Vorwissenschaftliche Arbeit

Entwicklung und Programmierung eines Robotik-Controllerboards basierend auf einem Raspberry Pi in C++

Vorwissenschaftliche Arbeit verfasst von

Matteo Valentini,

Klasse 8b im Schuljahr 2020/21

Betreuerin: Mag.[a] Nicole Bizjak

BRG Kepler

Keplerstraße 1

8020 Graz, Abgabe tbd.

Diese Arbeit wurde unter Zuhilfenahme des Bibliographieprogrammes `biber` erstellt und mittels des Textsatzprogrammes LaTeX gesetzt.

Die verwendete Vorlage, die auf KOMA-Script basiert, wurde von Alexander Leithner erstellt und gemäß der LaTeX Project Public License, Version 1.3, als quelloffen auf GitHub zur Verfügung gestellt.

Contents

1 Allgemeines

1.1 Abstract

1.2 Einleitung

2 Hardware

2.1 Layout und Anschlüsse

Die Platine hat eine Größe von 69 * 85.5mm und zwei Kupferschichten mit einer Stärke von 1 oz.

1 oz Kupfer ist definiert als 28.35g Kupfer verteilt über eine Fläche von 929.03cm^2. Das Resultiert in einer kupferbedeckten Platine mit einer durchschnittlichen Dicke von 35 μm[1]

Auf der Platine befindet sich fünf Printklemmen, vier davon sind für die Motoren (M1 bis M4) und eine dient als Stromanschluss (BATTERY). Zusätzlich dazu sind drei bis acht Pin XH-Buchsen von J.S.T. Mfg. Co. verbaut:

- 8 IO-Buchsen (IO1 bis IO8) mit je drei Pins
- 3 Analog-Digital-Wander-Buchsen (ADC1 bis ADC3) mit je vier Pins
- 1 UART-Buchse (UART) mit fünf Pins
- 4 I2C-Buchsen (I2C) mit je fünf Pins
- 1 User Interface Buchse (USER INTERFACE) mit sechs Pins
- 4 SPI-Buchsen (SPI1 bis SPI4) mit je sieben Pins
- 1 Display-Buchse (DISPLAY) mit acht Pins

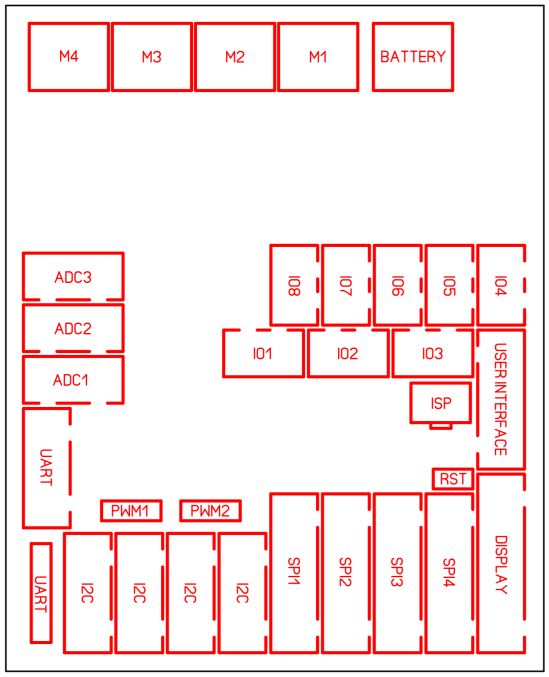


Abbildung 1: Übersicht über alle Buchsen und Stecker auf der Platine

Des weiteren sind noch zwei Male-Pin-Header mit je drei Pins für PWM (PWM), sowie ein Female-Pin-Header mit zwei Pins als Reset (RST), ein fünf-Pin für UART (UART) und einer mit sechs Pins in zwei Reihen für den Anschluss eines In-System-Programmers (ISP) verbaut. Alle dieser Pin-Header haben eine Abstand von 2.54mm zwischen den Mittelpunkten der einzelnen Pins.

[1] vgl. Spaziani, Larry: Using Copper PCB Etch for Low Value Resistance, S. 1.

2.2 Komponenten

Die wichtigste Hardware-Komponente ist ein Raspberry Pi Zero W, ein Single-Board-Computer basierend auf einem Broadcom BCM2835 SoC mit einer ARM11-Architektur. Das Raspberry Pi selbst hat eine Größe von 65 * 30mm, bei einer Höhe von circa 5mm. Dieses verfügt über einen Stecker mit 40 Pins, welche in zwei Reihen mit je 20 Pins angeordnet sind. Der Abstand zwischen den Mittelpunkten der Pins beträgt 2.54mm, was genau $\frac{1}{10}$ Zoll entspricht. 26 dieser Pins sind so genannte GPIOs, die Kurzform von **G**eneral **P**urpose **I**nput **O**utput. Jeder dieser Pins ist mit dem BCM2835 direkt verbunden und verträgt daher maximal 3.3 V und einem Strom von 10 mA. Des weiteren sind noch acht Ground-Pins sowie je zwei 3.3 V und 5 V Pins vorhanden. Die Stromversorgung des Raspberry Pi erfolgt über die zwei 5 V Pins sowie einem Ground-Pin.

Als Subprozessor dient ein ATmega 644, ein 8-Bit Mikrocontroller von Atmel/Microchip, welcher in einem TQFP-44-Package verbaut ist. Diese SMD-Bauform ist quadratisch bei einer Seitenlänge von 12mm mit je elf Pins pro Seite. 32 dieser Pins sind GPIOs, welche mit einer Spannung von 5 V arbeiten. Zusätzlich dazu existieren noch vier Ground-Pins, drei VCC-Pins, die zur Stromversorgung dienen, sowie ein AVCC- und AREF-Pin für den internen Analog-Digital-Wandler. Die übrigen drei Pins bestehen aus einem Reset-Pin, mit welchem der Chip deaktiviert werden kann, sowie zwei Pins, an die ein externer Quarz mit einer maximalen Frequenz von 20.0 MHz angeschlossen werden muss.

Die Ansteuerung der Motor-Ausgänge erfolgt über vier VNH7100ASTR, einem Motortreiber von STMicroelectronics. Der Motortreiber enthält eine zweifache Halbbrücke, was eine Steuerung des Motors in beide Richtungen ermöglicht. Die Bauform der Motortreiber ist SO-16N mit einer Größe von 9.9 * 6.0 mm und je acht Pins auf den beiden längeren Seiten. Jeder dieser kann einen Motor mit bis zu 41 V bei 12 A steuern, bei einer Betriebsspannung von maximal 38 V. Die Drehrichtung sowie die Geschwindigkeit des Motors werden über Input-Pins gesteuert, die mit dem ATmega verbunden sind.

Auf der Platine ist auch ein Bosch BNO055 verbaut, ein Gyroskop, Magnetometer und Beschleunigungssensor in einem Bauteil mit insgesamt neun Freiheitsgraden. Der BNO055 ist auf einem Breakout-Board mit 20.8 * 12.5 mm. Das Board wird mit einer Spannung von 3.3 V versorgt, die Kommunikation mit dem Raspberry Pi findet über I2C statt. Ebenfalls verfügt der Sensor über einen Interrupt-Pin, mit welchem das Raspberry Pi über neue Messdaten informiert wird.

2.3 Bussysteme

Die Kommunikation zwischen den verschiedenen aktiven Komponenten erfolgt über drei verschiedene Arten der Kommunikation:

Der ATmega 644 wird vom Raspberry Pi über SPI gesteuert, einem seriellen Bus mit vier Leitungen. Der Bus hat einen Master und beliebig viele Slaves, welche je mit einer so genannten Chip-Select-Leitung zum Master verbunden sind. Für die Datenübertragung hat der Bus eine Clock-Leitung, auf welcher der Master die Geschwindigkeit festlegt, sowie eine MOSI- und MISO-Leitung. MOSI bezeichnet hierbei den Output des Masters, daher **M**aster **O**utput **S**lave **I**nput genannt. MISO ist die Leitung für den Datenverkehr von Slave zu Master, daher wird diese **M**aster **I**nput **S**lave **O**utput genannt. Die Daten werden synchron zur Clock auf den Bus gelegt, es erfolgt gleichzeitig der Transfer eines Bytes in beide Richtungen.

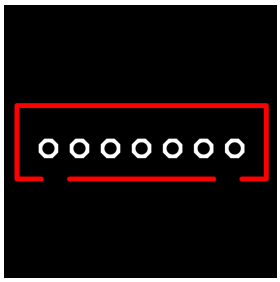
Der BNO055 ist über I2C mit dem Raspberry Pi verbunden, einem seriellen Bus mit zwei Leitungen. I2C steht für Inter-Integrated Circuit. Der Bus hat eine Clock-Leitung, über welche der Master die Übertragungsgeschwindigkeit festlegt,sowie eine Data-Leitung, auf welcher die Daten bidirektional übertragen werden. Das erste Byte jeder Übertragung besteht aus der Adresse des Slaves, an den sich die Kommunikation richtet, sowie ein Bit, welches die Richtung festlegt. Daher sind auf einem Bus maximal 127 Slaves und ein Master möglich. Die einzelnen Leitungen sind mit Pull-Up Widerständen versehen und werden zu Ground durchgeschalten, um eine logische Null am Bus zu senden.

Für eine einfache Datenübertragung zwischen dem Raspberry Pi und einem Computer steht auch noch UART zur Verfügung. UART ist die Kurzform von **U**niversal **A**synchronous **R**eciever **T**ransmitter, einem asynchronen Bus mit je einer Leitung pro Richtung. Der Bus selbst unterstützt jedoch nur zwei Geräte, die je einen Empfänger und einen Transmitter haben. Die Geschwindigkeit der Datenübertragung, Baudrate genannt muss für beide Geräte gleich sein, damit die Daten korrekt empfangen werden können. Auf dieser Platine wird eine Baudrate von 115200 Bit/s genutzt, da die meisten UART-to-USB-Wandler mit dieser Baudrate kommunizieren. Der Anschluss des UART-to-USB-Wandlers erfolgt über eine 5-Pin Buchse auf der Platine.

2.4 Pinouts

Alle Printklemmen haben nur 2 Terminals, welche immer Plus und Minus sind. Bei der Spannungsversorgung muss Plus auf der rechten Seite angeschlossen werden, damit die Komponenten mit

Strom versorgt werden. Ebenfalls ist eine Diode vom Typ ES3D von On Semiconductors verbaut, welche einen Stromfluss in die falsche Richtung verhindert. Die Diode selbst hält maximal 200 V und einen Spitzendurchlassstrom von 100 A aus. Wenn bei den Motorterminals die Kabel vertauscht werden, ändert das lediglich die Drehrichtung des Motors.



Jede der XH-Buchsen ermöglicht ein Anstecken des Steckers nur in eine Richtung und verhindert auch, dass ein Stecker falscher Größe angesteckt werden kann. Alle Buchsen haben einen Ground-Pin und einen 5V-Pin, zusätzlich ist bei allen außer den IOs auch ein 3.3V-Pin vorhanden. Die Buchsen werden für die Beschreibung wie im Bild gezeigt betrachtet und von links nach rechts aufgezählt.

- die IO-Buchsen haben drei Pins: GND, +5V und IO. IO steht hier für den IO Pin, welcher als Input sowie auch als Output verwendet werden kann.
- die ADC-Buchsen haben vier Pins: GND, +3.3V, +5V und AIO. AIO bezeichnet den Pin, der als Input, Output und analoger Input genutzt werden kann.
- die UART-Buchse hat fünf Pins: GND, +3.3V, +5V, RX und TX. RX ist die Reciever-Leitung des Raspberry Pi, TX die Transmitter-Leitung.
- die I2C-Buchsen haben ebenfalls fünf Pins: GND, +3.3V, +5V, SCL und SDA. SCL und SDA werden mit 5V betrieben.
- die User Interface-Buchse hat sechs Pins: GND, +5V und vier GPIOs. Jeder GPIO wird mit einer Maximalspannung von 3.3V betrieben und ist nicht 5V-kompatibel.
- die SPI-Buchsen haben sieben Pins: GND, +3.3V, +5V, MOSI, MISO, SCL und CS. Alle angeschlossenen Geräte dürfen nur als SPI Slave arbeiten und MISO mit maximal 5V steuern. Chip Select hat eine Spannung von 3.3V.
- die Display-Buchse hat acht Pins: GND, +3.3V, +5V, SCL, MISO, MOSI, CS und DC. Die Buchse ist einfach eine SPI-Buchse mit einem weiteren Pin, dem **D**ata-**C**ommand-Pin, welcher bei den meisten Displays benötigt wird.

Der ISP-Stecker entspricht dem In System Programmer mkII-Standard von Atmel/Microchip. Dieser beinhaltet ein SPI-Interface ohne Chip Select, ein Ground-Pin, ein VCC-Pin sowie ein Reset-Pin. Die PWM-Header entsprechen den standardmäßigen Servo-Anschlüssen mit einem 5V-Pin, einem Ground und einem PWM-Pin. Der PWM-Pin hat jedoch nur eine Spannung von 3.3V, da das

PWM-Signal über das Raspberry PI erzeugt wird. Der UART-Header ist für einen UART/TTY to USB-Wandler angepasst, der Pinout lautet von oben nach unten: +3.3V, RX, TX, GND, +5V.

2.5 SMD-Größen

Alle passiven Komponenten mit nur zwei Anschlüssen außer dem Quarz und den Kondensatoren für die Motortreiber sind als **S**urface **M**ounted **D**evice, verbaut. Alle Anschlüsse befinden sich auf einer Seite der Platine und werden dort verlötet. Das Gegenteil hierzu sind Through-Hole-Komponenten, welche Pins besitzen, die durch ein Lötauge gesteckt werden und auf einer oder beiden Seiten verlötet werden. Alle dieser passiven Komponenten sind in einem 0805-Formfaktor verbaut. 0805 steht hierbei für die Größe des einzelnen Bauteils in $\frac{1}{100}$ Zoll. Die größere Zahl hierbei ist die Länge, die andere die Breite. Die Höhe der Komponenten wird nicht über den Formfaktor angegeben, ist jedoch meist kleiner oder gleich der Breite. Somit ergibt sich für die Komponenten eine Größe von
2.032 * 1.27mm bei einer maximalen Höhe von 1.27mm. Die Entscheidung fiel auf diese Bauform, da jene noch händisch lötbar ist und gleichzeitig kaum einen Platz auf der Platine einnimmt.

Nur die Diode beim Stromanschluss ist in einer anderen Bauform, da diese bei Spannungen von 30 V Spitzenströme über 10 A aushalten muss. Diese Diode hat einen SMC / DO-214AB -Formfaktor, welcher eine Länge von 8 mm, eine Breite von 6 mm sowie eine Höhe von 2.65mm hat.[2]

Der ATmega644 ist in einem TQFP-44-Package verbaut, einem sehr flachem IC-Package mit elf Pins auf den vier Seiten. Der Chip selbst ist quadratisch bei einer Seitenlänge von 10mm und einer Höhe von 1mm. Jeder der elf Pins hat eine Länge von 1mm, jedoch liegen davon nur 0.45 bis 0.75mm auf der Platine auf. Die Breite jedes Pins beträgt 0.3 bis 0.45mm, der Abstand zwischen zwei Mitten der Pins beträgt 0.8mm. Auf dem mittleren Quadrat befindet sich ein kleiner Kreis, welcher zur Identifikation des 1. Pins notwendig ist.[3]

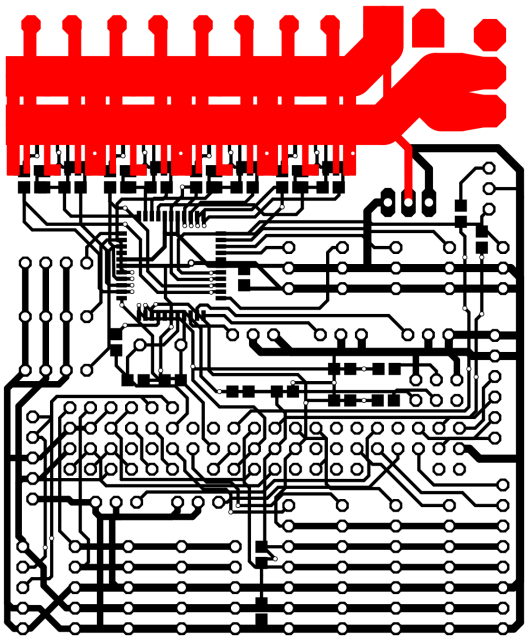
Die vier Motortreiber haben eine SO-16N-Baugröße kombiniert mit einem Widerstand von 100mΩ pro Pin. Dieser geringe Widerstand ermöglicht kurzzeitige Leistungen von maximal 492 Watt, weil die entstehende Wärme dadurch auf ein Minimum reduziert wird. Jeder Motortreiber ist ein Quader mit einer Größe von 9.9 * 3.9 * 1.25mm, zusätzlich dazu kommen auf den längeren Seiten je 8 Pins. Jeder dieser Pins hat eine Länge von 1.5mm bei einer Breite von 0.4mm. Der Abstand zwischen den Mitten zweier Pins beträgt 1.27mm. Der erste Pin ist mit einem Punkt auf dem Quader markiert.[4]

[2] vgl. Fairchild, ES3A-ES3J Fast Rectifiers, S. 5.
[3] vgl. Atmel Corporation, ATmega644/V, S. 362.
[4] vgl. STMicroelectronics, VNH7100AS, S. 1 und 33.

2.6 Abgrenzung der Spannungsniveaus

Die Platine wird über die BATTERY-Klemme mit Strom versorgt, die Spannung muss hierbei zwischen 7V und 36V liegen. Das Raspberry Pi und der ATmega644 benötigen eine Versorgungsspannung von 5V, daher ist ein OKI-78SR-Spannungswandler von Murata Power Solutions verbaut, welcher einen Output von 5V mit maximal 1.5A hat. Jeder Motortreiber wird mit VCC, der vollen Batteriespannung, versorgt. Die Abgrenzung zwischen VCC und dem 5V- / 3.3V-Bereich ist wichtig, weil die Motortreiber kurzzeitig Stromspitzen von über 10 A erzeugen können, welche bei benachbarten Leitungen Spannungsdifferenzen hervorrufen können, sodass die Spannung das Maximum mancher Bauteile überschreitet. Diese Spannungsspitzen können ebenfalls Kommunikationen stören, indem die Spannung eines logischen LOW über die Mindestspannung für ein logisches HIGH steigt und somit als logisches HIGH vom Empfänger der Kommunikation erkannt wird.

Die Grafik auf der rechten Seite stellt in rot den gesamten VCC-Teil dar, der mit voller Batteriespannung betrieben wird und durch Stromspitzen der Motortreiber beeinflusst wird. Die zwei dicken Leitungen, welche von rechts nach links verlaufen, sind die Stromversorgung für die Motortreiber. Diese Leitungen sind so breit, damit sie hohe Ströme bei geringer Erwärmung leiten können auf Grund des niedrigeren Innenwiderstandes.



In schwarz ist der 5V- / 3.3V-Teil, welcher durch den Spannungswandler mit einer stabilen Spannung von 5V versorgt wird. Nur in dem Moment, wo das Board eingeschalten wird, gibt es eine kurze Stromspitze von circa 200mA auf der 5V- / 3.3V-Seite, jedoch sind zu diesem Zeitpunkt noch keine Komponenten aktiv. Alle Bussysteme befinden sich auf der unteren Hälfte der Platine, damit keine Störungen durch die Motortreiber entstehen können.

2.7 Kondensatoren und Schwingquarz

Jeder der vier Motortreiber hat zwischen den Stromversorgungpins VCC und GND einen Elektrolytkondensator mit einer Kapazität von 270μF, welche jeweils die Maximalspannung von VCC aushalten müssen. Das Datenblatt der VNH7100ASTR-Motortreiber sieht diese Kondensatoren

zwischen VCC und GNG vor, sie können noch durch je einen 100 nF-Kondensator ergänzt werden, dieser dient jedoch nur zum Ausgleich hochfrequenter Schwingungen. Die großen Kondensatoren hierbei dienen zum Ausgleich von Spannungseinbrüchen über einen kurzen Zeitraum, sodass die Geschwindigkeit des Motors dennoch konstant bleiben kann.

Auch bei dem ATmega644 ist ein kleiner Kondensator mit einer Kapazität von 100 nF zwischen +5V (VCC) und GND verbaut, welcher geringe Spannungsdifferenzen ausgleicht. Bei dem ATmega ist das für akkurate Messwerte durch den Analog-Digital-Wandler wichtig, sowie für den Schwingquarz, damit der Takt stabil bleibt. Eine instabile Taktrate des ATmega kann zu Problemen bei Verzögerungen führen, da diese in eine fixe Anzahl an Warte-Anweisungen kompiliert werden. Ebenfalls können bei einer variierenden Taktrate Probleme bei diversen Kommunikationen wie zum Beispiel UART entstehen, auf dieser Platine sind jedoch alle Kommunikationen mit dem ATmega synchron, somit sind diese nicht durch Veränderungen in der Taktrate beeinflusst.

Das Raspberry PI sowie der BNO055 benötigen keinen externen Kondensator zur Spannungsstabilisierung, da beide Komponenten solch einen bereits auf dem jeweiligen Board haben. Der BNO055 ist auf einem Breakout-Board verbaut, welches einen Schwingquarz mit 32.768 MHz sowie den
dazugehörigen Bauteilen zur Spannungsstabilisierung enthält.

Die Taktrate des ATmega644 wird durch einen externen Schwingquarz gesteuert, welcher eine maximale Frequenz von 20 MHz haben darf, welche auch auf der Platine genutzt wird, um die volle Leistung des ATmega nutzen zu können. Der Schwingquarz hat zwei Pins, welche direkt mit den XTAL1- und XTAL2-Pins des ATmega verbunden sind. Zusätzlich ist jeder Pin des Schwingquarzes über einen 22 pF-Kondensator mit GND verbunden. Die Kapazität der Kondensatoren wird mit über die Eigenkapazität des Schwingquarzes sowie der Kapazität der Leitungen in der dazugehörigen Schaltung berechnet. Der ATmega selbst hat auch einen internen Taktgeber, welcher jedoch nur eine maximale Frequenz von 8 MHz taktet und durch die Spannung sowie die Temperatur beeinflusst wird. Dieser kann jedoch auf eine ausreichend hohe Genauigkeit kalibriert werden, jedoch kann der ATmega auf Grund der niedrigen Taktrate dennoch nur 40 Prozent der Leistung im Vergleich zu der Leistung bei einem Takt von 20 MHz erreichen.

2.8 Levelshifting

Alle GPIOs des Raspberry PI haben eine Maximalspannung von 3.3V, jedoch schalten viele Komponenten mit einer Spannung von 5V, ein Beispiel hierfür wäre der ATmega644. Um eine Kommu-

nikation zwischen einer 3.3V-Komponente und einer 5V-Komponente zu ermöglichen, kommt das so genannte Levelshiftig zum Einsatz. Auf der Platine werden zwei unterschiedliche Levelshiftingmethoden verwendet, da es in manchen Fällen bidirektional, aber oft auch nur in eine Richtung erfolgen muss.

Alle GPIOs des Raspberry Pi, welche als Output genutzt werden, werden nur mit 3.3V betrieben, da die meisten 5V-Komponenten ab 2.5V ein logisches HIGH erkennen. Die einzigen Systeme, die von 3.3V auf 5V erhöht werden müssen, sind I2C und UART. Auf der Platine ist ein bidirektionaler Levelshifter mit 4 Kanälen verbaut, welche für die I2C-Leitungen SDA und SCL sowie für die UART Recieve- und Transmitleitungen verwendet werden. Der Levelshifter selbst ist auf einem Breakout-Board mit 12 Pins. Jeder Kanal hat einen Mosfet verbaut, welcher auf dem Source-Pin mit dem Kanal auf der 3.3V-Seite, dem Gate-Pin mit +3.3V und dem Drain-Pin mit dem Kanal auf der 5V-Seite verbunden ist. Des weiteren ist auf der 3.3V-Seite jedes Kanals ein 10 kΩ Pull-Up Widerstand verbaut, welcher die Leitung auf +3.3V zieht. Auch auf der 5V-Seite ist ein 10 kΩ Pull-Up Widerstand, welcher die Leitung auf +5V zieht. Wenn der Kanal auf der 3.3V-Seite nun auf Ground durchgeschalten wird, dann existiert eine positive Spannung zwischen Gate und Source, somit schaltet der Mosfet durch und die 5V-Seite wird auch mit Ground verbunden, was ein logisches LOW darstellt. Um eine Signalübertragung in die umgekehrte Richtung zu ermöglichen, hat der Mosfet eine Diode verbaut, welche einen Stromfluss von der 3.3V-Seite zur 5V-Seite erlaubt. Sobald auf der 5V-Seite nun zu Ground durchgeschalten wird, fällt die Spannung auch auf der 3.3V-Seite auf ein logisches LOW.

Um ein 5V-Signal auf 3.3V zu reduzieren, kommt ein einfacher Spannungsteiler mit zwei Widerständen zum Einsatz, dies wird zum Beispiel bei der MISO-Leitung des SPI-Bus benötigt, da die Slaves diese mit 5V schalten, das Raspberry Pi aber maximal 3.3V aushält. Das Verhältnis der Widerstände muss dem Verhältnis der Spannungen entsprechen, bei 5V zu 3.3V ist dieses $1.\overline{51} : 1$. Da zwischen 5V und Ground optimalerweise ein Mindestwiderstand von 5 kΩ existieren sollte, lauten die Widerstände zur Spannungsteilung 2.4 kΩ und 4.7 kΩ, dies entspricht einem Verhältnis von 1.51 : 1 bei Berechnung des Gesamtwiderstands relativ zu dem 4.7 kΩ-Widerstand. Der Gesamtwiderstand zwischen dem 5V-Signal und Ground beträgt somit 7.1 kΩ, was einen Strom von 0.7 mA zu Folge hat. Der Aufbau sieht folgendermaßen aus: die 5V-Signalleitung ist über einen 2.4 kΩ-Widerstand mit der 3.3V-Signalleitung verbunden, von dieser führt ein 4.7 kΩ-Widerstand zu Ground. Ebenfalls limitiert dieser Spannungsteiler den Maximalstrom auf der Schaltleitung auf 0.7 mA, was weit unter den 10 mA Maximum für die GPIOs des Raspberry Pi liegt.

3 Bootloader

3.1 Bootprozess des Raspberry Pi

Der Broadcom BCM2835-SoC, welcher auf dem Raspberry Pi Zero W verbaut ist, besteht aus einem ARM1176JZF-S ARM-CPU, einem Single-Core Prozessor mit einem 1 GHz-Takt, sowie einem Broadcom Videocore IV mit 250 MHz. Das Raspberry Pi Zero W kann über die SD-Karte und über USB booten, in diesem Kapitel wird jedoch nur der Boot über eine SD-Karte abgedeckt. Die Bootreihenfolge sieht folgendermaßen aus:

Zu Beginn bleibt der ARM noch ausgeschaltet, der Videocore führt Code vom ROM aus, welcher die SD-Karte einließt und den 2. Teil des Bootloaders, **bootcode.bin**, lädt und ausführt. **bootcode.bin** aktiviert den SDRAM und lädt die GPU-Firmware **start.elf**. **start.elf** ließt die Konfigurationsdatei **config.txt** und lädt das kompilierte Programm **kernel.img** in den SDRAM. Zum Abschluss wird der ARM gestartet und die Anweisungen des Programms ab der SDRAM-Adresse 0x8000 ausgeführt.[1]

bootcode.bin und start.elf sind Dateien der Raspberry Pi Foundation, welche auf Github.com in der Repository "firmware" des Nutzers "raspberrypi" im Ordner "boot" gefunden werden können. Um eine SD-Karte als Bootmedium einzurichten, muss diese mit dem Filesystem FAT32 ausgestattet sein sowie die beiden Dateien und das Programm, als kernel.img benannt, beinhalten. Die Datei config.txt ist optional und erlaubt unter anderem die Steuerung der Taktrate des ARM sowie des Videocores, dem Einrichten von GPIOs und dem aktivieren oder deaktivieren diverser Schnittstellen, bevor der Kernel gestartet wird.

3.2 Toolchain

Für die Kompilierung des Codes kommt die arm-none-eabi-Toolchain der GNU Compiler Collection zum Einsatz. Diese Toolchain ist für die Programmiersprachen C, C++ und Assembly auf ARM-Bare-Metal-Plattformen geeignet. Eine Bare-Metal-Plattform bezeichnet eine Plattform, auf welcher kein Betriebssystem oder ähnliche Software läuft, welche zum Beispiel den Arbeitsspeicher oder diverse Schnittstellen verwaltet. Der Vorteil einer Bare-Metal-Plattform ist, dass bei höchster Performance nur das wichtigste ausgeführt wird. Würde auf dem Raspberry Pi zum Beispiel ein

[1] vgl. Soininen, Jonne: Linux boot loader and boot in Raspberry Pi, S. 4.

Linux laufen, dann würde ein Filesystem sowie das Terminal im Hintergrund geladen werden und die allgemeine Performance verschlechtern. Der Nachteil ist jedoch das Fehlen mancher Systeme wie zum Beispiel die Arbeitsspeicherverwaltung. Um solche Systeme jedoch dennoch nutzen zu können, muss im eigenen Programm entsprechender Source-Code verbaut werden.

Zum Kompilieren eines C/C++/Assembly-Programmes, wird der Konsolenbefehl `arm-none-eabi-g++` verwendet, dieser Befehl kompiliert .c/.cpp/.s-Dateien direkt in eine .elf-Datei. Dieser Befehl kombiniert den C/C++ Preprocessor, den Compiler, den Assembler sowie den Linker, daher kann mit einem Befehl ein ausführbares Programm erstellt werden. Da der Befehl so viele Kompilierungsschritte umfasst, gibt es auch viele Argumente, folgende werden für die Kompilierung genutzt:

- mit `-o [Dateiname]` wird die Outputdatei definiert
- mit `-Ox` wird ein Optimierungslevel definiert, x = 0 resultiert in keiner Optimierung, x = 3 erwirkt eine maximale Optimierung
- mit `-march=[Plattform]` wird die Zielplattform bestimmt, auf dieser Platine ARM1176JZF-s
- mit `-mtune=[Prozessor]` wird der Prozessor bestimmt, auf dieser Platine ARMv6-ZK
- mit `-mfpu=[Hardware]` wird die Floating-Point-Hardware definiert, auf dieser Platine "vfp"
- mit `-mfloat-abi=[Art]` wird der Umgang mit Floating-Point-Variablen definiert, "hard" nutzt die Floating-Point-Hardware für Berechnungen
- mit `-W[Warnung]` kann die angegebene Warnungsart unterdrückt werden
- mit `-nostartfiles` wird das Linken der Standard-Startfiles verhindert, sodass nur der eigene Code ausgeführt wird

Die zu kompilierenden Dateien werden ohne einen Präfix und durch Leerzeichen getrennt übergeben. Für Assembly-Dateien werden der Preprocessor und der Kompilierungsschritt übersprungen. Um aus der Ouput-Datei nun ein Datenträgerimage (.img) zu erzeugen, wird der Befehl `arm-none-eabi-objcopy` mit dem `-binary` Argument verwendet.

Für die Kompilierung des Source-Codes werden folgende zwei Befehle nacheinander ausgeführt:

1) `arm-none-eabi-g++ -Wwrite-strings -nostartfiles -mfloat-abi=hard -O0 -mfpu=vfp -march=armv6zk -mtune=arm1176jzf-s -o kernel.elf [Dateien]`

2) `arm-none-eabi-objcopy kernel.elf -O binary kernel.img`

3.3 Bootloader in Assembler

Die Aufgabe des Bootloaders besteht darin, den Vector-Floating-Point-Coprozessor (VFP) des ARM zu aktivieren, sowie den Stack Pointer auf die Adresse 0x8000 zu setzten und anschließend die Funktion `main` des C++-Programmes aufzurufen. Der Code zur Aktivierung der VFP stammt von dem Github.com-Nutzer dwelch67.

```
.globl _start
_start:
```

Als Erstes wird die globale Funktion `_start` definiert, welche den Startpunkt eines Programmes definiert. Der Compiler sucht nach der `.start`-Funktion, wenn diese nicht gefunden wird, startet das Programm bei `main`.

```
    mrc p15, 0, r0, c1, c0, 2
    orr r0,r0,#0x300000
    orr r0,r0,#0xC00000
    mcr p15, 0, r0, c1, c0, 2
```

Mit der MRC-Anweisung wird das c1-Register ausgelesen und in das Register r0 kopiert. Nun werden in r0 die Bits 20-23 auf 1 gesetzt und das Ergebnis mit MCR wieder in das Register c1 geschrieben. Dies hat zur Folge, dass nun ein Zugriff auf die VFP-Unit möglich ist.

```
    mov r0,#0x40000000
    fmxr fpexc,r0
    mov sp,#0x00008000
    bl main
```

Anschließend wird das Register r0 mit dem Hexadezimalwert 0x40000000 beschrieben und mit FMXR in das FPEXC-Register geschrieben, um die VFP-Unit zu aktivieren. Danach wird der Stack Pointer auf die Adresse 0x8000 verschoben, wo die erste Anweisung des C++-Programms liegt. Danach wird mit der Branch-Link-Anweisung die `main`-Funktion aufgerufen.

Der Bootloader wird in einer .s-Datei abgespeichert und diese wird bei der Kompilierung genau wie die .cpp-Dateien übergeben. Der Compiler erkennt dann das `.start`-Symbol und nutzt dies als Einstiegspunkt.

4 Raspberry Pi Software

4.1 Struktur und Register

Die C++-Library besteht ausschließlich aus Header-Dateien, welche mit `#include` in .cpp eingebunden werden können. Dies hat den Vorteil, dass nicht jede einzelne Datei im Vorhinein in eine Object-Datei kompiliert werden muss, welche dann in eine Shared-Object-Datei kombiniert werden würde. Eine Shared-Object-Datei ist eine Binärdatei, womit Änderungen im Source-Code der Library nicht mehr möglich wären.

Jede Header-Datei enthält den `#pragma once` Includeguard, welcher eine mehrfache Definition der gleichen Klassen und Funktionen verhindert. Die Datei `VWAPI.hpp` inkludiert alle anderen Header-Dateien, somit genügt `#include "VWAPI.hpp"` um Zugriff auf alle Funktionen der Library zu erhalten. Zu Beginn eines jeden Programmes sollte die Funktion `init` der Klasse `VWAPI` aufgerufen werden, welche alle GPIOs richtig konfiguriert und die Verbindung mit dem ATmega überprüft. Die Funktion kann einfach mit `VWAPI().init();` aufgerufen werden.

Der ARM11-Prozessor des BCM2835-SoC ist ein 32-Bit Prozessor, somit können die Speicheraddressen 0x00000000 bis 0xFFFFFFFF adressiert werden. Alle GPIOs werden über den Videocore gesteuert, somit sind die Adressen im Datenblatt des BCM2835 als Videocore-Adressen gegeben. Wenn nun von dem Programm, was auf dem ARM läuft, ein IO-Register adressiert wird, muss auf das Memorymapping geachtet werden. Alle IO-Adressen sind bei dem Videocore ab der Adresse 0x7F000000 zu finden, diese sind für dem ARM ab der Adresse 0x20000000 erreichbar.

Register können direkt mit Werten beschrieben werden, jedoch können auch OR, XOR und AND-Operationen auf diese angewendet werden. Wichtig ist hierbei, dass diese bei Registern mit Write-only Bits fehlschlagen und keine Auswirkung haben. Das Beschreiben eines Read-only Bits hat keine Wirkung, jedoch schlägt nicht der gesamte Schreibvorgang fehl. Alle Register werden im Code über `volatile unsigned int*` Pointer adressiert. Unsigned Integer erlaubt hierbei die Nutzung aller 32 Bits des Registers, ohne dass andernfalls negative Werte Auswirkungen auf das ganze Register haben. Volatile definiert, dass der Wert der Variable sich außerhalb des Programmes ändern kann, daher wird immer der aktuellste Wert von der Speicheradresse abgerufen.

4.2 GPIO

Die Steuerung der GPIOs erfolgt über Register mit der Grundadresse 0x20200000. Da der BCM2835 54 GPIOs hat, sind alle Funktionen auf mehrere Register aufgeteilt, weil pro Register nur 32 Bit zur Verfügung stehen. In der C++-Library steuert die Klasse **GPIO** alle Funktionen der GPIOs.

```
class GPIO {
    volatile unsigned int* reg;
    char id;
public:
    enum Mode { INPUT, OUTPUT, ALT0 = 4, ALT1 = 5,
                ALT2 = 6, ALT3 = 7, ALT4 = 3, ALT5 = 2};
```

Die Klasse GPIO enthält einen `volatile unsigned int*`-Pointer sowie eine `char`, welche die ID des Pins speichert. Des weiteren definiert die Klasse das Enum **Mode**, welches alle Modi der GPIOs enthält und diese mit der internen ID versieht.

```
    GPIO() {}

    GPIO(char pin) {
        reg = (unsigned int*) 0x20200000;
        id = pin;
    }

    GPIO(char pin, Mode mode) {
        reg = (unsigned int*) 0x20200000;
        id = pin;
        setMode(mode);
    }
```

Es gibt 3 Konstruktor, mit denen eine Instanz der Klasse erstellt werden kann. Der Default-Konstruktor hat keine Wirkung und dient nur zur Erstellung leerer Instanzen. Wenn nur ein Pin übergeben wird, wird dieser in das ID-Feld übertragen. Sollte auch noch ein Mode übergeben werden, wird die Funktion `setMode(mode)` aufgerufen. Beide Konstruktor setzen die Adresse des Pointers auf 0x20200000, der Grundadresse aller GPIO-Register.

```
    Mode getMode() {
        return static_cast<Mode>((reg[id/10] >> ((id % 10) * 3)) & 0x07);
    }

    void setMode(Mode mode) {
        reg[id/10] &= ~(0x07 << ((id % 10) * 3));
        reg[id/10] |= ((mode & 0x07) << ((id % 10) * 3));
    }
```

Der Modus eines Pins wird in den GPIO-Function-Select-Registern gespeichert, GPIO-Function-Select 1 ist beispielsweise für die Pins 10 - 19 zuständig. Jeder Pin hat 3 Bits, welche den Modus darstellen. Der Pin mit der Einerziffer 0 hat die niedrigsten Bits, der mit Einerziffer 9 die Bits 27-29.

Um einen Modus zu setzen, werden die drei Bits des Pins zuerst auf 0 gesetzt, dies erfolgt mit einer AND-Operation wo alle Bits außer den drei eine 1 sind. Anschließend wird die korrekte Zahl in das Register mit einer OR-Operation geschrieben. Beim Auslesen wird einfach der Wert der drei Bits mit Hilfe von dem `static_cast`-Operator in ein Objekt vom Typ `Mode` umgewandelt.

```
bool getLevel()  {
    return reg[13 + id/32] & (1 << (id % 32));
}

void setLevel(bool lvl) {
    reg[lvl?7:10 + (id/32)] = (1 << (id % 32));
}
```

Das Auslesen eines Pins erfolgt über eine Abfrage im GPIO-Level-Register. Für die Pins 0-31 hat dies einen Offset von 13, für alle darüber einen Offset von 14. Das Register wird ausgelesen und über ein logisches AND mit einer binären 1 an der richtigen Stelle gefiltert. Das Ergebnis wird als `bool` zurückgeliefert. Die Steuerung eines Outputs erfolgt über die GPIO-Set- und GPIO-Clear-Register welche für die Pins 0-31 einen Offset von 7 bzw. 10, für die Pins darüber einen Offset von 8 bzw. 11 haben. Um einen Pin auf HIGH zu schalten, wird in das GPIO-Set-Register eine 1 an der richtigen Stelle geschrieben, für LOW in das GPIO-Clear-Register.

4.3 Status LED

Die Status LED auf der Platine ist eine 3-Farben LED, welche aus einer roten und einer grünen LED besteht, welche sich die Anode teilen. Die grüne LED ist an GPIO 24, die Rote an GPIO 25 angeschlossen. Die LED kann folgende Zustände haben: AUS, ROT, GRÜN und GELB. Die Steuerung der LED erfolgt über die **LED**-Klasse.

```
#include "GPIO.hpp"

class LED {
public:
    enum Color {OFF, RED, GREEN, YELLOW};
```

Die Klasse **LED** nutzt die Klasse **GPIO** zur Steuerung der zwei Pins, daher wird `GPIO.hpp` inkludiert. Ebenfalls wird ein Enum mit allen Farben der LED erstellt, welches als Parameter für Funktionen dient.

```
LED() {
   GPIO(24,GPIO::OUTPUT);
   GPIO(25,GPIO::OUTPUT);
}
```

Der Konstruktor der Klasse benötigt keine Argumente und setzt beide GPIOs als Output. In der LED-Klasse wird auf Objekte verzichtet, welche im `private`-Bereich der Klasse gespeichert werde, Alle `GPIO`-Instanzen werden nur lokal in den Funktionen erstellt und genutzt.

```
    void set(Color c) {
        GPIO(25).setLevel(c & 0x01);
        GPIO(24).setLevel(c & 0x02);
    }
```

Das setzen der Farbe erfolgt über die Funktion `set`, welche ein Objekt vom Typ `Color` als Argument annimmt. Da jeder Wert in der Enum `Color` einen Wert von 0 bis 3 hat, kann über ein logisches AND dieser genutzt werden, um die Pins 24 bzw. 25 entsprechend anzusteuern.

4.4 Timer

Die Steuerung des Timers sowie Delay-Funktionen sind in der Klasse **Timer** verankert. Die Klasse nutzt den System-Timer des BCM2835, welcher einen Free-Running-Counter mit einer Frequenz von 1 MHz funktioniert. Eine Verzögerung von 1 Sekunde entspricht somit 1000000 Schritte des Counters. Die Grundadresse des Timer-Registers ist 0x20003000.

```
class Timer {
    bool paused;
    unsigned int val;
    unsigned int started;
    volatile unsigned int* reg;
public:
```

Die Klasse enthält vier Variablen im `private`-Bereich:

- `paused`, eine Boolean die darstellt, ob der Timer pausiert ist
- `val`, eine unsigned Integer, die den aktuellen Wert darstellt
- `started`, eine unsigned Integer, die den Startzeitpunkt speichert
- `reg`, ein `volatile unsigned int*`-Pointer, welcher auf die Grundadresse des Timer-Registers zeigt

```
    Timer() {
        reg = (volatile unsigned int*) 0x20003000;
        val = 0;
        started = 0;
        paused = true;
    }
```

Der Konstruktor setzt `reg` auf die Adresse 0x20003000, sowie die Variablen `val` und `started` auf 0 und die Boolean `paused` auf false.

```
void start() {
    started = reg[1];
    paused = false;
    val = 0;
}
```

Die Funktion `start` startet den Timer, indem sie den Wert des Free-Running-Counter aus dem unteren Counter-Register, welches einen Offset von 1 hat, in `started` speichert und `paused` auf false sowie `val` auf 0 setzt.

```
void stop() {
    paused = true;
    val += reg[1] - started;
    started = 0;
}

void pause() {
    paused = true;
    val += reg[1] - started;
    started = 0;
}
```

Die Funktion `stop` stoppt den Timer, indem sie `paused` auf false setzt. Anschließend wird der aktuelle Wert, welcher die Differenz des Counters und `started` ist, wird zu dem in `val` addiert. Abschließend wird `started` auf 0 gesetzt. Die Funktion `pause` funktioniert gleich und ist somit nur eine Alternative zu `stop`.

```
void resume() {
    if(!paused) return;
    paused = false;
    started = reg[1];
}
```

Während bei der Funktion `start` der Wert von `val` auf 0 gesetzt wird, wird dieser bei der Funktion `resume` nicht verändert. Somit ermöglicht `resume` das Fortsetzen eines pausierten Timers. Die if-Abfrage verhindert, dass ein aktiver Timer fortgesetzt wird.

```
bool isRunning() {
    return !paused;
}
```

Die Funktion `isRunnung` liefert den invertierten Wert von der Boolean `paused` zurück, somit wird dargestellt, ob der Timer gerade aktiv ist.

```
    unsigned int get() {
        return (val + paused?0:(reg[1] - started)) / 1000000;
    }

    unsigned int get_ms() {
         return (val + paused?0:(reg[1] - started)) / 1000;
    }

    unsigned int get_us() {
         return val + paused?0:(reg[1] - started);
    }
```

Die Funktionen `get`, `get_ms` und `get_us` liefern den aktuellen Wert des Timers zurück, indem der gespeicherte Wert in `val` mit dem aktuellen Wert addiert wird, so fern der Timer gerade aktiv ist. Ansonsten wird nur der Wert aus `val` zurückgeliefert. Bei `get` wird der Wert durch 1000000 dividiert, um diesen in Sekunden umzuwandeln, bei `get_ms` durch 1000, um diesen in Millisekunden umzuwandeln.

```
void wait(unsigned int seconds) {
    seconds = (seconds * 1000000) + *((volatile unsigned int*) 0x20003004);
    while(*((volatile unsigned int*) 0x20003004) < seconds);
}

void wait_ms(unsigned int millis) {
    millis = (millis * 1000) + *((volatile unsigned int*) 0x20003004);
    while(*((volatile unsigned int*) 0x20003004) < millis);
}

void wait_us(unsigned int micros) {
    micros = micros + *((volatile unsigned int*) 0x20003004);
    while(*((volatile unsigned int*) 0x20003004) < micros);
}
```

Die Funktionen `wait`, `wait_ms` und `wait_us` sind außerhalb der Klasse **Timer** definiert und verzögern das Programm für eine bestimmte Zeit. Die Funktion nimmt ein Argument vom Typ `unsigned int` an. Zuerst wird der Zeitpunkt berechnet, an welchem die Verzögerung abgeschlossen ist. Dieser ist die Summe der übergebenen Verzögerung in μs und dem aktuellen Wert des unteren Counter-Registers, welches auf der Adresse 0x20003004 liegt. Anschließend wird mit einer while-Schleife gewartet, bis der Wert größer oder gleich dem berechneten Zeitpunkt ist.

4.5 PWM

Die PWM-Controller des Raspberry Pi werden über die Klasse **PWM** gesteuert. GPIO 12 und 13 sind mit Kanal 0 und 1 des PWM-Controllers verbunden und können über diesen als PWM-Output

genutzt werden. Um PWM nutzen zu können, muss zuerst der Clock-Manager richtig konfiguriert werden.

```
#include "GPIO.hpp"

class PWM {
    char channel;
    volatile unsigned int* reg;
public:
```

Die Klasse **PWM** enthält eine `char`, welche den aktuellen Kanal speichert sowie einen `volatile unsigned int*`-Pointer, welcher auf die Grundadresse der PWM-Register zeigt.

```
    PWM() {}

    PWM(char channel) {
        this->reg = (unsigned int*) 0x2020C000;
        this->channel = (channel==0) ? 1 : 2;
```

Der Default-Konstruktor dient nur der Erstellung leerer Instanzen, der richtige Konstruktor nimmt ein Argument von Typ `char` an, welches den PWM-Kanal definiert (0 = GPIO 12, 1 = GPIO 13). Als erstes setzt der Konstruktor `reg` auf die Grundadresse der PWM-Register, 0x2020C000, anschließend wird der `channel` gemäß dem Argument gesetzt. Channel 0 wird zu Channel 1, Channel 1 zu Channel 2. Dies erlaubt eine einfachere Kontrolle der Register im Code.

```
        volatile unsigned int* clock = (unsigned int*) 0x201010A0;
        clock[1] = (0x5a << 24) | (8 << 12);
        clock[0] = (0x5a << 24) | (1 << 4) | 0x01;
```

Anschließend wird ein `volatile unsigned int*`-Pointer erstellt, welcher auf die Grundadresse des Clock-Managers, 0x201010A0, zeigt. Nun wird im Clock-Manager die Geschwindigkeit der Clock festgelegt, indem in das Clock-Divider-Register, welches einen Offset 1 hat, eine 8 auf die Bits 12-23 geschrieben wird. Ebenfalls muss in jedes Clock-Manager-Register in die obersten acht Bits der Wert 0x5A geschrieben werden, damit überhaupt ein Zugriff möglich ist. Danach wird die Clock gestartet, indem in das Clock-Control-Register, welches einen Offset von 0 hat, eine 1 auf das fünfte Bit geschrieben wird. Gleichzeitig wird die Quelle der Clock auf den internen Schwingquarz gesetzt, indem eine 1 auf das erste Bit geschrieben wird.

```
        GPIO(this->channel + 11,GPIO::ALT0);
        reg[0] |= (0x81) << (this->channel == 1 ? 0 : 8);
        reg[this->channel * 4] = 1000;
    }
```

Nach dem Einrichten der Clock wird der entsprechende GPIO auf die ALT0-Funktion gesetzt. Danach wird im PWM-Control-Register, welches einen Offset von 0 hat, der Wert 0x81 je nach Channel auf das erste oder zweite Byte geschrieben. Abschließend wird das Range-Register, welches je nach

Channel einen Offset von 4 oder 8 hat, mit dem Wert 1000 beschrieben. Dieses Register gibt die Auflösung des PWM-Signals an, somit gibt es 1000 Mögliche Stufen.

```
    void set(double value) {
        value = value > 1.0 ? 1.0 : (value < 0.0 ? 0.0 : value);
        reg[this->channel * 4 + 1] = value * 1000.0;
    }
```

Um ein PWM-Signal zu generieren, wird die Funktion `set` verwendet. Als Argument wird eine Variable vom Typ `double` übergeben, welche einen Wertebereich zwischen 0.0 und 1.0 haben darf. Sollte der Wert außerhalb dieses Bereiches liegen, wird er automatisch angepasst. Anschließend wird der Wert mit 1000 multipliziert und in das PWM-Data-Register geschrieben, welches je nach Channel einen Offset von 5 oder 9 hat.

4.6 UART

Der UART-Controller des Raspberry Pi wird mit der Klasse **UART** gesteuert und erlaubt die Verbindung von UART-Geräten an das Board. Ebenfalls kann er über einen UART/TTY-to-USB-Wandler als Debug-Schnittstelle zum PC genutzt werden. Auf der Platine sind nur die TX- und RX-Pins in Verwendung, die RTS- und CTS- sind anders verwendet. Der UART-Controller besitzt zwei FIFO-Buffer für Kommunikationen mit mehreren Bytes.

```
#include "GPIO.hpp"

class UART {
    volatile unsigned int* reg;
public:
```

Die Klasse **UART** enthält nur eine Variable im `private`-Bereich, einen `volatile unsigned int*`-Pointer, welcher auf die Grundadresse der UART-Register, 0x20201000 zeigt. Der Import der Klasse **GPIO** wird für das setzen der korrekten Pin-Funktion benötigt.

```
    UART() {
        reg = (unsigned int*) 0x20201000;
        GPIO(14, GPIO::ALT0);
        GPIO(15, GPIO::ALT0);
        reg[9] = 26;
        reg[10] = 5;
        reg[11] = (7 << 4);
        reg[12] = (3 << 7) | 0x01;
    }
```

Der einzige Konstruktor der Klasse nimmt keine Argumente an und macht somit den Default-Konstruktor nutzlos. Als erstes wird der Pointer `reg` auf die Adresse 0x20201000 gesetzt, der Grundadresse der UART-Register. Anschließend werden die GPIOs 14 und 15 mit der Alternativen Funktion 0 konfiguriert, um als UART-Pins zu fungieren. Die Frequenz wird mit Hilfe der Register 9 und 10 auf eine Baudrate von 115200 angepasst. Danach werden im Line-Control-Register, welches einen Offset von 11 hat, die FIFOs aktiviert und die Wortlänge auf 8 gesetzt. Dies erfolgt durch ein setzten der Bits 4, 5 und 6. Als letztes wird im Register mit Offset 12, den Control-Register, UART, der Reciever und der Transmitter aktiviert, indem das Bit 0, 7 und 8 auf 1 gesetzt wird.

```
void print(char* string) {
    while(*string) {
        while(reg[6] & (1 << 5));
        reg[0] = *(string++);
    }
}
```

Die Funktion `print` nimmt einen String als Argument an, welcher dann über UART geschrieben wird. Mit einer `while`-Schleife wird jedes Zeichen nacheinander behandelt: Zuerst wird mit einer `while`-Schleife gewartet, falls der Transfer-FIFO voll ist. Dies wird am Bit 5 im UART Flag Register, welches einen Offset von 6 hat, erkannt. Anschließend wird das Zeichen in den FIFO geladen, indem es auf die niedrigsten acht Bits des Registers mit Offset 0, dem UART Data Register, geschrieben wird. Der FIFO wird dann automatisch asynchron auf die Transfer-Leitung gesendet.

```
char* read(int bytes) {
    char data[bytes];
    if(reg[6] & (1 << 4)) return 0;
    for(int i = 0; !(reg[6] & (1 << 4)); i++) {
        data[i] = reg[0] & 0xFF;
    } char* ptr = data;
    return ptr;
}
```

Die Funktion `read` liefert einen `char*`-Pointer zu erhaltenen Daten zurück und nimmt die Anzahl der auszulesenden Byte als Argument vom Typ `int` an. Zuerst wird mit einer `if`-Abfrage überprüft, ob der Recieve-FIFO leer ist. Dies kann an Bit 4 im UART Flag Register, welches einen Offset von 6 hat, erkannt werden. Wenn der FIFO leer ist, wird ein Null-Pointer zurückgegeben. Ansonsten wird mit einer `for`-Schleife, mit der Bedingung, dass der FIFO nicht leer ist, das Array befüllt. Der Index an der aktuellen Zählvariable der `for`-Schleife wird mit dem Wert der untersten acht Bits des Data-Registers beschrieben. Nach der Befüllung des Arrays wird ein Pointer erstellt, welcher auf das Array zeigt und zurückgegeben.

4.7 SPI

Das SPI-Interface des Raspberry Pi wird über die Klasse **SPIDevice** gesteuert. Es könnte als Master und als Slave genutzt werden, hier wird jedoch nur die Master-Funktion verwendet. Der SPI-Master enthält wie der UART-Controller auch zwei FIFO-Buffer, einen für zu sendende und einen für empfangene Daten. Die Steuerung der Chip-Select-Pins erfolgt mit Hilfe der Klasse **GPIO**.

```
#include "GPIO.hpp"

class SPIDevice {
    char cs;
    char mode;
    bool pulse;
    unsigned int freq;
    volatile unsigned int* reg;
public:
```

Die Klasse enthält fünf Variablen im `private`-Bereich:

- `cs`, ein Character, welcher die ID des CS-Pins speichert.
- `mode`, ein Character, welcher den SPI-Mode(0 - 3) speichert.
- `pulse`, eine Boolean, die darstellt, ob der CS-Pin nach jedem Byte kurz auf High geht.
- `freq`, eine unsigned Integer, in der die Frequenz gespeichert wird.
- `reg`, einen `volatile unsigned int*`-Pointer, welcher auf die Grundadresse der SPI-Register zeigt.

Der Import der Klasse **GPIO** wird zur Steuerung der Chip-Select-Pins benötigt.

```
    SPIDevice() {}

    SPIDevice(char cs, char mode, bool pulse, unsigned int freq) {
        this->reg = (unsigned int*) 0x20204000;
        this->cs = cs;
        this->mode = mode;
        this->pulse = pulse;
        this->freq = (125000000/freq)*2;
        GPIO(9, GPIO::ALT0);
        GPIO(10, GPIO::ALT0);
        GPIO(11, GPIO::ALT0);
        GPIO(cs, GPIO::OUTPUT).setLevel(true);
    }
```

Die Klasse hat zwei Konstruktor, der Default-Konstruktor ist jedoch nur zur Erstellung leerer Instanzen gedacht. Der andere Konstruktor nimmt vier Argumente an, welche den Variablen im `private`-Bereich der Klasse abgesehen von `reg` entsprechen. Als erstes wird `reg` auf die Grundadresse der SPI-Register, 0x20204000, gesetzt. Anschließend wird der Wert der Argumente `cs`, `mode` und `pulse` in die gleichnamigen Variablen kopiert. Da die Frequenz in Clock-Ticks angegeben werden muss, muss sie zuerst mit folgender Formel umgerechnet werden, bevor sie in `freq` gespeichert wird: $freq_{ticks} = \frac{125Mhz}{freq_{Hz}} * 2$. Anschließend werden die GPIOs 9, 10 und 11, welche mit dem SPI-Controller verbunden sind, auf die Alternative Funktion 0 gesetzt, um die SPI-Funktion zu aktivierend. Zuletzt wird der CS-Pin als Output konfiguriert und auf HIGH gesetzt.

```
void setFrequency(unsigned int freq) {
    this->freq = (125000000/freq)*2;
}

void setMode(char mode) {
    this->mode = mode;
}

void setPulse(bool pulse) {
    this->pulse = pulse;
}
```

Die Funktion `setFrequency` nimmt ein Argument vom Typ unsigned Integer an und setzt den Wert der Variable `freq` mit der selben Formel wie im Konstruktor. Die Funktionen `setMode` und `setPulse` nehmen je eine Argument an und kopieren dieses in die gleichnamige Variable.

```
unsigned char transfer(unsigned char data) {
    GPIO(cs).setLevel(false);
    reg[0] = (mode << 2);
    reg[2] = freq;
    reg[0] |= (3 << 4) | (1 << 7);
```

Die Funktion `transfer` sendet ein Byte, welches als Argument vom Typ unsigned Character übergeben wird, und liefert die Antwort auch als unsigned Character zurück. Zuerst setzt die Funktion des CS-Pin auf LOW, danach wird im SPI-Control-and-Status-Register, welches einen Offset von 0 hat, der Wert der Variable `mode` auf die Bits 2 bis 3 geschrieben. Anschließend wird in das Clock-Divider-Register, welches einen Offset von 2 hat, der Wert der Variable `freq` geschrieben, um die Übertragungsgeschwindigkeit zu setzen. Nun wird auf die Bits 4, 5 und 7 im Control-and-Status-Register eine 1 geschrieben, welche den FIFO leert und die Kommunikation startet.

```
    reg[1] = data;
    while(!(reg[0] & (1<<16)));
    GPIO(cs).setLevel(true);
    return reg[1];
}
```

Danach wird in das FIFO-Register, welches einen Offset von 1 hat, das zu Übertragende Byte geschrieben, welches dann gesendet wird. Mit einer while-Schleife wird nun gewartet, bis die Kommunikation abgeschlossen wurde. Dies erfolgt über die Abfrage des Bit 16 im Control-and-Status-Register, welches beim Ende der Kommunikation auf 1 geht. Abschließend wird der CS-Pin wieder auf HIGH gesetzt und das empfangene Byte aus dem FIFO-Register gelesen und zurückgegeben.

```
unsigned char* transfer(unsigned char* data, unsigned int len) {
    unsigned char out[len];
    if(pulse) {
        for(int i = 0; i < len; i++) out[i] = transfer(data[i]);
    }
```

Eine Kommunikation über mehrere Bytes erfolgt über die Funktion `transfer`, welche einen `unsigned int*`-Pointer sowie eine unsigned Integer als Argumente annimmt und einen `unsigned int*`-Pointer zurückliefert. Die Übergabe der Daten erfolgt als Pointer zu einem Array, welches die Länge `len` hat. Der zurückgelieferte Pointer zeigt auf ein Array mit Länge `len`, in welchem die empfangenen Bytes gespeichert sind. Als erstes wird das Ausgabe-Array `out` mit der Länge `len` erstellt, danach wird mit einer if-Abfrage unterschieden, ob `pulse` true oder false ist. Wenn `pulse` true ist, dann wird mit einer for-Schleife für jedes Byte einzeln die Funktion `transfer` aufgerufen und der Rückgabewert an der richtige Stelle in das Array `out` gespeichert.

```
    else {
        GPIO(cs).setLevel(false);
        reg[0] = (mode << 2);
        reg[2] = freq;
        reg[0] |= (3 << 4) | (1 << 7);
        for(int i = 0; i < len; i++) reg[1] = data[i];
        while(!(reg[0] & (1<<16)));
        for(int i = 0; i < len; i++) out[i] = reg[1];
        GPIO(cs).setLevel(true);
    }
```

Wenn `pulse` jedoch false ist, findet der gleiche Vorgang wie in der Funktion `transfer` statt, jedoch wird statt dem einfachen Schreibvorgang in das FIFO-Register jedes Byte mit einer for-Schleife nacheinander hineingeschrieben. Beim Auslesen der empfangenen Daten wird wieder mit Hilfe einer for-Schleife ein Byte aus dem FIFO gelesen und in der richtigen Stelle im Array `out` gespeichert.

```
     unsigned char* ptr = out;
    return ptr;
}
```

Da in C++ keine Arrays als Argumente oder Rückgabewerte übergeben werden sollen, muss ein Pointer, welcher auf das Array `out` zeigt, erstellt und als Rückgabewert der Funktion ausgegeben werden.

4.8 I2C

Der I2C-Controller des Raspberry Pi wird mit der Klasse **I2CDevice** gesteuert, jedoch nur als I2C-Master, da auf der Platine keine andere Verwendung vorhergesehen ist. Der I2C-Controller enthält wie der SPI-Controller und der UART-Controller auch zwei FIFO-Buffer, welche eine Kommunikation mit mehreren Bytes ermöglichen.

4.9 BNO055

5 ATmega Software

5.1 8-Bit AVR Plattform

Die Programmierung des Subprozessors, einem ATMega644, erfolgt auch in C++. Als Entwicklungsumgebung kommt Atmel Studio zum Einsatz, da der `avr-gcc`-Compiler dort bereits inkludiert ist. Der Sourcecode wird in eine Binärdatei kompiliert, welche dann mit Hilfe eines AVR-In-System-Programmers auf den ATmega geladen wird. Die Verbindung mit dem ATmega erfolgt über den ISP-Stecker, welcher auf der Platine verbaut ist.

Der ATMega644 hat vier Fuses-Register, über welche ein paar grundlegende Einstellung vorgenommen werden können: Im Low-Fuse-Register kann die Taktrate gesteuert werden, sowie ob ein externer Quarz verwendet wird. Auf der Platine ist ein Full-Swing-Quarz mit einer Taktrate von 20 MHz verbaut, somit soll dieser auch verwendet werden. Die Option `Divide clock by 8 internally` sollte ausgeschaltet werden, damit eine höhere Leistung erreicht werden kann. Im High-Fuse-Register muss SPI aktiviert sein, um eine Kommunikation mit dem Raspberry Pi zu ermöglichen, aber JTAG deaktiviert sein, da das JTAG-Interface ein paar Pins in der Funktionalität einschränkt.

Der der ATMega644 ein 8-Bit-Prozessor ist, ist jedes Register auf eine Größe von 8 Bits beschränkt. Wenn etwas ein größeres Register benötigt, ist es auf mehrere Register aufgeteilt. Alle Register sind in der Library von Atmel/Microchip definiert, somit müssen sie nicht über die Speicheradresse adressiert werden. Auch die einzelnen Bits sind bereits mit Namen definiert, was die Programmierung wesentlich erleichtert.

5.2 IO

Alle 32 ansteuerbaren Pins des ATMega sind in 4 Gruppen aufgeteilt, welche mit A bis D benannt sind. Das Steuern der Pins erfolgt mithilfe der Data-Direction-(`DDR`), Port-(`PORT`) und Pin(`PIN`)-Register, welche pro Gruppe je einmal existieren. Um beispielsweise den Pin B2 als Output zu definieren, muss in das `DDRB`-Register auf Bit 2 eine 1 geschrieben werden.

Um einen Pin zwischen Output und Input zu wechseln, wird das `DDR`-Register genutzt. Wenn an das entsprechende Bit eine 1 geschrieben wird, ist der Pin ein Output. Um einen Output ein- oder

auszuschalten wird das `PORT`-Register genutzt, eine 1 auf dem entsprechendem Bit resultiert in einem logischen HIGH. Sollte der Pin ein Input sein, aktiviert eine 1 im `PORT`-Register den internen Pull-Up-Widerstand. Um einen Pin auszulesen, muss das entsprechende Bit im `PIN`-Register gelesen werden, eine 1 signalisiert ein logisches HIGH auf dem Pin.

5.3 SPI

Das SPI-Interface des ATmega liegt auf den Pins B4 bis B7, wovon B4 der Chip-Select-, B5 der MOSI-, B6 der MISO und B7 der Clock-Pin ist. Der SPI-Controller kann als SPI Slave sowie als SPI Master genutzt werden, hier auf der Platine ist der ATMega jedoch ein SPI Slave, welcher von dem Raspberry Pi als Master gesteuert wird. Das SPI-Interface besitzt im Gegensatz zum Raspberry Pi keinen Buffer, sondern nur ein 8-Bit Data-Register.

Zur Initialisierung des SPI-Controllers müssen folgende Schritte erfolgen: Die Pins B4, B5 und B7 müssen als Input konfiguriert sein, B6 jedoch als Output. Anschließend wird in das `SPCR`-Register, dem SPI-Control-Register, das SPI-Enable- und das SPI-Interrupt-Enable-Bit auf 1 gesetzt. Als nächstes wird in das `SPDR`-Register, dem SPI-Data-Register, der Wert 0x00 geschrieben. Abschließend werden mit einem Aufruf der Funktion `sei()` Interrupts aktiviert, da SPI am Besten mit Interrupts genutzt wird.

Wenn Daten empfangen werden, wird die SPI-Interrupt-Routine aufgerufen, welche wie folgt definiert wird: `ISR(SPI_STC_vect) { ... }`. Das empfangene Byte muss zuerst aus dem `SPDR`-Register ausgelesen werden, anschließend kann eigener Code ausgeführt werden. Um die Antwort auf die nächste Übertragung festzulegen, wird ein Byte in das `SPDR`-Register geschrieben. Die Interrupt-Routine sollte optimalerweise keine großen Verzögerungen enthalten, da sonst nicht genug Zeit für eine korrekte SPI-Antwort bleibt.

5.4 ADC

Jeder der 8 Pins der A-Gruppe ist mit dem Analog-Digital-Wandler verbunden, somit ist auf jedem Pin die Bestimmung der genauen Spannung möglich. Der Analog-Wandler funktioniert mithilfe einer internen Referenzspannung, welche langsam verändert wird und über einen Operationsverstärker ein digitales Signal auslöst. Es kann immer nur ein Kanal gleichzeitig ausgewertet werden, da alle Kanäle mit einem Multiplexer verbunden sind.

Die Initialisierung des ADC erfolgt durch folgende Schritte: Als Erstes müssen im `ADCSRA`-Register das ADC-Enable-, ADC-Interrupt-Enable-, ADC-Start-Conversion- und die drei ADC-Prescaler-Bits auf 1 gesetzt werden. Dies aktiviert den ADC und die dazugehörigen Interrupts und startet eine Messung. Die Taktrate des ADC wird über die drei Prescaler-Bits auf $\frac{20MHz}{128} = 156.25$ KHz gesetzt. Optional kann zuvor im `ADMUX`-Register, dem Multiplexer-Channel-Register, ein Channel ausgewählt werden. Um den Interrupt-Controller zu aktivieren, muss die Funktion `sei()` aufgerufen werden.

Sobald die Messung abgeschlossen ist, wird die ADC-Interrupt-Routine aufgerufen, welche mit `ISR(ADC_vect) { ... }` definiert wird. Dort müssen zuerst das `ADCH`- und `ADCL`-Register gelesen werden, in welchen das Ergebnis der Messung gespeichert ist. Nach der Ausführung von eigenem Code kann eine neue Messung gestartet werden, indem in das `ADCSRA`-Register das ADC-Start-Conversion- und das ADC-Interrupt-Flag-Bit auf 1 gesetzt werden. Im Gegensatz zum SPI-Interrupt dürfen in der Routine Verzögerungen verwendet werden.

5.5 PWM

Die Erzeugung von PWM-Signalen erfolgt auf dem ATMega mithilfe der internen Clocks, je nach einem Vergleichswert den Output auf HIGH oder auf LOW setzen. Der ATmega hat 6 Timer, auf dieser Platine werden nur die Timer 1A, 1B, 2A und 2B genutzt. Der Output des PWM-Signals erfolgt über die Pins D4 - D7, welche dafür als Output konfiguriert werden müssen.

Die Initialisierung des Timers 1 erfolgt durch das setzen der Output-Compare-Bits `COM1A0` und `COM1B0` sowie des `WGM0`-Bits im Timer 1 Control-Register `TCCR1A`. Im Register `TCCR1B` wird das `WGM2`- sowie das `CS1`-Bit auf 1 gesetzt. Dies hat zur Folge, dass der Timer im 8-Bit Fast-PWM Modus läuft und bei selben Wert wie im Timer-Compare-Register den dazugehörigen Pin an oder ausschaltet. Durch das `CS1`-Bit wird der Prescaler der PWm-Frequenz auf den Wert 8 gesetzt. Für Timer 2 ist der Vorgang fast ident, jedoch befindet sich das `WGM2`-Bit im `TCCR2A`-Register, statt in dem `TCCR2B`-Register.

Um nun ein PWM-Signal zu erzeugen, wird in das entsprechende Output-Compare-Register der Vergleichswert geschrieben. Für Channel 1A ist das Output-Compare-Register das `OCR1A`-Register. Der Wert 0 im Register resultiert in einem 0% Duty-Cycle, der Wert 255 in einem 100% Duty-Cycle. Bei der Verwendung von Motortreibern, wie den VNH7100ASTR, die zwei Richtungen unterstützen, müssen zusätzlich die Direction-Pins der Motortreiber richtig angesteuert werden. Dies erfolgt über das ein- oder ausschalten bestimmter Output-Pins des ATMega.

6 Raspberry Pi - ATmega

6.1 Protokoll

6.2 Automatischer Kommunikationstest

6.3 IO

6.4 ADC

6.5 Motor

7 Quellverzeichnis

7.1 Literaturverzeichnis

7.2 Bildverzeichnis

8 Fazit