

2.3.1 Programmverzweigungen

Die einfache if - Anweisung

```
if (Bedingung) { . . . . }
```

Die Anweisungen im Ausführungsblock werden **nur dann ausgeführt**, wenn die **Bedingung wahr** ist.
(wahr bedeutet: *Bedingung* $\neq 0$).

Bedingung	
J	N
Anweisung 1 ... Anweisung n	weiter ...

Beispiele für Bedingungen:

```
if ( i > 0 ) ...  
if ( (wert & 0x08) == 0x08 ) ...
```

Die mehrfache if - Anweisung

```
if (Bedingung_1) { . . . . }  
else if (Bedingung_2) { . . . . }  
else if (Bedingung_3) { . . . . }  
else { . . . . }
```

Es wird immer zuerst "Bedingung 1" geprüft.
Nur wenn diese falsch ist, wird "Bedingung 2" überprüft, usw.

Bedingung 1			
J	?		N
Anweisungs Block 1	J	Bedingung 2	
	?		N
	Anweisungs Block 2		
	J	Bedingung 3	
	?		N
	Anweisungs Block 3		
		Sonst Anweisungen	

Bsp.: In der Variablen "messwert" wird ein Messergebnis gespeichert.

Abhängig vom Wertebereich wird dann der Variablen "ausgang" ein Wert zugewiesen.

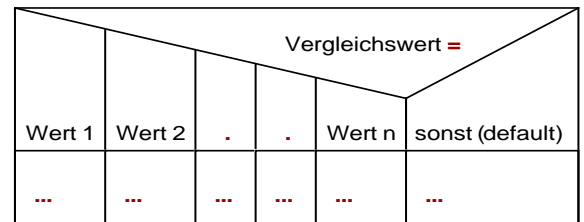
```
messwert = getMesswert();      // neuen Messwert holen  
  
if (messwert < 20)  
{  
    ausgang = true;            // Messwert < 20  
}  
else if (messwert < 30)  
{  
    ausgang = false;          // Messwert 20 ... 29  
}  
else if (messwert < 40)  
{  
    ausgang = !ausgang;       // Messwert 30 ... 39  
}  
else  
{  
    ausgang = ausgang;        // Messwert >= 40  
}
```

Die switch - Anweisung

```

switch (Vergleichswert)
{
    case Wert_1: ....
                break;
    case Wert_2: .....
                break;
    default:    .....
                break;
}

```



Mit der **“switch”**- Anweisung kann, abhängig vom Vergleichswert, aus einer **Reihe von Alternativen** ausgewählt werden. Es aber keine Bereichsüberprüfung erfolgen. Jedoch ist es zulässig, dass mehrere Alternativen dieselbe Wirkung haben. Sie werden dann einfach nacheinander (oder hintereinander) aufgelistet.

Passt **keine der Möglichkeiten**, dann wird die **“default”** - Einstellung ausgeführt. Break unterbricht jeweils die Abarbeitung der switch-Anweisung. (**break nicht vergessen!!!**)

Bsp.: Abhängig vom genauen Wert erfolgt nun die Reaktion.

```

switch (messwert)
{
    case 0:
    case 10:
    case 20:                // Messwert = 0, 10, 20
        ausgang = 1;
        break;
    case 30:                // Messwert = 30
        ausgang = 0;
        break;
    case 40:                // Messwert =40
        ausgang = ~ausgang;
        break;
    default:                // Alle anderen Messwerte
        ausgang = ausgang;
        break;
}

```

Hinweis: Für die „switch-Variable“ darf man nur Ganzzah- 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;
}

```

2.3.2 Programmschleifen

Schleifen 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. While- und For-Schleife sind meist alternativ verwendbar. (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*) { . . . }

Für (*Initialisierung; Bedingung; Zählen*)

Anweisungsblock

Initialisierung: Schleifenzähler setzen

Bedingung: i.d.R. Abfrage des Schleifenzählers

Zählen: Anweisung zum Erhöhen oder Erniedrigen des Schleifenzählers

Damit ist eine exakt **vorgeschriebene Anzahl von Wiederholungen eines Programmteiles** erreichbar.

Bsp.: Durchlaufen eines Arrays zur Initialisierung von 8 GPIO's als Ausgang.

```
const int led[] = {9,10,14,4,33,15,13,32};

void setup()
{
  for (int i = 0; i<8; i++)
  {
    pinMode(led[i],OUTPUT);           // Alle LEDs als Ausgang
    digitalWrite(led[i],HIGH);       // Alle LEDs aus
  }
}
```

Die while - Schleife

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

Wiederhole, solange Bedingung


Anweisungsblock

Prinzip: Wenn die am Schleifenanfang stehende Bedingung **nicht erfüllt ist (=falsch)**, dann wird die gesamte Schleife **übersprungen**.

Solange die am Schleifenanfang stehende Bedingung **erfüllt ist (=wahr)**, wird die Schleife **wiederholt**.

Die Prüfbedingung steht **vor** den Anweisungen. Sie heißt deshalb "kopfgesteuerte Schleife".

Bsp.: Die Schleife wird nur durchlaufen, wenn der Taster gedrückt ist (hier: 0-Aktiv). Solange der Taster gedrückt bleibt wird die Schleifenanweisung fortlaufend ausgeführt.

 Friedrich-Ebert-Schule Esslingen FES	IT: Hardwarenahes Programmieren	Name: Rahm Datum: 29.09.2022 2_3_Programmsteuerung_in_C.docx
	Anweisungen zur Programmsteuerung in C	2.3.4

```
while (digitalRead(pinTaster) == 0)
{
    ausgang = ~ausgang;
}
```

Anmerkung: Ist eine Bedingung **falsch**, wird dies in C als Zahlenwert 0 dargestellt (falsch = 0). Alle Werte $\neq 0$ entsprechen einer **wahren** Bedingung (wahr $\neq 0$).

Um den Programmablauf zu stoppen, wird häufig eine **Endlosschleife** programmiert. Die beiden folgenden Möglichkeiten sind in der Wirkung identisch:

```
while(1);    // Fortsetzungsbedingung ist immer 1 = true
for(;1;);    // dito.
```

Die do while - Schleife

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

Anweisungsblock

Wiederhole, solange Bedingung

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

Bsp.: Die Schleife wird maximal 100 mal und mindestens 1 mal durchlaufen. Sie wird frühzeitig abgebrochen, wenn der Taster gedrückt (= 0) wird.

```
uint8_t x = 100;
do
{
    x--;
}
while ((x > 0) && (digitalRead(pinTaster) == 1));
```

alternativ: `while ((x>0) && digitalRead(pinTaster));`

oder: `while (x && digitalRead(pinTaster));`

2.3.3 Funktionen

Funktionen werden in C verwendet, um die Lesbarkeit und die Wiederverwendbarkeit eines Programmes zu erhöhen. Dabei unterscheidet man die Deklaration (Bekanntmachung) und die Definition (Festlegung der Funktionalität) von Funktionen. In C müssen alle Funktionen vor ihrer Benutzung deklariert werden. Die Aufgabe von Funktionen ist es in erster Linie Eingabedaten entgegenzunehmen, zu bearbeiten und wieder auszugeben. Daten können dabei in internen (nur lokal gültigen) Variablen gespeichert werden, oder es können globale Variablen, die außerhalb der Funktion gültig sind bearbeitet werden.


Funktionsdeklaration

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

Funktionsdefinition

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

```
{
    [return rückgabewert] optional
}
```

 Friedrich-Ebert-Schule Esslingen FES	IT: Hardwarenahes Programmieren	Name: Rahm Datum: 29.09.2022 2_3_Programmsteuerung_in_C.docx
	Anweisungen zur Programmsteuerung in C	2.3.5

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)

```
void warten(void)      // Funktion wird vor Aufruf definiert
{
    for (uint8_t zaehl=0; zaehl<255; zaehl++)
        { };           // lokale Variable zaehl ist nur
                        // in der for-Schleife gültig

void loop()
{
    ...
    warten();          // Funktionsaufruf ohne Parameter
    ...
}
```

Erfolgt die Definition der Funktion im Quellprogramm nach der Funktion main(), muss eine explizite Deklaration (Funktionsprototyp) erfolgen:


```
void warten(void);     // Funktionsdeklaration vor Aufruf

void loop()
{
    ...
    warten();          // Funktionsaufruf ohne Parameter
    ...
}

void warten(void)      // Funktion wird nach Aufruf definiert
{
    for (uint8_t zaehl=0; zaehl<255; zaehl++) { };
}
```

Funktion mit Übergabeparameter

Variablen die einer Funktion übergeben werden, nennt man auch Funktionsargumente. Bei jedem Aufruf der Funktion wird daraus eine neue lokale Variable als Kopie erzeugt. Wird diese innerhalb der Funktion geändert, so wird die Änderung nach außen nicht sichtbar. Man nennt dieses Verhalten auch **call-by-value**.

 Friedrich-Ebert-Schule Esslingen FES	IT: Hardwarenahes Programmieren	Name: Rahm Datum: 29.09.2022 2_3_Programmsteuerung_in_C.docx
	Anweisungen zur Programmsteuerung in C	2.3.6

```

void warten(uint8_t zaehl)      // Ein Parameter wird übergeben
{
    while (zaehl != 0) zaehl--; // Herunterzählen bis 0
}

void loop()
{
    uint8_t n = 100;

    warten(n);                  // Funktionsaufruf mit Wert-Übergabe
    Serial.print(n);           // Es wird 100 ausgegeben
    while(1);                  // Endlosschleife
}

```

In manchen Fällen wird der Wert der Variablen von der aufrufenden Funktion benötigt. Dann muss ein **call-by-reference** verwendet werden. Hier wird nicht der Wert der Variablen übergeben, sondern die Speicheradresse. In C benötigt man dafür sogenannte **Zeiger** (engl. Pointer):

```

void warten(uint8_t *zaehl)     // Der Parameter *zaehl ist ein Zeiger
{                               // auf eine Speicherstelle (=Adresse).
    while (*zaehl != 0) *zaehl--; // Der Inhalt wird bis 0 runtergezählt
}


void loop()
{
    uint8_t n = 100;           // Die Variable n wird an einer festen
                               // Speicherstelle im RAM angelegt.

    warten(&n);                // Übergabe der Adresse von n
    Serial.print(n);           // Es wird 0 ausgegeben
    while(1);
}

```

Der &-Operator wird auch als Referenzierer (Adressoperator) bezeichnet, der *-Operator ist entsprechend der Dereferenzierer.

Hinweis: Wir versuchen Pointer möglichst zu vermeiden!

 Friedrich-Ebert-Schule Esslingen FES	IT: Hardwarenahes Programmieren	Name: Rahm Datum: 29.09.2022 2_3_Programmsteuerung_in_C.docx
	Anweisungen zur Programmsteuerung in C	2.3.7

Funktion liefert einen Wert

```
double square(float n)      // Rückgabotyp double
{
    double s = n*n;
    return s;               // Rückgabe mit return
}

void loop()
{
    double y;
    float x = 56.6;

    y = square(x);          // Zuweisung des Ergebnisses an y
    Serial.print("Quadrat = ");
    Serial.println(y);

    while(1);
}
```

Einer Funktion können auch mehrere Übergabewerte (auch von unterschiedlichen Datentypen) übergeben werden.

```
double pot(float n, uint8_t e)  // Verschiedene Datentypen
{
    double s = n;
    for (uint8_t i=e; i>1; i--)  // Berechnung der Potenz
        s = s * n;              // in der for-Schleife

    return s;                   // Rückgabe mit return
}

void loop()
{
    double y;
    float x = 23.76;

    y = pot(x,3);               // Funktionsaufruf
    Serial.print("Potenz = ");
    Serial.println(y);

    while(1);
}
```