Prof. Dr. Bernd vom Berg
Dipl.-Ing. Peter Groppe

´C´

Eine Kurzfassung für den Praktiker

Schwerpunkt: Die Programmierung von Mikrocontrollern in ´C´ am Beispiel der

IDE µC/51

der Firma Wickenhäuser Elektrotechnik

Stand: V4.0 - W / 17.08.2006

I n h a l t

1. Allgemeines

- Kennzeichnung von **Kommentaren**:
  /* Kommentar */ oder alternativ
  // Kommentar bis zum Zeilenende

- Kennzeichnung von **HEX-Zahlen**: durch vorangestellte 0x
  0x3a4 ≡ 3a4h

- **Zuweisungsoperator**:
  Variable Operator = Ausdruck
  ist identisch mit:
  Variable = Variable Operator (Ausdruck)

  Beispiel:
  x+=5; ist identisch mit x = x + (5);

- Einzelne **char-Variablen** werden in ' ' eingeschlossen
  Beachten: ' ≡ shift # !!

- **Mathematische Grundrechenarten**:
  + - * /
  Modulo: % ≡ Rest bei einer Integer-Division
  Beispiel: 14 % 5 =

2. Aufbau eines C-Programms

/* Kommentar: Sinn und Zweck des Programms, Autor, Version, Datum, etc. */

Anweisungen für den Präprozessor:
#....

Definition von globalen Variablen.

(A): Definition der selbst geschriebenen Funktionen.

Start des eigentlichen Hauptprogramms ≡ Start der Hauptfunktion main():
void main (void)
{
Zur Lösung der gewünschten Programmaufgabe:

Definition von lokalen Variablen,

Aufruf von C-Standard-Bibliotheks-Funktionen,

Aufruf von selbst geschriebenen Funktionen.
}
Ende des eigentlichen Hauptprogramms ≡ Ende der Hauptfunktion main().

(B): Alternativ können auch hier die Definitionen der selbst geschriebenen Funktionen stehen

3. Die Datentypen in C

Integer-Typen

Normale Typen: int
Immer 2 Byte lang
Untertypen:
- int:
  Wertebereich:
  -32768 ... +32767
- unsigned int:
  Wertebereich:
  0 ... +65535

Verlängerte Typen: long
Immer 4 Byte lang
Untertypen:
- long:
  Wertebereich:
  -2147483648 ... +2147483647
- unsigned long:
  Wertebereich:
  0 ... +4294967295

Verkürzte Typen: char
Immer 1 Byte lang
Untertypen:
- signed char:
  Wertebereich:
  -128 ... +127
- (unsigned) char:
  Wertebereich:
  0 ... +255

Sondertypen:
- ASCII-Zeichen

Float-Typen

Normale Typen: float
Immer 4 Byte lang
Wertebereich:
±1.175494E-38 ... ±3.402823E+38

Typen, spez. zur Programmierung von µC´s

Bit:
Untertypen:
- bit,
- unsigned char bit (SFR-Bits)

Wertebereich jeweils: 0, 1

Special Function Register (SFR) des Mikrocontroller:
- near unsigned char

Wertebereich: 0 ... +255

Datenspeicherstellen des externen RAM / SFR`s der externen Peripherieeinheiten
- xdata unsigned char

Die **Definition von Variablen** eines bestimmten Datentyps in einem C-Programm erfolgt durch bestimmte, festgelegte Schlüsselworte, die zuvor bereits erwähnt wurden:

Beispiele:

```
int i;                        // Variable mit Namen ´i´ von Typ Integer
unsigned char zeichen;        // Variable mit Namen ´zeichen´ vom
                              // Typ unsigned char
float mw,g1,g2;               // Float-Variablen ´mw´,´g1´,´g2
unsigned int mw_og,i1,wert;   // Unsigned Integer Variablen
                              //´mw_og´,´i1´,´wert´
```

Die Zuweisung von **einzelnen ASCII-Zeichen** zu char-Variablen geschieht mit einfachen Anführungszeichen:

```
zeichen = 'H';          // Zuweisung des ASCII-Zeichens ´H´ an die
                        // unsigned char Variable zeichen
                        // Achtung:  '  ≡  shift #  !!
```

Rechengenauigkeit bei Float-Zahlen: 6 Stellen nach dem Komma

Pointer: es gibt ebenfalls noch Pointer, die grundsätzlich 16 Bit breit sind.

4. Die Verwendung von Formatkennzeichen

Formatkennzeichen (FKZ) legen fest, in welchem Format und in welchem Zahlensystem ausgegebene Variablenwerte dargestellt werden.

Die Ausgabe von Variablen-Werten mit printf(…)

Jeder µC rechnet intern nur mit reinen Zahlen. Wie diese Zahlen dann nach außen hin interpretiert, d.h. im wesentlichen durch die printf()-Anweisung dargestellt werden, hängt sehr stark von den verwendeten FKZ ab.
FKZ´s werden mit in die printf()-Anweisung eingebaut.
FKZ´s sind immer gleich aufgebaut:

- ´%´-Zeichen, als Kennzeichen, dass jetzt eine Formatangabe folgt,
- ´Buchstabe´ zur Festlegung der Art des Ausgabe-Formates.

Formatkennzeichen für die Ausgabe von Variablenwerten durch die printf()-Funktion

Datentyp:	**Formatzeichen:**	**Bemerkung:**
unsigned char	%u	Ausgabe einer vorzeichenlosen 1 Byte großen Ganzzahl, im dezimalen Zahlensystem.
unsigned char	%x %X	Ausgabe einer vorzeichenlosen 1 Byte großen Ganzzahl, im hexadezimalen Zahlensystem. x ≡ mit kleinen Buchstaben. X ≡ mit großen Buchstaben.
signed char	%d	Ausgabe einer vorzeichenbehafteten 1 Byte großen Ganzzahl, im dezimalen Zahlensystem.
***	***	***
(unsigned) char	%c	Ausgabe eines Bytes, das als ASCII-Zeichen interpretiert und dargestellt wird (≡ Charakter).
***	***	***
unsigned int	%u	Ausgabe einer vorzeichenlosen 2 Byte großen Ganzzahl im dezimalen Zahlensystem.
unsigned int	%x %X	Ausgabe einer vorzeichenlosen 2 Byte großen Ganzzahl im hexadezimalen Zahlensystem. x ≡ mit kleinen Buchstaben. X ≡ mit großen Buchstaben.
int	%d	Ausgabe einer vorzeichenbehafteten 2 Byte großen Ganzzahl im dezimalen Zahlensystem.
***	***	***
unsigned long	%lu	Ausgabe einer vorzeichenlosen 4 Byte großen Ganzzahl im dezimalen Zahlensystem.
unsigned long	%lx %lX	Ausgabe einer vorzeichenlosen 4 Byte großen Ganzzahl im hexadezimalen Zahlensystem. x ≡ mit kleinen Buchstaben. X ≡ mit großen Buchstaben.
long	%ld	Ausgabe einer vorzeichenbehafteten 4 Byte große Ganzzahl im dezimalen Zahlensystem.
***	***	***
float	%f	Ausgabe einer Fließkommazahl in Form von: „Vorkommastellen **.** Nachkommastellen“, also: [-] dddd.dddd
float	%e %E	Ausgabe einer Fließkommazahl in Form von: „Mantisse **e** Exponent“, also: [-] d.ddd. **e** [-] dd. e ≡ der Exponent ist durch ein kleines **e** E ≡ der Exponent ist durch ein großes **E** gekennzeichnet.
float	%g %G	Hier wählt das Programm die kompaktere Ausgabeform aus, entweder %f oder %e , also: [-] dddd.dddd oder [-] d.ddd. **e** [-] dd. g ≡ der Exponent ist durch ein kleines **e**

		G ≡ der Exponent ist durch ein großes **E** gekennzeichnet

Hinweis: Eine Darstellung im **Binär-System** ist ***NICHT*** möglich !

Beispiele:

```
// Variablen-Definition
unsigned int w;

// Beispielhafte Wertzuweisung
w=90;            // w hat also den Wert 90 dezimal oder 5a hexadez.

// Aufbau eines möglichen Ausgabebefehls:
// Das FKZ dient als Platzhalter für den auszugebenden Wert
// Beachten: für FKZ wird nachfolgend der jeweilige Kennbuchstabe
// eingesetzt
// %  ≡  Kenzeichen für den Compiler, dass jetzt ein FKZ folgt.
// Beispiel:
//
//              printf("Das Zeichen ist: %FKZ ",w);

// Also ganz konkret:

printf("Das Zeichen ist: %d ",w);            // Dezimal
// Auf Bildschirm erscheint:      Das Zeichen ist: 90

printf("Das Zeichen ist: %x ",w);            // Hexadezimal, klein
// Auf Bildschirm erscheint:      Das Zeichen ist: 5a

printf("Das Zeichen ist: %X ",w);            // Hexadezimal, groß
// Auf Bildschirm erscheint:      Das Zeichen ist: 5A

printf("Das Zeichen ist: %c ",w);            // als ASCII-Zeichen
// Auf Bildschirm erscheint:      Das Zeichen ist: Z
// denn 5ah ist der ASCII-Code von ´Z´ !
```

Erweiterung von FKZ´s, insbesondere bei der Ausgabe von Float-Zahlen:

Bei den FKZ´s kann noch mit angegeben werden, mit wie vielen Vor- und Nachkomma-Stellen eine Zahl ausgegeben wird, wobei dann automatisch mathematisch korrekt gerundet wird:

%a.bFKZ

mit:

- a = Gesamtstellenanzahl für den Wert, incl. des Dezimalpunktes !

- ´.´ = Dezimalpunkt als Trennzeichen
- b = Anzahl der Nachkomma-Stellen

Beachten:

Bei der Ausgabe von **Integer Zahlen** gilt:

1) Das b-Feld wird ignoriert, braucht also gar nicht angegeben zu werden. Ebenso entfällt der Dezimalpunkt.
2) Das a-Feld gibt die Gesamtstellenanzahl der Ausgabe an.
3) Ist a größer als die auszugebende Zahl, so werden links Leerzeichen aufgefüllt.
4) Ist a kleiner als die auszugebende Zahl, so wird die Zahl trotzdem komplett ausgegeben, es gehen keine Stellen verloren.

Bei der Ausgabe von **Float-Zahlen** gilt:

1) a gibt nun die Gesamtanzahl der für die Ausgabe reservierten Stellen an, wobei der Dezimalpunkt als eine eigene Stelle mitgezählt wird. Das b-Feld gibt die Anzahl der Nachkomma-Stellen an:

   "%6.2f"

   bedeutet daher: insgesamt werden für f sechs Stellen reserviert bzw. f wird sechs-stellig ausgegeben mit:
   - zwei Nachkomma-Stellen,
   - einem Dezimal-Punkt,
   - drei Vorkomma-Stellen.

2) Muß, durch die b-Angabe bedingt, die Anzahl der Nachkomma-Stellen bei der Ausgabe reduziert werden, so wird automatisch entsprechend auf - oder abgerundet.

3) Ist die Anzahl der reservierten Vorkomma-Stellen größer die Vorkomma-Stellenanzahl der auszugebenden Float-Zahl, so werden rechtsbündig Leerzeichen ausgegeben.

4) Ist die Anzahl der reservierten Nachkomma-Stellen größer als die Nachkomma-Stellenanzahl der auszugebenden Float-Zahl, so werden die restlichen Nachkomma-Stellen mit Nullen aufgefüllt.

5) Ist die auszugebende Float-Zahl größer als der durch a und b reservierte Platz, so werden die Vorkomma-Stellen der Float-Zahl komplett, d.h. ohne Ziffernverlust, und die Nachkomma-Stellen der Float-Zahl gerundet ausgegeben.

Zusätzlich können Zahlen auch noch links- und rechtsbündig ausgegeben werden, mit führenden Nullen, etc.; s. dazu: C-Handbücher !

Beispiele:

```
// Variablen-Definition
unsigned int w;
float z;

// Beispielhafte Wertzuweisung
z=598.368;            // z hat also den Wert: 598,368 !!
w=90;

printf("Das Zeichen ist: %5d ",w);           // Integer, 5-stellig
// Auf Bildschirm erscheint:      Das Zeichen ist: ___90
// 3 Blanks vorangestellt.

printf("Das Zeichen ist: %6.2f ",z);         // Float mit
                                             // 2 Nachk., 3 Vork.
// Auf Bildschirm erscheint:      Das Zeichen ist: 598.37
// Mathem. gerundet

printf("Das Zeichen ist: %7.1f ",z);         // Float mit
                                             // 1 Nachk., 5 Vork.
// Auf Bildschirm erscheint:      Das Zeichen ist: __598.4
// Mathem. gerundet, 2 Blanks vorangestellt
```

Praxis-Empfehlung für die Verwendung von Datentypen und FKZ´s:

- Arbeiten mit SFR´s (Inhalts-Werte: 0 … 255)
  unsigned char … %u, %x, %X

- Variablen-Werte: 0 … 255
  unsigned char … %u, %x, %X

- Variablen-Werte: -128 … +127
  char … %d

- ASCII-Zeichen (Werte: 0 … 255)
  unsigned char %c

- Variablen-Werte: 0 … 65.535
  unsigned int … %u, %x, %X

- Variablen-Werte: -32.768 … +32767
  int … %d

- Variablen-Werte: $0 \ldots 2^{32} - 1$
  unsigned long … %lu, %lx, %lX

- Variablen-Werte: $-2^{31} \ldots +2^{31} - 1$
  long … %ld

- Variablen-Werte: Reelle Zahlen $\pm 1.175494 * 10^{-38} \ldots \pm 3.402823 * 10^{+38}$
  float … %f, %e

- Bit-Variablen: Werte 0, 1
  bit … ------- (keine direkte Ausgabe möglich)

Steuerzeichen in printf-Anweisungen

Steuerzeichen	**Bedeutung**
\a	**Alarm („Bell“)**
\\	Das \-Zeichen (Back Slash) wird ausgegeben
\b	Rückschritt nach links
\r	Wagenvorlauf
\“	Das “-Zeichen (doppeltes Hochkomma) wird ausgegeben
\f	Seitenvorschub (funktioniert u.U. nicht)
\t	Tabulatorschritt
\0	Das Nullzeichen (Stringende-Kennzeichen)
\‘	Das ‘-Zeichen (Hochkomma) wird ausgegeben
\v	Line-Feed
\n	**Wagenrücklauf mit neuer Zeile**
\c	Unterdrückung eines Zeilenvorschubs
\?	Das ?-Zeichen (Fragezeichen) wird ausgegeben
\onnn	Die oktale Zahl nnn wird ausgeben (Wertebereich 1Byte, 0..377 oktal)
\xnn	**Die hexadezimale Zahl nn wird ausgeben (Wertebereich 1 Byte, 0..ff hex.)**

5. Die Eingabe von Variablen-Werten mit inputse(…)

Da es bei µC/51 z.Zt. die ANSI-C-Funktion **scanf()** nicht gibt, muss bei der Eingabe von Variablen-Werten über die Tastatur des Terminals (≡ Einlesen von Zahlen-Werten über die serielle Schnittstelle des µCs) ein kleiner „Umweg“ mit Hilfe der Funktion **inputse(…)** gemacht werden !

Die Funktion inputse() liest zunächst immer einen String ein, aus dem dann in einem zweiten Schritt „eine Zahl gemacht wird", d.h. der String wird in eine Zahl umgewandelt.

Mit anderen Worte:
Die „Ein-Zeilen-Anweisung" scanf(…) aus dem ANSI-C wird durch eine „Zwei-Zeilen-Anweisung" bei µC/51 realisiert.

Da die Funktion inputse() bei µC/51 zur Standard Ein/Ausgabe gehört, muss bei ihrer Verwendung die Header Datei **stdio.h** per #include Anweisung eingebunden werden.

Beim Arbeiten mit inputse(…) müssen daher auch mindestens **zwei Variablen** definiert werden: eine String-Variable geeigneter Länge, die den eingegebenen String aufnimmt und eine Variable, die den umgewandelten String, also die gewünschte Zahl, aufnimmt.

Aufbau der Funktion: inputse(a, b);
a ≡ Stringvariable, die den einzulesenden String aufnimmt
b ≡ Anzahl der einzulesenden ASCII-Zeichen inklusive des Ende Zeichens (RETURN) ≡ Länge des Strings.

Beispiel:

```
char s[12];          // Def. des Strings s mit der Länge 12

inputse(s, 12);
```

In die Stringvariable s können nun maximal 11 ASCII-Zeichen (plus dem Ende Zeichen (RETURN)) eingelesen werden.

Um nun den mit inputse(…) eingelesenen String in eine **integer**, **long integer** oder **float** Zahl umzuwandeln, stehen drei weitere Funktionen zur Verfügung: **atoi(…)**, **atol(…)** und **atof(…).**

Während atoi(…) und atol(…) bei µC/51 zur Standard Ein/Ausgabe gehören, muss bei Verwendung von atof(…) zusätzlich noch die Header Datei **math.h** mit eingebunden werden.

Beispiele:

```
// Einbinden der Header-Dateien

#include <stdio.h>          // Zur Verwendung von inputse(...),
                            // atoi(...) und atol(...)
#include <math.h>           // Zur Verwendung von atof(...)

// Variablen-Definition
```

```
char s1[7], s2[10], s3[10];       // Strings zur Aufnahme der
                                  // einzulesenden Zeichen
int vfl;
long int bvb_s04;
float zahl;

// Beispielhaftes Einlesen

inputse(s1,7);   // Über die serielle Schnittstelle können max.
                 // 6 Zeichen plus <RETURN> in s1 eingelesen werden
```

*** Annahme, es werden die Zeichen <1> <8> <4> <8> und <RETURN> eingegeben ***

```
inputse(s2,10);  // Über die serielle Schnittstelle können max.
                 // 9 Zeichen plus <RETURN> in s2 eingelesen werden
```

*** Annahme, Eingabe von: <-> <1> <9> <0> <9> <1> <9> <0> <4> ***

```
inputse(s3,10);  // Über die serielle Schnittstelle können max.
                 // 9 Zeichen plus <RETURN> in s3 eingelesen werden
```

*** Annahme, Eingabe von: <-> <0> <.> <0> <9> <1> <9> und <RETURN> ***

```
// Nun: Beispielhaftes Umwandeln des eingelesenen Strings
// in integer, long integer und float Datentypen

vfl = atoi(s1);        // vfl besitzt den integer Zahlenwert 1848
bvb_s04 = atol(s2);    // bvb_s04 besitzt den long int Wert -19091904
zahl = atof(s3);       // zahl besitzt den float Zahlenwert -0,0919
```

WICHTIG: DIE EINGABE VON VARIABLEN MIT INPUTSE()

Beachtet werden muss, dass inputse(…) alle gültigen ASCII-Zeichen einliest, also eine Eingabe von **´04mies09´** (das ist ja auch ein gültiger ASCII-String !) zu keinem Fehler, jedoch zu einem völlig falschen Variablen-Wert bei der nachfolgenden Umwandlung (egal in welchen Datentyp) führt.
Mit anderen Worten:
Bei inputse(…) wird nicht überprüft, ob die Eingabe auch eine sinnvolle, d.h. gültige, Zahl darstellt. Hier muß der Programmierer selbst dafür sorgen, dass nicht falsches eingegeben bzw. weiter verarbeitet wird.

Ferner muss beachtet werden, dass bei der Eingabe von Gleitkomma Zahlen ein Punkt anstelle eines Kommas eingegeben wird, um Vor- und Nachkommastellen zu trennen, also:

23**.**56 anstatt 23**,**56 !

6. Variablenfelder (Arrays)

7. Datentyp: String

8. Programm-Konstrukte (Kontrollstrukturen)

if - Abfrage

```
if (Bedingung)                          // KEIN Semikolon
    {
        Anweisungsblock des if-Zweiges
    }
  else                                  // else-Zweig, optional, KEIN Semikolon
    {
        Anweisungsblock des else-Zweiges
    }
```

for - Schleife

```
for (Anweisung A; Bedingung; Anweisung B)        // KEIN Semikolon
    {
        Anweisungsblock der for-Schleife
    }
```

Abarbeitung:

1) Anweisung A ausführen.
2) Bedingung prüfen:
   - WAHR: Anweisungsblock ausführen und weiter bei 3).
   - FALSCH: Schleife verlassen.
3) Anweisung B ausführen.
4) Weiter bei 2).

Beachten:

1) Anstelle von Anweisung A und Anweisung B können auch mehrere Anweisungen, durch ***Komma*** getrennt, stehen.
2) Auf die Initialisierung der Laufvariablen (Anweisung A) kann auch verzichtet werden, wenn z.B. der Startwert zuvor eingegeben wird.
3) ***Zeitverzögerungsschleife*** mit for:

```
for (x=1; x<zv; x++);                  // zv = Wert für gewünschte Zeit-
                                       // verzögerung einsetzen
```

   z.B. `for (x=1; x<1000; x++);`

4) ***Endlosschleife*** mit for:

```
for (; 1 ;)
```

```
        {
            Anweisungsblock
        }
```

while - Schleife

```
while (Bedingung)                                   // KEIN Semikolon
    {
        Anweisungsblock der while-Schleife
    }
```

Beachten:

1) Realisierung einer Endlosschleife: Die Bedingung muß immer wahr sein !
   Beispiel:

```
        while(1)
         {
            Anweisungsblock
         }
```

do while - Schleife

```
do                                          // KEIN Semikolon
    {
        Anweisungsblock der do while-Schleife
    }
while (Bedingung);                          // Semikolon NICHT VERGESSEN !
```

Abarbeitung:
Solange die Bedingung wahr ist, wird der Anweisungsblock ausgeführt.

switch ... case - Unterscheidung

```
switch (variable)
    {
        case 1:     Anweisungsblock
                    break;
        case 2:     Anweisungsblock
                    break;
```

```
        .....          .....
        default: Anweisungsblock
                       break;
}
```

Beachten:

1) break; nicht vergessen, sonst werden ***ALLE*** nachfolgenden Anweisungen ausgeführt, auch die der anderen case-Blöcke !
2) Es sind verschiedene case-Anweisungen mit gleichem Ziel möglich:

```
case 1:
case 2:
case 3:     Anweisungsblock
            break;
```

break;

Die "break"-Anweisung unterbricht grundsätzlich immer

die Blockanweisung { ..... }, die for-, die do ... while- und die while-Schleife

in der sie steht, d.h. beim Auftreten von "break" in solch einem Programmkonstrukt verzweigt das Programm immer an das Ende dieses Konstruktes und macht mit den Anweisungen direkt nach diesem Konstrukt weiter.
So lassen sich z.B. Schleifenkonstrukte sehr einfach vorzeitig verlassen, wenn man z.B. in solch einer Schleife programmiert:

```
if (Bedingung) break;
```

Wenn also die Bedingung erfüllt ist, wird die Schleife sofort beendet, d.h. verlassen und mit den Anweisungen danach fortgefahren.

continue;

Diese Anweisung ist das "Gegenstück" zur "break"-Anweisung. Beim Auftreten von "continue" in einer:

- for-Schleife,
- do ... while-Schleife,
- while-Schleife

verzweigt das Programm immer zum Kopf der jeweiligen Schleife, d.h. der Schleifenkörper wird dann vom Anfang an wieder abgearbeitet.

Beachten:

1) Die Schleifenbedingung bei der for- bzw. der while-Schleife wird erneut überprüft, da sie ja am Schleifenkopf steht.
2) Eine erneute Anfangsinitialisierung der Schleifenvariablen bei der for-Schleife findet jedoch **nicht** statt.

return(...);

Dient zur Rückgabe von Werten aus einer Funktion (s. "Selbstgeschriebene Funktionen").

9. Grundlegende C-Funktionen aus den Standardbibliotheken

Standardbibliothek: *stdio.h (º Standard Input/Output)*

unsigned char getchar(void); (identisch mit getc() und _getc())

Ein einziges ASCII-Zeichen (Byte) wird über die serielle Schnittstelle eingelesen. Das empfangene Zeichen wird **nicht** als Echo wieder über die serielle Schnittstelle ausgesandt:

```
unsigned char c1;
c1=getchar();
```

Beachten: Eine gedrückte RETURN-Taste (am Terminal) stellt auch ein gesendetes Zeichen dar.

void putchar(unsigned char); (identisch mit putc() und _putc())

Ausgabe eines einzelnen ASCII-Zeichens (Bytes) über die serielle Schnittstelle:

```
putchar(58);
```

printf(...);

Formatierte Ausgabe von Daten über die serielle Schnittstelle.

inputse(…);

Eingabe von mehreren ASCII-Zeichen (String) über die serielle Schnittstelle.

_wait_ms(...)

Erzeugung definierter Wartezeiten im Millisekundentakt:

```
_wait_ms(500);          // µC wartet eine halbe Sekunde und macht „nichts“
```

Beachten: Damit die Wartezeiten genau eine Millisekunde lang sind, muss per MakeWiz auf der Registerkarte Components der zum Quarz gehörende Wert für CPU-Speed in nsec/Instr. eingegeben werden. Für den 11,0592MHz Quarz beträgt der zugehörige Wert 1085, bei 12,00 MHz genau 1000 !

Weitere Funktionen der Standardbibliothek: *stdio.h*

Funktion:	**Beispiel:**	**Wirkung:**
int atoi(far char* pc)	a = atoi("-2796");	**Umwandlung string in integer:** Wandelt den String in eine 16-Bit Ganzzahl um.
long int atol(far char* pc)	a = atol("296796");	**Umwandlung string in long integer:** Wandelt den String in eine 32-Bit Ganzzahl um.
unsigned int rand(void)	a = rand();	**Zufallszahl:** Erzeugt eine 16-Bit vorzeichenlose Zufallszahl.
unsigned int rand8(void)	a = rand8();	**Zufallszahl:** Erzeugt eine 8-Bit vorzeichenlose Zufallszahl.
unsigned int rand1(void)	a = rand1();	**Zufallszahl:** Erzeugt eine 1-Bit vorzeichenlose Zufallszahl (0 oder 1).

Standardbibliothek: *math.h (#include<math.h>)*

Funktion:	**Beispiel:**	**Wirkung:**
float atof(far char* pc)	a = atof("-2.796");	**Umwandlung string in float:** Wandelt den String in eine Gleitkommazahl um. Wichtig: Kein Komma sondern Dezimalpunkt zur Trennung von Vor- und Nachkommastelle eingeben!
float cos(float val)	a = cos(b);	**Cosinus:** Berechnet den Cosinus von b im Bogenmaß.
float exp(float val)	a = exp(b);	**Exponent:** Berechnet den Exponetialwert von e^b, wobei e die Eulersche Zahl (=2,7183) ist.
float log(float val)	a = log(b);	**Logarithmus Naturalis:** Berechnet den natürlichen Logarithmus von b zur Basis e.
float log10(float val)	a = log10(b);	**Dekadischer Logarithmus:** Berechnet den Logarithmus von b zur Basis 10.
float pow(float x, float y)	a = pow(b, c);	**Potenzfunktion:** Berechnet den Exponetialwert von b^c.
float sin(float val)	a = sin(b);	**Sinus:** Berechnet den Sinus von b im Bogenmaß.
float sqrt(float val)	a = sqrt(b);	**Quadratwurzelfunktion:** Berechnet die Quadratwurzel von b ($a=\sqrt{b}$)

Des weiteren sind in der Bibliothek folgende drei Konstanten definiert:

M_PI_2 (= 1,570796326794895)
M_PI (= 3,14159265358979)
M_TWO_PI (= 6,28318530717958)

10. Vergleichsoperatoren

```
a < b      ≡   KLEINER
a <= b     ≡   KLEINER GLEICH
a > b      ≡   GRÖSSER
a >= b     ≡   GRÖSSER GLEICH
a == b     ≡   GLEICH          // ACHTUNG: 2 Mal ´=´ verwenden !
a != b     ≡   UNGLEICH        // ≡ Ungleich; ´!´ ≡ Negations-Operator
```

Zuordnungen: "Wahr"≡ Wert ? 0, i.a. Wert = 1
"Falsch" ≡ Wert = 0

Negation einer Aussage: !(Aussage)
if (!(a==1)) ....≡ "Wenn a nicht gleich 1 .... "
≡ if (a!=1) .....

Beispiele:

```
if (x)
        {
                Anweisungsblock, wird also immer ausgeführt bei x ? 0
        }

if (!x)
        {
                Anweisungsblock, wird also immer ausgeführt bei x = 0
        }
```

11. Logische Operatoren

zur Verknüpfung von ***Aussagen*** !

&&	≡	log. AND
||	≡	log. OR
!	≡	Negations-Operator

Beispiel:

Realisierung der Antivalenz (EXOR) durch:

((x && !y) || (!x && y))

12. Bitweise Operatoren

Spaltenweise Verknüpfung der einzelnen Bits einer Zahl !

& ≡ Bitweises AND

x = x & 7 ≡ x & = 7

| ≡ Bitweises OR

x = x | 7 ≡ x | = 7

^ ≡ Bitweises EXOR (Antivalenz)

x = x ^ 7 ≡ x ^ = 7

Beispiel: 47h ^ afh = ?

~ ≡ NOT (Invertierung)

<<

>> ≡ Verschiebeoperatoren, wobei gilt:

- Hinausgeschobene Bits gehen verloren
- Reingezogene Bits = log.´0´
- Anzahl der Stellen der Verschiebung müssen mit angegeben werden:

```
x = x << 3;        // Verschiebung um 3 Bitstellen nach links.
identisch mit:
x << = 3;
```

13. Weitere Operatoren

Bedingungsoperator: ? ... :

Vereinfachung bei Zuweisungen und Ersatz von if-Anweisungen in manchen Fällen:

Beispiel:

```
max = (a > b) ? a : b
≡       max = a,   wenn (a > b),                d.h. Bedingung erfüllt
        max = b,   wenn nicht (a > b), d.h. Bedingung nicht erfüllt
```

14. Selbstgeschriebene Funktionen

Selbstgeschriebene Funktionen bilden den Kern der ***modularen Programmierung*** in C. Man unterscheidet hierbei:

- Funktionen ohne Parameter-Übergabe und ohne Wert-Rückgabe
- Funktionen ohne Parameter-Übergabe und mit Wert-Rückgabe
- Funktionen mit Parameter-Übergabe und ohne Wert-Rückgabe
- Funktionen mit Parameter-Übergabe und mit Wert-Rückgabe.

Definition von Funktionen i.a.:

- nach den globalen Variablen und Konstanten des Hauptprogramms,
- vor dem eigentlichen Hauptprogramm bzw. vor der Hauptfunktion main.
  (s. "Aufbau eines C-Programms, s. S.4")

Funktionen ohne Parameter-Übergabe und ohne Wert-Rückgabe

```
void    fkt_name  (void)                    // KEIN Semikolon in dieser Zeile !
                                            // ≡ Funktionskopf
        {
                Funktionskörper
        }
```

Aufruf der Funktion durch den Funktions-Namen:

```
fkt_name();
```

Funktionen ohne Parameter-Übergabe und mit Wert-Rückgabe

```
Datentyp fkt_name  (void)                    // KEIN Semikolon in dieser Zeile !
                                             // ≡ Funktionskopf
    {
        Funktionskörper mit:
          - Rückgabe des Ergebniswertes der Funktion mit return(.....); ent-
            sprechend dem Datentyp der Funktion.
          - Jeder "Endpunkt" der Funktion muß mit return(...); abgeschlossen
            werden.
    }
```

Der Rückgabewerte entspricht dem Datentyp der Funktion.

Aufruf der Funktion durch den Funktions-Namen. Zuordnung des Funktions-Rückgabewertes zu einer Variablen oder Einbau in einen entsprechenden Ausdruck:

```
Variable = fkt_name();              // Variable entspricht dem Datentyp der Funktion
printf("%FKZ",fkt_name());          // FKZ ≡ entsprechendes Formatkennzeichen
```

Rückgabe von Wahrheitswerten:

```
return(1);          ≡     Wahr;  bzw.: immer Rückgabe eines Wertes ? 0 !
return(0);          ≡     Falsch
```

Funktionen mit Parameter-Übergabe und ohne Wert-Rückgabe

```
void fkt_name (int a, unsigned char c, float f1)        // KEIN Semikolon !
    {
        Funktionskörper mit:
          - Ersetzung der Formal-Parameter durch die Aktual-Parameter
    }
```

An die Funktion werden hier drei verschiedenen Parameter übergeben. Es können auch berechenbare Ausdrücke übergeben werden.

Aufruf der Funktion durch den Funktions-Namen und Übergabe der Parameter:

```
fkt_name(-45,´k´,123.567*z);
```

Funktionen mit Parameter-Übergabe und mit Wert-Rückgabe

```
Datentyp fkt_name (int a)                    // KEIN Semikolon !
    {
        Funktionskörper mit:
          - Ersetzung der Formal-Parameter durch die Aktual-Parameter.
          - Rückgabe des Ergebniswertes der Funktion mit return(.....); ent-
            sprechend dem Datentyp der Funktion.
          - Jeder "Endpunkt" der Funktion muß mit return(...); abgeschlossen
            werden.
    }
```

An die Funktion wird hier ein Parameter übergeben. Es können auch berechenbare Ausdrücke übergeben werden.
Der Rückgabewerte entspricht dem Datentyp der Funktion.

Aufruf der Funktion durch den Funktions-Namen und Übergabe der Parameter. Zuordnung des Funktions-Rückgabewertes zu einer Variablen oder Einbau in einen entsprechenden Ausdruck:

```
Variable = fkt_name(2389);          // Variable entspricht dem Datentyp der Funktion
printf("%FKZ",fkt_name(4711));      // FKZ ≡ entsprechendes Formatkennzeichen
```

Rückgabe von Wahrheitswerten:

```
return(1);      ≡    Wahr;  bzw.: immer Rückgabe eines Wertes ? 0 !
return(0);      ≡    Falsch
```

15. Lokale, globale und externe Variablen

Lokale Variablen

besitzen ihre Gültigkeit in selbstgeschriebenen Funktionen, sind nur innerhalb dieser Funktion bekannt:

```
void fkt_1(void)
        {
                int i1;                 // i1 ist nur in fkt_1 bekannt
                Restlicher Funktionskörper
        }
```

Außerhalb von fkt_1 ist i1 nicht bekannt, d.h. ein Zugriff auf i1 führt dann zu einer Fehlmeldung.

Globale Variablen

besitzen im gesamten Programm (Modul, Datei) Gültigkeit, d.h. vom gesamten Programm aus (von der Hauptfunktion main() und von allen anderen Funktionen aus) kann auf diese Variablen zugegriffen werden.
Globale Variablen werden direkt am Anfang des Programms, außerhalb jeder Funktion, definiert (s. "Aufbau eines C-Programms"):

```
// Globale Variablen: von überall her kann auf c1 und c2 zugegriffen werden
unsigned char c1,c2;

// Def. selbstgeschriebener Funktionen

void main(void)
        {
                Hauptfunktion
        }
```

Beachten:

1) Globale und lokale Variablen können durchaus die gleichen Namen haben. Da sie aber unterschiedliche Gültigkeitsbereiche haben, beeinflussen sie sich nicht !
2) Bei geschachtelten Funktionsaufrufen hat man ebenfalls **KEINEN Zugriff** von außen auf die lokalen Variablen in den geschachtelten Funktionen !

Externe Variablen

werden verwendet, wenn Modul- bzw. Datei-übergreifend auf Variablen zugegriffen werden soll:

Im Modul (Datei) A wird eine Variable namens ´var1´ definiert:

```
int var1;          // Definition als globale Variable
```

Im Modul (Datei) B bzw. vom Modul B aus soll nun ebenfalls auf diese Variable zugegriffen werden:

```
extern int var1;   // Die Variable var1 wird hier also nicht erneut
                   // angelegt, sondern über ´extern´ wird der Bezug
                   // zur Variablen ´var1´ im Modul A hergestellt:
                   // ´var1´ ist in beiden Modulen also die gleiche
                   // Variable !
var1=5;            // Zugriff nun möglich
```

Anmerkung:

Während globale Variablen den unbeschränkten Zugriff auf Variablen innerhalb eines einzelnen Moduls (Datei) festlegen, dient die externe Variablendeklaration dazu, auf Variablen aus unterschiedlichen Modulen (Dateien) zuzugreifen (≡ Erweiterte Globalisierung des Variablen-Zugriffs auf Modul- bzw. Datei-Ebene).

16. Header-Dateien

17. Vektoren und Pointer (Zeiger)

18. Präprozessor-Direktiven

#define

Dient zur symbolischen Adressierung: einem Namen wird ein Wert oder ein Ausdruck zugewiesen.

Beispiel:

```
#define cs_basis        0x5000
```

≡ dem Namen ´cs_basis´ wird der Wert 0x5000 zugeordnet.
≡ ´cs_basis´ wird daraufhin im Programmtext vom Präprozessor "ganz stur" durch 0x5000 ersetzt.

#include

Einbinden, d.h. hineinkopieren, von Datei-Inhalten:

```
#include "dat_1"          // dat_1 steht im aktuellen Arbeitsverzeichnis
#include <dat_2>          // dat_2 ist eine hersteller-spezifische Datei des Compiler-
                          // Herstellers und steht immer im Verzeichnis *.INC
```

#pragma

Angabe von herstellerspezifischen Steueranweisungen an den Präprozessor bzw. Compiler, die nicht allgemein in C genormt sind, z.B. Auswahl des Speichermodells bei der Programmierung von µC´s.

19. Mikrocontroller-spezifische Besonderheiten

Für die Programmierung von Mikroprozessoren und Mikrocontrollern wurde die ´C´-Sprache um einige Besonderheiten erweitert:

Zusätzliche Datentypen

Typ:	Var.-Def.	Werte-bereich:	FKZ:	Größe:
Bit-Variable im bitadressierbaren, internen Daten-speicherbereich	**bit Variablenname;** Mit: Variablenname ≡ beliebiger Name,	0, 1		1 Bit
Bit im bitadressierbaren Special Function Register (SFR)	**unsigned char bit Bitname @ Bitadresse** Mit: Bitname ≡ beliebiger Name, Bitadresse ≡ reale Bitadresse laut Datenblatt	0, 1		1 Bit
Special Function Register (SFR) des Mikrocontrol-lers	**near unsigned char SFR-Name @ SFR-Adresse** Mit: SFR-Name ≡ beliebiger Name, SFR-Adresse ≡ reale SFR-Adresse laut Datenblatt	0 .. 255	%x, %X, %u	1 Byte
Byte im externen Datenspeicherbe-reich, z. B. SFR der RTC	**xdata unsigned char Name @ Adresse** Mit: Name ≡ beliebiger Name, Adresse ≡ reale Adresse im externen Datenspeicherbereich	0 .. 255	%x, %X, %u	1 Byte

Beachten:
Bei den 8-Bit-µC´s der 8051er Familie sind die Special Function Register (SFR) und die externen Datenspeicherstellen i.a. 1 Byte groß.

Besonderheiten zu ´Bit´:

1) Die Bit-Variablen werden im bitadressierbaren internen Datenspeicherbereich des jeweiligen µC´s abgelegt. Bei der Maximalanzahl dieser Bit-Variablen sind daher die entsprechenden Beschränkungen zu beachten. Bei 8051er-Mikrocontrollern liegt diese Anzahl i.a. bei 128 Stück.

2) Bits in den bitadressierbaren SFR´s werden direkt über ihre Bit-Adresse definiert:
unsigned char bit Bitname @ Bitadresse

3) Es gibt keine Bit-Arrays.

Besonderheiten zu Special Function Registern des µC´s

1) Special Function Register werden direkt über ihre SFR-Adresse definiert:
   near unsigned char SFR-Name @ SFR-Adresse

2) Die Adresse des SFR´s ist dem Datenblatt des verwendeten Mikrocontroller zu entnehmen!

Besonderheiten zum externen Datenspeicherbereich

1) Speicherstellen im externen Datenspeicherbereich sind Special Function Register der externen Peripherie Einheiten oder RAM Speicherstellen. Beide Typen werden identisch definiert:
   xdata unsigned char Name @ Adresse

2) Die Adressen der externen Datenspeicherstellen sind der Hardware Dokumentation zum jeweiligen µC-Board zu entnehmen.

Beispiele:

```
// Definition von Bit-Variablen
bit Alarm, Zustand;           // Definition der Bit-Variablen Alarm und
                              // Zustand

// Definition der Bits in bitadressierbaren Special Function Register
// des Mikrocontroller
unsigned char bit EA @ 0xaf;        // Definition des Bits EA
unsigned char bit TR1 @ 0x8e;       // Definition des Bits TR1

// Definition der Special Function Register des Mikrocontroller
near unsigned char TMOD @ 0x89;     // Definition des SFR´s TMOD
near unsigned char SCON @ 0x98;     // Definition des SFR´s SCON

// Definition der Special Function Register von externen
// Peripherieeinheiten
xdata unsigned char SEC0 @ 0xfff0;  // Definition des Sek.-Einer-
                                    // Registers der RTC
xdata unsigned char MIN10 @ 0xfff3; // Definition des Min. Zehner-
                                    // Registers der RTC
```

20. Interrupt-Behandlung unter ´C´

Die Kernaussage beim Arbeiten mit Interrupts lautet:

„Beim Auftreten eines Interrupts wird das Hauptprogramm unterbrochen und eine zuvor speziell für diesen Interrupt geschriebene Interrupt-Funktion (die so genannte Interrupt Service Routine (ISR)) aufgerufen und abgearbeitet, wenn der Interrupt zuvor frei geschaltet wurde."

Dementsprechend müssen durch den Anwender (C-Programmierer) zwei Punkte gelöst werden:

1) Wie schreibt man eine „zugehörige Interrupt Service Routine" ?

2) Wie bringt man den Mikrocontroller dazu, beim Auftreten des Interrupts genau zu dieser speziellen Interrupt Service Routine zu springen ?

Das **Problem** hierbei ist:

Die Antworten zu diesen beiden Fragen sind in der C-Syntax NICHT international genormt, da ´C´ ja zunächst nicht für Mikrocontroller entwickelt worden ist.
Vielmehr sehen die Lösungen bei jedem C-Compiler-Hersteller anders aus und müssen daher immer im jeweiligen Handbuch nachgelesen werden! (Sie sind zwar ähnlich, aber eben nicht gleich !)

Die nachfolgenden Erläuterungen beziehen sich daher ganz konkret (nur) auf die **IDE µC/51** der Firma Wickenhäuser Elektrotechnik !

Soll unter µC/51 mit Interrupts gearbeitet werden, so geschieht dieses in **vier Schritten**.
Wir machen das am Beispiel des Interrupts des Programmable Counter Arrays PCA des CC03ers klar:

PCA Interrupt:

- Interrupt-Vektor-Adresse: 0x33

Schritt 1:

Im Verzeichnis ´uC51\include´ befindet sich die Datei ´irq52.h´.
In dieser Datei sind bereits die Interrupt-Vektor-Adressen der jeweiligen 8051er-Standard-Interrupts hinterlegt.
Findet man die Adresse des gewünschten Interrupts dort nicht eingetragen, so muß man dieses zunächst einfach nachholen, **Abb.1**:

```
// Available Interrupts on 8051/8052 with known names (IRQ_
added to avoid conflicts)

#define IRQ_INT0 0x3
#define TIMER0    0xB
#define IRQ_INT1 0x13
#define TIMER1    0x1B
#define SERIAL    0x23
#define TIMER2    0x2B

// Hier: Eigene Festlegungen fuer CC03er
#define PCA_INT    0x33
```

Abb.1: Ausschnitt aus der Datei irq52.h

Mit der letzten ´#define´- Anweisung haben wir dem PCA-Interrupt einen (wahlfreien) Namen geben (hier: PCA_INT) und diesem Namen die zugehörige Interrupt-Vektor-Adresse (hier: 0x33) zugeordnet.
Danach wird die Datei ´irq52.h´ wieder abgespeichert.

Schritt 2:

Im C-Programm selber muß nun diese Header-Datei ´irq.52.h´ einfach, wie gehabt am Programm-Anfang, eingebunden werden, z.B. wie folgt:

```
/*** Einbinden von Include-Dateien ***/
#include <stdio.h>            // Standard Ein-/Ausgabefunktionen
```

```
#include <math.h>              // Mathematische Funktionen
#include <irq52.h>             // Festlegung der IR-Vektoradressen
#include <string.h>            // Stringverarbeitung
```

Damit sind nun alle in ´irq52.h´ festgelegten Interrupts mit ihren Interrupt-Vektor-Adressen im Programm verfügbar.

Schritt 3:

Nun können Sie die Interrupt-Service-Routine (≡ Funktion, die aufgerufen werden soll, wenn der entsprechende Interrupt auftritt) einfach schreiben.
Wir nennen die Funktion einmal: ´pca_da´ und damit ergibt sich:

```
void pca_da (void) interrupt
  {
     .....
          // Funktionsrumpf
     .....
  }
```

D.h. der Funktionskopf wird einfach durch das Schlüsselwort ´interrupt´ ergänzt, sonst bleibt alles wie gehabt.

Schritt 4:

Im letzten Schritt muß nun die gerade geschriebene Interrupt-Service-Routine noch mit dem zugehörigen Interrupt bzw. mit der zugehörigen Interrupt-Vektor-Adresse verbunden werden und das geschieht mit einem entsprechend vordefinierten Makro:

```
/*** Interrupt Service-Routine zum PCA-Interrupt ***/

// Zuordnung der nachfolgenden IR-Funktion zur zugehörenden
// IR-Vektor-Adresse
IRQ_VECTOR(pca_da,PCA_INT)


/*** Interrupt-Funktion: PCA-Interrupt ***/

void pca_da (void) interrupt
  {
     // Funktionsrumpf
  }
```

Das Makro lautet ´IRQ_VECTOR(…)´

```
IRQ_VECTOR(pca_da,PCA_INT)
```

und in der Klammer wird zuerst der Name der zuzuordnenden Interrupt-Service-Routine (hier pca_da) und dann der Name des auslösenden Interrupts, wie in der Datei ´irq52.h´ festgelegt (hier PCA_INT), übergeben.

Dieses Makro wird sinnvoller Weise, wie zuvor gezeigt, immer direkt vor der eigentlichen Interrupt-Service-Routine in das Programm eingefügt.

Und das war´s schon !

Das weitere Arbeiten mit Interrupts, wie Festlegung der Interrupt-Bedingungen, Enable/Disable der Interrupts, etc. im C-Programm, geschieht nun wie gewohnt.