzeichenkette, kann von der aufrufenden Funktion nicht verändert werden.

```
int printf (const char *,...);
```

Die Definition von Variablen kann ebenfalls um *const* erweitert werden.

2.13.3 Daten außer Kontrolle: volatile

Wenn das Typattribut *volatile* bei der Definition einer Variablen benutzt wird, nimmt der Compiler an den Anweisungen, in denen die so definierten Objekte verwendet werden, keine Optimierungen vor. Meist wird es eingesetzt, wenn das Objekt durch Prozesse verändert werden kann, die außerhalb der Kontrolle des Programms liegen. Die häufigste Optimierung ist statt des zweifachen Einlesens einer Variablen diese nur einmal auszulesen, in einem Register zwischenzuspeichern, und von dort wieder zu verwenden. Gerade in der Umgebung von Microcontrollern ist es üblich, daß, wenn eine Hardwarekomponente wie z.B. ein A/D-Wandler in den externen Speicherbereich mit ihrer Datenstruktur gemapped ist, diese Datenstruktur mit dem Kennzeichner *volatile* ausgestattet wird. Es gibt beispielsweise A/D-Wandler, die bei zweifachem Auslesen zuerst das höherwertige und dann das niederwertige Byte liefern. Bei einer Optimierung durch Zwischenspeicherung würde in diesem Fall der 16-Bit-Wert aus zwei höherwertigen Bytes gebildet werden, was ein falsches Ergebnis zur Folge hat.

2.13.4 Absolute Variablenplatzierung: **_at_**

Dieses nur in C51 ab der Version 3.40 implementierte Schlüsselwort ermöglicht es Variablen an absolute Adressen zu platzieren. In Verbindung mit dem Schlüsselwort *volatile* können hiermit Objekte entsprechend einer Hardware beispielsweise im *xdata*-Bereich abgebildet werden (memory mapped hardware).

```
xdata byte volatile AD_WANDLER _at_ 0x2000;
```

Auf `AD_WANDLER` kann nun wie auf eine Variable zugegriffen werden.
Variablen mit dem Schlüsselwort ***_at_*** können nur auf globaler Ebene deklariert oder definiert werden, und können nicht initialisiert werden. Der Zugriff ist genau so effizient, wie man es in Assembler nicht besser programmieren kann.

Für alle Compilerversionen (somit auch ältere) hat der Compilerhersteller KEIL eine allgemeine Formulierung ohne das Schlüsselwort ***_at_*** vorgesehen, die zur gleichen Effizienz führen soll: Die mitgelieferte Datei ABSACC.H bietet das Makro

```
#define XBYTE ((unsigned char volatile xdata *) 0)
```

wobei nach der Makrodefinition

```
#define AD_WANDLER XBYTE[0x2000]
```

auf `AD_WANDLER` als Makro wie eine Variable zugegriffen werden kann.
An dieser Stelle sei allerdings bei der Benutzung dieser Datei Vorsicht geboten, denn für ein Byte mag dieses Makro noch funktionieren. Da jedoch die angegebene Adresse als Index auf ein Array wirkt, also bei allen anderen Datentypen noch mit der Objektgröße multipliziert wird, erfolgt ein Zugriff auf das entsprechende Vielfache der gewünschten Adresse statt auf die Adresse selbst. Erst die Handbücher ab der Compilerversion V7.x weisen auf diesen Umstand hin.

Abhilfe schafft hier eine ähnliche Konstruktion:

```
#define X_ADR(DT,X) ((DT volatile xdata *) ((byte volatile xdata *) (X)))
#define X_BYTE(X) (*X_ADR(byte,X))
#define X_WORD(X) (*X_ADR(word,X))
```

Nach der anschließenden Definition von

```
#define AD_WANDLER X_BYTE(0x2000)
```

kann auf `AD_WANDLER` als Makro über den Indirektionsoperator wie auf eine Variable an der Adresse 2000h zugegriffen werden.
Die erste Zeile ist ein allgemeines Makro, welches in der zweiten Zeile für ein *byte* und in der dritten Zeile für ein *word* benutzt wird. Der Unterschied zu der Variante des Compilerherstellers ist der, daß zunächst für jede Adresse eine Typumwandlung (vgl. Kap. 3.8.5) in ein *byte* (*unsigned char*) erfolgt. Dann erst wird in den gewünschten Datentypen umgewandelt und somit stimmt die Adresse wieder.
Wie man in der zweiten und dritten Zeile erkennt, kann man jeden beliebigen Datentypen einsetzen, und ein eigenes Makro kreieren.

Auf diese Weise können auch komplexe Strukturen abgebildet werden. Beispiel:

```
typedef struct
    {   /* Datenstruktur eines Musterbausteins laut Datenblatt */
    word wKontrollregister;
    byte byWert1;
    word wDatum1;
    }typtMUSTER_BAUSTEIN;

#define MUSTER_BAUSTEIN    X_ADR(typtMUSTER_BAUSTEIN,0x2000)
```

Anschließend kann über den Strukturverweis-Operator auf das Kontrollregister unseres Musterbausteins mit `MUSTER_BAUSTEIN->wKontrollregister` an der Adresse 2000h wie auf ein Element einer Struktur zugegriffen werden. In älteren C51-Versionen ist an ANSI-C vorbei sogar ein Zugriff über `MUSTER_BAUSTEIN.wKontrollregister` möglich. Beispiele:

```
word wKoReg;
wKoReg=MUSTER_BAUSTEIN->wKontrollregister;      /* nach ANSI */
wKoReg=MUSTER_BAUSTEIN.wKontrollregister;       /* C51 an ANSI vorbei */
```

Die Effizienz ist wiederum optimal, d.h. der DPTR wird direkt mit der Zieladresse geladen, und anschließend wird der Inhalt in den Akku bzw. nach `wKoReg` kopiert.

Hier, wie auch in allen anderen Fällen, in denen `AD_WANDLER` nicht als Variable sondern als Makrodefinition auftaucht, kann sie bei einem Emulator nicht als Variable gedebuggt werden, auch wenn der Zugriff auf `AD_WANDLER` aussieht wie der auf eine Variable. Benutzt man diesen Konstrukt für eine memory mapped Hardware, dann spielt dies normalerweise keine Rolle, weil diese aufgrund der unkontrollierbaren Zugriffe durch den Debugger nicht gedebuggt werden sollte.

3 Operatoren

ANSI C kennt mehr als 40 Operatoren. Mit diesem Wert übersteigt, und das ist bei Programmiersprachen nicht üblich, bei C die Anzahl der Operatoren die Anzahl der Schlüsselworte der Sprache. Alle von ANSI C definierten Operatoren sind in C51 implementiert. Der große Teil der Operatoren stammt bereits aus der Zeit der ersten Sprachdefinition durch Kernighan & Ritchie.

3.1 Übersicht

Bevor die einzelnen Operatoren, ihre Syntax, Wirkung und den Verwendungszweck im einzelnen betrachtet werden, sind hier zunächst alle Operatoren auf einen Blick mit einer kurzen Beschreibung dargestellt.

Operator	Kurzbeschreibung	Gruppe
+	Addiert zwei Operanden.	binär
+=	Addiert zwei Operanden und weist Ergebnis zu.	binär
-	Negativer Wert eines Operanden	unär
-	Subtrahiert zwei Operanden.	binär
-=	Subtrahiert zwei Operanden und weist Ergebnis zu.	binär
*	Multipliziert zwei arithmetische Operanden.	binär
*=	Multipliziert zwei arithmetische Operanden und weist Ergebnis zu.	binär
/	Dividiert zwei arithmetische Operanden.	binär
/=	Dividiert zwei arithmetische Operanden und weist Ergebnis zu.	binär
%	Modulo-Operator, ermittelt den Rest einer Ganzzahldivision.	binär
%=	Modulo-Operator mit gleichzeitiger Zuweisung.	binär
!	Nicht-Operator, Verneinung des Ausdrucks	unär
!=	Vergleicht zwei arithmetische Operanden bzw. Ausdrücke auf Ungleichheit.	binär
&	Bei einem Operanden liefert dieser Operator die Adresse des Operanden.	unär
&	Bei zwei Operanden wird eine bitweise UND-Verknüpfung ausgeführt.	binär
&=	Bitweise UND-Verknüpfung und Zuweisung.	binär
&&	Logische UND-Verknüpfung.	binär
()	Cast-Operator, erzwingt Typumwandlung.	unär
,	Operator für sequentielle Auswertung, Kommaoperator.	-
++	Inkrementiert einen skalaren Operanden.	unär
--	Dekrementiert einen skalaren Operanden.	unär
->	Strukturverweis-Operator.	unär
.	Punkt-Operator.	unär
*	Indirektions-Operator.	unär
=	Zuweisungsoperator.	binär
==	Testet auf Gleichheit.	binär
<	Kleiner als.	binär
>	Größer als.	binär
<=	Kleiner als oder gleich.	binär
>=	Größer als oder gleich.	binär
<<	Bitweise Linksverschiebung.	binär
<<=	Bitweise Linksverschiebung und Zuweisung des Ergebnisses.	binär
>>	Bitweise Rechtsverschiebung.	binär
>>=	Bitweise Rechtsverschiebung und Zuweisung des Ergebnisses.	binär
?:	Bedingungsoperator.	ternär
[]	Array-Indizierung.	unär
^	Bitweises EXKLUSIV ODER.	binär
^=	Bitweises EXKLUSIV ODER und Zuweisung des Ergebnisses.	binär
|	Bitweises ODER.	binär
|=	Bitweises ODER und Zuweisung des Ergebnisses.	binär
||	Logisches ODER.	binär
~	Komplement (bitweise Invertierung).	unär
sizeof	Ermittelt Größe des Operanden in Bytes.	unär

Fast alle Operatoren lassen sich, je nachdem mit wie vielen Operanden sie arbeiten, einer der folgenden Gruppen zuordnen:

- unäre Operatoren (arbeiten mit einem Operanden),
- binäre (arbeiten mit zwei Operanden) und
- ternäre Operatoren (arbeiten mit drei Operanden).

3.2 Zuweisungen

Mit Zuweisungs-Operatoren können in C sowohl Werte umgeformt als auch zugewiesen werden. Neben dem einfachen Zuweisungs-Operator gibt es auch die sogenannten zusammengesetzten Zuweisungs-Operatoren, mit denen in einer Anweisung sowohl eine Berechnung als auch eine Zuweisung durchgeführt werden kann.

3.2.1 Lwert

Grundsätzlich gilt, daß mit einer Zuweisung dem linken Operanden der Wert des rechten Operanden zugewiesen wird. Damit dies möglich ist, muß der linke Operand veränderbar sein, sich also auf eine Adresse im Speicher beziehen. Diese Ausdrücke werden "Lwert-Ausdrücke". (Ivalue) genannt. Der Versuch, einem Operanden, der kein "Lwert-Ausdruck" ist, einen Wert zuzuweisen, erzeugt eine Fehlermeldung des Compilers.

3.2.2 Einfache Zuweisungen

C besitzt eine Fülle von Zuweisungs-Operatoren, von denen die einfache Zuweisung der einfachste ist. Der Operator besteht aus dem Gleichheitszeichen "=". Bei der einfachen Zuweisung erhält der linke Operand den Wert des rechten Operanden. Hierbei wird vor der Zuweisung der eine Typkonvertierung von rechts nach links vorgenommen.
Beispiel:

```
byZahl=15;
```

Die Zahl erhält den Wert 15.

3.2.3 Zusammengesetzte Zuweisungen

Neben dem einfachen Zuweisungs-Operator "=" existieren in C die sogenannten zusammengesetzten Zuweisungs-Operatoren, mit denen durch einen Operator gleichzeitig eine Berechnung und eine Zuweisung durchgeführt werden kann. Die zusammengesetzten Zuweisungs-Operatoren haben das folgende Format:

```
<X>=
```

wobei <X> durch die Operatoren "+", "-", "*", "/", "%", "<<", ">>", "&", "^" und "|" ersetzt werden kann.
Beispiel:

```
byZahl-=3;
```

Es wird der Wert 3 von `byZahl` abgezogen.

3.3 Datenzugriff

Mehrere Operatoren von C dienen dem Zugriff auf Daten. Die folgende Tabelle gibt einen Überblick:

Operator	Bedeutung
*	Indirektions-Operator
[]	Array-Indizierung
.	Punkt-Operator
->	Strukturverweis-Operator

*3.3.1 Indirektions-Operator *

Der Indirektions-Operator "*" verweist bei einem Operanden, der ein Zeigertyp ist, indirekt auf dessen Wert. Der Typ des Operanden bestimmt den Typ des Ergebnisses. Wenn also der Operand ein Zeiger auf *byte* ist, hat auch das Ergebnis den Datentyp *byte*.

Beispiel:

```
byte eine_funktion(byte *pbyZeiger)
{
byte byErgebnis=7;
*pbyZeiger = byErgebnis;
return 0;
}
```

- Der Indirektions-Operator kann nicht auf einen unvollständigen Typ, beispielsweise einen Zeiger auf void, angewendet werden.
- Der Indirektions-Operator wird auch dazu benutzt, um über an Funktionen übergebene Zeiger mehrere von der Funktion berechnete Werte zurückgeben zu können.
- Das Zeichen "*" wird ebenfalls bei der Deklaration von Variablen benutzt, um anzugeben, daß die deklarierte Variable ein Zeiger ist. Es handelt sich dann dabei nicht um den Indirektions-Operator. Die Bedeutung ergibt sich somit immer aus dem Kontext.

3.3.2 Array-Indizierung []

Der Operator "[]" kann in verschiedenen Zusammenhängen auftauchen. Er kann benutzt werden

- um ein Array zu deklarieren,
- um ein Array als Argument an eine Funktion zu übergeben,
- auf ein einzelnes Element innerhalb eines Arrays zuzugreifen.

Wenn `anArray` nach der Definition von

```
int anArray[30];
```

ein Array 30 Elementen des Typs *int* ist, dann entspricht der Ausdruck

```
anArray[4]
```

dem Ausdruck

```
*(anArray+4)
```

- Das erste Element eines Arrays besitzt den Index 0. Somit bezieht sich der Ausdruck
  `anArray[20]`
  auf das 21. Element des Arrays.
- Wenn ein Array an eine Funktion übergeben wird, erhält die Funktion die Adresse des ersten Elements des Arrays. Diese Übergabe ist identisch mit der Übergabe eines Zeigers auf das Array.

3.3.3 Punkt-Operator .

Der Punkt-Operator "." dient dazu, um auf einzelne Elemente einer Struktur oder einer Union zuzugreifen.

objekt.element

objekt Ist der Name einer Struktur oder einer Union.
element Ist der Name eines Elements der Struktur.

Beispiel:

Im folgenden Beispiel wird zuerst das Strukturmuster `typtPerson` deklariert und anschließend in Zeile 11 die Variable `tKunde` von diesem Typ definiert. In den Zeilen 13 und 14 werden beiden Elementen der Strukturvariablen Werte zugewiesen, und in Zeile 16 der Inhalt der Variablen `tKunde` ausgegeben. In beiden Fällen wird der Punkt-Operator benutzt, um auf die Elemente zuzugreifen.

```
1: #include <stdio.h>
2:
3: struct typtPerson
4:    {
5:    char acName[20];
6:    long lPlz;
7:    };
8:
9 : main()
10: {
11: struct typtPerson tKunde;
12:
13: strcpy(tKunde.acName, "Otter");
14: tKunde.lPlz = 76135;
15:
16: printf("Kunde: %s, Ort: %ld\", tKunde.acName, tKunde.lPlz);
17: }
```

3.3.4 Strukturverweis-Operator ->

Der Strukturverweis-Operator "->" dient zum Zugriff auf einzelne Elemente einer Struktur oder Union, wenn es sich bei der Variablen um einen Zeiger auf eine Struktur oder Union handelt.

pObjekt->Element

pObjekt Zeiger auf eine Variable vom Typ Struktur oder Union.
Element Ein Element der Struktur oder Union.

Das folgende Beispiel ist vom Aufbau her mit dem Beispiel zum Punkt-Operator identisch. Jedoch wird hier ein Zeiger auf die Strukturvariable `typtPerson` definiert und über diesen Zeiger auf die einzelnen Elemente zugegriffen.

```
1: #include <stdio.h>
2:
3: struct typtPerson
4:    {
5:    char acName[20];
6:    long lPlz;
7:    };
8:
9 : main()
10: {
11: struct typtPerson tKunde;
12: struct typtPerson * ptKunde = &tKunde;
13:
14: strcpy(ptKunde->acName, "Otter");
15: ptKunde->lPlz = 76135;
16: printf("Kunde: %s, Ort: %ld\", ptKunde->acName, ptKunde->lPlz);
17: }
```

3.4 Arithmetische Operatoren

Mit den arithmetischen Operatoren kann man, wie es der Name vermuten läßt, einfache arithmetische Operationen durchführen, wie sie auch in anderen Programmiersprachen üblich sind.

Operator	Bedeutung
+	Addiert zwei Operanden.
-	Subtrahiert zwei Operanden.
*	Multipliziert zwei arithmetische Operanden.
/	Dividiert zwei arithmetische Operanden.
%	Modulo-Operator, ermittelt den Rest einer Ganzzahldivision.

Alle arithmetischen Operatoren gehören zu den sogenannten binären Operatoren, d.h. sie arbeiten immer mit zwei Operanden. Bei der Auswertung gilt: Punktrechnung geht vor Strichrechnung. Die Operatoren "+" und "-" besitzen die gleiche Priorität, die jedoch geringer ist als die der Operatoren "*", "/" und "%", die untereinander wieder die gleiche Priorität besitzen. Bei gleicher Priorität werden die Operatoren von links nach rechts abgearbeitet.
Man kann durch Setzen von Klammern andere Prioritäten vergeben, was im Zweifelsfall immer sinnvoll ist.
Beispiel:

```
byZahl=3+5;
```

Der Zahl wird der Wert 8 zugewiesen.

3.5 Vergleichende Operatoren

Mit den vergleichenden Operatoren, die auch relationale Operatoren genannt werden, wird ein Vergleich zwischen zwei Operanden, die auch Ausdrücke sein können, durchgeführt. Das Ergebnis dieser Operation ist entweder wahr oder falsch. Ausdrücke, die vergleichende Operatoren enthalten, werden häufig als Kontrollbedingung in *do-while*- oder *for*-Schleifen und *if*-Anweisungen eingesetzt.

Operator	Bedeutung
==	gleich
!=	ungleich
<	kleiner als
<=	kleiner als oder gleich
>	größer als
>=	größer als oder gleich

Man beachte, daß es im ANSI-C keinen booleschen Datentyp gibt. Das Ergebnis einer Vergleichsoperation besitzt in ANSI-C den Datentyp *int*. In C51 hingegen gibt einen booleschen Datentyp, wobei Zwischenergebnisse meistens im Carry-Flag gespeichert werden.
C vergibt für das Ergebnis 'wahr' eine 1, für das Ergebnis 'unwahr' eine 0. Jeder auszuwertende Wert, der nicht 0 ist, wird von C als 'wahr' interpretiert.
Zu beachten ist, daß der Gleich-Operator aus zwei Gleichheitszeichen besteht, und der Operator für die einfache Zuweisung aus einem Gleichheitszeichen (häufige Fehlerquelle).

Beispiele:

```
if (byZahl1 == byZahl2)   /* Ist Zahl1 gleich Zahl2 dann */
    funktion();           /* wird aufgerufen */
if (byZahl1 = byZahl2)    /* Zahl1 bekommt den Wert von Zahl2 und */
    funktion();           /* wenn sie ungleich 0 sind wird aufgerufen */
```

3.6 Logische Operatoren

Die Programmiersprache C kennt drei logische Operatoren: UND, ODER und NICHT. Beim logischen UND- und ODER-Operator wird der Wahrheitswert der Operanden verknüpft, der logische NICHT-Operator konvertiert den logischen Wert seines Operanden in dessen Gegenteil.

Operator	Bedeutung
&&	logisches UND
||	logisches ODER
!	logisches NICHT

3.6.1 logisches UND &&

Mit dem Operator "&&" wird eine logische UND-Verknüpfung seiner Operanden durchgeführt. Beide Operanden müssen einen skalaren Datentyp besitzen. Das Ergebnis der Verknüpfung hat nur dann den Wert ‘wahr' (also ungleich Null), wenn beide Operanden den Wert 'wahr' besitzen. Die Operanden werden von links nach rechts ausgewertet. Wenn der erste Operand den Wert 'unwahr' (also Null) ergibt, wird der zweite Operand nicht ausgewertet. Die folgende Wahrheitstabelle zeigt Ihnen die möglichen Kombinationen.

1. Operand	2. Operand	Ergebnis
wahr	wahr	wahr
wahr	falsch	falsch
falsch	wahr	falsch
falsch	falsch	falsch

Beispiel:

```
if(byZahl1 == 5 && byZahl2==8)
...
```

3.6.2 logisches ODER ||

Der logische ODER-Operator "||" (2 mal ASCII 124) verknüpft seine Operanden insoweit, als daß das Ergebnis der Verknüpfung dann den Wert 'wahr' hat, wenn einer oder beide Operanden den Wert 'wahr' ergibt.

1. Operand	2. Operand	Ergebnis
wahr	wahr	wahr
wahr	falsch	wahr
falsch	wahr	wahr
falsch	falsch	falsch

Beispiel:

```
if(byZahl1 == 5 || byZahl2==8)
...
```

3.6.3 logisches NICHT !

Der logische NICHT-Operator verkehrt den Wahrheitswert seines logischen Operanden in dessen Gegenteil. Dabei wird der Wert 0 in den Wert 1 umgewandelt, und alle Werte ungleich 1 zu 0.

Beispiele:

```
if(!STOP_TASTE)
...

if(!(byZahl1 == 17 || byZahl1==13))
...
```

3.7 Bitweise Operatoren

Bei den bitweisen Operatoren wird im Gegensatz zu den logischen Operatoren nicht der Wahrheitswert der Operanden sondern die einzelnen Bits der Operanden verknüpft. Die Ähnlichkeit der Operationszeichen zu denen der logischen Operatoren ist eine häufige Fehlerquelle.

Operator	Bedeutung
&	Bitweises UND
|	Bitweises ODER
^	Bitweises EXKLUSIV ODER
~	Komplement (bitweise Invertierung)
<<	Bitweise Linksverschiebung
>>	Bitweise Rechtsverschiebung

3.7.1 bitweises UND &

Wenn der Operator "&" mit zwei Operanden benutzt wird, handelt es sich um den bitweisen UND-Operator. Die Operanden können beliebige Ganzzahlen sein. Die beiden Operanden werden bitweise verglichen. Das Bit im Ergebnis wird nur dann gesetzt, wenn in beiden Operanden die korrespondierenden Bits gesetzt sind. Dieser Operator kann eingesetzt werden, um gezielt Bits in seinem ersten Operanden zu löschen. In der ASCII-Tabelle unterscheiden sich die Klein- von den Großbuchstaben des englischen Alphabets (also nicht die deutschen Umlaute) binär dadurch, daß bei den Kleinbuchstaben immer Bit 5 gesetzt ist. Um einen Klein- in einen Großbuchstaben umzuwandeln, soll Bit 5 des ersten Operanden gelöscht werden. Um dies zu erreichen werden im zweiten Operanden, der Maske, alle Bits gesetzt, bis auf Bit 5, das gelöscht werden soll. Dadurch bleiben die anderen Bits unverändert.

```
  0 1 1 0  0 0 0 1    Buchstabe 'a'
& 1 1 0 1  1 1 1 1    Maske (DFh)
  0 1 0 0  0 0 0 1    Buchstabe'A'
```

3.7.2 bitweises ODER |

Der Operator für die bitweise ODER-Verknüpfung vergleicht die Bitmuster seiner Operanden und setzt das entsprechende Bit im Ergebnis, wenn eines oder beide der Bits in den Operanden gesetzt sind. Dieser Operand kann beispielsweise dazu verwendet werden, um gezielt einzelne Bits zu setzen, ohne die anderen Bits der ersten Operanden zu verändern. Ein Beispiel dafür ist die Umwandlung von Groß- in Kleinbuchstaben. In der ASCII-Tabelle unterscheiden sich die Klein- von den Großbuchstaben des englischen Alphabets (also nicht die deutschen Umlaute) binär dadurch, daß bei den Kleinbuchstaben immer Bit 5 gesetzt ist.

```
  0 1 0 0  0 0 0 1    Buchstabe 'A'
| 0 0 1 0  0 0 0 0    Maske (20h)
  0 1 1 0  0 0 0 1    Buchstabe 'a'
```

3.7.3 bitweises EXKLUSIV-ODER ^

Der Operator für bitweise exklusive ODER-Verknüpfung vergleicht die Bitmuster seiner Operanden und setzt das entsprechende Bit im Ergebnis, wenn nur eines der Bits in den Operanden, aber nicht beide, gesetzt sind. Beispiel:

```
  0 0 0 1  1 0 0 1
^ 0 0 0 0  1 1 0 0
  0 0 0 1  0 1 0 1
```

3.7.4 bitweises Komplement ~

Der bitweise unäre Komplement-Operator "~" kippt alle Bits seines Operanden. Durch die Invertierung werden im Ergebnis alle Bits, die im Operanden gesetzt waren, gelöscht, und alle Bits, die gelöscht waren, gesetzt. Beispiel:
Die Zahl 4 besitzt folgendes Bitmuster:

0 0 0 0 0 1 0 0

Die Operation ~4 erzeugt daraus folgendes Muster:

1 1 1 1 1 0 1 1

3.7.5 Linksverschiebung <<

Bei der bitweisen Linksverschiebung werden die einzelnen Bits eines Operanden um die festgelegte Anzahl nach links verschoben. Die niederwertigste Stelle, an der eine Lücke entstehen würde, wird mit einer 0 besetzt. Der Operator kann beispielsweise für eine schnelle Programmierung der Multiplikation mit den Potenzen der Zahl 2 benutzt werden. Der Ausdruck 3 << 2 entspricht 3 * 2^2 und erzeugt folgendes Ergebnis:

vor Verschiebung:	3	0000 0011b
nach Verschiebung:	12	0000 1100b

3.7.6 Rechtsverschiebung >>

Bei der bitweisen Rechtsverschiebung werden die einzelnen Bits eines Operanden um die festgelegte Anzahl nach rechts verschoben. Die höchstwertigste Stelle, an der eine Lücke entstehen würde, wird bei vorzeichenlosen Zahlen mit einer 0 besetzt. Der Operator kann beispielsweise für eine schnelle Programmierung der Division durch die Potenzen der Zahl 2 benutzt werden.
Der Ausdruck 80 >> 3 entspricht 80 / 8 und erzeugt folgendes Ergebnis:

vor Verschiebung:	80	0101 0000b
nach Verschiebung:	10	0000 1010b

3.8 Verschiedene Operatoren

3.8.1 sizeof-Operator

Der sizeof-Operator liefert als Ergebnis die Größe seines Operanden in Bytes. Hierzu muß die Größe zum Compilierzeitpunkt bekannt sein, ansonsten wird eine Fehlermeldung erzeugt. Das Ergebnis ist als Konstante der Größe eines Intergers zu betrachten.

- Wenn einer der Typen *char*, *unsigned char* oder *signed char* als Argument verwendet wird, ist das Ergebnis per Definition immer 1.
- Der Name eines Typs muß in runde Klammern eingeschlossen werden, Namen von Arrays, einer Funktion oder Struktur können in runde Klammern eingeschlossen werden.
- Wird *sizeof* bei einem Array angewendet, ist das Ergebnis die Gesamtgröße des Arrays in Bytes.
- Möchte man bei Arrays die Zahl der Elemente ermitteln, so sei hier auf das Makro *ELEMENTE()* in Kap. 8.2.6 verwiesen
- Handelt es sich beim Argument um eine Struktur oder Union, wird die tatsächliche Größe zurückgegeben, in der auch Füllbits zur Ausrichtung an Speichergrenzen enthalten sein können. Das Ergebnis der sizeof-Operation kann dann von der Summe des Speicherplatzbedarfs der einzelnen Elemente verschieden sein.
- Wenn *sizeof* innerhalb einer Funktion benutzt wird, um die Größe eines Arrays oder einer Funktion zu ermitteln, die als Funktionsargumente benutzt werden, liefert *sizeof* die Größe des Zeigers auf das Objekt zurück, da Funktions- und Arraynamen, die als Funktionsargumente benutzt werden, in Zeiger umgewandelt werden.

3.8.2 Adreß-Operator &

Wenn der Operator "&" mit einem Operanden benutzt wird, liefert er die Adresse seines Operanden als Zeiger auf den Datentyp des Operanden. Der Operand kann jeder Bezeichner sein, der einen Lwert verkörpert, sowie der Bezeichner einer Funktion. Der Operator kann jedoch weder auf ein Objekt des Typs *bit*, noch auf das Element eines Bitfeldes noch auf ein Objekt, das mit der Speicherklasse *register* definiert wurde, angewendet werden.

In folgendem Beispiel wird der Adreß-Operator eingesetzt, um die Adresse einer Variablen einem Zeiger zuzuweisen.

```
int xdata nMonat;
int xdata *pnMonat;

pnMonat=&nMonat;
```

Mit der ersten Anweisung wird eine Variable vom Typ *int* definiert, mit der zweiten ein Zeiger auf eine Variable vom Typ *int*. Mit der Zuweisung in der dritten Zeile wird der Variablen `pnMonat` die Adresse der Variablen `nMonat` zugewiesen.

3.8.3 Bedingungsoperator (bedingte Bewertung) ?:

ANSI C kennt einen ternären Operator, der Bedingungs-Operator oder bedingte Bewertung genannt wird. Der Bedingungs-Operator (?:) ist eigentlich eine Kurzform der if-else-Anweisung. Das allgemeine Format dieses Operators lautet folgendermaßen:

Bedingung ? Ausdruck1 : Ausdruck2

Die Auswertung geht folgendermaßen vor sich:

- Wenn die Auswertung von Bedingung nicht 0 (also 'wahr') ergibt, wird das Ergebnis von Ausdruck1 geliefert.
- Ergibt die Auswertung der Bedingung 0 (also 'unwahr') wird Ausdruck2 ausgewertet.

Das folgende Beispiel verwendet den Bedingungs-Operator, um der Variablen *nMax* die größere von zwei Zahlen *nEins, nZwei* zuzuweisen.

```
int nMax, nEins, nZwei;
nEins = 1;
nZwei = 2;
nMax = (nEins > nZwei) ? nEins : nZwei;
```

Die gleiche Zuweisung kann auch mit einer if-else-Anweisung geschrieben werden:

```
if(nEins > nZwei)
    nMax = nEins;
else
    nMax = nZwei;
```

Seinen Einsatz findet dieser Operator meistens bei der Parameterübergabe an eine Funktion, bei der ein Parameter abhängig von einer Bedingung zuvor noch ausgewertet werden soll. Hierbei wird dann keine Zwischenvariable benötigt. Im folgenden Beispiel soll der Variablen `fErgebnis` der kubische Wert der höheren von beiden Zahlen `fEins` oder `fZwei` zugewisen werden.

```
float fErgebnis, fEins, fZwei;
fEins = 1.8;
fZwei = 2.3;
fErgebnis = pow( (fEins>fZwei)?fEins:fZwei , 3.0 );
```

3.8.4 In- und Dekrement-Operator ++ --

Die Programmiersprache C stellt zwei Operatoren zur Verfügung, mit denen der Wert einer Variablen inkrementiert (um 1 erhöht) oder dekrementiert (um 1 vermindert) werden kann, wobei die Operatoren zusätzlich vor oder nach der Variablen stehen können.
Die Position des Operators legt fest, wann die Erhöhung oder Erniedrigung der Variablen berechnet wird. Die Notation, bei der der Operator vor der Variablen steht, wird Präfix-Schreibweise genannt, und führt dazu, daß der Wert der Variablen zuerst neu berechnet, bevor er benutzt wird. In der Postfix-Schreibweise (der Operator steht nach der Variablen) wird der Wert des Operanden benutzt, bevor er verändert wird.
Die In- und Dekrement-Operatoren können auch bei Zeigern benutzt werden. In diesem Fall wird die Adresse um die Größe des Objekts, auf das der Zeiger zeigt, erhöht oder vermindert.

3.8.5 Explizite Typumwandlung ()

Die runden Klammern "()" haben in C drei verschiedene Bedeutungen. Erstens werden sie bei Funktionen eingesetzt und stehen dann immer nach dem Funktionsbezeichner, zweitens dienen sie dazu den Vorrang bestimmter Operationen zu definieren. Darüber hinaus dienen sie auch als Operator zur expliziten Typumwandlung (type casting). Man erkennt diese Konstruktion daran, daß dem Namen eines Bezeichners die Klammern voran stehen, in denen sich nur eine Typbezeichnung befindet. Mit dem Cast-Operator wird der Typ des Bezeichners in den in Klammern eingeschlossenen Typ umgewandelt.

(Neuer Typ) Bezeichner

Beispiel:

```
#define ERROR -1
byte byI;

byI=(byte) ERROR;
```

3.8.6 Sequentielle Auswertung ,

Wenn das Zeichen "," außerhalb einer Deklaration oder Parameterliste eines Funktionsaufrufs benutzt wird, dient es als Komma-Operator. Er wird üblicherweise an Stellen verwendet, an denen nur ein Ausdruck erlaubt ist, um zwei oder mehr Ausdrücke auswerten.

- Das Ergebnis der durch den Komma-Operator getrennten Ausdrücke hat den gleichen Wert und Typ wie der rechte Operand.
- Die Operanden können einen beliebigen Typ besitzen.
- Typumwandlungen werden nicht durchgeführt.
- Funktionsaufrufe können nicht durch Komma-Operatoren getrennt werden.

Das häufigste Beispiel verwendet den Komma-Operator im Reinitialisierungsteil einer for-Schleife. Bei jedem Schleifendurchlauf werden die Variablen `nI` und `nJ` inkrementiert.

```
for (nI=nJ=0; nI<MAXWERT; nI++, nJ++)
    /* Anweisungsblock */
```

Im folgenden Beispiel wird der linke Operand (`nZahl2++`) zuerst ausgewertet. Wenn `nZahl2` nach dem Inkrementieren z.B. den Wert 8 hat, wird anschließend das Ergebnis von `nZahl2 / 4` (also 2) der Variablen `nZahl1` zugewiesen:

```
nZahl1 = (nZahl2++, nZahl2 / 4);
```

Mit folgendem Beispiel wird die geschweifte Klammerung für einen Anweisungsblock nach einer *if*-Abfrage umgangen:

```
if(++nZahl1 > 5)
    nZahl2++, nZahl1=0;
```

3.9 Vorrang und Assoziativität

Diese wichtigen Begriffe sind im Zusammenhang mit den verschiedenen C-Operatoren von Bedeutung, um die Auswertung komplexer Ausdrücke durch den Compiler vorhersehen zu können. Man kann als Beispiel den Ausdruck nehmen:

```
*apnZeiger[4]
```

Die Frage ist, ob sich mit diesem Ausdruck der Wert ergibt, auf den pnZeiger[4] zeigt, oder ob das fünfte Element der Stelle, dessen Adresse sich im Zeiger gebildet wird. Der Vorrang der Operatoren legt fest, wie ein Ausdruck, in dem mehrere Operatoren vorkommen, ausgewertet wird. Die folgende Tabelle zeigt den Standard-Vorrang der einzelnen Operatoren, wobei die Operatoren mit dem höchsten Vorrang zuerst aufgeführt werden. Enthält ein Ausdruck mehrere Operatoren, die den gleichen Vorrang besitzen, wird der Ausdruck je nach Assoziativität der Operatorengruppe von links nach rechts oder von rechts nach links ausgewertet.

Operatoren	Assoziativität
`() [] -> .`	von links her
`! ~ ++ -- + - * & (cast) sizeof`	von rechts her
`* / %`	von links her
`+ -`	von links her
`<< >>`	von links her
`< <= > >=`	von links her
`== !=`	von links her
`&`	von links her
`^`	von links her
`|`	von links her
`&&`	von links her
`||`	von links her
`?:`	von rechts her
`= += -= *= /= %= &= ^= |= <<= >>=`	von rechts her
`,`	von links her

Unär haben +, - und * mehr Vorrang als binär.

Wie der Tabelle zu entnehmen ist, besitzt der Operator zum Array-Zugriff eine höhere Priorität als der Indirektions-Operator, und wird darum zuerst ausgewertet. Es entsteht also mit *apnZeiger[4] der Wert, dessen Adresse sich in apnZeiger[4] befindet.

4 Kontrollstrukturen

Je komplexer Programme werden, desto mehr ist es erforderlich, je nach Situation und Zustand des Programms zu verschiedenen Anweisungen zu verzweigen, um damit flexiblere Programme schreiben zu können.
Daneben entsteht bei den meisten Fragestellungen, für die eine softwaremäßige Lösung gesucht werden muß, die Notwendigkeit, bestimmte Anweisungen mehrfach auszuführen, um damit Daten zu manipulieren.
Die so umschriebenen Möglichkeiten zur Ablaufkontrolle finden sich, wie in anderen Programmiersprachen auch, natürlich in C wieder.
Dieses Kapitel erläutert die Möglichkeiten der bedingten Verzweigung, der Konstruktion von Schleifenanweisungen, und schließt mit der Vorstellung der Schlüsselworte *break*, *continue*, *return* und *goto* die Diskussion ab.

4.1 Bedingte Verzweigungen

Bedingte Verzweigungen dienen der Fortsetzung des Programms aufgrund einer Bedingung in eine bestimmte Richtung, ähnlich einer Weggabelung, an der man sich entscheidet entweder nach links oder nach rechts zu gehen.

4.1.1 if

Durch die Verwendung des Schlüsselwortes *if* wird die einfachste Form der bedingten Verzweigung eingeleitet. Die zu *if* gehörende Bedingung wird ausgewertet. Ergibt diese Bewertung das Ergebnis 'wahr' (also einen Wert ungleich Null), wird die zu *if* gehörende Anweisung ausgeführt. Ergibt die Auswertung des Ausdrucks das Ergebnis 'unwahr', wird die Ausführung des Programms mit der ersten Anweisung hinter der Verzweigung fortgesetzt.

```
if (Bedingung)
        Anweisung;
```

Bedingung — Die Bedingung, die ausgewertet wird, muß einen skalaren Datentyp ergeben. Hier kann der Aufruf einer Funktion stehen, bei dem dann das Ergebnis der Funktion ausgewertet wird, eine Ganzzahl, das Ergebnis einer arithmetischen Operation, oder der bei einem Vergleich ermittelte Wert. Zu beachten ist, daß in C hinter der Bedingung kein *then* (wie in BASIC) steht.

Anweisung — Das Element Anweisung kann eine einzelne Anweisung sein oder aus einem Anweisungsblock bestehen. Eine einzelne Anweisung muß immer mit einem Semikolon (dem Anweisungsbegrenzer) beendet werden. Wenn ein Anweisungsblock ausgeführt werden soll, muß dieser in geschweifte Klammern eingeschlossen werden.

Das folgende Beispiel demonstriert eine Möglichkeit der Verzweigung mit *if*. Es verwendet eine vergleichende Operation. Der Inhalt der Variablen `cZeichen` wird mit der Konstanten `'a'` verglichen. Ergibt der Vergleich das Ergebnis 'wahr', wird mit `printf()` ein Text ausgegeben.

```
if (cZeichen == 'a')
    printf("Das Zeichen ist 'a'.");
```

4.1.2 if else

Es gibt auch Fälle, in denen alternativ (also genau dann, wenn die zu *if* gehörende Bedingung das Ergebnis 'unwahr' liefert) zu einem anderen Anweisungsblock verzweigt werden soll. Diese Sprachkonstruktion wird durch das Schlüsselwort *else* eingeleitet.

```
if (Bedingung)
        Anweisung1;
else
        Anweisung2;
```

Im Gegensatz zur Programmiersprache Pascal muß auch nach Anweisung1 vor *else* ein Semikolon stehen.
Dieses Fragment ist eine Erweiterung des ersten Beispiels zu *if*. Wenn der zu *if* gehörende Vergleich das Ergebnis 'unwahr' liefert, wird der zu *else* gehörende Anweisungsblock ausgeführt.

```
if (cZeichen == 'a')
    printf("Das Zeichen ist 'a'.");
else
    printf("Das Zeichen ist nicht 'a', sondern '%c'", cZeichen);
```

4.1.3 else if

Mit den Schlüsselworten *if* und *else* ist es auch möglich, sogenannte else-if-Ketten zu erstellen, bei denen mehrere Ausdrücke überprüft werden können.

```
if (Bedingung1)
        Anweisung1;
else if (Bedingung2)
        Anweisung2;
else if (Bedingung3)
        Anweisung3;
else
        Anweisung4;
```

Das folgende Beispiel realisiert mit Hilfe einer else-if-Kette einen kleinen Taschenrechner. Der Anwender wird zur Eingabe zweier Zahlen und eines Operators aufgefordert, der auf die Zahlen angewendet werden soll. In der folgenden else-if-Kette wird überprüft, ob es sich beim eingegebenen Operator um eine der unterstützten vier Grundrechenarten handelt. Ist dies der Fall, wird zur zugehörenden printf()-Anweisung verzweigt, in der das Ergebnis berechnet und ausgewertet wird.

Konnte der Operator nicht identifiziert werden, gibt das Programm eine entsprechende Meldung aus:

```
#include <stdio.h>

main()
{
float fZahl1, fZahl2;
char cOperator;

printf ("\nEin kleiner Taschenrechner:\n" );
printf ("Geben Sie ein: Zahl Operator Zahl <Return>\n");
scanf ("%f %c %f", &fZahl1, &cOperator, &fZahl2);

if (cOperator == '+')
    printf("= %f ", fZahl1 + fZahl2);
else if (cOperator == '-')
    printf("= %f ", fZahl1 - fZahl2);
else if (cOperator == '*' || cOperator == 'x')
    printf("= %f ", fZahl1 * fZahl2);
else if (cOperator == '/')
    printf("= %f ", fZahl1 / fZahl2);
else
    printf("Operator nicht bekannt!");
printf ("\n\n");
}
```

Auswertung der else-if-Ketten:

- Die Bedingungen werden in der Reihenfolge ausgewertet, in der sie im Programmcode stehen.
- Wenn eine der Bedingungen das Ergebnis 'wahr' liefert, wird der zugehörende Anweisungsteil ausgeführt und damit die Abarbeitung der Kette beendet.
- Die zum letzten *else* gehörenden Anweisungen werden ausgeführt, wenn keine der vorher überprüften Bedingungen das Ergebnis 'wahr' liefert.
- Das letzte *else* ist optional, kann also entfallen, wenn keine Standard-Aktion ausgeführt werden soll.

4.1.4 switch

Eine einfachere und meistens auch übersichtlichere Programmiertechnik, als die im vorigen Abschnitt beschriebenen else-if-Ketten, bietet die *switch*-Anweisung. Auch mit ihr kann ein Programm zwischen mehreren Alternativen auswählen.

```
switch(Ausdruck)
        {
        case Konstante1:
                Anweisung1;
                break;
        case Konstante2:
                Anweisung2;
                break;
        case Konstante3:
                Anweisung3;
                break;
        case Konstante4:
                Anweisung4;
                break;
        default:
                Anweisung5;
        }
```

Ausdruck	Ausdruck ist ein ganzzahliger Wert, der mit allen innerhalb der switch-Anweisung stehenden *case*-Marken verglichen wird. Hier kann auch der Aufruf einer Funktion stehen, die eine Ganzzahl als Ergebnis zurückgibt. Gleichzeitig kann hier, wie bei der *if*-Anweisung eine Zuweisung des Funktionsergebnisses erfolgen.
Konstante	Konstanten, die mit Ausdruck verglichen werden.
Anweisung	Anweisung wird ausgeführt, wenn die zugehörende Konstante und der *switch*-Ausdruck identisch sind. Wenn einmal eine Übereinstimmung gefunden wurde, werden alle Anweisungen (auch die zu anderen Konstanten gehörenden) ausgeführt, bis entweder das Schlüsselwort *break* die *switch*-Anweisung beendet oder der *switch*-Block beendet ist.
default	Zum *default* gehörenden Anweisungsblock wird verzweigt, wenn die Überprüfung des *switch* Ausdrucks mit allen *case*-Marken keine Ubereinstimmung ergibt. Um das Programm Laufzeitsicherer zu machen, sollte hier der dazu entsprechende Code stehen.

Das folgende Beispiel ist eine Überarbeitung des Taschenrechner-Programms, das bei den else-if-Ketten gezeigt wurde. Die Auswertung des Operators erfolgt hier jedoch mit einer *switch*-Anweisung. Zu beachten ist, daß die *default*-Marke die gleiche Funktion erfüllt, wie das letzte *else* beim vorigen Beispiel. Darüber hinaus akzeptiert das Programm weitere Operatoren für die Multiplikation und Division. Dies wurde erreicht, indem beispielsweise bei der *case*-Marke mit der Zeichenkonstanten 'x' das *break* weggelassen wurde, und somit für diesen Fall die gleichen Anweisungen ausgeführt werden wie für die Konstante '*'.

```
#include <stdio.h>

main()
{
float fZahl1, fZahl2;
char cOperator;

printf ("\nEin kleiner Taschenrechner:\n" );
printf ("Geben Sie ein: Zahl Operator Zahl <Return>\n");
scanf ("%f %c %f", &fZahl1, &cOperator, &fZahl2);

switch(cOperator)
    {
    case '+':
        printf("= %f ", fZahl1 + fZahl2);
        break;
    case '-':
        printf("= %f ", fZahl1 - fZahl2);
        break;
    case '*':
    case 'x':
    case 'X':
        printf("= %f ", fZahl1 * fZahl2);
        break;
    case '/':
    case ':':
        printf("= %f ", fZahl1 / fZahl2);
        break;
    default:
        printf("Operator nicht bekannt!");
    }
printf ("\n\n");
}
```

Auswertung der switch-Anweisung:

- Die Konstanten bei den *case*-Marken werden in der Reihenfolge, in der sie im Programmcode stehen, mit Ausdruck verglichen.
- Wenn an einer Stelle Ausdruck und Konstante identisch sind, wird zu der zugehörigen Anweisung verzweigt.
- Ergeben alle Vergleiche das Ergebnis ungleich, wird, falls die *default*-Marke vorhanden ist, zu den Anweisungen hinter *default* verzweigt.
- Da die Konstanten nur als Marken dienen, wird die Ausführung nach gefundener Obereinstimmung so lange fortgesetzt, bis die *switch*-Anweisung zu Ende ist. Darum wird (es sei denn, dieses Durchfallen wird explizit gewünscht), jede zu einem *case* gehörende Anweisung mit dem Schlüsselwort *break* beendet. Trifft der Compiler auf das *break*, wird die *switch*-Anweisung sofort verlassen.
- Innerhalb einer *switch*-Anweisung dürfen keine zwei Konstanten den gleichen Wert haben. Sollte dies doch der Fall sein, erzeugt der Compiler eine Fehlermeldung.
- Zeichenkonstanten werden in *switch*-Anweisungen automatisch in ihren Integerwert umgewandelt.
- *switch*-Anweisungen bzw. die Sprungmarken der Anweisungen nach *case* werden üblicherweise vom Compiler in Sprungtabellen übersetzt, bei C51-Compiler nur dann, wenn dies die Laufzeit verkürzt.

4.2 Schleifen

Oft ist es erforderlich, daß ein kleiner Programmteil in Abhängigkeit von einer oder mehreren Bedingungen mehrfach ausgeführt werden soll. Um Code zu sparen und vor allem der besseren Übersichtlichkeit halber dienen hierzu Programmschleifen, die solange gestartet werden, wie die zugehörige Bedingung erfüllt ist. Man unterscheidet zwischen kopf- und fußgesteuerten Schleifen. Bei ersterer wird die Bedingung vor dem Schleifendurchlauf geprüft, so daß bei Nichterfüllung die Anweisungen innerhalb der Schleife nie ausgeführt werden. Bei fußgesteuerten Schleifen wird in jedem Fall zunächst ein Schleifendurchlauf vollzogen, bevor die Bedingung an Ende der Schleife geprüft wird.

4.2.1 for

Mit dem Schlüsselwort *for* wird eine kopfgesteuerte, bedingte Schleife eingeleitet. Der Schleifenausdruck steht in runden Klammern und enthält die drei Elemente Initialisierung, Bedingung und Veränderung, die durch Semikolon voneinander getrennt werden.

for (Initialisierung; Bedingung; Veränderung)
    Anweisung;

Initialisierung Die im Element Initialisierung stehenden Anweisungen werden ausgeführt, bevor die Schleife zum ersten Mal beginnt. Dort werden üblicherweise die Schleifen- oder Zählvariablen initialisiert. Es ist auch möglich, in dieser Anweisung mehrere Variablen zu initialisieren. Die Zuweisungen werden dann durch Kommata voneinander getrennt (vgl. Kap. 3.8.6).

Bedingung Das Element Bedingung enthält einen Ausdruck, der wahr sein muß, damit der zur *for*-Schleife gehörende Anweisungsteil ausgeführt wird.

Veränderung Die hier stehenden Anweisungen werden nach jedem Durchlauf des zu *for* gehörenden Anweisungsblocks abgearbeitet. Hier wird meist die Zählervariable erhöht oder vermindert. Es kann dort jedoch auch jede andere Anweisung aufgenommen werden. Stehen hier mehrere Anweisungen, werden sie durch Kommata getrennt.

Anweisung Das Element Anweisung kann eine einzelne Anweisung sein oder aus einem Anweisungsblock bestehen. Eine einzelne Anweisung muß immer mit einem Semikolon (dem Anweisungsbegrenzer) beendet werden. Wenn ein Anweisungsblock ausgeführt werden soll, muß dieser in geschweifte Klammern eingeschlossen werden.

Das Beispielprogramm verwendet eine for-Schleife, um den sich verändernden Wert der Schleifenvariable auszugeben.

```
#include <stdio.h>

main()
{
word wZaehler;

for(wZaehler=0; wZaehler<10; wZaehler++)
    printf("\nDer Wert des Schleifenzählers beträgt %u.",wZaehler);
}
```

4.2.2 while

Eine *while*-Schleife, ebenfalls kopfgesteuert, besteht nur noch aus den Elementen "Bedingung" und "Anweisung" im Vergleich zur vorgenannten *for*-Schleife. Das Element "Initialisierung" muß, falls erforderlich, vor Beginn der Schleife und das Element "Veränderung" innerhalb der Schleife am Schluß implementiert werden.

while (Bedingung)
    Anweisung;

Auswertung der *while*-Schleife:
- Die Bedingung der *while*-Schleife wird getestet, bevor die zur Schleife gehörende Anweisung (bzw. Anweisungen in {}) ausgeführt wird.
- Die Schleife endet, wenn die Bedingung nicht mehr den Wert 'wahr' ergibt.
- Liefert die Auswertung der Bedingung der *while*-Schleife bereits beim Eintritt den Wert 'unwahr', werden die zur Schleife gehörenden Anweisungen nie ausgeführt.

Das Beispiel zeigt die gleiche Funktion wie das der *for*-Schleife, nur mit *while* formuliert:

```
#include <stdio.h>

main()
{
word wZaehler;

wZaehler=0;
while(wZaehler<10)
    {
    printf("\nDer Wert des Schleifenzählers beträgt %u.",wZaehler);
    wZaehler++;
    }
}
```

4.2.3 do

Neben der *while*-Schleife existiert in C das Konstrukt der *do*- oder *do-while*-Schleife. Der wichtigste Unterschied zur *while*-Schleife besteht darin daß sie fußgesteuert ist; der Anweisungsteil der *do*-Schleife wird mindestens einmal ausgeführt. Die Bewertung des Ausdrucks, der die Bedingung zum Abbruch der Schleife enthält, findet immer nach dem Durchlaufen des Anweisungsteils statt.

do
    Anweisung;
    while (Bedingung);

Die Bedeutungen der Elemente "Anweisung" und "Bedingung" entsprechen denen der vorgenannten Schleifen.

Das folgende Beispiel verwendet eine *do*-Schleife, in der der Anwender zur Eingabe einer Zahl aufgefordert wird. Mit Eingabe von 99 wird das Programm beendet. Bei der Definition der Variablen `wZahl` muß diese nicht initialisiert werden, weil sie beim Eintritt in die Schleife sowieso einen Wert zugewiesen bekommt, denn die *do*-Schleife wird in jedem Fall durchlaufen.

```
#include <stdio.h>

main()
{
word wZahl;

do
    {
    printf("\nGeben Sie eine Zahl ein. ");
    printf("99 um Programm zu beenden.\n");
    scanf("%u",&wZahl);
    printf("Die eingelesene Zahl war %u.\n",wZahl);
    }while(wZahl!=99);
}
```

Auswertung der *do*-Schleife:
- Die Bedingung der *do*-Schleife wird getestet, nachdem die zur Schleife gehörende Anweisung ausgeführt wurde.
- Die zur *do*-Schleife gehörenden Anweisungen werden mindestens einmal abgearbeitet.
- Die Schleife endet, wenn die Abbruchbedingung erfüllt ist.

4.2.4 Endlosschleifen

Eine Endlosschleife erhält man mit einer Sonderform der *for*-Schleife indem alle drei Elemente der Definition wegfallen und nur die die Elemente trennenden Semikolons stehenbleiben.

```
for(;;)
        Anweisung;
```

Eine Endlosschleife mit *while* läßt sich realisieren, indem abgefragt wird, ob er Wert 1 wahr (also ungleich 0) ist, was naturgemäß der Fall ist:

```
while(1)
        Anweisung;
```

Die gleiche Abfrage kann in der *do*-Schleife erfolgen, um eine Endlosschleife zu implementieren:

```
do
        Anweisung;
        while(1);
```

Um die Bedeutung einer Endlosschleife hervorzuheben ist in Kap. 8.2.6 das Makro *FOREVER* definiert, hinter dem eine Endlosschleife steckt.

Endlosschleifen können, da keine Abbruchbedingung vorhanden ist, nur mit *break*, *return* oder *goto* innerhalb der Anweisung verlassen werden.
Nicht alle Compiler können feststellen, ob eine Anweisung zum Verlassen der Endlosschleife eingebaut ist. Sollte das Programm hängen, dann tut eine Überprüfung in diese Richtung gut.

4.3 Sprunganweisungen

Sprunganweisungen dienen dazu unbedingte Sprünge an eine bestimmte Stelle des Programms auszuführen. Die Anweisungen *break* und *continue* können sich nur auf passende Kontrollstrukturen beziehen wohingegen *return* und *goto* auf Funktionsebene gültig sind. Darüber hinaus ist keine Möglichkeit gegeben durch eine Sprunganweisung im Programm wild umherzuspringen wie es beispielsweise unter BASIC möglich ist. Durch das Konzept der Sprache C (Strukturierung durch Hierarchie und Funktionen) geht dahingehend der Überblick nicht verloren.

4.3.1 break

Das Schlüsselwort *break* ist bei der Behandlung der *switch*-Anweisung bereits vorgestellt worden. Neben der Beendigung der Programmausführung bei den einzelnen *case*-Marken kann man den Ablauf eines Programms mit *break* abrupt ändern. Dies ist beispielsweise bei verschachtelten Schleifen sinnvoll, bei denen die Abarbeitung einer Schleife abgebrochen werden soll. Hierbei wird immer nur die innerste Kontrollstrukur verlassen, auf die die *break*-Anweisung paßt.
Im folgenden Beispiel wird durch die *if*-Abfrage und das darin enthaltene *break* die Abarbeitung der *for*-Schleife beendet, wenn die ESC-Taste gedrückt wird.

```
#include <stdio.h>

main()
{
word wZahl;

for(wZahl=3; wZahl<15; wZahl++)
    {
    printf("Zahl hat folgenden Wert: %u\n", wZahl);
    printf("Bitte Taste drücken (ESC=Ende)\n");
    if(getch()==27)
        break;
    }
}
```

In C gibt es keine Kontrollstruktur, die es möglich macht von einer Vielzahl von Anweisungen nur den ersten Teil auszuführen. Der Rest soll - meist abhängig von einer Bedingung - mit einem *break* übersprungen werden. Da ich eine solche Kontrollstruktur schon häufig vermißt habe, brütete ich sie aus und vergab ihr den Namen *DO_ONCE* (Kap. 8.2.6).

```
Anweisung1;             /* wird immer ausgeführt */
DO_ONCE
        {
        Anweisung2;  /* wird immer ausgeführt */
        if (Bedingung1)
                {
                Anweisung3;
                break;
                }
        weitere Anweisungen;
        }
```

Trifft Bedingung1 zu, dann wird alles übersprungen, was ab der break-Anweisung bis zum Abschluß des Blocks DO_ONCE steht.

4.3.2 continue

Im Gegensatz zu *break* wird mit der *continue*-Anweisung die Schleife nicht abgebrochen, sondern *continue* verzweigt zum Anfang der Schleife zurück. Die Ausführung aller nach *continue* stehenden Anweisungen findet also nicht statt. Auch hier gilt, daß *continue* sich nur auf die innerste darauf passende Schleife bezieht. Bei der Anwendung von *continue* auf *for*-Schleifen wird die Anweisung des Elementes "Veränderung" dennoch ausgeführt, auch wenn sie gedanklich als letzte Anweisung der Schleife verstanden werden kann. Auf *switch* paßt *continue* nicht.

Im Beispielprogramm wird der Benutzer aufgefordert, eine Zahl einzugeben. Ist die Zahl negativ (kleiner als Null), wird die eingelesene Zahl nicht ausgegeben, sondern mit *continue* wieder zum Anfang der *do*-Schleife verzweigt. *continue* wird hier zur Filterung der Zahlen, die ausgegeben werden, eingesetzt.

```
#include <stdio.h>

main()
{
int nZahl;
do
    {
    printf("\nGeben Sie eine Zahl ein. ");
    printf("99 um Programm zu beenden.\n");
    scanf("%d",&nZahl);
    if(nZahl<0)
        continue;
    printf("Die eingelesene Zahl war %d.\n",nZahl);
    }while(nZahl!=99);
}
```

4.3.3 return

Mit dem Schlüsselwort *return* kann die Ausführung einer Funktion sofort beendet werden. *return* ist damit auch hervorragend geeignet, um Endlosschleifen zu verlassen. Die genaue Syntax bei der Verwendung von *return* hängt vom Prototyp der Funktion ab. Hat sie einen Rückgabewert, so muß bei Anwendung von *return* auch ein entsprechend auswertbarer Wert zurückgeliefert werden, da sonst bei Auswertung des Rückgabewertes das Programm außer Kontrolle geraten kann. Üblicherweise warnt der Compiler vor einer *return*-Anweisung, die nicht auf den Funktionsprototypen paßt.
Im folgenden Beispiel wird durch die *if*-Abfrage und das darin enthaltene *return* sowohl die Abarbeitung der *for*-Schleife als auch die Funktion `main()` und somit das Programm beendet, sobald die ESC-Taste gedrückt wird.

```
#include <stdio.h>

main()
{
word wZahl;
for(wZahl=3; wZahl<15; wZahl++)
    {
    printf("Zahl hat folgenden Wert: %u\n", wZahl);
    printf("Bitte Taste drücken (ESC=Ende)\n");
    if(getch()==27)
        return;
    }
}
```

4.3.4 goto

Bereits aus der ersten C-Sprachdefinition von Kernighan und Ritchie stammt die Möglichkeit der unbedingten Verzweigung mit *goto*, auch wenn die Aufnahme dieses Schlüsselwortes eigentlich der Philosophie des modularen, strukturierten Programmierens widerspricht. Man sollte, auch wenn die Verführung groß ist, *goto* nicht verwenden. Der ausgiebige Gebrauch von *goto* führt, wie dies auch bei den ersten BASIC-Interpretern der Fall war, zu beinahe unüberschaubaren Programmen, deren Wartung, wegen der nach kurzer Zeit nicht mehr zu überblickenden Programmlogik, schier unmöglich ist.

goto Label;

Mit *goto* kann zu einem beliebigen anderen Punkt innerhalb der aktuellen Funktion verzweigt werden, der durch eine Sprungmarke gekennzeichnet wurde. Die Sprungmarke erhält einen Namen und wird, wie die Marken bei der *switch*-Anweisung, mit einem Doppelpunkt abgeschlossen.

Die einzige sinnvolle Anwendung von *goto*, die mir untergekommen ist, war der Ausstieg aus einem Auswahlmenü, welches aus verschachtelten *for*-Schleifen, *if*- und *switch*-Anweisungen bestand. Da nach dem Verlassen des Menüs die Abarbeitung von Abschlußaufgaben notwendig war, konnte keine *return*-Anweisung eingesetzt werden. Ein Auszug hieraus als Beispiel:

```
void menu(void)
{
bool bMenuAufbauen = TRUE;
while(1)
    {
    if(bMenuAufbauen)
        {
        bMenuAufbauen = FALSE;
        ClearDisplay();
        SetKeyRate(1 SEK);
        switch(getch())
            {
            case MENU_ALARM:
                WriteText(ALARM_TEXT);
                break;
            case MENU_UHR_STELLEN:
                UhrStellen();
                break;
            case MENU_ENDE:
                WriteText(ENDE_TEXT | CLEAR);
                goto MenuEnde;
            default:
                bMenuAufbauen = TRUE;
                break;
            }
        }
    }

MenuEnde:
SetKeyRate(0);
RestoreDisplay();
return;
}
```

5 Funktionen

Die Programmiersprache C hat einen reichhaltigen Satz an Operatoren, Schlüsselwörtern und Kontrollstrukturen um gute Programme möglich zu machen. Den Begriff "Befehl", wie er in andere Sprachen existiert, gibt es in C eigentlich nicht. Alle (allgemeiner gesagt) Aufrufe werden in C durch Funktionen realisiert, die als Unterprogramme aufgerufen werden. Diese Eigenschaft zwingt zu einer Strukturierung, die Sprünge quer durch das Programm (wie z.B. bei BASIC oder Assembler) unmöglich machen.
Den Erfordernissen entsprechend gibt es unterschiedliche Arten von Funktionen, die bis auf die "einfachen Funktionen" jeweils Spracherweiterungen sind.

5.1 Einfache Funktionen

Mit der Definition einer Funktion wird die Speicherklasse, der Typ und Name, die formalen Parameter sowie der Körper einer Funktion festgelegt. Durch die Definition einer Funktion wird Code erzeugt, da hier die eigentlichen Anweisungen angegeben werden. Bei der syntaktischen Gestaltung einer Funktionsdefinition kann man zwischen dem klassischen Stil, wie er beispielsweise in der ersten Ausgabe von "The C Programming Language" durch Kernighan und Ritchie verwendet wurde, und dem ANSI-Standard unterscheiden. Hier soll auf die Erläuterung der klassischen Form der Funktionsdefinition verzichtet und dafür die Funktionsdefinition nach ANSI C gezeigt werden, da dieses Muster auch sicherlich das Programmierverfahren der Zukunft bleibt.

[Speicherklasse] [Typ] Name ([formale Argumente])
{
Funktionskörper
}

Beispiel:
```
int Fehlermeldung(char *szHauptFehler, char *szZusatzInfo)
{
int nError;           /* lokale Variable */
/* Anweisungen */
return nError;         /* Rücklieferwert */
}
```

Speicherklasse
Das Element Speicherklasse ist optional. Dort können die Schlüsselworte *extern* und *static* eingesetzt werden, mit der die Sichtbarkeit einer Funktion festgelegt wird. Wenn die Speicherklasse nicht angegeben wird, besitzt die Funktion automatisch die Speicherklasse *extern*. Hat eine Funktion die Speicherklasse *extern*, kann sie von jeder Datei heraus aufgerufen werden. Ist die Speicherklasse *static*, kann die Funktion nur innerhalb der Datei, in der sie definiert wurde, aufgerufen werden, sie ist also nur für das Modul sichtbar.

Typ
Mit der Typangabe wird festgelegt, welchen Rückgabewert die Funktion hat. Hier kann jeder Typbezeichner (auch die von selbst definierten Typen) angegeben werden. Fehlt die Typangabe, geht der Compiler davon aus, daß die Funktion ein *int* zurückgibt. Funktionen, die keinen Wert zurückgeben, erkennt man an der Typangabe *void*. In C51 erfolgt die Rückgabe eines Wertes abhängig vom Datentyp standardmäßig in Registern:

Rückgabewert	Register	Bemerkung
bit	Carry-Flag	
(unsigned) char, 1-Byte-Zeiger	R7	
(unsigned) Int, 2-Byte-Zeiger	R6, R7	MSB in R6, LSB in, R7
(unsigned) long	R4 bis R7	MSB in R4, LSB in R7
float	R4 bis R7	32-Bit IEEE-Format
generic pointer	R1, R2, R3	Selector in R3, MSB R2, LSB R1

Name
Der Name einer Funktion kann die Zeichen gem. Kapitel 1.2.5 (Erlaubte Zeichen in Bezeichnern) enthalten. Er sollte so formuliert sein, daß die "Funktion" der Funktion zu erkennen ist.

Formale Argumente
Wenn die Funktion Argumente erhält, wird der Typ des Arguments und sein Name angegeben. Mehrere Argumente werden durch Kommata getrennt. Als einzige Speicherklasse für Parameter kann *register* angegeben werden. Dadurch wird beim Aufruf der Funktion, sofern möglich, das so deklarierte Argument in ein Prozessorregister geladen. Die Übergabe der Parameter erfolgt unter C grundsätzlich nach dem Verfahren "call by value", d.h. es wird immer eine Kopie des Übergabewertes übergeben.
Die Platzierung der Parameter im Speicher ist aufgrund der Rechnerarchitektur von der Compiler-Implementierung abhängig. Beim PC beispielsweise werden die Parameter auf dem Stack übergeben.
In C51 werden die Register herangezogen, was eine entsprechende Speicherklassenangabe überflüssig macht. Auf diese Weise können bei C51 bis zu drei Parameter in CPU-Registern übergeben werden. Diese Technik der Parameterübergabe führt zu effizientem Code, der mit Assembler-Programmierung vergleichbar ist. Ist für einen Parameter kein Register mehr verfügbar oder ist *#pragma NOREGPARMS* eingestellt, so erfolgt die Parameterübergabe in festen Speicherbereichen. Der Adressraum, der für die Parameterübergabe verwendet wird, ist vom Speichermodell abhängig.
Register-Möglichkeiten für Parameterübergabe in C51:

Parametertyp:	*char, 1-Byte-ptr*	*int, 2-Byte-ptr*	*long, float*	*generic-ptr*
1. Parameter	R7	R6, R7	R4 bis R7	R1, R2, R3
2. Parameter	R5	R4, R5	R4 bis R7	R1, R2, R3
3. Parameter	R3	R2, R3	----	R1, R2, R3

Wird ein Parameter des Typs *bit* übergeben, so erfolgt die Übergabe dieses und aller folgenden Parameter unabhängig von der Tabelle in festen Speicherbereichen.

Wird ein Zeiger (Kap. 2.12) als Argument übergeben, so ist eine Möglichkeit geschaffen, einen Wert "zurückzuliefern", der kein Rückgabewert ist. Da der Zeiger eine Adresse ist, kennt die Funktion die Speicherstelle, an der die Variable steht, die ihr übergeben wird. Sie kann mit dem Indirektions-Operator den Wert an dieser Stelle verändern. Der Vorteil bei diesem Verfahren besteht darin, daß es möglich ist, mehrere Werte von einer Funktion berechnen und verändern zu lassen. Diese Art der Parameterübergabe entspricht dem Verfahren "call by reference".
Die Reihenfolge, in der die Funktionsargumente ausgewertet werden, ist nicht festgelegt und ist somit von der Compiler-Implementierung abhängig.
Wenn der Funktion keine Argumente übergeben werden, steht an dieser Stelle das Schlüsselwort *void*.

Beispiel:
```
int nPosition;
nPosition=3;
Teste(nPosition++, nPosition);
```

Der Aufruf der Funktion `Teste()` kann bei verschiedenen Compilern zu anderen Resultaten führen. Die Frage ist, ob die Inkrementierung der Variablen `nPosition` im ersten stattfindet, bevor oder nachdem das zweite Argument an die Funktion übergeben wurde. Eine Möglichkeit, diese Nebeneffekte zu vermeiden, besteht darin, den Code aufzuschlüsseln:

```
int nPosition, nTemp;
nPosition=3;
nTemp=nPosition++;
Teste(nTemp, nPosition);
```

Funktionskörper
Der Funktionskörper ist ein Anweisungsblock, der in geschweifte Klammern eingeschlossen wird. Innerhalb des Funktionskörpers können eigene (lokale) Variablen definiert werden, die, wenn nicht anders angegeben, die Speicherklasse *auto* besitzen, womit deren Lebensdauer nur auf den Aufruf dieser und tiefer verschachtelten Funktionen begrenzt ist. Soll der Wert einer Variablen auch nach dem Verlassen der Funktion, und damit bis zum nächsten Aufruf erhalten bleiben, muß für lokale Variablen die Speicherklasse *static* benutzt werden.
Die Platzierung von Variablen der Speicherklasse *auto* ist ebenfalls aufgrund der Rechnerarchitektur implementierungsabhängig. So werden sie beim Funktionsaufruf auf einem PC auf dem Stack erzeugt und beim Verlassen der Funktion wieder entfernt. In C51 hingegen werden der Geschwindigkeitsoptimierung und Speicherplatzeinsparung halber Register oder feste Speicherbereiche dafür belegt. Hier sei das Stichwort Overlaytechnik genannt, was im Kapitel 9.4.1 näher erläutert wird.

5.2 Funktionsdeklarationen und Prototypen

Das Konzept der Prototypen wurde durch den ANSI-C-Standard offizieller Bestandteil der Sprache C. Vor dem ANSI-Standard wurden für den gleichen Zweck meist Funktionsdeklarationen verwendet, die dem Compiler nur einen Teil der von Prototypen zur Verfügung gestellten Informationen bereitstellt. Die klassische Deklarationsmethode sollte jedoch nicht mehr verwendet werden und erscheint deswegen hier nicht mehr. Grundlage beider Methoden ist die, daß der Funktionskörper durch das Semikolon ersetzt wird.

Prototypen enthalten die Deklaration einer Funktion, bei der der Rückgabewert, die Typen, die Anzahl und die Reihenfolge der Argumente festgelegt werden. Der Prototyp einer Funktion ist mit dem Kopf der Funktionsdefinition identisch. Der Unterschied besteht lediglich darin, daß der Prototyp mit einem Semikolon abgeschlossen wird, der quasi den Funktionskörper ersetzt. Durch die Verwendung von Prototypen kann der Compiler bei der Code-Generierung überprüfen, ob Anzahl und Typ der aktuellen Argumente (also die beim Aufruf der Funktion benutzten), mit der Deklaration übereinstimmen. Prototypen werden idealerweise in die zum Modul gehörende Headerdatei aufgenommen. Wenn die Headerdatei mit der #include-Direktive in eine andere Quelldatei aufgenommen wird, können die Funktionsaufrufe mit den formalen Argumenten verglichen werden. Dies ist auch das übliche Verfahren bei der Verwendung der Funktionen aus der Laufzeitbibliothek.

Implizite Deklaration durch Aufrufe
Der Compiler weiß sich zu helfen, wenn er keine Deklaration oder keinen Prototyp entdeckt hat, bevor er auf den Aufruf einer Funktion stößt. In diesem Fall geht der Compiler davon aus, daß die Funktion einen *int*-Wert zurückgibt und verwendet den Typ der aktuellen Argumente, um die Deklaration der formalen Argumente zu bilden. Problematisch ist dieses Verfahren jedoch deshalb, weil der Aufruf dieser Funktion mit falschen Argumenttypen auch eine unkorrekt erzeugte Deklaration zur Folge haben kann.

5.3 Assembler Interface

Unter C51 erfolgt die Parameter-Übergabe zu Assembler-Funktionen in CPU-Registern wie unter "Einfache Funktionen" beschrieben, oder bei Anwendung von *#pragma NOREGPARMS* in festen Speicherbereichen. Daher ist das Interface zu Assembler sehr einfach und übersichtlich. Die Rückgabe von Funktionswerten erfolgt ebenfalls wie unter "Einfache Funktionen" beschrieben in CPU-Registern. Ein Funktionsprototyp der Assembler-Funktion, wie sie in C verstanden wird sollte existieren. Andere Assembler-Schnittstellen (die erfahrungsgemäß nicht nötig sind) können im Compiler-Handbuch nachgelesen werden.

5.4 C51-Reentrant-Funktionen

Die Übergabe der Parameter (=lokale Variable *auto*) bzw. Rücklieferung von Werten in Registern sowie das Verfahren der Overlaytechnik macht es unter C51 unmöglich Funktionen rekursiv aufzurufen.
Auch Aufrufe von Interrupt-Routinen oder Echtzeit-Anwendungen sind durch diese Technik verwehrt, wenn mehrere Tasks in unterschiedlichen Prioritäten die gleiche Funktion aufrufen und diese seine lokalen Variablen nicht ausschließlich in Registern hält.
Um jedoch dem ANSI-Standard in dieser Hinsicht Rechnung zu tragen wurde eine Möglichkeit mit dem Schlüsselwort *reentrant* geschaffen.
Solche Funktionen können selektiv als reentrant-fähig (wiedereintrittsfähig) definiert werden. Je nach Speichermodell wird für Reentrant-Funktionen ein Stack-Bereich im internen oder externen Speicher nachgebildet. Eine Stack-Archikektur ist beim 8051 mangels geeigneter Adressierungsarten relativ aufwendig - reentrant Funktionen sollten daher sparsam eingesetzt werden. Die Stack-Nachbildung gewährleistet aber auch eine problemlose Einbindung in vorhandene Echtzeit-Betriebssysteme (nicht unbedingt gleichzusetzen mit schnellen Systemen) und eine zuverlässige Arbeitsweise der Software.

Beispiel einer Reentrant-Funktion:

```
int Calc (byte byIdx, int nWert) reentrant
{
int nI;
nI=anTABELLE[byIdx];
return nI * nWert;
}
```

An dieser Stelle möchte ich anmerken, daß seit ich u.a. in der Welt der Programmierer tätig bin, niemals von der Notwendigkeit eines Rekursivaufrufes oder einer Reentrant-Funktion Gebrauch gemacht habe. Zur Vermeidung eines versehentlich wiederholten Funktionsaufrufs während die Funktion selbst noch aktiv ist (z.B. Interrupt-Routine) kann man funktionsintern ein statisches Flag setzen, daß zur sofortigen Rückkehr zwingt, wenn es gesetzt ist. Das Flag wird bei gültgem Aufruf gesetzt und vor dem Verlassen wieder gelöscht.

5.5 C51-Interrupt-Funktionen

Die Progammierung eines Mikrocontrollers ohne Interrupts ist kaum denkbar. Die wohl am häufigsten verwendeten Interrupts sind die der seriellen Schnittstelle, eines Timers oder einer Reaktion auf ein Ereignis von außen über einen der Port-Pins.
C51 gestattet es, Interrupt-Routinen mittels des Schlüsselwortes *interrupt* direkt in C zu schreiben. Über das Schlüsselwort *using* kann auch die Registerbank angegeben werden, die für die Interrupt-Routine verwendet werden soll. Zur Laufzeitoptimierung ist die Verwendung einer anderen Registerbank immer dann sinnvoll, wenn eine Funktion in der Lage ist eine andere zu unterbrechen. Dies gilt allgemein, für Interrupts, und wiederum für solche höherer Priorität, womit Funktionen gleicher Priorität auch die gleiche Registerbank verwenden können.
Der Compiler generiert den erforderlichen Code für die Registerbank-Umschaltung und den dazugehörigen Interrupt-Vektor.
Beim Funktions-Eintritt werden die Inhalte der Special-Funtion-Register ACC, B, DPH, DPL und PSW (im Bedarfsfall) auf dem Stack gesichert.
Wird ohne Registerbank-Umschaltung (using-Attribut) gearbeitet, so werden auch alle in der Interrupt-Funktion benötigten Arbeitsregister (Rn) auf dem Stack abgelegt.
Vor dem Verlassen der Funktion werden alle auf dem Stack abgelegten Registerinhalte wieder hergestellt.
Die Funktion wird mit dem Maschinenbefehl *RETI* (Return from Interrupt) verlassen.

Beispiel einer Interruptfunktion:

```
sbit LED = P1^0;
#define MS

void Timer0(void) interrupt 1 using 2
{
static word wTimer0=0;

TR0=0;
LOAD_TIMER(TH0,TL0,US(1000));    /* Makro aus Kap. 8.2.2 */
TR0=1;
if(wTimer0)
    if(--wTimer0)
        return;
LED=!LED;
wTimer0=500 MS;                  /* LED mit Blinkfrequenz von 1Hz */
}
```

Für C51-Interrupt-Funktionen gelten folgenden Regeln:

- Die Übergabe von Parametern ist nicht möglich. Der Compiler erzeugt eine Fehlermeldung, wenn eine Interrupt-Funktion Parameter-Deklarationen enthält.
- Die Rückgabe eines Werts ist nicht möglich. Der Compiler toleriert die Angabe eines Integer-Rückgabewerts, weil der Typ optional und damit für den Compiler nicht ersichtlich ist. Andere Typen als *int* erzeugen eine Fehlermeldung.
- Der Compiler erkennt direkte Aufrufe von Interrupt-Routinen und weist diese ab. Es ist nicht sinnvoll, Interrupt-Routinen direkt aufzurufen, weil das Verlassen mit dem Befehl RETI durchgeführt wird. RETI wirkt auf das Hardware-Interruptsystem des 8051-Controllers, obwohl keine Interrupt Anforderung seitens der Hardware vorgelegen hat. Die Überprüfung wird unterlaufen, wenn statt des direkten ein indirekter Aufruf über einen Zeiger durchgeführt wird. Ein korrekter Ablauf kann nur dann geleistet werden, wenn der Interrupt z.B. per Software (setzen des entsprechenden Flags) ausgelöst wird.
- Der Compiler generiert einen Interrupt-Vektor an der absoluten Adresse *8*n+3*, wobei *n* die angegebene Interruptnummer ist. Der Vektor beinhaltet einen Sprung zur Interrupt-Funktion, was für unser `Timer0()`-Beispiel die Adresse 000Bh ist. Die Generierung des Vektors kann durch die Compileranweisung *NOINTVECTOR* unterbunden werden, womit der Anwender die Möglichkeit hat, Interrupt-Vektoren in separaten Assemblermodulen bereitzustellen.
- Der C51 Compiler gestattet Interruptnummern im Bereich 0 bis 31. Welche Interrupts tatsächlich zulässig sind, hängt vom verwendeten 8051-Derivat ab und wird vom Compiler nicht überprüft.
- Werden in der Interrupt-Routine *float*-Operationen ausgeführt, so muß bis einschließlich der Compilerversion C51-V4.x der Zustand der Floating-Point-Routinen gesichert werden. Diese Sicherung ist nicht erforderlich, wenn kein anderer Programmteil float-Operationen ausführt. Zur Sicherung des Floating-Point-Zustands sind die Funktionen *fpsave()* und *fprestore()* der Laufzeit-Bibliothek *MATH* auszuführen. Dessen Prototypen und Datenstrukturen sind in der Datei MATH.H deklariert .
- Funktionen, die von einer Interrupt-Funktion aufgerufen werden, müssen mit der gleichen Registerbank wie die Interrupt-Funktion arbeiten. Es entsteht eine Fehlfunktion, wenn der aufgerufenen Funktion eine andere als die eingestellte Registerbank zugeordnet ist. Da der Compiler dies nicht überprüfen kann, wird bei nicht passender Registerbank auch keine Fehlermeldung erzeugt. Ist generell die Compileranweisung *NOARGES* eingestellt, gibt es keine Probleme, sofern alle lokalen Variablen der aufgerufenen Funktion in Registern untergebracht sind (vgl. Kap. 6.6.10).

5.6 Intrinsic-Funktionen

Die Funktionen aus nachfolgender Liste werden von C51 als Intrinsic-Funktionen kodiert. Intrinsic-Funktionen sind reentrant-fähig und sehr effizient. Die meisten Intrinsic-Funktionen werden ohne Unterprogramm-Aufruf, direkt als Inline-Code erzeugt. Sie sind somit keine echten Funktionen, auch wenn sie von der Deklaration her so aussehen.

Funktionsname	Beschreibung
memcpy, memset, memchr, memmove, memcmp	ANSI 'memory'-Funktionen.
strcmp, strcpy	ANSI 'string'-Funktionen.
*_crol_, _irol_, _lrol_*	Rotate Left char, int, long.
*_cror_, _iror_, _lror_*	Rotate Right char, int, long.
*_nop_*	No Operation, (NOP-Befehl).
*_testbit_*	Test and Clear Bit (JBC-Befehl).

5.7 Inline-Assembler

Unter C51 besteht die Möglichkeit auch direkt in mnemonischer Maschinensprache zu programmieren. Dies ist sinnvoll z.B. bei besonders zeitkritischen (µs-Bereich) Programmteilen, oder solchen, für die das C-Pendant kompliziert ist (z.B. RRC).
Das Modul, in dem Inline-Assembler-Anweisungen stehen, kann nur in zwei Arbeitsgängen übersetzt werden. Der Compiler generiert im ersten Schritt einen Assembler-Quellcode, der dann im zweiten Schritt mit dem Assembler A51 zu einer Objekt-Datei zu übersetzen ist.
Für den ersten Arbeitsgang muß zu Beginn des Moduls die Compileranweisung *#pragma SRC* stehen. Dann kann innerhalb einer Funktion des Moduls so oft wie nötig mit *#pragma ASM* die mnemonische Schreibweise eingeleitet, und mit *#pragma ENDASM* wieder beendet werden. Beispiel:

```
#pragma SRC

byte Crc(byte byDat, byte byOldCrc)
{        /* Liefert schnell eine CRC-Checksumme aus OldCrc und Dat  */
B=8;     /* Schleifenzähler */
ACC=byDat;
#pragma ASM
CRC_LOOP:       XRL     A,R5            ; byOldCrc liegt in R5
                RRC     A
                MOV     A,R5
                JNC     ZERO
                XRL     A,#18H
ZERO:           RRC     A
                MOV     R5,A
                MOV     A,R7            ; rotate byDat
                RR      A
                MOV     R7,A
                DJNZ    B,CRC_LOOP
#pragma ENDASM
return byOldCrc;
}
```

6 Der Präprozessor

Neben der reinen Sprachdefinition beschreibt der ANSI-C-Standard unter anderem auch die Phasen, die erfolgen müssen, damit der Quellcode eines C-Programms in den von der Maschine ausführbaren Code übersetzt werden kann. Eines der beim Übersetzungsvorgang beteiligten Programme ist der sogenannte Präprozessor, der (der Name läßt es bereits vermuten) vorbereitende Arbeiten an dem Quellcode durchführt. Erst nachdem der Präprozessor den Code bearbeitet hat, wird er zur eigentlichen Code-Generierung an den Compiler weitergeleitet. Der Präprozessor führt u.a. folgende Arbeiten durch:

- Jedes Backslash-Zeichen '\', das vor einem Zeichen für Zeilenvorschub steht, wird zusammen mit dem Zeilenvorschubzeichen gelöscht, wodurch logisch zusammengehörende Zeilen in eine physikalische Zeile gebracht werden. Jede Zeile einer Quelldatei muß nach dieser Bearbeitung mit einem Zeichen für Zeilenvorschub enden, dem kein Backslash voranstehen darf.
- Der so erhaltene Quellcode wird in einzelne Token (Grundsymbole) aufgeteilt, die durch sogenannte Zwischenraumzeichen (Leerzeichen, Tabulator) getrennt sind.

Diese einzelnen Token werden nun weiterverarbeitet. Drei Aufgaben werden dabei erledigt:

- Textersatz und Makroerweiterung,
- Einfügen von Texten aus anderen Dateien in die Quelldatei,
- Ausschluß bzw. Aufnahme von bestimmten Teilen des Codes (bedingte Kompilierung).

Hierbei sollte man besonders bei intensiver Makrobenutzung darauf achten, daß die durch Textersatz und Makroerweiterung entstandene Zeile nicht die maximal zulässige Zeilenlänge überschreitet. Andernfalls meldet der Compiler Fehler, die man sich nicht erklären kann. Unter C51 kann man in einem solchen Fall die expandierte Zeile in der vom Präprozessor erzeugten Datei QUELLE.I daraufhin überprüfen (vgl. Kap. 9.3).

Präprozessor-Direktiven

Die Ausführung der zuletzt beschriebenen Aufgaben kann vom Programmierer durch die sogenannten Präprozessor-Direktiven beeinflußt werden. Die Anweisungen an den Präprozessor beginnen im Quellcode mit dem Zeichen "#". Dieses Zeichen kann am Anfang der Zeile stehen, es können ihm jedoch auch Zwischenraumzeichen voranstehen. Die nachfolgende Tabelle enthält alle Präprozessor-Direktiven, die der ANSI-C-Standard vorschreibt. Alle Direktiven werden vom C51-Compiler unterstützt.

defined	Testet, ob Makro definiert ist.
#	Operator zur Zeichenkettenbildung.
##	Operator zur Grundsymbolverbindung.
#define	Definiert ein Symbol oder Makro.
#elif	else-if-Operator.
#else	else-Operator.
#endif	Ende der #if-Direktive.
#error	Erzeugt eine Fehlermeldung.
#if	if-Operator.
#ifdef	Äquivalent zu *#if defined*.
#ifndef	Äquivalent zu *#if !defined*.
#include	Datei einbinden.
#line	Aktuelle Zeile verändern.
#pragma	Anweisung für den Compiler.
#undef	Definition eines Symbols entfernen.

Vordefinierte Symbole

Über die Präprozessor-Direktiven hinaus sind bestimmte Makros vom Compiler vordefiniert. Diese Makros müssen mit einem Unterstrich beginnen, dem Großbuchstaben oder ein weiterer Unterstrich folgt. Die Definition dieser Symbole kann mit der Direktive *#undef* nicht entfernt werden. Die nachfolgende Tabelle informiert über die von ANSI C vorgeschriebenen Symbole.

__FILE__	Name der Quelldatei.
__LINE__	Aktuelle Zeile in der Quelldatei.
__DATE__	Datum der Kompilierung der Datei.
__TIME__	Zeit der Kompilierung der Datei.
__STDC__	Compiler entspricht ANSI-C-Standard.
für C51 zusätzlich:	
__C51__	Versionsnummer des Compilers: z.B. 510 für V5.10
__MODELL__	eingestelltes Speichermodell: 0=SMALL, 1=COMPACT, 2=LARGE

6.1 Dateien einbinden mit #include

Die für den Programmierer wichtigste Direktive an den Präprozessor ist sicherlich die Direktive *#include*. Mit ihr wird der Präprozessor angewiesen, die als Argument angegebene Datei in die derzeit bearbeitete Datei aufzunehmen. Meist wird hiermit eine sogenannte Headerdatei aufgenommen, die Funktions-Prototypen, symbolische Konstanten, Makros und/oder Strukturdefinitionen enthält. Das Einbinden der Datei kann an jeder beliebigen Stelle der Quelldatei erfolgen. Es existiert jedoch eine stillschweigende Übereinkunft darüber, daß das Einbinden ziemlich am Anfang der Datei stattfindet. Die Verwendung von Headerdateien empfiehlt sich besonders für Programme, die aus mehreren Quelldateien bestehen. Die gemeinsam benutzbaren Strukturdefinitionen und Konstanten, die Prototypen der externen Funktionen, können somit durch eine einzeilige Anweisung allen beteiligten Dateien bekannt gemacht werden. Dadurch wird eine besonders unangenehme Fehlerquelle vermieden. Grundsätzlich sind zwei verschiedene Formen der *#include*-Direktive möglich:

#include "Datei"

Der Compiler sucht nach der angegebenen Datei zuerst im aktuellen Verzeichnis. Wenn sie dort nicht gefunden wird, oder wenn

#include <Datei>

verwendet wird, dann hängt die weitere Suche vom verwendeten Compiler ab.

Findet C51 die Datei nicht im aktuellen Verzeichnis, dann erzeugt die Dateiangabe in Doppelanführungszeichen eine Fehlermeldung. Anderfalls wird sie in den über die Umgebungsvariable *C51INC* (bzw. *:INCLUDE:* bei älteren Compilerversionen) Verzeichnissen gesucht. Wird die Datei dort ebenfalls nicht gefunden, dann führt dies zu einer Fehlermeldung.

Jede dieser beiden Formen kann auch in ein Makro verpackt sein:
Beispiel:

```
#define TASTEN_DEF "ta_flach.h"

#include TASTEN_DEF
```

Dieses Makro wiederum kann über eine bedingte Compilierung gesetzt sein.

6.2 Makro definieren mit #define

Mit der Präprozessor-Direktive *#define* wird die Definition eines Makros eingeleitet. Eine Makrodefinition besitzt die folgende Form:

#define MakroName ZuErsetzenderText

Der Präprozessor ersetzt, nachdem er die Makrodefinition zur Kenntnis genommen hat, jedes Vorkommen von "MakroName" mit "ZuErsetzenderText". Der Makroname beginnt nach dem Zwischenraumzeichen nach dem Wort *define* und hört mit dem ersten Zwischenraumzeichen nach den Namen auf. Der Name eines Makros wird nach den gleichen Regeln wie ein Variablenname gebildet. Es hat sich jedoch eingebürgert, Makronamen in Großbuchstaben zu schreiben. Diese Nomenklatur, wird sie konsequent eingehalten, macht auf den ersten Blick deutlich, ob es sich um einen Variablen-/Funktionsnamen oder ein Makro handelt (vgl. Kap. 7.6.1)
Die Festlegung des zu ersetzenden Textes hört am Ende der Zeile auf. Um auch Makros schreiben zu können, die lang und komplex sind, ist es möglich, vor der Zeilenschaltung das Zeichen Backslash "\" einzufügen, und die Definition des Makros auf der bzw. den folgenden Zeilen fortzuführen.

Makro ohne Ersatztext

Es ist auch möglich, Makros zu definieren, bei denen der Ersatztext aus Leerzeichen besteht. Mit dem Präprozessor-Operator *defined* und den Direktiven zur bedingten Kompilierung kann getestet werden, ob die Konstante definiert ist oder nicht. Daß dies oft ausreicht, zeigt das nachstehende Beispiel:

Das Codefragment enthält eine benutzerdefinierte Funktionen `Funktion1()`, die Bestandteil eines größeren Programms aus mehreren Modulen ist. Vor der Aufnahme in die Programmliste soll die Funktion getestet werden. Aus diesem Grunde existiert in der `main()`-Funktion ein Aufruf, mit der die Funktion getestet werden kann. Um den Testaufruf in der Datei belassen zu können (vielleicht wird die Funktion eines Tages modifiziert oder erweitert), wird der Test-Teil in `main()` mit den Direktiven zur bedingten Kompilierung eingeschlossen. Nur wenn das Symbol `DEBUG` definiert ist, wird der Test-Teil in `main()` mit kompiliert.

```
#define DEBUG

int Funktion1(void)
{
... /* Code für Funktion1 */
}

main()
{
#ifdef DEBUG     /* gleichbedeutend mit #if defined DEBUG */
/* Testaufrufe für Funktion1 */
#endif
... /* restlicher Code für main() */
}
```

Die Konstante `DEBUG` kann (wie auch alle anderen Makros und Konstanten) in einer Headerdatei, die in die Quelldatei eingebunden wird, oder in der Quelldatei selber definiert werden.

Makro als Ersatztext mit Argumenten

Es ist auch möglich, Makros zu definieren, denen eines oder mehrere Argumente übergeben werden können, um sie beispielsweise für Berechnungen heranzuziehen.
Als Beispiel soll ein Makro dienen, das aus den Seitenlängen eines Rechtecks dessen Fläche berechnen soll. Die Länge der Seiten A und B werden dem Makro als Argumente übergeben. Die Definition des Makros hat folgende Form:

```
#define FLAECHE(A,B)  (A*B)   /* Formulierung äußerst bedenklich */
```

Was passiert, wenn das Makro in folgender Programmzeile aufgerufen wird:

```
nErgebnis = FLAECHE(4,6);
```

Der Präprozessor setzt für das Makro folgendes ein:

```
nErgebnis = 4*6;
```

und die Berechnung ergibt:

```
nErgebnis = 24;
```

Was passiert, wenn die Seitenlängen des Rechtecks erst durch Addition gebildet werden müssen, und folgende Anweisung im Programm steht:

```
nErgebnis = FLAECHE(3+1, 2+4);
```

Weil der Präprozessor die Argumente einfach nur einsetzt wird daraus:

```
nErgebnis = (3+1*2+4);
```

und die Berechnungen ergeben:

```
nErgebnis = (3+2+4);
nErgebnis = 9;
```

Das falsche Ergebnis resultiert aus der nicht einwandfreien Definition des Makros. Alle Argumente, die in einer Makrodefinition vorkommen, sollten geklammert werden, um diese Nebeneffekte, die manchmal schwer zu finden sind, zu vermeiden. Ebenso den zu ersetzenden Text des Makro selbst, weil auch dieser wiederum als Term in einer Berechnung Verwendung finden kann.

```
#define FLAECHE(A,B)   ((A)*(B))    /* Einsatz funktioniert sicher */
```

Das Resultat wird dann (und so soll es sein):

```
nErgebnis = ((3+1)*(2+4));
nErgebnis = ((4)*(6));
nErgebnis = 24;
```

6.3 Makrodefinition entfernen mit #undef

Es ist möglich, Makrodefinitionen zu entfernen. Dies geschieht mit der Direktive *#undef* und wird meist eingesetzt, um sicherzustellen, daß es sich bei einer Routine wirklich um eine Funktion, und nicht um ein Makro handelt. Eine andere Einsatzmöglichkeit besteht darin, für bestimmte Bereiche in einer Quelldatei den gleichen Bezeichner zu verwenden, der dann jedoch eine neue Bedeutung bekommen soll.

#undef MakroName

- Mit *#undef* können alle definierten Makros "entdefiniert" werden, außer denen, die laut ANSI-Standard nicht verändert werden dürfen (siehe vordefinierte Makros).
- Bei der Entfernung der Makrodefinition wird nur der Makroname, ohne eine eventuell bei der Definition benutzte Parameterliste, angegeben.
- Die *#undef*-Direktive kann zusammen mit den Anweisungen zur bedingten Kompilierung eingesetzt werden.

6.4 Direktiven zur bedingten Kompilierung

Mit den Direktiven zur bedingten Kompilierung können verschiedene Bereiche einer Datei wahlweise kompiliert werden, um beispielsweise verschiedene Varianten eines Programms aufzubauen. Diese Varianten können sein:

- eine Programmvariante, die Debugging-Code enthält,
- Programmvarianten mit leicht unterschiedlichem Verhalten bei bestimmten Bedingungen,
- Programme, die auf unterschiedlicher Hardware laufen, oder
- Varianten, die mit unterschiedlichen Compilern arbeiten sollen.

Die Direktiven zur bedingten Kompilierung werten einen Bezeichner oder eine Konstante aus, um zu ermitteln, welche Bereiche der Quelldatei vom Präprozessor an den Compiler weitergeleitet bzw. aus dem Quellcode entfernt werden sollen. Der Anweisungsblock zur bedingten Kompilierung umfaßt mehrere Zeilen:

- Die Zeile, in der die zu testende Bedingung steht. Sie wird mit *#if*, *#ifdef* oder *#ifndef* eingeleitet.
- Zeilen, die den Code beinhalten, wenn die auszuwertende Bedingung den Wert 'wahr' ergibt (optional).
- Eine optionale else-Zeile, die mit *#else* oder *#elif* eingeleitet wird.
- Zeilen, die den Code beinhalten, wenn die auszuwertende Bedingung den Wert 'unwahr' ergibt (optional).
- Eine Zeile, mit der der Block zur bedingten Kompilierung abgeschlossen wird: die *#endif*-Anweisung.

6.4.1 Bedingte Kompilierung #if, #elif, #else

#if

Die Präprozessor-Direktive *#if* leitet die bedingte Kompilierung ein. Danach steht ein konstanter Ausdruck, der ausgewertet wird. Ergibt die Auswertung 'wahr', wird der nachfolgende Teil (entweder bis zum nächsten *#elif*, *#else* oder *#endif* an den Compiler weitergeleitet.

#elif

Die Direktive *#elif* ist eine Abkürzung für *"#else if*". Dieser Ausdruck wird nur dann ausgewertet, wenn die Auswertung der vorhergehenden Direktive das Resultat 'unwahr' erbrachte. *#elif* kann in einem Block zur bedingten Kompilierung mehrmals vorkommen. Ergibt die Auswertung 'wahr', wird der nachfolgende Teil (entweder bis zum nächsten *#elif*, *#else* oder *#endif* an den Compiler weitergeleitet.

#else

#else kann in einem Anweisungsblock nur einmal vorkommen. Die hinter *#else* stehenden Anweisungen werden bis zum abschließenden *#endif* an den Compiler weitergeleitet, wenn keine der vorhergehenden Auswertungen das Ergebnis 'wahr' erbrachten.

6.4.2 Ende des bedingten Kompilierblocks #endif

Die Präprozessor-Direktive *#endif* muß für jeden Kompilierblock, der mit *#if*, *#ifdef* oder *#ifndef* eingeleitet wird, verwendet werden, um das Ende des Blocks zur bedingten Kompilierung anzuzeigen.

6.4.3 Bedingtes Kompilieren und Makrodefinition #ifdef, #ifndef

Bei den Direktiven *#ifdef* und *#ifndef* handelt es sich um die alte Form der Direktiven zur bedingten Kompilierung. Mit ihnen kann nur getestet werden, ob der als Argument angegebene Bezeichner definiert ist oder nicht. Mit *#if defined* ist es darüberhinaus möglich, den zurückgelieferten Wert zu verknüpfen, was mit *#ifdef* und *#ifndef* nicht möglich ist.

#Ifdef Bezeichner

#ifndef Bezeichner

6.4.4 Der Operator defined

Der Präprozessor-Operator *defined* liefert den Wert 'wahr' (entspricht 1) zurück, wenn der als Argument angegebene Bezeichner zuvor mit *#define* definiert wurde. Ist der Bezeichner nicht definiert, gibt der Operator den Wert 'unwahr' (entspricht 0) zurück. Die beiden folgenden Anweisungen sind identisch:

#if defined MAXZEILEN
#ifdef MAXZEILEN

Beispiele

```
#undef ERR0R1
#if defined(ZIELSYSTEM) && (MAXSPALTEN==80)
  #define ARRAY GROESSE 160
#elif !defined(ZIELSYSTEM)
  #define ERR0R1
#endif

#ifdef ERR0R1
  #error Datei kann nicht kompiliert werden.
#endif
```

Das nachfolgende Beispiel berücksichtigt die Tatsache, daß ein Teil der Special-Function-Register bei Derivaten von 8051-Rechnern von ihrer Funktion her identisch sind, jedoch andere Namen benutzt werden. Ein Beispiel ist das serielle Senderegister.

```
#if defined __REG51_H_
  #define SER_SBUF SBUF
#elif defined __REG517_H_
  #define SER_SBUF S0BUF
#else
  #error Controller nicht bekannt!
#endif
```

6.5 Weitere Präprozessor-Direktiven

6.5.1 Leere Präprozessordirektive #

Die leere Präprozessor-Direktive besteht aus dem Zeichen *"#"* als einzigem Zeichen auf einer Zeile. Sie kann beispielsweise verwendet werden, um Bedingungen, bei denen keine Aktion erfolgt, im Programm besonders deutlich zu machen.

Beispiel:

```
#ifdef MAXZEILEN
  #
#else
  #define MAXZEILEN 25
#endif
```

6.5.2 Zeichenkettenbildung #

Der Operator zur Zeichenkettenbildung (stringize-operator) wird bei funktionsähnlichen Makros benutzt. Das Grundsymbol, das bei Abwesenheit von "#" zum aktuellen Argument gemacht würde, wird jetzt in eine Zeichenkette umgewandelt. Dieses Zeichenfolgeliteral ersetzt sowohl den Operator zur Zeichenkettenbildung als auch den formalen Parameter.

Beispiel:

Der Quellcode:
```
#include <stdio.h>
#define makestring(s) printf(#s "\n")
main()
{
makestring(Dies wird ein String);
makestring("Dies wird noch ein String");
}
```

Auszug aus der vom Präprozessor erzeugten Datei:
```
main()
{
printf("Dies wird ein String" "\n");
printf("\"Dies wird noch ein String\"" "\n");
}
```

Die Ausgabe des Programms:
```
Dies wird ein String
"Dies wird noch ein String"
```

6.5.3 Grundsymbolverbindung ##

Mit dem Operator zur Grundsymbolverbindung (token-pasting-operator) können einzelne Token zu einem Grundsymbol verknüpft werden. Der Operator "##" sowie alle zwischen den Token stehenden Zwischenraumzeichen werden vom Präprozessor entfernt.

Beispiel:

Die Quelldatei:

```
#include <stdio.h>
#define ZEIGEVAR(Z) printf("%d\n", nVar##Z)
main()
{
int nVar3=3;
ZEIGEVAR(3);
}
```

Auszug aus der vom Präprozessor erzeugten Datei:
```
main()
{
int nVar3=3;
printf("%d\n", nVar3);
}
```

6.5.4 Diagnosemeldung #error

Mit der Verwendung der Präprozessor-Direktive *#error* erreicht man, daß der Compiler den (optionalen) Text auf dem stderr-Gerät ausgibt und anschließend die Kompilierung abbricht. Der Text braucht nicht in Anführungszeichen eingeschlossen zu werden.

#error <text>

Beispiel:

```
#if !defined AUTO && !defined FAHRRAD
      #error Transportmittel ist nicht definiert!
#endif
```

Das vorstehende Beispiel könnte aus einem Programm stammen, bei dem über die bedingte Kompilierung, je nach gewünschtem Transportmittel, anderer Code erzeugt wird. Da das Programm nur einwandfrei kompiliert werden kann, wenn entweder AUTO oder FAHRRAD definiert ist, wird mit dem defined-Operator getestet, ob die Definition existiert. Ist dies nicht der Fall, wird der hinter *#error* stehende Text ausgegeben.

6.5.5 Zeilensteuerdirektive #line

Mit der Zeilen-Steuerdirektive *#line* kann die Zeilennummer innerhalb einer Datei eingestellt, und der Name der Datei geändert werden. Diese Daten werden vom Compiler benutzt, um auf eventuelle Fehler zu verweisen.

#line <Zeile> ["Dateiname"]

Zeile	Eine Ganzzahlkonstante, auf die der interne Zähler des Compilers gesetzt wird.
Dateiname	Dieses Argument ist optional und kann einen in Anführungszeichen eingeschlossenen Dateinamen enthalten.

Beispiel:

```
#line 200
#line 200 "ursprung.c"
```

Durch die Verwendung von *#line* werden die Werte in den Symbolen *__LINE__* und *__FILE__* geändert. Diese Direktive wird üblicherweise von Programmgeneratoren benutzt, um Fehler in dem erzeugten C-Code mit der Zeile und dem Dateinamen der Ursprungsdatei zu verknüpfen.

6.6 C51-Compileranweisungen

Unter den Compileranweisungen, die im C51-Compiler implementiert sind, gibt es einige, die mir wichtig erscheinen, weswegen sie hier aufgezeigt werden.

Die Steueranweisungen können in der Aufrufzeile oder innerhalb des Quelltextes in Präprozessorzeilen (*#pragma*) enthalten sein. Die Steueranweisungen können in zwei Klassen aufgeteilt werden: Primary Controls und General Controls.

- Ein Primary Control kann nur einmal angegeben werden. Die wiederholte Verwendung eines Primary Controls erzeugt einen Fehler und bricht den Compilerlauf ab.
- General Controls können nach Bedarf mehrfach in einer C Quelldatei verwendet werden.

Die Steueranweisungen können nach Ihrer Funktion in drei Gruppen eingeteilt werden:

- Quell-Steueranweisungen
  Mit den Anweisungen dieser Gruppe werden beispielsweise in der Aufrufzeile Makros definiert und der Name der zu übersetzenden Datei bekanntgegeben.
- Objekt-Steueranweisungen
  Mit angegebenen Anweisungen der Objektgruppe wird unter anderem der Optimierungsfaktor sowie die in der Objektdatei enthaltenen Debug-Informationen zum symbolischen Test des Programms freigegeben.
- Listing-Steueranweisungen
  Mit diesen Anweisungen wird unter anderem festgelegt, ob der Inhalt von Include-Dateien im Listing erscheint, ob nicht übersetzte Teile des Moduls zur besseren Übersicht gekennzeichnet werden.

Steuerparameter und deren Argumente können sowohl in Groß- als auch Kleinbuchstaben angegeben werden. Die einzige Ausnahme stellen die Argumente der Anweisung *DEFINE* dar, die sich mit der Schreibweise im Quellprogramm decken müssen, um von einer der Präprozessoranweisungen if/ifdef/ifndef erkannt werden zu können.

6.6.1 Anweisung PREPRINT

Diese List-Steueranweisung aus der Klasse der Primary Controls kann nur beim Compileraufruf (vgl. Kap. 9.3) verwendet werden. Eine Verwendung innerhalb des Quellcodes ist aus technischen Gründen nicht möglich, d.h. diese Anweisung kann nicht mit *#pragma* verwendet werden. Mit *PREPRINT* wird der Compiler angewiesen, ein Präprozessor-Listing zu erzeugen. In diesem Listing sind alle Makroaufrufe expandiert und alle Kommentare entfernt. *PREPRINT* ohne Argument verwendet als Dateinamen den Namen der Quelldatei mit der Erweiterung ".i". Wird ein anderer Dateiname gewünscht, so muß er entsprechend angegeben werden.

Abkürzung: PP

6.6.2 #pragma DEFINE

Wird dieser Primary-Quell-Steueranweisung ein oder mehrere durch Komma voneinander getrennte Namen entsprechend den Namenskonventionen der C Sprache in Klammern übergeben, dann entspricht für jeden Namen diese Anweisung der Präprozessoranweisung *#define*. Weil es die Präprozessoranweisung schon gibt, ist es eigentlich sinnvoll, diese Anweisung ausschließlich in der Aufrufzeile zu verwenden, um beispielsweise eine andere Programmvariante zu erzeugen. Mit *DEFINE* können in der Aufrufzeile Namen definiert werden, die innerhalb von Präprozessorzeilen mit *#if, #ifdef, #ifndef* zur bedingten Übersetzung abgefragt werden können (vgl. Kap. 6.2 und 6.4). Die definierten Namen werden so übernommen, Wie sie geschrieben sind, d.h. es wird Groß- und Kleinschrift beachtet. Als Option kann ein Argument für den Namen gegeben werden.

Beispiele:

C51 SAMPLE.C DEFINE (SERIAL1, TIME_TASTEN)
entspricht innerhalb des Codes den Präprozessoranweisungen
```
#define SERIAL1
#define TIME_TASTEN
```

C51 TEST.C DEFINE (PROG_VARIANTE=10)
entspricht innerhalb des Codes der Präprozessoranweisung
```
#define PROG_VARIANTE 10
```

Abkürzung: DF()

6.6.3 #pragma SMALL

Diese Primary-Objekt-Steueranweisung ist eine der drei, die das Speichermodell vorwählen. Wie in Kapitel 2.5 schon erwähnt, ist es am sinnvollsten das Speichermodell *SMALL* zu wählen, damit für die Overlaytechnik der zwar geschwindgkeitsmäßig leistungsfähigste jedoch speicherplatzmengenmäßig dürftigste interne Datenspeicher am besten genutzt werden kann.

Abkürzung: SM

6.6.4 #pragma PRINT

Die Primary-List-Steueranweisung *PRINT* veranlaßt den Compiler ein Listing mit der Dateiendung ".LST" zu erzeugen. Hier kann als Option ein anderer Dateiname in Klammern übergeben werden.

Abkürzung: PR

6.6.5 #pragma CODE

Die Primary-List-Steueranweisung *CODE* veranlaßt den Compiler dazu den erzeugten Maschinencode in Form von Mnemonics in das Listing mit aufzunehmen.

Abkürzung: CD

6.6.6 #pragma NOCOND

In der Standard-Einstellung erscheinen nicht übersetzte Zeilen des Quellprogramms ebenfalls im Listing. Zur besseren Überprüfbarkeit sind dabei jedoch die Zeilennummern sowie die Blocktiefe nicht angegeben. Damit ist ersichtlich, daß es sich dabei um übersprungene Quellzeilen handelt.
Die Primary-List-Steueranweisung *NOCOND* legt fest, daß der noch besseren Übersicht halber die bei der bedingten Übersetzung nicht übersetzten Teile des Quellprogramms im Listing auch nicht ausgegeben werden. Die Ausgabe erfolgt erst eine Zeile nach der Präprozessor-Anweisung, die die Übersetzung wieder aktivierte.

Abkürzung: NOCO

6.6.7 #pragma DEBUG

Mit der Primary-Objekt-Steueranweisung *DEBUG* wird der Compiler angewiesen, in die Objektdatei Debug-Informationen zu übernehmen. Debug-Informationen sind zum symbolischen Test von Programmen notwendig und beinhalten Variablen, Funktionsnamen und Zeilennummern, Diese Informationen bleiben auch nach dem Link/Locate-Vorgang erhalten, so daß der Programm-Test mit Source-Level-Debuggern oder -Emulatoren durchgeführt werden kann. Zu beachten ist, daß für komplette Typen-Information auch die Anweisung *OBJECTEXTEND* erforderlich ist.

6.6.8 #pragma OBJECTEXTEND

In Verbindung mit der Anweisung *DEBUG* wird mit der Primary-Objekt-Steueranweisung *OBJECTEXTEND* der Objektcode um Typenbeschreibungen für Objekte (Typedef-Records) erweitert. Gleichzeitig spiegelt die Anordnung der Records das Layout des Quellprogramms wider, so daß gleichnamige Objekte in verschiedenen Gültigkeitsberelchen einwandfrei identifiziert werden können.
Diese OMF-Erweiterungen setzen voraus, daß entsprechend erweiterte Programme zur Verarbeitung der Objektmodule verwendet werden. Werden Programme oder Emulatoren verwendet, die nur den Standard-Umfang des OMF-51 verstehen, so kann und darf die option *OBJECTEXTEND* nicht verwendet werden.

Abkürzung: OE

6.6.9 #pragma OPTIMIZE

Die General-Objekt-Steueranweisung *OPTIMIZE* wird eine in Klammern eingeschlossene dezimale Zahl zwischen 0 bis - je nach Compilerversion - 6 übergeben. Hierdurch wird der Optimierungsgrad eingestellt. Eine höhere Optimierungsstufe beinhaltet immer alle Optimlerungen niedrigerer Einstellungen. *OPTIMIZE* kann an beliebiger Stelle im Progarmm eingestellt werden.
Zusätzlich kann mit *OPTIMIZE(SIZE)* oder *OPTIMIZE(SPEED)* ausgewählt werden, ob bei der Optimierung mehr auf Code-Größe oder auf Ausführungszeit geachtet werden soll.

Beispiel:
```
#pragma OPTIMIZE(5, SPEED)
```

Was in den einzelnen Stufen wie optimiert ist, sollte im Handbuch zum Compiler nachgelesen werden.

Abkürzung: OT()

6.6.10 #pragma AREGS/NOAREGS

Die General-Objekt-Steueranweisung *AREGS* bewirkt, daß der Compiler absolute Register Adressierung verwendet. Die absolute Adressierung erhöht die Effizienz, da beispielsweise *PUSH* und *POP* Befehle nur mit direkten (absoluten) Adressen arbeiten. Die verwendete Registerbank kann mit der *REGISTERBANK*-Anweisung eingestellt werden, wobei im Gegensatz zum Schlüsselwort *using* die Registerbank nicht zur Laufzeit umgeschaltet wird. Mit der *NOAREGS* Anweisung wird die absolute Adressierung abgeschaltet. Funktionen, die mit *NOAREGS* übersetzt wurden, sind unabhängig von der Registerbank, d.h. sie können mit allen Registerbänken des 8051-Controllers arbeiten. *NOAREGS* sollte meiner Meinung nach die Voreinstellung sein, weil sonst Funktionen nicht aus unterschiedlichen Registerbank-Ebenen benutzt werden können. Ein typisches Beispiel hierzu ist der Zugriff auf die serielle Schnittstelle über eine Funktion wie z.B. *putchar()* sowohl von der Hauptebene als auch von einer Interrupt-Ebene aus (vorausgesetzt man verhindert gleichzeitigen Zugriff oder alle lokalen Variablen sind in Registern untergebracht). Zwar gäbe es hier die Möglichkeit die Routine *putchar()* als *reentrant* zu codieren, dies ist allerdings wiederum alles andere als Laufzeitoptimal (vgl. Kap. 5.4).
Die Option *AREGS/NOAREGS* kann nur außerhalb einer Funktion verändert werden.

6.6.11 #pragma SAVE/RESTORE

Diese beiden General-Quell-Steueranweisungen dienen zum Sichern bzw. Restaurieren der Einstellungen von *AREGS*, *REGPARMS*, den *OPTIMIZE*-Faktor und die *SPEED/SIZE*-Einstellung der Optimize-Option auf dem Savestack. Die maximale Schachteltiefe für *SAVE*- und *RESTORE*-Anweisungen beträgt acht.

Beispiel:
```
#pragma save
#pragma optimize(3,size)
void Test1 (char cZeichen, int nWert)
{
C-Anweisungen
}
#pragma restore
```

7 Software-Richtlinien (→ Qualität)

C ist eine Sprache, die quasi alles möglich macht. Daher ist die Anfälligkeit fehlerhafte Programme zu schreiben ebenfalls etwas höher, als dies z.B. bei den BASIC-Varianten möglich ist. Dennoch ist es möglich das Qualitätsniveau einer Firmware anzuheben indem folgende Teilziele angestrebt werden sollten:

- Kapselung (information hiding)
- Schichtentrennung, Hierarchisierung
- Robustheit, Fehlerarmut
- Lesbarkeit
- Flexibilität, Änder- und Erweiterbarkeit
- Diagnosefähigkeit, Wartbarkeit
- Benutzerfreundlichkeit
- Performance
- Modularisierung, Wiederverwendbarkeit

Die folgenden Richtlinien beziehen sich auf die Sprache C und sind für hardwarenahe Softwarekonzeptionen (Firmware) aufgestellt. Die Programmierung in Assembler speziell für Entrycode und zeitkritische Echtzeitanwendungen sollte dabei gedanklich in dieser Hinsicht nicht außer Acht bleiben. Die Richlinien können sicherlich auch auf andere Programmier-sprachen und –umgebungen übertragen werden.
Sie nutzen der Fehlervermeidung und der schnellen Fehlerauffindung.
Durch eine strukturierte Programmierung und durch die Modularisierung kann eine Wieder-verwendbarkeit von Programmteilen erreicht werden.

Die zeitliche Investition, die mit der Einhaltung dieser Richtlinien verbunden ist, ist lohnenswert, wenn man diese ins Verhältnis zum resultierenden Nutzen setzt. Erstens werden bei übersichtlicher und gut lesbarer Gestaltung von Software Fehler vermieden, und zweitens ist ein eventueller Gedankenfehler erheblich schneller zu finden. Bereits erstellte Module können quasi ohne Zeitaufwand erneut verwendet werden.

7.1 Fehlervermeidung / -suche

Es gibt viele Fehler, die durch erhöhte Aufmerksamkeit und Übung vermieden werden können. Hier sind einige typische Fehlerquellen aufgelistet, die trotz allem immer wiederkehren:

- unnötige Globalisierung von Daten
- fehlende Initialisierung
- Datenverlust durch unbemerkten impliziten Typecast
- fehlende Klammerung in #define (vgl. S. 67)
- Referenz ohne Ziel (wird meist vom Linker bemerkt)
- falsche Referenzstufe (fehlende Dereferenzierung "*" oder Adressoperator "&")
- Zuweisung statt Vergleich "=="
- Anwendung von "=="-Operator auf Gleitkomma-Rechenergebnis.
- unvollständiges Umeditieren von Passagen nach Copy+Paste
- Diskrepanz zwischen Layout und Block- / Klammerstruktur
- Über- / Unterlauf im Zwischenergebnis von arithmetischen Formeln
- unpassende Formatstrings bei Ein- / Ausgabe
- mangelnde Beachtung von Operatorprioritäten, insbes. bei Bit-Operationen
- "off-by-one", insbes. bei "<=", ">=" und bei 0-Basis / 1-Basis in Indizes
- switch-case ohne default
- Endlosschleife, in der keine Austrittsbedingung wahr wird
- undefinierter Rücklieferwert, weil ein Zweig nicht beachtet worden ist
- Überschreiten der Leistungsgrenzen des Compilers

Zur Fehlervermeidung gehört auch die defensive Programmierung ("fail-safe"), d.h. Begrenzung der Symptomausbreitung durch

- Abfangen von Ausnahmen
- lückenlose Fehlerbehandlung
- "pessimistische" Initialisierung
- Selbsstdiagnose durch Einbau von Absicherungen und zuschaltbaren Testausgaben
- Einbau eines Watchdogs (s. Kap. 8.2.5), der im Falle eines Absturzes das System wieder in einen sicheren Zustand bringt. Ideal ist der gleichzeitige Einbau einer Diagnosemeldung, mittels der sich der Ursprung des Fehlers lokalisieren läßt.

Bei der Fehlerbehandlung werden ca. 95% des zeitlichen Aufwands für das Lokalisieren des Fehlers und nur 5% für die Behebung des Fehlers verwendet.
Es hat sich gezeigt, daß es in der Regel am effektivsten ist, den Fehler durch Nachdenken zu lokalisieren. Meistens ist es überhaupt nicht erforderlich den Fehler durch Ausprobieren zu finden. Droht das Nachdenken bei gleichzeitigem Durchsehen des Codes hoffnungslos zu werden (temporäre Blindheit), dann sollte man einen neuen Versuch unter Zuhilfenahme eines Kollegen starten, der nicht einmal in der Thematik stecken muß. Erklärt man dem Kollegen, wie es funktionieren soll, dann entdeckt man plötzlich, warum die Codierung so nicht funktionieren kann.
Auch das Überschlafen der Sache hat schon einige Male geholfen.

7.2 Prinzipien

Eine gute Software zeichnet sich nach außen neben der Fehlerarmut (Fehlerfreiheit ist kaum erreichbar) auch durch hohe Performance aus. Da die Rechenleistung der 8051-Controller nicht die höchste ist, sollten der Steigerung der Performance wegen die Strategien gem. Kapitel 8.1 beachtet werden.

Zur Erreichung dieser beiden und der anderen o.g. Teilziele zur Qualitätsverbesserung einer Software gibt es aus der Lehre des SW-Engineerings verschiedene Prinzipien, welche die Gestaltung einer Software betreffen. Diese sollten meiner Meinung nach verstärkt Anwendung finden.

1. Das Prinzip der ***Modularisierung***.
2. Das Prinzip der ***Hierarchisierung***.
3. Das Prinzip der ***Verbalisierung***.
4. Das Prinzip der ***Strukturierung***.
5. Das Prinzip der ***schmalen Datenkopplung***.
6. ***Teststrategien*** (Tests, Verifikation).

7.2.1 Modularisierung

Ein Modul stellt eine überschaubare, weitgehend logisch in sich abgeschlossene, funktionale Einheit mit festgelegten Schnittstellen für Ein- und Ausgänge dar (Kapselung). Ein Modul sollte sich auf zwei Dateien beschränken, nämlich die *.c und die zugehörige *.h-Datei. Beispiele hierfür sind Anzeigentreiber, Schnittstellentreiber, Funktionensammlung zur Ansteuerung von komplexen IC´s, sowie Routinen für Drucker, Stringbearbeitung, usw. . Wird ein Modul mehrfach eingesetzt, so kann es als Baustein einer Bibliothek hinzugefügt werden.

7.2.2 Hierarchisierung

Auf Projektebene:
Es werden Module konzipiert, die auf unterster Ebene mit der zu programmierenden Hardware konfrontiert werden (Treiber). Hierzu gehören z.B. Module mit Funktionen zur Ansteuerung von IC´s, Displays oder Tasten(feldern).
Auf höherer Ebene wird dann auf Funktionen der unteren Modulebenen zurückgegriffen, Hardwarezugriffe existieren nicht mehr.

Auf Modulebene:
Funktionen innerhalb eines Moduls bauen aufeinander auf. Auf unterster Ebene werden die Funktionen direkt mit der anzusteuernden Hardware konfrontiert. Höhere Funktionen bilden dann eine Schnittstelle zur nächsten Modulebene. Sie können noch Hardwarezugriffe enthalten.

Führt eine starke Hierarchisierung bedingt durch die häufigen Funktionsaufrufe zu unvertretbaren Programmlaufzeitverlängerungen (Performanceverlust) und/oder Stackproblemen, so sollte man eine geeignete Kompromißlösung finden.

7.2.3 Verbalisierung

Um den Gedankengang und die Ideen des Programmierers leicht nachvollziehen zu können, sind aussagekräftige mnemotechnische Namen zu vergeben (nicht *a, b, c* als Variablennamen).
Anzustreben ist das Ziel, den Code selbstredend bzw. "lesbar wie eine Geschichte" zu machen.
Dann sind auch Kommentare nur noch bei Programmiertricks nötig.
Dadurch wird das Programm übersichtlicher.
Mit Makrodefinitionen ist nicht zu sparen, denn sie erleichtern die Lesbarkeit besonders.

Beispiel zum Einschalten einer Betriebsanzeige:

Oft gesehen:
```
P1.1 = 1;                    /* Power-Led einschalten */
```
besser:
```
POWER_LED = EIN;             /* Kommentar überflüssig */
```

Beispiel einer Warteschleife im Mikrosekundenbereich:

Oft gesehen:
```
for(i=0; i<15; i++);         /* Ein bißchen warten */
```
besser:
```
WAIT_uS(80);                 /* Kommentar überflüssig */
```

Beispiel zum Steuern einer Spannung mittels DA-Wandler:

Oft gesehen:
```
*((char *)(0x20000L)) = 220;  /* Offset 2,2V */
```
besser
```
OFFSET_SPG = 2200 MV;
```

Hierdurch sind verschiedene Realisierungen zentral änderbar, wenn Hardwareänderungen vorgenommen wurden, wie z.B. die Verlegung einer Portleitung oder der Einsatz verschiedener CPU-Taktfrequenzen.

In der Programmiersprache C ist keine grundsätzliche Möglichkeit vorgesehen, anhand des Variablennamens ihren Datentyp zu erkennen. Will man jedoch z.B. mit einer globalen Variablen arbeiten, die zudem in einer eingebundenen Datei deklariert wurde, so muß man sich zunächst über den Datentyp dieser Variablen informieren. Dies kann bei großen Programmen zu zeitraubenden Umständen führen.
Daher ist zur Unterscheidung der Datentypen von Variablen, der Deklarationsebene von Variablen, und der Makrodefinitionen die im Kapitel 7.6 befindliche Ungarische Notation anwendbar.

7.2.4 Strukturierung

Die modulare und hierarchische Programmgestaltung ist ein Teil der Strukturierung. Sie ermöglicht auch die Wiederverwendung von bereits erstellter Software speziell bei Modulen zu komplexeren IC´s oder Baugruppen.
Weiterhin bilden Funktionen übersichtliche kleine Einheiten (i.d. Regel max. 2 Bildschirmseiten).
Zusammengehörende Variablen werden in Strukturen zusammengefaßt. Zum Beispiel für die Konfiguration einer seriellen Schnittstelle:

```
struct
    {
    byte byDataBits;
    byte byStopBits;
    dword dwBaudrate;
    } typtSER_CONFIG;
```

7.2.5 Schmale Datenkopplung

Je geringer die Kopplung eines Moduls mit anderen Modulen ist, desto weniger ist es abhängig vom restlichen Kontext. Die ideale (schmalste) Kopplung ist die explizite Übergabe von Parametern als Einzelelemente und die Verwendung der Rückgabewerte aus der Funktion. Ein solches Modul mit hoher Wartungsfreundlichkeit läßt sich am besten wiederverwenden, weil es leicht aus dem Programm zu lösen ist.
Man sollte darauf achten, möglichst wenig gemeinsam benutzte Datenbereiche und globale Variablen zu verwenden. Auf diese Weise wird auch die Zahl der unerwünschten Seiteneffekte klein gehalten.
Die implizite Annahme von Eigenschaften aufgerufener Funktionen sollte vermieden werden. Bei Bedarf bestimmter Eigenschaften sind diese über die Parameterliste auszudrücken.

7.2.6 Teststrategien

Man kann zwischen Entwicklertests (Tests entwicklungsbegleitend vom Entwickler durchgeführt) und der Softwareverifizierung (Tests durch andere Personen nach Erstellung von lauffähigen Programmversionen) unterscheiden.

Entwicklertests:
Zum Testen von Firmware oder anderer hierarchisch strukturierter Software beginnt man mit den Funktionen auf der untersten Ebene (Bottom-Up-Strategie). Liefert die Hardware keine zu verarbeitenden Werte, so werden Dummy-Werte eingesetzt. Bereits getestete Funktionen dienen danach dem Testen von Funktionen höherer Ebene.
Grenzwerttests beinhalten die Überprüfung von Grenzwerten innerhalb einer Funktion und die der zu verarbeitenden Parameter (z.B. Abfangen einer Überschreitung des Gültigkeitsbereichs von Variablen oder Begrenzung von Indizes auf Arrays).
White-Box-Tests überprüfen die Reaktion auf jede mögliche Verzweigung innerhalb einer Funktion.

Softwareverifizierung:
Eine Art der Softwareverifizierung ist der Code-Review, bei dem durch fachkundige Kollegen abschnittsweise der Quellcode gelesen wird. Er ist ein effektives Mittel zur Risikoreduzierung, wobei nicht nur die korrekte Funktionalität der Software im Blickfeld sein soll, sondern auch die Einhaltung der geforderten Richtlinien.
Weitere Tests zur Prüfung der Robustheit sind die Eingabe von zufälligen Werten (Zufallstests), sowie die Über- und Unterlastung des Systems (Belastungstests).

7.3 Dateien

Ein Programm bzw. ein Projekt besteht nicht nur aus einer einzigen Quellcode-Datei, sondern ist in mehrere Module mit unterschiedlicher Bedeutung aufgeteilt, wobei jedem Modul eine *.c- und eine *.h-Datei zugeordnet sein sollte.

7.3.1 Codefiles

Der implementierte Code (Funktionen) befindet sich im *.c-File (Beispiel im Kapitel 7.7.1). Eingebunden werden Headerfiles des eigenen und anderer Module, sowie notwendige Headerfiles (allgemeine hardware- und systembezogene Definitionen).

7.3.2 Headerfiles

Zu jedem Modul (*.c-File) ist ein Headerfile *.h anzulegen (Beispiel im Kapitel 7.7.2).
Dieses enthält:

Sichtbar für alle einbindenden *.c-Files:
- Projektglobale Variablen (möglichst wenige)
- Hardwaredefinitionen
- Allgemeine Definitionen
- Datentypendefinitionen
- Funktionsmakros
- Funktionsprototypen mit streng funktionsbezogenen Definitionen

Sichtbar nur für das zugehörige *.c-File:
Nur auf Modulebene deklarierte Variablen
Deklaration nur auf Modulebene benutzter Funktionen
Andere, nur für das Modul sichtbare Definitionen

Headerfiles von SW-Modulen, die zu einer HW-Baugruppe oder komplexeren IC´s gehören, beschreiben die Baugruppe durch Definitionen (*#define*) oder Strukturen. So entfällt das Nachschlagen in Datenblättern zu Steuersequenzen oder baugruppeninternen Adressen. Bei Baugruppen die z.B. Tasten(felder) (oder Segment-Displays) enthält, werden die Tasten (Display-Segmente) definiert, sowie im Bedarfsfall Zeichensätze und Tastencodes festgelegt.

7.3.3 Notwendige Headerfiles

In der Datei '**compiler.h'** (Beispiel im Kap. 7.7.3) werden compilerspezifische Einstellungen vorgenommen (Präprozessoranweisungen) und grundlegende allgemeine Definitionen (nicht projektgebunden) festgelegt (z.B. `#define FALSE 0, typedef unsigned char byte;`).
Diese Datei wird als erste Datei in jedes *.c-File des Projektes eingebunden, damit alle Module die gleichen Einstellungen haben. Um die Litanei einzubindender allgemeiner Headerfiles nicht in jedem Modul auflisten zu müssen, können diese auch zentral am Schluß in die compiler.h eingebunden werden.
Wenn nicht eine eigens dafür vorgesehene Datei (z.B. **'version.h'**) existiert, werden in der compiler.h auch versions- und variantenspezifische Definitionen vorgenommen.

Zwischen der Software und der Hardware ist eine eindeutige Schnittstelle aufzubauen. Diese Schnittstelle wird realisiert durch eine Datei ´**map.h**´ (Beispiel in Kap. 7.7.4). Diese Datei enthält alle Festlegungen von Ports, Adressen von einfachen IC´s, für die sich keine eigene Header-Datei lohnt (wie z.B. AD-Wandler, Tastenbelegungen usw.).

Headerfiles mit modulübergreifenden globalen Variablen (z.B. oft gesehen: 'def.h') sollten vermieden werden. Eine saubere Programmstruktur hat eine solche Datei nicht nötig.

7.4 Aufbau / Darstellung

Die gesamte Software sollte ein einheitliches Bild ergeben. Dies erleichtert die Einarbeitung anderer Mitarbeiter (und die eigene, wenn man seine Arbeit zwei Jahre nicht gesehen hat), weil eine Umgewöhnung nicht nötig ist.
Der Überschaubarkeit halber sollten folgende Größen ein Anhaltspunkt sein:

- Die Zahl der Parameter für eine Funktion sollte 5 nicht überschreiten, bei C51 besser nur 3, damit die Parameter noch in die Register passen (vgl. Kap. 5.1)
- Die Zahl der Codezeilen für eine Funktion sollte bei max. 100 liegen.
- Die Länge einer Zeile sollte bei <= 80 Zeichen liegen
- Die Zeilenzahl pro *.c-File liegt gerne bei 300 bis 1000, die Obergrenze bei 4000.
- Es sollten nicht mehr als 5 Ebenen eingerückt werden.

Läuft man Gefahr eine oder mehrere dieser Grenzen zu sprengen, dann sollte man sich Gedanken um eine weitere Unterteilung in Funktionen oder Module machen.

7.4.1 Modulheader

Jede C-Datei hat einen Header mit folgendem Inhalt:

- Name und Adresse der Firma
- Projektbezeichnung
- Name der Datei
- Aufgabe des Moduls
- System, auf dem die Firmware läuft und Angaben über die Version des benutzten Compilers.
- Auflistung der enthaltenen Funktionen zur leichten Auffindung.
- Namenskürzel des Bearbeiters
- Datum der letzten Bearbeitung
- Vorgenommene Änderungen

Jede H-Datei hat einen Header mit dem gleichen Inhalt wie die C-Datei.
Abweichend davon entfällt die Auflistung der Funktionen. Bei "Aufgabe" ist meistens "Headerfile zu *.c" genannt.

7.4.2 Funktionsheader

Jede Funktion hat einen Header mit folgendem Inhalt:

- Aufgabe der Funktion
- wenn nicht offensichtlich, dann Erläuterungen zu Parametern und Rücklieferwerten.
- Namenskürzel des Bearbeiters
- Datum der letzten Bearbeitung
- Vorgenommene Änderungen

7.4.3 Verschiedenes

Um bei mehrfacher Einbindung eine mehrfache Übersetzung zu vermeiden beginnen alle Dateien (am Beispiel "datei.h") mit:

```
#ifndef __DATEI_H_
#define __DATEI_H_
/* Dies wird nur bei der ersten Einbindung eines Compilerlaufs übersetzt */
```

und enden mit

```
#endif  /* __DATEI_H_ */
```

Wird "datei.h" in einem Compilerlauf eingebunden, dann ist das Makro `__DATEI_H_` definiert. Wird sie im gleichen Compilerlauf ein zweites mal eingebunden (z.B. durch verschachtelte Einbindung), dann wird alles was innerhalb des o.g. Blocks steht, nicht mehr übersetzt.

Zusätzlich ist es dadurch einfacher Makros oder Variablen nur für bestimmte Module (meist das eigene) sichtbar zu machen. Beispiel innerhalb der "datei.h":

```
#ifdef __DATEI_C_
/* Dies hier ist nur sichtbar, wenn die "datei.c" übersetzt wird */
#define ...
#endif /*__DATEI_C_*/
```

Funktionen sind optisch so voneinander zu trennen, daß das Ende bzw. der Beginn einer Funktion deutlich ins Auge fällt.

Beim **Einrücken** innerhalb von Kontrollstrukturen (wie *if(), for(), ...*) beträgt die Einrücktiefe 4 Leerzeichen. Es wird immer, d.h. auch bei nur einer Bedingungsanweisung eine neue Zeile begonnen und eingerückt. Eine Ausnahme kann eine größere Anzahl sehr kurzer, gleichartiger Anweisungen sein, wenn dies die Übersicht verbessert (vornehmlich bei switch-Anweisungen). Beispiel zum Einrücken:

```
while(cZeichen = *szText)
    {
    putchar(cZeichen);
    szText++;
    }
```

Geht die Länge einer Zeile über das 80ste Zeichen hinaus, so wird mit einer weiteren Einrückung von zwei Zeichen in der nächsten Zeile fortgefahren.
Dieser Umstand tritt meist bei umfangreichen printf- oder if-Abfragen auf.

7.5 Dokumentation

Eine Software, die geschrieben worden ist, sollte, um eine leichte und zeitsparende Einarbeitung von Mitarbeitern zu ermöglichen, dokumentiert und beschreiben sein. Es ist zu viel verlangt, wenn der Quellcode an sich ausreichen soll, um den Gedankengängen des Programmierers zu folgen.

7.5.1 Inline-Dokumentation

Teil der Inline-Dokumentation sind die File- und Funktionsheader.
Ein weiterer Teil sind die Kommentare in den Programmierzeilen.
Programmiertricks, insbesondere schwer verständliche Makrodefinitionen sind ausführlich zu kommentieren.
Beispiel:

```
#define US(X)                ((X)*F_OSZ/12000UL)
/*                             |    |      |
Zeit in µS---------------------|    |      |
Oszillatorfrequenz in kHz----------|      |
12 Osz.perioden für 1 Zyklus *1000-------|
*/
```

In einem Headerfile zu einer Baugruppe die z.B. Tasten(felder) (oder Segment-Displays) enthält, so wird im Kommentar auch die Anordnung dargestellt. Hierzu kann man gut die Semigrafikzeichen aus der ASCII-Tabelle zu verwenden.

7.5.2 Firmware-Beschreibung

Zu einer Firmware gehört auch eine Beschreibung. Diese Beschreibung kann als "Technische Funktionsbeschreibung" (kurz TEFU) bezeichnet werden und soll die Einarbeitung neuer Mitarbeiter ohne weitere Worte ermöglichen.
Sie beschreibt neben den Hardwarefunktionen auch die einzelnen Firmware-Module und die zugehörigen Funktionen und Datenstrukturen. Der Netto-Umfang beträgt etwa eine Seite pro 10kByte Quellcode. Zusätzlich sind bildliche Darstellungen zu erzeugen (Programmablaufpläne bzw. Nassi-Schneidermann-Diagramme). Ebenfalls ist die Entwicklungsumgebung inklusive aller Hilfsprogramme wie Batch-, Make-, und andere Files sowie Hinweise mit deren Umgang zu beschreiben. Mit Hilfe dieser Beschreibung sollte es für andere Mitarbeiter möglich sein die Firmware zu verstehen, und ggf. Änderungen vorzunehmen.

7.5.3 Änderungen / -Versionen

Eine Firmware durchläuft mehrere Lebensphasen.
Zur Nachvollziehung der einzelnen Schritte, sind in der Entwicklungsphase, der Erprobungsphase und der endgültigen Serie Versions- und Revisionsnummern zu vergeben. Diese Vergabe kann mit den entsprechenden Werkzeugen automatisch geschehen. Hier sei von der manuellen Versionierung gesprochen, bzw. die Fähigkeiten, über die eine automatisierte Versionierung verfügen sollte.

Abgesehen von den Änderungseinträgen in Funktions- und Fileheadern sind Firmware-Änderungen in der Erprobungs- und Serienphase in Einzelheiten als Text gesammelt in einer Datei z.B. ´**revision.txt**´ zu dokumentieren. Zusätzlich werden angegeben: Die neue Firmware-Revision, Datum der Änderung und Namenskürzel des Programmierers.

Die Versionsbezeichnung ist in den Code so als Klartext zu implementieren, daß sie auch im Binärcode als solcher zu finden ist. Sie enthält:

- Firmenname und -sitz
- Gerätevariante (Gerätekurzbezeichnung, Variante)
- Firmware-Version: Vhh.nn.rrrr
  - hh: Laufende Hauptversionsnummer (Inkrementierung bei grundlegenden Änderungen, z.B. HW-Revision von 1 beginnend).
    Die Hauptversionsnummer 0 kennzeichnet eine Firmware im Entwicklungs-stadium.
  - nn: Nebenversionsnummer (Inkrementierung bei kleinen Änderungen mit Funktionserweiterungen von 00 beginnend)
  - rrrr: Revisionsnummer (Inkrementierung bei kleinsten Änderungen in Form von Bug-Fixings von 0000 beginnend)
- Datum des Compilats

7.5.4 Archivierung

In der Entwicklungsphase wird eine Sicherheitskopie eines jeden Arbeitstages, die Änderungen in der Firmware vorgebracht hat, angelegt. Diese Kopie umfaßt Soure-Codes, Makefiles, und auch die Objekt-Dateien.

Die Reproduktion von veralteten Firmware-Revisionen in der Erprobungs- und Serienphase ist in manchen Fällen zum Debuggen wichtig. Daher wird von jeder Revision eine vollständige Kopie der Quellcodes, der compilierten und gelinkten Dateien, sowie Makefiles auf einer CD-ROM bzw. auf dem Netzwerk-Server hinterlegt.

Die Archivierung kann auch über entsprechende Werkzeuge automatisch erfolgen.

7.6 Variablen

Bei der Deklaration von Variablen wird ihnen eine eindeutige Semantik nicht aber Speichersyntax zuordnet. Beispiel hierfür ist der Datentyp *int* (vgl. Kap. 2.2.3), der je nach Implementierung eine Länge von 2 oder 4 Bytes hat. Daher sollte für ganzzahlige 2 Byte-Variablen nur der eindeutige Datentyp *short* und für den ganzzahligen 4 Byte-Typ *long* verwendet werden.
Desweiteren sollten zur eindeutigen Klärung der Bedeutung für vorzeichenlose Zahlen die Typen *byte*, *word* und *dword* verwendet werden (vgl. Kap. 2.2.2 und 2.2.3)
Bei Konstruktionen, die auf der Speichersyntax von Zahlen beruhen, sollte man vorsichtig sein, da diese nicht eindeutig ist. So werden Worte oder Fließkommazahlen im big-endian-Format (high order first, auch Motorola-Format genannt) genau in umgekehrter Reihenfolge abgelegt als Worte im little-endian-Format (low order first, auch Intel-Format genannt). Dies spielt besonders bei der Versendung von Variableninhalten über die serielle Schnittstelle oder der residenten Speicherung eine Rolle. Die byteweise Operation ist in diesem Fall eine sichere Methode.

7.6.1 Deklarationsschreibweise

7.6.1.1 Präfix

In der Windows-Programmierung ist man zur besseren Lesbarkeit von Programmen (insbesondere Datentypen von Variablen) zu folgender Übereinkunft gekommen (Quelle: C. Petzold, Programming Windows, 2. Auflage):
Zur Identifikation des Datentyps einer Variablen erhalten diese einen Präfix. Dieser besteht nur aus Kleinbuchstaben, wie z.B. 'd' für den Datentyp 'double'.

Eine Auflistung der wichtigsten Kennzeichner in alphabetischer Reihenfolge:

a.....***a***rray, Feld, Vektor
b.....***b***ool, boolean type
by.....***by***te, unsigned char (unsigned 8 Bit)
c.....***c***haracter, char, Zeichen (signed 8 Bit)
d.....***d***ouble
dw.....***d***ouble ***w***ord, (unsigned 32 Bit)
f.....***f***loat
fn.....***fu****n***ction, Funktion
h.....***h***andle (16-Bit-Identifikation) Bsp.: hFile = fopen(...);
i.....***i***ndex, Indexzähler (Verwendung nur innerhalb einer Bildschirmseite)
l.....***l***ong, long integer (signed 32 Bit)
lp.....***l***ong (far) ***p***ointer, 32-Bit-Adresse
lpsz.....long pointer to string zero terminated (Far-Zeiger auf Zeichenkette)
n.....i***n***teger, Ganzzahl (je nach Rechner 16 oder 32 Bit)
o.....sh***o***rt (signed 16 Bit)
p.....***p***ointer, Zeiger (Bsp.: double *pdSehrGenau;)
s.....***s***tring (Nicht unbedingt mit ´\0´ abgeschlossen)
sz.....***s***tring ***z***ero terminated, mit Null beendete Zeichenkette
t.....s***t***ruct, Struktur
u.....***u***nion
x,y.....short (als ***x***- und ***y***-Koordinate)
w.....***w***ord, (unsigned 16 Bit)
typ.....***w***ord, (unsigned 16 Bit)

Die Präfixe *a* für array und *p* für pointer werden den anderen Kennzeichnern vorangestellt.

7.6.1.2 Deklarationsebene

Zusätzlich zur Ungarischen Notation sollte anhand der Deklarationsschreibweise auch die Deklarationsebene bzw. Sichtbarkeit erkennbar sein. So werden Variablen mit größerer als der lokalen Sichtbarkeit groß geschrieben. Mitglieder von Strukturen und Unions sind wie lokale Variablen zu notieren. Beispiele:

```
double dSehrGenau;      /* lokale Deklaration */
double dSEHR_GENAU;     /* modulglobale Deklaration (nur Großbuchstaben) */
double d_SEHR_GENAU;    /* projektglobale " (Nach Datentyp: Unterstrich) */
```

Makrodefinitionen sowie die Special Function Register der Controller werden immer nur mit Großbuchstaben und Unterstrich geschrieben,:

```
#define BUS_ERROR   5                    /* Makrodefinition */
#define MAX(a,b)   (((a)>(b))?(a):(b))   /* Makrofunktion */
```

7.6.1.3 Datentypen

Elementare Datentypen, bei denen es nur um eine Umbenennung geht, in der jedoch der Sinn allgemein bleibt und der Typ nicht veränderbar ist, bekommen einfach den neuen Namen. Z.B.:

```
typedef unsigned char   byte;
typedef unsigned int    word;
```

Die Präfixe für die Variablennamen verändern sich hierbei nicht:
Variablen des Typs *unsigned char* entsprechend dem Typ *byte* erhalten ein *by* als Präfix.

Datentypen, bei denen Sinn der Umdefinition eine zentrale Änderbarkeit des Zwecks der Variable ist, (elementare Datentypen und abgeleitete Datentypen wie Strukturen oder Unions) bekommen ihren Namen mit Zeichenfolge *typ* als Präfix. Z.B.:

```
typedef byte typTimer;
typedef struct
    {
    byte byAv;
    word wAvSum;
    byte byFiltertiefe;
    }typTiefpass;
```

Die Präfixe für die Variablennamen erhalten den zum Typ gehörenden Präfix, z.B.:

```
typTiefpass tTiefpass1;
typTimer byTimer1, byTimer2;
```

7.6.2 Globale Variablen

Der besseren Übersichlichkeit eines Programms und um dem Prinzip der schmalen Datenkopplung zu entsprechen, sollten globale Variablen grundsätzlich nur sehr sparsam eingesetzt werden. Für globale Variablen sollte die Sichtbarkeit nur soweit ausgedehnt sein, wie es unbedingt nötig ist (vgl. Kap.7.6.1.2). Globale Variablen, die nur innerhalb eines Moduls verwendet werden, könnten den zusätzlichen Deklarator *static* erhalten.
Unter Umständen kann es vorkommen, daß eine globale Variable an einer ganz bestimmten Speicherstelle liegen soll (z.B. in einem EEPROM). Die Adresszuordnung für derartige Variablen sollte nicht erst durch entsprechende Anweisungen beim Linken erfolgen, sondern zur Compilierzeit durch Defines festgelegt werden. Diese Adressen bzw. Offsets für globale Variablen in solchen Speicherbereichen werden an einer Stelle in einem Header definiert und können bei Änderungen dort zentral modifiziert werden. Durch das Einbinden des Headers von Modulen, die diese Variablen benötigen, sind die Adressen für alle Module nach dem Neuübersetzen auf gleichem Stand.

7.7 Beispiele von Quellcode-Dateien

7.7.1 Beispiel: C-File

```
/*****************************************************************************/
/* Vollständiger Firmenname                                                  */
/*---------------------------------------------------------------------------*/
/* Projekt: Projektbezeichnung                                               */
/*                                                                           */
/* Datei:   kopf.c                                                           */
/*                                                                           */
/* Aufgabe:                                                                  */
/*                                                                           */
/* System:  Controller?   ; Compiler-Version                                 */
/*---------------------------------------------------------------------------*/
/* Funktionen:                                                               */
/* void main (void)                                                          */
/* int Muster(byte byWert)                                                   */
/*---------------------------------------------------------------------------*/
/* Er   | 01.10.01 | erstellt                                                */
/*      |   .  .01 |                                                         */
/*      |   .  .01 |                                                         */
/*      |          |                                                         */
/*      |          |                                                         */
/*****************************************************************************/
#ifndef __KOPF_C_
#define __KOPF_C_


#include "compiler.h"
#include "kopf.h"


void main (void)
/*****************************************************************************/
/* Diese Funktion macht alles                                                */
/* Er   | 01.10.01 | erstellt                                                */
/* Er   | 26.10.01 | Änderung auf 2. Fassung                                 */
/*****************************************************************************/
{
}


int Muster (byte byWert)
/*****************************************************************************/
/* Diese Funktion verarbeitet byWert zu Apfelmus?                            */
/* Er   | 01.10.01 | erstellt                                                */
/*****************************************************************************/
{
}


#endif /*__KOPF_C_*/
/*****************************************************************************/
/*****************************************************************************/
```

7.7.2 Beispiel: H-File

```
/*****************************************************************************/
/* Vollständiger Firmenname                                                  */
/*---------------------------------------------------------------------------*/
/* Projekt: Projektbezeichnung                                               */
/*                                                                           */
/* Datei:   kopf.h                                                           */
/*                                                                           */
/* Aufgabe: Headerfile für "kopf.c" / Headerfile zu Baustein...              */
/*                                                                           */
/* System:  Controller?   ; Compiler-Version                                 */
/*---------------------------------------------------------------------------*/
/* Funktionen: keine                                                         */
/*---------------------------------------------------------------------------*/
/* Er   | 01.10.01 | erstellt                                               */
/*****************************************************************************/
#ifndef __KOPF_H_
#define __KOPF_H_

/*---------------------------------------------------------------------------*/
/* projektglobale Variablen                                                  */
/*---------------------------------------------------------------------------*/
#ifdef __KOPF_C_
#else
#endif /*__KOPF_C_*/

/*---------------------------------------------------------------------------*/
/* Hardware-Definitionen                                                     */
/*---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------*/
/* allgemeine Definitionen                                                   */
/*---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------*/
/* Datentypendefinitionen                                                    */
/*---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------*/
/* Funktionsmakros                                                           */
/*---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------*/
/* Funktionsprototypen mit funktionsbezogenen Definitionen                   */
/*---------------------------------------------------------------------------*/

#ifdef __KOPF_C_
/*---------------------------------------------------------------------------*/
/* Lokale Funktionen und Modulglobale Variablen                              */
/*---------------------------------------------------------------------------*/

/*---------------------------------------------------------------------------*/
#endif /*__KOPF_C_*/

#endif /*__KOPF_H_*/
/*****************************************************************************/
/*****************************************************************************/
```

7.7.3 Beispiel: compiler.h

```
/*****************************************************************************/
/* Vollständiger Firmenname                                                  */
/*---------------------------------------------------------------------------*/
/* Projekt:                                                                  */
/*                                                                           */
/* Datei:   compiler.h                                                       */
/*                                                                           */
/* Aufgabe: Compilerspezifische Einstellungen                                */
/*          Grundsätzliche Definitionen                                      */
/*                                                                           */
/* System:  Controller?     KEIL-C51 V5.10                                   */
/*---------------------------------------------------------------------------*/
/* Funktionen: keine                                                         */
/*---------------------------------------------------------------------------*/
/* Er   | 01.10.01 | erstellt                                                */
/*      |          |                                                         */
/*****************************************************************************/
#ifndef __COMPILER_H_
#define __COMPILER_H_

/*---------------------------------------------------------------------------*/
/* Präprozessoranweisungen                                                   */
/*---------------------------------------------------------------------------*/
#pragma CODE
#pragma DEBUG

/*---------------------------------------------------------------------------*/
/* allgemeine Definitionen                                                   */
/*---------------------------------------------------------------------------*/
#define FALSE   0
#define TRUE    1
#define LOW     0
#define HIGH    1

/*---------------------------------------------------------------------------*/
/* Datentypendefinitionen                                                    */
/*---------------------------------------------------------------------------*/
typedef unsigned char   byte    ;
typedef unsigned int    word    ;

/*---------------------------------------------------------------------------*/
/* Funktionsmakros                                                           */
/*---------------------------------------------------------------------------*/
#define HI_BYTE(VAL) (*((byte *)(&(VAL))  ))
#define LO_BYTE(VAL) (*((byte *)(&(VAL))+1))

#define X_ADR(DT,X) ((DT volatile xdata *) ((byte volatile xdata *) (X)))
#define X_BYTE(X) (*X_ADR(byte,X))
#define X_WORD(X) (*X_ADR(word,X))

#define MIN(A,B)    (((A)<(B))?(A):(B))
#define MAX(A,B)    (((A)>(B))?(A):(B))


#endif /*__COMPILER_H_*/
/*****************************************************************************/
/*****************************************************************************/
```

7.7.4 Beispiel: map.h

```
/*****************************************************************************/
/* Vollständiger Firmenname                                                  */
/*---------------------------------------------------------------------------*/
/* Projekt:                                                                  */
/*                                                                           */
/* Datei:   map.h                                                            */
/*                                                                           */
/* Aufgabe: Peripherieadressen und -Definitionen                             */
/*                                                                           */
/* System:  Controller?     KEIL-C51 V5.10                                   */
/*---------------------------------------------------------------------------*/
/* Funktionen: keine                                                         */
/*---------------------------------------------------------------------------*/
/* Er   | 01.10.01 | erstellt                                                */
/*      |          |                                                         */
/*****************************************************************************/
#ifndef __MAP_H_
#define __MAP_H_

/*---------------------------------------------------------------------------*/
/* Controller                                                                */
/*---------------------------------------------------------------------------*/
sbit P1_0 = P1^0;
#define TASTE_UP       !P1_0
#define ROT_LED         P3_4

/*---------------------------------------------------------------------------*/
/* Adresse des DA Wandlers                                                   */
/*---------------------------------------------------------------------------*/
#define OFFSET_SPG      X_BYTE(0x4000)      /* Offsetspannung X*10 mV */
#define MV  /10     /* Teiler für alle Angaben in mV    */
#define LED_STROM       X_BYTE(0x4001)      /* LED-Treiberstrom  X*141.5 µA */
#define UA  /141    /* Teiler für alle Angaben in µA    */



#endif /*__MAP_H_*/
/*****************************************************************************/
/*****************************************************************************/
```

8 Die Trickkiste

Neben dem vorausgegangenen ist dieses Kapitel eine Zusammenfassung der am meisten von mir in der Praxis verwendeten Ideen und Leitgedanken. Einige davon sind bereits in die vorausgegangenen Kapitel eingeflossen, weswegen sie hier allenfalls erwähnt werden.

8.1 Optimierungsstrategien

Es gibt eine Reihe von Punkten, die man bei C51 bezüglich der Performance bedenken sollte, bevor man in die "Tasten haut".

- Die Mikrocontroller der 8051-Familie sind 8-Bit-Maschinen mit integriertem Booleschem Prozessor (1 Bit). Es ist daher leicht zu ersehen, daß Operationen mit 8-Bit-Variablen (*byte, char*) und Bits erheblich effizienter sind als Operationen mit den Typen *int* oder *long*. Es sollte daher immer der kleinstmögliche Datentyp verwendet werden, also der, der den geforderten Darstellungsbereich gerade noch besitzt.
  Der C51-Compiler unterstützt alle Bit- und Byte-Operationen direkt, es werden daher keine Typenumwandlungen vorgenommen, wenn die Operanden dies nicht erforderlich machen. Ein anschauliches Beispiel stellt die Multiplikation zweier Variablen vom Typ *byte* dar; sie geschieht "inline" mit dem 8051-Befehl *MUL AB*, während der Typ *int* den Aufruf einer Bibliotheksfunktion notwendig macht.
- Die Mikrocontroller der 8051-Familie unterstützen keine Operationen mit vorzeichenbehafteten Zahlen. Der C51-Compiler muß daher entsprechenden Code generieren, um mit vorzeichenbehafteten Größen umzugehen. Code- und vor allem Laufzeiteinsparungen können erzielt werden, wenn nach Möglichkeit vorzeichenlose (unsigned) Variablen verwendet werden.
- Variablen, auf die häufig zugegriffen wird, sollten im internen RAM des 8051 platziert werden. Dies kann in jedem Speichermodell (COMPACT, LARGE) durch Angabe des Speichertyps (Kap. 2.4) durchgeführt werden. Zugriffe auf den internen RAM-Speicher des 8051 sind erheblich effizienter als Zugriffe auf den. externen Datenspeicher. Den internen Datenspeicher teilen sich die Registerbänke, der Bitbereich, der Stackbereich und die vom Anwender definierten Variablen mit dem Speichertyp *data* (Standard beim SMALL-Modell). Aufgrund des begrenzten internen RAM-Speichers von 128 bis 256 Byte je nach verwendetem 8051-Derivat müssen Kompromisse bezüglich Zugriffs-Effizienz und Anzahl der internen Objekte geschlossen werden.
- Der Compiler versucht, lokale Variablen in Registern zu halten. Dabei werden Schleifenindexvariablen bevorzugt. Dieser Optimierungsschritt wird nur bei lokalen Variablen durchgeführt. Die beste Wirkung erzielen auch hier wiederum vorzeichenlose Variablen.
- Zeiger sollten der Lauzeiteffizienz halber speicherbereichspezifisch sein (vgl. Kap. 2.12.3)
- Die Indizierung von eindimensionalen Arrays ist noch relativ schnell für einen 8051-Controller zu bewältigen. Deswegen sollte bei schnellen Applikationen darauf geachtet werden, daß die Zahl der Dimensionen möglichst bei 1 bleibt.
- Da die Ressourcen sind sowieso nicht endlos üppig sind, sollte das aktive Warten vermieden werden. Die 8051-Controller verfügen über zahlreiche Interruptquellen, die man nutzen sollte (z.B. Warten auf AD-Wandlung oder bis ein Zeichen über die serielle Schnittstelle versendet ist). Sinnvoll ist die Benutzung der Timer, zumindest eines Timers (Basistimer), von dem alle anderen Zeiten abgeleitet werden können (z.B. 1ms oder 10ms) (s. Kap. 8.2.2). Um z.B. die Timeouts bei Kommunikationsroutinen zu realisieren, sollten diese daher in state machines formuliert sein.

8.2 Makrosammlung

Wie schon in Kapitel 7.2.3 erwähnt, ist die Verbalisierung eine wichtige Strategie ein Programm lesbar zu machen. Makros dienen kaum einem anderen Zweck. Hier ist eine Liste von Makros dargestellt, die speziell auf die Architektur der 8051-Mikrocontrollerfamilie abgestimmt ist. Diese Makros arbeiten fast alle optimal, d.h. es wird ein Maschinencode erzeugt, der nicht effizienter in Assembler programmiert werden kann.

8.2.1 Direkter Speicherzugriff

Der direkte Speicherzugriff für memory-mapped Hardware ist in Kap. 2.13.4 bereits erörtert.

Ein weiterer direkter Zugriff ist der auf Variablen, bzw. Teile davon. Wie oben gesagt, sind die 8051-Controller 8-Bit-Maschinen. Entsprechende Zugriffe auf Variablen werden mit folgenden Makros realisiert:

```
#define HI_BYTE(VAL)   (*((byte *)(&(VAL))  )) /* Bestimmung eines Bytes */
#define LO_BYTE(VAL)   (*((byte *)(&(VAL))+1)) /* aus einem Wort */
#define HI_WORD(VAL)   (*((word *)(&(VAL))  )) /* Bestimmung eines Wortes */
#define LO_WORD(VAL)   (*((word *)(&(VAL))+1)) /* aus einem Doppelwort */
#define DW_BYTE0(VAL) (*((byte *)(&(VAL))+3)) /* Bestimmung eines Bytes */
#define DW_BYTE1(VAL) (*((byte *)(&(VAL))+2)) /* aus einem Doppelwort */
#define DW_BYTE2(VAL) (*((byte *)(&(VAL))+1))
#define DW_BYTE3(VAL) (*((byte *)(&(VAL))+0))
```

Entsprechendes für Konstanten

```
#define CHI_BYTE(KON)   ((byte)((KON)>> 8))    /* Bestimmung eines Bytes */
#define CLO_BYTE(KON)   ((byte)((KON)    ))    /* aus einem Wort */
#define CHI_WORD(KON)   ((word)((KON)>>16))    /* Bestimmung eines Wortes */
#define CLO_WORD(KON)   ((word)((KON)    ))    /* aus einem Doppelwort */
#define CDW_BYTE0(KON) ((byte)((KON)    ))    /* Bestimmung eines Bytes */
#define CDW_BYTE1(KON) ((byte)((KON)>> 8))    /* aus einem Doppelwort */
#define CDW_BYTE2(KON) ((byte)((KON)>>16))
#define CDW_BYTE3(KON) ((byte)((KON)>>24))
```

Schnelles inkrementieren eines Doppelwortes:

```
#define INC_DW(D) \
if(!(++DW_BYTE0(D)))if(!(++DW_BYTE1(D)))if(!(++DW_BYTE2(D)))++DW_BYTE3(D)
```

Da bei C-Compilern für den PC meist das little-endian-Format verwendet wird und somit umgekehrt zu der von C51 ist, ist für den Zugriff auf den "gleichen Speicher" eine Anpassung der Speichersyntax erforderlich, also eine Vertauschung der Bytefolge nötig (vgl. Kap. 7.6). Wird zum Beispiel eine Struktur 1:1 über die serielle Schnittstelle versendet, dann müssen deren Mitglieder "umgedreht" werden.
Vertauschen der Reihenfolge von Bytes im Wort (B-Register benutzt):

```
#define SWAP_WORD(W)  B=HI_BYTE(W),HI_BYTE(W)=LO_BYTE(W),LO_BYTE(W)=B
```

Vertauschen der Reihenfolge von Bytes im Doppelwort (B-Register benutzt):

```
#define SWAP_DWORD(D) B=DW_BYTE0(D),DW_BYTE0(D)=DW_BYTE3(D),DW_BYTE3(D)=B,\
                      B=DW_BYTE1(D),DW_BYTE1(D)=DW_BYTE2(D),DW_BYTE2(D)=B
```

8.2.2 Bestimmung von Zeiten

Sehr viele Anwendungen sind von Zeiten abhängig. Beispiele sind das Blinken von LEDs, äquidistante Abtastungen von Zuständen oder AD-Wandlern, Signalgeneratoren, die Ausgabe von Informationen über die serielle Schnittstelle, usw. Um sich wenig Gedanken darüber zu machen, *wie* diese Zeiten realisiert werden, sondern sich vielmehr darauf verlassen zu können, daß die Zeiten garantiert korrekt sind, gibt man sie auch in zeitlichen Einheiten an wie "ms", "µs" usw. Hierzu bedarf es einiger unterstützender Makros, die als Ganzes ein System von Makros darstellen.

Ermittlung der Taktzyklenzahl in Abhängigkeit von Zeit und Quarzfrequenz
Hierzu definiert man in der Systemdatei z.B. map.h die Quarzfrequenz in kHz mit

```
#define F_OSZ 11059UL
```

Danach läßt sich mit dem Makro *US(X)* (steht für µs)

```
#define US(X)                ((X)*F_OSZ/12000UL)
/*                             |    |      |
Zeit in µS---------------------|    |      |
Oszillatorfrequenz in kHz----------|      |
12 Osz.perioden für 1 Zyklus *1000-------|*/
```

die Taktzyklenzahl abhängig von der Zeit in µs bestimmt werden. Das Makro läßt sich verwenden, um z.B. einen intern getakteten Timer oder ein Compare-Capture-Register zu laden. Zu beachten ist hierbei, daß keine utopischen Zeiten angegeben werden, die nicht mehr in die entsprechenden Register passen.

Timer laden
Um einen der als 16-Bit-Timer konfigurierten Timer T0, T1 oder T2 zu laden, kann man sich nach Benutzung des Makros

```
#define LOAD_TIMER(TH,TL,TI) TH=((-(TI))/256),TL=((-(TI))&0x00FF)
```

und der anschließenden Verwendung z.B. für Timer0 wie

```
TR0=0;
LOAD_TIMER(TH0,TL0,US(5000));
TR0=1;
```

sicher sein, daß dieser mit 5000µs geladen wird und TF0 nach dieser Zeit gesetzt ist. Wird für TF0 eine Interrupt-Routine geschrieben, dann hat man auf diese Weise eine Basiszeit von 5ms, von der sich alle weiteren Zeiten ableiten lassen.

Ausdruck von abgeleiteten Zeiten
Um dem vorausgegangenen Beispiel zu folgen, sollen nun verschiedene Timer eingerichtet werden, die auf der Basis von 5ms arbeiten. Es soll beispielweise alle 100ms das Zeichen '#' (Lattenzaun) ausgegeben werden, und eine LED mit 2Hz blinken.

```
sfr LED = P1^0;
word w_SYSTEM_TIMER=0;
byte by_LATTEN_SENDE_TIMER=0;
byte by_LED_TIMER=0;
#define MS /5
#define SEK *200

void main(void)
{
/* Hier Timer0 als 16-Bit-Timer initialisieren und IR freigeben */
FOREVER
    {
    if(!by_LATTEN_SENDE_TIMER)
        {
        by_LATTEN_SENDE_TIMER=100 MS;
        SBUF='#';
        }
    if(!by_LED_TIMER)
        {
        by_LED_TIMER=250 MS;
        LED^=1;
        }
    }
}

void Timer0(void) interrupt 1 using 2
{
TR0=0;
LOAD_TIMER(TH0,TL0,US(5000));
TR0=1;
w_SYSTEM_TIMER++;
if(by_LATTEN_SENDE_TIMER)
    by_LATTEN_SENDE_TIMER--;
if(by_LED_TIMER)
    by_LED_TIMER--;
}
```

Hier sind die beiden Makros *MS* und *SEK* neu, von denen *MS* verwendet wurde. Für die LED gilt: Der Ausdruck *250 MS* wird im Compiler ersetzt durch die Konstante *50*, d.h. Timer0() wird 50mal aufgerufen, bevor die Variable *by_LED_TIMER* Null ist. Da der Timer auf 5ms programmiert ist, ergeben sich die gewünschten 50*5ms=250ms, was den wiederkehrenden LED-Zustand alle 500ms also mit 2Hz herstellt.
Entsprechendes gilt für das Senden von '#' alle 100ms über *by_LATTEN_SENDE_TIMER*.
Werden die Makros in Ausdrücken mit Operatoren verwendet, so sind sie gem Kap. 6.2 in Klammern zu setzen.

Aktives Warten im µs-Bereich
Manchmal ist es erforderlich aktiv bestimmte Zeiten im µs-Bereich abzuwarten. Beispiele sind AD-Wandler (Sample&Hold, Konvertierung) oder Bausteine mit seriellen Eindraht-Protokollen, die auf einer einzigen Leitung die serielle Information und die Stromversorgung tragen.
Hier sind zwei Gruppen von Makros vorgestellt, die jeweils ihre Vor- und Nachteile haben:

WAIT_uS(X)

```
#define WAIT_uS(X)      {byte data byMyTim=((US(X)-2+2)/4); while(--byMyTim);}
/* Zeit umgerechnent in µS  -----------------+     ¦ ¦  ¦
   Differenz zw. 1maliger und x-maliger Schleife --+ ¦  ¦
   Für Rundung auf nächstbeste Möglichkeit: 4/2 -----+  ¦
   Zahl der Zyklen für x-maligen Schleifendurchlauf-----+
   Gilt nur für Optimierungsstufe 5 und wird compiliert zu:
    R     MOV     byMyTim,#XXH      2 cycl. ¦once 6 cycl.
    ?C0001:                                 ¦
    R     DEC     byMyTim           1 cycle ¦ + ¦(XX-1)-times 4 cycl.
    R     MOV     A,byMyTim         1 cycle ¦   ¦
          JNZ     ?C0001            2 cycl. ¦   ¦                    */
```

Dieses Makro hat den Vorteil, daß es eine gute Auflösung (bei 12MHz-Takt: 4µs) und eine geringe Toleranz hat. Dem gegenüber stehen einige Nachteile:
- Da es eine aktive Schleife mit lokaler Variable ist, verlängert eine in dieser Warteschleife auftretende Interruptroutine die Wartezeit um die Durchlaufzeit der Interruptroutine.
- Die Funktion ist nur dann gewährleistet, wenn auch der oben gezeigte Maschinencode entsteht. Dies hängt insbesondere von der Compilerversion, der Optimierungsstufe und dem nachfolgenden Code ab. D.h. bei Veränderung der Funktion, die dieses Makro enthält, ist der entstandene Maschinencode im Listfile zu überprüfen.
- Es gibt 8051-Derivate (Dallas 80C320, Intel 80C251, Cygnal C8051Fxxx), bei denen ein Maschinenzyklus nicht 12 Taktzyklen entspricht. Die Ausführungszeit ist somit so unzureichend bzw. kompliziert definiert, daß man in diesem Fall besser auf dieses Makro verzichtet.

LESE_TIMER(TH,TL,W), WAIT_FOR_TIMER(TH,TL,W), WAIT_FOR_uS_TIMER(TM)

Hierbei handelt es sich um eine Reihe von Makros, die gemeinsam das aktive Warten realisieren. Sie basieren auf der Voraussetzung, daß der verwendete Timer als 16-Bit-Timer (wie oben beschrieben) immer mitläuft. Die Zeit, auf die er eingestellt ist, spielt nur eine untergeordnete Rolle:

```
/* Zum Auslesen eines laufenden 16-Bit-Timers */
#define LESE_TIMER(TH,TL,W) \
    {(HI_BYTE(W)=TH),(LO_BYTE(W)=TL);\
    if(HI_BYTE(W)!=TH)(++HI_BYTE(W)),(LO_BYTE(W)=TL);}
/* Wertevergleich eines laufenden 16-Bit-Timers mit einer Variablen */
#define WAIT_FOR_TIMER(TH,TL,W) \
    {while(TH<HI_BYTE(W));while((TH==HI_BYTE(W))&&(TL<LO_BYTE(W)));}
/* kalibrierter µs-Timer basierend auf Timer T0 für Zeiten >= 30µs */
#define WAIT_FOR_uS_TIMER(TM) \
    {word data wUsT; LESE_TIMER(TH0,TL0,wUsT); \
    wUsT+=((word)US(TM)-8); WAIT_FOR_TIMER(TH0,TL0,wUsT);}
```

Diese Makros haben die Vorteile, daß sie weder vom verwendeten Derivat noch von der verwendeten Compilerversion oder von der Optimierungsstufe abhängig sind. Findet eine Unterbrechung der Schleife durch einen Interrupt statt, dann weicht die Zeit nur dann ab, wenn die Interruptroutine länger braucht, als die eingestellte aktive Wartezeit. Der Nachteil besteht in der schlechteren Auflösung, die sich nur mit höher werdender CPU-Geschwindigkeit verbessert.

8.2.3 IDLE-Makros

Für einen energiesparenden Einsatz eines 8051-Controller-Derivates ist es opportun, daß es in den sog. IDLE-Modus versetzt wird, wenn keine Aufgaben anliegen. Die Controller (genauer die CPU) können über das Flag IDLE (PCON.0) in diesen (Schlaf-)Zustand versetzt werden. Hierbei bleiben aller Timer, serielle Schnittstelle..., sprich die gesamte Hardware des Controllers außer die CPU aktiv. Die CPU wacht erst dann wieder auf, sobald ein Ereignis z.B. in Form eines Interrupts eintritt.
Möchte man erreichen, daß das Programm in der Hauptebene erst dann fortfährt, wenn ein selektierter Interrupt aktiviert worden ist, dann können die folgenden Makros verwendet werden (es hat gedauert, bis mir die folgende wasserdichte Methode eingefallen ist):

```
/* Globale Systemvariable */
byte volatile data by_PCON;
/* Prüfen, ob CPU im IDLE-Modus bleiben soll (Selektion) */
  #define ASLEEP   (by_PCON&0x01)
/* CPU selektiv in den IDLE-Modus versetzen */
  #define SLEEP()   for(by_PCON=PCON|0x01;ASLEEP;PCON=(by_PCON=PCON|0x01))
/* CPU selektiert aus dem IDLE-Modus holen */
  #define WAKE_UP() by_PCON&=~0x01
```

Beispiel:

```
void main(void)
{
word wTimerTicks=0;
/* Hier Timer0 als 16-Bit-Timer initialisieren und IR freigeben */
FOREVER
    {
    SLEEP();               /* Synchronisation mit Timer0() */
    wTimerTicks++;
    Putchar(HI_BYTE(wTimerTicks));
    Putchar(LO_BYTE(wTimerTicks));
    }
}

void Timer0(void) interrupt 1 using 2
{
TR0=0;
LOAD_TIMER(TH0,TL0,US(5000));
TR0=1;
WAKE_UP();
}
```

Wie im Beispiel ist eine Anwendung das Fortsetzen des Programms synchron mit dem Systemtimer (z.B. wie hier *Timer0()*). Das Makro *SLEEP()* versetzt die CPU in den IDLE-Modus durch setzen des IDLE-Flags aus by_PCON. Wird die CPU durch einen anderen als den *Timer0()*-Interrupt geweckt, dann legt sie sich sofort nach Ausführung dieses Interrupts wieder schlafen, weil das IDLE-Flag in by_PCON noch gesetzt ist, womit die *SLEEP()*-Schleife nicht verlassen wird, und PCON erneut das IDLE-Flag über by_PCON gesetzt bekommt. Die Schleife wird nur dann verlassen, wenn die synchronisierende Interruptroutine dadurch ausgewählt wird, daß dort das Makro *WAKE_UP()* aufgerufen und so das IDLE-Flag in by_PCON gelöscht wird. Im Beispiel ist *Timer0()* selektiert und dadurch wird in *main()* erreicht, daß die lokale Variable *wTimerTicks* alle 5ms ausgegeben wird.

8.2.4 Reset-Makros

Zum Abfragen und Definieren von Reset-Zuständen wird das General Purpose Flag GF1 im SFR PCON benutzt. Dabei wird die Tatsache ausgenutzt, daß dieses Flag nach einen Hardware-Reset zurückgesetzt ist. Es sind folgende Makros realisiert:

Zur Abfrage, ob ein Hardware-Reset vorlag:
```
#define HW_RESET            (!(PCON&0x08))
```

Zur Abfrage, ob ein Software-Reset vorlag:
```
#define SW_RESET            (PCON&0x08)
```

Zum Feststellen eines Hardware-Resets, nach dessen Aufruf unter den richtigen Voraussetzungen ein Software-Reset detektiert werden kann:
```
#define HW_RESET_DETECTED()   if(HW_RESET) PCON|=0x08
```

Erzeugung eines Software-Resets:
```
#define RESET()          (((void (code*)(void))0)())
```

Systemhalt durch setzen des STOP-Flags (kann nur durch einen HW-Reset aufgehoben werden):
```
#define HALT()             EA=0,PCON|=0x02
```

Ein paar Takte zum Makro *RESET()*, weil es auf den ersten Blick nicht ganz trivial ist:
Dieser C-Code erzeugt ausschließlich den Maschinenbefehl `LCALL 00H`. Es handelt sich hierbei um die Konstante 0, die explizit zu einem Zeiger auf eine void-void-Funktion ohne Namen umgewandelt und gleichzeitig aufgerufen wird (vgl. hierzu die Kapitel 2.12.6 und 3.8.5.). Nach dem Sprung an die Adresse 0 wird der Stack sowie die statischen/globalen Variablen initialisiert und *main()* angesprungen.
Beim Aufruf ist zu beachten, daß dieser nicht aus einer Interrupt-Routine erfolgen darf, weil der Hardware-Interrupt-Status der CPU aufgrund fehlenden Maschinenbefehls RETI nicht verlassen wird (vgl. Kap. 5.5). In diesem Fall würde die erneut aufgerufene Funktion *main()* im Interruptstatus laufen, was eine weitere Verarbeitung von Interrupts gleicher oder niedrigerer Priorität verhindert. Das Progamm wird zur Verwunderung nur teilweise erneut laufen.
Es ist anzumerken, daß für einen Aufruf einer Funktion an beliebiger Stelle X im Programm nach ANSI-Standard auch ein generic pointer (also `(((void(*)(void))X)()))`) genügen würde, dieser wird jedoch bis zur Version V7.00 des Keil-Compilers immer zu `LCALL 00H` übersetzt.

8.2.5 Watchdog-Makros

Um ein System sicherer zu machen, ist der Einsatz eines Watchdogs nahezu unerläßlich. Viele 8051-Derivate bieten die Möglichkeit einen internen Watchdog zu starten. Es ist sinnvoll die Retriggerung dieses Watchdogs als Makro zu formulieren, weil dieses dann an zentraler Stelle abgeschaltet werden kann, um z.B. mit Emulatoren zu arbeiten, ohne daß das Programm immer von vorne anfängt. Bevor ich einige Beispiele vorstelle, sei auf ein in diesem Zusammenhang stehendes Makro hingewiesen:

Feststellung, ob sich das System in einer Interruptroutine befindet:

```
#define IR_RUNNING          (RS0||RS1)
```

Denn: Ob interner oder externer Watchdog, dies spielt keine Rolle für den dringenden Rat diesen niemals in einer Interrupt-Funktion oder von einer Funktion zu triggern, die von einer Interuptroutine aufgerufen wird. Ein Programm wird unsicher, wenn z. B. der Watchdog von einem Timer-Interrupt aus getriggert wird, und dieser Interrupt früher auftritt, als Watchdog einen Reset auslösen würde. Das Hauptprogamm könnte abstürzen, der gewünschte Reset erfolgt jedoch nie.

Nun die Makros:

Ist auf den 16-Bit-Timer1 eine Interrupt-Funktion mit höchster Priorität codiert, können in dieser Funktion noch abschließende Maßnahmen eingeleitet werden, die z.B. nach einem Neustart eine entsprechende Ausgabe erzeugen. Die Retriggerung besteht dann aus dem erneuten Laden des Timers, um einen Timer1-Überlauf zu verhindern.

```
#define WATCHDOG()    if(!IR_RUNNING)LOAD_TIMER(TH1, TL1, US(10000))
```

Schlägt der Timer-Interrupt dann doch zu, dann kann die zugehörige Routine z.B. den Stack zwischenspeichern, der dann nach Neustart ausgegeben wird. Dieser kann dann manuell analysiert werden, um die Stelle des Absturzes zu lokalisieren.
Entsprechend können diese Protokollmaßnahmen auch ergriffen werden, wenn der Controller einen Interrupt höchster Prio (NMI oder Trap) unmittelbar vor dem folgenden HW-Reset durchführen kann.

Triggerung für einen 80C251-Controller

```
#define WATCHDOG()    if(!IR_RUNNING)WDTRST=0x1e,WDTRST=0xe1
```

Triggerung für einen 80C320-Controller

```
#define WATCHDOG()    if(!IR_RUNNING)EWT=1,RWT=1
```

Triggerung für einen 80C552-Controller

```
#define WATCHDOG()    if(!IR_RUNNING)PCON|=0x10,T3=0
```

Beispiel für eine Hardware-Triggerung

```
#define WATCHDOG()          if(!IR_RUNNING)WDOG_PIN=LOW,WDOG_PIN=HIGH
```

8.2.6 Weitere Makros

MIN(A,B) weist den kleineren der beiden Ausdrücke zu
```
#define MIN(A,B)        (((A)<(B))?(A):(B))
```

MAX(A,B) weist den größeren der beiden Ausdrücke zu
```
#define MAX(A,B)        (((A)>(B))?(A):(B))
```

ABS(A,B) bildet den Betrag des Ausdrucks, entsprechende Funktionen der Laufzeitbibliothek des C51-Compilers sind langsamer und somit überflüssig.
```
#define ABS(A)          (((A)>=0)?(A):-(A))
```

ABS_DIFF(A,B) bildet den Betrag der Differenz der beiden Ausdrücke
```
#define ABS_DIFF(A,B)   (((A)>(B))?(A)-(B):(B)-(A))
```

SIGN(A) liefert dem Vorzeichen des Ausdrucks entsprechend eine –1, 0, oder 1
```
#define SIGN(A)         (((A)<0)?(-1):(((A)==0)?0:1))
```

Die Gruppe der L_-Makros liefert einen auf den entsprechenden Datentypen begrenzten Wert, so daß ein Unter- bzw. Überlauf verhindert werden kann.
```
#define L_LIMIT(V,L,H)  (((V)>(H))?(H):(((V)<(L))?(L):(V)))
#define L_WORD(V)       L_LIMIT(V,0,65535)
#define L_BYTE(V)       L_LIMIT(V,0,255)
#define L_INT(V)        L_LIMIT(V,-32768,32767)
#define L_CHAR(V)       L_LIMIT(V,-128,127)
      Beispiel:
      word wWord=1000;
      byte byByte;
      byByte=L_BYTE(wWord);    /* byByte erhält den Wert 255 statt 232 */
```

Der Schreibzugriff auf Arrays, bei dem der Laufindex überläuft ist eine häufige Fehlerquelle (vgl. Kap. 2.11.3 und 7.1). Um den Laufindex zu begrenzen kann folgendes Makro eingesetzt werden:
```
#define ELEMENTE(VAR, TYPE) (sizeof(VAR)/sizeof(TYPE))
```

Die Endlosschleife
```
#define FOREVER while(1)
#define FOREVER for(;;)
```

In C existiert keine Kontrollstruktur, die den einmaligen Durchlauf eines Stück Codes ermöglicht, der unterwegs mit dem Befehl *break* abgebrochen werden kann. Hier schon:
```
#define DO_ONCE switch(0)default:
```

Hier passiert nichts (zur Hervorhebung statt eines einfachen Semikolons)
```
#define DO_NOTHING()
```

Zustände:
```
#define FALSE   0
#define TRUE    1
#define LOW     0
#define HIGH    1
#define AUS     0       /* Schalten von Portbits in positiver Logik */
#define EIN     1
#define _EIN    0       /* Schalten von Portbits in negativer Logik */
#define _AUS    1
#define OK      0
#define ERROR  -1
```

9 Entwicklungsumgebung

In den nächsten Abschnitten wird erläutert, wie man mit dem C51-Compiler arbeitet, um ein Programm erstellen zu können.

9.1 Programmtexteditor

Vom Compiler unabhängig ist die Verwendung eines guten Programmtext-Editors. Dies sei an dieser Stelle erwähnt, weil das Erzeugen eines Programms mit der Erstellung des Quellcodes beginnt. Damit man Ideen bei der Programmierung und die Gestaltung des Quellcode-Aufbaus umgehend in die Tat umsetzen kann, sollte man einen Editor benutzen, mit dem man schnell und effizient arbeiten kann.
Zum C51-Compiler gibt es eine Entwicklungsumgebung, die auch einen Edtior enthält. Die Praxis zeigt, daß man letztlich doch den Editor benutzt, den man bereits von anderer Programmierung her gewohnt ist, und wegen der dadurch effizienteren Arbeitsweise einsetzen möchte. Dies sollte auch in jeder Entwicklungsumgebung möglich sein, andernfalls ist diese Umgebung unzweckmäßig.
Hierbei ist es egal, ob der Editor eine DOS- oder eine Windows-Applikation ist. Er sollte mindestens folgende Eigenschaften besitzen:

- Freie Belegung zusätzlicher (Funktions-)Tasten (Makros)
- Umschaltung der Editorebene zwischen mindestens 20 Dateien.
- Zwischenablage (Clipboard) zum Kopieren von Teilen innerhalb einer und zwischen Dateien und anderen Anwendungen.
- Kopier-, Ausschneide-, und Einfügemöglichkeit von Textteilen nicht nur zeilenweise, sondern auch von beliebigen rechteckigen Blöcken.
- Dateiübergreifende Such- und Ersetz-Funktion.
- Unterstützung richtiger Klammernsetzung (mind. für runde und geschweifte Klammern).
- Freie farbige Darstellung von Schlüsselwörtern, Compileranweisungen, Kommentaren, Konstanten, Strings, markierten Blöcken usw.
- Möglichkeit des Hinzufügens von Schlüsselwörtern.
- Multiple Kontextrestaurierung beim Wiederaufruf für alle offenen Dateien aller vergangenen Sitzungen.
- Rückgängig- und Wiederherstell-Funktion für alle offenen Dateien mit mindestens 2000 Schritten pro Datei.
- Semigrafikzeichen-Editor für Grafiken in der Inline-Dokumentation
- Keine Erzeugung von Steuerzeichen in der Programmtextdatei.
- evtl. eine Dateivergleich-Funktion
- Einstellbarer TAB-Abstand und das Ersetzen von TABs durch Leerzeichen.
- Automatische Speicherung, vorzugsweise in "Sekunden nach letztem Tastendruck"

9.2 Installation von C51

Eine Voraussetzung für ein Arbeiten mit dem C51-Compiler ist seine richtige Installation.
Die folgenden Empfehlungen beziehen sich auf die Anwendung des C51-Compilers unter MS-Windows 9x. Für WinNT und Win2000 gelten insofern Einschränkungen, als daß der mitgelieferte Dongle spezielle Treiber benötigt.
Auch wenn es hier und da den Anschein hat, daß der C51-Compiler der Fa. KEIL eine MS-Windows-Software ist (z.B. bei Benutzung von µVision), so sollte man sich im klaren darüber sein, daß der Compiler, Linker usw. an sich DOS-Applikationen sind.

Es hat sich herausgestellt, daß die folgende Verzeichnisstruktur günstig ist:
Auf dem Laufwerk C: sollten sich ein Unterverzeichnis TEMP und ein Unterverzeichnis C51 befinden. Unterhalb von C51 können eine oder mehrere Versionen des C51-Compilers nebeneinander liegen, was den Vorteil bietet, zu älteren C-Programmen kompatible Binärcodes erzeugen zu können.

Verzeichnisse:
C:\TEMP
C:\C51\C51_320 und/oder
C:\C51\C51_510

Die zu aktivierende Compilerversion ergibt sich aus den gesetzten Umgebungsvariablen. Dies ist beispielsweise an den Eintragungen eines Batch-Jobs gezeigt, hier für die Compilerversion V3.20:

```
SET :WORK:=C:\TEMP
SET C51INC=C:\C51\C51_320\INC
SET C51LIB=C:\C51\C51_320\LIB
SET PATH=C:\C51\C51_320\BIN;%PATH%
```

In älteren Compilerversionen ist statt *C51INC* die Umgebungsvariable *:INCLUDE:* erforderlich.
Der besagte Batch-Job ist entweder die AUTOEXEC.BAT, ein Aufruf von der AUTOEXEC.BAT oder einer, der in dem DOS-Fenster aktiviert wird, in dem der Compiler anschließend läuft.

Wichtig ist in jedem Fall das Ergänzen des folgenden Eintrags in die Datei CONFIG.SYS, damit überhaupt noch weitere Umgebungsvariablen gesetzt werden können (hierbei wird angenommen, daß die Datei COMMAND.COM auf c:\ liegt):

```
SHELL=C:\COMMAND.COM /E:4096 /P
```

Nach diesem Eintrag ist ein Neustarten des PCs erforderlich.

9.3 Compileraufruf

Der C51-Compiler wird durch Eingabe von C51 gestartet. In der Aufrufzeile wird der Name der zu übersetzenden C-Quelldatei angegeben und wahlweise weitere Steuerparameter, die die Funktionsweise des Compilers beeinflussen.

Beispiel:

C51 TEST.C PP

Der Compiler meldet sich mit seiner Version und zeigt am Ende die Zahl der Warnungen und Fehler an. Der Compiler generiert, falls nicht durch Aufrufoptionen anders bestimmt, folgenden Dateien:

- TEST.LST:

Das Listing wird unter der Voraussetzung generiert, daß der Compiler mit dem Steuerparameter (oder auch Compileranweisung) PRINT aufgerufen wird. Das Listing enthält den formatierten Quelltext sowie Fehlermeldungen, falls im Quellprogramm Fehler enthalten sind.

- TEST.OBJ:

Die Objektdatei mit der Erweiterung ".OBJ" enthält relokatiblen Objektcode und DEBUG-Informationen zur weiteren Verarbeitung durch den Linker/Locater L51.

- TEST.I:

Die Datei mit der Erweiterung ".I" wird nur generiert, wenn die Aufrufoption PREPRINT (kurz PP) ohne Dateinamen angegeben wurde. In dieser Datei ist der vom Präprozessor expandierte Quelltext enthalten. Dabei sind alle Makros expandiert sowie alle Kommentare entfernt. Sie ist die der Übersetzung zu Grunde liegende Datei, aus der auch sämtliche Fehlermeldungen resultieren.

Der Compiler setzt die DOS-Environment Variable ERRORLEVEL, die das Ergebnis des Programmlaufs anzeigt. Der Variable wird in Abhängigkeit von der Fehlerursache wie folgt gesetzt:

0	keine Fehler (ERRORS), keine Warnungen (WARNINGS).
1	nur Warnungen (WARNINGS) aufgetreten.
2	Fehler (ERRORS) und möglicherweise auch Warnungen.
3	Fatale Fehler (FATAL ERROR).

Die Variable ERRORLEVEL kann in DOS-Batch-Dateien zur bedingten Abfragen (IF) verwendet werden, um beispielsweise die Batch-Verarbeitung im Fehlerfall zu beenden.

9.4 Zusatzprogramme

Der Compiler allein erzeugt noch kein lauffähiges Programm. Dazu sind noch weitere Hilfsprogramme nötig.

9.4.1 Linker/Locater L51

Nachdem die einzelnen Module vom Compiler übersetzt worden sind, sind als Produkt *.obj-Dateien entstanden, die nun zu einem Programm zusammengelinkt werden müssen. Beim Linken werden die vorhandenen Speicherbereiche belegt, d.h. eigener Programmcode, Bibliotheksfunktionen und Tabellen werden in den *code*-Bereich verteilt, relokatible Variablen werden auf den internen und externen RAM-Speicherbereich verteilt. Hierzu kann dem Linker mitgeteilt werden, wie groß die einzelnen Speicherbereiche sind. Ein Aufruf könnte so aussehen:

L51 test.obj, modul2.obj TO main CODE(2000H) XDATA (6100H) RAMSIZE(256) OVERLAY SYMBOLS LINES PUBLICS

Alternativ kann der Linker/Locater auch mit
L51 @main.lnk
aufgerufen werden, wobei angenommen wird, daß die restlichen Angaben aus dem Beispiel in der Datei *MAIN.LNK* stehen.
In dem Beispiel werden die Module *TEST* und *MODUL2* zu dem Programm-Modul *MAIN* zusammengeführt. Es wird angegeben, daß der Programmspeicher bei der Adresse 2000h und das *xdata*-RAM bei der Adresse 6100h beginnt. die Größe des internen RAMs (*data*- und *idata*-Bereich) ist mit 256 angegeben. Am Ende des benutzen *idata*-Bereichs wird vom Linker der Stack angelegt. Weiterhin wird angegeben, daß der Linker die Overlaytechnik einsetzen und sämtliche Debug-Information mit in die Datei aufnehmen soll.

Overlaytechnik
Der knappe interne Datenspeicher der 8051-Controller wird vom Linker/Locater L51 durch die sogenannte "Overlaytechnik" verwaltet. Dabei werden Parameter und lokale Variablen der in C kodierten Funktionen überlagert, wenn sich die Funktionen gegenseitig nicht aufrufen. Dies ist möglich, da in C lokale Variablen der Speicherklasse *auto* keinen Anspruch auf Erhalt des Inhaltes bei erneutem Aufruf haben. Dadurch wird der Speicherbedarf der Anwendungen um ein vielfaches reduziert. Um dieses Overlay-Verfahren durchzuführen, analysiert L51 die Referenzen (Aufrufe) zwischen den einzelnen Funktionen und überlagert dann gezielt die zu den Funktionen gehörigen Daten- und Bit-Segmente. Der Speicherbereich, in dem diese Überlagerung stattfindet, ist vom eingestellten Speichermodell abhängig. Bei der Einstellung SMALL wird eine *data-group* gebildet, bei *COMPACT* eine *pdata-group*, bei *LARGE* eine *xdata-group* gebildet. Bit-Segmente werden unabhängig vom Speichermodell zur *bit-group* überlagert. Durch den Steuerparameter '*OVERLAY*' können dabei Segmente von der Überlagerung ausgenommen oder Referenzen zwischen Funktionen entfernt bzw. hinzugefügt werden. Dies ist sinnvoll, wenn Funktionen indirekt aufgerufen werden oder die Speicherüberlagerung zu Debug-Zwecken unterbleiben soll. Der Steuerparameter '*NOOVERLAY*' schaltet die Überlagerung der Segmente vollständig ab. Dies kann allerdings schon bei Programmen mit ca. 30 Funktionen dazu führen, daß bei Verwendung des Speichermodells *SMALL* der interne Speicherbereich überläuft. Es versteht sich, daß statische Variablen nicht überlagert werden.

Der Linker/Locater erzeugt neben dem gewünschten Programm auch eine entsprechende Protokolldatei (in unserem Fall die Datei *MAIN.M51*), in der die Speicherbelegung aller Objekte und Symbole zu ersehen ist. Ebenso sieht man das Ergebnis der Referenzanalyse der Funktionen untereinander, das für die Overlaytechnik notwendig ist.

9.4.2 Objekt-Hex-Symbol-Konverter OHS51

Mit OHS51 können absolute Objekt-Dateien in Intel HEX-Dateien umgesetzt werden. Außerdem kann die Symbol-Information (Debug-Information) der Objekt-Datei in eine Symbol-Datei umgesetzt werden. Die Symbol-Datei wird wahlweise im Mikrotek-Format oder im Digital-Research-Format geschrieben. Zusätzlich kann die Symbolinformation auch in die Intel HEX-Datei am Anfang geschrieben werden.

Das Programm OSH51 wird, wie jedes andere Anwenderprogramm durch Eingabe seines Namens gestartet. Beim Programmaufruf wird der Name der Objekt-Datei angegeben.
In unserem Beispiel dieses Abschnitts lautet der Aufruf:

ohs51 main

Wahlweise können weitere Parameter angegeben werden, die den Ablauf von OHS51 steuern. Diese sind dem Handbuch zu entnehmen.

Das Intel-HEX-Datei-Format
Das Intel-HEX-Datei-Format ist neben dem reinen Binärformat das gebräuchlichste Format zum Datenaustausch mit Programmiergeräten. Die nachfolgende Tabelle erklärt die Reihenfolge und den Inhalt der Bytes in einem Hexadezimal-Record. Jeder Record enthält eine Prüfsumme (modulo 256), die aus der Null minus der Summe aller Informations-Bytes berechnet wird. Eine komplette Intel-HEX-Datei enthält einen oder mehrere Hexadezimal-Records. Die Datei endet mit einem End-of-File-Record.

Hexadezimal-Record Inhalte:

Position	0	1	2	3	4	5	6	7	8	9	10	...	n-4	n-3	n-2	n-1	n
Inhalt	:	1	1	a	a	a	a	t	t	d	d	...	d	c	c	CR	LF

Es bedeuten:

Symbol	Beschreibung
1 1	Recordlänge: Anzahl der Datenbytes
a a a a	Ladeadresse: 16-bit Startadresse
t t	Recordtyp: 00 = Datenrecord, 01 = EOF-Record
d d	Datenbytes: 8-Bit-Werte für den Code Bereich
c c	Prüfsumme: 0 - (Summe aller Recordbytes) MOD 256

Alle Datenrecords einer Intel-HEX-Datei haben den Recordtyp 00. Der Recordtyp 01 signalisiert das Ende der Datei (EOF). Der EOF-Record hat keine zusätzliche Information und ist immer ":00000001FF".

Beispiel einer Intel-HEX-Datei:

```
:060038007E7D1EEE70FC4F
:06003E007E7D1EEE70FC49
:080044000090D000E06F70E7AE
:01004C002291
:0300000002004DAE
:0C004D00787FE4F6D8FD758109020003FD
:00000001FF
```

9.4.3 Binär-File-Generator

Einige Programmiergeräte kennen das Intel-HEX-Format nicht, und benötigen daher das reine Binärformat. Das Binärformat ist ein exaktes Abbild der Daten, wie sie im Codespeicher des Controllers (z.B. EPROM) abgelegt sind. Das Binärformat ist außerdem erheblich sparsamer im Speicherplatzbedarf.
Daher ist es sinnvoll aus dem Intel-HEX-Format das Binärformat zu erzeugen.
Der Aufruf für unser Beispiel lautet:

hexobj main.hex main.bin INTEL

Es wird die Datei MAIN.BIN erzeugt.

9.4.4 Symbolpräprozessor für Debugger/Emulator

Um die für 8051-Debugger und -Emulatoren der Fa. HITEX nötige Information aus Programm-Objekt-Datei zu holen und entsprechend aufzubereiten, ruft man folgendes Programm auf:

sp8051ke main

Es wird die Datei main.htx erzeugt.

9.4.5 Maker

Um Übersetzungszeit einzusparen ist es besonders bei größeren Programmen (ca. 5 Module oder mehr) sinnvoll, nur die Module neu zu übersetzen, die seit der letzten Übersetzung im Quelltext verändert worden sind. Hierzu benutzt man sogenannte Maker. Sie können von unterschiedlichen Herstellern sein. Meist sind es Compilerhersteller bzw. solche, die eine komplette Entwicklungsumgebung mitliefern. Das "Wie" der Verwendung des Makers ist dem entsprechenden Handbuch zu entnehmen.

10 Stichwortverzeichnis

#

.

_

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z