

# Aufgabe 1

## Aufgabe 1.1 Boot-Vorgang

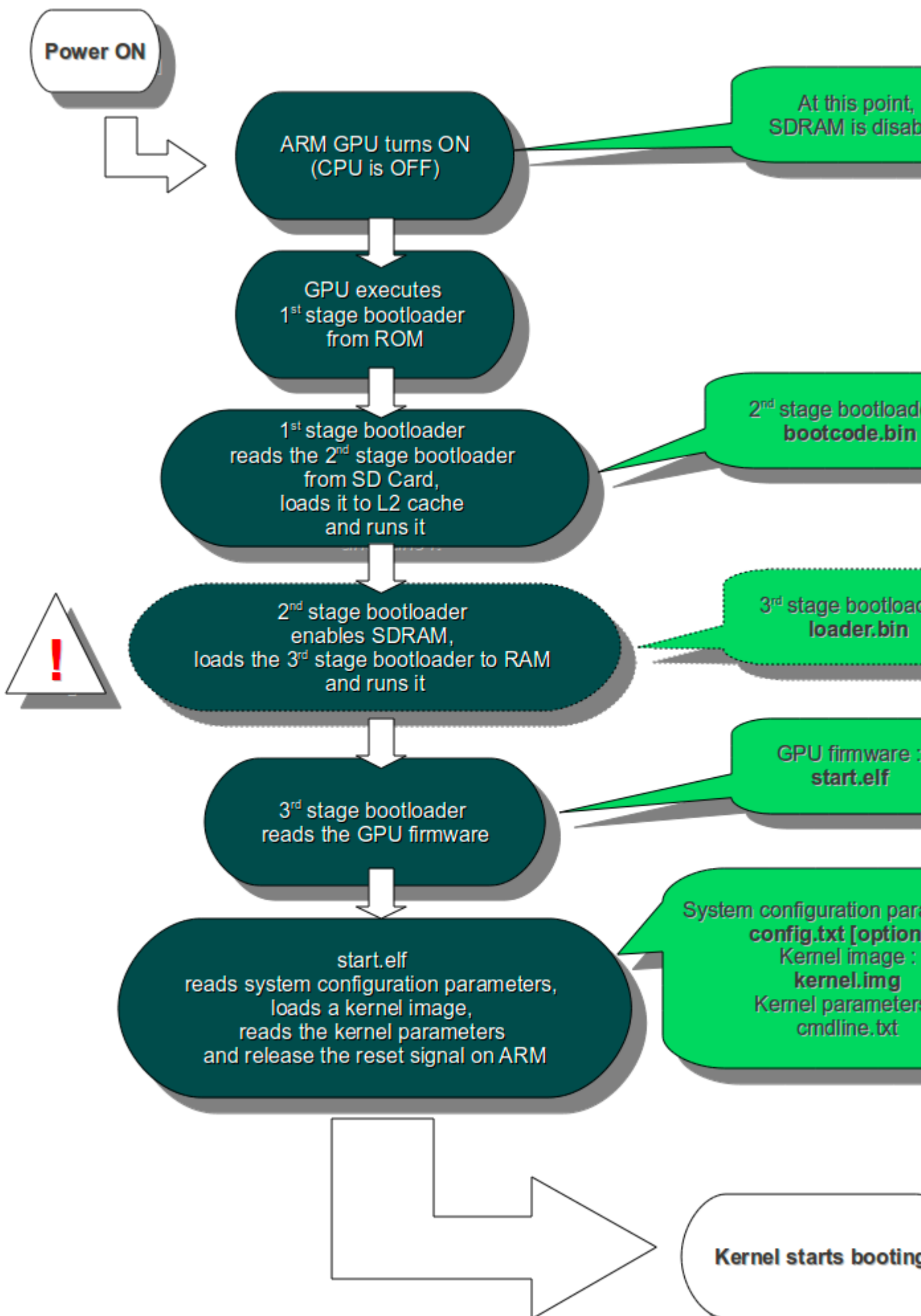
Nach Anlegen der Spannung wird der OTP(one-time programmable)-Block gelesen, um zu ermitteln, welcher Boot Mode aktiviert ist. (Beim BCM2835 wird vorher erst der SD-Card boot versucht, danach wird USB Device Boot versucht) Je nach boot mode wird GPIO PIN 48-53(Seite 102 ARM Peripherals Datasheet), primary SD, secondary SD, NAND, SPI oder USB nach der bootcode.bin gesucht. Bei den SD Karten wird nach einem FAIL ein 5 sekündiger Timeout eingeleitet.

Der Bootvorgang läuft in (grob) 5 Stages ab:

- Stage 1: Check Boot Mode + Stage 2 in L2 Cache laden
- Stage 2: Bootcode.bin ausführen --> SDRAM aktiviert + Stage 2 in L2 Cache laden
- Stage 3: loader.bin ausführen --> .elf Format bekannt + start.elf wird geladen
- Stage 4: start.elf ausführen --> config.txt, cmdline.txt und bcm2835.dtb werden gelesen (dtb Datei wird in 0x100 geladen und kernel in 0x8000) + kernel.img wird geladen
- Stage 5: kernel.img wird auf den ARM geladen und dort ausgeführt

*Bis kernel.img auf dem ARM ausgeführt wird, läuft alles auf der GPU!*

Das folgende Diagramm liefert einen detaillierteren Einblick:



## Aufgabe 1.2 Wie wird ein Bare-Metal-System für den Raspberry Pi erzeugt?

- **Cross-Compiler aussuchen und aufsetzen:** Es gibt mehrere verschiedene Compiler, die eingesetzt werden können. Wichtig ist hier die richtige Konfiguration entsprechend der Hardware.
- **RPIO Compiler Flags:**
- **Exit-Funktion für den Linker definieren:**
- **kernel.elf Datei erzeugen:** Diese Datei wird beim booten gelesen und entsprechend wird der Code gestartet.
- **GPIO Controller aufsetzen:**
- **kernel.img erzeugen:** Das Programm wird Mithilfe des Cross-Compilers kompiliert, im Normalfall wird eine IMG-Datei erzeugt.
- **kernel.img auf SD Karte laden:** Sehr kompliziert, würde den Rahmen dieser Dokumentation sprengen.
- **booten:** Strom anschalten und Daumen drücken

## Aufgabe 1.3 Unterschiede zum normalen Betrieb? + Besonderheiten bei der Programmierung

Im Bare-Metal-Betrieb ist ein uneingeschränkter Zugriff auf alle Register des SoC, wie Timer, GPIO-datalanes, etc. möglich. Es sind folglich keine virtuellen, Betriebssystem-Abhängigen Adressen nötig. Hierdurch wird die Programmierung tendentiell einfacher, da direkt mit den Werten, die im Chip-Handbuch genannt werden gearbeitet werden kann. Im Kernel-Betrieb ist es nötig, die Adressen mit einem Offset zu versehen. Hierdurch ist es umdenken notwendig, welches aber durch einen zu beginn definierten Offset behoben werden kann. Dennoch ist es theoretisch möglich den Speicherbereich des Kernels zu überschreiben und so Fehlermeldungen und Interrupts verursachen. Würde man rechenaufwendige Prozeduren in Bare-Metal programmieren wären diese sehr viel schneller, da alle Systemressourcen verwendet werden können, während man beim Betriebssystem Switching betreiben muss.

## Aufgabe 2

### Aufgabe 2.1

Wir haben uns für die Programmierung auf Linux-Basis entschieden. Gründe dafür waren:

#### - Debugging:

Aufgrund des bei Bare-Metal fehlendes Kernels ist es nicht möglich in irgendeiner Weise von außerhalb auf den im Moment ausgeführten Assembler-Code zuzugreifen. Dadurch ist ein Debugging mit einem

Programm von außerhalb nicht möglich. Man müsste sich mit anderen Methoden behelfen. Bei "normalen" Programmieren läuft während der Ausführung des Assembler-Codes der kernel mit. Hierdurch ist ein klassisches Debugging mit Eclipse möglich.

### **- Console:**

Aufgrund des Dateisystems, welche durch den Kernel aufrufbar sind, ist es möglich auch deren Methoden zu verwenden. So ist es beispielsweise möglich, durch den printf-befehl (C-Befehl) in den Output der angeschlossenen Konsole (bei uns der Eclipse-Debugger zu schreiben).

### **- Dateisystem:**

Im Prinzip ist dieses Argument das selbe wie bereits erwähnt: Es ist möglich auf die Funktionen zuzugreifen, welche im File-System des Kernels liegen.

### **- Multitasking:**

Eine Kombination aus der WLAN-Benutzung und Debugging. Durch einfacheres Debugging und einfacheres Laden des Programmes kann man bereits während der Ausführung Änderungen einpflegen und schnell deployen.

### **- WLAN (Remote execution):**

Durch den Wegfall der SD-Karte kann man Änderungen am Code schneller einpflegen und so Zeit beim Debuggen sparen. Natürlich bietet das Benutzen der WLAN-Schnittstelle auch einige Schwierigkeiten, die allerdings überschaubar sind.

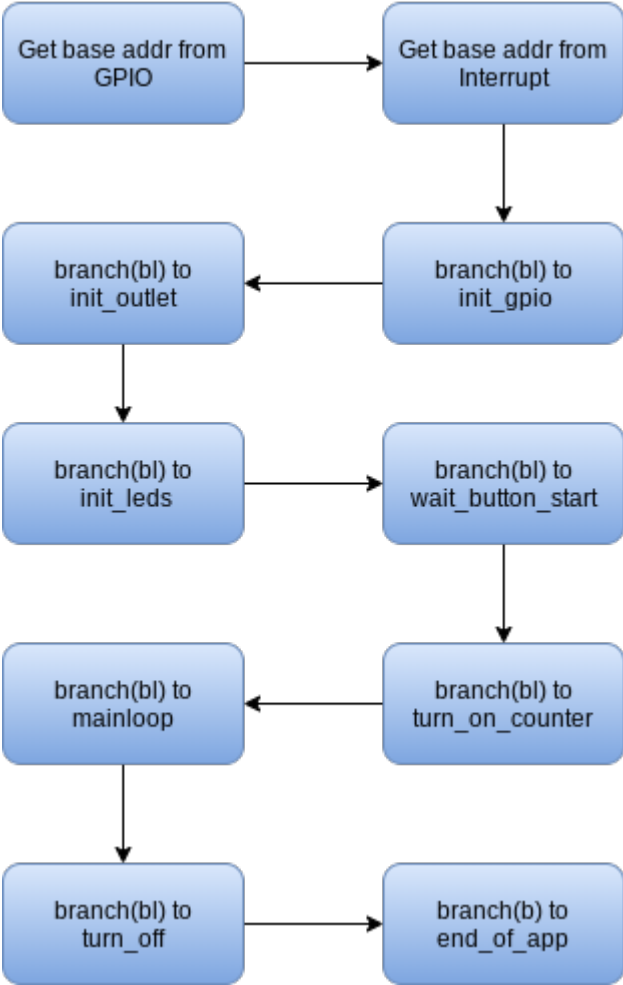
## **Aufgabe 2.2**

### **Grundsätzlicher Programmablauf / Bedienung**

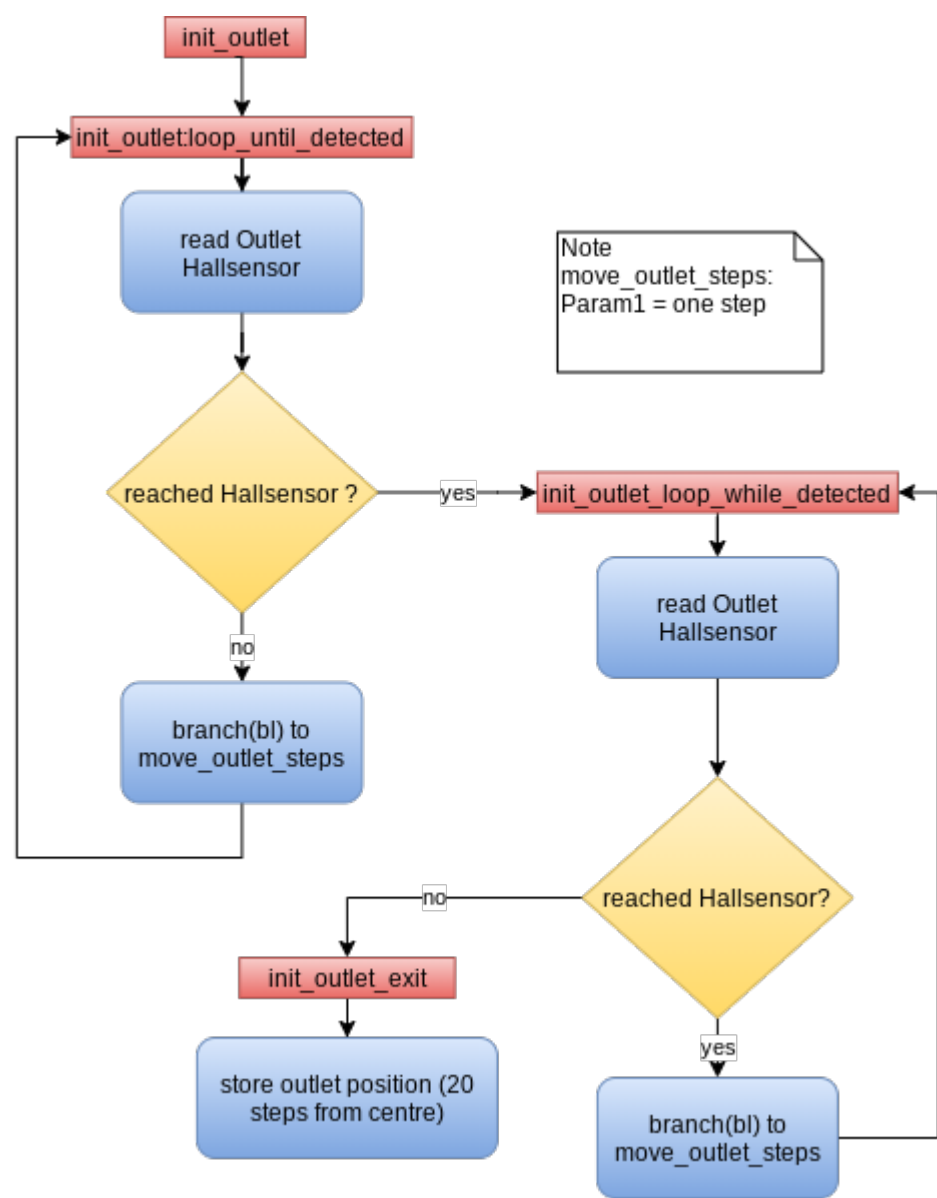
Nach dem Programmstart bewegt sich das Outlet in eine bekannte Position. Anschließend wird gewartet bis der rechte Taster gedrückt wird. Der Druck des Tasters startet der Sortiervorgang. Das Programm kann nach jedem Sortieren ordentlich beim Ende eines jeden Sortiervorgangs durch Tasterdruck beendet werden. Dazu muss der Taster betätigt werden, sobald sich das Colourwheel beim Auswurf des sortierten M&Ms aufhört zu drehen. Ein Sortiervorgang hat den folgenden Ablauf: Der Feeder wird aktiviert und das Colourwheel jeweils um 90° gedreht, solange der Farbsensor keine Farbe erkennt. Sobald eine Farbe erkannt wird, wird nur die LED mit der entsprechenden Farbe eingeschaltet, sowie das Outlet an die entsprechende Position bewegt. Anschließend wird das Colourwheel um eine Viertelumdrehung gedreht, um das M&M auszuwerfen.

# Programmablauf der Funktion hw\_init

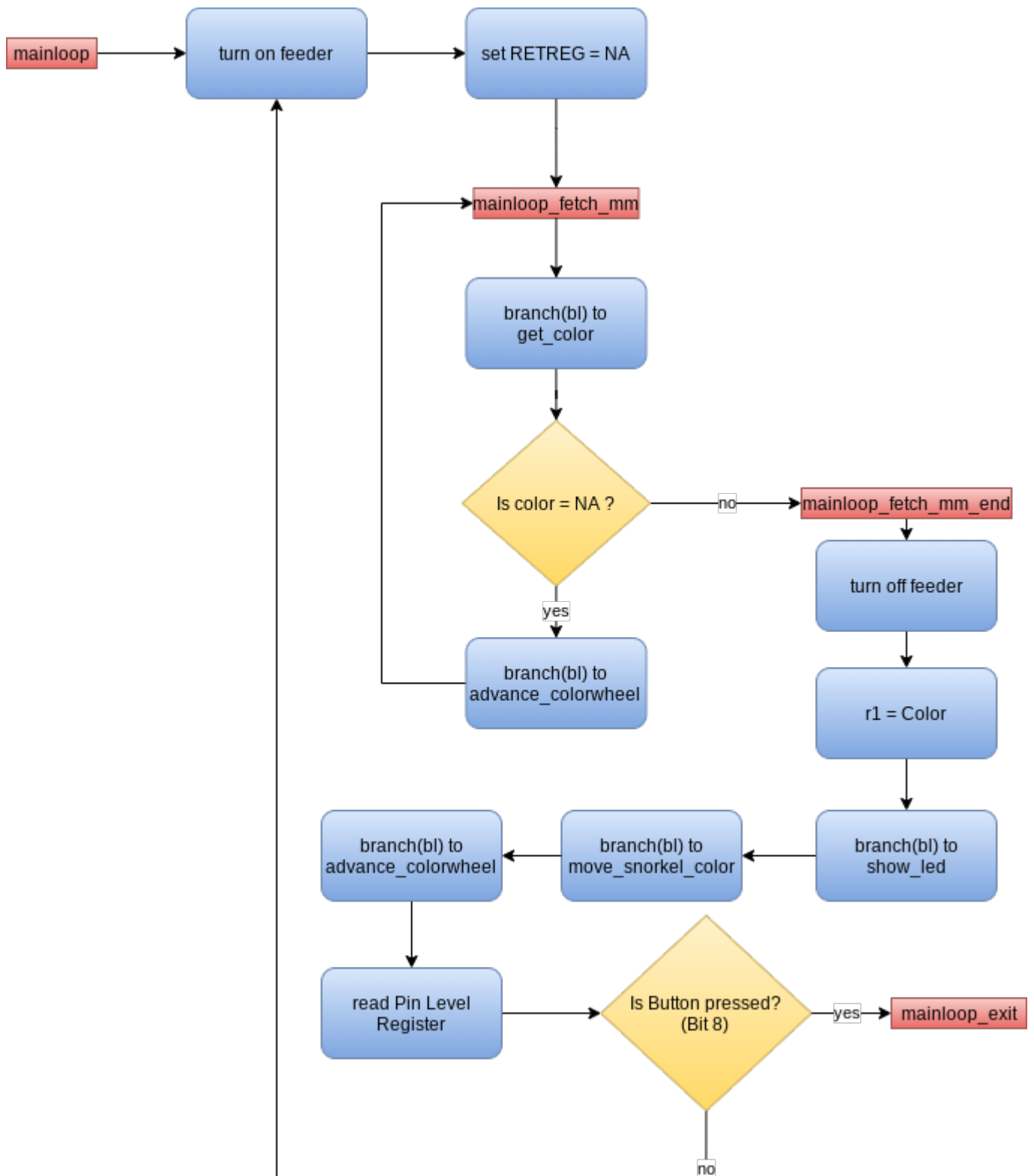
hw\_init:



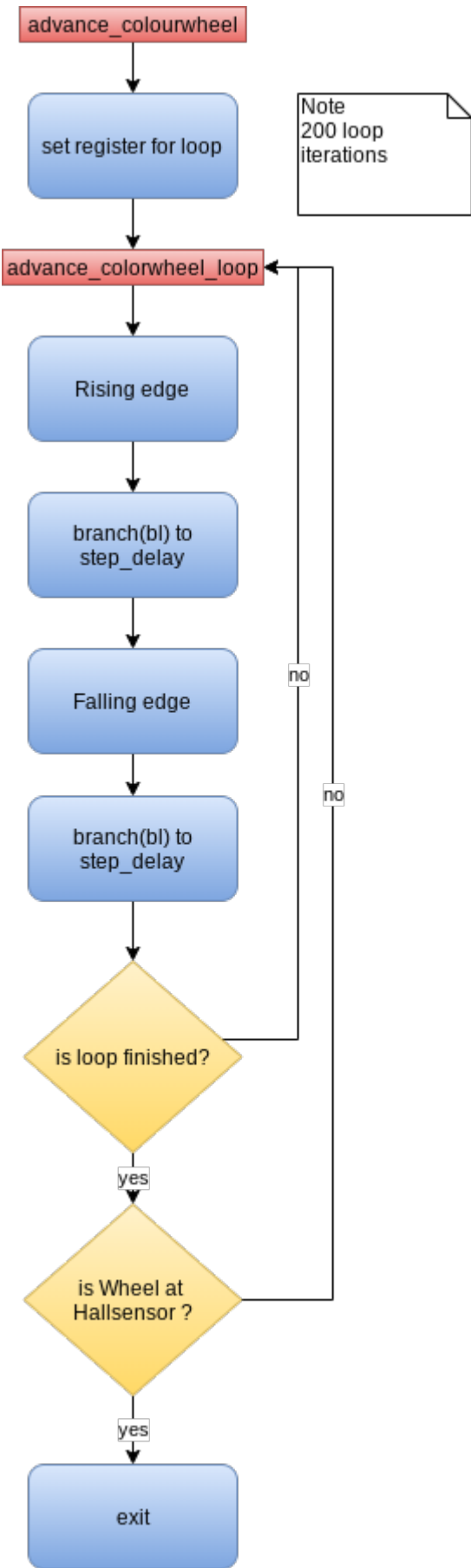
# Programmablauf der Funktion hwinitlet



## Programmablauf der Funktion mainloop

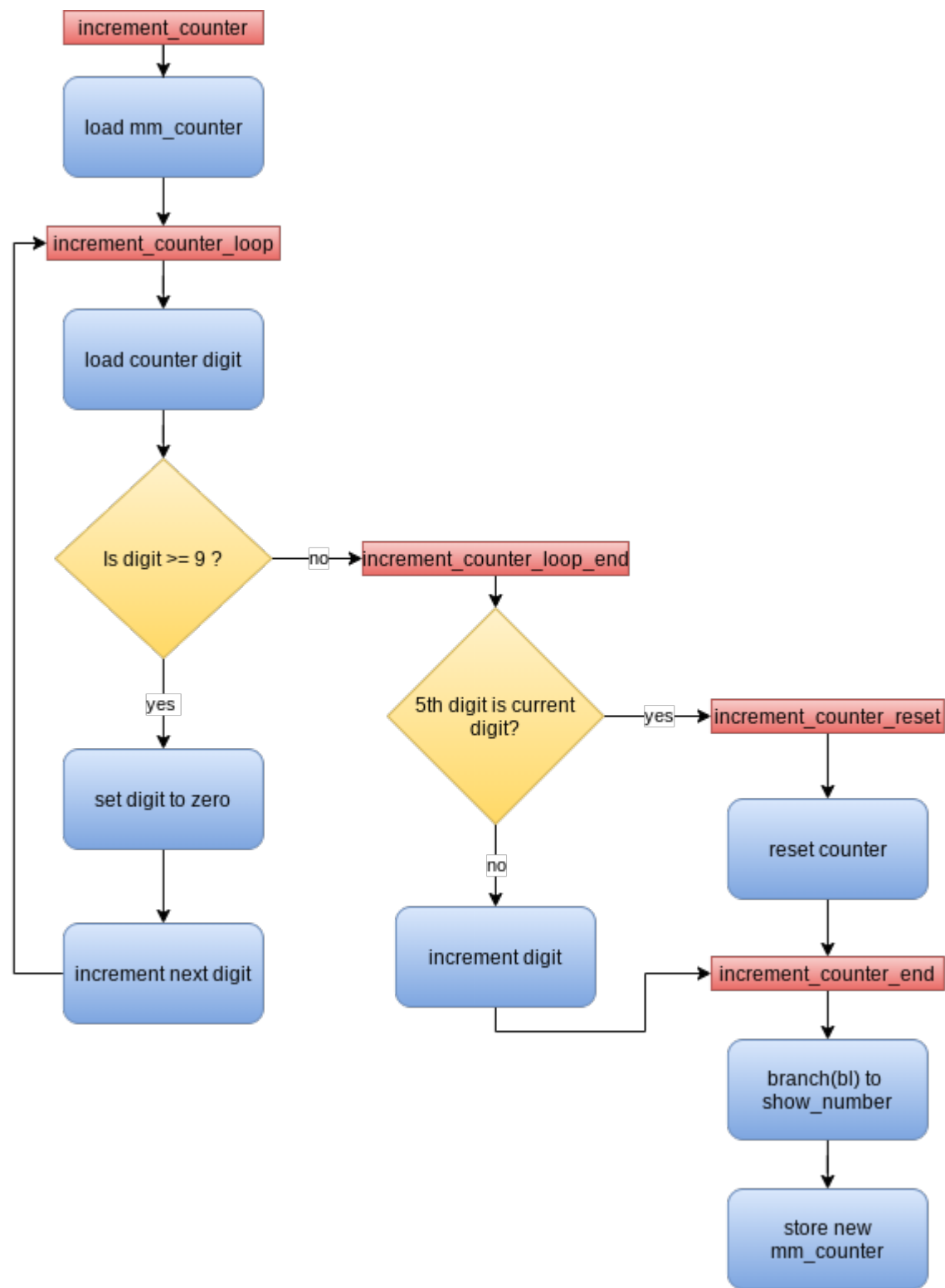


# Programmablauf der Funktion advance\_colorwheel

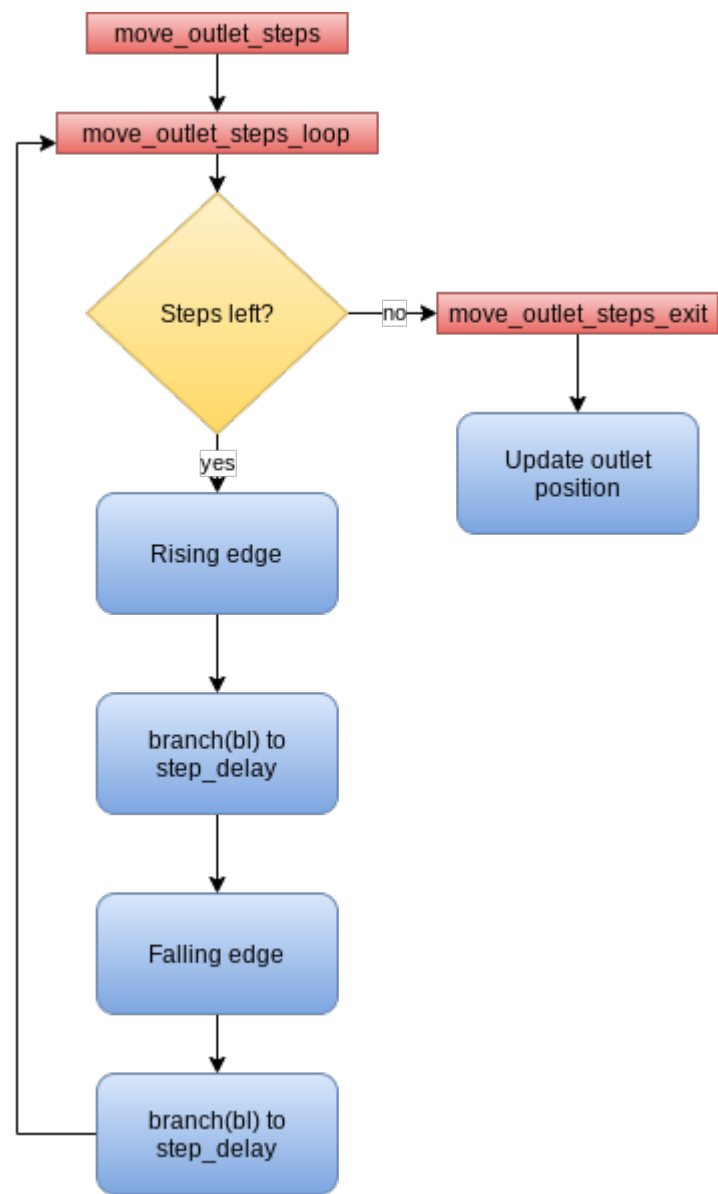




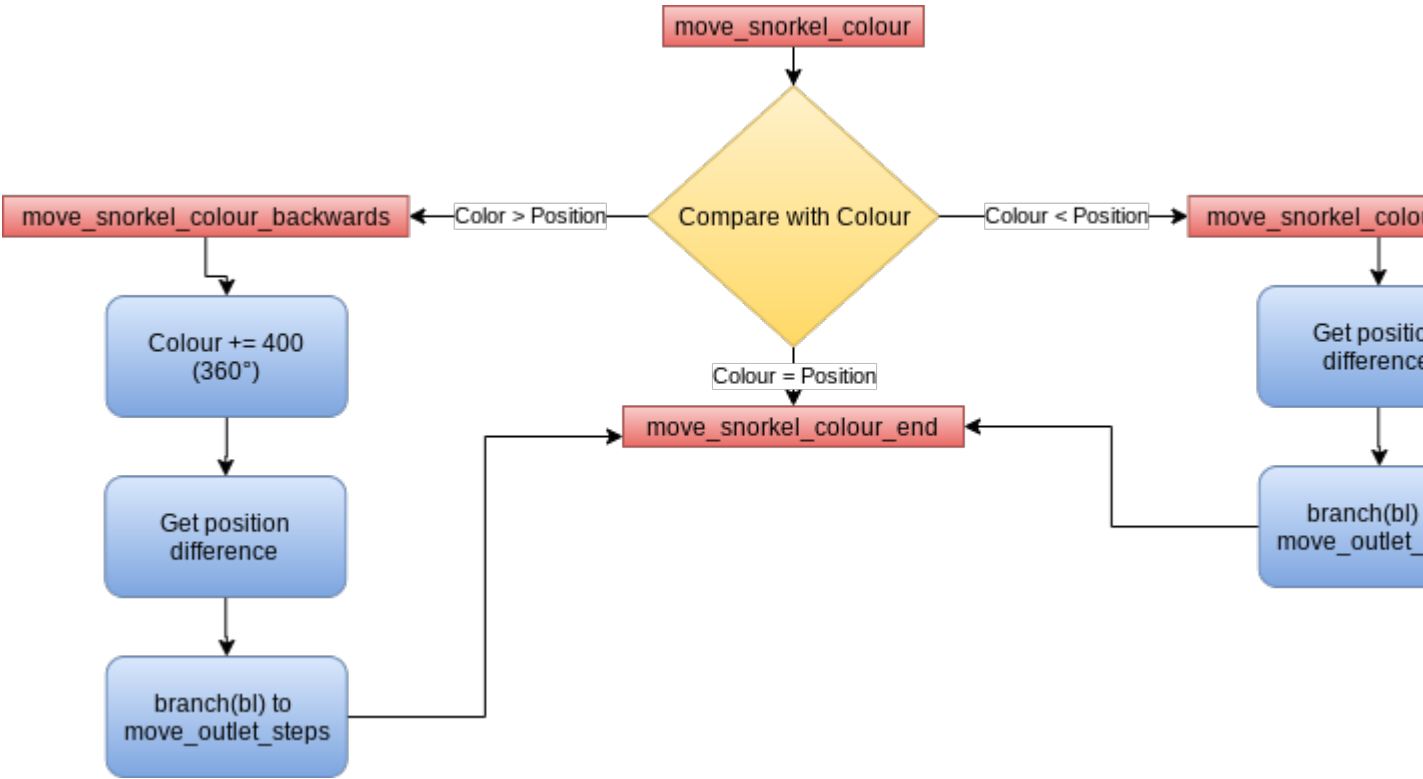
# Programmablauf der Funktion increment\_counter



**Programmablauf der Funktion *moveoutletsteps***



**Programmablauf der Funktion *movesnorkelcolor***



## Programmablauf der Funktion print\_digit

