Prinzipieller Grundaufbau eines einfachen C-Programmes

C unterscheidet zwischen Groß- und Kleinschreibung!

Siehe zu den folgenden Erklärungen auch das Programm am Ende der nächsten Seite.

Am Anfang **aller** C-Programme werden die sogenannten "**Header-Dateien**" eingebunden. Sie sorgen dafür, dass die nötigen Bibliotheksfunktionen zur Verfügung stehen und werden mit

```
#include <****.h>
```

aufgerufen.
In unseren Programmen werden immer die Adressen der Standardregister benötigt:

```
#include <at89c51ed2.h>   // Registerdefinitionen fuer Atmel AT89C51ED2
```

Zur LCD-Ausgabe benötigen wird Standardfunktionen und LCD-Treiber:

```
#include <stdio.h>        // Standard-Ein-Ausgabe-Funktionen
#include <lcd.h>          // LCD-Ausgabe-Funktionen
```

Anschließend werden eigene Registerdefinitionen, Abkürzungen oder Kurzschreibweisen angegeben.

```
sfr at   P1  Eingabe;          // Eingabeport ist P1 mit 8 Schaltern
sfr at   P2  Ausgabe;          // Ausgabeport ist P2 mit 8 LEDs
sbit at  P3_2  Taster;         // Taster an P3.2
```

oder auch

```
sfr at   0x90  Eingabe;
```

sfr heißt spezielles Funktions- Register, hinter **at** steht die Adresse,
also 0x90 (korrekte Schreibweise einer Hex-Zahl in C)
oder P1 (wobei der Compiler die Adresse von P1 aus der Datei at89c51ed2.h kennt.),
anschließend folgt der im Programm verwendete Name, z.B. Eingabe.

sbit heißt spezial Bit , hinter **at** steht die Adresse des Bit,
dahinter steht der im Programm verwendete Name, hier Taster.

Unter den Definitionen folgen die eigentlichen Programmanweisungen. Sie sind **immer** in sogenannten **"Funktionen**" enthalten.
Die nötige "**Start- und Grundfunktion**" (= sozusagen das **"Hauptprogramm**") trägt den Namen main ()

Da der Funktion main kein Wert übergeben wird, steht in den Klammen void.
Da die Funktion auch keinen Wert liefert, steht void auch davor.
Die korrekte Deklaration lautet also:

```
void main (void)
```

Die Funktionsdeklaration erfolgt **ohne Strichpunkt.**

In der nächsten Zeile steht nur **eine geöffnete, geschweifte Klammer**. Sie eröffnet die Sammlung von Anweisungen, die durchgeführt werden sollen.

Jede Anweisung innerhalb der geschweiften Klammer muss immer durch einen Strichpunkt beendet werden!

Hinter den Programmschritten wird (wieder in einer neuen, getrennten Zeile) die **geschweifte Klammer geschlossen** und damit die Funktion beendet.

Ein Mikrocontroller steuert normalerweise ein Gerät. Diese Steuerung läuft solange das Gerät eingeschaltet ist.
Daher besteht das Hauptprogramm aus einer Endlosschleife, die in C mit der **while**-Bedingung programmiert werden kann.
In der Klammer hinter **while** steht eine Bedingung, z.B. **while** (taster ==1);, was bedeutet: führe diesen Befehl oder diese Befehlsfolge solange aus, wie der Taster eine logische Eins liefert.
Die Bedingung **while** (1) ist immer erfüllt, die Eins steht für die logische 1, also wahr.
Zwischen den geschweiften Klammern hinter **while** (1) { .... } stehen die Anweisungen der Hauptprogrammschleife.
„Warten wenn Taster gedrückt“ kann mit **while** (taster ==1); oder **while** (taster ==1) { } realisiert werden, die Schliefe besteht aus einem Befehl.

Kommentare am Ende einer Zeile beginnen mit //
Nach /* ist alles Kommentar bis zum nächsten */ !!!!
Hiermit kann man wunderschön ganze Programmblöcke in einen Kommentar verwandeln und Fehler suchen.

Unser erstes Mikrocontroller-Programm:

```
/*-------------------------------------------------------------------+
|       File:       Aufgabe_01.c                                    |
|       Autor:      Bubbers                                         |
|       Klasse:     TGJ1                                            |
|       Controller: AT89C51ED2 (Atmel)                              |
|       Datum:      01.03.2006    Version 1.0                       |
+-------------------------------------------------------------------+
|  Beschreibung:                                                    |
|        8 Schalter an P1 an die 8 LEDs von P2 ausgeben             |
+-------------------------------------------------------------------*/

#include <stdio.h>            // Standard-Ein-Ausgabe-Funktionen
#include <at89c51ed2.h>       // Registerdefinitionen fuer Atmel AT89C51ED2

sfr  at      P1 Eingabe;      // Eingabeport ist P1 mit 8 Schaltern
sfr  at      P2 Ausgabe;      // Ausgabeport ist P2 mit 8 LEDs
sbit at      P3_2 Taster;     // Taster an P3.2

//------------------ Hauptprogramm -------------------------------------
void main (void)
{
      while(1)                        // Hauptprogramm ist eine Endlosschleife
      {
            Ausgabe = ~Eingabe;       // 8 Schalter invertiert an 8 LEDs
            while (Taster ==1);       // warten wenn Taster gedrückt
      }                               // Ende der Endlosschleife
}
```

Datentypen

Erklärung	Datentyp	Wertebereich	Belegter Speicherplatz
* Einzelnes Bit im Bitadressierbaren Bereich	**bit**	0 oder 1	1 Bit
* bitadressierbares Bit eines speziellen Funktionsregisters	**sbit**	0 oder 1	1 Bit
* 1 Byte eines speziellen Funktionsregisters	**sfr**	0 bis 255	1 Byte
Positive ganze Zahlen	**unsigned char**	0 bis 255	1 Byte
Ganze Zahlen	**(signed) char**	-128 bis +127	1 Byte
Positive ganze Zahlen	**unsigned short**	0 bis 65535	2 Byte
Positive ganze Zahlen	**unsigned int**	0 bis 65535 (beim 8-Bit-Controller)	2 Byte (beim 8-Bit-Controller)
Ganze Zahlen	**(signed) short**	-32768 bis +32767	2 Byte
Ganze Zahlen	**(signed) int**	-32768 bis +32767 (beim 8-Bit-Controller)	2 Byte (beim 8-Bit-Controller)
Positive ganze Zahlen	**unsigned long**	0 bis 4 294 967 295	4 Byte
Ganze Zahlen	**(signed) long**	-2 147 483 648 bis +2 147 483 647	4 Byte
Gleitkommazahlen mit 6 Nachkommastellen	**float**	±1.175494E-38 bis ±3.402823E+38	4 Byte
Gleitkommazahlen mit 16 Nachkommastellen	**double**	10^{-308} bis 10^{+308}	8 Byte

* Datentyp gibt es nur bei Mikrocontrollern

Bei unseren 8-Bit-Mikrocontrollern sollte möglichst häufig der **char**-Typ (character) verwendet werden, für größere Zahlen auch **int**, bei der Messwerterfassung mit dem Analog-Digital-Umsetzer zur Kalibrierung des Messbereichs (z.B. 0V bis 5V) auch **float**.
Char - Operationen sind sehr schnell, da häufig nur 1 oder 2-Byte-Befehle.
Int – Operationen müssen vom Compiler auf mehrere 8-Bit-Operationen zurückgeführt werden und sind daher deutlich langsamer.
Float - Operationen dauern sehr lange und belegen sehr viel Speicherplatz.

Im Normalfall erfolgt die **Deklaration** einer Variablen **in der Funktion**.
Dann ist sie auch nur in der Funktion gültig und belegt nur dort den Speicherplatz. Dieser wertvolle Speicherplatz kann außerhalb der Funktion anderweitig verwendet werden. Dies nennt man eine **lokale Variable**.

```
void main (void)
{
      unsigned char a,b,c;    // Definition von drei 1-Byte-Variablen
      . . . .                 // Programm
}
```

Zahlen

	Beispiele char signed char	Beispiele unsigned char	Beispiele int signed int	Beispiele unsigned int	Beispiele float
Dezimalzahl	15 -128	230	3039 -32535	55535	1.1 -1.3e3
Hexadezimalzahl	0x0F 0xFF	0xE6 0xe6	0x0BDF 0xFFFF	0xD8EF	
Dualzahl*	0b00001111	0b11100110			

* nicht in Ansi-C definiert

Operatoren

Das Gleichheitszeichen ist in C ein Zuweisungsoperator, d.h. man weist damit einer Variablen einen Wert zu.

Zuweisung eines bestimmten Zahlenwertes beim Programmstart ("Initialisierung")

char c = 5; bedeutet, dass für die Variable c der Startwert 5 gespeichert wird

Hochzählen einer Zahl

c = c + 5; bewirkt, dass zum vorhandenen Wert von c die Zahl 5 addiert und das Ergebnis wieder unter c gespeichert wird.

Zuweisung eines Rechenergebnisses

c = a / b; Das Ergebnis der Division von a durch b wird in der Variablen c gespeichert

Rechenoperatoren

Operator	Beschreibung	Beispiele
+	Addition	x = 34 + 45; x = x + 3;
+=	Addition mit Zuweisung	x += 3; wie x = x + 3
++	Inkrement	x++; wie x = x + 1;
-	Subtraktion	x = 45 – 34; x = x – 3;
-	Multiplikation mit -1	x = -x;
-=	Subtraktion mit Zuweisung	x -= 3; wie x = x - 3;
--	Dekrement	x --; wie x = x - 1;
*	Multiplikation	x = 3 * 5; x = x * 5;
*=	Multiplikation mit Zuweisung	x *=5; wie x = x * 5;
/	Division	x = 5 / 7; x = x / 7;
/=	Division mit Zuweisung	x /= 7; wie x = x / 7;
%	Modulo Division für ganze Zahlen, ergibt den Rest der Division	x = 14 % 4; // Ergebnis 2

Vergleichsoperatoren

Das Ergebnis von logischen Operatoren ist immer 0 oder 1 (falsch oder wahr)

Operator	Beschreibung	Beispiele
>	größer als	while (x > 5)
>=	größer gleich	if (x >=0)
<	kleiner als	while (a < b)
<=	kleiner gleich	while (y <= 4.6)
==	gleich	if (a == 0)
!=	ungleich	if (a !=0)

Logische Operatoren

Verknüpfung zweier 1-Bit-Ergebnisse

Operator	Beschreibung	Beispiele
&&	logisches UND	if ((a > 5) && (Taster == 1))
||	logisches ODER	while ((a ==0) || (Taster == 0))
!	logisches NICHT	if (!Taster)

Bitweise Operatoren

Bitweise Verknüpfungen von Variablen, Maskierung, Verschieben

Operator	Beschreibung	Beispiele

&	bitweise logisches UND	x = 0x03 & 0x06; // Ergebnis: 0x02
		0000 0011 ≙ 0x03
		0000 0110 ≙ 0x06
		0000 0010 ≙ 0x02

|	bitweise logisches ODER	x = a | 0x0F; // niederwertige 4 Bits eins setzen

^	bitweises EXOR	x = a ^ 0x0F // für a=0x55 wird x=0x5A
		EXOR mit 0 lässt das Bit unverändert, EXOR mit1 kippt das Bit.

>>	nach rechts schieben	x = x >> 1; // um 1 Bit nach rechts schieben
		// um 1 Bit nach rechts schieben
		Bei unsigned wird in die oberste Bitstelle eine 0 hinein geschoben, bei signed wird eine 1 (Vorzeichenerweiterung!)

<<	nach links schieben	x = x << 2; // um 2 Bits nach links schieben

~	bitweise Negation	P2.0 = ~P2.0; // blinken
		x = ~x; // alle Bits invertieren

Befehlabarbeitung:

- rechts neben dem = von links nach rechts (wird von RIDE nicht immer eingehalten!)
- Operator vor Punkt vor Strich, im Zweifelsfall Klammern setzten!

Beispiel: a= 15;
a += ++a + a++;
++a ergibt 16, plus 16 ergibt 32, anschließend a++, += addiert zu 32 die 17 hinzu -> Ergebnis a = 49

Schleifen

Sie dienen zur Wiederholung von Programmteilen.
C bietet **drei verschiedene Möglichkeiten**:

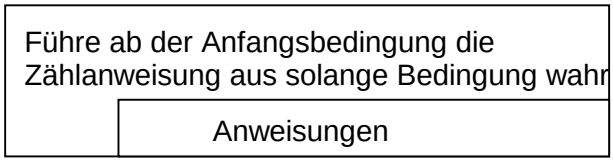
For-Schleife: Erzwingt eine genau berechenbare **Zahl der Wiederholungen:**

While-Schleife: Wird **nur solange** wiederholt, wie eine am **Schleifenanfang** stehende Bedingung **erfüllt** ist. (kopfgesteuerte Schleife)

Do-While-Schleife: Die Schleife wird prinzipiell erst mal durchlaufen.
Am Ende des **Durchganges** steht eine Prüfbedingung, die entscheidet, ob die Schleife wiederholt wird. (fußgesteuerte Schleife)

Die "for" - Schleife

Aufbau: for (Initialisierung; Bedingung; Zählen) { . . . . . }



Initialisierung: Anfangswert der Variablen
Bedingung: Schleife wird solange durchlaufen wie die Bedingung wahr ist
Zählen: Anweisung zum Erhöhen oder Erniedrigen der Variablen
Damit ist eine exakt **vorgeschriebene Anzahl von Wiederholungen eines Programmteiles** erreichbar.

Beispiel1: Der Ausgang soll 10mal invertiert werden.

```
for (x=10; x!=0; x--)
{
        ausgang =~ausgang;
}
```

Beispiel2: Warteschleifen

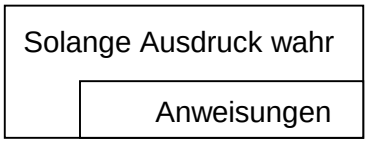
```
unsigned char x;
....
for (x=255; x!=0; x--);          // erzeugt Warteschleifen von ca. 255 x 2µs
                                 // (Vergleich Assembler: mit djnz Rn- Befehlen)

unsigned int x;
....
for (x=0xFFFF; x!=0; x--);       // erzeugt lange Warteschleifen
                                 // Zeit nicht berechenbar

unsigned char x,y;
....
for (x=255; x!=0; x--)           // erzeugt Warteschleifen von ca. 255 x 255 x 2µs
        {                        // da 2 ineinander verschachtelte Schleifen
        for (y=255; y!=0; y--);// mit 8 Bit-Variablen erzeugt werden
        }
```

Die “while”-Schleife

Aufbau: **while** (Bedingung) { . . . . . }



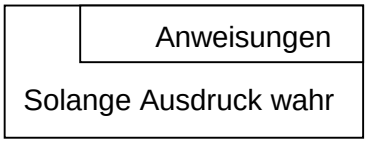
Prinzip: Wenn die am Schleifenanfang stehende Bedingung **nicht gilt**, dann wird die gesamte Schleife **übersprungen**
Solange die am Schleifenanfang stehende Bedingung **gilt**, wird die Schleife **wiederholt.**
Die Prüfbedingung steht **vor** den Anweisungen.
Sie heißt deshalb “kopfgesteuerte Schleife”.

Beispiel: Solange der Taster gedrückt ist (Taster ist 1) wird “ausgang” invertiert.
Wenn der Taster vor dem Schleifendurchlauf schon gedrückt ist,
wird der Ausgang nicht invertiert.

```
while (taster == 0)
{
        ausgang = ~ausgang;
}
```

Die “do while” - Schleife

Aufbau: **do** { . . . . . }
while (Bedingung)



Prinzip: Die Schleife wird **mindestens einmal durchlaufen** , die Bedingungsprüfung folgt am Schluss. Man spricht daher von einer „fußgesteuerten Schleife“.

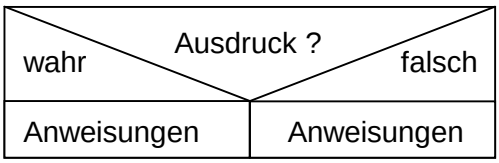
Beispiel: Die Schleife wird maximal 100 mal und mindestens 1 mal durchlaufen.
Sie wird frühzeitig abgebrochen, wenn der Taster gedrückt (= 1) wird.

```
x = 100;
do
{
    x--;
}
while ((x > 0) && (taster == 0));
```

Programmverzweigung

Die “if” - Anweisung

Aufbau:
if (Bedingung_1) { . . . . . }
[**else if** (Bedingung_2) { . . . . . }] optional
[**else if** (Bedingung_3) { . . . . . }] optional
[**else** { . . . . . }] optional



Bei der “if” - Anweisung werden die folgende Anweisung (oder ein ganzer Anwendungsblock) **nur dann ausgeführt**, wenn die **hinter ”if” stehende Bedingung wahr** ist.

Beispiel: Wenn “taster1” gedrückt ist, soll “ausgang1” eins und „ausgang2“ null werden. Drückt man dagegen “taster2”, wird nur “ausgang2” zu eins.

```
if (taster1 == 1)
{                                   // Block mit mehreren Anweisungen
    ausgang1 = 1;                   // wird ausgeführt, wenn die Bedingung
    ausgang2 = 0;                   // hinter if wahr ist
}

if (taster2 == 1) ausgang2 = 1;     // nur eine Anweisung, keine { } nötig
```

Mit **“if - else “** kann **nur zwischen zwei Alternativen** wählen.

Beispiel: Wenn “taster1” gedrückt ist, soll “ausgang1” eins und „ausgang2“ null werden, andernfalls soll “ausgang1” null und „ausgang2“ eins werden.

```
if (taster1 == 1)
{                                   // Block mit mehreren Anweisungen
    ausgang1 = 1;                   // wird ausgeführt, wenn die Bedingung
    ausgang2 = 0;                   // hinter if wahr ist
}
else
{                                   // Block mit mehreren Anweisungen
    ausgang1 = 0;                   // wird ausgeführt, wenn die Bedingung
    ausgang2 = 1;                   // hinter if nicht wahr ist
}
```

C-Spezialität: Das ?

das Programm

```
if (x > 9)
    y = 100;
else
    y = 200;
```

kann verkürzt dargestellt werden mit

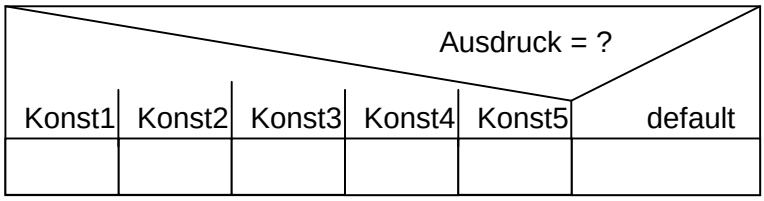
```
y = x > 9 ? 100 : 200;
```

Ob das zur Lesbarkeit des Programms beiträgt?

Die “switch” - Anweisung

Aufbau:

```
switch (Variable)
{
case Konstante_1:   { . . . . .
                      break;
                      }
case Konstante_2:   { . . . . .
                      break;
                      }
}
```



Mit der “**switch**”- Anweisung kann aus einer **Reihe von Alternativen** ausgewählt werden.
Es ist zulässig, dass mehrere Möglichkeiten gültig sind und dieselbe Wirkung haben.
Sie werden einfach nacheinander aufgelistet.
Passt **keine der Möglichkeiten**, dann wird die “**default**” - Einstellung ausgeführt.
Achtung! **Auf keinen Fall break vergessen!!!**

Beispiel: In der Variablen “ergebnis” ist ein Messergebnis oder eine Zahl gespeichert.
Abhängig vom genauen Wert sollen nun bestimmte Reaktionen erfolgen.

```
switch (ergebnis)
{
    case 0x00:
    case 0x10:
    case 0x20:
        ausgang1 = 1;
        break;
    case 0x30:
        ausgang1 = 0;
        break;
    case 0x40:
        ausgang1 = ~ ausgang1;
        break;
    default:
        ausgang2 = 1;
        break;
}
```

Hinweise: Für die „switch-Variable“ darf man nur einfache Datentypen verwenden.
Hinter case müssen Konstanten stehen.
Diese können mit #define am Anfang des Programms deklariert werden.
Oft schreibt man die Anweisungen hinter case in eine Zeile.

```
#define rechts      0x10          // ohne Semikolon!!
#define links       0x20
unsigned char richtung;
....
switch (richtung)
{
    case rechts:   motor = rechtskurve; break;
    case links:    motor = linkskurve; break;
    default:       motor = vorwaerts; break;
}
```

Funktionen

Aufbau:

Funktionsdeklaration

typ fkn_name (typ [,typ, . . .] optional);

Funktionsdefinition

typ fkn_name (typ [name] optional [,typ name, . . .] optional)
{
[return rückgabewert] optional
}

Funktionsaufruf

fkn_name ([übergabewert, ...] optional);

Achtung:

Wenn eine Funktion **definiert** wird, folgen **direkt hinter dem Funktionsnamen nur zwei runde Klammern, dahinter aber nix mehr (also NIE ein Strichpunkt)!!**
Wenn eine Funktion **aufgerufen (= also verwendet)** wird, dann **muss der Strichpunkt hinter der Funktion stehen...**

Funktion ohne Übergabewert

```
void zeit (void)                // Funktion wird vor der Verwendung definiert
{
    unsigned char x;            // lokale Variable
    for (x=255; x!=0; x--);     // erzeugt Zeitverzögerung von ca. 255 x 2µs
}

void main (void)
{
    unsigned char a,b,c;        // Definition von drei 1-Byte-Variablen
    . . . .                     // Programm
    zeit ();                    // Funktionsaufruf, keine Werteübergabe
}
```

Alternative Funktion ohne Übergabewert:

```
void zeit (void);               // Funktion muss vor der Verwendung
                                // deklariert (angemeldet) werden

void main (void)
{
    unsigned char a,b,c;        // Definition von drei 1-Byte-Variablen
    . . . .                     // Programm
    zeit ();                    // Funktionsaufruf, keine Werteübergabe
}

void zeit (void)                // Funktion wird hinter der Verwendung
{                               // definiert
    unsigned char x;            // lokale Variable
    for (x=255; x!=0; x--);     // erzeugt Zeitverzögerung von ca. 255 x 2µs
}
```

Funktion mit Übergabewert

```
void zeit (unsigned char ms)            // 8-Bit-Übergabevariable ms
{
    unsigned char t1,t2;                // lokale Variable
    for (t1 = ms; t1 != 0; t1--)        // äußere Schleife ms * 1Millisekunde
    {
        for (t2 = 249; t2 != 0; t2--);  // erzeugt Zeitverzögerung ca. 500µs
        for (t2 = 249; t2 != 0; t2--);  // erzeugt Zeitverzögerung ca. 500µs
    }
}

void main (void)
{
    . . . .                             // Programm
    zeit (100);                         // Funktionsaufruf mit Werteübergabe
                                        // hier: 100ms Zeitverzögerung
}
```

Funktion liefert einen Wert

```
// Funktion liefert den 8-Bit-Wert des Ports port_nr, wobei port_nr = 0 bis 3 sein kann
unsigned char hole_wert (unsigned char port_nr)
{
    unsigned char p;                    // lokale Variable
    if (port_nr == 0) p = P0;           // Port einlesen und zwischenspeichern
    if (port_nr = =1) p = P1;           // Port einlesen und zwischenspeichern
    if (port_nr = =2) p = P2;           // Port einlesen und zwischenspeichern
    if (port_nr = =3) p = P3;           // Port einlesen und zwischenspeichern
    return (p);
}

// Hauptprogramm mit Funktionsaufruf
void main (void)
{
    unsigned char a;                    // lokale Variable
    . . . .                             // Programm
    a = hole_wert (2);                  // Funktionsaufruf mit Werteübergabe
                                        // Funktion liefert den Wert von P2 zurück
}
```