PICOBOY / COLOR PLUS

BAND 04

# Programmieren in C

## Spiele, Grafik und Hardware mit dem Pico SDK




**FÜR EINSTEIGER OHNE VORKENNTNISSE**

DIE PROGRAMMIERREIHE FÜR DEN PICOBOY COLOR PLUS

Dieses Buch wurde mit künstlicher Intelligenz (KI) generiert. Die Programme wurden getestet – Fehler sind trotzdem möglich.

# Bevor es losgeht

In diesem Buch lernst du, in **C** zu programmieren – der Sprache, in der Betriebssysteme, Treiber und fast alle Mikrocontroller programmiert werden. Du lernst sie an einem kleinen Gerät, das du in der Hand halten kannst: dem **PicoBoy Color Plus** mit Farbbildschirm, Joystick, Tasten, LEDs, Lautsprecher und Neigungssensor. So siehst du sofort, was dein Programm tut.

Du brauchst keine Vorkenntnisse. Am Anfang schreibst du Texte auf den Bildschirm, dann rechnest du, zeichnest, reagierst auf Tasten und lässt Figuren über den Bildschirm laufen. Im letzten Teil schaust du unter die Haube und steuerst die Hardware direkt: Pins, PWM, I2C, SPI, DMA, Interrupts und den zweiten Prozessorkern. Am Ende baust du ein eigenes Projekt – auf Wunsch mit Hilfe von KI.

## So ist das Buch aufgebaut

Jedes Kapitel hat denselben Aufbau:

- **Worum geht's?** – was du in dem Kapitel lernst.
- **Erklärungen** mit kurzen Programmen. Jedes Programm hat Zeilennummern, und der Text erklärt, was in welcher Zeile passiert. Daneben siehst du, was auf dem Bildschirm des PicoBoy erscheint, oder darunter, was der Computer ausgibt.
- **Ausprobieren** – ein etwas größeres Programm, das das Kapitel zusammenfasst.
- **Übungsaufgaben** mit Sternen: ⭐ zum Einsteigen, ⭐⭐ wenn du weiter willst, ⭐⭐⭐ zum Knobeln. Die Lösungen stehen in Anhang C. Schau erst nach, wenn du es selbst versucht hast!
- **Das Wichtigste in Kürze** – zum Wiederholen.

Unterwegs findest du farbige Kästen:

> **Tipp:** Hier steht, wie etwas leichter geht.

> **Achtung:** Hier steht, worauf du aufpassen musst – oft mit der Fehlermeldung, die dann erscheint.

> **Info:** Hier steht etwas, das gut zu wissen ist.

## Was du brauchst

- einen Computer mit Windows, macOS oder Linux
- **VS Code** mit der Erweiterung **Raspberry Pi Pico** (Kapitel 2)
- die **Buchvorlage** `pbc-vorlage.zip` mit der Buchbibliothek `pbc`
- einen **PicoBoy Color Plus** mit USB-Kabel, das auch Daten überträgt

Alle Programme aus dem Buch liegen als `.c`-Dateien bei. Den Dateinamen findest du über jedem Programm. Abtippen lohnt sich trotzdem: Beim Tippen lernst du die Sprache am schnellsten – und nebenbei, Tippfehler zu finden. Jedes Programm ist eine vollständige `main.c` für die Vorlage.

## Die Teile des Buchs

| Teil | Darum geht es |
|---|---|
| **I – Los geht's** | Gerät, C und das Pico SDK, Werkzeuge, das erste Programm, die Konsole |
| **II – Grundlagen von C** | Variablen, Zeichnen, Schleifen, Entscheidungen, Tasten, LEDs und Töne, Funktionen, Arrays, Strukturen |
| **III – Spiele bauen** | flimmerfreie Animation, Bilder, Kollisionen, Neigungssensor, ein ganzes Spiel |
| **IV – Hinter die Kulissen** | Zeiger, eigene Bibliotheken, GPIO, PWM, ADC, I2C, SPI, DMA, Interrupts, zweiter Kern, Flash – und Programmieren mit KI |

Für dieses Buch brauchst du das Pico SDK 2.2 und VS Code.

Viel Spaß beim Programmieren!

# Inhalt


**Teil I – Los geht's**

1 Mikrocontroller, C und das Pico SDK ..... 5
2 Die Werkzeuge einrichten ..... 10
3 Das erste Programm ..... 16
4 Die Konsole: `printf` über USB ..... 27

**Teil II – Grundlagen von C**

5 Variablen und Datentypen ..... 35
6 Zeichnen und Farben ..... 50
7 Schleifen und Zeit ..... 60
8 Entscheidungen ..... 71
9 Tasten und Joystick ..... 84
10 LEDs, RGB-LED und Töne ..... 97
11 Funktionen ..... 109
12 Arrays und Texte ..... 124
13 Strukturen ..... 141

**Teil III – Spiele bauen**

14 Flimmerfrei animieren ..... 154
15 Bilder und Sprites ..... 167
16 Kollisionen ..... 177
17 Der Neigungssensor ..... 188
18 Projekt: Ein ganzes Spiel ..... 198

**Teil IV – Hinter die Kulissen**

19 Zeiger und Speicher ..... 210
20 Mehrere Dateien und eigene Bibliotheken ..... 229
21 GPIO, PWM und ADC – die Hardware direkt ..... 246
22 I2C: mit dem Neigungssensor sprechen ..... 259
23 SPI, DMA und das Display ..... 268
24 Interrupts, Timer und der zweite Kern ..... 278
25 Daten dauerhaft speichern ..... 291
26 Dein eigenes Projekt ..... 299
27 Programmieren mit KI: Vibecoding ..... 305

**Anhang**

A Die Befehle der Buchbibliothek `pbc` ..... 316
B Fehlermeldungen des Compilers und was sie bedeuten ..... 321
C Lösungen der Übungsaufgaben ..... 333
D Hardware-Referenz ..... 431
E Installation im Detail ..... 436

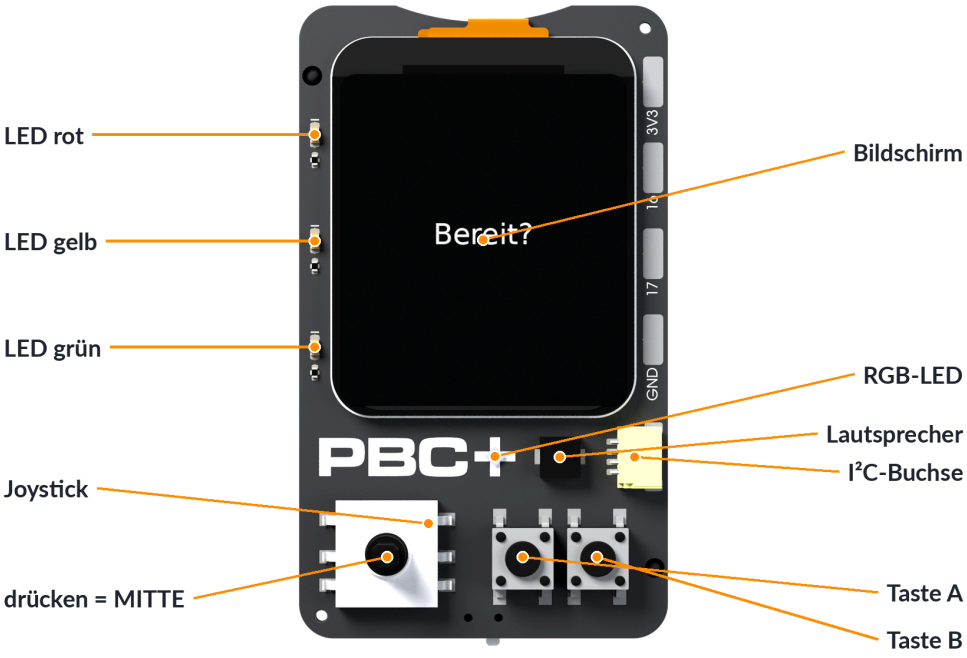
KAPITEL 1

# Mikrocontroller, C und das Pico SDK

**Worum geht's?** Bevor du die erste Zeile schreibst, lernst du dein Gerät kennen und erfährst, was mit einem C-Programm passiert, bis es auf dem PicoBoy läuft. In diesem Kapitel wird noch nichts installiert und nichts programmiert. Es geht um die Begriffe, die dir im ganzen Buch begegnen.

## 1.1 Das Gerät

Der **PicoBoy Color Plus** (kurz: PBC+) sieht aus wie eine kleine Spielkonsole. Im Inneren steckt ein **Mikrocontroller**: ein vollständiger Computer auf einem einzigen Chip, mit Prozessor, Arbeitsspeicher und Anschlüssen für Tasten, Bildschirm und Lautsprecher. Mikrocontroller stecken in Waschmaschinen, Fahrradcomputern, Fernbedienungen und Autos, oft Dutzende in einem einzigen Gerät.




| Teil | Wofür? |
|---|---|
| **Bildschirm** | Farbbildschirm, 240 Pixel breit und 280 Pixel hoch |
| **Joystick** | kleiner Steuerknüppel für hoch, runter, links und rechts. Drückst du ihn hinein, ist das die Taste **MITTE**. |
| **Taste A und Taste B** | zwei Knöpfe, z. B. zum Springen oder Bestätigen |
| **LEDs rot, gelb, grün** | drei kleine Lämpchen neben dem Bildschirm |
| **RGB-LED** | ein Lämpchen, das in jeder Farbe leuchten kann |

| Teil | Wofür? |
|---|---|
| **Lautsprecher** | spielt Töne und Melodien |
| **Neigungssensor** | misst, wie das Gerät gehalten und gekippt wird |
| **I²C-Buchse** | weiße Buchse, passend für STEMMA QT und Qwiic: Sensoren und andere I²C-Module anstecken, ohne Löten |
| **Batterien** | zwei AAA-Batterien auf der Rückseite versorgen das Gerät ohne Kabel |
| **Schalter BAT/USB** | unten auf der Rückseite: wählt, woher der Strom kommt – BAT von den Batterien, USB vom Kabel. Steht er auf USB und steckt kein Kabel, ist der PBC+ aus. |
| **Knöpfe BOOT und RES** | auf der Rückseite, direkt über dem Schalter: zum Aufspielen neuer Programme (Abschnitt 1.3) |

Der Mikrocontroller des PBC+ heißt **RP2350** und stammt von Raspberry Pi. Ein paar Zahlen, damit du ihn einordnen kannst:

| | RP2350 im PBC+ | zum Vergleich: ein Laptop |
|---|---|---|
| Prozessorkerne | 2, je 150 Millionen Takte pro Sekunde | 4 bis 16, je etwa 3 Milliarden |
| Arbeitsspeicher (RAM) | 520 Kilobyte | 8 bis 32 Gigabyte |
| Dauerhafter Speicher (Flash) | 16 Megabyte | 500 Gigabyte und mehr |

Der Arbeitsspeicher ist also etwa **zwanzigtausendmal kleiner** als bei einem Laptop. Das klingt nach wenig, reicht aber für erstaunlich viel, wenn man sparsam programmiert. Genau dafür ist die Sprache C gemacht.

## 1.2 Was ist ein Programm in C?

Ein Prozessor versteht nur **Maschinencode**: lange Folgen von Zahlen, die für einfache Befehle stehen wie „lies diese Speicherstelle“, „addiere“ oder „springe dorthin, wenn das Ergebnis null ist“. Kein Mensch schreibt so etwas gern von Hand.

Deshalb schreibst du dein Programm als **Quelltext**: als Text in einer **Programmiersprache**, die Menschen lesen können. Ein kleines Stück C sieht so aus:

```
pbc_clear(PBC_BLACK);
pbc_text24("Hallo!", 80, 120, PBC_WHITE);
pbc_show();
```

Ohne Vorkenntnisse kann man ahnen, was hier passiert: Der Bildschirm wird schwarz, dann erscheint „Hallo!“ in Weiß. Was die einzelnen Zeichen bedeuten, lernst du in Kapitel 3.

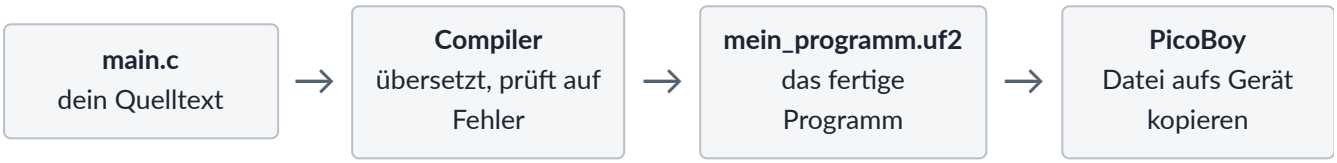
Aus dem Quelltext macht ein **Compiler** (deutsch: Übersetzer) den Maschinencode. Der Compiler ist selbst ein Programm. Er läuft auf deinem Computer, liest den Quelltext, prüft ihn auf Fehler und schreibt das übersetzte Programm in eine Datei.

**C** ist eine der ältesten Programmiersprachen, die heute noch viel benutzt werden. Sie entstand Anfang der 1970er-Jahre und steckt bis heute in fast allem, was nah an der Hardware ist: in Betriebssystemen wie Linux und Windows, in Treibern, in Datenbanken und in nahezu jedem Mikrocontroller. Wer C kann, versteht, wie ein Computer im Inneren arbeitet.

> Wer schon Python vom PicoBoy kennt, findet die Befehle für Display, Tasten und Ton unter denselben Namen wieder: Aus `pbc.text24(…)` wird `pbc_text24(…)`, aus `pbc.WHITE` wird `PBC_WHITE`. Dieses Buch setzt Python aber nicht voraus.

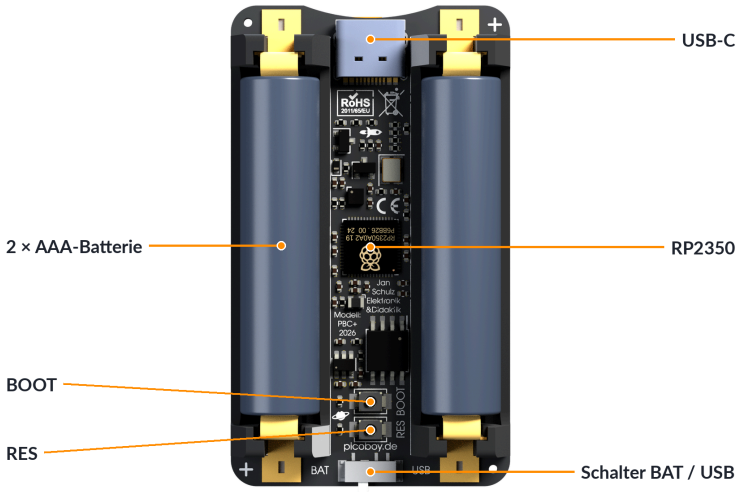
## 1.3 Vom Quelltext zur `.uf2`-Datei und aufs Gerät

Vom Quelltext bis zum laufenden Programm sind es immer dieselben Schritte:




1. Du schreibst den Quelltext in die Datei `main.c`. Die Endung `.c` steht für die Sprache C.
2. Der Compiler übersetzt ihn. Dieser Schritt heißt **Bauen** (englisch: *build* oder *compile*). Findet der Compiler einen Fehler, bricht er ab und sagt dir, in welcher Zeile.
3. Am Ende steht eine Datei mit der Endung `.uf2`. Sie enthält das ganze Programm, fertig zum Aufspielen.
4. Du bringst den PBC+ in den **Bootmodus**. Dann meldet er sich am Computer wie ein USB-Stick, als neues Laufwerk. Du kopierst die `.uf2`-Datei darauf. Der PBC+ schreibt das Programm in seinen Flash-Speicher, startet neu und führt es aus.

**So kommt der PBC+ in den Bootmodus:** Das USB-Kabel steckt am Computer. Halte den Knopf **BOOT** auf der Rückseite gedrückt und tippe kurz auf **RES** (Reset, Neustart). Lass BOOT danach los. Am Computer erscheint ein neues Laufwerk. Den Weg kannst du dir als „BOOT halten, RES tippen“ merken.



Die Werkzeuge, die du in Kapitel 2 einrichtest, nehmen dir das Kopieren später ab: Ein Klick baut das Programm und spielt es auf.

## 1.4 Was in C anders ist

Vielleicht hast du schon in einer anderen Sprache programmiert, zum Beispiel in Python oder Scratch. Dann fallen dir bei C einige Unterschiede auf.

**Erst übersetzen, dann ausführen.** Python liest ein Programm Zeile für Zeile und führt es sofort aus. Ein C-Programm wird vorher vollständig übersetzt. Das hat zwei Folgen. Viele Fehler findet der Compiler schon, bevor das Programm überhaupt läuft. Und das übersetzte Programm ist sehr schnell, oft zehn- bis hundertmal schneller als dasselbe in Python.

**Dein Programm ist die ganze Firmware.** Auf dem PBC+ läuft kein Betriebssystem, das Programme startet und beendet. Wenn du ein C-Programm aufspielst, ersetzt es alles, was vorher darauf lief. Dein Programm ist dann das Einzige, was der Prozessor tut, vom Einschalten bis zum Ausschalten. Deshalb nennt man es **Firmware**.

**Du bist für alles zuständig.** C nimmt dir wenig ab. Es prüft zum Beispiel nicht, ob du über das Ende einer Liste hinausschreibst, und es räumt Speicher nicht selbst auf. Das macht C schnell und sparsam, verlangt aber Sorgfalt. Das Buch zeigt dir an jeder dieser Stellen, worauf du achten musst.

**Das Pico SDK.** Damit du nicht jeden Anschluss des Chips selbst ansteuern musst, gibt es von Raspberry Pi eine große Sammlung fertiger C-Befehle: das **Pico SDK** (*Software Development Kit*, also Werkzeugkasten für die Entwicklung). Es enthält zum Beispiel `sleep_ms` zum Warten und `printf` für Ausgaben.

**Die Buchbibliothek `pbc`.** Für Bildschirm, Tasten, LEDs und Ton bringt dieses Buch eine eigene kleine Sammlung von Befehlen mit, die **Buchbibliothek `pbc`**. Sie baut auf dem Pico SDK auf. Mit ihr schreibst du schon im ersten Programm Text auf den Bildschirm, ohne zu wissen, wie der Bildschirm-Chip angesteuert wird. In Teil IV des Buchs schaust du in die Bibliothek hinein und lernst, die Hardware direkt mit dem SDK zu steuern.

## 1.5 Zurück zu MicroPython oder einem Spiel

Auf einem neuen PBC+ ist meist **MicroPython** installiert, damit man ihn in Python programmieren kann. Sobald du dein erstes C-Programm aufspielst, ist MicroPython weg. Das ist kein Problem: Jede Firmware lässt sich jederzeit wieder aufspielen, auf demselben Weg wie dein Programm.

> Hast du auf dem PBC+ Python-Programme, Bilder oder andere Dateien gespeichert, die du behalten willst? Dann sichere sie **vorher**: In PBC+ Studio (pbcstudio.de) geht das über **Dateien → Flash-Backup erstellen (.uf2)…**. Das Backup ist selbst eine `.uf2`-Datei, die du später wieder aufspielen kannst.

- **MicroPython zurückholen:** In PBC+ Studio unter **Dateien → MicroPython installieren…** die Firmware herunterladen und wie in Abschnitt 1.3 auf das Laufwerk kopieren.
- **Ein fertiges Spiel aufspielen:** Auch Spiele für den PBC+ gibt es als `.uf2`-Datei. Es läuft immer nur eine Firmware zur selben Zeit.
- **Dein C-Programm wieder aufspielen:** Die `.uf2`-Datei bleibt auf deinem Computer. Du kannst sie jederzeit erneut kopieren.

Kaputt machen kannst du dabei nichts. Der Bootmodus steckt fest im Chip und lässt sich durch kein Programm überschreiben. Auch wenn ein Programm völlig falsch ist und der Bildschirm schwarz bleibt: BOOT halten, RES tippen, und der PBC+ wartet wieder auf eine neue Datei.

## Übungsaufgaben

**Aufgabe 1** ⭐ **Begriffe ordnen** Bringe in die richtige Reihenfolge und erkläre jeden Begriff in einem Satz: `.uf2`-Datei, Quelltext, Bootmodus, Compiler, Maschinencode.

**Aufgabe 2** ⭐ **Mikrocontroller im Alltag** Nenne fünf Geräte bei dir zu Hause, in denen vermutlich ein Mikrocontroller steckt. Woran erkennst du das?

**Aufgabe 3** ⭐⭐ **Speicher vergleichen** Ein Foto vom Handy hat etwa 3 Megabyte. Wie viele solcher Fotos passen in den Arbeitsspeicher (520 Kilobyte) des PBC+, wie viele in seinen Flash-Speicher (16 Megabyte)? (1 Megabyte = 1024 Kilobyte.)

**Aufgabe 4** ⭐⭐ **Was passiert, wenn …?** Ein Freund hat ein Programm aufgespielt, und seitdem bleibt der Bildschirm schwarz. Er fürchtet, das Gerät sei kaputt. Was sagst du ihm, und was soll er tun?

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- Der PBC+ hat einen **Mikrocontroller** RP2350: 2 Kerne, 520 KB Arbeitsspeicher, 16 MB Flash.
- Du schreibst **Quelltext** in C. Der **Compiler** übersetzt ihn in Maschinencode. Das heißt **Bauen**.
- Das Ergebnis ist eine **`.uf2`-Datei**. Sie kommt im **Bootmodus** auf den PBC+: BOOT halten, RES tippen, Datei auf das neue Laufwerk kopieren.
- Ein C-Programm ist die ganze **Firmware** des Geräts. Es ersetzt MicroPython; MicroPython lässt sich jederzeit wieder aufspielen.
- Das **Pico SDK** liefert allgemeine Befehle, die **Buchbibliothek `pbc`** die Befehle für Bildschirm, Tasten, LEDs und Ton.

KAPITEL 2

# Die Werkzeuge einrichten

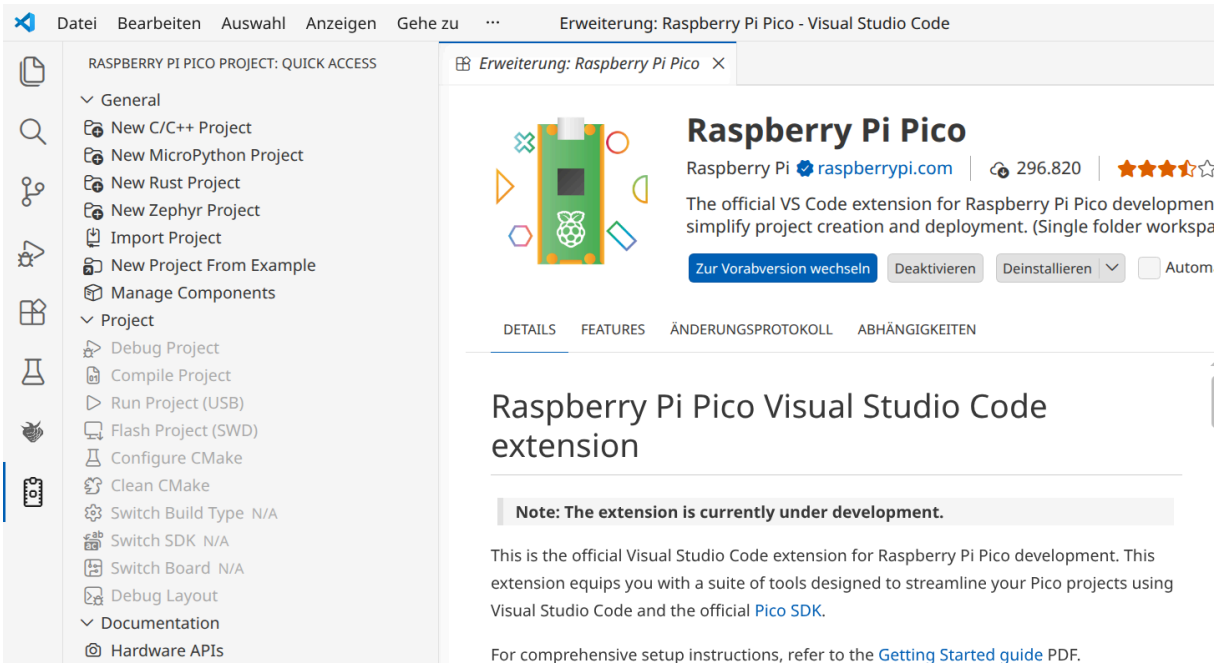
**Worum geht's?** Du installierst die Programmierumgebung VS Code mit der Erweiterung von Raspberry Pi, öffnest die Buchvorlage, baust das erste Programm und spielst es auf den PicoBoy. Das dauert beim ersten Mal etwa eine halbe Stunde, weil viel heruntergeladen wird. Danach ist alles mit zwei Klicks erledigt.

## 2.1 VS Code und die Erweiterung „Raspberry Pi Pico" installieren

**VS Code** (Visual Studio Code) ist ein kostenloser Editor für Quelltext von Microsoft. Er läuft unter Windows, macOS und Linux. Raspberry Pi bietet dafür eine **Erweiterung** an, die alles Weitere selbst besorgt: den Compiler, das Pico SDK und das Werkzeug zum Aufspielen.

1. Lade VS Code von **code.visualstudio.com** herunter und installiere es. Alle Voreinstellungen des Installationsprogramms passen.
2. Starte VS Code. Ist dein Computer auf Deutsch eingestellt, bietet VS Code rechts unten an, das deutsche Sprachpaket zu installieren; das ist praktisch, aber nicht nötig. Die Erweiterung von Raspberry Pi spricht ohnehin Englisch.
3. Klicke in der linken Leiste auf das Symbol **Erweiterungen** (vier Quadrate) oder drücke **Strg + Umschalt + X** (am Mac **Cmd + Umschalt + X**).
4. Suche nach **Raspberry Pi Pico**. Wähle die Erweiterung, deren Herausgeber **Raspberry Pi** ist, und klicke **Installieren**.
5. Suche außerdem nach **Serial Monitor** (Herausgeber Microsoft) und installiere auch diese Erweiterung. Sie kommt nicht von selbst mit. Mit ihr siehst du später die Ausgaben deiner Programme (Kapitel 4).

Die Pico-Erweiterung zieht weitere Erweiterungen nach: MicroPico, Python und Cortex-Debug. Links erscheinen deshalb mehrere neue Symbole. Das Symbol der Pico-Erweiterung ist ein Chip, die Himbeere gehört zu MicroPico.




Die Erweiterung „Raspberry Pi Pico“ in VS Code. Links ihr Bereich mit „Import Project“ und „Compile Project“.

Die Bildschirmfotos in diesem Buch sind unter Linux entstanden. Unter Windows und macOS sieht es ähnlich aus.

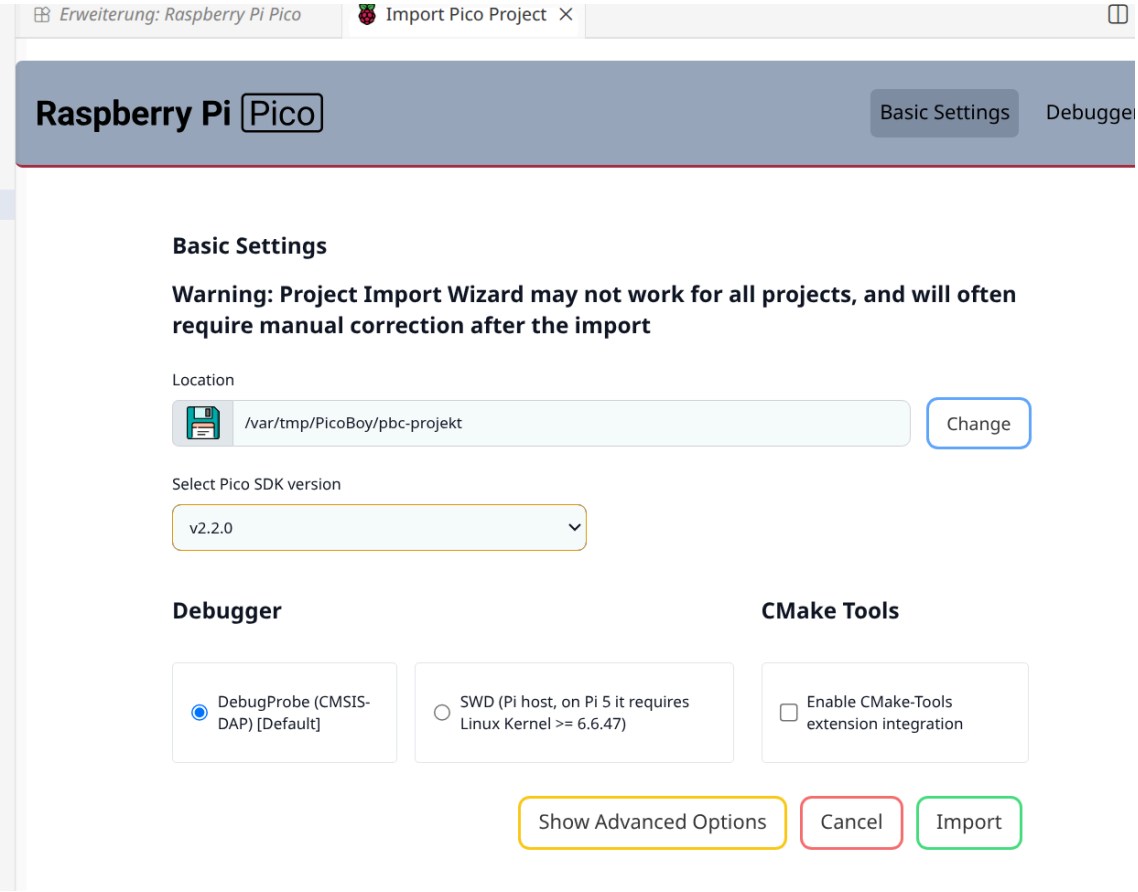
MicroPico kann sich automatisch mit dem PicoBoy verbinden. Gibt es dabei Probleme, schalte in den Einstellungen `micropico.autoConnect` ab.

> Unter **Windows** ist damit alles installiert. Unter **macOS** und **Linux** braucht die Erweiterung zusätzlich einige Werkzeuge des Betriebssystems. Was genau, steht in Anhang E.

## 2.2 Die Buchvorlage öffnen

Zu diesem Buch gehört eine **Vorlage**: ein fertiger Projektordner, in dem alles für den PicoBoy eingestellt ist. Du bekommst sie als ZIP-Datei `pbc-vorlage.zip` unter **pbcplus.de/downloads/pbc-vorlage.zip**.

1. Entpacke die ZIP-Datei an einen Ort, den du leicht wiederfindest. Am sichersten ist ein Pfad **ohne Leerzeichen und ohne Umlaute**, zum Beispiel `C:\PicoBoy\pbc-projekt` unter Windows oder `~/PicoBoy/pbc-projekt` unter macOS und Linux. Manche Werkzeuge des Compilers kommen mit Sonderzeichen im Pfad nicht zurecht.
2. Klicke in VS Code links auf das Symbol der Pico-Erweiterung und dann auf **Import Project**.
3. Wähle als Ort (*Location*) den Ordner `pbc-projekt` aus. Als Version des Pico SDK wähle **2.2.0**; voreingestellt ist 2.3.1. Die übrigen Einstellungen bleiben, wie sie sind. Klicke **Import**. Die Warnung „Project Import Wizard may not work for all projects“ kann erscheinen. Für die Vorlage ist sie harmlos.




Der Dialog „Import Project“: Ordner gewählt, Pico SDK 2.2.0 eingestellt.

Beim ersten Import lädt die Erweiterung das Pico SDK, den Compiler und weitere Werkzeuge herunter, zusammen mehrere hundert Megabyte. Das dauert je nach Internetverbindung einige Minuten. Unten rechts siehst du, wie weit es ist. Danach öffnet VS Code den Ordner.

Links im **Explorer** siehst du jetzt die Dateien der Vorlage:

| Datei oder Ordner | Was steht darin? |
|---|---|
| `main.c` | **dein Programm**. Nur diese Datei änderst du in den Teilen I bis III. |
| `pbc/` | die Buchbibliothek: Befehle für Bildschirm, Tasten, LEDs und Ton |
| `CMakeLists.txt` | die Bauanleitung: welche Dateien zum Programm gehören und wie es heißt |
| `pico_sdk_import.cmake` | sucht das Pico SDK; gehört zum SDK, nicht ändern |
| `bild2c.html` | wandelt Bilder in C-Code um (Kapitel 15); öffnet sich im Browser |
| `AGENTS.md`, `CLAUDE.md` | Hinweise für KI-Assistenten (Kapitel 27); für dich nicht wichtig |
| `.vscode/` | Einstellungen, die die Erweiterung beim Import angelegt hat |
| `build/` | entsteht beim ersten Bauen; hier landet die `.uf2`-Datei |

Öffne `main.c` mit einem Doppelklick. Darin steht schon ein kleines Programm:

k03_hallo.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_text24("Hallo!", 80, 120, PBC_WHITE);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

Was die Zeilen bedeuten, lernst du in Kapitel 3. Jetzt geht es erst einmal darum, dass es auf den PicoBoy kommt.

## 2.3 Bauen – was dabei passiert

Klicke unten in der Statusleiste auf **Compile** (oder im Bereich der Pico-Erweiterung auf **Compile Project**). Im unteren Teil von VS Code öffnet sich das **Terminal** und zeigt, was passiert. Beim ersten Mal werden neben deinem Programm auch die Buchbibliothek und viele Teile des Pico SDK übersetzt. Die Fortschrittszeile überschreibt sich dabei immer wieder selbst, sodass am Ende nur rund 5 Zeilen stehen. Das dauert etwa eine Minute. Beim nächsten Mal wird nur übersetzt, was sich geändert hat, und es geht in wenigen Sekunden.

Hat alles geklappt, beginnt die letzte Zeile mit einem Zähler wie `[161/161]` und endet mit `Linking CXX executable mein_programm.elf`, und nirgends steht `error`. Im Ordner `build` liegt jetzt die Datei **`mein_programm.uf2`**.




Nach dem Bauen: unten in der Statusleiste Compile und Run, darüber das Terminal mit der letzten Zeile des Compilers.

Beim Bauen passieren nacheinander drei Dinge, die du im Terminal wiedererkennst:

1. **Konfigurieren:** Das Werkzeug **CMake** liest die Bauanleitung `CMakeLists.txt` und plant, welche Dateien in welcher Reihenfolge übersetzt werden müssen. Das passiert nur beim ersten Bauen und wenn sich die Bauanleitung ändert.

2. **Übersetzen:** Der Compiler macht aus jeder `.c`-Datei Maschinencode. Zeilen wie `Building C object …` gehören dazu.
3. **Binden:** Der **Linker** (deutsch etwa: Verbinder) fügt alle übersetzten Teile zu einem einzigen Programm zusammen. Daraus entsteht die `.uf2`-Datei. Im Terminal steht dafür `Linking …`.

## 2.4 Aufspielen

Es gibt zwei Wege. Den ersten brauchst du nur beim allerersten Mal und wenn gar nichts mehr geht.

**Weg 1: per Bootmodus und Kopieren.**

1. Schließe den PicoBoy mit dem USB-Kabel an den Computer an.
2. Halte auf der Rückseite **BOOT** gedrückt, tippe kurz **RES**, lass BOOT los. Ein neues Laufwerk erscheint.
3. Ziehe die Datei `build/mein_programm.uf2` auf dieses Laufwerk. Du kannst sie dafür im Explorer von VS Code mit der rechten Maustaste anklicken und **Im Datei-Explorer anzeigen** wählen (englisch **Reveal in File Explorer**, am Mac **Im Finder anzeigen** bzw. **Reveal in Finder**).
4. Das Laufwerk verschwindet von selbst, der PicoBoy startet neu und zeigt „Hallo!“.

**Weg 2: mit einem Klick.** Klicke in der Statusleiste auf **Run** (oder **Run Project (USB)**). VS Code baut das Programm, falls nötig, und spielt es über das Kabel auf. Das klappt, wenn der PicoBoy im Bootmodus ist **oder** gerade ein Programm aus diesem Buch läuft. Die Vorlage ist so eingestellt, dass der PicoBoy über das Kabel auf diesen Befehl hört und sich selbst in den Bootmodus schaltet. Du musst also nur beim ersten Mal BOOT und RES drücken. Danach reicht **Run**.




> So arbeitest du ab jetzt: **`main.c` ändern → speichern (Strg + S) → Run.** Nach wenigen Sekunden läuft das neue Programm auf dem PicoBoy.

## 2.5 Wenn es nicht klappt

| Was passiert? | Was hilft? |
|---|---|
| **Import Project** meldet einen Fehler beim Herunterladen. | Internetverbindung prüfen, VS Code neu starten und den Import wiederholen. Bereits Geladenes wird nicht noch einmal geladen. |
| Beim Bauen erscheint `No such file or directory` mit einem Pfad. | Liegt der Projektordner in einem Pfad mit Leerzeichen oder Umlauten? Ordner verschieben (Abschnitt 2.2), VS Code neu öffnen. |

| Was passiert? | Was hilft? |
|---|---|
| Das Terminal zeigt `error:` und eine Zeilennummer. | Ein Fehler in deinem Programm. Wie du ihn liest, steht in Kapitel 3.5. |
| **Run** meldet, es sei kein Gerät gefunden worden. | Läuft auf dem PicoBoy noch MicroPython oder ein anderes Programm, das nicht aus diesem Buch stammt? Dann einmal über den Bootmodus aufspielen (Weg 1). Auch ein reines Ladekabel ohne Datenleitungen kann die Ursache sein. |
| Nach dem Aufspielen bleibt der Bildschirm schwarz. | Hat das Programm `pbc_show()`? (Kapitel 3.2) Sonst: BOOT halten, RES tippen und das Vorlagenprogramm erneut aufspielen. |
| Das Laufwerk erscheint nicht. | BOOT wirklich festhalten, während du RES tippst. Anderes USB-Kabel oder anderen Anschluss probieren. |

Lass die Vorlage `pbc-projekt` als Sicherung unverändert und arbeite in einer **Kopie**: Ordner kopieren, umbenennen (z. B. `uebungen`) und die Kopie mit **Import Project** öffnen. Für ein neues Programm genügt es, den Inhalt von `main.c` zu ersetzen.

## Übungsaufgaben

**Aufgabe 1 ⭐ Ein anderer Gruß** Ändere in `main.c` das Wort `Hallo!` in `Moin!` oder einen anderen kurzen Gruß. Speichere, baue und spiele das Programm mit **Run** auf.

**Aufgabe 2 ⭐ Wie lange dauert es?** Miss mit der Uhr, wie lange das erste Bauen dauert und wie lange das zweite Bauen nach einer kleinen Änderung. Woher kommt der Unterschied?

**Aufgabe 3 ⭐⭐ Hin und zurück** Spiele MicroPython wieder auf (Kapitel 1.5) und danach wieder dein C-Programm. Welcher Weg aus Abschnitt 2.4 funktioniert, solange MicroPython läuft, welcher nicht?

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- **VS Code** mit der Erweiterung **Raspberry Pi Pico** ist deine Programmierumgebung. Die Erweiterung lädt Compiler und Pico SDK selbst.
- Die Buchvorlage öffnest du mit **Import Project**. Dein Programm steht in **`main.c`**.
- **Compile** baut das Programm, das Ergebnis ist `build/mein_programm.uf2`.
- **Run** baut und spielt auf. Beim allerersten Mal: BOOT halten, RES tippen, `.uf2` aufs Laufwerk kopieren.
- Arbeitsablauf: **ändern → speichern → Run**.

KAPITEL 3

# Das erste Programm

**Worum geht's?** Du verstehst Zeile für Zeile, wie ein C-Programm aufgebaut ist, schreibst Texte in verschiedenen Größen und Farben auf den Bildschirm und lernst, wie der PicoBoy Positionen angibt. Außerdem liest du Fehlermeldungen des Compilers. Das wirst du beim Programmieren ständig brauchen.

## 3.1 Der Aufbau eines C-Programms

Das ist das Programm aus der Vorlage, mit Zeilennummern:

k03_hallo.c

```
 1 #include "pbc.h"
 2
 3 int main(void) {
 4     pbc_init();
 5     pbc_clear(PBC_BLACK);
 6     pbc_text24("Hallo!", 80, 120, PBC_WHITE);
 7     pbc_show();
 8     while (true) {
 9         sleep_ms(1000);
10     }
11 }
```




Die Zeilennummern gehören nicht zum Programm. VS Code zeigt sie links neben dem Text an, damit du dich orientieren kannst.

**Zeile 1:** `#include "pbc.h"` holt die Buchbibliothek in dein Programm. Ohne diese Zeile kennt der Compiler keinen einzigen Befehl für den PicoBoy. Die Datei `pbc.h` liegt im Ordner `pbc` der Vorlage. Sie enthält eine Liste aller Befehle der Bibliothek, damit der Compiler weiß, dass es sie gibt und wie man sie benutzt. Zeilen, die mit `#` beginnen, sind Anweisungen an den Compiler selbst, noch bevor er übersetzt.

**Zeile 3:** `int main(void) {` ist der Beginn der **Hauptfunktion**. Jedes C-Programm hat genau eine Funktion mit dem Namen `main`. Mit ihr fängt das Programm an. Was `int` und `void` bedeuten, lernst du in Kapitel 11. Bis dahin schreibst du diese Zeile einfach immer so.

**Die geschweiften Klammern `{` und `}`** fassen zusammen, was zur Funktion `main` gehört: alles von der `{` in Zeile 3 bis zur `}` in Zeile 11. Ein solcher Abschnitt heißt **Block**. Blöcke kommen in C ständig vor, auch ineinander verschachtelt: Die Zeilen 8 bis 10 bilden einen Block innerhalb von `main`.

**Zeilen 4 bis 7** sind **Anweisungen**. Der Prozessor führt sie von oben nach unten aus, eine nach der anderen. Jede Anweisung endet mit einem **Semikolon** `;`. Das Semikolon ist für C so wichtig wie der Punkt am Ende eines Satzes. Fehlt es, versteht der Compiler das Programm nicht mehr

(Abschnitt 3.5).

**Die Einrückung**, also die vier Leerzeichen am Zeilenanfang, ist für den Compiler bedeutungslos. Er würde das Programm auch in einer einzigen Zeile verstehen. Für Menschen ist sie aber wichtig: Man sieht sofort, was zu welchem Block gehört. VS Code rückt beim Tippen von selbst ein.

## 3.2 Den Bildschirm füllen und Text schreiben

`pbc_init();` in Zeile 4 richtet den PicoBoy ein: Bildschirm, Tasten, LEDs, Lautsprecher und die Verbindung zum Computer. Diese Zeile steht in jedem Programm in diesem Buch als **erste** Anweisung in `main`.

`pbc_clear(PBC_BLACK);` füllt den ganzen Bildschirm schwarz. Die runden Klammern zeigen, dass hier ein Befehl **aufgerufen** wird. Solche Befehle heißen **Funktionen**. Was in den Klammern steht, sind die **Argumente**: die Angaben, die die Funktion zum Arbeiten braucht. `pbc_clear` braucht eine Farbe. `pbc_init` braucht nichts, die Klammern müssen trotzdem dastehen.

`pbc_text24("Hallo!", 80, 120, PBC_WHITE);` schreibt einen Text. Die vier Argumente sind durch Kommas getrennt:

| Argument | Bedeutung |
|---|---|
| `"Hallo!"` | der Text, in Anführungszeichen |
| `80` | wie weit von links (x) |
| `120` | wie weit von oben (y) |
| `PBC_WHITE` | die Farbe |

Die Reihenfolge der Argumente ist fest. Texte stehen in C in geraden Anführungszeichen `"…"`, wie sie die Tastatur mit **Umschalt + 2** schreibt. Ein Text in Anführungszeichen heißt **Zeichenkette** (englisch *string*).

**`pbc_show();` bringt das Bild auf den Bildschirm.** Die Befehle zum Zeichnen malen nämlich zuerst in ein unsichtbares Bild im Arbeitsspeicher, den **Bildpuffer**. Erst `pbc_show()` schickt das fertige Bild auf einmal an den Bildschirm. Fehlt diese Zeile, bleibt der Bildschirm schwarz:

k03_ohne_show.c

```
 1 #include "pbc.h"
 2
 3 int main(void) {
 4     pbc_init();
 5     pbc_clear(PBC_BLACK);
 6     pbc_text24("Hallo!", 80, 120, PBC_WHITE);
 7     while (true) {
 8         sleep_ms(1000);
 9     }
10 }
```

Das Programm baut ohne Fehler und läuft auch. Es zeichnet „Hallo!“ in den Bildpuffer, zeigt ihn aber nie. Warum dieser Umweg sinnvoll ist, merkst du in Kapitel 7, wenn sich etwas bewegt.

> Erst zeichnen, dann **`pbc_show()`**. Was nach dem letzten `pbc_show()` gezeichnet wird, bleibt unsichtbar.

**Farben:** Die Bibliothek kennt elf fertige Farben: `PBC_BLACK`, `PBC_WHITE`, `PBC_RED`, `PBC_GREEN`, `PBC_BLUE`, `PBC_YELLOW`, `PBC_CYAN`, `PBC_MAGENTA`, `PBC_ORANGE`, `PBC_PURPLE` und `PBC_GRAY`. Gibst du `pbc_clear` eine andere Farbe, wird der Bildschirm in dieser Farbe gefüllt:

k03_farben.c

```c
#include "pbc.h"

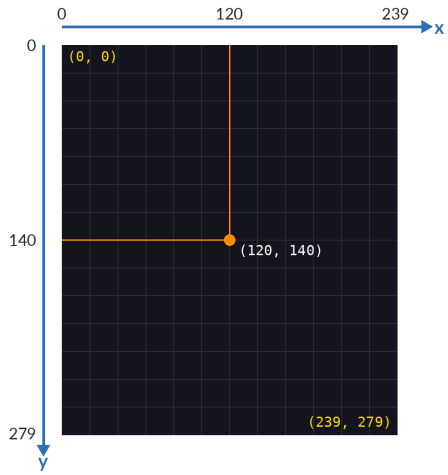
int main(void) {
    pbc_init();
    pbc_clear(PBC_BLUE);
    pbc_text24("Blau", 30, 40, PBC_WHITE);
    pbc_text24("Gelb", 30, 90, PBC_YELLOW);
    pbc_text24("Orange", 30, 140, PBC_ORANGE);
    pbc_text24("Cyan", 30, 190, PBC_CYAN);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```




Namen wie `PBC_WHITE` sind in Großbuchstaben geschrieben. In C ist das üblich für feste Werte, die sich nie ändern. Eigene Farben mischst du in Kapitel 6.

## 3.3 Das Koordinatensystem und die Schriftgrößen

Jede Stelle auf dem Bildschirm hat zwei Zahlen, ihre **Koordinaten**: **x** gibt an, wie weit rechts, **y**, wie weit unten. Der Punkt (0, 0) liegt in der **linken oberen Ecke**.



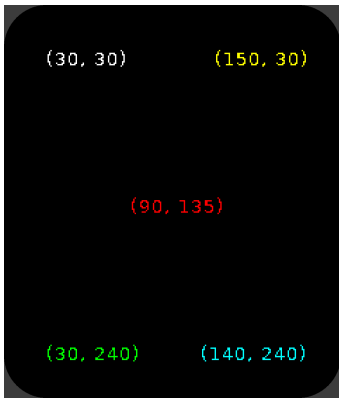
Anders als im Mathematikunterricht wächst y also **nach unten**. Das ist bei fast allen Bildschirmen so, denn ein Bild wird von oben nach unten aufgebaut, Zeile für Zeile. Der Bildschirm ist 240 Pixel breit und 280 Pixel hoch. x geht von 0 bis 239, y von 0 bis 279.

Bei einem Text gibt (x, y) die **linke obere Ecke** an:

k03_koordinaten.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_text12("(30, 30)", 30, 30, PBC_WHITE);
    pbc_text12("(150, 30)", 150, 30, PBC_YELLOW);
    pbc_text12("(30, 240)", 30, 240, PBC_GREEN);
    pbc_text12("(140, 240)", 140, 240, PBC_CYAN);
    pbc_text12("(90, 135)", 90, 135, PBC_RED);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```




> Der Bildschirm hat **runde Ecken**. Was näher als etwa 20 Pixel an einer Ecke liegt, ist nicht zu sehen. Halte mit wichtigen Dingen deshalb etwas Abstand zum Rand.

Für Text gibt es vier Größen. `pbc_text` hat eine feste Breite: Jedes Zeichen ist genau 8 Pixel breit und 8 Pixel hoch. Die drei anderen Schriften sind **proportional**: Ein „i“ ist schmaler als ein „m“, so wie in einem Buch. Die Zahl im Namen ist ungefähr die Höhe vom oberen Rand eines Großbuchstabens bis zur Unterkante von „g“ oder „y“. Über den Buchstaben bleiben noch 3 bis 5 Pixel frei: Mit y legst du die Oberkante der Zeile fest, nicht die der Buchstaben.

k03_groessen.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_text("text: Hallo", 30, 50, PBC_WHITE);
    pbc_text12("text12: Hallo", 30, 80, PBC_WHITE);
    pbc_text18("text18: Hallo", 30, 110, PBC_WHITE);
    pbc_text24("text24: Hallo", 30, 150, PBC_WHITE);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```




Ein Text, der über den rechten Rand hinausgeht, wird abgeschnitten. Er springt **nicht** in die nächste Zeile.

> Die Schriften kennen nur die Zeichen der englischen Tastatur. Umlaute und ß erscheinen als Fragezeichen. Schreib auf dem Bildschirm deshalb „ae“, „oe“, „ue“ und „ss“: `"Schoene Gruesse"`.

## 3.4 Kommentare

Alles hinter `//` bis zum Ende der Zeile ignoriert der Compiler. Solche Notizen heißen **Kommentare**. Sie erklären Menschen, was ein Programm tut. Für längere Kommentare gibt es `/*` … `*/`: Alles dazwischen ist Kommentar, auch über mehrere Zeilen.

k03_kommentare.c

```c
// Zeigt einen Gruss in zwei Farben.
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_text24("Guten", 40, 100, PBC_GREEN);   // erste Zeile
    pbc_text24("Morgen!", 40, 140, PBC_YELLOW);
    // pbc_text24("Abend!", 40, 180, PBC_RED);
    /* Dieser Kommentar
       geht ueber mehrere Zeilen. */
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

- Zeile 1 ist ein Kommentar über das ganze Programm.
- In Zeile 7 steht der Kommentar hinter einer Anweisung. Die Anweisung wird ausgeführt, der Kommentar nicht.
- Zeile 9 ist eine Anweisung, die „auskommentiert“ wurde. Sie bleibt im Programm stehen, wird aber nicht ausgeführt. So kannst du etwas vorübergehend abschalten, ohne es zu löschen.
- Die Zeilen 10 und 11 sind ein Kommentar über zwei Zeilen.




## 3.5 Fehlermeldungen des Compilers lesen

Beim Programmieren macht jeder Fehler, auch Profis, jeden Tag. Wichtig ist, dass du sie schnell findest. In C hilft dir der **Compiler** dabei: Er prüft das ganze Programm, bevor es auf den PicoBoy kommt. Findet er einen Fehler, bricht er das Bauen ab und schreibt eine **Fehlermeldung** ins Terminal von VS Code. Meist unterstreicht VS Code die Stelle zusätzlich im Editor rot.

### Beispiel 1: ein fehlendes Semikolon

k03_fehler_semikolon.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK)
    pbc_text24("Hallo!", 80, 120, PBC_WHITE);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:5:25: error: expected ';' before 'pbc_text24'
    5 |     pbc_clear(PBC_BLACK)
      |                         ^
      |                         ;
    6 |     pbc_text24("Hallo!", 80, 120, PBC_WHITE);
      |     ~~~~~~~~~~
```

So liest du die Meldung:

1. `main.c:5:25` nennt die **Datei**, die **Zeile** (5) und die **Spalte** (25), also das 25. Zeichen in der Zeile.
2. `error:` heißt: ein Fehler, das Programm kann nicht gebaut werden. Dahinter steht, was der Compiler erwartet hat: `expected ';' before 'pbc_text24'`, ein Semikolon vor `pbc_text24`.
3. Darunter zeigt der Compiler die Zeile und markiert die Stelle mit `^`. Hier schlägt er sogar vor, was fehlt: das `;`.

Im Terminal steht vor `main.c` noch der ganze Pfad deines Projektordners; im Buch ist er weggelassen. Hältst du **Strg** gedrückt (am Mac **Cmd**) und klickst auf die Angabe, springt der Cursor im Editor genau an diese Stelle.

### Beispiel 2: ein Tippfehler im Namen

k03_fehler_tippfehler.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
```

```
    pbc_txt24("Hallo!", 80, 120, PBC_WHITE);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
main.c: In function 'main':
main.c:6:5: error: implicit declaration of function 'pbc_txt24'; did you mean
  'pbc_text24'? [-Wimplicit-function-declaration]
    6 |     pbc_txt24("Hallo!", 80, 120, PBC_WHITE);
      |     ^~~~~~~~~
      |     pbc_text24
```

*Implicit declaration of function* heißt: Eine Funktion mit diesem Namen ist nirgends bekannt. Der Compiler hat in `pbc.h` aber einen ähnlichen Namen gefunden und fragt: *did you mean ‚pbc_text24'?*, meintest du `pbc_text24`?

### Beispiel 3: Groß- und Kleinschreibung

k03_fehler_gross.c

```
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_text24("Hallo!", 80, 120, PBC_white);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
main.c: In function 'main':
main.c:6:35: error: 'PBC_white' undeclared (first use in this function); did you mean
  'PBC_WHITE'?
    6 |     pbc_text24("Hallo!", 80, 120, PBC_white);
      |                                   ^~~~~~~~~
      |                                   PBC_WHITE
main.c:6:35: note: each undeclared identifier is reported only once for each function
  it appears in
```

C unterscheidet streng zwischen Groß- und Kleinbuchstaben. `PBC_WHITE` ist eine Farbe, `PBC_white` gibt es nicht. *Undeclared* bedeutet „nicht bekannt“. Die Zeile mit `note:` ist nur ein Hinweis, kein weiterer Fehler.

### Beispiel 4: eine Klammer fehlt

k03_fehler_klammer.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_text24("Hallo!", 80, 120, PBC_WHITE);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
```

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:10:5: error: expected declaration or statement at end of input
   10 |     }
      |     ^
```

Hier fehlt am Ende die `}`, die `main` schließt. Der Compiler merkt das erst, als die Datei zu Ende ist (*at end of input*), und meldet deshalb die letzte Zeile. Der Fehler liegt aber weiter oben: Irgendwo fehlt eine schließende Klammer.

- Der Fehler liegt in der genannten Zeile **oder kurz davor**. Ein vergessenes Semikolon oder eine vergessene Klammer merkt der Compiler oft erst in der nächsten Zeile.
- Meldet der Compiler mehrere Fehler, verbessere den **ersten** und baue neu. Spätere Meldungen sind oft nur Folgen des ersten Fehlers.
- Neben `error:` gibt es `warning:`. Eine **Warnung** hält das Bauen nicht auf, weist aber auf eine verdächtige Stelle hin. Nimm Warnungen ernst; das nächste Kapitel zeigt, warum.

### Die häufigsten Fehler am Anfang:

| Meldung | Bedeutung | Typische Ursache |
|---|---|---|
| `expected ';' before …` | Semikolon fehlt | `;` am Ende der Zeile davor vergessen |
| `implicit declaration of function …` | eine Funktion ist unbekannt | Name falsch geschrieben, `#include "pbc.h"` vergessen |
| `… undeclared` | ein Name ist unbekannt | Name falsch geschrieben, Groß- und Kleinschreibung |
| `expected declaration or statement at end of input` | die Datei endet zu früh | eine `}` fehlt |

## 3.6 Warum `main` nie enden darf

Am Ende jedes Programms in diesem Buch stehen diese drei Zeilen:

```
while (true) {
    sleep_ms(1000);
}
```

`while (true)` heißt: Wiederhole den folgenden Block **immer wieder**, ohne Ende. Das nennt man eine **Endlosschleife**. `sleep_ms(1000)` wartet 1000 Millisekunden, also eine Sekunde. Das Programm wartet also immer wieder eine Sekunde, für immer. Schleifen lernst du in Kapitel 7 genau kennen.

Warum so umständlich? Auf einem PC kehrt ein Programm, das fertig ist, zum Betriebssystem zurück. Auf dem PicoBoy gibt es kein Betriebssystem (Kapitel 1.4). Wenn `main` endet, hält der Prozessor einfach an. Das Bild bleibt zwar stehen, aber der PicoBoy reagiert auf nichts mehr, auch nicht mehr auf **Run** in VS Code. Zum nächsten Aufspielen müsstest du dann wieder BOOT und RES drücken. Die Endlosschleife hält das Programm am Leben.

## Ausprobieren: Eine Visitenkarte

k03_visitenkarte.c

```c
// Eine Visitenkarte auf dem Display
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_WHITE);

    // Name
    pbc_text24("Mia Beispiel", 30, 50, PBC_BLUE);

    // Angaben
    pbc_text12("Hobby: Gitarre", 30, 100, PBC_BLACK);
    pbc_text12("Lieblingsessen: Pizza", 30, 120, PBC_BLACK);
    pbc_text12("Programmiert in C", 30, 140, PBC_BLACK);

    // Fusszeile
    pbc_text("PicoBoy Color Plus", 48, 230, PBC_GRAY);

    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

- Zeile 6 füllt den Bildschirm weiß. Darauf ist schwarze und blaue Schrift gut zu lesen.
- Leere Zeilen und Kommentare teilen das Programm in Abschnitte: Name, Angaben, Fußzeile.
- Die drei Angaben stehen jeweils 20 Pixel untereinander (y = 100, 120, 140). Bei `pbc_text12` ist das ein angenehmer Zeilenabstand.




**Zum Verändern:** Schreib deine eigene Visitenkarte. Probiere andere Hintergrundfarben. Welche Schriftfarben sind darauf gut lesbar?

## Übungsaufgaben

**Aufgabe 1 ⭐ Wo erscheint was?** Lies das Programm, ohne es aufzuspielen. Wo auf dem Bildschirm erscheinen die drei Wörter, und in welcher Farbe? Skizziere den Bildschirm und prüfe danach auf dem PicoBoy.

k03_aufgabe1.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_text18("oben", 30, 30, PBC_WHITE);
    pbc_text18("Mitte", 95, 130, PBC_GREEN);
    pbc_text18("unten", 150, 230, PBC_RED);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 2 ⭐ Drei Zeilen über dich** Schreibe ein Programm, das deinen Vornamen groß (`pbc_text24`) und darunter zwei Angaben über dich (`pbc_text18`) zeigt, alles in verschiedenen Farben.

**Aufgabe 3 ⭐⭐ Finde die Fehler** Dieses Programm enthält drei Fehler. Baue es, lies die Meldungen, verbessere einen Fehler nach dem anderen und baue jedes Mal neu. Wie viele Fehler meldet der Compiler beim ersten Bauen? Warum nicht alle drei?

k03_aufgabe3.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_Text24("Treffen", 40, 80, PBC_YELLOW);
    pbc_text18("Heute 14 Uhr", 40, 130, PBC_WHITE)
    pbc_text18("Raum 2.04", 40, 160, PBC_white);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 4 ⭐⭐ Ein Schild** Schreibe ein Programm, das den Bildschirm rot füllt und in der Mitte groß „STOP“ in Weiß zeigt. Darunter steht klein „Bitte warten“. Probiere so lange verschiedene x-Werte aus, bis die Texte ungefähr in der Mitte stehen.

**Aufgabe 5 ⭐⭐⭐ Stundenplan** Zeige deinen Stundenplan für einen Tag als Tabelle mit den Spalten Stunde, Fach und Raum. Nimm dafür `pbc_text` und überlege: Wie viele Leerzeichen brauchst du, damit die Spalten genau untereinander stehen? Wie groß muss der Abstand der Zeilen mindestens sein?

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- Jedes Programm beginnt mit `#include "pbc.h"` und `int main(void) {`. Die erste Anweisung ist `pbc_init();`.
- Anweisungen enden mit `;`. Geschweifte Klammern `{ }` bilden einen **Block**.
- `pbc_clear(farbe)` füllt den Bildschirm, `pbc_text…(text, x, y, farbe)` schreibt Text. Sichtbar wird alles erst mit **`pbc_show()`**.
- Koordinaten: x von links (0–239), y von **oben** (0–279). Runde Ecken: Abstand zum Rand halten.
- `//` und `/* … */` sind Kommentare.
- Fehlermeldungen: `main.c:Zeile:Spalte: error: …`. Den ersten Fehler zuerst verbessern.
- `main` endet nie: Am Ende steht `while (true) { sleep_ms(1000); }`.

KAPITEL 4

# Die Konsole: `printf` über USB

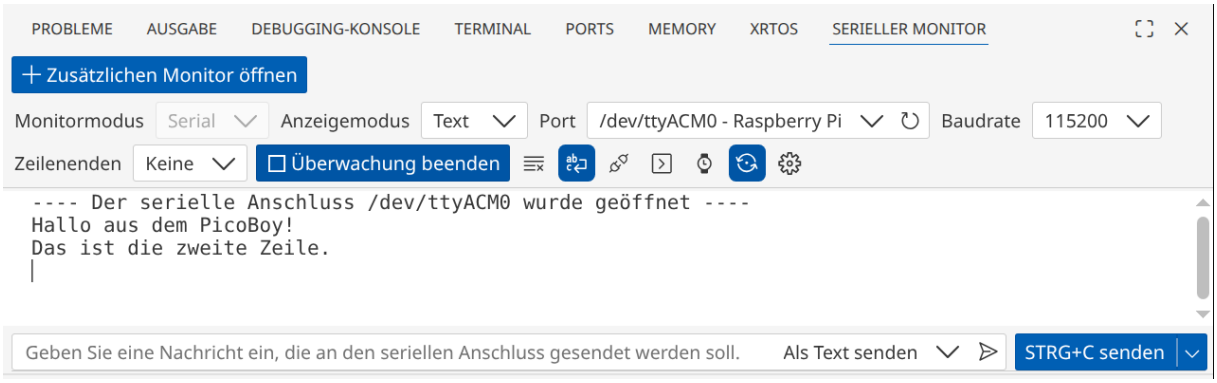
**Worum geht's?** Der PicoBoy kann nicht nur auf seinen Bildschirm schreiben, sondern auch Text über das USB-Kabel an den Computer schicken und von dort Eingaben empfangen. Du lernst den seriellen Monitor in VS Code kennen, gibst Texte und Zahlen mit `printf` aus und liest Eingaben. Das ist praktisch, um längere Ausgaben zu lesen und um nachzusehen, was ein Programm gerade tut.

## 4.1 Der serielle Monitor in VS Code

Über das USB-Kabel laufen nicht nur neue Programme zum PicoBoy. Solange ein Programm aus diesem Buch läuft, meldet sich der PicoBoy am Computer auch als **serielle Schnittstelle**: eine Leitung, über die Text Zeichen für Zeichen hin- und hergeht. Ein Programm auf dem Computer, das diesen Text anzeigt und Eingaben zurückschickt, heißt **serieller Monitor**. Das Fenster mit Text nennt man auch **Konsole**.

So öffnest du den seriellen Monitor:

1. Spiele ein Programm aus diesem Kapitel auf, zum Beispiel `k04_hallo_konsole.c` aus Abschnitt 4.2.
2. Öffne unten in VS Code den Bereich **Serial Monitor**. Findest du ihn nicht, drücke **Strg + Umschalt + P** (Mac: **Cmd + Umschalt + P**), tippe `Serial Monitor` und wähle **Serial Monitor: Focus on Serial Monitor View** (deutsch **Serieller Monitor: Fokus auf Ansicht „Serieller Monitor“**).
3. Wähle bei **Port** den PicoBoy aus. Unter Windows heißt er `COM` und eine Nummer, unter macOS `/dev/tty.usbmodem…`, unter Linux `/dev/ttyACM0`. Stehen mehrere zur Auswahl, zieh das Kabel kurz ab und sieh nach, welcher verschwindet.
4. Die **Baudrate** (Übertragungsgeschwindigkeit) spielt bei USB keine Rolle. Lass sie, wie sie ist.
5. Klicke **Start Monitoring** (auf Deutsch **Überwachung starten**).




Der Serial Monitor mit Port und Baudrate, hier mit der Ausgabe von k04_hallo_konsole.

Beim Aufspielen startet der PicoBoy neu und meldet sich kurz vom Computer ab. Der Monitor verbindet sich danach **nicht** von selbst wieder: Wähle den Port neu und klicke erneut auf **Start Monitoring** (**Überwachung starten**).

> `pbc_init()` wartet beim Start bis zu **2 Sekunden** darauf, dass sich ein serieller Monitor verbindet. Kommt keiner, läuft das Programm trotzdem weiter, und alles, was es ausgibt, bevor ein Monitor zuhört, geht verloren. Fehlen dir die ersten Zeilen: Drücke am PicoBoy einmal kurz **RES**. Das Programm startet neu, und wie beim Aufspielen trennt sich der Monitor dabei. Wähle sofort den Port neu und klicke **Start Monitoring**. Schaffst du das innerhalb der 2 Sekunden, hört der Monitor von Anfang an zu.

## 4.2 Ausgaben mit `printf`

`printf` schreibt einen Text in die Konsole. Der Name kommt von *print formatted*, formatiert ausgeben. `printf` gehört nicht zur Buchbibliothek, sondern zu C selbst: Es funktioniert auf jedem Computer, der C kann.

k04_hallo_konsole.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    printf("Hallo aus dem PicoBoy!\n");
    printf("Das ist die zweite Zeile.\n");
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Hallo aus dem PicoBoy!
Das ist die zweite Zeile.
```

Am Ende jedes Texts steht `\n`. Das ist ein einziges Zeichen, auch wenn man es mit zwei Zeichen schreibt: der **Zeilenumbruch** (*newline*). `printf` fängt nämlich nicht von selbst eine neue Zeile an:

k04_zeilenumbruch.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    printf("Hallo");
    printf("Welt\n");
    printf("eins\nzwei\ndrei\n");
    printf("Spalte\tSpalte\tSpalte\n");
```

```c
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
HalloWelt
eins
zwei
drei
Spalte    Spalte  Spalte
```

- Zeile 5 hat kein `\n`. Deshalb schreibt Zeile 6 direkt dahinter weiter: `HalloWelt`.
- In Zeile 7 stehen drei `\n` in einem Text. Er erscheint auf drei Zeilen.
- `\t` in Zeile 8 ist ein **Tabulator**: ein Sprung zur nächsten Spalte.

Zeichen wie `\n` und `\t`, die mit einem Rückstrich `\` beginnen, heißen **Escape-Sequenzen**. Den Rückstrich tippst du unter Windows mit **AltGr + ß**, am Mac mit **Alt + Umschalt + 7**.

## 4.3 Platzhalter: Zahlen und Texte einsetzen

Richtig nützlich wird `printf` durch **Platzhalter**. Ein Platzhalter beginnt mit `%`. Für jeden Platzhalter im Text gibst du hinter einem Komma einen Wert an, der an dieser Stelle eingesetzt wird:

k04_platzhalter.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    printf("Ich bin %d Jahre alt.\n", 15);
    printf("%d + %d = %d\n", 12, 30, 12 + 30);
    printf("Pi ist ungefaehr %f\n", 3.14159);
    printf("Mit zwei Stellen: %.2f\n", 3.14159);
    printf("Mein Name ist %s.\n", "Mia");
    printf("100 %% geschafft!\n");
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Ich bin 15 Jahre alt.
12 + 30 = 42
Pi ist ungefaehr 3.141590
Mit zwei Stellen: 3.14
Mein Name ist Mia.
100 % geschafft!
```

| Platzhalter | setzt ein | Beispiel |
|---|---|---|
| `%d` | eine ganze Zahl (*decimal*) | `printf("%d", 15)` → `15` |
| `%f` | eine Kommazahl (*floating point*), mit 6 Nachkommastellen | `printf("%f", 3.14159)` → `3.141590` |
| `%.2f` | eine Kommazahl mit 2 Nachkommastellen, gerundet | `printf("%.2f", 3.14159)` → `3.14` |
| `%s` | einen Text (*string*) | `printf("%s", "Mia")` → `Mia` |
| `%%` | das Zeichen `%` selbst | `printf("100 %%")` → `100 %` |

Beachte:

- Kommazahlen schreibt man in C mit **Punkt**: `3.14159`, nicht `3,14159`. Auch `printf` gibt einen Punkt aus.
- Hinter dem Komma darf auch eine Rechnung stehen, wie `12 + 30` in Zeile 6. C rechnet sie aus und setzt das Ergebnis ein.
- In Zeile 6 stehen drei Platzhalter. Die Werte werden **der Reihe nach** eingesetzt: der erste Wert für den ersten Platzhalter und so weiter.

**Der Platzhalter muss zum Wert passen.** Passt er nicht, gibt `printf` Unsinn aus. Zum Glück prüft der Compiler das:

k04_falscher_platzhalter.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    printf("Ergebnis: %d\n", 7.5);
    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
main.c: In function 'main':
main.c:5:24: warning: format '%d' expects argument of type 'int', but argument 2 has
  type 'double' [-Wformat=]
    5 |     printf("Ergebnis: %d\n", 7.5);
      |                        ~^     ~~~
      |                         |     |
      |                         int   double
      |                        %f
```

Das ist eine **Warnung** (`warning:`), kein Fehler. Das Programm wird trotzdem gebaut und aufgespielt. Es gibt dann aber statt `7.5` eine beliebige Zahl aus, denn `printf` liest an der Stelle des Werts etwas ganz anderes. Der Compiler zeigt sogar, was passt: `int` (ganze Zahl) erwartet der Platzhalter, `double` (Kommazahl) ist der Wert, und mit `%f` wäre es richtig.

> Eine Warnung bedeutet fast immer: Das Programm tut nicht, was du willst. Baue so, dass **keine einzige Warnung** übrig bleibt.

## 4.4 Eingaben aus der Konsole lesen

Der serielle Monitor kann auch Text an den PicoBoy schicken. Unten im Monitor gibt es dafür ein Eingabefeld. Was du dort tippst, schickt der Monitor ab, sobald du **Enter** drückst. Stelle dafür im Monitor **Zeilenenden** auf **LF**. Voreingestellt ist **Keine**, und dann reagiert `pbc_input` nicht auf Enter.

Um eine Eingabe zu lesen, braucht das Programm einen Platz, an dem es sie aufbewahrt. So einen Platz mit einem Namen nennt man **Variable**. Du lernst Variablen in Kapitel 5 genau kennen. Hier reicht: Eine Variable für einen Text legst du mit `char name[30];` an. `name` ist der Name, den du dir aussuchst, und `30` gibt an, wie viele Zeichen höchstens hineinpassen.

k04_eingabe.c

```
 1 #include "pbc.h"
 2
 3 int main(void) {
 4     pbc_init();
 5     char name[30];
 6     pbc_input("Wie heisst du? ", name, sizeof name);
 7     printf("Hallo %s, schoen, dass du da bist!\n", name);
 8     while (true) {
 9         sleep_ms(1000);
10     }
11 }
```

Serial Monitor

```
Wie heisst du? Mia
Hallo Mia, schoen, dass du da bist!
```

- Zeile 5 legt die Variable `name` an.
- `pbc_input` in Zeile 6 schreibt die Frage in die Konsole, wartet, bis eine Zeile mit Enter ankommt, und legt sie in `name` ab. Die drei Argumente sind: die Frage, die Variable und `sizeof name`, der Platz in der Variable. So schreibt `pbc_input` nie mehr hinein, als passt.
- Zeile 7 setzt den Inhalt von `name` mit `%s` in den Text ein.

**Zahlen eingeben:** Für ganze Zahlen gibt es `pbc_input_int`, für Kommazahlen `pbc_input_float`. Beide geben die Zahl als **Ergebnis** zurück. Mit `int zahl = …` legst du eine Variable für eine ganze Zahl an und speicherst das Ergebnis gleich darin, mit `float preis = …` eine für eine Kommazahl:

k04_zahl_eingeben.c

```
 1 #include "pbc.h"
 2
 3 int main(void) {
 4     pbc_init();
```

```
    int zahl = pbc_input_int("Gib eine ganze Zahl ein: ");
    printf("Das Doppelte von %d ist %d.\n", zahl, zahl * 2);
    float preis = pbc_input_float("Preis in Euro: ");
    printf("Mit 19 %% Steuer: %.2f Euro\n", preis * 1.19);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Gib eine ganze Zahl ein: 21
Das Doppelte von 21 ist 42.
Preis in Euro: 4,00
Mit 19 % Steuer: 4.76 Euro
```

`pbc_input_float` versteht ein Komma genauso wie einen Punkt. Die Ausgabe von `printf` hat trotzdem immer einen Punkt.

> Die Eingabefunktionen `pbc_input`, `pbc_input_int` und `pbc_input_float` gehören zur Buchbibliothek. C selbst hat auch Funktionen zum Einlesen, etwa `scanf`. Sie sind aber für Anfänger unpraktisch: Bei einer falschen Eingabe bleiben Zeichen liegen und stören die nächste Eingabe.

## Ausprobieren: Das Begrüßungsprogramm

k04_begruessung.c

```c
// Fragt nach Name und Geburtsjahr und begruesst auf Konsole und Display.
#include "pbc.h"

int main(void) {
    pbc_init();
    printf("Willkommen beim PicoBoy!\n");

    char name[30];
    pbc_input("Wie heisst du? ", name, sizeof name);
    int jahr = pbc_input_int("In welchem Jahr bist du geboren? ");

    printf("Hallo %s!\n", name);
    printf("Ende 2026 bist du %d Jahre alt.\n", 2026 - jahr);

    pbc_clear(PBC_BLACK);
    pbc_text18("Hallo", 30, 100, PBC_WHITE);
    pbc_text24(name, 30, 130, PBC_YELLOW);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Willkommen beim PicoBoy!
Wie heisst du? Mia
In welchem Jahr bist du geboren? 2011
Hallo Mia!
Ende 2026 bist du 15 Jahre alt.
```

- Das Programm fragt in der Konsole nach Name und Geburtsjahr.
- Zeile 13 rechnet das Alter aus: `2026 - jahr`.
- Zum Schluss erscheint der Name auch auf dem Bildschirm. `pbc_text24` bekommt dafür statt eines Texts in Anführungszeichen die Variable `name`.




**Zum Verändern:** Frag zusätzlich nach dem Wohnort und zeige ihn klein unter dem Namen an.

## Übungsaufgaben

**Aufgabe 1** ⭐ **Ein Gedicht** Gib ein kurzes Gedicht mit drei Zeilen aus, und zwar mit einem einzigen `printf`.

**Aufgabe 2** ⭐ **Zeit in Zahlen** Lass den PicoBoy ausrechnen und ausgeben, wie viele Stunden eine Woche hat, wie viele Minuten ein Tag und wie viele Sekunden ein Tag.

**Aufgabe 3** ⭐⭐ **Steckbrief** Frage nach Name, Lieblingsfarbe und Alter. Gib danach einen einzigen Satz aus, zum Beispiel: `Ben ist 14 Jahre alt und mag am liebsten Gruen.`

**Aufgabe 4** ⭐⭐ **Warnungen beseitigen** Dieses Programm baut, aber der Compiler warnt zweimal. Lies die Warnungen und verbessere das Programm, bis keine mehr übrig ist.

k04_aufgabe4.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    printf("Temperatur: %f Grad\n", 21);
    printf("Name: %d\n", "Mia");
    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 5 ⭐⭐⭐ Taschenrechner** Lies zwei Kommazahlen ein und gib Summe, Differenz, Produkt und Quotient aus, jeweils mit zwei Nachkommastellen, etwa so: `7.50 / 2.00 = 3.75`. Was gibt das Programm aus, wenn die zweite Zahl 0 ist?

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- Der **serielle Monitor** in VS Code zeigt, was der PicoBoy über USB schickt: **Port** wählen, **Start Monitoring** (**Überwachung starten**).
- `printf("Text\n")` gibt Text aus. `\n` beginnt eine neue Zeile.
- Platzhalter: `%d` ganze Zahl, `%f` Kommazahl, `%.2f` mit 2 Stellen, `%s` Text, `%%` das Prozentzeichen. Der Platzhalter muss zum Wert passen.
- Kommazahlen schreibt man in C mit Punkt: `3.14`.
- **Warnungen** ernst nehmen: Sie zeigen fast immer einen echten Fehler.
- Eingaben: `pbc_input(frage, text, sizeof text)`, `pbc_input_int(frage)`, `pbc_input_float(frage)`.

KAPITEL 5

# Variablen und Datentypen

**Worum geht's?** Ein Programm muss sich Dinge merken: Punkte, Leben, eine Position, ein Ergebnis. Dafür gibt es Variablen. Du lernst, wie du in C Variablen anlegst, welche Arten von Zahlen es gibt und was passiert, wenn eine Zahl zu groß für ihre Variable wird. Außerdem rechnest du mit ganzen Zahlen und Kommazahlen, lernst die Fallen der ganzzahligen Division kennen und lässt den PicoBoy würfeln.

## 5.1 Variablen anlegen: `int`, `float`, `char`, `bool`

Eine **Variable** ist ein Platz im Arbeitsspeicher mit einem Namen. Du kannst dort einen Wert ablegen, ihn später wieder lesen und ihn ändern. In Kapitel 4 hast du schon zwei Variablen benutzt: `int zahl = …` und `float preis = …`. Jetzt sehen wir genauer hin.

In C muss jede Variable **angelegt** werden, bevor du sie benutzt. Dabei sagst du dem Compiler, welche Art von Wert hineinkommt. Diese Art heißt **Datentyp** oder kurz **Typ**. Das Anlegen heißt **Deklaration**. Gibst du gleich einen ersten Wert mit, nennt man das **Initialisieren**:

```
int alter = 15;
```

Das liest man so: Lege eine Variable vom Typ `int` mit dem Namen `alter` an und speichere darin den Wert 15. Die vier wichtigsten Typen zeigt dieses Programm:

k05_variablen.c

```
1  // Vier Variablen mit vier Datentypen
2  #include "pbc.h"
3
4  int main(void) {
5      pbc_init();
6      int alter = 15;
7      float groesse = 1.68;
8      char anfang = 'M';
9      bool hat_haustier = true;
10
11     printf("Alter: %d\n", alter);
12     printf("Groesse: %.2f m\n", groesse);
13     printf("Anfangsbuchstabe: %c\n", anfang);
14     printf("Haustier: %d\n", hat_haustier);
15
16     alter = 16;
17     printf("Ein Jahr spaeter: %d\n", alter);
18     while (true) {
19         sleep_ms(1000);
20     }
21 }
```

Serial Monitor

```
Alter: 15
Groesse: 1.68 m
Anfangsbuchstabe: M
Haustier: 1
Ein Jahr spaeter: 16
```

| Typ | speichert | Beispiel | in `printf` |
|---|---|---|---|
| `int` | eine ganze Zahl (*integer*) | `15`, `-3`, `0` | `%d` |
| `float` | eine Kommazahl (*floating point*) | `1.68`, `-0.5` | `%f`, `%.2f` |
| `char` | ein einzelnes Zeichen (*character*) | `'M'`, `'?'`, `'7'` | `%c` |
| `bool` | wahr oder falsch | `true`, `false` | `%d` (gibt `1` oder `0` aus) |

- Zeile 6 bis 9 legen die vier Variablen an.
- Ein einzelnes Zeichen steht in **einfachen** Anführungszeichen: `'M'`. `"M"` in doppelten Anführungszeichen wäre eine Zeichenkette, also ein Text (Kapitel 12). Für Zeichen gibt es in `printf` den Platzhalter `%c`.
- Ein `bool` kennt nur zwei Werte: `true` (wahr) und `false` (falsch). `printf` hat dafür keinen eigenen Platzhalter. Mit `%d` erscheint `1` für wahr und `0` für falsch. In Kapitel 8 entscheidet ein Programm mit solchen Werten, was es tut.
- In Zeile 16 bekommt `alter` einen **neuen Wert**. Der Typ steht hier nicht mehr davor, denn die Variable gibt es schon. Der alte Wert 15 ist danach weg. Das `=` heißt in C deshalb **Zuweisung**: „Speichere rechts den Wert in die Variable links“. Es ist keine Gleichung wie in der Mathematik.

> In der Warnung aus Kapitel 4 kam der Typ `double` vor. Auch er speichert Kommazahlen, aber doppelt so genau wie `float` (etwa 16 statt 7 Stellen). Eine Kommazahl, die du ins Programm schreibst, etwa `1.68`, ist zunächst ein `double`. In Zeile 7 wandelt C sie beim Speichern in einen `float` um. Das Buch benutzt `float`: Der Prozessor des PicoBoy hat ein eigenes Rechenwerk für `float`-Zahlen und rechnet damit besonders schnell.

**Zeichen sind Zahlen.** Für den Computer ist ein `char` eine kleine Zahl: die Nummer des Zeichens in einer Tabelle, dem **ASCII-Code**. Das `A` hat die Nummer 65, das `B` die 66, das kleine `a` die 97:

k05_zeichen.c

```c
// Ein char ist eine kleine Zahl: die Nummer des Zeichens
#include "pbc.h"

int main(void) {
    pbc_init();
    char zeichen = 'A';
    printf("%c hat die Nummer %d\n", zeichen, zeichen);

    zeichen = zeichen + 1;
```

```
10      printf("Danach kommt %c mit der Nummer %d\n", zeichen, zeichen);
11
12      printf("Die Nummer 97 ist das Zeichen %c\n", 97);
13      while (true) {
14          sleep_ms(1000);
15      }
16  }
```

Serial Monitor

```
A hat die Nummer 65
Danach kommt B mit der Nummer 66
Die Nummer 97 ist das Zeichen a
```

Du kannst mit einem `char` also rechnen (Zeile 9). Ob `printf` das Zeichen oder die Zahl zeigt, entscheidet nur der Platzhalter: `%c` oder `%d`.

**Namen für Variablen** darfst du dir selbst ausdenken. Dabei gelten Regeln:

- erlaubt sind Buchstaben, Ziffern und der Unterstrich `_`;
- am Anfang steht keine Ziffer: `punkte2` geht, `2punkte` nicht;
- keine Umlaute, kein ß, keine Leerzeichen: `groesse`, `anzahl_leben`;
- Groß- und Kleinschreibung zählt: `Punkte` und `punkte` sind zwei verschiedene Variablen;
- Wörter, die zu C gehören, sind verboten, etwa `int`, `float`, `while`.

Wähle Namen, die sagen, was drinsteht. `leben` ist besser als `l`.

**Eine Variable ohne Wert.** Du darfst eine Variable auch ohne ersten Wert anlegen: `int punkte;`. Dann steht aber irgendetwas darin, was zufällig gerade an dieser Stelle im Speicher war. Der Compiler warnt, wenn du so eine Variable liest:

k05_ohne_wert.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int punkte;
    printf("Punkte: %d\n", punkte);
    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
Linking CXX executable k05_ohne_wert.elf
Built target k05_ohne_wert
```

```
main.c: In function 'main':
main.c:6:5: warning: 'punkte' is used uninitialized [-Wuninitialized]
    6 |     printf("Punkte: %d\n", punkte);
      |     ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
main.c:5:9: note: 'punkte' was declared here
    5 |     int punkte;
      |         ^~~~~~
```

*Used uninitialized* heißt: benutzt, ohne vorher einen Wert bekommen zu haben. Die Zeile mit `note:` zeigt, wo die Variable angelegt wurde. Gib jeder Variable deshalb gleich beim Anlegen einen Wert, notfalls `0`.

## 5.2 Feste Größen: `uint8_t`, `int16_t`, `uint32_t` – und der Überlauf

Der Speicher besteht aus **Bits**. Ein Bit ist 0 oder 1. Acht Bits sind ein **Byte**. Mit 8 Bits lassen sich $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 256$ verschiedene Werte darstellen, mit 16 Bits 65 536, mit 32 Bits über vier Milliarden. Jede Variable belegt eine feste Anzahl Bits. Ein `int` hat auf dem PicoBoy 32 Bits, also 4 Bytes.

Manchmal willst du genau festlegen, wie groß eine Variable ist: weil der Speicher knapp ist oder weil ein Bauteil genau 8 oder 16 Bits erwartet. Dafür gibt es Typen mit der Größe im Namen:

| Typ | Bits | kleinster Wert | größter Wert |
|---|---|---|---|
| `int8_t` | 8 | −128 | 127 |
| `uint8_t` | 8 | 0 | 255 |
| `int16_t` | 16 | −32 768 | 32 767 |
| `uint16_t` | 16 | 0 | 65 535 |
| `int32_t` | 32 | −2 147 483 648 | 2 147 483 647 |
| `uint32_t` | 32 | 0 | 4 294 967 295 |

So liest du die Namen: Die Zahl ist die Anzahl der Bits. Das `u` am Anfang steht für *unsigned*, **ohne Vorzeichen**: Diese Typen können nicht negativ werden, reichen dafür doppelt so weit ins Positive. Das `_t` am Ende steht für *type*. Du hast zwei davon schon gesehen: In `pbc.h` sind Farben vom Typ `uint16_t`, und `pbc_millis()` (Kapitel 7) liefert einen `uint32_t`.

**Der Überlauf.** Was passiert, wenn ein Ergebnis nicht in seine Variable passt?

k05_ueberlauf.c

```c
// Was passiert, wenn eine Zahl nicht mehr in ihre Variable passt?
#include "pbc.h"

int main(void) {
    pbc_init();
    uint8_t helligkeit = 250;
```

```
 7     helligkeit = helligkeit + 10;
 8     printf("uint8_t:  250 + 10 = %d\n", helligkeit);
 9
10     uint8_t leben = 0;
11     leben = leben - 1;
12     printf("uint8_t:  0 - 1 = %d\n", leben);
13
14     int16_t punkte = 32767;
15     punkte = punkte + 1;
16     printf("int16_t:  32767 + 1 = %d\n", punkte);
17
18     int gross = 2147483647;
19     gross = gross + 1;
20     printf("int:      2147483647 + 1 = %d\n", gross);
21     while (true) {
22         sleep_ms(1000);
23     }
24 }
```

Serial Monitor

```
uint8_t:  250 + 10 = 4
uint8_t:  0 - 1 = 255
int16_t:  32767 + 1 = -32768
int:      2147483647 + 1 = -2147483648
```

Die Zahl springt **ohne jede Warnung** auf die andere Seite. Das ist wie bei einem alten Kilometerzähler im Auto: Nach 99 999 kommt wieder 00 000.

- Zeile 7: 260 passt nicht in einen `uint8_t`. Es bleibt 260 − 256 = 4.
- Zeile 11: Unter 0 geht es bei `uint8_t` nicht. 0 − 1 wird zu 255.
- Zeile 15: Nach dem größten `int16_t` kommt der kleinste: −32 768.
- Zeile 19: Auch ein `int` läuft über, nur viel später.

Bei Typen **mit** Vorzeichen legt C eigentlich nicht fest, was beim Überlauf herauskommt. Der PicoBoy springt so wie gezeigt ins Negative; verlassen solltest du dich darauf nicht. Wähle den Typ lieber so, dass alle Werte hineinpassen.

> Für Zahlen nimmst du normalerweise `int` und `float`. Die Typen mit fester Größe brauchst du nur, wenn es auf die Größe ankommt. Dann überlege, ob alle Werte hineinpassen: Ein Überlauf passiert **ohne Warnung**.

## 5.3 Rechnen: `+ - * / %`, ganzzahlige Division, Typumwandlung

C kennt die vier Grundrechenarten und den **Rest** einer Division:

k05_rechnen.c

```c
// Die Grundrechenarten mit ganzen Zahlen und mit Kommazahlen
#include "pbc.h"

int main(void) {
    pbc_init();
    int a = 7;
    int b = 2;
    printf("%d + %d = %d\n", a, b, a + b);
    printf("%d - %d = %d\n", a, b, a - b);
    printf("%d * %d = %d\n", a, b, a * b);
    printf("%d / %d = %d\n", a, b, a / b);
    printf("%d %% %d = %d\n", a, b, a % b);

    float x = 7;
    printf("%.1f / %d = %.1f\n", x, b, x / b);

    printf("2 + 3 * 4 = %d\n", 2 + 3 * 4);
    printf("(2 + 3) * 4 = %d\n", (2 + 3) * 4);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
7 + 2 = 9
7 - 2 = 5
7 * 2 = 14
7 / 2 = 3
7 % 2 = 1
7.0 / 2 = 3.5
2 + 3 * 4 = 14
(2 + 3) * 4 = 20
```

| Zeichen | Bedeutung | Beispiel |
|---|---|---|
| `+` | plus | `7 + 2` ergibt `9` |
| `-` | minus | `7 - 2` ergibt `5` |
| `*` | mal | `7 * 2` ergibt `14` |
| `/` | geteilt | `7 / 2` ergibt `3`, aber `7.0 / 2` ergibt `3.5` |
| `%` | Rest der Division (*Modulo*) | `7 % 2` ergibt `1` |

**Ganzzahlige Division.** Zeile 11 ist die wichtigste Zeile dieses Kapitels: `7 / 2` ergibt in C **3**, nicht 3,5. Sind beide Zahlen ganze Zahlen, ist auch das Ergebnis eine ganze Zahl. Was hinter dem Komma stünde, fällt einfach weg. Es wird nicht gerundet: Auch `9 / 5` ergibt 1. Ist mindestens eine der

beiden Zahlen eine Kommazahl, rechnet C mit Kommazahlen (Zeile 15).

Der **Rest** mit `%` ist das, was bei der ganzzahligen Division übrig bleibt: 7 geteilt durch 2 ist 3, Rest 1. Das ist praktischer, als es aussieht. `zahl % 2` ist 0 für gerade und 1 für ungerade Zahlen, `sekunden % 60` sind die Sekunden, die nach den ganzen Minuten übrig bleiben.

**Punkt vor Strich** gilt wie in der Mathematik: `*`, `/` und `%` werden vor `+` und `-` gerechnet. Klammern ändern die Reihenfolge (Zeile 17 und 18).

**Typumwandlung.** Die ganzzahlige Division ist eine häufige Fehlerquelle, etwa bei einem Durchschnitt:

k05_durchschnitt.c

```
 1 // Ein Durchschnitt: Wann rechnet C mit ganzen Zahlen?
 2 #include "pbc.h"
 3
 4 int main(void) {
 5     pbc_init();
 6     int summe = 2 + 3 + 3;          // drei Noten
 7
 8     float falsch = summe / 3;
 9     float richtig = (float)summe / 3;
10     printf("summe / 3        = %.2f\n", falsch);
11     printf("(float)summe / 3 = %.2f\n", richtig);
12
13     int abgeschnitten = (int)richtig;
14     int gerundet = (int)(richtig + 0.5);
15     printf("(int)%.2f = %d\n", richtig, abgeschnitten);
16     printf("gerundet: %d\n", gerundet);
17     while (true) {
18         sleep_ms(1000);
19     }
20 }
```

Serial Monitor

```
summe / 3        = 2.00
(float)summe / 3 = 2.67
(int)2.67 = 2
gerundet: 3
```

In Zeile 8 steht das Ergebnis in einer `float`-Variable, trotzdem kommt 2.00 heraus. Der Grund: C rechnet zuerst `summe / 3` aus, und das sind zwei ganze Zahlen. Das Ergebnis 2 wird erst danach in einen `float` verwandelt.

In Zeile 9 steht `(float)` vor `summe`. Ein Typ in runden Klammern vor einem Wert wandelt den Wert in diesen Typ um. Das heißt **Typumwandlung** (englisch *cast*). Jetzt ist eine der beiden Zahlen eine Kommazahl, und C rechnet mit Kommazahlen: 2.67.

Umgekehrt wandelt `(int)` eine Kommazahl in eine ganze Zahl um (Zeile 13). Auch dabei wird **abgeschnitten**, nicht gerundet: Aus 2.67 wird 2. Zum Runden zählst du vorher 0.5 dazu (Zeile 14): 2.67 + 0.5 = 3.17, abgeschnitten 3. Dieser Trick gilt für positive Zahlen.

Die Typumwandlung hilft auch bei `printf`. `pbc_millis()` liefert einen `uint32_t`, und der passt nicht zum Platzhalter `%d`. Der Compiler warnt:

k05_millis_warnung.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    uint32_t zeit = pbc_millis();
    printf("Seit dem Start: %d ms\n", zeit);
    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
Linking CXX executable k05_millis_warnung.elf
Built target k05_millis_warnung
main.c: In function 'main':
main.c:6:30: warning: format '%d' expects argument of type 'int', but argument 2
  has type 'uint32_t' {aka 'long unsigned int'} [-Wformat=]
    6 |     printf("Seit dem Start: %d ms\n", zeit);
      |                              ~^        ~~~~
      |                               |        |
      |                               int      uint32_t {aka long unsigned int}
      |                              %ld
```

Mit `(int)` davor passt der Wert zu `%d`, und die Warnung ist weg: `printf("Seit dem Start: %d ms\n", (int)zeit);`. Das klappt, solange die Zahl in einen `int` passt, bei Millisekunden also gut drei Wochen lang. Die Typen `int8_t`, `uint8_t`, `int16_t` und `uint16_t` kannst du dagegen direkt mit `%d` ausgeben.

**Division durch 0.** In Kapitel 4, Aufgabe 5 hast du gesehen: Teilst du eine Kommazahl durch 0, kommt `inf` heraus, „unendlich“. Für ganze Zahlen gibt es so einen Wert nicht. Die Sprache C legt nicht fest, was dann passiert: Es ist ein Fehler im Programm. Auf einem PC bricht ein solches Programm meistens sofort ab. Der Prozessor des PicoBoy rechnet dagegen einfach weiter. Probier es mit diesem Programm aus:

k05_teilen_null.c

```c
// Ganzzahlige Division durch 0 - auf dem PicoBoy ausprobieren
#include "pbc.h"

int main(void) {
    pbc_init();
    int a = pbc_input_int("Erste Zahl: ");
```

```
    int b = pbc_input_int("Zweite Zahl: ");
    printf("%d / %d = %d\n", a, b, a / b);
    printf("%d %% %d = %d\n", a, b, a % b);
    while (true) {
        sleep_ms(1000);
    }
}
```

Gibst du 7 und 0 ein, erscheint `7 / 0 = 0` und `7 % 0 = 7`: Der Prozessor liefert als Ergebnis der Division 0, ohne jede Meldung. Den Rest berechnet das Programm aus diesem Ergebnis ($7 - 0 \cdot 0 = 7$). Das Programm läuft also weiter, aber mit einem falschen Wert.

> Teile nie eine ganze Zahl durch 0. Kann der Teiler 0 sein, etwa weil er aus einer Eingabe kommt, musst du das vorher prüfen. Wie ein Programm so etwas entscheidet, zeigt Kapitel 8.

## 5.4 `x++` und `x += 3`

Sehr oft verändert ein Programm eine Variable ausgehend von ihrem alten Wert: `punkte = punkte + 10;`. Rechts steht der alte Wert, links wird das Ergebnis gespeichert. Weil das so häufig vorkommt, gibt es kürzere Schreibweisen:

k05_kurz.c

```
// Kurzschreibweisen zum Veraendern einer Variablen
#include "pbc.h"

int main(void) {
    pbc_init();
    int punkte = 0;
    punkte = punkte + 10;
    printf("punkte = punkte + 10  ->  %d\n", punkte);
    punkte += 10;
    printf("punkte += 10          ->  %d\n", punkte);
    punkte -= 5;
    printf("punkte -= 5           ->  %d\n", punkte);
    punkte *= 3;
    printf("punkte *= 3           ->  %d\n", punkte);
    punkte /= 2;
    printf("punkte /= 2           ->  %d\n", punkte);

    int leben = 3;
    leben--;
    printf("leben--               ->  %d\n", leben);
    leben++;
    leben++;
    printf("2 x leben++           ->  %d\n", leben);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
punkte = punkte + 10  ->  10
punkte += 10          ->  20
punkte -= 5           ->  15
punkte *= 3           ->  45
punkte /= 2           ->  22
leben--               ->  2
2 x leben++           ->  4
```

| kurz | bedeutet |
|---|---|
| `x += 3;` | `x = x + 3;` |
| `x -= 3;` | `x = x - 3;` |
| `x *= 3;` | `x = x * 3;` |
| `x /= 3;` | `x = x / 3;` |
| `x++;` | `x = x + 1;` |
| `x--;` | `x = x - 1;` |

`x++` (sprich „x plus plus“) zählt um eins hoch, `x--` um eins herunter. Das brauchst du ständig, zum Beispiel für Zähler in Schleifen (Kapitel 7). Auch `/=` ist eine ganzzahlige Division, wenn `x` ein `int` ist: In Zeile 15 wird aus 45 die 22.

> Es gibt auch `++x` mit den Pluszeichen davor. Steht es allein in einer Zeile wie hier, ist es dasselbe wie `x++`. Einen Unterschied gibt es nur, wenn `x++` mitten in einer größeren Rechnung steht. Das macht Programme schwer lesbar; das Buch schreibt `x++` deshalb immer in eine eigene Zeile.

## 5.5 Konstanten: `const` und `#define`

Manche Werte ändern sich nie: die Zahl π, die Breite des Displays, die Zahl der Leben zu Spielbeginn. Solche Werte heißen **Konstanten**. Man könnte sie direkt als Zahl ins Programm schreiben. Mit einem Namen ist das Programm aber besser zu lesen, und wenn du den Wert ändern willst, änderst du ihn an einer einzigen Stelle. C hat dafür zwei Möglichkeiten:

k05_konstanten.c

```c
// Umfang und Flaeche eines Kreises mit zwei Konstanten
#include "pbc.h"

#define PI 3.14159

int main(void) {
    pbc_init();
    const float RADIUS = 2.5;
```

```
    float umfang = 2 * PI * RADIUS;
    float flaeche = PI * RADIUS * RADIUS;
    printf("Radius:  %.2f cm\n", RADIUS);
    printf("Umfang:  %.2f cm\n", umfang);
    printf("Flaeche: %.2f cm2\n", flaeche);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Radius:  2.50 cm
Umfang:  15.71 cm
Flaeche: 19.63 cm2
```

**`#define`** in Zeile 4 kennst du schon aus `pbc.h`: Dort steht zum Beispiel `#define PBC_WIDTH 240`. Wie `#include` ist `#define` eine Anweisung an den Compiler, die er **vor** dem Übersetzen ausführt. Sie bedeutet: Ersetze im ganzen Programm das Wort `PI` durch den Text `3.14159`. Beachte: kein `=`, kein Semikolon. Was hinter dem Namen steht, wird genau so eingesetzt, auch ein `=` oder `;`, und das gibt dann seltsame Fehler an einer ganz anderen Stelle. Wähle außerdem eindeutige Namen: Ein paar kurze wie `MIN` und `MAX` sind im SDK schon vergeben. Der Compiler warnt dann mit *redefined*, „neu festgelegt“.

**`const`** in Zeile 8 legt eine richtige Variable mit Typ an, die nach dem Initialisieren nicht mehr verändert werden darf. Versuchst du es doch, meldet der Compiler einen Fehler:

k05_const_fehler.c

```
#include "pbc.h"

int main(void) {
    pbc_init();
    const float RADIUS = 2.5;
    RADIUS = 3;
    printf("Radius: %.2f\n", RADIUS);
    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:6:12: error: assignment of read-only variable 'RADIUS'
    6 |     RADIUS = 3;
      |            ^
```

*Read-only* heißt „nur lesen“. Konstanten schreibt man in Großbuchstaben, damit man sie sofort von Variablen unterscheidet. Das Buch benutzt meistens `#define`; `const` kommt in Kapitel 19 noch einmal vor, wenn es um den Speicher geht.

## 5.6 Zufallszahlen

Für Spiele braucht man Zufall: einen Würfel, eine zufällige Position, einen zufälligen Zeitpunkt. Die Buchbibliothek hat dafür `pbc_random(min, max)`. Es liefert eine ganze Zahl von `min` bis `max`, **beide eingeschlossen**: `pbc_random(1, 6)` kann 1, 2, 3, 4, 5 oder 6 sein.

k05_zufall.c

```c
// Zufallszahlen in verschiedenen Bereichen
#include "pbc.h"

int main(void) {
    pbc_init();
    int wurf = pbc_random(1, 6);
    int zahl = pbc_random(1, 100);
    int muenze = pbc_random(0, 1);

    printf("Wuerfel: %d\n", wurf);
    printf("Zahl von 1 bis 100: %d\n", zahl);
    printf("Muenze (0 = Kopf, 1 = Zahl): %d\n", muenze);
    printf("Noch ein Wurf: %d\n", pbc_random(1, 6));
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Wuerfel: 3
Zahl von 1 bis 100: 8
Muenze (0 = Kopf, 1 = Zahl): 0
Noch ein Wurf: 3
```

Die Zahlen oben sind ein Beispiel. Bei dir kommen andere heraus, und **bei jedem Start** des Programms wieder andere. Der RP2350 hat dafür einen eingebauten Zufallsgenerator, der aus winzigen elektrischen Schwankungen im Chip zufällige Bits gewinnt. Das SDK mischt sie zusätzlich mit der Zeit seit dem Start.

In Zeile 13 steht `pbc_random` direkt in `printf`, ohne Variable. Das geht, wenn du die Zahl nur einmal brauchst. Jeder Aufruf liefert eine neue Zahl.

## Ausprobieren: Der Würfel

k05_wuerfel.c

```c
// Der Wuerfel: zwei Wuerfe, sobald Enter gedrueckt wird
#include "pbc.h"

#define SEITEN 6

int main(void) {
    pbc_init();
    char eingabe[10];
    printf("Zwei Wuerfel mit je %d Seiten\n", SEITEN);
    pbc_input("Druecke Enter zum Wuerfeln ", eingabe, sizeof eingabe);

    int wurf1 = pbc_random(1, SEITEN);
    int wurf2 = pbc_random(1, SEITEN);
    int summe = wurf1 + wurf2;

    printf("Erster Wuerfel:  %d\n", wurf1);
    printf("Zweiter Wuerfel: %d\n", wurf2);
    printf("Summe:           %d\n", summe);
    printf("Durchschnitt:    %.1f\n", summe / 2.0);
    printf("Neuer Wurf: RES am PicoBoy druecken\n");
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Zwei Wuerfel mit je 6 Seiten
Druecke Enter zum Wuerfeln
Erster Wuerfel:  3
Zweiter Wuerfel: 4
Summe:           7
Durchschnitt:    3.5
Neuer Wurf: RES am PicoBoy druecken
```

- Zeile 4 legt die Zahl der Seiten als Konstante fest. Ändere sie auf 20, und du hast einen Würfel für Rollenspiele: Das Programm passt sich überall an, auch in der Überschrift.
- Zeile 10 benutzt `pbc_input` nur zum Warten: Das Programm geht erst weiter, wenn du im seriellen Monitor Enter drückst. Was du dabei tippst, wird nicht gebraucht.
- In Zeile 19 steht `summe / 2.0`. Mit `/ 2` gäbe es eine ganzzahlige Division, und aus 7 würde 3 statt 3.5.
- Für den nächsten Wurf drückst du am PicoBoy **RES**: Das Programm startet neu. Verbinde danach den Monitor neu (Abschnitt 4.1). In Kapitel 7 lernst du Schleifen kennen, mit denen ein Programm beliebig oft würfeln kann.

**Zum Verändern:** Würfle mit drei Würfeln. Gib zusätzlich aus, wie viel noch bis zur höchsten möglichen Summe fehlt.

## Übungsaufgaben

**Aufgabe 1** ⭐ **Was kommt heraus?** Überlege ohne PicoBoy, welche fünf Zeilen dieses Programm ausgibt. Prüfe danach am Gerät.

k05_aufgabe1.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int a = 17;
    int b = 5;
    printf("%d\n", a / b);
    printf("%d\n", a % b);
    printf("%.1f\n", a / 2.0);
    a += b;
    b--;
    printf("%d %d\n", a, b);
    printf("%d\n", a / b * b);
    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 2** ⭐ **Fahrenheit** In den USA misst man Temperaturen in Grad Fahrenheit. Die Formel lautet: Fahrenheit = Celsius · 9 / 5 + 32. Lies eine Temperatur in Grad Celsius ein und gib sie in Grad Fahrenheit aus, mit einer Nachkommastelle.

**Aufgabe 3** ⭐⭐ **Sekunden zerlegen** Lies eine Anzahl Sekunden ein und gib aus, wie viele Stunden, Minuten und Sekunden das sind, etwa so: `4000 s = 1 h 6 min 40 s`. Tipp: `/` und `%`.

**Aufgabe 4** ⭐⭐ **Seltsame Werte** Dieses Programm gibt `Punkte: 44` und `Temperatur: -126` aus. Erkläre, wie diese Zahlen zustande kommen, und verbessere das Programm.

k05_aufgabe4.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    uint8_t punkte = 200;
    punkte += 100;
    printf("Punkte: %d\n", punkte);

    int8_t temperatur = 120;
```

```
10      temperatur += 10;
11      printf("Temperatur: %d\n", temperatur);
12      while (true) {
13          sleep_ms(1000);
14      }
15  }
```

**Aufgabe 5 ⭐⭐⭐ Rückgeld** Lies einen Preis und den bezahlten Betrag ein, beide in Cent. Gib das Rückgeld aus und zerlege es in möglichst wenige Münzen: 2 Euro, 1 Euro, 50, 20, 10, 5, 2 und 1 Cent. Warum ist es geschickt, in Cent zu rechnen und nicht in Euro mit Kommazahlen?

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- Eine **Variable** wird mit Typ und Namen angelegt und am besten gleich initialisiert: `int punkte = 0;`. `=` ist eine **Zuweisung**.
- Die wichtigsten Typen: `int` (ganze Zahl, `%d`), `float` (Kommazahl, `%f`), `char` (Zeichen, `'A'`, `%c`), `bool` (`true` / `false`).
- Feste Größen: `int8_t` … `uint32_t`. Zu große Werte **laufen über**, ohne Warnung.
- `/` mit zwei ganzen Zahlen schneidet ab: `7 / 2` ist `3`. `%` ist der Rest. Mit `(float)` oder `(int)` wandelst du einen Typ um.
- Nie eine ganze Zahl durch 0 teilen.
- `x += 3`, `x -= 3`, `x *= 3`, `x /= 3`, `x++`, `x--`.
- Konstanten: `#define NAME wert` (ohne `=` und `;`) oder `const`.
- `pbc_random(min, max)`: Zufallszahl, beide Grenzen eingeschlossen.

KAPITEL 6

# Zeichnen und Farben

**Worum geht's?** Bisher hast du nur Text auf den Bildschirm geschrieben. Jetzt zeichnest du Punkte, Linien, Rechtecke, Kreise und Ellipsen, mischst eigene Farben und erfährst, wie der PicoBoy Farben als Zahlen speichert. Außerdem setzt du Text genau in die Mitte und zeigst Zahlen auf dem Display an. Damit hast du alles beisammen, um Bilder für Spiele zu zeichnen.

## 6.1 Pixel, Linien, Rechtecke, Kreise

Alle Zeichenbefehle funktionieren wie `pbc_text`: Sie bekommen Koordinaten und eine Farbe und malen in den Bildpuffer. Sichtbar wird das Bild erst mit `pbc_show()`. Das Koordinatensystem kennst du aus Kapitel 3.3: (0, 0) ist oben links, x wächst nach rechts, y nach unten.

### Punkte und Linien

k06_linien.c

```c
// Punkte und Linien
#include "pbc.h"

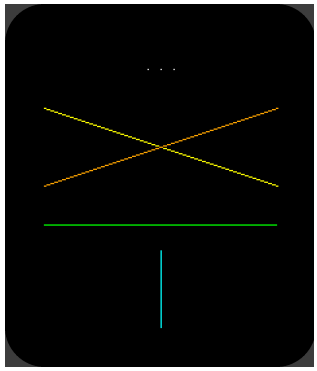
int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);

    pbc_pixel(110, 50, PBC_WHITE);   // drei Punkte
    pbc_pixel(120, 50, PBC_WHITE);
    pbc_pixel(130, 50, PBC_WHITE);

    pbc_line(30, 80, 210, 140, PBC_YELLOW);   // schraeg
    pbc_line(30, 140, 210, 80, PBC_ORANGE);   // schraeg
    pbc_hline(30, 170, 180, PBC_GREEN);       // waagrecht
    pbc_vline(120, 190, 60, PBC_CYAN);        // senkrecht

    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

- `pbc_pixel(x, y, farbe)` setzt einen einzelnen Bildpunkt, ein **Pixel**. Die drei Punkte in Zeile 8 bis 10 sind im Bild kaum zu sehen, denn ein einzelnes Pixel ist winzig.
- `pbc_line(x1, y1, x2, y2, farbe)` zieht eine Linie vom Punkt (x1, y1) zum Punkt (x2, y2). Beide Endpunkte gehören dazu.
- Für waagrechte und senkrechte Linien gibt es `pbc_hline(x, y, länge, farbe)` (*horizontal*) und `pbc_vline(x, y, länge, farbe)` (*vertikal*). Sie beginnen bei (x, y) und gehen nach rechts beziehungsweise nach unten.



### Rechtecke, Kreise und Ellipsen

k06_formen.c

```c
// Links Umrisse, rechts gefuellte Formen
#include "pbc.h"

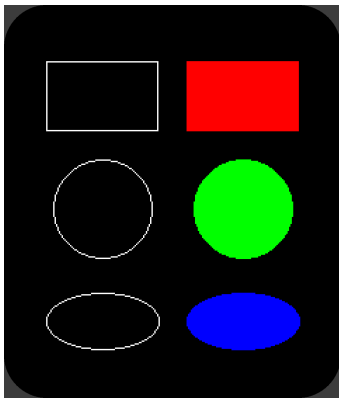
int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);

    pbc_rect(30, 40, 80, 50, PBC_WHITE);
    pbc_fill_rect(130, 40, 80, 50, PBC_RED);

    pbc_circle(70, 145, 35, PBC_WHITE);
    pbc_fill_circle(170, 145, 35, PBC_GREEN);

    pbc_ellipse(70, 225, 40, 20, PBC_WHITE);
    pbc_fill_ellipse(170, 225, 40, 20, PBC_BLUE);

    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```



Jede Form gibt es zweimal: als Umriss und gefüllt. Der gefüllte Befehl hat `fill_` im Namen.

| Befehl | Argumente |
|---|---|
| `pbc_rect(x, y, b, h, farbe)` | Rechteck: linke obere Ecke (x, y), Breite b, Höhe h |
| `pbc_fill_rect(x, y, b, h, farbe)` | dasselbe, gefüllt |
| `pbc_circle(mx, my, r, farbe)` | Kreis: Mittelpunkt (mx, my), Radius r |
| `pbc_fill_circle(mx, my, r, farbe)` | dasselbe, gefüllt |
| `pbc_ellipse(mx, my, rx, ry, farbe)` | Ellipse: Mittelpunkt, Radius waagrecht rx und senkrecht ry |
| `pbc_fill_ellipse(mx, my, rx, ry, farbe)` | dasselbe, gefüllt |

Achte auf den Unterschied: Beim Rechteck gibst du die **linke obere Ecke** an, beim Kreis und bei der Ellipse den **Mittelpunkt**. Ein Kreis mit dem Radius 35 ist 71 Pixel breit: 35 Pixel links vom Mittelpunkt, 35 rechts und der Mittelpunkt selbst.

Was über den Rand des Bildschirms hinausgeht, wird einfach weggelassen. Du darfst also Formen zeichnen, die nur zum Teil auf dem Bildschirm liegen.

**Die Reihenfolge zählt.** Die Zeichenbefehle malen wie mit deckender Farbe: Was später gezeichnet wird, liegt oben und verdeckt, was darunter ist.

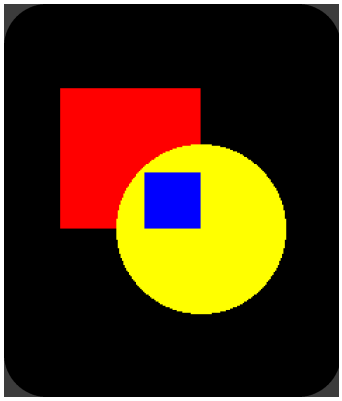
k06_reihenfolge.c

```c
// Was spaeter gezeichnet wird, liegt oben
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);

    pbc_fill_rect(40, 60, 100, 100, PBC_RED);
    pbc_fill_circle(140, 160, 60, PBC_YELLOW);
    pbc_fill_rect(100, 120, 40, 40, PBC_BLUE);

    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```



Der gelbe Kreis (Zeile 9) verdeckt einen Teil des roten Quadrats (Zeile 8). Das blaue Quadrat (Zeile 10) kommt zuletzt und liegt deshalb über beiden. Für ein Bild heißt das: erst den Hintergrund, dann alles, was davor steht.

## 6.2 Farben in 16 Bit: `pbc_rgb(r, g, b)` und die Farbkonstanten

Jedes Pixel des Displays besteht aus drei winzigen Lämpchen: einem roten, einem grünen und einem blauen. Aus diesen drei Farben mischt das Display alle anderen. Mischen heißt hier: Licht addieren.

| Rot | Grün | Blau | ergibt |
|---|---|---|---|
| voll | voll | aus | Gelb |
| voll | aus | voll | Magenta (pink-lila) |
| aus | voll | voll | Cyan (türkis) |
| voll | voll | voll | Weiß |
| halb | halb | halb | Grau |
| aus | aus | aus | Schwarz |

Mit `pbc_rgb(r, g, b)` mischst du eine Farbe selbst. Die drei Zahlen geben an, wie hell Rot, Grün und Blau leuchten, jeweils von 0 (aus) bis 255 (voll). `rgb` steht für Rot, Grün, Blau.

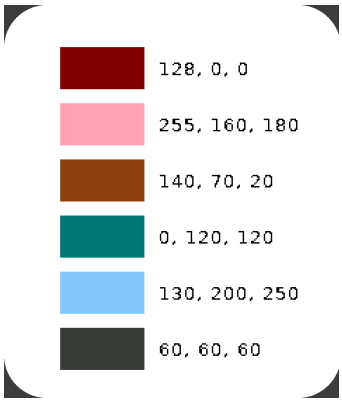
k06_farben_mischen.c

```c
// Eigene Farben mit pbc_rgb(rot, gruen, blau)
#include "pbc.h"

int main(void) {
    pbc_init();
    uint16_t dunkelrot = pbc_rgb(128, 0, 0);
    uint16_t rosa = pbc_rgb(255, 160, 180);
    uint16_t braun = pbc_rgb(140, 70, 20);
    uint16_t petrol = pbc_rgb(0, 120, 120);
    uint16_t hellblau = pbc_rgb(130, 200, 250);
    uint16_t dunkelgrau = pbc_rgb(60, 60, 60);

    pbc_clear(PBC_WHITE);
    pbc_fill_rect(40, 30, 60, 30, dunkelrot);
    pbc_text12("128, 0, 0", 110, 37, PBC_BLACK);
    pbc_fill_rect(40, 70, 60, 30, rosa);
    pbc_text12("255, 160, 180", 110, 77, PBC_BLACK);
    pbc_fill_rect(40, 110, 60, 30, braun);
    pbc_text12("140, 70, 20", 110, 117, PBC_BLACK);
    pbc_fill_rect(40, 150, 60, 30, petrol);
    pbc_text12("0, 120, 120", 110, 157, PBC_BLACK);
    pbc_fill_rect(40, 190, 60, 30, hellblau);
    pbc_text12("130, 200, 250", 110, 197, PBC_BLACK);
    pbc_fill_rect(40, 230, 60, 30, dunkelgrau);
    pbc_text12("60, 60, 60", 110, 237, PBC_BLACK);

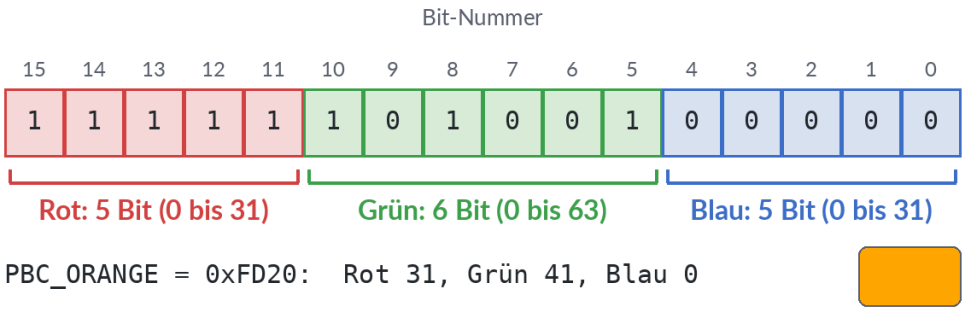
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```




`pbc_rgb` liefert die fertige Farbe als Ergebnis. Zeile 6 bis 11 speichern sie in Variablen vom Typ `uint16_t`, denn eine Farbe ist für den PicoBoy eine Zahl mit 16 Bits (Kapitel 5.2). Danach benutzt du die Variable wie eine der fertigen Farben, etwa in Zeile 14.

> Welche Zahlen ergeben welche Farbe? Suche im Internet nach „Farbwähler“ oder „color picker“. Dort stellst du eine Farbe ein und liest die drei Werte für Rot, Grün und Blau ab, oft mit der Abkürzung RGB. Auch Malprogramme zeigen sie an. Auf dem Display sehen Farben allerdings etwas anders aus als auf einem Computerbildschirm. Probiere es aus.

**Wie eine Farbe gespeichert wird.** Für jedes Pixel hat der PicoBoy genau 16 Bits Platz. Diese 16 Bits werden aufgeteilt: 5 Bits für Rot, 6 für Grün und 5 für Blau. Daher kommt der Name **RGB565**. Grün bekommt ein Bit mehr, weil das menschliche Auge Grüntöne am feinsten unterscheidet.




Mit 5 Bits gibt es für Rot nur 32 Stufen (0 bis 31), nicht 256. `pbc_rgb` rechnet deine Werte von 0 bis 255 deshalb in diese Stufen um. Dabei fallen feine Unterschiede weg: `pbc_rgb(250, 0, 0)` und `pbc_rgb(255, 0, 0)` ergeben dieselbe Farbe. Insgesamt kann der PicoBoy 65 536 Farben anzeigen. Das ist genug für schöne Bilder, und ein ganzes Bild braucht nur 240 · 280 · 2 Bytes = 134 400 Bytes Speicher.

Die elf fertigen Farben aus Kapitel 3 sind in `pbc.h` mit `#define` als solche Zahlen festgelegt, zum Beispiel `#define PBC_RED 0xF800`. Zahlen mit `0x` am Anfang sind **Hexadezimalzahlen**: eine Schreibweise mit sechzehn Ziffern 0 bis 9 und A bis F, in der man Bits gut ablesen kann. Du musst sie nicht selbst ausrechnen. Dafür gibt es `pbc_rgb`.

## 6.3 Text in vier Größen, Zahlen auf dem Display (`snprintf`)

Die vier Schriften kennst du aus Kapitel 3.3: `pbc_text` mit 8 × 8 Pixeln und fester Breite, dazu die Proportionalschriften `pbc_text12`, `pbc_text18` und `pbc_text24`. Ihre Zeilen sind 16, 24 und 32 Pixel hoch. Jetzt lernst du drei Dinge, die das Arbeiten mit Text viel leichter machen.

**Wie breit ist ein Text?** Zu jeder Schrift gibt es eine Funktion, die die Breite eines Texts in Pixeln ausrechnet, ohne ihn zu zeichnen: `pbc_text_width`, `pbc_text12_width`, `pbc_text18_width` und `pbc_text24_width`.

**Wo hört ein Text auf?** Die Textbefehle selbst liefern ebenfalls ein Ergebnis: die x-Position direkt hinter dem gezeichneten Text. Dort kannst du weiterschreiben, zum Beispiel in einer anderen Farbe.

k06_breite.c

```c
// Wie breit ist ein Text? Und wo hoert er auf?
#include "pbc.h"

int main(void) {
    pbc_init();
    printf("Breite von \"Hallo\":\n");
    printf("pbc_text:   %d Pixel\n", pbc_text_width("Hallo"));
    printf("pbc_text12: %d Pixel\n", pbc_text12_width("Hallo"));
    printf("pbc_text18: %d Pixel\n", pbc_text18_width("Hallo"));
    printf("pbc_text24: %d Pixel\n", pbc_text24_width("Hallo"));

    pbc_clear(PBC_BLACK);
```

```
13      int x = pbc_text18("Leben: ", 30, 120, PBC_WHITE);
14      pbc_text18("3", x, 120, PBC_RED);
15      pbc_show();
16      while (true) {
17          sleep_ms(1000);
18      }
19  }
```

Serial Monitor

```
Breite von "Hallo":
pbc_text:   40 Pixel
pbc_text12: 38 Pixel
pbc_text18: 52 Pixel
pbc_text24: 67 Pixel
```




- In Zeile 6 sollen Anführungszeichen ausgegeben werden. Ein `"` würde aber den Text beenden. Mit einem Rückstrich davor, `\"`, wird es zu einem gewöhnlichen Zeichen im Text, so wie `\n` aus Kapitel 4.
- Bei `pbc_text` ist jedes Zeichen 8 Pixel breit: 5 Zeichen sind 40 Pixel. Bei den anderen Schriften hängt die Breite von den Buchstaben ab.
- Zeile 13 speichert das Ergebnis von `pbc_text18` in der Variable `x`. Zeile 14 schreibt die rote 3 genau dort weiter.

**Text in die Mitte setzen.** Mit der Breite kannst du ausrechnen, wo ein Text beginnen muss, damit er genau in der Mitte steht: Vom ganzen Bildschirm bleibt neben dem Text `PBC_WIDTH - breite` frei. Davon soll die Hälfte links und die Hälfte rechts liegen.




Für die senkrechte Mitte geht es genauso mit der Höhe: `PBC_HEIGHT` minus die Höhe der Zeile, geteilt durch 2.

k06_zentrieren.c

```
1  // Text genau in die Mitte setzen
2  #include "pbc.h"
3
4  int main(void) {
5      pbc_init();
6      pbc_clear(PBC_BLUE);
7
8      int breite = pbc_text24_width("Spiel vorbei");
```

```
 9      int x = (PBC_WIDTH - breite) / 2;
10      int y = (PBC_HEIGHT - 32) / 2;  // Zeilenhoehe 32
11      pbc_text24("Spiel vorbei", x, y, PBC_WHITE);
12
13      x = (PBC_WIDTH - pbc_text12_width("Druecke RES")) / 2;
14      pbc_text12("Druecke RES", x, y + 40, PBC_YELLOW);
15
16      pbc_show();
17      while (true) {
18          sleep_ms(1000);
19      }
20  }
```

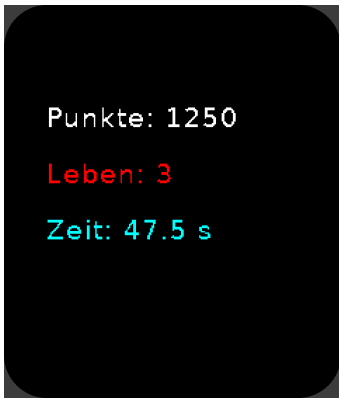
- Zeile 9 rechnet x aus, Zeile 10 y. Die Zeile von `pbc_text24` ist 32 Pixel hoch. Ganz genau in der Mitte steht der Text trotzdem nicht immer: Die ganzzahlige Division schneidet ein halbes Pixel ab, und über den Buchstaben ist in der Zeile etwas Platz frei. Das sieht niemand.
- In Zeile 13 steht der Aufruf von `pbc_text12_width` direkt in der Rechnung. Die Variable `x` bekommt einen neuen Wert; anlegen musst du sie nicht noch einmal.
- Zeile 14 setzt den zweiten Text 40 Pixel unter den ersten.




**Zahlen auf dem Display.** `pbc_text` und seine Geschwister zeichnen nur Texte. Eine Zahl wie `punkte` kannst du ihnen nicht direkt geben. Du musst die Zahl erst in einen Text verwandeln. Dafür gibt es `snprintf`. Es funktioniert genau wie `printf`, schreibt aber nicht in die Konsole, sondern in eine Textvariable:

k06_snprintf.c

```
 1  // Zahlen auf dem Display mit snprintf
 2  #include "pbc.h"
 3
 4  int main(void) {
 5      pbc_init();
 6      int punkte = 1250;
 7      int leben = 3;
 8      float zeit = 47.5;
 9      char text[30];
10
11      pbc_clear(PBC_BLACK);
12      snprintf(text, sizeof text, "Punkte: %d", punkte);
13      pbc_text18(text, 30, 70, PBC_WHITE);
14
15      snprintf(text, sizeof text, "Leben: %d", leben);
16      pbc_text18(text, 30, 110, PBC_RED);
17
18      snprintf(text, sizeof text, "Zeit: %.1f s", zeit);
```



```
19      pbc_text18(text, 30, 150, PBC_CYAN);
20
21      pbc_show();
22      while (true) {
23          sleep_ms(1000);
24      }
25  }
```

- Zeile 9 legt eine Variable für einen Text mit bis zu 30 Zeichen an, wie `name` in Kapitel 4.4.
- Zeile 12: Das erste Argument von `snprintf` ist die Textvariable, das zweite ihre Größe mit `sizeof`. Danach folgen der Text mit Platzhaltern und die Werte, genau wie bei `printf`.
- Zeile 13 zeichnet den fertigen Text.
- Dieselbe Variable `text` lässt sich danach für den nächsten Text benutzen (Zeile 15 und 18). Der alte Inhalt wird dabei überschrieben.

Das `s` in `snprintf` steht für *string*: Das Ergebnis ist eine Zeichenkette. Das `n` steht für die Größe, die du mitgibst. `snprintf` schreibt nie mehr in die Variable, als hineinpasst; ein zu langer Text wird abgeschnitten. Das ist wichtig, denn in C würde zu viel Text einfach über das Ende der Variable hinaus in fremden Speicher geschrieben. Mehr dazu in Kapitel 12.

> Eine Zahl auf dem Display: erst mit `snprintf(text, sizeof text, "…", wert)` in eine Textvariable schreiben, dann `text` zeichnen.

## Ausprobieren: Ein Haus mit Sonne

Dieses Programm zeichnet ein ganzes Bild. Es ist etwas länger, deshalb kommt es in drei Stücken.




k06_haus.c

```c
// Ein Haus mit Sonne
#include "pbc.h"

int main(void) {
    pbc_init();
    uint16_t himmel = pbc_rgb(130, 200, 250);
    uint16_t wiese = pbc_rgb(40, 150, 40);
    uint16_t wand = pbc_rgb(230, 200, 150);
    uint16_t holz = pbc_rgb(140, 70, 20);

    // Himmel, Sonne, Wiese
    pbc_clear(himmel);
    pbc_fill_circle(185, 60, 25, PBC_YELLOW);
    pbc_fill_rect(0, 215, PBC_WIDTH, 65, wiese);
```

Zuerst die eigenen Farben (Zeile 6 bis 9) und der Hintergrund: `pbc_clear` füllt alles mit Himmelblau, dann kommen die Sonne und die Wiese. Die Wiese ist `PBC_WIDTH` breit und geht von y = 215 bis zum unteren Rand (215 + 65 = 280).

k06_haus.c

```
15
16      // Haus mit Dach
17      pbc_fill_rect(50, 140, 110, 90, wand);
18      pbc_line(40, 140, 105, 85, PBC_RED);
19      pbc_line(105, 85, 170, 140, PBC_RED);
20      pbc_hline(40, 140, 131, PBC_RED);
21
22      // Tuer und Fenster
23      pbc_fill_rect(92, 180, 26, 50, holz);
24      pbc_fill_rect(62, 155, 22, 22, PBC_WHITE);
25      pbc_rect(62, 155, 22, 22, holz);
26      pbc_fill_rect(126, 155, 22, 22, PBC_WHITE);
27      pbc_rect(126, 155, 22, 22, holz);
```

Die Wand (Zeile 17) beginnt auf der Wiese, das Haus steht also im Gras. Das Dach sind drei Linien: zwei schräge zur Spitze bei (105, 85) und eine waagrechte als Unterkante. Danach kommen Tür und Fenster auf die Wand. Jedes Fenster besteht aus einem weißen gefüllten Rechteck und einem braunen Umriss an derselben Stelle (Zeile 24 und 25).

k06_haus.c

```
28
29      // Beschriftung, waagrecht zentriert
30      int x = (PBC_WIDTH - pbc_text18_width("Mein Haus")) / 2;
31      pbc_text18("Mein Haus", x, 238, PBC_WHITE);
32
33      pbc_show();
34      while (true) {
35          sleep_ms(1000);
36      }
37  }
```

Zum Schluss die Beschriftung, waagrecht zentriert, und `pbc_show()`.

**Zum Verändern:** Gib dem Haus einen Schornstein mit Rauch aus grauen Kreisen. Setze Wolken aus mehreren weißen Ellipsen an den Himmel. Mach aus dem Tag eine Nacht mit dunkelblauem Himmel, Mond und gelben Fenstern. Das leere Dach füllst du in Kapitel 7 mit einer Schleife.

## Übungsaufgaben

**Aufgabe 1** ⭐ **Eine Flagge** Zeichne die deutsche Flagge über den ganzen Bildschirm: drei gleich hohe Streifen in Schwarz, Rot und Gold. Für Gold passt `pbc_rgb(255, 204, 0)`.

**Aufgabe 2** ⭐ **Zielscheibe** Zeichne eine Zielscheibe aus fünf gefüllten Kreisen mit demselben Mittelpunkt, abwechselnd weiß und rot, in der Mitte gelb. In welcher Reihenfolge musst du die Kreise zeichnen?

**Aufgabe 3 ⭐⭐ Das Schild, genau in der Mitte** In Kapitel 3, Aufgabe 4 hast du ein rotes Schild mit „STOP“ und „Bitte warten“ gezeichnet und die x-Werte ausprobiert. Rechne sie jetzt aus. Setze außerdem beide Zeilen zusammen genau in die senkrechte Mitte.

**Aufgabe 4 ⭐⭐ Ampel** Zeichne eine Ampel: ein dunkelgraues Gehäuse auf einem Mast, darin drei Lichter. Das rote Licht leuchtet. Gelb und Grün sind aus, aber noch zu erkennen. Mische dafür dunkle Farben mit `pbc_rgb`.

**Aufgabe 5 ⭐⭐⭐ Balkendiagramm** Frag in der Konsole nach drei Werten von 0 bis 100, zum Beispiel wie viele Minuten du an drei Tagen gelesen hast. Zeichne für jeden Wert einen Balken, 2 Pixel hoch pro Minute, alle auf derselben Grundlinie. Über jedem Balken steht seine Zahl, zentriert. Tipp: Ein Balken mit der Höhe `h`, der auf der Linie y = 240 steht, beginnt oben bei `240 - h`.

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- `pbc_pixel`, `pbc_line`, `pbc_hline`, `pbc_vline` zeichnen Punkte und Linien.
- `pbc_rect` / `pbc_fill_rect` brauchen die **linke obere Ecke**, Breite und Höhe; `pbc_circle`, `pbc_ellipse` und die `fill_`-Befehle den **Mittelpunkt** und den Radius.
- Was später gezeichnet wird, liegt oben.
- Farben sind 16-Bit-Zahlen (RGB565, Typ `uint16_t`). Eigene Farben mischst du mit `pbc_rgb(r, g, b)`, jeweils 0 bis 255.
- `pbc_text…_width(text)` misst die Breite eines Texts. Mitte: `x = (PBC_WIDTH - breite) / 2`.
- Die Textbefehle liefern die x-Position hinter dem Text.
- Zahlen auf dem Display: `snprintf(text, sizeof text, "%d", zahl);`, dann `text` zeichnen.

KAPITEL 7

# Schleifen und Zeit

**Worum geht's?** Computer sind stark darin, dasselbe sehr oft zu tun. Du lernst, wie ein Programm wartet und die Zeit misst, wie es Anweisungen mit **Schleifen** wiederholt und wie aus vielen einzelnen Bildern eine **Animation** wird. Am Ende läuft ein Ladebalken über den Bildschirm.

## 7.1 Warten mit `sleep_ms`, die Uhr `pbc_millis()`

`sleep_ms(ms)` kennst du schon aus der Endlosschleife am Ende jedes Programms: Es wartet die angegebene Zahl von Millisekunden. 1000 ms sind eine Sekunde. Damit lässt sich zum Beispiel ein Countdown bauen:

k07_countdown.c

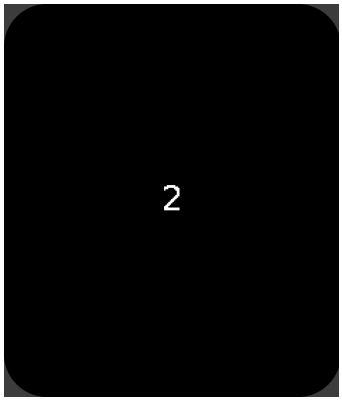
```c
// Ein Countdown - noch ohne Schleife
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_text24("3", 112, 124, PBC_WHITE);
    pbc_show();
    sleep_ms(1000);

    pbc_clear(PBC_BLACK);
    pbc_text24("2", 112, 124, PBC_WHITE);
    pbc_show();
    sleep_ms(1000);

    pbc_clear(PBC_BLACK);
    pbc_text24("1", 112, 124, PBC_WHITE);
    pbc_show();
    sleep_ms(1000);

    pbc_clear(PBC_BLACK);
    pbc_text24("Los!", 92, 124, PBC_GREEN);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```




Jede Zahl steht eine Sekunde lang auf dem Bildschirm (das Bild zeigt die 2), danach erscheint „Los!“. Während `sleep_ms` wartet, tut das Programm nichts anderes. Es reagiert dann auch nicht auf Tasten.

Das Programm funktioniert, aber es ist umständlich: Viermal stehen fast dieselben vier Zeilen untereinander. Bei einem Countdown von 10 wären es vierzig Zeilen. Abschnitt 7.2 zeigt, wie es kürzer geht.

**Die Uhr.** Der PicoBoy zählt ab dem Einschalten die Millisekunden mit. `pbc_millis()` liefert diesen Zählerstand als `uint32_t`. Eine einzelne Zahl daraus ist wenig nützlich: Wie lange `pbc_init()` beim Start braucht, hängt zum Beispiel davon ab, ob ein serieller Monitor verbunden ist. Interessant ist der **Unterschied** zwischen zwei Zeitpunkten:

k07_millis.c

```
1  // Die Uhr des PicoBoy: Millisekunden seit dem Start
2  #include "pbc.h"
3
4  int main(void) {
5      pbc_init();
6      uint32_t start = pbc_millis();
7      sleep_ms(1500);
8      uint32_t ende = pbc_millis();
9
10     int vergangen = ende - start;
11     printf("Zwischen start und ende: %d ms\n", vergangen);
12     while (true) {
13         sleep_ms(1000);
14     }
15 }
```

Serial Monitor

```
Zwischen start und ende: 1500 ms
```

Zeile 6 merkt sich die Zeit vor dem Warten, Zeile 8 die Zeit danach. Die Differenz ist die vergangene Zeit. Sie passt bequem in einen `int` und lässt sich dann mit `%d` ausgeben (Kapitel 5.3). Am PicoBoy kann auch einmal 1501 ms herauskommen: Die Uhr zählt in ganzen Millisekunden, und der Start der Messung fällt meist mitten in eine Millisekunde.

Mit `pbc_millis()` kann ein Programm messen, wie lange etwas gedauert hat, etwa eine Reaktion in einem Spiel (Kapitel 9). Und es kann Dinge nach der Uhr tun, statt mit `sleep_ms` stillzustehen (Aufgabe 5).

> Ein `uint32_t` reicht bis 4 294 967 295. So viele Millisekunden sind gut 49 Tage. Danach läuft die Uhr über und beginnt wieder bei 0 (Kapitel 5.2). Für die Differenz `ende - start` macht das nichts: Wegen des Überlaufs kommt auch dann die richtige Zahl heraus, wenn die Uhr zwischen den beiden Zeitpunkten übergelaufen ist.

## 7.2 `for`, `while`, `do … while`

Eine **Schleife** wiederholt einen Block von Anweisungen. Jeder Durchlauf heißt **Wiederholung**. C hat drei Arten von Schleifen.

**Die `for`-Schleife** ist die richtige Wahl, wenn du zählen willst: von 1 bis 10, von 3 herunter bis 1, für jede Zeile des Bildschirms.

k07_for.c

```c
// Zaehlen mit for
#include "pbc.h"

int main(void) {
    pbc_init();
    for (int i = 1; i <= 10; i++) {
        printf("%d ", i);
    }
    printf("\n");

    for (int i = 10; i > 0; i -= 2) {
        printf("%d ", i);
    }
    printf("\n");
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
1 2 3 4 5 6 7 8 9 10
10 8 6 4 2
```

In den runden Klammern hinter `for` stehen drei Teile, getrennt durch Semikolons:

| Teil | in Zeile 6 | bedeutet |
|---|---|---|
| Start | `int i = 1` | Lege die **Zählvariable** `i` an, beginne bei 1. |
| Bedingung | `i <= 10` | Wiederhole, **solange** `i` kleiner oder gleich 10 ist. |
| Schritt | `i++` | Zähle nach jedem Durchlauf eins hoch. |

Der Ablauf ist immer derselbe: Start. Bedingung prüfen. Ist sie erfüllt, den Block ausführen, dann den Schritt, dann wieder die Bedingung prüfen. Ist sie nicht erfüllt, ist die Schleife zu Ende, und das Programm macht nach der `}` weiter (Zeile 9). Zeile 11 zählt rückwärts in Zweierschritten: Start bei 10, Schritt `i -= 2`, so lange `i > 0` ist.

Eine Zählvariable heißt oft `i`. Das ist eine alte Gewohnheit; ein sprechender Name wie `zeile` ist genauso gut. Die Variable gibt es nur innerhalb der Schleife. Nach der `}` ist sie verschwunden, deshalb darf die zweite Schleife in Zeile 11 wieder ein `i` anlegen.

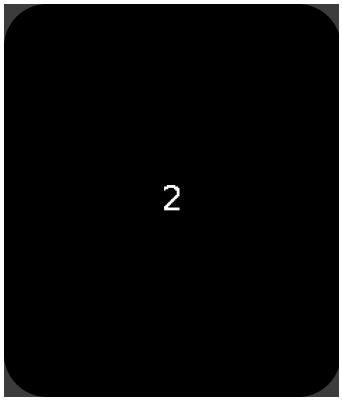
Für die Bedingung gibt es diese **Vergleiche**. Kapitel 8 zeigt sie ausführlich.

| Vergleich | bedeutet |
|---|---|
| `a < b` | a kleiner als b |
| `a <= b` | a kleiner oder gleich b |
| `a > b` | a größer als b |
| `a >= b` | a größer oder gleich b |
| `a == b` | a gleich b (zwei Gleichheitszeichen!) |
| `a != b` | a ungleich b |

Jetzt der Countdown aus Abschnitt 7.1 mit einer Schleife:

k07_countdown_for.c

```
 1 // Der Countdown mit einer for-Schleife
 2 #include "pbc.h"
 3
 4 int main(void) {
 5     pbc_init();
 6     char text[10];
 7     for (int zahl = 3; zahl >= 1; zahl--) {
 8         snprintf(text, sizeof text, "%d", zahl);
 9         int x = (PBC_WIDTH - pbc_text24_width(text)) / 2;
10         pbc_clear(PBC_BLACK);
11         pbc_text24(text, x, 124, PBC_WHITE);
12         pbc_show();
13         sleep_ms(1000);
14     }
15
16     pbc_clear(PBC_BLACK);
17     pbc_text24("Los!", (PBC_WIDTH - pbc_text24_width("Los!")) / 2, 124,
         PBC_GREEN);
18     pbc_show();
19     while (true) {
20         sleep_ms(1000);
21     }
22 }
```




- Die Schleife in Zeile 7 zählt `zahl` von 3 herunter bis 1. Für einen Countdown von 10 änderst du nur die 3.
- Weil die Zahl in einer Variable steht, braucht es `snprintf` (Kapitel 6.3), um sie zu zeichnen. Dafür steht sie jetzt auch genau in der Mitte.
- In Zeile 9 wird mitten im Block eine Variable `x` angelegt. Auch sie gibt es nur innerhalb des Blocks; bei jedem Durchlauf entsteht sie neu.
- In Zeile 17 steht die ganze Rechnung für x direkt im Aufruf. Das geht, macht die Zeile aber lang.

**Die `while`-Schleife** hat nur eine Bedingung. Sie passt, wenn du vorher nicht weißt, wie oft wiederholt werden muss. Eine bekannte Frage: Wie oft muss man ein Blatt Papier (0,1 mm dick) falten, bis es höher ist als der höchste Berg der Erde?

k07_while.c

```c
// Wie oft muss man ein Blatt Papier falten, bis es hoeher
// ist als der Mount Everest (8848 m)?
#include "pbc.h"

int main(void) {
    pbc_init();
    int dicke = 1;              // in Zehntelmillimetern: 0,1 mm
    int falten = 0;
    while (dicke < 88480000) {  // 8848 m = 88 480 000 Zehntelmillimeter
        dicke *= 2;
        falten++;
    }
    float meter = dicke / 10000.0;
    printf("Nach %d Mal Falten ist das Papier %.1f m dick.\n", falten, meter);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Nach 27 Mal Falten ist das Papier 13421.8 m dick.
```

Bei jedem Falten verdoppelt sich die Dicke (Zeile 10), und `falten` zählt mit. Die Schleife läuft, **solange** die Bedingung in Zeile 9 erfüllt ist. Ist das Papier dick genug, geht es in Zeile 13 weiter. Damit die Rechnung mit ganzen Zahlen auskommt, zählt das Programm in Zehntelmillimetern; erst Zeile 13 rechnet in Meter um. Geteilt wird dafür durch `10000.0`, also eine Kommazahl, damit keine ganzzahlige Division entsteht.

`while (true)` am Ende jedes Programms ist auch eine `while`-Schleife. Ihre Bedingung ist immer erfüllt, deshalb endet sie nie.

**Die `do … while`-Schleife** prüft die Bedingung erst **am Ende** jedes Durchlaufs. Der Block wird also mindestens einmal ausgeführt. Das passt genau für Eingaben: erst fragen, dann prüfen, ob die Antwort brauchbar ist.

k07_do_while.c

```c
// Nachfragen, bis die Eingabe passt
#include "pbc.h"

int main(void) {
    pbc_init();
    int anzahl = 0;
    do {
        anzahl = pbc_input_int("Wie viele Sterne? ");
    } while (anzahl < 1);
```

```
    for (int i = 0; i < anzahl; i++) {
        printf("*");
    }
    printf("\n");
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Wie viele Sterne? 0
Wie viele Sterne? -3
Wie viele Sterne? 12
************
```

Zeile 8 fragt, Zeile 9 prüft: Solange die Anzahl kleiner als 1 ist, wird noch einmal gefragt. Beachte das Semikolon hinter `while (anzahl < 1)`. Bei `do … while` gehört es dazu, bei den anderen Schleifen nicht.

> Eine Schleife, deren Bedingung immer erfüllt bleibt, läuft für immer. Bei `for (int i = 0; i < 10; i--)` zählt `i` nach unten und wird nie 10. Der Compiler meldet das nicht, und zur Laufzeit gibt es keine Fehlermeldung: Der PicoBoy bleibt einfach hängen. Dann hilft nur **RES** und ein verbessertes Programm. Prüfe bei jeder Schleife: Ändert sich im Block etwas, sodass die Bedingung irgendwann nicht mehr erfüllt ist?

## 7.3 Verschachtelte Schleifen

Im Block einer Schleife darf wieder eine Schleife stehen. Dann läuft die innere Schleife bei **jedem** Durchlauf der äußeren einmal ganz durch. So entstehen Tabellen:

k07_einmaleins.c

```c
// Das kleine Einmaleins als Tabelle
#include "pbc.h"

int main(void) {
    pbc_init();
    for (int zeile = 1; zeile <= 10; zeile++) {
        for (int spalte = 1; spalte <= 10; spalte++) {
            printf("%4d", zeile * spalte);
        }
        printf("\n");
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
   1   2   3   4   5   6   7   8   9  10
   2   4   6   8  10  12  14  16  18  20
   3   6   9  12  15  18  21  24  27  30
   4   8  12  16  20  24  28  32  36  40
   5  10  15  20  25  30  35  40  45  50
   6  12  18  24  30  36  42  48  54  60
   7  14  21  28  35  42  49  56  63  70
   8  16  24  32  40  48  56  64  72  80
   9  18  27  36  45  54  63  72  81  90
  10  20  30  40  50  60  70  80  90 100
```

Die äußere Schleife (Zeile 6) zählt die Zeilen, die innere (Zeile 7) die Spalten. Für jede Zeile schreibt die innere Schleife zehn Zahlen nebeneinander. Danach beginnt Zeile 10 eine neue Zeile. Insgesamt wird Zeile 8 also 10 · 10 = 100-mal ausgeführt.

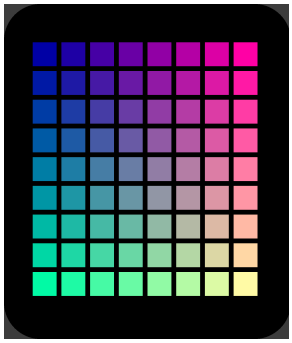
`%4d` in Zeile 8 ist ein Platzhalter mit **Breite**: Die Zahl wird rechts in 4 Stellen eingepasst und links mit Leerzeichen aufgefüllt. So stehen alle Spalten sauber untereinander.

Auf dem Display ist es genauso. Hier ergibt sich aus Zeile und Spalte die Position eines Quadrats und seine Farbe:

k07_raster.c

```
 1 // Ein Raster aus 8 x 9 Quadraten mit Farbverlauf
 2 #include "pbc.h"
 3
 4 int main(void) {
 5     pbc_init();
 6     pbc_clear(PBC_BLACK);
 7     for (int zeile = 0; zeile < 9; zeile++) {
 8         for (int spalte = 0; spalte < 8; spalte++) {
 9             int x = 24 + spalte * 24;
10             int y = 32 + zeile * 24;
11             uint16_t farbe = pbc_rgb(spalte * 36, zeile * 31, 160);
12             pbc_fill_rect(x, y, 20, 20, farbe);
13         }
14     }
15     pbc_show();
16     while (true) {
17         sleep_ms(1000);
18     }
19 }
```

- Die Quadrate sind 20 Pixel groß und liegen im Abstand von 24 Pixeln (Zeile 9 und 10). So bleiben 4 Pixel Lücke.
- 8 Spalten mal 24 Pixel sind 192 Pixel. Damit das Raster waagrecht in der Mitte steht, beginnt es bei x = (240 − 192) / 2 = 24. Senkrecht genauso: (280 − 9 · 24) / 2 = 32.
- Zeile 11 mischt die Farbe: Nach rechts nimmt Rot zu, nach unten Grün, Blau bleibt gleich.



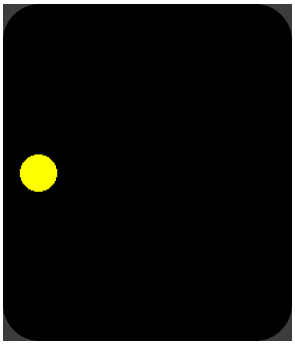
## 7.4 Animation: zeichnen – `pbc_show()` – warten

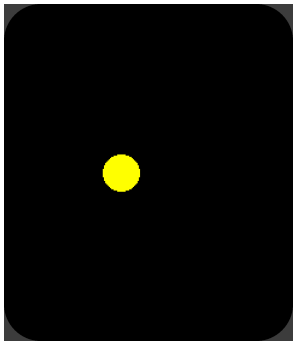
Ein Film besteht aus vielen Einzelbildern, die schnell hintereinander gezeigt werden. Weicht jedes Bild nur ein wenig vom vorigen ab, sieht das Auge eine Bewegung. Auf dem PicoBoy geht das genauso, mit einer Schleife, die immer wieder vier Dinge tut:

1. das Bild **löschen**,
2. alles an seiner neuen Position **zeichnen**,
3. das fertige Bild mit `pbc_show()` zeigen,
4. kurz **warten** und die Position ändern.

k07_ball.c

```
 1 // Ein Ball rollt immer wieder von links nach rechts
 2 #include "pbc.h"
 3
 4 int main(void) {
 5     pbc_init();
 6     int x = 0;
 7     while (true) {
 8         pbc_clear(PBC_BLACK);
 9         pbc_fill_circle(x - 15, 140, 15, PBC_YELLOW);
10         pbc_show();
11         sleep_ms(20);
12         x = (x + 4) % (PBC_WIDTH + 30);
13     }
14 }
```





Die beiden Bilder sind zwei Momente aus der laufenden Animation. Der Ball rollt von links nach rechts, verschwindet am rechten Rand und kommt links wieder herein.

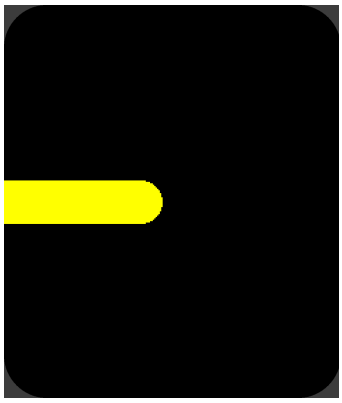
- Zeile 12 bewegt den Ball bei jedem Durchlauf um 4 Pixel. Wie schnell er rollt, hängt von diesen 4 Pixeln ab und davon, wie lange ein Durchlauf dauert: `sleep_ms(20)` plus die Zeit für das Zeichnen und für `pbc_show()`.
- Der Rest mit `%` hält `x` zwischen 0 und 269: Sobald `x + 4` den Wert `PBC_WIDTH + 30` = 270 erreicht, wird daraus wieder eine kleine Zahl.
- Gezeichnet wird bei `x - 15`. Bei `x = 0` liegt der Ball also ganz links außerhalb des Bildschirms, bei 269 ganz rechts außerhalb.

**Warum zuerst löschen?** Lässt du `pbc_clear` weg, bleibt jeder Ball im Bildpuffer stehen, und es entsteht eine Spur:

k07_spur.c

```c
// Der Ball ohne pbc_clear: Er hinterlaesst eine Spur
#include "pbc.h"

int main(void) {
    pbc_init();
    int x = 0;
    while (true) {
        pbc_fill_circle(x - 15, 140, 15, PBC_YELLOW);
        pbc_show();
        sleep_ms(20);
        x = (x + 4) % (PBC_WIDTH + 30);
    }
}
```



**Warum der Bildpuffer?** In Kapitel 3 hast du gelernt, dass alles Gezeichnete erst mit `pbc_show()` sichtbar wird. Jetzt zeigt sich, wozu das gut ist: Würde jeder Befehl sofort auf dem Display erscheinen, sähe man bei jedem Durchlauf kurz den schwarzen, leeren Bildschirm nach `pbc_clear`, und das Bild würde flackern. Mit dem Bildpuffer passieren Löschen und Zeichnen unsichtbar. Das Display bekommt immer nur fertige Bilder. Kapitel 14 geht darauf genauer ein und zeigt, wie man eine gleichmäßige Geschwindigkeit erreicht.

> Eine Animation ist eine Schleife: **löschen – zeichnen – `pbc_show()` – warten**, dann die Position ändern.

## Ausprobieren: Ein Ladebalken

k07_ladebalken.c

```c
// Ein Ladebalken von 0 bis 100 Prozent
#include "pbc.h"

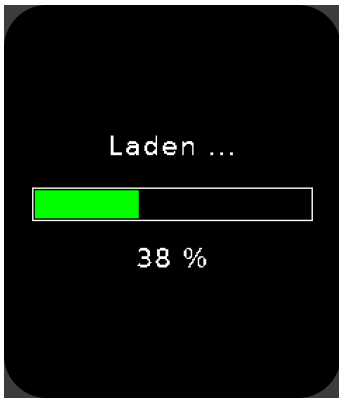
int main(void) {
    pbc_init();
    char text[10];
    int x;
    for (int prozent = 0; prozent <= 100; prozent++) {
        pbc_clear(PBC_BLACK);
        x = (PBC_WIDTH - pbc_text18_width("Laden ...")) / 2;
        pbc_text18("Laden ...", x, 90, PBC_WHITE);

        pbc_rect(20, 130, 200, 24, PBC_WHITE);
        pbc_fill_rect(22, 132, prozent * 196 / 100, 20, PBC_GREEN);

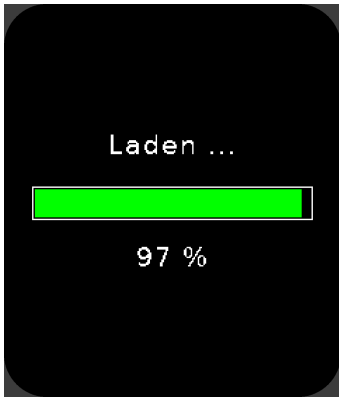
        snprintf(text, sizeof text, "%d %%", prozent);
        x = (PBC_WIDTH - pbc_text18_width(text)) / 2;
        pbc_text18(text, x, 170, PBC_WHITE);
```

```
19          pbc_show();
20          sleep_ms(30);
21      }
22
23      x = (PBC_WIDTH - pbc_text24_width("Fertig!")) / 2;
24      pbc_text24("Fertig!", x, 210, PBC_GREEN);
25      pbc_show();
26      while (true) {
27          sleep_ms(1000);
28      }
29  }
```

- Die Schleife in Zeile 8 läuft von 0 bis 100 Prozent. Jeder Durchlauf zeichnet ein ganzes Bild: Überschrift, Rahmen, Füllung und Prozentzahl.
- Der Rahmen (Zeile 13) ist 200 Pixel breit. Innen bleiben 196 Pixel für die Füllung. Bei `prozent` Prozent ist die Füllung `prozent * 196 / 100` Pixel breit. Die Reihenfolge ist wichtig: Erst malnehmen, dann teilen. `prozent / 100 * 196` wäre wegen der ganzzahligen Division fast immer 0.
- In Zeile 16 steht `%%`. Wie bei `printf` erzeugt es ein `%` im Text.
- Nach der Schleife (Zeile 23 bis 25) wird nicht gelöscht: „Fertig!“ kommt zu dem letzten Bild mit 100 % dazu.




**Zum Verändern:** Lass den Balken langsamer oder schneller laufen. Ändere seine Farbe mit dem Fortschritt, etwa mit `pbc_rgb(255 - prozent * 2, prozent * 2, 0)` von Rot nach Grün. Mach einen senkrechten Balken daraus, der von unten nach oben wächst.




## Übungsaufgaben

**Aufgabe 1** ⭐ **Zählen** Gib in einer Zeile die Zahlen 0, 5, 10, … bis 100 aus. Gib danach einen Countdown von 10 bis 0 aus, jede Zahl in einer eigenen Zeile, und zum Schluss `Start!`.

**Aufgabe 2** ⭐ **Warten auf die Sechs** Würfle mit `pbc_random` so lange, bis eine 6 kommt. Gib jeden Wurf aus und am Ende, wie viele Würfe es waren. Welche Schleife passt am besten?

**Aufgabe 3 ⭐⭐ Ein Dreieck aus Sternen** Frag nach einer Höhe und gib ein Dreieck aus Sternchen aus. Bei der Höhe 5 sieht es so aus:

```
*
**
***
****
*****
```

**Aufgabe 4 ⭐⭐ Das Dach füllen** Das Haus aus Kapitel 6 hat nur ein Dach aus Linien. Fülle es mit einer Schleife: Zeichne für jedes y von der Spitze (y = 85) bis zur Unterkante (y = 140) eine waagrechte Linie mit `pbc_hline`. Oben ist sie 1 Pixel breit, unten reicht sie von x = 40 bis x = 170. Wie breit ist sie dazwischen?

**Aufgabe 5 ⭐⭐⭐ Stoppuhr** Zeige auf dem Display, wie viel Zeit seit dem Start vergangen ist, mit Minuten, Sekunden und Zehntelsekunden: `01:05.2`. Die Anzeige soll sich ständig erneuern. Tipp: Mit `%02d` gibt `snprintf` eine Zahl mit zwei Stellen aus und füllt vorne mit einer 0 auf: aus 5 wird `05`.

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- `sleep_ms(ms)` wartet. `pbc_millis()` liefert die Millisekunden seit dem Start; die Differenz zweier Werte ist die vergangene Zeit.
- `for (start; bedingung; schritt) { … }` zum Zählen, `while (bedingung) { … }`, solange etwas gilt, `do { … } while (bedingung);` für mindestens einen Durchlauf.
- Vergleiche: `<`, `<=`, `>`, `>=`, `==`, `!=`.
- Eine Variable, die in einem Block angelegt wird, gibt es nur in diesem Block.
- Verschachtelte Schleifen: Die innere läuft bei jedem Durchlauf der äußeren ganz durch.
- Animation: **löschen – zeichnen – `pbc_show()` – warten**, Position ändern. Der Bildpuffer verhindert Flackern.

KAPITEL 8

# Entscheidungen

**Worum geht's?** Bisher haben deine Programme immer dasselbe getan. Jetzt lernst du, wie ein Programm abhängig von einer Eingabe, einem Zufall oder einem Zählerstand verschiedene Wege geht. Du verknüpfst Bedingungen, wählst mit `switch` zwischen vielen Fällen, steuerst Schleifen mit `break` und `continue` und lernst eine der bekanntesten Fallen der Sprache C kennen. Zum Schluss spielst du gegen den PicoBoy Zahlenraten.

## 8.1 `if`, `else if`, `else`

Mit `if` (englisch „wenn“) führt ein Programm einen Block nur aus, wenn eine **Bedingung** erfüllt ist. Mit `else` („sonst“) gibt es einen zweiten Block für den anderen Fall. Mit `else if` („sonst, wenn“) lassen sich mehrere Fälle der Reihe nach prüfen:

k08_if.c

```c
// Was ziehe ich an? Entscheidungen mit if, else if und else
#include "pbc.h"

int main(void) {
    pbc_init();
    int grad = pbc_input_int("Wie warm ist es draussen? ");

    if (grad < 0) {
        printf("Achtung, Glatteis!\n");
    }

    if (grad >= 25) {
        printf("T-Shirt und kurze Hose.\n");
    } else if (grad >= 15) {
        printf("Ein Pullover reicht.\n");
    } else if (grad >= 5) {
        printf("Nimm eine Jacke mit.\n");
    } else {
        printf("Muetze, Schal und Handschuhe.\n");
    }
    printf("Viel Spass draussen!\n");
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Wie warm ist es draussen? 18
Ein Pullover reicht.
Viel Spass draussen!
```

- Zeile 8 bis 10: ein `if` ohne `else`. Ist es kälter als 0 Grad, erscheint die Warnung. Sonst passiert hier gar nichts.
- Zeile 12 bis 20: eine Kette. Das Programm prüft die Bedingungen von oben nach unten. Die **erste**, die erfüllt ist, gewinnt; ihr Block wird ausgeführt, alle weiteren werden übersprungen. Bei 18 Grad ist `grad >= 25` nicht erfüllt, `grad >= 15` aber schon: „Ein Pullover reicht." Dass auch `grad >= 5` stimmen würde, spielt keine Rolle mehr.
- Der Block hinter `else` kommt nur dran, wenn keine einzige Bedingung erfüllt war.
- Zeile 21 steht hinter der Kette und wird immer ausgeführt.

Die Bedingung steht immer in runden Klammern. Danach folgt der Block in geschweiften Klammern. C erlaubt bei einer einzigen Anweisung auch, die geschweiften Klammern wegzulassen. Das Buch schreibt sie trotzdem immer: So kann man später nicht aus Versehen eine Zeile anfügen, die dann doch nicht zum `if` gehört.

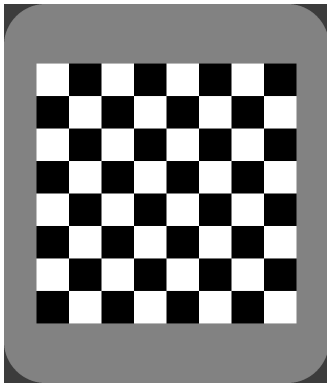
**Entscheidungen beim Zeichnen.** `if` kann überall stehen, auch in einer Schleife. Hier entscheidet es für jedes Feld eines Schachbretts, ob es weiß oder schwarz wird:

k08_schachbrett.c

```c
// Ein Schachbrett: Farbe aus Zeile und Spalte
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_GRAY);
    for (int zeile = 0; zeile < 8; zeile++) {
        for (int spalte = 0; spalte < 8; spalte++) {
            uint16_t farbe = PBC_BLACK;
            if ((zeile + spalte) % 2 == 0) {
                farbe = PBC_WHITE;
            }
            pbc_fill_rect(24 + spalte * 24, 44 + zeile * 24, 24, 24, farbe);
        }
    }
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

Zeile 9 legt die Farbe zuerst als Schwarz an. Ist die Summe aus Zeile und Spalte gerade (der Rest bei der Division durch 2 ist 0), wird sie in Zeile 11 zu Weiß. Ein `else` braucht es hier nicht. Gehst du ein Feld nach rechts oder nach unten, ändert sich die Summe um 1, und die Farbe wechselt.



## 8.2 Vergleiche und logische Verknüpfungen `&&`, `||`, `!`

Die Vergleiche kennst du schon aus den Schleifen in Kapitel 7:

| Vergleich | bedeutet |
|---|---|
| `a < b`, `a <= b` | kleiner, kleiner oder gleich |
| `a > b`, `a >= b` | größer, größer oder gleich |
| `a == b` | gleich |
| `a != b` | ungleich |

Ein Vergleich ist in C eine Rechnung wie jede andere. Ihr Ergebnis ist **1**, wenn er stimmt, und **0**, wenn nicht:

k08_vergleiche.c

```c
// Ein Vergleich ergibt 1 (wahr) oder 0 (falsch)
#include "pbc.h"

int main(void) {
    pbc_init();
    int a = 7;
    int b = 3;
    printf("a < b   ergibt %d\n", a < b);
    printf("a > b   ergibt %d\n", a > b);
    printf("a == 7  ergibt %d\n", a == 7);
    printf("a != 7  ergibt %d\n", a != 7);

    bool groesser = a > b;
    if (groesser) {
        printf("%d ist groesser als %d.\n", a, b);
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
a < b   ergibt 0
a > b   ergibt 1
a == 7  ergibt 1
a != 7  ergibt 0
7 ist groesser als 3.
```

Deshalb lässt sich das Ergebnis eines Vergleichs auch in einer Variable vom Typ `bool` speichern (Zeile 13) und später als Bedingung benutzen (Zeile 14). Ein guter Name wie `groesser` macht das `if` gut lesbar.

> Für C ist **0 falsch** und **jede andere Zahl wahr**. Ein Vergleich ergibt 1 oder 0, `true` ist 1, `false` ist 0.

**Bedingungen verknüpfen.** Oft hängt eine Entscheidung von mehreren Dingen ab. Dafür gibt es drei **logische Operatoren**:

| Zeichen | heißt | ist wahr, wenn … |
|---|---|---|
| `a && b` | und | **beide** wahr sind |
| `a \|\| b` | oder | **mindestens eine** wahr ist |
| `!a` | nicht | `a` falsch ist |

Das Zeichen `|` heißt senkrechter Strich. Du tippst es unter Windows mit **AltGr + <** (die Taste links neben Y), am Mac mit **Alt + 7**. Das `&` kommt mit **Umschalt + 6**.

k08_logik.c

```c
// Mehrere Bedingungen verknuepfen: &&, || und !
#include "pbc.h"

int main(void) {
    pbc_init();
    int tag = pbc_input_int("Tag (1 = Montag ... 7 = Sonntag): ");
    int grad = pbc_input_int("Temperatur: ");

    bool wochenende = tag == 6 || tag == 7;
    bool warm = grad >= 22;

    if (wochenende && warm) {
        printf("Ab ins Freibad!\n");
    }
    if (wochenende && !warm) {
        printf("Ein Tag fuer das Museum.\n");
    }
    if (!wochenende) {
        printf("Erst Schule oder Arbeit.\n");
    }
    if (tag < 1 || tag > 7) {
        printf("Den Tag %d gibt es nicht.\n", tag);
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Tag (1 = Montag ... 7 = Sonntag): 6
Temperatur: 25
Ab ins Freibad!
```

- Zeile 9: Wochenende ist, wenn der Tag 6 **oder** 7 ist.

- Zeile 12: Ins Freibad geht es nur, wenn Wochenende **und** warm ist.
- Zeile 15 und 18: `!warm` heißt „nicht warm“, `!wochenende` „kein Wochenende“.
- Zeile 21 prüft, ob die Eingabe ungültig ist: kleiner als 1 **oder** größer als 7.

> In der Mathematik schreibt man `1 ≤ tag ≤ 7`. In C geht das nicht so. Du musst zwei Vergleiche mit `&&` verbinden: `tag >= 1 && tag <= 7`.

Wie bei Punkt- und Strichrechnung gibt es eine Reihenfolge: Vergleiche kommen vor `&&`, und `&&` kommt vor `||`. Wenn du beides mischst, setze Klammern. Dann ist klar, was gemeint ist, auch für dich selbst.

## 8.3 `switch` und `case`

Soll ein Programm je nach dem Wert **einer** ganzen Zahl etwas anderes tun, wird eine `else if`-Kette schnell lang. Dafür gibt es `switch`:

k08_switch.c

```c
// Eine Zahl wird zum Namen des Wochentags
#include "pbc.h"

int main(void) {
    pbc_init();
    int tag = pbc_input_int("Tag (1 bis 7): ");
    switch (tag) {
    case 1:
        printf("Montag\n");
        break;
    case 2:
        printf("Dienstag\n");
        break;
    case 3:
        printf("Mittwoch\n");
        break;
    case 4:
        printf("Donnerstag\n");
        break;
    case 5:
        printf("Freitag\n");
        break;
    case 6:
    case 7:
        printf("Wochenende\n");
        break;
    default:
        printf("Diesen Tag gibt es nicht.\n");
        break;
    }
```

```
31      while (true) {
32          sleep_ms(1000);
33      }
34  }
```

Serial Monitor

```
Tag (1 bis 7): 3
Mittwoch
```

- `switch (tag)` in Zeile 6 schaut sich den Wert von `tag` an.
- Das Programm springt zu dem `case`, dessen Zahl passt, und macht dort weiter.
- `break` beendet den `switch`. Das Programm geht hinter der schließenden `}` weiter (Zeile 31).
- Zu `default` springt das Programm, wenn kein `case` passt. Es ist wie das `else` am Ende einer Kette.
- Zeile 23 und 24: Mehrere `case` dürfen untereinanderstehen. Für 6 und 7 wird derselbe Block ausgeführt.

Hinter `case` steht immer eine feste Zahl oder ein Zeichen wie `'a'`, kein Vergleich und keine Variable. Für Bereiche wie „ab 25 Grad“ brauchst du weiterhin `if`.

**Das vergessene `break`.** Ohne `break` hört ein `case` nicht auf, sondern läuft in den nächsten hinein:

k08_ohne_break.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int tag = pbc_input_int("Tag (1 bis 7): ");
    switch (tag) {
    case 1:
        printf("Montag\n");
    case 2:
        printf("Dienstag\n");
        break;
    default:
        printf("ein anderer Tag\n");
        break;
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
main.c: In function 'main':
main.c:8:9: warning: this statement may fall through [-Wimplicit-fallthrough=]
    8 |         printf("Montag\n");
      |         ^~~~~~~~~~~~~~~~~~
main.c:9:5: note: here
    9 |     case 2:
      |     ^~~~
```

Serial Monitor

```
Tag (1 bis 7): 1
Montag
Dienstag
```

Bei der Eingabe 1 erscheinen Montag **und** Dienstag. Das Programm ist in Zeile 8 in den `case 2` „hineingefallen“ (englisch *fall through*) und erst beim `break` in Zeile 11 ausgestiegen. Zum Glück warnt der Compiler: *this statement may fall through*, diese Anweisung fällt möglicherweise durch. `note: here` zeigt, wohin.

## 8.4 `break` und `continue`

`break` beendet nicht nur einen `switch`, sondern auch eine **Schleife**, und zwar sofort. `continue` springt dagegen an das Ende des aktuellen Durchlaufs. Die Schleife macht dann mit dem nächsten Durchlauf weiter.

k08_summe.c

```c
// Zahlen addieren, bis 0 kommt - mit break und continue
#include "pbc.h"

int main(void) {
    pbc_init();
    int summe = 0;
    while (true) {
        int zahl = pbc_input_int("Zahl (0 = Ende): ");
        if (zahl == 0) {
            break;                  // Schleife sofort verlassen
        }
        if (zahl < 0) {
            printf("Nur positive Zahlen, bitte.\n");
            continue;               // weiter mit der naechsten Eingabe
        }
        summe += zahl;
    }
    printf("Summe: %d\n", summe);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Zahl (0 = Ende): 5
Zahl (0 = Ende): -2
Nur positive Zahlen, bitte.
Zahl (0 = Ende): 10
Zahl (0 = Ende): 0
Summe: 15
```

- Die Schleife in Zeile 7 ist eine Endlosschleife. Trotzdem endet sie: Bei der Eingabe 0 verlässt `break` in Zeile 10 sie, und es geht in Zeile 18 weiter.
- Bei einer negativen Zahl springt `continue` in Zeile 14 zurück an den Anfang der Schleife. Zeile 16 wird für diese Zahl übersprungen.

`while (true)` mit einem `break` darin ist praktisch, wenn die Entscheidung zum Aufhören mitten in der Schleife fällt, wie hier nach der Eingabe.

Auch eine `for`-Schleife lässt sich mit `break` früher beenden, zum Beispiel bei einer Suche, sobald das Gesuchte gefunden ist:

k08_teiler.c

```
 1 // Den kleinsten Teiler einer Zahl suchen
 2 #include "pbc.h"
 3
 4 int main(void) {
 5     pbc_init();
 6     int zahl = pbc_input_int("Zahl (ab 2): ");
 7     int teiler;
 8     for (teiler = 2; teiler < zahl; teiler++) {
 9         if (zahl % teiler == 0) {
10             break;                  // gefunden, nicht weitersuchen
11         }
12     }
13     if (teiler < zahl) {
14         printf("%d = %d * %d\n", zahl, teiler, zahl / teiler);
15     } else {
16         printf("%d ist eine Primzahl.\n", zahl);
17     }
18     while (true) {
19         sleep_ms(1000);
20     }
21 }
```

Serial Monitor

```
Zahl (ab 2): 91
91 = 7 * 13
```

Die Schleife probiert die Teiler 2, 3, 4, … durch. Beim ersten, der ohne Rest aufgeht, hört sie auf. Weil `teiler` diesmal **vor** der Schleife angelegt ist (Zeile 7), gibt es die Variable auch danach noch. Zeile 13 prüft: Hat die Schleife vorzeitig aufgehört, ist `teiler` kleiner als die Zahl. Ist sie ganz durchgelaufen, gibt es keinen Teiler, und die Zahl ist eine Primzahl.

> Sind Schleifen ineinander verschachtelt oder steht ein `switch` in einer Schleife, beendet `break` nur das **Innerste** davon. Ein `break` in einem `case` beendet also den `switch`, nicht die Schleife um ihn herum.

## 8.5 Typische C-Falle: `=` statt `==`

Ein einzelnes `=` ist eine Zuweisung, zwei `==` sind ein Vergleich. Wer sich vertippt, bekommt ein Programm, das trotzdem gebaut wird:

k08_falle.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int zahl = pbc_input_int("Gib eine Zahl ein: ");
    if (zahl = 7) {
        printf("Das ist meine Glueckszahl!\n");
    }
    printf("Die Zahl ist jetzt %d.\n", zahl);
    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
Linking CXX executable k08_falle.elf
Built target k08_falle
main.c: In function 'main':
main.c:6:9: warning: suggest parentheses around assignment used as truth value [-
  Wparentheses]
    6 |     if (zahl = 7) {
      |         ^~~~
```

Serial Monitor

```
Gib eine Zahl ein: 3
Das ist meine Glueckszahl!
Die Zahl ist jetzt 7.
```

Die Eingabe war 3, trotzdem meldet das Programm die Glückszahl, und danach ist die Zahl 7. In Zeile 6 steht keine Frage, sondern ein Befehl: Speichere 7 in `zahl`. Eine Zuweisung hat in C selbst einen Wert, nämlich den zugewiesenen: 7. Und 7 ist nicht 0, also wahr. Die Bedingung ist deshalb **immer** erfüllt, und die Eingabe ist obendrein überschrieben.

Der Compiler ahnt das und warnt: *suggest parentheses around assignment used as truth value*, also: Wenn du hier wirklich eine Zuweisung als Bedingung meinst, setze Klammern darum. Gemeint ist aber fast immer `==`.

**Ein Semikolon zu viel.** Eine zweite Falle sieht ganz harmlos aus:

k08_semikolon.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int zahl = pbc_input_int("Gib eine Zahl ein: ");
    if (zahl > 100);
    {
        printf("Das ist eine grosse Zahl.\n");
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
Linking CXX executable k08_semikolon.elf
Built target k08_semikolon
main.c: In function 'main':
main.c:6:20: warning: suggest braces around empty body in an 'if' statement [-Wempty-
  body]
    6 |     if (zahl > 100);
      |                    ^
main.c:6:5: warning: this 'if' clause does not guard... [-Wmisleading-indentation]
    6 |     if (zahl > 100);
      |     ^~
main.c:7:5: note: ...this statement, but the latter is misleadingly indented as if it
  were guarded by the 'if'
    7 |     {
      |     ^
```

Serial Monitor

```
Gib eine Zahl ein: 5
Das ist eine grosse Zahl.
```

Das `;` am Ende von Zeile 6 ist für C eine leere Anweisung. Sie ist alles, was zum `if` gehört. Der Block darunter ist ein gewöhnlicher Block, der immer ausgeführt wird, auch bei der Zahl 5. Der Compiler meldet gleich zwei Verdachtsfälle: einen *empty body*, einen leeren Rumpf, und dass die Einrückung in die Irre führt (*misleadingly indented*). Die Einrückung lässt den Block so aussehen, als gehöre er zum `if`; der Compiler richtet sich aber nicht nach Einrückungen.

> Vergleich: `==`. Zuweisung: `=`. Hinter `if (…)` kommt **kein** Semikolon. Bei beiden Fehlern hilft der Compiler mit einer Warnung, deshalb: keine Warnung stehen lassen.

## Ausprobieren: Zahlen raten

Der PicoBoy denkt sich eine Zahl von 1 bis 100 aus. Du rätst, er sagt dir, ob dein Tipp zu groß oder zu klein ist.

k08_zahlenraten.c

```c
// Zahlen raten: Der PicoBoy denkt sich eine Zahl aus
#include "pbc.h"

#define GROESSTE_ZAHL 100

int main(void) {
    pbc_init();
    int geheim = pbc_random(1, GROESSTE_ZAHL);
    int versuche = 0;
    printf("Ich denke mir eine Zahl von 1 bis %d.\n", GROESSTE_ZAHL);

    while (true) {
        int tipp = pbc_input_int("Dein Tipp: ");
        versuche++;
        if (tipp < geheim) {
            printf("Zu klein.\n");
        } else if (tipp > geheim) {
            printf("Zu gross.\n");
        } else {
            break;
        }
    }
    printf("Richtig! Du hast %d Versuche gebraucht.\n", versuche);

    char text[30];
    snprintf(text, sizeof text, "%d Versuche", versuche);
    pbc_clear(PBC_BLACK);
    int x = (PBC_WIDTH - pbc_text24_width("Richtig!")) / 2;
    pbc_text24("Richtig!", x, 100, PBC_GREEN);
    x = (PBC_WIDTH - pbc_text18_width(text)) / 2;
    pbc_text18(text, x, 145, PBC_WHITE);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

```
Serial Monitor
Ich denke mir eine Zahl von 1 bis 100.
Dein Tipp: 50
Zu gross.
Dein Tipp: 25
Zu klein.
Dein Tipp: 37
Zu gross.
Dein Tipp: 31
Richtig! Du hast 4 Versuche gebraucht.
```




- Zeile 8 wählt die geheime Zahl mit `pbc_random`. Bei dir ist es bei jedem Start eine andere.
- Die Schleife in Zeile 12 läuft, bis der Tipp stimmt. Dann verlässt sie `break` in Zeile 20. Jeder Durchlauf zählt einen Versuch.
- Die Kette in Zeile 15 bis 21 hat drei Fälle: zu klein, zu groß und, wenn beides nicht stimmt, richtig.
- Zum Schluss erscheint das Ergebnis auch auf dem Display.

**Zum Verändern:** Mach mit `GROESSTE_ZAHL` in Zeile 4 das Spiel schwerer oder leichter. Begrenze die Zahl der Versuche auf 7 und lass das Programm danach die Lösung verraten. Mit einer klugen Strategie reichen 7 Versuche immer aus. Welche?

## Übungsaufgaben

**Aufgabe 1** ⭐ **Gerade oder ungerade** Lies eine Zahl ein und gib aus, ob sie gerade oder ungerade ist. Tipp: `%`.

**Aufgabe 2** ⭐ **Noten** Lies eine Punktzahl von 0 bis 100 ein und gib die Note aus: ab 90 Punkten eine 1, ab 80 eine 2, ab 65 eine 3, ab 50 eine 4, ab 30 eine 5, darunter eine 6. Ist die Punktzahl kleiner als 0 oder größer als 100, soll eine Meldung erscheinen.

**Aufgabe 3** ⭐⭐ **Schaltjahr** Ein Jahr ist ein Schaltjahr, wenn es durch 4 teilbar ist, aber nicht durch 100. Durch 400 teilbare Jahre sind trotzdem Schaltjahre. 2024 und 2000 waren Schaltjahre, 1900 nicht. Lies ein Jahr ein und gib aus, ob es ein Schaltjahr ist. Schaffst du es mit einer einzigen Bedingung?

**Aufgabe 4** ⭐⭐ **Taschenrechner mit Auswahl** Lies zwei ganze Zahlen ein und danach die Rechenart als Zahl: 1 = plus, 2 = minus, 3 = mal, 4 = geteilt. Rechne mit `switch`. Beim Teilen gib das Ergebnis mit Rest aus, zum Beispiel `17 / 5 = 3 Rest 2`. Was muss das Programm tun, wenn die zweite Zahl 0 ist? Denk an Kapitel 5.3.

**Aufgabe 5 ⭐⭐⭐ Der PicoBoy rät** Jetzt umgekehrt: Du denkst dir eine Zahl von 1 bis 100, und der PicoBoy rät. Nach jedem Tipp antwortest du mit 1 (zu klein), 2 (zu groß) oder 3 (richtig). Der PicoBoy merkt sich, in welchem Bereich die Zahl noch liegen kann, und tippt immer auf die Mitte dieses Bereichs. Am Ende sagt er, wie viele Versuche er gebraucht hat.

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- `if (bedingung) { … } else if (bedingung) { … } else { … }`: Die erste erfüllte Bedingung gewinnt.
- Vergleiche ergeben 1 (wahr) oder 0 (falsch). Für C ist jede Zahl außer 0 wahr.
- `&&` und, `||` oder, `!` nicht. Bereiche so prüfen: `x >= 1 && x <= 7`.
- `switch (zahl) { case 1: … break; default: … }`. Ohne `break` fällt das Programm in den nächsten `case`.
- `break` beendet eine Schleife sofort, `continue` springt zum nächsten Durchlauf.
- `==` vergleicht, `=` weist zu. Kein `;` hinter `if (…)`.

KAPITEL 9

# Tasten und Joystick

**Worum geht's?** Jetzt reagiert dein Programm auf den Joystick und die Tasten A und B. Du lernst den Unterschied zwischen „die Taste ist gerade unten" und „die Taste wurde gedrückt", steuerst einen Punkt über den Bildschirm und wartest auf eine bestimmte Taste. Am Ende steht ein Reaktionstest, der auf die Millisekunde genau misst.

Der PicoBoy hat sieben Eingaben: den Joystick mit den Richtungen hoch, runter, links und rechts, das Hineindrücken des Joysticks (die Taste **MITTE**) und die Tasten **A** (links) und **B** (rechts). Für jede gibt es in der Buchbibliothek zwei Funktionen:

| Eingabe | solange gedrückt | einmal pro Druck |
|---|---|---|
| Joystick hoch | `pbc_pressed_up()` | `pbc_was_pressed_up()` |
| Joystick runter | `pbc_pressed_down()` | `pbc_was_pressed_down()` |
| Joystick links | `pbc_pressed_left()` | `pbc_was_pressed_left()` |
| Joystick rechts | `pbc_pressed_right()` | `pbc_was_pressed_right()` |
| Joystick drücken (MITTE) | `pbc_pressed_center()` | `pbc_was_pressed_center()` |
| Taste A | `pbc_pressed_a()` | `pbc_was_pressed_a()` |
| Taste B | `pbc_pressed_b()` | `pbc_was_pressed_b()` |

Alle diese Funktionen brauchen kein Argument, die Klammern bleiben leer. Dafür **geben sie einen Wert zurück**: `true` oder `false`, also einen `bool` (Kapitel 5). Einen solchen Aufruf kannst du deshalb direkt als Bedingung in ein `if` oder `while` schreiben.

## 9.1 Gedrückt halten: `pbc_pressed_up()` …

`pbc_pressed_a()` fragt nach, ob die Taste A **in diesem Moment** unten ist. Das folgende Programm zeigt einen grünen Kreis, solange du A gedrückt hältst:

k09_halten.c

```c
// Ein Kreis, solange A gedrueckt ist
#include "pbc.h"

int main(void) {
    pbc_init();
    while (true) {
        pbc_clear(PBC_BLACK);
        pbc_text18("Halte A gedrueckt", 32, 50, PBC_WHITE);
```

```
 9         if (pbc_pressed_a()) {
10             pbc_fill_circle(120, 160, 50, PBC_GREEN);
11         }
12         pbc_show();
13         sleep_ms(20);
14     }
15 }
```




Die Endlosschleife ab Zeile 6 zeichnet immer wieder ein neues Bild: löschen, Text schreiben, eventuell den Kreis, zeigen, kurz warten. Bei jedem Durchlauf fragt Zeile 9 die Taste neu ab. Ist A unten, liefert `pbc_pressed_a()` den Wert `true`, und Zeile 10 malt den Kreis. Lässt du los, liefert die Funktion `false`; das nächste Bild hat keinen Kreis mehr.

Wie oft wird gefragt? `pbc_show()` dauert gut 31 ms, dazu kommen die 20 ms aus Zeile 13. Die Schleife läuft also ungefähr 20-mal in der Sekunde. Das ist schnell genug, dass du keine Verzögerung bemerkst.

**Die Klammern nicht vergessen.** `if (pbc_pressed_a)` ohne `()` ruft die Funktion nicht auf. Der Name allein steht für die Funktion selbst, und die zählt als „wahr“: Der Kreis wäre immer zu sehen. Der Compiler bemerkt das und warnt:

Terminal
```
main.c: In function 'main':
main.c:8:13: warning: the address of 'pbc_pressed_a' will always evaluate as
  'true' [-Waddress]
    8 |         if (pbc_pressed_a) {
      |             ^~~~~~~~~~~~~
```

Eine Warnung ist hier also ein echter Fehler.

Mit `else if` (Kapitel 8) fragst du mehrere Tasten nacheinander ab. Das folgende Programm zeigt, in welche Richtung der Joystick gerade zeigt:

k09_joystick.c
```
 1 // Zeigt, in welche Richtung der Joystick gerade gedrueckt ist
 2 #include "pbc.h"
 3
 4 int main(void) {
 5     pbc_init();
 6     while (true) {
 7         pbc_clear(PBC_BLACK);
 8         pbc_text12("Joystick:", 40, 100, PBC_GRAY);
 9         if (pbc_pressed_up()) {
10             pbc_text24("hoch", 40, 120, PBC_YELLOW);
11         } else if (pbc_pressed_down()) {
12             pbc_text24("runter", 40, 120, PBC_YELLOW);
```

```
13          } else if (pbc_pressed_left()) {
14              pbc_text24("links", 40, 120, PBC_YELLOW);
15          } else if (pbc_pressed_right()) {
16              pbc_text24("rechts", 40, 120, PBC_YELLOW);
17          } else if (pbc_pressed_center()) {
18              pbc_text24("MITTE", 40, 120, PBC_YELLOW);
19          } else {
20              pbc_text24("-", 40, 120, PBC_GRAY);
21          }
22          pbc_show();
23          sleep_ms(20);
24      }
25  }
```

Durch die `else if`-Kette erscheint immer nur **ein** Wort: das der ersten Abfrage, die `true` liefert. Ist gar nichts gedrückt, greift das `else` in Zeile 19 und zeigt einen grauen Strich. Das Bild entstand, während der Joystick nach links gedrückt war.




Es gibt noch `pbc_any_pressed()`. Die Funktion liefert `true`, wenn **irgendeine** der sieben Eingaben gerade unten ist.

> **Was in der Taste passiert.** Jede Taste verbindet beim Drücken einen Anschluss (Pin) des Mikrocontrollers mit dem Minuspol. `pbc_pressed_a()` liest, ob an diesem Pin gerade Spannung anliegt oder nicht. Wie das ohne Bibliothek geht, zeigt Kapitel 21.

## 9.2 Einmal pro Druck: `pbc_was_pressed_a()` …

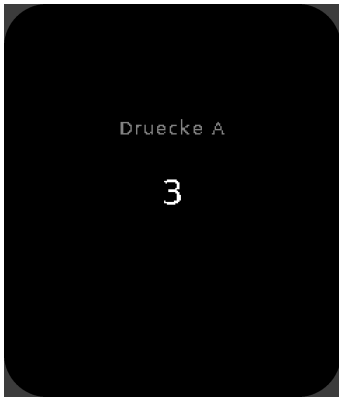
Ein Zähler soll bei jedem Druck auf A um 1 weiterzählen:

k09_zaehler.c

```
1  // Zaehlt jeden Druck auf A
2  #include "pbc.h"
3
4  int main(void) {
5      pbc_init();
6      int zaehler = 0;
7      char text[20];
8      while (true) {
9          if (pbc_was_pressed_a()) {
10             zaehler++;
11             printf("Zaehler: %d\n", zaehler);
12         }
13         pbc_clear(PBC_BLACK);
14         pbc_text12("Druecke A", 82, 80, PBC_GRAY);
```

```
15          snprintf(text, sizeof text, "%d", zaehler);
16          pbc_text24(text, (PBC_WIDTH - pbc_text24_width(text)) / 2, 120,
              PBC_WHITE);
17          pbc_show();
18          sleep_ms(20);
19      }
20  }
```

`pbc_was_pressed_a()` in Zeile 9 liefert `true`, wenn A **seit der letzten Abfrage neu gedrückt** wurde, und zwar genau ein Mal pro Druck. Bei der nächsten Abfrage ist der Druck „abgeholt“, die Funktion liefert wieder `false`, bis du loslässt und erneut drückst. Zeile 11 schreibt den neuen Stand zusätzlich in die Konsole.




Was passiert, wenn in Zeile 9 stattdessen `pbc_pressed_a()` steht? Links das richtige Programm nach drei kurzen Drücken, rechts die falsche Fassung (`k09_zaehler_falsch.c`) nach einem einzigen Druck von 0,3 Sekunden:

mit pbc_was_pressed_a(): 3 Drücke

```
Zaehler: 1
Zaehler: 2
Zaehler: 3
```

mit pbc_pressed_a(): 1 Druck

```
Zaehler: 1
Zaehler: 2
Zaehler: 3
Zaehler: 4
Zaehler: 5
```

Mit `pbc_pressed_a()` zählt jeder Durchlauf, in dem die Taste unten ist. Ein Druck von 0,3 Sekunden dauert hier fünf Durchläufe.

**`pbc_was_pressed_…` merkt sich den Druck.** Wie kann die Bibliothek wissen, dass eine Taste *gedrückt wurde*, wenn das Programm gerade gar nicht nachgesehen hat? Dafür sorgt ein **Interrupt** (Unterbrechung): Sobald eine Taste hinuntergeht, unterbricht der Prozessor kurz, was er gerade tut, und führt ein winziges Stück der Bibliothek aus. Das notiert sich: „A wurde gedrückt“. Danach macht dein Programm genau dort weiter, wo es war, und merkt davon nichts. `pbc_was_pressed_a()` schaut später nur nach, ob die Notiz da ist, und löscht sie.

Das zeigt das folgende Programm. Während der drei Sekunden in Zeile 7 wurde A kurz gedrückt und wieder losgelassen:

k09_merken.c

```
1 // was_pressed merkt sich einen Druck, auch wenn das Programm beschaeftigt ist
2 #include "pbc.h"
3
4 int main(void) {
5     pbc_init();
6     printf("Drei Sekunden lang beschaeftigt ...\n");
```

```
 7     sleep_ms(3000);
 8     printf("pbc_pressed_a():     %d\n", pbc_pressed_a());
 9     printf("pbc_was_pressed_a(): %d\n", pbc_was_pressed_a());
10     printf("pbc_was_pressed_a(): %d\n", pbc_was_pressed_a());
11     while (true) {
12         sleep_ms(1000);
13     }
14 }
```

Serial Monitor

```
Drei Sekunden lang beschaeftigt ...
pbc_pressed_a():     0
pbc_was_pressed_a(): 1
pbc_was_pressed_a(): 0
```

`printf` gibt einen `bool` mit `%d` als Zahl aus: `1` für `true`, `0` für `false`. Zeile 8 meldet `0`, denn jetzt ist die Taste nicht mehr unten. Zeile 9 meldet `1`: Der Druck wurde gemerkt. Die zweite Abfrage in Zeile 10 meldet `0`, denn der Druck ist schon abgeholt.

**Nur einmal pro Durchlauf fragen.** Weil die Abfrage den Druck abholt, darf in einer Schleife nur **eine** Stelle `pbc_was_pressed_a()` aufrufen. Ein zweiter Aufruf kurz danach findet die Notiz nicht mehr. Brauchst du das Ergebnis mehrmals, speichere es in einer Variable:

```
bool a = pbc_was_pressed_a();
if (a && leben > 0) { … }
if (a && leben == 0) { … }
```

Außerdem merkt sich die Bibliothek nur, **dass** gedrückt wurde, nicht **wie oft**. Drückst du dreimal, während das Programm beschäftigt ist, liefert die nächste Abfrage einmal `true`.

**Entprellen.** Eine Taste ist ein Schalter mit zwei Metallkontakten. Beim Drücken und beim Loslassen federn sie ein paar tausendstel Sekunden lang nach: Der Kontakt schließt, öffnet, schließt wieder. Man sagt, die Taste **prellt**. Ohne Gegenmaßnahme würde ein einziger Druck wie mehrere aussehen. Die Bibliothek **entprellt** deshalb: Ein Druck zählt nur, wenn die Taste davor mindestens 50 ms ruhig war. So zählt auch das Federn beim Loslassen nicht als neuer Druck. Schneller drückt sowieso niemand.

| Du willst … | Nimm … |
|---|---|
| etwas tun, **solange** die Taste unten ist (bewegen, Dauerfeuer, Licht) | `pbc_pressed_…()` |
| etwas **einmal pro Druck** tun (zählen, umschalten, Menü, springen) | `pbc_was_pressed_…()` |

## 9.3 Ein Punkt, der sich steuern lässt

Jetzt kommt beides zusammen: Der Joystick bewegt einen Punkt, solange er gedrückt ist, und jeder Druck auf A gibt dem Punkt eine neue, zufällige Farbe.

k09_punkt.c

```c
// Ein Punkt, der sich mit dem Joystick steuern laesst
#include "pbc.h"

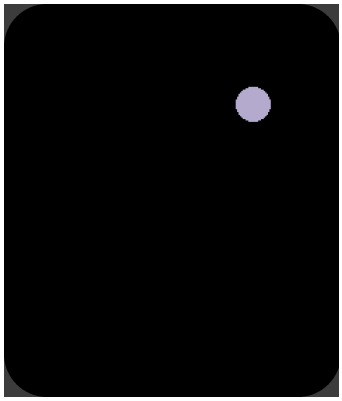
#define RADIUS 12
#define SCHRITT 3

int main(void) {
    pbc_init();
    int x = 120;
    int y = 140;
    uint16_t farbe = PBC_YELLOW;
    while (true) {
        // Lesen und rechnen
        if (pbc_pressed_left() && x > RADIUS) {
            x -= SCHRITT;
        }
        if (pbc_pressed_right() && x < PBC_WIDTH - RADIUS) {
            x += SCHRITT;
        }
        if (pbc_pressed_up() && y > RADIUS) {
            y -= SCHRITT;
        }
        if (pbc_pressed_down() && y < PBC_HEIGHT - RADIUS) {
            y += SCHRITT;
        }
        if (pbc_was_pressed_a()) {
            farbe = pbc_rgb(pbc_random(80, 255), pbc_random(80, 255),
              pbc_random(80, 255));
        }

        // Zeichnen und warten
        pbc_clear(PBC_BLACK);
        pbc_fill_circle(x, y, RADIUS, farbe);
        pbc_show();
        sleep_ms(20);
    }
}
```

Fast jedes Spiel ist eine solche Endlosschleife. Ein Durchlauf hat immer dieselben Schritte:

1. **Lesen:** Welche Tasten sind gedrückt?
2. **Rechnen:** Wohin bewegt sich der Punkt? Ändert sich die Farbe?
3. **Zeichnen:** Das neue Bild malen und mit `pbc_show()` zeigen.
4. **Warten:** Eine kurze Pause bestimmt, wie schnell alles läuft.



- `x` und `y` (Zeilen 9 und 10) speichern, wo der Punkt gerade ist. Er startet ungefähr in der Mitte.
- Die Zeilen 14 bis 25 sind vier **einzelne** `if`, kein `else if`. So wird jede Richtung geprüft, auch wenn schon eine zutraf.
- Die zweite Hälfte jeder Bedingung prüft, ob am Rand noch Platz ist. `x > RADIUS` heißt: Der Kreis ragt links noch nicht aus dem Bild. Weil `&&` nur wahr ist, wenn **beide** Teile wahr sind, bleibt der Punkt am Rand stehen.
- Zeile 26 benutzt `pbc_was_pressed_a()`: Ein Druck soll genau **eine** neue Farbe ergeben. Mit `pbc_pressed_a()` würde der Punkt flackern, solange A unten ist. `pbc_random(80, 255)` sorgt dafür, dass keine Farbe zu dunkel wird.
- Bei jedem Durchlauf bewegt sich der Punkt um `SCHRITT` = 3 Pixel. Mit etwa 27 Durchläufen pro Sekunde sind das rund 80 Pixel pro Sekunde.

> Halte die Schritte auch in deinen eigenen Programmen getrennt. Wenn etwas nicht stimmt, weißt du sofort, wo du suchen musst: Bewegt sich der Punkt falsch, liegt es am Rechnen; sieht er falsch aus, am Zeichnen.

## 9.4 Warten auf eine Taste

Oft soll ein Programm warten, bis jemand bereit ist. Dafür gibt es `pbc_wait_for_button()`. Die Funktion wartet, bis **irgendeine** Taste gedrückt **und wieder losgelassen** wurde. Erst dann geht es weiter:

k09_start.c

```c
// Wartet auf eine beliebige Taste
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_text24("Bereit?", 73, 100, PBC_WHITE);
    pbc_text12("Druecke eine Taste", 48, 150, PBC_GRAY);
    pbc_show();

    pbc_wait_for_button();

    pbc_clear(PBC_BLUE);
    pbc_text24("Los geht's!", 46, 120, PBC_WHITE);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```






`pbc_wait_for_button()` holt den abgewarteten Druck gleich mit ab. Eine spätere Abfrage mit `pbc_was_pressed_…()` meldet ihn also nicht noch einmal.

Soll es eine **bestimmte** Taste sein, schreibst du die Warteschleife selbst:

```
while (!pbc_was_pressed_a()) {
    sleep_ms(10);
}
```

Lies das so: „Solange A **nicht** gedrückt wurde, warte 10 ms.“ Sobald A gedrückt wird, ist die Bedingung falsch, und das Programm geht weiter. Die kurze Pause hält die Schleife ruhig; einen Druck verpasst sie nicht, denn der Interrupt merkt ihn sich ja.

Mit `break` (Kapitel 8) wartest du auf eine von mehreren Tasten und merkst dir, welche es war:

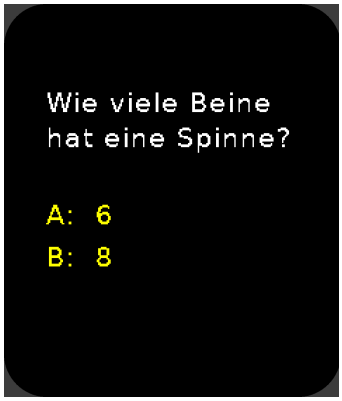
k09_quiz.c

```c
// Eine Quizfrage: Antwort mit A oder B
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_text18("Wie viele Beine", 30, 60, PBC_WHITE);
    pbc_text18("hat eine Spinne?", 30, 85, PBC_WHITE);
    pbc_text18("A:  6", 30, 140, PBC_YELLOW);
    pbc_text18("B:  8", 30, 170, PBC_YELLOW);
    pbc_show();

    bool richtig;
    while (true) {
        if (pbc_was_pressed_a()) {
            richtig = false;
            break;
        }
        if (pbc_was_pressed_b()) {
            richtig = true;
            break;
        }
        sleep_ms(10);
    }

    if (richtig) {
        pbc_clear(PBC_GREEN);
        pbc_text24("Richtig!", 68, 120, PBC_BLACK);
    } else {
        pbc_clear(PBC_RED);
        pbc_text24("Leider nein", 45, 120, PBC_WHITE);
    }
    pbc_show();
```



```
34      while (true) {
35          sleep_ms(1000);
36      }
37  }
```

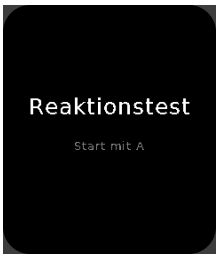
- Die Schleife ab Zeile 14 läuft so lange, bis A oder B gedrückt wurde. Dann speichert sie in `richtig`, ob die Antwort stimmt, und steigt mit `break` aus.
- Ab Zeile 26 wird die Antwort ausgewertet. Nach einem Druck auf B wird der Bildschirm grün und zeigt „Richtig!“.

**Ein alter Druck kann stören.** Die Notiz des Interrupts bleibt liegen, bis jemand sie abholt. Hat jemand A schon gedrückt, *bevor* dein Programm zur Warteschleife kommt, endet die Schleife sofort. Willst du nur Drücke zählen, die **ab jetzt** kommen, rufe die Funktion vorher einmal auf und wirf das Ergebnis weg:

```
pbc_was_pressed_a();          // alten Druck vergessen
while (!pbc_was_pressed_a()) {
    sleep_ms(10);
}
```

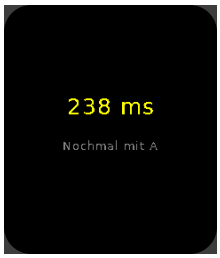
## Ausprobieren: Reaktionstest

Wie schnell bist du? Nach dem Start wird der Bildschirm rot. Nach einer zufälligen Zeit wird er grün, und du drückst so schnell wie möglich A. Wer schon bei Rot drückt, ist zu früh dran.













k09_reaktion.c

```
1  // Reaktionstest: Wie schnell drueckst du A, wenn der Bildschirm gruen wird?
2  #include "pbc.h"
3
4  int main(void) {
5      pbc_init();
6      pbc_clear(PBC_BLACK);
7      pbc_text24("Reaktionstest", 29, 100, PBC_WHITE);
8      pbc_text12("Start mit A", 80, 150, PBC_GRAY);
9      pbc_show();
```

```
11      while (true) {
12          // Auf A warten; einen alten Druck vorher vergessen
13          pbc_was_pressed_a();
14          while (!pbc_was_pressed_a()) {
15              sleep_ms(10);
16          }
17
18          // Rot: eine zufaellige Zeit warten. Wer jetzt drueckt, ist zu frueh.
19          pbc_clear(PBC_RED);
20          pbc_text24("Warte ...", 63, 120, PBC_WHITE);
21          pbc_show();
22          uint32_t gruen_ab = pbc_millis() + pbc_random(1500, 4000);
23          bool zu_frueh = false;
24          while (pbc_millis() < gruen_ab) {
25              if (pbc_was_pressed_a()) {
26                  zu_frueh = true;
27                  break;
28              }
29              sleep_ms(1);
30          }
```

- Zeile 13 vergisst einen alten Druck, Zeile 14 wartet auf A.
- Zeile 22 rechnet aus, **wann** der Bildschirm grün werden soll: jetzt plus 1,5 bis 4 Sekunden. `pbc_random` sorgt dafür, dass niemand den Moment vorhersagen kann.
- Die Schleife ab Zeile 24 wartet bis zu diesem Zeitpunkt. Sie fragt dabei ständig A ab. Kommt ein Druck, ist es ein Frühstart: Zeile 26 merkt ihn sich, `break` beendet das Warten vorzeitig.

k09_reaktion.c

```
31
32          // Gruen: Zeit messen bis zum Druck auf A
33          char text[20];
34          if (zu_frueh) {
35              snprintf(text, sizeof text, "Zu frueh!");
36          } else {
37              pbc_clear(PBC_GREEN);
38              pbc_text24("JETZT!", 77, 120, PBC_BLACK);
39              pbc_show();
40              uint32_t start = pbc_millis();
41              while (!pbc_was_pressed_a()) {
42                  sleep_ms(1);
43              }
44              int zeit = pbc_millis() - start;
45              printf("Reaktionszeit: %d ms\n", zeit);
46              snprintf(text, sizeof text, "%d ms", zeit);
47          }
48
49          // Ergebnis zeigen, in der Mitte
50          pbc_clear(PBC_BLACK);
```

```
51          int breite = pbc_text24_width(text);
52          pbc_text24(text, (PBC_WIDTH - breite) / 2, 100, PBC_YELLOW);
53          pbc_text12("Nochmal mit A", 66, 150, PBC_GRAY);
54          pbc_show();
55      }
56  }
```

- Ohne Frühstart wird der Bildschirm grün (Zeilen 37 bis 39). Erst **nach** `pbc_show()` merkt sich Zeile 40 die Startzeit, denn vorher war das Grün noch nicht zu sehen.
- Die Schleife ab Zeile 41 wartet auf A. Zeile 44 rechnet die vergangene Zeit aus: späterer Zeitpunkt minus früherer.
- Beide Fälle legen ihr Ergebnis in `text` ab. Die Zeilen 50 bis 54 zeigen es, mit `pbc_text24_width` in die Mitte gerückt (Kapitel 6).

Serial Monitor

```
Reaktionszeit: 238 ms
```

Die meisten Menschen brauchen zwischen 200 und 300 ms. Probiere aus, ob du mit der linken Hand anders abschneidest als mit der rechten.

**Zum Verändern:** Merke dir die beste Zeit in einer weiteren Variable und zeige sie unter dem Ergebnis an. Lass zwei Personen gegeneinander antreten: Wer zuerst drückt, A oder B, gewinnt.

## Übungsaufgaben

**Aufgabe 1 ⭐ Licht an, Licht aus** Lies das Programm, ohne es aufzuspielen. Was zeigt der Bildschirm, wenn du … (a) A kurz drückst, (b) danach A noch einmal kurz drückst, (c) danach B zwei Sekunden lang hältst und loslässt, (d) danach A zwei Sekunden lang hältst und loslässt? Prüfe danach auf dem PicoBoy.

k09_aufgabe1.c

```c
// Was passiert bei welchem Tastendruck?
#include "pbc.h"

int main(void) {
    pbc_init();
    bool licht = false;
    while (true) {
        if (pbc_was_pressed_a()) {
            licht = !licht;
        }
        if (licht || pbc_pressed_b()) {
            pbc_clear(PBC_WHITE);
```

```
        } else {
            pbc_clear(PBC_BLACK);
        }
        pbc_show();
        sleep_ms(20);
    }
}
```

**Aufgabe 2 ⭐ Zaubertafel** Ändere `k09_punkt.c` so, dass der Punkt eine Spur hinterlässt, mit der du malen kannst. Ein Druck auf B wischt die Tafel wieder leer. Tipp: Was passiert, wenn der Bildschirm nicht bei jedem Durchlauf gelöscht wird?

**Aufgabe 3 ⭐⭐ Finde die Fehler** Mit dem Joystick soll man eine Lautstärke von 0 bis 10 einstellen, ein Schritt pro Druck. Ist die Lautstärke schon 0 und man drückt nach unten, soll in der Konsole „Schon ganz leise!“ erscheinen. Das Programm baut ohne Warnung, macht aber zwei Dinge falsch. Welche? Verbessere es.

k09_aufgabe3.c

```c
// Lautstaerke von 0 bis 10 einstellen - mit zwei Fehlern
#include "pbc.h"

int main(void) {
    pbc_init();
    int stufe = 5;
    char text[20];
    while (true) {
        if (pbc_pressed_up() && stufe < 10) {
            stufe++;
        }
        if (pbc_was_pressed_down() && stufe > 0) {
            stufe--;
        }
        if (pbc_was_pressed_down() && stufe == 0) {
            printf("Schon ganz leise!\n");
        }

        pbc_clear(PBC_BLACK);
        snprintf(text, sizeof text, "Stufe %d", stufe);
        pbc_text24(text, 70, 100, PBC_WHITE);
        pbc_fill_rect(20, 150, stufe * 20, 20, PBC_GREEN);
        pbc_show();
        sleep_ms(20);
    }
}
```

**Aufgabe 4 ⭐⭐ Stoppuhr** Schreibe eine Stoppuhr: Ein Druck auf A startet sie, der nächste hält sie an, der übernächste startet sie wieder bei 0. Die Zeit steht in Sekunden mit einer Nachkommastelle in der Mitte des Bildschirms, etwa `12.3 s`. Tipp: Aus 12345 ms bekommst du mit `/` und `%` die 12 und die 3.

**Aufgabe 5 ⭐⭐⭐ Klick-Wettbewerb** Nach einem Druck auf B hast du 5 Sekunden Zeit, so oft wie möglich A zu drücken. Während der Runde zeigt der Bildschirm die Anzahl und die restlichen Sekunden. Danach erscheint wieder der Startbildschirm mit dem bisherigen Rekord. Wie viele Drücke pro Sekunde könnte die Bibliothek höchstens zählen, selbst bei sehr schnellen Fingern? Warum?

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- `pbc_pressed_…()` ist `true`, **solange** die Taste unten ist: für Bewegung und alles, was beim Halten passieren soll.
- `pbc_was_pressed_…()` ist genau **einmal pro Druck** `true`. Ein Interrupt merkt sich den Druck, auch wenn das Programm gerade beschäftigt ist. Die Abfrage holt ihn ab; pro Durchlauf nur einmal fragen.
- Die Bibliothek entprellt: Ein Druck zählt nur, wenn die Taste davor 50 ms ruhig war (auch das Federn beim Loslassen zählt nicht).
- Eine Spielschleife hat vier Schritte: lesen, rechnen, zeichnen, warten.
- `pbc_wait_for_button()` wartet auf irgendeine Taste. `while (!pbc_was_pressed_a()) { sleep_ms(10); }` wartet auf A. Vorher einmal `pbc_was_pressed_a();` vergisst einen alten Druck.

KAPITEL 10

# LEDs, RGB-LED und Töne

**Worum geht's?** Der PicoBoy kann mehr als Bilder zeigen: Er hat drei kleine Lämpchen, eine Lampe, die in jeder Farbe leuchtet, und einen Lautsprecher. Du schaltest Lichter, mischst Farben, spielst Töne und programmierst eine Melodie. Am Ende baust du eine Fußgängerampel mit Signalton.

Was in diesem Kapitel passiert, siehst und hörst du nur am PicoBoy selbst. Spiele die Programme deshalb auf und beobachte das Gerät genau.

## 10.1 Die drei Status-LEDs

Links neben dem Bildschirm sitzen drei **LEDs** (Leuchtdioden), von oben nach unten rot, gelb und grün. Jede schaltest du einzeln ein und aus:

| Befehl | Wirkung |
|---|---|
| `pbc_led_red(true)` | rote LED an |
| `pbc_led_red(false)` | rote LED aus |
| `pbc_led_yellow(…)`, `pbc_led_green(…)` | dasselbe für Gelb und Grün |
| `pbc_leds_off()` | alle drei aus |

Das folgende Programm schaltet die LEDs nacheinander ein und wieder aus:

k10_leds.c

```c
// Die drei Status-LEDs nacheinander ein- und ausschalten
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_led_red(true);
    sleep_ms(1000);
    pbc_led_yellow(true);
    sleep_ms(1000);
    pbc_led_green(true);
    sleep_ms(1000);

    pbc_led_yellow(false);
    sleep_ms(1000);
    pbc_leds_off();
    while (true) {
        sleep_ms(1000);
    }
}
```

- Zeile 6 schaltet die rote LED ein. Danach wartet das Programm eine Sekunde.
- In den Zeilen 8 und 10 kommen Gelb und Grün dazu. Eine LED bleibt an, bis das Programm sie ausschaltet. Nach Zeile 10 leuchten also alle drei.
- Zeile 13 schaltet nur die gelbe LED aus, Zeile 15 alle, die noch an sind.

> **Licht bleibt an.** Was dein Programm einschaltet, leuchtet weiter, auch wenn das Programm danach nur noch in seiner Endlosschleife wartet. Erst ein neues Programm (oder ein Druck auf RES) schaltet die LEDs wieder aus, denn `pbc_init()` beginnt immer mit allen Lichtern aus.

**Tasten schalten Licht.** In den Klammern steht ein `bool`. Das kann auch das Ergebnis einer Tastenabfrage oder einer Bedingung sein:

k10_tasten_leds.c

```c
// Tasten schalten LEDs
#include "pbc.h"

int main(void) {
    pbc_init();
    while (true) {
        pbc_led_red(pbc_pressed_a());
        pbc_led_green(pbc_pressed_b());
        pbc_led_yellow(pbc_pressed_up() || pbc_pressed_down());
        sleep_ms(10);
    }
}
```

`pbc_pressed_a()` liefert `true`, solange A unten ist, sonst `false`. Zeile 7 gibt diesen Wert direkt an `pbc_led_red` weiter: Die rote LED leuchtet genau so lange, wie du A drückst. In Zeile 9 entscheidet ein `||`: Gelb leuchtet, wenn der Joystick nach oben **oder** nach unten gedrückt ist.

**Blinken.** Eine blinkende LED wechselt immer wieder zwischen an und aus. Dafür merkst du dir den Zustand in einer `bool`-Variable und drehst ihn mit `!` (nicht) um:

k10_blinken.c

```c
// Die gruene LED blinkt; A schaltet das Blinken ein und aus
#include "pbc.h"

int main(void) {
    pbc_init();
    bool blinken = true;
    bool an = false;
    while (true) {
        if (pbc_was_pressed_a()) {
            blinken = !blinken;
        }
        if (blinken) {
            an = !an;
```

```
14          } else {
15              an = false;
16          }
17          pbc_led_green(an);
18          sleep_ms(250);
19      }
20  }
```

- `an = !an` in Zeile 13 macht aus `true` ein `false` und umgekehrt. Bei jedem Durchlauf, also alle 250 ms, wechselt die LED. Sie blinkt damit zweimal pro Sekunde.
- Ein Druck auf A schaltet mit demselben Trick das Blinken selbst ein und aus (Zeile 10). Ist es aus, bleibt die LED dunkel (Zeile 15).

## 10.2 Die RGB-LED: Farben mischen

Im Pluszeichen des Schriftzugs „PBC+“ sitzt eine besondere LED, die **RGB-LED**. Sie enthält drei winzige Lampen in Rot, Grün und Blau. Für jede gibst du an, wie hell sie leuchten soll, von 0 (aus) bis 255 (ganz hell):

```
pbc_rgb_led(rot, gruen, blau);
```

Die drei Farben mischen sich zu einer neuen. Das funktioniert wie auf einem Bildschirm, nicht wie beim Malen mit dem Pinsel: Rot und Grün ergeben **Gelb**, alle drei zusammen ergeben **Weiß**.

| `rot, gruen, blau` | Farbe | `rot, gruen, blau` | Farbe |
|---|---|---|---|
| `255, 0, 0` | Rot | `255, 255, 0` | Gelb |
| `0, 255, 0` | Grün | `255, 0, 255` | Magenta |
| `0, 0, 255` | Blau | `0, 255, 255` | Cyan |
| `255, 255, 255` | Weiß | `255, 100, 0` | Orange |

k10_rgb.c

```
1  // Die RGB-LED in verschiedenen Farben
2  #include "pbc.h"
3
4  int main(void) {
5      pbc_init();
6      pbc_rgb_led(255, 0, 0);       // Rot
7      sleep_ms(1000);
8      pbc_rgb_led(0, 255, 0);       // Gruen
9      sleep_ms(1000);
10     pbc_rgb_led(0, 0, 255);       // Blau
11     sleep_ms(1000);
12     pbc_rgb_led(255, 255, 0);     // Rot + Gruen = Gelb
13     sleep_ms(1000);
14     pbc_rgb_led(30, 30, 30);      // alle drei schwach: Weiss
```

```
15      sleep_ms(1000);
16      pbc_rgb_led_off();
17      while (true) {
18          sleep_ms(1000);
19      }
20  }
```

Jeder Aufruf von `pbc_rgb_led` ersetzt die vorige Farbe. Zeile 14 mischt alle drei Farben, aber nur schwach (30 von 255): Das ergibt ein gedämpftes Weiß. `pbc_rgb_led_off()` schaltet die RGB-LED aus; das ist dasselbe wie `pbc_rgb_led(0, 0, 0)`. Werte unter 0 behandelt die Bibliothek wie 0, Werte über 255 wie 255.

> Die Zahlen für `pbc_rgb_led` sind dieselben wie für `pbc_rgb(r, g, b)` aus Kapitel 6. Damit kannst du die LED und den Bildschirm in derselben Farbe leuchten lassen. Ganz gleich sehen sie trotzdem nicht aus: Eine LED ist eine Lampe, der Bildschirm ein beleuchtetes Bild.

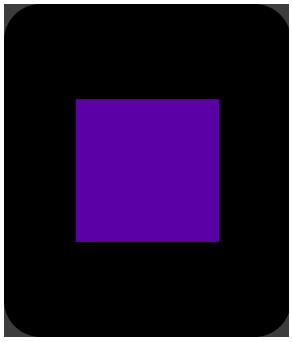
**Ein Farbverlauf.** Die Schleife zählt `rot` in Fünferschritten von 0 bis 255. Gleichzeitig wird `blau` immer kleiner. So wandert die Farbe langsam von Blau über Lila nach Rot, auf der LED und im Quadrat auf dem Bildschirm:

k10_farbverlauf.c

```c
// Farbverlauf von Blau nach Rot: RGB-LED und Bildschirm
#include "pbc.h"

int main(void) {
    pbc_init();
    for (int rot = 0; rot <= 255; rot += 5) {
        int blau = 255 - rot;
        pbc_rgb_led(rot, 0, blau);
        pbc_clear(PBC_BLACK);
        pbc_fill_rect(60, 80, 120, 120, pbc_rgb(rot, 0, blau));
        pbc_show();
        sleep_ms(50);
    }
    pbc_rgb_led_off();
    while (true) {
        sleep_ms(1000);
    }
}
```

- Zeile 6 läuft über 0, 5, 10, … bis 255.
- Zeile 7 rechnet das passende Blau aus: Ist `rot` 0, ist `blau` 255 und umgekehrt. Die Summe ist immer 255.
- Zeile 8 setzt die LED, Zeile 10 malt ein Quadrat in genau dieser Farbe.



Im Bild ist der Verlauf ungefähr in der Mitte, bei Lila.

## 10.3 Töne und Melodien

Ein Ton entsteht, wenn die Membran eines Lautsprechers sehr schnell hin und her schwingt. Wie oft sie das pro Sekunde tut, heißt **Frequenz** und wird in **Hertz** (Hz) gemessen. Je höher die Frequenz, desto höher der Ton. Der Ton a, nach dem Musiker ihre Instrumente stimmen, hat 440 Hz. Menschen hören ungefähr von 20 Hz bis 16 000 Hz.

| Befehl | Wirkung |
|---|---|
| `pbc_tone(440, 500)` | Ton mit 440 Hz, hört nach 500 ms von selbst auf |
| `pbc_tone(440, 0)` | **Dauerton** mit 440 Hz, klingt, bis `pbc_tone_off()` kommt |
| `pbc_tone_off()` | Ton sofort ausschalten |

k10_ton.c

```
 1 // Ein Dauerton und ein Ton mit fester Dauer
 2 #include "pbc.h"
 3
 4 int main(void) {
 5     pbc_init();
 6     // Dauerton: klingt, bis pbc_tone_off() kommt
 7     pbc_tone(440, 0);
 8     sleep_ms(1000);
 9     pbc_tone_off();
10
11     sleep_ms(500);                // Stille
12
13     // Ton mit Dauer: hoert nach 500 ms von selbst auf
14     pbc_tone(880, 500);
15     sleep_ms(500);
16     while (true) {
17         sleep_ms(1000);
18     }
19 }
```

- Zeile 7 startet einen Dauerton. Zeile 8 wartet eine Sekunde, so lange klingt der Ton. Zeile 9 schaltet ihn aus.
- Zeile 11 macht eine halbe Sekunde Pause, in der nichts zu hören ist.
- Zeile 14 spielt einen Ton, der doppelt so hoch ist (880 Hz) und nach 500 ms von selbst aufhört.

**`pbc_tone` wartet nicht.** Der Befehl schaltet den Ton nur ein, und das Programm läuft sofort weiter. Das ist praktisch: In einem Spiel kann ein Ton klingen, während sich die Figur weiterbewegt. Für Töne nacheinander musst du aber selbst warten. Was passiert sonst?

k10_ohne_pause.c

```c
// Fehler: drei Toene ohne Pause - zu hoeren ist nur der letzte
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_tone(262, 300);
    pbc_tone(330, 300);
    pbc_tone(392, 300);
    while (true) {
        sleep_ms(1000);
    }
}
```

Du erwartest drei Töne, hörst aber nur den letzten. Die Zeilen 6 bis 8 laufen in einem winzigen Augenblick ab, und jeder neue Ton ersetzt den vorigen sofort. Der Lautsprecher spielt immer nur **einen** Ton. Hinter jedes `pbc_tone(f, ms)` gehört deshalb ein `sleep_ms(…)`.

> **Ein Dauerton hört nicht von allein auf**, auch nicht, wenn das Programm in seiner Endlosschleife angekommen ist. Schreibe zu jedem `pbc_tone(f, 0)` auch ein `pbc_tone_off()`.

**Eine Melodie** ist eine Folge von Tönen mit bestimmter Höhe und Länge. Diese Frequenzen gehören zu den Tönen der C-Dur-Tonleiter:

| Ton | c | d | e | f | g | a | h | c’ |
|---|---|---|---|---|---|---|---|---|
| Hz | 262 | 294 | 330 | 349 | 392 | 440 | 494 | 523 |

Damit spielst du den Anfang von „Alle meine Entchen“:

k10_entchen.c

```c
// Der Anfang von "Alle meine Entchen"
#include "pbc.h"

#define TON_C 262
#define TON_D 294
#define TON_E 330
#define TON_F 349
#define TON_G 392

#define VIERTEL 300
#define HALBE 600
#define LUECKE 50

int main(void) {
    pbc_init();
    pbc_tone(TON_C, VIERTEL - LUECKE);
    sleep_ms(VIERTEL);
    pbc_tone(TON_D, VIERTEL - LUECKE);
```

```
    sleep_ms(VIERTEL);
    pbc_tone(TON_E, VIERTEL - LUECKE);
    sleep_ms(VIERTEL);
    pbc_tone(TON_F, VIERTEL - LUECKE);
    sleep_ms(VIERTEL);
    pbc_tone(TON_G, HALBE - LUECKE);
    sleep_ms(HALBE);
    pbc_tone(TON_G, HALBE - LUECKE);
    sleep_ms(HALBE);
    while (true) {
        sleep_ms(1000);
    }
}
```

- Die Zeilen 4 bis 8 geben den Frequenzen Namen. `pbc_tone(TON_E, …)` liest sich leichter als `pbc_tone(330, …)`.
- `VIERTEL` und `HALBE` legen fest, wie lang eine Note dauert. Änderst du nur die Zeilen 10 und 11, spielt das ganze Lied schneller oder langsamer.
- Jede Note besteht aus zwei Zeilen: Ton starten, dann warten. Der Ton ist um `LUECKE` = 50 ms **kürzer** als die Wartezeit. Diese kleine Stille trennt die Noten. Ohne sie würden die beiden g am Ende zu einem langen Ton verschmelzen.

**Töne ausrechnen.** Von einem Ton zum nächsten Halbton steigt die Frequenz immer um denselben Faktor, ungefähr 1,0595. Nach zwölf Halbtönen hat sie sich genau verdoppelt: Das ist eine **Oktave**.

k10_tonleiter.c

```
// Zwoelf Halbtoene aufwaerts: jeder ist 1,0595-mal so hoch wie der vorige
#include "pbc.h"

int main(void) {
    pbc_init();
    float frequenz = 440.0f;
    for (int schritt = 0; schritt <= 12; schritt++) {
        int hz = (int)frequenz;
        printf("Schritt %2d: %d Hz\n", schritt, hz);
        pbc_tone(hz, 250);
        sleep_ms(300);
        frequenz *= 1.0595f;
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Schritt  0: 440 Hz
Schritt  1: 466 Hz
Schritt  2: 493 Hz
Schritt  3: 523 Hz
Schritt  4: 554 Hz
```

```
Schritt  5: 587 Hz
Schritt  6: 622 Hz
Schritt  7: 659 Hz
Schritt  8: 698 Hz
Schritt  9: 740 Hz
Schritt 10: 784 Hz
Schritt 11: 830 Hz
Schritt 12: 880 Hz
```

`frequenz` ist eine Kommazahl, denn mit ganzen Zahlen ginge das Malnehmen mit 1,0595 nicht. `pbc_tone` erwartet aber eine ganze Zahl. Zeile 8 wandelt deshalb um: `(int)frequenz` schneidet die Nachkommastellen ab (Kapitel 5). Aus 493,88 wird so 493. Würdest du `frequenz` direkt übergeben, wandelt C ebenfalls um, ohne Warnung; mit `(int)` sieht man es im Programm.

**Geräusche.** Ändert sich die Frequenz schnell und gleichmäßig, klingt das wie eine Sirene:

k10_sirene.c

```c
// Eine Sirene: die Tonhoehe steigt und faellt
#include "pbc.h"

int main(void) {
    pbc_init();
    for (int runde = 0; runde < 3; runde++) {
        for (int hz = 400; hz <= 1200; hz += 10) {
            pbc_tone(hz, 0);
            sleep_ms(5);
        }
        for (int hz = 1200; hz >= 400; hz -= 10) {
            pbc_tone(hz, 0);
            sleep_ms(5);
        }
    }
    pbc_tone_off();
    while (true) {
        sleep_ms(1000);
    }
}
```

Die erste innere Schleife lässt den Ton in Zehnerschritten von 400 auf 1200 Hz steigen, die zweite wieder fallen. Jeder Schritt dauert nur 5 ms. Weil jeder neue Ton den alten ersetzt, hörst du ein gleitendes Heulen. Am Ende schaltet Zeile 16 den Dauerton aus.

> Die Bibliothek spielt Frequenzen von 20 bis 20 000 Hz; kleinere und größere Werte setzt sie auf diese Grenzen. Eine Frequenz von 0 oder weniger schaltet den Ton aus. Gut klingen auf dem kleinen Lautsprecher vor allem Töne zwischen etwa 200 und 2000 Hz.

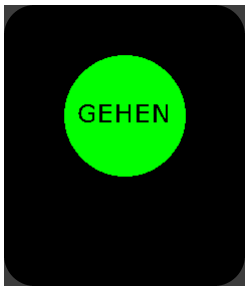
## Ausprobieren: Ampel mit Signalton

Eine Fußgängerampel: Die drei Status-LEDs sind die Ampel für die Autos. Die RGB-LED und der Bildschirm zeigen das Signal für die Fußgänger. Ein Druck auf A fordert Grün an. Solange die Fußgänger gehen dürfen, tickt ein Signalton, wie an vielen echten Ampeln für blinde Menschen.










k10_ampel.c

```
1  // Fussgaengerampel: Die Status-LEDs sind die Autoampel, die RGB-LED und
2  // der Bildschirm zeigen das Signal fuer Fussgaenger.
3  #include "pbc.h"
4
5  #define HELL 60                  // Helligkeit der RGB-LED
6
7  int main(void) {
8      pbc_init();
9      while (true) {
10         // Autos Gruen, Fussgaenger Rot
11         pbc_leds_off();
12         pbc_led_green(true);
13         pbc_rgb_led(HELL, 0, 0);
14         pbc_clear(PBC_BLACK);
15         pbc_fill_circle(120, 110, 60, PBC_RED);
16         pbc_text24("STEHEN", 67, 94, PBC_WHITE);
17         pbc_text18("Bitte A druecken", 33, 200, PBC_WHITE);
18         pbc_show();
19
20         // Auf A warten, dann den Druck bestaetigen
21         pbc_was_pressed_a();
22         while (!pbc_was_pressed_a()) {
23             sleep_ms(10);
24         }
25         pbc_tone(1000, 100);
26         pbc_fill_rect(0, 195, PBC_WIDTH, 30, PBC_BLACK);
27         pbc_text18("Signal kommt", 49, 200, PBC_YELLOW);
28         pbc_show();
29         sleep_ms(2000);
```

- Am Anfang jedes Durchlaufs haben die Autos Grün (Zeilen 11 und 12). `pbc_leds_off()` löscht vorher alles, was aus dem letzten Durchlauf noch leuchtet.
- `HELL` (Zeile 5) bestimmt die Helligkeit der RGB-LED. 60 von 255 reicht gut, ganz hell blendet sie.

- Die Zeilen 21 bis 24 warten auf A, wie in Kapitel 9. Der kurze Ton in Zeile 25 bestätigt den Druck.
- Zeile 26 übermalt nur den unteren Text mit einem schwarzen Rechteck. Der rote Kreis bleibt im Bildpuffer stehen.

k10_ampel.c

```
30
31          // Autos: Gelb, dann Rot
32          pbc_led_green(false);
33          pbc_led_yellow(true);
34          sleep_ms(2000);
35          pbc_led_yellow(false);
36          pbc_led_red(true);
37          sleep_ms(1000);
38
39          // Fussgaenger Gruen: 10 Sekunden, es tickt zweimal pro Sekunde
40          pbc_rgb_led(0, HELL, 0);
41          pbc_clear(PBC_BLACK);
42          pbc_fill_circle(120, 110, 60, PBC_GREEN);
43          pbc_text24("GEHEN", 73, 94, PBC_BLACK);
44          pbc_show();
45          for (int i = 0; i < 20; i++) {
46              pbc_tone(800, 30);
47              sleep_ms(500);
48          }
49
50          // Fussgaenger Rot, Autos Rot-Gelb (Gruen kommt oben in der Schleife)
51          pbc_rgb_led(HELL, 0, 0);
52          pbc_clear(PBC_BLACK);
53          pbc_fill_circle(120, 110, 60, PBC_RED);
54          pbc_text24("STEHEN", 67, 94, PBC_WHITE);
55          pbc_show();
56          sleep_ms(2000);
57          pbc_led_yellow(true);
58          sleep_ms(1000);
59      }
60  }
```

- Die Autoampel schaltet über Gelb auf Rot (Zeilen 32 bis 37). Erst eine Sekunde danach bekommen die Fußgänger Grün.
- Die Schleife in Zeile 45 tickt 20-mal im Abstand von 500 ms, also zehn Sekunden lang. Jeder Ton dauert nur 30 ms, den Rest der Zeit ist es still.
- Danach haben die Fußgänger wieder Rot. Die Autos bekommen Rot-Gelb (Zeile 57: Rot ist noch an, Gelb kommt dazu), und oben in der Schleife geht es mit Grün weiter.

**Zum Verändern:** Lass das grüne Fußgängersignal in den letzten drei Sekunden blinken. Spiele einen tieferen, langsameren Ton, solange die Fußgänger warten müssen.

## Übungsaufgaben

**Aufgabe 1** ⭐ **Welche Lichter brennen?** Lies das Programm, ohne es aufzuspielen. Welche LEDs leuchten in der ersten halben Sekunde? Welche leuchten am Ende, wenn du keine Taste drückst, und in welcher Farbe leuchtet dann die RGB-LED? Prüfe danach auf dem PicoBoy.

k10_aufgabe1.c

```c
// Welche Lichter brennen?
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_led_red(true);
    pbc_led_green(true);
    sleep_ms(500);

    pbc_led_red(false);
    pbc_led_yellow(5 > 3);
    pbc_rgb_led(0, 0, 255);
    pbc_rgb_led(255, 0, 0);
    sleep_ms(500);
    pbc_led_green(pbc_pressed_a());
    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 2** ⭐ **Finde die Fehler** Das Programm soll ein Signal spielen: kurz, kurz, lang, dabei leuchtet die rote LED. Danach folgt ein Schlusston von einer halben Sekunde. Es gibt keine Fehlermeldung, trotzdem stimmt einiges nicht. Was hörst und siehst du? Verbessere das Programm.

k10_aufgabe2.c

```c
// Signal: kurz, kurz, lang, dann ein Schlusston - mit Fehlern
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_led_red(true);
    pbc_tone(880, 150);
    pbc_tone(880, 150);
    pbc_tone(880, 600);
    pbc_led_red(false);

    sleep_ms(500);
    pbc_tone(440, 0);             // Schlusston, eine halbe Sekunde
    sleep_ms(500);
```

```
    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 3 ⭐⭐ Blaulicht** Schreibe ein Programm für ein Einsatzfahrzeug: Die RGB-LED wechselt fünfmal zwischen Blau und Rot, je eine halbe Sekunde. Bei Blau klingt ein Ton mit 440 Hz, bei Rot einer mit 587 Hz. Auf dem Bildschirm steht „Einsatz!“. Am Ende sind Ton und Licht aus.

**Aufgabe 4 ⭐⭐ Die Entchen schwimmen weiter** Ergänze `k10_entchen.c` um „schwimmen auf dem See“: viermal der Ton a als Viertelnote, dann g, so lang wie zwei halbe Noten. Tipp: Für die vier gleichen Töne passt eine `for`-Schleife.

**Aufgabe 5 ⭐⭐⭐ Ein Metronom** Ein Metronom gibt beim Musizieren den Takt vor. Schreibe ein Programm, das gleichmäßig klickt (kurzer Ton mit 1000 Hz) und bei jedem Klick kurz die rote LED aufleuchten lässt. Das Tempo steht in Schlägen pro Minute auf dem Bildschirm (Start: 100). Joystick nach oben macht es um 10 schneller, nach unten um 10 langsamer. Wie viele Millisekunden liegen bei 100 Schlägen pro Minute zwischen zwei Klicks? Tipp: Mit `sleep_ms` für die ganze Wartezeit würde das Programm in dieser Zeit nicht auf den Joystick reagieren. Miss die Zeit lieber mit `pbc_millis()`.

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- `pbc_led_red(true)` / `pbc_led_red(false)` schaltet eine Status-LED an und aus, ebenso `pbc_led_yellow` und `pbc_led_green`; `pbc_leds_off()` alle. In den Klammern darf jeder `bool` stehen, auch `pbc_pressed_a()`.
- `pbc_rgb_led(r, g, b)` mischt eine Farbe aus Rot, Grün und Blau (je 0–255), `pbc_rgb_led_off()` schaltet die RGB-LED aus.
- Lichter bleiben an, bis das Programm sie ausschaltet.
- `pbc_tone(f, ms)` spielt `f` Hertz für `ms` Millisekunden; `pbc_tone(f, 0)` ist ein Dauerton bis `pbc_tone_off()`.
- `pbc_tone` wartet **nicht**. Für eine Melodie nach jedem Ton `sleep_ms(…)`; ein neuer Ton ersetzt den alten.
- Höhere Frequenz = höherer Ton, doppelte Frequenz = eine Oktave höher.

KAPITEL 11

# Funktionen

**Worum geht's?** Bisher hast du Funktionen nur aufgerufen: `pbc_clear`, `pbc_text24`, `printf`. Jetzt schreibst du eigene. Damit gibst du einem Stück Programm einen Namen und benutzt es, so oft du willst. Du lernst Parameter und Rückgabewerte kennen, erfährst, warum C eine Funktion kennen muss, bevor sie benutzt wird, und wo Variablen gelten. Am Ende steht ein Zeichenbaukasten, mit dem eine ganze Landschaft entsteht.

## 11.1 Eigene Funktionen schreiben, `void`

Eine **Funktion** ist ein Stück Programm mit einem Namen. Du schreibst es einmal hin und rufst es danach überall auf, wo du es brauchst:

k11_trennlinie.c

```c
// Eine eigene Funktion ohne Parameter
#include "pbc.h"

void trennlinie(void) {
    printf("--------------------\n");
}

int main(void) {
    pbc_init();
    trennlinie();
    printf("Wetterbericht\n");
    trennlinie();
    printf("Sonnig, 24 Grad\n");
    printf("Wind aus Westen\n");
    trennlinie();
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
--------------------
Wetterbericht
--------------------
Sonnig, 24 Grad
Wind aus Westen
--------------------
```

Die Zeilen 4 bis 6 sind die **Definition** der Funktion `trennlinie`:

- **Zeile 4** ist der **Funktionskopf**. `void` vor dem Namen heißt „leer“: Die Funktion liefert kein Ergebnis zurück. `trennlinie` ist der Name, den du dir aussuchst, nach denselben Regeln wie bei Variablen. `(void)` in den Klammern heißt: Die Funktion braucht keine Angaben.

- **Zeile 5** ist der **Rumpf** zwischen `{` und `}`: die Anweisungen, die bei jedem Aufruf ausgeführt werden.

In `main` wird `trennlinie()` dreimal **aufgerufen** (Zeilen 10, 12 und 15), genau wie `pbc_init()`. Die Klammern sind nötig, auch wenn sie leer bleiben. Willst du später eine andere Linie, etwa aus Sternchen, änderst du nur Zeile 5, und alle drei Linien ändern sich mit.

**Was beim Aufruf passiert.** Bei einem Aufruf springt das Programm in die Funktion, führt ihren Rumpf aus und kehrt danach genau an die Stelle zurück, von der es kam. Eine Funktion darf selbst wieder Funktionen aufrufen:

k11_ablauf.c

```c
// In welcher Reihenfolge laufen die Anweisungen?
#include "pbc.h"

void zweite(void) {
    printf("  3. in zweite\n");
}

void erste(void) {
    printf(" 2. Anfang von erste\n");
    zweite();
    printf(" 4. Ende von erste\n");
}

int main(void) {
    pbc_init();
    printf("1. Anfang von main\n");
    erste();
    printf("5. Ende von main\n");
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
1. Anfang von main
 2. Anfang von erste
  3. in zweite
 4. Ende von erste
5. Ende von main
```

Die Nummern zeigen die Reihenfolge: `main` ruft `erste` auf, `erste` ruft `zweite` auf. Nach dem Ende von `zweite` geht es in `erste` weiter, nach dem Ende von `erste` wieder in `main`. Die Einrückung in der Ausgabe kommt nur von den Leerzeichen in den Texten.

> Gib Funktionen Namen, die sagen, was sie **tun**: `zeichne_haus`, `spiele_melodie`, `zeige_punkte`. Eine gute Funktion erledigt genau eine Aufgabe. Wird eine Funktion länger als eine Bildschirmseite, lohnt es sich meist, sie in mehrere kleinere aufzuteilen.

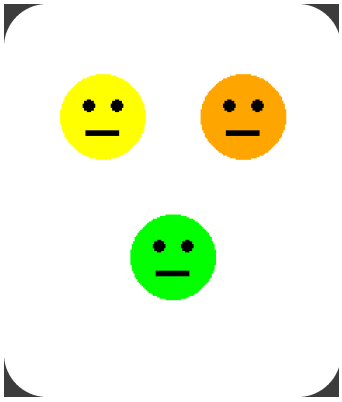
## 11.2 Parameter und Rückgabewert

**Parameter.** Viele Funktionen brauchen Angaben, um ihre Arbeit zu tun. `pbc_fill_circle` muss wissen, wo und wie groß und in welcher Farbe. Eine eigene Funktion bekommt solche Angaben über **Parameter**: Variablen, die im Funktionskopf in den Klammern stehen, jeweils mit Typ und Name.

k11_gesicht.c

```
 1 // Eine Funktion mit Parametern: ein Gesicht an beliebiger Stelle
 2 #include "pbc.h"
 3
 4 void zeichne_gesicht(int x, int y, uint16_t farbe) {
 5     pbc_fill_circle(x, y, 30, farbe);
 6     pbc_fill_circle(x - 10, y - 8, 4, PBC_BLACK);    // Augen
 7     pbc_fill_circle(x + 10, y - 8, 4, PBC_BLACK);
 8     pbc_fill_rect(x - 12, y + 10, 24, 4, PBC_BLACK); // Mund
 9 }
10
11 int main(void) {
12     pbc_init();
13     pbc_clear(PBC_WHITE);
14     zeichne_gesicht(70, 80, PBC_YELLOW);
15     zeichne_gesicht(170, 80, PBC_ORANGE);
16     zeichne_gesicht(120, 180, PBC_GREEN);
17     pbc_show();
18     while (true) {
19         sleep_ms(1000);
20     }
21 }
```

- `zeichne_gesicht` hat drei Parameter: `x` und `y` vom Typ `int` und `farbe` vom Typ `uint16_t`, so wie die Farben in `pbc.h`.
- Beim Aufruf in Zeile 14 stehen in den Klammern die **Argumente** `70, 80, PBC_YELLOW`. Sie werden der Reihe nach in die Parameter kopiert: `x` bekommt 70, `y` bekommt 80, `farbe` bekommt `PBC_YELLOW`.
- Im Rumpf (Zeilen 5 bis 8) rechnet alles mit `x` und `y`. So sitzen Augen und Mund immer an derselben Stelle **relativ zum Mittelpunkt**, egal wo das Gesicht gezeichnet wird.



Ein Aufruf, drei Gesichter: Das ist der große Vorteil von Funktionen mit Parametern.

**Parameter sind Kopien.** Eine Funktion bekommt nur die **Werte** der Argumente, nicht die Variablen selbst. Ändert sie einen Parameter, ändert sie nur ihre Kopie:

k11_kopie.c

```c
// Ein Parameter ist eine Kopie des Arguments
#include "pbc.h"

void verdopple(int zahl) {
    zahl = zahl * 2;
    printf("in verdopple: zahl = %d\n", zahl);
}

int main(void) {
    pbc_init();
    int zahl = 21;
    verdopple(zahl);
    printf("in main:      zahl = %d\n", zahl);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
in verdopple: zahl = 42
in main:      zahl = 21
```

Der Parameter `zahl` in `verdopple` und die Variable `zahl` in `main` heißen zwar gleich, sind aber zwei verschiedene Variablen. Zeile 5 verdoppelt nur die Kopie; in `main` bleibt es bei 21. Wie eine Funktion trotzdem eine Variable des Aufrufers ändern kann, zeigt Kapitel 19.

**Rückgabewert.** Eine Funktion kann ein Ergebnis an den Aufrufer zurückgeben, den **Rückgabewert**. Sein Typ steht im Funktionskopf vor dem Namen, anstelle von `void`. Mit `return` gibst du den Wert zurück:

k11_rueckgabe.c

```c
// Funktionen mit Rueckgabewert
#include "pbc.h"

int quadrat(int n) {
    return n * n;
}

float nach_fahrenheit(float celsius) {
    return celsius * 9.0f / 5.0f + 32.0f;
}

bool ist_gerade(int n) {
    return n % 2 == 0;
}
```

```
16 int main(void) {
17     pbc_init();
18     printf("7 * 7 = %d\n", quadrat(7));
19     int summe = quadrat(3) + quadrat(4);
20     printf("3*3 + 4*4 = %d\n", summe);
21     for (int c = -10; c <= 30; c += 20) {
22         printf("%d Grad C = %.1f Grad F\n", c, nach_fahrenheit(c));
23     }
24     if (ist_gerade(summe)) {
25         printf("%d ist gerade\n", summe);
26     } else {
27         printf("%d ist ungerade\n", summe);
28     }
29     while (true) {
30         sleep_ms(1000);
31     }
32 }
```

Serial Monitor

```
7 * 7 = 49
3*3 + 4*4 = 25
-10 Grad C = 14.0 Grad F
10 Grad C = 50.0 Grad F
30 Grad C = 86.0 Grad F
25 ist ungerade
```

- `int quadrat(int n)` bekommt eine ganze Zahl und liefert eine ganze Zahl zurück. Ein Aufruf wie `quadrat(7)` steht im Programm dann für ihr Ergebnis, also 49. Du kannst ihn überall dort hinschreiben, wo ein Wert stehen darf: in `printf` (Zeile 18) oder in einer Rechnung (Zeile 19).
- `nach_fahrenheit` rechnet mit Kommazahlen. In Zeile 22 wird ihr eine ganze Zahl `c` übergeben; C wandelt sie für den Parameter selbst in `float` um.
- `ist_gerade` liefert einen `bool`. `n % 2 == 0` ist wahr, wenn beim Teilen durch 2 kein Rest bleibt. So eine Funktion passt direkt in ein `if` (Zeile 24) und liest sich fast wie ein Satz.

`return` beendet die Funktion **sofort**. Was im Rumpf danach käme, wird nicht mehr ausgeführt. Auch eine `void`-Funktion darf mit `return;` (ohne Wert) vorzeitig enden.

> Jetzt verstehst du `int main(void)`: `main` ist eine Funktion ohne Parameter, die eine ganze Zahl zurückgeben würde. Auf einem PC meldet ein Programm damit dem Betriebssystem, ob alles geklappt hat. Auf dem PicoBoy endet `main` nie, die Zahl spielt also keine Rolle.

> **Typische Fehler mit Funktionen**
>
> Eine Funktion mit Rückgabetyp muss auf **jedem** Weg ein `return` erreichen. Hier fehlt es für

den Fall, dass `a` und `b` gleich sind:

k11_fehler_return.c

```c
// Fehler: Die Funktion gibt nicht in jedem Fall etwas zurueck
#include "pbc.h"

int groesser(int a, int b) {
    if (a > b) {
        return a;
    } else if (b > a) {
        return b;
    }
}

int main(void) {
    pbc_init();
    printf("%d\n", groesser(3, 3));
    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
main.c: In function 'groesser':
main.c:10:1: warning: control reaches end of non-void function [-Wreturn-type]
   10 | }
      | ^
```

*Control reaches end of non-void function* heißt: Das Programm kann das Ende einer Funktion erreichen, die kein `void` ist, ohne dass ein Wert zurückgegeben wird. Es ist nur eine Warnung; das Programm würde aufgespielt und für `groesser(3, 3)` irgendeine Zahl ausgeben. Die Lösung: am Ende ein `return a;` für den übrigen Fall.

Die Zahl der Argumente muss zur Zahl der Parameter passen:

k11_fehler_argumente.c

```c
// Fehler: ein Argument zu wenig
#include "pbc.h"

int quadrat(int n) {
    return n * n;
}

void zeichne_punkt(int x, int y, uint16_t farbe) {
    pbc_fill_circle(x, y, 5, farbe);
}

int main(void) {
    pbc_init();
```

```
    zeichne_punkt(120, 140);
    printf("%d\n", quadrat(2, 3));
    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:14:5: error: too few arguments to function 'zeichne_punkt'
   14 |     zeichne_punkt(120, 140);
      |     ^~~~~~~~~~~~~
main.c:8:6: note: declared here
    8 | void zeichne_punkt(int x, int y, uint16_t farbe) {
      |      ^~~~~~~~~~~~~
main.c:15:20: error: too many arguments to function 'quadrat'
   15 |     printf("%d\n", quadrat(2, 3));
      |                    ^~~~~~~
main.c:4:5: note: declared here
    4 | int quadrat(int n) {
      |     ^~~~~~~
```

*Too few arguments* heißt zu wenige, *too many arguments* zu viele. Die Zeile mit `note:` zeigt, wo die Funktion definiert ist, damit du nachsehen kannst, was sie erwartet.

## 11.3 Deklaration vor Gebrauch: Prototypen

Der Compiler liest dein Programm von oben nach unten, genau einmal. Ruft `main` eine Funktion auf, die erst **weiter unten** steht, kennt er sie an dieser Stelle noch nicht:

k11_fehler_reihenfolge.c

```c
// Fehler: Die Funktion steht erst hinter main
#include "pbc.h"

int main(void) {
    pbc_init();
    printf("%d\n", wuerfeln());
    while (true) {
        sleep_ms(1000);
    }
}

int wuerfeln(void) {
    return pbc_random(1, 6);
}
```

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:6:20: error: implicit declaration of function 'wuerfeln' [-Wimplicit-function-
  declaration]
    6 |     printf("%d\n", wuerfeln());
      |                    ^~~~~~~~
```

Diese Meldung kennst du aus Kapitel 3, dort von einem Tippfehler. Hier ist der Name richtig, er kommt nur zu spät. Es gibt zwei Lösungen:

1. Die Funktion **über** `main` schreiben. So haben es alle bisherigen Programme in diesem Kapitel gemacht.
2. Oben einen **Prototyp** hinschreiben: den Funktionskopf mit Semikolon, ohne Rumpf. Er sagt dem Compiler: „Diese Funktion gibt es, sie sieht so aus. Der Rumpf kommt später."

k11_prototyp.c

```c
// Mit Prototypen darf main oben stehen
#include "pbc.h"

// Prototypen: Name, Parameter und Rueckgabetyp, aber noch ohne Rumpf
int wuerfeln(void);
int zwei_wuerfel(void);

int main(void) {
    pbc_init();
    for (int i = 0; i < 5; i++) {
        printf("Wurf %d: %d\n", i + 1, zwei_wuerfel());
    }
    while (true) {
        sleep_ms(1000);
    }
}

int wuerfeln(void) {
    return pbc_random(1, 6);
}

int zwei_wuerfel(void) {
    return wuerfeln() + wuerfeln();
}
```

Serial Monitor

```
Wurf 1: 7
Wurf 2: 8
Wurf 3: 7
Wurf 4: 4
Wurf 5: 10
```

Die Zeilen 5 und 6 sind die Prototypen. Damit darf `main` ganz oben stehen, und man sieht beim Lesen zuerst, was das Programm tut. Die Einzelheiten folgen unten. Eine Funktion **ankündigen** heißt **Deklaration**, sie mit Rumpf hinzuschreiben **Definition**.

**Jetzt verstehst du `pbc.h`.** Die Datei, die du mit `#include "pbc.h"` in jedes Programm holst, besteht zum größten Teil aus Prototypen, etwa:

```
void pbc_fill_circle(int mx, int my, int r, uint16_t color);
bool pbc_pressed_a(void);
int pbc_random(int min, int max);
```

Die Rümpfe stehen in anderen Dateien der Bibliothek. `#include` kopiert die Prototypen an den Anfang deines Programms; so kennt der Compiler alle Befehle, bevor `main` beginnt. Mehr dazu in Kapitel 20.

## 11.4 Lokale und globale Variablen, `static`

**Lokale Variablen.** Eine Variable, die innerhalb einer Funktion angelegt wird, gibt es nur dort. Sie heißt **lokal**. Genauer gilt sie nur in dem Block `{ … }`, in dem sie angelegt wurde, und erst ab der Zeile, in der sie angelegt wird. Ist der Block zu Ende, ist sie weg.

k11_lokal.c

```
// Lokale Variablen: Jede Funktion hat ihre eigenen
#include "pbc.h"

void zaehle_bis_drei(void) {
    for (int i = 1; i <= 3; i++) {
        printf("%d ", i);
    }
    printf("\n");
}

void zeige_wert(void) {
    int wert = 7;                   // eine andere Variable als in main
    printf("zeige_wert: wert = %d\n", wert);
}

int main(void) {
    pbc_init();
    int wert = 100;
    zaehle_bis_drei();
    zeige_wert();
    printf("main:       wert = %d\n", wert);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
1 2 3
zeige_wert: wert = 7
main:       wert = 100
```

- Die Variable `wert` in `zeige_wert` (Zeile 12) und die in `main` (Zeile 18) haben nichts miteinander zu tun. Jede Funktion hat ihre eigene. Darum kannst du in jeder Funktion Namen wie `i`, `x` oder `text` benutzen, ohne dass sie sich in die Quere kommen.
- Das `i` aus der `for`-Schleife in Zeile 5 gibt es nur in dieser Schleife. Hinter ihrer `}` kennt der Compiler `i` nicht mehr.
- `zeige_wert` kann die Variable `wert` aus `main` nicht sehen. Braucht eine Funktion einen Wert, bekommt sie ihn als Parameter.

**Globale Variablen.** Eine Variable, die **außerhalb** jeder Funktion steht, heißt **global**. Alle Funktionen darunter können sie lesen und ändern:

k11_global.c

```c
// Eine globale Variable: alle Funktionen sehen dieselbe
#include "pbc.h"

int punkte = 0;

void treffer(void) {
    punkte += 10;
}

void zeige_punkte(void) {
    printf("Punkte: %d\n", punkte);
}

int main(void) {
    pbc_init();
    treffer();
    treffer();
    zeige_punkte();
    treffer();
    zeige_punkte();
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Punkte: 20
Punkte: 30
```

`punkte` (Zeile 4) gibt es nur einmal, und sie bleibt vom Start bis zum Ende des Programms bestehen. `treffer` erhöht sie, `zeige_punkte` gibt sie aus. Ohne Startwert beginnt eine globale Variable übrigens immer bei 0; eine lokale Variable ohne Startwert enthält dagegen irgendeine Zahl.

> Globale Variablen sind bequem, aber gefährlich: Jede Funktion kann sie ändern. Hat sie einen falschen Wert, musst du das ganze Programm durchsuchen. Benutze sie sparsam, für Dinge, die wirklich das ganze Programm betreffen, wie den Punktestand eines Spiels. Alles andere gehört in lokale Variablen und Parameter.

**`static`: lokal, aber mit Gedächtnis.** Eine lokale Variable entsteht bei jedem Aufruf neu und verschwindet am Ende wieder. Schreibst du `static` davor, bleibt sie bestehen und behält ihren Wert bis zum nächsten Aufruf. Sichtbar ist sie trotzdem nur in ihrer Funktion:

k11_static.c

```
1  // static: Die lokale Variable behaelt ihren Wert zwischen den Aufrufen
2  #include "pbc.h"
3
4  void klingeln(void) {
5      static int anzahl = 0;        // wird nur einmal auf 0 gesetzt
6      int nur_lokal = 0;            // wird bei jedem Aufruf auf 0 gesetzt
7      anzahl++;
8      nur_lokal++;
9      printf("Ding-dong! anzahl = %d, nur_lokal = %d\n", anzahl, nur_lokal);
10 }
11
12 int main(void) {
13     pbc_init();
14     klingeln();
15     klingeln();
16     klingeln();
17     while (true) {
18         sleep_ms(1000);
19     }
20 }
```

Serial Monitor

```
Ding-dong! anzahl = 1, nur_lokal = 1
Ding-dong! anzahl = 2, nur_lokal = 1
Ding-dong! anzahl = 3, nur_lokal = 1
```

- `anzahl` (Zeile 5) bekommt den Startwert 0 nur ein einziges Mal, beim Programmstart. Bei jedem Aufruf zählt Zeile 7 weiter.
- `nur_lokal` (Zeile 6) wird bei jedem Aufruf neu angelegt und auf 0 gesetzt. Nach `nur_lokal++` ist sie deshalb immer 1.

So kann sich eine Funktion etwas merken, zum Beispiel, wie oft sie schon aufgerufen wurde oder wann sie zuletzt etwas getan hat, ohne dass dafür eine globale Variable nötig ist.

| Art | wo angelegt | sichtbar | lebt |
|---|---|---|---|
| lokal | in einer Funktion, einem Block | nur in diesem Block | bis zum Ende des Blocks |
| `static` lokal | in einer Funktion, mit `static` | nur in dieser Funktion | das ganze Programm lang |
| global | außerhalb aller Funktionen | in allen Funktionen darunter | das ganze Programm lang |

## Ausprobieren: Ein Zeichenbaukasten

Jede Funktion zeichnet ein Teil einer Landschaft. `main` setzt die Teile nur noch zusammen:

k11_baukasten.c

```c
// Ein Zeichenbaukasten: Jede Funktion zeichnet ein Teil der Landschaft
#include "pbc.h"

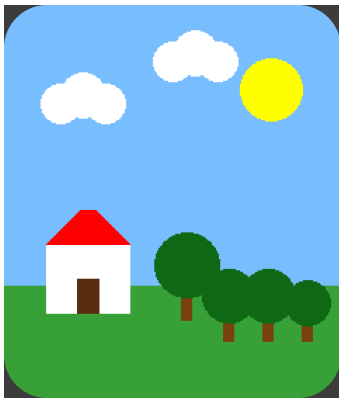
void zeichne_himmel(void) {
    pbc_clear(pbc_rgb(120, 190, 255));
    pbc_fill_rect(0, 200, PBC_WIDTH, 80, pbc_rgb(60, 160, 60));   // Wiese
}

void zeichne_sonne(int x, int y) {
    pbc_fill_circle(x, y, 22, PBC_YELLOW);
}

void zeichne_wolke(int x, int y) {
    pbc_fill_circle(x, y, 14, PBC_WHITE);
    pbc_fill_circle(x + 16, y - 6, 16, PBC_WHITE);
    pbc_fill_circle(x + 32, y, 14, PBC_WHITE);
}

void zeichne_baum(int x, int y, int hoehe) {
    pbc_fill_rect(x - 4, y - hoehe / 3, 8, hoehe / 3, pbc_rgb(120, 70, 20));
    pbc_fill_circle(x, y - hoehe / 3 - hoehe / 4, hoehe / 3, pbc_rgb(20, 110,
      30));
}

void zeichne_haus(int x, int y, uint16_t farbe) {
    pbc_fill_rect(x, y - 50, 60, 50, farbe);
    for (int i = 0; i < 25; i++) {                 // Dach aus Linien
        pbc_hline(x + i, y - 50 - i, 60 - 2 * i, PBC_RED);
    }
    pbc_fill_rect(x + 22, y - 25, 16, 25, pbc_rgb(90, 50, 20));   // Tuer
}
```



- `zeichne_himmel` braucht keine Parameter: Himmel und Wiese sind immer gleich.
- Sonne, Wolke, Baum und Haus bekommen eine Position. Bei Baum und Haus ist (x, y) der Punkt, an dem sie auf dem Boden stehen. So kannst du sie leicht auf die Wiese stellen.
- `zeichne_baum` hat noch den Parameter `hoehe`. Stamm und Krone werden daraus berechnet; ein Baum mit doppelter Höhe ist überall doppelt so groß.
- Das Dach des Hauses entsteht aus 25 waagrechten Linien, die nach oben immer kürzer werden (Zeilen 26 bis 28).

k11_baukasten.c

```
32 int main(void) {
33     pbc_init();
34     zeichne_himmel();
35     zeichne_sonne(190, 60);
36     zeichne_wolke(40, 70);
37     zeichne_wolke(120, 40);
38     zeichne_haus(30, 220, PBC_WHITE);
39     zeichne_baum(130, 225, 70);
40     for (int x = 160; x <= 216; x += 28) {
41         zeichne_baum(x, 240, pbc_random(40, 60));
42     }
43     pbc_show();
44     while (true) {
45         sleep_ms(1000);
46     }
47 }
```

`main` liest sich jetzt wie eine Beschreibung des Bilds: Himmel, Sonne, zwei Wolken, ein Haus, ein großer Baum und eine Reihe von Bäumen mit zufälliger Höhe. Die Schleife ruft dieselbe Funktion dreimal mit anderen Werten auf.

**Zum Verändern:** Schreibe eine Funktion `zeichne_blume(int x, int y, uint16_t farbe)` und setze ein paar Blumen auf die Wiese. Gib `zeichne_haus` einen weiteren Parameter für die Farbe des Dachs. Baue ein Dorf aus mehreren Häusern.

## Übungsaufgaben

> **Aufgabe 1 ⭐ Was gibt das Programm aus?** Lies das Programm, ohne es aufzuspielen. Was steht in der Konsole? Achte auf Kopien und auf die globale Variable.

k11_aufgabe1.c

```c
// Was gibt das Programm aus?
#include "pbc.h"

int zaehler = 0;

int naechste(int n) {
    zaehler++;
    return n + 1;
}

void aendern(int n) {
    n = 50;
    zaehler++;
    printf("n = %d\n", n);
}

int main(void) {
    pbc_init();
    int a = 5;
    aendern(a);
    int b = naechste(a);
    b = naechste(b) * 2;
    printf("a = %d, b = %d, zaehler = %d\n", a, b, zaehler);
    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 2 ⭐ Finde die Fehler** Das Programm soll die Fläche eines Rechtecks mit den Seiten 4 und 3 ausgeben. Es enthält drei Fehler. Baue es, verbessere den ersten gemeldeten Fehler und baue neu, bis es keine Fehler und keine Warnung mehr gibt.

k11_aufgabe2.c

```c
// Flaeche eines Rechtecks - mit drei Fehlern
#include "pbc.h"

int main(void) {
    pbc_init();
    int f = flaeche(4);
    printf("Flaeche: %d\n", f);
    while (true) {
        sleep_ms(1000);
    }
}

int flaeche(int breite, int hoehe) {
    int ergebnis = breite * hoehe;
}
```

**Aufgabe 3 ⭐⭐ Schaltjahre** Ein Jahr ist ein Schaltjahr, wenn es durch 4 teilbar ist. Ausnahme: Jahre, die durch 100 teilbar sind, sind keine Schaltjahre, außer sie sind auch durch 400 teilbar. Schreibe eine Funktion `bool ist_schaltjahr(int jahr)` und eine zweite Funktion, die für ein Jahr einen passenden Satz ausgibt. Prüfe 1900, 2000, 2024 und 2026.

**Aufgabe 4 ⭐⭐ Würfelbilder** Schreibe eine Funktion `void zeichne_wuerfel(int x, int y, int augen)`, die einen Würfel mit 1 bis 6 Augen zeichnet. Zeige drei zufällige Würfel nebeneinander und gib die Summe in der Konsole aus. Tipp: Eine kleine Hilfsfunktion, die ein einzelnes Auge zeichnet, spart viel Schreibarbeit.

**Aufgabe 5 ⭐⭐⭐ Primzahlen** Eine Primzahl ist eine Zahl größer als 1, die nur durch 1 und sich selbst teilbar ist. Schreibe `bool ist_primzahl(int n)` und gib damit alle Primzahlen bis 100 aus. Schreibe dann `int anzahl_primzahlen(int bis)`, die zählt, wie viele Primzahlen es bis zu einer Grenze gibt. Wie viele sind es bis 1000?

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- Eine Funktion: `rueckgabetyp name(parameter) { … }`. `void` heißt: kein Rückgabewert bzw. keine Parameter.
- Beim Aufruf werden die Argumente der Reihe nach in die Parameter **kopiert**. Ändert die Funktion einen Parameter, bleibt das Original unverändert.
- `return wert;` gibt einen Wert zurück und beendet die Funktion sofort. Eine Funktion mit Rückgabetyp braucht auf jedem Weg ein `return`.
- Eine Funktion muss vor ihrem ersten Aufruf bekannt sein: darüber definieren oder oben einen **Prototyp** hinschreiben. `pbc.h` ist eine Sammlung solcher Prototypen.
- **Lokale** Variablen gelten nur in ihrem Block, **globale** überall. Mit `static` behält eine lokale Variable ihren Wert von Aufruf zu Aufruf.

KAPITEL 12

# Arrays und Texte

**Worum geht's?** Ein Spiel hat zehn Gegner, eine Woche sieben Temperaturen, eine Melodie dreißig Töne. Für jeden Wert eine eigene Variable anzulegen, wäre mühsam. Ein **Array** fasst viele Werte unter einem Namen zusammen. Du lernst, Arrays anzulegen und zu durchlaufen, und erfährst, dass auch Texte in C nichts anderes sind als Arrays aus Zeichen. Außerdem siehst du die gefährlichste Eigenschaft von Arrays in C: Niemand passt auf, dass du an ihrem Ende aufhörst. Am Ende läuft ein bunter Text über den Bildschirm.

## 12.1 Arrays: viele Werte unter einem Namen

Ein **Array** (deutsch auch *Feld*) ist eine Reihe von Werten desselben Typs unter einem gemeinsamen Namen. Du legst es an wie eine Variable, mit der Anzahl der Werte in eckigen Klammern:

k12_array.c

```c
// Ein Array: fuenf Punktzahlen unter einem Namen
#include "pbc.h"

int main(void) {
    pbc_init();
    int punkte[5] = {12, 7, 30, 5, 18};

    printf("Erster Wert:  %d\n", punkte[0]);
    printf("Letzter Wert: %d\n", punkte[4]);

    punkte[1] = 25;               // den zweiten Wert ersetzen
    punkte[3] += 10;              // zum vierten Wert 10 addieren
    for (int i = 0; i < 5; i++) {
        printf("punkte[%d] = %d\n", i, punkte[i]);
    }

    int leer[4] = {0};            // alle vier Werte sind 0
    int teils[4] = {1, 2};        // die fehlenden Werte sind 0
    printf("leer[3] = %d, teils[3] = %d\n", leer[3], teils[3]);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Erster Wert:  12
Letzter Wert: 18
punkte[0] = 12
punkte[1] = 25
punkte[2] = 30
punkte[3] = 15
punkte[4] = 18
leer[3] = 0, teils[3] = 0
```

- Zeile 6 legt ein Array `punkte` mit Platz für fünf `int` an. In den geschweiften Klammern stehen die Startwerte, der Reihe nach.
- Auf einen einzelnen Wert greifst du mit seiner Nummer in eckigen Klammern zu: `punkte[0]`. Diese Nummer heißt **Index**. Ein einzelner Wert des Arrays heißt **Element**.
- **Der Index beginnt bei 0.** Das erste Element ist `punkte[0]`, das fünfte und letzte ist `punkte[4]`. Ein `punkte[5]` gibt es nicht (Abschnitt 12.5).
- Ein Element benutzt du wie eine gewöhnliche Variable: Du kannst ihm etwas zuweisen (Zeile 11) oder damit rechnen (Zeile 12).
- Der Index darf auch eine Variable sein. Die Schleife in Zeile 13 geht mit `i` alle Elemente von 0 bis 4 durch.

**Startwerte.** Gibst du weniger Startwerte an, als das Array Plätze hat, werden die übrigen auf 0 gesetzt. `{0}` (Zeile 17) ist deshalb der übliche Weg, ein Array mit lauter Nullen anzulegen. Gibst du **gar keine** Startwerte an, gilt dasselbe wie bei einfachen Variablen: Ein lokales Array enthält dann irgendwelche zufälligen Zahlen, ein globales lauter Nullen.

> Die Größe eines Arrays steht fest, sobald es angelegt ist. Es kann später weder wachsen noch schrumpfen. Lege es deshalb so groß an, wie du höchstens brauchst, und merke dir in einer eigenen Variable, wie viele Plätze tatsächlich benutzt sind.

## 12.2 Arrays durchlaufen, Größe mit `sizeof`

Fast immer bearbeitet man ein Array mit einer `for`-Schleife, Element für Element. Das folgende Programm wertet die Temperaturen einer Woche aus:

k12_wetter.c

```
 1 // Temperaturen einer Woche auswerten
 2 #include "pbc.h"
 3
 4 int main(void) {
 5     pbc_init();
 6     int temperatur[] = {14, 17, 21, 19, 23, 16, 15};
 7     int anzahl = sizeof temperatur / sizeof temperatur[0];
 8     printf("Das Array belegt %d Bytes, das sind %d Werte.\n",
 9            (int)sizeof temperatur, anzahl);
10
11     int summe = 0;
12     int warm_tag = 0;             // Index des waermsten Tages
13     for (int i = 0; i < anzahl; i++) {
14         summe += temperatur[i];
15         if (temperatur[i] > temperatur[warm_tag]) {
16             warm_tag = i;
17         }
18     }
```

```
19      printf("Durchschnitt: %.1f Grad\n", (float)summe / anzahl);
20      printf("Am waermsten: Tag %d mit %d Grad\n", warm_tag + 1,
          temperatur[warm_tag]);
21      while (true) {
22          sleep_ms(1000);
23      }
24  }
```

Serial Monitor

```
Das Array belegt 28 Bytes, das sind 7 Werte.
Durchschnitt: 17.9 Grad
Am waermsten: Tag 5 mit 23 Grad
```

- In Zeile 6 stehen die eckigen Klammern leer. Dann zählt der Compiler die Startwerte selbst und macht das Array genau so groß: hier 7.
- `sizeof temperatur` ist die Größe des ganzen Arrays in **Bytes**. Ein `int` belegt auf dem PicoBoy 4 Bytes, sieben davon also 28. `sizeof temperatur[0]` ist die Größe eines einzelnen Elements. Geteilt ergibt das die **Anzahl** der Elemente (Zeile 7). Kommt ein achter Tag dazu, passt sich `anzahl` von selbst an.
- Die Schleife ab Zeile 13 addiert alle Werte und merkt sich in `warm_tag` den **Index** des bisher wärmsten Tags. Sie beginnt mit Tag 0 und vergleicht jeden weiteren Tag mit dem bisherigen Spitzenreiter.
- Zeile 19 teilt die Summe durch die Anzahl. `(float)` sorgt dafür, dass mit Kommazahlen gerechnet wird, sonst fielen die Nachkommastellen weg (Kapitel 5).
- Für Menschen beginnen Tage bei 1, nicht bei 0. Deshalb gibt Zeile 20 `warm_tag + 1` aus.

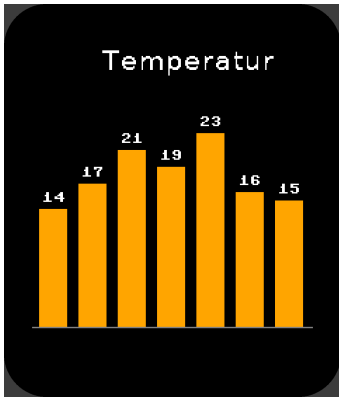
Ein Array eignet sich auch gut zum Zeichnen. Jeder Wert wird ein Balken:

k12_balken.c

```
 1  // Ein Balkendiagramm aus einem Array
 2  #include "pbc.h"
 3
 4  #define ANZAHL 7
 5
 6  int main(void) {
 7      pbc_init();
 8      int temperatur[ANZAHL] = {14, 17, 21, 19, 23, 16, 15};
 9      char text[8];
10      pbc_clear(PBC_BLACK);
11      pbc_text18("Temperatur", 70, 30, PBC_WHITE);
12      for (int i = 0; i < ANZAHL; i++) {
13          int x = 25 + i * 28;
14          int hoehe = temperatur[i] * 6;
15          pbc_fill_rect(x, 230 - hoehe, 20, hoehe, PBC_ORANGE);
16          snprintf(text, sizeof text, "%d", temperatur[i]);
17          pbc_text(text, x + 2, 230 - hoehe - 12, PBC_WHITE);
18      }
```

```
19      pbc_hline(20, 230, 200, PBC_GRAY);
20      pbc_show();
21      while (true) {
22          sleep_ms(1000);
23      }
24  }
```

`ANZAHL` steht hier als `#define` oben. Das ist der zweite übliche Weg: Dann steht die Größe nur an einer Stelle, und Schleife und Array passen immer zusammen. In der Schleife wird aus dem Index `i` die x-Position (Zeile 13) und aus dem Wert die Höhe des Balkens (Zeile 14). Weil y nach unten wächst, beginnt ein Balken der Höhe `hoehe` bei `230 - hoehe`.




**Arrays als Parameter.** Ein Array kannst du an eine Funktion übergeben. Im Funktionskopf schreibst du dafür leere eckige Klammern:

k12_funktion.c

```
1  // Arrays an Funktionen uebergeben
2  #include "pbc.h"
3
4  int summe(int werte[], int anzahl) {
5      int s = 0;
6      for (int i = 0; i < anzahl; i++) {
7          s += werte[i];
8      }
9      return s;
10 }
11
12 void verdopple_alle(int werte[], int anzahl) {
13     for (int i = 0; i < anzahl; i++) {
14         werte[i] *= 2;
15     }
16 }
17
18 int main(void) {
19     pbc_init();
20     int zahlen[4] = {1, 2, 3, 4};
21     printf("Summe: %d\n", summe(zahlen, 4));
22     verdopple_alle(zahlen, 4);
23     printf("Nach verdopple_alle: %d %d %d %d\n", zahlen[0], zahlen[1], zahlen[2],
         zahlen[3]);
24     printf("Summe: %d\n", summe(zahlen, 4));
25     while (true) {
26         sleep_ms(1000);
27     }
28 }
```

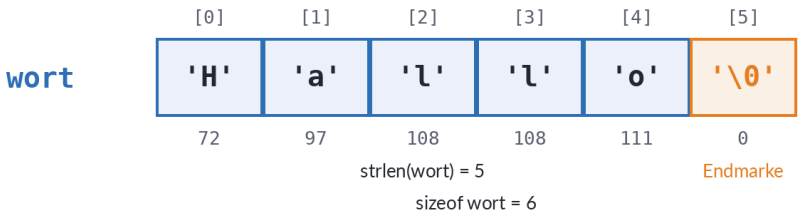
```
Serial Monitor
Summe: 10
Nach verdopple_alle: 2 4 6 8
Summe: 20
```

Zwei Dinge sind anders als bei einfachen Parametern:

- Die Funktion kennt die Größe des Arrays nicht. Du gibst sie deshalb als zweiten Parameter mit (`anzahl`). `sizeof` hilft innerhalb der Funktion nicht; Kapitel 19 zeigt, warum.
- Ein Array wird **nicht kopiert**. Die Funktion arbeitet mit dem Original. `verdopple_alle` ändert deshalb wirklich die Werte in `main`, anders als `verdopple` in Kapitel 11. Den Grund erfährst du ebenfalls in Kapitel 19.

## 12.3 Zeichenketten sind `char`-Arrays mit `'\0'`

Seit Kapitel 4 legst du Texte mit `char name[30];` an. Jetzt verstehst du die Schreibweise: `name` ist ein Array aus 30 `char`. Jeder Buchstabe belegt ein Element. Damit man weiß, wo der Text endet, steht hinter dem letzten Buchstaben ein besonderes Zeichen: `'\0'`, die **Null-Endmarke**, ein Byte mit dem Wert 0.




k12_zeichen.c

```c
// Eine Zeichenkette ist ein char-Array mit '\0' am Ende
#include "pbc.h"

int main(void) {
    pbc_init();
    char wort[] = "Hallo";
    printf("sizeof: %d, strlen: %d\n", (int)sizeof wort, (int)strlen(wort));

    for (int i = 0; wort[i] != '\0'; i++) {
        printf("wort[%d] = '%c' = %d\n", i, wort[i], wort[i]);
    }
    printf("wort[5] = %d (das Ende)\n", wort[5]);

    wort[0] = 'B';
    wort[4] = 'e';
    printf("%s\n", wort);
    wort[2] = '\0';
    printf("%s\n", wort);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
sizeof: 6, strlen: 5
wort[0] = 'H' = 72
wort[1] = 'a' = 97
wort[2] = 'l' = 108
wort[3] = 'l' = 108
wort[4] = 'o' = 111
wort[5] = 0 (das Ende)
Balle
Ba
```

- `char wort[] = "Hallo";` in Zeile 6 legt ein Array für fünf Buchstaben **und** die Endmarke an. `sizeof wort` ist deshalb 6.
- `strlen(wort)` zählt die Zeichen bis zur Endmarke, ohne sie mitzuzählen: 5. Der Name kommt von *string length*, Länge der Zeichenkette.
- Die Schleife in Zeile 9 läuft, bis sie auf `'\0'` stößt. So arbeiten alle Funktionen, die mit Texten umgehen, auch `printf` mit `%s` und `pbc_text24`.
- Jedes Zeichen ist intern eine Zahl. `%c` gibt sie als Zeichen aus, `%d` als Zahl: `'H'` ist 72, `'a'` ist 97. Diese Zuordnung heißt **ASCII**. Deshalb steht ein einzelnes Zeichen in einfachen Anführungszeichen (`'H'`, eine Zahl), ein Text in doppelten (`"H"`, ein Array aus `'H'` und `'\0'`).
- Einzelne Zeichen kannst du ändern (Zeilen 14 und 15). Setzt du mitten im Text eine `'\0'` (Zeile 17), endet der Text dort. Die Buchstaben dahinter sind noch im Array, aber niemand liest sie mehr.

**Mehrere Texte.** Ein Array kann auch Arrays enthalten. `char tage[7][3]` sind sieben Texte mit je drei `char`: zwei Buchstaben und die Endmarke.

k12_tage.c

```
 1 // Mehrere Texte: ein Array aus char-Arrays
 2 #include "pbc.h"
 3
 4 int main(void) {
 5     pbc_init();
 6     char tage[7][3] = {"Mo", "Di", "Mi", "Do", "Fr", "Sa", "So"};
 7     int temperatur[7] = {14, 17, 21, 19, 23, 16, 15};
 8     for (int i = 0; i < 7; i++) {
 9         printf("%s: %d Grad\n", tage[i], temperatur[i]);
10     }
11     printf("Erster Buchstabe von tage[4]: %c\n", tage[4][0]);
12     while (true) {
13         sleep_ms(1000);
14     }
15 }
```

Serial Monitor

```
Mo: 14 Grad
Di: 17 Grad
Mi: 21 Grad
Do: 19 Grad
Fr: 23 Grad
Sa: 16 Grad
So: 15 Grad
Erster Buchstabe von tage[4]: F
```

`tage[i]` ist der i-te Text, `tage[4][0]` das erste Zeichen des fünften Texts.

**Umlaute.** Das Programm ist in einer Datei gespeichert, und die speichert VS Code im Format **UTF-8**. Darin sind nur die Zeichen der englischen Tastatur ein Byte lang. Ein Umlaut oder ein ß belegt **zwei** Bytes:

k12_umlaute.c

```c
// Umlaute sind in C mehrere Bytes lang
#include "pbc.h"

int main(void) {
    pbc_init();
    char gruss[] = "Grüße";
    printf("%s hat %d Bytes\n", gruss, (int)strlen(gruss));
    for (int i = 0; gruss[i] != '\0'; i++) {
        printf("%d ", (uint8_t)gruss[i]);
    }
    printf("\n");

    pbc_clear(PBC_BLACK);
    pbc_text24(gruss, 60, 100, PBC_WHITE);
    pbc_text24("Gruesse", 60, 150, PBC_GREEN);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Grüße hat 7 Bytes
71 114 195 188 195 159 101
```




„Grüße“ hat fünf Buchstaben, aber sieben Bytes: ü ist 195 188, ß ist 195 159. Der serielle Monitor kennt UTF-8 und setzt die Bytes wieder zu Buchstaben zusammen. Die Schriften des PicoBoy kennen dagegen nur die Zeichen bis 127. Jedes andere Byte zeichnen sie als Fragezeichen, deshalb erscheinen hier **vier** Fragezeichen. Auch `strlen` zählt Bytes, nicht Buchstaben. Und löschst du bei einer Eingabe mit `pbc_input` einen Umlaut mit der Rücktaste, entfernt sie nur eines der beiden Bytes. Darum stehen in diesem Buch auch in der Konsole „ae“, „oe“, „ue“ und „ss“.

**Texte zuweisen und vergleichen.** Mit `=` und `==` kann man ganze Arrays nicht bearbeiten, also auch keine Texte. Eine Zuweisung lehnt der Compiler ab:

k12_zuweisung_falsch.c

```c
// Fehler: Texte lassen sich nicht mit = zuweisen
#include "pbc.h"

int main(void) {
    pbc_init();
    char name[20] = "Mia";
    name = "Ben";
    printf("%s\n", name);
    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:7:10: error: assignment to expression with array type
    7 |     name = "Ben";
      |          ^
```

Ein Vergleich mit `==` baut sogar, ist aber trotzdem falsch: Er vergleicht nicht die Buchstaben, sondern nur, ob beide Texte an derselben Stelle im Speicher liegen. Das ist praktisch nie der Fall. Der Compiler warnt:

k12_vergleich_falsch.c

```c
    if (wort == "sesam") {
        printf("Willkommen!\n");
    }
```

Terminal

```
main.c: In function 'main':
main.c:8:14: warning: comparison with string literal results in unspecified
  behavior [-Waddress]
    8 |     if (wort == "sesam") {
      |              ^~
```

Für Texte gibt es eigene Funktionen. Zum Vergleichen nimmst du `strcmp` (*string compare*). Sie liefert 0, wenn beide Texte gleich sind. Zum Kopieren nimmst du `snprintf`, das du schon kennst:

k12_passwort.c

```c
// Texte vergleichen mit strcmp, kopieren mit snprintf
#include "pbc.h"

int main(void) {
    pbc_init();
    char wort[20];
    char name[20] = "Gast";
    for (int versuch = 1; versuch <= 3; versuch++) {
        pbc_input("Passwort: ", wort, sizeof wort);
        if (strcmp(wort, "sesam") == 0) {
            snprintf(name, sizeof name, "Ali Baba");
            break;
        }
        printf("Falsch (Versuch %d von 3)\n", versuch);
    }
    printf("Hallo %s!\n", name);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Passwort: simsalabim
Falsch (Versuch 1 von 3)
Passwort: sesam
Hallo Ali Baba!
```

- Zeile 10: `strcmp(wort, "sesam") == 0` ist wahr, wenn `wort` genau „sesam“ ist, Buchstabe für Buchstabe, auch in Groß- und Kleinschreibung.
- Zeile 11 schreibt einen neuen Text in `name`. Ein `snprintf` ohne Platzhalter kopiert einfach den Text, und zwar nie mehr, als in `name` hineinpasst.

## 12.4 Texte zusammensetzen mit `snprintf`

In Kapitel 6 hast du mit `snprintf` Zahlen für den Bildschirm in Text verwandelt. `snprintf` kann alles, was `printf` kann. Der Unterschied: Das Ergebnis landet nicht in der Konsole, sondern in einem `char`-Array. So setzt du aus Texten und Zahlen neue Texte zusammen:

k12_snprintf.c

```c
// Texte zusammensetzen mit snprintf
#include "pbc.h"

int main(void) {
    pbc_init();
    char name[20];
    pbc_input("Name: ", name, sizeof name);
```

```
    int punkte = 1250;
    int minuten = 3;
    int sekunden = 7;
    char zeile[40];

    snprintf(zeile, sizeof zeile, "%s hat %d Punkte", name, punkte);
    printf("[%s]\n", zeile);

    snprintf(zeile, sizeof zeile, "Zeit %d:%02d", minuten, sekunden);
    printf("[%s]\n", zeile);

    snprintf(zeile, sizeof zeile, "%-12s|%6d|", name, punkte);
    printf("[%s]\n", zeile);

    snprintf(zeile, sizeof zeile, "Anfangsbuchstabe: %c", name[0]);
    printf("[%s]\n", zeile);

    // Passt der Text nicht hinein, wird er abgeschnitten
    char kurz[20];
    int noetig = snprintf(kurz, sizeof kurz, "%s hat %d Punkte", name, punkte);
    printf("[%s] noetig: %d Zeichen, Platz: %d\n", kurz, noetig, (int)sizeof kurz
      - 1);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Name: Maximilian
[Maximilian hat 1250 Punkte]
[Zeit 3:07]
[Maximilian  |  1250|]
[Anfangsbuchstabe: M]
[Maximilian hat 1250] noetig: 26 Zeichen, Platz: 19
```

Die eckigen Klammern in der Ausgabe zeigen, wo der Text anfängt und aufhört.

| Platzhalter | Bedeutung | Beispiel |
|---|---|---|
| `%s` | ein Text | `Maximilian` |
| `%02d` | ganze Zahl, mindestens 2 Stellen, vorne mit 0 aufgefüllt | `07` |
| `%6d` | ganze Zahl, mindestens 6 Zeichen breit, rechtsbündig | `1250` |
| `%-12s` | Text, mindestens 12 Zeichen breit, linksbündig | `Maximilian` |
| `%c` | ein einzelnes Zeichen | `M` |

Breiten wie `%6d` sind praktisch für Tabellen und für `pbc_text`, dessen Zeichen alle gleich breit sind: Dann stehen Zahlen genau untereinander.

**Wenn der Platz nicht reicht.** Zeile 27 schreibt in ein Array mit 20 Plätzen. Davon braucht die Endmarke einen, also passen 19 Zeichen hinein. `snprintf` schreibt genau so viele, hängt die `'\0'` an und lässt den Rest weg. Der Text ist dann zwar abgeschnitten, aber sicher. Der **Rückgabewert** von `snprintf` verrät, wie lang der ganze Text geworden wäre: hier 26. Ist er größer als der Platz, wurde gekürzt.

> `snprintf(text, sizeof text, …)` schreibt nie über das Ende von `text` hinaus und schließt immer mit `'\0'` ab. Nimm `snprintf` zum Kopieren und Zusammensetzen von Texten, immer mit `sizeof` des Ziels.

## 12.5 Über das Ende hinaus: warum C nicht warnt

Was passiert, wenn ein Programm `punkte[5]` benutzt, obwohl `punkte` nur fünf Elemente hat (Index 0 bis 4)? In vielen Sprachen bricht das Programm dann mit einer Fehlermeldung ab. **C prüft das nicht.** `punkte[5]` ist für C einfach die Speicherstelle direkt hinter `punkte[4]`. Was dort liegt, weiß niemand: vielleicht eine andere Variable, vielleicht gar nichts Wichtiges, vielleicht Daten, die das SDK braucht.

Dieses Programm fragt nach einem Platz und prüft die Eingabe nicht:

k12_ueber_ende.c

```c
// Gefaehrlich: Der Index wird nicht geprueft
#include "pbc.h"

int main(void) {
    pbc_init();
    int punkte[5] = {0};
    int leben = 3;
    while (true) {
        int platz = pbc_input_int("Welcher Platz (0 bis 4)? ");
        punkte[platz] += 10;      // auch bei platz = 5, 100 oder -1!
        printf("Platz %d: %d Punkte, leben = %d\n", platz, punkte[platz], leben);
    }
}
```

Es baut ohne Fehler und ohne Warnung, denn welche Zahl eingegeben wird, weiß der Compiler beim Bauen nicht. Für 0 bis 4 tut es, was es soll. Gibst du 5, 100 oder -1 ein, verändert Zeile 10 irgendeinen anderen Speicher. Was dann passiert, legt C **nicht** fest. Fachleute nennen das **undefiniertes Verhalten**. Mögliche Folgen:

- Scheinbar nichts. Das Programm läuft weiter, als wäre alles in Ordnung.
- Eine **andere Variable** ändert sich, etwa `leben`, obwohl keine Zeile im Programm sie ändert.
- Das Programm bleibt stehen oder startet neu, weil etwas Wichtiges überschrieben wurde.

Welche Folge eintritt, kann sich ändern, wenn du an einer ganz anderen Stelle im Programm etwas änderst. Gerade das macht solche Fehler so schwer zu finden.

**Manchmal warnt der Compiler doch.** Steht alles schon im Programm fest, rechnet er nach. Hier läuft die Schleife mit `<=` einen Schritt zu weit:

k12_schleife_zu_weit.c

```c
    int werte[5];
    for (int i = 0; i <= 5; i++) {
        werte[i] = i * 10;
    }
```

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
Linking CXX executable k12_schleife_zu_weit.elf
Built target k12_schleife_zu_weit
main.c: In function 'main':
main.c:8:18: warning: iteration 5 invokes undefined behavior [-Waggressive-loop-
  optimizations]
    8 |         werte[i] = i * 10;
      |         ~~~~~~~~~^~~~~~~
main.c:7:23: note: within this loop
    7 |     for (int i = 0; i <= 5; i++) {
      |                     ~~^~~~
main.c:8:14: warning: array subscript 5 is above array bounds of 'int[5]' [-Warray-
  bounds=]
    8 |         werte[i] = i * 10;
      |         ~~~~~^~~
main.c:6:9: note: while referencing 'werte'
    6 |     int werte[5];
      |         ^~~~~
```

*Iteration 5 invokes undefined behavior*: Der sechste Durchlauf (mit `i` = 5) führt zu undefiniertem Verhalten. *Array subscript 5 is above array bounds*: Index 5 liegt hinter dem Ende. Verlassen kannst du dich auf solche Warnungen aber nicht: Sobald der Index von einer Eingabe, einer Rechnung oder einer Zufallszahl abhängt, schweigt der Compiler.

**Die Lösung: selbst prüfen.** Kommt ein Index von außen, prüfst du ihn, bevor du ihn benutzt:

k12_index_pruefen.c

```c
// Einen Index pruefen, bevor man ihn benutzt
#include "pbc.h"

#define ANZAHL 5

int main(void) {
    pbc_init();
    int punkte[ANZAHL] = {0};
    while (true) {
        int platz = pbc_input_int("Welcher Platz (0 bis 4)? ");
```

```
11          if (platz < 0 || platz >= ANZAHL) {
12              printf("Den Platz %d gibt es nicht.\n", platz);
13              continue;
14          }
15          punkte[platz] += 10;
16          printf("Platz %d hat jetzt %d Punkte.\n", platz, punkte[platz]);
17      }
18  }
```

Serial Monitor

```
Welcher Platz (0 bis 4)? 2
Platz 2 hat jetzt 10 Punkte.
Welcher Platz (0 bis 4)? 2
Platz 2 hat jetzt 20 Punkte.
Welcher Platz (0 bis 4)? 7
Den Platz 7 gibt es nicht.
Welcher Platz (0 bis 4)? -1
Den Platz -1 gibt es nicht.
Welcher Platz (0 bis 4)? 4
Platz 4 hat jetzt 10 Punkte.
Welcher Platz (0 bis 4)?
```

Zeile 11 lässt nur Werte von 0 bis `ANZAHL - 1` durch. Alles andere wird gemeldet, und `continue` springt zur nächsten Eingabe.

**Bei Texten gilt dasselbe.** Ein Text, der länger ist als sein Array, schreibt über dessen Ende hinaus. Und ein Text ohne `'\0'` hat kein Ende: `printf` und `strlen` lesen dann weiter in fremdem Speicher. Darum bekommen `pbc_input` und `snprintf` immer die Größe des Ziels mit.

> C prüft keine Grenzen. Ein Index muss immer zwischen 0 und Anzahl − 1 liegen, dafür sorgst du selbst. Typische Ursachen: `<=` statt `<` in der Schleife, ein ungeprüfter Index aus einer Eingabe, ein Text ohne Platz für `'\0'`.

> **Große Arrays.** Lokale Variablen, also auch lokale Arrays, liegen in einem Speicherbereich, der auf dem PicoBoy nur 4 KB groß ist. Ein lokales Array mit tausend `int` (4000 Bytes) passt nicht hinein. Große Arrays legst du deshalb global an oder mit `static` (Kapitel 11.4). Kapitel 19 erklärt die Speicheraufteilung genau.

## Ausprobieren: Ein Lauftext

Eine Meldung läuft von rechts nach links über den Bildschirm, wie auf einer Anzeigetafel. Jedes Zeichen hat eine andere Farbe. Mit A schaltest du zur nächsten Meldung.







k12_lauftext.c

```c
// Lauftext: Die Meldung laeuft von rechts nach links, jedes Zeichen bunt
#include "pbc.h"

#define ANZAHL_MELDUNGEN 3

char meldungen[ANZAHL_MELDUNGEN][40] = {
    "Hallo PicoBoy!",
    "Programmieren in C macht Spass",
    "Druecke A fuer die naechste Meldung",
};
uint16_t farben[] = {PBC_RED, PBC_ORANGE, PBC_YELLOW, PBC_GREEN, PBC_CYAN,
  PBC_MAGENTA};

int main(void) {
    pbc_init();
    int anzahl_farben = sizeof farben / sizeof farben[0];
    int nr = 0;                    // welche Meldung gerade laeuft
    int x = PBC_WIDTH;             // wo ihr erstes Zeichen steht
    char kopf[30];
    while (true) {
        if (pbc_was_pressed_a()) {
            nr = (nr + 1) % ANZAHL_MELDUNGEN;
            x = PBC_WIDTH;
        }

        pbc_clear(PBC_BLACK);
        snprintf(kopf, sizeof kopf, "Meldung %d von %d", nr + 1,
          ANZAHL_MELDUNGEN);
        pbc_text12(kopf, 60, 60, PBC_GRAY);

        // Zeichen fuer Zeichen: jedes ist ein eigener Text aus Zeichen und '\0'
        int zx = x;
        for (int i = 0; meldungen[nr][i] != '\0'; i++) {
            char zeichen[2] = {meldungen[nr][i], '\0'};
            zx = pbc_text24(zeichen, zx, 124, farben[i % anzahl_farben]);
        }
        pbc_show();

        x -= 3;
        if (zx < 0) {              // ganz links hinaus: rechts neu anfangen
            x = PBC_WIDTH;
        }
```

```
41         sleep_ms(20);
42     }
43 }
```

- Die Zeilen 6 bis 10 legen drei Meldungen mit je höchstens 39 Zeichen an, ein Array aus `char`-Arrays wie `tage` oben. `farben` in Zeile 11 ist ein Array aus Farben; seine Größe zählt der Compiler.
- `x` ist die Position des ersten Zeichens. Bei jedem Durchlauf rückt sie um 3 Pixel nach links (Zeile 37). Ein Text, der links aus dem Bild ragt, wird einfach abgeschnitten, genau wie rechts.
- Um jedes Zeichen in einer eigenen Farbe zu zeichnen, braucht `pbc_text24` jedes Zeichen als eigenen Text. Zeile 32 baut ihn: ein Array aus zwei `char`, dem Zeichen und der Endmarke. `pbc_text24` gibt die x-Position hinter dem Zeichen zurück; dort beginnt das nächste.
- `i % anzahl_farben` geht die sechs Farben immer wieder von vorn durch.
- Nach der Schleife steht in `zx` das rechte Ende des Texts. Ist es kleiner als 0, ist der ganze Text links hinausgelaufen, und er beginnt rechts von vorn (Zeile 38).
- Zeile 26 setzt mit `snprintf` die Kopfzeile zusammen.

**Zum Verändern:** Füge eine vierte Meldung hinzu (Achtung: Was musst du noch ändern?). Lass den Text mit dem Joystick schneller und langsamer laufen. Baue mit `snprintf` eine Meldung, die die Zeit seit dem Start enthält.

## Übungsaufgaben

**Aufgabe 1 ⭐ Was gibt das Programm aus?** Lies das Programm, ohne es aufzuspielen. Welche zwei Zeilen erscheinen in der Konsole?

k12_aufgabe1.c

```c
// Was gibt das Programm aus?
#include "pbc.h"

int main(void) {
    pbc_init();
    int z[6] = {3, 1, 4, 1, 5, 9};
    int a = z[2] + z[5];
    z[0] = z[z[1]];
    int s = 0;
    for (int i = 1; i < 6; i += 2) {
        s += z[i];
    }
    printf("%d %d %d\n", a, z[0], s);

    char w[] = "Bilder";
    w[3] = '\0';
    printf("%s %d %d\n", w, (int)strlen(w), (int)sizeof w);
```

```
    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 2 ⭐⭐ Finde den Fehler** Dieses Programm baut ohne Fehler und ohne Warnung. Auf dem PicoBoy gibt es vielleicht sogar das Richtige aus, vielleicht aber auch einen falschen Buchstaben-Zähler oder Zeichensalat hinter „Hallo“. Warum? Verbessere es.

k12_aufgabe2.c

```c
// Warum kommt hier manchmal Unsinn heraus?
#include "pbc.h"

int main(void) {
    pbc_init();
    char wort[5] = "Hallo";
    int zahl = 42;
    printf("%s hat %d Buchstaben\n", wort, (int)strlen(wort));
    printf("Die Zahl ist %d\n", zahl);
    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 3 ⭐⭐ Würfelstatistik** Würfle 600-mal mit `pbc_random(1, 6)` und zähle in einem Array, wie oft jede Augenzahl kommt. Zeige das Ergebnis als Balkendiagramm und in der Konsole. Wie oft sollte jede Zahl ungefähr kommen?

**Aufgabe 4 ⭐⭐ Rückwärts** Lies immer wieder ein Wort aus der Konsole und gib es rückwärts aus. Ist das Wort vorwärts und rückwärts gleich (wie „rentner“ oder „otto“), soll zusätzlich „Ein Palindrom!“ erscheinen. Schreibe dafür eine Funktion `bool ist_palindrom(char wort[])`.

**Aufgabe 5 ⭐⭐⭐ Eine Melodie als Text** Schreibe eine Funktion `void spiele(const char melodie[])`, die eine Melodie aus einem Text spielt: Jeder Buchstabe von `c` bis `h` ist eine Viertelnote (Frequenzen in Kapitel 10), ein `-` ist eine Pause derselben Länge. Spiele damit „Alle meine Entchen“: `"cdefg-g-aaaag---aaaag---ffffe-e-ddddc---"`. Tipp: Eine zweite Funktion `int frequenz(char ton)` mit `switch` liefert zu jedem Buchstaben die Frequenz.

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- `int werte[5] = {…};` legt ein Array an. Der Index läuft von **0** bis Anzahl − 1. `{0}` setzt alle Elemente auf 0.
- Anzahl der Elemente: `sizeof werte / sizeof werte[0]` oder ein `#define`. Einer Funktion gibst du die Anzahl als Parameter mit; sie arbeitet mit dem Original-Array.
- Ein Text ist ein `char`-Array mit `'\0'` am Ende. `strlen` zählt die Zeichen ohne Endmarke. Ein Umlaut ist in UTF-8 zwei Bytes lang.
- Texte nicht mit `=` und `==` bearbeiten: `strcmp(a, b) == 0` vergleicht, `snprintf(ziel, sizeof ziel, …)` kopiert und setzt zusammen.
- C prüft keine Grenzen. Über das Ende hinaus schreiben ist undefiniertes Verhalten. Einen Index von außen immer selbst prüfen.

KAPITEL 13

# Strukturen

**Worum geht's?** Eine Spielfigur hat eine Position, eine Größe, eine Farbe, vielleicht eine Geschwindigkeit. Bisher brauchte jede dieser Angaben ihre eigene Variable. Eine **Struktur** bündelt sie unter einem Namen. Du lernst, Strukturen anzulegen, an Funktionen zu übergeben und in Arrays zu sammeln. Mit `typedef` gibst du ihnen kurze Namen, mit `enum` benennst du Zustände wie „hoch", „runter" oder „Rot", „Grün". Am Ende schwimmt ein ganzer Fischschwarm über den Bildschirm.

## 13.1 `struct`: Werte bündeln

Ein Punkt auf dem Bildschirm besteht aus zwei Zahlen, x und y. Mit `struct` machst du daraus einen neuen Typ:

k13_punkt.c

```c
// Eine Struktur buendelt zwei Werte zu einem Punkt
#include "pbc.h"

struct punkt {
    int x;
    int y;
};

int main(void) {
    pbc_init();
    struct punkt a = {10, 20};
    struct punkt b = {.x = 100, .y = 50};

    printf("a = (%d, %d)\n", a.x, a.y);
    a.x = a.x + 5;                // ein einzelnes Feld aendern
    printf("a = (%d, %d)\n", a.x, a.y);

    struct punkt c = b;           // kopiert alle Felder auf einmal
    c.y = 0;
    printf("b = (%d, %d), c = (%d, %d)\n", b.x, b.y, c.x, c.y);
    printf("Ein Punkt belegt %d Bytes\n", (int)sizeof(struct punkt));
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
a = (10, 20)
a = (15, 20)
b = (100, 50), c = (100, 0)
Ein Punkt belegt 8 Bytes
```

- Die Zeilen 4 bis 7 beschreiben, wie eine `struct punkt` aussieht: Sie hat zwei **Felder** (auch *Komponenten*), `x` und `y`, beide `int`. Das ist wie ein Bauplan. Eine Variable entsteht dadurch noch nicht.
- **Achtung, Semikolon:** Hinter der schließenden Klammer in Zeile 7 steht ein `;`. Bei Funktionen steht dort keins, bei Strukturen schon.
- Zeile 11 legt eine Variable `a` vom Typ `struct punkt` an. Die Startwerte in geschweiften Klammern gehen der Reihe nach in die Felder: `x` bekommt 10, `y` bekommt 20.
- Zeile 12 nennt die Felder beim Namen: `.x = 100`. Das ist deutlicher, und die Reihenfolge ist egal. Felder ohne Startwert werden 0.
- Auf ein Feld greifst du mit einem **Punkt** zu: `a.x`, `a.y`. Ein Feld benutzt du wie jede andere Variable (Zeile 15).
- Mit `=` wird eine ganze Struktur **kopiert**, alle Felder auf einmal (Zeile 18). `c` ist danach eine eigenständige Kopie. Ändert sich `c.y`, bleibt `b.y` unverändert.
- `sizeof(struct punkt)` ist 8: zwei `int` zu je 4 Bytes.

**Strukturen und Funktionen.** Eine Struktur kann Parameter und Rückgabewert einer Funktion sein. So beschreibt ein einziger Parameter eine ganze Figur:

k13_figur.c

```c
// Strukturen als Parameter und Rueckgabewert
#include "pbc.h"

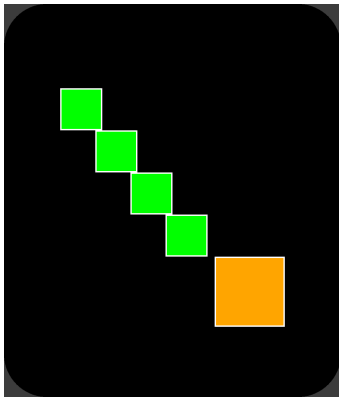
struct figur {
    int x;
    int y;
    int groesse;
    uint16_t farbe;
};

void zeichne_figur(struct figur f) {
    pbc_fill_rect(f.x, f.y, f.groesse, f.groesse, f.farbe);
    pbc_rect(f.x, f.y, f.groesse, f.groesse, PBC_WHITE);
}

struct figur verschiebe(struct figur f, int dx, int dy) {
    f.x += dx;
    f.y += dy;
    return f;
}

int main(void) {
    pbc_init();
    struct figur held = {40, 60, 30, PBC_GREEN};
    struct figur kiste = {150, 180, 50, PBC_ORANGE};
    pbc_clear(PBC_BLACK);
    for (int i = 0; i < 4; i++) {
        zeichne_figur(held);
```

```
29          held = verschiebe(held, 25, 30);
30      }
31      zeichne_figur(kiste);
32      pbc_show();
33      while (true) {
34          sleep_ms(1000);
35      }
36  }
```



- `struct figur` hat vier Felder: Position, Größe und Farbe. Die Felder dürfen verschiedene Typen haben.
- `zeichne_figur` (Zeile 11) bekommt eine ganze Figur und liest daraus alles, was sie zum Zeichnen braucht.
- Wie jeder Parameter ist auch eine Struktur eine **Kopie** (Kapitel 11). `verschiebe` ändert deshalb nur ihre Kopie `f`. Damit die Änderung ankommt, gibt sie die veränderte Figur zurück, und Zeile 29 speichert sie wieder in `held`.
- Die Schleife zeichnet den Helden viermal, jedes Mal ein Stück weiter rechts unten.

**Strukturen nicht mit `==` vergleichen.** Zuweisen geht mit `=`, vergleichen aber nicht. Der Compiler lehnt es ab:

k13_vergleich_falsch.c

```
11      struct punkt a = {1, 2};
12      struct punkt b = {1, 2};
13      if (a == b) {
14          printf("gleich\n");
15      }
```

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:13:11: error: invalid operands to binary == (have 'struct punkt' and
  'struct punkt')
   13 |     if (a == b) {
      |           ^~
```

*Invalid operands to binary ==*: `==` kann mit zwei Strukturen nichts anfangen. Vergleiche die Felder einzeln: `a.x == b.x && a.y == b.y`.

## 13.2 Arrays von Strukturen: viele Gegner

Richtig nützlich werden Strukturen im Array. Statt vier Arrays für `x`, `y`, Tempo und Größe gibt es ein einziges Array, in dem jedes Element ein ganzer Meteor ist:

k13_meteore.c

```c
// Ein Array von Strukturen: viele Meteore fallen
#include "pbc.h"

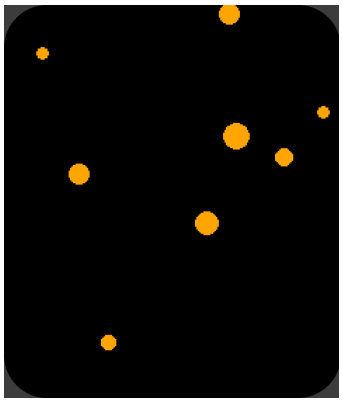
#define ANZAHL 8

struct meteor {
    int x;
    int y;
    int tempo;                    // Pixel pro Bild
    int radius;
};

struct meteor meteore[ANZAHL];

void neu_oben(int i) {
    meteore[i].x = pbc_random(10, PBC_WIDTH - 10);
    meteore[i].y = pbc_random(-200, 0);
    meteore[i].tempo = pbc_random(2, 6);
    meteore[i].radius = pbc_random(4, 10);
}

int main(void) {
    pbc_init();
    for (int i = 0; i < ANZAHL; i++) {
        neu_oben(i);
    }
    while (true) {
        pbc_clear(PBC_BLACK);
        for (int i = 0; i < ANZAHL; i++) {
            meteore[i].y += meteore[i].tempo;
            if (meteore[i].y - meteore[i].radius > PBC_HEIGHT) {
                neu_oben(i);      // unten hinaus: oben neu starten
            }
            pbc_fill_circle(meteore[i].x, meteore[i].y, meteore[i].radius,
              PBC_ORANGE);
        }
        pbc_show();
        sleep_ms(20);
    }
}
```



- Zeile 13 legt das Array `meteore` mit acht Meteoren an. Es steht global, damit `neu_oben` und `main` beide darauf zugreifen können.
- `meteore[i].y` liest sich von links nach rechts: im Array `meteore` das Element `i`, davon das Feld `y`.
- `neu_oben` (Zeile 15) gibt einem Meteor eine zufällige Position oberhalb des Bildschirms, ein zufälliges Tempo und eine zufällige Größe. Weil `y` zwischen −200 und 0 liegt, kommen die Meteore nicht alle gleichzeitig ins Bild.
- In jedem Bild bewegt die Schleife ab Zeile 29 jeden Meteor um sein eigenes Tempo nach unten. Ist er unten ganz hinaus (Zeile 31), startet er oben neu.

Willst du zwanzig Meteore? Dann änderst du nur Zeile 4. Alles andere richtet sich nach `ANZAHL`.

> Lege Arrays mit vielen Strukturen global an, wie hier `meteore`. Lokale Variablen liegen in einem kleinen Speicherbereich von nur 4 KB (Kapitel 12.5); ein großes Array von Strukturen passt dort schnell nicht mehr hinein.

## 13.3 `typedef` und `enum`

**`typedef`: ein kurzer Name.** Überall `struct punkt` zu schreiben, ist umständlich. Mit `typedef` gibst du einem Typ einen eigenen Namen:

```
typedef struct {
    int x;
    int y;
} punkt;
```

Danach heißt der Typ einfach `punkt`, und du schreibst `punkt a = {10, 20};` ohne `struct`. Die Bibliothek macht es genauso: `pbc_image` in `pbc.h` ist eine Struktur mit `typedef` (Kapitel 15).

**`enum`: Zustände mit Namen.** Oft hat eine Variable nur wenige mögliche Werte, die etwas bedeuten: eine Richtung, eine Ampelphase, der Zustand eines Spiels. Man könnte dafür 0, 1, 2, 3 nehmen, aber was bedeutet `richtung = 2`? Ein **`enum`** (*enumeration*, Aufzählung) gibt den Werten Namen:

```
typedef enum { HOCH, RUNTER, LINKS, RECHTS } richtung;
```

`HOCH`, `RUNTER`, `LINKS` und `RECHTS` sind jetzt Konstanten. Intern sind es ganze Zahlen, der Reihe nach 0, 1, 2 und 3, aber du benutzt nur die Namen. Wie bei `#define` schreibt man sie groß. Eine Variable vom Typ `richtung` soll einen dieser Werte enthalten.

Das folgende Programm benutzt beides. Eine Linie läuft immer weiter in ihre Richtung, der Joystick ändert die Richtung:

k13_richtung.c

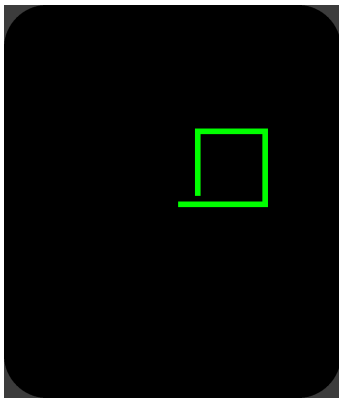
```c
// typedef und enum: eine Linie, die in ihre Richtung weiterlaeuft
#include "pbc.h"

typedef enum { HOCH, RUNTER, LINKS, RECHTS } richtung;

typedef struct {
    int x;
    int y;
} punkt;

punkt schritt(punkt p, richtung r) {
    switch (r) {
        case HOCH:
            p.y -= 4;
            break;
        case RUNTER:
            p.y += 4;
            break;
        case LINKS:
            p.x -= 4;
            break;
        case RECHTS:
            p.x += 4;
            break;
    }
    return p;
}
```

- `schritt` bekommt einen Punkt und eine Richtung und gibt den Punkt einen Schritt weiter zurück.
- Ein `enum` passt perfekt zu `switch` (Kapitel 8): Für jeden Namen gibt es ein `case`.



k13_richtung.c

```c
int main(void) {
    pbc_init();
    punkt kopf = {120, 140};
    richtung r = RECHTS;
    pbc_clear(PBC_BLACK);
```

```
34      while (true) {
35          if (pbc_pressed_up()) {
36              r = HOCH;
37          }
38          if (pbc_pressed_down()) {
39              r = RUNTER;
40          }
41          if (pbc_pressed_left()) {
42              r = LINKS;
43          }
44          if (pbc_pressed_right()) {
45              r = RECHTS;
46          }
47
48          kopf = schritt(kopf, r);
49          // am Rand auf der anderen Seite wieder hereinkommen
50          kopf.x = (kopf.x + PBC_WIDTH) % PBC_WIDTH;
51          kopf.y = (kopf.y + PBC_HEIGHT) % PBC_HEIGHT;
52
53          pbc_fill_rect(kopf.x, kopf.y, 4, 4, PBC_GREEN);
54          pbc_show();
55          sleep_ms(30);
56      }
57  }
```

- `r` merkt sich die Richtung. Der Joystick ändert sie, solange er gedrückt ist. Lässt du los, behält `r` den letzten Wert, und die Linie läuft weiter.
- Die Zeilen 50 und 51 lassen die Linie am Rand verschwinden und auf der anderen Seite wieder auftauchen. Bei `x` = −4 ergibt `(-4 + 240) % 240` = 236, bei `x` = 240 ergibt `480 % 240` = 0.
- Der Bildschirm wird nie gelöscht. Jeder neue Schritt kommt zum Bild dazu, so entsteht die Spur.

**Der Compiler passt auf.** Vergisst du in einem `switch` über einen `enum` einen Fall, warnt der Compiler:

k13_switch_warnung.c

```
4  typedef enum { HOCH, RUNTER, LINKS, RECHTS } richtung;
5
6  int main(void) {
7      pbc_init();
8      richtung r = LINKS;
9      switch (r) {
10         case HOCH:
11             printf("hoch\n");
12             break;
13         case RUNTER:
14             printf("runter\n");
15             break;
16         case RECHTS:
17             printf("rechts\n");
18             break;
19     }
```

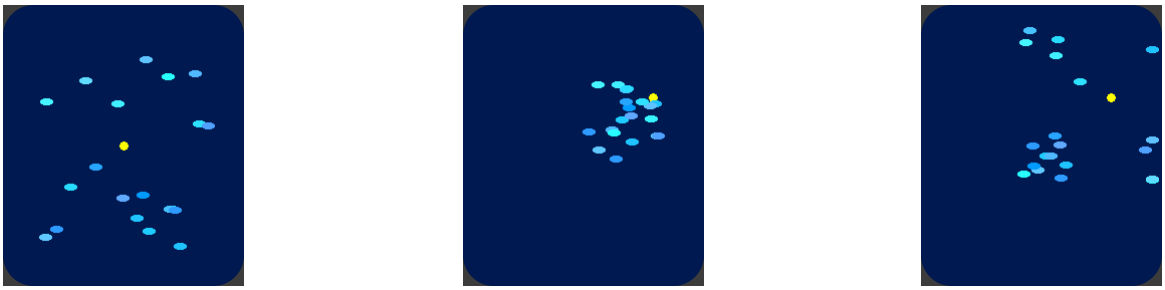
```
Terminal
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
Linking CXX executable k13_switch_warnung.elf
Built target k13_switch_warnung
main.c: In function 'main':
main.c:9:5: warning: enumeration value 'LINKS' not handled in switch [-Wswitch]
    9 |     switch (r) {
      |     ^~~~~~
```

*Enumeration value ‚LINKS' not handled in switch*: Der Wert `LINKS` wird im `switch` nicht behandelt. Mit Zahlen statt Namen könnte der Compiler das nicht bemerken. Auch deshalb lohnt sich ein `enum`.

> Mit `enum` beschreibt man auch die **Zustände eines Spiels**: Titelbild, Spiel läuft, Spiel vorbei. Kapitel 18 baut ein ganzes Spiel auf diese Weise.

## Ausprobieren: Ein Schwarm

Zwanzig Fische folgen dem gelben Futter, das du mit dem Joystick bewegst. Jeder Fisch hat sein eigenes Tempo und seinen Lieblingsplatz neben dem Futter. Ein Druck auf B erschreckt den Schwarm: Die Fische fliehen, bis du B noch einmal drückst.



Links der Start: Die Fische sind zufällig verteilt. In der Mitte sind sie dem Futter nach rechts oben gefolgt. Rechts sind sie nach einem Druck auf B vor dem Futter geflohen.

k13_schwarm.c

```c
// Ein Schwarm: 20 Fische folgen dem Futter, das du mit dem Joystick bewegst.
// B schreckt sie auf: Dann fliehen sie, bis du B noch einmal drueckst.
#include "pbc.h"

#define ANZAHL 20

typedef struct {
    float x;
    float y;
    float tempo;                // Anteil des Wegs pro Bild
```

```
11      int dx;                     // sein Lieblingsplatz neben dem Futter
12      int dy;
13      uint16_t farbe;
14  } fisch;
15
16  typedef enum { FOLGEN, FLIEHEN } modus;
17
18  fisch schwarm[ANZAHL];
19
20  float begrenzen(float wert, float min, float max) {
21      if (wert < min) {
22          return min;
23      }
24      if (wert > max) {
25          return max;
26      }
27      return wert;
28  }
29
30  fisch neuer_fisch(void) {
31      fisch f;
32      f.x = pbc_random(20, PBC_WIDTH - 20);
33      f.y = pbc_random(20, PBC_HEIGHT - 20);
34      f.tempo = pbc_random(2, 8) / 100.0f;
35      f.dx = pbc_random(-30, 30);
36      f.dy = pbc_random(-30, 30);
37      f.farbe = pbc_rgb(pbc_random(0, 100), pbc_random(150, 255), 255);
38      return f;
39  }
```

- Ein `fisch` (Zeilen 7 bis 14) hat eine Position, ein Tempo, einen Lieblingsplatz `dx`, `dy` relativ zum Futter und eine Farbe. Position und Tempo sind `float`, damit ein Fisch sich auch um Bruchteile eines Pixels bewegen kann. Erst beim Zeichnen wird auf ganze Pixel umgewandelt.
- `modus` (Zeile 16) hat zwei Werte: `FOLGEN` und `FLIEHEN`.
- `begrenzen` sorgt dafür, dass ein Wert zwischen `min` und `max` bleibt.
- `neuer_fisch` gibt eine fertig ausgewürfelte Struktur zurück. Das Tempo ist 0,02 bis 0,08: `pbc_random(2, 8)` liefert eine ganze Zahl, `/ 100.0f` macht daraus eine Kommazahl.

k13_schwarm.c

```
41  fisch schwimme(fisch f, int ziel_x, int ziel_y, modus m) {
42      if (m == FOLGEN) {
43          // jedes Bild ein Stueck des Wegs zum Lieblingsplatz
44          f.x += (ziel_x + f.dx - f.x) * f.tempo;
45          f.y += (ziel_y + f.dy - f.y) * f.tempo;
46      } else {
47          // vom Futter weg, schnelle Fische schneller
48          float schritt = f.tempo * 50;
49          if (f.x < ziel_x) {
50              f.x -= schritt;
```

```
51          } else {
52              f.x += schritt;
53          }
54          if (f.y < ziel_y) {
55              f.y -= schritt;
56          } else {
57              f.y += schritt;
58          }
59      }
60      f.x = begrenzen(f.x + pbc_random(-1, 1), 10, PBC_WIDTH - 10);
61      f.y = begrenzen(f.y + pbc_random(-1, 1), 10, PBC_HEIGHT - 10);
62      return f;
63  }
```

- Beim Folgen schwimmt jeder Fisch in jedem Bild einen festen **Anteil** des Wegs zu seinem Lieblingsplatz: ein Fisch mit Tempo 0,05 jeweils 5 %. Ist der Weg weit, ist der Schritt groß; kurz vor dem Ziel wird der Fisch langsam. Weil jeder Fisch einen anderen Lieblingsplatz hat, bleibt der Schwarm eine Wolke und wird kein einzelner Punkt.
- Beim Fliehen schwimmt er mit festem Schritt vom Futter weg, schnelle Fische mit größerem Schritt.
- Zeile 60 und 61 geben ein kleines Zittern dazu (−1 bis +1 Pixel) und halten den Fisch auf dem Bildschirm.

k13_schwarm.c

```
65  int main(void) {
66      pbc_init();
67      for (int i = 0; i < ANZAHL; i++) {
68          schwarm[i] = neuer_fisch();
69      }
70      int futter_x = 120;
71      int futter_y = 140;
72      modus m = FOLGEN;
73      while (true) {
74          // Lesen
75          if (pbc_pressed_left() && futter_x > 10) {
76              futter_x -= 3;
77          }
78          if (pbc_pressed_right() && futter_x < PBC_WIDTH - 10) {
79              futter_x += 3;
80          }
81          if (pbc_pressed_up() && futter_y > 10) {
82              futter_y -= 3;
83          }
84          if (pbc_pressed_down() && futter_y < PBC_HEIGHT - 10) {
85              futter_y += 3;
86          }
87          if (pbc_was_pressed_b()) {
88              if (m == FOLGEN) {
89                  m = FLIEHEN;
```

```
90              } else {
91                  m = FOLGEN;
92              }
93          }
94
95          // Rechnen
96          for (int i = 0; i < ANZAHL; i++) {
97              schwarm[i] = schwimme(schwarm[i], futter_x, futter_y, m);
98          }
99
100         // Zeichnen und warten
101         pbc_clear(pbc_rgb(0, 30, 80));
102         pbc_fill_circle(futter_x, futter_y, 4, PBC_YELLOW);
103         for (int i = 0; i < ANZAHL; i++) {
104             pbc_fill_ellipse((int)schwarm[i].x, (int)schwarm[i].y, 6, 3,
                  schwarm[i].farbe);
105         }
106         pbc_show();
107         sleep_ms(20);
108     }
109 }
```

`main` ist die bekannte Spielschleife: lesen, rechnen, zeichnen, warten. Die Schleife in Zeile 96 lässt jeden Fisch schwimmen und speichert das Ergebnis zurück ins Array. Die Schleife in Zeile 103 zeichnet jeden Fisch als kleine Ellipse.

**Zum Verändern:** Mach aus 20 Fischen 50 (eine einzige Änderung). Gib jedem Fisch ein Feld `groesse` und zeichne große und kleine Fische. Füge einen dritten Modus `WARTEN` hinzu, in dem die Fische nur zittern und an ihrer Stelle bleiben. Schreibe `schwimme` dafür mit `switch` statt `if`: Dann erinnert dich der Compiler an jeden Modus, den du vergisst.

## Übungsaufgaben

**Aufgabe 1 ⭐ Was gibt das Programm aus?** Lies das Programm, ohne es aufzuspielen. Welche vier Zeilen erscheinen in der Konsole? Achte darauf, wo kopiert wird.

k13_aufgabe1.c

```c
// Was gibt das Programm aus?
#include "pbc.h"

struct punkt {
    int x;
    int y;
};

struct punkt nach_rechts(struct punkt p) {
    p.x += 10;
```

```c
    return p;
}

void nach_unten(struct punkt p) {
    p.y += 10;
    printf("in nach_unten: (%d, %d)\n", p.x, p.y);
}

int main(void) {
    pbc_init();
    struct punkt a = {1, 2};
    struct punkt b = a;
    b.x = 5;
    nach_unten(a);
    struct punkt c = nach_rechts(b);
    printf("a = (%d, %d)\n", a.x, a.y);
    printf("b = (%d, %d)\n", b.x, b.y);
    printf("c = (%d, %d)\n", c.x, c.y);
    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 2 ⭐ Eine Spielertabelle** Lege mit `typedef` eine Struktur `spieler` an, mit Name (Text), Punkten und Level. Lege ein Array mit vier Spielern an und gib sie als Tabelle in der Konsole aus, die Spalten sauber untereinander. Gib danach aus, wer die meisten Punkte hat.

**Aufgabe 3 ⭐⭐ Finde die Fehler** Das Programm soll ein blaues Rechteck zeichnen. Der Compiler meldet sehr viele Fehler. Wie viele Fehler sind es wirklich? Verbessere sie.

k13_aufgabe3.c

```c
// Ein Rechteck als Struktur - mit Fehlern
#include "pbc.h"

struct rechteck {
    int x;
    int y;
    int breite;
    int hoehe;
}

int main(void) {
    pbc_init();
    rechteck r = {20, 40, 100, 60};
    pbc_fill_rect(r.x, r.y, r.Breite, r.hoehe, PBC_BLUE);
    pbc_show();
```

```
    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 4 ⭐⭐ Ampel mit `enum`** Beschreibe die vier Phasen einer Ampel (Rot, Rot-Gelb, Grün, Gelb) mit einem `enum`. Eine Funktion `naechste` bekommt eine Phase und liefert die nächste. Jeder Druck auf A schaltet weiter. Zeige die Ampel auf dem Bildschirm und mit den drei Status-LEDs.

**Aufgabe 5 ⭐⭐⭐ Bestenliste** Eine Bestenliste hat fünf Plätze, jeder mit Name und Punkten. Sortiere sie absteigend nach Punkten. Danach fragt das Programm immer wieder nach Name und Punkten. Ist das neue Ergebnis besser als der letzte Platz, kommt es an die richtige Stelle in die Liste, und der bisher Letzte fällt heraus. Tipp: Zwei Einträge tauschst du mit einer dritten Struktur als Zwischenspeicher.

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- `struct name { typ feld; … };` beschreibt eine Struktur (Semikolon am Ende!). Auf ein Feld greifst du mit `.` zu: `a.x`.
- `=` kopiert eine ganze Struktur. Als Parameter ist sie eine Kopie; eine Funktion gibt die geänderte Struktur mit `return` zurück. `==` funktioniert für Strukturen nicht.
- Ein Array von Strukturen: `meteore[i].y` ist das Feld `y` des Elements `i`. Große Arrays global anlegen.
- `typedef struct { … } punkt;` gibt dem Typ einen kurzen Namen.
- `typedef enum { HOCH, RUNTER, … } richtung;` benennt Zustände. Mit `switch` warnt der Compiler, wenn ein Fall fehlt.

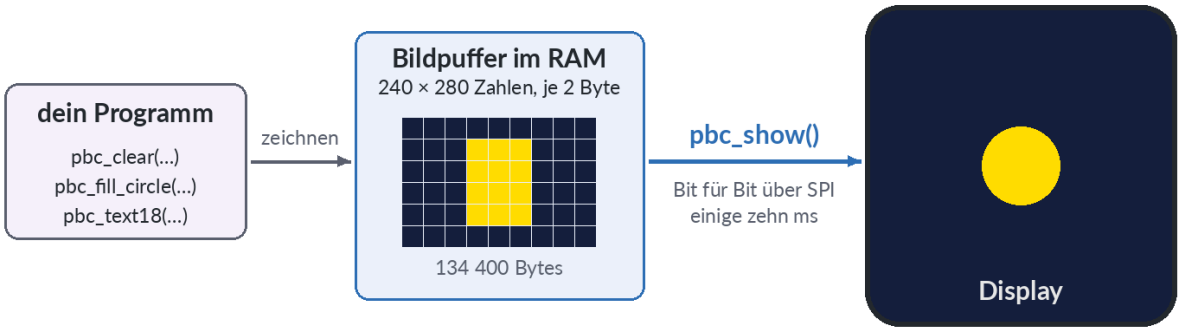
KAPITEL 14

# Flimmerfrei animieren

**Worum geht's?** In Teil III baust du Spiele. Dafür muss sich etwas gleichmäßig und ohne Flackern bewegen. Du erfährst, was beim Zeichnen im Speicher passiert, wie lange `pbc_show()` braucht und wie du eine feste Bildrate einhältst. Zum Schluss bekommt ein Ball eine Geschwindigkeit, prallt an Wänden ab und fällt mit Schwerkraft.

## 14.1 Der Bildpuffer: erst zeichnen, dann zeigen

Seit Kapitel 3 weißt du: Alles Gezeichnete erscheint erst nach `pbc_show()`. Jetzt sehen wir uns genauer an, warum das so ist.




Die Buchbibliothek hält im Arbeitsspeicher (RAM) ein Array mit einer Zahl für jedes Pixel des Bildschirms. Das ist der **Bildpuffer**. Jede Zahl ist eine Farbe im Format RGB565 (Kapitel 6), also ein `uint16_t` mit 2 Bytes. Befehle wie `pbc_clear`, `pbc_fill_circle` oder `pbc_text18` ändern nur Zahlen in diesem Array. Der Bildschirm merkt davon nichts. Erst `pbc_show()` schickt das ganze Array zum Display. Das Display hat einen eigenen Speicher, aus dem es das Bild immer wieder anzeigt, bis das nächste kommt.

Wie groß ist der Bildpuffer? Das rechnet der PicoBoy selbst aus:

k14_puffer_groesse.c

```c
// Wie viel Speicher braucht der Bildpuffer?
#include "pbc.h"

int main(void) {
    pbc_init();
    int pixel = PBC_WIDTH * PBC_HEIGHT;
    int bytes = pixel * (int)sizeof(uint16_t);

    printf("Pixel:          %d\n", pixel);
    printf("Bytes je Pixel: %d\n", (int)sizeof(uint16_t));
```

```
11      printf("Bildpuffer:     %d Bytes\n", bytes);
12      printf("                %d Bits\n", bytes * 8);
13
14      while (true) {
15          sleep_ms(1000);
16      }
17  }
```

Serial Monitor

```
Pixel:          67200
Bytes je Pixel: 2
Bildpuffer:     134400 Bytes
                1075200 Bits
```

134 400 Bytes, rund 134 KB. Der Arbeitsspeicher des RP2350 ist gut eine halbe Million Bytes groß, der Bildpuffer belegt also ungefähr ein Viertel davon. Wie der Speicher aufgeteilt ist, zeigt Kapitel 19.

Über eine Million Bits wandern bei jedem `pbc_show()` Bit für Bit über eine Leitung zum Display (wie genau, erklärt Kapitel 23). Das dauert einige **zehn Millisekunden**: Rechnet man mit dem Takt, den das Pico SDK für diese Leitung einstellt (37,5 Millionen Bits pro Sekunde), kommt man auf knapp 29 ms. Gemessen sind es auf dem PicoBoy 31,4 ms, etwas mehr als gerechnet; die Rechnung zählt nur die reinen Datenbits. Das misst du in Abschnitt 14.2 selbst. Während dieser Zeit wartet dein Programm. Mehr als etwa 1000 / 31,4 ≈ 32 Bilder pro Sekunde sind also nicht möglich, und jedes `pbc_show()`, das du dir sparst, ist gewonnene Zeit.

**Warum der Umweg?** Stell dir vor, jeder Befehl würde sofort auf dem Display zeichnen. Für ein bewegtes Bild musst du immer erst alles löschen und dann neu zeichnen. Das Auge sähe zwischendurch das gelöschte, leere Bild: Der Bildschirm **flackert**. Genau diesen Fehler kannst du mit der Bibliothek nachbauen, indem du das leere Bild selbst zeigst:

k14_flackern.c

```c
// So flackert es: zwei pbc_show() fuer jedes Bild
#include "pbc.h"

int main(void) {
    pbc_init();
    int x = 30;
    while (true) {
        pbc_clear(PBC_BLACK);
        pbc_show();                          // zeigt ein leeres Bild!
        pbc_fill_circle(x, 140, 20, PBC_YELLOW);
        pbc_show();                          // zeigt den Ball

        x = x + 2;
        if (x > 210) {
            x = 30;
        }
```

```
        sleep_ms(10);
    }
}
```

Zeile 9 schickt nach dem Löschen ein schwarzes Bild zum Display, Zeile 11 das Bild mit dem Ball. Auf dem PicoBoy ist der Ball in jedem Durchgang so lange nicht zu sehen, wie ein `pbc_show()` dauert. Er flackert deutlich. Lässt du Zeile 9 weg, läuft er ruhig:

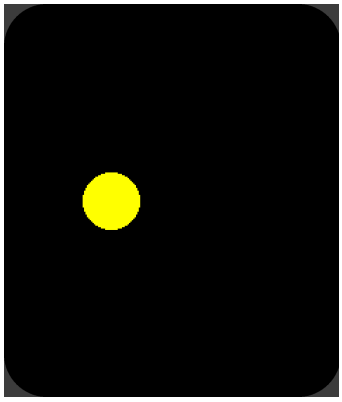
k14_ohne_flackern.c

```c
// Richtig: erst das ganze Bild zeichnen, dann einmal pbc_show()
#include "pbc.h"

int main(void) {
    pbc_init();
    int x = 30;
    while (true) {
        pbc_clear(PBC_BLACK);
        pbc_fill_circle(x, 140, 20, PBC_YELLOW);
        pbc_show();

        x = x + 2;
        if (x > 210) {
            x = 30;
        }
        sleep_ms(10);
    }
}
```

So sieht jede Animation in diesem Buch aus:



1. löschen (`pbc_clear`),
2. das ganze Bild zeichnen,
3. **einmal** `pbc_show()`,
4. warten bis zum nächsten Bild.

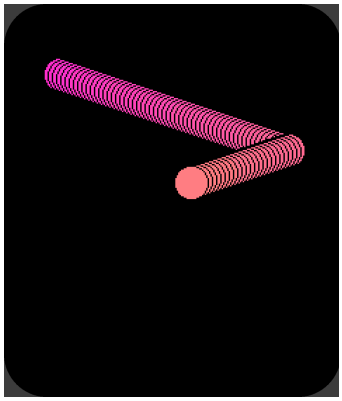
Das Display bekommt so nur fertige Bilder. Diese Technik heißt **Doppelpufferung** (englisch *double buffering*): Ein Bild ist zu sehen, während das nächste unsichtbar im Speicher entsteht.

> Pro Bild genau **ein** `pbc_show()`, und zwar ganz am Ende, wenn alles gezeichnet ist.

**Der Bildpuffer vergisst nichts.** `pbc_show()` löscht den Bildpuffer nicht. Was du einmal gezeichnet hast, bleibt darin, bis du es übermalst. Lässt du `pbc_clear` weg, hinterlässt ein bewegter Ball eine Spur:

k14_spur.c

```
1  // Ohne pbc_clear bleibt alles Alte im Bildpuffer stehen
2  #include "pbc.h"
3
4  int main(void) {
5      pbc_init();
6      pbc_clear(PBC_BLACK);
7      int x = 40;
8      int y = 50;
9      int dx = 3;
10     while (true) {
11         uint16_t farbe = pbc_rgb(255, y, 255 - y);
12         pbc_fill_circle(x, y, 12, farbe);
13         pbc_circle(x, y, 12, PBC_BLACK);
14         pbc_show();
15
16         x = x + dx;
17         y = y + 1;
18         if (x > 200 || x < 40) {
19             dx = -dx;                // am Rand umkehren
20         }
21         if (y > 230) {
22             y = 50;
23         }
24         sleep_ms(20);
25     }
26 }
```



- Zeile 6 löscht den Bildschirm nur einmal, vor der Schleife.
- Zeile 11 mischt die Farbe aus der y-Position. Je weiter unten der Ball, desto mehr Grün und desto weniger Blau.
- Zeile 13 zeichnet einen schwarzen Rand um jeden Kreis. So sieht man die einzelnen Bilder der Spur.

## 14.2 Feste Bildrate, Zeit pro Bild messen

Wie lange etwas dauert, misst du mit einer Uhr. `pbc_millis()` aus Kapitel 7 zählt Millisekunden. Für kurze Zeiten ist das zu grob. Das Pico SDK hat dafür `time_us_32()`: Es liefert die **Mikrosekunden** (µs) seit dem Start als `uint32_t`. Eine Mikrosekunde ist eine millionstel Sekunde, 1000 µs sind 1 ms.

Zum Messen merkst du dir die Uhrzeit vorher und nachher und ziehst ab:

k14_show_messen.c

```c
// Misst, wie lange pbc_show() dauert
#include "pbc.h"

int main(void) {
    pbc_init();
    for (int i = 1; i <= 5; i++) {
        pbc_clear(PBC_BLUE);
        uint32_t vorher = time_us_32();
        pbc_show();
        uint32_t nachher = time_us_32();
        int dauer = (int)(nachher - vorher);
        printf("pbc_show Nr. %d: %d us\n", i, dauer);
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Spiel das Programm auf und öffne den seriellen Monitor (Kapitel 4). Er zeigt fünf Zeilen der Form `pbc_show Nr. 1: … us`. Nach der Rechnung aus 14.1 sollten es knapp 29 000 µs sein. Notiere deinen Wert, du brauchst ihn gleich.

> Die Ausgaben in diesem Kapitel stammen aus den Prüfläufen der Programme auf dem PC. Die PC-Fassung der Bibliothek rechnet für jedes `pbc_show()` 31,4 ms, wie auf dem PicoBoy gemessen; das Zeichnen kostet dort aber gar keine Zeit. Auf dem PicoBoy sind manche Zahlen deshalb etwas größer. Am Prinzip ändert das nichts.

Die Differenz `nachher - vorher` rechnest du mit den `uint32_t`-Werten. Das ist auch dann richtig, wenn die Uhr zwischendurch übergelaufen ist: Nach etwa 71 Minuten springt `time_us_32()` von 4 294 967 295 wieder auf 0 (Kapitel 5.2). Die Subtraktion von vorzeichenlosen Zahlen „läuft dann genauso über“ und liefert trotzdem den richtigen Abstand. Erst das Ergebnis machen wir mit `(int)` zu einer gewöhnlichen ganzen Zahl für `printf`.

> Auch das Zeichnen selbst kostet Zeit. Ein `pbc_clear` beschreibt alle 67 200 Pixel, große gefüllte Kreise und viel Text brauchen ebenfalls messbar lange. Setze die Messung um deine Zeichenbefehle herum, um herauszufinden, wie lange dein Programm für ein Bild rechnet.

**Das Problem mit der festen Pause.** Bisher hast du nach jedem Bild eine feste Zeit gewartet, etwa mit `sleep_ms(20)`. Dann dauert ein Bild aber so lange wie Zeichnen, `pbc_show()` **und** Pause zusammen. Muss das Programm einmal mehr zeichnen, wird das Bild länger, und die Bewegung ruckelt. Das folgende Programm zeigt den Effekt: Ab dem vierten Bild hat es mehr zu tun (hier ein zweites `pbc_show()` als Ersatz für viel Zeichenarbeit).

k14_feste_pause.c

```c
// Feste Pause: Wie lange ein Bild dauert, haengt vom Zeichnen ab
#include "pbc.h"

int main(void) {
    pbc_init();
    uint32_t letztes = pbc_millis();
    for (int bild = 1; bild <= 6; bild++) {
        pbc_clear(PBC_BLACK);
        pbc_show();
        if (bild > 3) {
            pbc_show();              // ab Bild 4: doppelt so viel Arbeit
        }
        sleep_ms(23);

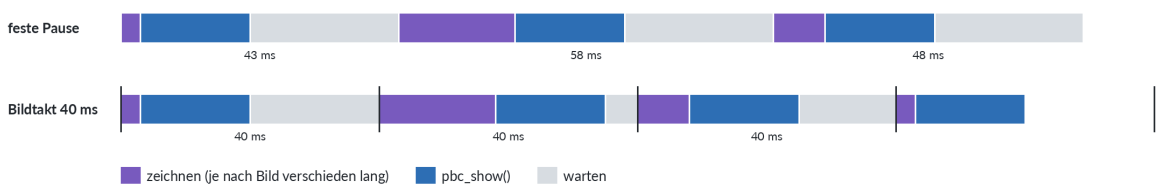
        uint32_t jetzt = pbc_millis();
        printf("Bild %d: %d ms\n", bild, (int)(jetzt - letztes));
        letztes = jetzt;
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Bild 1: 54 ms
Bild 2: 55 ms
Bild 3: 54 ms
Bild 4: 86 ms
Bild 5: 86 ms
Bild 6: 86 ms
```

Im Prüflauf dauert ein Bild erst 54 ms, ab Bild 4 dann 86 ms. Auf dem Gerät sind die Zahlen etwas anders, der Sprung bleibt.

**Die Lösung: ein Bildtakt.** Statt immer gleich lange zu warten, wartest du immer bis zu einem festen Zeitpunkt. Jedes Bild bekommt ein festes Zeitfenster, zum Beispiel 40 ms. Was nach Zeichnen und `pbc_show()` von diesem Fenster übrig ist, wird gewartet:




40 ms je Bild ergeben 1000 / 40 = **25 Bilder pro Sekunde**. Das ist ungefähr die Bildrate eines Kinofilms; Bewegungen sehen damit flüssig aus. Die Bildrate gibt man in **Bildern pro Sekunde** an, englisch *frames per second*, kurz **fps**.

k14_bildtakt.c

```c
// Feste Bildrate: jedes Bild dauert 40 ms, also 25 Bilder pro Sekunde
#include "pbc.h"

#define BILD_MS 40

int main(void) {
    pbc_init();
    uint32_t naechstes_bild = pbc_millis();
    int x = 30;

    while (true) {
        pbc_clear(PBC_BLACK);
        pbc_fill_circle(x, 140, 20, PBC_YELLOW);
        pbc_show();

        x = x + 3;
        if (x > 210) {
            x = 30;
        }

        // warten, bis die 40 ms dieses Bildes um sind
        naechstes_bild = naechstes_bild + BILD_MS;
        int rest = (int)(naechstes_bild - pbc_millis());
        if (rest > 0) {
            sleep_ms(rest);
        }
    }
}
```

- Zeile 8 merkt sich, wann das erste Bild beginnt.
- Zeile 22 rechnet aus, wann das nächste Bild beginnen soll: 40 ms nach dem Beginn des vorigen, nicht 40 ms nach jetzt.
- Zeile 23 rechnet, wie viele Millisekunden bis dahin übrig sind. Ist das Programm schon zu spät dran, ist die Differenz negativ. Deshalb wird sie mit `(int)` in eine Zahl mit Vorzeichen umgewandelt, bevor Zeile 24 vergleicht.
- Zeile 25 wartet nur die übrige Zeit.

**Als Funktion.** Diese Rechnung brauchst du in jedem Spiel. Darum steckt sie ab jetzt in einer Funktion `bildtakt()`. Die Funktion merkt sich den Beginn des nächsten Bilds in einer `static`-Variable (Kapitel 11.4): Sie behält ihren Wert von Aufruf zu Aufruf. Das nächste Programm zählt außerdem, wie viele Bilder in jeder Sekunde wirklich gezeigt werden:

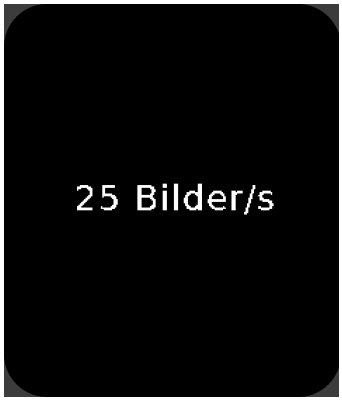
k14_bildrate.c

```c
// Zaehlt, wie viele Bilder pro Sekunde wirklich gezeigt werden
#include "pbc.h"

#define BILD_MS 40
```

```
 6 // Wartet, bis seit dem vorigen Aufruf BILD_MS vergangen sind.
 7 void bildtakt(void) {
 8     static uint32_t naechstes = 0;
 9     uint32_t jetzt = pbc_millis();
10     naechstes = naechstes + BILD_MS;
11     int rest = (int)(naechstes - jetzt);
12     if (rest > 0) {
13         sleep_ms(rest);
14     } else {
15         naechstes = jetzt;      // zu spaet dran: nicht versuchen aufzuholen
16     }
17 }
18
19 int main(void) {
20     pbc_init();
21     int bilder = 0;             // Bilder in der laufenden Sekunde
22     int bildrate = 0;           // Ergebnis der letzten vollen Sekunde
23     uint32_t sekunde_start = pbc_millis();
24     char text[20];
25
26     while (true) {
27         pbc_clear(PBC_BLACK);
28         snprintf(text, sizeof text, "%d Bilder/s", bildrate);
29         pbc_text24(text, 50, 124, PBC_WHITE);
30         pbc_show();
31
32         bilder++;
33         if (pbc_millis() - sekunde_start >= 1000) {
34             bildrate = bilder;
35             bilder = 0;
36             sekunde_start = sekunde_start + 1000;
37         }
38         bildtakt();
39     }
40 }
```

- Zeile 8: `naechstes` beginnt bei 0. Beim ersten Aufruf ist die Uhr schon weiter, `rest` ist negativ, und Zeile 15 setzt `naechstes` auf jetzt. Ab dann läuft der Takt.
- Zeile 15 sorgt auch dafür, dass ein zu langsames Bild nicht durch mehrere zu kurze Bilder „aufgeholt“ wird. Die Bewegung würde sonst ruckartig schneller.
- Zeilen 32–37 zählen die Bilder. Ist eine Sekunde um, wird das Ergebnis in `bildrate` gespeichert und die Zählung beginnt von vorn.




> Kopiere `bildtakt()` mit `#define BILD_MS 40` in deine Spiele und rufe die Funktion am Ende jedes Schleifendurchgangs auf, direkt nach `pbc_show()`.

## 14.3 Bewegung mit Geschwindigkeit, Abprallen

Bisher hast du eine Figur bewegt, indem du `x` um einen festen Wert erhöht hast. Für schräge Bewegungen und Abpraller ist es besser, wenn sich ein Objekt seine **Geschwindigkeit** merkt: wie viele Pixel es pro Bild in x- und in y-Richtung weiterrückt. Man nennt diese Werte meist `vx` und `vy` (*v* für *velocity*, Geschwindigkeit). Eine negative Geschwindigkeit bedeutet: nach links beziehungsweise nach oben.

In jedem Bild passiert dann:

```
ball.x = ball.x + ball.vx;
ball.y = ball.y + ball.vy;
```

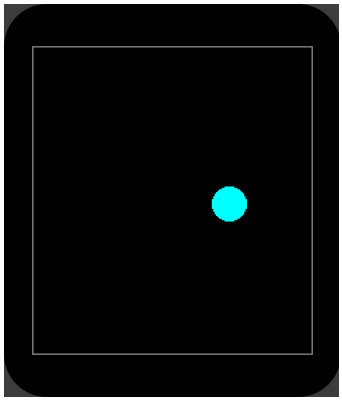
Position und Geschwindigkeit gehören zusammen, also bündeln wir sie in einer Struktur (Kapitel 13). Wir nehmen `float`, damit auch langsame Bewegungen wie 0,5 Pixel pro Bild möglich sind. Erst beim Zeichnen wird mit `(int)` auf ganze Pixel umgewandelt.

**Abprallen** heißt: Stößt der Ball an eine Wand, dreht sich eine Richtung um. Trifft er die linke oder rechte Wand, wird `vx` zu `-vx`, bei oben oder unten `vy` zu `-vy`.

k14_abprallen.c

```
28 int main(void) {
29     pbc_init();
30     kugel ball = { 60.0f, 80.0f, 4.0f, 2.5f };
31
32     while (true) {
33         ball.x = ball.x + ball.vx;          // bewegen
34         ball.y = ball.y + ball.vy;
35
36         if (ball.x - RADIUS < LINKS) {      // links anstossen
37             ball.x = LINKS + RADIUS;
38             ball.vx = -ball.vx;
39         }
40         if (ball.x + RADIUS > RECHTS) {     // rechts anstossen
41             ball.x = RECHTS - RADIUS;
42             ball.vx = -ball.vx;
43         }
44         if (ball.y - RADIUS < OBEN) {
45             ball.y = OBEN + RADIUS;
46             ball.vy = -ball.vy;
47         }
48         if (ball.y + RADIUS > UNTEN) {
49             ball.y = UNTEN - RADIUS;
50             ball.vy = -ball.vy;
51         }
52
53         pbc_clear(PBC_BLACK);
54         pbc_rect(LINKS, OBEN, RECHTS - LINKS, UNTEN - OBEN, PBC_GRAY);
55         pbc_fill_circle((int)ball.x, (int)ball.y, RADIUS, PBC_CYAN);
56         pbc_show();
```

```
57          bildtakt();
58      }
59  }
```



- Zeile 30 legt den Ball an: Mittelpunkt (60, 80), Geschwindigkeit 4 Pixel nach rechts und 2,5 Pixel nach unten pro Bild. Das `f` hinter den Zahlen macht sie zu `float`-Werten.
- Zeilen 36–39: Der linke Rand des Balls liegt bei `ball.x - RADIUS`. Ist er links von der Wand, prallt der Ball ab.
- Zeile 37 ist wichtig: Der Ball wird **zurück an die Wand gesetzt**. Ohne diese Zeile könnte er ein Stück in der Wand stecken. Im nächsten Bild wäre er dann immer noch „in der Wand“, die Richtung würde wieder umgedreht, und der Ball zappelte auf der Stelle.
- Zeile 54 zeichnet das Spielfeld als grauen Rahmen. Es hält Abstand zu den runden Ecken des Displays.

Die Zeilen 1–27 kennst du: die Konstanten, die Struktur `kugel` mit `x`, `y`, `vx`, `vy` und die Funktion `bildtakt()`.

**Schwerkraft.** Ein geworfener Ball wird nach unten immer schneller. Dafür ändert sich in jedem Bild die Geschwindigkeit selbst: `vy` wird um einen kleinen Wert größer. Aus Position, Geschwindigkeit und dieser **Beschleunigung** entsteht eine Wurfparabel, ohne dass du sie ausrechnen musst. Genau das zeigt das folgende Programm.

## Ausprobieren: Ein springender Ball

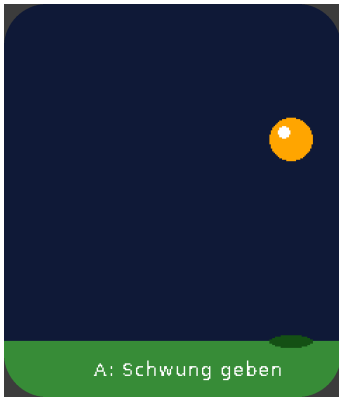
k14_springender_ball.c

```c
int main(void) {
    pbc_init();
    kugel ball = { 60.0f, 60.0f, 2.0f, 0.0f };

    while (true) {
        if (pbc_was_pressed_a()) {
            ball.vy = -14.0f;                   // kraeftig nach oben
        }
        ball.vy = ball.vy + SCHWERKRAFT;
        ball.x = ball.x + ball.vx;
        ball.y = ball.y + ball.vy;

        if (ball.y + RADIUS > BODEN) {          // Aufprall am Boden
            ball.y = BODEN - RADIUS;
            if (ball.vy > 3.0f) {
                pbc_tone(200, 30);
            }
            ball.vy = -ball.vy * DAEMPFUNG;
```

```
45              if (ball.vy > -1.0f) {
46                  ball.vy = 0.0f;                 // zu schwach: liegen bleiben
47              }
48          }
49          if (ball.x - RADIUS < 20 || ball.x + RADIUS > 220) {
50              ball.vx = -ball.vx;                 // an den Seiten umkehren
51              ball.x = ball.x + ball.vx;
52          }
53
54          pbc_clear(pbc_rgb(20, 30, 60));
55          pbc_fill_rect(0, BODEN, PBC_WIDTH, 40, pbc_rgb(60, 140, 60));
56          pbc_fill_ellipse((int)ball.x, BODEN, RADIUS, 4, pbc_rgb(30, 80, 30));
57          pbc_fill_circle((int)ball.x, (int)ball.y, RADIUS, PBC_ORANGE);
58          pbc_fill_circle((int)ball.x - 5, (int)ball.y - 5, 4, PBC_WHITE);
59          pbc_text12("A: Schwung geben", 64, 252, PBC_WHITE);
60          pbc_show();
61          bildtakt();
62      }
63  }
```




- Zeile 35: die Schwerkraft. In jedem Bild wird `vy` um 0,6 größer. Der Ball wird beim Fallen schneller und beim Steigen langsamer, bis er umkehrt.
- Zeilen 39–48: Trifft der Ball den Boden, kehrt sich `vy` um, aber nur 80 % des Tempos bleiben (`DAEMPFUNG`). Jeder Sprung ist deshalb niedriger als der vorige. Einen kurzen Ton gibt es nur bei einem kräftigen Aufprall (Zeile 41).
- Zeilen 45–47: Ist der Rückprall nur noch ganz schwach, bleibt der Ball liegen. Sonst würde er ewig um ein Pixel auf und ab zittern.
- Zeilen 49–52: An den Seiten kehrt `vx` um. Hier wird der Ball nicht an die Wand gesetzt, sondern mit der neuen Geschwindigkeit gleich einen Schritt zurückbewegt. Auch so kommt er sicher aus der Wand heraus.
- Zeile 32: Taste A setzt `vy` auf −14, ein kräftiger Stoß nach oben.
- Zeile 56: Der Schatten ist eine flache Ellipse am Boden. Er wandert mit `ball.x`, bleibt aber immer unten.

Das Bild rechts zeigt den Ball kurz nach einem Druck auf A.

**Zum Verändern:** Probiere andere Werte für `SCHWERKRAFT` und `DAEMPFUNG`. Wie sieht ein Flummi aus (`DAEMPFUNG` nahe 1), wie ein nasser Sack? Lass den Ball mit dem Joystick nach links und rechts anstoßen, indem du `vx` änderst.

## Übungsaufgaben

**Aufgabe 1 ⭐ Lahmer Sternenhimmel** Dieses Programm soll alle 100 ms einen neuen Sternenhimmel aus 20 Sternen zeigen. Es ist aber viel langsamer, und die Sterne erscheinen einzeln. Warum? Wie lange dauert ein Bild ungefähr? Verbessere das Programm.

k14_aufgabe1.c

```c
// Ein Sternenhimmel, der viel zu langsam blinkt. Warum?
#include "pbc.h"

int main(void) {
    pbc_init();
    while (true) {
        pbc_clear(pbc_rgb(10, 10, 40));
        for (int i = 0; i < 20; i++) {
            int x = pbc_random(30, 210);
            int y = pbc_random(30, 250);
            pbc_fill_circle(x, y, 2, PBC_WHITE);
            pbc_show();
        }
        sleep_ms(100);
    }
}
```

**Aufgabe 2 ⭐ Schneller geht's nicht** Setze in `k14_bildrate.c` `BILD_MS` auf 100, dann auf 10. Welche Bildrate zeigt der PicoBoy jeweils an? Warum erreichst du mit 10 keine 100 Bilder pro Sekunde? Rechne mit deinem Messwert aus 14.2 nach. Was passiert mit dem Ball aus `k14_bildtakt.c`, wenn du dort `BILD_MS` auf 80 setzt?

**Aufgabe 3 ⭐⭐ Bildschirmschoner** Lass den Schriftzug „PicoBoy" (`pbc_text18`) schräg über den Bildschirm gleiten und an allen Rändern abprallen. Bei jedem Abpraller bekommt er eine neue, zufällige Farbe. Die Breite des Texts liefert `pbc_text18_width`.

**Aufgabe 4 ⭐⭐ Zehn Bälle** Lege ein Array aus zehn `kugel`-Strukturen an. Jeder Ball startet an einer zufälligen Stelle mit zufälliger Geschwindigkeit zwischen −4 und 4 Pixeln pro Bild und hat eine eigene Farbe. Alle prallen im Spielfeld ab. Tipp: Nimm die Farbe mit in die Struktur auf.

**Aufgabe 5 ⭐⭐⭐ Pixel pro Sekunde** Bei „Pixel pro Bild“ hängt das Tempo von der Bildrate ab. Besser für Spiele, die unterschiedlich viel zeichnen: Gib die Geschwindigkeit in **Pixeln pro Sekunde** an. Miss in jedem Durchgang mit `time_us_32()`, wie viele Sekunden seit dem vorigen Bild vergangen sind (`dt`), und bewege um `vx * dt`. Lass einen Ball mit 120 Pixeln pro Sekunde hin- und herlaufen. Solange B gedrückt ist, soll das Programm künstlich 80 ms pro Bild „trödeln“ (`sleep_ms(80)`). Läuft der Ball trotzdem gleich schnell?

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- Der **Bildpuffer** ist ein Array mit 240 × 280 Farben zu je 2 Bytes, zusammen 134 400 Bytes im RAM. Zeichenbefehle ändern nur den Bildpuffer.
- `pbc_show()` schickt den ganzen Bildpuffer zum Display und dauert einige zehn Millisekunden (selbst messen). Pro Bild genau **ein** `pbc_show()`, ganz am Ende.
- Animation: löschen – zeichnen – `pbc_show()` – warten. Ohne `pbc_clear` bleibt das alte Bild stehen.
- `time_us_32()` misst Mikrosekunden. Dauer = nachher − vorher.
- Feste Bildrate: bis zu einem festen Zeitpunkt warten statt eine feste Zeit. 40 ms je Bild = 25 Bilder pro Sekunde. Die Funktion `bildtakt()` erledigt das.
- Bewegung: Position += Geschwindigkeit; Schwerkraft: Geschwindigkeit += Beschleunigung. Abprallen: Vorzeichen umdrehen **und** den Ball aus der Wand holen.

KAPITEL 15

# Bilder und Sprites

**Worum geht's?** Kreise und Rechtecke reichen für einen Ball, aber nicht für eine Spielfigur. In diesem Kapitel bringst du kleine Pixelbilder auf den PicoBoy. Du schreibst zuerst ein Bild von Hand als Array, wandelst dann PNG-Dateien mit dem Werkzeug `bild2c.html` um, zeichnest Bilder mit durchsichtigen Stellen und lässt eine Figur aus mehreren Bildern laufen.

## 15.1 Bilder als C-Array: das Umwandlungswerkzeug

Ein Bild besteht aus Pixeln, und jedes Pixel hat eine Farbe. Im Speicher ist ein Bild deshalb einfach eine Reihe von Farbwerten: Zeile für Zeile, in jeder Zeile von links nach rechts. Genauso ist auch der Bildpuffer aufgebaut (Kapitel 14).

**Ein Bild von Hand.** Ein kleines Bild kannst du direkt hinschreiben. Hier ein Herz aus 8 × 8 Pixeln. Damit man das Bild im Quelltext erkennt, stehen für die Farben einbuchstabige Abkürzungen:

k15_herz.c

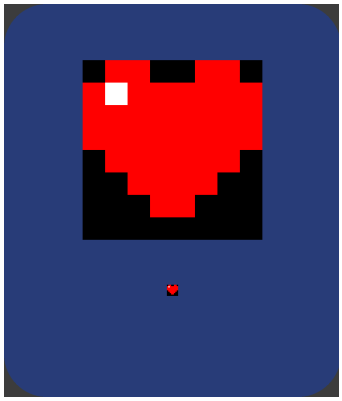
```c
// Ein Bild als Array: ein Herz aus 8 x 8 Pixeln, von Hand geschrieben
#include "pbc.h"

#define S PBC_BLACK
#define R PBC_RED
#define W PBC_WHITE

static const uint16_t herz_pixel[64] = {
    S, R, R, S, S, R, R, S,
    R, W, R, R, R, R, R, R,
    R, R, R, R, R, R, R, R,
    R, R, R, R, R, R, R, R,
    S, R, R, R, R, R, R, S,
    S, S, R, R, R, R, S, S,
    S, S, S, R, R, S, S, S,
    S, S, S, S, S, S, S, S,
};
static const pbc_image herz = { 8, 8, herz_pixel, NULL };

int main(void) {
    pbc_init();
    pbc_clear(pbc_rgb(40, 60, 120));
    // jedes Pixel des Bilds als Quadrat mit 16 Pixeln Kantenlaenge
    for (int zeile = 0; zeile < 8; zeile++) {
        for (int spalte = 0; spalte < 8; spalte++) {
            uint16_t farbe = herz_pixel[zeile * 8 + spalte];
            pbc_fill_rect(56 + spalte * 16, 40 + zeile * 16, 16, 16, farbe);
        }
    }
```

```
30      // dasselbe Bild in Originalgroesse
31      pbc_draw_image(&herz, 116, 200);
32      pbc_show();
33      while (true) {
34          sleep_ms(1000);
35      }
36  }
```



- Zeilen 4–6: `S` steht für Schwarz, `R` für Rot, `W` für Weiß. Der Compiler ersetzt jeden Buchstaben durch die Farbe.
- Zeilen 8–17: Das Array hat 64 Einträge, für jedes Pixel einen. Die Zeilenumbrüche im Quelltext sind nur für dich; für C ist es eine lange Liste.
- Zeile 26: Das Pixel in Zeile `zeile` und Spalte `spalte` steht an der Stelle `zeile * 8 + spalte`. Vor ihm liegen `zeile` volle Zeilen zu je 8 Pixeln und dann noch `spalte` Pixel.
- Zeile 27 malt jedes Pixel als Quadrat mit 16 Pixeln Kantenlänge. So siehst du das Bild stark vergrößert.

`static` vor dem Array bedeutet bei einer Variablen außerhalb von Funktionen: Sie gehört nur zu dieser Datei. `const` sagt: Die Werte ändern sich nie. Das hat einen großen Vorteil: Der Compiler darf konstante Daten im **Flash-Speicher** lassen, wo auch dein Programm liegt. Sie belegen dann keinen Platz im knappen Arbeitsspeicher. Der Flash des PicoBoy fasst 16 MB, da passen viele Bilder hinein (mehr dazu in Kapitel 19).

**Der Bildtyp `pbc_image`.** Für das Zeichnen in Originalgröße kennt die Bibliothek eine Struktur (Kapitel 13). So steht sie in `pbc.h`:

```
typedef struct {
    int width;
    int height;
    const uint16_t *pixels;   // width × height Farben, zeilenweise
    const uint8_t *mask;      // 1 = sichtbar, 0 = durchsichtig; NULL: alles sichtbar
} pbc_image;
```

Zeile 18 legt so ein Bild an: 8 breit, 8 hoch, die Farben aus `herz_pixel` und `NULL` für die Maske. `NULL` heißt „keine“: Das Bild hat keine durchsichtigen Stellen. Was die Sternchen `*` in der Struktur bedeuten, erklärt Kapitel 19. Für dich reicht: Hier wird auf die Arrays mit den Bilddaten **verwiesen**, die Daten selbst werden nicht kopiert.

Zeile 31 zeichnet das Bild mit seiner linken oberen Ecke bei (116, 200):

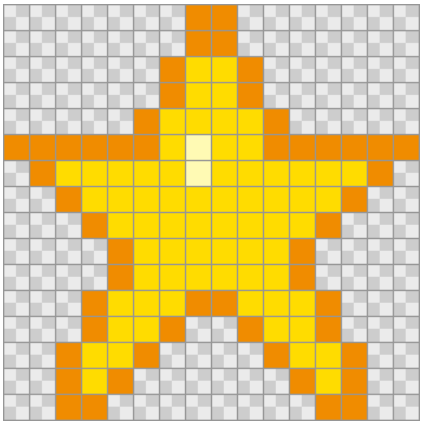
```
pbc_draw_image(&herz, 116, 200);
```

Achte auf das `&` vor dem Namen. Es muss dort stehen. Damit bekommt `pbc_draw_image` nicht eine Kopie des Bilds, sondern erfährt, wo das Bild im Speicher liegt. Auch das ist ein Thema für Kapitel 19. Vergisst du das `&`, meldet der Compiler den Fehler `incompatible type for argument`

`1 of 'pbc_draw_image'`: Das erste Argument hat den falschen Typ. In der Zeile mit `note:` darunter steht, was erwartet wird: `const pbc_image *`, mit Sternchen, also ein Verweis auf ein Bild.

`pbc_draw_image` setzt Pixel für Pixel in den Bildpuffer, genau wie die Schleife oben, nur ohne Vergrößerung. Ragt ein Bild über den Rand des Bildschirms, wird der überstehende Teil einfach weggelassen. Du kannst eine Figur also auch halb hinausschieben.

**Bilder aus einem Malprogramm.** Für größere Bilder ist das Abtippen zu mühsam. Zeichne sie lieber in einem Malprogramm, das einzelne Pixel setzen kann (viele nennen das „Pixel Art“), und speichere sie als **PNG**-Datei. Für Spielfiguren eignen sich Größen wie 16 × 16 oder 32 × 32 Pixel. Unsere Beispielbilder sind klein und liegen bei den Programmen des Buchs. So sieht der Stern `stern.png` vergrößert aus; die karierten Stellen sind durchsichtig:



Aus der PNG-Datei macht das Werkzeug **`bild2c.html`** C-Code. Es liegt im Ordner der Vorlage neben `main.c` (Kapitel 2.2) und ist eine Webseite, die ohne Internet in deinem Browser läuft. Die Bilder werden dabei nicht hochgeladen.

1. Öffne den Ordner der Vorlage im Datei-Explorer (Windows), im Finder (macOS) oder im Dateimanager (Linux) und öffne `bild2c.html` mit einem Doppelklick. Die Seite „Bild zu C“ erscheint im Browser.
2. Ziehe deine PNG-Datei in das gestrichelte Feld oder klicke hinein und wähle die Datei aus.
3. Das Feld **Name im Programm** füllt das Werkzeug mit dem Dateinamen. Ändere ihn, wenn du einen anderen Namen willst, zum Beispiel `stern`. Umlaute und Leerzeichen werden ersetzt, denn ein Name in C darf sie nicht enthalten.
4. Lass die Auswahl auf **C (Buchbibliothek pbc)**.
5. Unter der vergrößerten Vorschau steht, wie groß das Bild ist, ob es durchsichtige Stellen hat und wie viele KB Bilddaten es ergibt. Ist es größer als das Display, warnt das Werkzeug.
6. Klicke **Kopieren** und füge den Code in `main.c` ein, **über** `int main(void)`.

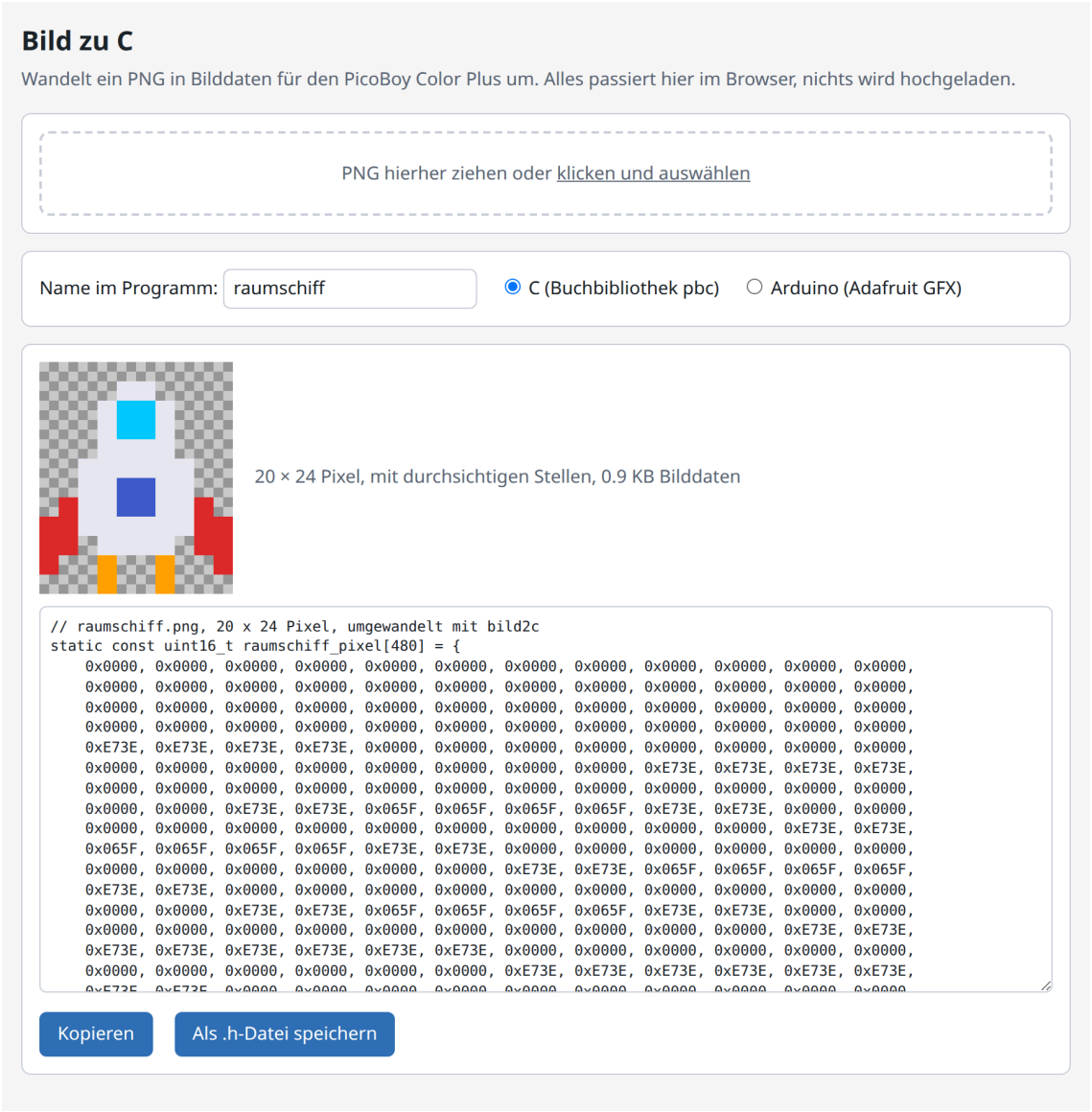



bild2c mit dem Raumschiff aus dem Buch

Der Knopf **Als .h-Datei speichern** legt den Code als eigene Datei ab. Den brauchst du erst, wenn ein Programm aus mehreren Dateien besteht (Kapitel 20). Bis dahin kommt alles in `main.c`.

Der Anfang des erzeugten Codes für `stern.png` sieht so aus (hier gekürzt, im Original stehen 12 Werte in jeder Zeile):

```
// stern.png, 16 x 16 Pixel, umgewandelt mit bild2c
static const uint16_t stern_pixel[256] = {
    0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0xF460, …
    …
};
static const uint8_t stern_maske[256] = {
    0, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
    …
};
static const pbc_image stern = { 16, 16, stern_pixel, stern_maske };
```

Das ist derselbe Aufbau wie beim Herz, nur stehen die Farben als Hexadezimalzahlen da: `0xF460` ist das Orange des Rands. Dazu kommt ein zweites Array, die **Maske**. Sie ist Thema des nächsten Abschnitts.

> **Wie viel Speicher braucht ein Bild?** Jedes Pixel kostet 2 Bytes Farbe, mit Maske noch 1 Byte mehr. Der Stern mit 16 × 16 = 256 Pixeln braucht also 512 Bytes plus 256 Bytes Maske. Ein Bild in Displaygröße hätte 134 400 Bytes – im Flash ist dafür Platz, aber bei vielen großen Bildern wird auch der irgendwann voll.

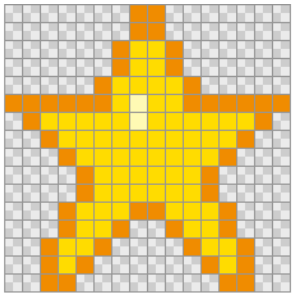
## 15.2 Bilder zeichnen, transparente Farbe

Ein Bild ist immer ein Rechteck. Eine Spielfigur ist aber selten rechteckig. Damit um sie herum der Hintergrund sichtbar bleibt, müssen manche Pixel **durchsichtig** (transparent) sein. PNG-Dateien können das: Jedes Pixel hat neben Rot, Grün und Blau einen vierten Wert, wie undurchsichtig es ist. `bild2c` macht daraus die Maske:

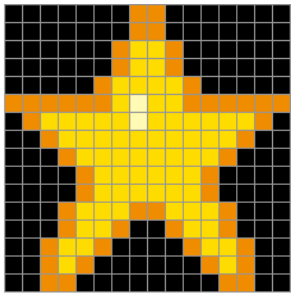
- Ein Pixel, das mindestens zur Hälfte deckt, bekommt in der Maske eine **1**: Es wird gezeichnet.
- Ein Pixel, das mehr als zur Hälfte durchsichtig ist, bekommt eine **0**: `pbc_draw_image` lässt es aus. Dort bleibt stehen, was vorher im Bildpuffer war.

Hat ein Bild keine einzige durchsichtige Stelle, lässt `bild2c` die Maske weg und schreibt `NULL`, wie beim Herz.

Der Unterschied ist deutlich zu sehen, wenn man denselben Stern einmal mit durchsichtigem und einmal mit schwarzem Hintergrund malt und beide umwandelt:



stern.png (durchsichtig)



stern_ohne.png

k15_stern.c

```c
int main(void) {
    pbc_init();
    // Himmel mit Farbverlauf: jede Zeile etwas heller
    for (int y = 0; y < PBC_HEIGHT; y++) {
        pbc_hline(0, y, PBC_WIDTH, pbc_rgb(20, 40 + y / 2, 100 + y / 2));
    }
    pbc_text12("mit Maske", 30, 70, PBC_WHITE);
    pbc_text12("ohne Maske", 140, 70, PBC_WHITE);
    for (int i = 0; i < 4; i++) {
        pbc_draw_image(&stern, 50, 100 + i * 40);
```

```
81          pbc_draw_image(&stern_ohne, 160, 100 + i * 40);
82      }
83      pbc_show();
84      while (true) {
85          sleep_ms(1000);
86      }
87  }
```

- Die Zeilen 1–69 sind die Ausgabe von `bild2c` für die beiden Bilder `stern.png` (Name `stern`, mit Maske) und `stern_ohne.png` (Name `stern_ohne`, ohne Maske). In Zeile 69 steht deshalb `NULL`.
- Zeilen 74–76 malen einen Himmel mit Farbverlauf: Jede Bildschirmzeile bekommt mit `pbc_hline` eine eigene Farbe, nach unten heller.
- Zeilen 79–82 zeichnen beide Sterne viermal untereinander.



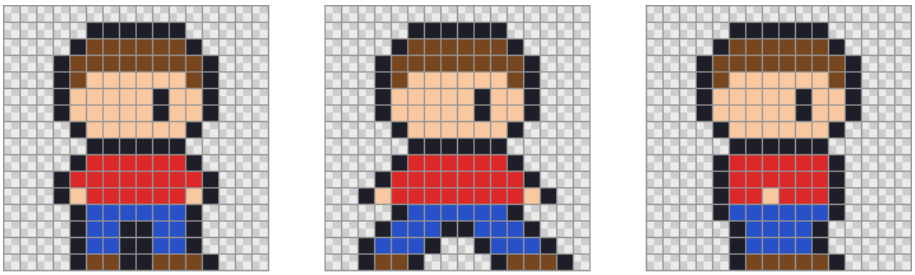

Links liegen die Sterne sauber auf dem Himmel. Rechts sitzt jeder Stern in einem schwarzen Kästchen.

> Ein Malprogramm, das keine Transparenz kennt, speichert den Hintergrund mit. Dann helfen zwei Wege: Nimm ein Malprogramm, das durchsichtige Pixel kann, und speichere wieder als PNG; die Durchsichtigkeit bleibt dabei erhalten. Oder schreib dir eine eigene Zeichenschleife, die eine bestimmte Farbe wie Magenta nicht zeichnet. Das ist Aufgabe 5.

**Was zuerst gezeichnet wird, liegt hinten.** Der Bildpuffer kennt keine Ebenen. Jedes Bild übermalt, was schon da ist. Zeichne also zuerst den Hintergrund, dann Gegenstände, zuletzt die Figuren und die Anzeige von Punkten und Leben.

## 15.3 Animierte Figuren aus mehreren Bildern

Ein Bild, das sich bewegt, aus vielen Einzelbildern zusammengesetzt, kennst du aus dem Daumenkino. Eine Figur auf dem Bildschirm, die sich so bewegt, nennt man **Sprite**. Damit sie läuft, braucht sie mehrere Bilder, die sich abwechseln. Unsere Figur hat drei: Stehen und zwei Schritte.



figur_stehen figur_schritt1 figur_schritt2

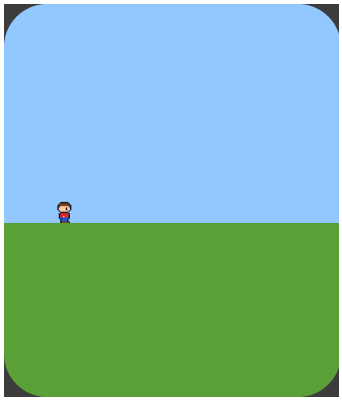
Zum Laufen wechseln sich `figur_schritt1` und `figur_schritt2` ab. Damit das Programm das richtige Bild mit einer Zahl auswählen kann, kommen die Bilder in ein Array:

```
const pbc_image bilder[2] = { lauf1, lauf2 };
```

Dann zeichnet `pbc_draw_image(&bilder[schritt], x, y)` das Bild mit der Nummer `schritt`. Das `&` steht vor dem ganzen Ausdruck `bilder[schritt]`.

k15_laufen.c

```
 98 int main(void) {
 99     pbc_init();
100     const pbc_image bilder[2] = { lauf1, lauf2 };
101     int x = 10;
102     int schritt = 0;            // welches Bild gerade gezeigt wird
103     int zaehler = 0;            // zaehlt die Bilder bis zum Wechsel
104
105     while (true) {
106         x = x + 1;
107         if (x > 214) {
108             x = 10;
109         }
110         zaehler++;
111         if (zaehler == 5) {     // alle 5 Bilder (200 ms) wechseln
112             zaehler = 0;
113             schritt = (schritt + 1) % 2;
114         }
115
116         pbc_clear(pbc_rgb(150, 200, 255));
117         pbc_fill_rect(0, 156, PBC_WIDTH, 124, pbc_rgb(90, 160, 60));
118         pbc_draw_image(&bilder[schritt], x, 140);
119         pbc_show();
120         bildtakt();
121     }
122 }
```



- Die Zeilen 1–96 enthalten die Bilder `lauf1` und `lauf2` aus `figur_schritt1.png` und `figur_schritt2.png` und die Funktion `bildtakt()` aus Kapitel 14.
- Zeile 100 legt das Array an. Die beiden Bilder werden dabei in das Array kopiert. Das sind nur die vier Werte der Struktur, nicht die Pixel.
- Zeilen 110–114: Das Programm zeigt 25 Bilder pro Sekunde. Würde die Figur bei jedem davon den Schritt wechseln, sähe es aus, als zappelte sie. Ein Zähler sorgt dafür, dass der Schritt nur alle 5 Bilder wechselt, also alle 200 ms. `% 2` macht aus 0, 1, 2, 3 … die Folge 0, 1, 0, 1 …
- Zeile 118: Die Figur ist 16 Pixel hoch. Mit y = 140 stehen ihre Füße genau auf dem Boden, der bei y = 156 beginnt.

> Bewegung und Bildwechsel sind zwei getrennte Dinge: Die Position ändert sich in jedem Bild ein wenig, das Sprite-Bild nur alle paar Bilder. Wie schnell eine Figur „trippelt“, stellst du mit dem Zähler ein.

**Nach links schauen.** Läuft die Figur nach links, sollte sie auch nach links schauen. `pbc_draw_image` kann ein Bild nicht spiegeln. Am einfachsten spiegelst du die Bilder im Malprogramm und wandelst sie ebenfalls um. Das nächste Programm benutzt deshalb sechs Bilder.

## Ausprobieren: Eine laufende Figur

Die Figur läuft mit dem Joystick nach links und rechts, schaut dabei in die richtige Richtung und bleibt stehen, wenn du loslässt.

k15_figur.c

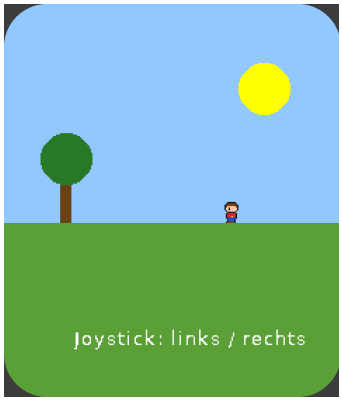
```c
void zeichne_landschaft(void) {
    pbc_clear(pbc_rgb(150, 200, 255));
    pbc_fill_circle(185, 60, 18, PBC_YELLOW);
    pbc_fill_rect(0, BODEN, PBC_WIDTH, PBC_HEIGHT - BODEN, pbc_rgb(90, 160, 60));
    pbc_fill_rect(40, 120, 8, 36, pbc_rgb(110, 70, 30));
    pbc_fill_circle(44, 110, 18, pbc_rgb(40, 120, 40));
    pbc_text12("Joystick: links / rechts", 50, 230, PBC_WHITE);
}

int main(void) {
    pbc_init();
    const pbc_image rechts[3] = { steht, lauf1, lauf2 };
    const pbc_image links[3] = { steht_l, lauf1_l, lauf2_l };
    int x = 112;
    bool nach_links = false;
    int zaehler = 0;

    while (true) {
        bool laeuft = false;
        if (pbc_pressed_left() && x > 10) {
            x = x - 2;
            nach_links = true;
            laeuft = true;
        }
        if (pbc_pressed_right() && x < 214) {
            x = x + 2;
            nach_links = false;
            laeuft = true;
        }

        int nummer = 0;                       // Bild 0: stehen
        if (laeuft) {
            zaehler++;
            nummer = 1 + (zaehler / 4) % 2;   // Bild 1 und 2 im Wechsel
        }
```

```
295         zeichne_landschaft();
296         if (nach_links) {
297             pbc_draw_image(&links[nummer], x, BODEN - 16);
298         } else {
299             pbc_draw_image(&rechts[nummer], x, BODEN - 16);
300         }
301         pbc_show();
302         bildtakt();
303     }
304 }
```




- Die Zeilen 1–245 sind die sechs umgewandelten Bilder: `figur_stehen.png`, `figur_schritt1.png` und `figur_schritt2.png` als `steht`, `lauf1` und `lauf2`, dieselben gespiegelt (Dateien mit `_links` am Ende) als `steht_l`, `lauf1_l` und `lauf2_l`. Danach folgt `bildtakt()`.
- Zeilen 259–266: Die Landschaft hat eine eigene Funktion. So bleibt die Hauptschleife übersichtlich.
- Zeilen 270–271: zwei Arrays mit je drei Bildern. Bild 0 ist Stehen, Bild 1 und 2 sind die Schritte.
- Zeilen 277–287: `laeuft` merkt sich, ob in diesem Durchgang eine Richtung gedrückt war. `nach_links` merkt sich die Blickrichtung. Sie bleibt erhalten, wenn du loslässt.
- Zeilen 289–293: Steht die Figur, ist `nummer` 0. Läuft sie, wechselt `nummer` alle 4 Bilder zwischen 1 und 2.
- Zeilen 296–300 wählen das Array nach der Blickrichtung.

Das Bild rechts zeigt die Figur, während sie nach links läuft.

**Zum Verändern:** Zeichne eigene Bilder für die Figur, etwa mit Hut oder als Tier, und wandle sie um. Lass die Figur mit Taste A springen: Gib ihr eine Geschwindigkeit `vy` und Schwerkraft wie dem Ball aus Kapitel 14.

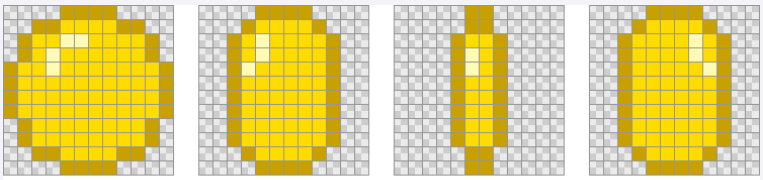
## Übungsaufgaben

> **Aufgabe 1** ⭐ **Speicher rechnen** Wie viele Bytes brauchen die Bilddaten (Farben und Maske) für eine Figur aus 32 × 32 Pixeln mit durchsichtigen Stellen? Wie viele für acht solcher Bilder? Und wie viele für ein Hintergrundbild in Displaygröße ohne durchsichtige Stellen?

> **Aufgabe 2** ⭐ **Ein Geist** Schreibe von Hand ein 8 × 8 Pixel großes Bild eines Geists (weiß, mit blauen Augen) wie das Herz in 15.1. Zeige ihn einmal stark vergrößert und darunter dreimal nebeneinander in Originalgröße.

**Aufgabe 3 ⭐⭐ Sternenhimmel** Wandle `stern.png` mit `bild2c.html` um. Zeichne einen dunklen Nachthimmel mit einer Mondsichel (zwei Kreise!) und 15 Sternen an zufälligen Stellen.

**Aufgabe 4 ⭐⭐ Münze dreht sich** Die Bilder `muenze_0.png` bis `muenze_3.png` zeigen eine Münze in vier Stellungen einer Drehung. Wandle sie um und lass fünf Münzen nebeneinander sich drehen, alle 100 ms ein Bild weiter. Damit es lebendig aussieht, soll jede Münze ein Bild weiter sein als ihre linke Nachbarin.




**Aufgabe 5 ⭐⭐⭐ Eigene Zeichenschleife** Schreibe einen Fisch aus 8 × 8 Pixeln von Hand als Array. Wo kein Fisch ist, steht `PBC_MAGENTA`. Schreibe eine Funktion `zeichne_fisch(x, y, faktor, gespiegelt)`, die den Fisch `faktor`-mal so groß zeichnet, Magenta-Pixel auslässt und den Fisch auf Wunsch spiegelt. Zeichne ihn in verschiedenen Größen und Richtungen.

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- Ein Bild ist ein Array von Farben, Zeile für Zeile. Das Pixel (spalte, zeile) steht an der Stelle `zeile * breite + spalte`.
- `bild2c.html` aus der Vorlage wandelt PNG-Dateien in C-Code um: Datei hineinziehen, Namen wählen, **Kopieren**, über `main` einfügen.
- Der Code enthält Farben (`name_pixel`), bei durchsichtigen Stellen eine Maske (`name_maske`, 1 = zeichnen, 0 = auslassen) und das Bild als `pbc_image`.
- `static const` lässt Bilddaten im Flash; sie kosten keinen Arbeitsspeicher.
- `pbc_draw_image(&name, x, y)` zeichnet ein Bild, (x, y) ist die linke obere Ecke. Das `&` muss sein.
- Was später gezeichnet wird, liegt vorn.
- Animation: mehrere Bilder in ein Array, die Nummer alle paar Bilder weiterzählen.

KAPITEL 16

# Kollisionen

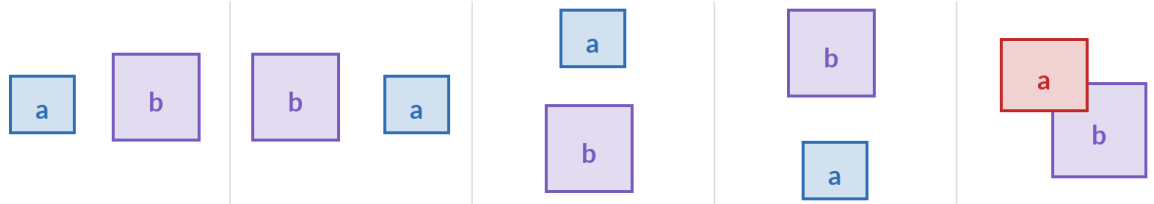
**Worum geht's?** In einem Spiel müssen Dinge aufeinandertreffen: Die Figur sammelt einen Stern ein, ein Stein trifft den Spieler, der Ball prallt vom Schläger ab. Du lernst, wie ein Programm erkennt, ob sich zwei Rechtecke oder zwei Kreise berühren, und wie man Treffer zählt, ohne dass ein einziger Zusammenstoß gleich mehrfach zählt.

## 16.1 Rechteck gegen Rechteck

Für den Computer sind die Dinge auf dem Bildschirm nur Zahlen. Ob sich zwei davon berühren, muss das Programm ausrechnen. Das nennt man **Kollisionserkennung** (*collision detection*). Am einfachsten geht es mit Rechtecken, deren Seiten waagrecht und senkrecht sind. Auch eine Spielfigur aus einem Bild (Kapitel 15) ist ja ein Rechteck. Ein Rechteck beschreiben wir wie `pbc_fill_rect` mit linker oberer Ecke, Breite und Höhe:

```
typedef struct {
    int x, y;                   // linke obere Ecke
    int w, h;                   // Breite und Hoehe
} rechteck;
```

Wann berühren sich zwei Rechtecke `a` und `b`? Leichter ist die umgekehrte Frage: Wann berühren sie sich **nicht**? Genau dann, wenn eins ganz links, ganz rechts, ganz über oder ganz unter dem anderen liegt:




`a.x + a.w` ist die x-Koordinate direkt rechts neben `a`. Ist sie kleiner oder gleich `b.x`, liegt `a` vollständig links von `b`. Trifft keiner der vier Fälle zu, überlappen sich die Rechtecke. Dreht man jede der vier Bedingungen um und verbindet sie mit `&&`, entsteht diese Funktion:

k16_rechtecke.c

```
1 // Rechteck gegen Rechteck: Beruehren sich die beiden Kaesten?
2 #include "pbc.h"
3
4 typedef struct {
5     int x, y;                 // linke obere Ecke
6     int w, h;                 // Breite und Hoehe
7 } rechteck;
```

```
 9 bool rechtecke_beruehren(rechteck a, rechteck b) {
10     return a.x < b.x + b.w && b.x < a.x + a.w &&
11            a.y < b.y + b.h && b.y < a.y + a.h;
12 }
13
14 void zeichne(rechteck r, uint16_t farbe) {
15     pbc_fill_rect(r.x, r.y, r.w, r.h, farbe);
16 }
```

Lies Zeile 10 so: „`a` fängt links vom rechten Rand von `b` an, **und** `b` fängt links vom rechten Rand von `a` an.“ Zeile 11 sagt dasselbe für oben und unten. Nur wenn alle vier Vergleiche wahr sind, liefert die Funktion `true`.

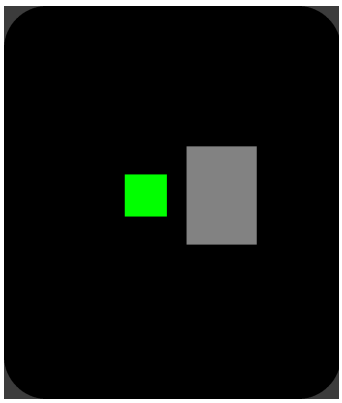
> Weil `<` statt `<=` dasteht, zählt es **nicht** als Berührung, wenn zwei Rechtecke nur genau aneinanderstoßen. Ein Rechteck mit `x = 10` und `w = 20` belegt die Pixel 10 bis 29. Eins, das bei `x = 30` beginnt, liegt direkt daneben, ohne ein Pixel zu teilen.

Die Funktion `zeichne` in Zeile 14 malt ein `rechteck`. Mit ihr bleibt die Hauptschleife kurz. Im Programm steuerst du einen grünen Kasten mit dem Joystick. Berührt er die graue Kiste, wird er rot:

k16_rechtecke.c

```
18 int main(void) {
19     pbc_init();
20     rechteck spieler = { 40, 120, 30, 30 };
21     rechteck kiste = { 130, 100, 50, 70 };
22
23     while (true) {
24         if (pbc_pressed_left()) {
25             spieler.x = spieler.x - 2;
26         }
27         if (pbc_pressed_right()) {
28             spieler.x = spieler.x + 2;
29         }
30         if (pbc_pressed_up()) {
31             spieler.y = spieler.y - 2;
32         }
33         if (pbc_pressed_down()) {
34             spieler.y = spieler.y + 2;
35         }
36
37         pbc_clear(PBC_BLACK);
38         zeichne(kiste, PBC_GRAY);
39         if (rechtecke_beruehren(spieler, kiste)) {
40             zeichne(spieler, PBC_RED);
41             pbc_text18("Beruehrt!", 75, 210, PBC_RED);
42         } else {
43             zeichne(spieler, PBC_GREEN);
44         }
45         pbc_show();
```

```
46          sleep_ms(20);
47      }
48  }
```

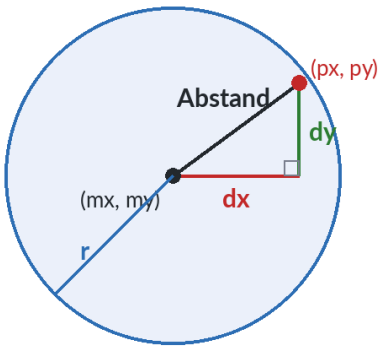


Das Bild rechts zeigt den Kasten, nachdem er nach rechts gegen die Kiste geschoben wurde.

**Kollisionsrechtecke.** Bei einem Sprite ist das Bild meist größer als die Figur: Um sie herum sind durchsichtige Pixel. Würde man das ganze Bild als Rechteck nehmen, zählten schon Berührungen der unsichtbaren Ecken. Viele Spiele prüfen deshalb mit einem etwas kleineren Rechteck, der **Hitbox**. Für eine 16 × 16 Pixel große Figur bei (`x`, `y`) wäre das zum Beispiel `{ x + 3, y + 2, 10, 14 }`. Das wirkt für Spieler fairer.

## 16.2 Punkt im Kreis

Bälle, Münzen und Explosionen sind rund. Für sie prüft man mit dem **Abstand**: Ein Punkt liegt im Kreis, wenn sein Abstand zum Mittelpunkt höchstens so groß ist wie der Radius.




`Abstand² = dx² + dy²`

Punkt im Kreis, wenn
`dx*dx + dy*dy <= r*r`

`dx` und `dy` sind die Unterschiede in x- und y-Richtung. Mit dem Abstand bilden sie ein rechtwinkliges Dreieck, also gilt der Satz des Pythagoras: Abstand² = dx² + dy². Für den Abstand selbst müsste man die Wurzel ziehen. Das kann man sich sparen: Statt „Abstand ≤ r“ prüft man „Abstand² ≤ r²“. Das ist gleichwertig, weil beide Seiten positiv sind, und kommt mit ganzen Zahlen und Malnehmen aus.

k16_kreis.c

```
1 // Punkt im Kreis: ein Fadenkreuz zielt auf eine Scheibe
2 #include "pbc.h"
3
4 #define MX 120                  // Mittelpunkt der Zielscheibe
5 #define MY 130
6 #define RADIUS 40
```

```c
bool punkt_im_kreis(int px, int py, int mx, int my, int r) {
    int dx = px - mx;
    int dy = py - my;
    return dx * dx + dy * dy <= r * r;      // Abstand hoch 2 <= r hoch 2
}
```

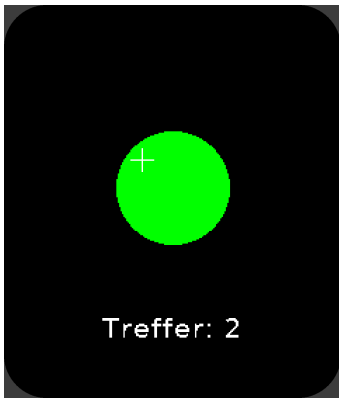
In diesem Programm steuerst du ein Fadenkreuz. Liegt es über der Scheibe, wird sie grün. Drückst du dann A, zählt ein Treffer:

k16_kreis.c

```c
        bool drin = punkt_im_kreis(x, y, MX, MY, RADIUS);
        if (pbc_was_pressed_a() && drin) {
            treffer++;
            pbc_tone(880, 80);
        }

        pbc_clear(PBC_BLACK);
        if (drin) {
            pbc_fill_circle(MX, MY, RADIUS, PBC_GREEN);
        } else {
            pbc_fill_circle(MX, MY, RADIUS, PBC_BLUE);
        }
        pbc_hline(x - 8, y, 17, PBC_WHITE);
        pbc_vline(x, y - 8, 17, PBC_WHITE);
        snprintf(text, sizeof text, "Treffer: %d", treffer);
        pbc_text18(text, 70, 220, PBC_WHITE);
```

- Zeile 34 prüft einmal pro Bild, ob das Fadenkreuz in der Scheibe ist, und merkt sich das Ergebnis in `drin`. Die Zeilen 35 und 41 benutzen es.
- Zeile 35: Ein Treffer zählt nur, wenn A gedrückt wurde **und** das Fadenkreuz drin ist. `pbc_was_pressed_a()` meldet jeden Druck genau einmal (Kapitel 9). Gedrückt halten zählt also nicht mehrfach.
- Zeilen 46–47: Das Fadenkreuz besteht aus zwei Linien, 17 Pixel lang mit (`x`, `y`) in der Mitte.




Das Bild zeigt den Stand nach zwei Treffern.

**Kreis gegen Kreis.** Zwei Kreise berühren sich, wenn der Abstand ihrer Mittelpunkte höchstens so groß ist wie **beide Radien zusammen**. Es ist dieselbe Rechnung, nur mit `r1 + r2` statt `r`:

```c
bool kreise_beruehren(float x1, float y1, float r1,
                      float x2, float y2, float r2) {
    float dx = x1 - x2;
    float dy = y1 - y2;
    float r = r1 + r2;
    return dx * dx + dy * dy <= r * r;
}
```

Hier mit `float`, weil Spielfiguren mit Geschwindigkeit ihre Position als Kommazahl speichern (Kapitel 14). Du findest die Funktion im Programm am Ende des Kapitels wieder.

> Ist ein Gegenstand weder rund noch eckig, nimm die Form, die besser passt. Für einen Stern ist ein Kreis, der etwas kleiner ist als das Bild, meist eine gute Wahl. Ganz genau muss es nicht sein. Wichtig ist, dass sich das Spiel fair anfühlt.

## 16.3 Treffer zählen, Leben verlieren

Eine Kollision dauert meist mehrere Bilder lang. Fällt ein Stein auf den Spieler, überlappen sich beide in vielen Bildern hintereinander. Zieht das Programm in jedem dieser Bilder ein Leben ab, ist das Spiel nach einem einzigen Stein vorbei. Zwei Mittel helfen dagegen:

1. **Das Objekt verschwinden lassen.** Ein eingesammelter Stern oder ein getroffener Stein wird sofort an eine neue Stelle gesetzt. Im nächsten Bild gibt es keine Überlappung mehr.
2. **Eine Schutzzeit.** Nach einem Treffer ist der Spieler eine Weile unverwundbar. Damit man das sieht, blinkt er in dieser Zeit.

Das folgende Programm benutzt beides. Steine fallen von oben; der Spieler weicht unten mit dem Joystick aus.

k16_leben.c

```
 1 // Steine fallen: Jeder Treffer kostet ein Leben, danach kurz Schutz
 2 #include "pbc.h"
 3
 4 #define BILD_MS 40
 5 #define ANZAHL 4
 6 #define SCHUTZ_MS 1000          // so lange nach einem Treffer unverwundbar
 7
 8 typedef struct {
 9     int x, y;
10     int w, h;
11 } rechteck;
12
13 typedef struct {
14     rechteck box;               // wo der Stein ist und wie gross
15     int tempo;                  // Pixel pro Bild nach unten
16 } stein;
17
18 stein steine[ANZAHL];
```

Neu ist Zeile 14: Die Struktur `stein` enthält eine andere Struktur, ein `rechteck`. Auf dessen Werte greifst du mit zwei Punkten zu: `steine[i].box.y` ist das `y` der `box` von Stein Nummer `i`. Das Array `steine` ist global, damit die Funktion `stein_neu` es ändern kann.

k16_leben.c

```c
// Setzt Stein i an eine zufaellige Stelle oberhalb des Bildschirms.
void stein_neu(int i) {
    steine[i].box.x = pbc_random(20, 204);
    steine[i].box.y = pbc_random(-200, -20);
    steine[i].box.w = 16;
    steine[i].box.h = 16;
    steine[i].tempo = pbc_random(2, 5);
}
```

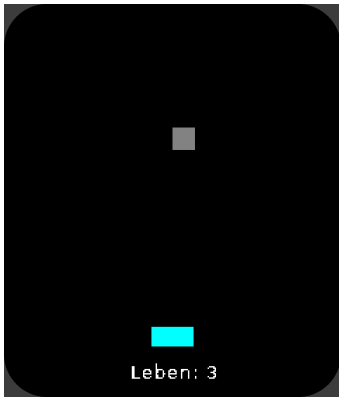
`stein_neu` setzt einen Stein **oberhalb** des Bildschirms ab, bei einem y zwischen −200 und −20. So erscheinen die Steine nicht plötzlich mitten im Bild, sondern fallen von oben herein. Was außerhalb des Bildschirms liegt, zeichnet `pbc_fill_rect` einfach nicht.

k16_leben.c

```c
int main(void) {
    pbc_init();
    rechteck spieler = { 105, 230, 30, 14 };
    int leben = 3;
    uint32_t schutz_bis = 0;    // bis zu dieser Zeit unverwundbar
    char text[20];
    for (int i = 0; i < ANZAHL; i++) {
        stein_neu(i);
    }

    while (true) {
        if (leben > 0) {
            if (pbc_pressed_left() && spieler.x > 20) {
                spieler.x = spieler.x - 4;
            }
            if (pbc_pressed_right() && spieler.x + spieler.w < 220) {
                spieler.x = spieler.x + 4;
            }
            for (int i = 0; i < ANZAHL; i++) {
                steine[i].box.y = steine[i].box.y + steine[i].tempo;
                if (steine[i].box.y > PBC_HEIGHT) {
                    stein_neu(i);                   // unten hinaus
                }
                if (pbc_millis() >= schutz_bis &&
                    rechtecke_beruehren(spieler, steine[i].box)) {
                    leben--;
                    schutz_bis = pbc_millis() + SCHUTZ_MS;
                    stein_neu(i);                   // nur einmal zaehlen
                    pbc_tone(150, 200);
                }
            }
        } else if (pbc_was_pressed_a()) {           // neues Spiel
            leben = 3;
            for (int i = 0; i < ANZAHL; i++) {
                stein_neu(i);
            }
        }
```

```
88          pbc_clear(PBC_BLACK);
89          for (int i = 0; i < ANZAHL; i++) {
90              zeichne(steine[i].box, PBC_GRAY);
91          }
92          // Solange der Schutz laeuft, blinkt der Spieler (100 ms an, 100 aus).
93          bool geschuetzt = pbc_millis() < schutz_bis;
94          if (!geschuetzt || (pbc_millis() / 100) % 2 == 0) {
95              zeichne(spieler, PBC_CYAN);
96          }
97          snprintf(text, sizeof text, "Leben: %d", leben);
98          pbc_text12(text, 90, 254, PBC_WHITE);
99          if (leben == 0) {
100             pbc_text24("Game over", 64, 100, PBC_RED);
101             pbc_text12("A: nochmal", 86, 140, PBC_WHITE);
102         }
103         pbc_show();
104         bildtakt();
105     }
106 }
```




- Zeile 54: `schutz_bis` ist der Zeitpunkt (in `pbc_millis()`), bis zu dem der Spieler geschützt ist. 0 heißt: von Anfang an nicht geschützt.
- Zeile 61: Nur solange es noch Leben gibt, bewegt sich etwas.
- Zeilen 70–72: Ein Stein, der unten hinausfällt, kommt oben neu.
- Zeilen 73–79: Ein Treffer zählt nur, wenn der Schutz abgelaufen ist. Dann kostet er ein Leben, startet die Schutzzeit von einer Sekunde, und der Stein kommt sofort neu (Zeile 77).
- Zeilen 81–86: Sind alle Leben weg, startet A ein neues Spiel.
- Zeilen 93–96: Ist der Spieler ungeschützt, wird er immer gezeichnet. Ist er geschützt, nur wenn `(pbc_millis() / 100) % 2` gleich 0 ist. Dieser Ausdruck wechselt alle 100 ms zwischen 0 und 1: Der Spieler blinkt.

> Ein Zusammenstoß dauert mehrere Bilder. Sorge dafür, dass er nur **einmal** zählt: Objekt neu setzen oder eine Schutzzeit starten.

## Ausprobieren: Sterne einsammeln

Ein Spiel mit allem aus diesem Kapitel: Du steuerst einen Kreis über das Spielfeld und sammelst Sterne ein. Ein Feuerball prallt im Feld herum; er kostet ein Leben, wenn er dich erwischt. Nach jedem Stern wird er etwas schneller.

Das Programm ist in kleine Funktionen aufgeteilt. Die Daten des Spiels stehen deshalb in globalen Variablen, damit jede Funktion sie lesen und ändern kann.

k16_sterne.c

```
51 typedef struct {
52     float x, y;                  // Mittelpunkt
53     float vx, vy;                // Pixel pro Bild
54     int r;                       // Radius
55 } kugel;
56
57 kugel spieler;
58 kugel feuer;
59 int stern_x, stern_y;           // linke obere Ecke des Sternbilds
60 int punkte;
61 int leben;
62 uint32_t schutz_bis;
```

Die Zeilen 1–49 enthalten die Konstanten für das Spielfeld und das Bild `stern` aus `stern.png` (Kapitel 15). In den Zeilen 64–74 steht `bildtakt()`.

k16_sterne.c

```
76 bool kreise_beruehren(float x1, float y1, float r1,
77                        float x2, float y2, float r2) {
78     float dx = x1 - x2;
79     float dy = y1 - y2;
80     float r = r1 + r2;
81     return dx * dx + dy * dy <= r * r;
82 }
83
84 void stern_neu(void) {
85     stern_x = pbc_random(LINKS, RECHTS - 16);
86     stern_y = pbc_random(OBEN + 20, UNTEN - 16);
87 }
88
89 void neues_spiel(void) {
90     spieler.x = 120.0f;
91     spieler.y = 200.0f;
92     spieler.r = 10;
93     feuer.x = 60.0f;
94     feuer.y = 60.0f;
95     feuer.vx = 2.5f;
96     feuer.vy = 2.0f;
97     feuer.r = 8;
98     punkte = 0;
99     leben = 3;
100    schutz_bis = 0;
101    stern_neu();
102 }
```

- Zeilen 76–82: Kreis gegen Kreis, wie in 16.2.
- Zeilen 89–102: `neues_spiel` stellt alles auf Anfang. Es wird beim Start und nach „Game over“ aufgerufen.

Die Funktionen `spieler_bewegen` (Zeilen 104–117) und `feuer_bewegen` (Zeilen 119–130) bewegen die beiden Kreise wie in Kapitel 14. Der Spieler bleibt dabei im Spielfeld, der Feuerball prallt an dessen Rand ab.

k16_sterne.c

```
132 void kollisionen_pruefen(void) {
133     // Stern: Mittelpunkt des 16 x 16 Bilds, Radius 7
134     if (kreise_beruehren(spieler.x, spieler.y, spieler.r,
135                          stern_x + 8, stern_y + 8, 7)) {
136         punkte++;
137         pbc_tone(1320, 60);
138         stern_neu();
139         feuer.vx = feuer.vx * 1.1f;         // das Feuer wird schneller
140         feuer.vy = feuer.vy * 1.1f;
141     }
142     if (pbc_millis() >= schutz_bis &&
143         kreise_beruehren(spieler.x, spieler.y, spieler.r,
144                          feuer.x, feuer.y, feuer.r)) {
145         leben--;
146         schutz_bis = pbc_millis() + SCHUTZ_MS;
147         pbc_tone(120, 300);
148     }
149 }
```

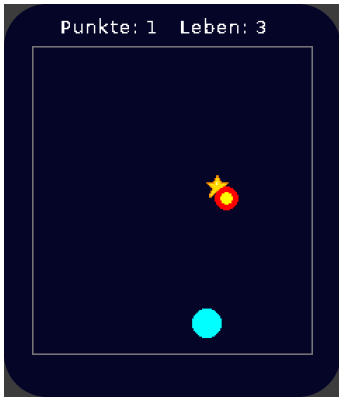
- Zeilen 134–135: Der Stern ist ein 16 × 16 Pixel großes Bild mit der linken oberen Ecke bei (`stern_x`, `stern_y`). Sein Mittelpunkt liegt deshalb 8 Pixel weiter rechts und unten. Als Radius nehmen wir 7, etwas kleiner als das Bild.
- Zeilen 136–140: Einsammeln gibt einen Punkt und einen hohen Ton. Der Stern springt an eine neue Stelle, und das Feuer wird 10 % schneller.
- Zeilen 142–148: Der Feuerball kostet ein Leben, aber nur ohne Schutz.

k16_sterne.c

```
172 int main(void) {
173     pbc_init();
174     neues_spiel();
175     while (true) {
176         if (leben > 0) {
177             spieler_bewegen();
178             feuer_bewegen();
179             kollisionen_pruefen();
180         } else if (pbc_was_pressed_a()) {
181             neues_spiel();
182         }
183         zeichnen();
184         pbc_show();
185         bildtakt();
186     }
187 }
```

Die Hauptschleife ist jetzt ganz kurz: bewegen, prüfen, zeichnen, zeigen, warten. Die Funktion `zeichnen` (Zeilen 151–170) malt Spielfeld, Anzeige, Stern, Feuer und Spieler und bei „Game over“ einen Kasten mit Text.

Das Bild zeigt das Spiel, kurz nachdem der erste Stern eingesammelt wurde.




**Zum Verändern:** Nimm statt des Kreises die laufende Figur aus Kapitel 15 als Spieler und prüfe mit einer Hitbox. Lass nach jeweils fünf Sternen einen zweiten Feuerball dazukommen (ein Array aus `kugel`).

## Übungsaufgaben

**Aufgabe 1 ⭐ Berühren sie sich?** Entscheide ohne Programm, ob sich die beiden Rechtecke (x, y, w, h) berühren, und begründe mit den vier Fällen aus 16.1.

1. a = (10, 10, 30, 20), b = (35, 25, 10, 10)
2. a = (10, 10, 30, 20), b = (40, 10, 10, 10)
3. a = (50, 50, 100, 10), b = (90, 20, 5, 100)
4. a = (0, 0, 20, 20), b = (5, 21, 5, 5)

**Aufgabe 2 ⭐ Finde den Fehler** Diese Funktion soll prüfen, ob ein Punkt in einem Kreis mit Radius 30 liegt. Der Punkt 20 Pixel rechts vom Mittelpunkt liegt eindeutig drin, trotzdem gibt das Programm 0 aus. Was ist falsch?

k16_aufgabe2.c

```c
bool punkt_im_kreis(int px, int py, int mx, int my, int r) {
    int dx = px - mx;
    int dy = py - my;
    return dx * dx + dy * dy <= r;
}

int main(void) {
    pbc_init();
    printf("Mitte:     %d\n", punkt_im_kreis(120, 140, 120, 140, 30));
    printf("5 rechts:  %d\n", punkt_im_kreis(125, 140, 120, 140, 30));
    printf("20 rechts: %d\n", punkt_im_kreis(140, 140, 120, 140, 30));
    printf("40 rechts: %d\n", punkt_im_kreis(160, 140, 120, 140, 30));
```

Serial Monitor

```
Mitte:     1
5 rechts:  1
20 rechts: 0
40 rechts: 0
```

**Aufgabe 3 ⭐⭐ Ein Menü mit Knöpfen** Zeichne drei Knöpfe untereinander mit den Aufschriften „Spielen“, „Optionen“ und „Hilfe“. Ein kleiner Zeiger fährt mit dem Joystick über den Bildschirm. Der Knopf, über dem er steht, wird ausgefüllt. Drückst du A, schreibt das Programm in die Konsole, welcher Knopf gewählt wurde. Schreib dazu eine Funktion `punkt_im_rechteck`. Tipp: Eine Struktur `knopf` kann ein `rechteck` und den Text (`char name[12]`) enthalten.

**Aufgabe 4 ⭐⭐ Ball und Schläger** Ein Ball prallt oben und an den Seiten ab. Unten steuerst du einen Schläger mit dem Joystick. Trifft der Ball den Schläger, prallt er nach oben ab und es gibt einen Punkt. Fällt er unten hinaus, kostet das ein Leben, und er startet neu. Nach drei Leben ist Schluss. Prüfe die Berührung mit einem Rechteck um den Ball herum.

**Aufgabe 5 ⭐⭐⭐ Kreis gegen Rechteck** Ein Rechteck um einen Kreis herum ist an den Ecken ungenau. Genauer geht es so: Suche den Punkt des Rechtecks, der dem Kreismittelpunkt am nächsten ist. Dazu begrenzt du die x-Koordinate des Mittelpunkts auf den Bereich von `b.x` bis `b.x + b.w - 1` und die y-Koordinate ebenso. Liegt dieser Punkt im Kreis, berühren sie sich. Schreib eine Funktion `kreis_beruehrt_rechteck` und teste sie mit einem Kreis, den du um eine Kiste herumsteuerst. Zeige den nächsten Punkt als kleines Kreuz an.

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- **Rechteck gegen Rechteck:** Sie berühren sich, wenn `a.x < b.x + b.w && b.x < a.x + a.w && a.y < b.y + b.h && b.y < a.y + a.h`.
- **Punkt im Kreis:** `dx * dx + dy * dy <= r * r`, ohne Wurzel.
- **Kreis gegen Kreis:** wie Punkt im Kreis mit `r1 + r2` als Radius.
- Für Sprites eine **Hitbox** nehmen, die etwas kleiner ist als das Bild.
- Eine Kollision dauert mehrere Bilder. Damit sie nur einmal zählt: Objekt neu setzen oder eine **Schutzzeit** starten (`schutz_bis`).
- Blinken: `(pbc_millis() / 100) % 2` wechselt alle 100 ms.

KAPITEL 17

# Der Neigungssensor

**Worum geht's?** Im PicoBoy steckt ein Sensor, der merkt, wie das Gerät gehalten und gekippt wird. Du liest seine drei Achsen aus, glättest zittrige Messwerte, steuerst eine Kugel durch Kippen und baust eine Wasserwaage.

## 17.1 Neigung lesen: `pbc_accel_x()` …

Der Neigungssensor ist ein **Beschleunigungssensor**, ein kleiner Chip namens STK8BA58 auf der Platine. Er misst, wie stark das Gerät in eine Richtung beschleunigt wird. Das Besondere: Auch die **Erdanziehung** wirkt auf den Sensor wie eine Beschleunigung. Liegt der PicoBoy ruhig auf dem Tisch, misst der Sensor deshalb etwa 9,81 Meter pro Sekunde zum Quadrat (m/s²), so stark zieht die Erde an allem. Diesen Wert nennt man auch **1 g** (g für *gravity*, Schwerkraft).

Der Sensor misst in drei Richtungen, den **Achsen**. In der Buchbibliothek sind sie so festgelegt:

- **x** zeigt nach **rechts**,
- **y** zeigt nach **unten**, zum Joystick, wie die y-Koordinate auf dem Display,
- **z** zeigt aus dem Display **heraus**, auf dich zu.




Die Zeichnung ist schematisch. Zeigt eine Achse nach **oben**, misst sie die Erdanziehung als positiven Wert, zeigt sie nach unten, als negativen. Liegt eine Achse waagrecht, misst sie 0. Daraus ergeben sich diese Werte für ein ruhig gehaltenes Gerät:

| Lage des PicoBoy | x | y | z |
|---|---|---|---|
| flach auf dem Tisch, Display oben | 0 | 0 | +1 g |
| flach, Display nach unten | 0 | 0 | −1 g |
| aufrecht wie ein Bild an der Wand, Joystick unten | 0 | −1 g | 0 |
| rechter Rand angehoben, um 30° | +0,5 g | 0 | +0,87 g |

Liegt das Gerät schräg, verteilt sich die Erdanziehung auf mehrere Achsen. Für eine Neigung um nur eine Achse gilt: Der Messwert in g ist der **Sinus** des Neigungswinkels. 0,5 g bedeutet 30°, 0,3 g ungefähr 17°, 0,1 g knapp 6°.

Sechs Funktionen lesen den Sensor:

| Funktion | liefert |
|---|---|
| `pbc_accel_x()`, `pbc_accel_y()`, `pbc_accel_z()` | Beschleunigung in m/s² |
| `pbc_accel_x_g()`, `pbc_accel_y_g()`, `pbc_accel_z_g()` | dasselbe in g (1.0 = Erdanziehung) |

Alle liefern eine Kommazahl (`float`). Für Spiele sind die Werte in g praktischer, weil 1.0 genau „ganz gekippt“ bedeutet. Bei jedem Aufruf fragt die Bibliothek den Sensor neu.

k17_werte.c

```c
// Der Neigungssensor: alle halbe Sekunde die drei Achsen ausgeben
#include "pbc.h"

int main(void) {
    pbc_init();
    printf("   x m/s2   y m/s2   z m/s2  |  x g    y g    z g\n");
    for (int i = 0; i < 8; i++) {
        float x = pbc_accel_x();
        float y = pbc_accel_y();
        float z = pbc_accel_z();
        printf("%8.2f %8.2f %8.2f  | %5.2f  %5.2f  %5.2f\n",
               x, y, z, pbc_accel_x_g(), pbc_accel_y_g(), pbc_accel_z_g());
        sleep_ms(500);
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
   x m/s2   y m/s2   z m/s2  |  x g    y g    z g
    0.00     0.00     9.81  |  0.00   0.00   1.00
    0.00     0.00     9.81  |  0.00   0.00   1.00
    0.00     0.00     9.81  |  0.00   0.00   1.00
    4.90     0.00     8.53  |  0.50   0.00   0.87
    4.90     0.00     8.53  |  0.50   0.00   0.87
    4.90     0.00     8.53  |  0.50   0.00   0.87
    0.00    -9.81     0.00  |  0.00  -1.00   0.00
    0.00    -9.81     0.00  |  0.00  -1.00   0.00
```

Zuerst lag der PicoBoy flach auf dem Tisch. Nach gut einer Sekunde wurde der rechte Rand um 30° angehoben: x steigt auf 0,5 g, z sinkt auf 0,87 g. Zum Schluss stand das Gerät aufrecht. Diese Ausgabe ist mit genau eingestellten Werten entstanden. Auf deinem Gerät sind die Zahlen nie ganz so glatt: Ein echter Sensor ist nicht perfekt ausgerichtet, und die letzten Stellen schwanken von Messung zu Messung.

> Der Sensor misst nicht nur die Erdanziehung. Bewegst du den PicoBoy ruckartig, kommt die Beschleunigung der Bewegung dazu. Beim Schütteln springen die Werte deshalb stark hin und her. Die Bibliothek stellt den Sensor auf einen Messbereich von ±2 g ein; stärkere Stöße werden abgeschnitten.

Fortlaufend als Balken siehst du die Werte mit diesem Programm:

k17_anzeige.c

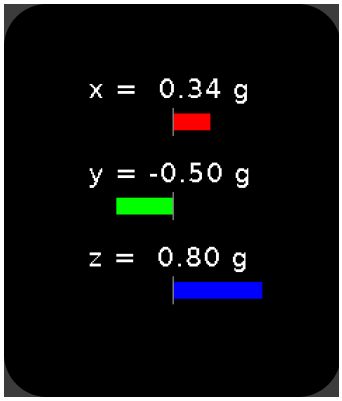
```c
// Die drei Achsen als Zahl und als Balken
#include "pbc.h"

#define MITTE 120               // Nullpunkt der Balken
#define PIXEL_PRO_G 80          // 1 g = 80 Pixel Balkenlaenge

// Zeichnet einen Wert (in g) als Zeile mit Name, Zahl und Balken.
void zeige_achse(char name, float wert, int y, uint16_t farbe) {
    char text[16];
    snprintf(text, sizeof text, "%c = %5.2f g", name, wert);
    pbc_text18(text, 60, y, PBC_WHITE);
    int laenge = (int)(wert * PIXEL_PRO_G);
    if (laenge >= 0) {
        pbc_fill_rect(MITTE, y + 28, laenge, 12, farbe);
    } else {
        pbc_fill_rect(MITTE + laenge, y + 28, -laenge, 12, farbe);
    }
    pbc_vline(MITTE, y + 24, 20, PBC_GRAY);
}

int main(void) {
    pbc_init();
    while (true) {
        pbc_clear(PBC_BLACK);
        zeige_achse('x', pbc_accel_x_g(), 50, PBC_RED);
        zeige_achse('y', pbc_accel_y_g(), 110, PBC_GREEN);
        zeige_achse('z', pbc_accel_z_g(), 170, PBC_BLUE);
        pbc_show();
        sleep_ms(50);
    }
}
```

- Zeile 8: Die Funktion bekommt den Namen der Achse als einzelnes Zeichen (`char`), etwa `'x'`. Einzelne Zeichen stehen in einfachen Anführungszeichen. Der Platzhalter `%c` setzt ein Zeichen ein.
- Zeile 12: 1 g ergibt einen 80 Pixel langen Balken.
- Zeilen 13–17: Ein positiver Wert zeichnet nach rechts von der grauen Mittellinie, ein negativer nach links. Weil ein Rechteck keine negative Breite haben kann, beginnt der Balken dann bei `MITTE + laenge` und ist `-laenge` breit.




Für das Bild war der rechte Rand etwas angehoben (x positiv) und die Joystick-Seite abgesenkt (y negativ).

**Am Gerät ausprobieren:** Kippe den PicoBoy langsam in alle Richtungen, stell ihn hin, dreh ihn um. Stimmen die Vorzeichen mit der Tabelle oben überein? Notiere, welche Werte dein Gerät zeigt, wenn es ruhig auf dem Tisch liegt.

## 17.2 Glätten gegen Zittern

Die Messwerte eines echten Sensors zittern: Selbst wenn das Gerät ruhig liegt, ändern sie sich von Messung zu Messung ein wenig. Eine Anzeige oder eine Spielfigur zittert dann mit. Abhilfe schafft **Glätten**: Man nimmt nicht jeden neuen Wert ganz, sondern nur einen Teil davon.

```
glatt = glatt + (roh - glatt) * 0.2f;
```

`roh` ist der neue Messwert, `glatt` der geglättete Wert. `roh - glatt` ist der Unterschied zwischen beiden. Davon kommen nur 20 % dazu. Ein einzelner Ausreißer verschiebt `glatt` deshalb nur wenig. Bleibt der neue Wert aber länger bestehen, wandert `glatt` Schritt für Schritt dorthin.

Damit die Wirkung deutlich wird, mischt das nächste Programm ein künstliches Zittern zum Messwert:

k17_glaetten.c

```c
// Glaetten: Ein Teil des neuen Messwerts kommt zum alten Wert dazu
#include "pbc.h"

#define ANTEIL 0.2f            // 20 % des Unterschieds pro Messung

int main(void) {
    pbc_init();
    float glatt = pbc_accel_x_g();
    printf("Nr.    roh   glatt\n");
    for (int i = 1; i <= 16; i++) {
        float roh = pbc_accel_x_g();
        // kuenstliches Zittern von -0.10 bis +0.10 g, nur fuer diesen Versuch
        roh = roh + pbc_random(-10, 10) / 100.0f;
        glatt = glatt + (roh - glatt) * ANTEIL;
        printf("%3d  %5.2f  %5.2f\n", i, roh, glatt);
        sleep_ms(100);
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Nr.    roh   glatt
  1   0.07   0.01
  2   0.08   0.03
  3   0.00   0.02
  4  -0.05   0.01
  5   0.07   0.02
  6   0.02   0.02
  7   0.05   0.03
  8   0.07   0.03
  9   0.47   0.12
 10   0.59   0.22
 11   0.46   0.26
```

```
12   0.44   0.30
13   0.46   0.33
14   0.46   0.36
15   0.56   0.40
16   0.47   0.41
```

- Zeile 13: `pbc_random(-10, 10) / 100.0f` ist eine zufällige Zahl von −0,10 bis 0,10. In deinen eigenen Programmen lässt du diese Zeile weg; das Zittern liefert der Sensor selbst.
- Zeile 14: die Glättung.
- In den ersten acht Zeilen liegt das Gerät flach. Die rohen Werte springen um bis zu 0,1 hin und her, die geglätteten bleiben viel näher bei 0.
- Ab Messung 9 ist der rechte Rand um 30° angehoben. Der rohe Wert springt sofort auf etwa 0,5. Der geglättete läuft langsam hinterher und hat ihn nach acht Messungen noch nicht ganz erreicht.

**Wie stark glätten?** Ein kleiner Anteil (etwa 0,1) glättet stark, aber der Wert folgt einer echten Bewegung nur langsam. Ein großer Anteil (etwa 0,5) reagiert schnell, zittert aber mehr. Probiere auf dem Gerät aus, was zu deinem Programm passt.

> Glätten: `glatt = glatt + (roh - glatt) * anteil;` mit einem Anteil zwischen 0 und 1. Je kleiner, desto ruhiger und desto träger.

## 17.3 Steuern durch Kippen

Mit dem Sensor kannst du ein Spiel durch **Kippen** steuern. Es gibt zwei Arten.

**Kippen wie eine Taste.** Der Messwert wird mit einer **Grenze** verglichen. Ist er stärker, gilt das als „gedrückt“. Unterhalb der Grenze reagiert das Programm nicht, auch nicht auf leichtes Zittern.

k17_kipprichtung.c

```c
// Kippen wie eine Taste: Ab einer Grenze gilt das Geraet als gekippt
#include "pbc.h"

#define GRENZE 0.3f            // in g, etwa 17 Grad

int main(void) {
    pbc_init();
    while (true) {
        float x = pbc_accel_x_g();
        float y = pbc_accel_y_g();

        pbc_clear(PBC_BLACK);
        pbc_text18("Tiefer liegt:", 62, 60, PBC_GRAY);
        if (x < -GRENZE) {
            pbc_text24("rechts", 84, 120, PBC_YELLOW);
        } else if (x > GRENZE) {
            pbc_text24("links", 90, 120, PBC_YELLOW);
        } else if (y < -GRENZE) {
            pbc_text24("unten", 88, 120, PBC_YELLOW);
```

```
20          } else if (y > GRENZE) {
21              pbc_text24("oben", 92, 120, PBC_YELLOW);
22          } else {
23              pbc_text24("nichts", 86, 120, PBC_GREEN);
24          }
25          pbc_show();
26          sleep_ms(50);
27      }
28  }
```

- Zeile 4: Erst ab 0,3 g, also etwa 17°, gilt das Gerät als gekippt.
- Zeile 14: Liegt der rechte Rand tiefer, zeigt die x-Achse schräg nach unten, und x wird negativ. Darum steht bei `x < -GRENZE` das Wort „rechts“.
- Zeilen 18–21: genauso für y. Liegt der untere Rand mit dem Joystick tiefer, ist y negativ.



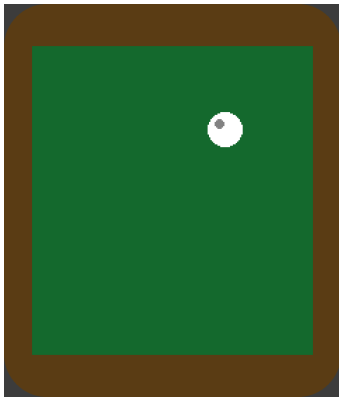

> Reagiert dein Programm „falsch herum“, tausche `<` und `>` oder die Vorzeichen. Prüfe die Richtungen immer zuerst mit dem Anzeigeprogramm aus 17.1.

**Je schräger, desto schneller.** Statt nur „gekippt oder nicht“ kann der Messwert auch die Bewegung bestimmen. Am natürlichsten wirkt es, wenn die Neigung wie die Schwerkraft in Kapitel 14 als **Beschleunigung** wirkt: Sie ändert die Geschwindigkeit, nicht direkt die Position. So rollt eine Kugel wie auf einem Tablett: Sie wird schneller, solange das Tablett schief ist, und rollt weiter, wenn du es wieder gerade hältst.

k17_kugel.c

```
25  int main(void) {
26      pbc_init();
27      float x = 120.0f, y = 140.0f;
28      float vx = 0.0f, vy = 0.0f;
29
30      while (true) {
31          // Die Kugel rollt zur tieferen Seite: dorthin, wo die Achse
32          // nach unten zeigt, also der Messwert negativ ist.
33          vx = (vx - pbc_accel_x_g() * KRAFT) * REIBUNG;
34          vy = (vy - pbc_accel_y_g() * KRAFT) * REIBUNG;
35          x = x + vx;
36          y = y + vy;
37
38          if (x - RADIUS < LINKS) {           // Rand: zurueck, halbes Tempo
39              x = LINKS + RADIUS;
40              vx = -vx * 0.5f;
41          }
```

```
42          if (x + RADIUS > RECHTS) {
43              x = RECHTS - RADIUS;
44              vx = -vx * 0.5f;
45          }
46          if (y - RADIUS < OBEN) {
47              y = OBEN + RADIUS;
48              vy = -vy * 0.5f;
49          }
50          if (y + RADIUS > UNTEN) {
51              y = UNTEN - RADIUS;
52              vy = -vy * 0.5f;
53          }
54
55          pbc_clear(pbc_rgb(90, 60, 30));
56          pbc_fill_rect(LINKS, OBEN, RECHTS - LINKS, UNTEN - OBEN,
57                        pbc_rgb(30, 110, 50));
58          pbc_fill_circle((int)x, (int)y, RADIUS, PBC_WHITE);
59          pbc_fill_circle((int)x - 4, (int)y - 4, 3, PBC_GRAY);
60          pbc_show();
61          bildtakt();
62      }
63  }
```



- Zeilen 33–34: Die Kugel rollt zur **tieferen** Seite, also dorthin, wo der Messwert negativ ist. Deshalb wird er **abgezogen**. `KRAFT` legt fest, wie viel Tempo pro g in jedem Bild dazukommt. `REIBUNG` (0,97) bremst die Kugel in jedem Bild ein wenig, sonst würde sie immer schneller.
- Zeilen 38–53: Am Rand wird die Kugel zurückgesetzt und prallt mit halbem Tempo ab, wie eine schwere Kugel an einer Holzkante.

Die Zeilen 1–24 enthalten die Konstanten und `bildtakt()`. Das Bild zeigt die Kugel, während der rechte Rand etwas tiefer und der Joystick-Rand etwas höher liegt: Sie rollt nach rechts oben.

## Ausprobieren: Die Wasserwaage

Eine Wasserwaage hat eine kleine Luftblase in einem Röhrchen mit Flüssigkeit. Die Blase steigt immer zur **höchsten** Stelle. Unser PicoBoy wird zu einer runden Wasserwaage: Liegt er genau waagrecht, steht die Blase in der Mitte und wird grün, und die grüne LED leuchtet.

k17_wasserwaage.c

```
1  // Die Wasserwaage: Die Blase wandert zur hoeheren Seite
2  #include "pbc.h"
3
4  #define MX 120                  // Mitte der Waage
5  #define MY 135
6  #define GROSS 90                // Radius des grossen Kreises
7  #define BLASE 14                // Radius der Blase
```

```c
#define PIXEL_PRO_G 300         // 0,1 g Neigung = 30 Pixel
#define ANTEIL 0.2f             // Glaetten
#define WAAGRECHT 0.02f         // so wenig Neigung gilt als waagrecht

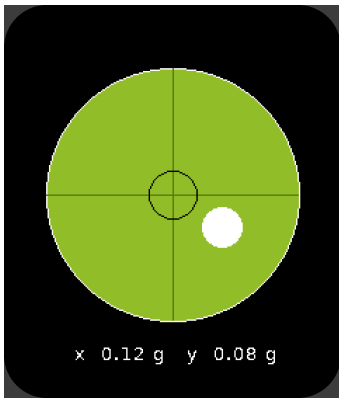
int main(void) {
    pbc_init();
    float gx = 0.0f, gy = 0.0f;             // geglaettete Messwerte
    char text[32];

    while (true) {
        gx = gx + (pbc_accel_x_g() - gx) * ANTEIL;
        gy = gy + (pbc_accel_y_g() - gy) * ANTEIL;

        // Die Blase steigt nach oben: zur Seite, deren Achse nach oben
        // zeigt. Dort ist der Messwert positiv.
        float dx = gx * PIXEL_PRO_G;
        float dy = gy * PIXEL_PRO_G;
        int frei = GROSS - BLASE;           // so weit darf sie wandern
        while (dx * dx + dy * dy > frei * frei) {
            dx = dx * 0.95f;                // in Richtung Mitte holen
            dy = dy * 0.95f;
        }

        bool waagrecht = gx > -WAAGRECHT && gx < WAAGRECHT &&
                         gy > -WAAGRECHT && gy < WAAGRECHT;
        pbc_led_green(waagrecht);

        pbc_clear(PBC_BLACK);
        pbc_fill_circle(MX, MY, GROSS, pbc_rgb(150, 190, 40));
        pbc_circle(MX, MY, GROSS, PBC_WHITE);
        pbc_hline(MX - GROSS, MY, 2 * GROSS, pbc_rgb(90, 120, 20));
        pbc_vline(MX, MY - GROSS, 2 * GROSS, pbc_rgb(90, 120, 20));
        pbc_circle(MX, MY, BLASE + 3, PBC_BLACK);    // Zielkreis
        if (waagrecht) {
            pbc_fill_circle(MX + (int)dx, MY + (int)dy, BLASE, PBC_GREEN);
        } else {
            pbc_fill_circle(MX + (int)dx, MY + (int)dy, BLASE, PBC_WHITE);
        }
        snprintf(text, sizeof text, "x %5.2f g   y %5.2f g", gx, gy);
        pbc_text12(text, 50, 240, PBC_WHITE);
        pbc_show();
        sleep_ms(30);
    }
}
```




- Zeilen 18–19: Beide Messwerte werden geglättet. So steht die Blase ruhig.
- Zeilen 23–24: Hebst du den rechten Rand an, wird x positiv, und die Blase soll nach rechts, also zur hohen Seite. Anders als die Kugel aus 17.3 folgt die Blase deshalb dem Messwert mit **gleichem** Vorzeichen. 0,1 g Neigung verschieben sie um 30 Pixel.
- Zeilen 25–29: Die Blase darf den großen Kreis nicht verlassen. Ist sie zu weit draußen, wird ihr Abstand zur Mitte so lange um 5 % verkürzt, bis sie hineinpasst. Der Vergleich ist derselbe wie bei „Punkt im Kreis“ (Kapitel 16), ohne Wurzel.
- Zeilen 31–32: Weniger als 0,02 g auf beiden Achsen gilt als waagrecht, das ist gut 1°.
- Zeile 33: `pbc_led_green` bekommt direkt den Wahrheitswert: LED an, wenn waagrecht.

Das Bild zeigt die Waage mit angehobenem rechtem Rand und angehobener Joystick-Seite. Auf dem Gerät kannst du damit prüfen, ob dein Tisch gerade ist. Liegt die Blase auf einem Tisch, der wirklich gerade ist, nicht genau in der Mitte, ist dein Sensor etwas schief eingebaut. Das ist normal. Wie man es ausgleicht, steht unter „Zum Verändern“.

**Zum Verändern:** Mach die Wasserwaage empfindlicher (größerer `PIXEL_PRO_G`) oder strenger (kleineres `WAAGRECHT`). Gleiche den Sensor aus: Miss beim Start, wenn das Gerät auf einer geraden Fläche liegt, x und y und ziehe diese Werte danach von jeder Messung ab. Lass einen kurzen Ton erklingen, sobald die Waage waagrecht wird.

## Übungsaufgaben

**Aufgabe 1 ⭐ Wie liegt das Gerät?** Das Programm prüft viermal, wie der PicoBoy liegt. Was gibt es aus, wenn er zuerst flach mit dem Display nach oben liegt, dann mit dem Display nach unten, dann aufrecht steht (Joystick unten) und zuletzt auf der rechten Kante steht, um 45° gekippt? Schau in die Tabelle in 17.1.

k17_aufgabe1.c

```c
int main(void) {
    pbc_init();
    for (int i = 1; i <= 4; i++) {
        float y = pbc_accel_y_g();
        float z = pbc_accel_z_g();
        printf("%d: ", i);
        if (z > 0.8f) {
            printf("liegt, Display oben\n");
        } else if (z < -0.8f) {
            printf("liegt, Display unten\n");
        } else if (y < -0.8f) {
            printf("steht aufrecht\n");
```

```
        } else if (y > 0.8f) {
            printf("steht auf dem Kopf\n");
        } else {
            printf("schraeg oder auf der Seite\n");
        }
        sleep_ms(1000);
    }
```

**Aufgabe 2 ⭐ Kipp-Ampel** Die RGB-LED soll grün leuchten, solange der PicoBoy ungefähr waagrecht liegt. Ist der linke Rand deutlich tiefer, leuchtet sie blau, ist der rechte Rand tiefer, rot. Schalte die LED nur um, wenn sich die Farbe ändert.

**Aufgabe 3 ⭐⭐ Schüttelwürfel** Wenn man den PicoBoy schüttelt, zeigt er einen Würfel mit einer zufälligen Augenzahl von 1 bis 6. Tipp: Merke dir die Werte der letzten Messung. Ist die Summe der drei Unterschiede größer als 0,8 g, wurde geschüttelt. Für den Betrag einer Zahl (ohne Vorzeichen) schreibst du eine kleine Funktion `betrag`. Damit man auch ohne Schütteln testen kann, soll auch Taste A würfeln.

**Aufgabe 4 ⭐⭐ Mittelwert der letzten acht** Eine andere Art zu glätten: Speichere die letzten acht Messwerte in einem Array und nimm ihren Mittelwert. Kommt ein neuer Wert dazu, ersetzt er den ältesten. Nimm `k17_glaetten.c` als Vorlage, mit künstlichem Zittern. Vergleiche: Wie schnell erreicht der Mittelwert die neuen 0,5 g?

**Aufgabe 5 ⭐⭐⭐ Münzen einsammeln** Erweitere die rollende Kugel aus 17.3 zu einem Spiel: An einer zufälligen Stelle liegt eine Münze. Rollt die Kugel darüber, gibt es einen Punkt und einen Ton, und die Münze springt an eine neue Stelle (Kreis gegen Kreis, Kapitel 16). Der Joystick soll die Kugel zusätzlich anschubsen können.

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- Der Neigungssensor misst Beschleunigungen in drei Achsen, auch die Erdanziehung: In Ruhe misst die Achse, die nach oben zeigt, etwa +9,81 m/s² = +1 g.
- Achsen: x nach rechts, y nach unten (zum Joystick), z aus dem Display heraus.
- `pbc_accel_x()` … liefern m/s², `pbc_accel_x_g()` … dasselbe in g. Bei Neigung um eine Achse ist der Wert in g der Sinus des Winkels.
- Glätten: `glatt = glatt + (roh - glatt) * anteil;`
- Kippen wie eine Taste: Messwert mit einer Grenze vergleichen. Kippen als Beschleunigung: Messwert zur Geschwindigkeit addieren, Reibung bremst.
- Ein echter Sensor zittert und ist nie ganz genau ausgerichtet. Richtungen und Grenzen am Gerät ausprobieren.

KAPITEL 18

# Projekt: Ein ganzes Spiel

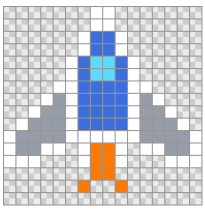
**Worum geht's?** Jetzt setzt du alles aus Teil III zusammen: Bildtakt, Sprites, Kollisionen und Neigung. Es entsteht ein vollständiges Spiel mit Titelbild, Spiel und „Game over", mit Punkten, Leben und Highscore. Du lernst, wie man ein größeres Programm plant und in überschaubare Teile zerlegt.

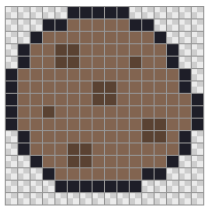
## 18.1 Planen: Zustände (Titel, Spiel, Ende)

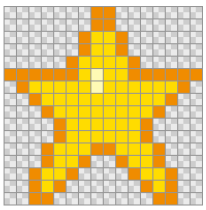
**Die Idee.** Unser Spiel heißt „Sternenfänger". Unten am Bildschirm fliegt ein Raumschiff. Von oben fallen Sterne und Meteore. Sterne fängst du ein, jeder bringt einen Punkt. Meteoren weichst du aus, denn jeder Treffer kostet ein Leben. Nach drei Treffern ist das Spiel vorbei. Mit der Zeit fällt alles schneller. Gesteuert wird mit dem Joystick oder durch Kippen.

Bevor du die erste Zeile schreibst, lohnt es sich, ein paar Fragen auf Papier zu beantworten:

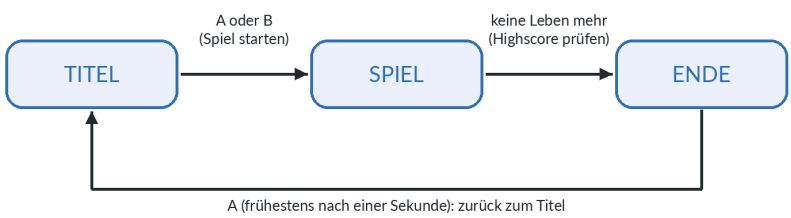
- **Was gibt es im Spiel?** Ein Schiff, mehrere fallende Dinge (Stern oder Meteor), einen Sternenhimmel im Hintergrund, Punkte, Leben, Highscore.
- **Welche Bilder brauche ich?** Schiff, Meteor und Stern, je 16 × 16 Pixel. Den Stern gibt es schon aus Kapitel 15.
- **Was passiert in jedem Bild?** Schiff bewegen, Dinge fallen lassen, Zusammenstöße prüfen, alles zeichnen.
- **Welche Bildschirme gibt es?** Ein Titelbild, das Spiel selbst und einen Bildschirm „Game over".


schiff.png


meteor.png


stern.png

Die letzte Frage ist die wichtigste für den Aufbau des Programms. Das Spiel ist immer in genau einem von drei **Zuständen**: Titel, Spiel oder Ende. Jeder Zustand zeichnet etwas anderes und reagiert anders auf Tasten. Bestimmte Ereignisse lassen das Spiel in einen anderen Zustand wechseln:



So ein Plan heißt **Zustandsdiagramm**, das Programm dazu **Zustandsautomat**. Fast jedes Spiel ist so gebaut, auch große. Oft gibt es noch mehr Zustände: Menü, Pause, nächster Level.

**Das Gerüst.** In C beschreibst du die Zustände mit einem `enum` (Kapitel 13). Eine Variable `modus` merkt sich den aktuellen Zustand, und ein `switch` (Kapitel 8) wählt in jedem Durchgang der Hauptschleife aus, was zu tun ist. Bevor das eigentliche Spiel dazukommt, probieren wir das Gerüst allein aus: drei Bildschirme mit Text, zwischen denen Tasten umschalten.

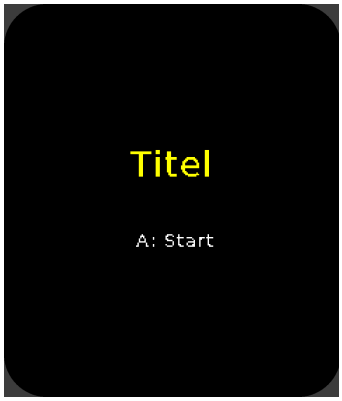
k18_geruest.c

```c
// Das Geruest des Spiels: drei Zustaende, noch ohne Spiel
#include "pbc.h"

typedef enum {
    TITEL,
    SPIEL,
    ENDE
} zustand;

int main(void) {
    pbc_init();
    zustand modus = TITEL;
    while (true) {
        pbc_clear(PBC_BLACK);
        switch (modus) {
            case TITEL:
                pbc_text24("Titel", 90, 100, PBC_YELLOW);
                pbc_text12("A: Start", 94, 160, PBC_WHITE);
                if (pbc_was_pressed_a()) {
                    modus = SPIEL;
                }
                break;
            case SPIEL:
                pbc_text24("Spiel", 90, 100, PBC_GREEN);
                pbc_text12("B: verlieren", 80, 160, PBC_WHITE);
                if (pbc_was_pressed_b()) {
                    modus = ENDE;
                }
                break;
            case ENDE:
                pbc_text24("Ende", 96, 100, PBC_RED);
                pbc_text12("A: zum Titel", 80, 160, PBC_WHITE);
                if (pbc_was_pressed_a()) {
                    modus = TITEL;
                }
                break;
        }
        pbc_show();
        sleep_ms(20);
    }
}
```

- Zeilen 4–8: die drei Zustände als `enum`. Der neue Typ heißt `zustand`.
- Zeile 12: Das Programm beginnt im Titel.
- Zeilen 15–37: Je nach `modus` wird ein anderer Teil ausgeführt. Jeder Teil zeichnet seinen Bildschirm und prüft, ob eine Taste den Zustand wechselt.
- Zeilen 38–39: `pbc_show()` und die Pause stehen **nach** dem `switch`. Sie gelten für alle Zustände gleich.




Ein Zustandswechsel ist nur eine Zuweisung, zum Beispiel `modus = SPIEL;`. Im nächsten Durchgang der Schleife landet das Programm dann im anderen `case`.

> Teste ein großes Programm in kleinen Schritten. Funktioniert das Gerüst, füge zuerst nur das Schiff hinzu, dann ein fallendes Ding, dann die Kollision, dann Punkte und Leben. Nach jedem Schritt bauen und ausprobieren. Ein Fehler steckt dann fast immer im zuletzt geschriebenen Teil.

## 18.2 Schritt für Schritt bauen

Das fertige Spiel steht in `k18_spiel.c`. Es ist rund 375 Zeilen lang; davon sind 120 Zeilen die drei Bilder. Wir gehen es Abschnitt für Abschnitt durch. Kommentarzeilen mit `// ----` trennen die Abschnitte, damit man sich im Quelltext zurechtfindet.

**Konstanten und Bilder.**

k18_spiel.c

```
1 // Sternenfaenger: Sterne fangen, Meteoren ausweichen
2 #include "pbc.h"
3
4 #define BILD_MS 40
5 #define ANZAHL 6                // so viele Dinge fallen gleichzeitig
6 #define HIMMEL 30               // Sterne im Hintergrund
7 #define SCHIFF_Y 236            // obere Kante des Schiffs
8 #define SCHUTZ_MS 1500          // unverwundbar nach einem Treffer
9 #define START_LEBEN 3
```

Alle Zahlen, an denen man das Spiel einstellt, stehen oben als Konstanten. Die Zeilen 11–131 enthalten die drei Bilder: `schiff.png` und `meteor.png` mit den Namen `schiff` und `meteor` und `stern.png` aus Kapitel 15, alle mit `bild2c.html` umgewandelt (Kapitel 15.1).

**Die Daten des Spiels.**

k18_spiel.c

```
133 // ---- Daten des Spiels ---------------------------------
134
135 typedef enum {
136     TITEL,
137     SPIEL,
138     ENDE
139 } zustand;
140
141 typedef struct {
142     int x, y;
143     int w, h;
144 } rechteck;
145
146 typedef struct {
147     float x, y;                 // linke obere Ecke des Bilds
148     float tempo;                // Pixel pro Bild nach unten
149     bool ist_stern;             // Stern oder Meteor?
150 } ding;
151
152 zustand modus = TITEL;
153 bool mit_kippen = false;        // Steuerung: Joystick oder Neigung
154 float schiff_x;
155 ding dinge[ANZAHL];
156 int himmel_x[HIMMEL], himmel_y[HIMMEL];
157 int punkte;
158 int leben;
159 int highscore = 0;              // nur im RAM: weg nach dem Ausschalten
160 bool neuer_rekord;
161 uint32_t spiel_start;           // pbc_millis() beim Start des Spiels
162 uint32_t schutz_bis;
163 uint32_t ende_seit;             // pbc_millis() beim Spielende
```

- Zeilen 135–139: die Zustände, wie im Gerüst.
- Zeilen 146–150: Ein `ding` ist etwas, das herunterfällt. `ist_stern` unterscheidet Stern und Meteor. Beide verhalten sich gleich, bis sie das Schiff berühren.
- Zeilen 152–163: Alles, was sich das Spiel merken muss, steht in **globalen Variablen**. So können alle Funktionen darauf zugreifen. Die Arrays liegen damit auch nicht auf dem kleinen Stack (Kapitel 19).
- Zeile 156: Für den Hintergrund merkt sich das Spiel die Positionen von 30 Sternen am Himmel in zwei Arrays.
- Zeilen 161–163: Drei Zeitpunkte aus `pbc_millis()`: wann das Spiel begann (für die steigende Geschwindigkeit), bis wann das Schiff nach einem Treffer geschützt ist (Kapitel 16.3) und wann das Spiel endete.

**Hilfsfunktionen.** Die Zeilen 165–194 enthalten Bekanntes: `bildtakt()` aus Kapitel 14, `rechtecke_beruehren()` aus Kapitel 16 und zwei neue kleine Funktionen:

k18_spiel.c

```
184 // Text waagrecht in der Mitte des Bildschirms
185 void text_mitte18(const char text[], int y, uint16_t farbe) {
186     pbc_text18(text, (PBC_WIDTH - pbc_text18_width(text)) / 2, y, farbe);
187 }
188
189 void himmel_zeichnen(void) {
190     pbc_clear(pbc_rgb(5, 5, 25));
191     for (int i = 0; i < HIMMEL; i++) {
192         pbc_pixel(himmel_x[i], himmel_y[i], PBC_GRAY);
193     }
194 }
```

- `text_mitte18` schreibt einen Text in der Mitte des Bildschirms. Die Breite des Texts liefert `pbc_text18_width`. Was übrig bleibt, wird halbiert: links und rechts gleich viel Platz. Ein Text als Parameter wird als `const char text[]` angegeben, so wie ein Array (Kapitel 12). Was dabei genau übergeben wird, erklärt Kapitel 19.
- `himmel_zeichnen` löscht den Bildschirm dunkelblau und setzt die Hintergrundsterne als einzelne graue Pixel. Titel, Spiel und Ende benutzen sie alle.

**Ein Spiel beginnen und beenden.**

k18_spiel.c

```
198 // Setzt Ding i oberhalb des Bildschirms neu ab. Mit der Zeit fallen
199 // die Dinge schneller: alle 20 Sekunden um 1 Pixel pro Bild.
200 void ding_neu(int i) {
201     float sekunden = (pbc_millis() - spiel_start) / 1000.0f;
202     dinge[i].x = pbc_random(20, 204);
203     dinge[i].y = pbc_random(-300, -20);
204     dinge[i].tempo = pbc_random(20, 35) / 10.0f + sekunden / 20.0f;
205     dinge[i].ist_stern = pbc_random(1, 3) == 1;     // jedes dritte
206 }
207
208 void spiel_starten(bool kippen) {
209     mit_kippen = kippen;
210     spiel_start = pbc_millis();
211     schiff_x = 112;
212     punkte = 0;
213     leben = START_LEBEN;
214     schutz_bis = 0;
215     for (int i = 0; i < ANZAHL; i++) {
216         ding_neu(i);
217     }
218     modus = SPIEL;
219 }
220
221 void spiel_beenden(void) {
222     neuer_rekord = punkte > highscore;
```

```
223     if (neuer_rekord) {
224         highscore = punkte;
225     }
226     ende_seit = pbc_millis();
227     pbc_tone(110, 600);
228     modus = ENDE;
229 }
```

- Zeilen 200–206: `ding_neu` setzt ein Ding an eine zufällige Stelle über dem Bildschirm. Die Geschwindigkeit liegt zwischen 2,0 und 3,5 Pixeln pro Bild, dazu kommt 1 Pixel für jede 20 Sekunden, die das Spiel schon läuft. So wird es allmählich schwerer. `pbc_random(1, 3) == 1` ist in einem Drittel der Fälle wahr: Jedes dritte Ding ist ein Stern.
- Zeilen 208–219: `spiel_starten` stellt alles auf Anfang und wechselt in den Zustand `SPIEL`. Der Parameter sagt, ob mit Kippen gesteuert wird.
- Zeilen 221–229: `spiel_beenden` prüft den Highscore (Abschnitt 18.3), merkt sich die Zeit, spielt einen tiefen Ton und wechselt zu `ENDE`.

**Das Schiff bewegen.**

k18_spiel.c

```
231 void schiff_bewegen(void) {
232     if (mit_kippen) {
233         schiff_x = schiff_x - pbc_accel_x_g() * 12;
234     } else {
235         if (pbc_pressed_left()) {
236             schiff_x = schiff_x - 5;
237         }
238         if (pbc_pressed_right()) {
239             schiff_x = schiff_x + 5;
240         }
241     }
242     if (schiff_x < 20) {
243         schiff_x = 20;
244     }
245     if (schiff_x > 204) {
246         schiff_x = 204;
247     }
248 }
```

Mit dem Joystick fährt das Schiff 5 Pixel pro Bild. Beim Kippen bestimmt die Neigung das Tempo: 0,3 g (etwa 17°) ergeben 3,6 Pixel pro Bild. Das Minuszeichen kennst du aus Kapitel 17.3: Ist der rechte Rand tiefer, ist x negativ, und das Schiff soll nach rechts. Hier steuert die Neigung direkt die Geschwindigkeit, ohne Schwung. Das ist für ein Ausweichspiel genauer als die rollende Kugel. Die Zeilen 242–247 halten das Schiff auf dem Bildschirm.

### Fallen und Zusammenstöße.

k18_spiel.c

```
250 void dinge_bewegen(void) {
251     // Hitboxen: etwas kleiner als die Bilder
252     rechteck schiff_box = { (int)schiff_x + 2, SCHIFF_Y + 1, 12, 13 };
253     for (int i = 0; i < ANZAHL; i++) {
254         dinge[i].y = dinge[i].y + dinge[i].tempo;
255         rechteck box = { (int)dinge[i].x + 2, (int)dinge[i].y + 2, 12, 12 };
256         if (dinge[i].y > PBC_HEIGHT) {
257             ding_neu(i);                          // unten hinaus
258         } else if (rechtecke_beruehren(schiff_box, box)) {
259             if (dinge[i].ist_stern) {
260                 punkte++;
261                 pbc_tone(1320, 50);
262                 ding_neu(i);
263             } else if (pbc_millis() >= schutz_bis) {
264                 leben--;
265                 schutz_bis = pbc_millis() + SCHUTZ_MS;
266                 pbc_tone(150, 250);
267                 ding_neu(i);
268             }
269         }
270     }
271 }
```

- Zeilen 252 und 255: Die Hitboxen sind etwas kleiner als die Bilder (Kapitel 16.1). So zählt ein Treffer erst, wenn sich die Figuren wirklich berühren, nicht schon ihre durchsichtigen Ecken.
- Zeilen 256–257: Was unten hinausfällt, kommt oben neu.
- Zeilen 259–262: Ein Stern bringt einen Punkt und einen hellen Ton.
- Zeilen 263–268: Ein Meteor kostet ein Leben, aber nur, wenn die Schutzzeit vorbei ist. Danach ist das Schiff 1,5 Sekunden lang unverwundbar. Getroffene Dinge kommen sofort neu, damit ein Zusammenstoß nur einmal zählt (Kapitel 16.3).

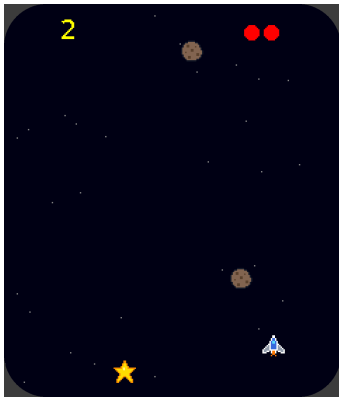
### Zeichnen und der Zustand `SPIEL`.

k18_spiel.c

```
273 void spiel_zeichnen(void) {
274     char text[20];
275     himmel_zeichnen();
276     for (int i = 0; i < ANZAHL; i++) {
277         if (dinge[i].ist_stern) {
278             pbc_draw_image(&stern, (int)dinge[i].x, (int)dinge[i].y);
279         } else {
280             pbc_draw_image(&meteor, (int)dinge[i].x, (int)dinge[i].y);
281         }
282     }
283     bool geschuetzt = pbc_millis() < schutz_bis;
```

```
284     if (!geschuetzt || (pbc_millis() / 100) % 2 == 0) {
285         pbc_draw_image(&schiff, (int)schiff_x, SCHIFF_Y);
286     }
287     snprintf(text, sizeof text, "%d", punkte);
288     pbc_text18(text, 40, 8, PBC_YELLOW);
289     for (int i = 0; i < leben; i++) {
290         pbc_fill_circle(190 - i * 14, 20, 5, PBC_RED);
291     }
292 }
293
294 void spiel(void) {
295     // der Hintergrund zieht langsam nach unten: Das Schiff fliegt
296     for (int i = 0; i < HIMMEL; i++) {
297         himmel_y[i] = (himmel_y[i] + 1) % PBC_HEIGHT;
298     }
299     schiff_bewegen();
300     dinge_bewegen();
301     spiel_zeichnen();
302     if (leben == 0) {
303         spiel_beenden();
304     }
305 }
```

- Zeilen 276–282: Jedes Ding wird mit seinem Bild gezeichnet.
- Zeilen 283–286: Das Schiff blinkt während der Schutzzeit.
- Zeilen 287–291: oben links die Punkte, oben rechts ein roter Punkt für jedes Leben.
- Zeilen 294–305: Die Funktion `spiel` ist der ganze Zustand `SPIEL`. Sie lässt den Himmel um 1 Pixel nach unten ziehen; `% PBC_HEIGHT` holt Sterne, die unten verschwinden, wieder nach oben. Das sieht aus, als flöge das Schiff. Danach: bewegen, prüfen, zeichnen. Sind alle Leben weg, endet das Spiel.




Das Bild zeigt das Spiel nach 30 Sekunden: zwei Sterne gefangen, ein Leben verloren.

**Titel und Ende.**

k18_spiel.c

```
309 void titel(void) {
310     char text[24];
311     himmel_zeichnen();
312     text_mitte18("Sternenfaenger", 60, PBC_YELLOW);
313     pbc_draw_image(&stern, 92, 100);
314     pbc_draw_image(&schiff, 112, 100);
315     pbc_draw_image(&meteor, 132, 100);
316     snprintf(text, sizeof text, "Highscore: %d", highscore);
317     text_mitte18(text, 140, PBC_WHITE);
318     text_mitte18("A: Joystick", 190, PBC_CYAN);
319     text_mitte18("B: Kippen", 216, PBC_CYAN);
```

```
320     if (pbc_was_pressed_a()) {
321         spiel_starten(false);
322     } else if (pbc_was_pressed_b()) {
323         spiel_starten(true);
324     }
325 }
```

Das Titelbild zeigt den Namen, die drei Figuren, den Highscore und die Steuerung. A startet mit dem Joystick, B mit Kippen.




k18_spiel.c

```
327 void ende(void) {
328     char text[24];
329     himmel_zeichnen();
330     text_mitte18("Game over", 70, PBC_RED);
331     snprintf(text, sizeof text, "Punkte: %d", punkte);
332     text_mitte18(text, 120, PBC_WHITE);
333     if (neuer_rekord) {
334         text_mitte18("Neuer Rekord!", 150, PBC_YELLOW);
335     } else {
336         snprintf(text, sizeof text, "Highscore: %d", highscore);
337         text_mitte18(text, 150, PBC_GRAY);
338     }
339     // Erst nach einer Sekunde weiter, damit ein Druck aus dem Spiel
340     // nicht gleich den Titel ueberspringt.
341     if (pbc_millis() - ende_seit > 1000) {
342         text_mitte18("A: weiter", 210, PBC_CYAN);
343         if (pbc_was_pressed_a()) {
344             modus = TITEL;
345         }
346     } else {
347         pbc_was_pressed_a();                // Druecke verwerfen
348         pbc_was_pressed_b();
349     }
350 }
```

- Zeilen 333–338: Bei einem neuen Rekord erscheint „Neuer Rekord!“, sonst der bisherige Highscore.
- Zeilen 341–349: Wer im Spiel gerade wild auf die Tasten drückt, soll das Ende nicht aus Versehen gleich wegdrücken. Deshalb geht es erst nach einer Sekunde mit A weiter. Bis dahin holt das Programm Tastendrücke mit `pbc_was_pressed_a()` und `pbc_was_pressed_b()` ab, ohne etwas damit zu tun: Sie werden verworfen.




**Das Hauptprogramm.**

k18_spiel.c

```
354 int main(void) {
355     pbc_init();
356     for (int i = 0; i < HIMMEL; i++) {
357         himmel_x[i] = pbc_random(0, PBC_WIDTH - 1);
358         himmel_y[i] = pbc_random(0, PBC_HEIGHT - 1);
359     }
360     while (true) {
361         switch (modus) {
362             case TITEL:
363                 titel();
364                 break;
365             case SPIEL:
366                 spiel();
367                 break;
368             case ENDE:
369                 ende();
370                 break;
371         }
372         pbc_show();
373         bildtakt();
374     }
375 }
```

Das Hauptprogramm ist kurz, weil jeder Zustand seine eigene Funktion hat. Es legt einmal die Hintergrundsterne fest und wiederholt dann für immer: Zustand ausführen, `pbc_show()`, `bildtakt()`. Genau wie das Gerüst aus 18.1, nur ruft jeder `case` jetzt eine Funktion auf.

> Ein großes Spiel ist eine Sammlung kleiner Funktionen. Jede erledigt eine Aufgabe und ist kurz genug, um sie auf einen Blick zu verstehen. Ein Zustandsautomat mit `enum` und `switch` hält das Ganze zusammen.

## 18.3 Highscore anzeigen

Der Highscore ist die beste Punktzahl seit dem Einschalten. Dafür reicht eine einzige globale Variable:

k18_spiel.c

```
159 int highscore = 0;              // nur im RAM: weg nach dem Ausschalten
160 bool neuer_rekord;
```

Am Ende jedes Spiels vergleicht `spiel_beenden` (Zeilen 222–225) die Punkte mit dem Highscore. Sind es mehr, wird der Highscore ersetzt und `neuer_rekord` auf `true` gesetzt. Das Titelbild und der Ende-Bildschirm zeigen den Wert mit `snprintf` an.




Das Bild zeigt das Titelbild nach dem ersten Spiel: 6 Punkte sind jetzt der Highscore.

Eine Schwäche hat dieser Highscore: Er steht nur im **Arbeitsspeicher**. Drückst du RES, spielst ein neues Programm auf oder sind die Batterien leer, beginnt er wieder bei 0. Der Arbeitsspeicher vergisst beim Ausschalten alles. Damit der Highscore auch das übersteht, muss er in den **Flash-Speicher** geschrieben werden, wo auch das Programm liegt. Wie das geht und worauf man dabei achten muss, zeigt Kapitel 25.

**Zum Verändern:** Stelle das Spiel nach deinem Geschmack ein: andere Zahlen bei `ANZAHL`, `SCHUTZ_MS` oder `START_LEBEN`, eine schnellere Steigerung in `ding_neu`. Zeichne eigene Bilder für Schiff und Meteor. Gib Töne und LEDs mehr zu tun, zum Beispiel eine kleine Melodie beim neuen Rekord oder die RGB-LED rot bei einem Treffer.

## Übungsaufgaben

> **Aufgabe 1** ⭐ **Einfacher und schwerer** Welche Zeilen in `k18_spiel.c` musst du ändern, damit das Spiel a) mit fünf Leben beginnt, b) acht Dinge gleichzeitig fallen, c) die Geschwindigkeit nur alle 40 Sekunden um 1 Pixel pro Bild steigt, d) nur jedes vierte Ding ein Stern ist? Probiere es aus.

> **Aufgabe 2** ⭐⭐ **Pause** Füge einen vierten Zustand `PAUSE` hinzu. Im Spiel hält Taste B das Spiel an; das Spielbild bleibt stehen, darüber steht „Pause“. Ein weiterer Druck auf B setzt das Spiel fort. Achtung: Während der Pause läuft `pbc_millis()` weiter. Sorge dafür, dass die Pause nicht zur Spielzeit zählt und die Schutzzeit nicht verbraucht.

**Aufgabe 3 ⭐⭐ Bestenliste** Statt eines einzigen Highscores soll sich das Spiel die drei besten Ergebnisse merken (in einem Array) und sie auf dem Titelbild als Liste zeigen. Ein neues Ergebnis wird an der richtigen Stelle einsortiert, die schlechteren rücken eine Stelle nach unten.

**Aufgabe 4 ⭐⭐⭐ Extraleben** Ab und zu (etwa jedes dreißigste Ding) soll ein Herz herunterfallen. Wer es fängt, bekommt ein Leben dazu, höchstens bis fünf. Das Bild `herz.png` liegt bei den Bildern zu diesem Kapitel. Tipp: Ersetze `bool ist_stern` durch ein `enum` mit drei Arten: Meteor, Stern und Herz.

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- Erst planen: Was gibt es im Spiel, was passiert in jedem Bild, welche Bildschirme gibt es?
- **Zustände** (Titel, Spiel, Ende) als `enum`, eine Variable `modus`, ein `switch` in der Hauptschleife. Ein Wechsel ist eine Zuweisung: `modus = ENDE;`.
- Jeder Zustand bekommt eine eigene Funktion; `pbc_show()` und `bildtakt()` stehen einmal am Ende der Hauptschleife.
- Größere Programme in Abschnitte gliedern und Schritt für Schritt bauen und testen.
- Der Highscore in einer Variablen bleibt nur bis zum Ausschalten oder Neustart erhalten. Dauerhaft speichern: Kapitel 25.

KAPITEL 19

# Zeiger und Speicher

**Worum geht's?** Mit diesem Kapitel beginnt der Blick hinter die Kulissen. Jede Variable liegt an einer bestimmten Stelle im Speicher, und diese Stelle hat eine Nummer: ihre **Adresse**. Du lernst **Zeiger** kennen, also Variablen, die sich eine Adresse merken. Mit ihnen können Funktionen die Variablen ihres Aufrufers ändern, und du verstehst endlich, was hinter Arrays und Texten steckt. Danach schaust du dir an, wie der Speicher des PicoBoy aufgeteilt ist, was im RAM und was im Flash landet und was passiert, wenn der Platz nicht reicht.

## 19.1 Adressen und Zeiger `&`, `*`

Der Arbeitsspeicher (RAM) ist eine lange Reihe von Speicherzellen. Jede Zelle fasst ein Byte, und jede hat eine Nummer, ihre **Adresse**, so wie jedes Haus in einer Straße eine Hausnummer hat. Eine Variable vom Typ `int` belegt 4 Bytes, also vier Zellen nebeneinander. Ihre Adresse ist die Nummer der ersten dieser Zellen.

Mit dem Zeichen `&` vor einem Variablennamen bekommst du die Adresse der Variable. Eine Variable, die sich eine Adresse merkt, heißt **Zeiger** (englisch *pointer*). Man legt sie mit einem `*` vor dem Namen an:

k19_zeiger.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int punkte = 10;
    int *zeiger = &punkte;   // zeiger merkt sich, wo punkte liegt

    printf("punkte  = %d\n", punkte);
    printf("*zeiger = %d\n", *zeiger);

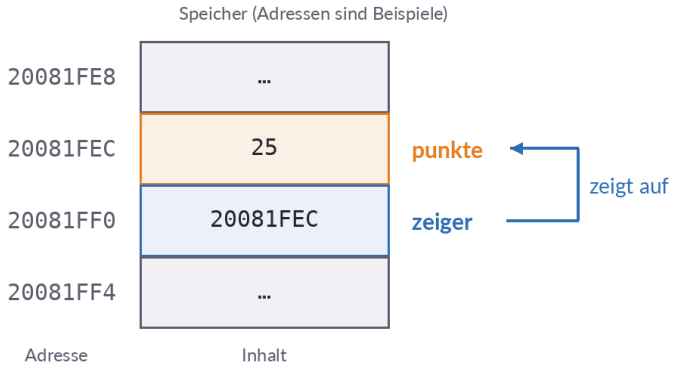
    *zeiger = 25;            // schreibt dorthin, wo zeiger hinzeigt
    printf("punkte  = %d\n", punkte);

    punkte = punkte + 1;
    printf("*zeiger = %d\n", *zeiger);

    while (true) {
        sleep_ms(1000);
    }
}
```

```
Serial Monitor
punkte  = 10
*zeiger = 10
punkte  = 25
*zeiger = 26
```

- Zeile 6 legt den Zeiger `zeiger` an. Sein Typ heißt `int *`, gelesen „Zeiger auf `int`“. Er bekommt die Adresse von `punkte`. Man sagt: `zeiger` **zeigt auf** `punkte`.
- In Zeile 9 steht `*zeiger`. Der Stern vor einem Zeiger heißt: Nimm nicht den Zeiger selbst, sondern den Wert an der Adresse, auf die er zeigt. Das ist der Wert von `punkte`, also 10.
- Zeile 11 schreibt über den Zeiger: `*zeiger = 25` ändert die Variable, auf die `zeiger` zeigt. Danach ist `punkte` 25, obwohl in dieser Zeile das Wort `punkte` gar nicht vorkommt.
- Umgekehrt geht es auch: Zeile 14 ändert `punkte`, und `*zeiger` liefert sofort den neuen Wert 26. Es gibt ja nur **eine** Variable; der Zeiger ist bloß ein zweiter Weg zu ihr.




Das Bild zeigt den Speicher nach Zeile 11. Auch der Zeiger ist eine Variable mit einer eigenen Adresse. Sein Inhalt ist eine Zahl: die Adresse von `punkte`. Adressen schreibt man meist **hexadezimal** (zur Basis 16, mit den Ziffern 0–9 und A–F), so wie du es von den Farben kennst.

> - `&name` ist die **Adresse** einer Variable.
> - `int *z;` legt einen **Zeiger** auf `int` an.
> - `*z` ist die Variable, **auf die `z` zeigt**: Du kannst sie lesen und beschreiben.

Der Stern hat also zwei Aufgaben: In einer Deklaration wie `int *z` kennzeichnet er einen Zeiger. Vor einem Zeiger in einer Rechnung oder Zuweisung wie `*z = 25` heißt er „folge dem Zeiger“. Zwischen zwei Zahlen wie in `3 * 4` ist er wie immer das Malzeichen.

**Adressen sehen.** `printf` gibt eine Adresse mit dem Platzhalter `%p` aus. Er erwartet einen Zeiger vom Typ `void *`: einen Zeiger, bei dem offen bleibt, worauf er zeigt. Mit `(void *)` davor wandelst du jeden Zeiger in diesen Typ um. Das Programm unten zeigt drei Adressen und die Größe eines `int` und eines Zeigers:

k19_adressen.c

```c
#include "pbc.h"

int rekord = 500;                              // globale Variable
const uint16_t FARBEN[3] = { PBC_RED, PBC_GREEN, PBC_BLUE };

int main(void) {
    pbc_init();
    int punkte = 10;                           // lokale Variable

    printf("punkte liegt bei %p\n", (void *)&punkte);
    printf("rekord liegt bei %p\n", (void *)&rekord);
    printf("FARBEN liegt bei %p\n", (void *)FARBEN);
    printf("Ein int braucht %d Byte, ein Zeiger %d Byte.\n",
           (int)sizeof(int), (int)sizeof(int *));

    while (true) {
        sleep_ms(1000);
    }
}
```

Auf dem PicoBoy gibt `%p` jede Adresse mit acht hexadezimalen Ziffern aus, ohne `0x` davor. Die erste Adresse beginnt mit `20081`, die zweite mit `2000`, die dritte mit `1000`. Warum die drei Variablen so weit auseinanderliegen, erklärt Abschnitt 19.4. Die letzte Zeile meldet: Ein `int` braucht 4 Bytes und ein Zeiger auch 4 Bytes. Der RP2350 ist ein **32-Bit-Prozessor**, seine Adressen sind 32-Bit-Zahlen.

> Dieses Programm gibt auf einem PC andere Zahlen aus: Dort sind Adressen meist 8 Bytes lang und liegen ganz woanders. Die Zahlen gelten nur für das Gerät, für das das Programm gebaut wurde.

**Der Zeiger ins Nichts: `NULL`.** Ein Zeiger, der auf nichts zeigen soll, bekommt den Wert `NULL`. Du kennst das von den Bildern aus Kapitel 15: Hat ein Bild keine Maske, steht dort `NULL`. Bevor eine Funktion einem solchen Zeiger folgt, muss sie prüfen, ob er `NULL` ist. Genau das tut `pbc_draw_image` mit `image->mask == NULL`.

> Folge nie einem Zeiger, dem du keine Adresse gegeben hast, und nie einem `NULL`-Zeiger. Ein Zeiger ohne Startwert enthält irgendeine Zahl. `*z = 5` schreibt dann an irgendeine Stelle im Speicher. Der Compiler meldet das nicht immer, und zur Laufzeit gibt es keine Fehlermeldung (wie bei Arrays, Kapitel 12.5): Das Programm macht still etwas Falsches oder bleibt stehen.

## 19.2 Zeiger als Parameter: Funktionen, die Werte ändern

In Kapitel 11 hast du gelernt: Eine Funktion bekommt **Kopien** ihrer Argumente. Ändert sie einen Parameter, ändert sie nur ihre Kopie. Daran scheitert eine Funktion, die zwei Variablen vertauschen soll:

k19_tausche.c

```
 1 #include "pbc.h"
 2
 3 // bekommt nur Kopien der Werte
 4 void tausche_kopien(int a, int b) {
 5     int merker = a;
 6     a = b;
 7     b = merker;
 8 }
 9
10 // bekommt die Adressen und arbeitet mit den Originalen
11 void tausche(int *a, int *b) {
12     int merker = *a;
13     *a = *b;
14     *b = merker;
15 }
16
17 int main(void) {
18     pbc_init();
19     int links = 3;
20     int rechts = 8;
21     printf("vorher:          links = %d, rechts = %d\n", links, rechts);
22
23     tausche_kopien(links, rechts);
24     printf("tausche_kopien:  links = %d, rechts = %d\n", links, rechts);
25
26     tausche(&links, &rechts);
27     printf("tausche:         links = %d, rechts = %d\n", links, rechts);
28
29     while (true) {
30         sleep_ms(1000);
31     }
32 }
```

Serial Monitor

```
vorher:          links = 3, rechts = 8
tausche_kopien:  links = 3, rechts = 8
tausche:         links = 8, rechts = 3
```

- `tausche_kopien` (Zeilen 4 bis 8) vertauscht ihre Kopien `a` und `b`. `links` und `rechts` in `main` bleiben, wie sie sind.
- `tausche` (Zeilen 11 bis 15) bekommt dagegen die **Adressen** der beiden Variablen. Mit `*a` und `*b` liest und schreibt sie die Originale.
- Beim Aufruf in Zeile 26 stehen deshalb `&links` und `&rechts`: Die Funktion braucht die Adressen, nicht die Werte.

Eine Funktion kann mit `return` nur **einen** Wert zurückgeben. Über Zeiger kann sie beliebig viele Variablen des Aufrufers füllen. Das kennst du schon: `pbc_input(frage, name, sizeof name)` aus Kapitel 4 schreibt die Eingabe in dein Array `name`. In `pbc.h` steht dafür der Parameter `char *text`, ein Zeiger auf das erste Zeichen deines Arrays.

**`const` bei Zeigern.** In `pbc.h` steht bei `pbc_text` der Parameter `const char *s`. Das `const` ist ein Versprechen: Die Funktion liest über diesen Zeiger nur und ändert nichts. Versucht sie es doch, meldet der Compiler einen Fehler. Schreib `const` immer dazu, wenn deine Funktion nur lesen will. Dann sieht jeder am Funktionskopf, was sie mit seinen Daten tut.

**Zeiger auf Strukturen.** Besonders oft übergibt man Zeiger auf Strukturen. Die Funktion arbeitet dann am Original, und der Prozessor muss die Struktur nicht erst kopieren:

k19_baelle.c

```c
#include "pbc.h"

typedef struct {
    int x, y, dx, dy, r;
    uint16_t farbe;
} ball;

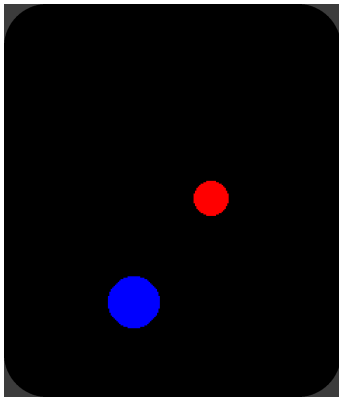
// Ein Zeiger auf einen Ball: die Funktion bewegt das Original.
void bewege(ball *b) {
    b->x += b->dx;
    b->y += b->dy;
    if (b->x < b->r || b->x > PBC_WIDTH - 1 - b->r) {
        b->dx = -b->dx;
    }
    if (b->y < b->r || b->y > PBC_HEIGHT - 1 - b->r) {
        b->dy = -b->dy;
    }
}

void zeichne(const ball *b) {
    pbc_fill_circle(b->x, b->y, b->r, b->farbe);
}

int main(void) {
    pbc_init();
    ball rot = { 60, 80, 3, 2, 12, PBC_RED };
    ball blau = { 150, 200, -2, 4, 18, PBC_BLUE };
    while (true) {
        bewege(&rot);
        bewege(&blau);
        pbc_clear(PBC_BLACK);
```

```
32          zeichne(&rot);
33          zeichne(&blau);
34          pbc_show();
35          sleep_ms(20);
36      }
37  }
```

- `bewege` in Zeile 9 bekommt einen Zeiger auf einen `ball`.
- `b->x` heißt: das Element `x` der Struktur, auf die `b` zeigt. Es ist eine Kurzschreibweise für `(*b).x`. Die Klammern wären nötig, weil der Punkt stärker bindet als der Stern; deshalb benutzt jeder den Pfeil `->`.
- `zeichne` in Zeile 20 will den Ball nur lesen und verspricht das mit `const ball *b`.
- In `main` stehen beim Aufruf `&rot` und `&blau`. So hast du auch in Kapitel 15 `pbc_draw_image(&bild, x, y)` geschrieben: `pbc_draw_image` bekommt einen Zeiger auf dein Bild.



> Soll eine Funktion eine Variable des Aufrufers **ändern**, bekommt sie einen Zeiger: `void f(int *wert)`, Aufruf `f(&x)`. Bei Strukturen greift sie mit `zeiger->element` auf die Teile zu.

## 19.3 Arrays und Zeiger

Arrays und Zeiger sind in C eng verwandt. Der Name eines Arrays steht für die **Adresse seines ersten Elements**. `int *p = punkte;` ist dasselbe wie `int *p = &punkte[0];`.

k19_array_zeiger.c

```c
#include "pbc.h"

// werte zeigt auf das erste Element; die Anzahl muss extra mit.
int summe(const int *werte, int anzahl) {
    int s = 0;
    for (int i = 0; i < anzahl; i++) {
        s += werte[i];
    }
    return s;
}

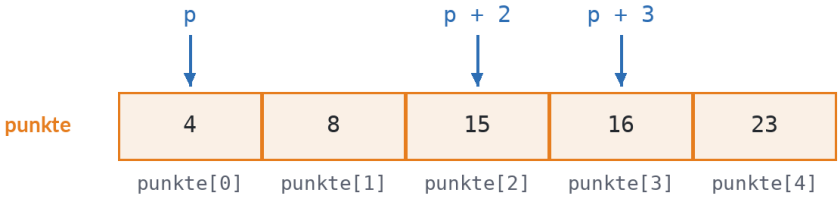
int main(void) {
    pbc_init();
    int punkte[5] = { 4, 8, 15, 16, 23 };
    int *p = punkte;              // dasselbe wie &punkte[0]

    printf("punkte[2] = %d\n", punkte[2]);
    printf("p[2]      = %d\n", p[2]);
```

```
19      printf("*(p + 2)  = %d\n", *(p + 2));
20
21      p = p + 3;                      // drei Elemente weiter
22      printf("*p        = %d\n", *p);
23
24      printf("Summe aller 5: %d\n", summe(punkte, 5));
25      printf("Summe ab [3]:  %d\n", summe(&punkte[3], 2));
26
27      while (true) {
28          sleep_ms(1000);
29      }
30  }
```

Serial Monitor

```
punkte[2] = 15
p[2]      = 15
*(p + 2)  = 15
*p        = 16
Summe aller 5: 66
Summe ab [3]:  39
```




Jedes Element ist ein int: 4 Byte. p + 1 zeigt 4 Byte weiter.

- Auf einen Zeiger darfst du eckige Klammern anwenden wie auf ein Array (Zeile 18). `p[2]` heißt: das Element, das zwei Plätze hinter `p` liegt.
- Dasselbe schreibt Zeile 19 als `*(p + 2)`. Zu einem Zeiger kann man ganze Zahlen addieren. `p + 2` zeigt **zwei Elemente** weiter, nicht zwei Bytes. Bei `int` sind das 8 Bytes, bei `uint16_t` wären es 4. Den Typ kennt der Compiler, er rechnet selbst richtig.
- Zeile 21 setzt den Zeiger um drei Elemente weiter. Danach zeigt `p` auf `punkte[3]`.

**Arrays an Funktionen übergeben.** Übergibst du ein Array an eine Funktion, bekommt sie nur einen Zeiger auf das erste Element (Zeile 4). Wie viele Elemente dahinter folgen, weiß sie nicht. Deshalb muss die Anzahl als zweiter Parameter mit. Dafür kann die Funktion auch mit einem Teil des Arrays arbeiten: `summe(&punkte[3], 2)` in Zeile 25 addiert nur die letzten beiden Elemente.

Hier lauert eine Falle. `sizeof` liefert in `main` die Größe des ganzen Arrays, in der Funktion aber nur die Größe eines Zeigers. Der Compiler warnt davor:

k19_sizeof_falle.c

```c
#include "pbc.h"

void zeige_groesse(int werte[5]) {
    printf("in der Funktion: %d Byte\n", (int)sizeof werte);
}

int main(void) {
    pbc_init();
    int werte[5] = { 1, 2, 3, 4, 5 };
    printf("in main:         %d Byte\n", (int)sizeof werte);
    zeige_groesse(werte);

    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
main.c: In function 'zeige_groesse':
main.c:4:54: warning: 'sizeof' on array function parameter 'werte' will return size
  of 'int *' [-Wsizeof-array-argument]
    4 |     printf("in der Funktion: %d Byte\n", (int)sizeof werte);
      |                                                      ^~~~~
main.c:3:24: note: declared here
    3 | void zeige_groesse(int werte[5]) {
      |                    ~~~~^~~~~~~
```

Auch wenn im Funktionskopf `int werte[5]` steht, ist `werte` dort ein Zeiger, ein `int *`. Auf dem PicoBoy meldet `main` deshalb 20 Bytes (5 × 4), die Funktion aber nur 4 Bytes, die Größe eines Zeigers.

**Texte sind auch Arrays.** Eine Zeichenkette ist ein `char`-Array, das mit `'\0'` endet (Kapitel 12.3). Funktionen für Texte bekommen also einen `char`-Zeiger und laufen Zeichen für Zeichen bis zur `'\0'`:

k19_texte.c

```c
#include "pbc.h"

// zaehlt Zeichen, bis die '\0' am Ende kommt
int laenge(const char *text) {
    int n = 0;
    while (*text != '\0') {
        n++;
        text++;                    // ein Zeichen weiter
    }
```

```
10      return n;
11  }
12
13  // ersetzt jedes Leerzeichen durch einen Unterstrich
14  void unterstriche(char *text) {
15      for (char *p = text; *p != '\0'; p++) {
16          if (*p == ' ') {
17              *p = '_';
18          }
19      }
20  }
21
22  int main(void) {
23      pbc_init();
24      char name[] = "PicoBoy Color Plus";
25      printf("%s hat %d Zeichen.\n", name, laenge(name));
26      unterstriche(name);
27      printf("%s\n", name);
28      printf("Ab Zeichen 8: %s\n", name + 8);
29
30      while (true) {
31          sleep_ms(1000);
32      }
33  }
```

Serial Monitor

```
PicoBoy Color Plus hat 18 Zeichen.
PicoBoy_Color_Plus
Ab Zeichen 8: Color_Plus
```

- `laenge` braucht keine Anzahl, denn das Ende erkennt sie an der `'\0'`. `text++` in Zeile 8 schiebt den Zeiger ein Zeichen weiter. Weil `text` eine Kopie der Adresse ist, bleibt `name` in `main` davon unberührt.
- `unterstriche` ändert den Text und hat deshalb `char *` ohne `const`.
- `name + 8` in Zeile 28 ist ein Zeiger auf das neunte Zeichen. `printf` gibt den Text von dort bis zur `'\0'` aus.

Genau so arbeitet auch `pbc_text` in der Bibliothek: Eine Schleife läuft mit dem Zeiger `s` durch den Text, bis `*s` die `'\0'` ist, und zeichnet jedes Zeichen 8 Pixel weiter rechts.

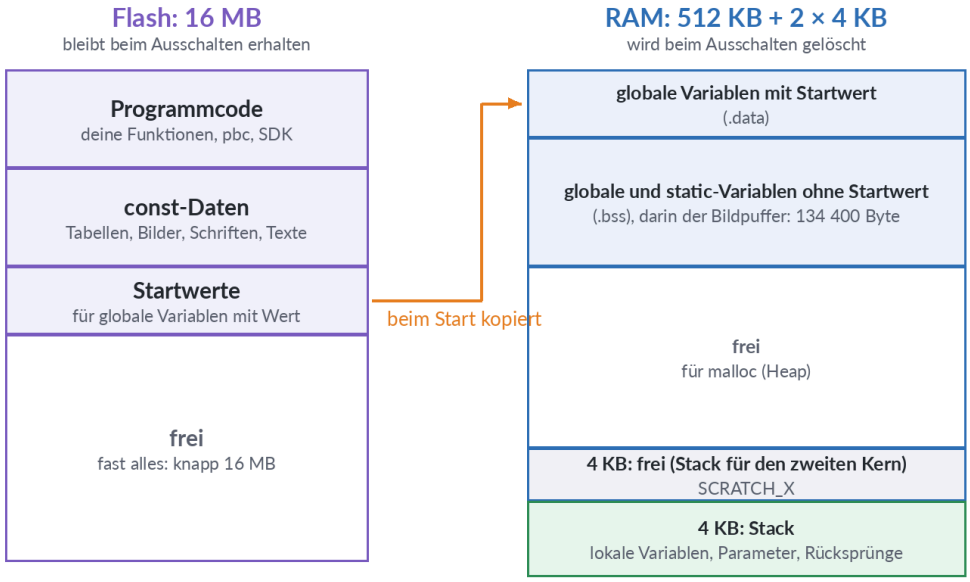
> `char name[] = "PicoBoy Color Plus";` legt ein **Array** an und kopiert den Text hinein. Das darfst du ändern. `const char *name = "PicoBoy";` ist dagegen nur ein Zeiger auf einen Text, der fest im Flash steht (Abschnitt 19.4). Diesen Text kann man nicht ändern.

## 19.4 512 KB RAM, 16 MB Flash: Stack, statische Daten, `const` im Flash

Der PicoBoy hat zwei Arten von Speicher:

- Der **Flash** (16 MB) behält seinen Inhalt ohne Strom. Dort liegt dein Programm. Schreiben ist langsam und geht nur in größeren Blöcken (Kapitel 25), Lesen geht schnell.
- Das **RAM** ist schnell zu lesen und zu schreiben, verliert aber beim Ausschalten alles. Der RP2350 hat 520 KB RAM: einen großen Bereich von 512 KB und zwei kleine von je 4 KB. (Bei Speicher rechnet man 1 KB = 1024 Bytes.)

Beide haben Adressen im selben Adressraum. Adressen ab `10000000` gehören zum Flash, Adressen ab `20000000` zum RAM. Die zwei kleinen Bereiche liegen bei `20080000` und `20081000`. Darum begann in 19.1 die Adresse von `FARBEN` mit `1000` und die von `rekord` mit `2000`.




Wohin eine Variable kommt, entscheidet die Art, wie du sie anlegst:

| So angelegt | liegt | Beispiel |
|---|---|---|
| Funktionen, also dein Programm | im Flash | `void bewege(…) { … }` |
| `const`-Variablen außerhalb von Funktionen, Texte in `"…"` | im Flash | `const uint16_t FARBEN[3] = …;` |
| globale Variablen und `static`-Variablen | im RAM, an fester Stelle | `int rekord = 500;` |
| lokale Variablen und Parameter | auf dem **Stack** | `int punkte = 10;` in `main` |

**Statische Daten.** Globale und `static`-Variablen gibt es vom Start bis zum Ausschalten. Der Linker gibt jeder von ihnen beim Bauen einen festen Platz im RAM. Hat eine Variable einen Startwert wie `rekord = 500`, steht dieser Wert zusätzlich im Flash, und das SDK kopiert ihn beim Start ins RAM. Variablen ohne Startwert setzt es beim Start auf 0. Auch der Bildpuffer der Bibliothek ist so eine Variable: ein Array mit 134 400 Bytes, rund ein Viertel des RAM.

**Der Stack.** Lokale Variablen gibt es nur, solange ihre Funktion läuft. Für sie gibt es den **Stack** (deutsch: Stapel). Ruft dein Programm eine Funktion auf, legt es ihre Parameter und lokalen Variablen oben auf den Stapel, dazu die Stelle, an der es nach dem Ende der Funktion weitergeht. Ist die Funktion fertig, wird dieser Platz wieder frei. Ruft eine Funktion eine weitere auf, wächst der Stapel weiter. Das geht schnell und kostet keinen Platz, solange die Funktion nicht läuft.

Der Stack des PicoBoy ist nur **4 KB** groß. Er liegt in einem der beiden kleinen RAM-Bereiche. Das legt die **Board-Datei** der Bibliothek fest, die das SDK über die Eigenschaften des PicoBoy informiert:

pbc_plus.h

```
 1 // Board-Datei für den PicoBoy Color Plus.
 2 // Alles wie beim Raspberry Pi Pico 2 (RP2350A), nur mit 16 MB Flash,
 3 // 4 KB Stack und ohne die Pico-LED an GP25.
 4
 5 pico_board_cmake_set(PICO_PLATFORM, rp2350)
 6 pico_board_cmake_set_default(PICO_FLASH_SIZE_BYTES, (16 * 1024 * 1024))
 7 // Wie beim Pico 2: UF2-Dateien bekommen den Zusatzblock für Chips der
 8 // Revision A2 (Erratum RP2350-E10), damit Drag & Drop dort sicher klappt.
 9 pico_board_cmake_set_default(PICO_RP2350_A2_SUPPORTED, 1)
10
11 #ifndef _BOARDS_PBC_PLUS_H
12 #define _BOARDS_PBC_PLUS_H
13
14 #define PICO_FLASH_SIZE_BYTES (16 * 1024 * 1024)
15 #include "boards/pico2.h"
16 #undef PICO_DEFAULT_LED_PIN
17
18 // Der Stack liegt in einem eigenen 4-KB-Speicherbereich; das SDK nutzt
19 // sonst nur die Hälfte davon. Läuft er trotzdem über, hält ein Wächter
20 // das Programm an, statt still andere Daten zu überschreiben.
21 #define PICO_STACK_SIZE 0x1000
22 #define PICO_USE_STACK_GUARDS 1
23
24 #endif
```

- Zeile 18 setzt die Größe des Stacks auf `0x1000`, hexadezimal für 4096 Bytes. Ohne diese Zeile würde das SDK nur 2 KB nutzen.
- Zeile 19 schaltet den **Wächter** ein. Was er tut, zeigt Abschnitt 19.5.

> Lokale Variablen kommen auf den Stack, und der ist nur 4 KB groß. Ein paar Zahlen und kleine Arrays sind kein Problem. **Große Arrays gehören nicht in eine Funktion**, sondern nach draußen (global), mit `static` oder, wenn sie sich nie ändern, mit `const` in den Flash.

**Nachmessen.** Der Linker kann am Ende des Bauens anzeigen, wie voll die Speicherbereiche sind. Füge dafür in der `CMakeLists.txt` deines Projekts ganz unten diese Zeile hinzu (Kapitel 20 erklärt die Datei genauer):

```
target_link_options(mein_programm PRIVATE -Wl,--print-memory-usage)
```

Nach dem nächsten **Compile** steht im Terminal unter `Linking CXX executable mein_programm.elf` eine Tabelle. Für das Programm `k19_tabelle_ohne.c`, das außer `pbc_init` und einem `printf` fast nichts tut, sieht sie so aus:

Terminal

```
Memory region         Used Size  Region Size  %age Used
           FLASH:       35292 B        16 MB      0.21%
             RAM:      144888 B       512 KB     27.64%
       SCRATCH_X:           0 B         4 KB      0.00%
       SCRATCH_Y:        4 KB         4 KB    100.00%
```

- **FLASH**: gut 35 000 Bytes Programm und Daten, das meiste davon SDK und Bibliothek. Von 16 MB ist das nur ein Fünftel Prozent.
- **RAM**: knapp 145 000 Bytes, davon 134 400 für den Bildpuffer. Den Rest brauchen die Bibliothek und das SDK, zum Beispiel für USB.
- **SCRATCH_X** und **SCRATCH_Y** sind die beiden kleinen 4-KB-Bereiche. SCRATCH_Y ist zu 100 % „belegt“: Das ist der Platz, der für den Stack reserviert ist. Wie viel davon gerade benutzt wird, steht hier nicht.

**Was `const` ausmacht.** Jetzt kommt eine Tabelle mit 10 000 Zahlen dazu, einmal ohne Startwerte, einmal mit und einmal mit `const`:

k19_tabelle_ram.c

```c
#include "pbc.h"

uint16_t tabelle[10000] = { 1, 2, 3 };

int main(void) {
    pbc_init();
    int i = pbc_random(0, 9999);
    printf("Eintrag %d: %d\n", i, tabelle[i]);

    while (true) {
        sleep_ms(1000);
    }
}
```

`k19_tabelle_bss.c` hat in Zeile 3 `uint16_t tabelle[10000];` ohne Startwerte, `k19_tabelle_flash.c` hat `const` davor. Sonst sind die drei Programme gleich. `pbc_random` wählt einen Eintrag aus, damit der Compiler die Tabelle nicht wegoptimieren kann. Das meldet der Linker:

ohne Startwerte

```
Memory region         Used Size  Region Size  %age Used
           FLASH:       35300 B        16 MB      0.21%
             RAM:      164888 B       512 KB     31.45%
       SCRATCH_X:           0 B         4 KB      0.00%
       SCRATCH_Y:        4 KB           4 KB    100.00%
```

mit Startwerten

```
Memory region         Used Size  Region Size  %age Used
           FLASH:       55300 B        16 MB      0.33%
             RAM:      164888 B       512 KB     31.45%
       SCRATCH_X:           0 B         4 KB      0.00%
       SCRATCH_Y:        4 KB           4 KB    100.00%
```

mit const

```
Memory region         Used Size  Region Size  %age Used
           FLASH:       55308 B        16 MB      0.33%
             RAM:      144888 B       512 KB     27.64%
       SCRATCH_X:           0 B         4 KB      0.00%
       SCRATCH_Y:        4 KB           4 KB    100.00%
```

| Tabelle | Flash | RAM |
|---|---|---|
| ohne Startwerte | + 8 Bytes | + 20 000 Bytes |
| mit Startwerten | + 20 008 Bytes | + 20 000 Bytes |
| mit `const` | + 20 016 Bytes | **+ 0 Bytes** |

10 000 `uint16_t` sind 20 000 Bytes. Ohne Startwerte braucht die Tabelle nur RAM. Mit Startwerten braucht sie den Platz **doppelt**: im Flash für die Startwerte und im RAM für die Variable. Mit `const` liegt sie nur im Flash, und davon gibt es mehr als dreißigmal so viel wie RAM. Die wenigen Bytes darüber hinaus sind der Maschinencode, der auf die Tabelle zugreift.

> Alles, was sich nie ändert, bekommt `const`: Tabellen, Texte, Level-Karten, Bilder. Das Umwandlungswerkzeug aus Kapitel 15 schreibt deshalb vor jedes Bild `const`. Auch die Schriften der Bibliothek liegen so im Flash.

## 19.5 Wenn der Speicher nicht reicht

Es gibt zwei Arten, wie der Speicher knapp wird. Die erste merkst du schon beim Bauen.

**Zu viele statische Daten.** Dieses Programm möchte drei weitere Bildpuffer anlegen:

k19_fehler_zu_gross.c

```c
#include "pbc.h"

uint16_t bilder[3][PBC_WIDTH * PBC_HEIGHT];   // drei weitere Bildpuffer

int main(void) {
    pbc_init();
    for (int i = 0; i < PBC_WIDTH * PBC_HEIGHT; i++) {
        bilder[0][i] = PBC_RED;
        bilder[1][i] = PBC_GREEN;
        bilder[2][i] = PBC_BLUE;
    }
    printf("%d\n", bilder[pbc_random(0, 2)][pbc_random(0, 1000)]);

    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
Linking CXX executable k19_fehler_zu_gross.elf
ld: k19_fehler_zu_gross.elf section `.bss' will not fit in region `RAM'
ld: section .stack_dummy VMA [20081000,20081fff] overlaps section .bss VMA
  [200012d8,200854f7]
ld: region `RAM' overflowed by 23800 bytes
collect2: error: ld returned 1 exit status
```

Diesmal meldet sich nicht der Compiler, sondern der **Linker** `ld`. Vorne steht deshalb nicht `main.c`, sondern der lange Pfad zum Linker. Er versucht, alle Variablen im RAM unterzubringen, und es fehlen 23 800 Bytes: `region `RAM' overflowed`, der Bereich RAM ist übergelaufen. Die zweite Zeile sagt dasselbe mit Adressen: Die Variablen würden bis in den Bereich des Stacks (`.stack_dummy`) reichen. Abhilfe: weniger oder kleinere Arrays, oder `const`, wenn sich die Daten nicht ändern.

**Ein übervoller Stack.** Den zweiten Fall bemerkt der Compiler nicht:

k19_stapel.c

```c
#include "pbc.h"

int summe_bis(int n) {
    int zahlen[2000];              // 2000 × 4 Byte = 8000 Byte auf dem Stack!
    for (int i = 0; i < n; i++) {
        zahlen[i] = i;
    }
    int s = 0;
```

```
 9      for (int i = 0; i < n; i++) {
10          s += zahlen[i];
11      }
12      return s;
13  }
14
15  int main(void) {
16      pbc_init();
17      pbc_led_green(true);
18      printf("Summe: %d\n", summe_bis(2000));
19      pbc_led_green(false);
20      pbc_led_red(true);              // kommt das Programm hier an?
21
22      while (true) {
23          sleep_ms(1000);
24      }
25  }
```

Das Array `zahlen` in Zeile 4 ist eine lokale Variable mit 8000 Bytes. Das Programm baut ohne Fehler und ohne Warnung. Denn wie voll der Stack zur Laufzeit wird, hängt davon ab, welche Funktionen sich gegenseitig aufrufen, und das kann der Compiler nicht überblicken.

Auf dem PicoBoy passiert Folgendes: Die grüne LED geht an, dann ruft `main` die Funktion `summe_bis` auf. Sie will 8000 Bytes auf dem Stack belegen, aber der hat nur 4096. Jetzt greift der **Wächter** aus der Board-Datei: Der Prozessor kennt die untere Grenze des Stacks und löst einen Fehler aus, sobald der Stack darüber hinauswachsen will. Das Programm bleibt sofort stehen. Die grüne LED leuchtet weiter, die rote geht nie an, und im seriellen Monitor erscheint keine Summe. Der PicoBoy reagiert auf nichts mehr. Zum nächsten Aufspielen brauchst du eventuell den Weg über BOOT und RES (Kapitel 2.4).

Ohne Wächter wäre es schlimmer: Der Stack würde still in den Speicher darunter hineinwachsen und dort andere Daten überschreiben. Das Programm liefe vielleicht weiter und täte irgendwann etwas Seltsames, weit weg von der eigentlichen Ursache.

Die Lösung ist ein Wort: Mit `static` liegt das Array nicht mehr auf dem Stack, sondern an einer festen Stelle im großen RAM (Kapitel 11.4):

k19_stapel_static.c

```
 3  int summe_bis(int n) {
 4      static int zahlen[2000];        // static: liegt im RAM, nicht auf dem Stack
 5      for (int i = 0; i < n; i++) {
 6          zahlen[i] = i;
 7      }
 8      int s = 0;
 9      for (int i = 0; i < n; i++) {
10          s += zahlen[i];
11      }
12      return s;
13  }
```

Serial Monitor

```
Summe: 1999000
```

**Wenn der Speicher knapp wird:**

- Große Arrays nie als lokale Variablen anlegen: `static`, global oder `const`.
- Was sich nicht ändert, mit `const` in den Flash legen.
- Kleine Typen wählen: Für Werte von 0 bis 255 reicht `uint8_t` (1 Byte) statt `int` (4 Bytes).
- Ausrechnen statt speichern: Oft lässt sich ein Wert schnell berechnen, statt ihn in einer Tabelle abzulegen.
- Mit `-Wl,--print-memory-usage` nachsehen, wie voll RAM und Flash sind.

Das freie RAM hinter den statischen Daten kann ein Programm mit der Funktion `malloc` zur Laufzeit anfordern. Dieser Bereich heißt **Heap**. Die Programme in diesem Buch brauchen ihn nicht: Auf einem Gerät ohne Betriebssystem ist es meist sicherer, allen Platz vorher festzulegen. Dann meldet der Linker beim Bauen, wenn es nicht reicht.

## Ausprobieren: Figuren auswählen

Drei Quadrate stehen auf dem Bildschirm. Mit dem Joystick bewegst du eines davon, mit A wählst du das nächste aus. Ein Zeiger merkt sich, welches gerade dran ist:

k19_auswahl.c

```c
#include "pbc.h"

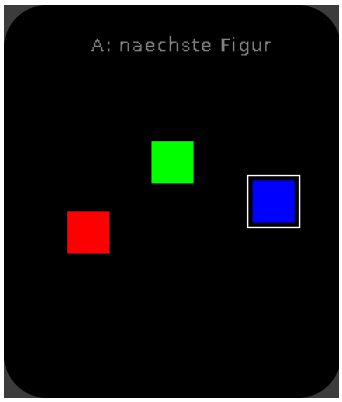
typedef struct {
    int x, y;
    uint16_t farbe;
} figur;

figur figuren[3] = {
    { 60, 140, PBC_RED },
    { 120, 140, PBC_GREEN },
    { 180, 140, PBC_BLUE },
};

void steuern(figur *f) {
    if (pbc_pressed_left() && f->x > 40) {
        f->x -= 2;
    }
    if (pbc_pressed_right() && f->x < 200) {
        f->x += 2;
    }
```

```
21      if (pbc_pressed_up() && f->y > 60) {
22          f->y -= 2;
23      }
24      if (pbc_pressed_down() && f->y < 230) {
25          f->y += 2;
26      }
27  }
28
29  int main(void) {
30      pbc_init();
31      int nummer = 0;
32      figur *aktiv = &figuren[nummer];      // die Figur, die sich gerade bewegt
33      while (true) {
34          if (pbc_was_pressed_a()) {
35              nummer = (nummer + 1) % 3;
36              aktiv = &figuren[nummer];
37          }
38          steuern(aktiv);
39
40          pbc_clear(PBC_BLACK);
41          pbc_text12("A: naechste Figur", 62, 20, PBC_GRAY);
42          for (int i = 0; i < 3; i++) {
43              pbc_fill_rect(figuren[i].x - 15, figuren[i].y - 15, 30, 30,
                  figuren[i].farbe);
44          }
45          pbc_rect(aktiv->x - 19, aktiv->y - 19, 38, 38, PBC_WHITE);
46          pbc_show();
47          sleep_ms(20);
48      }
49  }
```

- Zeile 32: `aktiv` zeigt am Anfang auf die erste Figur.
- Drückst du A, zeigt `aktiv` auf die nächste (Zeile 36). Nach der letzten kommt wegen `% 3` wieder die erste.
- `steuern(aktiv)` in Zeile 38 bewegt immer die Figur, auf die `aktiv` zeigt. Die Funktion muss nicht wissen, welche das ist. Ohne Zeiger bräuchtest du hier drei Fälle mit `if`.
- Zeile 45 zeichnet den weißen Rahmen um die aktive Figur, wieder über den Zeiger.




Im Bild wurde die rote Figur nach unten bewegt, die grüne nach oben und die blaue nach rechts.

**Zum Verändern:** Gib jeder Figur eine eigene Größe als weiteres Element der Struktur. Mit B soll die aktive Figur ihre Farbe wechseln.

## Übungsaufgaben

**Aufgabe 1 ⭐ Wer zeigt wohin?** Was gibt dieses Programm aus? Überlege zuerst mit Papier und Bleistift und prüfe dann am PicoBoy.

k19_aufgabe1.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int a = 5;
    int b = 7;
    int *p = &a;
    *p = *p + 10;
    p = &b;
    *p = *p * 2;
    printf("a = %d, b = %d, *p = %d\n", a, b, *p);

    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 2 ⭐ Verdoppeln** Schreibe eine Funktion `void verdopple(int *zahl)`, die die Variable verdoppelt, auf die `zahl` zeigt. Rufe sie in einer Schleife fünfmal für eine Variable mit dem Startwert 3 auf und gib jedes Mal den Wert aus.

**Aufgabe 3 ⭐⭐ Zwei Ergebnisse** Schreibe eine Funktion `void min_max(const int *werte, int anzahl, int *kleinstes, int *groesstes)`, die den kleinsten und den größten Wert eines Arrays findet. Teste sie mit den Temperaturen einer Woche: 12, 15, 9, 17, 21, 14, 11.

**Aufgabe 4 ⭐⭐ Finde den Fehler** Dieses Programm baut nicht. Lies die Fehlermeldung und verbessere es.

k19_aufgabe4.c

```c
#include "pbc.h"

// haelt *wert zwischen min und max
void begrenze(int *wert, int min, int max) {
    if (*wert < min) {
        *wert = min;
    }
```

```
    if (*wert > max) {
        *wert = max;
    }
}

int main(void) {
    pbc_init();
    int x = 300;
    begrenze(x, 0, 239);
    printf("x = %d\n", x);

    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 5 ⭐⭐⭐ Platz für Bilder** Ein Spiel soll 40 Hintergrundbilder in voller Bildschirmgröße mitbringen. Lass den PicoBoy ausrechnen und ausgeben, wie viele Bytes ein Bild und alle 40 Bilder brauchen und wie viele ganze Bilder ins RAM (512 KB) und in den Flash (16 MB) passen würden. Rechne mit `PBC_WIDTH`, `PBC_HEIGHT` und `sizeof(uint16_t)`. Wie müssen die Bilder im Programm angelegt werden?

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- Jede Variable hat eine **Adresse**: `&x`. Ein **Zeiger** merkt sich eine Adresse: `int *z = &x;`. Mit `*z` liest und schreibst du die Variable, auf die er zeigt. `NULL` zeigt auf nichts.
- Soll eine Funktion Variablen des Aufrufers ändern, bekommt sie Zeiger: `tausche(&a, &b)`. Bei Strukturen: `zeiger->element`.
- `const` beim Zeiger-Parameter heißt: Die Funktion liest nur.
- Ein Array-Name ist die Adresse des ersten Elements. `p[i]` ist `*(p + i)`. Eine Funktion bekommt nur den Zeiger und braucht die Anzahl extra; `sizeof` liefert dort nur die Größe des Zeigers (4 Bytes).
- **Flash** (16 MB): Programm und `const`-Daten. **RAM** (512 KB + 2 × 4 KB): globale und `static`-Variablen, Bildpuffer, Stack.
- Lokale Variablen liegen auf dem **Stack**, und der hat nur **4 KB**. Große Arrays: `static`, global oder `const`. Läuft der Stack über, hält der Wächter das Programm an.
- `-Wl,--print-memory-usage` in der `CMakeLists.txt` zeigt, wie voll der Speicher ist.

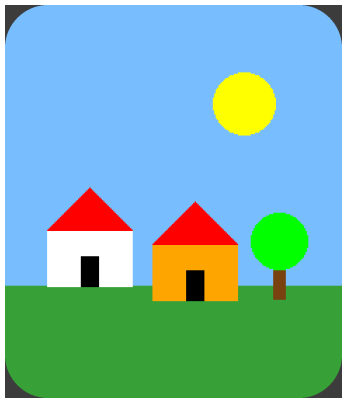
KAPITEL 20

# Mehrere Dateien und eigene Bibliotheken

**Worum geht's?** Bisher stand jedes Programm in einer einzigen Datei `main.c`. Größere Programme verteilt man auf mehrere Dateien: Zeichnen, Spielregeln und Musik bekommen jeweils ihre eigene. Du lernst, was in eine `.h`-Datei gehört und was in eine `.c`-Datei, wie du der Bauanleitung `CMakeLists.txt` neue Dateien bekannt machst und wie du eine eigene kleine Bibliothek baust. Zum Schluss schaust du in die Buchbibliothek `pbc` hinein. Sie ist genau so aufgebaut.

## 20.1 `.h` und `.c`: Deklaration und Definition

Dieses Programm zeichnet ein kleines Bild aus Figuren. Die Figuren sind eigene Funktionen:



k20_figuren_eine_datei.c

```c
#include "pbc.h"

int anzahl_figuren = 0;

static void zeichne_dach(int x, int y, int breite) {
    for (int i = 0; i <= breite / 2; i++) {
        pbc_hline(x + breite / 2 - i, y + i, 2 * i + 1, PBC_RED);
    }
}

void zeichne_haus(int x, int y, uint16_t farbe) {
    zeichne_dach(x, y, 60);
    pbc_fill_rect(x, y + 31, 61, 40, farbe);
    pbc_fill_rect(x + 24, y + 49, 13, 22, PBC_BLACK);   // Tuer
    anzahl_figuren++;
}

void zeichne_baum(int x, int y) {
    pbc_fill_rect(x - 4, y, 9, 30, pbc_rgb(120, 70, 20));
    pbc_fill_circle(x, y - 12, 20, PBC_GREEN);
```

```
    anzahl_figuren++;
}

void zeichne_sonne(int x, int y, int r) {
    pbc_fill_circle(x, y, r, PBC_YELLOW);
    anzahl_figuren++;
}

int main(void) {
    pbc_init();
    pbc_clear(pbc_rgb(120, 190, 255));
    pbc_fill_rect(0, 200, PBC_WIDTH, 80, pbc_rgb(60, 160, 60));
    zeichne_sonne(170, 70, 22);
    zeichne_haus(30, 130, PBC_WHITE);
    zeichne_haus(105, 140, PBC_ORANGE);
    zeichne_baum(195, 180);
    pbc_show();
    printf("Gezeichnete Figuren: %d\n", anzahl_figuren);

    while (true) {
        sleep_ms(1000);
    }
}
```

Die Funktionen `zeichne_haus`, `zeichne_baum` und `zeichne_sonne` wären auch in anderen Programmen nützlich. Du könntest sie jedes Mal hineinkopieren. Besser ist es, sie in eigene Dateien auszulagern, die mehrere Programme benutzen können. Dafür braucht man zwei Dateien: eine **Header-Datei** `figuren.h` und eine **Quelldatei** `figuren.c`.

**Die Header-Datei** (`.h` von *header*, Kopf) sagt, **was** es gibt. Sie enthält nur die Funktionsköpfe mit Semikolon, die **Prototypen** aus Kapitel 11.3. Einen Prototyp nennt man auch **Deklaration**: Er kündigt dem Compiler an, dass es die Funktion gibt, wie sie heißt und welche Parameter sie hat.

figuren.h

```
// figuren.h – fertige Figuren für Bilder auf dem PicoBoy
#ifndef FIGUREN_H
#define FIGUREN_H

#include <stdint.h>

// zaehlt mit, wie viele Figuren schon gezeichnet wurden
extern int anzahl_figuren;

// Haus mit rotem Dach; (x, y) ist die Spitze links oben, 61 x 71 Pixel
void zeichne_haus(int x, int y, uint16_t farbe);

// Baum; (x, y) ist die Mitte am oberen Ende des Stamms
void zeichne_baum(int x, int y);
```

```
16 // Sonne mit Mittelpunkt (x, y) und Radius r
17 void zeichne_sonne(int x, int y, int r);
18
19 #endif
```

- Die Kommentare beschreiben, was jede Funktion tut. Wer die Figuren benutzen will, muss nur diese Datei lesen.
- Zeile 5 holt `stdint.h`, denn `uint16_t` kommt in der Datei vor.
- Die Zeilen 2, 3 und 19 heißen **Include-Wächter**. Sie sorgen dafür, dass der Inhalt nur einmal zählt, auch wenn die Datei in einer `.c`-Datei versehentlich zweimal eingebunden wird. `#ifndef FIGUREN_H` heißt: „Wenn `FIGUREN_H` noch nicht definiert ist“. Beim ersten Mal ist es das nicht, also definiert Zeile 3 den Namen, und der Inhalt bis `#endif` zählt. Beim zweiten Mal ist `FIGUREN_H` schon bekannt, und der Compiler überspringt alles bis `#endif`. Jede Header-Datei hat einen solchen Wächter, mit einem eigenen Namen in Großbuchstaben.
- Zeile 8 mit `extern` erklärt Abschnitt „Gemeinsame Variablen“ unten.

**Die Quelldatei** sagt, **wie** es gemacht wird. Sie enthält die vollständigen Funktionen mit ihrem Rumpf in `{ }`. Das heißt **Definition**:

figuren.c

```
 1 // figuren.c — so werden die Figuren gezeichnet
 2 #include "pbc.h"
 3 #include "figuren.h"
 4
 5 int anzahl_figuren = 0;
 6
 7 // static: nur in dieser Datei bekannt
 8 static void zeichne_dach(int x, int y, int breite) {
 9     for (int i = 0; i <= breite / 2; i++) {
10         pbc_hline(x + breite / 2 - i, y + i, 2 * i + 1, PBC_RED);
11     }
12 }
13
14 void zeichne_haus(int x, int y, uint16_t farbe) {
15     zeichne_dach(x, y, 60);
16     pbc_fill_rect(x, y + 31, 61, 40, farbe);
17     pbc_fill_rect(x + 24, y + 49, 13, 22, PBC_BLACK);   // Tuer
18     anzahl_figuren++;
19 }
20
21 void zeichne_baum(int x, int y) {
22     pbc_fill_rect(x - 4, y, 9, 30, pbc_rgb(120, 70, 20));
23     pbc_fill_circle(x, y - 12, 20, PBC_GREEN);
24     anzahl_figuren++;
25 }
26
27 void zeichne_sonne(int x, int y, int r) {
28     pbc_fill_circle(x, y, r, PBC_YELLOW);
29     anzahl_figuren++;
30 }
```

- Zeile 3 bindet die eigene Header-Datei ein. So prüft der Compiler, ob die Definitionen zu den Deklarationen passen. Weicht ein Parameter ab, meldet er einen Fehler.
- `zeichne_dach` in Zeile 8 ist **`static`**. Bei einer Funktion heißt das: Sie ist nur in dieser Datei bekannt. Andere Dateien können sie nicht aufrufen. Sie ist ein Hilfsteil der Figuren und steht deshalb auch nicht in `figuren.h`.

**Das Hauptprogramm** wird jetzt kurz:

main.c

```
 1 #include "pbc.h"
 2 #include "figuren.h"
 3
 4 int main(void) {
 5     pbc_init();
 6     pbc_clear(pbc_rgb(120, 190, 255));
 7     pbc_fill_rect(0, 200, PBC_WIDTH, 80, pbc_rgb(60, 160, 60));
 8     zeichne_sonne(170, 70, 22);
 9     zeichne_haus(30, 130, PBC_WHITE);
10     zeichne_haus(105, 140, PBC_ORANGE);
11     zeichne_baum(195, 180);
12     pbc_show();
13     printf("Gezeichnete Figuren: %d\n", anzahl_figuren);
14
15     while (true) {
16         sleep_ms(1000);
17     }
18 }
```

`#include "figuren.h"` in Zeile 2 genügt, damit der Compiler die Figuren in `main.c` kennt. Das Bild ist dasselbe wie oben.

> - **`.h`**: Deklarationen (Prototypen, `extern`-Variablen, `#define`, `typedef`), mit Include-Wächter. Sie sagt, **was** es gibt.
> - **`.c`**: Definitionen, also die Funktionen mit Rumpf und die Variablen. Sie sagt, **wie** es geht.
> - Jede `.c`-Datei bindet die `.h`-Dateien ein, deren Funktionen sie benutzt, und ihre eigene.

**Was beim Bauen passiert.** Der Compiler übersetzt jede `.c`-Datei **einzeln**, ohne die anderen zu sehen. Beim Übersetzen von `main.c` weiß er über die Figuren nur, was in `figuren.h` steht. Das genügt ihm: Er weiß, wie man `zeichne_haus` aufruft, und lässt im Maschinencode eine Lücke für die Adresse der Funktion. Erst der **Linker** (Kapitel 2.3) setzt alle übersetzten Teile zusammen und füllt die Lücken. `#include` selbst ist übrigens ganz einfach: Der Compiler setzt an dieser Stelle den Text der Datei ein, als hättest du ihn abgetippt.

Vergisst du `#include "figuren.h"` in `main.c`, kennt der Compiler die Figuren dort nicht:

Terminal

```
main.c: In function 'main':
main.c:7:5: error: implicit declaration of function 'zeichne_sonne' [-Wimplicit-
  function-declaration]
    7 |     zeichne_sonne(170, 70, 22);
      |     ^~~~~~~~~~~~~
main.c:8:5: error: implicit declaration of function 'zeichne_haus' [-Wimplicit-
  function-declaration]
    8 |     zeichne_haus(30, 130, PBC_WHITE);
      |     ^~~~~~~~~~~~
main.c:10:5: error: implicit declaration of function 'zeichne_baum' [-Wimplicit-
  function-declaration]
   10 |     zeichne_baum(195, 180);
      |     ^~~~~~~~~~~~
main.c:12:41: error: 'anzahl_figuren' undeclared (first use in this function)
   12 |     printf("Gezeichnete Figuren: %d\n", anzahl_figuren);
      |                                         ^~~~~~~~~~~~~~
main.c:12:41: note: each undeclared identifier is reported only once for each
  function it appears in
```

**Gemeinsame Variablen.** Die Variable `anzahl_figuren` zählt mit, wie viele Figuren gezeichnet wurden. `figuren.c` erhöht sie, `main.c` gibt sie aus. Eine Variable darf es aber im ganzen Programm nur **einmal** geben. Deshalb steht die Definition `int anzahl_figuren = 0;` in `figuren.c`, und `figuren.h` sagt mit `extern` nur: „Diese Variable gibt es, sie ist woanders definiert." Schreibt man die Definition in die Header-Datei, steht sie in jeder `.c`-Datei, die den Header einbindet. Dann gibt es sie zweimal, und der Linker beschwert sich:

Terminal

```
ld: figuren.c.o:figuren.h:8: multiple definition of `anzahl_figuren';
  main.c.o:figuren.h:8: first defined here
collect2: error: ld returned 1 exit status
```

*Multiple definition* heißt: Die Variable ist mehrfach definiert, einmal beim Übersetzen von `main.c` und einmal bei `figuren.c`. Gemeinsame globale Variablen sollte es ohnehin nur wenige geben. Oft ist eine Funktion besser, die den Wert liefert.

## 20.2 Die `CMakeLists.txt` lesen und erweitern

Woher weiß das Bauwerkzeug, dass `figuren.c` zum Programm gehört? Das steht in der Bauanleitung `CMakeLists.txt`. Hier ist der wichtige Teil aus der Buchvorlage, wie du sie seit Kapitel 2 benutzt:

CMakeLists.txt

```
18 # Der PicoBoy Color Plus: Board-Datei in pbc/boards/pbc_plus.h
19 set(PICO_BOARD_HEADER_DIRS ${CMAKE_CURRENT_LIST_DIR}/pbc/boards)
20 set(PICO_BOARD pbc_plus CACHE STRING "Board type")
```

```
22 include(pico_sdk_import.cmake)
23
24 project(mein_programm C CXX ASM)
25 set(CMAKE_C_STANDARD 11)
26 pico_sdk_init()
27
28 add_subdirectory(pbc)
29
30 add_executable(mein_programm main.c)
31 target_link_libraries(mein_programm pbc)
32
33 # Der Compiler soll auf verdächtige Stellen hinweisen (Warnungen).
34 target_compile_options(mein_programm PRIVATE -Wall -Wextra)
35
36 # Beim Start bis zu 2 Sekunden warten, ob sich ein serieller Monitor
37 # verbindet – sonst gingen die ersten printf-Ausgaben verloren.
38 target_compile_definitions(mein_programm PRIVATE
     PICO_STDIO_USB_CONNECT_WAIT_TIMEOUT_MS=2000)
39
40 # printf und Eingaben über USB. Die serielle Schnittstelle (UART) bleibt aus:
41 # Sie läge auf GP0/GP1 – dort hängen Joystick-Mitte und -Rechts.
42 pico_enable_stdio_usb(mein_programm 1)
43 pico_enable_stdio_uart(mein_programm 0)
44
45 # Erzeugt neben der .elf auch die .uf2-Datei zum Aufspielen.
46 pico_add_extra_outputs(mein_programm)
```

Eine `CMakeLists.txt` besteht aus Befehlen der Form `befehl(argumente)`. Zeilen mit `#` sind Kommentare. Die Zeilen 1 bis 16 (hier nicht gezeigt) richten die Erweiterung „Raspberry Pi Pico“ ein; dort änderst du nichts.

| Zeile | Befehl | bedeutet |
|---|---|---|
| 19–20 | `set(PICO_BOARD …)` | Das Gerät ist der PicoBoy (`pbc_plus`); die Board-Datei liegt in `pbc/boards` (Kapitel 19.4). |
| 22 | `include(pico_sdk_import.cmake)` | Das Pico SDK finden. |
| 24 | `project(mein_programm …)` | Der Name des Projekts; die Sprachen C, C++ und Assembler. |
| 25 | `set(CMAKE_C_STANDARD 11)` | Die Fassung der Sprache C von 2011. |
| 26 | `pico_sdk_init()` | Das SDK einrichten. |
| 28 | `add_subdirectory(pbc)` | Im Ordner `pbc` steht eine weitere `CMakeLists.txt`: die der Bibliothek. |
| 30 | `add_executable(mein_programm main.c)` | **Das Programm `mein_programm` besteht aus der Datei `main.c`.** |
| 31 | `target_link_libraries(mein_programm pbc)` | Es benutzt die Bibliothek `pbc`. |

| Zeile | Befehl | bedeutet |
|---|---|---|
| 34 | `target_compile_options(… -Wall -Wextra)` | Der Compiler soll warnen (Kapitel 3.5). |
| 38 | `target_compile_definitions(…)` | Bis zu 2 s auf den seriellen Monitor warten (Kapitel 4.1). |
| 42–43 | `pico_enable_stdio_…` | `printf` über USB, nicht über die serielle Schnittstelle an GP0/GP1. |
| 46 | `pico_add_extra_outputs(…)` | Außer der `.elf`-Datei auch die `.uf2`-Datei erzeugen. |

Ein **Target** (deutsch: Ziel) ist etwas, das gebaut wird: ein Programm oder eine Bibliothek. Die Befehle, die mit `target_` beginnen, beziehen sich auf ein Target, hier auf `mein_programm`.

**Eine Datei hinzufügen.** In Zeile 30 stehen alle `.c`-Dateien des Programms. Für die Figuren kommt `figuren.c` dazu, durch ein Leerzeichen getrennt:

```
add_executable(mein_programm main.c figuren.c)
```

`.h`-Dateien gehören nicht in die Liste; sie werden ja mit `#include` in die `.c`-Dateien eingesetzt. Beim nächsten **Compile** merkt das Bauwerkzeug, dass sich die `CMakeLists.txt` geändert hat, und liest sie neu.

> Die fertigen Projekte zu diesem Kapitel liegen im Buchordner unter `projekte/`, zum Beispiel `projekte/k20_figuren/`. Damit nicht jedes eine eigene Kopie der Bibliothek braucht, zeigt ihre `CMakeLists.txt` auf den Ordner `pbc` der Vorlage. In deiner eigenen Kopie der Vorlage liegt `pbc` im Projektordner, und es bleibt bei `add_subdirectory(pbc)`.

Vergisst du, `figuren.c` einzutragen, übersetzt der Compiler `main.c` problemlos: Die Prototypen stehen ja in `figuren.h`. Erst der Linker merkt, dass die Funktionen nirgends definiert sind:

Terminal

```
ld: main.c.o: in function `main':
main.c:8:(.text.startup.main+0x38): undefined reference to `zeichne_sonne'
ld: (zeichne_sonne): Unknown destination type (ARM/Thumb) in main.c.o
main.c:8:(.text.startup.main+0x38): dangerous relocation: unsupported relocation
ld: main.c:9:(.text.startup.main+0x44): undefined reference to `zeichne_haus'
ld: (zeichne_haus): Unknown destination type (ARM/Thumb) in main.c.o
main.c:9:(.text.startup.main+0x44): dangerous relocation: unsupported relocation
ld: main.c:10:(.text.startup.main+0x50): undefined reference to `zeichne_haus'
ld: (zeichne_haus): Unknown destination type (ARM/Thumb) in main.c.o
main.c:10:(.text.startup.main+0x50): dangerous relocation: unsupported relocation
ld: main.c:11:(.text.startup.main+0x58): undefined reference to `zeichne_baum'
```

```
ld: (zeichne_baum): Unknown destination type (ARM/Thumb) in main.c.o
main.c:11:(.text.startup.main+0x58): dangerous relocation: unsupported relocation
ld: main.c:15:(.text.startup.main+0x74): undefined reference to `anzahl_figuren'
collect2: error: ld returned 1 exit status
```

Die Meldung stammt vom Linker `ld`. `undefined reference to `zeichne_sonne'` heißt: `main.c` ruft `zeichne_sonne` auf, aber keine der gebauten Dateien enthält diese Funktion. Die Zeilen mit *dangerous relocation* sind Folgefehler derselben Ursache. Die Zeilennummern `main.c:8` und so weiter zeigen ungefähr, wo die fehlenden Funktionen aufgerufen werden.

> `undefined reference to …` heißt fast immer: Eine `.c`-Datei fehlt in `add_executable`, oder der Name in der Definition ist anders geschrieben als im Aufruf.

## 20.3 Ein Blick in die Buchbibliothek `pbc`

Jetzt kannst du lesen, wie die Bibliothek gebaut ist, die du seit Kapitel 3 benutzt. Sie liegt im Ordner `pbc` deines Projekts:

| Datei | Inhalt |
|---|---|
| `CMakeLists.txt` | macht aus den `.c`-Dateien die Bibliothek `pbc` |
| `pbc.h` | alle Befehle für deine Programme (Anhang A) |
| `pbc_intern.h` | was die beiden `.c`-Dateien untereinander teilen |
| `pbc_zeichnen.c` | Zeichnen, Schrift, Zufall, Eingaben: alles ohne Hardware |
| `pbc_geraet.c` | Display, Tasten, LEDs, Ton, Sensor, Batteriespannung: die Hardware |
| `schrift8.h` … `schrift24.h` | die vier Schriften als `const`-Arrays |
| `boards/pbc_plus.h` | die Board-Datei (Kapitel 19.4) |

**Die Bauanleitung der Bibliothek:**

CMakeLists.txt

```
# Die Buchbibliothek pbc als eigene Bibliothek, die main.c dazubindet.
add_library(pbc STATIC
    pbc_zeichnen.c
    pbc_geraet.c
)
target_include_directories(pbc PUBLIC ${CMAKE_CURRENT_LIST_DIR})
target_link_libraries(pbc PUBLIC
    pico_stdlib
    pico_rand
    hardware_adc
    hardware_dma
    hardware_i2c
    hardware_pio
```

```
    hardware_pwm
    hardware_spi
)
```

- `add_library(pbc STATIC …)` in Zeile 2 ist das Gegenstück zu `add_executable`: Aus den beiden `.c`-Dateien wird eine **Bibliothek**, kein eigenes Programm. Eine Bibliothek hat kein `main`. Sie wird erst durch `target_link_libraries(mein_programm pbc)` Teil deines Programms.
- Zeile 6 sorgt dafür, dass jedes Programm, das `pbc` benutzt, die Datei `pbc.h` findet. Darum genügt dir `#include "pbc.h"`, obwohl die Datei in einem anderen Ordner liegt. `PUBLIC` heißt: Das gilt auch für alle, die die Bibliothek benutzen, nicht nur für die Bibliothek selbst.
- Die Zeilen 7 bis 16 nennen die Teile des Pico SDK, die die Bibliothek braucht. `pico_stdlib` enthält zum Beispiel `sleep_ms` und `printf`, `hardware_pwm` die Funktionen für PWM (Kapitel 21), `hardware_i2c` die für I2C (Kapitel 22).

**Die Header-Datei `pbc.h` beginnt so:**

pbc.h

```
#ifndef PBC_H
#define PBC_H

// Damit reicht in den Programmen des Buchs ein einziges #include "pbc.h".
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "pico/stdlib.h"

#ifdef __cplusplus
extern "C" {
#endif
```

Zeile 14 und 15 sind der Include-Wächter, das zugehörige `#endif` steht in der letzten Zeile der Datei. Die Zeilen 18 bis 23 binden die Standard-Header ein, die die Programme im Buch brauchen: `stdbool.h` für `bool`, `stdint.h` für `uint8_t` und Co., `stdio.h` für `printf`, `stdlib.h` und `string.h` für weitere Standardfunktionen und `pico/stdlib.h` für `sleep_ms`. Deshalb hast du bisher nie etwas anderes einbinden müssen als `pbc.h`. Die Zeilen 25 bis 27 kannst du übergehen: Sie erlauben es, die Bibliothek auch aus der Sprache C++ zu benutzen. Danach folgen nur noch `#define`-Konstanten, ein `typedef` und Prototypen.

**Die gemeinsame Variable.** Der Bildpuffer gehört beiden `.c`-Dateien: `pbc_zeichnen.c` malt hinein, `pbc_geraet.c` schickt ihn ans Display. Er ist also genau so verteilt wie `anzahl_figuren` oben:

pbc_intern.h

```c
// pbc_intern.h – was die Teile der Bibliothek untereinander teilen.
// Programme brauchen diese Datei nicht.

#ifndef PBC_INTERN_H
#define PBC_INTERN_H

#include <stdint.h>
#include "pbc.h"

// Der Bildpuffer: PBC_WIDTH × PBC_HEIGHT Farben, Zeile für Zeile.
extern uint16_t pbc_bildpuffer[PBC_WIDTH * PBC_HEIGHT];

// Vom Geräte- bzw. PC-Teil geliefert:
uint32_t pbc_zufall32(void);      // 32 zufällige Bits
int pbc_zeichen_lesen(void);      // wartet auf ein Zeichen aus der Konsole

#endif // PBC_INTERN_H
```

Zeile 11 deklariert den Bildpuffer mit `extern`. Definiert ist er in `pbc_zeichnen.c`:

pbc_zeichnen.c

```c
// pbc_zeichnen.c – alles, was in den Bildpuffer zeichnet
//
// Dieser Teil kennt keine Hardware: Er schreibt nur Zahlen in das Feld
// pbc_bildpuffer. Linien, Ellipsen und Schriften zeichnen dieselben Pixel
// wie pbc in MicroPython auf einem Canvas.

#include "pbc.h"
#include "pbc_intern.h"
#include "schrift8.h"
#include "schrift12.h"
#include "schrift18.h"
#include "schrift24.h"

uint16_t pbc_bildpuffer[PBC_WIDTH * PBC_HEIGHT];
```

Weil der Bildpuffer nicht in `pbc.h` steht, sondern nur in `pbc_intern.h`, sehen deine Programme ihn nicht. Sie zeichnen über die Funktionen der Bibliothek. Wie eine solche Funktion aussieht, zeigt `pbc_pixel`:

pbc_zeichnen.c

```c
void pbc_pixel(int x, int y, uint16_t color) {
    if (x < 0 || x >= PBC_WIDTH || y < 0 || y >= PBC_HEIGHT) return;
    pbc_bildpuffer[y * PBC_WIDTH + x] = color;
}
```

Zuerst prüft sie, ob der Punkt auf dem Bildschirm liegt. Wenn nicht, kehrt sie sofort zurück. Darum kannst du Figuren zeichnen, die über den Rand ragen, ohne über das Ende des Bildpuffers hinaus zu schreiben (Kapitel 12.5). Die Bibliothek schreibt hier übrigens das `if` mit `return` ohne geschweifte Klammern in eine Zeile. Das ist erlaubt, aber im Buch halten wir es anders: Mit Klammern passiert es nicht, dass eine später ergänzte Zeile versehentlich außerhalb des `if` landet.

**Die Hardware.** `pbc_geraet.c` beginnt mit der Pinbelegung als `#define`-Konstanten. Die Status-LEDs sind so einfach, dass du ihren Teil schon ganz verstehst:

pbc_geraet.c

```
231 // ---- Status-LEDs (aktiv HIGH) ------------------------------------------------
      -------
232
233 static void led_init(uint pin) {
234     gpio_init(pin);
235     gpio_set_dir(pin, GPIO_OUT);
236     gpio_put(pin, 0);
237 }
238
239 void pbc_led_red(bool on) { gpio_put(PIN_LED_ROT, on); }
240 void pbc_led_yellow(bool on) { gpio_put(PIN_LED_GELB, on); }
241 void pbc_led_green(bool on) { gpio_put(PIN_LED_GRUEN, on); }
242 void pbc_leds_off(void) {
243     pbc_led_red(false);
244     pbc_led_yellow(false);
245     pbc_led_green(false);
246 }
```

`led_init` ist `static`, also nur innerhalb der Bibliothek zu benutzen. `pbc_init` ruft sie für jede der drei LEDs auf. `gpio_init`, `gpio_set_dir` und `gpio_put` sind Funktionen des Pico SDK. Sie stellen einen Anschluss des Prozessors als Ausgang ein und schalten ihn an oder aus. Kapitel 21 erklärt sie genau. Die Kapitel 22 bis 24 nehmen sich den Rest der Datei vor: den Neigungssensor, das Display und die Tasten-Interrupts.

> **Eine Bibliothek, zwei Rückseiten.** `pbc_zeichnen.c` benutzt keine Hardware: Es rechnet nur Zahlen in ein Array. Diese Trennung hat einen Vorteil, von dem das Buch lebt. Für die Bilder und Konsolenausgaben in diesem Buch wurde jedes Programm auch auf einem PC gebaut, mit `pbc_zeichnen.c` und einer PC-Fassung statt `pbc_geraet.c`. Dein Programm merkt davon nichts, denn es kennt nur `pbc.h`. Eine Header-Datei ist ein **Vertrag**: Solange die Funktionen tun, was in `pbc.h` steht, darf dahinter stecken, was immer nötig ist.

## Ausprobieren: Eine eigene Bibliothek

Jetzt baust du eine Bibliothek wie `pbc`, nur kleiner: `anzeige` mit Bausteinen für einen Statusbildschirm. Sie bekommt einen eigenen Ordner mit eigener `CMakeLists.txt`. So lässt sie sich mit einem Handgriff in ein anderes Projekt kopieren. Das Projekt `projekte/k20_anzeige/` sieht so aus:

```
k20_anzeige/
    CMakeLists.txt
    main.c
    anzeige/
        CMakeLists.txt
        anzeige.h
        anzeige.c
```

anzeige.h

```c
// anzeige.h — Bausteine für Anzeigen: Überschrift, Balken, Zahl
#ifndef ANZEIGE_H
#define ANZEIGE_H

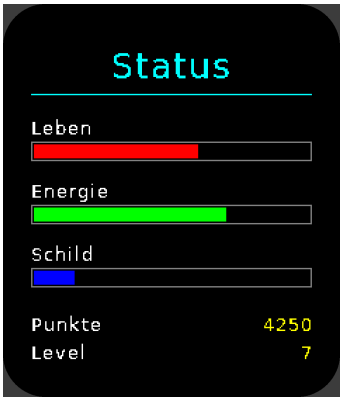
#include <stdint.h>

// Überschrift in der Mitte, mit einer Linie darunter
void anzeige_titel(const char *text, int y, uint16_t farbe);

// Name links, darunter ein Balken: wert von maximum, voll = 200 Pixel
void anzeige_balken(const char *name, int y, int wert, int maximum, uint16_t
    farbe);

// Name links und eine Zahl rechtsbündig in einer Zeile
void anzeige_zahl(const char *name, int y, int wert);

#endif
```




anzeige.c

```c
// anzeige.c – die Bausteine aus anzeige.h
#include "pbc.h"
#include "anzeige.h"

#define LINKS 20
#define BREITE 200

// x, bei dem ein Text der Breite breite genau in der Mitte steht
static int mitte(int breite) {
    return (PBC_WIDTH - breite) / 2;
}

void anzeige_titel(const char *text, int y, uint16_t farbe) {
    pbc_text24(text, mitte(pbc_text24_width(text)), y, farbe);
    pbc_hline(LINKS, y + 34, BREITE, farbe);
}

void anzeige_balken(const char *name, int y, int wert, int maximum, uint16_t
  farbe) {
    if (wert < 0) {
        wert = 0;
    }
    if (wert > maximum) {
        wert = maximum;
    }
    pbc_text12(name, LINKS, y, PBC_WHITE);
    pbc_rect(LINKS, y + 18, BREITE, 14, PBC_GRAY);
    pbc_fill_rect(LINKS + 2, y + 20, (BREITE - 4) * wert / maximum, 10, farbe);
}

void anzeige_zahl(const char *name, int y, int wert) {
    char text[12];
    snprintf(text, sizeof text, "%d", wert);
    pbc_text12(name, LINKS, y, PBC_WHITE);
    pbc_text12(text, LINKS + BREITE - pbc_text12_width(text), y, PBC_YELLOW);
}
```

- `LINKS` und `BREITE` in den Zeilen 5 und 6 und die Hilfsfunktion `mitte` braucht nur die Bibliothek selbst. Darum stehen sie in der `.c`-Datei, `mitte` mit `static`.

- `anzeige_balken` begrenzt den Wert selbst. Wer die Bibliothek benutzt, muss sich darum nicht kümmern.

Die Bauanleitung im Ordner `anzeige` ist fast eine Kopie der von `pbc`:

CMakeLists.txt

```
# Die Bibliothek "anzeige": fertige Bausteine für Anzeigen auf dem Display
add_library(anzeige STATIC
    anzeige.c
)
# Wer anzeige benutzt, findet anzeige.h in diesem Ordner.
target_include_directories(anzeige PUBLIC ${CMAKE_CURRENT_LIST_DIR})
# anzeige zeichnet mit pbc.
target_link_libraries(anzeige PUBLIC pbc)
target_compile_options(anzeige PRIVATE -Wall -Wextra)
```

Zeile 8 ist neu: `anzeige` zeichnet mit Funktionen aus `pbc` und braucht deshalb selbst die Bibliothek `pbc`. Zeile 9 schaltet die Warnungen auch für die neue Bibliothek ein. In der `CMakeLists.txt` des Projekts kommen zwei Dinge dazu:

```
add_subdirectory(anzeige)

add_executable(mein_programm main.c)
target_link_libraries(mein_programm pbc anzeige)
```

`main.c` bleibt eine einzige Datei und liest sich fast wie ein Text:

main.c

```c
#include "pbc.h"
#include "anzeige.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    anzeige_titel("Status", 30, PBC_CYAN);
    anzeige_balken("Leben", 80, 3, 5, PBC_RED);
    anzeige_balken("Energie", 125, 70, 100, PBC_GREEN);
    anzeige_balken("Schild", 170, 15, 100, PBC_BLUE);
    anzeige_zahl("Punkte", 220, 4250);
    anzeige_zahl("Level", 240, 7);
    pbc_show();

    while (true) {
        sleep_ms(1000);
    }
}
```

**Zum Verändern:** Ergänze in der Bibliothek eine Funktion `anzeige_text(const char *text, int y, uint16_t farbe)`, die einen Text in der Mitte zeigt, und benutze sie in `main.c` für eine Fußzeile.

## Übungsaufgaben

**Aufgabe 1 ⭐ Was gehört wohin?** In welche Datei gehören die folgenden Zeilen einer Bibliothek `musik`: `musik.h` oder `musik.c`?

1. `void spiele_melodie(int nummer);`
2. `void spiele_melodie(int nummer) { … }`
3. `#ifndef MUSIK_H`
4. `#define ANZAHL_MELODIEN 4`
5. `static const int melodie_1[] = { 262, 294, 330 };`
6. `extern int lautstaerke;`
7. `int lautstaerke = 5;`

**Aufgabe 2 ⭐ Ein Auto für die Figuren** Ergänze das Projekt `k20_figuren` um eine Figur `zeichne_auto(int x, int y, uint16_t farbe)` und stelle ein blaues Auto auf die Wiese. Welche drei Dateien musst du ändern? Welche nicht?

**Aufgabe 3 ⭐⭐ Meldungen lesen** Ein Projekt besteht aus `main.c`, `spieler.c` und `spieler.h`. Beim Bauen erscheint `undefined reference to 'spieler_bewegen'`. Nenne drei mögliche Ursachen. Was meldet der Compiler dagegen, wenn in `main.c` die Zeile `#include "spieler.h"` fehlt?

**Aufgabe 4 ⭐⭐ Den Würfel auslagern** Verteile dieses Programm auf `main.c`, `wuerfel.h` und `wuerfel.c` und trage `wuerfel.c` in die `CMakeLists.txt` ein. Was gehört in `wuerfel.h`, was nicht? Passt der Name `GROESSE` noch, wenn er in einer Header-Datei steht?

k20_aufgabe4.c

```c
#include "pbc.h"

#define GROESSE 70

static void auge(int x, int y, int spalte, int zeile) {
    pbc_fill_circle(x + 15 + spalte * 20, y + 15 + zeile * 20, 6,
      PBC_BLACK);
}

// zeichnet einen Wuerfel mit 1 bis 6 Augen; (x, y) ist die Ecke links oben
void zeichne_wuerfel(int x, int y, int augen) {
    pbc_fill_rect(x, y, GROESSE, GROESSE, PBC_WHITE);
    if (augen % 2 == 1) {
        auge(x, y, 1, 1);                          // Mitte
    }
```

```
    if (augen >= 2) {
        auge(x, y, 0, 0);
        auge(x, y, 2, 2);
    }
    if (augen >= 4) {
        auge(x, y, 2, 0);
        auge(x, y, 0, 2);
    }
    if (augen == 6) {
        auge(x, y, 0, 1);
        auge(x, y, 2, 1);
    }
}

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLUE);
    zeichne_wuerfel(35, 105, 5);
    zeichne_wuerfel(135, 105, 6);
    pbc_show();

    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 5 ⭐⭐⭐ Detektiv in `pbc`** Beantworte mit einem Blick in die Dateien der Bibliothek:

1. An welchem Pin hängt das Hintergrundlicht des Displays?
2. Was macht `pbc_rgb`, wenn du für Rot 300 angibst?
3. Wie viele Pixel breit ist bei `pbc_text` jedes Zeichen, und woran sieht man das in `pbc_text_width`?
4. Welche Funktionen ruft `pbc_init` auf, und warum schaltet sie das Hintergrundlicht erst ganz am Ende ein?

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- **`.h`-Datei**: Deklarationen (Prototypen, `extern`-Variablen, Konstanten, Typen) mit **Include-Wächter** `#ifndef` / `#define` / `#endif`. **`.c`-Datei**: Definitionen.
- `static` vor einer Funktion oder globalen Variable: nur in dieser Datei sichtbar.
- Eine globale Variable wird in **einer** `.c`-Datei definiert und im Header mit `extern` angekündigt.
- Der Compiler übersetzt jede `.c`-Datei einzeln, der **Linker** fügt alles zusammen. `undefined reference` und `multiple definition` sind Linker-Fehler.
- Neue `.c`-Dateien in `add_executable(mein_programm main.c …)` eintragen. Eigene Bibliothek: `add_library` + `target_include_directories` in einer eigenen `CMakeLists.txt`, im Projekt `add_subdirectory` und `target_link_libraries`.
- `pbc` ist genau so gebaut: `pbc.h` als Vertrag, `pbc_zeichnen.c` ohne Hardware, `pbc_geraet.c` mit Hardware.

KAPITEL 21

# GPIO, PWM und ADC – die Hardware direkt

**Worum geht's?** Bisher hat `pbc` für dich mit der Hardware gesprochen. Jetzt machst du es selbst, mit den Funktionen des Pico SDK. Du schaltest die LEDs über ihre Anschlüsse, liest Tasten, dimmst LEDs und das Display mit **PWM**, erzeugst Töne selbst und misst mit dem **ADC** die Spannung der Batterien. Dabei siehst du, was in `pbc_geraet.c` hinter `pbc_led_red`, `pbc_pressed_a`, `pbc_backlight`, `pbc_tone` und `pbc_battery_voltage` steckt.

> Die Programme dieses Kapitels benutzen Funktionen des Pico SDK, die es nur auf dem PicoBoy gibt. Deshalb zeigt das Buch hier keine Bildschirmbilder und Konsolenausgaben; probiere alles am Gerät aus. Alle Programme rufen trotzdem zuerst `pbc_init()` auf. Damit bleiben Display und Konsole nutzbar, und du arbeitest **neben** der Bibliothek. Was `pbc_init` dabei schon eingerichtet hat, steht jeweils dabei.

## 21.1 Ein Pin als Ausgang: die LEDs ohne `pbc`

Der RP2350 hat 30 Anschlüsse, an denen er Spannungen ausgeben oder messen kann: **GP0** bis **GP29**. Sie heißen **GPIO** (*General Purpose Input/Output*, Ein- und Ausgänge für alles Mögliche) oder kurz **Pins**. Auf dem PicoBoy ist jeder Pin fest mit einem Bauteil verbunden. Die Pinbelegung steht am Anfang von `pbc_geraet.c`; hier die wichtigsten:

| Pin | Bauteil | Pin | Bauteil |
|---|---|---|---|
| GP0–GP4 | Joystick (Mitte, rechts, runter, links, hoch) | GP15 | Lautsprecher |
| GP27, GP28 | Tasten A und B | GP26 | Hintergrundlicht des Displays |
| GP12, GP13, GP14 | LEDs grün, gelb, rot | GP20, GP21 | I2C zum Neigungssensor (Kapitel 22) |
| GP11 | RGB-LED | GP29 | Batteriespannung |

Ein Pin als **Ausgang** liefert entweder 3,3 Volt (**HIGH**, 1) oder 0 Volt (**LOW**, 0). Die LEDs sind so angeschlossen, dass sie bei HIGH leuchten (**aktiv HIGH**):

k21_led_direkt.c

```c
#include "pbc.h"
#include "hardware/gpio.h"

#define PIN_LED_ROT 14
```

```c
int main(void) {
    pbc_init();
    // pbc_init hat den Pin schon eingerichtet. Ohne pbc ginge es so:
    gpio_init(PIN_LED_ROT);                // Pin als einfacher Ein-/Ausgang
    gpio_set_dir(PIN_LED_ROT, GPIO_OUT);   // Richtung: Ausgang

    while (true) {
        gpio_put(PIN_LED_ROT, 1);          // HIGH: 3,3 V, die LED leuchtet
        sleep_ms(500);
        gpio_put(PIN_LED_ROT, 0);          // LOW: 0 V, die LED ist aus
        sleep_ms(500);
    }
}
```

- `#include "hardware/gpio.h"` in Zeile 2 holt die GPIO-Funktionen des SDK. Streng genommen kämen sie auch über `pbc.h` und `pico/stdlib.h` schon mit (Kapitel 20.3). Die Zeile zeigt aber, woher die Funktionen stammen. Für PWM, ADC und I2C sind solche Zeilen dagegen nötig.
- `gpio_init` in Zeile 9 macht den Pin zu einem einfachen Ein- und Ausgang des Prozessors. Danach ist er erst einmal ein Eingang und auf LOW gestellt.
- `gpio_set_dir` in Zeile 10 macht ihn zum Ausgang (`GPIO_OUT`).
- `gpio_put(pin, 1)` legt 3,3 V an, `gpio_put(pin, 0)` 0 V.

Die Zeilen 9 und 10 könntest du hier weglassen: Genau dasselbe tut die Funktion `led_init` in `pbc_geraet.c` für alle drei LEDs schon in `pbc_init`. Und `pbc_led_red(on)` ist nichts anderes als `gpio_put(PIN_LED_ROT, on)`.

**Mehrere Pins auf einmal.** Jeder Pin hat eine Nummer, und der Prozessor verwaltet alle Pins als Bits einer einzigen 32-Bit-Zahl: Bit 12 gehört zu GP12, Bit 13 zu GP13 und so weiter. Mit einer **Maske**, einer Zahl, in der genau die gewünschten Bits 1 sind, schaltest du mehrere Pins mit einem einzigen Befehl:

k21_lauflicht.c

```c
#include "pbc.h"
#include "hardware/gpio.h"

#define PIN_LED_ROT   14
#define PIN_LED_GELB  13
#define PIN_LED_GRUEN 12

// je ein Bit fuer jede der drei LEDs
#define LED_MASKE ((1u << PIN_LED_ROT) | (1u << PIN_LED_GELB) | (1u <<
    PIN_LED_GRUEN))

int main(void) {
    pbc_init();
    const uint pins[3] = { PIN_LED_GRUEN, PIN_LED_GELB, PIN_LED_ROT };
```

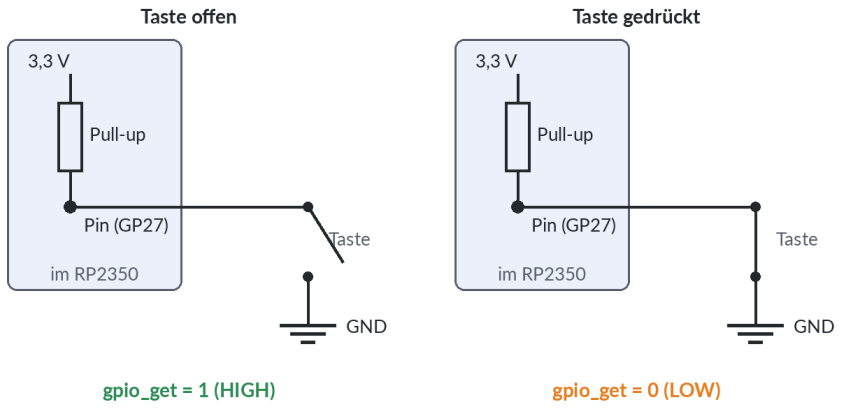
```
15      while (true) {
16          for (int i = 0; i < 3; i++) {
17              // alle drei LED-Pins auf einmal: nur der von pins[i] wird HIGH
18              gpio_put_masked(LED_MASKE, 1u << pins[i]);
19              sleep_ms(200);
20          }
21          gpio_set_mask(LED_MASKE);          // alle an
22          sleep_ms(400);
23          gpio_clr_mask(LED_MASKE);          // alle aus
24          sleep_ms(400);
25      }
26  }
```

**Bits schieben und verknüpfen.** `1u << 14` schiebt eine 1 um 14 Stellen nach links: Das Ergebnis ist eine Zahl, in der nur Bit 14 gesetzt ist (`u` steht für *unsigned*, ohne Vorzeichen). Der Operator `|` (**oder**, bitweise) verbindet zwei Zahlen so, dass jedes Bit gesetzt ist, das in einer der beiden gesetzt ist. `(1u << 14) | (1u << 13) | (1u << 12)` ist also eine Zahl mit genau drei gesetzten Bits, für die drei LEDs. Den Operator `&` (**und**, bitweise) siehst du in Kapitel 22: Er lässt nur die Bits übrig, die in beiden Zahlen gesetzt sind.

- Zeile 9 baut die Maske für die drei LED-Pins.
- `gpio_put_masked(maske, wert)` in Zeile 18 stellt **nur** die Pins der Maske ein, und zwar so, wie es die Bits in `wert` sagen. Alle anderen Pins bleiben, wie sie sind. Hier ist in `wert` nur das Bit einer LED gesetzt: Sie geht an, die beiden anderen gehen aus.
- `gpio_set_mask` schaltet alle Pins der Maske auf HIGH, `gpio_clr_mask` alle auf LOW (*clear*, löschen).

## 21.2 Ein Pin als Eingang: Pull-up, aktiv LOW, Entprellen

Ein Pin als **Eingang** misst, ob außen eine hohe oder eine niedrige Spannung anliegt. Die Tasten des PicoBoy verbinden ihren Pin beim Drücken mit **GND**, also 0 V. Ist die Taste offen, hinge der Pin aber in der Luft, und der Eingang läse zufällig mal 1, mal 0. Deshalb schaltet man im Chip einen **Pull-up-Widerstand** zu. Er zieht den Pin schwach auf 3,3 V hoch. Drückt man die Taste, gewinnt die direkte Verbindung zu GND:



Eine offene Taste liest also 1, eine gedrückte 0. Das nennt man **aktiv LOW**: Der aktive Zustand „gedrückt“ ist der niedrige Pegel.

k21_taste_direkt.c

```c
#include "pbc.h"
#include "hardware/gpio.h"

#define PIN_TASTE_A 27
#define PIN_LED_GRUEN 12

int main(void) {
    pbc_init();
    // Auch das hat pbc_init schon erledigt:
    gpio_init(PIN_TASTE_A);
    gpio_set_dir(PIN_TASTE_A, GPIO_IN);    // Richtung: Eingang
    gpio_pull_up(PIN_TASTE_A);             // ohne Druck: HIGH

    while (true) {
        bool pegel = gpio_get(PIN_TASTE_A);
        // aktiv LOW: gedrueckt ist die Taste, wenn der Pin LOW ist
        gpio_put(PIN_LED_GRUEN, !pegel);

        pbc_clear(PBC_BLACK);
        pbc_text18("GP27 ist", 70, 100, PBC_WHITE);
        if (pegel) {
            pbc_text24("HIGH", 85, 135, PBC_GREEN);
        } else {
            pbc_text24("LOW", 92, 135, PBC_RED);
        }
        pbc_show();
        sleep_ms(20);
    }
}
```

- Zeile 11 macht den Pin zum Eingang, Zeile 12 schaltet den Pull-up ein. Auch das hat `pbc_init` schon getan, für alle sieben Tasten.
- `gpio_get` in Zeile 15 liefert `true` für HIGH und `false` für LOW.
- Weil die Taste aktiv LOW ist, steht in Zeile 17 ein `!`: Die grüne LED leuchtet, solange der Pin LOW ist.

In `pbc_geraet.c` heißt das genauso: `static bool gedrueckt(int i) { return !gpio_get(tasten_pins[i]); }`. Alle `pbc_pressed_…`-Funktionen rufen diese Hilfsfunktion auf.

**Prellen.** Eine Taste ist ein kleines Stück Metall, das beim Drücken auf einen Kontakt schlägt. Dabei kann es ein paar Mal zurückfedern, bevor es liegen bleibt. Für einige Millisekunden wechselt der Pin dann mehrmals zwischen HIGH und LOW. Das heißt **Prellen**. Ein Programm, das einfach jeden Wechsel von HIGH nach LOW zählt, zählt dann manchmal einen Druck doppelt. Das passiert beim Drücken und auch beim Loslassen. Dagegen hilft **Entprellen**: Ein Druck zählt nur, wenn der Pin davor eine Weile ruhig war.

k21_prellen.c

```c
#include "pbc.h"
#include "hardware/gpio.h"

#define PIN_TASTE_A 27
#define ENTPRELLEN_US 50000

int main(void) {
    pbc_init();
    int roh = 0;                  // jede fallende Flanke
    int entprellt = 0;            // nur nach 50 ms Ruhe
    bool vorher = true;           // HIGH = nicht gedrueckt
    uint32_t letzte_flanke = 0;
    uint32_t letzte_anzeige = 0;

    while (true) {
        bool jetzt = gpio_get(PIN_TASTE_A);
        if (jetzt != vorher) {                        // eine Flanke
            if (!jetzt) {                             // HIGH -> LOW
                roh++;
                if (time_us_32() - letzte_flanke > ENTPRELLEN_US) {
                    entprellt++;
                }
            }
            letzte_flanke = time_us_32();             // auch beim Loslassen
        }
        vorher = jetzt;

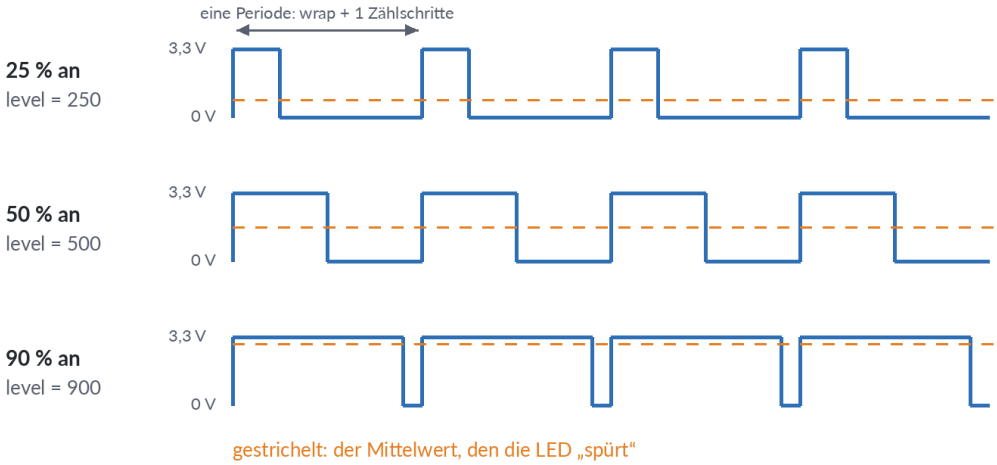
        if (time_us_32() - letzte_anzeige > 100000) { // 10-mal pro Sekunde
            letzte_anzeige = time_us_32();
            char text[32];
            pbc_clear(PBC_BLACK);
            snprintf(text, sizeof text, "roh: %d", roh);
            pbc_text24(text, 40, 90, PBC_YELLOW);
            snprintf(text, sizeof text, "entprellt: %d", entprellt);
            pbc_text24(text, 40, 140, PBC_GREEN);
            pbc_show();
        }
    }
}
```

- `vorher` merkt sich den Pegel vom letzten Durchlauf. Ist er jetzt anders (Zeile 17), hat der Pin eine **Flanke**. Ist er jetzt LOW (Zeile 18), ist es eine **fallende Flanke**: Die Taste wurde gerade gedrückt, oder sie prellt.
- `roh` zählt jede fallende Flanke.
- `entprellt` zählt eine fallende Flanke nur, wenn die vorige Flanke mehr als 50 ms her ist (Zeile 20). Zeile 24 merkt sich die Zeit **jeder** Flanke, auch beim Loslassen: Sonst würde das Federn beim Loslassen nach einem langen Druck als neuer Druck zählen. `time_us_32()` liefert die Zeit in Mikrosekunden seit dem Start.
- Das Display wird nur zehnmal pro Sekunde neu gezeichnet (Zeile 28). In der übrigen Zeit fragt die Schleife den Pin so oft ab, wie sie kann. Während `pbc_show` (gut 31 ms) sieht sie den Pin allerdings nicht; ein Prellen in dieser Zeit entgeht ihr.

Drücke A ein paarmal, halte die Taste auch mal länger, und vergleiche die beiden Zahlen. Wie oft die Tasten des PicoBoy prellen, hängt von der einzelnen Taste ab und davon, wie du drückst. Bei der Prüfung am Gerät prellten oben, links und A gelegentlich beim Loslassen. Die Bibliothek entprellt ebenfalls mit 50 ms Ruhe, merkt sich die Flanken aber mit einem **Interrupt**: Der Prozessor unterbricht das laufende Programm von selbst, sobald eine Flanke kommt. So verpasst `pbc_was_pressed_…` keinen Druck, auch nicht während `pbc_show`. Interrupts sind Thema von Kapitel 24.

## 21.3 PWM: LED-Helligkeit, Hintergrundlicht, Töne selbst erzeugen

Ein Pin kennt nur HIGH und LOW, nichts dazwischen. Trotzdem kann man eine LED halb hell leuchten lassen: Man schaltet sie sehr schnell an und aus, tausende Male pro Sekunde. Das Auge sieht kein Flackern, sondern eine mittlere Helligkeit. Ist die LED ein Viertel der Zeit an, wirkt sie ungefähr ein Viertel so hell. Das heißt **Pulsweitenmodulation**, kurz **PWM**. Der Anteil der Zeit, in der der Pin HIGH ist, heißt **Tastverhältnis**.




Der Prozessor muss dafür nicht selbst schalten. Der RP2350 hat einen **PWM-Baustein** mit 12 **Slices** (Scheiben). Jede Slice hat einen Zähler, der immer wieder von 0 bis zu einem Wert namens **wrap** zählt und dann von vorn beginnt. Jede Slice hat zwei **Kanäle**, A und B, für zwei Pins. Für jeden Kanal gibt es einen Vergleichswert **level**: Solange der Zähler kleiner als `level` ist, ist der Pin HIGH, danach LOW. Welcher Pin zu welcher Slice gehört, ist fest verdrahtet; `pwm_gpio_to_slice_num(pin)` rechnet es aus. GP12 gehört zu Slice 6, Kanal A.

k21_pwm_led.c

```c
#include "pbc.h"
#include "hardware/pwm.h"

#define PIN_LED_GRUEN 12

int main(void) {
    pbc_init();
```

```
 8     // Der Pin wird vom PWM-Baustein gesteuert, nicht mehr von gpio_put.
 9     gpio_set_function(PIN_LED_GRUEN, GPIO_FUNC_PWM);
10     uint slice = pwm_gpio_to_slice_num(PIN_LED_GRUEN);   // GP12: Slice 6
11     pwm_set_wrap(slice, 999);              // der Zaehler laeuft von 0 bis 999
12     pwm_set_enabled(slice, true);
13
14     while (true) {
15         for (int stufe = 0; stufe <= 1000; stufe += 10) {
16             pwm_set_gpio_level(PIN_LED_GRUEN, stufe);    // stufe von 1000 an
17             sleep_ms(10);
18         }
19         for (int stufe = 1000; stufe >= 0; stufe -= 10) {
20             pwm_set_gpio_level(PIN_LED_GRUEN, stufe);
21             sleep_ms(10);
22         }
23     }
24 }
```

- Zeile 9 übergibt den Pin an den PWM-Baustein. Ab jetzt schaltet ihn die PWM, und `gpio_put` (also auch `pbc_led_green`) hat keine Wirkung mehr. Zurück geht es mit `gpio_set_function(PIN_LED_GRUEN, GPIO_FUNC_SIO)`.
- Zeile 11: Der Zähler läuft von 0 bis 999, also 1000 Schritte.
- Zeile 12 startet die Slice.
- `pwm_set_gpio_level` in Zeile 16 stellt `level` für den Kanal des Pins ein: 0 heißt immer aus, 500 halb an, 1000 immer an.

Wie schnell das geht: Der Zähler zählt mit dem Takt des Prozessors, 150 Millionen Schritte pro Sekunde. 1000 Schritte dauern also 1000 / 150 000 000 Sekunden; das Muster wiederholt sich 150 000-mal pro Sekunde. Allgemein gilt:

**Frequenz = 150 000 000 / (Teiler × (wrap + 1))**

Der **Teiler** verlangsamt den Zähler. Er ist 1, solange du ihn nicht mit `pwm_set_clkdiv` änderst.

> Das Auge empfindet Helligkeit nicht gleichmäßig: Zwischen `level` 0 und 100 ändert sich viel, zwischen 900 und 1000 kaum etwas. Ein gleichmäßigeres Aufleuchten bekommst du, wenn du `level` quadratisch wachsen lässt (Aufgabe 3).

**Das Hintergrundlicht.** `pbc_init` steuert das Licht des Displays an GP26 genauso, mit `wrap = 255` (Slice 5, Kanal A). `pbc_backlight(wert)` ist im Kern nur `pwm_set_gpio_level(PIN_DISPLAY_LICHT, wert)`. Das kannst du auch direkt tun:

k21_licht.c

```
1 #include "pbc.h"
2 #include "hardware/pwm.h"
3
4 #define PIN_DISPLAY_LICHT 26
5
6 int main(void) {
7     pbc_init();
```

```
    int stufe = 255;              // pbc_init stellt wrap = 255 ein
    while (true) {
        if (pbc_was_pressed_up() && stufe < 255) {
            stufe += 15;
        }
        if (pbc_was_pressed_down() && stufe > 0) {
            stufe -= 15;
        }
        pwm_set_gpio_level(PIN_DISPLAY_LICHT, stufe);

        char text[24];
        snprintf(text, sizeof text, "Licht: %d / 255", stufe);
        pbc_clear(PBC_WHITE);
        pbc_text18(text, 45, 110, PBC_BLACK);
        pbc_text12("hoch / runter", 75, 150, PBC_GRAY);
        pbc_show();
        sleep_ms(20);
    }
}
```

Mit dem Joystick nach oben oder unten änderst du die Helligkeit in Schritten von 15. Bei 0 ist das Display ganz dunkel; das Bild ist trotzdem noch da.

**Töne.** Ein Lautsprecher macht einen Ton, wenn seine Membran schnell hin und her schwingt. Schaltet man den Lautsprecher-Pin 440-mal pro Sekunde an und aus, hört man den Kammerton a mit 440 Hertz. Das geht zuerst ganz ohne PWM:

k21_ton_gpio.c

```
#include "pbc.h"
#include "hardware/gpio.h"

#define PIN_LAUTSPRECHER 15

int main(void) {
    pbc_init();
    // pbc_init hat GP15 als Ausgang auf LOW gestellt.
    int frequenz = 440;
    int halbe_periode_us = 1000000 / frequenz / 2;    // 1136 us

    while (true) {
        // eine Sekunde lang: HIGH, warten, LOW, warten ...
        for (int i = 0; i < frequenz; i++) {
            gpio_put(PIN_LAUTSPRECHER, 1);
            sleep_us(halbe_periode_us);
            gpio_put(PIN_LAUTSPRECHER, 0);
            sleep_us(halbe_periode_us);
        }
        sleep_ms(1000);                                  // eine Sekunde Pause
    }
}
```

Eine Schwingung dauert 1 / 440 Sekunden, das sind 2272 Mikrosekunden: 1136 Mikrosekunden HIGH, 1136 LOW. Das funktioniert, hat aber einen großen Nachteil: Während der Ton spielt, kann das Programm nichts anderes tun.

Mit PWM erzeugt die Hardware die Schwingung selbst, und das Programm läuft weiter. Für einen Ton braucht man ein Tastverhältnis von 50 % und die passende Frequenz:

k21_ton_pwm.c

```
 1 #include "pbc.h"
 2 #include "hardware/clocks.h"
 3 #include "hardware/pwm.h"
 4
 5 #define PIN_LAUTSPRECHER 15
 6
 7 // Rechteckton mit der PWM; frequenz von 20 bis 20000 Hz
 8 void ton(int frequenz) {
 9     uint slice = pwm_gpio_to_slice_num(PIN_LAUTSPRECHER);   // Slice 7
10     // Teiler so, dass der Zaehler mit 1 MHz zaehlt
11     pwm_set_clkdiv(slice, clock_get_hz(clk_sys) / 1000000.0f);
12     uint32_t schritte = 1000000 / frequenz;          // eine Periode
13     pwm_set_wrap(slice, schritte - 1);
14     pwm_set_gpio_level(PIN_LAUTSPRECHER, schritte / 2);   // halb an, halb aus
15     gpio_set_function(PIN_LAUTSPRECHER, GPIO_FUNC_PWM);
16     pwm_set_enabled(slice, true);
17 }
18
19 int main(void) {
20     pbc_init();
21     const int melodie[] = { 262, 330, 392, 523 };   // C, E, G, hohes C
22     while (true) {
23         for (int i = 0; i < 4; i++) {
24             ton(melodie[i]);
25             sleep_ms(300);
26         }
27         pbc_tone_off();           // PWM aus, Pin auf LOW
28         sleep_ms(1000);
29     }
30 }
```

- Zeile 11 stellt den Teiler auf 150 (`clock_get_hz(clk_sys)` liefert die Taktfrequenz, 150 000 000). Der Zähler zählt dann eine Million Schritte pro Sekunde, einen pro Mikrosekunde.
- Zeile 12: Eine Schwingung bei 262 Hz dauert 1 000 000 / 262 = 3816 Mikrosekunden, also Schritte. `wrap` ist eins weniger, weil der Zähler bei 0 beginnt.
- Zeile 14: `level` ist die Hälfte davon, halb an, halb aus.
- Zeile 27: `pbc_tone_off` aus der Bibliothek schaltet die PWM ab und legt den Pin auf LOW.

`pbc_tone` in `pbc_geraet.c` rechnet ähnlich, beginnt aber mit `wrap = 1000` und berechnet dazu einen Teiler mit Nachkommastellen. Außerdem stellt es einen **Wecker** (*alarm*), der den Ton nach `ms` Millisekunden ausschaltet, während dein Programm weiterläuft.

> Der Lautsprecher an GP15 ist Kanal B von Slice 7. Kanal A derselben Slice gehört zu GP14, der **roten LED**. Beide Kanäle teilen sich Zähler, Teiler und `wrap`. Dimmst du die rote LED mit PWM, ändert jeder Ton ihre Helligkeit, und umgekehrt. Nimm zum Dimmen deshalb die grüne oder die gelbe LED (Slice 6).

## 21.4 ADC: die Batteriespannung messen

Ein **ADC** (*Analog-Digital-Converter*, Analog-Digital-Wandler) misst eine Spannung und macht daraus eine Zahl. Der ADC des RP2350 liefert 12 Bit, also Zahlen von 0 bis 4095: 0 für 0 V, 4095 für 3,3 V. Er hat fünf Eingänge: Eingang 0 bis 3 gehören zu GP26 bis GP29, Eingang 4 ist ein Temperatursensor im Chip.

Der PicoBoy bekommt seinen Strom aus zwei AAA-Batterien oder über das USB-Kabel; welche Quelle, wählt der Schalter BAT/USB auf der Rückseite. Eingang 3 an GP29 misst die Spannung, die dabei ankommt. Am USB-Kabel sind das etwa 5 V, mehr als die 3,3 V, die der ADC messen kann. Deshalb liegt die Spannung über einen **Spannungsteiler** am Pin: zwei Widerstände, die sie auf ein Drittel verkleinern. Das Programm muss das Ergebnis also mal 3 nehmen:

k21_batterie.c

```c
#include "pbc.h"
#include "hardware/adc.h"

int main(void) {
    pbc_init();
    // pbc_init hat schon adc_init() und adc_gpio_init(29) aufgerufen.
    adc_select_input(3);                        // Eingang 3 = GP29

    while (true) {
        uint16_t roh = adc_read();              // 0 bis 4095
        float am_pin = roh * 3.3f / 4095.0f;    // Spannung am Pin
        float batterie = am_pin * 3.0f;         // Spannungsteiler 1:3

        char text[32];
        pbc_clear(PBC_BLACK);
        snprintf(text, sizeof text, "Rohwert: %d", roh);
        pbc_text18(text, 40, 70, PBC_WHITE);
        snprintf(text, sizeof text, "am Pin: %.3f V", am_pin);
        pbc_text18(text, 40, 110, PBC_WHITE);
        snprintf(text, sizeof text, "Batterie: %.2f V", batterie);
        pbc_text18(text, 40, 150, PBC_YELLOW);
        snprintf(text, sizeof text, "pbc:    %.2f V", pbc_battery_voltage());
        pbc_text18(text, 40, 190, PBC_GRAY);
        pbc_show();
        sleep_ms(500);
    }
}
```

- `pbc_init` hat schon `adc_init()` (den ADC einschalten) und `adc_gpio_init(29)` aufgerufen. `adc_gpio_init` schaltet am Pin alles Digitale ab, damit nichts die Messung stört.

- `adc_select_input(3)` in Zeile 7 wählt den Eingang. Es gibt nur einen Wandler; er misst immer den zuletzt gewählten Eingang.
- `adc_read()` in Zeile 10 startet eine Messung und wartet auf das Ergebnis, das dauert nur ein paar Mikrosekunden.
- Zeile 11 rechnet die Zahl in Volt um, Zeile 12 nimmt den Teiler heraus.

Der Rest des Programms zeigt die Werte an, und in der letzten Zeile zum Vergleich `pbc_battery_voltage()`. Die Bibliothek misst gleich, nimmt aber den Mittelwert aus 16 Messungen. Dadurch schwankt ihr Wert weniger als die Einzelmessung darüber.

Welche Werte kannst du erwarten? Die beiden Batterien sind **hintereinandergeschaltet**, ihre Spannungen zählen also zusammen:

| So läuft der PicoBoy | Spannung |
|---|---|
| zwei frische Alkali-Batterien (je 1,5–1,6 V) | etwa 3,0–3,2 V |
| zwei fast leere Alkali-Batterien (je 1,0–1,1 V) | etwa 2,0–2,2 V |
| zwei wiederaufladbare NiMH-Akkus (je etwa 1,2 V) | etwa 2,4–2,7 V |
| am USB-Kabel | etwa 5 V |

Die Spannung sinkt also, je leerer die Batterien werden. Aufladen kann der PicoBoy wiederaufladbare Akkus nicht; dafür brauchst du ein eigenes Ladegerät. Weil am USB-Kabel deutlich mehr als bei Batterien ankommt, beruht `pbc_is_on_usb()` auf dieser Messung: Es liefert `true`, wenn `pbc_battery_voltage()` mehr als 4,0 V misst.

## Ausprobieren: Ein Spannungsschreiber

Das Programm misst jede Sekunde die Spannung und zeichnet die letzten 200 Messungen als Kurve, so wie ein Fieberthermometer mit Gedächtnis. Lass es mit dem Schalter auf BAT eine Weile laufen, schiebe ihn dann auf USB (Kabel steckt) und sieh zu, wie die Kurve springt.

k21_schreiber.c

```c
#include "pbc.h"
#include "hardware/adc.h"

#define ANZAHL 200                  // so viele Messwerte passen nebeneinander
#define OBEN 60                     // y fuer 5,5 V
#define UNTEN 240                   // y fuer 1,5 V

float messe_batterie(void) {
    adc_select_input(3);
    uint32_t summe = 0;
    for (int i = 0; i < 16; i++) {     // Mittelwert aus 16 Messungen
        summe += adc_read();
    }
    return summe / 16.0f * 3.3f / 4095.0f * 3.0f;
}
```

```c
int main(void) {
    pbc_init();
    static float werte[ANZAHL];        // static: nicht auf dem Stack
    int anzahl = 0;
    while (true) {
        if (anzahl == ANZAHL) {        // voll: alles eins nach links
            for (int i = 1; i < ANZAHL; i++) {
                werte[i - 1] = werte[i];
            }
            anzahl--;
        }
        werte[anzahl] = messe_batterie();
        anzahl++;

        char text[24];
        snprintf(text, sizeof text, "%.2f V", werte[anzahl - 1]);
        pbc_clear(PBC_BLACK);
        pbc_text24(text, 75, 15, PBC_YELLOW);
        pbc_rect(19, OBEN - 1, ANZAHL + 2, UNTEN - OBEN + 2, PBC_GRAY);
        for (int i = 0; i < anzahl; i++) {
            int y = UNTEN - (int)((werte[i] - 1.5f) * (UNTEN - OBEN) / 4.0f);
            if (y >= OBEN && y <= UNTEN) {
                pbc_pixel(20 + i, y, PBC_GREEN);
            }
        }
        pbc_text("5,5 V", 24, OBEN + 4, PBC_GRAY);
        pbc_text("1,5 V", 24, UNTEN - 12, PBC_GRAY);
        pbc_show();
        sleep_ms(1000);
    }
}
```

- `messe_batterie` bildet wie die Bibliothek den Mittelwert aus 16 Messungen.
- Das Array `werte` ist `static` (Kapitel 19.4). Mit 200 `float` sind es 800 Bytes; auf dem Stack wäre das schon ein Fünftel des Platzes.
- Ist das Array voll, rücken alle Werte eins nach links, der älteste fällt heraus. So läuft die Kurve von rechts nach links durchs Bild.
- Die Höhe rechnet die Spannung zwischen 1,5 V (unten) und 5,5 V (oben) auf die 180 Pixel zwischen `UNTEN` und `OBEN` um. So passen Batterien (2 bis 3,2 V) und USB (etwa 5 V) in dasselbe Bild.
- Mit Batterien sinkt die Kurve nur sehr langsam. In 200 Sekunden siehst du kaum eine Änderung; leer werden die Batterien erst nach Stunden.

**Zum Verändern:** Färbe die Kurve rot, sobald die Spannung unter einen Wert fällt, den du selbst wählst (zum Beispiel 2,2 V: dann sind die Batterien fast leer), und lass dann zusätzlich die rote LED leuchten.

## Übungsaufgaben

**Aufgabe 1 ⭐ Ampel ohne `pbc`** Lass die drei LEDs eine Ampel spielen: rot, rot und gelb, grün, gelb, und wieder von vorn. Benutze nur `gpio_put`, keine `pbc_led_…`-Funktion.

**Aufgabe 2 ⭐ Taste B direkt** Die gelbe LED soll leuchten, solange B gedrückt ist. Lies die Taste mit `gpio_get`. An welchem Pin hängt B?

**Aufgabe 3 ⭐⭐ Atmende LED** Lass die gelbe LED langsam heller und wieder dunkler werden, wie ein ruhiger Atem. Wähle `wrap = 9999` und lass `i` von 0 bis 100 laufen; als `level` nimmst du `i * i`. Vergleiche mit einem linearen Verlauf.

**Aufgabe 4 ⭐⭐ Tonleiter** Spiele mit der Funktion `ton` aus 21.3 die C-Dur-Tonleiter hinauf und wieder hinunter. Die Frequenzen: 262, 294, 330, 349, 392, 440, 494 und 523 Hz.

**Aufgabe 5 ⭐⭐⭐ Wie warm ist der Chip?** Der Temperatursensor hängt an ADC-Eingang 4. Er muss vorher mit `adc_set_temp_sensor_enabled(true)` eingeschaltet werden. Laut Datenblatt des Herstellers gilt für die gemessene Spannung U: Temperatur = 27 − (U − 0,706) / 0,001721 (in Grad Celsius). Zeige Spannung und Temperatur jede Sekunde an. Was passiert, wenn du den PicoBoy eine Weile in der Hand hältst?

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- **Ausgang:** `gpio_init(pin)`, `gpio_set_dir(pin, GPIO_OUT)`, `gpio_put(pin, 1/0)`. Mehrere Pins: Maske mit `1u << pin` und `|`, dann `gpio_put_masked`, `gpio_set_mask`, `gpio_clr_mask`.
- **Eingang:** `gpio_set_dir(pin, GPIO_IN)`, `gpio_pull_up(pin)`, `gpio_get(pin)`. Die Tasten sind **aktiv LOW**: gedrückt = 0.
- Tasten **prellen**; man entprellt, indem man nach einem Druck eine Weile (z. B. 50 ms) keine weitere Flanke zählt.
- **PWM:** `gpio_set_function(pin, GPIO_FUNC_PWM)`, Slice mit `pwm_gpio_to_slice_num`, `pwm_set_wrap`, `pwm_set_enabled`, `pwm_set_gpio_level`. Frequenz = 150 MHz / (Teiler × (wrap + 1)). Rote LED und Lautsprecher teilen sich Slice 7.
- **ADC:** 12 Bit, 0–4095 für 0–3,3 V. `adc_select_input(3)` und `adc_read()` messen die Batteriespannung an GP29; wegen des Teilers mal 3. Zwei AAA-Batterien: frisch etwa 3,0–3,2 V, fast leer etwa 2,0–2,2 V; am USB etwa 5 V.
- `pbc_init` hat alle diese Pins schon eingerichtet. Die SDK-Funktionen funktionieren daneben; wer einen Pin auf PWM umstellt, nimmt ihn der Bibliothek weg.

KAPITEL 22

# I2C: mit dem Neigungssensor sprechen

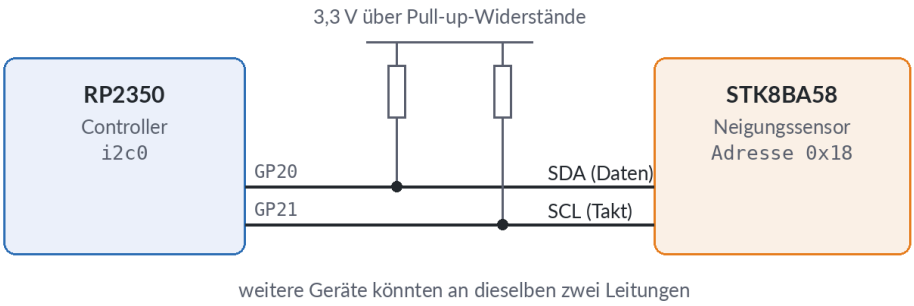
**Worum geht's?** Der Neigungssensor ist ein eigener kleiner Chip. Der RP2350 unterhält sich mit ihm über zwei Leitungen, den **I2C-Bus**. Du lernst, wie so ein Gespräch abläuft, suchst mit einem **I2C-Scan** nach allen Geräten am Bus, liest die Kennung des Sensors und seine Messwerte und rechnest sie selbst in g um. Am Ende weißt du, was `pbc_accel_x()` und seine Geschwister tun.

> Wie in Kapitel 21 laufen alle Programme nur auf dem PicoBoy; Bildschirm und Konsole zeigst du dir dort an. `pbc_init` hat den I2C-Bus schon eingerichtet. Die Programme benutzen ihn einfach mit.

## 22.1 Bus, Adresse, Register

**I2C** (gesprochen „I-Quadrat-C", englisch *I-squared-C*) ist eine Verbindung für Chips auf derselben Platine. Sie braucht nur zwei Leitungen:

- **SDA** (*serial data*): Über sie laufen die Daten, Bit für Bit.
- **SCL** (*serial clock*): der Takt. Bei jedem Takt gilt ein Bit auf SDA.




Beide Leitungen werden über **Pull-up-Widerstände** auf 3,3 V gehalten (wie die Tasten in 21.2). Wer etwas senden will, zieht sie kurz auf LOW. So können mehrere Geräte an denselben zwei Leitungen hängen, ohne sich zu stören. Darum heißt es **Bus**.

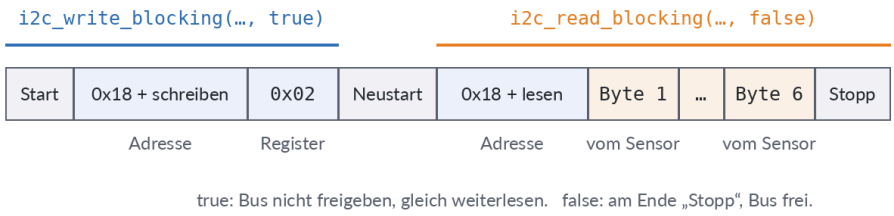
Ein Gerät am Bus führt das Gespräch: der **Controller**, hier der RP2350. Er erzeugt den Takt und spricht die anderen Geräte an. Damit klar ist, wen er meint, hat jedes Gerät eine **Adresse**, eine Zahl mit 7 Bit (0 bis 127). Der Neigungssensor STK8BA58 hat die Adresse **0x18**. Hexadezimal 0x18 ist 1 · 16 + 8 = 24.

Im Sensor gibt es eine Reihe von **Registern**: kleine Speicherzellen mit je einem Byte, wieder mit Nummern. Einige enthalten Messwerte, in andere schreibt man Einstellungen. Die wichtigsten des STK8BA58:

| Register | Inhalt |
|---|---|
| `0x00` | Kennung des Chips (*Chip-ID*): 0x87 |
| `0x02`, `0x03` | Messwert der x-Achse (unteres, oberes Byte) |
| `0x04`, `0x05` | Messwert der y-Achse |
| `0x06`, `0x07` | Messwert der z-Achse |
| `0x0F` | Messbereich: `0x03` = ±2 g, `0x05` = ±4 g, `0x08` = ±8 g |
| `0x10` | Bandbreite, also wie schnell neue Messwerte kommen |
| `0x11` | Betriebsart; `0x00` = normaler Betrieb |

Die Tabelle folgt den Treibern, die es für den PicoBoy schon gibt: der Firmware für MicroPython, der Buchbibliothek und einer Arduino-Bibliothek für den Sensor. Alle drei verwenden dieselben Registernummern und Werte.

**So liest man ein Register.** Der Controller schreibt dem Sensor zuerst die Nummer des Registers, das er lesen will. Dann liest er, ohne den Bus zwischendurch freizugeben. Der Sensor liefert das Register und, wenn der Controller weiterliest, die folgenden Register gleich mit:




true: Bus nicht freigeben, gleich weiterlesen. false: am Ende „Stopp“, Bus frei.

Im Pico SDK sind das zwei Funktionsaufrufe:

```
int i2c_write_blocking(i2c_inst_t *i2c, uint8_t adresse, const uint8_t *daten,
                       size_t anzahl, bool nostop);
int i2c_read_blocking(i2c_inst_t *i2c, uint8_t adresse, uint8_t *daten,
                      size_t anzahl, bool nostop);
```

- `i2c` ist der I2C-Baustein. Der RP2350 hat zwei, `i2c0` und `i2c1`. Der Sensor hängt an `i2c0`, über GP20 (SDA) und GP21 (SCL).
- Die Daten übergibst du als **Zeiger** (Kapitel 19) mit ihrer Anzahl.
- `nostop = true` heißt: den Bus am Ende nicht freigeben. Das braucht man zwischen „Registernummer schreiben“ und „lesen“.
- Der Rückgabewert ist die Zahl der übertragenen Bytes. Antwortet unter der Adresse kein Gerät, ist er negativ (`PICO_ERROR_GENERIC`, also −1).

Eingerichtet hat den Bus `pbc_init`, in der Funktion `sensor_init` in `pbc_geraet.c`: `i2c_init(i2c0, 400000)` startet den Baustein mit 400 000 Takten pro Sekunde, `gpio_set_function(…, GPIO_FUNC_I2C)` übergibt ihm die beiden Pins, und `gpio_pull_up` schaltet zusätzlich die Pull-ups im Chip ein.

## 22.2 Der I2C-Scan

Welche Geräte hängen eigentlich am Bus? Das findet ein **Scan** heraus: Er spricht jede mögliche Adresse an und merkt sich, wo jemand antwortet.

k22_scan.c

```
 1 #include "pbc.h"
 2 #include "hardware/i2c.h"
 3
 4 int main(void) {
 5     pbc_init();                  // richtet i2c0 an GP20 (SDA) und GP21 (SCL)
       ein
 6     pbc_clear(PBC_BLACK);
 7     pbc_text18("I2C-Scan", 75, 30, PBC_WHITE);
 8     printf("I2C-Scan an i2c0\n");
 9
10     int gefunden = 0;
11     for (int adresse = 0x08; adresse <= 0x77; adresse++) {
12         uint8_t daten;
13         // Antwortet ein Geraet unter dieser Adresse, kommt 1 Byte zurueck,
14         // sonst ein negativer Fehlercode.
15         int ergebnis = i2c_read_blocking(i2c0, adresse, &daten, 1, false);
16         if (ergebnis >= 0) {
17             char text[24];
18             snprintf(text, sizeof text, "0x%02X (%d)", adresse, adresse);
19             printf("Antwort von %s\n", text);
20             pbc_text18(text, 70, 80 + gefunden * 30, PBC_GREEN);
21             gefunden++;
22         }
23     }
24     printf("%d Geraet(e) gefunden\n", gefunden);
25     if (gefunden == 0) {
26         pbc_text18("nichts gefunden", 55, 80, PBC_RED);
27     }
28     pbc_show();
29
30     while (true) {
31         sleep_ms(1000);
32     }
33 }
```

- Die Schleife in Zeile 11 geht von 0x08 bis 0x77. Die Adressen darunter und darüber sind für besondere Zwecke reserviert.
- Zeile 15 versucht, ein Byte zu lesen. Unter einer Adresse ohne Gerät bestätigt niemand, und `i2c_read_blocking` gibt einen negativen Wert zurück.

- `%02X` in Zeile 18 gibt eine Zahl hexadezimal mit zwei Stellen und großen Buchstaben aus, `%d` daneben dieselbe Zahl dezimal.

Auf dem PicoBoy meldet sich der Neigungssensor unter `0x18 (24)`. Erscheint weiter nichts, ist er das einzige Gerät am Bus. Erscheint gar nichts, ist der Bus nicht richtig eingerichtet; in deinem Programm fehlt dann wahrscheinlich `pbc_init()`.

> Ein Scan ist der erste Schritt bei jedem neuen I2C-Gerät. Er beantwortet zwei Fragen auf einmal: Stimmt die Adresse? Und funktioniert die Verbindung überhaupt?

## 22.3 Rohwerte lesen und umrechnen

**Wer bist du?** Antwortet unter 0x18 tatsächlich ein STK8BA58? Das verrät Register 0x00:

k22_chip_id.c

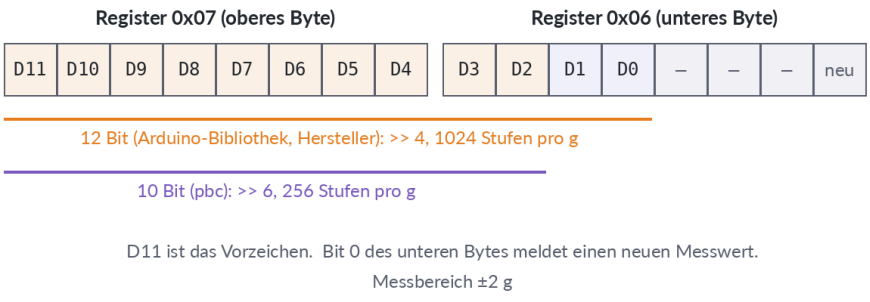
```c
#include "pbc.h"
#include "hardware/i2c.h"

#define SENSOR 0x18                // I2C-Adresse des Neigungssensors

// liest anzahl Bytes ab Register reg; Ergebnis < 0 bei einem Fehler
int lese_register(uint8_t reg, uint8_t *puffer, size_t anzahl) {
    // 1. Registernummer schreiben, ohne den Bus freizugeben (true)
    int r = i2c_write_blocking(i2c0, SENSOR, &reg, 1, true);
    if (r < 0) {
        return r;
    }
    // 2. Bytes lesen und den Bus freigeben (false)
    return i2c_read_blocking(i2c0, SENSOR, puffer, anzahl, false);
}
```

Die Funktion `lese_register` fasst die beiden Schritte aus 22.1 zusammen. Weil sie in `puffer` schreibt, bekommt sie einen Zeiger; für ein einzelnes Byte übergibt `main` die Adresse `&id`. Auf dem Display steht danach `Chip-ID: 0x87` und darunter `STK8BA58`.

**Die Messwerte.** Jede Achse belegt zwei Register. Der Sensor misst mit 12 Bit und legt die Zahl **linksbündig** in die beiden Bytes: die oberen 8 Bit in das obere Register, die unteren 4 Bit in die obere Hälfte des unteren Registers. Die übrigen 4 Bit gehören nicht zum Messwert; Bit 0 meldet, ob seit dem letzten Lesen ein neuer Wert gemessen wurde.



So setzt man die Zahl zusammen:

k22_rohwerte.c

```c
// Zwei Register zu einer Zahl: oberes Byte b[1], unteres Byte b[0].
// Die 12 Bit stehen linksbuendig; Bit 0 bis 3 von b[0] gehoeren nicht dazu.
int achse_12bit(const uint8_t *b) {
    int16_t roh = (int16_t)((b[1] << 8) | (b[0] & 0xF0));
    return roh >> 4;              // -2048 bis 2047
}
```

1. `b[1] << 8` schiebt das obere Byte um 8 Stellen nach links, `| b[0]` setzt das untere Byte daneben. `& 0xF0` lässt vom unteren Byte nur die oberen 4 Bit übrig.
2. `(int16_t)` macht daraus eine Zahl mit Vorzeichen. Das oberste Bit ist jetzt das Vorzeichen: Ist es gesetzt, ist die Zahl negativ. Die Beschleunigung kann ja in beide Richtungen zeigen.
3. `>> 4` schiebt die Zahl um 4 Stellen nach rechts, an den rechten Rand. Bei negativen Zahlen füllt der Compiler links mit Einsen auf; das Vorzeichen bleibt also erhalten. Das Ergebnis liegt zwischen −2048 und 2047.

Im Messbereich ±2 g stehen 4096 Stufen für 4 g, also **1024 Stufen pro g**. Teilt man den Rohwert durch 1024, hat man die Beschleunigung in g. Das vollständige Programm liest alle sechs Register mit einem einzigen Aufruf ab 0x02 und zeigt für jede Achse die beiden Bytes, den Rohwert und g:

k22_rohwerte.c

```c
int main(void) {
    pbc_init();
    const char *namen[3] = { "x", "y", "z" };
    while (true) {
        uint8_t b[6];             // ab Register 0x02: x unten, x oben, y unten
          ...
        pbc_clear(PBC_BLACK);
        pbc_text12("Chip-Achse  Bytes   12 Bit     g", 12, 40, PBC_GRAY);
        if (lese_register(0x02, b, 6) == 6) {
            for (int i = 0; i < 3; i++) {
                int wert = achse_12bit(&b[2 * i]);
                char text[48];
                snprintf(text, sizeof text, "%s   %02X %02X   %5d   %5.2f",
                         namen[i], b[2 * i + 1], b[2 * i], wert, wert / 1024.0f);
                pbc_text(text, 30, 80 + i * 30, PBC_WHITE);
            }
        }
        char text[48];
        snprintf(text, sizeof text, "pbc: %5.2f %5.2f %5.2f",
                 pbc_accel_x_g(), pbc_accel_y_g(), pbc_accel_z_g());
        pbc_text(text, 30, 200, PBC_YELLOW);
        pbc_show();
        sleep_ms(200);
    }
}
```

Liegt der PicoBoy flach auf dem Tisch, zeigt die z-Achse etwa 1024, also 1 g, die beiden anderen etwa 0. Kippst du ihn, wandert der Wert zwischen den Achsen. Kleine Abweichungen und ein Zittern in den letzten Stellen sind normal: Jeder Sensor hat eine kleine Ungenauigkeit und ein wenig Rauschen.

**Zwei Wege, ein Ergebnis.** In der Bibliothek steht es anders:

pbc_geraet.c

```
368 // Eine Achse: Der Chip liefert 12 Bit, linksbündig in zwei Registern (1024
369 // Stufen pro g bei ±2 g). Hier werden nur die oberen 10 Bit benutzt
370 // (256 Stufen pro g) – für Spiele fein genug; Kapitel 22 zeigt alle 12.
371 static float achse_g(uint8_t reg) {
372     uint8_t b[2];
373     if (!sensor_da || sensor_lesen(reg, b, 2) < 0) return 0.0f;
374     int16_t roh = (int16_t)((b[1] << 8) | b[0]);
375     return (float)(roh >> 6) * 2.0f / 512.0f;
376 }
```

Die Bibliothek nimmt nur die oberen 10 Bit: Sie schiebt um 6 statt um 4 Stellen und rechnet `* 2.0f / 512.0f`, teilt also durch 256 statt durch 1024. Die Arduino-Bibliothek für den Sensor und das Arduino-Beispiel aus dem PicoBoy-Projekt nehmen dagegen 12 Bit und teilen durch 1024. Wer hat recht?

Beide. Die Zahl steht linksbündig in den Registern. Wie viele Bits man von links nimmt, entscheidet man selbst; man muss nur passend teilen:

| genommen | Schieben | Bereich | Stufen pro g |
|---|---|---|---|
| 12 Bit | `>> 4` | −2048 … 2047 | 1024 |
| 10 Bit | `>> 6` | −512 … 511 | 256 |

Zwei Stellen weiter nach rechts schieben heißt durch 4 teilen. Der Rohwert wird viermal kleiner, die Stufen pro g auch. Das Ergebnis in g ist dasselbe; die Bibliothek verzichtet nur auf die beiden feinsten Bits. Ihre kleinste Stufe ist 1/256 g statt 1/1024 g. Für Spiele und eine Wasserwaage spielt das keine Rolle. Das Programm `k22_zwei_wege.c` rechnet für die z-Achse beides nebeneinander und schreibt es in die Konsole:

k22_zwei_wege.c

```
13         int16_t roh = (int16_t)((b[1] << 8) | b[0]);
14
15         int wert12 = (roh & ~0x000F) >> 4;     // 12 Bit: -2048 bis 2047
16         int wert10 = roh >> 6;                 // 10 Bit: -512 bis 511
17
18         printf("Bytes %02X %02X | 12 Bit: %5d -> %.3f g | 10 Bit: %4d -> %.3f
             g\n",
19                b[1], b[0], wert12, wert12 / 1024.0f, wert10, wert10 / 256.0f);
```

`roh & ~0x000F` löscht die unteren 4 Bit (`~` dreht alle Bits um, aus `0x000F` wird eine Maske, in der nur diese 4 Bit null sind). Beim Weg über 10 Bit fallen sie beim Schieben ohnehin heraus. In jeder Zeile stehen zwei Werte in g, die höchstens um 3/1024 g auseinanderliegen: Mehr als die zwei verworfenen Bits kann der Unterschied nicht ausmachen.

**Die Achsen des PicoBoy.** Der Chip ist gedreht auf die Platine gelötet. Seine x-Achse zeigt deshalb dorthin, wo beim PicoBoy y ist, und umgekehrt, und zwei Achsen zählen in die andere Richtung. Die Bibliothek gleicht das aus:

pbc_geraet.c

```
378 // Der Chip liegt gedreht auf der Platine: x und y vertauscht, Vorzeichen anders.
379 float pbc_accel_x_g(void) { return achse_g(0x04); }
380 float pbc_accel_y_g(void) { return -achse_g(0x02); }
381 float pbc_accel_z_g(void) { return -achse_g(0x06); }
```

`pbc_accel_x_g` liest also Register 0x04, die y-Achse des Chips; `pbc_accel_y_g` liest Register 0x02, die x-Achse, mit umgedrehtem Vorzeichen, und `pbc_accel_z_g` dreht auch z um. Die gelbe Zeile von `k22_rohwerte.c` zeigt die Werte der Bibliothek. Vergleiche: Der x-Wert der Bibliothek ist der y-Wert des Chips, der y-Wert der Bibliothek der negative x-Wert und der z-Wert der negative z-Wert des Chips. Flach auf dem Tisch zeigt der Chip also z ≈ −1 g, die Bibliothek z ≈ +1 g.

**Einstellen.** `sensor_init` schreibt drei Register: Messbereich ±2 g, Bandbreite 125 Hz, normaler Betrieb. Zum Schreiben schickt man Registernummer und Wert in einem Zug, ohne anschließend zu lesen:

```
uint8_t befehl[2] = { 0x0F, 0x03 };   // Register 0x0F bekommt 0x03
i2c_write_blocking(i2c0, 0x18, befehl, 2, false);
```

> Die Bibliothek rechnet fest mit ±2 g. Stellst du einen anderen Messbereich ein (Aufgabe 3), liefern die `pbc_accel_…`-Funktionen danach falsche Werte, bis zum nächsten Neustart.

## Ausprobieren: Die Libelle

Eine Wasserwaage hat eine Libelle: eine Luftblase in einem Glas, die immer zur höchsten Stelle wandert. Diese hier liest den Sensor selbst, ganz ohne `pbc_accel_…`:

k22_libelle.c

```
 1 #include "pbc.h"
 2 #include "hardware/i2c.h"
 3
 4 #define SENSOR 0x18
 5
 6 // liest die drei Achsen des Chips in g; false bei einem I2C-Fehler
 7 bool lese_chip(float *x, float *y, float *z) {
 8     uint8_t reg = 0x02;
 9     uint8_t b[6];
```

```
10      if (i2c_write_blocking(i2c0, SENSOR, &reg, 1, true) < 0) {
11          return false;
12      }
13      if (i2c_read_blocking(i2c0, SENSOR, b, 6, false) != 6) {
14          return false;
15      }
16      float g[3];
17      for (int i = 0; i < 3; i++) {
18          int16_t roh = (int16_t)((b[2 * i + 1] << 8) | (b[2 * i] & 0xF0));
19          g[i] = (roh >> 4) / 1024.0f;
20      }
21      // Der Chip liegt gedreht auf der Platine (wie in pbc_geraet.c).
22      *x = g[1];
23      *y = -g[0];
24      *z = -g[2];
25      return true;
26  }
27
28  int main(void) {
29      pbc_init();
30      float x = 0, y = 0, z = 0;
31      while (true) {
32          if (!lese_chip(&x, &y, &z)) {
33              pbc_led_red(true);                  // Sensor antwortet nicht
34          }
35          // Die Blase wandert zur hoeheren Seite. Liegt rechts tiefer,
36          // zeigt "oben" schraeg nach links: x wird negativ.
37          int bx = 120 + (int)(x * 90);
38          int by = 140 + (int)(y * 90);
39          pbc_clear(pbc_rgb(20, 60, 20));
40          pbc_circle(120, 140, 95, PBC_WHITE);
41          pbc_circle(120, 140, 16, PBC_WHITE);
42          pbc_fill_circle(bx, by, 12, PBC_YELLOW);
43          pbc_show();
44          sleep_ms(20);
45      }
46  }
```

- `lese_chip` liest alle drei Achsen und liefert sie über drei Zeiger, dazu `true` oder `false`, je nachdem, ob das Lesen geklappt hat (Kapitel 19.2).
- Die Zeilen 22 bis 24 drehen die Achsen genau wie die Bibliothek.
- Ein Beschleunigungssensor in Ruhe misst nicht die Schwerkraft nach unten, sondern die Gegenkraft nach **oben**: flach liegend z = +1 g. Liegt die rechte Seite tiefer, zeigt „oben“ für den PicoBoy schräg nach links, und x wird negativ. Die Blase soll nach links, zur höheren Seite: Zeile 37 addiert deshalb `x * 90`.

Hältst du den PicoBoy waagrecht, liegt die gelbe Blase im kleinen Kreis in der Mitte.

**Zum Verändern:** Lass die grüne LED leuchten, solange die Blase im kleinen Kreis liegt. Glätte die Bewegung wie in Kapitel 17.2.

## Übungsaufgaben

**Aufgabe 1** ⭐ **Zahlen umrechnen** Schreibe die Adresse 0x18 und die Chip-ID 0x87 dezimal und als Bitmuster mit 8 Stellen. Welche Zahl ergibt der Rohwert `0x0400` nach `>> 4`, und wie viel g sind das?

**Aufgabe 2** ⭐ **Den Messbereich nachlesen** Lies Register 0x0F und gib seinen Inhalt hexadezimal in der Konsole aus. Welcher Wert müsste nach `pbc_init()` darin stehen?

**Aufgabe 3** ⭐⭐ **±8 g** Stelle den Messbereich auf ±8 g und gib die z-Achse in g aus. Wie viele Stufen pro g sind es jetzt mit 12 Bit? Wozu könnte ein größerer Messbereich gut sein?

**Aufgabe 4** ⭐⭐ **Schüttelzähler** Berechne aus den drei Achsen die Stärke der Beschleunigung, $\sqrt{(x^2 + y^2 + z^2)}$ (mit `sqrtf` aus `math.h`). In Ruhe ist sie etwa 1 g. Zähle, wie oft sie über 1,8 g steigt, und zeige die Zahl an. Achte darauf, dass ein einziges Schütteln nicht mehrfach zählt.

**Aufgabe 5** ⭐⭐⭐ **Am Stück oder einzeln?** Lies die sechs Datenregister 100-mal am Stück (ein Aufruf mit 6 Bytes) und 100-mal einzeln (sechs Aufrufe mit je 1 Byte). Miss mit `time_us_32()`, wie lange eine Messung im Mittel jeweils dauert. Erkläre den Unterschied mit dem Bild aus 22.1.

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- **I2C**: zwei Leitungen, SDA (Daten) und SCL (Takt), mit Pull-ups. Ein Controller (der RP2350) spricht Geräte über ihre **7-Bit-Adresse** an. Der Neigungssensor STK8BA58 hängt an `i2c0` (GP20/GP21) unter **0x18**.
- Ein Gerät hat **Register**. Lesen: Registernummer schreiben mit `i2c_write_blocking(…, true)`, dann `i2c_read_blocking(…, false)`. Schreiben: Nummer und Wert in einem Aufruf. Negativer Rückgabewert: niemand hat geantwortet.
- **Scan**: jede Adresse von 0x08 bis 0x77 ansprechen und sehen, wer antwortet.
- Register 0x00 enthält die Chip-ID 0x87. Die Messwerte stehen in 0x02 bis 0x07, je Achse zwei Bytes, **12 Bit linksbündig**.
- Zusammensetzen: `(int16_t)((b[1] << 8) | (b[0] & 0xF0)) >> 4`, dann durch 1024 teilen (±2 g). Die Bibliothek nimmt nur 10 Bit (`>> 6`, durch 256): gleiches Ergebnis, etwas gröber.
- Auf dem PicoBoy liegt der Chip gedreht: x des PicoBoy = −y des Chips, y des PicoBoy = x des Chips.

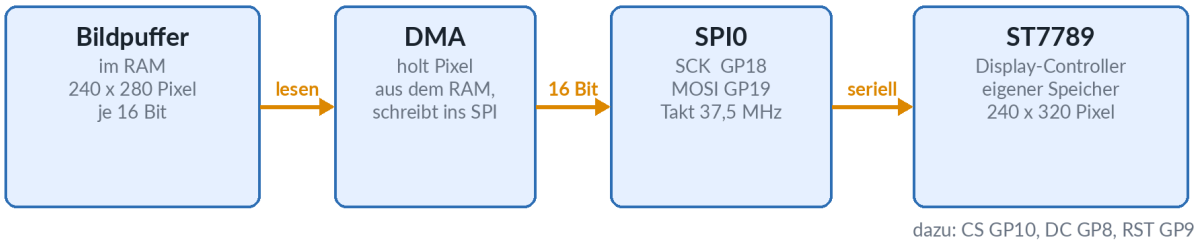
KAPITEL 23

# SPI, DMA und das Display

**Worum geht's?** Seit Kapitel 3 rufst du `pbc_show()` auf, und das Bild erscheint. Jetzt siehst du, was dabei auf den Leitungen passiert: wie der Display-Controller ST7789 Befehle und Pixel über den **SPI-Bus** bekommt, wie man ein Fenster auf dem Display setzt und direkt hineinzeichnet, und wie der **DMA** das ganze Bild verschickt, während der Prozessor etwas anderes tut. Danach kennst du jede Zeile des Display-Teils von `pbc_geraet.c`.

## 23.1 Wie der ST7789 Befehle und Pixel bekommt

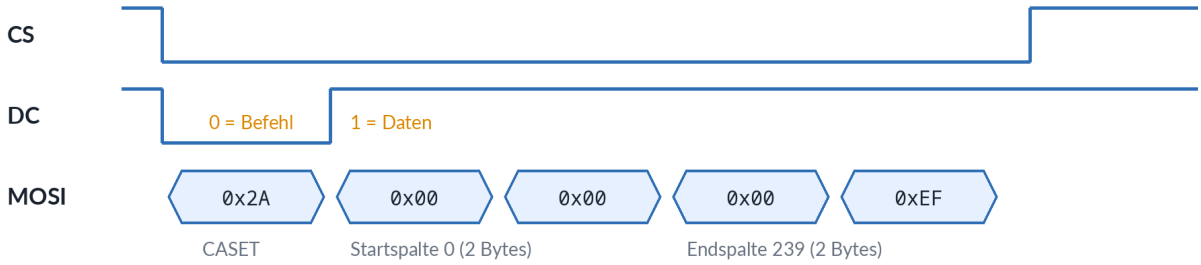
Das Display des PicoBoy hat einen eigenen kleinen Chip, den **Display-Controller** ST7789. Er hat einen eigenen Speicher für die Farben aller Pixel und frischt das Bild von selbst auf, viele Male pro Sekunde. Der RP2350 muss ihm nur sagen, was sich ändern soll. Dafür sind die beiden über den **SPI-Bus** verbunden (*Serial Peripheral Interface*). Wie beim I2C in Kapitel 22 gehen die Bits nacheinander über eine Leitung, aber einfacher und viel schneller:




| Leitung | Pin | Aufgabe |
|---|---|---|
| SCK | GP18 | Takt: bei jedem Takt geht ein Bit hinüber |
| MOSI | GP19 | die Daten (*Master Out, Slave In*), vom RP2350 zum Display |
| CS | GP10 | *Chip Select*: LOW heißt „du bist gemeint“ |
| DC | GP8 | *Data/Command*: LOW = Befehl, HIGH = Daten |
| RST | GP9 | Reset: LOW startet den Controller neu |
| Licht | GP26 | Hintergrundbeleuchtung, per PWM (Kapitel 21.3) |

Eine Leitung zurück vom Display gibt es nicht: Der PicoBoy schreibt nur, er liest nie etwas vom Display. Die Leitung CS ist ein gewöhnlicher Pin, den das Programm mit `gpio_put` selbst setzt (Kapitel 21.1), ebenso DC.

**Befehle und Daten.** Alles, was der ST7789 bekommt, sind Bytes. Ob ein Byte ein **Befehl** ist oder **Daten** zu einem Befehl, entscheidet die Leitung DC. So sieht der Befehl CASET mit seinen vier Datenbytes auf den Leitungen aus:




In `pbc_geraet.c` macht das die Funktion `display_befehl` (Zeilen 54 bis 64): Format auf 8 Bit stellen, CS auf LOW, DC auf LOW, das Befehlsbyte schicken; gibt es Daten, DC auf HIGH und die Daten hinterher; zum Schluss CS wieder auf HIGH. `spi_write_blocking` aus `hardware/spi.h` schickt eine Reihe von Bytes und kehrt erst zurück, wenn das letzte Bit draußen ist.

Die Befehle, die die Bibliothek benutzt, stehen in `display_init` (Zeilen 76 bis 116):

| Befehl | Name | Bedeutung |
|---|---|---|
| `0x01` | SWRESET | Neustart des Controllers |
| `0x11` | SLPOUT | aus dem Schlafmodus aufwachen (danach 120 ms warten) |
| `0x3A` | COLMOD | Farbformat; Datenbyte `0x55` = 16 Bit pro Pixel (RGB565) |
| `0x36` | MADCTL | Ausrichtung; `0xC0` = um 180° gedreht |
| `0x21` | INVON | Farben umkehren; dieses Panel braucht das |
| `0x13` | NORON | normaler Anzeigemodus |
| `0x29` | DISPON | Anzeige einschalten |
| `0x2A` | CASET | Spaltenbereich festlegen (4 Datenbytes) |
| `0x2B` | RASET | Zeilenbereich festlegen (4 Datenbytes) |
| `0x2C` | RAMWR | ab jetzt kommen Pixel |

**Eigene Befehle schicken.** Nach `pbc_init()` ist der SPI-Bus eingerichtet. Dein Programm kann also selbst Befehle schicken. Dieses Programm schaltet mit A die Umkehrung der Farben aus und wieder ein und mit B die Anzeige für eine Sekunde ab:

k23_befehle.c

```
// Eigene Befehle an den ST7789: Farben umkehren, Anzeige aus und an
#include "pbc.h"
#include "hardware/spi.h"

#define PIN_DISPLAY_CS 10
#define PIN_DISPLAY_DC 8
```

```
 8 #define BEFEHL_INVOFF  0x20   // Farben nicht umkehren
 9 #define BEFEHL_INVON   0x21   // Farben umkehren (so braucht es dieses Panel)
10 #define BEFEHL_DISPOFF 0x28   // Anzeige aus
11 #define BEFEHL_DISPON  0x29   // Anzeige an
12
13 static void befehl(uint8_t code) {
14     spi_set_format(spi0, 8, SPI_CPOL_0, SPI_CPHA_0, SPI_MSB_FIRST);
15     gpio_put(PIN_DISPLAY_CS, 0);   // Display ansprechen
16     gpio_put(PIN_DISPLAY_DC, 0);   // DC = 0: jetzt kommt ein Befehl
17     spi_write_blocking(spi0, &code, 1);
18     gpio_put(PIN_DISPLAY_CS, 1);   // fertig
19 }
20
21 int main(void) {
22     pbc_init();
23     pbc_clear(PBC_WHITE);
24     pbc_fill_rect(30, 60, 50, 50, PBC_RED);
25     pbc_fill_rect(95, 60, 50, 50, PBC_GREEN);
26     pbc_fill_rect(160, 60, 50, 50, PBC_BLUE);
27     pbc_text18("A: umkehren", 50, 150, PBC_BLACK);
28     pbc_text18("B: aus und an", 45, 185, PBC_BLACK);
29     pbc_show();
30
31     bool normal = true;
32     while (true) {
33         if (pbc_was_pressed_a()) {
34             normal = !normal;
35             befehl(normal ? BEFEHL_INVON : BEFEHL_INVOFF);
36         }
37         if (pbc_was_pressed_b()) {
38             befehl(BEFEHL_DISPOFF);
39             sleep_ms(1000);
40             befehl(BEFEHL_DISPON);
41         }
42         sleep_ms(20);
43     }
44 }
```

- Die Funktion `befehl` in den Zeilen 13 bis 19 ist eine kleine Fassung von `display_befehl`, für Befehle ohne Daten.
- Zeile 14 stellt das SPI-Format jedes Mal auf 8 Bit. Das ist nötig, weil `pbc_show()` es auf 16 Bit umstellt (Abschnitt 23.3).
- Drückst du A, erscheint das Bild mit vertauschten Farben: Weiß wird schwarz, Rot wird türkis. Der Bildpuffer ändert sich dabei nicht. Nur der Controller zeigt dieselben Zahlen anders an.
- `BEFEHL_DISPOFF` schaltet die Anzeige ab, das Hintergrundlicht bleibt an. Der Bildschirm wird dunkel, aber nicht ganz schwarz.

> Die Bedeutung aller Befehle steht im Datenblatt des ST7789. Für den PicoBoy gelten zwei Besonderheiten: Das Panel braucht **INVON**, sonst sind alle Farben umgekehrt, und die Bibliothek schickt Farben im normalen RGB565 (das BGR-Bit in MADCTL bleibt aus). Wer in anderen Quellen `MADCTL 0xC8` liest: Dort sind Rot und Blau in der Software vertauscht.

## 23.2 Ein Fenster setzen und füllen

Der ST7789 hat Speicher für 240 × 320 Pixel. Sichtbar sind beim PicoBoy nur die Zeilen 20 bis 299 des Controllers: Die Zeile 0 des Bildschirms ist für den Controller die Zeile 20. Diese Verschiebung heißt in der Bibliothek `DISPLAY_Y_VERSATZ`.

Um Pixel zu schreiben, legt man zuerst ein rechteckiges **Fenster** fest: CASET mit erster und letzter Spalte, RASET mit erster und letzter Zeile, je als zwei Bytes, das höhere zuerst. Nach RAMWR füllt jedes folgende Pixel das Fenster von links oben nach rechts unten, Zeile für Zeile. Ist das Fenster voll, ist es gleich, wie viele Pixel noch kommen; man schickt genau so viele, wie hineinpassen.

Dieses Programm zeichnet ohne Bildpuffer und ohne `pbc_show()` direkt aufs Display: alle 100 ms ein buntes Quadrat an eine zufällige Stelle.

k23_fenster.c

```c
static void fenster(int x0, int y0, int x1, int y1) {
    y0 += Y_VERSATZ;
    y1 += Y_VERSATZ;
    uint8_t spalten[4] = { x0 >> 8, x0 & 0xFF, x1 >> 8, x1 & 0xFF };
    uint8_t zeilen[4] = { y0 >> 8, y0 & 0xFF, y1 >> 8, y1 & 0xFF };
    befehl(0x2A, spalten, 4);   // CASET: von Spalte x0 bis x1
    befehl(0x2B, zeilen, 4);    // RASET: von Zeile y0 bis y1
    befehl(0x2C, NULL, 0);      // RAMWR: jetzt kommen die Pixel
}

static void fuellen(int x, int y, int breite, int hoehe, uint16_t farbe) {
    uint16_t zeile[PBC_WIDTH];
    for (int i = 0; i < breite; i++) {
        zeile[i] = farbe;
    }
    fenster(x, y, x + breite - 1, y + hoehe - 1);
    spi_set_format(spi0, 16, SPI_CPOL_0, SPI_CPHA_0, SPI_MSB_FIRST);
    gpio_put(PIN_DISPLAY_CS, 0);
    gpio_put(PIN_DISPLAY_DC, 1);   // DC = 1: Daten
    for (int j = 0; j < hoehe; j++) {
        spi_write16_blocking(spi0, zeile, breite);
    }
    gpio_put(PIN_DISPLAY_CS, 1);
}

int main(void) {
    pbc_init();
    while (true) {
        int x = pbc_random(0, 5) * 40;
        int y = pbc_random(0, 6) * 40;
        uint16_t farbe = pbc_rgb(pbc_random(0, 255), pbc_random(0, 255),
                                 pbc_random(0, 255));
        fuellen(x, y, 40, 40, farbe);
        sleep_ms(100);
    }
}
```

Die Zeilen 1 bis 19 sind nicht abgedruckt: Sie enthalten die Pin-Namen, `Y_VERSATZ` und die Funktion `befehl`, diesmal mit Daten. Sie ist genau `display_befehl` aus der Bibliothek.

- `fenster` (Zeilen 21 bis 29) entspricht `display_fenster` der Bibliothek. `x0 >> 8` ist das höhere Byte einer Zahl, `x0 & 0xFF` das niedrigere. Für Spalte 239 sind das `0x00` und `0xEF`.
- `fuellen` (Zeilen 31 bis 44) setzt das Fenster, stellt den SPI-Bus auf **16 Bit** und schickt dann so viele Zeilen gleicher Farbe, wie das Rechteck hoch ist. Mit 16 Bit geht jedes Pixel mit dem höheren Byte zuerst hinaus, so wie es der ST7789 erwartet. Mit 8 Bit müsstest du die beiden Bytes jeder Farbe selbst in die richtige Reihenfolge bringen.
- Das Array `zeile` in Zeile 32 hat 240 Einträge zu 2 Bytes, also 480 Bytes auf dem Stack. Das ist für die 4 KB noch in Ordnung (Kapitel 19.4).

Das Bild bleibt stehen, bis jemand die Stelle überschreibt: Der Controller merkt sich jedes Pixel selbst. Ruft dein Programm danach `pbc_show()` auf, ist alles weg, denn dann kommt der ganze Bildpuffer und überschreibt den Speicher des Controllers.

> Ein Pixel kommt so aufs Display: **CASET** und **RASET** legen ein Fenster fest, **RAMWR** startet, dann folgen die Farben, je 16 Bit, Zeile für Zeile. Beim PicoBoy kommt zu jeder Zeile **20** hinzu.

**Warum dann überhaupt ein Bildpuffer?** Direkt zeichnen spart 134 KB RAM. Aber jede Änderung braucht eigene Befehle, und wer erst löscht und dann neu zeichnet, sieht es flackern: Der Controller zeigt den gelöschten Zustand, bevor das Neue ankommt. Mit dem Bildpuffer entsteht das Bild unsichtbar im RAM und wird in einem Stück übertragen (Kapitel 14).

## 23.3 DMA: Daten schieben, während die CPU weiterrechnet

**Wie lange dauert ein Bild?** Ein Bild hat 240 × 280 = 67 200 Pixel zu 16 Bit, also 1 075 200 Bits. Wie schnell der SPI-Bus taktet, verrät `spi_get_baudrate`:

k23_spi_takt.c

```
 1 // Wie schnell ist der SPI-Bus, und wie lange dauert ein ganzes Bild?
 2 #include "pbc.h"
 3 #include "hardware/spi.h"
 4
 5 int main(void) {
 6     pbc_init();
 7     pbc_clear(PBC_BLUE);
 8     pbc_text18("SPI-Messung", 55, 120, PBC_WHITE);
 9
10     uint32_t start = time_us_32();
11     pbc_show();
12     uint32_t dauer = time_us_32() - start;
13
14     uint takt = spi_get_baudrate(spi0);
15     int bits = PBC_WIDTH * PBC_HEIGHT * 16;
16     float gerechnet_ms = bits * 1000.0f / takt;
17
18     printf("SPI-Takt:     %u Hz\n", takt);
19     printf("Ein Bild:     %d Bits\n", bits);
```

```
20      printf("Gerechnet:    %.1f ms\n", gerechnet_ms);
21      printf("Gemessen:     %.1f ms\n", dauer / 1000.0f);
22
23      while (true) {
24          sleep_ms(1000);
25      }
26  }
```

Die Bibliothek bestellt in `display_init` 62,5 MHz (Zeile 81). Der SPI-Teil des RP2350 kann aber nur seinen Eingangstakt von 150 MHz durch bestimmte ganze Zahlen teilen. Das SDK sucht den schnellsten Takt, der **nicht über** dem gewünschten liegt; nach seinem Quelltext (`spi_set_baudrate`) ist das 150 MHz / 4 = **37,5 MHz**. Dann dauert ein Bild 1 075 200 / 37 500 000 s, also rund **29 ms**, ohne die paar Befehle davor. Das Programm gibt beide Werte aus, den gerechneten und den gemessenen. Auf dem PicoBoy misst es 31,4 ms, etwas mehr als gerechnet: Die Rechnung zählt nur die reinen Datenbits. Vergleiche mit der Messung aus Kapitel 14.

> Manchmal liest man, ein Bild dauere etwa 17 ms. Das passt zu den **bestellten** 62,5 MHz (1 075 200 / 62 500 000 s = 17,2 ms). Der Bus läuft aber mit 37,5 MHz, ein Bild braucht also rund 29 ms (gemessen 31,4 ms). Merke dir daraus: `spi_init` liefert den Takt, der wirklich eingestellt ist, und Zeiten misst man, statt sie abzuschreiben.

**DMA.** In diesen rund 30 ms müssten 67 200 Werte einer nach dem anderen in das Datenregister des SPI-Teils geschrieben werden, jeder erst dann, wenn dort wieder Platz ist. Das kann die CPU mit `spi_write16_blocking` erledigen, wie in `fuellen`. Dann wartet sie die ganze Zeit. Der RP2350 hat dafür einen Helfer: den **DMA** (*Direct Memory Access*, direkter Speicherzugriff). Ein DMA-Kanal kopiert Daten von einer Adresse zu einer anderen, ohne dass die CPU dabei etwas tun muss. Der RP2350 hat 16 solcher Kanäle.

Einen Kanal stellt man so ein:

| Einstellung | Funktion | in `pbc_show()` |
|---|---|---|
| Größe eines Werts | `channel_config_set_transfer_data_size` | 16 Bit (`DMA_SIZE_16`) |
| Quelle wandert weiter? | `channel_config_set_read_increment` | ja: Pixel für Pixel |
| Ziel wandert weiter? | `channel_config_set_write_increment` | nein: immer dasselbe SPI-Register |
| Wer gibt das Tempo vor? | `channel_config_set_dreq` | der SPI-Teil: „ich habe Platz“ |
| Ziel, Quelle, Anzahl, Start | `dma_channel_configure` | SPI-Datenregister, Bildpuffer, 67 200, sofort |

Die Angabe **DREQ** (*Data Request*) ist der Trick: Der SPI-Teil meldet dem DMA, wann er den nächsten Wert aufnehmen kann. So läuft der DMA genau im Takt des Busses und verliert keinen Wert.

`pbc_show()` (Zeilen 123 bis 138 in `pbc_geraet.c`) startet den DMA und wartet dann mit `dma_channel_wait_for_finish_blocking`, bis er fertig ist. Das ist einfach und sicher. Aber es verschenkt die Zeit. Dieses Programm hat ein eigenes `pbc_show()` in zwei Teilen: `show_starten` setzt das Fenster und startet den DMA, `show_beenden` wartet auf das Ende. Dazwischen zählt die CPU, wie oft sie eine Schleife durchlaufen kann:

k23_dma.c

```
35 static void show_starten(void) {
36     fenster(0, 0, PBC_WIDTH - 1, PBC_HEIGHT - 1);
37     spi_set_format(spi0, 16, SPI_CPOL_0, SPI_CPHA_0, SPI_MSB_FIRST);
38     gpio_put(PIN_DISPLAY_CS, 0);
39     gpio_put(PIN_DISPLAY_DC, 1);
40     dma_channel_config c = dma_channel_get_default_config(kanal);
41     channel_config_set_transfer_data_size(&c, DMA_SIZE_16);   // je 16 Bit
42     channel_config_set_read_increment(&c, true);              // Quelle wandert
43     channel_config_set_write_increment(&c, false);            // Ziel bleibt
44     channel_config_set_dreq(&c, spi_get_dreq(spi0, true));    // Takt vom SPI
45     dma_channel_configure(kanal, &c, &spi_get_hw(spi0)->dr, pbc_bildpuffer,
46                           PBC_WIDTH * PBC_HEIGHT, true);   // sofort los
47 }
48
49 static void show_beenden(void) {
50     dma_channel_wait_for_finish_blocking(kanal);
51     while (spi_is_busy(spi0)) {
52         tight_loop_contents();   // letzte Bits hinausschieben lassen
53     }
54     gpio_put(PIN_DISPLAY_CS, 1);
55 }
56
57 int main(void) {
58     pbc_init();
59     kanal = dma_claim_unused_channel(true);
60     int runden = 0;
61     while (true) {
62         char text[30];
63         snprintf(text, sizeof text, "%d Runden", runden);
64         pbc_clear(PBC_BLACK);
65         pbc_text18("Nebenher gezaehlt:", 25, 110, PBC_WHITE);
66         pbc_text18(text, 25, 140, PBC_YELLOW);
67
68         uint32_t start = time_us_32();
69         show_starten();
70         runden = 0;
71         while (dma_channel_is_busy(kanal)) {
72             runden++;   // die CPU ist frei, solange der DMA arbeitet
73         }
```

```
74          show_beenden();
75          uint32_t dauer = time_us_32() - start;
76          printf("%d Runden in %d us\n", runden, (int)dauer);
77          sleep_ms(1000);
78      }
79  }
```

- Die Funktionen `befehl` und `fenster` (Zeilen 13 bis 33) sind dieselben wie in `k23_fenster.c`.
- `#include "pbc_intern.h"` gibt Zugriff auf `pbc_bildpuffer`, den Bildpuffer der Bibliothek. Normale Programme brauchen diese Datei nicht.
- `dma_claim_unused_channel(true)` in Zeile 59 sucht einen freien Kanal. Den ersten hat sich schon die Bibliothek genommen. `true` heißt: Gibt es keinen mehr, hält das Programm mit einer Fehlermeldung an.
- Zeile 45: `&spi_get_hw(spi0)->dr` ist die Adresse des Datenregisters von SPI0. Was dort hineingeschrieben wird, geht auf die Leitung.
- In den Zeilen 71 bis 73 fragt die CPU nur, ob der DMA noch arbeitet, und zählt. Statt zu zählen, könnte sie hier rechnen, Töne erzeugen oder Tasten auswerten.
- `show_beenden` wartet nach dem DMA noch, bis der SPI-Teil auch das letzte Bit hinausgeschoben hat (`spi_is_busy`), und setzt erst dann CS auf HIGH.

> Solange der DMA den Bildpuffer liest, darf das Programm **nicht hinein zeichnen**. Sonst landet ein halb altes, halb neues Bild auf dem Display. Wer während der Übertragung schon das nächste Bild zeichnen will, braucht zwei Bildpuffer und wechselt zwischen ihnen ab. Dafür reicht beim PicoBoy das RAM, aber es kostet weitere 134 KB.

**DMA ohne Display.** Ein DMA-Kanal kann auch einfach Speicher kopieren. Ohne DREQ läuft er dann so schnell, wie der Speicher es erlaubt:

k23_dma_kopie.c

```c
// Speicher kopieren: mit einer Schleife und mit dem DMA
#include "pbc.h"
#include "hardware/dma.h"

#define ANZAHL 10000

static uint32_t quelle[ANZAHL];
static uint32_t ziel[ANZAHL];

int main(void) {
    pbc_init();
    for (int i = 0; i < ANZAHL; i++) {
        quelle[i] = i * 3;
    }

    // 1. Mit einer Schleife: die CPU kopiert Wert für Wert
    uint32_t start = time_us_32();
```

```
18      for (int i = 0; i < ANZAHL; i++) {
19          ziel[i] = quelle[i];
20      }
21      printf("Schleife: %d us\n", (int)(time_us_32() - start));
22
23      // 2. Mit dem DMA
24      for (int i = 0; i < ANZAHL; i++) {
25          ziel[i] = 0;   // Ziel leeren, damit wir sehen, ob der DMA kopiert
26      }
27      int kanal = dma_claim_unused_channel(true);
28      dma_channel_config c = dma_channel_get_default_config(kanal);
29      channel_config_set_transfer_data_size(&c, DMA_SIZE_32);
30      channel_config_set_read_increment(&c, true);
31      channel_config_set_write_increment(&c, true);   // auch das Ziel wandert
32      start = time_us_32();
33      dma_channel_configure(kanal, &c, ziel, quelle, ANZAHL, true);
34      dma_channel_wait_for_finish_blocking(kanal);
35      printf("DMA:      %d us\n", (int)(time_us_32() - start));
36
37      bool gleich = true;
38      for (int i = 0; i < ANZAHL; i++) {
39          if (ziel[i] != quelle[i]) {
40              gleich = false;
41          }
42      }
43      printf("Kopie richtig: %s\n", gleich ? "ja" : "nein");
44      while (true) {
45          sleep_ms(1000);
46      }
47  }
```

Hier wandern Quelle und Ziel weiter (Zeilen 30 und 31). Die Standardeinstellung von `dma_channel_get_default_config` lässt das Ziel stehen; deshalb muss Zeile 31 das ausdrücklich einschalten. Welcher Weg auf deinem PicoBoy schneller ist, zeigt die Ausgabe. Der eigentliche Gewinn des DMA ist aber nicht die Geschwindigkeit, sondern dass die CPU während des Kopierens frei ist.

## Ausprobieren: Das Display kennenlernen

- Spiele `k23_befehle.c` auf und drücke A. Welche Farbe bekommt Rot, welche Grün, welche Blau? Prüfe es mit RGB565: Die Umkehrung macht aus jedem Bit sein Gegenteil, aus `0xF800` also `0x07FF`.
- Ändere in `k23_fenster.c` die Größe der Quadrate auf 20 Pixel. Was musst du an den Zeilen 49 und 50 noch ändern?
- Ersetze in `k23_dma.c` die Zählschleife durch `sleep_ms(10)` und miss die Dauer. Was passiert, wenn die Pause länger dauert als die Übertragung?

## Übungsaufgaben

**Aufgabe 1** ⭐ **Rechnen mit dem Bus** Wie viele Bytes hat ein ganzes Bild? Wie lange dauert seine Übertragung bei 37,5 MHz und wie lange bei 75 MHz? Wie viele Bilder pro Sekunde wären jeweils höchstens möglich?

**Aufgabe 2** ⭐⭐ **Ein Farbverlauf ohne Bildpuffer** Fülle mit `fuellen` aus `k23_fenster.c` jede Zeile des Displays mit einer eigenen Farbe, sodass oben Blau steht und unten Rot und dazwischen ein gleichmäßiger Übergang.

**Aufgabe 3** ⭐⭐ **Display schlafen legen** Der ST7789 kennt den Befehl SLPIN (`0x10`), der ihn in einen sparsamen Schlafmodus schickt, und SLPOUT (`0x11`), der ihn weckt. Schreibe ein Programm: B schaltet Hintergrundlicht und Anzeige ab und legt den Controller schlafen, A weckt alles wieder. Nach SLPOUT braucht der Controller 120 ms, bevor er neue Befehle annimmt.

**Aufgabe 4** ⭐⭐⭐ **`pbc_clear` per DMA** Ein DMA-Kanal kann auch immer **denselben** Wert lesen und ihn in viele aufeinanderfolgende Speicherstellen schreiben. Schreibe damit eine Funktion `dma_clear(farbe)`, die den Bildpuffer füllt, und miss, wie lange sie im Vergleich zu `pbc_clear` braucht.

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- Das Display hängt am **SPI-Bus**: SCK GP18, MOSI GP19, dazu CS GP10, DC GP8, RST GP9. **DC** unterscheidet Befehl (LOW) und Daten (HIGH).
- Ein Bereich wird mit **CASET**, **RASET** und **RAMWR** beschrieben; dann folgen die Pixel mit 16 Bit, Zeile für Zeile. Y-Versatz: **+20**.
- `spi_write_blocking` (8 Bit) und `spi_write16_blocking` (16 Bit) schicken Daten; das Format stellt `spi_set_format` ein.
- Der **DMA** kopiert Daten ohne die CPU. Mit **DREQ** läuft er im Takt des SPI-Busses. Solange er den Bildpuffer liest, nicht hineinzeichnen.
- Der SPI-Takt ist nicht beliebig: `spi_get_baudrate` zeigt, was wirklich eingestellt ist. Zeiten misst man, statt sie abzuschreiben.

KAPITEL 24

# Interrupts, Timer und der zweite Kern

**Worum geht's?** Bisher hat dein Programm alles nacheinander erledigt, in einer einzigen Schleife. Jetzt lernst du drei Wege kennen, mehrere Dinge gleichzeitig zu tun: einen **Timer**, der regelmäßig eine Funktion aufruft, **Interrupts**, die auf Tasten reagieren, und den **zweiten Kern** des RP2350, der ein ganz eigenes Programm ausführt. Zum Schluss geht es um die wichtigste Frage dabei: Wie teilen sich zwei Abläufe Daten, ohne sich gegenseitig zu stören?

## 24.1 Ein Timer, der regelmäßig etwas tut

Ein **Interrupt** (Unterbrechung) ist ein Signal der Hardware an den Prozessor: „Jetzt sofort etwas erledigen!" Der Prozessor unterbricht dann, was er gerade tut, ruft eine bestimmte Funktion auf, den **Interrupt-Handler**, und macht danach genau dort weiter, wo er war. Das Hauptprogramm merkt davon nichts, außer dass es einen Moment langsamer war. Kapitel 9 hat erzählt, dass die Bibliothek sich so jeden Tastendruck merkt.

Eine der Quellen für Interrupts ist der **Timer** des RP2350, eine Uhr, die in Mikrosekunden zählt. Das SDK baut darauf **wiederholende Timer**: Du nennst eine Funktion und einen Abstand, und das SDK ruft die Funktion in diesem Abstand immer wieder auf, ganz gleich, was `main` gerade tut.

k24_timer.c

```c
// Ein Timer lässt die grüne LED blinken, während main etwas anderes tut
#include "pbc.h"

static bool licht = false;

static bool blinken(repeating_timer_t *timer) {
    (void)timer;               // wird hier nicht gebraucht
    licht = !licht;
    pbc_led_green(licht);
    return true;               // true: weiter wiederholen
}

int main(void) {
    pbc_init();
    static repeating_timer_t timer;
    add_repeating_timer_ms(500, blinken, NULL, &timer);

    int zahl = 0;
    while (true) {
        char text[20];
        snprintf(text, sizeof text, "%d", zahl);
        pbc_clear(PBC_BLACK);
```

```
23          pbc_text24(text, 100, 120, PBC_WHITE);
24          pbc_show();
25          zahl++;
26          sleep_ms(100);
27      }
28  }
```

- Zeilen 6 bis 11: Die Funktion, die der Timer aufruft. Sie bekommt einen Zeiger auf den Timer (hier unbenutzt; `(void)timer;` sagt dem Compiler, dass das Absicht ist) und gibt `true` zurück, wenn es weitergehen soll. Mit `false` hört der Timer auf.
- Zeile 15: Das SDK speichert in einer Variablen vom Typ `repeating_timer_t`, was es über den Timer wissen muss. Sie muss so lange bestehen bleiben wie der Timer selbst, deshalb `static`.
- Zeile 16: `add_repeating_timer_ms(500, blinken, NULL, &timer)` startet den Timer: alle 500 ms `blinken` aufrufen. Das `NULL` ist ein Wert, den man der Funktion mitgeben könnte (hier nicht gebraucht).
- `main` zählt währenddessen auf dem Display hoch und weiß nichts vom Blinken.

Zum Abstand gibt es eine Feinheit, die in `pico/time.h` beschrieben ist: Eine **positive** Zahl ist der Abstand vom Ende eines Aufrufs bis zum Beginn des nächsten, eine **negative** der Abstand von Beginn zu Beginn. Mit `-10` kommt der Aufruf also genau alle 10 ms, auch wenn die Funktion selbst etwas Zeit braucht. Für Uhren nimmt man deshalb die negative Form.

**Eine Stoppuhr.** Hier zählt der Timer Hundertstelsekunden, und `main` zeigt sie an:

k24_stoppuhr.c

```c
// Eine Stoppuhr: Ein Timer zählt Hundertstelsekunden, main zeigt sie an
#include "pbc.h"

static volatile int hundertstel = 0;
static volatile bool laeuft = false;

static bool ticken(repeating_timer_t *timer) {
    (void)timer;
    if (laeuft) {
        hundertstel++;
    }
    return true;
}

int main(void) {
    pbc_init();
    static repeating_timer_t timer;
    add_repeating_timer_ms(-10, ticken, NULL, &timer);   // alle 10 ms

    while (true) {
        if (pbc_was_pressed_a()) {
            laeuft = !laeuft;           // A: Start und Stopp
        }
```

```
24          if (pbc_was_pressed_b() && !laeuft) {
25              hundertstel = 0;            // B: zurück auf null
26          }
27          int stand = hundertstel;        // einmal lesen, dann rechnen
28          char text[20];
29          snprintf(text, sizeof text, "%d.%02d s", stand / 100, stand % 100);
30          pbc_clear(PBC_BLACK);
31          pbc_text24(text, 60, 110, laeuft ? PBC_GREEN : PBC_WHITE);
32          pbc_text12("A: Start/Stopp   B: Null", 40, 170, PBC_GRAY);
33          pbc_show();
34      }
35  }
```

Neu ist das Wort **`volatile`** in den Zeilen 4 und 5. Es sagt dem Compiler: „Diese Variable kann sich jederzeit ändern, ohne dass du es in diesem Programmteil siehst." Warum das nötig ist, verstehst du, wenn du weißt, was der Compiler beim Optimieren tut: Er hält Werte gern in den schnellen Registern des Prozessors fest, statt sie jedes Mal neu aus dem Speicher zu lesen. In `main` wird `hundertstel` nirgends verändert, also könnte er annehmen, dass sich der Wert nie ändert. Der Timer ändert ihn aber im Interrupt. `volatile` verbietet diese Abkürzung: Jeder Zugriff geht wirklich in den Speicher. Aufgabe 2 zeigt, was ohne `volatile` passiert.

In Zeile 27 liest `main` den Zähler genau **einmal** in eine lokale Variable und rechnet danach nur noch mit ihr. Sonst könnte der Interrupt zwischen `stand / 100` und `stand % 100` zuschlagen, und Sekunden und Hundertstel passten nicht zusammen.

> Eine Variable, die ein Interrupt verändert und das Hauptprogramm liest (oder umgekehrt), bekommt **`volatile`**. Interrupt-Funktionen sind **kurz**: kein `sleep_ms`, kein `printf`, kein `pbc_show()`.

Warum so streng? Während ein Interrupt-Handler läuft, wartet alles andere, auch andere Interrupts mit gleicher oder niedrigerer Priorität, etwa die USB-Verbindung. Ein Handler, der lange wartet, legt den PicoBoy lahm.

## 24.2 Tasten per Interrupt

**So macht es die Bibliothek.** In `pbc_geraet.c` stehen die Tasten in den Zeilen 146 bis 229. Jetzt kannst du sie lesen:

- Zeilen 158 und 159: Für jede Taste merkt sich die Bibliothek, ob sie gedrückt wurde, und wann zuletzt eine Flanke kam. Beides ist `volatile`, denn der Interrupt schreibt und `pbc_was_pressed_…()` liest.
- Zeilen 175 bis 192, `tasten_init`: Jeder Tasten-Pin wird Eingang mit Pull-up (Kapitel 21.2). Dann meldet `gpio_set_irq_enabled` für jeden Pin einen Interrupt bei **beiden Flanken** an: bei der fallenden (`GPIO_IRQ_EDGE_FALL`, HIGH nach LOW, also beim Drücken) und der steigenden (`GPIO_IRQ_EDGE_RISE`, beim Loslassen).

- Zeilen 163 bis 173, `tasten_interrupt`: der Handler. Er fragt für jede Taste mit `gpio_get_irq_event_mask`, ob sie den Interrupt ausgelöst hat, und **quittiert** das Ereignis mit `gpio_acknowledge_irq`. Ohne Quittung würde derselbe Interrupt sofort wieder ausgelöst, immer wieder. Dann entprellt er: Jede Flanke merkt sich ihre Zeit. Ein Druck (fallende Flanke) zählt nur, wenn die Flanke davor mindestens 50 ms her ist. So zählt auch das Prellen beim Loslassen nicht als neuer Druck.
- Zeilen 196 bis 200: `war_gedrueckt` gibt den gemerkten Druck zurück und löscht ihn. Deshalb ist `pbc_was_pressed_a()` nur einmal pro Druck wahr.

Das SDK bietet zwei Arten, einen GPIO-Interrupt zu bedienen. Die einfache ist `gpio_set_irq_enabled_with_callback`: Man nennt eine Funktion, und das SDK ruft sie bei jedem Ereignis mit der Pin-Nummer auf. Es gibt aber nur **eine** solche Funktion für alle Pins. Die Bibliothek benutzt deshalb die zweite Art, einen „rohen“ Handler nur für ihre Pins (`gpio_add_raw_irq_handler_masked`, Zeile 183).

> Ein Blick in den Quelltext des SDK (`hardware_gpio/gpio.c`) zeigt: Die eine Funktion von `gpio_set_irq_enabled_with_callback` wird für Pins, die schon einen rohen Handler haben, **nie** aufgerufen. Nach `pbc_init()` gehören die sieben Tasten-Pins also der Bibliothek. Eigene Tasten-Interrupts gehen nur in einem Programm **ohne** `pbc_init()`.

**Selbst gemacht.** Dieses Programm verzichtet deshalb ausnahmsweise auf `pbc_init()`. Es richtet nur die Konsole, zwei LEDs und die Tasten A und B selbst ein. Das Display bleibt dunkel; die Ausgabe kommt im seriellen Monitor.

k24_taste_irq.c

```
 9 static volatile int zaehler_a = 0;
10 static volatile int zaehler_b = 0;
11 static volatile uint32_t zuletzt_us = 0;
12
13 // Läuft im Interrupt bei jeder Flanke an A oder B (drücken, loslassen).
14 static void taste_gedrueckt(uint pin, uint32_t ereignisse) {
15     uint32_t jetzt = time_us_32();
16     bool ruhig = jetzt - zuletzt_us >= 50000;
17     zuletzt_us = jetzt;                 // jede Flanke, auch beim Loslassen
18     if (!ruhig || !(ereignisse & GPIO_IRQ_EDGE_FALL)) {
19         return;                         // Prellen oder Loslassen
20     }
21     if (pin == PIN_TASTE_A) {
22         zaehler_a++;
23         gpio_xor_mask(1u << PIN_LED_GRUEN);   // grüne LED umschalten
24     } else if (pin == PIN_TASTE_B) {
25         zaehler_b++;
26         gpio_xor_mask(1u << PIN_LED_ROT);
27     }
28 }
```

```
30 static void taste_einrichten(uint pin) {
31     gpio_init(pin);
32     gpio_set_dir(pin, GPIO_IN);
33     gpio_pull_up(pin);
34 }
35
36 static void led_einrichten(uint pin) {
37     gpio_init(pin);
38     gpio_set_dir(pin, GPIO_OUT);
39     gpio_put(pin, 0);
40 }
41
42 int main(void) {
43     stdio_init_all();                   // nur die Konsole, kein pbc_init()
44     led_einrichten(PIN_LED_ROT);
45     led_einrichten(PIN_LED_GRUEN);
46     taste_einrichten(PIN_TASTE_A);
47     taste_einrichten(PIN_TASTE_B);
48     gpio_set_irq_enabled_with_callback(PIN_TASTE_A, GPIO_IRQ_EDGE_FALL |
       GPIO_IRQ_EDGE_RISE,
49                                        true, taste_gedrueckt);
50     gpio_set_irq_enabled(PIN_TASTE_B, GPIO_IRQ_EDGE_FALL | GPIO_IRQ_EDGE_RISE,
       true);
51
52     while (true) {
53         printf("A: %d mal, B: %d mal\n", zaehler_a, zaehler_b);
54         sleep_ms(1000);
55     }
56 }
```

- Zeilen 14 bis 28: der Handler. Er bekommt die Pin-Nummer und die Art des Ereignisses: `ereignisse & GPIO_IRQ_EDGE_FALL` ist wahr beim Drücken. Das Quittieren erledigt hier das SDK, bevor es die Funktion aufruft.
- Zeile 23: `gpio_xor_mask` kehrt die Pins in der Maske um, hier die grüne LED: an wird aus, aus wird an.
- Zeilen 48 und 49 legen die Funktion fest und schalten den Interrupt für A ein, Zeile 50 auch für B, jeweils für beide Flanken. Für B ist keine Funktion mehr nötig, es gibt ja nur eine.
- Zeile 43: `stdio_init_all()` richtet nur die Konsole ein. Das ist der erste Schritt von `pbc_init()` (Zeile 419 in `pbc_geraet.c`).

Zur Entprellung merkt sich der Handler, wann die letzte Flanke kam, auch beim Loslassen: Ein Druck zählt nur nach 50 ms Ruhe. Würde er nur die Zeit des letzten Drucks kennen, zählte das Prellen beim Loslassen nach einem langen Druck als neuer Druck (die Tasten des PicoBoy tun das manchmal). Er teilt sich die Zeit für beide Tasten; das ist einfacher als bei der Bibliothek, reicht aber für zwei Tasten, die man nicht gleichzeitig drückt.

## 24.3 `multicore_launch_core1`: Musik im Hintergrund

Der RP2350 hat **zwei Kerne**: zwei vollständige Prozessoren, die gleichzeitig arbeiten und denselben Speicher sehen. Bisher lief alles auf Kern 0; Kern 1 hat geschlafen. Mit `multicore_launch_core1(funktion)` startet Kern 1 und führt diese Funktion aus. Sie darf, wie `main`, nie enden.

**Die Bibliothek `pico_multicore`.** Die Funktionen für den zweiten Kern gehören nicht zu dem, was die Vorlage einbindet. Baust du das Programm unten mit der unveränderten Vorlage, übersetzt der Compiler `main.c` zwar, aber der Linker (Kapitel 2.3) findet den Code der Funktion nicht:

Terminal

```
ld: CMakeFiles/mein_programm.dir/main.c.o: in function `main':
main.c:35:(.text.startup.main+0xa): undefined reference to `multicore_launch_core1'
ld: (multicore_launch_core1): Unknown destination type (ARM/Thumb) in
  CMakeFiles/mein_programm.dir/main.c.o
main.c:35:(.text.startup.main+0xa): dangerous relocation: unsupported relocation
collect2: error: ld returned 1 exit status
```

*Undefined reference* heißt: Die Funktion ist in `pico/multicore.h` angekündigt, aber nirgends vorhanden. Die zweite und dritte Meldung sind nur Folgen davon. Deshalb musst du in der `CMakeLists.txt` deines Projekts (Kapitel 20.2) eine Zeile ändern:

```
target_link_libraries(mein_programm pbc)
```

wird zu

```
target_link_libraries(mein_programm pbc pico_multicore)
```

Die Programme dieses Abschnitts liegen deshalb als eigene Projekte im Ordner `projekte/` bei, jeweils mit `main.c` und der geänderten `CMakeLists.txt`.

Kern 1 spielt hier in Endlosschleife „Alle meine Entchen“, während Kern 0 einen Ball über den Bildschirm schickt. A schaltet die Musik aus und an:

main.c

```c
// Musik auf dem zweiten Kern, während der erste Kern animiert
#include "pbc.h"
#include "pico/multicore.h"

typedef struct {
    int frequenz;   // in Hz, 0 = Pause
    int dauer;      // in ms
} note;

static const note melodie[] = {   // "Alle meine Entchen"
    {262, 250}, {294, 250}, {330, 250}, {349, 250},
```

```
12      {392, 500}, {392, 500}, {440, 250}, {440, 250},
13      {440, 250}, {440, 250}, {392, 1000}, {0, 500},
14  };
15  #define ANZAHL_NOTEN (int)(sizeof melodie / sizeof melodie[0])
16
17  static volatile bool musik_an = true;   // schreibt Kern 0, liest Kern 1
18
19  // Diese Funktion läuft auf Kern 1 – für immer.
20  static void kern1(void) {
21      while (true) {
22          for (int i = 0; i < ANZAHL_NOTEN; i++) {
23              if (musik_an && melodie[i].frequenz > 0) {
24                  pbc_tone(melodie[i].frequenz, 0);
25              }
26              sleep_ms(melodie[i].dauer * 9 / 10);
27              pbc_tone_off();               // kurze Lücke zwischen den Noten
28              sleep_ms(melodie[i].dauer / 10);
29          }
30      }
31  }
32
33  int main(void) {
34      pbc_init();
35      multicore_launch_core1(kern1);        // ab hier arbeiten zwei Kerne
36
37      int x = 120;
38      int dx = 3;
39      while (true) {
40          if (pbc_was_pressed_a()) {
41              musik_an = !musik_an;
42          }
43          x += dx;
44          if (x < 30 || x > 210) {
45              dx = -dx;
46          }
47          pbc_clear(PBC_BLACK);
48          pbc_fill_circle(x, 140, 20, PBC_ORANGE);
49          pbc_text12(musik_an ? "A: Musik aus" : "A: Musik an", 80, 200,
50                     PBC_WHITE);
51          pbc_show();
52      }
53  }
```

- Zeilen 5 bis 15: Die Melodie als Array von Strukturen (Kapitel 13).
- Zeilen 20 bis 31: Das Programm für Kern 1. Es wartet mit `sleep_ms`, so lange jede Note dauert. Kern 0 merkt davon nichts.
- Zeile 35: Ab hier laufen zwei Programme gleichzeitig.
- Die Animation in `main` wartet nie auf die Musik, und die Musik nie auf `pbc_show()`. Ohne zweiten Kern müsste die Hauptschleife zwischen zwei Bildern ständig nachsehen, ob die nächste Note dran ist.
- Zeile 17: `musik_an` schreibt Kern 0 und liest Kern 1. Auch dafür braucht es `volatile`.

> Teile die Hardware auf: Hier gehört der Lautsprecher Kern 1, Display und Tasten gehören Kern 0. Benutzen beide Kerne denselben Teil, etwa beide `pbc_show()` oder beide `printf`, gibt es Durcheinander. Kern 1 hat einen eigenen Stack, ebenfalls 4 KB.

## 24.4 Gemeinsame Daten sicher teilen

Eine einzelne `bool`-Variable zu teilen, ist harmlos: Sie wird in einem Schritt geschrieben und gelesen. Schwieriger wird es, wenn beide Seiten denselben Wert **verändern**. Schon `zaehler++` besteht für den Prozessor aus drei Schritten: Wert lesen, eins dazuzählen, Wert zurückschreiben. Tun das zwei Kerne gleichzeitig, kann dies passieren:

| Kern 0 | Kern 1 | `zaehler` im Speicher |
|---|---|---|
| liest 41 | | 41 |
| | liest 41 | 41 |
| rechnet 42, schreibt 42 | | 42 |
| | rechnet 42, schreibt 42 | 42 |

Zweimal gezählt, aber nur um eins gewachsen. So ein Fehler heißt **Wettlauf** (englisch *race condition*). Er tritt nur manchmal auf, je nachdem, wie die Kerne zeitlich zueinander stehen. Deshalb ist er so schwer zu finden. `volatile` hilft hier **nicht**: Es sorgt nur dafür, dass wirklich gelesen und geschrieben wird, nicht dafür, dass niemand dazwischenkommt.

**Ein Mutex.** Die Lösung ist ein Schloss: Wer an die gemeinsamen Daten will, schließt vorher ab; wer abgeschlossen vorfindet, wartet. So ein Schloss heißt **Mutex** (von *mutual exclusion*, gegenseitiger Ausschluss). Das SDK hat ihn in `pico/mutex.h`:

| Funktion | Bedeutung |
|---|---|
| `mutex_init(&m)` | Mutex einrichten, einmal am Anfang |
| `mutex_enter_blocking(&m)` | abschließen; ist schon abgeschlossen, warten |
| `mutex_exit(&m)` | wieder aufschließen |

Dieses Programm lässt beide Kerne je 100 000-mal denselben Zähler erhöhen, erst ohne, dann mit Mutex:

main.c

```c
#define RUNDEN 100000

static volatile int zaehler = 0;
static volatile bool mit_mutex = false;
static mutex_t schloss;
```

```
12 static void zaehlen(void) {
13     for (int i = 0; i < RUNDEN; i++) {
14         if (mit_mutex) {
15             mutex_enter_blocking(&schloss);   // warten, bis frei, dann sperren
16             zaehler++;
17             mutex_exit(&schloss);             // wieder freigeben
18         } else {
19             zaehler++;
20         }
21     }
22 }
23
24 static void kern1(void) {
25     while (true) {
26         multicore_fifo_pop_blocking();        // auf das Startsignal warten
27         zaehlen();
28         multicore_fifo_push_blocking(1);      // melden: fertig
29     }
30 }
31
32 static int versuch(bool sperren) {
33     zaehler = 0;
34     mit_mutex = sperren;
35     multicore_fifo_push_blocking(1);          // Kern 1 losschicken
36     zaehlen();                                // Kern 0 zählt gleichzeitig
37     multicore_fifo_pop_blocking();            // warten, bis Kern 1 fertig ist
38     return zaehler;
39 }
40
41 int main(void) {
42     pbc_init();
43     mutex_init(&schloss);
44     multicore_launch_core1(kern1);
45     while (true) {
46         int ohne = versuch(false);
47         int mit = versuch(true);
48         printf("Erwartet: %d   ohne Mutex: %d   mit Mutex: %d\n",
49                2 * RUNDEN, ohne, mit);
50         char text[40];
51         pbc_clear(PBC_BLACK);
52         snprintf(text, sizeof text, "ohne Mutex: %d", ohne);
53         pbc_text18(text, 20, 100, PBC_RED);
54         snprintf(text, sizeof text, "mit Mutex:  %d", mit);
55         pbc_text18(text, 20, 140, PBC_GREEN);
56         pbc_show();
57         sleep_ms(2000);
58     }
59 }
```

- Zeilen 15 bis 17: Mit Mutex ist `zaehler++` geschützt. Nur ein Kern zugleich kann zwischen `mutex_enter_blocking` und `mutex_exit` sein.

- Zeilen 26, 28, 35 und 37: Die beiden Kerne geben sich über die **FIFO** Bescheid, eine kleine Warteschlange in der Hardware, die nur zwischen den beiden Kernen läuft. `multicore_fifo_push_blocking` legt eine Zahl hinein, `multicore_fifo_pop_blocking` wartet, bis eine da ist, und nimmt sie heraus.

Ohne Mutex ist das Ergebnis fast immer kleiner als 200 000, und bei jedem Durchlauf ein anderes. Mit Mutex stimmt es, kostet aber Zeit, denn das Abschließen braucht selbst einige Schritte.

**Eine Warteschlange.** Oft will eine Seite der anderen nur etwas mitteilen: „Spiel diesen Ton“, „Diese Taste wurde gedrückt“. Dafür gibt es in `pico/util/queue.h` die **Warteschlange** (*queue*). Sie ist intern mit einer Sperre geschützt, also sicher zwischen Kernen und auch zwischen Interrupt und Hauptprogramm. Was zuerst hineinkommt, kommt zuerst heraus.

| Funktion | Bedeutung |
|---|---|
| `queue_init(&q, groesse, anzahl)` | Platz für `anzahl` Einträge zu je `groesse` Bytes |
| `queue_try_add(&q, &wert)` | hineinlegen; `false`, wenn die Schlange voll ist |
| `queue_try_remove(&q, &wert)` | herausnehmen; `false`, wenn sie leer ist |
| `queue_add_blocking`, `queue_remove_blocking` | dasselbe, aber warten, bis es geht |

Die Einträge werden **kopiert**: Nach `queue_try_add` darfst du die Variable sofort wieder ändern. Hier wünscht sich Kern 0 Soundeffekte, und Kern 1 spielt sie ab:

main.c

```c
// Soundeffekte auf Kern 1: Kern 0 schickt Wünsche durch eine Warteschlange
#include "pbc.h"
#include "pico/multicore.h"
#include "pico/util/queue.h"

enum { TON_SPRUNG, TON_TREFFER };

static queue_t wuensche;

static void kern1(void) {
    while (true) {
        int wunsch;
        queue_remove_blocking(&wuensche, &wunsch);   // wartet auf einen Wunsch
        if (wunsch == TON_SPRUNG) {
            for (int f = 400; f <= 1200; f += 100) {
                pbc_tone(f, 0);
                sleep_ms(15);
            }
        } else {
            pbc_tone(150, 0);
            sleep_ms(300);
        }
```

```
23          pbc_tone_off();
24      }
25  }
26
27  static void wuenschen(int wunsch) {
28      if (!queue_try_add(&wuensche, &wunsch)) {
29          pbc_led_red(true);      // Warteschlange voll: Wunsch geht verloren
30      }
31  }
32
33  int main(void) {
34      pbc_init();
35      queue_init(&wuensche, sizeof(int), 8);   // Platz für 8 Wünsche
36      multicore_launch_core1(kern1);
37
38      int y = 200;
39      int vy = 0;
40      while (true) {
41          if (pbc_was_pressed_a() && y == 200) {
42              vy = -12;
43              wuenschen(TON_SPRUNG);
44          }
45          if (pbc_was_pressed_b()) {
46              wuenschen(TON_TREFFER);
47          }
48          y += vy;
49          vy += 1;
50          if (y >= 200) {
51              y = 200;
52              vy = 0;
53          }
54          pbc_clear(PBC_BLACK);
55          pbc_fill_rect(105, y, 30, 30, PBC_CYAN);
56          pbc_hline(20, 230, 200, PBC_WHITE);
57          pbc_text12("A: Sprung   B: Treffer", 45, 245, PBC_GRAY);
58          pbc_show();
59      }
60  }
```

- Zeile 13: Kern 1 wartet, bis ein Wunsch kommt. Solange keiner kommt, tut er nichts.
- Zeile 28: Kern 0 wartet nie. Ist die Schlange voll, weil jemand zu schnell drückt, geht der Wunsch verloren, und die rote LED geht an. Im Spiel ist das besser, als wenn das Bild hängen bliebe.
- Der Sprung in `main` läuft weiter flüssig, während der Ton spielt. In Kapitel 10 hättest du dafür auf `pbc_tone` mit Dauer zurückgreifen müssen, das nur einen einzigen Ton kann.

> Daten zwischen zwei Kernen (oder zwischen Interrupt und Hauptprogramm): Einzelne Merker mit `volatile`; alles, was beide verändern, mit **Mutex**; Nachrichten von einer Seite zur anderen mit einer **Warteschlange**.

## Ausprobieren: Eine Uhr im Hintergrund

- Ändere in `k24_stoppuhr.c` den Abstand von `-10` in `10`. Läuft die Uhr dann noch richtig? Vergleiche nach einer Minute mit einer echten Uhr.
- Spiele `k24_zaehler` mehrmals auf und notiere das Ergebnis ohne Mutex. Erhöhe `RUNDEN` auf 1 000 000. Was ändert sich?
- Gib in `k24_soundeffekte` der Schlange nur Platz für einen Eintrag und drücke B schnell hintereinander. Wann geht die rote LED an?

## Übungsaufgaben

**Aufgabe 1 ⭐ Lauflicht im Hintergrund** Lass einen wiederholenden Timer die drei Status-LEDs reihum einschalten: rot, gelb, grün, rot … alle 300 ms. `main` zeigt nur einen Text an.

**Aufgabe 2 ⭐⭐ Warum geht es nicht weiter?** Dieses Programm soll nach einer Sekunde „Fertig!“ zeigen. Es zeigt nur „Warte 1 Sekunde …“, für immer. Der Compiler warnt nicht. Finde den Fehler.

k24_aufgabe2.c

```c
// Aufgabe 24.2: Warum erscheint "Fertig!" nie?
#include "pbc.h"

static int zaehler = 0;

static bool zaehlen(repeating_timer_t *timer) {
    (void)timer;
    zaehler++;
    return true;
}

int main(void) {
    pbc_init();
    static repeating_timer_t timer;
    add_repeating_timer_ms(10, zaehlen, NULL, &timer);

    pbc_text18("Warte 1 Sekunde ...", 30, 110, PBC_WHITE);
    pbc_show();
    while (zaehler < 100) {
        // nichts tun, nur warten
    }
    pbc_text18("Fertig!", 80, 150, PBC_GREEN);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

**Aufgabe 3 ⭐⭐ Ein Regenbogen auf Kern 1** Kern 1 soll die RGB-LED langsam und endlos durch die Farben wandern lassen: von Rot über Grün nach Blau und zurück zu Rot. Kern 0 zählt währenddessen auf dem Display so schnell hoch, wie er kann.

**Aufgabe 4 ⭐⭐⭐ Musik auf Befehl** Kern 1 kennt zwei Melodien. Kern 0 schickt ihm über eine Warteschlange Befehle: A startet die erste Melodie, B die zweite, die Mitte des Joysticks schaltet die Musik ab. Kern 1 soll einen neuen Befehl schon zwischen zwei Noten bemerken, nicht erst am Ende der Melodie.

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- Ein **Interrupt** unterbricht das Programm und ruft einen **Handler** auf. Handler sind kurz: kein Warten, kein `printf`.
- `add_repeating_timer_ms(abstand, funktion, NULL, &timer)` ruft `funktion` immer wieder auf; negativer Abstand = von Beginn zu Beginn. Die Funktion gibt `true` zurück, um weiterzumachen.
- GPIO-Interrupts: `gpio_set_irq_enabled_with_callback` (eine Funktion für alle Pins). Die Tasten-Pins belegt `pbc_init()` mit einem eigenen Handler.
- `multicore_launch_core1(funktion)` startet Kern 1. Dafür braucht die `CMakeLists.txt` die Bibliothek **`pico_multicore`**.
- Gemeinsame Daten: **`volatile`** für Merker, **Mutex** für Daten, die beide ändern, **Warteschlange** (`pico/util/queue.h`) für Nachrichten.

KAPITEL 25

# Daten dauerhaft speichern

**Worum geht's?** Schaltest du den PicoBoy aus, ist alles im RAM verloren, auch der Highscore aus Kapitel 18. Der Flash-Speicher behält seinen Inhalt dagegen ohne Strom; dort steht ja auch dein Programm. Du lernst, wie man den Flash liest, löscht und beschreibt, speicherst einen Highscore, der das Ausschalten übersteht, und erfährst, was dabei schiefgehen kann.

## 25.1 Der Flash: löschen in Blöcken, schreiben in Seiten

Der PicoBoy hat einen Flash-Baustein mit **16 MB**. Am Anfang steht dein Programm, meist weniger als 1 MB. Der Rest ist frei.

**Lesen ist einfach.** Der RP2350 blendet den ganzen Flash in seinen Adressraum ein, ab der Adresse `0x10000000`. Im SDK heißt diese Adresse `XIP_BASE` (*Execute in Place*: Das Programm wird direkt aus dem Flash ausgeführt). Ein Byte im Flash liest du deshalb wie jedes andere Byte im Speicher, über einen Zeiger (Kapitel 19):

k25_flash_lesen.c

```c
// Den Flash lesen: Er ist ab XIP_BASE in den Adressraum eingeblendet
#include "pbc.h"
#include "hardware/flash.h"

#define SPEICHER_OFFSET (PICO_FLASH_SIZE_BYTES - 2 * FLASH_SECTOR_SIZE)

static void zeige(const char *name, uint32_t offset) {
    const uint8_t *p = (const uint8_t *)(XIP_BASE + offset);
    printf("%s (Adresse 0x%08X):", name, (unsigned)(XIP_BASE + offset));
    for (int i = 0; i < 8; i++) {
        printf(" %02X", p[i]);
    }
    printf("\n");
}

int main(void) {
    pbc_init();
    printf("Flash: %d Bytes, Sektor %u Bytes, Seite %u Bytes\n",
           PICO_FLASH_SIZE_BYTES, FLASH_SECTOR_SIZE, FLASH_PAGE_SIZE);
    zeige("Programmanfang  ", 0);
    zeige("Speicher-Sektor ", SPEICHER_OFFSET);
    while (true) {
        sleep_ms(1000);
    }
}
```

- Zeile 8: `XIP_BASE + offset` ist eine Zahl; `(const uint8_t *)` macht daraus einen Zeiger auf Bytes. `const`, weil man so nicht in den Flash schreiben kann.
- Zeile 5 legt fest, wo die Daten dieses Kapitels liegen. Warum gerade dort, steht weiter unten.

- `PICO_FLASH_SIZE_BYTES` stammt aus der Board-Datei der Vorlage, die anderen Namen aus `hardware/flash.h`.

Am Anfang stehen Bytes deines Programms. Was im Speicher-Sektor steht, hängt davon ab, was vorher auf dem PicoBoy war: `FF FF FF …`, wenn er gelöscht ist, sonst Reste eines früheren Programms.

**Schreiben ist schwieriger.** Ein Flash-Baustein kann nicht einfach ein Byte überschreiben. Es gibt zwei getrennte Vorgänge:

| Vorgang | Funktion | Einheit | Was passiert |
|---|---|---|---|
| **Löschen** | `flash_range_erase(offset, anzahl)` | ein **Sektor** = 4096 Bytes (`FLASH_SECTOR_SIZE`) | alle Bits werden **1**, jedes Byte also `0xFF` |
| **Schreiben** | `flash_range_program(offset, daten, anzahl)` | eine **Seite** = 256 Bytes (`FLASH_PAGE_SIZE`) | Bits können nur von **1 auf 0** wechseln |

Ein Sektor besteht aus 16 Seiten. `offset` ist bei beiden Funktionen der Abstand vom **Anfang des Flash**, nicht die Adresse: `0` ist das erste Byte, nicht `0x10000000`. Löschen beginnt an einer Sektorgrenze und umfasst ganze Sektoren; Schreiben beginnt an einer Seitengrenze und umfasst ganze Seiten.

Dass Schreiben nur Nullen machen kann, zeigt dieses Programm. Es löscht den Sektor, schreibt `0xF0` und dann, ohne zu löschen, `0x0F` darüber:

k25_bits.c

```
18
19 static void erstes_byte(const char *schritt) {
20     const uint8_t *flash = (const uint8_t *)(XIP_BASE + SPEICHER_OFFSET);
21     printf("%-22s erstes Byte: 0x%02X\n", schritt, flash[0]);
22 }
23
24 int main(void) {
25     pbc_init();
26     flash_safe_execute(loeschen, NULL, 100);
27     erstes_byte("nach dem Loeschen:");
28
29     memset(seite, 0xFF, sizeof seite);   // 0xFF: diese Bits nicht ändern
30     seite[0] = 0xF0;                     // binär 1111 0000
31     flash_safe_execute(schreiben, seite, 100);
32     erstes_byte("0xF0 geschrieben:");
33
34     seite[0] = 0x0F;                     // binär 0000 1111
35     flash_safe_execute(schreiben, seite, 100);
36     erstes_byte("0x0F darueber:");
```

```
37      while (true) {
38          sleep_ms(1000);
39      }
40  }
```

`1111 0000` und danach `0000 1111`: Wo im zweiten Wert eine 0 steht, wird das Bit 0; wo eine 1 steht, bleibt es, wie es war. Übrig bleibt `0000 0000`. Das Programm sollte also `0xFF`, `0xF0` und `0x00` ausgeben. Wer einen Wert ändern will, muss deshalb erst den ganzen Sektor löschen und dann neu schreiben.

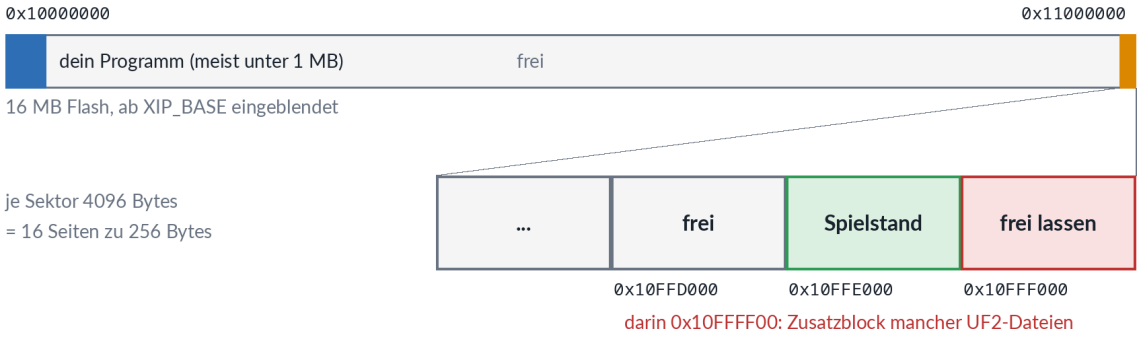
> Flash: **löschen** sektorweise (4096 Bytes, alles wird `0xFF`), **schreiben** seitenweise (256 Bytes, Bits nur 1 → 0). Die Funktionen nehmen den **Abstand vom Flash-Anfang**, gelesen wird über **`XIP_BASE` + Abstand**.

**Warum `flash_safe_execute`?** Während der Flash gelöscht oder beschrieben wird, kann der Prozessor nichts daraus lesen, also auch keinen Programmcode. Läuft in dieser Zeit ein Interrupt-Handler, der im Flash liegt (zum Beispiel der Tasten-Interrupt der Bibliothek), oder liest der zweite Kern Code aus dem Flash, stürzt das Programm ab. Die Funktionen `flash_range_erase` und `flash_range_program` selbst liegen im RAM, darum kümmert sich das SDK. Aber alles andere muss so lange warten.

Genau das erledigt `flash_safe_execute(funktion, daten, wartezeit_ms)` aus `pico/flash.h`: Es schaltet die Interrupts ab, sorgt dafür, dass der andere Kern stillhält, ruft `funktion(daten)` auf und schaltet danach alles wieder ein. Es gibt `PICO_OK` zurück, wenn es geklappt hat. Deshalb stehen das Löschen und das Schreiben in `k25_bits.c` in eigenen kleinen Funktionen mit einem `void *`-Parameter. Über ihn kann man beliebige Daten mitgeben, etwa die zu schreibende Seite.

Die Daten zum Schreiben müssen im **RAM** liegen, denn während des Schreibens ist der Flash ja nicht lesbar. `seite` ist deshalb eine `static`-Variable, kein `const`-Array.

**Wohin mit den Daten?** Am Anfang steht das Programm; ein Sektor dort zu löschen, zerstört es. Am sichersten liegt man ganz am Ende des Flash. Aber nicht im **letzten** Sektor:



Viele `.uf2`-Dateien für den RP2350 enthalten einen Zusatzblock für die Adresse `0x10FFFF00`, zum Beispiel die MicroPython-Firmware des PicoBoy, mehrere Spiele für ihn und auch die Programme aus diesem Buch. Das ist ein Ausweg des Herstellers für einen Fehler im Chip (im SDK „RP2350-E10“ genannt). Beim Aufspielen einer solchen Datei landet dieser Block im letzten Sektor. Das SDK selbst legt deshalb eigene Speicherdaten (für Bluetooth auf anderen Boards) nicht in den letzten Sektor. Wir halten es genauso und nehmen den **vorletzten**:

```
#define SPEICHER_OFFSET (PICO_FLASH_SIZE_BYTES - 2 * FLASH_SECTOR_SIZE)
```

Das ist der Abstand 16 777 216 − 8192 = 16 769 024, hexadezimal `0xFFE000`, also die Adresse `0x10FFE000`. Dein Programm müsste fast 16 MB groß sein, um bis dorthin zu reichen.

## 25.2 Einen Highscore speichern und lesen

In diesem kleinen Spiel drückst du fünf Sekunden lang so oft wie möglich auf A. Der Rekord bleibt erhalten, auch wenn du den PicoBoy ausschaltest oder ein neues Programm aufspielst.

k25_highscore.c

```c
// Ein Highscore, der das Ausschalten übersteht
#include "pbc.h"
#include "hardware/flash.h"
#include "pico/flash.h"

// Vorletzter Sektor des Flash; der letzte bleibt frei (Abschnitt 25.1)
#define SPEICHER_OFFSET (PICO_FLASH_SIZE_BYTES - 2 * FLASH_SECTOR_SIZE)
#define KENNUNG 0x50424331u   // zeigt: Hier liegt wirklich unser Spielstand

typedef struct {
    uint32_t kennung;
    uint32_t highscore;
    uint32_t pruefsumme;
} spielstand;

static uint32_t pruefsumme(const spielstand *s) {
    return (s->kennung ^ s->highscore) + 12345u;
}

static int highscore_lesen(void) {
    const spielstand *s = (const spielstand *)(XIP_BASE + SPEICHER_OFFSET);
    if (s->kennung != KENNUNG || s->pruefsumme != pruefsumme(s)) {
        return 0;                       // leer oder fremde Daten
    }
    return (int)s->highscore;
}

// Läuft mit gesperrten Interrupts (siehe flash_safe_execute).
static void flash_schreiben(void *daten) {
    flash_range_erase(SPEICHER_OFFSET, FLASH_SECTOR_SIZE);
```

```
31      flash_range_program(SPEICHER_OFFSET, daten, FLASH_PAGE_SIZE);
32  }
33
34  static bool highscore_speichern(int punkte) {
35      static uint8_t seite[FLASH_PAGE_SIZE];   // eine ganze Seite im RAM
36      spielstand s = { KENNUNG, (uint32_t)punkte, 0 };
37      s.pruefsumme = pruefsumme(&s);
38      memset(seite, 0xFF, sizeof seite);
39      memcpy(seite, &s, sizeof s);
40      return flash_safe_execute(flash_schreiben, seite, 100) == PICO_OK;
41  }
```

**Der Spielstand als Struktur.** `spielstand` (Zeilen 10 bis 14) enthält außer dem Highscore noch zwei Zahlen, die nur der Sicherheit dienen:

- **`kennung`**: eine feste Zahl, die zeigt: Hier liegt wirklich ein Spielstand dieses Programms und nicht `0xFF…` aus einem gelöschten Sektor oder Reste von etwas anderem. `0x50424331` sind übrigens die ASCII-Zeichen „PBC1“.
- **`pruefsumme`**: aus den anderen Werten errechnet (Zeilen 16 bis 18). Passt sie beim Lesen nicht, ist etwas kaputt, zum Beispiel weil der Strom mitten im Schreiben weg war.

**Lesen** (Zeilen 20 bis 26): Ein Zeiger auf `spielstand` zeigt direkt in den Flash. Stimmen Kennung und Prüfsumme nicht, gilt der Highscore als 0.

**Speichern** (Zeilen 34 bis 41):

1. Eine ganze Seite von 256 Bytes im RAM vorbereiten, denn `flash_range_program` schreibt immer ganze Seiten. `memset` füllt sie mit `0xFF`, `memcpy` kopiert die 12 Bytes des Spielstands an den Anfang. Beide stehen in `string.h`, das `pbc.h` schon einbindet.
2. `flash_safe_execute` ruft `flash_schreiben` auf (Zeilen 29 bis 32): Sektor löschen, Seite schreiben. Die Wartezeit von 100 ms betrifft nur die Abstimmung mit dem zweiten Kern.
3. Das Ergebnis meldet zurück, ob es geklappt hat.

Der Rest des Programms ist ein Spiel wie in Kapitel 18:

k25_highscore.c

```
43  static void anzeigen(const char *oben, int zahl, int rekord) {
44      char text[30];
45      pbc_clear(PBC_BLACK);
46      pbc_text18(oben, 30, 70, PBC_WHITE);
47      snprintf(text, sizeof text, "%d", zahl);
48      pbc_text24(text, 105, 120, PBC_YELLOW);
49      snprintf(text, sizeof text, "Rekord: %d", rekord);
50      pbc_text18(text, 65, 190, PBC_GREEN);
51      pbc_show();
52  }
53
54  int main(void) {
55      pbc_init();
```

```
56      int rekord = highscore_lesen();
57      while (true) {
58          anzeigen("A: 5 Sekunden lang", 0, rekord);
59          while (!pbc_was_pressed_a()) {
60              sleep_ms(10);
61          }
62          int punkte = 1;                  // der Startdruck zählt schon
63          uint32_t ende = pbc_millis() + 5000;
64          while (pbc_millis() < ende) {
65              if (pbc_was_pressed_a()) {
66                  punkte++;
67              }
68              anzeigen("Druecke A!", punkte, rekord);
69          }
70          if (punkte > rekord) {
71              rekord = punkte;
72              bool ok = highscore_speichern(rekord);
73              anzeigen(ok ? "Neuer Rekord!" : "Speichern ging schief",
74                       punkte, rekord);
75          } else {
76              anzeigen("Vorbei!", punkte, rekord);
77          }
78          sleep_ms(2000);
79          pbc_was_pressed_a();             // Druck aus der Pause verwerfen
80      }
81  }
```

Gespeichert wird nur, wenn es einen **neuen** Rekord gibt (Zeile 70). Das ist Absicht, wie der nächste Abschnitt zeigt.

> Aufspielen per **Run** oder per Kopieren überschreibt nur die Teile des Flash, in denen das neue Programm liegt. Dein Spielstand am Ende bleibt also erhalten, auch wenn du das Programm änderst. Anders kann es sein, wenn zwischendurch ein ganz anderes Programm lief, etwa MicroPython mit seinem Dateisystem: Das darf den Flash nach eigenen Regeln benutzen. Dann ist der Spielstand weg, und Kennung und Prüfsumme sorgen dafür, dass das Spiel das merkt. Löschen kannst du ihn mit einem Programm, das den Sektor löscht (Aufgabe 3), oder mit `picotool` (Anhang E).

## 25.3 Was dabei schiefgehen kann

**Zu oft schreiben.** Jeder Sektor verträgt nur eine begrenzte Zahl von Lösch- und Schreibvorgängen. Laut Datenblatt üblicher Flash-Bausteine sind es mindestens 100 000. Das klingt viel. Speichert ein Programm aber jede Sekunde, ist der Sektor nach gut einem Tag Spielzeit verbraucht. Deshalb: nur speichern, wenn sich etwas Wichtiges geändert hat, nie in jedem Bild. Aufgabe 4 zeigt, wie man den Sektor zusätzlich schont.

**Strom weg beim Schreiben.** Löschen und Schreiben dauern einen Moment, beim Löschen eines Sektors deutlich länger als beim Schreiben einer Seite. Fällt genau dann der Strom aus, ist der alte Spielstand gelöscht, der neue noch nicht geschrieben. Kennung und Prüfsumme sorgen dafür, dass das Programm dann nicht mit Unsinn weiterrechnet, sondern bei 0 beginnt. Wer auch den alten Wert retten will, benutzt zwei Sektoren abwechselnd und liest den neueren gültigen.

**Die falsche Stelle.** Ein Fehler bei `SPEICHER_OFFSET` kann dein eigenes Programm löschen. Das Programm stürzt dann sofort ab, und der PicoBoy startet erst wieder, wenn du über den Bootmodus (BOOT halten, RES tippen) ein neues Programm aufspielst. Kaputt ist dabei nichts. Abstände über 16 MB hinaus fängt das SDK ab: Es hält das Programm an (`hard_assert` in `flash_range_erase`).

**Vergessene Ausrichtung.** Der Abstand muss beim Löschen ein Vielfaches von 4096 sein, beim Schreiben eines von 256, die Anzahl ebenso. Das SDK prüft das in der Grundeinstellung **nicht** (`invalid_params_if` ist abgeschaltet). Was dann gelöscht oder geschrieben wird, ist nicht das, was du wolltest. Rechne deshalb immer mit `FLASH_SECTOR_SIZE` und `FLASH_PAGE_SIZE`, nie mit selbst ausgedachten Zahlen.

**Interrupts und der zweite Kern.** `flash_safe_execute` kümmert sich um beides. In einem Programm mit zweitem Kern (Kapitel 24.3) muss Kern 1 dafür einmal `flash_safe_execute_core_init()` aufrufen, bevor Kern 0 schreibt. Sonst kann `flash_safe_execute` den anderen Kern nicht anhalten und schreibt nicht, sondern meldet einen Fehler.

**Eine größere Struktur.** Ändert sich der Aufbau von `spielstand` in einer neuen Fassung deines Programms, passen die alten Daten nicht mehr. Ändere dann auch die Kennung, etwa auf `0x50424332` („PBC2“), damit das Programm alte Daten nicht falsch liest.

> Speichere im Flash keine Daten, die öfter als nötig neu geschrieben werden, und rufe `flash_range_erase` und `flash_range_program` nur über `flash_safe_execute` auf. Der letzte Sektor (ab `0x10FFF000`) bleibt frei.

## Ausprobieren: Der Rekord überlebt

- Spiele `k25_highscore.c` auf, stelle einen Rekord auf und ziehe dann das USB-Kabel ab (steht der Schalter auf BAT, schiebe ihn auf USB – ohne Kabel ist der PicoBoy dann aus). Ist der Rekord nach dem Einschalten noch da?
- Spiele danach `k25_flash_lesen.c` auf. Welche Bytes stehen jetzt im Speicher-Sektor? Findest du die Kennung wieder? (Sie steht mit dem niedrigsten Byte zuerst im Speicher: `31 43 42 50`.)
- Ändere in `k25_highscore.c` die Kennung. Was zeigt das Spiel dann als Rekord?

## Übungsaufgaben

**Aufgabe 1** ⭐ **Wie oft eingeschaltet?** Schreibe ein Programm, das bei jedem Start eine Zahl im Flash um eins erhöht und anzeigt: „Start Nr. 7“.

**Aufgabe 2** ⭐⭐ **Wie lange hält der Sektor?** Ein Sektor verträgt 100 000 Löschvorgänge. Wie lange hält er, wenn ein Programm alle 10 Sekunden speichert? Wie lange, wenn ein Spiel nur bei jedem neuen Rekord speichert und du jeden Tag zehn neue Rekorde aufstellst?

**Aufgabe 3** ⭐⭐ **Rekord zurücksetzen** Erweitere `k25_highscore.c`: Hält man beim Einschalten B gedrückt, wird der Rekord gelöscht.

**Aufgabe 4** ⭐⭐⭐ **Den Sektor schonen** Ein Sektor hat 16 Seiten. Statt bei jedem Speichern zu löschen, schreibe den neuen Rekord in die **nächste freie** Seite. Erst wenn alle 16 Seiten beschrieben sind, wird der Sektor gelöscht, und es geht bei Seite 0 weiter. Beim Lesen gilt die letzte beschriebene Seite. Um wie viel länger hält der Sektor so?

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- Der Flash ist ab **`XIP_BASE`** (`0x10000000`) eingeblendet: Lesen geht über einen Zeiger.
- **`flash_range_erase`** löscht ganze Sektoren (4096 Bytes, danach `0xFF`), **`flash_range_program`** schreibt ganze Seiten (256 Bytes, Bits nur 1 → 0). Beide nehmen den Abstand vom Flash-Anfang.
- Beide nur über **`flash_safe_execute`** aufrufen; die Daten liegen im RAM.
- Speicherort: der **vorletzte** Sektor, `PICO_FLASH_SIZE_BYTES - 2 * FLASH_SECTOR_SIZE`. Der letzte bleibt frei.
- Kennung und Prüfsumme erkennen leere oder kaputte Daten. Nur speichern, wenn nötig: Jeder Sektor verträgt nur eine begrenzte Zahl von Löschungen.

KAPITEL 26

# Dein eigenes Projekt

**Worum geht's?** Du kannst jetzt zeichnen, Tasten und Sensor abfragen, Töne spielen, Daten speichern und die Hardware direkt ansprechen. Zeit für ein eigenes Projekt. Dieses Kapitel hilft dir, eine Idee zu finden, sie in kleine Schritte zu zerlegen, Fehler systematisch zu suchen und das Projekt fertigzustellen. Es enthält wenig neuen Code, dafür eine Arbeitsweise, die für jedes größere Programm taugt.

## 26.1 Eine Idee finden

Das beste Projekt ist eines, das **dich** interessiert, und das klein genug ist, um in ein paar Abenden fertig zu werden. Ein fertiges kleines Spiel ist mehr wert als ein halbes großes. Ein paar Anregungen, sortiert nach dem, was sie aus dem Buch brauchen:

| Idee | Was du brauchst | Kapitel |
|---|---|---|
| **Snake**: eine Schlange frisst Punkte und wird länger | Arrays, Tasten, Kollision | 12, 9, 16 |
| **Flappy**: mit A durch Lücken in Röhren fliegen | Schwerkraft, Kollision, Highscore im Flash | 14, 16, 25 |
| **Murmellabyrinth**: eine Kugel durch Kippen ins Ziel rollen | Neigungssensor, Glätten, Kollision | 17, 16 |
| **Simon sagt**: Farbfolgen merken und nachdrücken | RGB-LED, Töne, Zufall, Arrays | 10, 5, 12 |
| **Reaktionsduell**: zwei Spieler, A gegen B | Zeit messen, Tasten | 7, 9 |
| **Würfelbecher**: Schütteln würfelt | Neigungssensor, Zufall, Bilder | 17, 15 |
| **Stoppuhr mit Rundenzeiten** | Timer-Interrupt, Speichern | 24, 25 |
| **Musikbox**: Melodien auswählen und abspielen, Animation dazu | Strukturen, zweiter Kern | 13, 24 |
| **Wetterstation ohne Wetter**: Batteriespannung als Kurve über die Zeit | ADC, Zeichnen, Arrays | 21, 6, 12 |

Oder du baust ein Spiel nach, das du kennst, in vereinfachter Form. Das ist eine gute Übung, weil du schon weißt, wie es sich anfühlen soll.

> Schreib deine Idee in **einem Satz** auf: „Eine Kugel rollt durch Kippen durch ein Labyrinth; wer schneller im Ziel ist, steht oben in der Liste." Passt sie nicht in einen Satz, ist sie wahrscheinlich noch zu groß. Spare Extras für später auf.

## 26.2 Planen: Zustände, Daten, Schritte

Bevor du die erste Zeile schreibst, beantworte drei Fragen auf Papier:

1. **Welche Zustände hat das Programm?** Fast jedes Spiel hat mindestens Titel, Spiel und Ende (Kapitel 18.1). Manche haben mehr: Pause, Levelwechsel, Einstellungen. Zeichne sie als Kästchen und die Übergänge als Pfeile: „A gedrückt“, „Leben = 0“, „10 Sekunden um“.
2. **Welche Daten braucht es?** Was muss sich das Programm merken? Spielerposition, Punkte, Leben, die Liste der Gegner … Daraus werden Variablen, Strukturen und Arrays. Überlege bei Arrays gleich, wie groß sie höchstens werden.
3. **In welchen Schritten baust du es?** Jeder Schritt endet mit einem Programm, das läuft und das du ausprobieren kannst. Nie mehr als eine neue Sache pro Schritt.

Für „Murmellabyrinth“ könnten die Schritte so aussehen:

1. Eine Kugel in der Mitte, die sich mit dem Joystick bewegt.
2. Statt Joystick: Kippen (Kapitel 17), mit Glätten.
3. Eine Wand als Rechteck; die Kugel prallt ab.
4. Mehrere Wände aus einem Array von Strukturen.
5. Ein Ziel; erreicht die Kugel es, erscheint „Geschafft“ mit der Zeit.
6. Titel- und Endbildschirm.
7. Die beste Zeit im Flash speichern.
8. Extras: Töne, mehrere Level, Löcher, in die man fällt.

**Das Gerüst.** Dieses Programm ist ein leerer Rahmen mit drei Zuständen und einer festen Bildrate. Im Zustand `SPIEL` steht ein Platzhalter: A gibt einen Punkt, nach zehn Sekunden ist Schluss. Hier hinein baust du dein Spiel.

k26_geruest.c

```c
// Ein Gerüst für dein eigenes Spiel: drei Zustände, eine Hauptschleife
#include "pbc.h"

typedef enum { TITEL, SPIEL, ENDE } zustand;

int main(void) {
    pbc_init();
    zustand modus = TITEL;
    int punkte = 0;
    uint32_t start = 0;

    while (true) {
        uint32_t bild_start = pbc_millis();
        pbc_clear(PBC_BLACK);
        if (modus == TITEL) {
            pbc_text24("Mein Spiel", 55, 100, PBC_YELLOW);
            pbc_text12("A: Start", 90, 160, PBC_WHITE);
```

```
18              if (pbc_was_pressed_a()) {
19                  punkte = 0;
20                  start = pbc_millis();
21                  modus = SPIEL;
22              }
23          } else if (modus == SPIEL) {
24              // Hier kommt dein Spiel hin. Platzhalter: A gibt Punkte.
25              if (pbc_was_pressed_a()) {
26                  punkte++;
27              }
28              char text[20];
29              snprintf(text, sizeof text, "Punkte: %d", punkte);
30              pbc_text18(text, 60, 120, PBC_WHITE);
31              if (pbc_millis() - start > 10000) {
32                  modus = ENDE;
33              }
34          } else {
35              pbc_text24("Ende", 90, 100, PBC_RED);
36              pbc_text12("B: zum Titel", 80, 160, PBC_WHITE);
37              if (pbc_was_pressed_b()) {
38                  modus = TITEL;
39              }
40          }
41          pbc_show();
42          uint32_t dauer = pbc_millis() - bild_start;
43          if (dauer < 40) {
44              sleep_ms(40 - dauer);   // höchstens 25 Bilder pro Sekunde
45          }
46      }
47  }
```

- Zeile 4: Die Zustände als `enum` (Kapitel 13.3).
- Zeilen 13 und 42 bis 45: Die Schleife misst, wie lange ein Bild gedauert hat, und wartet den Rest von 40 ms ab (Kapitel 14.2).
- Jeder Zustand zeichnet sein Bild und prüft, ob es zum nächsten Zustand weitergeht. `pbc_show()` steht nur einmal, am Ende der Schleife.




Wird das Programm länger als ein paar hundert Zeilen, teile es in Funktionen (`titel_zeichnen`, `spiel_schritt` …) und, wenn nötig, in mehrere Dateien (Kapitel 20). Eine Funktion sollte auf eine Bildschirmseite passen.

## 26.3 Fehler finden

Irgendwann tut dein Programm nicht, was es soll. Das ist normal. Wichtig ist ein Vorgehen, das sicher zum Fehler führt, statt wild herumzuprobieren.

**1. Den Fehler genau beschreiben.** Nicht „geht nicht“, sondern: „Der Ball verschwindet am unteren Rand, statt abzuprallen. Oben, links und rechts klappt es.“ Oft liegt die Lösung schon in dieser Beschreibung.

**2. Nachsehen statt raten.** Gib die Werte aus, um die es geht, mit `printf` in den seriellen Monitor. Gib nicht alles aus, sondern nur das, was an der verdächtigen Stelle passiert, sonst geht es in der Flut unter. Ein Beispiel: Bei diesem Ball klappt das Abprallen unten nicht.

k26_aufgabe3.c

```c
// Aufgabe 26.3: Der Ball soll an allen vier Rändern abprallen. Tut er nicht.
#include "pbc.h"

int main(void) {
    pbc_init();
    int x = 120;
    int y = 140;
    int dx = 4;
    int dy = 3;
    while (true) {
        x += dx;
        y += dy;
        if (x < 30 || x > 210) {
            dx = -dx;
        }
        if (y < 30 || x > 250) {
            dy = -dy;
        }
        pbc_clear(PBC_BLACK);
        pbc_fill_circle(x, y, 10, PBC_GREEN);
        pbc_show();
    }
}
```

Nach drei Sekunden ist der Bildschirm leer, der Ball ist unten verschwunden. Ein `printf` in der Schleife, das nur meldet, wenn der Ball schon tief unten ist (`if (y > 240)`), zeigt:

Serial Monitor

```
x=168 y=242 dx=-4 dy=3
x=164 y=245 dx=-4 dy=3
x=160 y=248 dx=-4 dy=3
x=156 y=251 dx=-4 dy=3
x=152 y=254 dx=-4 dy=3
x=148 y=257 dx=-4 dy=3
x=144 y=260 dx=-4 dy=3
x=140 y=263 dx=-4 dy=3
x=136 y=266 dx=-4 dy=3
x=132 y=269 dx=-4 dy=3
```

`y` wächst und wächst, `dy` bleibt 3. Die Bedingung, die `dy` umdrehen soll, wird also nie wahr. Was in ihr falsch ist, findest du in Aufgabe 3.

**3. Den Fehler eingrenzen.** Kommentiere Teile aus, bis der Fehler verschwindet. Dann liegt er im zuletzt auskommentierten Teil. Oder geh zurück zum letzten Schritt, der noch lief, und baue das Neue in kleineren Stücken ein.

**4. Die üblichen Verdächtigen prüfen:**

| Beobachtung | Häufige Ursache |
|---|---|
| Baut nicht | erste Fehlermeldung lesen (Kapitel 3.5, Anhang B) |
| Warnung beim Bauen | fast immer ein echter Fehler (Kapitel 4.3) |
| Bildschirm bleibt schwarz | `pbc_show()` fehlt, oder das Programm hängt vor dem ersten Bild |
| Programm bleibt stehen, reagiert auf nichts | Endlosschleife ohne Ausweg; Stack übergelaufen (Kapitel 19.5) |
| Etwas passiert genau einmal zu oft oder zu wenig | `<` statt `<=`, Zählung ab 0 oder ab 1 |
| Werte springen plötzlich auf Unsinn | über das Ende eines Arrays geschrieben (Kapitel 12.5) |
| `if` reagiert immer | `=` statt `==` (Kapitel 8.5) |
| Taste reagiert zweimal oder gar nicht | `pbc_pressed_…` und `pbc_was_pressed_…` verwechselt (Kapitel 9) |
| Interrupt- oder Kern-Variable ändert sich nicht | `volatile` fehlt (Kapitel 24) |

**5. Erklären.** Erkläre das Programm Zeile für Zeile jemandem, notfalls einem Stofftier. Dabei fällt einem erstaunlich oft selbst auf, wo das Programm etwas anderes tut, als man denkt.

> Fehlersuche: **beschreiben – nachsehen – eingrenzen**. Ändere immer nur eine Sache, und probiere nach jeder Änderung aus.

## 26.4 Fertig machen

Ein Projekt ist fertig, wenn jemand anderes es ohne deine Erklärung spielen kann. Prüfe dafür:

- Gibt es einen Titelbildschirm, der sagt, welche Taste was tut?
- Kann man nach dem Ende neu beginnen, ohne den PicoBoy neu zu starten?
- Ist nichts Wichtiges in den runden Ecken versteckt?
- Baut das Programm **ohne Warnung**?
- Stehen oben im Programm ein Kommentar mit Name und Zweck und ein paar Kommentare an den schwierigen Stellen?

**Weitergeben.** Die Datei `build/mein_programm.uf2` ist dein fertiges Programm. Jeder mit einem PicoBoy Color Plus kann sie über den Bootmodus aufspielen (BOOT halten, RES tippen, Datei kopieren), ganz ohne VS Code. Benenne sie vorher sinnvoll um, etwa in `murmellabyrinth.uf2`.

Wer auch den Quelltext haben möchte, bekommt den ganzen Projektordner ohne `build/`.

> Ein Projekt wächst gern weiter. Schreib Ideen für Extras in eine Liste, statt sie sofort einzubauen. So bleibt das Programm in jedem Moment lauffähig, und du entscheidest bewusst, was als Nächstes kommt.

## Ausprobieren: Das Gerüst füllen

Nimm `k26_geruest.c` und ersetze den Platzhalter im Zustand `SPIEL` durch das erste Stück deines eigenen Spiels: eine Figur, die sich mit dem Joystick bewegt. Was musst du beim Start eines neuen Spiels (Übergang von `TITEL` nach `SPIEL`) zurücksetzen?

## Übungsaufgaben

> **Aufgabe 1** ⭐ **Eine Idee in einem Satz** Wähle eine Idee aus 26.1 oder eine eigene. Beschreibe sie in einem Satz und schreibe auf, welche Kapitel des Buchs du dafür brauchst.

> **Aufgabe 2** ⭐⭐ **Der Plan** Zeichne für deine Idee die Zustände mit ihren Übergängen, liste die Daten auf, die das Programm braucht, und zerlege den Bau in höchstens zehn Schritte, von denen jeder ein lauffähiges Programm ergibt.

> **Aufgabe 3** ⭐⭐ **Der verschwundene Ball** Finde den Fehler in `k26_aufgabe3.c` (Abschnitt 26.3) und verbessere ihn. Warum prallt der Ball an den anderen drei Rändern richtig ab?

> **Aufgabe 4** ⭐⭐⭐ **Dein Projekt** Baue dein Projekt nach dem Plan aus Aufgabe 2. Führe ein Tagebuch: Was hast du in jedem Schritt gebaut, welcher Fehler ist aufgetreten, wie hast du ihn gefunden?

Die Lösungen stehen in Anhang C.

> **Das Wichtigste in Kürze**
>
> - Klein anfangen: eine Idee in einem Satz, Extras auf eine Liste.
> - Planen: **Zustände**, **Daten**, **Schritte**. Jeder Schritt ergibt ein lauffähiges Programm.
> - Ein Gerüst mit `enum` für die Zustände, fester Bildrate und einem einzigen `pbc_show()` pro Bild trägt fast jedes Spiel.
> - Fehlersuche: beschreiben, gezielt mit `printf` nachsehen, eingrenzen. Eine Änderung nach der anderen.
> - Fertig ist ein Projekt, wenn andere es ohne Erklärung bedienen können. Weitergeben als `.uf2`-Datei.

KAPITEL 27

# Programmieren mit KI: Vibecoding

**Worum geht's?** KI-Systeme wie ChatGPT, Claude oder Gemini schreiben auf Zuruf Programme, auch in C für den PicoBoy. Das kann eine große Hilfe sein, wenn man weiß, wie man fragt, und wenn man prüft, was zurückkommt. Du lernst, wie du einer KI die nötigen Angaben gibst, wie Agenten direkt in deinem Projektordner arbeiten, welche Fehler KI-Code typischerweise hat und was bei Datenschutz, Urheberrecht und in der Schule gilt. Zum Schluss entsteht ein kleines Spiel im Dialog mit einer KI.

## 27.1 Was ist Vibecoding?

Den Begriff **Vibecoding** hat Anfang 2025 der KI-Forscher Andrej Karpathy geprägt. Gemeint ist: Man beschreibt in Alltagssprache, was das Programm tun soll, lässt eine KI den Code schreiben, probiert das Ergebnis aus und sagt, was noch anders sein soll, ohne selbst jede Zeile zu lesen. Man programmiert nach Gefühl (*vibe*), nicht nach Plan.

Die KI-Systeme dahinter heißen **Sprachmodelle**. Sie haben aus riesigen Mengen Text gelernt, welches Wort, welche Zeile wahrscheinlich als Nächstes kommt, darunter sehr viel Programmcode. Daraus folgt, was sie gut können und wo sie danebenliegen:

| Gut | Schwach |
|---|---|
| Übliche Aufgaben, die es tausendfach gibt: Schleifen, Kollisionen, Menüs | Seltene Hardware wie den PicoBoy, den sie kaum kennen |
| Code erklären, Zeile für Zeile | Wissen, was in **deinen** Dateien steht, wenn du es nicht zeigst |
| Fehlermeldungen deuten | Merken, dass ihr Code falsch ist, ohne ihn auszuprobieren |
| Vorschläge für Struktur und Namen | Zugeben, dass sie etwas nicht wissen: Sie antworten trotzdem, oft überzeugend falsch |

Das Wichtigste daran: Eine KI schreibt **wahrscheinlichen** Code, nicht geprüften. Er sieht fast immer gut aus. Ob er stimmt, zeigen erst der Compiler und der PicoBoy.

> Du bleibst verantwortlich für dein Programm. Die KI ist eine schnelle Helferin, die sich gern irrt. Was du nicht verstehst, kannst du auch nicht reparieren.

## 27.2 Chat-Systeme: gute Fragen stellen

Bei einem **Chat-System** wie ChatGPT (OpenAI), Claude (Anthropic) oder Gemini (Google) tippst du im Browser oder in einer App eine Frage, und die KI antwortet mit Text und Code. Den Code kopierst du selbst nach `main.c`. Die KI sieht dabei nur, was du ihr schickst. Von deinem PicoBoy, deiner Vorlage und deinem Programm weiß sie nichts.

**Was in eine gute Anfrage gehört:**

1. **Worum es geht:** „C-Programm für den PicoBoy Color Plus, RP2350, Raspberry Pi Pico SDK 2.2.0. Kein Arduino."
2. **Die Bibliothek:** Füge den Inhalt von `pbc/pbc.h` ein und schreib dazu: „Benutze für Display, Tasten, LEDs und Ton nur diese Funktionen." Ohne `pbc.h` muss die KI raten, und sie rät falsch.
3. **Die Regeln des Buchs:** `main.c` beginnt mit `#include "pbc.h"` und `pbc_init()`, gezeichnet wird in den Bildpuffer, sichtbar erst mit `pbc_show()`, `main` endet nie, Display-Texte ohne Umlaute.
4. **Die Aufgabe, klein und genau:** nicht „Mach ein Spiel", sondern „Ein Rechteck unten auf dem Bildschirm soll sich mit dem Joystick nach links und rechts bewegen und nicht über den Rand hinaus."
5. **Was du zurückhaben willst:** „Das ganze `main.c`, mit kurzen Kommentaren, und erkläre danach in einfachen Worten, was neu ist."

Die Punkte 1 bis 3 musst du nicht jedes Mal tippen. Die Vorlage enthält sie schon, in der Datei `AGENTS.md` (Abschnitt 27.3). Kopiere ihren Inhalt und den von `pbc.h` an den Anfang des Gesprächs.

**In kleinen Schritten.** Lass dir nie ein ganzes Spiel auf einmal schreiben. Bitte um einen Schritt, baue ihn, probiere ihn aus, lies den Code, und erst dann kommt der nächste Schritt (wie in Kapitel 26.2). So bleibt jeder Fehler klein, und du verstehst, was entsteht.

**Fehlermeldungen vollständig mitgeben.** Baut der Code nicht, kopiere die **ganze** Meldung aus dem Terminal, ab der ersten Zeile mit `error:`, und schreib dazu, was du geändert hast. „Geht nicht" hilft der KI so wenig wie einem Menschen. Ebenso bei Fehlern zur Laufzeit: Beschreibe, was du erwartest und was stattdessen passiert, und gib Ausgaben aus dem seriellen Monitor mit.

> Frag die KI auch nach **Erklärungen**: „Was bedeutet `static` in Zeile 12?", „Warum steht hier `<` und nicht `<=`?", „Welche Fehler könnten in diesem Programm stecken?" Dafür ist sie oft besser als zum Schreiben.

## 27.3 Agenten im Projektordner

Ein **Agent** geht einen Schritt weiter als ein Chat-System: Er läuft auf deinem Computer im Projektordner, liest selbst die Dateien, ändert sie und führt Befehle aus, zum Beispiel das Bauen. Er sieht die Fehlermeldungen des Compilers selbst und versucht, sie zu beheben. Bekannte Agenten sind **Claude Code** von Anthropic und **Codex** von OpenAI. Beide gibt es für das Terminal und als Erweiterung für VS Code.

**Einrichten.** Wie man einen Agenten installiert und anmeldet, ändert sich häufig. Halte dich an die offiziellen Seiten der Hersteller (für Claude Code unter `code.claude.com/docs`, für Codex unter `developers.openai.com/codex`). Beide brauchen ein Konto beim Anbieter, und die Nutzung kann je nach Umfang kostenpflichtig sein.

Danach öffnest du den Agenten **im Ordner deines Projekts**, also dort, wo `main.c` und `CMakeLists.txt` liegen, und beschreibst die Aufgabe wie im Chat.

**Die Regeldatei der Vorlage.** Damit der Agent weiß, worum es geht, liegt in der Vorlage die Datei `AGENTS.md`. Sie ist für den Agenten geschrieben, nicht für dich, aber es lohnt sich, sie zu lesen. Darin steht in kurzen Stichpunkten:

- **Hardware:** RP2350, Display 240 × 280 mit runden Ecken, Tasten, LEDs, Lautsprecher, Sensor, und ausdrücklich: **kein Arduino**, keine `setup()` / `loop()`, kein `digitalWrite`.
- **So wird programmiert:** `main.c` mit `pbc_init()`, nur die Befehle aus `pbc/pbc.h` benutzen und keine erfinden, Bildpuffer und `pbc_show()`, `main` endet nie, ASCII-Texte, keine großen Arrays auf dem Stack (4 KB), keine Warnungen. `pbc/` und `CMakeLists.txt` nur ändern, wenn du es ausdrücklich verlangst.
- **Bauen und Aufspielen:** wie gebaut wird, und dass das Aufspielen der Mensch übernimmt.
- **Erklären:** Der Agent soll Änderungen in einfachen Worten erklären, weil am anderen Ende jemand programmieren lernt.

Codex liest `AGENTS.md` von sich aus: Laut seiner Dokumentation sucht es diese Datei im Projekt, vom obersten Ordner bis zum aktuellen. Claude Code liest dagegen Dateien namens `CLAUDE.md`. Deshalb enthält die Vorlage auch eine `CLAUDE.md`, die nur aus einer Zeile besteht:

```
@AGENTS.md
```

Das `@` vor einem Dateinamen bedeutet in einer `CLAUDE.md`: „Lies diese Datei hier mit ein.“ So stehen die Regeln nur an einer Stelle und gelten für beide Agenten. Du kannst `AGENTS.md` um eigene Regeln ergänzen, etwa „Alle Namen auf Deutsch“ oder „Benutze keine Zeiger, ich bin erst bei Kapitel 12“.

**Bauen lassen.** Am einfachsten baust du das Projekt einmal selbst in VS Code mit **Compile**. Danach gibt es den Ordner `build/`, und der Agent kann mit

```
cmake --build build
```

jederzeit neu bauen, genau wie VS Code. Ohne diesen ersten Bau braucht er den Weg aus Anhang E. Aufspielen solltest du selbst, mit **Run**: So siehst du, wann ein neues Programm auf dem PicoBoy landet.

> Ein Agent führt Befehle auf **deinem** Computer aus. Stelle ihn so ein, dass er nachfragt, bevor er Befehle ausführt oder Dateien ändert (wie das geht, steht in seiner Dokumentation). Lies, was er vorhat, bevor du zustimmst. Öffne einen Agenten nur im Projektordner, nicht in deinem ganzen Benutzerordner. Und sichere dein Projekt vorher (Ordner kopieren), dann kannst du jede Änderung zurücknehmen.

## 27.4 Prüfen, was die KI schreibt

KI-Code hat typische Fehler. Wer sie kennt, findet sie schnell.

**Arduino statt Pico SDK.** Für Mikrocontroller gibt es im Netz sehr viel Arduino-Code. Erkennt eine KI „RP2350“ und „Display“, schreibt sie gern so:

k27_ki_arduino.c

```c
// So sieht Code für den Arduino aus – nicht für das Pico SDK
#include "pbc.h"

void setup() {
    pinMode(14, OUTPUT);
}

void loop() {
    digitalWrite(14, HIGH);
    delay(500);
    digitalWrite(14, LOW);
    delay(500);
}
```

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'setup':
main.c:5:5: error: implicit declaration of function 'pinMode' [-Wimplicit-function-
  declaration]
    5 |     pinMode(14, OUTPUT);
      |     ^~~~~~
main.c:5:17: error: 'OUTPUT' undeclared (first use in this function)
    5 |     pinMode(14, OUTPUT);
      |                 ^~~~~~
main.c:5:17: note: each undeclared identifier is reported only once for each function
  it appears in
main.c: In function 'loop':
main.c:9:5: error: implicit declaration of function 'digitalWrite' [-Wimplicit-
  function-declaration]
    9 |     digitalWrite(14, HIGH);
      |     ^~~~~~~~~~~
main.c:9:22: error: 'HIGH' undeclared (first use in this function)
    9 |     digitalWrite(14, HIGH);
      |                      ^~~~
main.c:10:5: error: implicit declaration of function 'delay' [-Wimplicit-function-
  declaration]
   10 |     delay(500);
      |     ^~~~~
main.c:11:22: error: 'LOW' undeclared (first use in this function)
   11 |     digitalWrite(14, LOW);
      |                      ^~~
```

`setup`, `loop`, `pinMode`, `digitalWrite`, `delay`: Das alles gibt es im Pico SDK nicht. Außerdem fehlt `main`. Die Antwort an die KI: „Das ist Arduino-Code. Ich benutze das Pico SDK mit `pbc.h`, ohne Arduino.“

**Erfundene Funktionen.** Eine KI, die `pbc.h` nicht gesehen hat, erfindet Namen, die plausibel klingen:

k27_ki_erfunden.c

```c
// Eine Funktion, die es in pbc.h nicht gibt
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_draw_text("Hallo", 80, 120, PBC_WHITE);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

Terminal

```
main.c: In function 'main':
main.c:7:5: error: implicit declaration of function 'pbc_draw_text' [-Wimplicit-
  function-declaration]
    7 |     pbc_draw_text("Hallo", 80, 120, PBC_WHITE);
      |     ^~~~~~~~~~~~
```

`pbc_draw_text` gibt es nicht; richtig wäre `pbc_text` (oder `pbc_text12` …). Dieser Fehler ist harmlos, weil der Compiler ihn findet. Gib der KI die Meldung und `pbc.h`, dann verbessert sie ihn.

**Falsche Pins.** Das ist gefährlicher, denn der Compiler merkt es nicht. Viele Beispiele im Netz sind für das Raspberry Pi Pico 2 geschrieben, und dessen Standard-Pins passen nicht zum PicoBoy:

| Die KI nimmt oft … | beim Pico 2 | beim PicoBoy |
|---|---|---|
| eine LED an GP25 | die Platinen-LED | nichts angeschlossen |
| I2C an GP4 und GP5 | Standard-I2C | GP4 ist Joystick hoch; der Sensor hängt an GP20/GP21 |
| UART an GP0 und GP1 | Standard-Konsole | Joystick Mitte und rechts |

Die richtigen Pins stehen in Anhang D. Wenn ein Programm einen Pin direkt benutzt, vergleiche ihn dort.

**Zu viel Speicher auf dem Stack.** Die KI weiß nicht, dass der Stack des PicoBoy nur 4 KB hat (Kapitel 19.4). Ein Array mit 67 200 Pixeln als lokale Variable baut ohne Warnung, hält das Programm aber beim Start an. Aufgabe 2 zeigt so ein Programm.

**Code, der fast stimmt.** Am häufigsten sind kleine Logikfehler: `<` statt `<=`, eine vergessene Grenze, ein Wert, der beim Neustart nicht zurückgesetzt wird. Die findest du nur, indem du ausprobierst und liest.

**So prüfst du KI-Code:**

1. **Bauen.** Keine Fehler, **keine Warnungen**.
2. **Lesen.** Verstehst du jede Zeile? Wenn nicht: fragen, bevor du weitermachst. Stehen Funktionen darin, die nicht in `pbc.h` und nicht im Buch vorkommen? Feste Pin-Nummern?

3. **Ausprobieren.** Auch die Ränder: ganz links, ganz rechts, Taste gedrückt halten, sehr schnell drücken, neu starten.
4. **Ändern.** Kannst du selbst eine Kleinigkeit ändern, etwa die Geschwindigkeit? Dann hast du den Code verstanden.

## 27.5 Fair und sicher

**Datenschutz.** Was du in ein Chat-System tippst, geht an den Anbieter und kann dort gespeichert werden. Gib deshalb keine persönlichen Daten ein: keine Namen, Adressen, Fotos von dir oder anderen, keine Passwörter. Für Code ist das meist kein Problem, aber achte auf Kommentare und Texte im Programm. Viele Dienste haben in ihren Nutzungsbedingungen ein Mindestalter oder verlangen bei Jüngeren das Einverständnis der Eltern. Lies sie, bevor du ein Konto anlegst. In der Schule gilt, was die Schule erlaubt.

**Urheberrecht.** Eine KI hat aus fremdem Code gelernt und kann Stücke davon fast wörtlich wiedergeben, ohne zu sagen, woher sie stammen. Für dein eigenes Übungsprojekt ist das unkritisch. Willst du ein Programm veröffentlichen, schreib dazu, dass es mit KI-Hilfe entstanden ist, und übernimm keine großen Stücke, deren Herkunft du nicht kennst. Wie KI-erzeugter Code rechtlich zu behandeln ist, ist noch nicht in allen Punkten geklärt.

**Schule und Prüfungen.** Ob und wie du KI benutzen darfst, entscheidet deine Lehrkraft. Ein Programm von der KI als eigene Arbeit abzugeben, ist in einer Prüfung Täuschung. Und ganz praktisch: In einer Prüfung ohne KI hilft dir nur, was du selbst verstanden hast. Gib deshalb an, wofür du KI benutzt hast, und sei in der Lage, jede Zeile deines Programms zu erklären.

> Keine persönlichen Daten in KI-Dienste. KI-Hilfe angeben. Nur abgeben, was du erklären kannst.

## 27.6 Ein Spiel im Dialog mit der KI

Hier entsteht das Spiel „Ausweichen“: Unten steht ein Spieler, von oben fallen Blöcke, wer getroffen wird, verliert. Die Antworten der KI sind **Beispiele**: gekürzt und so formuliert, wie eine KI typischerweise antwortet. Echte Antworten sehen jedes Mal etwas anders aus. Der Code jedes Schritts aber ist ein echtes, geprüftes Programm.

Am Anfang des Gesprächs stehen, wie in 27.2 beschrieben, der Inhalt von `AGENTS.md` und von `pbc.h`.

**Schritt 1.**

> **Du:** Ich möchte ein Spiel „Ausweichen“ bauen, in kleinen Schritten. Erster Schritt: Unten auf dem Bildschirm (y = 230) ist ein Rechteck, 30 × 12 Pixel, türkis. Es bewegt sich mit Joystick links und rechts und bleibt mindestens 25 Pixel vom Rand weg (runde Ecken). Bitte das ganze `main.c`.
>
> **KI** *(Beispiel, gekürzt)*: Hier ist der erste Schritt. Die Größe des Spielers steht in Konstanten, damit du sie leicht ändern kannst. `pbc_pressed_left()` ist wahr, solange die Taste gedrückt ist; so bewegt sich der Spieler, solange du hältst …

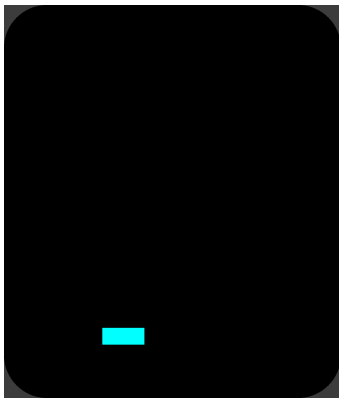
k27_schritt1.c

```c
// Ausweichen, Schritt 1: ein Spieler, der sich links und rechts bewegt
#include "pbc.h"

#define SPIELER_Y      230
#define SPIELER_BREITE 30
#define SPIELER_HOEHE  12

int main(void) {
    pbc_init();
    int spieler_x = 105;
    while (true) {
        if (pbc_pressed_left() && spieler_x > 25) {
            spieler_x -= 5;
        }
        if (pbc_pressed_right() && spieler_x < 215 - SPIELER_BREITE) {
            spieler_x += 5;
        }
        pbc_clear(PBC_BLACK);
        pbc_fill_rect(spieler_x, SPIELER_Y, SPIELER_BREITE, SPIELER_HOEHE,
                      PBC_CYAN);
        pbc_show();
        sleep_ms(20);
    }
}
```

Ausprobieren und lesen: Was passiert bei `spieler_x > 25`, wenn man links hält? Bleibt der Spieler wirklich im Bild? Das Bild zeigt ihn nach einem kurzen Druck nach links.



### Schritt 2.

> **Du:** Funktioniert. Jetzt soll von oben ein roter Block (20 × 20) fallen, 4 Pixel pro Bild. Ist er unten raus, beginnt er oben an einer zufälligen Stelle neu.
>
> **KI** *(Beispiel, gekürzt)*: Dafür brauchst du zwei neue Variablen für die Position des Blocks. Für die zufällige Stelle verwende ich `pbc_draw_random_x()` …

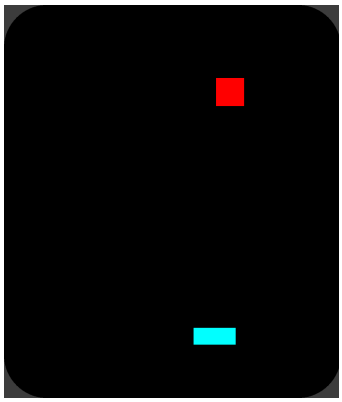
Eine solche Funktion steht nicht in `pbc.h`. Der Compiler meldet `implicit declaration of function` (wie oben bei `pbc_draw_text`).

> **Du:** Fehler beim Bauen: `implicit declaration of function 'pbc_draw_random_x'`. In `pbc.h` gibt es nur `pbc_random(min, max)`.
>
> **KI** *(Beispiel, gekürzt)*: Du hast recht, die Funktion gibt es nicht. Mit `pbc_random(25, 215 - BLOCK_GROESSE)` bekommst du eine Stelle, an der der Block ganz sichtbar ist …

k27_schritt2.c

```
10      pbc_init();
11      int spieler_x = 105;
12      int block_x = pbc_random(25, 215 - BLOCK_GROESSE);
13      int block_y = 0;
14      while (true) {
15          if (pbc_pressed_left() && spieler_x > 25) {
16              spieler_x -= 5;
17          }
18          if (pbc_pressed_right() && spieler_x < 215 - SPIELER_BREITE) {
19              spieler_x += 5;
20          }
21          block_y += 4;
22          if (block_y > PBC_HEIGHT) {          // unten raus: neu von oben
23              block_y = 0;
24              block_x = pbc_random(25, 215 - BLOCK_GROESSE);
25          }
26          pbc_clear(PBC_BLACK);
27          pbc_fill_rect(spieler_x, SPIELER_Y, SPIELER_BREITE, SPIELER_HOEHE,
28                        PBC_CYAN);
29          pbc_fill_rect(block_x, block_y, BLOCK_GROESSE, BLOCK_GROESSE, PBC_RED);
30          pbc_show();
31          sleep_ms(20);
32      }
33  }
```

Neu sind die Zeilen 12, 13 und 21 bis 25. Die Zeilen davor sind dieselben wie in Schritt 1, mit einer zusätzlichen Konstanten `BLOCK_GROESSE`.



### Schritt 3.

> **Du:** Jetzt: Trifft der Block den Spieler, ist das Spiel vorbei („Game over“, tiefer Ton), mit A geht es neu los. Für jeden Block, dem man ausweicht, gibt es einen Punkt, oben in der Mitte angezeigt. Alle 5 Punkte etwas schneller.
>
> **KI** *(Beispiel, gekürzt)*: Für den Treffer prüfe ich, ob sich die Rechtecke von Spieler und Block überlappen. Dafür schreibe ich eine kleine Funktion `ueberlappen`. Eine Variable `vorbei` merkt sich, ob das Spiel zu Ende ist …

k27_schritt3.c

```c
static bool ueberlappen(int x1, int y1, int b1, int h1,
                        int x2, int y2, int b2, int h2) {
    return x1 < x2 + b2 && x2 < x1 + b1 && y1 < y2 + h2 && y2 < y1 + h1;
}
```

k27_schritt3.c

```c
    while (true) {
        if (vorbei) {
            if (pbc_was_pressed_a()) {       // neues Spiel
                vorbei = false;
                punkte = 0;
                tempo = 4;
                block_y = 0;
            }
            sleep_ms(20);
            continue;
        }
        if (pbc_pressed_left() && spieler_x > 25) {
            spieler_x -= 5;
        }
        if (pbc_pressed_right() && spieler_x < 215 - SPIELER_BREITE) {
            spieler_x += 5;
        }
        block_y += tempo;
        if (block_y > PBC_HEIGHT) {
            block_y = 0;
            block_x = pbc_random(25, 215 - BLOCK_GROESSE);
            punkte++;
            if (punkte % 5 == 0) {
                tempo++;                     // alle 5 Punkte schneller
            }
        }
        if (ueberlappen(spieler_x, SPIELER_Y, SPIELER_BREITE, SPIELER_HOEHE,
                        block_x, block_y, BLOCK_GROESSE, BLOCK_GROESSE)) {
            vorbei = true;
            pbc_tone(150, 300);
        }
```




Die Zeilen 14 bis 21 legen die Variablen an, dazu kommen `tempo`, `punkte` und `vorbei`. Ab Zeile 54 wird gezeichnet wie in Schritt 2, dazu die Punkte und, wenn das Spiel vorbei ist, „Game over“. Lies den Code, bevor du weitermachst:

- `ueberlappen` (Zeilen 9 bis 12) ist der Rechteck-Test aus Kapitel 16.1.
- Zeilen 23 bis 32: Ist das Spiel vorbei, wartet die Schleife nur auf A und setzt dann Punkte, Tempo und Block zurück. Wird hier alles zurückgesetzt, was zurückgesetzt werden muss? Der Spieler bleibt, wo er war. Das ist in Ordnung.
- `continue` in Zeile 31 springt sofort zum nächsten Durchlauf der Schleife (Kapitel 8.4). Gezeichnet wird in diesem Zustand nicht mehr: Das letzte Bild mit „Game over“ bleibt stehen.

Das ganze Programm hat 68 Zeilen und verwendet nur, was du aus dem Buch kennst. Damit ist es **dein** Programm: Du kannst es erklären, ändern und erweitern, mit oder ohne KI.

## Ausprobieren: Deine Regeln für die KI

Öffne `AGENTS.md` in der Vorlage und lies sie ganz. Welche Regel hättest du vor diesem Buch nicht verstanden? Ergänze eine eigene Regel, die zu dir passt, und probiere aus, ob die KI sich daran hält.

## Übungsaufgaben

**Aufgabe 1 ⭐ Eine gute Anfrage** Schreibe eine Anfrage an eine KI für dieses Programm: Die drei Status-LEDs sollen eine Ampel sein (rot, rot-gelb, grün, gelb, rot …), und auf dem Display steht dazu „Halt“, „Achtung“ oder „Fahren“. Was gehört alles in die Anfrage?

**Aufgabe 2 ⭐⭐ Was stimmt hier nicht?** Dieses Programm stammt von einer KI. Es baut ohne Warnung, aber auf dem PicoBoy erscheint kein Bild, und die versprochene LED blinkt nicht. Finde die beiden Fehler.

k27_aufgabe2.c

```c
// Aufgabe 27.2: Von einer KI. Baut ohne Warnung. Was ist falsch?
#include "pbc.h"

int main(void) {
    pbc_init();
    uint16_t hintergrund[PBC_WIDTH * PBC_HEIGHT];   // ein Muster
    for (int i = 0; i < PBC_WIDTH * PBC_HEIGHT; i++) {
        hintergrund[i] = pbc_rgb(0, 0, i % 256);
    }
    gpio_init(25);
    gpio_set_dir(25, GPIO_OUT);
```

```
    while (true) {
        for (int y = 0; y < PBC_HEIGHT; y++) {
            for (int x = 0; x < PBC_WIDTH; x++) {
                pbc_pixel(x, y, hintergrund[y * PBC_WIDTH + x]);
            }
        }
        pbc_show();
        gpio_put(25, 1);            // LED an: Bild ist fertig
        sleep_ms(500);
        gpio_put(25, 0);
        sleep_ms(500);
    }
}
```

**Aufgabe 3 ⭐⭐ Zwei Blöcke** Erweitere „Ausweichen“ mit oder ohne KI so, dass zwei Blöcke gleichzeitig fallen, der zweite eine halbe Bildschirmhöhe hinter dem ersten.

**Aufgabe 4 ⭐⭐⭐ Den Agenten prüfen** Bitte einen Agenten (oder ein Chat-System), „Ausweichen“ um einen Highscore zu erweitern, der im Flash gespeichert wird (Kapitel 25). Prüfe das Ergebnis mit der Liste aus Kapitel 25.3: Welcher Sektor? Wird nur bei einem neuen Rekord gespeichert? Gibt es Kennung und Prüfsumme? Wird `flash_safe_execute` benutzt?

Die Lösungen stehen in Anhang C.

**Das Wichtigste in Kürze**

- **Vibecoding**: beschreiben, KI schreiben lassen, ausprobieren. KI-Code ist wahrscheinlich, nicht geprüft.
- Gute Anfrage: Gerät und SDK nennen, **`pbc.h`** und die Regeln aus **`AGENTS.md`** mitgeben, kleine Schritte, Fehlermeldungen vollständig.
- **Agenten** (Claude Code, Codex) arbeiten im Projektordner; sie lesen `AGENTS.md` (Claude Code über `CLAUDE.md` mit `@AGENTS.md`). Vor jeder Zustimmung lesen, was sie tun wollen.
- Typische KI-Fehler: Arduino-Code, erfundene Funktionen, falsche Pins (Anhang D), große Arrays auf dem Stack, kleine Logikfehler.
- Bauen ohne Warnung, lesen, ausprobieren, selbst ändern können.
- Keine persönlichen Daten eingeben, KI-Hilfe angeben, nur abgeben, was du erklären kannst.

ANHANG A

# Die Befehle der Buchbibliothek `pbc`

Alle Befehle stehen in der Datei `pbc/pbc.h` deiner Vorlage. Ein einziges `#include "pbc.h"` genügt; es bindet auch `stdbool.h`, `stdint.h`, `stdio.h`, `stdlib.h`, `string.h` und `pico/stdlib.h` mit ein. Die Spalte „Kap." nennt das Kapitel, in dem der Befehl eingeführt wird. Wie die Bibliothek innen aufgebaut ist, zeigt Kapitel 20.3.

## Start

| Befehl | Was er tut | Kap. |
|---|---|---|
| `void pbc_init(void)` | richtet Display, Tasten, LEDs, Ton, Sensor, Batterie-Messung und die Konsole ein. Muss als Erstes in `main` stehen. Wartet bis zu 2 s auf einen seriellen Monitor. | 3 |

## Display

Alles wird zuerst in den **Bildpuffer** gezeichnet. Sichtbar wird es erst mit `pbc_show()`. Koordinaten: (0, 0) oben links, x nach rechts (0–239), y nach unten (0–279). Was außerhalb liegt, wird einfach weggelassen.

| Befehl | Was er tut | Kap. |
|---|---|---|
| `PBC_WIDTH`, `PBC_HEIGHT` | Breite 240 und Höhe 280 des Displays | 6 |
| `void pbc_show(void)` | bringt den Bildpuffer aufs Display (gut 31 ms) | 3, 14 |
| `void pbc_clear(uint16_t color)` | füllt das ganze Bild mit einer Farbe | 3 |
| `void pbc_pixel(int x, int y, uint16_t color)` | setzt einen Punkt | 6 |
| `uint16_t pbc_get_pixel(int x, int y)` | liest die Farbe eines Punkts aus dem Bildpuffer; außerhalb 0 | – |
| `void pbc_line(int x1, int y1, int x2, int y2, uint16_t color)` | Linie von (x1, y1) nach (x2, y2) | 6 |
| `void pbc_hline(int x, int y, int length, uint16_t color)` | waagrechte Linie, `length` Pixel nach rechts | 6 |
| `void pbc_vline(int x, int y, int length, uint16_t color)` | senkrechte Linie, `length` Pixel nach unten | 6 |
| `void pbc_rect(int x, int y, int w, int h, uint16_t color)` | Rechteck-Umriss, (x, y) = linke obere Ecke | 6 |

| Befehl | Was er tut | Kap. |
|---|---|---|
| `void pbc_fill_rect(int x, int y, int w, int h, uint16_t color)` | gefülltes Rechteck | 6 |
| `void pbc_circle(int mx, int my, int r, uint16_t color)` | Kreis-Umriss um den Mittelpunkt (mx, my) | 6 |
| `void pbc_fill_circle(int mx, int my, int r, uint16_t color)` | gefüllter Kreis | 6 |
| `void pbc_ellipse(int mx, int my, int rx, int ry, uint16_t color)` | Ellipse-Umriss mit den Halbachsen rx, ry | 6 |
| `void pbc_fill_ellipse(int mx, int my, int rx, int ry, uint16_t color)` | gefüllte Ellipse | 6 |
| `void pbc_backlight(int value)` | Helligkeit des Displays, 0 (aus) bis 255 (voll) | 21 |

## Farben

| Befehl | Was er tut | Kap. |
|---|---|---|
| `uint16_t pbc_rgb(int r, int g, int b)` | mischt eine Farbe aus Rot, Grün und Blau (je 0–255) im Format RGB565; Werte außerhalb werden begrenzt | 6 |
| `PBC_BLACK`, `PBC_WHITE`, `PBC_RED`, `PBC_GREEN`, `PBC_BLUE`, `PBC_YELLOW`, `PBC_CYAN`, `PBC_MAGENTA`, `PBC_ORANGE`, `PBC_PURPLE`, `PBC_GRAY` | elf fertige Farben | 3 |

## Text

(x, y) ist die linke obere Ecke. Nur die Zeichen der englischen Tastatur (ASCII 32–126); Umlaute erscheinen als `?`. Die Schrift wird ohne Hintergrund gezeichnet. Der Rückgabewert ist das x direkt hinter dem Text.

| Befehl | Was er tut | Kap. |
|---|---|---|
| `int pbc_text(const char *s, int x, int y, uint16_t color)` | Schrift 8 × 8 Pixel, feste Breite | 3 |
| `int pbc_text12(…)`, `int pbc_text18(…)`, `int pbc_text24(…)` | Proportionalschriften; Zeilenhöhe 16, 24 und 32 Pixel; Parameter wie `pbc_text` | 3 |
| `int pbc_text_width(const char *s)` | Breite eines Texts in Pixeln für `pbc_text` (8 je Zeichen) | 6 |
| `int pbc_text12_width(…)`, `int pbc_text18_width(…)`, `int pbc_text24_width(…)` | Breite für die drei Proportionalschriften, zum Beispiel zum Zentrieren | 6 |

## Bilder

| Befehl | Was er tut | Kap. |
|---|---|---|
| `pbc_image` | Struktur für ein Bild: `width`, `height`, `pixels` (die Farben, Zeile für Zeile), `mask` (1 = sichtbar, 0 = durchsichtig; `NULL` = alles sichtbar). Erzeugt mit dem Umwandlungswerkzeug `bild2c.html`. | 15 |
| `void pbc_draw_image(const pbc_image *image, int x, int y)` | zeichnet ein Bild mit der linken oberen Ecke bei (x, y); Aufruf mit `&bild` | 15 |

## Tasten und Joystick

| Befehl | Was er tut | Kap. |
|---|---|---|
| `bool pbc_pressed_up(void)`, `…_down`, `…_left`, `…_right`, `…_center`, `…_a`, `…_b` | `true`, **solange** die Taste gedrückt ist; `center` = Joystick hineindrücken | 9 |
| `bool pbc_any_pressed(void)` | `true`, solange irgendeine Taste gedrückt ist | 9 |
| `bool pbc_was_pressed_up(void)`, `…_down`, `…_left`, `…_right`, `…_center`, `…_a`, `…_b` | `true` genau **einmal** pro Druck; die Bibliothek merkt sich jeden Druck, auch wenn das Programm gerade beschäftigt war (entprellt mit 50 ms) | 9 |
| `void pbc_wait_for_button(void)` | wartet, bis eine Taste gedrückt und wieder losgelassen wurde | 9 |

## LEDs

| Befehl | Was er tut | Kap. |
|---|---|---|
| `void pbc_led_red(bool on)`, `pbc_led_yellow`, `pbc_led_green` | schaltet eine der drei Status-LEDs an (`true`) oder aus (`false`) | 10 |
| `void pbc_leds_off(void)` | schaltet alle drei Status-LEDs aus | 10 |
| `void pbc_rgb_led(int r, int g, int b)` | Farbe der RGB-LED, je 0–255 | 10 |
| `void pbc_rgb_led_off(void)` | schaltet die RGB-LED aus | 10 |

## Ton

| Befehl | Was er tut | Kap. |
|---|---|---|
| `void pbc_tone(int frequency, int ms)` | spielt einen Ton mit `frequency` Hertz (20–20 000) für `ms` Millisekunden; `ms = 0`: bis `pbc_tone_off()`. **Wartet nicht**, das Programm läuft sofort weiter. | 10 |
| `void pbc_tone_off(void)` | beendet den Ton | 10 |

## Neigungssensor

Flach auf dem Tisch: x = 0, y = 0, z = 1 g. +x zeigt nach rechts, +y nach unten (zum Joystick), +z aus dem Display heraus.

| Befehl | Was er tut | Kap. |
|---|---|---|
| `float pbc_accel_x(void)`, `pbc_accel_y`, `pbc_accel_z` | Beschleunigung in m/s² (1 g = 9,81 m/s²) | 17 |
| `float pbc_accel_x_g(void)`, `pbc_accel_y_g`, `pbc_accel_z_g` | dasselbe in g | 17, 22 |

## Batterie

| Befehl | Was er tut | Kap. |
|---|---|---|
| `float pbc_battery_voltage(void)` | Spannung in Volt, Mittelwert aus 16 Messungen; am USB etwa 5 V | 21 |
| `bool pbc_is_on_usb(void)` | `true`, wenn die gemessene Spannung über 4,0 V liegt | 21 |

## Zeit, Zufall, Konsole

| Befehl | Was er tut | Kap. |
|---|---|---|
| `uint32_t pbc_millis(void)` | Millisekunden seit dem Start | 7 |
| `int pbc_random(int min, int max)` | zufällige ganze Zahl von `min` bis `max`, beide eingeschlossen | 5 |
| `void pbc_input(const char *prompt, char *text, int size)` | zeigt `prompt` in der Konsole, wartet auf eine Zeile mit Enter und legt sie in `text` ab (höchstens `size - 1` Zeichen); Aufruf meist mit `sizeof text` | 4 |
| `int pbc_input_int(const char *prompt)` | liest eine ganze Zahl | 4 |
| `float pbc_input_float(const char *prompt)` | liest eine Kommazahl; Punkt oder Komma | 4 |

## Aus dem Pico SDK

Diese Funktionen gehören nicht zu `pbc`, sondern zum SDK. Sie kommen über `pbc.h` mit und funktionieren auch in den Prüfläufen auf dem PC.

| Befehl | Was er tut | Kap. |
|---|---|---|
| `void sleep_ms(uint32_t ms)` | wartet `ms` Millisekunden | 3, 7 |
| `void sleep_us(uint64_t us)` | wartet `us` Mikrosekunden | 14, 21 |
| `uint32_t time_us_32(void)`, `uint64_t time_us_64(void)` | Mikrosekunden seit dem Start; die 32-Bit-Fassung läuft nach gut 71 Minuten über und beginnt wieder bei 0 | 21 |
| `printf(…)`, `snprintf(…)` | Ausgabe in die Konsole bzw. in ein `char`-Array (Standard-C) | 4, 6 |

Die Funktionen für die Hardware ohne `pbc` (GPIO, PWM, ADC, I2C …) stellen die Kapitel 21 bis 25 vor.

ANHANG B

# Fehlermeldungen des Compilers und was sie bedeuten

Hier stehen die Meldungen, die dir beim Programmieren mit dem PicoBoy am häufigsten begegnen. Jede wurde mit einem kleinen, absichtlich falschen Programm erzeugt (`kB_fehler_….c` und `kB_warnung_….c` im Ordner `programme`). Über jedem Kasten steht der Name dieses Programms. Die fehlerhafte Zeile zeigt der Compiler in der Meldung selbst.

**So liest du eine Meldung** (ausführlich in Kapitel 3.5):

- `main.c:5:25:` heißt Datei `main.c`, Zeile 5, Spalte 25.
- `error:` ist ein **Fehler**: Es entsteht kein Programm. `warning:` ist eine **Warnung**: Das Programm wird gebaut, tut aber wahrscheinlich nicht, was du willst. `note:` ist ein Hinweis, der zur Meldung darüber gehört.
- Am Ende steht oft in eckigen Klammern der Name der Prüfung, etwa `[-Wformat=]`. Mit diesem Namen findest du im Internet mehr dazu.
- Verbessere immer den **ersten** Fehler und baue neu. Spätere Meldungen sind oft nur Folgen davon.
- Der Fehler liegt in der genannten Zeile **oder kurz davor**.

Im Terminal von VS Code steht vor `main.c` noch der Pfad deines Projektordners. Meldungen zu `pbc.h` zeigen den vollständigen Pfad zu dieser Datei; im Buch ist es der Pfad auf dem Rechner, auf dem die Beispiele gebaut wurden.

## B.1 Fehler im Aufbau des Programms

kB_fehler_semikolon.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:5:25: error: expected ';' before 'pbc_show'
    5 |     pbc_clear(PBC_BLACK)
      |                         ^
      |                         ;
    6 |     pbc_show();
      |     ~~~~~~~~
```

**`expected ';' before …`** – Am Ende einer Anweisung fehlt das Semikolon. Der Compiler bemerkt es erst beim nächsten Wort, deshalb nennt er die Stelle direkt hinter der Anweisung davor. Abhilfe: `;` ergänzen, wie der Compiler es unter der Zeile vorschlägt.

kB_fehler_klammer_auf.c

```
main.c: In function 'main':
main.c:5:53: error: expected ')' before ';' token
    5 |     pbc_fill_rect(10, 10, 50, 50, pbc_rgb(255, 0, 0);
      |                   ~                                 ^
      |                                                     )
main.c:9:6: error: expected ';' before '}' token
    9 |     }
      |      ^
      |      ;
   10 | }
      | ~
```

**`expected ')' before ';' token`** – Eine runde Klammer ist nicht geschlossen. Die Tilde `~` unter der Zeile markiert die geöffnete Klammer, der Pfeil `^` die Stelle, an der die schließende fehlt. Zähle die Klammern: Jede `(` braucht eine `)`. Die zweite Meldung ist eine Folge der ersten.

kB_fehler_klammer_zu.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:11:1: error: expected declaration or statement at end of input
   11 | }
      | ^
```

**`expected declaration or statement at end of input`** – Die Datei ist zu Ende, aber ein Block ist noch offen: Eine `}` fehlt. Gemeldet wird die letzte Zeile, der Fehler liegt aber weiter oben. Hier fehlt die `}` der `for`-Schleife. Die Einrückung, die VS Code beim Tippen anlegt, hilft beim Suchen: Wo sie nicht mehr zur Klammer passt, fehlt etwas.

kB_fehler_anfuehrung.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:5:12: warning: missing terminating " character
    5 |     printf("Hallo\n);
      |            ^
main.c:5:12: error: missing terminating " character
    5 |     printf("Hallo\n);
      |            ^~~~~~~~~
main.c:6:5: error: expected expression before 'while'
    6 |     while (true) {
      |     ^~~~~
main.c:8:6: error: expected ';' before '}' token
    8 |     }
```

```
      |      ^
      |      ;
    9 | }
      | ~
```

**`missing terminating " character`** – Ein Text in Anführungszeichen ist nicht geschlossen. Alles bis zum Zeilenende gilt dann als Text, und die Meldungen danach sind Folgefehler. Abhilfe: das schließende `"` ergänzen, hier vor der `)`: `printf("Hallo\n");`.

kB_fehler_else.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:6:15: warning: suggest braces around empty body in an 'if' statement [-Wempty-
  body]
    6 |     if (x > 3); {
      |               ^
main.c:6:5: warning: this 'if' clause does not guard... [-Wmisleading-indentation]
    6 |     if (x > 3); {
      |     ^~
main.c:6:17: note: ...this statement, but the latter is misleadingly indented as if
  it were guarded by the 'if'
    6 |     if (x > 3); {
      |                 ^
main.c:8:7: error: 'else' without a previous 'if'
    8 |     } else {
      |       ^~~~
```

**`'else' without a previous 'if'`** – Zu einem `else` gibt es kein passendes `if`. Hier steht ein Semikolon direkt hinter `if (x > 3)`. Für C ist das `if` damit schon zu Ende (es tut „nichts“), der Block danach wird immer ausgeführt, und das `else` hängt in der Luft. Die beiden Warnungen davor zeigen genau darauf. Abhilfe: das Semikolon hinter der Bedingung löschen.

kB_fehler_break.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:7:9: error: break statement not within loop or switch
    7 |         break;
      |         ^~~~~
```

**`break statement not within loop or switch`** – `break` beendet eine Schleife oder einen `switch` (Kapitel 8.4), nicht ein `if`. Steht es außerhalb, weiß der Compiler nicht, wohin. Prüfe, ob es in der richtigen Schleife steht, oder ob du eigentlich `return` meinst.

kB_fehler_include.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c:2:10: fatal error: figur.h: No such file or directory
    2 | #include "figur.h"
      |          ^~~~~~~~~
compilation terminated.
```

**`fatal error: … No such file or directory`** – Eine Datei aus `#include` gibt es nicht. Prüfe den Namen (Groß- und Kleinschreibung!) und ob die Datei im Projektordner liegt. *Fatal* heißt: Der Compiler bricht sofort ab und prüft nichts weiter.

## B.2 Unbekannte und doppelte Namen

kB_fehler_name.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:6:5: error: implicit declaration of function 'pbc_txt'; did you mean
  'pbc_text'? [-Wimplicit-function-declaration]
    6 |     pbc_txt("Hallo", 10, 10, PBC_WHITE);
      |     ^~~~~~
      |     pbc_text
```

**`implicit declaration of function …`** – Eine Funktion mit diesem Namen ist unbekannt. Fast immer ein Tippfehler; oft schlägt der Compiler mit *did you mean* den richtigen Namen vor. Andere Ursachen: `#include` fehlt (Kapitel 20.1), oder die Funktion steht weiter unten in der Datei und hat keinen Prototyp (Kapitel 11.3).

kB_fehler_undeclared.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:6:14: error: 'pukte' undeclared (first use in this function); did you mean
  'punkte'?
    6 |     punkte = pukte + 10;
      |              ^~~~~
      |              punkte
main.c:6:14: note: each undeclared identifier is reported only once for each function
  it appears in
```

**`'…' undeclared (first use in this function)`** – Eine Variable oder Konstante ist unbekannt: Tippfehler, falsche Groß- und Kleinschreibung, oder die Variable ist an dieser Stelle nicht sichtbar, weil sie in einem anderen Block angelegt wurde (Kapitel 11.4). Die `note:`-Zeile sagt nur, dass jeder unbekannte Name pro Funktion einmal gemeldet wird.

kB_fehler_doppelt.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:6:9: error: redefinition of 'leben'
    6 |     int leben = 5;
      |         ^~~~~
main.c:5:9: note: previous definition of 'leben' with type 'int'
    5 |     int leben = 3;
      |         ^~~~~
```

**`redefinition of '…'`** – Eine Variable wird im selben Block zweimal angelegt. Die `note:` zeigt die erste Stelle. Willst du nur einen neuen Wert speichern, lass beim zweiten Mal den Typ weg: `leben = 5;`.

kB_fehler_typen.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c:5:7: error: conflicting types for 'doppelt'; have 'float(float)'
    5 | float doppelt(float zahl) {
      |       ^~~~~~~
main.c:3:5: note: previous declaration of 'doppelt' with type 'int(int)'
    3 | int doppelt(int zahl);
      |     ^~~~~~~
main.c: In function 'main':
main.c:11:14: warning: format '%d' expects argument of type 'int', but argument 2 has
  type 'double' [-Wformat=]
   11 |     printf("%d\n", doppelt(21));
      |             ~^     ~~~~~~~~~~
      |              |     |
      |              int   double
      |             %f
```

**`conflicting types for '…'`** – Prototyp und Funktion passen nicht zusammen: Hier kündigt der Prototyp `int doppelt(int)` an, definiert ist aber `float doppelt(float)`. Rückgabetyp und Parameter müssen genau gleich sein. Die Warnung danach ist eine Folge: `printf` bekommt jetzt eine Kommazahl.

## B.3 Falsche Typen

kB_fehler_zeiger.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:6:14: error: passing argument 1 of 'pbc_text' makes pointer from integer
  without a cast [-Wint-conversion]
    6 |     pbc_text(punkte, 10, 10, PBC_WHITE);
      |              ^~~~~~
      |              |
      |              int
In file included from main.c:1:
pbc/pbc.h:75:26: note: expected 'const char *' but argument is of type 'int'
   75 | int pbc_text(const char *s, int x, int y, uint16_t color);     // 8 × 8,
     feste Breite
      |              ~~~~~~~~~~~~~^
```

**`passing argument … makes pointer from integer without a cast`** – Die Funktion erwartet einen Zeiger, du übergibst eine Zahl. Hier will `pbc_text` einen Text (`const char *`), bekommt aber die Zahl `punkte`. Zahlen wandelst du mit `snprintf` in Text um (Kapitel 6.3). Dieselbe Meldung erscheint, wenn bei einem Zeiger-Parameter das `&` fehlt (Kapitel 19.2).

kB_fehler_argumente.c

```
main.c: In function 'main':
main.c:5:5: error: too few arguments to function 'pbc_fill_rect'
    5 |     pbc_fill_rect(10, 10, 50, PBC_RED);
      |     ^~~~~~~~~~~~
In file included from main.c:1:
pbc/pbc.h:66:6: note: declared here
   66 | void pbc_fill_rect(int x, int y, int w, int h, uint16_t color);  // gefüllt
      |      ^~~~~~~~~~~~
```

**`too few arguments to function '…'`** (oder `too many`) – Die Zahl der Argumente stimmt nicht. Die `note:` zeigt, wie die Funktion deklariert ist: `pbc_fill_rect` braucht x, y, Breite, Höhe **und** Farbe.

kB_fehler_array.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:6:10: error: assignment to expression with array type
    6 |     name = "Mia";
      |          ^
```

**`assignment to expression with array type`** – Einem Array kann man nicht mit `=` etwas zuweisen, auch keinen Text. Texte kopierst du mit `snprintf(name, sizeof name, "Mia");` (Kapitel 12.4). Beim Anlegen geht es dagegen: `char name[20] = "Mia";`.

kB_fehler_const.c

```
main.c: In function 'main':
main.c:6:15: error: assignment of read-only variable 'MAX_LEBEN'
    6 |     MAX_LEBEN = 5;
      |               ^
```

**`assignment of read-only variable '…'`** – Eine `const`-Variable lässt sich nach dem Anlegen nicht mehr ändern (Kapitel 5.5). Entweder das `const` weglassen oder die Zuweisung.

kB_fehler_void.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'zeige':
main.c:5:12: error: 'return' with a value, in function returning void [-Wreturn-
  mismatch]
    5 |     return wert;
      |            ^~~~
main.c:3:6: note: declared here
    3 | void zeige(int wert) {
      |      ^~~~
```

**`'return' with a value, in function returning void`** – Eine `void`-Funktion gibt nichts zurück. Soll sie einen Wert liefern, schreib den Typ statt `void` vor den Namen (`int zeige(int wert)`), sonst nur `return;`.

## B.4 Fehler beim Binden (Linker)

Diese Meldungen kommen nicht vom Compiler, sondern vom **Linker** `ld`, der am Ende alle Teile zusammenfügt (Kapitel 20.1). Sie beginnen deshalb mit dem Pfad zum Linker und enden mit `ld returned 1 exit status`.

kB_fehler_undefined.c

```
ld: main.c.o: in function `main':
main.c:7:(.text.startup.main+0xa): undefined reference to `zeichne_stern'
ld: (zeichne_stern): Unknown destination type (ARM/Thumb) in main.c.o
main.c:7:(.text.startup.main+0xa): dangerous relocation: unsupported relocation
collect2: error: ld returned 1 exit status
```

**`undefined reference to …`** – Die Funktion ist angekündigt (Prototyp), aber nirgends definiert. Ursachen: Die `.c`-Datei mit der Funktion steht nicht in `add_executable` (Kapitel 20.2), der Name ist in der Definition anders geschrieben, oder die Funktion wurde noch gar nicht geschrieben. Die Zeilen mit *dangerous relocation* sind Folgen derselben Ursache.

kB_fehler_ram.c

```
ld: kB_fehler_ram.elf section `.bss' will not fit in region `RAM'
ld: section .stack_dummy VMA [20081000,20081fff] overlaps section .bss VMA
  [200012d8,200b55b7]
ld: region `RAM' overflowed by 220600 bytes
collect2: error: ld returned 1 exit status
```

**`region `RAM' overflowed by … bytes`** – Die globalen und `static`-Variablen passen nicht ins RAM (512 KB). Abhilfe: große Tabellen mit `const` in den Flash legen, kleinere Typen wählen, weniger Puffer (Kapitel 19.5).

Eine weitere Linker-Meldung ist **`multiple definition of …`**: Eine Funktion oder globale Variable ist in zwei `.c`-Dateien definiert, oft weil die Definition in einer Header-Datei steht. Ein Beispiel zeigt Kapitel 20.1.

## B.5 Warnungen

Der Compiler der Buchvorlage läuft mit `-Wall -Wextra` und warnt deshalb vor vielen verdächtigen Stellen. Eine Warnung hält das Bauen nicht auf, aber fast jede zeigt einen echten Fehler. Baue so, dass **keine** Warnung übrig bleibt.

kB_warnung_zuweisung.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
Linking CXX executable kB_warnung_zuweisung.elf
Built target kB_warnung_zuweisung
main.c: In function 'main':
main.c:6:9: warning: suggest parentheses around assignment used as truth value [-
  Wparentheses]
    6 |     if (x = 6) {
      |         ^
```

**`suggest parentheses around assignment used as truth value`** – In der Bedingung steht `=` statt `==` (Kapitel 8.5). `x = 6` **speichert** 6 in `x`, und die Bedingung ist immer wahr. Gemeint ist `x == 6`.

kB_warnung_format.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
Linking CXX executable kB_warnung_format.elf
Built target kB_warnung_format
main.c: In function 'main':
main.c:6:14: warning: format '%d' expects argument of type 'int', but argument 2 has
  type 'double' [-Wformat=]
    6 |     printf("%d Euro\n", preis);
```

```
      |             ~^          ~~~~~
      |              |          |
      |              int        double
      |             %f
```

**`format '%d' expects argument of type 'int', but …`** – Platzhalter und Wert passen nicht zusammen (Kapitel 4.3). Darunter steht, welcher Platzhalter richtig wäre, hier `%f`.

kB_warnung_unbenutzt.c

```
main.c: In function 'main':
main.c:6:9: warning: unused variable 'leben' [-Wunused-variable]
    6 |     int leben = 3;
      |         ^~~~~
```

**`unused variable '…'`** – Eine Variable wird angelegt, aber nie benutzt. Entweder sie ist überflüssig, oder du hast vergessen, sie zu verwenden, oder an ihrer Stelle versehentlich eine andere benutzt.

kB_warnung_parameter.c

```
main.c: In function 'zeichne_ball':
main.c:3:37: warning: unused parameter 'r' [-Wunused-parameter]
    3 | void zeichne_ball(int x, int y, int r) {
      |                                 ~~~~^
```

**`unused parameter '…'`** – Ein Parameter wird in der Funktion nicht benutzt. Hier zeichnet `zeichne_ball` immer mit Radius 10 statt mit `r`.

kB_warnung_ohne_wert.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
Linking CXX executable kB_warnung_ohne_wert.elf
Built target kB_warnung_ohne_wert
main.c: In function 'main':
main.c:5:9: warning: 'summe' is used uninitialized [-Wuninitialized]
    5 |     int summe;
      |         ^~~~~
```

**`'…' is used uninitialized`** – Eine lokale Variable wird gelesen, bevor sie einen Wert bekommen hat. Lokale Variablen haben anfangs einen zufälligen Inhalt. Hier fehlt `= 0` bei `int summe`.

kB_warnung_return.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
Linking CXX executable kB_warnung_return.elf
Built target kB_warnung_return
main.c: In function 'groesser':
main.c:9:1: warning: control reaches end of non-void function [-Wreturn-type]
    9 | }
      | ^
```

**`control reaches end of non-void function`** – Die Funktion soll einen Wert zurückgeben, kann aber das Ende erreichen, ohne auf ein `return` zu treffen. Hier fehlt der Fall `a == b`. Am sichersten steht am Ende der Funktion immer ein `return`.

kB_warnung_vorzeichen.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
Linking CXX executable kB_warnung_vorzeichen.elf
Built target kB_warnung_vorzeichen
main.c: In function 'main':
main.c:7:15: warning: comparison of integer expressions of different signedness:
  'uint32_t' {aka 'long unsigned int'} and 'int' [-Wsign-compare]
    7 |     if (jetzt < grenze) {
      |               ^
```

**`comparison of integer expressions of different signedness`** – Eine Zahl mit Vorzeichen (`int`) wird mit einer ohne Vorzeichen (`uint32_t`) verglichen. Ist die `int`-Zahl negativ, geht der Vergleich schief, denn C macht sie dafür vorher zu einer riesigen Zahl ohne Vorzeichen. Abhilfe: beide Werte vom selben Typ, hier `uint32_t grenze = 1000;`.

kB_warnung_ueberlauf.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
Linking CXX executable kB_warnung_ueberlauf.elf
Built target kB_warnung_ueberlauf
main.c: In function 'main':
main.c:5:26: warning: unsigned conversion from 'int' to 'uint8_t' {aka 'unsigned
  char'} changes value from '300' to '44' [-Woverflow]
    5 |     uint8_t helligkeit = 300;
      |                          ^~~
```

**`… changes value from '300' to '44'`** – Der Wert passt nicht in den Typ (Kapitel 5.2). Ein `uint8_t` reicht nur bis 255; aus 300 wird 300 − 256 = 44. Größeren Typ wählen oder den Wert begrenzen.

kB_warnung_einrueckung.c

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
Linking CXX executable kB_warnung_einrueckung.elf
Built target kB_warnung_einrueckung
main.c: In function 'main':
main.c:6:5: warning: this 'if' clause does not guard... [-Wmisleading-indentation]
    6 |     if (x == 6)
      |     ^~
main.c:8:9: note: ...this statement, but the latter is misleadingly indented as if it
  were guarded by the 'if'
    8 |         printf("Noch einmal!\n");
      |         ^~~~~~
```

**`this 'if' clause does not guard...`** – Die Einrückung täuscht: Ohne geschweifte Klammern gehört nur **eine** Anweisung zum `if`. Die zweite wird immer ausgeführt, obwohl sie eingerückt ist. Darum steht im Buch hinter jedem `if` ein Block mit `{ }`.

kB_warnung_durchfallen.c

```
main.c: In function 'main':
main.c:8:9: warning: this statement may fall through [-Wimplicit-fallthrough=]
    8 |         printf("eins\n");
      |         ^~~~~~~~~~~~~~~
main.c:9:5: note: here
    9 |     case 2:
      |     ^~~~
```

**`this statement may fall through`** – In einem `switch` fehlt das `break` am Ende eines `case` (Kapitel 8.3). Das Programm läuft dann in den nächsten `case` weiter („fällt durch“) und gibt hier bei 1 „eins“ **und** „zwei“ aus.

Eine weitere Warnung kennst du aus Kapitel 19.3: **`'sizeof' on array function parameter … will return size of 'int *'`**.

## B.6 Übersicht

| Meldung | häufige Ursache |
|---|---|
| `expected ';' before …` | Semikolon vergessen |
| `expected ')' before …` | Klammer nicht geschlossen |
| `expected declaration or statement at end of input` | `}` fehlt |
| `missing terminating " character` | `"` am Textende fehlt |
| `'else' without a previous 'if'` | Semikolon hinter `if (…)` |
| `break statement not within loop or switch` | `break` außerhalb einer Schleife |

| Meldung | häufige Ursache |
|---|---|
| `No such file or directory` | Dateiname bei `#include` falsch |
| `implicit declaration of function …` | Funktionsname falsch, `#include` fehlt |
| `… undeclared` | Variablenname falsch, nicht sichtbar |
| `redefinition of …` | Variable zweimal angelegt |
| `conflicting types for …` | Prototyp passt nicht zur Funktion |
| `makes pointer from integer` | Zahl statt Text oder `&` vergessen |
| `too few / too many arguments` | Zahl der Argumente falsch |
| `assignment to expression with array type` | Text mit `=` an Array zugewiesen |
| `assignment of read-only variable` | `const` geändert |
| `'return' with a value, in function returning void` | Rückgabetyp falsch |
| `undefined reference to …` (ld) | `.c`-Datei fehlt in `add_executable`, Name falsch |
| `multiple definition of …` (ld) | Definition in einer Header-Datei |
| `region 'RAM' overflowed` (ld) | zu viele oder zu große globale Variablen |
| Warnung `suggest parentheses around assignment` | `=` statt `==` |
| Warnung `format '%d' expects …` | falscher Platzhalter |
| Warnung `unused variable / parameter` | Variable oder Parameter vergessen |
| Warnung `is used uninitialized` | Startwert fehlt |
| Warnung `control reaches end of non-void function` | `return` fehlt |
| Warnung `different signedness` | `int` mit `uint32_t` verglichen |
| Warnung `changes value from … to …` | Wert passt nicht in den Typ |
| Warnung `does not guard` | `if` ohne `{ }` |
| Warnung `may fall through` | `break` im `switch` vergessen |

# Lösungen zu Kapitel 1: Mikrocontroller, C und das Pico SDK

### Aufgabe 1 – Begriffe ordnen

1. **Quelltext:** das Programm als Text in einer Programmiersprache, hier C.
2. **Compiler:** das Programm, das den Quelltext prüft und übersetzt.
3. **Maschinencode:** das, was dabei herauskommt – Zahlen, die der Prozessor direkt ausführen kann.
4. **`.uf2`-Datei:** die Datei, in der das übersetzte Programm zum Aufspielen bereitliegt.
5. **Bootmodus:** der Zustand, in dem sich der PicoBoy als Laufwerk meldet und eine `.uf2`-Datei annimmt (BOOT halten, RES tippen).

### Aufgabe 2 – Mikrocontroller im Alltag

Zum Beispiel: Waschmaschine, Mikrowelle, Fernbedienung, Digitalwecker, Kaffeemaschine, Spielzeug mit Tönen, Fahrradcomputer, Ladegerät mit Anzeige, Rauchmelder. Ein Hinweis ist alles, was Tasten auswertet, etwas anzeigt oder nach einem Programm abläuft.

### Aufgabe 3 – Speicher vergleichen

Ein Foto mit 3 MB sind 3 · 1024 = 3072 KB. In den Arbeitsspeicher (520 KB) passt also **kein einziges** solches Foto. In den Flash (16 MB) passen 16 / 3 ≈ **5** Fotos.

### Aufgabe 4 – Was passiert, wenn …?

Das Gerät ist nicht kaputt. Ein Programm kann den Bootmodus nicht überschreiben. Er soll den PicoBoy mit dem Computer verbinden, BOOT halten, RES tippen und ein funktionierendes Programm (zum Beispiel die Vorlage oder MicroPython) auf das Laufwerk kopieren.



# Lösungen zu Kapitel 2: Die Werkzeuge einrichten

### Aufgabe 1 – Ein anderer Gruß

Nur der Text in Anführungszeichen ändert sich:

```
pbc_text24("Moin!", 80, 120, PBC_WHITE);
```

Ist der Gruß länger, rückt er nach rechts aus der Mitte; dann x kleiner machen.

### Aufgabe 2 – Wie lange dauert es?

Das erste Bauen dauert etwa eine Minute, das zweite nur wenige Sekunden. Beim ersten Mal werden die Buchbibliothek und viele Teile des Pico SDK übersetzt. Danach übersetzt der Compiler nur noch, was sich geändert hat – hier also nur `main.c` – und bindet alles neu zusammen.

### Aufgabe 3 – Hin und zurück

Solange MicroPython läuft, funktioniert nur **Weg 1** (BOOT halten, RES tippen, Datei kopieren). **Run** braucht ein laufendes Programm aus diesem Buch oder den Bootmodus: MicroPython hört nicht auf den Befehl, mit dem VS Code den PicoBoy in den Bootmodus schaltet.

---

ANHANG C – LÖSUNGEN

## Lösungen zu Kapitel 3: Das erste Programm

### Aufgabe 1 – Wo erscheint was?

„oben“ steht weiß links oben (x = 30, y = 30), „Mitte“ grün ungefähr in der Mitte (x = 95, y = 130), „unten“ rot rechts unten (x = 150, y = 230). Je größer x, desto weiter rechts; je größer y, desto weiter unten.




### Aufgabe 2 – Drei Zeilen über dich (`k03_loesung2.c`)

k03_loesung2.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_text24("Jonas", 30, 70, PBC_YELLOW);
    pbc_text18("Klasse 10b", 30, 120, PBC_CYAN);
    pbc_text18("Hobby: Klettern", 30, 150, PBC_GREEN);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```




Die Zeilen mit `pbc_text18` haben 30 Pixel Abstand, damit sie sich nicht berühren.

### Aufgabe 3 – Finde die Fehler (`k03_loesung3.c`)

Beim ersten Bauen meldet der Compiler **zwei** Fehler:

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:6:5: error: implicit declaration of function 'pbc_Text24'; did you mean
  'pbc_text24'? [-Wimplicit-function-declaration]
    6 |     pbc_Text24("Treffen", 40, 80, PBC_YELLOW);
      |     ^~~~~~~~~~
      |     pbc_text24
main.c:7:51: error: expected ';' before 'pbc_text18'
    7 |     pbc_text18("Heute 14 Uhr", 40, 130, PBC_WHITE)
      |                                                   ^
      |                                                   ;
    8 |     pbc_text18("Raum 2.04", 40, 160, PBC_white);
      |     ~~~~~~~~~~
```

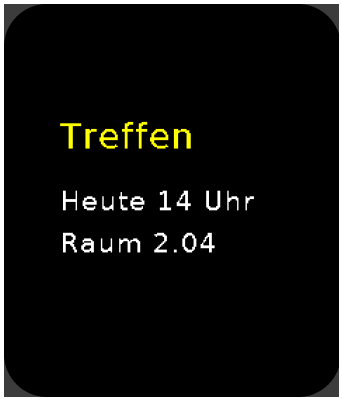
1. Zeile 6: `pbc_Text24` gibt es nicht, richtig ist `pbc_text24` (klein).
2. Zeile 7: Am Ende fehlt das Semikolon.

Den dritten Fehler meldet er erst nach dem zweiten Bauen: In Zeile 8 muss es `PBC_WHITE` heißen, nicht `PBC_white`. Wegen des fehlenden Semikolons hat der Compiler die Zeilen 7 und 8 für eine einzige Anweisung gehalten und Zeile 8 nicht mehr genau geprüft. Deshalb gilt: den ersten Fehler verbessern, neu bauen.

k03_loesung3.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_text24("Treffen", 40, 80, PBC_YELLOW);
    pbc_text18("Heute 14 Uhr", 40, 130, PBC_WHITE);
    pbc_text18("Raum 2.04", 40, 160, PBC_WHITE);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```



## Aufgabe 4 – Ein Schild (`k03_loesung4.c`)

k03_loesung4.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_RED);
    pbc_text24("STOP", 90, 115, PBC_WHITE);
    pbc_text12("Bitte warten", 85, 155, PBC_WHITE);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```




Die x-Werte sind ausprobiert. Wie man einen Text genau in die Mitte rechnet, lernst du in Kapitel 6.

## Aufgabe 5 – Stundenplan (`k03_loesung5.c`)

k03_loesung5.c

```c
// Stundenplan: pbc_text hat feste Breite (8 Pixel je Zeichen),
// deshalb stehen die Spalten mit Leerzeichen genau untereinander.
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_text18("Montag", 30, 30, PBC_YELLOW);
    pbc_text("Std Fach      Raum", 30, 70, PBC_CYAN);
    pbc_text("1   Mathe     204", 30, 90, PBC_WHITE);
    pbc_text("2   Mathe     204", 30, 105, PBC_WHITE);
    pbc_text("3   Englisch  112", 30, 120, PBC_WHITE);
    pbc_text("4   Physik    N3", 30, 135, PBC_WHITE);
    pbc_text("5   Sport     Halle", 30, 150, PBC_WHITE);
    pbc_text("6   Deutsch   108", 30, 165, PBC_WHITE);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

`pbc_text` hat feste Breite: Jedes Zeichen, auch das Leerzeichen, ist 8 Pixel breit. Wenn „Std“ mit dem folgenden Leerzeichen 4 Zeichen belegt und „Fach“ mit Leerzeichen 10, beginnen alle Spalten genau untereinander. Die Zeichen sind 8 Pixel hoch, der Zeilenabstand muss also mindestens 8 Pixel sein; mit 15 Pixeln ist es gut lesbar.




ANHANG C – LÖSUNGEN

# Lösungen zu Kapitel 4: Die Konsole

## Aufgabe 1 – Ein Gedicht (`k04_loesung1.c`)

k04_loesung1.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    printf("Rosen sind rot,\nVeilchen sind blau,\nC ist ganz schoen genau.\n");
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Rosen sind rot,
Veilchen sind blau,
C ist ganz schoen genau.
```

## Aufgabe 2 – Zeit in Zahlen (`k04_loesung2.c`)

k04_loesung2.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    printf("Stunden pro Woche: %d\n", 24 * 7);
    printf("Minuten pro Tag: %d\n", 24 * 60);
    printf("Sekunden pro Tag: %d\n", 24 * 60 * 60);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Stunden pro Woche: 168
Minuten pro Tag: 1440
Sekunden pro Tag: 86400
```

## Aufgabe 3 – Steckbrief (`k04_loesung3.c`)

k04_loesung3.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    char name[30];
    char farbe[30];
    pbc_input("Wie heisst du? ", name, sizeof name);
    pbc_input("Was ist deine Lieblingsfarbe? ", farbe, sizeof farbe);
    int alter = pbc_input_int("Wie alt bist du? ");
    printf("%s ist %d Jahre alt und mag am liebsten %s.\n", name, alter, farbe);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Wie heisst du? Ben
Was ist deine Lieblingsfarbe? Gruen
Wie alt bist du? 14
Ben ist 14 Jahre alt und mag am liebsten Gruen.
```

Jede Textvariable braucht ihre eigene Zeile mit `char …[30];`.

## Aufgabe 4 – Warnungen beseitigen (`k04_loesung4.c`)

Terminal

```
main.c: In function 'main':
main.c:5:26: warning: format '%f' expects argument of type 'double', but argument 2
  has type 'int' [-Wformat=]
    5 |     printf("Temperatur: %f Grad\n", 21);
      |                         ~^          ~~
      |                          |          |
      |                          double     int
      |                         %d
main.c:6:20: warning: format '%d' expects argument of type 'int', but argument 2 has
  type 'char *' [-Wformat=]
    6 |     printf("Name: %d\n", "Mia");
      |                   ~^     ~~~~~
      |                    |     |
      |                    int   char *
      |                   %s
```

Die Temperatur `21` ist eine ganze Zahl, also `%d` statt `%f`. Der Name `"Mia"` ist ein Text, also `%s` statt `%d`. Der Compiler schlägt beides unter der markierten Stelle vor.

k04_loesung4.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    printf("Temperatur: %d Grad\n", 21);
    printf("Name: %s\n", "Mia");
    while (true) {
        sleep_ms(1000);
    }
}
```

## Aufgabe 5 – Taschenrechner (`k04_loesung5.c`)

k04_loesung5.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    float a = pbc_input_float("Erste Zahl: ");
    float b = pbc_input_float("Zweite Zahl: ");
    printf("%.2f + %.2f = %.2f\n", a, b, a + b);
    printf("%.2f - %.2f = %.2f\n", a, b, a - b);
    printf("%.2f * %.2f = %.2f\n", a, b, a * b);
    printf("%.2f / %.2f = %.2f\n", a, b, a / b);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Erste Zahl: 7,5
Zweite Zahl: 2
7.50 + 2.00 = 9.50
7.50 - 2.00 = 5.50
7.50 * 2.00 = 15.00
7.50 / 2.00 = 3.75
```

Ist die zweite Zahl 0, ergibt die Division `inf` (*infinity*, unendlich):

Serial Monitor

```
Erste Zahl: 7,5
Zweite Zahl: 0
7.50 + 0.00 = 7.50
7.50 - 0.00 = 7.50
7.50 * 0.00 = 0.00
7.50 / 0.00 = inf
```

Bei Kommazahlen stürzt das Programm also nicht ab, sondern rechnet mit einem besonderen Wert „unendlich“ weiter. Mit ganzen Zahlen ist das anders; das zeigt Kapitel 5.



# Lösungen zu Kapitel 5: Variablen und Datentypen

## Aufgabe 1 – Was kommt heraus?

Serial Monitor

```
3
2
8.5
22 4
20
```

1. `17 / 5` ist eine ganzzahlige Division: 3 (Rest wird abgeschnitten).
2. `17 % 5` ist der Rest: 17 = 3 · 5 + 2, also 2.
3. `2.0` ist eine Kommazahl, deshalb rechnet C mit Kommazahlen: 8.5.
4. `a += b` macht aus `a` 17 + 5 = 22, `b--` aus `b` 5 − 1 = 4.
5. `a / b * b` wird von links nach rechts gerechnet: `22 / 4` ist 5 (ganzzahlig), `5 * 4` ist 20. Mit Kommazahlen käme wieder 22 heraus.

## Aufgabe 2 – Fahrenheit (`k05_loesung2.c`)

k05_loesung2.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    float celsius = pbc_input_float("Temperatur in Grad Celsius: ");
    float fahrenheit = celsius * 9 / 5 + 32;
    printf("%.1f Grad Celsius sind %.1f Grad Fahrenheit.\n", celsius,
      fahrenheit);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Temperatur in Grad Celsius: 21,5
21.5 Grad Celsius sind 70.7 Grad Fahrenheit.
```

Weil `celsius` ein `float` ist, rechnet C von Anfang an mit Kommazahlen. Vorsicht bei der Reihenfolge: `9 / 5 * celsius` ergäbe falsche Werte, denn `9 / 5` sind zwei ganze Zahlen und ergibt 1.

## Aufgabe 3 – Sekunden zerlegen (`k05_loesung3.c`)

k05_loesung3.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int gesamt = pbc_input_int("Sekunden: ");
    int stunden = gesamt / 3600;
    int rest = gesamt % 3600;
    int minuten = rest / 60;
    int sekunden = rest % 60;
    printf("%d s = %d h %d min %d s\n", gesamt, stunden, minuten, sekunden);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Sekunden: 4000
4000 s = 1 h 6 min 40 s
```

Eine Stunde hat 3600 Sekunden. `gesamt / 3600` sind die ganzen Stunden, `gesamt % 3600` die Sekunden, die danach übrig bleiben. Diesen Rest zerlegst du genauso in Minuten (`/ 60`) und Sekunden (`% 60`).

## Aufgabe 4 – Seltsame Werte (`k05_loesung4.c`)

Serial Monitor

```
Punkte: 44
Temperatur: -126
```

Beide Variablen laufen über. Ein `uint8_t` reicht nur bis 255. 200 + 100 = 300 passt nicht hinein; es bleibt 300 − 256 = 44. Ein `int8_t` reicht nur bis 127. 120 + 10 = 130 springt ins Negative: 130 − 256 = −126.

Die Lösung sind größere Typen. `int16_t` reicht bis 32 767 und kann negativ werden, das passt für Punkte und Temperaturen. Einfach `int` wäre genauso richtig.

k05_loesung4.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int16_t punkte = 200;
    punkte += 100;
    printf("Punkte: %d\n", punkte);

    int16_t temperatur = 120;
```

```
    temperatur += 10;
    printf("Temperatur: %d\n", temperatur);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Punkte: 300
Temperatur: 130
```

## Aufgabe 5 – Rückgeld (`k05_loesung5.c`)

k05_loesung5.c

```c
// Rueckgeld in Muenzen - alle Betraege in Cent
#include "pbc.h"

int main(void) {
    pbc_init();
    int preis = pbc_input_int("Preis in Cent: ");
    int bezahlt = pbc_input_int("Bezahlt in Cent: ");
    int rest = bezahlt - preis;
    printf("Rueckgeld: %d Cent\n", rest);

    printf("2 Euro:  %d\n", rest / 200);
    rest = rest % 200;
    printf("1 Euro:  %d\n", rest / 100);
    rest = rest % 100;
    printf("50 Cent: %d\n", rest / 50);
    rest = rest % 50;
    printf("20 Cent: %d\n", rest / 20);
    rest = rest % 20;
    printf("10 Cent: %d\n", rest / 10);
    rest = rest % 10;
    printf("5 Cent:  %d\n", rest / 5);
    rest = rest % 5;
    printf("2 Cent:  %d\n", rest / 2);
    rest = rest % 2;
    printf("1 Cent:  %d\n", rest);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Preis in Cent: 1370
Bezahlt in Cent: 2000
Rueckgeld: 630 Cent
2 Euro:  3
1 Euro:  0
50 Cent: 0
```

```
20 Cent: 1
10 Cent: 1
5 Cent:  0
2 Cent:  0
1 Cent:  0
```

Das Programm nimmt zuerst so viele 2-Euro-Stücke wie möglich: `rest / 200`. Was danach übrig ist, ist `rest % 200`. Mit diesem Rest geht es bei der nächstkleineren Münze weiter.

In Cent zu rechnen hat zwei Vorteile. Erstens gehen `/` und `%` nur mit ganzen Zahlen. Zweitens kann ein `float` viele Kommazahlen nicht genau speichern, etwa 0,10 oder 0,20: Er speichert eine Zahl, die ganz knapp daneben liegt. Bei Geld können sich solche winzigen Fehler zu einem falschen Cent aufsummieren. Ganze Zahlen sind dagegen immer genau.

---

ANHANG C – LÖSUNGEN

# Lösungen zu Kapitel 6: Zeichnen und Farben

## Aufgabe 1 – Eine Flagge (`k06_loesung1.c`)

k06_loesung1.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    uint16_t gold = pbc_rgb(255, 204, 0);
    pbc_clear(PBC_BLACK);
    pbc_fill_rect(0, 0, PBC_WIDTH, 93, PBC_BLACK);
    pbc_fill_rect(0, 93, PBC_WIDTH, 94, PBC_RED);
    pbc_fill_rect(0, 187, PBC_WIDTH, 93, gold);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```



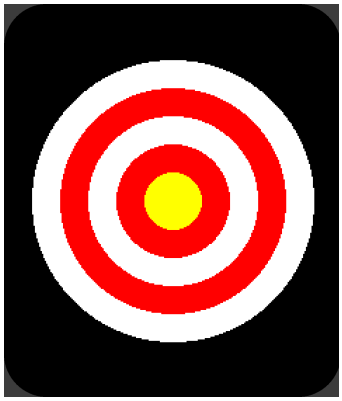
280 Pixel lassen sich nicht genau durch 3 teilen. Deshalb ist der rote Streifen 94 Pixel hoch, die beiden anderen 93. Der schwarze Streifen in Zeile 7 wäre eigentlich nicht nötig, weil `pbc_clear` schon alles schwarz gemacht hat. Mit ihm sieht man aber besser, wie die Flagge aufgebaut ist.

## Aufgabe 2 – Zielscheibe (`k06_loesung2.c`)

k06_loesung2.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    // vom groessten zum kleinsten Kreis
    pbc_fill_circle(120, 140, 100, PBC_WHITE);
    pbc_fill_circle(120, 140, 80, PBC_RED);
    pbc_fill_circle(120, 140, 60, PBC_WHITE);
    pbc_fill_circle(120, 140, 40, PBC_RED);
    pbc_fill_circle(120, 140, 20, PBC_YELLOW);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```



Der **größte** Kreis kommt zuerst. Jeder weitere, kleinere Kreis übermalt die Mitte des vorigen, und es bleibt ein Ring stehen. In umgekehrter Reihenfolge würde der große weiße Kreis zum Schluss alle anderen verdecken.

## Aufgabe 3 – Das Schild, genau in der Mitte (`k06_loesung3.c`)

k06_loesung3.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_RED);

    // beide Zeilen: 32 + 8 + 16 = 56 Pixel hoch
    int y = (PBC_HEIGHT - 56) / 2;
    int x = (PBC_WIDTH - pbc_text24_width("STOP")) / 2;
    pbc_text24("STOP", x, y, PBC_WHITE);

    x = (PBC_WIDTH - pbc_text12_width("Bitte warten")) / 2;
    pbc_text12("Bitte warten", x, y + 40, PBC_WHITE);

    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

Beide Zeilen zusammen sind 56 Pixel hoch: 32 für die Zeile von `pbc_text24`, 8 Pixel Abstand und 16 für die Zeile von `pbc_text12`. Dieser Block kommt in die senkrechte Mitte: `y = (280 - 56) / 2 = 112`. Die zweite Zeile beginnt 32 + 8 = 40 Pixel tiefer.



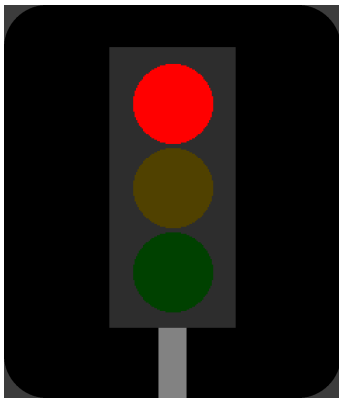

## Aufgabe 4 – Ampel (`k06_loesung4.c`)

k06_loesung4.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    uint16_t gehaeuse = pbc_rgb(50, 50, 50);
    uint16_t gelb_aus = pbc_rgb(80, 70, 0);
    uint16_t gruen_aus = pbc_rgb(0, 70, 0);

    pbc_clear(PBC_BLACK);
    pbc_fill_rect(110, 220, 20, 60, PBC_GRAY);  // Mast
    pbc_fill_rect(75, 30, 90, 200, gehaeuse);
    pbc_fill_circle(120, 70, 28, PBC_RED);      // an
    pbc_fill_circle(120, 130, 28, gelb_aus);    // aus
    pbc_fill_circle(120, 190, 28, gruen_aus);   // aus
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```



Ein ausgeschaltetes Licht ist dieselbe Farbe, nur viel dunkler: Statt Gelb (255, 255, 0) nimmt man etwa (80, 70, 0). Der Mast wird zuerst gezeichnet, damit das Gehäuse sein oberes Ende verdeckt.

## Aufgabe 5 – Balkendiagramm (`k06_loesung5.c`)

k06_loesung5.c

```c
// Balkendiagramm fuer drei Werte von 0 bis 100
#include "pbc.h"

#define UNTEN 240           // y der Grundlinie
#define BREITE 40           // Breite eines Balkens

int main(void) {
    pbc_init();
    int wert1 = pbc_input_int("Montag (0 bis 100): ");
    int wert2 = pbc_input_int("Dienstag (0 bis 100): ");
```

```
    int wert3 = pbc_input_int("Mittwoch (0 bis 100): ");
    int hoehe1 = wert1 * 2;     // 2 Pixel pro Punkt, hoechstens 200
    int hoehe2 = wert2 * 2;
    int hoehe3 = wert3 * 2;
    char text[10];
    int x;

    pbc_clear(PBC_BLACK);
    pbc_hline(30, UNTEN, 180, PBC_WHITE);

    // erster Balken bei x = 40, Zahl darueber zentriert
    pbc_fill_rect(40, UNTEN - hoehe1, BREITE, hoehe1, PBC_RED);
    snprintf(text, sizeof text, "%d", wert1);
    x = 40 + (BREITE - pbc_text12_width(text)) / 2;
    pbc_text12(text, x, UNTEN - hoehe1 - 18, PBC_WHITE);
    pbc_text12("Mo", 50, UNTEN + 4, PBC_GRAY);

    // zweiter Balken bei x = 100
    pbc_fill_rect(100, UNTEN - hoehe2, BREITE, hoehe2, PBC_GREEN);
    snprintf(text, sizeof text, "%d", wert2);
    x = 100 + (BREITE - pbc_text12_width(text)) / 2;
    pbc_text12(text, x, UNTEN - hoehe2 - 18, PBC_WHITE);
    pbc_text12("Di", 112, UNTEN + 4, PBC_GRAY);

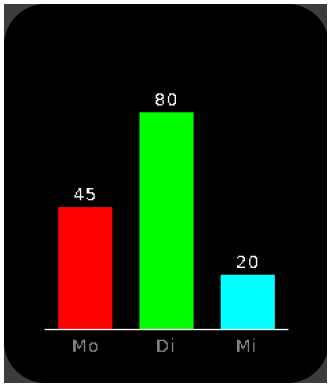
    // dritter Balken bei x = 160
    pbc_fill_rect(160, UNTEN - hoehe3, BREITE, hoehe3, PBC_CYAN);
    snprintf(text, sizeof text, "%d", wert3);
    x = 160 + (BREITE - pbc_text12_width(text)) / 2;
    pbc_text12(text, x, UNTEN - hoehe3 - 18, PBC_WHITE);
    pbc_text12("Mi", 171, UNTEN + 4, PBC_GRAY);

    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Montag (0 bis 100): 45
Dienstag (0 bis 100): 80
Mittwoch (0 bis 100): 20
```

Zeile 21 bis 26 zeichnen den ersten Balken, die beiden anderen sind genauso aufgebaut. Ein Balken der Höhe `hoehe1` steht auf der Grundlinie, wenn er bei `UNTEN - hoehe1` beginnt. Die Zahl soll über dem Balken stehen: Die Zeile von `pbc_text12` ist 16 Pixel hoch, mit 2 Pixeln Abstand beginnt sie 18 Pixel über der Oberkante. Um die Zahl über dem Balken zu zentrieren, rechnest du wie beim ganzen Bildschirm, nur mit der Balkenbreite statt `PBC_WIDTH`, und zählst den Anfang des Balkens dazu.




Dreimal fast derselbe Programmteil ist umständlich. In Kapitel 11 lernst du, wie du so etwas einmal als eigene Funktion schreibst.

ANHANG C – LÖSUNGEN

# Lösungen zu Kapitel 7: Schleifen und Zeit

## Aufgabe 1 – Zählen (`k07_loesung1.c`)

k07_loesung1.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    for (int i = 0; i <= 100; i += 5) {
        printf("%d ", i);
    }
    printf("\n");

    for (int i = 10; i >= 0; i--) {
        printf("%d\n", i);
    }
    printf("Start!\n");
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
0 5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100
10
9
8
7
6
5
4
3
```

```
2
1
0
Start!
```

Die erste Schleife zählt mit `i += 5` in Fünferschritten. Die zweite beginnt bei 10 und zählt mit `i--` herunter. Ihre Bedingung heißt `i >= 0`, damit auch die 0 noch ausgegeben wird.

## Aufgabe 2 – Warten auf die Sechs (`k07_loesung2.c`)

k07_loesung2.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int wurf;
    int anzahl = 0;
    do {
        wurf = pbc_random(1, 6);
        anzahl++;
        printf("%d ", wurf);
    } while (wurf != 6);
    printf("\nEine 6 nach %d Wuerfen.\n", anzahl);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
3 4 5 3 6
Eine 6 nach 5 Wuerfen.
```

Am besten passt `do … while`: Gewürfelt werden muss mindestens einmal, und erst danach steht fest, ob es weitergeht. `wurf` wird vor der Schleife angelegt, weil die Bedingung in Zeile 11 hinter der `}` steht und die Variable dort noch kennen muss. Bei dir kommen andere Würfe heraus.

## Aufgabe 3 – Ein Dreieck aus Sternen (`k07_loesung3.c`)

k07_loesung3.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int hoehe = pbc_input_int("Hoehe: ");
```

```
    for (int zeile = 1; zeile <= hoehe; zeile++) {
        for (int i = 0; i < zeile; i++) {
            printf("*");
        }
        printf("\n");
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Hoehe: 5
*
**
***
****
*****
```

Die äußere Schleife zählt die Zeilen von 1 bis zur Höhe. In jeder Zeile gibt die innere Schleife so viele Sterne aus, wie die Nummer der Zeile angibt: In Zeile 1 einen, in Zeile 5 fünf.

## Aufgabe 4 – Das Dach füllen (`k07_loesung4.c`)

k07_loesung4.c

```
    pbc_fill_rect(50, 140, 110, 90, wand);

    // Dach: Spitze bei (105, 85), Unterkante von x = 40 bis 170 bei y = 140.
    // Jede Zeile ist eine waagrechte Linie, nach unten immer breiter.
    for (int y = 85; y <= 140; y++) {
        int halb = (y - 85) * 65 / 55;
        pbc_hline(105 - halb, y, 2 * halb + 1, PBC_RED);
    }
```

Das Dach ist 55 Pixel hoch (von y = 85 bis 140). Auf diesem Weg wächst die Linie nach links und nach rechts um je 65 Pixel (von 105 bis 40 beziehungsweise bis 170). In der Zeile `y` ist sie deshalb nach jeder Seite `(y - 85) * 65 / 55` Pixel breiter als die Spitze. Eine Linie mit der halben Breite `halb` nach jeder Seite beginnt bei `105 - halb` und ist `2 * halb + 1` Pixel lang. Auch hier gilt: erst malnehmen, dann teilen. Der Rest des Programms ist das Haus aus Kapitel 6.



### Aufgabe 5 – Stoppuhr (`k07_loesung5.c`)

k07_loesung5.c

```c
// Eine Stoppuhr mit Minuten, Sekunden und Zehnteln
#include "pbc.h"

int main(void) {
    pbc_init();
    uint32_t start = pbc_millis();
    char text[20];
    while (true) {
        int ms = pbc_millis() - start;
        int minuten = ms / 60000;
        int sekunden = ms / 1000 % 60;
        int zehntel = ms / 100 % 10;
        snprintf(text, sizeof text, "%02d:%02d.%d", minuten, sekunden, zehntel);

        pbc_clear(PBC_BLACK);
        int x = (PBC_WIDTH - pbc_text24_width(text)) / 2;
        pbc_text24(text, x, 124, PBC_WHITE);
        pbc_show();
        sleep_ms(50);
    }
}
```

Zeile 9 rechnet aus, wie viele Millisekunden seit dem Start vergangen sind. Daraus zerlegen `/` und `%` die Zeit, wie in Kapitel 5, Aufgabe 3: `ms / 1000` sind die ganzen Sekunden, davon bleiben mit `% 60` die Sekunden innerhalb der Minute. `ms / 100` sind die Zehntelsekunden, `% 10` nimmt davon die letzte Stelle.




Die Schleife wartet nur 50 ms, damit sich die Zehntel flüssig ändern. Wie lange ein Durchlauf genau dauert, spielt keine Rolle: Die Anzeige rechnet immer mit der Uhr, nicht mit der Zahl der Durchläufe. Deshalb geht diese Stoppuhr genau.

ANHANG C – LÖSUNGEN

## Lösungen zu Kapitel 8: Entscheidungen

### Aufgabe 1 – Gerade oder ungerade (`k08_loesung1.c`)

k08_loesung1.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int zahl = pbc_input_int("Zahl: ");
```

```
    if (zahl % 2 == 0) {
        printf("%d ist gerade.\n", zahl);
    } else {
        printf("%d ist ungerade.\n", zahl);
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Zahl: 17
17 ist ungerade.
```

Eine Zahl ist gerade, wenn bei der Division durch 2 kein Rest bleibt: `zahl % 2 == 0`.

## Aufgabe 2 – Noten (`k08_loesung2.c`)

k08_loesung2.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int punkte = pbc_input_int("Punkte (0 bis 100): ");
    if (punkte < 0 || punkte > 100) {
        printf("%d Punkte gibt es nicht.\n", punkte);
    } else if (punkte >= 90) {
        printf("Note 1\n");
    } else if (punkte >= 80) {
        printf("Note 2\n");
    } else if (punkte >= 65) {
        printf("Note 3\n");
    } else if (punkte >= 50) {
        printf("Note 4\n");
    } else if (punkte >= 30) {
        printf("Note 5\n");
    } else {
        printf("Note 6\n");
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Punkte (0 bis 100): 72
Note 3
```

Die ungültigen Werte werden zuerst aussortiert. Danach prüft die Kette von der besten Note abwärts. Weil immer die erste erfüllte Bedingung gewinnt, genügt jeweils die untere Grenze: Bei 72 Punkten sind `punkte >= 90` und `punkte >= 80` falsch, `punkte >= 65` ist wahr.

## Aufgabe 3 – Schaltjahr (`k08_loesung3.c`)

k08_loesung3.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int jahr = pbc_input_int("Jahr: ");
    bool schaltjahr = (jahr % 4 == 0 && jahr % 100 != 0) || jahr % 400 == 0;
    if (schaltjahr) {
        printf("%d ist ein Schaltjahr.\n", jahr);
    } else {
        printf("%d ist kein Schaltjahr.\n", jahr);
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Jahr: 1900
1900 ist kein Schaltjahr.
```

Die Bedingung in Zeile 6 liest sich so: (durch 4 teilbar **und** nicht durch 100) **oder** durch 400 teilbar. Die Klammern sind hier nicht unbedingt nötig, weil `&&` vor `||` kommt, aber sie machen die Regel lesbar. 1900 ist durch 4 und durch 100 teilbar, aber nicht durch 400, also kein Schaltjahr.

## Aufgabe 4 – Taschenrechner mit Auswahl (`k08_loesung4.c`)

k08_loesung4.c

```c
// Taschenrechner fuer ganze Zahlen mit Rechenart-Auswahl
#include "pbc.h"

int main(void) {
    pbc_init();
    int a = pbc_input_int("Erste Zahl: ");
    int b = pbc_input_int("Zweite Zahl: ");
    printf("1 = plus, 2 = minus, 3 = mal, 4 = geteilt\n");
    int art = pbc_input_int("Rechenart: ");

    switch (art) {
    case 1:
        printf("%d + %d = %d\n", a, b, a + b);
        break;
    case 2:
        printf("%d - %d = %d\n", a, b, a - b);
        break;
    case 3:
        printf("%d * %d = %d\n", a, b, a * b);
        break;
```

```
    case 4:
        if (b == 0) {
            printf("Durch 0 kann man nicht teilen.\n");
        } else {
            printf("%d / %d = %d Rest %d\n", a, b, a / b, a % b);
        }
        break;
    default:
        printf("Rechenart %d gibt es nicht.\n", art);
        break;
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Erste Zahl: 17
Zweite Zahl: 5
1 = plus, 2 = minus, 3 = mal, 4 = geteilt
Rechenart: 4
17 / 5 = 3 Rest 2
```

Vor dem Teilen prüft das Programm, ob die zweite Zahl 0 ist. Dann gibt es eine Meldung statt einer Division, die auf dem PicoBoy einen falschen Wert ergäbe (Kapitel 5.3):

Serial Monitor

```
Erste Zahl: 17
Zweite Zahl: 0
1 = plus, 2 = minus, 3 = mal, 4 = geteilt
Rechenart: 4
Durch 0 kann man nicht teilen.
```

Im `case 4` steht ein ganzes `if … else`. In einem `case` darf jede Anweisung stehen, auch Schleifen.

## Aufgabe 5 – Der PicoBoy rät (`k08_loesung5.c`)

k08_loesung5.c

```
// Der PicoBoy raet deine Zahl, indem er den Bereich immer halbiert
#include "pbc.h"

int main(void) {
    pbc_init();
    printf("Denk dir eine Zahl von 1 bis 100.\n");
    int unten = 1;
    int oben = 100;
    int versuche = 0;

    while (true) {
        int tipp = (unten + oben) / 2;
```

```
13          versuche++;
14          printf("Ist es %d?\n", tipp);
15          int antwort = pbc_input_int("1 = zu klein, 2 = zu gross, 3 = richtig: ");
16          if (antwort == 3) {
17              break;
18          }
19          switch (antwort) {
20          case 1:
21              unten = tipp + 1;
22              break;
23          case 2:
24              oben = tipp - 1;
25              break;
26          default:
27              printf("Bitte 1, 2 oder 3 eingeben.\n");
28              versuche--;
29              break;
30          }
31          if (unten > oben) {
32              printf("Da stimmt etwas nicht. Hast du dich vertan?\n");
33              unten = 1;
34              oben = 100;
35          }
36      }
37      printf("Gefunden mit %d Versuchen.\n", versuche);
38      while (true) {
39          sleep_ms(1000);
40      }
41  }
```

Serial Monitor

```
Denk dir eine Zahl von 1 bis 100.
Ist es 50?
1 = zu klein, 2 = zu gross, 3 = richtig: 2
Ist es 25?
1 = zu klein, 2 = zu gross, 3 = richtig: 1
Ist es 37?
1 = zu klein, 2 = zu gross, 3 = richtig: 3
Gefunden mit 3 Versuchen.
```

Die Zahl liegt immer zwischen `unten` und `oben`. Der Tipp ist die Mitte (Zeile 12). Ist er zu klein, kann die Zahl nur noch darüber liegen: `unten` rückt auf `tipp + 1`. Ist er zu groß, rückt `oben` auf `tipp - 1`. So halbiert jeder Versuch den Bereich, und nach höchstens 7 Versuchen ist die Zahl gefunden, denn $2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 \cdot 2 = 128$ ist mehr als 100.

Die Antwort 3 wird vor dem `switch` mit `if` geprüft (Zeile 16). Ein `break` im `switch` würde nämlich nur den `switch` beenden, nicht die Schleife. Bei einer ungültigen Antwort zählt der Versuch nicht. Und wer sich bei einer Antwort vertan hat, bekommt irgendwann `unten > oben`; dann fängt der PicoBoy von vorn an.

# Lösungen zu Kapitel 9: Tasten und Joystick

## Aufgabe 1 – Licht an, Licht aus

(a) Der Bildschirm wird weiß und **bleibt** weiß. `pbc_was_pressed_a()` meldet den Druck einmal, und `licht = !licht` macht aus `false` ein `true`.

(b) Der Bildschirm wird wieder schwarz: `licht` wechselt zurück auf `false`.

(c) Weiß, solange B unten ist, danach wieder schwarz. `pbc_pressed_b()` ist nur während des Haltens `true`; durch das `||` reicht das für Weiß.

(d) Der Bildschirm wird beim Hinunterdrücken weiß und bleibt danach weiß. Auch ein langer Druck ist **ein** Druck: `licht` wechselt nur einmal.

## Aufgabe 2 – Zaubertafel (`k09_loesung2.c`)

k09_loesung2.c

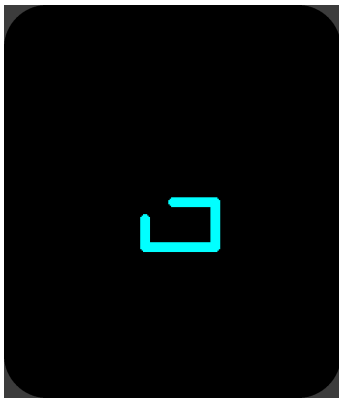
```c
// Zaubertafel: Der Punkt hinterlaesst eine Spur, B wischt die Tafel leer
#include "pbc.h"

#define RADIUS 3
#define SCHRITT 2

int main(void) {
    pbc_init();
    int x = 120;
    int y = 140;
    pbc_clear(PBC_BLACK);
    while (true) {
        if (pbc_pressed_left() && x > RADIUS) {
            x -= SCHRITT;
        }
        if (pbc_pressed_right() && x < PBC_WIDTH - RADIUS) {
            x += SCHRITT;
        }
        if (pbc_pressed_up() && y > RADIUS) {
            y -= SCHRITT;
        }
        if (pbc_pressed_down() && y < PBC_HEIGHT - RADIUS) {
            y += SCHRITT;
        }
        if (pbc_was_pressed_b()) {
            pbc_clear(PBC_BLACK);
        }

        pbc_fill_circle(x, y, RADIUS, PBC_CYAN);
        pbc_show();
        sleep_ms(20);
    }
}
```

Der Bildschirm wird nur einmal vor der Schleife gelöscht (Zeile 11) und danach nur noch, wenn B gedrückt wurde (Zeile 25). Alle früheren Punkte bleiben im Bildpuffer stehen, so entsteht die Spur. Mit einem kleineren Radius und kleineren Schritten wird die Linie feiner.



## Aufgabe 3 – Finde die Fehler (`k09_loesung3.c`)




**Fehler 1:** Zeile 9 fragt mit `pbc_pressed_up()` ab. Solange der Joystick oben ist, zählt **jeder** Durchlauf. Das Bild entstand nach einem Druck von 0,15 Sekunden: Die Stufe ist schon von 5 auf 9 gesprungen. Richtig ist `pbc_was_pressed_up()`.

**Fehler 2:** `pbc_was_pressed_down()` steht zweimal (Zeilen 12 und 15). Der erste Aufruf holt den Druck ab; der zweite findet ihn nicht mehr und liefert immer `false`. „Schon ganz leise!“ erscheint deshalb nie. Die Lösung liest jede Taste einmal und merkt sich das Ergebnis:

k09_loesung3.c

```
 1 // Lautstaerke von 0 bis 10 einstellen - verbessert
 2 #include "pbc.h"
 3
 4 int main(void) {
 5     pbc_init();
 6     int stufe = 5;
 7     char text[20];
 8     while (true) {
 9         bool hoch = pbc_was_pressed_up();
10         bool runter = pbc_was_pressed_down();
11         if (hoch && stufe < 10) {
12             stufe++;
13         }
14         if (runter && stufe > 0) {
15             stufe--;
16         } else if (runter && stufe == 0) {
17             printf("Schon ganz leise!\n");
18         }
19
20         pbc_clear(PBC_BLACK);
21         snprintf(text, sizeof text, "Stufe %d", stufe);
22         pbc_text24(text, 70, 100, PBC_WHITE);
23         pbc_fill_rect(20, 150, stufe * 20, 20, PBC_GREEN);
24         pbc_show();
```

```
        sleep_ms(20);
    }
}
```

Nach einem Druck nach oben und sieben Drücken nach unten:

Serial Monitor

```
Schon ganz leise!
```

Von Stufe 6 aus führen sechs Drücke auf 0; erst der siebte trifft auf Stufe 0 und erzeugt die Meldung.

## Aufgabe 4 – Stoppuhr (`k09_loesung4.c`)

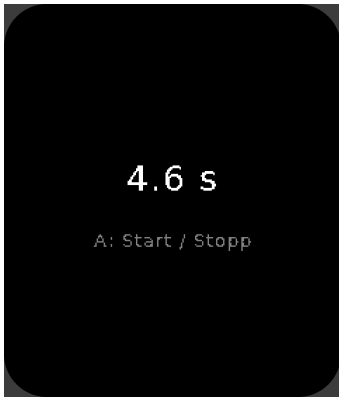
k09_loesung4.c

```c
// Stoppuhr: A startet, A haelt an, A startet wieder bei 0
#include "pbc.h"

int main(void) {
    pbc_init();
    bool laeuft = false;
    uint32_t start = 0;
    int ms = 0;
    char text[20];
    while (true) {
        if (pbc_was_pressed_a()) {
            if (laeuft) {
                laeuft = false;
                ms = pbc_millis() - start;
                printf("Gestoppt: %d ms\n", ms);
            } else {
                laeuft = true;
                start = pbc_millis();
            }
        }
        if (laeuft) {
            ms = pbc_millis() - start;
        }

        pbc_clear(PBC_BLACK);
        snprintf(text, sizeof text, "%d.%d s", ms / 1000, ms / 100 % 10);
        pbc_text24(text, (PBC_WIDTH - pbc_text24_width(text)) / 2, 110,
          PBC_WHITE);
        pbc_text12("A: Start / Stopp", 64, 160, PBC_GRAY);
        pbc_show();
        sleep_ms(20);
    }
}
```

- `laeuft` merkt sich, ob die Uhr gerade läuft. Jeder Druck auf A schaltet um (Zeilen 11 bis 20).
- Beim Start merkt sich Zeile 18 die Startzeit. Solange die Uhr läuft, rechnet Zeile 22 die vergangenen Millisekunden aus. Beim Anhalten wird `ms` ein letztes Mal berechnet und bleibt dann stehen.
- `ms / 1000` sind die ganzen Sekunden, `ms / 100 % 10` die Zehntel: Aus 4665 wird `4` und `6`, also `4.6 s`.




```
Serial Monitor
Gestoppt: 4680 ms
```

## Aufgabe 5 – Klick-Wettbewerb (`k09_loesung5.c`)

k09_loesung5.c

```c
// Klick-Wettbewerb: Wie oft schaffst du A in 5 Sekunden?
#include "pbc.h"

#define DAUER_MS 5000

int main(void) {
    pbc_init();
    int rekord = 0;
    char text[30];
    while (true) {
        pbc_clear(PBC_BLACK);
        pbc_text18("Start mit B", 65, 100, PBC_WHITE);
        snprintf(text, sizeof text, "Rekord: %d", rekord);
        pbc_text18(text, 65, 140, PBC_YELLOW);
        pbc_show();
        while (!pbc_was_pressed_b()) {
            sleep_ms(10);
        }

        int klicks = 0;
        pbc_was_pressed_a();                // alten Druck vergessen
        uint32_t start = pbc_millis();
        while (true) {
            int vergangen = pbc_millis() - start;
            if (vergangen >= DAUER_MS) {
                break;
            }
            if (pbc_was_pressed_a()) {
                klicks++;
            }
            pbc_clear(PBC_BLACK);
            snprintf(text, sizeof text, "%d", klicks);
            pbc_text24(text, (PBC_WIDTH - pbc_text24_width(text)) / 2, 100,
              PBC_WHITE);
            snprintf(text, sizeof text, "noch %d s", (DAUER_MS - vergangen + 999)
              / 1000);
```

```
35              pbc_text12(text, 90, 150, PBC_GRAY);
36              pbc_show();
37              sleep_ms(10);
38          }
39
40          printf("%d Klicks\n", klicks);
41          if (klicks > rekord) {
42              rekord = klicks;
43          }
44      }
45  }
```




- Die innere Schleife (ab Zeile 23) läuft, bis 5 Sekunden vergangen sind. `vergangen` wird einmal pro Durchlauf berechnet und dann für die Abfrage und die Anzeige benutzt.
- `(DAUER_MS - vergangen + 999) / 1000` rundet die restliche Zeit auf ganze Sekunden **auf**: Zu Beginn steht „noch 5 s“, kurz vor Schluss „noch 1 s“.
- Zeile 21 vergisst einen Druck auf A, der vor dem Start kam.

**Höchstens 20 Drücke pro Sekunde:** Die Bibliothek entprellt die Tasten. Ein neuer Druck zählt erst, wenn seit dem letzten gezählten mindestens 50 ms vergangen sind, und 1000 ms / 50 ms = 20. In 5 Sekunden sind also höchstens 100 Drücke möglich.

---

ANHANG C – LÖSUNGEN

# Lösungen zu Kapitel 10: LEDs, RGB-LED und Töne

## Aufgabe 1 – Welche Lichter brennen?

In der ersten halben Sekunde leuchten **Rot und Grün** (Zeilen 6 und 7).

Danach geht Rot aus (Zeile 10). `5 > 3` ist wahr, also geht in Zeile 11 **Gelb** an. Die RGB-LED wird kurz blau und sofort danach rot: Der zweite Aufruf ersetzt den ersten, das Blau ist nicht zu sehen. Nach einer weiteren halben Sekunde bekommt Grün den Wert von `pbc_pressed_a()`. Ohne Tastendruck ist das `false`, Grün geht aus.

Am Ende leuchten also **Gelb** und die **RGB-LED in Rot**.

## Aufgabe 2 – Finde die Fehler (`k10_loesung2.c`)

Du hörst nur **einen** langen Ton, und die rote LED blitzt höchstens kurz auf. Danach kommt der Schlusston, aber er hört nie auf. Drei Dinge sind falsch:

1. Hinter den Tönen fehlen die Pausen. Die drei `pbc_tone` laufen sofort nacheinander ab; jeder ersetzt den vorigen. Übrig bleibt der lange Ton. Auch die LED geht schon nach einem winzigen Augenblick wieder aus, nicht erst am Ende des Signals.

2. Die zwei kurzen Töne haben dieselbe Frequenz. Wartet das Programm genau 150 ms, gehen sie ohne Stille ineinander über. Die Wartezeit muss **länger** sein als der Ton, hier 250 ms.
3. Der Schlusston ist ein Dauerton (`0`). Ohne `pbc_tone_off()` klingt er für immer.

k10_loesung2.c

```c
// Signal: kurz, kurz, lang, dann ein Schlusston - verbessert
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_led_red(true);
    pbc_tone(880, 150);
    sleep_ms(250);                // 150 ms Ton + 100 ms Stille
    pbc_tone(880, 150);
    sleep_ms(250);
    pbc_tone(880, 600);
    sleep_ms(600);
    pbc_led_red(false);

    sleep_ms(500);
    pbc_tone(440, 0);             // Schlusston, eine halbe Sekunde
    sleep_ms(500);
    pbc_tone_off();
    while (true) {
        sleep_ms(1000);
    }
}
```

## Aufgabe 3 – Blaulicht (`k10_loesung3.c`)

k10_loesung3.c

```c
// Blaulicht: Blau und Rot im Wechsel, mit zwei Toenen
#include "pbc.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    pbc_text24("Einsatz!", 66, 120, PBC_WHITE);
    pbc_show();
    for (int i = 0; i < 5; i++) {
        pbc_rgb_led(0, 0, 255);
        pbc_tone(440, 0);
        sleep_ms(500);
        pbc_rgb_led(255, 0, 0);
        pbc_tone(587, 0);
        sleep_ms(500);
    }
    pbc_tone_off();
    pbc_rgb_led_off();
```



```
19      while (true) {
20          sleep_ms(1000);
21      }
22  }
```

Die Töne sind hier Dauertöne: Jeder neue Ton ersetzt den alten genau in dem Moment, in dem die Farbe wechselt. So entsteht ein lückenloses „Tatü-tata“. Nach der Schleife schalten die Zeilen 17 und 18 Ton und Licht aus.

### Aufgabe 4 – Die Entchen schwimmen weiter (`k10_loesung4.c`)

k10_loesung4.c

```c
// "Alle meine Entchen schwimmen auf dem See"
#include "pbc.h"

#define TON_C 262
#define TON_D 294
#define TON_E 330
#define TON_F 349
#define TON_G 392
#define TON_A 440

#define VIERTEL 300
#define HALBE 600
#define GANZE 1200
#define LUECKE 50

int main(void) {
    pbc_init();
    // Alle meine Entchen
    pbc_tone(TON_C, VIERTEL - LUECKE);
    sleep_ms(VIERTEL);
    pbc_tone(TON_D, VIERTEL - LUECKE);
    sleep_ms(VIERTEL);
    pbc_tone(TON_E, VIERTEL - LUECKE);
    sleep_ms(VIERTEL);
    pbc_tone(TON_F, VIERTEL - LUECKE);
    sleep_ms(VIERTEL);
    pbc_tone(TON_G, HALBE - LUECKE);
    sleep_ms(HALBE);
    pbc_tone(TON_G, HALBE - LUECKE);
    sleep_ms(HALBE);

    // schwimmen auf dem See
    for (int i = 0; i < 4; i++) {
        pbc_tone(TON_A, VIERTEL - LUECKE);
        sleep_ms(VIERTEL);
    }
    pbc_tone(TON_G, GANZE - LUECKE);
    sleep_ms(GANZE);
```

```
39      while (true) {
40          sleep_ms(1000);
41      }
42  }
```

Neu sind `TON_A` und `GANZE` (zwei halbe Noten). Die `for`-Schleife ab Zeile 33 spielt viermal dieselbe Viertelnote. Auch hier ist jeder Ton um `LUECKE` kürzer als die Wartezeit, sonst würden die vier a zu einem einzigen Ton verschmelzen.

### Aufgabe 5 – Ein Metronom (`k10_loesung5.c`)

Eine Minute hat 60 000 ms. Bei 100 Schlägen pro Minute liegen zwischen zwei Klicks 60 000 / 100 = **600 ms**, bei 120 Schlägen 500 ms.

k10_loesung5.c

```c
// Metronom: Klick und rote LED im Takt, Tempo mit dem Joystick
#include "pbc.h"

int main(void) {
    pbc_init();
    int tempo = 100;              // Schlaege pro Minute
    uint32_t letzter_klick = 0;
    char text[20];
    while (true) {
        if (pbc_was_pressed_up() && tempo < 240) {
            tempo += 10;
        }
        if (pbc_was_pressed_down() && tempo > 40) {
            tempo -= 10;
        }

        int seit_klick = pbc_millis() - letzter_klick;
        if (seit_klick >= 60000 / tempo) {
            pbc_tone(1000, 20);
            pbc_led_red(true);
            letzter_klick = pbc_millis();
        } else if (seit_klick >= 100) {
            pbc_led_red(false);
        }

        pbc_clear(PBC_BLACK);
        snprintf(text, sizeof text, "%d", tempo);
        pbc_text24(text, (PBC_WIDTH - pbc_text24_width(text)) / 2, 100,
          PBC_WHITE);
        pbc_text12("Schlaege pro Minute", 45, 140, PBC_GRAY);
        pbc_show();
        sleep_ms(5);
    }
}
```

- Die Schleife wartet jedes Mal nur 5 ms. So reagiert das Programm sofort auf den Joystick.
- `letzter_klick` merkt sich, wann zuletzt geklickt wurde. Zeile 17 rechnet aus, wie viel Zeit seitdem vergangen ist. Ist sie mindestens so groß wie der Abstand zwischen zwei Schlägen (`60000 / tempo`), klickt es, und die LED geht an.
- 100 ms nach dem Klick schaltet Zeile 23 die LED wieder aus.
- Ein Klick kann bis zu einem Schleifendurchlauf (etwa 25 ms) zu spät kommen. Für ein Metronom zum Üben reicht das.




ANHANG C – LÖSUNGEN

# Lösungen zu Kapitel 11: Funktionen

## Aufgabe 1 – Was gibt das Programm aus?

Serial Monitor
```
n = 50
a = 5, b = 14, zaehler = 3
```

- `aendern(a)` bekommt eine **Kopie** von `a`. Die Kopie wird 50 und so ausgegeben; `a` in `main` bleibt 5. `zaehler` ist jetzt 1.
- `naechste(a)` liefert 6, also ist `b` = 6. `zaehler` ist 2.
- `naechste(b) * 2`: Zuerst wird `naechste(6)` aufgerufen und liefert 7, dann wird mit 2 malgenommen. `b` = 14, `zaehler` = 3.
- `zaehler` ist global. Alle drei Aufrufe zählen dieselbe Variable hoch.

## Aufgabe 2 – Finde die Fehler (`k11_loesung2.c`)

Beim ersten Bauen meldet der Compiler einen Fehler und zwei Warnungen:

Terminal
```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:6:13: error: implicit declaration of function 'flaeche' [-Wimplicit-function-
  declaration]
    6 |     int f = flaeche(4);
      |             ^~~~~~
main.c: In function 'flaeche':
main.c:14:9: warning: unused variable 'ergebnis' [-Wunused-variable]
   14 |     int ergebnis = breite * hoehe;
```

```
      |         ^~~~~~~
main.c:15:1: warning: control reaches end of non-void function [-Wreturn-type]
   15 | }
      | ^
```

**Fehler 1:** `flaeche` steht hinter `main` und ist in Zeile 6 noch unbekannt. Die Funktion muss über `main` stehen (oder oben ein Prototyp). Danach meldet der Compiler den nächsten Fehler:

Terminal

```
main.c: In function 'flaeche':
main.c:5:9: warning: unused variable 'ergebnis' [-Wunused-variable]
    5 |     int ergebnis = breite * hoehe;
      |         ^~~~~~~
main.c: In function 'main':
main.c:10:13: error: too few arguments to function 'flaeche'
   10 |     int f = flaeche(4);
      |             ^~~~~~
main.c:4:5: note: declared here
    4 | int flaeche(int breite, int hoehe) {
      |     ^~~~~~
main.c: In function 'flaeche':
main.c:6:1: warning: control reaches end of non-void function [-Wreturn-type]
    6 | }
      | ^
```

**Fehler 2:** `flaeche` hat zwei Parameter, der Aufruf übergibt nur ein Argument. Richtig ist `flaeche(4, 3)`. Vorher konnte der Compiler das nicht bemerken: Er kannte die Funktion ja noch nicht.

**Fehler 3:** Die Warnungen sagen es schon: `ergebnis` wird berechnet, aber nie benutzt, und die Funktion gibt nichts zurück. Es fehlt `return ergebnis;`.

k11_loesung2.c

```c
// Flaeche eines Rechtecks - verbessert
#include "pbc.h"

int flaeche(int breite, int hoehe) {
    int ergebnis = breite * hoehe;
    return ergebnis;
}

int main(void) {
    pbc_init();
    int f = flaeche(4, 3);
    printf("Flaeche: %d\n", f);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Flaeche: 12
```

## Aufgabe 3 – Schaltjahre (`k11_loesung3.c`)

k11_loesung3.c

```c
// Schaltjahre erkennen
#include "pbc.h"

bool ist_schaltjahr(int jahr) {
    if (jahr % 400 == 0) {
        return true;
    }
    if (jahr % 100 == 0) {
        return false;
    }
    return jahr % 4 == 0;
}

void pruefe_jahr(int jahr) {
    if (ist_schaltjahr(jahr)) {
        printf("%d ist ein Schaltjahr.\n", jahr);
    } else {
        printf("%d ist kein Schaltjahr.\n", jahr);
    }
}

int main(void) {
    pbc_init();
    pruefe_jahr(1900);
    pruefe_jahr(2000);
    pruefe_jahr(2024);
    pruefe_jahr(2026);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

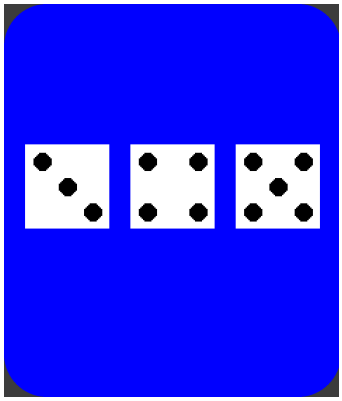
```
1900 ist kein Schaltjahr.
2000 ist ein Schaltjahr.
2024 ist ein Schaltjahr.
2026 ist kein Schaltjahr.
```

Die Reihenfolge der Prüfungen ist wichtig: Zuerst die Ausnahme von der Ausnahme (durch 400 teilbar), dann die Ausnahme (durch 100 teilbar), zuletzt die Regel (durch 4 teilbar). Weil `return` die Funktion sofort beendet, braucht es dafür kein `else`. `pruefe_jahr` ist eine `void`-Funktion, die selbst `ist_schaltjahr` aufruft.

## Aufgabe 4 – Würfelbilder (`k11_loesung4.c`)

Die Hilfsfunktion `auge` zeichnet ein Auge in einem Raster aus drei mal drei Plätzen: `spalte` und `zeile` sind jeweils 0, 1 oder 2. Damit wird `zeichne_wuerfel` kurz:

- Bei ungerader Augenzahl (1, 3, 5) gibt es ein Auge in der Mitte.
- Ab 2 Augen kommen zwei gegenüberliegende Ecken dazu, ab 4 die beiden anderen Ecken, bei 6 noch die Mitte links und rechts.



k11_loesung4.c

```c
// Drei Wuerfel mit einer Funktion zeichnen
#include "pbc.h"

#define SEITE 60

void auge(int x, int y, int spalte, int zeile) {
    // spalte und zeile: 0, 1 oder 2 (links/Mitte/rechts, oben/Mitte/unten)
    pbc_fill_circle(x + 12 + spalte * 18, y + 12 + zeile * 18, 6, PBC_BLACK);
}

void zeichne_wuerfel(int x, int y, int augen) {
    pbc_fill_rect(x, y, SEITE, SEITE, PBC_WHITE);
    if (augen % 2 == 1) {
        auge(x, y, 1, 1);                       // 1, 3, 5: Mitte
    }
    if (augen >= 2) {
        auge(x, y, 0, 0);                       // zwei Ecken
        auge(x, y, 2, 2);
    }
    if (augen >= 4) {
        auge(x, y, 2, 0);                       // die anderen Ecken
        auge(x, y, 0, 2);
    }
    if (augen == 6) {
        auge(x, y, 0, 1);                       // Mitte links und rechts
        auge(x, y, 2, 1);
    }
}

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLUE);
    int summe = 0;
    for (int i = 0; i < 3; i++) {
        int wurf = pbc_random(1, 6);
        summe += wurf;
        zeichne_wuerfel(15 + i * 75, 100, wurf);
    }
```

```
    printf("Summe: %d\n", summe);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Summe: 12
```

## Aufgabe 5 – Primzahlen (`k11_loesung5.c`)

k11_loesung5.c

```c
// Primzahlen finden und zaehlen
#include "pbc.h"

bool ist_primzahl(int n) {
    if (n < 2) {
        return false;
    }
    for (int teiler = 2; teiler * teiler <= n; teiler++) {
        if (n % teiler == 0) {
            return false;             // Teiler gefunden: fertig
        }
    }
    return true;
}

int anzahl_primzahlen(int bis) {
    int anzahl = 0;
    for (int n = 2; n <= bis; n++) {
        if (ist_primzahl(n)) {
            anzahl++;
        }
    }
    return anzahl;
}

int main(void) {
    pbc_init();
    for (int n = 1; n <= 100; n++) {
        if (ist_primzahl(n)) {
            printf("%d ", n);
        }
    }
    printf("\n");
    printf("Bis 100: %d Primzahlen\n", anzahl_primzahlen(100));
    printf("Bis 1000: %d Primzahlen\n", anzahl_primzahlen(1000));
    while (true) {
        sleep_ms(1000);
    }
}
```

```
Serial Monitor
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
Bis 100: 25 Primzahlen
Bis 1000: 168 Primzahlen
```

- `ist_primzahl` probiert alle Teiler ab 2 durch. Findet sie einen, ist die Zahl keine Primzahl, und `return false` beendet die Suche sofort.
- Es reicht, Teiler bis zur Wurzel von `n` zu prüfen: Hat `n` einen Teiler, der größer ist, gehört dazu immer ein kleinerer, der schon geprüft wurde. `teiler * teiler <= n` drückt das ohne Wurzel aus.
- `anzahl_primzahlen` benutzt `ist_primzahl`. Funktionen bauen aufeinander auf; jede einzelne bleibt kurz und lässt sich für sich prüfen.

---

ANHANG C – LÖSUNGEN

# Lösungen zu Kapitel 12: Arrays und Texte

## Aufgabe 1 – Was gibt das Programm aus?

```
Serial Monitor
13 1 11
Bil 3 7
```

- `a = z[2] + z[5]` = 4 + 9 = **13**. Der Index beginnt bei 0: `z[2]` ist das dritte Element.
- `z[0] = z[z[1]]`: Zuerst wird der innere Index ausgerechnet, `z[1]` ist 1. Also `z[0] = z[1]` = **1**.
- Die Schleife beginnt bei 1 und springt in Zweierschritten: `z[1] + z[3] + z[5]` = 1 + 1 + 9 = **11**.
- `w[3] = '\0'` beendet den Text nach drei Buchstaben: `Bil`, `strlen` ist 3. Das Array ist aber unverändert 7 Bytes groß (6 Buchstaben und die Endmarke von „Bilder“).

## Aufgabe 2 – Finde den Fehler (`k12_loesung2.c`)

„Hallo“ hat fünf Buchstaben, braucht mit der Endmarke aber **sechs** Plätze.
`char wort[5] = "Hallo";` füllt alle fünf Plätze mit Buchstaben, für die `'\0'` ist kein Platz mehr. C erlaubt das ohne Warnung: Ein Array, in das der Text genau ohne Endmarke passt, gilt als gewollt.

`printf` mit `%s` und `strlen` suchen das Ende aber an der `'\0'`. Sie lesen deshalb hinter `wort[4]` weiter, in fremdem Speicher, bis zufällig ein Byte 0 kommt. Ob dort sofort eine 0 steht oder erst nach ein paar anderen Bytes, hängt davon ab, wie der Compiler die Variablen anordnet. Das ist undefiniertes Verhalten (Abschnitt 12.5).

Am sichersten lässt du die eckigen Klammern leer. Dann zählt der Compiler selbst und denkt an die Endmarke:

k12_loesung2.c

```c
// Verbessert: Platz fuer das '\0' am Ende
#include "pbc.h"

int main(void) {
    pbc_init();
    char wort[] = "Hallo";        // der Compiler zaehlt selbst: 6 Bytes
    int zahl = 42;
    printf("%s hat %d Buchstaben\n", wort, (int)strlen(wort));
    printf("Die Zahl ist %d\n", zahl);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Hallo hat 5 Buchstaben
Die Zahl ist 42
```

## Aufgabe 3 – Würfelstatistik (`k12_loesung3.c`)

k12_loesung3.c

```c
// Wuerfelstatistik: 600 Wuerfe zaehlen und als Balken zeigen
#include "pbc.h"

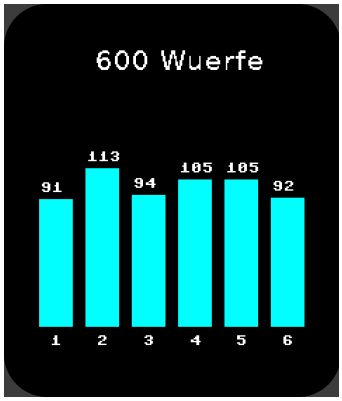
#define WUERFE 600

int main(void) {
    pbc_init();
    int anzahl[7] = {0};          // anzahl[1] bis anzahl[6]; anzahl[0] bleibt frei
    for (int i = 0; i < WUERFE; i++) {
        int augen = pbc_random(1, 6);
        anzahl[augen]++;
    }

    char text[12];
    pbc_clear(PBC_BLACK);
    pbc_text18("600 Wuerfe", 65, 30, PBC_WHITE);
    for (int augen = 1; augen <= 6; augen++) {
        printf("%d: %d-mal\n", augen, anzahl[augen]);
        int x = 25 + (augen - 1) * 33;
        int hoehe = anzahl[augen];            // 1 Pixel pro Wurf
        pbc_fill_rect(x, 230 - hoehe, 24, hoehe, PBC_CYAN);
        snprintf(text, sizeof text, "%d", augen);
        pbc_text(text, x + 8, 236, PBC_WHITE);
        snprintf(text, sizeof text, "%d", anzahl[augen]);
```

```
25          pbc_text(text, x + 1, 230 - hoehe - 12, PBC_WHITE);
26      }
27      pbc_show();
28      while (true) {
29          sleep_ms(1000);
30      }
31  }
```

- Der Trick steht in Zeile 11: Die Augenzahl selbst ist der Index. Deshalb hat `anzahl` sieben Plätze; `anzahl[0]` wird nie benutzt, dafür passt `anzahl[6]`.
- Jede Augenzahl sollte ungefähr 600 / 6 = 100-mal kommen. Genau 100 sind es fast nie; Abweichungen um 10 bis 20 sind bei 600 Würfen normal. Auf dem PicoBoy kommt bei jedem Start ein anderes Ergebnis heraus.




Serial Monitor

```
1: 91-mal
2: 113-mal
3: 94-mal
4: 105-mal
5: 105-mal
6: 92-mal
```

## Aufgabe 4 – Rückwärts (`k12_loesung4.c`)

k12_loesung4.c

```
 1  // Ein Wort rueckwaerts - und ist es ein Palindrom?
 2  #include "pbc.h"
 3
 4  bool ist_palindrom(char wort[]) {
 5      int links = 0;
 6      int rechts = strlen(wort) - 1;
 7      while (links < rechts) {
 8          if (wort[links] != wort[rechts]) {
 9              return false;
10          }
11          links++;
12          rechts--;
13      }
14      return true;
15  }
16
17  int main(void) {
18      pbc_init();
19      char wort[30];
20      char rueckwaerts[30];
21      while (true) {
22          pbc_input("Wort: ", wort, sizeof wort);
```

```
23          int laenge = strlen(wort);
24          for (int i = 0; i < laenge; i++) {
25              rueckwaerts[i] = wort[laenge - 1 - i];
26          }
27          rueckwaerts[laenge] = '\0';
28          printf("Rueckwaerts: %s\n", rueckwaerts);
29          if (ist_palindrom(wort)) {
30              printf("Ein Palindrom!\n");
31          }
32      }
33  }
```

Serial Monitor

```
Wort: Hallo
Rueckwaerts: ollaH
Wort: rentner
Rueckwaerts: rentner
Ein Palindrom!
Wort:
```

- Das letzte Zeichen hat den Index `laenge - 1`. Die Schleife in Zeile 24 holt deshalb `wort[laenge - 1 - i]`: für `i` = 0 das letzte Zeichen, für `i` = 1 das vorletzte und so weiter.
- Zeile 27 hängt die Endmarke an. Ohne sie wäre `rueckwaerts` kein gültiger Text.
- `ist_palindrom` vergleicht von außen nach innen: erstes mit letztem Zeichen, zweites mit vorletztem. Beim ersten Unterschied steht das Ergebnis fest. Groß- und Kleinbuchstaben sind verschiedene Zeichen: „Otto“ ist hier kein Palindrom, „otto“ schon.

## Aufgabe 5 – Eine Melodie als Text (`k12_loesung5.c`)

k12_loesung5.c

```c
// Eine Melodie als Text: jeder Buchstabe eine Viertelnote, '-' eine Pause
#include "pbc.h"

#define VIERTEL 300
#define LUECKE 50

int frequenz(char ton) {
    switch (ton) {
        case 'c': return 262;
        case 'd': return 294;
        case 'e': return 330;
        case 'f': return 349;
        case 'g': return 392;
        case 'a': return 440;
        case 'h': return 494;
        default:  return 0;       // '-' und alles andere: Pause
    }
}
```

```
20 void spiele(const char melodie[]) {
21     for (int i = 0; melodie[i] != '\0'; i++) {
22         int hz = frequenz(melodie[i]);
23         if (hz > 0) {
24             pbc_tone(hz, VIERTEL - LUECKE);
25         }
26         sleep_ms(VIERTEL);
27     }
28 }
29
30 int main(void) {
31     pbc_init();
32     spiele("cdefg-g-aaaag---aaaag---ffffe-e-ddddc---");
33     while (true) {
34         sleep_ms(1000);
35     }
36 }
```

- `frequenz` ordnet mit `switch` jedem Buchstaben seine Frequenz zu. Ein `return` beendet die Funktion sofort, deshalb braucht es hier kein `break`. Für alle anderen Zeichen gibt `default` 0 zurück.
- `spiele` läuft durch den Text bis zur Endmarke. Bei einer Pause wird kein Ton gestartet, gewartet wird trotzdem.
- Der Parameter ist `const char melodie[]`: Die Funktion verspricht, den Text nur zu lesen. So darf man ihr auch einen Text in Anführungszeichen direkt übergeben (Zeile 32).
- Eine neue Melodie ist jetzt nur noch eine Zeile. Probiere `"ccggaag-ffeeddc-"` („Morgen kommt der Weihnachtsmann“).

ANHANG C – LÖSUNGEN

# Lösungen zu Kapitel 13: Strukturen

## Aufgabe 1 – Was gibt das Programm aus?

Serial Monitor

```
in nach_unten: (1, 12)
a = (1, 2)
b = (5, 2)
c = (15, 2)
```

- `b = a` kopiert `a`. `b.x = 5` ändert nur die Kopie: `a` bleibt (1, 2), `b` wird (5, 2).
- `nach_unten(a)` bekommt eine Kopie von `a` und ändert nur sie. Die Funktion gibt deshalb (1, 12) aus, `a` bleibt unverändert.
- `nach_rechts(b)` ändert ebenfalls eine Kopie, gibt sie aber zurück. `c` wird (15, 2), `b` bleibt (5, 2).

## Aufgabe 2 – Eine Spielertabelle (`k13_loesung2.c`)

k13_loesung2.c

```c
// Eine Tabelle aus Strukturen: Spieler mit Name, Punkten und Level
#include "pbc.h"

#define ANZAHL 4

typedef struct {
    char name[12];
    int punkte;
    int level;
} spieler;

int main(void) {
    pbc_init();
    spieler alle[ANZAHL] = {
        {"Mia", 1250, 3},
        {"Ben", 2100, 4},
        {"Lea", 980, 2},
        {"Tom", 1730, 4},
    };

    printf("%-8s %6s %5s\n", "Name", "Punkte", "Level");
    int bester = 0;
    for (int i = 0; i < ANZAHL; i++) {
        printf("%-8s %6d %5d\n", alle[i].name, alle[i].punkte, alle[i].level);
        if (alle[i].punkte > alle[bester].punkte) {
            bester = i;
        }
    }
    printf("Vorne liegt %s mit %d Punkten.\n", alle[bester].name,
      alle[bester].punkte);
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Name     Punkte Level
Mia        1250     3
Ben        2100     4
Lea         980     2
Tom        1730     4
Vorne liegt Ben mit 2100 Punkten.
```

- Ein Text in einer Struktur ist ein `char`-Array mit fester Größe. Bei den Startwerten in Zeile 15 bis 18 kann er wie jeder Text in Anführungszeichen stehen.
- `%-8s` gibt den Namen linksbündig in 8 Zeichen Breite aus, `%6d` die Punkte rechtsbündig in 6 Zeichen. So stehen die Spalten untereinander.
- `bester` merkt sich wie in Kapitel 12 den **Index** des besten Spielers. Über den Index kommt man danach an alle seine Felder.

### Aufgabe 3 – Finde die Fehler (`k13_loesung3.c`)

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c:11:1: error: expected ';', identifier or '(' before 'int'
   11 | int main(void) {
      | ^~~
main.c: In function 'main':
main.c:13:5: error: unknown type name 'rechteck'; use 'struct' keyword to refer to
  the type
   13 |     rechteck r = {20, 40, 100, 60};
      |     ^~~~~~~~
      |     struct
main.c:13:23: warning: excess elements in scalar initializer
   13 |     rechteck r = {20, 40, 100, 60};
      |                       ^~
main.c:13:23: note: (near initialization for 'r')
main.c:13:27: warning: excess elements in scalar initializer
   13 |     rechteck r = {20, 40, 100, 60};
      |                           ^~~
main.c:13:27: note: (near initialization for 'r')
main.c:13:32: warning: excess elements in scalar initializer
   13 |     rechteck r = {20, 40, 100, 60};
      |                                ^~
main.c:13:32: note: (near initialization for 'r')
main.c:14:20: error: request for member 'x' in something not a structure or union
   14 |     pbc_fill_rect(r.x, r.y, r.Breite, r.hoehe, PBC_BLUE);
      |                    ^
main.c:14:25: error: request for member 'y' in something not a structure or union
   14 |     pbc_fill_rect(r.x, r.y, r.Breite, r.hoehe, PBC_BLUE);
      |                         ^
main.c:14:30: error: request for member 'Breite' in something not a structure or
  union
   14 |     pbc_fill_rect(r.x, r.y, r.Breite, r.hoehe, PBC_BLUE);
      |                              ^
main.c:14:40: error: request for member 'hoehe' in something not a structure or union
   14 |     pbc_fill_rect(r.x, r.y, r.Breite, r.hoehe, PBC_BLUE);
      |                                        ^
main.c:13:14: warning: variable 'r' set but not used [-Wunused-but-set-variable]
   13 |     rechteck r = {20, 40, 100, 60};
      |              ^
```

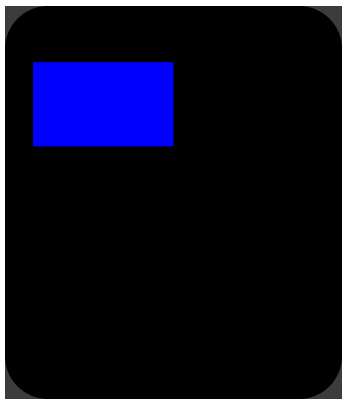
So viele Meldungen, aber nur **drei** Fehler. Die meisten Meldungen sind Folgefehler:

1. **Zeile 9:** Hinter der `}` der Struktur fehlt das Semikolon. Der Compiler merkt es erst in Zeile 11 und meldet `expected ';' … before 'int'`.
2. **Zeile 13:** Ohne `typedef` heißt der Typ `struct rechteck`, nicht `rechteck`. Der Compiler schlägt sogar vor, `struct` davorzuschreiben. Weil er den Typ nicht kennt, hält er `r` für eine einfache Zahl. Daher kommen die Warnungen *excess elements in scalar initializer* (zu viele

Startwerte für eine einzelne Zahl) und die Fehler *request for member … in something not a structure* in Zeile 14: Eine Zahl hat keine Felder.

3. **Zeile 14:** Das Feld heißt `breite`, nicht `Breite`. Diesen Fehler meldet der Compiler erst, wenn die ersten beiden behoben sind.

Den ersten Fehler zuerst verbessern und neu bauen: Das spart hier viel Rätselraten.



k13_loesung3.c

```c
// Ein Rechteck als Struktur - verbessert
#include "pbc.h"

struct rechteck {
    int x;
    int y;
    int breite;
    int hoehe;
};

int main(void) {
    pbc_init();
    struct rechteck r = {20, 40, 100, 60};
    pbc_fill_rect(r.x, r.y, r.breite, r.hoehe, PBC_BLUE);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

## Aufgabe 4 – Ampel mit `enum` (`k13_loesung4.c`)

k13_loesung4.c

```c
// Eine Ampel mit enum: A schaltet zur naechsten Phase
#include "pbc.h"

typedef enum { ROT, ROT_GELB, GRUEN, GELB } phase;
```

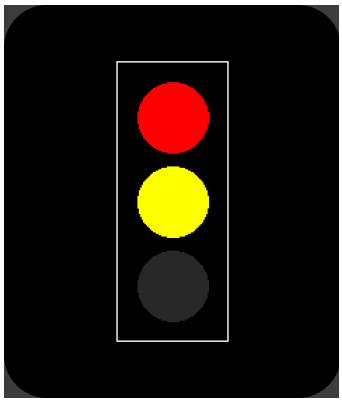
```c
phase naechste(phase p) {
    switch (p) {
        case ROT:
            return ROT_GELB;
        case ROT_GELB:
            return GRUEN;
        case GRUEN:
            return GELB;
        case GELB:
            return ROT;
    }
    return ROT;
}

uint16_t lampe(bool an, uint16_t farbe) {
    if (an) {
        return farbe;
    }
    return pbc_rgb(40, 40, 40);   // ausgeschaltete Lampe: dunkelgrau
}

void zeige(phase p) {
    bool rot = (p == ROT || p == ROT_GELB);
    bool gelb = (p == ROT_GELB || p == GELB);
    bool gruen = (p == GRUEN);
    pbc_led_red(rot);
    pbc_led_yellow(gelb);
    pbc_led_green(gruen);

    pbc_clear(PBC_BLACK);
    pbc_fill_rect(80, 40, 80, 200, PBC_BLACK);
    pbc_rect(80, 40, 80, 200, PBC_WHITE);
    pbc_fill_circle(120, 80, 25, lampe(rot, PBC_RED));
    pbc_fill_circle(120, 140, 25, lampe(gelb, PBC_YELLOW));
    pbc_fill_circle(120, 200, 25, lampe(gruen, PBC_GREEN));
    pbc_show();
}

int main(void) {
    pbc_init();
    phase p = ROT;
    zeige(p);
    while (true) {
        if (pbc_was_pressed_a()) {
            p = naechste(p);
            zeige(p);
        }
        sleep_ms(10);
    }
}
```



- `naechste` ordnet jeder Phase mit `switch` die folgende zu. Das `return ROT;` am Ende wird nie erreicht, beruhigt aber den Compiler: Ohne diese Zeile könnte er nicht sicher sein, dass jeder Weg ein `return` hat.
- `zeige` rechnet zuerst aus, welche Lampen leuchten. `rot` ist wahr bei Rot und bei Rot-Gelb. Die drei `bool` gehen direkt an die Status-LEDs.
- `lampe` liefert die Farbe einer Lampe: ihre Farbe, wenn sie an ist, sonst dunkelgrau.
- Neu gezeichnet wird nur, wenn A gedrückt wurde. Dazwischen wartet die Schleife jeweils 10 ms.

Das Bild zeigt die Phase Rot-Gelb nach dem ersten Druck auf A.

## Aufgabe 5 – Bestenliste (`k13_loesung5.c`)

k13_loesung5.c

```c
// Eine Bestenliste: sortieren und neue Ergebnisse eintragen
#include "pbc.h"

#define PLAETZE 5

typedef struct {
    char name[12];
    int punkte;
} eintrag;

eintrag liste[PLAETZE] = {
    {"Mia", 1250}, {"Ben", 2100}, {"Lea", 980}, {"Tom", 1730}, {"Ida", 1400},
};

// Bubblesort: Benachbarte Eintraege tauschen, bis alles absteigend sortiert ist
void sortiere(void) {
    for (int runde = 0; runde < PLAETZE - 1; runde++) {
        for (int i = 0; i < PLAETZE - 1 - runde; i++) {
            if (liste[i].punkte < liste[i + 1].punkte) {
                eintrag tausch = liste[i];
                liste[i] = liste[i + 1];
                liste[i + 1] = tausch;
            }
        }
    }
}
```

```c
void zeige_liste(void) {
    for (int i = 0; i < PLAETZE; i++) {
        printf("%d. %-10s %5d\n", i + 1, liste[i].name, liste[i].punkte);
    }
}

int main(void) {
    pbc_init();
    sortiere();
    zeige_liste();
    while (true) {
        eintrag neu;
        pbc_input("Name: ", neu.name, sizeof neu.name);
        neu.punkte = pbc_input_int("Punkte: ");
        if (neu.punkte > liste[PLAETZE - 1].punkte) {
            liste[PLAETZE - 1] = neu;     // den Letzten ersetzen ...
            sortiere();                   // ... und an die richtige Stelle
              bringen
            printf("Neu in der Bestenliste!\n");
        } else {
            printf("Leider nicht genug fuer die Bestenliste.\n");
        }
        zeige_liste();
    }
}
```

Serial Monitor

```
1. Ben         2100
2. Tom         1730
3. Ida         1400
4. Mia         1250
5. Lea          980
Name: Max
Punkte: 1500
Neu in der Bestenliste!
1. Ben         2100
2. Tom         1730
3. Max         1500
4. Ida         1400
5. Mia         1250
Name: Ole
Punkte: 500
Leider nicht genug fuer die Bestenliste.
1. Ben         2100
2. Tom         1730
3. Max         1500
4. Ida         1400
5. Mia         1250
Name:
```

- `sortiere` arbeitet nach dem Verfahren **Bubblesort**: Es vergleicht immer zwei Nachbarn und tauscht sie, wenn der hintere mehr Punkte hat. Nach der ersten Runde steht der Eintrag mit den wenigsten Punkten ganz hinten, nach jeder weiteren Runde ein weiterer an seinem Platz.

Deshalb wird die innere Schleife jede Runde um eins kürzer.

- Zum Tauschen (Zeilen 20 bis 22) braucht man einen Zwischenspeicher. Weil `=` ganze Strukturen kopiert, wandern Name und Punkte gemeinsam.
- `pbc_input` schreibt direkt in das Feld `neu.name`; `sizeof neu.name` ist seine Größe, 12.
- Ein neues Ergebnis ersetzt den letzten Platz. Ein erneutes Sortieren schiebt es an die richtige Stelle.



# Lösungen zu Kapitel 14: Flimmerfrei animieren

## Aufgabe 1 – Lahmer Sternenhimmel (`k14_loesung1.c`)

`pbc_show()` steht **in** der `for`-Schleife. Es wird also nach jedem einzelnen Stern aufgerufen, 20-mal pro Bild. Jedes Mal wandert der ganze Bildpuffer zum Display, und man sieht die Sterne einzeln dazukommen. Ein Bild dauert dadurch 20-mal so lange wie ein `pbc_show()`, dazu kommt die Pause von 100 ms. Misst man die Zeit für die Schleife (ohne Pause), bestätigt sich das. Mit den 31,4 ms je `pbc_show()`, die der PicoBoy misst, sind es 20 × 31,4 ms = 628 ms:

Serial Monitor

```
ein Bild: 628 ms
ein Bild: 628 ms
```

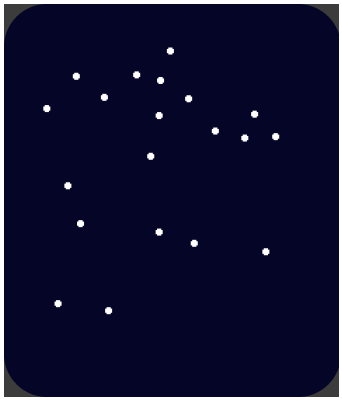
Die Verbesserung: `pbc_show()` hinter die Schleife, also erst alle Sterne zeichnen, dann einmal zeigen.

k14_loesung1.c

```
1 // Sternenhimmel: alle Sterne zeichnen, dann einmal zeigen
2 #include "pbc.h"
3
4 int main(void) {
5     pbc_init();
6     while (true) {
7         uint32_t start = pbc_millis();
8         pbc_clear(pbc_rgb(10, 10, 40));
9         for (int i = 0; i < 20; i++) {
10            int x = pbc_random(30, 210);
11            int y = pbc_random(30, 250);
12            pbc_fill_circle(x, y, 2, PBC_WHITE);
13        }
14        pbc_show();                       // nur noch einmal pro Bild
15        printf("ein Bild: %d ms\n", (int)(pbc_millis() - start));
16        sleep_ms(100);
17    }
18 }
```

```
Serial Monitor
ein Bild: 31 ms
ein Bild: 32 ms
ein Bild: 31 ms
ein Bild: 32 ms
ein Bild: 31 ms
```

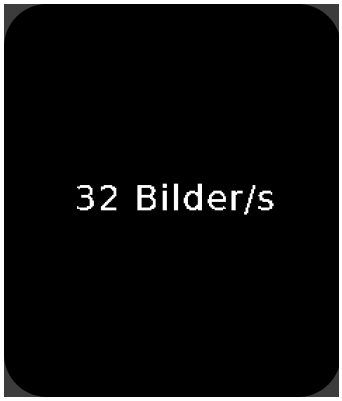
Auf dem Gerät sind beide Zahlen größer: `pbc_show()` dauert dort länger (dein Messwert aus 14.2), und auch das Zeichnen kostet Zeit. Das Ergebnis bleibt: Mit der Verbesserung ist ein Bild viel schneller fertig.



## Aufgabe 2 – Schneller geht's nicht (`k14_loesung2.c`)

Mit `BILD_MS 100` zeigt der PicoBoy 10 Bilder pro Sekunde an: 100 ms reichen locker für Zeichnen und `pbc_show()`, der Rest wird gewartet.

Mit `BILD_MS 10` kommt keine 100 heraus. Schon `pbc_show()` allein dauert länger als ein ganzes Zeitfenster. `rest` ist immer negativ, `bildtakt()` wartet gar nicht mehr, und das Programm läuft so schnell es kann: höchstens 1000 geteilt durch die Dauer eines `pbc_show()` in Millisekunden. Bei 31,4 ms (gemessen, 14.1) sind das etwa 32 Bilder pro Sekunde, wegen der Zeit fürs Zeichnen eher etwas weniger. Das Bild rechts stammt aus dem Prüflauf auf dem PC, der 31,4 ms je `pbc_show()` annimmt: 1000 / 31,4 ≈ 32.




Setzt du in `k14_bildtakt.c` `BILD_MS` auf 80, läuft der Ball **halb so schnell** und ruckelt sichtbar: Er rückt weiterhin 3 Pixel pro Bild vor, aber jetzt kommen nur 12,5 statt 25 Bilder pro Sekunde. Wer die Bildrate ändert, ändert bei „Pixel pro Bild" auch das Tempo. Aufgabe 5 zeigt, wie man das vermeidet.

## Aufgabe 3 – Bildschirmschoner (`k14_loesung3.c`)

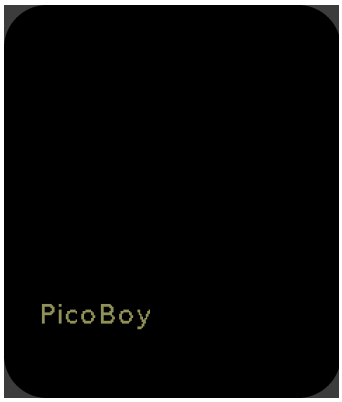
k14_loesung3.c

```c
uint16_t zufallsfarbe(void) {
    return pbc_rgb(pbc_random(80, 255), pbc_random(80, 255),
                   pbc_random(80, 255));
}

int main(void) {
    pbc_init();
```

```
27      int breite = pbc_text18_width(TEXT);
28      int x = 40, y = 60;
29      int vx = 3, vy = 2;
30      uint16_t farbe = PBC_YELLOW;
31
32      while (true) {
33          x = x + vx;
34          y = y + vy;
35          if (x < 20 || x + breite > 220) {
36              vx = -vx;
37              x = x + vx;
38              farbe = zufallsfarbe();
39          }
40          if (y < 25 || y + HOEHE > 255) {
41              vy = -vy;
42              y = y + vy;
43              farbe = zufallsfarbe();
44          }
45          pbc_clear(PBC_BLACK);
46          pbc_text18(TEXT, x, y, farbe);
47          pbc_show();
48          bildtakt();
49      }
50  }
```

- Zeilen 20–23: Eine eigene Funktion liefert eine zufällige, nicht zu dunkle Farbe.
- Zeile 27: Die Breite des Texts wird nur einmal ausgerechnet; sie ändert sich nicht.
- Der Text wird über seine linke obere Ecke (`x`, `y`) bewegt. Rechts stößt er an, wenn `x + breite` über die Grenze geht, unten bei `y + HOEHE`. Für die Höhe nehmen wir die Zeilenhöhe von `pbc_text18` (24 Pixel).




Die Zeilen 1–19 enthalten die Konstanten und `bildtakt()` wie im Kapitel.

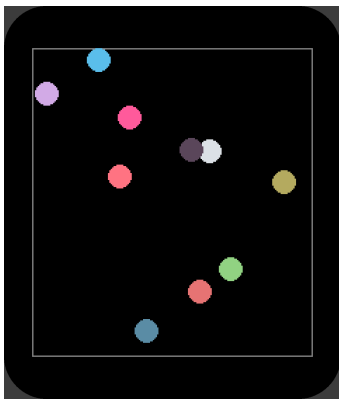
## Aufgabe 4 – Zehn Bälle (`k14_loesung4.c`)

k14_loesung4.c

```
12  typedef struct {
13      float x, y;
14      float vx, vy;
15      uint16_t farbe;
16  } kugel;
17
18  kugel baelle[ANZAHL];
```

k14_loesung4.c

```
32 int main(void) {
33     pbc_init();
34     for (int i = 0; i < ANZAHL; i++) {
35         baelle[i].x = pbc_random(LINKS + RADIUS, RECHTS - RADIUS);
36         baelle[i].y = pbc_random(OBEN + RADIUS, UNTEN - RADIUS);
37         baelle[i].vx = pbc_random(-40, 40) / 10.0f;
38         baelle[i].vy = pbc_random(-40, 40) / 10.0f;
39         baelle[i].farbe = pbc_rgb(pbc_random(60, 255), pbc_random(60, 255),
40                                   pbc_random(60, 255));
41     }
42
43     while (true) {
44         pbc_clear(PBC_BLACK);
45         pbc_rect(LINKS, OBEN, RECHTS - LINKS, UNTEN - OBEN, PBC_GRAY);
46         for (int i = 0; i < ANZAHL; i++) {
47             kugel b = baelle[i];            // eine Kopie bearbeiten ...
48             b.x = b.x + b.vx;
49             b.y = b.y + b.vy;
50             if (b.x - RADIUS < LINKS || b.x + RADIUS > RECHTS) {
51                 b.vx = -b.vx;
52                 b.x = b.x + b.vx;
53             }
54             if (b.y - RADIUS < OBEN || b.y + RADIUS > UNTEN) {
55                 b.vy = -b.vy;
56                 b.y = b.y + b.vy;
57             }
58             pbc_fill_circle((int)b.x, (int)b.y, RADIUS, b.farbe);
59             baelle[i] = b;                  // ... und zurueckschreiben
60         }
61         pbc_show();
62         bildtakt();
63     }
64 }
```



- Das Array `baelle` steht außerhalb von `main`, ist also global. So liegt es nicht auf dem kleinen Stack (Kapitel 19).
- Zeilen 37–38: `pbc_random(-40, 40) / 10.0f` ergibt eine Kommazahl von −4,0 bis 4,0. Wegen `10.0f` wird als `float` geteilt, nicht ganzzahlig.
- In der Schleife ab Zeile 46 wird jeder Ball bewegt, geprüft und gezeichnet. Das `pbc_show()` steht erst hinter der Schleife.
- Zeile 47 kopiert den Ball in die Variable `b`. Mit `b.x` liest es sich leichter als mit `baelle[i].x`. Weil `b` aber nur eine Kopie ist, muss Zeile 59 das Ergebnis zurück ins Array schreiben. Sonst bliebe jeder Ball stehen.

### Aufgabe 5 – Pixel pro Sekunde (`k14_loesung5.c`)

k14_loesung5.c

```c
// Tempo in Pixel pro Sekunde: gleich schnell, egal wie lange ein Bild dauert
#include "pbc.h"

#define TEMPO 120.0f            // Pixel pro Sekunde

int main(void) {
    pbc_init();
    float x = 30.0f;
    float vx = TEMPO;
    uint32_t vorher = time_us_32();

    while (true) {
        uint32_t jetzt = time_us_32();
        float dt = (jetzt - vorher) / 1000000.0f;   // in Sekunden
        vorher = jetzt;

        x = x + vx * dt;
        if (x < 30.0f || x > 210.0f) {
            vx = -vx;
            x = x + vx * dt;
        }

        pbc_clear(PBC_BLACK);
        pbc_fill_circle((int)x, 120, 20, PBC_YELLOW);
        if (pbc_pressed_b()) {
            pbc_text18("viel Arbeit", 66, 200, PBC_RED);
            sleep_ms(80);           // so tun, als gaebe es viel zu tun
        } else {
            pbc_text18("wenig Arbeit", 62, 200, PBC_GREEN);
        }
        pbc_show();
        sleep_ms(10);
    }
}
```

- Zeilen 13–15: `dt` ist die Zeit seit dem vorigen Durchgang in Sekunden, zum Beispiel 0,027 bei 27 ms. Geteilt wird durch `1000000.0f`, damit das Ergebnis eine Kommazahl ist.
- Zeile 17: In `dt` Sekunden legt der Ball `vx * dt` Pixel zurück. Dauert ein Bild länger, macht er einen größeren Schritt.
- Zeilen 25–27: Solange B gedrückt ist, dauert jedes Bild 80 ms länger.




Mit B gedrückt springt der Ball in größeren Schritten, das Bild ruckelt also. Er braucht für eine Strecke aber gleich lange: 180 Pixel von Rand zu Rand in 1,5 Sekunden. So machen es viele Spiele auf dem PC, deren Bildrate je nach Rechner schwankt.

# Lösungen zu Kapitel 15: Bilder und Sprites

## Aufgabe 1 – Speicher rechnen

- Eine Figur mit 32 × 32 = 1024 Pixeln: 1024 × 2 = 2048 Bytes Farben und 1024 Bytes Maske, zusammen **3072 Bytes** (3 KB).
- Acht solcher Bilder: 8 × 3072 = **24 576 Bytes** (24 KB).
- Ein Hintergrundbild in Displaygröße ohne Maske: 240 × 280 × 2 = **134 400 Bytes**, so viel wie der ganze Bildpuffer.

Im Flash (16 MB) ist dafür reichlich Platz, solange die Daten `const` sind.

## Aufgabe 2 – Ein Geist (`k15_loesung2.c`)

k15_loesung2.c

```c
// Ein Geist aus 8 x 8 Pixeln, von Hand geschrieben
#include "pbc.h"

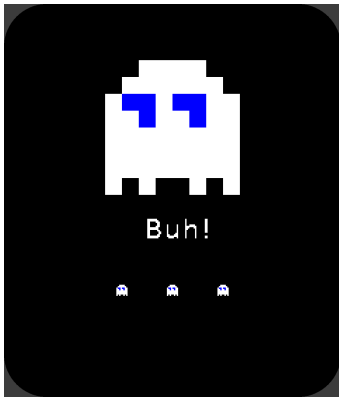
#define S PBC_BLACK
#define W PBC_WHITE
#define B PBC_BLUE

static const uint16_t geist_pixel[64] = {
    S, S, W, W, W, W, S, S,
    S, W, W, W, W, W, W, S,
    W, B, B, W, B, B, W, W,
    W, W, B, W, W, B, W, W,
    W, W, W, W, W, W, W, W,
    W, W, W, W, W, W, W, W,
    W, W, W, W, W, W, W, W,
    W, S, W, S, S, W, S, W,
};
static const pbc_image geist = { 8, 8, geist_pixel, NULL };

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLACK);
    // gross: jedes Pixel als Quadrat mit 12 Pixeln Kantenlaenge
    for (int zeile = 0; zeile < 8; zeile++) {
        for (int spalte = 0; spalte < 8; spalte++) {
            uint16_t farbe = geist_pixel[zeile * 8 + spalte];
            pbc_fill_rect(72 + spalte * 12, 40 + zeile * 12, 12, 12, farbe);
        }
    }
    pbc_text18("Buh!", 100, 150, PBC_WHITE);
```

```
31      // klein: dreimal in Originalgroesse
32      for (int i = 0; i < 3; i++) {
33          pbc_draw_image(&geist, 80 + i * 36, 200);
34      }
35      pbc_show();
36      while (true) {
37          sleep_ms(1000);
38      }
39  }
```

Das Programm ist aufgebaut wie `k15_herz.c`. Die Zeilen 24–29 zeichnen jedes Pixel als Quadrat mit 12 Pixeln Kantenlänge, die Zeilen 32–34 dreimal das Bild in Originalgröße, jeweils 36 Pixel weiter rechts. Der schwarze Hintergrund `S` fällt auf dem schwarzen Bildschirm nicht auf. Auf einer anderen Farbe sähe man das Rechteck um den Geist. Dann bräuchte das Bild eine Maske.




## Aufgabe 3 – Sternenhimmel (`k15_loesung3.c`)

k15_loesung3.c

```
44  int main(void) {
45      pbc_init();
46      pbc_clear(pbc_rgb(10, 10, 50));
47      pbc_fill_circle(60, 70, 25, pbc_rgb(240, 240, 200));     // Mond
48      pbc_fill_circle(72, 62, 22, pbc_rgb(10, 10, 50));        // Mondsichel
49      for (int i = 0; i < 15; i++) {
50          int x = pbc_random(20, 204);
51          int y = pbc_random(30, 230);
52          pbc_draw_image(&stern, x, y);
53      }
54      pbc_show();
55      while (true) {
56          sleep_ms(1000);
57      }
58  }
```

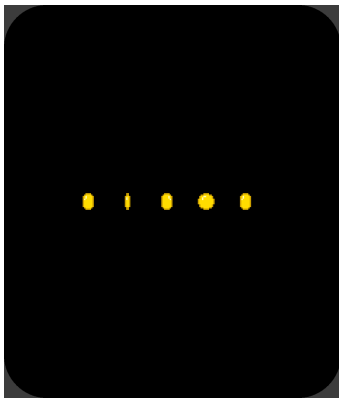


- Die Zeilen 1–42 sind die Ausgabe von `bild2c` für `stern.png` mit dem Namen `stern`, wie in `k15_stern.c`.
- Zeilen 47–48: Die Mondsichel entsteht aus einem hellen Kreis, über den ein Kreis in der Farbe des Himmels gemalt wird, etwas nach rechts oben versetzt.
- Zeilen 50–51: Ein Stern ist 16 Pixel groß. x geht deshalb nur bis 204, damit kein Stern über den rechten Rand ragt (204 + 16 = 220), und alle halten Abstand zu den runden Ecken.
- Weil der Stern eine Maske hat, liegen auch Sterne, die sich überdecken oder vor dem Mond stehen, sauber übereinander.

## Aufgabe 4 – Münze dreht sich (`k15_loesung4.c`)

k15_loesung4.c

```
104 int main(void) {
105     pbc_init();
106     const pbc_image muenze[4] = { muenze_0, muenze_1, muenze_2, muenze_3 };
107     int bild = 0;
108     while (true) {
109         pbc_clear(PBC_BLACK);
110         for (int i = 0; i < 5; i++) {
111             // jede Muenze ist ein Bild weiter, so drehen sie versetzt
112             pbc_draw_image(&muenze[(bild + i) % 4], 54 + i * 28, 134);
113         }
114         pbc_show();
115         bild = (bild + 1) % 4;
116         sleep_ms(100);
117     }
118 }
```



- Die Zeilen 1–102 sind die vier umgewandelten Bilder mit den Namen `muenze_0` bis `muenze_3`. Sie sind 12 × 12 Pixel groß.
- Zeile 106 sammelt sie in einem Array.
- Zeile 112: Münze Nummer `i` zeigt das Bild `bild + i`. `% 4` sorgt dafür, dass nach Bild 3 wieder Bild 0 kommt. So ist jede Münze der linken einen Schritt voraus.
- Zeile 115 schaltet nach jedem Bild eins weiter.

## Aufgabe 5 – Eigene Zeichenschleife (`k15_loesung5.c`)

k15_loesung5.c

```c
// Eigene Zeichenschleife: Magenta ist durchsichtig, vergroessert, gespiegelt
#include "pbc.h"

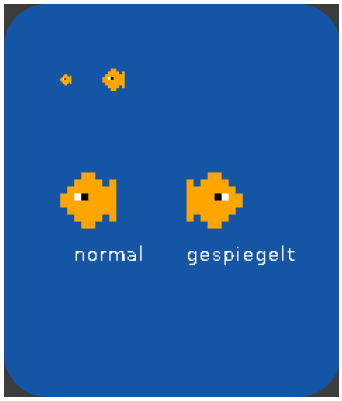
#define D PBC_MAGENTA           // durchsichtig
#define O PBC_ORANGE
#define K PBC_BLACK
#define W PBC_WHITE

static const uint16_t fisch[64] = {
    D, D, D, O, O, D, D, D,
    D, D, O, O, O, O, D, O,
    D, O, O, O, O, O, O, O,
    O, O, W, K, O, O, O, O,
    O, O, O, O, O, O, O, O,
    D, O, O, O, O, O, O, O,
    D, D, O, O, O, O, D, O,
    D, D, D, O, O, D, D, D,
};

// Zeichnet den Fisch faktor-mal so gross; gespiegelt: schaut nach rechts
void zeichne_fisch(int x, int y, int faktor, bool gespiegelt) {
    for (int zeile = 0; zeile < 8; zeile++) {
        for (int spalte = 0; spalte < 8; spalte++) {
            int quelle = spalte;
            if (gespiegelt) {
                quelle = 7 - spalte;
            }
            uint16_t farbe = fisch[zeile * 8 + quelle];
            if (farbe != D) {
                pbc_fill_rect(x + spalte * faktor, y + zeile * faktor,
                              faktor, faktor, farbe);
            }
        }
    }
}

int main(void) {
    pbc_init();
    pbc_clear(pbc_rgb(30, 90, 160));
    zeichne_fisch(40, 50, 1, false);
    zeichne_fisch(70, 46, 2, false);
    zeichne_fisch(40, 120, 5, false);
    zeichne_fisch(130, 120, 5, true);
    pbc_text12("normal", 50, 170, PBC_WHITE);
    pbc_text12("gespiegelt", 130, 170, PBC_WHITE);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

- Zeile 4: `D` steht für Magenta, die Farbe, die als durchsichtig gilt. Sie darf im Fisch selbst natürlich nicht vorkommen.
- Zeilen 24–27: Beim Spiegeln wird die Spalte von hinten gelesen. Für Spalte 0 des Bildschirms nimmt die Funktion Spalte 7 des Fischs, für Spalte 1 Spalte 6 und so weiter.
- Zeile 29: Magenta-Pixel werden einfach übersprungen. Dort bleibt der Hintergrund stehen.
- Zeilen 30–31: Jedes Pixel wird als Quadrat mit der Kantenlänge `faktor` gezeichnet.




Die Funktion arbeitet nur mit dem Array `fisch`. Eine Funktion, die beliebige Bilder spiegeln kann, braucht Zeiger (Kapitel 19).

---

ANHANG C – LÖSUNGEN

# Lösungen zu Kapitel 16: Kollisionen

## Aufgabe 1 – Berühren sie sich?

1. **Ja.** a reicht in x von 10 bis 39 und in y von 10 bis 29, b von 35 bis 44 und von 25 bis 34. Keiner der vier Fälle trifft zu: Die Pixel von (35, 25) bis (39, 29) gehören zu beiden.
2. **Nein.** `a.x + a.w` = 40 ist nicht größer als `b.x` = 40: a liegt ganz links von b. Die beiden stoßen nur aneinander.
3. **Ja.** Ein flaches, breites und ein schmales, hohes Rechteck bilden ein Kreuz. Keine Ecke liegt im anderen Rechteck, trotzdem überlappen sie sich. Die Funktion aus 16.1 erkennt auch diesen Fall.
4. **Nein.** `a.y + a.h` = 20 ist kleiner als `b.y` = 21: a liegt ganz über b.

## Aufgabe 2 – Finde den Fehler (`k16_loesung2.c`)

Die Funktion vergleicht `dx * dx + dy * dy`, also das **Quadrat** des Abstands, mit `r` statt mit `r * r`. Für einen Punkt 20 Pixel rechts ist `dx * dx` = 400, und 400 ≤ 30 ist falsch. Nur Punkte, die höchstens etwa 5 Pixel vom Mittelpunkt entfernt sind, gelten so als „drin“. Richtig ist:

```
return dx * dx + dy * dy <= r * r;
```

```
Serial Monitor
Mitte:     1
5 rechts:  1
20 rechts: 1
40 rechts: 0
```

## Aufgabe 3 – Ein Menü mit Knöpfen (`k16_loesung3.c`)

k16_loesung3.c

```c
// Drei Knoepfe: Der Knopf unter dem Zeiger leuchtet, A waehlt ihn aus
#include "pbc.h"

#define ANZAHL 3

typedef struct {
    int x, y;
    int w, h;
} rechteck;

typedef struct {
    rechteck box;
    char name[12];
} knopf;

bool punkt_im_rechteck(int px, int py, rechteck r) {
    return px >= r.x && px < r.x + r.w && py >= r.y && py < r.y + r.h;
}

int main(void) {
    pbc_init();
    knopf knoepfe[ANZAHL] = {
        { { 50, 50, 140, 40 }, "Spielen" },
        { { 50, 110, 140, 40 }, "Optionen" },
        { { 50, 170, 140, 40 }, "Hilfe" },
    };
    int x = 30, y = 240;                    // die Spitze des Zeigers

    while (true) {
        if (pbc_pressed_left()) {
            x = x - 3;
        }
        if (pbc_pressed_right()) {
            x = x + 3;
        }
        if (pbc_pressed_up()) {
            y = y - 3;
        }
        if (pbc_pressed_down()) {
            y = y + 3;
        }
        bool a = pbc_was_pressed_a();

        pbc_clear(PBC_BLACK);
        for (int i = 0; i < ANZAHL; i++) {
            rechteck k = knoepfe[i].box;
            if (punkt_im_rechteck(x, y, k)) {
                pbc_fill_rect(k.x, k.y, k.w, k.h, PBC_ORANGE);
```

```
49                 if (a) {
50                     printf("Gewaehlt: %s\n", knoepfe[i].name);
51                 }
52             } else {
53                 pbc_rect(k.x, k.y, k.w, k.h, PBC_ORANGE);
54             }
55             pbc_text18(knoepfe[i].name, k.x + 20, k.y + 8, PBC_WHITE);
56         }
57         pbc_fill_circle(x, y, 3, PBC_WHITE);
58         pbc_show();
59         sleep_ms(20);
60     }
61 }
```

- Zeile 17: Ein Punkt liegt im Rechteck, wenn er rechts vom linken Rand (`>=`) und links vom rechten Rand (`<`) liegt, und ebenso für oben und unten.
- Zeilen 22–26: Jeder Knopf hat ein `rechteck` und einen Namen. Bei der Anlage stehen die Werte des Rechtecks in eigenen geschweiften Klammern.
- Zeile 42 fragt `pbc_was_pressed_a()` **einmal** vor der Schleife ab und merkt sich das Ergebnis in `a`. Würde man es innerhalb der Schleife für jeden Knopf fragen, hätte schon der erste Knopf den Druck „verbraucht“.




Nach einem Druck auf A über „Optionen“ steht im seriellen Monitor:

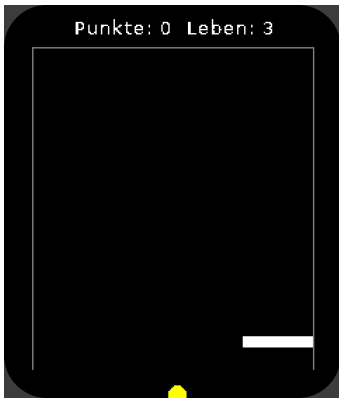
Serial Monitor
```
Gewaehlt: Optionen
```

## Aufgabe 4 – Ball und Schläger (`k16_loesung4.c`)

k16_loesung4.c
```
37 int main(void) {
38     pbc_init();
39     rechteck schlaeger = { 95, 236, 50, 8 };
40     kugel ball = { 120.0f, 60.0f, 3.0f, 3.5f };
41     int leben = 3;
42     int punkte = 0;
43     char text[40];
44 
45     while (true) {
46         if (leben > 0) {
47             if (pbc_pressed_left() && schlaeger.x > LINKS) {
48                 schlaeger.x = schlaeger.x - 5;
49             }
```

```
50            if (pbc_pressed_right() && schlaeger.x + schlaeger.w < RECHTS) {
51                schlaeger.x = schlaeger.x + 5;
52            }
53            ball.x = ball.x + ball.vx;
54            ball.y = ball.y + ball.vy;
55            if (ball.x - RADIUS < LINKS || ball.x + RADIUS > RECHTS) {
56                ball.vx = -ball.vx;
57                ball.x = ball.x + ball.vx;
58            }
59            if (ball.y - RADIUS < OBEN) {
60                ball.vy = -ball.vy;
61                ball.y = ball.y + ball.vy;
62            }
63            // Der Ball als Rechteck um den Kreis herum
64            rechteck box = { (int)ball.x - RADIUS, (int)ball.y - RADIUS,
65                             2 * RADIUS, 2 * RADIUS };
66            if (ball.vy > 0 && rechtecke_beruehren(box, schlaeger)) {
67                ball.vy = -ball.vy;         // nur, wenn er faellt
68                punkte++;
69                pbc_tone(660, 40);
70            }
71            if (ball.y > PBC_HEIGHT) {          // unten durchgelassen
72                leben--;
73                pbc_tone(150, 300);
74                ball.x = 120.0f;
75                ball.y = 60.0f;
76            }
77        }
```

- Die Zeilen 1–35 enthalten die Konstanten, die Strukturen `rechteck` und `kugel`, `bildtakt()` und `rechtecke_beruehren()`.
- Zeilen 64–65: das Rechteck um den Ball, 2 × `RADIUS` breit und hoch.
- Zeile 66: Der Ball prallt nur ab, wenn er gerade **fällt** (`vy > 0`). Sonst könnte er im Schläger hängen bleiben: Im nächsten Bild überlappt er noch, `vy` würde wieder umgedreht, und der Ball zappelte.
- Zeilen 71–76: Unten hinaus kostet ein Leben; der Ball startet wieder oben.

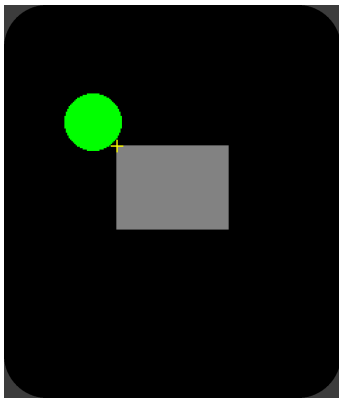



## Aufgabe 5 – Kreis gegen Rechteck (`k16_loesung5.c`)

k16_loesung5.c

```
11 // Begrenzt wert auf den Bereich von min bis max.
12 int begrenzen(int wert, int min, int max) {
13     if (wert < min) {
14         return min;
15     }
```

```
16      if (wert > max) {
17          return max;
18      }
19      return wert;
20  }
21
22  bool kreis_beruehrt_rechteck(int mx, int my, int r, rechteck b) {
23      int nx = begrenzen(mx, b.x, b.x + b.w - 1);   // naechster Punkt
24      int ny = begrenzen(my, b.y, b.y + b.h - 1);
25      int dx = mx - nx;
26      int dy = my - ny;
27      return dx * dx + dy * dy <= r * r;
28  }
```



- `begrenzen` liefert den Wert selbst, wenn er im Bereich liegt, sonst die nächste Grenze.
- Zeilen 23–24: Der nächste Punkt des Rechtecks hat als x den begrenzten x-Wert des Mittelpunkts und als y den begrenzten y-Wert. Liegt der Mittelpunkt im Rechteck, ist es der Mittelpunkt selbst, und der Abstand ist 0.
- Zeilen 25–27: danach wie Punkt im Kreis.

Im Bild ist der Kreis schräg vor der Ecke grün, obwohl ein Rechteck um den Kreis die Kiste schon berühren würde. Das gelbe Kreuz ist der nächste Punkt, hier die Ecke der Kiste. Rückt der Kreis noch etwas näher, wird er rot.

ANHANG C – LÖSUNGEN

# Lösungen zu Kapitel 17: Der Neigungssensor

## Aufgabe 1 – Wie liegt das Gerät?

```
Serial Monitor
1: liegt, Display oben
2: liegt, Display unten
3: steht aufrecht
4: schraeg oder auf der Seite
```

1. Flach, Display oben: z ≈ +1 g, also größer als 0,8.
2. Display unten: z ≈ −1 g.
3. Aufrecht mit dem Joystick unten: Die y-Achse zeigt nach unten, y ≈ −1 g.
4. Auf der rechten Kante, um 45° gekippt: x ≈ 0,7 g und z ≈ 0,7 g. Weder z noch y erreicht die Grenze von 0,8 g, also „schraeg oder auf der Seite“. Stünde das Gerät ganz auf der rechten Kante, wäre x ≈ −1 g; auch das prüft das Programm nicht eigens ab.

Die Ausgabe stammt aus einem Probelauf mit genau diesen vier Lagen. Auf dem Gerät musst du es selbst ausprobieren.

### Aufgabe 2 – Kipp-Ampel (`k17_loesung2.c`)

k17_loesung2.c

```c
// Kipp-Ampel: Die RGB-LED zeigt, zu welcher Seite das Geraet kippt
#include "pbc.h"

#define GRENZE 0.25f

int main(void) {
    pbc_init();
    int zuletzt = -1;               // zuletzt gezeigte Farbe
    while (true) {
        float x = pbc_accel_x_g();
        int farbe = 0;              // 0 waagrecht, 1 links tiefer, 2 rechts
        if (x > GRENZE) {
            farbe = 1;
        } else if (x < -GRENZE) {
            farbe = 2;
        }
        if (farbe != zuletzt) {     // nur bei einer Aenderung schalten
            if (farbe == 0) {
                pbc_rgb_led(0, 255, 0);
            } else if (farbe == 1) {
                pbc_rgb_led(0, 0, 255);
            } else {
                pbc_rgb_led(255, 0, 0);
            }
            zuletzt = farbe;
        }
        sleep_ms(50);
    }
}
```

- Liegt der linke Rand tiefer, zeigt die x-Achse nach oben: x ist positiv (Zeile 12, blau). Liegt der rechte Rand tiefer, ist x negativ (Zeile 14, rot).
- `zuletzt` merkt sich die gezeigte Farbe. Die LED wird nur umgeschaltet, wenn sich etwas ändert (Zeile 17). Am Anfang ist `zuletzt` −1, damit die erste Farbe auf jeden Fall gesetzt wird.

### Aufgabe 3 – Schüttelwürfel (`k17_loesung3.c`)

k17_loesung3.c

```c
float betrag(float wert) {
    if (wert < 0) {
        return -wert;
    }
    return wert;
}
```

```c
// Ein Auge des Wuerfels; spalte und zeile von 0 bis 2.
void auge(int spalte, int zeile) {
    pbc_fill_circle(80 + spalte * 40, 100 + zeile * 40, 10, PBC_BLACK);
}

void zeichne_wuerfel(int zahl) {
    pbc_clear(pbc_rgb(30, 60, 120));
    pbc_fill_rect(50, 70, 140, 140, PBC_WHITE);
    if (zahl == 1 || zahl == 3 || zahl == 5) {
        auge(1, 1);                                  // Mitte
    }
    if (zahl >= 2) {
        auge(0, 0);
        auge(2, 2);
    }
    if (zahl >= 4) {
        auge(2, 0);
        auge(0, 2);
    }
    if (zahl == 6) {
        auge(0, 1);
        auge(2, 1);
    }
    pbc_text12("Schuetteln oder A", 66, 230, PBC_WHITE);
    pbc_show();
}
```

- `betrag` macht aus einer negativen Zahl eine positive.
- `auge` zeichnet einen Punkt in einem Raster aus 3 × 3 Plätzen.
- Die Augen einer Würfelzahl setzen sich aus Gruppen zusammen: Die Mitte haben 1, 3 und 5. Zwei diagonale Ecken haben alle ab 2, die anderen beiden Ecken alle ab 4. Nur die 6 hat zusätzlich die beiden mittleren Punkte an den Seiten.

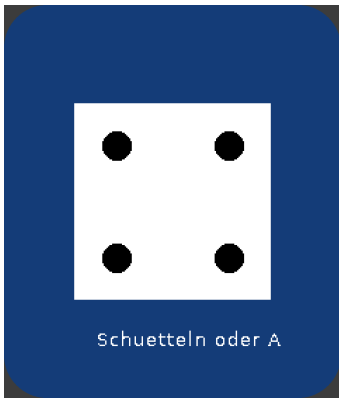
k17_loesung3.c

```c
int main(void) {
    pbc_init();
    float alt_x = pbc_accel_x_g();
    float alt_y = pbc_accel_y_g();
    float alt_z = pbc_accel_z_g();
    zeichne_wuerfel(pbc_random(1, 6));

    while (true) {
        float x = pbc_accel_x_g();
        float y = pbc_accel_y_g();
        float z = pbc_accel_z_g();
        float aenderung = betrag(x - alt_x) + betrag(y - alt_y) +
                          betrag(z - alt_z);
        alt_x = x;
        alt_y = y;
        alt_z = z;
```

```
57          if (aenderung > SCHUETTELN || pbc_was_pressed_a()) {
58              int zahl = pbc_random(1, 6);
59              zeichne_wuerfel(zahl);
60              printf("Gewuerfelt: %d\n", zahl);
61              pbc_tone(600, 50);
62              sleep_ms(500);          // nicht gleich noch einmal wuerfeln
63              alt_x = pbc_accel_x_g();
64              alt_y = pbc_accel_y_g();
65              alt_z = pbc_accel_z_g();
66          }
67          sleep_ms(30);
68      }
69  }
```

- Zeilen 51–52: Die Unterschiede zur vorigen Messung werden ohne Vorzeichen addiert. Liegt das Gerät still, ist die Summe nahe 0. Beim Schütteln wird sie groß.
- Zeile 62: Nach dem Würfeln wartet das Programm eine halbe Sekunde. Sonst würde ein einziges Schütteln mehrmals hintereinander würfeln.
- Zeilen 63–65: Danach werden die alten Werte neu gemessen, damit die Bewegung während der Pause nicht als neues Schütteln zählt.




## Aufgabe 4 – Mittelwert der letzten acht (`k17_loesung4.c`)

k17_loesung4.c

```
1  // Mittelwert der letzten 8 Messungen
2  #include "pbc.h"
3
4  #define ANZAHL 8
5
6  float werte[ANZAHL];            // die letzten Messwerte
7  int naechster = 0;              // hier kommt der naechste Wert hin
8
9  float mittelwert(void) {
10     float summe = 0.0f;
11     for (int i = 0; i < ANZAHL; i++) {
12         summe = summe + werte[i];
13     }
14     return summe / ANZAHL;
15 }
16
17 int main(void) {
18     pbc_init();
19     for (int i = 0; i < ANZAHL; i++) {
20         werte[i] = pbc_accel_x_g();
21     }
22     printf("Nr.    roh  mittel\n");
```

```
    for (int i = 1; i <= 16; i++) {
        float roh = pbc_accel_x_g();
        roh = roh + pbc_random(-10, 10) / 100.0f;   // kuenstliches Zittern
        werte[naechster] = roh;                     // den aeltesten ersetzen
        naechster = (naechster + 1) % ANZAHL;
        printf("%3d  %5.2f  %5.2f\n", i, roh, mittelwert());
        sleep_ms(100);
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Nr.    roh  mittel
  1   0.07   0.01
  2   0.08   0.02
  3   0.00   0.02
  4  -0.05   0.01
  5   0.07   0.02
  6   0.02   0.02
  7   0.05   0.03
  8   0.07   0.04
  9   0.47   0.09
 10   0.59   0.15
 11   0.46   0.21
 12   0.44   0.27
 13   0.46   0.32
 14   0.46   0.38
 15   0.56   0.44
 16   0.47   0.49
```

- Zeilen 19–21: Zu Beginn wird das Array mit dem ersten Messwert gefüllt. Sonst stünden dort Nullen, und der Mittelwert wäre anfangs falsch.
- Zeilen 26–27: `naechster` zeigt auf den ältesten Wert. Er wird ersetzt, dann rückt `naechster` eins weiter; nach 7 kommt wieder 0. So ein Array, das sich im Kreis füllt, heißt **Ringpuffer**.

Das künstliche Zittern ist dasselbe wie in `k17_glaetten.c`. Der Mittelwert erreicht die neuen 0,5 g nach genau acht Messungen (Messung 16), denn dann liegen nur noch neue Werte im Array. Die Glättung aus 17.2 ist dort erst bei 0,41. Dafür braucht der Mittelwert ein Array und mehr Rechenarbeit.

## Aufgabe 5 – Münzen einsammeln (`k17_loesung5.c`)

k17_loesung5.c

```
int main(void) {
    pbc_init();
    float x = 120.0f, y = 140.0f;
    float vx = 0.0f, vy = 0.0f;
    int mx = pbc_random(LINKS + 10, RECHTS - 10);
```

```
    int my = pbc_random(OBEN + 10, UNTEN - 10);
    int punkte = 0;
    char text[20];

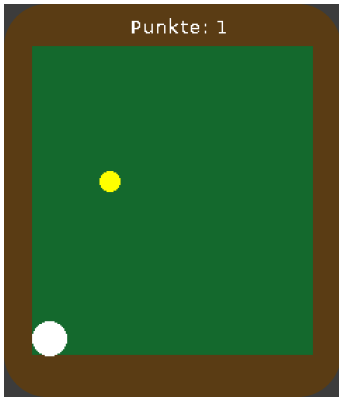
    while (true) {
        // Neigung wie beim Tablett, der Joystick schubst zusaetzlich
        float ax = -pbc_accel_x_g();
        float ay = -pbc_accel_y_g();
        if (pbc_pressed_left()) {
            ax = ax - 0.3f;
        }
        if (pbc_pressed_right()) {
            ax = ax + 0.3f;
        }
        if (pbc_pressed_up()) {
            ay = ay - 0.3f;
        }
        if (pbc_pressed_down()) {
            ay = ay + 0.3f;
        }
        vx = (vx + ax * KRAFT) * REIBUNG;
        vy = (vy + ay * KRAFT) * REIBUNG;
        x = x + vx;
        y = y + vy;
        if (x - RADIUS < LINKS) {
            x = LINKS + RADIUS;
            vx = -vx * 0.5f;
        }
        if (x + RADIUS > RECHTS) {
            x = RECHTS - RADIUS;
            vx = -vx * 0.5f;
        }
        if (y - RADIUS < OBEN) {
            y = OBEN + RADIUS;
            vy = -vy * 0.5f;
        }
        if (y + RADIUS > UNTEN) {
            y = UNTEN - RADIUS;
            vy = -vy * 0.5f;
        }

        if (kreise_beruehren(x, y, RADIUS, mx, my, MUENZE)) {
            punkte++;
            pbc_tone(1200, 60);
            mx = pbc_random(LINKS + 10, RECHTS - 10);
            my = pbc_random(OBEN + 10, UNTEN - 10);
        }

        pbc_clear(pbc_rgb(90, 60, 30));
        pbc_fill_rect(LINKS, OBEN, RECHTS - LINKS, UNTEN - OBEN,
                      pbc_rgb(30, 110, 50));
        pbc_fill_circle(mx, my, MUENZE, PBC_YELLOW);
        pbc_fill_circle((int)x, (int)y, RADIUS, PBC_WHITE);
```

```
92          snprintf(text, sizeof text, "Punkte: %d", punkte);
93          pbc_text12(text, 90, 8, PBC_WHITE);
94          pbc_show();
95          bildtakt();
96      }
97  }
```

- Die Zeilen 1–33 enthalten die Konstanten, `bildtakt()` und `kreise_beruehren()` aus Kapitel 16.
- Zeilen 45–58: Die Beschleunigung kommt aus der Neigung (mit umgedrehtem Vorzeichen wie bei der Kugel in 17.3). Der Joystick gibt zusätzlich 0,3 dazu, so als wäre das Gerät um etwa 17° gekippt.
- Zeile 80: Kugel und Münze sind Kreise. Berühren sie sich, gibt es einen Punkt, und die Münze springt an eine neue Stelle.




ANHANG C – LÖSUNGEN

# Lösungen zu Kapitel 18: Projekt: Ein ganzes Spiel

## Aufgabe 1 – Einfacher und schwerer

Alle Stellen sind in `k18_spiel.c`:

- a) Zeile 9: `#define START_LEBEN 5`. Die Anzeige oben rechts zeichnet einen roten Punkt pro Leben; fünf passen nebeneinander.
- b) Zeile 5: `#define ANZAHL 8`. Das Array `dinge` und alle Schleifen benutzen `ANZAHL`, sonst ist nichts zu ändern.
- c) Zeile 204: `sekunden / 40.0f` statt `sekunden / 20.0f`. Passe auch den Kommentar in Zeile 199 an.
- d) Zeile 205: `pbc_random(1, 4) == 1` ist nur in einem Viertel der Fälle wahr.

Gerade bei b) merkst du: Weil die Zahl nur an einer Stelle steht, reicht eine einzige Änderung. Das ist der Vorteil von Konstanten.

## Aufgabe 2 – Pause (`k18_loesung2.c`)

Gegenüber `k18_spiel.c` ändert sich wenig: Das `enum` bekommt den Zustand `PAUSE` (Zeile 138), eine neue Variable `pause_seit` merkt sich den Beginn der Pause (Zeile 165), und der `switch` im Hauptprogramm bekommt einen weiteren `case`:

k18_loesung2.c

```
380         switch (modus) {
381             case TITEL:
382                 titel();
```

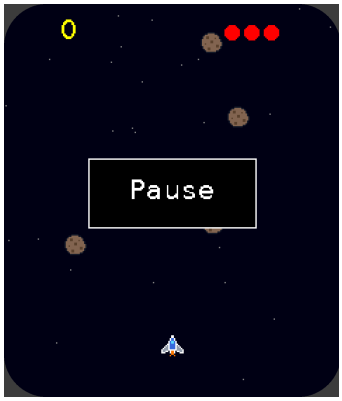
```
383                 break;
384             case SPIEL:
385                 spiel();
386                 break;
387             case PAUSE:
388                 pause();
389                 break;
390             case ENDE:
391                 ende();
392                 break;
393         }
```

In `spiel()` wechselt B in die Pause, und eine neue Funktion `pause()` führt den Zustand aus:

k18_loesung2.c

```
304     if (leben == 0) {
305         spiel_beenden();
306     } else if (pbc_was_pressed_b()) {
307         pause_seit = pbc_millis();
308         modus = PAUSE;
309     }
310 }
311
312 void pause(void) {
313     spiel_zeichnen();                       // das Spiel steht still
314     pbc_fill_rect(60, 110, 120, 50, PBC_BLACK);
315     pbc_rect(60, 110, 120, 50, PBC_WHITE);
316     text_mitte18("Pause", 122, PBC_WHITE);
317     if (pbc_was_pressed_b()) {
318         // Die Pause zaehlt nicht zur Spielzeit: Zeitpunkte verschieben
319         uint32_t dauer = pbc_millis() - pause_seit;
320         spiel_start = spiel_start + dauer;
321         schutz_bis = schutz_bis + dauer;
322         modus = SPIEL;
323     }
324 }
```

- Zeile 313: `pause()` zeichnet das Spiel, bewegt aber nichts. Darum steht alles still.
- Zeilen 319–321: Während der Pause ist die Uhr weitergelaufen. Würde man nichts tun, wäre das Spiel nach einer langen Pause plötzlich schneller (`spiel_start` liegt weiter zurück), und die Schutzzeit nach einem Treffer wäre abgelaufen. Deshalb werden beide Zeitpunkte um die Dauer der Pause nach hinten verschoben.




Im Kippmodus hat B im Titel das Spiel gestartet. Im Spiel selbst ist B frei, deshalb passt es gut für die Pause.

### Aufgabe 3 – Bestenliste (`k18_loesung3.c`)

Statt `int highscore` gibt es ein Array (Zeile 160):

```
int bestenliste[PLAETZE];       // absteigend sortiert, nur im RAM
```

Globale Arrays beginnen in C mit lauter Nullen. Das Einsortieren übernimmt eine eigene Funktion:

k18_loesung3.c

```
222 // Sortiert wert in die Bestenliste ein, wenn es fuer einen Platz reicht.
223 void eintragen(int wert) {
224     for (int platz = 0; platz < PLAETZE; platz++) {
225         if (wert > bestenliste[platz]) {
226             // alle ab hier eins nach unten schieben, der letzte faellt raus
227             for (int i = PLAETZE - 1; i > platz; i--) {
228                 bestenliste[i] = bestenliste[i - 1];
229             }
230             bestenliste[platz] = wert;
231             return;
232         }
233     }
234 }
235
236 void spiel_beenden(void) {
237     neuer_rekord = punkte > bestenliste[0];
238     eintragen(punkte);
239     ende_seit = pbc_millis();
240     pbc_tone(110, 600);
241     modus = ENDE;
242 }
```

- Zeile 225: Die Liste wird von oben durchsucht. Der erste Platz, dessen Wert kleiner ist als das neue Ergebnis, ist der richtige.
- Zeilen 227–229: Ab diesem Platz rücken alle um eins nach unten. Die Schleife läuft **rückwärts**, vom letzten Platz aus. Liefe sie vorwärts, würde sie jeden Wert mit seinem Vorgänger überschreiben, bevor er verschoben ist.
- Zeile 231: Nach dem Einsortieren ist die Funktion fertig. `return` beendet sie sofort, auch mitten in der Schleife.
- Zeile 237: Ein neuer Rekord ist es nur, wenn das Ergebnis besser ist als Platz 1.

k18_loesung3.c

```
329     for (int platz = 0; platz < PLAETZE; platz++) {
330         snprintf(text, sizeof text, "%d.  %d", platz + 1, bestenliste[platz]);
331         text_mitte18(text, 130 + platz * 22, PBC_WHITE);
332     }
```

Das Titelbild zeigt die drei Plätze untereinander. Das Bild zeigt die Liste nach dem ersten Spiel.




## Aufgabe 4 – Extraleben (`k18_loesung4.c`)

Das Herz wird mit `bild2c.html` umgewandelt und steht als viertes Bild in der Datei. Statt `bool ist_stern` bekommt jedes Ding eine Art:

k18_loesung4.c

```
187 typedef enum {
188     METEOR,
189     STERN,
190     HERZ
191 } art;
192
193 typedef struct {
194     float x, y;                 // linke obere Ecke des Bilds
195     float tempo;                // Pixel pro Bild nach unten
196     art was;                    // Meteor, Stern oder Herz
197 } ding;
```

In `ding_neu` wird die Art ausgelost: Bei einer Zufallszahl von 1 bis 30 ist 1 ein Herz, 2 bis 10 ein Stern, der Rest ein Meteor.

k18_loesung4.c

```
252     int los = pbc_random(1, 30);
253     if (los == 1) {
254         dinge[i].was = HERZ;                        // selten
255     } else if (los <= 10) {
256         dinge[i].was = STERN;
257     } else {
258         dinge[i].was = METEOR;
259     }
```

Beim Zusammenstoß gibt es nun drei Fälle. Das Herz bringt ein Leben, aber nicht mehr als `MAX_LEBEN` (Zeile 10: 5):

k18_loesung4.c

```
313             if (dinge[i].was == STERN) {
314                 punkte++;
315                 pbc_tone(1320, 50);
316                 ding_neu(i);
```

```
317             } else if (dinge[i].was == HERZ) {
318                 if (leben < MAX_LEBEN) {
319                     leben++;
320                 }
321                 pbc_tone(880, 150);
322                 ding_neu(i);
323             } else if (pbc_millis() >= schutz_bis) {
324                 leben--;
325                 schutz_bis = pbc_millis() + SCHUTZ_MS;
326                 pbc_tone(150, 250);
327                 ding_neu(i);
328             }
```

Zum Zeichnen wählt ein `switch` das passende Bild:

k18_loesung4.c

```
336     for (int i = 0; i < ANZAHL; i++) {
337         int x = (int)dinge[i].x;
338         int y = (int)dinge[i].y;
339         switch (dinge[i].was) {
340             case METEOR:
341                 pbc_draw_image(&meteor, x, y);
342                 break;
343             case STERN:
344                 pbc_draw_image(&stern, x, y);
345                 break;
346             case HERZ:
347                 pbc_draw_image(&herz, x, y);
348                 break;
349         }
350     }
```

Auf die Art greift das Programm an drei Stellen zu. Wer später eine vierte Art hinzufügt, etwa einen Schild, muss alle drei Stellen anpassen. Ein `switch` über ein `enum` hilft dabei: Fehlt ein `case`, warnt der Compiler mit `-Wall` (*enumeration value not handled in switch*).

ANHANG C – LÖSUNGEN

# Lösungen zu Kapitel 19: Zeiger und Speicher

## Aufgabe 1 – Wer zeigt wohin?

Serial Monitor

```
a = 15, b = 14, *p = 14
```

Zuerst zeigt `p` auf `a`. `*p = *p + 10` macht aus `a` also 15. Dann zeigt `p` auf `b`, und `*p = *p * 2` verdoppelt `b` auf 14. Am Ende zeigt `p` noch immer auf `b`, deshalb ist `*p` ebenfalls 14.

## Aufgabe 2 – Verdoppeln (`k19_loesung2.c`)

k19_loesung2.c

```c
#include "pbc.h"

void verdopple(int *zahl) {
    *zahl = *zahl * 2;
}

int main(void) {
    pbc_init();
    int wert = 3;
    for (int i = 0; i < 5; i++) {
        verdopple(&wert);
        printf("%d\n", wert);
    }

    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
6
12
24
48
96
```

Beim Aufruf steht `&wert`: Die Funktion braucht die Adresse, um das Original zu ändern.

## Aufgabe 3 – Zwei Ergebnisse (`k19_loesung3.c`)

k19_loesung3.c

```c
#include "pbc.h"

// zwei Ergebnisse: Sie landen dort, wohin kleinstes und groesstes zeigen
void min_max(const int *werte, int anzahl, int *kleinstes, int *groesstes) {
    *kleinstes = werte[0];
    *groesstes = werte[0];
    for (int i = 1; i < anzahl; i++) {
        if (werte[i] < *kleinstes) {
            *kleinstes = werte[i];
        }
        if (werte[i] > *groesstes) {
            *groesstes = werte[i];
        }
    }
}

int main(void) {
    pbc_init();
```

```c
    int temperaturen[7] = { 12, 15, 9, 17, 21, 14, 11 };
    int tief, hoch;
    min_max(temperaturen, 7, &tief, &hoch);
    printf("Tiefstwert %d, Hoechstwert %d\n", tief, hoch);

    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Tiefstwert 9, Hoechstwert 21
```

`tief` und `hoch` brauchen keinen Startwert, denn `min_max` schreibt beide über die Zeiger hinein, bevor sie gelesen werden. Das Array wird mit seinem Namen übergeben, also als Zeiger auf das erste Element; deshalb muss die Anzahl 7 mit.

## Aufgabe 4 – Finde den Fehler (`k19_loesung4.c`)

Terminal

```
Built target bs2_default
Built target bs2_default_library
Built target pbc
Building C object main.c.o
main.c: In function 'main':
main.c:16:14: error: passing argument 1 of 'begrenze' makes pointer from integer
  without a cast [-Wint-conversion]
   16 |     begrenze(x, 0, 239);
      |              ^
      |              |
      |              int
main.c:4:20: note: expected 'int *' but argument is of type 'int'
    4 | void begrenze(int *wert, int min, int max) {
      |               ~~~~~^~~~
```

*Makes pointer from integer without a cast* heißt: An dieser Stelle wird ein Zeiger erwartet (`int *`, siehe die `note:`-Zeile), übergeben wird aber eine ganze Zahl (`int`). Es fehlt das `&`: Die Funktion braucht die Adresse von `x`, nicht seinen Wert.

k19_loesung4.c

```c
    pbc_init();
    int x = 300;
    begrenze(&x, 0, 239);
    printf("x = %d\n", x);
```

Serial Monitor

```
x = 239
```

### Aufgabe 5 – Platz für Bilder (`k19_loesung5.c`)

k19_loesung5.c

```c
#include "pbc.h"

int main(void) {
    pbc_init();
    int bild = PBC_WIDTH * PBC_HEIGHT * (int)sizeof(uint16_t);
    int ram = 512 * 1024;
    int flash = 16 * 1024 * 1024;

    printf("Ein ganzes Bild: %d Byte\n", bild);
    printf("40 Bilder:       %d Byte\n", 40 * bild);
    printf("Ins RAM passen:   %d Bilder\n", ram / bild);
    printf("In den Flash:     %d Bilder\n", flash / bild);

    while (true) {
        sleep_ms(1000);
    }
}
```

Serial Monitor

```
Ein ganzes Bild: 134400 Byte
40 Bilder:       5376000 Byte
Ins RAM passen:   3 Bilder
In den Flash:     124 Bilder
```

Die 40 Bilder brauchen gut 5 MB. Ins RAM passen nicht einmal vier Bilder, und dort sind ja auch noch der Bildpuffer und alle anderen Variablen. In den Flash passen alle 40 bequem. Die Bilder müssen deshalb mit `const` angelegt werden, so wie es das Umwandlungswerkzeug aus Kapitel 15 tut. Ohne `const` würde das Programm jedes Bild zusätzlich ins RAM kopieren wollen, und der Linker würde beim Bauen `region `RAM' overflowed` melden.

ANHANG C – LÖSUNGEN

## Lösungen zu Kapitel 20: Mehrere Dateien und eigene Bibliotheken

### Aufgabe 1 – Was gehört wohin?

| Zeile | Datei | Grund |
|---|---|---|
| 1. `void spiele_melodie(int nummer);` | `musik.h` | ein Prototyp, also eine Deklaration |
| 2. `void spiele_melodie(int nummer) { … }` | `musik.c` | die Definition mit Rumpf |
| 3. `#ifndef MUSIK_H` | `musik.h` | Beginn des Include-Wächters |

| Zeile | Datei | Grund |
|---|---|---|
| 4. `#define ANZAHL_MELODIEN 4` | `musik.h` | wenn auch `main.c` die Zahl braucht; sonst `musik.c` |
| 5. `static const int melodie_1[] = …;` | `musik.c` | Daten, die nur die Bibliothek benutzt (`static`) |
| 6. `extern int lautstaerke;` | `musik.h` | kündigt die Variable an |
| 7. `int lautstaerke = 5;` | `musik.c` | die einzige Definition der Variable |

## Aufgabe 2 – Ein Auto für die Figuren (`projekte/k20_loesung2`)

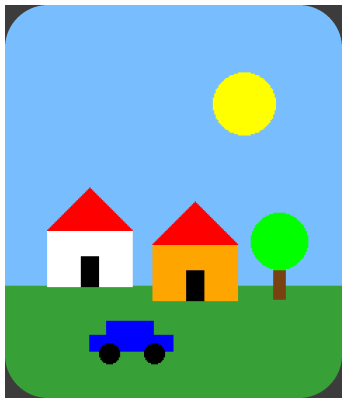
Zu ändern sind `figuren.h` (Prototyp), `figuren.c` (Definition) und `main.c` (Aufruf). Die `CMakeLists.txt` bleibt, wie sie ist: `figuren.c` steht schon in `add_executable`.

figuren.h

```
19 // Auto; (x, y) ist die linke obere Ecke, 60 x 30 Pixel
20 void zeichne_auto(int x, int y, uint16_t farbe);
```

figuren.c

```
32 void zeichne_auto(int x, int y, uint16_t farbe) {
33     pbc_fill_rect(x + 12, y, 34, 12, farbe);          // Dach
34     pbc_fill_rect(x, y + 10, 60, 12, farbe);          // Karosserie
35     pbc_fill_circle(x + 14, y + 23, 7, PBC_BLACK);    // Raeder
36     pbc_fill_circle(x + 46, y + 23, 7, PBC_BLACK);
37     anzahl_figuren++;
38 }
```



main.c

```
11     zeichne_baum(195, 180);
12     zeichne_auto(60, 225, PBC_BLUE);
13     pbc_show();
```

## Aufgabe 3 – Meldungen lesen

`undefined reference` kommt vom Linker. Mögliche Ursachen:

1. `spieler.c` steht nicht in `add_executable` der `CMakeLists.txt`.

2. In `spieler.c` ist die Funktion anders geschrieben, etwa `spieler_bewegn` oder `Spieler_bewegen`.
3. Die Funktion ist in `spieler.c` mit `static` definiert und deshalb außerhalb der Datei unbekannt; oder es gibt sie nur als Prototyp in `spieler.h`, aber gar nicht in `spieler.c`.

Fehlt `#include "spieler.h"` in `main.c`, meldet schon der Compiler für `main.c` den Fehler `implicit declaration of function 'spieler_bewegen'`, wie in Abschnitt 20.1 bei den Figuren.

### Aufgabe 4 – Den Würfel auslagern (`projekte/k20_loesung4`)

In `wuerfel.h` gehören der Prototyp von `zeichne_wuerfel` und die Größe, denn auch `main.c` kann sie zum Anordnen brauchen. Die Hilfsfunktion `auge` bleibt mit `static` in `wuerfel.c`. Der Name `GROESSE` ist in einer Header-Datei zu allgemein: Jede Datei, die `wuerfel.h` einbindet, bekommt ihn, und ein anderes `GROESSE` würde damit zusammenstoßen. Darum heißt er jetzt `WUERFEL_GROESSE`. Es ist üblich, alle Namen einer Bibliothek mit ihrem Namen beginnen zu lassen, wie `pbc_` bei der Buchbibliothek.

wuerfel.h

```c
// wuerfel.h – einen Spielwuerfel zeichnen
#ifndef WUERFEL_H
#define WUERFEL_H

// Kantenlaenge eines Wuerfels in Pixeln
#define WUERFEL_GROESSE 70

// zeichnet einen Wuerfel mit 1 bis 6 Augen; (x, y) ist die Ecke links oben
void zeichne_wuerfel(int x, int y, int augen);

#endif
```

wuerfel.c

```c
// wuerfel.c – so wird ein Wuerfel gezeichnet
#include "pbc.h"
#include "wuerfel.h"

static void auge(int x, int y, int spalte, int zeile) {
    pbc_fill_circle(x + 15 + spalte * 20, y + 15 + zeile * 20, 6, PBC_BLACK);
}

// zeichnet einen Wuerfel mit 1 bis 6 Augen; (x, y) ist die Ecke links oben
void zeichne_wuerfel(int x, int y, int augen) {
    pbc_fill_rect(x, y, WUERFEL_GROESSE, WUERFEL_GROESSE, PBC_WHITE);
    if (augen % 2 == 1) {
        auge(x, y, 1, 1);                          // Mitte
    }
    if (augen >= 2) {
        auge(x, y, 0, 0);
        auge(x, y, 2, 2);
    }
```

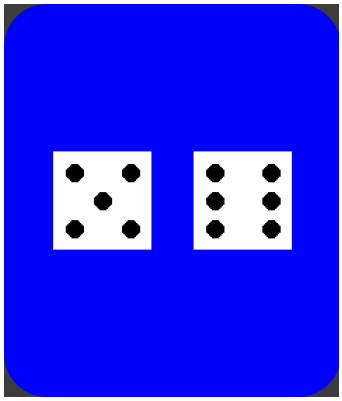
```
    if (augen >= 4) {
        auge(x, y, 2, 0);
        auge(x, y, 0, 2);
    }
    if (augen == 6) {
        auge(x, y, 0, 1);
        auge(x, y, 2, 1);
    }
}
```

main.c

```
#include "pbc.h"
#include "wuerfel.h"

int main(void) {
    pbc_init();
    pbc_clear(PBC_BLUE);
    zeichne_wuerfel(35, 105, 5);
    zeichne_wuerfel(135, 105, 6);
    pbc_show();

    while (true) {
        sleep_ms(1000);
    }
}
```



In der `CMakeLists.txt` steht:

```
add_executable(mein_programm main.c wuerfel.c)
```

## Aufgabe 5 – Detektiv in `pbc`

1. In `pbc_geraet.c` steht `#define PIN_DISPLAY_LICHT 26`: an GP26. `pbc_backlight` stellt dort die PWM ein (Kapitel 21.3).
2. `pbc_rgb` in `pbc_zeichnen.c` begrenzt jeden Wert zuerst: Ist `r` größer als 255, wird es 255. Rot 300 ergibt also dasselbe wie Rot 255.
3. 8 Pixel: `pbc_text_width` rechnet `8 * (int)strlen(s)`, und `pbc_text` setzt `x` nach jedem Zeichen um 8 weiter (`x += 8`).
4. `pbc_init` ruft nacheinander `stdio_init_all`, dreimal `led_init`, `pbc_tone_off`, `tasten_init`, `rgb_init`, `pbc_rgb_led_off`, `sensor_init`, `akku_init`, `display_init`, `pbc_clear`, `pbc_show` und `pbc_backlight` auf. Das Licht kommt zuletzt, weil das Display nach dem Einschalten zufälligen Inhalt zeigt. Erst wenn ein schwarzes Bild übertragen ist, wird es hell; so sieht man keinen „Pixelmüll“. Das steht auch im Kommentar in der letzten Zeile von `pbc_init`.

# Lösungen zu Kapitel 21: GPIO, PWM und ADC

Die Programme dieses Kapitels laufen nur auf dem PicoBoy. Probiere sie dort aus.

## Aufgabe 1 – Ampel ohne `pbc` (`k21_loesung1.c`)

k21_loesung1.c

```c
#include "pbc.h"
#include "hardware/gpio.h"

#define PIN_LED_ROT   14
#define PIN_LED_GELB  13
#define PIN_LED_GRUEN 12

void ampel(bool rot, bool gelb, bool gruen, int ms) {
    gpio_put(PIN_LED_ROT, rot);
    gpio_put(PIN_LED_GELB, gelb);
    gpio_put(PIN_LED_GRUEN, gruen);
    sleep_ms(ms);
}

int main(void) {
    pbc_init();
    while (true) {
        ampel(true, false, false, 3000);      // rot
        ampel(true, true, false, 1000);       // rot und gelb
        ampel(false, false, true, 3000);      // gruen
        ampel(false, true, false, 1000);      // gelb
    }
}
```

Die Funktion `ampel` stellt alle drei LEDs auf einmal und wartet dann. Ein `bool` lässt sich direkt an `gpio_put` übergeben: `true` ist 1 (HIGH), `false` ist 0 (LOW). `pbc_init` hat die drei Pins schon als Ausgänge eingerichtet.

## Aufgabe 2 – Taste B direkt (`k21_loesung2.c`)

B hängt an **GP28** (Tabelle in 21.1 oder `PIN_TASTE_B` in `pbc_geraet.c`).

k21_loesung2.c

```c
#include "pbc.h"
#include "hardware/gpio.h"

#define PIN_TASTE_B  28
#define PIN_LED_GELB 13

int main(void) {
    pbc_init();
```

```
    while (true) {
        // aktiv LOW: gedrueckt = 0, also umdrehen
        gpio_put(PIN_LED_GELB, !gpio_get(PIN_TASTE_B));
        sleep_ms(5);
    }
}
```

Weil die Taste aktiv LOW ist, dreht `!` den gelesenen Wert um.

## Aufgabe 3 – Atmende LED (`k21_loesung3.c`)

k21_loesung3.c

```c
#include "pbc.h"
#include "hardware/pwm.h"

#define PIN_LED_GELB 13

int main(void) {
    pbc_init();
    gpio_set_function(PIN_LED_GELB, GPIO_FUNC_PWM);
    uint slice = pwm_gpio_to_slice_num(PIN_LED_GELB);   // GP13: Slice 6, Kanal B
    pwm_set_wrap(slice, 9999);
    pwm_set_enabled(slice, true);

    while (true) {
        for (int i = 0; i <= 100; i++) {          // heller werden
            pwm_set_gpio_level(PIN_LED_GELB, i * i);
            sleep_ms(15);
        }
        for (int i = 100; i >= 0; i--) {          // dunkler werden
            pwm_set_gpio_level(PIN_LED_GELB, i * i);
            sleep_ms(15);
        }
        sleep_ms(400);
    }
}
```

`i * i` läuft von 0 bis 10 000, genau bis `wrap + 1`: Dann ist die LED ganz an. Am Anfang wächst `level` langsam (1, 4, 9, …), am Ende schnell. Weil das Auge kleine Helligkeiten viel feiner unterscheidet als große, wirkt das gleichmäßiger als ein linearer Verlauf, der die meiste Zeit „fast hell“ aussieht. GP13 ist Kanal B von Slice 6; `pwm_set_gpio_level` sucht den Kanal selbst heraus.

## Aufgabe 4 – Tonleiter (`k21_loesung4.c`)

k21_loesung4.c

```c
#include "pbc.h"
#include "hardware/clocks.h"
#include "hardware/pwm.h"

#define PIN_LAUTSPRECHER 15
```

```c
void ton(int frequenz) {
    uint slice = pwm_gpio_to_slice_num(PIN_LAUTSPRECHER);
    pwm_set_clkdiv(slice, clock_get_hz(clk_sys) / 1000000.0f);
    uint32_t schritte = 1000000 / frequenz;
    pwm_set_wrap(slice, schritte - 1);
    pwm_set_gpio_level(PIN_LAUTSPRECHER, schritte / 2);
    gpio_set_function(PIN_LAUTSPRECHER, GPIO_FUNC_PWM);
    pwm_set_enabled(slice, true);
}

int main(void) {
    pbc_init();
    // C-Dur-Tonleiter: c d e f g a h c
    const int tonleiter[8] = { 262, 294, 330, 349, 392, 440, 494, 523 };
    while (true) {
        for (int i = 0; i < 8; i++) {
            ton(tonleiter[i]);
            sleep_ms(250);
        }
        for (int i = 7; i >= 0; i--) {
            ton(tonleiter[i]);
            sleep_ms(250);
        }
        pbc_tone_off();
        sleep_ms(2000);
    }
}
```

Die zweite Schleife zählt rückwärts von 7 bis 0. Weil `ton` nur umschaltet und nicht wartet, bestimmt das `sleep_ms(250)` die Länge jedes Tons.

## Aufgabe 5 – Wie warm ist der Chip? (`k21_loesung5.c`)

k21_loesung5.c

```c
#include "pbc.h"
#include "hardware/adc.h"

int main(void) {
    pbc_init();
    adc_set_temp_sensor_enabled(true);
    while (true) {
        adc_select_input(4);                    // Eingang 4: Temperatursensor
        float spannung = adc_read() * 3.3f / 4095.0f;
        float grad = 27.0f - (spannung - 0.706f) / 0.001721f;

        char text[32];
        pbc_clear(PBC_BLACK);
        snprintf(text, sizeof text, "%.3f V", spannung);
        pbc_text18(text, 70, 100, PBC_WHITE);
```

```
        snprintf(text, sizeof text, "%.1f Grad C", grad);
        pbc_text24(text, 45, 140, PBC_ORANGE);
        pbc_show();
        sleep_ms(1000);
    }
}
```

Eingang 4 wird in jedem Durchlauf neu gewählt. Das schadet nicht und ist sicher, falls zwischendurch etwas anderes (etwa `pbc_battery_voltage`) Eingang 3 gewählt hat. Der Sensor misst die Temperatur **im Chip**, nicht die der Luft. Sie liegt meist etwas über der Raumtemperatur und steigt langsam, wenn du den PicoBoy in der Hand hältst. Sehr genau ist der Sensor nicht: Schon ein Fehler von 1 mV in der Spannung macht etwa ein halbes Grad aus.

---

ANHANG C – LÖSUNGEN

# Lösungen zu Kapitel 22: I2C

Die Programme dieses Kapitels laufen nur auf dem PicoBoy.

## Aufgabe 1 – Zahlen umrechnen

| hexadezimal | dezimal | Bitmuster |
|---|---|---|
| `0x18` | 1 · 16 + 8 = 24 | `00011000` |
| `0x87` | 8 · 16 + 7 = 135 | `10000111` |

Jede hexadezimale Ziffer steht für genau 4 Bit: 1 = `0001`, 8 = `1000`, 7 = `0111`. Darum liest man Bitmuster so leicht aus Hexzahlen ab.

`0x0400` ist 1024. Nach `>> 4` bleibt 1024 / 16 = 64 übrig, also `0x0040`. Bei 1024 Stufen pro g sind das 64 / 1024 = 0,0625 g.

## Aufgabe 2 – Den Messbereich nachlesen (`k22_loesung2.c`)

k22_loesung2.c

```c
#include "pbc.h"
#include "hardware/i2c.h"

#define SENSOR 0x18

int main(void) {
    pbc_init();
    uint8_t reg = 0x0F;                         // Messbereich
    uint8_t wert = 0;
    i2c_write_blocking(i2c0, SENSOR, &reg, 1, true);
```

```
    int r = i2c_read_blocking(i2c0, SENSOR, &wert, 1, false);
    printf("Register 0x0F: 0x%02X (Ergebnis %d)\n", wert, r);

    while (true) {
        sleep_ms(1000);
    }
}
```

`sensor_init` in `pbc_geraet.c` schreibt beim Start `0x03` in Register 0x0F (±2 g). Genau das sollte die Konsole zeigen. Steht dort eine negative Zahl als Ergebnis, hat der Sensor nicht geantwortet.

### Aufgabe 3 – ±8 g (`k22_loesung3.c`)

k22_loesung3.c

```c
#include "pbc.h"
#include "hardware/i2c.h"

#define SENSOR 0x18

int main(void) {
    pbc_init();
    // Messbereich auf +-8 g stellen: Register 0x0F bekommt 0x08
    uint8_t befehl[2] = { 0x0F, 0x08 };
    i2c_write_blocking(i2c0, SENSOR, befehl, 2, false);

    while (true) {
        uint8_t reg = 0x06;
        uint8_t b[2];
        i2c_write_blocking(i2c0, SENSOR, &reg, 1, true);
        i2c_read_blocking(i2c0, SENSOR, b, 2, false);
        int wert = (int16_t)((b[1] << 8) | (b[0] & 0xF0)) >> 4;
        // +-8 g mit 12 Bit: 4096 Stufen fuer 16 g, also 256 Stufen pro g
        printf("z: %5d -> %.2f g\n", wert, wert / 256.0f);
        sleep_ms(500);
    }
}
```

Bei ±8 g verteilen sich die 4096 Stufen auf 16 g: **256 Stufen pro g**. Der Sensor misst gröber, kann dafür aber auch heftige Stöße messen, etwa wenn das Gerät herunterfällt oder kräftig geschüttelt wird. Bei ±2 g bleibt der Wert ab 2 g einfach am Rand stehen. Denk daran: Die `pbc_accel_…`-Funktionen rechnen weiter mit ±2 g und liefern jetzt viermal zu kleine Werte.

### Aufgabe 4 – Schüttelzähler (`k22_loesung4.c`)

k22_loesung4.c

```c
#include "pbc.h"
#include "hardware/i2c.h"
#include <math.h>

#define SENSOR 0x18
```

```c
#define SCHWELLE 1.8f            // ab hier gilt es als Schuetteln (in g)

float staerke(void) {
    uint8_t reg = 0x02;
    uint8_t b[6];
    i2c_write_blocking(i2c0, SENSOR, &reg, 1, true);
    if (i2c_read_blocking(i2c0, SENSOR, b, 6, false) != 6) {
        return 0.0f;
    }
    float summe = 0.0f;
    for (int i = 0; i < 3; i++) {
        float g = ((int16_t)((b[2 * i + 1] << 8) | (b[2 * i] & 0xF0)) >> 4) /
          1024.0f;
        summe += g * g;
    }
    return sqrtf(summe);         // Betrag der Beschleunigung
}

int main(void) {
    pbc_init();
    int zaehler = 0;
    bool geschuettelt = false;
    while (true) {
        float s = staerke();
        if (s > SCHWELLE && !geschuettelt) {
            zaehler++;
            geschuettelt = true;
        }
        if (s < 1.2f) {          // erst nach ruhigerer Phase wieder zaehlen
            geschuettelt = false;
        }
        char text[24];
        snprintf(text, sizeof text, "%d", zaehler);
        pbc_clear(PBC_BLACK);
        pbc_text18("Schuettel mich!", 55, 80, PBC_WHITE);
        pbc_text24(text, 110, 130, PBC_YELLOW);
        pbc_show();
        sleep_ms(10);
    }
}
```

Damit ein Schütteln nur einmal zählt, merkt sich `geschuettelt`, dass die Schwelle schon überschritten wurde. Erst wenn die Stärke wieder unter 1,2 g fällt, darf der nächste Stoß zählen. Zwei Schwellen statt einer verhindern, dass ein Wert, der um 1,8 g herum zittert, viele Zählungen auslöst. Die Schwellen sind Erfahrungswerte; probiere am Gerät, was sich gut anfühlt.

### Aufgabe 5 – Am Stück oder einzeln? (`k22_loesung5.c`)

k22_loesung5.c

```c
#include "pbc.h"
#include "hardware/i2c.h"

#define SENSOR 0x18
#define DURCHGAENGE 100

int lese_register(uint8_t reg, uint8_t *puffer, size_t anzahl) {
    int r = i2c_write_blocking(i2c0, SENSOR, &reg, 1, true);
    if (r < 0) {
        return r;
    }
    return i2c_read_blocking(i2c0, SENSOR, puffer, anzahl, false);
}

int main(void) {
    pbc_init();
    uint8_t b[6];
    while (true) {
        uint32_t start = time_us_32();
        for (int n = 0; n < DURCHGAENGE; n++) {
            lese_register(0x02, b, 6);                 // alle 6 Bytes am Stueck
        }
        uint32_t am_stueck = time_us_32() - start;

        start = time_us_32();
        for (int n = 0; n < DURCHGAENGE; n++) {
            for (int i = 0; i < 6; i++) {
                lese_register(0x02 + i, &b[i], 1);     // jedes Byte einzeln
            }
        }
        uint32_t einzeln = time_us_32() - start;

        printf("am Stueck: %lu us, einzeln: %lu us (je Messung)\n",
               (unsigned long)(am_stueck / DURCHGAENGE), (unsigned long)(einzeln
                 / DURCHGAENGE));
        sleep_ms(1000);
    }
}
```

Jeder Aufruf von `lese_register` kostet einen Start, die Adresse, die Registernummer, einen Neustart, noch einmal die Adresse und am Ende einen Stopp, und erst dazwischen kommen die Daten. Bei 400 000 Takten pro Sekunde dauert ein Bit 2,5 Mikrosekunden, ein Byte mit Bestätigung 9 Takte. Am Stück fällt dieser Aufwand einmal an, einzeln sechsmal. Das Lesen am Stück sollte deshalb mehrere Male schneller sein. Wie groß der Unterschied genau ist, zeigt dir die Konsole am Gerät.

# Lösungen zu Kapitel 23: SPI, DMA und das Display

## Aufgabe 1 – Rechnen mit dem Bus

Ein Bild hat 240 × 280 = 67 200 Pixel zu je 2 Bytes, also **134 400 Bytes** oder 1 075 200 Bits.

| SPI-Takt | Dauer eines Bilds | höchstens Bilder pro Sekunde |
|---|---|---|
| 37,5 MHz | 1 075 200 / 37 500 000 s ≈ 28,7 ms | 1000 / 28,7 ≈ 34 |
| 75 MHz | 1 075 200 / 75 000 000 s ≈ 14,3 ms | 1000 / 14,3 ≈ 69 |

In Wirklichkeit sind es etwas weniger Bilder, denn das Programm braucht auch Zeit zum Zeichnen, und vor jedem Bild kommen noch die Befehle für das Fenster. 75 MHz bekäme man mit `spi_init(spi0, 75000000)`: 150 MHz / 2. Ob das Panel so schnell mitkommt, müsste man am Gerät ausprobieren.

## Aufgabe 2 – Ein Farbverlauf ohne Bildpuffer (`k23_loesung2.c`)

k23_loesung2.c

```c
int main(void) {
    pbc_init();
    for (int y = 0; y < PBC_HEIGHT; y++) {
        int anteil = y * 255 / (PBC_HEIGHT - 1);   // 0 oben, 255 unten
        fuellen(0, y, PBC_WIDTH, 1, pbc_rgb(anteil, 0, 255 - anteil));
    }
    while (true) {
        sleep_ms(1000);
    }
}
```

Die Zeilen 1 bis 45 sind dieselben wie in `k23_fenster.c`. Jede Zeile des Displays ist ein Fenster mit 240 × 1 Pixeln. `anteil` wächst von 0 (oben) bis 255 (unten); Rot nimmt damit zu und Blau ab.

## Aufgabe 3 – Display schlafen legen (`k23_loesung3.c`)

k23_loesung3.c

```c
// Lösung 23.3: Das Display schlafen legen (B) und wecken (A)
#include "pbc.h"
#include "hardware/spi.h"

#define PIN_DISPLAY_CS 10
#define PIN_DISPLAY_DC 8

#define BEFEHL_SLPIN   0x10   // Schlafmodus an
#define BEFEHL_SLPOUT  0x11   // Schlafmodus aus
#define BEFEHL_DISPOFF 0x28
#define BEFEHL_DISPON  0x29
```

```c
static void befehl(uint8_t code) {
    spi_set_format(spi0, 8, SPI_CPOL_0, SPI_CPHA_0, SPI_MSB_FIRST);
    gpio_put(PIN_DISPLAY_CS, 0);
    gpio_put(PIN_DISPLAY_DC, 0);
    spi_write_blocking(spi0, &code, 1);
    gpio_put(PIN_DISPLAY_CS, 1);
}

int main(void) {
    pbc_init();
    pbc_clear(PBC_PURPLE);
    pbc_text18("B: schlafen", 50, 110, PBC_WHITE);
    pbc_text18("A: aufwachen", 45, 145, PBC_WHITE);
    pbc_show();
    while (true) {
        if (pbc_was_pressed_b()) {
            pbc_backlight(0);          // das Licht braucht den meisten Strom
            befehl(BEFEHL_DISPOFF);
            befehl(BEFEHL_SLPIN);
            pbc_led_green(true);       // zeigt: PicoBoy läuft noch
        }
        if (pbc_was_pressed_a()) {
            befehl(BEFEHL_SLPOUT);
            sleep_ms(120);             // Zeit zum Aufwachen
            befehl(BEFEHL_DISPON);
            pbc_backlight(255);
            pbc_led_green(false);
        }
        sleep_ms(20);
    }
}
```

Die Reihenfolge ist wichtig: erst das Licht aus, dann die Anzeige, dann schlafen; beim Aufwachen umgekehrt. Die grüne LED zeigt, dass das Programm weiterläuft, auch wenn der Bildschirm dunkel ist. Das Bild im Speicher des Controllers bleibt im Schlafmodus erhalten: Nach dem Aufwachen ist es sofort wieder da, ohne `pbc_show()`.

### Aufgabe 4 – `pbc_clear` per DMA (`k23_loesung4.c`)

k23_loesung4.c

```c
// Lösung 23.4: Den Bildpuffer per DMA mit einer Farbe füllen
#include "pbc.h"
#include "pbc_intern.h"
#include "hardware/dma.h"

static int kanal;

static void dma_clear(uint16_t farbe) {
    static uint16_t quelle;   // muss noch da sein, während der DMA liest
    quelle = farbe;
    dma_channel_config c = dma_channel_get_default_config(kanal);
```

```
12      channel_config_set_transfer_data_size(&c, DMA_SIZE_16);
13      channel_config_set_read_increment(&c, false);   // immer derselbe Wert
14      channel_config_set_write_increment(&c, true);   // Ziel wandert weiter
15      dma_channel_configure(kanal, &c, pbc_bildpuffer, &quelle,
16                            PBC_WIDTH * PBC_HEIGHT, true);
17      dma_channel_wait_for_finish_blocking(kanal);
18  }
19
20  int main(void) {
21      pbc_init();
22      kanal = dma_claim_unused_channel(true);
23
24      uint32_t start = time_us_32();
25      pbc_clear(PBC_RED);
26      printf("pbc_clear: %d us\n", (int)(time_us_32() - start));
27
28      start = time_us_32();
29      dma_clear(PBC_GREEN);
30      printf("dma_clear: %d us\n", (int)(time_us_32() - start));
31
32      pbc_text18("Gruen per DMA", 50, 125, PBC_BLACK);
33      pbc_show();
34      while (true) {
35          sleep_ms(1000);
36      }
37  }
```

- Zeile 13: Die Quelle wandert **nicht** weiter. Der DMA liest 67 200-mal denselben Wert und schreibt ihn jedes Mal an die nächste Stelle des Bildpuffers.
- Zeile 9: Die Quelle ist `static`. Eine gewöhnliche lokale Variable läge auf dem Stack; das geht hier nur gut, weil `dma_clear` auf das Ende wartet, bevor die Funktion zurückkehrt. Mit `static` ist man auf der sicheren Seite, falls man das Warten einmal weglässt.
- Welche Zeit schneller ist, zeigt die Ausgabe im seriellen Monitor. Auch hier gilt: Wartet das Programm nicht auf den DMA, kann die CPU in der Zwischenzeit anderes tun.

ANHANG C – LÖSUNGEN

# Lösungen zu Kapitel 24: Interrupts, Timer und der zweite Kern

## Aufgabe 1 – Lauflicht im Hintergrund (`k24_loesung1.c`)

k24_loesung1.c

```
1  // Lösung 24.1: Ein Lauflicht, das ein Timer weiterschaltet
2  #include "pbc.h"
3
4  static int schritt = 0;
```

```c
static bool weiter(repeating_timer_t *timer) {
    (void)timer;
    pbc_led_red(schritt == 0);
    pbc_led_yellow(schritt == 1);
    pbc_led_green(schritt == 2);
    schritt = (schritt + 1) % 3;
    return true;
}

int main(void) {
    pbc_init();
    static repeating_timer_t timer;
    add_repeating_timer_ms(-300, weiter, NULL, &timer);

    pbc_text18("Lauflicht", 75, 110, PBC_WHITE);
    pbc_text12("laeuft im Timer-Interrupt", 40, 150, PBC_GRAY);
    pbc_show();
    while (true) {
        sleep_ms(1000);
    }
}
```

`schritt == 0` ist `true` oder `false`, genau das, was `pbc_led_red` braucht. So schaltet jeder Aufruf eine LED ein und die beiden anderen aus. `schritt` braucht kein `volatile`, denn nur der Timer benutzt die Variable.

### Aufgabe 2 – Warum geht es nicht weiter? (`k24_loesung2.c`)

`zaehler` fehlt das Wort `volatile`. In der Schleife

```c
while (zaehler < 100) {
    // nichts tun, nur warten
}
```

ändert nichts den Zähler, jedenfalls nicht, soweit der Compiler sehen kann. Er darf den Wert deshalb **einmal** lesen und annehmen, dass er gleich bleibt. Das sieht man am Maschinencode, den der Compiler aus der Schleife gemacht hat (gezeigt mit dem Werkzeug `arm-none-eabi-objdump`):

Maschinencode ohne volatile

```
100002ba: ldr     r3, [pc, #48]   @ (100002ec <main+0x68>)
100002bc: ldr     r3, [r3, #0]
100002be: cmp     r3, #99 @ 0x63
100002c0: bgt.n   100002c4 <main+0x40>
100002c2: b.n     100002c2 <main+0x3e>
```

`ldr` liest den Zähler aus dem Speicher, `cmp` vergleicht ihn mit 99, `bgt` springt hinter die Schleife, wenn er größer ist. Sonst folgt `b.n 100002c2`: ein Sprung auf **sich selbst**, für immer. Den Zähler liest das Programm nie wieder. Mit `volatile` sieht die Schleife so aus:

Maschinencode mit volatile

```
100002ba: ldr     r2, [pc, #48]   @ (100002ec <main+0x68>)
100002bc: ldr     r3, [r2, #0]
100002be: cmp     r3, #99 @ 0x63
100002c0: ble.n   100002bc <main+0x38>
```

Jetzt springt `ble` zurück zum zweiten `ldr`: Der Zähler wird in jeder Runde neu aus dem Speicher gelesen. Die einzige Änderung im Programm ist Zeile 4:

k24_loesung2.c

```
4 static volatile int zaehler = 0;
```

So ein Fehler ist tückisch: Mit einer anderen Optimierungsstufe des Compilers, oder wenn in der Schleife zufällig ein Funktionsaufruf steht, läuft das Programm ohne `volatile` vielleicht doch. Darauf darf man sich nicht verlassen.

## Aufgabe 3 – Ein Regenbogen auf Kern 1 (Projekt `k24_loesung3`)

main.c

```
 1 // Lösung 24.3: Kern 1 lässt die RGB-LED durch die Farben wandern
 2 #include "pbc.h"
 3 #include "pico/multicore.h"
 4
 5 static void kern1(void) {
 6     while (true) {
 7         for (int i = 0; i < 256; i++) {   // Rot -> Grün
 8             pbc_rgb_led(255 - i, i, 0);
 9             sleep_ms(8);
10         }
11         for (int i = 0; i < 256; i++) {   // Grün -> Blau
12             pbc_rgb_led(0, 255 - i, i);
13             sleep_ms(8);
14         }
15         for (int i = 0; i < 256; i++) {   // Blau -> Rot
16             pbc_rgb_led(i, 0, 255 - i);
17             sleep_ms(8);
18         }
19     }
20 }
21
22 int main(void) {
23     pbc_init();
24     multicore_launch_core1(kern1);
25     int zahl = 0;
26     while (true) {
27         char text[20];
```

```
        snprintf(text, sizeof text, "%d", zahl);
        pbc_clear(PBC_BLACK);
        pbc_text12("Kern 0 zaehlt:", 75, 100, PBC_GRAY);
        pbc_text24(text, 95, 125, PBC_WHITE);
        pbc_show();
        zahl++;
    }
}
```

Wie in allen Projekten mit zweitem Kern braucht die `CMakeLists.txt` die Bibliothek `pico_multicore`. Die RGB-LED gehört hier allein Kern 1; Kern 0 benutzt nur das Display.

## Aufgabe 4 – Musik auf Befehl (Projekt `k24_loesung4`)

main.c

```c
// Lösung 24.4: Kern 1 spielt die Melodie, die Kern 0 ihm schickt
#include "pbc.h"
#include "pico/multicore.h"
#include "pico/util/queue.h"

typedef struct {
    int frequenz;   // 0 = Pause
    int dauer;
} note;

static const note tonleiter[] = {
    {262, 200}, {294, 200}, {330, 200}, {349, 200},
    {392, 200}, {440, 200}, {494, 200}, {523, 400}, {0, 400},
};
static const note sirene[] = {
    {600, 300}, {800, 300},
};
#define ANZAHL(a) (int)(sizeof a / sizeof a[0])

enum { STILLE, TONLEITER, SIRENE };

static queue_t befehle;

static void kern1(void) {
    int stueck = STILLE;
    int i = 0;
    while (true) {
        if (queue_try_remove(&befehle, &stueck)) {
            i = 0;   // neuer Befehl: von vorn beginnen
        }
        const note *n = NULL;
        if (stueck == TONLEITER) {
            n = &tonleiter[i % ANZAHL(tonleiter)];
        } else if (stueck == SIRENE) {
            n = &sirene[i % ANZAHL(sirene)];
        }
```

```
37          if (n == NULL) {
38              pbc_tone_off();
39              sleep_ms(20);
40              continue;
41          }
42          if (n->frequenz > 0) {
43              pbc_tone(n->frequenz, 0);
44          }
45          sleep_ms(n->dauer);
46          pbc_tone_off();
47          i++;
48      }
49  }
```

`main` (ab Zeile 51, hier nicht abgedruckt) schickt bei A, B oder der Mitte des Joysticks eine Zahl in die Warteschlange und wartet 20 ms. Die Arbeit macht Kern 1:

- Zeile 28: `queue_try_remove` wartet **nicht**. Ist ein Befehl da, steht er danach in `stueck`, und das neue Stück beginnt mit der ersten Note. Ist keiner da, bleibt `stueck`, wie es war.
- Weil Kern 1 die Schlange vor **jeder** Note fragt, wechselt die Musik spätestens nach einer Note.
- `n` ist ein Zeiger auf die aktuelle Note (Kapitel 19). `n->frequenz` ist dasselbe wie `(*n).frequenz`. `i % ANZAHL(…)` lässt die Melodie von vorn beginnen, wenn sie zu Ende ist.

ANHANG C – LÖSUNGEN

# Lösungen zu Kapitel 25: Daten dauerhaft speichern

## Aufgabe 1 – Wie oft eingeschaltet? (`k25_loesung1.c`)

k25_loesung1.c

```c
// Lösung 25.1: Wie oft wurde der PicoBoy schon eingeschaltet?
#include "pbc.h"
#include "hardware/flash.h"
#include "pico/flash.h"

#define SPEICHER_OFFSET (PICO_FLASH_SIZE_BYTES - 2 * FLASH_SECTOR_SIZE)
#define KENNUNG 0x53544152u

static void flash_schreiben(void *daten) {
    flash_range_erase(SPEICHER_OFFSET, FLASH_SECTOR_SIZE);
    flash_range_program(SPEICHER_OFFSET, daten, FLASH_PAGE_SIZE);
}

int main(void) {
    pbc_init();
    const uint32_t *flash = (const uint32_t *)(XIP_BASE + SPEICHER_OFFSET);
    uint32_t starts = 0;
```

```
18      if (flash[0] == KENNUNG) {
19          starts = flash[1];
20      }
21      starts++;
22
23      static uint32_t seite[FLASH_PAGE_SIZE / 4];   // 256 Bytes als 64 Zahlen
24      memset(seite, 0xFF, sizeof seite);
25      seite[0] = KENNUNG;
26      seite[1] = starts;
27      flash_safe_execute(flash_schreiben, seite, 100);
28
29      char text[30];
30      snprintf(text, sizeof text, "Start Nr. %d", (int)starts);
31      pbc_text24(text, 35, 120, PBC_WHITE);
32      pbc_show();
33      while (true) {
34          sleep_ms(1000);
35      }
36  }
```

- Zeile 16: Diesmal liest ein Zeiger auf `uint32_t` den Flash, also gleich vier Bytes auf einmal. `flash[0]` ist die Kennung, `flash[1]` der Zähler.
- Zeile 23: Die Seite ist hier ein Array aus 64 Zahlen zu je 4 Bytes, also ebenfalls 256 Bytes. `flash_safe_execute` gibt den Zeiger als `void *` an `flash_schreiben` weiter, und ein `void *` passt zu jedem Zeigertyp, auch zu den Bytes, die `flash_range_program` erwartet.
- Eine Prüfsumme fehlt hier der Kürze halber. Fällt der Strom beim Schreiben aus, beginnt der Zähler wieder bei 1.

Achtung: Dieses Programm schreibt bei **jedem** Start in den Flash. Für einen Startzähler ist das in Ordnung, so oft schaltet niemand ein.

### Aufgabe 2 – Wie lange hält der Sektor?

- Alle 10 Sekunden: 100 000 × 10 s = 1 000 000 s. Ein Tag hat 86 400 s, das sind also nur gut **11 Tage**, wenn das Programm ununterbrochen läuft.
- Zehn neue Rekorde am Tag: 100 000 / 10 = 10 000 Tage, mehr als **27 Jahre**.

Der Unterschied zeigt, warum man nur speichert, wenn sich etwas Wichtiges geändert hat.

### Aufgabe 3 – Rekord zurücksetzen (`k25_loesung3.c`)

Neu sind die Funktion `flash_loeschen` und der Anfang von `main`:

k25_loesung3.c

```
34  static void flash_loeschen(void *daten) {
35      (void)daten;
36      flash_range_erase(SPEICHER_OFFSET, FLASH_SECTOR_SIZE);
37  }
```

k25_loesung3.c

```c
int main(void) {
    pbc_init();
    if (pbc_pressed_b()) {              // B beim Einschalten gehalten?
        flash_safe_execute(flash_loeschen, NULL, 100);
        pbc_text18("Rekord geloescht", 40, 130, PBC_RED);
        pbc_show();
        sleep_ms(1500);
    }
    int rekord = highscore_lesen();
    while (true) {
```

`pbc_pressed_b()` fragt, ob B **gerade jetzt** gedrückt ist, also schon beim Einschalten. Nach dem Löschen stehen im Sektor nur `0xFF`-Bytes; die Kennung passt nicht mehr, und `highscore_lesen` liefert 0. Ein Spielstand mit dem Wert 0 muss also gar nicht geschrieben werden.

## Aufgabe 4 – Den Sektor schonen (`k25_loesung4.c`)

k25_loesung4.c

```c
// Lösung 25.4: Den Sektor schonen – jeder Rekord in die nächste Seite
#include "pbc.h"
#include "hardware/flash.h"
#include "pico/flash.h"

#define SPEICHER_OFFSET (PICO_FLASH_SIZE_BYTES - 2 * FLASH_SECTOR_SIZE)
#define SEITEN (FLASH_SECTOR_SIZE / FLASH_PAGE_SIZE)   // 16 Seiten je Sektor
#define KENNUNG 0x50424332u

typedef struct {
    uint32_t kennung;
    uint32_t highscore;
} eintrag;

typedef struct {
    uint32_t offset;     // wohin
    bool loeschen;       // vorher den Sektor löschen?
    uint8_t *daten;      // eine Seite im RAM
} auftrag;

static const eintrag *seite_im_flash(int nr) {
    return (const eintrag *)(XIP_BASE + SPEICHER_OFFSET + nr * FLASH_PAGE_SIZE);
}

// Die letzte beschriebene Seite finden; -1: keine
static int letzte_seite(void) {
    int letzte = -1;
    for (int nr = 0; nr < (int)SEITEN; nr++) {
        if (seite_im_flash(nr)->kennung == KENNUNG) {
            letzte = nr;
        }
    }
```

```
33      return letzte;
34  }
35
36  static void flash_arbeit(void *p) {
37      auftrag *a = p;
38      if (a->loeschen) {
39          flash_range_erase(SPEICHER_OFFSET, FLASH_SECTOR_SIZE);
40      }
41      flash_range_program(a->offset, a->daten, FLASH_PAGE_SIZE);
42  }
43
44  static void speichern(int punkte) {
45      static uint8_t seite[FLASH_PAGE_SIZE];
46      eintrag e = { KENNUNG, (uint32_t)punkte };
47      memset(seite, 0xFF, sizeof seite);
48      memcpy(seite, &e, sizeof e);
49      int nr = letzte_seite() + 1;
50      auftrag a = { 0, false, seite };
51      if (nr >= (int)SEITEN) {         // Sektor voll: erst jetzt löschen
52          nr = 0;
53          a.loeschen = true;
54      }
55      a.offset = SPEICHER_OFFSET + nr * FLASH_PAGE_SIZE;
56      flash_safe_execute(flash_arbeit, &a, 100);
57      printf("Rekord %d in Seite %d gespeichert%s\n", punkte, nr,
58             a.loeschen ? " (Sektor vorher geloescht)" : "");
59  }
```

- `letzte_seite` (Zeilen 26 bis 34) sucht die letzte Seite mit gültiger Kennung. Gelöschte Seiten enthalten `0xFFFFFFFF` und zählen nicht.
- `speichern` schreibt in die Seite dahinter. Nur wenn alle 16 Seiten voll sind, löscht es vorher den Sektor und beginnt wieder bei Seite 0.
- Die Struktur `auftrag` bündelt, was `flash_arbeit` wissen muss; über den `void *`-Parameter von `flash_safe_execute` kommt ein Zeiger darauf an.

So wird der Sektor nur bei jedem 16. Speichern gelöscht. Er hält also **16-mal so lange**. Seiten werden dabei nur einmal je Löschung beschrieben, so wie es sein soll.

`main` zeigt den Rekord an; A erhöht ihn um eins und speichert. Im seriellen Monitor siehst du, in welche Seite gespeichert wurde.

# Lösungen zu Kapitel 26: Dein eigenes Projekt

## Aufgabe 1 – Eine Idee in einem Satz

Hier gibt es keine einzig richtige Lösung. Ein Beispiel: „Eine Schlange frisst Punkte, wird mit jedem Punkt länger und darf sich selbst nicht berühren." Gebraucht werden Tasten (Kapitel 9), Arrays für die Glieder der Schlange (Kapitel 12), Strukturen für die Positionen (Kapitel 13), Kollisionen (Kapitel 16), Zufall für den Punkt (Kapitel 5.6) und das Gerüst mit Zuständen (Kapitel 18 und 26.2).

## Aufgabe 2 – Der Plan

Ein Plan für Snake könnte so aussehen:

- **Zustände:** Titel → (A) → Spiel → (Schlange stößt an) → Ende → (B) → Titel.
- **Daten:** Array von Positionen für die Glieder (höchstens zum Beispiel 200), aktuelle Länge, Richtung, Position des Punkts, Punkte, Highscore.
- **Schritte:** 1. ein Quadrat, das sich ständig in eine Richtung bewegt; 2. Richtung mit dem Joystick ändern; 3. Bewegung in festen Kästchen (Raster von 10 Pixeln); 4. ein Punkt an zufälliger Stelle, der beim Fressen neu erscheint; 5. die Schlange als Array, wächst beim Fressen; 6. Ende beim Stoß an Rand oder an sich selbst; 7. Titel und Ende; 8. Highscore im Flash; 9. Töne; 10. schneller werden.

## Aufgabe 3 – Der verschwundene Ball (`k26_loesung3.c`)

In der zweiten Bedingung steht `x > 250` statt `y > 250`:

k26_aufgabe3.c

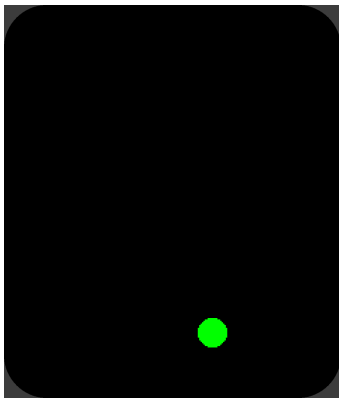
```
16          if (y < 30 || x > 250) {
17              dy = -dy;
18          }
```

Am unteren Rand müsste `y` geprüft werden. `x` wird nie größer als 212 (an der 210 dreht die erste Bedingung um), deshalb wird die Bedingung unten nie wahr. Oben klappt es, weil der Teil `y < 30` richtig ist; links und rechts, weil die erste Bedingung stimmt. Der Fehler fällt dem Compiler nicht auf: `x > 250` ist ein gültiger Vergleich. So sieht es richtig aus:

k26_loesung3.c

```c
// Lösung 26.3: Der Ball prallt jetzt an allen Rändern ab
#include "pbc.h"

int main(void) {
    pbc_init();
    int x = 120;
    int y = 140;
    int dx = 4;
    int dy = 3;
    while (true) {
        x += dx;
        y += dy;
        if (x < 30 || x > 210) {
            dx = -dx;
        }
        if (y < 30 || y > 250) {
            dy = -dy;
        }
        pbc_clear(PBC_BLACK);
        pbc_fill_circle(x, y, 10, PBC_GREEN);
        pbc_show();
    }
}
```



## Aufgabe 4 – Dein Projekt

Auch hier gibt es keine Musterlösung. Ein Tagebucheintrag könnte lauten: „Schritt 5: Die Schlange wächst. Fehler: Nach dem dritten Punkt erschienen Glieder in der linken oberen Ecke. Gefunden mit `printf` der Länge und der Positionen: Die neuen Glieder hatten die Position (0, 0), weil ich sie beim Wachsen nicht gesetzt hatte. Lösung: Ein neues Glied bekommt die Position des bisher letzten."

# Lösungen zu Kapitel 27: Programmieren mit KI

## Aufgabe 1 – Eine gute Anfrage

Ein Beispiel; am Anfang des Gesprächs stehen wie in 27.2 der Inhalt von `AGENTS.md` und von `pbc/pbc.h`:

```
Schreibe main.c für den PicoBoy Color Plus (RP2350, Pico SDK 2.2.0,
kein Arduino). Benutze nur die Funktionen aus pbc.h oben.

Die drei Status-LEDs sollen eine Ampel sein, in dieser Folge:
rot (3 s), rot und gelb (1 s), grün (3 s), gelb (1 s), dann von vorn.
Auf dem Display steht dazu groß und mittig "Halt" (rot), "Achtung"
(rot-gelb und gelb) oder "Fahren" (grün), ohne Umlaute.

Bitte das ganze main.c mit kurzen deutschen Kommentaren, ohne Warnungen
bei -Wall -Wextra. Erkläre danach in einfachen Worten, wie die Folge der
Phasen gespeichert ist.
```

Darin stecken alle Punkte aus 27.2: Gerät und SDK, die Bibliothek, die Regeln (über `AGENTS.md`), eine genaue Beschreibung mit Zeiten und Texten und was zurückkommen soll.

## Aufgabe 2 – Was stimmt hier nicht? (`k27_loesung2.c`)

1. **Das Array auf dem Stack.** `hintergrund` in Zeile 6 ist eine lokale Variable mit 67 200 Einträgen zu 2 Bytes, also 134 400 Bytes. Der Stack hat 4 KB. Sobald `main` das Array anlegt, läuft der Stack über, und der Wächter hält das Programm an (Kapitel 19.5). Deshalb erscheint kein Bild. Abhilfe: `static` oder außerhalb von `main`.
2. **Die LED an GP25.** Das ist die LED des Raspberry Pi Pico 2. Am PicoBoy ist an GP25 nichts angeschlossen. Die Status-LEDs hängen an GP12 bis GP14 (Anhang D); am einfachsten nimmt man `pbc_led_green`.

k27_loesung2.c

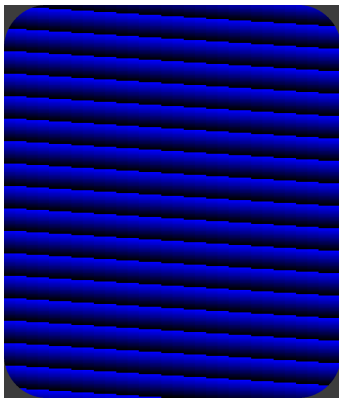
```c
// Lösung 27.2: Array static, LED über pbc
#include "pbc.h"

static uint16_t hintergrund[PBC_WIDTH * PBC_HEIGHT];   // nicht im Stack

int main(void) {
    pbc_init();
    for (int i = 0; i < PBC_WIDTH * PBC_HEIGHT; i++) {
        hintergrund[i] = pbc_rgb(0, 0, i % 256);
    }
```

```
    while (true) {
        for (int y = 0; y < PBC_HEIGHT; y++) {
            for (int x = 0; x < PBC_WIDTH; x++) {
                pbc_pixel(x, y, hintergrund[y * PBC_WIDTH + x]);
            }
        }
        pbc_show();
        pbc_led_green(true);        // LED an: Bild ist fertig
        sleep_ms(500);
        pbc_led_green(false);
        sleep_ms(500);
    }
}
```

Das Programm braucht jetzt 134 400 Bytes zusätzlich im RAM, neben dem Bildpuffer der Bibliothek. Das passt; schöner wäre es, das Muster direkt in den Bildpuffer zu zeichnen, ohne Kopie.



## Aufgabe 3 – Zwei Blöcke (`k27_loesung3.c`)

Aus den beiden Variablen für den Block werden zwei Arrays mit je zwei Einträgen:

k27_loesung3.c

```
    int block_x[2] = { pbc_random(25, 215 - BLOCK_GROESSE),
                       pbc_random(25, 215 - BLOCK_GROESSE) };
    int block_y[2] = { 0, -PBC_HEIGHT / 2 };   // der zweite startet später
```

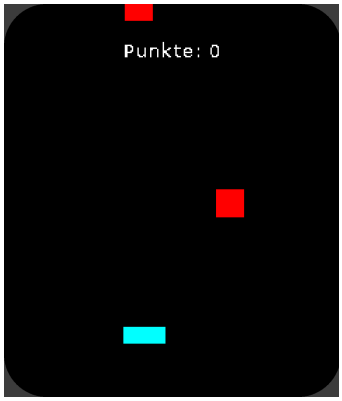
Der zweite Block startet bei y = −140, also eine halbe Bildschirmhöhe **über** dem Bildschirm. `pbc_fill_rect` zeichnet nur den sichtbaren Teil, deshalb erscheint er erst später. Bewegung, Punkte und Treffer laufen in einer Schleife über beide Blöcke:

k27_loesung3.c

```
        for (int i = 0; i < 2; i++) {
            block_y[i] += tempo;
            if (block_y[i] > PBC_HEIGHT) {
                block_y[i] = 0;
                block_x[i] = pbc_random(25, 215 - BLOCK_GROESSE);
                punkte++;
                if (punkte % 5 == 0) {
                    tempo++;
                }
            }
            if (ueberlappen(spieler_x, SPIELER_Y, SPIELER_BREITE, SPIELER_HOEHE,
```

```
                            block_x[i], block_y[i],
                            BLOCK_GROESSE, BLOCK_GROESSE)) {
                vorbei = true;
                pbc_tone(150, 300);
            }
        }
```

Auch beim Neustart (nach „Game over“) und beim Zeichnen werden beide Blöcke behandelt. Mit einer Konstanten `ANZAHL_BLOECKE` statt der festen 2 ließe sich die Zahl leicht ändern.




## Aufgabe 4 – Den Agenten prüfen

Hier gibt es keine feste Lösung, denn jede KI antwortet anders. Ein gutes Ergebnis erfüllt alle Punkte:

| Prüfpunkt | richtig, wenn … |
|---|---|
| Speicherort | `PICO_FLASH_SIZE_BYTES - 2 * FLASH_SECTOR_SIZE`, nicht der letzte Sektor und nicht eine feste, ausgedachte Adresse |
| Wann gespeichert | nur bei einem neuen Rekord, nie in jedem Bild |
| Schutz | Kennung und Prüfsumme; leerer Flash (`0xFF…`) ergibt Rekord 0 |
| Schreiben | erst `flash_range_erase`, dann `flash_range_program` mit einer ganzen Seite aus dem RAM, beides über `flash_safe_execute` |
| Lesen | über `XIP_BASE` + Abstand |

Häufige KI-Fehler hier: der letzte Sektor, kein Löschen vor dem Schreiben, Schreiben von nur 12 Bytes statt einer ganzen Seite, Aufruf von `flash_range_program` ohne `flash_safe_execute`. Vergleiche das Ergebnis mit `k25_highscore.c`.

ANHANG D

# Hardware-Referenz

Dieser Anhang fasst zusammen, was du über die Hardware des PicoBoy Color Plus wissen musst, um ihn ohne die Buchbibliothek zu programmieren. Die Angaben stammen aus der Buchbibliothek (`pbc/pbc_geraet.c`, `pbc/boards/pbc_plus.h`), aus den Treibern der MicroPython-Firmware des PicoBoy und aus dem Pico SDK 2.2.0.

## D.1 Überblick

| Teil | Bauteil und Daten |
|---|---|
| Mikrocontroller | RP2350 (Bauform A, 30 GPIOs), zwei Kerne ARM Cortex-M33, 150 MHz |
| Arbeitsspeicher | 520 KB SRAM: 512 KB Hauptspeicher, dazu zweimal 4 KB (Stacks der beiden Kerne) |
| Flash | 16 MB, über QSPI angeschlossen, ab `0x10000000` eingeblendet (W25Q128 laut Firmware) |
| Display | ST7789, IPS, 240 × 280 Pixel sichtbar, runde Ecken, an SPI0 |
| Eingabe | Joystick mit fünf Richtungen (hoch, runter, links, rechts, hineindrücken), Tasten A und B |
| LEDs | drei Status-LEDs (rot, gelb, grün), eine RGB-LED SK6805-EC14 |
| Ton | kleiner Lautsprecher an einem PWM-Ausgang mit Tiefpass |
| Sensor | Beschleunigungssensor STK8BA58 an I2C0 |
| Stromversorgung | zwei AAA-Batterien oder USB-C; der Schiebeschalter wählt die Quelle (BAT = Batterien, USB = Kabel, auf USB ohne Kabel aus); Spannung über einen Teiler am ADC messbar |
| USB | USB-C, nur Full Speed (12 Mbit/s) |
| Knöpfe | BOOT und RES auf der Rückseite (Bootmodus, Neustart) |

## D.2 Pinbelegung

| GPIO | Anschluss | Richtung | in `pbc` | Hinweis |
|---|---|---|---|---|
| GP0 | Joystick Mitte | Eingang | `pbc_pressed_center` | aktiv LOW, Pull-up; Standard-UART-TX des Pico 2 |
| GP1 | Joystick rechts | Eingang | `pbc_pressed_right` | aktiv LOW, Pull-up; Standard-UART-RX des Pico 2 |
| GP2 | Joystick runter | Eingang | `pbc_pressed_down` | aktiv LOW, Pull-up |
| GP3 | Joystick links | Eingang | `pbc_pressed_left` | aktiv LOW, Pull-up |
| GP4 | Joystick hoch | Eingang | `pbc_pressed_up` | aktiv LOW, Pull-up; Standard-I2C-SDA des Pico 2 |
| GP8 | Display DC | Ausgang | | LOW = Befehl, HIGH = Daten |

| GPIO | Anschluss | Richtung | in `pbc` | Hinweis |
|---|---|---|---|---|
| GP9 | Display RST | Ausgang | | LOW = Reset |
| GP10 | Display CS | Ausgang | | normaler GPIO, LOW = ausgewählt |
| GP11 | RGB-LED | Ausgang | `pbc_rgb_led` | über PIO, WS2812-Protokoll |
| GP12 | LED grün | Ausgang | `pbc_led_green` | aktiv HIGH |
| GP13 | LED gelb | Ausgang | `pbc_led_yellow` | aktiv HIGH |
| GP14 | LED rot | Ausgang | `pbc_led_red` | aktiv HIGH |
| GP15 | Lautsprecher | Ausgang | `pbc_tone` | PWM, Slice 7, Kanal B |
| GP18 | Display SCK | Ausgang | | SPI0 |
| GP19 | Display MOSI | Ausgang | | SPI0 |
| GP20 | Sensor SDA | | | I2C0 |
| GP21 | Sensor SCL | | | I2C0 |
| GP22 | Sensor INT | Eingang | | Interrupt-Ausgang des Sensors; von `pbc` nicht benutzt |
| GP26 | Hintergrundlicht | Ausgang | `pbc_backlight` | PWM, Slice 5, Kanal A |
| GP27 | Taste A | Eingang | `pbc_pressed_a` | aktiv LOW, Pull-up |
| GP28 | Taste B | Eingang | `pbc_pressed_b` | aktiv LOW, Pull-up |
| GP29 | Batteriespannung | Eingang | `pbc_battery_voltage` | ADC-Eingang 3, Teiler 1 : 3 |

Die übrigen GPIOs (GP5 bis GP7, GP16, GP17, GP23 bis GP25) kommen in den genannten Quellen nicht vor. Ob und wo sie auf der Platine angeschlossen sind, ist dort nicht beschrieben.

> **Pins des Pico 2 sind nicht die des PicoBoy.** Die Board-Datei der Vorlage übernimmt die Grundeinstellungen des Raspberry Pi Pico 2. Dessen Standard-Pins (`PICO_DEFAULT_…`) passen hier nicht: UART an GP0/GP1 sind zwei Joystick-Richtungen, I2C an GP4/GP5 träfe Joystick hoch, und an GP25 ist keine LED (die Vorlage löscht `PICO_DEFAULT_LED_PIN` deshalb). Benutze immer die Nummern aus dieser Tabelle.

## D.3 Die Bauteile im Einzelnen

**Display ST7789.** SPI0, Modus 0 (CPOL = 0, CPHA = 0), höchstwertiges Bit zuerst. Die Bibliothek bestellt 62,5 MHz; das SDK stellt daraus nach seinem Quelltext 37,5 MHz ein (Kapitel 23.3). Der Controller hat Speicher für 240 × 320 Pixel; sichtbar sind die Zeilen **20 bis 299**. Einrichtung in `pbc`:

| Schritt | Befehl | Daten | danach warten |
|---|---|---|---|
| Hardware-Reset | RST LOW, dann HIGH | | 50 ms LOW, 150 ms danach |
| SWRESET | `0x01` | | 150 ms |
| SLPOUT | `0x11` | | 120 ms |

| Schritt | Befehl | Daten | danach warten |
|---|---|---|---|
| COLMOD | `0x3A` | `0x55` (16 Bit, RGB565) | |
| MADCTL | `0x36` | `0xC0` (180° gedreht, RGB) | |
| INVON | `0x21` | | 10 ms |
| NORON | `0x13` | | 10 ms |
| DISPON | `0x29` | | 100 ms |

Pixel: CASET `0x2A` und RASET `0x2B` mit je zwei 16-Bit-Zahlen (höheres Byte zuerst), dann RAMWR `0x2C` und die Farben in RGB565, höheres Byte zuerst. Das Hintergrundlicht an GP26 muss per PWM eingeschaltet werden, sonst bleibt der Bildschirm schwarz, auch wenn alles andere stimmt.

> In anderen Unterlagen zum PicoBoy steht MADCTL `0xC8`, also mit gesetztem BGR-Bit. Dort werden die Farben in der Software mit vertauschtem Rot und Blau gebildet. Die Buchbibliothek und die MicroPython-Firmware benutzen `0xC0` und normales RGB565.

**Tasten.** Alle sieben schließen beim Drücken den Pin mit GND kurz (aktiv LOW). Der interne Pull-up des RP2350 hält den Pin sonst auf HIGH. Die Bibliothek meldet für jeden Tasten-Pin einen Interrupt bei fallender Flanke an und entprellt mit 50 ms (Kapitel 24.2).

**Status-LEDs.** Aktiv HIGH: `gpio_put(pin, 1)` schaltet ein.

**RGB-LED SK6805-EC14.** Kompatibel zum WS2812: 24 Bit pro Farbe in der Reihenfolge **Grün, Rot, Blau**, höchstwertiges Bit zuerst, 800 kHz. Die Bibliothek erzeugt das Signal mit einem PIO-Programm aus den pico-examples (4 Befehle, eine Zustandsmaschine).

**Lautsprecher.** GP15 ist Kanal B von PWM-Slice 7. Ein Rechtecksignal mit halber Einschaltzeit ergibt einen Ton; die Bibliothek legt den Pin danach auf LOW, damit der Lautsprecher nicht auf halber Spannung hängt.

**Beschleunigungssensor STK8BA58.** I2C0 an GP20 (SDA) und GP21 (SCL), 400 kHz, Adresse **`0x18`**. Pull-up-Widerstände sind auf der Platine. Die Bibliothek benutzt diese Register:

| Register | Bedeutung | in `pbc` |
|---|---|---|
| `0x00` | Chip-Kennung | wird gelesen, um zu prüfen, ob der Sensor antwortet |
| `0x02` / `0x03` | x-Achse des Chips, 10 Bit, linksbündig in 16 Bit | wird zu `pbc_accel_y` |
| `0x04` / `0x05` | y-Achse des Chips | wird zu `pbc_accel_x`, mit umgekehrtem Vorzeichen |
| `0x06` / `0x07` | z-Achse | `pbc_accel_z` |

| Register | Bedeutung | in `pbc` |
|---|---|---|
| `0x0F` | Messbereich | `0x03` = ±2 g (512 Schritte = 2 g) |
| `0x10` | Bandbreite | `0x0C` = 125 Hz |
| `0x11` | Betriebsart | `0x00` = normaler Betrieb |

Der Chip liegt gedreht auf der Platine; daher die vertauschten Achsen. Die Richtungen in `pbc.h`: +x nach rechts, +y nach unten (zum Joystick), +z aus dem Display heraus.

**Batterien.** GP29 ist ADC-Eingang 3. Ein Spannungsteiler legt ein Drittel der Batterie- bzw. USB-Spannung an den Pin; also gilt: Spannung = ADC-Wert × 3,3 V / 4095 × 3. Ein Kondensator am Teiler braucht nach dem Einschalten des ADC etwa 50 ms, bis die Werte stimmen. Am USB misst man etwa 5 V, mit zwei frischen AAA-Batterien etwa 3,0 bis 3,2 V, mit fast leeren um 2 V.

> In der Datei `pbc_hw_config.h` der MicroPython-Firmware steht für den Teiler noch der Wert 2. Der Treiber (`battery.c`) ignoriert ihn ausdrücklich und rechnet mit 3, weil das auf dem Gerät so gemessen wurde.

## D.4 Was die Buchbibliothek belegt

Wer eigene SDK-Programme schreibt und dabei `pbc_init()` aufruft, sollte wissen, welche Teile des RP2350 schon vergeben sind:

| Teil | belegt durch `pbc` | frei für eigene Programme |
|---|---|---|
| SPI | SPI0 (Display) | SPI1 |
| I2C | I2C0 (Sensor) | I2C1 |
| DMA | ein Kanal (`pbc_show`) | die übrigen 15; mit `dma_claim_unused_channel` holen |
| PIO | eine Zustandsmaschine in PIO0 (oder PIO1) | die übrigen; der RP2350 hat drei PIO-Blöcke |
| PWM | Slice 5 (Licht), Slice 7 (Lautsprecher) | die übrigen Slices |
| ADC | Eingang 3 (Batterie) | |
| GPIO-Interrupts | eigener Handler für die sieben Tasten-Pins | `gpio_set_irq_callback` für andere Pins (Kapitel 24.2) |
| Timer | ein Alarm für das Ende von `pbc_tone` | weitere Alarme und Timer |
| Kern 1 | nicht benutzt | ganz frei (Kapitel 24.3) |
| UART | aus (`pico_enable_stdio_uart(… 0)`) | Pins sind Joystick-Tasten |

## D.5 Speicheraufteilung

**RAM** (aus dem Linkerskript des SDK für den RP2350):

| Bereich | Adresse | Größe | Inhalt |
|---|---|---|---|
| RAM | `0x20000000` | 512 KB | globale und `static`-Variablen, Bildpuffer (134 400 Bytes), Heap |
| SCRATCH_X | `0x20080000` | 4 KB | Stack von Kern 1 |
| SCRATCH_Y | `0x20081000` | 4 KB | Stack von Kern 0 (`main`) |

Die Board-Datei setzt `PICO_STACK_SIZE` auf 4 KB (sonst nutzt das SDK nur die Hälfte) und `PICO_USE_STACK_GUARDS`, damit ein übergelaufener Stack das Programm anhält, statt still Daten zu überschreiben (Kapitel 19).

**Flash** (16 MB, eingeblendet ab `XIP_BASE` = `0x10000000`):

| Bereich | Adresse | Inhalt |
|---|---|---|
| Anfang | `0x10000000` | dein Programm (meist unter 1 MB) |
| Mitte | | frei; andere Firmware (etwa MicroPython mit Dateisystem) benutzt Teile davon |
| vorletzter Sektor | `0x10FFE000` bis `0x10FFEFFF` | Speicherort für Spielstände in Kapitel 25 |
| letzter Sektor | `0x10FFF000` bis `0x10FFFFFF` | frei lassen: Viele `.uf2`-Dateien schreiben einen Zusatzblock nach `0x10FFFF00` |

Löschen geht in Sektoren zu 4096 Bytes, Schreiben in Seiten zu 256 Bytes (Kapitel 25.1). Aufspielen per Kopieren oder **Run** überschreibt nur die Bereiche, die das neue Programm belegt.

ANHANG E

# Installation im Detail

Kapitel 2 beschreibt den einfachsten Weg: VS Code mit der Erweiterung „Raspberry Pi Pico“. Dieser Anhang ergänzt, was unter macOS und Linux zusätzlich nötig ist, wie man ganz ohne VS Code baut und aufspielt, und was bei typischen Fehlermeldungen hilft. Programme ändern sich; maßgeblich ist immer die Dokumentation der Erweiterung (auf ihrer Seite im Erweiterungs-Marktplatz von VS Code und unter `github.com/raspberrypi/pico-vscode`) und die des Pico SDK.

## E.1 Die Erweiterung unter Windows, macOS und Linux

Die Erweiterung lädt beim ersten **Import Project** selbst herunter, was sie braucht: das Pico SDK in der Version aus der `CMakeLists.txt` (2.2.0), den Compiler (Arm GNU Toolchain 14.2, `toolchainVersion` in der `CMakeLists.txt`), `picotool` und die Bauwerkzeuge. Sie legt alles im Ordner `.pico-sdk` in deinem Benutzerordner ab. Die ersten Zeilen der `CMakeLists.txt` der Vorlage („DO NOT EDIT …“) binden von dort die Datei `cmake/pico-vscode.cmake` ein, wenn es sie gibt. Einige Werkzeuge muss das Betriebssystem aber schon mitbringen. Laut der Dokumentation der Erweiterung gilt:

| System | Was zusätzlich nötig ist |
|---|---|
| **Windows** | nichts |
| **macOS** | die Kommandozeilen-Werkzeuge von Apple: im Terminal `xcode-select --install` ausführen. Damit kommen unter anderem Git und Tar. Unter Umständen muss man zusätzlich die Xcode-Lizenz annehmen: `sudo xcodebuild -license` |
| **Raspberry Pi OS** (64 Bit, mit Desktop) | nichts |
| **andere Linux-Systeme** | Python ab 3.10, Git ab 2.28 und Tar, jeweils im Suchpfad (`PATH`) |

Die Erweiterung selbst verlangt eine aktuelle Version von VS Code. Ist deine zu alt, meldet sie das bei der Installation.

## E.2 Linux: Rechte für USB und serielle Schnittstelle

**Aufspielen mit Run.** Die Erweiterung spielt Programme mit `picotool` auf. Unter Linux darf ein normaler Benutzer ohne passende **udev-Regel** nicht auf das USB-Gerät zugreifen. `picotool` meldet dann, es finde kein erreichbares Gerät (*no accessible RP2350 devices*), obwohl der PicoBoy

angeschlossen ist. Die Regel liegt dem Quelltext von `picotool` bei, als Datei `udev/60-picotool.rules` (zu finden unter `github.com/raspberrypi/picotool`). Laut seiner Anleitung kopiert man sie so und lädt die Regeln neu:

```
sudo cp 60-picotool.rules /etc/udev/rules.d/
sudo udevadm control --reload
```

Danach den PicoBoy einmal abziehen und wieder anstecken. Die Regel gibt dem angemeldeten Benutzer Zugriff auf die USB-Geräte von Raspberry Pi (Hersteller-Kennung `2e8a`), im Bootmodus und mit laufendem Programm.

**Der serielle Monitor.** Unter Linux heißt die Schnittstelle des PicoBoy `/dev/ttyACM0` (oder mit einer höheren Nummer). Auf vielen Systemen, etwa Debian und Ubuntu, gehört sie der Gruppe `dialout`. Meldet der Serial Monitor, er dürfe den Port nicht öffnen, nimm dich in diese Gruppe auf und melde dich danach einmal ab und wieder an:

```
sudo usermod -aG dialout $USER
```

## E.3 Bauen ohne VS Code

Wer lieber im Terminal arbeitet, braucht dieselben Werkzeuge, die die Erweiterung sonst besorgt:

| Werkzeug | wofür |
|---|---|
| **Arm GNU Toolchain** (`arm-none-eabi-gcc`) | der Compiler für den RP2350 |
| **CMake** ab 3.13 | liest die `CMakeLists.txt` und plant den Bau |
| **Make** oder **Ninja** | führt den Bau aus |
| **Python 3** | braucht das SDK beim Bauen |
| **Git** | um das SDK zu holen |
| ein Compiler für den PC (z. B. `gcc`) | das SDK baut damit Hilfsprogramme wie `pioasm` und bei Bedarf `picotool` |

**Unter Debian, Ubuntu und Raspberry Pi OS** nennt die Anleitung des SDK (Datei `README.md`) diesen Befehl:

```
sudo apt install cmake python3 build-essential gcc-arm-none-eabi libnewlib-arm-none-
  eabi libstdc++-arm-none-eabi-newlib
```

**Unter macOS und Windows** bekommst du den Compiler von Arm (`developer.arm.com`, Suchbegriff „Arm GNU Toolchain“) und CMake von `cmake.org`. Einfacher ist es dort meist, die Erweiterung einmal alles installieren zu lassen und danach in VS Code zu bauen.

**Das SDK holen.** Das Pico SDK liegt bei GitHub. Die Vorlage ist für die Version 2.2.0 gemacht:

```
git clone -b 2.2.0 https://github.com/raspberrypi/pico-sdk.git ~/pico/pico-sdk
cd ~/pico/pico-sdk
git submodule update --init
```

Der letzte Befehl holt unter anderem **TinyUSB**, die Bibliothek für USB. Fehlt sie, warnt CMake *TinyUSB submodule has not been initialized; USB support will be unavailable*, und `printf` über USB funktioniert nicht.

**Bauen.** Im Projektordner (dort, wo `CMakeLists.txt` liegt):

```
export PICO_SDK_PATH=~/pico/pico-sdk
cmake -B build
cmake --build build
```

`PICO_SDK_PATH` sagt der Datei `pico_sdk_import.cmake`, wo das SDK liegt. Statt mit `export` kann man es auch einmalig angeben: `cmake -B build -DPICO_SDK_PATH=$HOME/pico/pico-sdk`. Der erste Befehl richtet den Ordner `build` ein (Konfigurieren, Kapitel 2.3), der zweite übersetzt und bindet. Danach liegt dort `mein_programm.uf2`. Nach einer Änderung an `main.c` genügt `cmake --build build`.

> Hat VS Code den Ordner `build` schon einmal eingerichtet, kann man im Terminal direkt mit `cmake --build build` neu bauen: Alle Einstellungen (SDK, Compiler, Bauwerkzeug) stehen dann schon in `build`. So arbeiten auch die Agenten aus Kapitel 27.3.

## E.4 Aufspielen mit `picotool`

Statt die `.uf2`-Datei auf das Laufwerk zu ziehen, kann man sie mit `picotool` aufspielen. Die Erweiterung bringt es mit; ohne sie installiert man es nach der Anleitung in `github.com/raspberrypi/picotool`. Findet das SDK beim Bauen kein passendes `picotool`, lädt es seinen Quelltext selbst herunter und baut es.

| Befehl | Wirkung |
|---|---|
| `picotool info -a` | zeigt, was auf dem angeschlossenen PicoBoy ist; prüft die Verbindung |
| `picotool load -x build/mein_programm.uf2` | aufspielen und gleich starten (`-x`) |
| `picotool load -f -x build/mein_programm.uf2` | dasselbe, wenn gerade ein Programm läuft: `-f` schaltet es erst in den Bootmodus |

| Befehl | Wirkung |
|---|---|
| `picotool reboot` | neu starten |
| `picotool save -r 0x10FFE000 0x10FFF000 stand.bin` | den Spielstand-Sektor aus Kapitel 25 in eine Datei sichern |
| `picotool erase -r 0x10FFE000 0x10FFF000` | nur diesen Sektor löschen |

Ohne `-f` muss der PicoBoy im Bootmodus sein (BOOT halten, RES tippen). `-f` klappt nur, wenn das laufende Programm auf diesen Befehl hört. Die Programme dieses Buchs tun das, weil die Vorlage die Konsole über USB einschaltet; das SDK richtet dabei eine Schnittstelle für genau diesen Neustart ein. Läuft MicroPython oder ein fremdes Spiel, geht es nur über den Bootmodus.

## E.5 Typische Fehlermeldungen

| Meldung | Ursache und Abhilfe |
|---|---|
| `SDK location was not specified. Please set PICO_SDK_PATH …` | CMake findet das SDK nicht: `PICO_SDK_PATH` setzen (E.3) oder in VS Code bauen |
| `Directory '…' does not appear to contain the Raspberry Pi Pico SDK` | `PICO_SDK_PATH` zeigt auf den falschen Ordner |
| `TinyUSB submodule has not been initialized` | im SDK-Ordner `git submodule update --init` |
| `Incompatible picotool installation found` | ein altes `picotool` ist installiert. Neueres installieren oder CMake mit `-DPICOTOOL_FORCE_FETCH_FROM_GIT=1` aufrufen, dann baut das SDK ein eigenes |
| `no accessible RP2350 devices` (Linux) | udev-Regel fehlt (E.2), oder der PicoBoy ist nicht im Bootmodus und das laufende Programm hört nicht auf `-f` |
| Bau bricht ab, Pfad mit Leerzeichen oder Umlaut in der Meldung | Projektordner an einen Ort ohne Sonderzeichen verschieben (Kapitel 2.2) |
| Nach dem Aufspielen zeigt das Display noch das alte Bild | einmal stromlos machen: USB abziehen, kurz warten, wieder anstecken |