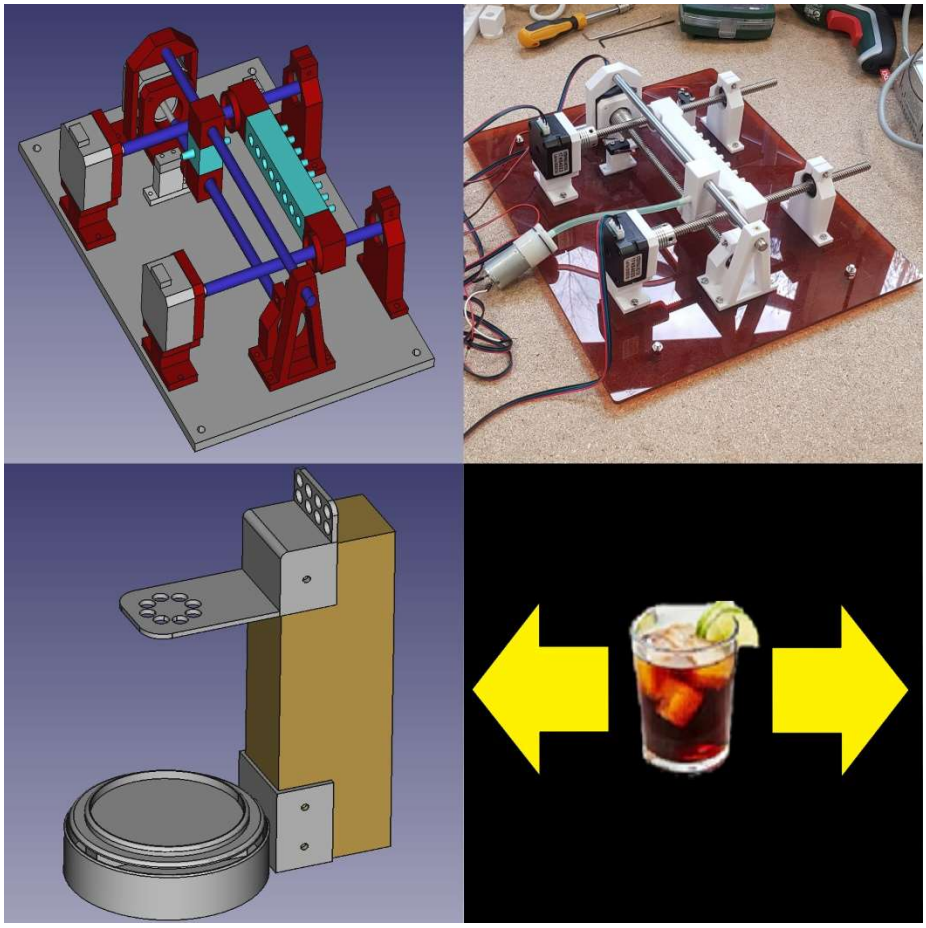
Projekt:
Automatischer Barkeeper



Von der Idee zur Umsetzung

von

Nick Fischer

Betreuungslehrer: Herr Nebe

St. Michael Gymnasium, Bad Münstereifel

Inhaltsverzeichnis

1. Projektidee 4
2. Das innovative Pumpsystem 4
3. Ziele 4
   3.1. Primäre Ziele 4
   3.2. Sekundäre Ziele 5
4. Erster Prototyp 5
   4.1. Hardware 5
      4.1.1. Grundlegendes Design 5
      4.1.2. „Universeller Verschluss“ 5
      4.1.3. Das der exakten Ausgabemenge 5
      4.1.4. Das Verhindern eines ungewollten Nachlaufes 6
      4.1.5. Das Verhindern von einem Rücklauf 6
   4.2. Software 6
      4.2.1. Softwareerstellung mit Touchdisplay 6
   4.3. Zusammenbau 7
   4.4. Erster Prototyp & Konsequenzen aus den Testergebnissen 7
5. Zweiter Prototyp 9
   5.1. Hardware 9
      5.1.1. Ein neues, optimiertes Fördersystem 9
      5.1.2. Flaschenhalter und Getränkeausgabe mit Wiegeeinheit 10
   5.2. Software 10
      5.2.1. Überlegung zu den wichtigsten Komponenten der Steuerung 10
      5.2.2. Bauen einer Webseite die den Grundbedingungen entspricht I 10
      5.2.3. Bauen einer Webseite die den Grundbedingungen entspricht II 12
      5.2.4. Hintergrundbilder 12
      5.2.5. Kommunikation vom Webserver zum ESP 13
      5.2.6. Auslagern von dem Code der Webseite in eine eigene Datei zur besseren Übersicht 13
      5.2.7. Kommunikation vom ESP zum Webserver 14
      5.2.8. Work in progress Seite 14
      5.2.9. Mixing Code 15
      5.2.10. Homing Funktion 16
      5.2.11. Docking Funktion 16
      5.2.12. Erweiterungen der Grundfunktionen 16
         5.2.12.1. Custom Seite 17
         5.2.12.2. Preset Option 17

5.2.12.3. Drink Save Funktion 17
6. Anhang 18
6.1. vollständiger Code ..... **Fehler! Textmarke nicht definiert.**

Abbildungsverzeichnis

Abbildung 1: FreeCAD-Zeichnung des ersten Prototyps (Querschnitt) 7
Abbildung 2: Mehrfachverteiler 8
Abbildung 3: FreeCAD-Zeichnung des neu entwickelten Luftverteilers 9
Abbildung 4: Problem Hintergrundbild des Buttons 11
Abbildung 5: Einbinden der Bibliotheken 12
Abbildung 6: Einbinden von ausgelagertem HTML Code 13
Abbildung 7: WIP Seite 14
Abbildung 8: Loop Code für Case 0 15
Abbildung 9: 2D Array 15
Abbildung 10: Homing Code 16
Abbildung 11: Docking Code 16
Abbildung 12: Custom Seite 17
Abbildung 13: Custom Seite Icon 17
Abbildung 14: Einstellungsseite Icon 17
Abbildung 15: Einstellungsseite 17

1. Projektidee

Ziel dieses Projektes ist es, einen automatischen Getränkemixer zu bauen, der Flüssigkeiten mit Hilfe eines innovativen Pumpsystems befördern und als besonderes Feature mit Hilfe eines beliebigen Smartphones über den Browser gesteuert werden kann.

2. Das innovative Pumpsystem

Bei regulären Pumpen muss die zu fördernde Flüssigkeit die Pumpe durchlaufen. Genau das verursacht aber vor allen Dingen bei Flüssigkeiten, die sauer, basisch oder aus anderen Gründen korrosiv sind, eine schnellere Abnutzung oder ggf. sogar Beschädigung der Pumpe. Um dies zu vermeiden, werden in diesem Kontext üblicherweise sogenannte Peristaltikpumpen eingesetzt. Hierbei wird die zu befördernde Flüssigkeit durch einen Schlauch geleitet, der von der Pumpe periodisch komprimiert wird, wodurch die Flüssigkeit innerhalb des Schlauches vorwärtsgetrieben wird, ohne dass die Pumpe in direkten Kontakt mit ihr kommt. Diese Art der Pumpe hat jedoch mehrere, entscheidende Nachteile: Einerseits ist die Fördermenge pro Zeit mit ca. 25-80ml / min sehr gering, was sie für unsere Zwecke ungeeignet macht. Des Weiteren ist der Verschleiß, der durch das Walken des Förderschlauches entsteht, erheblich und verkürzt die Lebensdauer des Schlauches drastisch. Und schließlich ist eine solche Pumpe sehr viel teurer als normale Pumpen (durchschnittlich 30€ pro Pumpe). Um diese Nachteile zu eliminieren und dennoch problematische Flüssigkeiten sicher und schnell befördern zu können, haben wir ein innovatives Pumpsystem entwickelt, das mit Luftdruck funktioniert. Dabei wird auf die Flasche mit der zu pumpenden Flüssigkeit ein luftdichter Stopfen platziert. In diesem Stopfen sind zwei Löcher gebohrt, durch die je ein Luft- und ein Ausgabeschlauch verlaufen. Dabei endet der Luftschlauch direkt am unteren Ende des Stopfens, während der Ausgabeschlauch bis zum Grund der Flasche reicht. Durch das Pumpen von Luft durch den Luftschlauch, entsteht ein Überdruck, durch den die in der Flasche befindliche Flüssigkeit verdrängt wird. Diese sucht sich den Weg mit dem geringsten Widerstand aus der Flasche heraus. Dieser Weg ist der des in die Flasche hinein reichenden Ausgabeschlauches. Somit muss die Pumpe selbst zu keiner Zeit die Flüssigkeit berühren und bleibt infolgedessen unbeschadet. Die entscheidenden Vorteile bei unserem System sind einerseits, dass auch größere Flüssigkeitsmengen sehr schnell ohne direkten Kontakt mit einer Pumpe befördert werden können. Andererseits hat unser System ein unschlagbares Preis-Leistungs-Verhältnis: Während eine Peristaltikpumpe mind. 30€ benötigt würde, kommt unser finales System mit einer einzigen 5V Aquarienluftpumpe für ca. 4€ aus.

3. Ziele

3.1. Primäre Ziele

Das erste und wichtigste Ziel ist, dass der Barkeeper automatisiert Getränke mixen und ausgeben kann. Dabei werden zwei zentrale Komponenten benötigt. Die eine ist ein funktionierendes Pumpsystem und die andere ist eine Bedienoberfläche, über die mit entsprechender Software Getränke ausgewählt und ausgegeben werden können. Dafür soll anfangs ein Touchdisplay verwendet werden können. Des Weiteren sollte das System eine präzise Dosierung haben, um immer wieder das gleiche Getränk ohne einen variierenden Geschmack erstellen zu können. Ein weiteres primäres Ziel ist, dass die Herstellung des Barkeepers niedrige Herstellungskosten aufweist. Außerdem wäre es sinnvoll, wenn das Gerät skalierbar ist, damit es auf die Konsumenten besser zugeschnitten werden kann. Dazu zählt zum Beispiel die Anzahl der gleichzeitig anzuschließenden Flüssigkeiten.

3.2. Sekundäre Ziele

Zu den sekundären Zielen zählt ein kompaktes Design, welches nicht allzu viel Platz wegnimmt. Des Weiteren wäre es sinnvoll, wenn das Gerät von einem Handy aus steuerbar wäre. Dadurch muss der Benutzer nicht direkt mit dem Gerät interagieren. Hierdurch können Beschädigungen durch unsachgemäßen Gebrauch vermieden werden. Dafür würde der Barkeeper ein eigenes Netzwerk aufbauen, bei dem der Nutzer sich anmelden muss und dann im Browser über eine entsprechende Webseite den Barkeeper steuern kann. Die Steuerung über den Browser hat den Vorteil, dass sich nicht auf ein Betriebssystem spezialisiert werden muss und jeder Nutzer somit unabhängig von seinem Gerät den Barkeeper ohne Probleme steuern kann. Als letzter Punkt soll das Erstellen eigener Getränke über die Webseite und deren Speicherung auf dem Barkeeper für den Verbraucher möglich werden.

4. Erster Prototyp

4.1. Hardware

4.1.1. Grundlegendes Design

Initial galt es sich zu überlegen, wie der Barkeeper aufgebaut sein soll. Das Gehäuse sollte über ausreichend Platz verfügen, um alle benötigten Komponenten problemlos integrieren zu können. Zudem sollte das Design des Gehäuses simpel wie auch praktisch sein. Daher wurde zunächst eine rechteckige Form gewählt, die an der Vorderseite eine Ausbuchtung hat, so dass darunter ein entsprechendes Glas passt, in welches die Flüssigkeit ausgegeben werden kann. Die Flaschen mit den verschiedenen Flüssigkeiten, die an das System angeschlossen werden, wurden außerhalb des Gehäuses platziert. Dies hat mehrere Vorteile. Einerseits können hierdurch die sich in dem Gehäuse befindlichen elektronischen Komponenten nicht durch austretende Flüssigkeiten in den Flaschen beschädigt werden. Andererseits wird dadurch für den Wechsel einer Flasche das Öffnen des Gehäuses vermieden. Zusätzlich können so die Größen der Flaschen variieren, was bei einer Anordnung der Flaschen innerhalb des Gehäuses nicht möglich wäre. Die Schläuche werden dann ins Innere geführt, wo alle weiteren Komponenten verbaut sind bzw. sukzessive auch neue Teile verbaut wurden. Die Entwurfszeichnungen wurden dabei in FreeCAD erstellt und der erste Prototyp dann aus Polycarbonatplatten gefertigt.

4.1.2. „Universeller Verschluss“

Um die Flaschen passend anschließen zu können und nicht für jede Flasche einen eigenen Verschluss entwickeln zu müssen, wurde ein Verschluss benötigt, der auf jeden Flaschenkopf so passt, dass die Flasche luftdicht verschlossen wird. Hierzu wurde ein lebensmittelechter, konischer Gummiverschluss verwendet, da dieser sich dem Durchmesser des Flaschenkopfs anpasst und demnach auf die meisten Flaschen passt und diese auch luftdicht verschließt.

4.1.3. Das der exakten Ausgabemenge

Um eine präzise Dosierung vornehmen zu können, haben wir uns entschieden mit einem Durchflussmesser zu arbeiten. Dieser kann die Flüssigkeitsmenge, welche pro Zeiteinheit hindurchfließt, messen. Es gibt jedoch verschiedene Arten von Durchflussmessern. Wir haben beschlossen einen Durchflussmesser zu benutzen, der mit dem Hall-Effekt arbeitet. Dieser funktioniert dadurch, dass durch die Flüssigkeit ein sich innerhalb des Durchflussmessers befindliches Rad zu drehen beginnt. An einer Seite des Rades befindet sich ein kleiner Magnet, welcher jedes Mal, wenn dieser wieder an derselben Stelle

ankommt, eine geringe Spannung erzeugt. Diese kann gemessen werden und damit kann dann die Menge, die durch das System fließt bestimmt werden. Es war hierbei das sinnvollste, den Durchflussmesser am Ende, kurz bevor die Flüssigkeit in das Glas fließt, anzubringen. Dies hat zwar den Nachteil, dass man genau bestimmen muss, wie viele Milliliter Flüssigkeit nach dem Stopp der Pumpen noch nachläuft. Jedoch sind die Schläuche noch kurz bevor sie ins Glas gehen, getrennt voneinander, womit wir dann acht Durchflussmesser benötigt hätten. Es also am Ende des Systems anzubringen, spart Kosten und Platz.

4.1.4. Das Verhindern eines ungewollten Nachlaufes

Ein großes Problem beim ersten Design war, dass bei der Konstruktion des Systems ein Nachlauf entstand, der noch eine beträchtliche Menge an Flüssigkeit förderte, nachdem das Flowmeter die Luftpumpe bereits abgestellt hatte. Dies war der Fall, da im System noch ein Überdruck vorhanden war. Um dieses Problem zu lösen wurden im ersten Prototyp Magnetventile verbaut, die gleichzeitig mit dem Abschalten der Luftpumpe durch die Steuersoftware aktiviert wurden und die Schlauchausgabe blockierten, was sehr gut funktionierte.

4.1.5. Das Verhindern von einem Rücklauf

Ein weiteres Problem stellte zudem ein ungewollter Rücklauf durch den Schlauchverteiler dar. Dieser entsteht dadurch, dass die Schläuche kurz bevor sie in das Glas kommen zusammengeführt werden, so dass nur ein einziges Flowmeter am finalen Ausgabeschlauch verbaut werden musste. So konnte allerdings die Flüssigkeit, die von einem Schlauch befördert wird, durch einen momentan nicht benutzten Schlauch wieder zurück in eine andere Flasche laufen. Dieses Problem wurde gelöst, indem Rückschlagventile in das Schlauchsystem eingebaut wurden, so dass eine geförderte Flüssigkeit nur durch den korrekten Ausgabeschlauch mit dem Flowmeter fließen konnte.

4.2. Software

4.2.1. Softwareerstellung mit Touchdisplay

Um einen ersten Testlauf mit dem Design machen zu können, müssen die Pumpen ein- und ausschaltbar sein. Außerdem sollte es eine Auswahl an Drinks geben, die man auswählen kann, so dass jede Pumpe angesteuert werden kann. Um ein solches Interface zu schaffen, haben wir erst einmal einen Touchscreen genutzt, mit dem man verschiedene Drinks auswählen kann. Eine ausführliche Erläuterung des Codes wird hier nicht vorgenommen, da dies zum einen den Rahmen sprengen würde und schlussendlich auf die Steuerung des Barkeepers mit einem Handy umgestiegen wurde. Der Code für die Steuerung des Barkeepers mit dem Handy wird jedoch später ausführlich erklärt.

4.3. Zusammenbau

Im ersten Schritt wurde das Gehäuse, ausgenommen der Oberseite, zusammengebaut. Dabei hat sich herausgestellt, dass die Schrauben, welche verwendet wurden, um die einzelnen Seiten zu montieren, nicht für das Material der Wände geeignet sind. Die zunächst verwendeten Schrauben führten nämlich dazu, dass sich einerseits das Material an dieser Stelle ausgedehnt hat und andererseits die Schrauben nicht mehr herausgeschraubt werden konnten, da diese abbrachen. Zudem wurde das Material während dem Schrauben so heiß, dass dieses dabei zusammengeschmolzen ist. Um nun die Druckluftpumpen befestigen zu können, wurden passend große Löcher in die linke Seite des Gehäuses gebohrt. Danach wurde die Platine zur Steuerung der Pumpen mit den Pumpen verkabelt und schließlich für die ersten Tests lose in das Gehäuse gelegt. Als nächstes wurden der Verteiler, der wie alle eigenen Bauteile selbst mit Hilfe der Software FreeCAD entworfen und dann auf unserem 3D-Drucker aus PLA gedruckt wurde, die Magnetventile und die Rückschlagventile an die obere Platte geschraubt. Außerdem wurde ein rechteckiges Loch in diese Platte geschnitten, in welche dann der Touchscreen eingelassen wurde. Dieser wurde dann mit dem Arduino-Board verbunden. Hiernach musste noch der Durchflussmesser angebracht werden. Dieser wurde in die Ausbuchtung hinein geschraubt. Schließlich mussten noch die Schläue verbunden werden. Nachdem auch das geschafft war, wurde mit den ersten Testläufen begonnen.

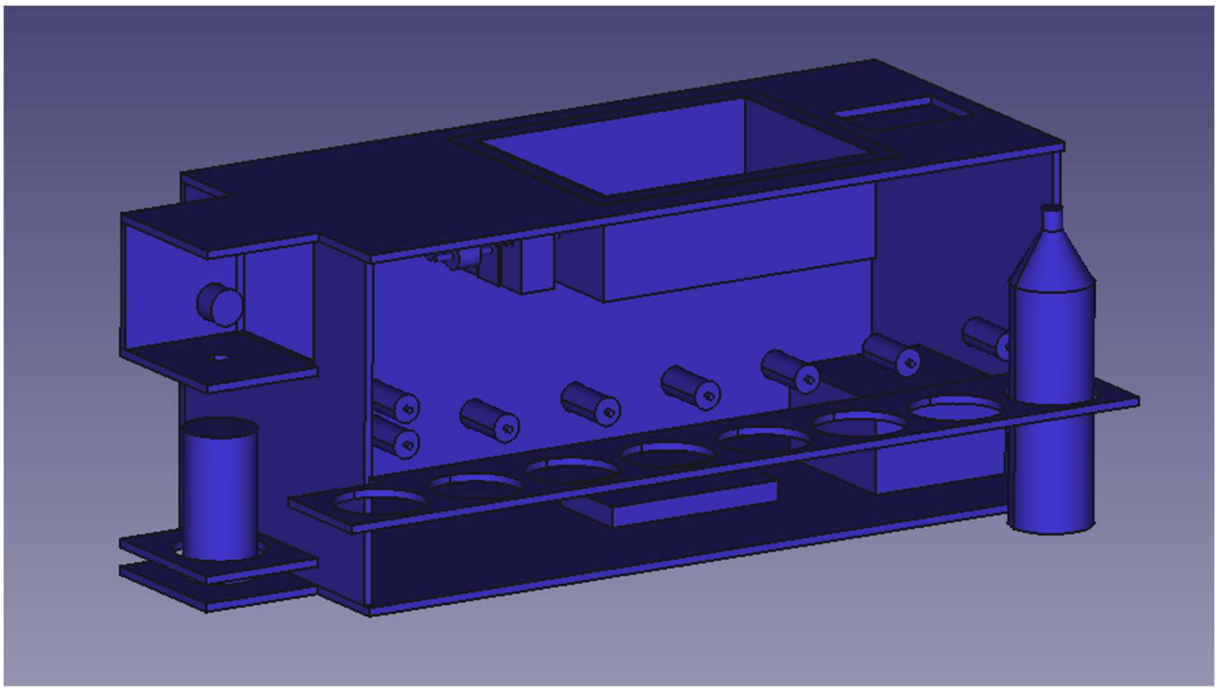


Abbildung 1: FreeCAD-Zeichnung des ersten Prototyps (Querschnitt)

4.4. Erster Prototyp & Konsequenzen aus den Testergebnissen

Zu sehen ist die Anordnung der Förderpumpen in der seitlichen Wand (ausgeblendet), die Halterung für die Flaschen, die zentrale Eisbox, dass Touchdisplay rechts in der Deckplatte, die Magnetventile, die Rückschlagventile (nicht zu sehen der Mehrfachverteiler), sowie links die Ausgabe mit Flowmeter. Nicht dargestellt sind sämtliche Schlauchverbindungen.

Wie sich herausstellte, war der aus PLA gedruckte Flüssigkeitsmehrfachverteiler leider nicht vollständig wasserdicht, da offenbar eine zu geringen Ausfüllung beim Druck gewählt wurde. Auch war der gesamte erste Prototyp am Ende sehr sperrig und unhandlich. Es gab zu viel ungenutzten Raum, so dass er sehr viel größer wurde, als er sein müsste. Zudem gab es ein gewisses Sicherheitsrisiko, da das ATX-Netzteil am Boden des Prototyps verbaut war, so dass bei einem Austritt von Flüssigkeiten innerhalb des Prototyps diese in Kontakt mit dem Netzteil kommen könnten. Der Hauptgrund, warum das erste Design aber letztendlich komplett verworfen wurde, war, dass die Kosteneffizienz – immerhin ein designiertes Hauptmerkmal der zu entwickelnden Getränkemixers – nicht mehr wirklich gegeben war: Um eine funktionierenden Beförderung zu realisieren, musste ein druckdichtes System geschaffen werden. Daraus ergab sich, dass jede Schlauchleitung über eine eigene Luftpumpe, ein eigenes Magnetventil und ein eigenen Rückschlagventil verfügen musste. Und damit war der Kostenabstand zu einer Peristaltikpumpe nicht mehr so erheblich, dass es dem innovativen Charakter entsprach, der das Ziel war. Daher wurde das erste Design komplett verworfen und mit Hilfe der gesammelten Erfahrungen ein zweiter, optimierter Prototyp geschaffen.

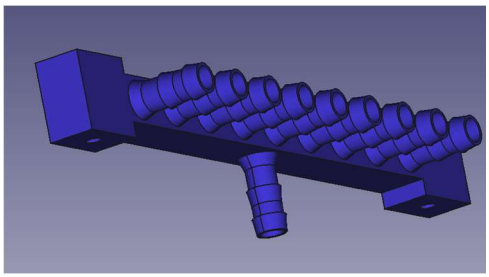


Abbildung 2: Mehrfachverteiler

5. Zweiter Prototyp

5.1. Hardware

5.1.1. Ein neues, optimiertes Fördersystem

Um das entstandene Kernproblem zu lösen, dass zu viele Einzelkomponenten in einem druckgeschlossenen System erforderlich werden, um Flüssigkeitsbeförderung, Flüssigkeitsmessung und Nach- bzw. Rücklaufverhinderung zu realisieren, wurde folgender Lösungsansatz gefunden: Der Nachlauf ergab sich aus dem Überdruck, da der Luftschlauch fest an einer dazugehörigen Luftpumpe angeschlossen war. Würde man jedoch ein Design entwickeln, bei dem der Luftschlauch jeweils von der Luftpumpe entkoppelt wird, so gäbe es keinen Nachlauf. Damit wären sofort die Magnetventile – ein wesentlicher Kostenfaktor – obsolet. Und man würde mit nur einer einzigen Luftpumpe auskommen! Ein Nachteil wäre jedoch, dass man in einem nicht durchgehend geschlossenem System die Flüssigkeitsausgabemessung nicht mehr mit einem Flowmeter realisieren kann. In der Gesamtbetrachtung ist ein solches System dem alten Ansatz aber stark überlegen, so dass ein entsprechendes, komplett neues Design entwickelt wurde: Kernelement ist hierbei ein Luftverteiler, der in einer Art CNC-Steuerung mit Hilfe von insgesamt 3 NEMA17-Schrittmotoren in zwei Achsen auf Kugelumlaufspindeln gesteuert wird. Dabei ist an einem pneumatischen Verteilerblock auf der Y-Achse ein Luftschlauch der einzig verwendeten Luftpumpe fest angeschlossen, während das freie Ende konisch zuläuft. Auf der X-Achse befindet sich dann ein entsprechend der gewünschten Gesamtzahl von anzuschließenden Getränken skalierbarer Luftverteilerkorpus. In unserem Falle haben wir ein Design mit acht verschiedenen Ausgaben verwendet. Mit Hilfe von Endstopschaltern können sowohl die Ausgangspositionen des Verteilerblockes als auch des Verteilerkorpus genau bestimmt werden. Mit Hilfe der zu entwickelnden Steuersoftware kann dann der Luftverteilerblock an jede beliebige Ausgabestelle luftdicht andocken, Luft in das System pressen, und dann bei Erreichen der gewünschten Flüssigkeitsausgabemenge direkt wieder abdocken. Hierbei fällt der Druck im System direkt wieder, so dass praktisch kein nennenswerter Nachlauf entsteht.

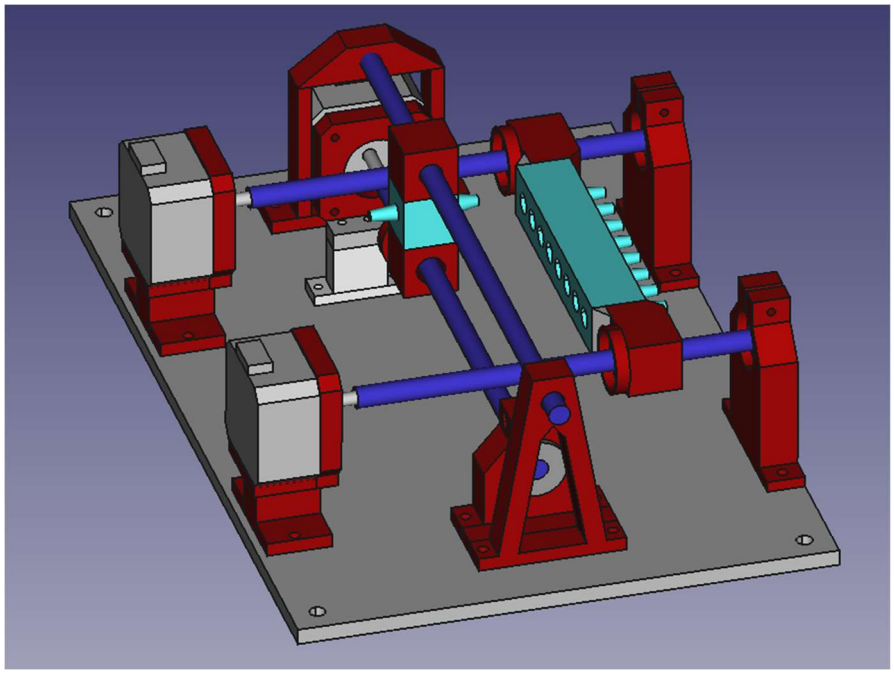


Abbildung 3: FreeCAD-Zeichnung des neu entwickelten Luftverteilers

Um den einzigen Nachteil der neuen Konstruktion zu kompensieren, in dem nun aufgrund der Druckoffenheit kein Flowmeter mehr genutzt werden kann, wird nun der Ansatz verfolgt, die ausgegebene Flüssigkeitsmenge über die Masse mit Hilfe einer HX711-Wiegezelle zu bestimmen. Hierbei wird die Wiegezelle direkt in der Getränkeausgabe verbaut und mit einem Tare-Mechanismus versehen, um einen Ausgabefehler durch eine evtl. unterschiedliche Masse verschiedener Gläser, die unter die Ausgabe gestellt werden könnten, zu vermeiden.

5.1.2. Flaschenhalter und Getränkeausgabe mit Wiegeeinheit

Durch den neuen Ansatz wurde es nötig, eine neue, praktische Flaschenhalterung mit Schlauchanschlüssen, sowie eine Ausgabeeinheit zu entwickeln, die über eine fest verbaute Wiegeeinheit verfügt. Zum Zeitpunkt des Schreibens dieser Arbeit befinden sich diese beiden Komponenten noch in der Entwicklung- und Testphase, so dass an dieser Stelle noch kein fertiges Design präsentiert werden kann.

5.2. Software

Bei dem zweiten Prototyp hat sich auch die Software zur Steuerung des Barkeepers grundlegend verändert. Denn dieser Prototyp sollte über ein Handy steuerbar sein. Dafür musste die Steuerungseinheit in der Lage sein, ein eigenes Netzwerk aufbauen zu können. In unserem Fall haben wir uns dafür entschieden, als Steuerungseinheit einen ESP32 zu verwenden. Dabei war der Grundgedanke, dass man das ganze über einen Browser steuern kann. Denn wenn man eine App entwickelt hätte, hätte man sich entscheiden müssen, ob man diese für Android, Apple oder ein anderes Betriebssystem entwickelt. Durch die Steuerung über einen Browser kann jeder, der ein Handy besitzt, den Barkeeper steuern. Das gute hierbei ist, dass dies nicht nur auf Handys begrenzt ist. Denn jedes digitale Endgerät kann sich mit einem Netzwerk verbinden und Browser nutzen.

5.2.1. Überlegung zu den wichtigsten Komponenten der Steuerung

Als erstes haben wir uns gefragt, was die Software denn auf jeden Fall können muss. Wir brauchen auf jeden Fall eine Webseite bei der man sich zwischen mehreren Drinks entscheiden und diese dann mixen lassen kann. Der ausgewählte Drink muss dann als Information an den ESP32 übertragen und gemixt werden können. In der weiteren Planung sollte dann, nachdem diese genannten Grundlagen geschaffen wurden, überlegt werden, wie man die Funktionen der Software ausbauen könnte.

5.2.2. Bauen einer Webseite die den Grundbedingungen entspricht I

Als erstes haben wir uns entschieden mit der Programmierung der Benutzeroberfläche in HTML zu beginnen. Die Webseite sollte einen Button mit einem Bild als Hintergrund in der Mitte der Webseite haben. Mit Druck auf diesen Button sollte der auf dem Hintergrundbild abgebildete Drink, gemixt werden können. Außerdem sollte es links und rechts neben dem mittleren Button jeweils einen Menübutton geben. Diese sollten große Pfeile, die in entgegengesetzte Richtungen zeigen, als Hintergrund haben. Mit diesen sollte man in der Lage sein durch die Drinks gehen zu können. Zunächst wurden daher drei Buttons implementiert und diesen ein Hintergrundbild gegeben. Für die Hintergründe der Bilder wurde dann der sogenannte „Stylesheet“ implementiert. In diesem wird der komplette CSS-Teil der Webseite geschrieben. Danach wurde die grobe Position der Buttons angepasst. Die Positionen der Buttons wurde mit dem Attribut „absolut“ und einer Angabe der Positionen in Prozent angegeben. Jedoch stellt sich hierbei das Problem, dass sich die Hintergrundbilder nicht automatisch an die Größe der Buttons anpassen. Es hat sich herausgestellt, dass wenn das Bild zu klein ist um den Button vollständig zu bedecken, HTML das Bild einfach noch mal an die Stelle, wo der Button noch nicht mit dem Bild vollständig bedeckt ist, kopiert (siehe

Abbildung). Als Lösung hierfür haben wir das Attribut „background-size“ verwendet. Denn mit diesem Attribut kann genau festgelegt werden, wie viel von dem Button durch das Hintergrundbild ausgefüllt werden soll. Es könnte hier zum Festlegen die Einheit Pixel (px) verwendet werden, es ist aber auch hier sinnvoller direkt Prozent zu verwenden, denn so ist die absolute Größe des Buttons egal, da so entsprechend der Größe des Buttons das Bild immer einen

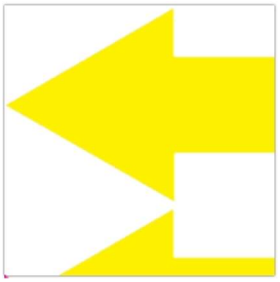


Abbildung 4: Problem Hintergrundbild des Buttons

bestimmten Anteil des Buttons ausfüllt. Wenn die Webseite also sich hinterher der Größe des Geräts des Users anpassen soll und damit auch die Buttons skaliert werden, wird es hierbei keine Probleme mit der Skalierung des Bilder innerhalb der Buttons geben, da in unserem Fall das Bild immer 100% des Buttons ausfüllt. Damit zwischen den verschiedenen Drinks gewechselt werden kann, wurde JavaScript genutzt. In JavaScript wird immer, wenn auf den rechten Pfeil gedrückt wird, eine Variable namens „Drink“ um eins erhöht. Außer wenn die Variable den Wert neun schon erreicht hat. Dann wird die Variable wieder auf null gesetzt, damit, wenn alle Drinks durchgewechselt wurden, wieder zu dem ersten Drink gewechselt wird und somit eine Loop entsteht. Dasselbe gilt für den linken Pfeil, hier wird jedoch immer „Drink“ um eins erniedrigt und nachdem der Wert null erreicht, wird die Variable wieder auf 9 gesetzt. Eine Herausforderung war jedoch, dass das Hintergrundbild des mittleren Buttons dabei der „Drink“ Variable entsprechend durchwechseln muss. Denn um das Hintergrundbild eines Buttons anzugeben braucht man CSS. Jedoch unterstützt CSS weder Variablen noch Events und kann damit nicht direkt verwendet werden um das Hintergrundbild zu ändern, sondern nur um das anfängliche Bild festzulegen. Um dies zu erreichen kann der Code *document.getElementById("X").style.backgroundImage = „url(‚Hintergrundbild.png‘)“;* verwendet werden. Hierbei wird zuerst nach unserem Element, welches unser Button ist, mithilfe seiner ID gesucht und eine Referenz zu diesem Objekt zurückgegeben. Danach wird das Style-Objekt aufgerufen und die Eigenschaft „backgroundImage“ des Buttons der URL entsprechend verändert. Diese Zeile musste dann noch so modifiziert werden, dass sie auf unseren Code abgestimmt ist. Denn momentan würde hierbei das Hintergrundbild unabhängig von der „Drink“-Variable zu „Hintergrundbild.png“ wechseln. Ziel ist es aber, dass sich das Hintergrundbild abhängig von „Drink“ ändert. Um dies zu schaffen, muss sich erst einmal angeguckt werden, was in die Definition von „url(‚)“ hinterlegt werden muss. Denn hier muss der Pfad des Bildes, ausgehend von dort wo die HTML-Datei gespeichert ist, angegeben werden. In unserem Projekt haben wir zwecks Übersichtlichkeit die Ordnerstruktur „Pictures/Drinks gewählt. In dem Ordner „Drinks“ befinden sich die jeweiligen Bilder, daher lautet der Speicherpfad für die Bilder: „Pictures/Drinks/Hintergrundbild.png“. Um nun variabel die Bilder ändern zu können, wurde eine weitere Variable erstellt. Diese Variable ist ein Array mit Strings und beinhaltet die einzelnen Pfade zu den verschiedenen Bildern, wie zum Beispiel: „url(Pictures/Drinks/colarum.png)“. Um nun das Bild zu ändern, wird jedes Mal, wenn auf den Pfeil nach rechts oder nach links geklickt wird, diese Codezeile ausgeführt: *document.getElementById("m").style.backgroundImage = DrinkImageloc[Drink];* Hierbei wird auf die Variable DrinkImageloc zugegriffen, welche unser Array ist und davon wird als Index die „Drink“ Variable benutzt. Damit haben wir eine voll funktionsfähige Webseite, die je nachdem welchen Pfeil man drückt, das Hintergrundbild des mittleren Buttons wechselt. Bis zu diesem Punkt hat der mittlere Button selber jedoch noch keine Funktion. Anschließend erfolgten weitere Anpassungen der Webseite mittels CSS, wie zum Beispiel, dass die Buttons sich hoch und runter skalieren, je nachdem wie groß der Bildschirm des Users ist oder ein Feedback bei drücken eines Buttons. Beim Drücken des Buttons wird dieser kleiner und der Nutzer erhält damit eine visuelle Rückmeldung, dass er den Button korrekt gedrückt hat.

5.2.3. Bauen einer Webseite die den Grundbedingungen entspricht II

Nachdem die Basis für eine funktionelle Webseite gegeben war, galt es diese so in Arduino zu übertragen, dass die Webseite auch auf ESP32 ausführbar ist. Dies Bedarf einiger Anpassungen. Dafür werden erst einmal vier Bibliotheken benötigt, welche eingebunden werden müssen.

```
#include <SPIFFS.h>
#include <WiFi.h>
#include <ESPAsyncWebServer.h>
#include <AsyncTCP.h>
```

Abbildung 5: Einbinden der Bibliotheken

Zuerst brauchten wir die WiFi-Bibliothek. Diese wird verwendet, damit der ESP32 ein eigenes Netzwerk aufbauen kann. In unserem Fall ist der ESP32 ein WiFi Access Point (AP), damit dieser sich direkt mit einem Client/User verbinden kann, ohne dass ein Router oder eine andere Netzwerkstruktur benötigt wird. Die nächste Bibliothek ist die ESPAsyncWebServer-Bibliothek. Diese ermöglicht es dem ESP einen Webserver einzurichten und HTTP-Anfragen asynchron zu bearbeiten. Dies bedeutet, dass der Server mehrere Anfragen gleichzeitig bearbeiten kann ohne, dass eine Anfrage blockiert oder auf die Bearbeitung der nächsten Anfrage gewartet werden muss. Dadurch können sich mehrere User gleichzeitig auf der Webseite befinden und ihren Drink aussuchen. Damit diese Bibliothek vollumfänglich genutzt werden kann, wird indirekt noch eine weitere Bibliothek benötigt. Und zwar die AsyncTCP-Bibliothek. Diese unterstützt das Transmission Control Protocol (TCP), welches ein Kommunikationsprotokoll ist, dass eine Verbindung zwischen einem Client und einem Server aufbauen kann. Eine weitere Bibliothek, die verwendet werden musste, ist die SPIFFS-Bibliothek. SPIFFS steht für „Serial Peripheral Interface Flash File System“ und wird benutzt, um Dateien in einem Flash-Speicher speichern zu können. Im folgenden Verlauf wird noch erklärt wofür genau wir diese Bibliothek verwenden. Nachdem die Bibliotheken inkludiert wurden, musste ein Char Array definiert werden. Dieser wird dann im „PROGMEM“ gespeichert. Das ist ein Flash-Speicher, welcher für Daten genutzt wird, welche persistent, also auch wenn keine Spannung am Barkeeper anliegt, gespeichert bleiben müssen. In diesen Char Array wird die gesamte Webseite hineingeschrieben (siehe Abschnitt 5.2.2). Nach dem einbinden der Bibliotheken mussten zwei sogenannte „pointer chars“ festgelegt werden, welche sich nicht während der Ausführung des Codes verändern und deswegen als „const char“ deklariert wurden. Der erste dieser Chars ist für die Benennung des Netzwerks zuständig, während der zweite für das dazugehörige Passwort verwendet werden muss. In dem letzten Teil des Codes vor der Setup Funktion musste noch deklarieret werden, dass es sich um einen asynchronen Webserver handelt, welcher auf Port 80 läuft. Danach folgt der Setup-Teil. Als erstes muss, damit später auf den SPIFFS zugegriffen werden kann, dieser initialisiert werden. Dies erfolgt durch das Statement „SPIFFS.begin();“. Im nächsten Schritt wird der Aufbau eines eigenen Netzwerkes codiert. Da ein eigenes Netzwerk aufgebaut werden soll, handelt es sich um einen „softAP“. In dem nächsten Statement wird der vorher im const char festgelegte Name des Netzwerks und das dazugehörige Passwort initialisiert. Danach wird, nachdem der User eine Anfrage zum Öffnen der Webseite gestellt hat, diesem der Code für die Webseite geschickt. Dies ist der Teil, der als Char Array im „PROGMEM“ gespeichert wurde.

5.2.4. Hintergrundbilder

Würde der Code, wie er bisher beschrieben wurde, ausgeführt werden, so würden die Hintergrundbilder der Buttons fehlen. Dies liegt daran, dass der ESP diese Bilder auf den vorher angegebenen Pfaden nicht finden kann, da diese nirgendwo in dem Speicher des ESPs vorhanden sind. Hierfür kommt nun die SPIFFS-Bibliothek zum Einsatz. Denn hiermit können die Bilder, welche für die Webseite benötigt werden, in den Flash-Speicher des ESPs hochgeladen werden und somit dem Server den Zugriff auf diese ermöglichen.

5.2.5. Kommunikation vom Webserver zum ESP

Um nun einen Drink mixen zu können, muss der ESP wissen, welchen Drink er mixen soll. Damit dies funktioniert muss es eine Kommunikation vom Webserver zum ESP geben. Dies ist ein komplexes Thema und es gibt mehrere Lösungsansätze um eine solche Kommunikation zu ermöglichen. Eine erster Ansatz war, dass die Webseite weiß welcher Drink als Hintergrund gezeigt werden soll, weil die Variable „Drink“ in dem JavaScript-Teil der Webseite dies vorgibt. Die beste Lösung wäre es natürlich immer, wenn sich der Wert dieser Variable ändert, dieser Wert direkt an den ESP gesendet wird. Dies hat sich aber anfänglich als schwer zu bewerkstelligen herausgestellt, weswegen der erste Ansatz daraus bestand, dass auf dem Server eine Subdomain genutzt wird, um zwischen dem ESP und dem Webserver zu kommunizieren. Dabei wird an die Hauptdomain ein „/“ mit einer Spezifikation des weiteren Pfades angehängt. Durch diesen Pfad wird eine bestimmte Ressource abgefragt, auf welche der ESP reagieren kann. Diese Reaktion des ESPs ist eine Funktion, die beim Abrufen dieses Pfades aufgerufen und ausgeführt wird. Erstellt wurden zwei solcher Pfade. Der erste Pfad für den linken Pfeil und der zweite Pfad für den rechten Pfeil. Immer wenn auf einen der beiden Pfeile gedrückt wird, wird die im Arduino dazugehörige Funktion ausgeführt, welche dann entweder eine in Arduino erstellte Variable addiert oder subtrahiert. Diese Variable wäre damit kohärent zu der „Drink“ Variablen auf der Webseite. Diese Herangehensweise beruht aber auf einen fundamentalen Fehler. Denn immer, wenn dieser Pfad aufgerufen wird, wird von dem Server als Antwort darauf der Code für die Webpage ein weiteres Mal gesendet und somit die Webseite aktualisiert. Dadurch dass die Seite neu lädt, resettet sich aber auch die „Drink“ Variable im JavaScript Code der Webseite, wodurch immer das erste Bild gezeigt werden würde. Der zweite Ansatz, beruht auch auf dem Prinzip definierter Pfade. Dafür wird in dem JavaScript-Teil, wenn auf den Drink, der gemixt werden soll, geklickt wird, eine switch case Funktion im JavaScript-Teil ausgeführt, welche abhängig von der Variablen „Drink“ ist. Je nachdem welchen Wert die Variable hat, wird an die Domain als Pfad noch der Name des ausgewählten Drinks angehängt. So können dann in der Funktion im Arduino Code die für die spezifische Anfrage zugehörigen Statements ausgeführt werden. Nachdem diese Idee implementiert wurde, wurde jedoch weiter recherchiert, wie sich die „Drink“ Variable von der Webseite zum ESP übergeben lässt. Schlussendlich wurde ein Lösungsansatz gewählt, bei dem das HTTP_GET Protokoll genutzt wird. Bei diesem wird in JavaScript eine XMLHttpRequest geöffnet. Dort wird der Pfad angegeben, welcher angefragt werden soll. Auf diesen Pfad reagiert dann der ESP entsprechend. Hinter dem Namen des Pfades wird dann noch der Wert einer Variablen angegeben. Diese Variable kann dann vom ESP ausgelesen werden. Dafür muss der Pfad wie folgt angegeben werden: „.../pfad?value=+Variable“. Dieser Teil hinter dem Pfad wird als Parameter bezeichnet und kann so dann vom ESP als String ausgelesen und danach wieder mit der „.toInt()-Funktion“ zu einem Integer umgewandelt und in einer Variablen gespeichert werden.

5.2.6. Auslagern von dem Code der Webseite in eine eigene Datei zur besseren Übersicht

Die Auslagerung des HTML Codes zu einer eigenen Datei funktioniert relativ simpel. Hierfür wird eine neue Textdatei in demselben Ordner wie der Sketch erstellt. Danach wird der Textdatei anstelle der Endung „.txt“ die Endung „.h“ gegeben. Dadurch wird diese Datei ein C++ Header. Diese Datei kann nun in den Code implementiert werden, indem einfach *#include „Datei.h“;* an den Anfang des Arduino Codes geschrieben wird.

```
#include "HTMLMain.h"
```

Abbildung 6: Einbinden von ausgelagertem HTML Code

Danach kann der Char Array für die Webseite einfach in die neue „.h-Datei“ geschrieben werden. Hiermit kann nun diese Variable abgerufen werden, als wäre diese eine ganz

normale Variable, die im selben Code wie der C++ Code steht. Jedoch gibt es zusätzlich noch den Vorteil, dass der Code um einiges aufgeräumter und übersichtlicher aussieht.

5.2.7. Kommunikation vom ESP zum Webserver

Die Kommunikation vom ESP zum Webserver ist von essentieller Bedeutung für die Work in progress Seite, welche im späteren Verlauf dieser schriftlichen Arbeit noch im Detail erklärt wird (siehe Abschnitt 5.2.8). Um eine Kommunikation von einem ESP zu einem Webserver zu ermöglichen, muss der Webserver eine Anfrage stellen. Denn es gibt keinen direkten Weg, wie der ESP Daten zum Webserver schicken kann. Es muss erst eine Anfrage der Daten von dem Webserver geben. Damit eine Variable zum Webserver geschickt werden kann, wird eine setInterval()-Funktion im JavaScript-Teil der Webseite benutzt. Diese führt eine Funktion innerhalb der setInterval()-Funktion immer wieder nach einer festgelegten Zeit aus. Damit nun Daten von dem ESP zum Webserver gesendet werden können, verwenden wir das XHTTP Protokoll. Dafür wird eine neue Anfrage geöffnet. Danach wird darauf gewartet, dass eine sogenannte ReadyState-Variable den Wert vier annimmt. Die ReadyState-Variable gibt den Status der Anfrage an. Erreicht die Variable den Wert vier wurde eine Anfrage an den ESP gestellt, eine Antwort empfangen und die Anfrage abgeschlossen. Dabei wird eine bestimmte Subdomain abgefragt, auf welche der Server dann mit einem definierten Wert antwortet.

5.2.8. Work in progress Seite

Die WIP (Work in Progress) Seite öffnet sich, sobald ein Drink ausgewählt wurde und somit der Mixvorgang begonnen hat. Ziel dieser Seite ist es, dass der User weiß, dass sein Drink gerade gefertigt wird und vor allem, dass dem User die Möglichkeit gegeben wird, den Mixprozess des Drinks, mit Hilfe eines sich auf dieser Webseite befindlichen Buttons, abzubrechen. Nach dem dann der Drink fertig gemixt wurde, wird der User wieder auf die Hauptseite geleitet. Damit dies funktioniert wird eine Kommunikation von dem ESP zum Webserver benötigt. In dem Arduino Code gibt es eine Variable, welche den Wert null oder eins annehmen kann. Wenn ein Drink gerade gemixt wird, ist dieser Wert null. Wenn nun der Drink fertig gemixt wurde, ändert sich dieser für eine kurze Zeit zu dem Wert eins. Dieser Wert wird dann, wie in Abschnitt 5.2.7 beschrieben, durch eine setInterval()-Funktion mit dazugehöriger Anfrage auf der Webseite durchweg abgefragt. Dabei wird der Wert als String gesendet und im JavaScript Teil wieder mit einer parseInt()-Funktion zu einem Integer umgeformt. Wenn der User sich nun auf dieser Seite befindet und der Wert der Variablen auf null bleibt, passiert nichts. Ändert sich der Wert jedoch zu eins, ist die in der setInterval-Funktion gestellte if-Bedingung erfüllt und der User wird durch eine Subdomain wieder zur Hauptseite zurückgeleitet. Betätigt der User auf der Seite den Abbruch-Knopf, taucht ein Popup auf, welches noch einmal um Bestätigung für den Abbruch des Drinks bittet. Nachdem auch hier auf Okay gedrückt wird, wird man zur Startseite zurückgeleitet und der in Auftrag gegebene Drink wird abgebrochen. Würde der Code nun so verwendet werden, gäbe es ein Problem, denn wenn nun der User aus Versehen die Webseite schließt und dann wieder den Link für die Hauptseite eingibt oder eine andere Person sich einloggt, gäbe es keine Möglichkeit mehr, den Drink abzubrechen. Deswegen muss jedes Mal, wenn die Hauptseite geöffnet wird, je nachdem ob sich ein Drink gerade in Bearbeitung befindet oder nicht entweder der Code für die WIP Seite oder die Hauptseite an den User gesendet werden.

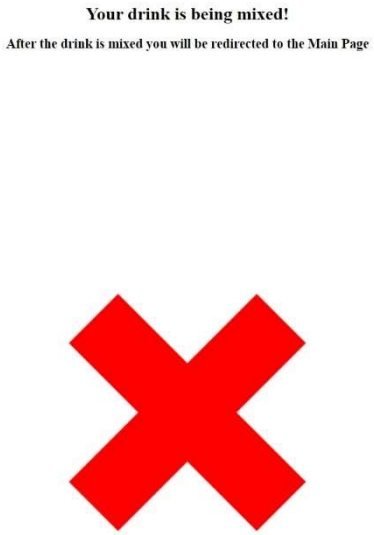


Abbildung 7: WIP Seite

5.2.9. Mixing Code

Damit der richtige Drink nun gemixt wird, muss der Code erst einmal wissen, welcher Drink gemixt werden soll. Dazu wird die am Ende von Abschnitt 5.2.5 bereits angesprochene Variable benötigt. Diese Variable wird im Code „bw“ genannt und hat standardmäßig den Wert -1. Sie gibt Auskunft darüber, welcher Drink gemixt werden soll. Zum Beispiel bedeutet der Wert 0, dass Cola-Rum gemixt werden soll und 3 Gin Tonic. In dem Loop-Teil des Arduino Codes gibt es eine switch case Funktion. Diese ist abhängig vom „bw“ Wert. In der switch case Funktion gibt es die Cases von 0-8. Solange „bw“ -1 beträgt, passiert nichts. Wenn nun aber der User einen Drink ausgesucht hat, ändert sich diese Variable zu dem Wert des jeweiligen Drinks. Den darauffolgenden Ablauf des Codes wird an Case 0 erklärt, was bedeuten würde, dass der User sich für Cola-Rum entschieden hat. Nachdem sich der Wert der Variable „bw“ geändert hat, wird das nächste Mal, wenn der Loop-Teil durchlaufen wird, Switch case 0 ausgeführt. Das erste, was hier passiert ist, dass der Wert der „bw“ Variable direkt wieder zu -1 geändert wird, damit der Case kein weiteres Mal ausgeführt wird. Als nächstes wird eine weitere Variable namens WIP auf true gestellt. Diese verhindert, dass ein anderer User einen Drink in Auftrag geben kann, während gerade schon ein Drink gemixt wird. Die Variable sagt damit aus, dass sich gerade ein Drink in Auftrag befindet. Als nächstes wird die Funktion „homing()“ aufgerufen. Diese dient dazu den Barkeeper auf seine Grundposition zu fahren. Dies dient unter anderem der Vorsorge bei einer Spannungsunterbrechung. Denn wird der Barkeeper vom Netz getrennt, während dieser gerade dabei ist einen Drink zu mixen, geht die Information über die aktuelle Position verloren. Durch den Start von der Grundposition aus, ist dies jedoch für das Mixen eines neuen Drinks unerheblich. Damit der Barkeeper weiß, wie viel von einer bestimmten Flüssigkeit für einen bestimmten Drink verwendet werden muss, ist am Anfang des Codes noch ein 2D-Array definiert. Bei diesem stehen die Spalten jeweils für die einzelnen Drinks, die man mixen kann und die Zeilen für die einzelnen Zutaten, die für den jeweiligen Drink benötigt werden. Dabei wird die Menge der benötigten Flüssigkeiten in Gramm angegeben. Dies liegt daran, dass wir eine Waage verwenden, um zu messen, wie viel von einer Flüssigkeit ausgegeben wurde. Dabei müssen wir davon ausgehen, dass ein Gram einem Milliliter entspricht. In dem Array gibt es in jeder Zeile jeweils 8 Flüssigkeiten, zu denen die Grammzahl, die für den jeweiligen Drink benötigt wird, genau spezifiziert ist. Nachdem die Homing Routine ausgeführt wurde, wird eine for-Schleife genutzt, um jede einzelne der Flüssigkeiten durchzugehen. In dieser for-Schleife wird dann die Grammanzahl

```
switch (bw) {
  case 0:
    bw = -1;
    WIP = true;
    homing();
    if (cancel == 0) {
      for (int i = 0; i < 8; i++) {
        if (PreDrink[0][i] > 0) {
          docking();
          // Alles was jetzt kommt
          scale.tare();
          while(scale.get_units() < PreDrink[0][i]){
            digitalWrite(pump, HIGH);
          }
          digitalWrite(pump, LOW);
          Xpos0();
          Multiply = 1;
        }
        else {
          Multiply++;
        }
      }
      Ready0 = 1;
    }
    else {
      cancel = 0;
    }
    WIP = false;
    break;
```

Abbildung 8: Loop Code für Case 0

```
//Reihenfolge: Gin, Rum, Lillet, Cola, Tonic, Wild Berry, Zitronensaft
int PreDrink[9][8] = {
  {0, 40, 0, 200, 0, 0, 20, 0}, //Colarum
  {0, 60, 0, 0, 0, 0, 30, 0}, //daiquiri
  {40, 0, 0, 0, 0, 0, 20, 0}, //gimlet
  {40, 0, 0, 0, 200, 0, 0, 0}, //gintonic
  {0, 0, 60, 0, 120, 0, 0, 0}, //lillettonic
  {0, 0, 60, 0, 0, 120, 0, 0}, //lilletwb
  {40, 0, 0, 0, 150, 50, 0, 0}, //pinkgt
  {0, 0, 0, 0, 0, 0, 0, 0}, //Placeholder
  {25, 25, 25, 25, 25, 25, 25, 25}  //Reinigung
};
```

Abbildung 9: 2D Array

für die jeweilige Flüssigkeit des jeweiligen Drinks (in unserem Fall für Cola-Rum) abgerufen und verglichen, ob diese größer als null ist. Ist dies nicht der Fall, wird direkt zur nächsten Flüssigkeit gesprungen. Wenn nun eine Zutat des Drinks größer als null ist, weiß der Barkeeper, dass für diese Flüssigkeit an dem richtigen Anschluss angedockt werden muss. Deswegen wird dann die docking()-Funktion ausgeführt, welche dann zu dem entsprechenden Anschluss fährt. Anschließend wird die verwendete Waage tariert und solange das entsprechende Gewicht auf der Waage kleiner als das des vorgegebenen Werts ist, die Pumpe aktiviert und somit die Flüssigkeit in das Glas befördert. Dies wird für alle acht Flüssigkeiten wiederholt. Nachdem dies abgeschlossen ist, wird die WIP-Variable wieder auf false gesetzt.

5.2.10. Homing Funktion

Die Homing Funktion dient dazu, dass bevor ein Drink gemixt werden kann, der Barkeeper auf die Grundposition fährt. Dafür wird zuerst der Motor, welcher für die X-Achse zuständig ist, solange aktiviert, bis der Endstop aktiviert wurde. Danach wird dies für die Y-Achse wiederholt. Nachdem auch das geschehen ist, fährt der Barkeeper auf der Y-Achse auf die Position des ersten Anschlusses, dockt jedoch nicht mit der X-Achse an. Dies passiert hier, da der Abstand zwischen den einzelnen Anschlüssen immer gleich ist, jedoch der Abstand zwischen dem Endstop und dem ersten Anschluss anders ist, als der Abstand zwischen den einzelnen Anschlüssen. Hierdurch wird die Programmierung der Docking Funktion erleichtert.

```
void homing() {
  while (digitalRead(endstopX) == 1 && cancel == 0) {
    digitalWrite(dirX, HIGH);
    digitalWrite(stpX, HIGH);
    delay(2);
    digitalWrite(stpX, LOW);
    delay(2);
  }
  while (digitalRead(endstopY) == 1 && cancel == 0) {
    digitalWrite(dirY, LOW);
    digitalWrite(stpY, HIGH);
    delay(2);
    digitalWrite(stpY, LOW);
    delay(2);
  }
  for (int i = 0; i < 110; i++) {
    digitalWrite(dirY, HIGH);
    digitalWrite(stpY, HIGH);
    delay(2);
    digitalWrite(stpY, LOW);
    delay(2);
  }
}
```

Abbildung 10: Homing Code

5.2.11. Docking Funktion

Die Docking Funktion wird dazu verwendet, den Barkeeper an den richtigen Anschluss zu fahren und anzudocken. Dazu wird ein for-loop benutzt. In diesem wird der Motor für die Y-Achse für 2 Millisekunden angeschaltet. Dies entspricht genau einem halben Schritt. Durch vorangegangene Tests ist bekannt, dass der Abstand zwischen den einzelnen Anschlüssen 570 halbe Schritte beträgt. Wenn nun also der dritte Anschluss angefahren werden soll, muss dieser Wert somit nur noch mit 2 multipliziert werden.

```
void docking() {
  for (int i = 0; i < 570 * Multiply; i++) {
    digitalWrite(dirY, HIGH);
    digitalWrite(stpY, HIGH);
    delay(2);
    digitalWrite(stpY, LOW);
    delay(2);
  }
  for (int i = 0; i < 560; i++) {
    digitalWrite(dirX, LOW);
    digitalWrite(stpX, HIGH);
    delay(2);
    digitalWrite(stpX, LOW);
    delay(2);
  }
}
```

Abbildung 11: Docking Code

5.2.12. Erweiterungen der Grundfunktionen

Nachdem die Programmierung der Grundfunktionen abgeschlossen war, lag der Fokus auf der Erweiterung der Grundfunktionen des Barkeepers. Ziel war es ein besseres Benutzererlebnis zu schaffen. In dem folgenden Abschnitt werden diese Erweiterungen der Grundfunktionen erklärt. Auf eine detaillierte Erläuterung des Codes wird verzichtet, da dies den Rahmen der schriftlichen Arbeit sprengen würde.

5.2.12.1. Custom Seite

Bis zu dem jetzigem Zeitpunkt konnten nur schon vordefinierte Drinks ausgewählt werden. Das limitiert die möglichen Drinks, die der User zur Auswahl hat, stark. Die Custom Seite ist eine weitere Funktion, die im Hauptmenü am Ende aller voreingestellten Drinks gefunden werden kann. Durch einen Klick auf das dazugehörige Icon, wird der User zur der sogenannten Custom Seite geleitet. Hier hat der User volle Kontrolle über die verschiedenen Flüssigkeiten, die angeschlossen sind. Dafür gibt es auf dieser Seite acht Zahleneingabefelder und jedes Zahlenfeld steht für einen der Anschlüsse. In die Zahlenfelder kann der User dann eingeben, wie viele Milliliter von den bestimmten Flüssigkeiten gewünscht sind. Des Weiteren, gibt es auf dieser Seite einen Zurück-Knopf mit dem der User wieder zur Hauptseite gelangen kann. Wenn der User mit seinen Eingaben fertig ist, kann der Button „Mix“ betätigt werden. Anschließend fängt der Barkeeper an den Drink zu mixen.

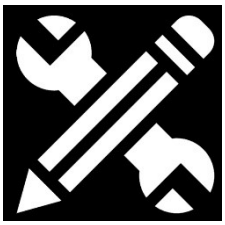


Abbildung 13: Custom Seite Icon

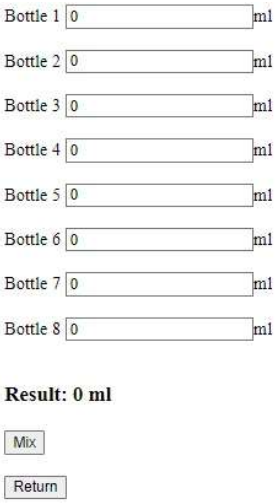


Abbildung 12: Custom Seite

5.2.12.2. Preset Option

Die Preset Option kann als letztes Element auf der Hauptseite unter den Einstellungen gefunden werden. Hier kann eine Option ausgewählt werden, die mit einer Checkbox funktioniert. Ist diese Option aktiviert, wird der User, sobald dieser auf einen Preset Drink geklickt hat, nicht direkt zur WIP Seite, sondern auf eine ähnliche Seite wie die Custom Seite geleitet. Der einzige Unterschied zu der Custom Seite ist hierbei, dass die Einstellungen für den Preset Drink, welchen man ursprünglich ausgewählt hat, schon in den Zahleneingabefeldern eingegeben sind. Dies hat den Vorteil, dass wenn dem User einer der Preset-Drinks grundsätzlich gefällt, dieser aber noch eine gewissen Menge einer Flüssigkeit verändern möchte, genau das getan werden kann.

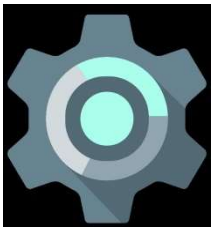


Abbildung 14: Einstellungsseite Icon

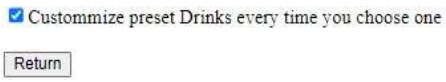


Abbildung 15: Einstellungsseite

5.2.12.3. Drink Save Funktion

Die Drink Save Funktion ist noch nicht fertig und es wird noch an dieser gearbeitet. Mit der Drink Save Funktion soll auf der Custom Seite noch ein weiterer Button hinzugefügt werden. Mit diesem Button kann der User seinen selbst erstellten Drink speichern und somit immer wieder auswählen. Bei Nutzung des Button wird nach einem Namen für den Drink gefragt unter dem man diesen speichern möchte. Der Drink steht dem User ab diesem Zeitpunkt durch ein eigenes Icon auf der Hauptseite zur erneuten Erstellung zur Auswahl. Zudem können dort auch Drink gelöscht oder umbenannt werden.

6. Anhang

6.1. Haupt-Code

```
#include <SPIFFS.h>
#include <WiFi.h>
#include <ESPAsyncWebServer.h>
#include <AsyncTCP.h>
#include <HX711.h>
#include "HTMLMain.h"
#include "HTMLCustom.h"
#include "HTMLWIP.h"
#include "HTMLSettings.h"
#include "HTMLPresetCustom.h"

////////////////////////////////////////////////////////////////////////////////

const char *SSID = "ESP-AP-Barkeeper";
const char *PASSWORD = "12345678";
bool WIP = false;
int tasterstatus;
int steps = 2048;
int bw = -1;
int cancel;
int Ready0;
String Ready1() {
  return String(Ready0);
}
int Settinga;
String Settingstring() {
  return String(Settinga);
}
int Multiply;

////////////////////////////////////////////////////////////////////////////////

int B[] = {0, 0, 0, 0, 0, 0, 0, 0};
//Reihenfolge: Gin, Rum, Lillet, Cola, Tonic, Wild Berry, Zitronensaft
int PreDrink[9][8] = {
  {0, 40, 0, 200, 0, 0, 20, 0}, //Colarum
  {0, 60, 0, 0, 0, 0, 30, 0}, //daiquiri
  {40, 0, 0, 0, 0, 0, 20, 0}, //gimlet
  {40, 0, 0, 0, 200, 0, 0, 0}, //gintonic
  {0, 0, 60, 0, 120, 0, 0, 0}, //lillettonic
  {0, 0, 60, 0, 0, 120, 0, 0}, //lilletwb
  {40, 0, 0, 0, 150, 50, 0, 0}, //pinkgt
```

```
// Pins in the right Order: 15, 16, 17, 19, 21, 22

////////////////////////////////////////////////////////////////////////////

String processor1() {
  if (WIP) {
    return WIPpage;
  }
  else {
    return webpage;
  }
}

AsyncWebServer server(80);

////////////////////////////////////////////////////////////////////////////

void homing() {
  while (digitalRead(endstopX) == 1 && cancel == 0) {
    digitalWrite(dirX, HIGH);
    digitalWrite(stpX, HIGH);
    delay(2);
    digitalWrite(stpX, LOW);
    delay(2);
  }
  while (digitalRead(endstopY) == 1 && cancel == 0) {
    digitalWrite(dirY, LOW);
    digitalWrite(stpY, HIGH);
    delay(2);
    digitalWrite(stpY, LOW);
    delay(2);
  }
  for (int i = 0; i < 110; i++) {
    digitalWrite(dirY, HIGH);
    digitalWrite(stpY, HIGH);
    delay(2);
    digitalWrite(stpY, LOW);
    delay(2);
  }
}

void docking() {
```

```
  for (int i = 0; i < 570 * Multiply; i++) {
    digitalWrite(dirY, HIGH);
    digitalWrite(stpY, HIGH);
    delay(2);
    digitalWrite(stpY, LOW);
    delay(2);
  }
  for (int i = 0; i < 560; i++) {
    digitalWrite(dirX, LOW);
    digitalWrite(stpX, HIGH);
    delay(2);
    digitalWrite(stpX, LOW);
    delay(2);
  }
}

void Xpos0() {
  for (int i = 0; i < 500; i++) {
    digitalWrite(dirX, HIGH);
    digitalWrite(stpX, HIGH);
    delay(2);
    digitalWrite(stpX, LOW);
    delay(2);
  }
}

HX711 scale;
////////////////////////////////////////////////////////////////////////////////
void setup() {
  Serial.begin(115200);
  SPIFFS.begin();
  WiFi.softAP(SSID, PASSWORD);
  Serial.println(WiFi.softAPIP());
  pinMode(stpX, OUTPUT);
  pinMode(dirX, OUTPUT);
  pinMode(stpY, OUTPUT);
  pinMode(dirY, OUTPUT);
  pinMode(pump, OUTPUT);
  pinMode(endstopY, INPUT_PULLUP);
  pinMode(endstopX, INPUT_PULLUP);
  scale.begin(wdout, wclock);
  scale.set_scale(416.6152915013);
```

```
  server.on("/", [](AsyncWebServerRequest * request) {
    request->send_P(200, "text/html", processor1().c_str());
  });

////////////////////////////////////////////////////////////////////////////////

  server.on("/Arrowleft", HTTP_GET, [](AsyncWebServerRequest * request) {
    request->send(SPIFFS, "/Arrowleft.png", "image/png");
  });
  server.on("/Arrowright", HTTP_GET, [](AsyncWebServerRequest * request) {
    request->send(SPIFFS, "/Arrowright.png", "image/png");
  });
  server.on("/colarum", HTTP_GET, [](AsyncWebServerRequest * request) {
    request->send(SPIFFS, "/colarum.png", "image/png");
  });
  server.on("/daiquiri", HTTP_GET, [](AsyncWebServerRequest * request) {
    request->send(SPIFFS, "/daiquiri.png", "image/png");
  });
  server.on("/gimlet", HTTP_GET, [](AsyncWebServerRequest * request) {
    request->send(SPIFFS, "/gimlet.png", "image/png");
  });
  server.on("/gintonic", HTTP_GET, [](AsyncWebServerRequest * request) {
    request->send(SPIFFS, "/gintonic.png", "image/png");
  });
  server.on("/lillettonic", HTTP_GET, [](AsyncWebServerRequest * request) {
    request->send(SPIFFS, "/lillettonic.png", "image/png");
  });
  server.on("/lilletwb", HTTP_GET, [](AsyncWebServerRequest * request) {
    request->send(SPIFFS, "/lilletwb.png", "image/png");
  });
  server.on("/pinkgt", HTTP_GET, [](AsyncWebServerRequest * request) {
    request->send(SPIFFS, "/pinkgt.png", "image/png");
  });
  server.on("/custom", HTTP_GET, [](AsyncWebServerRequest * request) {
    request->send(SPIFFS, "/custom.png", "image/png");
  });
  server.on("/reinigung", HTTP_GET, [](AsyncWebServerRequest * request) {
    request->send(SPIFFS, "/reinigung.png", "image/png");
  });
  server.on("/cancelp", HTTP_GET, [](AsyncWebServerRequest * request) {
    request->send(SPIFFS, "/cancelp.png", "image/png");
```

```
  });
  server.on("/settingsp", HTTP_GET, [](AsyncWebServerRequest * request) {
    request->send(SPIFFS, "/settingsp.png", "image/png");
  });

////////////////////////////////////////////////////////////////////////////

  server.on("/selected", HTTP_GET, [] (AsyncWebServerRequest * request) {
    bw = request->getParam("value")->value().toInt();
  });

////////////////////////////////////////////////////////////////////////////

  server.on("/B1", HTTP_GET, [] (AsyncWebServerRequest * request) {
    B[0] = request->getParam("value")->value().toInt();
  });

  server.on("/B2", HTTP_GET, [] (AsyncWebServerRequest * request) {
    B[1] = request->getParam("value")->value().toInt();
  });

  server.on("/B3", HTTP_GET, [] (AsyncWebServerRequest * request) {
    B[2] = request->getParam("value")->value().toInt();
  });

  server.on("/B4", HTTP_GET, [] (AsyncWebServerRequest * request) {
    B[3] = request->getParam("value")->value().toInt();
  });

  server.on("/B5", HTTP_GET, [] (AsyncWebServerRequest * request) {
    B[4] = request->getParam("value")->value().toInt();
  });

  server.on("/B6", HTTP_GET, [] (AsyncWebServerRequest * request) {
    B[5] = request->getParam("value")->value().toInt();
  });

  server.on("/B7", HTTP_GET, [] (AsyncWebServerRequest * request) {
    B[6] = request->getParam("value")->value().toInt();
  });

  server.on("/B8", HTTP_GET, [] (AsyncWebServerRequest * request) {
```

```
    request->send_P(200, "text/plain", PreD7().c_str());
  });
  server.on("/pre8", HTTP_GET, [](AsyncWebServerRequest * request) {
    request->send_P(200, "text/plain", PreD8().c_str());
  });

////////////////////////////////////////////////////////////////////////////////

  server.on("/custom7", [](AsyncWebServerRequest * request) {
    request->send_P(200, "text/html", custompage);
  });

  server.on("/Return", [](AsyncWebServerRequest * request) {
    request->send_P(200, "text/html", webpage);
  });

  server.on("/WIP", [](AsyncWebServerRequest * request) {
    request->send_P(200, "text/html", WIPpage);
  });

  server.on("/cancel", [](AsyncWebServerRequest * request) {
    cancel = 1;
    Ready0 = 0;
    request->send_P(200, "text/html", webpage);
  });

  server.on("/finished", [](AsyncWebServerRequest * request) {
    cancel = 0;
    Ready0 = 0;
    WIP = false;
    request->send_P(200, "text/html", webpage);
  });

  server.on("/settings", [](AsyncWebServerRequest * request) {
    request->send_P(200, "text/html", settingspage);
  });

  server.on("/Precustom", [](AsyncWebServerRequest * request) {
    memcpy(B, PreDrink[bw], sizeof(PreDrink[bw]));
    request->send_P(200, "text/html", presetcustompage);
  });
```

```
  server.begin();
}

////////////////////////////////////////////////////////////////////////////////

void loop() {
  if (Settinga == 0 || bw == 7) {
    switch (bw) {
      case 0:
        bw = -1;
        WIP = true;
        homing();
        if (cancel == 0) {
          for (int i = 0; i < 8; i++) {
            if (PreDrink[0][i] > 0) {
              docking();
              // Alles was jetzt kommt
              scale.tare();
              while(scale.get_units() < PreDrink[0][i]){
                digitalWrite(pump, HIGH);
              }
              digitalWrite(pump, LOW);
              Xpos0();
              Multiply = 1;
            }
            else {
              Multiply++;
            }
          }
          Ready0 = 1;
        }
        else {
          cancel = 0;
        }
        WIP = false;
        break;

      case 1:
        bw = -1;
        WIP = true;
        homing();
        if (cancel == 0) {
```

```
          for (int i = 0; i < 8; i++) {
            if (PreDrink[1][i] > 0) {
              docking();
              // Alles was jetzt kommt
              scale.tare();
              while(scale.get_units() < PreDrink[1][i]){
                digitalWrite(pump, HIGH);
              }
              digitalWrite(pump, LOW);
              Xpos0();
              Multiply = 1;
            }
            else {
              Multiply++;
            }
          }
          Ready0 = 1;
        }
        else {
          cancel = 0;
        }
        WIP = false;
        break;
////////////////////////////////////////////////////////////////////////////////
      case 2:
        bw = -1;
        WIP = true;
        homing();
        if (cancel == 0) {
          for (int i = 0; i < 8; i++) {
            if (PreDrink[2][i] > 0) {
              docking();
              // Alles was jetzt kommt
              scale.tare();
              while(scale.get_units() < PreDrink[2][i]){
                digitalWrite(pump, HIGH);
              }
              digitalWrite(pump, LOW);
              Xpos0();
              Multiply = 1;
            }
            else {
```

```
            Multiply++;
          }
        }
        Ready0 = 1;
      }
      else {
        cancel = 0;
      }
      WIP = false;
      break;

    case 3:
      bw = -1;
      WIP = true;
      homing();
      if (cancel == 0) {
        for (int i = 0; i < 8; i++) {
          if (PreDrink[3][i] > 0) {
            docking();
            // Alles was jetzt kommt
            scale.tare();
            while(scale.get_units() < PreDrink[3][i]){
              digitalWrite(pump, HIGH);
            }
            digitalWrite(pump, LOW);
            Xpos0();
            Multiply = 1;
          }
          else {
            Multiply++;
          }
        }
        Ready0 = 1;
      }
      else {
        cancel = 0;
      }
      WIP = false;
      break;
////////////////////////////////////////////////////////////////////////////////
    case 4:
      bw = -1;
```

```
        WIP = true;
        homing();
        if (cancel == 0) {
          for (int i = 0; i < 8; i++) {
            if (PreDrink[4][i] > 0) {
              docking();
              // Alles was jetzt kommt
              scale.tare();
              while(scale.get_units() < PreDrink[4][i]){
                digitalWrite(pump, HIGH);
              }
              digitalWrite(pump, LOW);
              Xpos0();
              Multiply = 1;
            }
            else {
              Multiply++;
            }
          }
          Ready0 = 1;
        }
        else {
          cancel = 0;
        }
        WIP = false;
        break;
////////////////////////////////////////////////////////////////////////////////
      case 5:
        bw = -1;
        WIP = true;
        homing();
        if (cancel == 0) {
          for (int i = 0; i < 8; i++) {
            if (PreDrink[5][i] > 0) {
              docking();
              // Alles was jetzt kommt
              scale.tare();
              while(scale.get_units() < PreDrink[5][i]){
                digitalWrite(pump, HIGH);
              }
              digitalWrite(pump, LOW);
              Xpos0();
```

```
            Multiply = 1;
          }
          else {
            Multiply++;
          }
        }
        Ready0 = 1;
      }
      else {
        cancel = 0;
      }
      WIP = false;
      break;
////////////////////////////////////////////////////////////////////////////////
    case 6:
      bw = -1;
      WIP = true;
      homing();
      if (cancel == 0) {
        for (int i = 0; i < 8; i++) {
          if (PreDrink[6][i] > 0) {
            docking();
            // Alles was jetzt kommt
            scale.tare();
            while(scale.get_units() < PreDrink[6][i]){
              digitalWrite(pump, HIGH);
            }
            digitalWrite(pump, LOW);
            Xpos0();
            Multiply = 1;
          }
          else {
            Multiply++;
          }
        }
        Ready0 = 1;
      }
      else {
        cancel = 0;
      }
      WIP = false;
      break;
```

```
      case 7:
        bw = -1;
        WIP = true;
        homing();
        if (cancel == 0) {
          for (int i = 0; i < 8; i++) {
            if (B[i] > 0) {
              docking();
              // Alles was jetzt kommt
              scale.tare();
              while(scale.get_units() < B[i]){
                digitalWrite(pump, HIGH);
              }
              digitalWrite(pump, LOW);
              Xpos0();
              Multiply = 1;
            }
            else {
              Multiply++;
            }
          }
          memset(B, 0, sizeof(B));
          Ready0 = 1;
        }
        else {
          memset(B, 0, sizeof(B));
          cancel = 0;
        }
        WIP = false;
        break;
////////////////////////////////////////////////////////////////////////////////
      case 8:
        bw = -1;
        WIP = true;
        homing();
        if (cancel == 0) {
          for (int i = 0; i < 8; i++) {
            if (PreDrink[8][i] > 0) {
              docking();
              // Alles was jetzt kommt
              scale.tare();
```

```
            while(scale.get_units() < PreDrink[8][i]){
              digitalWrite(pump, HIGH);
            }
            digitalWrite(pump, LOW);
            Xpos0();
            Multiply = 1;
          }
          else {
            Multiply++;
          }
        }
        Ready0 = 1;
      }
      else {
        cancel = 0;
      }
      WIP = false;
      break;
    }
  }
}
```