

# Metalab Kurs $\mu$ C-Programmierung in C

Clifford Wolf  
Stefan Farthofer

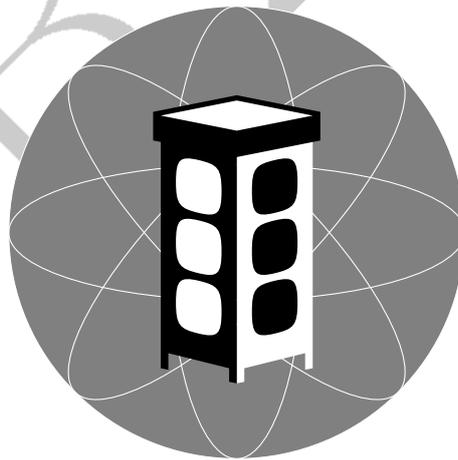
im April 2012

Skriptum zum Metalab Kurs „ $\mu$ C-Programmierung in C“. Diese Unterlagen sind begleitend zum Kurs gedacht. Die Kursinhalte werden in diesem Skriptum kompakt zusammengefasst.

[http://metalab.at/wiki/uCProg\\_Kurs](http://metalab.at/wiki/uCProg_Kurs)

Dieses Skriptum ist vollständig in  $\LaTeX$  gesetzt. Die Zeichnungen, Schaltpläne und Diagramme wurden mittels PGF und TikZ direkt in  $\TeX$  erstellt.

Lektorat: Astrid Gruber und Daniel Maierhofer



Dieses Werk ist unter der Creative Commons BY-NC-SA-Lizenz lizenziert.  
<http://creativecommons.org/licenses/by-nc-sa/3.0/at/>

This page is intentionally left blank.

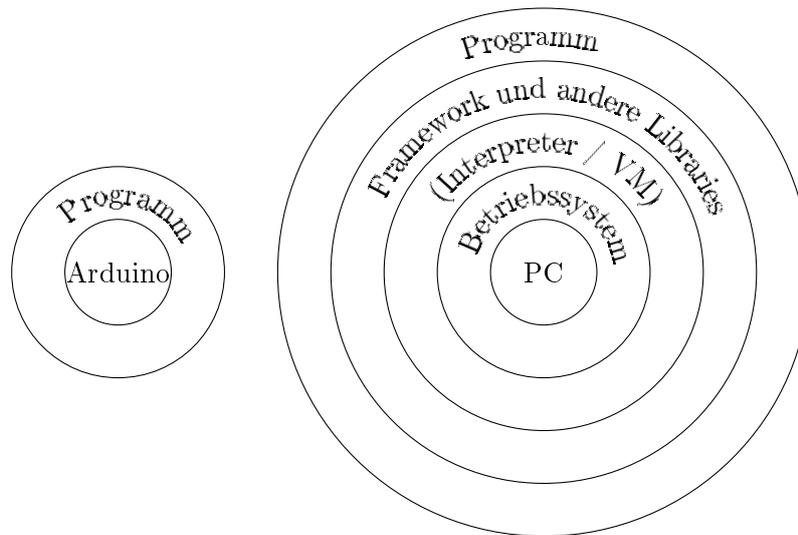
## Inhaltsverzeichnis

Block 0: Die Arduino Experimentierumgebung .....	4
Block 1: Ein Arduino Hello-World Programm .....	5
Block 2: Rechnen mit dem Arduino .....	6
Block 3: Arduino IO-Pins .....	8
Block 4: Definieren weiterer Funktionen .....	9
Block 5: Rückgabewerte .....	11
Block 6: Globale und lokale Variablen .....	12
Block 7: Struktogramme .....	14
Block 8: Fallunterscheidungen .....	15
Block 9: Boolesche Ausdrücke .....	17
Block 10: Schleifen .....	19
Block 11: Zählerschleifen .....	21
Block 12: Pre- und Postinkrement .....	23
Block 13: Einfache Datentypen .....	24
Block 14: Arrays .....	25
Block 15: Mehr zu Arrays .....	27
Block 16: Zusammengesetzte Datentypen .....	29
Block 17: Pointer .....	31
Block 18: Pointer auf Arrays .....	34
Block 19: Textzeichen .....	36
Block 20: Strings (Zeichenketten) .....	39
Block 21: break, continue und goto .....	41
Block 22: Das switch-Statement .....	42
Block 23: Binärdarstellung von Ganzzahlen .....	44
Block 24: Hezadezimal- und Oktalzahlen .....	45
Block 25: Operationen auf Bitmuster .....	47
Block 26: Division und Modulo .....	49
Block 27: Zusammengesetzte Zuweisungsoperatoren .....	50
Block 28: Der ?: Operator .....	51
Block 29: Operatoren und Operatorbindungen .....	52
Block 30: Defines und bedingte Übersetzung .....	53
Anhang I: Protokoll der Einheiten .....	56
Anhang II: Shield .....	57

## Block 0: Die Arduino Experimentierumgebung

Der Arduino ist ein kleiner Computer. Mit der Arduino-Software kann der Arduino in C programmiert werden. Diese Programme können über die USB-Schnittstelle auf den Arduino übertragen werden.

Im Gegensatz zum PC hat man beim Arduino direkten Zugriff auf die Hardware.



Das verringert die Komplexität des Gesamtsystems. Somit ist weniger Hintergrundwissen notwendig um Programme zu schreiben.

Genausowenig wie ein PC ohne Maus, Tastatur und Bildschirm sinnvoll ist, so ist ein Arduino ohne die sogenannten *Shields*, die den Arduino um Peripherielemente erweitern, sinnvoll.

Wir verwenden für unsere Übungen unser eigenes „Ampel-Shield“. Üblicherweise entwickelt man für seine Projekte eigene Shields. Es gibt aber eine große Menge verschiedenster fertiger Shields in diversen Online-Shops zu kaufen.

### Übung:

Installieren Sie die Arduino Software und spielen Sie das **Digital/Blink** Example in den Arduino ein.

### Diskussion:

Welche Eingabe-/Ausgabe-Schnittstellen hat ein Arduino? Recherchieren Sie gegebenenfalls im Internet.

### Übung:

Installieren Sie die **Console** Library, die wir für diesen Kurs verwenden. Folgen Sie dazu den Anweisungen im **README**-File.

## Block 1: Ein Arduino Hello-World Programm

Wir schreiben unser erstes Arduino Programm.

Ein Programm besteht aus beliebig vielen Funktionen die wir selbst schreiben müssen. Darüber hinaus existiert eine Sammlung vorgefertigter Funktionen, die wir verwenden können (so eine Sammlung heißt *Library*).

```
1 #include <Console.h>
2
3 void setup()
4 {
5     consoleInit(9600);
6     consolePrint("Hello World!\n");
7 }
8
9 void loop()
10 {
11     // Hier passiert nichts.
12 }
```

block01\_hallo.pde

Um Funktionen aus einer Library einzubinden wird das **#include** Statement verwendet. Die Funktionen **consoleInit** und **consolePrint** stammen aus der **console** Library. Um diese verwenden zu können muss die Datei **Console.h** eingebunden werden.

Jedes Arduino-Programm muss die Funktionen **setup** und **loop** definieren. Die **setup** Funktion wird einmal beim Programmstart ausgeführt. Die **loop** Funktion danach in einer Endlosschleife.

Eine Funktion kann andere Funktionen hintereinander aufrufen.

Funktionen können Werte übergeben bekommen und einen Wert zurückliefern. Das Schlüsselwort **void** im Funktionskopf der Funktionen **setup** und **loop** signalisiert, dass diese Funktionen keine Werte zurückliefern.

Die Funktion **consoleInit** initialisiert die serielle Schnittstelle auf eine Baudrate von 9600 Baud. Die Funktion **consolePrint** gibt einen Text aus. Texte werden in C in doppelten Anführungszeichen geschrieben, wobei „\n“ hier für einen Zeilenumbruch steht.

Der Text nach einem „//“ (bis zum Zeilenende) ist ein Kommentar und wird von der Entwicklungsumgebung ignoriert.

### Übung:

Führen Sie das Programm am Arduino aus und sehen Sie sich die Ausgabe auf der seriellen Konsole der Arduino-Software an.

Was passiert, wenn Sie den Reset-Knopf am Arduino drücken?

### Diskussion:

Wie ist die exakte Syntax für Funktionsaufrufe und Funktionsdefinitionen? Welchen Zweck haben die Strichpunkte im Programmcode?

### Diskussion und Experiment:

Was passiert, wenn Sie den **consolePrint** Aufruf in die **loop** Funktion verschieben?

Was passiert, wenn Sie statt des Funktionsnamens **consolePrint** den Funktionsnamen **consoleprint** verwenden?

## Block 2: Rechnen mit dem Arduino

Eine Variable ist ein Name für eine Speicherstelle. Eine Variable wird durch Angabe des Datentyps (hier **int**) gefolgt vom Variablennamen definiert.

Der Datentyp **int** kann Ganzzahlen im Wertebereich  $-32768 \dots 32767$  speichern<sup>1</sup>.

Mit dem Operator = (Zuweisungsoperator) kann ein Wert in einer solchen Speicherstelle gespeichert werden.

```
1 #include <Console.h>
2
3 int summe;
4 int eingabe;
5
6 void setup()
7 {
8     consoleInit(9600);
9     summe = 0;
10 }
11
12 void loop()
13 {
14     eingabe = consoleReadDecimal("Geben Sie eine Zahl ein: ");
15     summe = summe + eingabe;
16
17     consolePrintf("Die Summe ist derzeit: %d\n", summe);
18 }
```

block02\_summe.pde

Die Funktion `consoleReadDecimal` gibt den als Parameter übergebenen Prompt aus und liest eine Ganzzahl ein, die von der Funktion als Rückgabewert zurückgeliefert wird.

Die Funktion `consolePrintf` gibt wie `consolePrint` ihren ersten Parameter auf der Konsole aus, wobei dieser erste Parameter Platzhalter wie „%d“ beinhalten kann, an deren Stelle Textrepräsentationen der folgenden Parameter gesetzt werden.

Variablen, die außerhalb von Funktionen definiert werden, behalten ihren Wert über die gesamte Lebensdauer des Programms bei.

---

<sup>1</sup>Dieser Wertebereich gilt für den Prozessor am Arduino. Auf anderen Plattformen kann dieser Wertebereich ganz anders aussehen. So kann z.B. ein **int** am PC gewöhnlich Zahlen von  $-2\,147\,483\,648$  bis  $2\,147\,483\,647$  speichern.

### Diskussion:

Umgangssprachlich wird der Begriff der Variable und der Begriff der Speicherstelle austauschbar verwendet.

Wie verwaltet der Computer (Arduino) intern die Speicherstellen? Muss man als Programmierer auch auf dieser hardwarenäheren Ebene mit dem Speicher arbeiten?

### Diskussion:

Wie werden im Computer (Arduino) Zahlenwerte intern gespeichert?

### Übung:

Was passiert, wenn man dem Programm 4 mal die Eingabe „10000“ übergibt? Erklären Sie Ihre Beobachtungen.

## Block 3: Arduino IO-Pins

Die meisten Pins des Arduino (ATMega AVR  $\mu C$ ) sind frei als Eingabe- oder Ausgabepins verwendbar. Wenn der Pin für Eingabe verwendet wird, kann optional ein interner Pullup-Widerstand zugeschaltet werden.

```
1  int pinValue;
2
3  void setup()
4  {
5      pinMode(15, INPUT);
6      pinMode(3, OUTPUT);
7      digitalWrite(15, 1); // enable pullup resistor
8  }
9
10 void loop()
11 {
12     pinValue = digitalRead(15);
13     digitalWrite(3, pinValue);
14 }
```

block03\_pinio.pde

Die `pinMode` Funktion konfiguriert einen Pin als Ein- oder Ausgabepin. In diesem Beispiel wird Pin Nummer 2 als Eingabepin und Pin Nummer 13 als Ausgabepin verwendet. Pin Nummer 13 ist am Arduino auch direkt mit der LED „L“ verbunden.

Für Eingabepins aktiviert/deaktiviert `digitalWrite` den Pullup-Widerstand und für Ausgabepins setzt `digitalWrite` den Wert.

Bei einem Eingabepin kann mit der Funktion `digitalRead` der aktuelle Wert ausgelesen werden.

Die Wörter `INPUT` und `OUTPUT` werden in der Arduino-Library als Synonyme für die Zahlenwerte 0 und 1 definiert.

### Diskussion und Übung:

Was macht das Beispielprogramm? Welche externe Beschaltung ist notwendig?

Probieren Sie das Programm aus.

### Diskussion:

Weshalb definiert man Synonyme (wie `INPUT` und `OUTPUT`) für Zahlenwerte? Was sind die Vor- und Nachteile?

### Diskussion:

In welchen Fällen braucht man einen Pullup-Widerstand? Wann kann ein Pullup-Widerstand stören?

## Block 4: Definieren weiterer Funktionen

Wir schreiben eine eigene Funktion mit Parameter:

```
1 void blinkPin(int pin)
2 {
3     digitalWrite(pin, 1);
4     delay(1000);
```

## Block 4: Definieren weiterer Funktionen

---

```
5     digitalWrite(pin, 0);
6     delay(100);
7 }
8
9 void setup()
10 {
11     pinMode(3, OUTPUT);
12     pinMode(4, OUTPUT);
13 }
14
15 void loop()
16 {
17     blinkPin(3);
18     blinkPin(3);
19     blinkPin(4);
20 }
```

block04\_blink1.pde

Wie die Funktionen `setup` und `loop` liefert unsere `blinkPin`-Funktion keinen Rückgabewert, was durch das Schlüsselwort `void` signalisiert wird. Im Gegensatz zu `setup` und `loop` akzeptiert `blinkPin` einen Parameter `pin` vom Datentyp `int`. Dazu wurde der Parameter in die runden Klammern gesetzt, die bei parameterlosen Funktionen wie `setup` und `loop` leer bleiben. Bei Funktionen mit mehreren Parametern werden diese durch Kommata getrennt:

```
1 void blinkPin(int pin, int lengthMs)
2 {
3     digitalWrite(pin, 1);
4     delay(lengthMs);
5     digitalWrite(pin, 0);
6     delay(100);
7 }
8
9 void setup()
10 {
11     pinMode(3, OUTPUT);
12     pinMode(4, OUTPUT);
13 }
14
15 void loop()
16 {
17     blinkPin(3, 1000);
18     blinkPin(3, 2000);
19     blinkPin(4, 3000);
20 }
```

block04\_blink2.pde

Die Funktion `delay` wartet übrigens die angegebene Anzahl von Millisekunden.

### Diskussion:

Wie ist die vollständige Syntax für eine Funktionsdefinition mit Parametern?

Wozu sind die Parameter überhaupt gut? Welche andere Möglichkeit gibt es Daten zwischen aufrufender und aufgerufener Funktion auszutauschen?

### Übung:

Schreiben Sie eine Funktion `sumDiff` mit zwei `int`-Parametern, die die Summe und die Differenz der beiden Parameter auf der seriellen Konsole ausgibt.

### Diskussion:

Für die beiden `blink`-Programme werden zwei LEDs benötigt. Wie muss man diese mit dem Arduino verschalten, damit das Programm funktioniert?

## Block 5: Rückgabewerte

Wir schreiben eine Funktion, die einen Wert zurückliefert:

```
1  #include <Console.h>
2
3  int getFlashDurationMs()
4  {
5      return consoleReadDecimal("Wieviele Sekunden? ") * 1000;
6  }
7
8  void blinkPin(int pin, int lengthMs)
9  {
10     digitalWrite(pin, 1);
11     delay(lengthMs);
12     digitalWrite(pin, 0);
13 }
14
15 void setup()
16 {
```

```
17   consoleInit(9600);
18   pinMode(3, OUTPUT);
19 }
20
21 void loop()
22 {
23   blinkPin(3, getFlashDurationMs());
24 }
```

block05\_flash.pde

Wenn eine Funktion einen Rückgabewert hat (kein **void**), dann muss das letzte Statement der Funktion ein **return**-Statement sein, das den Wert zurückliefert.

Im Übrigen: der Stern ist der Operator zum Multiplizieren und wie man in Zeile 21 sehen kann, ist es auch möglich Funktionsaufrufe ineinander zu verschachteln.

### Experiment:

Was passiert, wenn man nach dem **return**-Statement weiteren Code (zum Beispiel `consoleWrite("Hallo Welt!\n");`) einfügt?

### Diskussion:

Wie lautet die Syntax des **return**-Statements?

Wozu sind Rückgabewerte überhaupt gut? Welche andere Möglichkeit gibt es, Daten zwischen aufgerufener und aufrufender Funktion auszutauschen?

## Block 6: Globale und lokale Variablen

Variablen, die außerhalb von Funktionen definiert werden, heißen „globale Variablen“. Sie behalten ihren Wert über die gesamte Lebensdauer des Programmes bei.

Variablen, die innerhalb von Funktionen definiert werden, heißen „lokale Variablen“. Sie behalten ihren Wert lediglich über die Laufzeit der Funktion bei. Mit Hilfe von lokalen Variablen kann die Funktion aus Block 5 wie folgt umgeschrieben werden:

```
3 int getFlashDurationMs()  
4 {  
5     int durationSeconds;  
6     durationSeconds = consoleReadDecimal("Wieviele Sekunden? ");  
7     return durationSeconds * 1000;  
8 }
```

block06\_durationms.pde

Da hier der Zwischenwert einen Namen hat, ist diese Version der Funktion lesbarer.

Da lokale Variablen ihren Wert nur über die Laufzeit der Funktion beibehalten braucht die lokale Variable (im Gegensatz zur globalen Variable) nicht permanent Speicher. Diese Variante der Funktion ist sogar genauso effizient wie die vorhergehende, weil der Compiler in der vorhergehenden Version der Funktion automatisch eine lokale Variable ohne Namen für den Zwischenwert anlegen musste.

### Diskussion:

Was sind die Parallelen und Unterschiede zwischen lokalen Variablen und Parametern?

Was passiert, wenn eine lokale Variable und eine globale Variable den gleichen Namen haben? Was passiert, wenn lokale Variablen verschiedener Funktionen den gleichen Namen haben?

### Diskussion:

Kann man eine oder beide der globalen Variablen im Beispiel zu Block 2 auch als lokale Variablen definieren?

### Diskussion:

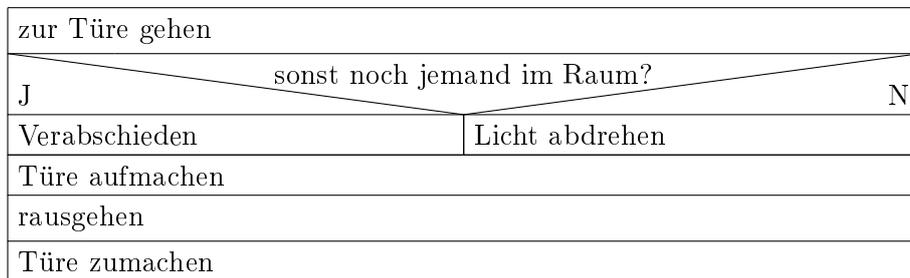
Man kann mit Kommata getrennt auch mehrere Variablen auf einmal definieren und man kann den einzelnen Variablen bereits direkt in der Definition einen initialen Wert zuweisen. Zum Beispiel:

```
int a = 1, b = 2, c = a + b;
```

Was sind dabei die Vor- und Nachteile?

## Block 7: Struktogramme

Ein Struktogramm ist eine graphische Darstellungsform einer Funktion, die besonders leicht zu lesen ist. Dabei wird die Funktion als Rechteck dargestellt, wobei zur Visualisierung der einzelnen Verarbeitungsschritte das Rechteck in kleinere Rechtecke unterteilt wird:



Im Struktogramm stellen Blöcke untereinander eine Sequenz von hintereinander auszuführenden Aktionen dar. Blöcke, die nebeneinander stehen, stellen alternative Sequenzen dar.

### Diskussion:

Wir haben noch keine C Sprachkonstrukte für Fallunterscheidungen kennengelernt. Wie sehen daher die Struktogramme für alle C-Funktionen aus, die wir bis jetzt geschrieben haben?

### Diskussion:

Was ist der Vorteil von einem Struktogramm? Wozu braucht man Struktogramme?

### Übung:

Verarbeiten Sie folgende Abfragen und Aktionen in ein sinnvolles Struktogramm:

- Ist Milch da?
- Ist die Milchpackung offen?
- Ist die Milchpackung leer?
- Kühlschrank aufmachen.
- Kühlschrank zumachen.
- Milchpackung aufmachen.
- Milchpackung wegwerfen.
- Milch in Heissgetränk leeren.
- Milch kaufen.

(Mehrfachverwendungen sind möglich.)

## Block 8: Fallunterscheidungen

In C werden Fallunterscheidungen mit dem **if**-Statement umgesetzt:

```
1 void setup()
2 {
3     pinMode(15, INPUT);
4     pinMode(16, INPUT);
5     pinMode(17, INPUT);
6
7     pinMode(3, OUTPUT);
8     pinMode(4, OUTPUT);
9
10    // enable pullup resistors
11    digitalWrite(15, 1);
12    digitalWrite(16, 1);
13    digitalWrite(17, 1);
14 }
15
16 void loop()
17 {
18     int valueA, valueB;
19     int doSwitch = digitalRead(17);
20
21     if (doSwitch)
22     {
```

```
23     valueA = digitalRead(15);
24     valueB = digitalRead(16);
25 }
26 else
27 {
28     valueA = digitalRead(16);
29     valueB = digitalRead(15);
30 }
31
32 digitalWrite(3, valueA);
33 digitalWrite(4, valueB);
34 }
```

block08\_if.pde

Hierbei werden mehrere Statements zu Blöcken zusammengefasst, die von geschwungenen Klammern eingeschlossen sind. Die Einrückungen dienen der besseren Lesbarkeit.

Die Bedingung, die dem **if**-Statement in runden Klammern mitgegeben wird, ist vom Datentyp **int**. Ein Zahlenwert ungleich 0 führt zur Ausführung des Blocks oder Statements unmittelbar nach dem **if**-Statement. Ein Zahlenwert von 0 führt zur Ausführung des **else**-Zweiges, sofern vorhanden.

Wenn ein Zahlenwert auf diese Weise interpretiert wird, nennt man ihn Wahrheitswert. Ein Zahlenwert von ungleich 0 wird *Wahr* und ein Zahlenwert von 0 wird *Falsch* genannt. Ausdrücke, die auf Wahrheitswerte führen werden *boolesche Ausdrücke* genannt.

### Diskussion:

Wie lautet die Syntax des **if**-Statements?

Wann kann man die geschwungenen Klammern weglassen?

### Übung:

Schreiben Sie ein Programm, mit dem Sie feststellen können ob die Bindung des **else**-Statements, so wie im linken oder so wie im rechten Listing angedeutet ist, funktioniert:

```
if (foo)
  if (bar)
    a = 1;
  else
    a = 2;

if (foo)
  if (bar)
    a = 1;
  else
    a = 2;
```

### Diskussion:

Warum ist es sinnvoll und wichtig, richtig einzurücken?

## Block 9: Boolesche Ausdrücke

Wahrheitswerte können durch den Vergleich von Zahlenwerten gebildet werden:

```
3 int getFlashDurationMs()
4 {
5     int durationSeconds =
6         consoleReadDecimal("Wieviele Sekunden? ");
7
8     if (durationSeconds < 0)
9         durationSeconds = 0;
10    else if (durationSeconds > 10)
11        durationSeconds = 10;
12
13    return durationSeconds * 1000;
14 }
```

block09\_rel.pde

Dabei können folgende Operatoren verwendet werden:

<code>a &lt; b</code>	a kleiner b
<code>a &lt;= b</code>	a kleiner oder gleich b
<code>a == b</code>	a gleich b
<code>a != b</code>	a ungleich b
<code>a &gt;= b</code>	a größer oder gleich b
<code>a &gt; b</code>	a größer b

Wahrheitswerte können auch durch Verknüpfungen von mehreren Wahrheitswerten gebildet werden:

```
1 void setup()
2 {
3   pinMode(15, INPUT);
4   pinMode(16, INPUT);
5   pinMode(3, OUTPUT);
6   digitalWrite(15, 1); // enable pullup resistor
7   digitalWrite(16, 1); // enable pullup resistor
8 }
9
10 void loop()
11 {
12   if (!digitalRead(15) && !digitalRead(16))
13     digitalWrite(3, 1);
14   else
15     digitalWrite(3, 0);
16 }
```

block09\_logic.pde

Dabei können folgende Operatoren verwendet werden:

<code>a &amp;&amp; b</code>	a und b
<code>a    b</code>	a oder b
<code>!a</code>	nicht a

### Diskussion:

Wie interpretiert der C-Compiler folgenden Ausdruck?

`a < b < c`

Wie kann man diesen Ausdruck so umschreiben, dass er die in der Mathematik übliche Bedeutung hat?

### Übung:

Schreiben Sie das Programm `block09_logic.pde` so um, dass die LED leuchtet, wenn mindestens eine der Tasten gedrückt ist.

### Diskussion:

Wie müsste das Programm verändert werden, sodass die LED leuchtet, wenn genau eine der Tasten gedrückt ist?

### Diskussion:

Erklären Sie Zweck und Funktionsprinzip des folgenden Programmes:

```
1 void setup()
2 {
3   pinMode(15, INPUT);
4   pinMode(16, INPUT);
5   pinMode(3, OUTPUT); // red LED
6   pinMode(4, OUTPUT); // green LED
7   digitalWrite(15, 1); // enable pullup resistor
8   digitalWrite(16, 1); // enable pullup resistor
9 }
10
11 void loop()
12 {
13   if (!digitalRead(15) || !digitalRead(16))
14   {
15     delay(100);
16     if (!digitalRead(15) && !digitalRead(16))
17       digitalWrite(4, 1);
18     else
19       digitalWrite(3, 1);
20     delay(1000);
21   }
22   digitalWrite(3, 0);
23   digitalWrite(4, 0);
24 }
```

block09\_sync.pde

## Block 10: Schleifen

Oft ist es sinnvoll einen Programmblock mehrmals hintereinander auszuführen. Dazu verwendet man Schleifen. Eine Schleife besteht aus einer Bedingung die bestimmt wie lange (oft) die Schleife laufen soll und aus dem Schleifenkörper. Im Struktogramm sieht das so aus:

summe ← 0
input ← Eingabe()
<i>solange</i> input ≠ 0
summe ← summe + input
input ← Eingabe()
Ausgabe(summe)

In C werden Schleifen mit dem **while**-Statement umgesetzt:

```
1  #include <Console.h>
2
3  void setup()
4  {
5      consoleInit(9600);
6  }
7
8  void loop()
9  {
10     int summe = 0;
11     int input = consoleReadDecimal("Erster Summand: ");
12     while (input != 0)
13     {
14         summe = summe + input;
15         input = consoleReadDecimal("Weiterer Summand: ");
16     }
17     consolePrintf("Die Summe ist %d\n", summe);
18 }
```

block10\_loop.pde

Das **while**-Statement ist dem **if**-Statement ähnlich. Der einzige Unterschied besteht darin, dass das **while**-Statement immer wieder ausgeführt wird solange die Bedingung erfüllt ist.

## Übung:

Wie ist die Syntax des **while**-Statements?

Wann kann man die geschwungenen Klammern weglassen?

### Diskussion:

Es gibt eine übergeordnete Funktion, die `setup()` und `while()` aufruft. Wie könnte diese Funktion aussehen?

### Übung:

Erweitern Sie `setup()` in `block04_blink1.pde`, sodass das Programm erst startet wenn die Taste an Pin 2 gedrückt wird.

### Diskussion:

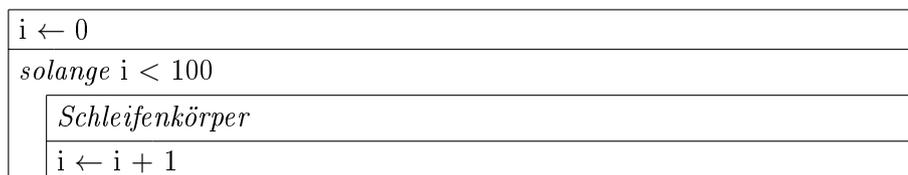
Bei der `while()`-Schleife wird die Laufbedingung vor dem Ausführen des Schleifenkörpers geprüft. Bei der `do-while`-Schleife wird die Laufbedingung erst nach dem Schleifenkörper geprüft, sodass der Schleifenkörper mindestens ein mal ausgeführt wird:

```
int input;
do {
    blink();
    input = consoleReadDecimal("Nochmal blinken? (1/0) ");
} while (input);
```

In welchen Fällen ist diese Form der Schleife sinnvoller?

## Block 11: Zählerschleifen

Die meisten Schleifen sind Zählerschleifen. Zählerschleifen zählen eine Variable von einem Startwert zu einem Endwert:



In C gibt es dafür die `for`-Schleife:

```
int i;

for (i = 0; i < 100; i = i + 1)
```

```
{  
    //Schleifenkoerper  
}
```

In einem umfangreicheren Programm verwendet sieht das dann so aus:

```
1 void blinkPin(int pin)  
2 {  
3     digitalWrite(pin, 1);  
4     delay(1000);  
5     digitalWrite(pin, 0);  
6     delay(100);  
7 }  
8  
9 void setup()  
10 {  
11     pinMode(3, OUTPUT);  
12     pinMode(4, OUTPUT);  
13     pinMode(5, OUTPUT);  
14 }  
15  
16 void loop()  
17 {  
18     int i,j;  
19  
20     for (i = 3; i <= 5; i = i + 1)  
21     {  
22         for (j = 0; j < 3; j = j + 1)  
23             blinkPin(i);  
24     }  
25 }
```

block11\_blink.pde

### Diskussion:

Wie ist die Syntax des **for**-Statements?

Was sind die Vor- und Nachteile gegenüber der **while**-Schleife?

### Übung:

Schreiben Sie ein Programm, das auf der seriellen Konsole das kleine  $1 \times 1$  ausgibt.

Für Experten: Modifizieren Sie das Programm so, dass doppelte Nennungen von Paaren von Faktoren vermieden werden.

### Diskussion:

Wie kann man in C eine Zählerschleife realisieren die statt hinauf herunter zählt?

## Block 12: Pre- und Postinkrement

U.a. in Zählerschleifen findet man häufig folgendes Konstrukt:

```
i = i + 1;
```

Das kann man in C mit dem Befehl

```
++i;
```

abkürzen. Wenn man `++i` als Ausdruck verwendet, dann evaluiert dieser Ausdruck auf den neuen Wert von `i`.

Der Ausdruck `i++` zählt ebenfalls den Wert von `i` um 1 hoch, evaluiert aber auf den alten Wert.

Analog dazu Zählen die Ausdrücke `--i` und `i--` den Wert von `i` um 1 herunter und evaluieren auf den neuen bzw. alten Wert von `i`.

### Experiment:

Überlegen Sie welche Ausgabe folgendes Programm liefert:

```
1 #include <Console.h>
2
3 void setup()
4 {
5     int i = 0;
6     consoleInit(9600);
7     consolePrintf("i == %d\n", i);
8     consolePrintf("++i == %d\n", ++i);
9     consolePrintf("i == %d\n", i);
10    consolePrintf("i++ == %d\n", i++);
11    consolePrintf("i == %d\n", i);
```

```
12 }
13
14 void loop()
15 {
16 }
```

block12\_inc.pde

Überprüfen Sie Ihre Vorhersage.

### Diskussion:

Vergleichen Sie folgende Programmfragmente:

```
for (i=0; i<10; i++)
    consolePrintf("%d\n", i);
```

```
for (i=0; i<10; ++i)
    consolePrintf("%d\n", i);
```

Gibt es einen Unterschied im Verhalten dieser beiden Programmfragmente?

### Diskussion:

Was ist das Problem mit folgendem Programmcode?

```
i = i + i++;
```

## Block 13: Einfache Datentypen

Neben dem Datentyp `int` gibt es in C u.A. folgende Integer Datentypen:

Datentyp	Wertebereich
<code>int8_t</code>	-128 ... 127
<code>int16_t</code>	-32 768 ... 32 767
<code>int32_t</code>	-2 147 483 648 ... 2 147 483 647
<code>uint8_t</code>	0 ... 255
<code>uint16_t</code>	0 ... 65 535
<code>uint32_t</code>	0 ... 4 294 967 295

Neben den Integer Datentypen, in denen ganze Zahlen gespeichert werden können, gibt es noch Fließkommadatentypen:

Datentyp	Genauigkeit
<b>float</b>	6 Dezimalstellen
<b>double</b>	15 Dezimalstellen

Wenn mit einem arithmetischen Operator ein Integer- und ein Fließkommawert miteinander verknüpft werden, so wird der Integerwert vor der Operation automatisch in einen Fließkommawert umgewandelt.

**Diskussion:**

Kleinere Datentypen brauchen weniger Platz und Berechnungen mit ihnen sind schneller.

Unter welchen Umständen ist es daher sinnvoll in Ihrem Programm möglichst kleine Datentypen zu verwenden und wann ist es der Mühe nicht wert?

**Diskussion:**

Was sind Fließkommazahlen? Wie ist die Genauigkeit zu verstehen?

**Diskussion:**

Speziell auf einem Mikrokontroller sind Berechnungen mit Fließkommazahlen sehr langsam. Wann muss man sie dennoch verwenden?

## Block 14: Arrays

Oft ist es sinnvoll einzelne Speicherstellen aus einem zusammenhängenden Block von Speicherstellen über einen Index anzusprechen. Solche Blöcke nennt man Arrays. In C werden Arrays wie Variablen definiert, nur dass dem Arraynamen die Anzahl der Elemente des Arrays in eckigen Klammern folgt. z.B.:

```
int a[3];
```

Die Arrayelemente werden von 0 beginnend durchnummeriert. Um ein Element anzusprechen verwendet man den Arraynamen gefolgt vom angesprochenen Index in eckigen Klammern. Dieser Ausdruck kann wie ein Variablenname verwendet werden:

```
a[0] = 0;
a[1] = 1;
a[2] = 2;
```

In den eckigen Klammern kann jeder beliebige Ausdruck stehen der auf eine Zahl evaluiert:

```
for (i=0; i<3; i++)
    a[i] = i;
```

### Diskussion:

Wie lautet die genaue Syntax zum definieren und ansprechen von Arrays?

Was ist der erlaubte Wertebereich für den Index? Was passiert, wenn dieser nicht eingehalten wird?

### Diskussion:

Erklären Sie die Funktionsweise des folgenden Programms:

```
1  #include <Console.h>
2
3  int pattern[10];
4
5  void blinkPin(int pin)
6  {
7      digitalWrite(pin, 1);
8      delay(1000);
9      digitalWrite(pin, 0);
10     delay(100);
11 }
12
13 void setup()
14 {
15     int i;
16     pinMode(3, OUTPUT);
17     pinMode(4, OUTPUT);
18     pinMode(5, OUTPUT);
19     consoleInit(9600);
20
21     for (i=0; i < 10; i++)
22     {
23         consolePrintf("LED fuer Schritt %d: ", i);
```

```
24     pattern[i] = consoleReadDecimal("");
25     }
26 }
27
28 void loop()
29 {
30     int i;
31     for (i = 0; i < 10; i++)
32         blinkPin(pattern[i]);
33
34     delay(2000);
35 }
```

block14\_blinkpattern.pde

### Übung:

Schreiben Sie eine Datenbank, die beim Programmstart 10 Zahlen vom Benutzer einliest und dem Benutzer dann in einer Endlosschleife die Möglichkeit gibt die Zahlen beliebig Auszulesen.

## Block 15: Mehr zu Arrays

In C gibt es sogenannte mehrdimensionale Arrays. Das sind Arrays mit zwei oder mehr Indizes. Man kann sie sich als Arrays von Arrays vorstellen. Zweidimensionale Arrays können als Tabelle veranschaulicht werden, wobei die Indizes Zeilen- und Spalten-Nummern darstellen:

```
int i, j, a[3][4];
for (i=0; i<3; i++)
for (j=0; j<4; j++)
    a[i][j] = 10*i + j;
```

In C gibt es eine eigene Syntax um ein Array bei der Definition mit Werten zu initialisieren:

```
int a[3][4] = {
    { 0, 1, 2, 3 },
    { 10, 11, 12, 13 },
    { 20, 21, 22, 23 }
};
```

Wenn ein Array bei der Definition initialisiert wird kann die Größenangabe bei der ersten Dimension auch weggelassen werden:

```
int a[][4] = {
    { 0, 1, 2, 3 },
    { 10, 11, 12, 13 },
    { 20, 21, 22, 23 }
};
```

**Diskussion:**

Wie kann man dreidimensionale Arrays veranschaulichen?

Wfür kann man mehrdimensionale Arrays brauchen?

**Diskussion:**

Erklären Sie die Funktionsweise des folgenden Programms:

```
1 int pattern[2][2][2] = {
2     { { 0, 1 }, { 0, 0 } },
3     { { 0, 0 }, { 2, 0 } }
4 };
5
6 void setup()
7 {
8     pinMode(15, INPUT);
9     pinMode(16, INPUT);
10    pinMode(17, INPUT);
11    pinMode(3, OUTPUT);
12    digitalWrite(15, 1); // enable pullup resistor
13    digitalWrite(16, 1); // enable pullup resistor
14    digitalWrite(17, 1); // enable pullup resistor
15 }
16
17 void loop()
18 {
19     int key1 = !digitalRead(15);
20     int key2 = !digitalRead(16);
21     int key3 = !digitalRead(17);
22     int command = pattern[key1][key2][key3];
23
24     if (command == 1)
25         digitalWrite(3, 1);
26
27     if (command == 2)
28         digitalWrite(3, 0);
29 }
```

block15\_combination.pde

### Übung:

Verändern Sie das Programm so, dass folgende Codes zusätzlich zu den bestehenden codes zum Ein- und Ausschalten der LED verwendet werden können:

zum Einschalten: Ein, Aus, Ein  
zum Ausschalten: Aus, Ein, Ein

## Block 16: Zusammengesetzte Datentypen

Eine Art von zusammengesetzten Datentypen sind Arrays. Bei Arrays haben alle Elemente den selben Typ und die Elemente sind durchnummeriert. Ein weiteres Beispiel für zusammengesetzte Datentypen sind Structs. Bei Structs können die Elemente voneinander verschiedene Datentypen haben und die Elemente werden über Namen angesprochen:

```
struct {  
    float f;  
    int i;  
} s;  
  
s.i = 42;  
s.f = s.i / 23.0;
```

Praktisch immer möchte man in einem Programm mehrere Struct-Variablen mit gleicher Felddefinition verwenden. Dazu kann man Namen für Struct-Typen definieren:

```
struct fi {  
    float f;  
    int i;  
};  
  
struct fi s1, s2;  
  
s1.i = 42;  
s1.f = s1.i / 23.0;  
s2 = s1;
```

Zuweisungen zwischen Structs sind nur möglich, wenn die Structs mit dem selben Struct-Typen erstellt wurden.

Wie bei Arrays können Structs bei der Definition mit Werten initialisiert werden:

```
struct fi s3 = { 1.5, 2 };
```

### Diskussion:

Erklären Sie das Funktionsprinzip dieses Programms:

```
1 struct note { int freq; int duration; };
2
3 struct note melody[4] = {
4     { 440, 1000 },
5     { 880, 1000 },
6     { 440, 1000 },
7     { 880, 1000 }
8 };
9
10 void setup()
11 {
12 }
13
14 void playNote(struct note n)
15 {
16     tone(6, n.freq);
17     delay(n.duration);
18     noTone(6);
19 }
20
21 void loop()
22 {
23     int i;
24
25     for (i=0; i < 4; i++)
26         playNote(melody[i]);
27
28     delay(5000);
29 }
```

block16\_song.pde

### Diskussion:

Erklären Sie das Funktionsprinzip dieses Programms:

```
1 struct state
2 {
3     bool led1, led2;
4     int targetStates[2][2];
5 };
6
7 struct state states[] = {
8     { 0, 0, { { 0, 1 }, { 2, 0 } } },
9     { 0, 1, { { 1, 1 }, { 1, 3 } } },
10    { 1, 0, { { 2, 2 }, { 2, 3 } } },
11    { 1, 1, { { 0, 3 }, { 3, 3 } } }
12 };
13
14 int currentState = 0;
15
16 void setup()
17 {
18     pinMode(3, OUTPUT);
19     pinMode(4, OUTPUT);
20     pinMode(15, INPUT);
21     pinMode(16, INPUT);
22     digitalWrite(15, 1); // enable pullup resistor
23     digitalWrite(16, 1); // enable pullup resistor
24 }
25
26 void loop()
27 {
28     digitalWrite(3, states[currentState].led1);
29     digitalWrite(4, states[currentState].led2);
30
31     bool b1, b2;
32     b1 = !digitalRead(15);
33     b2 = !digitalRead(16);
34
35     currentState = states[currentState].targetStates[b1][b2];
36 }
```

block16\_fsm.pde

## Block 17: Pointer

Ein Pointer ist eine Variable deren Wert ein Verweis auf eine andere Variable ist. Ein Pointer wird genauso wie eine Variable des Typs auf den er zeigt definiert, nur dass zusätzlich ein (weiterer) Stern vor den Variablennamen gestellt wird. Um einen Pointer-wert zu einer Variable zu bilden wird ein

Ampersent (&) vor den Variablennamen gestellt. z.B.:

```
int v = 5, *p = &v;
```

Um auf die Variable zuzugreifen auf die der Pointer zeigt wird ein Stern vor die Pointervariable gesetzt. z.B.:

```
*p = *p * *p;  
consolePrintf("v = %d, *p = %d\n", v, *p);
```

Um auf Elemente eines structs zuzugreifen kann man entweder ebenfalls den Stern-Operator verwenden. Schoener ist es jedoch den eigens dafür vorgesehenen Pfeil-Operator (->) zu verwenden:

```
struct foobar { int foo, bar; } fb;  
struct foobar *p = &fb.  
(*p).foo = 23;  
p->bar = 42;
```

### Diskussion:

Erklären Sie das Funktionsprinzip des folgenden Programms:

```
1 struct note { int freq; int duration; };  
2  
3 struct note melody[64] = {  
4     { 440, 1000 },  
5     { 880, 1000 },  
6     { 440, 1000 },  
7     { 880, 1000 }  
8 };  
9  
10 void setup()  
11 {  
12     consoleInit(9600);  
13 }  
14  
15 void editMelody(struct note *n)  
16 {  
17     n->freq = consoleReadDecimal("Frequenz (Hz): ");  
18     n->duration = consoleReadDecimal("Laenge (ms): ");  
19 }  
20  
21 void playNote(struct note n)  
22 {  
23     tone(8, n.freq);  
24     delay(n.duration);
```

```
25     noTone(8);
26 }
27
28 void playMelody()
29 {
30     int i;
31     for (i = 0; i < 64; i++)
32         playNote(melody[i]);
33 }
34
35 void loop()
36 {
37     int i;
38
39     i = consoleReadDecimal("Welche Note wollen sie editieren?");
40
41     if (i < 64)
42         editNote(&melody[i]);
43
44     playMelody();
45 }
```

block17\_composer.pde

### Übung:

Erweitern Sie das Programm `block16_fsm.pde` um eine Editierfunktion für die State-Tabelle.

### Diskussion:

Wie hätte man die beiden Beispiele (composer und fsm mit Editor) ohne Einsatz von Pointern lösen können?

In welchen Fällen kommt man um den Einsatz von Pointern nicht herum?

### Diskussion:

Ein Pointer kann auch auf "Nichts" zeigen. Dazu wird dem Pointer der spezielle Wert `NULL` zugewiesen. Ein Pointer mit dem Wert `NULL` evaluiert als boolescher Ausdruck auf falsch. Jeder andere Pointer evaluiert als boolescher Ausdruck auf wahr.

Ein Pointer mit dem Wert `NULL` kann nur als boolescher Ausdruck verwendet und mit anderen Pointern verglichen werden. Man darf einen solchen Pointer aber niemals mit `*` oder `->` auflösen.

Wozu kann man diesen speziellen Wert `NULL` für Pointer brauchen?

## Block 18: Pointer auf Arrays

Man kann einen Pointer auf das erste Element eines Arrays wie folgt bilden:

```
int a[3];
int *p = &a[0];
```

In C gibt es jedoch auch folgende einfachere Schreibweise:

```
int *p = &a;
```

Oder noch einfacher:

```
int *p = a;
```

Auf das erste Element des Arrays kann mit dem Pointer wie gehabt zugegriffen werden:

```
*p = 0;
```

Man kann aus einem Pointer auf ein Arrayelement einen Pointer auf ein anderes Arrayelement bilden, indem man die Differenz der Indizes der Arrayelemente zum ersten Pointer hinzuaddiert:

```
*(p + 1) = 1; // äquivalent zu a[1] = 1;
```

Der `[]`-Operator dereferenziert allgemein die Summe aus einem Pointer und einem Integer:

```
p[2] = 2; // äquivalent zu a[2] = 2;
```

### Diskussion:

Das Rechnen mit Pointern nennt man Pointerarithmetik. Welche Operationen sind in diesem Zusammenhang sinnvoll? Welche sind unsinnig?

### Diskussion:

Was passiert, wenn man mit der Pointerarithmetik die Grenzen der Arrays verlässt?

### Übung:

Erklären Sie das Funktionsprinzip des folgenden Programms:

```
1  struct note { int freq; int duration; };
2
3  struct note melodyA[] = {
4      { 440, 1000 },
5      { 880, 1000 },
6      { 440, 1000 },
7      { 880, 1000 },
8      { 0, 0 }
9  };
10
11 struct note melodyB[] = {
12     { 440, 1000 },
13     { 880, 1000 },
14     { 440, 1000 },
15     { 880, 1000 },
16     { 0, 0 }
17 };
18
19 struct note melodyC[] = {
20     { 440, 1000 },
21     { 880, 1000 },
22     { 440, 1000 },
23     { 880, 1000 },
24     { 0, 0 }
25 };
26
27 void setup()
28 {
29     pinMode(2, INPUT);
30     pinMode(3, INPUT);
31     pinMode(4, INPUT);
32 }
33
34 void playNote(struct note *n)
35 {
36     tone(8, n->freq);
37     delay(n->duration);
```

```
38     noTone(8);
39 }
40
41 void playMelody(struct note *melody)
42 {
43     while(melody->duration)
44     {
45         playNote(melody);
46         melody++;
47     }
48 }
49
50 void loop()
51 {
52     if (digitalRead(2))
53         playMelody(melodyA);
54     else if (digitalRead(3))
55         playMelody(melodyB);
56     else if (digitalRead(4))
57         playMelody(melodyC);
58 }
```

block18\_multiplay.pde

## Block 19: Textzeichen

Textzeichen werden im Computer als Ganzzahlen gespeichert. Die Zuordnung zwischen Textzeichen und Zahlen wird Encoding genannt. In der Microcontroller-Programmierung findet fast ausschliesslich das ASCII-Encoding Verwendung:

Block 19: Textzeichen

Zahl	Zeichen	Zahl	Zeichen	Zahl	Zeichen	Zahl	Zeichen
0	NUL	32	SP	64	@	96	'
1	SOH	33	!	65	A	97	a
2	STX	34	"	66	B	98	b
3	ETX	35	#	67	C	99	c
4	EOT	36	\$	68	D	100	d
5	ENQ	37	%	69	E	101	e
6	ACK	38	&	70	F	102	f
7	BEL	39	'	71	G	103	g
8	BS	40	(	72	H	104	h
9	TAB	41	)	73	I	105	i
10	LF	42	*	74	J	106	j
11	VT	43	+	75	K	107	k
12	FF	44	,	76	L	108	l
13	CR	45	-	77	M	109	m
14	SO	46	.	78	N	110	n
15	SI	47	/	79	O	111	o
16	DLE	48	0	80	P	112	p
17	DC1	49	1	81	Q	113	q
18	DC2	50	2	82	R	114	r
19	DC3	51	3	83	S	115	s
20	DC4	52	4	84	T	116	t
21	NAK	53	5	85	U	117	u
22	SYN	54	6	86	V	118	v
23	ETB	55	7	87	W	119	w
24	CAN	56	8	88	X	120	x
25	EM	57	9	89	Y	121	y
26	SUB	58	:	90	Z	122	z
27	ESC	59	;	91	[	123	{
28	FS	60	«	92	\	124	
29	GS	61	=	93	]	125	}
30	RS	62	»	94	^	126	-
31	US	63	?	95	_	127	DEL

Das ASCII-Encoding hat folgende interessante Eigenschaften:

- Es sind 128 (7 Bit) Zeichen definiert.
- Die ersten 32 sowie das letzte Zeichen sind nicht druckbare Steuerzeichen.
- Die Ziffernzeichen sowie Klein- und Großbuchstaben sind jew. in zusammenhängenden Blöcken aufsteigend abgelegt.

In C verwendet man zum Speichern eines solchen Textzeichens den Datentyp

`char`, der den selben Wertebereich wie `int8_t` hat. In C kann man statt dem Zahlenwert eines Zeichens auch das Zeichen selbst in einfachen Anführungszeichen schreiben. z.B.:

```
char ch = 'a'; // 97
```

Um nicht-druckbare Zeichen sowie Anführungszeichen und Backslashes in Anführungszeichen verwenden zu können gibt es folgende Escape-Sequenzen:

Escape-Sequenz	Zahl	Zeichen
<code>\b</code>	8	BS
<code>\n</code>	10	LF
<code>\r</code>	13	CR
<code>\"</code>	34	"
<code>\'</code>	39	'
<code>\\</code>	92	\

### Diskussion:

Das folgende Programm gibt eine ASCII-Tabelle aus. Diskutieren Sie die Rolle des Platzhalters `%c`. Wie ist der Unterschied zum Platzhalter `%d`?

```

1 void setup()
2 {
3   consoleInit(9600);
4
5   for (char c=32; c < 127; c++)
6     consolePrintf("%d - %c\n", c, c);
7 }
8
9 void loop()
10 {
11 }
```

block19\_ascii.pde

### Diskussion:

Erklären Sie das Funktionsprinzip des folgenden Programms.

```

1 void setup()
2 {
3   consoleInit(9600);
```

```
4
5     for (char c=32; c < 127; c++)
6         printf("%d - %c\n", c, c);
7 }
8
9 void loop()
10 {
11     char c;
12     printf("Geben Sie ein Zeichen ein:");
13     do {
14         c = getch();
15     } while(c < 32);
16
17     printf("%c\n", c);
18
19     if ('a' <= c && c <= 'z')
20         printf("Kleinbuchstabe\n");
21     else if ('A' <= c && c <= 'Z')
22         printf("Grossbuchstabe\n");
23     else if ('0' <= c && c <= '9')
24         printf("Ziffer\n");
25     else
26         printf("Sonderzeichen\n");
27 }
```

block19\_charclass.pde

## Block 20: Strings (Zeichenketten)

Strings werden in C als Arrays von Zeichen gespeichert. Das Ende des Strings wird durch den Zahlenwert 0 repräsentiert:

```
char txt[] = { 'T', 'e', 's', 't', 0 };
printf(txt);
```

Dafür gibt es in C auch eine kürzere Schreibweise mit doppelten Anführungszeichen:

```
char txt[] = "Test";
printf(txt);
```

Die abschliessende Null wird in dieser Notation automatisch hinzugefügt.

Diese Notation kann auch überall dort verwendet werden wo ein Ausdruck vom Typ `char*` akzeptiert wird:

```
printf("Test");
```

Um Strings zeichenweise abzuarbeiten wird i.d.R. eine **for**-Schleife verwendet die als Laufbedingung auf die abschliessende Null prüft:

```
1 void myConsolePrint(char *str)
2 {
3     int i;
4     for (i = 0; str[i]; i++)
5         consolePutChar(str[i]);
6 }
7
8 void setup()
9 {
10    consoleInit(9600);
11    myConsolePrint("Hello World!\n");
12 }
13
14 void loop()
15 {
16     // Hier passiert nichts.
17 }
```

block20\_print.pde

### Diskussion:

Erklären Sie das Funktionsprinzip des folgenden Programms:

```
1 void stringAppend(char *a, char *b)
2 {
3     int i, j;
4
5     for (i = 0; a[i]; i++);
6     for (j = 0; b[j]; j++)
7         a[i+j] = b[j];
8     a[i+j] = 0;
9 }
10
11 void setup()
12 {
13     char buffer[100] = "Hallo";
14     consoleInit(9600);
15
16     stringAppend(buffer, " ");
17     stringAppend(buffer, "Welt!");
18     stringAppend(buffer, "\n");
19     consolePrint(buffer);
20 }
```

```
20 }
21
22 void loop()
23 {
24     // Hier passiert nichts.
25 }
```

block20\_append.pde

Was passiert bei `stringAppend()` wenn das Zielarray zu klein ist?

### Übung:

Schreiben Sie eine Funktion die alle Grossbuchstaben in einem String durch Kleinbuchstaben ersetzt.

## Block 21: break, continue und goto

Das **break**-Statement beendet die innerste Schleife unabhängig von der Laufbedingung:

```
while (keep_running) {
    do_something();
    if (stop_running)
        break;
    do_some_more();
}
```

Das **continue**-Statement beendet den Schleifenkörper der innersten Schleife:

```
for (i = 0; i < 20; i++) {
    if (i == 13)
        continue;
    consolePrintf("%d\n", i);
}
```

Um aus einer Schleife auszubrechen die nicht die innerste Schleife ist wird das **goto**-Statement verwendet. Dazu muss das Sprungziel explizit im Code definiert werden:

```
int list[5] = { 1, 2, 3, 2, 5 };
int i, j;
for (i = 0; i < 5; i++)
    for (j = i+1; j < 5; j++)
```

```
        goto found;
consolePrint("Keine Dubletten gefunden.\n");
if (0)
found:
    consolePrintf("Dublette bei %d/%d.\n", i, j);
```

### Diskussion:

Man kann eine Funktion jederzeit mit einem **return** beenden. Bei Funktionen ohne Rückgabewert wird dieser beim **return**-Statement weggelassen.

In welchen Fällen kann die Funktionalität von **break**, **continue** und **goto** mit einem **return** nachgebaut werden? Was sind die Vor- und Nachteile?

### Diskussion:

In welchen Fällen kann man die Funktionalität von **break**, **continue** und **goto** mit **bool**-Variablen nachbauen? Was sind die Vor- und Nachteile?

### Diskussion:

Wie kann man das **break**-Statement verwenden um eine Schleife zu konstruieren die die Laufbedingung in der Mitte des Schleifenkörpers hat?

## Block 22: Das switch-Statement

Wenn man Code in Abhängigkeit von einem Integer-Wert ausführen möchte, kann man natürlich eine **if-else**-Kaskade verwenden:

```
if (n == 0)
    consolePrint("keines\n");
else if (n == 1)
    consolePrint("eines\n");
else if (n == 2 || n == 3)
    consolePrint("wenige\n");
else
    consolePrint("viele\n");
```

Wenn – wie in diesem Beispiel – nur auf Gleichheit geprüft wird, kann man statt dessen auch das **switch**-Statement verwenden:

```
switch(n)
{
case 0:
    consolePrint("keines\n");
    break;

case 1:
    consolePrint("eines\n");
    break;

case 2:
case 3:
    consolePrint("wenige\n");
    break;

default:
    consolePrint("viele\n");
}
```

Dabei verhalten sich die **case**-Labels und das **default**-Label ähnlich wie **goto**-Labels, wobei das **switch**-Statement entscheidet welches dieser Labels angesprungen werden soll.

Falls kein **default**-Label angegeben wurde, wird an das Ende des **switch**-Blocks gesprungen. Ebenso bewirkt ein **break**-Statement innerhalb des **switch**-Blocks dass an das Ende des **switch**-Blocks gesprungen wird.

### Diskussion:

Wann muss man in einem **switch**-Block ein **break** verwenden? Was passiert, wenn man es weglässt?

### Diskussion:

In welchen Fällen sollte man eine **if-else**-Kaskade und wann ein **switch**-Statement verwenden?

### Übung:

Schreiben Sie eine Funktion die zu einem übergebenem Zeichen ermittelt, ob es sich um ein Interpunktionszeichen, Selbstlaut oder sonstiges Zeichen handelt! Erzeugen Sie eine Entsprechende Ausgabe auf der seriellen Konsole.

### Übung für Fortgeschrittene:

Schreiben Sie eine Funktion die zwischen Interpunktionszeichen, kleinen Selbstlauten, grossen Selbstlauten, kleinen Konsonanten, grossen Konsonanten und sonstigen Zeichen unterscheidet.

## Block 23: Binärdarstellung von Ganzzahlen

(Dieser Block ist im Wesentlichen eine Kopie des Block 81 des Skriptum zum Metalab Elektronik-Kurs.)

Unser „normales“ Zahlensystem – das Dezimalsystem – ist ein sogenanntes Stellenwertsystem zur Basis 10: Wir unterscheiden 10 verschiedene Ziffernzeichen, wobei der „Wert“ eines Ziffernzeichens nicht nur vom Zeichen selbst, sondern vor allem von seiner Stelle innerhalb der Zahl abhängt.

Beispiel:

$$290 = 2 \cdot 10^2 + 9 \cdot 10^1 + 0 \cdot 10^0 = 200 + 90 + 0$$

Das heißt, jedes Ziffernzeichen hat einen „Grundwert“. Dieser wird jedoch noch mit  $B^{k-1}$  multipliziert, wobei  $B$  die Anzahl der verschiedenen Ziffernzeichen und  $k$  die Stelle der Ziffer innerhalb der Zahl angibt.

Im Falle von Binärzahlen beschränkt man sich auf  $B = 2$  Zeichen (die Ziffern 0 und 1).

Beispiel:

$$100100010_2 = 1 \cdot 2^8 + 1 \cdot 2^5 + 1 \cdot 2^1 = 256 + 32 + 2 = 290$$

Da digitale Signale zwei Zustände besitzen, eignet sich ein Bus mit  $n$  Bit Breite zum Übertragen einer  $n$ -stelligen Binärzahl.

### Übung:

Rechnen Sie die Binärzahl  $100100_2$  ins Dezimalsystem um.

Rechnen Sie die Dezimalzahl  $112_{10}$  ins Binärsystem um.

Als Hilfestellung zeigt die folgende Tabelle die Werte von  $2^n$  für die ersten 18  $n \in \mathbb{N}_0$ :

$2^0 = 1$	$2^8 = 256$
$2^1 = 2$	$2^9 = 512$
$2^2 = 4$	$2^{10} = 1024$
$2^3 = 8$	$2^{11} = 2048$
$2^4 = 16$	$2^{12} = 4096$
$2^5 = 32$	$2^{13} = 8192$
$2^6 = 64$	$2^{14} = 16384$
$2^7 = 128$	$2^{15} = 32768$

**Diskussion:**

Was ist die dezimale Darstellung von  $11111111_2$ ? Wie kann man diese Darstellung ermitteln, ohne viel rechnen zu müssen?

**Diskussion:**

Bei Vorzeichenbehafteten Datentypen haben die einzelnen Bits die gleiche Wertigkeit. Lediglich das höchstwertigste Bit hat ein negatives Vorzeichen. z.B. bei 8 Bit breiten Datentypen:

Bit	7	6	5	4	3	2	1	0
Wertigkeit vorzeichenlos	128	64	32	16	8	4	2	1
Wertigkeit vorzeichenbehaftet	-128	64	32	16	8	4	2	1

Wie ergeben sich die Wertebereiche der Integer Datentypen? (Siehe Tabelle Block 13.)

## Block 24: Hexadezimal- und Oktalzahlen

Zwei weitere für das Programmieren relevante Stellenwertsysteme sind die Hexadezimalzahlen und die Oktalzahlen.

Das Hexadezimalsystem ist ein Stellenwertsystem zur Basis 16, wobei für die Ziffern mit den Wertigkeiten 10 bis 15 die Buchstaben A bis F verwendet

werden. Das Hexadezimalsystem ist deshalb von Interesse, weil eine Hexadezimale Ziffer genau 4 binären Ziffern entspricht:

Hexadezimal	Binär	Hexadezimal	Binär
0	0000	8	1000
1	0001	9	1001
2	0010	A	1010
3	0011	B	1011
4	0100	C	1100
5	0101	D	1101
6	0110	E	1110
7	0111	F	1111

Das Oktalsystem ist ein Stellenwertsystem zur Basis 8. Das Oktalsystem ist deshalb von Interesse, weil eine Oktale Ziffer genau 3 binären Ziffern entspricht:

Oktal	Binär
0	000
1	001
2	010
3	011
4	100
5	101
6	110
7	111

In C werden hexadezimale Zahlen mit dem Präfix `0x` und oktale Zahlen mit dem Präfix `0` geschrieben. Deshalb ist es in C nicht möglich Dezimalzahlen mit führenden Nullen zu schreiben.

Innerhalb doppelter und einfacher Anführungszeichen können neben den Backslash-Sequenzen aus Block 19 auch Zahlenwerte für Zeichen direkt als dreistellige Oktalzahlen oder zweistellige Hexadezimalzahlen angegeben werden. Dazu wird einfach nur ein Backslash vor die Oktalzahl bzw. ein Backslash gefolgt vom Buchstaben `x` vor die Hexadezimalzahl gestellt. D.h. es sind z.B. die Ausdrücke `'\n'`, `'\012'` und `'\x0A'` gleichwertig.

### Übung:

Wandeln Sie die Binärzahl

110000000001110111000000111111111110

in die Hexadezimalschreibweise und Oktalschreibweise um.

### Übung:

So wie `consolePrintf` mit `%d` angewiesen werden kann eine Zahl in Dezimalschreibweise auszugeben, so kann `consolePrintf` mit `%x` bzw. `%o` angewiesen werden, eine Zahl in Hexadezimalschreibweise bzw. Oktalschreibweise auszugeben.

Schreiben Sie ein Programm das von der seriellen Schnittstelle Zahlen (dezimal) einliest und diese hexadezimal sowie oktal ausgibt!

### Diskussion:

Manche C-Compiler (unter anderem der GCC, der von der Arduino Entwicklungsumgebung verwendet wird) unterstützen ein Präfix `0b` zur direkten Angabe von Binärzahlen. Was sind die Vorteile und was die Gefahren von solchen compilerspezifischen Spracherweiterungen?

## Block 25: Operationen auf Bitmuster

Wenn man mit Bitmustern arbeitet, speichert man diese in der Regel in Integervariablen. Für diese gibt es neben den arithmetischen Operatoren auf Bit-Operatoren, die den Variableninhalt nicht als Ganzzahl interpretieren, sondern direkt die einzelnen Bits manipulieren.

Der `~`-Operator bildet sein Ergebnisbitmuster indem er die Bits des Operanden invertiert:

a	~a
0	1
1	0

Das heisst z.B. der Ausdruck `~0xAA01` ist gleichbedeutend mit `0x55FE`.

Die Operatoren `&`, `|` sowie `^` verknüpfen die Operanden mit einer Bitweisen AND-, OR- sowie XOR-Operation:

a	b	a & b	a   b	a ^ b
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Der Ausdruck `a << b` verschiebt die Bits im Bitmuster `a` um `b` Stellen nach links. Dabei werden die `b` obersten Bits von `a` verworfen und die `b` untersten Bits des Ergebnisses auf 0 gesetzt.

Analog dazu verschiebt `a >> b` die Bits im Bitmuster `a` um `b` Stellen nach rechts. Wenn `a` ein vorzeichenbehafteter Integer ist und das oberste Bits von `a` gesetzt ist, werden die obersten `b` Bits des Ergebnisses auf 1 statt auf 0 gesetzt.

### Übung:

Auf welche Werte evaluieren die folgenden Ausdrücke?

```
0x1234 << 2   0x800 >> 2   0x888 >> 3   (~0) << 1
0xF0 & 15    0xF0 | 15   073 ^ 037  (3|5)&(~2)
```

Überprüfen Sie Ihre Ergebnisse mit einem Programm!

### Diskussion:

Wir haben bis jetzt `==` und `!=` zum Vergleichen von Ganzzahlen verwendet. Kann man diese Operatoren auch zum Vergleichen von Bitmustern verwenden?

### Diskussion:

Erklären Sie das Funktionsprinzip des folgenden Programms:

```
1  int input = 0;
2
3  void setup()
4  {
5      pinMode(2, INPUT);
6      pinMode(3, INPUT);
7      pinMode(13, OUTPUT);
8      digitalWrite(2, 1); // enable pullup resistor
9      digitalWrite(3, 1); // enable pullup resistor
10 }
11
12 void loop()
13 {
14     uint8_t onCode = 0x75;
```

```
15  uint8_t offCode = ~onCode;
16  uint8_t codeMask = 0x7F;
17  int bit;
18
19  // make sure no button is pressed
20  while (!digitalRead(2) || !digitalRead(3)) { }
21
22  // read a bit
23  while (1)
24  {
25      if (!digitalRead(2)) {
26          bit = 0;
27          break;
28      }
29
30      if (!digitalRead(3)) {
31          bit = 1;
32          break;
33      }
34  }
35
36  // update input
37  input = (input << 1) | bit;
38
39  // check on and off condition
40  if ((input & codeMask) == onCode)
41      digitalWrite(13, 1);
42  else if ((input & codeMask) == offCode)
43      digitalWrite(13, 0);
44  }
```

block25\_bincode.pde

## Block 26: Division und Modulo

Bisher haben wir die arithmetischen Operatoren +, - und \* für die Addition, Subtraktion und Multiplikation kennengelernt.

Zum Dividieren wird der Operator / verwendet.

Beim Dividieren von Integern wird der Nachkommaanteil verworfen. D.h. das Divisionsergebnis wird zu 0 hin gerundet. z.B.:

```
a = 7 / 3; // ergibt 2
```

Der Operator % (Modulo) liefert den Rest, der bei dieser Division bleibt:

```
a = 7 % 3; // ergibt 1
```

Anders als beim / Operator ist der % Operator nur für Integer verfügbar.

Das Verhalten eines Programms mit einer Division durch 0 ist nicht definiert. Daher müssen Programme so geschrieben werden, dass eine Division durch 0 nie auftreten kann.

### Übung:

Auf welche Werte evaluieren die folgenden Ausdrücke?

12 / 4	12 % 4
12 / 5	12 % 5
-9 / 4	-9 % 4
7 / -3	7 % -3
-5 / -2	-5 % -2

Überprüfen Sie Ihre Ergebnisse mit einem Programm!

## Block 27: Zusammengesetzte Zuweisungsoperatoren

Häufig findet man Ausdrücke der Form:

```
a = a + 5;
```

Diese können in C mit Hilfe der zusammengesetzten Zuweisungsoperatoren kürzer geschrieben werden:

```
a += 5;
```

Solche zusammengesetzte Zuweisungsoperatoren gibt es für folgende Infix-Operatoren:

traditionell	kurz
<code>a = a + b;</code>	<code>a += b;</code>
<code>a = a - b;</code>	<code>a -= b;</code>
<code>a = a * b;</code>	<code>a *= b;</code>
<code>a = a / b;</code>	<code>a /= b;</code>
<code>a = a % b;</code>	<code>a %= b;</code>
<code>a = a &lt;&lt; b;</code>	<code>a &lt;&lt;= b;</code>
<code>a = a &gt;&gt; b;</code>	<code>a &gt;&gt;= b;</code>
<code>a = a &amp; b;</code>	<code>a &amp;= b;</code>
<code>a = a   b;</code>	<code>a  = b;</code>
<code>a = a ^ b;</code>	<code>a ^= b;</code>

## Block 28: Der ?: Operator

Der ?: Operator ermöglicht eine Fallunterscheidung innerhalb eines Ausdrucks:

*Bedingung ? Ausdruck<sub>1</sub> : Ausdruck<sub>2</sub>*

Wenn die Bedingung wahr ist wird der erste Ausdruck verwendet, ansonsten wird der zweite Ausdruck verwendet.

Damit lässt sich z.B. der Code

```
int max;
if (a > b)
    max = a;
else
    max = b;
```

auch kürzer schreiben:

```
int max = a > b ? a : b;
```

Beim ?: Operator müssen die alternativen Ausdrücke auf den gleichen Datentyp evaluieren.

### Übung:

Implementieren Sie mit dem ?: Operator eine Funktion die das Minimum ihrer 3 Integer Argumente zurückliefert.

## Diskussion

Was sind die Vor- und Nachteile des `?:` Operators gegenüber dem `if` Statement.

## Block 29: Operatoren und Operatorbindungen

Wie in arithmetischen Ausdrücken üblich kann man Teilausdrücke mit Hilfe der runden Klammern gruppieren um deren Auswertungsreihenfolge festzusetzen.

Wenn keine Klammern verwendet werden dann entscheiden Präzedenz und Assoziativität der Operatoren über die Auswertungsreihenfolge. Operatoren mit niedriger Präzedenzzahl werden zuerst ausgewertet. Bei einer Liste von Operatoren gleicher Präzedenz entscheidet die Assoziativität der Operatoren ob die Liste von links nach rechts oder von rechts nach links abgearbeitet wird.

Präzedenzzahl	Assoziativität	Operator
1	Links-nach-Rechts	++ und -- (Postfix) [] (Array-Zugriff) . und -> (Struct-Zugriff)
2	Rechts-nach-Links	++ und -- (Prefix) + und - (Prefix) * und & (Prefix) ! und ~
3	Links-nach-Rechts	*, / und %
4	Links-nach-Rechts	+ und -
5	Links-nach-Rechts	<< und >>
6	Links-nach-Rechts	<, <=, >= und >
7	Links-nach-Rechts	== und !=
8	Links-nach-Rechts	&
9	Links-nach-Rechts	^
10	Links-nach-Rechts	
11	Links-nach-Rechts	&&
12	Links-nach-Rechts	
13	Rechts-nach-Links	?:

## Übung

Auf welche Werte evaluieren die folgenden Ausdrücke?

5+3\*2

5+3 < 7\*2

6&3 == 2

Überprüfen Sie Ihre Ergebnisse mit einem Programm!

### Diskussion

In welchen Fällen ist es sinnvoll Klammern zu setzen obwohl sich dadurch nichts an der Auswertungsreihenfolge ändert?

## Block 30: Defines und bedingte Übersetzung

Mit dem **#define** Statement kann man einen Namen für ein Code-Fragment definieren:

```
#define ONE 1
```

Ab diesem Statement wird der Name durch das Code-Fragment ersetzt:

```
consolePrintf("%d\n", ONE);
```

Das ist unter anderem nützlich um sprechende Namen für Pin-Nummern zu vergeben:

```
1 #define LED_PIN 3
2
3 void setup()
4 {
5     pinMode(LED_PIN, OUTPUT);
6 }
7
8 void loop()
9 {
10    digitalWrite(LED_PIN, 1);
11    delay(500);
12    digitalWrite(LED_PIN, 0);
13    delay(500);
14 }
```

block30\_blink1.pde

Mit dem `#ifdef ... #endif` Statement kann man in Abhängigkeit davon, ob ein Name mit dem `#define` Statement definiert worden ist, Code ein- oder ausblenden:

```
1 #define LED_PIN 3
2 #define LED_PIN2 4
3
4 void setup()
5 {
6     pinMode(LED_PIN, OUTPUT);
7     #ifdef LED_PIN2
8         pinMode(LED_PIN2, OUTPUT);
9     #endif
10 }
11
12 void loop()
13 {
14     digitalWrite(LED_PIN, 1);
15     delay(500);
16     digitalWrite(LED_PIN, 0);
17     delay(500);
18     #ifdef LED_PIN2
19         digitalWrite(LED_PIN2, 1);
20         delay(500);
21         digitalWrite(LED_PIN2, 0);
22         delay(500);
23     #endif
24 }
```

block30\_blink2.pde

Analog dazu kann mit dem `#ifndef ... #endif` Statement auf nicht-definiert-heit eines Namens geprüft werden. Für den Fall, dass für beide Fälle Code vorgesehen ist gibt es auch ein `#else` Statement:

```
1 #define LED_PIN 3
2 #define LED_PIN2 4
3
4 void setup()
5 {
6     pinMode(LED_PIN, OUTPUT);
7     #ifdef LED_PIN2
8         pinMode(LED_PIN2, OUTPUT);
9     #endif
10 }
11
12 void loop()
13 {
```

```
14 #ifndef LED_PIN2
15     digitalWrite(LED_PIN, 1);
16     delay(500);
17     digitalWrite(LED_PIN, 0);
18     delay(500);
19 #else
20     digitalWrite(LED_PIN, 1);
21     digitalWrite(LED_PIN2, 0);
22     delay(500);
23     digitalWrite(LED_PIN, 0);
24     digitalWrite(LED_PIN2, 1);
25     delay(500);
26 #endif
27 }
```

block30\_blink3.pde

## Diskussion

Was ist der Unterschied zwischen den folgenden Code-Fragmenten?

```
#define TWO 1+1           #define TWO (1+1)
consolePrintf("%d\n", TWO*2); consolePrintf("%d\n", TWO*2);
```

## Übung

Suchen Sie sich eines der bisherigen Programme aus und bauen Sie sinnvolle Defines ein.

## Diskussion

Statt eines Defines könnte man auch eine globale Variable verwenden der nur bei der initialisierung ein Wert zugewiesen wird. Was ist der Vorteil eines Defines gegenüber so einer globalen Variable?

## Anhang I: Protokoll der Einheiten

Dieser Anhang beinhaltet eine Aufstellung der Kurstage aus dem ersten Turnus 2012/2013. Sie soll als Orientierungshilfe für zukünftige Turnusse dienen.

Tag	Datum	Inhalte	Dauer
1	??, ??.??2012	Blöcke 0 - ?	?:??

## Anhang II: Shield

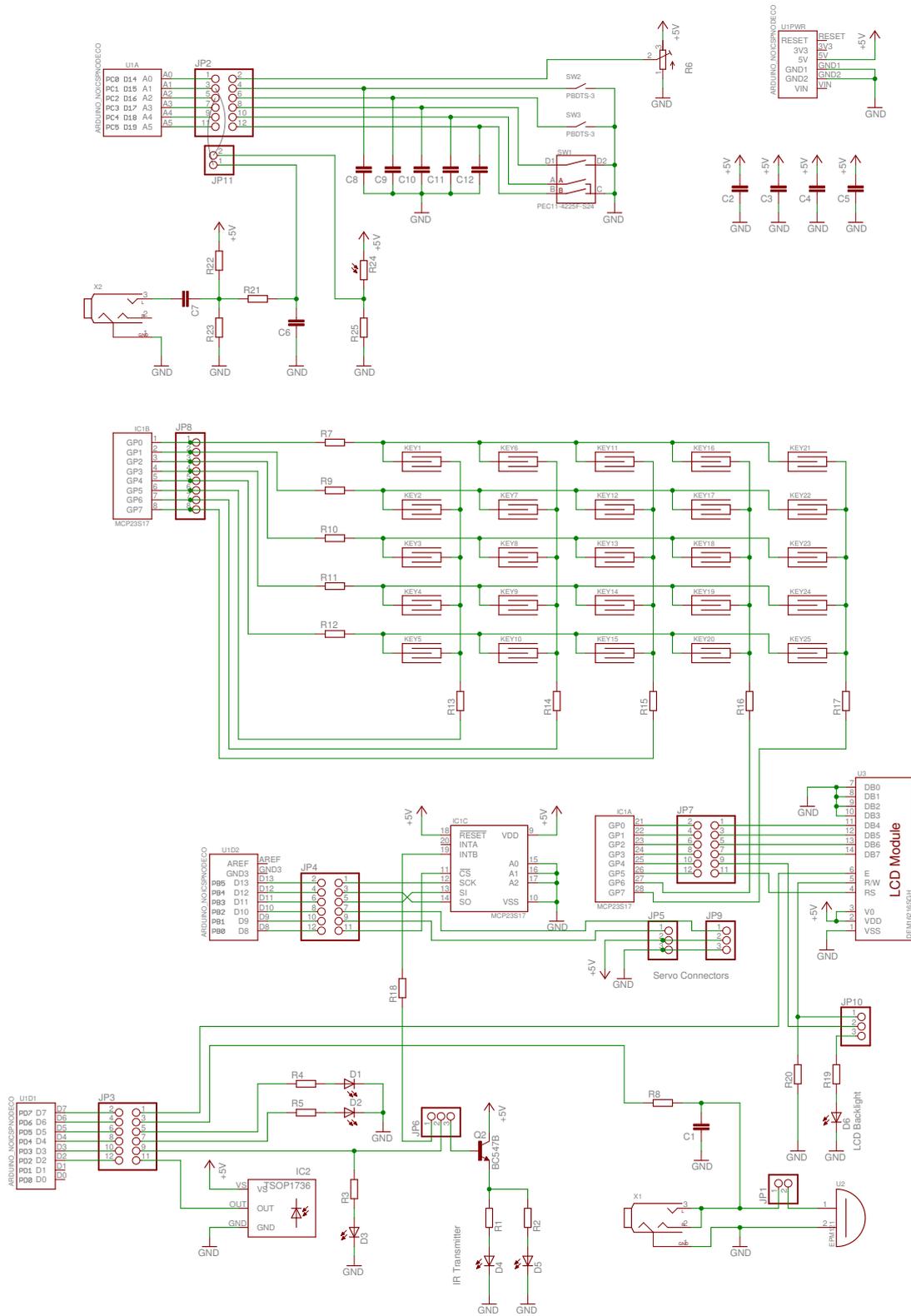
Das Arduino-Shield für den Kurs hat folgende Pinbelegung:

I/O-Pin	Funktion
PD0	0 RS232 RX (serielle Konsole)
PD1	1 RS232 TX (serielle Konsole)
PD2	2 IR-Empfänger (Interrupt)
PD3	3 Rote LED und IR-LED (PWM)
PD4	4 Grüne LED (GPIO)
PD5	5 Blaue LED (GPIO)
PD6	6 Lautsprecher (PWM)
PD7	7 LCD Modul E (Clock)
PB0	8 I/O-Expander $\overline{CS}$
PB1	9 Servo Anschluss A (16 Bit PWM)
PB2	10 Servo Anschluss B (16 Bit PWM)
PB3	11 I/O-Expander MOSI
PB4	12 I/O-Expander MISO
PB5	13 I/O-Expander SCK
PC0	A0 / 14 Poti (ADC)
PC1	A1 / 15 Button 1
PC2	A2 / 16 Button 2
PC3	A3 / 17 Rotary Encoder Button
PC4	A4 / 18 Rotary Encoder Bit A
PC5	A5 / 19 Rotary Encoder Bit B

Das Display und das Keyboard sind über einen SPI I/O-Expander angebunden. Die Pinbelegung am I/O-Expander ist:

I/O-Pin	Funktion	I/O-Pin	Funktion
GPA0	LCD Modul DB4	GPB0	Tastefeld Zeile 1
GPA1	LCD Modul DB5	GPB1	Tastefeld Zeile 2
GPA2	LCD Modul DB6	GPB2	Tastefeld Zeile 3
GPA3	LCD Modul DB7	GPB3	Tastefeld Zeile 4
GPA4	LCD Modul R/W	GPB4	Tastefeld Zeile 5
GPA5	LCD Modul RS	GPB5	Tastefeld Spalte 1
GPA6	Tastefeld Spalte 4	GPB6	Tastefeld Spalte 2
GPA7	Tastefeld Spalte 5	GPB7	Tastefeld Spalte 3

## Anhang II: Shield



## Anhang III: Angehängte Dateien

Dieses PDF-Dokument enthält angehängte Dateien. Die meisten PDF-Reader erlauben es, diese Dateien zu extrahieren. Um das zu tun, muss man mit der rechten (zweiten) Maustaste auf den rot gesetzten Dateinamen klicken und den entsprechenden Eintrag im Kontextmenu wählen.

**Datei 1: `texsourcen.zipx`**

Die vollständigen  $\text{\LaTeX}$ -Sourcen zu diesem Skriptum als ZIP Datei.  
(Nach dem Abspeichern aus dem PDF nach `*.zip` umbenennen! Acrobat Reader erlaubt leider keine `*.zip` Dateien in PDFs. Daher dieser Hack..)

**Datei 2: `examples.zipx`**

Der Sourcecode zu allen Beispielprogrammen.

**Datei 3: `Console.zipx`**

Die Arduino `Console` Library, die wir in diesem Kurs verwenden.