1. Die Software kann wahlweise für das Linux-Betriebssystem des Raspberry Pi programmiert oder mit Hilfe eines Bare-Metal Systems implementiert werden.
   1.1. Wie sieht der Boot-Vorgang des Raspberry Pis nach dem Anlegen der Versorgungsspannung aus? (7P)
      - Power on
      - GPU startet (Programmcode im ROM) / CPU ist aus
      - GPU lädt weiteren Code von der SD-Karte (bootcode.bin, FAT32 boot-Partition)
      - GPU lädt weiteren Code (Firmware) von SD-Karte (start.elf)
      - GPU liest config.txt und lädt kernel.bin an festgelegte Adresse im RAM
      - CPU führt Code aus RAM aus (Standardadresse 0x8000)
   1.2. Wie wird ein Bare-Metal-System für den Raspberry Pi erzeugt? (3P)
      - erste Partition auf SD Karte FAT32 formatieren
      - benötigte Binärdateien auf erster Partion speichern -> bootloader.bin, start.elf, fixup.dat, kernel.img, config.txt
   1.3. Was sind die Unterschiede zum „normalen“ Betrieb? Muss bei der Programmierung auf etwas geachtet werden? (6P)
      - kein Debugger
      - keine Abstraktionsebene und dadurch ist Bare-Metal deutlich performanter
      - Im normalen Betrieb kann es passieren, dass das Betriebssystem ein Register überschreibt, was dann im weiteren Programmablauf zu Fehlern führen kann
      - volle Kontrolle und direkter Hardwarezugriff mit Bare-Metal
      - Nur Assembler oder C
2. Systemarchitektur / Dokumentation
   2.1. Für welche Systembasis entscheidet sich die Gruppe? (6P)
      - Bare-Metal-System oder Linux-Basis?
        Linux!
      - Die Gruppe hat sich für ein Linux-basiertes System entscheiden, da aus Performancegründen ein Bare-Metal System nicht notwendig ist und wir nicht auf die Debugger-Funktionalität verzichten wollen, durch unsere geringe Assembler Vorerfahrung gehen wir auch davon aus einige Fehler zu machen, welche mit einem Debugger einfacher gefunden und behoben werden können. Die WiringPi Bibliothek kann uns zusätzlich weitere Arbeit abnehmen.

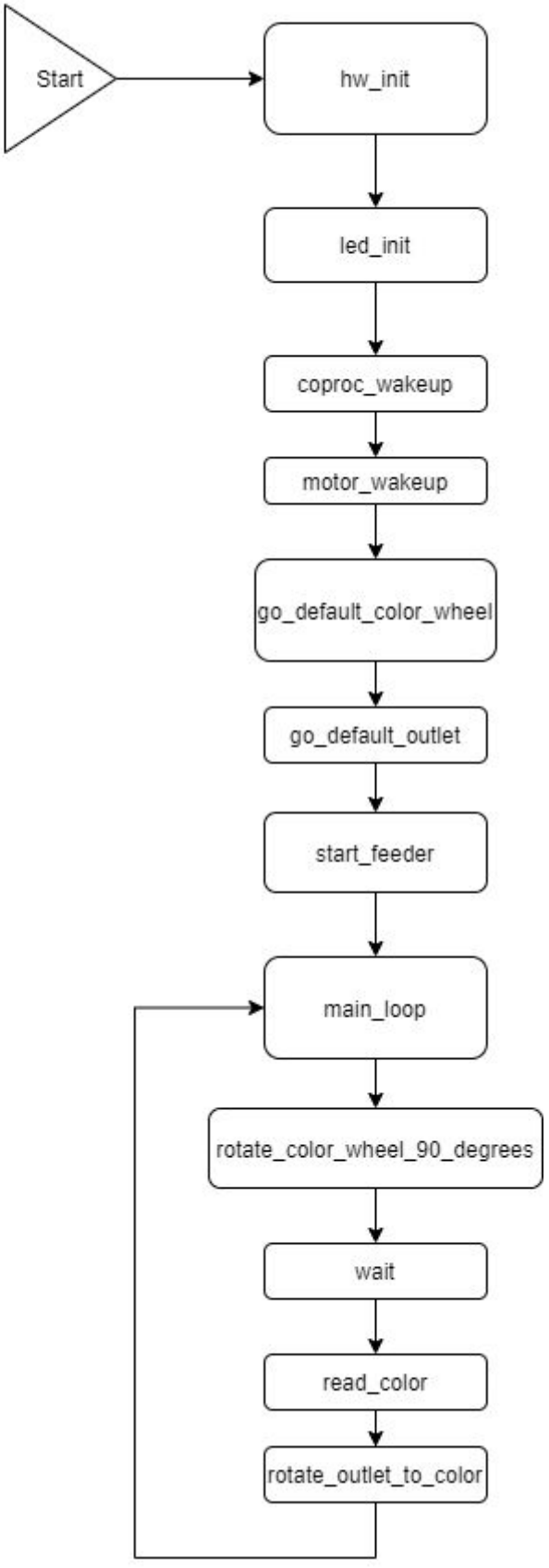
   2.2. Wie ist die Software aufgebaut? Was passiert wann? Wie spielen die Komponenten zusammen? (12P)

      Das Programm liest die Hall-Sensoren an Color Wheel und Outlet aus um die Grundpositionen, der beiden Motoren (Color Wheel & Outlet) anzufahren. Von dort aus kann über einzelne Steps jede gewünschte

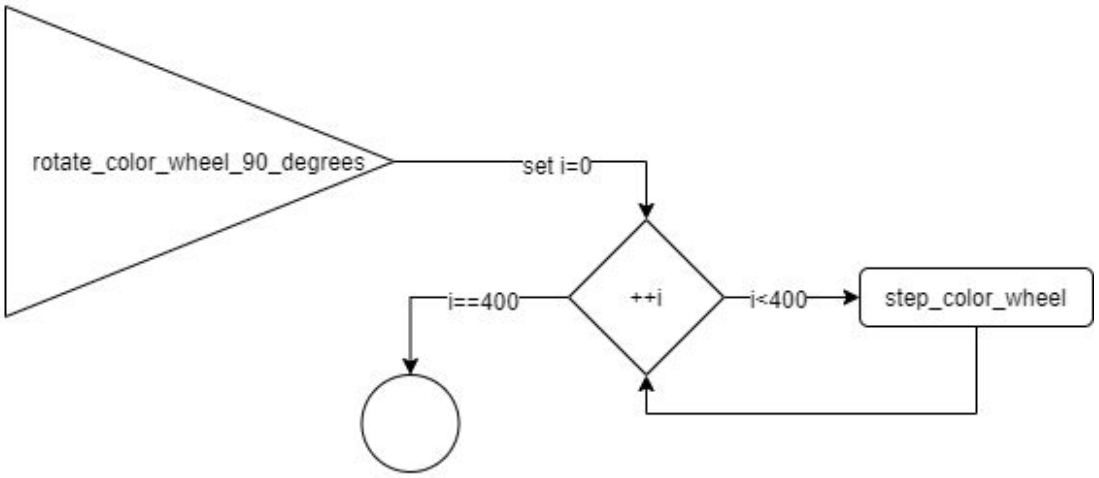
position angefahren werden. Der Feeder dreht sich, damit neue M&M’s ins Color Wheel fallen können. Dieses dreht dann eine viertel Umdrehung, damit der Farbsensor die Farbe bestimmen kann und gleichzeitig ein neues M&M nachkommt. Das Outlet fährt dann die richtige Position an und das Color Wheel dreht eine viertel Umdrehung weiter, sodass wieder ein neues M&M ausgelesen werden kann, ein neues hineinfällt und das ausgelesene durch das Outlet in die richtige Schüssel fällt.

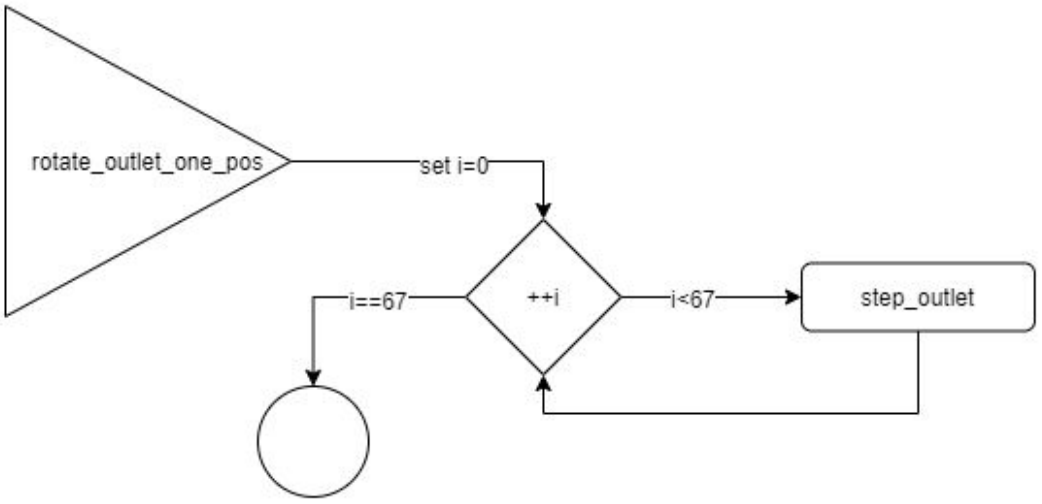
t

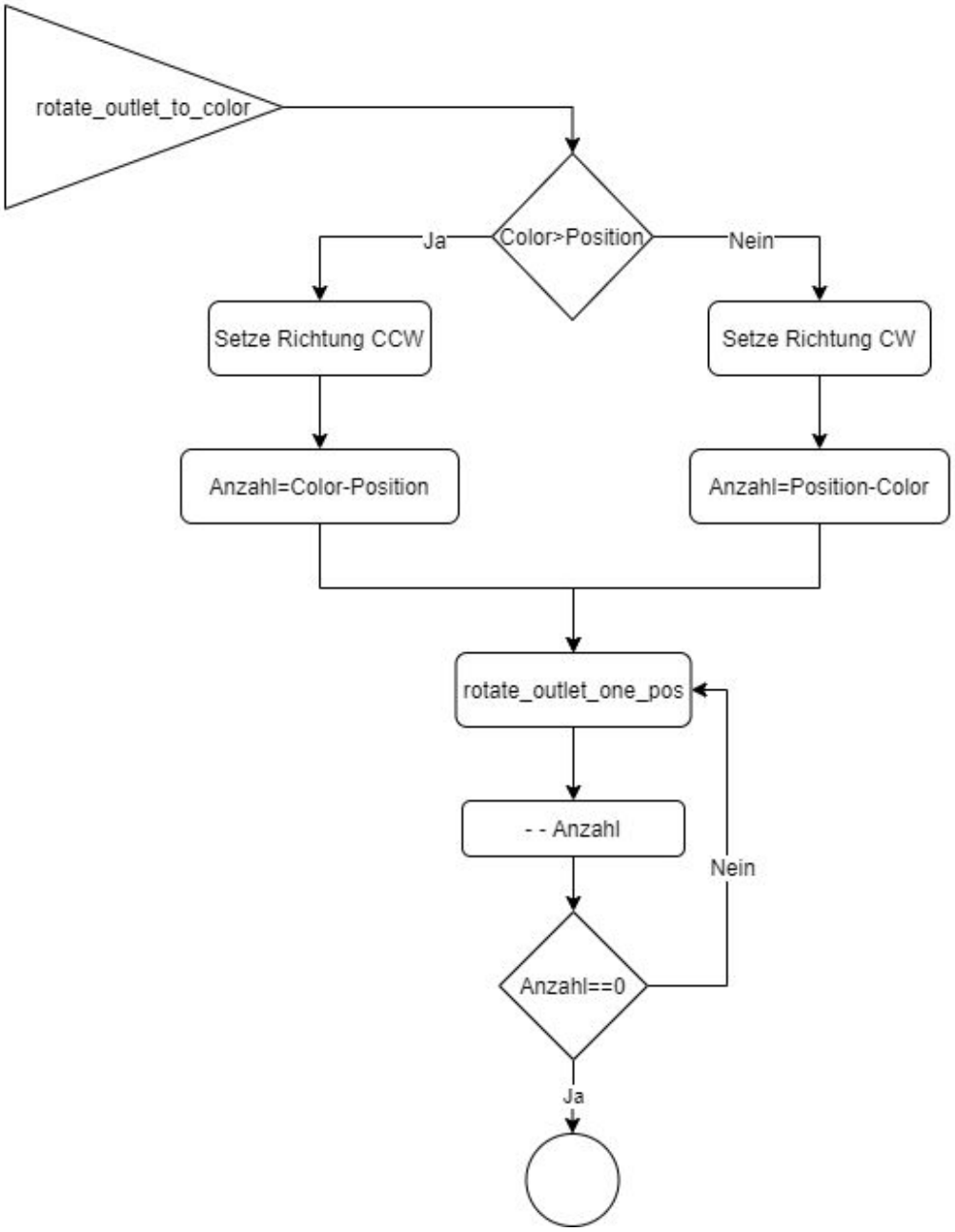
Funktionsablauf:



Funktionen:


rotate_color_wheel_90_degrees
set i=0
++i
i==400
i<400
step_color_wheel


rotate_outlet_one_pos
set i=0
++i
i==67
i<67
step_outlet


rotate_outlet_to_color
Color>Position
Ja
Nein
Setze Richtung CCW
Setze Richtung CW
Anzahl=Color-Position
Anzahl=Position-Color
rotate_outlet_one_pos
- - Anzahl
Nein
Anzahl==0
Ja

2.3. Kommentare im Code (6P)

- Jede Funktion muss kurz beschrieben werden. Was ist die Aufgabe der Funktion? Welche Eingangssignale/-variablen werden erwartet? Resultiert eine Ausgabe zur Weiterverarbeitung?
- Codezeilen werden bitte durch Eure Gedanken und den Sinn / Zweck kommentiert. Vermeidet bitte Kommentar wie z.B.: add r1,#1 @ addiere 1 zu r1 … besser: „Warum tue ich XY hier?“