Kapitel 1
Zu diesem Buch

1 Zu diesem Buch

Seit 1990 arbeite ich mit 8051-Mikrocontrollern und deren Derivaten. Zu dieser Zeit entwickelte ich auch die ersten Programme mit dem C51-Compiler der Firma Keil. Sehr schnell zeigte sich, daß sich C für Mikrocontroller nicht mit der gängigen C-Syntax vergleichen läßt. Zum ANSI-C sind controller-spezifische Eigenschaften hinzugekommen, z.B. Bitvariable, Speicherbereiche, Memory-specific-Pointer, SFR-Adressen, usw. .

1992 entstand eine erste Version dieses Buches. Zielsetzung war die Erstellung eines Nachschlagewerkes für den C51-Compiler ab Version 3.0 und dessen Einstellungen. Zudem sollte der Umstieg von einem Borland / Microsoft C auf den C51-Compiler erleichtert werden.

Dieses Buch soll Ihnen den Umstieg auf die neueren Versionen bis einschließlich 6.0 inklusive der Benutzeroberflächen µVision und µVision2 erleichtern. Viele Leserwünsche sind in dieses Buch eingeflossen.

Ich vergleiche das Programmieren in C oft mit dem Schachspielen. Was nützt es, einige unüberlegte Züge zu machen und nach kürzester Zeit mit dem Schachmatt konfrontiert zu werden. Aus diesen und anderen Erfahrungen heraus versuche ich, Ihnen die wesentlichen Regeln in C und den Einsatz des C51-Compilers nach und nach zu vermitteln.

Im ersten Teil des Buches gehe ich auf den C51-Compiler und auf den 8051 Befehlssatz ein. Übungen zu dem jeweiligen Thema sollen das Einarbeiten erleichtern. Um die Übungen praxisnah durchführen zu können, werden Evaluation Boards der Firma Keil, PHYTEC oder vom Elektronikladen verwendet. Die Übungen sind zudem so ausgelegt, daß Sie das beiliegende Keil-Demopaket (C51-Compiler, Simulator) verwenden können.

Im zweiten Teil des Buches werden Schaltungen behandelt. Diese sind so modular aufgebaut, daß Sie diese ohne zusätzlichen Aufwand miteinander verbinden können (z.B. Tastatur, Display, RTC).

Ein weiteres Thema in diesem Buch ist die Handhabung von Projekten unter µVision und µVision2 sowie der Umstieg von µVision nach µVision2.

Außerdem wird das Tool MKS angesprochen, das unter Windows ein professionelles Versions-Handling unterstützt. Dieses Tool läßt sich unter µVision2 Step 2* mit in die Oberfläche integrieren. Die Anbindung von MKS an µVision2 wird in Teil 2 besprochen.

*Step 2 wird voraussichtlich Mitte 99 erscheinen.

1.1 Beschreibungskonventionen

Um die Übersichtlichkeit für den Leser zu gewährleisten, wurden verschiedene Gestaltungsformen verwendet. Somit haben Sie einen leichteren Überblick und können schneller an einzelne Informationen gelangen.

Ausdruck Bezeichner 1 Bezeichner n

– **Ausdruck**: Beschreibung des Ausdrucks
– **Bezeichner 1**: Beschreibung von Bezeichner 1
– **Bezeichner n**: Beschreibung von Bezeichner n

Abbildung 1 Syntaxaufbau

Die Syntax eines Befehles oder eines Steuerparameters wird mit einer Beschreibungsregel dargestellt (siehe Abbildung 1 dunkelgrau hinterlegt). Der Ausdruck, um den es sich handelt, wird fett dargestellt. Der Ausdruck sowie die Bezeichner werden in dem darunterliegenden Abschnitt erklärt (hellgrau hinterlegt).

In dem Buch sind die einzelnen Kapitel bzw. Abschnitte mit graphischen Symbolen versehen. Diese zeigen den Informationsgehalt an. Hier finden Sie eine Erklärung zu allen im Buch verwendeten Symbolen.

	In diesem Kapitel oder Abschnitt werden Grundkenntnisse zu den Tools (C51-Compiler, µVision Oberfläche) vermittelt.
	In diesem Kapitel bzw. Abschnitt finden Sie allgemeine Grundlagen in C. Zudem wird auf C51-Besonderheiten eingegangen.
	In diesem Kapitel bzw. Abschnitt werden fortgeschrittene Techniken in C bzw. C51 erklärt.
	Dieses Kapitel bzw. dieser Abschnitt enthält Tips und Tricks, wie Sie den Code vereinfachen können, wie Sie Problemstellen umgehen oder effektiv programmieren.
	Diese Kapitel enthalten alle wichtigen C51-Informationen, wie z.B. Steuerparameter und deren Verwendungszweck.
	Wird dieses Symbol angegeben, so befindet sich an dieser Stelle ein (in sich abgeschlossenes) Beispiel oder eine Übung.
	Wenn Sie dieses Symbol im Kapitel finden, dann sollten Sie diese Informationen unbedingt beachten, um Fehler zu vermeiden.
☞	Dieses Symbol wird als Tip-Marker verwendet. Hier wird ein Verfahren oder ein Trick angegeben.
NEW	Alle mit NEW gekennzeichneten Beschreibungen sind ab Version 6.0 enthalten.

Tabelle 1 Erklärung der verwendeten Symbole

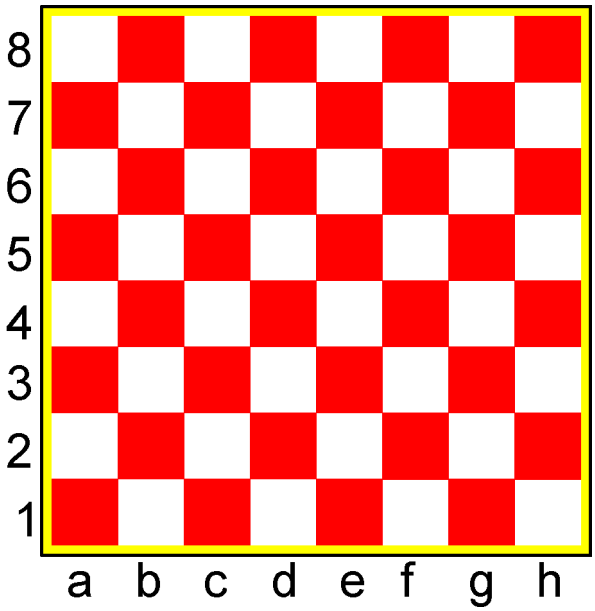
1.2 Glossar

Erklärung technischer Begriffe

- **Adresse**: Zahl oder symbolischer Name zur Identifizierung eines Registers, Speicherworts, Speicherbereichs, Eingabe- / Ausgabe-Kanals.
- **ANSI-C**: Eine Defintionsnorm für die Programmiersprache C. ANSI-C wurde 1989 vom **A**merican **N**ational **S**tandards **I**nstitute definiert.
- **Anweisung**: Eine Anweisung kann eine Operation oder ein Funktionsaufruf sein. Jede Anweisung wird mit einem Semikolon abgeschlossen.
- **Argument**: Werden bei einem Funktionsaufruf Werte übergeben, so spricht man von einer Argumentübergabe.
- **A51-Assembler**: Ein Programm zur Übersetzung eines in Assemblersprache geschriebenen Programms in ein auf dem Zielprozessor ablauffähiges Maschinenprogramm.
- **Compiler**: Ein Programm zum Übersetzen von Source-Modulen in einen Objektcode.
- **Cross-Compiler**: Ein Compiler, der aus Source-Modulen einen Objektcode für andere Prozessoren erzeugt.
- **Definition**: Bei der Definition einer Variablen, Funktion, usw. werden vom C-Compiler Speicherplatz z.B., im RAM, ROM, reserviert.
- **Deklaration**: Einem Source-Modul wird eine Variable oder Funktion bekannt gegeben. Bei einer Deklaration wird kein Speicherplatz reserviert.
- **Funktion**: Eine Funktion ist eine in sich abgeschlossene Operation. Die Funktion wird über ein CALL aufgerufen und mit einem RET abgeschlossen.
- **Harvard-Architektur**: Die Speicherbereiche vom ROM und RAM werden mit eigenen Befehlsgruppen verarbeitet. ROM und RAM können denselben Adreßraum verwenden.
- **von Neumann Architektur (VNM)**: Die Speicherbereiche ROM und RAM werden mit derselben Befehlsgruppe abgearbeitet.
- **H-File (Header-File)**: In diesem File stehen alle Definitionen und Deklarationen, die ein Programm benötigt.
- **Ini-File**: Mit diesem File können Kommandos an dScope über das Command-Fenster eingelesen werden.
- **I-File (Preprint-File)**: Zusätzliches Output-File des C51-Compilers, wenn der Steuerparameter PREPRINT verwendet wurde.
- **BL51-Linker**: Ein Programm zum Binden von OBJ-Files zu einem ablauffähigen Programm. Der Output vom Linker ist ein ablauffähiges Programm.
- **LNK-File**: Eingabefile für den Linker. In diesem sind die Informationen enthalten, welche OBJ-Files und welche Libraries mit hinzugebunden werden sollen. Zudem können in ihm Angaben enthalten sein, wie die einzelnen Programmsegmente im Speicher abzulegen sind.
- **LST-File**: Compiler und Assembler liefern als Output ein File mit der Erweiterung LST. Dieses File enthält in Abhängigkeit von Steuerparametern Informationen über den Verlauf einer Compilierung bzw. Assemblierung.
- **main()**: Jedes C-Programm wird mit dieser Funktion gestartet. Dies gilt auch für die C51-Umgebung.

- **Makro**: Ein im Sourcecode erzeugter Ausdruck, der während des Compilerlaufes in einen Sourcecode expandiert wird.
- **M51-File**: Der Linker erzeugt beim Linken ein File mit der Erweiterung M51. In ihm sind alle Informationen über die Aufteilung des Adreßraumes sowie den benötigten Speicherbereich vorhanden.
- **OBJ-File**: Der Output von Compiler und Assembler. Dieser Output wird dem Linker als Input zugefügt.
- **Operation**: Arithmetischer oder logischer Ausdruck
- **#pragma Steuerparameter**: Sie beeinflussen die Tools, z.B. C51-Compiler bei der OBJ-Code Erstellung oder den BL51-Linker bei der Adreßvergabe. Diese Steuerparameter sind Schlüsselwörter.
- **PRJ-File**: Unter µVision werden die Informationen von Projekten in diesen Files abgespeichert. Es handelt sich bei diesen Files um ASCII-Dateien.
- **RAM**: Random Access Memory
- **REG-FILE:** Dieses File wird vom BL51/L51 generiert und dient als zweite Optimierungsstufe beim C51-Compiler. In diesem File sind Informationen über die verwendeten Register enthalten.
- **ROM**: Read Only Memory
- **Schlüsselwort**: Der C-Compiler hat einige englische Wörter reserviert. Diese dürfen nicht anderweitig verwendet werden.
- **SFR**: Special-Function-Register
- **dScope-Simulator**: Der Simulator ist ein Programm, das einen Zielprozessor mittels Software nachbildet. Es können damit Programme ohne die Zielhardware getestet werden.
- **Source-Modul**: Ein File, in dem der Sourcecode enthalten ist. Ein Programm wird meist in mehrere Module aufgeteilt. Ein Modul sollte nicht mehr als fünf DIN A4 Seiten Sourcecode enthalten, da alles darüber hinausgehende unübersichtlich wird.
- **SRC-File**: Über den Steuerparameter SRC erzeugt der C51-Compiler als Output eine Assemblerdatei. Dieses Modul hat die Extension SRC. Dieses Modul kann direkt als Input für den A51 Assembler genommen werden.
- **XRAM**: eXternal Random Access Memory

Kapitel 2
Einführung



2 Einführung

*Was ist Schach? Schach ist ein aus dem Orient nach Europa gekommenes altes Brettspiel. Es wird auf 64 (Aufbau 8*8) abwechselnd weißen und schwarzen Quadratfeldern gespielt. Die Spalten haben die Bezeichnung A - H, die Zeilen die Bezeichnung 1 - 8. Das Feld A1 hat die Farbe schwarz. Gespielt wird dieses Spiel von zwei Spielern.*

Ein Compiler ist ein Programm zur Übersetzung eines in einer höheren Programmiersprache geschriebenen Programms in einen Objektcode. Dieser ist noch nicht ablauffähig, da ihm wesentliche Teile wie die Adreßfestlegung usw. fehlen. Damit das Programm ablauffähig wird, muß noch ein Linker/ Locater verwendet werden. Der Linker fügt Programmteile aneinander, wandelt die relativ auf das Modul bezogenen (relokatiblen) Adressen in absolute Adressen um und bindet, falls notwendig, Routinen aus den Libraries hinzu. Beim C51-Compiler handelt es sich um einen Cross-Compiler. Von einem Cross-Compiler wird gesprochen, wenn der Code, der vom Compiler erzeugt wird, nicht auf dem System ablaufen kann, auf dem er erstellt wurde. Alle hier abgebildeten Beispiele finden Sie auf der beiliegenden CD im jeweiligen Kapitel wieder.

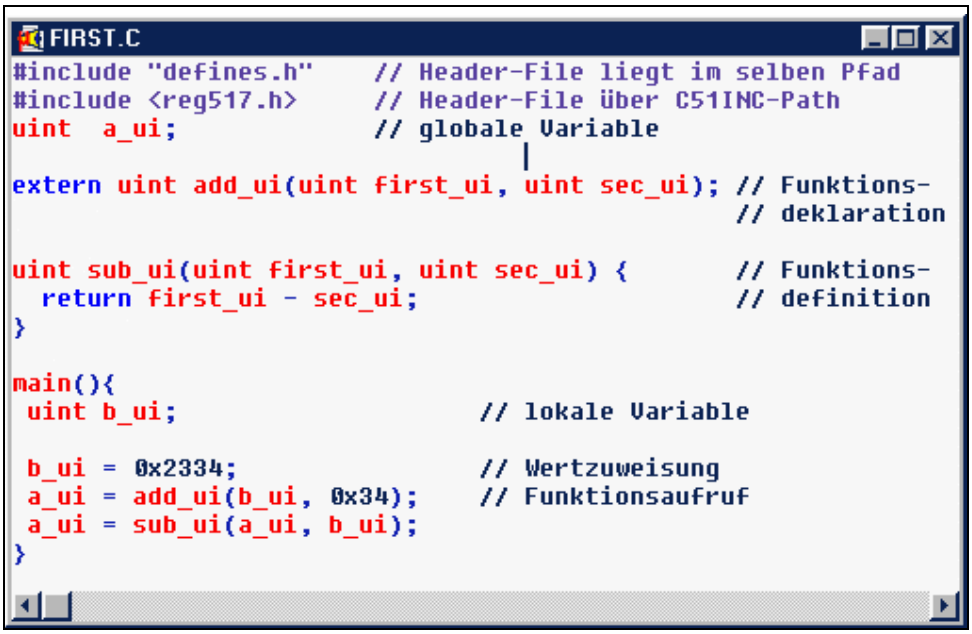
2.1 Aufbau eines Programms

Für die ersten Übungen mit dem C-Compiler sollten Ihnen folgende Ausdrücke geläufig sein:

Source-Modul, H-File, Makro, Definition, Deklaration, lokal, global, public, extern.

Ist dies der Fall, so können Sie dieses Kapitel übergehen und bei Kapitel 2.2 wieder einsteigen.

Ein Programm besteht aus mindestens einem Source-Modul. Dieses Source-Modul enthält seinerseits wieder den Quelltext. Dieser kann in einer Hochsprache, wie z.B. C, PASCAL, Basic oder in Assembler, geschrieben sein. Jede Hochsprache, so auch C, hat bestimmte Regeln, wie ein solches Source-Modul aufgebaut sein muß. In Listing 1 ist das Source-Modul **first.c** abgebildet. Dieses Modul beinhaltet alle Ausdrücke, die in diesem Kapitel mit Ausnahme der Funktionen besprochen werden. Zudem soll das „Learning by doing“ nicht zu kurz kommen. Neben den Erklärungen sind Übungen enthalten, die einen leichten Einstieg in die Oberflächen µVision bzw. µVision2 ermöglichen sollen. Öffnen Sie unter µVision über **File \Open** das Source-Modul **first.c**. Es befindet sich auf der CD unter dem Verzeichnis C51_Buch\ Kap_2 \examples\. Falls Sie das Softwarepaket von der Firma Keil noch nicht installiert haben sollten, müssen Sie dies zuerst durchführen. In Kapitel 17 finden Sie die Installationsanweisungen für die einzelnen Software Versionen.



Listing 1 Beispiel First.c

Nachdem Sie **First.c** unter µVision geladen haben, sollten Sie den Sourcecode in verschiedenen Farben dargestellt bekommen. Falls nicht, so klicken Sie in der Steuerleiste von µVision den Button **Color Syntax**. Mit den Farben werden schon wesentliche Informationen über den Inhalt des Source-Moduls gegeben. Alle C-Schlüsselwörter (siehe Kapitel 3.1), wie z.B. **extern**, **return,** werden in blau, alle Preprozessor-Anweisungen (siehe Kapitel 3.2), wie z.B. **#include** in violett, Konstanten in dunkelblau und die Kommentare in grün dargestellt. Die Farben können Sie unter der Oberfläche µVision über das **Menü Options\Editor Colors\Color Syntax** verändern. Die Bedeutung der einzelnen Bezeichnungen im Source-Modul wird in den nachfolgenden Abschnitten erklärt.

H-File (Header-File)

In den ersten zwei Zeilen von Listing 1 finden Sie in der Kommentarsparte Header-File. Was verbirgt sich hinter einem H-File? Was nutzt ein Programm davon und wie setzt man ein H-File sinnvoll ein?

Das englische Wort Header bedeutet Kopf und gibt schon einen ersten Hinweis, wie das File eingesetzt werden sollte. H-Files enthalten Informationen, die beim Start des C-Compilers bekannt sein müssen. Der Preprozessor (Teil des Compilers) bindet beim Compilerlauf das H-File an die Stelle des Sourcecodes ein, an dem der Verweis auf das H-File über die **#include** Anweisung angegeben ist. Der C-Compiler arbeitet nach dem Prinzip der Vorwärtsdeklaration. Alles, was dem C-Compiler bekannt gegeben wird, kann vom C-Compiler verarbeitet werden. Alles andere erzeugt Fehler. In den H-Files werden Definitionen, Deklarationen und **#pragma** Steuerparameter eingetragen, die eine ordnungsgemäße Compilierung ermöglichen. Sie können die Pflege ihres Programmes erleichtern, wenn die Einstellungen zentral in einem H-File

vorhanden sind und verschiedene Source-Module auf dieses zugreifen (siehe Abbildung 2).

☞ !! H-Files sollten nach den #pragma Steuerparametern des C-Compilers angegeben werden. !!

☞ !! H-Files haben die Extension **h**. !!

☞ !! Zu jedem Source-Modul sollte ein H-File mit gleichem Namen vorhanden sein. Somit können alle Public Informationen sofort abgefragt werden. !!

💣 !! Es dürfen **keine** #pragma Steuerparameter in H-Files stehen, da es sonst zu unübersichtlich wird (Ausnahme SAVE, RESTORE, und REGPARMS). !!

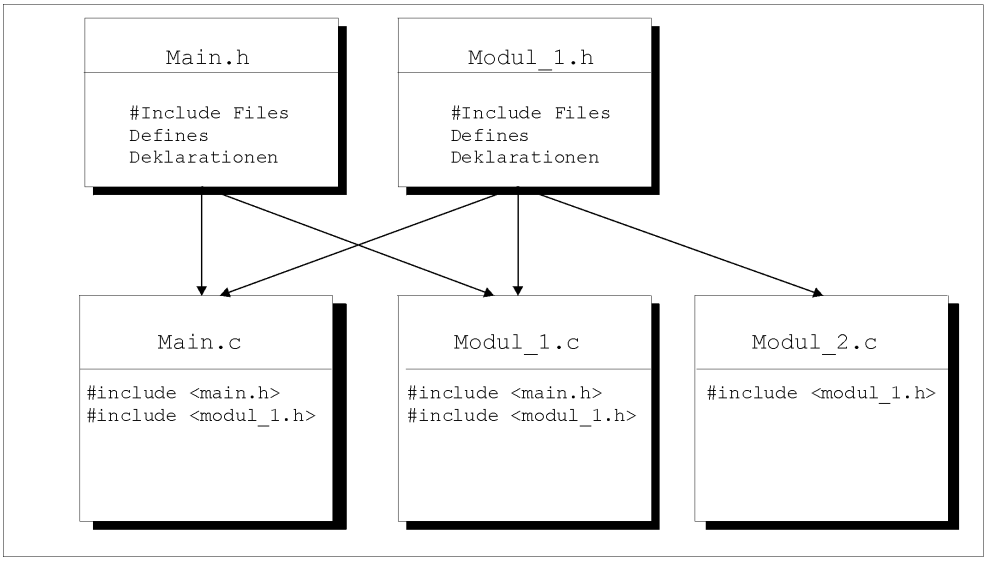


Abbildung 2 Aufbau eines Programms in Modulen

In Abbildung 2 befinden sich zwei H-Files, die Informationen für die Source-Module enthalten. Die Pfeile geben die Zusammenhänge zwischen den Modulen und den H-Files an.

☞ !! Ein H-File sollte nur folgende Informationen enthalten: !!

– Deklarationen von Funktionen und Variablen
– Definitionen von Konstanten
– Strukturdefinitionen
– Definitionen von Makros
– Definitionen von Adressen (z.B. SFR vom 8051)

☞ !! Folgende Bestandteile sollte das H-File **nicht** enthalten. !!

– Definitionen von Variablen
– Programmteile oder Funktionen

Übersetzen Sie das Programm **\c51_buch\kap_2\example\first.c** von der CD mit den C51-Compiler Steuerparameter **LIST #include Files (LC)** sowie dem Steuerparameter **PREPRINT (PP)**. Die Beschreibung der einzelnen Steuerparameter finden Sie in Kapitel 3.2. Sie können zur Compilierung eine DOS-Box unter Windows öffnen oder das Projekt First in µVision bzw. µVision2 laden. Falls Sie sich zuerst noch mit dem Aufbau von Projekten vertraut machen wollen, gehen Sie zu Kapitel 16.

Um die Steuerparameter bei der Compilierung mit aufzunehmen, haben Sie folgende Möglichkeiten:

1. Sie verwenden die Preprozessor Steuerparameter **#pragma** (siehe Listing 2 ①).

☞ !! Dabei ist es wichtig, daß die **#pragma** Steuerparameter in den ersten Zeilen Ihres Source-Moduls stehen. !!

```
①  #pragma LC
   #pragma OE
   #include "defines.h"    // H-File liegt im selben Pfad
   #include <reg517.h>     // H-File über C51INC-Path
```

Listing 2 Auszug aus First.c

💣 !! Die Steuerparameter PREPRINT und DEFINE können als Steuerparameter nicht über **#pragma** mit ins Source-Modul aufgenommen werden. Sie müssen den Steuerparameter beim Compileraufruf angeben (siehe Abbildung 3 bzw. Abbildung 6 über die Zusatzangabe **pp**). !!

2. Der Compiler wird aus einer DOS-Box gestartet. Die Steuerparameter werden nach dem Source-Modul angegeben (siehe Abbildung 3).

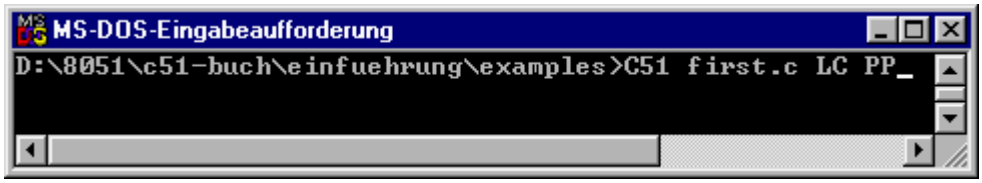


Abbildung 3 Aufruf aus DOS-Box (die Angaben des Pfads können abweichend sein)

Überprüfen Sie die Pfadangabe in der Autoexec.bat, falls sich der C51-Compiler nicht aufrufen läßt.

3. Sie geben unter µVision bzw. µVision2 die Steuerparameter für die Compileroptionen mit an. **Alle** Dateien in Ihrem Projekt werden unter µVision ab diesem Zeitpunkt mit dieser Option übersetzt. Der Steuerparameter **LC** wird bei µVision über das Menü **Options/ C51 Compiler/ Listing/** List #include files eingestellt (siehe Abbildung 4). Unter µVision2 erfolgt der Eintrag über **Options for Target/ Listing**/ #include Files.

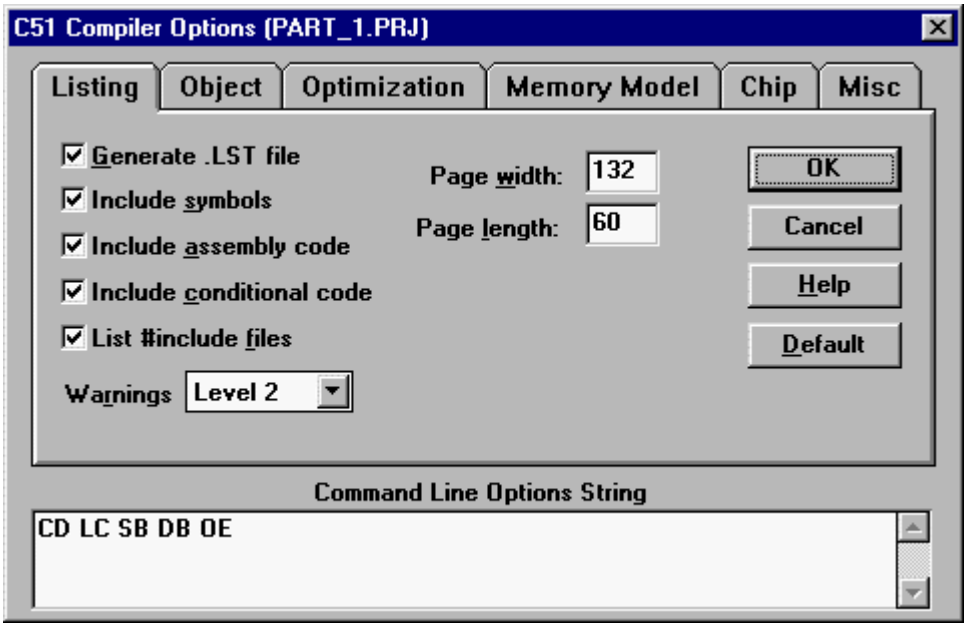


Abbildung 4 (List #include files) Steuerparameter für den Compiler

Der Steuerparameter **PREPRINT** wird im Menü **Options\C51 Compiler\Misc\ Additional Options** eingetragen (siehe Abbildung 6). Der Steuerparameter **LC** (List #include files) bewirkt, daß das LST-File den kompletten Inhalt der eingebundenen H-Files enthält. Der Steuerparameter **PP** erzeugt nach dem Lauf des Preprozessors (siehe Abbildung 5) ein I-File. Im I-File sind alle #define Anweisungen ersetzt sowie alle Makros expandiert worden. Zudem sind nur die relevanten Informationen der H-Files im I-File enthalten. Dieses File hat somit alle notwendigen Informationen für einen Compilerlauf und kann vom C51-Compiler ohne H-Files übersetzt werden.

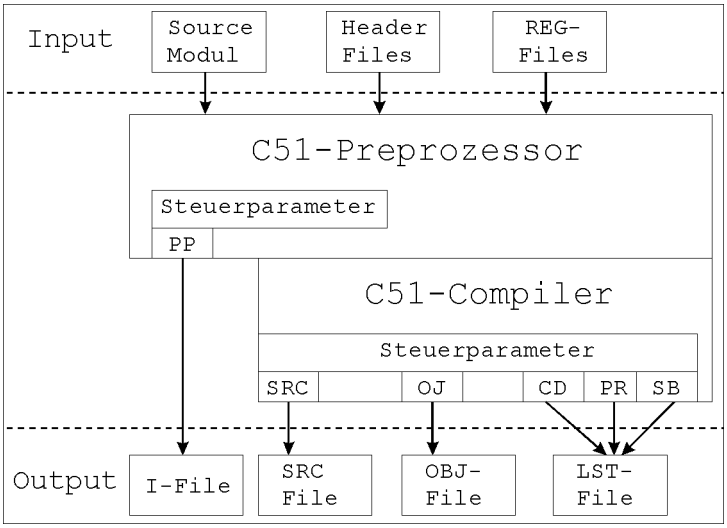


Abbildung 5 Schematischer Ablauf des C51-Compilers

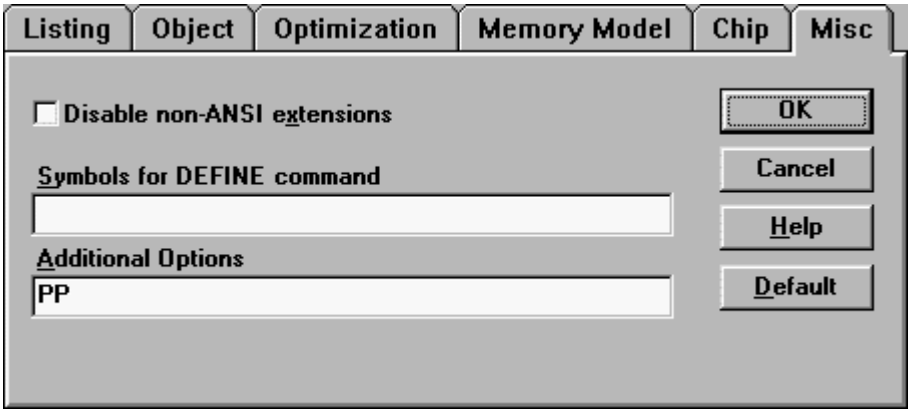


Abbildung 6 Einstellung für den Steuerparameter Preprint

Nach der Compilierung finden Sie in Ihrem Verzeichnis zwei zusätzliche Dateien:

– **First.LST**: Listing von First.c
– **First.I**: Output-File durch den Steuerparameter PREPRINT

```
1 #include "defines.h"// Header-File liegt im selben Pfad
1  =1  #define ulong unsigned long
2  =1  #define uint  unsigned int
3  =1  #define uchar unsigned char
2 #include <reg517.h>    // Header-File über C51INC-Pfad
1  =1  /* © Copyright KEIL ELEKTRONIK GmbH. 1990 , All rights */
2  =1  /*  Register Declarations for the 80C517 Processor */
3  =1
4  =1  /*  BYTE Register  */
5  =1  sfr   P0     = 0x80;
6  =1  sfr   SP     = 0x81;
```

Listing 3 Auszug aus First.LST

Die Datei First.LST enthält durch den **LISTINCLUDE** Steuerparameter zusätzliche Informationen, die durch die H-Files eingebracht wurden. In diesem Beispiel sind es **#define** Anweisungen sowie Definitionen von SFR-Adressen. Die **#define** Anweisung hat folgenden Aufbau:

#define Bezeichner Ersatztext

– **#define:** Steuerparameter für den Preprozessor
– **Bezeichner:** Der Aufbau des Bezeichners unterliegt den Regeln der Variablennamen. C-Schlüsselwörter (siehe Kapitel 3.1) sowie Steuerparameter (siehe Kapitel 3.2) dürfen als Bezeichner nicht verwendet werden.
– **Ersatztext:** Konstanten, Namen, Bezeichnungen, usw.

Die Bezeichner (**ulong, uint,** ..) werden vom Preprozessor (siehe Kapitel 2.4) durch den jeweiligen Ersatztext ersetzt. Um dieses nachzuvollziehen, öffnen Sie das File First.I. In Zeile 2 und 3 der Datei First.I finden Sie die vom Compiler eingebundenen H-Files. Die Pfadangabe **<reg517.h>** wurde in einen absoluten Pfad umgewandelt (siehe ①). Zudem wurde die Bezeichnung **uint** gegen den Ersatztext **unsigned int** ausgetauscht (siehe ②).

```
   #line 1 "FIRST.C" /0
   #line 1 "defines.h" /0
①  #line 1 "C:\KEIL_SW\C51\INC\REG517.H" /0

    sfr   P0     = 0x80;
    sfr   SP     = 0x81;
    ... ;
    sbit  IEX2   = 0xC1;
    sbit  IADC   = 0xC0;
   #line 2 "FIRST.C" /0

②    unsigned int a_ui;
③  extern unsigned int add_ui(unsigned int first_ui, unsigned int
   sec_ui);

②  unsigned int sub_ui(unsigned int first_ui, unsigned int sec_ui){
    return first_ui - sec_ui;
    }

    main(){
②    unsigned int b_ui;

    b_ui = 0x2334;
    a_ui = add_ui(b_ui, 0x34);
    a_ui = sub_ui(a_ui, b_ui);
    }
```

Listing 4 Auszug aus First.I

In Listing 4 finden Sie das Schlüsselwort **extern** (siehe ③). Werden Variablen oder Funktionen mit **extern** versehen, so handelt es sich um Deklarationen. Eine Deklaration ist eine Bekanntgabe von Variablen und Funktionen an den C-Compiler.

extern Bezeichner;

- **extern:** Schlüsselwort von ANSI-C (siehe Kapitel 3.1).
- **Bezeichner:** Die Variable bzw. die Funktion die hier angegeben wird, muß an einer Stelle in Ihrem Programm noch definiert sein. Sie müssen die Definitionen sowie die Deklarationen von Variablen bzw. Funktionen gleich schreiben.

💣 !! C-Schlüsselwörter **müssen** klein geschrieben werden. !!

Der C-Compiler benötigt die **extern** Informationen, damit er den nötigen Code für den Funktionsaufruf oder die Variablenbehandlung aufbauen kann. In unserem Beispiel First.c ist eine Deklaration auf die Funktion **add_ui()** vorhanden. Der C-Compiler kann nun über die Deklaration der Funktion alle nötigen Informationen, wie z.B., daß es sich um eine Funktion mit Parameterübergabe handelt, die zwei Parameter vom Datentyp **unsigned int** beim Aufruf erwartet, sowie einen Rückgabewert **unsigned int** liefert, entnehmen. Weitere Informationen über die Deklaration von Funktionen können Sie dem Kapitel 5.1 entnehmen.

```
    a_ui = add_ui(b_ui, 0x34);
①   0006 AF00    R     MOV     R7,b_ui+01H
①   0008 AE00    R     MOV     R6,b_ui
①   000A 7D34          MOV     R5,#034H
①   000C 7C00          MOV     R4,#00H
②   000E 120000  E     LCALL   _add_ui
③   0011 8E00    R     MOV     a_ui,R6
③   0013 8F00    R     MOV     a_ui+01H,R7
```

Listing 5 Auszug aus First.LST E = extern R = relocatibel

In Listing 5 ist die für den Funktionsaufruf benötigte Codesequenz abgebildet. Im Bereich ① wird die Variablenübergabe vorbereitet, in ② wird der Funktionsaufruf durchgeführt und im Bereich ③ wird das Ergebnis ausgewertet (in diesem Beispiel wird das Ergebnis der Variablen a_ui zugewiesen).

Falls bei Ihrem LST-File der Assemblercode nicht mit ausgegeben wird, müssen Sie in µVision unter **Options\C51 Compiler\Listing** die Option **Include assembly code** ankreuzen. Falls Sie hier schon mehr über die Parametertechnik bei Funktionsaufrufen des C51-Compilers wissen möchten, so können Sie dies im Kapitel 5.3 nachlesen.

2.2 Aufbau des Source-Moduls

In den Beispielen wird ein Aufbau verwendet, der sich bei meinen Projekten bewährt hat. Er ist so ausgelegt, daß ihn das Projekttool MKS optimal nutzen kann. Einige Gliederungen werden zudem von C vorgegeben. Da dieser Aufbau bei den Abbildungen im Buch zuviel Platz einnehmen würde, werden die Source-Module nur auszugsweise dargestellt.

```
    //*************************************************************
    // Modulname:    $Source: c:/c51_buch/io/IO_TEST.c $
①   // User:         $Autor: MEBA $
①   // Version:      $Name: VERSION1_ENTW $  $Revision: 1.7$
①   // Datum:        $Date: 1998/08/23 21:20:51 $
①   // Qualität:     $State: Entwicklung $
①   //*************************************************************
①   // Beschreibung:
①   //*************************************************************
①   // Historie:
①   //==========
①   // $Log: IO_TEST.c $
①   //*************************************************************
    // Steuerparameter:
    //*************************************************************
②   #pragma LC OE
②   #pragma OT(2,size)
    //*************************************************************
    // globale Definitionen
    //*************************************************************
③   #define IO_TEST
③   #define _80517
    //*************************************************************
```

```
   // verwendete Include Dateien
   //*************************************************************
④  #include "defines.h"
④  #include <reg517.h>
④  #include "mat_tast.h"
   //*************************************************************
   // Deklarationen von Funktionen und Variablen
   //*************************************************************
⑤  extern uchar pdata port_uc;
   //*************************************************************
   // Definitionen von Variablen
   //*************************************************************
⑥  uchar data erg_uc;

   //*************************************************************
   // Funktionsname main
   // Version: 1.0    Datum 13.9.98
   // Übergabeparameter: keine
   // Beschreibung:
   // Auswertung von Matrix-Tastaturen
   //*************************************************************
⑦  void main(void){
⑦    erg_uc = init_matrix(CHECK);
⑦    if (erg_uc == OK) {
⑦      port_uc = init_matrix(INIT);
⑦      erg_uc = wait_for_value();
⑦    }
⑦    erg_uc = init_matrix(CLOSE);
⑦  }
```

Listing 6 Aufbau des Source-Moduls

In Listing 6 ist der Aufbau des Source-Moduls IO_TEST abgebildet. In Bereich ① des Source-Moduls sind alle globalen Informationen, wie Modulname, Version usw. enthalten. Der Bereich ② enthält die Steuerparameter für den C51-Compiler. Dieser Bereich sollte nur dann verwendet werden, wenn Sie gezielt einen Steuerparameter für dieses Source-Modul verwenden wollen. In Bereich ③ werden alle globalen Definitionen festgelegt. Diese können somit auch Einfluß auf die nachfolgenden #include Dateien haben.

☞ !! Verwenden Sie als Konstanten immer #define Anweisungen. Sie können Ihr Programm dann schneller bei Änderungswünschen anpassen. Der Sourcecode kann zudem besser gelesen werden. !!

Bereich ④ enthält die #include Anweisungen für die Header Dateien. In Bereich ⑤ werden die Funktionen und Variablen deklariert. Deklarationen von Variablen und Funktionen sollten vorrangig in Header Dateien stehen, die wiederum von Ihrem Source-Modul über #include mit hinzugebunden werden. Bereich ⑥ enthält alle globalen Variablen. In Bereich ⑦ sind die Funktionen definiert.

Folgende Regeln sollten Sie beim Aufbau Ihres Source-Moduls beachten:

– Ihr Source-Modul sollte **nicht mehr** als 5 DIN A4 Seiten umfassen.
– Verwenden Sie **keine** magic numbers in Ihrem Sourcecode (z.B. erg_c = 0x23).

– Deklarationen von Variablen und Funktionen **sollten immer** in einem H-File erfolgen.
– Fügen Sie in die H-Files **keinen** Sourcecode ein.
– Fügen Sie die H-Files nur **am Anfang** des Source-Moduls ein.
– Verwenden Sie **keine** lokalen Definitionen.

2.3 Aufbau des LST-Files

Der C-Compiler erzeugt bei der Compilierung standardmäßig ein LST-File. Das LST-File kann über die Steuerparameter **NOPRINT** abgeschaltet werden. Wird unter µVision bei der Compilierung kein LST-File erzeugt, so müssen Sie die Compiler-Option **Generate .LST file** überprüfen. Im LST-File sind folgende Angaben enthalten:

Die verwendete Version des C51-Compilers (siehe ①); der Name des compilierten Source-Moduls (siehe ②); es wird angezeigt, ob ein OBJ-File erzeugt wurde und welchen Name dieses File hat. Dieses File wird nicht erzeugt, wenn der Steuerparameter **SRC** verwendet wurde (siehe ③). Es werden die Teile des Sourcecodes angegeben, die für die Compilierung verwendet wurden (siehe ④). Sie finden auch Angaben über den benötigten Speicherplatz im ROM und RAM (siehe ⑤). Je nach verwendeten Steuerparametern des C51-Compilers kann noch der Assemblercode (**CODE**), die Inhalte der H-Files (**LISTINCLUDE**), usw. in das LST-File mit aufgenommen werden (siehe ⑥).

Das Listing 7 enthält das LST-File des Beispiels first.c.

```
①  C51 COMPILER 5.50,  FIRST            11/07/98  15:12:27  PAGE 1

   DOS C51 COMPILER 5.50, COMPILATION OF MODULE FIRST
③  OBJECT MODULE PLACED IN FIRST.OBJ
②  COMPILER INVOKED BY: D:\8051\C51\BIN\C51.EXE FIRST.C NOCO NOIP

   stmt level    source
      1          #include "defines.h"  // H-File liegt im aktuellen
      2          #include <reg517.h>   // H-File über C51INC
      3          uint  a_ui;           // globale Variable
      4
      5          extern uint add_ui(uint first_ui, uint sec_ui); //
      6                                                          //
      7
      8          uint sub_ui(uint first_ui, uint sec_ui) {       //
④     9   1        return first_ui - sec_ui;                     //
④    10   1      }
     11
     12          main(){
④    13   1       uint b_ui;                    // lokale Variable
④    14   1
④    15   1       b_ui = 0x2334;                // Wertzuweisung
④    16   1       a_ui = add_ui(b_ui, 0x34);    // Funktionsaufruf
④    17   1       a_ui = sub_ui(a_ui, b_ui);
④    18   1      }

⑤  MODULE INFORMATION:   STATIC OVERLAYABLE
      CODE SIZE        =     41    ----
```

```
   CONSTANT SIZE    =   ----    ----
   XDATA SIZE       =   ----    ----
   PDATA SIZE       =   ----    ----
   DATA SIZE        =      2       2
   IDATA SIZE       =   ----    ----
   BIT SIZE         =   ----    ----
END OF MODULE INFORMATION.
C51 COMPILATION COMPLETE.  0 WARNING(S),  0 ERROR(S)
```

Listing 7 Beispiel für ein LST-File

Falls Ihr Programm Fehler enthält, so wird an diesen Stellen im LST-File die Warnung bzw. der Fehler eingetragen (siehe Listing 8).

```
 9   1        return first_ui – sec_ui
10   1      }
ERROR 141 IN LINE 9 OF FIRST.C:syntax error near '}', expected ';'
```

Listing 8 Angaben der Fehler im LST-File

2.4 C-Preprozessor

Der C-Preprozessor ist ein dem C-Compiler vorgeschaltetes Modul (siehe Abbildung 5, Seite 24). Mit ihm kann der Sourcecode manipuliert werden, bevor der Compiler den Sourcecode übersetzt. Der Preprozessor kann über Anweisungen gesteuert werden. Die Anweisungen werden mit einem #eingeleitet und enden mit dem Zeilenende (bei C mit dem ;). Sie haben die Möglichkeit, die Zeile durch einen **** in der nächsten Zeile fortzusetzen. In den vorigen Kapiteln haben Sie schon mit den Anweisungen des Preprozessors **#include, #define** und **#pragma** gearbeitet. In Tabelle 2 finden Sie eine Übersicht über alle Anweisungen des Preprozessors.

Steuerparameter	Beschreibung:
#pragma	Steuert die C51-Compiler Direktiven.
#include	Fügt eine Datei an die Stelle ein, an der die #include Anweisung steht.
#define	Definiert einen Bezeichner.
#undef	Ein vorhandener Bezeichner wird für ungültig erklärt.
#if	Abfrage, ob der Bezeichner das Ergebnis **true** liefert.
#ifdef	Abfrage, ob ein Bezeichner schon definiert ist.
#ifndef	Abfrage, ob der Bezeichner noch nicht definiert ist.
#elif	Bedingte Abfrage, wenn das Ergebnis einer vorangegangenen #if oder #elif- Abfrage **false** war.
#else	Bedingte Verzweigung, falls das Ergebnis einer vorangegangenen #if oder #elif- Abfrage **false** war.
#endif	Endekennung für #if und #ifndef Steuerparameter.
#line	Fügt eine Kommentarzeile ein.
#error	Erzeugt eine Fehlermeldung.

Tabelle 2 Preprozessor Steuerparameter

Der Steuerparameter **PREPRINT** des C51-Compilers gibt die Umsetzung des Preprozessors in einem I-File aus. Der Preprozessor ist für folgende Aufgaben verantwortlich:

– Einkopieren von Files über die **#include** Anweisung
– Die \ (Backslash) - CR Kombinationen werden entfernt und somit die Zeilen verbunden.
– Aufbau von Definitionen mittels **#define** und **#undef**.
– Makro Bearbeitung
– Steuerung des Preprozessors mittels **#ifndef**, **#elif** und **#endif**
– Ersetzen der Drei-Zeichen-Folgen

In den folgenden Abschnitten werden die einzelnen Steuerparametern des Preprozessors ausführlich beschrieben.

Steuerparameter #include

Mit dem Steuerparameter **#include** werden Files mit in das Source-Modul aufgenommen. Diese Files enthalten Definitionen von Konstanten und Makros sowie Deklarationen von Variablen und Funktionen.

☞ !! Die **#include** Steuerparametern sollten direkt nach den **#pragma** Anweisungen folgen. !!

Folgende Syntax ist für die **#include** Steuerparameter zulässig:

#include <Filename.h>
#include "Filename.h" oder "\Pfad\Filename.h"

– **#include**: Steuerparameter für den Preprozessor
– **<>**: Das H-File wird über den Pfad, der in der Umgebungsvariable **C51INC** eingetragen ist, gesucht.
– **\Pfad**: Es sind absolute und relative Angaben zulässig.
– **""**: Das H-File wird in dem Verzeichnis gesucht, in dem auch das Source-Modul steht.

In Listing 9 finden Sie einige Beispiele von erlaubten (siehe ①) und unzulässigen #include Anweisungen (siehe ②).

```
   //*************************************************************
   // zulässige #include Anweisungen
   //*************************************************************
   // H-File wird über die Environment Variable gesucht
①  #include <reg51.h>
   // H-File wird aus definiertem Verzeichnis gelesen
①  #include "D:\8051\C51-Buch\test\test.h"
   // Das H-File liegt in einem Parallelverzeichnis des
   // Source-Moduls
①  #include <..\inc\set.h>

   //*************************************************************
   // unzulässige #include Anweisungen
   //*************************************************************
   //Es ist nicht zulässig Preprozessor-Anweisungen zu verschachteln
```

```
② #define PATH D:\8051\C51-Buch\test
② #include "PATH\test.h"
```

Listing 9 Beispiele von möglichen #include Anweisungen

Es ist zulässig, H-Files ineinander zu verschachteln. Wie das Verschachteln aufgebaut wird, hängt von Ihrer Programmstruktur ab. Listing 10 enthält ein Beispiel mit zwei Verschachtelungen. Der Preprozessor liest das Source-Modul **test.c** von oben beginnend ein. Dabei findet der Preprozessor den Steuerparameter **#include** (siehe ①). Der Preprozessor unterbricht das Einlesen von test.c, öffnet die Datei test.h und beginnt diese einzulesen. Dabei stößt er erneut auf den **#include** Steuerparameter (siehe ②). Der Preprozessor verfährt erneut nach dem gleichen Schema. Nachdem die Datei **defines.h** vollständig eingelesen wurde, fährt er mit der Datei **test.h** fort. Erst wenn dieses komplett eingelesen ist, wird das Source-Modul test.c vollständig eingelesen.

```
  //*********************************************************
  // Modul test.c
  //*********************************************************
① #include "test.h"
  //*********************************************************
  // H-File test.h
  //*********************************************************
  #include "defines.h"
② extern uint  a_ui;
  extern uchar a_uc;
  //*********************************************************
  // H-File defines.h
  //*********************************************************
  #define uchar unsigned char
  #define uint  unsigned int
```

Listing 10 Beispiel für verschachtelte H-Files

☞ !! Es können maximal 9 H-Files ineinander verschachtelt werden. !!

Damit ein H-File nicht mehrmals bei einem Compilerlauf des Preprozessors mit eingebunden wird, sollten die H-Files wie folgt aufgebaut sein:

```
① #ifndef Name       // Abfrage, ob der Name schon definiert wurde
    #define Name     // definiere den Namen
    ....             // weitere Anweisungen für den Preprozessor
  #endif             // Ende der ifndef Verschachtelung
```

Listing 11 Aufbau von H-Files

Der Inhalt des H-Files wird nur dann vom Preprozessor zum Compilerlauf mit hinzugebunden, wenn die Definition Name (siehe ①) noch nicht existiert. Somit wird sichergestellt, daß der Inhalt des H-Files nur einmal eingelesen wird. Die Preprozessor Anweisungen **#ifndef** und **#endif** sind im Abschnitt „Bedingte Übersetzung“ beschrieben.

Makro Definitionen

Ein Makro ist eine Kurzschreibweise, um bestimmte Ausdrücke vereinfacht darstellen zu können. Der Preprozessor expandiert das Makro in den jeweiligen Source-Zeilen. Er expandiert die Makros nicht einfach dadurch, daß er den Bezeichner durch einen

Ersatztext ersetzt. Er ist in der Lage, die Namen von Variablen innerhalb eines Makros auf die von Ihnen benötigten Bezeichner zu ändern. Somit können Makros geschrieben werden, die sich in vielen unterschiedlichen Programmen einsetzen lassen.

Das Makro wird über die **#define** Anweisung erzeugt. Dabei wird der Bezeichner durch den Ersatztext ausgetauscht. Werden Werte als Ersatztext verwendet, so sind diese vom Datentyp int.

#define Bezeichner Ersatztext

- **#define**: Steuerparameter des Preprozessors.
- **Bezeichner**: Der Aufbau des Bezeichners unterliegt den Regeln der Variablennamen. C-Schlüsselwörter (siehe Kapitel 3.1) sowie Steuerparameter (siehe Kapitel 3.2) dürfen für den Bezeichner nicht verwendet werden.
- **Ersatztext**: Konstanten, Namen, Bezeichnungen, usw.

☞ !! Wird derselbe Bezeichner ein zweites Mal definiert, so wird eine Fehlermeldung ausgegeben. Eine Ausnahme gibt es nur, wenn der Ersatztext in beiden Fällen identisch ist. !!

```
#define forever for(;;)
#define uchar unsigned char
#define B00000001 0x01
#define TRUE 0
#define BUF_LENGTH 256

uchar buffer[BUF_LENGTH];
uchar a_uc = B00000001;
```

Listing 12 Beispiele von Definitionen

Werden runde Klammern nach dem Bezeichner geschrieben, so handelt es sich um ein parametrisiertes Makro. Die Parameter werden durch Kommata getrennt angegeben. Wenn Leerzeichen zwischen Makroname und den Klammern stehen, nimmt der Preprozessor an, daß die in Klammern gefaßten Zeichen Bestandteil des Ersatztextes der Konstanten sind.

#define Bezeichner(Parameterliste) (Sequenz)

- **#define**: Steuerparameter des Preprozessors.
- **Bezeichner**: Siehe Regeln der Variablennamen (Kapitel 4).
- **Parameterliste**: Die Parameterliste wird in runden Klammern eingeschlossen. Bei mehreren Parametern werden diese durch Kommata getrennt angegeben.
- **Sequenz**: Die Sequenz wird auch in runden Klammern eingeschlossen. Es gelten die Verarbeitungsregeln von C für Operatoren.

☞ !! Die linke Klammer **muß** direkt nach dem Bezeichner folgen. Es darf kein Leerzeichen dazwischen stehen. !!

```
#define ABSDIFF(a,b)          (((a) > (b)) ? (a)-(b) :(b) – (a))
#define MAX(a,b)              (((a) > (b)) ? (a) : (b))
#define HIGH_2_LOW(a)         ((unsigned char) (( a ) >> 8))
#define LOW_2_HIGH(a)         (((unsigned int) a <<8))
```

```
#define SQUARE(x)             ((x) * (x))
#define UNITE_2CHAR_INT(a, b) ((((unsigned int) a) <<8) | b)
```

Listing 13 Beispiele Makros aus dem H-File makros.h

Die Makros in Listing 13 wurden absichtlich mit so komplexen Klammern versehen. Dies wird dann benötigt, wenn die Makros verschachtelt verwendet werden.

```
#define MAX(a,b)              (((a) > (b)) ? (a) : (b))
#define SQUARE(x)             ((x) * (x))

main() {
  char erg_c;
  erg_c = MAX(SQUARE(2),3);           // erg_c = 4
  erg_c = MAX(MAX(19,SQUARE(5)),24); // erg_c = 25
}
```

Listing 14 verschachtelte Makros

Um das Abarbeiten eines Makros verständlich zu machen, ist in Listing 15 das Makro **MAX(a,b)** erklärt. Das Makro MAX liefert den größeren der beiden Parameter zurück.

```
#define MAX(a,b)              (((a) > (b)) ? (a) : (b))

main(){
 char erg_c;
 erg_c = MAX(17,3);
}
```

```
         MAX (17,3)       // Makro Aufruf
          a > b ?  a : b  // definiertes Makro
 erg_c = ((17) > (3) ? (17) : (3); // Aufgelöstes Makro
```

```
; FUNCTION main (BEGIN)   // Vom C51-Compiler erzeugter Code
0000 750011      R     MOV     erg_c,#011H
0003 22                RET
; FUNCTION main (END)
```

Listing 15 Beispiel Makro Auflösung

☞ !! Werden wie in Listing 15 Makros mit Konstanten aufgerufen, so wird das Ergebnis schon bei der Compilierung berechnet und wie in diesem Fall der Variablen zugewiesen. !!

Um einen Bezeichner ein zweites Mal nutzen zu können, müssen Sie den aktuellen Bezeichner für ungültig erklären. Die Syntax lautet wie folgt:

#undef Bezeichner

– **#undef**: Steuerparameter des Preprozessors.
– **Bezeichner**: Siehe Regeln des Variablennamen (Kapitel 4).

```
#define MIKROPRO 515
#include <reg515.h>
#undef MIKROPRO

#define MIKROPRO 8051
```

Listing 16 Beispiel mit dem Preprozessor Steuerparameter #undef

Bedingte Übersetzung

Der Preprozessor verfügt über Anweisungen, mit denen Sie Ihr Source-Modul bedingt übersetzen können. Die Steuerung wird mit den Anweisungen **#if**, **#elif**, **#else**, **#endif** sowie mit **#ifdef** und **#ifndef** realisiert.

#if Bezeichner Sequenz

- **#if:** Steuerparameter des Preprozessors.
- **Bezeichner:** Siehe Regeln des Variablennamen (Kapitel 4).
- **Sequenz:** Sie können auf ganzzahlige Werte abfragen. Es sind alle Vergleichsoperatoren zulässig.

💣 !! In der Sequenz dürfen keine sizeof Ausdrücke, Umwandlungsoperationen oder Aufzählungskonstanten (enum) vorkommen. !!

Die **#if** Anweisung bewertet den Bezeichner in Abhängigkeit von der Sequenz. Die Sequenz selbst ist optional. Wird keine Sequenz angegeben, so wird der Bezeichner auf ungleich 0 überprüft. Ist das Ergebnis ungleich 0, so wird der Textteil zwischen der **#if** Anweisung und der nächsten Anweisung **#elif**, **#else** oder **#endif** in den Sourcecode eingefügt.

💣 !! Bei **#if** und **#elif** Abfragen **muß** bei der Definition der Bezeichner mit einem Ersatztext versehen sein. Der Compilerlauf wird sonst mit Error abgebrochen (siehe Listing 17 ①). !!

💣 !! Der Ersatztext **muß** vom Datentyp int sein, andernfalls gibt der Preprozessor eine Warnung aus (siehe ②). !!

```
   #define SMALL
   #define FLOAT 1

   main() {
①  #if SMALL        // #if defined SMALL ist die korrekte Anweisung
    char a_c;       // oder #define SMALL 1
   #endif
   Error 308 invalid integer const expression
②  #if FLOAT == 1.1
     float a_f;
   #endif
   }
   Warning 323 Newline expected, extra character
```

Listing 17 Beispiele mit unzulässigen bzw. unvollständigen #define Anweisungen

#elif Bezeichner Sequenz

- **#elif**: Steuerparameter des Preprozessors.
- **Bezeichner**: Siehe Regeln des Variablennamen (Kapitel 4).
- **Sequenz**: Die Abfrage kann nur auf ganzzahlige Werte erfolgen. Es sind alle Vergleichsoperatoren zulässig.

☞ !! Der Steuerparameter **#elif** kann nur nach einem **#if** eingesetzt werden. !!

In Listing 18 wird der Variablen a_c über ein #if - #elif Konstrukt der Wert 3 zugewiesen.

#else

– **#else**: Steuerparameter des Preprozessors

💣 !! Der Steuerparameter **#else** kann nur nach einem **#if** bzw. **#elif** folgen. !!

Der **#else** Zweig wird nur dann angesprungen, wenn die vorhergehenden Abfragen das Ergebnis **false** lieferten.

#endif

– **#endif**: Steuerparameter des Preprozessors

💣 !! Die Steuerparameter **#if**, **#ifndef**, **#ifdef** müssen mit **#endif** abgeschlossen werden. !!

```
#define A 1
#define B 2

main(){
 char a_c;
#if A == 1  && B == 0 // Der Gesamtausdruck von A und B muß true
   a_c = 1;           // ergeben
#elif (A * 3) > 5     // Abfrage auf größer 5
   a_c = 5;
#elif A != 3          // Abfrage auf ungleich 3
   a_c = 3;
#else                 // Diese Bedingung tritt ein, wenn keine
   a_c = 0;           // der anderen Abfragen das Ergebnis true
#endif                // liefert.
}
```

Listing 18 Beispiel für eine bedingte Übersetzung

Mit dem Preprozessor können Sie zudem abfragen, ob ein Bezeichner definiert bzw. noch nicht definiert wurde. Folgende Abfragen sind möglich:

#if defined Bezeichner

– **#if**: Steuerparameter des Preprozessors.
– **defined**: Gibt das Ergebnis true, wenn der Bezeichner schon definiert ist.
– **Bezeichner**: Siehe Regeln von Variablennamen (Kapitel 4).

#ifdef Bezeichner

– **#ifdef**: Steuerparameter des Preprozessors.
– **Bezeichner**: Siehe Regeln von Variablennamen (Kapitel 4).

Mit **#if** defined bzw. **#ifdef** wird überprüft, ob der Bezeichner schon definiert wurde. Das Ergebnis dieser Überprüfung ist **true,** wenn der Bezeichner existiert.

#if !defined Bezeichner

– **#if**: Steuerparameter des Preprozessors.
– **defined**: Gibt das Ergebnis true, wenn der Bezeichner noch nicht definiert wurde.

#ifndef Bezeichner

– **#ifndef**: Steuerparameter des Preprozessors.
– **Bezeichner**: Siehe Regeln von Variablennamen (Kapitel 4).

Mit **#ifndef** bzw. **#if !defined** wird überprüft, ob der Bezeichner noch nicht existiert. Ist dies der Fall, so ist das Ergebnis true.

```
#if !defined MICROPRO
 #define MICROPRO 515
 ...
 #if MICROPRO == 515
   #include <reg515.h>
 #elif MICROPRO == 517
   #include <reg517.h>
 #else
   #include <reg51.h>
 #endif
#endif
```

Listing 19 Beispiel bedingte Übersetzung für das Einbinden von H-Files

In Listing 19 wird überprüft, ob der Bezeichner MICROPRO schon definiert wurde. Wenn nicht, so soll der Bezeichner definiert werden und das dazugehörige H-File soll mit eingebunden werden.

```
 #if MICROPRO == 515
   #include <reg515.h>
 #elif MICROPRO == 517
   #include <reg517.h>
 #else
   #include <reg51.h>
 #endif
```

Listing 20 Abfrage eines Makros mit Fallunterscheidung

In Listing 21 wird die Problematik Definition und Deklaration von globalen Variablen mittels einer Definition umgangen. Dabei wird TEST_HEADER im Source-Modul test.c definiert. In diesem Fall werden die Variablen definiert. In allen anderen Fällen werden die Variablen deklariert.

☞ !! Für SFR-Register und SBIT-Variablen kann dieses Verfahren nicht angewendet werden. !!

```
//*************************************************************
// Modul: TEST.H
//*************************************************************
#ifdef TEST_HEADER
 #define EXTERN
#else
 #define EXTERN extern
#endif

EXTERN char a_c;
```

```
//*************************************************************
// Modul: TEST.C
//*************************************************************
#define TEST_HEADER
#include "TEST.H"

main() {
  a_c = 3;
}
```

Listing 21 Umsetzung von Deklaration und Definition von Variablen im H-File

Beim C51-Compiler sind einige Makro Konstanten vordefiniert. Diese werden bei der Compilierung mit Werten belegt. Folgende Makros sind vorhanden:

- **__C51__** Die Versionsnummer des C-Compilers. Der Wert 550 entspricht der Version 5.5.
- **__DATE__** Das Datum, an dem der Compiler gestartet wurde.
- **__FILE__** Name des compilierten Source-Moduls.
- **__LINE__** Aktuelle Zeilennummer während der Compilierung.
- **__MODEL__** Angabe, welches Memory Model verwendet wurde (0 für SMALL, 1 für COMPACT und 2 für LARGE).
- **__STDC__** Der Wert 1, wenn der ANSI-C Standard verwendet wurde.

NEW

- **__KEIL__** Den Wert 1, wenn ein Keil C-Compiler verwendet wurde.
- **__VER__** Entspricht der Funktionsweise von __C51__.

In Listing 22 wird überprüft, ob das Source-Modul von einem C51-Compiler übersetzt wird. Ist dies der Fall, wird zudem überprüft, ob der C51-Compiler die Option **_at_** unterstützt. Diese Option ist ab der Version 3.40 enthalten. Andernfalls werden die Variablen mit den ANSI-Vorgaben definiert.

```
ifdef __C51__
  #if __C51__ >= 340
    char data  a_c _at_ 0x20;
    int  idata a_i _at_ 0x80;
  #else // diese Versionen unterstützen kein _at_
    char data  a_c;
    int  idata a_i;
  #endif
#else // es wird kein C51-Compiler verwendet
   char a_c;
   int  a_i;
#endif
```

Listing 22 Abfrage von vordefinierten Makros

Drei-Zeichen-Folgen

Die Zeichen **#**, ****, **^**, **[**, **]**, **|**, **{**, **}** sowie **~** sind im 7-bit ASCII-Zeichensatz nicht vorhanden. Damit diese Zeichen dennoch verwendet werden können, wurden dafür sogenannte Drei-Zeichen-Folgen (trigraph sequences) definiert. Diese finden heute noch Verwendung, man denke nur an das VT-10 Terminal. Da der C51-Preprozessor diese Folgen unterstützt, werden sie mit aufgeführt. Die Zeichen sind wie folgt definiert:

#	**??=**	**[**	**??(**	**{**	**??<**
****	**??/**	**]**	**??)**	**}**	**??>**
^	**??´**	**|**	**??!**	**~**	**??-**

Tabelle 3 Abbildung der Drei-Zeichen-Folgen

```
main(){
 char a_c = '??=';  // ??= entspricht 0x23 bzw. #
}
```

```
MOV     a_c,#023H
```

Listing 23 Beispiel mit trigraph sequences

2.5 Gültigkeitsbereiche (global, local, public, static)

Ein Programm besteht meist aus mehreren Funktionen und Variablen. Diese müssen nicht im selben Source-Modul definiert worden sein. Ihr Programm benötigt nicht zu jedem Zeitpunkt alle definierten Variablen und Funktionen. Der Zeitpunkt, zu dem sie benötigt werden, kann in einem Gültigkeitsbereich umgesetzt werden. Der Gültigkeitsbereich eines Namens ist der Teil des Programms, in dem der Name benutzt werden kann. In C wurden die Gültigkeitsbereiche global, local, public und static eingeführt.

☞ !! Der Gültigkeitsbereich **static** ist ein Schlüsselwort des C-Compilers. !!

global

Wird eine Variable außerhalb einer Funktion definiert, spricht man von einer **globalen** Variablen. Diese Variable kann von allen Funktionen verwendet werden, die sich nach der Variablendefinition befinden und im selben Source-Modul stehen (siehe z.B. Fehler (Listing 24 ①). Dieses Verfahren wird Vorwärtsdeklaration bzw. Vorwärtsdefinition genannt.

```
char a_c;
main() {
  a_c = 0;
  b_c = 0;
}
```

```
Error 202: 'b_c': undefined identifier
```

```
char b_c;
void test(void){
  a_c = 1;
  b_c = 1;   // OK
}
```

Listing 24 Fehlerhafte Vorwärtsdeklaration

Um den Fehler aus Listing 24 bei der Vorwärtsdeklaration zu umgehen, können Sie die Variable am Anfang des Moduls deklarieren. Somit ist während der Compilierung der Prototyp der Variablen bekannt (siehe Listing 25 ①).

```
①  extern char b_c;
   char a_c;

   main() {
     a_c = 0;
     b_c = 0;   // OK
   }

   char b_c;

   void test(void){
     a_c = 1;
     b_c = 1;   // OK
   }
```

```
   Auszug aus M51-File

   VALUE           TYPE          NAME
   ----------------------------------
   -------         MODULE        GLO_EXT
   C:0000H         SYMBOL        _ICE_DUMMY_
②  D:0008H         PUBLIC        b_c
②  D:0009H         PUBLIC        a_c
   C:0016H         PUBLIC        main
```

Listing 25 Vorwärtsdeklaration einer Variablen

Global definierte Variablen sind vom Typ **public** (siehe ②), d.h., sie können auch von anderen Source-Modulen über eine externe Deklaration bekannt gemacht und somit ihr Wert verändert werden.

☞ !! Globale Variablen haben einen fest definierten Speicherplatz und sind von einer Überlagerung ausgenommen. !!

☞ !! Globale Variablen haben, soweit sie von Ihnen nicht explizit initialisiert wurden, beim Programmstart den Wert 0. Diese wird im Modul **startup.a51** vorgenommen. !!

static

Soll der Gültigkeitsbereich einer global definierten Variablen oder einer Funktion auf das eigene Source-Modul beschränkt sein, so müssen die Variablen bzw. die Funktionen mit dem Typ **static** versehen werden. Wird eine Variable, die in einer Funktion definiert wurde, mit static versehen, so wird der Speicherbereich der Variablen nach Verlassen der Funktion nicht freigegeben. Wird die Funktion erneut aufgerufen, so ist der zuletzt abgespeicherte Wert noch vorhanden.

☞ !! Beim C51-Compiler werden Variablen, die mit static definiert sind, vom Overlay Mechanismus ausgenommen (siehe Kapitel 15 Overlay-Technik). !!

static Datentyp Speichertyp Var_Name;
static Datentyp F_Name (Datentyp Var_Name, ..) {};

– **static:** Schlüsselwort in ANSI-C.
– **Speichertyp:** Diesen können Sie optional angeben (data, bdata, idata, pdata, xdata)
– **Datentyp:** bit, char, short, int, long, float, double
– **Var_Name:** Siehe Regeln des Variablennamen (Kapitel 4).
– **F_Name:** Siehe Regeln des Variablennamen (Kapitel 4).

```
extern char b_c;
static char a_c;

main() {
  a_c = 0;
  b_c = 0;   // OK
}
static char b_c;
```

Listing 26 Vorwärtsdeklaration von Variablen mit der Speicherklasse static

In Listing 26 ist die Gültigkeit der Variablen a_c und b_c auf das Source-Modul begrenzt, obwohl die Variable b_c die **extern** Deklaration enthält. Diese **extern** Referenzierung ist nur für das Modul gültig. Im Linker werden für das Beispiel folgende Angaben eingetragen:

```
LST-File Information:
 NAME                      CLASS   MSPACE  TYPE    OFFSET  SIZE
 ====                      =====   ======  ====    ======  ====
  a_c. . . . . . . . . .   STATIC   DATA   CHAR     0000H  1
  b_c. . . . . . . . . .   STATIC   DATA   CHAR     0001H  1
  c_c. .   . . . . . . .   PUBLIC   DATA   CHAR     0002H  1

M51-File Information:
 VALUE           TYPE          NAME
 ----------------------------------
 D:0008H         SYMBOL        A_C
 D:0009H         SYMBOL        B_C
 D:000AH         PUBLIC        C_C
```

Listing 27 Speicherbelegung von Variablen mit der Speicherklasse static

☞ !! Werden Funktionen mit static versehen, so wird ein Kapseln der Funktionen im Source-Modul erreicht. !!

lokal

Der Gültigkeitsbereich lokal kann nur auf Variablen bezogen werden, die in einer Funktion definiert sind. Somit ist auch nachvollziehbar, daß es keine externen Bezüge auf diesen Gültigkeitsbereich geben kann. Der Gültigkeitsbereich wird durch die {} Klammern eingeschränkt. Somit können Sie ein zusätzliches Kapseln von Variablen erreichen. Der C51-Compiler überprüft diese und bildet dementsprechend den Gültigkeitsbereich dafür nach. Mit diesem Verfahren können Sie Speicherplatz einsparen, ohne daß Sie eine zusätzliche Variable definieren, bzw. eine Variable anderweitig benutzen müssen. Werden lokale Variablen definiert, die denselben Namen wie globale Variablen aufweisen, so verwendet der C-Compiler bei Namensgleichheit immer die

lokale Variable in der Funktion. Sie haben keine Möglichkeit, an die globale Variable zu kommen.

```
   char a_c;

   void test(void){
     char a_c;

①    a_c++;
   }
```

Listing 28 Beispiel Namensgleichheit bei lokaler und globaler Variable.

In Listing 28 sind zwei gleichlautende Variablen definiert worden. Die Inkrement-Anweisung der Variablen a_c (siehe ①) wird auf die lokale Definition bezogen.

☞ !! Lokale Variablen haben, soweit sie von Ihnen nicht explizit initialisiert wurden, einen unbestimmten Inhalt (Fehler bei Variable a_c). !!

☞ !! Lokale Variablen sind dynamische Variablen und haben keinen festen Speicherplatz. Lokale Variablen unterliegen dem Overlay-Verfahren. !!

```
   void test(void){
     char a_c;
     static char b_c =33;

     if (a_c == 30) a_c = 0; // Fehler, da Wert von a_c unbekannt!!
     If (b_c > 3) b_c = 0;   // OK, da Datenerhalt mit static
   }                         // gegeben
```

Listing 29 Fehlabfrage auf undefinierte lokale Variable

```
   void test(void) {
     char a_c, b_c, c_c;
     char zaehl_c;

     b_c = 3;
     c_c = 4;
     a_c = c_c + b_c;
      for (zaehl_c = 0; zaehl_c <100; zaehl_c++);
   }
```

```
   MODULE INFORMATION:   STATIC OVERLAYABLE
      DATA SIZE        =   ----       4
   END OF MODULE INFORMATION.
```

Listing 30 Alle Variablen werden am Funktionsanfang definiert

In Listing 30 werden die Variablen am Funktionsanfang definiert. Dies hat den Nachteil, daß der C-Compiler für alle Variablen Speicherplatz reserviert.

```
   void test(void) {
     char a_c, b_c;
     char zaehl_c;

     b_c = 3;
     zaehl_c = 4;
     a_c = zaehl_c + b_c;
      for (zaehl_c = 0; zaehl_c <100; zaehl_c++);
   }
```

```
   MODULE INFORMATION:   STATIC OVERLAYABLE
```

```
   DATA SIZE        =   ----       3
END OF MODULE INFORMATION.
```

Listing 31 Eine Variable wird doppelt verwendet

In Listing 31 wird die Variable zaehl_c doppelt verwendet. Dies hat zwar den gewünschten Effekt, daß der Speicherbereich entlastet wird, erschwert aber die Lesbarkeit des Programms, so daß die Fehleranfälligkeit steigt.

```
test(){
  char a_c;
  {
   char b_c, c_c;

   b_c = 3;
   c_c = 4;
   a_c = c_c + b_c;
  }
  {
   char zaehl_c;
    for (zaehl_c = 0; zaehl_c <100; zaehl_c++);
  }
}
```

```
NAME                          CLASS   MSPACE  TYPE    OFFSET
SIZE
====                         =====   ======  ====    ======  ====
test . . . . . .  PUBLIC   CODE   PROC     -----  ----
  b_c. . . . . .  AUTO     DATA   CHAR     0001H  1
  c_c. . . . . .  AUTO     DATA   CHAR     0002H  1
  zaehl_c. . . . . .  AUTO     DATA   CHAR     0001H  1
  a_c. . . . . .  AUTO     DATA   CHAR     0000H  1
MODULE INFORMATION:   STATIC OVERLAYABLE
DATA SIZE        =   ----       3
END OF MODULE INFORMATION.
```

Listing 32 Die Variablen werden gekapselt definiert

In Listing 32 wird die gleiche Einsparung von Speicherplatz wie in Listing 31 erreicht. Das Programm ist jetzt wesentlich leichter lesbar. Die lokalen Variablen sind beim LST-File mit dem Typ **auto** bezeichnet. Die Überlagerung des Speichers können Sie aus dem LST-File entnehmen. In diesem Beispiel verwenden die Variablen b_c und zaehl_c denselben Speicherplatz.

extern

Werden Variablen mit dem Schlüsselwort extern versehen, so handelt es sich um eine Deklaration. Diese Variablen müssen in einem Source-Modul als global public definiert sein. Ansonsten wird beim Linkerlauf für diese Variablen eine Fehlermeldung ausgegeben.

2.6 Übungen

1. Wie muß ein H-File aufgebaut sein, damit es nicht mehrmals vom C-Prepro-zessor eingebunden wird?
2. Was ist in diesen Steueranweisungen falsch?

```
#pragma OE PP LC DB
#define XY 1.312

#include <2_CONVERT.H>

#if XY > 0
 float a_f = XY;
#endif
```

3. Wie können Sie die Verschachtelungstiefe in einer Funktion feststellen?
4. Erstellen Sie ein Makro, das einen Dreieckstausch vornimmt.
5. Was ist hier falsch?

```
#include<REG.H>

char test_it(char a_c){
  extern char b_c;
  if (b_c > 10) b_c = b_c + a_c;
}
```

6. Welchen Wert haben die Variablen a_c und erg_c?

```
char a_c = 0;

char test_it(char a_c){
  char b_c;
  b_c = a_c +1;
  return (b_c);
}

main() {
  char erg_c;
  erg_c = test_it(a_c);
}
```