27.03.2025

Vario

Microcomputertechnik PStA von Bastian Herold und Leonard Rother

Inhaltsverzeichnis

[1. Vorwort 2](#_Toc193980256)

[2. Aufgabenstellung 2](#_Toc193980257)

[3. Informationsbeschaffung 2](#_Toc193980258)

[4. Schaltplan 3](#_Toc193980259)

[5. Programmablaufplan 3](#_Toc193980260)

[6. Programmierung 4](#_Toc193980261)

[6.1. Cube MX 4](#_Toc193980262)

[6.2. Code 5](#_Toc193980263)

[6.2.1. LED blinken 5](#_Toc193980264)

[6.2.2. Analoge Spannung einlesen 5](#_Toc193980265)

[6.2.3. I2C Luftdrucksensor 6](#_Toc193980266)

[6.2.4. Buzzer 7](#_Toc193980267)

[6.2.5. Display 9](#_Toc193980268)

[7. Versionsverwaltung via Github 9](#_Toc193980269)

[8. Bilder 10](#_Toc193980270)

[9. Lessions learned 10](#_Toc193980271)

[10. x 10](#_Toc193980272)

# 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 überhaupt mit dem Projekt zu starten war es wichtig ersteinmal die relevanten Informationen zu den genutzten Komponenten zu sammeln.  
Dazu im Internet einfach die Bezeichnung der Komponente + „datasheet“ eingegeben und eine der ersten Ergebnisse waren ein Treffer.

Verwendete Komponenten:

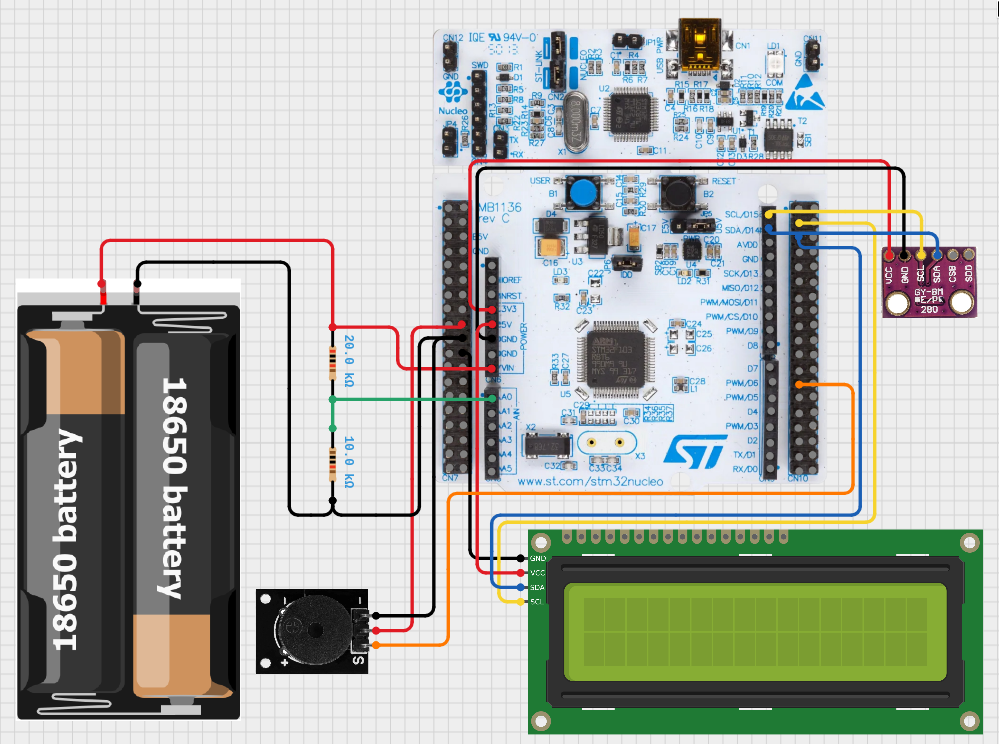
* Microcontroller: STM32L476 Nucleo – NUCLEO-L476RG
* Display: I2C 1602 LCD  
  2 Zeilen mit je 16 Zeichen
* Buzzer (Standard Aruduino)
* Barometer – Aus letztem Semester E-Praxis: Bosch BMP280, i2c  
  Hier zwar das Datenblatt von Adafuit und deren Sensormodul, allerings finde ich dieses viel besser und genauer beschrieben als das von Bosch selbst.

# Schaltplan

Nach dem Lesen der Aufgabenstellung habe ich angenommen, dass keine technische Schaltung, sondern eine visuelle Verknüpfung der Bauteile erwünsch 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 verwießen 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 weiters Forum hat noch Fritzing empfohen, 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:



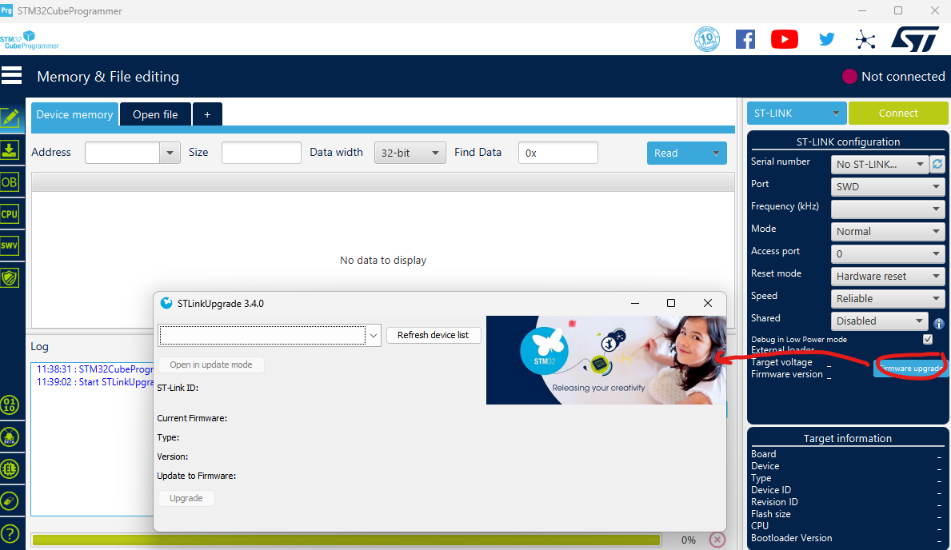
# Programmablaufplan

TODO: Leonard

# Programmierung

Zuerst wollte ich sicherstellen, dass mein Nucleo-Board auch funktionert. Dazu das Board an den PC angesteckt und mit dem Programm „STM32CubeProgrammer“ probiert eine Verbindung aufzubauen.  
Dies ist leider immer wieder fehlgeschlagen aufgrund eines veralteten Softwarestands des inkludierten ST-Link Debugger auf dem Nucleo-Board.   
In dem CubeProgrammer ist auch das ST-Link-Utility integriert, wo gleich ein solches Softwareupdate gemacht werden kann.

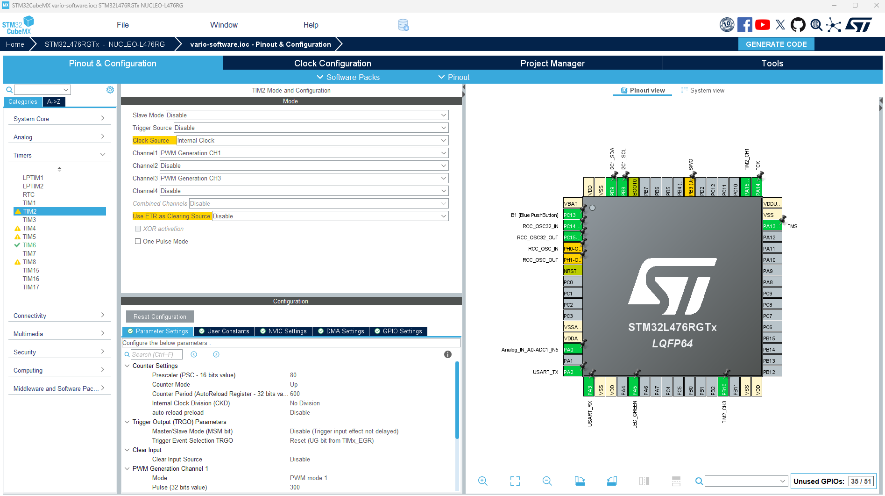
Dies hat allerdings nicht so reibungslos funktionert, da die Kommunikation beim starten des upgrades immer wieder abgebrochen ist. Nach etwas Recherche war klar, dass es warscheinlich am verwendeten Kabel liegen muss. Ich habe noch ein anderes gutes geschirmtes USB Kabel gefunden, welche das Problem löste.



Die Debugmöglichkeiten sind sehr viel umfangreicher und weitreichender als bei Arduino. Um aber trotzdem das bekannte „printf()“ benutzen zu können müssen in der IDE ein paar Einstellungen vorgenommen werden. 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 von ST eigene Tool STM32CubeIDE.

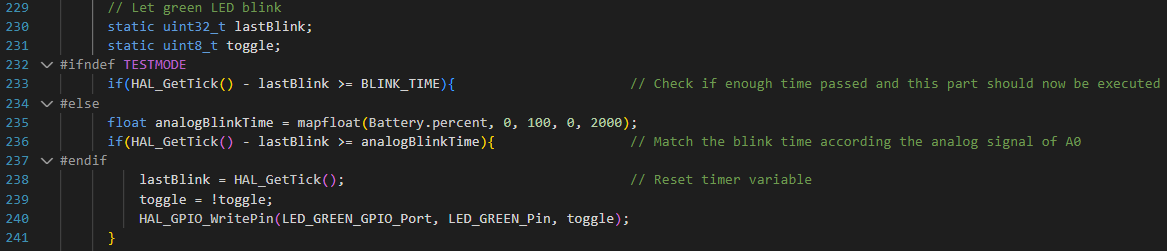
Der Code wurde so gegliedern, dass es eine separate Settings.h gibt, in die alle Einstellungen kommen, die später im Programm genutzt werden.

Um die anderen Funktionen nicht durch ein Delay zu verzögern, oder anders herum sich nicht von anderen Delays verzögern lassen, konnte kein Delay verwendet werden.  
Deswegen wurde die Methode gewählt, die aktuelle Systemzeit abzufragen und die jeweilige Funktion erst dann ausführen, wenn die Systemzeit um die gewünschte „Delay“ Zeit vorangeschritten ist. Ist die Systemzeit noch bevor der nächsten geplaten Ausführung, wird dieser Codeteil einfach übersprungen.   
(Dies ähnelt ganz fern einem Betriebssystem, ist allerdings sehr sehr einfach zu implementieren.)

### 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 begeisteter elektronik entwickler bin und in meiner Freizeit viele kleine Projekte gemacht habe, konnte ich diese Teilaufgabe aus einem alten Projekt von mir kopieren.

Anzupassen war das ADC Element, sowie die Parameter, welche in die Setting.h ausgelagert wurden.

Da das eigentliche Akkumodul keinen direkten Anschluss für das Nucleoboard hat, habe ich zum Testen ein Potetniometer angesteckt.

|  |
| --- |
| Innerhalb main -> Endlosschleife: |
| Ausgelagerte Funktion -> ReadVoltage: |
| Aufwändigere Funktion, um die Batteriespannung in einen Prozentwert umzurechenen |
| Um die Spannungswerte der Alkalibatterie zu bekommen, habe ich ChatGPT befragt: |

### I2C Luftdrucksensor

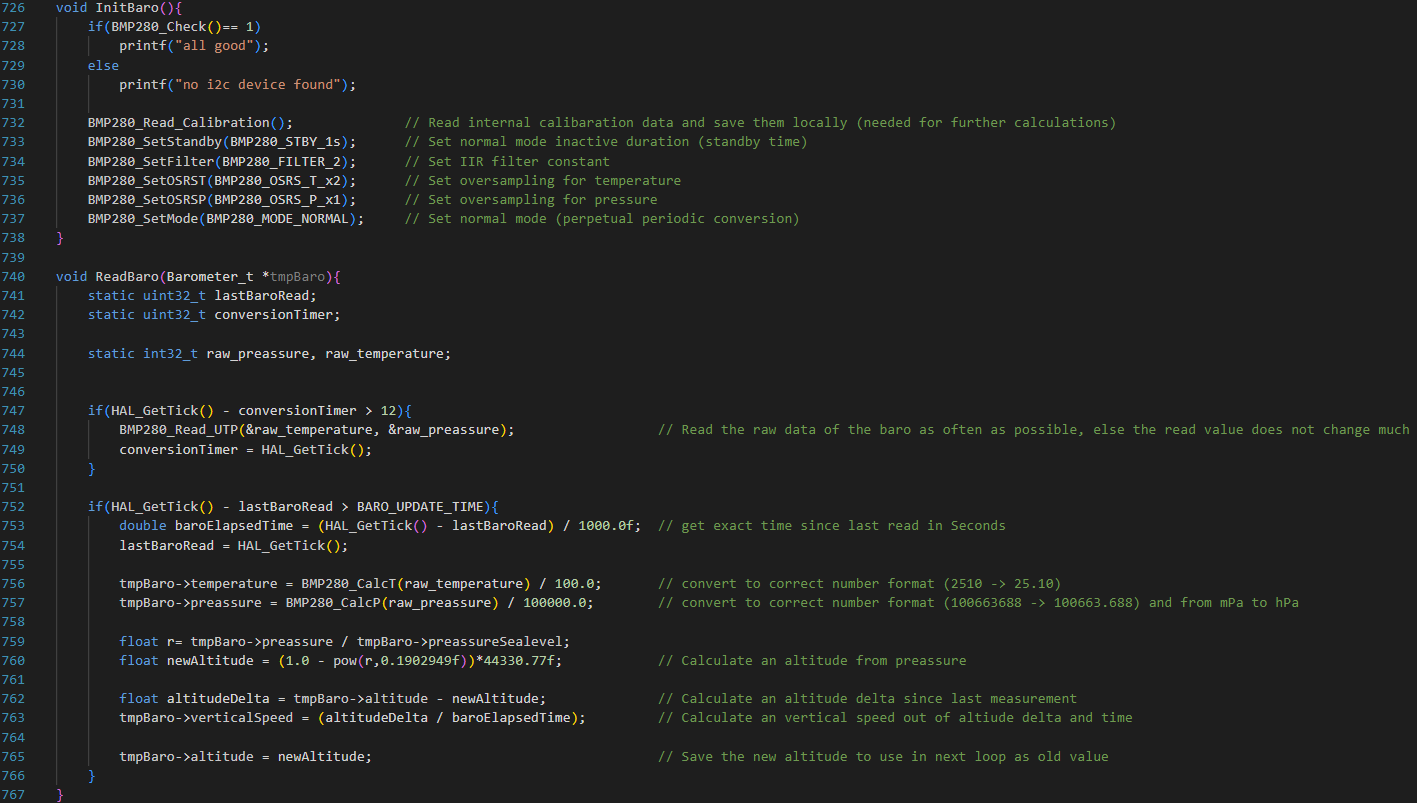
Zuerst nach einem I2C Scanner gegoogelt und diesen laufen lassen.   
Dieser hat auch sofot bei 0x76 einen Treffer gelandet. Somit wusste ich, dass die grundsätzliche Komminikation funktioniert und der Sensor antwortet.

Hier habe ich im Internet nach einer passenden Library gesucht und bin auch sehr schnell fündig geworden:

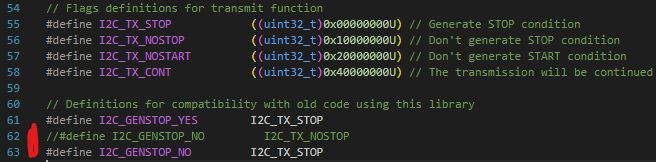
Allerdings gab es viele Compiler-errors, welche es erstmal zu lösen gab.   
Ebenfalls musste die i2c Library aus dem gleichen Git-Repository heruntergeladen werden.

Diese war allerdings wieder von sehr vielen anderen Librarys von „Ihm“ abhängig, weshalb ich mir den Sourcecode doch genauer angeschaut habe.   
Dort fand ich eine I2C\_Init Funktion, welche diese vielen Abhängigkeiten benötigt. Diese konnte ich einfach löschen, da der i2c Bus bereits in unserer main.c initialisiert wird.

Noch die richtige I2C Addresse ausgewählt und der Sensor konnte angesprochen werden.

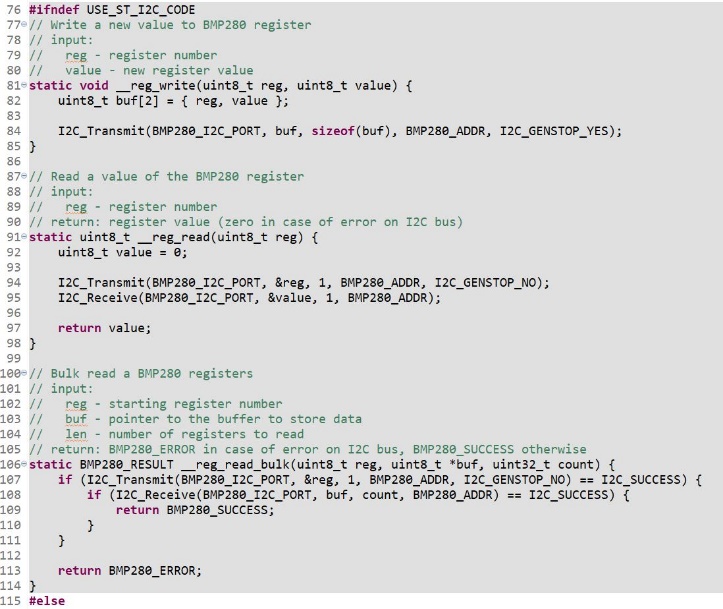
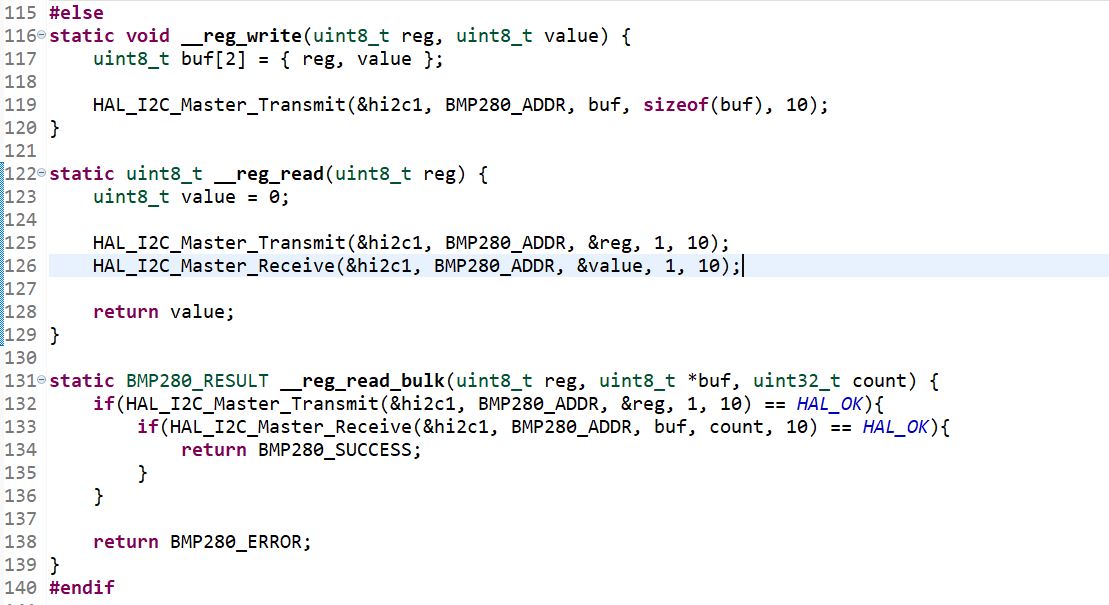
Die InitBaro() Funktion wird einmal zum Programmstart aufgerufen, die ReadBaro() zyklisch wieder in der main-loop.

Später als auch noch das Display mit auf den I2C Bus angesteckt wurde, ist ein Fehler aufgetreten. Das Display einzeln konnte angesprochen werden, der I2C Sensor ebenfalls, nur beides zusammen nicht.

Darum nochmal im Code auf die Suche begeben nach möglichen Fehlerstellen. Dort hab ich in bmp280.h gefunden, dass ein Flag übergeben wird, welches die Stoppbedingung des I2C Protokolls unterdrückt.   
Dieses Flag habe ich im I2C.h per #define verändert:

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

Leider hat dieser Fix keine verbesserung gezeigt, darum habe ich nochmal genauer nachgeschaut.  
Nach etwas Nachdenken, war mir klar, es muss an der proprietären Implementation des I2C Protokolls liegen. Diese habe ich über ein Define deaktiviert und durch die Implementierung aus der stm32hal ersetzt:

Daraufhin hat die Kommunikation beider Komponenten funktioniert.

### Buzzer

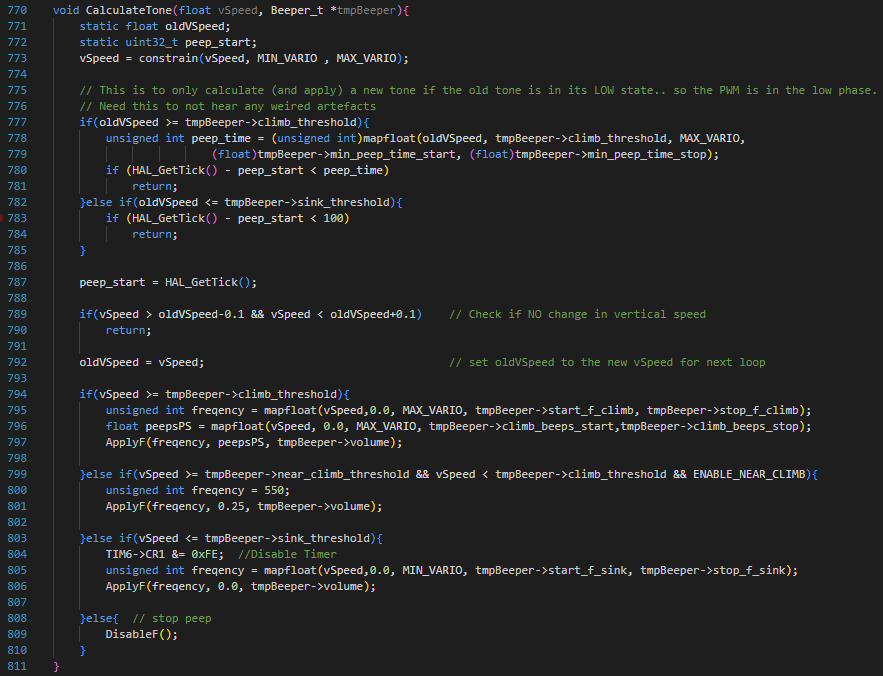
Beim Vario ist es so, dass sich mit mehr vertikaler Geschwindigkeit der Ton auch in seiner Frequenz steigt. Gleichzeitig ist noch ein niederfeqentes signal aufmoduliert, welches den eigentlichen Ton 1 bis 5 mal pro sekunde „piepen“ lässt.

Hierfür wurde im Cube MX bereits der Timer 2 aktiviert und die PWM Generation für Channel 3, also den passenen GPIO aktiviert.

Um später den Timer mit einer updaterate von 1MHz benutzen muss der 80MHz takt mit einem Prescaler von 80 heruntergeteilt werden.   
Die restlichen Werte Zur PWM Generation wie auto-reload Wert und output-compare Wert, werden im code zur laufzeit verändert.

Ebenfalls wurde im Cube MX der Timer 6 aktiviert und die Interrupt-Funktion eingeschalten.  
Hier wird eine Updatefrequenz von 40kHz benötigt, darum in Prescaler von 2000.  
Wie auch schon bei TIM2 werden auch hier AR und OC zur der Laufzeit gesetzt.

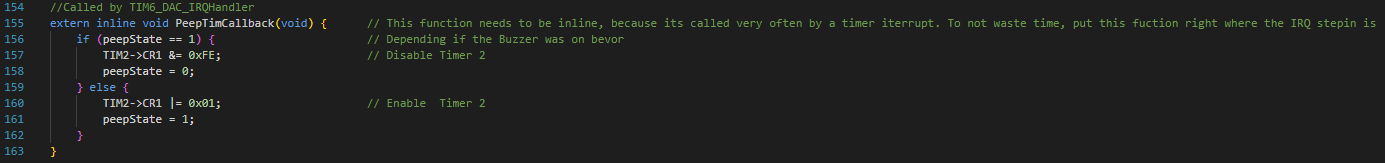
Dies habe ich in in zwei wesentliche Funktionen unterteile, einmal eine CalculateTone und eine ApplyF. CalculateTone ist dabei dafür zuständig, überhaupt zu entscheiden ob ein Sink- oder Steigton ausgegeben wird. Dann erfolgt die Berechnung der peep Frequenz sowie der langsameren „aufmodulierten“ Frequenz anhand verticalen Geschwindigkeit. Das Ergebnis dieser Berechnung wird an die Funktion ApplyF weitergegeben. Noch wichtig zu erwähnen ist, dass nur dann eine neue Frequenz auf den Buzzer geschalten wird, sollte dieser sich gerade im off-Zustand der aktuellen PWM Periode befinden. Dies ist nötig, damit kein komischer Zwischenton entsteht.



Die ApplyF Funktion ist nun etwas kryptischer geschrieben, allerdings einfach zu verstehen. Übergeben wird eine Ton-Frequenz, die BeepsPerSecond und die Lautstärke.

Nun wird die Ton-Frequenz in eine Periodendauer (in µs) umgerechnet und aus der Lautstärke wird die On-Zeit für das PWM Signal generiert.  
Da der Timer 2 genau mit 1MHz taktet, kann die eben errechnete Periodendauer in das AutoReloadRegister (ARR) geschrieben werden. Die Pulsdauer des PWM Signals wird über das CaptureCompareRegister1 (CCR1) festgelegt.  
Timer 6 ist für die BeepsPerSecond zuständig und schaltet bei jedem überlauf den Timer 2 entweder an oder aus. Die entsprechende Frequenz/Periodendauer wird ebenfalls über das ARR gesetzt.

Dieser Überlauf ruft die Interrupt Route des Timer 6 auf, in welcher das ein/ausschalten des Timer 2 implementiert ist.



### Display

Das Display besitzt SDA und SCL Anschluss. Deshalb erstmal nach STM32 with led display via I2C googlen. Das erste Video(https://www.youtube.com/watch?v=czcEovgO4Gk) gibt einen super Überblick zur Installation und den Möglichkeiten mit den Display. Das Video verweißt 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 Addresse zu

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

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

# Versionsverwaltung via Github

Um auch später, nach dem Studium weiterhin mit dem Projekt weitermachen zu können und auch das „eigentliche“ Git zu benutzen, haben wir uns entschieden dieses Projekt nicht mit Gitlab, sondern mit Github zu machen.

Wir haben das neue Repository eingerichtet und Beide auf unsere Rechner geclont.

Anfangs wollten wir uns noch absprechen, wer wann arbeitet, allerdings haben wir das dann doch nicht gemacht.   
Trotzdem kam es nie zu „merge“ Problemen und das Arbeiten im GIT hat sehr gut funktioniert.

Mit der Zeit kam auch noch ein „.gitignore“ File hinzu, welches das unnötige hochladen von den Built-Daten verhindert.   
Damit der jeweils Andere immer weiß was zuletzt gemacht wurde, haben wir zu jedem „commit“ eine Nachricht geschrieben. Diese taucht im GIT neben dem Filenamen auf.

Alles was zu viel für eine einfache commit Nachricht ist, kam in die README des jeweiligen Ordners.  
Hier steht z.B. der Link zur printf Anleitung / der Pfad zu meinem lokalen Projektverzeichnis (um mit der commandozeile leichter hinzukommen).

# 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 angesteck, um unterschiedliche Ladestände der Batterie zu simulieren.

HIER BILDER VOM „FERTIGEN“ VARIO EINFÜGEN BZW NOCH MACHEN

# Lessions learned