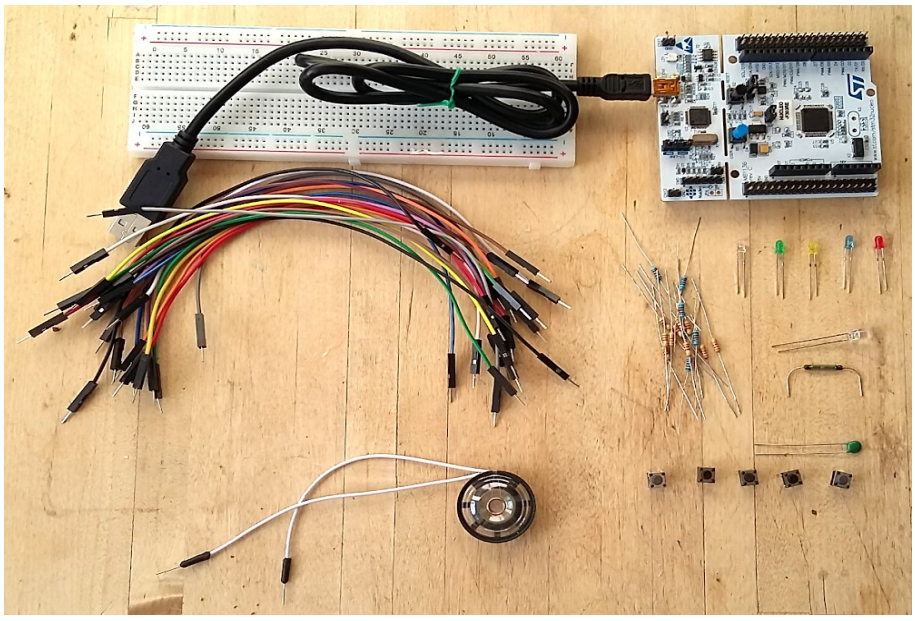
# Einblick in die moderne Elektronik

## ohne viel Theorie

zweite Auflage
von Stefan Frings



# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Vorwort

Unser ganzes Leben lang werden wir von elektronischen Geräten unterstützt und beeinflusst. Und überall stecken Mikrocontroller drin. Deswegen finde ich Mikrocontroller super spannend. Was macht sie so vielseitig? Wie kontrolliert man sie? Was können sie und was nicht?

Mit diesem neuen Schnupperkurs möchte ich Teenagern helfen, ihr Talent für die moderne Elektronik zu entdecken. Anstatt mit theoretischen Grundlagen zu beginnen, machen wir es wie in der Fahrschule: Einfach losfahren...

Stefan Frings, Juli 2020

## 1.2 Material

Besorge das auf der Titelseite dargestellte Material:

| **Anzahl** | **Artikel** |
|---|---|
| 1 | Nucleo-F303RE Board von ST, mit STM32F303RE Mikrocontroller |
| 1 | USB Kabel mit Mini-USB Stecker |
| 1 | Steckbrett, Breadboard mit ca. 830 Kontakten |
| 20 | Dupont Kabel Stecker-Stecker, Jumper Wire Male-Male |
| 1 | Digital-Multimeter, das einfachste Modell genügt.<br>Zum Beispiel ein Voltcraft VC-11 oder ein DT-830D |
| 1 | Kleiner Lautsprecher mit 32 Ohm (z.B. aus einem Kopfhörer) |
| 4 | Kurzhubtaster, tactile switch, 6x6mm |
| 5 | Standard Leuchtdioden 3 mm<br>jeweils eine in rot, gelb, grün, blau und weiß |
| 1 | Heißleiter, NTC, 10k Ohm |
| 1 | Phototransistor PT331C |
| 1 | Reed-Kontakt, 1 Schließer |
| 1 | Kleiner Magnet, beliebige Form, um den Reed-Kontakt zu betätigen |
| 5 | Bedrahtete Widerstände 220 Ohm ¼ Watt |
| 5 | Bedrahtete Widerstände 2,2k Ohm ¼ Watt |

# 2 Es werde Licht

Es geht direkt mit einem Experiment los. Schaue dir die zuerst die Bilder an und lese die Erklärungen dazu, bevor du die Teile zusammen steckst. Das folgende Bild zeigt anhand der weißen Linien, welche Löcher in deinem Steckbrett miteinander verbunden sind:

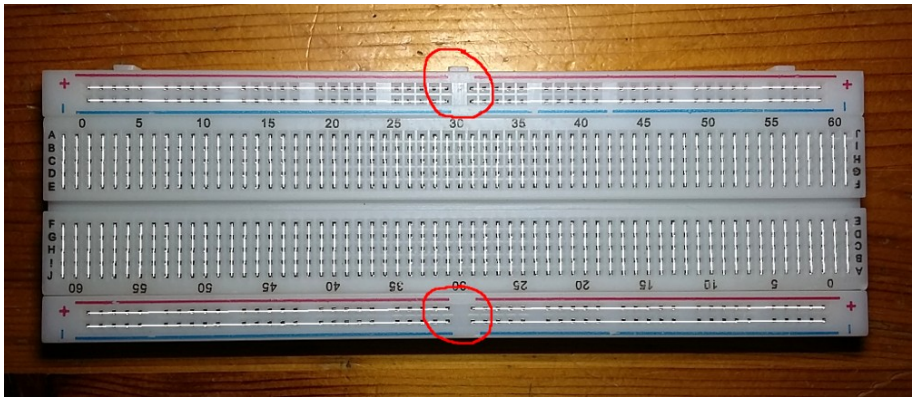


*Abbildung 1: Steckbrett mit Hervorhebung der Verbindungen*

Bei vielen aber nicht allen Steckbrettern sind die horizontalen Verbindungen an den rot eingekreisten Stellen unterbrochen. Die Experimente in diesem Buch funktionieren mit beiden Varianten.

Im ersten Experiment soll eine rote Leuchtdiode gemäß dem folgenden Schaltplan angeschlossen werden. Verwende dabei irgendeinen Widerstand von der Materialliste:

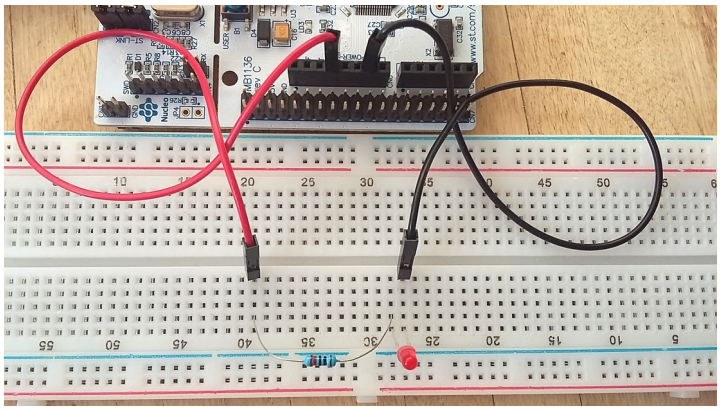


*Abbildung 2: Aufbau mit LED und Widerstand*

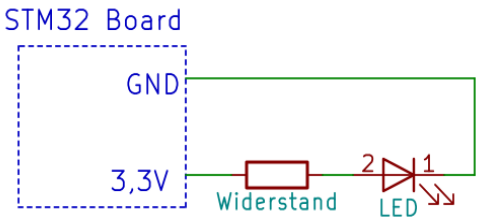



*Abbildung 3: Schaltplan, LED und Widerstand*

Detail-Ansicht:

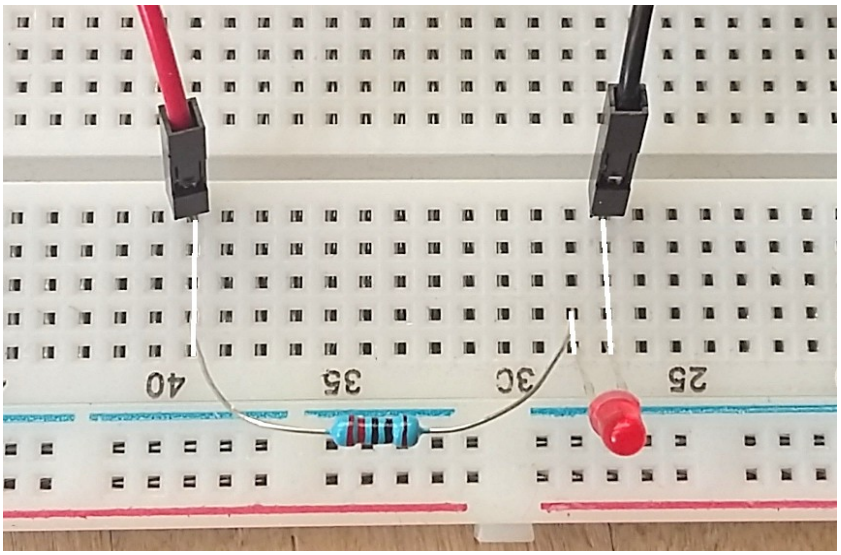



*Abbildung 4: Aufbau mit LED und Widerstand, Positionierung der Bauteile*

Schließe das schwarze Kabel an eine „GND“ Buchse auf dem Nucleo-Board an. Das rote Kabel gehört an die 3,3 Volt Versorgungsspannung. Die entsprechende Buchse ist mit „3V3“ beschriftet. Bei der LED gehört der längere Draht auf die linke Seite. Ich habe in dem obigen Foto die relevanten Verbindungen durch das Steckbrett eingezeichnet.

| |
|---|
| Tipp: Neue Steckbretter enthalten oft sehr stramme Kontaktfedern. Eventuell magst du die Kontakte vorher mit einer stumpfen Nadel lockern. |

Verbinde dein Nucleo-Board jetzt mit einem PC. Die Leuchtdiode sollte nun leuchten. Probiere auch die anderen Leuchtdioden aus. Leuchten sie gleich hell? Schau dir deine Widerstände genau an. Sie haben unterschiedliche bunte Ringe aufgedruckt. Probiere einen anderen Widerstand aus, und beobachte seine Wirkung auf die Helligkeit der Leuchtdiode.

## 2.1 Leuchtdiode

LED ist die englische Abkürzung für „Light Emitting Diode“, auf Deutsch: Leuchtdiode. Sie leuchtet, wenn sie von Strom durch-flossen wird. Die LED lässt den Strom nur dann fließen, wenn der längere Anschluss zum Plus-Pol der Stromquelle führt (in diesem Fall der 3,3V Anschluss des Mikrocontroller Boards).

Es gibt auch Leuchtdioden mit unsichtbarem Licht, denn der Mensch kann nicht alle Farben sehen. Diese findest du zum Beispiel in der Fernbedienung von Fernsehgeräten.

Schau dir die weiße Leuchtdiode einmal genauer an. In ihrem Innern befindet sich ein Reflektor, in dessen Mitte ein winzig kleiner Kristall fest geklebt ist. Dieser ist über einem sehr dünnen Draht mit dem anderen Anschlussbein verbunden. In dem Foto rechts habe ich ihn gelb hervorgehoben.

*Abbildung 5: Blick ins Innere einer LED*

Der Kristall leuchtet, wenn er von Strom durch-flossen wird. Allerdings muss man unbedingt dafür sorgen, dass nicht zu viel Strom fließt, weil er sonst überhitzt und kaputt geht. Standard Leuchtdioden wie die rechts abgebildete vertragen eine Stromstärke von maximal 20 mA. Ihre Betriebsspannung liegt je nach Modell zwischen 1,6 und 3,5 Volt.

Ich werde in den nächsten Kapiteln erklären, was das genau bedeutet. Doch zuvor möchte ich noch klarstellen, wozu der Widerstand gut ist.

## 2.2 Widerstand

Widerstände (auf Englisch abgekürzt mit R für „Resistor“) leisten dem elektrischen Strom Widerstand. Dass heißt, sie bremsen den Strom aus. Das ist mit einer besonders dünnen Wasserleitung vergleichbar, welche die Durchflussmenge reduziert.

*Abbildung 6: Widerstand*

In der Schaltung mit der Leuchtdiode dient der Widerstand dazu, die Stromstärke zu reduzieren, damit die Leuchtdiode nicht überhitzt. Je nach dem, welchen Widerstand du benutzt, leuchtet die LED hell oder dunkel.

Die bunten Farbringe geben an, wie stark der Widerstand bremst.

| | | | |
|---|---|---|---|
| 220 Ohm: | rot-rot-schwarz-schwarz | oder | rot-rot-braun |
| 2200 Ohm: | rot-rot-braun-schwarz | oder | rot-rot-rot |

Dahinter kommt noch ein weiterer Ring mit etwas mehr Abstand, der die Präzision des Bauteils angibt. Für die Experimente in diesem Buch ist die Präzision allerdings unwichtig. Die genaue Bedeutung der Farbcodes kannst du bei Wikipedia nachlesen, falls es dich interessiert.

https://de.wikipedia.org/wiki/Widerstand_(Bauelement)#Farbkodierung_auf_Widerst.C3.A4nden

Elektroniker benutzen als Abkürzung für Ohm den griechischen Buchstaben Ω (Omega). Gelegentlich findet man das R anstelle von Ω, zum Beispiel 220 R für 220 Ω oder 1R8 für 1,8 Ω. Die Angabe 2k2 bedeutet 2,2 kΩ (=2200 Ω)

# 3 Das Multimeter

Zu deinem Material gehört ein digitales Multimeter, zum Beispiel dieses Modell:


*Abbildung 7: Ein einfaches Multimeter von Conrad Elektronik*

Bevor du dein Multimeter benutzt, solltest du dessen Bedienungsanleitung lesen. Besonders wichtig sind darin die Sicherheitshinweise, die Beschreibung der Steckbuchsen (falls vorhanden) und die Information, wo sich die Sicherung befindet, denn diese muss man manchmal erneuern.

Zuerst zeige ich dir, wie man Spannungen misst. Stecke dazu die Kabel in die Buchsen „V“ (rot) und „COM“ (schwarz), falls sie steckbar sind. Bei dem oben abgebildetem Gerät sind die Kabel fest angeschlossen. Stelle den Drehschalter auf den 200 Volt Bereich. So kannst du nun Spannungen bis zu diesem Höchstwert messen. Halte dann die beiden Mess-Spitzen an die Enden einer Batterie:

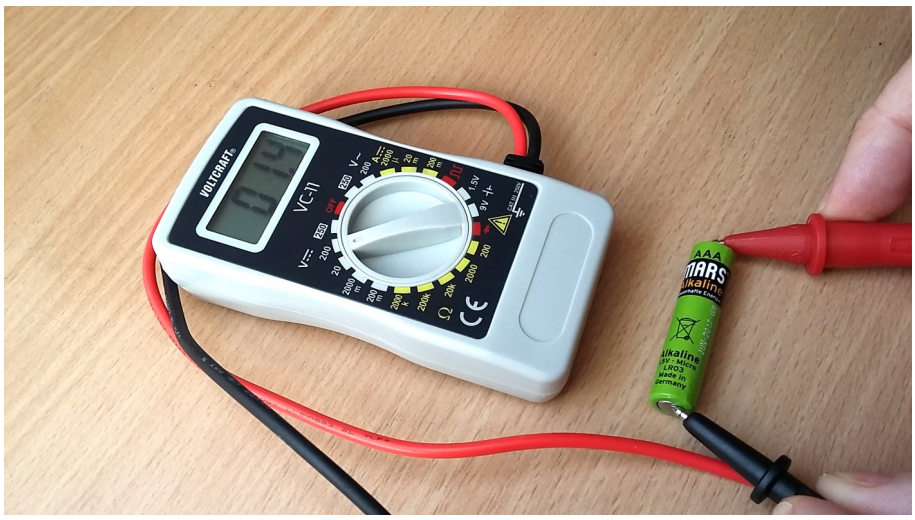


*Abbildung 8: Messung der Spannung einer Batterie*

Das Messgerät zeigt 1,4 Volt an. Das ist die Spannung, die meine Batterie gerade abgibt.

Du kannst das Gerät auch in den 20 V Bereich umschalten, dann zeigt es eine Nachkommastelle mehr an. Bei der Wahl des Messbereiches ist zu beachten, dass man das Messgerät niemals mit mehr Spannung belastet, als der Messbereich erlaubt. Denn dabei kann es kaputt gehen. Für dieses Buch ist der 20 V Bereich ideal.

Halte das Messgerät an die beiden Anschluss-Beinchen der Leuchtdiode, während sie leuchtet. Dann siehst du, welche Betriebsspannung sie gerade hat:

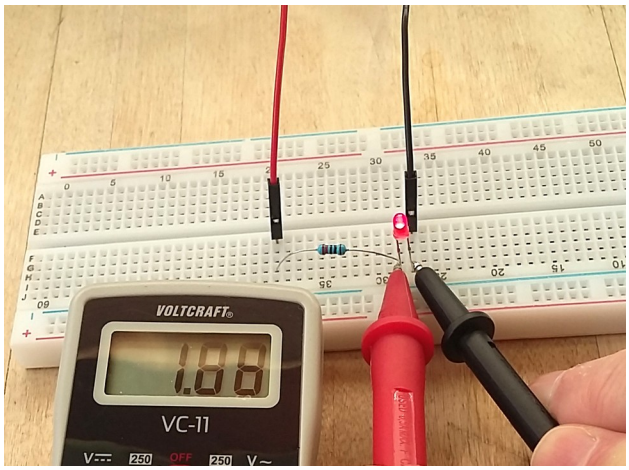



*Abbildung 9: Messung der an der LED abfallenden Spannung*

In diesem Foto habe ich den 220 Ω Widerstand verwendet. Das Messgerät zeigt 1,88 V an. Benutze nun den anderen 2200 Ω Widerstand. Da er nur ein Zehntel so viel Strom fließen lässt, leuchtet die LED viel schwächer. Dennoch ist die Spannung fast unverändert geblieben, nämlich 1,69 V.

| |
|---|
| Bei Leuchtdioden ist die Betriebsspannung fast konstant. Ihre Helligkeit ergibt sich aus der Stromstärke. |

Übungsaufgabe:

Messe die Betriebsspannung der anderen Leuchtdioden.

Jetzt habe ich schon mehrfach geschrieben, dass die Widerstände mehr oder weniger Strom fließen lassen. Mit der Leuchtdiode konntest du den Unterschied anhand ihrer Helligkeit auch sehen. Nun wollen wir die tatsächliche Stromstärke messen.

Stelle das Multimeter dazu auf den 200 mA Bereich ein. Wenn es steckbare Anschlusskabel hat, musst du das rote jetzt eventuell in die „mA“ Buchse umstecken. Um die Stromstärke zu messen, muss man den Stromkreis so ändern, dass der Strom durch das Multimeter hindurch fließen muss (wie bei der Wasseruhr im Keller).

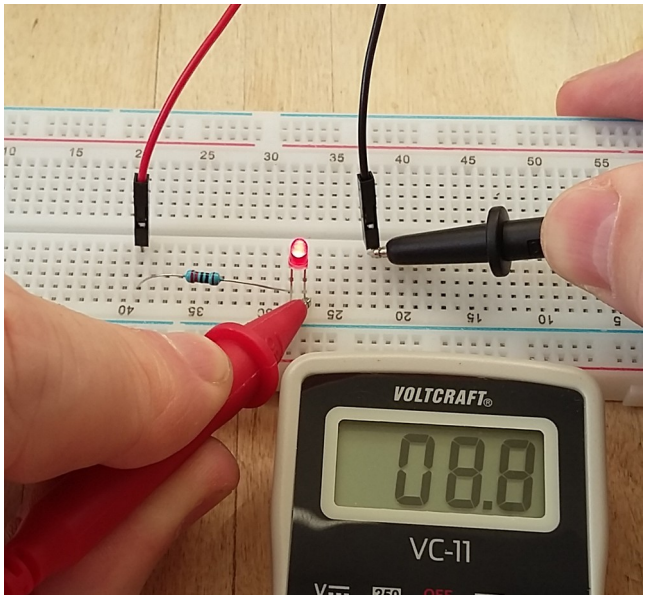



*Abbildung 10: Messung der Stromstärke im Stromkreis einer LED*

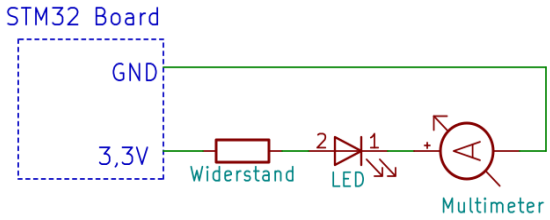



*Abbildung 11: Schaltplan, Messung der Stromstärke im Stromkreis einer LED*

Das Multimeter zeigt an, dass in diesem Stromkreis 8,8 mA (Milliampere) fließen. Wenn du den anderen 2200 Ω widerstand verwendest, beträgt die Stromstärke etwa 0,8 mA. Du siehst: Da der Widerstand zehn mal so stark bremst, fließt nur ein Zehntel so viel Strom.

Spannungen misst du im „20 V“ Messbereich, indem du das Messgerät an zwei beliebige Punkte in der bestehenden Schaltung hältst.

Stromstärken misst du im „200 mA“ Bereich, wobei du die Schaltung so änderst, dass der Strom durch das Messgerät hindurch fließen muss.

Wenn das Messgerät auf Strom-Messung („A“ oder „mA“) eingestellt ist, darfst du die Mess-Spitzen auf keinen Fall direkt an eine Stromquelle halten.

Der Strom würde ungebremst durch das Messgerät fließen und dadurch die Sicherung zerstören. Deswegen zeigt meins nun stets 00.0 an. Es muss repariert werden.

Kontrolliere also vor jeder Messung, ob das Messgerät richtig eingestellt ist und ob die Kabel in den richtigen Buchsen stecken (falls sie steckbar sind).

In den allermeisten Fällen wirst du Spannungen im 20 V Bereich messen.

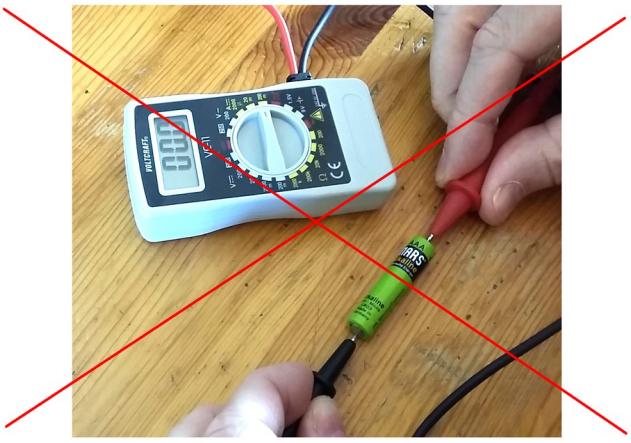


*Abbildung 12: Falsche Verwendung des Multimeters bei der Messung der Stromstärke*

# 4 Spannung und Strom

Du hast bisher Spannungen und Ströme gemessen. Du hast gesehen, dass die Stromstärke eine LED mehr oder weniger hell leuchten lässt. In diesem Kapitel erkläre ich diese beiden Begriffe.

Alles, was elektrischen Strom leiten kann (z.B. Kabel) enthält bewegliche Elektronen. Wenn diese gezielt in eine bestimmte Richtung wandern, dann „fließt“ der Strom. Dabei gibt es zwei wichtige Messgrößen, nämlich die Spannung und die Stromstärke.

Spannung = Druck

Die Spannung sagt aus, wie viel „Druck“ auf der Leitung ist. Bei dem Begriff denke ich an einen Wasserfall. Er hat viel Druck, weil das Wasser von weit oben herab fällt. Hohe Spannung hat die Fähigkeit, Isolationen (auch Luft) zu durchschlagen.

Strom = Menge

Die Stromstärke (oder kurz: der Strom) sagt aus, wie viele Elektronen durch die Leitung fließen. Denke an den breiten Rhein oder die Donau im Vergleich zu einem kleinen Bach.

Alle elektronischen Bauteile vertragen nur eine bestimmte Spannung und auch nur eine bestimmte Stromstärke. Beide Grenzwerte dürfen nicht überschritten werden.

Leistung = Strom · Spannung

Die Leistung (Power) von elektrischen Geräten ist das Produkt aus Stromstärke und Spannung in der Einheit Watt. Mein Wasserkocher hat zum Beispiel 10 Ampere · 230 Volt, also 2300 Watt. Das ist mehr als eine Pferdestärke!

## 4.1 Stromkreis

Jetzt habe ich den elektrischen Strom so schön mit Wasser verglichen, aber der Vergleich passt nicht bei allen Eigenschaften. Wenn du den Wasserhahn von der Wand abschraubst, fließt das Wasser heraus auf den Fußboden. Der Strom kommt jedoch nicht von alleine aus den offenen Löchern der Steckdose heraus, denn Strom kann nicht durch Luft fließen (jedenfalls nicht ohne besonderen Aufwand). Außerdem fließt der Strom nur dann, wenn er zu seiner Quelle zurück kehren kann.

Das hier klappt also nicht:

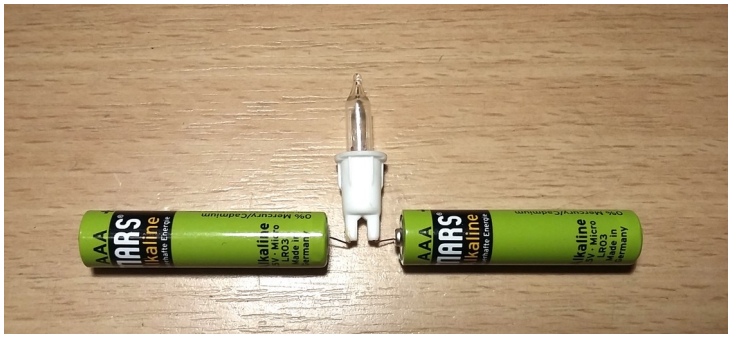


*Abbildung 13: Offener Stromkreis (es fließt kein Strom)*

Man muss schon einen Kreislauf bilden, damit der Strom fließen kann und die Glühbirne leuchtet. Also so:

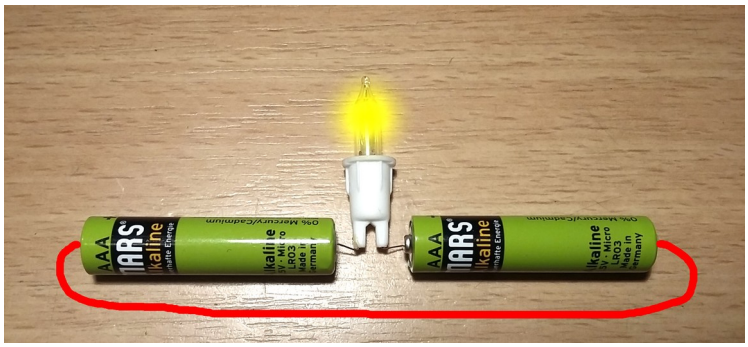


*Abbildung 14: Geschlossener Stromkreis (der Strom fließt)*

## 4.2 Statische Ladung

Nicht nur Batterien können mit Strom aufgeladen werden, sondern auch viele andere Gegenstände – sogar Menschen. Während Batterien den Stromfluss durch einen chemischen Vorgang erzeugen, entsteht statische Ladung meistens durch Reibung. Zum Beispiel wenn du dir einen Polyester Pullover über frisch gewaschene und getrocknete Haare ziehst. Dabei werden viele Elektronen vom Pullover auf den Körper verschoben und verbleiben dort, so dass ein Ungleichgewicht entsteht. Die Spannung, die sich dadurch aufbaut ist oft so hoch, dass kleine Blitze entstehen.

Bei diesem Jungen, dessen Foto ich auf Wikimedia gefunden habe, stehen die Haare buchstäblich zu Berge, weil er sehr stark aufgeladen ist. Wenn du in so einem Zustand die Anschlüsse eines Mikrochips anfasst, geht er mit 99 % Wahrscheinlichkeit kaputt. Auch geringere Ladungen können bereits schädlich wirken.

*Abbildung 15: Statisch geladener Junge*

Beim Umgang mit Elektronik ist es daher wichtig, den Arbeitsplatz und den Körper zu entladen. Für den Hausgebrauch genügt es, einen unlackierten Tisch aus Holz zu benutzen. Holz ist nämlich ein kleines bisschen leitfähig. Der Raum soll möglichst keinen Teppichboden haben und Kleidung aus Synthetik (Polyester, Fleece) oder Wolle soll man meiden.

Wer auf Nummer sicher gehen will, besorgt sich im Elektronik Handel eine Anti-Statik Matte und ein Armband, welche die Ladung zuverlässig abführen.

# 5 Das Nucleo-64 Board

Schau Dir dein Nucleo-Board an. Es ist in zwei Teile aufgeteilt.

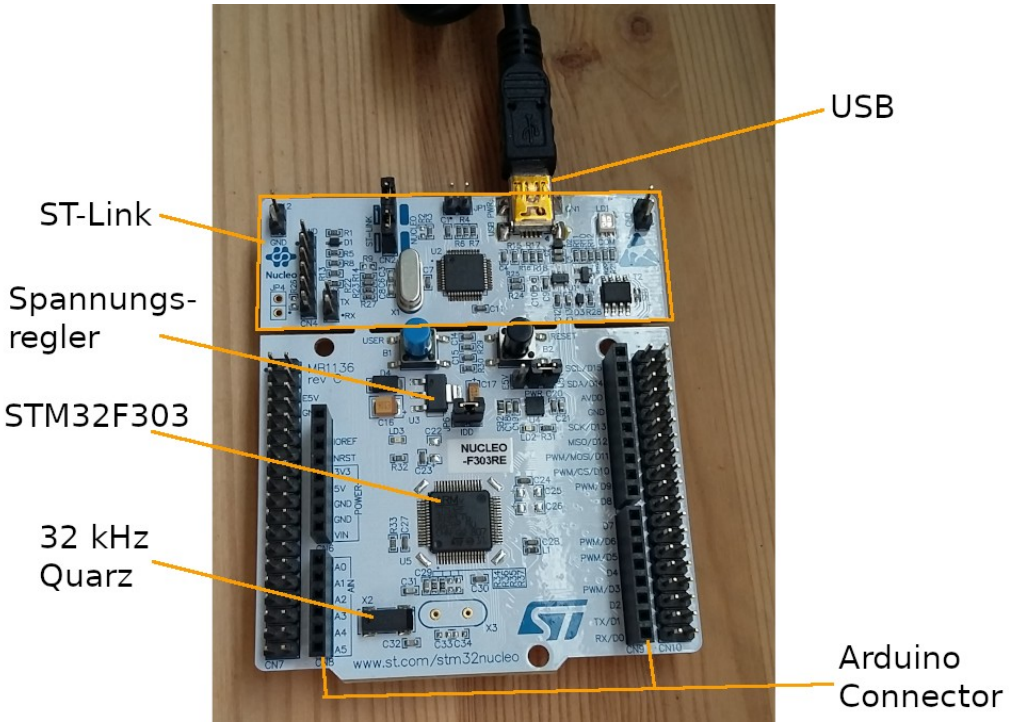



*Abbildung 16: Nucleo-F303RE Board*

Benutze das Board mit Vorsicht und Ruhe. Achte darauf, dass die Platine keinen ungewollten Kontakt mit Metall-Teilen bekommt, um Kurzschlüsse zu vermeiden. Durch die Verbindung zum PC besteht nämlich ein gewisses Risiko, nicht nur das Board sondern auch den USB Anschluss des PC zu zerstören. Theoretisch sollten USB Anschlüsse Kurzschlussfest sein, doch das ist leider nicht immer der Fall.

## 5.1 ST-Link

Der obere Teil heißt „ST-Link“. Das ist ein Programmieradapter, mit dessen Hilfe du deine eigenen Programme in den Mikrocontroller laden wirst und Debuggen (schrittweise ausführen) kannst. Den ST-Link kann man theoretisch vom unteren Teil abtrennen um damit andere Mikrocontroller der STM32 Serie zu programmieren.

Der ST-Link Adapter ist mit der SWJ Schnittstelle des Mikrocontrollers verbunden:

- **SWDIO** = Serial Wire Data (in beide Richtungen, PA13)
- **SWCLK** = Serial Wire Clock (vom ST-Link zum Mikrocontroller, PA14)
- **SWO** = Serial Wire Output (optional zur Ausgabe von Trace Meldungen, PB3)
- **NRST**= Reset-Signal
- **GND** = Ground, Masse

## 5.2 USB-UART

Der ST-Link enthält einen USB-UART Adapter, damit dein PC auf einfache Art mit dem Mikrocontroller kommunizieren kann. Auf dem PC wird dazu ein serieller Port emuliert. Wenn du das Board mit dem PC verbindest, wird der nötige Treiber automatisch geladen. Im Gerätemanager sieht das dann so aus:

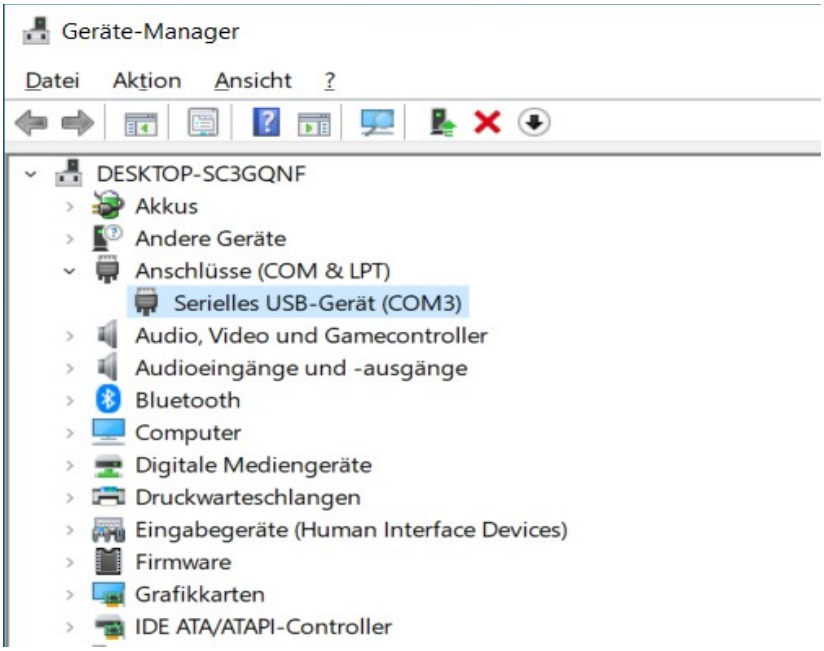



*Abbildung 17: Virtueller COM-Port im Gerätemanager von Windows*

Unter Linux gibt der Befehl „dmesg“ über den Ladevorgang des Treibers Auskunft. Linux nennt den emulierten seriellen Port „/dev/ttyACM0“ oder so ähnlich.

Der USB-UART Adapter ist über drei Leitungen mit dem zweiten seriellen Port des Mikrocontrollers verbunden:

- **TxD** = Ausgang von gesendete Daten → RxD (PA3) am Mikrocontroller
- **RxD** = Eingang für empfangene Daten ← TxD (PA2) am Mikrocontroller
- **GND** = Ground, Masse

Die GND Leitung ist sehr wichtig, weil sie das Bezugspotential für alle Signale führt. Sie wird oft auch als „Masse“ bezeichnet, denn sie ist mit dem massiven Metallgehäuse des Computers verbunden. Auf der TxD Leitung sendet der Mikrocontroller Zeichen an den PC. Das Gegenstück dazu ist die RxD Leitung, auf dieser empfängt der Mikrocontroller Zeichen vom PC.

## 5.3 Mikrocontroller-Teil

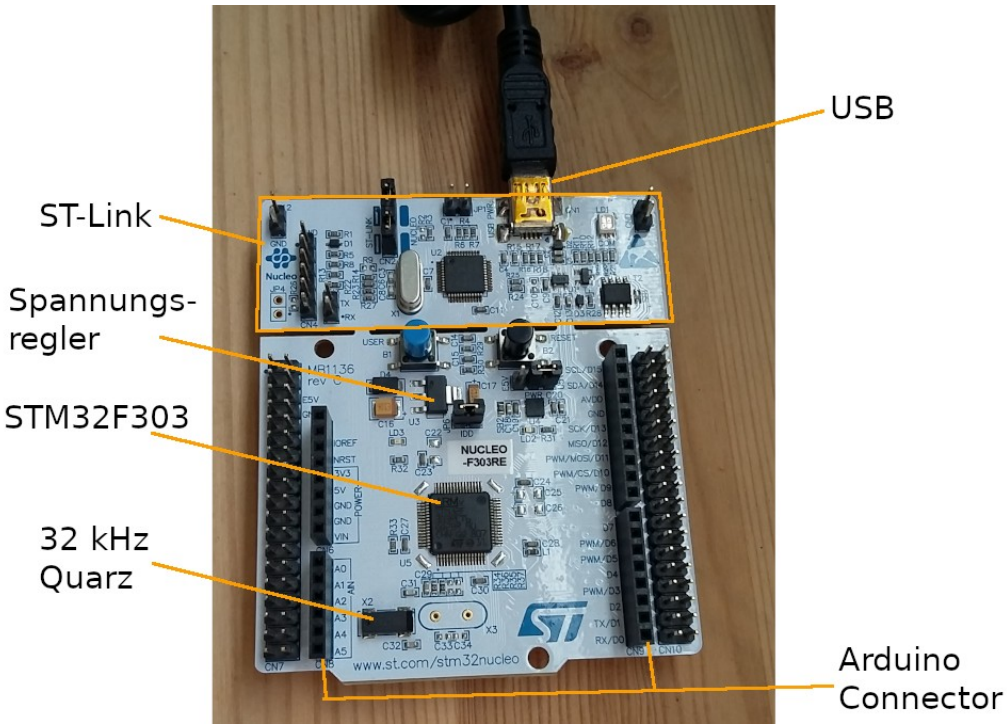



*Abbildung 18: Nucleo-F303RE Board*

Der untere Teil enthält den eigentlichen Mikrocontroller (Target genannt), den du programmieren wirst. Ein Spannungsregler bringt die 5 V vom USB Anschluss auf stabile 3,3 V. Das ist die Versorgungsspannung des Mikrocontrollers STM32F303RE.

Der 32 kHz Quarz (X2) treibt Uhr im Mikrocontroller an. In diesem Buch werden wir die Uhr allerdings nicht benutzen. Der andere Quarz (X3) kann die übrigen Funktionen des Mikrocontrollers antreiben. Er ist ist nicht bestückt, weil der Mikrocontroller bereits einen internen Oszillator hat.

Die äußeren Stiftleisten heißen „Morpho-Connector“. Ihre Pins sind auf der beiliegenden Papp-Karte so beschriftet, wie die entsprechenden Pins des Mikrocontrollers: PA0 bis PC15.

Die schmalen inneren Buchsen-Leisten sind zu einigen Arduino Boards kompatibel und werden daher „Arduino Connector“ genannt. Auf dem Board sind deren Pins mit den Arduino Nummern gekennzeichnet (A0-A5 und D0-D15). Auf der Papp-Karte sind die Pins zusätzlich so beschriftet, wie im Datenblatt des Mikrocontrollers. Dazu möchte ich ergänzen, dass die beiden Arduino Pins RX/D0 und TX/D1 (ganz rechts unten) mit nichts verbunden sind. Insofern finde ich ihre Beschriftung irreführend.

Die Ausgänge des Mikrocontrollers sind einzeln bis zu 25 mA belastbar, alle zusammen jedoch maximal 80 mA.

| |
|---|
| Einige Anschlüsse des Mikrocontrollers vertragen nicht mehr als 3,3 Volt. Ich empfehle dir daher, den „5V“ Ausgang nicht zu benutzen. |

## 5.4 Dokumentation

Die Anleitung samt Schaltplan von dem Board findest du auf der Seite:

- https://www.st.com/resource/en/user_manual/dm00105823.pdf

Die Beschreibung des Mikrocontrollers ist auf vier Dokumente verteilt:

- https://www.st.com/resource/en/datasheet/stm32f303zd.pdf
- https://www.st.com/resource/en/reference_manual/dm00043574-stm32f303xb-c-d-e-stm32f303x6-8-stm32f328x8-stm32f358xc-stm32f398xe-advanced-arm-based-mcus-stmicroelectronics.pdf
- https://www.st.com/resource/en/reference_manual/dm00043574-stm32f303xb-c-d-e-stm32f303x6-8-stm32f328x8-stm32f358xc-stm32f398xe-advanced-arm-based-mcus-stmicroelectronics.pdf
- https://www.st.com/resource/en/programming_manual/dm00046982-stm32-cortex-m4-mcus-and-mpus-programming-manual-stmicroelectronics.pd

# 6 Entwicklungsumgebung

Wir wollen dem kleinen Mikrocontroller bald Befehle erteilen, damit er etwas Sinnvolles für uns tut. Dazu braucht man eine Entwicklungsumgebung (abgekürzt: IDE). Sie besteht im Wesentlichen aus einem Text-Editor, womit man das Programm schreibt, sowie einem Compiler, der das Programm in Maschinencode übersetzt. Dieser Maschinencode wird in den Mikrocontroller übertragen und dann dort ausgeführt. Wir werden die kostenlose „STM32 Cube IDE“ verwenden:

https://www.st.com/en/development-tools/stm32cubeide.html

Du musst dich vor dem Download registrieren. Installiere die IDE und starte sie. Beim ersten mal wirst du aufgefordert, einen Datei-Ordner festzulegen, wo all deine Projekte gespeichert werden sollen. Das soll einfach ein leerer Ordner sein, den du später gut wieder finden kannst. Danach erscheint diese Welcome-Seite:

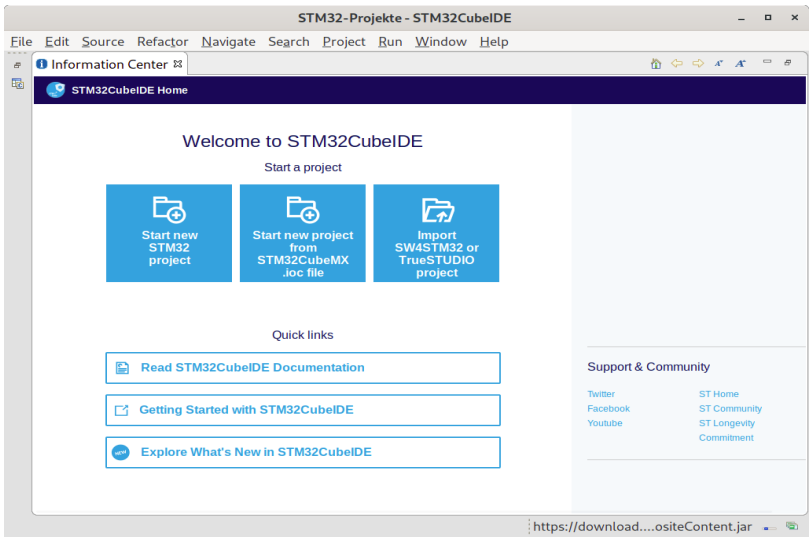



*Abbildung 19: Startbild der STM32 Cube IDE*

Da diese Welcome-Seite ständig umgestaltet wird, benutze ich sie im Rahmen dieses Buches nicht. Die Bildschirmfotos in diesem Buch stammen von der STM32 Cube IDE in Version 1.3.0.

## 6.1 Projekt „Blinker“ erzeugen

Starte die „STM32 Cube IDE“ und wähle dann den Menüpunkt File/New/STM32 Project. Es erscheint der folgende Dialog:

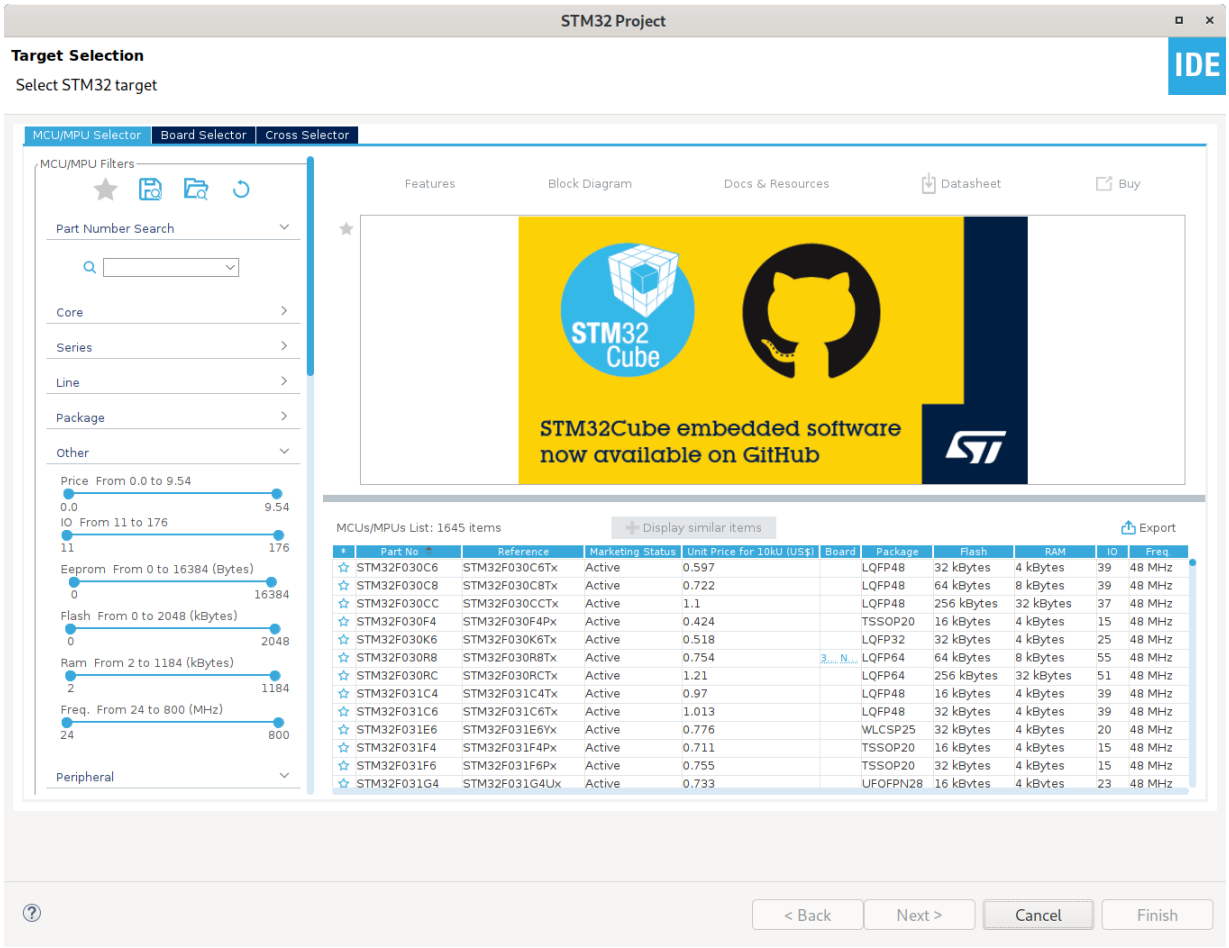



*Abbildung 20: Target Selection Dialog*

Hier musst du den STM32F303RE suchen, auswählen und dann auf „Next“ klicken. Gebe im folgenden Dialog den Projektnamen „Blinker“ ein . Ganz unten bei „Targeted Project Type“ musst du die Option „Empty“ wählen.

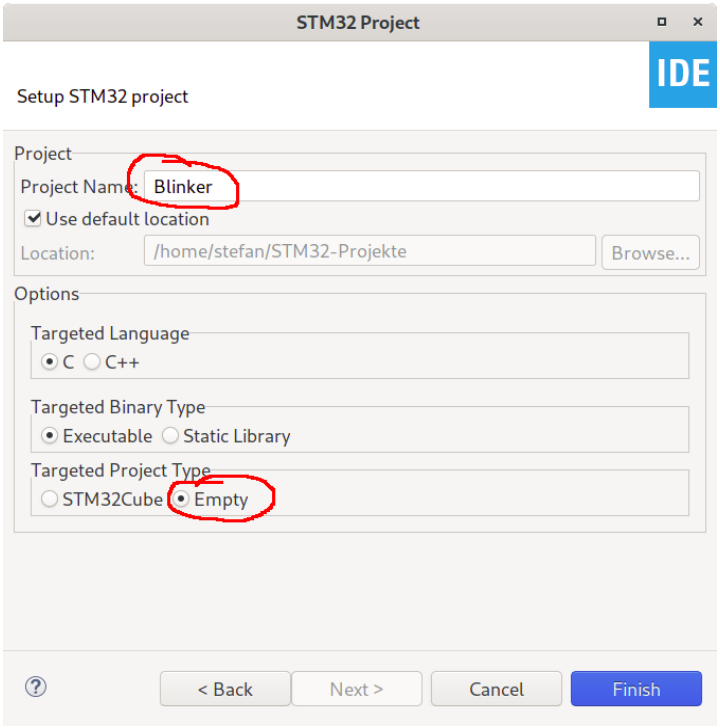



*Abbildung 21: Erste Projekt Einstellungen*

Weiter geht es mit einem Klick auf „Finish“. Das Projekt wird nun erzeugt und geöffnet.

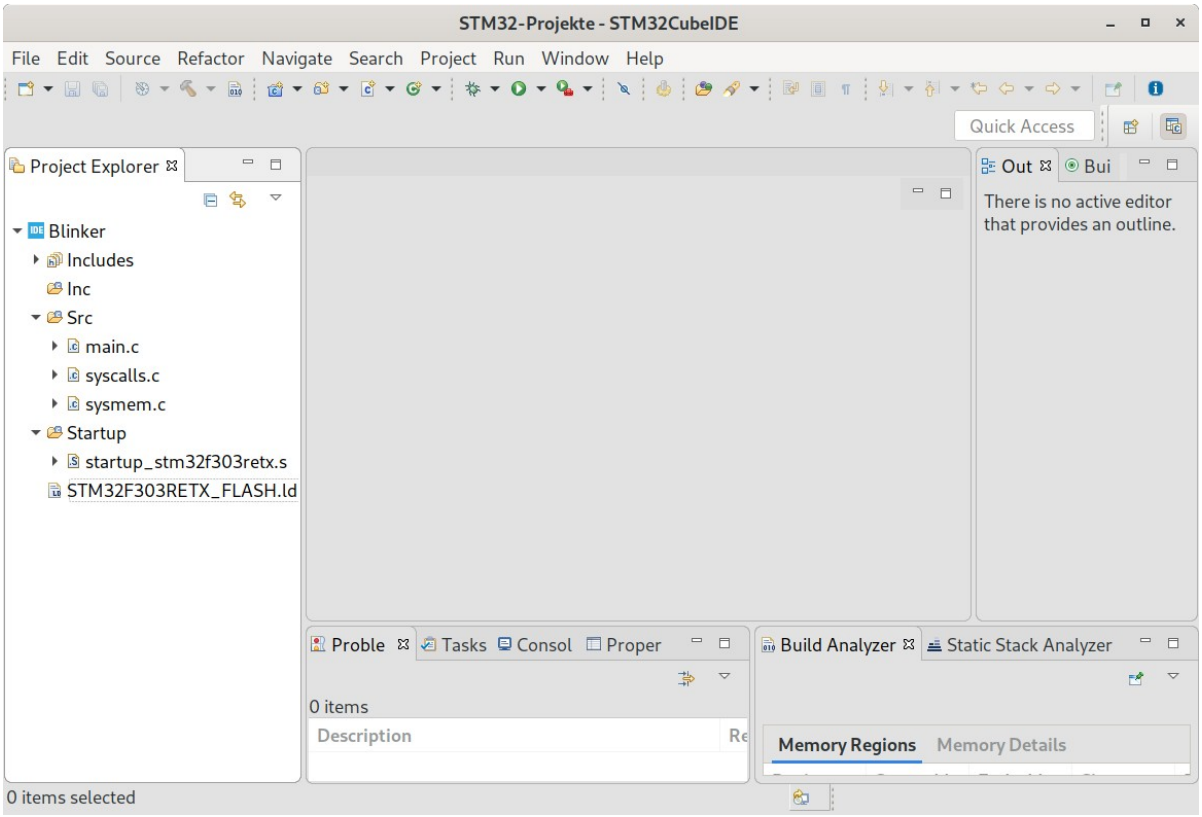



*Abbildung 22: Das Blinker-Projekt wurde erzeugt und geöffnet*

Jetzt ist es notwendig, die CMSIS Bibliotheken zum Projekt hinzuzufügen. Lade das Paket

http://stefanfrings.de/stm32/CMSIS-STM32.zip

von meiner Homepage herunter. Von diesem Paket musst du das core Verzeichnis und das device Verzeichnis für STM32F3 in dein Projektverzeichnis kopieren.

Nun füge diese beiden Verzeichnisse zur Projekt-Konfiguration hinzu:

Markiere dazu den Projektnamen ganz links im „Project Explorer“ und drücke die Taste F5 damit die IDE die neuen Verzeichnisse erkennt.

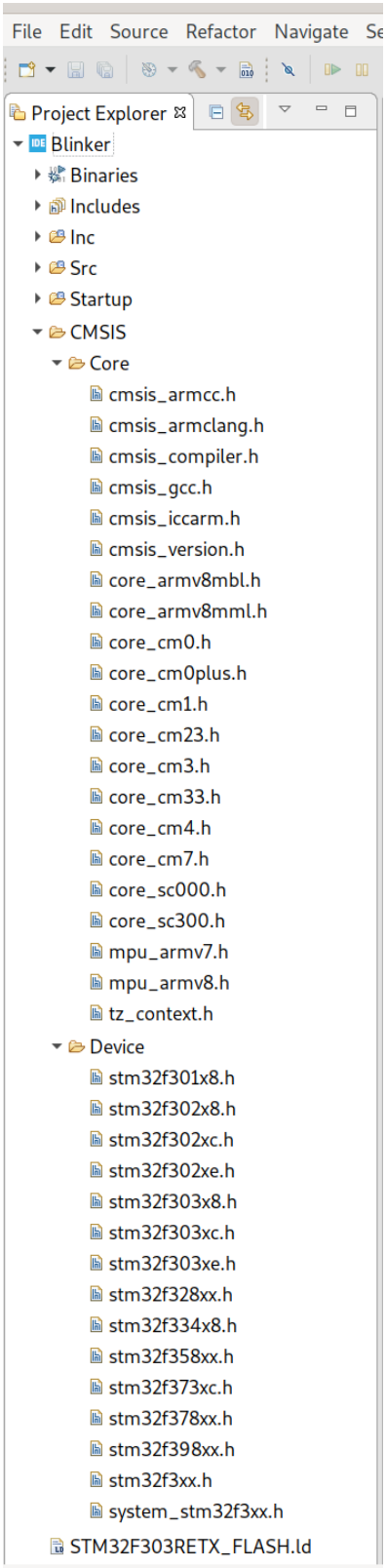



*Abbildung 23: CMSIS Verzeichnisse*

Gehe dann oben auf den Menüpunkt Project/Properties. Ich habe im folgenden Bildschirmfoto Nummern eingefügt, die zeigen, was du anklicken musst:

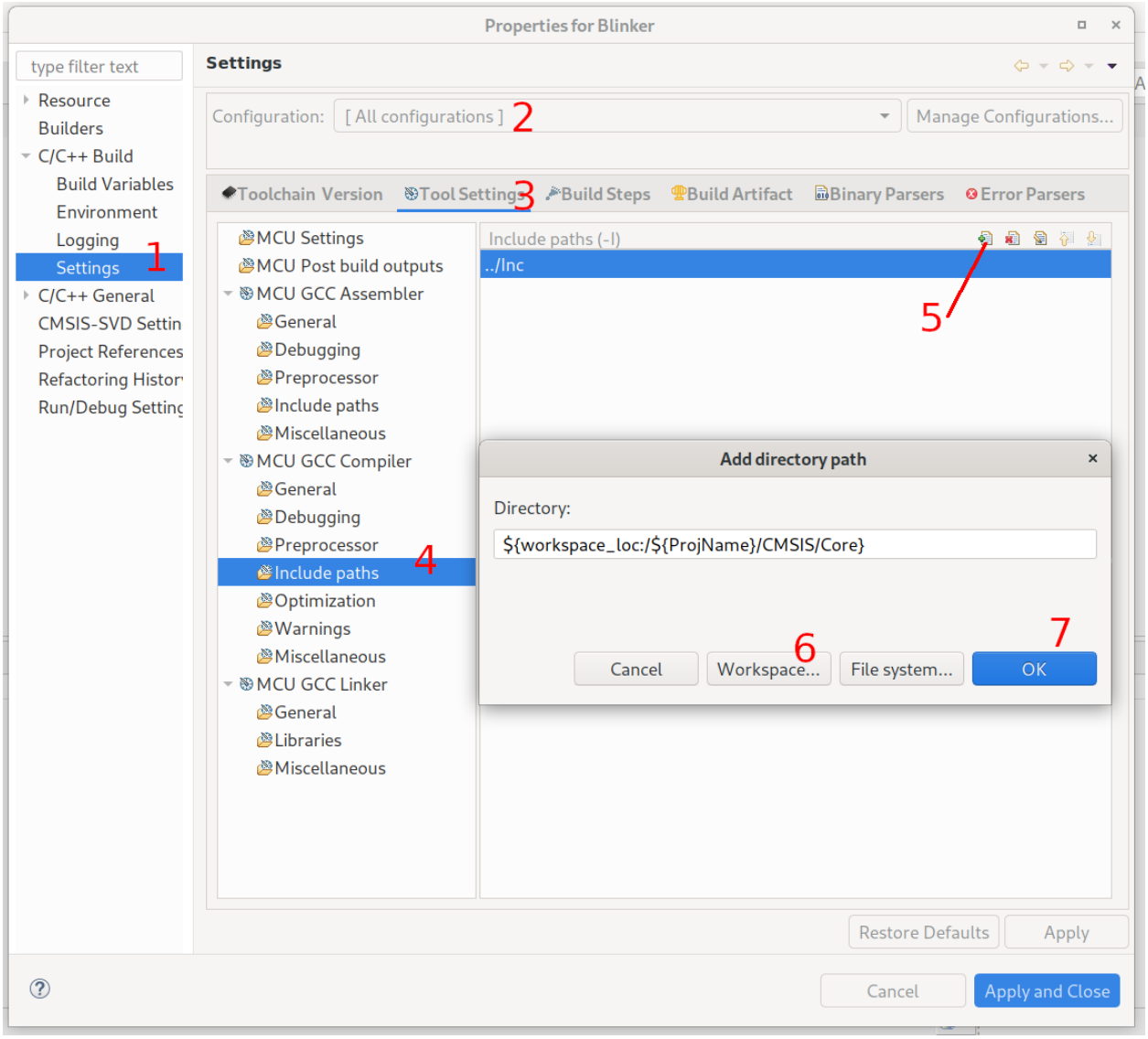



*Abbildung 24: Include Verzeichnis hinzufügen*

Achte darauf, diese Einstellung für „All Configurations“ zu machen, sonst musst du es für jede Konfiguration (Debug und Release) wiederholen.

Außerdem musst du das Device Verzeichnis mit den Include-Dateien für deine Mikrocontroller-Serie hinzufügen:

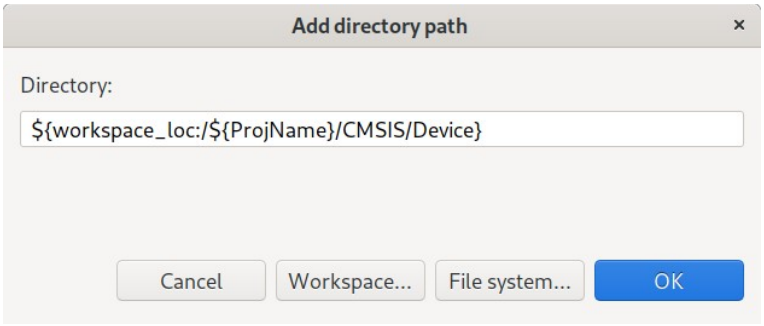



*Abbildung 25: Include Verzeichnis für STM32F3xx*

Klicke danach auf „Apply and Close“ um die Änderungen zu aktivieren. In der Datei STM32F3xx.h musst du nun die richtige Zeile für deinen Mikrocontroller aktivieren, indem du die Kommentar-Zeichen entfernst:

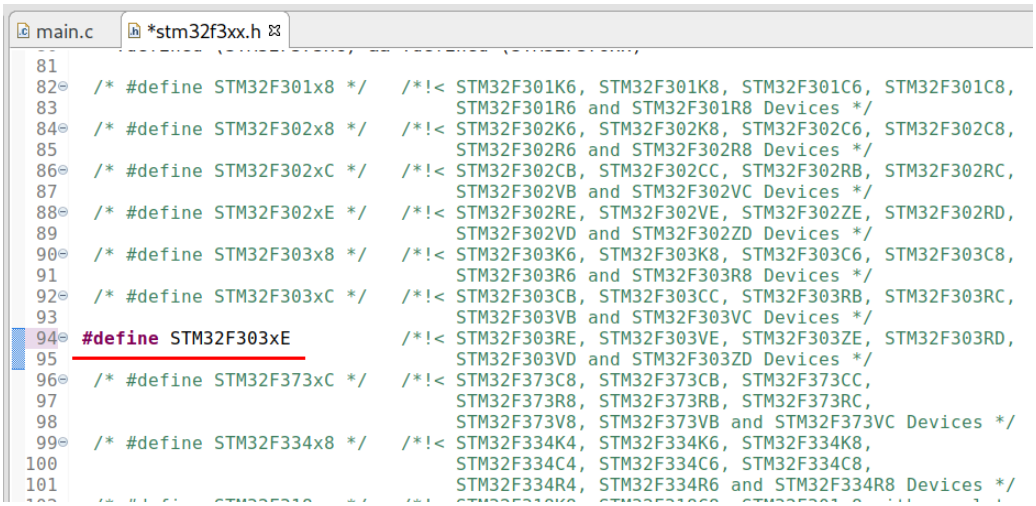



*Abbildung 26: Datei editieren*

Speichere die Änderung an dieser Datei ab.

In der Datei main.c steht möglicherweise ein Hinweis, dass die FPU aktiviert werden soll. Wir kommen darauf später zurück. Ersetze den kompletten Inhalt der Datei main.c durch folgendes Programm:

```
#include <stdint.h>
#include "stm32f3xx.h"

void delay(uint32_t msec)
{
    for (uint32_t j=0; j < msec * 2000; j++)
    {
        __NOP();
    }
}

int main()
{
    SET_BIT(RCC->AHBENR, RCC_AHBENR_GPIOAEN);
                MODIFY_REG(GPIOA->MODER,    GPIO_MODER_MODER5,    0b01    <<
GPIO_MODER_MODER5_Pos);

    while(1)
    {
        WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5);
        delay(500);
        WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5);
        delay(500);
    }
}
```

Dieses Programm kannst du nun durch Klick auf den Hammer compilieren. Die Ausgabe des Compilers müsste dann ungefähr so aussehen:

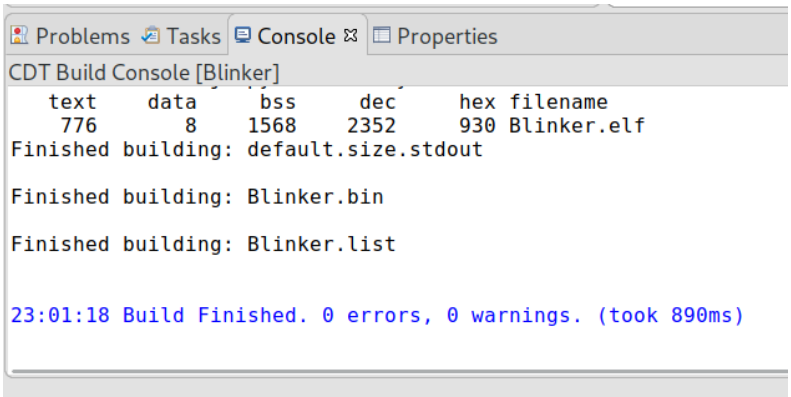



*Abbildung 27: Ausgabe des Compilers*

Das Programm umfasst nur wenige Zeilen Quelltext. Vermutlich hast du noch keinen blassen Schimmer, was diese Zeilen bedeuten, aber das macht nichts. In diesem Kapitel geht es nämlich nur darum, die Bedienung der Software kennen zu lernen. Vertraue fürs Erste darauf, dass dieses Programm funktioniert.

## 6.2 Programm mit dem Debugger starten

Klicke mit der rechten Maustaste auf den Projektnamen und dann auf „Debug As...“, dann auf „STM32 Cortex-M C/C++ Application“. Im folgenden Dialog, der dann erscheint wird angezeigt, dass du nun die Datei „Debug/Blinker.elf“ debuggen wirst. Darin befindet sich der Maschinencode deines Programms und Informationen für den Debugger.

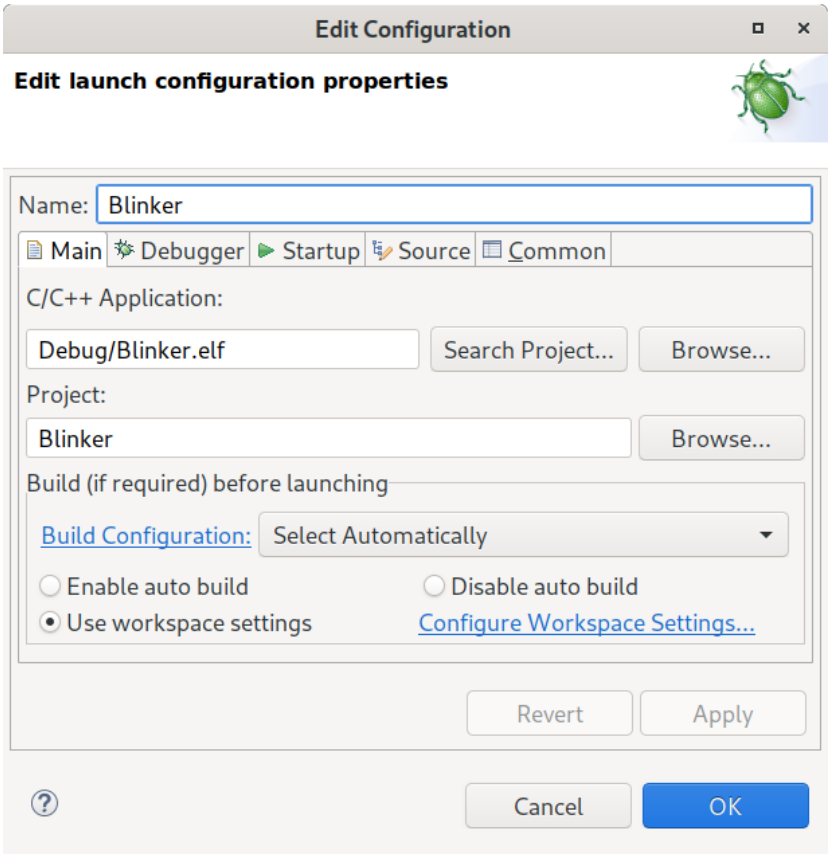



*Abbildung 28: Launch Configuration*

Bestätige dies mit Ok.

Nun startet direkt der Debugger und hält das Programm in der ersten Zeile bei „Enable Port A“ an. Drücke wiederholt die Taste F6 um eine einzelne Zeile auszuführen oder F8, um das Programm von alleine durch laufen zu lassen. Das Programm lässt die grüne LED auf dem Nucleo-Board blinken.

Wenn du möchtest, kannst du nun ein bisschen mit den Funktionen des Debuggers herum spielen. Am rechten Bildschirmrand erscheinen einige Fenster mit interessanten Informationen zum laufendem Programm:

**Variables** zeigt den aktuellen Wert von Variablen an. Dieses Blinker Programm hat allerdings keine Variablen im der main() Funktion.

**Breakpoints** zeigt Unterbrechungspunkte an. Durch Doppelklick auf eine Zeilennummer im Quelltext-Fenster kann man Unterbrechungspunkt hinzufügen. Wenn das Programm nach Druck auf F8 so einen Punkt erreicht, hält es an. Du kannst es dann mit der Taste F8 fortsetzen.

**SFR** zeigt den Inhalt von I/O Registern an. Einige davon wirst du in den folgenden Kapiteln kennen lernen.

Stoppe den Debugger durch Klick auf die rote Schaltfläche oder mit der Tastenkombination Strg-F2.

Der Mikrocontroller kann dein Programm jetzt übrigens ohne PC/Debugger ausführen. Sobald er an ein Netzteil angeschlossen wird, beginnt die LED zu blinken. Du kannst jederzeit den Reset-Knopf auf dem Board drücken, um es neu zu starten.

## 6.3 Release Version

Die „Release“ Version eines Programms ist stärker auf Performance optimiert. Sie ist kleiner und läuft schneller. Dafür funktioniert der Debugger aber nicht mehr. Probiere es aus: Stelle den Modus mit dem Pfeil neben dem Hammer auf „Release“ um. Klicke dann auf den Hammer, um es erneut zu compilieren. Klicke mit der rechten Maustaste auf den Projektnamen und dann auf „Run As...“, dann auf „STM32 Cortex-M C/C++ Application“. Dieses mal erscheint ein anderer Dialog:

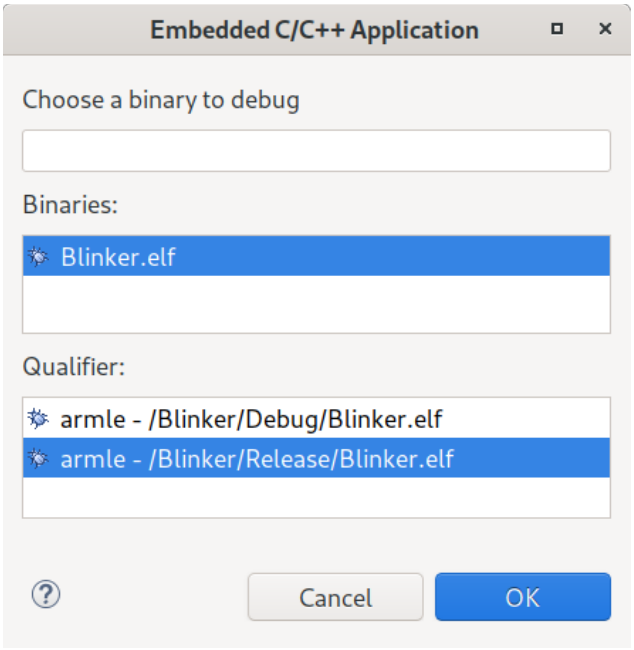



*Abbildung 29: Choose a binary to debug*

Wähle die Datei Blinker.elf aus dem Release Verzeichnis aus und klicke dann auf Ok

Das Programm wird nun auf den Mikrocontroller übertragen uns gestartet. Die LED blinkt jetzt im Sekundentakt, das ist deutlich schneller als vorher.

Mache die Änderung rückgängig, so dass du wieder im Debug Modus arbeitest.

## 6.4 UART Kommunikation

UART ist die Abkürzung von „Universal Asynchronous Receiver and Transmitter“. Selbst geschriebene Programme können diese Schnittstelle benutzen, um Texte an den PC zu senden, die dann dort auf dem Bildschirm erscheinen. Dazu erweitern wir das Blinker-Programm. Ersetze den Inhalt der Datei main.c durch folgendes Programm:

```
#include <stdint.h>
#include <stdio.h>
#include "stm32f3xx.h"

/************************************
 * LED Blinker mit serieller Ausgabe *
 ************************************/
```

```
// Taktfrequenz des Mikrocontrollers
// Diese Variable wird von einigen CMSIS Funktionen benutzt
uint32_t SystemCoreClock=8000000;

// Millisekunden Zähler
volatile uint32_t systick_count=0;

// Interrupt Service Routine
void SysTick_Handler()
{
    systick_count++;
}

// Warte ein paar Millisekunden
void delay_ms(int ms)
{
    uint32_t start=systick_count;
    while (systick_count-start<ms);
}

// Standard Ausgaben auf den seriellen Port leiten
int _write(int file, char *ptr, int len)
{
    for (int i=0; i<len; i++)
    {
        while(!(USART2->ISR & USART_ISR_TXE));
        USART2->TDR = *ptr++;
    }
    return len;
}

// I/O Pins und Funktionen konfigurieren
void init_io()
{
    // Port A einschalten
    SET_BIT(RCC->AHBENR, RCC_AHBENR_GPIOAEN);

    // PA5 = Ausgang für die grüne LED
                MODIFY_REG(GPIOA->MODER,    GPIO_MODER_MODER5,    0b01    <<
GPIO_MODER_MODER5_Pos);

    // USART2 einschalten und Taktquelle einstellen
    SET_BIT(RCC->APB1ENR, RCC_APB1ENR_USART2EN);
                 MODIFY_REG(RCC->CFGR3,    RCC_CFGR3_USART2SW,    0b01    <<
RCC_CFGR3_USART2SW_Pos);

    // PA2 nutzt die alternative Funktion 7: USART2 TxD
    MODIFY_REG(GPIOA->AFR[0], GPIO_AFRL_AFRL2,  7    << GPIO_AFRL_AFRL2_Pos);
                MODIFY_REG(GPIOA->MODER,    GPIO_MODER_MODER2,    0b10    <<
GPIO_MODER_MODER2_Pos);

    // Baudrate auf 9600 einstellen
    USART2->BRR = (SystemCoreClock / 9600);

    // Sender von USART2 einschalten
    USART2->CR1 = USART_CR1_UE + USART_CR1_TE;
}

int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();
```

```
    // Richte den System-Zeitmesser ein
    SysTick_Config(SystemCoreClock/1000);

    while(1)
    {
        // LED Ein
        WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5);

        // Warte eine halbe Sekunde
        delay_ms(500);

        // Text ausgeben
        puts("Hello");

        // LED Aus
        WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5);

        // Warte eine halbe Sekunde
        delay_ms(500);
    }
}
```

Compiliere das Programm und lasse es laufen. Die grüne LED auf dem Board sollte wieder blinken.

Jetzt brauchst du ein Terminal-Programm. Ich empfehle CuteCom (Linux) oder das Hammer Terminal (Linux, Windows). Öffne damit den seriellen Port des ST-Link Adapters, mit den folgenden Settings:

- 9600 Baud (das ist die Übertragungsgeschwindigkeit)
- 8 Bits
- keine Parität
- 1 Stop-Bit

Das Programm zeigt dann in regelmäßigen Intervallen den Text „Hello“ an, weil dein Mikrocontroller ihn genau so an den PC sendet:

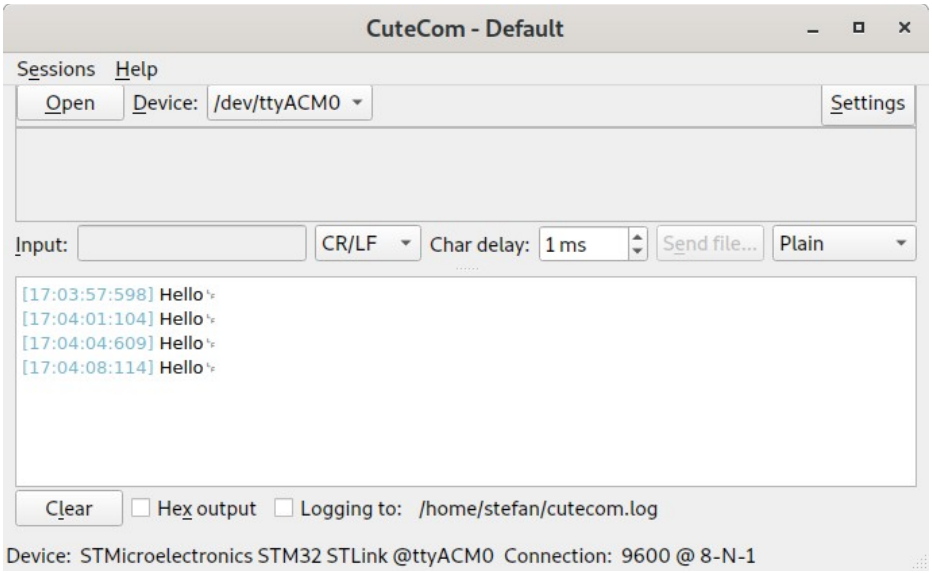



*Abbildung 30: Serielle Ausgabe im Terminalprogramm*

Beim Hammer Terminal musst du eventuell die Option „Newline at: LF“ einstellen, damit die Zeilenumbrüche richtig dargestellt werden. Wenn du den Mikrocontroller im Debugger pausierst oder seine Reset-Taste gedrückt hältst, hören die Ausgaben auf.

Auch dieses mal verzichte ich darauf, das Programm im Detail zu erklären. Momentan sollst du dich nur mit den Arbeitsmitteln vertraut machen. Die gleiche Schnittstelle könnte auch für die umgekehrte Richtung (vom PC zum Mikrocontroller) verwendet werden. Im Rahmen dieses Buches machen wir das allerdings nicht. Wenn du mehr technische Infos zur UART Schnittstelle erfahren möchtest, dann schau in Wikipedia nach.

https://de.wikipedia.org/wiki/Universal_Asynchronous_Receiver_Transmitter

Übungsaufgabe:

Wenn du magst, kannst du als erste Übung mal versuchen, diesen „Hello“ Text oder die Übertragungsgeschwindigkeit (Baudrate) zu ändern.

# 7 Programmieren in C

Du hast nun die Arbeitsmittel kennengelernt und sogar schon ein paar Änderungen an dem Blinker-Beispielprogramm vorgenommen. Die Hardware funktioniert. Damit bist du bereit, die Programmiersprache kennen zu lernen. Wir verwenden die Programmiersprache „C“. Sie ist schon sehr alt, aber immer noch allgemeiner Standard im Mikrocontroller Umfeld.

Das vorherige Blinker Projekt wird von nun an deine Kopiervorlage sein. Jedes neue Projekt, dass du anfängst wird als Kopie dieses Projektes beginnen. Dadurch werden dir eine Menge Einstell-Arbeiten in der IDE erspart.

## 7.1 Projektvorlage kopieren

Links im Bereich „Project Explorer“ werden alle deine Projekte angezeigt. Im Moment ist es nur das eine „Blinker“ Projekt. Nun lege die erste Kopie dieser Vorlage an, indem du mit der rechten Maustaste auf den Projektnamen „Blinker“ klickst, und dann den „Copy“ Befehl wählst. Danach klickst du im „Project Explorer“ mit der rechten Maustaste  in den leeren weißen Bereich und wählst den Befehl „Paste“. Jetzt erstellt die IDE eine Kopie des ganzen Projektes. Dabei wirst du gebeten, einen Namen für das neue Projekt festzulegen:

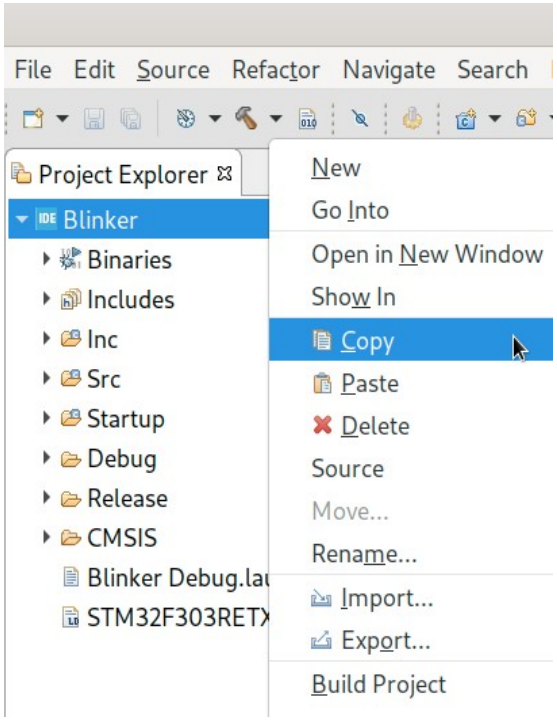



*Abbildung 31: Copy-Befehl im Kontext-Menü des Projektes*

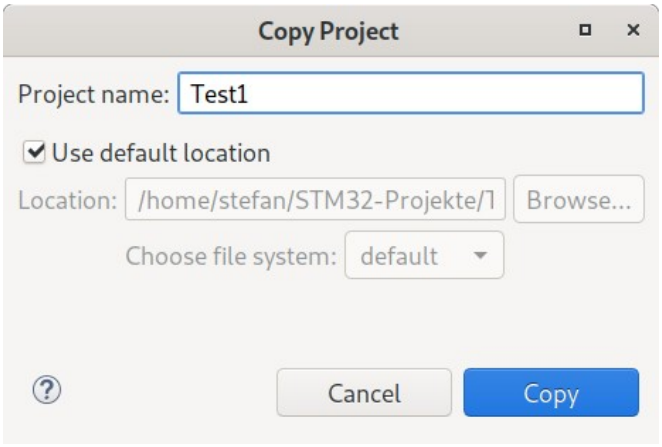



*Abbildung 32: Eingabe des neuen Projektnamens*

Nenne das neue Projekt „Test1“ und bestätige die Eingabe mit Klick auf „Copy“. Klicke dann mit der rechten Maustaste auf den Namen des neuen Projektes und wähle den Befehl „Clean Project“. Dadurch werden alle temporären Dateien gelöscht, die der Compiler später automatisch neu erzeugt.

## 7.2 Projekt-Struktur

Schau dir die Struktur des Projektes an:

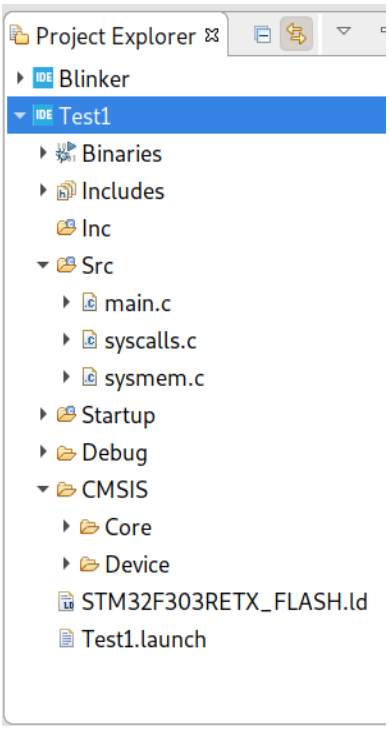



*Abbildung 33: Projekt-Struktur*

„Binaries“ und „Includes“ sind keine echten Projetverzeichnisse, dort werden Dateien angezeigt, die sich woanders befinden.

Das Verzeichnis „inc“ wird leer bleiben. Dort könnte man Header Dateien ablegen, aber wir werden sie wie üblich im „src“ Verzeichnis speichern.

Im „src“ Ordner liegen deine Programm-Quelltexte. Alle drei Dateien wurden automatisch generiert, aber die main.c hast du durch dein eigenes Programm ersetzt. Lasse die beiden Dateien syscalls.c und sysmem.c bitte unverändert. Darin befinden sich hardware-spezifische Implementierungen für Standard-Funktionen, die von der C-Bibliothek benötigt werden.

Im „startup“ Ordner befindet sich die automatisch generierte Dateie startup_stm32f303retx.s. Sie enthält hardware-spezifischen Quelltext in der Sprache Assembler. Der darin befindliche Code wird vor deinem C-Quelltext ausgeführt, um den Arbeitsspeicher so vorzubereiten, wie es die Programmiersprache C erfordert.

In dem Ordner „Debug“ oder „Release“ legt der Compiler temporäre Dateien und sein Ergebnis ab. Diese Ordner darfst du löschen, sie werden ggf. automatisch neu angelegt.

Im CMSIS Ordner befindet sich die CMSIS Bibliothek, in der die ganzen Register der ARM Kerne und die I/O Register der STM32F3 Serie definiert sind.

Die Datei STM32F303RETX_FLASH.ld konfiguriert den Linker, das ist eine Teilfunktion des C-Compilers.

Die Datei Test1.launch konfiguriert den Debugger. Sie wird automatisch angelegt, wenn du die Funktionen „Run As...“ oder „Debug As...“ verwendest.

## 7.3 Programm-Struktur

Der Mikrocontroller beginnt mit der Ausführung des Programms, wenn du das USB Kabel zur Stromversorgung einsteckst. Durch Betätigung des Reset-Knopfes kannst du ihn jederzeit neu starten.

Dein Programm besteht in der Regel aus vielen Abschnitten, die man „Funktionen“ (manchmal auch: Prozeduren) nennt. Die erste Funktion, die ausgeführt wird, heißt „main“ und befindet sich in der Datei „main.c“. Schauen wir uns die main Funktion des Beispielprojektes einmal näher an.

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    // Richte den System-Zeitmesser ein
    SysTick_Config(SystemCoreClock/1000);

    ...
}
```

Die erste Zeile sagt, dass hier eine Funktion mit dem Namen „main“ beginnt, und dass sie ein Ergebnis vom Typ Integer zurück liefert. Standardmäßig ist das die Zahl 0.

Die geschweiften Klammern markieren Anfang und Ende eines sogenannten Blockes. In diesem Fall kennzeichnen sie den Anfang und das Ende der Befehle, die zur main Funktion gehören. Innerhalb eines Blockes rückt man alle Befehle ein bisschen nach rechts ein, um den Quelltext übersichtlich zu gestalten. Wie viele Leerzeichen du verwendest, ist dem Compiler ziemlich egal. Auch die leeren Zeilen interessieren den Compiler nicht.

Zeilen, die mit // beginnen, sind Kommentare. Der Compiler ignoriert sie. Man benutzt Kommentare, um den Quelltext zu erklären. Mehrzeilige Kommentare muss man am Anfang mit /* und am Ende mit */ kennzeichnen. Zum Beispiel:

```
/*
   Hier kommt jetzt die Haupt-Funktion.
   Sie soll die rote LED blinken lassen.
*/
```

Manche Entwickler bevorzugen diese Gestaltungs-Variante:

```
/*
 * Hier kommt jetzt die Haupt-Funktion.
 * Sie soll die rote LED blinken lassen.
 */
```

Oder noch auffälliger:

```
/**************************************
 * Hier kommt jetzt die Haupt-Funktion.  *
 * Sie soll die rote LED blinken lassen. *
 **************************************/
```

Für den Compiler haben Kommentare wie gesagt keine Bedeutung. Die Entwicklungsumgebung hebt sie daher mit einer anderen Farbe (hellgrün) hervor.

Wenn du in der Datei „main.c“ jetzt mal nach oben scrollst, siehst du noch weitere Funktionen. Anfang und Ende jeder Funktion ist mit einer geschweiften Klammer gekennzeichnet.

Ganz oben kommt noch etwas anderes:

```
#include <stdint.h>
#include <stdio.h>
#include "stm32f3xx.h"
```

Die „#include“ Anweisungen sagen dem Compiler, dass er zusätzlich zu dieser vorliegenden Datei auch noch die dort genannten Dateien als Bestandteil des Quelltextes betrachten soll. Dateien, die zum Lieferumfang des Compilers gehören, werden in spitze Klammern <...> eingeschlossen. Dateien, die im Projektverzeichnis liegen, werden in doppelte Anführungsstriche (oben) eingeschlossen.Darunter werden zwei Variablen definiert:

```
// Taktfrequenz des Mikrocontrollers
// Diese Variable wird von einigen CMSIS Funktionen benutzt
uint32_t SystemCoreClock=8000000;

// Millisekunden Zähler
volatile uint32_t systick_count=0;
```

Die erste Variable hat den Namen „SystemCoreClock“ und soll zu Beginn den Zahlenwert 8000000 enthalten. Die zweite heißt „systick_count“ und soll zu Beginn den Wert 0 enthalten. Variablen reservieren ein Stück vom Arbeitsspeicher, um dort irgendwelche Daten abzulegen (in diesem Fall: Zahlen).

Ich möchte noch einmal den Blick auf die main Funktion lenken. Scrolle wieder herunter und schau Sie dir an.

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    // Richte den System-Zeitmesser ein
    SysTick_Config(SystemCoreClock/1000);

    while(1)
    {
        // LED Ein
        WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5);

        // Warte eine halbe Sekunde
        delay_ms(500);

        // Text ausgeben
        puts("Hello");

        // LED Aus
        WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5);

        // Warte eine halbe Sekunde
        delay_ms(500);
    }
}
```

Die Kommentare erklären ganz grob, was die einzelnen Befehle tun. Der erste Befehl lautet:

```
init_io();
```

Am Namen erkenne ich, dass hier eine Funktion aufgerufen wird. Diese findest du etwas weiter oben im Quelltext wieder. Dass bedeutet, dass dein Hauptprogramm (main) damit beginnt, die Befehle auszuführen, die in der Funktion „init_io“ stehen. Der zweite Befehl lautet:

```
SysTick_Config(SystemCoreClock/1000);
```

Das ist auch wieder ein Funktionsaufruf. Der entsprechende Quelltext befindet sich in irgendeiner CMSIS Datei. Du kannst mit gedrückter Strg Taste auf den Funktionsnamen klicken, um die Datei zu öffnen. Funktionen können Parameter erfordern die man zwischen die runden Klammern (…) schreibt. Was die Parameter genau bewirken, müsste man in der Dokumentation der jeweiligen Funktion nachlesen.

Das soll jetzt erst einmal genügen. Lass und etwas programmieren.

## 7.4 Dein erstes Programm

Dein erstes Programm soll die programmierbare LED einschalten und einmal den Text „Hallo!“ an den Computer senden. Die Blinker Vorlage hast du bereits in ein neues Projekt mit dem Namen „Test1“ kopiert. Ändere nun in diesem Projekt die „main“ Funktion so ab, dass sie nur noch diese drei Befehle enthält:

```
int main()
{
    init_io();
    WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS_5);
    puts("Hallo!");
}
```

Compiliere das Programm indem du auf den Hammer klickst. Wenn du keinen Fehler gemacht hast, müssten die Meldungen im „Console“ View ungefähr so aussehen:

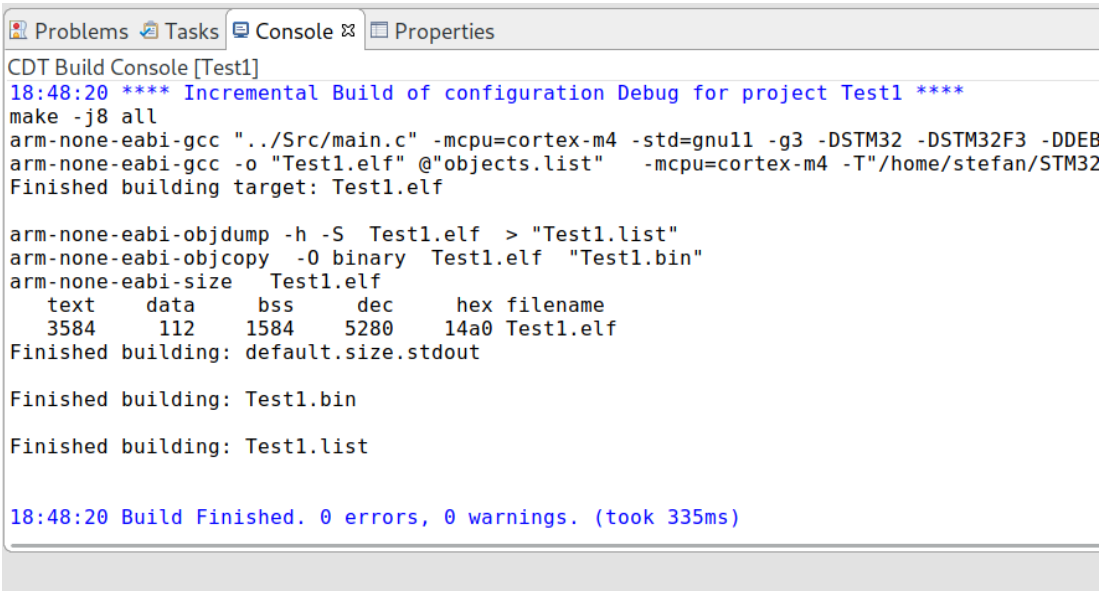



*Abbildung 34: Erfolgreich compiliert*

Falls der Compiler jedoch Fehlermeldungen anzeigt, die dir unklar sind, dann suche mit Google nach dem Text der Meldung (ohne Zahlen und ohne Dateinamen). So finde ich meistens am schnellsten eine Erklärung. Selbst für Leute mit sehr guten Englisch Kenntnissen sind die Fehlermeldungen des Compilers oft sehr kryptisch. Lass dich dadurch nicht entmutigen.

Übertrage das Programm wie zuvor geübt in den Mikrocontroller und starte es. Dieses mal leuchtet die grüne LED dauerhaft. Drücke jetzt den Reset-Knopf auf dem Mikrocontroller Board. Die LED geht kurz aus und dann wieder an. Im Terminal Programm erscheint der Text „Hallo!“. Drücke den Reset-Knopf erneut, so dass dein Programm nochmal ausgeführt wird. Beobachte dabei die Ausgabe im Terminal Programm.

Du hast jetzt dein erstes Programm auf Basis der Kopiervorlage vollbracht. Es schaltet die LED ein und sendet einen Text an den Computer. Jedes mal, wenn du den Reset Knopf drückst, wird das Programm erneut ausgeführt.

Ich werde dir nun die drei Befehle erklären, aus denen dein Hauptprogramm besteht:

```
    init_io();
```

Diese Funktion initialisiert die Anschlüsse des Mikrocontrollers. Ein Blick in den Quelltext der Funktion verrät, was da genau passiert:

- **Port A einschalten**
  Wenn du dir die Papp-Karte zum Mikrocontroller Board anschaust, siehst du ganz viele Nummern, die mit PA anfangen. Zusammen gehören sie zum Port A, der hiermit eingeschaltet wird. Erst danach kann man die Pins, die mit PA anfangen, benutzen.
- **PA5 = Ausgang für die grüne LED**
  Hier wird der Anschluss PA5 so konfiguriert, dass er ein Ausgang ist. Damit wird die grüne LED angesteuert, denn sie ist mit diesem Anschluss verbunden.
- **USART2 einschalten und Taktquelle einstellen**
  Hier wird der zweite serielle Port des Mikrocontrollers eingeschaltet. Außerdem wird festgelegt, dass er mit dem Systemtakt angetrieben werden soll.
- **PA2 nutzt die alternative Funktion 7: USART2 TxD**
  Jetzt muss der Anschluss PA2 mit dem Ausgang des seriellen Ports verbunden werden. Das ist bei fast allen Pins nötig, die nicht für einfache digitale Ausgabe (wie bei der LED)

verwendet werden. Im Datenblatt des Mikrochips gibt es dazu eine Tabelle mit dem Titel „alternate function mapping“. Dort findest du die Information, dass bei PA2 die alternative Funktion 7 benötigt wird. https://www.st.com/resource/en/datasheet/stm32f303zd.pdf

- **Baudrate auf 9600 einstellen**
  Hier wird die Übertragungsgeschwindigkeit der seriellen Schnittstelle eingestellt. 9600 Baud entsprechen bis zu 960 Zeichen pro Sekunde
- **Sender von USART2 einschalten**
  Hier wird der Sender vom zweiten seriellen Port eingeschaltet. Ab jetzt kann er Daten an den PC senden.

Weiter geht es in der Hauptfunktion main():

```
    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5);
```

Hier wird der Anschluss PA5 auf High gesetzt, so dass die LED an geht.

- BS_5 setzt den Pin 5 auf 3,3 Volt
- BR_5 würde den Pin 5 wieder auf 0 Volt zurück setzen.

```
    puts("Hallo");
```

Die Funktion „puts“ sendet den angegebenen Text. Die Zeichenkette „Hallo“ ist der Parameter, mit dem die Funktion puts aufgerufen wird.

Übungsaufgabe:

Ändere die Baudrate auf 19200 und sende mehrere lange Texte. Danach soll die LED wieder ausgeschaltet werden. Führe das Programm mehrmals aus und kontrolliere dessen Ausgabe im Hyper Terminal. Kontrolliere, ob die LED nach dem Senden der Texte wieder aus geht.

## 7.5 Register

Alle Funktionen des Mikrocontrollers werden durch die sogenannten „Register“ gesteuert. Du kannst dir ein Register als eine lange Reihe von Tastern oder Schaltern vorstellen, mit denen man irgend etwas ein und aus schaltet. Das „STM32F3 Reference Manual“ beschreibt alle Register der kompletten Mikrocontroller Serie im Detail:

https://www.st.com/resource/en/reference_manual/dm00043574.pdf

(Das Referenzhandbuch musst du jetzt nicht komplett durch lesen)

Ein Register hast du bereits mehrfach benutzt, und zwar das Register „GPIOA->BSRR“. Dieses Register enthält 32 „Taster“, mit denen man die Ausgänge von Port C kontrollieren kann. In der Computersprache nennt man diese 32 Dinger jedoch „Bits“. Im Referenzhandbuch werden die 32 Bits so dargestellt:

### 11.4.7 GPIO port bit set/reset register (GPIOx_BSRR) (x = A..H)

Address offset: 0x18

Reset value: 0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| BR15 | BR14 | BR13 | BR12 | BR11 | BR10 | BR9 | BR8 | BR7 | BR6 | BR5 | BR4 | BR3 | BR2 | BR1 | BR0 |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| BS15 | BS14 | BS13 | BS12 | BS11 | BS10 | BS9 | BS8 | BS7 | BS6 | BS5 | BS4 | BS3 | BS2 | BS1 | BS0 |
| w | w | w | w | w | w | w | w | w | w | w | w | w | w | w | w |

Bits 31:16 **BRy:** Port x reset bit y (y = 0..15)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Resets the corresponding ODRx bit
*Note: If both BSx and BRx are set, BSx has priority.*

Bits 15:0 **BSy:** Port x set bit y (y= 0..15)
These bits are write-only. A read to these bits returns the value 0x0000.
0: No action on the corresponding ODRx bit
1: Sets the corresponding ODRx bit

*Abbildung 35: Beschreibung des BSRR Registers*

Die Kästchen stellen die 32 Bits mit ihren abgekürzten Namen dar. Die Angabe „w“ bedeutet „writeable“, auf deutsch: Beschreibbar. Dass soll bedeuten, dass dein Programm sie verändern kann.

Wenn dein Programm ein „BR“ Bit beschreibt, wird der entsprechende Ausgang des Mikrocontrollers auf 0 Volt geschaltet. Man sagt auch „Low Pegel“ oder „Low Level“.

Wenn dein Programm ein „BS“ Bit beschreibt, wird der entsprechende Ausgang des Mikrocontrollers auf 3,3 Volt geschaltet. Das nennt man „High Pegel“ oder „High Level“.

Die grüne Leuchtdiode des Mikrocontroller Boards ist mit dem Anschluss PA5 verbunden.

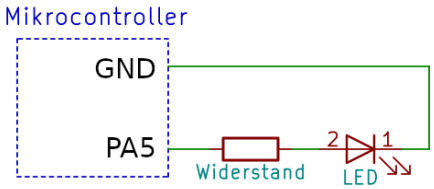



*Abbildung 36: LED an Port PA5*

Damit sie von Strom durchflossen werden kann, muss

1. Der Port A eingeschaltet werden
2. Der Anschluss PA5 als Ausgang konfiguriert werden.
3. Der Ausgang PA5 auf 3,3 Volt (= High) gesetzt werden

Wenn der Ausgang hingegen 0 Volt (= Low) liefert, bleibt die Leuchtdiode dunkel, denn es fließt kein Strom.

Der Befehl zum Einschalten der LED lautet:

```
    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5);
```

Mit ein bisschen Englisch kannst du dir einen Reim darauf machen, was dieser Befehl bedeutet. Ich versuche das mal auf deutsch zu formulieren: „Beschreibe das Register BSRR vom Port A, und zwar mit dem Wert für das Bit BS 5“.

Der Befehl zum Ausschalten der LED lautet:

```
    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5);
```

Der Unterschied zwischen BS und BR ist:

- BS 5 setzt den Ausgang 5 auf 3,3 Volt (High)
- BR 5 setzt den Ausgang 5 zurück auf 0 Volt (Low)

Das gleiche Register gibt es logischerweise auch für Port B und Port C.

Lass uns noch ein paar weitere Leuchtdioden hinzufügen. Die rote LED kommt an PA0 und die blaue LED kommt an PA1:

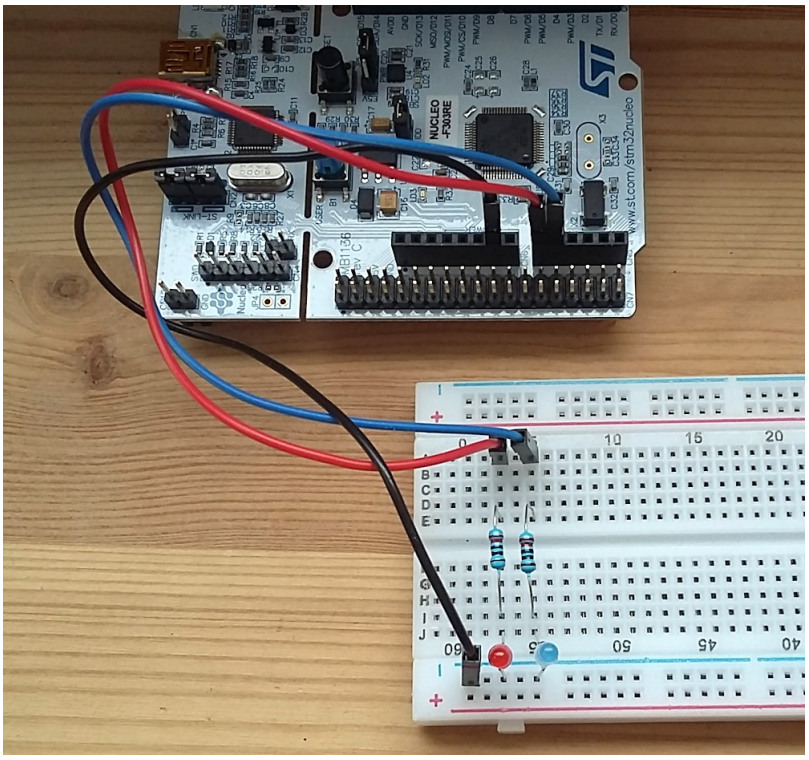


*Abbildung 37: Anschluss von zwei LEDs an PA0 und PA1*

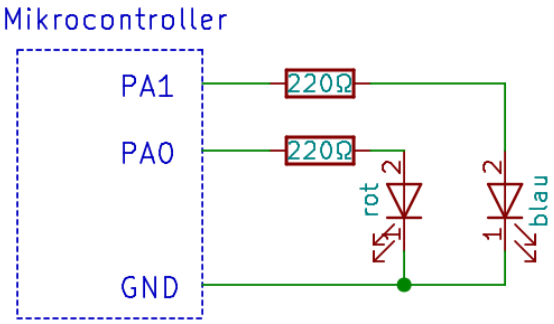



*Abbildung 38: Anschluss von zwei LEDs an PA0 und PA1*

| |
|---|
| Immer das USB Kabel ab stecken bevor du deinen Aufbau änderst! Dadurch vermeidest du Kurzschlüsse, die deine Bauteile zerstören würden. Benutze einen USB Hub, damit die Anschlüsse deines PC durch die vielen Steckvorgänge nicht frühzeitig verschleißen. |

Jetzt ändern wir das Programm „Test1“ so, dass diese beiden hinzugefügten LEDs leuchten. In der „init_io“ Funktion müssen nun alle drei I/O Pins von Port A als Ausgang konfiguriert werden:

```
void init_io()
{
    // Port A einschalten
    SET_BIT(RCC->AHBENR, RCC_AHBENR_GPIOAEN);

    // PA5 = Ausgang für die grüne LED
                MODIFY_REG(GPIOA->MODER,    GPIO_MODER_MODER5,    0b01    <<
GPIO_MODER_MODER5_Pos);

    // PA0 = Ausgang für die rote LED
                MODIFY_REG(GPIOA->MODER,    GPIO_MODER_MODER0,    0b01    <<
GPIO_MODER_MODER0_Pos);

    // PA1 = Ausgang für die blaue LED
                MODIFY_REG(GPIOA->MODER,    GPIO_MODER_MODER1,    0b01    <<
GPIO_MODER_MODER1_Pos);

    ...
}
```

In der „main“ Funktion werden alle drei Leuchtdioden eingeschaltet :

```
int main()
{
    init_io();
    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5);
    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_0);
    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_1);
}
```

Die entscheidenden Stellen habe ich passend zu den Farben der LEDs markiert.

Compiliere das Programm und übertrage es dann auf den Mikrocontroller, um es auszuprobieren. Nach dem Start müssen alle drei Leuchtdioden leuchten. Wenn du es im Debugger mit der taste F6 Zeilenweise ausführst, kannst du schön sehen, wie die drei Befehle in der „main“ Funktion nacheinander deine drei LEDs einschalten. Schauen wir uns den modify Befehl genauer an:

```
MODIFY_REG(GPIOA->MODER, GPIO_MODER_MODER5, 0b01 << GPIO_MODER_MODER5_Pos);
```

Dies bedeutet: Modifiziere im Register GPIOA->MODER das Feld MODER5. Stelle die beiden Bits des Feldes auf den binären Wert 01 ein („0b“ ist das Präfix für binäre Zahlen.). Alle anderen Felder bleiben unverändert. Um zu verstehen, was das bewirkt, schauen wir uns die Beschreibung des MODER Registers im Referenzhandbuch an:

### 11.4.1 GPIO port mode register (GPIOx_MODER) (x =A..H)

Address offset:0x00

Reset values:

- 0xA800 0000 for port A
- 0x0000 0280 for port B
- 0x0000 0000 for other ports

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| MODER15[1:0] | | MODER14[1:0] | | MODER13[1:0] | | MODER12[1:0] | | MODER11[1:0] | | MODER10[1:0] | | MODER9[1:0] | | MODER8[1:0] | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| MODER7[1:0] | | MODER6[1:0] | | MODER5[1:0] | | MODER4[1:0] | | MODER3[1:0] | | MODER2[1:0] | | MODER1[1:0] | | MODER0[1:0] | |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

Bits 2y+1:2y **MODERy[1:0]:** Port x configuration bits (y = 0..15)
These bits are written by software to configure the I/O mode.
00: Input mode (reset state)
01: General purpose output mode
10: Alternate function mode
11: Analog mode

*Abbildung 39: Beschreibung des MODER Registers*

Die relevanten Felder habe ich hier bunt markiert. Alle drei Felder umfassen jeweils zwei Bits, die folgendes bewirken. Ich übersetze mal die Beschreibung:

00 = Eingang (Reset-Zustand)

01 = Allgemeiner Ausgang

10 = Alternative Funktion

11 = Analoger Modus

Im Rahmen dieses Buches nutzen wir nur eine einzige alternative Funktion, und zwar den Ausgang der seriellen Schnittstelle an PA2. Den analogen Modus wirst du auch noch kennen lernen.

Ich hatte beim Programmieren anfangs Schwierigkeiten, die Namen der Register und Bits richtig zu schreiben. Denn die Schreibweise in der Programmiersprache stimmt teilweise nicht exakt mit der Schreibweise im Referenzhandbuch überein. In diesem Fall hilft ein Blick in die CMSIS Datei „stm32f303xe.h“. Dort sind die Namen alle aufgelistet, schön nach Register Gruppiert.

Übungsaufgabe:

Schließe die gelbe LED an PA8 (= Arduino Connector D7) an und lasse sie aufleuchten. Wenn das geklappt hat, versuche die weiße LED an PC0 (= Arduino Connector A5) ans Leuchten zu bringen. Vergiss dabei nicht, den Port C ein zu schalten.

## 7.6 Textersetzungen

Wir haben einige male LEDs ein oder aus geschaltet und dazu ziemlich kryptische Befehle verwendet. Wäre es nicht praktisch, wenn man stattdessen einfach „LED_ROT_AN“ oder „LED_ROT_AUS“ schreiben könnte? In der Tat ist das machbar. Und zwar, indem du Textersetzungen definierst. Auf Englisch nennt man das „Makro“. Tausche die alte Funktion „main“ durch diesen Text aus:

```
#define LED_GRUEN_AN   WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5)
#define LED_ROT_AN     WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_0)
#define LED_BLAU_AN    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_1)

#define LED_GRUEN_AUS  WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5)
#define LED_ROT_AUS    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_0)
#define LED_BLAU_AUS   WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_1)

int main()
{
    init_io();

    LED_GRUEN_AN;
    LED_ROT_AN;
    LED_BLAU_AN;
}
```

Auf diese Weise wird der Quelltext deutlich besser lesbar, besonders wenn die gleiche Zeile an vielen Stellen vorkommt. Auch die Wartbarkeit des Quelltextes verbessert sich. Stelle dir vor, eine LED soll einem anderen Pin zugeordnet werden. Dann brauchst du nur doch diese „#define“ Zeilen anzupassen, anstatt zahlreiche Stellen im Quelltext.

## 7.7 Quelltext aufteilen

Ernsthafte C Programme sind oft sehr viel größer als dein „Test1“ Projekt. Um die Übersicht besser behalten zu können, soll man große Quelltexte auf mehrere Dateien aufteilen.Dein Programm ist inzwischen groß genug, um dies zu verdeutlichen. In diesem Kapitel werden wir dein Testprojekt zerlegen, aber natürlich so, dass es trotzdem noch funktioniert. Klicke dazu in der Entwicklungsumgebung mit der rechten Maustaste auf den Ordner „src“ und wähle den Befehl „New / File“. Die Datei soll „init.c“ genannt werden.

Benutze die Zwischenablage, um Teile von der Datei „main.c“ in die neue Datei „init.c“ zu verschieben. Falls du die Tastenkombinationen nicht kennst:

- Strg-C kopiert den markierten Text in die Zwischenablage
- Strg-X schneidet den markierten Text aus und legt ihn in die Zwischenablage.
- Strg-V fügt den Inhalt der Zwischenablage an der Stelle des Cursors ein.

Die neue Datei „init.c“ soll folgenden Inhalt haben:

```
#include <stdint.h>
#include "stm32f3xx.h"

/************************************
 * LED Blinker mit serieller Ausgabe *
 ************************************/

// Taktfrequenz des Mikrocontrollers
// Diese Variable wird von einigen CMSIS Funktionen benutzt
uint32_t SystemCoreClock=8000000;

// Millisekunden Zähler
volatile uint32_t systick_count=0;

// Interrupt Service Routine
void SysTick_Handler()
{
    systick_count++;
}
```

```
// Warte ein paar Millisekunden
void delay_ms(uint32_t ms)
{
    uint32_t start=systick_count;
    while (systick_count-start<ms);
}

// Standard Ausgaben auf den seriellen Port leiten
int _write(int file, char *ptr, int len)
{
    for (int i=0; i<len; i++)
    {
        while(!(USART2->ISR & USART_ISR_TXE));
        USART2->TDR = *ptr++;
    }
    return len;
}

// I/O Pins und Funktionen konfigurieren
void init_io()
{
    // Port A einschalten
    SET_BIT(RCC->AHBENR, RCC_AHBENR_GPIOAEN);

    // PA5 = Ausgang für die grüne LED
                MODIFY_REG(GPIOA->MODER,    GPIO_MODER_MODER5,    0b01    <<
GPIO_MODER_MODER5_Pos);

    // PA0 = Ausgang für die rote LED
                MODIFY_REG(GPIOA->MODER,    GPIO_MODER_MODER0,    0b01    <<
GPIO_MODER_MODER0_Pos);

    // PA1 = Ausgang für die blaue LED
                MODIFY_REG(GPIOA->MODER,    GPIO_MODER_MODER1,    0b01    <<
GPIO_MODER_MODER1_Pos);

    // USART2 einschalten und Taktquelle einstellen
    SET_BIT(RCC->APB1ENR, RCC_APB1ENR_USART2EN);
                 MODIFY_REG(RCC->CFGR3,    RCC_CFGR3_USART2SW,    0b01    <<
RCC_CFGR3_USART2SW_Pos);

    // PA2 nutzt die alternative Funktion 7: USART2 TxD
    MODIFY_REG(GPIOA->AFR[0], GPIO_AFRL_AFRL2,  7    << GPIO_AFRL_AFRL2_Pos);
                MODIFY_REG(GPIOA->MODER,    GPIO_MODER_MODER2,    0b10    <<
GPIO_MODER_MODER2_Pos);

    // Baudrate auf 9600 einstellen
    USART2->BRR = (SystemCoreClock / 9600);

    // Sender von USART2 einschalten
    USART2->CR1 = USART_CR1_UE + USART_CR1_TE;
}
```

Und die Datei „main.c“ soll auf folgende Zeilen reduziert werden:

```
#include <stdint.h>
#include "stm32f3xx.h"

#define LED_GRUEN_AN   WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5)
#define LED_ROT_AN     WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_0)
#define LED_BLAU_AN    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_1)

#define LED_GRUEN_AUS  WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5)
```

```
#define LED_ROT_AUS    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_0)
#define LED_BLAU_AUS   WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_1)

int main()
{
    init_io();

    LED_GRUEN_AN;
    LED_ROT_AN;
    LED_BLAU_AN;
}
```

Versuche, das Programm zu compilieren. Es wird klappen, allerdings gibt der Compiler eine Warnmeldung aus:

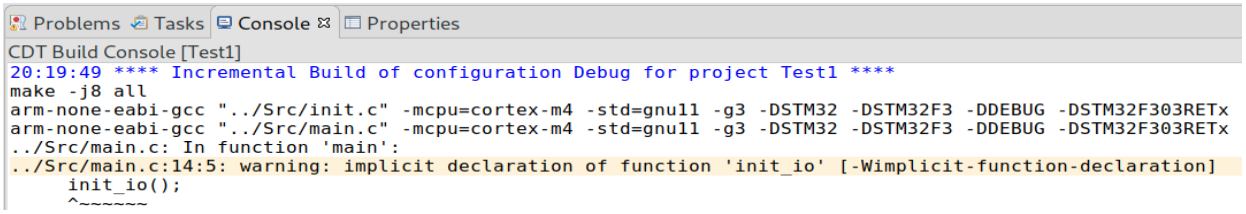



*Abbildung 40: Warnmeldung des Compilers im Console View*

Diese Warnmeldung bedeutet, dass er die Funktion „init_io“ nicht mehr finden kann. In solchen Fällen verlangt die Programmiersprache nach einem Hinweis, dass diese Funktion in einer anderen Quelltextdatei zu suchen ist. Das geht so:

```
#include <stdint.h>
#include "stm32f3xx.h"

#define LED_GRUEN_AN   WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5)
#define LED_ROT_AN     WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_0)
#define LED_BLAU_AN    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_1)

#define LED_GRUEN_AUS  WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5)
#define LED_ROT_AUS    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_0)
#define LED_BLAU_AUS   WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_1)

void init_io();

int main()
{
    init_io();

    LED_GRUEN_AN;
    LED_ROT_AN;
    LED_BLAU_AN;
}
```

Starte den Compiler nochmal. Dieses mal wird er nicht mehr meckern.

Eine alternative und wesentlich gebräuchlichere Methode ist, solche Ankündigungen in eine separate sogenannte Header-Datei zu schreiben. Diese Dateien heißen so, weil sie nur die Kopfzeilen der Funktionen enthalten. Klicke dazu mit der rechten Maustaste auf den „src“ Ordner und wähle den Befehl „New / Header File“. Der Dateiname soll „init.h“ lauten, mit folgendem Inhalt:

```
#ifndef INIT_H_
#define INIT_H_

void init_io();

#endif /* INIT_H_ */
```

Die drei automatisch erzeugten Zeilen verhindern Fehlermeldungen bezüglich doppelter Namen, für den Fall, dass man die Datei mehrfach einbindet. In diesem kleinen Projekt werden sie nicht wirklich gebraucht, in größeren Projekten manchmal aber schon.

Apropos Einbinden - wir müssen diese Datei jetzt in der „main.c“ einbinden. Und zwar so:

```
#include <stdint.h>
#include "stm32f3xx.h"
#include "init.h"

#define LED_GRUEN_AN   WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5)
#define LED_ROT_AN     WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_0)
#define LED_BLAU_AN    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_1)

#define LED_GRUEN_AUS  WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5)
#define LED_ROT_AUS    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_0)
#define LED_BLAU_AUS   WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_1)

int main()
{
    init_io();

    LED_GRUEN_AN;
    LED_ROT_AN;
    LED_BLAU_AN;
}
```

Probiere, das Programm zu compilieren. Es sollte ohne Warnmeldung klappen.Du kannst noch einen Schritt weiter gehen, und die „#define“ Zeilen auch in die Header Datei verschieben. Probiere es aus. Dadurch wird das Hauptprogramm „main.c“ ziemlich klein und übersichtlich, nicht wahr?

## 7.8 Printf

In diesem Kapitel zeige ich dir, wie man Zahlen ausgibt. Erstelle dazu eine neue Kopie vom „Blinker“ Projekt, welches „Test2“ heißen soll. Vergiss nicht, den Befehl „Clean Project“ aufzurufen. Ändere die „main“ Funktion, dass sie so aussieht:

```
int main()
{
    init_io();
    printf("SystemCoreClock hat den Wert %lu \n", SystemCoreClock);
}
```

Compiliere das Programm und übertrage es in den Mikrocontroller. Benutze das Terminal-Programm, um die Ausgabe zu sehen. Du kannst jederzeit den Reset-Knopf drücken, um das Programm erneut zu starten. Dann erscheint:

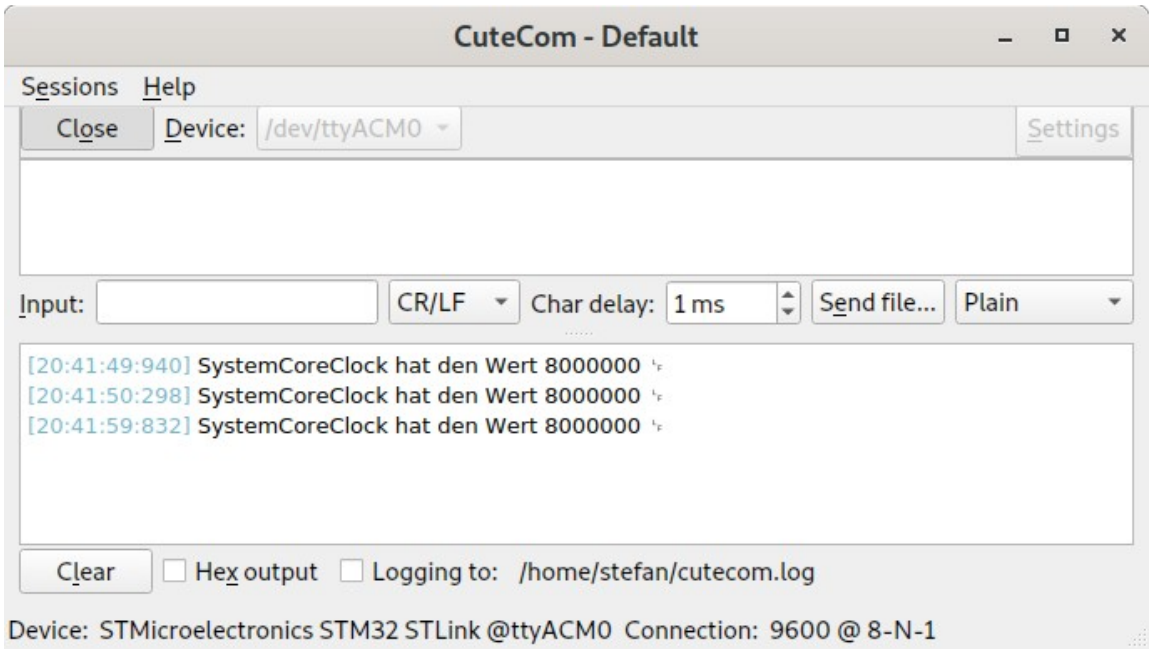



*Abbildung 41: Serielle Ausgabe in CuteCom*

Die Variable SystemCoreClock ist ganz oben in der „main.c“ mit einem Startwert von 800000 definiert. Genau diese Zahl siehst du hier in der Ausgabe. Die „printf“ Funktion gibt also nicht nur den Text aus, sondern ersetzt den Platzhalter %lu durch den Inhalt der Variablen.

Warum gerade %lu ? Das steht so in der Dokumentation der C Bibliothek. In deinem Fall ist es die „newlib“. Deren Dokumentation liegt hier:

https://www.sourceware.org/newlib/libc.html

Ich schätze allerdings, dass du mit dieser deutschen Dokumentation mehr anfangen kannst:

https://de.wikibooks.org/wiki/C-Programmierung:_Standard_Header

Ich benutze die folgenden Platzhalter:

| | |
|---|---|
| %c | Ein einzelnes Zeichen (char) |
| %s | Eine Zeichenkette (char[]) |
| %i oder %d | Ganze Zahl mit Vorzeichen (int) |
| %u | Ganze Zahl ohne Vorzeichen (unsigned int) |
| %x | Ganze Zahl im Hexadezimal Format (unsigned int) |
| %f | Fließkomma Zahl (float) |
| %% | Um das Prozent-Zeichen selbst auszugeben |

Für „lange“ 32 bit Zahlen soll man ein „l“ vor den Buchstaben Schreiben, obwohl int bei diesem Mikrocontroller ebenfalls 32bit groß ist. Der Compiler gibt eine Warnung aus, wenn du das „l“ vergessen hast:

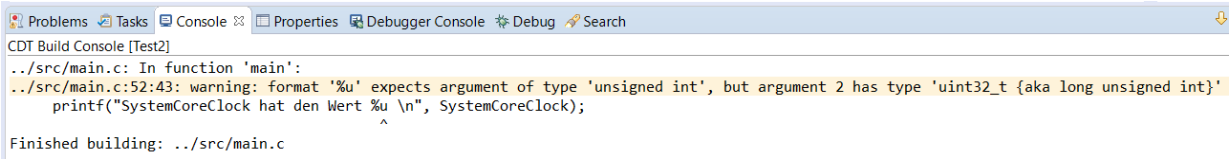



*Abbildung 42: Warnmeldung des Compilers im Console View*

Du kannst festlegen, dass die Zahl auf eine bestimmte Länge rechtsbündig ausgegeben wird. Wenn du noch eine „0“ vor die 12 schreibst, wird die Lücke mit Nullen aufgefüllt:

```
int main()
{
      init_io();
      printf("SystemCoreClock hat den Wert %lu \n", SystemCoreClock);
      printf("SystemCoreClock hat den Wert %12lu \n", SystemCoreClock);
      printf("SystemCoreClock hat den Wert %012lu \n", SystemCoreClock);
}
```

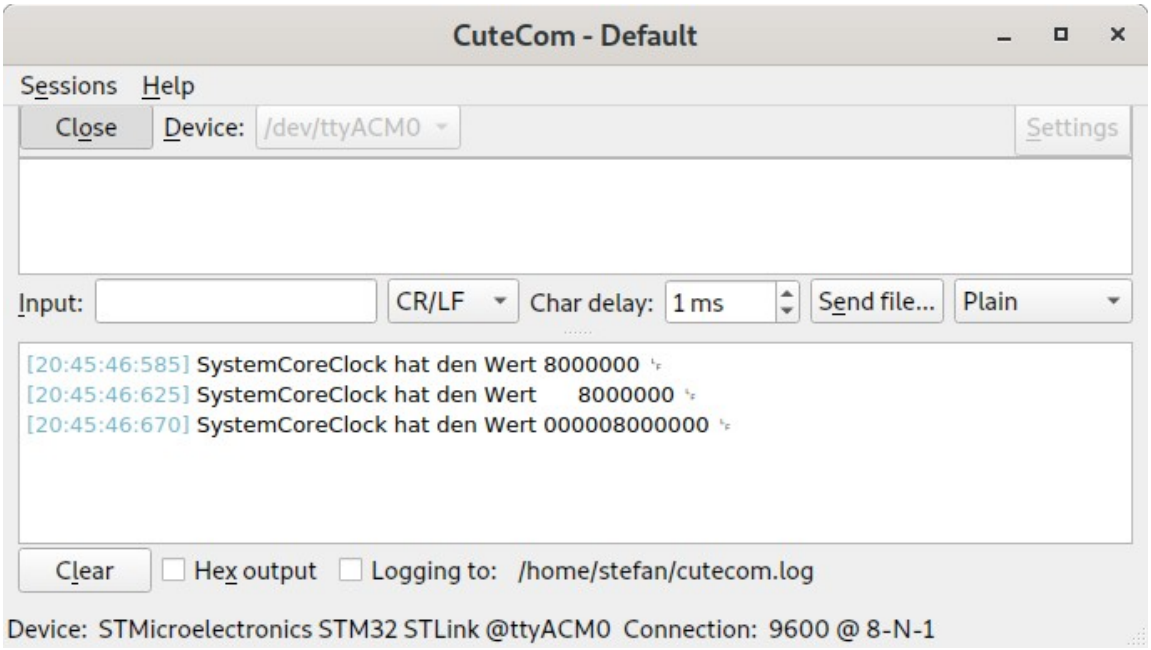



*Abbildung 43: Ausgabe in CuteCom*

Bei der zweiten Zeile sieht man deutlich, dass Leerzeichen eingefügt wurden. Weil CuteCom allerdings eine Proportionalschrift zur Darstellung verwendet, erscheinen Leerzeichen schmaler als Ziffern. Ein anderes Terminalprogramm mit einem „fixed“ oder „monspaced“ Font würde die beiden Zahlen korrekt rechtsbündig darstellen.

Die printf Funktion kann auch mehrere Platzhalter ersetzen:

```
int main()
{
    init_io();
    printf("Es ist %i Uhr und %02d Minuten \n", 9, 5);
}
```

Probiere es aus, die Ausgabe ist:

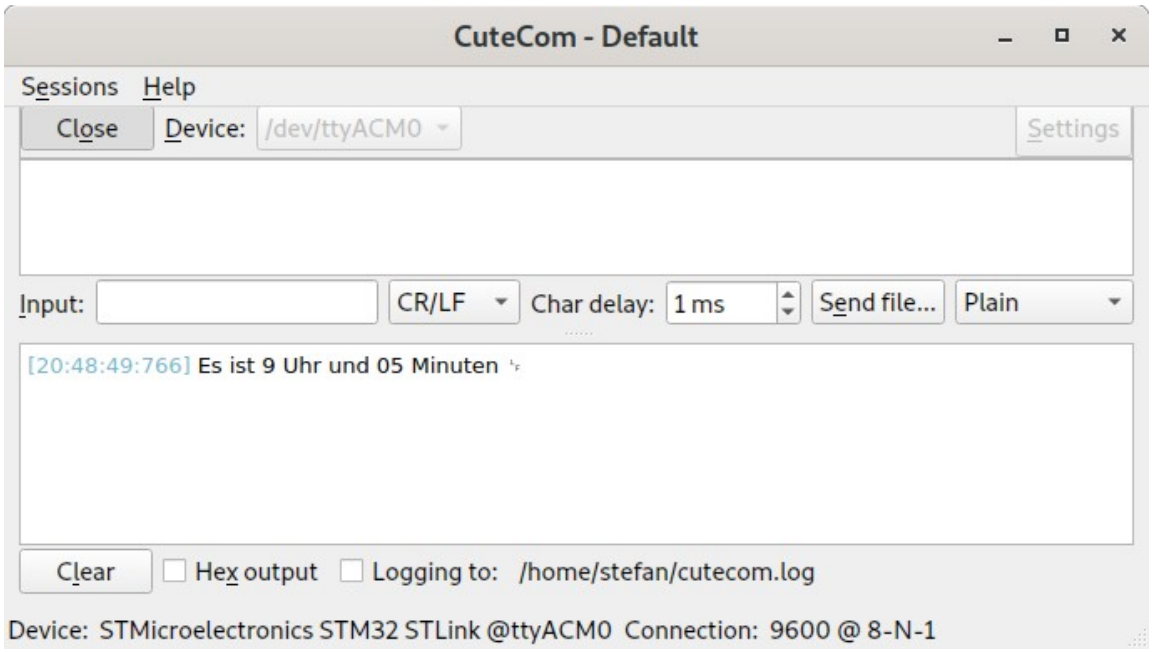



*Abbildung 44: Ausgabe in CuteCom*

Am Ende des Textes steht im Quelltext „\n“. Dieses Zeichen bedeutet „new line“ (neue Zeile), manchmal auch „line feed“ (Zeilenvorschub) genannt. Es ist eine Eigenart der newlib Library, dass die Ausgabe bis zum „\n“ verzögert wird, deswegen wirst du ohne „\n“ gar nichts sehen.

## 7.9 Variablen

Variablen dienen dazu, einen Platz im Arbeitsspeicher des Mikrocontrollers zu reservieren, wo du Daten (Zahlen und Texte) ablegen kannst. Die Variable „SystemCoreClock“ hast du bereits gesehen. Ihr Name ist durch die CMSIS festgelegt und soll stets die aktuelle Taktfrequenz des Mikrocontroller enthalten.

```
uint32_t SystemCoreClock=8000000;
```

Weitere Variablen kannst du nach belieben hinzufügen. Jede Variablen-Definition hat einen Typ, einen Namen und eventuell auch einen Startwert. Beispiele für Typen:

| **Typ** | **Bedeutung** | **Gültige Werte** |
|---|---|---|
| char | Ein Zeichen | a bis z, A bis Z, 0-9 und einige Sonderzeichen. Siehe https://de.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange |
| bool | Kann nur zwei Zustände haben | 0 (=false) oder 1 (=true) |
| int8_t | Ein Byte mit Vorzeichen | -128 bis 127 |
| uint8_t | Ein Byte ohne Vorzeichen | 0 bis 255 |
| int16_t | Zwei Bytes mit Vorzeichen | -32768 bis 32767 |
| uint16_t | Zwei Bytes ohne Vorzeichen | 0 bis 65535 |
| int32_t oder int | Vier Bytes mit Vorzeichen | -2147483648 bis 2147483647 |
| uint32_t oder unsigned int | Vier Bytes ohne Vorzeichen | 0 bis 4294967295 |
| int64_t | Acht Bytes mit Vorzeichen | -9223372036854775808 bis 9223372036854775807 |

| | | |
|---|---|---|
| uint64_t | Acht Bytes ohne Vorzeichen | 0 bis 18446744073709551615 |
| float | Fließkommazahl, vier Bytes mit 6 Stellen Genauigkeit | $-1{,}17 \cdot 10^{38}$ bis $3{,}40 \cdot 10^{38}$ |
| double | Fließkommazahl, acht Bytes, mit 15 Stellen Genauigkeit | $-2{,}22 \cdot 10^{308}$ bis $1{,}79 \cdot 10^{308}$ |

Der Typ „bool“ steht nur zur Verfügung, wenn dein Quelltext diese Zeile enthält:

```
#include <stdbool.h>
```

Die Typen uint8_t bis uint64_t stehen nur zur Verfügung, wenn dein Quelltext diese Zeile enthält:

```
#include <stdint.h>
```

Bedenke, dass sich der Mikrocontroller mit dem Berechnen von Fließkommazahlen schwer tut. Sie machen das Programm groß und träge, sind daher möglichst zu vermeiden. Die „printf“ Funktion kann deswegen normalerweise keine Fließkommazahlen ausgeben. Wenn du das unbedingt brauchst, musst eine Einstellung ändern: Markiere den Projektnamen und klicke dann auf den Menüpunkt Project/Properties. Dort musst du die Option „Use float with printf“ aktivieren:

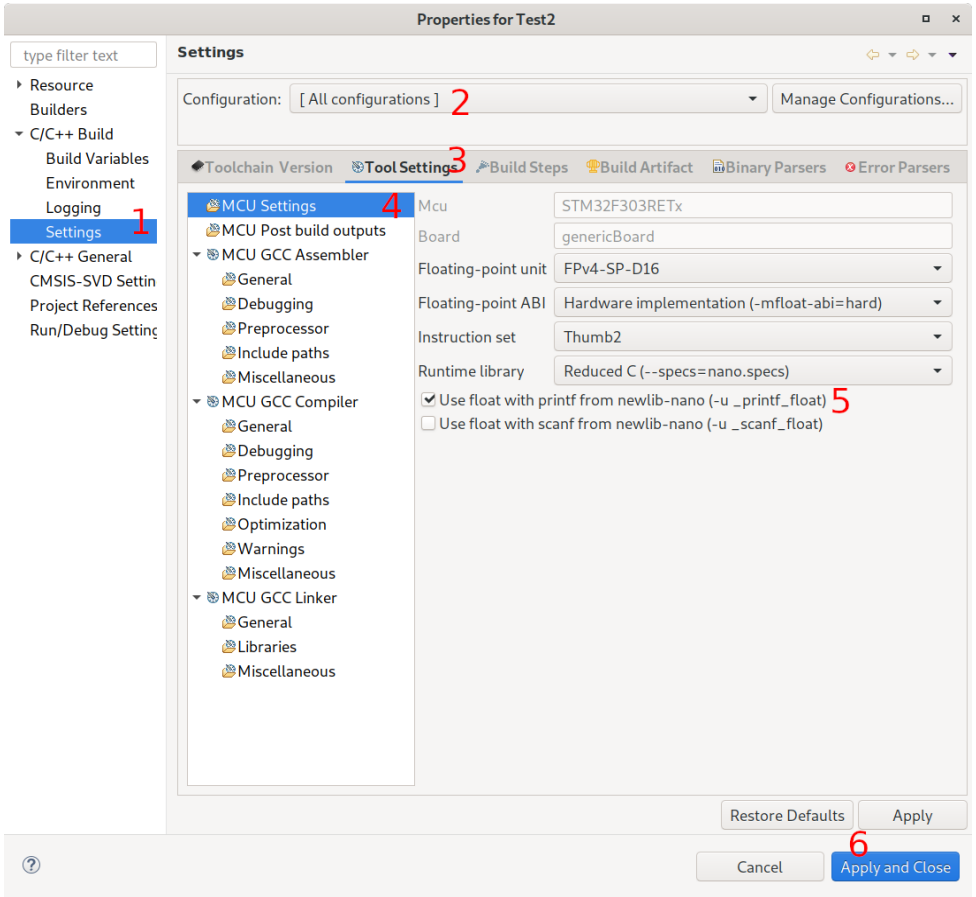



*Abbildung 45: Float für printf aktivieren*

Die Fließkomma-Einheit (FPU) des ARM Prozessors musst man extra einschalten, indem man folgende Zeile hinzufügt:

```
void init_io()
{
    ...

    // Die Fließkomma-Einheit einschalten
```

```
    SCB->CPACR = 0x00F00000;
}
```

Lass uns versuchen, ein paar Variablen mit printf zu auszugeben. Kopiere den folgenden Quelltext:

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

    int a=100;
    int b=200;
    int c;
    c=a + b;
    printf("a ist %i, b ist %i, beides zusammen ergibt %i \n", a,b,c);

    float pi=3.14159;
    float daumen=2.0;
    printf("pi mal daumen ist %f \n", pi * daumen);

    char ausrufezeichen='!';
    char name[30]="Stefan";
    printf("Hallo %s%c \n", name, ausrufezeichen);

    int zahlen[3];
    zahlen[0]=99;
    zahlen[1]=88;
    zahlen[2]=77;
    printf("Zahlen: %i, %i, %i \n", zahlen[0], zahlen[1], zahlen[2]);
}
```

Compiliere das Programm und führe es aus, um zu sehen, was dabei heraus kommt:

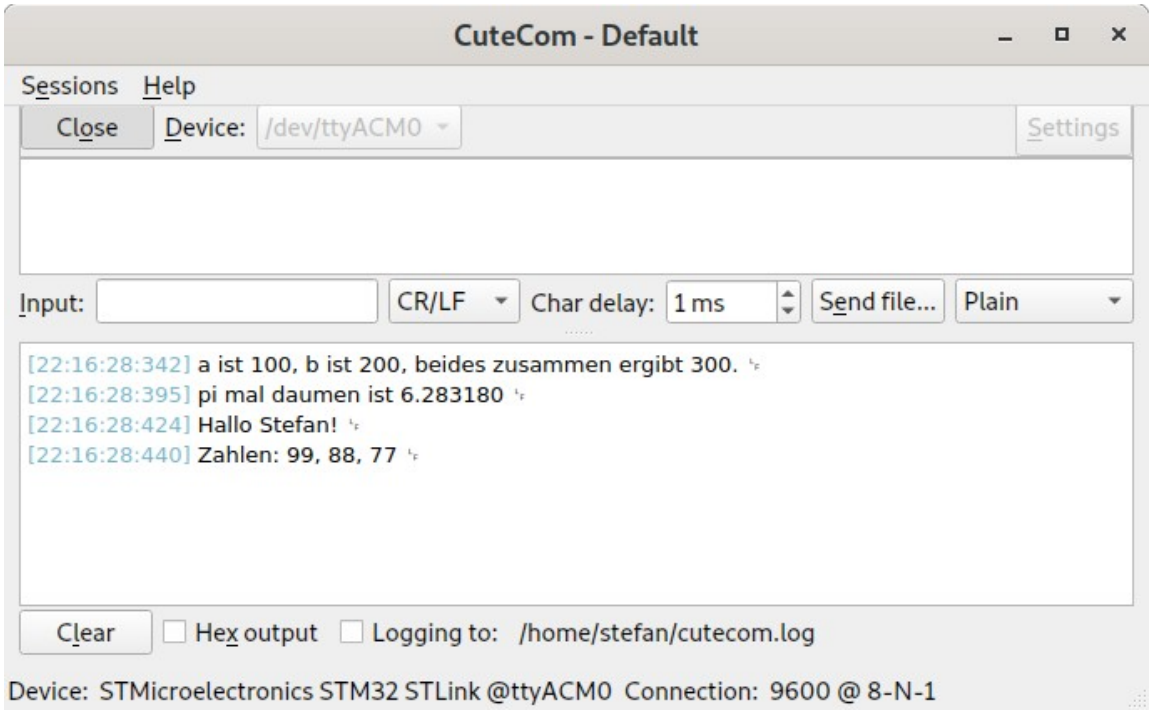



*Schaubild 1: Ausgabe in CuteCom*

Allerdings ist das Programm dreimal so groß geworden! Weil der Code zur Ausgabe von Fließkommazahlen richtig viel Platz und Rechenzeit benötigt, ist diese Funktion standardmäßig deaktiviert.

Ich erkläre nun diese Quelltext-Zeilen. Fangen wir mit dem ersten Block an:

```
int a=100;
int b=200;
int c;
c=a + b;
printf("a ist %i, b ist %i, beides zusammen ergibt %i \n", a,b,c);
```

Hier werden drei sogenannte Integer Variablen definiert. Die Variable a bekommt den Startwert 100, die Variable b bekommt den Startwert 200 und die Variable c bekommt keinen Startwert.

Danach wird ausgerechnet, wie viel a+b ergibt. Das Ergebnis wird in c gespeichert.

Die „printf“ Funktion gibt den Wert von allen drei Variablen aus.

Schauen wir uns den nächsten Block an:

```
float pi=3.14259;
float daumen=2.0;
printf("pi mal daumen ist %f \n", pi * daumen);
```

Hier werden zwei Variablen für Fließkommazahlen definiert. Beide haben den angegebenen Startwert. Beachte, dass Fließkommazahlen mit Punkt anstatt Komma geschrieben werden!

Die „printf“ Funktion gibt das Ergebnis der Multiplikation aus.

Schauen wir uns den nächsten Block an:

```
char ausrufezeichen='!';
char name[30]="Stefan";
printf("Hallo %s%c \n", name, ausrufezeichen);
```

Hier wird eine variable mit Namen „ausrufezeichen“ definiert, die als Startwert das Ausrufezeichen enthält. Beachte, dass man in C einzelne Zeichen zwischen **einfache** Anführungsstriche schreiben muss.

Die Variable „name“ kann bis zu 30 Zeichen[1] speichern, deswegen spricht man hier von einer „Zeichenkette“ (auf Englisch: String). Der Startwert soll mein Name sein. Beachte, dass man Zeichenketten zwischen **doppelte** Anführungsstriche schreiben muss.

Wieder wird die „printf“ Funktion verwendet, um den Inhalt dieser beiden Variablen auszugeben. %s muss für die Zeichenkette benutzt werden und %c für einzelne Zeichen.

Kommen wir zum letzten Block:

```
int zahlen[3];
zahlen[0]=99;
zahlen[1]=88;
zahlen[2]=77;
printf("Zahlen: %i, %i, %i \n", zahlen[0], zahlen[1], zahlen[2]);
```

Hier wird eine Variable definiert, die drei Integer Zahlen speichern kann. Solche Variablen nennt man „Array“ - in diesem Fall ein „Array von Integern“. In den darauf folgenden Zeilen werden die drei Speicherplätze mit Werten belegt. Beachte, dass die Nummerierung der Speicherplätze bei 0 beginnt.

Und wieder wird die „printf“ Funktion verwendet, um die Werte der drei Speicherplätze auszugeben.

Variablen, die innerhalb einer Funktion definiert wurden, sind nur dort erreichbar. Wenn du 5 Funktionen hättest, könnte jede Funktion eine eigene Variable „a“ haben. Der Compiler würde sie automatisch auseinander halten.

---

1 Die 30 Zeichen sind nicht ganz korrekt. Denn Zeichenketten enthalten an ihrem Ende immer eine unsichtbare Ende-Markierung mit dem Wert 0. Also haben wir eigentlich nur für maximal 29 Zeichen Platz.

Variablen, die außerhalb von Funktionen definiert wurden (wie „SystemCoreClock“), sind für das gesamte Programm erreichbar. Es kann also nur eine einzige Variable mit diesem Namen geben.

## 7.10 Rechnen

Im vorherigen Kapitel über die Variablen habe ich dir schon zwei Rechen-Operationen untergeschoben.

**+** für die Addition

***** für die Multiplikation

**-** Für die Subtraktion

Für die Division gibt es zwei Zeichen:

**/** berechnet die normale Division. Zum Beispiel ergibt 12/3 die Zahl 4. Wenn beide Operanden ganzzahlig sind und sich nicht glatt teilen lassen, wird die Zahl abgerundet. Also ergibt 9/2 die Zahl 4.

**%** berechnet den Rest der Division. 9%2 ergibt 1.

## 7.11 Wiederholschleifen

In dem Blinker Programm wird die LED immer wieder ein und aus geschaltet. Um solche Wiederholungen zu programmieren, brauchst du eine sogenannte „Wiederholschleife“. Die Programmiersprache C bietet dazu mehrere Möglichkeiten.

### 7.11.1 While

Fangen wir mit der „while“ Schleife an. Diese wird so lange wiederholt, wie eine Bedingung zutrifft. Die folgende main() Funktion verdeutlicht das:

```
int main()
{
    init_io();

    int zahl=0;
    while (zahl < 10)
    {
      printf("Die Zahl ist %i \n", zahl);
      zahl=zahl+1;
    }
}
```

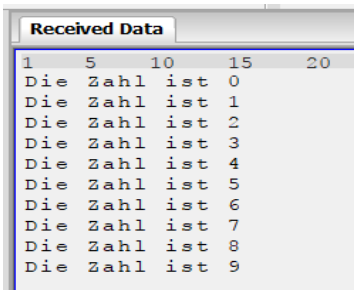



*Abbildung 46: Ausgabe im Hammer-Terminal*

Erklärung:

Es wird eine Variable mit dem Namen „zahl“ und dem Startwert 0 definiert. Dann kommt ein Block, der so lange wiederholt werden soll, wie die Zahl einen Wert kleiner als 10 enthält. Innerhalb des Blockes wird die Zahl ausgegeben und dann um eins erhöht.

Damit hast du den ersten Vergleichsoperator kennen gelernt. Es gibt noch mehr davon:

| | |
|---|---|
| < | kleiner als |
| <= | kleiner oder gleich |
| > | größer als |
| >= | größer oder gleich |
| == | gleich (bitte nicht mit „=“ verwechseln!) |
| != | nicht gleich |

Schau dir nochmal die Datei „main.c“ von dem Blinker Projekt an. Dort wird auch eine „while“ Schleife verwendet, aber irgendwie fehlt da eine ordentliche Bedingung.

```
while(1)
{
    …
}
```

Das macht man bei Schleifen, die für immer laufen sollen. Solange der Ausdruck zwischen den Klammern wahr ist, wird die Schleife wiederholt. Die Erfinder der Programmiersprache haben festgelegt, dass numerische Werte als „wahr“ gewertet werden, wenn sie nicht 0 sind. Demnach ist die Zahl 1 immer wahr, so das die Schleife ewig wiederholt wird.

### 7.11.2 For-Schleife

Die for-Schleife wird von vielen Programmierern alternativ zur While Schleife benutzt, wenn darin eine Variable fortlaufend erhöht oder verringert wird. Zum Beispiel kann man diese „while“ Schleife

```
int zahl=0;
while (zahl < 10)
{
  printf("Die Zahl ist %i \n", zahl);
  zahl=zahl+1;
}
```

wie folgt in eine „for-Schleife“ umschreiben:

```
for(int zahl=0; zahl < 10; zahl=zahl+1)
{
  printf("Die Zahl ist %i \n", zahl);
}
```

Das ist ein bisschen kompakter, bewirkt letztendlich aber genau das Selbe. Ob du „while“ oder „for“ verwenden wirst, bleibt ganz deinem persönlichen Geschmack überlassen. Interessanter ist eine andere Abkürzung, die man häufig in Zusammenhang mit Schleifen sieht:

```
zahl=zahl+1;
zahl+=1;
zahl++;
```

Alle drei Zeilen Varianten erhöhen den Wert der Variable „zahl“ um eins. Die for Schleife kann man demnach noch etwas kompakter schreiben:

```
for(int zahl=0; zahl < 10; zahl++)
{
  printf("Die Zahl ist %i \n", zahl);
}
```

### 7.11.3 Schleifen abbrechen

Den normalen Ablauf von Schleifen kann man mit zwei Befehlen unterbrechen:

„break“ bricht die Schleife sofort ab. Die restlichen Befehle innerhalb der Schleife werden nicht mehr ausgeführt.

„continue“ bricht nur den aktuellen Durchlauf ab, und macht danach mit der nächsten Wiederholung weiter.

```
int main()
{
    init_io();

    for(int zahl=0; zahl < 10; zahl++)
    {
        printf("Die Zahl ist %i \n", zahl);
        if (zahl > 3)
        {
            continue;
        }
        puts("la");
        puts("le");
        puts("lu");
    }
}
```

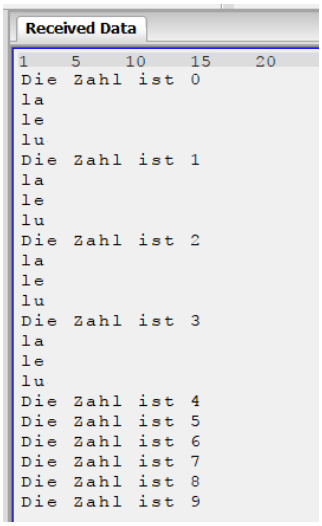



*Abbildung 47: Ausgabe im Hammer-Terminal*

Probiere das mal aus. Compiliere das Programm und führe es auf dem Mikrocontroller aus.

Du siehst, dass bei den Zahlen 0 bis 3 noch die Ausgabe von „lalelu“ erfolgt. Aber danach nicht mehr. Denn bei Zahlen die größer als 3 sind, wird der „continue“ Befehl ausgeführt.

Damit hast du gleich noch einen Befehl kennen gelernt, nämlich „if“. Mit „if“ kennzeichnet man einen Block, der nur ausgeführt wird, wenn eine bestimmte Bedingung erfüllt ist.

Übungsaufgabe:

Überlege, was wohl passieren wird, wenn du „continue“ durch „break“ ersetzt. Probiere es danach aus.

## 7.12 Bedingungen

Du hast oben in Zusammenhang mit dem „continue“ Befehl gesehen, wie man einen Block Quelltext bedingt ausführt. Die vollständige Schreibweise des „if“ Befehls ist:

```
if (Bedingung)
{
    tu etwas;
}
else
{
    tu etwas anderes;
}
```

Der „else“ Block wird ausgeführt, wenn die Bedingung **nicht** erfüllt ist. Mehrere Bedingungen kann man miteinander Verknüpfen. Ich verdeutliche das mal an einem theoretischen Beispiel:

```
int alter=14;
bool eltern_sind_anwesend = false;
```

```
if ( (alter<18) && (eltern_sind_anwesend==false) )
{
   puts("du kommst hier nicht rein");
}
else
{
    puts("zutritt ist genehmigt");
}
```

Hier müssen zwei Bedingungen gleichzeitig erfüllt sein. Man nennt dies eine UND Verknüpfung. Das Gegenstück dazu wäre die ODER-Verknüpfung:

```
int alter=14;

if ( (alter<0) || (alter>200) )
{
   puts("das kann kein gültiges alter sein");
}
```

Durch Benutzung von Klammern kann man komplexere Bedingungen formulieren, ganz ähnlich wie die Mathematik Klammern verwendet. Das Ausrufezeichen negiert den dahinter stehenden Ausdruck:

```
float kontostand=0.59;

if ( !(kontostand>20) )
{
   puts("du bist fast Pleite");
}
```

Die Fließkomma-Erweiterung des ARM Prozessors musst man extra einschalten, indem man folgende Zeile hinzufügt:

## 7.13 System-Timer

Alle ARM Mikrocontroller enthalten einen sogenannten „System-Timer“ oder „SysTick Timer“. Er teilt die Taktfrequenz des Mikrocontrollers herunter und ruft dann die C Funktion „SysTick_Handler“ in den entsprechenden Intervallen regelmäßig auf. Dies ist der relevante Teil vom Quelltext:

```
uint32_t SystemCoreClock=8000000;
volatile uint32_t systick_count=0;

void SysTick_Handler(void)
{
    systick_count++;
}

int main()
{
    SysTick_Config(SystemCoreClock/1000);
}
```

Die Variable SystemCoreClock muss der Taktfrequenz des Mikrocontrollers entsprechen. Das ist Standardmäßig 8 MHz.

In der „main“ Funktion wird der Teiler-Faktor des Timers eingestellt. In diesem konkreten Beispiel sollen die 8 MHz so geteilt werden, dass die C Funktion 1000 mal pro Sekunde aufgerufen wird. Also in 0,001 s Intervallen (=1 ms).

Die C-Funktion „SysTick_Handler“ erhöht bei jedem Aufruf einfach nur den Wert der Variable „systick_count“. Da diese einen Startwert von 0 hat kann man an der Variable also jederzeit ablesen, wie viele Millisekunden der Mikrocontroller seit dem letzten Reset gelaufen ist.

Der Mikrocontroller hat nur einen Prozessor-Kern, damit kann er nur ein Programm gleichzeitig ausführen. Damit diese Funktion trotzdem regelmäßig ausgeführt werden kann während das Hauptprogramm (main) läuft, muss der Mikrocontroller das Hauptprogramm kurzzeitig unterbrechen. Deswegen nennt man diese Funktion „Interrupt Handler“ oder „Interrupt Routine“.

| |
|---|
| Variablen, die von einem Interrupt-Handler verändert werden und außerhalb des Interrupts-Handlers benutzt werden, muss man als „volatile“ kennzeichnen. Sonst optimiert der Compiler den Programmcode falsch, was zu unerwarteten Fehlfunktionen führt. |

Wir sind noch im Projekt „Test2“. Ändere dort die Baudrate der seriellen Schnittstelle von 9600 auf 300, um eine langsame Ausgabe zu erzwingen. Kopiere dann das folgende Hauptprogramm:

```
int main()
{
    init_io();
    SysTick_Config(SystemCoreClock/1000);

    while(1)
    {
      printf("systick_count ist %lu \n",systick_count);
    }
}
```

Führe das Programm auf dem Mikrocontroller aus, und schau dir die Ausgabe im Terminal Programm an.

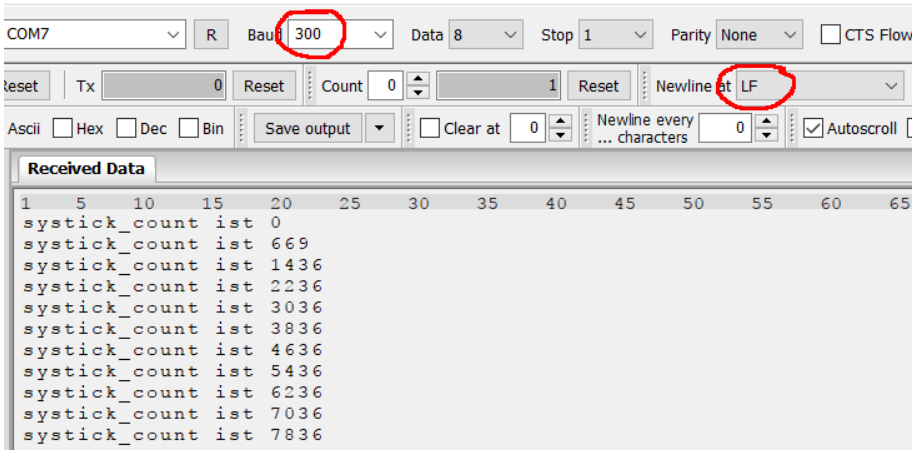



*Abbildung 48: Serielle Ausgabe im Hammer-Terminal nach Einstellung von Baudrate und Zeilenumbruch*

Du siehst hier, dass diese Variable fortlaufend hochgezählt wird. Da die Ausgabe ziemlich langsam stattfindet, kannst du nicht jede einzelne Zählung sehen, sondern ungefähr jede 800te. Während der Mikrocontroller den Text an deinen Computer sendet, wird der Interrupt-Handler offensichtlich 800 mal ausgeführt. Am Anfang ist die Ausgabe jedoch etwas schneller, weil die Zahlen kürzer sind.

Diese Zählvariable kannst du benutzen, um Zeiten zu messen oder um Pausen mit definierter Dauer einzufügen. Genau das macht das „Blinker“ Programm. Der relevante Quelltext ist:

```
    while(1)
    {
        // LED Aus
        WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5);

        // Warte 1 Sekunde
        uint32_t start=systick_count;
        while (systick_count-start<1000);

        // LED Ein
        WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5);

        // Warte 1 Sekunde
        start=systick_count;
        while (systick_count-start<1000);
    }
```

Hier merkt sich das Programm in der Variable „start“ zunächst den aktuellen Zählerstand. Danach wartet es so lange, bis weitere 1000 Millisekunden verstrichen sind. Diese „while“ Schleifen haben die Besonderheit, dass sie keinen Block enthalten. Während die Schleife wartet, wird hier einfach „nichts“ gemacht.

## 7.14 Funktionen

Du hast bereits einige C Funktionen benutzt und auch gesehen, wie man sie definiert. Ich habe dir aber noch nicht gezeigt, wie man eigene Funktionen **mit Parametern** definiert. Als Beispiel soll eine mathematische Funktion dienen, die eine Zahl verdoppelt:

```
int verdoppeln(int zahl)
{
    zahl=zahl*2;
    return zahl;
}

int main(void)
{
    init_io();
    int x=7;
    int ergebnis=verdoppeln(x);
    printf("Das doppelte von 7 ist %i \n", ergebnis);
}
```

Links vom Funktionsnamen „verdoppeln“ ist angegeben, dass diese Funktion eine Integer Zahl zurück liefert. In den Klammern ist angegeben, dass diese Funktion einen Parameter vom Typ Integer erwartet. Funktionen können mehrere Parameter haben, aber nur einen Rückgabewert. Die Funktion wird durch das Schlüsselwort „return“ beendet, und dabei wird das Ergebnis abgeliefert. Funktionen ohne Rückgabewert bzw. ohne Parameter benutzen hingegen das Schlüsselwort „void“ was so viel wie „nichts“ bedeutet. Den obigen Quelltext kann man so verkürzen:

```
int verdoppeln(int zahl)
{
    return zahl*2;
}

int main(void)
{
    init_io();
    printf("Das doppelte von 7 ist %i \n", verdoppeln(7));
}
```

Das Ergebnis bleibt identisch.

Die „math“ Library enthält viele nützliche mathematische Funktionen für Fließkommazahlen, zum Beispiel Sinus, Wurzel, Abrunden, usw. Ein Anwendungsbeispiel für die „math“ Library:

```
#include <math.h>
...
int main(void)
{
    init_io();
    printf("Die Wurzel von 16 ist %f \n", sqrt(16));
}
```

Die Ausgabe ist:

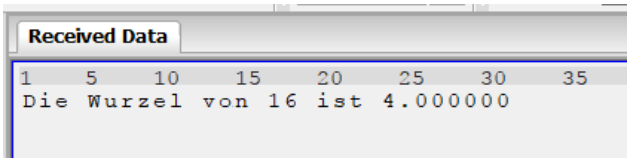



*Abbildung 49: Ausgabe im Hammer-Terminal*

Falls du das Ergebnis mit weniger Nachkommastellen anzeigen möchtest, kannst du den Platzhalter ändern: %0.2f Reserviert keinen Platz vor dem Komma und beschränkt die Ausgabe auf 2 Nachkommastellen.

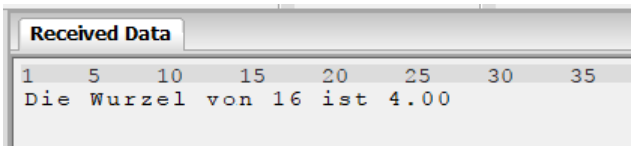



*Abbildung 50: Ausgabe im Hammer-Terminal*

Die Dokumentation der „math“ Library findest du auf der Seite

https://sourceware.org/newlib/libm.html

Oder auf deutsch:

https://de.wikibooks.org/wiki/C-Programmierung:_Standard_Header#math.h

## 7.15 Digitale Eingänge

In diesem Kapitel lernst du, wie man digitale Eingänge benutzt. Schließe einen Taster an PA6 (= Arduino Connector D12) an.

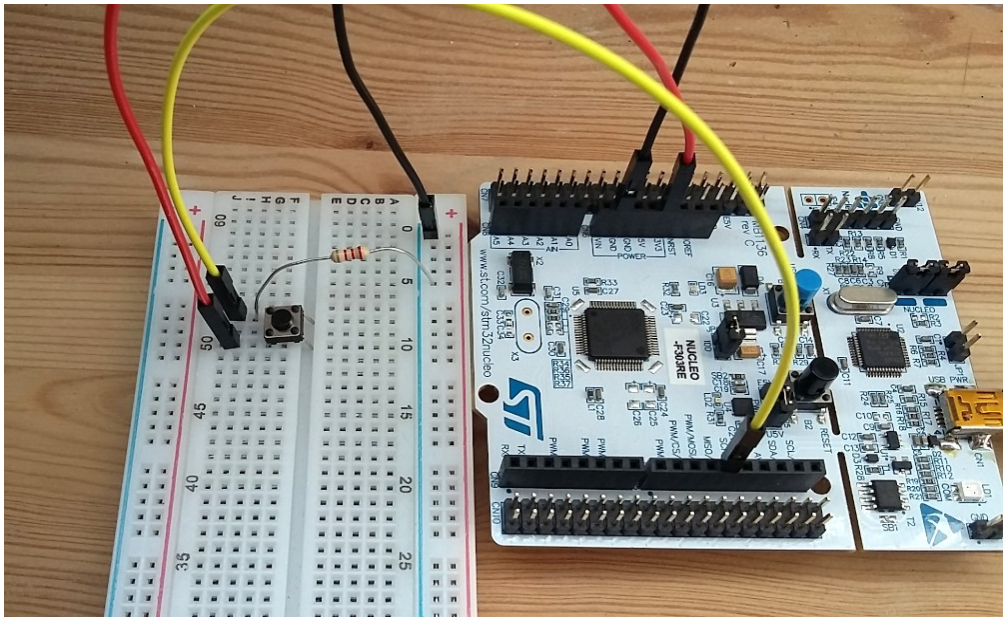


*Abbildung 51: Taster an PA6*

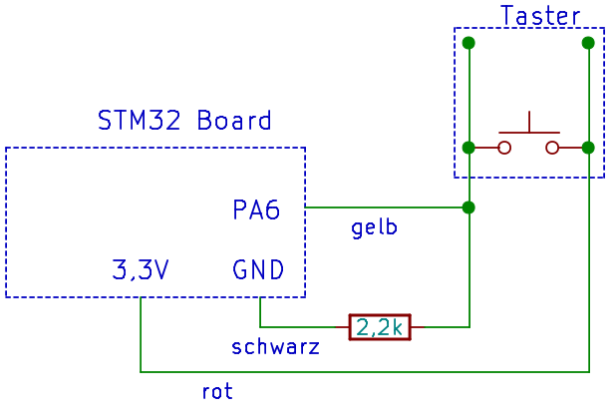



*Abbildung 52: Taster an PA6*

Die Anschlussbeinchen des Tasters musst du mit einer Zange glätten, damit sie in das Steckbrett passen. Bei diesen Tastern sind jeweils zwei Anschlüsse intern miteinander verbunden. Deswegen habe ich das so gezeichnet.

Im Ruhezustand zieht der Widerstand den Eingang PA6 auf GND (= 0 Volt, Low Pegel). Wenn du den Taster drückst, wir der Eingang PA6 mit 3,3 V verbunden, also High Pegel.

Wir werden das mit dem Multimeter überprüfen. Stelle dazu dein Multimeter auf den 20 V Messbereich. Halte die schwarze Mess-Spitze den Rahmen der USB Buchse (dort ist auch GND) und die rote Mess-Spitze an den Anschluss A6 (wo das gelbe Kabel steckt).

Das Messgerät zeigt zunächste 0 Volt an. Wenn du den Taster drückst zeigt es 3,3 Volt an. Dieses Signal werden wir jetzt im nächsten Programm auswerten, indem wir das IDR Register auslesen.

Damit ein Port als Eingang verwendet werden kann, muss er eingeschaltet werden, was bereits in der Funktion init_io() der Kopiervorlage gemacht wird:

```
SET_BIT(RCC->AHBENR, RCC_AHBENR_GPIOAEN);
```

Danach muss der gewünschte Eigang im MODER Register auf „00“ gestellt werden. Das ist bei fassten Pins bereits die Standard-Vorgabe, deswegen brauchen wir hierzu nichts zu programmieren. Den Status der Eingänge kann man dann auf dem IDR Register auslesen. Im Referenzhandbuch ist es so beschrieben:

### 11.4.5 GPIO port input data register (GPIOx_IDR) (x = A..H)

Address offset: 0x10

Reset value: 0x0000 XXXX (where X means undefined)

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. |
| | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| IDR15 | IDR14 | IDR13 | IDR12 | IDR11 | IDR10 | IDR9 | IDR8 | IDR7 | IDR6 | IDR5 | IDR4 | IDR3 | IDR2 | IDR1 | IDR0 |
| r | r | r | r | r | r | r | r | r | r | r | r | r | r | r | r |

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **IDRy:** Port input data bit (y = 0..15)
These bits are read-only. They contain the input value of the corresponding I/O port.

*Abbildung 53: Beschreibung des IDR Registers*

Wie du sehen kannst, sind alle Bits in diesem Register nur lesbar (r). Die Bits zeigen den aktuellen Zustand der Eingänge an.

Erstelle eine neue Kopie vom „Blinker“ Projekt und nenne es „Test3“. Vergiss nicht, das Projekt mit „Clean“ zu bereinigen . Kopiere die folgende „main“ Funktion:

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    while(1)
    {
        // Wenn der Taster gedrückt ist
        if (READ_BIT(GPIOA->IDR, GPIO_IDR_6))
        {
            // LED Ein
            WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5);
        }
        else
        {
            // LED Aus
            WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5);
        }
    }
}
```

Compiliere das Programm und führe es auf dem Mikrocontroller aus. Wenn du den Taster drückst, geht die grüne LED an. Wenn du ihn loslässt, geht sie wieder aus.

Dieses Programm hat ein neues Element, und zwar die Abfrage des Tasters:

```
READ_BIT(GPIOA->IDR, GPIO_IDR_6)
```

Das bedeutet soviel wie „Lese vom Port A das Bit 6 ein“. READ_BIT sieht wie eine Funktion aus, aber in Wirklichkeit steckt eine Textersetzung dahinter. Das kannst du herausfinden, indem du mit gedrückter Strg-Taste drauf klickst. Mache das mal, dann wirst du noch ein paar andere Textersetzungen entdecken, die du teilweise schon kennst.

Jedenfalls liefert READ_BIT eine Zahl die größer als 0 ist, wenn das Bit einen High Pegel hat. Und das ist der Fall, wenn der Taster gedrückt ist, weil der Taster den entsprechenden Eingang mit 3,3V (=High) verbindet.

Zahlen größer als 0 bedeuten für den „if“ Befehl, dass die Bedingung erfüllt ist. Deswegen wird der if-Block ausgeführt, wenn der Taster gedrückt ist. Der else-Block wird ausgeführt, wenn der Taster nicht gedrückt ist.

Ändere das Programm jetzt so, dass die LED nur dann an geht, wenn der Taster **nicht** gedrückt ist. Das geht zum Beispiel so:

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    while(1)
    {
        // Wenn der Taster gedrückt ist
        if (!READ_BIT(GPIOA->IDR, GPIO_IDR_6))
        {
            // LED Ein
            WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5);
        }
        else
        {
            // LED Aus
            WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5);
        }
    }
}
```

Eine andere Möglichkeit, die Aufgabe zu erfüllen wäre:

```
if (READ_BIT(GPIOA->IDR, GPIO_IDR_6) == 0)
```

Das bedeutet so viel wie: „Wenn READ_BIT() eine 0 zurück gibt, dann“. Und das ist der Fall, wenn der Taster **nicht** gedrückt ist.

Noch eine andere Möglichkeit wäre, die if-Bedingung einfach nicht zu ändern, sondern stattdessen die beiden Schaltbefehle für die LED zu vertauschen:

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    while(1)
    {
        // Wenn der Taster gedrückt ist
        if (READ_BIT(GPIOA->IDR, GPIO_IDR_6))
        {
            // LED Aus
            WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5);
        }
```

```
        else
        {
            // LED Ein
            WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5);
        }
    }
}
```

Übungsaufgabe:

Probiere alle drei Varianten aus.

Wir wollen nun einen zweiten Taster an PA7 (=Arduino Connector D11) hinzufügen.

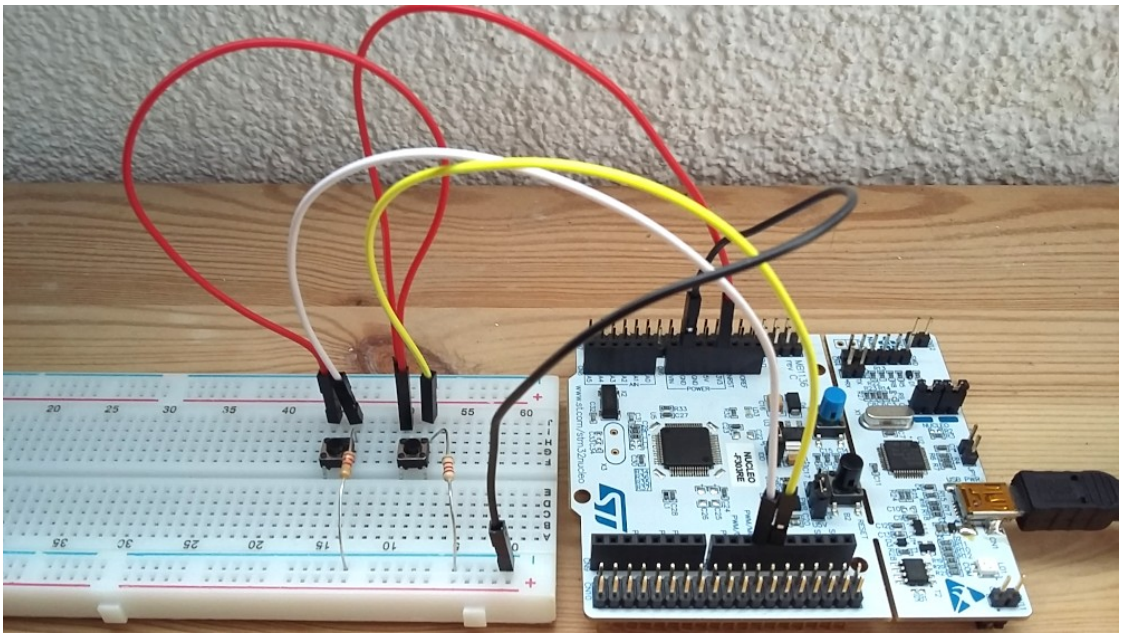


*Abbildung 54: Zwei Taster an PA6 und PA7*

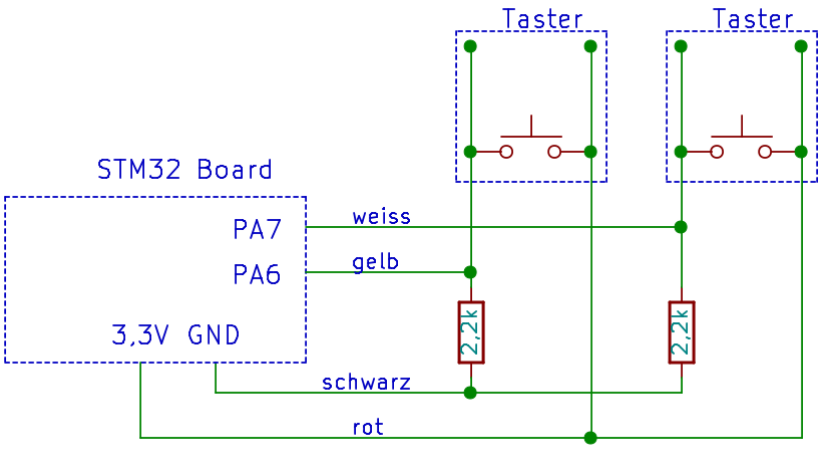



*Abbildung 55: Zwei Taster an PA6 und PA7*

Das dazu passende Programm sieht so aus:

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    while(1)
    {
        // Wenn der Taster an PA6 gedrückt ist
        if (READ_BIT(GPIOA->IDR, GPIO_IDR_6))
        {
            // LED Ein
            WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5);
        }

        // Wenn der Taster an PA7 gedrückt ist
        if (READ_BIT(GPIOA->IDR, GPIO_IDR_7))
        {
            // LED Aus
            WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5);
        }
    }
}
```

Übertrage das Programm in den Mikrocontroller um es auszuprobieren. Nun kannst du das grüne Licht mit einem Taster ein schalten und mit dem anderen wieder aus schalten. Vergleiche dieses Verhalten mit dem Quelltext des Programms.

Übungsaufgabe:

Was wird wohl passieren, wenn du beide Taster gleichzeitig drückst? Warum?

## 7.16 Digitale Ausgänge

Digitale Ausgänge hast du nun schon einige male benutzt, nämlich um Leuchtdioden anzusteuern. Dazu musstest du den jeweiligen Anschluss als Ausgang konfigurieren und dann auf High oder Low Pegel schalten. Ich fasse hier nochmal die Register und Befehle zusammen.

Zunächst muss der gewünschte Port im Register AHBENR eingeschaltet werden:

```
SET_BIT(RCC->AHBENR, RCC_AHBENR_GPIOAEN);
```

Im MODER Register konfiguriert man den Anschluss als Ausgang:

```
MODIFY_REG(GPIOA->MODER, GPIO_MODER_MODER5, 0b01 << GPIO_MODER_MODER5_Pos);
```

Über das BSRR Register kann man danach einzelne Pins auf Low oder High schalten:

```
WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5); // High
WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5); // Low
```

Man kann auch mehrere Pins gleichzeitig schalten, zum Beispiel PA5 und PA6:

```
WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR_5 + GPIO_BSRR_BR_6);
```

Jetzt kommt etwas neues. Das ODR Register repräsentiert den aktuellen Zustand aller Pins von einem Port. Im Referenzhandbuch ist es so dargestellt:

### 11.4.6 GPIO port output data register (GPIOx_ODR) (x = A..H)

Address offset: 0x14

Reset value: 0x0000 0000

| 31 | 30 | 29 | 28 | 27 | 26 | 25 | 24 | 23 | 22 | 21 | 20 | 19 | 18 | 17 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. | Res. |
| | | | | | | | | | | | | | | | |
| 15 | 14 | 13 | 12 | 11 | 10 | 9 | 8 | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| ODR15 | ODR14 | ODR13 | ODR12 | ODR11 | ODR10 | ODR9 | ODR8 | ODR7 | ODR6 | ODR5 | ODR4 | ODR3 | ODR2 | ODR1 | ODR0 |
| rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw | rw |

Bits 31:16 Reserved, must be kept at reset value.

Bits 15:0 **ODRy:** Port output data bit (y = 0..15)
These bits can be read and written by software.
*Note: For atomic bit set/reset, the ODR bits can be individually set and/or reset by writing to the GPIOx_BSRR or GPIOx_BRR registers (x = A..F).*

*Abbildung 56: Beschreibung des ODR Registers*

Dieses Register kann man sowohl auslesen als auch beschreiben:

```
uint16_t portA=GPIOA->ODR;      // status lesen
GPIOA->ODR=0;                   // setze alle Pins auf Low
GPIOA->ODR=65535;               // setze alle Pins auf High
```

Ich benutze an dieser Stelle gerne Zahlen im Binär-Format:

```
GPIOA->ODR=0b0000000000000000;  // setze alle Pins auf Low
GPIOA->ODR=0b1111111111111111;  // setze alle Pins auf High
```

Bei den Binärzahlen steht jede Ziffer für ein Bit, also einen Anschluss. Falls du Binärzahlen interessant findest, kannst du dich auf der folgenden Seite weiter informieren:

https://de.wikibooks.org/wiki/Mathematik:_Schulmathematik:_Zahlensysteme:_Bin%C3%A4re_Zahlen

Programmierer benutzen auch relativ häufig Hexadezimal-Zahlen. In der Programmiersprache C erkennt sie daran, dass sie mit „0x“ beginnen. Zum Beispiel „0xFF“.

https://de.wikibooks.org/wiki/Mathematik:_Schulmathematik:_Zahlensysteme:_Hexadezimale_Zahlen

Damit kannst du dich ruhig später befassen.

## 7.17 Analoge Eingänge

Viele Eingänge können analog gelesen werden. Dass heißt, jede beliebige Spannung zwischen 0 Volt und 3,3 Volt kann gemessen und verarbeitet werden. Du findest die geeigneten Pins im Datenblatt in der Tabelle „STM32F303xD/E pin definitions“ in der Spalte „Additional Functions“ unter dem Stichwort „ADC“. Der Chip enthält 3 Analog-zu-digital Konverter, wir werden aber nur den ersten davon benutzen:

PA0 = ADC1 IN1 (=Ardiuno Connector A0)

PA1 = ADC1 IN2 (=Ardiuno Connector A1)

~~PA2 = ADC1 IN3~~ Ist von der seriellen Schnittstelle zum ST-Link belegt

~~PA3 = ADC1 IN4~~ Ist von der seriellen Schnittstelle zum ST-Link belegt

PC4 = ADC1 IN5

PC0 = ADC1 IN6 (=Arddunio Connector A5)

PC1 = ADC1 IN7 (=Ardduino Connector A4)

PC2 = ADC1 IN8

PC3 = ADC1 IN9

PB11 = ADC1 IN14

Um einen Anschluss analog zu verwenden, muss du im Register MODER beide Bits einschalten. Füge zwei Zeilen zur Funktion init_io() hinzu, damit die Anschlüsse PA0 und PA1 analog funktionieren:

```
// Konfiguriere PA0 als analogen Eingang für ADC1 IN1
MODIFY_REG(GPIOA->MODER, GPIO_MODER_MODER0, 0b11 << GPIO_MODER_MODER0_Pos);

// Konfiguriere PA1 als analogen Eingang für ADC1 IN2
MODIFY_REG(GPIOA->MODER, GPIO_MODER_MODER1, 0b11 << GPIO_MODER_MODER1_Pos);
```

Nun brauchst du eine neue Funktion, welche den ADC (Analog zu digital Konverter) initialisiert. Kopiere das:

```
// Initialisiere den ADC1 für "Single conversion mode"
void init_analog()
{
    // Taktversorgung von ADC1 (und 2) einschalten
    SET_BIT(RCC->AHBENR, RCC_AHBENR_ADC12EN);

    // Den ADC deaktivieren
    if (READ_BIT(ADC1->ISR, ADC_ISR_ADRDY))
    {
        SET_BIT(ADC1->ISR, ADC_ISR_ADRDY);
    }
    if (READ_BIT(ADC1->CR, ADC_CR_ADEN))
    {
        SET_BIT(ADC1->CR, ADC_CR_ADDIS);
    }

    // Warte bis der ADC deaktiviert ist
    while (READ_BIT(ADC1->CR, ADC_CR_ADEN));

    // Den Spannungsregler vom ADC einschalten
    MODIFY_REG(ADC1->CR, ADC_CR_ADVREGEN, 0b00 << ADC_CR_ADVREGEN_pos);
    MODIFY_REG(ADC1->CR, ADC_CR_ADVREGEN, 0b01 << ADC_CR_ADVREGEN_pos);

    // Warte ein bisschen, damit sich die Spannung einpendeln kann
    delay_ms(2);

    // ADC Clock = HCLK/4
                            MODIFY_REG(ADC12_COMMON->CCR,       ADC12_CCR_CKMODE,
0b11<<ADC12_CCR_CKMODE_Pos);

    // Single ended mode für ale Kanäle
    WRITE_REG(ADC1->DIFSEL,0);

    // Starte die Kalibrierung
    CLEAR_BIT(ADC1->CR, ADC_CR_ADCALDIF);
    SET_BIT(ADC1->CR, ADC_CR_ADCAL);

    // Warte, bis die Kalibrierung beendet ist
    while (READ_BIT(ADC1->CR, ADC_CR_ADCAL));
```

```
    // Lösche das Bereit-Signal
    SET_BIT(ADC1->ISR, ADC_ISR_ADRDY);

    // Aktiviere den ADC wiederholt, bis es klappt (siehe Errata Dokument)
    do
    {
        SET_BIT(ADC1->CR, ADC_CR_ADEN);
    }
    while (!READ_BIT(ADC1->ISR, ADC_ISR_ADRDY));

    // Wähle Start-Signal von Software
    MODIFY_REG(ADC1->CFGR, ADC_CFGR_EXTEN, 0b00 << ADC_CFGR_EXTEN_Pos);

    // Wähle einfache Messung
    CLEAR_BIT(ADC1->CFGR, ADC_CFGR_CONT);

    // Stelle die Mess-Zeit auf 32 Zyklen ein
    MODIFY_REG(ADC1->SMPR1, ADC_SMPR1_SMP1, 0b100 << ADC_SMPR1_SMP1_Pos);
}
```

Als nächstes brauchst du eine Funktion, die einen analogen Eingang einliest. Kopiere das:

```
// Lese einen Eingang von ADC1 ein
uint32_t read_analog(uint32_t channel)
{
    // Anzahl der zu messenden Kanäle: 1
    MODIFY_REG(ADC1->SQR1, ADC_SQR1_L, 0); // Er misst immer einen mehr

    // Wähle den Kanal aus
    MODIFY_REG(ADC1->SQR1, ADC_SQR1_SQ1, channel << ADC_SQR1_SQ1_Pos);

    // Lösche das Fertig-Signal
    CLEAR_BIT(ADC1->ISR, ADC_ISR_EOC);

    // Starte die Messung
    SET_BIT(ADC1->CR, ADC_CR_ADSTART);

    // Warte bis die Messung abgeschlossen ist
    while (!READ_BIT(ADC1->ISR, ADC_ISR_EOC));
    while (READ_BIT(ADC1->CR, ADC_CR_ADSTART));

    // Liefere die unteren 12 Bits vom Ergebnis zurück
    return ADC1->DR & 0b111111111111;
}
```

Nun kannst du das Hauptprogramm umschreiben, so dass es diese beiden analogen Eingänge einlist und die Messwerte über den seriellen Port an den PC sendet:

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    // Initialisiere den System-Timer
    SysTick_Config(SystemCoreClock/1000);

    // Initialisiere den ADC
    init_analog();

    while(1)
    {
        // LED Blitzen
        delay_ms(1000);
        WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5);
```

```
        delay_ms(100);
        WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5);

        // Lese die analogen Eingänge IN1 und IN2 (=PA0 und PA1)
        uint32_t in1=read_analog(1);
        uint32_t in2=read_analog(2);
        printf("analog: %lu, %lu \n", in1, in2);
    }
}
```

Hier ist wichtig, dass der System-Timer vor der Funktion „init_analog“ gestartet wird, weil sie den System-Timer benutzt. Probiere das Programm aus, ohne dass irgendwelche Bauteile an das Mikrocontroller Board angeschlossen sind. Die Ausgabe wird ungefähr so aussehen:

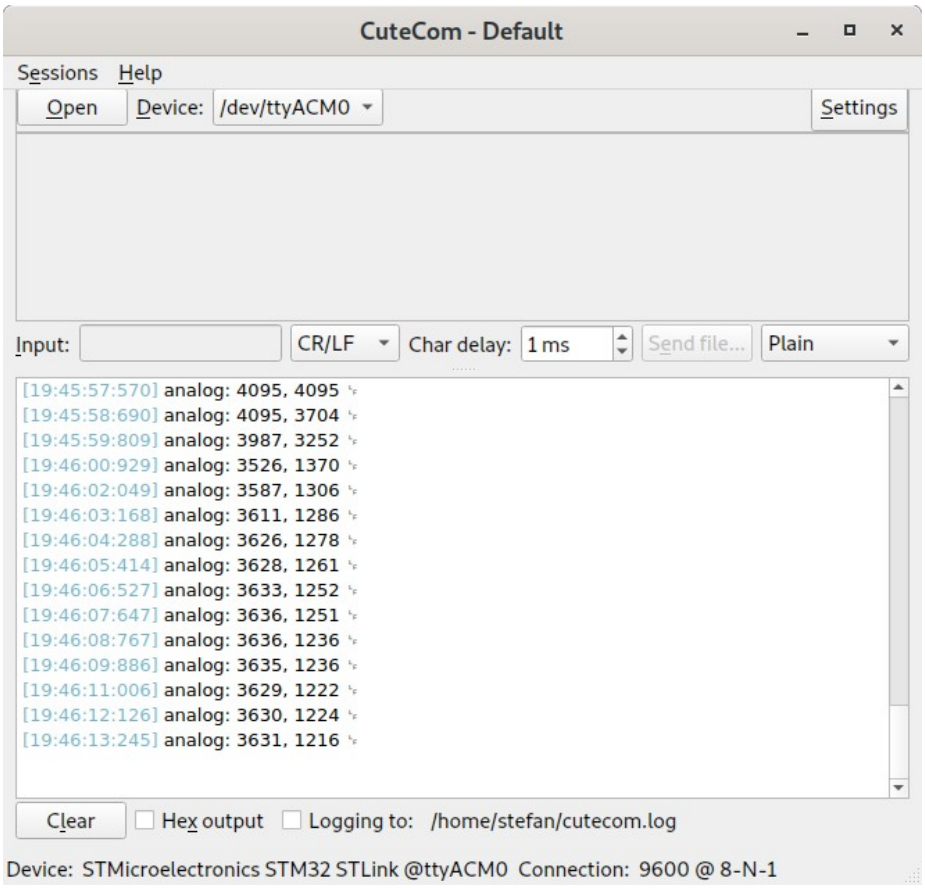



*Abbildung 57: Analoge Messwerte in CuteCom*

Du erhältst offensichtlich zufällige Messwerte, was auch logisch ist, denn an den analogen Eingängen PA0 und PA1 ist noch nichts angeschlossen. Der Mikrocontroller reagiert sehr empfindlich auf elektromagnetische Felder, welche diese Schwankungen bewirken.

Verbinde den Eingang PA0 nun mit GND. Du erhältst sofort ungefähr den Messwert 0. Verbinde den Eingang PA1 mit 3,3V. Du erhältst ungefähr den Messwert 4095. Das gleiche kannst du auch mit PA1 probieren. Du wirst feststellen, dass die beiden Eingänge sich gegenseitig beeinflussen, solange sie „offen“ in der Luft hängen. Wenn du später richtige analoge Signalquellen anschließt, wird dieser Effekt verschwinden.

Zwischen 0 und 4095 liegen noch viele andere mögliche Werte, und die bekommst du, wenn die Spannung zwischen 0 und 3,3 Volt liegt. Das wollen wir jetzt mit einer echten analogen Signalquelle (einem Lichtsensor) ausprobieren.

Baue die folgende Schaltung auf:

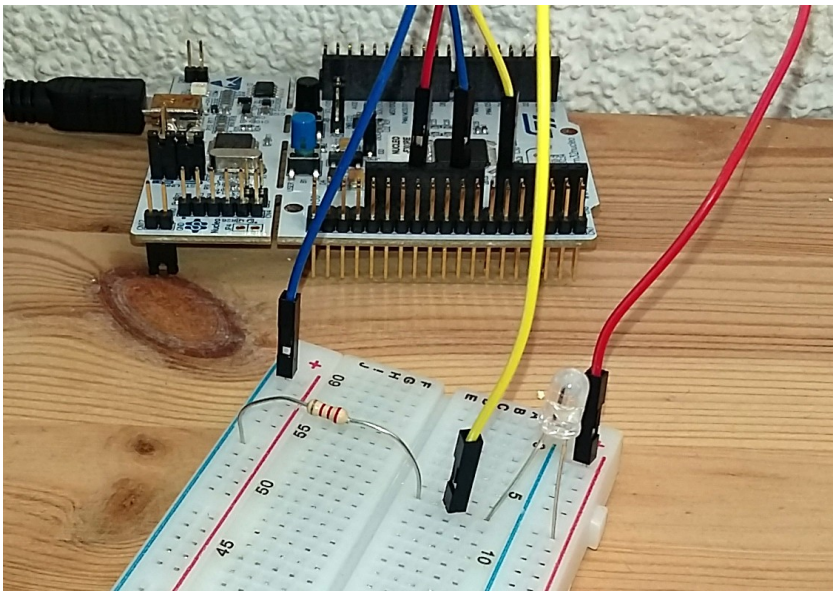


*Abbildung 58: Lichtsensor am analogen Eingang A0*

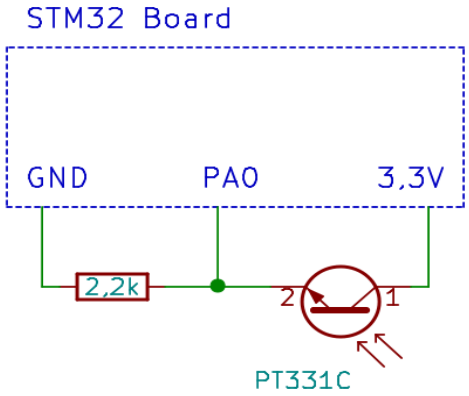



*Abbildung 59: Lichtsensor am analogen Eingang A0*

Das große durchsichtige Ding, dass wie eine LED aussieht, ist der Lichtsensor, genauer gesagt ein Fototransistor. Dessen langer Anschluss gehört nach links.

Der Widerstand zieht den Anschluss PA0 auf 0 Volt (GND). Wenn Licht auf den Sensor fällt, fließt aber auch ein bisschen Strom vom 3,3V Anschluss, so dass sich die gemessene Spannung erhöht. Du kannst das mit dem Multimeter überprüfen. Halte die schwarze Messleitung an GND und die rote an PA0. Je heller das Licht ist, umso größer die Spannung.

Schau dir die Ausgabe im Terminal-Programm an und beobachte, wie sich Helligkeitsänderungen auf den angezeigten Messwert auswirken.

| |
|---|
| Das Hammer-Terminal hat einen Fehler. Wenn man es für sehr lange Zeit laufen lässt, geraten irgendwann die Buchstaben durcheinander. Das Problem verschwindet, wenn du die Verbindung mit „Disconnect“ trennst und dann wieder neu aufbaust. |

Übungsaufgabe:

Ersetze den Lichtsensor durch den Temperatursensor (NTC). Reagiert er auf Wärme? Spielt es eine Rolle, wie herum er gedreht eingebaut wird?

Schließe den Lichtsensor an PA0 an und den Temperatursensor an PA1. Du brauchst dazu einen zweiten 2,2kΩ Widerstand und einige Dupont-Kabel.

Noch eine Übungsaufgabe:

Wenn du gut Englisch kannst, dann vergleiche die Registerzugriffe mit dem Referenzhandbuch des Mikrocontrollers. So kannst du Zeile für Zeile nachvollziehen, wie der ADC benutzt wird.

*Abbildung 60: NTC*

## 7.18 Bit-Operationen

Die Funktion „read_analog“ enthält eine Operation, die du noch nicht kennen gelernt hast:

```
// Liefere die unteren 12 Bits vom Ergebnis zurück
return ADC1->DR & 0b111111111111;
```

Hier werden Bits maskiert – so nennt man das.

Dieses DR Register ist 32 Bit groß und enthält mehr als nur die eine Zahl, die wir haben wollen. Wie der Kommentar sagt, sollen nur die unteren 12 Bits verwendet werden. Die restlichen Bits werden durch die Maske entfernt. Wobei ich bei der Maske die führenden Nullen einfach weg gelassen habe. Folgendes passiert hier:

```
Operand DR:      11010101010101111100100111101101
Operand Maske:   00000000000000000000111111111111
Ergebnis:  00000000000000000000100111101101
```

Im Ergebnis haben nur die Bits den Wert 1, wo beide Operanden eine 1 haben. Hier findet also eine bitweise UND Verknüpfung statt.

Das Gegenstück dazu ist die Bitweise ODER Verknüpfung, für die man „|“ schreiben muss. Bei der ODER Verknüpfung sind im Ergebnis die Bits auf 1 gesetzt, wo der erste oder der zweite Operand eine 1 hat:

```
Operand 1: 11010101010101111100100111101101
Operand 2: 00000000000000000000111111111111
Ergebnis:  11010101010101111100111111111111
```

| |
|---|
| Verwechsle die bitweisen Verknüpfungen „&“ sowie „|“ nicht mit den logischen Verknüpfungen „&&“ sowie „||“. |

# 8 Anwendungsbeispiele

Du hast bis jetzt nur einen winzigen Bruchteil der Funktionen deines Mikrocontrollers kennen gelernt. Und auch von der Programmiersprache C hast du noch nicht alles gesehen. Aber es reicht schon aus, um viele tolle Sachen zu bauen. Darum geht es in diesem Kapitel. Ich habe weitere Details zur Programmiersprache und zur Elektronik in diese Anwendungsbeispiele einfließen lassen. Bitte lese diese Kapitel der Reihe nach durch, auch wenn du nicht alle Schaltungen ausprobierst.

## 8.1 Dämmerungsschalter

Das erste Anwendungsbeispiel wird ein Dämmerungsschalter sein. Dazu kannst du den letzten Aufbau aus dem Kapitel „Programmieren in C“ erweitern. Dämmerungsschalter dienen dazu, eine Lampe automatisch einzuschalten, wenn es dunkel wird. Wir verwenden den Fototransistor, um die Helligkeit des Lichts zu messen. Eine kleine weiße LED wird als Modell für eine größere Lampe dienen:

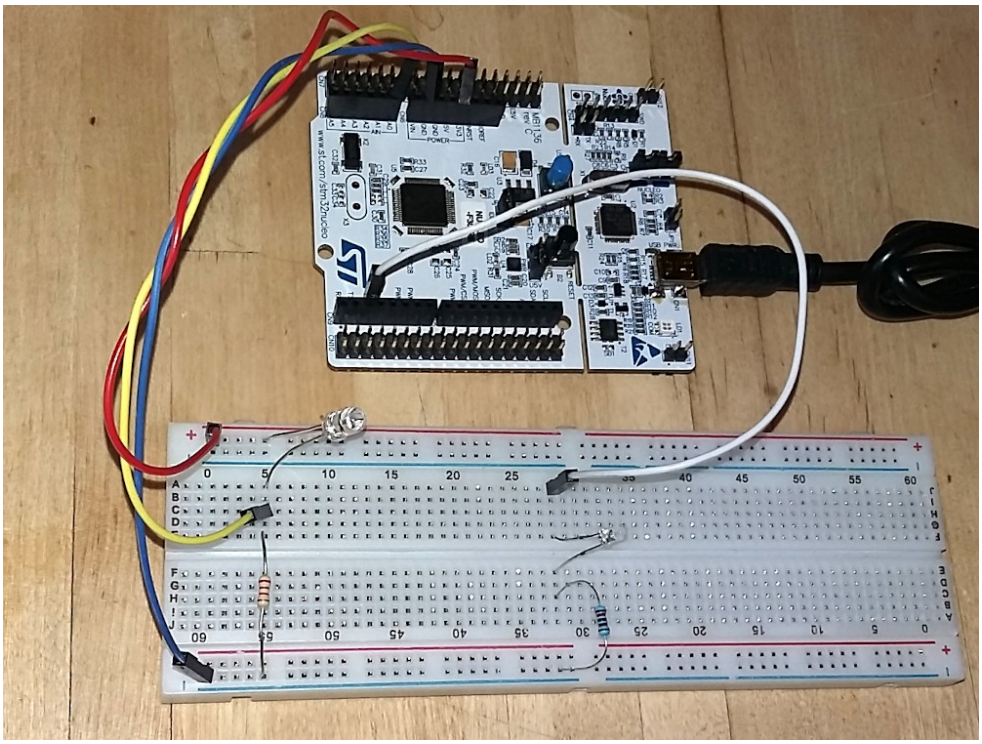


*Abbildung 61: Dämmerungsschalter*

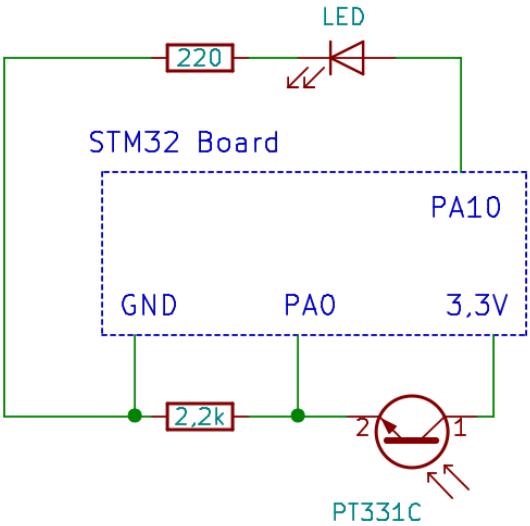



*Abbildung 62: Dämmerungsschalter*

Beachte beim Aufbauen, dass der lange Anschluss der LED an den PA10 gehört. Beim Fototransistor gehört der lange Anschluss an PA0. Die Schaltung funktioniert so:

Der 2,2k Ohm Widerstand zieht den analogen Eingang PA0 auf 0 Volt (GND). Der Fototransistor hingegen zieht das Signal hoch auf 3,3 Volt. Je heller es ist, umso höher steigt die Spannung an PA0, weil der Fototransistor dann besser leitet.

Erstelle ein neues Programm auf Basis der „Blinker“ Vorlage. Zur Initialisierung der Anschlüsse PA0 und PA10 musst du die Funktion „init_io“ erweitern:

```
// Konfiguriere PA0 als analogen Eingang für ADC1 IN1
MODIFY_REG(GPIOA->MODER, GPIO_MODER_MODER0, 0b11 << GPIO_MODER_MODER0_Pos);

// Konfiguriere PA10 = Ausgang für die weiße LED
MODIFY_REG(GPIOA->MODER, GPIO_MODER_MODER10, 0b01 << GPIO_MODER_MODER10_Pos);
```

Kopiere die beiden Funktionen „init_analog“ und „read_analog“ aus dem Kapitel „Analoge Eingänge“. Die Funktion „main“ soll so aussehen:

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    // Initialisiere den System-Timer
    SysTick_Config(SystemCoreClock/1000);

    // Initialisiere den ADC
    init_analog();

    while(1)
    {
        // Lese den analogen Eingang IN1 (=PA0)
        uint32_t helligkeit=read_analog(1);
        printf("helligkeit: %lu \n", helligkeit);

        // Wenn es dunkel ist
        if (helligkeit < 300)
        {
            // Licht an
            WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_10);
        }
        else
        {
            // Licht aus
            WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_10);
        }

        // Warte eine Sekunde
        delay_ms(1000);
    }
}
```

Wenn die gemessene Helligkeit unter 300 liegt, wird das Licht eingeschaltet. Ansonsten wird es aus geschaltet. Du kannst diesen Schwellwert ändern, wenn er dir nicht gefällt. Die Ausgabe im Terminal Programm wird dabei hilfreich sein.

Beleuchte den Lichtsensor mit einer Lampe, dann geht die LED aus. Beschatte den Lichtsensor, dann geht die LED an. Im Terminal-Programm erscheinen fortlaufend die aktuellen Messwerte:

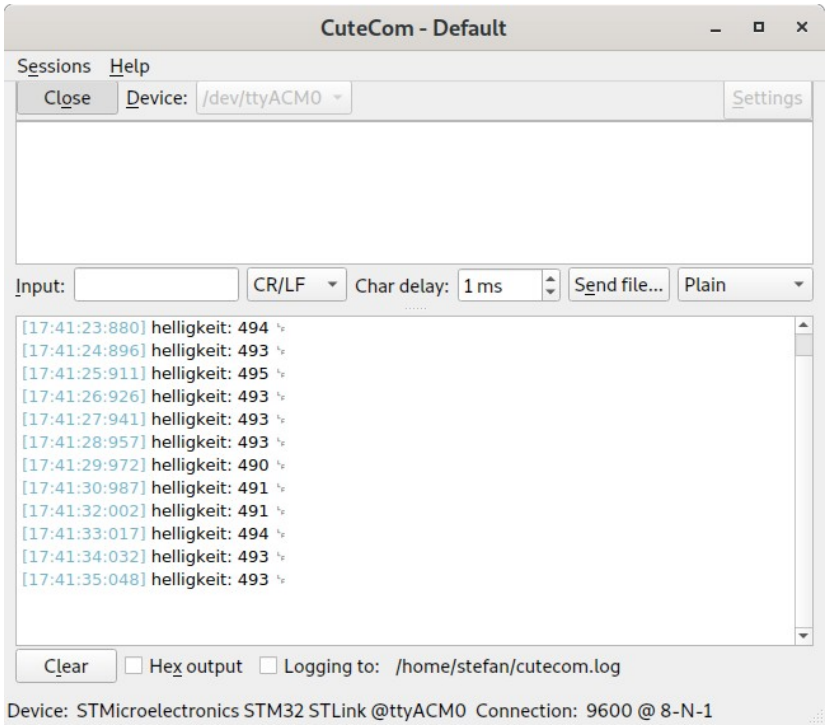



*Abbildung 63: Terminal-Ausgabe vom Dämmerungsschalter*

Wie du siehst, schwanken sie Zahlen immer ein kleines bisschen. Wenn jetzt die Schaltschwelle genau dazwischen wäre, würde die „Lampe“ immer wieder an und aus gehen, vielleicht sogar wild flackern. Das würde einem schnell auf die Nerven gehen.

Probiere es aus: Ändere den Schwellwert (die 300) im Programm so, das er dem Mittelwert der gerade aktuellen Ausgabe entspricht. In meinem Fall wäre das die Zahl 491 oder 491.

Um den Dämmerungsschalter zu verbessern, richtet man zwei Schaltschwellen ein:

```
        // Wenn es dunkel ist
        if (helligkeit < 250)
        {
            // Licht an
            WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_10);
        }

        // Wenn es hell ist
        else if (helligkeit >300)
        {
            // Licht aus
            WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_10);
        }
```

Den Bereich zwischen 250 und 300 nennt man „Hysterese“. Die Hysterese hat die Größe 50, so stark darf die Helligkeit schwanken, ohne dass der Dämerungsschalter darauf reagiert.

Teste das Ergebnis. Eventuell möchtest du andere Schaltschwellen verwenden oder die Hysterese verringern.

Eine Hysterese kennst du vielleicht auch von der Heizung deiner Wohnung. Wenn du das Thermostat auf 20°C stellst, dann springt die Heizung erst bei 19,5°C an und geht bei 21,5°C aus. Das eine Grad dazwischen ist die Hysterese. Sie verhindert, dass die Anlage übermäßig häufig ein/aus schaltet.

Hast du bemerkt, dass ich bei der zweiten Bedingung den Befehl „else if“ anstatt „if“ verwendet habe? Dadurch wird die Geschwindigkeit der Programmausführung geringfügig verbessert. So wird die zweite Bedingung nur dann geprüft, wenn die erste nicht erfüllt war.

In diesem Anwendungsbeispiel ist es allerdings ziemlich egal, ob das Programm ein paar Mikrosekunden mehr oder weniger benötigt. Immerhin folgt danach eine absichtliche Verzögerung von einer ganzen Sekunde.

Bei dem Dämmerungsschalter kannst du einen interessanten Effekt auslösen. Stelle die Schaltschwellen so ein, dass die Lampe bei den aktuellen Lichtverhältnisse ein geschaltet ist. Biege dann die Leuchtdiode und den Phototransistor so, dass sie aufeinander zeigen:

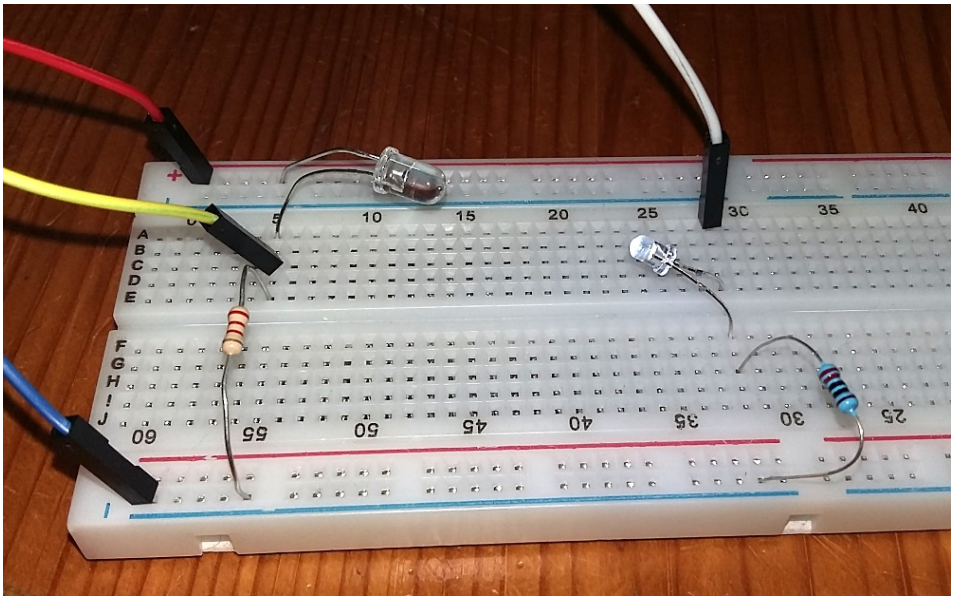


*Abbildung 64: LED zeigt auf Lichtsensor*

Jetzt blinkt die LED regelmäßig im Sekundentakt. Bevor du weiter liest, überlege selbst, warum das passiert.

Erklärung:

Zunächst ist es dunkel, so dass die LED eingeschaltet wird. Eine Sekunde später bemerkt das Programm, dass es nun hell ist. Deswegen schaltet es die LED wieder aus. Dann bemerkt das Programm wieder eine Sekunde später, dass es dunkel ist. Damit schließt sich der Kreis.

Du hast einen sogenannten „Schwingkreis“ gebaut. Bei einer realen Anwendung will man das natürlich nicht haben. Deswegen muss man dafür sorgen, dass das Licht der Lampe nicht direkt auf den Sensor fällt.

## 8.2 Eieruhr

Die Eieruhr hilft beim Kochen von idealen Frühstückseiern. Nach genau 3 Minuten erzeugt sie einen Signalton. Wir schließen einen Lautsprecher an den Ausgang PC0 an, das ist am Arduino Connector der Stift A5.

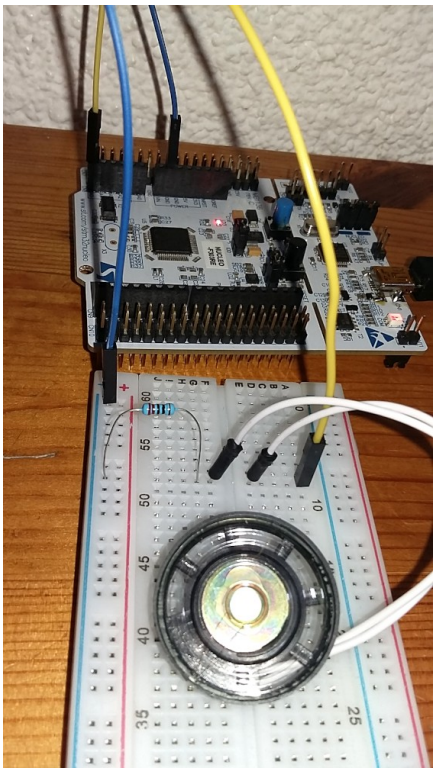


*Abbildung 65: Lautsprecher an PC0*

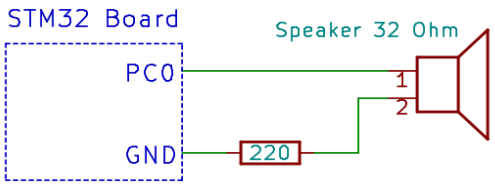



*Abbildung 66: Lautsprecher an PC0*

Der Widerstand ist nötig, damit der Ausgang des Mikrocontrollers nicht überlastet wird. Ohne Widerstand würde der Lautsprecher zu viel Strom auf nehmen.

Die Stromstärke ergibt sich aus der Spannung geteilt durch den Gesamt-Widerstand (220 Ω + 32 Ω), also

3,3 V / 252 Ω = 0,013 A = 13 mA

Der Mikrocontroller verträgt bis zu 25 mA, das passt also.

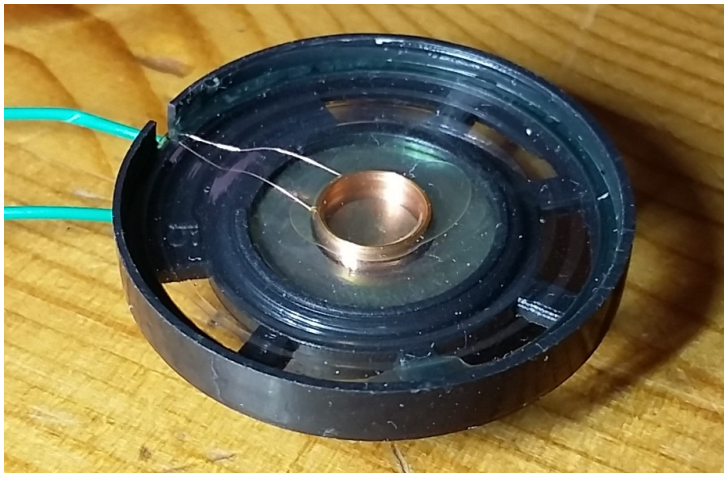


*Abbildung 67: Ansicht der Spule, die den Lautsprecher antreibt*

Der Lautsprecher besteht aus einer Membran, die mit einer Spule aus Kupferdraht verklebt ist. Die Spule taucht in das Feld eines Magneten ein. Wenn sie von Strom durchflossen wird, erzeugt sie selbst auch ein Magnetfeld, was je nach Stromrichtung entweder zu einer Anziehung oder Abstoßung führt. Indem man den Strom wiederholt ein und aus schaltet, regt man den Lautsprecher zu einer Schwingung an. Es entsteht ein hörbarer Ton.

Probiere das aus, indem du ein neues Programm auf Basis der „Blinker“ Vorlage erstellst und dann diesen Quelltext kopierst. Initialisierung des Ausgangs:

```
void init_io()
{
    ...

    // Port C einschalten
    SET_BIT(RCC->AHBENR, RCC_AHBENR_GPIOCEN);

    // Konfiguriere PC0 = Ausgang für den Lautsprecher
                MODIFY_REG(GPIOC->MODER,    GPIO_MODER_MODER0,    0b01    <<
GPIO_MODER_MODER0_Pos);
}
```

Und das Hauptprogramm:

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    // Initialisiere den System-Timer
    SysTick_Config(SystemCoreClock/1000);

    while(1)
    {
        // Lautsprecher an PC0 auf High (Strom an)
        WRITE_REG(GPIOC->BSRR, GPIO_BSRR_BS_0);

        // Warte eine Millisekunde
        delay_ms(1);

        // Lautsprecher an PC0 auf Low (Strom aus)
        WRITE_REG(GPIOC->BSRR, GPIO_BSRR_BR_0);

        // Warte eine Millisekunde
```

```
        delay_ms(1);
    }
}
```

Du wirst einen Ton mit einer Frequenz von 500 Hertz hören. Wenn du den Lautsprecher wie im folgenden Foto in der Hand hältst, wird er viel lauter:

*Abbildung 68: Lautsprecher in der Hand gehalten*

Das kommt daher, dass der Lautsprecher die Luft auf seiner Vorderseite drückt und hinten ansaugt (oder umgekehrt). Die Hand verhindert, dass die umgekehrt polarisierte Schallwelle der Rückseite nach vorne gelangt. Aus diesem Grund baut man Lautsprecher normalerweise in Gehäuse ein, anstatt sie einfach offen auf den Tisch zu legen.

Der Membran wird jeweils 1ms gedrückt und wieder losgelassen. Das macht zusammen 2 ms. Die Frequenz berechnet man aus 1 geteilt durch diese Zeit, also:

F = 1 / 2 ms = 1 / 0,002 s = 500 Hz

Nun wollen wir Töne mit höheren Frequenzen erzeugen. Da der Systemtimer nur Intervalle von Millisekunden zählt, eignet er sich nicht, um Töne mit anderen Frequenzen zu erzeugen. Stattdessen bauen wir kleine Verzögerungsscheifen, die 300 mal den NOP Befehl ausführen. Das ist ein besonderer Befehl der einfach nichts tut.

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    // Initialisiere den System-Timer
    SysTick_Config(SystemCoreClock/1000);

    while(1)
    {
        // Lautsprecher an PC0 auf High (Strom an)
        WRITE_REG(GPIOC->BSRR, GPIO_BSRR_BS_0);

        // Warte ein bisschen
        for (int j=0; j < 300; j++)
        {
            __NOP ();
        }

        // Lautsprecher an PC0 auf Low (Strom aus)
        WRITE_REG(GPIOC->BSRR, GPIO_BSRR_BR_0);
```

```
        // Warte ein bisschen
        for (int j=0; j < 300; j++)
        {
            __NOP ();
        }
    }
}
```

Probiere das Programm aus. Der Ton ist jetzt deutlich höher.

Wenn die for-Schleifen leer gewesen wäre, hätte der Compiler sie für nutzlos gehalten und einfach komplett weg optimiert. Das verhindere ich mit dem NOP Befehl. Dieser Befehl macht nichts, außer ein kleines bisschen Zeit zu vertrödeln.

Schalte jetzt mal die Entwicklungsumgebung (neben dem Hammer) auf den Release Modus um. Compiliere es und lasse das Programm dann erneut laufen. Der Ton hat nun eine deutlich höhere Frequenz! Das kommt daher, dass der C-Compiler im Release-Modul kompakteren schnelleren Code erzeugt. Das Programm läuft schneller, so dass die Warteschleifen schneller zu Ende sind. Der Lautsprecher schwingt daher auch schneller.

Stelle die IDE wieder auf den Debug-Modus zurück.

Nach diesen Vor-Experimenten kannst du nun eine Funktion schreiben, die Töne mit beliebigen Frequenzen erzeugt:

```
// Erzeuge einen Ton mit der angegebenen Frequenz und Dauer
void ton(uint32_t frequenz, uint32_t millisekunden)
{
    int wartezeit=610000/frequenz;
    uint32_t start=systick_count;

    // Wiederhole so und so viele Millisekunden lang
    while (systick_count-start < millisekunden)
    {
        // Lautsprecher an PC0 auf High (Strom an)
        WRITE_REG(GPIOC->BSRR, GPIO_BSRR_BS_0);

        // Warte
        for (uint32_t j=0; j < wartezeit; j++)
        {
            __NOP ();
        }

        // Lautsprecher an PC0 auf Low (Strom aus)
        WRITE_REG(GPIOC->BSRR, GPIO_BSRR_BR_0);

        // Warte
        for (uint32_t j=0; j < wartezeit; j++)
        {
            __NOP ();
        }
    }
}
```

Probiere die Funktion mit folgendem Hauptprogramm aus:

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    // Initialisiere den System-Timer
    SysTick_Config(SystemCoreClock/1000);
```

```
    while(1)
    {
        ton(600,500);
        ton(800,500);
    }
}
```

Das Ergebnis ist ein Tatü-Tata Ton. Für eine Eieruhr ist jedoch sicher ein anderer Soundeffekt angemessener. Ich mache mal einen Vorschlag:

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    // Initialisiere den System-Timer
    SysTick_Config(SystemCoreClock/1000);

    while(1)
    {
        ton(600,50);
        ton(800,50);
        ton(1000,50);
        ton(1200,50);
    }
}
```

Nicht gut? Dann denke dir einen eigenen Soundeffekt aus.

Jetzt kommt noch der 3-Minuten Timer davor, und fertig ist die Eieruhr:

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    // Initialisiere den System-Timer
    SysTick_Config(SystemCoreClock/1000);

    uint32_t start=systick_count;

    // Wiederhole so lange, bis 3 Minuten abgelaufen sind
    while (systick_count-start < 3*60*1000 )
    {
      // Einmal kurz Piepsen
      ton(1000,50);

      // Warte 1 Sekunde
      delay_ms(1000)
    }

    // Und jetzt kommt der ultimative Sound-Effekt :-)
    while(1)
    {
        // 16 mal Trillern
        for (uint32_t i=0; i<16; i++)
        {
            ton(600,50);
            ton(800,50);
            ton(1000,50);
            ton(1200,50);
            ton(600,50);
            ton(1200,50);
            ton(1800,50);
```

```
            ton(2400,50);
        }

        // Ansteigend 200 - 4000 Hz
        for (uint32_t i=200; i <= 4000; i=i+200)
        {
            ton(i,50);
        }

        // Abfallend 4000 - 200 Hz
        for (uint32_t i=4000; i >= 200; i=i-200)
        {
            ton(i,50);
        }
    }
}
```

Damit du dich während der drei Minuten Wartezeit nicht allzu sehr langweilst, habe ich da auch noch einen Ton mit eingebaut. Lass ihn mal laufen. Herrlich, nicht wahr?

Übungsaufgabe:

Baue das Programm so um, wie es dir persönlich am besten gefällt. Du könntest die Soundeffekte ändern oder die Zeit für einen anderen Anwendungsfall anpassen. Zum Beispiel als Zahnputz-Uhr.

## 8.3 Lichtschranke

Die Lichtschranke reagiert auf Unterbrechung eines Lichtstrahls. Damit könnte man zum Beispiel eine Klingel ansteuern, oder einen Runden-Zähler für eine Rennbahn mit Modellautos bauen.

Der Aufbau ist mit dem obigen Dämmerungsschalter identisch:

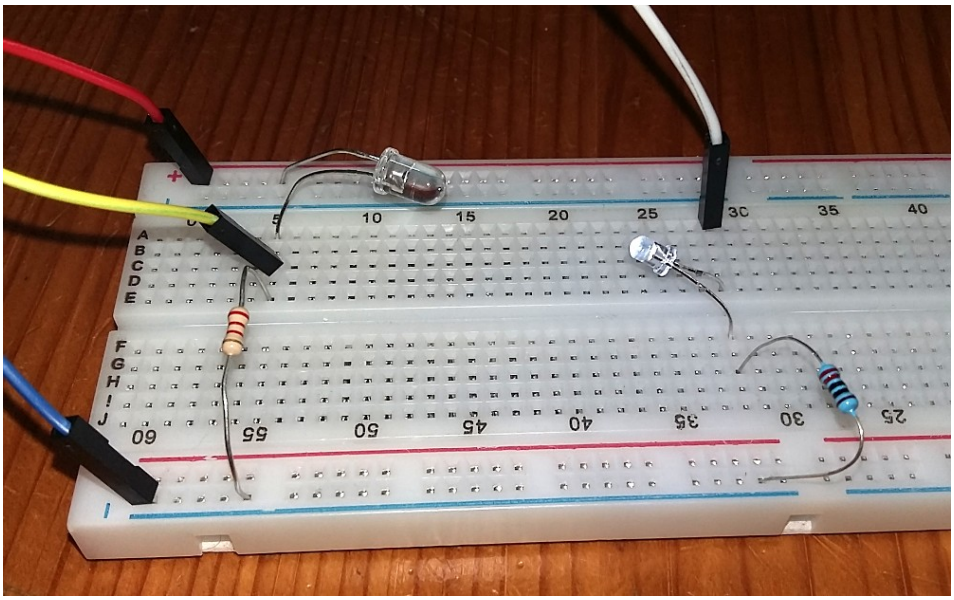


*Abbildung 69: Lichtschranke*

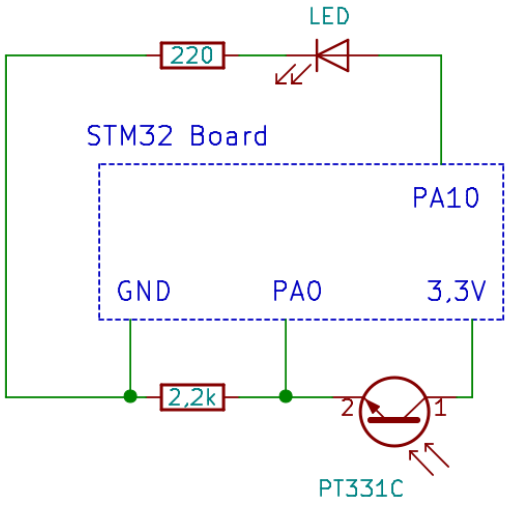



*Abbildung 70: Lichtschranke*

Richte die Leuchtdiode so aus, dass sie auf den den Lichtsensor zeigt. Folgendes soll passieren: Wenn man den Lichtstrahl unterbricht, dann soll die LED auf dem Board aus gehen. Wenn man den Lichtstrahl wieder frei gibt, soll die LED wieder an gehen. Alaos genau das umgekehrte Verhalten, als wie beim Dämmerungsschalter.

Lege ein neues Projekt auf Basis der „Blinker“ Vorlage an. Kopiere wieder die beiden Funktionen „init_analog“ und „read_analog“ aus dem Kapitel „Analoge Eingänge“. Erweitere die „init_io“ Funktion, damit die beiden Anschlüsse für die LED und den Fototransistor richtig konfiguriert sind:

```
// Konfiguriere PA0 als analogen Eingang für ADC1 IN1
MODIFY_REG(GPIOA->MODER, GPIO_MODER_MODER0, 0b11 << GPIO_MODER_MODER0_Pos);

// Konfiguriere PA10 = Ausgang für die weiße LED
MODIFY_REG(GPIOA->MODER, GPIO_MODER_MODER10, 0b01 << GPIO_MODER_MODER10_Pos);
```

Einfach nur zwischen Hell und dunkel zu unterscheiden, würde nicht genügen. Denn die Helligkeit im Raum hängt ja sehr stark von der Tageszeit ab. Die Lichtschranke soll aber nur auf Änderungen reagieren, die von der Unterbrechung des Lichtstrahls her kommen. Deswegen soll das Programm zusätzlich die Helligkeit der Umgebung messen und heraus rechnen. Das macht die folgende Funktion:

```
int32_t messen()
{
    // weiße LED aus
    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_10);

    // Warte
    delay_ms(200);

    // Umgebungslicht messen
    uint32_t umgebung=read_analog(1);
    printf("Umgebung %lu \n",umgebung);

    // weiße LED an
    WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_10);

    // Warte
    delay_ms(200);

    // Helligkeit messen
    uint32_t helligkeit=read_analog(1);
    printf("Helligkeit %lu \n",helligkeit);


    // Differenz berechnen
    int32_t differenz=helligkeit-umgebung;
    printf("Differenz %ld \n",differenz);

    return differenz;
}
```

Die Messung der Helligkeit geschieht hier in zwei Schritten:

1. Während die weiße LED ausgeschaltet ist, wird die Helligkeit der Umgebung gemessen.
2. Während die weiße LED eingeschaltet ist, wird die Helligkeit des Lichtstrahls gemessen (was das Umgebungslicht mit beinhaltet).

Danach wird die Differenz gebildet und zurück gegeben. Die „printf“ Ausgaben ermöglichen es dir, die Zahlen zu kontrollieren und nachzuvollziehen. Das Hauptprogramm ist jetzt ganz einfach:

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    // Initialisiere den System-Timer
    SysTick_Config(SystemCoreClock/1000);

    // Initialisiere den ADC
    init_analog();

    while(1)
    {
        if (messen() > 500)
        {
            // grüne LED ein
```

```
            WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BS_5);
        }
        else
        {
            // grüne LED aus
            WRITE_REG(GPIOA->BSRR, GPIO_BSRR_BR_5);
        }
    }
}
```

Die grüne LED auf dem Mikrocontroller-Board geht aus, sobald die Lichtschranke durch ein Hindernis (z.B. deine Hand) unterbrochen wird.

Probiere das Programm aus. Du kannst sehen, dass die weiße LED fortlaufend an und aus geschaltet wird. Während dessen erscheinen im Terminal-Programm die gemessenen Helligkeits-Werte.

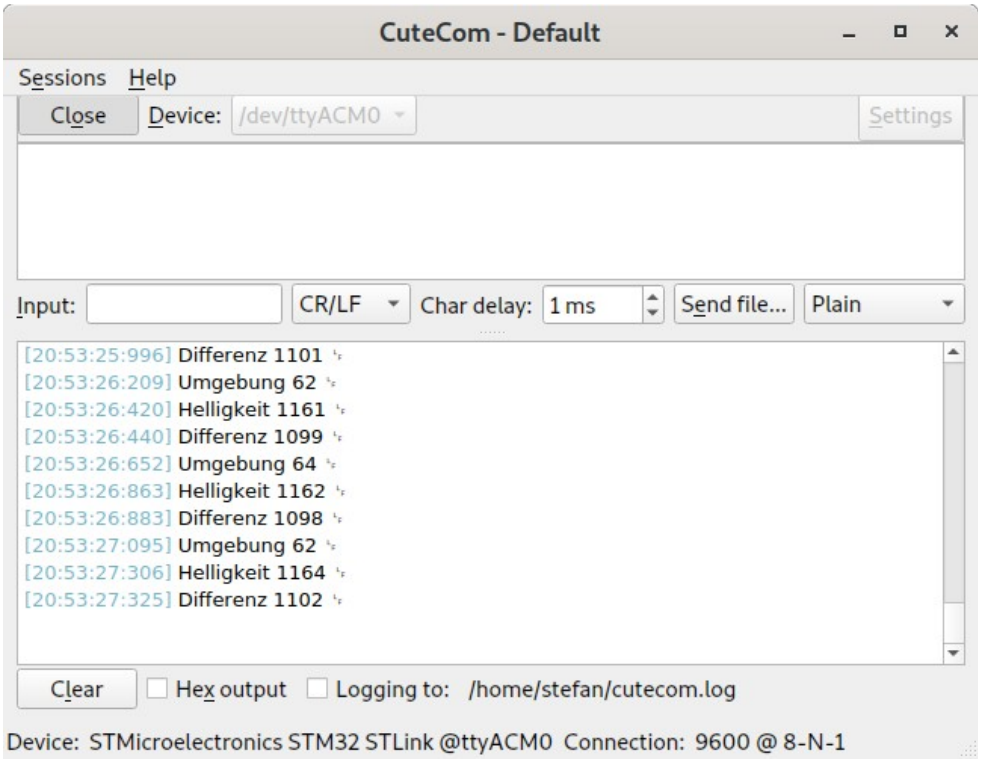



*Abbildung 71: Ausgabe in CuteCom*

Eine große Differenz bedeutet, dass viel Licht von der LED auf den Sensor fällt. Eine kleine Differenz bedeutet, dass wenig Licht von der LED auf den Sensor fällt. Dann ist die Lichtschranke eindeutig unterbrochen.

Wenn du möchtest, kannst du die Wartezeiten nun von 200 ms auf 2ms verringern. Das schnellere Blinken der weißen LED kann man dann nicht mehr wahrnehmen und die Lichtschranke reagiert flotter. Allerdings solltest du dann die printf() Ausgaben weg lassen, weil man sie so schnell nicht mehr mitlesen kann und die Ausgaben das Programm unnötig ausbremsen würden.

Übungsaufgabe:

Füge den Lautsprecher hinzu und sorge dafür, dass er einen passenden Soundeffekt macht, wenn man den Lichtstrahl unterbricht.

## 8.4 Raum-Thermometer

In diesem Experiment nutzen wir einen Temperatur-Sensor um die Temperatur deines Zimmers zu kontrollieren. Drei LEDs zeigen das Messergebnis an:

- rot = es ist zu warm
- grün = die Temperatur ist Ok
- blau = es ist zu kalt

*Abbildung 72: Heißleiter*

Als Sensor verwenden wir einen sogenannten „Heißleiter“ oder „Thermistor“. Abgekürzt nennt man das Ding „NTC“, was „Negative Temperature Coefficient“ bedeutet. Der Heißleiter ist ein Widerstand, dessen Leitfähigkeit mit steigender Temperatur zunimmt. Er wird den analogen Eingang PA0 hoch ziehen. Je wärmer es ist ist, umso höher wird die Spannung sein.

Aufbau der Schaltung:

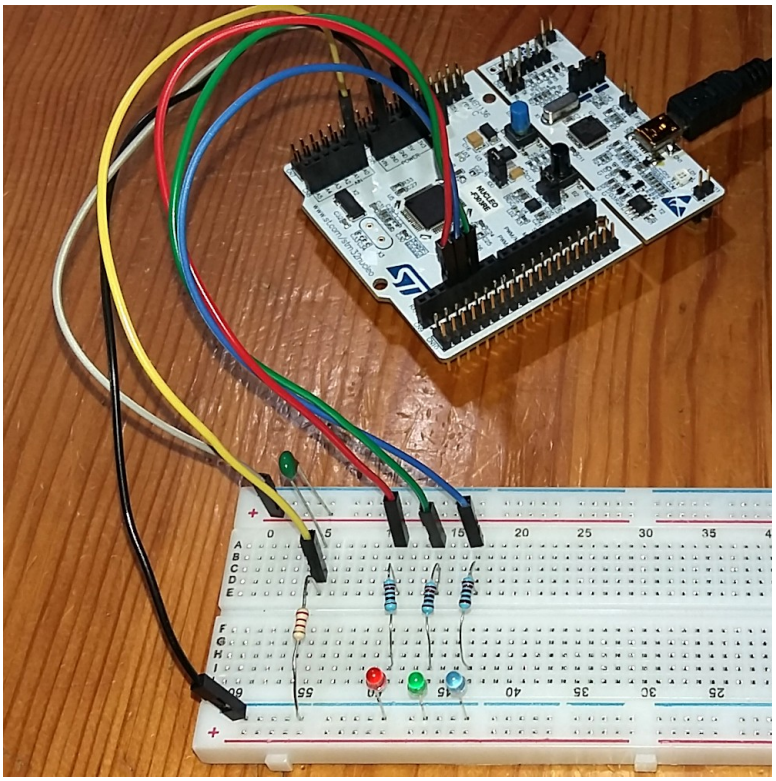


*Abbildung 73: Raum-Thermometer*

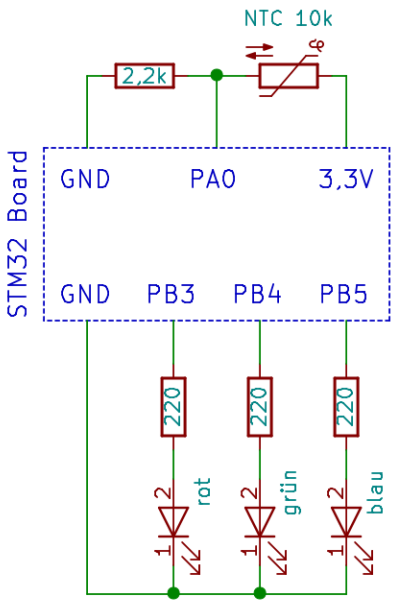



*Abbildung 74: Raum-Thermometer*

Lege ein neues Projekt auf Basis der „Blinker“ Vorlage an. Kopiere die beiden Funktionen „init_analog“ und „read_analog“ aus dem Kapitel „Analoge Eingänge“. Erweitere die „init_io“ Funktion, damit die Anschlüsse für den Temperatursensor und die LEDs richtig konfiguriert sind:

```
// Konfiguriere PA0 als analogen Eingang für ADC1 IN1
MODIFY_REG(GPIOA->MODER, GPIO_MODER_MODER0, 0b11 << GPIO_MODER_MODER0_Pos);

// Port B einschalten
SET_BIT(RCC->AHBENR, RCC_AHBENR_GPIOBEN);

// PB3 = Ausgang rote LED
MODIFY_REG(GPIOB->MODER, GPIO_MODER_MODER3, 0b01 << GPIO_MODER_MODER3_Pos);

// PB4 = Ausgang rote LED
MODIFY_REG(GPIOB->MODER, GPIO_MODER_MODER4, 0b01 << GPIO_MODER_MODER4_Pos);

// PB5 = Ausgang rote LED
MODIFY_REG(GPIOB->MODER, GPIO_MODER_MODER5, 0b01 << GPIO_MODER_MODER5_Pos);
```

Das Hauptprogramm sieht so aus:

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    // Initialisiere den System-Timer
    SysTick_Config(SystemCoreClock/1000);

    // Initialisiere den ADC
    init_analog();

    while(1)
        {
            // messen
            uint32_t temperatur=read_analog(1);
            printf("Temperatur %lu \n",temperatur);

            if (temperatur>800)
            {
                // blau=aus, grün=aus, rot=an
                WRITE_REG(GPIOB->BSRR,
                    GPIO_BSRR_BR_5 + GPIO_BSRR_BR_4 + GPIO_BSRR_BS_3);
            }
            else if (temperatur>700)
            {
                // blau=aus, grün=an, rot=aus
                WRITE_REG(GPIOB->BSRR,
                    GPIO_BSRR_BR_5 + GPIO_BSRR_BS_4 + GPIO_BSRR_BR_3);
            }
            else
            {
                // blau=an, grün=aus, rot=aus
                WRITE_REG(GPIOB->BSRR,
                    GPIO_BSRR_BS_5 + GPIO_BSRR_BR_4 + GPIO_BSRR_BR_3);
            }

            // Warte 1 Sekunde
            delay_ms(1000);
        }
}
```

Schau dir die Ausgaben im Terminal-Programm an. Bei mir wird der Messwert 772 angezeigt, deswegen habe ich die Schaltschwellen ein bisschen darunter und darüber eingestellt (700 und 800). Die musst die Schwellwerte an deine Raumtemperatur anpassen. Danach werden die Leuchtdioden korrekt „zu warm“ und „zu kalt“ anzeigen.

Am Ende habe ich eine Warteschleife eingefügt, damit das Terminal-Programm nicht durch zu viele Ausgaben überflutet wird.

Übungsaufgabe:

Baue die Schaltung und das Programm so um, dass es die Helligkeit misst und im Terminal-Fenster „zu hell“ bzw. „zu dunkel“ ausgibt.

## 8.5 Kühlschrank-Alarm

Der Kühl-Schrank Alarm erzeugt einen Signal-Ton, wenn der Kühlschrank für mehr als 20 Sekunden zu warm wird oder wenn die Türe nicht richtig geschlossen ist. Wir verwenden wieder den Heißleiter als Temperatursensor. Den Reed-Kontakt benutzen wir, um die Türe zu kontrollieren.

*Abbildung 75: Reed-Kontakt*

In dem Reed-Kontakt befinden sich zwei Zungen aus Metall, die durch einen Magnet dazu gebracht werden, sich gegenseitig anzuziehen.

Aufbau der Schaltung:

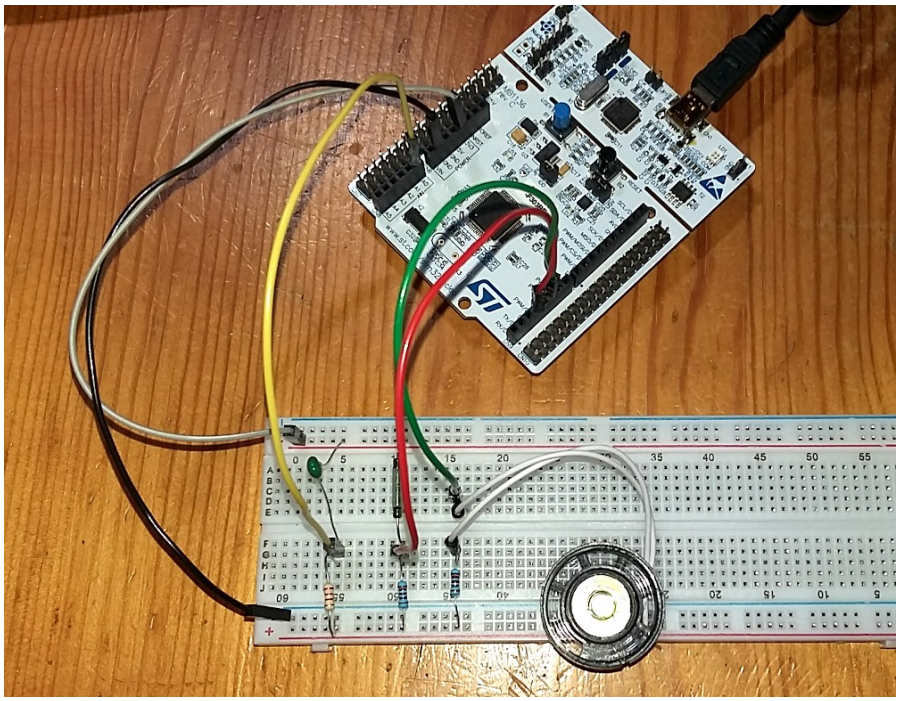


*Abbildung 76: Kühlschrank-Alarm*

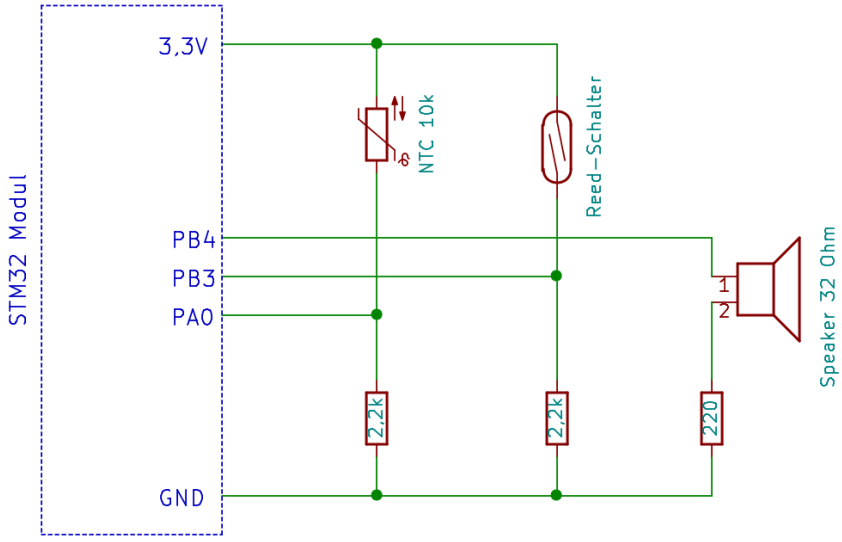



*Abbildung 77: Kühlschrank-Alarm*

Die Eingangsspannung an PA0 ist niedrig wenn der Kühlschrank kalt genug ist. Sie steigt an, wenn der Kühlschrank wärmer wird.

Der Eingang PB3 wird vom Reed-Kontakt auf High Pegel (3,3V) gezogen, wenn ein Magnet daneben liegt. Im realen Aufbau würdest du den Reed-Kontakt am Gehäuse des Kühlschrankes befestigen und den Magneten an der Türe. Wenn man die Türe öffnet, entfernt sich der Magnet, so dass der Reed-Kontakt öffnet. Solche Kontakte werden häufig auch bei Alarmanlagen verwendet.

In unserem Experiment stecken wir die Teile jedoch einfach nur auf das Steckbrett und legen den Magneten (bzw. den Lautsprecher) dicht daneben. Beim Biegen der Anschlüsse des Reed-Kontaktes musst du sehr vorsichtig sein, damit das gläserne Gehäuse nicht zerbricht.

Lege ein neues Projekt auf Basis der „Blinker“ Vorlage an. Kopiere die beiden Funktionen „init_analog“ und „read_analog“ aus dem Kapitel „Analoge Eingänge“. Erweitere die „init_io“ Funktion, damit die Anschlüsse richtig konfiguriert sind:

```
// Konfiguriere PA0 als analogen Eingang für ADC1 IN1
MODIFY_REG(GPIOA->MODER, GPIO_MODER_MODER0, 0b11 << GPIO_MODER_MODER0_Pos);

// Port B einschalten
SET_BIT(RCC->AHBENR, RCC_AHBENR_GPIOBEN);

// PB4 = Ausgang Lautsprecher
MODIFY_REG(GPIOB->MODER, GPIO_MODER_MODER4, 0b01 << GPIO_MODER_MODER4_Pos);
```

Hier ist keine Zeile für PB3 nötig, weil (fast) alle Pins schon Standardmäßig als Eingang konfiguriert sind. Die Funktion zur Tonausgabe auf den Lautsprecher kennst du schon, aber die steuert dieses mal einen anderen Ausgangs-Pin an:

```
// Erzeuge einen Ton mit der angegebenen Frequenz und Dauer
void ton(uint32_t frequenz, uint32_t millisekunden)
{
    int wartezeit=610000/frequenz;
    uint32_t start=systick_count;

    // Wiederhole so und so viele Millisekunden lang
    while (systick_count-start < millisekunden)
    {
        // Lautsprecher an PB4 auf High (Strom an)
        WRITE_REG(GPIOB->BSRR, GPIO_BSRR_BS_4);

        // Warte
        for (uint32_t j=0; j < wartezeit; j++)
        {
            __NOP ();
        }

        // Lautsprecher an PB4 auf Low (Strom aus)
        WRITE_REG(GPIOB->BSRR, GPIO_BSRR_BR_4);

        // Warte
        for (uint32_t j=0; j < wartezeit; j++)
        {
            __NOP ();
        }
    }
}
```

Das Hauptprogramm sieht so aus:

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    // Initialisiere den System-Timer
    SysTick_Config(SystemCoreClock/1000);

    // Initialisiere den ADC
    init_analog();

    // Zeitpunkt, wann zuletzt alles Ok war (Türe zu und Temperatur niedrig)
```

```
    uint32_t okZeit=0;

    while(1)
    {
        // messen
        uint32_t temperatur=read_analog(1);
        printf("Temperatur %lu \n",temperatur);

        uint32_t reedKontakt=READ_BIT(GPIOB->IDR, GPIO_IDR_3);
        if (reedKontakt>0)
        {
            puts("Tuer ist zu");
        }
        else
        {
            puts("Tuer ist auf");
        }

        // Ist es kalt genug und ist die Türe geschlossen?
        if (temperatur<800 && reedKontakt>0)
        {
            // Kein Problem
            okZeit=systick_count;

            // grüne LED aus
            WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR_5);
        }
        else
        {
            // grüne  LED an
            WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS_5);

            // Besteht das Problem seit mehr als 20 Sekunden?
            if (systick_count - okZeit > 20000)
            {
                puts("Alarm!");
                ton(1000, 500);
            }
        }

        // Warte 1 Sekunde
        delay_ms(1000);
    }
}
```

Lege für den Anfang einen Magneten neben den Reed-Kontakt (Tipp: Der Lautsprecher ist magnetisch). Starte das Programm und beobachte die Ausgaben im Terminal.

Zum bequemeren Experimentieren kannst du einfach mal so tun, als würde sich dein Arbeitstisch im Kühlschrank befinden. Die aktuelle Zimmertemperatur soll also als „Ok“ gewertet werden. Wahrscheinlich musst du die Schaltschwelle der Temperatur (800) anpassen, so dass sie knapp über den tatsächlichen Messwerten liegt.

Die LED auf dem Board müsste jetzt aus sein, was „Alles Ok“ bedeutet. Fasse nun den Heißleiter mit deinen warmen Fingern an, um einen defekten Kühlschrank zu simulieren. Zuerst geht die grüne LED an. Wenn du 20 Sekunden länger abwartest, gibt der Lautsprecher zusätzlich einen Alarm-Ton von sich. Beobachte dabei die Ausgaben im Terminal-Programm.

Lasse den Temperatursensor wieder abkühlen. Die LED geht dann aus und der Alarm-Ton verstummt. Jetzt simuliere eine offene Türe, indem du den Magneten entfernst. Wieder geht die LED an und 20 Sekunden später ertönt auch der Alarm.

Übungsaufgabe:

Der Alarm soll nur bei zu hoher Temperatur ertönen, und erst nach 120 Sekunden. Eine offene Türe soll nur optisch durch die LED angezeigt werden. Ändere den Quelltext entsprechend.

## 8.6 Quiz-Buzzer

Der Quiz-Buzzer ist für Ratespiele gedacht. Ein Moderator richtet seine Quiz-Fragen an bis zu vier Spieler. Wer die Antwort kennt, drückt auf einen Knopf. Der schnellste Spieler darf antworten und gewinnt somit Punkte.

Die Schaltung muss also vier Taster bekommen und erkennen, welcher Taster als erstes gedrückt wurde. Zusätzlich soll ein kurzer Soundeffekt zu hören sein.

Ich schlage folgenden Aufbau vor:

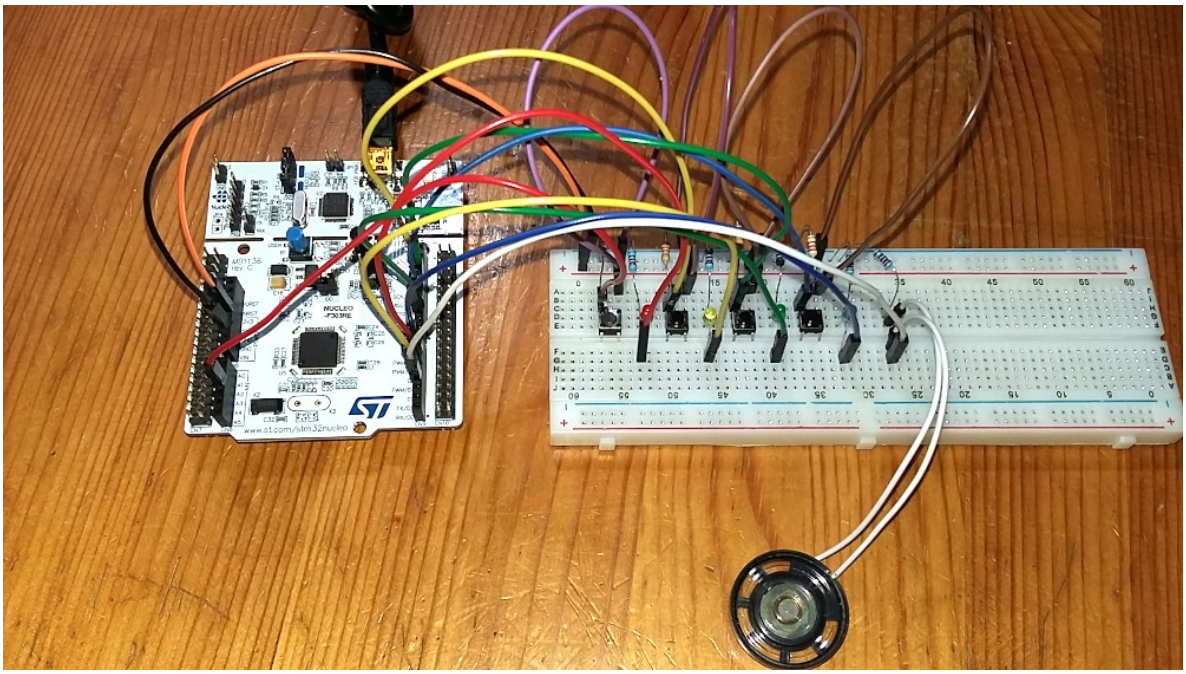


*Abbildung 78: Quiz-Buzzer*

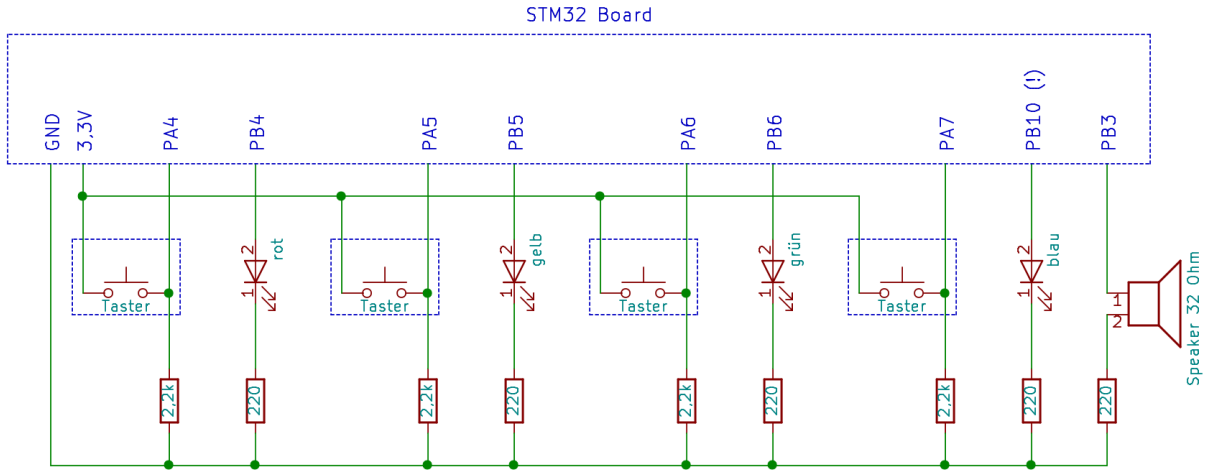



*Abbildung 79: Quiz-Buzzer*

Vergiss beim aufbauen nicht, dass die horizontalen Schienen zur Strom-Verteilung eventuell in der Mitte unterbrochen sind!

Lege ein neues Projekt auf Basis der „Blinker“ Vorlage an. Bei der Init-Funktion müssen wir die Zeile entfernen, die PA5 als Ausgang konfiguriert, weil wir diesen Pin nun als Eingang benutzen.

```
void init_io()
{
    // Port A einschalten
    SET_BIT(RCC->AHBENR, RCC_AHBENR_GPIOAEN);

    // PA4, PA5, PA6 und PA7 werden als Eingänge für die Taster benutzt

    // Port B einschalten
    SET_BIT(RCC->AHBENR, RCC_AHBENR_GPIOBEN);

    // PB3 = Ausgang Lautsprecher
                MODIFY_REG(GPIOB->MODER,    GPIO_MODER_MODER3,    0b01    <<
GPIO_MODER_MODER3_Pos);

    // PB4 = Ausgang rote LED
                MODIFY_REG(GPIOB->MODER,    GPIO_MODER_MODER4,    0b01    <<
GPIO_MODER_MODER4_Pos);

    // PB5 = Ausgang gelbe LED
                MODIFY_REG(GPIOB->MODER,    GPIO_MODER_MODER5,    0b01    <<
GPIO_MODER_MODER5_Pos);

    // PB6 = Ausgang grüne LED
                MODIFY_REG(GPIOB->MODER,    GPIO_MODER_MODER6,    0b01    <<
GPIO_MODER_MODER6_Pos);

    // PB10 = Ausgang blaue LED
                        MODIFY_REG(GPIOB->MODER,      GPIO_MODER_MODER10,0b01
<<GPIO_MODER_MODER10_Pos);
}
```

Die ton() Funktion steuert den Lautsprecher an PB3 an:

```
// Erzeuge einen Ton mit der angegebenen Frequenz und Dauer
void ton(uint32_t frequenz, uint32_t millisekunden)
{
    int wartezeit=610000/frequenz;
```

```
    uint32_t start=systick_count;

    // Wiederhole so und so viele Millisekunden lang
    while (systick_count-start < millisekunden)
    {
        // Lautsprecher an PB3 auf High (Strom an)
        WRITE_REG(GPIOB->BSRR, GPIO_BSRR_BS_3);

        // Warte
        for (uint32_t j=0; j < wartezeit; j++)
        {
            __NOP ();
        }

        // Lautsprecher an PB3 auf Low (Strom aus)
        WRITE_REG(GPIOB->BSRR, GPIO_BSRR_BR_3);

        // Warte
        for (uint32_t j=0; j < wartezeit; j++)
        {
            __NOP ();
        }
    }
}
```

Das Haupt-Programm kann man so schreiben:

```
int main()
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

    // Initialisiere den System-Timer
    SysTick_Config(SystemCoreClock/1000);


    // Warte bis ein Buzzer gedrückt wird
    while(1)
    {

        // Wenn Spieler "rot" gedrückt hat
        if (READ_BIT(GPIOA->IDR, GPIO_IDR_4))
        {
            // rote LED an
            WRITE_REG(GPIOB->BSRR,GPIO_BSRR_BS_4);
            break;
        }

        // Wenn Spieler "gelb" gedrückt hat
        if (READ_BIT(GPIOA->IDR, GPIO_IDR_5))
        {
            // gelbe LED an
            WRITE_REG(GPIOB->BSRR,GPIO_BSRR_BS_5);
            break;
        }

        // Wenn Spieler "grün" gedrückt hat
        if (READ_BIT(GPIOA->IDR, GPIO_IDR_6))
        {
            // gelbe LED an
            WRITE_REG(GPIOB->BSRR,GPIO_BSRR_BS_6);
            break;
        }
```

```
        // Wenn Spieler "blau" gedrückt hat
        if (READ_BIT(GPIOA->IDR, GPIO_IDR_7))
        {
            // blaue LED an
            WRITE_REG(GPIOB->BSRR,GPIO_BSRR_BS_10);
            break;
        }
    }

    // Eine Sekunde lang Sound-Effekt
    uint32_t start=systick_count;
    while(systick_count-start<1000)
    {
        ton(600,50);
        ton(800,50);
        ton(1000,50);
        ton(1200,50);
    }
}
```

In der ersten while-Schleife wartet das Programm darauf, dass ein Taster gedrückt wird. Wenn das der Fall ist, wir die zugehörige LED eingeschaltet und dann wird die Hauptschleife mit dem „break“ Befehl verlassen. Danach wird ein in der zweiten while-Schleife ein Sound-Effekt erzeugt, bevor das Programm endet. Der Moderator kann den Reset-Knopf (auf dem Mikrocontroller-Board) drücken, um das Programm für die nächste Quiz-Frage zurück zu setzen und neu zu starten.

Übungsaufgaben:

1. Spiele für jeden Button einen anderen Soundeffekt ab.
2. Während die Spieler über die Frage nachdenken, soll jede Sekunde ein Tickendes Geräusch zu hören sein.
3. Messe die Zeit, die vom Programmstart bis zum Drücken eines Buzzers verstreicht. Sorge dafür, dass diese Zeit zusammen mit der Farbe des Buzzers im Terminal-Programm ausgegeben wird.
4. Sorge dafür, dass ein Tastendruck nur dann etwas bewirkt, wenn er länger als 50ms andauert. Dadurch wird das Gerät bei langen Kabeln unempfindlich gegen Funk-Störungen.

### 8.6.1 Fester Aufbau

Falls du diese Schaltung zur echten dauerhaften Anwendung ausbauen möchtest, solltest du die ganzen Bauteile auf eine **Lochrasterplatine** löten. Mit einer **Lötstation** gelingt das besser, als mit einem ungeregeltem Lötkolben. Die Verbindungen zwischen den Bauteilen kann man gut mit **verzinntem** oder **versilbertem Draht** herstellen.

Das Mikrocontroller-Board würde ich durch ein kleineres ersetzen, dass man in **Buchsen-Leisten** stecken kann, damit man es notfalls austauschen kann.

Für ein lauteres Geräusch empfehle ich einen fertigen **Verstärker** zusammen mit einem größeren **Lautsprecher** zu verwenden. Kleine Verstärkermodule bekommt man im Internet ab 2 Euro. Ich denke, dass 1 Watt für den Hausgebrauch schon genügen.

Einen größeren Reset Taster kannst du an den Reset-Eingang des Mikrocontroller-Moduls anschließen. Er muss das Signal auf GND herunter ziehen.

Damit hast du die richtigen Stichwörter für den nächsten Einkauf.

# 9 Nachwort

Ich habe mich in diesem Buch sehr darum bemüht, den STM32 Mikrocontroller so einfach wie möglich darzustellen. Diese Mikrochips haben allerdings noch viel mehr Funktionen. Wenn man die alle verwenden möchte, wird es doch ziemlich kompliziert.

Wenn du Lust hast, etwas aufwändigeres mit STM32 zu bauen, dann schaue dir mal meine Binäruhr auf http://stefanfrings.de/binaeruhr/ an. Dort verwende ich einige Kniffe, die in meiner Infosammlung auf http://stefanfrings.de/stm32/ beschrieben sind.

Um weitere Bauteile kennen zu lernen, empfehle ich den Band 2 von meinem Buch „Einstieg in die Elektronik mit Mikrocontrollern“ http://stefanfrings.de/mikrocontroller_buch/index.html .

Im „Elektronik Kompendium“ http://www.elektronik-kompendium.de/ findest du zahlreiche Grundlagen der Elektronik anschaulich erklärt.

Das Buch „C von A bis Z“ http://openbook.rheinwerk-verlag.de/c_von_a_bis_z/ vermittelt Grundlagen der Programmiersprache C. Ich finde, dass man die Programmiersprache besser auf einem normalen PC oder Laptop (ohne Mikrocontroller) erlernen kann.

Folgende Händler kann ich empfehlen: tme.eu, reichelt.de, kessler-electronic.de, berrybase.de, lcsc.com, pololu.com, az-delivery.de, robotshop.com.

Manche Sachen gibt es auch günstig bei Amazon, Ebay und Aliexpress. Lieferungen aus Asien dauern allerdings meistens 1 bis 2 Monate und man bekommt von dort oft schlechte Fälschungen.