Aufgabe 1

Aufgabe 1.1 Boot-Vorgang

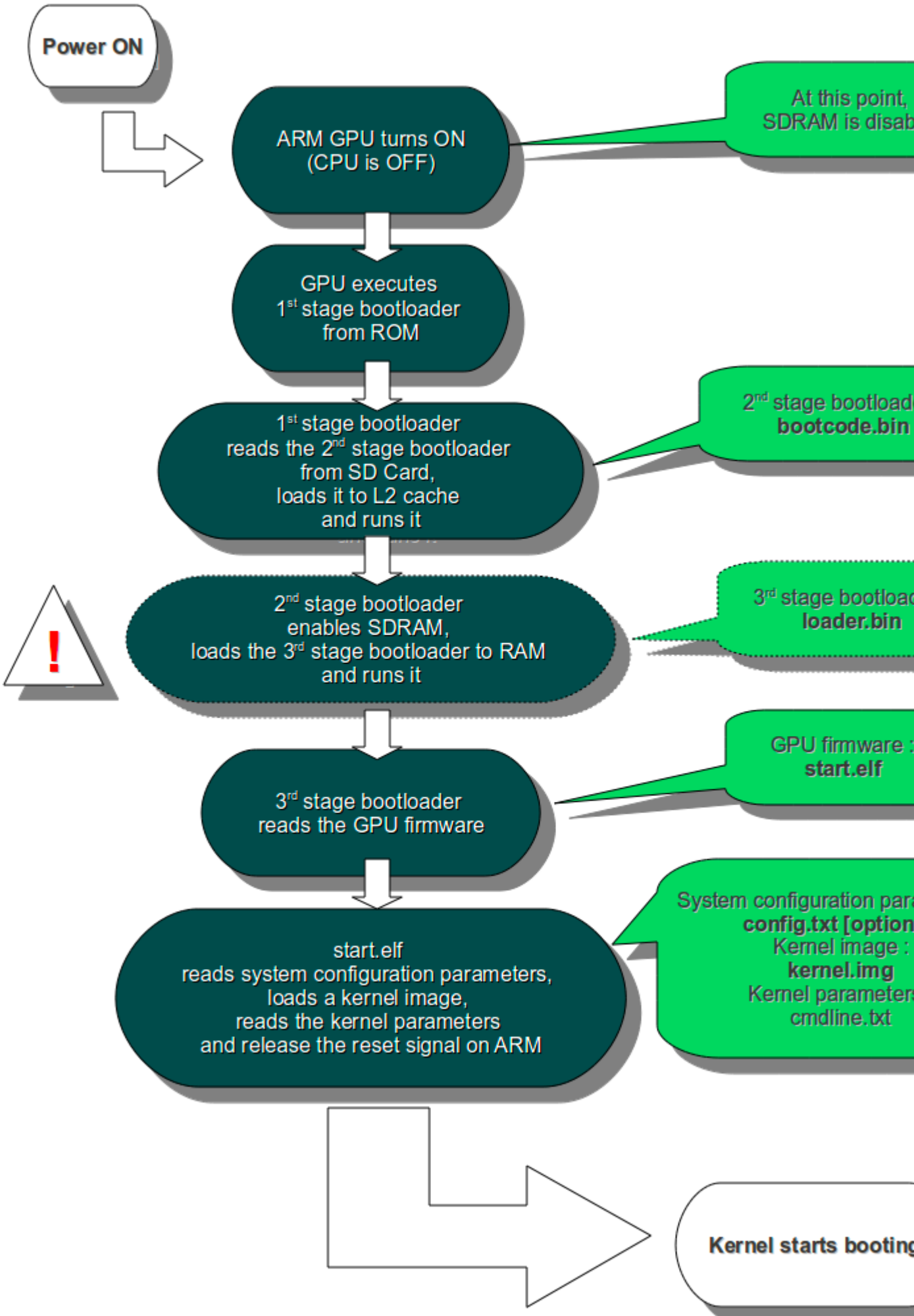
Nach Anlegen der Spannung wird der OTP(one-time programmable)-Block gelesen, um zu ermitteln, welcher Boot Mode aktiviert ist. (Beim BCM2835 wird vorher erst der SD-Card boot versucht, danach wird USB Device Boot versucht) Je nach boot mode wird GPIO PIN 48-53(Seite 102 ARM Peripherals Datasheet), primary 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 ausfürhen --> 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:


Power ON
ARM GPU turns ON
(CPU is OFF)
At this point,
SDRAM is disab
GPU executes
1st stage bootloader
from ROM
1st stage bootloader
reads the 2nd stage bootloader
from SD Card,
loads it to L2 cache
and runs it
2nd stage bootload
bootcode.bin
2nd stage bootloader
enables SDRAM,
loads the 3rd stage bootloader to RAM
and runs it
3rd stage bootloa
loader.bin
!
3rd stage bootloader
reads the GPU firmware
GPU firmware
start.elf
start.elf
reads system configuration parameters,
loads a kernel image,
reads the kernel parameters
and release the reset signal on ARM
System configuration para
config.txt [option
Kernel image :
kernel.img
Kernel parameter
cmdline.txt
Kernel starts booting

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

- **Cross-Compiler aussuchen und aufsetzten:** Es gibt mehrere verschiedene Compiler, die eingesetzt werden können. Wichtig ist hier die richtige Konfiguration entsprechend der Hardware.
- **RPI0 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 Prgramm wird Mithilfe des Cross-Compilers compiliert, im Normallfall 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, Betriebsystem-Abhängigen Adressen nötig. Hierdurch wird die Prgrammierung 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 is ein umdenken notwendig, welches aber durch einen zu beginn definierten Offset behoben werden kann. Denoch ist es theoretisch möäglich den Speicherbereich des Kernels zu überschreiben und so Fehlermeldungen und Interrupts verursachen. Würde man rechenaufwendige Prozeduren in Bare-Metal programmieren währen 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 Weiße 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 Dateisysteme, welche durch den Kernel aufrubar 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:

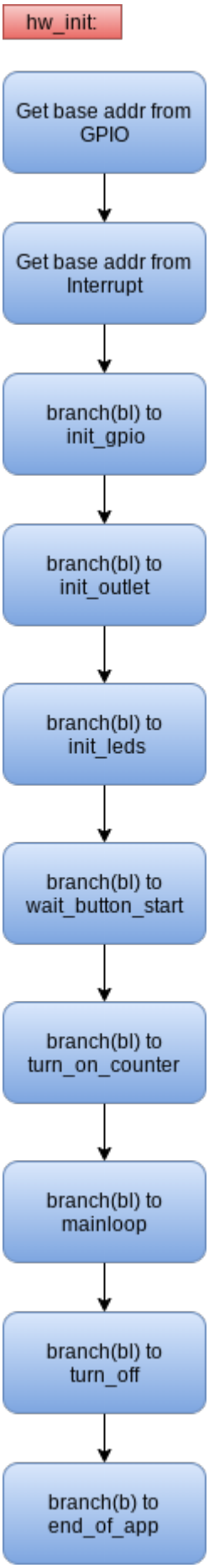
Eine Kombination aus der WLAN-Benutzung und Debugging. Durch einfacheres Debugging und einfacheres Laden des Programmes kann man bereits während der Ausführung änderungen einpflegen und schnell deployen.

- WLAN (Remote execution):

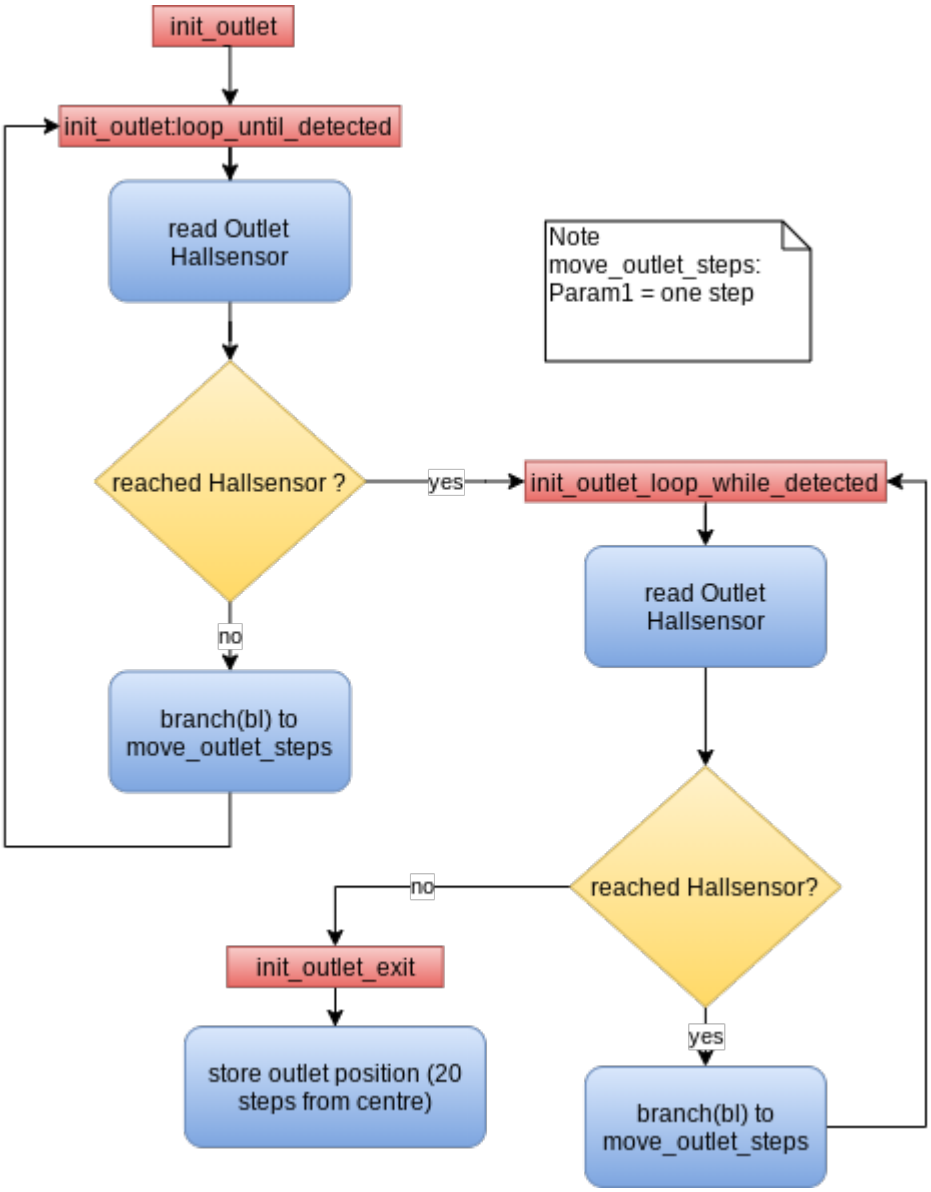
Durch den Wegfall der SD-Karte kann man Änderungen am Code schneller einpflegen und so Zeit beim Debuggen sparen. Natürlich bietet das Benutzen der WLAN-Schnittstelle auch einige Schwierigkeiten, die allerdings überschaubar sind.

Aufgabe 2.2

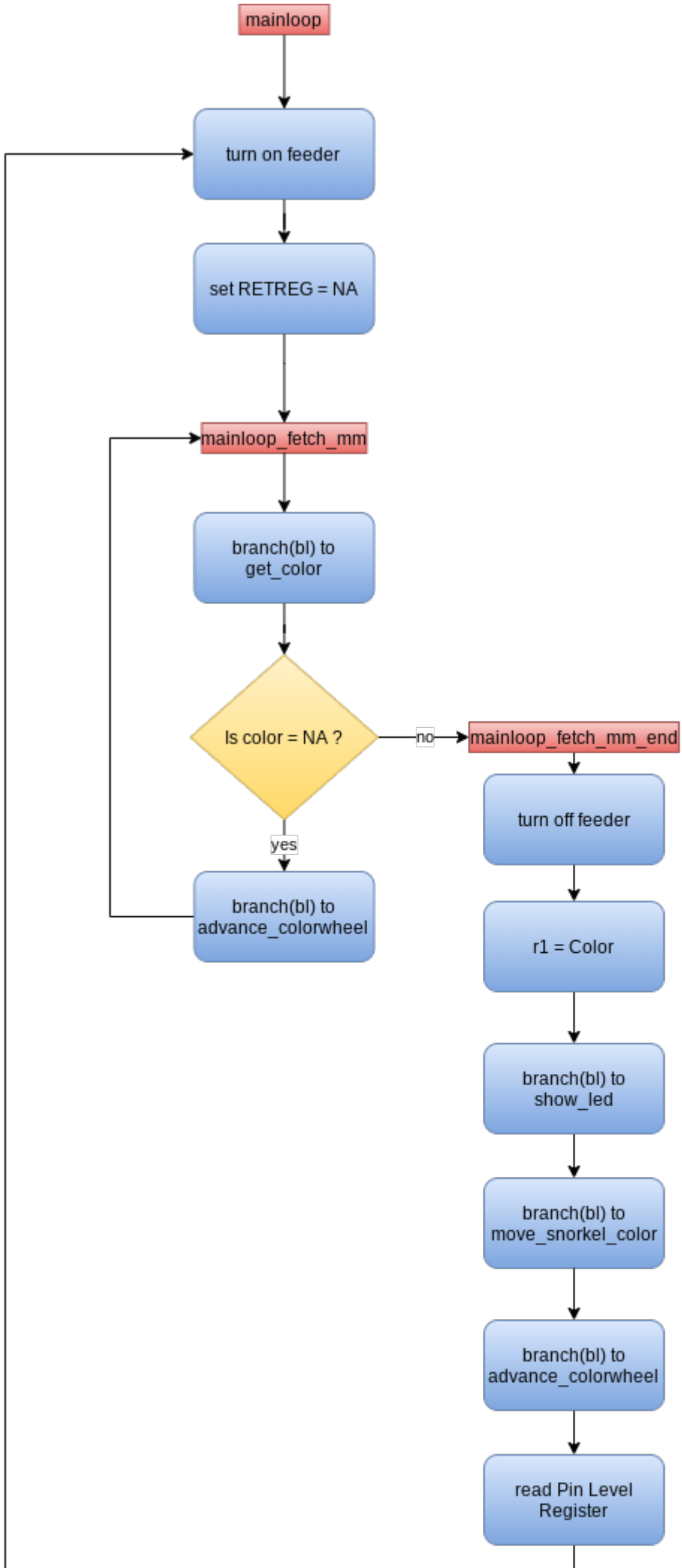
Programmablauf der Funktion hw_init


hw_init:
Get base addr from GPIO
Get base addr from Interrupt
branch(bl) to init_gpio
branch(bl) to init_outlet
branch(bl) to init_leds
branch(bl) to wait_button_start
branch(bl) to turn_on_counter
branch(bl) to mainloop
branch(bl) to turn_off
branch(b) to end_of_app

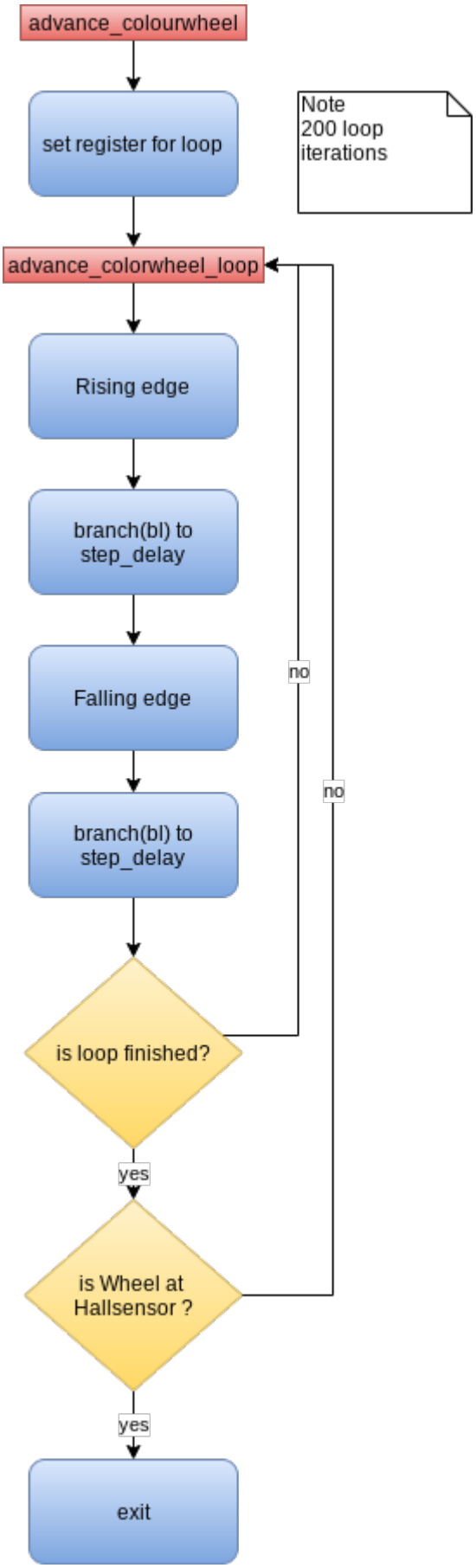
Programmablauf der Funktion hw*init*outlet



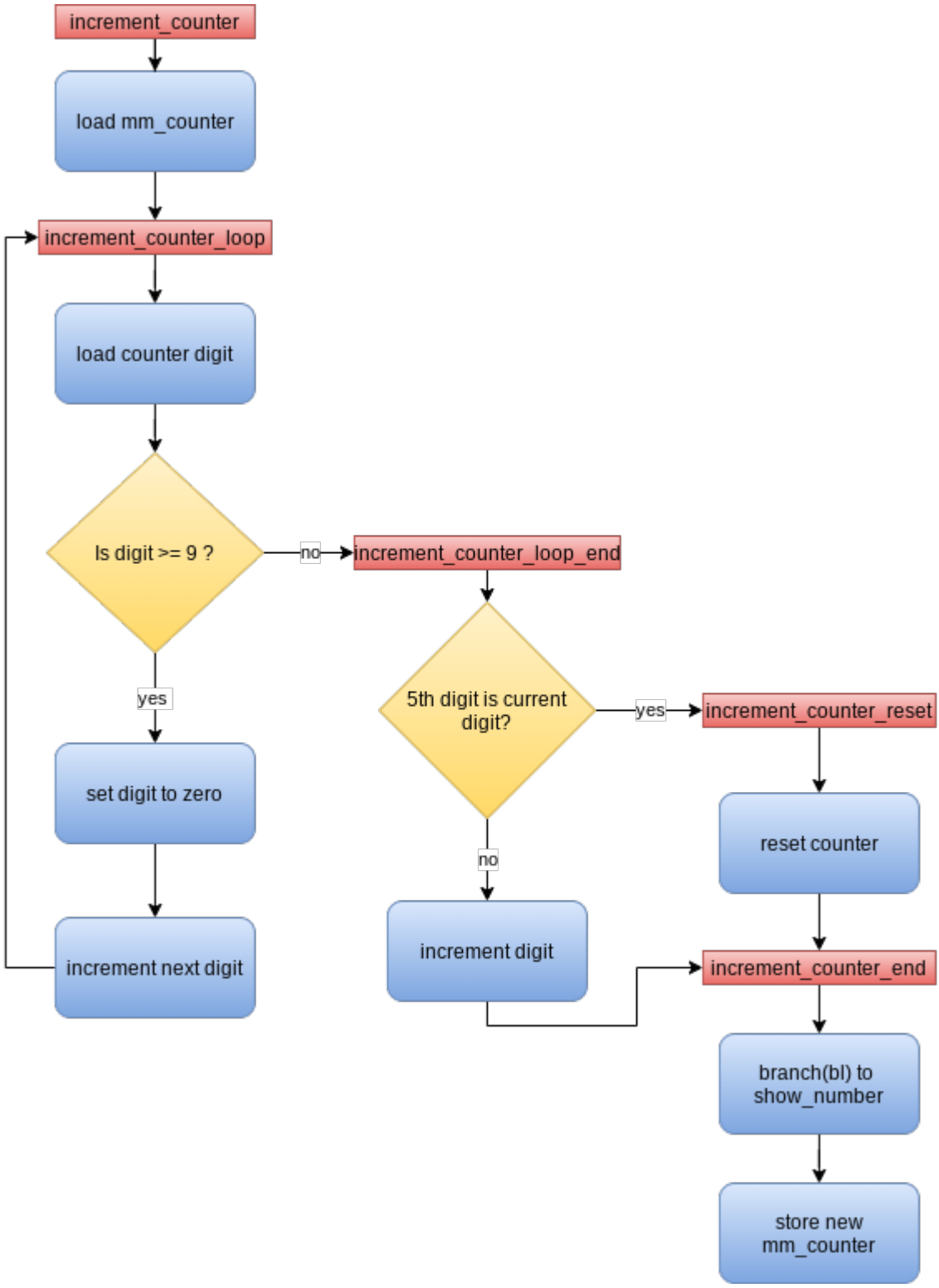
Programmablauf der Funktion mainloop


mainloop
turn on feeder
set RETREG = NA
mainloop_fetch_mm
branch(bl) to get_color
Is color = NA ?
no
mainloop_fetch_mm_end
yes
branch(bl) to advance_colorwheel
turn off feeder
r1 = Color
branch(bl) to show_led
branch(bl) to move_snorkel_color
branch(bl) to advance_colorwheel
read Pin Level Register

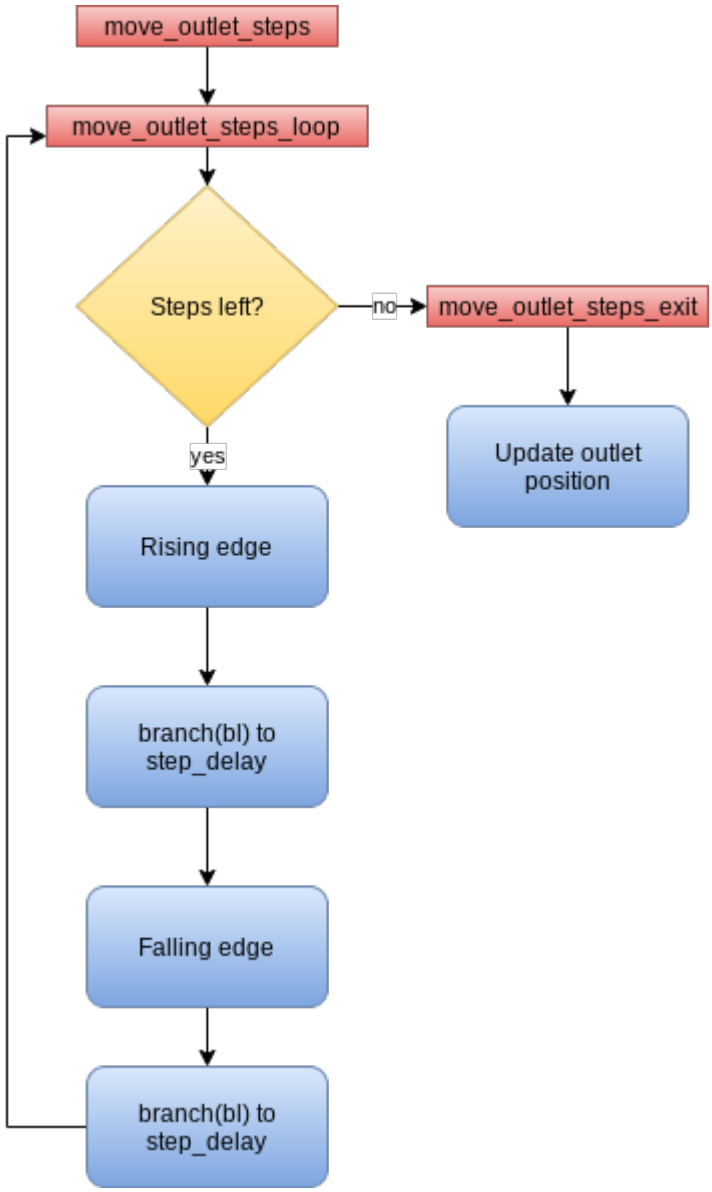
Programmablauf der Funktion advance_colorwheel



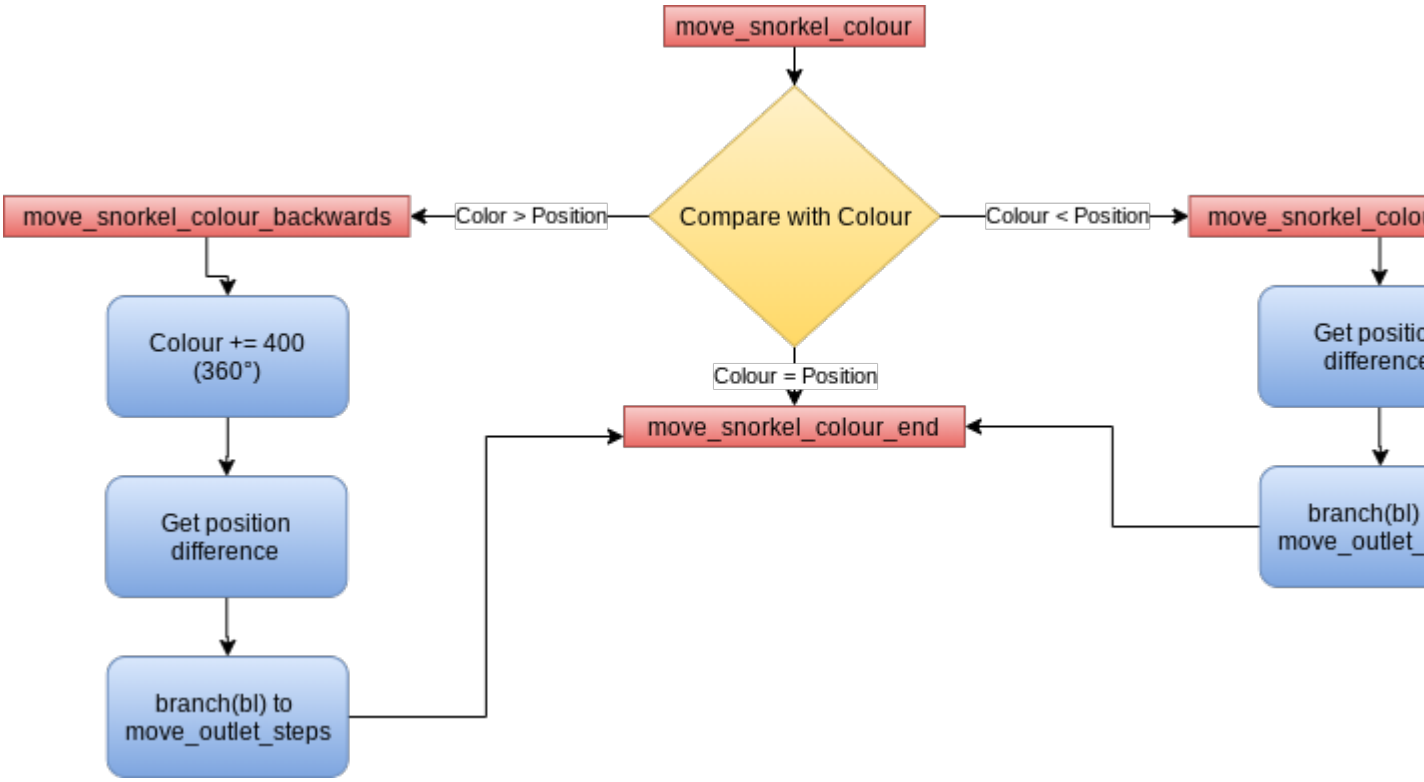
Programmablauf der Funktion increment_counter



Programmablauf der Funktion move*outlet*steps



Programmablauf der Funktion move*snorkel*color



Programmablauf der Funktion print_digit