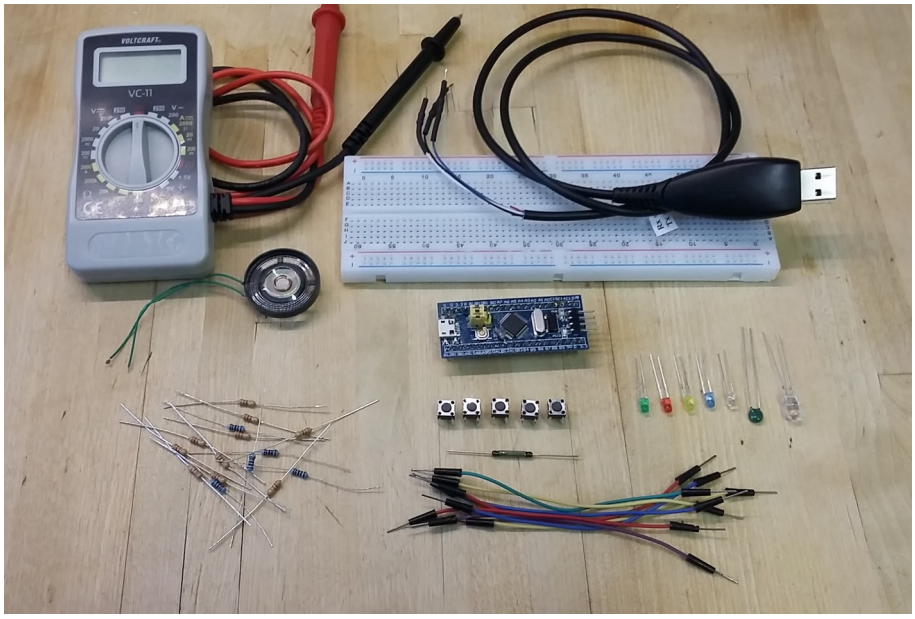
Einblick in die moderne Elektronik

ohne viel Theorie

von Stefan Frings



Inhaltsverzeichnis

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.

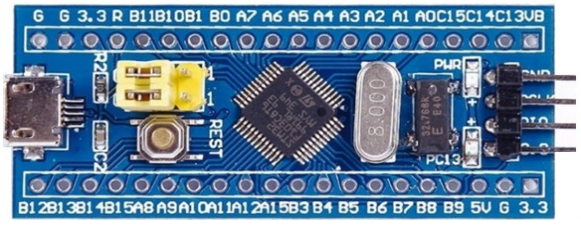
Anstatt mit theoretischen Grundlagen zu beginnen, machen wir es wie in der Fahrschule: Einfach losfahren...

Stefan Frings, November 2017

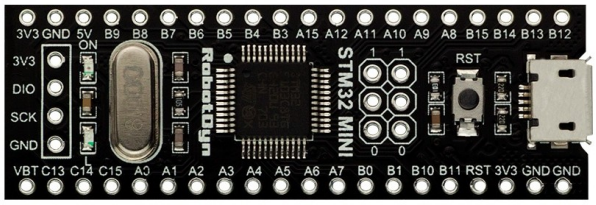
2 Material

Besorge das auf der Titelseite dargestellte Material:

Anzahl	**Artikel**
1	Ein „Blue Pill“ Board mit STM32F103C8T6
1	Steckbrett (Breadboard) mit ca. 830 Kontakten
20	Dupont Kabel Stecker-Stecker (Jumper Wire Male-Male)
1	USB-UART Kabel (USB seriell TTL Kabel)
1	Digital-Multimeter, das einfachste Modell genügt. Zum Beispiel ein Voltcraft VC-11
1	Kleiner Lautsprecher mit 32 Ohm (aus einem Kopfhörer)
5	Kurzhubtaster (tactile switch) 6x6mm (Höhe ist egal)
5	Standard Leuchtdioden 3 mm jeweils eine in den Farben rot, gelb, grün, blau und weiß
1	Heißleiter (NTC) 10k Ohm
1	Phototransistor PT331C
1	Reed Kontakt mit 1 Schließer, Größe ist egal
1	Kleiner Magnet, beliebige Form, um den Reed-Kontakt zu betätigen
5	Bedrahtete Widerstände 220 Ohm ¼ Watt
10	Bedrahtete Widerstände 2,2k Ohm ¼ Watt



Blue Pill Board



Kompatibles Board von RobotDyn

Achtung: Seit Anfang 2020 werden die Blue Pill Boards oft mit schlecht gefälschten Chips verkauft. Es gibt inzwischen bessere dazu kompatible Boards mit originalen STM32F103Cxxx wie das gezeigte schwarze Board.

Zur Stromversorgung verwenden wir ein Smartphone Netzteil mit USB Stecker. Die Programme wirst du auf einem Laptop oder Desktop PC mit Windows schreiben.

Du brauchst eventuell kurzzeitig Lötwerkzeug, um die Stiftleisten zu montieren und um Kabel an den Lautsprecher zu löten.

3 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:

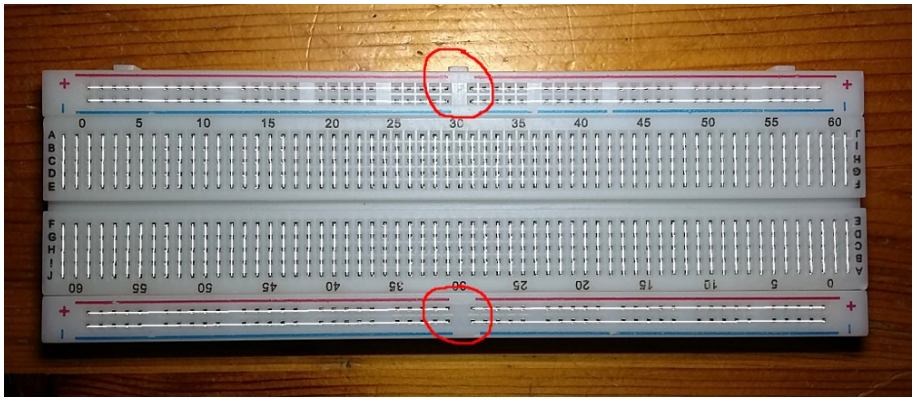


Abbildung 1: Steckbrett mit Hervorhebung der Verbindungen

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.

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

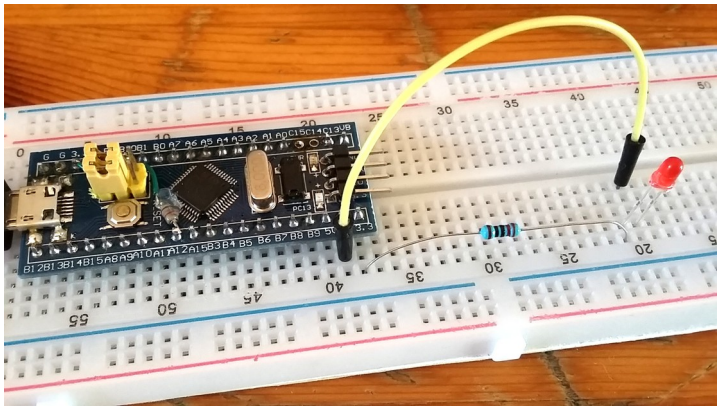


Abbildung 2: Aufbau mit LED und Widerstand

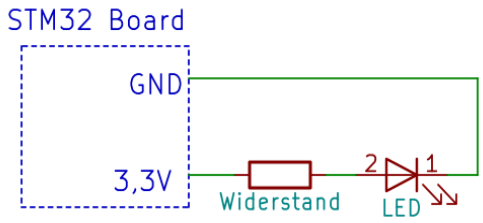


Abbildung 3: Schaltplan, LED und Widerstand

Ansicht von oben:

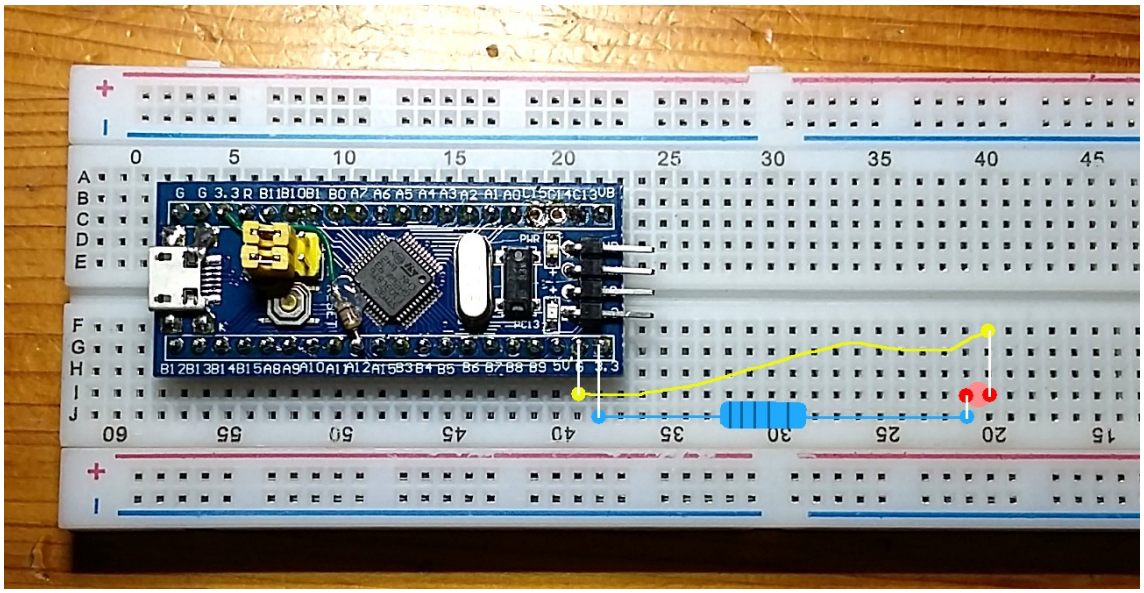


Abbildung 4: Aufbau mit LED und Widerstand, Positionierung der Bauteile

Stecke nun die Teile wie in den Bildern gezeigt in das Brett. Bei der LED gehört der längere Draht auf die linke Seite. Das Mikrocontroller-Board muss ganz in das Brett hinein gedrückt werden, so dass dessen Stifte von außen nicht mehr sichtbar sind.

Tipp: Neue Steckbretter enthalten oft sehr stramme Kontaktfedern. Drücke dann nur auf die Kanten des blauen Boardes, damit dessen Bauteile nicht zerbrechen. Eventuell magst du die Kontakte vorher mit einer stumpfen Nadel lockern.

Verbinde dein Smartphone Netzteil mit dem Mikrocontroller Board. Die LED sollte jetzt 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.

3.1 Leuchtdiode

LED ist die englische Abkürzung für „Light Emitting Diode“, auf Deutsch: Leuchtdiode. Sie leuchtet, wenn sie von Strom durchflossen 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 durchflossen wird. Allerdings muss man unbedingt dafür sorgen, dass nicht zu viel Strom fließt, weil er sonst überhitzt und geht kaputt.

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.

3.2 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 Omega „Ω“, manchmal auf „R“.

4 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:

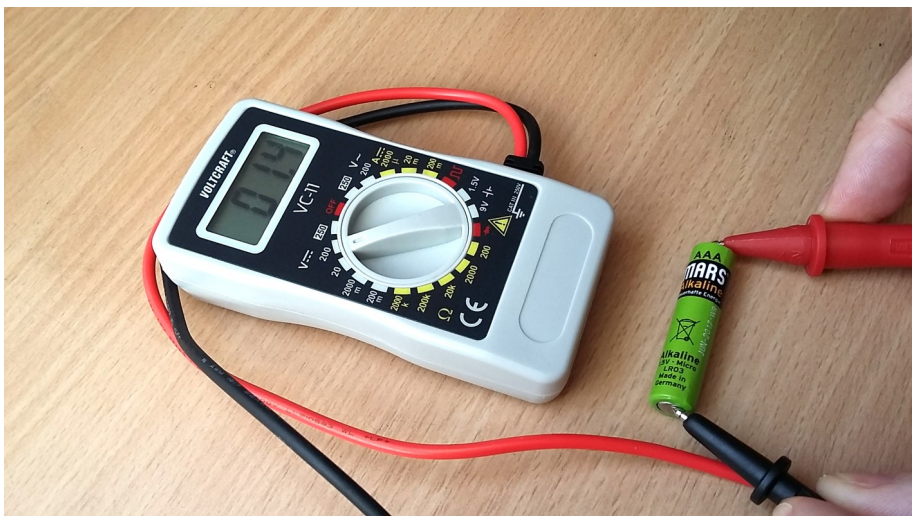


Abbildung 8: Messung der Spannung einer Batterie

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:

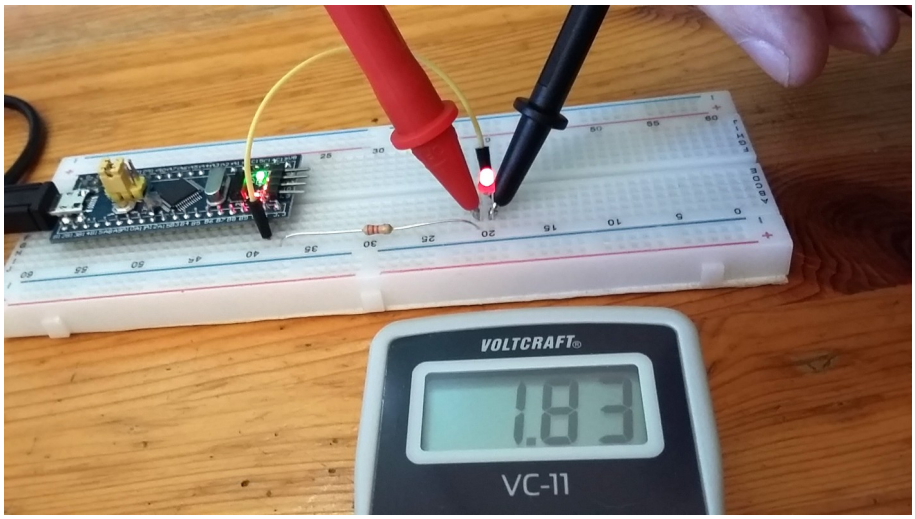


Abbildung 9: Messung der Stromstärke im Stromkreis einer LED

In diesem Foto habe ich den 2200 Ω Widerstand verwendet. Das Messgerät zeigt 1,83 V an.

Benutze nun den anderen 220 Ω Widerstand und messe die Spannung erneut. Bei mir ergibt das 1,95 V. Die LED leuchtet nun viel heller weil dieser Widerstand zehn mal mehr Strom fließen lässt. Aber ihre Betriebsspannung hat sich nur geringfügig verändert. Das ist eine ganz wichtige Eigenschaft, wo sich LEDs grundsätzlich anders verhalten, als Glühlampen.

Merke: 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).

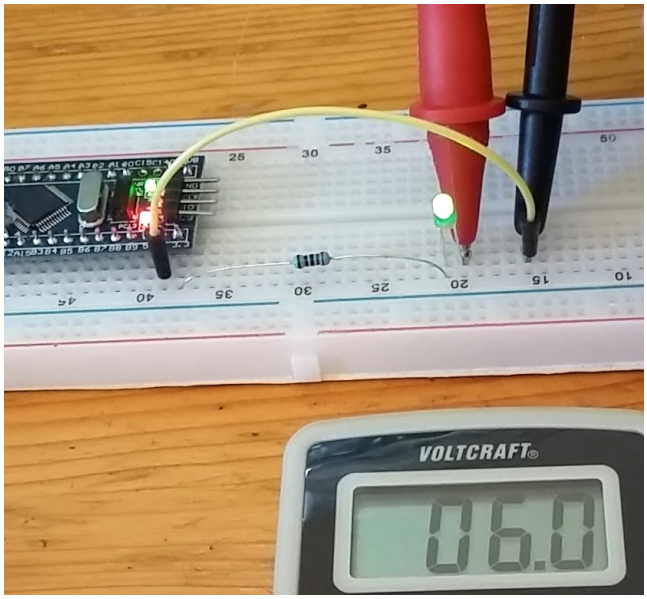


Abbildung 10: Messung der Stromstärke im Stromkreis einer LED

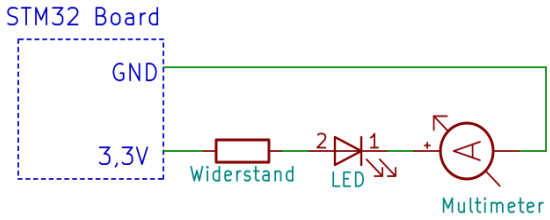


Abbildung 11: Schaltplan, Messung der Stromstärke im Stromkreis einer LED

Das Multimeter zeigt an, dass in diesem Stromkreis ein Strom mit der Stärke 6,0 mA (Milli-Ampere) fließt. Das war mit dem 220 Ω Widerstand.

Tausche den Widerstand aus. Bei mir fließt durch den 2200 Ω Widerstand eine Stromstärke von 0,6 mA. Du siehst: Da der Widerstand zehn mal so stark bremst, fließt nur ein Zehntel so viel Strom.

Merke: 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, indem 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.

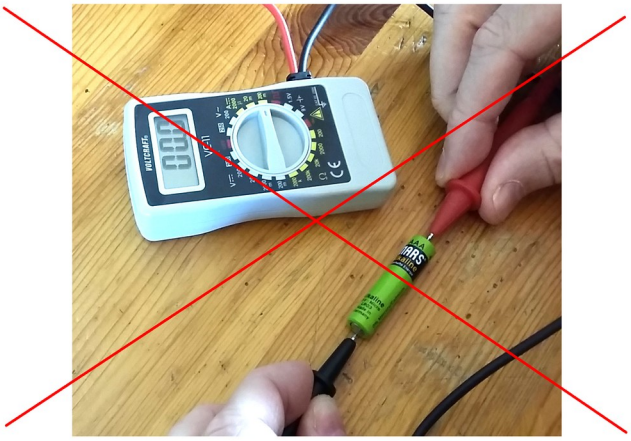


Abbildung 12: Falsche Verwendung des Multimeters bei der Messung der Stromstärke

5 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!

5.1 Stromkreis

Jetzt haben 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:

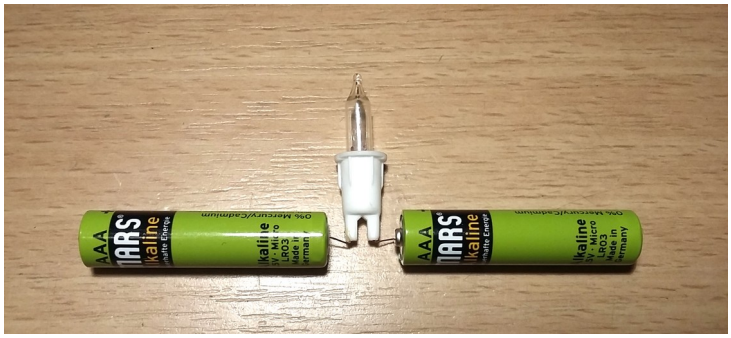


Abbildung 13: Offener Stromkreis (es fließt kein Strom)

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

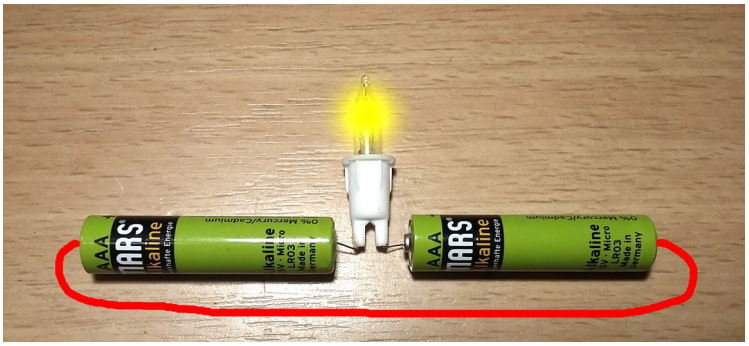


Abbildung 14: Geschlossener Stromkreis (der Strom fließt)

5.2 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:

Abbildung 15: Statisch geladener Junge aufgrund von Reibung mit der Rustsche (von Wikimedia)

Wenn du in so einem Zustand die Anschlüsse eines Mikrochips anfasst, geht er mit 99 % Wahrscheinlichkeit kaputt. Auch geringere Ladungen können bereits schädlich wirken.

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

6 USB-UART Kabel

Die USB Buchse des Mikrocontroller-Moduls verwenden wir in diesem Buch ausschließlich zur Stromversorgung! Um die USB Buchse zum Datenaustausch zu nutzen, bräuchte man ein komplexes Programm, das den Rahmen dieses Buches sprengen würde. Damit kannst du dich später befassen, wenn du willst (siehe http://stefanfrings.de/stm32/stm32f1.html#usb).

Wir benutzen zum Datenaustausch ein USB-UART Kabel. Der erste Anwendungsfall besteht darin, ein Beispielprogramm vom Laptop oder Desktop Computer in den Speicher des Mikrocontrollers zu übertragen.

Abbildung 16: USB-UART Adapterkabel

Stecke das Kabel in den Computer ein und installiere den Treiber. Das sollte normalerweise automatisch ablaufen. Falls nicht, musst du den Treiber selbst aus dem Internet herunter laden und installieren. Vermutlich wird einer der folgenden passen:

- für PL2303: http://www.prolific.com.tw/US/ShowProduct.aspx?p_id=225&pc
- für alte und gefälschte PL2303: http://stefanfrings.de/avr_tools/index.html#pl2303
- für CH340 und CH341: http://www.wch.cn/download/CH341SER_ZIP.html
- für FTDI: http://www.ftdichip.com/Drivers/VCP.htm

Wenn der Treiber richtig installiert ist, sollte das Kabel im Gerätemanager als COM-Port erscheinen. Etwa so:

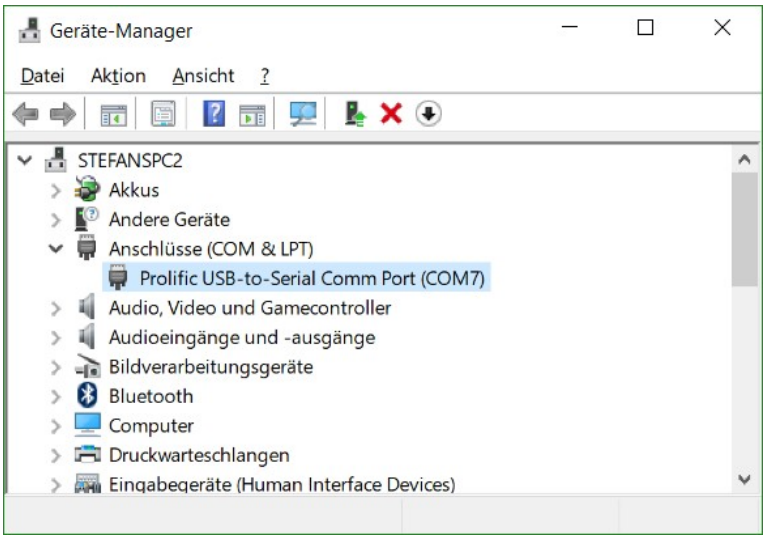


Abbildung 17: Virtueller COM-Port im Gerätemanager von Windows

Unter Linux gibt der Befehl „dmesg“ über den Ladevorgang des Treibers Auskunft.

Das USB-UART Kabel hat drei Leitungen, die für uns wichtig sind:

- TxD = Ausgang von gesendete Daten
- RxD = Eingang für empfangene Daten
- 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.

Die GND Leitung ist fast immer immer schwarz. Sie ist mit dem Rahmen des USB Steckers verbunden. Mit deinem Multimeter kannst du das überprüfen. Stelle das Messgerät dazu auf den 200 Ω Messbereich ein. Dann halte eine Mess-Spitze an den Rahmen des USB Steckers und die andere an die schwarze Leitung.

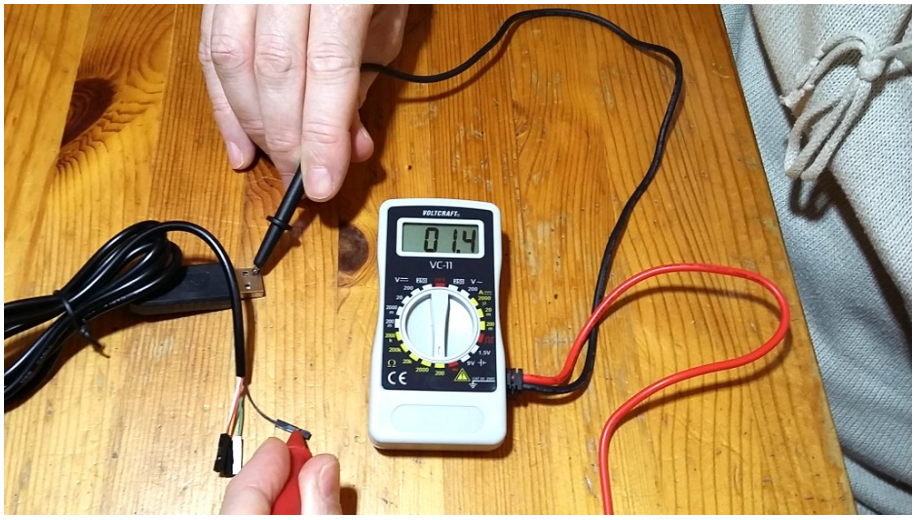


Abbildung 18: Finden der GND Leitung mit Multimeter

Der angezeigte Messwert von annähernd 0 Ohm (in diesem Fall 1,4 Ohm) bestätigt, dass ich die GND Leitung gefunden habe. Falls das bei dir nicht klappt, dann versuche es mit den anderen Farben.

Um das Kabel weiter zu untersuchen, brauchst du ein sogenanntes Terminal Programm. Ich empfehle das „Hammer Terminal“ von der Seite http://der-hammer.info/pages/terminal.html. Alternativ geht es auch gut mit Putty (Windows) und Cutecom (Linux).

In dem Programm kannst du Text eintippen, der dann vom Computer über das Kabel gesendet wird. Außerdem zeigt es empfangene Texte auf dem Bildschirm an. Wir werden das Terminal Programm und eine LED benutzen, um herauszufinden, auf welcher der vielen bunten Leitungen der Computer sendet. Die LED wird an der richtigen Leitung flackern.

Starte das Hammer Terminal Programm.

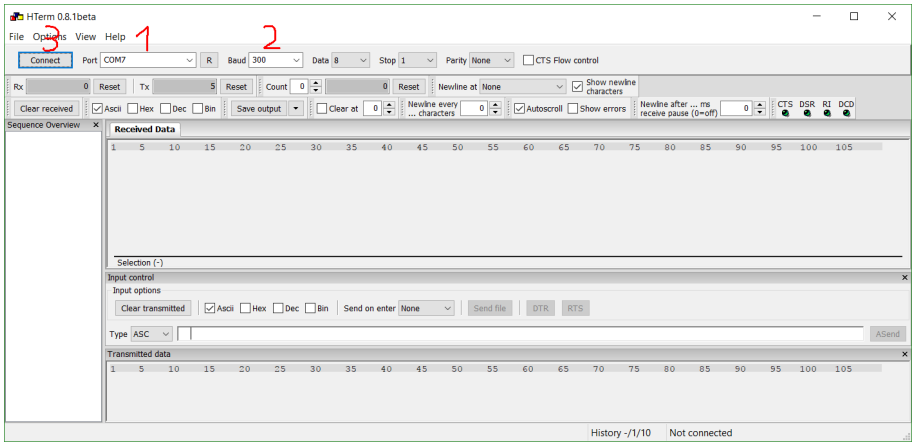


Abbildung 19: Verbindungsaufbau mit dem Programm Hammer-Terminal

Stelle oben links den COM Port deines Kabels ein und die kleinste Baud Rate, die dort möglich ist. Je kleiner die Baudrate ist, umso besser kann man die LED flackern sehen. Klicke dann links oben auf den „Connect“ Knopf.

Verbinde einen 2200 Ω Widerstand und eine rote Leuchtdiode wie folgt mit dem USB-UART Kabel:

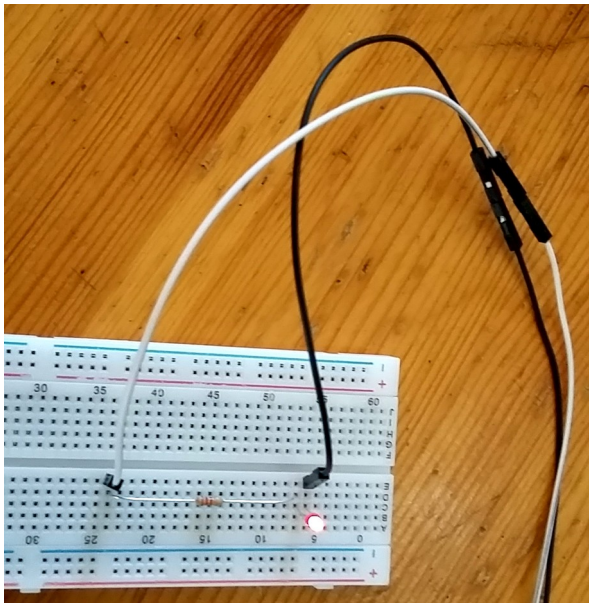


Abbildung 20: LED an USB-UART Adapter

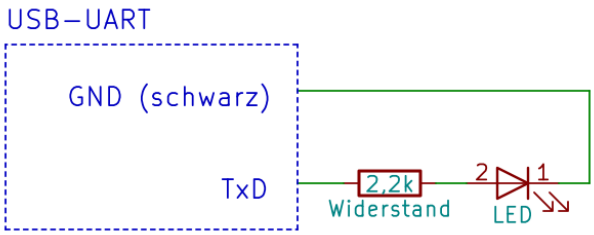


Abbildung 21: LED an USB-UART Adapter

Achte darauf, die LED richtig herum einzubauen. Der längere Draht gehört auf die Seite des Widerstandes. Da du noch nicht weißt, welche der Leitungen das TxD Signal führt, probiere einfach irgendeine Leitung aus. Fange zum Beispiel mit der weißen an. Die LED muss zunächst leuchten. Wenn sie nicht leuchtet, war das die falsche Leitung. Probiere dann eine andere Farbe.

Tippe dann einen langen Text ein und drücke die Enter Taste:

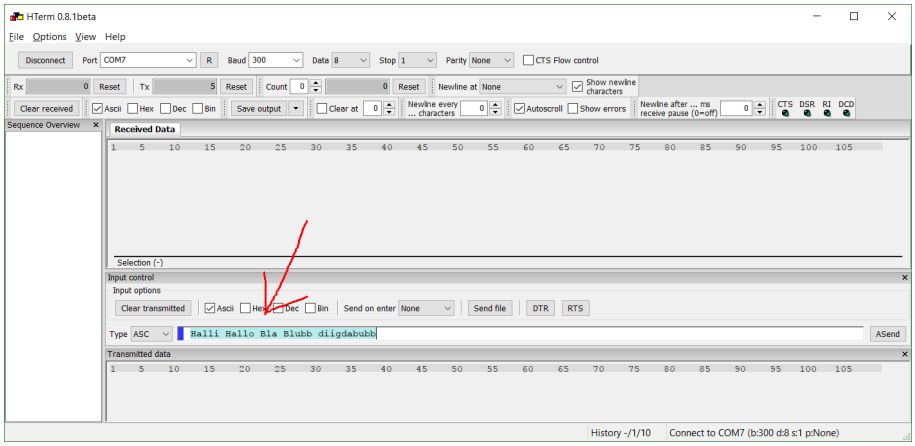


Abbildung 22: Text senden mit Hammer-Terminal

Wenn du die richtige Leitung erwischt hast, flackert die LED ein bisschen, sobald du die Enter Taste drückst. Falls die LED nicht flackert, versuche eine andere Leitung. Bei einer wird es klappen. In meinem Fall ist es die weiße. Auf dieser Leitung sendet der Computer also Texte heraus.

Tipp: Du kannst in dem Eingabefeld die Pfeil-nach-oben Taste drücken um den selben Text erneut zu senden.

Auf der TxD Leitung sendet der Computer Texte. Das Gegenstück dazu ist die RxD Leitung, auf dieser empfängt der Computer Texte. Wir werden den TxD Ausgang mit dem RxD Eingang

verbinden, so dass der Computer seine gesendeten Texte wie ein Echo wieder empfängt und anzeigt.

Benutze dabei sicherheitshalber einen weiteren 2200 Ohm Widerstand zum Schutz, damit nichts kaputt gehen kann, falls du die falschen Leitungen erwischst. Probiere aus, welche Farbe die RxD Leitung hat. Du könntest zum Beispiel mit der blauen Leitung beginnen:

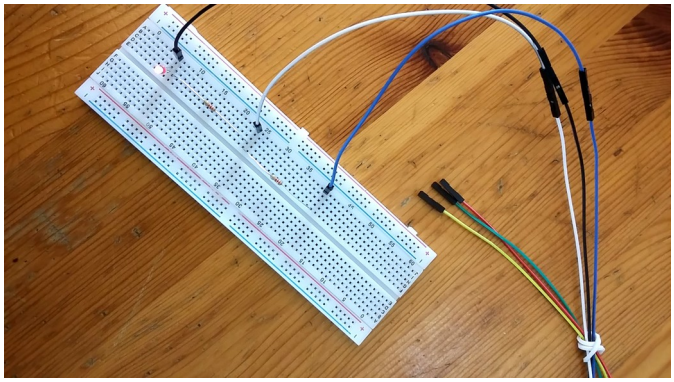


Abbildung 23: Echo-Schleife am USB-UART Adapter

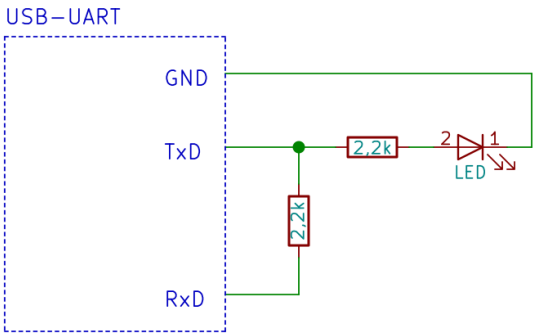


Abbildung 24: Echo-Schleife am USB-UART Adapter

Wenn du die richtige Leitung gefunden hast, dann zeigt das Terminal Programm den Text im oberen Empfangsbereich an, sobald du ihn nochmal sendest:

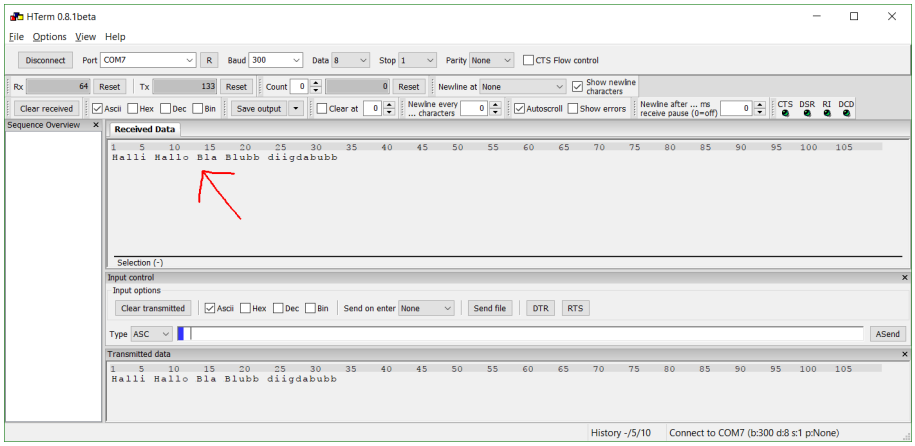


Abbildung 25: Empfangenes Echo vom USB-UART Adapter

Du hast nun die drei benötigten Leitungen gefunden: RxD, TxD und GND. Beschrifte dein Kabel mit einem Etikett, so dass die Zuordnung der Farben in Zukunft klar ist.

An die Enden von RxD und TxD sollst du nun jeweils einen 2200 Ohm Widerstand stecken oder löten. Er dient zum Schutz gegen Kurzschlüsse und Überspannung:

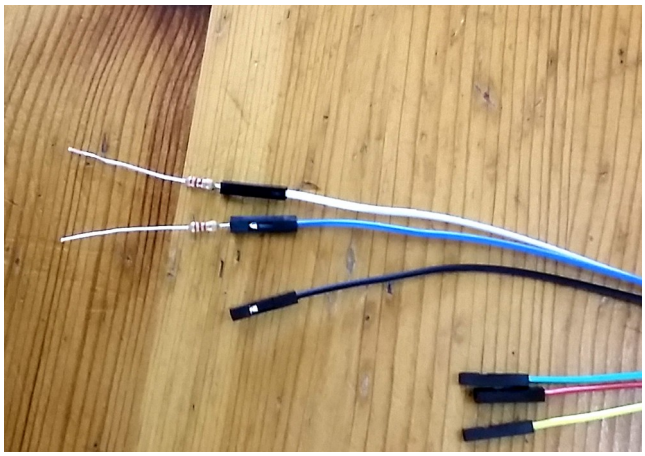


Abbildung 26: Schutzwiderstände an RxD und TxD

Ich habe das Ganze noch mit etwas Schrumpfschlauch überzogen, so ist es besonders stabil. Wer keinen Schrumpfschlauch besitzt, kann Isolierband nehmen. Zur Not geht auch Tesafilm.

Das wird ab jetzt also dein Kabel sein, welches den kleinen Mikrocontroller mit dem großen PC oder Laptop verbindet. Über dieses Kabel können die beiden Computer miteinander kommunizieren.

Tipp: Die Widerstände beschützen den Computer nur vor geringer Überspannung. Einen vollständigen Schutz bieten die sogenannten USB Isolatoren an. Bei AliExpress bekommt man sie besonders preisgünstig.

7 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.

Als kostenlose IDE kommt die „System Workbench for STM32“ in Frage, die du bitte von der Seite http://www.openstm32.org/Downloading+the+System+Workbench+for+STM32+installer herunter laden sollst. Du musst dich vor dem Download registrieren.

Alternativ kannst du auch die „STM32 Cube IDE“ von https://www.st.com/content/st_com/en/products/development-tools/software-development-tools/stm32-software-development-tools/stm32-ides/stm32cubeide.html verwenden. Sie ist der System Workbench ähnlich.

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

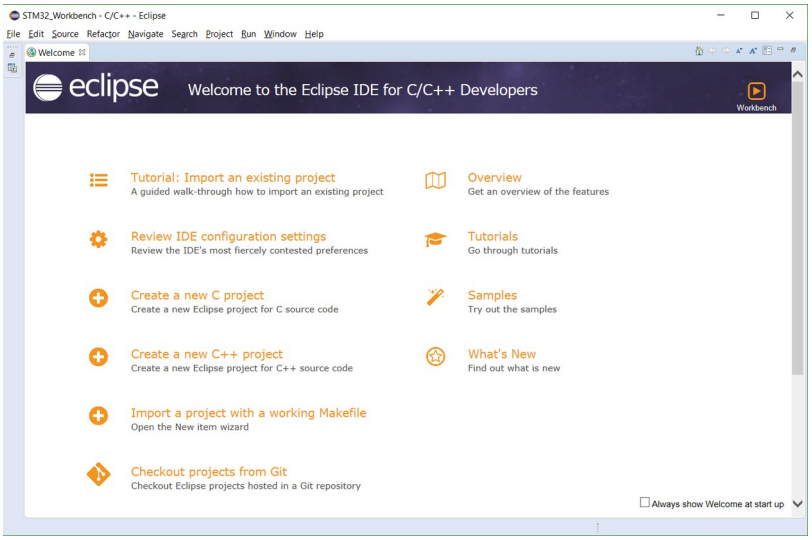


Abbildung 27: Startbild der System Workbench for STM32

Die Bildschirmfotos in diesem Buch stammen von der System Workbench Version 2.2. Gehe in das „Help“ Menü, dann auf den Punkt „Check for Updates“. Installiere alle Aktualisierungen, die dir dort angeboten werden.

Um das eigene Programm in den Mikrocontroller zu übertragen, brauchst du den „STM32 Flash loader demonstrator“, den du von der Seite http://www.st.com/en/development-tools/flasher-stm32.html herunterladen sollst.

Unter Linux kann man stattdessen das Programm „STM32 Flash“ von https://sourceforge.net/p/stm32flash/code/ci/master/tree/ im Terminalfenster benutzen. Ich erkläre hier im Buch aber nur das Windows Programm.

Du hast nun alle nötige Software installiert, jetzt kann es los gehen.

7.1 Beispiel-Projekt öffnen

Ich habe für dich ein Beispielprojekt vorbereitet, welches die untere LED auf dem Mikrocontroller Board blinken lässt. Lade es von http://stefanfrings.de/mikrocontroller_buch2/Blinker.zip herunter und packe es in deinem noch leeren Projekt-Ordner aus.

Starte die System Workbench. Damit das neue Projekt angezeigt wird, muss es über den Menüpunkt „File / Import / General / Existing Projects into Workspace “ importiert werden. In der STM32 Cube IDE heißt der entsprechende Befehl „File / Import / General / import ac6 System Workbench for STM32 Project“.

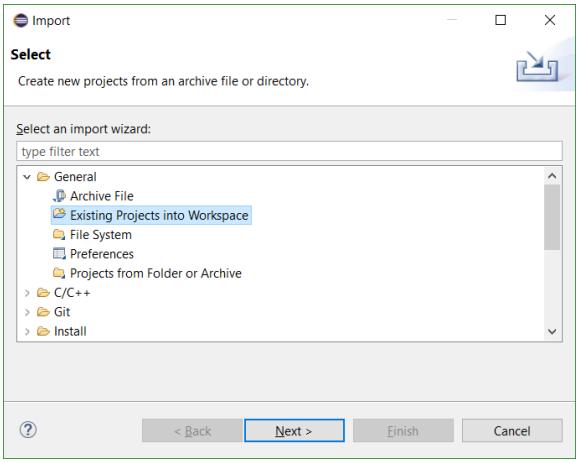


Abbildung 28: Projekt-Assistent der System Workbench for STM32

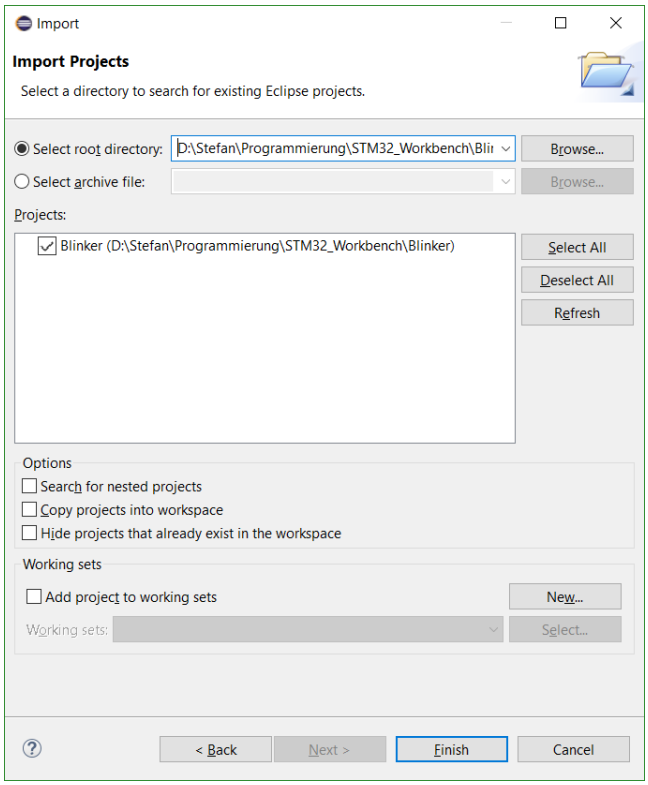


Abbildung 29: Projekt-Assistent der System Workbench for STM32

Im nächsten Dialog musst den Ordner von dem ausgepackten Beispielprojekt angeben und dann auf „Finish“ klicken.

Zurück im Hauptfenster der System Workbench kannst du jetzt am linken Rand das Beispielprojekt „Blinker“ finden. Öffne es und klicke dann doppelt auf die Datei “src/main.c“, damit sie im Text-Editor angezeigt wird. Es sieht dann so aus:

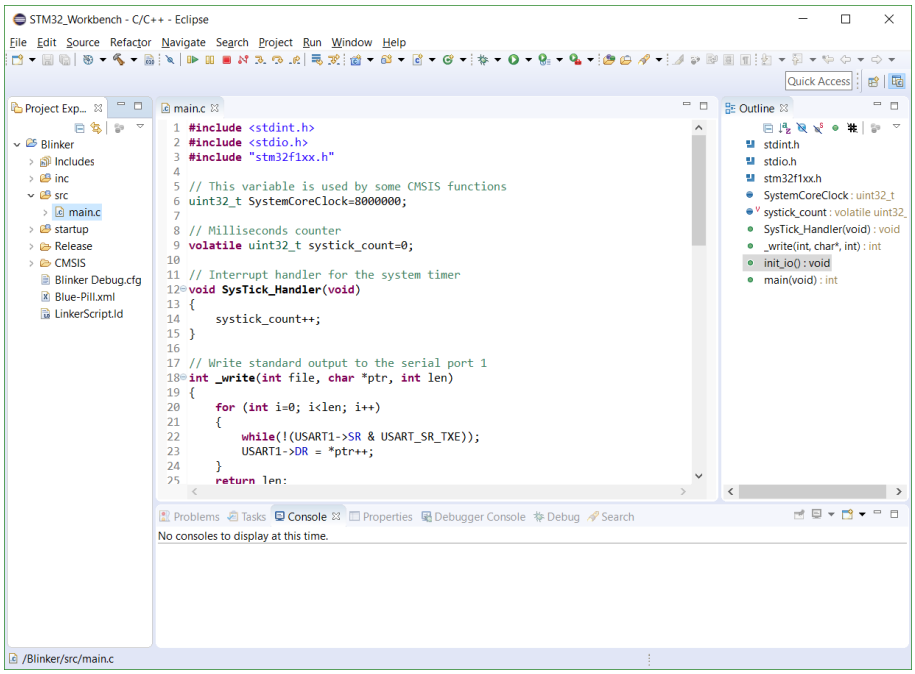


Abbildung 30: Ansicht der Datei main.c in der System Workbench

Das Programm ist nicht sehr groß, es sind nur 75 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 für‘s Erste darauf, dass dieses Programm funktioniert.

Klicke auf den kleinen schwarzen Pfeil neben dem Hammer und wähle dann „Release“ aus, falls das nicht schon der Fall ist.

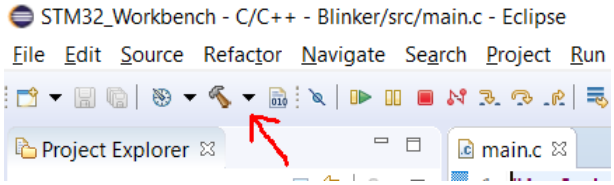


Abbildung 31: Schaltfläche für die Funktion "Build"

Dadurch bestimmst du, nach welcher Strategie der Compiler dein Programm optimieren soll. Die Release Variante ist kompakter und schneller, als die Debug Variante. Die Debug Variante brauchst du vielleicht später mal, wenn du einen Debugger benutzt.

Beim Umschalten startet die Entwicklungsumgebung den Compiler automatisch. Ansonsten klickst du auf den Hammer, um den Compiler manuell zu starten. Dieser übersetzt das Programm in Maschinencode. Die Meldungen des Compilers siehst du unten im „Console“ View:

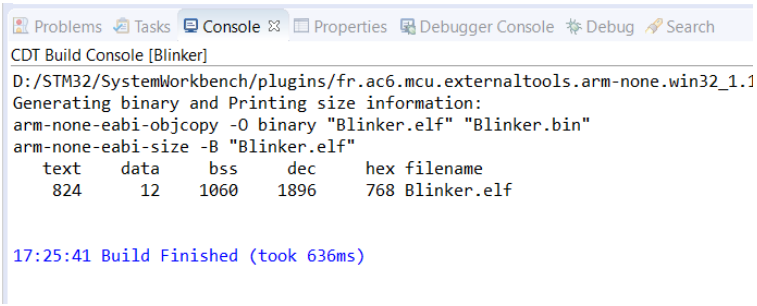


Abbildung 32: Meldungen des Compilers in der Console View

So sieht es aus, wenn keine Probleme auftraten. Falls bei dir die „Console“ View fehlt, kannst du sie über das Menü „Window / Show View / Console“ hinzufügen. Du kannst später jederzeit auf den Hammer klicken, um das Programm erneut zu compilieren.

7.2 Programm übertragen

Als nächstes wird das Programm in den Mikrocontroller übertragen. Er hat wie dein Smartphone einen integrierten Flash-Speicher, der seinen Inhalt auch ohne Strom behalten kann.

Dazu musst du das bereits vorbereitete USB-UART Kabel (mit den Schutzwiderständen) benutzen. Die schwarze GND Leitung gehört an irgendeinen Anschluss, der mit „G“ beschriftet ist. Davon gibt es mehrere.

- Die RxD Leitung muss mit dem Anschluss A9 verbunden werden.
- Die TxD Leitung muss mit dem Anschluss A10 verbunden werden.

Außerdem sollst du ein Smartphone Netzteil zur Stromversorgung anstecken. So sieht es aus:

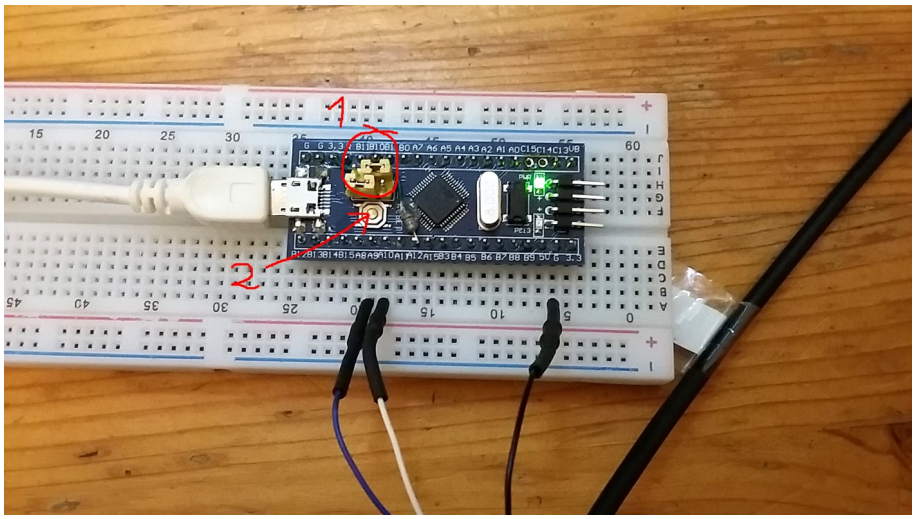


Abbildung 33: Starten des seriellen Bootloaders

Das Blue Pill Board hat oben eine grüne Power-LED, die immer leuchtet, und unten eine rote programmierbare LED, die momentan keine Rolle spielt. Beim schwarzen Board von RobotDyn ist die programmierbare LED blau.

Jetzt wird der Mikrocontroller in den „Firmware Update“ oder „Bootloader“ Modus versetzt, damit er bereit ist, das Programm über die gerade hergestellte Verbindung zu empfangen. Dazu stellst du die beiden gelben Steckbrücken wie im obigen Foto gezeigt ein. Die obere nach rechts, und die untere nach links. Dann musst du den Reset Knopf auf dem Board drücken.

Von jetzt an wartet der Mikrocontroller darauf, dass du das Programm „STM32 Flash loader demonstrator“ benutzt. Nach der Installation findet man es im Windows Startmenü unter dem Namen „Demonstrator GUI“.

Abbildung 34: STM32 Flash loader demonstrator, Verbindungsaufbau

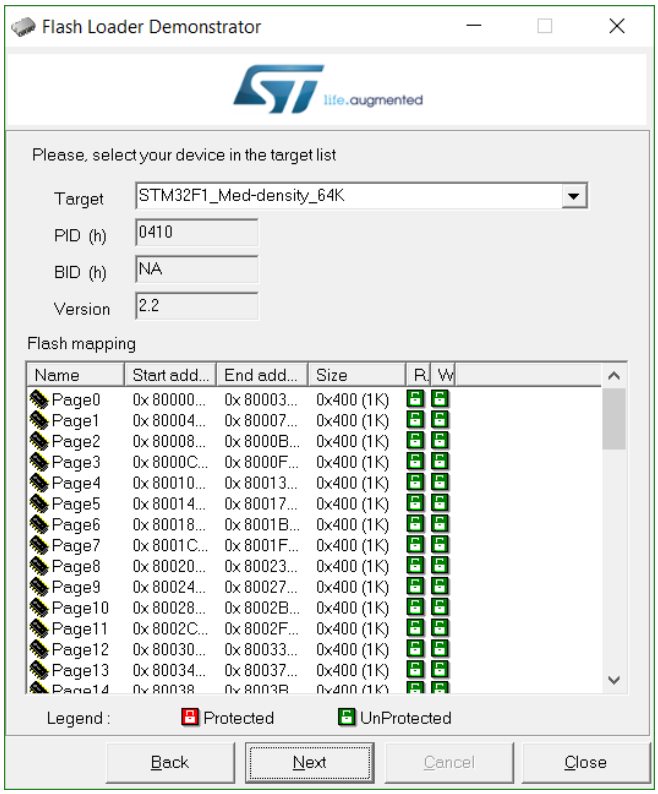


Abbildung 35: STM32 Flash loader demonstrator, nach dem Verbindungsaufbau

Stelle den COM Port deines USB-UART Kabels ein. Die anderen Parameter sollten meinem Bildschirmfoto entsprechen. Nach einem Klick auf „Next“ beginnt die Kommunikation zwischen Computer und Mikrocontroller. Eine grüne Ampel zeigt an, dass die Kommunikation funktioniert. Klicke nochmal auf „Next“.

Im nächsten Dialog werden die ersten Zeichen der Modellnummer des Mikrocontrollers angezeigt. STM32F1 ist hier richtig und der Chip hat 64k Byte Flash Speicher.

Klicke nochmal auf „Next“, um zum nächsten Formular zu gelangen.

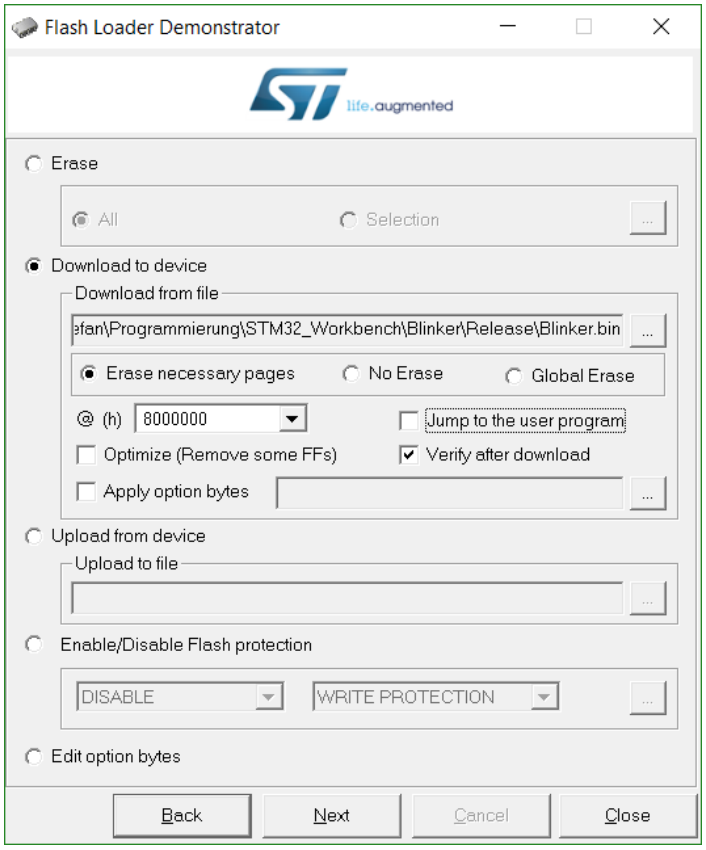


Abbildung 36: STM32 Flash loader demonstrator, Dateitransfer

Dieser Dialog ist in fünf Abschnitte unterteilt:

- **Erase**: Speicher löschen
- **Download to Device**: Ein Programm vom Computer in den Speicher des Mikrocontrollers übertragen
- **Upload from Device**: Den Speicher des Mikrocontrollers auslesen
- **Enable/Disable Flash protection**: Hiermit kann man den Speicher des Mikrocontrollers vor dem Auslesen oder verändern schützen. Mach das besser nicht, denn diese Einstellung lässt sich unter Umständen nicht mehr Rückgängig machen.
- **Edit option bytes**: Hier kann man ein paar Einstellungen vornehmen, die schon vor dem Programmstart wirksam sein müssen. Wir werden das nicht benutzen.

Wähle „Download to Device“ und gebe als Dateinamen die Datei Blinker\Release\Blinker.bin aus deinem Projektverzeichnis an. Darin befindet sich der Maschinencode. Darunter sollte die Option „Erase necessary pages“ eingeschaltet sein.

Das Häkchen „Verify after Download“ bedeutet, dass der Speicher nach der Übertragung nochmal überprüft wird. So werden eventuelle Übertragungsfehler und verschlissener Flash Speicher erkannt – sollte man immer benutzen.

Klicke nun auf „Next“. Ein Fortschrittsbalken begleitet die Datenübertragung. Danach wirst du eine grüne Erfolgsmeldung erhalten:

Abbildung 37: STM32 Flash loader demonstrator, Fortschrittsanzeige

Das Programm (genauer gesagt: der Maschinencode) ist nun in den Speicher des Mikrocontrollers übertragen worden. Um es zu starten, musst du die obere gelbe Steckbrücke nach links umstecken und dann den Reset Knopf drücken. Die untere LED blinkt jetzt.

Falls du die ganzen Arbeitsschritte bis hier hin nicht erfolgreich nachvollziehen konntest, suche im mikrocontroller.net Forum Hilfe oder kontaktiere mich per Email unter stefan@stefanfrings.de. Es macht keinen Sinn, mit den nächsten Kapiteln fortzufahren, solange das hier nicht klappt.

7.3 SWD Schnittstelle

Neben der UART Schnittstelle hat dieses Mikrocontroller Board noch einen zweiten Anschluss, mit dem man Maschinencode (=Firmware) übertragen kann. Und zwar ist das die vierpolige Stiftleiste am rechten Rand der Platine.

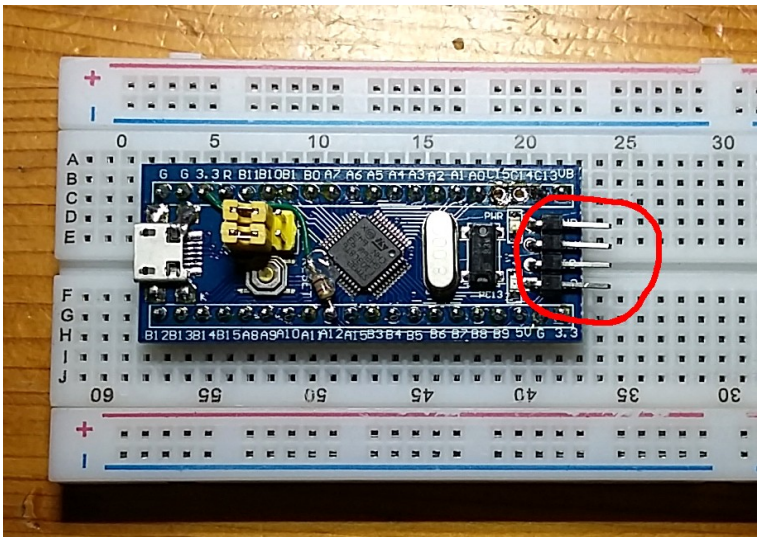


Abbildung 38: SWD Schnittstelle des Blue Pill Boardes

Daran kann man einen sogenannten ST-Link Adapter anschließen. Damit kann man den Mikrocontroller nicht nur programmieren, sondern auch Debuggen. Das bedeutet: man kann den Programmablauf pausieren und untersuchen, zum Beispiel den Inhalt aller Speicherzellen.

In diesem Buch wird die SWD Schnittstelle nicht benutzt. Weiter führende Informationen dazu findest du auf http://stefanfrings.de/stm32/stm32f1.html#swj.

8 UART Kommunikation

Du hast das USB-UART Kabel bereits benutzt, um den Maschinencode des Blinker Programms auf den Mikrocontroller zu übertragen.

USB-UART ist übrigens die Abkürzung von „Uniserval Serial Bus – Universal Asynchronous Receiver and Transmitter“.

Selbst geschriebene Programme können dieses Kabel ebenfalls benutzen. Zum Beispiel, um Texte an den Computer zu senden, die dann dort auf dem Bildschirm erscheinen. Probiere es aus:

Dein Mikrocontroller Board soll wie zuvor über das USB-UART Kabel mit dem Computer verbunden sein. Nach dem Anschließen der Stromversorgung beginnt die LED zu blinken.

Das Programm „STM32 Flash loader demonstrator“ musst du schließen, falls es noch offen ist. Starte jetzt das Hammer Terminal (oder Cutecom). Stelle die Baudrate auf 115200 um und klicke dann auf „Connect“.

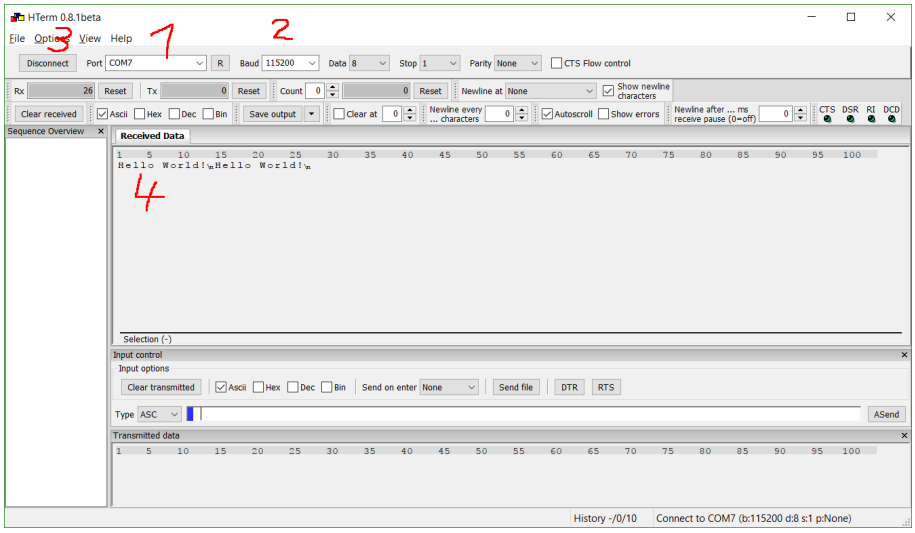


Abbildung 39: Einstellungen in Hammer-Terminal

Jedes mal, wenn die LED an geht, erscheint im Terminal Programm ein „Hello World!“. Dieser Text wird vom Blinker-Programm im Mikrocontroller über das USB-UART Kabel an deinen Computer gesendet.

Wenn du die „Newline at“ Option auf „LF“ stellst, werden Zeilenumbrüche eingefügt, dann sieht die Anzeige etwas schöner aus:

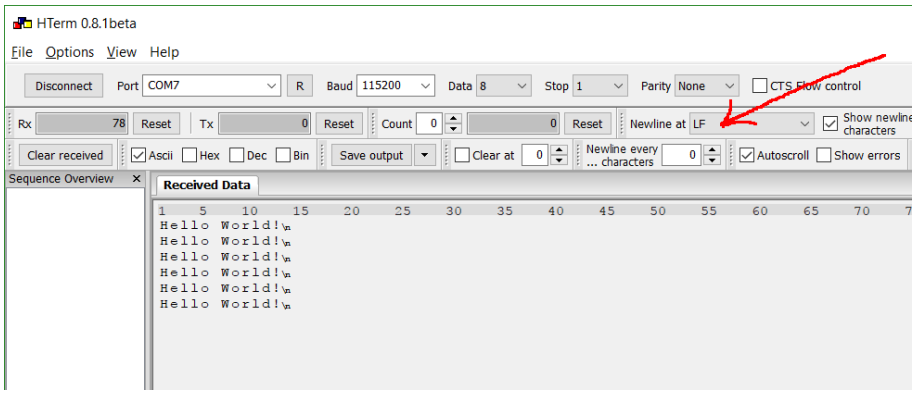


Abbildung 40: Zeilenumbruch im Hammer-Terminal ändern

Das der Mikrocontroller Texte an den Computer sendet, ist natürlich kein Zufall. Ich habe das Blinker-Programm dafür vorbereitet. Öffne es in der System Workbench und schau dir die Datei „main.c“ an. Dort wirst du ziemlich weit unten die markierten Zeilen finden:

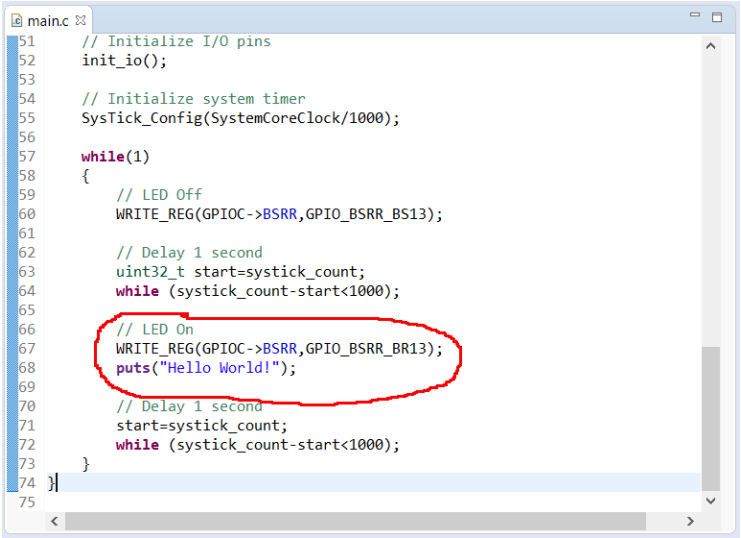


Abbildung 41: Änderung des auszugebenden Textes

Als erste Programmierübung kannst du ja mal versuchen, diesen „Hello World!“ Text zu ändern.

Schreibe zum Beispiel „Hallo Welt!“. Dann klickst du ganz oben links auf den Hammer, damit der Compiler diesen geänderten Quelltext in Maschinencode übersetzt. Das Ergebnis wird eine neue „Blinker.bin“ Datei sein.

Schließe das Terminal Programm und übertrage den neuen Maschinencode mit dem „STM32 Flash loader demonstrator“ in den Mikrocontroller.

Nachdem das geänderte Programm übertragen und gestartet ist, sollst du den „STM32 Flash loader demonstrator“ schließen und wieder das Terminal Programm starten. Verbinde dich mit 115200 Baud, und voilà – der geänderte Text erscheint:

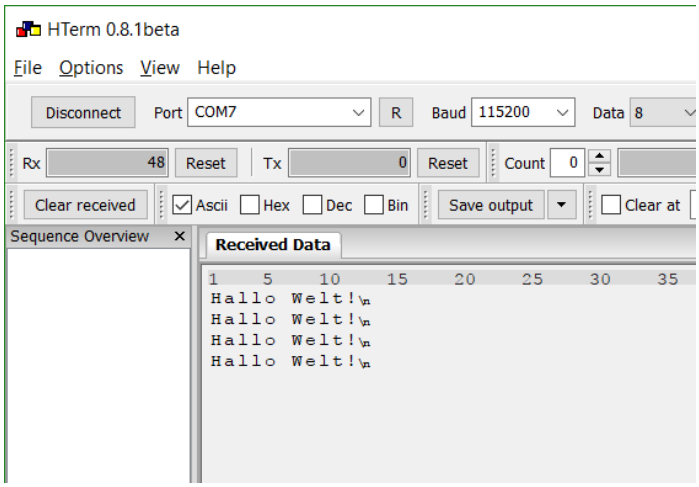


Abbildung 42: Anzeige des geänderten Textes im Hammer-Terminal

3

8.1 TxD und RxD

Der Anschluss PA9 hat die Funktion Txd (Transmit Data). Hier sendet der Mikrocontroller Daten an den PC. Der Anschluss PA10 hat die Funktion RxD (Receive Data), damit kann der Mikrocontroller Daten vom PC empfangen.

Der TxD Ausgang des Mikrocontroller muss mit dem RxD Eingang des Computer verbunden werden. Gleiches gilt umgekehrt für die entgegengesetzte Richtung, falls man sie benutzt.

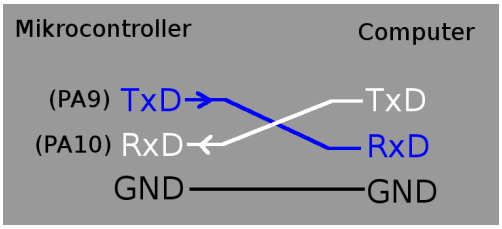


Abbildung 43: Serielle Verbindung "über Kreuz"

8.2 Baudrate

Die Baudrate legt fest, wie schnell die Daten übertragen werden. Eine niedrige Baudrate bewirkt eine langsame Übertragung. Probiere es aus. Suche dazu im Programm-Quelltext die rot markierte Stelle:

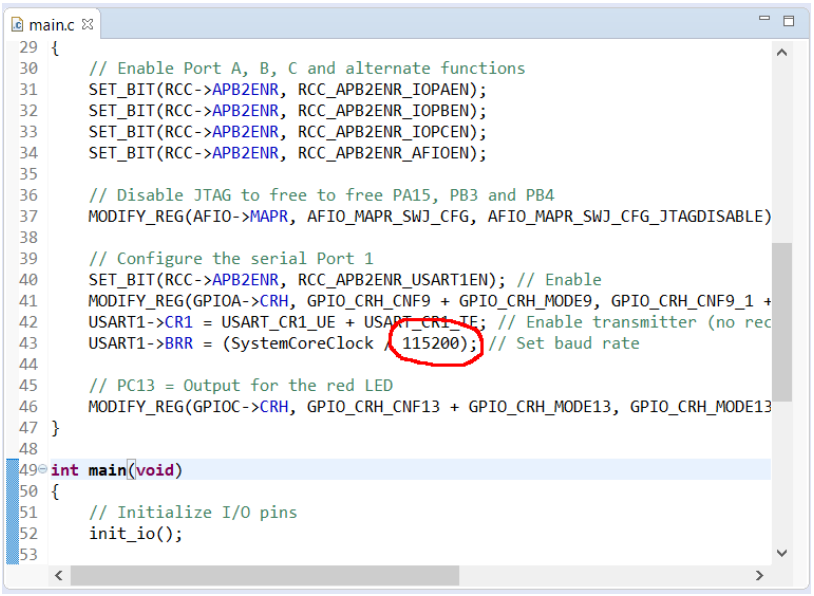


Abbildung 44: Änderung der Baudrate im Quelltext

Ändere diese Zahl auf 300, und klicke dann auf den Hammer um das Programm wieder zu Compilieren. Es wird eine neue Datei Blinker.bin erstellt.

Übertrage das Programm in den Mikrocontroller und starte es.

Benutze dann wieder das Terminal Programm, um die übertragenen Texte anzuzeigen. Dieses mal musst du auch dort die Baudrate auf 300 einstellen.

Wie zuvor erscheint jedes mal der Text „Hallo Welt!“, wenn die LED an geht. Aber du kannst deutlich sehen, dass die Übertragung viel langsamer stattfindet. Man kann sehen, wie die Buchstaben einer nach dem anderen erscheinen.

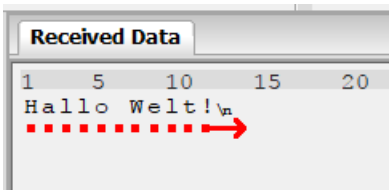


Abbildung 45: Sichtbar langsame Übertragung

Du hast im Hammer Terminal (oder Cutecom) gesehen, dass dein USB-UART Kabel viele unterschiedliche Baudraten unterstützt. Wenn man eine zu schnelle Baudrate verwendet, kommt es zu Übertragungsfehlern.

Bei 115200 Baud werden bis zu 11520 Zeichen pro Sekunde übertragen (immer durch 10 geteilt).

Weil die Texte über eine einzige Leitung übertragen werden, spricht man hier von einer seriellen Kommunikation. Diese UART Schnittstelle wird daher sehr häufig auch „serielle Schnittstelle“ genannt.

Wenn du mehr Details zur seriellen UART Schnittstelle erfahren möchtest, dann schau in Wikipedia nach: https://de.wikipedia.org/wiki/Universal_Asynchronous_Receiver_Transmitter

9 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.

Öffne in der System Workbench die Datei main.c von dem Blinker Projekt. Ändere die Baudrate zurück auf 115200, danach speicherst du die Datei ab indem du links oben auf das „Speichern“ Symbol klickst.

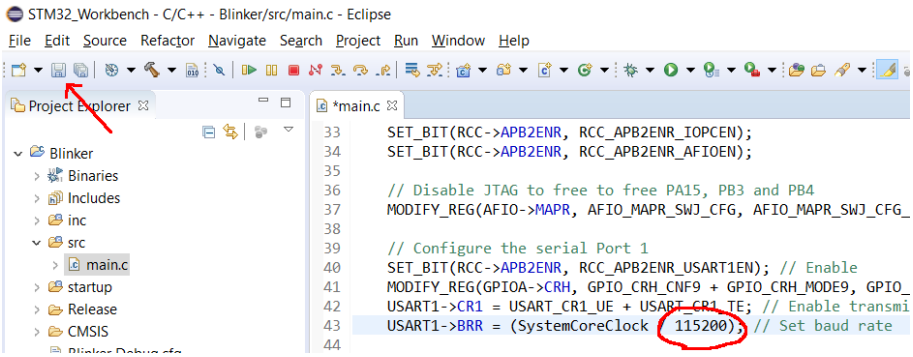


Abbildung 46: Änderung der Baudrate im Quelltext

Das 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.

9.1 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:

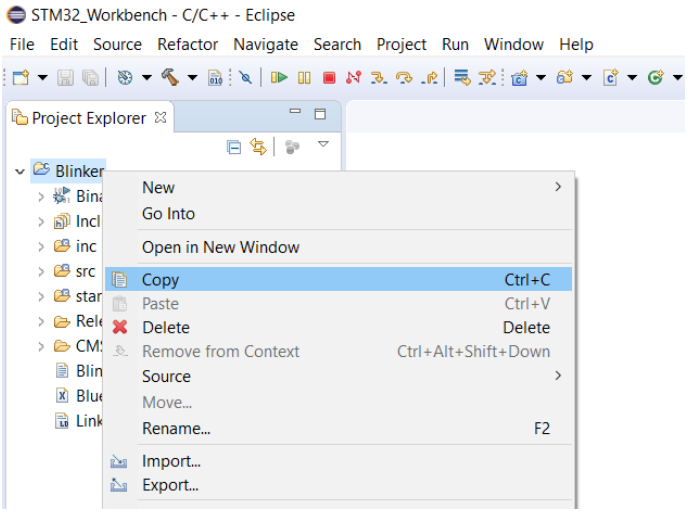


Abbildung 47: Copy-Befehl im Kontext-Menü des Projektes

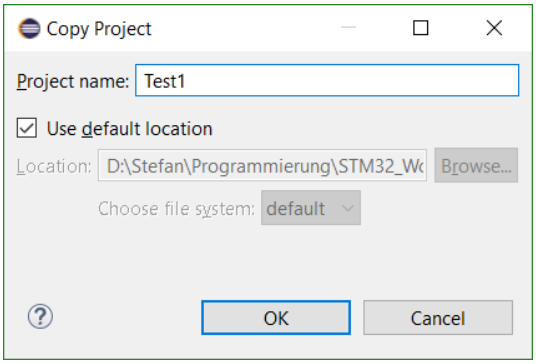


Abbildung 48: Eingabe des neuen Projektnamens

Nenne das neue Projekt „Test1“ und bestätige die Eingabe mit „Ok“.

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. Klicke dann mit der linken Maustaste auf den Projektnamen, so dass er markiert ist und drücke die Taste F5 damit die IDE zur Kenntnis nimmt, welche Dateien gelöscht wurden.

Eigentlich sollte das automatisch passieren, tut es aber nicht immer.

9.2 Projekt-Struktur

Schau dir die Struktur des Projektes an:

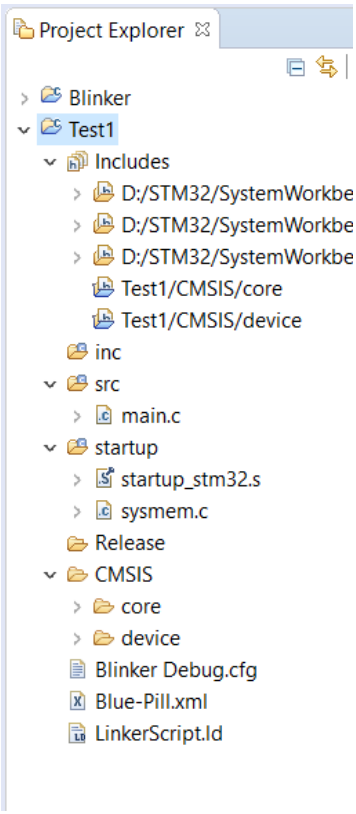


Abbildung 49: Projekt-Struktur

Ganz oben werden die „Includes“ aufgelistet. Diese Dateien enthalten die technische Beschreibung sämtlicher Funktionen der Programmiersprache C.

Darunter siehst du den leeren „inc“ Ordner. Im Rahmen dieses Buches wird dieser Ordner immer leer bleiben.

Im „src“ Ordner liegen deine Programm-Quelltexte.

Im „startup“ Ordner befinden sich zwei automatisch generierte Dateien. Die Datei startup_stm32.s enthält Quelltext in der Sprache Assembler. Der darin befindliche Code wird noch vor deinem Hauptprogramm (main) ausgeführt, um den Arbeitsspeicher so vorzubereiten, wie es die Programmiersprache C erfordert.

Der „Release“ Ordner ist momentan leer, weil du gerade den „Clean Project“ Befehl ausgeführt hast. Der Compiler legt dort temporäre Dateien und das Ergebnis der Kompilierung ab, also den Maschinencode. Wenn du auf den Hammer klickst, um den Compiler zu starten, wirst du dort etwas sehen.

Unter dem Release Ordner werden die Ordner „CMSIS/core“ und „CMSIS/device“ angezeigt.

Darin befindet sich die technische Beschreibung des Mikrocontroller Modells. Die braucht der Compiler, um den Maschinencode richtig zu erstellen. Aber auch die Rechtschreibkontrolle der IDE verwendet sie, um falsch geschriebene Namen zu erkennen.

Diese CMSIS Dateien eignen sich für alle Mikrocontroller, deren Nummer mit STM32F1 beginnt. Man muss allerdings beim Wechsel des Modells eine Anpassung in der Datei stm32f1xx.h vornehmen. Kommentare innerhalb der Datei helfen dabei.

Die drei Dateien ganz unten konfigurieren den Debugger und den Compiler. Sie wurden automatisch erzeugt und brauchen normalerweise nicht geändert zu werden.

9.3 Programm-Struktur

Der Mikrocontroller beginnt mit der Ausführung des Programms, wenn du die 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(void)
{
    // Initialize I/O pins
    init_io();

    // Initialize system timer
    SysTick_Config(SystemCoreClock/1000);

    ...
}
```

Die erste Zeile der main Funktion muss immer exakt so aussehen, sonst findet der Compiler sie nicht und kann keinen funktionsfähigen Maschinencode erstellen.

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 System Workbench 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 "stm32f1xx.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:

```
// This variable is used by some CMSIS functions
uint32_t SystemCoreClock=8000000;

// Milliseconds counter
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(void)
{
    // Richte die Ein-/Ausgabe Anschlüsse ein
    init_io();

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

    while(1)
    {
        // LED aus
        WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13);

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

        // LED ein
        WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13);
        puts("Hello World!");

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

Die Kommentare erklären oberflächlich, was die einzelnen Befehle tun. Ich habe sie hier mal auf deutsch übersetzt. 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.

9.4 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(void)
{
    init_io();
    WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13);
    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:

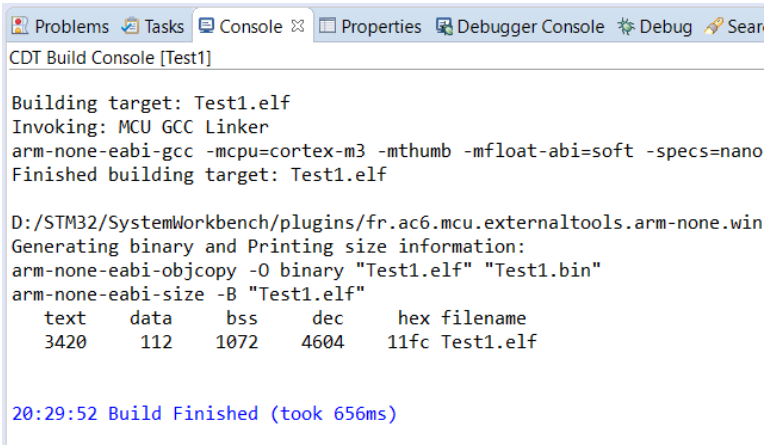


Abbildung 50: Meldungen des Compilers im Console View

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. Achte dabei darauf, die Datei Test1.bin zu übertragen (nicht die alte Blinker.bin). Dieses mal muss die programmierbare LED leuchten, nicht blinken.

Beende den „STM32 Flash loader demonstrator“ und starte das Hammer Terminal (oder Cutecom). Verbinde es mit 115200 Baud.

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. Konzentrieren wir uns auf die Kommentare. Ich übersetze sie auf deutsch:

- **Enable Port A, B, C and alternate functions**
  Aktiviere Port A, B, C und alternative Funktionen

  Wenn du dir die Beschriftung des Mikrocontroller Boards anschaust, siehst du ganz viele Nummern, die mit A und B anfangen. Oben rechts gibt es auch welche, die mit C anfangen. Diese Anschlüsse werden hier aktiviert, so dass dein Programm sie anschließend benutzen kann.

- **Disable JTAG to free to free PA15, PB3 and PB4**
  Deaktiviere JTAG um PA15, PB3 und PB4 zu befreien

  Die drei genannten Anschlüsse sind standardmäßig für einen sogenannten JTAG Debugger reserviert. Das ist eine Alternative zum weiter oben genannten SWD Debugger. Da du keinen JTAG Debugger benutzen wirst, deaktivieren wir diese Funktion. Dadurch werden die drei genannten Anschlüsse für andere Verwendung frei. Jetzt könntest du dort zum Beispiel Leuchtdioden anschließen.

- **Configure the serial Port 1**
  Konfiguriere den seriellen Port 1

  Hier wird der serielle Anschluss so eingestellt, dass er zum Senden von Text benutzt werden kann. Der Anschluss PA9 wird damit zum Ausgang TxD.

- **PC13 = Output for the red LED**
  PC13 = Ausgang für die rote LED

  Hier wird der Anschluss PC13 so eingestellt, dass er ein Ausgang ist. Da kommt nun also Strom heraus. PC13 ist auf dem Board mit der roten LED verbunden. Dein Programm bekommt dadurch die Möglichkeit, die rote LED ein und aus zu schalten.

  (Beim schwarzen Board von RobotDyn ist die LED blau)

  Alle anderen Anschlüsse sind standardmäßig als digitaler Eingang konfiguriert. Dort könnte man zum Beispiel Taster oder Sensoren anschließen. Das Programm kann dann ihren Zustand einlesen und verarbeiten.

```
    WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13);
```

Hier wird der Anschluss PC13 geschaltet, so dass die LED an geht. Ich erkläre diesen Befehl später noch detaillierter. Im Moment sollst du nur zur Kenntnis nehmen, dass hier die Zeichen „C“ und „13“ vorkommen, um anzugeben, dass eben dieser Anschluss geschaltet wird.

```
    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 300 und sende mehrere lange Texte. Danach soll die LED durch den folgenden Befehl wieder ausgeschaltet werden:

```
WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13);
```

Man beachte den winzigen Unterschied am Ende der Zeile:

BR13 versus BS13

Führe das Programm mehmals aus und kontrolliere dessen Ausgabe im Hyper Terminal. Kontrolliere, ob die LED nach dem Senden der Texte aus geht.

9.5 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 „STM32F1 Reference Manual“ beschreibt alle Register der kompletten Mikrocontroller Serie im Detail: https://www.st.com/resource/en/reference_manual/cd00171190-stm32f101xx-stm32f102xx-stm32f103xx-stm32f105xx-and-stm32f107xx-advanced-arm-based-32-bit-mcus-stmicroelectronics.pdf

(Das Referenzhandbuch musst du jetzt nicht komplett durch lesen)

Ein Register hast du bereits mehrfach benutzt, und zwar das Register „GPIOC->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:

Port bit set/reset register (GPIOx_BSRR) (x=A..G)

Address offset: 0x10

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 and can be accessed in Word mode only.
0: No action on the corresponding ODRx bit
1: Reset 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 and can be accessed in Word mode only.
0: No action on the corresponding ODRx bit
1: Set the corresponding ODRx bit

Abbildung 51: Auszug aus dem Referenzhandbuch: Beschreibung des BSSR 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 rote Leuchtdiode des Mikrocontroller Boards ist auf folgende Weise mit dem Mikrocontroller verbunden:

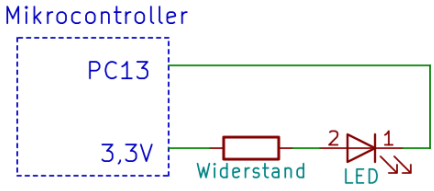


Abbildung 52: LED an Port C13

Damit sie von Strom durchflossen werden kann, muss der Ausgang PC13 auf 0V (Low) geschaltet werden. Wenn der Ausgang hingegen 3,3V (High) liefert, bleibt die Leuchtdiode dunkel, denn es fließt kein Strom.

Strom kann nämlich nur fließen, wenn am Anschluss 2 der Leuchtdiode eine wesentlich höhere Spannung anliegt, als am Anschluss 1. Und das ist eben dann der Fall, wenn der Ausgang des Mikrocontrollers auf 0 Volt steht.

- PC13 auf 0 Volt = Licht an
- PC13 auf 3,3 Volt = Licht aus.

Kommt dir das seltsam vor? Das ist es auch. Für solche Konstrukte wurde daher ein spezieller Name erfunden, nämlich „Negative Logik“.

Der Befehl zum Einschalten der LED lautet:

```
WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13);
```

Mit ein bisschen Englisch kannst du dir nun einen Reim darauf machen, was dieser Befehl bedeutet. Ich versuche das mal auf deutsch zu formulieren: „Beschreibe das Register BSSR vom Port C, und zwar mit dem Wert für das Bit BR13“.

Der Befehl zum Ausschalten der LED lautet:

```
WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13);
```

Der Unterschied ist BR versus BS.

- BR13 setzt den Ausgang auf 0 Volt (Low) = LED an
- BS13 setzt den Ausgang auf 3,3 Volt (High) = LED aus

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

Lass es uns ausprobieren, indem wir noch ein paar Leuchtdioden hinzufügen:

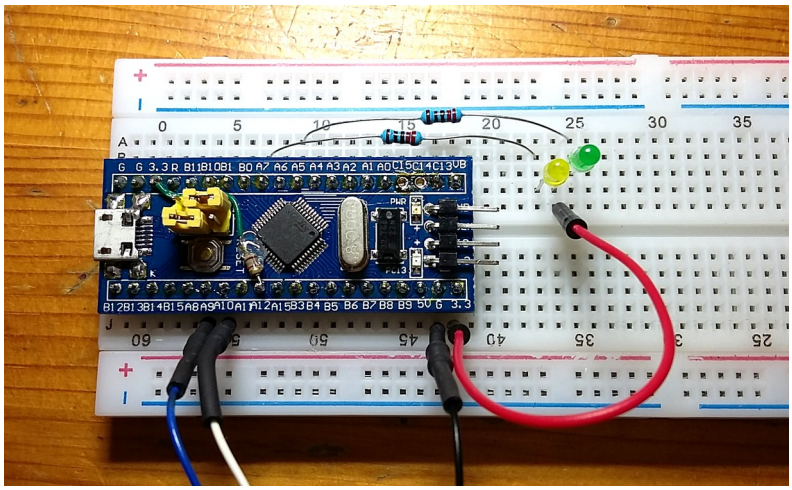


Abbildung 53: Aufbau von zwei LEDs and Port A5 und A7

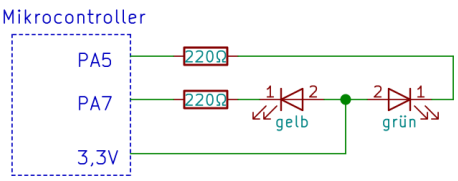


Abbildung 54: Zwei LEDs an Port A5 und A7

Immer die Stromversorgung aus stecken bevor du deinen Aufbau änderst! Dadurch vermeidest du Kurzschlüsse, die deine Bauteile zerstören würden.

Jetzt ändern wir das Programm „Test1“ so, dass zusätzlich zur LED auf dem Board auch die beiden hinzugefügten LEDs leuchten.

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

```
int main(void)
{
    init_io();
    WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13);
    WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR7);
    WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR5);
}
```

Die entscheidenden Stellen habe ich passend zu den Farben der LEDs markiert. Das war der einfache Teil.

Allerdings genügt das noch nicht, denn alle Anschlüsse (auch „Pins“ genannt) sind standardmäßig als Eingang konfiguriert. So kommt da noch kein Strom heraus. Du musst also die Anschlüsse PA5 und PA7 als Ausgang konfigurieren.

Füge dazu zwei Befehle zur Funktion „init_io“ hinzu:

```
void init_io(void)
{
    …
    // PC13 = Output for the red LED
    MODIFY_REG(GPIOC->CRH, GPIO_CRH_CNF13 + GPIO_CRH_MODE13,
       GPIO_CRH_MODE13_0);

    // PA5 and PA7 = Output
    MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF7 + GPIO_CRL_MODE7,
        GPIO_CRL_MODE7_0);
    MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF5 + GPIO_CRL_MODE5,
        GPIO_CRL_MODE5_0);
}
```

Zu beachten ist, dass hier für den Anschluss PC13 das Register CRH verwendet wurde, aber für PA5 und PA7 muss das Register CRL verwendet werden.

Compiliere das Programm und übertrage es dann auf den Mikrocontroller, um es auszuprobieren. Nach dem Start müssen alle drei Leuchtdioden leuchten.

Schauen wir uns einen der drei Befehle genauer an:

```
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF7 + GPIO_CRL_MODE7, GPIO_CRL_MODE7_0);
```

Auf deutsch übersetzt bedeutet das: „Modifiziere das Register CRL von Port A, und zwar nur die Felder CNF7 und MODE7. Beim MODE7 soll das Bit 0 gesetzt werden.“

Puh, das ist kompliziert. Um diesen Text auseinander zu dröseln, müssen wir uns die Beschreibung des CRL Registers im Referenzhandbuch anschauen:

Port configuration register low (GPIOx_CRL) (x=A..G)

Address offset: 0x00

Reset value: 0x4444 4444

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[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
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Bits 31:30, 27:26, 23:22, 19:18, 15:14, 11:10, 7:6, 3:2 **CNFy[1:0]:** Port x configuration bits (y= 0 .. 7)
These bits are written by software to configure the corresponding I/O port.
Refer to *Table 20: Port bit configuration table*.
In input mode (MODE[1:0]=00):
00: Analog mode
01: Floating input (reset state)
10: Input with pull-up / pull-down
11: Reserved
In output mode (MODE[1:0] > 00):
00: General purpose output push-pull
01: General purpose output Open-drain
10: Alternate function output Push-pull
11: Alternate function output Open-drain

Bits 29:28, 25:24, 21:20, 17:16, 13:12, 9:8, 5:4, 1:0 **MODEy[1:0]:** Port x mode bits (y= 0 .. 7)
These bits are written by software to configure the corresponding I/O port.
Refer to *Table 20: Port bit configuration table*.
00: Input mode (reset state)
01: Output mode, max speed 10 MHz.
10: Output mode, max speed 2 MHz.
11: Output mode, max speed 50 MHz.

Abbildung 55: Auszug aus dem Referenzhandbuch: Beschreibung des CRL Registers

Die beiden relevanten Felder habe ich hier Gelb markiert:

CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

Abbildung 56: Die Bits für Port A7 im Register GPIOA-CRL

Beide Felder umfassen jeweils zwei Bits. Das jeweils rechte Bit hat die Nummer 0, und das linke Bit hat die Nummer 1. Bits werden immer von rechts nach links durchnummeriert.

GPIO_CRL_MODE7_0 bedeutet, dass das rechte Bit 0 vom Feld MODE7 eingeschaltet werden soll, also das rot unterstrichene. Alle anderen Bits der beiden Felder werden ausgeschaltet.

Und was bewirkt das? Ich übersetze mal die Beschreibung:

00 = Eingang (Input)

01 = Ausgang, maximal 10MHz

10 = Ausgang, maximal 2 MHz

11 = Ausgang, maximal 50 MHz

Die Frequenz-Angabe sagt aus, wie schnell der Pin höchstens zwischen 0 Volt (Low) und 3,3 Volt (High) wechseln kann. Für die Experimente im Rahmen dieses Buches spielt das keine Rolle. Wichtig ist hier nur, dass der Pin als Ausgang konfiguriert wird.

Jetzt müssen wir uns noch das CNF Feld anschauen. Da gibt es laut Referenzhandbuch die folgenden Möglichkeiten:

00 = Allgemeiner Zweck Ausgang Drücken-Ziehen
Der Ausgang kann auf 3,3V drücken und auf 0V ziehen.

01 = Allgemeiner Zweck Ausgang Offener-Abfluss
Der Ausgang kann nur auf 0V (Low Pegel) ziehen. Wenn man versucht, ihn auf 3,3V (High Pegel) zu schalten, dann kommt nichts heraus.

10 = Alternative Funktion Ausgang Drücken-Ziehen
Der Ausgang wird für eine alternative Funktion verwendet. Er kann auf 3,3V drücken und auf 0V ziehen.

11 = Alternative Funktion Ausgang Offener-Abfluss
Der Ausgang wird für eine alternative Funktion verwendet. Dabei kann er nur auf 0V (Low Pegel) ziehen, jedoch keine 3,3V (High Pegel) drücken.

Im Rahmen dieses Buches nutzen wir nur eine einzige alternative Funktion, und zwar den Ausgang der seriellen Schnittstelle an PA9, der dadurch zu TxD wird. Alle anderen Ausgänge werden wir stets in der Konfiguration 00 als „Allgemeinen Ausgang“ verwenden.

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 „stm32f103xb.h“. Dort sind die Namen alle aufgelistet, schön nach Register Gruppiert.

Übungsaufgabe:

Schließe die grüne LED an PB9 an und lasse sie aufleuchten.

9.6 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_GELB_AN“ oder „LED_GELB_AUS“ schreiben könnte?

In der Tat ist das machbar. Und zwar, indem du Textersetzungen definierst. Auf Englisch nennt man das „Makro“.

Füge in der Datei „main.c“ vom „Test1“ Projekt ganz oben unter die „#include“ Zeilen dies ein:

```
#define LED_ROT_AN     WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13)
#define LED_GELB_AN    WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR7)
#define LED_GRUEN_AN   WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR5)

#define LED_ROT_AUS    WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13)
#define LED_GELB_AUS   WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS7)
#define LED_GRUEN_AUS  WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS5)
```

Danach kannst du die „main“ Funktion folgendermaßen vereinfachen:

```
int main(void)
{
    init_io();
    LED_ROT_AN;
    LED_GELB_AN;
    LED_GRUEN_AN;
}
```

Auf diese Weise wird der Quelltext deutlich besser lesbar.

Wenn du dir jetzt mal in Gedanken ein größeres Programm vorstellst, wo die gelbe LED an hundert Stellen geschaltet wird, kommt noch ein weiterer Vorteil zum Tragen: Die Wartbarkeit des Quelltextes verbessert sich. Stelle dir vor, die LED soll an einen anderen Anschluss verschoben werden. Dann brauchst du nur doch diese „#define“ Zeilen anzupassen, anstatt hundert Stellen im Quelltext.

9.7 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.

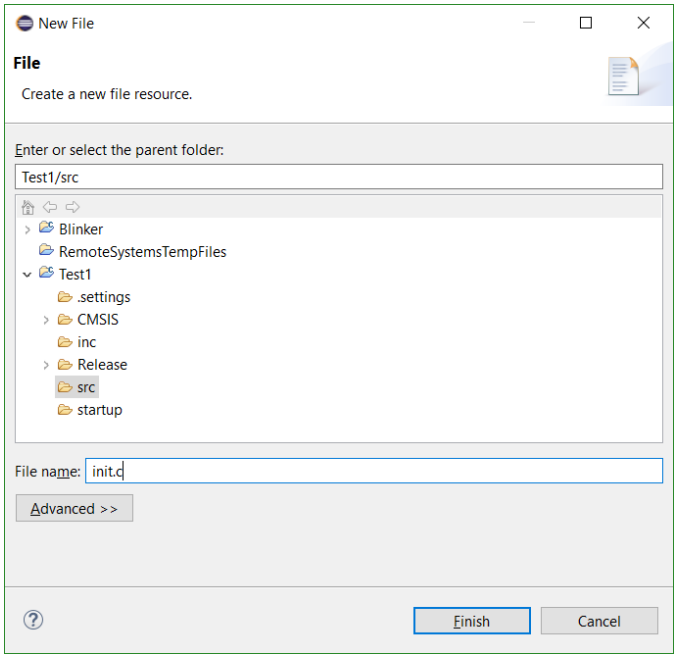


Abbildung 57: Assistent zum Anlegen einer neuen Datei

Klicke dazu in der System Workbench 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 "stm32f1xx.h"

// This variable is used by some CMSIS functions
uint32_t SystemCoreClock=8000000;

// Milliseconds counter
volatile uint32_t systick_count=0;

// Interrupt handler for the system timer
void SysTick_Handler(void)
{
    systick_count++;
}
```

```
// Write standard output to the serial port 1
int _write(int file, char *ptr, int len)
{
    for (int i=0; i<len; i++)
    {
        while(!(USART1->SR & USART_SR_TXE));
        USART1->DR = *ptr++;
    }
    return len;
}

void init_io(void)
{
    // Enable Port A, B, C and alternate functions
    SET_BIT(RCC->APB2ENR, RCC_APB2ENR_IOPAEN);
    SET_BIT(RCC->APB2ENR, RCC_APB2ENR_IOPBEN);
    SET_BIT(RCC->APB2ENR, RCC_APB2ENR_IOPCEN);
    SET_BIT(RCC->APB2ENR, RCC_APB2ENR_AFIOEN);

    // Disable JTAG to free to free PA15, PB3 and PB4
    MODIFY_REG(AFIO->MAPR, AFIO_MAPR_SWJ_CFG,
       AFIO_MAPR_SWJ_CFG_JTAGDISABLE);

    // Configure the serial Port 1
    SET_BIT(RCC->APB2ENR, RCC_APB2ENR_USART1EN);            // Enable
    MODIFY_REG(GPIOA->CRH, GPIO_CRH_CNF9 + GPIO_CRH_MODE9,
        GPIO_CRH_CNF9_1 + GPIO_CRH_MODE9_1);                // PA9=Output for alternate function
    USART1->CR1 = USART_CR1_UE + USART_CR1_TE;              // Enable transmitter (no receiver)
    USART1->BRR = (SystemCoreClock / 115200);               // Set baud rate

    // PC13 = Output for the red LED
    MODIFY_REG(GPIOC->CRH, GPIO_CRH_CNF13 + GPIO_CRH_MODE13,
        GPIO_CRH_MODE13_0);

    // PB9 and PA7 = Output
    MODIFY_REG(GPIOB->CRH, GPIO_CRH_CNF9 + GPIO_CRH_MODE9,
        GPIO_CRH_MODE9_0);
    MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF7 + GPIO_CRL_MODE7,
       GPIO_CRL_MODE7_0);
}
```

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

```
#include "stm32f1xx.h"

#define LED_ROT_AN     WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13)
#define LED_GELB_AN    WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR7)
#define LED_GRUEN_AN   WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR5)

#define LED_ROT_AUS    WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13)
#define LED_GELB_AUS   WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS7)
#define LED_GRUEN_AUS  WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS5)

int main(void)
{
    init_io();
    LED_ROT_AN;
    LED_GELB_AN;
    LED_GRUEN_AN;
}
```

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

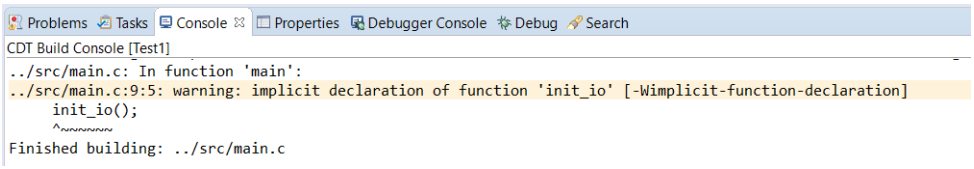


Abbildung 58: Warnmeldung des Compilers im Console View

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 "stm32f1xx.h"

#define LED_ROT_AN     WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13)
#define LED_GELB_AN    WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR7)
#define LED_GRUEN_AN   WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR5)

#define LED_ROT_AUS    WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13)
#define LED_GELB_AUS   WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS7)
#define LED_GRUEN_AUS  WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS5)

void init_io();

int main(void)
{
    init_io();
    LED_ROT_AN;
    LED_GELB_AN;
    LED_GRUEN_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.

Verschiebe die Kopfzeile in diese Datei:

```
#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 "stm32f1xx.h"
#include "init.h"

#define LED_ROT_AN     WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13)
#define LED_GELB_AN    WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR7)
#define LED_GRUEN_AN   WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR5)

#define LED_ROT_AUS    WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13)
#define LED_GELB_AUS   WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS7)
#define LED_GRUEN_AUS  WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS5)

int main(void)
{
    init_io();
    LED_ROT_AN;
    LED_GELB_AN;
    LED_GRUEN_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?

9.8 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 und danach das Projekt mit F5 zu aktualisieren.

Ändere die „main“ Funktion, dass sie so aussieht:

```
int main(void)
{
    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.

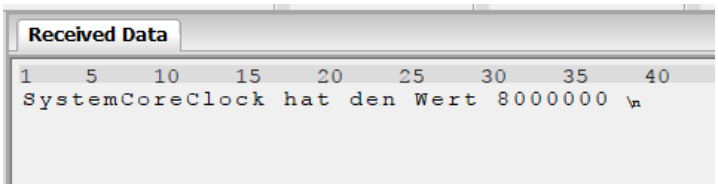


Abbildung 59: Serielle Ausgabe im Hammer-Terminal

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 ist nun der Moment, wo ein Blick in die Dokumentation angebracht wäre. Dazu musst du wissen, dass der C Compiler zusammen mit der „newlib“ Library daher kommt. 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 (integer)
%u	Ganze Zahl ohne Vorzeichen (unsigned integer)
%x	Ganze Zahl im Hexadezimal Format (unsigned integer)
%f	Fließkomma Zahl (float)
%%	Um das Prozent-Zeichen selbst auszugeben

Für lange Zahlen muss man ein „l“ vor den Buchstaben Schreiben. Der Compiler gibt so eine Warnung aus, wenn du das „l“ vergessen hast:

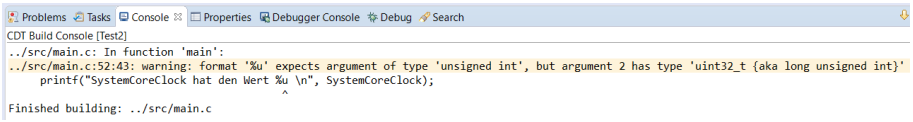


Abbildung 60: Warnmeldung des Compilers im Console View

```
format ‚%u‘ expects argument of type ‚unsigned int‘, but argument 2 has type
‚uint_32 {aka long unsigned int}‘
```

Du kannst festlegen, dass die Zahl auf eine bestimmte Länge rechtsbündig ausgegeben wird (wie die Geldbeträge auf einem Kassenbon):

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

Probiere es aus:

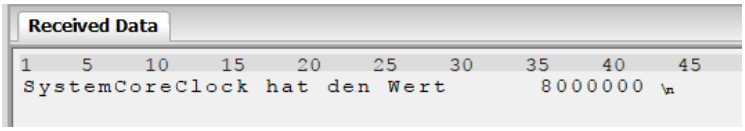


Abbildung 61: Serielle Ausgabe im Hammer-Terminal

An dieser Ausgabe kannst du sehen, dass für 12 Ziffern Platz reserviert wurde. Wenn du noch eine „0“ vor die 12 schreibst, wird die Lücke mit Nullen aufgefüllt:

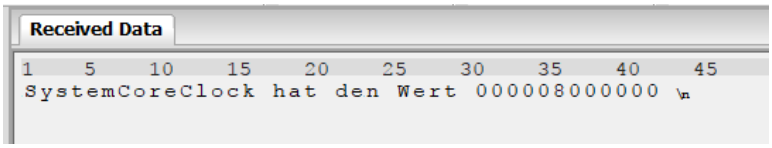


Abbildung 62: Serielle Ausgabe im Hammer-Terminal

Die printf Funktion kann auch mehrere Platzhalter ersetzen:

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

Probiere es aus, die Ausgabe ist:

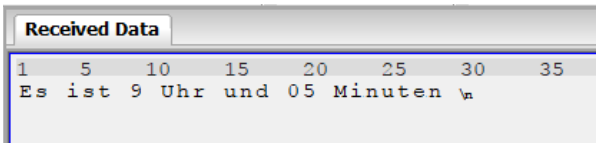


Abbildung 63: Serielle Ausgabe im Hammer-Terminal

Am Ende des Textes steht „\n“. Dieses Zeichen bedeutet „new line“ (neue Zeile), manchmal auch „line feed“ (Zeilenvorschub) genannt. Es ist eine Eigenart der newlib Library, dass jede Zeile mit „\n“ enden muss, sonst wird sie nicht ausgegeben.

Du kannst im Terminal Programm einstellen, wie es auf dieses Steuerzeichen reagieren soll:

Abbildung 64: Zeilenumbruch im Hammer-Terminal Einstellen

Wenn du das Terminal so einstellst, beginnt es beim Empfang von \n eine neue Zeile und verbirgt es in der Ausgabe. Bei anderen Terminal-Programmen ist dies üblicherweise die Standardvorgabe.

9.9 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: Klicke mit der rechten Maustaste auf den Projektnamen, dann auf „Properties / C/C++ Build / Settings / Tool Settings / MCU GCC Linker / Miscellaneous“. Im Textfeld „Linker flags“ musst du „-u _printf_float“ hinzufügen:

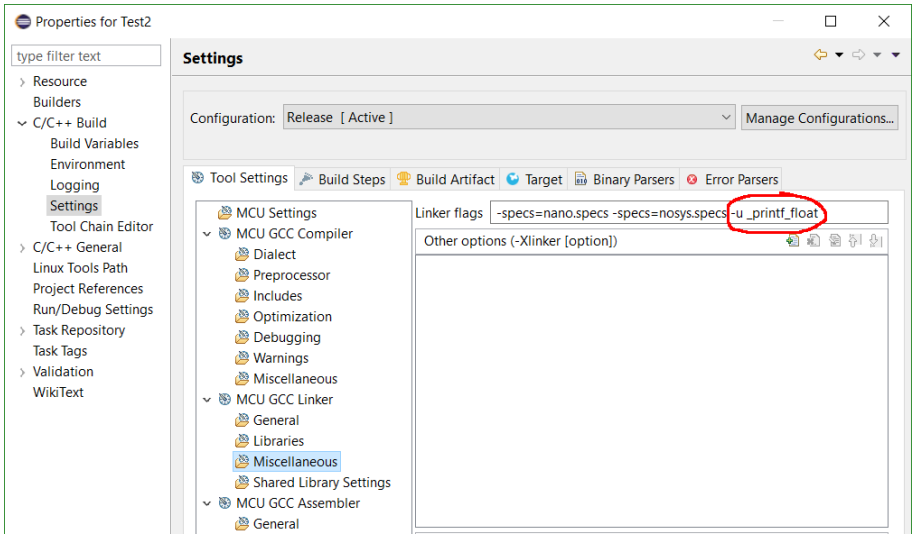


Abbildung 65: Linker Optionen in den Projekteinstellungen der System Workbench

Dadurch wird der Maschinencode allerdings erheblich größer.

Lass uns versuchen, ein paar Variablen zu benutzen. 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:

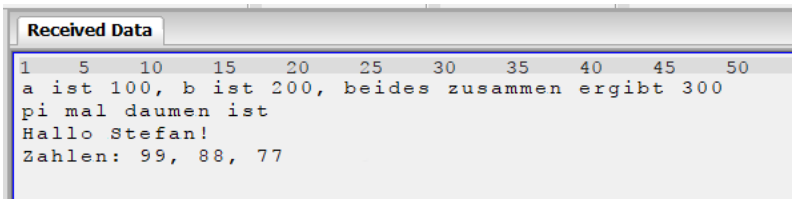


Abbildung 66: Serielle Ausgabe im Hammer-Terminal

Hinter „pi mal daumen ist“ fehlt die Zahl, weil ich die Unterstützung für Fließkommazahlen (siehe oben) noch nicht eingestellt habe.

Nach der Korrektur sieht das Ergebnis so aus:

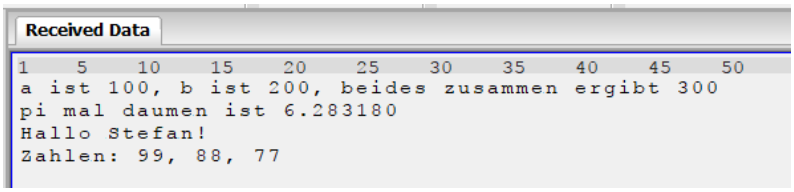


Abbildung 67: Serielle Ausgabe im Hammer-Terminal

So ist es besser. Allerdings ist das Programm dadurch dreimal so groß geworden!

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 die ausgerechnet, wie viel a+b ergibt und 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* 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.

*) 30 Zeichen ist 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.

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.

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.

9.10 Rechnen

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

+ für die Addition und * für die Multiplikation. Für die Subtraktion benutzt man logischerweise das Minus Zeichen. 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.

9.11 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.

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

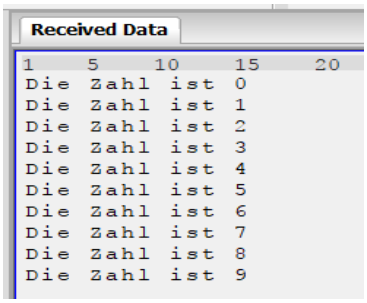


Abbildung 68: Serielle Ausgabe im Hammer-Terminal

Erklärung:

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

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.

9.11.2 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);
    }
```

9.11.3 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(void)
{
    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");
    }
}
```

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.

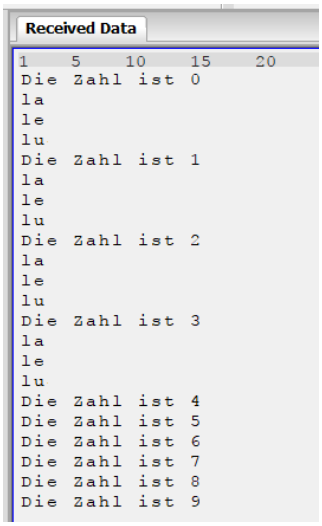


Abbildung 69: Serielle Ausgabe im Hammer-Terminal

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.

9.12 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");
}
```

9.13 System-Timer

Alle ARM Mikrocontroller enthalten einen sogenannten „System-Timer“ (oder „SysTick Timer“) - auch deiner. 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(void)
{
    SysTick_Config(SystemCoreClock/1000);
}
```

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

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

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“.

Merke: 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 115200 auf 300, um eine langsame Ausgabe zu erzwingen. Kopiere dann das folgende Hauptprogramm:

```
int main(void)
{
    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.

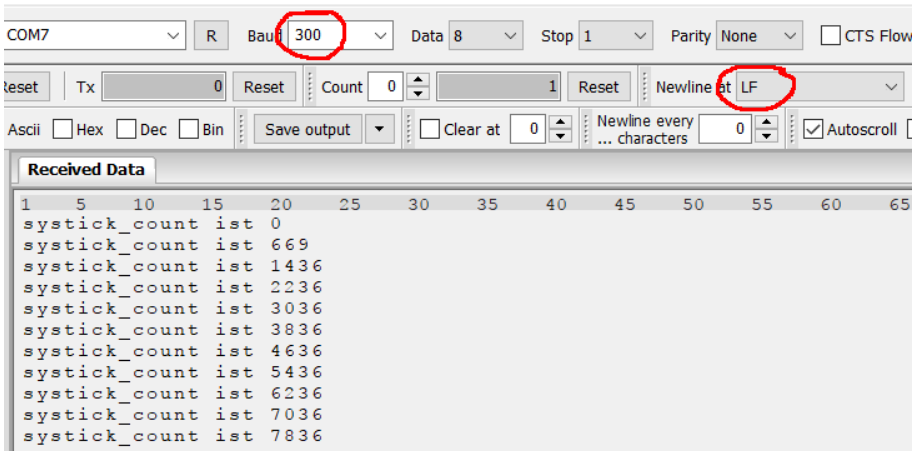


Abbildung 70: Serielle Ausgabe im Hammer-Terminal nach Einstellung von Baudrate und Zeilenumbruch

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 Off
        WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13);

        // Delay 1 second
        uint32_t start=systick_count;
        while (systick_count-start<1000);

        // LED On
        WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13);

        // Delay 1 second
        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.

9.14 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:

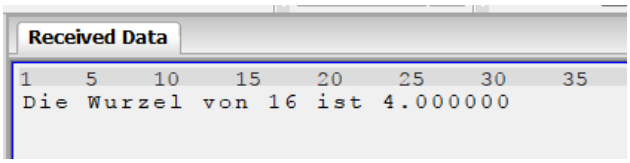


Abbildung 71: Serielle Ausgabe im Hammer-Terminal

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.

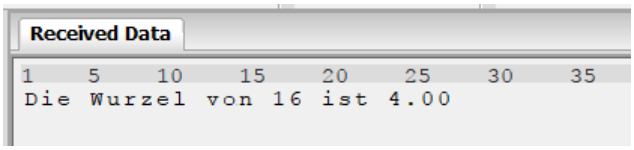


Abbildung 72: Serielle Ausgabe im Hammer-Terminal

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

9.15 Digitale Eingänge

In diesem Kapitel lernst du, wie man digitale Eingänge benutzt. Taster schließt man üblicherweise auf die folgende Weise an Mikrocontroller an:

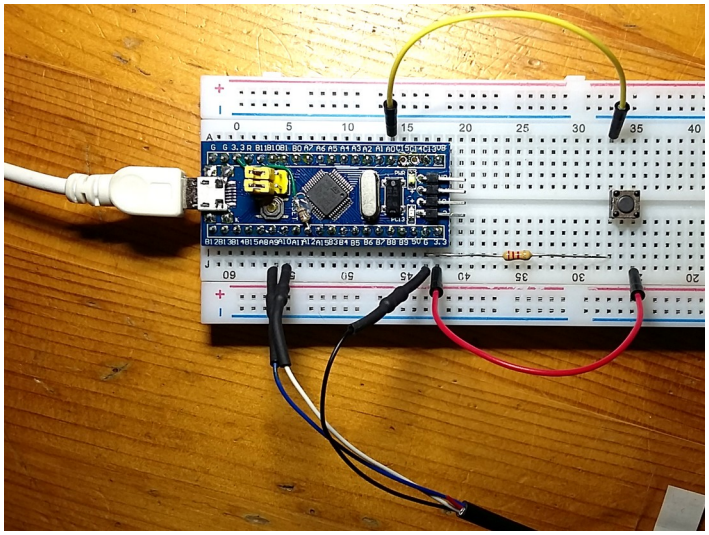


Abbildung 73: Aufbau vom taster mit Pull-Down Widerstand an Port A0

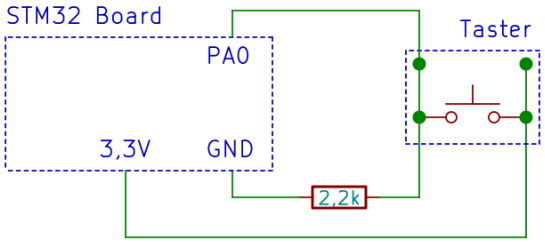


Abbildung 74: Taster mit Pull-Down Widerstand an Port A0

Die Anschlussbeinchen der Taster musst du mit einer Zange glätten, damit sie in das Steckbrett passen. Bei diesen Tastern sind die beiden linken Anschlüsse intern miteinander verbunden. Ebenso sind auch die beiden rechten Anschlüsse miteinander verbunden. Deswegen habe ich das so gezeichnet.

Solange der Taster nicht gedrückt wird, liegt der Eingang PA0 durch den Widerstand auf Low Pegel (GND = 0 V). Wenn du den Taster drückst, zieht er das Signal auf High Pegel (3,3 V).

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 (ist mit GND verbunden) und die rote Mess-Spitze an den Anschluss A0 (wo das gelbe Kabel steckt).

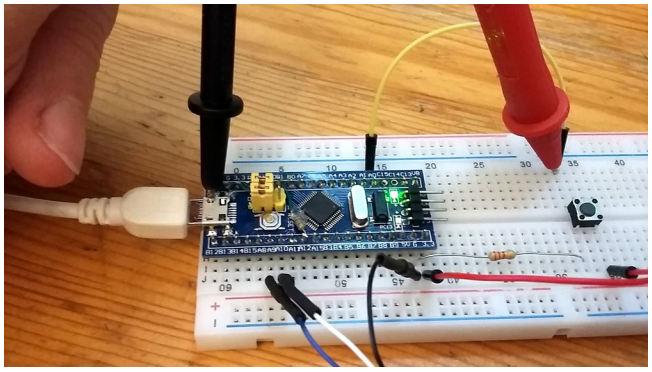
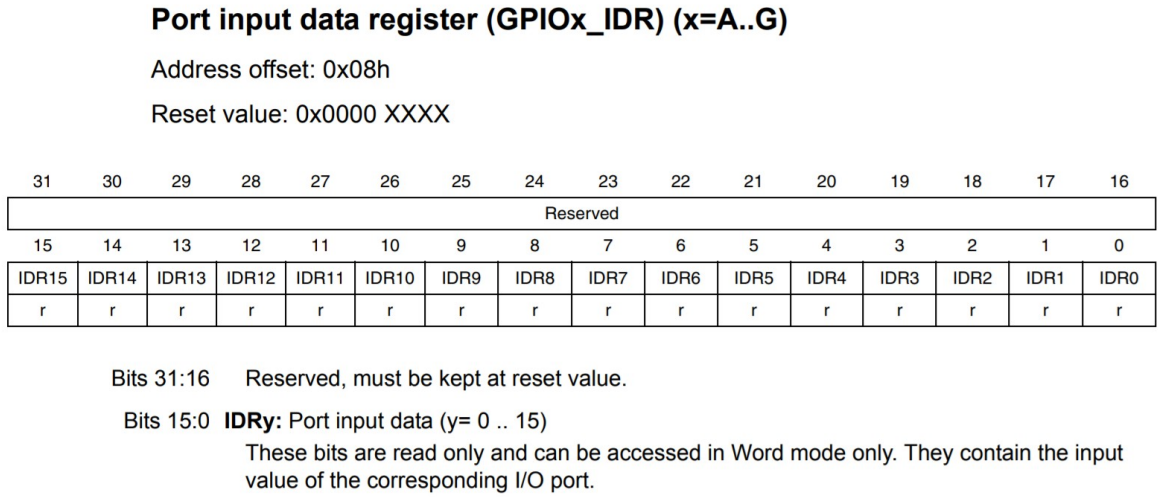


Abbildung 75: Messung der Spannung am Ausgang des Tasters

Das Messgerät zeigt 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.

Im Referenzhandbuch wird es so beschrieben:



Port input data register (GPIOx_IDR) (x=A..G)

Address offset: 0x08h

Reset value: 0x0000 XXXX

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
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 (y= 0 .. 15)
These bits are read only and can be accessed in Word mode only. They contain the input value of the corresponding I/O port.

Abbildung 76: Auszug aus dem Referenzhandbuch: Beschreibung des IDR Registers

Wie du sehen kannst, kann man dieses Register per Programm nur auslesen, aber nicht verändern. Die 16 Bits spiegeln stets den aktuellen Zustand der 16 Eingänge eines Ports dar.

Erstelle eine neue Kopie vom „Blinker“ Projekt und nenne es „Test3“. Vergiss nicht, das Projekt mit „Clean“ zu bereinigen und mit der Taste F5 zu aktualisieren.

Kopiere die folgende „main“ Funktion:

```
int main(void)
{
    // Initialisiere die I/O Pins
    init_io();

    while(1)
    {
        // Wenn der Taster gedrückt ist
        if (READ_BIT(GPIOA->IDR, GPIO_IDR_IDR0))
        {
            // LED Ein
            WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13);
        }
        else
        {
            // LED Aus
            WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13);
        }
    }
}
```

Compiliere das Programm und übertrage die Datei „Test3.bin“ auf den Mikrocontroller, um sie auszuführen. Wenn du den Taster drückst, geht die 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_IDR0)
```

Das bedeutet soviel wie „Lese vom Port A das Bit 0 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.

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. Und 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(void)
{
    // Initialisiere die I/O Pins
    init_io();

    while(1)
    {
        // Wenn der Taster nicht gedrückt ist
        if (!READ_BIT(GPIOA->IDR, GPIO_IDR_IDR0))
        {
            // LED Ein
            WRITE_REG(GPIOC→BSRR,GPIO_BSRR_BR13);
        }
        else
        {
            // LED Aus
            WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13);
        }
    }
}
```

Eine andere Möglichkeit, die Aufgabe zu erfüllen wäre:

```
if (READ_BIT(GPIOA->IDR, GPIO_IDR_IDR0) == 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. Das habe ich oben ja indirekt geschrieben:

„Jedenfalls liefert READ_BIT eine Zahl die größer als 0 ist, wenn das Bit einen High Pegel hat.“

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

```
int main(void)
{
    // Initialisiere die I/O Pins
    init_io();

    while(1)
    {
        // Wenn der Taster gedrückt ist
        if (READ_BIT(GPIOA->IDR, GPIO_IDR_IDR0))
        {
            // LED Aus
            WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13);
        }
        else
        {
            // LED Ein
            WRITE_REG(GPIOC→BSRR,GPIO_BSRR_BR13);
        }
    }
}
```

Übungsaufgabe: Probiere alle drei Varianten aus.

Wir wollen nun zwei Taster anschließen. Der linke Taster soll die LED ein schalten und der rechte Taster soll sie aus schalten.

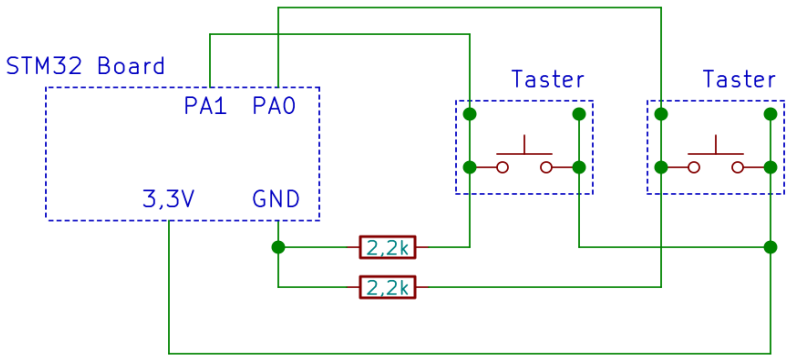


Abbildung 77: Zwei Taster mit Pull-Down Widerstand an Port A0 und A1

Foto vom Aufbau:

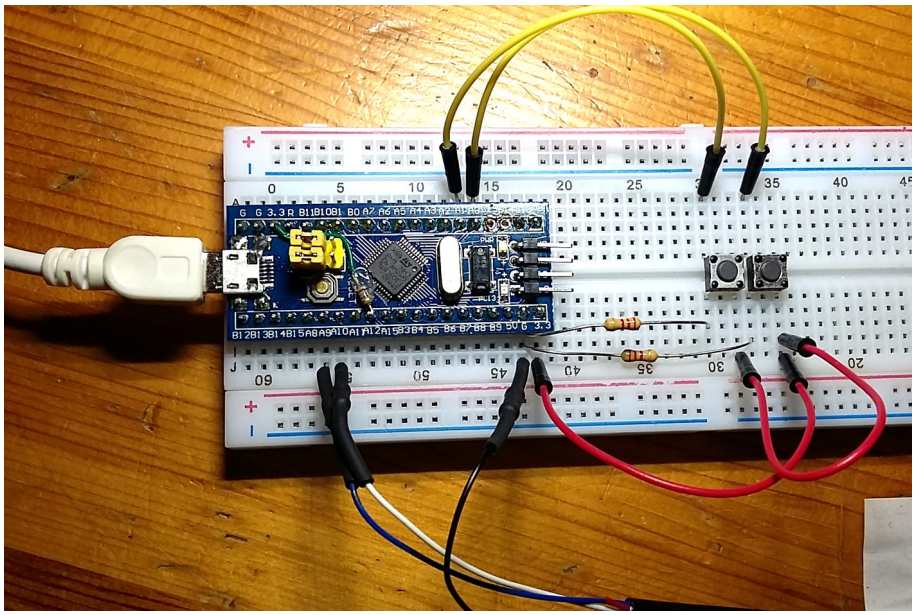


Abbildung 78: Aufbau von zwei Tastern mit Pull-Down Widerstand an Port A0 und A1

Das dazu passende Hauptprogramm sieht so aus:

```
int main(void)
{
    // Initialisiere die I/O Pins
    init_io();

    while(1)
    {

        // Wenn der Taster an PA1 gedrückt ist
        if (READ_BIT(GPIOA->IDR, GPIO_IDR_IDR1))
        {
            // LED Ein
            WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13);
        }

        // Wenn der Taster an PA0 gedrückt ist
        if (READ_BIT(GPIOA->IDR, GPIO_IDR_IDR0))
        {
            // LED Aus
            WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13);
        }
    }
}
```

Übertrage das Programm in den Mikrocontroller um es auszuprobieren.

Du wirst sehen, dass das Programm im Prinzip funktioniert, es hat aber einen kleinen Schönheitsfehler: Die LED ist beim Start schon eingeschaltet, bevor ein Taster gedrückt wird.

Der Grund dafür ist, dass die LED Leuchtet, wenn der Ausgang auf Low (0 Volt) geschaltet ist. Und das ist die Standardvorgabe für alle Ausgänge.

Um den Fehler zu beheben, füge einen weiteren Befehl ein, der die LED direkt beim Programmstart aus schaltet:

```
int main(void)
{
    // Initialisiere die I/O Pins
    init_io();

    // LED Aus
    WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13);

    while(1)
    {
        ...
    }
}
```

Probiere das Programm wieder aus. Nun funktioniert es korrekt.

Wenn es dir besser gefällt, kannst du diesen Befehl alternativ ans Ende der Funktion „init_io“ platzieren, schließlich ist das auch ein Teil der Initialisierung.

Die beiden Widerstände könnte man theoretisch auch weg lassen, und stattdessen die sogenannten „Pull-Down“ Widerstände im Innern des Mikrocontrollers aktivieren. Allerdings sind die internen Widerstände sehr hochohmig (ca. 40 kΩ), was die Schaltung störungsempfindlich machen würde. Deswegen benutzen wir lieber externe Widerstände.

9.16 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.

Über die Register CRL (für Pin 0-7) oder CRH (für Pin 8-15) konfiguriert man den Anschluss als Ausgang:

```
MODIFY_REG(GPIOC->CRH, GPIO_CRH_CNF13 + GPIO_CRH_MODE13,
    GPIO_CRH_MODE13_0);
```

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

```
WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13); // Low
WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13); // High
```

Man kann auch mehrere Pins gleichzeitig schalten:

```
WRITE_REG(GPIOB->BSRR,GPIO_BSRR_BR10 + GPIO_BSRR_BR11);
```

Jetzt kommt etwas neues. Das ODR Register repräsentiert den aktuellen Zustand aller 16 Pins von einem Port. Im Referenzhandbuch ist es so dargestellt:

Port output data register (GPIOx_ODR) (x=A..G)

Address offset: 0x0C

Reset value: 0x0000 0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
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 (y= 0 .. 15)
These bits can be read and written by software and can be accessed in Word mode only.

Abbildung 79: Auszug aus dem Referenzhandbuch: Beschreibung des ODR Registers

Dieses Register kann man sowohl auslesen als auch beschreiben:

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

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

```
GPIOB->ODR=0b0000000000000000;  // setze alle Pins auf Low
GPIOB->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.

9.17 Analoge Eingänge

Die Anschlüsse PA0 bis PA7, sowie PB0 und PB1 können auch analog verwendet werden. Dass heißt, jede beliebige Spannung zwischen 0 Volt und 3,3 Volt kann gemessen und verarbeitet werden.

Um einen Anschluss analog zu verwenden, muss du ihn im Register CRL konfigurieren. Wir wollen das mal mit dem Anschluss PA1 versuchen.

Erweitere dazu die Funktion „init_io“ um folgende Zeilen:

```
// Konfiguriere analoge Eingänge
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF1 + GPIO_CRL_MODE1, 0);
```

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

```
// Initialisiere den Aanalog zu digital Konverter
void init_analog(void)
{
    // Aktiviere die Stromversorgung des ADC
    SET_BIT(RCC->APB2ENR, RCC_APB2ENR_ADC1EN);

    // Schalte den ADC ein
    SET_BIT(ADC1->CR2, ADC_CR2_ADON);

    // Die Messung soll per Software gestartet werden
    MODIFY_REG(ADC1->CR2, ADC_CR2_EXTSEL,
        ADC_CR2_EXTSEL_0 + ADC_CR2_EXTSEL_1 + ADC_CR2_EXTSEL_2);

    // Stelle eine mittlere Mess-Zeit ein
    MODIFY_REG(ADC1->SMPR2, ADC_SMPR2_SMP0, ADC_SMPR2_SMP0_2);

    // Warte 20 Millisekunden vor der Kalibrierung
    uint32_t start=systick_count;
    while (systick_count-start<20);

    // Starte die Kalibrierung
    SET_BIT(ADC1->CR2, ADC_CR2_ADON + ADC_CR2_CAL);

    // Warte, bis die Kalibrierung abgeschlossen ist
    while (READ_BIT(ADC1->CR2, ADC_CR2_CAL));
}
```

Als nächstes braucht du eine Funktion, die den analogen Eingang einliest. Kopiere das:

```
// Lese von einem analogen Eingang
uint16_t read_analog(int channel)
{
    // Wähle den Kanal aus
    MODIFY_REG(ADC1->SQR3, ADC_SQR3_SQ1, channel);

    // Lösche die "beendet" Markierung
    CLEAR_BIT(ADC1->SR, ADC_SR_EOC);

    // Beginne eine Messung
    SET_BIT(ADC1->CR2, ADC_CR2_ADON);
    SET_BIT(ADC1->CR2, ADC_CR2_SWSTART);

    // Warte bis die Messung beendet ist
    while (!READ_BIT(ADC1->SR, ADC_SR_EOC));

    // Liefere nur die unteren 12 Bit des Ergebnisses ab
    return ADC1->DR & 0b111111111111;
}
```

Die Funktion erwartet als Parameter eine Kanal-Nummer. Die Kanäle 0 bis 7 entsprechen den Anschlüssen PA0 bis PA7. Die Kanäle 8 und 9 entsprechen den Anschlüssen PB0 und PB1.

Kopiere diese „main“ Funktion:

```
int main(void)
{
    // Initialisiere die I/O Pins
    init_io();

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

    // Initialisiere analoge Eingänge und den ADC
    init_analog();

    while(1)
    {
        // Verzögere eine Sekunde
        uint32_t start=systick_count;
        while (systick_count-start<1000);

        // Lese den analogen Eingang
        uint16_t a=read_analog(1);
        printf("analog: %i \n", a);
    }
}
```

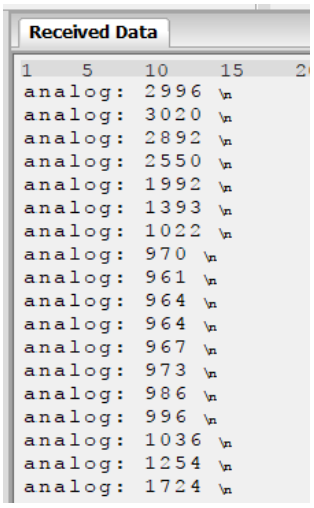


Abbildung 80: Serielle Ausgabe im Hammer-Terminal

Hier ist wichtig, dass der System-Timer vor der Funktion „init_analog“ ausgeführt 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 der obigen Abbildung entsprechen.

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

Verbinde den Eingang PA1 nun mit GND. Du erhältst sofort den Messwert 0. Verbinde den Eingang PA1 mit 3,3V. Du erhältst den Messwert 4095.

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 ausprobieren.

Baue die folgende Schaltung auf:

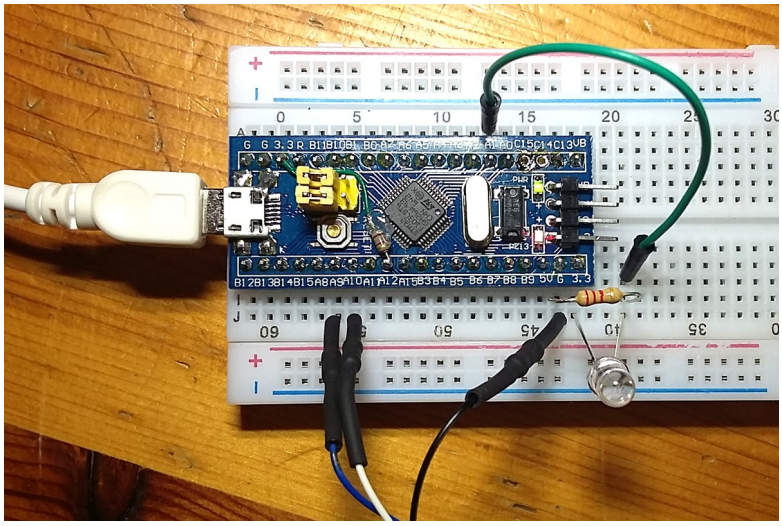


Abbildung 81: Aufbau vom Fototransistor mit Pull-Down Widerstand an Port A1

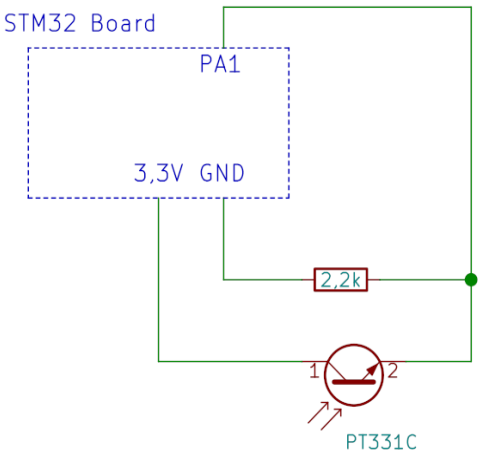


Abbildung 82: Fototransistor mit Pull-Down Widerstand an Port A1

Das große durchsichtige Ding, dass wie eine LED aussieht, ist der Fototransistor. Dessen langer Anschluss gehört nach rechts.

Der Widerstand zieht den Anschluss PA1 auf 0 Volt (GND).

Der Fototransistor zieht den Anschluss PA1 jedoch hoch auf bis zu 3,3 Volt. Jede mehr Licht auf ihn fällt, umso höher ist die Spannung, weil der Fototransistor den elektrischen Strom mit zunehmender Helligkeit besser leitet.

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

Hinweis: Wenn du die Ausgabe im Terminal-Programm lange Zeit laufen lässt, geraten die Buchstaben irgendwann durcheinander. Das ist ein Fehler im Terminal Programm. Er verschwindet, wenn du auf den „Disconnect“ Knopf klickst und dann wieder auf „Connect“.

Übungsaufgabe:

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

9.18 Bit-Operationen

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

```
// Liefere nur die unteren 12 Bit des Ergebnisses ab
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
```

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

10 Pinbelegung

Für den Fall dass die Pinbelegung deines Mikrocontroller Boards unleserlich ist, habe ich sie hier noch einmal schön groß notiert:

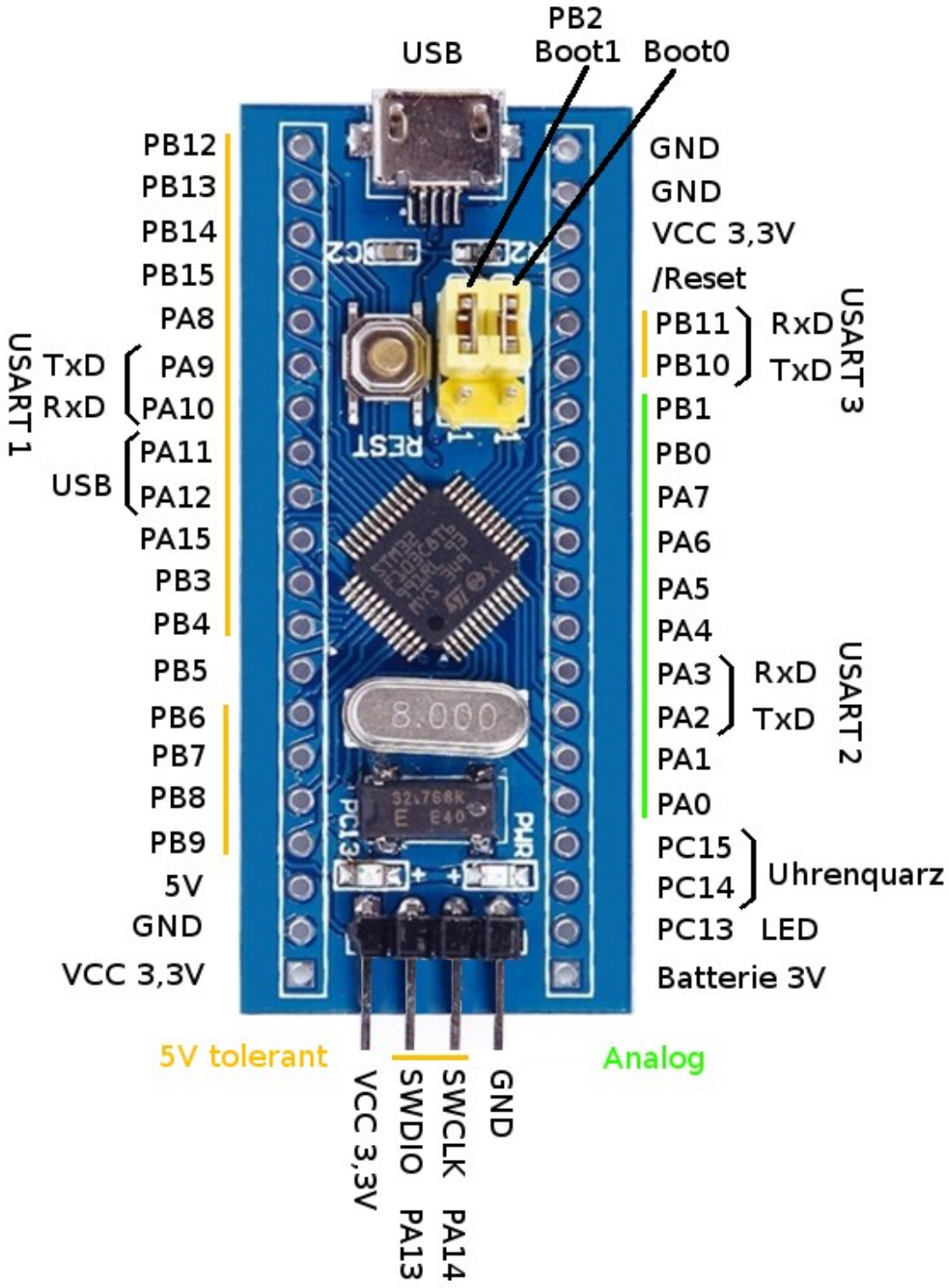


Abbildung 83: Pinbelegung des Blue Pill Boardes

Die grüne Power-LED ist mit der Versorgungsspannung (VCC) verbunden.

Die andere LED leuchtet, wenn man PC13 als Ausgang auf Low Pegel schaltet.

PC14 und PC15 sind mit dem schwarzen Uhren-Quarz verbunden. Der Batterieanschluss ist ebenfalls für die Uhr, damit sie bei Stromausfall weiter läuft. Im Rahmen dieses Buches werden wir die Uhr aber nicht benutzen.

PA11 und PA12 sind mit den Datenleitungen der USB Buchse verbunden. Daher darfst du sie nicht benutzen, wenn die USB Buchse belegt ist.

Somit bleiben dir 30 Anschlüsse für allgemeine Verwendung übrig. Diese nennt man üblicherweise „I/O Pins“ oder „GPIO Pins“ (General Purpose Input and Output).

Die Ausgänge sind einzeln mit bis zu 25 mA belastbar, alle zusammen jedoch maximal 150 mA.

Auf dem Board befindet sich ein Spannungsregler, der aus den grob geregelten 5 V vom Netzteil eine sehr stabile 3,3 V Versorgungsspannung gewinnt. Damit wird nicht nur der Mikrocontroller versorgt, sondern auch die externen Bauteile, die du mit ihm verbindest.

Der 5 V Anschluss kann alternativ zur USB Buchse für die Stromversorgung verwendet werden.

11 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.

11.1 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 weiße LED wird als Modell für eine größere Lampe dienen:

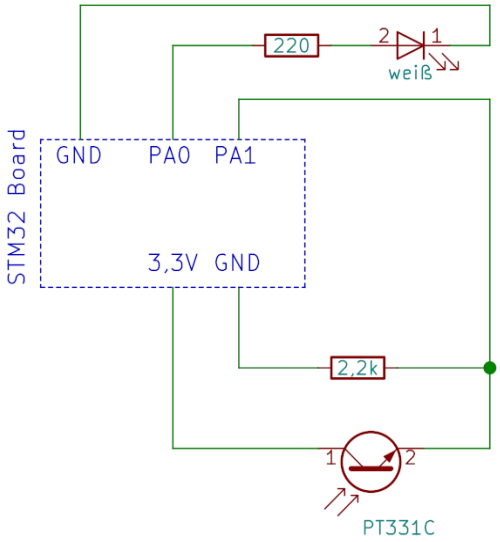


Abbildung 84: Dämmerungsschalter mit Fototransistor an PA1 und LED an PA0

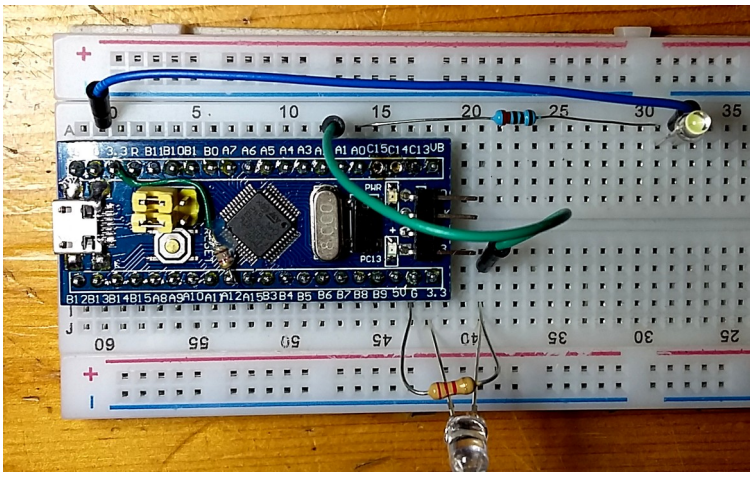


Abbildung 85: Aufbau vom Dämmerungsschalter

Beachte beim Aufbauen, dass der lange Anschluss der LED nach links gehört. Beim Fototransistor gehört der lange Anschluss nach rechts. Die Schaltung funktioniert so:

Der 2,2k Ohm Widerstand zieht den analogen Eingang PA1 auf 0 Volt (GND). Der Fototransistor hingegen zieht das Signal hoch auf 3,3 Volt. Je heller es ist, umso besser leitet der Transistor, so dass die Spannung an PA1 höher ist.

Am Anschluss PA0 hängt die weiße Leuchtdiode. Im Gegensatz zu den vorherigen Experimenten ist die LED dieses mal so angeschlossen worden, dass sie leuchtet, wenn der Ausgang PA0 einen High Pegel (3,3V) hat.

Erstelle ein neues Programm auf Basis der „Blinker“ Vorlage. Zur Initialisierung der Anschlüsse PA0 und PA1 musst du die Funktion „init_io“ erweitern:

```
// PA0 = Ausgang für die weiße LED
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF0 + GPIO_CRL_MODE0, GPIO_CRL_MODE0_0);

// PA1 = Analoger Eingang für den Helligkeits-Sensor
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF1 + GPIO_CRL_MODE1, 0);
```

Kopiere die beiden Funktionen „init_analog“ und „read_analog“ aus dem Kapitel Analoge Eingänge.

Die Funktion „main“ soll so aussehen:

```
int main(void)
{
    // Initialize I/O pins
    init_io();

    // Initialize system timer
    SysTick_Config(SystemCoreClock/1000);

    // Initialisiere den ADC
    init_analog();

    while(1)
    {
        // Helligkeit messen
        uint16_t helligkeit=read_analog(1);
        printf("Helligkeit %i \n", helligkeit);

        // Wenn es dunkel ist
        if (helligkeit < 300)
        {
            // Licht an
            WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS0);
        }
        else
        {
            // Licht aus
            WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR0);
        }

        // Verzögere eine Sekunde
        uint32_t start=systick_count;
        while (systick_count-start<1000);
    }
}
```

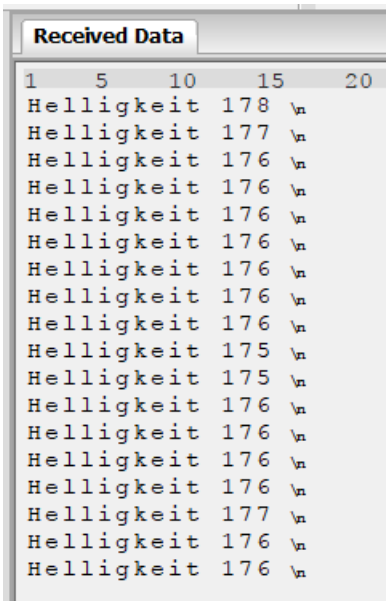


Abbildung 86: Serielle Ausgabe im Hammer-Terminal

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.

Schau dir mal die Ausgaben an. Wie du siehst, schwanken die Messwerte immer ein kleines bisschen. Wenn jetzt die Schaltschwelle 176 oder 177 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 im Programm so, das er dem Mittelwert der gerade aktuellen Ausgabe entspricht.

Um den Dämmerungsschalter zu verbessern, führt man eine sogenannte „Hysterese“ ein. Das ist ein Bereich, in dem die „Lampe“ nicht umgeschaltet wird. Passend zur obigen Ausgabe schlage ich vor, die „Lampe“ bei 150 ein zu schalten und bei 200 aus zu schalten.

Die Hysterese ist dann der „tote“ Bereich dazwischen, den das Programm ignorieren soll, also 50. Das ist die Differenz zwischen 200 und 150.

Gesagt, getan:

```
// Wenn es dunkel ist
if (helligkeit < 150)
{
    // Licht an
    WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS0);
}

// Wenn es hell ist
else if (helligkeit > 200)
{
    // Licht aus
    WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR0);
}
```

Wenn die Helligkeit jetzt ein bisschen schwankt, macht das nichts aus. Teste das Ergebnis. Eventuell möchtest du andere Schaltschwellen verwenden oder die Hysterese verringern.

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.

Ich möchte dir noch eine Feinheit der Programmiersprache C zeigen. Diese Verzögerungsschleife kann man auf eine Zeile reduzieren.

Aus:

```
uint32_t start=systick_count;
while (systick_count-start<1000);
```

wird:

```
for (uint32_t start=systick_count; systick_count-start<1000;);
```

Das habe ich mal irgendwo in einem fremden Programm gesehen. Mir gefällt es nicht, aber dir vielleicht.

Bei diesem 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 zueinander zeigen:

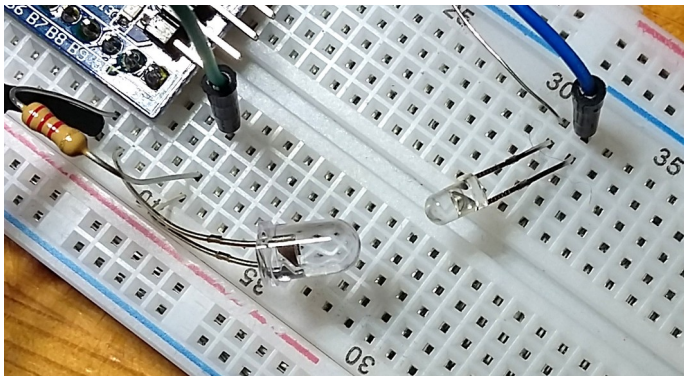


Abbildung 87: Schwingung durch optische Rückkoppelung

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. Außerdem muss die Hysterese groß genug sein, dass der Dämmerungsschalter nicht voreilig auf winzige Helligkeits-Schwankungen reagiert.

Übungsaufgabe:

Messe die analoge Spannung an den beiden Beinchen des Fototransistors bei unterschiedlichen Helligkeiten. Erkläre, warum die Spannung sinkt, wenn es heller wird.

Messe nun die analoge Spannung zwischen GND und dem Anschluss PA1. Erkläre, warum die Spannung an dieser Stelle steigt, wenn es heller wird.

11.2 Eieruhr

Die Eieruhr hilft beim Kochen von idealen Frühstückseiern. Nach genau 3 Minuten erzeugt sie einen Signalton.

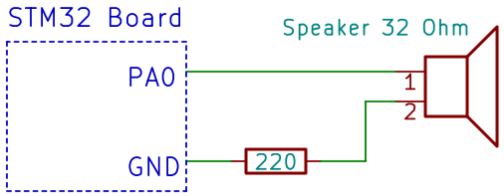


Abbildung 88: Lautsprecher an Port A0

Der Lautsprecher wird zusammen mit einem Widerstand verwendet, um die Stromstärke zu reduzieren. Denn ein wesentlich größerer Strom würde den Mikrocontroller und vielleicht auch den Lautsprecher zerstören.

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

I = 3,3 V / 252 Ω = 0,013 A = 13 mA

Der Mikrocontroller verträgt maximal 20 mA, passt also.

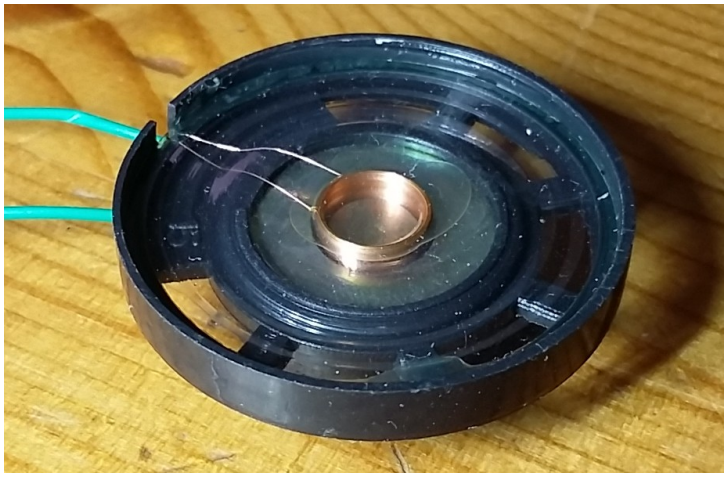


Abbildung 89: Ansicht der Spule, die den Lautsprecher antreibt

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:

```
int main(void)
{
    // Initialize I/O pins
    init_io();

    // Initialize system timer
    SysTick_Config(SystemCoreClock/1000);

    while(1)
    {
        // Lautsprecher auf High schalten
        WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS0);

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

        // Lautsprecher auf Low schalten
        WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR0);

        // Warte 1ms
        start=systick_count;
        while (systick_count-start<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 lauter:

Abbildung 90: 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 1ms gezogen. 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

Um andere Frequenzen zu erzeugen, müsste man die Intervalle des System-Timers ändern, aber das wäre sehr unüblich. Stattdessen schreiben wir die Warteschleifen um:

```
// Warte 500µs
for (int j=0; j<1000; j++)
{
    __NOP ();
}
```

Der Ton hat jetzt 1 kHz, das ist genau eine Oktave höher. Probiere es aus.

Wenn die Schleife 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.

Diese neue Warteschleife führt also 1000 mal den Befehl NOP aus, was 500 µs dauert.

Solche Warteschleifen hängen von der Taktfrequenz des Mikrocontrollers ab. Momentan benutzt du den Mikrocontroller mit einer Taktfrequenz von 8 Mhz. Man könnte die Taktfrequenz theoretisch auf bis zu 72 MHz erhöhen, dann würden die Warteschleifen viel schneller ablaufen.

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

```
// Erzeuge einen Ton mit der angegebenenn Frequenz und Dauer
void ton(int frequenz, int millisekunden)
{
    int wartezeit=1000000/frequenz;
    uint32_t start=systick_count;

    // Wiederhole so und so viele Millisekunden lang
    while (systick_count-start<millisekunden)
    {
        // Lautsprecher auf High schalten
        WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS0);

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

        // Lautsprecher auf Low schalten
        WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR0);

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

Probiere die Funktion mit folgendem Hauptprogramm aus:

```
int main(void)
{
    // Initialize I/O pins
    init_io();

    // Initialize 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(void)
{
    // Initialize I/O pins
    init_io();

    // Initialize 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(void)
{
    // Initialize I/O pins
    init_io();

    // Initialize system timer
    SysTick_Config(SystemCoreClock/1000);

    // Warte 3 Minuten
    uint32_t start=systick_count;
    while (systick_count-start < 3*60*1000 )
    {
        // Einmal kurz Piepsen
        ton(1000,50);

        // Warte 1 Sekunde
        uint32_t start2=systick_count;
        while (systick_count-start2 < 1000);
    }

    // Und jetzt kommt der ultimative Sound-Effekt :-)
    while(1)
    {
        ton(600,50);
        ton(800,50);
        ton(1000,50);
        ton(1200,50);
    }
}
```

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.

11.3 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 Schaltplan ist mit dem obigen Dämmerungsschalter identisch:

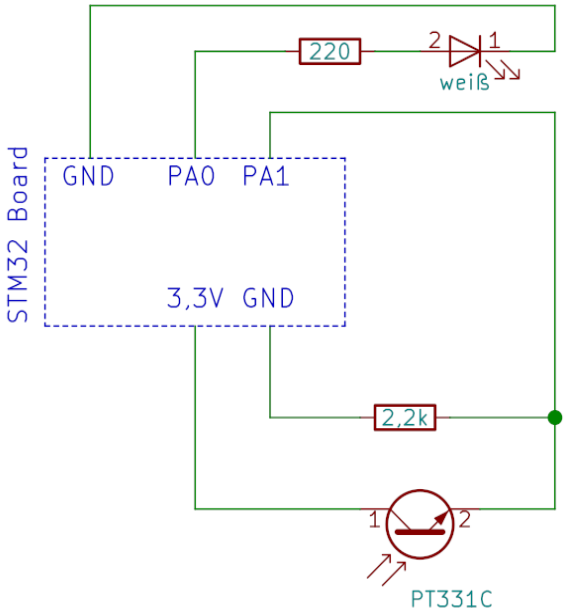


Abbildung 91: Schaltung der Lichtschranke mit LED und Fototransistor

Auf dem Steckbrett sollen die Bauteile jedoch anders angeordnet werden. Nämlich so, dass die LED auf den Fototransistor zeigt und dass dazwischen ein paar Zentimeter Platz frei sind.

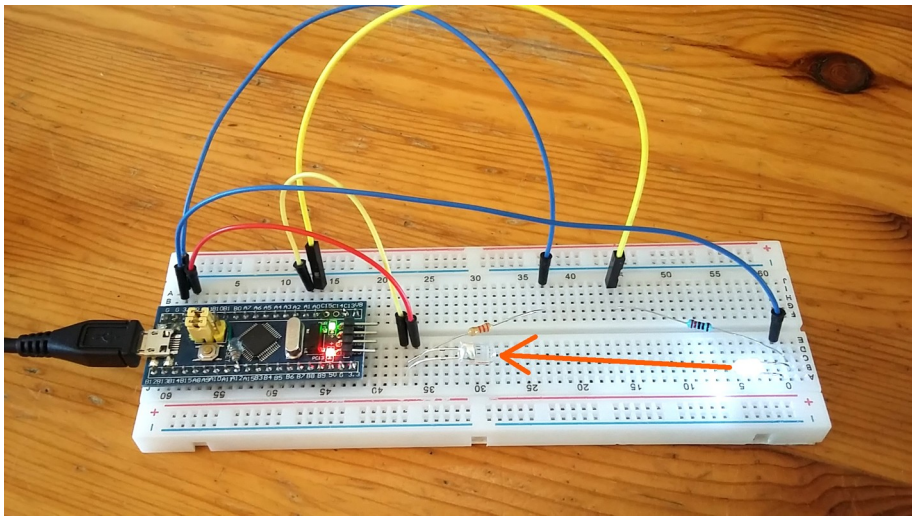


Abbildung 92: Aufbau der Lichtschranke mit LED und Fototransistor

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.

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:

```
// PA0 = Ausgang für die weiße LED
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF0 + GPIO_CRL_MODE0, GPIO_CRL_MODE0_0);

// PA1 = Analoger Eingang für den Helligkeits-Sensor
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF1 + GPIO_CRL_MODE1, 0);
```

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:

```
uint16_t messen(void)
{
    // Weiße LED aus
    WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BR0);

    // Warte
    uint32_t start=systick_count;
    while (systick_count-start<200);

    // Umgebungslicht messen
    uint16_t umgebung=read_analog(1);
    printf("Umgebung %i \n",umgebung);

    // Weiße LED an
    WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS0);

    // Warte
    start=systick_count;
    while (systick_count-start<200);

    // Helligkeit messen
    uint16_t helligkeit=read_analog(1);
    printf("Helligkeit %i \n",helligkeit);

    // Differenz bilden
    uint16_t differenz=helligkeit-umgebung;
    printf("Differenz %i \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(void)
{
    // Initialize I/O pins
    init_io();

    // Initialize system timer
    SysTick_Config(SystemCoreClock/1000);

    // Initialisiere den ADC
    init_analog();

    while(1)
    {
        if (messen() > 500)
        {
            // LED an
            WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13);
        }
        else
        {
            // LED aus
            WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13);
        }
    }
}
```

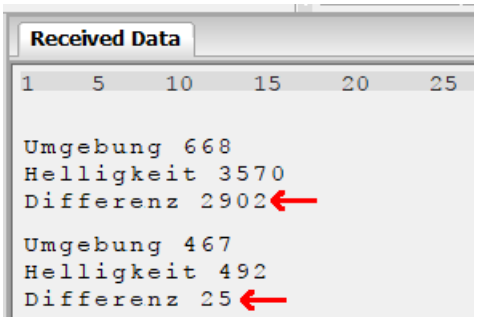


Abbildung 93: Serielle Ausgabe im Hammer-Terminal

Wenn genug Licht auf den Sensor fällt, soll die LED an gehen. Wenn wenig Licht auf den Sensor fällt, soll sie aus gehen.

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.

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 das Terminal-Programm nicht mehr benutzen, weil es nun mit den schnellen Ausgaben überflutet wird.

Übungsaufgabe:

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

11.4 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 94: 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 PB0 hoch ziehen. Je wärmer es ist ist, umso höher wird die Spannung sein.

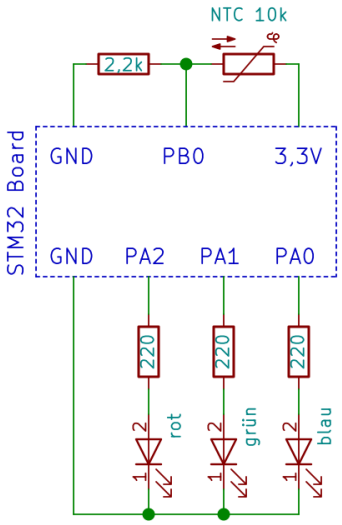


Abbildung 95: Schaltplan des Thermometers

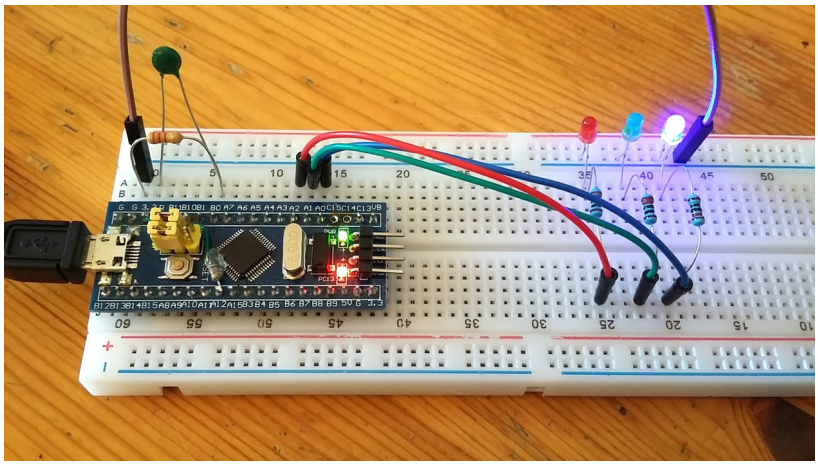


Abbildung 96: Aufbau des Thermometers

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 die LED und den Temperatursensor richtig konfiguriert sind:

```
// PA0 = Ausgang für die blaue LED
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF0 + GPIO_CRL_MODE0, GPIO_CRL_MODE0_0);

// PA1 = Ausgang für die grüne LED
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF1 + GPIO_CRL_MODE1, GPIO_CRL_MODE1_0);

// PA2 = Ausgang für die rote LED
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF2 + GPIO_CRL_MODE2, GPIO_CRL_MODE2_0);

// PB0 = analoger Temperatursensor
MODIFY_REG(GPIOB->CRL, GPIO_CRL_CNF0 + GPIO_CRL_MODE0, 0);
```

Das Hauptprogramm sieht so aus:

```
int main(void)
{
    // Initialize I/O pins
    init_io();

    // Initialize system timer
    SysTick_Config(SystemCoreClock/1000);

    // Initialisiere den ADC
    init_analog();

    while(1)
    {
        // messen
        uint16_t temperatur=read_analog(8);
        printf("Temperatur %i \n",temperatur);

        if (temperatur>700)
        {
            // blau=aus, grün=aus, rot=an
            WRITE_REG(GPIOA->BSRR,
                GPIO_BSRR_BR0 + GPIO_BSRR_BR1 + GPIO_BSRR_BS2);
        }
        else if (temperatur>650)
        {
            // blau=aus, grün=an, rot=aus
            WRITE_REG(GPIOA->BSRR,
                GPIO_BSRR_BR0 + GPIO_BSRR_BS1 + GPIO_BSRR_BR2);
        }
        else
        {
            // blau=an, grün=aus, rot=aus
            WRITE_REG(GPIOA->BSRR,
                GPIO_BSRR_BS0 + GPIO_BSRR_BR1 + GPIO_BSRR_BR2);
        }

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

Schau dir die Ausgaben im Terminal-Programm an. Bei mir wird der Messwert 675 angezeigt, deswegen habe ich die Schaltschwellen ein bisschen darunter und darüber eingestellt (650 und 700).

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 „zu hell“ bzw. „zu dunkel“ meldet.

11.5 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 97: Reed-Kontakt

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

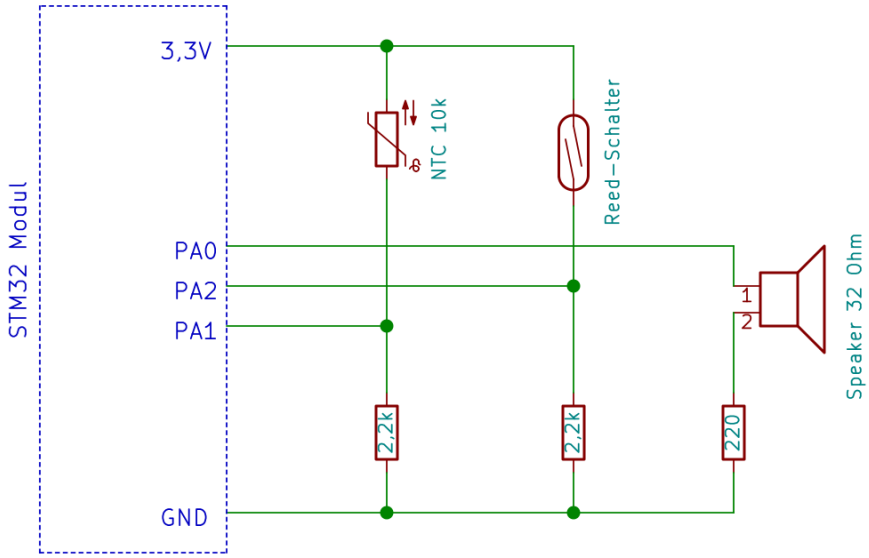


Abbildung 98: Schaltplan vom Kühlschrank-Alarm

Die Eingangsspannung an PA1 ist niedrig wenn der Kühlschrank kalt genug ist. Sie steigt an, wenn der Kühlschrank wärmer wird.

Der Eingang PA2 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.

Mein Aufbau sieht so aus:

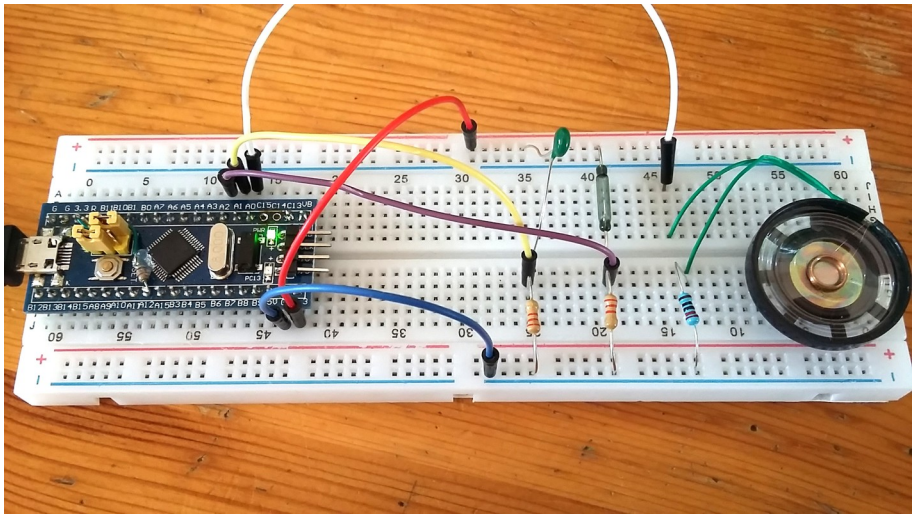


Abbildung 99: Aufbau des Kühlschrank-Alarms

Lege ein neues Projekt auf Basis der „Blinker“ Vorlage an. Kopiere die beiden Funktionen „init_analog“ und „read_analog“ aus dem Kapitel Analoge Eingänge. Kopiere außerdem die Funktion „ton“ aus dem Kapitel Eieruhr.

Erweitere die „init_io“ Funktion, damit die Anschlüsse richtig konfiguriert sind:

```
// PA0 = Ausgang Lautprecher
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF0 + GPIO_CRL_MODE0, GPIO_CRL_MODE0_0);

// PA1 = analoger Temperaturfühler
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF1 + GPIO_CRL_MODE1, 0);

// PA2 = Eingang Reed Kontakt
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF2 + GPIO_CRL_MODE2, GPIO_CRL_CNF2_1);
```

Die Initialisierung vom Anschluss PA2 ist nicht unbedingt nötig, weil ohnehin alle Anschlüsse standardmäßig digitale Eingänge sind.

Das Hauptprogramm sieht so aus:

```
int main(void)
{
    // Initialize I/O pins
    init_io();

    // Initialize system timer
    SysTick_Config(SystemCoreClock/1000);

    // Initialisiere den ADC
    init_analog();

    // Beginn der Zeit, wo ein Problem erkannt wurde
    uint32_t okZeit=0;
```

```
    while(1)
    {
        // messen
        uint16_t temperatur=read_analog(1);
        printf("Temperatur %i \n",temperatur);

        // Ist es kalt genug und ist die Türe geschlossen?
        if (temperatur<800 && READ_BIT(GPIOA->IDR, GPIO_IDR_IDR2))
        {
            // Kein Problem
            okZeit=systick_count;

            // LED aus
            WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BS13);
        }
        else
        {
            // LED an
            WRITE_REG(GPIOC->BSRR,GPIO_BSRR_BR13);

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

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

Lege für den Anfang den Magneten neben den Reed-Kontakt. 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. Warscheinlich 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 LED an. Wenn du 20 Sekunden länger abwartest, gibt der Lautsprecher zusätzlich einen Alarm-Ton von sich.

Lass 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.

11.6 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 Schaltplan vor:

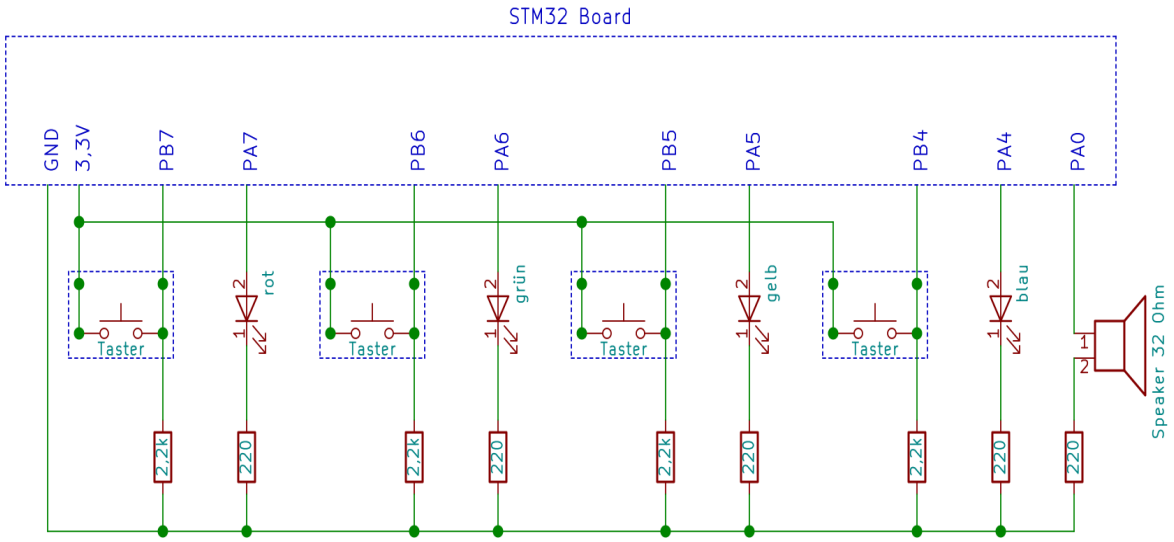


Abbildung 100: Schaltplan vom Quiz-Buzzer

Mein Draht-Verhau sieht so aus:

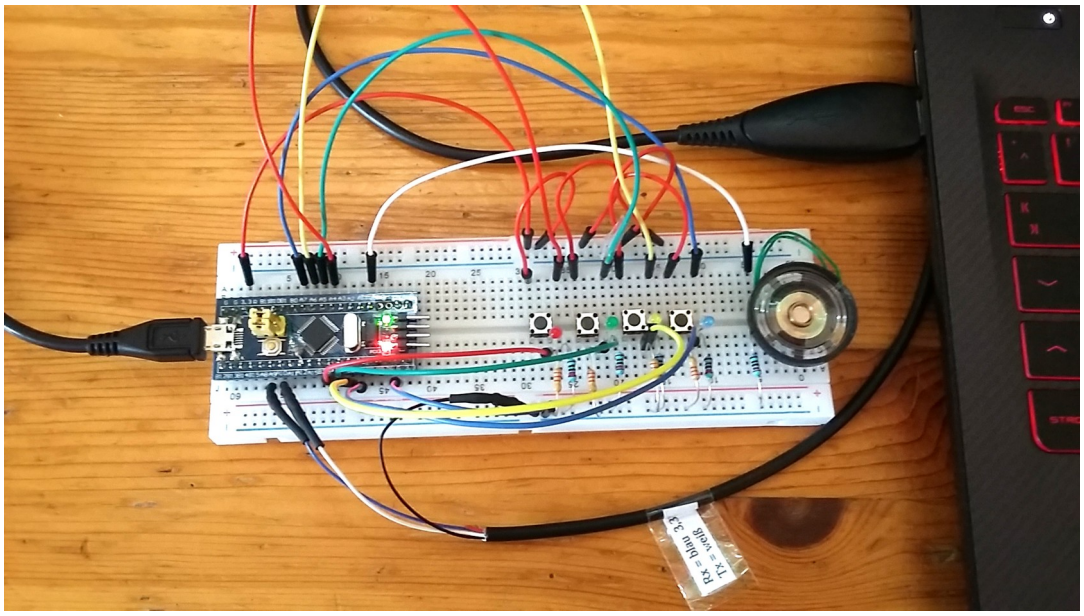


Abbildung 101: Aufbau des Quiz-Buzzers

Lege ein neues Projekt auf Basis der „Blinker“ Vorlage an und kopiere die Funktion „ton“ aus dem Kapitel Eieruhr.

Erweitere die „init_io“ Funktion, damit die Anschlüsse richtig konfiguriert sind:

```
// PA0 = Ausgang Lautprecher
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF0 + GPIO_CRL_MODE0, GPIO_CRL_MODE0_0);

// PA4,5,6,7 = Ausgänge für LEDs
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF4 + GPIO_CRL_MODE4, GPIO_CRL_MODE4_0);
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF5 + GPIO_CRL_MODE5, GPIO_CRL_MODE5_0);
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF6 + GPIO_CRL_MODE6, GPIO_CRL_MODE6_0);
MODIFY_REG(GPIOA->CRL, GPIO_CRL_CNF7 + GPIO_CRL_MODE7, GPIO_CRL_MODE7_0);
```

Das Programm kann man so schreiben:

```
int main(void)
{
    // Initialize I/O pins
    init_io();

    // Initialize system timer
    SysTick_Config(SystemCoreClock/1000);

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

        // Wenn Spieler "rot" gedrückt hat
        if (READ_BIT(GPIOB->IDR, GPIO_IDR_IDR4))
        {
            // rote LED an
            WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS4);
            break;
        }

        // Wenn Spieler "grün" gedrückt hat
        if (READ_BIT(GPIOB->IDR, GPIO_IDR_IDR5))
        {
            // grüne LED an
            WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS5);
            break;
        }

        // Wenn Spieler "gelb" gedrückt hat
        if (READ_BIT(GPIOB->IDR, GPIO_IDR_IDR6))
        {
            // gelbe LED an
            WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS6);
            break;
        }

        // Wenn Spieler "blau" gedrückt hat
        if (READ_BIT(GPIOB->IDR, GPIO_IDR_IDR7))
        {
            // blaue LED an
            WRITE_REG(GPIOA->BSRR,GPIO_BSRR_BS7);
            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 Hauptschleife 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 Sound-Effekt erzeugt bevor das Programm endet.

Der Moderator kann den Reset-Knopf drücken, um das Programm neu zu starten.

Übungsaufgaben:

1. Erweitere die Schaltung und das Programm für 5 Spieler.
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 ausgegeben wird.
4. Erfinde für jeden Spieler einen eigenen Soundeffekt.
5. 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.

11.6.1 Fester Aufbau

Falls du diese Schaltung zur echten 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-Modul würde ich nicht direkt fest löten, sondern besser in **Buchsen-Leisten** stecken, 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.

12 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.