

# Anwendungsorientierte Softwareentwicklung

## Laborblatt 1: Blinkende LED

### Lernziele

- **Arbeiten im Terminal unter Unix:** grundlegende Befehle wie `cat`, `ls`, `cd`, `pwd`, `mkdir` verwenden; verstehen, wie das Unix-Dateisystem hierarchisch aufgebaut ist und wie man sich darin bewegt; einfache Textdateien mit einem Editor wie `nano` erstellen und bearbeiten.
- **Den Mikrocontroller direkt programmieren:** ein einfaches Programm schreiben, das eine LED an einem bestimmten Pin (z. B. Pin 2) blinken lässt – zunächst direkt in Maschinencode, um ein Gefühl dafür zu bekommen, wie der Mikrocontroller Programme "versteht".
- **Messung mit dem Oszilloskop:** verstehen, warum die LED bei sehr schnellen Programmen für das Auge nicht mehr sichtbar blinkt, und wie man mit einem Oszilloskop das tatsächliche Verhalten des Programms überprüfen kann.
- **Verstehen der Toolchain zur Mikrocontroller-Programmierung:** nachvollziehen, welche Werkzeuge eingesetzt werden, um ein Programm auf den Mikrocontroller zu übertragen – z. B. wie mit `avrdude`, `avr-ld` und `avr-objcopy` aus Assemblercode Maschinencode erzeugt wird, und wie man diesen mit `avrdude` auf den Mikrocontroller (z. B. Adafruit Metro mit ATmega328P) flasht. Ziel ist es, zu verstehen, was „hinter den Kulissen“ passiert, wenn man später in der Arduino-IDE auf einen Knopf drückt.

### Hinweis zur Verwendung des Laborblatt

Dieses Laborblatt dient in erster Linie als praktische Anleitung zur Durchführung einzelner Versuche und Experimente. Es zeigt Schritt für Schritt, was zu tun ist, und enthält – wo sinnvoll – kurze Hinweise zur Bedeutung einzelner Befehle oder Schritte. Die technischen Hintergründe werden bewusst nur knapp angerissen.

Eine ausführlichere Erklärung der theoretischen und technischen Zusammenhänge findet sich im separat bereitgestellten **Projektbegleiter**. Beide Materialien können unabhängig voneinander verwendet werden: Man kann zum Beispiel zuerst das Laborblatt durcharbeiten, um ein Gefühl für die Abläufe zu bekommen, und danach die Hintergründe nachlesen – oder umgekehrt. Auch ein Wechsel zwischen beiden Herangehensweisen ist möglich. Jeder kann seinen eigenen Zugang wählen – je nach Lernstil und Interesse.

### Schritt 1: Unser erstes Programm für den ATmega328P

In diesem Schritt soll zunächst eine Datei `led_pin2.hex` mit folgendem Inhalt erzeugt werden:

```
:08000000529A5A9A5A98FDCF5A  
:00000001FF
```

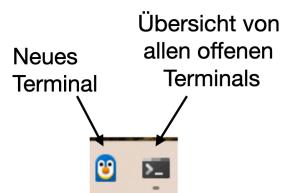
Diese Datei enthält Maschinencode für den ATmega328P. Eine ausführlichere Erklärung dazu findet sich im **Projektbegleiter**, eine knappe Erläuterung steht weiter unten im Arbeitsblatt.

Mit dem Programm `avrdude` kann dieser Maschinencode in den Programmspeicher des Mikrocontrollers geschrieben werden (auch „flashen“ genannt).

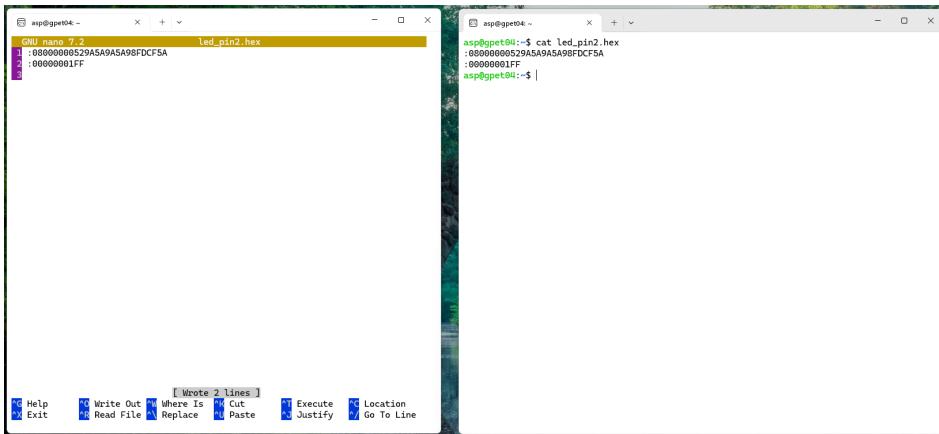
## Zwei Terminals – warum?

Für diesen Arbeitsschritt empfiehlt es sich, mit zwei Terminalfenstern zu arbeiten. Im Labor verwenden wir zwar Windows-PCs, aber zum Arbeiten nutzen wir Linux über WSL (Windows Subsystem for Linux).

Unten in der Taskleiste findet ihr: das Icon, um ein neues Linux-Terminal zu öffnen, und die Übersicht aller bereits offenen Terminals.



Am besten öffnet ihr zwei Terminals nebeneinander – zum Beispiel so:



| Terminal links | Terminal rechts                      |
|----------------|--------------------------------------|
| Editor (nano)  | Werkzeuge wie<br>ls, cat,<br>avrdude |

Der Vorteil: Ihr habt einen stressfreien Workflow. Links editiert ihr eure Datei, rechts testet ihr sie oder gebt Befehle ein. Wenn z. B. ein Fehler auftaucht, geben Werkzeuge oft eine Zeilennummer an – und die habt ihr direkt im Blick, wenn ihr links korrigiert.

### Aufgaben:

#### 1. Datei erstellen und Inhalt eingeben

- (a) Gebt im linken Terminal folgendes Kommando ein:

```
nano led_pin2.hex
```

Das startet den Texteditor `nano` mit einer (noch leeren) Datei namens `led_pin2.hex`.

- (b) Gebt nun den oben stehenden Code ein (zwei Zeilen).

Achtet darauf, ihn exakt zu übernehmen – inklusive Doppelpunkten. Die Buchstaben A–F dürfen hier klein oder groß geschrieben werden, aber denkt daran: In vielen Programmiersprachen macht die Groß-/Kleinschreibung einen Unterschied. Ein nicht seltener Fehler ist übrigens auch, dass man `0` ("Oh") statt `\0` ("Null") tippt!

- (c) Speichert die Datei mit der Tastenkombination `Control + S`.

#### 2. Erste Schritte mit Unix-Kommandos

Im rechten Terminal könnt ihr nun die Datei überprüfen:

- (a) Gebt ein:

```
ls
```

Das listet alle Dateien im aktuellen Verzeichnis auf. Die Datei `led_pin2.hex` sollte jetzt erscheinen.

- (b) Gebt ein:

```
cat led_pin2.hex
```

Das gibt den Inhalt der Datei im Terminal aus. Vergleicht ihn mit dem, was ihr links in `nano` eingegeben habt.

## Schritt 2: Mikrocontroller verbinden und Gerät erkennen

Bevor wir den Mikrocontroller anschließen, schauen wir uns kurz an, wie man unter Linux erkennen kann, ob ein Gerät (z.B. ein USB-Seriell-Wandler) vom System erkannt wurde.

### Terminal-Tipp:

Wenn ihr im Terminal ein Kommando erneut eingeben wollt, könnt ihr einfach die Pfeil-hoch-Taste drücken. Das ist besonders praktisch, wenn man ein Kommando öfter ausführt oder leicht anpassen will.

### Aufgaben:

#### 1. Noch kein Mikrocontroller anschließen

Gebt im Terminal folgendes Kommando ein:

```
ls /dev/*
```

Dieser Befehl listet alle Geräte-Dateien auf, die sich im Verzeichnis `/dev/` befinden.

Unter Linux wird fast alles als „Gerätedatei“ behandelt – auch USB-Geräte. Ein USB-Seriell-Wandler wie der auf unserem Mikrocontroller-Board würde hier z. B. als `/dev/ttyUSB0` auftauchen.

Da noch kein Mikrocontroller angeschlossen ist, sollte natürlich in der Ausgabe **kein** Eintrag wie `/dev/ttyUSB0` erscheinen.

#### 2. Mikrocontroller anschließen

Jetzt schließt ihr den Mikrocontroller per USB-Kabel an den Rechner an. Auf einem echten (nativen) Linux-System würde nun bei erneutem Ausführen von

```
ls /dev/*
```

ein zusätzlicher Eintrag wie `/dev/ttyUSB0` erscheinen – das wäre euer Gerät.

Da wir aber in WSL unter Windows arbeiten, läuft Linux hier als „Gast“, und der USB-Anschluss muss durchgereicht werden. Dafür haben wir ein Hilfsskript namens `usb_check` vorbereitet.

#### 3. `usb_check` aufrufen

Gebt ein:

```
usb_check
```

Das Skript sorgt dafür, dass WSL Zugriff auf den USB-Anschluss bekommt. Prüft danach nochmal mit:

```
ls /dev/*
```

ob jetzt `/dev/ttyUSB0` erscheint. Wenn ja: alles bereit zum Flashen!

### Was tun bei Problemen?

Wenn `usb_check` einen Fehler liefert oder nach dem Admin-Passwort fragt: Keine Panik. Solche Dinge können vorkommen – besonders, wenn kurz vor Semesterbeginn noch an Workarounds geschraubt wurde 😊 Lehnt euch kurz zurück, bleibt entspannt – wir helfen euch gleich weiter.

### Schritt 3: Mikrocontroller flashen

Jetzt wird's ernst: Wir übertragen unser Programm auf den Mikrocontroller – also in den Flash-Speicher, wo es nach dem Einschalten automatisch ausgeführt wird.  
Dazu gebt ihr im rechten Terminal den folgenden Befehl ein:

```
avrduude -c arduino -P /dev/ttyUSB0 -p atmega328p -U flash:w:led_pin2.hex:i
```

Wenn alles klappt, sieht die Ausgabe von avrdude ungefähr so aus:

```
asp@gpet04:~$ cat led_pin2.hex
:08000000529A5A9A5A98FDCF5A
:00000001FF
asp@gpet04:~$ avrdude -c arduino -P /dev/ttyUSB0 -p atmega328p -U flash:w:led_pin2.hex:i

avrduude: AVR device initialized and ready to accept instructions
avrduude: device signature = 0x1e950f (probably m328p)
avrduude: Note: flash memory has been specified, an erase cycle will be performed.
          To disable this feature, specify the -D option.
avrduude: erasing chip
avrduude: reading input file led_pin2.hex for flash
          with 8 bytes in 1 section within [0, 7]
          using 1 page and 120 pad bytes
avrduude: writing 8 bytes flash ...

Writing | ##### | 100% 0.03 s

avrduude: 8 bytes of flash written
avrduude: verifying flash memory against led_pin2.hex

Reading | ##### | 100% 0.02 s

avrduude: 8 bytes of flash verified

avrduude done. Thank you.

asp@gpet04:~$
```

Jetzt machen wir eine kleine Fallunterscheidung – ganz so wie beim Programmieren mit `if` und `else`:

- Falls in der Ausgabe etwas steht wie:

```
8 bytes of flash written
8 bytes of flash verified
```

Dann war das Flashen erfolgreich. Weiter mit dem nächsten Schritt!

- Wenn dagegen `avrduude` einen Fehler meldet – oder „verifying failed“ erscheint – dann:

**Zurück zu Schritt 1!** Wahrscheinlich hat sich beim Eingeben der Hex-Datei ein Tippfehler eingeschlichen. Geht noch mal in `nano`, überprüft alles, speichert erneut und versucht es dann wieder.

#### Fehler gehören zum Workflow

Fehler beim Flashen (oder allgemein beim Programmieren) sind völlig normal. Wichtig ist, dass man sie **als Teil des Workflows versteht**:

- Man erkennt, dass ein Fehler aufgetreten ist (z. B. durch eine ungewöhnliche `avrduude`-Ausgabe),
- Man lernt, solche Meldungen nicht zu überlesen, sondern sie irgendwann sogar unbewusst wahrzunehmen – so wie man eine rote Ampel erkennt, ohne bewusst hinzusehen.
- Und man weiß, zu welchem Schritt man zurückgehen muss, um ihn zu beheben.

Genau das ist professionelle Arbeitsweise: Fehler sind kein Rückschritt – sie sind der Weg zur Lösung.

## Schritt 4: Nachmessen, was das Programm macht

Gratulation – du hast es bis Schritt 4 geschafft!

Das Maschinenprogramm wurde bereits erfolgreich auf den Mikrocontroller übertragen und läuft jetzt. Es wird also langsam Zeit, herauszufinden, was wir da eigentlich genau gemacht haben.

Die Details findest du zwar im Projektbegleiter – aber ein Handbuch liest man ja bekanntlich gerne erst im Nachhinein. Deshalb hier das Wichtigste in Kürze. Unser Programm besteht aus genau vier Maschinenbefehlen:

- Pin 2 als Ausgang setzen
- Pin 2 auf HIGH setzen
- Pin 2 auf LOW setzen
- und ein Sprungbefehl

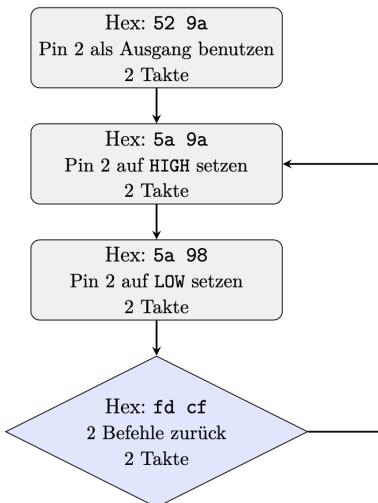


Abb. 1: Ablaufdiagramm

Diese Befehle werden – wie in Abbildung 1 dargestellt – nacheinander ausgeführt. Jeder dieser Befehle benötigt zwei CPU-Takte. Bei einer Taktfrequenz von 16 MHz dauert ein Takt genau 62,5 ns. Das bedeutet: Der Pin 2 sollte 125 ns HIGH sein, danach 250 ns LOW – und dann beginnt der Ablauf wieder von vorne. Abbildung 2 skizziert diesen Ablauf.

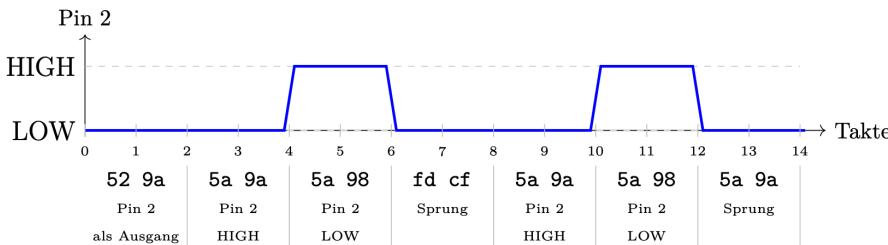
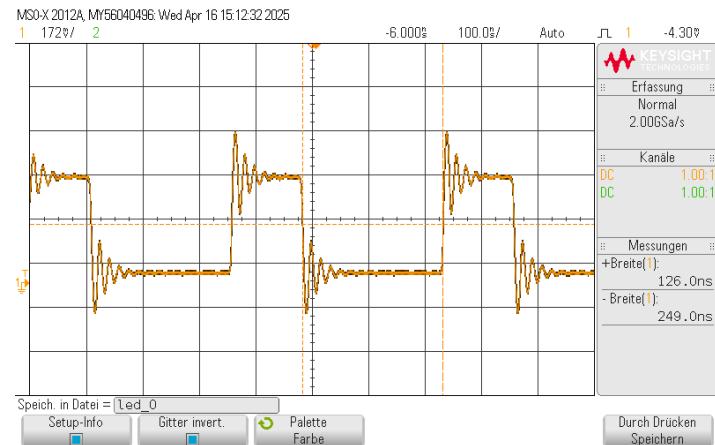
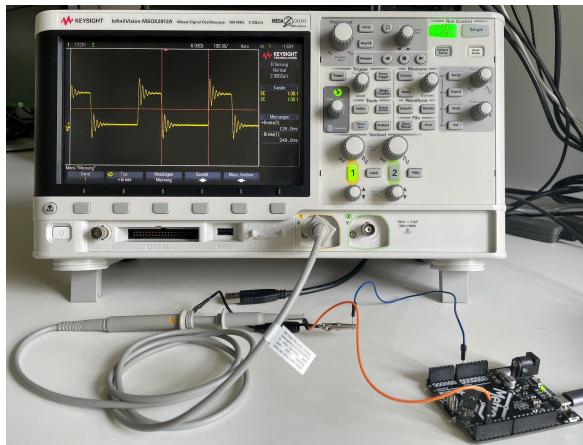


Abb. 2: Signalverlauf an Pin2.

Aber das kann ja jeder behaupten. Wir aber messen es nach – mit dem Oszilloskop!



1. Messleitung anschließen:
2. Trigger einstellen:
3. Zeitbasis anpassen:
4. Automatische Messung aktivieren:

Kanal 1 (gelb) an Pin 2 des Mikrocontrollers mit Masseleitung (Kabel mit Krokodilklemme) an GND. Mode: Edge, Source: Chan 1, Level ca. 2.5V Time/Div: ca. 100 ns Drücke die Taste Meas und wähle nacheinander: +Breite (misst HIGH-Dauer), -Breite (misst LOW-Dauer)

## Schritt 5: Kleiner Hack im Maschinencode

Im Projektbegleiter ist das sogenannte Intel Hex Format ausführlich erklärt. Hier die Kurzfassung. Die Datei led\_pin2.hex enthält mehr als nur das eigentliche Programm. Neben dem Maschinenprogramm enthält sie z. B.:

- wie viele Bytes übertragen werden sollen
- an welche Adresse sie geschrieben werden sollen
- eine Prüfsumme zur Fehlererkennung

Die erste Zeile hat zum Beispiel folgenden Aufbau

|  |      |      |    |                  |    |
|--|------|------|----|------------------|----|
|  |      |      |    |                  |    |
|  | : 08 | 0000 | 00 | 529A5A9A5A98FDCF | 5A |

**08** = Länge der Datenbytes, **0000** = Startadresse, **00** = Record-Typ (Daten), **5A** = Prüfsumme

Das Maschinenprogramm selbst besteht nur aus den grün hinterlegten Bytes

529a5a9a5a98fdc f

Die zweite Zeile der Datei (:00000001FF) bedeutet übrigens nur: Es folgen keine weiteren Daten.

**Jetzt kommt der Clou:** Die Prüfsumme im Intel-Hex-Format erkennt viele Arten von Übertragungsfehlern – aber nicht alle. Wenn zwei aufeinanderfolgende Befehle vertauscht werden, ändert sich zwar das Programmverhalten – die Prüfsumme bleibt aber gleich.

### Aufgabe:

Vertausche in led\_pin2.hex die Befehle für „Pin 2 auf HIGH“ und „Pin 2 auf LOW“. Wenn alles klappt, solltest du im Oszilloskop sehen: 250 ns HIGH und 125 ns LOW

**Unterschied zu Schritt 4:** Dort haben wir nachgemessen, um zu verstehen, was passiert. Jetzt machen wir eine Vorhersage und überprüfen dann, ob sie stimmt.

## Schritt 6: Hack für symmetrisches Signal

Im Projektbegleiter wird erklärt, wie man das Programm verändern kann, sodass **Pin 2 für 250 ns HIGH** und danach **für 250 ns LOW** ist.

Das Signal besteht dann aus **gleich langen Phasen** – es ist also **symmetrisch** und hat eine saubere Frequenz.

Dazu fügen wir einfach zwei zusätzliche Befehle ein: sogenannte **NOPs**:

- NOP steht für „No Operation“ – also „Tu nichts“.
- Der Befehl verbraucht einen CPU-Takt, verändert aber nichts am Zustand des Pins:
  - Ist Pin2 HIGH, bleibt er HIGH.
  - Ist er LOW, bleibt er LOW.

In unserem Fall fügen wir die beiden NOPs direkt nach dem „Pin auf HIGH“-Befehl ein. Dadurch verlängert sich die HIGH-Phase um

$$125 \text{ ns} + 125 \text{ ns} = 250 \text{ ns}.$$

Wie das genau aussieht zeigt Abbildung 3.

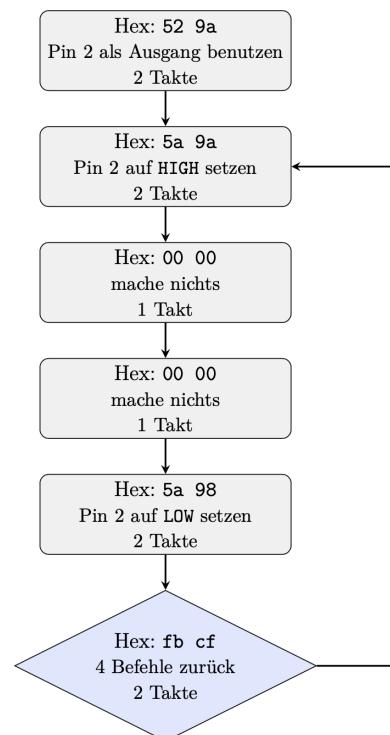


Abb. 3: Symmetrisches Signal

Das neue Programm besteht nun aus den folgenden Bytes:

529A5A9A000000005A98FBCF

Also zusammengefasst:

In `led_pin2.hex` muss jetzt nur die erste Zeile angepasst werden.

Das einzig Knifflige: Wie lautet die neue Prüfsumme?

|              |                          |    |
|--------------|--------------------------|----|
| : OC 0000 00 | 529A5A9A000000005A98FBCF | ?? |
|--------------|--------------------------|----|

Natürlich könnte man im Projektbegleiter genau nachlesen, wie man sie berechnet. Aber hier reicht das Wichtigste:

Zunächst werden alle Bytes bis auf die Prüfsumme addiert – gerechnet wird dabei immer zweistellig in Hex, also mit 8-Bit-Überlauf: höhere Stellen werden einfach abgeschnitten. Die Prüfsumme ist dann genau die Zahl, die man noch addieren müsste, damit das Ergebnis wieder 0 ist.

Das Entscheidende für uns: Wir müssen nur schauen, **wo sich etwas verändert hat**, und **wie stark sich die Summe verändert**.

|                     |   |                     |
|---------------------|---|---------------------|
| <b>um 4 größer</b>  | <b>neu</b>                                  | <b>um 2 kleiner</b> |
| : <b>OC</b> 0000 00 | 529A5A9 <b>A0000000</b> 5A98 <b>FBCF</b> ?? |                     |

Und das sind genau drei Dinge:

1. Die Anzahl der Bytes hat sich um 4 erhöht.
2. Die beiden neuen NOP-Befehle (`00 00 00 00`) tragen 0 zur Summe bei.
3. Der Sprungbefehl wurde verändert: aus `FD CF` wird `FB CF`, d.h. das erste Byte ist also um 2 kleiner

Damit ist die Summe um 2 kleiner geworden

- die Prüfsumme muss also um 2 größer werden, um das wieder auszugleichen.
- Die alte Prüfsumme war `5A`, also sollte die neue `58` sein.

### Aufgabe:

Ändere `led_pin2.hex`, indem du

- die neuen Maschinenbefehle einfügst,
- die Datenlänge entsprechend erhöhst,
- und die Prüfsumme von `5A` auf `58` änderst.

Wenn wir keinen Denkfehler und keinen Rechenfehler gemacht haben, wird avrdude die Prüfsumme kommentarlos akzeptieren, und das Oszilloskop wird ein symmetrisches Signal anzeigen.

## Intermezzo: Ein Blick ans Ende

In Schritt 7 klonen wir ein fertiges Spiel und spielen es gemeinsam. Nicht, weil wir den Code analysieren wollen – sondern um einen Eindruck zu bekommen, worauf dieser Kurs hinausläuft. Was ihr da gleich spielt, ist ein Beispiel für das, was am Ende möglich ist:

### Ein selbst geschriebenes Spiel in Python, das man mit einem Mikrocontroller steuern kann.

Was genau ihr programmiert, könnt ihr natürlich selbst entscheiden. Für Klassiker wie Pong oder Breakout (das ist das Spiel, bei dem man mit einem Paddel unten einen Ball reflektiert und oben Blöcke trifft) wird es schrittweise Anleitungen geben – aber auch genug Raum für eigene Ideen.

Im Laufe des Kurses bewegen wir uns von der hardwarenahen Programmierung (Maschinencode, Assembler, Ports, Timing ...) bis hin zu plattformunabhängiger Softwareentwicklung in Python. Dabei geht es nicht darum, Lösungen auswendig zu lernen, sondern zu verstehen, welches Problem jeweils gelöst wird – und warum ein bestimmter Ansatz sinnvoll ist. Wenn man das Problem kennt, versteht man auch den Zweck der Lösung – ganz gleich ob es um Assembler, C oder objektorientierte Programmierung geht. Und man kann besser einschätzen, wann und wie man solche Werkzeuge sinnvoll einsetzt.

## Schritt 7: TRON Lightcycle spielen

Zum Abschluss des ersten Treffens wird gespielt!

Das Spiel erinnert auf den ersten Blick vielleicht an **Snake** – hat aber andere Wurzeln: **TRON** war ein Science-Fiction-Film aus dem Jahr **1982** und der erste Kinofilm, der umfangreich mit **Computer-Generated Imagery (CGI)** gearbeitet hat. Die Lightcycle-Rennen daraus haben viele Spielideen beeinflusst – darunter auch dieses.

### Aufgaben

1. Das Spiel liegt auf GitHub unter: <https://github.com/michael-lehn/tron.git>

Zum Klonen gebt ihr im Terminal Folgendes ein:

```
git clone https://github.com/michael-lehn/tron.git
```

2. **Wer macht was?**

Das Spiel kann man zu viert spielen – und das passt perfekt, denn im Labor gibt es vier Rechner pro Gang. Einer von euch startet den Server, die anderen verbinden sich mit dem Client.

**(A) Server starten:** Öffne ein Terminal, wechsle ins Verzeichnis und starte den Server:

```
cd tron  
python new-tron-server.py 1000 1000 4
```

Das startet ein Spiel auf einem  $1000 \times 1000$  Grid für vier Spieler. Wer den Server startet, sollte auch die eigene IP-Adresse ermitteln und den anderen mitteilen:

```
curl -4 ifconfig.me
```

**(B) Client starten:** Die Clients werden auf jedem Rechnern wie folgt gestartet:

```
cd tron  
python tron-2d.py      (oder python tron-3d.py)
```

Im Startbildschirm könnt ihr dann euren Namen eingeben und die IP-Adresse des Servers eintragen.