PICOBOY / COLOR PLUS

BAND 03

Programmieren mit Arduino

Spiele, Grafik und Sensoren mit der Arduino IDE



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, mit **Arduino** zu programmieren – der wohl beliebtesten Art, Mikrocontroller zum Leben zu erwecken. Arduino-Sketche laufen auf Tausenden verschiedenen Platinen; was du hier lernst, kannst du also auch später gebrauchen. Du lernst es 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 Sketch 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: Zeiger, eigene Bibliotheken, PWM, I2C, SPI, Interrupts, der zweite Prozessorkern und Dateien im Flash. 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 Sketchen. Jeder Sketch hat Zeilennummern, und der Text erklärt, was in welcher Zeile passiert. Daneben siehst du, was auf dem Bildschirm des PicoBoy erscheint, oder darunter, was im seriellen Monitor steht.
- **Ausprobieren** – ein etwas größerer Sketch, der 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
- die **Arduino IDE 2** mit dem Kern für den RP2350 (Kapitel 2)
- die Bibliotheken **Adafruit ST7735 and ST7789**, **Adafruit NeoPixel** und **STK8BA58** (Kapitel 2)
- einen **PicoBoy Color Plus** mit USB-Kabel, das auch Daten überträgt

Alle Sketche aus dem Buch liegen bei. Den Namen findest du über jedem Sketch. Abtippen lohnt sich trotzdem: Beim Tippen lernst du am schnellsten – und nebenbei, Tippfehler zu finden.

Die Teile des Buchs

Teil	Darum geht es
I – Los geht's	Gerät und Arduino, die IDE einrichten, der erste Sketch, der serielle Monitor
II – Grundlagen	Variablen, Zeichnen, Schleifen, Entscheidungen, Tasten, LEDs, RGB-LED und Töne, Funktionen, Arrays und Texte, Strukturen und Klassen
III – Spiele bauen	flimmerfreie Animation, Bilder, Kollisionen, Neigungssensor, ein ganzes Spiel
IV – Hinter die Kulissen	Zeiger, eigene Bibliotheken, PWM und ADC, I2C, SPI, Interrupts, zweiter Kern, Dateien – und Programmieren mit KI

Für dieses Buch brauchst du die Arduino IDE 2 und den Kern arduino-pico.

Viel Spaß beim Programmieren!

Inhalt

Teil I – Los geht's

1 Mikrocontroller und Arduino 5
2 Die Arduino IDE einrichten 12
3 Der erste Sketch 20
4 Der serielle Monitor 36

Teil II – Grundlagen

5 Variablen und Datentypen 48
6 Zeichnen und Farben 63
7 Schleifen und Zeit 74
8 Entscheidungen 85
9 Tasten und Joystick 96
10 LEDs, RGB-LED und Töne 110
11 Funktionen 124
12 Arrays und Texte 137
13 Strukturen und eine erste eigene Klasse 151

Teil III – Spiele bauen

14 Flimmerfrei: `GFXcanvas16` und `drawRGBBitmap` 165
15 Bilder und Sprites 178
16 Kollisionen 190
17 Der Neigungssensor mit der Bibliothek STK8BA58 200
18 Projekt: Ein ganzes Spiel 210

Teil IV – Hinter die Kulissen

19 Zeiger und Speicher 221
20 Eigene Bibliotheken – ein Blick in STK8BA58 235
21 PWM und ADC: Licht dimmen, Batterie messen 247
22 I2C mit `Wire`: der Neigungssensor ohne Bibliothek 255
23 SPI und das Display 265
24 Interrupts und der zweite Kern 276
25 Daten dauerhaft speichern: LittleFS und EEPROM 291
26 Dein eigenes Projekt 302
27 Programmieren mit KI: Vibecoding 311

Anhang

A Die wichtigsten Befehle 324
B Fehlermeldungen beim Kompilieren und Hochladen 333
C Lösungen der Übungsaufgaben 341
D Hardware-Referenz: Pinbelegung des PBC+ 438
E Installation im Detail (Windows, macOS, Linux), arduino-cli 443

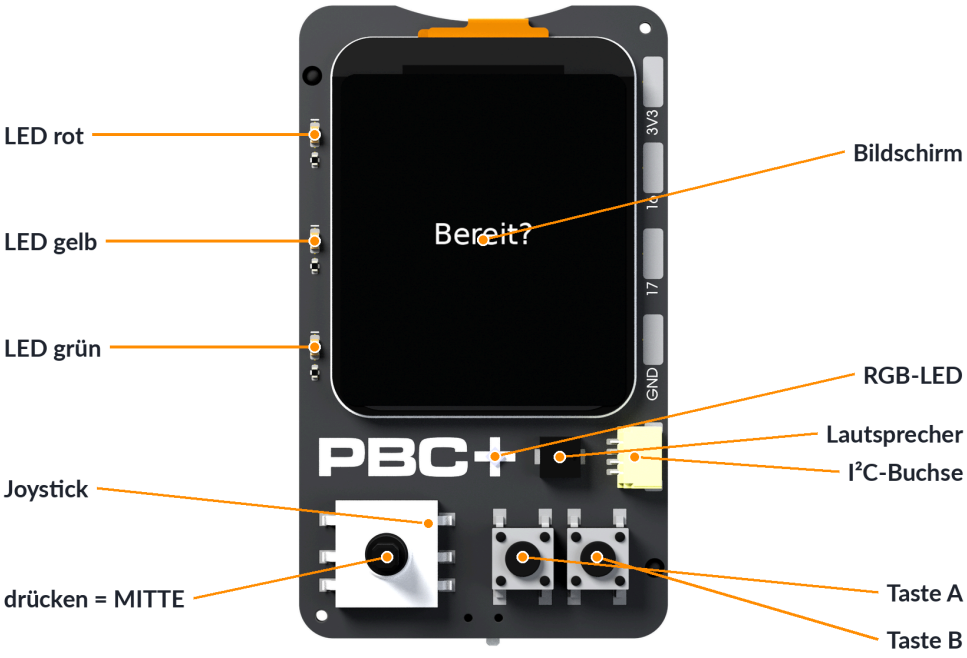
KAPITEL 1

Mikrocontroller und Arduino

Worum geht's? Bevor du den ersten Sketch schreibst, lernst du dein Gerät kennen und erfährst, was Arduino eigentlich ist und was mit einem Sketch passiert, bis er 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 (Display), 240 Pixel breit und 280 Pixel hoch, mit runden Ecken
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, wenn sonst nichts mehr geht (Abschnitt 1.3)

Der Mikrocontroller des PBC+ heißt **RP2350** und stammt von Raspberry Pi. Derselbe Chip steckt auch im Raspberry Pi Pico 2, einer kleinen Platine für Bastelprojekte. 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: für Spiele mit bewegten Figuren, Musik und Messwerten vom Neigungssensor. Im **Flash-Speicher** steht dein Programm. Er behält seinen Inhalt auch ohne Strom, so wie ein USB-Stick.

1.2 Was ist Arduino?

Arduino ist ein Projekt, das 2005 in Italien begann. Studierende sollten ohne lange Vorbereitung Elektronik programmieren können: Lämpchen blinken lassen, Taster abfragen, Motoren drehen. Daraus ist weltweit eine der bekanntesten Arten geworden, Mikrocontroller zu programmieren. Mit „Arduino“ sind drei Dinge gemeint:

- **Die Platinen.** Die bekannteste ist der Arduino Uno, eine blaue Platine mit Steckleisten. Inzwischen gibt es Dutzende Arduino-Platinen.
- **Die Programmierumgebung**, die **Arduino IDE** (*Integrated Development Environment*, Entwicklungsumgebung). In ihr schreibst du deine Programme, übersetzt sie und spielst sie mit einem Klick auf das Gerät. Sie ist kostenlos und läuft unter Windows, macOS und Linux.
- **Die Gemeinschaft.** Millionen Menschen arbeiten mit Arduino. Für fast jedes Bauteil gibt es eine fertige **Bibliothek**: eine Sammlung von Befehlen, die jemand anders geschrieben hat und die du in deinem Programm benutzen kannst. Zu fast jeder Frage findest du im Internet eine Antwort, meist mit Beispiel.

Ein Arduino-Programm heißt **Sketch** (englisch: Skizze). Der Name kommt daher, dass man schnell etwas ausprobieren soll, so wie man eine Idee skizziert. Ein Sketch ist eine Datei mit der Endung `.ino` und liegt immer in einem Ordner mit demselben Namen: Der Sketch `hallo.ino` steht im Ordner `hallo`.

Der PicoBoy ist keine Platine von Arduino. Trotzdem kannst du ihn mit der Arduino IDE programmieren. Dafür sorgt ein **Kern** (englisch *core*): ein Zusatzpaket für die IDE, das ihr einen neuen Mikrocontroller beibringt. Für den RP2350 gibt es den Kern **„Raspberry Pi Pico/RP2040/RP2350“** von Earle Philhower. Ihn installierst du in Kapitel 2, zusammen mit drei Bibliotheken für Display, RGB-LED und Neigungssensor.

> Den PicoBoy kann man auch in Python (MicroPython) oder in C mit dem Pico SDK von Raspberry Pi programmieren; dafür gibt es eigene Bücher. Dieses Buch setzt beides nicht voraus.

1.3 Vom Sketch zur Firmware

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 deinen Sketch als **Quelltext**: als Text in einer **Programmiersprache**, die Menschen lesen können. Ein kleines Stück davon sieht so aus:

```
tft.fillScreen(ST77XX_BLACK);
tft.setCursor(66, 128);
tft.print("Hallo!");
```

Ohne Vorkenntnisse kann man ahnen, was hier passiert: Der Bildschirm wird schwarz, dann erscheint „Hallo!“. 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. Die Arduino IDE bringt ihn mit; du rufst ihn mit einem Klick auf.

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

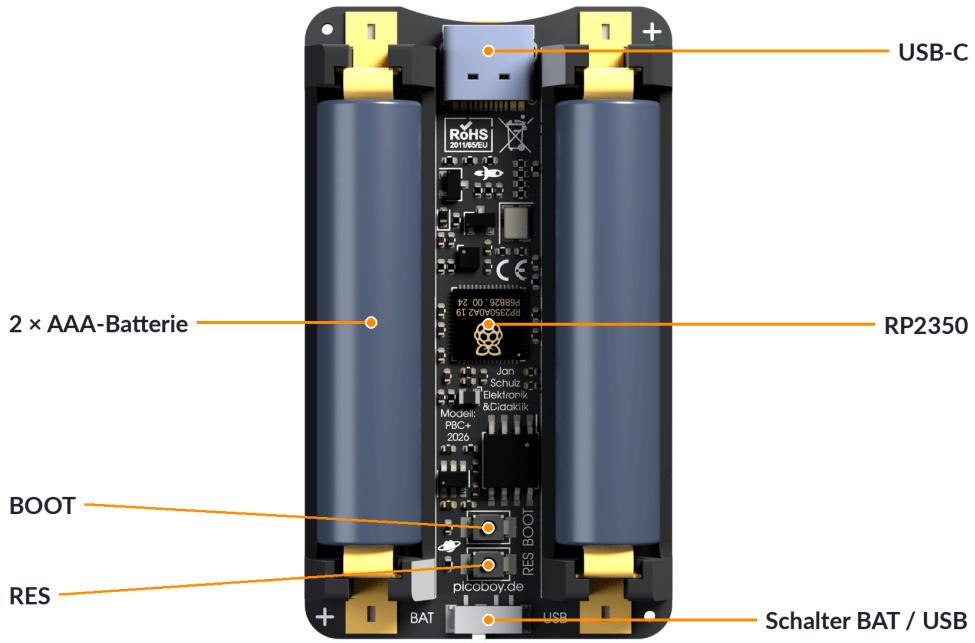


1. Du schreibst den Sketch in der Arduino IDE.
2. Du klickst auf **Überprüfen**. Der Compiler übersetzt den Sketch. Dieser Schritt heißt **Kompilieren** (englisch *compile*). Findet der Compiler einen Fehler, bricht er ab und sagt dir, in welcher Zeile.

3. Am Ende steht eine Datei mit der Endung `.uf2`. Sie enthält das ganze Programm, fertig zum Aufspielen. Du bekommst sie meist gar nicht zu sehen, die IDE erledigt den nächsten Schritt selbst.
4. Du klickst auf **Hochladen**. Die IDE kompiliert, falls nötig, und bringt das Programm über das USB-Kabel auf den PicoBoy. Der PicoBoy schreibt es in seinen Flash-Speicher, startet neu und führt es aus.

Der Bootmodus. Zum Aufspielen muss der PicoBoy im **Bootmodus** sein. Dann meldet er sich am Computer wie ein USB-Stick, als neues Laufwerk, und nimmt eine `.uf2`-Datei an. Läuft auf dem PicoBoy schon ein Sketch, schickt die IDE ihm über das Kabel ein verabredetes Signal, und der PicoBoy schaltet sich selbst in den Bootmodus. Das klappt aber nur, wenn ein Arduino-Sketch läuft. Beim allerersten Mal und wenn gar nichts mehr geht, bringst du ihn von Hand in den Bootmodus:

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.



Wie das Hochladen in der IDE genau aussieht, zeigt Kapitel 2.5.

1.4 Arduino-C++ und das, was es dir abnimmt

Die Sprache, in der du Sketches schreibst, ist **C++**. C++ ist eine der wichtigsten Programmiersprachen überhaupt. Sie steckt in Spielen, Browsern, Autos und in fast jedem Gerät mit Mikrocontroller. Arduino hat keine eigene Sprache erfunden, sondern macht dir C++ an einigen Stellen bequemer. Was du in diesem Buch lernst, gilt also auch außerhalb von Arduino.

Erst übersetzen, dann ausführen. Ein C++-Programm wird vollständig übersetzt, bevor es läuft. Das hat zwei Folgen. Viele Fehler findet der Compiler schon, bevor das Programm überhaupt auf dem PicoBoy ist. Und das übersetzte Programm ist sehr schnell.

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

Was Arduino dir abnimmt:

- **Den Programmrahmen.** Ein reines C++-Programm braucht einen festen Einstieg und muss sich selbst um vieles kümmern, bevor es loslegen kann. Ein Sketch besteht stattdessen aus zwei Teilen: `setup()` läuft einmal beim Start, `loop()` danach immer wieder. Den Rest erledigt der Kern (Kapitel 3.1).
- **Die Hardware-Einzelheiten.** Um ein Lämpchen einzuschalten, müsstest du ohne Arduino wissen, welche Bits in welchen Steuerspeicher des Chips gehören. Mit Arduino schreibst du `digitalWrite(14, HIGH)`. Für Pausen gibt es `delay`, für Töne `tone`, für die Verbindung zum Computer `Serial`. Diese Befehle heißen auf jedem Arduino-Gerät gleich.
- **Bauteile mit Bibliotheken.** Das Display des PBC+ versteht eine eigene Reihe von Steuerbefehlen. Die Bibliothek von Adafruit kennt sie; du schreibst einfach `tft.print("Hallo!")`.
- **Die Werkzeuge.** Compiler, Hilfsprogramme zum Aufspielen und Bibliotheken lädt die IDE selbst herunter. Du musst nichts von Hand einrichten.

Was dir niemand abnimmt: C++ ist streng. Ein vergessenes Semikolon, ein falsch geschriebener Name oder ein Großbuchstabe an der falschen Stelle, und der Compiler verweigert die Arbeit. Das ist am Anfang lästig, aber nützlich: Lieber meldet der Compiler einen Fehler, als dass der PicoBoy still das Falsche tut. Wie du die Meldungen liest, lernst du in Kapitel 3.6.

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 deinen ersten Sketch hochlädst, ist MicroPython weg. Das ist kein Problem: Jede Firmware lässt sich jederzeit wieder aufspielen.

> 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. Den PicoBoy in den Bootmodus bringen (BOOT halten, RES tippen) und die `.uf2`-Datei auf das neue Laufwerk ziehen. Das Laufwerk verschwindet von selbst, der

PicoBoy startet neu.

- **Ein fertiges Spiel aufspielen:** Auch Spiele für den PBC+ gibt es als `.uf2`-Datei. Sie kommen auf demselben Weg auf das Gerät. Es läuft immer nur eine Firmware zur selben Zeit.
- **Deinen Sketch wieder aufspielen:** Der Sketch bleibt auf deinem Computer. Öffne ihn in der Arduino IDE und lade ihn erneut hoch. Weil auf dem PicoBoy dann kein Arduino-Sketch läuft, der auf das Signal der IDE hört, brauchst du dafür wieder einmal den Bootmodus (Kapitel 2.5).

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

Übungsaufgaben

> **Aufgabe 1** ⭐ **Begriffe ordnen** Bringe in die richtige Reihenfolge und erkläre jeden Begriff in einem Satz: `.uf2`-Datei, Sketch, Hochladen, 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 …?** Eine Freundin hat einen Sketch hochgeladen, und seitdem bleibt der Bildschirm schwarz. Auch **Hochladen** in der IDE klappt nicht mehr. Sie fürchtet, das Gerät sei kaputt. Was sagst du ihr, und was soll sie tun?

Aufgabe 5 ⭐⭐ Kern oder Bibliothek? Ordne zu, ob es sich um einen **Kern** oder um eine **Bibliothek** handelt, und begründe: a) Adafruit NeoPixel, b) „Raspberry Pi Pico/RP2040/RP2350“, c) STK8BA58, d) Adafruit ST7735 and ST7789.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Der PBC+ hat einen **Mikrocontroller** RP2350: 2 Kerne, 520 KB Arbeitsspeicher, 16 MB Flash.
- **Arduino** sind Platinen, die **Arduino IDE** und eine große Gemeinschaft mit vielen **Bibliotheken**. Ein Arduino-Programm heißt **Sketch** (`name/name.ino`).
- Den RP2350 bringt der **Kern** „Raspberry Pi Pico/RP2040/RP2350“ in die IDE.
- Sketches schreibst du in **C++**. **Überprüfen** kompiliert, **Hochladen** kompiliert und spielt auf.
- Ein Sketch ist die ganze **Firmware** des Geräts. Er ersetzt MicroPython; MicroPython lässt sich jederzeit wieder aufspielen.
- **Bootmodus:** BOOT halten, RES tippen. Er geht immer, egal was vorher lief.

KAPITEL 2

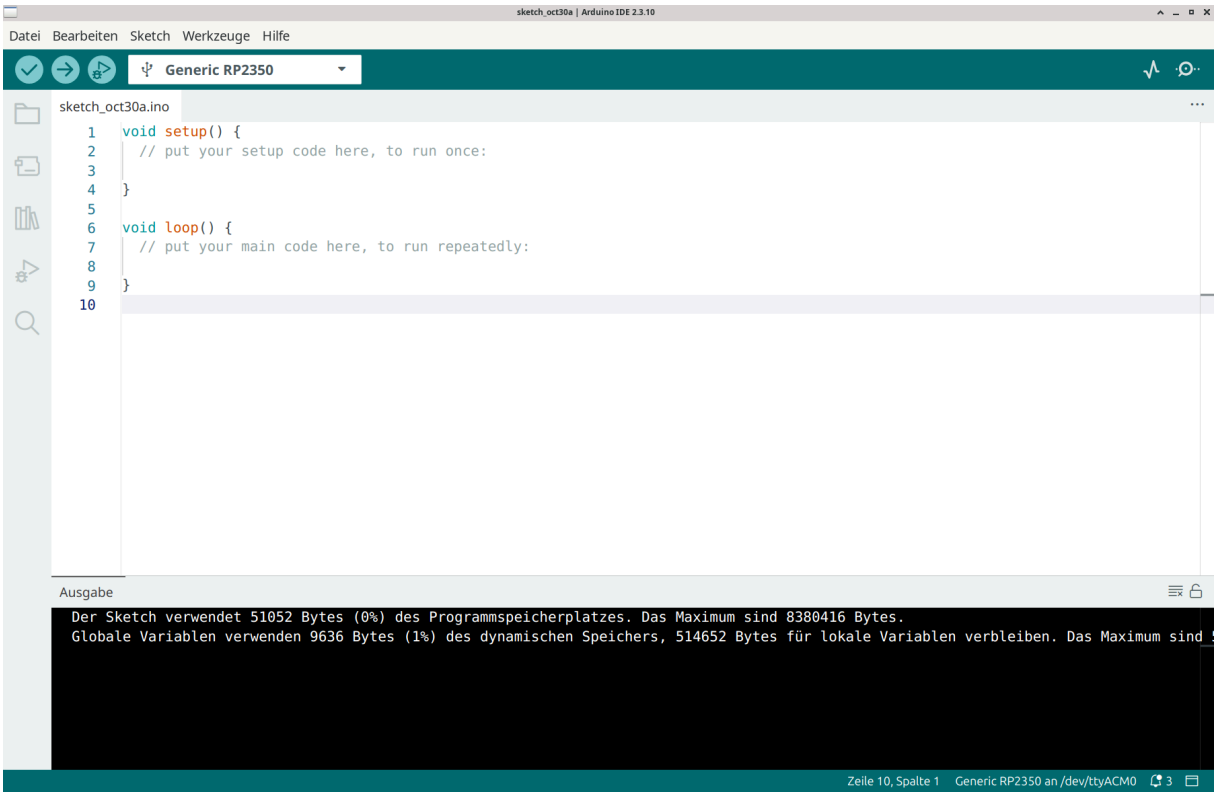
Die Arduino IDE einrichten

Worum geht's? Du installierst die Arduino IDE, bringst ihr den Mikrocontroller des PicoBoy bei, stellst das richtige Board ein, installierst drei Bibliotheken und lädst deinen ersten Sketch hoch. Das dauert beim ersten Mal etwa eine halbe Stunde, weil viel heruntergeladen wird. Danach ist alles mit einem Klick erledigt.

2.1 Arduino IDE 2 installieren

1. Lade die **Arduino IDE 2** von **arduino.cc** herunter (Bereich *Software*). Es gibt sie für Windows, macOS und Linux.
2. Installiere sie. Unter **Windows** nimm das Installationsprogramm oder die ZIP-Datei von arduino.cc, **nicht** die Fassung aus dem Microsoft Store: Mit ihr erkennt die IDE den PicoBoy nicht zuverlässig. Bietet das Installationsprogramm Treiber an, lass sie installieren. Unter **macOS** ziehst du die IDE in den Ordner *Programme*. Unter **Linux** gibt es eine AppImage-Datei, die du ausführbar machst und startest.
3. Starte die IDE. Beim ersten Start lädt sie noch einige Teile nach. Das siehst du unten rechts.

Die IDE zeigt ihre Menüs auf Englisch oder Deutsch. Die Sprache stellst du in den Einstellungen um: **File → Preferences…** (deutsch **Datei → Einstellungen…**), dort bei **Language** (Sprache). Dieses Buch nennt die deutschen Bezeichnungen. Einige Menüpunkte bleiben immer englisch, weil sie aus dem Kern stammen, etwa **Flash Size**.



Die Arduino IDE 2 mit leerem Sketch: oben die Werkzeugleiste, unten die Ausgabe, links die Seitenleiste.

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

So sieht das Fenster aus:

Bereich	Wofür?
Werkzeugleiste oben links	**Überprüfen** (Häkchen) kompiliert, **Hochladen** (Pfeil nach rechts) kompiliert und spielt auf; dazu das **Debug**-Symbol, das du in diesem Buch nicht brauchst
Board-Auswahl daneben	zeigt das eingestellte Board; der Port steht unten rechts im Fenster
Editor in der Mitte	hier schreibst du deinen Sketch
Ausgabe unten	Meldungen beim Kompilieren und Hochladen
Seitenleiste links	Sketchbook, **Board-Verwaltung**, **Bibliotheksverwalter**, Suche
oben rechts	**Serieller Plotter** und **Serieller Monitor** (Kapitel 4)

> Unter **Linux** kann die IDE als Flatpak installiert sein (das ist bei Mint, Fedora oder Pop!_OS üblich). Dann darf sie nicht auf das Laufwerk des PicoBoy zugreifen, das beim Hochladen auftaucht. Abhilfe: im Terminal einmal `flatpak override --user --filesystem=host cc.arduino.IDE2` eingeben. Mehr zu Windows, macOS und Linux steht in Anhang E.

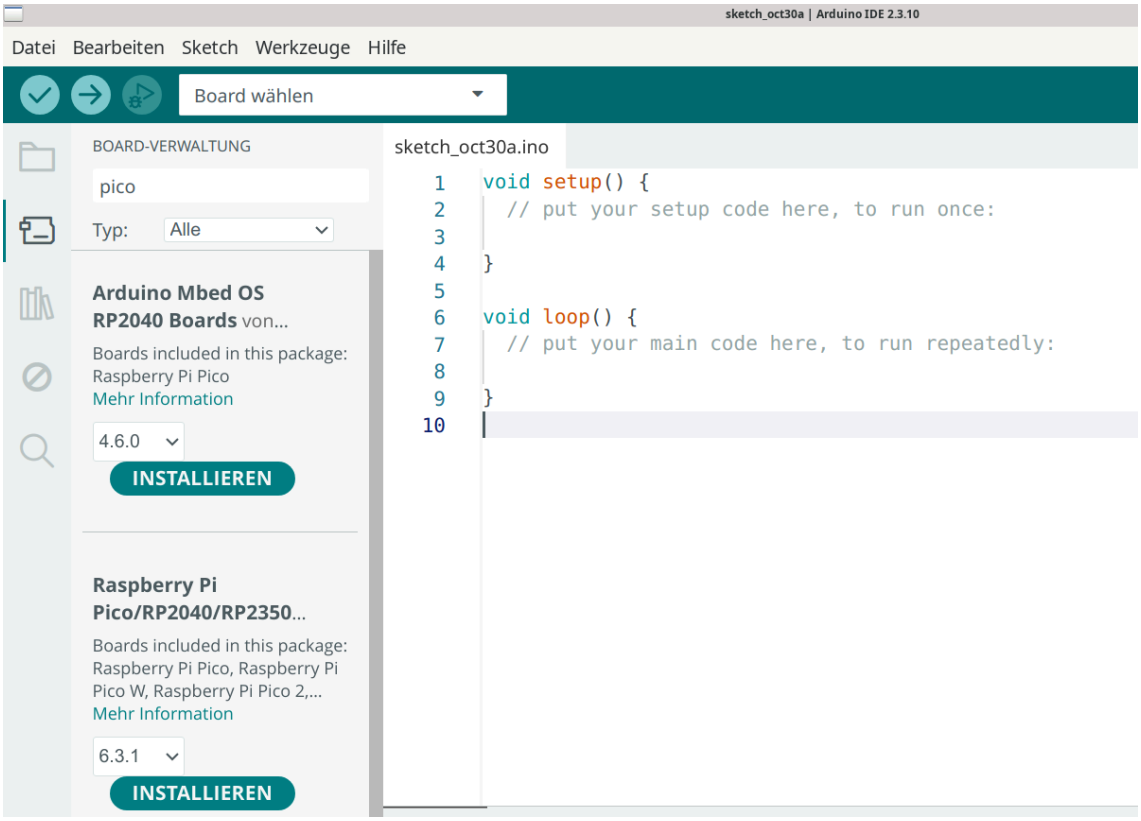
2.2 Den RP2350 nachrüsten

Von sich aus kennt die IDE nur Platinen von Arduino. Den RP2350 bringt ihr der **Kern** „Raspberry Pi Pico/RP2040/RP2350“ von Earle Philhower bei (Kapitel 1.2). Er steht nicht in der Liste, die die IDE von Arduino lädt. Deshalb sagst du ihr zuerst, wo sie seine Beschreibung findet.

1. Öffne **Datei → Einstellungen…** (Tastenkürzel **Strg + Komma**, am Mac **Cmd + Komma**).
2. Trage in das Feld **Zusätzliche Boardverwalter-URLs** (*Additional boards manager URLs*) diese Adresse ein:

   `https://github.com/earlephilhower/arduino-pico/releases/download/global/package_rp2040_index.json`

   Steht dort schon eine andere Adresse, klicke auf das Symbol neben dem Feld. Dann öffnet sich eine Liste, in der jede Adresse eine eigene Zeile bekommt.
3. Klicke **OK**. Die IDE lädt die neue Liste.
4. Öffne die **Board-Verwaltung**: über das zweite Symbol in der Seitenleiste oder über **Werkzeuge → Board → Board-Verwaltung…**. Das Tastenkürzel **Strg + Umschalt + B** klappt nicht zuverlässig.
5. Suche nach **pico**. Wähle den Eintrag **Raspberry Pi Pico/RP2040/RP2350** von **Earle F. Philhower, III** und klicke **Installieren**.



Die Board-Verwaltung nach der Suche „pico“. Der Eintrag „Raspberry Pi Pico/RP2040/RP2350“ ist der zweite.

Die IDE lädt jetzt den Kern mit dem Compiler und allen Hilfsprogrammen herunter, zusammen rund 1,5 GB. Das dauert je nach Internetverbindung einige Minuten. Nimm die Version **6.3 oder neuer**; die Sketche dieses Buchs sind mit 6.3.0 geprüft.

2.3 Board und Flash-Größe einstellen

Der Kern kennt Hunderte Platinen mit RP2040- und RP2350-Chip. Den PicoBoy kennt er nicht beim Namen. Du wählst deshalb die allgemeine Einstellung für den RP2350 und sagst dazu, wie viel Flash-Speicher eingebaut ist.

1. **Werkzeuge → Board → Raspberry Pi Pico/RP2040/RP2350 → Generic RP2350**. Die Liste ist lang; ganz oben hilft das Suchfeld, wenn du die Auswahl über die Board-Auswahl in der Werkzeugleiste öffnest (**Wähle ein anderes Board und einen anderen Port …**).
2. **Werkzeuge → Flash Size → 16MB (Sketch: 8MB, FS: 8MB)**.

Alle anderen Punkte im Menü **Werkzeuge** bleiben, wie sie sind. Die IDE merkt sich die Einstellungen; du machst das nur einmal.

Warum nicht „Raspberry Pi Pico 2“? Im Pico 2 steckt zwar derselbe Chip. Er hat aber nur 4 MB Flash, und deshalb bietet der Kern für ihn nur Einstellungen bis 4 MB an. Der PBC+ hat **16 MB**. Mit „Generic RP2350“ kannst du die Größe selbst wählen. Ohne deine Wahl würde der Kern dort mit 2 MB rechnen.

Was bedeutet „Sketch: 8MB, FS: 8MB“? Der Flash wird zweigeteilt. Die ersten 8 MB nehmen deinen Sketch auf, also das Programm samt Bildern und Tönen, die darin stecken. Die anderen 8 MB werden zu einem **Dateisystem** (*file system*, FS): einem Speicher, in den dein Sketch selbst Dateien schreiben kann, etwa einen Punkterekord. Wie das geht, zeigt Kapitel 25. 8 MB für den Sketch sind sehr viel; die Sketche in diesem Buch brauchen selten mehr als ein paar hundert Kilobyte.

> Stelle außerdem in **Datei → Einstellungen…** den Punkt **Compiler-Meldungen** (*Compiler warnings*) auf **Alle**. Voreingestellt ist **Keine**; dann verschweigt der Compiler **Warnungen**, also Hinweise auf verdächtige Stellen. Dieses Buch ist mit **Alle** geprüft.

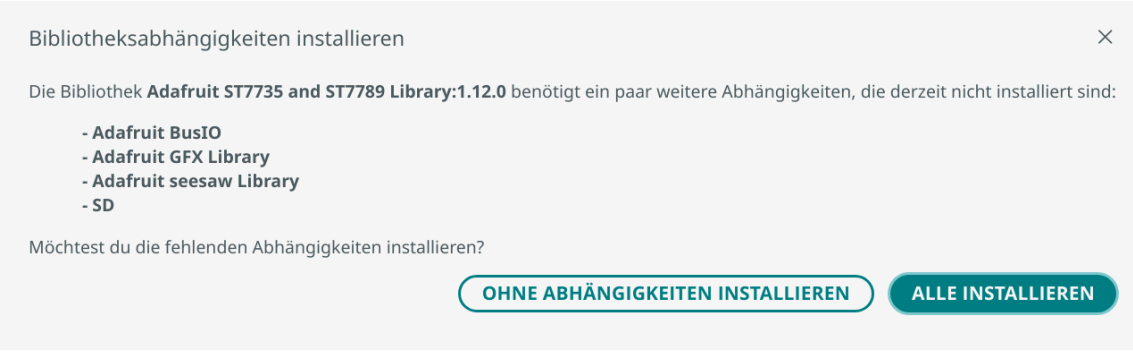
2.4 Bibliotheken installieren

Für Display, RGB-LED und Neigungssensor brauchst du drei Bibliotheken. Zwei davon installierst du mit dem **Bibliotheksverwalter** der IDE, die dritte aus einer ZIP-Datei.

Adafruit ST7735 and ST7789 (Display):

1. Öffne den **Bibliotheksverwalter**: **Werkzeuge → Bibliotheken verwalten …**, das dritte Symbol in der Seitenleiste oder **Strg + Umschalt + I**.
2. Suche nach **Adafruit ST7735 and ST7789** und klicke beim Eintrag **Adafruit ST7735 and ST7789 Library** auf **Installieren**.
3. Die IDE meldet, dass die Bibliothek weitere Bibliotheken braucht, ihre **Abhängigkeiten**: unter anderem **Adafruit GFX Library** (die eigentlichen Zeichenbefehle) und **Adafruit BusIO**. Klicke **Alle installieren**.

Adafruit NeoPixel (RGB-LED): Suche nach **Adafruit NeoPixel** und klicke **Installieren**.



Die IDE fragt nach den Abhängigkeiten. Klicke „Alle installieren“.

STK8BA58 (Neigungssensor): Für den Sensor des PBC+ gab es noch keine Arduino-Bibliothek; sie ist für dieses Buch geschrieben worden. Du bekommst sie als Datei `STK8BA58.zip` unter **pbcplus.de/downloads/STK8BA58.zip**.

1. Lade `STK8BA58.zip` herunter. Entpacke sie **nicht**.
2. Wähle **Sketch → Bibliothek einbinden → .ZIP-Bibliothek hinzufügen…** und dann die Datei `STK8BA58.zip`.
3. Unten erscheint die Meldung, dass die Bibliothek aus dem Archiv installiert wurde.

Alle Bibliotheken landen im Ordner `libraries` deines **Sketchbooks**. Das Sketchbook ist der Ordner, in dem die IDE deine Sketche ablegt. Wo er liegt, steht in **Datei → Einstellungen…** unter **Pfad für Sketchbook**; unter Windows und macOS ist es meist `Dokumente/Arduino`.

> Geprüft sind die Sketche mit Adafruit GFX Library 1.12.6, Adafruit ST7735 and ST7789 Library 1.11.0 und Adafruit NeoPixel 1.15.5. Bietet die IDE später Aktualisierungen an, kannst du sie ohne Weiteres installieren.

2.5 Hochladen

Jetzt kommt der erste Sketch auf den PicoBoy.

1. Öffne mit **Datei → Neu** (**Strg + N**) einen neuen Sketch. Er enthält schon ein kleines Gerüst. Lösche es und tippe stattdessen diesen Sketch ab (oder kopiere ihn aus `k03_hallo.ino` der Sketche zum Buch):

k03_hallo.ino

```
#include <Adafruit_ST7789.h>

#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);

void setup() {
    tft.init(240, 280);
    tft.fillScreen(ST77XX_BLACK);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);

    tft.setCursor(66, 128);
    tft.setTextColor(ST77XX_WHITE);
    tft.setTextSize(3);
    tft.print("Hallo!");
}

void loop() {
}
```

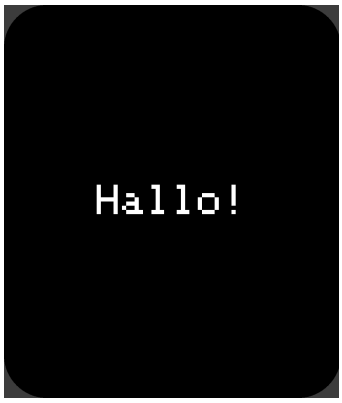
2. Speichere ihn mit **Datei → Speichern** (**Strg + S**) unter dem Namen `hallo`. Die IDE legt dabei den Ordner `hallo` mit der Datei `hallo.ino` an.
3. Klicke auf **Überprüfen** (Häkchen, **Strg + R**). Beim ersten Mal dauert das eine Weile, denn neben deinem Sketch übersetzt der Compiler auch den Kern und die Display-Bibliothek. Hat alles geklappt, steht im Ausgabefenster am Ende, wie viel Speicher der Sketch belegt, und unten rechts erscheint **Kompilieren erfolgreich**. Steht dort ein Fehler, hilft Kapitel 3.6.

Was die Zeilen bedeuten, lernst du in Kapitel 3. Jetzt geht es darum, dass der Sketch auf den PicoBoy kommt. Es gibt zwei Wege. Den ersten brauchst du nur beim allerersten Mal und wenn gar nichts mehr geht.

Weg 1: über den Bootmodus.

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. Wähle **Werkzeuge → Port → UF2 Board**. So nennt der Kern einen RP2350 im Bootmodus. Der Eintrag steht in einer eigenen Gruppe, nicht bei den seriellen Ports.
4. Klicke auf **Hochladen** (Pfeil, **Strg + U**). Die IDE kompiliert, sucht das Laufwerk und kopiert das Programm darauf.
5. Das Laufwerk verschwindet von selbst, der PicoBoy startet neu und zeigt „Hallo!“.

Jetzt läuft auf dem PicoBoy ein Arduino-Sketch. Er meldet sich am Computer als **serieller Port**: eine Verbindung über das USB-Kabel, über die Text hin- und hergehen kann (mehr in Kapitel 4). Wähle diesen Port unter **Werkzeuge → Port**. Unter Windows heißt er `COM` und eine Nummer, unter macOS `/dev/cu.usbmodem…`, unter Linux `/dev/ttyACM0`. Stehen mehrere zur Auswahl, zieh das Kabel kurz ab und sieh nach, welcher verschwindet.



Weg 2: mit einem Klick. Ist der serielle Port gewählt, klickst du nur noch auf **Hochladen**. Die IDE kompiliert, schickt dem laufenden Sketch über den Port ein verabredetes Signal (sie öffnet den Port kurz mit der Geschwindigkeit 1200 Baud), und der PicoBoy schaltet sich selbst in den Bootmodus. Dann kopiert die IDE das Programm wie bei Weg 1. Du musst also nur beim ersten Mal BOOT und RES drücken.

> So arbeitest du ab jetzt: **Sketch ändern → Hochladen (Strg + U).** Nach wenigen Sekunden läuft die neue Fassung auf dem PicoBoy.

Das Signal versteht nur ein Arduino-Sketch, weil der Kern das Zuhören in jeden Sketch einbaut. Läuft auf dem PicoBoy MicroPython oder ein Spiel, oder ist ein Sketch völlig abgestürzt, nimmst du wieder Weg 1.

2.6 Wenn es nicht klappt

Was passiert?	Was hilft?
In der Board-Verwaltung findest du „Raspberry Pi Pico/RP2040/RP2350“ nicht.	Die Adresse in **Zusätzliche Boardverwalter-URLs** prüfen (ein Tippfehler genügt), IDE neu starten.
Beim Überprüfen erscheint `fatal error: Adafruit_ST7789.h: No such file or directory`.	Die Bibliothek fehlt (Abschnitt 2.4). Bei `STK8BA58.h` entsprechend die ZIP-Bibliothek.
Unter Windows bricht das Kompilieren mit `cannot open linker script file` ab.	Der Windows-Benutzername enthält Umlaute oder Akzente (etwa „Jürgen“). Damit kommt der Compiler nicht zurecht. Abhilfe: ein zweites Windows-Benutzerkonto mit einfachem Namen anlegen und darin arbeiten.
Hochladen meldet `No drive to deploy.`	Der PicoBoy war nicht im Bootmodus und hat auch nicht auf das Signal reagiert. Weg 1 nehmen: BOOT halten, RES tippen, Port **UF2 Board** wählen.
Unter **Werkzeuge → Port** steht gar nichts.	Ein reines Ladekabel ohne Datenleitungen? Anderes Kabel, anderen USB-Anschluss probieren. Unter Windows nicht die Store-Fassung der IDE benutzen.
Das Laufwerk erscheint nicht.	BOOT wirklich festhalten, während du RES tippst.
Nach dem Hochladen bleibt der Bildschirm schwarz.	Fehlen im Sketch die Zeilen für das Hintergrundlicht (Kapitel 3.2)?
Das Ausgabefenster zeigt `error:` und eine Zeilennummer.	Ein Fehler im Sketch. Wie du ihn liest, steht in Kapitel 3.6.

Die IDE kann im Ausgabefenster genauer zeigen, was beim Hochladen passiert: In **Datei → Einstellungen…** gibt es Häkchen für eine ausführliche Ausgabe beim Kompilieren und beim Hochladen. Das hilft bei der Fehlersuche, macht die Ausgabe aber sehr lang.

Übungsaufgaben

Aufgabe 1 ⭐ **Ein anderer Gruß** Ändere im Sketch `hallo` das Wort `Hallo!` in `Moin!` oder einen anderen kurzen Gruß und lade den Sketch hoch.

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

Aufgabe 3 ⭐⭐ **Hin und zurück** Spiele MicroPython wieder auf (Kapitel 1.5) und danach wieder deinen Sketch. Welcher Weg aus Abschnitt 2.5 funktioniert, solange MicroPython läuft, welcher nicht? Welchen Port musst du wählen?

Aufgabe 4 ⭐⭐ **Wo liegt was?** Finde den Ordner deines Sketchbooks. Welche Ordner liegen darin im Unterordner `libraries`? Warum sind es mehr als die drei Bibliotheken, die du ausgewählt hast?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- **Arduino IDE 2** von arduino.cc installieren (unter Windows nicht aus dem Store).
- Kern: Boardverwalter-URL eintragen, in der **Board-Verwaltung** „Raspberry Pi Pico/RP2040/RP2350“ installieren.
- Board **Generic RP2350**, **Flash Size 16MB (Sketch: 8MB, FS: 8MB)**.
- Bibliotheken: **Adafruit ST7735 and ST7789** (mit allen Abhängigkeiten), **Adafruit NeoPixel**, **STK8BA58** als ZIP.
- **Überprüfen** (Strg + R) kompiliert, **Hochladen** (Strg + U) kompiliert und spielt auf.
- Beim ersten Mal: BOOT halten, RES tippen, Port **UF2 Board**. Danach den seriellen Port wählen, und ein Klick genügt.

KAPITEL 3

Der erste Sketch

Worum geht's? Du verstehst Zeile für Zeile, wie ein Sketch aufgebaut ist, schaltest das Display ein, schreibst Texte in verschiedenen Größen und Farben und lernst, wie der PicoBoy Positionen angibt. Außerdem liest du Fehlermeldungen des Compilers. Das wirst du beim Programmieren ständig brauchen.

3.1 `setup()` und `loop()`

Jeder Sketch besteht mindestens aus zwei Teilen. Ein neuer, leerer Sketch sieht etwa so aus:

k03_leer.ino

```
1 void setup() {
2     // Was hier steht, passiert einmal beim Start.
3 }
4
5 void loop() {
6     // Was hier steht, passiert danach immer wieder.
7 }
```

Die Zeilennummern gehören nicht zum Sketch. Die IDE zeigt sie links neben dem Text an, damit du dich orientieren kannst.

`setup` und `loop` sind **Funktionen**: benannte Abschnitte mit Anweisungen. Eine **Anweisung** ist ein einzelner Befehl an den PicoBoy, etwa „schreibe Hallo". Die Anweisungen einer Funktion stehen zwischen den **geschweiften Klammern** `{` und `}`. Einen solchen Abschnitt nennt man **Block**. Was `void` und die runden Klammern `()` bedeuten, lernst du in Kapitel 11. Bis dahin schreibst du die beiden Zeilen einfach immer so.

- **`setup()`** (englisch: Einrichtung) läuft **einmal**, gleich nach dem Einschalten oder Hochladen. Hier stellst du alles ein, was der Sketch braucht, und zeichnest zum Beispiel das erste Bild.
- **`loop()`** (englisch: Schleife) läuft danach **immer wieder**, ohne Ende. Ist die letzte Anweisung in `loop()` erledigt, beginnt `loop()` von vorn, Tausende Male pro Sekunde, bis du den Strom abschaltest. Hier steht später alles, was laufend passiert: Tasten abfragen, Figuren bewegen, Punkte zählen.

Beide Funktionen müssen in jedem Sketch vorkommen, auch wenn eine davon leer ist. Aufgerufen werden sie vom Kern: Er richtet beim Start den Chip und die USB-Verbindung ein, ruft einmal `setup()` auf und dann in einer Endlosschleife `loop()`. Ein Sketch endet also nie von selbst.

Alles hinter `//` ist ein **Kommentar**: eine Notiz für Menschen, die der Compiler überspringt. Mehr dazu in Abschnitt 3.5.

3.2 Das Display einschalten

Das ist der Sketch aus Kapitel 2:

k03_hallo.ino

```
#include <Adafruit_ST7789.h>

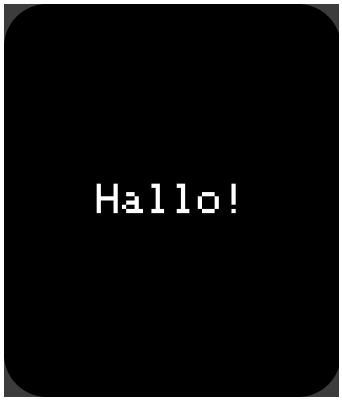
#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);

void setup() {
    tft.init(240, 280);
    tft.fillScreen(ST77XX_BLACK);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);

    tft.setCursor(66, 128);
    tft.setTextColor(ST77XX_WHITE);
    tft.setTextSize(3);
    tft.print("Hallo!");
}

void loop() {
}
```



Zeile 1: `#include <Adafruit_ST7789.h>` holt die Display-Bibliothek in deinen Sketch. Ohne diese Zeile kennt der Compiler keinen einzigen Befehl für das Display. `Adafruit_ST7789.h` ist eine Datei der Bibliothek, die aufzählt, welche Befehle es gibt. ST7789 ist der Name des Chips, der das Display steuert. Zeilen, die mit `#` beginnen, sind Anweisungen an den Compiler selbst, noch bevor er übersetzt.

Zeilen 3 bis 6: `#define` gibt einer Zahl einen Namen. Ab hier ersetzt der Compiler jedes `TFT_CS` durch `10`, jedes `TFT_LICHT` durch `26`. Die Zahlen sind die Nummern der **Pins**: der Anschlüsse des Chips, an die das Display angeschlossen ist. Der RP2350 hat 30 solcher Anschlüsse, GP0 bis GP29. Am PBC+ sind sie fest verdrahtet; Anhang D listet alle auf. Mit Namen statt Zahlen liest sich der Sketch leichter, und ein Tippfehler im Namen fällt dem Compiler auf.

Zeile 8 legt ein **Objekt** für das Display an. `Adafruit_ST7789` ist die Art des Objekts, `tft` sein Name (TFT ist die Bauart des Displays). In den Klammern stehen die drei Steuerleitungen: CS (*chip select*, „du bist gemeint“), DC (*data/command*, „jetzt kommen Daten“) und RST (*reset*, Neustart). Die Zeile steht außerhalb von `setup()`, damit `tft` im ganzen Sketch bekannt ist.

Zeile 11: `tft.init(240, 280);` startet das Display mit 240 × 280 Pixeln. Der Punkt zwischen `tft` und `init` bedeutet: Der Befehl `init` geht an das Objekt `tft`. Solche Befehle heißen **Funktionen**, genau wie `setup`. Die Zahlen in den Klammern sind die **Argumente**: die Angaben, die die Funktion zum Arbeiten braucht.

Zeile 12: `tft.fillScreen(ST77XX_BLACK);` füllt den ganzen Bildschirm schwarz. `init` löscht das Display nämlich nicht; ohne diese Zeile wäre noch zu sehen, was vorher im Speicher des Displays stand.

Zeilen 13 und 14 schalten das **Hintergrundlicht** ein. Das Display leuchtet nicht selbst. Hinter ihm sitzen Leuchtdioden, die an Pin 26 hängen. `pinMode(TFT_LICHT, OUTPUT)` macht den Pin zu einem **Ausgang**, der Strom liefern kann. `digitalWrite(TFT_LICHT, HIGH)` schaltet ihn ein (`HIGH` heißt: Spannung an). `pinMode` und `digitalWrite` gehören nicht zur Display-Bibliothek, sondern zu Arduino selbst; Kapitel 10 schaltet damit die LEDs.

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 den Sketch nicht mehr (Abschnitt 3.6). Groß- und Kleinschreibung zählt: `fillScreen` ist richtig, `fillscreen` kennt der Compiler nicht.

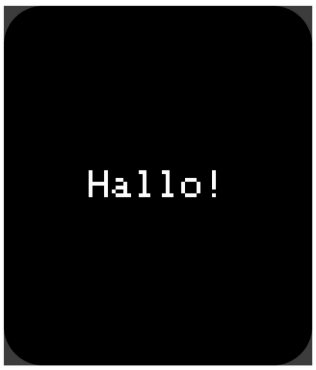
Die Einrückung, also die vier Leerzeichen am Zeilenanfang, ist für den Compiler bedeutungslos. Für Menschen ist sie wichtig: Man sieht sofort, was zu welchem Block gehört. Leere Zeilen teilen den Sketch in Abschnitte.

Ohne Licht bleibt alles schwarz. In diesem Sketch fehlen die beiden Zeilen für das Hintergrundlicht:

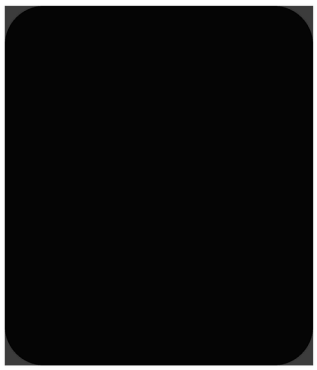
k03_ohne_licht.ino

```
10 void setup() {
11     tft.init(240, 280);
12     tft.fillScreen(ST77XX_BLACK);
13
14     tft.setCursor(66, 128);
15     tft.setTextColor(ST77XX_WHITE);
16     tft.setTextSize(3);
17     tft.print("Hallo!");
18 }
```

Er lässt sich fehlerfrei kompilieren und hochladen, und er schreibt auch „Hallo!“ in den Speicher des Displays. Zu sehen ist trotzdem nichts, denn ohne Hintergrundlicht ist das Display dunkel:



mit Hintergrundlicht



ohne Hintergrundlicht

> Jeder Sketch mit Display beginnt mit `tft.init(240, 280);`, füllt den Bildschirm und schaltet das Hintergrundlicht ein: `pinMode(TFT_LICHT, OUTPUT);` und `digitalWrite(TFT_LICHT, HIGH);`.

Was du zeichnest, ist sofort zu sehen. Jeder Befehl an `tft` geht gleich über eine Leitung zum Display, und das Display zeigt das Ergebnis, bis es überschrieben wird. Die Übertragung braucht aber Zeit: Den ganzen Bildschirm neu zu füllen dauert etwa 43 Millisekunden. Bei einem einzelnen Bild merkst du davon nichts. Sobald sich etwas bewegt, siehst du dem Display beim Zeichnen zu, und es flackert. Woher das kommt, zeigt Kapitel 7; wie es flimmerfrei geht, Kapitel 14.

3.3 Text: `setCursor`, `setTextColor`, `setTextSize`, `print`

Für einen Text sind vier Befehle zuständig (Zeilen 16 bis 19):

Befehl	Bedeutung
`tft.setCursor(66, 128)`	Hier beginnt der Text: 66 Pixel von links, 128 von oben
`tft.setTextColor(ST77XX_WHITE)`	in Weiß
`tft.setTextSize(3)`	in dreifacher Größe
`tft.print("Hallo!")`	schreibt den Text

Die Reihenfolge der ersten drei ist egal; sie müssen nur vor `print` stehen. Sie gelten so lange, bis du sie änderst. Der Text selbst steht in geraden Anführungszeichen `"…"`, wie sie die Tastatur mit **Umschalt + 2** schreibt. Ein Text in Anführungszeichen heißt **Zeichenkette** (englisch *string*).

Farben: Die Bibliothek kennt neun fertige Farben: `ST77XX_BLACK`, `ST77XX_WHITE`, `ST77XX_RED`, `ST77XX_GREEN`, `ST77XX_BLUE`, `ST77XX_CYAN`, `ST77XX_MAGENTA`, `ST77XX_YELLOW` und `ST77XX_ORANGE`. Namen in Großbuchstaben stehen in C++ meist für feste Werte, die sich nie ändern. Eigene Farben mischst du in Kapitel 6.

k03_farben.ino

```
10 void setup() {
11     tft.init(240, 280);
12     tft.fillScreen(ST77XX_BLUE);
13     pinMode(TFT_LICHT, OUTPUT);
14     digitalWrite(TFT_LICHT, HIGH);
15
16     tft.setTextSize(3);
17     tft.setCursor(40, 50);
18     tft.setTextColor(ST77XX_YELLOW);
19     tft.print("Gelb");
20     tft.setCursor(40, 90);
21     tft.setTextColor(ST77XX_WHITE);
22     tft.print("Weiss");
23     tft.setCursor(40, 130);
24     tft.setTextColor(ST77XX_CYAN);
```



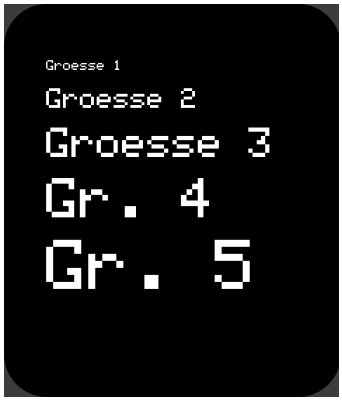
```
25      tft.print("Cyan");
26      tft.setCursor(40, 170);
27      tft.setTextColor(ST77XX_ORANGE);
28      tft.print("Orange");
29      tft.setCursor(40, 210);
30      tft.setTextColor(ST77XX_BLACK);
31      tft.print("Schwarz");
32  }
```

`print` zeichnet nur die Buchstaben selbst. Was dahinter liegt, bleibt stehen; deshalb ist hier der blaue Hintergrund zwischen den Buchstaben zu sehen. Gibst du `setTextColor` eine zweite Farbe mit, etwa `tft.setTextColor(ST77XX_WHITE, ST77XX_BLACK)`, füllt `print` den Platz hinter jedem Zeichen mit dieser Farbe. Das brauchst du später, um eine Zahl zu überschreiben.

Die Größe: Die eingebaute Schrift hat eine feste Breite. Bei Größe 1 ist jedes Zeichen genau **6 Pixel breit und 8 Pixel hoch**, ein Pixel Abstand eingeschlossen. Mit `setTextSize(2)` wird jedes Pixel der Schrift zu einem Quadrat aus 2 × 2 Pixeln, das Zeichen also 12 × 16 Pixel groß, bei Größe 3 18 × 24 Pixel und so weiter:

k03_groessen.ino

```
16      tft.setTextColor(ST77XX_WHITE);
17      tft.setCursor(30, 40);
18      tft.setTextSize(1);
19      tft.print("Groesse 1");
20      tft.setCursor(30, 60);
21      tft.setTextSize(2);
22      tft.print("Groesse 2");
23      tft.setCursor(30, 88);
24      tft.setTextSize(3);
25      tft.print("Groesse 3");
26      tft.setCursor(30, 124);
27      tft.setTextSize(4);
28      tft.print("Gr. 4");
29      tft.setCursor(30, 168);
30      tft.setTextSize(5);
31      tft.print("Gr. 5");
```



Größe	ein Zeichen	Zeichen pro Zeile (240 Pixel)
1	6 × 8	40
2	12 × 16	20
3	18 × 24	13
4	24 × 32	10
5	30 × 40	8

Größe 1 ist auf dem PicoBoy sehr klein, aber lesbar. Für die meisten Texte passt Größe 2. Je größer, desto deutlicher sieht man die Treppenstufen der Pixel. Schönere Schriften gibt es in Kapitel 6.

Mehrere Zeilen: `print` setzt den **Cursor**, die unsichtbare Schreibposition, hinter das letzte Zeichen. Der nächste `print` schreibt dort weiter. `println` (*print line*) springt nach dem Text in die nächste Zeile, und zwar an den **linken Rand**, nicht an die Stelle, an der der Text begann. Ein Text, der nicht mehr in die Zeile passt, läuft ebenfalls am linken Rand weiter:

k03_zeilen.ino

```
16      tft.setTextSize(2);
17      tft.setTextColor(ST77XX_WHITE);
18      tft.setCursor(30, 40);
19      tft.println("Zeile 1");
20      tft.println("Zeile 2");
21      tft.setTextColor(ST77XX_CYAN);
22      tft.setCursor(30, 120);
23      tft.print("Dieser Text ist zu lang fuer eine Zeile.");
```

„Zeile 2“ beginnt bei x = 0, und der lange Text wird mitten im Wort umbrochen. Willst du eine zweite Zeile genau unter der ersten, setzt du den Cursor selbst mit `setCursor`. Mit `tft.setTextWrap(false)` schaltest du den Umbruch aus; dann wird abgeschnitten, was nicht passt.



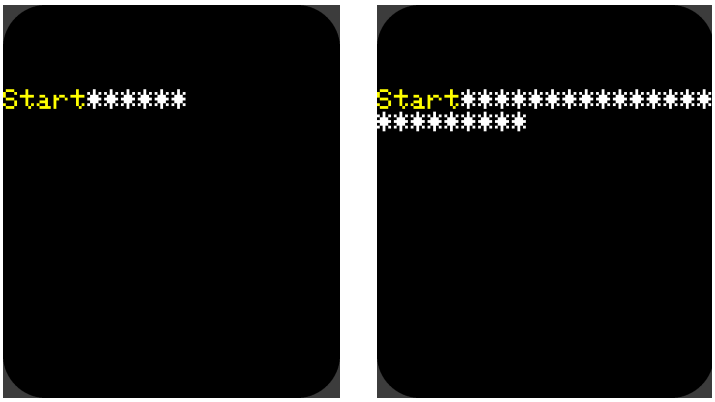
> Schreib auf dem Display „ae“, „oe“, „ue“ und „ss“ statt Umlauten und ß: `"Schoene Gruesse"`. Die Schrift kennt zwar ein paar Sonderzeichen, aber nicht in der Form, in der der Editor sie speichert. Statt „ü“ erscheinen deshalb andere Zeichen.

`loop()` in Aktion: Dieser Sketch schreibt in `setup()` „Start“ und in `loop()` einen Stern. `delay(500)` wartet 500 Millisekunden, also eine halbe Sekunde (mehr in Kapitel 7):

k03_sterne.ino

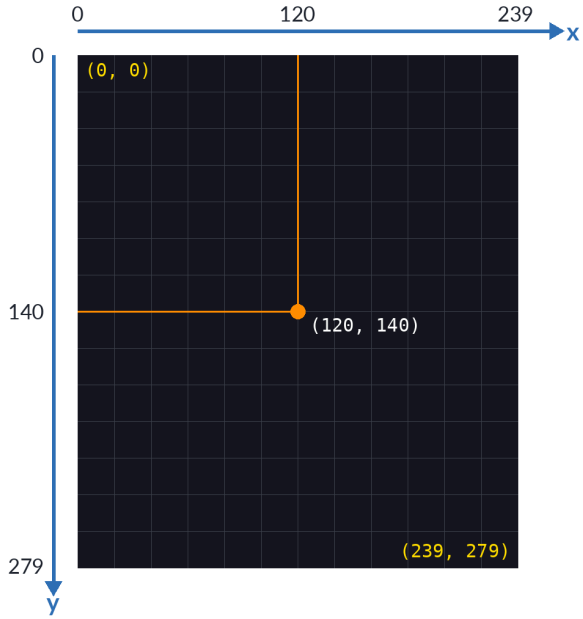
```
10 void setup() {
11     tft.init(240, 280);
12     tft.fillScreen(ST77XX_BLACK);
13     pinMode(TFT_LICHT, OUTPUT);
14     digitalWrite(TFT_LICHT, HIGH);
15
16     tft.setCursor(0, 60);
17     tft.setTextSize(2);
18     tft.setTextColor(ST77XX_YELLOW);
19     tft.print("Start");
20     tft.setTextColor(ST77XX_WHITE);
21 }
22
23 void loop() {
24     tft.print("*");
25     delay(500);
26 }
```

Weil `loop()` immer wieder läuft, kommt jede halbe Sekunde ein Stern dazu. Links siehst du den Bildschirm nach 3 Sekunden, rechts nach 12. Jeder Stern ist sofort sichtbar, und am rechten Rand geht es in der nächsten Zeile weiter. Nach gut zwei Minuten verschwinden die Sterne unten aus dem Bild: Was außerhalb des Bildschirms gezeichnet wird, lässt das Display einfach weg.



3.4 Koordinaten und runde Ecken

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**.

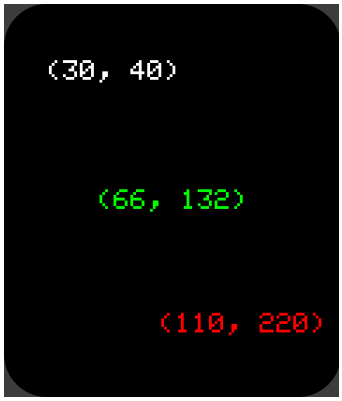


Anders als im Mathematikunterricht wächst y also **nach unten**. Das ist bei fast allen Bildschirmen so, denn ein Bild wird von oben nach unten aufgebaut, Zeile für Zeile. Der Bildschirm ist 240 Pixel breit und 280 Pixel hoch. x geht von 0 bis 239, y von 0 bis 279.

Bei einem Text gibt `setCursor(x, y)` die **linke obere Ecke** des ersten Zeichens an:

k03_koordinaten.ino

```
16      tft.setTextSize(2);
17      tft.setTextColor(ST77XX_WHITE);
18      tft.setCursor(30, 40);
19      tft.print("(30, 40)");
20      tft.setTextColor(ST77XX_GREEN);
21      tft.setCursor(66, 132);
22      tft.print("(66, 132)");
23      tft.setTextColor(ST77XX_RED);
24      tft.setCursor(110, 220);
25      tft.print("(110, 220)");
26  }
```

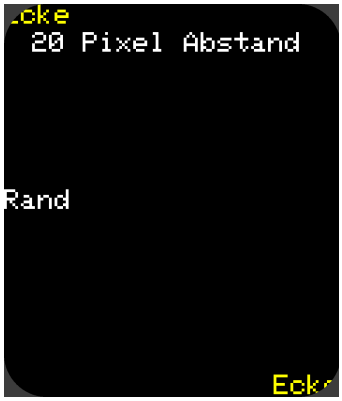


Text in die Mitte setzen: Weil jedes Zeichen gleich breit ist, kannst du die Breite eines Texts ausrechnen: Zeichenzahl × 6 × Größe. „Hallo!“ hat 6 Zeichen, bei Größe 3 sind das 6 × 6 × 3 = 108 Pixel. Damit links und rechts gleich viel Platz bleibt, beginnt der Text bei x = (240 − 108) / 2 = 66. Genauso die Höhe: 8 × 3 = 24 Pixel, also y = (280 − 24) / 2 = 128. Das sind die Zahlen aus `k03_hallo`.

Runde Ecken: Das Display hat abgerundete Ecken. Was zu nah an einer Ecke liegt, ist nicht zu sehen:

k03_ecken.ino

```
16      tft.setTextSize(2);
17      tft.setTextColor(ST77XX_YELLOW);
18      tft.setCursor(0, 0);
19      tft.print("Ecke");
20      tft.setCursor(192, 264);
21      tft.print("Ecke");
22      tft.setTextColor(ST77XX_WHITE);
23      tft.setCursor(0, 132);
24      tft.print("Rand");
25      tft.setCursor(20, 20);
26      tft.print("20 Pixel Abstand");
27  }
```



Die beiden Wörter „Ecke“ in der linken oberen und der rechten unteren Ecke sind angeschnitten. „Rand“ am linken Rand in der Mitte ist ganz zu sehen: Nur in den Ecken fehlt etwas.

> Halte mit allem Wichtigen mindestens **20 Pixel** Abstand von den Ecken.

3.5 Kommentare

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

k03_kommentare.ino

```
1  // Zeigt einen Gruss in zwei Farben.
2  #include <Adafruit_ST7789.h>
3
4  #define TFT_CS 10
5  #define TFT_DC 8
6  #define TFT_RST 9
7  #define TFT_LICHT 26   // Hintergrundlicht
8
9  Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
10
11 void setup() {
12     tft.init(240, 280);
13     tft.fillScreen(ST77XX_BLACK);
14     pinMode(TFT_LICHT, OUTPUT);
15     digitalWrite(TFT_LICHT, HIGH);   // ohne Licht bleibt alles schwarz
16
17     tft.setTextSize(3);
18     tft.setTextColor(ST77XX_GREEN);
19     tft.setCursor(40, 100);
20     tft.print("Guten");
21     tft.setTextColor(ST77XX_YELLOW);
22     tft.setCursor(40, 140);
23     tft.print("Morgen!");
24     // tft.print(" Abend!");
25     /* Dieser Kommentar
26        geht ueber mehrere Zeilen. */
27 }
28
29 void loop() {
30 }
```



- Zeile 1 ist ein Kommentar über den ganzen Sketch.
- In den Zeilen 7 und 15 steht der Kommentar hinter einer Anweisung. Die Anweisung gilt, der Kommentar wird übersprungen.
- Zeile 24 ist eine Anweisung, die „auskommentiert“ wurde. Sie bleibt im Sketch stehen, wird aber nicht ausgeführt. So kannst du etwas vorübergehend abschalten, ohne es zu löschen. In der IDE geht das schnell: Zeilen markieren und **Strg + /** drücken; noch einmal **Strg + /** schaltet sie wieder ein.
- Die Zeilen 25 und 26 sind ein Kommentar über zwei Zeilen.

Gute Kommentare erklären, **warum** etwas da steht („ohne Licht bleibt alles schwarz“), nicht, was ohnehin dasteht.

3.6 Fehlermeldungen beim Kompilieren lesen

Beim Programmieren macht jeder Fehler, auch Profis, jeden Tag. Wichtig ist, dass du sie schnell findest. Dabei hilft dir der **Compiler**: Er prüft den ganzen Sketch, bevor er auf den PicoBoy kommt. Findet er einen Fehler, bricht er ab und schreibt eine **Fehlermeldung** ins Ausgabefenster der IDE. Die IDE markiert die Zeile zusätzlich im Editor und zeigt unten rechts eine Kurzfassung.

Beispiel 1: ein fehlendes Semikolon

k03_fehler_semikolon.ino

```
16      tft.setCursor(66, 128);
17      tft.setTextColor(ST77XX_WHITE);
18      tft.setTextSize(3)
19      tft.print("Hallo!");
20  }
```

Ausgabe

```
k03_fehler_semikolon.ino: In function 'void setup()':
k03_fehler_semikolon.ino:18:23: error: expected ';' before 'tft'
   18 |      tft.setTextSize(3)
      |                      ^
      |                      ;
   19 |      tft.print("Hallo!");
      |      ~~~
Fehler beim Build: exit status 1
```

So liest du die Meldung:

1. Die erste Zeile sagt, wo der Fehler steckt: in der Funktion `setup()`.
2. `k03_fehler_semikolon.ino:18:23` nennt die **Datei**, die **Zeile** (18) und die **Spalte** (23), also das 23. Zeichen in der Zeile.
3. `error:` heißt: ein Fehler, der Sketch kann nicht kompiliert werden. Dahinter steht, was der Compiler erwartet hat: `expected ';' before 'tft'`, ein Semikolon vor dem `tft` der nächsten Zeile.
4. Darunter zeigt der Compiler die Zeile und markiert die Stelle mit `^`. Hier schlägt er sogar vor, was fehlt: das `;`.
5. Die letzte Zeile sagt nur, dass das Kompilieren abgebrochen wurde.

In der IDE steht vor dem Dateinamen noch der ganze Pfad zum Sketch-Ordner; im Buch ist er weggelassen.

Beispiel 2: ein Tippfehler im Namen

k03_fehler_tippfehler.ino

```
16      tft.setCursor(66, 128);
17      tft.setTextColour(ST77XX_WHITE);
18      tft.setTextSize(3);
19      tft.print("Hallo!");
20  }
```

Ausgabe

```
k03_fehler_tippfehler.ino: In function 'void setup()':
k03_fehler_tippfehler.ino:17:9: error: 'class Adafruit_ST7789' has no member named
  'setTextColour'; did you mean 'setTextColor'?
   17 |     tft.setTextColour(ST77XX_WHITE);
      |         ^~~~~~~~~~~~
      |         setTextColor
Fehler beim Build: exit status 1
```

Has no member named heißt: Das Objekt `tft` (genauer: die Art `Adafruit_ST7789`) kennt keinen Befehl mit diesem Namen. Der Compiler hat aber einen ähnlichen gefunden und fragt: *did you mean ‚setTextColor'?*, meintest du `setTextColor`? Die Bibliothek benutzt amerikanisches Englisch: *color*, nicht *colour*.

Beispiel 3: Groß- und Kleinschreibung

k03_fehler_gross.ino

```
16      tft.setCursor(66, 128);
17      tft.setTextColor(ST77XX_white);
18      tft.setTextSize(3);
19      tft.print("Hallo!");
20  }
```

Ausgabe

```
k03_fehler_gross.ino: In function 'void setup()':
k03_fehler_gross.ino:17:22: error: 'ST77XX_white' was not declared in this scope; did
  you mean 'ST77XX_WHITE'?
   17 |     tft.setTextColor(ST77XX_white);
      |                      ^~~~~~~~~~~~
      |                      ST77XX_WHITE
Fehler beim Build: exit status 1
```

C++ unterscheidet streng zwischen Groß- und Kleinbuchstaben. `ST77XX_WHITE` ist eine Farbe, `ST77XX_white` gibt es nicht. *Was not declared in this scope* bedeutet „ist hier nicht bekannt“. Auch hier steht der richtige Name gleich darunter.

Beispiel 4: eine Klammer fehlt

k03_fehler_klammer.ino

```
10 void setup() {
11     tft.init(240, 280);
12     tft.fillScreen(ST77XX_BLACK);
13     pinMode(TFT_LICHT, OUTPUT);
14     digitalWrite(TFT_LICHT, HIGH);
15
16     tft.setCursor(66, 128);
17     tft.setTextColor(ST77XX_WHITE);
18     tft.setTextSize(3);
19     tft.print("Hallo!");
20
21 void loop() {
22 }
```

Ausgabe

```
k03_fehler_klammer.ino: In function 'void setup()':
k03_fehler_klammer.ino:21:13: error: a function-definition is not allowed here before
  '{' token
   21 | void loop() {
      |             ^
k03_fehler_klammer.ino:22:2: error: expected '}' at end of input
   22 | }
      |  ^
k03_fehler_klammer.ino:10:14: note: to match this '{'
   10 | void setup() {
      |              ^
Fehler beim Build: exit status 1
```

Hier fehlt die `}`, die `setup()` schließt. Der Compiler hält deshalb `void loop() {` für einen Teil von `setup()` und beschwert sich, dass man innerhalb einer Funktion keine neue Funktion anlegen darf (*a function-definition is not allowed here*). Am Ende der Datei merkt er, dass noch eine `}` fehlt (*expected ‚}' at end of input*). Die Zeile mit `note:` ist ein Hinweis, kein weiterer Fehler: Sie zeigt die `{` in Zeile 10, zu der die schließende Klammer fehlt. Der Fehler liegt also weiter oben als die gemeldeten Zeilen.

Beispiel 5: die Bibliothek fehlt

k03_fehler_include.ino

```
1 #define TFT_CS 10
2 #define TFT_DC 8
3 #define TFT_RST 9
4 #define TFT_LICHT 26
5
6 Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
7
8 void setup() {
9     tft.init(240, 280);
```

```
10      tft.fillScreen(ST77XX_BLACK);
11      pinMode(TFT_LICHT, OUTPUT);
12      digitalWrite(TFT_LICHT, HIGH);
```

Ausgabe

```
k03_fehler_include.ino:6:1: error: 'Adafruit_ST7789' does not name a type
    6 | Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
      | ^~~~~~~~~~~~~~~
k03_fehler_include.ino: In function 'void setup()':
k03_fehler_include.ino:9:5: error: 'tft' was not declared in this scope
    9 |     tft.init(240, 280);
      |     ^~~
k03_fehler_include.ino:10:20: error: 'ST77XX_BLACK' was not declared in this scope
   10 |     tft.fillScreen(ST77XX_BLACK);
      |                    ^~~~~~~~~~~~
k03_fehler_include.ino:15:22: error: 'ST77XX_WHITE' was not declared in this scope
   15 |     tft.setTextColor(ST77XX_WHITE);
      |                      ^~~~~~~~~~~~
Fehler beim Build: exit status 1
```

Hier fehlt die Zeile `#include <Adafruit_ST7789.h>`. Der Compiler kennt `Adafruit_ST7789` deshalb nicht (*does not name a type*: ist keine bekannte Art). Alle weiteren Meldungen sind Folgefehler: Weil `tft` nicht angelegt werden konnte, ist auch `tft` unbekannt, und die Farbnamen stammen ebenfalls aus der Bibliothek. Vier Meldungen, ein einziger Fehler.

- 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 kompiliere neu. Spätere Meldungen sind oft nur Folgen des ersten Fehlers.
- Neben `error:` gibt es `warning:`. Eine **Warnung** hält das Kompilieren nicht auf, weist aber auf eine verdächtige Stelle hin. Nimm Warnungen ernst. Damit du sie siehst, muss in den Einstellungen **Compiler-Meldungen** auf **Alle** stehen (Kapitel 2.3).

Die häufigsten Fehler am Anfang:

Meldung	Bedeutung	Typische Ursache
`expected ';' before …`	Semikolon fehlt	`;` am Ende der Zeile davor vergessen
`… has no member named …`	das Objekt kennt diesen Befehl nicht	Befehl falsch geschrieben
`… was not declared in this scope`	ein Name ist unbekannt	Name falsch geschrieben, Groß- und Kleinschreibung

Meldung	Bedeutung	Typische Ursache
`… does not name a type`	eine Art ist unbekannt	`#include` vergessen, Bibliothek nicht installiert
`expected '}' at end of input`	die Datei endet zu früh	eine `}` fehlt
`a function-definition is not allowed here`	eine Funktion steht in einer anderen	die `}` der Funktion davor fehlt

Ausprobieren: Eine Visitenkarte

k03_visitenkarte.ino

```
10
11 void setup() {
12     tft.init(240, 280);
13     tft.fillScreen(ST77XX_WHITE);
14     pinMode(TFT_LICHT, OUTPUT);
15     digitalWrite(TFT_LICHT, HIGH);
16
17     // Name
18     tft.setTextSize(3);
19     tft.setTextColor(ST77XX_BLUE);
20     tft.setCursor(30, 50);
21     tft.print("Mia");
22     tft.setCursor(30, 80);
23     tft.print("Beispiel");
24
25     // Angaben
26     tft.setTextSize(2);
27     tft.setTextColor(ST77XX_BLACK);
28     tft.setCursor(30, 130);
29     tft.print("Hobby: Gitarre");
30     tft.setCursor(30, 155);
31     tft.print("Mag: Pizza");
32     tft.setCursor(30, 180);
33     tft.print("Kann: Arduino");
34
35     // Fusszeile
36     tft.setTextSize(1);
37     tft.setCursor(66, 240);
38     tft.print("PicoBoy Color Plus");
```



- Zeile 12 füllt den Bildschirm weiß. Darauf ist schwarze und blaue Schrift gut zu lesen.
- Leere Zeilen und Kommentare teilen `setup()` in Abschnitte: Name, Angaben, Fußzeile.
- Die drei Angaben stehen je 25 Pixel untereinander (y = 130, 155, 180). Bei Größe 2 sind die Zeichen 16 Pixel hoch; so bleiben 9 Pixel Luft.
- Die Fußzeile ist mit Größe 1 genau mittig: 18 Zeichen × 6 = 108 Pixel, x = (240 − 108) / 2 = 66.

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

Übungsaufgaben

Aufgabe 1 ⭐ **Wo erscheint was?** Lies den Sketch, ohne ihn hochzuladen. Wo auf dem Bildschirm erscheinen die drei Wörter, wie groß und in welcher Farbe? Skizziere den Bildschirm und prüfe danach auf dem PicoBoy.

k03_aufgabe1.ino

```
16      tft.setTextSize(2);
17      tft.setTextColor(ST77XX_WHITE);
18      tft.setCursor(30, 30);
19      tft.print("oben");
20      tft.setTextSize(3);
21      tft.setTextColor(ST77XX_GREEN);
22      tft.setCursor(75, 128);
23      tft.print("Mitte");
24      tft.setTextSize(2);
25      tft.setTextColor(ST77XX_RED);
26      tft.setCursor(150, 230);
27      tft.print("unten");
28  }
```

Aufgabe 2 ⭐ **Drei Zeilen über dich** Schreibe einen Sketch, der deinen Vornamen groß (Größe 4) und darunter zwei Angaben über dich (Größe 2) zeigt, alles in verschiedenen Farben.

Aufgabe 3 ⭐⭐ **Finde die Fehler** Dieser Sketch enthält drei Fehler. Kompiliere ihn, lies die Meldungen, verbessere einen Fehler nach dem anderen und kompiliere jedes Mal neu. Wie viele Fehler meldet der Compiler beim ersten Mal, und welcher Hinweis hilft dir jeweils?

k03_aufgabe3.ino

```
16      tft.setTextSize(3);
17      tft.setTextcolor(ST77XX_YELLOW);
18      tft.setCursor(40, 80);
19      tft.print("Treffen")
20      tft.setTextSize(2);
21      tft.setTextColor(ST77XX_white);
22      tft.setCursor(40, 130);
23      tft.print("Heute 14 Uhr");
24      tft.setCursor(40, 160);
25      tft.print("Raum 2.04");
26  }
```

Aufgabe 4 ⭐⭐ **Ein Schild** Schreibe einen Sketch, der den Bildschirm rot füllt und genau in der Mitte in Größe 6 „STOP“ in Weiß zeigt. Darunter steht in Größe 2 „Bitte warten“, ebenfalls waagerecht in der Mitte. Rechne die Positionen aus, statt sie auszuprobieren.

Aufgabe 5 ⭐⭐⭐ **Stundenplan** Zeige deinen Stundenplan für einen Tag als Tabelle mit den Spalten Stunde, Fach und Raum, in Größe 2. Überlege: Wie viele Leerzeichen brauchst du, damit die Spalten genau untereinander stehen? Wie viele Zeichen passen höchstens in eine Zeile, wenn links und rechts je 20 Pixel frei bleiben sollen? Wie groß muss der Abstand der Zeilen mindestens sein?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Jeder Sketch hat `setup()` (einmal beim Start) und `loop()` (danach immer wieder). Anweisungen enden mit `;`, geschweifte Klammern `{ }` bilden einen **Block**.
- Display: `#include <Adafruit_ST7789.h>`, Objekt `Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);`, dann `tft.init(240, 280);`, `tft.fillScreen(…)` und das **Hintergrundlicht** an Pin 26 einschalten.
- Text: `setCursor(x, y)`, `setTextColor(farbe)`, `setTextSize(n)`, `print("…")`. Ein Zeichen ist 6 × 8 Pixel mal Größe.
- Gezeichnetes ist **sofort** sichtbar.
- Koordinaten: x von links (0–239), y von **oben** (0–279). Runde Ecken: 20 Pixel Abstand halten.
- `//` und `/* … */` sind Kommentare.
- Fehlermeldungen: `name.ino:Zeile:Spalte: error: …`. Den ersten Fehler zuerst verbessern.

KAPITEL 4

Der serielle Monitor

Worum geht's? Der PicoBoy kann nicht nur auf sein Display schreiben, sondern auch Text über das USB-Kabel an den Computer schicken und von dort Eingaben empfangen. Du lernst den seriellen Monitor der Arduino IDE kennen, gibst Texte und Zahlen mit `Serial` aus, liest Eingaben und lässt Messwerte als Kurve zeichnen. Das ist praktisch, um längere Ausgaben zu lesen und um nachzusehen, was ein Sketch gerade tut.

4.1 `Serial.begin`, `Serial.print`, `Serial.println`

Solange ein Arduino-Sketch läuft, meldet sich der PicoBoy am Computer als **serieller Port** (Kapitel 2.5): eine Leitung, über die Text Zeichen für Zeichen hin- und hergeht. Im Sketch heißt diese Verbindung `Serial`. Ein Fenster auf dem Computer, das den Text anzeigt und Eingaben zurückschickt, heißt **serieller Monitor**.

k04_hallo_monitor.ino

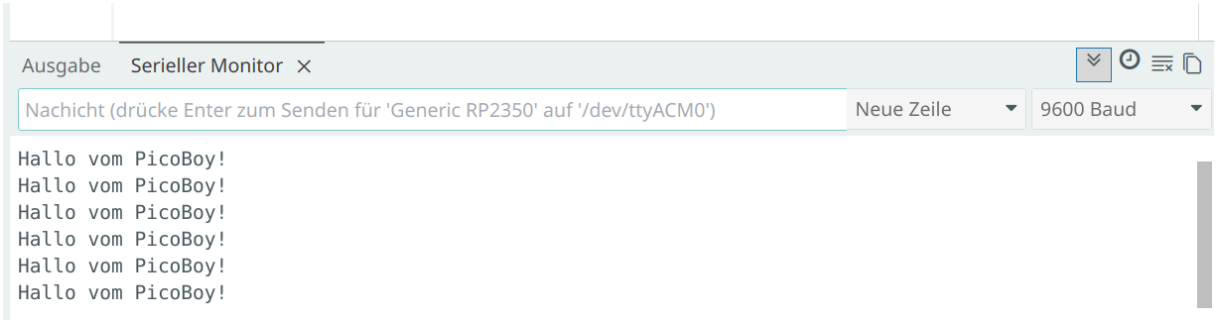
```
1 void setup() {
2     Serial.begin(115200);
3 }
4
5 void loop() {
6     Serial.println("Hallo vom PicoBoy!");
7     delay(1000);
8 }
```

- `Serial.begin(115200)` startet die Verbindung. Die Zahl ist die Geschwindigkeit in **Baud** (Zeichenschritten pro Sekunde). Bei vielen Arduino-Platinen muss sie mit der Einstellung im Monitor übereinstimmen. Beim PicoBoy läuft `Serial` über USB, und die Zahl spielt keine Rolle. Die Zeile gehört trotzdem in jeden Sketch, der `Serial` benutzt, als erste Anweisung in `setup()`.
- `Serial.println("…")` schickt einen Text und beginnt danach eine neue Zeile.
- Weil die Ausgabe in `loop()` steht, kommt jede Sekunde eine Zeile dazu.

So öffnest du den seriellen Monitor:

1. Lade den Sketch hoch. Unter **Werkzeuge → Port** muss der serielle Port des PicoBoy gewählt sein.
2. Wähle **Werkzeuge → Serieller Monitor**, klicke auf das Symbol oben rechts im Fenster oder drücke **Strg + Umschalt + M**. Der Monitor erscheint unten als eigener Bereich neben der Ausgabe.
3. Rechts in der Leiste des Monitors steht eine Baudrate, etwa `9600 Baud`. Beim PicoBoy kannst du sie lassen, wie sie ist. Nur **1200 Baud** darfst du nicht wählen: Das ist das Signal, mit dem die IDE den PicoBoy in den Bootmodus schaltet.

```
Serieller Monitor
Hallo vom PicoBoy!
Hallo vom PicoBoy!
Hallo vom PicoBoy!
Hallo vom PicoBoy!
```



Der serielle Monitor: oben die Eingabezeile, rechts daneben Zeilenende und Baudrate, darunter die Ausgabe des PicoBoy.

Beim Hochladen startet der PicoBoy neu, und der Port verschwindet kurz. Lass den Monitor dabei offen: Er verbindet sich danach von selbst wieder.

Ausgaben vor dem Öffnen gehen verloren. Der PicoBoy schickt Text nur ab, wenn auf dem Computer ein Monitor zuhört. Was der Sketch vorher ausgibt, verwirft er. Das ist auch nach jedem Hochladen so: Der Sketch startet sofort, der Monitor verbindet sich erst einen Moment später. Bei einem Sketch, der jede Sekunde etwas ausgibt, stört das nicht. Was aber nur einmal in `setup()` ausgegeben wird, sähest du so nie. Dagegen hilft eine Warteschleife am Anfang von `setup()`:

```
void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    Serial.println("Jetzt hoert jemand zu.");
}
```

`while (!Serial)` heißt: Solange kein Monitor verbunden ist, wiederhole den Block darunter. `delay(10)` wartet jeweils 10 Millisekunden. Erst wenn du den Monitor öffnest, geht es weiter. Wie `while` genau funktioniert, zeigt Kapitel 7; bis dahin schreibst du diese vier Zeilen einfach so ab.

> Ein Sketch mit `while (!Serial)` wartet, bis ein Monitor zuhört, notfalls für immer. Läuft der PicoBoy ohne Computer mit Batterien, bleibt er an dieser Stelle hängen. Benutze die Warteschleife deshalb nur in Sketchen, die ohne seriellen Monitor ohnehin nichts tun, wie den Beispielen in diesem Kapitel.

`print` und `println`: `Serial.print` schickt Text, ohne eine neue Zeile anzufangen. Der nächste Text schließt direkt an. `Serial.println` hängt am Ende einen Zeilenumbruch an; ohne Text in den Klammern schickt es nur den Zeilenumbruch.

k04_print_println.ino

```
 1 void setup() {
 2     Serial.begin(115200);
 3     while (!Serial) {
 4         delay(10);
 5     }
 6     Serial.print("Hallo");
 7     Serial.print("Welt");
 8     Serial.println();
 9     Serial.println("Eins\nZwei\nDrei");
10     Serial.println("Name:\tMia");
11     Serial.println("Alter:\t15");
12 }
13
14 void loop() {
15 }
```

Serieller Monitor

```
HalloWelt
Eins
Zwei
Drei
Name:     Mia
Alter:    15
```

- Die Zeilen 6 und 7 schreiben „Hallo“ und „Welt“ ohne Lücke hintereinander, Zeile 8 beendet die Zeile.
- In Zeile 9 steht `\n`. Das ist ein einziges Zeichen, auch wenn man es mit zwei Zeichen schreibt: der **Zeilenumbruch** (*newline*). Ein Text mit zwei `\n` erscheint auf drei Zeilen.
- `\t` in den Zeilen 10 und 11 ist ein **Tabulator**: ein Sprung zur nächsten Spalte. So stehen „Mia“ und „15“ untereinander.

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**.

Zahlen: `print` und `println` geben auch Zahlen aus. Steht dort eine Rechnung, rechnet der PicoBoy sie aus und schickt das Ergebnis:

k04_zahlen.ino

```
 1 void setup() {
 2     Serial.begin(115200);
 3     while (!Serial) {
 4         delay(10);
 5     }
 6     Serial.println(42);
 7     Serial.println(12 + 30);
 8     Serial.print("12 + 30 = ");
 9     Serial.println(12 + 30);
10     Serial.println(3.14159);
11     Serial.println(3.14159, 4);
12 }
```

Serieller Monitor

```
42
42
12 + 30 = 42
3.14
3.1416
```

Kommazahlen schreibt man in C++ mit **Punkt**: `3.14159`, nicht `3,14159`. `println` gibt sie mit **zwei** Nachkommastellen aus, gerundet. Eine zweite Zahl in den Klammern legt die Stellenzahl fest: `Serial.println(3.14159, 4)` ergibt vier Nachkommastellen.

4.2 `Serial.printf` mit Platzhaltern

Soll in einem Satz eine Zahl stehen, brauchst du mit `print` mehrere Zeilen. Kürzer geht es mit **`Serial.printf`** (*print formatted*, formatiert ausgeben). Der Text enthält **Platzhalter**, die mit `%` beginnen. Für jeden Platzhalter gibst du hinter einem Komma einen Wert an, der an dieser Stelle eingesetzt wird:

k04_printf.ino

```ino
void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    Serial.printf("Ich bin %d Jahre alt.\n", 15);
    Serial.printf("%d + %d = %d\n", 12, 30, 12 + 30);
    Serial.printf("Pi ist ungefaehr %f\n", 3.14159);
    Serial.printf("Mit zwei Stellen: %.2f\n", 3.14159);
    Serial.printf("Hallo %s!\n", "Mia");
    Serial.printf("100 %% geschafft\n");
}

void loop() {
}
```

Serieller Monitor

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

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

Beachte:

- `printf` beginnt keine neue Zeile von selbst. Am Ende steht deshalb fast immer `\n`.
- In Zeile 7 stehen drei Platzhalter. Die Werte werden **der Reihe nach** eingesetzt: der erste Wert für den ersten Platzhalter und so weiter. Hinter dem Komma darf auch eine Rechnung stehen, wie `12 + 30`.

> `printf` gehört eigentlich nicht zu Arduino, sondern stammt aus der Sprache C. Der Kern für den RP2350 hat es `Serial` zusätzlich spendiert. Bei manchen anderen Arduino-Platinen gibt es `Serial.printf` nicht.

Der Platzhalter muss zum Wert passen. Hier steht `%d` für eine ganze Zahl, der Wert ist aber die Kommazahl `7.5`:

k04_falscher_platzhalter.ino

```
6     Serial.printf("Preis: %d Euro\n", 7.5);
```

Der Sketch lässt sich ohne Fehler kompilieren, und der Compiler warnt nicht einmal, auch nicht mit **Compiler-Meldungen: Alle**. Bei `Serial.printf` prüft er die Platzhalter nicht. Auf dem PicoBoy erscheint dann statt `7.5` eine sinnlose Zahl, denn `printf` liest an der Stelle des Werts etwas ganz anderes. Prüfe die Platzhalter also selbst, Wert für Wert.

> `%d` für ganze Zahlen, `%f` oder `%.2f` für Kommazahlen, `%s` für Text. Ein falscher Platzhalter liefert Unsinn, ohne dass dich jemand warnt.

4.3 Eingaben lesen

Der serielle Monitor kann auch Text an den PicoBoy schicken. Oben im Monitor gibt es dafür ein Eingabefeld (**Nachricht**). Was du dort tippst, schickt der Monitor ab, sobald du **Enter** drückst. In der Ausgabe erscheint dein Text nicht; die zeigt nur, was der PicoBoy schickt.

Das Zeilenende: Neben dem Eingabefeld legt eine Auswahl fest, was der Monitor beim Enter an deinen Text anhängt:

Einstellung	angehängt wird
Kein Zeilenende	nichts
Neue Zeile	`\n` (voreingestellt)
Zeilenumbruch (CR)	`\r`, das „Wagenrücklauf“-Zeichen
Beides CR/LF	`\r\n`

Für dieses Buch stellst du **Neue Zeile** ein. Dann weiß der Sketch, wo deine Eingabe zu Ende ist.

Text lesen: Die Zeichen, die ankommen, sammelt der PicoBoy in einem Zwischenspeicher, bis der Sketch sie abholt. `Serial.available()` sagt, wie viele Zeichen dort warten. `Serial.readStringUntil('\n')` holt Zeichen bis zum Zeilenende ab und gibt sie als Text zurück; das `\n` selbst wird verbraucht, aber nicht mitgeliefert.

Um die Eingabe aufzubewahren, braucht der Sketch einen Platz mit Namen, eine **Variable**. Variablen lernst du in Kapitel 5 genau kennen. Hier reicht: `String name = …;` legt eine Variable für Text an, die `name` heißt, und legt gleich hinein, was rechts vom `=` steht.

k04_eingabe.ino

```
1  void setup() {
2      Serial.begin(115200);
3      while (!Serial) {
4          delay(10);
5      }
6      Serial.println("Wie heisst du?");
7      while (Serial.available() == 0) {
8          delay(10);
9      }
10     String name = Serial.readStringUntil('\n');
11     name.trim();
12     Serial.print("Hallo ");
13     Serial.print(name);
14     Serial.println(", schoen dich kennenzulernen!");
15 }
16
17 void loop() {
18 }
```

Serieller Monitor

```
Wie heisst du?
Hallo Mia, schoen dich kennenzulernen!
```

Eingegeben wurde `Mia`.

- Die Zeilen 7 bis 9 warten, bis etwas ankommt: Solange `Serial.available()` null ist, wartet der Sketch 10 Millisekunden und sieht wieder nach. Dieselben drei Zeilen stehen vor jeder Eingabe.
- Zeile 10 liest die Zeile und legt sie in die Variable `name`.

- `name.trim()` in Zeile 11 schneidet Leerzeichen und unsichtbare Steuerzeichen an Anfang und Ende ab. Steht das Zeilenende auf **Beides CR/LF**, bliebe sonst das `\r` am Namen hängen.
- Die Zeilen 12 bis 14 geben den Namen in einem Satz aus.

> Warum die Warteschleife? `readStringUntil` wartet nicht beliebig lange. Kommt **eine Sekunde** lang kein weiteres Zeichen, gibt es zurück, was bis dahin da ist, notfalls einen leeren Text. Ohne Warteschleife wäre die Frage also nach einer Sekunde vorbei, bevor du fertig getippt hast. Diese Wartezeit heißt **Timeout**. Mit `Serial.setTimeout(5000)` stellst du sie auf 5 Sekunden.

Zahlen lesen: `Serial.parseInt()` liest eine ganze Zahl, `Serial.parseFloat()` eine Kommazahl. Beide überspringen zuerst alles, was keine Ziffer ist, lesen dann die Ziffern (und ein Minus davor) und hören beim ersten Zeichen auf, das nicht mehr dazugehört. `int zahl = …` legt eine Variable für eine ganze Zahl an, `float preis = …` eine für eine Kommazahl:

k04_zahl_eingeben.ino

```
 1 void setup() {
 2     Serial.begin(115200);
 3     while (!Serial) {
 4         delay(10);
 5     }
 6     Serial.println("Gib eine ganze Zahl ein:");
 7     while (Serial.available() == 0) {
 8         delay(10);
 9     }
10     int zahl = Serial.parseInt();
11     Serial.readStringUntil('\n');   // Rest der Zeile wegwerfen
12     Serial.printf("Das Doppelte von %d ist %d.\n", zahl, zahl * 2);
13
14     Serial.println("Preis ohne Mehrwertsteuer in Euro?");
15     while (Serial.available() == 0) {
16         delay(10);
17     }
18     float preis = Serial.parseFloat();
19     Serial.readStringUntil('\n');
20     Serial.printf("Mit 19 %% Mehrwertsteuer: %.2f Euro\n", preis * 1.19);
21 }
22
23 void loop() {
24 }
```

Serieller Monitor

```
Gib eine ganze Zahl ein:
Das Doppelte von 21 ist 42.
Preis ohne Mehrwertsteuer in Euro?
Mit 19 % Mehrwertsteuer: 4.76 Euro
```

Eingegeben wurden `21` und `4.00`.

`parseInt` hört beim Zeilenende auf, **lässt es aber liegen**. Deshalb wirft `Serial.readStringUntil('\n');` in Zeile 11 den Rest der Zeile weg. Hier wird das Ergebnis von `readStringUntil` gar nicht aufgehoben; die Zeile ist nur dazu da, das `\n` abzuholen.

Was passiert ohne diese Zeile? Dann liegt das `\n` beim nächsten Lesen noch im Zwischenspeicher:

k04_rest_der_zeile.ino

```
 6     Serial.println("Wie alt bist du?");
 7     while (Serial.available() == 0) {
 8         delay(10);
 9     }
10     int alter = Serial.parseInt();
11
12     Serial.println("Wie heisst du?");
13     while (Serial.available() == 0) {
14         delay(10);
15     }
16     String name = Serial.readStringUntil('\n');
17     name.trim();
18     Serial.print("Hallo ");
19     Serial.print(name);
20     Serial.printf(", du bist %d.\n", alter);
21 }
```

Serieller Monitor

```
Wie alt bist du?
Wie heisst du?
Hallo , du bist 15.
```

Eingegeben wurde `15`, danach hatte niemand mehr Gelegenheit, den Namen einzugeben. Die Warteschleife in den Zeilen 13 bis 15 hält nicht an, weil schon ein Zeichen wartet: das liegen gebliebene `\n`. `readStringUntil` findet sofort das Zeilenende und liefert einen leeren Text.

> Nach `parseInt` oder `parseFloat` immer den Rest der Zeile wegwerfen: `Serial.readStringUntil('\n');`

Der Timeout bei Zahlen: Auch `parseInt` und `parseFloat` warten höchstens eine Sekunde. Kommt bis dahin keine Ziffer, liefern sie **0**. Ohne Warteschleife sieht das so aus:

k04_ohne_warten.ino

```
 6     Serial.println("Gib eine Zahl ein:");
 7     int zahl = Serial.parseInt();
 8     Serial.printf("Du hast %d eingegeben.\n", zahl);
```

Serieller Monitor

```
Gib eine Zahl ein:
Du hast 0 eingegeben.
```

Eine Sekunde nach der Frage ist alles vorbei, ohne dass du etwas eingeben konntest. Und eine 0 sieht aus wie eine echte Eingabe. Deshalb steht vor jeder Eingabe die Warteschleife mit `Serial.available()`.

> `parseFloat` versteht nur den **Punkt** als Komma: `4.50`. Bei `4,50` liest es `4` und hört beim Komma auf.

4.4 Der serielle Plotter

Zahlen, die sich ständig ändern, liest man schlecht als Zahlenkolonne. Der **serielle Plotter** der IDE zeichnet sie stattdessen als Kurven. Du öffnest ihn mit **Werkzeuge → Serieller Plotter** oder dem Symbol oben rechts im Fenster. Er erscheint in einem eigenen Fenster.

Der Plotter versteht Zeilen mit Zahlen. Jede Zeile ist ein Messpunkt. Stehen mehrere Zahlen in einer Zeile, getrennt durch Komma, Leerzeichen oder Tabulator, zeichnet er mehrere Kurven. Schreibst du `Name:Wert`, bekommt jede Kurve einen Namen (ohne Leerzeichen um den Doppelpunkt):

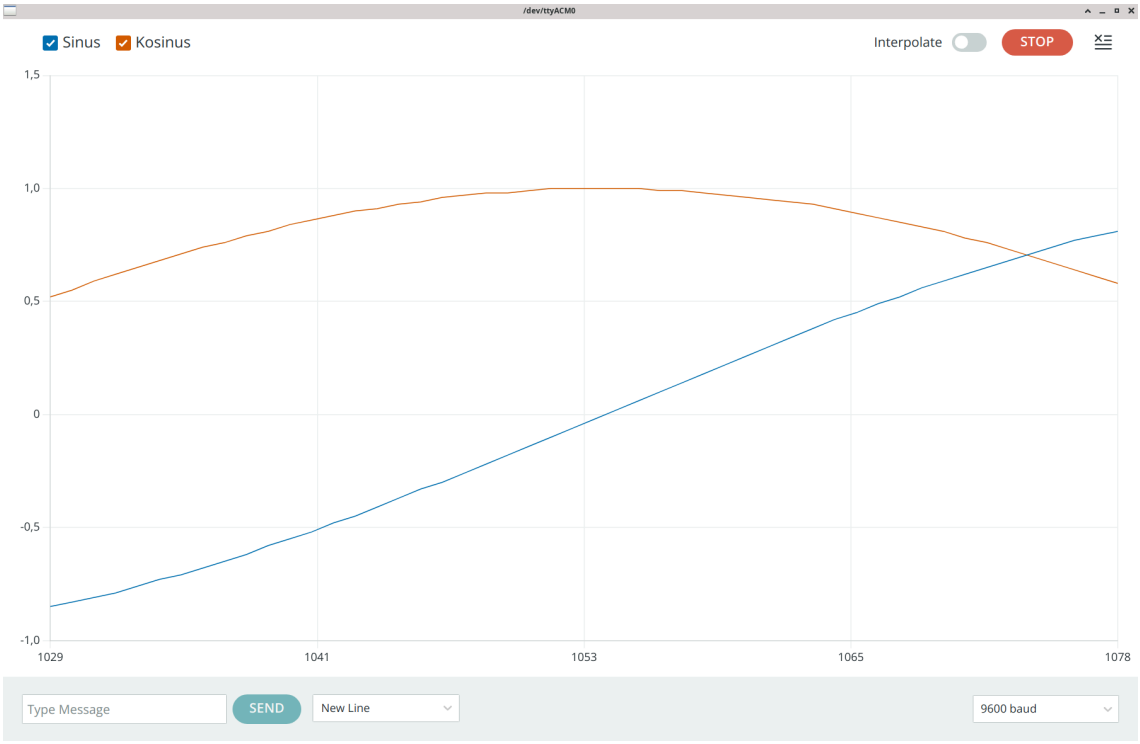
k04_plotter.ino

```
1 void setup() {
2     Serial.begin(115200);
3 }
4
5 void loop() {
6     Serial.printf("Sinus:%.2f,Kosinus:%.2f\n", sin(millis() / 500.0),
        cos(millis() / 500.0));
7     delay(20);
8 }
```

Serieller Monitor

```
Sinus:0.00,Kosinus:1.00
Sinus:0.04,Kosinus:1.00
Sinus:0.08,Kosinus:1.00
Sinus:0.12,Kosinus:0.99
Sinus:0.16,Kosinus:0.99
Sinus:0.20,Kosinus:0.98
Sinus:0.24,Kosinus:0.97
Sinus:0.28,Kosinus:0.96
Sinus:0.31,Kosinus:0.95
Sinus:0.35,Kosinus:0.94
```

`millis()` ist die Zeit seit dem Start in Millisekunden (Kapitel 7), `sin` und `cos` sind Sinus und Kosinus aus dem Mathematikunterricht. Du musst sie hier nicht verstehen; sie liefern nur zwei schöne Wellen zwischen −1 und 1. Die Werte stehen alle 20 Millisekunden in einer Zeile. Im Plotter erscheinen zwei Kurven mit den Namen „Sinus“ und „Kosinus“, die nach links wandern. Er zeigt immer die letzten 50 Punkte, das ist weniger als eine Welle. Seine Oberfläche bleibt englisch („Type Message“, „Send“, „New Line“).



Der serielle Plotter zeigt die Kurven „Sinus“ und „Kosinus“, die der PicoBoy schickt.

Spannend wird der Plotter mit echten Messwerten, etwa vom Neigungssensor in Kapitel 17. Monitor und Plotter teilen sich die Verbindung; was der Sketch schickt, kannst du also wahlweise als Text oder als Kurve ansehen.

Ausprobieren: Das Begrüßungsprogramm

k04_begruessung.ino

```
10 void setup() {
11     Serial.begin(115200);
12     tft.init(240, 280);
13     tft.fillScreen(ST77XX_BLACK);
14     pinMode(TFT_LICHT, OUTPUT);
15     digitalWrite(TFT_LICHT, HIGH);
16     tft.setTextSize(2);
17     tft.setCursor(30, 120);
18     tft.print("Bitte Monitor");
19     tft.setCursor(30, 140);
20     tft.print("oeffnen ...");
21     while (!Serial) {
22         delay(10);
23     }
24
25     Serial.println("Wie heisst du?");
26     while (Serial.available() == 0) {
27         delay(10);
28     }
29     String name = Serial.readStringUntil('\n');
30     name.trim();
```

```
32      Serial.println("In welchem Jahr bist du geboren?");
33      while (Serial.available() == 0) {
34          delay(10);
35      }
36      int jahr = Serial.parseInt();
37      Serial.readStringUntil('\n');
38
39      Serial.print("Hallo ");
40      Serial.print(name);
41      Serial.printf("! Du wirst 2026 %d Jahre alt.\n", 2026 - jahr);
42
43      tft.fillScreen(ST77XX_BLACK);
44      tft.setTextSize(3);
45      tft.setTextColor(ST77XX_YELLOW);
46      tft.setCursor(30, 110);
47      tft.print("Hallo");
48      tft.setCursor(30, 145);
49      tft.print(name);
50  }
51
52  void loop() {
53  }
```

Serieller Monitor

```
Wie heisst du?
In welchem Jahr bist du geboren?
Hallo Mia! Du wirst 2026 15 Jahre alt.
```

Eingegeben wurden `Mia` und `2011`.



- Solange kein Monitor verbunden ist, bittet das Display darum (Zeilen 16 bis 23). So weiß man auch ohne Computer, warum der PicoBoy wartet.
- Der Sketch fragt nach Name und Geburtsjahr. `2026 - jahr` in Zeile 41 rechnet das Alter aus.
- Zum Schluss erscheint der Name auch auf dem Display. `tft.print` 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 vier Zeilen aus, und zwar mit einem einzigen `Serial.println`.

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 ⭐⭐ Zwei Fehler, keine Meldung Dieser Sketch lässt sich ohne Fehler und ohne Warnung kompilieren. Auf dem PicoBoy gibt er aber weder die Temperatur noch den Namen richtig aus. Finde beide Fehler und verbessere sie.

k04_aufgabe4.ino

```
void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    String name = "Mia";
    Serial.printf("Temperatur: %f Grad\n", 21);
    Serial.printf("Name: %s\n", name);
}

void loop() {
}
```

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 der Sketch aus, wenn die zweite Zahl 0 ist? Und was, wenn du `7,5` mit Komma eingibst?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `Serial.begin(115200);` als erste Zeile in `setup()`. Der **serielle Monitor** (Strg + Umschalt + M) zeigt, was der PicoBoy schickt.
- Ausgaben, bevor der Monitor zuhört, gehen verloren. `while (!Serial)` wartet auf ihn, notfalls für immer.
- `Serial.print` ohne, `Serial.println` mit Zeilenumbruch; `\n` und `\t` im Text.
- `Serial.printf`: `%d` ganze Zahl, `%f` Kommazahl, `%.2f` mit 2 Stellen, `%s` Text, `%%` Prozentzeichen. Es gibt **keine** Warnung bei falschen Platzhaltern.
- Eingaben: Zeilenende **Neue Zeile**; warten mit `Serial.available()`, dann `readStringUntil('\n')` und `trim()`, oder `parseInt()` / `parseFloat()` und den Rest der Zeile wegwerfen.
- Lesen wartet höchstens 1 Sekunde (**Timeout**); Zahlen sind dann 0.
- Der **serielle Plotter** zeichnet Zahlen einer Zeile als Kurven, `Name:Wert,Name:Wert`.

KAPITEL 5

Variablen und Datentypen

Worum geht's? Ein Spiel muss sich vieles merken: Punkte, Leben, die Position der Spielfigur, die verstrichene Zeit. Dafür gibt es Variablen. Du lernst, welche Arten von Zahlen der PicoBoy kennt, wie er mit ihnen rechnet und wo er sich verrechnet, wenn man nicht aufpasst. Zum Schluss würfelt der PicoBoy, und du erfährst, warum er ohne einen Trick bei jedem Start dasselbe würfelt.

5.1 Variablen und Datentypen

Eine **Variable** ist ein Platz im Arbeitsspeicher mit einem Namen. In Kapitel 4 hast du schon welche benutzt: `String name = …` für einen Text und `int zahl = …` für eine Zahl. Jetzt genauer:

k05_variablen.ino

```
1  // Eine Variable anlegen, lesen und veraendern
2  void setup() {
3      Serial.begin(115200);
4      while (!Serial) {
5          delay(10);
6      }
7      int punkte = 0;
8      Serial.println(punkte);
9      punkte = 10;
10     Serial.println(punkte);
11     punkte = punkte + 5;
12     Serial.println(punkte);
13     Serial.printf("Du hast %d Punkte.\n", punkte);
14 }
15
16 void loop() {
17 }
```

Serieller Monitor

```
0
10
15
Du hast 15 Punkte.
```

- **Zeile 7** legt die Variable an. Das nennt man **deklarieren**: Zuerst kommt der **Datentyp** (`int`, eine ganze Zahl), dann der Name (`punkte`), dann mit `=` der Anfangswert (`0`).
- **Zeile 9** schreibt einen neuen Wert hinein. Das heißt **Zuweisung**. Der alte Wert ist danach weg. Den Typ schreibst du hier nicht mehr hin: Die Variable gibt es ja schon.
- **Zeile 11** sieht für Mathematiker seltsam aus: `punkte = punkte + 5`. Das ist keine Gleichung, sondern ein Befehl: Rechne den Wert von `punkte` plus 5 aus und schreib das Ergebnis zurück in `punkte`. Das `=` bedeutet in C++ immer „bekommt den Wert".

- Überall, wo eine Zahl stehen darf, darf auch eine Variable stehen, etwa in `Serial.printf` in Zeile 13.

Namen dürfen Buchstaben, Ziffern und `_` enthalten, aber nicht mit einer Ziffer beginnen. Umlaute, ß und Leerzeichen sind verboten. Groß- und Kleinschreibung zählt: `punkte` und `Punkte` sind zwei verschiedene Variablen. In diesem Buch sind Namen von Variablen deutsch, klein und mit `_` zwischen den Wörtern: `punkte`, `leben_start`, `letzte_zeit`.

Datentypen: Der Typ legt fest, was in eine Variable hineinpasst und wie viel Speicher sie belegt. Die wichtigsten:

k05_datentypen.ino

```
 7     int anzahl = -42;
 8     float preis = 2.49;
 9     bool fertig = true;
10     char zeichen = 'A';
11     byte helligkeit = 200;
12     unsigned long dauer = 3000000000;
13
14     Serial.print("int:           ");
15     Serial.println(anzahl);
16     Serial.print("float:         ");
17     Serial.println(preis);
18     Serial.print("bool:          ");
19     Serial.println(fertig);
20     Serial.print("char:          ");
21     Serial.println(zeichen);
22     Serial.print("byte:          ");
23     Serial.println(helligkeit);
24     Serial.print("unsigned long: ");
25     Serial.println(dauer);
```

Serieller Monitor

```
int:           -42
float:         2.49
bool:          1
char:          A
byte:          200
unsigned long: 3000000000
```

Typ	speichert	Bereich auf dem RP2350
`int`	ganze Zahl	−2 147 483 648 bis 2 147 483 647
`unsigned int`	ganze Zahl ohne Vorzeichen	0 bis 4 294 967 295
`long`	ganze Zahl	wie `int`
`unsigned long`	ganze Zahl ohne Vorzeichen	0 bis 4 294 967 295
`byte`	ganze Zahl ohne Vorzeichen	0 bis 255
`float`	Kommazahl	etwa 7 gültige Ziffern

Typ	speichert	Bereich auf dem RP2350
`double`	Kommazahl	etwa 15 gültige Ziffern
`bool`	wahr oder falsch	`true` oder `false`
`char`	ein Zeichen	ein Buchstabe, eine Ziffer, ein Satzzeichen

- **`int`** (*integer*, ganze Zahl) ist der Normalfall für Punkte, Zähler, Koordinaten. **`unsigned`** heißt „ohne Vorzeichen“: Solche Variablen können nicht negativ werden, reichen dafür doppelt so weit nach oben.
- **`float`** (*floating point*, Gleitkomma) speichert Kommazahlen. Im Code schreibst du sie mit **Punkt**: `2.49`. `println` zeigt zwei Nachkommastellen (Kapitel 4.1).
- **`bool`** (nach dem Mathematiker George Boole) kennt nur `true` (wahr) und `false` (falsch). `Serial.println` gibt sie als `1` und `0` aus. Gebraucht werden sie in Kapitel 8.
- **`char`** (*character*, Zeichen) speichert genau ein Zeichen in **einfachen** Anführungszeichen: `'A'`. Ein Text in doppelten Anführungszeichen `"A"` ist etwas anderes (Kapitel 12).
- **`byte`** ist eine kleine Zahl von 0 bis 255. Sie belegt nur ein Byte Speicher; das reicht etwa für eine Helligkeit.
- **`unsigned long`** braucht man für Zeiten: `millis()` liefert die Millisekunden seit dem Start als `unsigned long` (Kapitel 7). Die 3 Milliarden aus Zeile 12 passen hinein, in ein `int` nicht.

Wie groß ist welcher Typ? Das hängt vom Mikrocontroller ab. `sizeof` sagt, wie viele **Bytes** (je 8 Bit) ein Typ belegt. Lade diesen Sketch hoch und öffne den seriellen Monitor:

k05_groessen.ino

```
 7     Serial.printf("bool:          %d\n", sizeof(bool));
 8     Serial.printf("char:          %d\n", sizeof(char));
 9     Serial.printf("byte:          %d\n", sizeof(byte));
10     Serial.printf("int:           %d\n", sizeof(int));
11     Serial.printf("long:          %d\n", sizeof(long));
12     Serial.printf("unsigned long: %d\n", sizeof(unsigned long));
13     Serial.printf("float:         %d\n", sizeof(float));
14     Serial.printf("double:        %d\n", sizeof(double));
```

Auf dem PicoBoy ergibt das:

Typ	`bool`	`char`	`byte`	`int`	`long`	`unsigned long`	`float`	`double`
Bytes	1	1	1	4	4	4	4	8

Ein `int` hat also 4 Bytes = 32 Bit, und `long` ist **genauso groß** wie `int`. Auf dem PC wäre das anders: Dort hat ein `long` meist 8 Bytes.

> In vielen älteren Arduino-Anleitungen steht `long` für alles, was größer als 32 767 werden kann. Das stammt vom Arduino Uno: Dessen kleiner Chip hat ein `int` mit nur 16 Bit (−32 768 bis 32 767). Auf dem RP2350 kannst du `int` und `long` gleichwertig benutzen. Für Zeiten nimm trotzdem `unsigned long`, wie es `millis()` vorgibt.

`float` oder `double`? Beide speichern Kommazahlen, aber verschieden genau:

k05_genauigkeit.ino

```
    float f = 1234.56789;
    double d = 1234.56789;
    Serial.println(f, 5);
    Serial.println(d, 5);

    float gross = 123456789;
    Serial.println(gross, 0);
```

Serieller Monitor

```
1234.56787
1234.56789
123456792
```

Mit 5 Nachkommastellen (zweites Argument von `println`) zeigt sich, dass `float` nur etwa 7 Ziffern genau speichert; die letzte Ziffer stimmt schon nicht mehr. Bei großen Zahlen fehlen deshalb sogar die Einer. `double` ist doppelt so groß und rechnet mit etwa 15 Ziffern. Für Koordinaten, Geschwindigkeiten und Messwerte im Spiel reicht `float` völlig. Der RP2350 hat ein Rechenwerk für `float`, mit `double` rechnet er langsamer.

Wo eine Variable steht, zählt

Eine Variable gibt es nur in dem Block, in dem sie angelegt wurde, zwischen ihren geschweiften Klammern. Für `loop()` hat das eine wichtige Folge:

k05_zaehler_falsch.ino

```
// Die Variable steht in loop() und wird jedes Mal neu angelegt
void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
}

void loop() {
    int zaehler = 0;
    zaehler = zaehler + 1;
    Serial.println(zaehler);
    delay(1000);
}
```

Serieller Monitor

```
1
1
1
1
```

Der Zähler kommt nie über 1 hinaus. Jeder Durchlauf von `loop()` legt `zaehler` in Zeile 10 neu an, mit dem Wert 0. Am Ende des Durchlaufs wird die Variable wieder weggeworfen. Steht die Variable dagegen **außerhalb** aller Funktionen, gibt es sie, solange der Sketch läuft:

k05_zaehler.ino

```
 1 // Die Variable steht ausserhalb von loop() und behaelt ihren Wert
 2 int zaehler = 0;
 3
 4 void setup() {
 5     Serial.begin(115200);
 6     while (!Serial) {
 7         delay(10);
 8     }
 9 }
10
11 void loop() {
12     zaehler = zaehler + 1;
13     Serial.println(zaehler);
14     delay(1000);
15 }
```

Serieller Monitor

```
1
2
3
4
```

Eine solche Variable heißt **global**; `setup()`, `loop()` und alle anderen Funktionen können sie benutzen. Variablen innerhalb einer Funktion heißen **lokal**.

> Was sich über mehrere Durchläufe von `loop()` merken soll (Punkte, Leben, Positionen), muss **außerhalb** von `loop()` stehen.

5.2 Rechnen, Überlauf, ganzzahlige Division

Der PicoBoy rechnet mit `+`, `-`, `*` (mal) und `/` (geteilt). Dazu kommt `%`, der **Rest** einer Division:

k05_rechnen.ino

```
 7     Serial.printf("17 + 5 = %d\n", 17 + 5);
 8     Serial.printf("17 - 5 = %d\n", 17 - 5);
 9     Serial.printf("17 * 5 = %d\n", 17 * 5);
10     Serial.printf("17 / 5 = %d\n", 17 / 5);
11     Serial.printf("17 %% 5 = %d\n", 17 % 5);
12     Serial.printf("17.0 / 5 = %.2f\n", 17.0 / 5);
13     Serial.printf("2 + 3 * 4 = %d\n", 2 + 3 * 4);
14     Serial.printf("(2 + 3) * 4 = %d\n", (2 + 3) * 4);
```

Serieller Monitor

```
17 + 5 = 22
17 - 5 = 12
17 * 5 = 85
17 / 5 = 3
17 % 5 = 2
17.0 / 5 = 3.40
2 + 3 * 4 = 14
(2 + 3) * 4 = 20
```

- **Ganzzahlige Division:** `17 / 5` ergibt **3**, nicht 3,4. Sind beide Zahlen ganze Zahlen, ist auch das Ergebnis eine ganze Zahl. Der Rest nach dem Komma wird abgeschnitten, nicht gerundet: Auch `19 / 5` ist 3.
- **Der Rest:** `17 % 5` ist **2**, denn 17 = 3 × 5 + 2. Das brauchst du öfter, als du denkst: `zahl % 2` ist 0 bei geraden Zahlen, und `sekunden % 60` macht aus Sekunden die Sekundenanzeige einer Uhr. Im `printf` muss das Prozentzeichen verdoppelt werden: `%%` (Kapitel 4.2).
- **Mit Komma:** Ist mindestens eine der beiden Zahlen eine Kommazahl, wie `17.0`, rechnet der PicoBoy mit Kommazahlen: 3.40.
- **Punkt vor Strich** gilt wie in der Mathematik. Klammern ändern die Reihenfolge.

Die Falle mit dem Mittelwert: Hier soll der Mittelwert von 2 und 3 herauskommen, also 2,5:

k05_mittelwert.ino

```
 7     int a = 2;
 8     int b = 3;
 9
10     float mittel = (a + b) / 2;
11     Serial.println(mittel);
12
13     mittel = (a + b) / 2.0;
14     Serial.println(mittel);
```

Serieller Monitor

```
2.00
2.50
```

Beim ersten Mal kommt 2.00 heraus, obwohl `mittel` ein `float` ist. Der PicoBoy rechnet nämlich zuerst die rechte Seite aus, und dort stehen nur ganze Zahlen: (2 + 3) / 2 = 5 / 2 = 2, ganzzahlig. Erst dieses Ergebnis wird in `mittel` geschrieben, dann als 2.00. In Zeile 13 steht `2.0`, eine Kommazahl. Jetzt wird mit Komma gerechnet, und es kommt 2.50 heraus.

> Ganze Zahl durch ganze Zahl ergibt eine ganze Zahl; der Rest fällt weg. Willst du Nachkommastellen, schreib eine der Zahlen mit Komma: `/ 2.0`.

Kurzformen: Variablen ändert man so oft, dass es Abkürzungen dafür gibt:

k05_kurzformen.ino

```
 7     int punkte = 0;
 8     punkte += 10;       // wie punkte = punkte + 10
 9     Serial.println(punkte);
10     punkte *= 3;        // wie punkte = punkte * 3
11     Serial.println(punkte);
12     punkte -= 5;        // wie punkte = punkte - 5
13     Serial.println(punkte);
14     punkte++;           // eins dazu
15     Serial.println(punkte);
16     punkte--;           // eins weniger
17     Serial.println(punkte);
```

Serieller Monitor

```
10
30
25
26
25
```

`punkte += 10` ist dasselbe wie `punkte = punkte + 10`, entsprechend `-=`, `*=` und `/=`. `punkte++` erhöht um genau 1, `punkte--` verringert um 1. Das `++` hat übrigens der Sprache C++ ihren Namen gegeben: „C, um eins erweitert“.

Der Überlauf

Jeder Typ hat einen größten und einen kleinsten Wert. Was passiert, wenn eine Rechnung darüber hinausgeht?

k05_ueberlauf.ino

```
 7     byte b = 255;
 8     b = b + 1;
 9     Serial.printf("byte: 255 + 1 = %d\n", b);
10
11     int i = 2147483647;
12     i = i + 1;
13     Serial.printf("int: 2147483647 + 1 = %d\n", i);
```

```
15      unsigned int u = 0;
16      u = u - 1;
17      Serial.printf("unsigned int: 0 - 1 = %u\n", u);
```

Serieller Monitor

```
byte: 255 + 1 = 0
int: 2147483647 + 1 = -2147483648
unsigned int: 0 - 1 = 4294967295
```

Die Zahl **läuft über** und kommt am anderen Ende wieder heraus, wie bei einem Kilometerzähler, der nach 999 999 wieder bei 000 000 anfängt:

- Ein `byte` geht nur bis 255. 255 + 1 ergibt 0.
- Ein `int` geht bis 2 147 483 647. Eins mehr ergibt die **kleinste** negative Zahl.
- Ein `unsigned int` kennt keine negativen Zahlen. 0 − 1 ergibt die größte Zahl. Der Platzhalter `%u` gibt Zahlen ohne Vorzeichen aus; mit `%d` stünde hier −1.

Eine Fehlermeldung gibt es dabei nicht. Der Sketch rechnet einfach mit dem falschen Wert weiter. Ein Punktestand, der plötzlich negativ wird, ist ein typisches Zeichen dafür.

> Beim Überlauf eines `int` (mit Vorzeichen) legt C++ genau genommen gar nicht fest, was passiert. Der RP2350 verhält sich wie im Beispiel, aber verlassen sollte man sich darauf nicht. Bei Typen ohne Vorzeichen ist das Überlaufen dagegen fest geregelt; davon lebt der `millis`-Takt in Kapitel 7.4.

Manchmal bemerkt der Compiler einen Überlauf, nämlich wenn er die Rechnung schon beim Kompilieren ausrechnen kann. Wie viele Millisekunden haben 30 Tage?

k05_monat_falsch.ino

```
7       int ms_pro_monat = 30 * 24 * 60 * 60 * 1000;
8       Serial.println(ms_pro_monat);
```

Ausgabe

```
k05_monat_falsch.ino: In function 'void setup()':
k05_monat_falsch.ino:7:42: warning: integer overflow in expression of type 'int'
  results in '-1702967296' [-Woverflow]
    7 |     int ms_pro_monat = 30 * 24 * 60 * 60 * 1000;
      |                        ~~~~~~~~~~~~~~~~~~^~~~~~
```

Serieller Monitor

```
-1702967296
```

Das Ergebnis, 2 592 000 000, passt nicht in ein `int`. Die Warnung sagt genau das: *integer overflow in expression of type ‚int'*, ganzzahliger Überlauf in einem Ausdruck vom Typ `int`, mit dem Ergebnis −1 702 967 296. Ein größerer Typ für die Variable allein hilft nicht, denn gerechnet wird mit dem Typ der **Zahlen**, und `30`, `24` und so weiter sind `int`. Die Abhilfe: Die erste Zahl bekommt die Endung `UL` (*unsigned long*). Dann wird die ganze Rechnung mit `unsigned long` ausgeführt:

k05_monat_richtig.ino

```
7     unsigned long ms_pro_monat = 30UL * 24 * 60 * 60 * 1000;
8     Serial.println(ms_pro_monat);
```

Serieller Monitor

```
2592000000
```

Division durch 0

Durch 0 darf man nicht teilen, das weißt du aus der Mathematik. Was macht der PicoBoy, wenn es doch passiert? Dieser Sketch fragt nach dem Teiler:

k05_division_null.ino

```
7     Serial.println("Durch welche Zahl soll ich 100 teilen?");
8     while (Serial.available() == 0) {
9         delay(10);
10    }
11    int teiler = Serial.parseInt();
12    Serial.readStringUntil('\n');
13
14    Serial.printf("100 / %d = %d\n", teiler, 100 / teiler);
15    Serial.printf("100 %% %d = %d\n", teiler, 100 % teiler);
16    Serial.printf("100.0 / %d = %.2f\n", teiler, 100.0 / teiler);
```

Gibst du `0` ein, stürzt der PicoBoy **nicht** ab. Er rechnet einfach weiter, und zwar so:

- `100 / 0` ergibt bei ganzen Zahlen **0**.
- `100 % 0` ergibt **100**.
- `100.0 / 0` ergibt bei Kommazahlen `inf` (*infinity*, unendlich), wie in Aufgabe 5 von Kapitel 4.

Das liegt am Prozessor des RP2350 (einem Cortex-M33). Er hat einen eigenen Befehl für die ganzzahlige Division, der bei 0 als Teiler einfach 0 liefert, solange niemand eine Fehlerprüfung einschaltet. Der Kern für den RP2350 schaltet sie nicht ein. Auf einem PC dagegen bricht ein Programm mit einer ganzzahligen Division durch 0 ab.

Beides ist kein gutes Verhalten für dein Spiel: Eine stille 0 ist ebenso falsch wie ein Absturz. Wenn ein Teiler 0 werden kann, prüfe das vorher (Kapitel 8).

5.3 `const` und `#define`

Manche Werte ändern sich nie: die Breite des Displays, die Zahl der Leben am Anfang, die Punkte für eine Münze. Solchen Werten gibst du einen Namen, statt überall die Zahl hinzuschreiben. Dafür gibt es zwei Wege:

k05_konstanten.ino

```
// Feste Werte bekommen einen Namen
#define LEBEN_START 3
const int PUNKTE_PRO_MUENZE = 10;
const float BONUS = 1.5;

void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    int leben = LEBEN_START;
    int punkte = 0;

    punkte += PUNKTE_PRO_MUENZE;        // eine Muenze
    punkte += PUNKTE_PRO_MUENZE;        // noch eine
    leben--;                            // einmal erwischt
    float gesamt = punkte * BONUS;

    Serial.printf("Leben: %d, Punkte: %d, mit Bonus: %.1f\n", leben, punkte,
      gesamt);
}

void loop() {
}
```

Serieller Monitor

```
Leben: 2, Punkte: 20, mit Bonus: 30.0
```

- `#define` kennst du von den Pins (Kapitel 3.2). Vor dem Übersetzen ersetzt der Compiler jedes `LEBEN_START` durch `3`, wie mit Suchen und Ersetzen. Kein `=`, kein Semikolon.
- `const` (*constant*, unveränderlich) vor einer Variablen macht sie zu einer **Konstanten**: Sie bekommt einmal einen Wert, und der bleibt. Anders als bei `#define` hat sie einen Typ, hier `int` und `float`.
- Namen von Konstanten schreibt man in Großbuchstaben. So sieht man im Sketch sofort, was sich nie ändert.

Beides hat zwei Vorteile: Der Sketch liest sich besser (`PUNKTE_PRO_MUENZE` statt `10`), und soll eine Münze einmal 20 Punkte geben, änderst du nur eine Zeile. Versucht der Sketch, eine Konstante zu ändern, meldet der Compiler einen Fehler:

k05_const_fehler.ino

```
 2 const int PUNKTE_PRO_MUENZE = 10;
 3
 4 void setup() {
 5     Serial.begin(115200);
 6     PUNKTE_PRO_MUENZE = 20;
 7     Serial.println(PUNKTE_PRO_MUENZE);
```

Ausgabe

```
k05_const_fehler.ino: In function 'void setup()':
k05_const_fehler.ino:6:23: error: assignment of read-only variable
  'PUNKTE_PRO_MUENZE'
    6 |     PUNKTE_PRO_MUENZE = 20;
      |     ~~~~~~~~~~~~~~~~~~^~~~
Fehler beim Build: exit status 1
```

Assignment of read-only variable: Zuweisung an eine Variable, die nur gelesen werden darf.

> `const` oder `#define`? Für Pins ist `#define` in Arduino-Sketches üblich, so steht es auch in diesem Buch. Für alle anderen festen Werte ist `const` die bessere Wahl: Der Compiler kennt den Typ und kann Fehler genauer melden. Bei `#define` darf hinten **kein Semikolon** stehen; `#define LEBEN_START 3;` würde das `;` überall mit einsetzen.

5.4 Zufall mit `random()`

Ohne Zufall gäbe es kaum Spiele. `random` liefert eine zufällige ganze Zahl:

Aufruf	liefert
`random(10)`	0 bis 9
`random(1, 7)`	1 bis 6
`random(-5, 6)`	−5 bis 5

Die obere Grenze ist **nicht** dabei. Für einen Würfel schreibst du deshalb `random(1, 7)`.

k05_zufall.ino

```
 1 // Jede halbe Sekunde ein Wurf mit einem Wuerfel
 2 void setup() {
 3     Serial.begin(115200);
```

```
    while (!Serial) {
        delay(10);
    }
}

void loop() {
    int wurf = random(1, 7);        // 1 bis 6
    Serial.println(wurf);
    delay(500);
}
```

Serieller Monitor

```
6
4
3
6
1
1
5
1
3
2
4
```

Das sieht zufällig aus. Schreib dir die ersten Zahlen auf, drücke am PicoBoy **RES** und sieh im Monitor nach, sobald er sich wieder verbunden hat: Es kommen **genau dieselben Zahlen** in derselben Reihenfolge. Bei jedem Start. Welche Zahlen es sind, kann bei dir anders aussehen als hier im Buch; entscheidend ist, dass sie sich wiederholen.

Warum? Ein Computer kann nicht von sich aus würfeln. `random` berechnet jede Zahl mit einer festen Formel aus der vorigen. Das Ergebnis sieht wild aus, ist aber vollständig vorherbestimmt; man nennt es **Pseudozufall**. Die Formel braucht einen Startwert, den **Seed** (englisch: Saatkorn). Der Kern benutzt dafür die Funktion `rand()` der Sprache C, und deren Startwert ist bei jedem Start derselbe. Gleicher Startwert, gleiche Folge.

Echter Zufall: Mit `randomSeed(startwert)` setzt du den Startwert selbst. Der muss bei jedem Start anders sein. Der RP2350 hat dafür einen eingebauten **Hardware-Zufallsgenerator**, der aus elektrischem Rauschen im Chip echte Zufallszahlen gewinnt. `rp2040.hwrand32()` liefert eine davon:

k05_zufall_echt.ino

```
void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    randomSeed(rp2040.hwrand32());
}
```

Jetzt kommt nach jedem RES eine andere Folge. `randomSeed` gehört einmal in `setup()`, nicht in `loop()`. Das Objekt heißt `rp2040`, auch beim RP2350; der Kern wurde für den Vorgänger-Chip geschrieben und hat den Namen behalten.

> In vielen Arduino-Anleitungen steht `randomSeed(analogRead(A0));`: Ein unbenutzter Pin liefert ein zufälliges Rauschen. Beim PicoBoy ist `A0` aber der Pin GP26, und an dem hängt das **Hintergrundlicht** des Displays. Er rauscht nicht. Nimm `rp2040.hwrand32()`.

Ausprobieren: Der Würfelbecher

k05_wuerfelbecher.ino

```
 9 const int GROESSE = 6;      // Groesse der Augenzahlen
10
11 Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
12
13 void setup() {
14     tft.init(240, 280);
15     tft.fillScreen(ST77XX_BLACK);
16     pinMode(TFT_LICHT, OUTPUT);
17     digitalWrite(TFT_LICHT, HIGH);
18     // randomSeed(rp2040.hwrand32());  // echter Zufall
19
20     int wuerfel1 = random(1, 7);
21     int wuerfel2 = random(1, 7);
22     int summe = wuerfel1 + wuerfel2;
23
24     tft.setTextSize(2);
25     tft.setTextColor(ST77XX_WHITE);
26     tft.setCursor(60, 40);
27     tft.print("Dein Wurf:");
28
29     tft.setTextSize(GROESSE);
30     tft.setTextColor(ST77XX_YELLOW);
31     tft.setCursor(42, 80);
32     tft.print(wuerfel1);
33     tft.setCursor(162, 80);
34     tft.print(wuerfel2);
35
36     tft.setTextSize(3);
37     tft.setTextColor(ST77XX_CYAN);
38     tft.setCursor(57, 180);
39     tft.printf("Summe %d", summe);
40 }
```



- Die Konstante `GROESSE` (Zeile 9) legt fest, wie groß die Augenzahlen erscheinen.
- Zwei Würfe, eine Summe: drei lokale Variablen in `setup()`.
- `tft.print` gibt auch Zahlen aus, und `tft.printf` in Zeile 39 funktioniert wie `Serial.printf`, nur auf dem Display.
- Gewürfelt wird nur einmal beim Start. Für einen neuen Wurf drückst du **RES**.

Drücke RES ein paar Mal: Es ist immer derselbe Wurf. Entferne dann in Zeile 18 die beiden Schrägstriche vor `randomSeed` und lade den Sketch neu hoch. Jetzt würfelt der PicoBoy wirklich.

Zum Verändern: Würfle mit drei Würfeln und zeige alle drei und ihre Summe an.

Übungsaufgaben

Aufgabe 1 ⭐ **Welcher Typ passt?** Welchen Datentyp nimmst du für: das Alter einer Person, den Preis eines Brötchens, ob das Spiel vorbei ist, den Anfangsbuchstaben eines Namens, die Zeit seit dem Start in Millisekunden, die Helligkeit einer LED von 0 bis 255, die Einwohnerzahl Deutschlands?

Aufgabe 2 ⭐ **Was kommt heraus?** Überlege für jede Zeile, was der Sketch ausgibt. Prüfe danach auf dem PicoBoy.

k05_aufgabe2.ino

```
7     int a = 7;
8     int b = 2;
9     Serial.println(a / b);
10    Serial.println(a % b);
11    Serial.println(a / 2.0);
12    float c = a / b;
13    Serial.println(c);
14    a += b;
15    Serial.println(a * b);
```

Aufgabe 3 ⭐⭐ **Sekunden umrechnen** Rechne 100 000 Sekunden in Tage, Stunden, Minuten und Sekunden um, nur mit `/` und `%`, und gib das Ergebnis in einem Satz aus.

Aufgabe 4 ⭐⭐ **Zufall in Bereichen** Erzeuge mit `random` und gib aus: einen Münzwurf (0 oder 1), einen Wurf mit einem Würfel mit 20 Seiten, eine x-Position für einen 20 Pixel breiten Ball, der ganz auf das 240 Pixel breite Display passen soll, und eine Temperatur von −5 bis 30 Grad. Achte genau auf die Grenzen.

Aufgabe 5 ⭐⭐⭐ Wann läuft die Uhr über? `millis()` liefert ein `unsigned long`. Berechne mit einem Sketch, nach wie vielen Tagen, Stunden, Minuten und Sekunden es überläuft. Und wann wäre es so weit, wenn jemand die Zeit in einem `int` speichert?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Variable anlegen: `typ name = wert;`, ändern: `name = neuer_wert;`. Was `loop()` sich merken soll, steht **außerhalb** von `loop()`.
- Typen: `int` (32 Bit), `unsigned long` (für `millis()`), `float` (Kommazahl, etwa 7 Ziffern), `bool`, `char`, `byte` (0 bis 255). Auf dem RP2350 ist `long` so groß wie `int`.
- `/` mit ganzen Zahlen schneidet ab, `%` ist der Rest. Kommazahl mit `2.0` erzwingen.
- Zu große Zahlen **laufen über**, ohne Fehlermeldung. `30UL * …` rechnet mit `unsigned long`.
- Ganzzahlige Division durch 0 ergibt auf dem PicoBoy still 0.
- Feste Werte: `const int NAME = 10;` oder `#define NAME 10`.
- `random(1, 7)` würfelt 1 bis 6. Ohne `randomSeed` bei jedem Start dieselbe Folge; echter Zufall mit `randomSeed(rp2040.hwrand32());` in `setup()`.

KAPITEL 6

Zeichnen und Farben

Worum geht's? Bisher hast du nur Text auf das Display geschrieben. Jetzt zeichnest du Linien, Rechtecke, Kreise und Dreiecke, mischst eigene Farben und benutzt schönere Schriften. Am Ende setzt du einen Text in jeder Schrift genau in die Mitte. Alle Befehle dieses Kapitels stammen aus der Bibliothek **Adafruit GFX**, die mit der Display-Bibliothek installiert wurde (Kapitel 2.4). GFX steht für *graphics*, Grafik.

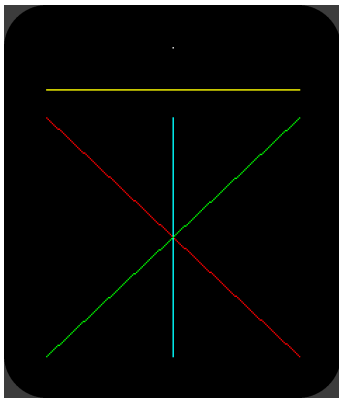
6.1 Linien, Rechtecke, Kreise, Dreiecke

Alle Zeichenbefehle funktionieren gleich: Erst kommen Zahlen für die Position und die Größe, als letztes Argument die Farbe. Die Sketches in diesem Abschnitt beginnen wie in Kapitel 3: Display starten, schwarz füllen, Licht an. Im Buch ist dieser Anfang ab jetzt meist weggelassen; im Sketch-Ordner steht immer der ganze Sketch.

Pixel und Linien:

k06_linien.ino

```
17      tft.drawPixel(120, 30, ST77XX_WHITE);
18      tft.drawLine(30, 60, 210, 60, ST77XX_YELLOW);
19      tft.drawLine(120, 80, 120, 250, ST77XX_CYAN);
20      tft.drawLine(30, 80, 210, 250, ST77XX_RED);
21      tft.drawLine(210, 80, 30, 250, ST77XX_GREEN);
```



- `drawPixel(x, y, farbe)` färbt ein einziges Pixel (Zeile 17). Du siehst es oben in der Mitte als winzigen Punkt. Ein Pixel ist kleiner als ein Viertel Millimeter.
- `drawLine(x0, y0, x1, y1, farbe)` zieht eine gerade Linie von (x0, y0) nach (x1, y1). Beide Endpunkte gehören dazu. Die Linie in Zeile 18 ist waagerecht, weil beide Enden dasselbe y haben; die in Zeile 19 ist senkrecht, weil beide dasselbe x haben.
- Schräge Linien (Zeilen 20 und 21) setzt die Bibliothek aus einzelnen Pixeln zusammen. Aus der Nähe sieht man deshalb kleine Treppenstufen.

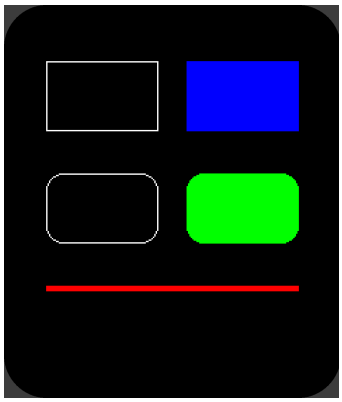
Linien sind immer genau ein Pixel dick. Eine dicke Linie zeichnest du als schmales, gefülltes Rechteck.

Rechtecke gibt es als Rahmen (`drawRect`) und gefüllt (`fillRect`), beide auch mit runden Ecken:

k06_rechtecke.ino

```
17      tft.drawRect(30, 40, 80, 50, ST77XX_WHITE);
18      tft.fillRect(130, 40, 80, 50, ST77XX_BLUE);
19      tft.drawRoundRect(30, 120, 80, 50, 12, ST77XX_WHITE);
20      tft.fillRoundRect(130, 120, 80, 50, 12, ST77XX_GREEN);
21      tft.fillRect(30, 200, 180, 4, ST77XX_RED);
```

Die vier Zahlen sind **x, y, Breite, Höhe**: (x, y) ist die linke obere Ecke, dann folgen Breite (*width*, `w`) und Höhe (*height*, `h`). Das erste Rechteck reicht also von x = 30 bis x = 109 und von y = 40 bis y = 89. Bei `drawRoundRect` und `fillRoundRect` kommt vor der Farbe noch der **Radius** der Ecken dazu, hier 12 Pixel. Zeile 21 zeigt den Trick mit der dicken Linie: ein Rechteck, das 180 Pixel breit und nur 4 Pixel hoch ist.

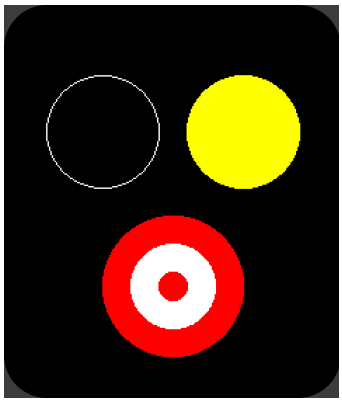


> Die dritte und vierte Zahl sind **Breite und Höhe**, nicht die rechte untere Ecke. `fillRect(130, 40, 80, 50, …)` zeichnet ein Rechteck von 80 × 50 Pixeln, nicht eines bis zum Punkt (80, 50).

Kreise brauchen den **Mittelpunkt** und den **Radius**:

k06_kreise.ino

```
17      tft.drawCircle(70, 90, 40, ST77XX_WHITE);
18      tft.fillCircle(170, 90, 40, ST77XX_YELLOW);
19      tft.fillCircle(120, 200, 50, ST77XX_RED);
20      tft.fillCircle(120, 200, 30, ST77XX_WHITE);
21      tft.fillCircle(120, 200, 10, ST77XX_RED);
```



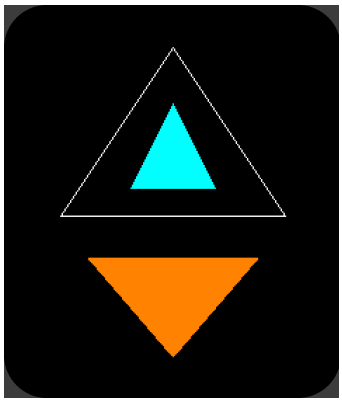
Ein Kreis mit Radius 40 ist 81 Pixel breit: 40 nach links, 40 nach rechts und das Pixel in der Mitte. Die Zielscheibe unten entsteht aus drei gefüllten Kreisen mit demselben Mittelpunkt. Die Reihenfolge ist wichtig: erst der große rote, dann der weiße darauf, zuletzt der kleine rote. Was später gezeichnet wird, liegt **oben** und verdeckt, was darunter war.

Dreiecke bestimmst du durch ihre drei Ecken, also sechs Zahlen:

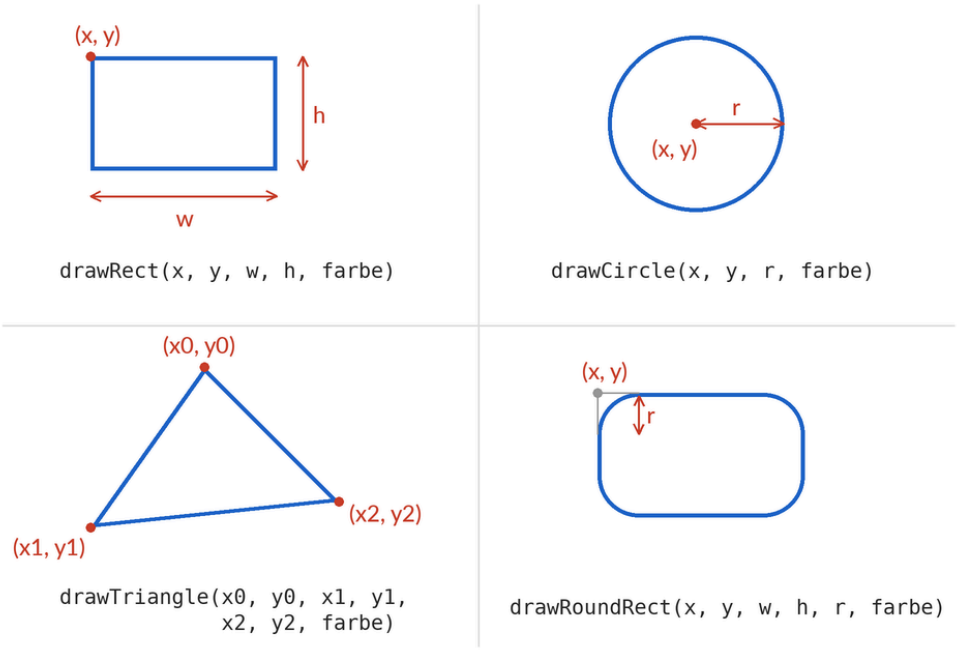
k06_dreiecke.ino

```
17      tft.drawTriangle(120, 30, 40, 150, 200, 150, ST77XX_WHITE);
18      tft.fillTriangle(60, 180, 120, 250, 180, 180, ST77XX_ORANGE);
19      tft.fillTriangle(120, 70, 90, 130, 150, 130, ST77XX_CYAN);
```

Die Reihenfolge der Ecken ist egal. Das orange Dreieck steht auf der Spitze, weil seine dritte Ecke (120, 250) unter den beiden anderen liegt.



Die Übersicht zeigt, was die Zahlen bei jeder Form bedeuten:



Befehl	zeichnet
`drawPixel(x, y, farbe)`	ein Pixel
`drawLine(x0, y0, x1, y1, farbe)`	eine Linie von (x0, y0) nach (x1, y1)
`drawRect(x, y, w, h, farbe)` / `fillRect(…)`	ein Rechteck, Rahmen / gefüllt
`drawRoundRect(x, y, w, h, r, farbe)` / `fillRoundRect(…)`	ein Rechteck mit runden Ecken
`drawCircle(x, y, r, farbe)` / `fillCircle(…)`	einen Kreis um (x, y)
`drawTriangle(x0, y0, x1, y1, x2, y2, farbe)` / `fillTriangle(…)`	ein Dreieck
`fillScreen(farbe)`	den ganzen Bildschirm

Alles, was über den Rand hinausragt, schneidet die Bibliothek einfach ab. Ein Kreis darf also auch halb aus dem Bild ragen.

> Für waagerechte und senkrechte Linien gibt es zusätzlich `drawFastHLine(x, y, länge, farbe)` und `drawFastVLine(x, y, länge, farbe)`. Sie sind schneller als `drawLine`, weil die Bibliothek nicht rechnen muss, wo die Pixel liegen. Wie lange das Zeichnen dauert, untersuchst du in Kapitel 7.

6.2 Farben: `ST77XX_…` und `color565(r, g, b)`

Neun Farben hat die Bibliothek fertig dabei. Du kennst sie aus Kapitel 3:

k06_farbkonstanten.ino

```
17      tft.setTextSize(2);
18      tft.setTextColor(ST77XX_WHITE);
19
20      tft.fillRect(40, 30, 40, 18, ST77XX_BLACK);
21      tft.drawRect(40, 30, 40, 18, ST77XX_WHITE);
22      tft.setCursor(92, 32);
23      tft.print("BLACK");
24      tft.fillRect(40, 55, 40, 18, ST77XX_WHITE);
25      tft.setCursor(92, 57);
26      tft.print("WHITE");
```

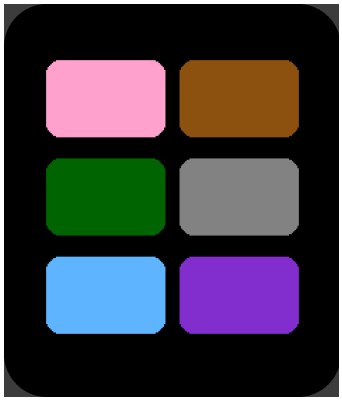


Schwarz auf schwarzem Grund sähe man nicht, deshalb hat das erste Feld einen weißen Rahmen (Zeile 21). Für alle anderen Farben mischst du selbst.

Farben mischen: Ein Pixel des Displays besteht aus drei winzigen Leuchtpunkten in **Rot**, **Grün** und **Blau**. Aus diesen drei Grundfarben setzt sich jede Farbe zusammen: Rot und Grün zusammen ergeben Gelb, alle drei voll ergeben Weiß, keiner ergibt Schwarz. `tft.color565(r, g, b)` mischt eine Farbe aus drei Anteilen von **0** (aus) bis **255** (voll):

k06_mischen.ino

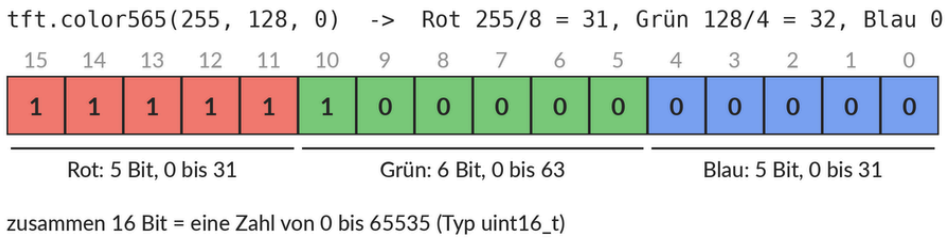
```
17      uint16_t rosa = tft.color565(255, 160, 200);
18      uint16_t braun = tft.color565(140, 80, 20);
19      uint16_t dunkelgruen = tft.color565(0, 100, 0);
20      uint16_t grau = tft.color565(128, 128, 128);
21      uint16_t himmelblau = tft.color565(100, 180, 255);
22      uint16_t lila = tft.color565(130, 50, 200);
23
24      tft.fillRoundRect(30, 40, 85, 55, 10, rosa);
25      tft.fillRoundRect(125, 40, 85, 55, 10, braun);
26      tft.fillRoundRect(30, 110, 85, 55, 10, dunkelgruen);
27      tft.fillRoundRect(125, 110, 85, 55, 10, grau);
28      tft.fillRoundRect(30, 180, 85, 55, 10, himmelblau);
29      tft.fillRoundRect(125, 180, 85, 55, 10, lila);
```



- Die gemischten Farben stehen in **Variablen** (Kapitel 5). Ihr Typ ist `uint16_t`: eine ganze Zahl ohne Vorzeichen (*unsigned*, `u`) mit genau 16 Bit, also von 0 bis 65535. Das `_t` am Ende steht für *type*. Eine Farbe des Displays ist genau so eine Zahl.
- Grau entsteht, wenn alle drei Anteile gleich sind: `(128, 128, 128)`. Je größer die Zahl, desto heller.
- Braun ist ein dunkles Orange: viel Rot, etwas Grün, fast kein Blau.

Die richtigen Zahlen für eine Farbe findest du durch Ausprobieren oder mit einem Farbwähler. Fast jedes Malprogramm zeigt die Anteile als „RGB“ an, ebenso die Suche nach „color picker“ im Internet.

Warum 565? Das Display bekommt für jedes Pixel 16 Bit. Für drei Farben reicht das nicht ganz zu je 8 Bit (0 bis 255). Deshalb bekommt Rot 5 Bit, Grün 6 Bit und Blau 5 Bit; Grün bekommt das zusätzliche Bit, weil unsere Augen Grüntöne am feinsten unterscheiden. `color565` teilt die Anteile dafür ganzzahlig durch 8 (Rot, Blau) beziehungsweise 4 (Grün):



Darum gibt es auf dem Display „nur“ 32 Stufen Rot, 64 Stufen Grün und 32 Stufen Blau, zusammen 65 536 Farben. `color565(255, 160, 200)` und `color565(254, 161, 201)` ergeben dieselbe Farbe. Für die Augen reicht das völlig.

> Die fertigen Farben sind genau solche Zahlen. In der Bibliothek steht zum Beispiel `#define ST77XX_RED 0xF800`. Das `0x` kennzeichnet eine Zahl im **Sechzehnersystem** (hexadezimal), das Programmierer für Bitmuster mögen: `0xF800` sind die 16 Bit `11111 000000 00000`, also Rot voll, Grün und Blau aus. Du musst das Sechzehnersystem nicht beherrschen; `color565` rechnet für dich.

6.3 Schöne Schriften: die FreeFonts

Die eingebaute Schrift aus Kapitel 3 ist klein und grob; vergrößert sieht man die Pixel-Treppen. Adafruit GFX bringt deshalb einen Satz besserer Schriften mit, die **FreeFonts**. Jede Schrift steckt in einer eigenen Datei, die du mit `#include` holst:

```
#include <Fonts/FreeSans18pt7b.h>
```

Danach schaltest du mit `tft.setFont(&FreeSans18pt7b);` auf diese Schrift um. Das `&` vor dem Namen gehört dazu; was es bedeutet, erfährst du in Kapitel 19. `tft.setFont();` ohne Argument schaltet zurück zur eingebauten Schrift.

Die wichtigste Regel: Der Cursor steht auf der Grundlinie. Bei der eingebauten Schrift legt `setCursor(x, y)` die **linke obere Ecke** des Texts fest (Kapitel 3). Bei einer FreeFont legt y die **Grundlinie** fest: die gedachte Linie, auf der die Buchstaben stehen, wie die Linie im Schulheft. Buchstaben wie g und y hängen darunter.

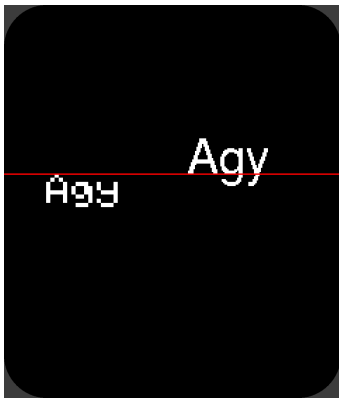
k06_grundlinie.ino

```
1 // Die Standardschrift haengt unter dem Cursor, eine FreeFont steht darauf
2 #include <Adafruit_ST7789.h>
3 #include <Fonts/FreeSans18pt7b.h>
```

k06_grundlinie.ino

```
18      tft.setTextColor(ST77XX_WHITE);
19
20      // Standardschrift: y ist die Oberkante
21      tft.setTextSize(3);
22      tft.setCursor(30, 120);
23      tft.print("Agy");
24
25      // FreeFont: y ist die Grundlinie
26      tft.setFont(&FreeSans18pt7b);
27      tft.setTextSize(1);
28      tft.setCursor(130, 120);
29      tft.print("Agy");
30
31      tft.drawLine(0, 120, 239, 120, ST77XX_RED);     // Hilfslinie bei y = 120
```

Beide Texte haben dasselbe y = 120; die rote Linie in Zeile 31 markiert es. Der linke Text in der eingebauten Schrift **hängt** unter der Linie, der rechte in FreeSans **steht** darauf.

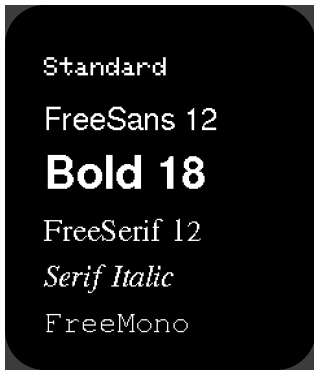


> Setzt du eine FreeFont mit einem y wie bei der eingebauten Schrift ein, etwa `setCursor(30, 0)`, steht der Text **über** dem oberen Rand und ist unsichtbar. Für eine FreeFont muss y mindestens so groß sein wie die Höhe der Großbuchstaben.

Welche Schriften gibt es? Drei Familien, je in vier Größen und vier Schnitten:

k06_schriften.ino

```
1 // Einige FreeFonts von Adafruit GFX
2 #include <Adafruit_ST7789.h>
3 #include <Fonts/FreeSans12pt7b.h>
4 #include <Fonts/FreeSansBold18pt7b.h>
5 #include <Fonts/FreeSerif12pt7b.h>
6 #include <Fonts/FreeSerifItalic12pt7b.h>
7 #include <Fonts/FreeMono12pt7b.h>
```



Familie	Aussehen
`FreeSans`	schlicht, ohne Füßchen (wie diese Schrift)
`FreeSerif`	mit Füßchen (Serifen), wie in Zeitungen
`FreeMono`	alle Zeichen gleich breit, wie auf der Schreibmaschine

Dazu kommen die Schnitte `Bold` (fett), `Oblique` bei Sans und Mono beziehungsweise `Italic` bei Serif (schräg) und beides zusammen, etwa `FreeSansBoldOblique`. Die Größe steht im Namen: `9pt`, `12pt`, `18pt` oder `24pt` (*pt* ist eine Maßeinheit für Schriftgrößen). Das `7b` am Ende bedeutet 7 Bit: Die Schriften enthalten nur die Zeichen der englischen Tastatur. Umlaute fehlen ganz, sie werden einfach weggelassen. Die Dateien liegen im Ordner `Fonts` der Bibliothek Adafruit GFX.

`setTextSize` funktioniert auch mit FreeFonts, vergrößert aber wieder Pixel für Pixel. Nimm lieber gleich die größere Schrift und lass die Größe auf 1.

Kein Hintergrund: `setTextColor(farbe, hintergrund)` aus Kapitel 3 füllt nur bei der eingebauten Schrift den Platz hinter den Zeichen. Bei FreeFonts wird der Hintergrund absichtlich nie gezeichnet. Soll ein Text in einer FreeFont einen anderen ersetzen, übermalst du den alten erst mit einem gefüllten Rechteck. Wie groß das sein muss, sagt dir `getTextBounds`.

Text zentrieren mit `getTextBounds`

Die Zeichen einer FreeFont sind verschieden breit: Ein „i“ ist schmaler als ein „W“. Die Breite eines Texts lässt sich deshalb nicht mehr wie in Kapitel 3 einfach ausrechnen. Das übernimmt `getTextBounds`. Es berechnet den kleinsten Kasten, der den Text umschließt, ohne etwas zu zeichnen:

k06_textkasten.ino

```
18      tft.setFont(&FreeSans18pt7b);
19      tft.setTextColor(ST77XX_WHITE);
20
21      int16_t x1, y1;
22      uint16_t w, h;
23      tft.getTextBounds("Hallo Welt", 30, 140, &x1, &y1, &w, &h);
24
25      tft.drawRect(x1, y1, w, h, ST77XX_RED);
26      tft.drawLine(0, 140, 239, 140, ST77XX_BLUE);
27      tft.setCursor(30, 140);
28      tft.print("Hallo Welt");
29
30      tft.setFont();
31      tft.setTextSize(2);
32      tft.setCursor(30, 180);
33      tft.printf("x1=%d y1=%d", x1, y1);
34      tft.setCursor(30, 200);
35      tft.printf("w=%d h=%d", w, h);
```

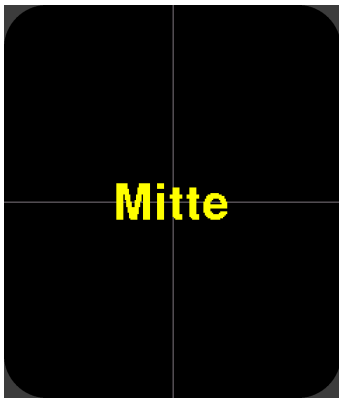


- Zeilen 21 und 22 legen vier Variablen für das Ergebnis an. `int16_t` ist eine ganze Zahl mit 16 Bit und Vorzeichen, `uint16_t` kennst du aus 6.2. Genau diese Typen verlangt `getTextBounds`.
- Zeile 23 fragt: Wo läge der Text „Hallo Welt“, wenn der Cursor bei (30, 140) stünde? Die Antwort schreibt `getTextBounds` in die vier Variablen. Das `&` vor jedem Namen erlaubt der Funktion, in deine Variablen zu schreiben (Kapitel 19).
- `x1` und `y1` sind die linke obere Ecke des Kastens, `w` und `h` seine Breite und Höhe. Zeile 25 zeichnet den Kasten rot ein.
- Der Kasten beginnt bei y1 = 115, also 25 Pixel **über** der Grundlinie (blau, y = 140), so hoch ist das H. Die Unterkante liegt hier genau auf der Grundlinie, weil „Hallo Welt“ nichts Hängendes wie g oder y hat.
- Auch x1 = 33 ist nicht genau 30: Das H beginnt ein paar Pixel rechts vom Cursor.
- Zeile 30 schaltet mit `setFont()` zurück zur eingebauten Schrift. `tft.printf` in Zeile 33 funktioniert genauso wie `Serial.printf` aus Kapitel 4, nur schreibt es auf das Display.

Mit dem Kasten lässt sich ein Text genau in die Mitte setzen. Der Trick: `getTextBounds` mit dem Cursor bei (0, 0) aufrufen. Dann sagen `x1` und `y1`, wie weit der Kasten vom Cursor entfernt liegt.

k06_zentriert.ino

```
18    // Hilfslinien durch die Mitte
19    uint16_t grau = tft.color565(90, 90, 90);
20    tft.drawLine(120, 0, 120, 279, grau);
21    tft.drawLine(0, 140, 239, 140, grau);
22
23    tft.setFont(&FreeSansBold18pt7b);
24    tft.setTextColor(ST77XX_YELLOW);
25
26    int16_t x1, y1;
27    uint16_t w, h;
28    tft.getTextBounds("Mitte", 0, 0, &x1, &y1, &w, &h);
29    int x = (240 - w) / 2 - x1;
30    int y = (280 - h) / 2 - y1;
31    tft.setCursor(x, y);
32    tft.print("Mitte");
```



So kommt die Rechnung in den Zeilen 29 und 30 zustande:

- Der Kasten ist `w` breit. Damit links und rechts gleich viel Platz bleibt, muss seine linke Kante bei `(240 - w) / 2` liegen.
- Die linke Kante liegt immer `x1` Pixel rechts vom Cursor. Also gehört der Cursor nach `(240 - w) / 2 - x1`.
- Genauso senkrecht: Die Oberkante soll bei `(280 - h) / 2` liegen und liegt `y1` Pixel unter dem Cursor. `y1` ist hier **negativ**, weil der Kasten über der Grundlinie beginnt. Minus eine negative Zahl ergibt plus: Der Cursor wandert nach unten auf die Grundlinie.

Die grauen Hilfslinien kreuzen sich genau in der Mitte des Displays, und „Mitte“ steht genau darauf.

Text waagerecht zentrieren, in jeder Schrift:

```
tft.getTextBounds("Text", 0, 0, &x1, &y1, &w, &h);
tft.setCursor((240 - w) / 2 - x1, y);    // y: die Grundlinie
tft.print("Text");
```

Ausprobieren: Eine Wetteranzeige

k06_wetter.ino

```
// Eine Wetteranzeige aus Formen, Farben und FreeFonts
#include <Adafruit_ST7789.h>
#include <Fonts/FreeSansBold18pt7b.h>
#include <Fonts/FreeSans12pt7b.h>
```

k06_wetter.ino

```
    tft.fillScreen(tft.color565(70, 150, 230));     // Himmel

    // Sonne mit acht Strahlen
    tft.fillCircle(150, 80, 30, ST77XX_YELLOW);
    tft.drawLine(150, 44, 150, 32, ST77XX_YELLOW);
    tft.drawLine(150, 116, 150, 128, ST77XX_YELLOW);
    tft.drawLine(114, 80, 102, 80, ST77XX_YELLOW);
    tft.drawLine(186, 80, 198, 80, ST77XX_YELLOW);
    tft.drawLine(175, 55, 184, 46, ST77XX_YELLOW);
    tft.drawLine(125, 55, 116, 46, ST77XX_YELLOW);
    tft.drawLine(175, 105, 184, 114, ST77XX_YELLOW);
    tft.drawLine(125, 105, 116, 114, ST77XX_YELLOW);

    // Wolke vor der Sonne: drei Kreise und ein Rechteck mit runden Ecken
    uint16_t wolke = tft.color565(240, 240, 240);
    tft.fillCircle(80, 120, 22, wolke);
    tft.fillCircle(110, 106, 28, wolke);
    tft.fillCircle(142, 120, 20, wolke);
    tft.fillRoundRect(58, 118, 106, 24, 12, wolke);

    // Texte, waagerecht in der Mitte
    int16_t x1, y1;
    uint16_t w, h;
    tft.setTextColor(ST77XX_WHITE);
    tft.setFont(&FreeSansBold18pt7b);
    tft.getTextBounds("Heiter", 0, 0, &x1, &y1, &w, &h);
    tft.setCursor((240 - w) / 2 - x1, 200);
    tft.print("Heiter");

    tft.setFont(&FreeSans12pt7b);
    tft.getTextBounds("24 Grad", 0, 0, &x1, &y1, &w, &h);
    tft.setCursor((240 - w) / 2 - x1, 235);
    tft.print("24 Grad");
```



- Zeile 19 färbt den ganzen Bildschirm himmelblau. Die Farbe wird hier direkt als Argument gemischt, ohne Variable.
- Die Sonne ist ein gefüllter Kreis mit acht Linien als Strahlen. Jede Linie beginnt 6 Pixel außerhalb des Kreises (Radius 30, Linie ab 36) und ist 12 Pixel lang.
- Die Wolke besteht aus drei weißen Kreisen und einem Rechteck mit runden Ecken als flachem Boden. Weil sie **nach** der Sonne gezeichnet wird, schiebt sie sich davor.
- Beide Texte sind waagerecht zentriert. Ihre Grundlinien (y = 200 und 235) sind fest gewählt. Die Variablen `x1`, `y1`, `w` und `h` werden für den zweiten Text einfach wiederverwendet.

Zum Verändern: Zeichne Regentropfen unter die Wolke (kurze Linien) und schreib „Regen“ statt „Heiter“. Oder mach aus dem Tag eine Nacht: dunkelblauer Himmel, ein Mond aus zwei Kreisen (ein heller und ein himmelfarbener, der ihn teilweise verdeckt) und ein paar Sterne aus einzelnen Pixeln.

Übungsaufgaben

Aufgabe 1 ⭐ Eine Flagge Zeichne die Flagge Deutschlands aus drei gefüllten Rechtecken: Schwarz, Rot (`color565(221, 0, 0)`) und Gold (`color565(255, 206, 0)`), auf grauem Grund. Oder eine andere Flagge deiner Wahl.

Aufgabe 2 ⭐ Was wird das? Lies den Sketch, ohne ihn hochzuladen, und skizziere auf Papier, was er zeichnet. Prüfe danach auf dem PicoBoy.

k06_aufgabe2.ino

```
17      tft.fillScreen(ST77XX_BLUE);
18      tft.fillRect(0, 240, 240, 40, ST77XX_WHITE);
19      tft.fillCircle(120, 205, 45, ST77XX_WHITE);
20      tft.fillCircle(120, 130, 33, ST77XX_WHITE);
21      tft.fillCircle(120, 75, 24, ST77XX_WHITE);
22      tft.fillRect(98, 44, 44, 8, ST77XX_BLACK);
23      tft.fillRect(106, 20, 28, 26, ST77XX_BLACK);
24      tft.fillCircle(112, 68, 3, ST77XX_BLACK);
25      tft.fillCircle(128, 68, 3, ST77XX_BLACK);
26      tft.fillTriangle(120, 76, 120, 84, 146, 80, ST77XX_ORANGE);
27      tft.fillCircle(120, 120, 4, ST77XX_BLACK);
28      tft.fillCircle(120, 140, 4, ST77XX_BLACK);
```

Aufgabe 3 ⭐⭐ Zielscheibe Zeichne eine Zielscheibe aus fünf Ringen, abwechselnd weiß und rot, mit einem kleinen roten Punkt in der Mitte. Der äußere Ring hat den Radius 100. Was passiert, wenn du die Kreise in umgekehrter Reihenfolge zeichnest?

Aufgabe 4 ⭐⭐ **Namensschild** Zeichne ein farbiges Schild mit runden Ecken (200 × 80 Pixel) und schreib deinen Namen in `FreeSansBold18pt7b` genau in seine Mitte, waagerecht **und** senkrecht.

Aufgabe 5 ⭐⭐⭐ **Text mit Rahmen** Setze ein Wort in einer FreeFont genau in die Mitte des Displays und zeichne einen Rahmen mit runden Ecken darum, der auf allen vier Seiten genau 10 Pixel Abstand zum Text hat. Der Rahmen soll stimmen, egal welches Wort du einsetzt.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `draw…` zeichnet den Umriss, `fill…` die gefüllte Form: `Line`, `Rect`, `RoundRect`, `Circle`, `Triangle`. Die Farbe steht immer zuletzt.
- Rechteck: **x, y, Breite, Höhe**. Kreis: **Mittelpunkt** und **Radius**. Dreieck: drei Ecken.
- Was später gezeichnet wird, liegt oben.
- Eigene Farben: `tft.color565(rot, grün, blau)` mit 0 bis 255. Eine Farbe ist eine 16-Bit-Zahl, Typ `uint16_t`.
- FreeFonts: `#include <Fonts/FreeSans12pt7b.h>`, `tft.setFont(&FreeSans12pt7b);`, zurück mit `tft.setFont();`. Der Cursor steht auf der **Grundlinie**. Keine Umlaute, kein Hintergrund.
- `getTextBounds(text, x, y, &x1, &y1, &w, &h)` misst einen Text. Zentriert: Cursor bei `(240 - w) / 2 - x1`.

KAPITEL 7

Schleifen und Zeit

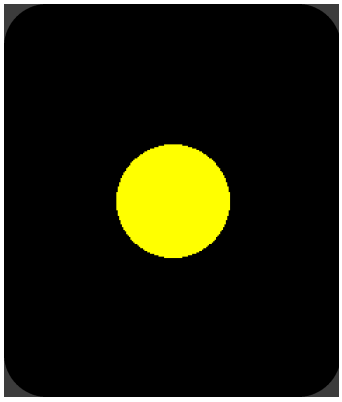
Worum geht's? Ein Mikrocontroller ist schnell und wird nie müde: Er wiederholt dieselben Anweisungen tausendfach, ohne sich zu beschweren. In diesem Kapitel lernst du, wie du Wiederholungen mit `for` und `while` aufschreibst, wie der PicoBoy die Zeit misst und wie sich etwas über das Display bewegt. Dabei begegnet dir das Flackern, das Kapitel 3 angekündigt hat, und du lernst, wie man wartet, ohne den ganzen Sketch anzuhalten.

7.1 `delay` und `millis`

`delay(ms)` kennst du schon: Es wartet die angegebene Zeit in **Millisekunden** (ms). 1000 ms sind eine Sekunde. Während `delay` wartet, tut der Sketch nichts anderes.

k07_blinken.ino

```
18 void loop() {
19     tft.fillCircle(120, 140, 40, ST77XX_YELLOW);
20     delay(500);
21     tft.fillCircle(120, 140, 40, ST77XX_BLACK);
22     delay(500);
23 }
```



Weil `loop()` immer wieder läuft, blinkt der Kreis: eine halbe Sekunde gelb, eine halbe Sekunde schwarz. Ein Durchlauf von `loop()` dauert hier eine Sekunde, fast nur Warten.

`millis()` ist die eingebaute Uhr. Sie liefert, wie viele Millisekunden seit dem Start des Sketches vergangen sind:

k07_millis.ino

```
1 // Wie lange laeuft der Sketch schon?
2 void setup() {
3     Serial.begin(115200);
4 }
5
6 void loop() {
7     Serial.printf("Seit dem Start: %lu ms\n", millis());
8     delay(1000);
9 }
```

Serieller Monitor

```
Seit dem Start: 0 ms
Seit dem Start: 1000 ms
Seit dem Start: 2000 ms
Seit dem Start: 3000 ms
```

- Der Platzhalter `%lu` steht für *long unsigned*: `millis()` liefert ein `unsigned long` (Kapitel 5), und dafür ist `%d` nicht der richtige Platzhalter.
- Mit jeder Zeile wächst die Zahl um 1000, denn `delay(1000)` wartet eine Sekunde. Die Ausgabe und der Rest von `loop()` dauern nur Bruchteile einer Millisekunde; nach vielen Zeilen kann deshalb eine Millisekunde mehr dazukommen.
- Der Sketch wartet nicht auf den Monitor. Auf deinem PicoBoy fehlen deshalb meist die ersten Zeilen; sie sind weg, bevor der Monitor zuhört (Kapitel 4.1). Die Zeit läuft trotzdem ab dem Start.

`millis()` zählt bis 4 294 967 295, das sind gut 49 Tage. Danach beginnt es wieder bei 0 (Überlauf, Kapitel 5.2). Für Spiele spielt das keine Rolle.

Eine Dauer messen: Merkst du dir die Uhrzeit vor und nach einer Anweisung, ist der Unterschied die Dauer. So findest du heraus, wie lange es dauert, den ganzen Bildschirm zu füllen:

k07_dauer.ino

```
19 void loop() {
20     unsigned long start = millis();
21     tft.fillScreen(ST77XX_BLUE);
22     unsigned long dauer = millis() - start;
23     Serial.printf("fillScreen dauert %lu ms\n", dauer);
24     delay(1000);
25 }
```

Serieller Monitor

```
fillScreen dauert 43 ms
fillScreen dauert 43 ms
fillScreen dauert 43 ms
```

Rund 43 ms für ein einziges `fillScreen`! Die Zahl kommt so zustande: Jedes der 240 × 280 = 67 200 Pixel braucht 16 Bit für seine Farbe, und die gehen Bit für Bit über die Leitung zum Display. Bei 25 Millionen Bit pro Sekunde dauert das 67 200 × 16 / 25 000 000 s ≈ 0,043 s. Kleine Formen gehen entsprechend schneller: Ein Kreis mit Radius 20 hat nur etwa 1 300 Pixel. Diese Zahl wird gleich wichtig.

7.2 `for` und `while`

Eine **Schleife** wiederholt einen Block. `loop()` ist schon eine Schleife, aber eine, die nie endet. Oft soll etwas aber genau zehnmal passieren, oder so lange, bis eine Bedingung nicht mehr stimmt.

Die `for`-Schleife zählt:

k07_for.ino

```
1 // Von 1 bis 10 zaehlen
2 void setup() {
3     Serial.begin(115200);
```

```
4     while (!Serial) {
5         delay(10);
6     }
7     for (int i = 1; i <= 10; i++) {
8         Serial.println(i);
9     }
10    Serial.println("Fertig!");
11 }
12
13 void loop() {
14 }
```

Serieller Monitor

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

In der runden Klammer hinter `for` stehen drei Teile, getrennt durch Semikolons:

Teil	hier	Bedeutung
Anfang	`int i = 1`	legt die **Zählvariable** `i` an, einmal vor dem ersten Durchlauf
Bedingung	`i <= 10`	vor **jedem** Durchlauf geprüft: Stimmt sie, läuft der Block noch einmal
Schritt	`i++`	nach jedem Durchlauf ausgeführt: `i` um 1 erhöhen

Der Ablauf ist also: `i` ist 1. Ist 1 ≤ 10? Ja, also Block ausführen (gibt 1 aus), dann `i++`. Ist 2 ≤ 10? Ja … und so weiter, bis `i` 11 ist. Dann stimmt die Bedingung nicht mehr, und der Sketch macht hinter der Schleife weiter (Zeile 10). Jeder Durchlauf heißt **Durchgang** oder **Iteration**.

Die Bedingung ist ein **Vergleich**. Diese Vergleiche gibt es:

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

Kapitel 8 zeigt, was man mit Vergleichen noch alles anstellen kann.

Anfang, Bedingung und Schritt kannst du frei wählen, etwa rückwärts oder in größeren Schritten:

k07_for_varianten.ino

```
 7      for (int i = 10; i >= 0; i--) {
 8          Serial.print(i);
 9          Serial.print(" ");
10      }
11      Serial.println();
12
13      for (int i = 0; i <= 100; i += 25) {
14          Serial.print(i);
15          Serial.print(" ");
16      }
17      Serial.println();
```

Serieller Monitor

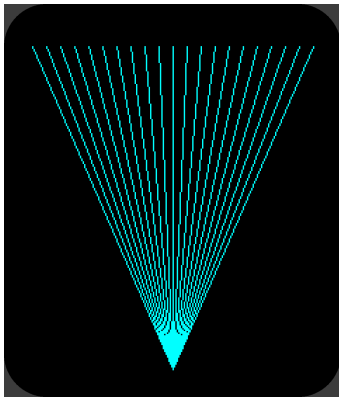
```
10 9 8 7 6 5 4 3 2 1 0
0 25 50 75 100
```

Die Zählvariable heißt traditionell `i`. Sie gilt nur **innerhalb** der Schleife: Hinter der schließenden Klammer ist `i` wieder unbekannt. Deshalb darf die zweite Schleife ein neues `i` anlegen.

Schleifen zum Zeichnen: Mit der Zählvariablen als Koordinate entstehen Muster, für die du sonst Dutzende Zeilen bräuchtest:

k07_faecher.ino

```
17      for (int x = 20; x <= 220; x += 10) {
18          tft.drawLine(120, 260, x, 30, ST77XX_CYAN);
19      }
```

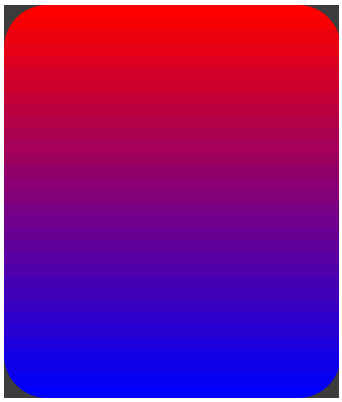


Alle Linien beginnen unten in der Mitte (120, 260). Ihr oberes Ende wandert in Schritten von 10 Pixeln von x = 20 bis x = 220. Das sind 21 Linien mit drei Zeilen Code.

k07_verlauf.ino

```
17      for (int y = 0; y < 280; y++) {
18          int blau = y * 255 / 279;
19          tft.drawLine(0, y, 239, y, tft.color565(255 - blau, 0, blau));
20      }
```

Hier zeichnet die Schleife 280 waagerechte Linien, eine pro Zeile des Displays. Die Farbe hängt von `y` ab: In Zeile 18 wird aus y = 0 … 279 ein Blauanteil von 0 … 255, der Rotanteil ist der Rest bis 255. Oben ist es rot, unten blau, dazwischen Lila. Beachte die Reihenfolge `y * 255 / 279`: erst malnehmen, dann teilen. Andersherum wäre `255 / 279` ganzzahlig 0 (Kapitel 5.2).

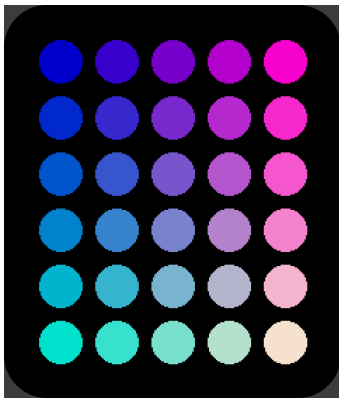


Schleifen in Schleifen: Steht eine Schleife im Block einer anderen, läuft die innere bei jedem Durchgang der äußeren vollständig durch. Das ergibt Zeilen und Spalten:

k07_raster.ino

```
17      for (int zeile = 0; zeile < 6; zeile++) {
18          for (int spalte = 0; spalte < 5; spalte++) {
19              int x = 40 + spalte * 40;
20              int y = 40 + zeile * 40;
21              tft.fillCircle(x, y, 15, tft.color565(spalte * 60, zeile * 45, 200));
22          }
23      }
```

Die äußere Schleife zählt sechs Zeilen, die innere fünf Spalten. Zusammen sind das 6 × 5 = 30 Kreise. Aus Zeile und Spalte berechnen die Zeilen 19 und 20 den Mittelpunkt, Zeile 21 die Farbe: nach rechts mehr Rot, nach unten mehr Grün.



Die `while`-Schleife hat nur eine Bedingung. Sie wiederholt ihren Block, **solange** die Bedingung stimmt. Das passt, wenn man vorher nicht weiß, wie oft es nötig ist:

k07_while.ino

```
 7      int zahl = 1;
 8      while (zahl < 1000) {
 9          Serial.println(zahl);
10          zahl = zahl * 2;
11      }
12      Serial.printf("Erst %d ist nicht mehr kleiner als 1000.\n", zahl);
```

Serieller Monitor

```
1
2
4
```

```
8
16
32
64
128
256
512
Erst 1024 ist nicht mehr kleiner als 1000.
```

Wie oft muss man 1 verdoppeln, bis es 1000 erreicht? Das weiß der Sketch vorher nicht; er probiert es einfach aus. Die Schleife aus Kapitel 4, `while (!Serial) { delay(10); }`, funktioniert genauso: Solange kein Monitor zuhört, wartet sie 10 ms und prüft erneut.

> Ändert sich im Block nichts an der Bedingung, endet eine `while`-Schleife nie. Vergisst du hier `zahl = zahl * 2;`, gibt der Sketch für immer „1“ aus. Das ist kein Fehler, den der Compiler findet; der PicoBoy hängt einfach in der Schleife.

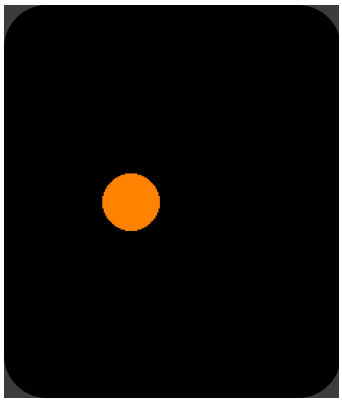
> `for` oder `while`? Weißt du vorher, wie oft, nimm `for`. Hängt es von etwas ab, das sich erst unterwegs zeigt (eine Eingabe, eine Taste, eine Rechnung), nimm `while`.

7.3 Animation – und warum es flackert

Eine **Animation** ist eine schnelle Folge von Bildern, die sich jeweils nur ein wenig unterscheiden. Ab etwa 20 Bildern pro Sekunde sieht das Auge eine Bewegung. Der naheliegende Weg: Bildschirm löschen, Ball an neuer Stelle zeichnen, kurz warten, wiederholen.

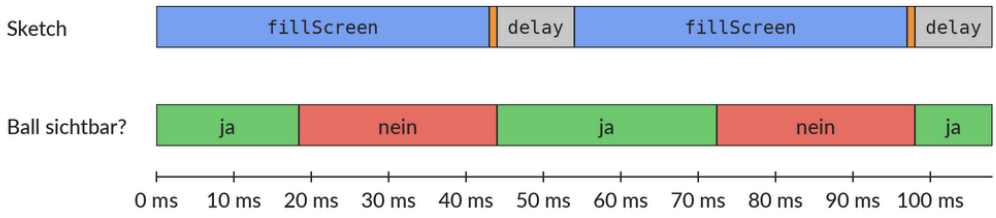
k07_ball_flackert.ino

```
18 void loop() {
19     for (int x = 30; x <= 210; x += 4) {
20         tft.fillScreen(ST77XX_BLACK);
21         tft.fillCircle(x, 140, 20, ST77XX_ORANGE);
22         delay(10);
23     }
24 }
```



Der Ball rollt von links nach rechts, springt zurück und rollt wieder los. Auf dem PicoBoy **flackert** er aber stark. Das Bild hier im Buch kann das nicht zeigen; probiere es aus.

Woher kommt das Flackern? Das Display zeigt jeden Befehl sofort an (Kapitel 3.2). `fillScreen` braucht 43 ms und malt den Bildschirm dabei von oben nach unten schwarz. Sobald es bei der Mitte ankommt, nach etwa 18 ms, ist der Ball übermalt. Erst wenn `fillScreen` fertig ist, zeichnet `fillCircle` ihn neu. So sieht ein Durchgang der Schleife aus:



orange: fillCircle, etwa 1 ms. Werte gerundet, Ball in der Mitte des Displays.

In jedem Bild ist der Ball etwa 25 ms lang **gar nicht zu sehen**, fast die Hälfte der Zeit. Das bemerkt das Auge als Flackern. Mehr warten hilft nicht: Mit einem längeren `delay` ist der Ball zwar länger zu sehen, aber die Bewegung wird ruckelig, und die Lücke bleibt.

Teilweise Abhilfe: nur das Nötige übermalen. Statt den ganzen Bildschirm zu löschen, übermalst du nur den alten Ball mit der Hintergrundfarbe:

k07_ball_ruhig.ino

```
18 void loop() {
19     for (int x = 30; x <= 210; x += 4) {
20         tft.fillCircle(x, 140, 20, ST77XX_ORANGE);
21         delay(10);
22         tft.fillCircle(x, 140, 20, ST77XX_BLACK);   // nur den Ball loeschen
23     }
24 }
```

Ein Kreis mit Radius 20 ist in rund 1 ms gezeichnet. Die Lücke zwischen Löschen (Zeile 22) und neu Zeichnen (Zeile 20 im nächsten Durchgang) ist also winzig, und der Ball ist fast die ganze Zeit zu sehen. Das Flackern ist weitgehend weg. Außerdem läuft die Animation schneller, weil kein `fillScreen` mehr bremst.

Ganz zufrieden kann man trotzdem nicht sein:

- Für einen kurzen Moment ist der Ball noch immer weg. Bei großen oder vielen Figuren fällt das wieder auf.
- Liegt der Ball über einem Hintergrundbild oder einem Text, löscht das schwarze Übermalen auch den Hintergrund. Man müsste ihn jedes Mal wiederherstellen.

Die richtige Lösung lernst du in Kapitel 14: Das ganze Bild wird zuerst unsichtbar im Arbeitsspeicher gezeichnet und dann auf einmal zum Display geschickt. Bis dahin hilft der Trick mit dem Übermalen.

> Direkt auf `tft` gezeichnet, ist jeder Schritt sofort zu sehen, auch das Löschen. Darum flackert „alles löschen, neu zeichnen“. Lösche nur, was sich bewegt.

7.4 Warten ohne `delay`: der `millis`-Takt

`delay` hat einen Nachteil: Während es wartet, kann der Sketch nichts anderes tun. Sollen zwei Dinge in verschiedenem Tempo passieren, etwa eine Uhr, die jede Sekunde weiterzählt, und eine Figur, die sich alle 20 ms bewegt, dann blockieren sie sich mit `delay` gegenseitig.

Die Lösung: Statt zu warten, schaust du in jedem Durchlauf von `loop()` auf die Uhr und fragst: Ist es schon wieder Zeit? Dafür brauchst du `if`:

```
if (bedingung) {
    // nur, wenn die Bedingung stimmt
}
```

`if` funktioniert wie `while`, nur **ohne Wiederholung**: Stimmt die Bedingung, wird der Block einmal ausgeführt, sonst übersprungen. Alles Weitere über `if` steht in Kapitel 8.

k07_takt.ino

```
unsigned long letzte_sekunde = 0;
unsigned long letzter_schritt = 0;
int sekunden = 0;
int position = 0;

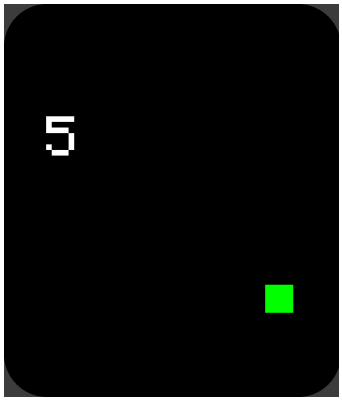
void setup() {
    tft.init(240, 280);
    tft.fillScreen(ST77XX_BLACK);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
    tft.setTextSize(4);
    tft.setTextColor(ST77XX_WHITE, ST77XX_BLACK);
}

void loop() {
    unsigned long jetzt = millis();

    if (jetzt - letzte_sekunde >= 1000) {       // einmal pro Sekunde
        letzte_sekunde = jetzt;
        sekunden++;
        tft.setCursor(30, 80);
        tft.print(sekunden);
    }

    if (jetzt - letzter_schritt >= 20) {        // alle 20 Millisekunden
        letzter_schritt = jetzt;
        tft.fillRect(30 + position, 200, 20, 20, ST77XX_BLACK);
        position = (position + 2) % 160;
```

```
39          tft.fillRect(30 + position, 200, 20, 20, ST77XX_GREEN);
40      }
41  }
```



Oben zählt eine Zahl die Sekunden, unten fährt ein grünes Quadrat immer wieder von links nach rechts. So funktioniert es:

- Für jede Aufgabe merkt sich eine Variable, wann sie zuletzt dran war: `letzte_sekunde` und `letzter_schritt` (Zeilen 11 und 12). Sie stehen außerhalb von `loop()`, damit sie ihren Wert behalten (Kapitel 5.1), und haben den Typ `unsigned long`, wie `millis()`.
- Zeile 26 liest einmal die Uhr.
- Zeile 28 fragt: Ist seit dem letzten Mal mindestens 1000 ms her? Wenn ja, merkt sich Zeile 29 die neue Zeit, und die Sekunden werden weitergezählt. Wenn nein, passiert nichts, und `loop()` geht weiter.
- Zeile 35 macht dasselbe alle 20 ms für das Quadrat: altes Quadrat übermalen, Position weiterrücken, neu zeichnen.
- `% 160` in Zeile 38 lässt die Position von 158 wieder auf 0 springen (Rest der Division, Kapitel 5.2). Das Quadrat fährt also immer wieder von vorn.
- `setTextColor(ST77XX_WHITE, ST77XX_BLACK)` in Zeile 22 sorgt dafür, dass jede neue Zahl die alte überschreibt (Kapitel 3.3).

`loop()` hat hier kein einziges `delay`. Es läuft Tausende Male pro Sekunde, und fast immer stellt es nur fest, dass noch nichts zu tun ist. Genau das ist gewollt: Jede weitere Aufgabe, etwa eine Taste abfragen (Kapitel 9), bekommt ihren eigenen Block mit eigener Zeitvariable und kommt sofort dran.

Der `millis`-Takt:

```
unsigned long letztes_mal = 0;     // außerhalb von loop()

void loop() {
    if (millis() - letztes_mal >= 500) {
        letztes_mal = millis();
        // was alle 500 ms passieren soll
    }
}
```

Schreib die Bedingung immer als `millis() - letztes_mal >= dauer`, also als **Unterschied**. Dann stimmt sie auch noch, wenn `millis()` nach 49 Tagen überläuft: Die Subtraktion von `unsigned long`-Werten läuft genauso über und ergibt trotzdem die richtige Dauer.

Ausprobieren: Eine Stoppuhr

k07_stoppuhr.ino

```
11 unsigned long start = 0;
12 unsigned long letzte_anzeige = 0;
13
14 void setup() {
15     tft.init(240, 280);
16     tft.fillScreen(ST77XX_BLACK);
17     pinMode(TFT_LICHT, OUTPUT);
18     digitalWrite(TFT_LICHT, HIGH);
19
20     tft.setTextSize(2);
21     tft.setTextColor(ST77XX_CYAN);
22     tft.setCursor(66, 80);
23     tft.print("Stoppuhr");
24     tft.setTextSize(4);
25     tft.setTextColor(ST77XX_WHITE, ST77XX_BLACK);
26     start = millis();
27 }
28
29 void loop() {
30     unsigned long jetzt = millis();
31     if (jetzt - letzte_anzeige >= 100) {
32         letzte_anzeige = jetzt;
33         unsigned long zeit = jetzt - start;
34         int minuten = zeit / 60000;
35         int sekunden = zeit / 1000 % 60;
36         int zehntel = zeit / 100 % 10;
37         tft.setCursor(36, 124);
38         tft.printf("%02d:%02d.%d", minuten, sekunden, zehntel);
39     }
40 }
```

Das Bild entstand nach gut einer Minute.



- `start` merkt sich in Zeile 26 die Uhrzeit am Ende von `setup()`. Die Stoppuhr beginnt also bei 0, egal wie lange der Start gedauert hat.
- Alle 100 ms (Zeile 31) wird die Anzeige erneuert, öfter ist nicht nötig, denn sie zeigt nur Zehntelsekunden.
- Die Zeilen 34 bis 36 zerlegen die Millisekunden mit `/` und `%` in Minuten, Sekunden und Zehntel, wie in Kapitel 5.2.
- `%02d` im `printf` (Zeile 38) bedeutet: eine ganze Zahl mit mindestens 2 Stellen, vorn mit Nullen aufgefüllt. Aus 5 wird `05`. So bleibt die Anzeige immer gleich breit, und die neue Zeit überschreibt die alte genau.

Zum Verändern: Lass die Stoppuhr zusätzlich Stunden anzeigen. Oder mache aus ihr einen Countdown, der von 3 Minuten rückwärts zählt.

Übungsaufgaben

Aufgabe 1 ⭐ **Countdown** Gib im seriellen Monitor die Zahlen von 10 bis 1 im Sekundentakt aus und danach „Start!“.

Aufgabe 2 ⭐ **Einmaleins** Gib die Siebenerreihe aus, von `1 x 7 = 7` bis `10 x 7 = 70`, mit einer `for`-Schleife und `Serial.printf`.

Aufgabe 3 ⭐⭐ **Ladebalken** Zeichne einen Rahmen und fülle ihn innerhalb von 5 Sekunden von links nach rechts mit einem grünen Balken. Darunter läuft die Prozentzahl von 0 bis 100 mit.

Aufgabe 4 ⭐⭐ **Schachbrett** Zeichne ein Schachbrett aus 8 × 8 Feldern mit zwei verschachtelten Schleifen. Tipp: Zeichne erst eine große helle Fläche und dann nur die dunklen Felder darauf. In welcher Spalte beginnt das erste dunkle Feld einer Zeile?

Aufgabe 5 ⭐⭐⭐ **Bilder pro Sekunde** Wie viele Bilder pro Sekunde schafft die flackernde Animation aus 7.3? Zähle in `loop()` die gezeichneten Bilder und gib die Zahl mit dem `millis`-Takt jede Sekunde im seriellen Monitor aus. Passt das Ergebnis zur Zeitleiste aus Abschnitt 7.3?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `delay(ms)` wartet und hält dabei alles an. `millis()` ist die Zeit seit dem Start in ms, Typ `unsigned long`, Platzhalter `%lu`.
- `for (int i = 0; i < 10; i++) { … }`: Anfang, Bedingung, Schritt. `while (bedingung) { … }`: solange die Bedingung stimmt.
- Vergleiche: `<`, `<=`, `>`, `>=`, `==`, `!=`.
- `fillScreen` dauert etwa 43 ms. Direkt gezeichnete Animationen flackern, wenn der ganze Bildschirm gelöscht wird. Besser: nur den alten Platz übermalen; ganz flimmerfrei in Kapitel 14.
- Der `millis`-Takt: `if (millis() - letztes_mal >= dauer) { letztes_mal = millis(); … }` – mehrere Aufgaben, jede in ihrem Tempo, ohne `delay`.

KAPITEL 8

Entscheidungen

Worum geht's? Bisher hat jeder Sketch alle Anweisungen der Reihe nach ausgeführt. Ein Spiel muss aber reagieren: Ist der Ball am Rand? Hat der Spieler noch Leben? War der Tipp zu klein oder zu groß? In diesem Kapitel lernst du, wie ein Sketch mit `if`, `else` und `switch` Entscheidungen trifft, wie man Bedingungen verknüpft und welche Falle in einem einzelnen `=` lauert.

8.1 `if`, `else`, `switch`

`if` (englisch: wenn) führt einen Block nur aus, wenn eine Bedingung stimmt. In Kapitel 7.4 hast du es schon für den `millis`-Takt benutzt. Hier würfelt der PicoBoy jede Sekunde und freut sich über jede Sechs:

k08_if.ino

```
 9 void loop() {
10     int wurf = random(1, 7);
11     Serial.printf("Gewuerfelt: %d\n", wurf);
12     if (wurf == 6) {
13         Serial.println("Eine Sechs! Du darfst noch einmal.");
14     }
15     delay(1000);
16 }
```

Serieller Monitor

```
Gewuerfelt: 6
Eine Sechs! Du darfst noch einmal.
Gewuerfelt: 4
Gewuerfelt: 3
Gewuerfelt: 6
Eine Sechs! Du darfst noch einmal.
Gewuerfelt: 1
Gewuerfelt: 1
Gewuerfelt: 5
Gewuerfelt: 1
Gewuerfelt: 3
```

Zeile 11 läuft bei jedem Wurf. Die Zeile 13 dagegen nur, wenn die Bedingung `wurf == 6` stimmt. Ist der Wurf keine Sechs, überspringt der Sketch den ganzen Block und macht bei Zeile 15 weiter. Beachte die **zwei** Gleichheitszeichen: `==` vergleicht, ein einzelnes `=` würde etwas hineinschreiben (Abschnitt 8.3).

`else` (englisch: sonst) gibt einen zweiten Block an, der läuft, wenn die Bedingung **nicht** stimmt. Von beiden Blöcken läuft also immer genau einer:

k08_else.ino

```
 9 void loop() {
10     int muenze = random(2);
11     if (muenze == 0) {
12         Serial.println("Kopf");
13     } else {
14         Serial.println("Zahl");
15     }
16     delay(1000);
17 }
```

Serieller Monitor

```
Zahl
Zahl
Kopf
Zahl
Kopf
Kopf
```

`random(2)` liefert 0 oder 1 (Kapitel 5.4). Bei 0 heißt es „Kopf“, in allen anderen Fällen „Zahl“.

`else if`: Gibt es mehr als zwei Möglichkeiten, hängst du weitere `if` hinten an. Der Sketch prüft die Bedingungen der Reihe nach von oben nach unten. Die **erste**, die stimmt, gewinnt; alle weiteren werden gar nicht mehr geprüft. Stimmt keine, läuft das `else` am Ende:

k08_noten.ino

```
 7     for (int punkte = 100; punkte >= 0; punkte -= 15) {
 8         int note;
 9         if (punkte >= 90) {
10             note = 1;
11         } else if (punkte >= 75) {
12             note = 2;
13         } else if (punkte >= 60) {
14             note = 3;
15         } else if (punkte >= 45) {
16             note = 4;
17         } else if (punkte >= 20) {
18             note = 5;
19         } else {
20             note = 6;
21         }
22         Serial.printf("%3d Punkte: Note %d\n", punkte, note);
23     }
```

Serieller Monitor

```
100 Punkte: Note 1
 85 Punkte: Note 2
 70 Punkte: Note 3
 55 Punkte: Note 4
 40 Punkte: Note 5
 25 Punkte: Note 5
 10 Punkte: Note 6
```

Bei 85 Punkten stimmt `punkte >= 90` nicht, aber `punkte >= 75`. Also Note 2, und die Bedingungen darunter sind egal, obwohl 85 auch größer als 60 ist. Deshalb müssen die Grenzen von oben nach unten kleiner werden. Andersherum (zuerst `punkte >= 20`) bekäme fast jeder eine 5.

Zeile 8 legt `note` ohne Wert an: `int note;`. Das geht, weil jeder der sechs Zweige einen Wert hineinschreibt, bevor Zeile 22 ihn liest. `%3d` füllt die Punktzahl vorn mit Leerzeichen auf 3 Stellen auf, damit die Spalten untereinanderstehen.

`switch`: Hängt die Entscheidung nur davon ab, welchen **ganzzahligen Wert** eine Variable hat, ist `switch` übersichtlicher als eine lange Kette aus `else if`:

k08_switch.ino

```
 7      for (int tag = 1; tag <= 8; tag++) {
 8          Serial.printf("%d: ", tag);
 9          switch (tag) {
10              case 1:
11                  Serial.println("Montag");
12                  break;
13              case 2:
14                  Serial.println("Dienstag");
15                  break;
16              case 3:
17                  Serial.println("Mittwoch");
18                  break;
19              case 4:
20                  Serial.println("Donnerstag");
21                  break;
22              case 5:
23                  Serial.println("Freitag");
24                  break;
25              case 6:
26              case 7:
27                  Serial.println("Wochenende!");
28                  break;
29              default:
30                  Serial.println("diesen Tag gibt es nicht");
31                  break;
32          }
33      }
```

Serieller Monitor

```
1: Montag
2: Dienstag
3: Mittwoch
4: Donnerstag
5: Freitag
6: Wochenende!
7: Wochenende!
8: diesen Tag gibt es nicht
```

- `switch (tag)` springt zu der Marke `case`, deren Zahl zu `tag` passt.
- Ab dort laufen die Anweisungen, bis ein **`break`** kommt. `break` beendet das `switch`; weiter geht es hinter seiner schließenden Klammer.
- Mehrere Marken dürfen untereinanderstehen (Zeilen 25 und 26). Bei 6 und bei 7 heißt es „Wochenende!“.
- `default` (Standard) läuft, wenn keine Marke passt, hier bei 8.

`switch` funktioniert nur mit ganzen Zahlen und Zeichen wie `'A'`, nicht mit Kommazahlen und nicht mit Texten. Und es prüft nur auf **Gleichheit**; für „größer als 75“ brauchst du `if`.

Das vergessene `break`: Fehlt ein `break`, läuft der Sketch einfach in den nächsten `case` hinein:

k08_switch_ohne_break.ino

```
 7      for (int tag = 1; tag <= 3; tag++) {
 8          Serial.printf("%d: ", tag);
 9          switch (tag) {
10              case 1:
11                  Serial.print("Montag ");
12              case 2:
13                  Serial.print("Dienstag ");
14                  break;
15              case 3:
16                  Serial.print("Mittwoch ");
17                  break;
18          }
19          Serial.println();
20      }
```

Serieller Monitor

```
1: Montag Dienstag
2: Dienstag
3: Mittwoch
```

Bei Tag 1 steht „Montag Dienstag“. Der Compiler bemerkt das und warnt, wenn die Compiler-Meldungen auf **Alle** stehen (Kapitel 3.6):

Ausgabe

```
k08_switch_ohne_break.ino: In function 'void setup()':
k08_switch_ohne_break.ino:11:29: warning: this statement may fall through [-
  Wimplicit-fallthrough=]
   11 |                 Serial.print("Montag ");
      |                 ~~~~~~~~~~~~^~~~~~~~~~
k08_switch_ohne_break.ino:12:13: note: here
   12 |             case 2:
      |             ^~~~
```

This statement may fall through: Diese Anweisung könnte in die nächste durchfallen. Das Hineinlaufen heißt auf Englisch *fall through*. Manchmal ist es gewollt, wie bei 6 und 7 oben, wo zwischen den Marken gar nichts steht. Dort warnt der Compiler auch nicht.

8.2 Vergleiche und logische Verknüpfungen

Die Vergleiche kennst du aus Kapitel 7.2: `<`, `<=`, `>`, `>=`, `==` und `!=`. Jeder Vergleich hat ein Ergebnis: **wahr** (`true`) oder **falsch** (`false`), also genau das, was in eine Variable vom Typ `bool` passt (Kapitel 5.1). Mit `&&`, `||` und `!` verknüpfst du mehrere Vergleiche:

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

Das `&` tippst du mit **Umschalt + 6**, den senkrechten Strich `|` unter Windows mit **AltGr + <** (die Taste links neben Y), am Mac mit **Alt + 7**.

k08_vergleiche.ino

```
 7     int alter = 15;
 8     Serial.println(alter > 12);
 9     Serial.println(alter == 18);
10     Serial.println(alter != 18);
11
12     bool jugendlich = alter >= 12 && alter < 18;
13     bool kind_oder_rentner = alter < 12 || alter >= 67;
14     Serial.println(jugendlich);
15     Serial.println(kind_oder_rentner);
16     Serial.println(!jugendlich);
```

Serieller Monitor

```
1
0
1
1
0
0
```

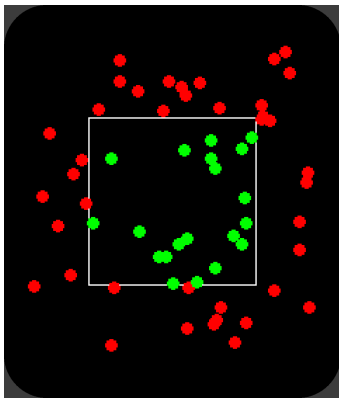
- `Serial.println` gibt `true` als 1 und `false` als 0 aus.
- Zeile 12: Jugendlich ist, wer mindestens 12 **und** unter 18 ist. Beide Vergleiche stimmen bei 15, also 1.
- Zeile 13: Wer unter 12 **oder** ab 67 ist, bekommt hier den Rabatt. Keiner der beiden Vergleiche stimmt, also 0.
- Zeile 16: `!` dreht das Ergebnis um.

Eine Bedingung wie `12 <= alter < 18` aus der Mathematik gibt es in C++ nicht in dieser Form. Du schreibst immer zwei Vergleiche und verknüpfst sie mit `&&`.

Liegt ein Punkt im Rechteck? Das ist eine der häufigsten Fragen in einem Spiel: Hat die Spielfigur die Münze berührt? Ist der Ball im Tor? Ein Punkt (x, y) liegt in einem Rechteck, wenn **vier** Dinge zugleich stimmen:

k08_im_rechteck.ino

```
17      tft.drawRect(60, 80, 120, 120, ST77XX_WHITE);
18      for (int i = 0; i < 60; i++) {
19          int x = random(20, 220);
20          int y = random(30, 250);
21          if (x >= 60 && x < 180 && y >= 80 && y < 200) {
22              tft.fillCircle(x, y, 4, ST77XX_GREEN);
23          } else {
24              tft.fillCircle(x, y, 4, ST77XX_RED);
25          }
26      }
```



Das Rechteck reicht von x = 60 bis 179 und von y = 80 bis 199. In Zeile 21 müssen alle vier Vergleiche stimmen: nicht zu weit links, nicht zu weit rechts, nicht zu weit oben, nicht zu weit unten. Dann wird der Punkt grün, sonst rot. Punkte, die auf dem Rand liegen, sehen halb drinnen aus; es zählt ihr Mittelpunkt.

> Für die Reihenfolge gilt: `!` vor `&&` vor `||`, so wie Punktrechnung vor Strichrechnung. Setze bei gemischten Bedingungen trotzdem Klammern, dann musst du nicht nachdenken: `(a && b) || c`.

8.3 Die Falle `=` statt `==`

Ein einzelnes `=` **schreibt** einen Wert in eine Variable. Zwei `==` **vergleichen**. Im Eifer passiert es jedem, dass in einer Bedingung nur eins steht. Das ist kein Fehler im Sinne des Compilers, der Sketch lässt sich kompilieren. Er tut aber etwas ganz anderes:

k08_falle.ino

```
 7     int leben = 3;
 8     leben = leben - 3;          // dreimal erwischt
 9
10     if (leben = 0) {
11         Serial.println("Game Over");
12     } else {
13         Serial.printf("Weiter, du hast noch %d Leben.\n", leben);
14     }
```

Serieller Monitor

```
Weiter, du hast noch 0 Leben.
```

Die Spielfigur hat 0 Leben, und trotzdem geht es weiter. Was ist passiert?

1. `leben = 0` in Zeile 10 schreibt 0 in `leben`. Die Variable war schon 0, aber das spielt keine Rolle: Hier wird nicht verglichen.
2. Eine Zuweisung hat selbst einen Wert, nämlich den geschriebenen: 0.
3. Für `if` gilt jede ganze Zahl außer 0 als wahr, 0 als falsch. Die Bedingung ist also **immer falsch**, egal, wie viele Leben übrig sind. Mit `if (leben = 1)` wäre sie immer wahr, und `leben` wäre danach 1.

Zum Glück bemerkt das auch der Compiler:

Ausgabe

```
k08_falle.ino: In function 'void setup()':
k08_falle.ino:10:15: warning: suggest parentheses around assignment used as truth
  value [-Wparentheses]
   10 |     if (leben = 0) {
      |         ~~~~~~^~~
```

Suggest parentheses around assignment used as truth value: Er schlägt Klammern vor um eine Zuweisung, die als Wahrheitswert benutzt wird. Gemeint ist: Wenn du das wirklich so willst, schreib `if ((leben = 0))`; dann weiß er, dass es Absicht ist. Fast immer ist es aber ein Tippfehler, und richtig ist `if (leben == 0)`.

> Diese Warnung siehst du nur, wenn unter **Datei → Einstellungen…** die **Compiler-Meldungen** auf **Alle** stehen (Kapitel 2.3). Mit der Voreinstellung bleibt die IDE stumm, und der Sketch tut stillschweigend das Falsche.

Noch eine Falle: das Semikolon hinter `if`. Ein `;` direkt hinter der Bedingung ist eine leere Anweisung. Das `if` gilt dann nur für dieses Nichts, und der Block darunter läuft **immer**:

k08_semikolon.ino

```
 7      int leben = 3;
 8
 9      if (leben == 0);
10      {
11          Serial.println("Game Over");
12      }
```

Serieller Monitor

```
Game Over
```

Drei Leben, und trotzdem „Game Over“. Auch hier warnt der Compiler:

Ausgabe

```
k08_semikolon.ino: In function 'void setup()':
k08_semikolon.ino:9:20: warning: suggest braces around empty body in an 'if'
  statement [-Wempty-body]
    9 |     if (leben == 0);
      |                    ^
```

Suggest braces around empty body: Er schlägt geschweifte Klammern um den leeren Rumpf vor. Die Abhilfe ist natürlich, das Semikolon zu löschen. Das gilt genauso für `while` und `for`: Kein Semikolon zwischen der Klammer mit der Bedingung und `{`.

> `=` schreibt, `==` vergleicht. Hinter `if (…)`, `while (…)` und `for (…)` kommt nie ein Semikolon, sondern `{`. Stell die Compiler-Meldungen auf **Alle**, dann findet der Compiler beide Fehler für dich.

Ausprobieren: Zahlen raten

Der PicoBoy denkt sich eine Zahl von 1 bis 100 aus, und du rätst im seriellen Monitor. Er sagt dir jedes Mal, ob dein Tipp zu klein oder zu groß war.

k08_raten.ino

```
 1 // Zahlen raten: Der PicoBoy denkt sich eine Zahl von 1 bis 100 aus.
 2 int geheim = 0;
 3 int versuche = 0;
 4
 5 void setup() {
 6     Serial.begin(115200);
 7     while (!Serial) {
 8         delay(10);
 9     }
10     randomSeed(micros());       // wann der Monitor kommt, ist nie gleich
```

```
11      geheim = random(1, 101);
12      Serial.println("Ich denke mir eine Zahl von 1 bis 100.");
13  }
14
15  void loop() {
16      Serial.println("Dein Tipp?");
17      while (Serial.available() == 0) {
18          delay(10);
19      }
20      int tipp = Serial.parseInt();
21      Serial.readStringUntil('\n');
22      versuche++;
23
24      if (tipp < 1 || tipp > 100) {
25          Serial.println("Bitte nur Zahlen von 1 bis 100.");
26      } else if (tipp < geheim) {
27          Serial.printf("%d ist zu klein.\n", tipp);
28      } else if (tipp > geheim) {
29          Serial.printf("%d ist zu gross.\n", tipp);
30      } else {
31          Serial.printf("Richtig, %d! Du hast %d Versuche gebraucht.\n", geheim,
              versuche);
32          geheim = random(1, 101);
33          versuche = 0;
34          Serial.println("Neue Runde: Ich denke mir eine neue Zahl.");
35      }
36  }
```

Serieller Monitor

```
Ich denke mir eine Zahl von 1 bis 100.
Dein Tipp?
50 ist zu gross.
Dein Tipp?
25 ist zu klein.
Dein Tipp?
37 ist zu gross.
Dein Tipp?
31 ist zu klein.
Dein Tipp?
Richtig, 34! Du hast 5 Versuche gebraucht.
Neue Runde: Ich denke mir eine neue Zahl.
Dein Tipp?
Bitte nur Zahlen von 1 bis 100.
Dein Tipp?
```

Eingegeben wurden `50`, `25`, `37`, `31`, `34` und zum Schluss `abc`.

- `geheim` und `versuche` stehen außerhalb der Funktionen (Zeilen 2 und 3), denn sie müssen über viele Durchläufe von `loop()` erhalten bleiben.
- **Zufall:** Zeile 10 startet den Zufall mit `micros()`, der Zeit seit dem Start in Mikrosekunden (millionstel Sekunden). Weil der Sketch vorher in der Schleife auf den seriellen Monitor wartet, hängt die Zahl davon ab, wann genau du den Monitor öffnest. Das ist auf die Mikrosekunde nie

gleich. `randomSeed(rp2040.hwrand32())` aus Kapitel 5.4 ginge genauso.

- Jeder Durchlauf von `loop()` ist ein Tipp: fragen, auf die Eingabe warten, Zahl lesen, Rest der Zeile wegwerfen (Kapitel 4.3).
- Die `if`-Kette ab Zeile 24 prüft zuerst, ob die Eingabe überhaupt zwischen 1 und 100 liegt. Bei Buchstaben liefert `parseInt` nämlich 0, und auch dann kommt die erste Meldung.
- Nach einem Treffer denkt sich der PicoBoy in Zeile 32 eine neue Zahl aus, und die nächste Runde beginnt.

Mit der richtigen Strategie brauchst du nie mehr als 7 Versuche: Rate immer genau die Mitte des Bereichs, der noch in Frage kommt. Jeder Tipp halbiert dann den Bereich: 100, 50, 25, 13, 7, 4, 2, 1.

Zum Verändern: Begrenze die Zahl der Versuche auf 7. Wer sie aufbraucht, verliert, und der Sketch verrät die Zahl.

Übungsaufgaben

Aufgabe 1 ⭐ Gerade oder ungerade Gib für die Zahlen von 1 bis 10 aus, ob sie gerade oder ungerade sind. Tipp: Eine gerade Zahl lässt beim Teilen durch 2 keinen Rest.

Aufgabe 2 ⭐ Was kommt heraus? Was gibt dieser Sketch aus? Und was, wenn `temperatur` 25 ist? Und bei 5?

k08_aufgabe2.ino

```
7     int temperatur = 18;
8
9     if (temperatur > 25) {
10        Serial.println("heiss");
11    } else if (temperatur > 15) {
12        Serial.println("angenehm");
13    } else if (temperatur > 5) {
14        Serial.println("kuehl");
15    } else {
16        Serial.println("kalt");
17    }
```

Aufgabe 3 ⭐⭐ Schaltjahr Lies im seriellen Monitor eine Jahreszahl ein und gib aus, ob es ein Schaltjahr ist. Die Regel: Ein Jahr ist ein Schaltjahr, wenn es durch 4 teilbar ist, aber nicht durch 100 – außer es ist durch 400 teilbar. 2024 und 2000 sind Schaltjahre, 1900 und 2025 nicht. Schreib alles in **eine** Bedingung.

Aufgabe 4 ⭐⭐ Hüpfender Ball Lass einen Ball über das Display fliegen, der an allen vier Rändern abprallt. Tipp: Speichere für x und y je eine Schrittweite `dx` und `dy`. Erreicht der Ball einen Rand, dreht sich das Vorzeichen um: `dx = -dx;`. Lösche nur den alten Ball (Kapitel 7.3).

Aufgabe 5 ⭐⭐⭐ **Würfelaugen** Zeichne die sechs Seiten eines Würfels mit ihren Augen, in einer `for`-Schleife von 1 bis 6. Versuche, mit möglichst wenigen Bedingungen auszukommen: Welche Augen haben 3 und 5 gemeinsam? Welche kommen ab der 2 immer vor, welche ab der 4?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `if (bedingung) { … } else if (…) { … } else { … }`: Die erste Bedingung, die stimmt, gewinnt; genau ein Block läuft.
- `switch (zahl) { case 1: … break; default: … }` für viele feste Werte. `break` nicht vergessen, sonst läuft der nächste `case` mit.
- Vergleiche liefern `true` oder `false`. Verknüpfen mit `&&` (und), `||` (oder), `!` (nicht).
- Punkt im Rechteck: `x >= links && x < rechts && y >= oben && y < unten`.
- `=` schreibt, `==` vergleicht. Kein `;` hinter `if (…)`. Mit Compiler-Meldungen **Alle** warnt der Compiler bei beiden Fallen.

KAPITEL 9

Tasten und Joystick

Worum geht's? Bisher laufen deine Sketches einfach ab. Jetzt hört der PicoBoy auf dich: Du liest den Joystick und die Tasten A und B, erkennst einen einzelnen Druck, bringst wackelnden Kontakten Ruhe bei und steuerst zum Schluss einen Punkt über den Bildschirm. Am Ende misst der PicoBoy deine Reaktionszeit.

9.1 `INPUT_PULLUP` und `digitalRead` – gedrückt ist `LOW`

Der PicoBoy hat sieben Tasten: den Joystick mit fünf Richtungen (hoch, runter, links, rechts und Mitte, wenn du ihn hineindrückst) und die Tasten A und B. Jede Taste hängt an einem eigenen **Pin** des Mikrocontrollers. Ein Pin ist ein Anschluss, über den der Chip mit der Außenwelt verbunden ist. Die Pins heißen GP0, GP1, GP2 und so weiter; im Sketch schreibst du nur die Nummer. Damit niemand Nummern auswendig lernen muss, bekommt jede Taste oben im Sketch einen Namen:

Taste	Pin	Name im Sketch
Joystick hoch	4	`TASTE_HOCH`
Joystick runter	2	`TASTE_RUNTER`
Joystick links	3	`TASTE_LINKS`
Joystick rechts	1	`TASTE_RECHTS`
Joystick Mitte	0	`TASTE_MITTE`
A	27	`TASTE_A`
B	28	`TASTE_B`

Ein Pin kann **Ausgang** sein (der Chip schaltet dort Strom ein und aus, so wie beim Hintergrundlicht an Pin 26) oder **Eingang** (der Chip misst, ob dort Spannung anliegt). Für eine Taste brauchst du einen Eingang. Das legst du in `setup()` mit `pinMode` fest:

k09_taste_lesen.ino

```
// Den Zustand der Taste A im seriellen Monitor anzeigen
#define TASTE_A 27

void setup() {
    Serial.begin(115200);
    pinMode(TASTE_A, INPUT_PULLUP);
}

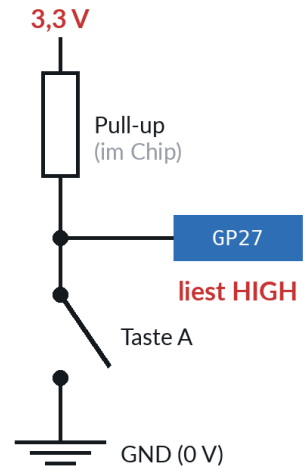
void loop() {
    int wert = digitalRead(TASTE_A);
    Serial.println(wert);
    delay(250);
}
```

```
Serieller Monitor
1
1
1
1
1
0
0
0
1
1
1
```

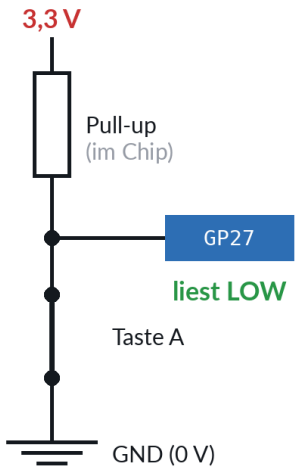
`pinMode(TASTE_A, INPUT_PULLUP);` macht Pin 27 zum Eingang und schaltet zusätzlich den eingebauten **Pull-up-Widerstand** ein. Was der tut, erklärt gleich die Abbildung.

`digitalRead(TASTE_A)` liest den Pin. Das Ergebnis ist `HIGH` (Spannung liegt an, Zahlenwert 1) oder `LOW` (keine Spannung, Zahlenwert 0). Der Sketch gibt es alle 250 ms aus. In der Ausgabe oben wurde A nach etwa einer Sekunde gedrückt und eine knappe Sekunde lang festgehalten: Solange die Taste gedrückt ist, liest der Sketch **0**, sonst **1**.

Das ist auf den ersten Blick verkehrt herum. Der Grund liegt in der Schaltung:



losgelassen: Pull-up zieht auf 3,3 V



gedrückt: Taste verbindet mit 0 V

- Die Taste verbindet den Pin beim Drücken mit **GND**, also mit 0 Volt. Der Pin liest dann `LOW`.
- Ist die Taste losgelassen, ist der Pin mit nichts verbunden. Ohne Hilfe würde er dann zufällig mal `HIGH`, mal `LOW` lesen, je nachdem, was in der Nähe gerade elektrisch stört. Man sagt: Der Eingang **hängt in der Luft**.
- Der **Pull-up-Widerstand** im Chip verbindet den Pin schwach mit 3,3 Volt und „zieht“ ihn so nach oben (englisch *pull up*). Losgelassen liest der Pin deshalb sicher `HIGH`. Drückst du, gewinnt die direkte Verbindung zu GND, und der Pin liest `LOW`.

> Tasten am PicoBoy: in `setup()` `pinMode(pin, INPUT_PULLUP);`, dann gilt **gedrückt = `LOW`**, losgelassen = `HIGH`. Die Frage „Ist A gedrückt?“ heißt im Sketch also `digitalRead(TASTE_A) == LOW`.

Mit einem `if` aus Kapitel 8 reagiert der Sketch auf die Taste. Hier leuchtet ein Kreis grün, **solange** A gedrückt ist, sonst rot:

k09_anzeige.ino

```
24 void loop() {
25     if (digitalRead(TASTE_A) == LOW) {
26         tft.fillCircle(120, 160, 50, ST77XX_GREEN);    // gedrueckt
27     } else {
28         tft.fillCircle(120, 160, 50, ST77XX_RED);      // nicht gedrueckt
29     }
30     delay(10);
31 }
```

Der Sketch zeichnet den Kreis bei jedem Durchlauf von `loop()` neu, viele Male pro Sekunde. Weil an derselben Stelle ein gleich großer Kreis entsteht, sieht man davon nichts; es ändert sich nur die Farbe. Den ganzen Sketch findest du als `k09_anzeige`.



Der Joystick besteht aus fünf solchen Tasten. Der nächste Sketch zeigt in großer Schrift, wohin du ihn drückst:

k09_joystick.ino

```
16 void setup() {
17     tft.init(240, 280);
18     pinMode(TFT_LICHT, OUTPUT);
19     digitalWrite(TFT_LICHT, HIGH);
20     tft.fillScreen(ST77XX_BLACK);
21     pinMode(TASTE_HOCH, INPUT_PULLUP);
22     pinMode(TASTE_RUNTER, INPUT_PULLUP);
23     pinMode(TASTE_LINKS, INPUT_PULLUP);
24     pinMode(TASTE_RECHTS, INPUT_PULLUP);
25     pinMode(TASTE_MITTE, INPUT_PULLUP);
26     tft.setTextSize(4);
27     tft.setTextColor(ST77XX_YELLOW, ST77XX_BLACK);  // Schrift, Hintergrund
28 }
29
30 void loop() {
31     tft.setCursor(48, 124);
32     if (digitalRead(TASTE_HOCH) == LOW) {
33         tft.print("hoch  ");
34     } else if (digitalRead(TASTE_RUNTER) == LOW) {
35         tft.print("runter");
36     } else if (digitalRead(TASTE_LINKS) == LOW) {
37         tft.print("links ");
```

```
38      } else if (digitalRead(TASTE_RECHTS) == LOW) {
39          tft.print("rechts");
40      } else if (digitalRead(TASTE_MITTE) == LOW) {
41          tft.print("Mitte ");
42      } else {
43          tft.print("      ");
44      }
45      delay(10);
46  }
```



- Jede der fünf Tasten braucht ihr eigenes `pinMode` (Zeilen 21 bis 25).
- Die `if`-`else if`-Kette fragt die Richtungen nacheinander ab. Die erste, die gedrückt ist, gewinnt.
- `tft.setTextColor(ST77XX_YELLOW, ST77XX_BLACK);` in Zeile 27 bekommt zwei Farben, Schrift und Hintergrund (Kapitel 3). Dann malt `print` auch das Kästchen hinter jedem Zeichen. Ein neuer Text überschreibt so den alten, ohne dass du vorher den Bildschirm löschen musst. Damit auch das Ende von „runter“ verschwindet, wenn danach „hoch“ kommt, sind alle Wörter mit Leerzeichen auf sechs Zeichen aufgefüllt. Das klappt nur mit der eingebauten Schrift, nicht mit den FreeFonts aus Kapitel 6.

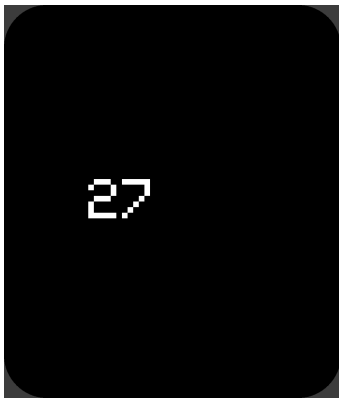
> Der Joystick ist ein Schalter, kein Messgerät: Jede Richtung ist nur gedrückt oder nicht. Drückst du schräg, können zwei Richtungen gleichzeitig gedrückt sein, zum Beispiel hoch und rechts. Der Sketch oben zeigt dann nur „hoch“, weil die `if`-Kette bei der ersten passenden Richtung aufhört.

9.2 Flanken erkennen: einmal pro Druck

Oft willst du nicht wissen, **ob** eine Taste gedrückt ist, sondern **dass** sie gerade gedrückt wurde: Bei jedem Druck soll ein Zähler um eins steigen. Der erste Versuch liegt nahe:

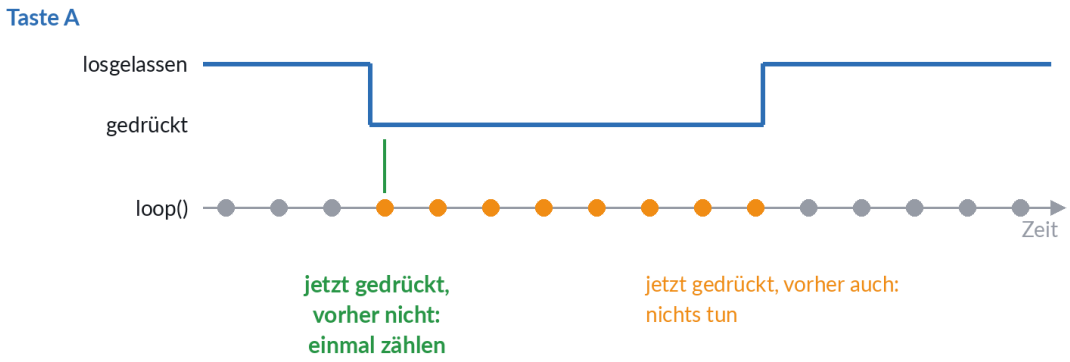
k09_zaehler_falsch.ino

```
24  void loop() {
25      if (digitalRead(TASTE_A) == LOW) {
26          zaehler = zaehler + 1;
27      }
28      tft.setCursor(60, 124);
29      tft.print(zaehler);
30      delay(10);
31  }
```



Ein einziger kurzer Druck, und der Zähler steht auf 28. Ein Druck mit dem Finger dauert mindestens eine Zehntelsekunde, meist länger. `loop()` läuft in dieser Zeit aber viele Male durch, hier etwa alle 10 ms. Bei jedem Durchlauf ist A noch gedrückt, und bei jedem Durchlauf zählt der Sketch.

Der Sketch muss sich also merken, wie die Taste **beim letzten Durchlauf** war. Gezählt wird nur, wenn sie **jetzt** gedrückt ist und **vorher** nicht. Diesen Moment, in dem sich der Zustand ändert, nennt man **Flanke**: So sieht der Übergang von oben nach unten aus, wenn man den Zustand der Taste über die Zeit aufzeichnet.



Jeder Punkt ist ein Durchlauf von loop(), in dem digitalRead die Taste liest.

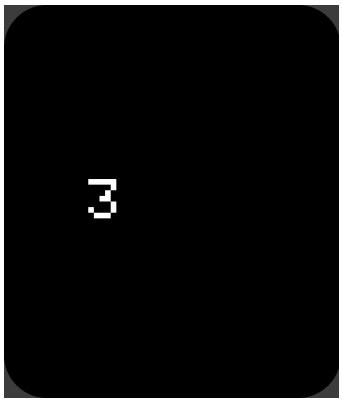
k09_flanke.ino

```
12 int zaehler = 0;
13 bool vorher = false;    // war A beim letzten Durchlauf gedrueckt?
14
15 void setup() {
16     Serial.begin(115200);
17     tft.init(240, 280);
18     pinMode(TFT_LICHT, OUTPUT);
19     digitalWrite(TFT_LICHT, HIGH);
20     tft.fillScreen(ST77XX_BLACK);
21     tft.setTextSize(4);
22     tft.setTextColor(ST77XX_WHITE, ST77XX_BLACK);
23     pinMode(TASTE_A, INPUT_PULLUP);
24 }
25
26 void loop() {
27     bool jetzt = digitalRead(TASTE_A) == LOW;
28     if (jetzt && !vorher) {             // eben heruntergedrueckt
29         zaehler = zaehler + 1;
30         Serial.printf("Druck Nr. %d\n", zaehler);
31     }
32     vorher = jetzt;
33     tft.setCursor(60, 124);
34     tft.print(zaehler);
35     delay(10);
36 }
```

Serieller Monitor

```
Druck Nr. 1
Druck Nr. 2
Druck Nr. 3
```

- Die Variable `vorher` in Zeile 13 steht **außerhalb** von `loop()`. So behält sie ihren Wert von einem Durchlauf zum nächsten.
- Zeile 27 legt das Ergebnis der Frage „Ist A gedrückt?“ in der `bool`-Variable `jetzt` ab: `true` oder `false`.
- Zeile 28: `jetzt && !vorher` heißt „jetzt gedrückt und vorher nicht gedrückt“. Das ist genau einmal pro Druck wahr.
- Zeile 32 merkt sich den Zustand für den nächsten Durchlauf. Diese Zeile vergisst man leicht. Ohne sie bleibt `vorher` immer `false`, und der Sketch zählt wieder bei jedem Durchlauf.



> **Flanke erkennen:** `jetzt` lesen, `if (jetzt && !vorher)` reagieren, dann `vorher = jetzt;`. Mit `if (!jetzt && vorher)` erkennst du umgekehrt das **Loslassen**.

Warten auf eine Taste. Manchmal soll ein Sketch einfach stehen bleiben, bis jemand drückt, etwa um eine Seite weiterzublättern. Dafür reicht eine `while`-Schleife ohne Inhalt:

k09_warten.ino

```
21 void loop() {
22     tft.fillScreen(ST77XX_BLUE);
23     tft.setTextColor(ST77XX_WHITE);
24     tft.setTextSize(4);
25     tft.setCursor(48, 110);
26     tft.print("Seite ");
27     tft.print(seite);
28     tft.setTextSize(2);
29     tft.setCursor(48, 200);
30     tft.print("weiter mit A");
31
32     while (digitalRead(TASTE_A) == HIGH) {
33         // warten, bis A gedrueckt wird
34     }
35     while (digitalRead(TASTE_A) == LOW) {
36         // warten, bis A wieder losgelassen ist
37     }
38     delay(20);          // Prellen abwarten (Abschnitt 9.3)
39
40     seite = seite + 1;
41     if (seite > 3) {
42         seite = 1;
43     }
44 }
```

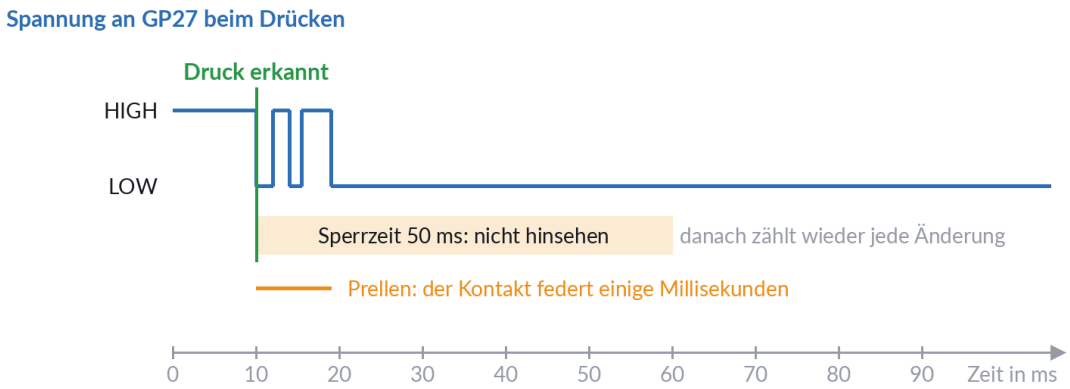
- Die erste Schleife (Zeile 32) läuft, solange A **nicht** gedrückt ist. Sie tut nichts außer immer wieder nachzusehen. Sobald du drückst, endet sie.
- Die zweite Schleife (Zeile 35) wartet, bis du A **wieder loslässt**. Ohne sie würde der nächste Durchlauf von `loop()` sofort weiterblättern, weil A ja noch gedrückt ist.
- Zeile 38 wartet noch 20 ms. Warum, zeigt der nächste Abschnitt.



Warten mit `while` ist bequem, hat aber denselben Haken wie `delay`: In der Zeit tut der Sketch nichts anderes. Soll sich nebenher etwas bewegen oder blinken, nimmst du die Flankenerkennung in `loop()`.

9.3 Entprellen

Eine Taste ist innen ein kleines Stück Metall, das beim Drücken auf einen Kontakt schnappt. Dabei federt es oft ein paarmal zurück, bevor es liegen bleibt. Für ein paar Millisekunden wechselt der Pin dann schnell zwischen `LOW` und `HIGH` hin und her. Das nennt man **Prellen**. Auch beim Loslassen kann eine Taste prellen.



Für dich ist das ein einziger Druck. Ein Sketch, der schnell genug nachsieht, erkennt aber jedes Zurückfedern als neue Flanke. Der folgende Sketch ist die Flankenerkennung von oben, nur ohne `delay` in `loop()` und mit der Uhrzeit jedes erkannten Drucks:

k09_prellen.ino

```
1 // Flankenerkennung ohne Pause: zaehlt eine prellende Taste mehrfach
2 #define TASTE_A 27
3
4 int zaehler = 0;
5 bool vorher = false;
6
7 void setup() {
8     Serial.begin(115200);
```

```
    pinMode(TASTE_A, INPUT_PULLUP);
}

void loop() {
    bool jetzt = digitalRead(TASTE_A) == LOW;
    if (jetzt && !vorher) {
        zaehler = zaehler + 1;
        Serial.printf("Druck Nr. %d bei %lu ms\n", zaehler, millis());
    }
    vorher = jetzt;
}
```

Serieller Monitor

```
Druck Nr. 1 bei 1000 ms
Druck Nr. 2 bei 1004 ms
Druck Nr. 3 bei 1009 ms
Druck Nr. 4 bei 1311 ms
Druck Nr. 5 bei 2000 ms
```

Die Ausgabe stammt aus einem Testlauf, in dem die Taste absichtlich geprellt hat: Sie wurde bei 1000 ms gedrückt, federte zweimal zurück und prellte auch beim Loslassen bei etwa 1310 ms. Bei 2000 ms kam ein sauberer zweiter Druck. Gedrückt hast du zweimal, gezählt hat der Sketch fünfmal. Wie stark eine Taste prellt, ist von Taste zu Taste verschieden und ändert sich mit dem Alter. Probiere den Sketch an deinem PicoBoy aus: Drück A einige Male schnell und auch einmal ganz langsam.

Die Lösung heißt **Entprellen**: Nachdem sich der Zustand der Taste geändert hat, sieht der Sketch eine kurze Zeit lang, die **Sperrzeit**, gar nicht hin. Das Prellen ist dann vorbei. 50 ms sind ein guter Wert: lang genug für fast jede Taste und so kurz, dass du es nicht merkst. Zum Messen der Zeit nimmst du `millis()` aus Kapitel 7:

k09_entprellen.ino

```
// Flankenerkennung mit Entprellen: nach jeder Aenderung 50 ms nicht hinsehen
#define TASTE_A 27
#define ENTPRELLZEIT 50     // in Millisekunden

int zaehler = 0;
bool vorher = false;
unsigned long letzte_aenderung = 0;

void setup() {
    Serial.begin(115200);
    pinMode(TASTE_A, INPUT_PULLUP);
}

void loop() {
    bool jetzt = digitalRead(TASTE_A) == LOW;
    if (jetzt != vorher && millis() - letzte_aenderung >= ENTPRELLZEIT) {
        letzte_aenderung = millis();
```

```
18          vorher = jetzt;
19          if (jetzt) {                    // die Aenderung war ein Druck
20              zaehler = zaehler + 1;
21              Serial.printf("Druck Nr. %d bei %lu ms\n", zaehler, millis());
22          }
23      }
24  }
```

Serieller Monitor

```
Druck Nr. 1 bei 1000 ms
Druck Nr. 2 bei 2000 ms
```

Derselbe Testlauf mit derselben prellenden Taste, und jetzt stimmt die Zahl.

- `letzte_aenderung` (Zeile 7) speichert, wann der Sketch zuletzt eine Änderung angenommen hat. Für Zeiten von `millis()` nimmt man immer `unsigned long` (Kapitel 5).
- Zeile 16 nimmt eine Änderung nur an, wenn **zwei** Dinge stimmen: Der Zustand ist anders als vorher, **und** seit der letzten angenommenen Änderung sind mindestens 50 ms vergangen.
- Erst dann wird `vorher` umgestellt (Zeile 18). Ist die Änderung ein Druck, zählt der Sketch (Zeilen 19 bis 22).
- Das Loslassen ist auch eine Änderung. Sie startet die Sperrzeit ebenfalls neu, so wird auch das Prellen beim Loslassen verschluckt.

Die Sketches in 9.1 und 9.2 haben ein `delay(10)` in `loop()`. Das ist bereits ein einfaches Entprellen: Der Sketch sieht nur alle 10 ms nach und verpasst so die meisten kurzen Zuckungen. Ganz sicher ist das nicht: Fällt ein Blick zufällig mitten ins Prellen, zählt er doch doppelt. Wenn es auf jeden Druck ankommt, etwa bei einem Zähler, nimm die Sperrzeit mit `millis()`.

Arduino hat **kein** eingebautes Entprellen: `digitalRead` liefert immer genau das, was gerade am Pin anliegt. Es gibt Bibliotheken dafür; in Kapitel 13 baust du dir selbst eine kleine.

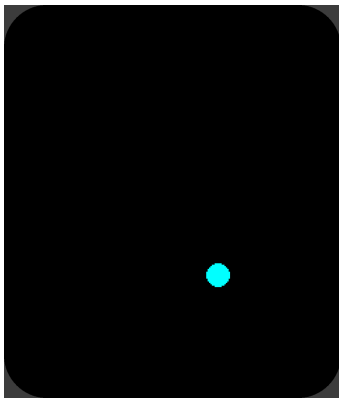
9.4 Ein Punkt, der sich steuern lässt

Jetzt kommt alles zusammen: Der Joystick bewegt einen Punkt, **solange** du ihn drückst. Ein Druck auf die Mitte wechselt die Farbe, **einmal pro Druck**.

k09_punkt.ino

```
17  int x = 120;
18  int y = 140;
19  uint16_t farbe = ST77XX_YELLOW;
20  bool mitte_vorher = false;
21
22  void setup() {
23      tft.init(240, 280);
24      pinMode(TFT_LICHT, OUTPUT);
25      digitalWrite(TFT_LICHT, HIGH);
```

```
26      tft.fillScreen(ST77XX_BLACK);
27      pinMode(TASTE_HOCH, INPUT_PULLUP);
28      pinMode(TASTE_RUNTER, INPUT_PULLUP);
29      pinMode(TASTE_LINKS, INPUT_PULLUP);
30      pinMode(TASTE_RECHTS, INPUT_PULLUP);
31      pinMode(TASTE_MITTE, INPUT_PULLUP);
32  }
33
34  void loop() {
35      tft.fillCircle(x, y, RADIUS, ST77XX_BLACK);     // alten Punkt loeschen
36
37      // solange gedrueckt: bewegen
38      if (digitalRead(TASTE_LINKS) == LOW && x > 30) {
39          x = x - 2;
40      }
41      if (digitalRead(TASTE_RECHTS) == LOW && x < 210) {
42          x = x + 2;
43      }
44      if (digitalRead(TASTE_HOCH) == LOW && y > 30) {
45          y = y - 2;
46      }
47      if (digitalRead(TASTE_RUNTER) == LOW && y < 250) {
48          y = y + 2;
49      }
50
51      // einmal pro Druck: Farbe wechseln
52      bool mitte = digitalRead(TASTE_MITTE) == LOW;
53      if (mitte && !mitte_vorher) {
54          if (farbe == ST77XX_YELLOW) {
55              farbe = ST77XX_CYAN;
56          } else {
57              farbe = ST77XX_YELLOW;
58          }
59      }
60      mitte_vorher = mitte;
61
62      tft.fillCircle(x, y, RADIUS, farbe);            // neuen Punkt zeichnen
63      delay(10);
64  }
```



- `x` und `y` (Zeilen 17 und 18) sind die Position des Punkts. Sie stehen außerhalb von `loop()`, damit sie erhalten bleiben.
- `farbe` hat den Typ `uint16_t`: eine ganze Zahl ohne Vorzeichen mit 16 Bit, genau das, was eine Farbe des Displays braucht (Kapitel 6).
- Jeder Durchlauf **löscht** zuerst den alten Punkt, indem er ihn in Schwarz übermalt (Zeile 35), ändert dann die Position und malt den Punkt an der neuen Stelle (Zeile 62).
- Die vier Richtungen werden **nicht** mit `else if` verbunden. So kann sich der Punkt auch schräg bewegen, wenn zwei Richtungen gleichzeitig gedrückt sind.

- Die zweite Bedingung, zum Beispiel `x > 30`, hält den Punkt vom Rand fern. An den runden Ecken wäre er sonst nicht mehr zu sehen.
- Die Mitte (Zeilen 52 bis 60) benutzt die Flankenerkennung aus 9.2. Probiere aus, was passiert, wenn du stattdessen nur `if (mitte)` schreibst und die Mitte gedrückt hältst.

Beim Bewegen flackert der Punkt ein wenig, denn zwischen Löschen und neu Malen ist er kurz verschwunden. Wie man das ganz vermeidet, lernst du in Kapitel 14.

Zum Verändern: Mach den Punkt schneller, indem er sich um 4 statt um 2 Pixel bewegt. Lass den Punkt eine Spur hinterlassen, indem du die Zeile zum Löschen weglässt.

Ausprobieren: Ein Reaktionstest

Drück A. Der Bildschirm wird rot. Irgendwann zwischen 2 und 5 Sekunden später wird er grün, und dann musst du so schnell wie möglich A drücken. Wer schon drückt, solange er rot ist, schummelt.



k09_reaktion.ino

```
20 void loop() {
21     tft.fillScreen(ST77XX_BLACK);
22     tft.setTextColor(ST77XX_WHITE);
23     tft.setTextSize(3);
24     tft.setCursor(57, 110);
25     tft.print("Bereit?");
26     tft.setTextSize(2);
27     tft.setCursor(54, 160);
28     tft.print("Start mit A");
29
30     while (digitalRead(TASTE_A) == HIGH) {
31     }
32     randomSeed(micros());   // Zeitpunkt deines Drucks
33     while (digitalRead(TASTE_A) == LOW) {
34     }
35     delay(20);
36
37     tft.fillScreen(ST77XX_RED);
38     tft.setTextSize(3);
39     tft.setCursor(48, 124);
40     tft.print("Warte...");
```

```
42      // 2 bis 5 Sekunden warten - wer vorher drueckt, schummelt
43      unsigned long wartezeit = random(2000, 5000);
44      unsigned long start = millis();
45      bool zu_frueh = false;
46      while (millis() - start < wartezeit && !zu_frueh) {
47          zu_frueh = digitalRead(TASTE_A) == LOW;
48      }
49
50      if (zu_frueh) {
51          tft.fillScreen(ST77XX_ORANGE);
52          tft.setCursor(39, 124);
53          tft.print("Zu frueh!");
54          Serial.println("Zu frueh!");
55      } else {
56          unsigned long signal = millis();    // die Uhr laeuft ab jetzt
57          tft.fillScreen(ST77XX_GREEN);
58          while (digitalRead(TASTE_A) == HIGH) {
59          }
60          unsigned long reaktion = millis() - signal;
61          tft.fillScreen(ST77XX_BLACK);
62          tft.setTextSize(4);
63          tft.setCursor(48, 124);
64          tft.print(reaktion);
65          tft.print(" ms");
66          Serial.printf("Reaktionszeit: %lu ms\n", reaktion);
67      }
68
69      while (digitalRead(TASTE_A) == LOW) {
70      }
71      delay(2000);
72  }
```

Serieller Monitor

```
Reaktionszeit: 268 ms
```



- Gewartet wird mit `while`-Schleifen wie in 9.2. Der Sketch hat in diesen Momenten nichts anderes zu tun.
- **Zufall:** `random(2000, 5000)` (Zeile 43) liefert eine Zahl von 2000 bis 4999. Ohne `randomSeed` wäre das bei jedem Start dieselbe Folge (Kapitel 5), und nach ein paar Runden wüsstest du, wann es grün wird. Zeile 32 startet den Zufall deshalb mit `micros()`, der Zeit in Mikrosekunden in dem Moment, in dem du zum ersten Mal drückst. Kein Mensch drückt auf die Mikrosekunde genau gleich.
- Die Warteschleife in Zeile 46 hat zwei Bedingungen: Sie läuft, solange die Wartezeit nicht um ist **und** noch niemand zu früh gedrückt hat.
- Zeile 56 merkt sich die Zeit, **bevor** der Bildschirm grün wird. Das Füllen des ganzen Bildschirms dauert etwa 43 ms; die oberen Zeilen sind aber sofort grün, und ab da läuft deine Reaktionszeit.
- Am Ende wartet der Sketch, bis A losgelassen ist, und zeigt das Ergebnis noch 2 Sekunden lang.

Zum Verändern: Gib bei „Zu frueh!“ einen tiefen Ton aus, sobald du Kapitel 10 gelesen hast. Merke dir die beste Zeit in einer Variable und zeige sie unter dem Ergebnis an.

Übungsaufgaben

Aufgabe 1 ⭐ Zwei Tasten Schreibe einen Sketch, der groß „A“ zeigt, solange A gedrückt ist, „B“, solange B gedrückt ist, und „A+B“, wenn beide gedrückt sind. Ist keine Taste gedrückt, bleibt der Bildschirm leer.

Aufgabe 2 ⭐ Hoch- und runterzählen Eine Zahl auf dem Bildschirm steigt bei jedem Druck auf Joystick-hoch um eins und fällt bei jedem Druck auf Joystick-runter. Die Mitte setzt sie auf 0 zurück. Was musst du tun, damit von „10“ kein Rest übrig bleibt, wenn die Zahl auf „9“ fällt?

Aufgabe 3 ⭐⭐ Wie lange gedrückt? Miss, wie lange A gedrückt war, und gib die Zeit nach dem Loslassen im seriellen Monitor aus, etwa `A war 350 ms gedrueckt`. Tipp: Merke dir bei der Flanke „gedrückt“ die Zeit mit `millis()` und rechne bei der Flanke „losgelassen“.

Aufgabe 4 ⭐⭐ Finde die Fehler Dieser Sketch soll jeden Druck auf A einmal zählen. Er übersetzt ohne Fehlermeldung, zählt aber trotzdem nicht richtig. Finde drei Fehler.

k09_aufgabe4.ino

```ino
// Soll zaehlen, wie oft A gedrueckt wurde. Was ist falsch?
#define TASTE_A 27

int zaehler = 0;
bool vorher = false;

void setup() {
    Serial.begin(115200);
    pinMode(TASTE_A, INPUT);
}

void loop() {
    bool jetzt = digitalRead(TASTE_A) == HIGH;
    if (jetzt && !vorher) {
        zaehler = zaehler + 1;
        Serial.println(zaehler);
    }
    delay(10);
}
```

Aufgabe 5 ⭐⭐⭐ Doppelklick Erkenne, ob A einmal oder zweimal schnell hintereinander gedrückt wurde, und gib `Einzelklick` oder `Doppelklick` aus. Als Doppelklick zählen zwei Drücke, zwischen denen höchstens 400 ms liegen. Überlege: Wann weiß der Sketch frühestens, dass es ein Einzelklick war?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Tasten: `pinMode(pin, INPUT_PULLUP);` in `setup()`. Der Pull-up hält den Eingang auf `HIGH`; die Taste zieht ihn auf `LOW`. **Gedrückt = `LOW`.**
- `digitalRead(pin) == LOW` ist wahr, **solange** die Taste gedrückt ist.
- **Einmal pro Druck:** Zustand in `jetzt` lesen, `if (jetzt && !vorher)`, danach `vorher = jetzt;`. `vorher` steht außerhalb von `loop()`.
- Tasten **prellen**: Ein Druck kann mehrere schnelle Wechsel erzeugen. **Entprellen** mit einer Sperrzeit: Änderungen erst wieder annehmen, wenn seit der letzten 50 ms vergangen sind (`millis()`).
- `while (digitalRead(pin) == HIGH) { }` wartet auf einen Druck; danach auf das Loslassen warten.
- `setTextColor(schrift, hintergrund)` überschreibt alten Text mit der eingebauten Schrift.

KAPITEL 10

LEDs, RGB-LED und Töne

Worum geht's? Der PicoBoy hat drei kleine Leuchtdioden in Rot, Gelb und Grün, eine RGB-LED, die jede Farbe mischen kann, und einen Lautsprecher. Du schaltest die LEDs ein und aus, dimmst sie, lässt die RGB-LED durch den Regenbogen laufen und spielst Töne und Melodien. Zum Schluss baust du eine Fußgängerampel mit Signalton.

10.1 `digitalWrite` und die drei LEDs

Eine **LED** (Leuchtdiode, englisch *light emitting diode*) leuchtet, wenn Strom durch sie fließt. Die drei LEDs des PicoBoy hängen an je einem Pin:

LED	Pin	Name im Sketch
rot	14	`LED_ROT`
gelb	13	`LED_GELB`
grün	12	`LED_GRUEN`

Damit der Chip an einem Pin Strom liefern kann, muss der Pin ein **Ausgang** sein. Das kennst du schon vom Hintergrundlicht aus Kapitel 3: `pinMode(pin, OUTPUT)` in `setup()`, danach schaltet `digitalWrite(pin, HIGH)` ein und `digitalWrite(pin, LOW)` aus. Der klassische erste Sketch jedes Arduino-Buchs ist eine blinkende LED:

k10_blinken.ino

```
// Die rote LED blinkt im Sekundentakt
#define LED_ROT 14

void setup() {
    pinMode(LED_ROT, OUTPUT);
}

void loop() {
    digitalWrite(LED_ROT, HIGH);    // an
    delay(500);
    digitalWrite(LED_ROT, LOW);     // aus
    delay(500);
}
```

Die LEDs des PicoBoy leuchten bei `HIGH` und sind bei `LOW` aus. Das ist genau umgekehrt wie bei den Tasten aus Kapitel 9 und nicht bei jedem Gerät so: Es hängt davon ab, wie die LED angeschlossen ist. Am PicoBoy ist sie zwischen Pin und GND geschaltet; `HIGH` legt 3,3 V an, und Strom fließt.

> Vergisst du `pinMode(pin, OUTPUT)`, bleibt der Pin ein Eingang. `digitalWrite` übersetzt dann ohne Fehlermeldung, aber die LED bleibt dunkel. Das ist der häufigste Grund, wenn „nichts passiert“.

Drei LEDs nacheinander ergeben ein Lauflicht. Die Wartezeit steht als `PAUSE` oben im Sketch, so änderst du das Tempo an einer einzigen Stelle:

k10_lauflicht.ino

```
 6
 7 void setup() {
 8     pinMode(LED_ROT, OUTPUT);
 9     pinMode(LED_GELB, OUTPUT);
10     pinMode(LED_GRUEN, OUTPUT);
11 }
12
13 void loop() {
14     digitalWrite(LED_ROT, HIGH);
15     delay(PAUSE);
16     digitalWrite(LED_ROT, LOW);
17     digitalWrite(LED_GELB, HIGH);
18     delay(PAUSE);
19     digitalWrite(LED_GELB, LOW);
20     digitalWrite(LED_GRUEN, HIGH);
21     delay(PAUSE);
22     digitalWrite(LED_GRUEN, LOW);
23 }
```

Blinken und gleichzeitig etwas anderes tun. `delay` hält den ganzen Sketch an. Soll die rote LED blinken und die grüne gleichzeitig auf Taste A reagieren, brauchst du den `millis`-Takt aus Kapitel 7:

k10_blinken_taste.ino

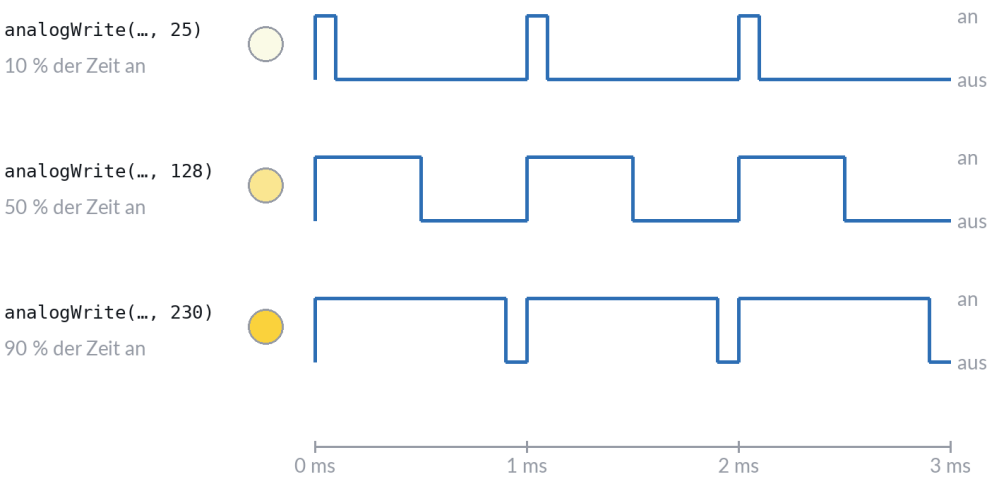
```
 6 bool rot_an = false;
 7 unsigned long letzter_wechsel = 0;
 8
 9 void setup() {
10     pinMode(LED_ROT, OUTPUT);
11     pinMode(LED_GRUEN, OUTPUT);
12     pinMode(TASTE_A, INPUT_PULLUP);
13 }
14
15 void loop() {
16     // Aufgabe 1: alle 500 ms die rote LED umschalten
17     if (millis() - letzter_wechsel >= 500) {
18         letzter_wechsel = millis();
19         rot_an = !rot_an;
20         if (rot_an) {
21             digitalWrite(LED_ROT, HIGH);
```

```
22          } else {
23              digitalWrite(LED_ROT, LOW);
24          }
25      }
26
27      // Aufgabe 2: die gruene LED folgt der Taste A
28      if (digitalRead(TASTE_A) == LOW) {
29          digitalWrite(LED_GRUEN, HIGH);
30      } else {
31          digitalWrite(LED_GRUEN, LOW);
32      }
33  }
```

- `rot_an` merkt sich, ob die rote LED gerade leuchtet. `!rot_an` in Zeile 19 dreht den Wert um: aus `true` wird `false` und umgekehrt.
- `loop()` läuft ohne jede Pause durch. Bei jedem Durchlauf prüft es: Ist es Zeit zum Umschalten? Ist A gedrückt? Beides geht so „gleichzeitig".

10.2 Helligkeit mit `analogWrite`

Ein Pin kennt nur zwei Zustände: an oder aus. Trotzdem kann eine LED halb hell leuchten. Der Trick: Der Chip schaltet die LED sehr schnell ein und aus, 1000-mal pro Sekunde. So schnell kann das Auge nicht folgen; es sieht nur den Durchschnitt. Ist die LED die Hälfte der Zeit an, wirkt sie halb so hell. Dieses Verfahren heißt **Pulsweitenmodulation**, kurz **PWM**: Die Breite (Weite) der Ein-Pulse bestimmt die Helligkeit.



Die Arduino-Funktion dafür heißt `analogWrite(pin, wert)`. Der Wert reicht von 0 (immer aus) bis 255 (immer an):

k10_helligkeit.ino

```
 3
 4 void setup() {
 5     Serial.begin(115200);
 6 }
 7
 8 void loop() {
 9     analogWrite(LED_GELB, 0);
10     Serial.println("0 von 255: aus");
11     delay(1000);
12     analogWrite(LED_GELB, 5);
13     Serial.println("5 von 255: ganz schwach");
14     delay(1000);
15     analogWrite(LED_GELB, 30);
16     Serial.println("30 von 255");
17     delay(1000);
18     analogWrite(LED_GELB, 100);
19     Serial.println("100 von 255");
20     delay(1000);
21     analogWrite(LED_GELB, 255);
```

Serieller Monitor

```
0 von 255: aus
5 von 255: ganz schwach
30 von 255
100 von 255
255 von 255: ganz hell
0 von 255: aus
```

- `analogWrite` braucht kein `pinMode`. Es stellt den Pin selbst auf PWM um.
- Der Name ist irreführend: „analog“ ist hier nichts. Der Pin schaltet weiterhin nur zwischen 0 und 3,3 Volt hin und her, nur eben sehr schnell.
- Unser Auge empfindet Helligkeit nicht gleichmäßig. Schon 5 von 255 sind deutlich zu sehen, während 100 und 255 sich weniger unterscheiden, als die Zahlen vermuten lassen.

Nachsehen statt raten. Woher weiß man, dass es 1000-mal pro Sekunde sind und dass der Bereich bis 255 geht? Bei Arduino ist das je nach Board verschieden, und Beschreibungen im Internet gelten oft für ein anderes Board. Die sicherste Quelle ist der Quelltext des Kerns selbst: Die Boardverwaltung der IDE hat ihn beim Installieren mitgebracht. Er liegt im Ordner `Arduino15` (Windows: `C:\Users\<Name>\AppData\Local\Arduino15`, macOS: `~/Library/Arduino15`, Linux: `~/.arduino15`), darin unter `packages/rp2040/hardware/rp2040/<Version>/cores/rp2040/`. Die Datei `wiring_analog.cpp` beginnt so:

```
static uint32_t analogScale = 255;
static uint32_t analogFreq = 1000;
```

Zwei Variablen mit den Startwerten: Bereich 255, Frequenz 1000 Hertz (Schwingungen pro Sekunde). Etwas weiter unten stehen die Funktionen, mit denen du beides änderst:

Funktion	bewirkt	erlaubt laut Quelltext
`analogWriteRange(bereich)`	größter Wert für `analogWrite`	3 bis 65535
`analogWriteResolution(bits)`	Bereich als Bitzahl: 8 → 255, 10 → 1023	2 bis 16
`analogWriteFreq(hertz)`	Frequenz des Ein-Aus-Takts	100 bis 10 000 000

Aus dem Quelltext erfährst du auch, was bei falschen Werten passiert: Werte unter 0 macht `analogWrite` zu 0, Werte über dem Bereich zu „immer an“. Mit `analogWriteRange(100)` werden die Werte zu Prozent:

k10_prozent.ino

```
 5
 6 void setup() {
 7     analogWriteRange(100);      // ab jetzt: 0 bis 100
 8     analogWrite(LED_ROT, 10);   // 10 Prozent der Zeit an
 9     analogWrite(LED_GELB, 50);
10     analogWrite(LED_GRUEN, 100);
11 }
12
13 void loop() {
14 }
```

Langsam hell und dunkel. Mit zwei `for`-Schleifen aus Kapitel 7 „atmet“ die grüne LED:

k10_atmen.ino

```
 4 void setup() {
 5 }
 6
 7 void loop() {
 8     for (int wert = 0; wert <= 255; wert++) {
 9         analogWrite(LED_GRUEN, wert);
10         delay(4);
11     }
12     for (int wert = 255; wert >= 0; wert--) {
13         analogWrite(LED_GRUEN, wert);
14         delay(4);
15     }
16     delay(300);
17 }
```

Jede Schleife hat 256 Schritte zu je 4 ms, ein Atemzug dauert also gut zwei Sekunden.

> Auch das Hintergrundlicht des Displays an Pin 26 lässt sich so dimmen. Kapitel 21 zeigt das genauer.

10.3 Die RGB-LED mit Adafruit NeoPixel

Die vierte LED des PicoBoy ist eine **RGB-LED**: In einem Gehäuse sitzen eine rote, eine grüne und eine blaue LED dicht nebeneinander. Leuchten alle drei, mischen sie sich für das Auge zu einer Farbe, genau wie die Pixel des Displays. Dazu sitzt in der LED ein winziger Chip. Er bekommt die Farbe über **eine** Leitung (Pin 11) als Folge schneller Pulse gesagt und hält sie dann, bis eine neue Farbe kommt. LEDs dieser Art heißen bei der Firma Adafruit **NeoPixel**, und die Bibliothek **Adafruit NeoPixel** aus Kapitel 2 erzeugt die Pulse für dich.

k10_rgb_farben.ino

```
// Die RGB-LED zeigt nacheinander fuenf Farben
#include <Adafruit_NeoPixel.h>

#define RGB_LED 11

Adafruit_NeoPixel rgb(1, RGB_LED, NEO_GRB + NEO_KHZ800);

void setup() {
    Serial.begin(115200);
    rgb.begin();
    rgb.setBrightness(40);      // 40 von 255: hell genug, blendet nicht
}

void loop() {
    rgb.setPixelColor(0, rgb.Color(255, 0, 0));
    rgb.show();
    Serial.println("rot");
    delay(1000);
    rgb.setPixelColor(0, rgb.Color(0, 255, 0));
    rgb.show();
    Serial.println("gruen");
    delay(1000);
    rgb.setPixelColor(0, rgb.Color(0, 0, 255));
    rgb.show();
    Serial.println("blau");
    delay(1000);
    rgb.setPixelColor(0, rgb.Color(255, 160, 0));
    rgb.show();
    Serial.println("orange");
    delay(1000);
    rgb.setPixelColor(0, rgb.Color(255, 255, 255));
    rgb.show();
    Serial.println("weiss");
    delay(1000);
}
```

Serieller Monitor

```
rot
gruen
blau
orange
weiss
rot
```

Zeile 6 legt das Objekt `rgb` an. Die drei Angaben in den Klammern:

Angabe	Bedeutung
`1`	wie viele NeoPixel an der Leitung hängen; am PicoBoy einer
`RGB_LED`	der Pin, Nummer 11
`NEO_GRB + NEO_KHZ800`	die Bauart: Farben in der Reihenfolge Grün, Rot, Blau, mit 800 000 Bit pro Sekunde

Die Bauart musst du nicht verstehen, aber genau so schreiben. Mit einer falschen Reihenfolge, etwa `NEO_RGB`, wären Rot und Grün vertauscht.

- **`rgb.begin()`** in `setup()` bereitet den Pin vor.
- **`rgb.setBrightness(40)`** stellt die Helligkeit für alles Folgende ein: 0 ist aus, 255 volle Kraft. Die LED ist sehr hell; 40 ist angenehm.
- **`rgb.Color(rot, gruen, blau)`** mischt eine Farbe aus drei Anteilen von 0 bis 255 und packt sie in eine einzige Zahl.
- **`rgb.setPixelColor(0, farbe)`** gibt dem NeoPixel Nummer 0 eine neue Farbe. Gezählt wird ab 0, der erste und einzige NeoPixel hat also die Nummer 0.
- **`rgb.show()`** schickt die Farbe an die LED. **Erst jetzt** ändert sie sich. Ohne `show()` bleibt sie, wie sie war.

> RGB-LED: einmal `rgb.begin()` und `rgb.setBrightness(…)` in `setup()`, dann für jede Farbe `rgb.setPixelColor(0, rgb.Color(r, g, b));` und **`rgb.show();`**.

> `setBrightness` ist für den Aufruf **einmal** in `setup()` gedacht, nicht für Helligkeitsänderungen im laufenden Sketch. Die Bibliothek rechnet die Helligkeit nämlich sofort in die gespeicherten Farben ein; dabei gehen Stufen verloren, und nach mehrmaligem Ändern stimmen die Farben nicht mehr genau. So steht es in der Beschreibung der Funktion im Quelltext der Bibliothek. Willst du die LED dunkler machen, nimm kleinere Farbwerte, zum Beispiel `rgb.Color(100, 0, 0)` statt `rgb.Color(255, 0, 0)`.

Ein Regenbogen. Durch alle Farben zu laufen ist mit Rot-, Grün- und Blauanteilen umständlich. Einfacher ist der **Farbton** (englisch *hue*): eine einzige Zahl, die einmal rund um den Farbkreis führt, von Rot über Gelb, Grün, Blau und Lila zurück zu Rot. `rgb.ColorHSV(farbton)` macht daraus eine Farbe. Der Farbton reicht von 0 bis 65535:

Farbton	0	10923	21845	32768	43691	54613	65535
Farbe	Rot	Gelb	Grün	Cyan	Blau	Magenta	wieder fast Rot

k10_regenbogen.ino

```
1  // Die RGB-LED laeuft langsam durch alle Farben des Regenbogens
2  #include <Adafruit_NeoPixel.h>
3
4  #define RGB_LED 11
5
6  Adafruit_NeoPixel rgb(1, RGB_LED, NEO_GRB + NEO_KHZ800);
7
8  uint16_t farbton = 0;       // 0 bis 65535: einmal rund um den Farbkreis
9
10 void setup() {
11     rgb.begin();
12     rgb.setBrightness(40);
13 }
14
15 void loop() {
16     uint32_t farbe = rgb.gamma32(rgb.ColorHSV(farbton));
17     rgb.setPixelColor(0, farbe);
18     rgb.show();
19     farbton = farbton + 256;    // nach 256 Schritten ist der Kreis rund
20     delay(20);
21 }
```

- `farbton` (Zeile 8) hat den Typ `uint16_t`, eine ganze Zahl von 0 bis 65535. Zeile 19 erhöht ihn bei jedem Durchlauf um 256. Nach 65280 kommt rechnerisch 65536, das passt nicht hinein: Der Wert **läuft über** und fängt wieder bei 0 an (Kapitel 5). Sonst ein Fehler, hier genau richtig, denn der Farbkreis ist ja rund.
- `uint32_t` in Zeile 16 ist eine ganze Zahl mit 32 Bit. Darin passen die drei Farbanteile, die `ColorHSV` und `Color` liefern.
- **`rgb.gamma32(…)`** gleicht aus, dass das Auge Helligkeit ungleichmäßig empfindet (wie in 10.2). Ohne diese Korrektur wirken die Farben laut Beschreibung der Bibliothek blass. Probiere es aus: Lass `gamma32` weg und vergleiche.
- Ein Umlauf dauert 256 Schritte zu gut 20 ms, also etwa 5 Sekunden.

`ColorHSV` kann noch mehr: `rgb.ColorHSV(farbton, saettigung, helligkeit)` mit zwei weiteren Werten von 0 bis 255. Eine kleine Sättigung macht die Farbe blasser (0 ist Weiß), eine kleine Helligkeit dunkler. Ohne Angabe sind beide 255.

10.4 `tone` und `noTone`: Töne und Melodien

Der PicoBoy hat an Pin 15 einen kleinen Lautsprecher. `tone(pin, frequenz, dauer)` spielt dort einen Ton. Die **Frequenz** in Hertz gibt an, wie oft pro Sekunde die Membran des Lautsprechers schwingt: Je höher die Zahl, desto höher der Ton. Der Ton a, nach dem Orchester stimmen, hat 440 Hertz. Die **Dauer** ist in Millisekunden.

Das Wichtigste an `tone`: **Es wartet nicht.** Der Ton wird angestoßen, und der Sketch läuft sofort weiter, während der Ton noch klingt:

k10_ton.ino

```
 1 // tone wartet nicht: das Programm laeuft sofort weiter
 2 #define LAUTSPRECHER 15
 3
 4 void setup() {
 5     Serial.begin(115200);
 6     while (!Serial) {
 7         delay(10);      // warten, bis der serielle Monitor offen ist
 8     }
 9     delay(500);
10     Serial.printf("%lu ms: tone fuer 1000 ms\n", millis());
11     tone(LAUTSPRECHER, 440, 1000);
12     Serial.printf("%lu ms: tone ist schon zurueck\n", millis());
13     delay(1500);
14     Serial.printf("%lu ms: jetzt ein Ton ohne Dauer\n", millis());
15     tone(LAUTSPRECHER, 660);
16     delay(500);
17     noTone(LAUTSPRECHER);
18     Serial.printf("%lu ms: noTone\n", millis());
19 }
20
21 void loop() {
22 }
```

Serieller Monitor

```
500 ms: tone fuer 1000 ms
500 ms: tone ist schon zurueck
2000 ms: jetzt ein Ton ohne Dauer
2500 ms: noTone
```

Beide Zeilen um `tone` (Zeilen 10 bis 12) tragen dieselbe Zeit: `tone` kehrt sofort zurück. Danach klingt der Ton eine Sekunde lang, während der Sketch schon in `delay(1500)` wartet. In `wiring_analog.cpp` hast du gesehen, wie man im Kern nachliest; bei `tone` steht es in `Tone.cpp`: Die Funktion stellt einen Zeitgeber, der den Ton nach der Dauer abschaltet, und kehrt zurück.

Ohne Dauer (Zeile 15) klingt der Ton so lange, bis `noTone(pin)` ihn abschaltet. `while (!Serial)` am Anfang kennst du aus Kapitel 4: Der Sketch wartet, bis der serielle Monitor zuhört.

Zwei Töne direkt hintereinander: Hier hörst du nur den zweiten.

k10_ton_falle.ino

```
4 void setup() {
5     tone(LAUTSPRECHER, 440, 500);
6     tone(LAUTSPRECHER, 660, 500);   // ersetzt den ersten Ton sofort
7 }
```

Der zweite `tone`-Aufruf kommt Mikrosekunden nach dem ersten und ersetzt ihn sofort, denn ein Pin spielt immer nur einen Ton. Zwischen zwei Töne gehört deshalb ein `delay` (oder ein `millis`-Takt), der so lang ist wie der Ton.

Eine Melodie ist also eine Folge von `tone` und `delay`. Für die Noten legst du mit `#define` Namen für die Frequenzen fest:

k10_melodie.ino

```
1  // Spielt den Anfang von "Alle meine Entchen"
2  #define LAUTSPRECHER 15
3
4  // Frequenzen in Hertz (gerundet)
5  #define TON_C 262
6  #define TON_D 294
7  #define TON_E 330
8  #define TON_F 349
9  #define TON_G 392
10 #define TON_A 440
11
12 #define ACHTEL 250      // Dauer einer Achtelnote in ms
13
14 void setup() {
15     tone(LAUTSPRECHER, TON_C, ACHTEL - 30);
16     delay(ACHTEL);
17     tone(LAUTSPRECHER, TON_D, ACHTEL - 30);
18     delay(ACHTEL);
19     tone(LAUTSPRECHER, TON_E, ACHTEL - 30);
20     delay(ACHTEL);
21     tone(LAUTSPRECHER, TON_F, ACHTEL - 30);
22     delay(ACHTEL);
23     tone(LAUTSPRECHER, TON_G, 2 * ACHTEL - 30);
24     delay(2 * ACHTEL);
25     tone(LAUTSPRECHER, TON_G, 2 * ACHTEL - 30);
26     delay(2 * ACHTEL);
27     tone(LAUTSPRECHER, TON_A, ACHTEL - 30);
28     delay(ACHTEL);
29     tone(LAUTSPRECHER, TON_A, ACHTEL - 30);
30     delay(ACHTEL);
31     tone(LAUTSPRECHER, TON_A, ACHTEL - 30);
32     delay(ACHTEL);
33     tone(LAUTSPRECHER, TON_A, ACHTEL - 30);
34     delay(ACHTEL);
```

```
    tone(LAUTSPRECHER, TON_G, 4 * ACHTEL - 30);
    delay(4 * ACHTEL);
}
```

- Eine Achtelnote dauert hier 250 ms, eine Viertel doppelt so lang.
- Jeder Ton ist 30 ms **kürzer** als das `delay` danach. So entsteht eine winzige Pause, und zwei gleiche Töne hintereinander (das doppelte „g“, die vier „a“) klingen getrennt statt wie ein langer Ton.
- Die Melodie steht in `setup()` und spielt deshalb einmal. In `loop()` würde sie sich endlos wiederholen.

Das sind viele fast gleiche Zeilen. In Kapitel 12 schreibst du die Noten in eine Liste und spielst sie mit einer Schleife ab.

Ton	c’	d’	e’	f’	g’	a’	h’	c’’
Hertz (gerundet)	262	294	330	349	392	440	494	523

Eine Oktave höher ist immer die doppelte Frequenz: c’’ hat 523 Hz, das doppelte von c’.

Ein Ton, solange eine Taste gedrückt ist, braucht die beiden Flanken aus Kapitel 9: Beim Drücken startet `tone` ohne Dauer, beim Loslassen stoppt `noTone`:

k10_dauerton.ino

```
bool vorher = false;

void setup() {
    pinMode(TASTE_A, INPUT_PULLUP);
}

void loop() {
    bool jetzt = digitalRead(TASTE_A) == LOW;
    if (jetzt && !vorher) {          // gerade gedrueckt
        tone(LAUTSPRECHER, 523);     // ohne Dauer: klingt bis noTone
    }
    if (!jetzt && vorher) {          // gerade losgelassen
        noTone(LAUTSPRECHER);
    }
    vorher = jetzt;
    delay(10);
}
```

Würdest du `tone` bei jedem Durchlauf aufrufen, solange A gedrückt ist, finge der Ton hundertmal pro Sekunde neu an. Je nach Frequenz kann man das als Knacken oder Schnarren hören.

Willst du Ruhe beim Ausprobieren, lass `tone` weg oder ersetze es durch eine Ausgabe im seriellen Monitor. Die Lautstärke selbst lässt sich mit `tone` nicht einstellen.

Ausprobieren: Eine Fußgängerampel

Die drei LEDs sind die Ampel für die Autos, die RGB-LED ist die Ampel für die Fußgänger. Drückst du A, bestätigt ein Piepton den Wunsch, und kurz darauf schaltet die Autoampel über Gelb auf Rot. Die Fußgänger bekommen Grün, und ein Tick-Ton hilft Menschen, die nicht gut sehen.

k10_ampel.ino

```
33 void loop() {
34     // Autos fahren, Fussgaenger warten
35     digitalWrite(LED_GRUEN, HIGH);
36     rgb.setPixelColor(0, rgb.Color(255, 0, 0));
37     rgb.show();
38     tft.fillScreen(ST77XX_BLACK);
39     tft.setCursor(39, 124);
40     tft.print("Druecke A");
41
42     while (digitalRead(TASTE_A) == HIGH) {
43     }
44     tone(LAUTSPRECHER, 880, 100);           // Quittung
45     tft.fillScreen(ST77XX_BLACK);
46     tft.setCursor(66, 104);
47     tft.print("Signal");
48     tft.setCursor(75, 144);
49     tft.print("kommt");
50     delay(2000);
51
52     // Autos: gruen -> gelb -> rot
53     digitalWrite(LED_GRUEN, LOW);
54     digitalWrite(LED_GELB, HIGH);
55     delay(1500);
56     digitalWrite(LED_GELB, LOW);
57     digitalWrite(LED_ROT, HIGH);
58     delay(1000);
59
60     // Fussgaenger gehen, die Ampel tickt
61     rgb.setPixelColor(0, rgb.Color(0, 255, 0));
62     rgb.show();
63     tft.fillScreen(ST77XX_GREEN);
64     tft.setTextColor(ST77XX_BLACK);
65     tft.setCursor(75, 124);
66     tft.print("Gehen");
67     for (int i = 0; i < 10; i++) {
68         tone(LAUTSPRECHER, 1200, 30);
69         delay(500);
70     }
71     tft.setTextColor(ST77XX_WHITE);
72
73     // Fussgaenger rot, Autos: rot -> rot+gelb -> gruen
74     rgb.setPixelColor(0, rgb.Color(255, 0, 0));
75     rgb.show();
76     tft.fillScreen(ST77XX_BLACK);
77     delay(1500);
```



```
78      digitalWrite(LED_GELB, HIGH);
79      delay(1000);
80      digitalWrite(LED_ROT, LOW);
81      digitalWrite(LED_GELB, LOW);
82  }
```

- Der Ablauf steht einfach Schritt für Schritt in `loop()`, mit `delay` dazwischen. Das ist hier in Ordnung, denn außer der Taste A am Anfang muss der Sketch auf nichts reagieren.
- Zeile 44 spielt den Quittungston mit Dauer: Er klingt 100 ms, während der Sketch schon den Bildschirm beschreibt.
- Die Schleife in Zeile 67 tickt zehnmal: ein Ton von 30 ms, dann eine halbe Sekunde Pause. Grün dauert also 5 Sekunden.
- Am Ende zeigt die Autoampel kurz Rot und Gelb gleichzeitig, dann beginnt `loop()` von vorn mit Grün.

Zum Verändern: Lass die Fußgängerampel vor dem Ende grün blinken. Zeige auf dem Bildschirm einen Countdown, wie viele Sekunden noch Grün sind.

Übungsaufgaben

Aufgabe 1 ⭐ **SOS** Lass die rote LED das Notsignal SOS blinken: dreimal kurz, dreimal lang, dreimal kurz, dann eine Pause von zwei Sekunden. Kurz ist 200 ms, lang 600 ms.

Aufgabe 2 ⭐ **Blaulicht** Lass die RGB-LED wie ein Blaulicht blitzen: zweimal kurz blau, dann eine längere Pause.

Aufgabe 3 ⭐⭐ **Dimmer** Mit Joystick-hoch wird die gelbe LED um eine Stufe heller, mit Joystick-runter dunkler, einmal pro Druck. Es gibt die Stufen 0, 25, 50 … 250. Zeige den Wert groß auf dem Bildschirm.

Aufgabe 4 ⭐⭐ **Sirene** Lass den Ton gleichmäßig von 500 Hz auf 1000 Hz steigen und wieder fallen, immer wieder. Tipp: eine `for`-Schleife, die die Frequenz in Schritten von 10 Hz ändert, und `tone` ohne Dauer.

Aufgabe 5 ⭐⭐⭐ **Ein Klavier** Die sieben Tasten spielen die Töne c', d', e', f', g', a' und h', solange sie gedrückt sind. Die RGB-LED leuchtet dabei für jeden Ton in einer anderen Farbe. Ist keine Taste gedrückt, ist es still und die LED aus. Tipp: Bestimme zuerst die Frequenz der gedrückten Taste (0, wenn keine gedrückt ist), und reagiere nur, wenn sie sich seit dem letzten Durchlauf geändert hat.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- LEDs: `pinMode(pin, OUTPUT);` in `setup()`, dann `digitalWrite(pin, HIGH)` (an) und `digitalWrite(pin, LOW)` (aus). Rot 14, Gelb 13, Grün 12.
- `analogWrite(pin, wert)` dimmt mit **PWM**: schnelles Ein und Aus, 1000-mal pro Sekunde, Werte 0 bis 255. Ändern mit `analogWriteRange`, `analogWriteResolution`, `analogWriteFreq`. Genaues steht im Quelltext des Kerns (`wiring_analog.cpp`).
- RGB-LED: `Adafruit_NeoPixel rgb(1, RGB_LED, NEO_GRB + NEO_KHZ800);`, `rgb.begin()`, `rgb.setBrightness(…)` einmal, dann `rgb.setPixelColor(0, …)` und **`rgb.show()`**.
- Farben: `rgb.Color(r, g, b)` oder `rgb.ColorHSV(farbton)` mit Farbton 0 bis 65535; `rgb.gamma32(…)` für natürlich wirkende Farben.
- `tone(LAUTSPRECHER, frequenz, dauer)` **wartet nicht**. Ohne Dauer klingt der Ton bis `noTone(LAUTSPRECHER)`. Ein neuer Ton ersetzt den alten.
- Melodien: `tone` und danach `delay` in der Länge des Tons.

KAPITEL 11

Funktionen

Worum geht's? Deine Sketches werden länger, und manche Zeilen stehen immer wieder fast gleich da. Mit eigenen **Funktionen** gibst du einem Stück Sketch einen Namen und rufst es auf, so oft du willst: mit verschiedenen Werten und, wenn nötig, mit einem Ergebnis. Du lernst, wo Variablen gelten, und baust zum Schluss einen Würfel.

11.1 Eigene Funktionen schreiben

Funktionen benutzt du seit Kapitel 3: `setup` und `loop` sind Funktionen, die du schreibst, `delay`, `digitalRead` oder `tft.fillCircle` sind Funktionen, die andere geschrieben haben. Eine **Funktion** ist ein Block von Anweisungen mit einem Namen. Mit dem Namen und runden Klammern **rufst** du sie **auf**: Der Sketch springt in die Funktion, führt ihre Anweisungen aus und macht danach hinter dem Aufruf weiter.

Eine eigene Funktion schreibst du genauso wie `setup`:

k11_erste_funktion.ino

```
 1 // Eine eigene Funktion schreiben und zweimal aufrufen
 2 void setup() {
 3     Serial.begin(115200);
 4     while (!Serial) {
 5         delay(10);
 6     }
 7     trennlinie();
 8     Serial.println("Willkommen beim PicoBoy!");
 9     trennlinie();
10 }
11
12 void loop() {
13 }
14
15 // gibt eine Linie aus Strichen aus
16 void trennlinie() {
17     Serial.println("------------------------");
18 }
```

Serieller Monitor

```
------------------------
Willkommen beim PicoBoy!
------------------------
```

- Die Zeilen 16 bis 18 **definieren** die Funktion `trennlinie`: Sie legen fest, was die Funktion tut. Dabei passiert noch nichts.
- Die Zeilen 7 und 9 **rufen** sie auf. Erst dann wird die Linie ausgegeben, und zwar bei jedem Aufruf.

- `void` vor dem Namen heißt „leer“: Die Funktion liefert kein Ergebnis zurück. Was es mit einem Ergebnis auf sich hat, zeigt 11.3.
- Die runden Klammern hinter dem Namen sind leer: Die Funktion braucht keine Angaben. Hinschreiben musst du sie trotzdem, bei der Definition und beim Aufruf.

Für Namen eigener Funktionen gelten dieselben Regeln wie für Variablen: Buchstaben, Ziffern und `_`, keine Umlaute, nicht mit einer Ziffer beginnen. In diesem Buch sind sie klein und beschreiben, was die Funktion **tut**: `trennlinie`, `zeichne_smiley`, `wuerfeln`.

Warum das Ganze? Funktionen haben drei Vorteile:

1. **Kein doppelter Code.** Was mehrmals vorkommt, steht nur einmal da. Willst du die Linie ändern, änderst du eine einzige Zeile.
2. **Lesbarkeit.** `zeichne_smiley(70, 80, ST77XX_YELLOW)` sagt sofort, was passiert. Sechs Zeilen mit `fillCircle` und `fillRect` muss man erst entziffern.
3. **Ordnung.** `loop()` wird kurz und zeigt den Ablauf; die Einzelheiten stehen in den Funktionen darunter.

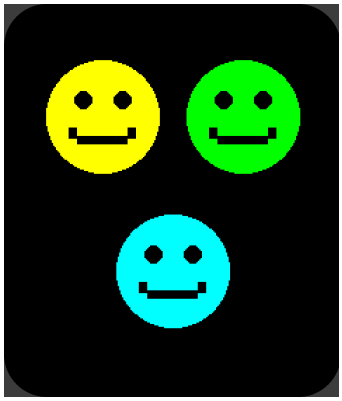
11.2 Parameter: Werte an die Funktion übergeben

Eine Funktion, die immer genau dasselbe tut, ist nur begrenzt nützlich. Ein Smiley soll an verschiedenen Stellen und in verschiedenen Farben erscheinen. Die Werte, die sich ändern, gibst du der Funktion beim Aufruf mit:

k11_smileys.ino

```
11 void setup() {
12     tft.init(240, 280);
13     tft.fillScreen(ST77XX_BLACK);
14     pinMode(TFT_LICHT, OUTPUT);
15     digitalWrite(TFT_LICHT, HIGH);
16
17     zeichne_smiley(70, 80, ST77XX_YELLOW);
18     zeichne_smiley(170, 80, ST77XX_GREEN);
19     zeichne_smiley(120, 190, ST77XX_CYAN);
20 }
21
22 void loop() {
23 }
24
25 // Smiley mit Mittelpunkt (x, y) und Radius 40
26 void zeichne_smiley(int x, int y, uint16_t farbe) {
27     tft.fillCircle(x, y, 40, farbe);
28     tft.fillCircle(x - 14, y - 12, 6, ST77XX_BLACK);    // Augen
29     tft.fillCircle(x + 14, y - 12, 6, ST77XX_BLACK);
30     tft.fillRect(x - 18, y + 14, 36, 6, ST77XX_BLACK);  // Mund
```

```
31      tft.fillRect(x - 24, y + 8, 6, 8, ST77XX_BLACK);
32      tft.fillRect(x + 18, y + 8, 6, 8, ST77XX_BLACK);
33  }
```



- In der Definition (Zeile 26) stehen in den Klammern die **Parameter**: `int x, int y, uint16_t farbe`. Jeder Parameter hat einen Typ und einen Namen, getrennt durch Kommas. Innerhalb der Funktion sind sie wie Variablen.
- Beim Aufruf (Zeilen 17 bis 19) stehen in den Klammern die **Argumente**: die Werte, die in die Parameter kommen, in derselben Reihenfolge. Beim ersten Aufruf ist `x` gleich 70, `y` gleich 80 und `farbe` gelb.
- Alle Maße im Smiley sind relativ zu `x` und `y` gerechnet: `x - 14` ist das linke Auge, `x + 14` das rechte. Deshalb passt das Gesicht überall hin.

Parameter sind Kopien. Die Funktion bekommt den **Wert** des Arguments, nicht die Variable selbst. Ändert sie ihren Parameter, ändert sich draußen nichts:

k11_kopie.ino

```
1  // Parameter sind Kopien: die Funktion aendert nur ihre eigene Kopie
2  void setup() {
3      Serial.begin(115200);
4      while (!Serial) {
5          delay(10);
6      }
7      int punkte = 10;
8      verdoppeln(punkte);
9      Serial.printf("Nach verdoppeln(punkte): %d\n", punkte);
10     punkte = doppelt(punkte);
11     Serial.printf("Nach punkte = doppelt(punkte): %d\n", punkte);
12 }
13
14 void loop() {
15 }
16
17 void verdoppeln(int zahl) {
18     zahl = zahl * 2;
19     Serial.printf("In verdoppeln: %d\n", zahl);
20 }
21
22 int doppelt(int zahl) {
23     return zahl * 2;
24 }
```

Serieller Monitor

```
In verdoppeln: 20
Nach verdoppeln(punkte): 10
Nach punkte = doppelt(punkte): 20
```

`verdoppeln` rechnet richtig, aber nur mit seiner eigenen Kopie `zahl`. `punkte` in `setup()` bleibt 10. Soll ein Ergebnis zurück nach draußen, gibt die Funktion es mit `return` zurück, wie `doppelt` in Zeile 23. Das ist das Thema des nächsten Abschnitts.

11.3 Rückgabewert: `return`

Viele Funktionen rechnen etwas aus und liefern ein Ergebnis, das du weiterverwendest. `digitalRead` liefert `HIGH` oder `LOW`, `millis` eine Zeit, `random` eine Zufallszahl. Bei eigenen Funktionen steht statt `void` der **Typ des Ergebnisses** vor dem Namen, und `return` gibt das Ergebnis zurück:

k11_rueckgabe.ino

```
 1 // Funktionen mit Rueckgabewert
 2 void setup() {
 3     Serial.begin(115200);
 4     while (!Serial) {
 5         delay(10);
 6     }
 7     int q = quadrat(7);
 8     Serial.printf("7 zum Quadrat ist %d\n", q);
 9     Serial.printf("12 zum Quadrat ist %d\n", quadrat(12));
10     Serial.printf("Kreis mit Radius 2.5: Flaeche %.2f\n", kreisflaeche(2.5));
11     Serial.printf("Mittelwert von 3 und 8: %.1f\n", mittelwert(3, 8));
12 }
13
14 void loop() {
15 }
16
17 int quadrat(int zahl) {
18     return zahl * zahl;
19 }
20
21 float kreisflaeche(float radius) {
22     return 3.14159 * radius * radius;
23 }
24
25 float mittelwert(float a, float b) {
26     return (a + b) / 2;
27 }
```

Serieller Monitor

```
7 zum Quadrat ist 49
12 zum Quadrat ist 144
Kreis mit Radius 2.5: Flaeche 19.63
Mittelwert von 3 und 8: 5.5
```

- `int quadrat(int zahl)` (Zeile 17): Die Funktion bekommt eine ganze Zahl und liefert eine ganze Zahl.
- **`return zahl * zahl;`** rechnet den Ausdruck aus, beendet die Funktion sofort und gibt das Ergebnis zurück. Was hinter einem `return` steht, wird nicht mehr ausgeführt.

- Der Aufruf `quadrat(7)` steht da, wo sonst eine Zahl stünde: rechts vom `=` (Zeile 7) oder direkt als Argument von `printf` (Zeile 9).
- `kreisflaeche` und `mittelwert` liefern `float`. `mittelwert(3, 8)` gibt 5.5 zurück, denn die Parameter sind `float`; die ganzen Zahlen 3 und 8 werden beim Aufruf in Kommazahlen umgewandelt.

Jeder Weg durch die Funktion muss bei einem `return` ankommen. Diese Funktion vergisst den Fall `zahl == 0`:

k11_fehler_return.ino

```
 1 // Eine Funktion mit Rueckgabewert muss immer etwas zurueckgeben
 2 void setup() {
 3     Serial.begin(115200);
 4     Serial.println(vorzeichen(-5));
 5 }
 6
 7 void loop() {
 8 }
 9
10 int vorzeichen(int zahl) {
11     if (zahl > 0) {
12         return 1;
13     } else if (zahl < 0) {
14         return -1;
15     }
16 }
```

Ausgabe

```
k11_fehler_return.ino: In function 'int vorzeichen(int)':
k11_fehler_return.ino:16:1: error: control reaches end of non-void function [-
  Werror=return-type]
   16 | }
      | ^
cc1plus: some warnings being treated as errors
Fehler beim Build: exit status 1
```

Control reaches end of non-void function heißt: Der Ablauf kann das Ende einer Funktion erreichen, die etwas zurückgeben muss, ohne dass ein `return` kam. Bei vielen anderen Arduino-Boards ist das nur eine Warnung, und der Sketch liefert dann irgendeinen Wert. Der Kern für den RP2350 macht daraus einen Fehler (`-Werror=return-type` heißt „diese Warnung als Fehler behandeln“). Das ist gut so: Ein `return 0;` am Ende behebt ihn.

Die Zahl der Argumente muss stimmen. Der Compiler prüft bei jedem Aufruf, ob die Argumente zu den Parametern passen:

k11_fehler_argumente.ino

```
// Beim Aufruf fehlt ein Argument
void setup() {
    Serial.begin(115200);
    Serial.println(mittelwert(3));
}

void loop() {
}

float mittelwert(float a, float b) {
    return (a + b) / 2;
}
```

Ausgabe

```
k11_fehler_argumente.ino: In function 'void setup()':
k11_fehler_argumente.ino:4:30: error: too few arguments to function 'float
  mittelwert(float, float)'
    4 |     Serial.println(mittelwert(3));
      |                    ~~~~~~~~~~^~~
k11_fehler_argumente.ino:10:7: note: declared here
   10 | float mittelwert(float a, float b) {
      |       ^~~~~~~~~~
Fehler beim Build: exit status 1
```

Too few arguments, zu wenige Argumente. Die `note:` darunter zeigt, wo die Funktion steht und welche Parameter sie erwartet.

> `typ name(typ parameter, …) { … return wert; }`: `void`, wenn die Funktion nichts zurückgibt; sonst der Typ des Ergebnisses und ein `return` auf jedem Weg. Parameter sind Kopien der Argumente.

11.4 Wo die Funktion steht: Prototypen

In C++ muss eine Funktion eigentlich **bekannt** sein, bevor sie aufgerufen wird. In allen Sketches dieses Kapitels stehen die eigenen Funktionen aber **unter** `setup()` und `loop()`, die sie aufrufen. Dass das trotzdem geht, liegt an der Arduino IDE: Bevor sie den Sketch dem Compiler gibt, sucht sie alle Funktionen heraus und schreibt für jede eine Ankündigung an den Anfang, einen sogenannten **Prototyp**. Für `k11_rueckgabe` sind das Zeilen wie diese:

```
int quadrat(int zahl);
float kreisflaeche(float radius);
float mittelwert(float a, float b);
```

Ein Prototyp ist die erste Zeile der Funktion mit einem Semikolon statt des Blocks. Er sagt dem Compiler: Diese Funktion gibt es, sie hat diese Parameter und diesen Typ. Du siehst die Prototypen nicht, die IDE fügt sie nur in eine Kopie deines Sketches ein. Du darfst sie aber auch selbst schreiben, oben im Sketch; das schadet nichts. In normalen C++-Dateien (`.cpp`), etwa in einer Bibliothek (Kapitel 20), musst du es sogar.

> Die IDE setzt die Prototypen vor die **erste Funktion** im Sketch. Alles, was darüber steht, ist bei den Prototypen schon bekannt: `#include`, `#define`, globale Variablen. Wenn du in Kapitel 13 eigene Typen (`struct`) erfindest, schreib sie deshalb ganz nach oben, vor `setup()`.

11.5 Lokale und globale Variablen, `static`

Eine Variable gilt nur in dem Block, in dem sie angelegt wurde. Diesen Bereich nennt man ihren **Gültigkeitsbereich**.

- Eine **lokale** Variable steht in einer Funktion (oder in einem anderen Block, etwa einer `for`-Schleife). Sie entsteht, wenn der Ablauf an ihre Zeile kommt, und verschwindet am Ende des Blocks wieder. Andere Funktionen kennen sie nicht.
- Eine **globale** Variable steht außerhalb aller Funktionen, meist oben im Sketch. Sie gibt es die ganze Zeit, und alle Funktionen können sie lesen und ändern. Die Variablen `vorher` und `zaehler` aus Kapitel 9 waren global, damit sie von einem `loop()`-Durchlauf zum nächsten erhalten bleiben.

k11_lokal_global.ino

```
// Lokale und globale Variablen
int punkte = 0;             // global: ueberall bekannt

void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    treffer();
    treffer();
    treffer();
    Serial.printf("Punkte nach drei Treffern: %d\n", punkte);
}

void loop() {
}

void treffer() {
    int bonus = 5;          // lokal: gibt es nur in treffer()
    punkte = punkte + bonus;
    bonus = bonus + 1;      // hilft nichts: beim naechsten Mal wieder 5
}
```

Serieller Monitor

```
Punkte nach drei Treffern: 15
```

`punkte` ist global, also zählt jeder Aufruf von `treffer` dazu. `bonus` ist lokal: Bei jedem Aufruf entsteht es neu mit dem Wert 5. Die Erhöhung in Zeile 21 ist am Ende der Funktion verloren.

Lieber lokal. Globale Variablen sind bequem, aber in einem großen Sketch weiß man bald nicht mehr, welche Funktion sie wann ändert. Nimm global nur, was wirklich mehrere Funktionen brauchen oder was von einem `loop()`-Durchlauf zum nächsten erhalten bleiben muss.

`static`: lokal, aber mit Gedächtnis. Manchmal braucht nur **eine** Funktion einen Wert, der von Aufruf zu Aufruf erhalten bleibt. Dann schreibst du `static` vor die lokale Variable:

k11_static.ino

```
15 void zaehle() {
16     int normal = 0;             // wird bei jedem Aufruf neu angelegt
17     static int bleibt = 0;      // wird nur einmal angelegt
18     normal = normal + 1;
19     bleibt = bleibt + 1;
20     Serial.printf("normal = %d, bleibt = %d\n", normal, bleibt);
21 }
```

Serieller Monitor

```
normal = 1, bleibt = 1
normal = 1, bleibt = 2
normal = 1, bleibt = 3
```

`normal` wird bei jedem Aufruf neu auf 0 gesetzt. `bleibt` wird nur **einmal** angelegt, beim ersten Aufruf, und behält danach seinen Wert. Von außen ist es trotzdem unsichtbar: Keine andere Funktion kann es versehentlich ändern.

> **Lokal** (in einer Funktion): nur dort bekannt, bei jedem Aufruf neu. **Global** (außerhalb): überall bekannt, die ganze Zeit da. **`static`** lokal: nur in der Funktion bekannt, behält aber ihren Wert.

11.6 Funktionen für die Tasten

Mit Funktionen wird die Tastenabfrage aus Kapitel 9 kurz und lesbar:

k11_taste_funktion.ino

```
1 // Tasten abfragen mit eigenen Funktionen
2 #define TASTE_A 27
3 #define TASTE_B 28
4
5 int zaehler = 0;
6
7 void setup() {
8     Serial.begin(115200);
```

```
 9     pinMode(TASTE_A, INPUT_PULLUP);
10     pinMode(TASTE_B, INPUT_PULLUP);
11 }
12
13 void loop() {
14     if (a_neu_gedrueckt()) {
15         zaehler = zaehler + 1;
16         Serial.printf("A: %d\n", zaehler);
17     }
18     if (gedrueckt(TASTE_B)) {
19         zaehler = 0;
20     }
21     delay(10);
22 }
23
24 // true, solange die Taste am Pin gedrueckt ist
25 bool gedrueckt(int pin) {
26     return digitalRead(pin) == LOW;
27 }
28
29 // true genau einmal pro Druck auf A
30 bool a_neu_gedrueckt() {
31     static bool vorher = false;
32     bool jetzt = gedrueckt(TASTE_A);
33     bool neu = jetzt && !vorher;
34     vorher = jetzt;
35     return neu;
36 }
```

Serieller Monitor

```
A: 1
A: 2
A: 1
```

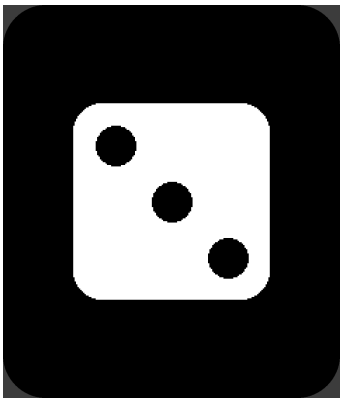
- `gedrueckt(pin)` (Zeile 25) gibt das Ergebnis des Vergleichs direkt zurück: `true` oder `false`. Sie funktioniert für jede Taste, denn der Pin ist ein Parameter.
- `a_neu_gedrueckt()` (Zeile 30) ist die Flankenerkennung aus Kapitel 9. `vorher` ist hier `static`: Es gehört nur dieser Funktion.
- In `loop()` liest man jetzt fast wie einen Satz: „Wenn A neu gedrückt ist, zähle. Wenn B gedrückt ist, setze zurück."

Hier stößt die Funktion an eine Grenze. Für B bräuchtest du eine zweite Funktion `b_neu_gedrueckt()` mit eigenem `vorher`, für den Joystick fünf weitere. Ein einziges `static vorher` für alle Tasten geht nicht, denn jede Taste braucht ihr eigenes Gedächtnis. Daten und Funktionen, die zusammen gehören, bündelt man in C++ zu einer **Klasse**. Das ist das Ziel von Kapitel 13.

> Mehrere Funktionen dürfen in C++ denselben Namen haben, wenn sich ihre Parameter unterscheiden. Das heißt **Überladen**. Deshalb gibt es `Serial.print` für Zahlen, Kommazahlen und Texte: Es sind verschiedene Funktionen mit demselben Namen, und der Compiler wählt nach dem Typ des Arguments die passende aus.

Ausprobieren: Ein Würfel

Drück A, und der Würfel rollt: Er zeigt schnell hintereinander zufällige Zahlen, wird langsamer und bleibt stehen. Jeder Wechsel klickt leise.



k11_wuerfel.ino

```
25 void loop() {
26     if (a_neu_gedrueckt()) {
27         if (!gestartet) {
28             randomSeed(micros());
29             gestartet = true;
30         }
31         int augen = wuerfeln();
32         Serial.printf("Gewuerfelt: %d\n", augen);
33     }
34     delay(10);
35 }
36
37 // rollt kurz und gibt die geworfene Zahl zurueck
38 int wuerfeln() {
39     int augen = 1;
40     for (int i = 0; i < 8; i++) {
41         augen = random(1, 7);
42         zeichne_wuerfel(augen);
43         tone(LAUTSPRECHER, 2000, 5);
44         delay(40 + i * 15);     // wird langsamer, wie ein echter Wuerfel
45     }
46     return augen;
47 }
48
49 // zeichnet den Wuerfel mit 1 bis 6 Augen in die Bildschirmmitte
50 void zeichne_wuerfel(int augen) {
51     tft.fillRoundRect(50, 70, 140, 140, 20, ST77XX_WHITE);
```

```
52      if (augen == 1 || augen == 3 || augen == 5) {
53          zeichne_auge(0, 0);                          // Mitte
54      }
55      if (augen >= 2) {
56          zeichne_auge(-1, -1);                        // links oben
57          zeichne_auge(1, 1);                          // rechts unten
58      }
59      if (augen >= 4) {
60          zeichne_auge(1, -1);                         // rechts oben
61          zeichne_auge(-1, 1);                         // links unten
62      }
63      if (augen == 6) {
64          zeichne_auge(-1, 0);                         // links Mitte
65          zeichne_auge(1, 0);                          // rechts Mitte
66      }
67  }
68
69  // ein Auge; spalte und zeile sind -1, 0 oder 1
70  void zeichne_auge(int spalte, int zeile) {
71      tft.fillCircle(120 + spalte * 40, 140 + zeile * 40, 14, ST77XX_BLACK);
72  }
73
74  bool a_neu_gedrueckt() {
75      static bool vorher = false;
76      bool jetzt = digitalRead(TASTE_A) == LOW;
77      bool neu = jetzt && !vorher;
78      vorher = jetzt;
79      return neu;
80  }
```

Serieller Monitor

```
Gewuerfelt: 5
Gewuerfelt: 3
```

- `loop()` ist nur zehn Zeilen lang. Was ein Wurf ist und wie ein Würfel aussieht, steht in den Funktionen darunter.
- `wuerfeln()` (Zeile 38) zeigt achtmal eine zufällige Zahl und gibt die letzte zurück. `delay(40 + i * 15)` wartet jedes Mal etwas länger: erst 40 ms, zuletzt 145 ms.
- `random(1, 7)` liefert 1 bis 6: Die obere Grenze ist **nicht** dabei.
- `zeichne_wuerfel(augen)` malt den weißen Würfel und dann die Augen. Ungerade Zahlen haben ein Auge in der Mitte, ab 2 kommen links oben und rechts unten dazu, ab 4 die anderen beiden Ecken, bei 6 die Mitte links und rechts.
- `zeichne_auge(spalte, zeile)` rechnet die Lage aus: Spalte und Zeile sind −1, 0 oder 1, mal 40 Pixel Abstand von der Mitte des Würfels. Eine Funktion ruft hier eine andere auf.
- Der Zufall startet wie beim Reaktionstest in Kapitel 9 mit `micros()` beim ersten Druck. `gestartet` sorgt dafür, dass das nur einmal passiert.

Zum Verändern: Würfle mit B zwei Würfel nebeneinander. Dafür bekommt `zeichne_wuerfel` zwei weitere Parameter für die Position, und `zeichne_auge` braucht sie auch.

Übungsaufgaben

Aufgabe 1 ⭐ **Sterne** Schreibe eine Funktion `void sterne(int anzahl)`, die `anzahl` Sterne in eine Zeile des seriellen Monitors schreibt. Rufe sie in einer Schleife auf, sodass ein Dreieck entsteht: erst ein Stern, dann zwei, … bis fünf.

Aufgabe 2 ⭐ **Fahrenheit** In den USA misst man Temperaturen in Grad Fahrenheit: F = C · 9 / 5 + 32. Schreibe eine Funktion `float fahrenheit(float celsius)` und gib damit eine Tabelle von −10 °C bis 40 °C in Zehnerschritten aus.

Aufgabe 3 ⭐⭐ **Schaltjahr** Ein Jahr ist ein Schaltjahr, wenn es durch 4 teilbar ist, außer es ist durch 100 teilbar, außer es ist durch 400 teilbar. Schreibe `bool ist_schaltjahr(int jahr)` und prüfe 1900, 2000, 2024 und 2026. Tipp: `jahr % 4 == 0` ist wahr, wenn beim Teilen durch 4 kein Rest bleibt.

Aufgabe 4 ⭐⭐ **Unbekannte Messwerte** Dieser Sketch übersetzt nicht. Lies die Fehlermeldung, erkläre sie und verbessere den Sketch so, dass `mittel` die beiden Messwerte als Parameter bekommt. Ein zweiter Fehler zeigt sich erst, wenn der Sketch läuft: Was fehlt in `setup()`, damit du die Ausgabe sicher siehst?

k11_aufgabe4.ino

```
// Soll den Mittelwert von zwei Messungen ausgeben. Findest du die Fehler?
void setup() {
    Serial.begin(115200);
    int messung1 = 17;
    int messung2 = 24;
    ausgeben(mittel());
}

void loop() {
}

float mittel() {
    return (messung1 + messung2) / 2.0;
}

void ausgeben(float wert) {
    Serial.printf("Mittelwert: %.1f\n", wert);
}
```

Aufgabe 5 ⭐⭐⭐ Ein Zeichenbaukasten Schreibe Funktionen `zeichne_haus(x, y, farbe)` und `zeichne_baum(x, y, hoehe)` und baue damit ein Bild mit mindestens zwei Häusern und zwei Bäumen, Himmel, Wiese und Sonne. Überlege dir vorher, welcher Punkt eines Hauses (x, y) sein soll, damit alle Häuser leicht auf dieselbe Wiese passen.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Eine **Funktion** ist ein benannter Block. Definieren: `void name() { … }`, aufrufen: `name();`.
- **Parameter** in der Definition, **Argumente** beim Aufruf, in derselben Reihenfolge. Parameter sind **Kopien**.
- **Rückgabewert:** Typ statt `void` vor dem Namen, `return wert;` beendet die Funktion. Jeder Weg muss ein `return` erreichen (sonst Fehler `control reaches end of non-void function`).
- Die Arduino IDE schreibt die **Prototypen** selbst; eigene Funktionen dürfen deshalb unter `loop()` stehen.
- **Lokal** gilt nur im Block, **global** überall; `static` macht eine lokale Variable dauerhaft.

KAPITEL 12

Arrays und Texte

Worum geht's? Sieben Tasten, sieben `pinMode`-Zeilen, sieben Variablen für „vorher“: Das geht besser. Ein **Array** fasst viele Werte unter einem Namen zusammen, und eine Schleife bearbeitet sie alle. Texte sind in C++ ebenfalls Arrays, nämlich aus Zeichen. Daneben gibt es bei Arduino die bequeme Klasse `String`. Du lernst beide kennen, ihre Vor- und Nachteile, und warum ein „ü“ zwei Zeichen lang ist. Zum Schluss läuft ein Text über den Bildschirm.

12.1 Arrays: viele Werte unter einem Namen

Ein **Array** (deutsch etwa „Reihe“ oder „Feld“) ist eine Reihe von Variablen gleichen Typs, die hintereinander im Speicher liegen. Statt fünf Variablen `punkte1` bis `punkte5` legst du eine an:

k12_array.ino

```
// Ein Array: fuenf Werte unter einem Namen
int punkte[5] = {12, 7, 30, 18, 25};

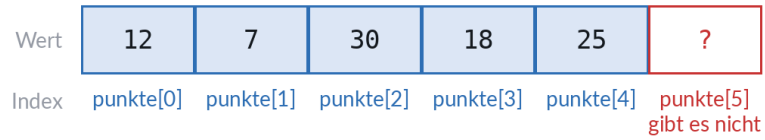
void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    Serial.printf("Erster Wert:  punkte[0] = %d\n", punkte[0]);
    Serial.printf("Letzter Wert: punkte[4] = %d\n", punkte[4]);
    punkte[1] = 99;                          // einen Wert aendern
    for (int i = 0; i < 5; i++) {
        Serial.printf("punkte[%d] = %d\n", i, punkte[i]);
    }
}

void loop() {
}
```

Serieller Monitor

```
Erster Wert:  punkte[0] = 12
Letzter Wert: punkte[4] = 25
punkte[0] = 12
punkte[1] = 99
punkte[2] = 30
punkte[3] = 18
punkte[4] = 25
```

```
int punkte[5] = {12, 7, 30, 18, 25};
```



- `int punkte[5]` legt ein Array aus fünf `int` an. Die Zahl in den eckigen Klammern ist die **Größe**. Die Werte in `{ }` sind die Startwerte.
- Die einzelnen Werte heißen **Elemente**. Du erreichst sie über ihre Nummer, den **Index**: `punkte[0]`, `punkte[1]` und so weiter.
- **Der Index beginnt bei 0.** Das erste Element ist `punkte[0]`, das letzte bei fünf Elementen `punkte[4]`. Ein `punkte[5]` gibt es nicht.
- Ein Element benutzt du wie eine normale Variable: lesen, ausgeben, mit `=` ändern (Zeile 11).
- Der Index darf eine Variable sein. Deshalb passen Arrays so gut zu `for`-Schleifen: `punkte[i]` mit `i` von 0 bis 4 (Zeilen 12 bis 14).

> `typ name[anzahl] = {wert, wert, …};` legt ein Array an. Der Index läuft von **0** bis **anzahl − 1**. Die Schleife dazu: `for (int i = 0; i < anzahl; i++)`, mit `<`, nicht `<=`.

12.2 Arrays durchlaufen und `sizeof`

Lässt du die Größe in den eckigen Klammern weg, zählt der Compiler die Startwerte selbst. Damit du die Anzahl nicht von Hand nachzählen musst, rechnest du sie aus:

k12_auswerten.ino

```
 1 // Ein Array durchlaufen: Summe, Mittelwert und groesster Wert
 2 int temperaturen[] = {14, 17, 21, 19, 23, 16, 12};
 3 const int ANZAHL = sizeof(temperaturen) / sizeof(temperaturen[0]);
 4
 5 void setup() {
 6     Serial.begin(115200);
 7     while (!Serial) {
 8         delay(10);
 9     }
10     Serial.printf("%d Werte, %d Bytes\n", ANZAHL, sizeof(temperaturen));
11
12     int summe = 0;
13     int groesster = temperaturen[0];
14     for (int i = 0; i < ANZAHL; i++) {
15         summe = summe + temperaturen[i];
```

```
16          if (temperaturen[i] > groesster) {
17              groesster = temperaturen[i];
18          }
19      }
20      Serial.printf("Mittelwert: %.1f Grad\n", (float)summe / ANZAHL);
21      Serial.printf("Hoechster Wert: %d Grad\n", groesster);
22  }
23
24  void loop() {
25  }
```

Serieller Monitor

```
7 Werte, 28 Bytes
Mittelwert: 17.4 Grad
Hoechster Wert: 23 Grad
```

- `sizeof` liefert die Größe in **Bytes**. Das ganze Array hat 28 Bytes, ein einzelnes `int` hat 4. 28 / 4 ergibt die Anzahl der Elemente, 7 (Zeile 3). Fügst du einen Wert hinzu, stimmt `ANZAHL` von selbst.
- `ANZAHL` ist mit `const` fest (Kapitel 5); es ändert sich nie.
- Zeile 13 nimmt das erste Element als vorläufig größtes. Die Schleife vergleicht jedes weitere damit und merkt sich ein größeres.
- `(float)summe` in Zeile 20 macht aus der ganzen Zahl eine Kommazahl, bevor geteilt wird. Sonst wäre das eine ganzzahlige Division (Kapitel 5) und der Mittelwert 17 statt 17,4.

Sieben Tasten mit Arrays. Hier zeigt sich, was Arrays können. Die Pins stehen in einem Array, die Namen in einem zweiten, und für jede Taste gibt es ein eigenes `vorher` und einen eigenen Zähler:

k12_tasten_array.ino

```
11  const int TASTEN[] = {4, 2, 3, 1, 0, 27, 28};
12  String namen[] = {"hoch", "runter", "links", "rechts", "Mitte", "A", "B"};
13  const int ANZAHL = 7;
14
15  bool vorher[ANZAHL];        // alle false
16  int zaehler[ANZAHL];        // alle 0
17
18  void setup() {
19      tft.init(240, 280);
20      tft.fillScreen(ST77XX_BLACK);
21      pinMode(TFT_LICHT, OUTPUT);
22      digitalWrite(TFT_LICHT, HIGH);
23      tft.setTextSize(2);
24      tft.setTextColor(ST77XX_WHITE, ST77XX_BLACK);
25      for (int i = 0; i < ANZAHL; i++) {
26          pinMode(TASTEN[i], INPUT_PULLUP);
27          tft.setCursor(40, 50 + i * 26);
28          tft.print(namen[i]);
29      }
30  }
```

```
32 void loop() {
33     for (int i = 0; i < ANZAHL; i++) {
34         bool jetzt = digitalRead(TASTEN[i]) == LOW;
35         if (jetzt && !vorher[i]) {
36             zaehler[i] = zaehler[i] + 1;
37         }
38         vorher[i] = jetzt;
39         tft.setCursor(160, 50 + i * 26);
40         tft.print(zaehler[i]);
41     }
42     delay(10);
43 }
```

- `TASTEN[i]`, `namen[i]`, `vorher[i]` und `zaehler[i]` gehören zur selben Taste Nummer `i`. Die Reihenfolge ist in allen Arrays gleich.
- Eine Schleife in `setup()` erledigt alle sieben `pinMode`. Eine Schleife in `loop()` macht für alle sieben Tasten die Flankenerkennung aus Kapitel 9.
- Globale Arrays ohne Startwerte (Zeilen 15 und 16) sind mit 0 gefüllt, bei `bool` also mit `false`. Lokale Arrays in einer Funktion dagegen enthalten zufällige Reste, bis du ihnen Werte gibst.
- `namen` ist ein Array aus `String`, der Klasse für Texte aus 12.7.



Vergleiche mit Kapitel 9: Dort wäre derselbe Sketch etwa dreimal so lang.

12.3 Arrays und Funktionen: eine Melodie

In Kapitel 10 stand jede Note in zwei eigenen Zeilen. Mit zwei Arrays, eins für die Töne und eins für die Dauern, wird daraus eine kurze Schleife:

k12_melodie.ino

```
1  // "Alle meine Entchen" aus zwei Arrays: Toene und Dauern
2  #define LAUTSPRECHER 15
3
4  // 0 bedeutet Pause
5  int toene[] = {262, 294, 330, 349, 392, 392, 440, 440, 440, 440, 392,
6                 440, 440, 440, 440, 392, 349, 349, 349, 349, 330, 330,
7                 294, 294, 294, 294, 262};
8  int dauern[] = {1, 1, 1, 1, 2, 2, 1, 1, 1, 1, 4,
9                  1, 1, 1, 1, 4, 1, 1, 1, 1, 2, 2,
10                 1, 1, 1, 1, 4};
11 const int ANZAHL = sizeof(toene) / sizeof(toene[0]);
12
13 #define ACHTEL 250
```

```
15 void setup() {
16     spiele(toene, dauern, ANZAHL);
17 }
18
19 void loop() {
20 }
21
22 void spiele(int t[], int d[], int anzahl) {
23     for (int i = 0; i < anzahl; i++) {
24         int ms = d[i] * ACHTEL;
25         if (t[i] > 0) {
26             tone(LAUTSPRECHER, t[i], ms - 30);
27         }
28         delay(ms);
29     }
30 }
```

- `toene[i]` und `dauern[i]` gehören zusammen. Die Dauer ist in Achteln angegeben: 1 ist eine Achtel, 4 eine halbe Note.
- Zeile 25 lässt Töne mit der Frequenz 0 aus. So kannst du eine Pause in die Melodie schreiben: Ton 0 mit der Länge der Pause.
- Die Funktion `spiele` (Zeile 22) bekommt die Arrays als Parameter. Ein Array-Parameter schreibt man mit leeren eckigen Klammern: `int t[]`.

> Eine Funktion erfährt **nicht**, wie groß ein Array ist, das sie bekommt. `sizeof(t)` in `spiele` liefert nicht die Größe des Arrays. Gib die Anzahl deshalb immer als eigenen Parameter mit, wie hier `anzahl`. Außerdem bekommt die Funktion, anders als bei `int` oder `float`, **keine Kopie**: Ändert sie `t[0]`, ändert sie das Original. Warum das so ist, erklärt Kapitel 19.

12.4 Über das Ende hinaus

C++ prüft beim Ablauf **nicht**, ob ein Index gültig ist. `werte[3]` in einem Array mit drei Elementen liest einfach den Speicher hinter dem Array, was immer dort gerade steht. Beim Schreiben ist es noch schlimmer: Dann überschreibst du eine fremde Variable, und der Sketch tut irgendetwas Seltsames oder stürzt ab. Der häufigste Grund ist `<=` statt `<` in der Schleife:

k12_ueber_ende.ino

```
1 // Ueber das Ende eines Arrays hinaus: der Compiler warnt nur, wenn er es sieht
2 int werte[3] = {10, 20, 30};
3
4 void setup() {
5     Serial.begin(115200);
```

```
 6     while (!Serial) {
 7         delay(10);
 8     }
 9     for (int i = 0; i <= 3; i++) {      // Fehler: <= statt <
10         Serial.println(werte[i]);
11     }
12 }
13
14 void loop() {
15 }
```

Ausgabe

```
k12_ueber_ende.ino: In function 'void setup()':
k12_ueber_ende.ino:10:23: warning: iteration 3 invokes undefined behavior [-
  Waggressive-loop-optimizations]
   10 |         Serial.println(werte[i]);
      |         ~~~~~~~~~~~~~~^~~~~~~~~~
k12_ueber_ende.ino:9:23: note: within this loop
    9 |     for (int i = 0; i <= 3; i++) {      // Fehler: <= statt <
      |                     ~~^~~
```

Hier merkt es der Compiler, denn die Grenzen stehen fest im Sketch: *iteration 3 invokes undefined behavior*, der vierte Durchlauf (gezählt ab 0) führt zu undefiniertem Verhalten. **Undefiniertes Verhalten** heißt: Es kann alles passieren. Diese Warnung siehst du nur, wenn in den Einstellungen der IDE die Compiler-Meldungen auf **Alle** stehen (Kapitel 2). Kommt der Index dagegen aus einer Rechnung oder einer Eingabe, warnt niemand. Dann musst du selbst prüfen, etwa mit `if (i >= 0 && i < ANZAHL)`.

12.5 Texte sind `char`-Arrays

Ein Text in Anführungszeichen ist für C++ ein Array aus Zeichen vom Typ `char`. Jedes Zeichen ist eine kleine Zahl, sein **Zeichencode**: `'A'` ist 65, `'a'` ist 97, `'0'` ist 48. Einzelne Zeichen stehen in einfachen Anführungszeichen, Texte in doppelten. Am Ende jedes Texts steht ein zusätzliches Zeichen mit dem Code 0, geschrieben `'\0'`. Es markiert das Ende; so wissen Funktionen wie `print`, wo der Text aufhört.

k12_char_array.ino

```
 1 // Ein Text ist ein Array aus char mit einer 0 am Ende
 2 #include <string.h>
 3
 4 char name[] = "Mia";
 5
 6 void setup() {
 7     Serial.begin(115200);
 8     while (!Serial) {
 9         delay(10);
10     }
11     Serial.printf("Text: %s\n", name);
```

```
12      Serial.printf("strlen: %d, sizeof: %d\n", strlen(name), sizeof(name));
13      for (int i = 0; i < (int)sizeof(name); i++) {
14          Serial.printf("name[%d] = '%c' = %d\n", i, name[i], name[i]);
15      }
16      name[0] = 'L';                      // ein Zeichen austauschen
17      Serial.printf("Jetzt: %s\n", name);
18  }
19
20  void loop() {
21  }
```

Serieller Monitor

```
Text: Mia
strlen: 3, sizeof: 4
name[0] = 'M' = 77
name[1] = 'i' = 105
name[2] = 'a' = 97
name[3] = ' ' = 0
Jetzt: Lia
```

- `char name[] = "Mia";` legt ein Array aus **vier** `char` an: M, i, a und die abschließende 0. `sizeof` zählt sie mit.
- **`strlen(name)`** (*string length*) zählt die Zeichen bis zur 0, ohne sie: 3. Die Funktionen für `char`-Texte kommen aus `<string.h>` (Zeile 2).
- `%c` in `printf` gibt ein einzelnes Zeichen aus, `%d` seinen Code. Das letzte Element `name[3]` ist die 0; als Zeichen ausgegeben ist es unsichtbar.
- Zeile 16 tauscht ein einzelnes Zeichen aus. Aus „Mia“ wird „Lia“.

Ein `char`-Array hat eine **feste Größe**. In `name` passen nie mehr als drei Zeichen. Für längere Texte legst du von vornherein mehr Platz an, etwa `char zeile[20];`.

12.6 Texte zusammensetzen: `snprintf` und `strcmp`

Mit `+` lassen sich `char`-Arrays nicht verbinden. Dafür gibt es **`snprintf`**: Es funktioniert wie `Serial.printf` aus Kapitel 4, schreibt aber in ein `char`-Array statt in den seriellen Monitor. Das Ergebnis kannst du dann zum Beispiel aufs Display bringen:

k12_snprintf.ino

```
12  char spieler[] = "Mia";
13  int punkte = 1250;
14
15  void setup() {
16      Serial.begin(115200);
17      tft.init(240, 280);
18      tft.fillScreen(ST77XX_BLACK);
19      pinMode(TFT_LICHT, OUTPUT);
20      digitalWrite(TFT_LICHT, HIGH);
```

```
22      char zeile[20];
23      snprintf(zeile, sizeof(zeile), "%s: %d", spieler, punkte);
24      tft.setTextSize(3);
25      tft.setTextColor(ST77XX_YELLOW);
26      tft.setCursor(30, 100);
27      tft.print(zeile);
28
29      char kurz[8];
30      snprintf(kurz, sizeof(kurz), "%s: %d", spieler, punkte);
31      tft.setTextColor(ST77XX_CYAN);
32      tft.setCursor(30, 160);
33      tft.print(kurz);                    // abgeschnitten, aber sicher
34
35      if (strcmp(spieler, "Mia") == 0) {
36          Serial.println("Hallo Mia!");
37      }
38  }
```



- `snprintf(ziel, sizeof(ziel), format, …)`: Das zweite Argument ist der Platz im Ziel. `snprintf` schreibt **nie** mehr hinein, als passt.
- In `kurz` (Zeile 29) passen nur 7 Zeichen und die 0. Der Text wird abgeschnitten, aber es entsteht kein Schaden. Ohne die Größenangabe würde der Rest über das Ende des Arrays hinaus geschrieben (12.4).
- **Vergleichen** kannst du `char`-Texte nicht mit `==`. Das würde prüfen, ob es dasselbe Array ist, nicht ob dieselben Buchstaben darin stehen. **`strcmp(a, b)`** (*string compare*) vergleicht Zeichen für Zeichen und liefert **0**, wenn beide gleich sind.

> Weitere Funktionen aus `<string.h>`: `strcpy(ziel, quelle)` kopiert einen Text, `strcat(ziel, quelle)` hängt einen an. Beide prüfen den Platz nicht; `snprintf` ist sicherer und kann beides.

12.7 Die Klasse `String`

Arduino hat für Texte eine bequemere Lösung: die **Klasse `String`** (mit großem S). Sie kennst du schon aus Kapitel 4, `Serial.readStringUntil` liefert einen `String`. Was eine Klasse genau ist, erklärt Kapitel 13; hier reicht: Ein `String` ist ein Text, der selbst weiß, wie lang er ist, der wachsen kann und eigene Funktionen mitbringt.

k12_string.ino

```
1 // Die Klasse String: bequem Texte bauen und zerlegen
2 void setup() {
3     Serial.begin(115200);
```

```
    while (!Serial) {
        delay(10);
    }
    String name = "Mia";
    String gruss = "Hallo " + name + "!";
    Serial.println(gruss);
    Serial.printf("Laenge: %d\n", gruss.length());

    gruss.toUpperCase();                // aendert gruss selbst
    Serial.println(gruss);

    String eingabe = "punkte=1250";
    int stelle = eingabe.indexOf('=');
    String wort = eingabe.substring(0, stelle);
    int zahl = eingabe.substring(stelle + 1).toInt();
    Serial.printf("Wort: %s, Zahl: %d\n", wort.c_str(), zahl);

    String text = "Stufe " + String(3) + " von " + String(10);
    Serial.println(text);
    if (wort == "punkte") {
        Serial.println("Vergleich mit == klappt");
    }
}

void loop() {
}
```

Serieller Monitor

```
Hallo Mia!
Laenge: 10
HALLO MIA!
Wort: punkte, Zahl: 1250
Stufe 3 von 10
Vergleich mit == klappt
```

Schreibweise	bewirkt
`a + b`	hängt Texte aneinander; mindestens einer muss ein `String` sein
`s.length()`	Anzahl der Zeichen (genauer: Bytes, siehe 12.9)
`s.toUpperCase()`, `s.toLowerCase()`	ändert `s` selbst in Groß- oder Kleinbuchstaben
`s.indexOf('=')`	Stelle des ersten `=`, ab 0 gezählt; −1, wenn keins da ist
`s.substring(a, b)`	Teil von Stelle `a` bis vor Stelle `b`; ohne `b` bis zum Ende
`s.toInt()`, `s.toFloat()`	macht aus dem Text eine Zahl
`String(3)`	macht aus einer Zahl einen Text

Schreibweise	bewirkt
`s == "punkte"`	vergleicht den Inhalt, anders als bei `char`-Arrays
`s[i]`	einzelnes Zeichen, wie beim Array
`s.trim()`	entfernt Leerzeichen und Zeilenenden vorn und hinten

> `Serial.printf("%s", s)` mit einem `String` geht schief: `%s` erwartet ein `char`-Array. Gib stattdessen `s.c_str()` an (Zeile 19). Das liefert den Text als `char`-Array. Eine Warnung gibt es nicht (Kapitel 4).

12.8 `char`-Array oder `String`?

Der Unterschied liegt im **Speicher**. Ein `char`-Array hat eine feste Größe, die beim Übersetzen feststeht. Ein `String` dagegen besteht aus zwei Teilen: einer kleinen Verwaltung (12 Bytes: wo der Text liegt, wie viel Platz da ist und wie lang er ist) und dem eigentlichen Text in einem Bereich des Arbeitsspeichers, der während des Laufs vergeben wird, dem **Heap**. Wird der Text länger, besorgt sich der `String` dort einen größeren Platz und kopiert um. Jedes `+` erzeugt dabei einen neuen, vorübergehenden `String`.

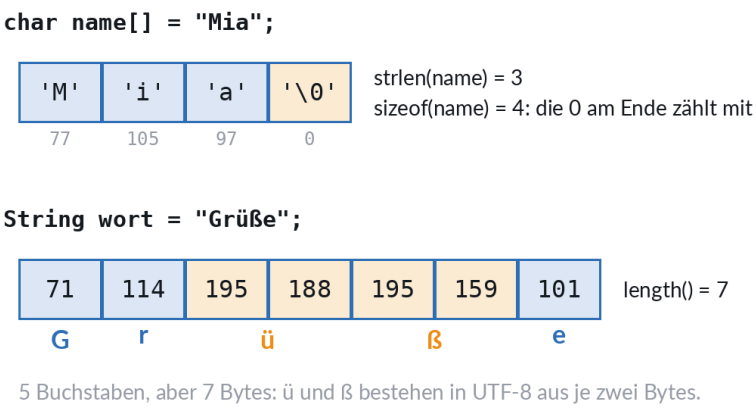
	`char`-Array	`String`
Größe	fest	wächst bei Bedarf
Zusammensetzen	`snprintf`	`+`
Vergleichen	`strcmp(a, b) == 0`	`a == b`
zu langer Text	wird abgeschnitten (mit `snprintf`) oder überschreibt Fremdes	kein Problem, solange Speicher frei ist
Speicher	genau berechenbar	vom Heap, kann zerstückeln
Tempo	schnell	jedes `+` kostet Zeit

Was heißt „zerstückeln“? Werden immer wieder Texte verschiedener Länge angelegt und freigegeben, entstehen auf dem Heap Lücken. Irgendwann ist zwar insgesamt noch genug frei, aber kein zusammenhängendes Stück groß genug. Auf kleinen Arduinos mit 2 KB Arbeitsspeicher ist das ein echtes Problem, deshalb liest man im Internet oft „Nimm nie `String`!“. Der RP2350 hat 520 KB, da passiert das in Sketches wie denen in diesem Buch praktisch nicht.

Faustregel: `String` für Eingaben und für Texte, die selten entstehen; `char`-Arrays mit `snprintf` für Anzeigen, die sich viele Male pro Sekunde ändern, etwa einen Punktestand in jedem Bild. Wenn du vorher weißt, wie lang ein `String` ungefähr wird, kann `s.reserve(50)` den Platz gleich am Anfang besorgen.

12.9 Umlaute sind mehrere Bytes lang

Die Zeichencodes von 0 bis 127 (der **ASCII**-Code) enthalten nur englische Buchstaben, Ziffern und Satzzeichen. Für alle anderen Zeichen der Welt, also auch ä, ö, ü und ß, benutzt die Arduino IDE die Kodierung **UTF-8**: Ein solches Zeichen wird aus **zwei bis vier Bytes** zusammengesetzt. C++ zählt aber Bytes, keine Buchstaben. Zum Vergleich zeigt die Abbildung noch einmal einen reinen ASCII-Text:



5 Buchstaben, aber 7 Bytes: ü und ß bestehen in UTF-8 aus je zwei Bytes.



k12_umlaute.ino

```
11 void setup() {
12     Serial.begin(115200);
13     while (!Serial) {
14         delay(10);
15     }
16     tft.init(240, 280);
17     tft.fillScreen(ST77XX_BLACK);
18     pinMode(TFT_LICHT, OUTPUT);
19     digitalWrite(TFT_LICHT, HIGH);
20
21     String wort = "Grüße";
22     Serial.println(wort);
23     Serial.printf("length(): %d\n", wort.length());
24     for (unsigned int i = 0; i < wort.length(); i++) {
25         Serial.printf("%d ", (unsigned char)wort[i]);
26     }
```

```
    Serial.println();

    tft.setTextSize(4);
    tft.setTextColor(ST77XX_WHITE);
    tft.setCursor(30, 80);
    tft.print(wort);                    // falsche Zeichen
    tft.setCursor(30, 160);
    tft.print("Gruesse");               // so geht es
}
```

Serieller Monitor

```
Grüße
length(): 7
71 114 195 188 195 159 101
```

- „Grüße“ hat fünf Buchstaben, aber `length()` meldet **7**: ü ist das Bytepaar 195 188, ß das Paar 195 159. `strlen` würde genauso zählen.
- Der serielle Monitor versteht UTF-8 und zeigt das Wort richtig an.
- Die Schrift des Displays kennt UTF-8 nicht. Sie malt für jedes Byte ein eigenes Zeichen aus einer alten Zeichentabelle, daher die seltsamen Zeichen im Bild.
- Ein Index wie `wort[2]` liefert nur die erste Hälfte des ü.

Deshalb schreiben die Sketches in diesem Buch auf dem Display und im seriellen Monitor „ae“, „oe“, „ue“ und „ss“. Das ist einfach und funktioniert überall.

Ausprobieren: Ein Lauftext

Ein Text läuft in großer Schrift von rechts nach links über den Bildschirm, immer wieder. Tippst du im seriellen Monitor einen neuen Text ein, läuft ab dann dieser.

k12_lauftext.ino

```
#define GROESSE 5           // Zeichen 30 x 40 Pixel
#define ZEICHENBREITE (6 * GROESSE)

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);

String text = "Hallo PicoBoy!";
int x = 240;                        // linker Rand des Textes

void setup() {
    Serial.begin(115200);
    tft.init(240, 280);
    tft.fillScreen(ST77XX_BLACK);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
    tft.setTextSize(GROESSE);
    tft.setTextColor(ST77XX_YELLOW, ST77XX_BLACK);
```

```
24      tft.setTextWrap(false);         // nicht in die naechste Zeile umbrechen
25  }
26
27  void loop() {
28      // neuer Text aus dem seriellen Monitor?
29      if (Serial.available() > 0) {
30          String neu = Serial.readStringUntil('\n');
31          neu.trim();
32          if (neu.length() > 0) {
33              text = neu;
34              Serial.println("Neuer Text: " + text);
35          }
36      }
37
38      tft.setCursor(x, 120);
39      tft.print(text);
40      tft.print(" ");                 // loescht das letzte Zeichen von vorher
41      x = x - 3;
42      int breite = text.length() * ZEICHENBREITE;
43      if (x < -breite) {              // ganz links hinaus: rechts neu anfangen
44          x = 240;
45      }
46      delay(20);
47  }
```

Serieller Monitor

```
Neuer Text: Arrays sind toll
```



- `x` ist der linke Rand des Texts. Er beginnt rechts außerhalb (240) und wird bei jedem Durchlauf um 3 Pixel kleiner. Bald ist er negativ: Dann liegt der Anfang des Texts links außerhalb des Bildschirms. Die Bibliothek malt nur, was sichtbar ist.
- **`setTextWrap(false)`** (Zeile 24) verhindert, dass ein zu langer Text in die nächste Zeile umbricht. Normalerweise bricht `print` am rechten Rand um.
- Die Schrift hat Größe 5. Jedes Zeichen der eingebauten Schrift ist 6 × 8 Pixel groß, mal 5 also 30 × 40 (Zeilen 8 und 9). Daraus rechnet Zeile 42 die Breite des ganzen Texts. Ist er ganz links hinaus, beginnt er rechts von vorn.
- Mit Hintergrundfarbe überschreibt jeder Durchlauf den alten Text. Nur ganz rechts bliebe ein Rest des letzten Buchstabens stehen, denn der Text ist 3 Pixel nach links gerückt. Das Leerzeichen in Zeile 40 malt dort Hintergrund.
- Neue Texte kommen als `String` (Zeile 30), denn sie können beliebig lang sein. Hier ist `String` genau richtig: Er entsteht nur selten.

Zum Verändern: Lass den Text schneller laufen. Ändere bei jedem neuen Durchgang die Farbe. Mit `tft.setTextSize(…)` und einer zweiten Zeile kannst du zwei Texte in verschiedenen Tempi laufen lassen.

Übungsaufgaben

Aufgabe 1 ⭐ **Notenschnitt** Lege ein Array mit deinen Noten an. Gib den Durchschnitt mit zwei Nachkommastellen, die beste und die schlechteste Note aus. Bei Schulnoten ist die beste die kleinste.

Aufgabe 2 ⭐ **Rückwärts** Gib den Text `"Arduino macht Spass"` rückwärts aus. Tipp: eine Schleife, die bei `strlen(text) - 1` beginnt und bis 0 zählt.

Aufgabe 3 ⭐⭐ **Melodie mit Lauflicht** Schreibe eine eigene kurze Melodie mit mindestens einer Pause in zwei Arrays. Bei jedem Ton leuchtet eine der drei LEDs, reihum rot, gelb, grün. Lege dafür auch die Pins der LEDs in ein Array. Tipp: `i % 3` ist der Rest beim Teilen durch 3, also immer 0, 1 oder 2.

Aufgabe 4 ⭐⭐ **Vokale zählen** Lies einen Satz aus dem seriellen Monitor und gib aus, wie viele Vokale (a, e, i, o, u) er enthält, egal ob groß oder klein geschrieben.

Aufgabe 5 ⭐⭐⭐ **Morsen** Lies einen Text aus dem seriellen Monitor und morse ihn mit der roten LED und dem Lautsprecher. Ein Punkt dauert 120 ms, ein Strich dreimal so lang; zwischen den Zeichen eines Buchstabens ist eine Punktlänge Pause, zwischen Buchstaben drei, zwischen Wörtern sieben. Die Codes für A bis Z stehen in einem Array aus `String`: `".-"`, `"-..."`, `"-.-."` … Tipp: Für einen Großbuchstaben `c` ist `c - 'A'` seine Nummer im Alphabet, ab 0 gezählt.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- **Array:** `int werte[5] = {…};`, Index von **0** bis 4. Anzahl: `sizeof(werte) / sizeof(werte[0])`. Schleife mit `i < anzahl`.
- C++ prüft Indizes **nicht**. Über das Ende hinaus ist undefiniertes Verhalten; der Compiler warnt nur, wenn er es sicher erkennt.
- An Funktionen: `void f(int a[], int anzahl)`, die Anzahl immer mitgeben. Das Array wird nicht kopiert.
- **`char`-Array:** Text mit `'\0'` am Ende, feste Größe. `strlen`, `strcmp(a, b) == 0`, `snprintf(ziel, sizeof(ziel), …)`.
- **`String`:** wächst, `+`, `==`, `length()`, `indexOf`, `substring`, `toInt`; Speicher vom Heap. Für `printf` mit `%s`: `s.c_str()`.
- Umlaute sind in UTF-8 **zwei Bytes**; das Display kann sie nicht. Auf dem Bildschirm „ae“, „oe“, „ue“, „ss“ schreiben.

KAPITEL 13

Strukturen und eine erste eigene Klasse

Worum geht's? Ein Ball hat eine Position, eine Richtung, eine Größe und eine Farbe. Bisher waren das lauter einzelne Variablen. Jetzt bündelst du sie mit `struct` zu einem eigenen Typ, legst ganze Arrays davon an und gibst Zuständen mit `enum` lesbare Namen. Zum Schluss schreibst du deine erste **Klasse**: eine Taste, die selbst weiß, ob sie gerade gedrückt wurde. Damit verstehst du auch, was `tft` und `rgb` eigentlich sind. Im Ausprobieren schwimmt ein ganzer Schwarm über den Bildschirm.

13.1 `struct`: Werte bündeln

Ein Punkt auf dem Bildschirm besteht aus zwei Zahlen, x und y. Mit **`struct`** (von *structure*, Struktur) machst du daraus einen eigenen Typ:

k13_struct.ino

```
// struct: zusammengehoerige Werte buendeln
struct Punkt {
    int x;
    int y;
};

Punkt start = {20, 40};

void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    Punkt ziel = {200, 240};
    Serial.printf("start: (%d, %d)\n", start.x, start.y);
    ziel.x = ziel.x - 20;               // ein Feld aendern
    Serial.printf("ziel:  (%d, %d)\n", ziel.x, ziel.y);

    Punkt m = mitte(start, ziel);
    Serial.printf("Mitte: (%d, %d)\n", m.x, m.y);

    Punkt kopie = start;                // kopiert beide Felder
    kopie.x = 0;
    Serial.printf("kopie.x = %d, start.x = %d\n", kopie.x, start.x);
}

void loop() {
}
```

```
30 // liefert den Punkt in der Mitte zwischen a und b
31 Punkt mitte(Punkt a, Punkt b) {
32     Punkt m;
33     m.x = (a.x + b.x) / 2;
34     m.y = (a.y + b.y) / 2;
35     return m;
36 }
```

Serieller Monitor

```
start: (20, 40)
ziel:  (180, 240)
Mitte: (100, 140)
kopie.x = 0, start.x = 20
```

- **Zeilen 2 bis 5** erfinden den Typ `Punkt`. In den geschweiften Klammern stehen seine **Felder** (auch **Elemente** genannt), jedes mit Typ und Namen. Wichtig: das **Semikolon** nach der schließenden Klammer.
- Danach ist `Punkt` ein Typ wie `int`: `Punkt start = {20, 40};` legt eine Variable an. Die Werte in `{ }` kommen der Reihe nach in die Felder.
- Mit dem **Punkt-Operator** erreichst du ein Feld: `start.x`, `ziel.y`. Ein Feld ist eine ganz normale Variable (Zeile 16).
- Eine Funktion kann eine Struktur bekommen und eine zurückgeben (Zeilen 31 bis 36). So liefert `mitte` zwei Zahlen auf einmal, was mit `int` nicht ginge.
- `=` kopiert eine Struktur mit allen Feldern (Zeile 22). Danach sind es zwei unabhängige Variablen: `kopie.x = 0` ändert `start` nicht.

Typnamen beginnen in diesem Buch mit einem Großbuchstaben (`Punkt`, `Ball`), Variablen klein (`start`, `ball`). So sieht man sofort, was was ist.

Schreibe eigene Typen **ganz oben** in den Sketch, vor `setup()` und vor alle Funktionen. Die Arduino IDE setzt die Prototypen deiner Funktionen vor die erste Funktion (Kapitel 11.4). Steht `struct Punkt` erst danach, kennt der Prototyp `Punkt mitte(Punkt a, Punkt b);` den Typ noch nicht:

k13_fehler_reihenfolge.ino

```
1  // Der eigene Typ steht zu weit unten
2  void setup() {
3      Serial.begin(115200);
4  }
5
6  void loop() {
7  }
8
9  struct Punkt {
10     int x;
```

```
    int y;
};

Punkt mitte(Punkt a, Punkt b) {
    Punkt m;
    m.x = (a.x + b.x) / 2;
    m.y = (a.y + b.y) / 2;
    return m;
}
```

Ausgabe

```
k13_fehler_reihenfolge.ino:14:1: error: 'Punkt' does not name a type
   14 | Punkt mitte(Punkt a, Punkt b) {
      | ^~~~~
Fehler beim Build: exit status 1
```

‚Punkt' does not name a type: „Punkt" ist (an dieser Stelle) kein Typ. Verwirrend ist, dass die Meldung auf Zeile 14 zeigt, obwohl `Punkt` direkt darüber steht. Der Fehler steckt im unsichtbaren Prototyp, den die IDE vor `setup()` eingefügt hat.

Ein Ball als Struktur. Alles, was einen springenden Ball ausmacht, steht jetzt beisammen:

k13_ball.ino

```
struct Ball {
    float x, y;         // Mittelpunkt
    float dx, dy;       // Schritt pro Bild
    int radius;
    uint16_t farbe;
};

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);

Ball ball = {120, 140, 2.5, -1.5, 10, ST77XX_ORANGE};

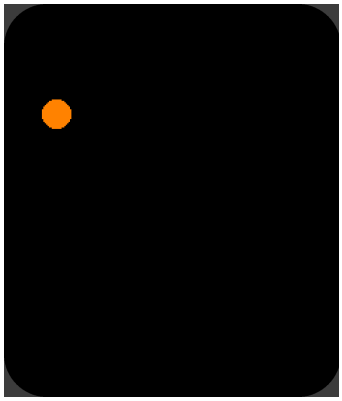
void setup() {
    tft.init(240, 280);
    tft.fillScreen(ST77XX_BLACK);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
}

void loop() {
    tft.fillCircle(ball.x, ball.y, ball.radius, ST77XX_BLACK);
    ball = bewege(ball);
    tft.fillCircle(ball.x, ball.y, ball.radius, ball.farbe);
    delay(15);
}

Ball bewege(Ball b) {
    b.x = b.x + b.dx;
```

```
    b.y = b.y + b.dy;
    if (b.x < 20 + b.radius || b.x > 220 - b.radius) {
        b.dx = -b.dx;   // an der Wand abprallen
    }
    if (b.y < 20 + b.radius || b.y > 260 - b.radius) {
        b.dy = -b.dy;
    }
    return b;
}
```

- `float x, y;` legt zwei Felder desselben Typs in einer Zeile an.
- `dx` und `dy` sind der Schritt pro Bild in x- und y-Richtung. Prallt der Ball an eine Wand, dreht sich das Vorzeichen um.
- `bewege` bekommt eine **Kopie** des Balls (Kapitel 11), ändert sie und gibt sie zurück. Erst `ball = bewege(ball);` in Zeile 29 schreibt das Ergebnis in die globale Variable zurück.
- `fillCircle` erwartet ganze Zahlen; die Kommazahlen `ball.x` und `ball.y` werden beim Aufruf abgeschnitten.



13.2 Arrays von Strukturen

Strukturen und Arrays aus Kapitel 12 passen zusammen: Ein Array aus Strukturen ist eine Tabelle, jedes Element eine Zeile.

k13_bestenliste.ino

```
// Ein Array aus Strukturen: eine Bestenliste
struct Eintrag {
    String name;
    int punkte;
};

Eintrag liste[] = {
    {"Mia", 1250},
    {"Ben", 980},
    {"Ida", 1710},
    {"Tom", 640},
};
const int ANZAHL = sizeof(liste) / sizeof(liste[0]);

void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    int beste = 0;                      // Index des bisher besten Eintrags
    for (int i = 0; i < ANZAHL; i++) {
        Serial.printf("%-5s %5d\n", liste[i].name.c_str(), liste[i].punkte);
```

```
23          if (liste[i].punkte > liste[beste].punkte) {
24              beste = i;
25          }
26      }
27      Serial.printf("Bestes Ergebnis: %s mit %d Punkten\n",
28                    liste[beste].name.c_str(), liste[beste].punkte);
29  }
30
31  void loop() {
32  }
```

Serieller Monitor

```
Mia    1250
Ben     980
Ida    1710
Tom     640
Bestes Ergebnis: Ida mit 1710 Punkten
```

- `liste[i]` ist ein ganzer `Eintrag`, `liste[i].punkte` sein Feld.
- Jeder Eintrag in den Zeilen 8 bis 11 hat eigene geschweifte Klammern. Das Komma nach dem letzten ist erlaubt; so kannst du leicht Zeilen ergänzen.
- `beste` merkt sich nicht die Punkte, sondern den **Index** des besten Eintrags. Darüber erreicht der Sketch am Ende Name und Punkte.
- `%-5s` gibt einen Text linksbündig auf 5 Zeichen aufgefüllt aus, `%5d` eine Zahl rechtsbündig. So stehen die Spalten untereinander.

Soll eine Funktion ein Element eines globalen Arrays **ändern**, gibst du ihr am einfachsten den Index mit: `void bewege(int i)` arbeitet dann direkt mit `schwarm[i].x` und so weiter. So macht es der Schwarm am Ende dieses Kapitels und der Bildschirmschoner in Kapitel 14.

13.3 `enum`: Namen für Zustände

Viele Sketches befinden sich in einem von wenigen **Zuständen**: Ein Spiel wartet auf den Start, läuft oder ist vorbei. Man könnte die Zustände als Zahlen 0, 1 und 2 speichern, aber `if (zustand == 2)` versteht niemand. `enum` (*enumeration*, Aufzählung) gibt ihnen Namen:



k13_stoppuhr.ino

```
enum Zustand {
    BEREIT,
    LAEUFT,
    ANGEHALTEN
};

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);

Zustand zustand = BEREIT;
unsigned long start = 0;        // millis() beim Start
unsigned long gemessen = 0;     // angezeigte Zeit in ms

void setup() {
    Serial.begin(115200);
    tft.init(240, 280);
    tft.fillScreen(ST77XX_BLACK);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
    pinMode(TASTE_A, INPUT_PULLUP);
    pinMode(TASTE_B, INPUT_PULLUP);
    tft.setTextColor(ST77XX_WHITE, ST77XX_BLACK);
}

void loop() {
    bool a = a_neu_gedrueckt();
    bool b = b_neu_gedrueckt();

    switch (zustand) {
        case BEREIT:
            if (a) {
                start = millis();
                zustand = LAEUFT;
            }
```

```
44              break;
45          case LAEUFT:
46              gemessen = millis() - start;
47              if (a) {
48                  zustand = ANGEHALTEN;
49                  Serial.printf("Angehalten bei %lu ms\n", gemessen);
50              }
51              break;
52          case ANGEHALTEN:
53              if (a) {
54                  start = millis() - gemessen;    // weiterlaufen lassen
55                  zustand = LAEUFT;
56              }
57              if (b) {
58                  gemessen = 0;
59                  zustand = BEREIT;
60              }
61              break;
62      }
63      zeige(gemessen, zustand);
64      delay(10);
65  }
66
67  void zeige(unsigned long ms, Zustand z) {
68      char text[12];
69      snprintf(text, sizeof(text), "%2lu.%02lu", ms / 1000, ms % 1000 / 10);
70      tft.setTextSize(6);
71      tft.setCursor(30, 100);
72      tft.print(text);
73      tft.setTextSize(2);
74      tft.setCursor(30, 180);
75      if (z == BEREIT) {
76          tft.print("Start mit A   ");
77      } else if (z == LAEUFT) {
78          tft.print("Halt mit A    ");
79      } else {
80          tft.print("A weiter, B 0 ");
81      }
82  }
```

Serieller Monitor

```
Angehalten bei 3198 ms
```

- Die Zeilen 11 bis 15 erfinden den Typ `Zustand` mit drei möglichen Werten. Intern sind das die Zahlen 0, 1 und 2, aber im Sketch schreibst du nur noch die Namen. Werte eines `enum` schreibt man üblicherweise groß.
- `switch` aus Kapitel 8 passt genau dazu: Für jeden Zustand ein `case`, darin steht, was in diesem Zustand passiert und wann er wechselt.
- So ein Sketch heißt **Zustandsautomat**: Er ist immer in genau einem Zustand, und Ereignisse wie ein Tastendruck führen von einem zum nächsten. Fast jedes Spiel ist so aufgebaut.

- Weiterlaufen nach einer Pause (Zeile 54): Die Startzeit wird so weit zurückgesetzt, dass `millis() - start` wieder die bisher gemessene Zeit ergibt.
- `zeige` bekommt den Zustand als Parameter vom Typ `Zustand` (Zeile 67).
- `%02lu` in Zeile 69 gibt eine Zahl vom Typ `unsigned long` mit mindestens zwei Stellen aus, vorn mit 0 aufgefüllt: 5 wird zu `05`.

> Neben `enum` gibt es in C++ auch `enum class Zustand { … };`. Dann schreibt man `Zustand::LAEUFT`, und der Compiler achtet strenger darauf, dass man Zustände nicht mit Zahlen verwechselt. Für den Anfang reicht `enum`.

13.4 Eine erste eigene Klasse

Erinnere dich an Kapitel 11.6: Für jede Taste brauchte die Flankenerkennung eine eigene Funktion mit eigenem `vorher`. In der Stoppuhr oben siehst du das: `a_neu_gedrueckt` und `b_neu_gedrueckt` sind fast gleich. Was fehlt, ist eine Möglichkeit, **Daten** (Pin, `vorher`, Zeitpunkt der letzten Änderung) und **Funktionen** (gedrückt? neu gedrückt?) zusammenzupacken und davon so viele Exemplare anzulegen, wie man Tasten hat. Genau das ist eine **Klasse**.

k13_klasse_taste.ino

```
 1 // Eine eigene Klasse: Taste mit Flankenerkennung und Entprellen
 2 #define TASTE_HOCH 4
 3 #define TASTE_RUNTER 2
 4 #define TASTE_A 27
 5
 6 class Taste {
 7 public:
 8     // Konstruktor: legt fest, an welchem Pin die Taste haengt
 9     Taste(int anschluss) {
10         pin = anschluss;
11     }
12
13     // in setup() aufrufen
14     void begin() {
15         pinMode(pin, INPUT_PULLUP);
16     }
17
18     // true, solange die Taste gedrueckt ist
19     bool gedrueckt() {
20         return digitalRead(pin) == LOW;
21     }
22
23     // true genau einmal pro Druck, entprellt
24     bool neu_gedrueckt() {
25         bool neu = false;
26         bool jetzt = gedrueckt();
27         if (jetzt != vorher && millis() - letzte_aenderung >= 50) {
28             letzte_aenderung = millis();
29             vorher = jetzt;
```

```
30              neu = jetzt;
31          }
32          return neu;
33      }
34
35  private:
36      int pin;
37      bool vorher = false;
38      unsigned long letzte_aenderung = 0;
39  };
40
41  Taste hoch(TASTE_HOCH);
42  Taste runter(TASTE_RUNTER);
43  Taste a(TASTE_A);
44
45  int zahl = 0;
46
47  void setup() {
48      Serial.begin(115200);
49      hoch.begin();
50      runter.begin();
51      a.begin();
52  }
53
54  void loop() {
55      if (hoch.neu_gedrueckt()) {
56          zahl = zahl + 1;
57          Serial.printf("hoch:   %d\n", zahl);
58      }
59      if (runter.neu_gedrueckt()) {
60          zahl = zahl - 1;
61          Serial.printf("runter: %d\n", zahl);
62      }
63      if (a.neu_gedrueckt()) {
64          Serial.printf("A:      %d\n", zahl);
65      }
66  }
```

Serieller Monitor

```
hoch:   1
hoch:   2
runter: 1
A:      1
```

Die Ausgabe stammt aus einem Testlauf, in dem Joystick-hoch beim ersten Druck geprellt hat. Trotzdem zählt jeder Druck nur einmal.

Die Klasse (Zeilen 6 bis 39) ist ein Bauplan, ähnlich wie `struct`:

- **`class Taste { … };`** beginnt wie eine Struktur, ebenfalls mit Semikolon am Ende.
- Unter **`private:`** (Zeilen 35 bis 38) stehen die Daten. Jede Taste hat ihren eigenen `pin`, ihr eigenes `vorher`, ihre eigene `letzte_aenderung`. „Privat“ heißt: Nur die Funktionen der Klasse selbst kommen daran. Von außen kann niemand versehentlich `vorher` verstellen.

- Unter `public:` stehen die Funktionen, die man von außen aufrufen darf. Funktionen, die zu einer Klasse gehören, heißen **Methoden**. Sie benutzen die Daten der Klasse direkt, ohne Punkt davor.
- **Der Konstruktor** (Zeilen 9 bis 11) ist eine besondere Methode: Er heißt wie die Klasse, hat keinen Rückgabetyp und läuft automatisch, wenn ein Exemplar entsteht. Hier merkt er sich den Pin.
- `neu_gedrueckt()` ist die entprellte Flankenerkennung aus Kapitel 9.3, nur dass sie jetzt mit den Daten der eigenen Taste arbeitet.

Die Objekte (Zeilen 41 bis 43): `Taste hoch(TASTE_HOCH);` baut nach dem Bauplan ein **Objekt** namens `hoch` und gibt dem Konstruktor den Pin mit. Ein Objekt nennt man auch **Instanz** der Klasse. Drei Zeilen, drei Tasten, jede mit ihrem eigenen Gedächtnis. Aufgerufen werden die Methoden mit dem Punkt: `hoch.begin()`, `runter.neu_gedrueckt()`.

> Eine **Klasse** bündelt Daten und die Funktionen (**Methoden**), die mit ihnen arbeiten. `private:` Daten nur für die Klasse, `public:` Methoden für alle. Der **Konstruktor** heißt wie die Klasse und läuft beim Anlegen eines Objekts: `Taste a(27);`.

Warum `begin()`? `pinMode` könnte doch gleich im Konstruktor stehen. Globale Objekte wie `hoch` entstehen aber, **bevor** `setup()` läuft, sehr früh beim Start des Chips. Ob der Kern zu diesem Zeitpunkt schon alles eingerichtet hat, ist nicht sicher. Deshalb merkt sich der Konstruktor nur Werte, und alles, was die Hardware anfasst, kommt in eine Methode, die du in `setup()` aufrufst. Fast alle Arduino-Bibliotheken machen es so.

Das kennst du schon. Seit Kapitel 3 benutzt du Objekte:

Zeile	Klasse	Objekt	Konstruktor bekommt
`Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);`	`Adafruit_ST7789`	`tft`	die drei Pins
`Adafruit_NeoPixel rgb(1, RGB_LED, …);`	`Adafruit_NeoPixel`	`rgb`	Anzahl, Pin, Bauart
`Taste a(TASTE_A);`	`Taste`	`a`	den Pin

`tft.init(240, 280)` und `rgb.begin()` sind genau so ein `begin()`: Der Konstruktor merkt sich nur die Pins, erst die Methode in `setup()` spricht mit der Hardware. Auch `String` aus Kapitel 12 ist eine Klasse, und `gruss.length()` ist eine Methode. `Serial` ist ein Objekt, das der Kern schon für dich angelegt hat.

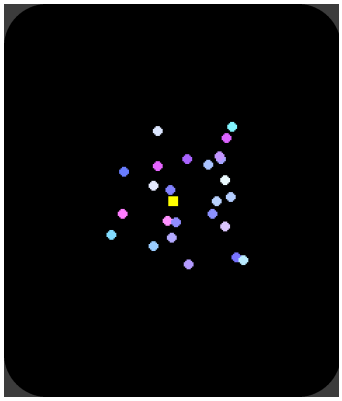
> Eine Klasse kann auch in eigene Dateien wandern, `Taste.h` und `Taste.cpp`, und so in vielen Sketches benutzt werden. Dann ist sie eine kleine **Bibliothek**. Wie das geht, zeigt Kapitel 20 am Beispiel des Neigungssensors.

Ausprobieren: Ein Schwarm

25 Fische schwimmen durcheinander und folgen dem gelben Futter, das du mit dem Joystick bewegst. Drückst du A, erschrickt der Schwarm und stiebt auseinander, bevor er sich wieder sammelt.

k13_schwarm.ino

```
14 #define ANZAHL 25
15 #define TEMPO 2.5           // hoechstes Tempo
16
17 struct Fisch {
18     float x, y;
19     float dx, dy;
20     uint16_t farbe;
21 };
```



Die Klasse `Taste` aus 13.4 steht in den Zeilen 23 bis 48, ohne Kommentare, sonst gleich. Danach geht es so weiter:

k13_schwarm.ino

```
50 Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
51 Taste hoch(TASTE_HOCH);
52 Taste runter(TASTE_RUNTER);
53 Taste links(TASTE_LINKS);
54 Taste rechts(TASTE_RECHTS);
55 Taste a(TASTE_A);
56
57 Fisch schwarm[ANZAHL];
58 int futter_x = 120;
59 int futter_y = 140;
60
61 void setup() {
62     tft.init(240, 280);
63     tft.fillScreen(ST77XX_BLACK);
64     pinMode(TFT_LICHT, OUTPUT);
65     digitalWrite(TFT_LICHT, HIGH);
66     hoch.begin();
67     runter.begin();
68     links.begin();
69     rechts.begin();
70     a.begin();
71     for (int i = 0; i < ANZAHL; i++) {
72         schwarm[i].x = random(30, 210);
73         schwarm[i].y = random(30, 250);
```

```
        schwarm[i].dx = 0;
        schwarm[i].dy = 0;
        schwarm[i].farbe = tft.color565(random(100, 256), random(100, 256), 255);
    }
}

void loop() {
    // Futter mit dem Joystick bewegen
    tft.fillRect(futter_x - 3, futter_y - 3, 7, 7, ST77XX_BLACK);
    if (links.gedrueckt() && futter_x > 30) {
        futter_x = futter_x - 3;
    }
    if (rechts.gedrueckt() && futter_x < 210) {
        futter_x = futter_x + 3;
    }
    if (hoch.gedrueckt() && futter_y > 30) {
        futter_y = futter_y - 3;
    }
    if (runter.gedrueckt() && futter_y < 250) {
        futter_y = futter_y + 3;
    }
    if (a.neu_gedrueckt()) {
        erschrecken();
    }

    for (int i = 0; i < ANZAHL; i++) {
        tft.fillCircle(schwarm[i].x, schwarm[i].y, 3, ST77XX_BLACK);
        bewege(i);
        tft.fillCircle(schwarm[i].x, schwarm[i].y, 3, schwarm[i].farbe);
    }
    tft.fillRect(futter_x - 3, futter_y - 3, 7, 7, ST77XX_YELLOW);
    delay(10);
}

// Fisch Nummer i einen Schritt weiter: zum Futter hin, etwas zufaellig
void bewege(int i) {
    float zufall_x = random(-10, 11) / 100.0;
    float zufall_y = random(-10, 11) / 100.0;
    schwarm[i].dx = schwarm[i].dx + (futter_x - schwarm[i].x) * 0.002 + zufall_x;
    schwarm[i].dy = schwarm[i].dy + (futter_y - schwarm[i].y) * 0.002 + zufall_y;
    schwarm[i].dx = constrain(schwarm[i].dx, -TEMPO, TEMPO);
    schwarm[i].dy = constrain(schwarm[i].dy, -TEMPO, TEMPO);
    schwarm[i].x = constrain(schwarm[i].x + schwarm[i].dx, 25, 215);
    schwarm[i].y = constrain(schwarm[i].y + schwarm[i].dy, 25, 255);
}

// jeder Fisch bekommt eine zufaellige schnelle Richtung
void erschrecken() {
    for (int i = 0; i < ANZAHL; i++) {
        schwarm[i].dx = random(-30, 31) / 10.0;
        schwarm[i].dy = random(-30, 31) / 10.0;
    }
}
```

- Jeder Fisch ist eine Struktur `Fisch`, alle 25 stehen im Array `schwarm` (Zeile 57). `ANZAHL` steht oben: Ändere sie, und der Schwarm wird größer oder kleiner.
- Fünf Objekte der Klasse `Taste` (Zeilen 51 bis 55). Das Futter bewegt sich, **solange** eine Richtung gedrückt ist (`gedrueckt`), A wirkt **einmal pro Druck** (`neu_gedrueckt`). Beides kann die Klasse.
- `bewege(i)` zieht jeden Fisch ein wenig zum Futter hin: Je weiter weg er ist, desto stärker ändert sich seine Richtung (`* 0.002`). Dazu kommt ein kleiner Zufall, damit nicht alle gleich schwimmen.
- `constrain(wert, unten, oben)` ist eine Arduino-Funktion, die einen Wert auf einen Bereich begrenzt: Ist er kleiner als `unten`, liefert sie `unten`, ist er größer als `oben`, liefert sie `oben`, sonst den Wert selbst. So werden die Fische nie schneller als `TEMPO` und verlassen nie den sichtbaren Bereich.
- Weil die Fische nicht bremsen, schießen sie über das Futter hinaus und kehren um. So entsteht das typische Hin und Her eines Schwarms.
- `erschrecken()` gibt jedem Fisch eine zufällige schnelle Richtung.
- Jeder Fisch wird einzeln gelöscht und neu gemalt. Das flackert, besonders bei vielen Fischen. Kapitel 14 zeigt, wie man ein ganzes Bild erst im Speicher malt und dann auf einmal anzeigt.

Zum Verändern: Gib den Fischen verschiedene Größen; dafür braucht `Fisch` ein Feld `radius`. Lass B das Futter an eine zufällige Stelle springen.

Übungsaufgaben

Aufgabe 1 ⭐ Rechtecke Lege eine Struktur `Rechteck` mit Position, Breite, Höhe und Farbe an und ein Array mit drei Rechtecken. Schreibe die Funktionen `int flaeche(Rechteck r)` und `void zeichne(Rechteck r)`, zeichne alle drei und gib ihre Flächen im seriellen Monitor aus.

Aufgabe 2 ⭐ Ampelphasen Lege ein `enum Phase` mit den Werten `ROT`, `ROT_GELB`, `GRUEN` und `GELB` an. Schreibe eine Funktion `Phase naechste(Phase p)`, die mit `switch` die folgende Phase liefert, und eine Funktion, die den Namen einer Phase als `String` zurückgibt. Gib damit sechs Phasen nacheinander aus.

Aufgabe 3 ⭐⭐ Lang oder kurz? Erweitere die Klasse `Taste` so, dass sie auch das Loslassen meldet und sagt, wie lange die Taste gedrückt war. Gib bei A aus, ob es ein kurzer oder ein langer Druck (ab einer Sekunde) war. Tipp: Eine Methode `pruefen()`, die in jedem Durchlauf von `loop()` einmal aufgerufen wird, merkt sich in privaten Variablen, was passiert ist. Die anderen Methoden geben das nur noch zurück.

Aufgabe 4 ⭐⭐ Eine Klasse für LEDs Schreibe eine Klasse `Led` mit Konstruktor (Pin), `begin()`, `an()`, `aus()` und `umschalten()`. Lege ein Array aus drei `Led`-Objekten an und baue damit ein Lauflicht. Tipp: `Led leds[3] = {Led(14), Led(13), Led(12)};`

Aufgabe 5 ⭐⭐⭐ Der Hai Ändere den Schwarm: Statt Futter steuerst du einen roten Hai. Kommt er einem Fisch näher als 50 Pixel, schwimmt der Fisch von ihm weg. Sonst schwimmen die Fische frei herum und prallen an den Rändern ab. Tipp: Den Abstand musst du nicht ausrechnen. Es reicht zu prüfen, ob `ax * ax + ay * ay < 50 * 50` ist, wobei `ax` und `ay` die Abstände in x- und y-Richtung sind (Satz des Pythagoras ohne Wurzel).

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- **`struct Name { typ feld; … };`** bündelt Werte zu einem eigenen Typ. Zugriff mit dem Punkt: `ball.x`. `=` kopiert alle Felder. Eigene Typen gehören ganz oben in den Sketch.
- Ein **Array aus Strukturen** ist eine Tabelle: `liste[i].punkte`.
- **`enum Name { A, B, C };`** gibt Zuständen Namen; dazu passt `switch`.
- Eine **Klasse** bündelt private Daten und öffentliche **Methoden**. Der **Konstruktor** heißt wie die Klasse und läuft beim Anlegen des **Objekts**. Hardware erst in `begin()` anfassen.
- `tft`, `rgb`, `Serial` und jeder `String` sind Objekte von Klassen.
- `constrain(wert, unten, oben)` begrenzt einen Wert auf einen Bereich.

KAPITEL 14

Flimmerfrei: `GFXcanvas16` und `drawRGBBitmap`

Worum geht's? In Kapitel 7 hat deine Animation geflackert. Jetzt beseitigst du das Flackern endgültig: Du zeichnest jedes Bild zuerst unsichtbar im Arbeitsspeicher und schickst es dann auf einmal zum Display. Du misst, wie lange das dauert, machst die Übertragung schneller und lässt Bälle gleichmäßig über den Bildschirm fliegen und abprallen. Das ist die Grundlage für alle Spiele in Teil III.

14.1 Warum es flackert

Dieser Sketch lässt einen Ball über den Bildschirm fliegen. Er macht es so, wie du es bisher kennst: Bildschirm schwarz füllen, Ball zeichnen, ein Stück weiter, von vorn.

k14_flackern.ino

```
// Ein Ball fliegt ueber den Bildschirm - direkt gezeichnet, er flackert
#include <Adafruit_ST7789.h>

#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);

int x = 30;

void setup() {
    tft.init(240, 280);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
}

void loop() {
    tft.fillScreen(ST77XX_BLACK);
    tft.fillCircle(x, 140, 15, ST77XX_YELLOW);
    x = x + 4;
    if (x > 210) {
        x = 30;
    }
}
```

Auf dem PicoBoy flackert der Ball stark. Woran liegt das? Jeder Befehl an `tft` wird **sofort** über die Leitung zum Display geschickt, und das Display zeigt ihn sofort an. `fillScreen` muss dafür alle 240 · 280 = 67 200 Pixel übertragen. Das dauert etwa **43 Millisekunden** (warum genau so lange,

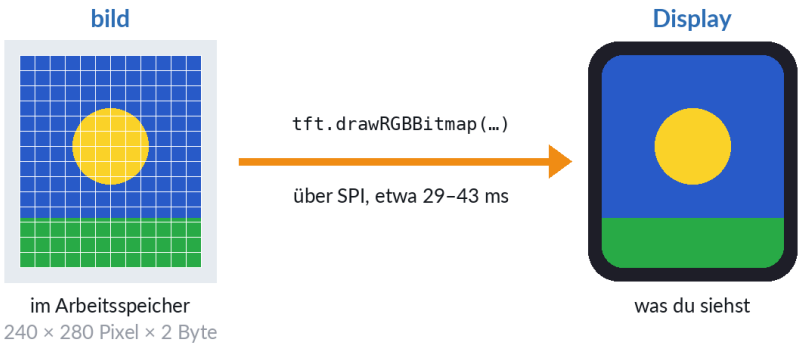
steht in Abschnitt 14.4). Während dieser Zeit wird das Display von oben nach unten schwarz, und der Ball verschwindet. Erst danach zeichnet `fillCircle` ihn neu. Der Ball ist also ungefähr die Hälfte der Zeit gar nicht zu sehen:



Kleine Tricks helfen nur ein wenig, zum Beispiel nur den alten Ball schwarz zu übermalen statt des ganzen Bildschirms (so wie der Punkt in Kapitel 9.4). Sobald sich mehrere Dinge bewegen und überlappen, flackert es trotzdem. Die richtige Lösung zeigt der nächste Abschnitt.

14.2 Ein Bild im Speicher: `GFXcanvas16`

Die Idee: Du zeichnest nicht direkt auf das Display, sondern in ein Bild im **Arbeitsspeicher** (englisch *RAM*) des PicoBoy. Dort ist es unsichtbar. Erst wenn das ganze Bild fertig ist, schickst du es mit einem einzigen Befehl zum Display. Das Display ersetzt dann das alte Bild durch das neue, ohne dass dazwischen etwas Schwarzes zu sehen ist.



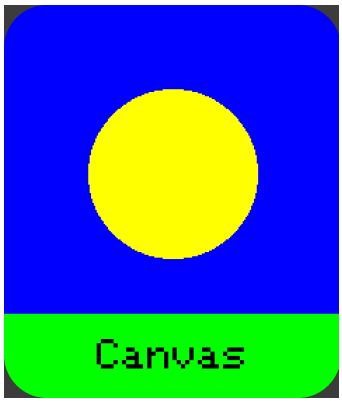
Adafruit GFX hat dafür fertige Klassen. Ein solches Bild im Speicher heißt **Canvas** (englisch für Leinwand). Für das Display des PicoBoy nimmst du `GFXcanvas16`: Die 16 bedeutet 16 Bit pro Pixel, also genau die Farben aus Kapitel 6 (RGB565).

k14_canvas.ino

```
1 // Erst im Speicher zeichnen, dann alles auf einmal anzeigen
2 #include <Adafruit_ST7789.h>
3
4 #define TFT_CS 10
```

```
 5 #define TFT_DC 8
 6 #define TFT_RST 9
 7 #define TFT_LICHT 26
 8
 9 Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
10 GFXcanvas16 bild(240, 280);         // ein Bild im Arbeitsspeicher
11
12 void setup() {
13     tft.init(240, 280);
14     pinMode(TFT_LICHT, OUTPUT);
15     digitalWrite(TFT_LICHT, HIGH);
16
17     // dieselben Befehle wie fuer tft, nur auf bild
18     bild.fillScreen(ST77XX_BLUE);
19     bild.fillCircle(120, 120, 60, ST77XX_YELLOW);
20     bild.fillRect(0, 220, 240, 60, ST77XX_GREEN);
21     bild.setTextColor(ST77XX_BLACK);
22     bild.setTextSize(3);
23     bild.setCursor(66, 238);
24     bild.print("Canvas");
25
26     // das fertige Bild zum Display schicken
27     tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
28 }
29
30 void loop() {
31 }
```

Zeile 10 legt den Canvas an: ein Objekt mit dem Namen `bild`, 240 Pixel breit und 280 hoch, so groß wie das Display. Wie `tft` steht es oben, vor `setup()`.



Zeile 18 bis 24 zeichnen. Ein Canvas kennt **alle** Zeichenbefehle aus Kapitel 3 und 6: `fillScreen`, `fillCircle`, `drawLine`, `setCursor`, `print`, Schriften … Du schreibst nur `bild.` statt `tft.` davor. Dabei erscheint noch nichts auf dem Display.

Zeile 27 schickt das fertige Bild zum Display:

```
tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
```

`drawRGBBitmap` zeichnet ein Bild aus lauter einzelnen Farbwerten (eine **Bitmap**) an die Stelle (0, 0), 240 breit und 280 hoch. `bild.getBuffer()` sagt ihr, wo im Speicher die Pixel des Canvas liegen. Genau genommen liefert `getBuffer()` einen **Zeiger** auf die Pixel; was das ist, erklärt Kapitel 19. Bis dahin schreibst du diese Zeile einfach immer so ab.

> Mit Canvas gilt: **erst alles in `bild` zeichnen, dann einmal `tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);`**. Fehlt diese Zeile, bleibt das Display, wie es war.

Wie viel Speicher braucht das? Jedes Pixel braucht 16 Bit, also 2 Byte.
240 · 280 Pixel · 2 Byte = 134 400 Byte, rund **134 KB**. Der RP2350 im PicoBoy hat 520 KB Arbeitsspeicher; ein Canvas in Bildschirmgröße passt also gut hinein, und es bleibt reichlich Platz für deinen Sketch. Auf einem klassischen Arduino Uno mit 2 KB Arbeitsspeicher wäre das undenkbar.

> Lege den Canvas immer **oben bei den Objekten** an, nie innerhalb von `loop()`. Er soll einmal beim Start entstehen und dann bleiben. Die 134 KB jedes Mal neu zu besorgen, wäre langsam und unnötig.

14.3 Animation ohne Flackern

Jetzt bekommt der Ball aus 14.1 einen Canvas. In `loop()` stehen nur noch drei Schritte: **löschen, zeichnen, übertragen**.

k14_ball.ino

```
1  // Derselbe Ball wie in k14_flackern, aber ohne Flackern
2  #include <Adafruit_ST7789.h>
3
4  #define TFT_CS 10
5  #define TFT_DC 8
6  #define TFT_RST 9
7  #define TFT_LICHT 26
8
9  Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
10 GFXcanvas16 bild(240, 280);
11
12 int x = 30;
13
14 void setup() {
15     tft.init(240, 280);
16     pinMode(TFT_LICHT, OUTPUT);
17     digitalWrite(TFT_LICHT, HIGH);
18 }
19
20 void loop() {
21     bild.fillScreen(ST77XX_BLACK);
22     bild.fillCircle(x, 140, 15, ST77XX_YELLOW);
23     tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
24     x = x + 4;
25     if (x > 210) {
26         x = 30;
27     }
28 }
```

Das Löschen mit `bild.fillScreen` (Zeile 21) geschieht im Speicher und ist unsichtbar. Das Display sieht nur fertige Bilder, und auf jedem ist der Ball zu sehen (siehe die untere Hälfte der Abbildung in 14.1). Probiere beide Sketches nacheinander auf dem PicoBoy aus: Der Unterschied ist deutlich.

Das ist das Muster für alle bewegten Sketches in den nächsten Kapiteln:

```
void loop() {
    // 1. rechnen: Tasten lesen, Dinge bewegen, Treffer pruefen
    // 2. zeichnen: bild.fillScreen(...), dann alles andere auf bild
    // 3. uebertragen:
    tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
}
```

Weil du jedes Bild vollständig neu zeichnest, musst du dir nie merken, wo etwas vorher war, und nichts mehr „wegradieren“.

14.4 Wie lange dauert ein Bild?

Die **Bildrate** gibt an, wie viele Bilder pro Sekunde erscheinen (englisch *frames per second*, kurz fps). Kinofilme haben 24, viele Spiele 30 oder 60. Wie viele schafft der PicoBoy? Das misst dieser Sketch:

k14_messen.ino

```
// Wie lange dauert die Uebertragung eines Bildes? Wie viele Bilder pro Sekunde?
#include <Adafruit_ST7789.h>

#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
GFXcanvas16 bild(240, 280);

int x = 30;
int anzahl = 0;                    // Bilder in der laufenden Sekunde
unsigned long sekunde_start = 0;

void setup() {
    Serial.begin(115200);
    tft.init(240, 280);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
    sekunde_start = millis();
}

void loop() {
    bild.fillScreen(ST77XX_BLACK);
    bild.fillCircle(x, 140, 15, ST77XX_YELLOW);
    x = x + 4;
```

```
28      if (x > 210) {
29          x = 30;
30      }
31
32      unsigned long start = micros();
33      tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
34      unsigned long dauer = micros() - start;
35      anzahl = anzahl + 1;
36
37      if (millis() - sekunde_start >= 1000) {
38          Serial.printf("Uebertragung: %lu us, %d Bilder pro Sekunde\n", dauer,
              anzahl);
39          anzahl = 0;
40          sekunde_start = millis();
41      }
42  }
```

Serieller Monitor

```
Uebertragung: 43012 us, 24 Bilder pro Sekunde
Uebertragung: 43012 us, 24 Bilder pro Sekunde
Uebertragung: 43012 us, 24 Bilder pro Sekunde
Uebertragung: 43012 us, 24 Bilder pro Sekunde
```

Zeile 32 bis 34 messen die Zeit mit `micros()`. Das ist wie `millis()` aus Kapitel 7, nur zählt es **Mikrosekunden** (µs): 1 ms = 1000 µs. So lassen sich auch kurze Abläufe genau messen. Die Zeit vor der Übertragung merkt sich `start`; nach der Übertragung ist `micros() - start` die Dauer. Im seriellen Monitor steht `us`, weil wir dort keine Sonderzeichen wie µ benutzen.

Zeile 35 bis 41 zählen die Bilder. Ist eine Sekunde vorbei, gibt der Sketch die Zahl aus und fängt von vorn an.

Ein Bild dauert also gut 43 ms, das ergibt etwa 23 Bilder pro Sekunde. Das Zeichnen in den Canvas geht im Vergleich dazu sehr schnell; die Bildrate bestimmt fast allein die Übertragung.

> Die Ausgaben in diesem Buch stammen aus einem Prüflauf auf dem PC, der die Übertragung zum Display genau nachrechnet, das Rechnen im Sketch aber kostenlos erledigt. Auf dem PicoBoy kommt für Zeichnen und Rechnen noch etwas Zeit dazu, und die Übertragung selbst dauert dort etwa ein Zehntel länger als gerechnet. Am Gerät gemessen sind es gut 47 statt 43 ms, also 21 statt 23 Bilder pro Sekunde, und mit `setSPISpeed` aus 14.5 knapp 32 statt 29 ms, also 31 statt 35 Bilder pro Sekunde.

Woher kommen die 43 ms? Das Display hängt an einer Leitung namens **SPI** (Kapitel 23 erklärt sie genau). Darüber geht ein Bit nach dem anderen, im Takt einer Uhr. Ohne weitere Angabe läuft dieser Takt mit **25 MHz**, also 25 Millionen Bit pro Sekunde. Ein Bild hat
67 200 Pixel · 16 Bit = 1 075 200 Bit. Geteilt durch 25 000 000 Bit pro Sekunde ergibt das 0,043 Sekunden, also 43 ms.

14.5 Schneller übertragen: `setSPISpeed`

Mit `tft.setSPISpeed(…)` stellst du den SPI-Takt selbst ein. Die Zeile gehört **nach** `tft.init`, denn `init` stellt den Takt neu ein. Sonst ist der Sketch derselbe wie `k14_messen`:

k14_schneller.ino

```
16 void setup() {
17     Serial.begin(115200);
18     tft.init(240, 280);
19     tft.setSPISpeed(62500000);      // nach init: Wunsch 62,5 MHz, real 37,5 MHz
20     pinMode(TFT_LICHT, OUTPUT);
21     digitalWrite(TFT_LICHT, HIGH);
22     sekunde_start = millis();
23 }
```

Serieller Monitor

```
Uebertragung: 28676 us, 35 Bilder pro Sekunde
Uebertragung: 28676 us, 35 Bilder pro Sekunde
Uebertragung: 28676 us, 35 Bilder pro Sekunde
Uebertragung: 28676 us, 35 Bilder pro Sekunde
Uebertragung: 28676 us, 35 Bilder pro Sekunde
```

Statt 43 ms dauert ein Bild jetzt knapp 29 ms, das sind etwa 35 Bilder pro Sekunde. Aber warum? Gewünscht waren 62,5 MHz, und 1 075 200 Bit / 62 500 000 Bit pro Sekunde wären nur 17 ms.

Die Teilerstufen. Der Prozessor läuft mit 150 MHz. Der SPI-Baustein kann höchstens die Hälfte davon, 75 MHz, und sonst nur 75 MHz geteilt durch eine ganze Zahl:

Teiler	1	2	3	4	5	6
SPI-Takt	75 MHz	37,5 MHz	25 MHz	18,75 MHz	15 MHz	12,5 MHz
ein ganzes Bild	14 ms	29 ms	43 ms	57 ms	72 ms	86 ms

Von deinem Wunsch nimmt er die **nächstkleinere** Stufe. Der Grund für die 25 MHz ohne `setSPISpeed`: Die Bibliothek wünscht sich von sich aus 32 MHz, und die nächstkleinere Stufe ist 25 MHz. Aus 62,5 MHz wird 37,5 MHz.

Und warum nicht gleich 75 MHz? Der Displaychip ist nur für einen Takt bis etwa 62,5 MHz gebaut. Mit 75 MHz kann es funktionieren, muss aber nicht; dann erscheinen falsche Farben oder Streifen. In diesem Buch steht deshalb ab jetzt in jedem bewegten Sketch:

```
tft.setSPISpeed(62500000);      // nach tft.init: real 37,5 MHz
```

> Die Teilerstufen gelten für den voreingestellten Prozessortakt von 150 MHz. Wer ihn in der Arduino IDE unter **Werkzeuge → CPU Speed** ändert, bekommt andere Stufen. Miss dann einfach mit `k14_messen` nach.

14.6 Gleichmäßig bewegen und abprallen

Bisher bewegt sich der Ball in jedem Durchlauf von `loop()` um 4 Pixel. Wie schnell er fliegt, hängt also davon ab, wie lange ein Durchlauf dauert: mit `setSPISpeed` schneller als ohne, mit vielen Bällen langsamer als mit einem. Für ein Spiel ist das ungünstig. Besser ist eine **feste Bildrate**: Ein neues Bild gibt es genau alle 33 ms, also etwa 30-mal pro Sekunde. Dafür nimmst du den `millis`-Takt aus Kapitel 7.

Dazu bekommt der Ball eine Richtung in x **und** in y, und er prallt von allen vier Rändern ab:

k14_abprallen.ino

```
// Ein Ball prallt von allen vier Raendern ab, 30 Bilder pro Sekunde
#include <Adafruit_ST7789.h>

#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26
#define RADIUS 12
#define TAKT 33                     // ms pro Bild: etwa 30 Bilder pro Sekunde

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
GFXcanvas16 bild(240, 280);

float x = 60;
float y = 50;
float dx = 3.0;                     // Pixel pro Bild nach rechts
float dy = 2.5;                     // Pixel pro Bild nach unten
unsigned long letztes_bild = 0;

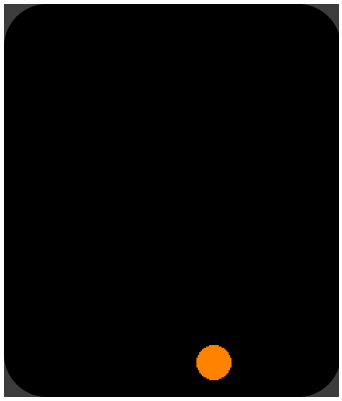
void setup() {
    tft.init(240, 280);
    tft.setSPISpeed(62500000);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
}

void loop() {
    if (millis() - letztes_bild >= TAKT) {
        letztes_bild = millis();

        // bewegen
        x = x + dx;
        y = y + dy;

        // abprallen: zurueck an den Rand, Richtung umdrehen
        if (x < RADIUS) {
            x = RADIUS;
            dx = -dx;
        }
        if (x > 239 - RADIUS) {
            x = 239 - RADIUS;
```

```
42              dx = -dx;
43          }
44          if (y < RADIUS) {
45              y = RADIUS;
46              dy = -dy;
47          }
48          if (y > 279 - RADIUS) {
49              y = 279 - RADIUS;
50              dy = -dy;
51          }
52
53          // zeichnen
54          bild.fillScreen(ST77XX_BLACK);
55          bild.fillCircle(x, y, RADIUS, ST77XX_ORANGE);
56          tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
57      }
58  }
```



- **Zeile 14 bis 17:** Position und Geschwindigkeit sind `float`. So kann der Ball auch 2,5 Pixel pro Bild fliegen. `dx` und `dy` sagen, wie viele Pixel er pro Bild nach rechts und nach unten fliegt. Ein negativer Wert heißt: nach links oder nach oben.
- **Zeile 28 und 29:** Erst wenn seit dem letzten Bild 33 ms vergangen sind, entsteht ein neues. Dauert das Übertragen 29 ms, wartet `loop()` die restlichen 4 ms einfach ab.
- **Zeile 36 bis 51: Abprallen.** Der Ball ist ein Kreis um (x, y). Er berührt den linken Rand, wenn `x` kleiner als der Radius ist, den rechten, wenn `x` größer als `239 - RADIUS` ist. Dann dreht sich die Richtung um: Aus `dx = 3` wird `dx = -3` und umgekehrt. Zusätzlich setzt der Sketch den Ball genau an den Rand zurück. Ohne diese Zeile könnte er bei schrägen Bahnen ein Stück im Rand stecken, und im nächsten Bild würde sich die Richtung schon wieder umdrehen. Er bliebe dann zitternd am Rand hängen.
- **Zeile 55:** `fillCircle` erwartet ganze Zahlen. Die Nachkommastellen von `x` und `y` werden beim Aufruf einfach abgeschnitten.

Ohne `setSPISpeed` dauert die Übertragung allein schon 43 ms. Dann schafft der Sketch keine 30 Bilder pro Sekunde, sondern nur etwa 23, und alles bewegt sich langsamer. Die feste Bildrate funktioniert nur, wenn ein Bild schneller fertig ist als der Takt.

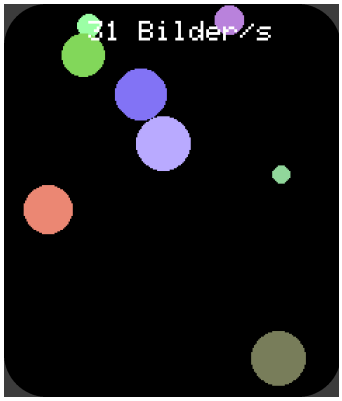
Ausprobieren: Ein Bildschirmschoner

Acht Bälle in zufälligen Farben und Größen fliegen durcheinander. Oben steht, wie viele Bilder pro Sekunde wirklich erscheinen. Jeder Ball ist eine Struktur aus Kapitel 13, alle zusammen stehen in einem Array:

k14_baelle.ino

```
11 struct Ball {
12     float x, y;
13     float dx, dy;
14     int radius;
15     uint16_t farbe;
16 };
17
18 Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
19 GFXcanvas16 bild(240, 280);
20
21 Ball baelle[ANZAHL];
22 unsigned long letztes_bild = 0;
23 unsigned long sekunde_start = 0;
24 int anzahl = 0;                    // Bilder in dieser Sekunde
25 int bildrate = 0;                  // Bilder in der letzten Sekunde
26
27 void setup() {
28     tft.init(240, 280);
29     tft.setSPISpeed(62500000);
30     pinMode(TFT_LICHT, OUTPUT);
31     digitalWrite(TFT_LICHT, HIGH);
32     for (int i = 0; i < ANZAHL; i++) {
33         baelle[i].radius = random(6, 20);
34         baelle[i].x = random(30, 210);
35         baelle[i].y = random(40, 240);
36         baelle[i].dx = random(-40, 41) / 10.0;
37         baelle[i].dy = random(-40, 41) / 10.0;
38         baelle[i].farbe = tft.color565(random(80, 256), random(80, 256),
             random(80, 256));
39     }
40 }
```

- Die Struktur `Ball` (Zeile 11 bis 16) fasst alles zusammen, was einen Ball ausmacht.
- Zeile 36 und 37: `random(-40, 41)` liefert eine ganze Zahl von −40 bis 40. Geteilt durch `10.0` (mit Komma!) wird daraus eine Kommazahl von −4,0 bis 4,0. Geteilt durch `10` ohne Komma käme eine ganze Zahl heraus (Kapitel 5).



k14_baelle.ino

```
42 // Ball Nummer i einen Schritt weiter bewegen und an den
     Raendern abprallen
43 void bewege(int i) {
44     baelle[i].x = baelle[i].x + baelle[i].dx;
45     baelle[i].y = baelle[i].y + baelle[i].dy;
46     int r = baelle[i].radius;
47     if (baelle[i].x < r || baelle[i].x > 239 - r) {
48         baelle[i].x = constrain(baelle[i].x, r, 239 -
             r);
49         baelle[i].dx = -baelle[i].dx;
50     }
51     if (baelle[i].y < r || baelle[i].y > 279 - r) {
52         baelle[i].y = constrain(baelle[i].y, r, 279 -
             r);
53         baelle[i].dy = -baelle[i].dy;
54     }
55 }
56
57 void loop() {
58     if (millis() - letztes_bild >= TAKT) {
59         letztes_bild = millis();
60         bild.fillScreen(ST77XX_BLACK);
61         for (int i = 0; i < ANZAHL; i++) {
62             bewege(i);
63             bild.fillCircle(baelle[i].x, baelle[i].y,
               baelle[i].radius, baelle[i].farbe);
64         }
65         bild.setTextColor(ST77XX_WHITE);
66         bild.setTextSize(2);
67         bild.setCursor(60, 12);
68         bild.printf("%d Bilder/s", bildrate);
69         tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240,
             280);
70         anzahl = anzahl + 1;
71     }
72     if (millis() - sekunde_start >= 1000) {
73         bildrate = anzahl;
74         anzahl = 0;
75         sekunde_start = millis();
76     }
77 }
```

- `bewege(i)` (Zeile 43 bis 55) ist das Abprallen aus 14.6, nur für Ball Nummer `i`. Statt vier `if` nimmt sie `constrain(wert, min, max)`: Diese Arduino-Funktion liefert `wert`, aber nie weniger als `min` und nie mehr als `max`.
- Zeile 72 bis 76 zählen die Bilder pro Sekunde wie in `k14_messen`. Die Zahl der letzten Sekunde steht in `bildrate` und wird oben angezeigt.

Zum Verändern: Nimm 30 Bälle statt 8. Bleibt die Bildrate bei 30? Lass die Zeile mit `setSPISpeed` weg. Was passiert mit der Bildrate und mit dem Tempo der Bälle?

Übungsaufgaben

Aufgabe 1 ⭐ Teilerstufen Der Sketch `k14_aufgabe1` misst die Übertragung eines ganzen Bildes für vier verschiedene Wünsche an `setSPISpeed`: 20, 32, 40 und 62,5 MHz. Sage für jeden Wunsch den echten SPI-Takt und die Dauer voraus, bevor du ihn laufen lässt. Welche beiden Wünsche ergeben dieselbe Zeit, und warum?

k14_aufgabe1.ino

```
12 unsigned long wunsch[4] = { 20000000, 32000000, 40000000, 62500000 };
13
14 void setup() {
15     Serial.begin(115200);
16     tft.init(240, 280);
17     pinMode(TFT_LICHT, OUTPUT);
18     digitalWrite(TFT_LICHT, HIGH);
19     bild.fillScreen(ST77XX_BLUE);
20
21     for (int i = 0; i < 4; i++) {
22         tft.setSPISpeed(wunsch[i]);
23         unsigned long start = micros();
24         tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
25         unsigned long dauer = micros() - start;
26         Serial.printf("Wunsch %lu Hz: %lu us\n", wunsch[i], dauer);
27     }
28 }
```

Aufgabe 2 ⭐ Bunte Abpraller Ändere `k14_abprallen` so, dass der Ball bei jedem Abprallen eine neue, zufällige Farbe bekommt. Tipp: eine Variable `farbe` und eine Funktion `neue_farbe()`.

Aufgabe 3 ⭐⭐ Der Punkt ohne Flackern Baue den steuerbaren Punkt aus Kapitel 9.4 auf einen Canvas um. Zeichne zusätzlich einen blauen Rahmen, in dem sich der Punkt bewegt. Was kannst du im Vergleich zum alten Sketch weglassen?

Aufgabe 4 ⭐⭐ Springender Ball Ein Ball fällt herunter, springt vom Boden ab und wird dabei jedes Mal etwas niedriger. So geht es: In jedem Bild wird `dy` um 0,5 größer (die Schwerkraft zieht nach unten). Trifft der Ball den Boden, wird `dy = -dy * 0.8`: Er fliegt mit 80 % des Tempos wieder nach oben. Lass ihn außerdem langsam nach rechts fliegen und von den Seiten abprallen.

Aufgabe 5 ⭐⭐⭐ Laufschrift im Streifen Ein Canvas muss nicht so groß wie das Display sein. Lege einen Canvas `GFXcanvas16 streifen(240, 40);` an und lass darin einen langen Text von rechts nach links laufen. Übertrage nur den Streifen nach y = 120: `tft.drawRGBBitmap(0, 120, streifen.getBuffer(), 240, 40);`. Der Rest des Bildschirms wird nur einmal in `setup()` direkt auf `tft` gezeichnet. Miss, wie lange die Übertragung des Streifens dauert, und rechne nach. Tipp: `streifen.setTextWrap(false);` verhindert, dass ein zu langer Text in die nächste Zeile umbricht.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Direkt auf `tft` gezeichnet ist sofort sichtbar, auch das Löschen. Das ergibt Flackern.
- `GFXcanvas16 bild(240, 280);` ist ein Bild im Arbeitsspeicher (134 KB). Es kennt alle Zeichenbefehle: `bild.fillScreen(…)`, `bild.print(…)` …
- Sichtbar wird es mit `tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);`. `getBuffer()` liefert einen Zeiger auf die Pixel (Kapitel 19).
- Muster für jedes Bild: rechnen, `bild` ganz neu zeichnen, übertragen.
- `micros()` misst in Mikrosekunden. Ein ganzes Bild dauert ohne `setSPISpeed` 43 ms (25 MHz), mit `tft.setSPISpeed(62500000)` nach `tft.init` 29 ms (37,5 MHz); gerechnet, am Gerät gemessen etwa 47 und 32 ms. Möglich sind nur 75 MHz geteilt durch eine ganze Zahl.
- Feste Bildrate mit dem `millis`-Takt, zum Beispiel alle 33 ms.
- Abprallen: an den Rand zurücksetzen und `dx = -dx` bzw. `dy = -dy`.

KAPITEL 15

Bilder und Sprites

Worum geht's? Spiele leben von Figuren: einer Rakete, einem Fisch, einer Münze. In diesem Kapitel siehst du, wie ein Bild als Liste von Zahlen im Sketch steht, wie du ein PNG mit einem Werkzeug in eine solche Liste umwandelst, wie Bilder durchsichtige Stellen bekommen und wie aus mehreren Bildern eine Animation wird.

15.1 Ein Bild ist ein Array von Farben

Ein Bild besteht aus Pixeln, und jedes Pixel hat eine Farbe. Eine Farbe ist auf dem PicoBoy eine 16-Bit-Zahl (Kapitel 6). Ein Bild aus 8 · 8 Pixeln ist also einfach ein Array aus 64 solchen Zahlen. Die Pixel stehen darin **Zeile für Zeile**: zuerst die 8 Pixel der obersten Zeile von links nach rechts, dann die 8 der zweiten Zeile und so weiter.

k15_herz.ino

```
const uint16_t S = ST77XX_BLACK;
const uint16_t R = ST77XX_RED;

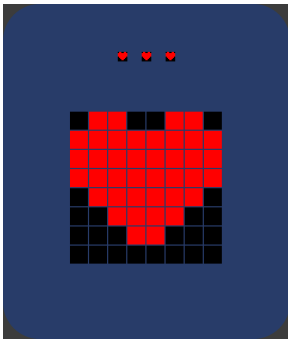
// Zeile fuer Zeile, von links nach rechts
const uint16_t herz[64] = {
    S, R, R, S, S, R, R, S,
    R, R, R, R, R, R, R, R,
    R, R, R, R, R, R, R, R,
    R, R, R, R, R, R, R, R,
    S, R, R, R, R, R, R, S,
    S, S, R, R, R, R, S, S,
    S, S, S, R, R, S, S, S,
    S, S, S, S, S, S, S, S,
};

void setup() {
    tft.init(240, 280);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);

    bild.fillScreen(tft.color565(40, 60, 110));
    // in Originalgroesse: x, y, Array, Breite, Hoehe
    bild.drawRGBBitmap(96, 40, herz, 8, 8);
    bild.drawRGBBitmap(116, 40, herz, 8, 8);
    bild.drawRGBBitmap(136, 40, herz, 8, 8);

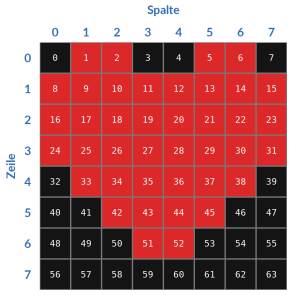
    // vergroessert: jedes Pixel als Quadrat
    for (int zeile = 0; zeile < 8; zeile++) {
        for (int spalte = 0; spalte < 8; spalte++) {
            uint16_t farbe = herz[zeile * 8 + spalte];
```

```
42              bild.fillRect(56 + spalte * 16, 90 + zeile * 16, 15, 15, farbe);
43          }
44      }
45      tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
46  }
```



- **Zeile 12 und 13:** Zwei Konstanten mit kurzen Namen, damit das Array gut lesbar bleibt. `S` steht für Schwarz, `R` für Rot.
- **Zeile 16 bis 25:** das Bild. Weil jede Zeile des Bildes eine eigene Zeile im Sketch hat, kannst du das Herz im Quelltext erkennen. Dem Compiler ist das egal; für ihn ist es eine Liste von 64 Zahlen.
- **Zeile 34 bis 36:** `drawRGBBitmap(x, y, bild, breite, hoehe)` zeichnet das Bild mit seiner linken oberen Ecke an (x, y). Diesen Befehl kennst du schon aus Kapitel 14. Dort hat er den ganzen Canvas zum Display geschickt, hier zeichnet er ein kleines Bild **in** den Canvas. Breite und Höhe musst du angeben, denn ein Array weiß nicht, wie viele Pixel eine Zeile hat.
- **Zeile 39 bis 44** zeichnen das Herz vergrößert: jedes Pixel als Quadrat von 15 · 15 Pixeln. Die Farbe des Pixels in Zeile `zeile` und Spalte `spalte` steht im Array an der Stelle `zeile * 8 + spalte`.

Die Abbildung zeigt die Nummer jedes Pixels im Array. Das Pixel in Zeile 2, Spalte 5 hat die Nummer 2 · 8 + 5 = 21: Vor ihm liegen zwei ganze Zeilen zu je 8 Pixeln und in seiner Zeile noch 5 Pixel.



Sieh dir das Herz auf dem Display genau an: Die schwarzen Ecken sind als schwarze Pixel zu sehen. `drawRGBBitmap` zeichnet **jedes** Pixel des Arrays, auch die schwarzen. Wie man Pixel durchsichtig macht, zeigt Abschnitt 15.4.

Kleine Bilder kann man so von Hand schreiben. Für eine Rakete aus 16 · 24 = 384 Pixeln ist das schon mühsam, für ein Foto unmöglich. Dafür gibt es ein Werkzeug.

15.2 Vom PNG zum Array: das Werkzeug `bild2c`

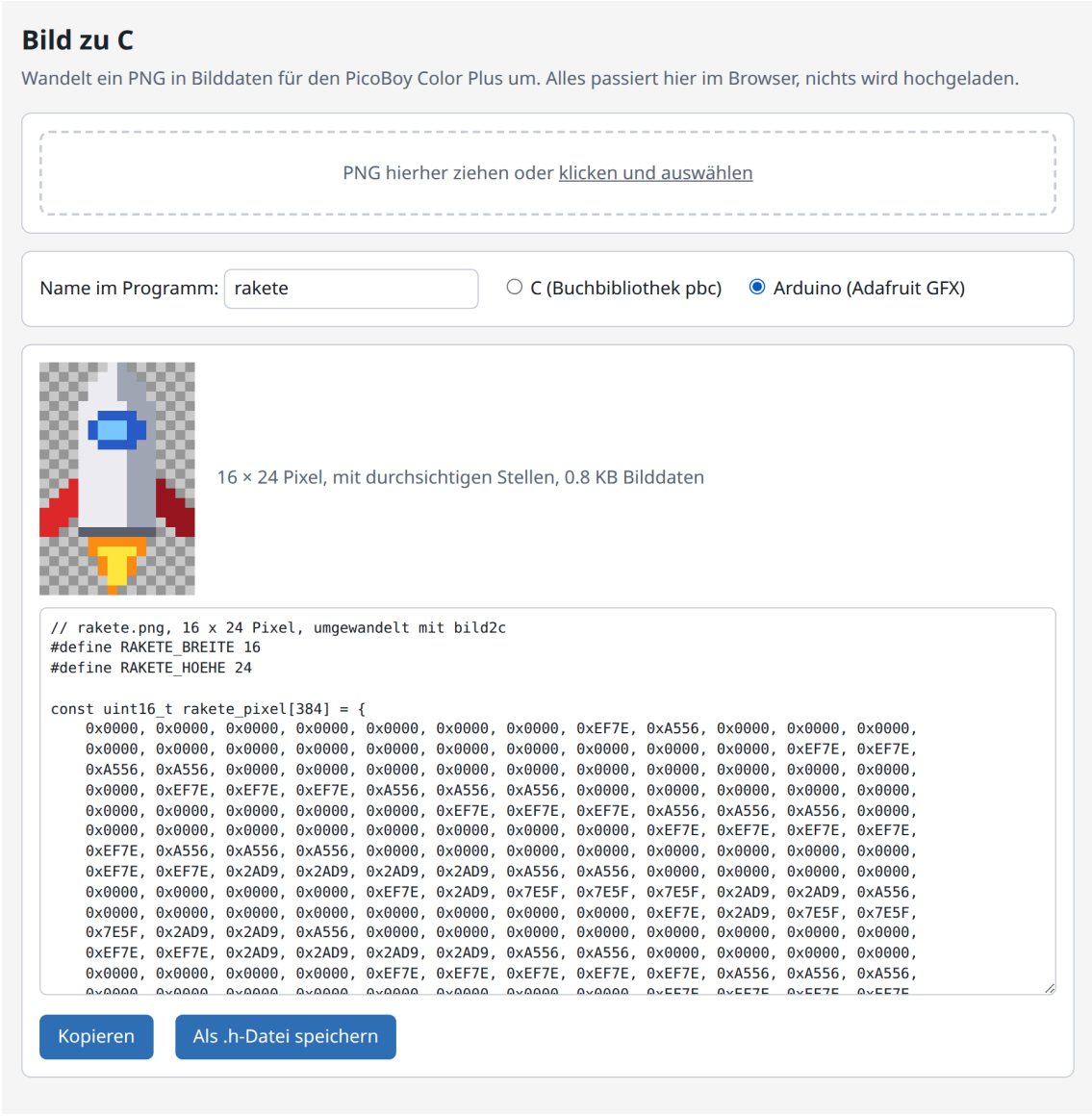
Zeichne dein Bild in einem Malprogramm, das PNG-Dateien speichern und durchsichtige Stellen malen kann, zum Beispiel GIMP oder ein Pixel-Malprogramm wie Piskel im Browser. Für Figuren in Spielen sind kleine Bilder richtig: 8 bis 32 Pixel breit und hoch. Die Bilder in diesem Kapitel sind so entstanden:



Das Schachbrettmuster steht für **durchsichtige** Pixel. Sie gehören nicht zur Figur.

Zum Umwandeln gibt es zum Buch das Werkzeug **`bild2c.html`**. Es liegt bei den Sketches und läuft in jedem Browser, auch ohne Internet. Dein Bild verlässt dabei deinen Computer nicht.

1. Öffne `bild2c.html` mit einem Doppelklick im Browser.
2. Ziehe dein PNG in das gestrichelte Feld oder klicke hinein und wähle die Datei aus.
3. Prüfe den **Namen im Programm**. Er wird aus dem Dateinamen gebildet und darf nur kleine Buchstaben, Ziffern und `_` enthalten.
4. Wähle **Arduino (Adafruit GFX)**.
5. Klicke auf **Als .h-Datei speichern**. Du bekommst eine Datei mit dem Namen des Bildes und der Endung `.h`, zum Beispiel `rakete.h`.



So beginnt die Datei `rakete.h`:

rakete.h

```
1 // rakete.png, 16 x 24 Pixel, umgewandelt mit bild2c
2 #define RAKETE_BREITE 16
3 #define RAKETE_HOEHE 24
4
5 const uint16_t rakete_pixel[384] = {
6     0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0xEF7E, 0xA556,
        0x0000, 0x0000, 0x0000,
7     0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000, 0x0000,
        0x0000, 0xEF7E, 0xEF7E,
```

- Zeile 1 ist ein Kommentar: woher das Bild stammt und wie groß es ist.
- Zeile 2 und 3 legen Breite und Höhe als Konstanten fest: `RAKETE_BREITE` und `RAKETE_HOEHE`. So musst du die Zahlen nicht selbst abtippen.
- Ab Zeile 5 kommt das Array `rakete_pixel` mit $16 \cdot 24 = 384$ Farben, zwölf pro Zeile. Die Farben sind als **Hexadezimalzahlen** geschrieben: `0xEF7E` ist das helle Grau der Rakete, `0x0000` Schwarz. Weiter unten folgt noch ein zweites Array, `rakete_maske`. Was es bedeutet, zeigt Abschnitt 15.4.

> Die Arrays sind mit `const` als unveränderlich gekennzeichnet. Solche Daten bleiben im großen **Flash-Speicher** des PicoBoy (16 MB), in dem auch dein Sketch liegt, und belegen keinen Arbeitsspeicher. Mehr dazu in Kapitel 19.

15.3 Eine zweite Datei im Sketch

Die Datei `rakete.h` gehört **in den Ordner deines Sketches**, neben die `.ino`-Datei:

```
k15_rakete/
    k15_rakete.ino
    rakete.h
```

Die Arduino IDE zeigt jede Datei im Sketch-Ordner als eigenen **Tab** (eine Registerkarte) oben über dem Text an: `k15_rakete.ino` und daneben `rakete.h`. Ein Klick auf den Tab zeigt den Inhalt. So kommt die Datei in den Sketch:

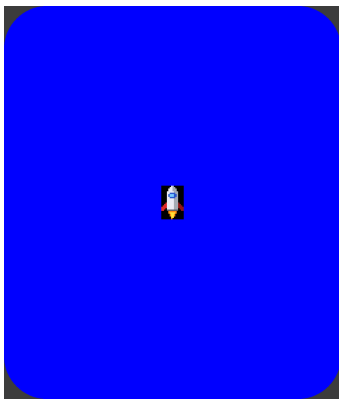
- **Über das Menü: Sketch → Datei hinzufügen…** und die gespeicherte `rakete.h` auswählen. Die IDE kopiert sie in den Sketch-Ordner.
- **Von Hand: Sketch → Sketch-Ordner anzeigen** öffnet den Ordner im Dateimanager. Kopiere `rakete.h` hinein, schließe den Sketch und öffne ihn neu.
- **Als neuer Tab:** Über den Knopf mit den drei Punkten rechts neben den Tabs legst du mit **Neuer Tab** eine leere Datei an, nennst sie `rakete.h` und fügst den Text aus `bild2c` mit **Kopieren** und Einfügen ein.

Im Sketch holst du die Datei mit `#include` herein:

k15_rakete.ino

```
1  // Ein Bild aus einer zweiten Datei: rakete.h
2  #include <Adafruit_ST7789.h>
3  #include "rakete.h"
4
5  #define TFT_CS 10
6  #define TFT_DC 8
7  #define TFT_RST 9
8  #define TFT_LICHT 26
9
10 Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
11 GFXcanvas16 bild(240, 280);
12
13 void setup() {
14     tft.init(240, 280);
15     pinMode(TFT_LICHT, OUTPUT);
16     digitalWrite(TFT_LICHT, HIGH);
17
18     bild.fillScreen(ST77XX_BLUE);
19     bild.drawRGBBitmap(112, 128, rakete_pixel, RAKETE_BREITE, RAKETE_HOEHE);
20     tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
21 }
22
23 void loop() {
24 }
```

Zeile 3: `#include "rakete.h"` setzt den Inhalt von `rakete.h` an dieser Stelle in den Sketch ein, so als stünde er dort. Beachte die Anführungszeichen: `"…"` heißt „im Sketch-Ordner suchen“, `<…>` wie in Zeile 2 heißt „bei den installierten Bibliotheken suchen“.



Zeile 19 zeichnet die Rakete. Die Konstanten `RAKETE_BREITE` und `RAKETE_HOEHE` stammen aus `rakete.h`.

Und das Ergebnis? Die Rakete steht in einem schwarzen Kasten. Die durchsichtigen Pixel des PNG sind im Array einfach schwarz (`0x0000`), und `drawRGBBitmap` zeichnet sie mit.

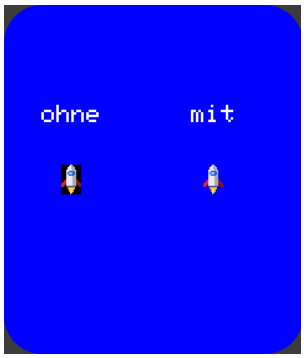
15.4 Durchsichtig: die Maske

Damit der Hintergrund durchscheint, braucht `drawRGBBitmap` eine zweite Angabe: Welche Pixel sollen gezeichnet werden und welche nicht? Das steht in der **Maske**. Hat das PNG durchsichtige Stellen, erzeugt `bild2c` sie automatisch als zweites Array `rakete_maske`. Du gibst sie nach den Pixeln mit an:

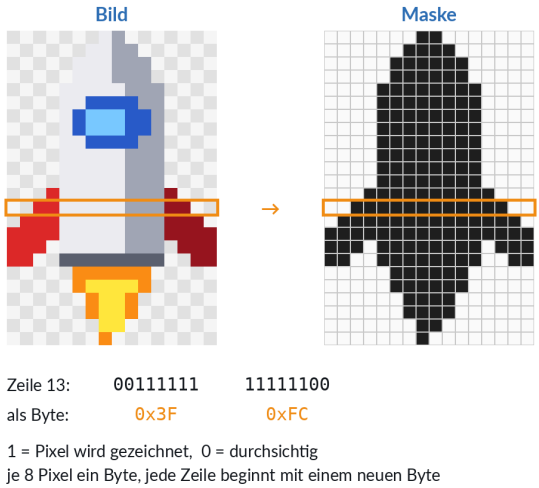
k15_maske.ino

```
26     bild.drawRGBBitmap(46, 128, rakete_pixel, RAKETE_BREITE, RAKETE_HOEHE);
27     bild.drawRGBBitmap(160, 128, rakete_pixel, rakete_maske, RAKETE_BREITE,
         RAKETE_HOEHE);
```

Links die Rakete ohne Maske, rechts mit: Nur die Pixel der Rakete werden gezeichnet, der blaue Hintergrund bleibt sichtbar.



Wie ist die Maske aufgebaut? Für jedes Pixel genügt eine Ja/Nein-Angabe: 1 heißt „zeichnen“, 0 heißt „durchsichtig“. Ein solcher Wert braucht nur ein einziges **Bit**. Damit die Maske klein bleibt, packt `bild2c` jeweils 8 Pixel in ein **Byte**, das linke Pixel ins höchste Bit. Jede Zeile beginnt mit einem neuen Byte. Eine Zeile der Rakete mit 16 Pixeln braucht also 2 Byte, die ganze Maske $24 \cdot 2 = 48$ Byte. So erwartet es Adafruit GFX.



So sieht das Ende von `rakete.h` aus. Jede Zeile der Datei enthält 12 Byte, also 6 Zeilen der Rakete. Das Byte-Paar `0x3F, 0xFC` aus der Abbildung ist das zweite Paar in Zeile 43 der Datei: Zeile 13 der Rakete (gezählt wird ab 0).

rakete.h

```
40 const uint8_t rakete_maske[48] = {
41     0x01, 0x80, 0x03, 0xC0, 0x07, 0xE0, 0x07, 0xE0, 0x0F, 0xF0, 0x0F, 0xF0,
42     0x0F, 0xF0, 0x0F, 0xF0, 0x0F, 0xF0, 0x0F, 0xF0, 0x0F, 0xF0, 0x0F, 0xF0,
43     0x1F, 0xF8, 0x3F, 0xFC, 0x7F, 0xFE, 0xFF, 0xFF, 0xEF, 0xF7, 0xCF, 0xF3,
44     0x07, 0xE0, 0x07, 0xE0, 0x03, 0xC0, 0x03, 0xC0, 0x01, 0x80, 0x01, 0x00,
45 };
```

- Ohne Maske: `bild.drawRGBBitmap(x, y, name_pixel, BREITE, HOEHE);` zeichnet alle Pixel, auch die durchsichtigen als Schwarz.
- Mit Maske: `bild.drawRGBBitmap(x, y, name_pixel, name_maske, BREITE, HOEHE);` zeichnet nur die Pixel der Figur.

Hat ein PNG keine durchsichtigen Stellen, erzeugt `bild2c` auch keine Maske.

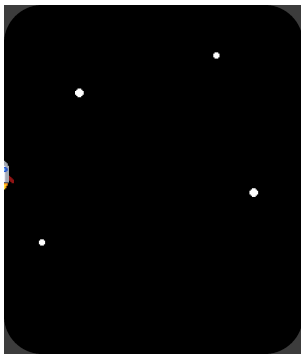
15.5 Sprites bewegen

Eine Figur, die sich über den Bildschirm bewegt, heißt **Sprite** (englisch für Kobold oder Geist, sprich „spreit“). Ein Sprite ist nichts anderes als ein Bild, das du in jedem Bild der Animation an einer anderen Stelle zeichnest. Mit dem Canvas aus Kapitel 14 ist das einfach:

k15_sprite.ino

```
32 void loop() {
33     if (digitalRead(TASTE_LINKS) == LOW) {
34         x = x - TEMPO;
35     }
36     if (digitalRead(TASTE_RECHTS) == LOW) {
37         x = x + TEMPO;
38     }
39     if (digitalRead(TASTE_HOCH) == LOW) {
40         y = y - TEMPO;
41     }
42     if (digitalRead(TASTE_RUNTER) == LOW) {
43         y = y + TEMPO;
44     }
45     // die Rakete darf halb aus dem Bild ragen, aber nicht ganz verschwinden
46     x = constrain(x, -RAKETE_BREITE / 2, 239 - RAKETE_BREITE / 2);
47     y = constrain(y, 0, 279 - RAKETE_HOEHE);
48
49     bild.fillScreen(ST77XX_BLACK);
50     bild.fillCircle(60, 70, 3, ST77XX_WHITE);
51     bild.fillCircle(170, 40, 2, ST77XX_WHITE);
52     bild.fillCircle(200, 150, 3, ST77XX_WHITE);
53     bild.fillCircle(30, 190, 2, ST77XX_WHITE);
54     bild.drawRGBBitmap(x, y, rakete_pixel, rakete_maske, RAKETE_BREITE,
         RAKETE_HOEHE);
55     tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
56 }
```

- `x` und `y` sind die **linke obere Ecke** des Sprites, nicht seine Mitte. Die Mitte der Rakete liegt bei `x + RAKETE_BREITE / 2`.
- Zeile 46 und 47 halten die Rakete im Bild. Nach links darf sie bis zur Hälfte hinausragen: Ihr `x` wird dann negativ. Das ist erlaubt. `drawRGBBitmap` zeichnet nur die Pixel, die im Canvas liegen, und lässt die anderen einfach weg. Das Bild rechts zeigt die Rakete am linken Rand.
- Zeile 49 bis 54: erst der Hintergrund, dann das Sprite. Was später gezeichnet wird, liegt **oben**.

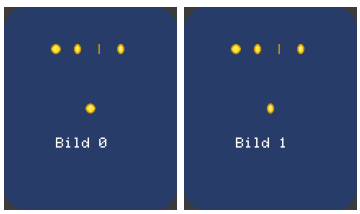


15.6 Animation aus mehreren Bildern

Eine Münze, die sich dreht, ist eigentlich eine Folge von Bildern: von vorn, schräg, von der Seite, wieder schräg. Zeigt man sie schnell nacheinander, sieht das Auge eine Drehung. So funktioniert jeder Zeichentrickfilm. Die vier Bilder `muenze_0.png` bis `muenze_3.png` sind mit `bild2c` umgewandelt und stehen zusammen in einer Datei `muenze.h`.

k15_muenze.ino

```
14 // zeichnet Bild Nummer 0 bis 3 der Muenze
15 void zeichne_muenze(int x, int y, int nummer) {
16     switch (nummer) {
17         case 0:
18             bild.drawRGBBitmap(x, y, muenze_0_pixel, muenze_0_maske, 16, 16);
19             break;
20         case 1:
21             bild.drawRGBBitmap(x, y, muenze_1_pixel, muenze_1_maske, 16, 16);
22             break;
23         case 2:
24             bild.drawRGBBitmap(x, y, muenze_2_pixel, muenze_2_maske, 16, 16);
25             break;
26         default:
27             bild.drawRGBBitmap(x, y, muenze_3_pixel, muenze_3_maske, 16, 16);
28             break;
29     }
30 }
31
32 void setup() {
33     tft.init(240, 280);
34     tft.setSPISpeed(62500000);
35     pinMode(TFT_LICHT, OUTPUT);
36     digitalWrite(TFT_LICHT, HIGH);
37 }
38
39 void loop() {
40     int nummer = (millis() / BILDDAUER) % 4;    // 0, 1, 2, 3, 0, 1, ...
41
42     bild.fillScreen(tft.color565(40, 60, 110));
43     // oben alle vier Bilder nebeneinander
44     for (int i = 0; i < 4; i++) {
45         zeichne_muenze(64 + i * 30, 50, i);
46     }
47     // in der Mitte die Animation
48     zeichne_muenze(112, 132, nummer);
49     bild.setTextColor(ST77XX_WHITE);
50     bild.setTextSize(2);
51     bild.setCursor(72, 180);
52     bild.printf("Bild %d", nummer);
53     tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
54 }
```



- **Zeile 15 bis 30: `zeichne_muenze`.** Die Funktion bekommt die Nummer des Bildes und wählt mit `switch` das passende Array. Alle vier Bilder sind 16 · 16 Pixel groß.
- **Zeile 40: Welches Bild ist dran?** `millis() / BILDDAUER` zählt alle 120 ms um eins weiter. Der Rest der Division durch 4 (`% 4`) läuft dann immer im Kreis: 0, 1, 2, 3, 0, 1 … So wechselt das Bild alle 120 ms, egal wie schnell `loop()` läuft.
- Die zwei Displaybilder rechts sind 130 ms nacheinander entstanden: einmal Bild 0, einmal Bild 1.

Mit `BILDDAUER` stellst du das Tempo der Animation ein, unabhängig davon, wie schnell sich die Figur bewegt. Für eine gehende Figur oder flatternde Flügel genügen oft schon zwei Bilder im Wechsel.

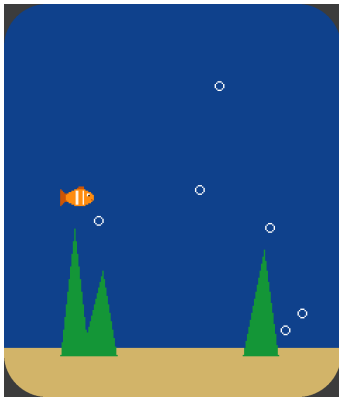
Ausprobieren: Das Aquarium

Ein Fisch schwimmt hin und her und schaut dabei immer in die Richtung, in die er schwimmt. Blasen steigen vom Boden auf. Für die beiden Blickrichtungen gibt es zwei Bilder, `fisch_rechts.png` und `fisch_links.png`, gemeinsam in `fisch.h`.

k15_aquarium.ino

```
12 struct Blase {
13     float x, y;
14     float tempo;
15 };
16
17 Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
18 GFXcanvas16 bild(240, 280);
19
20 float fisch_x = 40;
21 float fisch_dx = 1.5;              // positiv: nach rechts
22 Blase blasen[BLASEN];
23 unsigned long letztes_bild = 0;
24
25 void neue_blase(int i) {
26     blasen[i].x = random(20, 220);
27     blasen[i].y = random(240, 300);
28     blasen[i].tempo = random(5, 15) / 10.0;
29 }
30
31 void zeichne_aquarium() {
32     uint16_t wasser = tft.color565(20, 70, 140);
33     uint16_t sand = tft.color565(210, 180, 110);
34     uint16_t pflanze = tft.color565(30, 150, 60);
35     bild.fillScreen(wasser);
36     bild.fillRect(0, 245, 240, 35, sand);
37     bild.fillTriangle(40, 250, 50, 160, 60, 250, pflanze);
38     bild.fillTriangle(55, 250, 70, 190, 80, 250, pflanze);
39     bild.fillTriangle(170, 250, 185, 175, 195, 250, pflanze);
40 }
41
42 void setup() {
43     tft.init(240, 280);
44     tft.setSPISpeed(62500000);
45     pinMode(TFT_LICHT, OUTPUT);
46     digitalWrite(TFT_LICHT, HIGH);
```

```
47      for (int i = 0; i < BLASEN; i++) {
48          neue_blase(i);
49      }
50  }
```



- Die Blasen sind Strukturen in einem Array, wie die Bälle in Kapitel 14. `neue_blase(i)` setzt Blase Nummer `i` an eine zufällige Stelle unter dem Boden; von dort steigt sie auf.
- `zeichne_aquarium()` malt Wasser, Sand und drei Pflanzen. Diese Funktion ist der **Hintergrund**. Sie läuft vor jedem Bild, so wie `fillScreen` in den Sketches davor.

k15_aquarium.ino

```
52  void loop() {
53      if (millis() - letztes_bild >= TAKT) {
54          letztes_bild = millis();
55
56          // der Fisch dreht am Rand um
57          fisch_x = fisch_x + fisch_dx;
58          if (fisch_x < 10 || fisch_x > 230 -
              FISCH_RECHTS_BREITE) {
59              fisch_dx = -fisch_dx;
60          }
61          // auf und ab wie eine Welle
62          int fisch_y = 120 + 15 * sin(millis() / 600.0);
63
64          zeichne_aquarium();
65          for (int i = 0; i < BLASEN; i++) {
66              blasen[i].y = blasen[i].y - blasen[i].tempo;
67              if (blasen[i].y < 20) {
68                  neue_blase(i);
69              }
70              bild.drawCircle(blasen[i].x, blasen[i].y, 3,
                  ST77XX_WHITE);
71          }
72          if (fisch_dx > 0) {
73              bild.drawRGBBitmap(fisch_x, fisch_y,
                    fisch_rechts_pixel, fisch_rechts_maske,
74                                  FISCH_RECHTS_BREITE,
                                      FISCH_RECHTS_HOEHE);
75          } else {
76              bild.drawRGBBitmap(fisch_x, fisch_y,
                    fisch_links_pixel, fisch_links_maske,
77                                  FISCH_LINKS_BREITE,
                                      FISCH_LINKS_HOEHE);
78          }
79          tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240,
              280);
80      }
81  }
```

- **Zeile 58 und 59:** Am Rand dreht der Fisch um: `fisch_dx` wechselt das Vorzeichen.

- **Zeile 62:** `sin()` ist die Sinusfunktion aus der Mathematik. Sie liefert Werte, die gleichmäßig zwischen −1 und 1 hin- und herschwingen. Mal 15 genommen, schwimmt der Fisch zwischen y = 105 und y = 135 sanft auf und ab. Die Zahl 600.0 bestimmt, wie langsam.
- **Zeile 72 bis 78:** Das Vorzeichen von `fisch_dx` entscheidet, welches der beiden Bilder gezeichnet wird.

Zum Verändern: Setze einen zweiten Fisch ins Aquarium, der schneller und in einer anderen Höhe schwimmt. Lass die Blasen beim Aufsteigen leicht hin und her wackeln.

Übungsaufgaben

Aufgabe 1 ⭐ **Ein Smiley von Hand** Schreibe ein 8 · 8-Bild eines gelben Smileys als Array, so wie das Herz in 15.1. Zeige es fünfmal nebeneinander in Originalgröße und einmal vergrößert.

Aufgabe 2 ⭐ **Bytes zählen** a) Wie viele Byte brauchen `rakete_pixel` (16 · 24) und `rakete_maske`? b) Wie viele Byte brauchen die Pixel und die Maske des Fischs (24 · 16)? c) Eine Zeile einer Maske lautet `0x18, 0x00`. Wie viele Pixel ist das Bild mindestens breit, und welche Pixel der Zeile werden gezeichnet? d) Wie viele Byte bräuchte ein Hintergrundbild in voller Displaygröße?

Aufgabe 3 ⭐⭐ **Münzregen** Fünf Münzen fallen von oben herunter, jede mit eigenem Tempo und an einer zufälligen Stelle. Unten angekommen, starten sie wieder oberhalb des Bildschirms. Alle drehen sich, aber nicht im Gleichschritt. Tipp: Jede Münze bekommt einen eigenen `versatz` von 0 bis 3, der zur Bildnummer addiert wird.

Aufgabe 4 ⭐⭐ **Fisch am Joystick** Steuere den Fisch mit dem Joystick in alle vier Richtungen. Er soll in die Richtung schauen, in die du ihn **zuletzt** nach links oder rechts bewegt hast, auch wenn er gerade still steht oder nur nach oben schwimmt.

Aufgabe 5 ⭐⭐⭐ Deine eigene Figur Male eine eigene Figur mit durchsichtigem Hintergrund in zwei Bildern, die sich ein wenig unterscheiden (zum Beispiel Flügel oben und unten). Wandle beide mit `bild2c` um und speichere sie zusammen in einer `.h`-Datei. Lass die Figur vor einem selbst gezeichneten Hintergrund hin- und herfliegen und dabei zwischen den beiden Bildern wechseln.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Ein Bild ist ein Array von Farben, Zeile für Zeile. Das Pixel in Zeile `z` und Spalte `s` steht an der Stelle `z * breite + s`.
- `bild2c.html` wandelt ein PNG in eine `.h`-Datei um (Einstellung **Arduino**): `name_pixel`, bei durchsichtigen Stellen `name_maske`, dazu `NAME_BREITE` und `NAME_HOEHE`.
- Die `.h`-Datei kommt in den Sketch-Ordner (eigener Tab in der IDE) und wird mit `#include "name.h"` eingebunden.
- `bild.drawRGBBitmap(x, y, name_pixel, name_maske, NAME_BREITE, NAME_HOEHE);` zeichnet nur die Pixel der Figur; ohne Maske werden durchsichtige Pixel schwarz.
- Die Maske hat 1 Bit pro Pixel, 8 Pixel pro Byte, jede Zeile beginnt mit einem neuen Byte.
- Ein Sprite ist ein Bild, das in jedem Bild an einer neuen Stelle gezeichnet wird. Was über den Rand ragt, wird einfach abgeschnitten.
- Animation: mehrere Bilder im Wechsel, Bildnummer zum Beispiel `(millis() / 120) % 4`.

KAPITEL 16

Kollisionen

Worum geht's? Ein Spiel muss merken, wann sich zwei Dinge berühren: Der Ball trifft den Schläger, die Rakete den Meteor, die Spielfigur die Münze. So eine Berührung heißt **Kollision**. Du lernst, sie für Rechtecke, Punkte und Kreise auszurechnen, passende Trefferflächen für Sprites zu wählen und typische Fehler zu vermeiden. Am Ende steht ein kleines Ballspiel.

16.1 Am Rand: festhalten oder abprallen

Die einfachste Kollision kennst du schon: die mit dem Bildschirmrand. Es gibt zwei Arten, darauf zu reagieren.

Festhalten. Eine Spielfigur soll den Bildschirm nicht verlassen. Nach jeder Bewegung wird ihre Position in den erlaubten Bereich zurückgeholt, am einfachsten mit `constrain` (Kapitel 14):

```
x = constrain(x, 20, 220 - BREITE);   // linke Kante 20 bis rechte Kante 220
```

Achte auf die **rechte Kante**: `x` ist die linke obere Ecke der Figur. Ihre rechte Kante liegt bei `x + BREITE`, und die darf höchstens 220 sein. Also darf `x` höchstens `220 - BREITE` sein. Wegen der runden Ecken des Displays hält man außerdem etwa 20 Pixel Abstand zum Rand.

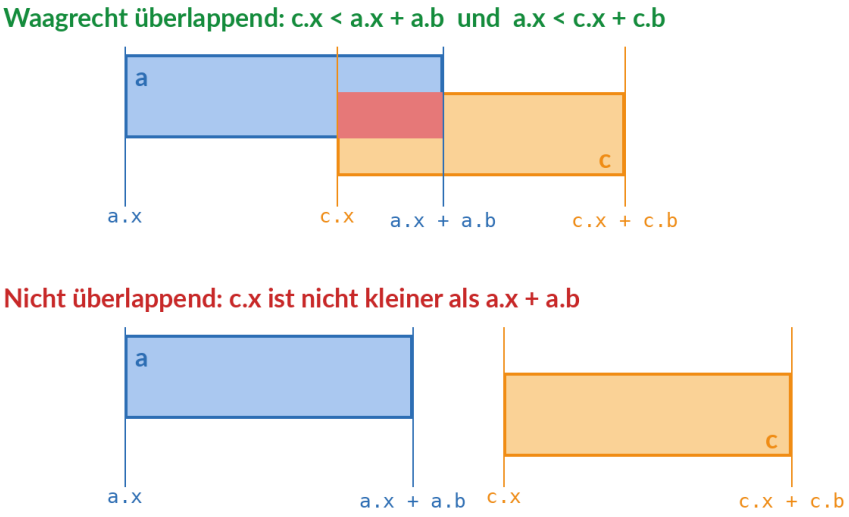
Abprallen. Ein Ball soll am Rand umkehren. Wie in Kapitel 14.6 wird er an den Rand zurückgesetzt, und seine Richtung dreht sich um: `dx = -dx`.

16.2 Zwei Rechtecke

Fast alle Figuren in Spielen werden für Kollisionen wie **Rechtecke** behandelt, auch wenn sie anders aussehen. Ein Rechteck beschreibst du mit seiner linken oberen Ecke `x`, `y` und seiner Breite `b` und Höhe `h`. Dafür eignet sich eine Struktur (Kapitel 13):

```
struct Rechteck {
    int x, y;       // linke obere Ecke
    int b, h;       // Breite und Hoehe
};
```

Wann überlappen sich zwei Rechtecke `a` und `c`? Zuerst nur waagrecht betrachtet: Sie überlappen sich, wenn `c` links **vor** dem rechten Rand von `a` beginnt **und** `a` links vor dem rechten Rand von `c` beginnt.



Senkrecht gilt dasselbe mit `y` und `h`. Überlappen müssen sie sich in **beiden** Richtungen. Daraus wird eine Funktion, die `true` oder `false` liefert:

k16_abstand.ino

```
 7 // ueberlappen sich a und c?
 8 bool trifft(Rechteck a, Rechteck c) {
 9     return a.x < c.x + c.b && c.x < a.x + a.b &&
10            a.y < c.y + c.h && c.y < a.y + a.h;
11 }
12
13 // liegt der Punkt (px, py) im Rechteck r?
14 bool punkt_in(int px, int py, Rechteck r) {
15     return px >= r.x && px < r.x + r.b && py >= r.y && py < r.y + r.h;
16 }
```

Die zweite Funktion `punkt_in` prüft, ob ein einzelner Punkt in einem Rechteck liegt, zum Beispiel eine Fadenkreuz-Spitze in einem Knopf.

Bevor du solche Funktionen in einem Spiel benutzt, prüfe sie mit ein paar Beispielen, deren Ergebnis du kennst. Das erledigt der Rest des Sketches:

k16_abstand.ino

```
25 void zeige(String frage, bool antwort) {
26     Serial.print(frage);
27     if (antwort) {
28         Serial.println(": ja");
29     } else {
30         Serial.println(": nein");
31     }
32 }
33
34 void setup() {
35     Serial.begin(115200);
36     Rechteck ufo = { 0, 0, 28, 20 };
37     Rechteck stern1 = { 24, 12, 16, 16 };
```

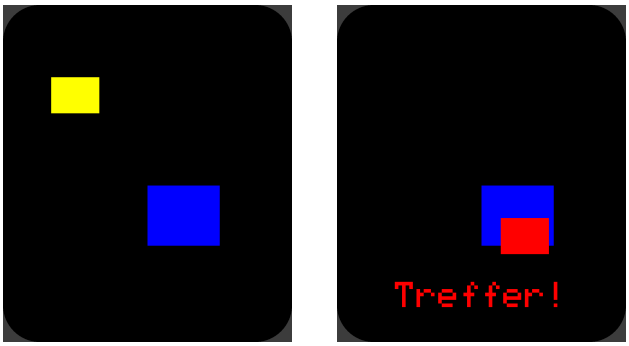
```
38      Rechteck stern2 = { 28, 12, 16, 16 };
39      zeige("ufo trifft stern1", trifft(ufo, stern1));
40      zeige("ufo trifft stern2", trifft(ufo, stern2));
41      zeige("(27, 19) im ufo", punkt_in(27, 19, ufo));
42      zeige("(28, 19) im ufo", punkt_in(28, 19, ufo));
43      zeige("Kreise, Abstand 15, Radien 10 und 5", kreise_beruehren(0, 0, 10, 9,
          12, 5));
44      zeige("Kreise, Abstand 16, Radien 10 und 5", kreise_beruehren(0, 0, 10, 0,
          16, 5));
45  }
```

Serieller Monitor

```
ufo trifft stern1: ja
ufo trifft stern2: nein
(27, 19) im ufo: ja
(28, 19) im ufo: nein
Kreise, Abstand 15, Radien 10 und 5: ja
Kreise, Abstand 16, Radien 10 und 5: nein
```

- Das Ufo reicht von x = 0 bis x = 27; sein rechter Rand liegt also bei 0 + 28 = 28. `stern1` beginnt bei x = 24, also innerhalb: Treffer.
- `stern2` beginnt bei x = 28, genau dort, wo das Ufo aufhört. Die beiden liegen direkt nebeneinander, aber kein Pixel gehört zu beiden. Deshalb steht in `trifft` ein `<` und kein `<=`.
- Genauso liegt der Punkt (27, 19) noch im Ufo, (28, 19) schon nicht mehr.

Und so sieht das in Bewegung aus. Das gelbe Rechteck steuerst du mit dem Joystick. Berührt es die blaue Kiste, wird es rot:



k16_rechtecke.ino

```
22  Rechteck spieler = { 40, 60, 40, 30 };
23  Rechteck kiste = { 120, 150, 60, 50 };
24
25  bool trifft(Rechteck a, Rechteck c) {
26      return a.x < c.x + c.b && c.x < a.x + a.b &&
27             a.y < c.y + c.h && c.y < a.y + a.h;
28  }
```

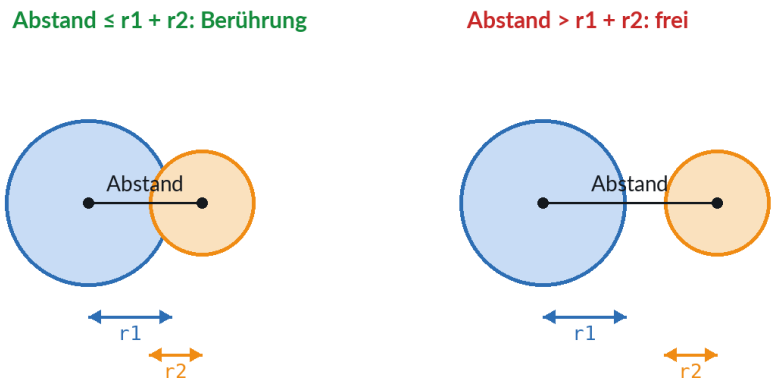
k16_rechtecke.ino

```
55      bild.fillScreen(ST77XX_BLACK);
56      bild.fillRect(kiste.x, kiste.y, kiste.b, kiste.h, ST77XX_BLUE);
57      if (trifft(spieler, kiste)) {
58          bild.fillRect(spieler.x, spieler.y, spieler.b, spieler.h, ST77XX_RED);
59          bild.setTextColor(ST77XX_RED);
60          bild.setTextSize(3);
61          bild.setCursor(48, 230);
62          bild.print("Treffer!");
63      } else {
64          bild.fillRect(spieler.x, spieler.y, spieler.b, spieler.h, ST77XX_YELLOW);
65      }
66      tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
67  }
```

Die Rechtecke `spieler` und `kiste` dienen gleich doppelt: zum Zeichnen mit `fillRect` und zum Prüfen mit `trifft`. So passen Bild und Kollision immer zusammen.

16.3 Zwei Kreise

Für runde Dinge wie Bälle oder Planeten ist ein Rechteck zu ungenau: An den Ecken des Rechtecks wäre schon ein Treffer, obwohl sich die Kreise noch gar nicht berühren. Kreise prüft man anders: Zwei Kreise berühren sich, wenn der **Abstand ihrer Mittelpunkte** höchstens so groß ist wie die **Summe ihrer Radien**.



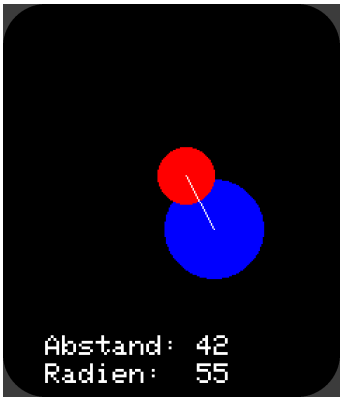
Den Abstand liefert der Satz des Pythagoras: Liegen die Mittelpunkte `dx` Pixel waagrecht und `dy` Pixel senkrecht auseinander, ist der Abstand $\sqrt{(dx^2 + dy^2)}$. Die Wurzel ist für den Prozessor aufwendig, und man braucht sie gar nicht: Wenn der Abstand höchstens `r1 + r2` ist, dann ist auch das Quadrat des Abstands höchstens `(r1 + r2)²`. Also vergleicht man die Quadrate:

k16_abstand.ino

```
18  // beruehren sich zwei Kreise?
19  bool kreise_beruehren(int x1, int y1, int r1, int x2, int y2, int r2) {
20      int dx = x2 - x1;
21      int dy = y2 - y1;
22      return dx * dx + dy * dy <= (r1 + r2) * (r1 + r2);
23  }
```

Im Test oben liegen die Mittelpunkte 9 Pixel waagrecht und 12 Pixel senkrecht auseinander, der Abstand ist also 15 ($9^2 + 12^2 = 81 + 144 = 225 = 15^2$). Bei den Radien 10 und 5 ist das genau eine Berührung.

Der Sketch `k16_kreise` zeigt das mit zwei Kreisen: Den grünen steuerst du, bei Berührung wird er rot. Unten stehen der Abstand und die Summe der Radien. Nur für diese Anzeige rechnet er mit `sqrt` (englisch *square root*, Quadratwurzel) den echten Abstand aus:



k16_kreise.ino

```
55      uint16_t farbe = ST77XX_GREEN;
56      if (kreise_beruehren(sx, sy, sr, zx, zy, zr)) {
57          farbe = ST77XX_RED;
58      }
59      bild.fillScreen(ST77XX_BLACK);
60      bild.fillCircle(zx, zy, zr, ST77XX_BLUE);
61      bild.fillCircle(sx, sy, sr, farbe);
62      bild.drawLine(sx, sy, zx, zy, ST77XX_WHITE);
63      bild.setTextColor(ST77XX_WHITE);
64      bild.setTextSize(2);
65      bild.setCursor(30, 236);
66      bild.printf("Abstand: %d", (int)sqrt((zx - sx) * (zx - sx) + (zy - sy) * (zy
          - sy)));
67      bild.setCursor(30, 256);
68      bild.printf("Radien:  %d", sr + zr);
69      tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
```

16.4 Sprites: Bildrahmen oder Trefferbox?

Ein Sprite ist ein rechteckiges Bild. Nimmst du das ganze Bild als Rechteck für `trifft`, meldet das Spiel manchmal einen Treffer, obwohl sich nur die durchsichtigen Ecken überlappen. Der Spieler sieht dann: Die Rakete ist am Meteor vorbeigeflogen und hat trotzdem ein Leben verloren. Das wirkt ungerecht.

Besser ist eine kleinere **Trefferbox** (englisch *hitbox*): ein Rechteck, das nur den Kern der Figur abdeckt. Ein bisschen zu klein ist dabei besser als zu groß. Streift ein Meteor die Rakete nur knapp, freut sich der Spieler, dass er noch einmal davongekommen ist.

k16_trefferbox.ino

```
39    int rx = 100;                    // Rakete
40    int ry = 110;
41    int mx = 113;                    // Meteor
42    int my = 98;
43    // ganze Bilder
44    Rechteck rakete_rahmen = { rx, ry, RAKETE_BREITE, RAKETE_HOEHE };
45    Rechteck meteor_rahmen = { mx, my, METEOR_BREITE, METEOR_HOEHE };
46    // kleinere Trefferboxen: nur der Rumpf der Rakete, der Kern des Meteors
47    Rechteck rakete_box = { rx + 4, ry + 2, 8, 20 };
48    Rechteck meteor_box = { mx + 2, my + 2, 12, 12 };
49
50    bild.fillScreen(ST77XX_BLACK);
51    bild.drawRGBBitmap(rx, ry, rakete_pixel, rakete_maske, RAKETE_BREITE,
        RAKETE_HOEHE);
52    bild.drawRGBBitmap(mx, my, meteor_pixel, meteor_maske, METEOR_BREITE,
        METEOR_HOEHE);
53    bild.drawRect(rx, ry, RAKETE_BREITE, RAKETE_HOEHE, ST77XX_WHITE);
54    bild.drawRect(mx, my, METEOR_BREITE, METEOR_HOEHE, ST77XX_WHITE);
55    bild.drawRect(rakete_box.x, rakete_box.y, rakete_box.b, rakete_box.h,
        ST77XX_GREEN);
56    bild.drawRect(meteor_box.x, meteor_box.y, meteor_box.b, meteor_box.h,
        ST77XX_GREEN);
57
58    bild.setTextSize(2);
59    bild.setTextColor(ST77XX_WHITE);
60    zeige(180, "Rahmen: ", trifft(rakete_rahmen, meteor_rahmen));
61    bild.setTextColor(ST77XX_GREEN);
62    zeige(205, "Boxen:  ", trifft(rakete_box, meteor_box));
```

- Weiß sind die Bildrahmen: Sie überlappen sich oben rechts an der Rakete, obwohl sich kein sichtbares Pixel berührt.
- Grün sind die Trefferboxen: Bei der Rakete nur der Rumpf ohne Flügel und Flamme, beim Meteor der Kern ohne den Rand. Sie überlappen nicht.



Die Zahlen für die Trefferbox (4 Pixel nach rechts, 8 breit …) liest du aus dem vergrößerten Bild ab, zum Beispiel aus der Abbildung in Kapitel 15.2.

16.5 Nach dem Treffer

Eine Kollision zu erkennen, ist die eine Hälfte. Die andere ist, richtig darauf zu reagieren. Drei Fehler kommen dabei immer wieder vor.

Mehrfach zählen. Ein Ball, der den Schläger trifft, steckt danach oft noch ein, zwei Bilder lang im Schläger. Prüft man nur `trifft`, wird er jedes Mal wieder umgedreht und zittert im Schläger. Abhilfe: nur reagieren, wenn der Ball **auf** den Schläger zufliegt (`dy > 0`). Genauso muss eine

eingesammelte Münze sofort verschwinden oder woanders hin, sonst zählt sie in jedem Bild einen Punkt.

Durchtunneln. Bewegt sich etwas pro Bild weiter, als das Hindernis dick ist, kann es in einem Bild **vor** und im nächsten schon **hinter** dem Hindernis sein. Eine Überlappung gibt es dann nie. Aufgabe 4 zeigt das und eine Abhilfe.

Reihenfolge. Erst alles bewegen, dann alle Kollisionen prüfen, dann zeichnen. Wer zeichnet und danach erst prüft, zeigt ein Bild, in dem der Ball schon im Schläger steckt.

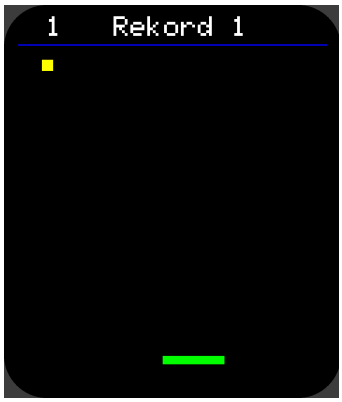
Ausprobieren: Ball und Schläger

Ein Ball prallt von den Wänden und vom Schläger ab, den du mit dem Joystick nach links und rechts steuerst. Jeder Treffer gibt einen Punkt. Fällt der Ball unten hinaus, beginnt alles von vorn; der beste Stand bleibt als Rekord stehen.

k16_pingpong.ino

```
46 void loop() {
47     if (millis() - letztes_bild < TAKT) {
48         return;                     // noch nicht Zeit fuer das naechste Bild
49     }
50     letztes_bild = millis();
51
52     // Schlaeger
53     if (digitalRead(TASTE_LINKS) == LOW) {
54         schlaeger_x = schlaeger_x - 5;
55     }
56     if (digitalRead(TASTE_RECHTS) == LOW) {
57         schlaeger_x = schlaeger_x + 5;
58     }
59     schlaeger_x = constrain(schlaeger_x, 10, 230 - SCHLAEGER_B);
60
61     // Ball und Waende
62     ball_x = ball_x + dx;
63     ball_y = ball_y + dy;
64     if (ball_x < 10 || ball_x > 230 - BALL) {
65         ball_x = constrain(ball_x, 10, 230 - BALL);
66         dx = -dx;
67     }
68     if (ball_y < 30) {
69         ball_y = 30;
70         dy = -dy;
71     }
72
73     // Ball gegen Schlaeger: nur, wenn er nach unten fliegt
74     Rechteck ball = { (int)ball_x, (int)ball_y, BALL, BALL };
75     Rechteck schlaeger = { schlaeger_x, SCHLAEGER_Y, SCHLAEGER_B, 6 };
76     if (dy > 0 && trifft(ball, schlaeger)) {
77         dy = -dy;
```

```
78          // je weiter aussen getroffen, desto schraeger
79          dx = (ball_x + BALL / 2 - (schlaeger_x + SCHLAEGER_B / 2)) / 6.0;
80          punkte = punkte + 1;
81          rekord = max(rekord, punkte);
82      }
83
84      // unten hinaus: von vorn
85      if (ball_y > 280) {
86          ball_x = 116;
87          ball_y = 40;
88          dx = 2.0;
89          dy = 3.0;
90          punkte = 0;
91      }
```



- **Zeile 47 bis 50:** der Takt von 33 ms, diesmal anders geschrieben: Ist noch keine Zeit für ein neues Bild, beendet `return` die Funktion `loop()` sofort (Kapitel 11). Der Kern ruft `loop()` gleich danach wieder auf. Das spart eine Einrückungsebene für den ganzen Rest.
- **Zeile 74 und 75:** Ball und Schläger werden für die Prüfung zu Rechtecken. `(int)ball_x` macht aus der Kommazahl eine ganze Zahl.
- **Zeile 76:** `dy > 0`: nur ein Ball, der nach unten fliegt, prallt ab.
- **Zeile 79:** Wo der Ball den Schläger trifft, bestimmt den Abprallwinkel. In der Mitte fliegt er fast senkrecht nach oben, am Rand schräg zur Seite. So kann der Spieler zielen.
- **Zeile 81:** `max(a, b)` liefert die größere von zwei Zahlen. Der Rekord wächst also mit, sobald die Punkte ihn überholen.

Zum Verändern: Mach den Ball nach je fünf Treffern etwas schneller. Lass einen Ton erklingen, wenn der Ball den Schläger trifft (Kapitel 10).

Übungsaufgaben

Aufgabe 1 ⭐ Treffer vorhersagen Zeichne die fünf Rechtecke aus `k16_aufgabe1` auf Karopapier und sage voraus, was der Sketch für jedes Paar ausgibt. Prüfe danach mit dem seriellen Monitor.

k16_aufgabe1.ino

```
21 void setup() {
22     Serial.begin(115200);
23     Rechteck a = { 10, 10, 20, 20 };
24     Rechteck b = { 25, 25, 10, 10 };
25     Rechteck c = { 30, 10, 10, 10 };
26     Rechteck d = { 0, 0, 100, 100 };
```

```
    Rechteck e = { 15, 40, 5, 5 };
    zeige("a und b", trifft(a, b));
    zeige("a und c", trifft(a, c));
    zeige("a und d", trifft(a, d));
    zeige("b und c", trifft(b, c));
    zeige("a und e", trifft(a, e));
    zeige("d und e", trifft(d, e));
}
```

Aufgabe 2 ⭐ Ein Knopf zum Klicken Auf dem Display ist ein Knopf, ein abgerundetes Rechteck mit der Aufschrift „Klick“. Ein Fadenkreuz lässt sich mit dem Joystick bewegen. Liegt die Mitte des Fadenkreuzes auf dem Knopf, wird er grün. Drückst du dann A, zählt ein Zähler eins hoch. Benutze `punkt_in`.

Aufgabe 3 ⭐⭐ Münzen sammeln Ein Quadrat (die Spielfigur) wird mit dem Joystick gesteuert. Irgendwo liegt eine Münze (`muenze_0` aus Kapitel 15). Berührt die Figur die Münze, gibt es einen Punkt und einen kurzen hohen Ton, und die Münze taucht an einer zufälligen anderen Stelle auf.

Aufgabe 4 ⭐⭐ Durch die Wand Ein Ball, 4 Pixel breit, fliegt mit 12 Pixeln pro Bild auf eine 4 Pixel dicke Wand zu. Lass den Sketch laufen. Warum meldet er nie einen Treffer? Ändere ihn so, dass der Ball jeden Schritt in Teilschritte von einem Pixel zerlegt und nach jedem Teilschritt prüft. Trifft er die Wand, setzt er ihn einen Teilschritt zurück und dreht die Richtung um.

k16_aufgabe4.ino

```
Rechteck wand = { 118, 0, 4, 280 };
Rechteck ball = { 2, 100, 4, 4 };
int dx = 12;

void setup() {
    Serial.begin(115200);
    for (int n = 0; n < 15; n++) {
        ball.x = ball.x + dx;
        Serial.printf("x = %3d", ball.x);
        if (trifft(ball, wand)) {
            Serial.print("  Treffer!");
            dx = -dx;
        }
        Serial.println();
    }
}
```

Aufgabe 5 ⭐⭐⭐ Die Mauer Erweitere „Ball und Schläger“ zu einem Mauerspiel: Oben stehen 4 Reihen zu je 8 Steinen. Trifft der Ball einen Stein, verschwindet dieser, und der Ball prallt ab. Speichere, welche Steine noch da sind, in einem Array `bool steine[32]`; Stein Nummer `i` liegt in Spalte `i % 8` und Reihe `i / 8`, so wie die Pixel in Kapitel 15.1. Zeige an, wie viele Steine noch übrig sind.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Rand: festhalten mit `constrain(x, links, rechts - BREITE)` oder abprallen mit `dx = -dx`.
- Zwei Rechtecke überlappen, wenn `a.x < c.x + c.b && c.x < a.x + a.b && a.y < c.y + c.h && c.y < a.y + a.h`.
- Zwei Kreise berühren sich, wenn `dx * dx + dy * dy <= (r1 + r2) * (r1 + r2)` (ohne Wurzel).
- Für Sprites eine kleinere **Trefferbox** statt des ganzen Bildes nehmen.
- Nach einem Treffer: nur in der passenden Richtung reagieren (`dy > 0`), Eingesammeltes sofort wegnehmen, zu schnelle Dinge in Teilschritten bewegen.
- Reihenfolge in jedem Bild: bewegen, prüfen, zeichnen.

KAPITEL 17

Der Neigungssensor mit der Bibliothek STK8BA58

Worum geht's? Im PicoBoy steckt ein Sensor, der merkt, wie das Gerät gehalten, gekippt oder geschüttelt wird. Du liest ihn mit der Bibliothek STK8BA58 aus, verstehst, was die drei Messwerte bedeuten, glättest zittrige Werte und steuerst eine Kugel durch Kippen. Zum Schluss baust du eine Wasserwaage.

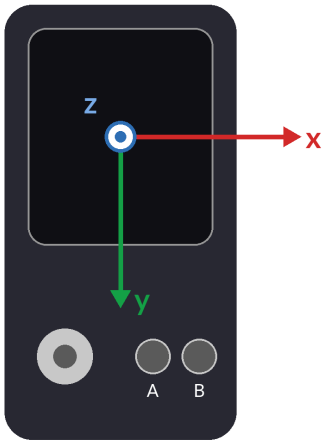
17.1 Was misst der Sensor?

Der Sensor heißt **STK8BA58** und ist ein **Beschleunigungssensor**. Er misst, wie stark das Gerät in eine Richtung beschleunigt wird. Das Besondere: Auch die **Erdanziehung** wirkt auf ihn wie eine Beschleunigung. Selbst wenn der PicoBoy ruhig auf dem Tisch liegt, misst der Sensor deshalb etwas: die Erdanziehung, 9,81 m/s². Diese Stärke heißt **1 g** (g für *gravity*, Schwerkraft). Die Bibliothek liefert alle Werte in g.

Der Sensor misst in drei Richtungen, den **Achsen**:

- **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.

Für jede Achse gilt: Zeigt sie **nach oben**, misst sie die Erdanziehung als **+1 g**. Zeigt sie nach unten, misst sie −1 g. Liegt sie waagrecht, misst sie 0. Daraus ergeben sich diese Werte für ein ruhig gehaltenes Gerät:



z zeigt aus dem Display heraus (auf dich zu)

Lage des PicoBoy	x	y	z
flach auf dem Tisch, Display nach oben	0	0	+1
flach, Display nach unten	0	0	−1
aufrecht wie ein Bild an der Wand	0	−1	0
rechter Rand etwas tiefer als der linke	negativ	0	knapp unter 1
rechter Rand höher als der linke	positiv	0	knapp unter 1

Liegt das Gerät schräg, verteilt sich die Erdanziehung auf mehrere Achsen. Bei 30° Neigung nach rechts zum Beispiel misst x etwa −0,5 g und z etwa 0,87 g. Merke dir: **Die tiefere Seite misst einen negativen Wert.**

Bewegst du den PicoBoy, kommt zur Erdanziehung die Beschleunigung der Bewegung dazu. Beim Schütteln springen die Werte deshalb stark hin und her.

17.2 Die Bibliothek STK8BA58

Für diesen Sensor gab es keine Arduino-Bibliothek; diese hier ist zum Buch geschrieben. Du hast sie in Kapitel 2 als ZIP-Datei installiert. Mit der C-Bibliothek des Pico SDK oder mit MicroPython wird derselbe Sensor über dieselben Register angesprochen, nur mit anderen Befehlen.

k17_werte.ino

```
1  // Der Neigungssensor: drei Messwerte in g im seriellen Monitor
2  #include <STK8BA58.h>
3
4  STK8BA58 sensor;
5
6  void setup() {
7      Serial.begin(115200);
8      if (!sensor.begin(20, 21)) {        // SDA an GP20, SCL an GP21
9          Serial.println("Sensor antwortet nicht");
10     }
11     sensor.setOrientation(STK8BA58::PICOBOY_COLOR_PLUS);
12 }
13
14 void loop() {
15     float x, y, z;
16     sensor.readAcceleration(x, y, z);   // schreibt die Messwerte in x, y und z
17     Serial.printf("x = %5.2f   y = %5.2f   z = %5.2f\n", x, y, z);
18     delay(500);
19 }
```

Serieller Monitor

```
x =  0.00   y =  0.00   z =  1.00
x =  0.00   y =  0.00   z =  1.00
x =  0.00   y =  0.00   z =  1.00
x = -0.50   y =  0.00   z =  0.87
x = -0.50   y =  0.00   z =  0.87
x =  0.00   y = -1.00   z =  0.00
x =  0.00   y = -1.00   z =  0.00
x =  0.00   y =  0.00   z = -1.00
x =  0.00   y =  0.00   z = -1.00
```

Zeile 2 bindet die Bibliothek ein, **Zeile 4** legt das Objekt `sensor` an.

Zeile 8: `sensor.begin(20, 21)` startet die Verbindung zum Sensor. Er hängt an einer Zweidraht-Leitung namens **I2C** (Kapitel 22): Daten über GP20 (SDA), Takt über GP21 (SCL). `begin` prüft, ob sich der Sensor meldet, und stellt ihn ein: Messbereich ±2 g, 125 Messungen pro Sekunde. Es

liefert `true`, wenn alles geklappt hat, sonst `false`. Mit `!` davor (nicht) wird die Meldung nur ausgegeben, wenn etwas schiefging.

Zeile 11: `setOrientation`. Der Sensor ist auf der Platine des PicoBoy gedreht eingebaut. Ohne diese Zeile wären x und y vertauscht und die Vorzeichen falsch: Flach auf dem Tisch zeigte z dann −1 statt 1. `STK8BA58::PICOBOY_COLOR_PLUS` dreht die Werte so, dass sie zu den Achsen aus 17.1 passen.

Zeile 16: `readAcceleration(x, y, z)` liest alle drei Werte auf einmal und schreibt sie in **deine** drei Variablen. Bisher haben Funktionen ein Ergebnis mit `return` zurückgegeben. Hier sind es drei Ergebnisse. In der Beschreibung der Bibliothek steht deshalb `float &x`: Das `&` heißt, die Funktion bekommt nicht eine Kopie des Werts, sondern die Variable selbst und darf hineinschreiben. Wie das genau funktioniert, erklärt Kapitel 19. Für dich heißt das nur: drei `float`-Variablen anlegen und übergeben.

Die Ausgabe oben stammt aus dem Prüflauf auf dem PC; er hat den PicoBoy flach hingelegt, dann um 30° nach rechts gekippt, aufgestellt und umgedreht. Auf dem echten Gerät zittern die Werte in der zweiten oder dritten Nachkommastelle, und selbst flach auf dem Tisch ist z selten genau 1,00.

> **Ausprobieren auf dem Gerät:** Lass `k17_werte` laufen und öffne den **seriellen Plotter** (Kapitel 4.4). Kippe den PicoBoy langsam nach links und rechts, stelle ihn auf, dreh ihn um. Welche Linie bewegt sich? Stimmen die Vorzeichen mit der Tabelle in 17.1 überein?

Warum nicht einfach `sensor.begin()`? Ohne Pins nimmt die Bibliothek die Standardleitungen des Boards für I2C. Beim Board „Generic RP2350“ sind das GP4 und GP5. Am PicoBoy hängt dort aber kein Sensor, sondern **GP4 ist der Joystick nach oben**. Die Folge:

k17_ohne_pins.ino

```
 6 void setup() {
 7     Serial.begin(115200);
 8     if (sensor.begin()) {
 9         Serial.println("Sensor gefunden");
10     } else {
11         Serial.println("Sensor antwortet nicht");
12     }
13 }
```

Serieller Monitor

```
Sensor antwortet nicht
```

Der Sensor antwortet nicht, und GP4 ist danach auf I2C umgeschaltet statt auf den Joystick.

> Auf dem PicoBoy immer `sensor.begin(20, 21);` und danach `sensor.setOrientation(STK8BA58::PICOBOY_COLOR_PLUS);`.

Das kannst du mit dem Sensor außerdem machen:

Befehl	Wirkung
`sensor.readAcceleration(x, y, z)`	alle drei Werte in g; `false` bei einem Fehler
`sensor.readAccelerationMS2(x, y, z)`	dasselbe in m/s²
`sensor.readX()`, `readY()`, `readZ()`	eine Achse in g (liest jedes Mal alle drei neu)
`sensor.readRaw(x, y, z)`	Rohwerte (Stufen, im Bereich ±2048) als `int16_t`
`sensor.setRange(g)`	Messbereich ±2, ±4 oder ±8 g
`sensor.range()`	der eingestellte Messbereich
`sensor.accelerationAvailable()`	`true`, wenn seit dem letzten Lesen neue Werte da sind

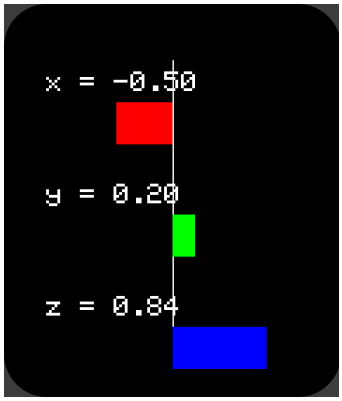
17.3 Die Werte als Balken

Zahlen im seriellen Monitor sind praktisch, aber ein Bild zeigt schneller, was passiert. Dieser Sketch zeichnet für jede Achse einen Balken: nach rechts für positive, nach links für negative Werte, 80 Pixel pro g.

k17_balken.ino

```
15 void balken(int y, String name, float wert, uint16_t farbe) {
16     int laenge = wert * 80;
17     if (laenge >= 0) {
18         bild.fillRect(MITTE, y, laenge, 30, farbe);
19     } else {
20         bild.fillRect(MITTE + laenge, y, -laenge, 30, farbe);
21     }
22     bild.setCursor(30, y - 22);
23     bild.print(name);
24     bild.print(wert, 2);            // zwei Stellen nach dem Komma
25 }
```

- Ein Rechteck soll keine negative Breite haben. Deshalb beginnt der Balken für negative Werte bei `MITTE + laenge` (links von der Mitte) und ist `-laenge` breit (Zeile 20).
- Zeile 24: `print(wert, 2)` gibt eine Kommazahl mit zwei Nachkommastellen aus.



Das Bild zeigt den PicoBoy mit tieferem rechtem Rand und etwas angehobenem unterem Rand: x ist negativ, y positiv.

17.4 Der Messbereich

Der Sensor liefert für jede Achse eine ganze Zahl von −2048 bis 2047, den **Rohwert**. Wie viele g das sind, hängt vom **Messbereich** ab:

Messbereich	Stufen pro g	feinste Stufe	größter Wert
±2 g (Vorgabe)	1024	etwa 0,001 g	knapp 2 g
±4 g	512	etwa 0,002 g	knapp 4 g
±8 g	256	etwa 0,004 g	knapp 8 g

Zum Kippen und Steuern ist ±2 g ideal: Mehr als 1 g misst ein ruhiges Gerät nie, und die Stufen sind am feinsten. Nur wer kräftige Stöße messen will, braucht einen größeren Bereich. Ein Stoß von 3 g ist im Bereich ±2 g abgeschnitten:

k17_messbereich.ino

```
 6 void messen(int bereich) {
 7     sensor.setRange(bereich);
 8     float x, y, z;
 9     int16_t rx, ry, rz;
10     sensor.readAcceleration(x, y, z);
11     sensor.readRaw(rx, ry, rz);
12     Serial.printf("+-%d g:  x = %6.3f g   roh = %5d\n", sensor.range(), x, rx);
13 }
```

Serieller Monitor

```
bei 25 ms:
+-2 g:  x =  0.000 g   roh =     0
+-4 g:  x =  0.000 g   roh =     0
+-8 g:  x =  0.000 g   roh =     0
bei 1086 ms:
+-2 g:  x =  0.000 g   roh =     0
+-4 g:  x =  0.000 g   roh =     0
+-8 g:  x =  0.000 g   roh =     0
bei 2148 ms:
+-2 g:  x =  1.999 g   roh =  2047
+-4 g:  x =  3.000 g   roh =  1536
+-8 g:  x =  0.500 g   roh =   128
```

- Bei 25 ms und 1086 ms liegt das Gerät flach, x ist 0.
- Bei 2148 ms wirkt in x-Richtung ein Stoß von 3 g. Im Bereich ±2 g bleibt der Rohwert beim größten möglichen Wert stehen: 2047 Stufen, also 1,999 g. Erst ±4 g und ±8 g zeigen die 3 g richtig.

> Warum 1,999 g und nicht 2,000 g? Der Rohwert reicht von −2048 bis 2047, nach oben also eine Stufe weniger weit als nach unten. Ein Stoß von 3 g in die Gegenrichtung bliebe bei −2048 stehen, genau −2,000 g.

17.5 Werte glätten

Die Werte eines echten Sensors zittern. Selbst ein ruhig liegendes Gerät misst nie zweimal genau dasselbe. Eine Figur, die direkt dem Messwert folgt, zittert mit. Abhilfe schafft **Glätten**: Man übernimmt nicht den ganzen neuen Wert, sondern nur einen Teil des Unterschieds:

```
glatt = glatt + (roh - glatt) * 0.2;
```

`roh - glatt` ist der Unterschied zwischen neuem Messwert und bisherigem geglättetem Wert. Davon kommen 20 % dazu. Ein einzelner Ausreißer verschiebt `glatt` deshalb nur wenig. Bleibt der neue Wert aber bestehen, wandert `glatt` Schritt für Schritt dorthin.

k17_glaetten.ino

```
15 void loop() {
16     float x, y, z;
17     sensor.readAcceleration(x, y, z);
18     glatt = glatt + (x - glatt) * ANTEIL;
19     Serial.printf("%5lu ms   roh = %5.2f   glatt = %5.2f\n", millis(), x, glatt);
20     delay(100);
21 }
```

Serieller Monitor

```
   25 ms   roh =  0.00   glatt =  0.00
  125 ms   roh =  0.00   glatt =  0.00
  226 ms   roh =  0.00   glatt =  0.00
  326 ms   roh =  0.00   glatt =  0.00
  426 ms   roh =  0.00   glatt =  0.00
  526 ms   roh =  0.00   glatt =  0.00
  626 ms   roh =  0.00   glatt =  0.00
  727 ms   roh =  0.00   glatt =  0.00
  827 ms   roh =  0.00   glatt =  0.00
  927 ms   roh =  0.00   glatt =  0.00
 1027 ms   roh = -0.50   glatt = -0.10
 1128 ms   roh = -0.50   glatt = -0.18
 1228 ms   roh = -0.50   glatt = -0.24
 1328 ms   roh = -0.50   glatt = -0.30
 1428 ms   roh = -0.50   glatt = -0.34
 1529 ms   roh = -0.50   glatt = -0.37
 1629 ms   roh = -0.50   glatt = -0.40
 1729 ms   roh = -0.50   glatt = -0.42
 1829 ms   roh = -0.50   glatt = -0.43
 1929 ms   roh = -0.50   glatt = -0.45
 2030 ms   roh = -0.50   glatt = -0.46
```

Im Prüflauf wird der PicoBoy bei 1000 ms plötzlich nach rechts gekippt. Der Rohwert springt sofort auf −0,50, der geglättete braucht etwa eine Sekunde, bis er fast dort ist.

Wie stark glätten? Ein kleiner Anteil (zum Beispiel 0,1) glättet stark, aber der Wert folgt einer echten Bewegung nur träge. Ein großer Anteil (0,5) reagiert schnell, zittert aber mehr. Probiere am Gerät aus, was zu deinem Sketch passt.

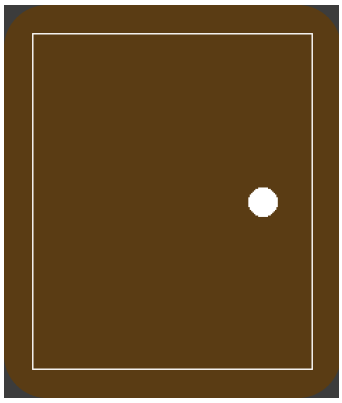
17.6 Steuern durch Kippen

Jetzt steuert der Sensor eine Kugel. Sie rollt wie auf einem Tablett zur tieferen Seite, und zwar umso schneller, je schräger das Gerät ist:

k17_kugel.ino

```
30 void loop() {
31     if (millis() - letztes_bild >= TAKT) {
32         letztes_bild = millis();
33         float x, y, z;
34         sensor.readAcceleration(x, y, z);
35 
36         // tiefere Seite = negativer Messwert, also abziehen
37         kx = kx - x * TEMPO;
38         ky = ky - y * TEMPO;
39         kx = constrain(kx, 20 + RADIUS, 220 - RADIUS);
40         ky = constrain(ky, 20 + RADIUS, 260 - RADIUS);
41 
42         bild.fillScreen(tft.color565(90, 60, 30));
43         bild.drawRect(20, 20, 200, 240, ST77XX_WHITE);
44         bild.fillCircle(kx, ky, RADIUS, ST77XX_WHITE);
45         tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
46     }
47 }
```

- **Zeile 37 und 38:** Ist der rechte Rand tiefer, ist x negativ. Die Kugel soll dann aber nach **rechts** rollen, x auf dem Display soll also größer werden. Deshalb wird der Messwert **abgezogen**: `kx - (negativ)` wird größer. Für y gilt dasselbe: Ist der untere Rand tiefer, ist y negativ, und die Kugel rollt nach unten.
- `TEMPO` legt fest, wie viele Pixel die Kugel pro Bild bei 1 g rollt. Bei 0,3 g Neigung sind das 2,4 Pixel pro Bild.
- **Zeile 39 und 40** halten die Kugel im Rahmen.



Das Bild zeigt die Kugel, nachdem der PicoBoy eine Sekunde lang nach rechts gekippt war.

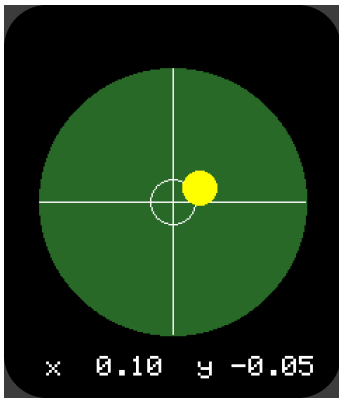
> Reagiert dein Sketch am Gerät „falsch herum“, prüfe zuerst mit `k17_werte`, welche Vorzeichen dein Sensor zeigt, und dreh dann im Sketch das Vorzeichen um (`+` statt `-`).

Ausprobieren: Die Wasserwaage

Eine Wasserwaage hat eine Luftblase in einem Röhrchen mit Flüssigkeit. Die Blase steigt immer zur **höchsten** Stelle. Der PicoBoy wird zur runden Wasserwaage: Liegt er genau waagrecht, steht die Blase in der Mitte, wird grün, und die grüne LED leuchtet.

k17_wasserwaage.ino

```
30 void loop() {
31     float x, y, z;
32     sensor.readAcceleration(x, y, z);
33     gx = gx + (x - gx) * ANTEIL;
34     gy = gy + (y - gy) * ANTEIL;
35
36     // 0,1 g Neigung schiebt die Blase um 20 Pixel, hoechstens bis an den Rand
37     int bx = 120 + constrain(gx * 200, -80, 80);
38     int by = 140 + constrain(gy * 200, -80, 80);
39     bool waagrecht = abs(gx) < WAAGRECHT && abs(gy) < WAAGRECHT;
40
41     bild.fillScreen(ST77XX_BLACK);
42     bild.fillCircle(120, 140, 95, tft.color565(40, 110, 40));
43     bild.drawFastHLine(25, 140, 190, ST77XX_WHITE);
44     bild.drawFastVLine(120, 45, 190, ST77XX_WHITE);
45     bild.drawCircle(120, 140, 16, ST77XX_WHITE);
46     if (waagrecht) {
47         bild.fillCircle(bx, by, 12, ST77XX_GREEN);
48     } else {
49         bild.fillCircle(bx, by, 12, ST77XX_YELLOW);
50     }
51     bild.setTextColor(ST77XX_WHITE);
52     bild.setTextSize(2);
53     bild.setCursor(30, 250);
54     bild.printf("x %5.2f  y %5.2f", gx, gy);
55     tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
56     digitalWrite(LED_GRUEN, waagrecht);
57     delay(20);
58 }
```



- **Zeile 33 und 34:** Die Messwerte werden geglättet (17.5). So zittert die Blase nicht.
- **Zeile 37 und 38:** Hebst du den rechten Rand an, wird x positiv, und die Blase wandert nach rechts, zur hohen Seite. Anders als die Kugel folgt die Blase dem Messwert deshalb mit **gleichem** Vorzeichen. 0,1 g Neigung verschiebt sie um 20 Pixel; `constrain` hält sie im großen Kreis.
- **Zeile 39:** Weniger als 0,03 g auf beiden Achsen gilt als waagrecht. `abs()` liefert den Betrag einer Zahl, also die Zahl ohne Minuszeichen.
- **Zeile 56:** `digitalWrite` bekommt hier direkt den Wahrheitswert: `true` schaltet die LED ein, `false` aus.

Das Bild entstand mit leicht angehobenem rechtem Rand (x = 0,1 g) und leicht angehobenem oberem Rand (y = −0,05 g): Die Blase steht rechts oben, gelb. Mit `k17_wasserwaage` kannst du am Gerät prüfen, ob dein Tisch gerade ist.

Übungsaufgaben

Aufgabe 1 ⭐ **Kipp-Ampel** Die drei LEDs zeigen, wie der PicoBoy gehalten wird: Ist der linke Rand tiefer, leuchtet Rot, ist der rechte Rand tiefer, Grün, und liegt er ungefähr waagrecht, Gelb. Als „gekippt“ gilt eine Neigung ab 0,2 g.

Aufgabe 2 ⭐ **Rohwerte vorhersagen** Der PicoBoy ist so gekippt, dass x genau 0,5 g misst. `k17_aufgabe2` gibt den Rohwert von x in den drei Messbereichen aus. Welche drei Zahlen erwartest du? Prüfe am Gerät (dort ungefähr) oder mit dem Ergebnis im Anhang.

k17_aufgabe2.ino

```
 6 void setup() {
 7     Serial.begin(115200);
 8     sensor.begin(20, 21);
 9     sensor.setOrientation(STK8BA58::PICOBOY_COLOR_PLUS);
10     delay(100);
11     for (int bereich = 2; bereich <= 8; bereich = bereich * 2) {
12         sensor.setRange(bereich);
13         int16_t x, y, z;
14         sensor.readRaw(x, y, z);
15         Serial.printf("+-%d g: roh x = %d\n", bereich, x);
16     }
17 }
```

Aufgabe 3 ⭐⭐ **Schüttelzähler** Zähle, wie oft der PicoBoy kräftig geschüttelt wird, und gib jedes Schütteln im seriellen Monitor aus. Rechne dazu die Gesamtstärke $\sqrt{(x^2 + y^2 + z^2)}$ aus (`sqrt`). In Ruhe ist sie etwa 1 g. Ein Schütteln liegt vor, wenn sie über 1,6 g steigt. Achtung: Ein Schütteln dauert viele Messungen lang. Es soll trotzdem nur einmal zählen (denk an die Flankenerkennung bei den Tasten in Kapitel 9).

Aufgabe 4 ⭐⭐ **Ein Pfeil nach unten** Zeichne einen großen Pfeil, der immer zur tieferen Seite zeigt: nach links, rechts, oben oder unten, je nachdem, welche Achse stärker geneigt ist. Liegt das Gerät flach (beide Werte unter 0,15 g), erscheint ein grüner Kreis.

Aufgabe 5 ⭐⭐⭐ Kugel mit Schwung Die Kugel aus 17.6 soll sich wie eine echte Kugel verhalten: Die Neigung ändert nicht die Position, sondern die **Geschwindigkeit** (`vx`, `vy`). Eine Reibung bremst sie in jedem Bild ein wenig (zum Beispiel `vx = vx * 0.98`). An den Wänden prallt sie ab und verliert dabei die Hälfte ihres Tempos. Irgendwo auf dem Brett ist ein Loch. Rollt die Kugel hinein, ist das Spiel gewonnen, und die gebrauchte Zeit wird angezeigt.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

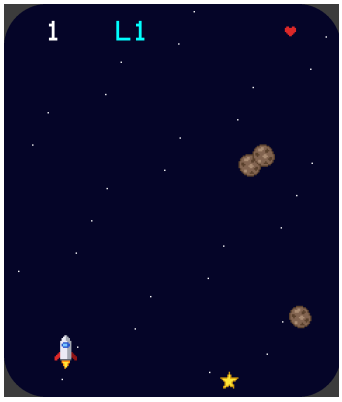
- `#include <STK8BA58.h>`, `STK8BA58 sensor;`, in `setup()`: `sensor.begin(20, 21);` und `sensor.setOrientation(STK8BA58::PICOBOY_COLOR_PLUS);`. Nie `begin()` ohne Pins: GP4 ist der Joystick.
- `float x, y, z; sensor.readAcceleration(x, y, z);` liefert die Beschleunigung in g. Flach: x = 0, y = 0, z = 1.
- x nach rechts, y nach unten, z aus dem Display. Die tiefere Seite misst einen negativen Wert.
- `setRange(2/4/8)`: ±2 g ist am feinsten (1024 Stufen pro g) und reicht zum Kippen.
- Glätten: `glatt = glatt + (roh - glatt) * 0.2;`
- Kugel rollt zur tiefen Seite: Position minus Messwert mal Tempo. Blase steigt zur hohen Seite: Position plus Messwert.

KAPITEL 18

Projekt: Ein ganzes Spiel

Worum geht's? Du baust ein vollständiges Spiel: „Sternenjäger". Eine Rakete sammelt Sterne und weicht Meteoren aus. Dabei kommt fast alles zusammen, was du bisher gelernt hast: Canvas, Sprites, Kollisionen, der Neigungssensor, Töne, Strukturen und Arrays. Neu ist, wie man ein Spiel in **Zustände** gliedert: Startbild, Spiel, Spielende. Vor allem lernst du, ein größeres Programm in Stufen zu bauen, die jede für sich laufen.

18.1 Die Idee und der Plan



Bevor du die erste Zeile schreibst, beschreibst du das Spiel in wenigen Sätzen:

- Unten fliegt eine **Rakete**. Du steuerst sie mit dem Joystick nach links und rechts oder durch Kippen des PicoBoy.
- Von oben fallen **Meteore**. Trifft einer die Rakete, kostet das ein **Leben**. Am Anfang hast du drei.
- Ab und zu fällt ein **Stern**. Fängst du ihn, gibt es einen **Punkt**.
- Alle fünf Punkte steigt das **Level**: Alles fällt schneller.
- Ohne Leben ist das Spiel zu Ende. Der beste Punktestand bleibt als **Rekord** stehen, bis du den PicoBoy ausschaltest.

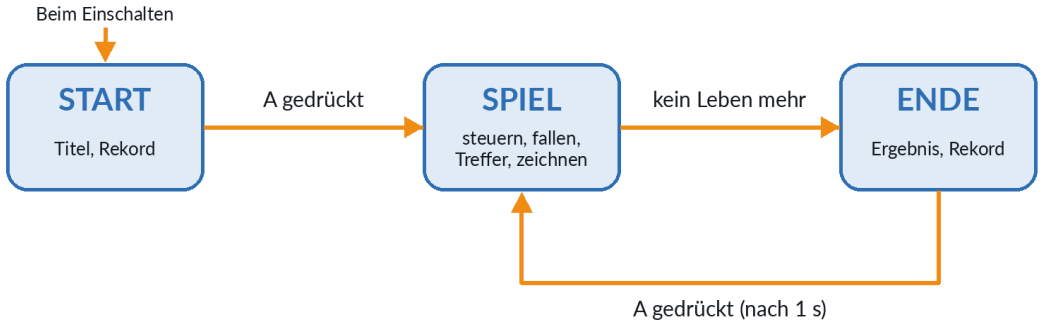
Ein ganzes Spiel auf einmal zu schreiben und erst am Ende zu testen, geht fast immer schief: Es gibt zehn Fehler gleichzeitig, und du weißt nicht, wo du suchen sollst. Baue das Spiel deshalb in **Stufen**. Jede Stufe ist ein lauffähiger Sketch, den du testest, bevor die nächste dazukommt:

Stufe	Sketch	Was kommt dazu?	Test: Das muss klappen
1	`k18_stufe1`	Zustände, Rakete	A startet, Rakete fährt, B beendet, A startet neu
2	`k18_stufe2`	Meteore, Treffer, Leben	Treffer kosten ein Leben, bei 0 ist Schluss
3	`k18_spiel`	Sterne, Punkte, Level, Rekord, Ton, LED, Kippen	das fertige Spiel

Die Bilder der Rakete, des Meteors, des Sterns und des Herzens für die Leben stammen aus Kapitel 15. Sie sind mit `bild2c` umgewandelt und stehen zusammen in der Datei `bilder.h` im Sketch-Ordner.

18.2 Zustände

Ein Spiel zeigt nicht immer dasselbe. Zuerst kommt ein **Startbild**, dann das eigentliche **Spiel**, am Ende eine Anzeige mit dem Ergebnis. Was der PicoBoy gerade tun soll und wie er auf Tasten reagiert, hängt davon ab, in welchem **Zustand** das Spiel ist. Die Taste A zum Beispiel startet im Startbild das Spiel, während des Spiels tut sie nichts.



Die Pfeile heißen **Übergänge**. An ihnen steht, was passieren muss, damit das Spiel in den nächsten Zustand wechselt. Ein solches Bild ist ein guter Plan: Jeder Kasten wird ein `case`, jeder Pfeil eine `if`-Abfrage.

Für die Zustände bietet sich ein `enum` an (Kapitel 13): eine Liste von Namen, die eine Variable annehmen kann. Stufe 1 enthält nur das Gerüst: drei Zustände, eine steuerbare Rakete, und zum Testen beendet die Taste B das Spiel.

k18_stufe1.ino

```
18 enum Zustand { START, SPIEL, ENDE };
19
20 Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
21 GFXcanvas16 bild(240, 280);
22
23 Zustand zustand = START;
24 bool a_vorher = false;
25 unsigned long letztes_bild = 0;
26 float rakete_x = 112;
27
28 bool a_gedrueckt() {
29     bool jetzt = digitalRead(TASTE_A) == LOW;
30     bool neu = jetzt && !a_vorher;
31     a_vorher = jetzt;
32     return neu;
33 }
```



- **Zeile 18:** `enum Zustand { START, SPIEL, ENDE };` legt den neuen Typ `Zustand` mit drei möglichen Werten an.
- **Zeile 23:** Die Variable `zustand` beginnt mit `START`.
- **Zeile 28 bis 33:** `a_gedrueckt()` ist die Flankenerkennung aus Kapitel 9 als Funktion: Sie liefert genau einmal pro Druck `true`.

k18_stufe1.ino

```
46 void loop() {
47     if (millis() - letztes_bild < TAKT) {
48         return;
49     }
50     letztes_bild = millis();
51 
52     bild.fillScreen(tft.color565(10, 10, 40));
53     bild.setTextSize(2);
54     switch (zustand) {
55         case START:
56             bild.setTextColor(ST77XX_YELLOW);
57             bild.setCursor(66, 100);
58             bild.print("STARTBILD");
59             bild.setTextColor(ST77XX_GREEN);
60             bild.setCursor(72, 200);
61             bild.print("A: Start");
62             if (a_gedrueckt()) {
63                 rakete_x = 112;
64                 zustand = SPIEL;
65             }
66             break;
67         case SPIEL:
68             if (digitalRead(TASTE_LINKS) == LOW) {
69                 rakete_x = rakete_x - RAKETE_TEMPO;
70             }
71             if (digitalRead(TASTE_RECHTS) == LOW) {
72                 rakete_x = rakete_x + RAKETE_TEMPO;
73             }
74             rakete_x = constrain(rakete_x, 20, 220 - RAKETE_BREITE);
75             bild.drawRGBBitmap(rakete_x, RAKETE_Y, rakete_pixel, rakete_maske,
76                                RAKETE_BREITE, RAKETE_HOEHE);
77             bild.setTextColor(ST77XX_WHITE);
78             bild.setCursor(48, 100);
79             bild.print("B: Spiel aus");
80             if (digitalRead(TASTE_B) == LOW) {      // nur zum Testen
81                 zustand = ENDE;
82             }
83             break;
84         case ENDE:
85             bild.setTextColor(ST77XX_RED);
86             bild.setCursor(48, 100);
87             bild.print("SPIEL VORBEI");
88             bild.setTextColor(ST77XX_GREEN);
89             bild.setCursor(60, 200);
90             bild.print("A: Nochmal");
91             if (a_gedrueckt()) {
92                 rakete_x = 112;
93                 zustand = SPIEL;
94             }
95             break;
96     }
97     tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
98 }
```

- **Zeile 54:** `switch (zustand)` wählt, was in diesem Bild passiert. Jeder Zustand zeichnet sein eigenes Bild und prüft seine eigenen Übergänge.
- **Zeile 62 bis 65, 80 bis 82, 91 bis 94:** die drei Pfeile aus dem Plan. Ein Übergang ist nur eine Zuweisung: `zustand = SPIEL;`. Ab dem nächsten Bild läuft dann der andere `case`.
- **Zeile 97:** Das Übertragen steht **nach** dem `switch`, einmal für alle Zustände.

Testen: Startet A das Spiel? Fährt die Rakete und bleibt am Rand stehen? Führt B zum Spielende und A wieder ins Spiel? Erst wenn alles klappt, geht es weiter.

18.3 Meteore, Treffer und Leben

Stufe 2 macht daraus ein Spiel. Die Meteore sind Strukturen in einem Array, wie die Bälle in Kapitel 14. Damit der `switch` übersichtlich bleibt, wandert jeder Teil in eine eigene Funktion. Das fertige Spiel `k18_spiel` übernimmt diesen Aufbau; ab hier siehst du es Abschnitt für Abschnitt.

Der Kopf: Einstellungen und Daten

k18_spiel.ino

```
16 #define TAKT 33                    // ms pro Bild
17 #define RAKETE_Y 236               // die Rakete fliegt immer auf dieser Hoehe
18 #define RAKETE_TEMPO 4             // Pixel pro Bild mit dem Joystick
19 #define KIPP_TEMPO 12              // Pixel pro Bild bei 1 g Neigung
20 #define TOTZONE 0.08               // so wenig Neigung wird nicht beachtet
21 #define METEORE 4
22 #define START_LEBEN 3
23 #define PUNKTE_PRO_LEVEL 5
24
25 enum Zustand { START, SPIEL, ENDE };
26
27 struct Objekt {
28     float x, y;                    // linke obere Ecke
29     float tempo;                   // Pixel pro Bild nach unten
30 };
31
32 struct Rechteck {
33     int x, y;
34     int b, h;
35 };
36
37 Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
38 GFXcanvas16 bild(240, 280);
39 STK8BA58 sensor;
40
41 Zustand zustand = START;
42 bool sensor_da = false;
43 bool a_vorher = false;
44 unsigned long letztes_bild = 0;
45 unsigned long ende_zeit = 0;       // wann das letzte Spiel endete
46 unsigned long treffer_zeit = 0;    // wann die Rakete zuletzt getroffen wurde
```

```
48 float rakete_x = 112;
49 Objekt meteore[METEORE];
50 Objekt stern;
51 int punkte = 0;
52 int leben = START_LEBEN;
53 int level = 1;
54 int rekord = 0;                    // nur im RAM: nach dem Ausschalten weg
55 bool neuer_rekord = false;
```

- **Zeile 16 bis 23:** Alle Zahlen, an denen du später drehen willst, stehen oben als Konstanten. Willst du fünf Leben, änderst du nur Zeile 22.
- **Zeile 27 bis 30:** Ein `Objekt` ist alles, was herunterfällt: Position und Tempo. Meteore und Stern benutzen dieselbe Struktur.
- **Zeile 41 bis 55:** Der ganze Zustand des Spiels steht in wenigen globalen Variablen. Die Rakete braucht nur `rakete_x`; ihre Höhe ist fest (`RAKETE_Y`).

Neu starten und nachschieben

k18_spiel.ino

```
79 void meteor_nach_oben(int i) {
80     meteore[i].x = random(20, 220 - METEOR_BREITE);
81     meteore[i].y = random(-200, -METEOR_HOEHE);
82     meteore[i].tempo = random(15, 30) / 10.0 + level * 0.5;
83 }
84
85 void stern_nach_oben() {
86     stern.x = random(20, 220 - STERN_BREITE);
87     stern.y = random(-60, -STERN_HOEHE);
88     stern.tempo = 2.0 + level * 0.3;
89 }
90
91 void neues_spiel() {
92     randomSeed(micros());          // der Zeitpunkt des Drucks auf A ist Zufall
93     punkte = 0;
94     leben = START_LEBEN;
95     level = 1;
96     neuer_rekord = false;
97     rakete_x = 112;
98     for (int i = 0; i < METEORE; i++) {
99         meteor_nach_oben(i);
100    }
101    stern_nach_oben();
102    zustand = SPIEL;
103 }
```

- `meteor_nach_oben(i)` setzt einen Meteor an eine zufällige Stelle **über** dem Bildschirm. Weil jeder eine andere Höhe bekommt, kommen sie nacheinander herunter. Ihr Tempo wächst mit dem Level.
- `neues_spiel()` setzt alles auf den Anfang und wechselt in den Zustand `SPIEL`.

- **Zeile 92:** `randomSeed(micros())`. Ohne diese Zeile fielen die Meteore nach jedem Einschalten an denselben Stellen (Kapitel 5). Der Zeitpunkt, zu dem du A drückst, ist auf die Mikrosekunde genau nie gleich. Er ist ein guter Startwert für den Zufall.

Ein Schritt des Spiels

k18_spiel.ino

```
126 void spiel_schritt() {
127     steuern();
128     Rechteck rakete = { (int)rakete_x + 4, RAKETE_Y + 2, 8, 20 };
129
130     // Meteore: fallen, treffen, kommen oben wieder
131     for (int i = 0; i < METEORE; i++) {
132         meteore[i].y = meteore[i].y + meteore[i].tempo;
133         Rechteck m = { (int)meteore[i].x + 2, (int)meteore[i].y + 2, 12, 12 };
134         if (trifft(rakete, m)) {
135             leben = leben - 1;
136             treffer_zeit = millis();
137             tone(LAUTSPRECHER, 150, 300);
138             meteor_nach_oben(i);
139         } else if (meteore[i].y > 280) {
140             meteor_nach_oben(i);
141         }
142     }
```

- **Zeile 128:** Die Trefferbox der Rakete ist ihr Rumpf, wie in Kapitel 16.4. Der Meteor bekommt in Zeile 133 seinen Kern als Trefferbox.
- **Zeile 134 bis 138:** Ein Treffer kostet ein Leben, merkt sich die Zeit des Treffers und spielt einen tiefen Ton. Der Meteor kommt sofort wieder nach oben. Sonst steckte er im nächsten Bild noch in der Rakete und kostete ein weiteres Leben (Kapitel 16.5).
- **Zeile 139 und 140:** Ein Meteor, der unten hinausgefallen ist, kommt ebenfalls wieder nach oben. Die Meteore werden also nie neu angelegt, sondern immer wieder verwendet.

Testen (Stufe 2): Kostet jeder Treffer genau ein Leben? Verschwindet ein Herz? Endet das Spiel bei 0?

18.4 Sterne, Punkte und Level

Der Stern funktioniert wie ein Meteor, nur mit anderer Wirkung:

k18_spiel.ino

```
144     // der Stern: einsammeln gibt einen Punkt
145     stern.y = stern.y + stern.tempo;
146     Rechteck s = { (int)stern.x, (int)stern.y, STERN_BREITE, STERN_HOEHE };
147     if (trifft(rakete, s)) {
148         punkte = punkte + 1;
149         tone(LAUTSPRECHER, 1320, 80);
```

```
150         if (punkte % PUNKTE_PRO_LEVEL == 0) {
151             level = level + 1;
152         }
153         stern_nach_oben();
154     } else if (stern.y > 280) {
155         stern_nach_oben();
156     }
157
158     // rote LED leuchtet kurz nach einem Treffer
159     digitalWrite(LED_ROT, millis() - treffer_zeit < 300);
160
161     if (leben <= 0) {
162         if (punkte > rekord) {
163             rekord = punkte;
164             neuer_rekord = true;
165         }
166         Serial.printf("Spiel vorbei. Punkte: %d, Level %d, Rekord %d\n", punkte,
              level, rekord);
167         ende_zeit = millis();
168         zustand = ENDE;
169     }
170 }
```

- **Zeile 150:** `punkte % PUNKTE_PRO_LEVEL == 0` ist bei 5, 10, 15 … Punkten wahr. Dann steigt das Level, und alle Meteore, die danach neu starten, sind schneller.
- **Zeile 159:** Die rote LED leuchtet, solange der letzte Treffer weniger als 300 ms zurückliegt. Das funktioniert, ohne dass das Spiel warten muss.
- **Zeile 161 bis 169: Spielende.** Sind die Leben aufgebraucht, wird der Rekord geprüft und das Ergebnis im seriellen Monitor ausgegeben. Dann wechselt das Spiel in den Zustand `ENDE`. `ende_zeit` merkt sich, wann.

18.5 Steuern mit Joystick und Kippen

k18_spiel.ino

```
107 void steuern() {
108     if (digitalRead(TASTE_LINKS) == LOW) {
109         rakete_x = rakete_x - RAKETE_TEMPO;
110     }
111     if (digitalRead(TASTE_RECHTS) == LOW) {
112         rakete_x = rakete_x + RAKETE_TEMPO;
113     }
114     if (sensor_da) {
115         float x, y, z;
116         sensor.readAcceleration(x, y, z);
117         if (abs(x) > TOTZONE) {
118             rakete_x = rakete_x - x * KIPP_TEMPO;   // tiefere Seite: x negativ
119         }
120     }
121     rakete_x = constrain(rakete_x, 20, 220 - RAKETE_BREITE);
122 }
```

Die Rakete lässt sich auf zwei Arten steuern, sogar gleichzeitig: mit dem Joystick und durch Kippen.

- Der Sensor wird nur gelesen, wenn er beim Start geantwortet hat (`sensor_da`, gesetzt in `setup()`). So läuft das Spiel auch, falls der Sensor einmal fehlt.
- **Zeile 117: die Totzone.** Kaum jemand hält den PicoBoy genau waagrecht. Ohne Totzone würde die Rakete schon bei einer winzigen Neigung langsam wegdriften. Neigungen unter 0,08 g werden deshalb nicht beachtet.
- **Zeile 118:** wie die Kugel in Kapitel 17.6: tiefere Seite, negativer Wert, also abziehen.

18.6 Zeichnen

k18_spiel.ino

```
174 void zeichne_himmel() {
175     bild.fillScreen(tft.color565(10, 10, 40));
176     // feste "Sterne" im Hintergrund, aus der Nummer berechnet
177     for (int i = 0; i < 30; i++) {
178         bild.drawPixel((i * 73) % 240, (i * 131) % 280, ST77XX_WHITE);
179     }
180 }
181
182 void zeichne_start() {
183     zeichne_himmel();
184     text_mitte("STERNEN-", 50, 3, ST77XX_YELLOW);
185     text_mitte("JAEGER", 80, 3, ST77XX_YELLOW);
186     bild.drawRGBBitmap(112, 125, rakete_pixel, rakete_maske, RAKETE_BREITE,
          RAKETE_HOEHE);
187     text_mitte("Sterne sammeln,", 165, 1, ST77XX_WHITE);
188     text_mitte("Meteoren ausweichen", 177, 1, ST77XX_WHITE);
189     text_mitte("Rekord: " + String(rekord), 200, 2, ST77XX_CYAN);
190     text_mitte("A: Start", 235, 2, ST77XX_GREEN);
191 }
192
193 void zeichne_spiel() {
194     zeichne_himmel();
195     bild.drawRGBBitmap(stern.x, stern.y, stern_pixel, stern_maske, STERN_BREITE,
          STERN_HOEHE);
196     for (int i = 0; i < METEORE; i++) {
197         bild.drawRGBBitmap(meteore[i].x, meteore[i].y, meteor_pixel,
              meteor_maske,
198                            METEOR_BREITE, METEOR_HOEHE);
199     }
200     // nach einem Treffer blinkt die Rakete eine Sekunde lang
201     bool sichtbar = millis() - treffer_zeit > 1000 || (millis() / 100) % 2 == 0;
202     if (sichtbar) {
203         bild.drawRGBBitmap(rakete_x, RAKETE_Y, rakete_pixel, rakete_maske,
204                            RAKETE_BREITE, RAKETE_HOEHE);
205     }
206     // oben: Punkte, Level und Leben als Herzen
207     bild.setTextSize(2);
208     bild.setTextColor(ST77XX_WHITE);
```

```
209     bild.setCursor(30, 12);
210     bild.print(punkte);
211     bild.setTextColor(ST77XX_CYAN);
212     bild.setCursor(80, 12);
213     bild.printf("L%d", level);
214     for (int i = 0; i < leben; i++) {
215         bild.drawRGBBitmap(200 - i * 12, 16, herz_pixel, herz_maske, HERZ_BREITE,
              HERZ_HOEHE);
216     }
217 }
```

- **`zeichne_himmel()`** ist der Hintergrund aller drei Zustände. Die 30 weißen Punkte sind feste Sterne. Ihre Lage wird aus der Nummer `i` berechnet: `(i * 73) % 240` springt wild hin und her, ist aber bei jedem Bild gleich. So flimmern sie nicht, und es braucht kein Array.
- **`zeichne_start()`** benutzt `text_mitte` (Zeile 72 bis 77): Ein Zeichen der eingebauten Schrift ist bei Größe 1 genau 6 Pixel breit. Ein Text mit `n` Zeichen ist bei Größe `g` also `n * 6 * g` Pixel breit; seine linke Kante liegt eine halbe Breite links von der Mitte (x = 120).
- **Zeile 201: Blinken nach einem Treffer.** Eine Sekunde lang nach einem Treffer ist `millis() / 100` abwechselnd gerade und ungerade, und die Rakete wird nur jedes zweite Zehntel gezeichnet. Der Spieler sieht sofort: Das war ein Treffer.
- Die Leben werden als Herzen gezeichnet, von rechts nach links.

18.7 Die Hauptschleife und der Rekord

k18_spiel.ino

```
247 void loop() {
248     if (millis() - letztes_bild < TAKT) {
249         return;
250     }
251     letztes_bild = millis();
252 
253     switch (zustand) {
254         case START:
255             if (a_gedrueckt()) {
256                 neues_spiel();
257             }
258             zeichne_start();
259             break;
260         case SPIEL:
261             spiel_schritt();
262             zeichne_spiel();
263             break;
```

```
264         case ENDE:
265             // erst nach einer Sekunde: nicht aus Versehen gleich weiterspielen
266             if (a_gedrueckt() && millis() - ende_zeit > 1000) {
267                 neues_spiel();
268             }
269             zeichne_ende();
270             break;
271     }
272     tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
273 }
```



Durch die Funktionen ist `loop()` kurz geworden und liest sich fast wie der Plan aus 18.2. Im Zustand `ENDE` startet A erst nach einer Sekunde ein neues Spiel. Sonst könnte ein Spieler, der gerade hektisch drückt, das Ergebnis gar nicht sehen.

Der Prüflauf auf dem PC hat einmal gespielt, mit festen Joystick-Bewegungen und ohne zu kippen:

```
Serieller Monitor
Spiel vorbei. Punkte: 1, Level 1, Rekord 1
```

Der Rekord steht in der Variablen `rekord` im Arbeitsspeicher. Er bleibt, solange der PicoBoy läuft, auch über viele Spiele. Schaltest du aus oder drückst RES, ist er weg. Wie man Daten dauerhaft im Flash-Speicher ablegt, so dass sie das Ausschalten überstehen, zeigt Kapitel 25.

18.8 Ideen zum Weiterbauen

Das Spiel ist fertig, aber noch lange nicht zu Ende gedacht. Baue immer nur **eine** Idee auf einmal ein und teste sie, bevor die nächste kommt. Speichere vorher die funktionierende Fassung unter neuem Namen (**Datei → Speichern unter…**).

Idee	Was du dafür brauchst
Pause mit der Taste B	ein vierter Zustand `PAUSE`
nach einem Treffer kurz unverwundbar	`treffer_zeit` gibt es schon
Bestenliste mit den drei besten Ergebnissen	ein Array, einsortieren
ein Herz fällt selten und bringt ein Leben	ein weiteres `Objekt`, `random`
der Stern dreht sich	die Münzbilder aus Kapitel 15.6
Musik beim Spielende	`tone` und eine Melodie aus Kapitel 10
Rekord bleibt nach dem Ausschalten	Kapitel 25

Übungsaufgaben

Aufgabe 1 ⭐ **An den Schrauben drehen** Ändere nur Konstanten oben im Sketch: fünf Leben statt drei, sechs Meteore, ein neues Level erst alle zehn Punkte. Welche Zeilen änderst du? Was wird leichter, was schwerer?

Aufgabe 2 ⭐ **Pause** Mit der Taste B soll das Spiel anhalten und „PAUSE“ anzeigen. Ein weiterer Druck auf B setzt es fort. Ergänze dazu einen Zustand `PAUSE`. Tipp: Auch B braucht eine Flankenerkennung, sonst wechselt das Spiel bei jedem Bild hin und her, solange B gedrückt ist.

Aufgabe 3 ⭐⭐ **Schutzschild** Solange die Rakete nach einem Treffer blinkt (eine Sekunde lang), soll sie kein weiteres Leben verlieren können. Die Meteore fliegen in dieser Zeit einfach hindurch.

Aufgabe 4 ⭐⭐ **Bestenliste** Statt nur eines Rekords merkt sich das Spiel die drei besten Ergebnisse in einem Array `int beste[3]`, sortiert vom größten zum kleinsten. Am Spielende wird das neue Ergebnis an der richtigen Stelle einsortiert, und das Startbild zeigt alle drei.

Aufgabe 5 ⭐⭐⭐ **Ein Extraleben** Selten (etwa jedes zehnte Mal, wenn ein Stern neu startet) fällt statt des Sterns ein Herz. Fängt die Rakete es, bekommt sie ein Leben dazu, höchstens aber fünf. Überlege: Wie merkt sich das Spiel, ob gerade ein Stern oder ein Herz fällt?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Größere Programme in **Stufen** bauen; jede Stufe läuft und wird getestet.
- Ein Spiel hat **Zustände** (Start, Spiel, Ende). `enum` für die Namen, `switch (zustand)` in `loop()`, ein Übergang ist `zustand = …;`.
- Jeder Teil in eine eigene Funktion: `neues_spiel`, `steuern`, `spiel_schritt`, `zeichne_…`. Dann bleibt `loop()` kurz.
- Alle Einstellungen als Konstanten oben im Sketch.
- `randomSeed(micros())` beim Druck auf Start macht jedes Spiel anders.
- Nach einem Treffer das Objekt sofort neu setzen; mit gemerkter Zeit (`treffer_zeit`) lassen sich Blinken und LED steuern, ohne zu warten.
- Ein Rekord in einer Variablen hält nur bis zum Ausschalten (dauerhaft: Kapitel 25).

KAPITEL 19

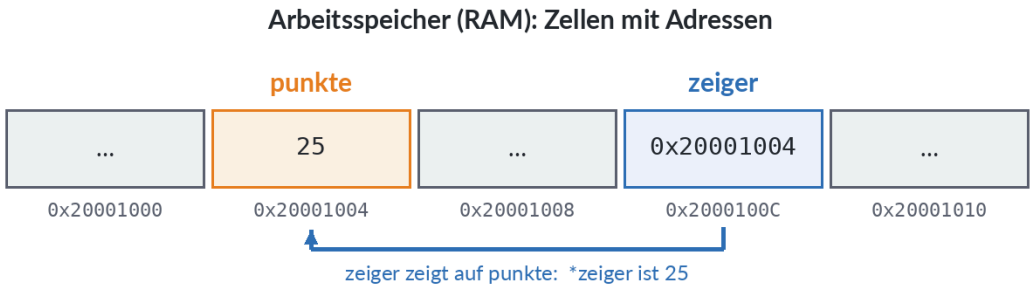
Zeiger und Speicher

Worum geht's? Jede Variable wohnt an einer bestimmten Stelle im Speicher, und diese Stelle hat eine Nummer: ihre **Adresse**. Du lernst, was ein **Zeiger** ist, wie Funktionen über Zeiger die Variablen des Aufrufers ändern und was Zeiger mit Arrays zu tun haben. Damit verstehst du endlich, was `bild.getBuffer()` liefert, und schreibst Pixel direkt in den Bildpuffer. Im zweiten Teil siehst du dir an, wie viel Speicher der PicoBoy hat, was die Meldung nach dem Kompilieren bedeutet und warum `const` Speicher spart.

19.1 Adressen und Zeiger

Der Arbeitsspeicher (RAM) ist eine lange Reihe von Speicherzellen, jede ein Byte groß. Jede Zelle hat eine Nummer, die **Adresse**. Legst du eine Variable an, sucht der Compiler einen freien Platz dafür aus. Ein `int` braucht 4 Byte, also vier Zellen hintereinander; seine Adresse ist die Nummer der ersten.

Ein **Zeiger** (englisch *pointer*) ist eine Variable, die eine Adresse speichert. Er „zeigt" damit auf eine andere Variable:



Die Adressen im Bild sind ausgedacht, aber so sehen Adressen auf dem PicoBoy aus: Der RAM beginnt bei der Adresse `0x20000000`. Adressen schreibt man fast immer **hexadezimal** (mit `0x` davor), also mit den Ziffern 0–9 und A–F, wie du es von den Farben kennst.

In C++ brauchst du dafür zwei Zeichen:

Schreibweise	Bedeutung
`&punkte`	die **Adresse** von `punkte` („wo steht punkte?")
`int *zeiger`	eine Variable `zeiger`, die die Adresse eines `int` speichert
`*zeiger`	**das, worauf** `zeiger` zeigt („was steht dort?")

k19_zeiger.ino

```
1  // Ein Zeiger merkt sich, WO eine Variable steht
2  void setup() {
3      Serial.begin(115200);
4      while (!Serial) {          // warten, bis der serielle Monitor offen ist
5          delay(10);
6      }
7
8      int punkte = 10;
9      int *zeiger = &punkte;     // zeiger zeigt auf punkte
10
11     Serial.printf("punkte  = %d\n", punkte);
12     Serial.printf("*zeiger = %d\n", *zeiger);
13
14     *zeiger = 25;              // ueber den Zeiger in punkte schreiben
15     Serial.printf("punkte  = %d\n", punkte);
16
17     *zeiger = *zeiger + 5;     // lesen, rechnen, zurueckschreiben
18     Serial.printf("punkte  = %d\n", punkte);
19 }
20
21 void loop() {
22 }
```

Serieller Monitor

```
punkte  = 10
*zeiger = 10
punkte  = 25
punkte  = 30
```

- **Zeile 9** legt den Zeiger an und gibt ihm gleich die Adresse von `punkte`. Der Stern in `int *zeiger` gehört zum Typ: „Zeiger auf `int`“.
- **Zeile 12** liest über den Zeiger: `*zeiger` ist der Wert, der an der gespeicherten Adresse steht, also 10.
- **Zeile 14** schreibt über den Zeiger. Geändert wird nicht der Zeiger, sondern `punkte`. Danach ist `punkte` 25, obwohl in dieser Zeile das Wort `punkte` gar nicht vorkommt.
- **Zeile 17** liest, rechnet und schreibt zurück: Aus 25 wird 30.

> Der Stern hat zwei Bedeutungen: In der **Deklaration** (`int *zeiger`) sagt er „das ist ein Zeiger“. **Vor einem Zeiger** im Programm (`*zeiger = 25`) heißt er „dort, wohin der Zeiger zeigt“. Das `&` vor einer Variable liefert ihre Adresse.

Auf dem PicoBoy ist jede Adresse 32 Bit lang. Ein Zeiger braucht also 4 Byte, egal ob er auf ein `int`, ein `float` oder ein Pixel zeigt.

> Ein Zeiger, dem du nie eine Adresse gegeben hast, zeigt irgendwohin. Schreibst du über ihn, überschreibst du irgendeine Speicherzelle. Der Compiler merkt das nicht, und eine Fehlermeldung gibt es nicht: Der Sketch macht dann still Unsinn oder hängt. Gib jedem Zeiger deshalb sofort eine Adresse, oder `nullptr` („zeigt auf nichts“), und prüfe vor dem Benutzen auf `nullptr`.

19.2 Zeiger als Parameter

Eine Funktion bekommt von ihren Argumenten nur **Kopien** (Kapitel 11). Ändert sie einen Parameter, bleibt das Original unberührt. Mit Zeigern geht es doch: Die Funktion bekommt die Adressen der Originale.

k19_tauschen.ino

```
1  // Eine Funktion, die zwei Variablen des Aufrufers vertauscht
2  void tausche_falsch(int a, int b) {
3      int merker = a;           // a und b sind nur Kopien
4      a = b;
5      b = merker;
6  }
7
8  void tausche(int *a, int *b) {
9      int merker = *a;          // a und b zeigen auf die Originale
10     *a = *b;
11     *b = merker;
12 }
13
14 void setup() {
15     Serial.begin(115200);
16     while (!Serial) {
17         delay(10);
18     }
19
20     int links = 3;
21     int rechts = 8;
22     tausche_falsch(links, rechts);
23     Serial.printf("nach tausche_falsch: %d %d\n", links, rechts);
24     tausche(&links, &rechts);
25     Serial.printf("nach tausche:        %d %d\n", links, rechts);
26 }
27
28 void loop() {
29 }
```

Serieller Monitor

```
nach tausche_falsch: 3 8
nach tausche:        8 3
```

- `tausche_falsch` (Zeilen 2–6) vertauscht nur ihre eigenen Kopien `a` und `b`. Die Variablen `links` und `rechts` in `setup()` bleiben, wie sie waren.

- `tausche` (Zeilen 8–12) bekommt zwei Zeiger. `*a` und `*b` sind die Originale.
- Beim Aufruf in Zeile 24 steht deshalb `&links` und `&rechts`: Übergeben werden die Adressen, nicht die Werte.

Referenzen. C++ kennt dafür noch eine bequemere Schreibweise, die **Referenz**: `void tausche(int &a, int &b)`. Dann schreibst du in der Funktion einfach `a` statt `*a` und beim Aufruf `tausche(links, rechts)` ohne `&`. Hinter den Kulissen übergibt der Compiler trotzdem die Adresse. Diese Form kennst du schon aus Kapitel 17: `sensor.readAcceleration(x, y, z)` schreibt seine Messwerte über Referenzen in deine drei Variablen. In Kapitel 22 schreibst du selbst eine solche Funktion.

19.3 Arrays und Zeiger

Ein Array liegt am Stück im Speicher: Element nach Element, ohne Lücke. Der **Name des Arrays** steht dabei für die Adresse seines ersten Elements. Du kannst ihn also direkt einem Zeiger geben:

k19_array_zeiger.ino

```
1  // Arrays und Zeiger: der Name eines Arrays ist die Adresse seines ersten
       Elements
2  int summe(const int *werte, int anzahl) {
3      int s = 0;
4      for (int i = 0; i < anzahl; i++) {
5          s = s + werte[i];      // dasselbe wie *(werte + i)
6      }
7      return s;
8  }
9
10 void setup() {
11     Serial.begin(115200);
12     while (!Serial) {
13         delay(10);
14     }
15
16     int wuerfe[5] = { 4, 6, 1, 3, 6 };
17     int *p = wuerfe;           // wie &wuerfe[0]
18
19     Serial.printf("p[0] = %d, *p = %d\n", p[0], *p);
20     Serial.printf("p[2] = %d, *(p + 2) = %d\n", p[2], *(p + 2));
21
22     p = p + 3;                 // drei Elemente weiter
23     Serial.printf("nach p + 3: *p = %d\n", *p);
24
25     Serial.printf("Summe: %d\n", summe(wuerfe, 5));
26     Serial.printf("Summe der letzten zwei: %d\n", summe(&wuerfe[3], 2));
27     Serial.printf("sizeof wuerfe = %d Bytes\n", (int)sizeof wuerfe);
28 }
29
30 void loop() {
31 }
```

Serieller Monitor

```
p[0] = 4, *p = 4
p[2] = 1, *(p + 2) = 1
nach p + 3: *p = 3
Summe: 20
Summe der letzten zwei: 9
sizeof wuerfe = 20 Bytes
```

- **Zeile 17:** `int *p = wuerfe;` ist dasselbe wie `int *p = &wuerfe[0];`.
- **Zeilen 19 und 20:** Einen Zeiger kannst du wie ein Array benutzen. `p[2]` und `*(p + 2)` bedeuten genau dasselbe: „zwei Elemente hinter der Adresse in `p`“.
- **Zeile 22:** `p + 3` rechnet in **Elementen**, nicht in Bytes. Der Compiler weiß, dass `p` auf `int` zeigt (4 Byte), und zählt 3 · 4 = 12 Byte weiter. Dieses Rechnen mit Zeigern heißt **Zeigerarithmetik**.
- **Zeile 2:** Die Funktion `summe` bekommt einen Zeiger auf den Anfang und die Anzahl. So bekommt **jede** Funktion ein Array: Kopiert wird nie das ganze Array, nur seine Adresse. Deshalb muss die Anzahl immer extra mitkommen. Das `const` verspricht, dass `summe` die Werte nur liest.
- **Zeile 26:** `&wuerfe[3]` ist die Adresse des vierten Elements. Die Funktion sieht ab dort ein Array mit zwei Elementen.

> **Die `sizeof`-Falle.** `sizeof wuerfe` liefert 20 (5 · 4 Byte), denn hier kennt der Compiler das ganze Array. In der Funktion `summe` ist `werte` aber nur ein Zeiger: `sizeof werte` wäre dort 4, die Größe einer Adresse. Darum geht die Anzahl immer als eigener Parameter mit.

Auch Texte sind Arrays (Kapitel 12): `"Hallo"` ist ein `char`-Array mit 6 Zeichen (das letzte ist die `0` am Ende). Funktionen wie `Serial.print` bekommen davon nur die Adresse des ersten Zeichens, als `const char *`, und lesen bis zur `0`.

19.4 Der Bildpuffer von `GFXcanvas16`

Jetzt kannst du verstehen, was in Kapitel 14 noch ein Rätsel war:

```
tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
```

`getBuffer()` liefert einen **Zeiger** vom Typ `uint16_t *` auf das erste Pixel des Canvas. Die Pixel liegen Zeile für Zeile hintereinander: erst die 240 Pixel der Zeile 0, dann die der Zeile 1 und so weiter. Das Pixel (x, y) hat also die Nummer `y * 240 + x`. `drawRGBBitmap` bekommt nur diese Anfangsadresse, Breite und Höhe, und schickt die 67 200 Zahlen zum Display.

Über den Zeiger kannst du auch selbst Pixel setzen, ganz ohne Zeichenbefehle:

k19_getbuffer.ino

```
14 void setup() {
15     tft.init(240, 280);
16     pinMode(TFT_LICHT, OUTPUT);
17     digitalWrite(TFT_LICHT, HIGH);
18
19     uint16_t *pixel = bild.getBuffer();      // Adresse des ersten Pixels
20     for (int y = 0; y < HOEHE; y++) {
21         for (int x = 0; x < BREITE; x++) {
22             int rot = x * 255 / (BREITE - 1);
23             int blau = y * 255 / (HOEHE - 1);
24             pixel[y * BREITE + x] = tft.color565(rot, 0, blau);
25         }
26     }
27
28     bild.setTextColor(ST77XX_WHITE);
29     bild.setTextSize(2);
30     bild.setCursor(54, 132);
31     bild.print("getBuffer()");
32     tft.drawRGBBitmap(0, 0, bild.getBuffer(), BREITE, HOEHE);
33 }
```

- **Zeile 19** holt den Zeiger einmal und merkt ihn sich in `pixel`.
- **Zeile 24** schreibt die Farbe direkt in den Speicher. Der Rotanteil wächst nach rechts, der Blauanteil nach unten.
- Danach zeichnen die gewohnten Befehle (Zeilen 28–31) **in denselben Speicher** weiter: Der Text liegt über dem Farbverlauf.

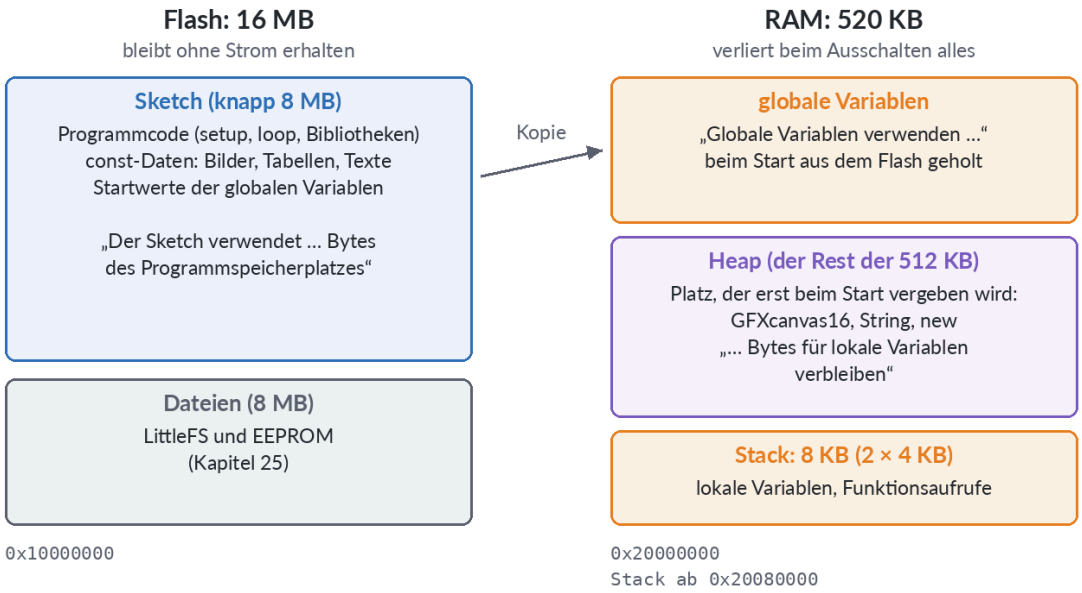


Das ist schneller als `bild.drawPixel(x, y, farbe)`, denn `drawPixel` prüft bei jedem Aufruf erst, ob das Pixel im Bild liegt, und rechnet die Drehung (`setRotation`) ein. Dafür bist du beim direkten Schreiben selbst verantwortlich: Ein `x` über 239 landet in der nächsten Zeile, ein `y` über 279 hinter dem Puffer, in fremdem Speicher.

> Die Farben im Puffer sind genau die 16-Bit-Zahlen aus Kapitel 6. `pixel[i] == ST77XX_BLACK` prüft also, ob ein Pixel schwarz ist. Damit kann ein Sketch im Bild „nachsehen“, was an einer Stelle gezeichnet ist. Das nutzt das Beispiel am Ende des Kapitels.

19.5 Wie viel Speicher hat der PicoBoy?

Der RP2350 hat zwei ganz verschiedene Speicher:



- Der **Flash** (16 MB) ist ein eigener Chip neben dem Prozessor. Sein Inhalt bleibt ohne Strom erhalten. Beim Hochladen landet dort der übersetzte Sketch. Mit der Einstellung **Flash Size: 16MB (Sketch: 8MB, FS: 8MB)** aus Kapitel 2 gehört knapp die Hälfte dem Sketch, der Rest ist für Dateien da (Kapitel 25).
- Der **RAM** (520 KB) sitzt im Prozessor. Er ist schnell, verliert aber beim Ausschalten alles. Hier liegen alle Variablen. 512 KB davon sind für den Sketch, die übrigen 2 · 4 KB nimmt der Kern für den **Stack** (Abschnitt 19.6).

Wie viel davon dein Sketch belegt, schreibt die IDE nach jedem **Überprüfen** und **Hochladen** ins Ausgabefenster. Für einen Sketch, der nur das Display blau füllt (`k19_leer`):

Ausgabe

```
Der Sketch verwendet 61652 Bytes (0%) des Programmspeicherplatzes. Das Maximum sind
  8380416 Bytes.
Globale Variablen verwenden 10184 Bytes (1%) des dynamischen Speichers, 514104 Bytes
  für lokale Variablen verbleiben. Das Maximum sind 524288 Bytes.
```

- **Programmspeicherplatz** ist der Flash: Programmcode und konstante Daten. Rund 60 KB, davon das meiste für den Kern und die Bibliotheken. Das Maximum von 8 380 416 Byte sind die knapp 8 MB, die die Einstellung „Sketch: 8MB“ übrig lässt.
- **Dynamischer Speicher** ist der RAM. 524 288 Byte sind genau 512 KB. Schon ein fast leerer Sketch belegt rund 10 KB für die Variablen von Kern und Bibliotheken (zum Beispiel für `Serial` und den USB-Anschluss).

Ein Bildpuffer, zweimal. Legst du den Bildpuffer selbst als globales Array an (`k19_puffer_global`) …

```
uint16_t puffer[240 * 280];        // 67 200 Pixel zu je 2 Bytes
```

… zählt der Compiler ihn mit. Die globalen Variablen wachsen um gut 134 KB:

```
Ausgabe
Der Sketch verwendet 62004 Bytes (0%) des Programmspeicherplatzes. Das Maximum sind
  8380416 Bytes.
Globale Variablen verwenden 144648 Bytes (27%) des dynamischen Speichers, 379640
  Bytes für lokale Variablen verbleiben. Das Maximum sind 524288 Bytes.
```

Derselbe Puffer als `GFXcanvas16 bild(240, 280);` (`k19_canvas`) taucht dagegen kaum auf:

```
Ausgabe
Der Sketch verwendet 62724 Bytes (0%) des Programmspeicherplatzes. Das Maximum sind
  8380416 Bytes.
Globale Variablen verwenden 10292 Bytes (1%) des dynamischen Speichers, 513996 Bytes
  für lokale Variablen verbleiben. Das Maximum sind 524288 Bytes.
```

Der Speicher fehlt aber trotzdem. `GFXcanvas16` holt sich seine 134 400 Byte erst **beim Start** des Sketches aus dem freien Rest des RAM, dem **Heap**. So steht es im Quelltext von Adafruit GFX (`Adafruit_GFX.cpp`, Konstruktor `GFXcanvas16::GFXcanvas16`): Er ruft `malloc(w * h * 2)` auf, die Funktion, die zur Laufzeit Speicher vom Heap anfordert. Der Compiler sieht davon nichts. Die Zahl „… Bytes für lokale Variablen verbleiben“ ist also nur eine Obergrenze für das, was beim Start noch zu haben ist.

19.6 Stack und Heap: wohin mit großen Daten?

Variablen landen an drei verschiedenen Orten:

Wo	was	wie groß
globaler Bereich	Variablen außerhalb von Funktionen	zählt der Compiler: „Globale Variablen verwenden …“
Heap	Speicher, der erst zur Laufzeit vergeben wird: `GFXcanvas16`, `String`, `new`	der Rest der 512 KB
Stack	lokale Variablen in Funktionen, Parameter, Rücksprungadressen	nur **8 KB**

Der **Stack** (deutsch Stapel) ist der Platz, an dem eine Funktion ihre lokalen Variablen ablegt. Beim Verlassen der Funktion wird der Platz wieder frei. Auf dem PicoBoy ist er klein: Laut Dokumentation des Kerns (`docs/multicore.rst`, Abschnitt „Stack Sizes“) hat der erste Kern 8 KB Stack. Laufen beide Kerne (Kapitel 24), teilen sie sich diese 8 KB.

Ein großes Array gehört **nie** als lokale Variable in eine Funktion:

```
void loop() {
    uint16_t puffer[240 * 280];    // 134 KB auf einem 8-KB-Stack!
    ...
}
```

Der Compiler übersetzt das ohne Fehler und ohne Warnung. Auf dem Gerät überschreibt das Array dann anderen Speicher; der Sketch stürzt ab oder macht Unsinn. Große Arrays gehören nach oben zu den globalen Variablen, Bildpuffer in einen `GFXcanvas16`.

Wenn der Heap nicht reicht. Drei Canvas in Bildschirmgröße brauchen 3 · 134 400 = 403 200 Byte, das geht. Ein vierter passt nicht mehr in die 512 KB. Dann liefert `malloc` keinen Speicher, und `getBuffer()` gibt `nullptr` zurück. Zeichnen in diesen Canvas bewirkt nichts, und `drawRGBBitmap` mit einem `nullptr` würde Unsinn schicken. Wer viele oder große Canvas benutzt, prüft deshalb in `setup()`:

```
if (bild.getBuffer() == nullptr) {
    Serial.println("Kein Platz fuer den Canvas!");
}
```

Nachsehen auf dem Gerät. Der Kern hat Funktionen, die den Heap messen: `rp2040.getTotalHeap()`, `rp2040.getUsedHeap()` und `rp2040.getFreeHeap()`. Dieser Sketch gibt sie aus, dazu die Adressen einiger Dinge. Er läuft nur auf dem PicoBoy:

k19_adressen.ino

```
// Wo liegt was im Speicher? Die Adressen zeigt nur der PicoBoy richtig.
#include <Adafruit_ST7789.h>

GFXcanvas16 bild(240, 280);
int zaehler = 0;                          // globale Variable
const int TABELLE[4] = { 2, 3, 5, 7 };    // konstante Tabelle

void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    int lokal = 1;                        // lokale Variable

    Serial.printf("globale Variable: %p\n", (void *)&zaehler);
    Serial.printf("lokale Variable:  %p\n", (void *)&lokal);
    Serial.printf("const-Tabelle:    %p\n", (void *)TABELLE);
    Serial.printf("Canvas-Puffer:    %p\n", (void *)bild.getBuffer());
    Serial.printf("Funktion setup:   %p\n", (void *)setup);
    Serial.println();
```

```
21     Serial.printf("Heap gesamt: %d Bytes\n", rp2040.getTotalHeap());
22     Serial.printf("Heap belegt: %d Bytes\n", rp2040.getUsedHeap());
23     Serial.printf("Heap frei:   %d Bytes\n", rp2040.getFreeHeap());
24 }
25
26 void loop() {
27 }
```

Was er ausgibt, ergibt sich aus der Speicheraufteilung des Kerns (`lib/rp2350/memmap_default.ld`): Globale Variablen und der Canvas-Puffer haben Adressen ab `0x2000…` (RAM), die lokale Variable eine Adresse ab `0x2008…` (der Stack am Ende des RAM), die `const`-Tabelle und die Funktion `setup` Adressen ab `0x1000…`. Das ist der Flash: Der Prozessor kann ihn direkt lesen, als wäre er Speicher. Der freie Heap liegt nach dem Start um die 134 400 Byte des Canvas niedriger als ohne ihn.

`%p` ist der Platzhalter für eine Adresse. Er erwartet einen „Zeiger auf irgendetwas“, `void *`; das `(void *)` vor jedem Wert wandelt den Zeiger in diese Form um.

19.7 `const`: Daten bleiben im Flash

In Kapitel 15 hat das Werkzeug `bild2c` Bilder als `const`-Arrays erzeugt. Das `const` hat einen guten Grund. Vergleiche ein Bild mit 100 · 100 Pixeln (20 000 Byte) einmal mit und einmal ohne `const`:

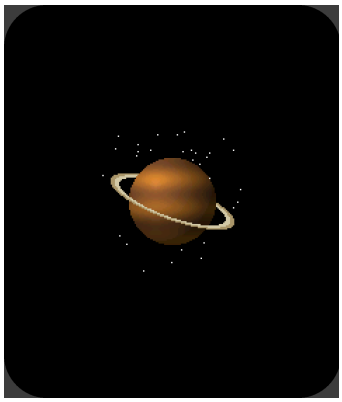
```
const uint16_t planet_pixel[10000] = { ... };    // k19_bild_const
uint16_t planet_pixel[10000] = { ... };          // k19_bild_ram
```

Mit `const`:

Ausgabe
```
Der Sketch verwendet 81780 Bytes (0%) des Programmspeicherplatzes. Das Maximum sind
  8380416 Bytes.
Globale Variablen verwenden 10184 Bytes (1%) des dynamischen Speichers, 514104 Bytes
  für lokale Variablen verbleiben. Das Maximum sind 524288 Bytes.
```

Ohne `const`:

Ausgabe
```
Der Sketch verwendet 61988 Bytes (0%) des Programmspeicherplatzes. Das Maximum sind
  8380416 Bytes.
Globale Variablen verwenden 30248 Bytes (5%) des dynamischen Speichers, 494040 Bytes
  für lokale Variablen verbleiben. Das Maximum sind 524288 Bytes.
```



- **Mit `const`** wächst der Programmspeicher um gut 20 000 Byte (verglichen mit `k19_leer`), der RAM bleibt gleich. Das Bild kann sich nie ändern, also darf es im Flash bleiben. Der Prozessor liest es von dort.
- **Ohne `const`** könnte der Sketch das Bild verändern. Also muss es in den RAM: Die globalen Variablen wachsen um 20 000 Byte. Die **Startwerte** stehen trotzdem im Flash, denn beim Einschalten ist der RAM leer. Der Kern kopiert sie vor `setup()` in den RAM. Die Meldung der IDE zählt diese Kopie im Flash allerdings nicht mit; so ist der Kern eingestellt (`platform.txt`, Zeile `recipe.size.regex`).

Das Bild sieht in beiden Fällen gleich aus. Der Unterschied: 20 KB RAM.

> Alles, was sich nie ändert, bekommt `const`: Bilder, Tabellen, Melodien, Level. Dann bleibt es im Flash (davon gibt es 8 MB) und belegt keinen RAM (davon gibt es 512 KB).

`PROGMEM` braucht man hier nicht. In Anleitungen für den Arduino Uno liest du oft:

```
const uint16_t bild[] PROGMEM = { ... };
uint16_t farbe = pgm_read_word(&bild[i]);
```

Der Uno hat einen Prozessor, der Daten im Flash nicht wie gewöhnlichen Speicher lesen kann; dort braucht man `PROGMEM` und die Lesefunktionen `pgm_read_…`. Der RP2350 kann das. Im Kern ist `PROGMEM` deshalb leer: In der Datei `ArduinoCore-API/api/deprecated-avr-comp/avr/pgmspace.h` des Kerns steht

```
#define PROGMEM
```

und `pgm_read_byte(addr)` liest dort einfach, was an der Adresse steht. Fremde Sketche mit `PROGMEM` laufen also auch auf dem PicoBoy, das Wort bewirkt nur nichts. Für eigene Sketche reicht `const`. Dasselbe gilt für das Makro `F("…")` um Texte: Es schadet nicht, nützt hier aber nichts.

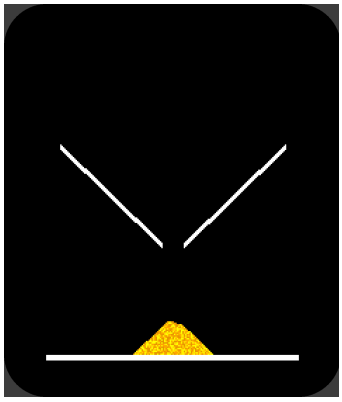
Ausprobieren: Rieselnder Sand

Jedes Sandkorn ist ein einziges Pixel im Bildpuffer. Ein Korn fällt, wenn unter ihm frei ist; sonst rutscht es schräg nach links oder rechts. Halte **A** gedrückt, dann rieselt Sand in den Trichter.

k19_sand.ino

```
34 void loop() {
35     if (digitalRead(TASTE_A) == LOW) {
36         for (int i = 0; i < 6; i++) {                // neue Koerner oben
37             int x = 100 + random(40);
```

```
38              pixel[40 * BREITE + x] = SANDFARBEN[random(4)];
39          }
40      }
41      for (int schritt = 0; schritt < 4; schritt++) {  // 4 Pixel pro Bild fallen
42          rieseln();
43      }
44      tft.drawRGBBitmap(0, 0, pixel, BREITE, HOEHE);
45  }
46
47  // Von unten nach oben: So faellt jedes Korn pro Durchgang nur ein Pixel.
48  void rieseln() {
49      for (int y = HOEHE - 2; y >= 0; y--) {
50          uint16_t *zeile = pixel + y * BREITE;         // Anfang der Zeile y
51          uint16_t *darunter = zeile + BREITE;          // Anfang der Zeile y + 1
52          for (int x = 1; x < BREITE - 1; x++) {
53              uint16_t korn = zeile[x];
54              if (korn != LEER && korn != WAND) {
55                  int seite = 1;                        // zuerst rechts ...
56                  if (random(2) == 0) {
57                      seite = -1;                       // ... oder zuerst links
58                  }
59                  if (darunter[x] == LEER) {
60                      darunter[x] = korn;
61                      zeile[x] = LEER;
62                  } else if (darunter[x + seite] == LEER) {
63                      darunter[x + seite] = korn;
64                      zeile[x] = LEER;
65                  } else if (darunter[x - seite] == LEER) {
66                      darunter[x - seite] = korn;
67                      zeile[x] = LEER;
68                  }
69              }
70          }
71      }
72  }
```



- Der Sketch merkt sich den Zeiger auf den Puffer in der globalen Variable `pixel` (Zeile 16) und benutzt ihn überall.
- **Zeile 38:** Ein neues Korn ist einfach eine Farbe an der Stelle `40 * BREITE + x`, also in Zeile 40. Die Farben stehen in der `const`-Tabelle `SANDFARBEN` im Flash.
- **Zeilen 50 und 51:** Zwei Zeiger auf den Anfang der aktuellen Zeile und der Zeile darunter. `zeile[x]` ist das Korn, `darunter[x]` der Platz darunter.
- **Zeile 49:** Die Schleife läuft von **unten nach oben**. Liefe sie nach unten, würde ein Korn, das gerade eine Zeile tiefer gerutscht ist, in derselben Runde gleich noch einmal bewegt und fiele in einem Rutsch durch.
- Die Wände sind weiße Pixel. Sie bewegen sich nicht, weil Zeile 54 nur Körner bewegt, die weder leer noch Wand sind.

Zum Verändern: Zeichne andere Hindernisse. Lass den Sand mit **B** wieder verschwinden. Lass die Körner mit dem Neigungssensor zur Seite rieseln.

Übungsaufgaben

Aufgabe 1 ⭐ **Was gibt das aus?** Überlege ohne PicoBoy, was der Sketch ausgibt. Zeichne dir dazu Kästchen für `a`, `b`, `p` und `q` und verfolge Zeile für Zeile, worauf die Zeiger zeigen. Prüfe danach am Gerät.

k19_aufgabe1.ino

```
// Aufgabe 1: Was gibt der Sketch aus? Erst ueberlegen, dann pruefen.
void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    int a = 5;
    int b = 7;
    int *p = &a;
    int *q = &b;
    *p = *p + *q;
    p = q;
    *p = 1;
    Serial.printf("a = %d, b = %d, *p = %d\n", a, b, *p);

    int zahlen[4] = { 10, 20, 30, 40 };
    int *r = zahlen + 1;
    Serial.printf("%d %d %d\n", *r, r[1], *(r + 2));
}

void loop() {
}
```

Aufgabe 2 ⭐ **Begrenzen** Schreibe eine Funktion `void begrenze(int *wert, int kleinster, int groesster)`, die einen Wert des Aufrufers in die Grenzen zwingt. Teste sie mit drei Variablen und den Werten 250, −12 und 100 (Grenzen 20 bis 220 bzw. 20 bis 260).

Aufgabe 3 ⭐⭐ **Streifen mit `p++`** Fülle einen Canvas mit waagerechten Streifen, je 20 Pixel hoch, abwechselnd orange und blau. Benutze dafür keinen Zeichenbefehl und keine Rechnung `y * 240 + x`, sondern einen Zeiger, der mit `*p = farbe; p++;` Pixel für Pixel durch den ganzen Puffer läuft.

Aufgabe 4 ⭐⭐ Ein großes Level Ein Spiel speichert sein Level als `uint8_t level[150][200]`, ein Byte pro Feld. Wie viele Bytes sind das? Lege das Array einmal mit und einmal ohne `const` an (es reicht, die ersten Zeilen anzugeben, der Rest ist dann 0) und benutze es im Sketch. Überprüfe beide Sketche und vergleiche die Meldungen mit `k19_leer`. Wo landet das Level jeweils?

Aufgabe 5 ⭐⭐⭐ Spiegelbild Schreibe eine Funktion `void spiegeln(uint16_t *puffer)`, die ein Bild von 240 · 280 Pixeln links und rechts vertauscht. Nimm für jede Zeile zwei Zeiger: einen auf das erste und einen auf das letzte Pixel. Tausche, was dort steht, und lass die beiden aufeinander zulaufen, bis sie sich treffen. Teste mit einem Text: Er muss danach in Spiegelschrift erscheinen.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `&x` ist die **Adresse** von `x`. `int *p = &x;` legt einen **Zeiger** an. `*p` ist der Wert, auf den `p` zeigt; `*p = 5` schreibt dorthin.
- Über Zeiger (oder Referenzen `int &a`) kann eine Funktion Variablen des Aufrufers ändern.
- Der Name eines Arrays ist die Adresse des ersten Elements. `p[i]` ist dasselbe wie `*(p + i)`. Funktionen bekommen Arrays als Zeiger, die Anzahl muss mitkommen.
- `bild.getBuffer()` liefert einen `uint16_t *` auf die Pixel des Canvas; Pixel (x, y) ist `pixel[y * 240 + x]`.
- Flash 16 MB (8 MB für den Sketch), RAM 520 KB (512 KB für Variablen, dazu 8 KB Stack). Die Meldung nach dem Überprüfen nennt Programmspeicher (Flash) und globale Variablen (RAM).
- `GFXcanvas16` holt seinen Speicher erst beim Start vom **Heap**; bei Platzmangel ist `getBuffer()` `nullptr`. Große Arrays nie lokal anlegen: Der **Stack** hat nur 8 KB.
- `const`-Daten bleiben im Flash und kosten keinen RAM. `PROGMEM` ist auf dem RP2350 leer und überflüssig.

KAPITEL 20

Eigene Bibliotheken – ein Blick in STK8BA58

Worum geht's? Seit Kapitel 3 benutzt du Bibliotheken: Adafruit ST7789, Adafruit GFX, Adafruit NeoPixel und STK8BA58. Jetzt siehst du dir an, wie eine Bibliothek gebaut ist, und zwar an STK8BA58, die zu diesem Buch geschrieben wurde. Du verteilst eine eigene Klasse auf mehrere Dateien, lernst den Unterschied zwischen `.h` und `.cpp` und machst daraus eine richtige Bibliothek, die in jedem Sketch mit `#include` bereitsteht.

20.1 Was ist eine Bibliothek?

Eine **Bibliothek** ist Programmcode, den jemand fertig geschrieben hat, damit andere ihn benutzen können, ohne ihn zu verstehen. Du rufst `tft.fillCircle(…)` auf und musst nicht wissen, wie man einen Kreis aus Pixeln baut oder wie die Pixel zum Display kommen.

Für die Arduino IDE ist eine Bibliothek nichts weiter als ein **Ordner** im Unterordner `libraries` deines Sketchbooks (Kapitel 2.4) mit einem festen Aufbau. So sieht STK8BA58 aus:

```
libraries/
    STK8BA58/
        library.properties     Steckbrief: Name, Version, Autor …
        keywords.txt           welche Wörter der Editor einfärbt
        src/
            STK8BA58.h         was die Bibliothek kann
            STK8BA58.cpp       wie sie es macht
        examples/
            Werte_anzeigen/Werte_anzeigen.ino
            Wasserwaage_PicoBoy/Wasserwaage_PicoBoy.ino
```

Alles Wichtige steckt in `src/`. Die anderen Dateien helfen der IDE und den Menschen, die die Bibliothek benutzen. Bevor du dir die einzelnen Dateien ansiehst, baust du selbst eine kleine Klasse und verteilst sie auf Dateien. Dann erkennst du in STK8BA58 alles wieder.

20.2 Eine Klasse in einem eigenen Tab

In Kapitel 13 stand deine erste Klasse direkt in der `.ino`-Datei. Das wird schnell unübersichtlich. Seit Kapitel 15 weißt du, wie eine zweite Datei als **Tab** in den Sketch kommt. Genauso kann eine Klasse in eine eigene Datei wandern.

Das Beispiel ist eine Klasse `Balken` für Anzeigen wie Lebensenergie oder Ladefortschritt. Sie steht ganz in der Datei `Balken.h`:

Balken.h

```
1  // Balken.h – ein Fortschrittsbalken fuer jedes Ziel von Adafruit GFX
2  // (Display oder Canvas). Alles steht in dieser einen Datei.
3  #pragma once
4  #include <Adafruit_GFX.h>
5
6  class Balken {
7  public:
8      Balken(int x, int y, int breite, int hoehe, uint16_t farbe)
9          : _x(x), _y(y), _breite(breite), _hoehe(hoehe), _farbe(farbe) {
10     }
11
12     // zeichnet Rahmen und Fuellung; prozent von 0 bis 100
13     void zeichne(Adafruit_GFX &ziel, int prozent) {
14         prozent = constrain(prozent, 0, 100);
15         int innen = _breite - 4;
16         int voll = innen * prozent / 100;
17         ziel.drawRect(_x, _y, _breite, _hoehe, _farbe);
18         ziel.fillRect(_x + 2, _y + 2, voll, _hoehe - 4, _farbe);
19         ziel.fillRect(_x + 2 + voll, _y + 2, innen - voll, _hoehe - 4, 0x0000);
20     }
21
22 private:
23     int _x, _y, _breite, _hoehe;
24     uint16_t _farbe;
25 };
```

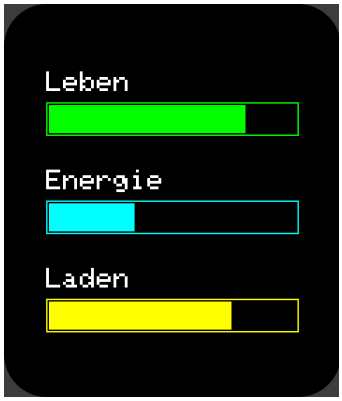
- **Zeile 3:** `#pragma once` sorgt dafür, dass der Compiler die Datei nur einmal liest, auch wenn sie mehrfach mit `#include` eingebunden wird. Sonst wäre die Klasse doppelt definiert, und das ist ein Fehler.
- **Zeile 4:** Eine `.h`-Datei muss selbst einbinden, was sie braucht. Die Arduino IDE ergänzt `#include <Arduino.h>` nur in der `.ino`-Datei von selbst, nicht in anderen Dateien. `Adafruit_GFX.h` bringt `Arduino.h` mit; deshalb kennt `Balken.h` auch `constrain`.
- **Zeilen 8 und 9:** Der **Konstruktor** speichert die Angaben. Die Form `: _x(x), _y(y), …` heißt **Initialisierungsliste**: Sie füllt die Attribute, bevor der Rumpf `{ }` läuft. So steht es auch in vielen Bibliotheken.
- **Zeilen 23 und 24:** Die Attribute beginnen mit einem Unterstrich. Das ist eine verbreitete Gewohnheit, damit `_x` (das Attribut) nicht mit `x` (dem Parameter) verwechselt wird.
- **Zeile 13:** `Adafruit_GFX &ziel` ist eine Referenz (Kapitel 19.2) auf „irgendetwas, das Adafruit GFX kann“. `Adafruit_ST7789` und `GFXcanvas16` sind beide von `Adafruit_GFX` **abgeleitet**: Sie können alles, was `Adafruit_GFX` kann, und noch mehr. Deshalb kannst du `zeichne` sowohl `tft` als auch `bild` geben.

Der Sketch selbst bleibt kurz:

k20_balken.ino

```
1  // Drei Balken mit der eigenen Klasse aus dem Tab Balken.h
2  #include <Adafruit_ST7789.h>
3  #include "Balken.h"
4
5  #define TFT_CS 10
6  #define TFT_DC 8
7  #define TFT_RST 9
8  #define TFT_LICHT 26
9
10 Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
11 GFXcanvas16 bild(240, 280);
12 Balken leben(30, 70, 180, 24, ST77XX_GREEN);
13 Balken energie(30, 140, 180, 24, ST77XX_CYAN);
14 Balken laden(30, 210, 180, 24, ST77XX_YELLOW);
15
16 void setup() {
17     tft.init(240, 280);
18     pinMode(TFT_LICHT, OUTPUT);
19     digitalWrite(TFT_LICHT, HIGH);
20     bild.setTextColor(ST77XX_WHITE);
21     bild.setTextSize(2);
22 }
23
24 void loop() {
25     int prozent = (millis() / 40) % 101;     // laeuft von 0 bis 100
26     bild.fillScreen(ST77XX_BLACK);
27     bild.setCursor(30, 48);
28     bild.print("Leben");
29     bild.setCursor(30, 118);
30     bild.print("Energie");
31     bild.setCursor(30, 188);
32     bild.print("Laden");
33     leben.zeichne(bild, 80);
34     energie.zeichne(bild, 35);
35     laden.zeichne(bild, prozent);
36     tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
37 }
```

Zeile 3 holt die Klasse mit Anführungszeichen herein: `"Balken.h"` liegt im Sketch-Ordner. Der Rest benutzt sie wie jede andere Klasse.



20.3 Trennen: `.h` und `.cpp`

Bibliotheken trennen ihre Klassen fast immer in zwei Dateien:

- Die **Header-Datei** `.h` (von *header*, Kopf) sagt, **was** die Klasse kann: welche Methoden es gibt, welche Parameter sie haben, was sie zurückgeben. Das ist die **Deklaration**.
- Die **Quelltextdatei** `.cpp` sagt, **wie** sie es macht: die Rümpfe der Methoden. Das ist die **Definition**.

Dieselbe Klasse, getrennt:

Balken.h

```
// Balken.h – ein Fortschrittsbalken fuer jedes Ziel von Adafruit GFX.
// Hier steht nur, WAS die Klasse kann (Deklaration).
#pragma once
#include <Adafruit_GFX.h>

class Balken {
public:
    Balken(int x, int y, int breite, int hoehe, uint16_t farbe);

    // zeichnet Rahmen und Fuellung; prozent von 0 bis 100
    void zeichne(Adafruit_GFX &ziel, int prozent);

private:
    int _x, _y, _breite, _hoehe;
    uint16_t _farbe;
};
```

Balken.cpp

```
// Balken.cpp – WIE die Klasse es macht (Definition)
#include "Balken.h"

Balken::Balken(int x, int y, int breite, int hoehe, uint16_t farbe)
    : _x(x), _y(y), _breite(breite), _hoehe(hoehe), _farbe(farbe) {
}

void Balken::zeichne(Adafruit_GFX &ziel, int prozent) {
    prozent = constrain(prozent, 0, 100);
    int innen = _breite - 4;
    int voll = innen * prozent / 100;
    ziel.drawRect(_x, _y, _breite, _hoehe, _farbe);
    ziel.fillRect(_x + 2, _y + 2, voll, _hoehe - 4, _farbe);
    ziel.fillRect(_x + 2 + voll, _y + 2, innen - voll, _hoehe - 4, 0x0000);
}
```

- In `Balken.h` stehen nur noch die Köpfe der Methoden, jeweils mit `;` statt eines Rumpfs.
- In `Balken.cpp` steht vor jedem Namen `Balken::`. Das heißt: „die Methode `zeichne`, die zur Klasse `Balken` gehört“. Ohne `Balken::` wäre es eine gewöhnliche Funktion, die von `_x` und `_farbe` nichts weiß.
- `Balken.cpp` bindet seinen eigenen Header ein, damit der Compiler die Klasse kennt.

Der Sketch `k20_balken_getrennt` ist bis auf den Kommentar derselbe wie `k20_balken`. Die IDE übersetzt jede `.cpp`-Datei im Sketch-Ordner automatisch mit und fügt am Ende alles zusammen.

Warum der Aufwand? Wer die Bibliothek benutzt, liest nur die `.h`-Datei: Sie ist kurz und zeigt alles, was man wissen muss. Und der Compiler übersetzt die `.cpp`-Datei nur einmal, auch wenn zehn Dateien den Header einbinden. Bei einer kleinen Klasse wie `Balken` reicht auch eine einzige `.h`-Datei; bei großen Bibliotheken ist die Trennung die Regel.

> Steht in einer `.h`-Datei eine **Funktion außerhalb einer Klasse** mit Rumpf, und binden zwei `.cpp`-Dateien diese `.h` ein, gibt es die Funktion zweimal. Der Linker (der Teil, der am Ende alles zusammenfügt) meldet dann `multiple definition of …`. Abhilfe: die Funktion in die `.cpp` verschieben oder `inline` davorschreiben. Methoden, deren Rumpf in der Klasse steht, sind von selbst `inline`.

20.4 Ein Blick in STK8BA58

Jetzt erkennst du den Aufbau der Sensor-Bibliothek. Die Header-Datei beginnt mit einem langen Kommentar, der erklärt, wie man sie benutzt. Danach kommt die Klasse:

STK8BA58.h

```
20 #ifndef STK8BA58_H
21 #define STK8BA58_H
22
23 #include <Arduino.h>
24 #include <Wire.h>
25
26 class STK8BA58 {
27 public:
28     // Ausrichtung der Achsen
29     enum Orientation {
30         CHIP,                // Achsen genau wie im Datenblatt des Chips
31         PICOBOY_COLOR_PLUS   // wie oben beschrieben für den PicoBoy Color Plus
32     };
33
34     static const uint8_t DEFAULT_ADDRESS = 0x18;   // SDO an GND
35     static const uint8_t CHIP_ID = 0x87;            // Inhalt von Register 0x00
36
37     explicit STK8BA58(TwoWire &wire = Wire, uint8_t address = DEFAULT_ADDRESS);
38
39     // Startet Wire, prüft die Chip-ID und stellt ±2 g, 125 Hz, Normalbetrieb
40     // ein. Gibt false zurück, wenn der Sensor nicht antwortet.
41     bool begin();
42 #if defined(ARDUINO_ARCH_RP2040)
43     // Wie begin(), legt vorher die I2C-Pins fest (nur RP2040/RP2350).
44     bool begin(int sdaPin, int sclPin);
45 #endif
```

```
47      // Messbereich ±2, ±4 oder ±8 g. Größerer Bereich = gröber aufgelöst.
48      bool setRange(int g);
49      int range() const { return _range; }
50
51      void setOrientation(Orientation orientation) { _orientation = orientation; }
52
53      // Neue Messwerte seit dem letzten Lesen?
54      bool accelerationAvailable();
55
56      // Beschleunigung in g (1 g = 9,81 m/s²). false bei einem I2C-Fehler.
57      bool readAcceleration(float &x, float &y, float &z);
58      // Dasselbe in m/s².
59      bool readAccelerationMS2(float &x, float &y, float &z);
60      // Rohwerte (-2048 … 2047), schon gedreht nach setOrientation().
61      bool readRaw(int16_t &x, int16_t &y, int16_t &z);
62
63      // Einzelne Achse in g (liest jeweils alle drei Achsen neu).
64      float readX();
65      float readY();
66      float readZ();
67
68      // Registerzugriff für eigene Experimente
69      uint8_t readRegister(uint8_t reg);
70      bool writeRegister(uint8_t reg, uint8_t value);
71
72  private:
73      bool readRegisters(uint8_t reg, uint8_t *buffer, size_t length);
74
75      TwoWire &_wire;
76      uint8_t _address;
77      int _range = 2;
78      Orientation _orientation = CHIP;
79  };
80
81  #endif
```

- **Zeilen 20, 21 und 81:** `#ifndef STK8BA58_H` … `#define STK8BA58_H` … `#endif` ist die ältere, überall gültige Form von `#pragma once`: „Wenn `STK8BA58_H` noch nicht definiert ist, definiere es und lies den Rest.“ Beim zweiten Einbinden ist es definiert, und der Compiler überspringt alles bis `#endif`.
- **Zeilen 23 und 24:** Die Bibliothek braucht `Arduino.h` und `Wire.h` (Kapitel 22).
- **Zeilen 29 bis 32:** Ein `enum` ist eine Liste benannter Werte. Daher kommt `STK8BA58::PICOBOY_COLOR_PLUS`, das du an `setOrientation` übergibst.
- **Zeilen 34 und 35:** Konstanten, die zur Klasse gehören. `static` heißt hier: Es gibt sie einmal für die Klasse, nicht in jedem Objekt neu.
- **Zeile 37:** Der Konstruktor hat **Standardwerte**: `TwoWire &wire = Wire` und `address = DEFAULT_ADDRESS`. Schreibst du nur `STK8BA58 sensor;`, gelten beide. Wer den Sensor an der zweiten I2C-Schnittstelle hat, schreibt `STK8BA58 sensor(Wire1);`.

- **Zeilen 42 bis 45:** `begin(int sdaPin, int sclPin)` gibt es nur, wenn `ARDUINO_ARCH_RP2040` definiert ist. Der Kern von Earle Philhower definiert das für RP2040 und RP2350. Auf anderen Boards gibt es diese Fassung von `begin` nicht, weil dort `setSDA` / `setSCL` fehlen. So übersetzt die Bibliothek trotzdem überall.
- **Zeilen 41 und 44:** Zwei Methoden mit demselben Namen und verschiedenen Parametern: Das heißt **Überladen**. Der Compiler wählt anhand der Argumente.
- **Zeilen 57 bis 61:** Die Messwerte kommen über Referenzen (`float &x`) zurück, der Rückgabewert `bool` meldet, ob das Lesen geklappt hat.
- **Zeilen 72 bis 78:** Alles unter `private:` ist nur für die Klasse selbst. `readRegisters` ist ein Hilfsmittel, das Benutzer nicht brauchen.

Und ein Stück aus `STK8BA58.cpp`:

STK8BA58.cpp

```cpp
#include "STK8BA58.h"

// Register (Datenblatt STK8BA58)
static const uint8_t REG_CHIP_ID  = 0x00;
static const uint8_t REG_XOUT1    = 0x02;   // danach XOUT2, YOUT1, YOUT2, ZOUT1, ZOUT2
static const uint8_t REG_RANGESEL = 0x0F;
static const uint8_t REG_BWSEL    = 0x10;
static const uint8_t REG_POWMODE  = 0x11;
static const uint8_t REG_SWRST    = 0x14;

static const uint8_t RANGE_2G = 0x03;
static const uint8_t RANGE_4G = 0x05;
static const uint8_t RANGE_8G = 0x08;
static const uint8_t BW_125HZ = 0x0C;
static const uint8_t SOFT_RESET = 0xB6;

static const float G_TO_MS2 = 9.80665f;

STK8BA58::STK8BA58(TwoWire &wire, uint8_t address) : _wire(wire), _address(address) {}

#if defined(ARDUINO_ARCH_RP2040)
bool STK8BA58::begin(int sdaPin, int sclPin) {
    _wire.setSDA(sdaPin);
    _wire.setSCL(sclPin);
    return begin();
}
#endif

bool STK8BA58::begin() {
    _wire.begin();
    if (readRegister(REG_CHIP_ID) != CHIP_ID) {
        return false;
    }
    writeRegister(REG_SWRST, SOFT_RESET);
```

```
35      delay(5);
36      writeRegister(REG_POWMODE, 0x00);   // Normalbetrieb
37      writeRegister(REG_BWSEL, BW_125HZ);
38      return setRange(2);
39  }
```

- **Zeilen 3 bis 15:** Die Registernummern des Chips als Konstanten mit Namen. `static` vor einer Variablen **außerhalb** einer Klasse heißt etwas anderes: Sie ist nur in dieser `.cpp`-Datei sichtbar. So stören die Namen keine anderen Bibliotheken.
- **Zeile 19:** Der Konstruktor mit Initialisierungsliste, wie bei `Balken`. Er tut sonst nichts.
- **Zeilen 29 bis 39:** `begin()` startet I2C, prüft die Chip-ID und stellt den Sensor ein. Was die Register bedeuten, erfährst du in Kapitel 22.

Warum `begin()`? Fast jede Bibliothek trennt das Anlegen des Objekts (Konstruktor) vom Einschalten (`begin`). Der Konstruktor eines globalen Objekts läuft nämlich, bevor `setup()` beginnt, sogar bevor der Kern alles vorbereitet hat. Auf vielen Boards ist die Hardware in diesem Moment noch nicht bereit. `begin()` rufst du dagegen in `setup()` auf, wenn alles läuft. Außerdem kann `begin()` melden, ob es geklappt hat.

Die Bibliothek hat sogar zwei Methoden für eigene Experimente: `readRegister` und `writeRegister`. Damit schaust du direkt in den Chip:

k20_sensor_register.ino

```
 6  void setup() {
 7      Serial.begin(115200);
 8      while (!Serial) {
 9          delay(10);
10      }
11      if (!sensor.begin(20, 21)) {
12          Serial.println("Sensor antwortet nicht");
13      }
14      sensor.setOrientation(STK8BA58::PICOBOY_COLOR_PLUS);
15
16      Serial.printf("Register 0x00 (Chip-ID):     0x%02X\n",
          sensor.readRegister(0x00));
17      Serial.printf("Register 0x0F (Messbereich): 0x%02X\n",
          sensor.readRegister(0x0F));
18      Serial.printf("Register 0x10 (Bandbreite):  0x%02X\n",
          sensor.readRegister(0x10));
19
20      int16_t x, y, z;
21      sensor.readRaw(x, y, z);
22      Serial.printf("Rohwerte: x %d  y %d  z %d\n", x, y, z);
23      Serial.printf("Messbereich: +-%d g\n", sensor.range());
24  }
```

Serieller Monitor

```
Register 0x00 (Chip-ID):     0x87
Register 0x0F (Messbereich): 0x03
Register 0x10 (Bandbreite):  0x0C
Rohwerte: x 0  y 0  z 1024
Messbereich: +-2 g
```

`%02X` gibt eine Zahl hexadezimal mit mindestens zwei Stellen aus. Das Register `0x00` enthält `0x87`, die Kennung dieses Chips; `begin()` hat gerade genau das geprüft. Was die anderen Werte bedeuten, steht in Kapitel 22. Flach auf dem Tisch misst die z-Achse etwa 1024 Stufen, das ist 1 g.

20.5 Eine eigene Bibliothek anlegen

Aus `Balken.h` und `Balken.cpp` wird jetzt eine Bibliothek, die in jedem Sketch bereitsteht. Lege dazu im Ordner `libraries` deines Sketchbooks (**Datei → Einstellungen… → Pfad für Sketchbook**) diese Ordner und Dateien an:

```
libraries/
    Balken/
        library.properties
        keywords.txt
        src/
            Balken.h
            Balken.cpp
        examples/
            Balken_Demo/Balken_Demo.ino
```

`Balken.h` und `Balken.cpp` sind genau die Dateien aus Abschnitt 20.3. Die Bibliothek liegt auch fertig bei den Sketchen dieses Buchs, im Ordner `k20_bibliothek/Balken`; du kannst den Ordner `Balken` einfach nach `libraries` kopieren.

`library.properties` ist der Steckbrief, ein Eintrag pro Zeile:

library.properties

```
name=Balken
version=1.0.0
author=Dein Name
maintainer=Dein Name <dein.name@example.org>
sentence=Fortschrittsbalken fuer Displays mit Adafruit GFX.
paragraph=Zeichnet einen Balken mit Rahmen, der 0 bis 100 Prozent anzeigt. Geht mit
  jedem Display und jedem Canvas von Adafruit GFX.
category=Display
url=https://example.org/balken
architectures=*
depends=Adafruit GFX Library
includes=Balken.h
```

Eintrag	Bedeutung
`name`	Name, wie ihn der Bibliotheksverwalter zeigt
`version`	Versionsnummer, drei Zahlen mit Punkten
`author`, `maintainer`	wer sie geschrieben hat, wer sie pflegt
`sentence`, `paragraph`	kurze und längere Beschreibung
`category`	eine der festen Kategorien, zum Beispiel `Display`, `Sensors`, `Communication`, `Other`
`url`	Webseite der Bibliothek
`architectures`	für welche Boards; `*` heißt „für alle“
`depends`	Bibliotheken, die zusätzlich installiert sein müssen
`includes`	welche `.h`-Datei **Sketch → Bibliothek einbinden** einfügt

Vergleiche mit dem Steckbrief von STK8BA58:

library.properties

```
name=STK8BA58
version=1.0.0
author=PicoBoy Color Plus
maintainer=PicoBoy Color Plus
sentence=Driver for the Sensortek STK8BA58 3-axis accelerometer (I2C).
paragraph=Reads acceleration in g, m/s2 or raw counts; ranges 2/4/8 g; new-data flag.
  Includes an axis preset for the PicoBoy Color Plus. Uses only the Wire library.
category=Sensors
url=https://pbcstudio.de
architectures=*
includes=STK8BA58.h
depends=
```

`keywords.txt` sagt dem Editor, welche Wörter er einfärben soll. Jede Zeile enthält ein Wort, einen **Tabulator** (keine Leerzeichen!) und die Art:

```
Balken    KEYWORD1
zeichne   KEYWORD2

```

`KEYWORD1` ist für Klassen und Typen, `KEYWORD2` für Methoden und Funktionen, `LITERAL1` für Konstanten. Die Datei ist freiwillig; ohne sie funktioniert die Bibliothek genauso, nur ohne Farben.

Das Beispiel in `examples/Balken_Demo/Balken_Demo.ino` ist der Sketch aus Abschnitt 20.3, mit einem Unterschied in der Zeile mit `#include`:

```
#include <Balken.h>
```

Spitze Klammern: Jetzt sucht der Compiler bei den Bibliotheken, nicht im Sketch-Ordner.

Ausprobieren: Starte die Arduino IDE neu, damit sie den neuen Ordner findet. Unter **Datei → Beispiele** erscheint dann `Balken` mit `Balken_Demo`. Öffne es und lade es hoch.

> **Weitergeben als ZIP.** Packe den Ordner `Balken` in eine ZIP-Datei (`Balken.zip`, darin der Ordner `Balken`). Wer sie bekommt, installiert sie wie STK8BA58 in Kapitel 2 mit **Sketch → Bibliothek einbinden → .ZIP-Bibliothek hinzufügen…**. Damit eine Bibliothek im Bibliotheksverwalter für alle auftaucht, muss sie auf GitHub liegen und bei Arduino angemeldet werden; das beschreibt die Arduino-Dokumentation zur „Library Manager"-Aufnahme.

> Halte von einer Bibliothek nur **eine** Kopie im Ordner `libraries`. Liegt dieselbe `.h`-Datei in zwei Bibliotheksordnern (etwa `Balken` und eine alte Kopie `Balken_alt`), muss die IDE eine davon auswählen. Im Ausgabefenster steht dann ein Hinweis, dass mehrere Bibliotheken gefunden wurden, und welche benutzt wird. Änderungen an der anderen Kopie bewirken nichts.

20.6 Gute Gewohnheiten für Bibliotheken

- **Ein eindeutiger Name.** Gibt es zwei Bibliotheken mit einer Datei `Balken.h`, muss die IDE raten. Ein Vorname oder ein Kürzel im Namen hilft (`MiaBalken`).
- **Keine Pins und Einstellungen fest einbauen.** STK8BA58 bekommt die Pins über `begin(20, 21)` und die Lage über `setOrientation`. Deshalb läuft sie auch auf anderen Boards.
- **Ein Kommentar oben in der `.h`-Datei**, der in drei Zeilen zeigt, wie man die Bibliothek benutzt.
- **Mindestens ein Beispiel.** Damit sieht jeder sofort, ob die Bibliothek bei ihm läuft.
- **Die Versionsnummer erhöhen**, wenn du etwas änderst, und Rückgabewerte wie `bool` nutzen, um Fehler zu melden, statt still nichts zu tun.

Übungsaufgaben

> **Aufgabe 1 ⭐ Wörter einfärben** Sieh dir `keywords.txt` von STK8BA58 an (im Ordner der Bibliothek). Welche Wörter erscheinen im Editor in der Farbe für Klassen, welche in der Farbe für Methoden, welche als Konstanten? Welche öffentlichen Namen aus `STK8BA58.h` stehen nicht darin?

> **Aufgabe 2 ⭐⭐ Warnfarbe** Erweitere `Balken.h` aus Abschnitt 20.2: Unter 25 % soll der Balken rot gezeichnet werden, egal welche Farbe er sonst hat. Teste mit zwei Balken bei 60 % und 20 %.

Aufgabe 3 ⭐⭐ Eine Klasse für Tasten Schreibe eine Klasse `Taste` in einer Datei `Taste.h`. Der Konstruktor bekommt den Pin. `beginne()` stellt den Pin auf `INPUT_PULLUP`. `gedrueckt()` liefert `true`, solange die Taste unten ist, `neu_gedrueckt()` genau einmal pro Druck (Kapitel 9), mit Entprellen. Teste mit A und B und zähle die Drücke im seriellen Monitor. Warum gehört `pinMode` nicht in den Konstruktor?

Aufgabe 4 ⭐⭐⭐ Deine PicoBoy-Bibliothek In jedem Sketch stehen dieselben `#define`-Zeilen für die Pins. Schreibe eine Datei `PicoBoy.h` mit allen Pins des PBC+ und einer Funktion `picoboy_start(tft)`, die das Display einschaltet und alle Tasten und LEDs einrichtet. Teste sie als Tab in einem Sketch. Mache danach eine Bibliothek `PicoBoy` daraus (mit `library.properties`). Worauf musst du bei der Funktion achten, damit sie in einer Bibliothek keine Fehler macht?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Eine Bibliothek ist ein Ordner in `libraries` des Sketchbooks mit `library.properties`, `src/` (`.h` und `.cpp`), `examples/` und `keywords.txt`.
- Die `.h`-Datei **deklariert** (was), die `.cpp`-Datei **definiert** (wie). In der `.cpp` steht `Klasse::methode`.
- `#pragma once` oder `#ifndef … #define … #endif` verhindern doppeltes Einlesen. `.h`- und `.cpp`-Dateien binden selbst ein, was sie brauchen.
- Tabs im Sketch: `#include "Datei.h"`. Bibliotheken: `#include <Datei.h>`.
- Bibliotheken haben ein `begin()`, weil Konstruktoren globaler Objekte zu früh laufen. Pins und Einstellungen kommen als Parameter, nicht fest eingebaut.
- Weitergeben: Ordner als ZIP, installieren mit **Sketch → Bibliothek einbinden → .ZIP-Bibliothek hinzufügen…**.

KAPITEL 21

PWM und ADC: Licht dimmen, Batterie messen

Worum geht's? Ein Pin kennt eigentlich nur an und aus. Trotzdem kann der PicoBoy das Hintergrundlicht stufenlos dimmen und eine Spannung messen. Du lernst, was dabei im Chip passiert: Für das Dimmen sorgt die **Pulsweitenmodulation** (PWM), fürs Messen der **Analog-Digital-Wandler** (ADC). Am Ende baust du eine Batterieanzeige.

21.1 Das Hintergrundlicht dimmen

In Kapitel 10 hast du LEDs mit `analogWrite` gedimmt: Der Chip schaltet den Pin 1000-mal pro Sekunde ein und aus, und das Verhältnis von „an" zu „aus" bestimmt die Helligkeit. Das Hintergrundlicht des Displays hängt an GP26 (`TFT_LICHT`). Bisher stand in jedem Sketch `digitalWrite(TFT_LICHT, HIGH)`: Licht voll an. Mit `analogWrite` geht es dunkler:

k21_licht_stufen.ino

```
13 const int STUFEN[8] = { 2, 5, 10, 20, 40, 80, 150, 255 };
14 int stufe = 7;
15 bool hoch_vorher = false;
16 bool runter_vorher = false;
17
18 void setup() {
19     Serial.begin(115200);
20     tft.init(240, 280);
21     pinMode(TASTE_HOCH, INPUT_PULLUP);
22     pinMode(TASTE_RUNTER, INPUT_PULLUP);
23     analogWrite(TFT_LICHT, STUFEN[stufe]);     // statt digitalWrite(TFT_LICHT,
         HIGH)
24     zeichne();
25 }
26
27 void loop() {
28     bool hoch = digitalRead(TASTE_HOCH) == LOW;
29     bool runter = digitalRead(TASTE_RUNTER) == LOW;
30     int vorher = stufe;
31     if (hoch && !hoch_vorher && stufe < 7) {
32         stufe++;
33     }
34     if (runter && !runter_vorher && stufe > 0) {
35         stufe--;
36     }
37     hoch_vorher = hoch;
38     runter_vorher = runter;
```

```
40      if (stufe != vorher) {
41          analogWrite(TFT_LICHT, STUFEN[stufe]);
42          Serial.printf("Stufe %d: analogWrite(TFT_LICHT, %d)\n", stufe,
              STUFEN[stufe]);
43          zeichne();
44      }
45      delay(20);
46  }
```

Serieller Monitor

```
Stufe 6: analogWrite(TFT_LICHT, 150)
Stufe 5: analogWrite(TFT_LICHT, 80)
Stufe 4: analogWrite(TFT_LICHT, 40)
Stufe 5: analogWrite(TFT_LICHT, 80)
```

- **Zeile 23** ersetzt die beiden Zeilen `pinMode` und `digitalWrite`. `analogWrite` braucht kein `pinMode`; es stellt den Pin selbst auf PWM um.
- **Zeile 13:** Die Stufen sind nicht gleichmäßig verteilt, sondern verdoppeln sich ungefähr. Das Auge nimmt Helligkeit nicht gleichmäßig wahr: Der Schritt von 2 auf 5 ist deutlich zu sehen, der von 150 auf 255 kaum. Mit doppelten Werten wirken die Stufen ungefähr gleich groß.
- Mit `analogWrite(TFT_LICHT, 0)` ist das Licht ganz aus, das Bild im Displaychip bleibt aber erhalten. Das spart Strom, etwa in einer Pause.

> Je dunkler das Licht, desto weniger Strom fließt durch seine LEDs. Ein gedimmtes Display hält also länger mit einem Satz Batterien. Aufgabe 4 baut daraus einen Energiesparmodus.

21.2 Was `analogWrite` im Kern tut

Wie in Kapitel 10 lohnt der Blick in den Quelltext des Kerns, in die Datei `cores/rp2040/wiring_analog.cpp`. Gekürzt steht dort in `analogWrite`:

```
pwm_config c = pwm_get_default_config();
pwm_config_set_clkdiv(&c, clock_get_hz(clk_sys) / ((float)analogScale * analogFreq));
pwm_config_set_wrap(&c, analogScale - 1);
pwm_init(pwm_gpio_to_slice_num(pin), &c, true);
...
gpio_set_function(pin, GPIO_FUNC_PWM);
pwm_set_gpio_level(pin, val);
```

Das sind Funktionen aus dem **Pico SDK**, der Programmbibliothek von Raspberry Pi, auf der der Arduino-Kern aufbaut. Übersetzt heißt das:

1. Der RP2350 hat eigene **PWM-Bausteine** (englisch *slices*). Jeder zählt im Takt immer wieder von 0 bis zu einer Obergrenze (*wrap*) und fängt dann von vorn an. `pwm_gpio_to_slice_num(pin)` sucht den Baustein, der zu diesem Pin gehört.
2. Der Prozessortakt (150 MHz) wird so heruntergeteilt (*clkdiv*), dass der Zähler genau `analogFreq`-mal pro Sekunde (Vorgabe 1000) einmal durchläuft.

3. `gpio_set_function(pin, GPIO_FUNC_PWM)` verbindet den Pin mit dem Baustein.
4. `pwm_set_gpio_level(pin, val)`: Solange der Zähler unter `val` ist, ist der Pin an, danach aus.

Der Prozessor selbst hat danach **nichts mehr zu tun**. Der PWM-Baustein schaltet den Pin ganz allein, auch während dein Sketch rechnet oder in `delay` wartet.

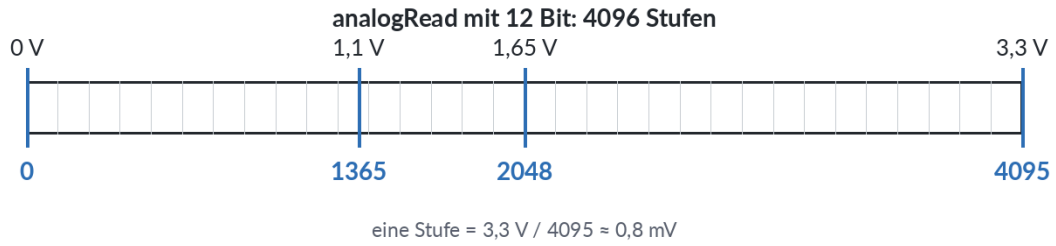
Daraus folgen ein paar Regeln:

- **Einstellungen gelten für alle Pins.** `analogWriteFreq`, `analogWriteRange` und `analogWriteResolution` ändern die gemeinsamen Variablen `analogFreq` und `analogScale`. Danach richtet der nächste `analogWrite`-Aufruf seinen Baustein neu ein. Stellst du für das Licht `analogWriteRange(100)` ein, gilt der Bereich auch für die LEDs.
- **`digitalWrite` beendet PWM.** `digitalWrite` stellt den Pin mit `gpio_set_function(…, GPIO_FUNC_SIO)` wieder auf „gewöhnlicher Pin“ um (`cores/rp2040/wiring_digital.cpp`). Nach `digitalWrite(TFT_LICHT, HIGH)` ist das Licht also wieder voll an, egal was vorher eingestellt war.
- **`tone` stört nicht.** Der Lautsprecher an GP15 wird bei `tone` nicht von einem PWM-Baustein bedient, sondern von einem **PIO**-Baustein (`cores/rp2040/Tone.cpp`), einem kleinen Zusatzprozessor für Pin-Signale.

Die Frequenz. 1000 Hz sind für das Auge schnell genug. Mit `analogWriteFreq(100)` (dem kleinsten erlaubten Wert) schaltet das Licht nur noch 100-mal pro Sekunde. Das Auge sieht es meist noch ruhig, eine Handykamera kann aber Streifen zeigen. Höhere Frequenzen sind möglich; laut Quelltext verkleinert der Kern dann von selbst den Bereich, wenn der Takt nicht mehr für alle Stufen reicht. Für das Licht gibt es keinen Grund, die Vorgabe zu ändern.

21.3 Spannung messen: der ADC

Jetzt die andere Richtung: Statt einen Pin zu schalten, misst der PicoBoy, welche **Spannung** an einem Pin anliegt. Das erledigt der **Analog-Digital-Wandler**, kurz **ADC** (*analog-digital converter*). Er vergleicht die Spannung mit 3,3 V und liefert eine ganze Zahl:



Der ADC des RP2350 hat **12 Bit**, also 2^{12} = 4096 Stufen: 0 heißt 0 V, 4095 heißt 3,3 V. Arduino-Sketche für andere Boards rechnen aber oft mit 10 Bit (0 bis 1023). Deshalb liefert `analogRead` im Kern von Earle Philhower zunächst 10 Bit; erst `analogReadResolution(12)` schaltet auf die vollen 12 Bit um. Im Kern steht dazu `static uint8_t _readBits = 10;` und bei 10 Bit wird der 12-Bit-Messwert einfach um zwei Stellen nach rechts geschoben, also durch 4 geteilt.

Nicht jeder Pin kann messen. Beim RP2350 im PicoBoy sind es GP26 bis GP29, in Arduino `A0` bis `A3`. An GP26 (`A0`) hängt das Hintergrundlicht; **`A3` (GP29)** misst die Batteriespannung.

k21_batterie_roh.ino

```
// Die Batteriespannung als Zahl: erst mit 10, dann mit 12 Bit
#define BATTERIE A3

void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    analogRead(BATTERIE);             // die erste Messung verwerfen
    delay(100);                       // der Eingang braucht einen Moment
    int roh10 = analogRead(BATTERIE);
    Serial.printf("10 Bit: %d  (0 bis 1023)\n", roh10);

    analogReadResolution(12);
    int roh12 = analogRead(BATTERIE);
    Serial.printf("12 Bit: %d  (0 bis 4095)\n", roh12);
}

void loop() {
}
```

Serieller Monitor

```
10 Bit: 310  (0 bis 1023)
12 Bit: 1241  (0 bis 4095)
```

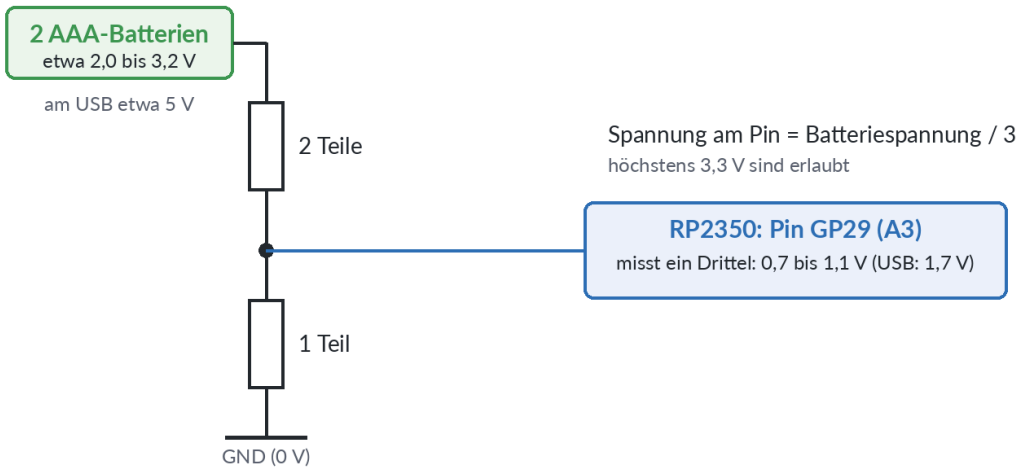
Die allererste Messung verwirft der Sketch: Nach dem ersten Zugriff braucht der Eingang rund 100 ms, bis er die richtige Spannung zeigt.

Beide Zahlen gehören zur selben Spannung: 310 von 1023 und 1241 von 4095 sind beide etwa 30 % von 3,3 V. Mit 12 Bit ist die Messung nur feiner gestuft.

> Diese und die folgenden Ausgaben stammen aus dem Prüflauf auf dem PC. Dort war eine Batteriespannung von 3,0 V eingestellt. Dein PicoBoy misst seine eigene Spannung; die Zahlen springen dann außerdem um ein paar Stufen hin und her, weil jede Messung ein wenig schwankt.

21.4 Vom Messwert zur Batteriespannung

Der PicoBoy läuft mit zwei AAA-Batterien. Sie liegen **in Reihe**, also hintereinander: Ihre Spannungen zählen zusammen. Eine frische Alkali-Batterie hat etwa 1,5 V, beide zusammen also etwa 3 V. Am USB-Kabel misst der PicoBoy an dieser Stelle sogar etwa 5 V. Das ist mehr als die 3,3 V, die ein Pin des RP2350 verträgt. Deshalb hängt GP29 nicht direkt an den Batterien, sondern an einem **Spannungsteiler**: zwei Widerstände, die die Spannung im Verhältnis teilen. Beim PicoBoy kommt ein Drittel der Spannung am Pin an.



Die Rechnung hat also zwei Schritte:

1. Messwert in Volt am Pin: `wert * 3.3 / 4095`.
2. Mal 3, wegen des Teilers: Das ist die Batteriespannung.

k21_batterie_volt.ino

```
1  // Vom Messwert zur Batteriespannung
2  #define BATTERIE A3
3
4  void setup() {
5      Serial.begin(115200);
6      analogReadResolution(12);
7  }
8
9  void loop() {
10     int wert = analogRead(BATTERIE);
11     float am_pin = wert * 3.3 / 4095;      // Spannung am Pin GP29
12     float batterie = am_pin * 3;           // Spannungsteiler 1:3
13     Serial.printf("Messwert %d -> am Pin %.3f V -> Batterie %.2f V\n", wert,
         am_pin, batterie);
14     delay(1000);
15 }
```

Serieller Monitor

```
Messwert 1241 -> am Pin 1.000 V -> Batterie 3.00 V
Messwert 1241 -> am Pin 1.000 V -> Batterie 3.00 V
Messwert 1241 -> am Pin 1.000 V -> Batterie 3.00 V
```

- **Zeile 6:** `analogReadResolution(12)` einmal in `setup()`, dann gilt es für alle folgenden Messungen.
- **Zeile 11:** `wert * 3.3 / 4095` rechnet mit Kommazahlen, weil `3.3` eine ist. Stünde dort `wert * 3 / 4095`, käme in ganzen Zahlen fast immer 0 heraus (Kapitel 5).

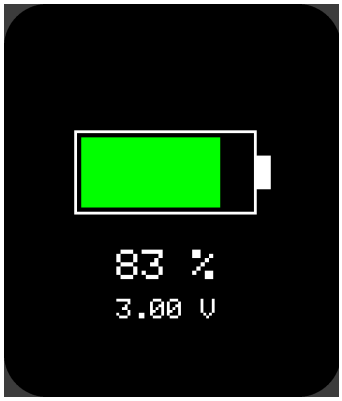
> Ist der PicoBoy per USB angeschlossen, zeigt `A3` nicht die Batteriespannung, sondern etwa 5 V. Wie voll die Batterien sind, siehst du verlässlich nur ohne Kabel.

Ausprobieren: Eine Batterieanzeige

k21_batterieanzeige.ino

```
25 void loop() {
26     float volt = batterie_messen();
27     int prozent = (volt - LEER_VOLT) / (VOLL_VOLT - LEER_VOLT) * 100;
28     prozent = constrain(prozent, 0, 100);
29 
30     uint16_t farbe = ST77XX_GREEN;
31     if (prozent < 50) {
32         farbe = ST77XX_YELLOW;
33     }
34     if (prozent < 20) {
35         farbe = ST77XX_RED;
36     }
37 
38     bild.fillScreen(ST77XX_BLACK);
39     bild.drawRect(50, 90, 130, 60, ST77XX_WHITE);      // Gehaeuse
40     bild.drawRect(51, 91, 128, 58, ST77XX_WHITE);
41     bild.fillRect(180, 108, 10, 24, ST77XX_WHITE);     // Pluspol
42     bild.fillRect(55, 95, 120 * prozent / 100, 50, farbe);
43     bild.setTextColor(ST77XX_WHITE);
44     bild.setTextSize(3);
45     bild.setCursor(80, 175);
46     bild.printf("%d %%", prozent);
47     bild.setTextSize(2);
48     bild.setCursor(80, 210);
49     bild.printf("%.2f V", volt);
50     if (volt > 4.0) {
51         bild.setCursor(60, 240);
52         bild.print("(USB)");
53     }
54     tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
55     Serial.printf("%.2f V, %d %%\n", volt, prozent);
56     delay(1000);
57 }
58 
59 // 16 Messungen mitteln: einzelne Ausreisser fallen kaum ins Gewicht
60 float batterie_messen() {
61     long summe = 0;
62     for (int i = 0; i < 16; i++) {
63         summe = summe + analogRead(BATTERIE);
64     }
```

```
65      float mittel = summe / 16.0;
66      return mittel * 3.3 / 4095 * 3;
67  }
```



- **Zeilen 60 bis 67:** Statt einer Messung nimmt `batterie_messen` den Mittelwert aus 16 Messungen. Einzelne Ausreißer fallen dann kaum ins Gewicht, und die Anzeige zappelt nicht. Die Summe steht in einem `long`; 16 · 4095 passt zwar auch in ein `int`, aber so bleibt es auch bei mehr Messungen sicher.
- **Zeile 27** rechnet die Spannung in Prozent um, mit 2,0 V als „leer“ und 3,2 V als „voll“. Das ist nur eine grobe Schätzung. Zwei frische Alkali-Batterien haben zusammen etwa 3,0 bis 3,2 V, fast leere etwa 2,0 bis 2,2 V. Dazwischen sinkt die Spannung nach und nach.
- Wiederaufladbare AAA-Akkus (NiMH) haben nur etwa 1,2 V pro Stück, zusammen also etwa 2,4 bis 2,7 V. Mit ihnen zeigt die Anzeige schon bei vollen Akkus nur etwa 30 bis 60 %.
- Über 4 V kann die Spannung nicht von zwei AAA-Batterien kommen. Dann hängt der PicoBoy am USB, und es erscheint „(USB)“.

Zum Verändern: Zeige die Anzeige klein in einer Ecke deines Spiels aus Kapitel 18. Miss nur einmal pro Sekunde; die Batteriespannung ändert sich nicht schneller.

Übungsaufgaben

Aufgabe 1 ⭐ **Rechnen wie der ADC** a) Ein Sketch mit 12 Bit misst an `A3` den Wert 900. Wie viel Volt sind am Pin, wie viel haben die Batterien zusammen? b) Welchen Wert liefert dieselbe Spannung ohne `analogReadResolution(12)`? c) Welcher 12-Bit-Wert gehört zu zwei frischen Batterien mit zusammen 3,2 V?

Aufgabe 2 ⭐ **Sanft einschalten** Schreibe einen Sketch, der beim Start einen Text zeichnet, während das Licht noch aus ist, und das Licht dann in etwa einer Sekunde von 0 auf voll hochdreht.

Aufgabe 3 ⭐⭐ **Batterien wechseln!** Miss alle halbe Sekunde die Batteriespannung. Liegt sie unter 2,2 V, soll „Wechseln!“ in Rot auf dem Display stehen und die rote LED blinken. Sonst steht dort „Batterie ok“ in Grün. Zeige die Spannung darunter an.

Aufgabe 4 ⭐⭐ Energiesparmodus Wird 10 Sekunden lang keine der sieben Tasten gedrückt, soll das Licht auf den Wert 10 gedimmt werden. Beim nächsten Tastendruck geht es wieder voll an. Gib im seriellen Monitor aus, wann gedimmt und wann wieder hell geschaltet wird.

Aufgabe 5 ⭐⭐⭐ Gleichmäßige Stufen Statt die Stufen auszuprobieren wie in Abschnitt 21.1, kann man sie ausrechnen. Bildschirme und Kameras machen das mit der **Gammakorrektur**: Stufe i von n wird zu `255 * pow(i / (n - 1.0), 2.2)`. Berechne damit in `setup()` zehn Stufen (die kleinste soll mindestens 1 sein), gib sie im seriellen Monitor aus und benutze sie wie in `k21_licht_stufen`. Wirken die Schritte gleichmäßig?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- `analogWrite(TFT_LICHT, wert)` dimmt das Hintergrundlicht (0 bis 255), ohne `pinMode`. Ein späteres `digitalWrite` auf denselben Pin beendet PWM.
- Ein PWM-Baustein des RP2350 schaltet den Pin selbstständig; Frequenz (Vorgabe 1000 Hz) und Bereich (Vorgabe 255) gelten für alle Pins.
- Das Auge sieht Helligkeit nicht gleichmäßig: Stufen lieber verdoppeln oder mit Gammakorrektur berechnen.
- `analogRead(pin)` misst 0 bis 3,3 V an GP26–GP29 (`A0` – `A3`), zunächst mit 10 Bit (0–1023), nach `analogReadResolution(12)` mit 12 Bit (0–4095).
- Batterien: `volt = analogRead(A3) * 3.3 / 4095 * 3` (12 Bit, Spannungsteiler 1:3). Mehrere Messungen mitteln. Zwei AAA-Batterien: etwa 3 V frisch, etwa 2 V fast leer; am USB etwa 5 V.

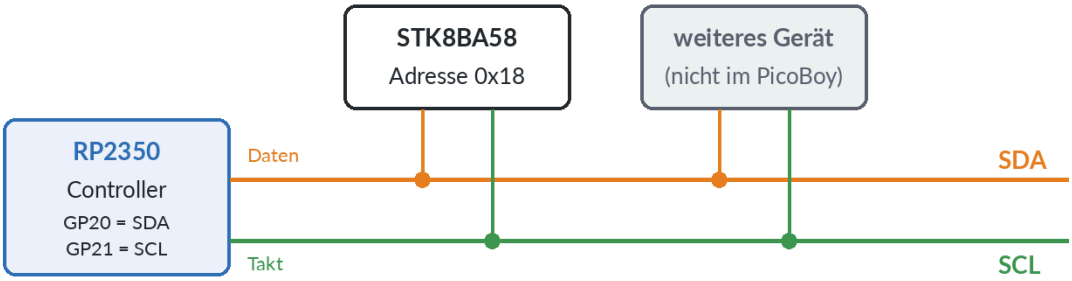
KAPITEL 22

I2C mit `Wire`: der Neigungssensor ohne Bibliothek

Worum geht's? In Kapitel 17 hat die Bibliothek STK8BA58 alles für dich erledigt. Jetzt sprichst du selbst mit dem Sensor. Er hängt an einem **I2C-Bus**, zwei Leitungen, über die der Prozessor mit anderen Chips redet. Du suchst mit einem **I2C-Scanner**, wer am Bus hängt, liest Register, setzt aus zwei Bytes einen 12-Bit-Messwert zusammen und baust die Wasserwaage noch einmal, ganz ohne Bibliothek. Nebenbei erfährst du, warum die Standardeinstellung des Boards hier nicht passt.

22.1 Zwei Leitungen für viele Geräte

I2C (gesprochen „I-Quadrat-C", von *Inter-Integrated Circuit*) ist eine Verbindung zwischen Chips auf derselben Platine. Sie braucht nur zwei Leitungen:



Beide Leitungen haben Pull-up-Widerstände nach 3,3 V:
Ohne Verkehr sind sie HIGH, jedes Gerät kann sie nur auf LOW ziehen.

- **SCL** (*serial clock*) ist der Takt. Ihn gibt immer der **Controller** vor, hier der RP2350.
- **SDA** (*serial data*) trägt die Daten, Bit für Bit, in beide Richtungen.

An denselben zwei Leitungen können viele Geräte hängen. Damit klar ist, wer gemeint ist, hat jedes Gerät eine **Adresse** zwischen 1 und 127 (7 Bit). Der Neigungssensor STK8BA58 hat auf dem PicoBoy die Adresse `0x18`. Jedes Gespräch beginnt damit, dass der Controller eine Adresse auf den Bus legt. Nur das Gerät mit dieser Adresse antwortet; alle anderen hören weg.

Wie schnell das geht, legt der Takt fest. `Wire` benutzt ohne weitere Angabe 100 000 Bit pro Sekunde (im Kern: `TWI_CLOCK = 100000` in `libraries/Wire/src/Wire.h`). Das ist viel langsamer als die SPI-Leitung zum Display, reicht aber für ein paar Messwerte locker.

22.2 `Wire` auf dem PicoBoy einrichten

Die Arduino-Bibliothek für I2C heißt **`Wire`**. Sie gehört zum Kern, du musst nichts installieren. Für den PicoBoy brauchst du drei Zeilen:

```
#include <Wire.h>
...
Wire.setSDA(20);      // Daten an GP20
Wire.setSCL(21);      // Takt an GP21
Wire.begin();
```

Warum `setSDA` und `setSCL`? Das Board „Generic RP2350“ legt `Wire` von sich aus auf **GP4 und GP5**; so steht es im Kern in `variants/generic_rp2350/pins_arduino.h`:

```
#define PIN_WIRE0_SDA  (4u)
#define PIN_WIRE0_SCL  (5u)
```

Das passt für ein nacktes Pico-Board, aber nicht für den PicoBoy: Dort sitzt der Sensor an **GP20/GP21**, und an **GP4 hängt der Joystick (hoch)**. Mit den Standardpins findet `Wire` den Sensor nicht, und ein Druck nach oben zieht die Datenleitung auf LOW. Deshalb steht in diesem Buch immer `sensor.begin(20, 21)` – und das tut, wie du in Kapitel 20 gesehen hast, genau `setSDA(20)`, `setSCL(21)`, `begin()`.

Zwei Regeln aus dem Quelltext des Kerns (`libraries/Wire/src/Wire.cpp`):

- `setSDA` und `setSCL` gehören **vor** `Wire.begin()`. Danach lassen sich die Pins nicht mehr ändern.
- Nicht jeder Pin geht. `Wire` (der erste I2C-Baustein) erlaubt SDA nur an GP0, 4, 8, 12, 16, 20, 24 oder 28 und SCL an den Pins direkt dahinter (1, 5, 9, …, 21, …). GP20/GP21 passt.

Hältst du dich nicht daran, ruft der Kern `panic(…)` auf: Der Sketch bleibt sofort stehen, ohne Fehlermeldung beim Kompilieren. Vertauschte Pins wie `Wire.setSDA(21)` sind dafür ein typischer Grund.

> In Anleitungen aus dem Internet steht oft nur `Wire.begin();`. Auf dem PicoBoy reicht das nicht. Vor jedem `Wire.begin()` gehören `Wire.setSDA(20);` und `Wire.setSCL(21);`.

22.3 Der I2C-Scanner

Wer hängt eigentlich am Bus? Das findet ein **I2C-Scanner** heraus: Er ruft der Reihe nach jede Adresse auf und schaut, ob sich jemand meldet.

k22_scanner.ino

```
1 // I2C-Scanner: Wer antwortet auf dem Bus an GP20/GP21?
2 #include <Wire.h>
3
4 void setup() {
5     Serial.begin(115200);
```

```
 6     while (!Serial) {
 7         delay(10);
 8     }
 9     Wire.setSDA(20);          // vor Wire.begin()!
10     Wire.setSCL(21);
11     Wire.begin();
12
13     int gefunden = 0;
14     for (int adresse = 1; adresse < 127; adresse++) {
15         Wire.beginTransmission(adresse);
16         if (Wire.endTransmission() == 0) {   // 0 = Geraet hat geantwortet
17             Serial.printf("Geraet an Adresse 0x%02X\n", adresse);
18             gefunden++;
19         }
20     }
21     Serial.printf("%d Geraet(e) gefunden\n", gefunden);
22 }
23
24 void loop() {
25 }
```

Serieller Monitor

```
Geraet an Adresse 0x18
1 Geraet(e) gefunden
```

- `Wire.beginTransmission(adresse)` (Zeile 15) beginnt ein Gespräch mit einem Gerät. Noch geht nichts über die Leitung; `Wire` sammelt, was gesendet werden soll.
- `Wire.endTransmission()` (Zeile 16) schickt alles ab. Hier ist es nichts außer der Adresse. Der Rückgabewert sagt, wie es lief: **0** heißt, ein Gerät hat seine Adresse erkannt und mit einem kurzen „Ja“ (englisch *ACK*) geantwortet.

Im Quelltext des Kerns stehen die möglichen Rückgabewerte über `endTransmission`:

Wert	Bedeutung
0	Erfolg
1	zu viele Daten für den Puffer
2	keine Antwort auf die Adresse
3	keine Antwort auf ein Datenbyte
4	anderer Fehler
5	Zeitüberschreitung

Auf dem PicoBoy meldet sich nur der Sensor an `0x18`. Derselbe Scanner **ohne** `setSDA`/`setSCL` (`k22_scanner_standard`) fragt an GP4/GP5 und findet nichts:

Serieller Monitor

```
0 Geraet(e) gefunden
```

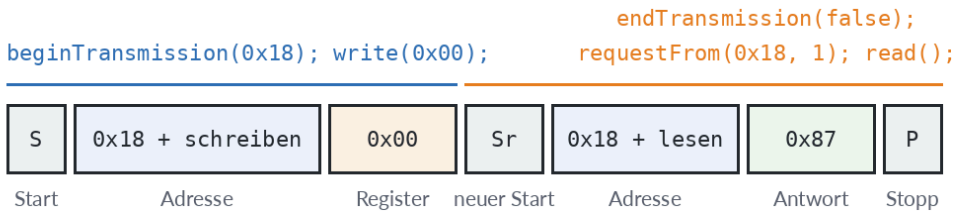
> Der Scanner ist das erste Werkzeug, wenn ein I2C-Gerät „nicht geht“. Findet er nichts, stimmen die Pins nicht oder das Gerät hat keinen Strom. Findet er eine andere Adresse als erwartet, steht im Sketch die falsche.

22.4 Ein Register lesen

Ein Sensorchip hat innen eine Reihe von **Registern**: kleine Speicherplätze mit Nummern, jeder ein Byte groß. Manche enthalten Messwerte, in andere schreibst du Einstellungen. Was in welchem Register steht, verrät das **Datenblatt** des Chips. Für den STK8BA58 brauchst du diese (so stehen sie auch in `STK8BA58.cpp`):

Register	Name	Inhalt
`0x00`	CHIP_ID	Kennung des Chips, immer `0x87`
`0x02` … `0x07`	XOUT1 … ZOUT2	Messwerte: je zwei Bytes für x, y, z
`0x0F`	RANGESEL	Messbereich: `0x03` = ±2 g, `0x05` = ±4 g, `0x08` = ±8 g
`0x10`	BWSEL	Bandbreite; `0x0C` = 125 Hz
`0x11`	POWMODE	`0x00` = Normalbetrieb
`0x14`	SWRST	`0xB6` hineinschreiben = Neustart

Ein Register lesen geht in zwei Schritten: Erst **schreibst** du die Registernummer, dann **liest** du:



Der Sensor antwortet nur, wenn er seine Adresse hört; „Sr“ statt „P, S“ hält den Bus fest.

k22_chip_id.ino

```
#define SENSOR 0x18           // I2C-Adresse des STK8BA58

const uint8_t REG_CHIP_ID = 0x00;   // Registernummer der Chip-ID

void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    Wire.setSDA(20);
    Wire.setSCL(21);
    Wire.begin();

    Wire.beginTransmission(SENSOR);
```

```
18      Wire.write(REG_CHIP_ID);              // welches Register?
19      Wire.endTransmission(false);          // false: kein Stopp, gleich lesen
20      Wire.requestFrom(SENSOR, 1);          // ein Byte anfordern
21      int id = Wire.read();
22      Serial.printf("Chip-ID: 0x%02X\n", id);
23      if (id == 0x87) {
24          Serial.println("Das ist ein STK8BA58.");
25      }
26  }
```

Serieller Monitor

```
Chip-ID: 0x87
Das ist ein STK8BA58.
```

- **Zeilen 17 und 18:** Das Gespräch mit `0x18` beginnt; das erste Byte ist die Registernummer `REG_CHIP_ID`, also `0x00`.
- **Zeile 19:** `endTransmission(false)` schickt ab, beendet das Gespräch aber **nicht** mit einem Stopp. Der Bus bleibt belegt, und gleich folgt ein neuer Start zum Lesen. So kann kein anderer dazwischenfunken.
- **Zeile 20:** `requestFrom(SENSOR, 1)` holt ein Byte vom Sensor. Er liefert den Inhalt des Registers, dessen Nummer er gerade bekommen hat.
- **Zeile 21:** `Wire.read()` nimmt das Byte aus dem Empfangspuffer.

`0x87`: Das ist ein STK8BA58. Genau diese Prüfung macht `sensor.begin()` in der Bibliothek.

22.5 Messwerte lesen

Nach dem Einschalten muss der Sensor eingerichtet werden. Schreiben ist einfacher als Lesen: Adresse, Registernummer, neuer Wert, fertig. Die Messwerte liegen in sechs Registern hintereinander, und die liest man am besten in einem Rutsch:

k22_rohwerte.ino

```
 6  void schreibe_register(uint8_t reg, uint8_t wert) {
 7      Wire.beginTransmission(SENSOR);
 8      Wire.write(reg);
 9      Wire.write(wert);
10      Wire.endTransmission();
11  }
12
13  void setup() {
14      Serial.begin(115200);
15      Wire.setSDA(20);
16      Wire.setSCL(21);
17      Wire.begin();
18      schreibe_register(0x14, 0xB6);    // Neustart (Soft-Reset)
19      delay(5);
20      schreibe_register(0x11, 0x00);    // Normalbetrieb
21      schreibe_register(0x10, 0x0C);    // Bandbreite 125 Hz
```

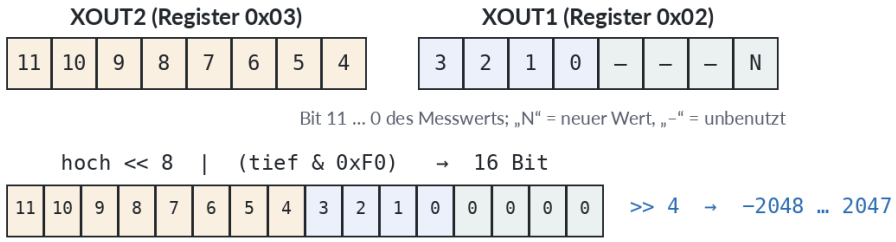
```
22      schreibe_register(0x0F, 0x03);    // Messbereich +-2 g
23  }
24
25  void loop() {
26      uint8_t b[6];
27      Wire.beginTransmission(SENSOR);
28      Wire.write(0x02);                 // ab Register 0x02 ...
29      Wire.endTransmission(false);
30      Wire.requestFrom(SENSOR, 6);      // ... sechs Bytes am Stueck
31      for (int i = 0; i < 6; i++) {
32          b[i] = Wire.read();
33      }
34      Serial.printf("Bytes: %02X %02X  %02X %02X  %02X %02X\n", b[0], b[1], b[2],
          b[3], b[4], b[5]);
35
36      // 12 Bit: OUT2 enthaelt Bit 11..4, das obere Halbbyte von OUT1 Bit 3..0
37      int16_t x = (int16_t)(b[1] << 8 | (b[0] & 0xF0)) >> 4;
38      int16_t y = (int16_t)(b[3] << 8 | (b[2] & 0xF0)) >> 4;
39      int16_t z = (int16_t)(b[5] << 8 | (b[4] & 0xF0)) >> 4;
40      Serial.printf("Chip-Achsen: x %5d  y %5d  z %5d\n", x, y, z);
41      delay(1000);
42  }
```

Serieller Monitor

```
Bytes: 01 00  01 00  01 C0
Chip-Achsen: x     0  y     0  z -1024
Bytes: 01 00  01 00  01 C0
Chip-Achsen: x     0  y     0  z -1024
Bytes: 01 00  01 20  51 C8
Chip-Achsen: x     0  y   512  z  -891
```

- **Zeilen 6 bis 11:** `schreibe_register` schickt zwei Bytes: Nummer und Wert. Die Reihenfolge in `setup()` ist die aus der Bibliothek: Neustart, Normalbetrieb, Bandbreite, Messbereich ±2 g.
- **Zeile 30:** `requestFrom(SENSOR, 6)` liest sechs Bytes. Der Sensor zählt nach jedem Byte die Registernummer weiter: `0x02`, `0x03`, … `0x07`.

Zwei Bytes, ein Wert. Jede Achse liefert einen Wert mit **12 Bit** (−2048 bis 2047), verteilt auf zwei Register:



Weil Bit 11 ganz links landet, wird der Wert als int16_t richtig negativ; das Schieben um 4 nach rechts behält das Vorzeichen.

Zeile 37 setzt ihn mit **Bit-Operationen** zusammen. Die arbeiten auf den einzelnen Bits einer Zahl:

Operation	bedeutet	Beispiel
`a << 8`	alle Bits 8 Stellen nach links schieben (mal 256)	`0x40 << 8` = `0x4000`
`a >> 4`	alle Bits 4 Stellen nach rechts schieben (durch 16)	`0x4000 >> 4` = `0x0400`
`a & 0xF0`	nur die Bits behalten, die in `0xF0` gesetzt sind	`0x31 & 0xF0` = `0x30`
`a | b`	Bits zusammenfügen: gesetzt, wenn in `a` oder `b` gesetzt	`0x4000 | 0x30` = `0x4030`

Schritt für Schritt für die y-Achse, wenn der PicoBoy um 30° nach rechts gekippt ist (letzter Block der Ausgabe, Bytes `01 20`): `0x20 << 8` ergibt `0x2000`, `0x01 & 0xF0` ergibt `0x00` (das Bit „neuer Wert“ fällt weg), zusammen `0x2000`. Als `int16_t` gelesen und um 4 nach rechts geschoben: `0x0200` = 512. Bei ±2 g sind 1024 Stufen genau 1 g, 512 also 0,5 g.

Warum dieser Umweg über die linke Kante? Ein negativer Wert hat Bit 11 gesetzt. Landet Bit 11 ganz links in einem `int16_t`, ist die ganze Zahl negativ, und das Schieben nach rechts behält das Vorzeichen. So kommen aus den Bytes `01 C0` die −1024 heraus, die der Chip flach auf dem Tisch auf seiner z-Achse misst: `0xC000` ist als `int16_t` negativ, um 4 geschoben −1024, also −1 g. Die z-Achse des Chips zeigt nach unten, in den PicoBoy hinein.

Die Achsen des PicoBoy. Der Chip liegt gedreht auf der Platine. Seine x-Achse zeigt dorthin, wo beim PicoBoy y ist, und umgekehrt, und zwei Achsen zählen in die andere Richtung. Das rechnet `lese_achsen` um und teilt gleich durch 1024:

k22_beschleunigung.ino

```
14 // liest die drei Achsen; x, y, z sind Referenzen auf Variablen des Aufrufers
15 void lese_achsen(float &x, float &y, float &z) {
16     uint8_t b[6];
17     Wire.beginTransmission(SENSOR);
18     Wire.write(0x02);
19     Wire.endTransmission(false);
20     Wire.requestFrom(SENSOR, 6);
21     for (int i = 0; i < 6; i++) {
22         b[i] = Wire.read();
23     }
24     int16_t cx = (int16_t)(b[1] << 8 | (b[0] & 0xF0)) >> 4;
25     int16_t cy = (int16_t)(b[3] << 8 | (b[2] & 0xF0)) >> 4;
26     int16_t cz = (int16_t)(b[5] << 8 | (b[4] & 0xF0)) >> 4;
27     // Der Chip liegt gedreht auf der Platine
28     x = cy / STUFEN_PRO_G;
29     y = -cx / STUFEN_PRO_G;
30     z = -cz / STUFEN_PRO_G;
31 }
```

Serieller Monitor

```
x  0.00 g   y  0.00 g   z  1.00 g
x  0.00 g   y  0.00 g   z  1.00 g
x  0.50 g   y  0.00 g   z  0.87 g
x  0.50 g   y  0.00 g   z  0.87 g
x  0.00 g   y -0.50 g   z  0.87 g
x  0.00 g   y -0.50 g   z  0.87 g
```

`lese_achsen` gibt drei Werte zurück, über Referenzen (Kapitel 19.2): `float &x` heißt, `x` ist die Variable des Aufrufers selbst. Genau so sieht `readAcceleration` in der Bibliothek aus. Im Prüflauf wurde der PicoBoy nach einer Sekunde nach rechts gekippt und nach zwei Sekunden so, dass die Seite mit dem Joystick nach oben zeigt.

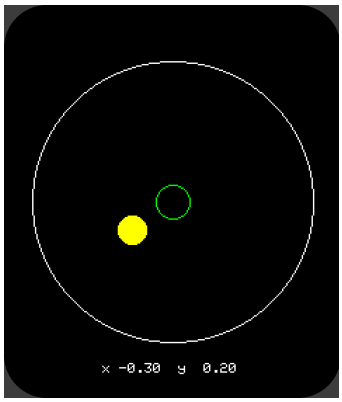
> Die Ausgaben dieses Kapitels stammen aus dem Prüflauf auf dem PC. Dort antwortet am I2C-Bus eine Nachbildung des Sensors an `0x18` mit vorgegebenen Neigungen. Am PicoBoy schwanken die Werte in den letzten Stellen, weil der Sensor auch kleinste Erschütterungen misst.

Ausprobieren: Die Wasserwaage ohne Bibliothek

k22_wasserwaage.ino

```
21 int16_t achse(uint8_t *b) {              // b zeigt auf OUT1, b[1] ist OUT2
22     return (int16_t)(b[1] << 8 | (b[0] & 0xF0)) >> 4;
23 }
24
25 void setup() {
26     tft.init(240, 280);
27     tft.setSPISpeed(62500000);
28     pinMode(TFT_LICHT, OUTPUT);
29     digitalWrite(TFT_LICHT, HIGH);
30     Wire.setSDA(20);
31     Wire.setSCL(21);
32     Wire.begin();
33     schreibe_register(0x14, 0xB6);
34     delay(5);
35     schreibe_register(0x11, 0x00);
36     schreibe_register(0x10, 0x0C);
37     schreibe_register(0x0F, 0x03);
38 }
39
40 void loop() {
41     uint8_t b[6];
42     Wire.beginTransmission(SENSOR);
43     Wire.write(0x02);
44     Wire.endTransmission(false);
45     Wire.requestFrom(SENSOR, 6);
46     for (int i = 0; i < 6; i++) {
47         b[i] = Wire.read();
48     }
```

```
49      float x = -achse(b + 2) / 1024.0;     // PicoBoy-x = -Chip-y
50      float y = achse(b) / 1024.0;          // PicoBoy-y =  Chip-x
51
52      int bx = 120 + (int)(x * 100);
53      int by = 140 + (int)(y * 100);
54      bild.fillScreen(ST77XX_BLACK);
55      bild.drawCircle(120, 140, 100, ST77XX_WHITE);
56      bild.drawCircle(120, 140, 12, ST77XX_GREEN);
57      bild.fillCircle(bx, by, 10, ST77XX_YELLOW);
58      bild.setTextColor(ST77XX_WHITE);
59      bild.setCursor(70, 255);
60      bild.printf("x %5.2f  y %5.2f", x, y);
61      tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
62      delay(20);
63  }
```



Das ist das Beispiel `Wasserwaage_PicoBoy` der Bibliothek STK8BA58, nur mit `Wire` statt `sensor`:

- `achse(b)` setzt eine Achse zusammen. Sie bekommt einen **Zeiger** auf das erste ihrer zwei Bytes (Kapitel 19.3). `achse(b + 2)` ist deshalb die y-Achse des Chips.
- **Zeilen 49 und 50** rechnen auf die Achsen des PicoBoy um.
- **Zeilen 52 und 53:** Die Blase wandert zur höchsten Stelle; je stärker gekippt, desto weiter weg von der Mitte.

Zum Verändern: Stelle den Messbereich auf ±8 g (`0x08`). Wie musst du die Umrechnung ändern? Warum wird die Blase dann „zittriger“?

Übungsaufgaben

Aufgabe 1 ⭐ **Hex und binär** Schreibe `0x87`, `0xB6` und `0x18` als Binärzahlen (8 Stellen). Was ist `0xB6 & 0xF0`, was `0x05 << 4`? Welche Zahl kommt aus den Bytes `F0 FF` (erst OUT1, dann OUT2) heraus?

Aufgabe 2 ⭐ **Scanner als Tabelle** Baue den Scanner so um, dass er eine Tabelle mit 8 Zeilen und 16 Spalten ausgibt: Zeile = erste Hex-Ziffer der Adresse, Spalte = zweite. Für jede Adresse steht `--` oder, wenn sich jemand meldet, die Adresse selbst.

Aufgabe 3 ⭐⭐ **Messbereich ±4 g** Richte den Sensor mit ±4 g ein. Schreibe eine Funktion `uint8_t lese_register(uint8_t reg)` und prüfe damit, dass Register `0x0F` wirklich `0x05` enthält. Lies dann z (Register `0x06` und `0x07`) und rechne in g um. Wie viele Stufen sind jetzt 1 g?

Aufgabe 4 ⭐⭐ Vorher und nachher Gib die Register `0x00`, `0x0F`, `0x10` und `0x11` einmal direkt nach dem Neustart (`0x14` ← `0xB6`) und einmal nach dem Einrichten aus. Was hat sich geändert?

Aufgabe 5 ⭐⭐⭐ Schüttelzähler In Ruhe misst der Sensor insgesamt 1 g, egal wie der PicoBoy liegt: Die **Stärke** `sqrt(x * x + y * y + z * z)` ist 1. Beim Schütteln wird sie größer. Zähle, wie oft geschüttelt wird (Stärke über 1,6 g), zeige die Zahl groß auf dem Display und spiele bei jedem Schüttler einen kurzen Ton. Ein Schüttler soll höchstens einmal pro halbe Sekunde zählen.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- **I2C**: zwei Leitungen, SDA (Daten) und SCL (Takt); jedes Gerät hat eine Adresse. Der Sensor des PicoBoy: `0x18` an GP20/GP21.
- Auf dem PicoBoy immer `Wire.setSDA(20); Wire.setSCL(21);` **vor** `Wire.begin()`. Die Standardpins GP4/GP5 sind falsch: GP4 ist Joystick-hoch.
- Schreiben: `beginTransmission(adr)`, `write(reg)`, `write(wert)`, `endTransmission()`. Rückgabe 0 = Erfolg.
- Lesen: `beginTransmission(adr)`, `write(reg)`, `endTransmission(false)`, `requestFrom(adr, n)`, dann `n`-mal `read()`.
- Scanner: Wer bei `endTransmission()` 0 liefert, ist da.
- STK8BA58: Chip-ID `0x87` in Register `0x00`, Messwerte in `0x02` – `0x07`, 12 Bit linksbündig: `(int16_t)(hoch << 8 | (tief & 0xF0)) >> 4`; 1024 Stufen = 1 g bei ±2 g.

KAPITEL 23

SPI und das Display

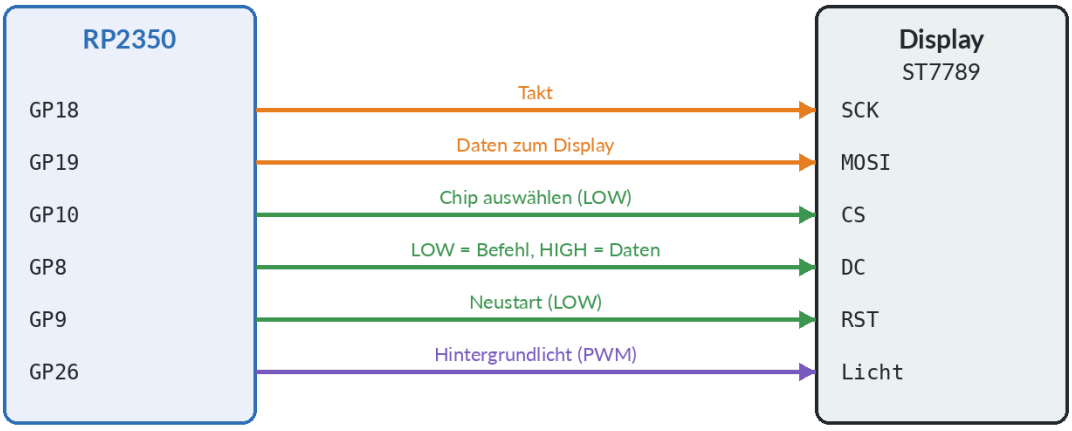
Worum geht's? Hinter `tft.fillScreen` steckt eine Menge: Leitungen, Befehle, ein Bildspeicher im Displaychip. Du schaust in den Quelltext der Bibliothek Adafruit ST7789 und siehst, was `tft.init(240, 280)` wirklich tut, warum auf dem PicoBoy 20 Zeilen „fehlen" und wie ein Pixel auf den Bildschirm kommt. Zum Schluss steuerst du das Display ganz ohne Display-Bibliothek, nur mit `SPI`.

23.1 SPI: schnell, aber nur in eine Richtung

Das Display hängt nicht am I2C-Bus aus Kapitel 22, sondern an einer schnelleren Verbindung, **SPI** (*Serial Peripheral Interface*). Der Unterschied:

	I2C (Sensor)	SPI (Display)
Leitungen	2: SDA, SCL	Takt, Daten hin, Daten zurück, dazu je Gerät ein CS
Wer ist gemeint?	Adresse im ersten Byte	eigene Leitung **CS** (*chip select*) je Gerät
Antwort	jedes Byte wird bestätigt	keine Bestätigung
Takt auf dem PicoBoy	100 kHz	25 MHz (mit `setSPISpeed` 37,5 MHz)

SPI ist so schnell, weil es einfach ist: Bei jedem Takt geht ein Bit über die Leitung, ohne Adressen und ohne Rückmeldung. Das Display schickt nichts zurück; die Leitung „Daten zurück" (MISO) bleibt beim PicoBoy deshalb unbenutzt. Dafür hat das Display zwei zusätzliche Leitungen:



orange: vom SPI-Baustein bedient grün/lila: gewöhnliche Pins

- **CS** (GP10): LOW heißt „du bist gemeint". Solange CS HIGH ist, ignoriert das Display alles auf den Leitungen.
- **DC** (GP8, *data/command*): LOW heißt „dieses Byte ist ein Befehl", HIGH heißt „das sind Daten zum Befehl". Das ist eine Eigenheit von Displays, kein Teil von SPI.
- **RST** (GP9): LOW startet den Displaychip neu.

Warum stimmen die Pins von selbst? Im Konstruktor `Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST)` kommen Takt und Daten gar nicht vor. Die Bibliothek nimmt dann die SPI-Schnittstelle `SPI` mit ihren Standardpins. Für das Board „Generic RP2350“ stehen sie in `variants/generic_rp2350/pins_arduino.h`:

```
#define PIN_SPI0_MOSI  (19u)
#define PIN_SPI0_SCK   (18u)
```

GP18 und GP19 sind genau die Pins des Displays. Anders als bei I2C (Kapitel 22) passt hier die Voreinstellung zufällig.

23.2 Was `tft.init(240, 280)` tut

Die Datei `Adafruit_ST7789.cpp` liegt im Ordner der Bibliothek (`libraries/Adafruit_ST7735_and_ST7789_Library` im Sketchbook). `init` macht vier Dinge.

1. Pins und SPI einrichten, Hardware-Reset. `init` ruft über `commonInit` und `begin` die Funktion `initSPI` aus Adafruit GFX auf (`Adafruit_SPITFT.cpp`). Sie stellt CS und DC auf Ausgang, startet `SPI` mit dem gewünschten Takt (ohne Angabe 32 MHz, so steht es als `SPI_DEFAULT_FREQ` in `Adafruit_ST77xx.cpp`) und zieht RST kurz auf LOW:

```
digitalWrite(_rst, HIGH);
delay(100);
digitalWrite(_rst, LOW);
delay(100);
digitalWrite(_rst, HIGH);
delay(200);
```

2. Die Startbefehle schicken. Am Anfang von `Adafruit_ST7789.cpp` steht eine Tabelle `generic_st7789[]`. Jeder Eintrag ist ein Befehl an den Displaychip ST7789, dazu seine Daten und eine Wartezeit:

Befehl	Nummer	Bedeutung	danach warten
SWRESET	`0x01`	Neustart des Chips	150 ms
SLPOUT	`0x11`	aus dem Schlafmodus aufwachen	10 ms
COLMOD	`0x3A`	Farbformat; Daten `0x55` = 16 Bit pro Pixel (RGB565)	10 ms
MADCTL	`0x36`	Speicherrichtung (siehe unten)	–
CASET	`0x2A`	Spalten 0 bis 240	–
RASET	`0x2B`	Zeilen 0 bis 320	–

Befehl	Nummer	Bedeutung	danach warten
INVON	`0x21`	Farben umkehren	10 ms
NORON	`0x13`	normale Anzeige	10 ms
DISPON	`0x29`	Anzeige einschalten	10 ms

Das Programmstück `displayInit` geht diese Tabelle durch. Zusammen mit dem Reset dauert `init` also gut 0,6 Sekunden.

Der Kommentar bei INVON lautet im Quelltext schlicht `// 7: hack`. Viele dieser Displays, auch das des PicoBoy, zeigen die Farben nur mit eingeschalteter Umkehrung richtig an. Deshalb gehört INVON hier zum **normalen** Zustand.

3. Den Ausschnitt festlegen. Der Chip ST7789 hat Speicher für 240 · 320 Pixel. Das Glas des PicoBoy zeigt aber nur 280 Zeilen. Für alle Größen außer 240 · 240 und 135 · 240 rechnet `init`:

```
_rowstart = _rowstart2 = (int)((320 - height) / 2);
_colstart = _colstart2 = (int)((240 - width) / 2);
```

Mit `init(240, 280)` ist das `_rowstart = 20` und `_colstart = 0`:



Die sichtbaren Zeilen sind die Zeilen 20 bis 299 des Chipspeichers. Die Bibliothek zählt zu jedem y heimlich 20 dazu. Deshalb ist es so wichtig, `init(240, 280)` zu schreiben: Mit `init(240, 240)`, der Größe vieler anderer Displays, rechnet die Bibliothek mit `_rowstart = 80`, und das Bild säße verschoben.

4. Die Richtung einstellen. Zum Schluss ruft `init` `setRotation(0)` auf. Das schickt den Befehl MADCTL mit dem Wert `ST77XX_MADCTL_MX | ST77XX_MADCTL_MY | ST77XX_MADCTL_RGB`, also `0xC0`:

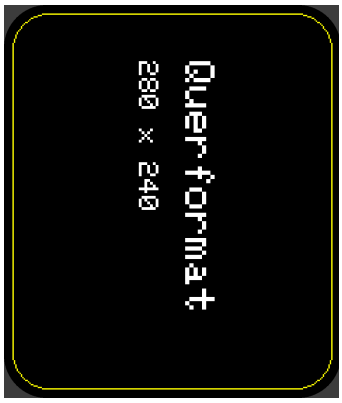
Bit	Name	wirkt
`0x80`	MY	Zeilen von unten nach oben schreiben
`0x40`	MX	Spalten von rechts nach links schreiben
`0x20`	MV	Zeilen und Spalten tauschen (Querformat)
`0x08`	BGR	Blau und Rot in den Farbzahlen tauschen

Wie der Chip eingebaut ist, entscheidet, welche Kombination ein richtig herum stehendes Bild ergibt. Für `setRotation(1)` bis `(3)` stehen in `Adafruit_ST7789.cpp` die anderen Kombinationen, zusammen mit dem passenden Versatz. Probiere es aus:

k23_rotation.ino

```
10
11 void setup() {
12     tft.init(240, 280);
13     pinMode(TFT_LICHT, OUTPUT);
14     digitalWrite(TFT_LICHT, HIGH);
15     tft.setRotation(1);                        // um 90 Grad gedreht
16     tft.fillScreen(ST77XX_BLACK);
17     tft.drawRoundRect(6, 6, tft.width() - 12, tft.height() - 12, 26,
         ST77XX_YELLOW);
18     tft.setTextColor(ST77XX_WHITE);
19     tft.setTextSize(3);
20     tft.setCursor(40, 90);
21     tft.print("Querformat");
22     tft.setTextSize(2);
23     tft.setCursor(40, 130);
24     tft.printf("%d x %d", tft.width(), tft.height());
```

Nach `setRotation(1)` ist das Display 280 Pixel breit und 240 hoch; `tft.width()` und `tft.height()` liefern die neuen Werte. Alle Zeichenbefehle rechnen von da an im Querformat. Die runden Ecken bleiben, wo sie sind.



23.3 Wie ein Pixel auf das Display kommt

Jeder Zeichenbefehl der Bibliothek läuft auf dasselbe Muster hinaus. Für ein Rechteck (`fillRect`) oder ein Bild (`drawRGBBitmap`):

1. **CS auf LOW** (`startWrite`).
2. **Fenster festlegen** (`setAddrWindow` in `Adafruit_ST77xx.cpp`): Befehl CASET mit erster und letzter Spalte, Befehl RASET mit erster und letzter Zeile, jeweils plus Versatz. Danach Befehl RAMWR (`0x2C`, „Speicher schreiben“).
3. **Die Farben schicken:** für jedes Pixel 2 Byte, Zeile für Zeile durch das Fenster. Der Chip zählt selbst weiter.
4. **CS auf HIGH** (`endWrite`).

Schritt 2 kostet jedes Mal 11 Byte und das Umschalten von DC, egal wie groß das Fenster ist. Bei einem Vollbild fällt das nicht ins Gewicht. Bei einzelnen Pixeln dagegen schon:

k23_pixel_einzeln.ino

```
21
22      unsigned long start = millis();
23      for (int y = 0; y < 280; y++) {
24          for (int x = 0; x < 240; x++) {
25              tft.drawPixel(x, y, ST77XX_RED);      // jedes Mal: Fenster + 2 Bytes
26          }
27      }
28      Serial.printf("drawPixel einzeln:  %lu ms\n", millis() - start);
29
30      bild.fillScreen(ST77XX_GREEN);
31      start = millis();
32      tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);   // ein Fenster
33      Serial.printf("drawRGBBitmap:      %lu ms\n", millis() - start);
34  }
```

Serieller Monitor

```
drawPixel einzeln:  201 ms
drawRGBBitmap:      29 ms
```

`drawPixel` legt für jedes Pixel ein eigenes Fenster an: Für 2 Byte Farbe kommen jedes Mal noch Befehle und Umschalten dazu. `drawRGBBitmap` legt ein einziges Fenster an und schickt dann 134 400 Byte am Stück. Deshalb ist der Canvas aus Kapitel 14 so viel schneller als viele kleine Zeichenbefehle auf `tft`.

> Die Zeiten oben stammen aus dem Prüflauf auf dem PC. Er rechnet für jedes Pixel und jeden Befehl die Zeit auf der SPI-Leitung aus. Auf dem PicoBoy kommt für jedes `drawPixel` noch die Rechenzeit des Prozessors dazu, der Abstand wird also eher größer. Lade den Sketch hoch und miss selbst.

Wie `drawRGBBitmap` die Pixel schickt, steht für den RP2350 in `Adafruit_SPITFT.cpp`, in der Funktion `writePixels`: Sie stellt den SPI-Baustein für die Dauer des Bildes auf 16 Bit pro Übertragung um und ruft `spi_write16_blocking` aus dem Pico SDK auf. So gehen die 16-Bit-Farben ohne Umweg hinaus, eine Zeile nach der anderen.

23.4 Der SPI-Takt

Kapitel 14 hat die Takt-Stufen gezeigt: 75 MHz geteilt durch eine ganze Zahl. Woher kommen sie? Der SPI-Baustein bekommt seinen Takt von `clk_peri`, und der ist so schnell wie der Prozessor, 150 MHz. Die Funktion `spi_set_baudrate` im Pico SDK (`pico-sdk/src/rp2_common/hardware_spi/spi.c`) teilt ihn in zwei Schritten: durch einen geraden **Vorteiler** von 2 bis 254 und durch einen **Nachteiler** von 1 bis 256. Sie nimmt den kleinsten Vorteiler, der passt, und dazu den größten Nachteiler, bei dem der Takt nicht **über** dem Wunsch liegt. Für alle Wünsche über etwa 0,3 MHz ist der Vorteiler 2, und es bleibt 150 MHz / 2 / n = 75 MHz / n.

k23_spi_takt.ino

```
12
13 void setup() {
14     Serial.begin(115200);
15     while (!Serial) {
16         delay(10);
17     }
18     tft.init(240, 280);
19     pinMode(TFT_LICHT, OUTPUT);
20     digitalWrite(TFT_LICHT, HIGH);
21     bild.fillScreen(ST77XX_BLUE);
22
23     for (int i = 0; i < 5; i++) {
24         tft.setSPISpeed(WUENSCHE[i]);
25         unsigned long start = micros();
26         tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
27         unsigned long dauer = micros() - start;
28         Serial.printf("Wunsch %4.1f MHz: %5lu us\n", WUENSCHE[i] / 1000000.0,
             dauer);
29     }
30     tft.setSPISpeed(62500000);
31 }
```

Serieller Monitor

```
Wunsch 32.0 MHz: 43012 us
Wunsch 40.0 MHz: 28676 us
Wunsch 50.0 MHz: 28676 us
Wunsch 62.5 MHz: 28676 us
Wunsch 75.0 MHz: 14340 us
```

- 32 MHz (die Vorgabe der Bibliothek) ergibt 25 MHz, ein Bild dauert etwa 43 ms.
- 40, 50 und 62,5 MHz ergeben alle 37,5 MHz: Erst der Wunsch 75 MHz erreicht die nächste Stufe.

- 75 MHz halbiert die Zeit noch einmal. Der Displaychip ist aber nicht für so viel gebaut; ob es auf deinem PicoBoy fehlerfrei läuft, musst du ausprobieren. Dieses Buch bleibt bei `setSPISpeed(62500000)`.

Auch diese Zeiten sind im Prüflauf aus dem Takt berechnet; auf dem Gerät kommen ein paar Mikrosekunden für die Befehle dazu.

> Den tatsächlichen Takt sagt dir `SPI` nicht direkt, aber das Pico SDK: `spi_get_baudrate(spi0)` liefert ihn in Hertz. Die Funktion kannst du im Sketch direkt aufrufen, nach `#include <hardware/spi.h>`; sie läuft nur auf dem Gerät.

23.5 Befehle selbst schicken

Die Bibliothek hat eine Methode, mit der du dem Chip beliebige Befehle schickst: `tft.sendCommand(befehl, daten, anzahl)`. Damit kannst du ausprobieren, was die Befehle aus Abschnitt 23.2 bewirken. Dieser Sketch läuft nur auf dem PicoBoy; der Prüflauf auf dem PC bildet den Displaychip nicht nach.

k23_befehle.ino

```
22 void richtung(uint8_t madctl) {
23     tft.sendCommand(ST77XX_MADCTL, &madctl, 1);
24     zeichne();                      // MADCTL wirkt auf das, was danach kommt
25 }
26
27 void setup() {
28     Serial.begin(115200);
29     tft.init(240, 280);
30     pinMode(TFT_LICHT, OUTPUT);
31     digitalWrite(TFT_LICHT, HIGH);
32     zeichne();
33 }
34
35 void loop() {
36     Serial.println("INVOFF: Farben umgekehrt");
37     tft.sendCommand(ST77XX_INVOFF);
38     delay(2000);
39     tft.sendCommand(ST77XX_INVON);          // so stellt init() es ein
40
41     Serial.println("MADCTL 0xC8: mit BGR-Bit");
42     richtung(0xC8);                         // MY | MX | BGR
43     delay(2000);
44
45     Serial.println("MADCTL 0x00: ohne MY und MX");
46     richtung(0x00);
47     delay(2000);
```

```
    Serial.println("MADCTL 0xC0: wie setRotation(0)");
    richtung(0xC0);                          // MY | MX | RGB
    delay(2000);

    Serial.println("DISPOFF: Anzeige aus, Bild bleibt im Chip");
    tft.sendCommand(ST77XX_DISPOFF);
    delay(2000);
    tft.sendCommand(ST77XX_DISPON);
```

- **INVOFF** schaltet die Farbumkehr aus: Schwarz wird weiß, Rot wird türkis. Das wirkt sofort auf das ganze Bild.
- **MADCTL** legt fest, wie die nächsten Pixel in den Speicher geschrieben werden. Was schon im Speicher steht, bleibt, wie es ist; deshalb zeichnet `richtung()` danach neu. Mit dem BGR-Bit (`0xC8`) tauschen Rot und Blau die Plätze, ohne MY und MX (`0x00`) steht das Bild auf dem Kopf.
- **DISPOFF** schaltet nur die Anzeige ab. Das Bild bleibt im Speicher des Chips und ist mit DISPON sofort wieder da.

Die Bibliothek hat für die wichtigsten Befehle auch eigene Methoden: `tft.invertDisplay(bool)` (INVON/INVOFF), `tft.enableDisplay(bool)` (DISPON/DISPOFF) und `tft.enableSleep(bool)` (SLPIN/SLPOUT). Achtung bei `invertDisplay`: Wegen des „hack“ ist hier `invertDisplay(true)` der Normalzustand, `invertDisplay(false)` kehrt die Farben um.

Ausprobieren: Das Display ohne Display-Bibliothek

Jetzt weißt du genug, um das Display ganz allein anzusteuern: mit der Bibliothek `SPI` aus dem Kern, drei Pins und den Befehlen aus Abschnitt 23.2. Auch dieser Sketch läuft nur auf dem PicoBoy.

k23_spi_direkt.ino

```
// Das Display ganz ohne Display-Bibliothek: nur SPI und drei Pins
#include <SPI.h>

#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26

SPISettings einstellung(25000000, MSBFIRST, SPI_MODE0);

void start() {                    // Bus belegen, Chip auswaehlen
    SPI.beginTransaction(einstellung);
    digitalWrite(TFT_CS, LOW);
}

void ende() {                     // Chip abwaehlen, Bus freigeben
    digitalWrite(TFT_CS, HIGH);
    SPI.endTransaction();
}
```

```
21 void kommando(uint8_t nummer) {   // DC = LOW: dieses Byte ist ein Befehl
22     digitalWrite(TFT_DC, LOW);
23     SPI.transfer(nummer);
24     digitalWrite(TFT_DC, HIGH);   // alles Weitere sind Daten
25 }
26
27 void befehl(uint8_t nummer, int pause) {
28     start();
29     kommando(nummer);
30     ende();
31     delay(pause);
32 }
33
34 void befehl_mit_wert(uint8_t nummer, uint8_t wert) {
35     start();
36     kommando(nummer);
37     SPI.transfer(wert);
38     ende();
39 }
40
41 void fuelle(int x, int y, int breite, int hoehe, uint16_t farbe) {
42     start();
43     kommando(0x2A);               // CASET: Spalten von ... bis
44     SPI.transfer16(x);
45     SPI.transfer16(x + breite - 1);
46     kommando(0x2B);               // RASET: Zeilen, sichtbar ab Zeile 20
47     SPI.transfer16(y + 20);
48     SPI.transfer16(y + 20 + hoehe - 1);
49     kommando(0x2C);               // RAMWR: jetzt kommen die Pixel
50     for (long i = 0; i < (long)breite * hoehe; i++) {
51         SPI.transfer16(farbe);
52     }
53     ende();
54 }
55
56 void setup() {
57     pinMode(TFT_CS, OUTPUT);
58     digitalWrite(TFT_CS, HIGH);
59     pinMode(TFT_DC, OUTPUT);
60     digitalWrite(TFT_DC, HIGH);
61     pinMode(TFT_RST, OUTPUT);
62     SPI.begin();                  // GP18 = Takt, GP19 = Daten
63
64     digitalWrite(TFT_RST, HIGH);  // Hardware-Reset wie in der Bibliothek
65     delay(100);
66     digitalWrite(TFT_RST, LOW);
67     delay(100);
68     digitalWrite(TFT_RST, HIGH);
69     delay(200);
70
71     befehl(0x01, 150);            // SWRESET: Neustart
72     befehl(0x11, 10);             // SLPOUT: aufwachen
73     befehl_mit_wert(0x3A, 0x55);  // COLMOD: 16 Bit pro Pixel
```

```
74      befehl_mit_wert(0x36, 0xC0);  // MADCTL: Richtung wie setRotation(0)
75      befehl(0x21, 10);             // INVON: dieses Panel braucht es
76      befehl(0x13, 10);             // NORON: normale Anzeige
77      befehl(0x29, 10);             // DISPON: Anzeige an
78
79      pinMode(TFT_LICHT, OUTPUT);
80      digitalWrite(TFT_LICHT, HIGH);
81      fuelle(0, 0, 240, 280, 0x0000);
82      fuelle(20, 40, 200, 60, 0xF800);
83      fuelle(20, 110, 200, 60, 0x07E0);
84      fuelle(20, 180, 200, 60, 0x001F);
85  }
86
87  void loop() {
88  }
```

- **Zeile 9:** `SPISettings` fasst Takt, Bitreihenfolge (das höchste Bit zuerst, `MSBFIRST`) und SPI-Modus zusammen. Modus 0 ist der, den auch die Bibliothek benutzt (`SPI_MODE0` als Standardwert von `init`).
- **Zeilen 11 bis 19:** `beginTransaction` stellt den SPI-Baustein ein und reserviert ihn; dann geht CS auf LOW. `ende()` macht beides rückgängig. Genau das tun `startWrite` und `endWrite` in Adafruit GFX.
- **Zeilen 21 bis 25:** `kommando` schickt ein Byte mit DC auf LOW. Alles danach, mit DC auf HIGH, sind Daten.
- **Zeilen 41 bis 54:** `fuelle` ist das Muster aus Abschnitt 23.3: Fenster mit CASET und RASET (mit dem Versatz 20), RAMWR, dann die Farbe für jedes Pixel. `SPI.transfer16` schickt eine 16-Bit-Zahl, das höhere Byte zuerst; genau in dieser Reihenfolge erwartet der Chip die Farbe.
- **Zeilen 64 bis 77:** Reset und Startbefehle wie in der Bibliothek. `0xC0` für MADCTL ist der Wert von `setRotation(0)`.
- **Zeile 80:** Ohne Hintergrundlicht bliebe trotzdem alles schwarz.

Auf dem Display erscheinen drei Balken: rot, grün, blau. Das ganze Programm braucht keine Display-Bibliothek und ist in gut 80 Zeilen fertig. Die Adafruit-Bibliotheken bringen dafür Schriften, Kreise, Linien, Bitmaps und die Pflege für Dutzende Displays mit.

Zum Verändern: Ändere den Versatz von 20 auf 0. Was passiert mit den Balken? Schreibe eine Funktion `pixel(x, y, farbe)` und zeichne damit eine Linie.

Übungsaufgaben

> **Aufgabe 1** ⭐ **Nachgerechnet** Wie viele Bytes muss der PicoBoy für ein ganzes Bild über SPI schicken? Wie lange dauert das bei 25 MHz, bei 37,5 MHz und bei 75 MHz? Wie viele Bilder pro Sekunde sind damit höchstens möglich?

Aufgabe 2 ⭐ Umkehren auf Knopfdruck Zeichne einen roten Kreis auf weißem Grund. Jeder Druck auf A soll mit `tft.invertDisplay` zwischen normaler und umgekehrter Darstellung wechseln. Mit welchem Wert musst du anfangen?

Aufgabe 3 ⭐⭐ Alle vier Richtungen Schreibe einen Sketch, der mit jedem Druck auf A die nächste Rotation (0, 1, 2, 3) einstellt und „oben“ an den oberen Rand schreibt, dazu die Nummer der Rotation sowie Breite und Höhe. Halte den PicoBoy jeweils so, dass du „oben“ lesen kannst.

Aufgabe 4 ⭐⭐ Nur den Ausschnitt senden Ein Ball von 24 Pixeln Durchmesser springt über einen schwarzen Bildschirm. Statt jedes Mal das ganze Bild zu senden, zeichne den Ball in einen kleinen `GFXcanvas16` von 40 · 40 Pixeln und sende nur diesen an die Stelle des Balls. Miss im Mittel über 100 Bilder, wie lange das Senden dauert, und vergleiche mit einem Vollbild. Warum muss der kleine Canvas größer sein als der Ball?

Aufgabe 5 ⭐⭐⭐ Schachbrett ohne Bibliothek Erweitere `k23_spi_direkt` so, dass es ein Schachbrett aus Feldern von 30 · 30 Pixeln zeichnet. Achte auf die letzte Reihe: 280 ist nicht durch 30 teilbar. Miss, wie lange das ganze Brett dauert, und vergleiche mit einem einzigen `fuelle` über den ganzen Bildschirm.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- **SPI**: Takt (GP18) und Daten (GP19) vom SPI-Baustein, dazu CS (GP10), DC (GP8, LOW = Befehl) und RST (GP9) als gewöhnliche Pins. Die Standardpins von `SPI` passen zum PicoBoy.
- `tft.init(240, 280)`: Reset, Startbefehle (SWRESET, SLPOUT, COLMOD 16 Bit, MADCTL, INVON, NORON, DISPON), Versatz 20 Zeilen, `setRotation(0)` mit MADCTL `0xC0`. INVON ist bei diesem Display der Normalzustand.
- Der Chip hat 240 · 320 Pixel Speicher; sichtbar sind die Zeilen 20 bis 299.
- Jeder Zeichenbefehl: Fenster (CASET, RASET), RAMWR, Farben. Ein großes Fenster ist viel schneller als viele kleine.
- SPI-Takt = 75 MHz / n; ohne `setSPISpeed` 25 MHz, mit `setSPISpeed(62500000)` 37,5 MHz.
- `tft.sendCommand(befehl, daten, anzahl)` schickt eigene Befehle; ganz ohne Display-Bibliothek geht es mit `SPI.beginTransaction`, `SPI.transfer` und `SPI.transfer16`.

KAPITEL 24

Interrupts und der zweite Kern

Worum geht's? Bisher macht dein Sketch alles nacheinander in `loop()`. Ist `loop()` gerade beschäftigt, bemerkt er einen kurzen Tastendruck nicht. In diesem Kapitel lernst du zwei Auswege: **Interrupts**, bei denen der Chip das laufende Programm kurz unterbricht, sobald an einem Pin etwas passiert, und den **zweiten Prozessorkern** des RP2350, auf dem eine zweite Schleife gleichzeitig läuft. Am Ende spielt ein Kern Musik, während der andere ein Spiel zeichnet.

24.1 Abfragen oder unterbrechen lassen

Seit Kapitel 9 fragst du Tasten so ab: In jedem Durchlauf von `loop()` liest `digitalRead` den Pin und vergleicht mit dem letzten Mal. Das heißt **Abfragen** (englisch *polling*). Es funktioniert gut, solange `loop()` oft genug vorbeikommt. Was aber, wenn ein Durchlauf lange dauert? Im folgenden Sketch steht `delay(1000)` für eine Arbeit, die eine Sekunde braucht. Im Prüflauf wurde Taste A fünfmal kurz gedrückt, jeweils 150 ms lang:

k24_verpasst.ino

```
// Taste A abfragen, waehrend loop() lange mit etwas anderem beschaeftigt ist
#define TASTE_A 27

int zaehler = 0;
bool vorher = false;

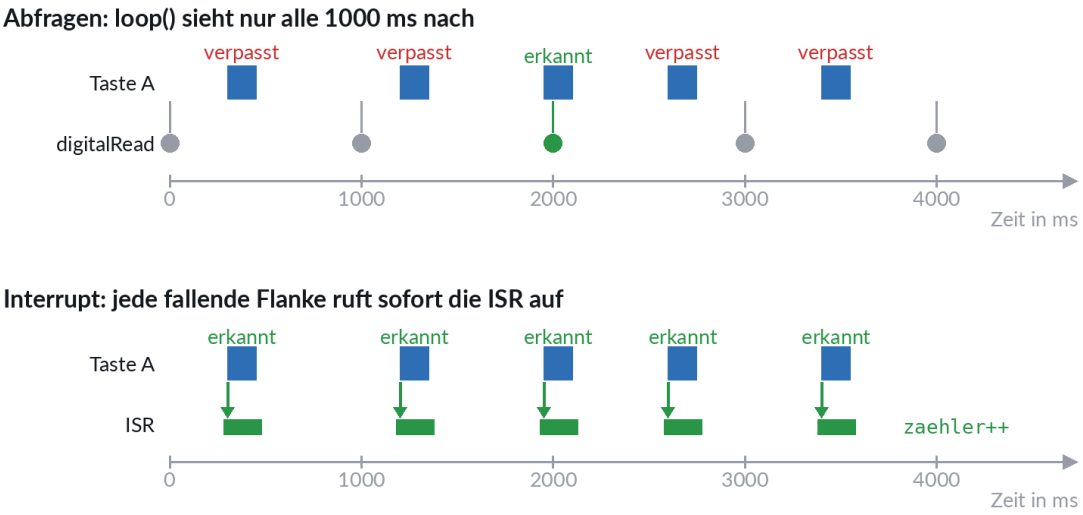
void setup() {
    Serial.begin(115200);
    pinMode(TASTE_A, INPUT_PULLUP);
}

void loop() {
    bool jetzt = digitalRead(TASTE_A) == LOW;
    if (jetzt && !vorher) {
        zaehler++;
    }
    vorher = jetzt;
    Serial.printf("%lu ms: %d Druecke gezaehlt\n", millis(), zaehler);
    delay(1000);    // steht fuer eine Arbeit, die lange dauert
}
```

Serieller Monitor

```
0 ms: 0 Druecke gezaehlt
1000 ms: 0 Druecke gezaehlt
2000 ms: 1 Druecke gezaehlt
3000 ms: 1 Druecke gezaehlt
4000 ms: 1 Druecke gezaehlt
```

Von fünf Drücken hat der Sketch einen bemerkt. Nur dieser eine fiel zufällig in den Augenblick, in dem `digitalRead` nachsah. Die anderen vier begannen und endeten, während der Sketch in `delay` wartete:



Die Lösung ist ein **Interrupt** (deutsch *Unterbrechung*). Dabei passt nicht dein Sketch auf den Pin auf, sondern die Hardware des Chips. Sobald sich der Pegel ändert, unterbricht sie das laufende Programm, egal wo es gerade steht, auch mitten in `delay`. Dann führt sie eine Funktion aus, die du vorher angemeldet hast. Diese Funktion heißt **Interrupt-Service-Routine**, kurz **ISR**. Ist sie fertig, macht das Programm genau dort weiter, wo es unterbrochen wurde.

k24_interrupt.ino

```
// Taste A per Interrupt zaehlen: kein Druck geht verloren
#define TASTE_A 27

volatile int zaehler = 0;

// die Interrupt-Routine: kurz und ohne Warten
void taste_a_gedrueckt() {
    zaehler++;
}

void setup() {
    Serial.begin(115200);
    pinMode(TASTE_A, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(TASTE_A), taste_a_gedrueckt, FALLING);
}

void loop() {
    Serial.printf("%lu ms: %d Druecke gezaehlt\n", millis(), zaehler);
    delay(1000);    // steht fuer eine Arbeit, die lange dauert
}
```

```
Serieller Monitor
0 ms: 0 Druecke gezaehlt
1000 ms: 1 Druecke gezaehlt
2000 ms: 3 Druecke gezaehlt
3000 ms: 4 Druecke gezaehlt
4000 ms: 5 Druecke gezaehlt
```

Jetzt sind alle fünf Drücke gezählt, obwohl `loop()` genauso lange schläft.

Zeile 14 meldet die ISR an:

```
attachInterrupt(digitalPinToInterrupt(TASTE_A), taste_a_gedrueckt, FALLING);
```

Teil	Bedeutung
`digitalPinToInterrupt(TASTE_A)`	die Nummer des Interrupts für diesen Pin
`taste_a_gedrueckt`	die ISR: der Name einer Funktion **ohne** Klammern, denn sie soll ja nicht jetzt aufgerufen werden, sondern später
`FALLING`	wann: bei einer **fallenden Flanke**, also wenn der Pin von `HIGH` auf `LOW` wechselt

Die Taste ist gedrückt `LOW` (Kapitel 9). Eine fallende Flanke ist also genau der Moment, in dem du drückst. Außer `FALLING` kennt der Kern `RISING` (steigende Flanke, hier: Loslassen), `CHANGE` (beides), `LOW` und `HIGH` (solange der Pegel so ist). Bei Tasten brauchst du fast immer `FALLING`.

Auf vielen Arduino-Platinen haben nur wenige Pins einen Interrupt, und `digitalPinToInterrupt` rechnet die Pinnummer in die Interruptnummer um. Beim RP2350 kann **jeder** Pin einen Interrupt auslösen, und `digitalPinToInterrupt(pin)` ist im Kern einfach `pin`. Schreib es trotzdem hin: So sieht jeder sofort, was gemeint ist, und der Sketch passt auch auf andere Platinen. Mit `detachInterrupt(digitalPinToInterrupt(TASTE_A))` meldest du die ISR wieder ab.

Zeile 7 bis 9 sind die ISR. Sie hat keine Parameter und keinen Rückgabewert (`void`), denn niemand ruft sie im Sketch auf; die Hardware tut es. Sie zählt nur `zaehler` um eins hoch. Das neue Wort `volatile` in Zeile 4 erklärt der nächste Abschnitt.

> Die Prüfung der Sketche auf dem PC bildet Interrupts nach: Zu Beginn jedes Tastendrucks aus dem Prüfskript ruft sie die angemeldete ISR auf. Auf dem PicoBoy macht das die Hardware des RP2350.

24.2 `volatile` und kurze ISRs

`volatile` (deutsch *flüchtig*) sagt dem Compiler: Diese Variable kann sich jederzeit ändern, auch ohne dass der Code drumherum sie anfasst. Der Compiler optimiert nämlich gern. Sieht er, dass `loop()` eine Variable nur liest und nie ändert, darf er ihren Wert einmal holen und sich eine Kopie

merken. Dass zwischendurch eine ISR hineinschreibt, kann er nicht wissen, denn die ISR wird ja nirgends im Sketch aufgerufen. Mit `volatile` muss er die Variable bei jedem Lesen frisch aus dem Speicher holen.

> Jede Variable, die eine ISR verändert und das übrige Programm liest (oder umgekehrt), bekommt **`volatile`**.

Während die ISR läuft, steht das Hauptprogramm. Deshalb gilt: **Eine ISR ist kurz.** Sie merkt sich, was passiert ist, und kehrt sofort zurück. Was in einer ISR nichts zu suchen hat:

Nicht in der ISR	Warum?
`delay`	Der Kern wartet mit den Wartefunktionen des Pico SDK, und dessen Beschreibung sagt ausdrücklich: nicht aus einem Interrupt aufrufen.
`Serial.print…`	Der USB-Teil ist mit einer Sperre gegen gleichzeitigen Zugriff geschützt. Hält das unterbrochene Programm die Sperre gerade selbst, gibt die Ausgabe in der ISR einfach auf, und der Text fehlt.
`tone`	auch mit einer Sperre geschützt, legt außerdem Speicher an
Zeichnen auf dem Display	dauert viele Millisekunden

Erlaubt und nützlich sind dagegen `millis()` und `micros()`: Sie lesen nur die Uhr des Chips. Damit kann sich die ISR merken, **wann** etwas passiert ist.

Das übliche Muster: Die ISR setzt eine **Flagge** (eine `bool`-Variable, die sagt „es ist etwas passiert“) und merkt sich vielleicht die Zeit. Die Arbeit macht `loop()`, sobald es die Flagge sieht:

k24_flagge.ino

```
// Die ISR setzt nur eine Flagge, die Arbeit macht loop()
#define TASTE_B 28
#define LED_GELB 13
#define LAUTSPRECHER 15
#define SPERRZEIT 200

volatile bool gedrueckt = false;
volatile unsigned long druck_zeit = 0;
bool led_an = false;

void taste_b_gedrueckt() {
    unsigned long jetzt = millis();
    if (jetzt - druck_zeit >= SPERRZEIT) {
        druck_zeit = jetzt;
        gedrueckt = true;          // Flagge setzen, sonst nichts
    }
}

void setup() {
    Serial.begin(115200);
```

```
21      pinMode(TASTE_B, INPUT_PULLUP);
22      pinMode(LED_GELB, OUTPUT);
23      attachInterrupt(digitalPinToInterrupt(TASTE_B), taste_b_gedrueckt, FALLING);
24  }
25
26  void loop() {
27      if (gedrueckt) {
28          noInterrupts();             // beide Werte zusammen abholen
29          unsigned long zeit = druck_zeit;
30          gedrueckt = false;
31          interrupts();
32
33          led_an = !led_an;
34          digitalWrite(LED_GELB, led_an);
35          tone(LAUTSPRECHER, 880, 100);
36          Serial.printf("Taste B bei %lu ms, bearbeitet bei %lu ms\n", zeit,
              millis());
37      }
38      delay(300);     // loop() ist mit anderem beschaeftigt
39  }
```

Serieller Monitor

```
Taste B bei 500 ms, bearbeitet bei 600 ms
Taste B bei 1700 ms, bearbeitet bei 1800 ms
```

- Die ISR (Zeile 11 bis 17) setzt nur `druck_zeit` und `gedrueckt`. Was es mit `SPERRZEIT` auf sich hat, zeigt Abschnitt 24.3.
- `loop()` prüft in Zeile 27 die Flagge. Erst dort leuchtet die LED und klingt der Ton. Weil `loop()` jedes Mal 300 ms schläft, wird der Druck bis zu 300 ms später **bearbeitet**. Den genauen **Zeitpunkt** kennt der Sketch trotzdem, denn die ISR hat ihn sofort notiert.

Zeile 28 bis 31 holen beide Werte ab. `noInterrupts()` verbietet Interrupts für einen Moment, `interrupts()` erlaubt sie wieder. Dazwischen kann keine ISR dazwischenfunken. Ohne diesen Schutz könnte die ISR genau zwischen Zeile 29 und 30 zuschlagen: `zeit` hätte dann noch den alten Zeitpunkt, und Zeile 30 würde die Flagge für den neuen Druck löschen, der damit verloren wäre. Halte solche Abschnitte so kurz wie hier; solange Interrupts verboten sind, bemerkt der Chip auch keine Tasten.

24.3 Prellen und die Sperrzeit

In Kapitel 9 hast du gesehen, dass ein Tastenkontakt **prellt**: Er federt beim Drücken einige Millisekunden lang und schließt und öffnet dabei mehrmals. Für einen Interrupt ist jedes dieser Schließen eine neue fallende Flanke. Dieser Sketch macht das sichtbar. Seine ISR schreibt den Zeitpunkt jeder Flanke in ein Array, `loop()` gibt die neuen Einträge aus:

k24_prellen.ino

```
// Prell-Detektor: jede fallende Flanke von Taste A mit ihrem Zeitpunkt
#define TASTE_A 27
#define MAX_FLANKEN 20

volatile unsigned long zeiten[MAX_FLANKEN];
volatile int anzahl = 0;
int ausgegeben = 0;

void taste_a_flanke() {
    if (anzahl < MAX_FLANKEN) {
        zeiten[anzahl] = micros();
        anzahl++;
    }
}

void setup() {
    Serial.begin(115200);
    pinMode(TASTE_A, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(TASTE_A), taste_a_flanke, FALLING);
}

void loop() {
    while (ausgegeben < anzahl) {
        Serial.printf("Flanke %d bei %.3f ms\n", ausgegeben + 1,
          zeiten[ausgegeben] / 1000.0);
        ausgegeben++;
    }
    delay(100);
}
```

Serieller Monitor

```
Flanke 1 bei 1000.001 ms
Flanke 2 bei 1003.001 ms
Flanke 3 bei 1007.001 ms
Flanke 4 bei 1150.001 ms
Flanke 5 bei 2000.001 ms
```

Das Prüfskript hat hier eine prellende Taste nachgemacht: drei Flanken innerhalb von 7 ms beim Drücken, eine weitere beim Loslassen nach 150 ms, dann ein sauberer zweiter Druck. Ob und wie stark die Tasten deines PicoBoy prellen, zeigt dir derselbe Sketch auf dem Gerät. Drücke dort ein paar Mal langsam und schnell und sieh dir die Zeiten an.

Der Sketch aus 24.1 hätte hier **fünf** Drücke gezählt. Abhilfe schafft eine **Sperrzeit**: Nach einem gezählten Druck ignoriert die ISR alle weiteren Flanken, bis die Sperrzeit vorbei ist.

k24_sperrzeit.ino

```
// Interrupt mit Sperrzeit: Prellen wird nicht mitgezaehlt
#define TASTE_A 27
#define SPERRZEIT 200   // in Millisekunden

volatile int zaehler = 0;
volatile unsigned long letzter_druck = 0;

void taste_a_gedrueckt() {
    unsigned long jetzt = millis();
    if (jetzt - letzter_druck >= SPERRZEIT) {
        zaehler++;
        letzter_druck = jetzt;
    }
}

void setup() {
    Serial.begin(115200);
    pinMode(TASTE_A, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(TASTE_A), taste_a_gedrueckt, FALLING);
}

void loop() {
    Serial.printf("%lu ms: %d Druecke gezaehlt\n", millis(), zaehler);
    delay(1000);
}
```

Serieller Monitor

```
0 ms: 0 Druecke gezaehlt
1000 ms: 1 Druecke gezaehlt
2000 ms: 2 Druecke gezaehlt
```

Bei derselben prellenden Taste zählt der Sketch richtig zwei Drücke.

> Die Sperrzeit hat eine Grenze: Hältst du eine Taste **länger** als die Sperrzeit fest, kann das Prellen beim **Loslassen** noch eine fallende Flanke erzeugen, und die zählt wieder. Für gewöhnliche Tasten im Spiel ist deshalb das Abfragen mit Entprellen aus Kapitel 9 meist die bessere Wahl. Interrupts lohnen sich, wenn `loop()` lange beschäftigt ist, wenn ein Zeitpunkt genau gemessen werden soll oder wenn ohnehin nur der **erste** Druck zählt, wie beim Reaktionstest am Ende dieses Kapitels.

24.4 Der zweite Kern: `setup1()` und `loop1()`

Der RP2350 hat zwei **Prozessorkerne**: zwei vollständige Rechenwerke in einem Chip, die gleichzeitig arbeiten können. Bisher lief dein Sketch nur auf **Kern 0**, Kern 1 schlief. Im Kern von Earle Philhower weckst du ihn ganz einfach: Schreibe zusätzlich die Funktionen `setup1()` und `loop1()` in deinen Sketch. Alles, was darin steht, läuft auf Kern 1.

k24_zwei_kerne.ino

```
// Zwei Kerne, zwei Schleifen: jede LED blinkt in ihrem eigenen Takt
#define LED_ROT 14
#define LED_GRUEN 12

// Kern 0
void setup() {
    Serial.begin(115200);
    pinMode(LED_ROT, OUTPUT);
}

void loop() {
    digitalWrite(LED_ROT, HIGH);
    delay(500);
    digitalWrite(LED_ROT, LOW);
    delay(500);
    Serial.printf("loop() laeuft auf Kern %d\n", rp2040.cpuid());
}

// Kern 1
void setup1() {
    pinMode(LED_GRUEN, OUTPUT);
}

void loop1() {
    digitalWrite(LED_GRUEN, HIGH);
    delay(150);
    digitalWrite(LED_GRUEN, LOW);
    delay(150);
}
```

Die rote LED blinkt im Sekundentakt, die grüne viel schneller. Jede Schleife hat ihr eigenes `delay`, und keine bremst die andere. Mit nur einem Kern müsstest du dafür mit `millis()` rechnen (Kapitel 7.4).

So startet der Kern die beiden Schleifen (aus seiner Beschreibung, Abschnitt *Multicore Processing*):

- `setup()` und `setup1()` beginnen **gleichzeitig**.
- `loop()` beginnt, sobald `setup()` fertig ist, `loop1()`, sobald `setup1()` fertig ist. Die beiden Schleifen laufen also nicht im Gleichschritt.
- Fehlen `setup1()` und `loop1()`, bleibt Kern 1 schlafen, wie bisher.

`rp2040.cpuid()` in Zeile 16 liefert die Nummer des Kerns, auf dem der Code gerade läuft. Im seriellen Monitor erscheint deshalb jede Sekunde `loop() laeuft auf Kern 0`. (Das Objekt heißt `rp2040`, obwohl der Chip ein RP2350 ist; der Kern hat den Namen aus seiner Anfangszeit behalten.)

> Der zweite Kern lässt sich in der Prüfung auf dem PC nicht nachbilden. Die Sketche mit `setup1()` und `loop1()` in diesem Kapitel sind für den PicoBoy übersetzt und geprüft, laufen aber nur dort.

Wichtig zu wissen (alles aus der Beschreibung des Kerns):

- Auf Kern 1 läuft nur dein Code, keine Interrupts und nichts im Hintergrund. Die USB-Verbindung kümmert sich um Kern 0.
- Der **Stack**, der Speicher für lokale Variablen (Kapitel 19), wird geteilt: Mit `setup1()` / `loop1()` hat jeder Kern nur noch **4 KB** statt 8 KB. Große lokale Arrays gehören deshalb nicht in `loop1()`. Braucht Kern 1 mehr, bekommt er mit der Zeile `bool core1_separate_stack = true;` (außerhalb jeder Funktion) einen eigenen Stack von 8 KB.
- Schreibt Kern 0 in den Flash (`EEPROM.commit()`, Dateien in `LittleFS`, Kapitel 25), hält der Kern Kern 1 so lange an.

24.5 Gemeinsame Daten

Beide Kerne sehen denselben Arbeitsspeicher. Eine Variable außerhalb der Funktionen kann Kern 0 schreiben und Kern 1 lesen. Wie bei der ISR muss sie `volatile` sein, denn für den Compiler ändert sie sich „von außen“. Für kleine Werte wie ein `int` oder ein `bool` reicht das.

Soll ein Kern dem anderen eine **Nachricht** schicken, die genau einmal ankommen soll, gibt es eine **FIFO** (*first in, first out*): eine Warteschlange für Zahlen. Was zuerst hineinkommt, kommt zuerst heraus.

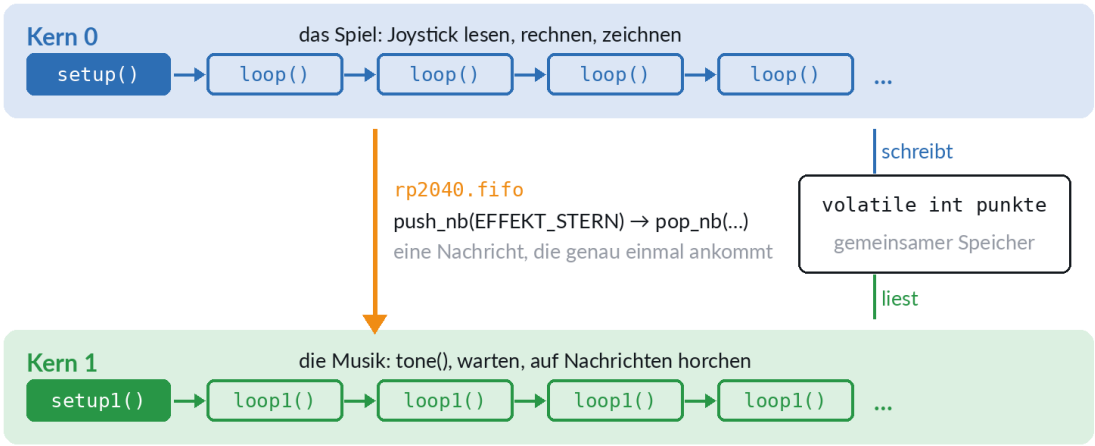
Befehl	Bedeutung
`rp2040.fifo.push(zahl)`	Zahl an den anderen Kern schicken; wartet, wenn die Warteschlange voll ist
`rp2040.fifo.push_nb(zahl)`	ebenso, wartet aber nie: gibt `false` zurück, wenn kein Platz ist (*nb = non-blocking*)
`rp2040.fifo.pop()`	die nächste Zahl abholen; wartet, bis eine da ist
`rp2040.fifo.pop_nb(&zahl)`	abholen, ohne zu warten: `true` und die Zahl in `zahl`, oder `false`, wenn nichts da ist
`rp2040.fifo.available()`	ob etwas zum Abholen da ist

Es gibt zwei Warteschlangen, eine in jede Richtung. Die Zahlen sind `uint32_t`, also ganze Zahlen ohne Vorzeichen. Beim RP2350 benutzt der Kern dafür die FIFO-Hardware des Chips, die nur wenige Zahlen aufnimmt. Die Kerne sollten Nachrichten also bald abholen.

Was sich die Kerne teilen müssen, und was nicht:

- **Nur ein Kern zeichnet auf das Display.** Beide würden sich sonst auf der SPI-Leitung (Kapitel 23) in die Quere kommen.
- `Serial` darf auf beiden Kernen schreiben; der Kern schützt die USB-Verbindung mit einer Sperre. Die Zeilen können sich aber mischen.
- `tone` ist ebenfalls geschützt und darf von beiden Kernen kommen.

Jetzt das Beispiel aus der Einleitung: Kern 0 lässt dich mit dem Joystick Sterne einsammeln, Kern 1 spielt dazu eine Melodie. Fängst du einen Stern, schickt Kern 0 über die FIFO die Nachricht `EFFEKT_STERN`, und Kern 1 spielt einen kurzen Klang. Je mehr Punkte, desto schneller die Musik: Die Punktzahl ist eine gemeinsame `volatile`-Variable.



Der Teil für Kern 0 ist ein gewöhnliches Spiel mit dem Bildpuffer aus Kapitel 14:

k24_musikspiel.ino

```
42 void loop() {
43     if (digitalRead(TASTE_LINKS) == LOW && x > 30) {
44         x -= 3;
45     }
46     if (digitalRead(TASTE_RECHTS) == LOW && x < 210) {
47         x += 3;
48     }
49     if (digitalRead(TASTE_HOCH) == LOW && y > 40) {
50         y -= 3;
51     }
52     if (digitalRead(TASTE_RUNTER) == LOW && y < 250) {
53         y += 3;
54     }
55     if (abs(x - stern_x) < 14 && abs(y - stern_y) < 14) {
56         punkte++;
```

```
57          rp2040.fifo.push_nb(EFFEKT_STERN);  // Kern 1 spielt den Effekt
58          neuer_stern();
59      }
60
61      bild.fillScreen(ST77XX_BLACK);
62      bild.fillCircle(stern_x, stern_y, 6, ST77XX_YELLOW);
63      bild.fillRect(x - 8, y - 8, 16, 16, ST77XX_CYAN);
64      bild.setTextColor(ST77XX_WHITE);
65      bild.setTextSize(2);
66      bild.setCursor(60, 12);
67      bild.printf("Punkte: %d", punkte);
68      tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
69  }
```

Zeile 57 schickt die Nachricht mit `push_nb`. Das Spiel soll niemals warten, auch nicht, wenn die Warteschlange einmal voll wäre; dann fiele eben ein Klang aus. Und der Teil für Kern 1:

k24_musikspiel.ino

```
82  void loop1() {
83      for (int i = 0; i < ANZAHL_NOTEN; i++) {
84          int viertel = 300 - punkte * 10;    // je mehr Punkte, desto schneller
85          if (viertel < 120) {
86              viertel = 120;
87          }
88          int dauer = LAENGE[i] * viertel;
89          if (MELODIE[i] > 0) {
90              tone(LAUTSPRECHER, MELODIE[i], dauer * 9 / 10);
91          }
92          warten_und_horchen(dauer);
93      }
94  }
95
96  // wartet ms Millisekunden und spielt dabei Effekte, die Kern 0 schickt
97  void warten_und_horchen(int ms) {
98      unsigned long start = millis();
99      while (millis() - start < (unsigned long)ms) {
100         uint32_t nachricht;
101         if (rp2040.fifo.pop_nb(&nachricht) && nachricht == EFFEKT_STERN) {
102             tone(LAUTSPRECHER, 1568, 60);
103             delay(60);
104             tone(LAUTSPRECHER, 2093, 100);
105             delay(100);
106         }
107     }
108 }
```

- `loop1()` spielt die Melodie Note für Note. Die Länge einer Viertelnote hängt von `punkte` ab (Zeile 84), das Kern 0 hochzählt.
- Statt `delay(dauer)` steht `warten_und_horchen(dauer)`. Diese Funktion wartet genauso lange, sieht dabei aber ständig mit `pop_nb` nach, ob Kern 0 etwas geschickt hat. Kommt `EFFEKT_STERN`, spielt sie zwei hohe Töne. Der neue `tone`-Befehl ersetzt dabei die Melodienote, denn der Lautsprecher kann nur einen Ton zugleich.

- Kern 1 darf hier ruhig mit `delay` warten: Das Spiel auf Kern 0 läuft davon unberührt weiter.

Den ganzen Sketch findest du als `k24_musikspiel`. Probiere aus, was ohne den zweiten Kern passieren würde: Mit `delay` für die Noten in `loop()` würde das Spiel bei jeder Note stehen bleiben.

Ausprobieren: Ein Reaktionstest mit Interrupt

In Kapitel 9 hast du einen Reaktionstest gebaut, der in Warteschleifen ständig die Taste abfragt. Mit einem Interrupt wird er einfacher: Die ISR merkt sich den **ersten** Druck und seinen Zeitpunkt. Das Prellen stört dabei nicht, denn alle weiteren Flanken ignoriert sie, bis `loop()` die Flagge wieder zurücksetzt.

k24_reaktion.ino

```
// Reaktionstest mit Interrupt: die ISR merkt sich den Zeitpunkt des Drucks
#include <Adafruit_ST7789.h>

#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26
#define TASTE_A 27

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);

volatile bool gedrueckt = false;
volatile unsigned long druck_zeit = 0;

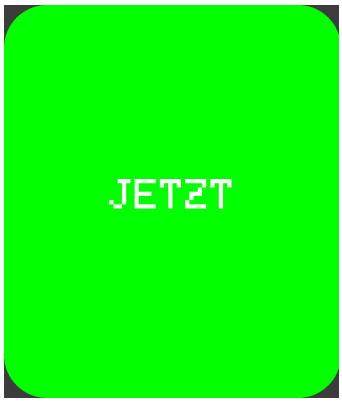
void taste_a_gedrueckt() {
    if (!gedrueckt) {               // nur der erste Druck zaehlt
        druck_zeit = millis();
        gedrueckt = true;
    }
}

void setup() {
    Serial.begin(115200);
    tft.init(240, 280);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
    pinMode(TASTE_A, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(TASTE_A), taste_a_gedrueckt, FALLING);

    zeige(ST77XX_BLACK, 21, "Start mit A");
    while (!gedrueckt) {
        delay(10);                  // warten, bis die ISR den Druck meldet
    }
    randomSeed(druck_zeit);         // dein Druck kommt nie genau gleich
```

```
35      delay(500);
36  }
37
38  void loop() {
39      zeige(ST77XX_RED, 48, "Warte...");
40      gedrueckt = false;
41      delay(random(1500, 4000));      // die ISR passt in dieser Zeit auf
42
43      if (gedrueckt) {
44          zeige(ST77XX_ORANGE, 39, "Zu frueh!");
45          Serial.println("Zu frueh!");
46      } else {
47          unsigned long signal = millis();    // die Uhr startet mit dem Umfaerben
48          zeige(ST77XX_GREEN, 75, "JETZT");
49          while (!gedrueckt && millis() - signal < 2000) {
50              // warten auf den Druck, hoechstens 2 Sekunden
51          }
52          if (gedrueckt) {
53              unsigned long reaktion = druck_zeit - signal;
54              Serial.printf("Reaktionszeit: %lu ms\n", reaktion);
55              tft.fillScreen(ST77XX_BLACK);
56              tft.setCursor(66, 124);
57              tft.print(reaktion);
58              tft.print(" ms");
59          } else {
60              zeige(ST77XX_BLUE, 21, "Zu langsam!");
61              Serial.println("Zu langsam!");
62          }
63      }
64      delay(2500);
65  }
66
67  void zeige(uint16_t farbe, int x, const char *text) {
68      tft.fillScreen(farbe);
69      tft.setTextColor(ST77XX_WHITE);
70      tft.setTextSize(3);
71      tft.setCursor(x, 124);
72      tft.print(text);
73  }
```

- Zeile 41 wartet einfach mit `delay` eine zufällige Zeit. Drückt jemand zu früh, verpasst der Sketch das nicht: Die ISR hat es notiert, und Zeile 43 sieht es danach.
- Ab Zeile 47 läuft die Uhr. Die Schleife in Zeile 49 wartet auf die Flagge, höchstens zwei Sekunden.
- Zeile 53 rechnet mit dem Zeitpunkt, den die ISR gemessen hat, nicht mit dem Moment, in dem `loop()` den Druck bemerkt.
- `randomSeed(druck_zeit)` in Zeile 34: Wann genau du zum Start drückst, ist jedes Mal anders. Das macht die Wartezeiten zufällig (Kapitel 5.4).



Serieller Monitor

```
Reaktionszeit: 306 ms
Zu frueh!
```



Zum Verändern: Zeige nach fünf Runden den Durchschnitt deiner Reaktionszeiten. Lass bei „Zu frueh!“ die rote LED aufleuchten.

Übungsaufgaben

Aufgabe 1 ⭐ A und B zählen Erweitere `k24_interrupt` um eine zweite ISR für Taste B. Der Sketch gibt jede Sekunde aus, wie oft A und wie oft B gedrückt wurde.

Aufgabe 2 ⭐⭐ Was ist hier falsch? Diese ISR verstößt gegen mehrere Regeln aus 24.2. Finde alle Fehler und schreibe den Sketch so um, dass die ISR nur noch zählt und `loop()` den Rest erledigt.

k24_aufgabe2.ino

```ino
// Diese ISR hat gleich mehrere Fehler. Findest du sie?
#define TASTE_A 27
#define LAUTSPRECHER 15

int treffer = 0;

void taste_a_gedrueckt() {
    treffer++;
    Serial.printf("Treffer: %d\n", treffer);
    tone(LAUTSPRECHER, 440);
    delay(200);
    noTone(LAUTSPRECHER);
}

void setup() {
    Serial.begin(115200);
    pinMode(TASTE_A, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(TASTE_A), taste_a_gedrueckt,
      FALLING);
}

void loop() {
    if (treffer >= 10) {
        Serial.println("Gewonnen!");
```

```
        treffer = 0;
    }
}
```

Aufgabe 3 ⭐⭐ Duell Ein Reaktionstest für zwei: Nach dem Signal gewinnt, wer zuerst drückt, A oder B. Wer vor dem Signal drückt, hat verloren. Benutze für jede Taste eine ISR. Beide tragen in dieselbe Variable ein, wer zuerst war, aber nur, wenn dort noch niemand steht.

Aufgabe 4 ⭐⭐⭐ Lauflicht auf Kern 1 Kern 1 lässt die drei LEDs nacheinander aufleuchten (rot, gelb, grün, rot …). Kern 0 fragt den Joystick ab: hoch macht das Lauflicht schneller, runter langsamer. Wie kommt die Geschwindigkeit von Kern 0 zu Kern 1?

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- **Abfragen** verpasst kurze Ereignisse, wenn `loop()` lange beschäftigt ist. Ein **Interrupt** unterbricht das Programm sofort und ruft eine **ISR** auf.
- `attachInterrupt(digitalPinToInterrupt(pin), isr, FALLING);` – bei Tasten ist `FALLING` der Moment des Drückens. Beim RP2350 kann jeder Pin einen Interrupt auslösen.
- Variablen, die eine ISR und das übrige Programm gemeinsam benutzen, sind **`volatile`**.
- Eine ISR ist **kurz**: Flagge setzen, Zeit merken (`millis()`), fertig. Kein `delay`, kein `Serial`, kein `tone`, nicht zeichnen.
- Mehrere zusammengehörige Werte zwischen `noInterrupts()` und `interrupts()` abholen.
- Gegen Prellen hilft eine **Sperrzeit** in der ISR; für gewöhnliche Tasten ist Abfragen mit Entprellen oft einfacher.
- `setup1()` und `loop1()` laufen auf **Kern 1**, gleichzeitig mit `setup()` / `loop()` auf Kern 0.
- Gemeinsame Daten: `volatile`-Variablen oder Nachrichten mit `rp2040.fifo.push_nb(…)` / `pop_nb(…)`. Nur ein Kern zeichnet auf das Display.

KAPITEL 25

Daten dauerhaft speichern: LittleFS und EEPROM

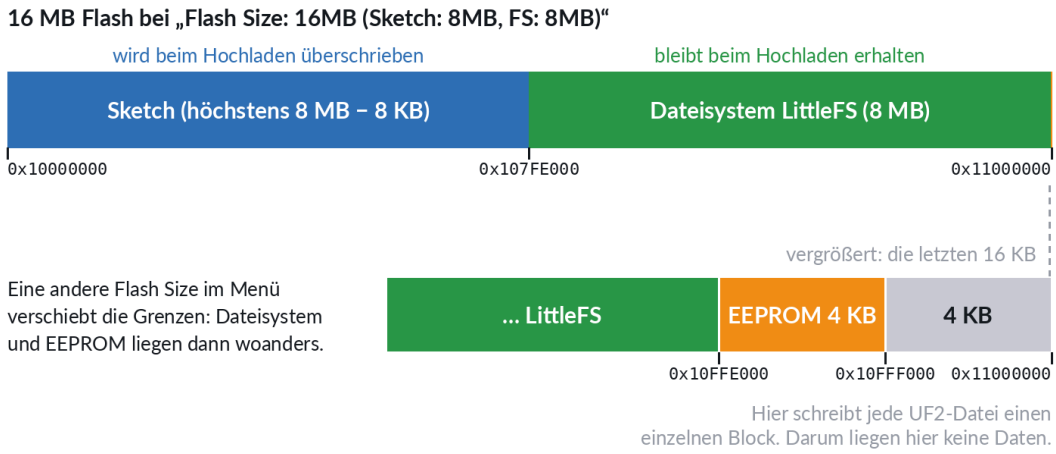
Worum geht's? Alles, was dein Sketch in Variablen hat, ist weg, sobald der PicoBoy ausgeht oder neu startet. Ein Highscore, eine Einstellung oder ein Messprotokoll soll aber bleiben. In diesem Kapitel speicherst du Daten im **Flash**, dem Speicher, in dem auch dein Sketch liegt: zuerst ganz einfach mit der **EEPROM-Nachbildung**, dann in richtigen **Dateien** mit **LittleFS**. Am Ende überlebt die Bestenliste eines Spiels jeden Neustart und jedes Hochladen.

25.1 RAM und Flash

Der RP2350 hat zwei Arten von Speicher (Kapitel 19):

- Der **Arbeitsspeicher** (RAM, 520 KB) ist schnell und lässt sich beliebig oft beschreiben. Hier stehen alle Variablen. Ohne Strom vergisst er alles.
- Der **Flash** (16 MB) behält seinen Inhalt ohne Strom, wie ein USB-Stick. Hier liegt dein Sketch. Beschreiben ist umständlich: Der Chip muss vorher einen ganzen Block von 4 KB **löschen**, und jeder Block verträgt nur eine begrenzte Zahl solcher Löschvorgänge.

Damit dein Sketch selbst Daten im Flash ablegen kann, teilt der Kern den Flash auf. Das hast du in Kapitel 2 schon eingestellt, ohne es zu brauchen: **Werkzeuge → Flash Size → 16MB (Sketch: 8MB, FS: 8MB)**. So sieht die Aufteilung dann aus (die Zahlen stehen in der Board-Beschreibung des Kerns):



Bereich	Größe	Wofür?
Sketch	bis 8 MB – 8 KB	dein Programm
Dateisystem (FS)	8 MB	Dateien mit **LittleFS** (Abschnitt 25.4)
EEPROM	4 KB	die **EEPROM-Nachbildung** (Abschnitt 25.2)

> Steht **Flash Size** auf einer Einstellung mit „no FS“, gibt es kein Dateisystem; `LittleFS.begin()` liefert dann `false`. Die Voreinstellung für „Generic RP2350“ ist **2MB (no FS)**! Prüfe also im Menü **Werkzeuge**, ob wirklich **16MB (Sketch: 8MB, FS: 8MB)** eingestellt ist. Mit einer anderen Flash Size liegen Dateisystem und EEPROM außerdem an anderer Stelle: Was du vorher gespeichert hast, findet der Sketch dann nicht mehr.

25.2 Die EEPROM-Nachbildung

Viele Arduino-Platinen haben einen kleinen Extraspeicher für Einstellungen, das **EEPROM** (*electrically erasable programmable read-only memory*). Der RP2350 hat keinen. Der Kern **bildet ihn nach**: mit einem Block von 4 KB am Ende des Flash und derselben Bibliothek `EEPROM`, die auch andere Platinen benutzen. So funktioniert sie:

1. `EEPROM.begin(größe)` holt eine Kopie des Blocks in den Arbeitsspeicher, `größe` Byte, von 256 bis 4096.
2. `EEPROM.get(adresse, variable)` liest ab Byte `adresse` so viele Byte, wie die Variable groß ist, und füllt sie damit. `EEPROM.put(adresse, variable)` schreibt eine Variable hinein. `EEPROM.read(adresse)` und `EEPROM.write(adresse, wert)` tun dasselbe für ein einzelnes Byte.
3. All das ändert nur die **Kopie im RAM**. Erst `EEPROM.commit()` löscht den Block im Flash und schreibt die Kopie hinein.

Der erste Sketch zählt, wie oft der PicoBoy gestartet wurde:

k25_startzaehler.ino

```
// Startzaehler: Wie oft wurde der PicoBoy mit diesem Sketch gestartet?
#include <Adafruit_ST7789.h>
#include <EEPROM.h>

#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26

#define KENNUNG 0x50424331      // "PBC1": daran erkennen wir unsere Daten

struct Speicher {
    uint32_t kennung;
    uint32_t starts;
};

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
Speicher daten;

void setup() {
    Serial.begin(115200);
    tft.init(240, 280);
```

```
23      pinMode(TFT_LICHT, OUTPUT);
24      digitalWrite(TFT_LICHT, HIGH);
25
26      EEPROM.begin(256);              // 256 Byte aus dem Flash in den RAM holen
27      EEPROM.get(0, daten);
28      if (daten.kennung != KENNUNG) { // noch nie gespeichert (oder fremde Daten)
29          daten.kennung = KENNUNG;
30          daten.starts = 0;
31      }
32      daten.starts++;
33      EEPROM.put(0, daten);
34      EEPROM.commit();                // erst jetzt wird der Flash beschrieben
35
36      tft.fillScreen(ST77XX_BLACK);
37      tft.setTextColor(ST77XX_WHITE);
38      tft.setTextSize(2);
39      tft.setCursor(48, 100);
40      tft.print("Start Nr.");
41      tft.setTextSize(5);
42      tft.setCursor(80, 140);
43      tft.print(daten.starts);
44      delay(2000);                    // Zeit, den seriellen Monitor zu oeffnen
45      Serial.printf("Start Nr. %lu\n", (unsigned long)daten.starts);
46  }
47
48  void loop() {
49  }
```

- Zeile 12 bis 15 bündeln, was gespeichert wird, in einer Struktur (Kapitel 13). `uint32_t` ist eine ganze Zahl mit genau 32 Bit, also 4 Byte; so ist die Größe auf jeden Fall gleich.
- Zeile 26 und 27 holen den Block in den RAM und lesen ab Adresse 0 die ganze Struktur.
- Zeile 32 bis 34 zählen hoch, legen die Struktur zurück in die Kopie und schreiben sie mit `commit()` in den Flash.

Starte den PicoBoy ein paar Mal neu (Taste **RES** oder Kabel abziehen): Die Zahl wächst. Im seriellen Monitor erscheint dazu `Start Nr. 3` und so weiter.

Wozu die Kennung in Zeile 28? Was in einem Flash-Block steht, den noch nie jemand beschrieben hat, ist nicht null. Gelöschter Flash enthält lauter Bytes mit dem Wert 255 (`0xFF`). Und hat vorher ein anderes Programm den Speicher benutzt, etwa MicroPython oder ein Spiel, stehen dort dessen Reste. Deshalb speichert der Sketch zuerst eine **Kennung**, eine Zahl, die er sich ausgedacht hat. Nur wenn sie stimmt, sind die Daten seine eigenen. Andernfalls fängt er bei null an.

> Gespeicherte Daten immer mit einer **Kennung** prüfen. Ändert sich der Aufbau der Struktur, nimm eine neue Kennung, damit alte Daten nicht falsch gelesen werden.

> Die Speicherfunktionen dieses Kapitels lassen sich in der Prüfung auf dem PC nicht nachbilden. Alle Sketche sind für den PicoBoy übersetzt und geprüft, laufen aber nur dort. Deshalb gibt es hier keine Bilder und Ausgaben aus der Prüfung.

25.3 Den Highscore speichern

Das Spiel aus Kapitel 18 merkt sich seinen Rekord in der Variablen `rekord` – aber nur im Arbeitsspeicher. Nach dem Ausschalten ist er weg. Hier ist ein kleines Spiel, an dem du sehen kannst, wie man ihn behält: Wie oft schaffst du es, Taste A in fünf Sekunden zu drücken?

k25_highscore.ino

```
// Tastenhammer: Wie oft schaffst du A in 5 Sekunden? Der Rekord bleibt im
   EEPROM.
#include <Adafruit_ST7789.h>
#include <EEPROM.h>

#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26
#define TASTE_A 27
#define TASTE_B 28
#define SPIELZEIT 5000
#define KENNUNG 0x48494753      // "HIGS"

struct Rekord {
    uint32_t kennung;
    int punkte;
};

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
Rekord rekord;

void setup() {
    Serial.begin(115200);
    tft.init(240, 280);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
    pinMode(TASTE_A, INPUT_PULLUP);
    pinMode(TASTE_B, INPUT_PULLUP);
    tft.setTextColor(ST77XX_WHITE);
    rekord_laden();
}

void loop() {
    tft.fillScreen(ST77XX_BLACK);
    tft.setTextSize(2);
    tft.setCursor(36, 90);
    tft.printf("Rekord: %d", rekord.punkte);
    tft.setCursor(36, 140);
```

```
39      tft.print("Start mit B");
40      while (digitalRead(TASTE_B) == HIGH) {
41          delay(10);
42      }
43
44      int punkte = spielen();
45
46      tft.fillScreen(ST77XX_BLACK);
47      tft.setTextSize(3);
48      tft.setCursor(36, 90);
49      tft.printf("%d Punkte", punkte);
50      if (punkte > rekord.punkte) {
51          rekord.punkte = punkte;
52          rekord_speichern();
53          tft.setTextColor(ST77XX_YELLOW);
54          tft.setCursor(36, 140);
55          tft.print("Rekord!");
56          tft.setTextColor(ST77XX_WHITE);
57      }
58      delay(3000);
59  }
60
61  int spielen() {
62      int punkte = 0;
63      bool vorher = false;
64      unsigned long start = millis();
65      tft.fillScreen(ST77XX_BLUE);
66      while (millis() - start < SPIELZEIT) {
67          bool jetzt = digitalRead(TASTE_A) == LOW;
68          if (jetzt && !vorher) {
69              punkte++;
70          }
71          vorher = jetzt;
72          delay(10);                  // einfaches Entprellen
73      }
74      return punkte;
75  }
76
77  void rekord_laden() {
78      EEPROM.begin(256);
79      EEPROM.get(0, rekord);
80      if (rekord.kennung != KENNUNG) {
81          rekord.kennung = KENNUNG;
82          rekord.punkte = 0;
83      }
84      Serial.printf("Rekord geladen: %d\n", rekord.punkte);
85  }
86
87  void rekord_speichern() {
88      EEPROM.put(0, rekord);
89      EEPROM.commit();
90      Serial.printf("Neuer Rekord gespeichert: %d\n", rekord.punkte);
91  }
```

Alles, was mit dem Speichern zu tun hat, steckt in zwei Funktionen:

- `rekord_laden()` (Zeile 77 bis 85) wird einmal in `setup()` aufgerufen.
- `rekord_speichern()` (Zeile 87 bis 91) wird nur aufgerufen, wenn es wirklich einen neuen Rekord gibt (Zeile 50).

Genau so baust du es in das Spiel aus Kapitel 18 ein: Übernimm die Struktur `Rekord` und die beiden Funktionen und rufe `rekord_laden()` am Ende von `setup()` auf. In `k18_spiel` wird der Rekord in `spiel_schritt()` im Block `if (leben <= 0)` gesetzt (`rekord = punkte;`). Direkt dahinter gehört der Aufruf von `rekord_speichern()` – so wird nur geschrieben, wenn es wirklich einen neuen Rekord gibt. Weil `rekord` dabei von einer Zahl zur Struktur wird (`Rekord rekord;`), schreibst du im Spiel überall `rekord.punkte`, wo bisher `rekord` stand.

> Schreibe den Flash **nur, wenn sich etwas geändert hat**, und nie in jedem Durchlauf von `loop()`. Jedes `commit()` löscht einen Block, und das verträgt der Flash nicht beliebig oft. `commit()` hilft ein wenig mit: Hat sich seit dem letzten Mal kein Byte der Kopie geändert, schreibt es gar nicht erst. Außerdem hält der Kern während des Schreibens kurz alle Interrupts und den zweiten Kern an (Kapitel 24).

25.4 Dateien mit LittleFS

Für mehr als ein paar Zahlen ist ein **Dateisystem** bequemer. Es verwaltet den Speicher in **Dateien** mit Namen, wie auf einem Computer, und es verteilt das Schreiben selbst über den ganzen Bereich, damit kein Block übermäßig oft gelöscht wird. Der Kern bringt dafür **LittleFS** mit, ein Dateisystem für Flash-Speicher. Du brauchst nur die Zeile `#include <LittleFS.h>`; installieren musst du nichts.

k25_littlefs.ino

```
1  // Erste Schritte mit LittleFS: eine Datei schreiben und wieder lesen
2  #include <LittleFS.h>
3
4  void setup() {
5      Serial.begin(115200);
6      delay(2000);                    // Zeit, den seriellen Monitor zu oeffnen
7
8      if (!LittleFS.begin()) {
9          Serial.println("Kein Dateisystem - Flash Size mit FS eingestellt?");
10         return;
11     }
12
13     File datei = LittleFS.open("/gruss.txt", "w");    // "w": neu schreiben
14     datei.println("Hallo aus dem Flash!");
15     datei.printf("Geschrieben nach %lu ms\n", millis());
16     datei.close();
17
18     datei = LittleFS.open("/gruss.txt", "r");         // "r": lesen
```

```
19      Serial.print("Groesse in Byte: ");
20      Serial.println(datei.size());
21      while (datei.available()) {
22          String zeile = datei.readStringUntil('\n');
23          Serial.println(zeile);
24      }
25      datei.close();
26  }
27
28  void loop() {
29  }
```

- **Zeile 8:** `LittleFS.begin()` bindet das Dateisystem ein. Findet es im Flash noch keines, **formatiert** es den Bereich (legt ein leeres Dateisystem an). Das passiert also beim allerersten Start von selbst. Ist keine Flash Size mit FS eingestellt, liefert `begin()` `false`.
- **Zeile 13:** `LittleFS.open(name, art)` öffnet eine Datei und gibt ein Objekt vom Typ `File` zurück. Dateinamen beginnen mit `/`.
- **Zeile 14 bis 15:** In eine Datei schreibst du mit denselben Befehlen wie in den seriellen Monitor: `print`, `println`, `printf`.
- **Zeile 16:** `close()` schließt die Datei. Erst dann ist sicher alles im Flash.
- **Zeile 18 bis 25:** lesen. `available()` sagt, ob noch etwas da ist, `readStringUntil('\n')` liest bis zum Zeilenende, wie beim seriellen Monitor in Kapitel 4. `size()` ist die Größe der Datei in Byte.

Die **Art** beim Öffnen:

Art	Bedeutung
`"r"`	lesen (*read*); gibt es die Datei nicht, ist das `File`-Objekt ungültig
`"w"`	schreiben (*write*); die Datei wird neu angelegt, ein alter Inhalt ist weg
`"a"`	anhängen (*append*); neue Zeilen kommen ans Ende, der alte Inhalt bleibt

Ob das Öffnen geklappt hat, prüfst du mit `if (datei)`. Das nutzt der nächste Sketch: ein **Logbuch**, das bei jedem Start eine Zeile mit der Batteriespannung anhängt und danach alle Dateien auflistet.

k25_logbuch.ino

```
1  // Logbuch: Jeder Start haengt eine Zeile an, dann zeigt der Sketch alle Dateien
2  #include <LittleFS.h>
3
4  #define BATTERIE A3
5  #define LOGBUCH "/logbuch.txt"
6
7  void setup() {
8      Serial.begin(115200);
9      delay(2000);
10     analogReadResolution(12);
```

```
11      if (!LittleFS.begin()) {
12          Serial.println("Kein Dateisystem - Flash Size mit FS eingestellt?");
13          return;
14      }
15
16      // Zeilen zaehlen (die Datei gibt es beim ersten Start noch nicht)
17      int zeilen = 0;
18      File datei = LittleFS.open(LOGBUCH, "r");
19      if (datei) {
20          while (datei.available()) {
21              datei.readStringUntil('\n');
22              zeilen++;
23          }
24          datei.close();
25      }
26
27      // eine Zeile anhaengen
28      float volt = analogRead(BATTERIE) * 3.3 / 4095 * 3;
29      datei = LittleFS.open(LOGBUCH, "a");              // "a": anhaengen
30      datei.printf("Start %d, Batterie %.2f V\n", zeilen + 1, volt);
31      datei.close();
32
33      // das ganze Logbuch ausgeben
34      Serial.println("--- Logbuch ---");
35      datei = LittleFS.open(LOGBUCH, "r");
36      while (datei.available()) {
37          Serial.write(datei.read());
38      }
39      datei.close();
40
41      // alle Dateien im Hauptordner und der belegte Platz
42      Serial.println("--- Dateien ---");
43      Dir ordner = LittleFS.openDir("/");
44      while (ordner.next()) {
45          Serial.printf("%-16s %6d Byte\n", ordner.fileName().c_str(),
            (int)ordner.fileSize());
46      }
47      FSInfo info;
48      LittleFS.info(info);
49      Serial.printf("Belegt: %d von %d KB\n", (int)(info.usedBytes / 1024),
50                    (int)(info.totalBytes / 1024));
51  }
52
53  void loop() {
54  }
```

- Zeile 18 bis 25 zählen die vorhandenen Zeilen. Beim ersten Start gibt es die Datei noch nicht; dann ist `datei` ungültig, und die Schleife wird übersprungen.
- Zeile 29 öffnet mit `"a"`: Die neue Zeile kommt ans Ende.
- Zeile 43 bis 46 gehen mit `openDir("/")` durch den Hauptordner. `ordner.next()` springt zum nächsten Eintrag und liefert `false`, wenn keiner mehr kommt. `fileName()` und `fileSize()` nennen Name und Größe.

- `LittleFS.info(info)` in Zeile 48 füllt eine Struktur mit Angaben über das Dateisystem, darunter `totalBytes` (Größe) und `usedBytes` (belegt).

Im seriellen Monitor siehst du nach dem dritten Start drei Zeilen im Logbuch, darunter die Dateien `gruss.txt` (vom vorigen Sketch) und `logbuch.txt`. Weitere Befehle: `LittleFS.exists(name)` prüft, ob es eine Datei gibt, `LittleFS.remove(name)` löscht sie, `LittleFS.rename(alt, neu)` benennt sie um.

> Damit du die ersten Zeilen im seriellen Monitor nicht verpasst, warten die Sketche am Anfang zwei Sekunden (`delay(2000)`). Öffne den Monitor, dann drücke **RES**.

25.5 Eine Bestenliste

EEPROM oder LittleFS, was nimmst du wann?

	EEPROM-Nachbildung	LittleFS
Platz	höchstens 4 KB	8 MB
Aufbau	Bytes an festen Adressen	Dateien mit Namen
gut für	ein paar Zahlen: Rekord, Einstellungen	Listen, Texte, Messprotokolle
Schreiben	jedes `commit()` löscht denselben Block	verteilt sich über den Flash

Eine **Bestenliste** mit den fünf besten Ergebnissen passt gut in eine Textdatei: eine Zahl pro Zeile, die beste zuerst. Das Spiel ist dasselbe wie in 25.3; neu sind die Funktionen für die Liste:

k25_bestenliste.ino

```
46 void liste_laden() {
47     for (int i = 0; i < PLAETZE; i++) {
48         liste[i] = 0;
49     }
50     File datei = LittleFS.open(DATEI, "r");
51     if (!datei) {
52         return;                    // noch keine Bestenliste
53     }
54     for (int i = 0; i < PLAETZE && datei.available(); i++) {
55         liste[i] = datei.readStringUntil('\n').toInt();
56     }
57     datei.close();
58 }
59
60 void liste_speichern() {
61     File datei = LittleFS.open(DATEI, "w");
62     for (int i = 0; i < PLAETZE; i++) {
63         datei.println(liste[i]);
64     }
65     datei.close();
66 }
```

```
68 // sortiert die Punkte ein; gibt den Platz (0 bis 4) zurueck oder -1
69 int eintragen(int punkte) {
70     for (int i = 0; i < PLAETZE; i++) {
71         if (punkte > liste[i]) {
72             for (int j = PLAETZE - 1; j > i; j--) {
73                 liste[j] = liste[j - 1];    // die Schlechteren ruecken nach
                      unten
74             }
75             liste[i] = punkte;
76             return i;
77         }
78     }
79     return -1;
80 }
```

- `liste_laden()` setzt erst alle Plätze auf 0. Gibt es die Datei, liest es Zeile für Zeile und macht mit `toInt()` aus dem Text eine Zahl (Kapitel 12).
- `eintragen()` sucht den ersten Platz, auf dem ein schlechteres Ergebnis steht, schiebt alle darunter um einen Platz nach unten (das letzte fällt heraus) und setzt das neue ein. Ist das Ergebnis für die Liste zu schlecht, gibt es `-1` zurück.
- `liste_speichern()` öffnet mit `"w"` und schreibt die ganze Liste neu. Sie wird nur aufgerufen, wenn sich die Liste geändert hat.

Den ganzen Sketch findest du als `k25_bestenliste`. Der neue Eintrag erscheint auf dem Display gelb.

25.6 Was die Daten löscht – und was nicht

Hochladen löscht sie nicht. Die Beschreibung des Kerns sagt dazu: Ein neuer Sketch verändert weder das Dateisystem noch die EEPROM-Daten. Das passt zur UF2-Datei, die beim Hochladen auf den PicoBoy kommt: Sie enthält nur den Bereich des Sketches und einen einzelnen Block in den letzten 4 KB des Flash, die genau deshalb frei bleiben (Abbildung in 25.1). Deine Bestenliste überlebt also, wenn du das Spiel verbesserst und neu hochlädst.

Gelöscht oder unauffindbar werden die Daten, wenn …

- dein Sketch `LittleFS.format()` aufruft oder Dateien mit `remove` löscht,
- du eine andere **Flash Size** wählst: Dann liegen Dateisystem und EEPROM woanders. Der Sketch findet sie nicht mehr, und `LittleFS.begin()` legt an der neuen Stelle ein leeres Dateisystem an,
- ein anderes Programm den Flash benutzt, etwa MicroPython mit seinem eigenen Dateisystem oder ein großes Spiel. Danach sind die Daten meist kaputt; die Kennung schützt deinen Sketch davor, Unsinn zu lesen.

> Willst du die Daten absichtlich löschen, etwa um einen Rekord zurückzusetzen, lass das den Sketch selbst tun, zum Beispiel wenn beim Start Taste B gedrückt ist (Aufgabe 1).

Ausprobieren: Ein Messprotokoll

Hänge an das Logbuch aus 25.4 alle zehn Sekunden eine Zeile mit der Batteriespannung an und lass den PicoBoy eine Stunde mit dem Schalter auf BAT ohne USB-Kabel laufen. Danach schließt du ihn an, öffnest den seriellen Monitor und lässt dir die Datei ausgeben. Wie schnell sinkt die Spannung? Aufgabe 3 zeigt, wie man das Protokoll zusätzlich auf Tastendruck ausgibt und löscht. Die Zahlen aus dem seriellen Monitor kannst du in eine Tabellenkalkulation kopieren und als Diagramm zeichnen lassen: Die Trennung mit `;` versteht sie als Spalten.

Übungsaufgaben

Aufgabe 1 ⭐ **Zurück auf null** Ändere den Startzähler: Ist beim Einschalten Taste B gedrückt, beginnt die Zählung wieder bei 1.

Aufgabe 2 ⭐⭐ **Lieblingsfarbe** Mit dem Joystick (links/rechts) wählst du eine von sechs Farben für die RGB-LED. Taste A speichert die Wahl im EEPROM. Nach dem nächsten Start leuchtet die LED gleich in dieser Farbe.

Aufgabe 3 ⭐⭐ **Batterie-Protokoll** Alle zehn Sekunden hängt der Sketch eine Zeile `sekunden;volt` an die Datei `/batterie.csv` an. Taste A gibt die Datei im seriellen Monitor aus, Taste B löscht sie.

Aufgabe 4 ⭐⭐⭐ **Rekord mit Namen** Erweitere `k25_highscore`: Wer einen neuen Rekord aufstellt, gibt mit dem Joystick ein Kürzel aus drei Buchstaben ein (hoch/runter ändert den Buchstaben, rechts geht zum nächsten, A ist fertig). Kürzel und Punkte werden zusammen gespeichert und auf dem Startbildschirm angezeigt. Denke an die Kennung!

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- RAM vergisst ohne Strom alles, der **Flash** nicht. Flash wird in Blöcken von 4 KB gelöscht und verträgt das nur begrenzt oft.
- **Flash Size 16MB (Sketch: 8MB, FS: 8MB)** einstellen; mit „no FS“ gibt es kein Dateisystem.
- **EEPROM-Nachbildung:** `EEPROM.begin(256)`, `get` / `put`, dann **`commit()`** – erst das schreibt in den Flash.
- Gespeicherte Daten mit einer **Kennung** prüfen.
- **LittleFS:** `#include <LittleFS.h>`, `LittleFS.begin()`, `File datei = LittleFS.open("/name.txt", "r" | "w" | "a")`, lesen und schreiben wie mit `Serial`, `close()`.
- Nur schreiben, wenn sich etwas geändert hat.
- **Hochladen löscht die Daten nicht**; eine andere Flash Size, `format()` oder ein anderes Programm im Flash schon.

KAPITEL 26

Dein eigenes Projekt

Worum geht's? Jetzt baust du einen Sketch nach deiner eigenen Idee. In diesem Kapitel geht es weniger um neue Befehle als um die Arbeitsweise: eine Idee finden und planen, in kleinen Stufen bauen, Fehler systematisch finden und das Ergebnis vorstellen und weitergeben. Die Beispiele sind klein, die Arbeitsweise funktioniert aber auch für große Projekte.

26.1 Eine Idee finden und planen

Gute Projekte sind **klein genug, um fertig zu werden**, und lassen sich später erweitern. Hier sind Ideen zum Anfangen, sortiert nach Aufwand:

Bereich	eher leicht	mittel	anspruchsvoll
Spiele	Münzwurf, Zahlenraten	Reaktionsduell (Kapitel 24), Hindernislauf mit Sprites	Snake, Labyrinth, Breakout
Werkzeuge	Würfel, Stoppuhr	Zufallsnamen für die Klasse, Vokabeltrainer mit Bestenliste	Taschenrechner mit dem Joystick
Musik und Licht	Ampel, Metronom	Mini-Klavier, Lichtorgel mit der RGB-LED	Melodie-Rekorder mit Datei (Kapitel 25)
Grafik	Muster-Generator	Uhr mit Zeigern, Bildbetrachter	Malprogramm mit dem Joystick
Messen	Batterieanzeige	Wasserwaage, Schüttel-Würfel (Kapitel 17)	Schrittzähler, Messprotokoll

Hast du eine Idee, beantworte vor dem Programmieren drei Fragen:

1. **Was soll der Sketch tun?** In ein, zwei Sätzen. Brauchst du dafür eine halbe Seite, ist die Idee zu groß.
2. **Wie bedient man ihn?** Welche Taste macht was? Was steht auf dem Display, was im seriellen Monitor? Leuchtet oder klingt etwas?
3. **Was ist die kleinste Fassung, die schon funktioniert?** Alles andere sind Extras für später.

Danach teilst du die Arbeit in **Stufen**. Jede Stufe endet mit einem Sketch, der läuft. Für einen elektronischen Würfel könnte der Plan so aussehen:

Stufe	Was kommt dazu?	Fertig, wenn …
1	Zufallszahl von 1 bis 6 im seriellen Monitor	nur Zahlen von 1 bis 6 erscheinen
2	Würfeln mit Taste A, Zahl auf dem Display	jeder Druck auf A genau eine neue Zahl zeigt
3	Würfelaugen statt Ziffer	alle sechs Seiten richtig aussehen
4 (Extra)	Statistik: Wie oft kam welche Zahl?	Taste B die Häufigkeiten ausgibt

> Zeichne vorher auf Karopapier, wie der Bildschirm aussehen soll. Nimm ein Kästchen für 10 Pixel, dann ist der Bildschirm 24 × 28 Kästchen groß. So liest du die Koordinaten ab, statt sie auszuprobieren. Denk an die runden Ecken.

26.2 In kleinen Stufen bauen und testen

Stufe 1 ist winzig. Trotzdem lohnt sich der Test: Stimmt der Bereich von `random`?

k26_wuerfel1.ino

```
// Wuerfel, Stufe 1: zwanzig Wuerfe im seriellen Monitor
void setup() {
    Serial.begin(115200);
    for (int i = 0; i < 20; i++) {
        Serial.print(random(1, 7));     // 1 bis 6, die 7 ist nicht dabei
        Serial.print(" ");
    }
    Serial.println();
}

void loop() {
}
```

Serieller Monitor

```
6 4 3 6 1 1 5 1 3 2 4 6 4 6 1 6 1 1 6 3
```

Kommen alle Zahlen von 1 bis 6 vor, aber nie 0 oder 7? `random(1, 7)` liefert Zahlen **ab** 1 und **unter** 7 (Kapitel 5.4). Starte den Sketch zweimal: Ohne `randomSeed` kommt jedes Mal dieselbe Folge. Das ist beim Testen sogar praktisch.

Stufe 2 würfelt bei jedem Druck auf Taste A:

k26_wuerfel2.ino

```
// Wuerfel, Stufe 2: Taste A wuerfelt, die Zahl steht gross auf dem Display
#include <Adafruit_ST7789.h>

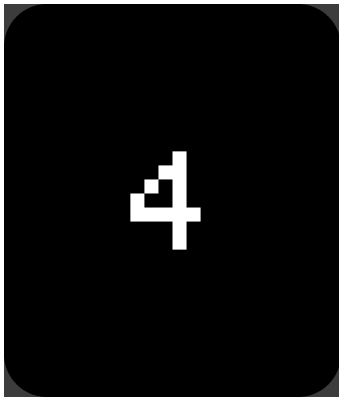
#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26
#define TASTE_A 27

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
bool vorher = false;

void setup() {
    Serial.begin(115200);
    tft.init(240, 280);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
```

```
18      pinMode(TASTE_A, INPUT_PULLUP);
19      tft.fillScreen(ST77XX_BLACK);
20      tft.setTextColor(ST77XX_WHITE, ST77XX_BLACK);   // mit Hintergrund: uebermalt
          die alte Zahl
21      tft.setTextSize(10);
22  }
23
24  void loop() {
25      bool jetzt = digitalRead(TASTE_A) == LOW;
26      if (jetzt && !vorher) {
27          int zahl = random(1, 7);
28          Serial.printf("Gewuerfelt: %d\n", zahl);
29          tft.setCursor(90, 105);
30          tft.print(zahl);
31      }
32      vorher = jetzt;
33      delay(20);
34  }
```

- Die Flankenerkennung aus Kapitel 9 (Zeile 25, 26 und 32) sorgt dafür, dass ein Druck genau einmal würfelt.
- Zeile 28 schreibt jeden Wurf zusätzlich in den seriellen Monitor. So siehst du auch dann, was passiert, wenn auf dem Display etwas nicht stimmt.
- `setTextColor` mit **zwei** Farben (Zeile 20) malt hinter jedes Zeichen den Hintergrund. Die neue Ziffer übermalt so die alte.



Stufe 3 ist Aufgabe 3. Den schwierigsten Teil, die Augen zu zeichnen, steckst du am besten in eine eigene Funktion `zeichne_wuerfel(zahl)`.

Jeden Fall testen, nicht nur zufällige. Mit Taste A würfelst du zufällig. Ob wirklich alle sechs Seiten stimmen, siehst du so nur mit Glück. Ruf deine Funktion deshalb zum Testen einmal mit 1 bis 6 nacheinander auf, mit einer Sekunde Pause dazwischen.

> **Nach jeder Stufe testen.** Geht plötzlich etwas nicht mehr, liegt der Fehler fast immer im Stück, das du zuletzt geschrieben hast. Speichere jede funktionierende Stufe unter eigenem Namen: **Datei → Speichern unter…** (`wuerfel_1`, `wuerfel_2` …). Dann kannst du jederzeit zurück.

26.3 Fehler finden

Fehler gibt es in drei Sorten:

Sorte	Woran erkennst du sie?	Was hilft?
Der Compiler versteht den Sketch nicht	`error:` beim **Überprüfen**	Meldung lesen, ersten Fehler zuerst (Kapitel 3.6, Anhang B)
Verdächtige Stelle	`warning:` beim Überprüfen	Warnung ernst nehmen und beheben
Logikfehler	keine Meldung, aber der Sketch tut das Falsche	nachsehen, was er wirklich tut

Ein Absturz zur Laufzeit zeigt auf dem PicoBoy keine Meldung. Der Sketch bleibt hängen oder startet neu. Auch das ist für dich ein Logikfehler.

Mit `Serial.printf` nachsehen. Ein Logikfehler meldet sich nicht. Dieser Sketch soll den Durchschnitt von vier Sprungweiten ausrechnen:

k26_fehler_mittelwert.ino

```
// Durchschnitt von vier Sprungweiten - mit einem Fehler
float weiten[4] = { 4.10, 3.85, 4.32, 4.03 };

void setup() {
    Serial.begin(115200);
    float summe = 0;
    for (int i = 0; i < 4; i++) {
        summe = weiten[i];
    }
    Serial.printf("Durchschnitt: %.2f m\n", summe / 4);
}

void loop() {
}
```

Serieller Monitor

```
Durchschnitt: 1.01 m
```

Ein Durchschnitt um 1 m bei Weiten um 4 m kann nicht stimmen. Wo geht es schief? Eine zusätzliche Ausgabe in der Schleife zeigt, welche Werte die Variablen in jedem Durchlauf haben:

k26_mittelwert_suche.ino

```
 7      for (int i = 0; i < 4; i++) {
 8          summe = weiten[i];
 9          Serial.printf("i: %d  weite: %.2f  summe: %.2f\n", i, weiten[i], summe);
10      }
```

Serieller Monitor

```
i: 0  weite: 4.10  summe: 4.10
i: 1  weite: 3.85  summe: 3.85
i: 2  weite: 4.32  summe: 4.32
i: 3  weite: 4.03  summe: 4.03
Durchschnitt: 1.01 m
```

Jetzt sieht man es sofort: `summe` ist immer nur die aktuelle Weite, statt mitzuwachsen. In Zeile 8 muss `summe += weiten[i];` stehen. Danach kommt die zusätzliche Zeile wieder heraus:

k26_mittelwert.ino

```
 7      for (int i = 0; i < 4; i++) {
 8          summe += weiten[i];
 9      }
10      Serial.printf("Durchschnitt: %.2f m\n", summe / 4);
```

Serieller Monitor

```
Durchschnitt: 4.08 m
```

> Schreib in die Ausgabe immer den **Namen** der Variable dazu (`summe: %.2f`). Sonst stehen im seriellen Monitor nur Zahlen, und du weißt nicht, welche zu welcher Variable gehört. Ohne seriellen Monitor helfen die LEDs: Lass zum Beispiel die gelbe LED aufleuchten, wenn der Sketch eine bestimmte Stelle erreicht.

Sketch verkleinern. Manchmal versteht man einen Fehler nicht, weil im Sketch zu viel anderes passiert. Dann hilft es, eine **Kopie** so lange zu verkleinern, bis nur noch das übrig ist, was den Fehler auslöst. Nach jedem Löschen testest du: Kommt der Fehler noch? Ein verkleinerter Sketch eignet sich auch gut, um jemand anderen um Hilfe zu bitten, auch eine KI (Kapitel 27).

Wenn du feststeckst: Erkläre deinen Sketch Zeile für Zeile laut, einer Mitschülerin, einem Kollegen oder einem Kuscheltier. Beim Erklären merkt man oft selbst, wo man etwas anderes geschrieben als gemeint hat.

26.4 Vorstellen und weitergeben

Bevor du dein Projekt vorstellst, räumst du den Sketch auf, damit andere ihn lesen können. Die Ordnung ist immer dieselbe: Kopfkommentar, `#include`, Pins, Konstanten, Objekte und Variablen, `setup()`, `loop()`, eigene Funktionen. Diese Vorlage zeigt das Gerüst, mit zwei Zuständen wie in Kapitel 18:

k26_vorlage.ino

```
// Projekt: Mein Spiel
// Von: Vorname Name, Datum
// Bedienung: A startet, Joystick steuert, B zurueck zum Titel
// Board: Generic RP2350, Flash Size: 16MB (Sketch: 8MB, FS: 8MB)
// Bibliotheken: Adafruit ST7735 and ST7789
#include <Adafruit_ST7789.h>

#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26
#define TASTE_A 27
#define TASTE_B 28

#define TITEL 0
#define SPIEL 1

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
GFXcanvas16 bild(240, 280);
int zustand = TITEL;

void setup() {
    Serial.begin(115200);
    tft.init(240, 280);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
    pinMode(TASTE_A, INPUT_PULLUP);
    pinMode(TASTE_B, INPUT_PULLUP);
}

void loop() {
    // 1. Tasten lesen und rechnen
    if (zustand == TITEL && digitalRead(TASTE_A) == LOW) {
        zustand = SPIEL;
    }
```

Der Kopfkommentar (Zeile 1 bis 5) sagt, was der Sketch ist, wer ihn geschrieben hat und wie man ihn bedient. Er nennt auch **Board-Einstellung und Bibliotheken**: Wer deinen Sketch bekommt, weiß dann, was er installieren und einstellen muss. Den ganzen Sketch findest du als `k26_vorlage`.



Für die Vorstellung (etwa 3 bis 5 Minuten) hat sich diese Reihenfolge bewährt:

1. **Was ist es?** Ein Satz zur Idee.
2. **Vorführen.** Auf dem PicoBoy, eventuell mit einer Kamera auf den Beamer übertragen.
3. **Ein Stück Code erklären**, auf das du stolz bist, zum Beispiel eine Funktion.
4. **Was war schwierig?** Welcher Fehler hat lange gedauert, und wie hast du ihn gefunden?
5. **Was kommt als Nächstes?** Welche Erweiterung würdest du noch bauen?

Weitergeben. Einen Sketch gibst du als ganzen Ordner weiter (mit allen `.h`-Dateien darin), am einfachsten als ZIP-Datei. Wer keine Arduino IDE hat, kann trotzdem dein fertiges Programm aufspielen: als **UF2-Datei**. **Sketch → Kompilierte Binärdatei exportieren** legt im Sketch-Ordner einen Unterordner `build` an, darin unter anderem eine Datei mit der Endung `.uf2`. Die kopiert man im Bootmodus (BOOT halten, RES tippen) einfach auf das Laufwerk des PicoBoy, wie in Kapitel 1.5.

Checkliste vor dem Vorstellen:

☐ Der Sketch übersetzt ohne Fehler und ohne Warnungen.
☐ Ganz oben steht ein Kopfkommentar mit Name, Zweck, Bedienung, Board und Bibliotheken.
☐ Variablen und Funktionen haben Namen, die sagen, was sie bedeuten.
☐ Wiederholte Teile stecken in Funktionen.
☐ Feste Zahlen, die man ändern möchte, stehen als `#define` oben.
☐ Ausgaben, die nur zur Fehlersuche da waren, sind entfernt.
☐ Auf dem Display stehen keine Umlaute („Tschuess“ statt „Tschüss“).
☐ Nichts Wichtiges liegt in den runden Ecken.

Übungsaufgaben

> **Aufgabe 1 ⭐ Ein Plan für die Stoppuhr** Plane eine Stoppuhr: Taste A startet und stoppt, Taste B setzt auf 0 zurück, die Zeit steht in Sekunden mit einer Nachkommastelle auf dem Display. Beantworte die drei Fragen aus 26.1 und schreibe einen Plan mit 3 bis 4 Stufen. Programmieren musst du (noch) nicht.

Aufgabe 2 ⭐⭐ **Fehlersuche** Der Sketch soll zählen, an wie vielen Tagen der Woche es wärmer als 20 Grad war. Es waren drei Tage, er gibt aber `Warme Tage: 2` aus. Finde den Fehler mit einer zusätzlichen Ausgabe in der Schleife und korrigiere ihn.

k26_aufgabe2.ino

```
// Wie viele Tage der Woche waren waermer als 20 Grad? (Das Ergebnis stimmt
  nicht.)
int temperatur[7] = { 22, 18, 19, 24, 21, 17, 20 };

void setup() {
    Serial.begin(115200);
    int warm = 0;
    for (int i = 1; i < 7; i++) {
        if (temperatur[i] > 20) {
            warm++;
        }
    }
    Serial.printf("Warme Tage: %d\n", warm);
}

void loop() {
}
```

Aufgabe 3 ⭐⭐ **Würfelaugen** Baue Stufe 3 des Würfels: Statt der Ziffer zeigt das Display ein weißes Quadrat mit abgerundeten Ecken und schwarzen Augen. Tipp: Die Mitte hat ein Auge bei 1, 3 und 5. Die Ecken links oben und rechts unten haben eins ab 2, die beiden anderen Ecken ab 4, und die Mitte links und rechts nur bei 6.

Aufgabe 4 ⭐⭐⭐ Die Stoppuhr bauen Setze deinen Plan aus Aufgabe 1 um, Stufe für Stufe. Läuft die Uhr, ist die Zeit grün, steht sie, weiß. Taste B setzt nur zurück, wenn die Uhr steht.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Eine Idee in ein, zwei Sätzen beschreiben, die kleinste funktionierende Fassung festlegen, dann in **Stufen** planen.
- Nach jeder Stufe testen. Funktionierende Stufen unter eigenem Namen speichern.
- Funktionen mit **allen** Fällen testen, nicht nur mit zufälligen.
- Logikfehler mit `Serial.printf("name: %d\n", name)` sichtbar machen, danach die Zeile wieder entfernen.
- Unverständliche Fehler: eine Kopie des Sketches verkleinern, bis nur der Fehler übrig bleibt.
- Vor dem Vorstellen aufräumen: Kopfkommentar mit Board und Bibliotheken, gute Namen, Funktionen, Konstanten.
- Weitergeben als Sketch-Ordner oder als `.uf2`-Datei (**Kompilierte Binärdatei exportieren**).

KAPITEL 27

Programmieren mit KI: Vibecoding

Worum geht's? KI-Systeme wie ChatGPT, Claude oder Gemini schreiben auf Zuruf ganze Sketche. Das ist verlockend und oft hilfreich, aber für den PicoBoy liegen sie auch oft daneben. In diesem Kapitel lernst du, wie du eine KI gut fragst, wie du einem KI-Agenten mit einer **Regeldatei** die Besonderheiten des PicoBoy beibringst und vor allem, wie du prüfst, was die KI geschrieben hat. Alles, was du in diesem Buch gelernt hast, brauchst du dafür.

27.1 Was ist Vibecoding?

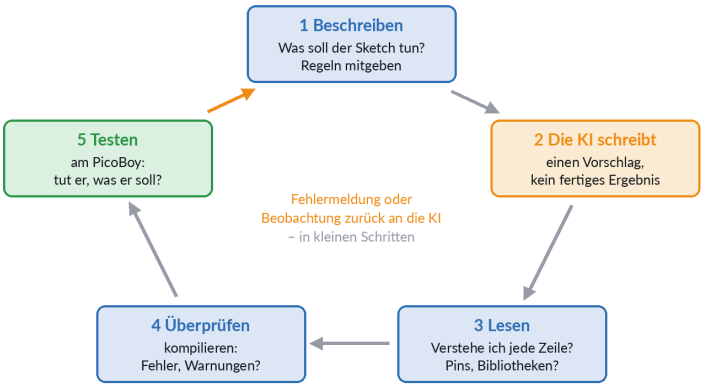
Vibecoding heißt: Man beschreibt einer KI in Alltagssprache, was ein Programm tun soll, und lässt sie den Code schreiben. Man liest ihn kaum, probiert aus und beschreibt, was noch nicht passt. Der Begriff kam 2025 auf und ist eher scherzhaft gemeint: programmieren nach Gefühl (*vibe*).

Die KI dahinter ist ein **Sprachmodell**: ein Programm, das aus riesigen Mengen Text gelernt hat, welche Wörter und Zeichen typischerweise aufeinander folgen. Darunter ist sehr viel Programmcode, auch für Arduino. Daraus folgt, was sie kann und was nicht:

Das kann eine KI gut	Da liegt sie oft daneben
ein Gerüst für eine Idee schreiben	Besonderheiten **deines** Geräts: Pins, Bibliotheken, Board
Fehlermeldungen erklären	Funktionen, die es gar nicht gibt, aber geben könnte
Code erklären, Zeile für Zeile	Code für die häufigste Platine (Arduino Uno) statt für den PicoBoy
Ideen für Erweiterungen	ob der Sketch auf dem Gerät wirklich läuft: das kann sie nicht sehen

Das Gefährliche: Eine KI klingt **immer** überzeugt, auch wenn sie sich etwas ausdenkt. Sie schreibt einen Sketch für den PicoBoy, der aussieht wie richtiger Code, und benutzt dabei die Pins eines ganz anderen Boards.

> Was die KI schreibt, ist ein **Vorschlag**. Du bist verantwortlich für deinen Sketch: Du liest ihn, verstehst ihn, überprüfst ihn und testest ihn am Gerät.



27.2 Chat-Systeme: gut fragen

Chat-Systeme wie ChatGPT (von OpenAI), Claude (von Anthropic) oder Gemini (von Google) benutzt du im Browser oder in einer App. Du schreibst eine Frage oder einen Auftrag, den **Prompt**, und bekommst eine Antwort. Den Code kopierst du in die Arduino IDE. Wie man ein Konto anlegt und was es kostet, steht auf den Seiten der Anbieter; das ändert sich laufend.

Je genauer dein Prompt, desto besser die Antwort. Ein schlechter Prompt:

```
Schreib ein Spiel für den PicoBoy.
```

Die KI weiß nicht, welcher PicoBoy, welche Programmiersprache, welche Bibliotheken. Sie rät. Ein guter Prompt nennt das **Ziel**, die **Hardware** und die **Regeln**:

```
Schreibe einen Arduino-Sketch für den PicoBoy Color Plus (RP2350,
Kern arduino-pico von Earle Philhower, Board "Generic RP2350").
Display ST7789 240x280 mit Adafruit_ST7789: CS 10, DC 8, RST 9,
Hintergrundlicht an Pin 26 (muss HIGH sein). Taste A an Pin 27,
INPUT_PULLUP, gedrückt = LOW.
Der Sketch soll bei jedem Druck auf A einen Würfel werfen und die Zahl
groß auf dem Display zeigen. Kommentare auf Deutsch, keine Umlaute
auf dem Display. Erkläre danach kurz jede Zeile.
```

Noch einfacher: Kopiere den Inhalt der Regeldatei `AGENTS.md` aus 27.3 an den Anfang deines Prompts. Dann kennt die KI alle Pins und Regeln.

Tipps für die Arbeit mit einem Chat-System:

- **In kleinen Schritten.** Erst die kleinste Fassung, dann eine Erweiterung nach der anderen, wie in Kapitel 26. Ein riesiger Auftrag liefert einen riesigen Sketch, in dem du Fehler nicht mehr findest.
- **Fehlermeldungen vollständig einfügen.** Kopiere die Meldung aus dem Ausgabefenster der IDE samt Zeilennummer und schreib dazu, welche Zeile das ist.
- **Beobachtungen genau beschreiben.** Nicht „geht nicht“, sondern: „Das Display bleibt schwarz, im seriellen Monitor steht `Neigungssensor antwortet nicht`.“
- **Erklären lassen.** „Was macht Zeile 12?“ oder „Warum `volatile`?“ Eine KI ist ein geduldiger Lehrer, solange du prüfst, ob die Erklärung zu dem passt, was du in diesem Buch gelernt hast.
- **Neu anfangen**, wenn sich das Gespräch im Kreis dreht. Ein neuer Chat mit einem besseren Prompt hilft oft mehr als die zehnte Korrektur.

27.3 Agenten und die Regeldatei `AGENTS.md`

Ein **KI-Agent** wie **Claude Code** (von Anthropic) oder **Codex** (von OpenAI) geht einen Schritt weiter. Er läuft auf deinem Computer in einem Ordner, den du ihm gibst. Er liest dort selbst Dateien, schreibt und ändert Sketche und kann Befehle ausführen, zum Beispiel den Compiler. So kann er

seinen Sketch selbst übersetzen, die Fehlermeldungen lesen und verbessern, bis er ohne Fehler und Warnungen durchläuft. Wie du einen Agenten installierst, steht auf den Seiten der Anbieter; zum Übersetzen braucht er **arduino-cli**, das Kommandozeilen-Werkzeug von Arduino (Anhang E).

Damit ein Agent nicht jedes Mal raten muss, für welches Gerät er schreibt, liest er beim Start eine **Regeldatei** im Ordner: Codex und viele andere Agenten eine Datei `AGENTS.md`, Claude Code eine Datei `CLAUDE.md`. Für dieses Buch gibt es beide im Ordner `vorlage` der Sketche zum Buch:

- **`AGENTS.md`** enthält alle Regeln für den PicoBoy.
- **`CLAUDE.md`** enthält nur die Zeile `@AGENTS.md`. Claude Code liest damit dieselbe Datei; du pflegst die Regeln also nur an einer Stelle.

Wohin damit? Kopiere beide Dateien in den Ordner, in dem du den Agenten startest:

- entweder in dein **Sketchbook**, den Ordner mit allen Sketchen (unter Windows und macOS meist `Dokumente/Arduino`, Kapitel 2.4); dann gelten sie für alle Sketche darin,
- oder in einen eigenen **Projektordner**, in dem der Sketch-Ordner liegt.

Starte den Agenten dann in genau diesem Ordner. Die Arduino IDE stört sich an den beiden Dateien nicht.

So sieht die Regeldatei aus (`vorlage/AGENTS.md`):

````
# Hinweise für KI-Assistenten (Codex, Claude Code und andere)

Die Sketches in diesem Ordner sind für den **PicoBoy Color Plus** (PBC+) und
die **Arduino IDE 2**. Sie stammen aus dem Buch „Programmieren mit Arduino –
PicoBoy Color Plus“. Der Mensch am anderen Ende lernt gerade programmieren:
Erkläre Änderungen in einfachen Worten.

## Hardware und Board

- Mikrocontroller **RP2350** (wie Raspberry Pi Pico 2), 2 Kerne, 150 MHz,
  520 KB RAM, 16 MB Flash. **Kein** AVR, **kein** Arduino Uno, **kein** ESP32.
- Kern „Raspberry Pi Pico/RP2040/RP2350“ von Earle Philhower (arduino-pico),
  Board **Generic RP2350**, Flash Size **16MB (Sketch: 8MB, FS: 8MB)**.
  FQBN: `rp2040:rp2040:generic_rp2350:flash=16777216_8388608`
- Display ST7789, **240 × 280 Pixel**, (0, 0) oben links, runde Ecken
  (etwa 20 Pixel Rand freilassen).
- Joystick (hoch, runter, links, rechts, Mitte), Tasten A und B, LEDs
  rot/gelb/grün, eine RGB-LED (für Adafruit NeoPixel), Lautsprecher, Neigungssensor
  STK8BA58, zwei AAA-Batterien.

## Pins (immer diese Namen und Nummern)

```cpp
#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26      // Hintergrundlicht
#define LAUTSPRECHER 15
#define LED_ROT 14
````
````

```
#define LED_GELB 13
#define LED_GRUEN 12
#define RGB_LED 11
#define TASTE_HOCH 4
#define TASTE_RUNTER 2
#define TASTE_LINKS 3
#define TASTE_RECHTS 1
#define TASTE_MITTE 0
#define TASTE_A 27
#define TASTE_B 28
#define BATTERIE A3       // Batteriespannung, Spannungsteiler 1:3
```

Benutze nur diese Pins. `LED_BUILTIN` (GP25) ist keine der LEDs.

Bibliotheken

Nur diese, nichts anderes installieren oder einbinden:

- **Adafruit ST7735 and ST7789** (mit Adafruit GFX): `#include <Adafruit_ST7789.h>`, `Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);`, `tft.init(240, 280);`
- **Adafruit NeoPixel**: `Adafruit_NeoPixel rgb(1, RGB_LED, NEO_GRB + NEO_KHZ800);`, nach `setPixelColor` immer `rgb.show()`.
- **STK8BA58** (Neigungssensor): `sensor.begin(20, 21);` und `sensor.setOrientation(STK8BA58::PICOBOY_COLOR_PLUS);`, dann `sensor.readAcceleration(x, y, z)` in g.
- Aus dem Kern: `Wire`, `EEPROM`, `LittleFS`, `tone`, `analogRead` usw.

Erfinde keine Funktionen. Wenn du nicht sicher bist, ob es eine Funktion gibt, lies die Header-Datei der Bibliothek (z. B. `Adafruit_GFX.h`) oder sag, dass du es nicht weißt.

Regeln für jeden Sketch

- Arduino-Aufbau: `setup()` und `loop()`, kein eigenes `main()`.
- Display: nach `tft.init(240, 280);` immer das Hintergrundlicht einschalten: `pinMode(TFT_LICHT, OUTPUT); digitalWrite(TFT_LICHT, HIGH);` – sonst bleibt alles schwarz.
- Flimmerfrei mit `GFXcanvas16 bild(240, 280);` zeichnen und mit `tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);` zeigen.
- Tasten: `pinMode(…, INPUT_PULLUP)`, gedrückt ist `LOW`. Entprellen selbst machen.
- I2C: **niemals** `Wire.begin()` ohne vorher die Pins zu setzen. Die Standard-Pins GP4/GP5 sind Joystick-Hoch und ein freier Pin. Richtig: `Wire.setSDA(20); Wire.setSCL(21); Wire.begin();` oder einfach `sensor.begin(20, 21)`. Der Sensor hat die Adresse 0x18.
- **`Serial1` nie benutzen** (liegt auf GP0/GP1 = Joystick Mitte/Rechts), `Serial2` auch nicht (GP8/GP9 = Display). Ausgaben über `Serial` (USB), `Serial.begin(115200);` als erste Zeile in `setup()`.
- **Keine AVR-Register** wie `PORTB`, `DDRB`, `PIND`, kein `_delay_ms`, keine `avr/…`-Header außer dem, was der Kern selbst mitbringt.
- Interrupts (`attachInterrupt`): Die Routine ist kurz, setzt nur `volatile`-Variablen. **Kein `delay`, kein `Serial`, kein `tone`, kein

```
  Zeichnen in einer ISR.**
- Zweiter Kern: `setup1()`/`loop1()`. Nur ein Kern zeichnet auf dem Display.
  Daten zwischen den Kernen über `volatile`-Variablen oder `rp2040.fifo`.
- Texte auf dem Display und im seriellen Monitor ohne Umlaute und ß
  („Schoen“, „Gruesse“).
- Zufall: `randomSeed(rp2040.hwrand32());` in `setup()`.
- Flash schonen: `EEPROM.commit()` und Dateien in `LittleFS` nur schreiben,
  wenn sich etwas geändert hat, nicht in jedem `loop()`-Durchlauf.
- Namen deutsch, klein, ohne Umlaute; 4 Leerzeichen Einrückung; immer
  geschweifte Klammern.

## Prüfen

Der Sketch muss **ohne Fehler und ohne Warnungen** übersetzen. Wenn
`arduino-cli` installiert ist:

```
arduino-cli compile -b rp2040:rp2040:generic_rp2350:flash=16777216_8388608 --warnings
  all <Sketchordner>
```

Hochladen macht der Mensch (Arduino IDE: **Hochladen**). Was du nicht selbst
testen konntest (Display, Ton, Tasten), nenne ausdrücklich, damit es am Gerät
geprüft wird.
```

Ein paar Punkte darin kennst du aus diesem Buch, andere wirst du gleich als typische KI-Fehler wiedersehen. Du darfst die Datei erweitern, etwa um deine eigenen Vorlieben: „Erkläre jede Änderung in zwei Sätzen.“

> Ein Agent führt Befehle auf **deinem Computer** aus. Lass ihn nur in einem Ordner arbeiten, in dem nichts Wichtiges liegt, das er nicht anfassen soll, und mach vorher eine Sicherungskopie deiner Sketche. Lies, welche Befehle er ausführen will, bevor du zustimmst. Bei Zweifeln: ablehnen und nachfragen.

27.4 Prüfen, was die KI schreibt

Hier sind die häufigsten Fehler, die KI-Systeme bei Sketchen für den PicoBoy machen. Alle Beispiele sind echt übersetzt und geprüft.

AVR-Code. Die meisten Arduino-Beispiele im Netz sind für den Arduino Uno mit dem Chip ATmega328P aus der Familie **AVR** geschrieben. Für Tempo schreiben sie manchmal direkt in dessen **Register**, das sind Speicherstellen, die die Hardware steuern. Der RP2350 hat ganz andere:

k27_ki_avr.ino

```
// So antwortet eine KI manchmal auf "Lass eine LED blinken" - Code fuer den
   Arduino Uno
void setup() {
    DDRB |= (1 << DDB5);        // Pin 13 des Uno als Ausgang
}

void loop() {
    PORTB ^= (1 << PORTB5);     // LED umschalten
    delay(500);
}
```

Ausgabe

```
k27_ki_avr.ino: In function 'void setup()':
k27_ki_avr.ino:3:5: error: 'DDRB' was not declared in this scope
    3 |     DDRB |= (1 << DDB5);        // Pin 13 des Uno als Ausgang
      |     ^~~~
k27_ki_avr.ino:3:19: error: 'DDB5' was not declared in this scope; did you mean 'D5'?
    3 |     DDRB |= (1 << DDB5);        // Pin 13 des Uno als Ausgang
      |                   ^~~~
      |                   D5
k27_ki_avr.ino: In function 'void loop()':
k27_ki_avr.ino:7:5: error: 'PORTB' was not declared in this scope
    7 |     PORTB ^= (1 << PORTB5);     // LED umschalten
      |     ^~~~~
k27_ki_avr.ino:7:20: error: 'PORTB5' was not declared in this scope
    7 |     PORTB ^= (1 << PORTB5);     // LED umschalten
      |                    ^~~~~~
Fehler beim Build: exit status 1
```

`DDRB`, `PORTB`, `PINB`, `_delay_ms` und Header wie `avr/io.h` gibt es auf dem PicoBoy nicht. Richtig ist `pinMode` und `digitalWrite` mit den Pins des PicoBoy.

Erfundene Funktionen. Eine KI kennt viele Bibliotheken ungefähr. Sie erfindet gern eine Funktion, die praktisch wäre:

k27_ki_erfunden.ino

```
// Eine erfundene Funktion: Adafruit GFX kennt kein drawCenteredText
#include <Adafruit_ST7789.h>

#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
```

```
11 void setup() {
12     tft.init(240, 280);
13     pinMode(TFT_LICHT, OUTPUT);
14     digitalWrite(TFT_LICHT, HIGH);
15     tft.fillScreen(ST77XX_BLACK);
16     tft.drawCenteredText("Hallo", 120, 140, ST77XX_WHITE);
17 }
18
19 void loop() {
20 }
```

Ausgabe

```
k27_ki_erfunden.ino: In function 'void setup()':
k27_ki_erfunden.ino:16:9: error: 'class Adafruit_ST7789' has no member named
  'drawCenteredText'
   16 |     tft.drawCenteredText("Hallo", 120, 140, ST77XX_WHITE);
      |         ^~~~~~~~~~~~~~~~
Fehler beim Build: exit status 1
```

has no member named heißt: Die Klasse hat keine Funktion dieses Namens. Ein Blick in die Header-Datei der Bibliothek (Kapitel 20) oder in Anhang A zeigt, was es wirklich gibt. Text zentrieren geht mit `getTextBounds` (Kapitel 6.3).

Diese beiden Fehler sind harmlos: Der Compiler findet sie. Gefährlicher sind Fehler, die sich übersetzen lassen.

`Wire.begin()` ohne Pins. Fast jedes I2C-Beispiel beginnt mit `Wire.begin()`. Auf dem PicoBoy benutzt der Kern dann die Standard-Pins GP4/GP5. Der Neigungssensor hängt aber an GP20/GP21, und GP4 ist der Joystick (hoch):

k27_ki_wire.ino

```
4 void setup() {
5     Serial.begin(115200);
6     Wire.begin();
```

Serieller Monitor

```
0 Geraet(e) gefunden
```

Mit den richtigen Pins findet derselbe Scanner den Sensor:

k27_wire_richtig.ino

```
4 void setup() {
5     Serial.begin(115200);
6     Wire.setSDA(20);
7     Wire.setSCL(21);
8     Wire.begin();
```

Serieller Monitor

```
Geraet an Adresse 0x18
1 Geraet(e) gefunden
```

Dasselbe passiert mit `sensor.begin()` ohne Pins. Richtig ist `sensor.begin(20, 21)` (Kapitel 17).

Weitere typische Fehler:

Die KI schreibt	Was passiert	Richtig
kein `digitalWrite(TFT_LICHT, HIGH)`	Display bleibt schwarz	Hintergrundlicht einschalten (Kapitel 3.2)
Pins aus einem Tutorial für ein anderes Board	nichts reagiert, oder ein Pin stört ein anderes Bauteil	die Pins aus Anhang D
`Serial1.begin(…)`	Joystick Mitte und rechts reagieren nicht mehr (GP0/GP1)	`Serial` über USB
`LED_BUILTIN`	keine LED blinkt (GP25 ist keine der LEDs)	`LED_ROT`, `LED_GELB`, `LED_GRUEN`
`delay`, `Serial` oder `tone` in einer ISR	hängt oder verliert Ausgaben	Flagge setzen, Arbeit in `loop()` (Kapitel 24)
`EEPROM.write` ohne `commit()`	nach dem Neustart ist alles weg	`commit()` (Kapitel 25)
`EEPROM.commit()` in jedem `loop()`	nutzt den Flash ab	nur bei Änderungen schreiben
Bibliothek für ein anderes Display	übersetzt nicht oder zeigt nichts	Adafruit ST7735 and ST7789
zweiter `GFXcanvas16` in voller Größe	der Speicher wird knapp, Absturz	ein Bildpuffer genügt (Kapitel 14 und 19)

So prüfst du einen KI-Sketch:

☐ **Lesen:** Verstehst du jede Zeile? Wenn nicht: nachfragen oder nachschlagen.

☐ **Pins:** Stimmen alle Pinnummern mit Anhang D überein?

☐ **Bibliotheken:** Nur die aus Kapitel 2 und dem Kern? Gibt es jede benutzte Funktion wirklich?

☐ **Überprüfen:** Übersetzt der Sketch ohne Fehler *und* ohne Warnungen?

☐ **Testen:** Am PicoBoy jede Funktion ausprobieren, auch Randfälle (alle Tasten, alle Ränder).

☐ **Serieller Monitor:** Meldungen wie `antwortet nicht` ernst nehmen.

27.5 Fair und sicher

Datenschutz. Was du in einen Chat schreibst, landet auf den Servern des Anbieters und wird dort gespeichert, oft auch, um die KI weiter zu trainieren. Gib deshalb keine persönlichen Daten ein: keine Namen, Adressen, Fotos oder Noten von dir oder anderen, keine Passwörter. Ein Sketch für den PicoBoy enthält normalerweise nichts davon; achte trotzdem auf Kommentare wie „Von: …“. Viele Dienste haben ein Mindestalter oder verlangen die Zustimmung der Eltern. Lies die Nutzungsbedingungen, und benutze in der Schule die Dienste, die die Schule vorsieht.

Urheberrecht. Eine KI hat aus fremdem Code gelernt, und manchmal gibt sie größere Stücke davon fast wörtlich wieder. Für eigene Sketche zum Lernen spielt das kaum eine Rolle. Willst du ein Projekt veröffentlichen, schreib dazu, welche Teile von einer KI stammen, und achte bei Bildern, Musik und Code aus dem Netz auf die Lizenz. Bibliotheken wie Adafruit GFX haben eigene Lizenzen, die du beim Weitergeben einhalten musst.

Schule und Prüfungen. Ob und wie du KI benutzen darfst, entscheidet deine Lehrkraft. Bei Hausaufgaben und Projekten gilt: Kennzeichne, was von einer KI stammt, und erkläre, was du selbst geändert hast. In Prüfungen ist KI meist nicht erlaubt; und dort hilft dir nur, was du selbst verstanden hast. Gute Faustregel: Benutze die KI so, dass du danach **mehr** kannst, nicht weniger. Lass dir erklären, statt nur kopieren.

27.6 Ein Spiel im Dialog mit der KI

> **Beispiel.** Der folgende Dialog ist gekürzt und zeigt, wie eine solche Arbeit ablaufen **kann**. Echte KI-Systeme antworten jedes Mal etwas anders. Echt und geprüft sind nur die Sketche `k27_entwurf` und `k27_ergebnis`.

Du: *(Inhalt von `AGENTS.md` eingefügt)* Schreibe einen Sketch: Eine Kugel rollt über das Display, je nachdem, wie man den PicoBoy neigt. An den Rändern prallt sie ab. Taste A wechselt die Farbe der Kugel. Fang mit einem kleinen Test an, der nur die Neigung im seriellen Monitor ausgibt.

KI: Gern, hier ist ein erster Test für den Neigungssensor … *(es folgt ein Sketch; der wichtige Ausschnitt:)*

k27_entwurf.ino

```
13 void setup() {
14     Serial.begin(115200);
15     tft.init(240, 280);
16     pinMode(TFT_LICHT, OUTPUT);
17     digitalWrite(TFT_LICHT, HIGH);
18     if (!sensor.begin()) {
19         Serial.println("Neigungssensor antwortet nicht");
20     }
21 }
22
23 void loop() {
24     float ax, ay, az;
25     sensor.readAcceleration(ax, ay, az);
26     Serial.printf("x %.2f  y %.2f  z %.2f\n", ax, ay, az);
27     delay(500);
28 }
```

Wir laden den Test hoch. Der serielle Monitor zeigt:

Serieller Monitor

```
Neigungssensor antwortet nicht
x 0.00  y 0.00  z 0.00
x 0.00  y 0.00  z 0.00
```

Trotz Regeldatei hat die KI `sensor.begin()` ohne Pins geschrieben, und auch `setOrientation` fehlt. Genau dafür lohnt sich der kleine Test zuerst: Der Fehler fällt sofort auf, nicht erst im fertigen Spiel.

Du: Der serielle Monitor zeigt `Neigungssensor antwortet nicht`, und alle Werte sind 0. Laut Regeln hängt der Sensor an GP20/GP21.

KI: Du hast recht, ich habe die Pins vergessen. Richtig ist `sensor.begin(20, 21)` und danach `sensor.setOrientation(STK8BA58::PICOBOY_COLOR_PLUS)` … *(korrigierter Test, danach auf Nachfrage das Spiel)*

Das Spiel der KI flackerte zunächst, weil es direkt auf `tft` zeichnete; auf den Hinweis „bitte mit `GFXcanvas16` wie in den Regeln“ hat sie es umgebaut. Wir haben den Sketch dann gelesen, die Zahlen für Reibung und Abprall selbst ausprobiert und einen Rahmen ergänzt. Das Ergebnis:

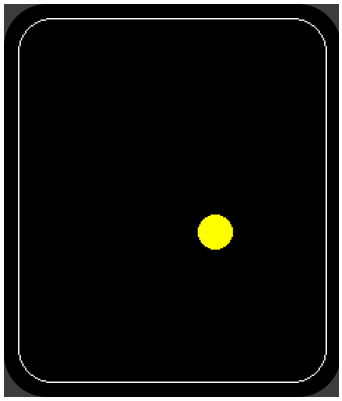
k27_ergebnis.ino

```
void loop() {
    float ax, ay, az;
    sensor.readAcceleration(ax, ay, az);
    vx = (vx + ax * 0.8) * 0.98;    // Neigung beschleunigt, Reibung bremst
    vy = (vy + ay * 0.8) * 0.98;
    x += vx;
    y += vy;
    if (x < LINKS + RADIUS) {
        x = LINKS + RADIUS;
        vx = -vx * 0.6;             // abprallen und etwas Schwung verlieren
    }
    if (x > RECHTS - RADIUS) {
        x = RECHTS - RADIUS;
        vx = -vx * 0.6;
    }
    if (y < OBEN + RADIUS) {
        y = OBEN + RADIUS;
        vy = -vy * 0.6;
    }
    if (y > UNTEN - RADIUS) {
        y = UNTEN - RADIUS;
        vy = -vy * 0.6;
    }

    bool a_jetzt = digitalRead(TASTE_A) == LOW;
    if (a_jetzt && !a_vorher) {
        farbe = (farbe + 1) % 4;
    }
    a_vorher = a_jetzt;

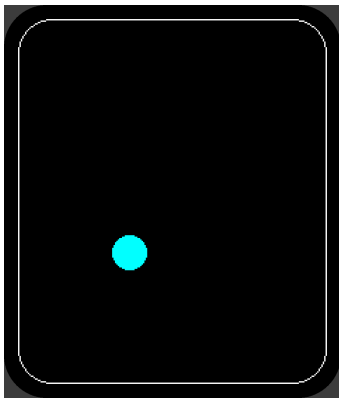
    bild.fillScreen(ST77XX_BLACK);
    bild.drawRoundRect(LINKS, OBEN, RECHTS - LINKS, UNTEN - OBEN, 24,
      ST77XX_WHITE);
    bild.fillCircle((int)x, (int)y, RADIUS, FARBEN[farbe]);
    tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
}
```

- Zeile 44 und 45: Die Neigung (in g) beschleunigt die Kugel, der Faktor 0,98 bremst sie in jedem Bild etwas ab, wie Reibung.
- Zeile 48 bis 63: Stößt die Kugel an den Rand, wird sie zurückgesetzt, und ihre Geschwindigkeit dreht sich um. Mit dem Faktor 0,6 verliert sie dabei Schwung.
- Der Kopf des Sketches (`k27_ergebnis`) enthält wie in Kapitel 26 Board und Bibliotheken und den Hinweis, dass der Sketch mit einer KI entstanden ist.



Im Prüflauf ist der PicoBoy erst nach rechts unten geneigt, nach einer Sekunde nach links oben; dann wechselt A die Farbe:

Was jetzt noch fehlt, merkst du erst am Gerät: Fühlt sich die Kugel richtig an? Ist sie zu schnell oder zu träge? Solche Fragen beantwortet keine KI für dich. Probiere die Zahlen in Zeile 44, 45 und 50 aus.



Übungsaufgaben

Aufgabe 1 ⭐ KI-Fehler finden Diesen Sketch hat eine KI für „Zeige die Neigung als Zahl an und blinke dabei“ geschrieben. Er übersetzt ohne Fehler, tut auf dem PicoBoy aber nicht, was er soll. Finde mindestens vier Fehler und verbessere sie.

k27_aufgabe1.ino

```
1  // Von einer KI: "Zeigt die Neigung als Zahl an und blinkt dabei"
2  // Der Sketch baut ohne Fehler - und tut auf dem PicoBoy trotzdem nicht, was
       er soll.
3  #include <Adafruit_ST7789.h>
4  #include <STK8BA58.h>
5
6  #define TFT_CS 10
7  #define TFT_DC 8
8  #define TFT_RST 9
9
10 Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
11 STK8BA58 sensor;
12
13 void setup() {
14     Serial1.begin(115200);
15     tft.init(240, 280);
```

```
    pinMode(LED_BUILTIN, OUTPUT);
    Wire.begin();
    sensor.begin();
}

void loop() {
    float x, y, z;
    sensor.readAcceleration(x, y, z);
    tft.fillScreen(ST77XX_BLACK);
    tft.setTextSize(3);
    tft.setCursor(40, 120);
    tft.printf("x %.2f", x);
    Serial1.printf("x %.2f\n", x);
    digitalWrite(LED_BUILTIN, !digitalRead(LED_BUILTIN));
    delay(500);
}
```

Aufgabe 2 ⭐⭐ Ein guter Prompt Schreibe einen Prompt für ein Chat-System: Die drei LEDs sollen eine Ampel nachmachen (rot, rot-gelb, grün, gelb, rot …), mit Taste A schaltet sie eine Fußgängerampel mit Ton. Welche Angaben braucht die KI, damit sie nicht raten muss?

Aufgabe 3 ⭐⭐ Erfunden? Eine KI liefert diesen Sketch für „Die RGB-LED soll rot leuchten“. Überprüfe ihn, lies die Fehlermeldung und schreibe ihn mit den echten Befehlen der Bibliothek Adafruit NeoPixel (Kapitel 10.3).

k27_aufgabe3.ino

```
// Von einer KI: "Die RGB-LED soll rot leuchten"
#include <Adafruit_NeoPixel.h>

#define RGB_LED 11

Adafruit_NeoPixel rgb(1, RGB_LED, NEO_GRB + NEO_KHZ800);

void setup() {
    rgb.begin();
    rgb.setColor(255, 0, 0);
}

void loop() {
}
```

Aufgabe 4 ⭐⭐⭐ **Weiter mit KI** Erweitere `k27_ergebnis` mit Hilfe einer KI: Prallt die Kugel kräftig gegen den Rand, klingt ein Ton, dessen Höhe vom Aufprall abhängt, und ein Zähler oben auf dem Display zählt die Treffer. Notiere deine Prompts, was die KI falsch gemacht hat und was du selbst geändert hast.

Die Lösungen stehen in Anhang C.

Das Wichtigste in Kürze

- Eine KI schreibt **Vorschläge**. Lesen, verstehen, überprüfen und am Gerät testen musst du selbst.
- Gute Prompts nennen Ziel, Hardware (Board, Pins, Bibliotheken) und Regeln. Arbeite in kleinen Schritten, gib Fehlermeldungen vollständig zurück.
- Agenten (Claude Code, Codex) lesen eine Regeldatei: **`AGENTS.md`**, für Claude Code **`CLAUDE.md`** mit `@AGENTS.md`. Beide gehören in den Ordner, in dem du den Agenten startest (Sketchbook oder Projektordner).
- Typische KI-Fehler: AVR-Register wie `PORTB`, erfundene Funktionen, `Wire.begin()` ohne Pins 20/21, falsche Pins, `Serial1`, fehlendes Hintergrundlicht, `delay` in einer ISR.
- Keine persönlichen Daten in den Chat; KI-Anteile kennzeichnen; in der Schule gelten die Regeln der Lehrkraft.

ANHANG A

Die wichtigsten Befehle

Die Übersicht gilt für den Kern „Raspberry Pi Pico/RP2040/RP2350“ von Earle Philhower (6.x), Board **Generic RP2350**, und die Bibliotheken Adafruit GFX, Adafruit ST7735 and ST7789, Adafruit NeoPixel und STK8BA58. Die Zahl in Klammern nennt das Kapitel, in dem der Befehl erklärt wird.

A.1 Die Pins des PicoBoy Color Plus

```
#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26
#define LAUTSPRECHER 15
#define LED_ROT 14
#define LED_GELB 13
#define LED_GRUEN 12
#define RGB_LED 11
#define TASTE_HOCH 4
#define TASTE_RUNTER 2
#define TASTE_LINKS 3
#define TASTE_RECHTS 1
#define TASTE_MITTE 0
#define TASTE_A 27
#define TASTE_B 28
#define BATTERIE A3
```

Neigungssensor: I2C an GP20 (SDA) und GP21 (SCL), Adresse `0x18`. Display: SPI an GP18 (Takt) und GP19 (Daten).

A.2 Grundgerüst eines Sketches

```
#include <Adafruit_ST7789.h>

#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
GFXcanvas16 bild(240, 280);

void setup() {
    Serial.begin(115200);
    tft.init(240, 280);
    tft.setSPISpeed(62500000);
    pinMode(TFT_LICHT, OUTPUT);
```

```
    digitalWrite(TFT_LICHT, HIGH);
}

void loop() {
    bild.fillScreen(ST77XX_BLACK);
    // ... in bild zeichnen ...
    tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
}
```

A.3 Arduino-Kern

Pins (9, 10, 21)

Befehl	Wirkung
`pinMode(pin, OUTPUT)`	Pin als Ausgang
`pinMode(pin, INPUT_PULLUP)`	Eingang mit Pull-up; Tasten: gedrückt = `LOW`
`digitalWrite(pin, HIGH)` / `LOW`	Ausgang an / aus; beendet auch PWM auf dem Pin
`digitalRead(pin)`	`HIGH` oder `LOW`
`analogWrite(pin, wert)`	PWM, 0 bis 255 (Vorgabe), 1000 Hz; kein `pinMode` nötig
`analogWriteRange(bereich)`	größter Wert für `analogWrite` (3 bis 65535), gilt für alle Pins
`analogWriteResolution(bits)`	Bereich als Bitzahl (2 bis 16)
`analogWriteFreq(hertz)`	PWM-Frequenz (100 bis 10 000 000), gilt für alle Pins
`analogRead(A3)`	Spannung an GP26–GP29 (`A0` – `A3`); 0–1023, nach `analogReadResolution(12)` 0–4095
`analogReadResolution(bits)`	Bitzahl für `analogRead`; 12 = volle Auflösung
`attachInterrupt(digitalPinToInterrupt(pin), funktion, FALLING)`	Funktion bei fallender Flanke aufrufen (24)
`detachInterrupt(digitalPinToInterrupt(pin))`	wieder abschalten (24)

Batteriespannung (12 Bit): `float volt = analogRead(A3) * 3.3 / 4095 * 3;`

Zeit (7)

Befehl	Wirkung
`delay(ms)`	warten (Millisekunden)
`delayMicroseconds(us)`	warten (Mikrosekunden)
`millis()`	Millisekunden seit dem Start (`unsigned long`)
`micros()`	Mikrosekunden seit dem Start (`unsigned long`)

Ton (10)

Befehl	Wirkung
`tone(LAUTSPRECHER, frequenz)`	Dauerton
`tone(LAUTSPRECHER, frequenz, dauer)`	Ton für `dauer` ms; wartet **nicht**
`noTone(LAUTSPRECHER)`	Ton aus

Zahlen und Zufall (5, 7)

Befehl	Wirkung
`random(max)`	Zufallszahl 0 bis `max - 1`
`random(min, max)`	Zufallszahl `min` bis `max - 1`
`randomSeed(rp2040.hwrand32())`	bei jedem Start andere Zufallszahlen
`constrain(x, a, b)`	`x` auf den Bereich `a` bis `b` begrenzen
`map(x, a1, b1, a2, b2)`	`x` von einem Bereich in einen anderen umrechnen (ganzzahlig)
`min(a, b)`, `max(a, b)`, `abs(x)`	kleinerer, größerer Wert, Betrag
`sqrt(x)`, `pow(x, y)`, `sin(x)`, `cos(x)`	Wurzel, Potenz, Winkelfunktionen (Bogenmaß)

Serieller Monitor (4)

Befehl	Wirkung
`Serial.begin(115200)`	starten (erste Zeile in `setup`)
`while (!Serial) { delay(10); }`	warten, bis ein Monitor verbunden ist
`Serial.print(x)` / `Serial.println(x)`	ausgeben, ohne / mit Zeilenumbruch
`Serial.printf("%d %.2f %s\n", ...)`	formatiert ausgeben (Kern-Erweiterung)
`Serial.available()`	Anzahl empfangener Zeichen

Befehl	Wirkung
`Serial.read()`	ein Zeichen lesen (−1, wenn keins da ist)
`Serial.readStringUntil('\n')`	Zeile als `String` lesen
`Serial.parseInt()`, `Serial.parseFloat()`	Zahl lesen

Kern-Erweiterungen `rp2040.` (19)

Befehl	Wirkung
`rp2040.hwrand32()`	echte Zufallszahl aus der Hardware
`rp2040.getFreeHeap()`	freier Heap in Byte
`rp2040.getUsedHeap()`, `rp2040.getTotalHeap()`	belegter, gesamter Heap
`rp2040.reboot()`	Neustart

Zweiter Kern (24): `void setup1()` und `void loop1()` laufen auf dem zweiten Prozessorkern, sobald sie im Sketch stehen.

A.4 Adafruit GFX: Zeichnen (3, 6, 14)

Gilt für `tft` (sofort sichtbar) und für jeden Canvas wie `bild` (sichtbar erst mit `drawRGBBitmap`). Koordinaten: (0, 0) links oben, x bis 239, y bis 279. Farben: 16 Bit (RGB565).

Befehl	Wirkung
`fillScreen(farbe)`	alles füllen
`drawPixel(x, y, farbe)`	ein Pixel
`drawLine(x0, y0, x1, y1, farbe)`	Linie
`drawFastHLine(x, y, b, farbe)` / `drawFastVLine(x, y, h, farbe)`	waagerechte / senkrechte Linie
`drawRect(x, y, b, h, farbe)` / `fillRect(…)`	Rechteck, Umriss / gefüllt
`drawRoundRect(x, y, b, h, r, farbe)` / `fillRoundRect(…)`	Rechteck mit runden Ecken (Radius `r`)
`drawCircle(x, y, r, farbe)` / `fillCircle(…)`	Kreis um (x, y)
`drawEllipse(x, y, rx, ry, farbe)` / `fillEllipse(…)`	Ellipse
`drawTriangle(x0, y0, x1, y1, x2, y2, farbe)` / `fillTriangle(…)`	Dreieck
`drawRGBBitmap(x, y, pixel, b, h)`	Bild aus 16-Bit-Farben (15)
`drawRGBBitmap(x, y, pixel, maske, b, h)`	dasselbe mit durchsichtigen Pixeln (15)

Befehl	Wirkung
`drawBitmap(x, y, bits, b, h, farbe)`	einfarbiges Bild, 1 Bit pro Pixel
`width()`, `height()`	Breite, Höhe (nach `setRotation`)
`setRotation(0 … 3)`	Richtung; 1 und 3 = Querformat (23)

Text

Befehl	Wirkung
`setCursor(x, y)`	Position der nächsten Ausgabe (linke obere Ecke; mit FreeFont: Grundlinie)
`setTextColor(farbe)`	Schriftfarbe, ohne Hintergrund
`setTextColor(farbe, hintergrund)`	mit Hintergrund (nur Standardschrift)
`setTextSize(n)`	Vergrößerung; 1 = 6 · 8 Pixel pro Zeichen
`setTextWrap(false)`	nicht in die nächste Zeile umbrechen
`print(x)`, `println(x)`, `printf(…)`	Text oder Zahl ausgeben
`setFont(&FreeSans12pt7b)`	FreeFont benutzen (`#include <Fonts/FreeSans12pt7b.h>`); `setFont()` = Standardschrift (6)
`getTextBounds(text, x, y, &x1, &y1, &b, &h)`	Maße eines Texts, zum Zentrieren (6)

Farben

Konstante	Wert	Konstante	Wert
`ST77XX_BLACK`	`0x0000`	`ST77XX_WHITE`	`0xFFFF`
`ST77XX_RED`	`0xF800`	`ST77XX_GREEN`	`0x07E0`
`ST77XX_BLUE`	`0x001F`	`ST77XX_CYAN`	`0x07FF`
`ST77XX_MAGENTA`	`0xF81F`	`ST77XX_YELLOW`	`0xFFE0`
`ST77XX_ORANGE`	`0xFC00`		

Eigene Farben: `tft.color565(rot, gruen, blau)` mit Werten 0 bis 255.

Canvas (14, 19)

Befehl	Wirkung
`GFXcanvas16 bild(240, 280);`	Bild im RAM (134 400 Byte vom Heap)
`bild.getBuffer()`	Zeiger `uint16_t *` auf die Pixel; `nullptr`, wenn kein Platz war
`bild.getPixel(x, y)`	Farbe eines Pixels lesen
`tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280)`	Canvas zum Display schicken

A.5 Adafruit ST7789: das Display (3, 14, 23)

Befehl	Wirkung
`Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);`	Display-Objekt (SPI an GP18/GP19)
`tft.init(240, 280)`	Display starten; **immer** mit 240, 280
`tft.setSPISpeed(62500000)`	nach `init`: SPI-Takt real 37,5 MHz (sonst 25 MHz)
`tft.invertDisplay(true / false)`	Farben normal / umgekehrt (bei diesem Display ist `true` normal)
`tft.enableDisplay(true / false)`	Anzeige an / aus, Bild bleibt im Chip
`tft.enableSleep(true / false)`	Schlafmodus an / aus
`tft.sendCommand(befehl, &daten, anzahl)`	eigenen Befehl an den Chip schicken

Das Hintergrundlicht ist ein eigener Pin: `digitalWrite(TFT_LICHT, HIGH)` oder gedimmt `analogWrite(TFT_LICHT, 0 … 255)` (21).

A.6 Adafruit NeoPixel: die RGB-LED (10)

Befehl	Wirkung
`#include <Adafruit_NeoPixel.h>`	Bibliothek einbinden
`Adafruit_NeoPixel rgb(1, RGB_LED, NEO_GRB + NEO_KHZ800);`	eine LED an GP11
`rgb.begin()`	in `setup()`
`rgb.setPixelColor(0, rot, gruen, blau)`	Farbe setzen (0–255), noch unsichtbar
`rgb.setPixelColor(0, farbe)`	mit `uint32_t`-Farbe

Befehl	Wirkung
`rgb.Color(rot, gruen, blau)`	Farbe als `uint32_t`
`rgb.ColorHSV(farbton, saettigung, helligkeit)`	Farbe aus dem Farbkreis; Farbton 0–65535
`rgb.gamma32(farbe)`	für das Auge gleichmäßigere Helligkeit
`rgb.setBrightness(0 … 255)`	Gesamthelligkeit
`rgb.clear()`	aus (Farbe 0)
`rgb.show()`	Farbe zur LED schicken – erst jetzt sichtbar

A.7 STK8BA58: der Neigungssensor (17, 20)

Befehl	Wirkung
`#include <STK8BA58.h>`	Bibliothek einbinden
`STK8BA58 sensor;`	Sensor-Objekt (Wire, Adresse `0x18`)
`sensor.begin(20, 21)`	starten an GP20/GP21; `false`, wenn er nicht antwortet
`sensor.setOrientation(STK8BA58::PICOBOY_COLOR_PLUS)`	Achsen des PicoBoy: x rechts, y zum Joystick, z aus dem Display
`sensor.readAcceleration(x, y, z)`	drei `float` in g (flach: 0, 0, 1)
`sensor.readAccelerationMS2(x, y, z)`	dasselbe in m/s²
`sensor.readX()`, `readY()`, `readZ()`	eine Achse in g
`sensor.readRaw(x, y, z)`	Rohwerte als `int16_t`, −2048 bis 2047
`sensor.setRange(2 / 4 / 8)`	Messbereich ±2, ±4, ±8 g
`sensor.range()`	eingestellter Messbereich
`sensor.accelerationAvailable()`	`true`, wenn ein neuer Messwert da ist
`sensor.readRegister(reg)`, `sensor.writeRegister(reg, wert)`	Register direkt lesen / schreiben (22)

A.8 Wire: I2C ohne Bibliothek (22)

Befehl	Wirkung
`#include <Wire.h>`	einbinden
`Wire.setSDA(20); Wire.setSCL(21);`	Pins des PicoBoy, **vor** `begin`
`Wire.begin()`	starten (100 kHz)
`Wire.setClock(hertz)`	anderer Takt
`Wire.beginTransmission(adresse)`	Gespräch beginnen
`Wire.write(byte)`	Byte zum Senden vormerken
`Wire.endTransmission()`	senden, mit Stopp; 0 = Erfolg
`Wire.endTransmission(false)`	senden, ohne Stopp (dann lesen)
`Wire.requestFrom(adresse, anzahl)`	Bytes anfordern; Rückgabe: wie viele kamen
`Wire.read()`	ein empfangenes Byte

A.9 SPI ohne Display-Bibliothek (23)

Befehl	Wirkung
`#include <SPI.h>`	einbinden
`SPI.begin()`	starten (GP18 Takt, GP19 Daten)
`SPI.beginTransaction(SPISettings(takt, MSBFIRST, SPI_MODE0))`	einstellen und belegen
`SPI.transfer(byte)`, `SPI.transfer16(zahl)`	8 bzw. 16 Bit senden
`SPI.endTransaction()`	freigeben

A.10 Daten dauerhaft speichern (25)

Befehl	Wirkung
`#include <EEPROM.h>`	einbinden
`EEPROM.begin(groesse)`	Speicherbereich anlegen (im Flash)
`EEPROM.read(adresse)`, `EEPROM.write(adresse, byte)`	ein Byte lesen / schreiben
`EEPROM.get(adresse, variable)`, `EEPROM.put(adresse, variable)`	ganze Variable lesen / schreiben
`EEPROM.commit()`	Änderungen wirklich in den Flash schreiben
`#include <LittleFS.h>`, `LittleFS.begin()`	Dateisystem im Flash (FS: 8 MB)

A.11 Sprachmittel in Kürze

Schreibweise	Bedeutung
`int`, `long`, `unsigned long`, `float`, `bool`, `char`, `byte`, `uint8_t`, `int16_t`, `uint16_t`, `uint32_t`	Datentypen (5)
`const int N = 5;`, `#define N 5`	Konstanten (5)
`if`, `else`, `switch`, `for`, `while`	Entscheidungen und Schleifen (7, 8)
`int werte[5];`, `char text[20];`, `String s;`	Arrays und Texte (12)
`struct`, `class`, `public:`, `private:`	Strukturen und Klassen (13)
`&x`, `int *p`, `*p`, `p[i]`	Adresse, Zeiger, Inhalt (19)
`void f(int &x)`	Referenz-Parameter (19)
`a << n`, `a >> n`, `a & b`, `a | b`	Bit-Operationen (22)
`#include <…>` / `#include "…"`	Bibliothek / Datei im Sketch-Ordner (15, 20)

ANHANG B

Fehlermeldungen beim Kompilieren und Hochladen

Dieser Anhang sammelt die Meldungen, die dir am häufigsten begegnen. Alle Meldungen zum Kompilieren stammen aus echten Sketchen, die mit dem Kern von Earle Philhower für den PicoBoy übersetzt wurden. Wie man eine Meldung grundsätzlich liest, steht in Kapitel 3.6. Die wichtigsten Regeln:

- Die Meldung nennt **Datei:Zeile:Spalte**. Der Fehler steht dort **oder kurz davor**.
- Verbessere den **ersten** Fehler und kompiliere neu. Spätere Meldungen sind oft nur Folgen.
- `error:` hält das Kompilieren auf. `warning:` nicht, deutet aber fast immer auf einen echten Fehler hin. `note:` ist ein Hinweis zur Meldung davor.
- Die Meldungen des Compilers sind englisch, auch wenn die IDE deutsch eingestellt ist. Die letzte Zeile, `Fehler beim Build: exit status 1`, kommt von der IDE und heißt nur: Es hat nicht geklappt.

B.1 Tippfehler und Satzzeichen

`expected ';' before …` – ein Semikolon fehlt.

Ausgabe

```
kB_fehler_semikolon.ino: In function 'void setup()':
kB_fehler_semikolon.ino:11:23: error: expected ';' before 'pinMode'
   11 |     tft.init(240, 280)
      |                       ^
      |                       ;
   12 |     pinMode(TFT_LICHT, OUTPUT);
      |     ~~~~~~~
Fehler beim Build: exit status 1
```

Das `;` fehlt am Ende von Zeile 11. Der Compiler merkt es erst beim nächsten Wort (`pinMode` in Zeile 12) und schlägt das `;` gleich vor.

`… was not declared in this scope` – ein Name ist unbekannt.

Ausgabe

```
kB_fehler_unbekannt.ino: In function 'void setup()':
kB_fehler_unbekannt.ino:14:5: error: 'punkte' was not declared in this scope
   14 |     punkte = 0;
      |     ^~~~~~
Fehler beim Build: exit status 1
```

Die Variable `punkte` wurde nie angelegt, oder sie ist woanders angelegt (in einer anderen Funktion, Kapitel 11) und hier nicht sichtbar. Prüfe auch die Schreibweise. Ist der Name ähnlich wie ein bekannter, fragt der Compiler nach:

```
Ausgabe
kB_fehler_gross.ino: In function 'void setup()':
kB_fehler_gross.ino:14:20: error: 'ST77XX_White' was not declared in this scope; did
  you mean 'ST77XX_WHITE'?
   14 |     tft.fillScreen(ST77XX_White);
      |                    ^~~~~~~~~~~~
      |                    ST77XX_WHITE
Fehler beim Build: exit status 1
```

C++ unterscheidet Groß- und Kleinschreibung: `ST77XX_White` ist nicht `ST77XX_WHITE`.

`… has no member named …` – eine Methode gibt es nicht.

```
Ausgabe
kB_fehler_methode.ino: In function 'void setup()':
kB_fehler_methode.ino:14:9: error: 'class Adafruit_ST7789' has no member named
  'fillscreen'; did you mean 'fillScreen'?
   14 |     tft.fillscreen(ST77XX_BLUE);
      |         ^~~~~~~~~~
      |         fillScreen
Fehler beim Build: exit status 1
```

`tft` kennt `fillScreen`, aber nicht `fillscreen`. Welche Methoden es gibt, steht in Anhang A oder in der `.h`-Datei der Bibliothek (Kapitel 20).

`missing terminating " character` – ein Text ist nicht abgeschlossen.

```
Ausgabe
kB_fehler_anfuehrung.ino:3:20: warning: missing terminating " character
    3 |     Serial.println("Hallo);
      |                    ^
kB_fehler_anfuehrung.ino:3:20: error: missing terminating " character
    3 |     Serial.println("Hallo);
      |                    ^~~~~~~
kB_fehler_anfuehrung.ino: In function 'void setup()':
kB_fehler_anfuehrung.ino:4:1: error: expected primary-expression before '}' token
    4 | }
      | ^
Fehler beim Build: exit status 1
```

Nach `"Hallo` fehlt das schließende `"`. Der Compiler liest den Rest der Zeile als Text und versteht danach nichts mehr; die zweite Meldung ist nur eine Folge.

B.2 Klammern

`expected '}' at end of input` – eine schließende Klammer fehlt.

Ausgabe

```
kB_fehler_klammer.ino: In function 'void setup()':
kB_fehler_klammer.ino:18:13: error: a function-definition is not allowed here before
  '{' token
   18 | void loop() {
      |             ^
kB_fehler_klammer.ino:19:2: error: expected '}' at end of input
   19 | }
      |  ^
kB_fehler_klammer.ino:10:14: note: to match this '{'
   10 | void setup() {
      |              ^
Fehler beim Build: exit status 1
```

Hier fehlt die `}` des `if`-Blocks in `setup()`. Der Compiler hält deshalb `void loop()` für einen Teil von `setup()` – daher die erste Meldung (*a function-definition is not allowed here*). Am Ende der Datei fehlt ihm dann eine Klammer. Die `note:` zeigt die `{`, die noch offen ist: Der Fehler steckt irgendwo **in** diesem Block. Die Einrückung hilft beim Suchen: **Werkzeuge → Automatische Formatierung** (**Strg + T**) rückt den Sketch neu ein, und die Stelle fällt auf.

`variable or field 'setup' declared void` – die runden Klammern fehlen.

Ausgabe

```
kB_fehler_setup_fehlt_klammer.ino:1:6: error: variable or field 'setup' declared void
    1 | void setup {
      |      ^~~~~
Fehler beim Build: exit status 1
```

Es muss `void setup() {` heißen. Ohne `()` hält der Compiler `setup` für eine Variable vom Typ `void`, und die gibt es nicht.

B.3 Typen, Argumente, doppelte Namen

`invalid conversion from … to …` – der Wert passt nicht zum Typ.

Ausgabe

```
kB_fehler_typ.ino: In function 'void setup()':
kB_fehler_typ.ino:3:16: error: invalid conversion from 'const char*' to 'int' [-
  fpermissive]
    3 |     int name = "Mia";
      |                ^~~~~
      |                |
      |                const char*
Fehler beim Build: exit status 1
```

Ein Text (`const char*`) passt nicht in eine Variable für ganze Zahlen. Richtig wäre `const char *name = "Mia";` oder `String name = "Mia";` (Kapitel 12).

`no matching function for call to …` – Anzahl oder Art der Argumente stimmt nicht.

```
Ausgabe
kB_fehler_argumente.ino: In function 'void setup()':
kB_fehler_argumente.ino:14:19: error: no matching function for call to
  'Adafruit_ST7789::drawCircle(int, int, int)'
   14 |     tft.drawCircle(120, 140, 50);
      |     ~~~~~~~~~~~~~~^~~~~~~~~~~~~~
  • there is 1 candidate
```

`drawCircle` braucht vier Argumente: x, y, Radius **und Farbe**. Die Meldung zeigt in Klammern, womit du aufgerufen hast (`int, int, int`). Im Ausgabefenster der IDE folgen darunter noch Zeilen aus der Bibliothek mit dem Kopf der passenden Funktion (*candidate*); dort steht, was sie erwartet.

`redefinition of …` – ein Name ist doppelt vergeben.

```
Ausgabe
kB_fehler_doppelt.ino:2:5: error: redefinition of 'int punkte'
    2 | int punkte = 10;
      |     ^~~~~~
kB_fehler_doppelt.ino:1:5: note: 'int punkte' previously defined here
    1 | int punkte = 0;
      |     ^~~~~~
Fehler beim Build: exit status 1
```

Jede Variable darf nur einmal angelegt werden. Willst du ihr später einen anderen Wert geben, schreibst du nur `punkte = 10;` ohne `int`.

`no return statement in function returning non-void` – `return` fehlt.

```
Ausgabe
kB_fehler_rueckgabe.ino: In function 'int doppelt(int)':
kB_fehler_rueckgabe.ino:2:9: warning: unused variable 'ergebnis' [-Wunused-variable]
    2 |     int ergebnis = zahl * 2;
      |         ^~~~~~~~
kB_fehler_rueckgabe.ino:3:1: error: no return statement in function returning non-
  void [-Werror=return-type]
    3 | }
      | ^
cc1plus: some warnings being treated as errors
Fehler beim Build: exit status 1
```

Die Funktion verspricht mit `int`, eine Zahl zurückzugeben, tut es aber nicht: Es fehlt `return ergebnis;`. Bei vielen anderen Compilern ist das nur eine Warnung. Der Kern von Earle Philhower macht daraus mit `-Werror=return-type` einen Fehler (*warnings being treated as errors*), denn ohne `return` liefert die Funktion irgendeinen Wert.

B.4 Fehlende Bibliotheken

`fatal error: … .h: No such file or directory` – eine Datei für `#include` wurde nicht gefunden.

```
Ausgabe
kB_fehler_include.ino:1:10: fatal error: Adafruit_ST7798.h: No such file or directory
    1 | #include <Adafruit_ST7798.h>
      |          ^~~~~~~~~~~~~~~~~~~
compilation terminated.
Fehler beim Build: exit status 1
```

Entweder ist der Name falsch geschrieben (hier `ST7798` statt `ST7789`), oder die Bibliothek ist nicht installiert (Kapitel 2.4). `fatal` heißt: Der Compiler bricht sofort ab, weitere Meldungen gibt es nicht. Bei `#include "datei.h"` mit Anführungszeichen muss die Datei im Sketch-Ordner liegen (Kapitel 15).

B.5 Fehler beim Zusammenfügen (Linker)

Nach dem Compiler fügt der **Linker** (`ld`) alle Teile zu einem Programm zusammen. Seine Meldungen sehen anders aus: Sie nennen keine Zeile im Sketch, sondern Dateien des Kerns. Hier mit gekürzten Pfaden; im Ausgabefenster stehen davor lange Ordnernamen.

`undefined reference to …` – eine Funktion fehlt ganz.

```
Ausgabe
…/ld: …/core.a(main.cpp.o): in function `main':
…/cores/rp2040/main.cpp:183:(.text.startup.main+0xbe): undefined reference to `loop'
collect2: error: ld returned 1 exit status
Fehler beim Build: exit status 1
```

Der Kern (`main.cpp`) ruft `loop()` auf, aber der Sketch hat keine Funktion mit diesem Namen – hier heißt sie `schleife()`. Jeder Sketch braucht genau ein `setup()` und ein `loop()`, auch wenn eines davon leer ist. Dieselbe Meldung kommt, wenn eine Methode in einer `.h`-Datei deklariert, aber in keiner `.cpp`-Datei definiert ist (Kapitel 20).

`cannot move location counter backwards` – der RAM ist zu klein.

```
Ausgabe
…/ld:…/memmap_default.ld:254 cannot move location counter backwards (from 2009551c to
  20080000)
collect2: error: ld returned 1 exit status
Fehler beim Build: exit status 1
```

Der Sketch legt ein globales Array mit 600 000 Byte an; der RAM hat aber nur 512 KB (524 288 Byte, Kapitel 19). Die Zahlen sind Adressen: Die Variablen würden bis `0x2009551c` reichen, der RAM endet bei `0x20080000`. Abhilfe: Daten, die sich nicht ändern, mit `const` in den Flash legen, Arrays verkleinern.

> Ist ein Sketch für den **Flash** zu groß (mehr als 8 380 416 Byte), meldet die IDE das in ihrer Speicherzeile nach dem Kompilieren. Bei Bildern und Klängen kann das passieren. Unter **Werkzeuge → Flash Size** gibt es Einstellungen mit mehr Platz für den Sketch und weniger für Dateien.

B.6 Warnungen

Warnungen halten das Kompilieren nicht auf. Die IDE zeigt sie nur, wenn unter **Datei → Einstellungen…** bei **Compiler-Meldungen** etwas anderes als „Keine“ eingestellt ist. Stelle dort **Alle** ein: Die Sketche dieses Buchs kompilieren ohne eine einzige Warnung.

`suggest parentheses around assignment used as truth value` – `=` statt `==` (Kapitel 8.3).

Ausgabe

```
kB_warnung_zuweisung.ino: In function 'void loop()':
kB_warnung_zuweisung.ino:8:15: warning: suggest parentheses around assignment used as
  truth value [-Wparentheses]
    8 |     if (leben = 0) {
      |         ~~~~~~^~~
```

`if (leben = 0)` setzt `leben` auf 0, statt zu vergleichen. Richtig: `if (leben == 0)`.

`comparison of integer expressions of different signedness` – Vergleich von Zahlen mit und ohne Vorzeichen.

Ausgabe

```
kB_warnung_vergleich.ino: In function 'void loop()':
kB_warnung_vergleich.ino:8:27: warning: comparison of integer expressions of
  different signedness: 'long unsigned int' and 'int' [-Wsign-compare]
    8 |     if (millis() - letzte > pause) {
      |         ~~~~~~~~~~~~~~~~~~^~~~~~~
```

`millis()` liefert `unsigned long`, `pause` ist ein `int`. Hier ist es harmlos, weil `pause` nie negativ ist. Sauber ist es, `pause` auch als `unsigned long` anzulegen.

`unused variable` – eine Variable wird nie benutzt.

Ausgabe

```
kB_warnung_unbenutzt.ino: In function 'void setup()':
kB_warnung_unbenutzt.ino:2:9: warning: unused variable 'zaehler' [-Wunused-variable]
    2 |     int zaehler = 0;
      |         ^~~~~~~
```

Harmlos, aber oft ein Zeichen, dass etwas vergessen wurde.

`array subscript … is above array bounds` – Zugriff hinter das Ende eines Arrays.

Ausgabe

```
kB_warnung_array.ino: In function 'void setup()':
kB_warnung_array.ino:6:18: warning: iteration 5 invokes undefined behavior [-
  Waggressive-loop-optimizations]
    6 |         werte[i] = i * 10;
      |         ~~~~~~~~~^~~~~~~
kB_warnung_array.ino:5:23: note: within this loop
    5 |     for (int i = 0; i <= 5; i++) {
      |                     ~~^~~~
kB_warnung_array.ino:6:16: warning: array subscript 5 is above array bounds of 'int
  [5]' [-Warray-bounds=]
    6 |         werte[i] = i * 10;
      |         ~~~~~~~^
kB_warnung_array.ino:1:5: note: while referencing 'werte'
    1 | int werte[5];
      |     ^~~~~
```

`werte` hat 5 Elemente, `werte[0]` bis `werte[4]`. Die Schleife läuft mit `i <= 5` aber bis `werte[5]` und schreibt in fremden Speicher. Richtig ist `i < 5`. Diese Warnung ist ernst: Auf dem Gerät überschreibt der Sketch dabei andere Variablen.

B.7 Fehler beim Hochladen

Beim Hochladen (Kapitel 2.5) startet die IDE das Hochladewerkzeug des Kerns (`tools/uf2conv.py`). Es schickt dem laufenden Sketch über den seriellen Port das Signal für den Bootmodus, sucht dann **10 Sekunden lang** nach dem Laufwerk des RP2350 und kopiert das Programm darauf. So sieht es aus, wenn kein PicoBoy antwortet (hier mit `arduino-cli`, das dasselbe Werkzeug wie die IDE benutzt):

Ausgabe

```
Resetting /dev/ttyACM99
Converting to uf2, output size: 161280, start address: 0x2000
Scanning for RP2040 devices
No drive to deploy.
Fehlgeschlagenes Hochladen: Hochladefehler: exit status 1
```

- `Resetting …` – das Signal an den Port. Klappt das nicht, gibt das Werkzeug keine Meldung aus und macht trotzdem weiter.
- `Scanning for RP2040 devices` – die Suche nach dem Laufwerk. Der Name „RP2040“ steht dort auch beim RP2350.
- `No drive to deploy.` – nach 10 Sekunden kein Laufwerk gefunden.

Ist gar kein Port gewählt, bricht das Hochladen sofort ab:

Ausgabe

```
Fehlgeschlagenes Hochladen: Kein Upload-Port vorhanden
```

Meldung oder Beobachtung	Mögliche Ursache	Was hilft
`No drive to deploy.`	Der PicoBoy hat nicht auf das Signal reagiert: falscher Port, auf dem PicoBoy läuft MicroPython, ein Spiel oder ein abgestürzter Sketch.	Bootmodus von Hand: **BOOT** halten, **RES** tippen, Port **UF2 Board** wählen, noch einmal hochladen (Kapitel 2.5, Weg 1).
`Kein Upload-Port vorhanden`	Unter **Werkzeuge → Port** ist nichts gewählt.	Port wählen; erscheint keiner: Bootmodus von Hand.
Unter **Werkzeuge → Port** steht nichts.	Reines Ladekabel ohne Datenleitungen, Kabel steckt schlecht.	Anderes Kabel, anderer USB-Anschluss.
Hochladen klappt, aber der serielle Port verschwindet danach.	Der neue Sketch ist sofort abgestürzt oder hängt (zum Beispiel `panic` durch falsche I2C-Pins, Kapitel 22), bevor USB läuft.	Bootmodus von Hand, einen einfachen Sketch hochladen, Fehler suchen.
Hochladen klappt, Bildschirm bleibt schwarz.	Hintergrundlicht nicht eingeschaltet, `tft.init` fehlt oder hat falsche Werte.	`pinMode(TFT_LICHT, OUTPUT);` `digitalWrite(TFT_LICHT, HIGH);` und `tft.init(240, 280);` prüfen.
Ausgaben im seriellen Monitor fehlen am Anfang.	Der Sketch hat schon geschrieben, bevor der Monitor verbunden war.	`while (!Serial) { delay(10); }` nach `Serial.begin` (Kapitel 4).

> Was genau die IDE bei einem Fehler zusätzlich anzeigt (Hinweisfenster, Farben), hängt von ihrer Version ab. Die Zeilen im Ausgabefenster stammen aber immer vom Compiler, vom Linker oder vom Hochladewerkzeug und sehen so aus wie hier. Für mehr Einzelheiten beim Hochladen gibt es in **Datei → Einstellungen…** ein Häkchen für eine ausführliche Ausgabe.

Lösungen zu Kapitel 1: Mikrocontroller und Arduino

Aufgabe 1 – Begriffe ordnen

1. **Sketch:** das Programm als Text in der Programmiersprache C++, in der Arduino IDE geschrieben (`name.ino`).
2. **Compiler:** das Programm, das den Sketch 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. **Hochladen:** Die IDE bringt die `.uf2`-Datei über das USB-Kabel auf den PicoBoy; der PicoBoy schreibt sie in seinen Flash-Speicher und startet das Programm.

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 Sketch kann den Bootmodus nicht überschreiben. **Hochladen** ohne Bootmodus klappt nur, wenn ein Sketch läuft, der auf das Signal der IDE hört; ein Sketch, der hängen bleibt, tut das womöglich nicht mehr. Sie soll den PicoBoy mit dem Computer verbinden, BOOT halten, RES tippen und dann in der IDE den Port des Bootmodus wählen und einen funktionierenden Sketch hochladen (Kapitel 2.5). Alternativ kann sie eine `.uf2`-Datei, etwa MicroPython, auf das Laufwerk ziehen.

Aufgabe 5 – Kern oder Bibliothek?

- b) „Raspberry Pi Pico/RP2040/RP2350“ ist ein **Kern**: Er bringt der IDE einen ganzen Mikrocontroller bei, mit Compiler, den Grundbefehlen wie `digitalWrite` und dem Werkzeug zum Hochladen.
- a), c) und d) sind **Bibliotheken**: Sie ergänzen Befehle für ein bestimmtes Bauteil – a) die RGB-LED, c) den Neigungssensor, d) das Display. Sie funktionieren auf vielen verschiedenen Mikrocontrollern.

Lösungen zu Kapitel 2: Die Arduino IDE einrichten

Aufgabe 1 – Ein anderer Gruß

Nur der Text in Anführungszeichen ändert sich:

```
tft.print("Moin!");
```

„Moin!“ ist ein Zeichen kürzer als „Hallo!“ und steht deshalb etwas links von der Mitte. Wie man die Mitte ausrechnet, steht in Kapitel 3.4.

Aufgabe 2 – Wie lange dauert es?

Das erste Überprüfen dauert deutlich länger als das zweite. Beim ersten Mal übersetzt der Compiler neben deinem Sketch auch den Kern und die Bibliotheken. Die Ergebnisse hebt die IDE auf. Danach übersetzt sie nur noch, was sich geändert hat – hier also nur deinen Sketch – und fügt alles neu zusammen. (Wechselst du das Board oder eine Einstellung im Menü **Werkzeuge**, wird wieder alles übersetzt.)

Aufgabe 3 – Hin und zurück

Solange MicroPython läuft, funktioniert nur **Weg 1**: BOOT halten, RES tippen, Port **UF2 Board** wählen, hochladen. Der serielle Port, den MicroPython anbietet, nützt nichts: MicroPython hört nicht auf das Signal, mit dem die IDE einen Arduino-Sketch in den Bootmodus schaltet. Nach dem Hochladen wählst du wieder den seriellen Port des Sketches; dann funktioniert **Weg 2**.

Aufgabe 4 – Wo liegt was?

Den Pfad zeigt **Datei → Einstellungen…** unter **Pfad für Sketchbook**. Im Unterordner `libraries` liegen bei der Einrichtung aus Kapitel 2 diese Ordner:

- `Adafruit_ST7735_and_ST7789_Library`, `Adafruit_NeoPixel`, `STK8BA58` – die drei ausgewählten Bibliotheken,
- `Adafruit_GFX_Library`, `Adafruit_BusIO`, `Adafruit_seesaw_Library` und `SD` – die Abhängigkeiten.

Die Display-Bibliothek gibt an, welche anderen Bibliotheken sie braucht, und diese wiederum nennen ihre eigenen. Mit **Alle installieren** hat der Bibliotheksverwalter die ganze Kette installiert. Liegen weitere Ordner darin, stammen sie aus früheren Projekten.

Lösungen zu Kapitel 3: Der erste Sketch

Aufgabe 1 – Wo erscheint was?

„oben“ steht in Weiß und Größe 2 links oben (x = 30, y = 30). „Mitte“ steht in Grün und Größe 3 genau in der Mitte: 5 Zeichen × 6 × 3 = 90 Pixel breit, x = (240 − 90) / 2 = 75; 24 Pixel hoch, y = (280 − 24) / 2 = 128. „unten“ steht in Rot und Größe 2 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`)

k03_loesung2.ino

```
10 void setup() {
11     tft.init(240, 280);
12     tft.fillScreen(ST77XX_BLACK);
13     pinMode(TFT_LICHT, OUTPUT);
14     digitalWrite(TFT_LICHT, HIGH);
15
16     tft.setTextSize(4);
17     tft.setTextColor(ST77XX_YELLOW);
18     tft.setCursor(30, 60);
19     tft.print("Ben");
20     tft.setTextSize(2);
21     tft.setTextColor(ST77XX_CYAN);
22     tft.setCursor(30, 120);
23     tft.print("15 Jahre alt");
24     tft.setTextColor(ST77XX_GREEN);
25     tft.setCursor(30, 150);
26     tft.print("spielt Fussball");
27 }
```



Bei Größe 4 ist der Name 32 Pixel hoch (y = 60 bis 91). Die erste Angabe beginnt deshalb erst bei y = 120, die zweite 30 Pixel darunter.

Aufgabe 3 – Finde die Fehler (`k03_loesung3`)

Der Compiler meldet alle **drei** Fehler auf einmal:

Ausgabe

```
k03_aufgabe3.ino: In function 'void setup()':
k03_aufgabe3.ino:17:9: error: 'class Adafruit_ST7789' has no member named
  'setTextcolor'; did you mean 'setTextColor'?
   17 |     tft.setTextcolor(ST77XX_YELLOW);
```

```
      |         ^~~~~~~~~~~
      |         setTextColor
k03_aufgabe3.ino:19:25: error: expected ';' before 'tft'
   19 |     tft.print("Treffen")
      |                         ^
      |                         ;
   20 |     tft.setTextSize(2);
      |     ~~~
k03_aufgabe3.ino:21:22: error: 'ST77XX_white' was not declared in this scope; did you
  mean 'ST77XX_WHITE'?
   21 |     tft.setTextColor(ST77XX_white);
      |                      ^~~~~~~~~~~
      |                      ST77XX_WHITE
Fehler beim Build: exit status 1
```

1. Zeile 17: `setTextcolor` gibt es nicht, richtig ist `setTextColor` mit großem C. Der Compiler schlägt es vor.
2. Zeile 19: Am Ende fehlt das Semikolon. Gemeldet wird es vor dem `tft` der nächsten Zeile.
3. Zeile 21: `ST77XX_white` muss `ST77XX_WHITE` heißen.

Nicht immer findet der Compiler alle Fehler im ersten Durchgang. Manchmal verdeckt ein Fehler den nächsten, oder ein einziger Fehler erzeugt mehrere Meldungen. Deshalb gilt: den ersten Fehler verbessern, neu kompilieren.

k03_loesung3.ino

```
16      tft.setTextSize(3);
17      tft.setTextColor(ST77XX_YELLOW);
18      tft.setCursor(40, 80);
19      tft.print("Treffen");
20      tft.setTextSize(2);
21      tft.setTextColor(ST77XX_WHITE);
22      tft.setCursor(40, 130);
23      tft.print("Heute 14 Uhr");
24      tft.setCursor(40, 160);
25      tft.print("Raum 2.04");
26  }
```



Aufgabe 4 – Ein Schild (`k03_loesung4`)

k03_loesung4.ino

```
10  void setup() {
11      tft.init(240, 280);
12      tft.fillScreen(ST77XX_RED);
13      pinMode(TFT_LICHT, OUTPUT);
14      digitalWrite(TFT_LICHT, HIGH);
15
16      // 4 Zeichen * 6 * 6 = 144 Pixel breit, 8 * 6 = 48 Pixel hoch
17      tft.setTextSize(6);
18      tft.setTextColor(ST77XX_WHITE);
19      tft.setCursor(48, 116);
20      tft.print("STOP");
```

```
    // 12 Zeichen * 6 * 2 = 144 Pixel breit
    tft.setTextSize(2);
    tft.setCursor(48, 190);
    tft.print("Bitte warten");
}
```

„STOP“ hat 4 Zeichen, bei Größe 6 ist jedes 36 Pixel breit und 48 Pixel hoch. Breite 4 × 36 = 144, also x = (240 − 144) / 2 = 48; y = (280 − 48) / 2 = 116. „Bitte warten“ hat 12 Zeichen zu je 12 Pixeln, ist also ebenfalls 144 Pixel breit und beginnt auch bei x = 48. Die Zahl y = 190 ist frei gewählt: deutlich unter dem „STOP“, das bei y = 164 endet.



Aufgabe 5 – Stundenplan (`k03_loesung5`)

k03_loesung5.ino

```
    tft.setTextSize(3);
    tft.setTextColor(ST77XX_YELLOW);
    tft.setCursor(30, 30);
    tft.print("Montag");

    // Groesse 2: jedes Zeichen 12 Pixel breit, 16 Pixel hoch
    tft.setTextSize(2);
    tft.setTextColor(ST77XX_CYAN);
    tft.setCursor(24, 76);
    tft.print("# Fach     Raum");
    tft.setTextColor(ST77XX_WHITE);
    tft.setCursor(24, 100);
    tft.print("1 Mathe    204");
    tft.setCursor(24, 120);
    tft.print("2 Mathe    204");
    tft.setCursor(24, 140);
    tft.print("3 Englisch 112");
    tft.setCursor(24, 160);
    tft.print("4 Physik   N3");
    tft.setCursor(24, 180);
    tft.print("5 Sport    Halle");
    tft.setCursor(24, 200);
    tft.print("6 Deutsch  108");
}
```

Die eingebaute Schrift hat feste Breite: Jedes Zeichen, auch das Leerzeichen, ist bei Größe 2 genau 12 Pixel breit. Wenn die Stunde mit Leerzeichen 2 Zeichen belegt und das Fach mit Leerzeichen 9, beginnen alle Spalten genau untereinander. Bleiben links und rechts je 20 Pixel frei, sind 200 Pixel nutzbar, also höchstens 200 / 12 = 16 Zeichen pro Zeile („5 Sport Halle“ hat genau 16). Die Zeichen sind 16 Pixel hoch, der Zeilenabstand muss also mindestens 16 Pixel sein; mit 20 Pixeln ist es gut lesbar.



Lösungen zu Kapitel 4: Der serielle Monitor

Aufgabe 1 – Ein Gedicht (`k04_loesung1`)

k04_loesung1.ino

```
1 void setup() {
2     Serial.begin(115200);
3     while (!Serial) {
4         delay(10);
5     }
6     Serial.println("Rosen sind rot,\nVeilchen sind blau,\nder PicoBoy
        rechnet\nund du bist schlau.");
7 }
8
9 void loop() {
10 }
```

Serieller Monitor

```
Rosen sind rot,
Veilchen sind blau,
der PicoBoy rechnet
und du bist schlau.
```

Drei `\n` im Text ergeben vier Zeilen; den letzten Zeilenumbruch hängt `println` an.

Aufgabe 2 – Zeit in Zahlen (`k04_loesung2`)

k04_loesung2.ino

```
6     Serial.printf("Eine Woche hat %d Stunden.\n", 7 * 24);
7     Serial.printf("Ein Tag hat %d Minuten.\n", 24 * 60);
8     Serial.printf("Ein Tag hat %d Sekunden.\n", 24 * 60 * 60);
```

Serieller Monitor

```
Eine Woche hat 168 Stunden.
Ein Tag hat 1440 Minuten.
Ein Tag hat 86400 Sekunden.
```

Aufgabe 3 – Steckbrief (`k04_loesung3`)

k04_loesung3.ino

```
    Serial.println("Wie heisst du?");
    while (Serial.available() == 0) {
        delay(10);
    }
    String name = Serial.readStringUntil('\n');
    name.trim();

    Serial.println("Was ist deine Lieblingsfarbe?");
    while (Serial.available() == 0) {
        delay(10);
    }
    String farbe = Serial.readStringUntil('\n');
    farbe.trim();

    Serial.println("Wie alt bist du?");
    while (Serial.available() == 0) {
        delay(10);
    }
    int alter = Serial.parseInt();
    Serial.readStringUntil('\n');

    Serial.print(name);
    Serial.printf(" ist %d Jahre alt und mag am liebsten ", alter);
    Serial.print(farbe);
    Serial.println(".");
```

Serieller Monitor

```
Wie heisst du?
Was ist deine Lieblingsfarbe?
Wie alt bist du?
Ben ist 14 Jahre alt und mag am liebsten Gruen.
```

Eingegeben wurden `Ben`, `Gruen` und `14`. Jeder Text braucht seine eigene Variable. Den Satz setzt der Sketch aus `print` und `printf` zusammen; wie ein `String` direkt in `printf` passt, zeigt Aufgabe 4.

Aufgabe 4 – Zwei Fehler, keine Meldung (`k04_loesung4`)

Beide Fehler stecken in den Platzhaltern, und `Serial.printf` prüft sie nicht (Kapitel 4.2):

1. Die Temperatur `21` ist eine ganze Zahl, der Platzhalter muss also `%d` heißen, nicht `%f`.
2. `name` ist ein `String`. `%s` erwartet aber einen einfachen Text, wie er in Anführungszeichen steht. Den liefert der `String` mit dem Zusatz `.c_str()`: `name.c_str()`. Ohne ihn gibt der Sketch Zeichensalat aus oder stürzt sogar ab. Alternativ schreibt man den Namen mit `Serial.print(name)` aus.

k04_loesung4.ino

```
6     String name = "Mia";
7     Serial.printf("Temperatur: %d Grad\n", 21);
8     Serial.printf("Name: %s\n", name.c_str());
```

Serieller Monitor

```
Temperatur: 21 Grad
Name: Mia
```

Aufgabe 5 – Taschenrechner (`k04_loesung5`)

k04_loesung5.ino

```
6     Serial.println("Erste Zahl:");
7     while (Serial.available() == 0) {
8         delay(10);
9     }
10    float a = Serial.parseFloat();
11    Serial.readStringUntil('\n');
12
13    Serial.println("Zweite Zahl:");
14    while (Serial.available() == 0) {
15        delay(10);
16    }
17    float b = Serial.parseFloat();
18    Serial.readStringUntil('\n');
19
20    Serial.printf("%.2f + %.2f = %.2f\n", a, b, a + b);
21    Serial.printf("%.2f - %.2f = %.2f\n", a, b, a - b);
22    Serial.printf("%.2f * %.2f = %.2f\n", a, b, a * b);
23    Serial.printf("%.2f / %.2f = %.2f\n", a, b, a / b);
```

Serieller Monitor

```
Erste Zahl:
Zweite Zahl:
7.50 + 2.00 = 9.50
7.50 - 2.00 = 5.50
7.50 * 2.00 = 15.00
7.50 / 2.00 = 3.75
```

Eingegeben wurden `7.5` und `2`. Ist die zweite Zahl 0, ergibt die Division `inf` (*infinity*, unendlich):

Serieller Monitor

```
Erste Zahl:
Zweite Zahl:
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 der Sketch also nicht ab, sondern rechnet mit einem besonderen Wert „unendlich“ weiter. Mit ganzen Zahlen ist das anders; das zeigt Kapitel 5.

Mit `7,5` und `2` kommt dagegen Folgendes heraus:

```
Serieller Monitor
Erste Zahl:
Zweite Zahl:
7.00 + 2.00 = 9.00
7.00 - 2.00 = 5.00
7.00 * 2.00 = 14.00
7.00 / 2.00 = 3.50
```

`parseFloat` liest `7` und hört beim Komma auf. Den Rest `,5` wirft `readStringUntil` mit dem Zeilenende weg. Der Sketch rechnet also still mit 7 weiter – eine Fehlermeldung gibt es nicht.

Lösungen zu Kapitel 5: Variablen und Datentypen

Aufgabe 1 – Welcher Typ passt?

Was	Typ	Warum
Alter einer Person	`int`	eine ganze Zahl
Preis eines Brötchens	`float`	hat Nachkommastellen
ob das Spiel vorbei ist	`bool`	nur ja oder nein
Anfangsbuchstabe	`char`	genau ein Zeichen
Zeit seit dem Start in ms	`unsigned long`	das liefert `millis()`
Helligkeit 0 bis 255	`byte`	passt genau, `int` ginge auch
Einwohner Deutschlands	`int`	rund 84 Millionen, `int` reicht bis gut 2 Milliarden

Beim Preis gibt es eine Alternative, die in der Praxis oft besser ist: den Preis in **Cent** als `int` speichern (249 statt 2.49). Ganze Zahlen rechnen immer genau, Kommazahlen nicht immer.

Aufgabe 2 – Was kommt heraus?

```
Serieller Monitor
3
1
3.50
3.00
18
```

1. `7 / 2` ist ganzzahlig 3, der Rest fällt weg.
2. `7 % 2` ist der Rest: 1.
3. `7 / 2.0` rechnet mit Komma: 3.50.
4. `float c = a / b;` rechnet zuerst ganzzahlig 7 / 2 = 3 und speichert dann 3 als Kommazahl: 3.00. Das ist die Falle aus Abschnitt 5.2.

5. `a += b` macht `a` zu 9, und 9 × 2 ist 18.

Aufgabe 3 – Sekunden umrechnen (`k05_loesung3`)

k05_loesung3.ino

```
 7     int gesamt = 100000;
 8
 9     int tage = gesamt / 86400;          // 86400 Sekunden hat ein Tag
10     int rest = gesamt % 86400;
11     int stunden = rest / 3600;
12     rest = rest % 3600;
13     int minuten = rest / 60;
14     int sekunden = rest % 60;
15
16     Serial.printf("%d Sekunden sind %d Tag(e), %d Stunden, %d Minuten und %d
         Sekunden.\n",
17                   gesamt, tage, stunden, minuten, sekunden);
```

Serieller Monitor

```
100000 Sekunden sind 1 Tag(e), 3 Stunden, 46 Minuten und 40 Sekunden.
```

Ein Tag hat 24 × 60 × 60 = 86 400 Sekunden. Die ganzzahlige Division `gesamt / 86400` zählt die vollen Tage, `gesamt % 86400` ist, was übrig bleibt. Mit diesem Rest geht es genauso weiter: durch 3600 für die Stunden, durch 60 für die Minuten. Ein `printf` darf über zwei Zeilen gehen; die Werte stehen dann in der zweiten.

Aufgabe 4 – Zufall in Bereichen (`k05_loesung4`)

k05_loesung4.ino

```
 7     int muenze = random(2);                 // 0 oder 1
 8     int w20 = random(1, 21);                // 1 bis 20
 9     int x = random(0, 221);                 // 0 bis 220, ein 20 Pixel breiter
         Ball passt dann noch
10     int temperatur = random(-5, 31);        // -5 bis 30
11
12     Serial.printf("Muenze: %d\n", muenze);
13     Serial.printf("W20: %d\n", w20);
14     Serial.printf("x: %d\n", x);
15     Serial.printf("Temperatur: %d Grad\n", temperatur);
```

Serieller Monitor

```
Muenze: 1
W20: 4
x: 198
Temperatur: 0 Grad
```

Die obere Grenze ist immer um eins größer als die größte gewünschte Zahl. Beim Ball: Er ist 20 Pixel breit. Beginnt er bei x = 220, reicht er bis x = 239, dem letzten Pixel. Die größte erlaubte Position ist also 220, und die Grenze muss 221 heißen. Bei dir können andere Zahlen stehen; weil `randomSeed` fehlt, sind es aber bei jedem Start dieselben.

Aufgabe 5 – Wann läuft die Uhr über? (`k05_loesung5`)

k05_loesung5.ino

```
7     unsigned long groesste = 4294967295;    // groesster Wert von unsigned long
8     int groesste_int = 2147483647;          // groesster Wert von int
9
10    unsigned long s = groesste / 1000;      // in Sekunden
11    Serial.printf("unsigned long: %lu Tage, %lu Stunden, %lu Minuten, %lu
        Sekunden\n",
12                  s / 86400, s % 86400 / 3600, s % 3600 / 60, s % 60);
13
14    int t = groesste_int / 1000;
15    Serial.printf("int: %d Tage, %d Stunden, %d Minuten, %d Sekunden\n",
16                  t / 86400, t % 86400 / 3600, t % 3600 / 60, t % 60);
```

Serieller Monitor

```
unsigned long: 49 Tage, 17 Stunden, 2 Minuten, 47 Sekunden
int: 24 Tage, 20 Stunden, 31 Minuten, 23 Sekunden
```

- Erst werden die Millisekunden in Sekunden umgerechnet, dann wie in Aufgabe 3 zerlegt. Die Rechnungen stehen direkt im `printf`.
- `%lu` ist der Platzhalter für `unsigned long` (Kapitel 7.1).
- `s % 86400 / 3600`: `%` und `/` haben denselben Rang und werden von links nach rechts gerechnet, erst der Rest des Tages, dann die Stunden darin.

`millis()` läuft also nach gut 49 Tagen über. Wer die Zeit in einem `int` speichert, bekommt schon nach knapp 25 Tagen negative Zahlen, denn ein `int` reicht nur halb so weit.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 6: Zeichnen und Farben

Aufgabe 1 – Eine Flagge (`k06_loesung1`)

k06_loesung1.ino

```
17    tft.fillScreen(tft.color565(90, 90, 90));
18    tft.fillRect(30, 80, 180, 40, ST77XX_BLACK);
19    tft.fillRect(30, 120, 180, 40, tft.color565(221, 0, 0));
20    tft.fillRect(30, 160, 180, 40, tft.color565(255, 206, 0));
```

Die drei Streifen sind je 40 Pixel hoch und liegen direkt übereinander: Der rote beginnt bei y = 120, genau da, wo der schwarze (y = 80, Höhe 40) aufhört. Der graue Grund sorgt dafür, dass man den schwarzen Streifen sieht.



Aufgabe 2 – Was wird das?

Ein Schneemann im Schnee: blauer Himmel, unten ein weißer Streifen als Boden, drei weiße Kreise übereinander (der größte unten), ein schwarzer Zylinderhut aus zwei Rechtecken, zwei schwarze Augen, eine orange Karottennase als Dreieck und zwei schwarze Knöpfe.

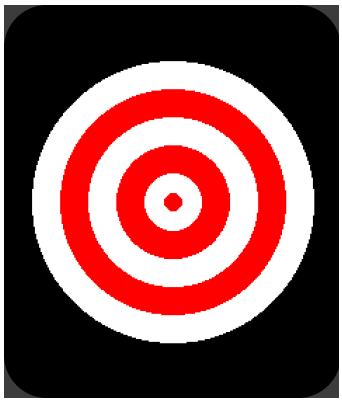


Der Hut steht auf dem Kopf, weil seine Krempe (y = 44 bis 51) in den Kreis des Kopfs (y = 51 bis 99) hineinragt. Die Nase wird erst nach dem Kopf gezeichnet, sonst wäre sie verdeckt.

Aufgabe 3 – Zielscheibe (`k06_loesung3`)

k06_loesung3.ino

```
17      tft.fillCircle(120, 140, 100, ST77XX_WHITE);
18      tft.fillCircle(120, 140, 80, ST77XX_RED);
19      tft.fillCircle(120, 140, 60, ST77XX_WHITE);
20      tft.fillCircle(120, 140, 40, ST77XX_RED);
21      tft.fillCircle(120, 140, 20, ST77XX_WHITE);
22      tft.fillCircle(120, 140, 6, ST77XX_RED);
```

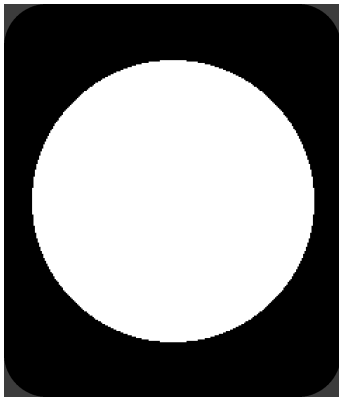


Jeder Kreis ist 20 Pixel kleiner als der vorige und liegt auf ihm. Vom größeren bleibt deshalb nur ein 20 Pixel breiter Ring sichtbar.

In umgekehrter Reihenfolge, vom kleinsten zum größten, übermalt jeder Kreis alle vorigen vollständig. Am Ende ist nur der letzte zu sehen, eine weiße Scheibe:

k06_loesung3_falsch.ino

```
17      tft.fillCircle(120, 140, 6, ST77XX_RED);
18      tft.fillCircle(120, 140, 20, ST77XX_WHITE);
19      tft.fillCircle(120, 140, 40, ST77XX_RED);
20      tft.fillCircle(120, 140, 60, ST77XX_WHITE);
21      tft.fillCircle(120, 140, 80, ST77XX_RED);
22      tft.fillCircle(120, 140, 100, ST77XX_WHITE);
```



Aufgabe 4 – Namensschild (`k06_loesung4`)

k06_loesung4.ino

```
1 // Ein Namensschild: Name in der Mitte eines Schilds
2 #include <Adafruit_ST7789.h>
3 #include <Fonts/FreeSansBold18pt7b.h>
```

k06_loesung4.ino

```
18    // das Schild: x = 20, y = 100, 200 breit, 80 hoch
19    tft.fillRoundRect(20, 100, 200, 80, 15, tft.color565(0, 90, 160));
20
21    tft.setFont(&FreeSansBold18pt7b);
22    tft.setTextColor(ST77XX_WHITE);
23    int16_t x1, y1;
24    uint16_t w, h;
25    tft.getTextBounds("Mia", 0, 0, &x1, &y1, &w, &h);
26    int x = 20 + (200 - w) / 2 - x1;
27    int y = 100 + (80 - h) / 2 - y1;
28    tft.setCursor(x, y);
29    tft.print("Mia");
```

Die Rechnung ist dieselbe wie in `k06_zentriert`, nur nicht für das ganze Display, sondern für das Schild: Statt 240 steht dort die Breite des Schilds (200), statt 280 seine Höhe (80). Weil das Schild nicht bei (0, 0) beginnt, kommt seine linke obere Ecke (20, 100) dazu.

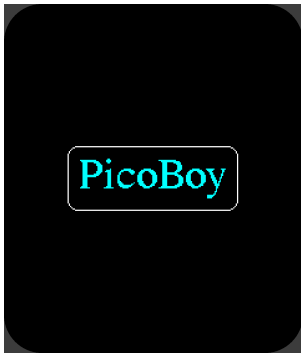


Aufgabe 5 – Text mit Rahmen (`k06_loesung5`)

k06_loesung5.ino

```
18    tft.setFont(&FreeSerif18pt7b);
19    tft.setTextColor(ST77XX_CYAN);
20
21    int16_t x1, y1;
22    uint16_t w, h;
23    tft.getTextBounds("PicoBoy", 0, 0, &x1, &y1, &w, &h);
24    int links = (240 - w) / 2;          // linke Kante des Texts
25    int oben = (280 - h) / 2;           // obere Kante des Texts
26
27    tft.setCursor(links - x1, oben - y1);
28    tft.print("PicoBoy");
29    tft.drawRoundRect(links - 10, oben - 10, w + 20, h + 20, 8, ST77XX_WHITE);
```

- `links` und `oben` (Zeilen 24 und 25) sind die Kanten des Texts auf dem Display. Der Cursor kommt dann nach `links - x1` und `oben - y1`, wie im Kapitel.
- Der Rahmen beginnt 10 Pixel links und oberhalb des Texts. Er ist um 2 × 10 Pixel breiter und höher als der Text, denn links **und** rechts sollen je 10 Pixel frei bleiben.
- Ersetzt du „PicoBoy“ in den Zeilen 23 und 28 durch ein anderes Wort, passt sich der Rahmen von selbst an.



Lösungen zu Kapitel 7: Schleifen und Zeit

Aufgabe 1 – Countdown (`k07_loesung1`)

k07_loesung1.ino

```
 7     for (int i = 10; i >= 1; i--) {
 8         Serial.println(i);
 9         delay(1000);
10     }
11     Serial.println("Start!");
```

Serieller Monitor

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

Die Schleife zählt rückwärts: Anfang 10, Bedingung `i >= 1`, Schritt `i--`. Weil der Countdown nur einmal laufen soll, steht er in `setup()`.

Aufgabe 2 – Einmaleins (`k07_loesung2`)

k07_loesung2.ino

```
 7     for (int i = 1; i <= 10; i++) {
 8         Serial.printf("%d x 7 = %d\n", i, i * 7);
 9     }
```

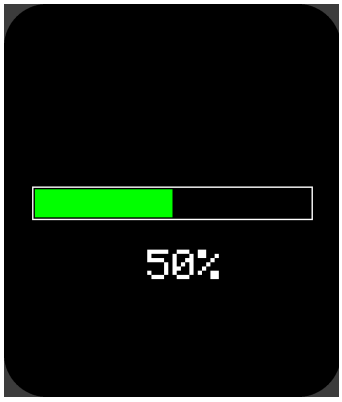
Serieller Monitor

```
1 x 7 = 7
2 x 7 = 14
3 x 7 = 21
4 x 7 = 28
5 x 7 = 35
6 x 7 = 42
7 x 7 = 49
8 x 7 = 56
9 x 7 = 63
10 x 7 = 70
```

Aufgabe 3 – Ladebalken (`k07_loesung3`)

k07_loesung3.ino

```
17      tft.drawRect(20, 130, 200, 24, ST77XX_WHITE);
18      tft.setTextSize(3);
19      tft.setTextColor(ST77XX_WHITE, ST77XX_BLACK);
20
21      for (int prozent = 0; prozent <= 100; prozent++) {
22          int breite = prozent * 196 / 100;          // innen ist Platz fuer 196
              Pixel
23          tft.fillRect(22, 132, breite, 20, ST77XX_GREEN);
24          tft.setCursor(84, 175);
25          tft.printf("%3d%%", prozent);
26          delay(50);
27      }
```



- Die Schleife zählt von 0 bis 100 Prozent. 101 Durchgänge mit je 50 ms `delay` ergeben gut 5 Sekunden.
- Innen im Rahmen ist 196 Pixel Platz (200 minus je 2 Pixel links und rechts). Zeile 22 rechnet Prozent in Pixel um, erst mal, dann geteilt (Kapitel 5.2).
- Jeder Durchgang malt den ganzen Balken bis zur aktuellen Breite neu. Das ist einfach und flackert nicht, weil sich der grüne Teil nur verlängert.
- `%3d` füllt die Zahl auf drei Stellen auf, `%%` ist das Prozentzeichen. Zusammen mit der Hintergrundfarbe aus Zeile 19 überschreibt jede neue Zahl die alte vollständig.

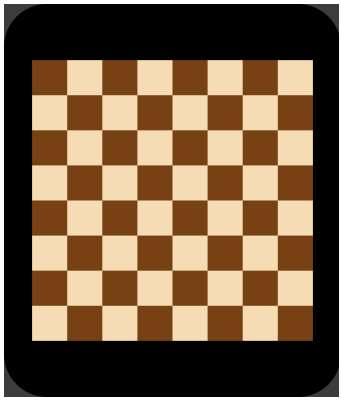
Das Bild zeigt den Balken nach etwa 3 Sekunden.

Aufgabe 4 – Schachbrett (`k07_loesung4`)

k07_loesung4.ino

```
17      uint16_t hell = tft.color565(240, 220, 180);
18      uint16_t dunkel = tft.color565(120, 70, 30);
19
20      tft.fillRect(20, 40, 200, 200, hell);
21      for (int zeile = 0; zeile < 8; zeile++) {
22          for (int spalte = zeile % 2; spalte < 8; spalte += 2) {
23              tft.fillRect(20 + spalte * 25, 40 + zeile * 25, 25, 25, dunkel);
24          }
25      }
```

Zuerst füllt Zeile 20 das ganze Brett hell. Dann zeichnet die innere Schleife nur jedes zweite Feld (`spalte += 2`) dunkel. Der Trick steckt im Anfang: `spalte = zeile % 2` ist in geraden Zeilen 0 und in ungeraden 1. So beginnt jede Zeile um ein Feld versetzt, und es entsteht das Schachmuster.



Aufgabe 5 – Bilder pro Sekunde (`k07_loesung5`)

k07_loesung5.ino

```
11 unsigned long letzte_ausgabe = 0;
12 int bilder = 0;
13 int x = 30;
14
15 void setup() {
16     Serial.begin(115200);
17     tft.init(240, 280);
18     tft.fillScreen(ST77XX_BLACK);
19     pinMode(TFT_LICHT, OUTPUT);
20     digitalWrite(TFT_LICHT, HIGH);
21 }
22
23 void loop() {
24     tft.fillScreen(ST77XX_BLACK);
25     tft.fillCircle(x, 140, 20, ST77XX_ORANGE);
26     x = 30 + (x - 30 + 4) % 184;        // 30 bis 210, dann wieder links
27     bilder++;
28
29     if (millis() - letzte_ausgabe >= 1000) {
30         letzte_ausgabe = millis();
31         Serial.printf("%d Bilder pro Sekunde\n", bilder);
32         bilder = 0;
33     }
34 }
```

Serieller Monitor

```
15 Bilder pro Sekunde
23 Bilder pro Sekunde
23 Bilder pro Sekunde
23 Bilder pro Sekunde
23 Bilder pro Sekunde
```

- `bilder` zählt in jedem Durchlauf von `loop()` eins hoch. Einmal pro Sekunde gibt der `millis`-Takt die Zahl aus und setzt sie auf 0 zurück.
- Statt einer `for`-Schleife bewegt Zeile 26 den Ball: `% 184` lässt die Position nach 210 wieder bei 30 anfangen.

- Ein Bild dauert rund 43 ms für `fillScreen` und etwa 1 ms für den Kreis. 1000 / 44 ergibt knapp 23 Bilder pro Sekunde, genau was die Zeitleiste in Abschnitt 7.3 erwarten lässt. Im ersten Takt sind es weniger, weil der Start des Displays darin steckt.

Lösungen zu Kapitel 8: Entscheidungen

Aufgabe 1 – Gerade oder ungerade (`k08_loesung1`)

k08_loesung1.ino

```
    for (int zahl = 1; zahl <= 10; zahl++) {
        if (zahl % 2 == 0) {
            Serial.printf("%d ist gerade\n", zahl);
        } else {
            Serial.printf("%d ist ungerade\n", zahl);
        }
    }
```

Serieller Monitor

```
1 ist ungerade
2 ist gerade
3 ist ungerade
4 ist gerade
5 ist ungerade
6 ist gerade
7 ist ungerade
8 ist gerade
9 ist ungerade
10 ist gerade
```

Aufgabe 2 – Was kommt heraus?

Serieller Monitor

```
angenehm
```

Bei 18 stimmt `temperatur > 25` nicht, `temperatur > 15` aber schon: „angenehm“. Bei **25** kommt ebenfalls „angenehm“ heraus, denn 25 ist nicht **größer** als 25; dafür müsste dort `>=` stehen. Bei **5** stimmt keine der drei Bedingungen (5 ist nicht größer als 5), also läuft das `else`: „kalt“.

Aufgabe 3 – Schaltjahr (`k08_loesung3`)

k08_loesung3.ino

```
void loop() {
    Serial.println("Welches Jahr?");
    while (Serial.available() == 0) {
        delay(10);
    }
    int jahr = Serial.parseInt();
```

```
15      Serial.readStringUntil('\n');
16
17      if ((jahr % 4 == 0 && jahr % 100 != 0) || jahr % 400 == 0) {
18          Serial.printf("%d ist ein Schaltjahr.\n", jahr);
19      } else {
20          Serial.printf("%d ist kein Schaltjahr.\n", jahr);
21      }
22  }
```

Serieller Monitor

```
Welches Jahr?
2024 ist ein Schaltjahr.
Welches Jahr?
2025 ist kein Schaltjahr.
Welches Jahr?
1900 ist kein Schaltjahr.
Welches Jahr?
2000 ist ein Schaltjahr.
Welches Jahr?
```

Eingegeben wurden `2024`, `2025`, `1900` und `2000`.

„Teilbar durch 4“ heißt: Der Rest bei der Division durch 4 ist 0, also `jahr % 4 == 0`. Die Bedingung in Zeile 17 liest sich so: (durch 4 teilbar **und** nicht durch 100) **oder** durch 400 teilbar. Die Klammern sind hier nicht nötig, weil `&&` ohnehin vor `||` kommt, machen die Regel aber lesbarer.

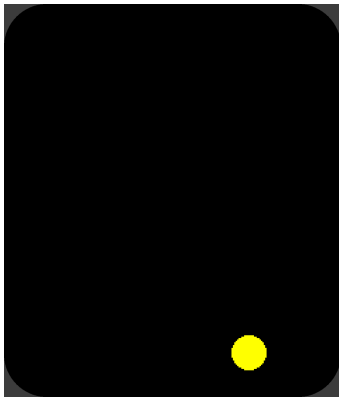
Aufgabe 4 – Hüpfender Ball (`k08_loesung4`)

k08_loesung4.ino

```
 9  const int RADIUS = 12;
10
11  Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
12
13  int x = 60;
14  int y = 100;
15  int dx = 3;         // Schritt nach rechts (negativ: nach links)
16  int dy = 2;         // Schritt nach unten (negativ: nach oben)
17
18  void setup() {
19      tft.init(240, 280);
20      tft.fillScreen(ST77XX_BLACK);
21      pinMode(TFT_LICHT, OUTPUT);
22      digitalWrite(TFT_LICHT, HIGH);
23  }
24
25  void loop() {
26      tft.fillCircle(x, y, RADIUS, ST77XX_BLACK);     // alten Ball loeschen
27      x += dx;
28      y += dy;
29      if (x - RADIUS <= 0 || x + RADIUS >= 239) {
30          dx = -dx;
31      }
```

```
32      if (y - RADIUS <= 0 || y + RADIUS >= 279) {
33          dy = -dy;
34      }
35      tft.fillCircle(x, y, RADIUS, ST77XX_YELLOW);
36      delay(15);
37  }
```

- `x`, `y`, `dx` und `dy` stehen außerhalb von `loop()`, damit der Ball seine Position und Richtung behält.
- Zeile 29 prüft den linken und den rechten Rand mit `||`. Der linke Rand des Balls ist `x - RADIUS`, der rechte `x + RADIUS`. Berührt einer davon den Bildschirmrand, kehrt `dx = -dx` die Richtung um: aus 3 wird −3 und umgekehrt.
- Zeile 32 macht dasselbe für oben und unten.
- Wegen der runden Ecken verschwindet der Ball kurz teilweise, wenn er genau in eine Ecke fliegt.

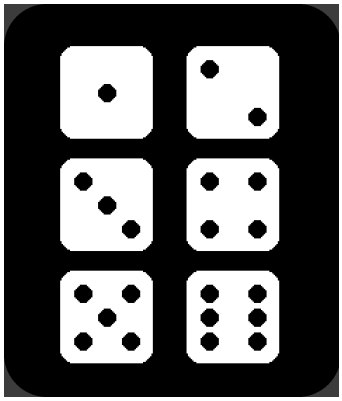


Aufgabe 5 – Würfelaugen (`k08_loesung5`)

k08_loesung5.ino

```
17      for (int augen = 1; augen <= 6; augen++) {
18          // linke obere Ecke des Wuerfels: zwei Spalten, drei Zeilen
19          int x = 40 + (augen - 1) % 2 * 90;
20          int y = 30 + (augen - 1) / 2 * 80;
21          tft.fillRoundRect(x, y, 66, 66, 10, ST77XX_WHITE);
22
23          int links = x + 16;
24          int mitte = x + 33;
25          int rechts = x + 50;
26          int oben = y + 16;
27          int unten = y + 50;
28          if (augen % 2 == 1) {                   // 1, 3, 5: Auge in der Mitte
29              tft.fillCircle(mitte, y + 33, 6, ST77XX_BLACK);
30          }
31          if (augen >= 2) {                       // links oben und rechts unten
32              tft.fillCircle(links, oben, 6, ST77XX_BLACK);
33              tft.fillCircle(rechts, unten, 6, ST77XX_BLACK);
34          }
35          if (augen >= 4) {                       // rechts oben und links unten
36              tft.fillCircle(rechts, oben, 6, ST77XX_BLACK);
37              tft.fillCircle(links, unten, 6, ST77XX_BLACK);
38          }
39          if (augen == 6) {                       // links und rechts in der Mitte
40              tft.fillCircle(links, y + 33, 6, ST77XX_BLACK);
41              tft.fillCircle(rechts, y + 33, 6, ST77XX_BLACK);
42          }
43      }
```

- Zeilen 19 und 20 verteilen die sechs Würfel auf zwei Spalten und drei Zeilen: `% 2` wechselt zwischen 0 und 1 (Spalte), `/ 2` zählt 0, 0, 1, 1, 2, 2 (Zeile).
- Statt sechs Fälle einzeln zu zeichnen, genügen vier Bedingungen, weil sich die Augen überschneiden:
  - Die ungeraden Zahlen 1, 3, 5 haben ein Auge in der Mitte.
  - Ab der 2 gibt es immer links oben und rechts unten ein Auge.
  - Ab der 4 kommen rechts oben und links unten dazu.
  - Nur die 6 hat zusätzlich die beiden Augen in der Mitte links und rechts.



- Die vier `if` stehen nacheinander, nicht als `else if`, denn bei einer 5 sollen gleich drei davon zutreffen.

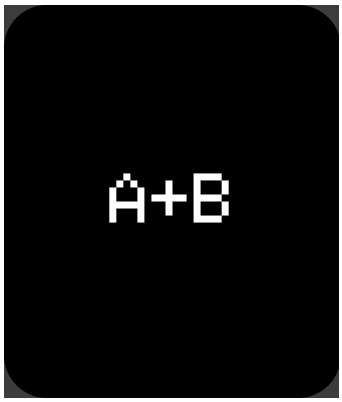
ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 9: Tasten und Joystick

Aufgabe 1 – Zwei Tasten (`k09_loesung1`)

k09_loesung1.ino

```
13 void setup() {
14     tft.init(240, 280);
15     pinMode(TFT_LICHT, OUTPUT);
16     digitalWrite(TFT_LICHT, HIGH);
17     tft.fillScreen(ST77XX_BLACK);
18     pinMode(TASTE_A, INPUT_PULLUP);
19     pinMode(TASTE_B, INPUT_PULLUP);
20     tft.setTextSize(5);
21     tft.setTextColor(ST77XX_WHITE, ST77XX_BLACK);
22 }
23
24 void loop() {
25     bool a = digitalRead(TASTE_A) == LOW;
26     bool b = digitalRead(TASTE_B) == LOW;
27     tft.setCursor(75, 120);
28     if (a && b) {
29         tft.print("A+B");
30     } else if (a) {
31         tft.print("A  ");
32     } else if (b) {
33         tft.print("B  ");
34     } else {
35         tft.print("   ");
36     }
37     delay(10);
38 }
```



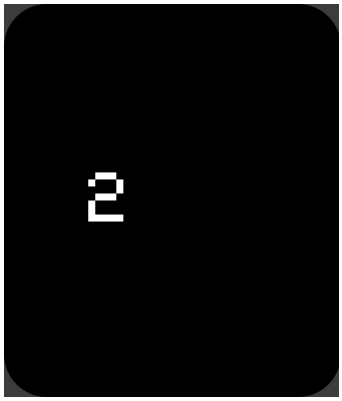
Die Reihenfolge der Fragen ist wichtig: `a && b` muss zuerst kommen. Stünde `if (a)` vorn, wäre diese Bedingung auch dann wahr, wenn beide Tasten gedrückt sind, und „A+B“ käme nie an die Reihe. Kürzere Texte sind mit Leerzeichen auf drei Zeichen aufgefüllt, damit vom „A+B“ nichts stehen bleibt.

Aufgabe 2 – Hoch- und runterzählen (`k09_loesung2`)

k09_loesung2.ino

```
14 int zahl = 0;
15 bool hoch_vorher = false;
16 bool runter_vorher = false;
17 bool mitte_vorher = false;
18
19 void setup() {
20     Serial.begin(115200);
21     tft.init(240, 280);
22     pinMode(TFT_LICHT, OUTPUT);
23     digitalWrite(TFT_LICHT, HIGH);
24     tft.fillScreen(ST77XX_BLACK);
25     pinMode(TASTE_HOCH, INPUT_PULLUP);
26     pinMode(TASTE_RUNTER, INPUT_PULLUP);
27     pinMode(TASTE_MITTE, INPUT_PULLUP);
28     tft.setTextSize(5);
29     tft.setTextColor(ST77XX_WHITE, ST77XX_BLACK);
30 }
31
32 void loop() {
33     bool hoch = digitalRead(TASTE_HOCH) == LOW;
34     bool runter = digitalRead(TASTE_RUNTER) == LOW;
35     bool mitte = digitalRead(TASTE_MITTE) == LOW;
36     if (hoch && !hoch_vorher) {
37         zahl = zahl + 1;
38         Serial.println(zahl);
39     }
40     if (runter && !runter_vorher) {
41         zahl = zahl - 1;
42         Serial.println(zahl);
43     }
44     if (mitte && !mitte_vorher) {
45         zahl = 0;
46         Serial.println(zahl);
47     }
48     hoch_vorher = hoch;
49     runter_vorher = runter;
50     mitte_vorher = mitte;
51
52     tft.setCursor(60, 120);
53     tft.print(zahl);
54     tft.print("   ");       // Reste einer laengeren Zahl ueberschreiben
55     delay(10);
56 }
```

Jede Taste braucht ihre eigene Variable für den Zustand vorher. Damit aus „10“ beim Herunterzählen nicht „90“ wird (die 0 bliebe stehen), schreibt Zeile 54 hinter jede Zahl einige Leerzeichen. Sie übermalen mit der Hintergrundfarbe, was von einer längeren Zahl noch übrig ist. Das hilft auch beim Minuszeichen negativer Zahlen.



Serieller Monitor

```
1
2
3
2
```

Aufgabe 3 – Wie lange gedrückt? (`k09_loesung3`)

k09_loesung3.ino

```ino
// Loesung 9.3: misst, wie lange A gedrueckt wurde
#define TASTE_A 27

bool vorher = false;
unsigned long druck_beginn = 0;

void setup() {
    Serial.begin(115200);
    pinMode(TASTE_A, INPUT_PULLUP);
}

void loop() {
    bool jetzt = digitalRead(TASTE_A) == LOW;
    if (jetzt && !vorher) {             // gerade gedrueckt
        druck_beginn = millis();
    }
    if (!jetzt && vorher) {             // gerade losgelassen
        unsigned long dauer = millis() - druck_beginn;
        Serial.printf("A war %lu ms gedrueckt\n", dauer);
    }
    vorher = jetzt;
    delay(10);
}
```

Serieller Monitor

```
A war 350 ms gedrueckt
A war 1200 ms gedrueckt
```

Beim Drücken merkt sich der Sketch die Zeit, beim Loslassen zieht er sie von der aktuellen Zeit ab. Weil `loop()` nur alle 10 ms nachsieht, ist das Ergebnis auf etwa 10 ms genau.

Aufgabe 4 – Finde die Fehler (`k09_loesung4`)

1. **`pinMode(TASTE_A, INPUT)`** schaltet den Pull-up-Widerstand nicht ein. Losgelassen hängt der Eingang dann in der Luft, und der Sketch liest zufällige Werte. Richtig ist `INPUT_PULLUP`.
2. **`digitalRead(TASTE_A) == HIGH`** fragt „losgelassen?“. Gedrückt ist die Taste bei `LOW`.

3. **`vorher = jetzt;` fehlt.** `vorher` bleibt immer `false`, und der Sketch zählt bei jedem Durchlauf, in dem `jetzt` wahr ist, also hundertmal pro Sekunde statt einmal pro Druck.

Der Compiler findet keinen dieser Fehler: Der Sketch ist richtiges C++, er tut nur nicht das, was gemeint war.

k09_loesung4.ino

```ino
// Loesung 9.4: zaehlt jeden Druck auf A einmal
#define TASTE_A 27

int zaehler = 0;
bool vorher = false;

void setup() {
    Serial.begin(115200);
    pinMode(TASTE_A, INPUT_PULLUP);            // Fehler 1: Pull-up fehlte
}

void loop() {
    bool jetzt = digitalRead(TASTE_A) == LOW;   // Fehler 2: gedrueckt ist LOW
    if (jetzt && !vorher) {
        zaehler = zaehler + 1;
        Serial.println(zaehler);
    }
    vorher = jetzt;                             // Fehler 3: fehlte ganz
    delay(10);
}
```

Aufgabe 5 – Doppelklick (`k09_loesung5`)

k09_loesung5.ino

```ino
// Loesung 9.5: erkennt Einzel- und Doppelklick auf A
#define TASTE_A 27
#define DOPPELKLICK_ZEIT 400    // ms zwischen zwei Druecken

bool vorher = false;
bool wartet = false;            // erster Klick da, wartet auf den zweiten?
unsigned long erster_klick = 0;

void setup() {
    Serial.begin(115200);
    pinMode(TASTE_A, INPUT_PULLUP);
}

void loop() {
    bool jetzt = digitalRead(TASTE_A) == LOW;
    if (jetzt && !vorher) {
        if (wartet) {
            Serial.println("Doppelklick");
            wartet = false;
        } else {
            wartet = true;
```

```
            erster_klick = millis();
        }
    }
    vorher = jetzt;

    // kein zweiter Klick in der Zeit: es war ein Einzelklick
    if (wartet && millis() - erster_klick > DOPPELKLICK_ZEIT) {
        Serial.println("Einzelklick");
        wartet = false;
    }
    delay(10);
}
```

Serieller Monitor

```
Einzelklick
Doppelklick
```

Nach dem ersten Klick weiß der Sketch noch nicht, was kommt. Er merkt sich in `wartet`, dass ein Klick da war, und die Zeit. Kommt innerhalb von 400 ms ein zweiter Druck, ist es ein Doppelklick. Sind die 400 ms ohne zweiten Druck um, war es ein Einzelklick. Der Sketch kann ihn also **frühestens 400 ms nach dem Druck** melden. Deshalb reagieren Geräte, die Doppelklicks kennen, auf einen einfachen Klick immer etwas verzögert.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 10: LEDs, RGB-LED und Töne

Aufgabe 1 – SOS (`k10_loesung1`)

k10_loesung1.ino

```
// Loesung 10.1: SOS mit der roten LED (kurz = 200 ms, lang = 600 ms)
#define LED_ROT 14
#define KURZ 200
#define LANG 600

void setup() {
    pinMode(LED_ROT, OUTPUT);
}

void loop() {
    for (int i = 0; i < 3; i++) {       // S: dreimal kurz
        digitalWrite(LED_ROT, HIGH);
        delay(KURZ);
        digitalWrite(LED_ROT, LOW);
        delay(KURZ);
    }
    delay(2 * KURZ);                    // Pause zwischen Buchstaben
    for (int i = 0; i < 3; i++) {       // O: dreimal lang
        digitalWrite(LED_ROT, HIGH);
        delay(LANG);
```

```
        digitalWrite(LED_ROT, LOW);
        delay(KURZ);
    }
    delay(2 * KURZ);
    for (int i = 0; i < 3; i++) {       // S: dreimal kurz
        digitalWrite(LED_ROT, HIGH);
        delay(KURZ);
        digitalWrite(LED_ROT, LOW);
        delay(KURZ);
    }
    delay(2000);                        // Pause vor dem naechsten SOS
}
```

Jeder Buchstabe ist eine `for`-Schleife mit drei Durchläufen. Nach jedem Aufleuchten folgt eine kurze Pause, sonst verschmelzen die Zeichen. Die Pausen zwischen den Buchstaben sind länger. `KURZ` und `LANG` stehen oben; ändere sie dort, und das ganze Signal wird schneller oder langsamer.

Aufgabe 2 – Blaulicht (`k10_loesung2`)

k10_loesung2.ino

```
// Loesung 10.2: Blaulicht mit der RGB-LED
#include <Adafruit_NeoPixel.h>

#define RGB_LED 11

Adafruit_NeoPixel rgb(1, RGB_LED, NEO_GRB + NEO_KHZ800);

void setup() {
    rgb.begin();
    rgb.setBrightness(60);
}

void loop() {
    rgb.setPixelColor(0, rgb.Color(0, 0, 255));
    rgb.show();
    delay(150);
    rgb.setPixelColor(0, rgb.Color(0, 0, 0));
    rgb.show();
    delay(100);
    rgb.setPixelColor(0, rgb.Color(0, 0, 255));
    rgb.show();
    delay(150);
    rgb.setPixelColor(0, rgb.Color(0, 0, 0));
    rgb.show();
    delay(600);
}
```

`rgb.Color(0, 0, 0)` ist Schwarz, also aus. Mit `rgb.setBrightness(60)` blitzt es etwas heller als in den Beispielen des Kapitels.

Aufgabe 3 – Dimmer (`k10_loesung3`)

k10_loesung3.ino

```ino
int helligkeit = 0;
bool hoch_vorher = false;
bool runter_vorher = false;

void setup() {
    tft.init(240, 280);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
    tft.fillScreen(ST77XX_BLACK);
    tft.setTextSize(5);
    tft.setTextColor(ST77XX_YELLOW, ST77XX_BLACK);
    pinMode(TASTE_HOCH, INPUT_PULLUP);
    pinMode(TASTE_RUNTER, INPUT_PULLUP);
    analogWrite(LED_GELB, helligkeit);
    tft.setCursor(75, 120);
    tft.print(helligkeit);
}

void loop() {
    bool hoch = digitalRead(TASTE_HOCH) == LOW;
    bool runter = digitalRead(TASTE_RUNTER) == LOW;
    if (hoch && !hoch_vorher && helligkeit < 250) {
        helligkeit = helligkeit + 25;
    }
    if (runter && !runter_vorher && helligkeit > 0) {
        helligkeit = helligkeit - 25;
    }
    if (hoch != hoch_vorher || runter != runter_vorher) {
        analogWrite(LED_GELB, helligkeit);
        tft.setCursor(75, 120);
        tft.print(helligkeit);
        tft.print("  ");
    }
    hoch_vorher = hoch;
    runter_vorher = runter;
    delay(10);
}
```

- Die Bedingungen `helligkeit < 250` und `helligkeit > 0` halten den Wert im erlaubten Bereich. Ohne sie käme man mit runter auf −25; `analogWrite` würde daraus zwar 0 machen, aber auf dem Bildschirm stünde −25.
- `analogWrite` und die Anzeige laufen nur, wenn sich an den Tasten etwas geändert hat, nicht bei jedem Durchlauf.



Aufgabe 4 – Sirene (`k10_loesung4`)

k10_loesung4.ino

```
// Loesung 10.4: eine Sirene - der Ton steigt und faellt
#define LAUTSPRECHER 15

void setup() {
}

void loop() {
    for (int frequenz = 500; frequenz <= 1000; frequenz = frequenz + 10) {
        tone(LAUTSPRECHER, frequenz);
        delay(15);
    }
    for (int frequenz = 1000; frequenz >= 500; frequenz = frequenz - 10) {
        tone(LAUTSPRECHER, frequenz);
        delay(15);
    }
}
```

`tone` ohne Dauer stellt nur die Frequenz um; der Ton klingt einfach mit der neuen Höhe weiter. 51 Schritte zu je 15 ms ergeben etwa 0,8 Sekunden für jede Richtung.

Aufgabe 5 – Ein Klavier (`k10_loesung5`)

k10_loesung5.ino

```
int frequenz_vorher = 0;

void setup() {
    pinMode(TASTE_HOCH, INPUT_PULLUP);
    pinMode(TASTE_RUNTER, INPUT_PULLUP);
    pinMode(TASTE_LINKS, INPUT_PULLUP);
    pinMode(TASTE_RECHTS, INPUT_PULLUP);
    pinMode(TASTE_MITTE, INPUT_PULLUP);
    pinMode(TASTE_A, INPUT_PULLUP);
    pinMode(TASTE_B, INPUT_PULLUP);
    rgb.begin();
    rgb.setBrightness(40);
}

void loop() {
    // welcher Ton gehoert zur gedrueckten Taste? 0 = keiner
    int frequenz = 0;
    if (digitalRead(TASTE_LINKS) == LOW) {
        frequenz = 262;     // c
    } else if (digitalRead(TASTE_RUNTER) == LOW) {
        frequenz = 294;     // d
    } else if (digitalRead(TASTE_RECHTS) == LOW) {
        frequenz = 330;     // e
    } else if (digitalRead(TASTE_HOCH) == LOW) {
        frequenz = 349;     // f
    } else if (digitalRead(TASTE_MITTE) == LOW) {
        frequenz = 392;     // g
```

```
43      } else if (digitalRead(TASTE_B) == LOW) {
44          frequenz = 440;     // a
45      } else if (digitalRead(TASTE_A) == LOW) {
46          frequenz = 494;     // h
47      }
48
49      // nur etwas tun, wenn sich der Ton aendert
50      if (frequenz != frequenz_vorher) {
51          if (frequenz == 0) {
52              noTone(LAUTSPRECHER);
53              rgb.setPixelColor(0, 0);
54          } else {
55              tone(LAUTSPRECHER, frequenz);
56              // hoehere Toene bekommen einen Farbton weiter im Farbkreis
57              rgb.setPixelColor(0, rgb.ColorHSV((frequenz - 262) * 180));
58          }
59          rgb.show();
60          frequenz_vorher = frequenz;
61      }
62      delay(10);
63  }
```

- Die `if`-Kette bestimmt die Frequenz der gedrückten Taste. Wie beim Joystick in Kapitel 9 gewinnt die erste, wenn mehrere gedrückt sind.
- Zeile 50 vergleicht mit dem letzten Durchlauf. Nur bei einer Änderung startet ein neuer Ton oder `noTone`. Das ist die Flankenerkennung aus Kapitel 9, nur für eine Zahl statt für eine einzelne Taste.
- `rgb.setPixelColor(0, 0)` macht die LED aus: Die Zahl 0 ist die Farbe Schwarz.
- Der Farbton `(frequenz - 262) * 180` ist 0 (Rot) für c' und wächst mit der Tonhöhe; für h' ist er 41760, das ist Blau.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 11: Funktionen

Aufgabe 1 – Sterne (`k11_loesung1`)

k11_loesung1.ino

```
1  // Loesung 11.1: ein Dreieck aus Sternen
2  void setup() {
3      Serial.begin(115200);
4      while (!Serial) {
5          delay(10);
6      }
7      for (int i = 1; i <= 5; i++) {
8          sterne(i);
9      }
10 }
```

```
12 void loop() {
13 }
14
15 // gibt anzahl Sterne in einer Zeile aus
16 void sterne(int anzahl) {
17     for (int i = 0; i < anzahl; i++) {
18         Serial.print("*");
19     }
20     Serial.println();
21 }
```

Serieller Monitor

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

Die Schleife in `setup()` ruft `sterne` mit 1, 2, 3, 4 und 5 auf. In `sterne` steht eine zweite Schleife mit eigenem `i`. Die beiden stören sich nicht: Jedes `i` ist lokal in seiner Schleife. `Serial.println()` ohne Argument beendet nur die Zeile.

Aufgabe 2 – Fahrenheit (`k11_loesung2`)

k11_loesung2.ino

```
 1 // Loesung 11.2: Celsius in Fahrenheit umrechnen
 2 void setup() {
 3     Serial.begin(115200);
 4     while (!Serial) {
 5         delay(10);
 6     }
 7     for (int c = -10; c <= 40; c = c + 10) {
 8         Serial.printf("%4d Grad C = %5.1f Grad F\n", c, fahrenheit(c));
 9     }
10 }
11
12 void loop() {
13 }
14
15 float fahrenheit(float celsius) {
16     return celsius * 9 / 5 + 32;
17 }
```

Serieller Monitor

```
-10 Grad C =  14.0 Grad F
  0 Grad C =  32.0 Grad F
 10 Grad C =  50.0 Grad F
 20 Grad C =  68.0 Grad F
 30 Grad C =  86.0 Grad F
 40 Grad C = 104.0 Grad F
```

`c` ist ein `int`; beim Aufruf wird es in den `float`-Parameter `celsius` umgewandelt. Deshalb rechnet `celsius * 9 / 5` mit Kommazahlen. Mit einem `int`-Parameter wäre `-10 * 9 / 5` eine ganzzahlige Division (Kapitel 5). `%4d` und `%5.1f` sorgen für gleich breite Spalten.

Aufgabe 3 – Schaltjahr (`k11_loesung3`)

k11_loesung3.ino

```ino
// Loesung 11.3: Ist ein Jahr ein Schaltjahr?
void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    pruefe(1900);
    pruefe(2000);
    pruefe(2024);
    pruefe(2026);
}

void loop() {
}

bool ist_schaltjahr(int jahr) {
    if (jahr % 400 == 0) {
        return true;
    }
    if (jahr % 100 == 0) {
        return false;
    }
    return jahr % 4 == 0;
}

void pruefe(int jahr) {
    if (ist_schaltjahr(jahr)) {
        Serial.printf("%d ist ein Schaltjahr\n", jahr);
    } else {
        Serial.printf("%d ist kein Schaltjahr\n", jahr);
    }
}
```

Serieller Monitor

```
1900 ist kein Schaltjahr
2000 ist ein Schaltjahr
2024 ist ein Schaltjahr
2026 ist kein Schaltjahr
```

`ist_schaltjahr` prüft die Regeln von der strengsten zur allgemeinsten. Ein `return` beendet die Funktion sofort, deshalb braucht es kein `else`. Die letzte Zeile gibt das Ergebnis des Vergleichs direkt zurück. `pruefe` ist eine zweite Funktion, damit die Ausgabe nicht viermal dasteht.

Aufgabe 4 – Unbekannte Messwerte (`k11_loesung4`)

Ausgabe

```
k11_aufgabe4.ino: In function 'void setup()':
k11_aufgabe4.ino:4:9: warning: unused variable 'messung1' [-Wunused-variable]
    4 |     int messung1 = 17;
      |         ^~~~~~~~
k11_aufgabe4.ino:5:9: warning: unused variable 'messung2' [-Wunused-variable]
    5 |     int messung2 = 24;
      |         ^~~~~~~~
k11_aufgabe4.ino: In function 'float mittel()':
k11_aufgabe4.ino:13:13: error: 'messung1' was not declared in this scope
   13 |     return (messung1 + messung2) / 2.0;
      |             ^~~~~~~~
k11_aufgabe4.ino:13:24: error: 'messung2' was not declared in this scope
   13 |     return (messung1 + messung2) / 2.0;
      |                        ^~~~~~~~
Fehler beim Build: exit status 1
```

`messung1` und `messung2` sind **lokale** Variablen von `setup()`. In `mittel()` sind sie unbekannt (*was not declared in this scope*, „in diesem Bereich nicht angelegt“). Die Lösung: Die Werte kommen als Parameter in die Funktion. Die beiden Warnungen *unused variable* (nicht benutzte Variable) davor deuten auf dasselbe Problem: In `setup()` werden die Variablen angelegt, aber nie gebraucht. Der zweite Fehler: Ohne `while (!Serial)` gibt der Sketch seine einzige Zeile aus, bevor der serielle Monitor zuhört, und sie geht verloren (Kapitel 4).

k11_loesung4.ino

```
// Loesung 11.4: die Messwerte kommen als Parameter in die Funktion
void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    int messung1 = 17;
    int messung2 = 24;
    ausgeben(mittel(messung1, messung2));
}

void loop() {
}

float mittel(int a, int b) {
    return (a + b) / 2.0;
}

void ausgeben(float wert) {
    Serial.printf("Mittelwert: %.1f\n", wert);
}
```

Serieller Monitor

```
Mittelwert: 20.5
```

Aufgabe 5 – Ein Zeichenbaukasten (`k11_loesung5`)

k11_loesung5.ino

```ino
void setup() {
    tft.init(240, 280);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
    tft.fillScreen(tft.color565(120, 190, 255));            // Himmel
    tft.fillRect(0, 210, 240, 70, tft.color565(60, 160, 60)); // Wiese
    tft.fillCircle(190, 50, 22, ST77XX_YELLOW);              // Sonne

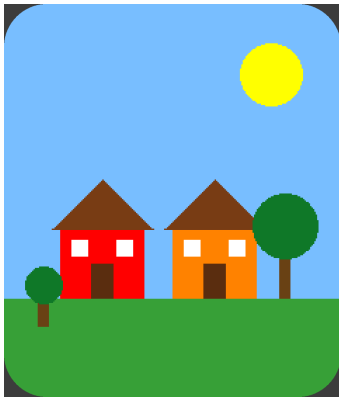
    zeichne_haus(40, 210, ST77XX_RED);
    zeichne_haus(120, 210, ST77XX_ORANGE);
    zeichne_baum(200, 210, 70);
    zeichne_baum(28, 230, 40);
}

void loop() {
}

// Haus mit linker unterer Ecke bei (x, y), 60 Pixel breit
void zeichne_haus(int x, int y, uint16_t farbe) {
    tft.fillRect(x, y - 50, 60, 50, farbe);                       // Wand
    tft.fillTriangle(x - 6, y - 50, x + 66, y - 50, x + 30, y - 85,
                     tft.color565(120, 60, 30));                  // Dach
    tft.fillRect(x + 22, y - 25, 16, 25, tft.color565(90, 50, 20)); // Tuer
    tft.fillRect(x + 8, y - 42, 12, 12, ST77XX_WHITE);            // Fenster
    tft.fillRect(x + 40, y - 42, 12, 12, ST77XX_WHITE);
}

// Baum, der bei (x, y) auf dem Boden steht
void zeichne_baum(int x, int y, int hoehe) {
    tft.fillRect(x - 4, y - hoehe / 2, 8, hoehe / 2, tft.color565(110, 70, 30));
    int krone = y - hoehe / 2 - hoehe / 4;
    tft.fillCircle(x, krone, hoehe / 3, tft.color565(20, 120, 40));
}
```

Bei Haus und Baum ist (x, y) der Punkt, an dem sie **auf dem Boden** stehen: beim Haus die linke untere Ecke, beim Baum die Mitte des Stamms unten. Alles andere wird nach oben gerechnet (`y - 50`). So stellst du jedes Haus mit `y = 210` auf die Wiese, egal wie hoch es ist. Der Baum rechnet Stamm und Krone aus seiner Höhe; ein kleiner Baum ist einfach ein verkleinerter großer.



Lösungen zu Kapitel 12: Arrays und Texte

Aufgabe 1 – Notenschnitt (`k12_loesung1`)

k12_loesung1.ino

```
// Loesung 12.1: Noten auswerten
int noten[] = {2, 3, 1, 4, 2, 2, 3};
const int ANZAHL = sizeof(noten) / sizeof(noten[0]);

void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    int summe = 0;
    int beste = noten[0];
    int schlechteste = noten[0];
    for (int i = 0; i < ANZAHL; i++) {
        summe = summe + noten[i];
        if (noten[i] < beste) {
            beste = noten[i];
        }
        if (noten[i] > schlechteste) {
            schlechteste = noten[i];
        }
    }
    Serial.printf("Durchschnitt: %.2f\n", (float)summe / ANZAHL);
    Serial.printf("Beste Note: %d, schlechteste Note: %d\n", beste,
      schlechteste);
}

void loop() {
}
```

Serieller Monitor

```
Durchschnitt: 2.43
Beste Note: 1, schlechteste Note: 4
```

`beste` und `schlechteste` beginnen beide mit dem ersten Element. Ohne `(float)` vor `summe` käme bei der Division 2 heraus statt 2,43.

Aufgabe 2 – Rückwärts (`k12_loesung2`)

k12_loesung2.ino

```
// Loesung 12.2: einen Text rueckwaerts ausgeben
#include <string.h>

char text[] = "Arduino macht Spass";
```

```
 6 void setup() {
 7     Serial.begin(115200);
 8     while (!Serial) {
 9         delay(10);
10     }
11     for (int i = strlen(text) - 1; i >= 0; i--) {
12         Serial.print(text[i]);
13     }
14     Serial.println();
15 }
16
17 void loop() {
18 }
```

Serieller Monitor

```
ssapS thcam oniudrA
```

Das letzte Zeichen hat den Index `strlen(text) - 1`. Die Schleife zählt mit `i--` abwärts bis einschließlich 0. `i` muss dafür ein `int` sein: Eine Zahl ohne Vorzeichen kann nie kleiner als 0 werden, und die Schleife liefe ewig.

Aufgabe 3 – Melodie mit Lauflicht (`k12_loesung3`)

k12_loesung3.ino

```
 1 // Loesung 12.3: die Melodie mit einer Pause und einem Lauflicht
 2 #define LAUTSPRECHER 15
 3
 4 const int LEDS[] = {14, 13, 12};
 5 int toene[] = {392, 330, 0, 392, 330, 0, 392, 440, 392, 349, 330, 294, 262};
 6 int dauern[] = {2, 2, 1, 2, 2, 1, 1, 1, 1, 1, 1, 1, 4};
 7 const int ANZAHL = sizeof(toene) / sizeof(toene[0]);
 8
 9 void setup() {
10     for (int i = 0; i < 3; i++) {
11         pinMode(LEDS[i], OUTPUT);
12     }
13     for (int i = 0; i < ANZAHL; i++) {
14         int ms = dauern[i] * 200;
15         int led = LEDS[i % 3];          // 0, 1, 2, 0, 1, 2, ...
16         if (toene[i] > 0) {
17             tone(LAUTSPRECHER, toene[i], ms - 30);
18             digitalWrite(led, HIGH);
19         }
20         delay(ms);
21         digitalWrite(led, LOW);
22     }
23 }
24
25 void loop() {
26 }
```

- Die Töne mit dem Wert 0 sind Pausen; dann bleibt auch die LED aus.
- `i % 3` liefert der Reihe nach 0, 1, 2, 0, 1, 2 … und wählt so die LED aus dem Array `LEDS`.

- Die LED wird nach dem `delay` ausgeschaltet, also am Ende jedes Tons.

Aufgabe 4 – Vokale zählen (`k12_loesung4`)

k12_loesung4.ino

```
10 void loop() {
11     if (Serial.available() > 0) {
12         String satz = Serial.readStringUntil('\n');
13         satz.trim();
14         String klein = satz;
15         klein.toLowerCase();
16         int vokale = 0;
17         for (unsigned int i = 0; i < klein.length(); i++) {
18             char c = klein[i];
19             if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') {
20                 vokale = vokale + 1;
21             }
22         }
23         Serial.printf("\"%s\" hat %d Vokale.\n", satz.c_str(), vokale);
24     }
25 }
```

Serieller Monitor

```
Gib einen Satz ein:
"Der PicoBoy kann Arduino" hat 9 Vokale.
```

Statt fünf Vergleiche für Groß- und noch einmal fünf für Kleinbuchstaben zu schreiben, macht der Sketch eine kleine Kopie (`klein`) und wandelt sie mit `toLowerCase()` um. Der ursprüngliche Satz bleibt für die Ausgabe erhalten. Umlaute zählt der Sketch nicht mit; sie bestehen aus zwei Bytes (12.9).

Aufgabe 5 – Morsen (`k12_loesung5`)

k12_loesung5.ino

```
 1 // Loesung 12.5: Text aus dem seriellen Monitor als Morsezeichen
 2 #define LAUTSPRECHER 15
 3 #define LED_ROT 14
 4 #define PUNKT 120                  // ms; ein Strich ist dreimal so lang
 5
 6 String morse[26] = {
 7     ".-", "-...", "-.-.", "-..", ".", "..-.", "--.", "....", "..",      // A bis
       I
 8     ".---", "-.-", ".-..", "--", "-.", "---", ".--.", "--.-", ".-.",    // J bis
       R
 9     "...", "-", "..-", "...-", ".--", "-..-", "-.--", "--.."            // S bis
       Z
10 };
11
12 void setup() {
13     Serial.begin(115200);
```

```
    pinMode(LED_ROT, OUTPUT);
}

void loop() {
    if (Serial.available() > 0) {
        String text = Serial.readStringUntil('\n');
        text.trim();
        text.toUpperCase();
        for (unsigned int i = 0; i < text.length(); i++) {
            sende_zeichen(text[i]);
        }
        Serial.println();
    }
}

void sende_zeichen(char c) {
    if (c >= 'A' && c <= 'Z') {
        String code = morse[c - 'A'];   // 'A' - 'A' = 0, 'B' - 'A' = 1, ...
        Serial.print(code + " ");
        for (unsigned int k = 0; k < code.length(); k++) {
            int dauer = PUNKT;
            if (code[k] == '-') {
                dauer = 3 * PUNKT;
            }
            digitalWrite(LED_ROT, HIGH);
            tone(LAUTSPRECHER, 700, dauer);
            delay(dauer);
            digitalWrite(LED_ROT, LOW);
            delay(PUNKT);               // Pause im Zeichen
        }
        delay(2 * PUNKT);               // Pause zwischen Buchstaben
    } else if (c == ' ') {
        Serial.print("/ ");
        delay(4 * PUNKT);               // Pause zwischen Woertern
    }
}
```

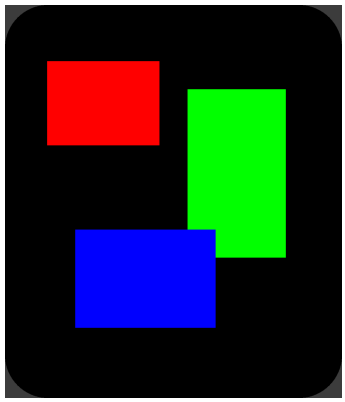
Serieller Monitor

```
... --- ... / .... .. .--- ..-. .
```

- Das Array `morse` hat 26 Einträge in der Reihenfolge des Alphabets. `c - 'A'` rechnet mit den Zeichencodes: `'A'` ist 65, `'C'` ist 67, also ist `'C' - 'A'` gleich 2, der dritte Eintrag.
- `toUpperCase()` macht aus der Eingabe Großbuchstaben, so passen auch kleine Buchstaben. Andere Zeichen außer dem Leerzeichen werden übersprungen.
- Die Pausen addieren sich: Nach jedem Zeichen eine Punktlänge, nach dem letzten Zeichen eines Buchstabens zwei weitere (zusammen drei), beim Leerzeichen noch einmal vier (zusammen sieben).

Lösungen zu Kapitel 13: Strukturen und eine erste eigene Klasse

Aufgabe 1 – Rechtecke (`k13_loesung1`)



k13_loesung1.ino

```
 9 struct Rechteck {
10     int x, y;               // linke obere Ecke
11     int breite, hoehe;
12     uint16_t farbe;
13 };
14
15 Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
16
17 Rechteck rechtecke[] = {
18     {30, 40, 80, 60, ST77XX_RED},
19     {130, 60, 70, 120, ST77XX_GREEN},
20     {50, 160, 100, 70, ST77XX_BLUE},
21 };
22
23 void setup() {
24     Serial.begin(115200);
25     tft.init(240, 280);
26     tft.fillScreen(ST77XX_BLACK);
27     pinMode(TFT_LICHT, OUTPUT);
28     digitalWrite(TFT_LICHT, HIGH);
29     for (int i = 0; i < 3; i++) {
30         zeichne(rechtecke[i]);
31         Serial.printf("Rechteck %d: Flaeche %d\n", i, flaeche(rechtecke[i]));
32     }
33 }
34
35 void loop() {
36 }
37
38 int flaeche(Rechteck r) {
39     return r.breite * r.hoehe;
40 }
```

```
42 void zeichne(Rechteck r) {
43     tft.fillRect(r.x, r.y, r.breite, r.hoehe, r.farbe);
44 }
```

Serieller Monitor

```
Rechteck 0: Flaeche 4800
Rechteck 1: Flaeche 8400
Rechteck 2: Flaeche 7000
```

Beide Funktionen bekommen ein ganzes `Rechteck` und holen sich daraus, was sie brauchen. `zeichne` muss nichts zurückgeben, `flaeche` liefert eine Zahl.

Aufgabe 2 – Ampelphasen (`k13_loesung2`)

k13_loesung2.ino

```
 1 // Loesung 13.2: die Phasen einer Ampel als enum
 2 enum Phase {
 3     ROT,
 4     ROT_GELB,
 5     GRUEN,
 6     GELB
 7 };
 8
 9 void setup() {
10     Serial.begin(115200);
11     while (!Serial) {
12         delay(10);
13     }
14     Phase p = ROT;
15     for (int i = 0; i < 6; i++) {
16         Serial.println(name(p));
17         p = naechste(p);
18     }
19 }
20
21 void loop() {
22 }
23
24 Phase naechste(Phase p) {
25     switch (p) {
26         case ROT:
27             return ROT_GELB;
28         case ROT_GELB:
29             return GRUEN;
30         case GRUEN:
31             return GELB;
32         case GELB:
33             return ROT;
34     }
35     return ROT;
36 }
```

```
String name(Phase p) {
    switch (p) {
        case ROT:
            return "rot";
        case ROT_GELB:
            return "rot und gelb";
        case GRUEN:
            return "gruen";
        case GELB:
            return "gelb";
    }
    return "?";
}
```

Serieller Monitor

```
rot
rot und gelb
gruen
gelb
rot
rot und gelb
```

In einem `case` beendet `return` die ganze Funktion, ein `break` ist dann nicht nötig. Das `return` hinter dem `switch` wird nie erreicht, solange `p` einer der vier Werte ist. Ohne diese Zeile meldet der Compiler aber *control reaches end of non-void function* (Kapitel 11.3), denn er kann nicht sicher sein, dass in `p` nie etwas anderes steht.

Aufgabe 3 – Lang oder kurz? (`k13_loesung3`)

k13_loesung3.ino

```
// Loesung 13.3: die Klasse Taste erkennt auch das Loslassen und die Druckdauer
#define TASTE_A 27

class Taste {
public:
    Taste(int anschluss) {
        pin = anschluss;
    }
    void begin() {
        pinMode(pin, INPUT_PULLUP);
    }
    bool gedrueckt() {
        return digitalRead(pin) == LOW;
    }
    // prueft die Taste; einmal pro Durchlauf von loop() aufrufen
    void pruefen() {
        neu = false;
        los = false;
        bool jetzt = gedrueckt();
        if (jetzt != vorher && millis() - letzte_aenderung >= 50) {
            letzte_aenderung = millis();
            vorher = jetzt;
```

```
            if (jetzt) {
                neu = true;
                beginn = millis();
            } else {
                los = true;
            }
        }
    }
    bool neu_gedrueckt() {
        return neu;
    }
    bool losgelassen() {
        return los;
    }
    // wie lange die Taste schon gedrueckt ist (oder zuletzt war), in ms
    unsigned long dauer() {
        if (vorher) {
            return millis() - beginn;
        }
        return letzte_aenderung - beginn;
    }
private:
    int pin;
    bool vorher = false;
    bool neu = false;
    bool los = false;
    unsigned long letzte_aenderung = 0;
    unsigned long beginn = 0;
};

Taste a(TASTE_A);

void setup() {
    Serial.begin(115200);
    a.begin();
}

void loop() {
    a.pruefen();
    if (a.neu_gedrueckt()) {
        Serial.println("A gedrueckt");
    }
    if (a.losgelassen()) {
        if (a.dauer() >= 1000) {
            Serial.printf("A lang gedrueckt (%lu ms)\n", a.dauer());
        } else {
            Serial.printf("A kurz gedrueckt (%lu ms)\n", a.dauer());
        }
    }
}
```

Serieller Monitor

```
A gedrueckt
A kurz gedrueckt (200 ms)
A gedrueckt
A lang gedrueckt (1500 ms)
```

- `pruefen()` liest die Taste einmal pro Durchlauf und merkt sich in `neu` und `los`, ob gerade eine Flanke war. `neu_gedrueckt()` und `losgelassen()` geben das nur zurück. So kann man sie im selben Durchlauf beliebig oft fragen.
- `beginn` speichert die Zeit des Drucks. `dauer()` rechnet bei gedrückter Taste bis jetzt, nach dem Loslassen bis zum Loslassen.
- Die Klasse ist gewachsen, aber `loop()` bleibt kurz und lesbar. Das ist der Sinn einer Klasse.

Aufgabe 4 – Eine Klasse für LEDs (`k13_loesung4`)

k13_loesung4.ino

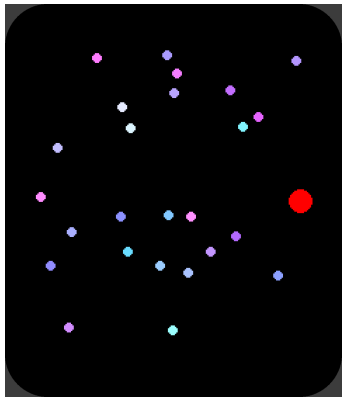
```
// Loesung 13.4: eine Klasse fuer LEDs und ein Lauflicht mit drei Objekten
class Led {
public:
    Led(int anschluss) {
        pin = anschluss;
    }
    void begin() {
        pinMode(pin, OUTPUT);
        aus();
    }
    void an() {
        digitalWrite(pin, HIGH);
        leuchtet = true;
    }
    void aus() {
        digitalWrite(pin, LOW);
        leuchtet = false;
    }
    void umschalten() {
        if (leuchtet) {
            aus();
        } else {
            an();
        }
    }
private:
    int pin;
    bool leuchtet = false;
};

Led leds[3] = {Led(14), Led(13), Led(12)};     // rot, gelb, gruen
```

```
33 void setup() {
34     for (int i = 0; i < 3; i++) {
35         leds[i].begin();
36     }
37 }
38
39 void loop() {
40     for (int i = 0; i < 3; i++) {
41         leds[i].umschalten();
42         delay(300);
43     }
44 }
```

`leuchtet` merkt sich, ob die LED an ist, damit `umschalten()` weiß, was zu tun ist. Das Array wird mit drei Konstruktor-Aufrufen gefüllt. Das Lauflicht schaltet jede LED abwechselnd an und beim nächsten Durchgang wieder aus.

Aufgabe 5 – Der Hai (`k13_loesung5`)



k13_loesung5.ino

```
69 void bewege(int i) {
70     float abstand_x = schwarm[i].x - hai_x;
71     float abstand_y = schwarm[i].y - hai_y;
72     // Abstand im Quadrat: spart die Wurzel
73     if (abstand_x * abstand_x + abstand_y * abstand_y < FLUCHT * FLUCHT) {
74         schwarm[i].dx = schwarm[i].dx + abstand_x * 0.05;    // weg vom Hai
75         schwarm[i].dy = schwarm[i].dy + abstand_y * 0.05;
76     }
77     float zufall_x = random(-10, 11) / 100.0;
78     float zufall_y = random(-10, 11) / 100.0;
79     schwarm[i].dx = constrain(schwarm[i].dx + zufall_x, -TEMPO, TEMPO);
80     schwarm[i].dy = constrain(schwarm[i].dy + zufall_y, -TEMPO, TEMPO);
81     schwarm[i].x = schwarm[i].x + schwarm[i].dx;
82     schwarm[i].y = schwarm[i].y + schwarm[i].dy;
83     if (schwarm[i].x < 25 || schwarm[i].x > 215) {
84         schwarm[i].dx = -schwarm[i].dx;
85         schwarm[i].x = constrain(schwarm[i].x, 25, 215);
86     }
87     if (schwarm[i].y < 25 || schwarm[i].y > 255) {
88         schwarm[i].dy = -schwarm[i].dy;
```

```
89          schwarm[i].y = constrain(schwarm[i].y, 25, 255);
90      }
91  }
```

- `abstand_x` und `abstand_y` zeigen vom Hai zum Fisch. Werden sie zur Richtung addiert, schwimmt der Fisch vom Hai weg, und zwar umso kräftiger, je näher er ist.
- `FLUCHT * FLUCHT` vergleicht mit dem Quadrat des Abstands; das spart die Wurzel (`sqrt`), die viel länger dauert.
- An den Rändern dreht sich die Richtung um wie beim Ball in 13.1.

Den ganzen Sketch findest du als `k13_loesung5`.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 14: Flimmerfrei

Aufgabe 1 – Teilerstufen

Für jeden Wunsch nimmt der SPI-Baustein die nächstkleinere Stufe 75 MHz / k:

Wunsch	echter Takt	ein ganzes Bild
20 MHz	18,75 MHz (75 / 4)	1 075 200 Bit / 18 750 000 Bit/s ≈ 57 ms
32 MHz	25 MHz (75 / 3)	≈ 43 ms
40 MHz	37,5 MHz (75 / 2)	≈ 29 ms
62,5 MHz	37,5 MHz (75 / 2)	≈ 29 ms

Serieller Monitor

```
Wunsch 20000000 Hz: 57348 us
Wunsch 32000000 Hz: 43012 us
Wunsch 40000000 Hz: 28676 us
Wunsch 62500000 Hz: 28676 us
```

40 MHz und 62,5 MHz ergeben dieselbe Zeit: Beide liegen zwischen 37,5 und 75 MHz, also ist 37,5 MHz die nächstkleinere Stufe. Mehr als 37,5 MHz gibt es erst mit 75 MHz, und das ist für den Displaychip zu schnell.

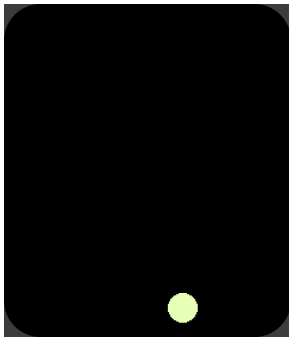
Aufgabe 2 – Bunte Abpraller (`k14_loesung2`)

k14_loesung2.ino

```
12 GFXcanvas16 bild(240, 280);
13
14 float x = 60;
15 float y = 50;
16 float dx = 3.0;
17 float dy = 2.5;
```

```
18 uint16_t farbe = ST77XX_ORANGE;
19 unsigned long letztes_bild = 0;
20
21 void neue_farbe() {
22     farbe = tft.color565(random(80, 256), random(80, 256), random(80, 256));
23 }
```

`neue_farbe()` mischt eine zufällige Farbe aus drei Anteilen von 80 bis 255; so wird sie nie zu dunkel. In `loop()` steht der Aufruf in jedem der vier `if`-Blöcke, zum Beispiel:



k14_loesung2.ino

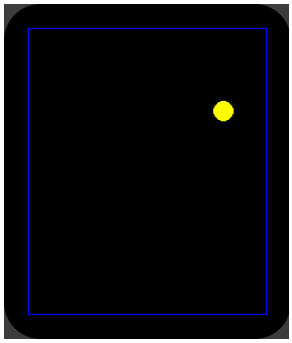
```
37         if (x < RADIUS) {
38             x = RADIUS;
39             dx = -dx;
40             neue_farbe();
41         }
```

Aufgabe 3 – Der Punkt ohne Flackern (`k14_loesung3`)

k14_loesung3.ino

```
32 void loop() {
33     if (digitalRead(TASTE_LINKS) == LOW && x > 30) {
34         x = x - TEMPO;
35     }
36     if (digitalRead(TASTE_RECHTS) == LOW && x < 210) {
37         x = x + TEMPO;
38     }
39     if (digitalRead(TASTE_HOCH) == LOW && y > 30) {
40         y = y - TEMPO;
41     }
42     if (digitalRead(TASTE_RUNTER) == LOW && y < 250) {
43         y = y + TEMPO;
44     }
45
46     // jedes Mal das ganze Bild neu: Loeschen ist nicht noetig
47     bild.fillScreen(ST77XX_BLACK);
48     bild.drawRect(20, 20, 200, 240, ST77XX_BLUE);
49     bild.fillCircle(x, y, RADIUS, ST77XX_YELLOW);
50     tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
51 }
```

Weglassen kannst du das Übermalen des alten Punkts mit Schwarz: Jedes Bild wird vollständig neu gezeichnet. Der Rahmen stört deshalb auch nicht; im alten Sketch hätte der schwarze Kreis ihn beim Darüberfahren gelöscht.



Aufgabe 4 – Springender Ball (`k14_loesung4`)

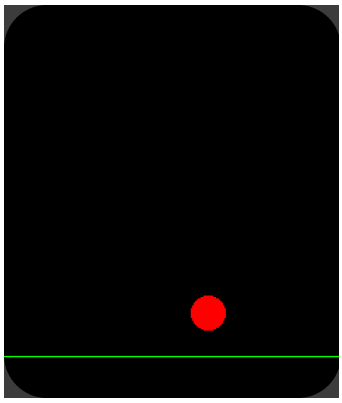
k14_loesung4.ino

```
#define RADIUS 12
#define TAKT 33
#define BODEN 250                  // y der Bodenlinie
#define SCHWERKRAFT 0.5            // so viel schneller pro Bild
#define VERLUST 0.8                // 80 % der Geschwindigkeit bleiben
```

k14_loesung4.ino

```
void loop() {
    if (millis() - letztes_bild >= TAKT) {
        letztes_bild = millis();
        dy = dy + SCHWERKRAFT;
        x = x + dx;
        y = y + dy;
        if (y > BODEN - RADIUS) {
            y = BODEN - RADIUS;
            dy = -dy * VERLUST;
        }
        if (x < RADIUS || x > 239 - RADIUS) {
            x = constrain(x, RADIUS, 239 - RADIUS);
            dx = -dx;
        }
        bild.fillScreen(ST77XX_BLACK);
        bild.drawFastHLine(0, BODEN, 240, ST77XX_GREEN);
        bild.fillCircle(x, y, RADIUS, ST77XX_RED);
        tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
    }
}
```

Die Schwerkraft macht `dy` in jedem Bild etwas größer: Der Ball wird beim Fallen schneller und beim Steigen langsamer, bis er umkehrt. Am Boden kehrt `dy = -dy * VERLUST` die Richtung um und nimmt 20 % des Tempos weg. Nach einiger Zeit hüpft der Ball nur noch ganz wenig und rollt am Boden entlang.



Aufgabe 5 – Laufschrift im Streifen (`k14_loesung5`)

k14_loesung5.ino

```
// Laufschrift in einem kleinen Canvas: nur ein Streifen wird uebertragen
#include <Adafruit_ST7789.h>

#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
```

```
#define TFT_LICHT 26

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
GFXcanvas16 streifen(240, 40);      // nur 240 x 40 Pixel

int x = 240;
unsigned long letzte_ausgabe = 0;

void setup() {
    Serial.begin(115200);
    tft.init(240, 280);
    tft.setSPISpeed(62500000);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
    // der Rest des Bildschirms wird nur einmal gezeichnet
    tft.fillScreen(ST77XX_BLUE);
    tft.setTextColor(ST77XX_WHITE);
    tft.setTextSize(2);
    tft.setCursor(48, 60);
    tft.print("Nachrichten");
}

void loop() {
    streifen.fillScreen(ST77XX_BLACK);
    streifen.setTextColor(ST77XX_YELLOW);
    streifen.setTextSize(3);
    streifen.setTextWrap(false);    // nicht in die naechste Zeile umbrechen
    streifen.setCursor(x, 10);
    streifen.print("+++ PicoBoy flimmert nicht +++");

    unsigned long start = micros();
    tft.drawRGBBitmap(0, 120, streifen.getBuffer(), 240, 40);
    unsigned long dauer = micros() - start;

    x = x - 3;
    if (x < -540) {                 // Text ganz links hinaus: von rechts neu
        x = 240;
    }
    if (millis() - letzte_ausgabe >= 1000) {
        Serial.printf("Streifen: %lu us\n", dauer);
        letzte_ausgabe = millis();
    }
    delay(20);
}
```

Serieller Monitor

```
Streifen: 4100 us
Streifen: 4100 us
```

Nachgerechnet: 240 · 40 Pixel · 16 Bit = 153 600 Bit. Bei 37,5 MHz sind das 153 600 / 37 500 000 s ≈ 4,1 ms, ein Siebtel eines ganzen Bildes. Ein kleiner Canvas spart also Zeit **und** Speicher (19 200 statt 134 400 Byte).

Der Text ist mit Größe 3 genau 30 Zeichen · 18 Pixel = 540 Pixel lang. Wenn `x` kleiner als −540 ist, ist er ganz links hinausgelaufen und beginnt wieder rechts. `delay(20)` bremst die Schrift; ohne würde sie bei über 200 Bildern pro Sekunde unlesbar schnell laufen.



ANHANG C – LÖSUNGEN

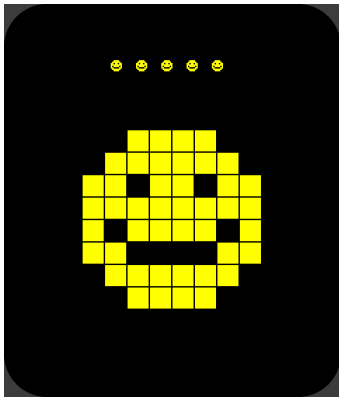
Lösungen zu Kapitel 15: Bilder und Sprites

Aufgabe 1 – Ein Smiley von Hand (`k15_loesung1`)

k15_loesung1.ino

```
12 const uint16_t S = ST77XX_BLACK;
13 const uint16_t G = ST77XX_YELLOW;
14
15 const uint16_t smiley[64] = {
16     S, S, G, G, G, G, S, S,
17     S, G, G, G, G, G, G, S,
18     G, G, S, G, G, S, G, G,
19     G, G, G, G, G, G, G, G,
20     G, S, G, G, G, G, S, G,
21     G, G, S, S, S, S, G, G,
22     S, G, G, G, G, G, G, S,
23     S, S, G, G, G, G, S, S,
24 };
25
26 void setup() {
27     tft.init(240, 280);
28     pinMode(TFT_LICHT, OUTPUT);
29     digitalWrite(TFT_LICHT, HIGH);
30
31     bild.fillScreen(ST77XX_BLACK);
32     for (int i = 0; i < 5; i++) {
33         bild.drawRGBBitmap(76 + i * 18, 40, smiley, 8, 8);
34     }
35     for (int zeile = 0; zeile < 8; zeile++) {
36         for (int spalte = 0; spalte < 8; spalte++) {
37             bild.fillRect(56 + spalte * 16, 90 + zeile * 16, 15, 15, smiley[zeile
                * 8 + spalte]);
38         }
39     }
40     tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
41 }
```

Die Augen sind die schwarzen Pixel in Zeile 2, der Mund die schwarzen Pixel in Zeile 4 und 5. Die vier Ecken sind ebenfalls schwarz, damit das Gesicht rund wirkt. Auf schwarzem Hintergrund fällt nicht auf, dass sie mit gezeichnet werden.



Aufgabe 2 – Bytes zählen

a) `rakete_pixel` hat 16 · 24 = 384 Farben zu je 2 Byte, also **768 Byte**. Die Maske braucht pro Zeile 16 / 8 = 2 Byte, bei 24 Zeilen **48 Byte**.

b) Der Fisch hat ebenfalls 24 · 16 = 384 Pixel, also **768 Byte**. Seine Zeilen sind aber 24 Pixel breit, das sind 3 Byte pro Zeile; bei 16 Zeilen wieder **48 Byte** Maske.

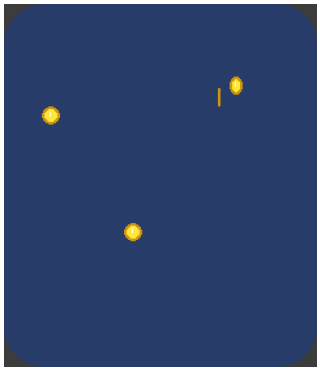
c) Zwei Byte pro Zeile heißt: Das Bild ist 9 bis 16 Pixel breit (bei 8 Pixeln oder weniger genügte ein Byte). `0x18` ist binär `00011000`: Gezeichnet werden nur die Pixel 3 und 4 der Zeile (gezählt ab 0). Das zweite Byte `0x00` sagt: Alle weiteren Pixel sind durchsichtig.

d) 240 · 280 · 2 = **134 400 Byte**, genauso viel wie ein Canvas in Displaygröße. Im Flash-Speicher (16 MB) haben trotzdem viele solche Bilder Platz.

Aufgabe 3 – Münzregen (`k15_loesung3`)

k15_loesung3.ino

```
12 struct Muenze {
13     float x, y;
14     float tempo;
15     int versatz;                  // damit sich nicht alle im Gleichschritt
         drehen
16 };
17
18 Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
19 GFXcanvas16 bild(240, 280);
20
21 Muenze muenzen[ANZAHL];
22 unsigned long letztes_bild = 0;
```



k15_loesung3.ino

```
41 void nach_oben(int i) {
42     muenzen[i].x = random(20, 204);
43     muenzen[i].y = random(-120, -16);
44     muenzen[i].tempo = random(15, 40) / 10.0;
45     muenzen[i].versatz = random(4);
46 }
47
48 void setup() {
49     tft.init(240, 280);
50     tft.setSPISpeed(62500000);
51     pinMode(TFT_LICHT, OUTPUT);
52     digitalWrite(TFT_LICHT, HIGH);
53     for (int i = 0; i < ANZAHL; i++) {
54         nach_oben(i);
55         muenzen[i].y = random(0, 260);       // beim Start schon verteilt
56     }
57 }
58
59 void loop() {
60     if (millis() - letztes_bild >= TAKT) {
61         letztes_bild = millis();
62         int nummer = millis() / 120;
63         bild.fillScreen(tft.color565(40, 60, 110));
64         for (int i = 0; i < ANZAHL; i++) {
65             muenzen[i].y = muenzen[i].y + muenzen[i].tempo;
66             if (muenzen[i].y > 280) {
67                 nach_oben(i);
68             }
69             zeichne_muenze(muenzen[i].x, muenzen[i].y, (nummer +
                 muenzen[i].versatz) % 4);
70         }
71         tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
72     }
73 }
```

- `nach_oben(i)` setzt Münze `i` an eine zufällige Stelle **über** dem Bildschirm (y zwischen −120 und −17) und gibt ihr ein neues Tempo und einen neuen Versatz.
- Beim Start (Zeile 55) sollen die Münzen schon über den Bildschirm verteilt sein, sonst wäre er zuerst eine Weile leer.

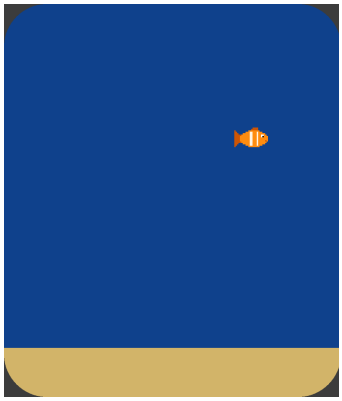
- Zeile 62 zählt alle 120 ms um eins weiter. Zeile 69 addiert den Versatz der Münze und nimmt erst danach `% 4`. So zeigt jede Münze ein anderes Bild, alle drehen sich aber gleich schnell.

Aufgabe 4 – Fisch am Joystick (`k15_loesung4`)

k15_loesung4.ino

```
18 int x = 108;
19 int y = 130;
20 bool nach_rechts = true;            // Blickrichtung
21
22 void setup() {
23     tft.init(240, 280);
24     tft.setSPISpeed(62500000);
25     pinMode(TFT_LICHT, OUTPUT);
26     digitalWrite(TFT_LICHT, HIGH);
27     pinMode(TASTE_HOCH, INPUT_PULLUP);
28     pinMode(TASTE_RUNTER, INPUT_PULLUP);
29     pinMode(TASTE_LINKS, INPUT_PULLUP);
30     pinMode(TASTE_RECHTS, INPUT_PULLUP);
31 }
32
33 void loop() {
34     if (digitalRead(TASTE_LINKS) == LOW) {
35         x = x - TEMPO;
36         nach_rechts = false;
37     }
38     if (digitalRead(TASTE_RECHTS) == LOW) {
39         x = x + TEMPO;
40         nach_rechts = true;
41     }
42     if (digitalRead(TASTE_HOCH) == LOW) {
43         y = y - TEMPO;
44     }
45     if (digitalRead(TASTE_RUNTER) == LOW) {
46         y = y + TEMPO;
47     }
48     x = constrain(x, 10, 230 - FISCH_RECHTS_BREITE);
49     y = constrain(y, 20, 240 - FISCH_RECHTS_HOEHE);
50
51     bild.fillScreen(tft.color565(20, 70, 140));
52     bild.fillRect(0, 245, 240, 35, tft.color565(210, 180, 110));
53     if (nach_rechts) {
54         bild.drawRGBBitmap(x, y, fisch_rechts_pixel, fisch_rechts_maske,
55                            FISCH_RECHTS_BREITE, FISCH_RECHTS_HOEHE);
56     } else {
57         bild.drawRGBBitmap(x, y, fisch_links_pixel, fisch_links_maske,
58                            FISCH_LINKS_BREITE, FISCH_LINKS_HOEHE);
59     }
60     tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
61 }
```

Die Blickrichtung steht in der Variablen `nach_rechts`. Sie ändert sich nur bei links und rechts (Zeile 36 und 40). Hoch, runter oder gar nichts drücken lässt sie, wie sie ist. Das Bild rechts entstand, nachdem der Fisch nach rechts und dann nach oben geschwommen ist: Er schaut weiter nach rechts.



Aufgabe 5 – Deine eigene Figur (`k15_loesung5`)

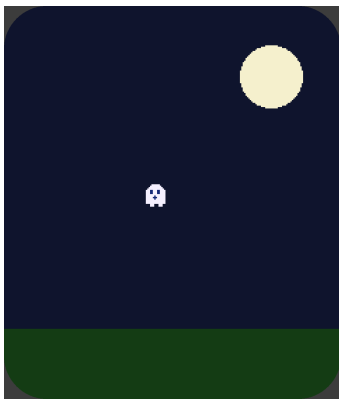
Ein Beispiel: ein kleiner Geist aus zwei Bildern, `geist_0.png` und `geist_1.png`, je 16 · 16 Pixel. Im zweiten Bild sind die Zacken am Saum verschoben; im Wechsel sieht es aus, als flattere er. Beide Bilder sind mit `bild2c` umgewandelt und nacheinander in `geist.h` gespeichert. Achte beim Umwandeln auf verschiedene Namen (`geist_0`, `geist_1`), sonst gibt es zwei Arrays mit demselben Namen, und der Compiler meldet einen Fehler.

k15_loesung5.ino

```
14 float x = 20;
15 float dx = 1.2;
16 unsigned long letztes_bild = 0;
17
18 void setup() {
19     tft.init(240, 280);
20     tft.setSPISpeed(62500000);
21     pinMode(TFT_LICHT, OUTPUT);
22     digitalWrite(TFT_LICHT, HIGH);
23 }
24
25 void loop() {
26     if (millis() - letztes_bild >= TAKT) {
27         letztes_bild = millis();
28         x = x + dx;
29         if (x < 20 || x > 220 - GEIST_0_BREITE) {
30             dx = -dx;
31         }
32         int y = 130 + 25 * sin(millis() / 400.0);
33
34         bild.fillScreen(tft.color565(20, 20, 50));
35         bild.fillCircle(190, 50, 22, tft.color565(240, 240, 200));     // Mond
36         bild.fillRect(0, 230, 240, 50, tft.color565(30, 60, 30));
37         if ((millis() / 250) % 2 == 0) {
38             bild.drawRGBBitmap(x, y, geist_0_pixel, geist_0_maske,
                GEIST_0_BREITE, GEIST_0_HOEHE);
39         } else {
40             bild.drawRGBBitmap(x, y, geist_1_pixel, geist_1_maske,
                GEIST_1_BREITE, GEIST_1_HOEHE);
41         }
```

```
42          tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
43      }
44  }
```

- Zeile 32: Mit `sin()` schwebt der Geist auf und ab, wie der Fisch im Aquarium.
- Zeile 37: `(millis() / 250) % 2` ist abwechselnd 0 und 1, jeweils für 250 ms. Danach richtet sich, welches der beiden Bilder gezeichnet wird.



ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 16: Kollisionen

Aufgabe 1 – Treffer vorhersagen

Serieller Monitor

```
a und b: ja
a und c: nein
a und d: ja
b und c: nein
a und e: nein
d und e: ja
```

- `a` reicht von (10, 10) bis (29, 29), `b` von (25, 25) bis (34, 34): Die Ecke (25…29, 25…29) gehört zu beiden.
- `c` beginnt bei x = 30, genau rechts neben `a` (10 + 20 = 30). Kein Pixel gemeinsam.
- `d` ist ein großes Rechteck, das `a` ganz enthält. Auch das ist eine Überlappung.
- `b` und `c` überlappen sich waagrecht (x 30 bis 34), aber nicht senkrecht: `c` endet bei y = 19, `b` beginnt bei y = 25.
- `e` liegt unter `a` (y ab 40, `a` endet bei 29), aber in `d`.

Aufgabe 2 – Ein Knopf zum Klicken (`k16_loesung2`)

k16_loesung2.ino

```
58      px = constrain(px, 20, 219);
59      py = constrain(py, 20, 259);
60
61      bool drauf = punkt_in(px, py, knopf);
62      bool a = digitalRead(TASTE_A) == LOW;
```

```
63      if (a && !a_vorher && drauf) {
64          zaehler = zaehler + 1;
65      }
66      a_vorher = a;
67
68      bild.fillScreen(ST77XX_BLACK);
69      if (drauf) {
70          bild.fillRoundRect(knopf.x, knopf.y, knopf.b, knopf.h, 8, ST77XX_GREEN);
71      } else {
72          bild.fillRoundRect(knopf.x, knopf.y, knopf.b, knopf.h, 8, ST77XX_BLUE);
73      }
74      bild.setTextColor(ST77XX_WHITE);
75      bild.setTextSize(2);
76      bild.setCursor(knopf.x + 30, knopf.y + 13);
77      bild.print("Klick");
78      bild.setCursor(60, 200);
79      bild.printf("Klicks: %d", zaehler);
80      bild.drawFastHLine(px - 8, py, 17, ST77XX_WHITE);
81      bild.drawFastVLine(px, py - 8, 17, ST77XX_WHITE);
82      tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
83  }
```

- Zeile 61 merkt sich in `drauf`, ob die Mitte des Fadenkreuzes auf dem Knopf liegt. Die Variable wird zweimal gebraucht: für die Farbe und für den Klick.
- Zeile 63: Ein Klick zählt nur, wenn A gerade heruntergedrückt wurde (Flanke) **und** das Fadenkreuz auf dem Knopf liegt.
- Das Fadenkreuz wird zuletzt gezeichnet, damit es über dem Knopf liegt.



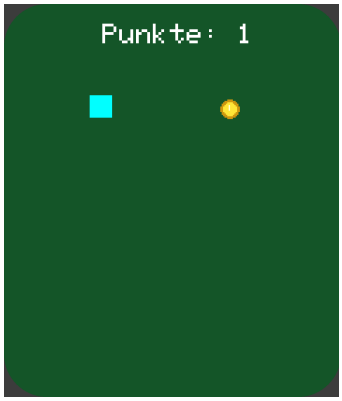
Aufgabe 3 – Münzen sammeln (`k16_loesung3`)

k16_loesung3.ino

```
57      spieler.x = constrain(spieler.x, 20, 220 - spieler.b);
58      spieler.y = constrain(spieler.y, 40, 260 - spieler.h);
59
60      if (trifft(spieler, muenze)) {
61          punkte = punkte + 1;
62          tone(LAUTSPRECHER, 1320, 60);
63          muenze.x = random(20, 220 - muenze.b);
64          muenze.y = random(40, 260 - muenze.h);
65      }
66
67      bild.fillScreen(tft.color565(30, 90, 40));
68      bild.drawRGBBitmap(muenze.x, muenze.y, muenze_0_pixel, muenze_0_maske, 16,
          16);
69      bild.fillRect(spieler.x, spieler.y, spieler.b, spieler.h, ST77XX_CYAN);
70      bild.setTextColor(ST77XX_WHITE);
```

```
71      bild.setTextSize(2);
72      bild.setCursor(70, 14);
73      bild.printf("Punkte: %d", punkte);
74      tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
75  }
```

Die Münze ist ein `Rechteck` mit 16 · 16 Pixeln, genauso groß wie ihr Bild. Nach einem Treffer bekommt sie sofort eine neue Position. Deshalb zählt jede Münze nur einmal. Die Grenzen bei `random` sorgen dafür, dass sie ganz im erlaubten Bereich liegt.



Aufgabe 4 – Durch die Wand

So läuft der Sketch aus der Aufgabe:

Serieller Monitor

```
x =  14
x =  26
x =  38
x =  50
x =  62
x =  74
x =  86
x =  98
x = 110
x = 122
x = 134
x = 146
x = 158
x = 170
x = 182
```

Die Wand reicht von x = 118 bis 121. Der Ball steht bei x = 110 (Pixel 110 bis 113), also noch davor, und im nächsten Bild bei x = 122, schon dahinter. In keinem Bild überlappen sich Ball und Wand.

Die Lösung zerlegt jeden Schritt in Teilschritte von einem Pixel:

k16_loesung4.ino

```
18      for (int n = 0; n < 15; n++) {
19          int schritt = 1;            // Richtung des Teilschritts: +1 oder -1
20          if (dx < 0) {
21              schritt = -1;
22          }
23          bool treffer = false;
```

```
24          for (int i = 0; i < abs(dx) && !treffer; i++) {
25              ball.x = ball.x + schritt;
26              if (trifft(ball, wand)) {
27                  ball.x = ball.x - schritt;      // zurueck vor die Wand
28                  treffer = true;
29              }
30          }
31          Serial.printf("x = %3d", ball.x);
32          if (treffer) {
33              Serial.print("  Treffer!");
34              dx = -dx;
35          }
36          Serial.println();
37      }
```

Serieller Monitor

```
x =  14
x =  26
x =  38
x =  50
x =  62
x =  74
x =  86
x =  98
x = 110
x = 114  Treffer!
x = 102
x =  90
x =  78
x =  66
x =  54
```

- Zeile 24: Die Schleife läuft höchstens `abs(dx)`-mal (`abs` liefert den Betrag, also 12 auch bei `dx = -12`) und hört früher auf, sobald es einen Treffer gab.
- Zeile 27: Nach einem Treffer geht der Ball einen Teilschritt zurück. Er steht dann genau vor der Wand (bei 114 berührt sein rechter Rand, Pixel 117, die Wand noch nicht).

Teilschritte kosten Rechenzeit. In Spielen genügt es meist, Hindernisse etwas dicker zu machen als das größte Tempo.

Aufgabe 5 – Die Mauer (`k16_loesung5`)

k16_loesung5.ino

```
15  #define SPALTEN 8
16  #define REIHEN 4
17  #define STEIN_B 26                  // Breite eines Steins
18  #define STEIN_H 10
19
20  struct Rechteck {
21      int x, y;
```

```
    int b, h;
};

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
GFXcanvas16 bild(240, 280);

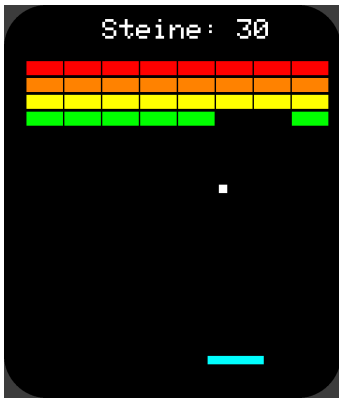
bool steine[SPALTEN * REIHEN];      // true: Stein ist noch da
int uebrig = SPALTEN * REIHEN;
float ball_x = 117;
float ball_y = 200;
float dx = 1.5;
float dy = -3.0;
int schlaeger_x = 100;
unsigned long letztes_bild = 0;

bool trifft(Rechteck a, Rechteck c) {
    return a.x < c.x + c.b && c.x < a.x + a.b &&
           a.y < c.y + c.h && c.y < a.y + a.h;
}

Rechteck stein_rechteck(int nummer) {
    int spalte = nummer % SPALTEN;
    int reihe = nummer / SPALTEN;
    Rechteck r = { 16 + spalte * (STEIN_B + 1), 40 + reihe * (STEIN_H + 2),
      STEIN_B, STEIN_H };
    return r;
}

void setup() {
    tft.init(240, 280);
    tft.setSPISpeed(62500000);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
    pinMode(TASTE_LINKS, INPUT_PULLUP);
    pinMode(TASTE_RECHTS, INPUT_PULLUP);
    for (int i = 0; i < SPALTEN * REIHEN; i++) {
        steine[i] = true;
    }
}
```

- `stein_rechteck(nummer)` rechnet aus der Nummer eines Steins sein Rechteck aus. Die Funktion gibt eine ganze Struktur zurück. Zum Prüfen und zum Zeichnen wird dieselbe Funktion benutzt, so passen beide immer zusammen.
- Zwischen den Steinen bleibt 1 Pixel Platz (`STEIN_B + 1`), zwischen den Reihen 2 Pixel.



k16_loesung5.ino

```
    Rechteck ball = { (int)ball_x, (int)ball_y, BALL, BALL };

    // Steine: hoechstens einen pro Bild abraeumen
    for (int i = 0; i < SPALTEN * REIHEN; i++) {
        if (steine[i] && trifft(ball, stein_rechteck(i))) {
            steine[i] = false;
            uebrig = uebrig - 1;
            dy = -dy;
            tone(LAUTSPRECHER, 880, 30);
            break;
        }
    }
```

Trifft der Ball einen Stein, verschwindet der Stein, und der Ball prallt senkrecht ab. `break` beendet die Schleife nach dem ersten getroffenen Stein. Ohne `break` könnte der Ball zwei nebeneinanderliegende Steine gleichzeitig treffen, `dy` würde zweimal umgedreht, und der Ball flöge durch die Mauer hindurch weiter.

Beim Zeichnen bestimmt die Reihe `i / SPALTEN` die Farbe. Sind alle Steine weg, erscheint „Geschafft!“.

Lösungen zu Kapitel 17: Der Neigungssensor

Aufgabe 1 – Kipp-Ampel (`k17_loesung1`)

k17_loesung1.ino

```
void loop() {
    float x, y, z;
    sensor.readAcceleration(x, y, z);
    // die tiefere Seite misst einen negativen Wert
    digitalWrite(LED_ROT, x > GRENZE);      // linker Rand tiefer
    digitalWrite(LED_GRUEN, x < -GRENZE);   // rechter Rand tiefer
    digitalWrite(LED_GELB, abs(x) <= GRENZE);
    delay(50);
}
```

Die tiefere Seite misst einen negativen Wert. Ist der **linke** Rand tiefer, zeigt die x-Achse (nach rechts) schräg nach oben, und x ist **positiv**. Deshalb leuchtet Rot bei `x > GRENZE`. `abs(x) <= GRENZE` heißt: x liegt zwischen −0,2 und 0,2. `digitalWrite` bekommt den Vergleich direkt: wahr schaltet die LED ein, falsch aus.

Im Prüflauf liegt der PicoBoy erst flach, wird dann nach rechts und danach nach links gekippt. Das Ereignisprotokoll zeigt die Pins 13 (gelb), 12 (grün) und 14 (rot) mit Zeit in ms und Zustand:

Ereignisse (PC-Prüflauf)

```
25 pin 13 1
1030 pin 12 1
1030 pin 13 0
2034 pin 14 1
2034 pin 12 0
3039 pin 14 0
3039 pin 13 1
3541 ende
```

Aufgabe 2 – Rohwerte vorhersagen

Im Bereich ±2 g sind 1024 Stufen ein g, 0,5 g also 512. Bei ±4 g sind es 512 Stufen pro g, also 256, bei ±8 g 128:

Serieller Monitor

```
+-2 g: roh x = 512
+-4 g: roh x = 256
+-8 g: roh x = 128
```

Am Gerät weichen die Zahlen etwas ab, weil man den PicoBoy nie genau auf 0,5 g kippt und die Werte zittern. Das Verhältnis 4 : 2 : 1 bleibt.

Aufgabe 3 – Schüttelzähler (`k17_loesung3`)

k17_loesung3.ino

```
16 void loop() {
17     float x, y, z;
18     sensor.readAcceleration(x, y, z);
19     float staerke = sqrt(x * x + y * y + z * z);   // Betrag in g, in Ruhe etwa 1
20     bool jetzt = staerke > SCHWELLE;
21     if (jetzt && !vorher) {
22         zaehler = zaehler + 1;
23         Serial.printf("%lu ms: Schuetteln Nr. %d (%.2f g)\n", millis(), zaehler,
             staerke);
24     }
25     vorher = jetzt;
26     delay(20);
27 }
```

Serieller Monitor

```
1016 ms: Schuetteln Nr. 1 (2.07 g)
1502 ms: Schuetteln Nr. 2 (1.99 g)
2210 ms: Schuetteln Nr. 3 (2.00 g)
```

- Zeile 19: Die Gesamtstärke ist die Länge des Pfeils aus x, y und z (Pythagoras in drei Richtungen). In Ruhe ist sie 1 g, egal wie das Gerät gehalten wird. Deshalb funktioniert der Zähler in jeder Lage.

- Zeile 20 bis 25 sind die Flankenerkennung aus Kapitel 9: gezählt wird nur der Moment, in dem die Stärke die Schwelle **überschreitet**, nicht jede Messung darüber.

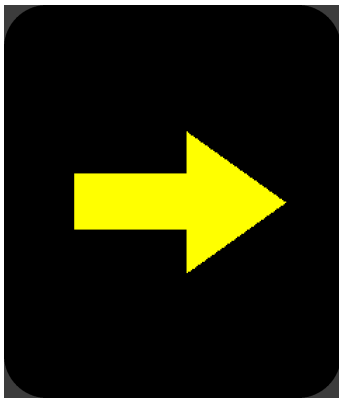
Im Prüflauf wird dreimal kurz geschüttelt. Am Gerät erzeugt ein einziges kräftiges Hin-und-her oft mehrere Spitzen kurz hintereinander. Wer das nicht will, sperrt nach jedem Zählen 200 ms lang (wie beim Entprellen der Tasten in Kapitel 9.3).

Aufgabe 4 – Ein Pfeil nach unten (`k17_loesung4`)

k17_loesung4.ino

```
24 void loop() {
25     float x, y, z;
26     sensor.readAcceleration(x, y, z);
27 
28     bild.fillScreen(ST77XX_BLACK);
29     uint16_t f = ST77XX_YELLOW;
30     if (abs(x) < GRENZE && abs(y) < GRENZE) {
31         bild.fillCircle(120, 140, 30, ST77XX_GREEN);           // flach
32     } else if (abs(x) > abs(y)) {
33         if (x < 0) {                                            // rechts tiefer
34             bild.fillTriangle(200, 140, 130, 90, 130, 190, f);
35             bild.fillRect(50, 120, 80, 40, f);
36         } else {                                                // links tiefer
37             bild.fillTriangle(40, 140, 110, 90, 110, 190, f);
38             bild.fillRect(110, 120, 80, 40, f);
39         }
40     } else {
41         if (y < 0) {                                            // unten tiefer
42             bild.fillTriangle(120, 240, 70, 170, 170, 170, f);
43             bild.fillRect(100, 60, 40, 110, f);
44         } else {                                                // oben tiefer
45             bild.fillTriangle(120, 40, 70, 110, 170, 110, f);
46             bild.fillRect(100, 110, 40, 110, f);
47         }
48     }
49     tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
50     delay(20);
51 }
```

`abs(x) > abs(y)` entscheidet, welche Neigung stärker ist. Danach sagt das Vorzeichen, welche Seite tiefer liegt. Ein Pfeil besteht aus einem Dreieck für die Spitze und einem Rechteck für den Schaft. Das Bild zeigt den PicoBoy mit tieferem rechtem Rand (x = −0,3 g, y = 0,1 g).

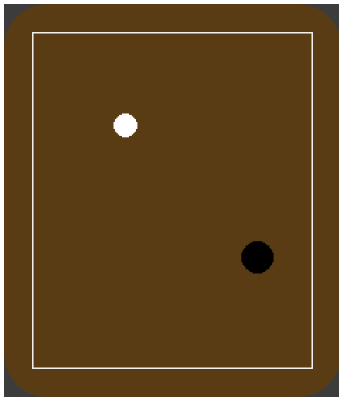


Aufgabe 5 – Kugel mit Schwung (`k17_loesung5`)

k17_loesung5.ino

```
45      if (!im_loch) {
46          float x, y, z;
47          sensor.readAcceleration(x, y, z);
48          vx = (vx - x * BESCHLEUNIGUNG) * REIBUNG;
49          vy = (vy - y * BESCHLEUNIGUNG) * REIBUNG;
50          kx = kx + vx;
51          ky = ky + vy;
52          // an den Waenden abprallen und dabei Schwung verlieren
53          if (kx < 20 + RADIUS || kx > 220 - RADIUS) {
54              kx = constrain(kx, 20 + RADIUS, 220 - RADIUS);
55              vx = -vx * 0.5;
56          }
57          if (ky < 20 + RADIUS || ky > 260 - RADIUS) {
58              ky = constrain(ky, 20 + RADIUS, 260 - RADIUS);
59              vy = -vy * 0.5;
60          }
61          // ins Loch: Mittelpunkt der Kugel liegt im Loch
62          float dx = kx - LOCH_X;
63          float dy = ky - LOCH_Y;
64          if (dx * dx + dy * dy < LOCH_R * LOCH_R) {
65              im_loch = true;
66              zeit = millis();
67              tone(LAUTSPRECHER, 660, 300);
68          }
69      }
```

- Zeile 48 und 49: Die Neigung ändert die Geschwindigkeit, nicht die Position. Danach nimmt die Reibung 2 % weg. Kippst du das Gerät zurück, rollt die Kugel deshalb noch ein Stück weiter, wie eine echte Kugel.
- Zeile 53 bis 60: Abprallen an den Wänden mit halbem Tempo.
- Zeile 62 bis 68: Die Kugel fällt ins Loch, wenn ihr Mittelpunkt im Loch liegt (Abstand kleiner als der Radius des Lochs, ohne Wurzel wie in Kapitel 16.3). Danach bewegt sie sich nicht mehr, und die Zeit seit dem Start wird angezeigt.



Im Prüflauf ist der PicoBoy nach rechts unten gekippt, und die Kugel rollt schräg in das Loch:

Zum Weiterbauen: Zeichne Hindernisse, an denen die Kugel abprallt (Rechtecke wie in Kapitel 16), oder Fallen, in denen sie verloren geht.



Lösungen zu Kapitel 18: Ein ganzes Spiel

Alle Lösungen sind Abwandlungen von `k18_spiel`. Hier stehen nur die geänderten und neuen Zeilen; die vollständigen Sketches liegen bei.

Aufgabe 1 – An den Schrauben drehen

Es genügen drei Zeilen im Kopf von `k18_spiel`:

```
#define METEORE 6
#define START_LEBEN 5
#define PUNKTE_PRO_LEVEL 10
```

Mehr Leben machen das Spiel leichter, mehr Meteore schwerer. Ein Level erst alle zehn Punkte heißt: Das Spiel wird langsamer schneller. Weil das Array `meteore[METEORE]` und alle Schleifen dieselbe Konstante benutzen, passt sich der Rest des Sketches von selbst an. Das ist der Vorteil von Konstanten statt fester Zahlen im Sketch.

Aufgabe 2 – Pause (`k18_loesung2`)

Neu sind die Taste B, ein vierter Zustand und eine zweite Flankenerkennung:



k18_loesung2.ino

```
26 enum Zustand { START, SPIEL, PAUSE, ENDE };
```

k18_loesung2.ino

```
69 // true genau einmal pro Druck auf B
70 bool b_gedrueckt() {
71     bool jetzt = digitalRead(TASTE_B) == LOW;
72     bool neu = jetzt && !b_vorher;
73     b_vorher = jetzt;
74     return neu;
75 }
```

In `loop()` bekommt `SPIEL` einen Übergang nach `PAUSE`, und `PAUSE` hat einen eigenen `case`:

k18_loesung2.ino

```
271         case SPIEL:
272             if (b_gedrueckt()) {
273                 zustand = PAUSE;
274             }
275             spiel_schritt();
276             zeichne_spiel();
277             break;
```

```
278         case PAUSE:
279             // nichts bewegen, nur das Spiel zeigen und darueber "PAUSE"
280             if (b_gedrueckt()) {
281                 zustand = SPIEL;
282             }
283             zeichne_spiel();
284             text_mitte("PAUSE", 120, 4, ST77XX_YELLOW);
285             break;
```

Im Zustand `PAUSE` wird nichts bewegt; `zeichne_spiel()` zeigt einfach den eingefrorenen Stand, darüber steht groß „PAUSE“. Ohne Flankenerkennung würde ein Druck auf B, der ja mehrere Bilder lang dauert, zwischen `SPIEL` und `PAUSE` hin- und herschalten.

Aufgabe 3 – Schutzschild (`k18_loesung3`)

k18_loesung3.ino

```
24 #define SCHUTZZEIT 1000          // ms nach einem Treffer ohne weiteren Schaden
```

k18_loesung3.ino

```
128     steuern();
129     Rechteck rakete = { (int)rakete_x + 4, RAKETE_Y + 2, 8, 20 };
130
131     // waehrend der Schutzzeit fliegen Meteore durch die Rakete hindurch
132     bool geschuetzt = millis() - treffer_zeit < SCHUTZZEIT;
133
134     // Meteore: fallen, treffen, kommen oben wieder
135     for (int i = 0; i < METEORE; i++) {
136         meteore[i].y = meteore[i].y + meteore[i].tempo;
137         Rechteck m = { (int)meteore[i].x + 2, (int)meteore[i].y + 2, 12, 12 };
138         if (!geschuetzt && trifft(rakete, m)) {
139             leben = leben - 1;
140             treffer_zeit = millis();
141             tone(LAUTSPRECHER, 150, 300);
142             meteor_nach_oben(i);
143         } else if (meteore[i].y > 280) {
144             meteor_nach_oben(i);
145         }
146     }
```

`treffer_zeit` gab es schon fürs Blinken. Jetzt entscheidet sie auch, ob ein Treffer zählt: Solange seit dem letzten Treffer weniger als `SCHUTZZEIT` vergangen ist, ist `geschuetzt` wahr, und `trifft` wird gar nicht erst geprüft. Weil Blinken und Schutz dieselbe Konstante benutzen, sieht der Spieler genau so lange ein Blinken, wie er geschützt ist.

Aufgabe 4 – Bestenliste (`k18_loesung4`)

k18_loesung4.ino

```
54 int beste[3] = { 0, 0, 0 };          // groesstes zuerst, nur im RAM
55 int platz = -1;                      // Platz des letzten Spiels, -1: keiner
```

k18_loesung4.ino

```
124 // ---- Bestenliste -------------------------------------------------------
125
126 // neues Ergebnis einsortieren: schlechtere rutschen eine Stelle nach hinten
127 void einsortieren(int wert) {
128     for (int i = 0; i < 3; i++) {
129         if (wert > beste[i]) {
130             for (int k = 2; k > i; k--) {
131                 beste[k] = beste[k - 1];
132             }
133             beste[i] = wert;
134             platz = i;
135             return;
136         }
137     }
138 }
139
140 void zeichne_bestenliste(int y) {
141     for (int i = 0; i < 3; i++) {
142         uint16_t farbe = ST77XX_CYAN;
143         if (i == platz) {
144             farbe = ST77XX_YELLOW;          // das eben erreichte Ergebnis
145         }
146         text_mitte(String(i + 1) + ". " + String(beste[i]), y + i * 20, 2,
              farbe);
147     }
148 }
```

- `einsortieren` sucht die erste Stelle, an der das neue Ergebnis besser ist als das dort stehende. Alle Ergebnisse ab dieser Stelle rutschen um eins nach hinten (die innere Schleife läuft **rückwärts**, sonst würde sie sich selbst überschreiben); das letzte fällt heraus. Dann kommt das neue hinein. Ist es schlechter als alle drei, ändert sich nichts.
- `platz` merkt sich, wo das letzte Ergebnis gelandet ist. Diese Zeile wird gelb hervorgehoben.



Im Prüflauf wurden mehrere Spiele hintereinander gespielt:

Serieller Monitor

```
Spiel vorbei. Punkte: 1, Bestenliste 1 0 0
Spiel vorbei. Punkte: 2, Bestenliste 2 1 0
Spiel vorbei. Punkte: 1, Bestenliste 2 1 1
Spiel vorbei. Punkte: 0, Bestenliste 2 1 1
Spiel vorbei. Punkte: 1, Bestenliste 2 1 1
Spiel vorbei. Punkte: 2, Bestenliste 2 2 1
```

Aufgabe 5 – Ein Extraleben (`k18_loesung5`)

Der Stern wird wiederverwendet: Ein `bool` merkt sich, ob er gerade ein Herz ist. Das wird jedes Mal neu ausgelost, wenn er oben startet:

k18_loesung5.ino

```
52 bool ist_herz = false;              // faellt gerade ein Herz statt des Sterns?
```

k18_loesung5.ino

```
87 void stern_nach_oben() {
88     stern.x = random(20, 220 - STERN_BREITE);
89     stern.y = random(-60, -STERN_HOEHE);
90     stern.tempo = 2.0 + level * 0.3;
91     ist_herz = random(10) == 0;     // etwa jedes zehnte Mal
92 }
```

`random(10)` liefert 0 bis 9; nur bei 0, also etwa jedes zehnte Mal, wird es ein Herz. Beim Einsammeln und beim Zeichnen unterscheidet der Sketch die beiden Fälle:

k18_loesung5.ino

```
150     if (trifft(rakete, s)) {
151         if (ist_herz) {
152             leben = min(leben + 1, MAX_LEBEN);
153             tone(LAUTSPRECHER, 880, 200);
154         } else {
155             punkte = punkte + 1;
156             tone(LAUTSPRECHER, 1320, 80);
157             if (punkte % PUNKTE_PRO_LEVEL == 0) {
158                 level = level + 1;
159             }
160         }
161         stern_nach_oben();
162     } else if (stern.y > 280) {
163         stern_nach_oben();
164     }
```

k18_loesung5.ino

```
203     if (ist_herz) {
204         bild.drawRGBBitmap(stern.x + 2, stern.y + 2, herz_pixel, herz_maske,
              HERZ_BREITE, HERZ_HOEHE);
205     } else {
206         bild.drawRGBBitmap(stern.x, stern.y, stern_pixel, stern_maske,
              STERN_BREITE, STERN_HOEHE);
207     }
```

`min(leben + 1, MAX_LEBEN)` liefert die kleinere der beiden Zahlen; mehr als fünf Leben gibt es also nie. Das Herz ist mit 8 · 8 Pixeln kleiner als der Stern (13 · 13); um 2 Pixel versetzt gezeichnet, liegt es ungefähr in der Mitte der Trefferbox des Sterns.

Lösungen zu Kapitel 19: Zeiger und Speicher

Aufgabe 1 – Was gibt das aus?

Serieller Monitor

```
a = 12, b = 1, *p = 1
20 30 40
```

- `p` zeigt auf `a`, `q` auf `b`. `*p = *p + *q;` rechnet 5 + 7 und schreibt das Ergebnis über `p` in `a`: `a` ist 12.
- `p = q;` ändert **den Zeiger**, nicht einen Wert: Ab jetzt zeigt `p` auf `b`.
- `*p = 1;` schreibt deshalb in `b`. Am Ende: `a` = 12, `b` = 1, `*p` = 1.
- `r` zeigt auf `zahlen[1]`, also auf 20. `r[1]` ist das Element danach (30), `*(r + 2)` das übernächste (40).

Aufgabe 2 – Begrenzen (`k19_loesung2`)

k19_loesung2.ino

```
// Loesung 2: einen Wert des Aufrufers in Grenzen halten
void begrenze(int *wert, int kleinster, int groesster) {
    if (*wert < kleinster) {
        *wert = kleinster;
    }
    if (*wert > groesster) {
        *wert = groesster;
    }
}

void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    int x = 250;
    int y = -12;
    int z = 100;
    begrenze(&x, 20, 220);
    begrenze(&y, 20, 260);
    begrenze(&z, 20, 220);
    Serial.printf("x = %d, y = %d, z = %d\n", x, y, z);
}

void loop() {
}
```

Serieller Monitor

```
x = 220, y = 20, z = 100
```

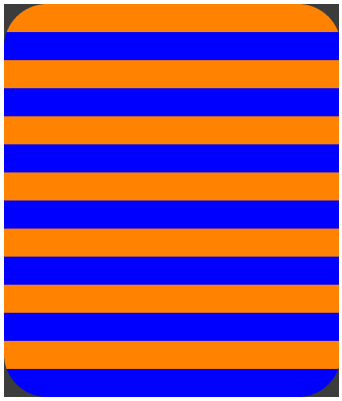
Beim Aufruf stehen die Adressen `&x`, `&y`, `&z`. Arduino hat für denselben Zweck übrigens `constrain(wert, min, max)`; das gibt den begrenzten Wert zurück, statt die Variable zu ändern.

Aufgabe 3 – Streifen mit `p++` (`k19_loesung3`)

k19_loesung3.ino

```
14 void setup() {
15     tft.init(240, 280);
16     pinMode(TFT_LICHT, OUTPUT);
17     digitalWrite(TFT_LICHT, HIGH);
18
19     uint16_t *p = bild.getBuffer();
20     for (int y = 0; y < HOEHE; y++) {
21         uint16_t farbe = ST77XX_ORANGE;
22         if ((y / 20) % 2 == 1) {
23             farbe = ST77XX_BLUE;
24         }
25         for (int x = 0; x < BREITE; x++) {
26             *p = farbe;          // Pixel setzen ...
27             p++;                 // ... und zum naechsten Pixel weiter
28         }
29     }
30     tft.drawRGBBitmap(0, 0, bild.getBuffer(), BREITE, HOEHE);
31 }
32
33 void loop() {
```

Weil die Pixel Zeile für Zeile lückenlos hintereinanderliegen, braucht der Zeiger keine Rechnung: Nach dem letzten Pixel einer Zeile steht er von selbst auf dem ersten der nächsten. `(y / 20) % 2` ist 0 für die Zeilen 0–19, 1 für 20–39 und so weiter.



Aufgabe 4 – Ein großes Level (`k19_loesung4`, `k19_loesung4_ohne`)

150 · 200 = 30 000 Felder zu je einem Byte, also 30 000 Byte.

k19_loesung4.ino

```
13 // nur die ersten Zeilen sind angegeben, der Rest ist 0
14 const uint8_t level[ZEILEN][SPALTEN] = {
15     { 1, 1, 1, 1, 1, 1, 1, 1 },
16     { 1, 0, 0, 0, 0, 0, 0, 1 },
17     { 1, 0, 2, 0, 0, 3, 0, 1 },
18     { 1, 1, 1, 1, 1, 1, 1, 1 },
19 };
```

Mit `const`:

```
Ausgabe
Der Sketch verwendet 91724 Bytes (1%) des Programmspeicherplatzes. Das Maximum sind
  8380416 Bytes.
Globale Variablen verwenden 10184 Bytes (1%) des dynamischen Speichers, 514104 Bytes
  für lokale Variablen verbleiben. Das Maximum sind 524288 Bytes.
```

Ohne `const` (`k19_loesung4_ohne`, sonst gleich):

```
Ausgabe
Der Sketch verwendet 61724 Bytes (0%) des Programmspeicherplatzes. Das Maximum sind
  8380416 Bytes.
Globale Variablen verwenden 40184 Bytes (7%) des dynamischen Speichers, 484104 Bytes
  für lokale Variablen verbleiben. Das Maximum sind 524288 Bytes.
```

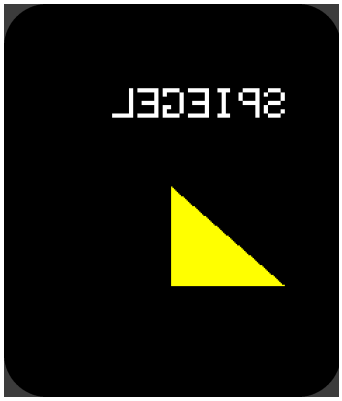
Verglichen mit `k19_leer` (Abschnitt 19.5) wächst mit `const` der Programmspeicher um rund 30 000 Byte: Das Level liegt im Flash. Ohne `const` wachsen die globalen Variablen um rund 30 000 Byte: Das Level liegt im RAM, seine Startwerte zusätzlich im Flash (die Meldung zählt sie nicht mit, siehe Abschnitt 19.7). Wenn sich das Level im Spiel nicht ändert, ist `const` die richtige Wahl.

Aufgabe 5 – Spiegelbild (`k19_loesung5`)

k19_loesung5.ino

```
13
14 void spiegeln(uint16_t *puffer) {
15     for (int y = 0; y < HOEHE; y++) {
16         uint16_t *links = puffer + y * BREITE;     // erstes Pixel der Zeile
17         uint16_t *rechts = links + BREITE - 1;     // letztes Pixel der Zeile
18         while (links < rechts) {
19             uint16_t merker = *links;
20             *links = *rechts;
21             *rechts = merker;
22             links++;
23             rechts--;
24         }
25     }
```

`links < rechts` vergleicht zwei Adressen: Solange der linke Zeiger noch vor dem rechten steht, gibt es etwas zu tauschen. Bei 240 Pixeln treffen sich die beiden nach 120 Tauschschritten in der Mitte.



Lösungen zu Kapitel 20: Eigene Bibliotheken

Aufgabe 1 – Wörter einfärben

keywords.txt

```
STK8BA58  KEYWORD1
begin     KEYWORD2
setRange  KEYWORD2
range     KEYWORD2
setOrientation    KEYWORD2
accelerationAvailable     KEYWORD2
readAcceleration  KEYWORD2
readAccelerationMS2       KEYWORD2
readRaw   KEYWORD2
readX     KEYWORD2
readY     KEYWORD2
readZ     KEYWORD2
readRegister      KEYWORD2
writeRegister     KEYWORD2
CHIP      LITERAL1
PICOBOY_COLOR_PLUS        LITERAL1
```

- `KEYWORD1` (Klassen): `STK8BA58`.
- `KEYWORD2` (Methoden): `begin`, `setRange`, `range`, `setOrientation`, `accelerationAvailable`, `readAcceleration`, `readAccelerationMS2`, `readRaw`, `readX`, `readY`, `readZ`, `readRegister`, `writeRegister`.
- `LITERAL1` (Konstanten): `CHIP`, `PICOBOY_COLOR_PLUS`.

Nicht darin stehen der Typ `Orientation` und die Konstanten `DEFAULT_ADDRESS` und `CHIP_ID`. Sie funktionieren trotzdem; der Editor färbt sie nur nicht ein. `readRegisters` fehlt zu Recht: Es ist `private`.

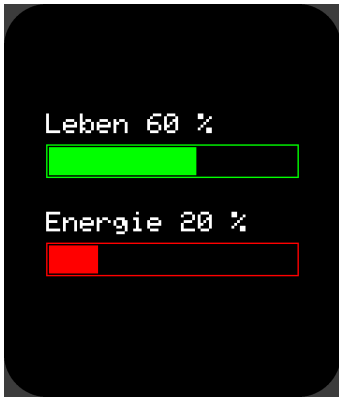
Aufgabe 2 – Warnfarbe (`k20_loesung2`)

Nur die Methode `zeichne` in `Balken.h` ändert sich:

Balken.h

```
12      // zeichnet Rahmen und Fuellung; prozent von 0 bis 100
13      void zeichne(Adafruit_GFX &ziel, int prozent) {
14          prozent = constrain(prozent, 0, 100);
15          int innen = _breite - 4;
16          int voll = innen * prozent / 100;
17          uint16_t farbe = _farbe;
18          if (prozent < 25) {
19              farbe = 0xF800;                  // rot
20          }
21          ziel.drawRect(_x, _y, _breite, _hoehe, farbe);
22          ziel.fillRect(_x + 2, _y + 2, voll, _hoehe - 4, farbe);
23          ziel.fillRect(_x + 2 + voll, _y + 2, innen - voll, _hoehe - 4, 0x0000);
24      }
```

Die Farbe kommt in eine lokale Variable `farbe`. Das Attribut `_farbe` bleibt unverändert, damit der Balken wieder grün wird, sobald der Wert über 25 % steigt.



Aufgabe 3 – Eine Klasse für Tasten (`k20_loesung3`)

Taste.h

```
// Taste.h – eine Taste des PicoBoy mit Flankenerkennung und Entprellen
#pragma once
#include <Arduino.h>

class Taste {
public:
    Taste(int pin) : _pin(pin) {
    }

    // in setup() aufrufen (wie begin() bei den Bibliotheken)
    void beginne() {
        pinMode(_pin, INPUT_PULLUP);
    }

    // true, solange die Taste unten ist
    bool gedrueckt() {
        return digitalRead(_pin) == LOW;
    }

    // true genau einmal pro Druck; Prellen kuerzer als 30 ms zaehlt nicht
    bool neu_gedrueckt() {
        bool jetzt = gedrueckt();
        bool neu = false;
        if (jetzt != _vorher && millis() - _letzte_aenderung > 30) {
            _letzte_aenderung = millis();
            neu = jetzt;
            _vorher = jetzt;
        }
        return neu;
    }

private:
    int _pin;
    bool _vorher = false;
    unsigned long _letzte_aenderung = 0;
};
```

k20_loesung3.ino

```ino
// Loesung 3: die Klasse Taste aus dem Tab Taste.h
#include "Taste.h"

#define TASTE_A 27
#define TASTE_B 28

Taste a(TASTE_A);
Taste b(TASTE_B);
int zaehler_a = 0;
int zaehler_b = 0;

void setup() {
    Serial.begin(115200);
    a.beginne();
    b.beginne();
}

void loop() {
    if (a.neu_gedrueckt()) {
        zaehler_a++;
        Serial.printf("A gedrueckt (%d)\n", zaehler_a);
    }
    if (b.neu_gedrueckt()) {
        zaehler_b++;
        Serial.printf("B gedrueckt (%d)\n", zaehler_b);
    }
}
```

Serieller Monitor

```
A gedrueckt (1)
A gedrueckt (2)
B gedrueckt (1)
```

`neu_gedrueckt()` merkt sich den letzten Zustand und den Zeitpunkt der letzten Änderung. Ein Wechsel zählt nur, wenn seit dem letzten mehr als 30 ms vergangen sind; so fällt das Prellen weg. Geliefert wird `true` nur beim Wechsel nach „gedrückt“.

`pinMode` gehört nicht in den Konstruktor: Die Objekte `a` und `b` sind global, ihr Konstruktor läuft also vor `setup()`, bevor der Kern alles vorbereitet hat. Darum gibt es `beginne()`, das wie `begin()` bei den Bibliotheken in `setup()` aufgerufen wird.

Aufgabe 4 – Deine PicoBoy-Bibliothek (`k20_loesung4`)

PicoBoy.h

```h
// PicoBoy.h – alle Pins des PicoBoy Color Plus und ein Start in einer Zeile
#pragma once
#include <Adafruit_ST7789.h>

#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
```

```
#define TFT_LICHT 26
#define LAUTSPRECHER 15
#define LED_ROT 14
#define LED_GELB 13
#define LED_GRUEN 12
#define RGB_LED 11
#define TASTE_HOCH 4
#define TASTE_RUNTER 2
#define TASTE_LINKS 3
#define TASTE_RECHTS 1
#define TASTE_MITTE 0
#define TASTE_A 27
#define TASTE_B 28
#define AKKU A3

// Display, Licht, Tasten und LEDs einrichten.
// inline: Die Funktion darf in mehreren .cpp-Dateien eingebunden werden.
inline void picoboy_start(Adafruit_ST7789 &tft) {
    tft.init(240, 280);
    pinMode(TFT_LICHT, OUTPUT);
    digitalWrite(TFT_LICHT, HIGH);
    tft.fillScreen(ST77XX_BLACK);
    const int tasten[7] = { TASTE_HOCH, TASTE_RUNTER, TASTE_LINKS, TASTE_RECHTS,
                            TASTE_MITTE, TASTE_A, TASTE_B };
    for (int i = 0; i < 7; i++) {
        pinMode(tasten[i], INPUT_PULLUP);
    }
    pinMode(LED_ROT, OUTPUT);
    pinMode(LED_GELB, OUTPUT);
    pinMode(LED_GRUEN, OUTPUT);
}
```

k20_loesung4.ino

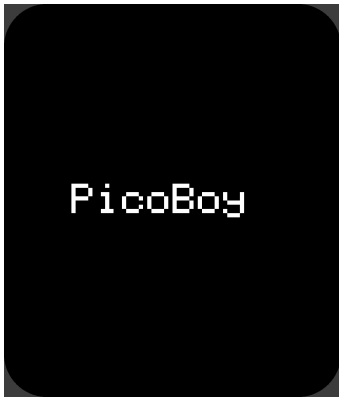
```
// Loesung 4: Sketch mit der eigenen Datei PicoBoy.h
#include "PicoBoy.h"

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);

void setup() {
    picoboy_start(tft);
    tft.setTextColor(ST77XX_WHITE);
    tft.setTextSize(3);
    tft.setCursor(48, 128);
    tft.print("PicoBoy");
}

void loop() {
    digitalWrite(LED_GRUEN, digitalRead(TASTE_A) == LOW);
}
```

Die Funktion `picoboy_start` steht außerhalb einer Klasse in einer `.h`-Datei. Solange nur der Sketch sie einbindet, ist das gleich. In einer Bibliothek mit mehreren `.cpp`-Dateien gäbe es sie aber mehrfach (`multiple definition`). Deshalb steht `inline` davor (Abschnitt 20.3). Die Referenz `Adafruit_ST7789 &tft` sorgt dafür, dass die Funktion das Display-Objekt des Sketches einrichtet, keine Kopie.



Für die Bibliothek kommt `PicoBoy.h` nach `libraries/PicoBoy/src/`, dazu eine `library.properties`, zum Beispiel:

```
name=PicoBoy
version=1.0.0
author=Dein Name
maintainer=Dein Name <dein.name@example.org>
sentence=Pins und Start fuer den PicoBoy Color Plus.
paragraph=Alle Pins als #define und eine Funktion, die Display, Tasten und LEDs
  einrichtet.
category=Other
url=https://example.org/picoboy
architectures=rp2040
depends=Adafruit ST7735 and ST7789 Library
includes=PicoBoy.h
```

`architectures=rp2040` statt `*`: Die Pins gelten nur für den PicoBoy, und der Kern von Earle Philhower meldet sich für RP2040 und RP2350 als Architektur `rp2040`. Im Sketch steht dann `#include <PicoBoy.h>`.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 21: PWM und ADC

Aufgabe 1 – Rechnen wie der ADC

a) Am Pin: 900 · 3,3 V / 4095 ≈ 0,73 V. Die Batterien haben zusammen das Dreifache, also etwa 2,18 V – fast leer.

b) Mit 10 Bit schiebt der Kern den 12-Bit-Wert um zwei Stellen nach rechts, teilt also durch 4: 900 / 4 = 225.

c) 3,2 V / 3 ≈ 1,07 V am Pin. 1,07 V / 3,3 V · 4095 ≈ 1324.

Aufgabe 2 – Sanft einschalten (`k21_loesung2`)

k21_loesung2.ino

```
11 void setup() {
12     tft.init(240, 280);
13     analogWrite(TFT_LICHT, 0);              // erst dunkel ...
14     tft.fillScreen(ST77XX_BLUE);
15     tft.setTextColor(ST77XX_WHITE);
16     tft.setTextSize(3);
17     tft.setCursor(48, 128);
18     tft.print("Guten Tag");
19     for (int wert = 0; wert <= 255; wert = wert + 5) {
20         analogWrite(TFT_LICHT, wert);       // ... dann in 52 Schritten hell
21         delay(20);
22     }
23 }
```

Die Schleife zählt in Fünferschritten von 0 bis 255: 52 Schritte zu je 20 ms, gut eine Sekunde. Weil das Licht während des Zeichnens noch aus ist, erscheint das Bild „aus dem Dunkeln“.

Aufgabe 3 – Batterien wechseln! (`k21_loesung3`)

k21_loesung3.ino

```
25 void loop() {
26     float volt = analogRead(BATTERIE) * 3.3 / 4095 * 3;
27     tft.fillRect(20, 100, 200, 80, ST77XX_BLACK);
28     tft.setCursor(20, 110);
29     if (volt < GRENZE) {
30         tft.setTextColor(ST77XX_RED);
31         tft.print("Wechseln!");
32         led_an = !led_an;                    // blinken
33     } else {
34         tft.setTextColor(ST77XX_GREEN);
35         tft.print("Batterie ok");
36         led_an = false;
37     }
38     digitalWrite(LED_ROT, led_an);
39     tft.setTextColor(ST77XX_WHITE);
40     tft.setCursor(20, 150);
41     tft.printf("%.2f V", volt);
42     delay(500);
43 }
```



Das Bild entstand im Prüflauf mit 2,1 V. `led_an = !led_an` kehrt den Zustand bei jedem Durchlauf um; bei 500 ms Pause blinkt die LED also einmal pro Sekunde. `fillRect` löscht nur den Bereich mit dem Text, bevor er neu geschrieben wird.

Aufgabe 4 – Energiesparmodus (`k21_loesung4`)

k21_loesung4.ino

```
void loop() {
    bool taste = false;
    for (int i = 0; i < 7; i++) {
        if (digitalRead(TASTEN[i]) == LOW) {
            taste = true;
        }
    }
    if (taste) {
        letzte_taste = millis();
        if (gedimmt) {
            analogWrite(TFT_LICHT, HELL);
            gedimmt = false;
            Serial.printf("%lu ms: hell\n", millis());
        }
    } else if (!gedimmt && millis() - letzte_taste > WARTEZEIT) {
        analogWrite(TFT_LICHT, DUNKEL);
        gedimmt = true;
        Serial.printf("%lu ms: gedimmt\n", millis());
    }
    delay(20);
}
```

Serieller Monitor

```
14094 ms: gedimmt
16015 ms: hell
26103 ms: gedimmt
```

Im Prüflauf wurde nach 4 s A und nach 16 s B gedrückt. Jeder Tastendruck setzt `letzte_taste` neu. Gedimmt wird 10 s nach dem letzten Druck, also bei etwa 14 s; der Druck auf B macht das Licht wieder hell. `gedimmt` sorgt dafür, dass `analogWrite` und die Ausgabe nur einmal pro Wechsel kommen.

Aufgabe 5 – Gleichmäßige Stufen (`k21_loesung5`)

k21_loesung5.ino

```
void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    tft.init(240, 280);
    tft.fillScreen(ST77XX_WHITE);
    pinMode(TASTE_HOCH, INPUT_PULLUP);
    pinMode(TASTE_RUNTER, INPUT_PULLUP);
    for (int i = 0; i < 10; i++) {
        // Gammakorrektur: Anteil hoch 2,2
        stufen[i] = round(1 + 254 * pow(i / 9.0, 2.2));
```

```
29          Serial.printf("Stufe %d: %d\n", i, stufen[i]);
30      }
31      analogWrite(TFT_LICHT, stufen[stufe]);
32  }
```

Serieller Monitor

```
Stufe 0: 1
Stufe 1: 3
Stufe 2: 10
Stufe 3: 24
Stufe 4: 44
Stufe 5: 71
Stufe 6: 105
Stufe 7: 147
Stufe 8: 197
Stufe 9: 255
```

`1 + 254 * …` sorgt dafür, dass die kleinste Stufe 1 statt 0 ist (sonst wäre das Licht ganz aus). `round` rundet auf die nächste ganze Zahl. Die unteren Stufen liegen dicht beieinander, die oberen weit auseinander – genau so, dass das Auge gleich große Schritte sieht. Probiere auch andere Werte als 2,2: Mit 1,0 sind die Zahlen gleichmäßig, die Helligkeit wirkt es nicht mehr.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 22: I2C mit `Wire`

Aufgabe 1 – Hex und binär

Jede Hex-Ziffer steht für genau vier Bits:

Hex	binär
`0x87`	`1000 0111`
`0xB6`	`1011 0110`
`0x18`	`0001 1000`

- `0xB6 & 0xF0` behält nur die oberen vier Bits: `1011 0000` = `0xB0`.
- `0x05 << 4` schiebt um eine Hex-Ziffer nach links: `0x50` (= 80).
- Bytes `F0 FF`: `0xFF << 8 | (0xF0 & 0xF0)` = `0xFFF0`. Als `int16_t` ist das −16, um 4 nach rechts geschoben −1. Die kleinste Stufe unter null.

Aufgabe 2 – Scanner als Tabelle (`k22_loesung2`)

k22_loesung2.ino

```
13      Serial.println("     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F");
14      for (int zeile = 0; zeile < 8; zeile++) {
15          Serial.printf("%X0: ", zeile);
```

```
        for (int spalte = 0; spalte < 16; spalte++) {
            int adresse = zeile * 16 + spalte;
            if (adresse < 1 || adresse > 126) {
                Serial.print("   ");
            } else {
                Wire.beginTransmission(adresse);
                if (Wire.endTransmission() == 0) {
                    Serial.printf("%02X ", adresse);
                } else {
                    Serial.print("-- ");
                }
            }
        }
        Serial.println();
    }
```

Serieller Monitor

```
     0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
00:    -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- 18 -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
```

`%X0` gibt die Zeilennummer als Hex-Ziffer aus, gefolgt von einer 0. Die Adressen 0 und 127 sind für besondere Zwecke reserviert und werden nicht abgefragt. Werkzeuge auf Linux-Rechnern (`i2cdetect`) zeigen dieselbe Art Tabelle.

Aufgabe 3 – Messbereich ±4 g (`k22_loesung3`)

k22_loesung3.ino

```
uint8_t lese_register(uint8_t reg) {
    Wire.beginTransmission(SENSOR);
    Wire.write(reg);
    Wire.endTransmission(false);
    Wire.requestFrom(SENSOR, 1);
    return Wire.read();
}

void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    Wire.setSDA(20);
    Wire.setSCL(21);
    Wire.begin();
    schreibe_register(0x14, 0xB6);
    delay(5);
```

```
    schreibe_register(0x11, 0x00);
    schreibe_register(0x10, 0x0C);
    schreibe_register(0x0F, 0x05);              // +-4 g
    Serial.printf("Register 0x0F: 0x%02X\n", lese_register(0x0F));
    delay(20);                                  // erste Messung abwarten

    uint8_t z1 = lese_register(0x06);
    uint8_t z2 = lese_register(0x07);
    int16_t z = (int16_t)(z2 << 8 | (z1 & 0xF0)) >> 4;
    Serial.printf("z roh: %d  ->  %.2f g\n", z, z / 512.0);
}
```

Serieller Monitor

```
Register 0x0F: 0x05
z roh: -512  ->  -1.00 g
```

Der Bereich ist doppelt so groß, die 12 Bit bleiben: Jetzt sind 512 Stufen 1 g, jede Stufe ist doppelt so grob. Die Bibliothek rechnet genauso (`proG = 2048.0f / _range`). Die beiden Register werden hier einzeln gelesen; schneller und sicherer ist es, sie wie in Abschnitt 22.5 in einem Rutsch zu lesen, damit beide Bytes zur selben Messung gehören.

Aufgabe 4 – Vorher und nachher (`k22_loesung4`)

k22_loesung4.ino

```
void zeige_register() {
    const uint8_t NUMMERN[4] = { 0x00, 0x0F, 0x10, 0x11 };
    for (int i = 0; i < 4; i++) {
        Serial.printf("  0x%02X: 0x%02X\n", NUMMERN[i],
          lese_register(NUMMERN[i]));
    }
}

void setup() {
    Serial.begin(115200);
    while (!Serial) {
        delay(10);
    }
    Wire.setSDA(20);
    Wire.setSCL(21);
    Wire.begin();
    schreibe_register(0x14, 0xB6);
    delay(5);
    Serial.println("nach dem Neustart:");
    zeige_register();
    schreibe_register(0x11, 0x00);
    schreibe_register(0x10, 0x0C);
    schreibe_register(0x0F, 0x03);
    Serial.println("nach dem Einrichten:");
    zeige_register();
}
```

Serieller Monitor

```
nach dem Neustart:
  0x00: 0x87
  0x0F: 0x03
  0x10: 0x1F
  0x11: 0x00
nach dem Einrichten:
  0x00: 0x87
  0x0F: 0x03
  0x10: 0x0C
  0x11: 0x00
```

Die Chip-ID bleibt immer `0x87`. Geändert haben sich Register `0x10` (Bandbreite auf `0x0C`) und die Einstellungen, die der Sketch geschrieben hat. Die Werte direkt nach dem Neustart kommen hier aus dem Prüflauf auf dem PC; welche Startwerte der echte Chip hat, steht in seinem Datenblatt. Prüfe es mit diesem Sketch an deinem PicoBoy.

Aufgabe 5 – Schüttelzähler (`k22_loesung5`)

k22_loesung5.ino

```
24 float staerke() {
25     uint8_t b[6];
26     Wire.beginTransmission(SENSOR);
27     Wire.write(0x02);
28     Wire.endTransmission(false);
29     Wire.requestFrom(SENSOR, 6);
30     for (int i = 0; i < 6; i++) {
31         b[i] = Wire.read();
32     }
33     float x = ((int16_t)(b[1] << 8 | (b[0] & 0xF0)) >> 4) / 1024.0;
34     float y = ((int16_t)(b[3] << 8 | (b[2] & 0xF0)) >> 4) / 1024.0;
35     float z = ((int16_t)(b[5] << 8 | (b[4] & 0xF0)) >> 4) / 1024.0;
36     return sqrt(x * x + y * y + z * z);   // Betrag: Drehung egal
37 }
```



k22_loesung5.ino

```
66 void loop() {
67     float s = staerke();
68     // hoechstens ein Schuettler pro halbe Sekunde
69     if (s > SCHWELLE && millis() - letzter > 500) {
70         letzter = millis();
71         schuettler++;
72         Serial.printf("%lu ms: %.2f g -> Schuettler %d\n", millis(), s,
             schuettler);
73         tone(LAUTSPRECHER, 660, 80);
74         zeige();
75     }
76     delay(10);
77 }
```

Serieller Monitor

```
1004 ms: 1.96 g -> Schuettler 1
2500 ms: 1.85 g -> Schuettler 2
```

Im Prüflauf gab es drei Stöße: nach 1,0 s, 1,2 s und 2,5 s. Der zweite kam weniger als 500 ms nach dem ersten und zählt deshalb nicht. Die Stärke hängt nicht davon ab, wie der PicoBoy gehalten wird; deshalb braucht `staerke()` die Achsen nicht auf den PicoBoy umzurechnen. Bei ±2 g kann eine Achse höchstens 2 g zeigen; für die Schwelle 1,6 g reicht das.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 23: SPI und das Display

Aufgabe 1 – Nachgerechnet

240 · 280 Pixel · 2 Byte = 134 400 Byte = 1 075 200 Bit.

SPI-Takt	ein Bild	höchstens Bilder pro Sekunde
25 MHz	1 075 200 / 25 000 000 s ≈ 43 ms	etwa 23
37,5 MHz	≈ 28,7 ms	etwa 34
75 MHz	≈ 14,3 ms	etwa 69

Das sind Obergrenzen: Das Zeichnen in den Canvas kostet zusätzlich Zeit.

Aufgabe 2 – Umkehren auf Knopfdruck (`k23_loesung2`)

k23_loesung2.ino

```
11 bool invertiert = true;          // init() schickt INVON
12 bool a_vorher = false;
13
14 void setup() {
15     tft.init(240, 280);
16     pinMode(TFT_LICHT, OUTPUT);
17     digitalWrite(TFT_LICHT, HIGH);
18     pinMode(TASTE_A, INPUT_PULLUP);
19     tft.fillScreen(ST77XX_WHITE);
20     tft.fillCircle(120, 140, 60, ST77XX_RED);
21 }
22
23 void loop() {
24     bool a = digitalRead(TASTE_A) == LOW;
25     if (a && !a_vorher) {
26         invertiert = !invertiert;
27         tft.invertDisplay(invertiert);
28     }
29     a_vorher = a;
30     delay(20);
31 }
```

Anfangen musst du mit `true`: `init()` hat INVON geschickt, und das ist bei diesem Display die normale Darstellung. Der erste Druck schickt dann `invertDisplay(false)`, also INVOFF, und kehrt die Farben um: Der Grund wird schwarz, der Kreis türkis. Der Sketch läuft nur auf dem PicoBoy, weil der Prüflauf auf dem PC die Farbumkehr des Chips nicht nachbildet.

Aufgabe 3 – Alle vier Richtungen (`k23_loesung3`)

k23_loesung3.ino

```
14 void zeichne() {
15     tft.setRotation(richtung);
16     tft.fillScreen(ST77XX_BLACK);
17     tft.setTextColor(ST77XX_WHITE);
18     tft.setTextSize(3);
19     tft.setCursor(tft.width() / 2 - 36, 40);
20     tft.print("oben");
21     tft.setTextSize(2);
22     tft.setCursor(tft.width() / 2 - 66, tft.height() / 2);
23     tft.printf("Rotation %d", richtung);
24     tft.setCursor(tft.width() / 2 - 42, tft.height() / 2 + 30);
25     tft.printf("%d x %d", tft.width(), tft.height());
26 }
```

Die Texte werden mit `tft.width() / 2` ungefähr in die Mitte gesetzt; so stimmt es im Hoch- wie im Querformat. Bei Rotation 1 und 3 ist das Display 280 · 240, bei 0 und 2 240 · 280. Das Bild zeigt Rotation 1; bei Rotation 2 steht alles auf dem Kopf.



Aufgabe 4 – Nur den Ausschnitt senden (`k23_loesung4`)

k23_loesung4.ino

```
28 void loop() {
29     x = x + dx;
30     y = y + dy;
31     if (x < 20 || x > 240 - 20 - KLEIN) {
32         dx = -dx;
33     }
34     if (y < 20 || y > 280 - 20 - KLEIN) {
35         dy = -dy;
36     }
37     // Der Ausschnitt ist groesser als der Ball: sein schwarzer Rand
38     // loescht die alte Stelle gleich mit.
39     ausschnitt.fillScreen(ST77XX_BLACK);
40     ausschnitt.fillCircle(KLEIN / 2, KLEIN / 2, 12, ST77XX_YELLOW);
41     unsigned long start = micros();
42     tft.drawRGBBitmap(x, y, ausschnitt.getBuffer(), KLEIN, KLEIN);
43     summe_us = summe_us + (micros() - start);
44     bilder++;
45     if (bilder == 100) {
46         Serial.printf("Ausschnitt %d x %d: im Mittel %lu us pro Bild\n", KLEIN,
           KLEIN, summe_us / 100);
47         bilder = 0;
48         summe_us = 0;
49     }
50     delay(10);
51 }
```

Serieller Monitor

```
Ausschnitt 40 x 40: im Mittel 686 us pro Bild
Ausschnitt 40 x 40: im Mittel 686 us pro Bild
```

40 · 40 Pixel sind 3 200 Byte statt 134 400: Das Senden dauert nur einen Bruchteil eines Vollbilds (die Zeiten stammen aus dem Prüflauf, siehe Abschnitt 23.3). Der Ausschnitt ist größer als der Ball, weil sein schwarzer Rand die Stelle übermalt, an der der Ball eben noch war. Der Ball bewegt sich höchstens 3 Pixel pro Bild, der Rand ist 8 Pixel breit; das reicht. Ohne diesen Rand bliebe eine Spur stehen. Das Verfahren klappt nur, solange sich wenig ändert und der Hintergrund einfarbig ist.

Aufgabe 5 – Schachbrett ohne Bibliothek (`k23_loesung5`)

Die Funktionen `start`, `ende`, `kommando`, `befehl`, `befehl_mit_wert` und `fuelle` sowie die Startbefehle sind dieselben wie in `k23_spi_direkt`. Neu ist das Brett:

k23_loesung5.ino

```
80
81      unsigned long beginn = millis();
82      for (int zeile = 0; zeile < 280 / FELD + 1; zeile++) {
83          for (int spalte = 0; spalte < 240 / FELD; spalte++) {
84              uint16_t farbe = 0xFFFF;
85              if ((zeile + spalte) % 2 == 1) {
86                  farbe = 0x0000;
87              }
88              int hoehe = FELD;
89              if (zeile * FELD + hoehe > 280) {
90                  hoehe = 280 - zeile * FELD;     // letzte Reihe ist niedriger
91              }
92              fuelle(spalte * FELD, zeile * FELD, FELD, hoehe, farbe);
93          }
94      }
95      Serial.printf("Schachbrett: %lu ms\n", millis() - beginn);
96  }
```

10 Reihen zu je 8 Feldern; die letzte Reihe beginnt bei y = 270 und ist nur 10 Pixel hoch. Jedes Feld ist ein eigenes Fenster mit Befehlen. Die Zahl der Pixel ist dieselbe wie bei einem einzigen `fuelle` über den ganzen Bildschirm, dazu kommen 80 Fenster. Weil `fuelle` jedes Pixel einzeln mit `SPI.transfer16` schickt und auf das Ende wartet, ist dieser Sketch insgesamt deutlich langsamer als `drawRGBBitmap` der Bibliothek. Die Messung gibt es nur am Gerät.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 24: Interrupts und der zweite Kern

Aufgabe 1 – A und B zählen (`k24_loesung1`)

k24_loesung1.ino

```
1  // Loesung 24.1: Druecke auf A und B mit je einer ISR zaehlen
2  #define TASTE_A 27
3  #define TASTE_B 28
4
5  volatile int zaehler_a = 0;
6  volatile int zaehler_b = 0;
7
8  void taste_a_gedrueckt() {
9      zaehler_a++;
10 }
```

```
12 void taste_b_gedrueckt() {
13     zaehler_b++;
14 }
15
16 void setup() {
17     Serial.begin(115200);
18     pinMode(TASTE_A, INPUT_PULLUP);
19     pinMode(TASTE_B, INPUT_PULLUP);
20     attachInterrupt(digitalPinToInterrupt(TASTE_A), taste_a_gedrueckt, FALLING);
21     attachInterrupt(digitalPinToInterrupt(TASTE_B), taste_b_gedrueckt, FALLING);
22 }
23
24 void loop() {
25     Serial.printf("A: %d  B: %d\n", zaehler_a, zaehler_b);
26     delay(1000);
27 }
```

Jede Taste bekommt ihre eigene ISR und ihren eigenen `volatile`-Zähler. Im Prüflauf wurde A dreimal und B zweimal gedrückt:

Serieller Monitor

```
A: 0  B: 0
A: 1  B: 1
A: 3  B: 2
```

Aufgabe 2 – Was ist hier falsch? (`k24_loesung2`)

Die Fehler in `k24_aufgabe2`:

1. `treffer` ist nicht `volatile` (Zeile 5). `loop()` liest die Variable, die ISR schreibt sie.
2. `Serial.printf` in der ISR (Zeile 9): Hält `loop()` die Sperre der USB-Verbindung gerade selbst, geht der Text verloren.
3. `tone` in der ISR (Zeile 10): ebenfalls mit einer Sperre geschützt und legt Speicher an.
4. `delay(200)` in der ISR (Zeile 11): Warten ist in einer ISR verboten, und das ganze Programm stünde so lange still.

Außerdem setzt `loop()` in Zeile 23 den Zähler zurück, während die ISR ihn jederzeit verändern kann. Die Lösung schützt das mit `noInterrupts()`:

k24_loesung2.ino

```
 1 // Loesung 24.2: Die ISR zaehlt nur noch, alles andere macht loop()
 2 #define TASTE_A 27
 3 #define LAUTSPRECHER 15
 4
 5 volatile int treffer = 0;
 6 int gemeldet = 0;
 7
 8 void taste_a_gedrueckt() {
 9     treffer++;
10 }
```

```
12 void setup() {
13     Serial.begin(115200);
14     pinMode(TASTE_A, INPUT_PULLUP);
15     attachInterrupt(digitalPinToInterrupt(TASTE_A), taste_a_gedrueckt, FALLING);
16 }
17
18 void loop() {
19     if (treffer != gemeldet) {
20         gemeldet = treffer;
21         Serial.printf("Treffer: %d\n", gemeldet);
22         tone(LAUTSPRECHER, 440, 200);   // wartet nicht
23     }
24     if (gemeldet >= 10) {
25         Serial.println("Gewonnen!");
26         noInterrupts();
27         treffer = 0;
28         interrupts();
29         gemeldet = 0;
30     }
31     delay(10);
32 }
```

Der Ton kommt jetzt aus `loop()` und mit Dauer: `tone(LAUTSPRECHER, 440, 200)` wartet nicht, ein `noTone` ist überflüssig. `gemeldet` merkt sich, welcher Stand zuletzt ausgegeben wurde. Im Prüflauf wurde A elfmal gedrückt:

Serieller Monitor

```
Treffer: 1
Treffer: 2
Treffer: 3
Treffer: 4
Treffer: 5
Treffer: 6
Treffer: 7
Treffer: 8
Treffer: 9
Treffer: 10
Gewonnen!
Treffer: 1
```

Aufgabe 3 – Duell (`k24_loesung3`)

k24_loesung3.ino

```
1 // Loesung 24.3: Duell - wer drueckt nach dem Signal zuerst, A oder B?
2 #include <Adafruit_ST7789.h>
3
4 #define TFT_CS 10
5 #define TFT_DC 8
6 #define TFT_RST 9
7 #define TFT_LICHT 26
8 #define TASTE_A 27
9 #define TASTE_B 28
```

```
11 Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
12
13 volatile char sieger = ' ';     // ' ' = noch niemand
14 volatile unsigned long sieger_zeit = 0;
15
16 void taste_a_gedrueckt() {
17     if (sieger == ' ') {
18         sieger = 'A';
19         sieger_zeit = millis();
20     }
21 }
22
23 void taste_b_gedrueckt() {
24     if (sieger == ' ') {
25         sieger = 'B';
26         sieger_zeit = millis();
27     }
28 }
29
30 void setup() {
31     Serial.begin(115200);
32     tft.init(240, 280);
33     pinMode(TFT_LICHT, OUTPUT);
34     digitalWrite(TFT_LICHT, HIGH);
35     pinMode(TASTE_A, INPUT_PULLUP);
36     pinMode(TASTE_B, INPUT_PULLUP);
37     attachInterrupt(digitalPinToInterrupt(TASTE_A), taste_a_gedrueckt, FALLING);
38     attachInterrupt(digitalPinToInterrupt(TASTE_B), taste_b_gedrueckt, FALLING);
39     tft.setTextColor(ST77XX_WHITE);
40     tft.setTextSize(3);
41 }
42
43 void loop() {
44     tft.fillScreen(ST77XX_RED);
45     tft.setCursor(66, 124);
46     tft.print("Warte...");
47     sieger = ' ';
48     delay(random(1500, 4000));
49
50     if (sieger != ' ') {
51         Serial.printf("%c war zu frueh!\n", sieger);
52         tft.fillScreen(ST77XX_ORANGE);
53         tft.setCursor(30, 124);
54         tft.printf("%c zu frueh!", sieger);
55     } else {
56         unsigned long signal = millis();
57         tft.fillScreen(ST77XX_GREEN);
58         while (sieger == ' ' && millis() - signal < 3000) {
59             // warten, bis eine ISR den Sieger eintraegt
60         }
61         if (sieger != ' ') {
62             Serial.printf("%c gewinnt nach %lu ms\n", sieger, sieger_zeit -
                signal);
```

```
            tft.fillScreen(ST77XX_BLACK);
            tft.setCursor(48, 124);
            tft.printf("%c gewinnt", sieger);
        }
    }
    delay(2500);
}
```

Die Variable `sieger` ist ein `char`: `' '` heißt „noch niemand“, sonst `'A'` oder `'B'`. Jede ISR trägt sich nur ein, wenn dort noch ein Leerzeichen steht. Damit zählt automatisch nur der allererste Druck, auch wenn die Tasten prellen. Drückt jemand während `delay(random(…))`, steht danach schon ein Name in `sieger`: zu früh.



Serieller Monitor

```
A gewinnt nach 924 ms
B war zu frueh!
```

Aufgabe 4 – Lauflicht auf Kern 1 (`k24_loesung4`)

k24_loesung4.ino

```
// Loesung 24.4: Lauflicht auf Kern 1, Tempo mit dem Joystick auf Kern 0
#define LED_ROT 14
#define LED_GELB 13
#define LED_GRUEN 12
#define TASTE_HOCH 4
#define TASTE_RUNTER 2

volatile int pause_ms = 300;    // Kern 0 schreibt, Kern 1 liest

// Kern 0: Joystick abfragen
void setup() {
    Serial.begin(115200);
    pinMode(TASTE_HOCH, INPUT_PULLUP);
    pinMode(TASTE_RUNTER, INPUT_PULLUP);
}

void loop() {
    if (digitalRead(TASTE_HOCH) == LOW && pause_ms > 50) {
        pause_ms -= 10;
        Serial.printf("Pause: %d ms\n", pause_ms);
    }
    if (digitalRead(TASTE_RUNTER) == LOW && pause_ms < 1000) {
        pause_ms += 10;
        Serial.printf("Pause: %d ms\n", pause_ms);
    }
    delay(50);
}
```

```
29 // Kern 1: das Lauflicht
30 void setup1() {
31     pinMode(LED_ROT, OUTPUT);
32     pinMode(LED_GELB, OUTPUT);
33     pinMode(LED_GRUEN, OUTPUT);
34 }
35
36 void loop1() {
37     const int LEDS[3] = { LED_ROT, LED_GELB, LED_GRUEN };
38     for (int i = 0; i < 3; i++) {
39         digitalWrite(LEDS[i], HIGH);
40         delay(pause_ms);
41         digitalWrite(LEDS[i], LOW);
42     }
43 }
```

Die Geschwindigkeit steckt in der gemeinsamen Variable `pause_ms`. Kern 0 schreibt sie, Kern 1 liest sie vor jedem `delay`; deshalb ist sie `volatile`. Ein `int` schreibt der Chip in einem Schritt, Kern 1 sieht also nie einen halb geschriebenen Wert. Eine FIFO ist hier nicht nötig: Kern 1 braucht immer nur den neuesten Wert, nicht jede einzelne Änderung. Dieser Sketch läuft nur auf dem PicoBoy.

Lösungen zu Kapitel 25: Daten dauerhaft speichern

Alle Sketche dieses Kapitels laufen nur auf dem PicoBoy (Flash Size **16MB (Sketch: 8MB, FS: 8MB)** einstellen).

Aufgabe 1 – Zurück auf null (`k25_loesung1`)

k25_loesung1.ino

```
28     EEPROM.begin(256);
29     EEPROM.get(0, daten);
30     if (daten.kennung != KENNUNG || digitalRead(TASTE_B) == LOW) {
31         daten.kennung = KENNUNG;    // neu anfangen
32         daten.starts = 0;
33     }
34     daten.starts++;
35     EEPROM.put(0, daten);
36     EEPROM.commit();
```

Nur die Bedingung in Zeile 30 ist neu: Neu angefangen wird, wenn die Kennung nicht stimmt **oder** Taste B gedrückt ist. Danach zählt Zeile 34 wie immer hoch, die Anzeige beginnt also bei 1. Damit `digitalRead` die Taste richtig liest, steht vorher `pinMode(TASTE_B, INPUT_PULLUP);` in `setup()`.

Aufgabe 2 – Lieblingsfarbe (`k25_loesung2`)

k25_loesung2.ino

```
// Loesung 25.2: Lieblingsfarbe der RGB-LED waehlen und dauerhaft speichern
#include <Adafruit_NeoPixel.h>
#include <EEPROM.h>

#define RGB_LED 11
#define TASTE_LINKS 3
#define TASTE_RECHTS 1
#define TASTE_A 27
#define KENNUNG 0x46415242      // "FARB"
#define ANZAHL_FARBEN 6

struct Einstellung {
    uint32_t kennung;
    int farbe;
};

const uint32_t FARBEN[ANZAHL_FARBEN] = {
    0xFF0000, 0xFFFF00, 0x00FF00, 0x00FFFF, 0x0000FF, 0xFF00FF
};

Adafruit_NeoPixel rgb(1, RGB_LED, NEO_GRB + NEO_KHZ800);
Einstellung einstellung;

void setup() {
    Serial.begin(115200);
    pinMode(TASTE_LINKS, INPUT_PULLUP);
    pinMode(TASTE_RECHTS, INPUT_PULLUP);
    pinMode(TASTE_A, INPUT_PULLUP);
    rgb.begin();
    rgb.setBrightness(40);

    EEPROM.begin(256);
    EEPROM.get(0, einstellung);
    if (einstellung.kennung != KENNUNG || einstellung.farbe < 0 ||
        einstellung.farbe >= ANZAHL_FARBEN) {
        einstellung.kennung = KENNUNG;
        einstellung.farbe = 0;
    }
}

void loop() {
    if (digitalRead(TASTE_RECHTS) == LOW) {
        einstellung.farbe = (einstellung.farbe + 1) % ANZAHL_FARBEN;
        delay(250);
    }
    if (digitalRead(TASTE_LINKS) == LOW) {
        einstellung.farbe = (einstellung.farbe + ANZAHL_FARBEN - 1) %
          ANZAHL_FARBEN;
        delay(250);
    }
```

```
50      if (digitalRead(TASTE_A) == LOW) {
51          EEPROM.put(0, einstellung);
52          EEPROM.commit();            // schreibt nur, wenn sich etwas geaendert
              hat
53          Serial.printf("Farbe %d gespeichert\n", einstellung.farbe);
54          delay(500);
55      }
56      rgb.setPixelColor(0, FARBEN[einstellung.farbe]);
57      rgb.show();
58      delay(10);
59  }
```

- Die sechs Farben stehen als Zahlen `0xRRGGBB` in einem Array. `setPixelColor` versteht diese Schreibweise direkt.
- Gespeichert wird nur die **Nummer** der Farbe. Beim Laden prüft Zeile 34 zusätzlich, ob die Nummer zwischen 0 und 5 liegt. Sonst würde ein kaputter Wert über das Ende des Arrays hinaus lesen.
- `% ANZAHL_FARBEN` lässt die Auswahl im Kreis laufen. Beim Zurückgehen wird erst `ANZAHL_FARBEN` addiert, damit nie eine negative Zahl entsteht.
- Drückst du A zweimal ohne Änderung, schreibt `commit()` beim zweiten Mal nichts, denn die Kopie im RAM hat sich nicht verändert.

Aufgabe 3 – Batterie-Protokoll (`k25_loesung3`)

k25_loesung3.ino

```
1  // Loesung 25.3: Batteriespannung alle 10 s protokollieren; A zeigt, B loescht
2  #include <LittleFS.h>
3
4  #define BATTERIE A3
5  #define TASTE_A 27
6  #define TASTE_B 28
7  #define DATEI "/batterie.csv"
8
9  unsigned long letzte_messung = 0;
10
11 void setup() {
12     Serial.begin(115200);
13     pinMode(TASTE_A, INPUT_PULLUP);
14     pinMode(TASTE_B, INPUT_PULLUP);
15     analogReadResolution(12);
16     if (!LittleFS.begin()) {
17         Serial.println("Kein Dateisystem - Flash Size mit FS eingestellt?");
18     }
19 }
20
21 void loop() {
22     if (millis() - letzte_messung >= 10000) {
23         letzte_messung = millis();
24         float volt = analogRead(BATTERIE) * 3.3 / 4095 * 3;
25         File datei = LittleFS.open(DATEI, "a");
```

```
        datei.printf("%lu;%.2f\n", millis() / 1000, volt);
        datei.close();
    }
    if (digitalRead(TASTE_A) == LOW) {
        File datei = LittleFS.open(DATEI, "r");
        if (datei) {
            Serial.println("Sekunden;Volt");
            while (datei.available()) {
                Serial.write(datei.read());
            }
            datei.close();
        } else {
            Serial.println("Noch keine Messwerte");
        }
        delay(500);
    }
    if (digitalRead(TASTE_B) == LOW) {
        LittleFS.remove(DATEI);
        Serial.println("Protokoll geloescht");
        delay(500);
    }
}
```

Gemessen wird mit dem `millis`-Takt aus Kapitel 7.4, nicht mit `delay`, damit die Tasten jederzeit reagieren. Jede Messung öffnet die Datei mit `"a"`, hängt eine Zeile an und schließt sie gleich wieder. So geht höchstens die letzte Zeile verloren, wenn die Batterien leer werden. `LittleFS.remove` löscht die Datei; die nächste Messung legt sie mit `"a"` neu an.

Aufgabe 4 – Rekord mit Namen (`k25_loesung4`)

Die Struktur bekommt ein `char`-Array für drei Buchstaben und das Endzeichen `'\0'` (Kapitel 12). Weil sich der Aufbau geändert hat, gibt es eine neue Kennung: Ein alter Rekord ohne Namen wird so nicht falsch gelesen.

k25_loesung4.ino

```
struct Rekord {
    uint32_t kennung;
    int punkte;
    char name[4];               // drei Buchstaben und das Endzeichen
};
```

Die Eingabe des Kürzels:

k25_loesung4.ino

```
// Joystick hoch/runter: Buchstabe, rechts: naechster Buchstabe, A: fertig
void name_eingeben() {
    char name[4] = "AAA";
    int stelle = 0;
    tft.setTextSize(2);
    tft.setCursor(24, 230);
    tft.print("Name? A = fertig");
```

```
    while (digitalRead(TASTE_A) == LOW) {
        delay(10);                  // A ist vom Spiel vielleicht noch gedrueckt
    }
    while (digitalRead(TASTE_A) == HIGH) {
        if (digitalRead(TASTE_HOCH) == LOW) {
            name[stelle] = name[stelle] == 'Z' ? 'A' : name[stelle] + 1;
        }
        if (digitalRead(TASTE_RUNTER) == LOW) {
            name[stelle] = name[stelle] == 'A' ? 'Z' : name[stelle] - 1;
        }
        if (digitalRead(TASTE_RECHTS) == LOW) {
            stelle = (stelle + 1) % 3;
        }
        tft.fillRect(0, 140, 240, 70, ST77XX_BLACK);
        tft.setTextSize(4);
        tft.setTextColor(ST77XX_YELLOW);
        tft.setCursor(84, 150);
        tft.print(name);
        tft.fillRect(84 + stelle * 24, 186, 20, 4, ST77XX_YELLOW);   // Strich
          unter der Stelle
        tft.setTextColor(ST77XX_WHITE);
        delay(200);
    }
    strcpy(rekord.name, name);
}
```

- Zuerst wartet die Funktion, bis A losgelassen ist. Sonst würde der letzte Druck aus dem Spiel die Eingabe sofort beenden.
- Hoch und runter zählen den Buchstaben weiter; nach `'Z'` kommt wieder `'A'`. Mit Buchstaben kann man rechnen, denn ein `char` ist eine Zahl.
- `strcpy` kopiert den fertigen Namen in die Struktur, danach speichert `rekord_speichern()` alles zusammen.
- Beim Laden setzt der Sketch `rekord.name[3] = '\0'`. Selbst wenn im Flash Unsinn stünde, endet der Text so spätestens nach drei Zeichen.

ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 26: Dein eigenes Projekt

Aufgabe 1 – Ein Plan für die Stoppuhr

So könnten die Antworten aussehen:

1. **Was soll der Sketch tun?** Er misst Zeiten in Zehntelsekunden, wie eine Stoppuhr beim Sport.
2. **Wie bedient man ihn?** A startet und stoppt, B setzt auf 0. Das Display zeigt die Zeit groß in der Mitte, darunter klein die Bedienung.
3. **Kleinste Fassung:** A startet, A stoppt, die Zeit steht im seriellen Monitor.

Stufe	Was kommt dazu?	Fertig, wenn …
1	A startet und stoppt, gestoppte Zeit im seriellen Monitor	zwei Drücke im Abstand von etwa 3 s ungefähr `3.0` ergeben
2	laufende Zeit auf dem Display, flimmerfrei mit dem Bildpuffer	die Zehntel gleichmäßig weiterzählen
3	B setzt auf 0; nach erneutem Start zählt die Uhr weiter	Start, Stopp, Start addiert beide Zeiten
4 (Extra)	Farben: grün, solange die Uhr läuft; Rundenzeiten	

Aufgabe 2 – Fehlersuche (`k26_loesung2`)

Eine Ausgabe in der Schleife, etwa `Serial.printf("i: %d temperatur: %d warm: %d\n", i, temperatur[i], warm);`, zeigt: Die erste Zeile beginnt mit `i: 1`. Der erste Tag (`temperatur[0]`, 22 Grad) wird nie angesehen. Arrays beginnen bei 0, die Schleife muss also mit `int i = 0` anfangen:

k26_loesung2.ino

```
 1 // Loesung 26.2: Die Zaehlschleife muss bei 0 beginnen
 2 int temperatur[7] = { 22, 18, 19, 24, 21, 17, 20 };
 3
 4 void setup() {
 5     Serial.begin(115200);
 6     int warm = 0;
 7     for (int i = 0; i < 7; i++) {
 8         if (temperatur[i] > 20) {
 9             warm++;
10         }
11     }
12     Serial.printf("Warme Tage: %d\n", warm);
13 }
14
15 void loop() {
16 }
```

Serieller Monitor

```
Warme Tage: 3
```

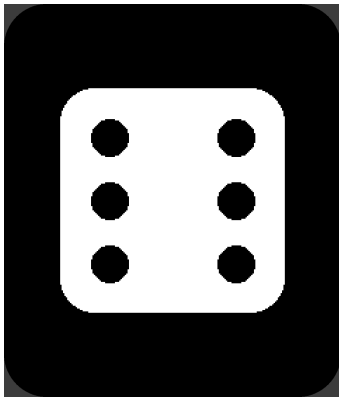
Aufgabe 3 – Würfelaugen (`k26_loesung3`)

k26_loesung3.ino

```
37 void auge(int dx, int dy) {
38     tft.fillCircle(MITTE_X + dx * ABSTAND, MITTE_Y + dy * ABSTAND, 13,
         ST77XX_BLACK);
39 }
40
41 void zeichne_wuerfel(int zahl) {
42     tft.fillRoundRect(MITTE_X - 80, MITTE_Y - 80, 160, 160, 24, ST77XX_WHITE);
```

```
    if (zahl % 2 == 1) {        // 1, 3, 5: Auge in der Mitte
        auge(0, 0);
    }
    if (zahl >= 2) {            // links oben und rechts unten
        auge(-1, -1);
        auge(1, 1);
    }
    if (zahl >= 4) {            // rechts oben und links unten
        auge(1, -1);
        auge(-1, 1);
    }
    if (zahl == 6) {            // Mitte links und Mitte rechts
        auge(-1, 0);
        auge(1, 0);
    }
}
```

`auge(dx, dy)` zeichnet ein Auge an einer von neun Stellen: `dx` und `dy` sind −1, 0 oder 1 und werden mit `ABSTAND` (45 Pixel) malgenommen. So steht jede Würfelseite in wenigen Zeilen. `zahl % 2 == 1` ist wahr für ungerade Zahlen. Beim Start zeigt der Sketch die Sechs, dann würfelt A. Zum Testen aller Seiten rufst du `zeichne_wuerfel` in `setup()` nacheinander mit 1 bis 6 auf, je mit `delay(1000)`.



Aufgabe 4 – Die Stoppuhr bauen (`k26_loesung4`)

k26_loesung4.ino

```
void loop() {
    bool a_jetzt = digitalRead(TASTE_A) == LOW;
    if (a_jetzt && !a_vorher) {
        if (laeuft) {
            gemessen += millis() - start;
            Serial.printf("Gestoppt: %.1f s\n", gemessen / 1000.0);
        } else {
            start = millis();
        }
        laeuft = !laeuft;
    }
    a_vorher = a_jetzt;
    if (digitalRead(TASTE_B) == LOW && !laeuft) {
        gemessen = 0;
    }

    unsigned long zeit = gemessen;
    if (laeuft) {
        zeit += millis() - start;
    }
```

```
    bild.fillScreen(ST77XX_BLACK);
    bild.setTextColor(laeuft ? ST77XX_GREEN : ST77XX_WHITE);
    bild.setTextSize(5);
    bild.setCursor(45, 115);
    bild.printf("%5.1f", zeit / 1000.0);
    bild.setTextSize(2);
    bild.setTextColor(ST77XX_WHITE);
    bild.setCursor(42, 200);
    bild.print("A Start/Stopp");
    bild.setCursor(78, 225);
    bild.print("B auf 0");
    tft.drawRGBBitmap(0, 0, bild.getBuffer(), 240, 280);
    delay(20);
}
```

- `gemessen` enthält die Zeit aller abgeschlossenen Messungen, `start` den Zeitpunkt des letzten Starts. Die angezeigte Zeit ist `gemessen` plus, falls die Uhr läuft, die Zeit seit `start`. So addieren sich mehrere Start-Stopp-Abschnitte von selbst.
- `%5.1f` schreibt die Zahl mit einer Nachkommastelle, immer 5 Zeichen breit. Die Anzeige springt dadurch nicht hin und her.

Serieller Monitor
```
Gestoppt: 2.7 s
```



ANHANG C – LÖSUNGEN

Lösungen zu Kapitel 27: Programmieren mit KI

Aufgabe 1 – KI-Fehler finden (`k27_loesung1`)

Die Fehler in `k27_aufgabe1`:

1. **`Serial1`** (Zeile 14 und 28): liegt auf GP0/GP1, also auf Joystick Mitte und rechts, und kommt nicht im seriellen Monitor an. Richtig ist `Serial`.
2. **Kein Hintergrundlicht:** `TFT_LICHT` fehlt ganz. Das Display bleibt schwarz.
3. **`LED_BUILTIN`** (Zeile 16 und 29) ist GP25; dort hängt keine der LEDs.
4. **`Wire.begin()` und `sensor.begin()` ohne Pins** (Zeile 17 und 18): Der Sensor hängt an GP20/GP21. `Wire.begin()` ist zudem überflüssig, das erledigt `sensor.begin(20, 21)`.
5. **`setOrientation` fehlt:** Ohne sie stimmen die Achsen nicht mit dem PicoBoy überein (Kapitel 17).

k27_loesung1.ino
```
// Loesung 27.1: Neigung anzeigen und dabei blinken - fuer den PicoBoy angepasst
#include <Adafruit_ST7789.h>
#include <STK8BA58.h>
```

```
#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26
#define LED_GRUEN 12

Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);
STK8BA58 sensor;
bool led_an = false;

void setup() {
    Serial.begin(115200);              // USB statt Serial1 (GP0/GP1 = Joystick)
    tft.init(240, 280);
    pinMode(TFT_LICHT, OUTPUT);        // ohne Hintergrundlicht bleibt alles
      schwarz
    digitalWrite(TFT_LICHT, HIGH);
    pinMode(LED_GRUEN, OUTPUT);        // LED_BUILTIN (GP25) ist keine der LEDs
    if (!sensor.begin(20, 21)) {       // der Sensor haengt an GP20/GP21
        Serial.println("Neigungssensor antwortet nicht");
    }
    sensor.setOrientation(STK8BA58::PICOBOY_COLOR_PLUS);
}

void loop() {
    float x, y, z;
    sensor.readAcceleration(x, y, z);
    tft.fillScreen(ST77XX_BLACK);
    tft.setTextColor(ST77XX_WHITE);
    tft.setTextSize(3);
    tft.setCursor(48, 124);
    tft.printf("x %.2f", x);
    Serial.printf("x %.2f\n", x);
    led_an = !led_an;
    digitalWrite(LED_GRUEN, led_an);
    delay(500);
}
```

Im Prüflauf ist der PicoBoy um 0,25 g nach rechts geneigt:

Serieller Monitor

```
x 0.25
x 0.25
```



Aufgabe 2 – Ein guter Prompt

Ein möglicher Prompt (am Anfang steht der Inhalt von `AGENTS.md`; ohne ihn gehören mindestens Board und Pins hinein):

```
Schreibe einen Arduino-Sketch für den PicoBoy Color Plus (Regeln oben).
Die drei LEDs (rot 14, gelb 13, grün 12) sollen eine Autoampel nachmachen:
rot 3 s, rot-gelb 1 s, grün 3 s, gelb 1 s, dann wieder rot.
Wenn jemand Taste A (Pin 27, INPUT_PULLUP, gedrückt = LOW) drückt, wird
die Ampel nach der nächsten Grünphase rot und bleibt 5 s rot. In diesen
5 s piept der Lautsprecher (Pin 15) einmal pro Sekunde mit tone().
Warte nicht mit delay(), damit kein Tastendruck verloren geht; nimm
millis(). Kommentare auf Deutsch. Erkläre den Ablauf kurz.
```

Wichtig sind: das Gerät, die Pins, wie die Taste angeschlossen ist, der genaue Ablauf mit Zeiten, und Wünsche zur Bauweise (hier: `millis` statt `delay`).

Aufgabe 3 – Erfunden? (`k27_loesung3`)

Beim Überprüfen meldet der Compiler:

Ausgabe

```
k27_aufgabe3.ino: In function 'void setup()':
k27_aufgabe3.ino:10:9: error: 'class Adafruit_NeoPixel' has no member named
  'setColor'; did you mean 'Color'?
   10 |     rgb.setColor(255, 0, 0);
      |         ^~~~~~~~
      |         Color
Fehler beim Build: exit status 1
```

`setColor` gibt es in Adafruit NeoPixel nicht. Die Farbe setzt `setPixelColor(nummer, rot, gruen, blau)`, und erst `show()` schickt sie zur LED:

k27_loesung3.ino

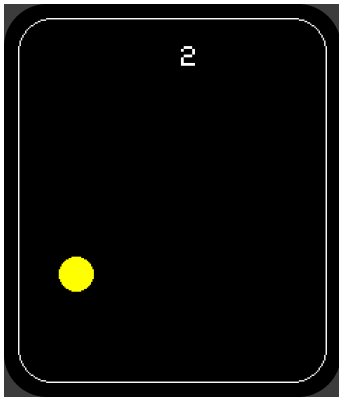
```
 1 // Loesung 27.3: Die RGB-LED leuchtet rot
 2 #include <Adafruit_NeoPixel.h>
 3
 4 #define RGB_LED 11
 5
 6 Adafruit_NeoPixel rgb(1, RGB_LED, NEO_GRB + NEO_KHZ800);
 7
 8 void setup() {
 9     rgb.begin();
10     rgb.setBrightness(40);
11     rgb.setPixelColor(0, 255, 0, 0);    // LED Nummer 0, Rot, Gruen, Blau
12     rgb.show();                         // erst show() schickt die Farbe zur LED
13 }
14
15 void loop() {
16 }
```

Aufgabe 4 – Weiter mit KI (`k27_loesung4`)

Eine mögliche Lösung: Die vier Stellen, an denen die Kugel abprallt, rufen jetzt eine gemeinsame Funktion auf. Sie bekommt die Geschwindigkeit als **Referenz** (`float &v`, Kapitel 19) und kann sie deshalb direkt umdrehen.

k27_loesung4.ino

```
82 // kehrt die Geschwindigkeit um; ein kraeftiger Aufprall klingt und zaehlt
83 void abprallen(float &v) {
84     if (abs(v) > 1.5) {
85         treffer++;
86         tone(LAUTSPRECHER, 200 + (int)(abs(v) * 60), 40);
87     }
88     v = -v * 0.6;
89 }
```



- Nur ein Aufprall mit mehr als 1,5 Pixeln pro Bild zählt. Sonst würde die Kugel, die ruhig am Rand liegt, in jedem Bild einen Treffer zählen und dauernd brummen. Achte bei einer KI-Lösung auf diesen Fehler; man sieht ihn erst am Gerät oder beim genauen Lesen.
- Die Tonhöhe wächst mit der Geschwindigkeit, `tone` wartet nicht.
- Der Zähler steht oben in der Mitte, mit Abstand zu den runden Ecken.

Zu deinen Notizen: Typisch sind fehlende Kennzeichnung im Kopfkommentar, `delay` nach dem Ton (das Spiel ruckelt) oder ein Zähler, der bei jedem Bild weiterläuft.

ANHANG D

Hardware-Referenz: Pinbelegung des PBC+

Alles Wichtige über die Hardware des PicoBoy Color Plus auf einen Blick: die Einstellungen in der Arduino IDE, die Pins mit den Namen aus diesem Buch, die Bauteile, die Aufteilung des Flash und die Fallen, in die man mit den Voreinstellungen des Kerns tappt.

D.1 Board-Einstellungen

Menü **Werkzeuge**	Einstellung
Board	Raspberry Pi Pico/RP2040/RP2350 → **Generic RP2350**
Flash Size	**16MB (Sketch: 8MB, FS: 8MB)** (Voreinstellung wäre 2MB (no FS))
CPU Speed	150 MHz (Voreinstellung)
Upload Method	Default (UF2) (Voreinstellung)
USB Stack	Pico SDK (Voreinstellung)
alle anderen	Voreinstellung lassen

Kern: „Raspberry Pi Pico/RP2040/RP2350“ von Earle F. Philhower, III, Version 6.x (geprüft mit 6.3.0). Boardverwalter-URL: `https://github.com/earlephilhower/arduino-pico/releases/download/global/package_rp2040_index.json`

Für `arduino-cli` (Anhang E) lautet das Board mit Flash-Einstellung (**FQBN**, *fully qualified board name*): `rp2040:rp2040:generic_rp2350:flash=16777216_8388608`

Bibliotheken: **Adafruit ST7735 and ST7789** (mit Adafruit GFX und Adafruit BusIO), **Adafruit NeoPixel**, **STK8BA58** (ZIP zum Buch).

D.2 Die Pins

Die Nummern sind die GPIO-Nummern des RP2350 (GP0, GP1 …). Im Sketch schreibst du nur die Zahl, besser aber den Namen aus diesem Buch.

Pin	Bauteil	Name im Sketch	Richtung	Hinweis
0	Joystick Mitte	`TASTE_MITTE`	Eingang	`INPUT_PULLUP`, gedrückt `LOW`
1	Joystick rechts	`TASTE_RECHTS`	Eingang	`INPUT_PULLUP`, gedrückt `LOW`
2	Joystick runter	`TASTE_RUNTER`	Eingang	`INPUT_PULLUP`, gedrückt `LOW`
3	Joystick links	`TASTE_LINKS`	Eingang	`INPUT_PULLUP`, gedrückt `LOW`

Pin	Bauteil	Name im Sketch	Richtung	Hinweis
4	Joystick hoch	`TASTE_HOCH`	Eingang	`INPUT_PULLUP`, gedrückt `LOW`
8	Display DC	`TFT_DC`	Ausgang	Befehl oder Daten
9	Display Reset	`TFT_RST`	Ausgang	
10	Display CS	`TFT_CS`	Ausgang	Chip Select
11	RGB-LED	`RGB_LED`	Ausgang	Adafruit NeoPixel, `NEO_GRB + NEO_KHZ800`
12	LED grün	`LED_GRUEN`	Ausgang	`HIGH` = an
13	LED gelb	`LED_GELB`	Ausgang	`HIGH` = an
14	LED rot	`LED_ROT`	Ausgang	`HIGH` = an
15	Lautsprecher	`LAUTSPRECHER`	Ausgang	`tone`, PWM
18	Display SCK	–	Ausgang	SPI-Takt, belegt die Bibliothek
19	Display MOSI	–	Ausgang	SPI-Daten, belegt die Bibliothek
20	Neigungssensor SDA	–	I2C	`sensor.begin(20, 21)`
21	Neigungssensor SCL	–	I2C	Adresse des Sensors: `0x18`
26	Hintergrundlicht	`TFT_LICHT`	Ausgang	`HIGH` oder `analogWrite` zum Dimmen
27	Taste A	`TASTE_A`	Eingang	`INPUT_PULLUP`, gedrückt `LOW`
28	Taste B	`TASTE_B`	Eingang	`INPUT_PULLUP`, gedrückt `LOW`
29 (`A3`)	Batteriespannung	`BATTERIE`	analog	Spannungsteiler 1:3

Zum Abschreiben, so steht es in den Sketchen dieses Buchs:

```
#define TFT_CS 10
#define TFT_DC 8
#define TFT_RST 9
#define TFT_LICHT 26
#define LAUTSPRECHER 15
#define LED_ROT 14
#define LED_GELB 13
#define LED_GRUEN 12
#define RGB_LED 11
#define TASTE_HOCH 4
#define TASTE_RUNTER 2
#define TASTE_LINKS 3
#define TASTE_RECHTS 1
```

```
#define TASTE_MITTE 0
#define TASTE_A 27
#define TASTE_B 28
#define BATTERIE A3
```

Der Sketch **`vorlage/Selbsttest`** bei den Sketchen zum Buch probiert alle Bauteile auf einmal aus: Tasten, LEDs, RGB-LED, Töne, Neigung und Batteriespannung.

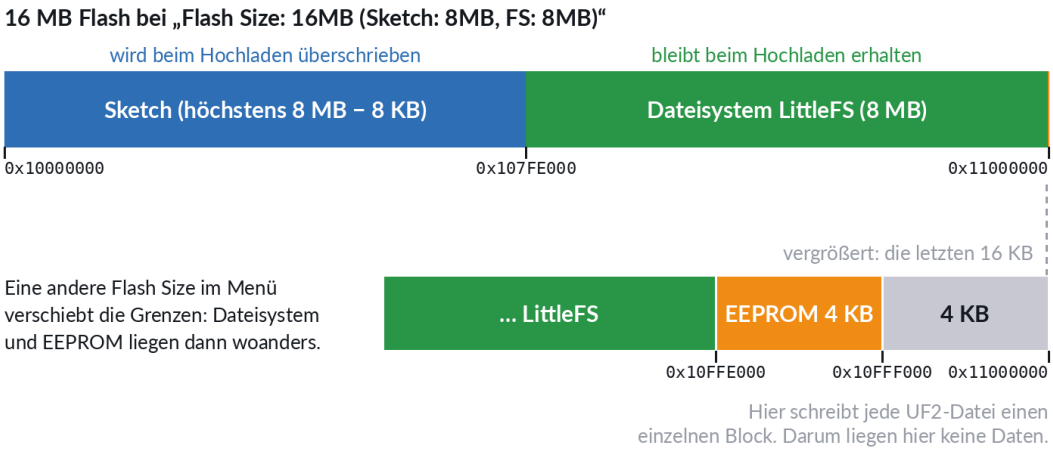
D.3 Die Bauteile

Bauteil	Daten	Im Sketch
Mikrocontroller	RP2350, 2 Kerne (Arm Cortex-M33), 150 MHz, 520 KB RAM	Kapitel 1, 19, 24
Flash	16 MB	Kapitel 19, 25
Display	ST7789, 240 × 280 Pixel, Farbe in 16 Bit (RGB565), runde Ecken: etwa 20 Pixel Rand freilassen	`Adafruit_ST7789 tft(TFT_CS, TFT_DC, TFT_RST);` `tft.init(240, 280);`
Hintergrundlicht	an Pin 26; ohne Licht bleibt das Display schwarz	`digitalWrite(TFT_LICHT, HIGH);`
Joystick, A, B	sieben Tasten, kein Entprellen eingebaut	`digitalRead`, Kapitel 9
LEDs	rot, gelb, grün	`digitalWrite`, `analogWrite`, Kapitel 10
RGB-LED	eine LED für Adafruit NeoPixel	`Adafruit_NeoPixel rgb(1, RGB_LED, NEO_GRB + NEO_KHZ800);`
Lautsprecher	kleiner Lautsprecher hinter einem Tiefpass	`tone(LAUTSPRECHER, frequenz, dauer)` wartet nicht
Neigungssensor	STK8BA58, 3 Achsen, I2C-Adresse `0x18`	Bibliothek STK8BA58, Kapitel 17 und 22
Batterien	zwei AAA-Batterien in Reihe; Spannung über Teiler 1:3 an `A3`	`analogReadResolution(12);` `volt = analogRead(BATTERIE) * 3.3 / 4095 * 3;`
USB-C	Hochladen, serieller Monitor (`Serial`), Stromversorgung	Kapitel 2 und 4
Schalter BAT/USB	Rückseite unten: Strom von den Batterien (BAT) oder vom Kabel (USB); auf USB ohne Kabel ist das Gerät aus	–
BOOT, RES	Rückseite: BOOT halten, RES tippen = Bootmodus	Kapitel 1.3

Am USB-Kabel misst `BATTERIE` etwa 5 V. Mit zwei frischen Alkali-Batterien sind es etwa 3,0 bis 3,2 V, fast leer etwa 2,0 bis 2,2 V. Wiederaufladbare AAA-Akkus (NiMH) liefern nur etwa 1,2 V pro Zelle, zusammen etwa 2,4 bis 2,7 V.

D.4 Die Aufteilung des Flash

Mit **Flash Size 16MB (Sketch: 8MB, FS: 8MB)** teilt der Kern die 16 MB so auf (Kapitel 25):



Bereich	Adressen	Größe
Sketch	`0x10000000` bis `0x107FE000`	8 MB – 8 KB
Dateisystem LittleFS	`0x107FE000` bis `0x10FFE000`	8 MB
EEPROM-Nachbildung	`0x10FFE000` bis `0x10FFF000`	4 KB
frei (die UF2-Datei schreibt hier einen Block)	`0x10FFF000` bis `0x11000000`	4 KB

Beim Hochladen werden nur der Sketch-Bereich und der eine Block am Ende geschrieben. Dateisystem und EEPROM bleiben erhalten.

D.5 Fallen

Der Kern kennt den PicoBoy nicht. Mit „Generic RP2350“ nimmt er die Belegung des Raspberry Pi Pico 2 an. Einige dieser **Standard-Pins** sind auf dem PicoBoy schon belegt:

Was	Standard-Pins des Kerns	Auf dem PicoBoy	Was tun?
`Wire` (I2C)	GP4 (SDA), GP5 (SCL)	GP4 ist Joystick hoch	`Wire.setSDA(20); Wire.setSCL(21);` vor `Wire.begin()`, oder `sensor.begin(20, 21)`
`Wire1`	GP26, GP27	Hintergrundlicht und Taste A	nicht benutzen

Was	Standard-Pins des Kerns	Auf dem PicoBoy	Was tun?
`Serial1` (UART)	GP0 (TX), GP1 (RX)	Joystick Mitte und rechts	nicht benutzen; `Serial` geht über USB
`Serial2`	GP8, GP9	Display DC und Reset	nicht benutzen
`SPI`	GP18, GP19 (dazu GP16, GP17)	Display	gehört der Display-Bibliothek
`SPI1`	GP12 bis GP15	LEDs und Lautsprecher	nicht benutzen
`LED_BUILTIN`	GP25	keine der LEDs	`LED_ROT`, `LED_GELB`, `LED_GRUEN`
`A0`, `A1`, `A2`	GP26, GP27, GP28	Hintergrundlicht, Taste A, Taste B	nur `A3` (Batterie) ist ein Messeingang

Weitere Fallen:

- **Hintergrundlicht vergessen:** Ohne `digitalWrite(TFT_LICHT, HIGH)` bleibt das Display schwarz, obwohl alles richtig gezeichnet wird.
- **Flash Size:** Mit der Voreinstellung „2MB (no FS)“ gibt es kein Dateisystem (`LittleFS.begin()` liefert `false`). Eine andere Flash Size verschiebt Dateisystem und EEPROM; gespeicherte Daten sind dann nicht mehr zu finden.
- **SPI-Takt:** Ohne `setSPISpeed` läuft die Verbindung zum Display mit real 25 MHz; ein ganzes Bild dauert etwa 43 ms. Mit `tft.setSPISpeed(62500000)` sind es real 37,5 MHz und etwa 29 ms (Kapitel 14 und 23; gerechnet, am Gerät gemessen sind es etwa 47 und 32 ms). Der Chip teilt seinen Takt nur in bestimmten Stufen, deshalb kommt nicht genau der gewünschte Wert heraus.
- **Tasten prellen:** Es gibt kein eingebautes Entprellen (Kapitel 9).
- **`random()` ohne `randomSeed`** liefert bei jedem Start dieselbe Folge. Echter Zufall: `randomSeed(rp2040.hwrand32());`.
- **AVR-Code** aus dem Netz (`PORTB`, `DDRB`, `_delay_ms`) gibt es auf dem RP2350 nicht (Kapitel 27).
- **Windows-Benutzername mit Umlauten** stört den Compiler (Anhang E).

ANHANG E

Installation im Detail (Windows, macOS, Linux), arduino-cli

Kapitel 2 zeigt den kurzen Weg zur fertigen Arduino IDE. Hier stehen die Einzelheiten für jedes Betriebssystem, die Lösungen für typische Probleme und der Weg ohne IDE: mit dem Kommandozeilen-Werkzeug **arduino-cli**, das auch KI-Agenten benutzen (Kapitel 27).

E.1 Arduino IDE 2

Die Arduino IDE gibt es kostenlos auf **arduino.cc** im Bereich *Software*. Nimm immer die Fassung **2.x**. Danach folgen für alle Systeme dieselben Schritte wie in Kapitel 2: Boardverwalter-URL eintragen, Kern installieren, Board und Flash Size einstellen, Bibliotheken installieren.

Windows

- Lade das **Installationsprogramm** (oder die ZIP-Datei) direkt von arduino.cc. **Nicht** die Fassung aus dem Microsoft Store: Laut der Beschreibung des Kerns erkennt sie die Platinen mit RP2040/RP2350 nicht zuverlässig.
- Bietet das Installationsprogramm an, **Treiber** zu installieren, stimme zu.
- **Benutzername mit Umlauten oder Akzenten** (etwa `C:\Users\Jürgen`): Der Compiler des Kerns kommt mit solchen Zeichen im Pfad nicht zurecht. Das Kompilieren bricht dann mit einer Meldung wie `cannot open linker script file` ab. Abhilfe laut Beschreibung des Kerns: ein zusätzliches Windows-Benutzerkonto mit einem Namen ohne Sonderzeichen anlegen und darin arbeiten.
- Der PicoBoy erscheint als serieller Port `COM` mit einer Nummer, im Bootmodus als Laufwerk.
- Ein Virenscanner kann während der Installation des Kerns Dateien sperren. Meldet die Board-Verwaltung „Zugriff verweigert“, starte den Rechner neu und installiere den Kern noch einmal.

macOS

- Lade die Datei für macOS (`.dmg`) und ziehe die **Arduino IDE** in den Ordner **Programme**. Achte auf die passende Fassung für deinen Mac (Apple-Chip oder Intel), falls arduino.cc beide anbietet.
- Beim ersten Start fragt macOS nach, ob du ein Programm aus dem Internet öffnen willst. Bestätige.
- Der PicoBoy erscheint als Port `/dev/cu.usbmodem…`, im Bootmodus als Laufwerk auf dem Schreibtisch.

Linux

- Von arduino.cc gibt es eine **AppImage**-Datei. Mach sie ausführbar und starte sie:

  ```
  chmod +x arduino-ide_*.AppImage
  ./arduino-ide_*.AppImage
  ```

  (Der genaue Dateiname hängt von der Version ab.)

- **Flatpak:** Hast du die IDE aus der Software-Verwaltung deiner Distribution installiert (bei Mint, Fedora oder Pop!_OS oft als Flatpak), darf sie nicht auf das Laufwerk zugreifen, das beim Hochladen erscheint. Das Hochladen endet dann mit `No drive to deploy.` Abhilfe aus der Beschreibung des Kerns, danach die IDE neu starten:

  ```
  flatpak override --user --filesystem=host cc.arduino.IDE2
  ```

- **Zugriff auf den seriellen Port:** Der PicoBoy erscheint als `/dev/ttyACM0` (oder mit höherer Nummer). Auf vielen Distributionen dürfen nur Mitglieder einer bestimmten Gruppe darauf zugreifen. Welche, zeigt `ls -l /dev/ttyACM0` (bei Debian und Ubuntu meist `dialout`). So trägst du dich ein; danach einmal ab- und wieder anmelden:

  ```
  sudo usermod -aG dialout $USER
  ```

- **Das Laufwerk im Bootmodus** muss eingehängt werden. Das Hochladeprogramm des Kerns versucht das selbst mit `udisksctl` und sucht außerdem unter `/media/<benutzer>` und `/run/media/<benutzer>`. Auf einem gewöhnlichen Desktop klappt das von selbst. Ohne Desktop hängst du das Laufwerk von Hand ein, bevor du hochlädst.

Braucht man udev-Regeln? Für das gewöhnliche Hochladen (Upload Method **Default (UF2)**) nicht: Der Kern schaltet den PicoBoy über den seriellen Port in den Bootmodus und kopiert die UF2-Datei auf das Laufwerk. Nur wer unter **Werkzeuge → Upload Method** auf **Picotool** umstellt, braucht Regeln, damit `picotool` ohne Administratorrechte auf das Gerät zugreifen darf. Die Beschreibung des Kerns nennt dafür diese Befehle:

```
echo 'SUBSYSTEM=="usb", ATTRS{idVendor}=="2e8a", ATTRS{idProduct}=="0003",
  MODE="660", GROUP="plugdev"' | sudo tee -a /etc/udev/rules.d/98-Picotool.rules
echo 'SUBSYSTEM=="usb", ATTRS{idVendor}=="2e8a", ATTRS{idProduct}=="000a",
  MODE="660", GROUP="plugdev"' | sudo tee -a /etc/udev/rules.d/98-Picotool.rules
echo 'SUBSYSTEM=="usb", ATTRS{idVendor}=="2e8a", ATTRS{idProduct}=="f00a",
  MODE="660", GROUP="plugdev"' | sudo tee -a /etc/udev/rules.d/98-Picotool.rules
echo 'SUBSYSTEM=="usb", ATTRS{idVendor}=="2e8a", ATTRS{idProduct}=="000f",
  MODE="660", GROUP="plugdev"' | sudo tee -a /etc/udev/rules.d/98-Picotool.rules
echo 'SUBSYSTEM=="usb", ATTRS{idVendor}=="2e8a", ATTRS{idProduct}=="f00f",
  MODE="660", GROUP="plugdev"' | sudo tee -a /etc/udev/rules.d/98-Picotool.rules
sudo udevadm control --reload
```

Laut Beschreibung muss `plugdev` auf manchen Distributionen `dialout` heißen. Danach das Kabel einmal ab- und wieder anstecken. Bei der Voreinstellung **Default (UF2)** kannst du dir das sparen.

E.2 Wenn das Hochladen nicht klappt

Meldung oder Beobachtung	Ursache und Abhilfe
`No drive to deploy.`	Der PicoBoy ist nicht im Bootmodus, oder das Laufwerk ist nicht erreichbar (Linux: Flatpak, nicht eingehängt). BOOT halten, RES tippen, Port **UF2 Board** wählen, noch einmal hochladen.
Kein Port unter **Werkzeuge → Port**	Ladekabel ohne Datenleitungen? Anderes Kabel, anderen USB-Anschluss probieren. Unter Windows nicht die Store-Fassung. Unter Linux: Gruppe für den seriellen Port (E.1).
Hochladen meldet, der Port sei belegt	Andere Programme, die den Port benutzen, schließen (ein zweites IDE-Fenster, ein Terminalprogramm).
Nach dem Hochladen reagiert der PicoBoy nicht mehr auf **Hochladen**	Der Sketch ist abgestürzt oder hat USB lahmgelegt. Bootmodus (BOOT halten, RES tippen), dann einen funktionierenden Sketch hochladen.
`cannot open linker script file` (Windows)	Benutzername mit Sonderzeichen, siehe E.1.

E.3 arduino-cli: Arduino ohne IDE

arduino-cli ist das Kommandozeilen-Werkzeug von Arduino. Die IDE benutzt im Hintergrund dieselben Bausteine. Mit arduino-cli kompilierst du und lädst hoch, ohne ein Fenster zu öffnen; das ist praktisch für Skripte und für KI-Agenten, die ihre Sketche selbst überprüfen sollen (Kapitel 27). Wie du arduino-cli für dein System installierst, steht in der offiziellen Anleitung unter **arduino.github.io/arduino-cli** (Abschnitt *Installation*). Die Befehle unten sind mit arduino-cli 1.5 geprüft.

Einrichten (einmal):

```
arduino-cli config init
arduino-cli config add board_manager.additional_urls
  https://github.com/earlephilhower/arduino-
  pico/releases/download/global/package_rp2040_index.json
arduino-cli core update-index
arduino-cli core install rp2040:rp2040
```

`config init` legt eine Konfigurationsdatei `arduino-cli.yaml` an, `config add` trägt die Boardverwalter-URL ein (wie in Kapitel 2.2), und `core install rp2040:rp2040` installiert den Kern. Die Liste aller Boards des Kerns zeigt `arduino-cli board listall rp2040`.

Bibliotheken:

```
arduino-cli lib install "Adafruit ST7735 and ST7789 Library" "Adafruit NeoPixel"
arduino-cli config set library.enable_unsafe_install true
arduino-cli lib install --zip-path STK8BA58.zip
```

Die ersten beiden kommen aus dem Bibliotheksverzeichnis von Arduino, samt Abhängigkeiten wie Adafruit GFX. Eine Bibliothek aus einer ZIP-Datei installiert arduino-cli erst, wenn du das mit der zweiten Zeile erlaubt hast; ohne sie meldet es, dass `--zip-path` abgeschaltet ist.

Kompilieren:

```
arduino-cli compile -b rp2040:rp2040:generic_rp2350:flash=16777216_8388608 --warnings
  all mein_sketch
```

- `-b` nennt das Board samt Flash Size (die **FQBN**, Anhang D). Der Teil `flash=16777216_8388608` ist die Einstellung **16MB (Sketch: 8MB, FS: 8MB)**: 16 777 216 Byte Flash, davon 8 388 608 Byte Dateisystem.
- `--warnings all` entspricht **Compiler-Meldungen: Alle** in der IDE.
- `mein_sketch` ist der **Sketch-Ordner** (mit `mein_sketch.ino` darin).
- Mit `-e` legt arduino-cli die fertigen Dateien, darunter die `.uf2`-Datei, in einem Unterordner von `build` im Sketch-Ordner ab. Die kannst du im Bootmodus auch von Hand auf das Laufwerk des PicoBoy kopieren.

Hochladen und serieller Monitor:

```
arduino-cli board list
arduino-cli upload -b rp2040:rp2040:generic_rp2350:flash=16777216_8388608 -p
  /dev/ttyACM0 mein_sketch
arduino-cli monitor -p /dev/ttyACM0 -c baudrate=115200
```

`board list` zeigt die angeschlossenen Platinen und ihre Ports. Unter Windows heißt der Port `COM3` oder ähnlich, unter macOS `/dev/cu.usbmodem…`. `upload` kompiliert **nicht**; kompiliere also vorher. `monitor` ist der serielle Monitor im Terminal; beenden mit **Strg + C**. Ist der PicoBoy im Bootmodus (beim allerersten Mal), kopiere die `.uf2`-Datei von Hand auf das Laufwerk.

> **Eine eigene Konfiguration.** Willst du arduino-cli neben der IDE benutzen, ohne dass beide sich in die Quere kommen, gib ihm eine eigene Konfigurationsdatei mit eigenen Ordnern. So sind die Sketche dieses Buchs geprüft worden. In der Datei stehen die Ordner für Kern, Downloads und Sketchbook sowie die Boardverwalter-URL:
>
> ```
> directories:
>   data: /home/ich/.cache/pbc-arduino/data
>   downloads: /home/ich/.cache/pbc-arduino/staging
> ```

```
  user: /home/ich/.cache/pbc-arduino/sketchbook
board_manager:
  additional_urls:
    - https://github.com/earlephilhower/arduino-
      pico/releases/download/global/package_rp2040_index.json
```

Jeder Befehl bekommt dann `--config-file` mit dem Pfad zu dieser Datei, zum Beispiel `arduino-cli --config-file ~/.cache/pbc-arduino/arduino-cli.yaml core install rp2040:rp2040`.