Metalab Kurs
μC-Programmierung in C

Clifford Wolf
Stefan Farthofer

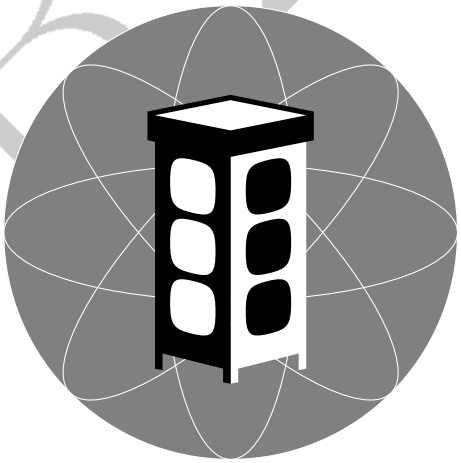
im August 2012

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

`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



This page is intentionally left blank.

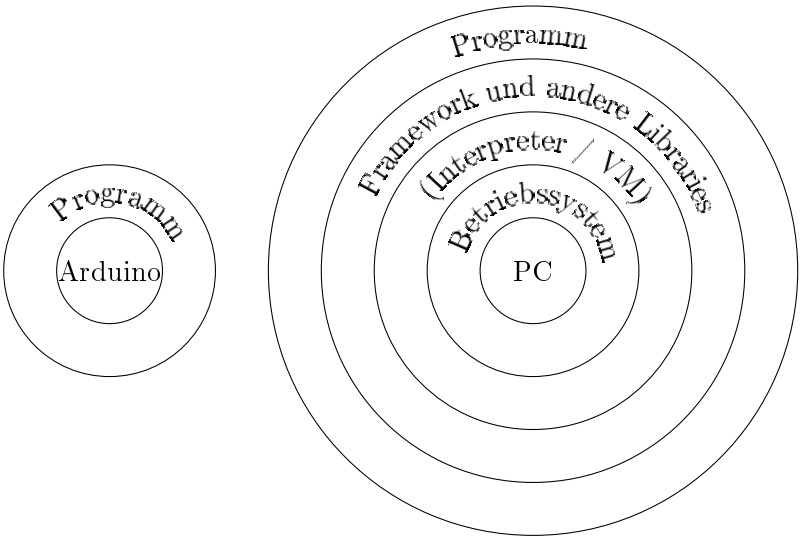
Inhaltsverzeichnis

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 12
Block 6: Globale und lokale Variablen 13
Block 7: Struktogramme 14
Block 8: Fallunterscheidungen 16
Block 9: Boolsche Ausdrücke 17
Block 10: Schleifen 20
Block 11: Zählerschleifen 23
Block 12: Pre- und Postinkrement 24
Block 13: Einfache Datentypen 26
Block 14: Arrays 27
Block 15: Mehr zu Arrays 29
Block 16: Zusammengesetzte Datentypen 31
Block 17: Pointer 33
Block 18: Pointer auf Arrays 36
Block 19: Textzeichen 38
Block 20: Strings (Zeichenketten) 41
Block 21: break, continue und goto 43
Block 22: Das switch-Statement 44
Block 23: Binärdarstellung von Ganzzahlen 46
Block 24: Hezadezimal- und Oktalzahlen 48
Block 25: Operationen auf Bitmuster 49
Block 26: Division und Modulo 52
Block 27: Zusammengesetzte Zuweisungsoperatoren 53
Block 28: Der ?: Operator 53
Block 29: Operatoren und Operatorbindungen 54
Block 30: Defines und bedingte Übersetzung 56
Block 31: IR-Empfänger 58
Block 32: IR-Empfänger und -Sender 61
Block 33: SPI I/O-Expander 64
Anhang I: Protokoll der Einheiten 66
Anhang II: Shield 67

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 Peripherieelemente erweitern, sinnvoll.

Wir verwenden für unsere Übungen unser eigenes 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 "lib_console.c"
2  #include "lib_main.c"
3
4  void setup()
5  {
6      consoleInit(9600);
7      consolePrint("Hello World!\n");
8  }
9
10 void loop()
11 {
12     // Hier passiert nichts.
13 }
```

`block01_hallo.c`

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 `lib_console.c` eingebunden werden.

Die Library `lib_main.c` bewirkt, dass die Funktion `setup` einmal beim Programmstart und die Funktion `loop` danach in einer Endlosschleife ausgeführt wird.

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 $-32768 \ldots 32767$ speichern[1].

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

```
1  #include "lib_main.c"
2  #include "lib_console.c"
3
4  int summe;
5  int eingabe;
6
7  void setup()
8  {
9      consoleInit(9600);
10     summe = 0;
11 }
12
13 void loop()
14 {
15     eingabe = consoleReadDecimal("Geben Sie eine Zahl ein: ");
16     summe = summe + eingabe;
17
18     consolePrintf("Die Summe ist derzeit: %d\n", summe);
19 }
```

block02_summe.c

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.

[1] Dieser Wertebereich gilt für den Prozessor am Ardunio. 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 μC) sind frei als Eingabe- oder Ausgabepins verwendbar. Wenn der Pin für Eingabe verwendet wird, kann optional ein interner Pullup-Widerstand zugeschaltet werden.

```
1  #include "lib_main.c"
2  #include "lib_pinio.c"
3
4  int pinValue;
5
6  void setup()
7  {
8      pinMode(15, INPUT);
9      pinMode(3, OUTPUT);
10     digitalWrite(15, 1); // enable pullup resistor
11 }
12
13 void loop()
14 {
15     pinValue = digitalRead(15);
16     digitalWrite(3, pinValue);
17 }
```

`block03_pinio.c`

Die Library `lib_pinio.c` stellt die Funktionen `pinMode`, `digitalWrite` und `digitalRead` zur Verfügung.

Die `pinMode` Funktion konfiguriert einen Pin als Ein- oder Ausgabepin. In diesem Beispiel wird Pin Nummer 15 als Eingabepin und Pin Nummer 3 als Ausgabepin verwendet.

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:

```c
#include "lib_console.c"
#include "lib_pinio.c"
#include "lib_main.c"
#include <util/delay.h>

void blinkPin(int pin)
{
    digitalWrite(pin, 1);
    _delay_ms(1000);
    digitalWrite(pin, 0);
    _delay_ms(100);
}

void setup()
{
    pinMode(3, OUTPUT);
    pinMode(4, OUTPUT);
}

void loop()
{
    blinkPin(3);
    blinkPin(3);
    blinkPin(4);
}
```

block04_blink1.c

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:

```c
#include "lib_pinio.c"
#include "lib_main.c"
#include <util/delay.h>

void blinkPin(int pin, int lengthMs)
{
    digitalWrite(pin, 1);
    _delay_ms(lengthMs);
    digitalWrite(pin, 0);
    _delay_ms(100);
}

```

```
13 void setup()
14 {
15     pinMode(3, OUTPUT);
16     pinMode(4, OUTPUT);
17 }
18
19 void loop()
20 {
21     blinkPin(3, 1000);
22     blinkPin(3, 2000);
23     blinkPin(4, 3000);
24 }
```

block04_blink2.c

Die Funktion `_delay_ms` wartet übrigens die angegebene Anzahl von Millisekunden. Um diese Funktion verwenden zu können muss erst die Datei `util/delay.h` eingebunden werden. Da es sich hierbei um keine Datei aus dem lokalen Verzeichnis handelt wird der Dateiname mit spitzen Klammern statt doppelten Anführungszeichen angegeben.

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 "lib_console.c"
2  #include "lib_pinio.c"
3  #include "lib_main.c"
4  #include <util/delay.h>
5
6  int getFlashDurationMs()
7  {
8      return consoleReadDecimal("Wieviele Sekunden? ") * 1000;
9  }
10
11 void blinkPin(int pin, int lengthMs)
12 {
13     digitalWrite(pin, 1);
14     _delay_ms(lengthMs);
15     digitalWrite(pin, 0);
16 }
17
18 void setup()
19 {
20     consoleInit(9600);
21     pinMode(3, OUTPUT);
22 }
23
24 void loop()
25 {
26     blinkPin(3, getFlashDurationMs());
27 }
```

block05_flash.c

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:

```
6  int getFlashDurationMs()
7  {
8      int durationSeconds;
9      durationSeconds = consoleReadDecimal("Wieviele Sekunden? ");
10     return durationSeconds * 1000;
11 }
```

block06_durationms.c

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:

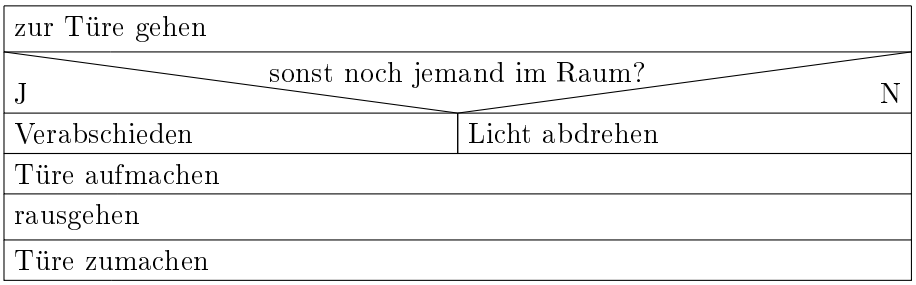
Man kann mit Kommata getrennt auch mehrere Variablen auf einmal definieren und man kann den einzelnen Variablen bereits direkt in der Definition einen initialen Wert zuweisen. Zum Beispiel:

```
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:

```c
#include "lib_pinio.c"
#include "lib_main.c"

void setup()
{
    pinMode(15, INPUT);
    pinMode(16, INPUT);
    pinMode(17, INPUT);

    pinMode(3, OUTPUT);
    pinMode(4, OUTPUT);

    // enable pullup resistors
    digitalWrite(15, 1);
    digitalWrite(16, 1);
    digitalWrite(17, 1);
}

void loop()
{
    int valueA, valueB;
    int doSwitch = digitalRead(17);

    if (doSwitch)
    {
        valueA = digitalRead(15);
        valueB = digitalRead(16);
    }
    else
    {
        valueA = digitalRead(16);
        valueB = digitalRead(15);
    }

    digitalWrite(3, valueA);
    digitalWrite(4, valueB);
}
```

block08_if.c

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 sinvoll und wichtig, richtig einzurücken?

Block 9: Boolsche Ausdrücke

Wahrheitswerte können durch den Vergleich von Zahlenwerten gebildet werden:

```
6 int getFlashDurationMs()
```

```
{
    int durationSeconds =
            consoleReadDecimal("Wieviele Sekunden? ");

    if (durationSeconds < 0)
        durationSeconds = 0;
    else if (durationSeconds > 10)
        durationSeconds = 10;

    return durationSeconds * 1000;
}
```

block09_rel.c

Dabei können folgende Operatoren verwendet werden:

`a < b`	`a` kleiner `b`
`a <= b`	`a` kleiner oder gleich `b`
`a == b`	`a` gleich `b`
`a != b`	`a` ungleich `b`
`a >= b`	`a` größer oder gleich `b`
`a > b`	`a` größer `b`

Wahrheitswerte können auch durch Verknüpfungen von mehreren Wahrheitswerten gebildet werden:

```
#include "lib_pinio.c"
#include "lib_main.c"

void setup()
{
    pinMode(15, INPUT);
    pinMode(16, INPUT);
    pinMode(3, OUTPUT);
    digitalWrite(15, 1); // enable pullup resistor
    digitalWrite(16, 1); // enable pullup resistor
}

void loop()
{
    if (!digitalRead(15) && !digitalRead(16))
        digitalWrite(3, 1);
    else
        digitalWrite(3, 0);
}
```

block09_logic.c

Dabei können folgende Operatoren verwendet werden:

`a && b`	`a` und `b`
`a || b`	`a` oder `b`
`!a`	nicht `a`

Grundsätzlich können Wahrheitswerte in Variablen des Typs **`int`** gespeichert werden, besser ist es jedoch den dafür vorgesehenen Datentyp **`bool`** zu verwenden. Dazu muss die Datei **`stdbool.h`** mit **`#include <stdbool.h>`** eingebunden werden.

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.c`** 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 #include "lib_pinio.c"
2 #include "lib_main.c"
3 #include <util/delay.h>
4
5 void setup()
6 {
```

```
7     pinMode(15, INPUT);
8     pinMode(16, INPUT);
9     pinMode(3, OUTPUT); // red LED
10    pinMode(4, OUTPUT); // green LED
11    digitalWrite(15, 1); // enable pullup resistor
12    digitalWrite(16, 1); // enable pullup resistor
13 }
14
15 void loop()
16 {
17    if (!digitalRead(15) || !digitalRead(16))
18    {
19        _delay_ms(100);
20        if (!digitalRead(15) && !digitalRead(16))
21            digitalWrite(4, 1);
22        else
23            digitalWrite(3, 1);
24        _delay_ms(1000);
25    }
26    digitalWrite(3, 0);
27    digitalWrite(4, 0);
28 }
```

block09_sync.c

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 $\leftarrow 0$
input $\leftarrow$ Eingabe()
solange input $\neq 0$
summe $\leftarrow$ summe + input
input $\leftarrow$ Eingabe()
Ausgabe(summe)

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

```
1 #include "lib_console.c"
```

```
2  #include "lib_main.c"
3
4  void setup()
5  {
6      consoleInit(9600);
7  }
8
9  void loop()
10 {
11     int summe = 0;
12     int input = consoleReadDecimal("Erster Summand: ");
13     while (input != 0)
14     {
15         summe = summe + input;
16         input = consoleReadDecimal("Weiterer Summand: ");
17     }
18     consolePrintf("Die Summe ist %d\n", summe);
19 }
```

block10_loop.c

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.c`, 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 Laufbedinung 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?

Diskussion:

Bis jetzt haben wir `lib_main.c` verwendet und eigene Funktionen `setup` und `loop` definiert, wobei `setup` einmal beim Programmstart und `loop` danach in einer Endlosschleife ausgeführt wird.

Üblicherweise wird aber in C-Programmen einfach nur eine Funktion `main` definiert die genau ein mal ausgeführt wird. Somit kann das oben stehende Programm ohne `lib_main.c` auch folgendermassen geschrieben werden:

```
1  #include "lib_console.c"
2
3  int main()
4  {
5      consoleInit(9600);
6
7      while(1)
8      {
9          int summe = 0;
10         int input = consoleReadDecimal("Erster Summand: ");
11         while (input != 0)
12         {
13             summe = summe + input;
14             input = consoleReadDecimal("Weiterer Summand: ");
15         }
16         consolePrintf("Die Summe ist %d\n", summe);
17     }
18
19     return 0;
20 }
```

block10_main.c

Zu bechten ist, dass `main` einen Wert vom Typ `int` zurückliefert. Dieser wird in Microcontroller-Anwendungen jedoch nie verwendet.

Block 11: Zählerschleifen

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

i $\leftarrow$ 0
solange i $<$ 100
Schleifenkörper
i $\leftarrow$ i + 1

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  #include "lib_pinio.c"
2  #include <util/delay.h>
3
4  void blinkPin(int pin)
5  {
6      digitalWrite(pin, 1);
7      _delay_ms(1000);
8      digitalWrite(pin, 0);
9      _delay_ms(100);
10 }
11
12 int main()
13 {
14     pinMode(3, OUTPUT);
15     pinMode(4, OUTPUT);
16     pinMode(5, OUTPUT);
```

```
17
18      while(1)
19      {
20          int i,j;
21
22          for (i = 3; i <= 5; i = i + 1)
23          {
24              for (j = 0; j < 3; j = j + 1)
25                  blinkPin(i);
26          }
27      }
28
29      return 0;
30  }
```

block11_blink.c

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 1×1 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:

```c
#include "lib_console.c"

int main()
{
    int i = 0;
    consoleInit(9600);
    consolePrintf("i == %d\n", i);
    consolePrintf("++i == %d\n", ++i);
    consolePrintf("i == %d\n", i);
    consolePrintf("i++ == %d\n", i++);
    consolePrintf("i == %d\n", i);

    return 0;
}
```

block12_inc.c

Überprüfen Sie Ihre Vorhersage.

Diskussion:

Vergleichen Sie folgende Programmfragmente:

```c
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		
`int8_t`	−128	...	127
`int16_t`	−32 768	...	32 767
`int32_t`	−2 147 483 648	...	2 147 483 647
`uint8_t`	0	...	255
`uint16_t`	0	...	65 535
`uint32_t`	0	...	4 294 967 295

Um diese Datentypen verwenden zu können muss die Datei `stdint.h` mit `#include <stdint.h>` eingebunden werden.

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

Datentyp	Genauigkeit
`float`	6 Dezimalstellen
`double`	15 Dezimalstellen

Wenn mit einem arithmetischen Operator ein Integer- und ein Fliesskommawert miteinander verknüpft werden, so wird der Integerwert vor der Operation autmatisch 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 Ausrruck 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:

```c
#include "lib_console.c"
#include "lib_pinio.c"
#include <util/delay.h>

int pattern[10];

void blinkPin(int pin)
{
    digitalWrite(pin, 1);
    _delay_ms(1000);
    digitalWrite(pin, 0);
    _delay_ms(100);
}

int main()
{
    int i;
    pinMode(3, OUTPUT);
    pinMode(4, OUTPUT);
    pinMode(5, OUTPUT);
    consoleInit(9600);

    for (i=0; i < 10; i++)
    {
        consolePrintf("LED fuer Schritt %d: ", i);
        pattern[i] = consoleReadDecimal("");
    }

    while(1)
    {
        int i;
        for (i = 0; i < 10; i++)
            blinkPin(pattern[i]);

        _delay_ms(2000);
    }

    return 0;
}
```

block14_blinkpattern.c

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 dreidemsionale Arrays veranschaulichen?

Wfür kann man mehrdimensionale Arrays brauchen?

Diskussion:

Erklären Sie die Funktionsweise des folgenden Programms:

```c
#include "lib_pinio.c"

int pattern[2][2][2] = {
    { { 0, 1 }, { 0, 0 } },
    { { 0, 0 }, { 2, 0 } }
};

int main()
{
    pinMode(15, INPUT);
    pinMode(16, INPUT);
    pinMode(17, INPUT);
    pinMode(3, OUTPUT);
    digitalWrite(15, 1); // enable pullup resistor
    digitalWrite(16, 1); // enable pullup resistor
    digitalWrite(17, 1); // enable pullup resistor

    while(1)
    {
        int key1 = !digitalRead(15);
        int key2 = !digitalRead(16);
        int key3 = !digitalRead(17);
        int command = pattern[key1][key2][key3];

        if (command == 1)
            digitalWrite(3, 1);

        if (command == 2)
            digitalWrite(3, 0);
    }

    return 0;
}
```

block15_combination.c

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 #include "lib_pinio.c"
2 #include "lib_tone.c"
```

```c
#include <util/delay.h>

struct note { int freq; int duration; };

struct note melody[8] = {
    { 200, 3 },
    { 277, 3 },
    { 330, 3 },
    { 277, 1 },
    { 294, 3 },
    { 370, 3 },
    { 294, 1 },
    { 440, 5 }
};

void playNote(struct note n)
{
    int i;
    tone(n.freq);
    for (i = 0; i < n.duration; i++)
        _delay_ms(150);
    tone(0);
    _delay_ms(50);
}

int main()
{
    int i;

    while(1)
    {
        for (i=0; i < 8; i++)
            playNote(melody[i]);

        _delay_ms(5000);
    }

    return 0;
}
```

block16_song.c

Diskussion:

Erklären Sie das Funktionsprinzip dieses Programms:

```c
#include "lib_pinio.c"
```

```
2
3  struct state
4  {
5      bool led1, led2;
6      int targetStates[2][2];
7  };
8
9  struct state states[] = {
10     { 0, 0, { { 0, 1 }, { 2, 0 } } },
11     { 0, 1, { { 1, 1 }, { 1, 3 } } },
12     { 1, 0, { { 2, 2 }, { 2, 3 } } },
13     { 1, 1, { { 0, 3 }, { 3, 3 } } }
14 };
15
16 int currentState = 0;
17
18 int main()
19 {
20     pinMode(3, OUTPUT);
21     pinMode(4, OUTPUT);
22     pinMode(15, INPUT);
23     pinMode(16, INPUT);
24     digitalWrite(15, 1); // enable pullup resistor
25     digitalWrite(16, 1); // enable pullup resistor
26
27     while(1)
28     {
29         digitalWrite(3, states[currentState].led1);
30         digitalWrite(4, states[currentState].led2);
31
32         bool b1, b2;
33         b1 = !digitalRead(15);
34         b2 = !digitalRead(16);
35
36         currentState = states[currentState].targetStates[b1][b2];
37     }
38
39     return 0;
40 }
```

block16_fsm.c

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  #include "lib_console.c"
2  #include "lib_pinio.c"
3  #include "lib_tone.c"
4  #include <util/delay.h>
5
6  struct note { int freq; int duration; };
7
8  struct note melody[64] = {
9      { 440, 1000 },
10     { 880, 1000 },
11     { 440, 1000 },
12     { 880, 1000 }
13 };
14
15 void editNote(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(n.freq);
24     _delay_ms(n.duration);
25     tone(0);
26 }
27
28 void playMelody()
29 {
30     int i;
31     for (i = 0; i < 64; i++)
32         playNote(melody[i]);
33 }
34
35 int main()
36 {
37     consoleInit(9600);
38
39     while(1)
40     {
41         int i;
42
43         i = consoleReadDecimal("Welche Note wollen sie editieren?");
44
45         if (i < 64)
46             editNote(&melody[i]);
47
48         playMelody();
49     }
50
51     return 0;
52 }
```

block17_composer.c

Übung:

Erweitern Sie das Programm `block16_fsm.c` 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 boolscher Ausdruck auf falsch. Jeder andere Pointer evaluiert als boolscher Ausdruck auf wahr.

Ein Pointer mit dem Wert `NULL` kann nur als boolscher 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; // aequivialent zu a[1] = 1;
```

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

```
p[2] = 2; // aequivialent 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:

```
#include "lib_console.c"
#include "lib_pinio.c"
#include "lib_tone.c"
#include <util/delay.h>

struct note { int freq; int duration; };

struct note melodyA[] = {
    { 440, 1000 },
    { 880, 1000 },
    { 440, 1000 },
    { 880, 1000 },
    { 0, 0 }
};

struct note melodyB[] = {
    { 440, 1000 },
    { 880, 1000 },
    { 440, 1000 },
    { 880, 1000 },
    { 0, 0 }
};

struct note melodyC[] = {
    { 440, 1000 },
    { 880, 1000 },
    { 440, 1000 },
    { 880, 1000 },
    { 0, 0 }
};
```

```
31
32 void playNote(struct note *n)
33 {
34     tone(n->freq);
35     _delay_ms(n->duration);
36     tone(0);
37 }
38
39 void playMelody(struct note *melody)
40 {
41     while(melody->duration)
42     {
43         playNote(melody);
44         melody++;
45     }
46 }
47
48 int main()
49 {
50     pinMode(2, INPUT);
51     pinMode(3, INPUT);
52     pinMode(4, INPUT);
53
54     while(1)
55     {
56         if (digitalRead(2))
57             playMelody(melodyA);
58         else if (digitalRead(3))
59             playMelody(melodyB);
60         else if (digitalRead(4))
61             playMelody(melodyC);
62     }
63
64     return 0;
65 }
```

block18_multiplay.c

Block 19: Textzeichen

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

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
\b	8	BS
\n	10	LF
\r	13	CR
\“	34	“
\’	39	’
\\	92	\

Diskussion:

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

```
#include "lib_console.c"

int main()
{
    consoleInit(9600);

    for (char c=32; c < 127; c++)
        consolePrintf("%3d - %c\n", c, c);

    return 0;
}
```

block19_ascii.c

(Wenn zwischen dem Prozent-Zeichen und dem Typ-Klassifier eine Zahl steht, dann gibt diese Zahl die Textlaenge an auf die der eingesetzte Text lingsbündig mit Blanks ergänzt werden soll.)

Diskussion:

Erklären Sie das Funktionsprinzip des folgenden Programms.

```c
#include "lib_console.c"

int main()
{
    consoleInit(9600);

    for (char c=32; c < 127; c++)
        consolePrintf("%d - %c\n", c, c);

    while(1)
    {
        char c;
        consolePrint("Geben Sie ein Zeichen ein:");
        do {
            c = consoleGetChar();
        } while(c < 32);

        consolePrintf("%c\n", c);

        if ('a' <= c && c <= 'z')
            consolePrint("Kleinbuchstabe\n");
        else if ('A' <= c && c <= 'Z')
            consolePrint("Grossbuchstabe\n");
        else if ('0' <= c && c <= '9')
            consolePrint("Ziffer\n");
        else
            consolePrint("Sonderzeichen\n");
    }

    return 0;
}
```

block19_charclass.c

Block 20: Strings (Zeichenketten)

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

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

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

```c
char txt[] = "Test";
```

```
consolePrint(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:

```
consolePrint("Test");
```

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

```
#include "lib_console.c"

void myConsolePrint(char *str)
{
    int i;
    for (i = 0; str[i]; i++)
        consolePutChar(str[i]);
}

int main()
{
    consoleInit(9600);
    myConsolePrint("Hello World!\n");

    return 0;
}
```

block20_print.c

Diskussion:

Erklären Sie das Funktionsprinzip des folgenden Programms:

```
#include "lib_console.c"

void stringAppend(char *a, char *b)
{
    int i, j;

    for (i = 0; a[i]; i++);
    for (j = 0; b[j]; j++)
        a[i+j] = b[j];
    a[i+j] = 0;
```

```
11 }
12
13 int main()
14 {
15     char buffer[100] = "Hallo";
16     consoleInit(9600);
17
18     stringAppend(buffer, " ");
19     stringAppend(buffer, "Welt!");
20     stringAppend(buffer, "\n");
21     consolePrint(buffer);
22
23     return 0;
24 }
```

block20_append.c

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 unabhänig von der Laufbedingung:

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

Das **continue**-Statement beendet den Schleifenkörper der innerste 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 Abhänigkeit 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 werdeb 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: Hezadezimal- und Oktalzahlen

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

Das Hezadezimalsystem 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 desshalb 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 desshalb 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. Desshalb ist es in C nicht möglich Dezimalzahlen mit führenden Nullen zu schreiben.

Innerhalb doppelter und einfacher Anfürungszeichen 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 Okalzahl 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

11000000000111011100000111111111110

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) unterstüzen 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:

```c
#include "lib_pinio.c"

int input = 0;

void loop()
{
    uint8_t onCode = 0x75;
    uint8_t offCode = 0x0A;
    uint8_t codeMask = 0x7F;
    int bit;

    // make sure no button is pressed
    while (!digitalRead(15) || !digitalRead(16)) { }

    // read a bit
    while (1)
    {
        if (!digitalRead(15)) {
            bit = 0;
            break;
        }

        if (!digitalRead(16)) {
            bit = 1;
            break;
        }
    }

    // update input
    input = (input << 1) | bit;

    // check on and off condition
    if ((input & codeMask) == onCode)
        digitalWrite(3, 1);
    else if ((input & codeMask) == offCode)
        digitalWrite(3, 0);
}

int main()
{
    pinMode(15, INPUT);
    pinMode(16, INPUT);
    pinMode(3, OUTPUT);
    digitalWrite(15, 1); // enable pullup resistor
    digitalWrite(16, 1); // enable pullup resistor

    while(1)
        loop();

```

```
50      return 0;
51  }
```

block25_bincode.c

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 Programe 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
`a = a + b;`	`a += b;`
`a = a - b;`	`a -= b;`
`a = a * b;`	`a *= b;`
`a = a / b;`	`a /= b;`
`a = a % b;`	`a %= b;`
`a = a << b;`	`a <<= b;`
`a = a >> b;`	`a >>= b;`
`a = a & b;`	`a &= b;`
`a = a | b;`	`a |= b;`
`a = a ^ b;`	`a ^= b;`

Block 28: Der ?: Operator

Der `?:` Operator ermöglicht eine Fallunterscheidung innerhalb eines Ausdrucks:

$$\textit{Begingung}\ \texttt{?}\ \textit{Ausdruck}_1\ \texttt{:}\ \textit{Ausdruck}_2$$

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 dur das Code-Fragment ersetzt:

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

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

```
#include "lib_pinio.c"
#include <util/delay.h>

#define LED_PIN 3

int main()
{
    pinMode(LED_PIN, OUTPUT);

    while(1)
    {
        digitalWrite(LED_PIN, 1);
        _delay_ms(500);
        digitalWrite(LED_PIN, 0);
        _delay_ms(500);
    }

    return 0;
}
```

block30_blink1.c

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

```
#include "lib_pinio.c"
#include <util/delay.h>

#define LED_PIN 3
#define LED_PIN2 4

```

```c
int main()
{
    pinMode(LED_PIN, OUTPUT);
#ifdef LED_PIN2
    pinMode(LED_PIN2, OUTPUT);
#endif
    while(1)
    {
        digitalWrite(LED_PIN, 1);
        _delay_ms(500);
        digitalWrite(LED_PIN, 0);
        _delay_ms(500);
#ifdef LED_PIN2
        digitalWrite(LED_PIN2, 1);
        _delay_ms(500);
        digitalWrite(LED_PIN2, 0);
        _delay_ms(500);
#endif
    }

    return 0;
}
```

block30_blink2.c

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

```c
#include "lib_pinio.c"
#include <util/delay.h>

#define LED_PIN 3
#define LED_PIN2 4

int main()
{
    pinMode(LED_PIN, OUTPUT);
#ifdef LED_PIN2
    pinMode(LED_PIN2, OUTPUT);
#endif
    while(1)
    {
#ifndef LED_PIN2
        digitalWrite(LED_PIN, 1);
        _delay_ms(500);
        digitalWrite(LED_PIN, 0);
        _delay_ms(500);
```

```
20 #else
21         digitalWrite(LED_PIN, 1);
22         digitalWrite(LED_PIN2, 0);
23         _delay_ms(500);
24         digitalWrite(LED_PIN, 0);
25         digitalWrite(LED_PIN2, 1);
26         _delay_ms(500);
27 #endif
28     }
29
30     return 0;
31 }
```

block30_blink3.c

Diskussion

Was ist der Unterschied zwischen den folgenden Code-Fragmenten?

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

```
#define TWO (1+1)
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 Initializierung ein Wert zugewiesen wird. Was ist der Vorteil eines Defines gegenüber so einer globalen Variable?

Block 31: IR-Empfänger

Wir wenden das bisher gelernte in einem Projekt praktisch an: Wir schreiben das Programm für einen Infrarot-Fernbedienungsempfänger.

Das Experimentierboard kann dazu mit einem TSOP1736 IR-Empfänger-Baustein bestückt werden. Dieser ist dann mit Pin 2 des Arduino verbun-

den. Das demodulierte IR-Signal liegt dann an diesem Pin an. Üblicherweise dauern Pulse bei IR-Signalen mindestens 0,5 ms.

Im folgenden nimmt die Funktion `readSignal()` ein Signal von der IR-Schnittstelle entgegen und speichert es im Array `signalData[]` und die Funktion `printSignal()` gibt das gespeicherte Signal auf der Konsole aus:

```
5  #define IR_PIN 2
6  #define MAX_SIGNAL_LEN 128
7
8  uint8_t signalData[MAX_SIGNAL_LEN] = { /* zeros */ };
9  int signalLen;
10
11 void readSignal()
12 {
13     while (digitalRead(IR_PIN) == 1) { /* wait */ }
14
15     for (signalLen = 0; signalLen < MAX_SIGNAL_LEN; signalLen++) {
16         signalData[signalLen] = 0;
17         while (digitalRead(IR_PIN) == signalLen % 2) {
18             if (++signalData[signalLen] == 255)
19                 return;
20             _delay_us(10);
21         }
22     }
23 }
24
25 void printSignal()
26 {
27     int i;
28
29     for (i = 0; i < signalLen; i++)
30         consolePrintf("%2d ", signalData[i]);
31
32     consolePrintf("\n");
33 }
```

block31_ir1.c

Ein IR-Signal beginnt mit einer positiven Flanke (Signal wechselt von 0 auf 1). Das `signalData[]` Array beinhaltet die Zeitdauer zwischen den Flanken (Wechsel zwischen 0 und 1) im Signal vom IR-Empfänger-Baustein in der Einheit $\approx 10\,\mu$s. In der Variable `signalLen` wird gespeichert wie viele Flanken empfangen wurden.

Diskussion

Wesshalb ist es nicht sinnvoll in einer Funktion gleichzeitig Daten von der IR-Schnittstelle zu Empfangen und auf der Konsole auszugeben?

Übung

Bauen Sie die beiden Funktionen zu einem vollständigen Programm aus und analysieren Sie die signale von IR-Fernbedienungen.

Diskussion

Das wiederholte drücken der selben Taste auf einer Fernbedienung führt nicht zwangslaufig zu exakt identischen Werten in `signalData[]`. Wie kann ein Histogramm wie es der folgende Code ereugt dabei helfen ein Programm entwickeln, dass in der Lage ist einen bestimmten Tastendruck trotz dieser Schwankungen zuverlässig zu erkennen?

```
35 uint8_t max_hist = 0;
36 uint8_t hist[255];
37
38 void addSamplesToHistogram()
39 {
40     int i;
41
42     // increment histogram slot for each unique sample value
43     // use the most significant bit of each histogram slot to
44     // remember whether we already incremented it
45     for (i = 0; i < signalLen; i++) {
46         if ((hist[signalData[i]] & 0x80) == 0) {
47             hist[signalData[i]]++;
48             if (hist[signalData[i]] > max_hist)
49                 max_hist = hist[signalData[i]];
50             hist[signalData[i]] |= 0x80;
51         }
52     }
53
54     // clear flags
55     for (i = 0; i < signalLen; i++)
56         hist[signalData[i]] &= 0x7f;
57 }
58
59 void printHistogram()
60 {
```

```
61      int i,j;
62
63      consolePrint("\n----- HISTOGRAM -----\n");
64      for (i = 0; i < 255; i++) {
65          if (hist[i] == 0) {
66              if (i > 0 && hist[i-1] != 0)
67                  consolePrint("\n");
68              continue;
69          }
70          consolePrintf("%3d %3d ", i, hist[i]);
71          for (j = 0; j < hist[i]; j++)
72              consolePrint("=");
73          consolePrint("\n");
74      }
75
76      consolePrint("\n");
77  }
78
79  void clearHistogram()
80  {
81      int i;
82      for (i = 0; i < 255; i++)
83          hist[i] = 0;
84      max_hist = 0;
85  }
```

block31_ir1.c

Block 32: IR-Empfänger und -Sender

Das Histogramm ausdem vorhergehenden Block zeigt, dass die Einträge in `signalData[]` (Zeitdauern zwischen den Flanken in $\approx 10\,\mu$s) in jeweils eine von drei Kategorien fallen. Wenn man die Einträge in `signalData[]` durch Symbole für die entsprechende Kategorien ersetzt, erhält man eine eindeutige Darstellung des Signals die frei von Rauschen ist. Diese Darstellung hat insbesondere den Vorteil, dass sie leicht mit einem vorab gespeicherten Code verglichen werden kann:

```
10  uint8_t signalData[MAX_SIGNAL_LEN+1] = { /* zeros */ };
11  int signalLen;
12
13  void readSignal()
14  {
15      while (digitalRead(IR_PIN) == 1) { /* wait */ }
16
```

```
    for (signalLen = 0; signalLen < MAX_SIGNAL_LEN; signalLen++) {
        signalData[signalLen] = 0;
        while (digitalRead(IR_PIN) == signalLen % 2) {
            if (++signalData[signalLen] == 255)
                return;
            _delay_us(10);
        }
    }
}

void mapSignalToString()
{
    int i;

    for (i = 0; i < signalLen; i++) {
        if (signalData[i] < 15)
            signalData[i] = 'A';
        else if (signalData[i] < 40)
            signalData[i] = 'B';
        else
            signalData[i] = 'C';
    }
    signalData[signalLen] = 0;
}

// sometimes the sequence 'BAB' is detected as single 'C'
// as 'C' is only valid as first token in the sequence we
// can substitute 'C' tokens inside the sequence with 'BAB'.
void fixupString()
{
    int i;
    int trgPos = 1;

    for (i = 1; i < signalLen; i++)
        trgPos += signalData[i] == 'C' ? 3 : 1;

    signalData[trgPos--] = 0;
    for (i = signalLen-1; i != trgPos; i--) {
        if (signalData[i] == 'C') {
            signalData[trgPos--] = 'B';
            signalData[trgPos--] = 'A';
            signalData[trgPos--] = 'B';
            signalLen += 2;
        } else {
            signalData[trgPos--] = signalData[i];
        }
    }
}

```

```
66 int matchString(char *str)
67 {
68     uint8_t *p = signalData;
69     while (*p || *str)
70         if (*p++ != *str++)
71             return 0;
72     return 1;
73 }
```

block32_ir2.c

Diese Darstellung eignet sich auch als Anleitung um die IR-Signale selbst zu senden:

```
11 void waitSteps(uint8_t n)
12 {
13     for (uint8_t i = 0; i < n; i++)
14         _delay_us(27);
15 }
16
17 void irSend(char *code)
18 {
19     bool on_state = 0;
20     while (*code) {
21         on_state = !on_state;
22         if (on_state)
23             irsend_on();
24         else
25             irsend_off();
26         switch (*code) {
27         case 'A': waitSteps(8); break;
28         case 'B': waitSteps(28); break;
29         case 'C': waitSteps(62); break;
30         }
31         code++;
32     }
33     irsend_off();
34     _delay_ms(1);
35 }
```

block32_ir3.c

Hier ist auffällig, dass die Funktion `waitSteps()` nicht 10 μs sondern 27 μs für jeden Zeitschritt wartet. Die 17 zusätzlichen μs kommen daher, dass die Empfangs-Routine neben dem `_delay_us(10);` eben auch den eigentlichen Programmcode ausführen muss. Messungen mit einem Oszilloskop haben ergeben, dass um das gleiche Timing beim Senden und Empfangen zu erreichen bei senden zusaetzliche 17 μs gewartet werden müssen.

Übung

Vervollständigen Sie diese Codefragmente zu einem sinnvollen Programm.

Diskussion

Welchen Vorteil hat es ASCII-Zeichen als Symbole für die drei Kategorien zu verwenden?

Übung

Recherchieren Sie im Internet zu den verschiedenen Protokollen die für IR-Fernbedienungen verwendet werden. Insbesondere von Interesse ist das „RC-5“ Protokoll (nicht zu verwechseln mit dem „RC5“ Kryptocypher).

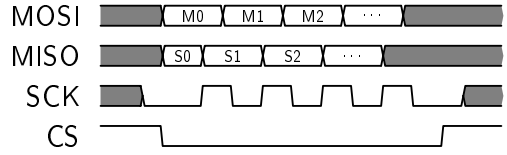
Block 33: SPI I/O-Expander

Ein I/O-Expander ist ein Bauteil, das über wenige Steuerleitungen vom Microcontroller angesteuert werden kann und (im Vergleich zur Anzahl der Steuerleitungen) viele I/O-Pins zur Verfügung stellt. Das ist nützlich wenn die I/O-Pins am Microcontroller ausgehen.

Der I/O-Expander auf unserer Experimentierplatine wird über ein SPI-Interface angesprochen. Das SPI-Interface verwendet 4 Leitungen:

- MOSI – Daten vom Microcontroller zum I/O-Expander
- MISO – Daten vom I/O-Expander zum Microcontroller
- SCK – Taktleitung
- CS – Aktivieren des SPI-Interfaces für einen Transfer

Ein SPI Datentransfer läuft folgendermaßen ab:



D.h. über die fallende Flanke von CS wird der Anfang eines neuen Transfers markiert und CS bleibt auf 0 bis das Ende des Transfers erreicht ist.

Die Bits werden mit der steigenden Flanke von SCK übertragen. D.h. die Daten auf MOSI müssen vor der steigenden Flanke von SCK geschrieben und die Daten auf MISO vor der steigenden Flanke von SCK gelesen werden. Dabei wird das höchstwertigste Bit jedes Bytes jeweils. zuerst übertragen.

Diskussion

Kann man bei SPI ein Byte vom Master (Microcontroller) zum Slave (I/O-Expander) übertragen ohne gleichzeitig ein Byte vom Slave zum Master zu übertragen? Wie müssen Protokolle, die auf SPI aufbauen, daher gestaltet und spezifiziert werden?

Übung

LED D6 am Experimentierboard wird über den I/O-Expander angesteuert. Zur aktivierung des entsprechenden I/O-Pins muss folgende Sequenz an den I/O-Expander gesendet werden:

```
0x40 0x00 0xEF
```

Danach kann die LED mit folgender Sequenz eingeschaltet werden:

```
0x40 0x12 0x10
```

Und folgende Sequenz schaltet die LED wieder aus:

```
0x40 0x12 0x00
```

Schreiben Sie ein Programm das die LED blinken lässt.

Diskussion

Was sind die Vor- und Nachteile eines I/O-Expanders gegenüber den I/O-Pins direkt am Microcontroller?

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

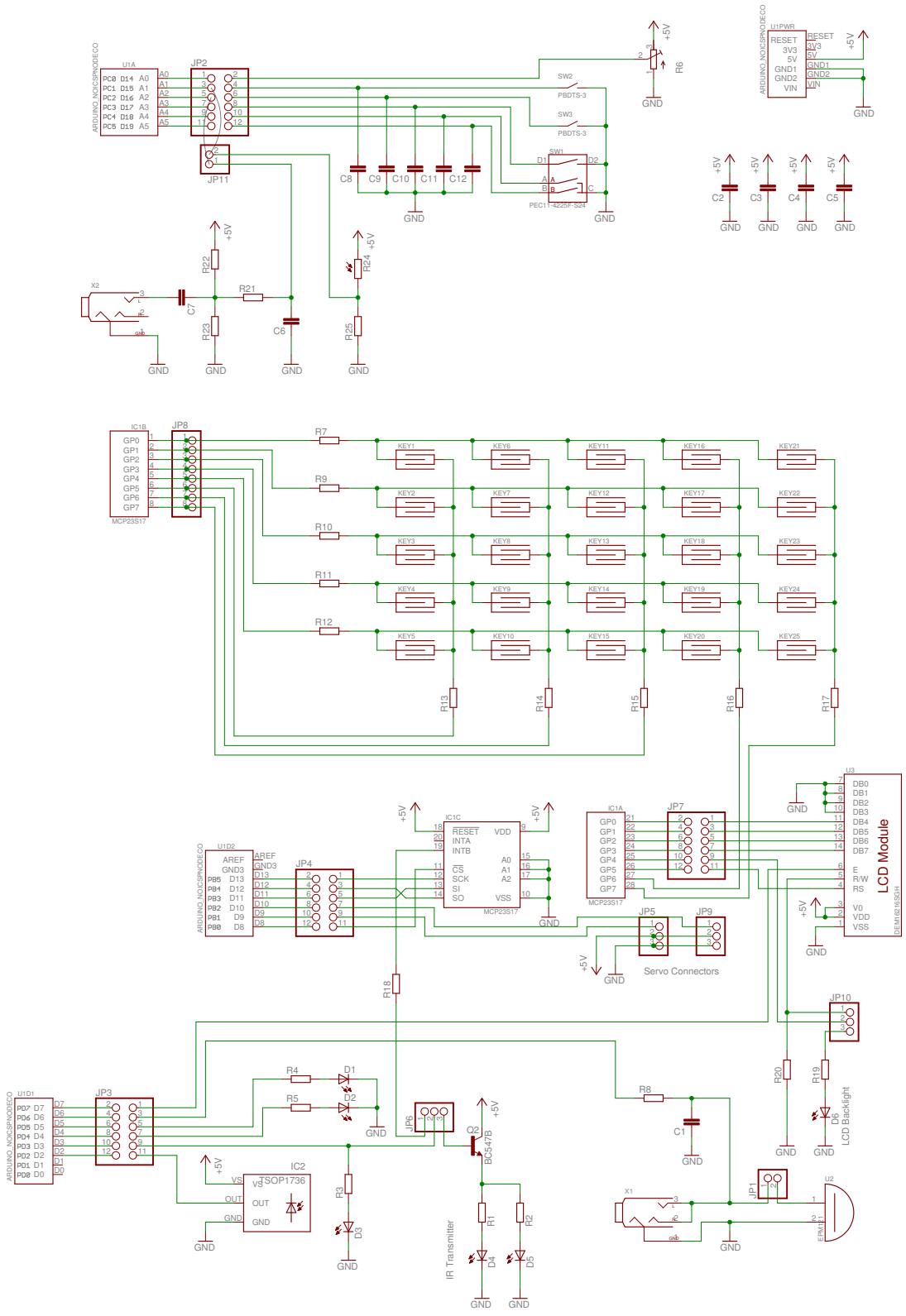
Das Arduino-Shield für den Kurs hat folgende Pinbelegung:

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 $\overline{\text{CS}}$
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	Rotory Encoder Button
PC4	A4 / 18	Rotory Encoder Bit A
PC5	A5 / 19	Rotory 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
GPA0	LCD Modul DB4
GPA1	LCD Modul DB5
GPA2	LCD Modul DB6
GPA3	LCD Modul DB7
GPA4	LCD Modul R/W
GPA5	LCD Modul RS
GPA6	Tastenfeld Spalte 4
GPA7	Tastenfeld Spalte 5

I/O-Pin	Funktion
GPB0	Tastenfeld Zeile 1
GPB1	Tastenfeld Zeile 2
GPB2	Tastenfeld Zeile 3
GPB3	Tastenfeld Zeile 4
GPB4	Tastenfeld Zeile 5
GPB5	Tastenfeld Spalte 1
GPB6	Tastenfeld Spalte 2
GPB7	Tastenfeld Spalte 3


LCD Module
Servo Connectors
LCD Backlight
IR Transmitter
TSOP1736
BC547B
PBDTS-3
PEC11-4225F-S24
MCP23S17
DEM16216SGH
GND
+5V

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 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.