Digitaltechnik

Festverdrahtete Schaltung zur Lösung einer konkreten Aufgabe.
Bei Veränderung der Aufgabe Neuentwurf und Implementierung der Schaltung notwendig.

Mikrorechentechnik

Mehr Flexibilität durch Aufteilung in:

Hardware	**Software**
universelle Digitalschaltung (μR), deren Verhalten durch ein Programm (Software) gesteuert wird	Folge von Anweisungen (Programm) z.B. Zustände der Flipflops innerhalb eines Speicher-/Steuerregisters

- Bei Veränderung der Aufgabe nur Neuentwurf und Implentierung der Software notwendig
- Lösung vieler Aufgaben mit der selben Hardware möglich

Einfache Programmiersprachen

Assembler: Programm benutzt spezielle Hardwareresourcen wie Register, Interrupts und wird vom Assembler in für diese Hardware speziellen Maschinencode übersetzt. Programmierer gibt Mneomics (Kürzel z.B. MOV, ADD, IN, OUT) ein, welche vom Assembler mit Hilfe einer simplen Umsetzungstabelle (LookUp Table) 1:1 in Maschinenbefehle umgesetzt werden (NOP -> 80h).

Vorteil: - sehr effizienter Maschinencode

Nachteil: - aufwendigere Programmentwicklung
- hardwareabhängig (Programm nicht ohne Veränderung auf anderem μR benutzbar)

Hardwareabstraktion – Höhere Programmiersprachen

C, C++, Java, Pascal, Delphi:

Hardwarehersteller implementiert zusätzlich einen Treiber.
Anwendungsprogrammierer benutzt nicht mehr direkt Register und Interrupts, sondern schreibt in der Programmiersprache entsprechenden hardwareunabhängigen Code.
Compiler übersetzt Programm mit Hilfe des Treibers in für die konkrete Hardware abhängigen Maschinencode.

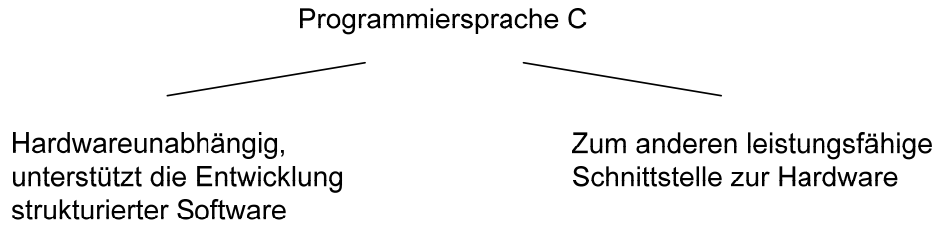
Vorteil: - Programm hardwareunabhängig
(andere Hardware $\rightarrow$ anderer Treiber vom Hersteller
Programm kann unverändert weiterverwendet werden:
- muss neu compiliert werden: C, C++, Pascal, Delphi
- muss nicht neu compiliert werden: Java, .NET)

Nachteil: nicht ganz so effektiver (bzgl. Programmgröße, Ausführungsgeschwindigkeit) Maschinencode

1. Sprachbeschreibung ANSI – C

1.1 Einführung

- viele Mikrocontroller besitzen im Gegensatz zum PC kein Betriebssystem (WindowsXP, Unix) und sind nicht mit einer objektorientierten Programmiersprache (C++, C#, Java) programmierbar
- Programmierung nur in C oder Assembler möglich
- C stellt Kompromiss zwischen höherer Programmiersprache und Assemblersprache dar:



Entwicklung

- 1978 K & R – C (Keringhan & Ritchie)
  Unix 6 zu 95% in K & R-C geschrieben

- 1988 ANSI-C Standard durch ANSI-Komitee

Sprachumfang ist der Speicherausstattung und Geschwindigkeit der oftmals verwenden 8-Bit und 16-Bit Mikrocontroller angepasst.

1.2 Programmstruktur

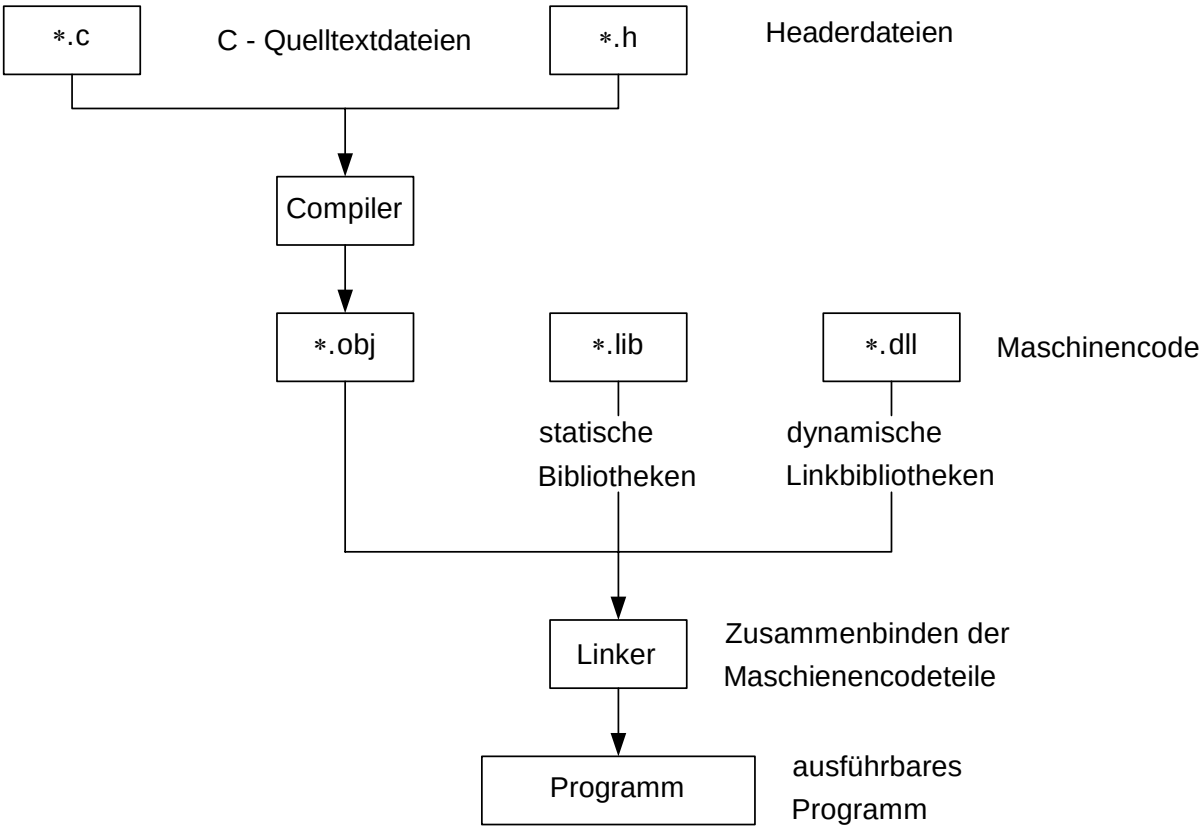
```
#include < stdio.h >
    main ( )                        /*  Kommentar !  */
{
    printf (”Mein erstes C-Prorgamm ! \n ”) ;
}
```

Jedes C-Programm besteht aus Funktionen.
Die main ( ) – Funktion bildet das Hauptprogramm.
Alle anderen Funktionen sind Unterprogramme.
In jedem C-Programm muss main ( ) vorhanden sein.

Erklärung:

main ( )	Hauptfunktion
{ }	Funktionskörper
#include < stdio.h >	Einfügen der Standardbibliothek zur Ein- und Ausgabe
printf ( )	Bibliotheksfunktion zur Ausgabe
;	Anweisungen werden mit ; abgeschlossen
/* Kommentar */	Text innerhalb /* */ wird vom Compiler nicht beachtet (// Zeile auskommentiert)

Allgemein:



1.3 Grunddatentypen

Datentyp: Menge von Werten und Operationen, die auf diese angewendet werden können.

Datentyp:

char	1 Byte	ASCII-Zeichen
short	1 Byte	ganzzahliger, vorzeichenbehafteter Wert – 128 , ... , 127
int	2 Byte	ganzzahliger, vorzeichenbehafteter Wert – 32768 , ... , 32767
long	4 Byte	ganzzahliger, vorzeichenbehafteter Wert
float	Gleitkommazahl	3 Byte Mantisse
		1 Byte Exponent
double	Gleitkommazahl	6,5 Byte Mantisse
		1,5 Byte Exponent

1.4 Bezeichner

Namen für Konstanten, Variablen, Funktionen:

- Bestehen aus Buchstaben, Ziffern oder _
- Es wird zwischen Groß- und Kleinschreibung unterschieden !
- Leerzeichen sind nicht zugelassen

Deklaration: Bedeutung eines Bezeichners wird festgelegt

Definition: Bedeutung eines Bezeichners wird festgelegt und Speicherplatz reserviert (allokiert)

1.5 Konstanten

```
const  int  x = 5 ;                    /* Definition der Konstanten x = 5 */
# define      PI      3.14             /* Gleitkommakonstante */
# define      LOGO    " HS-ZIGR "      /* Zeichenkettenkonstante */
```

allgemein:

```
# define      < bezeichner >      < zeichenfolge >
```

Präprozessor ersetzt die symbolische Konstante < bezeichner > im nachfolgenden Quelltext durch < zeichenfolge >. In spitze Klammern eingeschlossene Begriffe dienen als Platzhalter für im konkreten Fall einzusetzende Ausdrücke.

1.6 Variablen

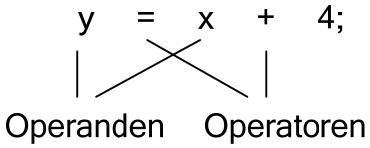
Symbolische Repräsentanten von Speicherplatz z.B.:

```
int  x;          /*  x  ist Variable für ganze Zahl */
float  y,z ;     /* y,z  sind Variablen für reelle Zahlen */
int  Y ;         /* Y ist Variable für ganze Zahl */
```

- Bestehen aus Buchstaben, Ziffern oder _
- Es wird zwischen Groß- und Kleinschreibung unterschieden !
- Leerzeichen sind nicht zugelassen -> siehe 1.4 Bezeichner

```
x = 0xF7;       /*  x bekommt den Hexadezimalwert F7 (=247) zugewiesen   */
                /*   Hexadezimalzahlen haben den Präfix 0x                */
x = 3.14;       /*   oder x = 0.314E1; x bekommt reelle Zahl 3.14 zugewiesen*/
```

Ausdruck: Folge von Operanden und Operatoren



1.7 Ausdrücke und Operatoren

1.7.1 Wertzuweisungsoperator =

```
x = 0;
y = x + 4;
```

Der rechts von = stehende Ausdruck wird berechnet und sein Wert der Variablen links vom = zugewiesen.

1.7.2 Arithmetische Operatoren

+ , -	Addition, Subtraktion
* , /	Multiplikation, Division
%	Rest der ganzzahligen Division

z.B.

```
int   x,y,z;

x = 5;
y = 2;

z  = x % y ;            /* z = 1 */
```

1.7.3 Inkrement- und Dekrement Operatoren

```
++ < Value >
- -   < Value >
      < Value >  ++
      < Value >  - -
```

z.B.

```
int   x,y ;

x = 3 ;
y = ++x ;               /*  x = x + 1 ;   y = x ;  */
                        /*  y = 4  */
x = 3 ;
y = x++ ;               /*  y = x ;  x = x + 1  */
                        /*  y = 3  */
```

1.7.4 Vergleichsoperatoren

```
> , > =          /* grösser, grösser gleich */
< , < =          /* kleiner, kleiner gleich */
= =              /* gleich */
! =              /* ungleich */
```

z.B.

```
x = 4;
y = 2;

wert =  x > y;                /*  wert = 1 */
wert =  x = = y;              /*  wert = 0 */
```

1.7.5 Logische Operatoren

```
& &              /*  logisches UND */
| |              /*  logisches ODER */
!                /* logische Negation */
```

z.B.

```
int  x , y;
x = 3;
y = 4;

if  ( ( x < y )   & &   ( x > y ) )
        printf ( " kann niemals sein. \n ");

if  ( ( x < 0 )   ||   ( y > 0 ) )
        printf ( " x > 0   oder   y > 0. \n ");
```

1.7.6 Bitorientierte Operatoren

(nicht verwechseln mit den logischen Operatoren)

```
&          /* UND */
|          /*  inklusives ODER */
^          /*  exklusives ODER */
<<         /*  Linksverschiebung */
>>         /*  Rechtsverschiebung */
~          /*  Bitweise Invertierung: 0->1 und 1->0*/
```

Die Operanden (z.B. x und y) bitorinentierter Operatoren werden in jeder Bitposition nach folgenden Regeln miteinander verknüpft:

x	y	x & y	x | y	x ^ y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Die Operatoren >> und << führen eine Bitverschiebung um n Stellen nach rechts bzw. links durch. In die neu entstanden Stellen werden 0 Werte nachgeschoben.

```
/* Anweisung */          /* Bitmuster */                        /* int-Wert */

int x = 1 ;              /* 0 0 0 0  0 0 0 1                      1     */

x = x << 3 ;             /* 0 0 0 0  0 0 0 1 << 3
                            = 0 0 0 0  1 0 0 0                  8     */

x = x >> 2 ;             /* 0 0 0 0  1 0 0 0 >> 2
                            = 0 0 0 0  0 0 1 0                  2     */

x = x | 5 ;              /* 0 0 0 0  0 0 1 0 | 0 0 0 0  0 1 0 1
                            = 0 0 0 0  0 1 1 1                  7     */

x = x & 3 ;              /* 0 0 0 0  0 1 1 1 & 0 0 0 0  0 0 1 1
                            = 0 0 0 0  0 0 1 1                  3     */

x = x ^ 5 ;              /* 0 0 0 0  0 0 1 1 ^ 0 0 0 0  0 1 0 1
                            = 0 0 0 0  0 1 1 0                  6     */

x = ~ x;                 /* = 1 1 1 1  … 1 1 1 1  1 0 0 1       abhängig von
                                                                 Datentyp int */
```

1.8 Kontrollstrukturen

1.8.1 if – else

```
if ( < Ausdruck > )
        < Anweisung 1 >;            /* Ausdruck  ! = 0 */
else
        < Anweisung 2 >;            /* Ausdruck = = 0 */
```

z.B.

```
if ( z ! = 0 )
        y = x / z ;
else
        printf  ( ” Division durch 0 !\ n ” ) ;
```

Werden mehr als eine Anweisung verwendet, so müssen sie in geschweifte Klammern { ... } gesetzt werden:

```
if ( z ! = 0 ) {
        y = x / z ;
        printf ( ” y = %i \ n ”, y ) ;
}
else
        printf ( ” Division durch 0 !\ n ” ) ;
```

1.8.2 switch – case

Mehrfachauswahl :

```
int i = 2 ;
```

anstelle :

```
if ( i = = 0 )
        printf ( ” i = 0 \ n ” ) ;

if ( i = = 1 )
        printf ( ” i = 1 \ n ” ) ;

if ( i = = 2 )
        printf ( ” i = 2 \ n ” ) ;
```

```
switch ( i ) {
        case 0 :
                printf ( ” i = 0 \ n ” ) ;
                break;   /* break nicht vergessen ! */
        case 1 :
                printf ( ” i = 1 \ n ” ) ;
                break;
        case 2 :
                printf ( ” i = 2 \ n ” ) ;
                break;
}
```

1.8.3 Entscheidungsoperator ? :

< ausdruck 1 > ? < ausdruck 2 > : < ausdruck 3 >

```
if  ( < ausdruck 1 > )
     < ausdruck 2 > ) ;
else
    < ausdruck 3 > ;
```

```
int  x , abs_x ;                    /* Absolutbetrag von x */
x = - 5 ;
abs_x = ( x < 0 ? – x : x ) ;     /* abs_x = 5 */
```

1.9 Schleifen

1.9.1 while – Schleife

```
while  ( < ausdruck > )  {
          < anweisung >

}
```

Schleifenbedingung < ausdruck > wird vor dem Eintritt in die Schleife getestet.
Schleifenbedingung ist erfüllt, wenn < ausdruck > != 0

z.B.

```
while ( a > b )  {
            .                       die Schleife wird so lange abgearbeitet, wie a < b ist
            .
 }
```

```
                  ausdruck
while(!((c == getchar()) == 'a'))
        printf ( " Bitte <a> druecken ! \ n " ) ;
```

Schleife wird so lange durchlaufen, bis das Zeichen a von Standardeingabe eingegeben wurde. Erst dann wird ausdruck 0

```
while ( 1 ) {
     .
     .
}
```

Endlosschleife

1.9.2 do – while – Schleife

```
do {
     < anweisung >;
}
while ( < ausdruck > ) ;
```

Schleifenbedingung < ausdruck > wird erst nach einem Durchlaufen der Schleife überprüft.

1.9.3 for – Schleife (Zählschleife)

```
for  ( < anfangsinitialisierung >; < schleifenbedingung >;  < wiederinitialisierung > ) {
       < anweisung >
}
```

Nach < anfangsinitialisierung > wird Schleife durchlaufen, solange < schleifenbedingung > erfüllt ist. Nach jedem Schleifendurchgang erfolgt < wiederinitalisierung > .

```
                    < anfangsinitialisierung > ;
                    while ( < schleifenbedingung > ) {
                                < anweisung >;
                                < wiederinitialisierung >;
                    }
```

```
for ( i = 0 ; i < 5 ; i ++ )
                printf ( " %i \ n " , i ) ;
```

```
i = 0 ;
while ( i < 5 ) {
        printf ( " %i \ n " , i ) ;
        i ++ ;   /*  i = i + 1  */
}
```

```
/*  Ausgabe:   0
               1
               2
               3
               4        */
```

1.9.4 Schleifenabbruch

1.9.4.1 break

break wird verwendet, um die Abarbeitung einer unmittelbar übergeordneten switch-, while-do-while- bzw. for-Anweisung abzubrechen:

```
                    for ( … ; … ; … ) {
                            …
                            if ( < ausdruck > )
                                    break;
                            printf("Abbruch mit break.\n");
                    }
```

1.9.4.2 goto-Sprunganweisung

Herausspringen aus tieferen Schachtelungsstufen

```
for ( … ; … ; … )  {
        for ( i = 0 ;  i < MAX ; i ++ ) {
                switch ( getchar ( ) ) {
                        case ' c ' :
                                break ;
                        case ' a ' :
                                goto ende ;
                }
                printf("Herausgesprungen durch Drücken <c> - Taste.\n");
        }
}

ende:   printf ( " Herausgesprungen durch Drücken <a> - Taste. \ n " ) ;
```

2. Strukturierte Datentypen

2.1 Felder (Arrays)

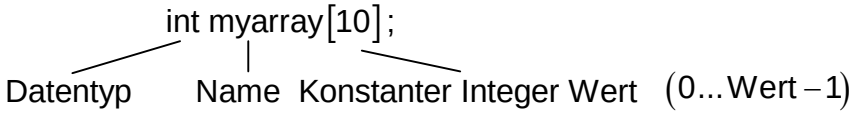
Zusammenfassen von Datenobjekten des gleichen Typs

2.1.1 Statische Arrays

Größe des Arrays steht zur Zeit der Programmentwicklung fest:

```
int i ;
int array [10] ;                                /* array [0] , ... , array [9]
char str [10] ;                                 /* Folge von char = string */
for (i = 0 ; i < 10 ; i ++ ) {
                array [i] = i ;
                printf ( "[ %i ] = %i \ n ", i , array [i]);
}

/* Ausgabe      array [0] = 0
                array [1] = 1
                        .
                        .
                array [9] = 9           */
```

Allokieren des Speicherbereichs für das Integer-Array



Arrays vom Datentyp char werden als Strings bezeichnet:

```
str [0] = ' H ' ;
str [1] = ' a ' ;
str [2] = ' l ' ;
str [3] = ' l ' ;
str [4] = ' o ' ;

printf ( " %s  \ n " , str ) ;
```

ist dasselbe wie

```
str = " Hallo " ;
printf ( " %s \ n " , str ) ;
```

Besonderheit bei Strings : Ende wird mit \0 markiert

```
i=0;
while( str[i] != ' \0') {
        printf("%c", str[i]);
        i++;
}
```

Beispiel: Skalarprodukt zweier Vektoren $\vec{x}, \vec{y}$:

```
#define N 10                        /* N ist Konstanter Wert 10 */
int i ;
float  sp, x [N], y [N] ;           /* x,y 10 dim. reellwertige Vektoren */

sp = 0 ;
i = 0 ;
while ( i < N ) {
        sp + = x [i] * y [i] ;      /* sp + = … bedeutet sp = sp + … */
        i ++ ;                      /* i ++ bedeutet i = i + 1 */
}
printf ( " Skalarprodukt x * y = %d \ n " , sp ) ;
```

Mehrdimensionale Statische Arrays:

```
int  x [2] [3] ;

/* x [0] [0]    x [0] [1]    x [0] [2]
   x [1] [0]    x [1] [1]    x [1] [2]    */
```

2.1.2 Dynamische Arrays

Größe des Arrays steht erst zur Laufzeit des Programms fest:

```
# include  < stdio.h >              /* printf, scanf */
# include < malloc.h >
```

```
int   *p, n, i;                          /* p ist ein Zeiger (Pointer) auf einen noch zu
                                            allokierenden Speicherbereich vom Typ int */
printf (" Dimension des Vektors : " ) ;
scanf ( " %d " , n ) ;                   /* n von Standardeingabe einlesen */

p = ( int *) ( malloc ( n * size of ( int ))) ;     /* Speicher allokieren */

for ( i = 0 ; i < n ; i ++)
        p [i] = i+i;

free (p) ;                                          /* Speicher freigeben */
```

Mehrdimensionale dynamische Arrays werden nach demselben Prinzip allokiert und freigegeben:

2 dim. dynamische Arrays in C: (z.B. zum Laden einer Bilddatei)

```
double **p, n, m, i , j ;                /* p ist Zeiger auf Zeiger für einen noch zu allokierenden
                                            Speicherbereich vom Typ double */
printf ( " Anzahl Zeilen der Matrix : \ n " ) ;
scanf ( " %d " , n ) ;
printf ( " Anzahl Spalten der Matrix : \ n " ) ;
scanf ( " %d " , m ) ;

p = ( double ** ) ( malloc ( n * size of ( double * ))) ;
/* Speicher für Feld von n Zeigern vom Typ double
    p [0] , … , p [ n-1] allokieren

            double *   p [0]
                  .
                  .
            double *   p [n-1]                    */

for ( i = 0 ; i < n ; i ++ )
        p [i] = ( double * ) ( malloc ( m * sizeof ( double )));

/* Speicher für  1 dim. Arrays für p [0] , … , p [n-1]
   allokieren:       p [0] [0]   , … ,  p [0] [ m - 1]
                                  .
                                  .
                     p [n-1] [0] , … , p[n-1] [m – 1]          */

/* Matrix kann verwendet werden*/
for ( i = 0 ; i < n ; i ++ )
        for ( j = 0 ; j < m ; j ++ )
                p[i][j] = i + j;

/* Speicher der Matrix freigeben */
for (i = 0 ; i < n ; i ++)
        free ( p [i] ) ;

free ( p ) ;
```

2.2 Strukturen

Zusammenfassung von Objekten unterschiedlichen Typs :

2.2.1 Struct

```
struct   adresse        {               /* Deklaration des Strukturtyps adresse */

         int      plz ;
         char     ort[10] ;
         char     str[20] ;
         int      nummer ;
};

struct   adresse    meine_adresse ;     /* meine_adresse Instanz (Variable) des Typs adresse */
/* Zugriff auf Strukturkomponenten mit . */

meine_adresse.plz = 14542 ;
meine_adresse.ort  = "Werder";
```

2.2.2 Union

```
/* Deklaration der Union zahlen */

union  zahlen  {
         int i ;
         float f ;
         long l ;
};

/* Instanz oder Variable meine_zahlen vom Typ zahlen deklarieren*/

union   zahlen meine_zahlen ;

/* Zugriff auf Union Komponenten mit . analog zu Struct*/

meine_zahlen . i = 1 ;
meine_zahlen . f = 3.14 ;
```

2.3 Selbst definierte Datentypen

```
typedef         int             MEININT ;  /* neuer Datentyp MEININT */
typedef         struct {
                short  jahr ;
                char   monat [4] ;
                short  Tag ;
}       MEINDATUM ;                     /* neuer Datentyp MEINDATUM */
```

```
void main (void)
{
        MEININT      i ;                    /* Variable i vom Typ MEININT */
        MEINDATUM    geb_datum ;            /* Variable geb_datum vom Typ
                                               MEINDATUM */

        geb_datum . jahr = 1980 ;

        .
        .

}
```

2. Zeiger (Pointer)

Zeiger enthalten die Adresse einer Variablen eines bestimmten Datentyps (einfacher, zusammengesetzter (Arrays, Strukturen), selbst definierter).

2.1 Deklaration

```
/* Deklaration eines Zeigers vom Typ int */

int *p ;
```

allgemein :

< Datentyp > * < Bezeichner > ;

z.B.

```
char    *cp ;              /* Pointer vom Typ char */
int     x, *ip ;           /* x int-Variable, ip int-Pointer */
float   **fp ;             /* Pointer auf Pointer vom Typ float */
                           /* z.B. bei dynamischen 2 dim. Arrays */

struct adresse *dat ;      /* dat ist Pointer vom Typ adresse */
                           /* adresse ist Struktur */

float *fp [10];            /* Feld von Pointern fp[0] … fp[9] vom Typ float */
char *argv[]               /* Feld von Pointern auf char */
```

2.2 Zeigeroperatoren

&	Adreßoperator
*	Wertzuweisungsoperator

z.B.

```
/*Deklaration*/
int     *p ;                /* Zeiger vom Typ int */
int     i, j ;              /* Variablen vom Typ int */

i = 5 ;

/* Anweisung */
p = &i ;                    /* Zeiger p erhält Adresse von i */
j = *p ;                    /* j erhält den Inhalt der Adresse auf die p zeigt */

printf ( " j = %d \ n " , j ) ;     /* j = 5 */
```

2.3 Zeigerarithmetik

Zu Pointern können ganzzahlige Werte addiert oder subtrahiert werden.
Die Adresse der Pointers wird um die Anzahl Bytes verändert, die ein Objekt des Typ des Pointers benötigt:

```
int array [100], *p, i ;
p = &array[0] ;
p = p + 1 ;                 /* bzw. p ++ ; oder p + = 1 ; */

/* Die Adresse von p wird um soviel Byte hochgesetzt, wie eine Variable vom Typ int
   benötigt (2 Byte). Damit zeigt der Pointer automatisch auf das nächste Feldelement
   array [1] */

p = & array [50] ;
i = 10 ;
p = p + i ;                 /* p + = i */

/* Adresse von p wird um soviel Byte hochgesetzt, wie i * (eine Variable vom Typ int
   benötigt).
   Damit zeigt p automatisch auf array [60] */

p = p – i ;                 /* p - = i ; */

/* p zeigt nun auf array [50] */
```

Mit Pointern können Vergleichsoperationen = = und ! = durchgeführt werden:

```
if ( p1 = = p2 )
        printf ( " p1 zeigt auf selbe Adresse wie p2. \ n " ) ;

if ( p1 ! = p2 )
        printf ( " p1 zeigt auf andere Adresse als p2. \ n " );
```

2.4 Zusammenhang zwischen Zeigern (Pointern) und Feldern (Arrays)

In C stehen Felder in enger Beziehung zu Zeigern.

z.B.

```
int array [100] , *p ;

p = & array [0] ;                    /* p ist Zeiger auf array [0] */
```

Falls p auf irgendein Feldelement von array zeigt, so bezieht sich der Ausdruck

```
p + 1;          /* auf das nächste Feldelement */
p – 1;          /* auf das vorhergehende Feldelement */
```

allgemein:

```
p + i ;         /* dasjenige Feldelemente, das sich i Elemente nach dem durch p
                   adressierten befindet */

p – i ;         /* dasjenige Feldelement, das sich i Elemente vor dem durch p
                    adressierten befindet */

/* Zeigt p auf array [0]  */
   p = &array[0] ;

/* so ist array[i]      dasselbe wie *(p + i)        und
       &array[i]       dasselbe wie   (p + i)              */
```

In C wird ein statisches Feld vom Compiler in einen Zeiger auf das erste Element des Feldes transformiert. Demzufolge ist jedes statische Array ein Zeiger und

als Feld:		als Zeiger:
&array[0]	dasselbe wie	array
array[i]	dasselbe wie	*(array + i)
&array[i]	dasselbe wie	(array + i).

Unterschied statisches Feld und Zeiger:

Beim statischen Array ist der Speicherplatz für alle Arrayelemente bereits allokiert.

2.5 Zeiger auf Strukturen

```
struct  adresse  {
        int plz ;
        char ort [10] ;
        char str [10] ;
        int nummer ;
};

struct  adresse        *p ;          /* Pointer vom Typ adresse */
```

```
/* Zugriff auf Strukturkomponenten */

        entweder

(*p).plz = 14542;
(*p).ort  = "Werder";

        oder

p->plz = 14542;
p->ort = "Werder";
```

3. Funktionen

Funktionen dienen der Mehrfachausnutzung von Programmcode und führen zu einer Strukturierung und Erhöhung der Übersichtlichkeit des Programms.

Sie haben folgenden Aufbau:

```
< rückgabetyp >        < funktionsnahme >   (< übergabevariablen >) {
                       < definitionen > ; /* z.B. lokale Konstanten, lokale Variablen*/
                                          /*  nur innerhalb der Funktion gültig          */
                       < anweisungen > ;
}

/* Definition der Funktion kreis ( ) */

float    kreis    ( float radius ) {
         float    pi = 3.1415 ;
         return ( pi * radius * radius ) ;
}

/* Aufruf der Funktion kreis ( ) */

float radius, flaeche;
radius = 2.0 ;
flaeche = kreis ( radius ) ;
printf ( " Flaeche des Kreises : % f \n " , flaeche ) ;
```

- Die Reihenfolge der Übergabevariablen in der Definition muss mit der Reihenfolge beim Aufruf übereinstimmen.
- In C wird nicht zwischen Funktion und Prozedur unterschieden. Hat eine Funktion keinen Rückgabetyp, ist sie vom Typ void:

```
void warten ( void ) {
        getchar ( ) ;     /* Warten auf Tastatureingabe */
}
```

- Die Abarbeitung einer Funktion endet bei der Anweisung

return < rückgabewert >;

Funktionen die keinen Funktionswert zurückgeben, beenden ihre Arbeit durch die Anweisung

return;

bzw. beim Erreichen der schließenden geschweiften Klammer } des Funktionsblocks.

- Übergabevariablen und Rückgabetyp können Grunddatentypen (int, float, double), strukturierte Datentypen (Arrays, struct) oder selbst definierte Datentypen sein.

```
void     print_address ( struct adresse ){

                printf ( " PLZ : %d \n " , adresse.plz ) ;
                printf ( " Wohnort : %s \n " , adresse.ort ) ;
                printf ( " Strasse : %s \n " , adresse.str ) ;
                printf ( " Nummer : %d \n " , adresse.nummer ) ;

                return ;        /* Kann bei Rückgabewert void auch */
                                /* weggelassen werden */
}
```

Die innerhalb einer Funktion definierten Konstanten und Variablen sind nur innerhalb des Funktionskörpers (Unterprogramm) gültig.

3.1 Werteübergabe Call by Value

```
void tausche (int x1 , int x2 ) {

                int x ;
                x = x1 ;
                x1 = x2 ;
                x2 = x ;
                printf ( " x1 = %i und x2 = %i \n ", x1 , x2 ) ;
}
void main( void ) {
int       x_1 , x_2 ;
x_1 = 1 ;
x_2 = 2 ;

tauschen ( x_1 , x_2 ) ;
printf ( " x_1 = %i und x_2 = %i \n ", x_1 , x_2 ) ;

/* x1 = 2 und x2 = 1 */
/* x_1 = 1 und x_2 = 2 */
```

Bei einer Übergabe Call by Value wird lediglich eine Kopie der Variablen übergeben. Änderungen innerhalb der Funktion haben keinen Einfluss auf die übergebenen Variablen.

3.2 Werteübergabe Call by Reference

```
void tauschen ( int  *x1 , int *x2 ) { /* x1 und x2 Zeiger vom Typ int */
        int  x ;
        x = *x1 ;                       /*  * Wert des Zeigers */
        *x1 = *x2;
        *x2 = x ;
        printf ( " x1 = %i und x2 = %i \ n ", *x1 , *x2 ) ;

}

void main (void) {

int x_1, x_2 ;

x_1 = 1 ;
x_2 = 2 ;

tauschen (&x_1, &x_2) ;    /* Adressen von x_1 und x_2 */
printf (" x_1 = %i und x_2 = %i \ n ", x_1, x_2) ;

/* x1 = 2 und x2 = 1 */
/* x_1 = 2 und x_2 = 1 */
```

Bei einer Übergabe Call by Reference wird ein Zeiger, der auf die Adresse der Übergabevariablen zeigt, übergeben.
Damit wird direkt auf die Übergabevariablen zugegriffen.
Die Werte der Übergebevariablen können innerhalb der Funktion geändert werden.

3.3 Häufig benötigte Funktionen

3.3.1 Standard Ein- und Ausgabe

```
# include       < stdio.h >
```

Zeichenweise Ein- und Ausgabe

putchar (char)	Ausgabe eines Zeichens auf Standardausgabe (Bildschirm)
char getchar (void)	Einlesen eines Zeichens von der Standardeingabe (Tastatur)
char getkey (void)	Einlesen eines Zeichens von der Standardeingabe (Tastatur)

Zeilenweise Ein- und Ausgabe

puts(char*)	Zeichenfolge (String) ausgeben
gets(char*)	Zeichenfolge (String) einlesen

z.B.

```
# include <stdio.h>
void main (void) {
char str[80], c ;
int i = 0 ;
while ( (c = getchar() ) ! = ' a ' )
        str [ i++ ] = c ;
/* Zeichen c von Tastatur einlesen bis Abbruch mit Taste <a> */

str [ i ] = ' \0 ' ;
puts ( str ) ;  /* String ausgeben*/

/* dasselbe wie */
gets(str);
puts(str);

}
```

Formatierte Ein- und Ausgabe

int printf (char *format , arg1 , arg2 , …)

Formatierte Ausgabe von arg1 und arg2 auf Standardausgabe

int scanf (char * format , arg1 , arg2 , …)

Formatierte Eingabe von arg1 und arg2 von Standardeingabe

Wichtig :
Die Argumente von scanf() arg1 und arg2 müssen Zeiger sein (Call by Reference)

Beispiel :
```
int i ;
scanf ( " %d " , & i ) ;

scanf ( " %d " , i ) ;     Falsch
```

char *format ist eine Zeichenfolge, in der festgelegt wird, in welchem Format die Argumente arg1, arg2 ausgegeben bzw. eingelesen werden sollen.

Formatbuchstaben für printf

Formatbuchstabe	Argumenttyp	
%d ,% i	int	Integer Zahl
%c	char	ASCII Zeichen
%s	char*	String (Zeiger auf Feld von char)
%f	float	Reelle Zahl

```
int i = 10 ;
char c = ' a ' ;
char *s = " Hallo " ;
float f = 3.14 ;

printf ( " i = %d und c : %c und s : %s und f = % f \n ", i , c , s , f);  /* Call by Value */

/* i = 10 und c : a und s : Hallo und f = 3.14 */
```

Formatbuchstaben für scanf

Buchstabe	Argumenttyp	
%d, %i	int*	Integer Zahl
%c	char *	ASCII Zeichen
%s	char *	String
%f	float *	Reelle Zahl

```
scanf ( " %d " , &i );       /* Call by Reference */
scanf ( " %c " , &c );       /* Call by Reference */
scanf ( " %s " , s );        /* Call by Reference: s ist dasselbe wie &s[0] */
scanf ( " % f " , &f );      /* Call by Reference */
```

3.3.2 Dateizugriff

Oftmals sollen Programme Daten bearbeiten, die in Dateien aufbewahrt und nach der Bearbeitung wieder abgespeichert werden. Zu diesem Zweck muss ein Programm auf eine Datei zum Lesen und Schreiben zugreifen können.
Vor ihrer Berarbeitung muss die Datei geöffnet werden. Dies kann mit der Funktion fopen

```
FILE*    fopen ( char *datei , char *type )
```

realisiert werden.

z.B.

```
FILE *          pdatei ;                    /* Pointer auf FILE */
char            datei[20] = "bild1.pgm" ;
pdatei = fopen( datei, " r " ) ;            /* zum Lesen geöffnet */
```


read

```
pdatei = fopen ( datei , " w " ) ;          /* oder zum Schreiben geöffnet
                                               wobei ein bereits existierendes File gelösch
                                               wird. Andernfalls wird das File erzeugt */
```


write

Für das Lesen und Schreiben in eine Datei stehen vergleichbare Funktionen wie für die Standard Ein- und Ausgabe über die Tastatur zur Verfügung:

```
FILE *          pdatei ;
char            datei[20] = " bild1.pgm " ;
pdatei = fopen ( datei , " r " ) ;
```

Zeichenweise aus Datei lesen

```
char c ;
c = fgetc( pdatei ) ;                   /* Lesen des 1.Zeichens aus Datei bild1.pgm */
```

Zeilenweise aus Datei lesen

```
char sbuf [100] ;                       /* string (Buffer) mit Platz für 100 char Zeichen */

fgets ( sbuf , sizeof (sbuf ) , pdatei ) ;      /* liest entweder die folgenden (ab Zeichen 2)
                                                   99 Zeichen bzw. bis Newline ein und schließt
                                                   Zeichenkette mit \0 ab (sbuf [99] = = ' \ 0 ' ) */
```

Formatiertes Lesen aus Datei

```
int i ;
fscanf ( pdatei , " %d " , &i ) ;
/* das folgende Zeichen wird gelesen und als Integer Zahl interpretiert */

fclose ( pdatei ) ;

FILE *          pdatei ;
char            datei[20] = " bild1.pgm "

pdatei = fopen ( datei , " w " ) ;      /* Datei bild1.pgm wird erzeugt und zum Schreiben
                                           geöffnet */
```

Zeichenweise Schreiben in Datei

```
char c = 'A' ;
fputc ( c , pdatei ) ;                  /* Als 1.Zeichen wird A in bild1.pgm geschrieben */
```

Zeilenweise schreiben in Datei

```
char sbuf [100]
sbuf = "Hallo";
fputs ( sbuf , pdatei ) ;               /* Als 2. – 6.Zeichen wird Hallo in Datei bild1.pgm
                                           geschrieben */
```

Formatiertes Schreiben in Datei

```
int i , j;
i = 100;
j = 100;
fprintf ( pdatei , " \ n%i %i \ n " , i , j ) ;
                                        /* in neue Zeile wird Wert von i (100) und Wert von
                                           j (100) geschrieben */

fclose( pdatei ) ;
```

bild1.pgm

```
A Hallo
100  100
```

3.3.3 Weitere Funktionen

Nach ANSI-C gehören zu einer C-Implementierung eine Vielzahl weiterer Funktionen, die in den Standardbibliotheken zur Verfügung gestellt werden:

Characterbehandlung # include < ctype.h >

Zeichenfolgenfunktionen # include < string.h >

strcpy (char *s1 , char *s2)
Kopiert s2 in s1

strcat (char *s1 , char *s2)
Kopiert s2 aus Ende von s1

Formatierte Ein- Ausgabe #include < stdio.h>

int printf (char* , …)
int scanf (char* , …)

Mathematische Funktionen # < math.h >

```
double      sin     ( double x )            /* sin (x) */
double      exp     ( double x )            /* e^x     */
double      pow     ( double x, double y )  /* x^y     */
double      sqrt    ( double x )            /* √x      */
double      fabs    ( double x )            /* |x|     */
```

Speicherverwaltungsfunktionen # include < malloc.h >

void * malloc (size_t size)

Allokiert einen Speicherbereich der Größe size und liefert Zeiger auf diesen

void free(void *)

Speicher wieder freigeben

Hauptfunktion main()

```
/* test.c*/
# include<stdio. >
void main ( int argc , char  *argv[] )
{       /*Ausgabe aller Argumente der Kommandozeile*/
        int i ;
        for ( i = 0 ; i < argc ; i++ )
                printf ( "argv[%i]: %s \n " , i, argv [i] );
}
```

/* argc Anzahl der in der Konsole übergebenen Zeichenketten (Strings)

/* argv[i] i. Zeichenkette

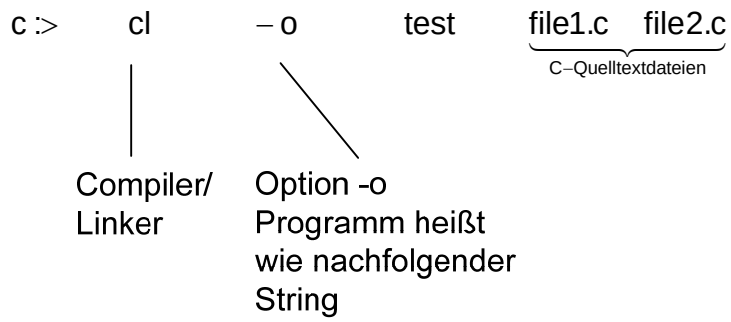
```
c:\ > cl test.c
c:\ > test.exe   hallo   arg2   *.*
      argv[0]: test
      argv[1]: hallo
      argv[2]: arg2
      argv[3]: *.*
```

4. Projektverwaltung - mehrere Quelltextdateien

Um die Strukturiertheit von Programmen zu erhöhen, kann der Sourcecode auf mehrere Dateien aufgeteilt werden:

```
/* file1.c */
#include"file2.h"
void main ( int argc , char * argv[])
{
        /* Aufruf von f aus file2.c */
        f ( ) ;
}

/* file2.c */
#include< stdio.h >

/* Definition der Funktion f */
void f ( ) {
        printf ( " Hallo Welt ! \n " );
}


/* file2.h */
/* Deklaration der Funktion f(), damit file1.c kompiliert werden kann*/
void f (void) ;
```

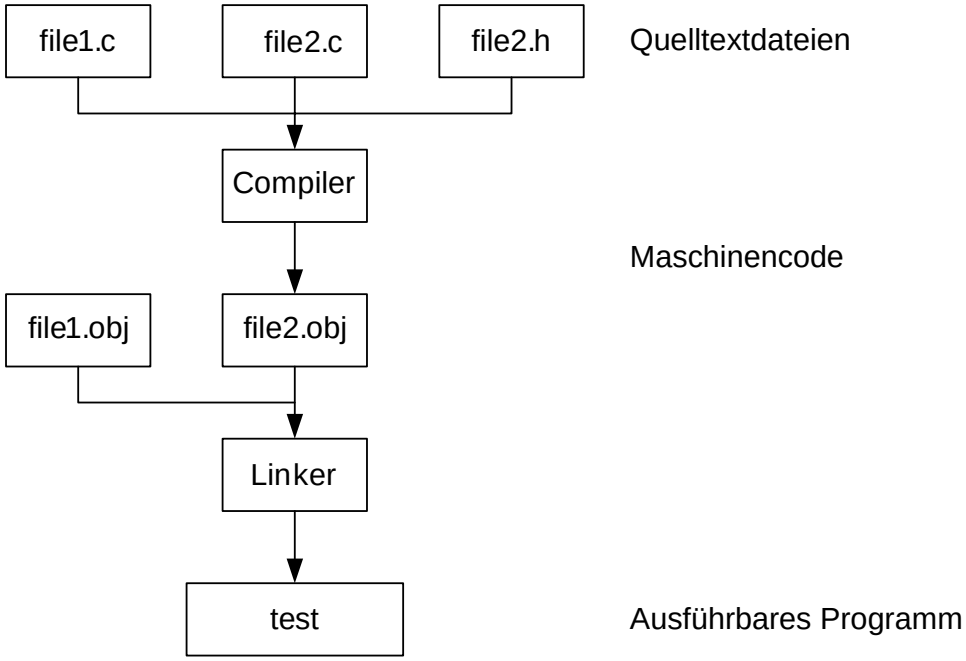
Kompilieren / Linken in der Console:



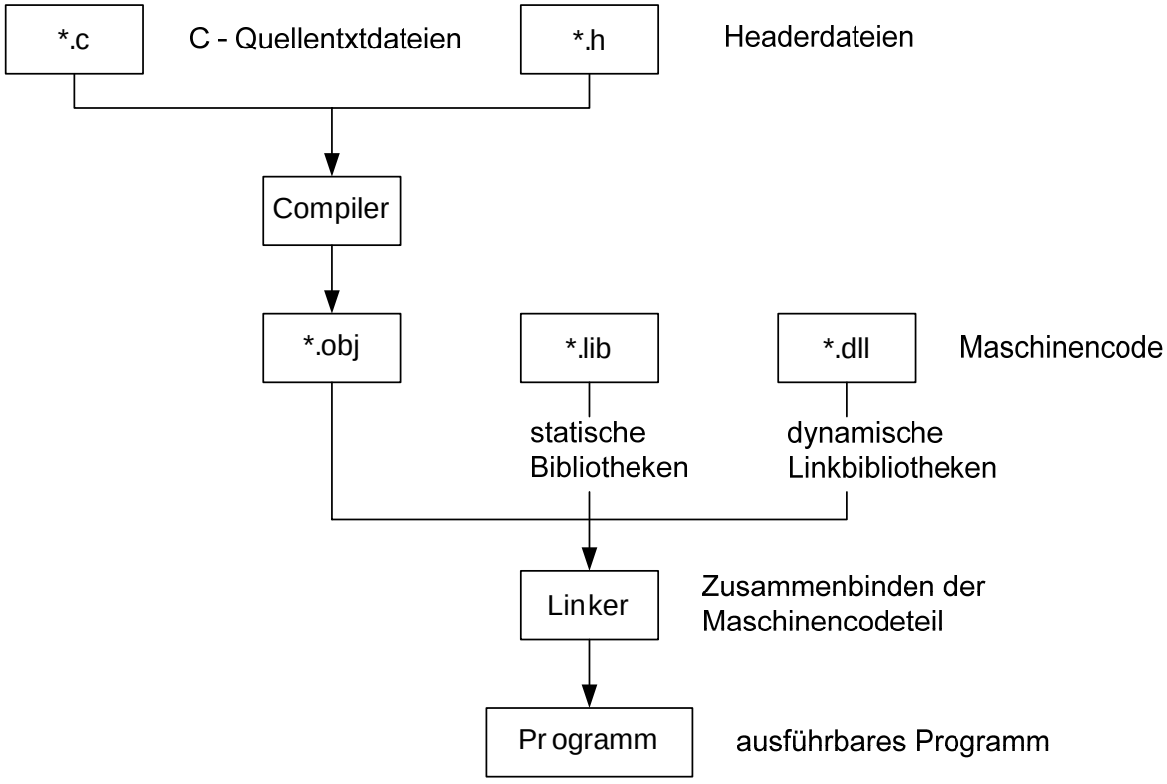
Aufruf des ausführbaren Programms in der Console:

```
c:\ > test.exe          oder
c:\ > test

     Hallo Welt !

c:\ >
```

Ablauf der Programmerstellung:



Allgemein:



4.1 Verwaltung größerer Projekte mit Hilfe eines Makefiles

nmake bewirkt, dass nur die Teile des Programms kompiliert werden, die seit der letzten Kompilierung geändert wurden:

nmake liest Dateien, so genannte Makefiles

Beispiel für ein Makefile:

```
test.exe:    file1.obj   file2.obj
             cl  -o  test.exe   file1.obj   file2.obj

file1.obj:   file1.cpp
             cl  -o   file1.c

file2.obj:   file2.cpp
             cl  -o   file2.c
```

Eingebaute Regeln

Links steht die abhängige Datei
- Rechts von einem Doppelpunkt und Leerzeichen getrennt die Dateien, von denen sie abhängt.
- in der darauf folgenden Zeile steht eine Kommandozeile

⇒ Ist der Zeitstempel der abhängigen Datei älter als einer der Dateien, von denen sie abhängt, wird die Kommandozeile ausgeführt

Aufruf (es muss ein Makefile vorhanden sein)

```
c:\ > nmake
```

Vorteil:
- es werden nur die Dateien neu kompiliert, die sich verändert haben
- zur Kompilierung des gesamten Projektes reicht Eingabe nmake

4.2 Programmerstellung mit einer Integrierten Entwicklungsumgebung (IDE)

1994:
- Makefile: verwaltet Projekt (mehrere Quelltextdateien)
- Texteditor, Compiler, Linker, Debugger
  getrennte Einheiten

IDE:
- Projektverwaltung, Texteditor, Compiler, Linker, Debugger unter einer gemeinsamen Oberfläche
- Sourcecode für häufig vorkommende Programmieraufgaben wird von der IDE automatisch generiert.
- graphische Programmierung: Zusammenklicken von graphischen IDE-Elementen -> IDE generiert zugehörigen Sourcecode automatisch (z.B. Erstellung graphischer Benutzeroberflächen GUI)

5. Mikrocontroller-Programmierung in C

Im Gegensatz zum PC geht es bei der Controllerprogrammierung meist weniger um die Ein- oder Ausgabe bzw. Bearbeitung von Texten. Im Vordergrund steht hier üblicherweise die Erfassung, Manipulation und Ausgabe irgendwelcher Messwerte oder Ansteuersignale, die Abfrage von Tasten oder Schaltern, die Verwirklichung von Zeitschleifen oder Verzögerungen, die Erzeugung bestimmter Signalformen oder Frequenzen, die Bereitstellung von Bitmustern usw. …

5.1 Umgang mit der IDE Keil µ-Vision

Siehe hierzu die gesonderte Anleitung zum Umgang mit der KEIL-Software. Sie enthält auch alle erforderlichen Voreinstellungen und Handgriffe, um das Programm zum Leben zu erwecken.

5.2 Programmerstellung für den µC DS80C320 mit der IDE Keil µ-Vision

Am Anfang brauchen wir grundsätzlich **immer** mindestens folgenden Vorspann:

```
#include <reg320.h>    /* Registerdefinitionen für den µC  DS80C320              */
#include “init.h“      /* Definition der Ports und Einstellungen RS232, Timer   */
```

Beispiel 1: Ausgabe über die serielle Schnittstelle

```
#include <reg320.h>
#include <stdio.h>              /*  printf(), scanf() */
#include "init.h"

void main (void) {

    init (); /* Einstellungen des Controllers vornehmen*/

    /*-------------------------------------------------
    Note that an embedded program never exits (because
    there is no operating system to return to).  It
    must loop and execute forever...
    -------------------------------------------------*/

    while (1) {

            printf("Hello World\n");

    }
}
```

Beispiel 2: Ausgabe und Einlesen über die serielle Schnittstelle und Ausgabe auf Port1

```
#include <reg320.h>
#include <stdio.h>
#include "init.h"
```

```
void wait(void){}

void blinky (void){
    int i, j;
    P1=1;
    for (i=0; i<8; i++){
            for (j = 0; j < 30000; j++)
                    wait ();
            P1 = P1<<1; /*P1=P1*2;*/
    }
}

void main (void) {

    char name[20];
    int i;
    float f;

    init ();   /* Einstellungen des Controllers vornehmen*/

    printf(" 1. Demonstration Ein- Ausgabe");
    printf(" über serielle Schnittstelle RS232\n");
    printf("   mit printf() und scanf()\n");
    printf("2. Demonstration Ausgabe auf Port1 (Blinklicht)\n\n1.\n\n");

    printf("Bitte einen String eingeben: ");
    scanf("%s",name);
    printf("String ist: %s\n\n",name);

    printf("Bitte eine ganze Zahl eingeben: ");
    scanf("%i",&i);
    printf("Integer ist: %i\n\n",i);

    printf("Bitte eine reelle Zahl eingeben: ");
    scanf("%f",&f);
    printf("Float ist: %f\n\n2.\n\n",f);

    printf("Prima\n -> Starte Blinklicht auf Port1 ...");

    /*-----------------------------------------------
    Note that an embedded program never exits (because
    there is no operating system to return to).  It
    must loop and execute forever...
    -----------------------------------------------*/

    while (1) {

            blinky();
            printf(".");

    }
}
```

Beispiel 3: Einlesen und Ausgabe der einzelnen Bits von Port1 und Port3

```
#include <reg320.h>
#include <stdio.h>
#include "init.h"

void main (void) {

    init ();   /* Einstellungen des Controllers vornehmen*/

    printf(" Praktikumsversuch 1: Ein- Ausgabe über Port1 und Port3:\n\n");

    /*---------------------------------------------
    Note that an embedded program never exits (because
    there is no operating system to return to).  It
    must loop and execute forever...
    ---------------------------------------------*/

    while (1) {

            P1_0 = P3_2;
            P1_1 = P3_3;
            P1_2 = P3_4;
            P1_3 = P3_5;
    }
}
```

Definition der Ports und Einstellungen RS232, Timer

```
/*************************************************************/
/*                          init.c                           */
/*************************************************************/
#include <reg320.h>

void init(){
    SCON0 = 0x50;        /* SCON0: mode 1, 8-bit UART, enable rcvr   */
    TMOD |= 0x20;        /* TMOD: timer 1, mode 2, 8-bit reload      */
    TH1   = 0xF7;        /* TH1:  reload value for 9600 baud @ 33MHz */
    TR1   = 1;           /* TR1:  timer 1 run                        */
    TI    = 1;           /* TI:   set TI to send first char of UART  */
}

/*************************************************************/
/*                          init.h                           */
/*************************************************************/
sbit P0_0 = P0^0;
sbit P0_1 = P0^1;
sbit P0_2 = P0^2;
sbit P0_3 = P0^3;
```

```
sbit P0_4 = P0^4;
sbit P0_5 = P0^5;
sbit P0_6 = P0^6;
sbit P0_7 = P0^7;

sbit P1_0 = P1^0;
sbit P1_1 = P1^1;
sbit P1_2 = P1^2;
sbit P1_3 = P1^3;
sbit P1_4 = P1^4;
sbit P1_5 = P1^5;
sbit P1_6 = P1^6;
sbit P1_7 = P1^7;

sbit P2_0 = P2^0;
sbit P2_1 = P2^1;
sbit P2_2 = P2^2;
sbit P2_3 = P2^3;
sbit P2_4 = P2^4;
sbit P2_5 = P2^5;
sbit P2_6 = P2^6;
sbit P2_7 = P2^7;

sbit P3_0 = P3^0;
sbit P3_1 = P3^1;
sbit P3_2 = P3^2;
sbit P3_3 = P3^3;
sbit P3_4 = P3^4;
sbit P3_5 = P3^5;
sbit P3_6 = P3^6;
sbit P3_7 = P3^7;

void init(void);
```