29.03.2025

Vario

Microcomputertechnik PStA von Bastian Herold und Leonard Rother

Inhaltsverzeichnis

[1. Vorwort 2](#_Toc194140123)

[2. Aufgabenstellung 2](#_Toc194140124)

[3. Informationsbeschaffung 2](#_Toc194140125)

[4. Schaltplan 3](#_Toc194140126)

[5. Programmablaufplan 3](#_Toc194140127)

[6. Programmierung 6](#_Toc194140128)

[6.1. Cube MX 6](#_Toc194140129)

[6.2. Code 7](#_Toc194140130)

[6.2.1. LED blinken 7](#_Toc194140131)

[6.2.2. Analoge Spannung einlesen 7](#_Toc194140132)

[6.2.3. I2C Luftdrucksensor 8](#_Toc194140133)

[6.2.4. Buzzer 10](#_Toc194140134)

[6.2.4.1. Timer-Konfiguration in CubeMX 10](#_Toc194140135)

[6.2.4.2. Implementierung der Tonsteuerung 10](#_Toc194140136)

[6.2.4.3. BeepsPerSecond-Steuerung mit Timer 6 11](#_Toc194140137)

[6.2.5. Display 11](#_Toc194140138)

[7. Versionsverwaltung via Github 12](#_Toc194140139)

[8. Bilder 13](#_Toc194140140)

[9. Lessons learned 13](#_Toc194140141)

[10. Anhang 15](#_Toc194140142)

# Vorwort

Kurz zu mir, Bastian Herold:

Ich habe bereits sowohl in der Arbeit als auch Privat schon einige Projekte mit STM32 Microcontrollern gemacht. Deswegen habe ich schon Erfahrung was den Umgang mit den Tools und die Programmierung angeht. Deshalb musste von mir nicht so oft „gegoogelt“ werden um ein bestimmtes Problem zu lösen.

Ich bin es gewohnt, Variablennamen und Kommentare auf englisch zu schreiben, darum ist das hier genauso.

# Aufgabenstellung

|  |  |  |
| --- | --- | --- |
| Aufgabe | Aufgabenstellung | Status |
| A | Messen des Akkufüllstandes | Fertig |
| A | Anzeige des aktuellen Akkufüllstands in % auf dem Display. | Fertig |
| B | Luftdrucksensor zyklisch auslesen (= Polling). | Fertig |
| B | Wenn ein Schwellwert der Geschwindigkeit v überschritten wird, dies über einen Buzzer akustisch ausgeben (→ PWM). | Fertig |
| B | Umrechnen mit QNH auf absolute Höhe. | Fertig(mit statischen QNH). |
| B | Umrechnen zu einer vertikalen Bewegungsgeschwindigkeit in m/s. | Fertig |
| B | Anzeige von Höhe und vertikaler Geschwindigkeit auf dem Display. | Fertig |

# Informationsbeschaffung

Um mit dem Projekt zu starten, war es zunächst wichtig, relevante Informationen zu den verwendeten Komponenten zu sammeln. Dazu habe ich im Internet einfach die Bezeichnung der Komponente zusammen mit dem Begriff "datasheet" gesucht – meist war bereits eines der ersten Ergebnisse ein Treffer.

Verwendete Komponenten:

* Microcontroller: STM32L476 Nucleo – NUCLEO-L476RG
* Display: I2C 1602 LCD (2 Zeilen mit je 16 Zeichen)
* Buzzer: Standard-Arduino-Buzzer
* Barometer: Bosch BMP280 (I2C) – aus dem letzten Semester E-Praxis

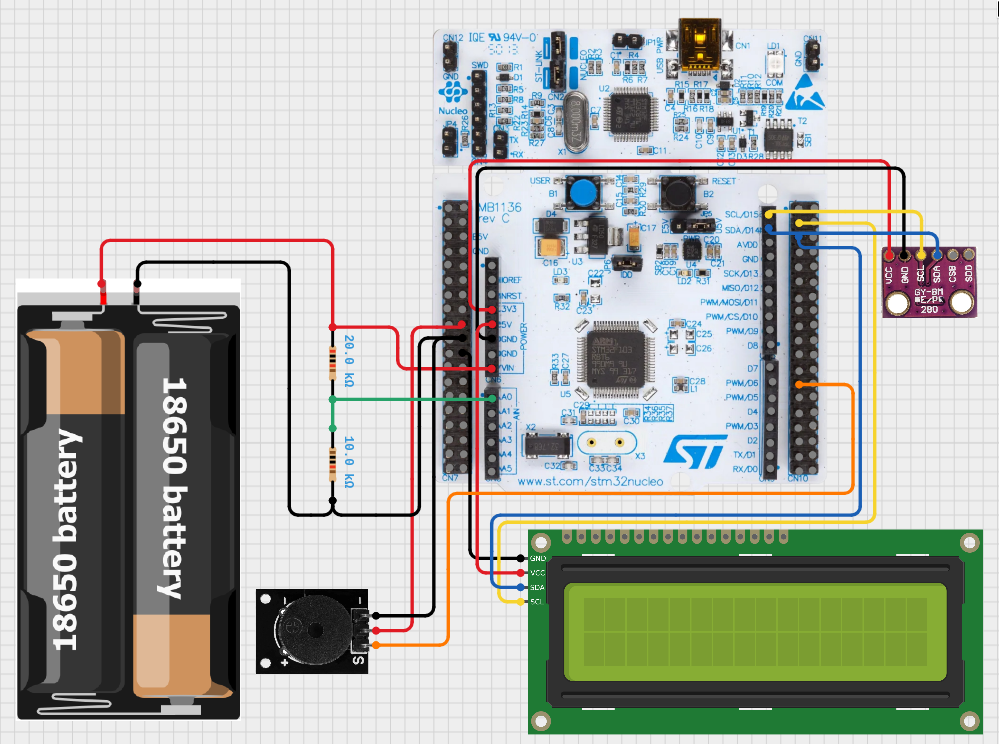
Für den BMP280 habe ich zwar das Datenblatt von Adafruit sowie deren Sensormodul verwendet, da es meiner Meinung nach deutlich besser und genauer beschrieben ist als das offizielle Datenblatt von Bosch.

# Schaltplan

Nach dem Lesen der Aufgabenstellung habe ich angenommen, dass keine technische Schaltung, sondern eine visuelle Verknüpfung der Bauteile erwünscht ist. Deshalb habe ich mich gegen eine Zeichnung in KiCad entschieden und nach schönen grafischen Editor gesucht, wie die Bilder aus Arduino Anleitungen.

Youtube: draw arduino sketch software, ergab das Ergebnis: <https://www.youtube.com/watch?v=hEsd1u290rQ.> Welches auf Cirkit Design verwiesen hat( <https://www.cirkitstudio.com/).> Ohne das YouTube Video anzuschauen habe ich mir Cirkit Design Web einfach ausprobiert und da es alle notwendigen Module hatte entschied ich mich dafür. Ein weiteres Forum hat noch Fritzing empfohlen, aber nach dem langen Download war ich mit Cirkit Designer schon fertig und bei Fritzing konnte ich den BMP280 nicht finden.

Daher entschied ich mich für Cirkit Designer, aber Fritzing macht auch einen Professionellen Eindruck mit einen anderen Schwerpunkt an Möglichkeiten und für jeden interessierten bestimmt eine Empfehlung wert. Der fertige, visuelle Schaltplan:



# Programmablaufplan

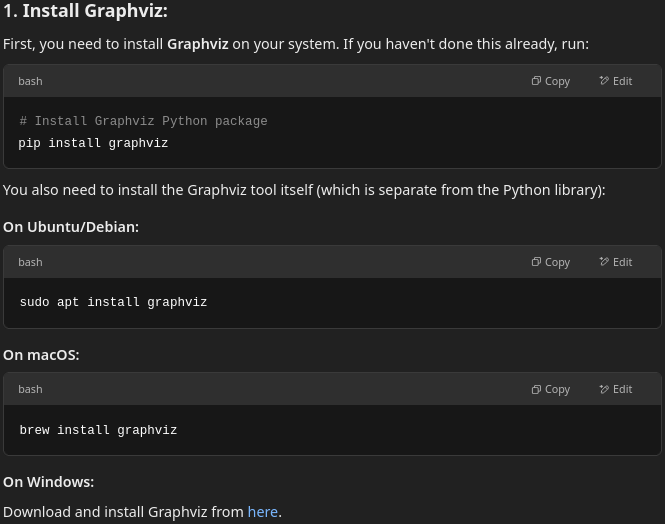
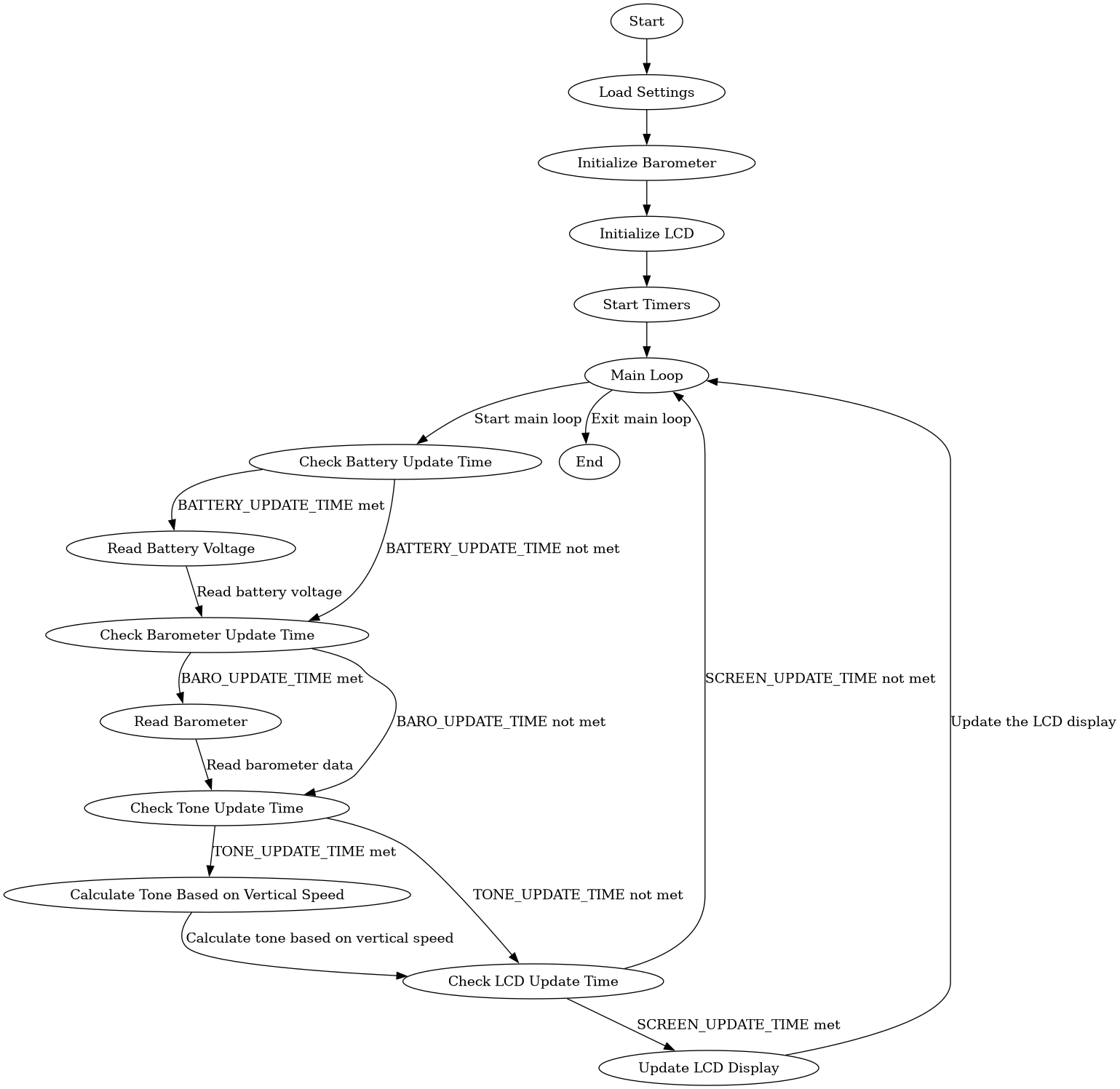
Einen PAP zu schreiben fällt mir schwer, da der Code von zwei Personen geschrieben ist und ich somit nicht intuitiv den Ablauf komplett weiß.

Um die Aufgabe interessanter zu Gestalten, wollte ich moderne Möglichkeiten ausprobieren, indem ich diesen Teil der Aufgabe automatisch erzeugen lasse. Ich wusste, dass man UML automatisch erzeugen kann und habe das zunächst ausprobiert. Nach etwas rumprobieren(1h), wurde meine Vermutung durch ein Forum bestätigt(https://stackoverflow.com/questions/49817793/how-to-generate-uml-sequence-diagrams-from-c-code), dass man UML in C nicht sinnvoll automatisch generieren kann, da C nicht Objektorientiert ist.

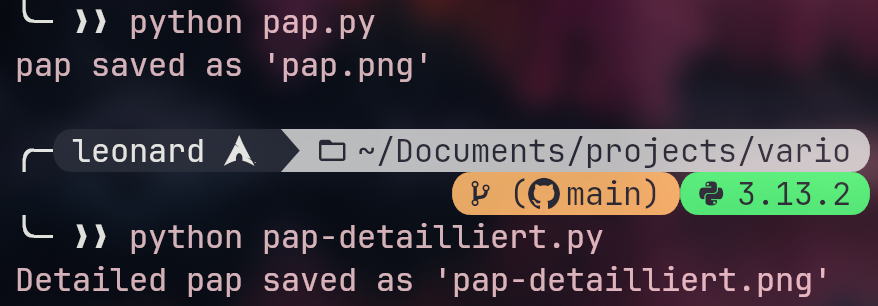
Einen funktionierenden Struktogramm-generator konnte ich auch nicht finden. Die App: Structorizer hat uint nicht erkannt und war daher keine Option.

Der letzte Weg war mit AI ein Python Script zu generieren, welches einen den PAP anzeigt, indem man es mit dem Source Code füttert. Dazu habe ich noch den Source Code noch auf den Selbstgeschriebenen Teil reduziert, um nur relevantes mitzuteilen und die maximal zugelassene Informationsmenge anzupassen.

Erst durch wiederholtes Fragen, war es Möglich ein detailliertes Python Skript zu bekommen, welches den PAP anzeigt. Der Vorteil ist dabei noch, dass man es im nach hinein selber bearbeiten kann (siehe Python Script im Anhang und auf Git):Dazu muss man Python und Graphviz auf seinen PC installieren und im Terminal sich für Python noch das Graphviz Paket holen:

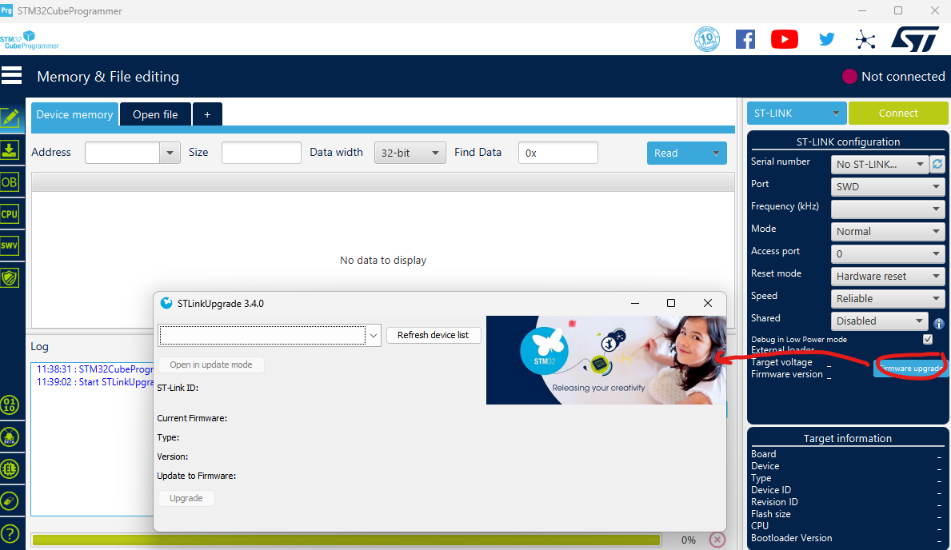


Nachdem man die .py datei gespeichert hat, muss man das Python Skript noch vom Terminal ausführen, um die .png Dateien zu erzeugen:



# Programmierung

Zunächst wollte ich sicherstellen, dass mein Nucleo-Board einwandfrei funktioniert. Dazu habe ich das Board an den PC angeschlossen und versucht, mit dem Programm STM32CubeProgrammer eine Verbindung herzustellen.

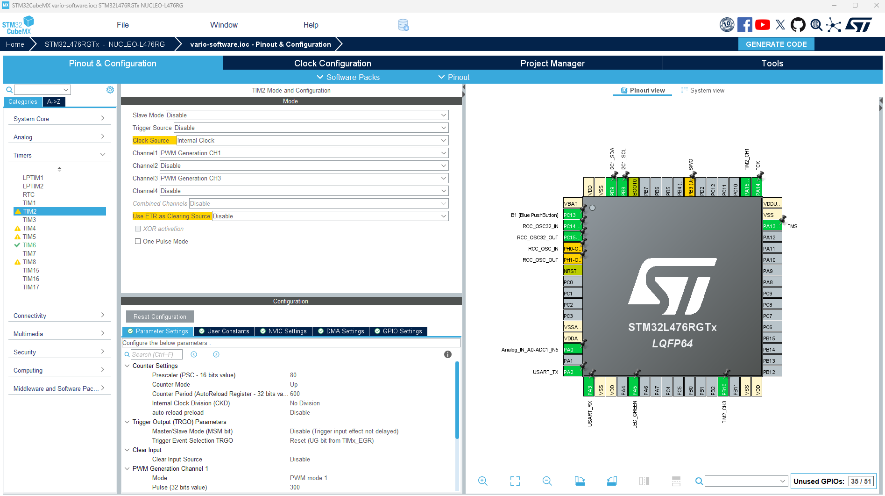
Leider schlug dies wiederholt fehl, da die auf dem Nucleo-Board integrierte ST-Link-Debugger-Software veraltet war. Glücklicherweise bietet der CubeProgrammer eine integrierte ST-Link-Utility, mit der ein entsprechendes Software-Update durchgeführt werden kann.

Das Update verlief jedoch nicht reibungslos, da die Verbindung während des Upgrade-Prozesses immer wieder abbrach. Nach einiger Recherche stellte sich heraus, dass das Problem höchstwahrscheinlich am verwendeten USB-Kabel lag. Nachdem ich ein hochwertiges, gut geschirmtes USB-Kabel ausprobiert hatte, konnte ich das Problem schließlich lösen.

Die Debugging-Möglichkeiten des Nucleo-Boards sind deutlich umfangreicher als bei Arduino. Dennoch wollte ich das gewohnte printf() nutzen, wofür in der IDE einige Einstellungen erforderlich sind. Dazu habe ich diese Anleitung gefunden:

<https://www.embedded-communication.com/en/misc/printf-with-st-link/>

## Cube MX

Die Pinbelegung wurde mithilfe von dem ST eigenen Tool Cube-MX gemacht, dort ist es sehr einfach und schön grafisch dargestellt welche Funktion ein bestimmter Pin belegen kann. Die jeweilige Funktion einem Pin zuweisen, geht mit einem Mausklick. So kann man sich sicher sein, dass keine Fehler bei der Pinbelegung gemacht werden, somit keine Überraschungen später bei der Programmierung.  
Dabei wurde darauf geachtet, die „Arduino“-Pins zu verwenden, d.h. jene, die als Buchse auf dem Nucleo-Board herausgeführt sind.

## Code

Als Entwicklungsumgebung nutzen wir das hauseigene Tool STM32CubeIDE von ST.

Der Code wurde so strukturiert, dass alle relevanten Einstellungen in einer separaten Settings.h-Datei gesammelt werden. Dadurch bleiben die Konfigurationen übersichtlich und können später im Programm einfach verwendet werden.

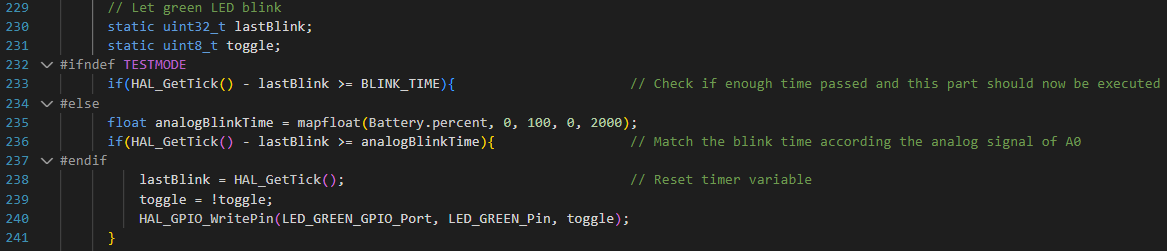
Um zu verhindern, dass einzelne Funktionen durch delay()-Aufrufe blockiert werden – oder umgekehrt selbst durch andere Delays verzögert werden – wurde bewusst auf delay() verzichtet. Stattdessen wird die aktuelle Systemzeit abgefragt, und jede Funktion wird erst dann ausgeführt, wenn die geplante Verzögerungszeit seit der letzten Ausführung verstrichen ist. Falls die Systemzeit den geplanten Zeitpunkt noch nicht erreicht hat, wird der entsprechende Codeabschnitt einfach übersprungen.

(Dies erinnert entfernt an die Funktionsweise eines Betriebssystems, ist aber eine sehr einfache und leicht umsetzbare Lösung.)

### LED blinken

Sehr einfach, da nur ein Wert auf ein GPIO geschrieben werden muss und der Wert wird jedes mal invertiert.

Später habe ich noch eine Funktion hinzugefügt, die die Frequenz des Blinkens über die analoge Spannung steuert.



### Analoge Spannung einlesen

Da ich selbst ein begeisterter Elektronikentwickler bin und in meiner Freizeit bereits viele kleine Projekte umgesetzt habe, konnte ich diese Teilaufgabe aus einem früheren Projekt übernehmen.

Angepasst werden mussten insbesondere das ADC-Element sowie die entsprechenden Parameter, die in der Settings.h-Datei ausgelagert wurden.

Da das eigentliche Akkumodul keinen direkten Anschluss für das Nucleo-Board besitzt, habe ich zum Testen ein Potentiometer angeschlossen.

Um die eingelesene Spannung in einen Prozentwert umzurechnen, für eine benutzerfreundliche Lesbarkeit, wurde ChatGPT zur Unterstützung herangezogen.

### I2C Luftdrucksensor

Zunächst habe ich nach einem I2C-Scanner gegoogelt und diesen ausgeführt. Der Scanner erkannte sofort ein Gerät bei der Adresse 0x76, was bestätigte, dass die grundlegende Kommunikation funktioniert und der Sensor antwortet.

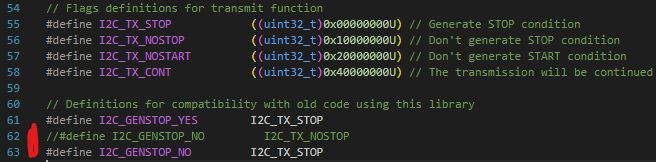
Anschließend suchte ich im Internet nach einer passenden Library für den Drucksensor und wurde schnell fündig:  
https://github.com/LonelyWolf/stm32/tree/master/bmp280

Allerdings traten beim Kompilieren zahlreiche Fehler auf, die zunächst behoben werden mussten. Zudem erforderte die Library eine separate I2C-Bibliothek aus dem gleichen Repository. Diese wiederum war von mehreren weiteren Bibliotheken desselben Entwicklers abhängig, was den Einrichtungsprozess komplizierter machte.

Beim genaueren Untersuchen des Sourcecodes fand ich eine I2C\_Init()-Funktion, die viele dieser Abhängigkeiten benötigte. Da der I2C-Bus bereits in unserer main.c initialisiert wird, konnte ich diese Funktion einfach entfernen.

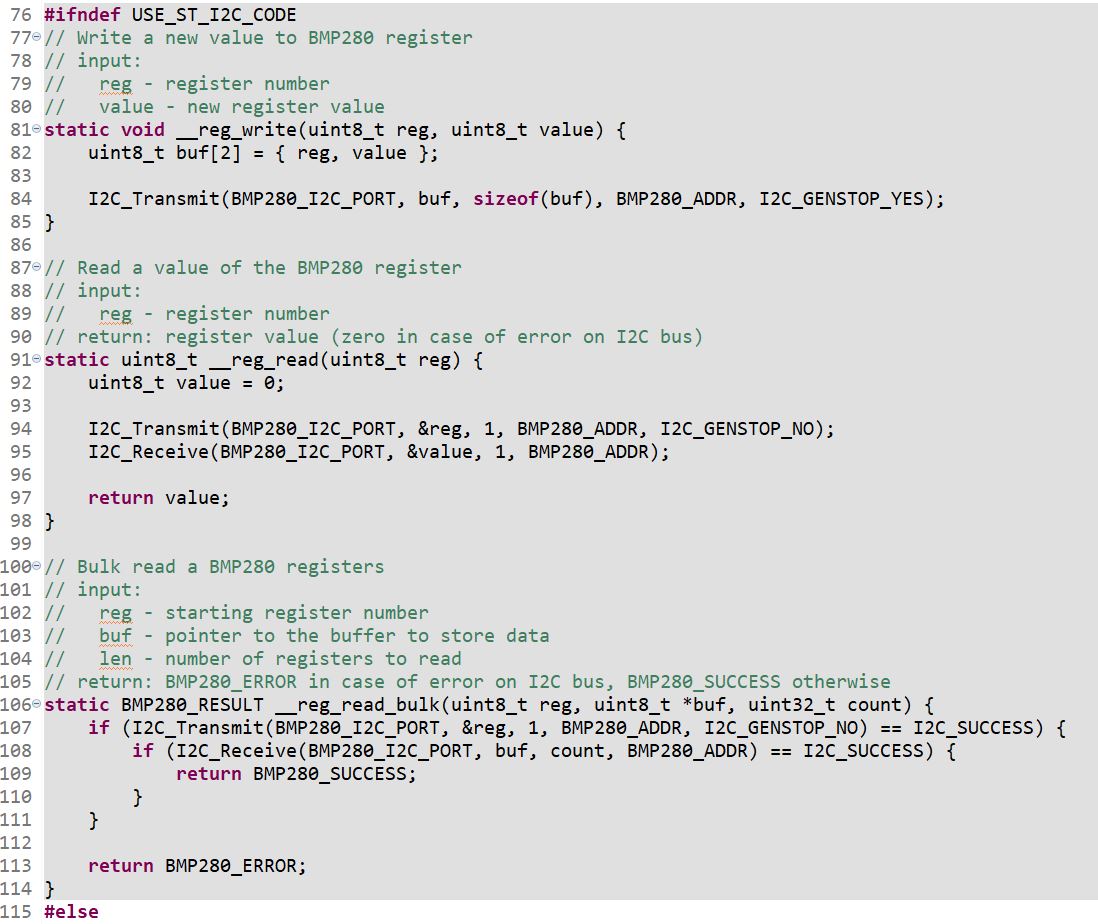
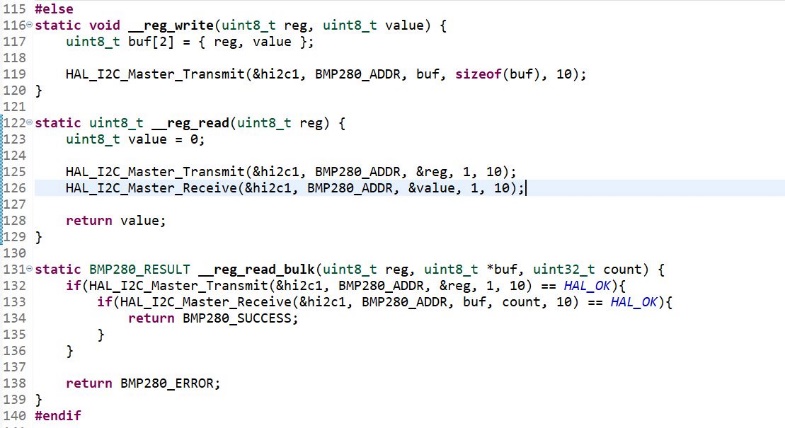
Nachdem ich schließlich die korrekte I2C-Adresse ausgewählt hatte, ließ sich der Sensor erfolgreich ansprechen.

Später, als zusätzlich das Display an den I2C-Bus angeschlossen wurde, trat ein Fehler auf. Während sowohl das Display als auch der I2C-Sensor einzeln problemlos funktionierten, konnten sie zusammen nicht angesprochen werden.

Also habe ich den Code nach möglichen Fehlerquellen durchsucht. Dabei entdeckte ich in bmp280.h, dass ein Flag übergeben wurde, welches die Stoppbedingungdes I2C-Protokolls unterdrückt.  
Um das Problem zu lösen, habe ich dieses Flag in der I2C.h-Datei per #define angepasst:

(nicht auf die schönste Art, aber so war am wenigsten Änderung im fremden Code nötig, trotzdem betrifft die Änderung alle auftretenden Stellen)

Leider brachte dieser Fix keine Verbesserung, also habe ich die Ursache noch einmal genauer untersucht. Nach etwas Überlegen wurde mir klar, dass das Problem an der proprietären Implementierung des I2C-Protokolls liegen musste.

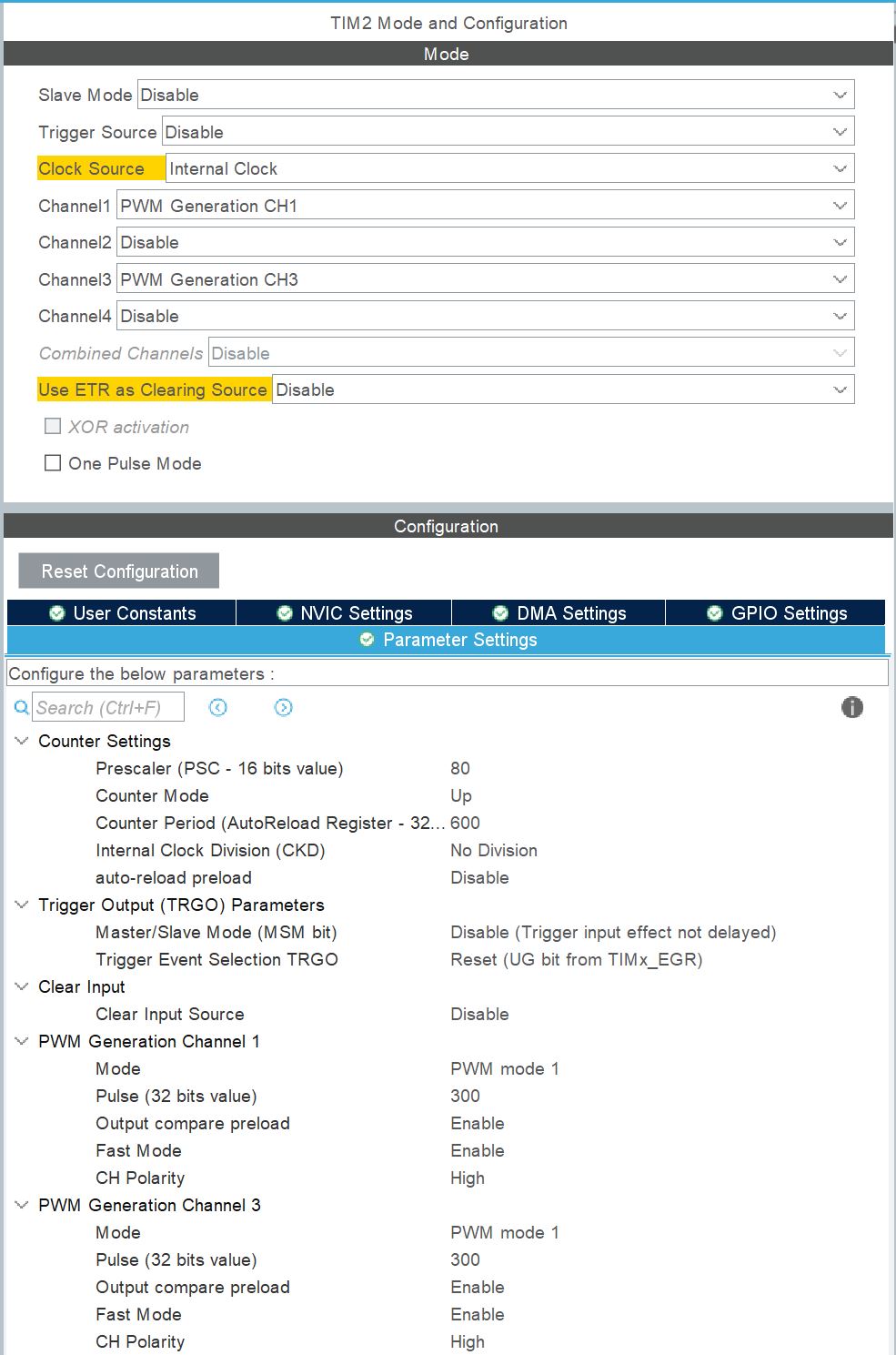
Daher habe ich diese über ein #define deaktiviert und stattdessen die I2C-Implementierung aus der STM32 HAL verwendet.

Daraufhin hat die Kommunikation beider Komponenten funktioniert.

### Buzzer

Beim **Vario** verändert sich die **Tonfrequenz** mit zunehmender vertikaler Geschwindigkeit – je schneller die Steig- oder Sinkrate, desto höher bzw. tiefer der Ton. Zusätzlich wird ein **niederfrequentes Signal** aufmoduliert, das den eigentlichen Ton **1 bis 5 Mal pro Sekunde „piepen“** lässt.

### **Timer-Konfiguration in CubeMX**

****In **CubeMX** wurde bereits **Timer 2** aktiviert und die **PWM-Generierung für Channel 3** konfiguriert, um den entsprechenden **GPIO-Pin** anzusteuern. Damit der Timer mit einer **Update-Rate von 1 MHz** läuft, muss der **80 MHz-Takt** mit einem **Prescaler von 80** heruntergeteilt werden.  
Die für die PWM-Generierung erforderlichen Werte wie **Auto-Reload-Wert (ARR)** und **Output-Compare-Wert (CCR1)** werden zur **Laufzeit im Code angepasst.**

Zusätzlich wurde **Timer 6** aktiviert und seine **Interrupt-Funktion** eingeschaltet. Da hier eine **Update-Frequenz von 40 kHz** benötigt wird, wurde ein **Prescaler von 2000** gewählt. Auch bei **Timer 6** werden **ARR und CCR1** zur Laufzeit gesetzt.

### **Implementierung der Tonsteuerung**

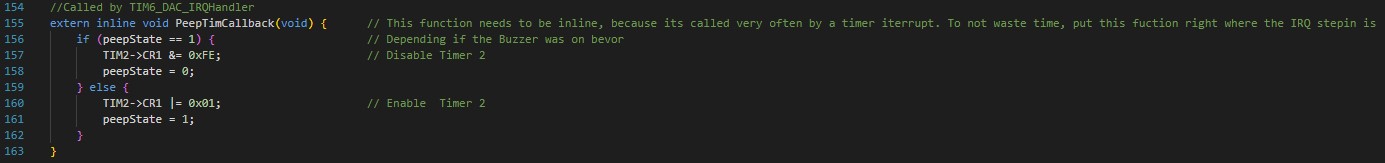
Die Logik wurde in zwei zentrale Funktionen unterteilt:

1. **CalculateTone()**
   * Entscheidet, ob ein **Steig- oder Sinkton** ausgegeben wird.
   * Berechnet die **Piep-Frequenz** sowie die langsamere **aufmodulierte Frequenz** anhand der **vertikalen Geschwindigkeit.**
   * Ein Bild, das Text, Screenshot, Software enthält.

     KI-generierte Inhalte können fehlerhaft sein.Das Ergebnis wird an die Funktion ApplyF() weitergegeben.
2. **ApplyF()**
   * Setzt die **Tonfrequenz**, die **Anzahl der Pieps pro Sekunde** (BeepsPerSecond) und die **Lautstärke**.
   * Wandelt die Tonfrequenz in eine **Periodendauer (in µs)** um.
   * Bestimmt die **Pulsdauer des PWM-Signals** basierend auf der Lautstärke.
   * Da **Timer 2 mit genau 1 MHz taktet**, kann die berechnete Periodendauer direkt ins **Auto-Reload-Register (ARR)** geschrieben werden.
   * Die Pulsdauer wird über das **Capture-Compare-Register (CCR1)** gesetzt.

### **BeepsPerSecond-Steuerung mit Timer 6**

**Timer 6** regelt die **Piep-Frequenz** (BeepsPerSecond) und schaltet bei jedem **Überlauf Timer 2** entweder **ein oder aus**. Die entsprechende **Frequenz/Periodendauer** wird ebenfalls über **ARR** festgelegt.

Der **Timer-6-Überlauf** ruft die **Interrupt-Routine** auf, in der das **Ein- und Ausschalten von Timer 2** implementiert ist. Ein wichtiger Punkt ist, dass eine **neue Frequenz erst dann auf den Buzzer geschaltet wird**, wenn sich dieser gerade **im "Aus"-Zustand** der aktuellen PWM-Periode befindet. Dadurch werden unerwünschte Zwischenfrequenzen vermieden.

### Display

Das Display besitzt SDA und SCL Anschluss. Deshalb erst mal nach “STM32 with lcd display via I2C” googeln. Das erste Video(https://www.youtube.com/watch?v=czcEovgO4Gk) gibt einen super Überblick zur Installation und den Möglichkeiten mit den Display. Das Video verweist auf diese Git Libary: <https://github.com/CircuitGatorHQ/stm32_hal_i2c_lcd_display_library.>

Nachdem man source und header Dateien runter geladen hat und im stm32 Projekt unter Core/Src und Core/Inc hinzugefügt hat, muss man nur noch die Header Datei anpassen in dem in Zeile 4

**#include** "stm32f4xx\_hal.h"

zu

**#include** "stm32l4xx\_hal.h"

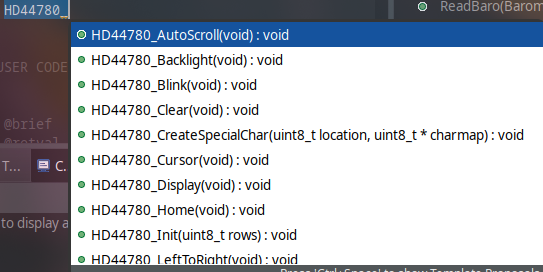
ändert, da wir ein Nucleo L4 board benutzen und in Zeile 58 die Adresse zu

**#define** **DEVICE\_ADDR** (0x27 << 1)

anpasst, um die vorgefertigten Beispiele, welche in der README.md Datei zu finden sind nutzen zu können.

Nachdem der SDA, SCL Konflikt gelöst ist kann man einfach mit den Funktionen InitLcd und UpdateLcd das Display, wie im Beispiel vollschreiben, wobei man den tmpLcd struct überwiesen bekommt, um die Daten auf dem Display dynamisch zu ändern.

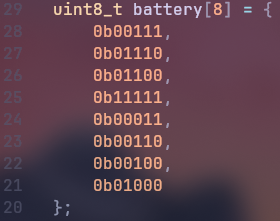
Um Platz zu sparen und Spaß zu haben kann man noch selbstgemachte Symbole hinzufügen. Mit der Vermutung, dass man ein Symbol selber erstellen kann, da man wahrscheinlich jeden Pixel ansteuern kann, findet man durch Autokorrektur heraus:



, dass es einen CreateSpecialChar Befehl gibt, welcher eine Charmap erfordert und den in einer Location(z.B 0) abspeichert. Dadurch kann mittels

HD44780\_PrintSpecialChar(0);

das Symbol dann auf den LCD drucken. Wenn man online nach einem Beispiel für eine Charmap schaut, merkt man wie intuitive es ist jedes Pixel anzusteuern. Jetzt kommt der lustige Teil, jeden Pixel so oft ändern, bauen, debugen bis man mit dem Symbol zu Frieden ist:



Der Pfeil ist natürlich ein bisschen leichter als ein Blitz für ein Batterie Symbol. Das fertige Display kann man sich unter Bilder anschauen zum Schluss.

# Versionsverwaltung via Github

Um auch nach dem Studium weiterhin an diesem Projekt arbeiten zu können und dabei das „eigentliche“ Git zu nutzen, haben wir uns entschieden, das Repository nicht mit GitLab, sondern mit GitHub zu hosten.

Nach der Einrichtung des Repositories haben wir es beide auf unseren Rechnern geklont. Anfangs wollten wir uns noch absprechen, wer wann am Code arbeitet, doch letztendlich war das nicht notwendig. Trotzdem gab es keinerlei Merge-Konflikte, und die Zusammenarbeit mit Git hat reibungslos funktioniert.

Im Laufe der Zeit haben wir eine .gitignore-Datei hinzugefügt, um das unnötige Hochladen von Build-Daten zu verhindern.

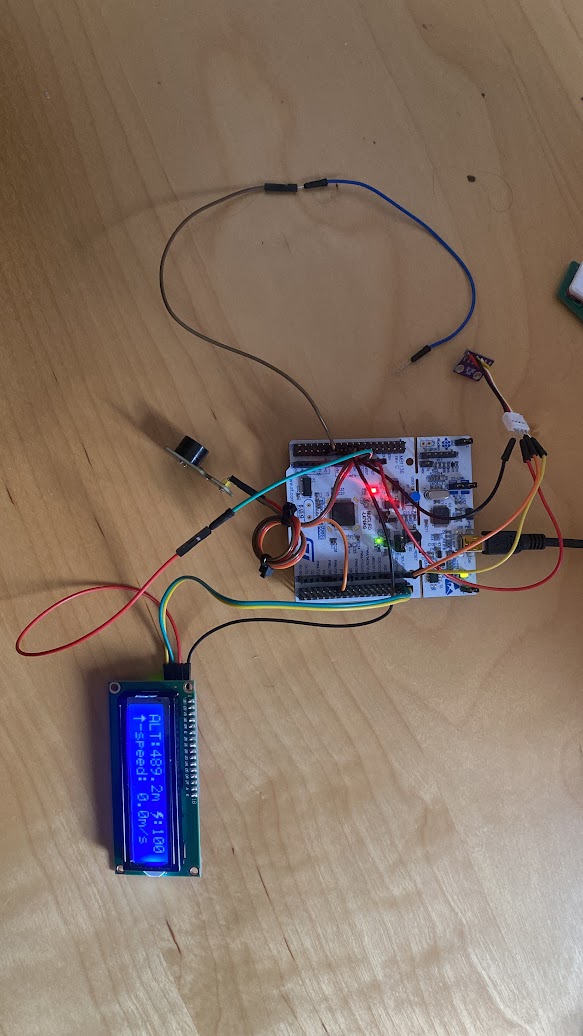
Damit jeder stets über die neuesten Änderungen informiert ist, haben wir zu jedem Commit eine aussagekräftige Nachricht hinterlassen. Diese wird in Git direkt neben den geänderten Dateien angezeigt. Falls eine Änderung zu umfangreich für eine kurze Commit-Nachricht war, haben wir zusätzliche Informationen in die README-Datei des jeweiligen Ordners geschrieben.

Dort sind unter anderem wichtige Links vermerkt, z. B. zur printf()-Anleitung oder der Pfad zu meinem lokalen Projektverzeichnis, um dieses in der Befehlszeile schneller aufzurufen.

# Bilder

Auf dem Bild ist der Aufbau zu sehen, wie er im (fast) Schaltplan vorgeben ist.   
Zugegeben sieht es etwas unübersichtlich aus, was den langen Jumper-Leitungen geschuldet ist, aber es Funktioniert.

Die einzige Anpassung gegenüber dem Schaltplan ist, am analog Pin A0 ist anstatt der Batterie ein Potentiometer angesteckt, um unterschiedliche Ladestände der Batterie zu simulieren.



# Lessons learned

Leonard:

- STM32 Projekt erstellen, konfigurieren + Pinout

- programmieren eines LCD Displays mit Charmaps

- Erstellen einer visuellen ansprechenden Verschaltung mittels Cirkit Design

- generieren von PAP mittels AI und Python

- Grundlagen Handhabung von Python mit praktischer Anwendung

- Festigung meiner persönlichen Lernstrategie: Bei Problem: selber Versuchen(eigene Idee) -> mit gegeben Informationen arbeiten, wie Autokorrektur, Fehlermeldung, Dokumentationen -> Frage in Foren suchen -> ChatGPT.

Ausnahme war AI Autogenerierung von PAP. Letzten Endes hätte man so ein simples PAP auch schneller selber hinbekommen, aber es war eine bewusste Entscheidung im Vorhinein, um zukünftige Arbeitsabläufe auch zu beschleunigen und diente als gute Vermittlung von Python Aspekten.

Bastian:

Dies war mein erstes Projekt, bei dem ich GitHub verwendet habe, und ich war positiv überrascht, wie einfach und unkompliziert die Zusammenarbeit damit funktioniert.

Die einzige Ausnahme war dieses Word-Dokument, da es weder ein klassisches Code-File noch eine Binärdatei ist. Dadurch traten häufiger Merge-Probleme auf, und Zwischenstände gingen verloren oder mussten erneut erstellt werden.

Für solche Dokumente wäre ein geteiltes Google Docs deutlich besser geeignet gewesen.

# Anhang

ChatGPT Erzeugung von Python Skript. Es wurde der selbstgeschriebene Teil vom Source Code ChatGPT davor geschickt.

