RWTH AACHEN
III. Physikalisches Institut B

Bachelorarbeit

Entwicklung eines Strahlprofilscanners für JEDI

Alexander Krampe

Gutachter: Univ.-Prof. Dr. rer. nat. Jörg Pretz

Verfasser: Alexander Krampe
Adresse: Erkensgasse 19, 52353 Düren
Email: alexander.krampe@rwth-aachen.de

Abgabetermin: 01. Februar 2018

Inhaltsverzeichnis

Abbildungsverzeichnis iv

Tabellenverzeichnis vi

1 Motivation 1

2 Theoretischer Hintergrund 3

2.1 Szintillatoren . . . 3

2.2 Silicon photomultiplier . . . 5

3 Entwicklung des Scanners 10

3.1 Mechanik . . . 10

3.2 Hardware . . . 12

3.2.1 Allegro A4988 Schrittmotor Treiber IC . . . 12

3.2.2 Counter ICs . . . 15

3.2.3 Steuerung & Kommunikation der Komponenten . . . 19

3.3 Gehäuseanfertigung . . . 25

3.4 Software . . . 26

3.4.1 pigpio . . . 26

3.4.2 I^2C Bibliothek ATMega328p . . . 26

4 Steuerungsprogramm 27

4.1 Messparameter & Konfigurationsdatei Datei . . . 27

4.2 Hauptprogramm . . . 28

5 Test 29

6 Zusammenfassung 30

7 Abkürzungsverzeichnis 32

8 Danksagung 33

9 Anhang **35**

Literaturverzeichnis **58**

Abbildungsverzeichnis

2.1 Energielevel eines organischen Szintillators 4
2.2 Emissions- und Absorptionsspektrum eines organischen Szintillators . . . 5
2.3 Schematische Darstellung eines SiPMs 6
2.4 Dotierungsprofil einer Si-APD 6
2.5 Betriebszustände einer APD 7
2.6 Aufgenommenes SiPM Signal 8
2.7 Äquivalenter Schaltkreis einer Mikrozelle 8

3.1 Verwendete Aluminiumprofile zum Aufbau des Scanners 10
3.2 Befestigung des Gurtes an der Laufkatze 11
3.3 Ansicht des erstellten Verbindungselementes in OpenSCAD. 11
3.4 Nema 17 Schrittmotor 12
3.5 Allegro A4988 IC 13
3.6 Fertiger Scanner 14
3.7 Pin Layout ATMega 328P 15
3.8 Spannungssignal SiPM 15
3.9 Signalfluss 16
3.10 Interner Timer Aufbau 16
3.11 Interner RC Oszillator 17
3.12 Raspberry Pi 2 19
3.13 I^2C Bus open drain Betrieb 20
3.14 Start Stop Bedingung der Übertragung 21
3.15 Beispielhafte Übertragung 21
3.16 Fehlerhaftes clock stretching 23
3.17 Oszilloskopbild I^2C bug 23
3.18 Oszilloskopbild SCL 24
3.19 Oszilloskopbild Pullup Widerstand 25
3.20 Gehäuseansichten 26

4.1 Falsch gesetzter Messparameter 27

4.2 Hauptmenü des Kontrollprogramms 28

5.1 Test mit Bismuth Quelle 29

Tabellenverzeichnis

3.1 Übersichtstabelle zur Funktion der einzelnen Pins. 14
3.2 Eingestellte Frequenz am AWG gegenüber der von den Mikrocontrollern gemessenen Frequenz 19

1 Motivation

Eines der bis heute nicht gelösten Probleme der Physik ist es, die Asymmetrie zwischen Materie und Antimaterie im beobachtbaren Universum zu erklären. Mit Hilfe des Standardmodells der Teilchenphysik lässt sich zwar eine Asymmetrie erklären, allerdings liegen die Vorhersagen des Standardmodells acht Größenordnungen unter dem beobachteten Wert der Baryon Asymmetrie η, für welchen gilt (vgl. (5), S. 4)

$$5.7 \cdot 10^{-10} \leq \eta = \frac{n_B - n_{\overline{B}}}{n_\gamma} \leq 6.7 \cdot 10^{-10}.$$

Hierbei bezeichnen n_B und $n_{\overline{B}}$ die Baryonen- bzw. Antibaryonendichte und n_γ die Photonendichte.

Baryogenese Modelle versuchen zu erklären, wie diese Asymmetrie, zwischen Materie und Antimaterie, aus einem anfangs symmetrischen Universum entstanden sein könnte. All diese Modelle müssen den von Sakharov aufgestellten Bedingungen der Baryogenese genügen (vgl. (5), S. 1).

1. Verletzung des Erhalts der Baryonenzahl B
2. Fehlen von thermischem Gleichgewicht
3. Verletzung der C und der CP Symmetrie

Elektrische Dipolmomente (EDMs) bieten einen direkten experimentellen Zugang zur CP Verletzung. Ein solches EDM würde die Parität P und die Zeitumkehr T verletzen. Eine Verletzung der Zeitumkehr T ist aber, unter der Annahme, dass das Produkt CPT invariant ist, äquivalent zu einer CP Verletzung.
Zur Messung eines EDMs $\vec{d}$ wird dazu die Änderung des Spins

$$\frac{d\vec{S}}{dt} = \vec{d} \times \vec{S}$$

gemessen. Allgemein folgt die Änderung des Spins in Speicherringen für vertikales Magnetfeld und radiales elektrisches Feld der Thomas BMT Gleichung (vgl. (21), S. 19)

$$\frac{d\vec{S}}{dt} = -\frac{q}{m_0}\left(G\vec{B} + \left(\frac{1}{\gamma^2 - 1} - G\right)\frac{\vec{\beta}\times\vec{E}}{c} + d\frac{m_0}{q\hbar S}(\vec{E} + c\vec{\beta}\times\vec{B})\right)\times\vec{S},$$

wobei G das anormale magnetische Moment, γ den Lorentz Faktor und d das elektrische Dipolmoment beschreibt. Im Fall von reinen magnetischen Speicherringen ($\vec{E} = 0$) vereinfacht sich die Thomas BMT Gleichung zu

$$\frac{d\vec{S}}{dt} = -\frac{q}{m_0}\left(G\vec{B} + d\frac{m_0}{q\hbar S}(c\vec{\beta}\times\vec{B})\right)\times\vec{S}.$$

Im Speicherring COSY, der einen rein magnetischen Speicherring darstellt, führt ein nicht verschwindenes EDM zu einem vertikalen Aufbau des Spins, welcher durch ein Polarimeter gemessen werden kann. Die vorliegende Arbeit beschreibt die Entwicklung eines Strahlprofilscanners mit hoher Zählrate, dessen Aufgabe es ist, ein Intensitätsprofil des Strahls zu erstellen. Dieses Intensitätsprofil kann anschließend für den Betrieb des Polarimeters genutzt werden.

2 Theoretischer Hintergrund

In diesem Kapitel sollen die theoretisch notwendigen Hintergründe vermittelt werden, die zum Verständnis des Strahlprofilscanners relevant sind. Detailliert soll dabei auf die Funktion und Arbeitsweise der verwendeten *silicon photo multiplier* (SiPMs) eingegangen werden.

2.1 Szintillatoren

Als Szintillatoren bezeichnet man Materialien, die nach der Absorption von Photonen oder geladenen Teilchen den Effekt der *Szintillation* zeigen, d.h. selbst Photonen emittieren, nachdem Moleküle im Szintillator durch die Absorption angeregt wurden. In der folgenden Betrachtung der Funktionsweise wird sich auf organische Szintillatoren beschränkt, da diese zur Konstruktion des Strahlprofilscanners benutzt wurden. Die Funktionsweise von anorganischen Szintillatoren wird beispielsweise in (9) beschrieben.
Zur Emission von Photonen des Szintillators tragen drei Prozesse bei: Zum einen die *Floureszenz* und zum Anderen die *Phosphoreszenz*. Der dritte Prozess ist die *verzögerte Floureszenz*. Die Zeit, die die Differenz zwischen Absorption von Photonen oder geladenen Teilchen und Emission von Photonen des Szintillators angibt, charakterisiert um welchen Prozess es sich handelt. Als Floureszenz wird die schnelle Emission von Photonen bezeichnet. In den meisten organischen Szintillatoren findet diese Emission nach wenigen Nanosekunden statt. Im Unterschied dazu beschreibt die Phosphoreszenz die verzögerte Emission von Photonen einer größeren Wellenlänge. Bei der verzögerten Floureszenz wird Licht der gleichen Wellenlänge, wie es bei bei der normalen Floureszenz auftritt, emittiert, jedoch tritt die Emission auch hier verzögert auf.

Die Ursache der Lichtemission in organischen Szintillatoren ist die Anregung von Elektronen in Singulett oder Triplett Zustände nach der Absorption von Photonen oder geladenen Teilchen.

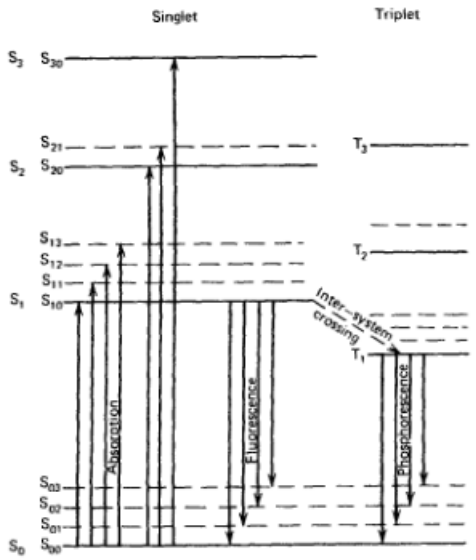


Abb. 2.1: Energielevel eines organischen Szintillators: Die Länge der aufwärts gerichteten Pfeile ist ein Maß für jene Photonenenergien, die stark absorbiert werden. (Quelle: (8), S. 221)

.

Höhere Singulett Zustände wie S_2 oder S_3 gehen durch innere Konversion in den S_1 Zustand über. Die in Abb. 2.1 gezeigten Energielevel der Singulett Zustände (S_0, S_1, ...) und der Triplett Zustände (T_0, T_1, ...) zeigen eine feinere Unterteilung in Schwingungszustände, die mit einem weiteren Index kenntlich gemacht werden. Zustände, die sich nicht im Schwingungsgrundzustand, z.B. in S_{12}, befinden, sind nicht im thermischen Gleichgewicht mit ihren Nachbarn und gehen schnell in einen Schwingungsgrundzustand über, so dass sich die Moleküle in einem organischen Szintillator nach der Anregung nach kurzer Zeit im Zustand S_1 befinden. Von diesem gehen sie schließlich unter Emission von Floureszenzlicht in einen der Grundzustände $S_{00}, S_{01}, ...$ über. Für die Intensität des emittierten Lichts gilt

$$I = I_0 e^{-t/\tau},$$

wobei τ die Zeit für einen Übergang $S_{10} \rightarrow S_{0n}$ angibt.

Anhand der Abb. 2.1 lässt sich auch erklären, warum organische Szintillatoren die durch Floureszenz emitierten Photonen nur zu einem kleinem Teil wieder selbst absorbieren: Da die Länge der aufwärts gerichteten Pfeile ein Maß für solche Photonenenergien ist, die stark absorbiert werden, und die nach unten gerichteten Pfeile der Floureszenz alle (mit Ausnahme des Übergangs $S_{10} \rightarrow S_{00}$) eine kleinere Länge aufweisen als für

die Anregung notwendig ist, überlappen sich das Emissions- und Absorptionsspektrum eines organischen Szintillators nur geringfügig.

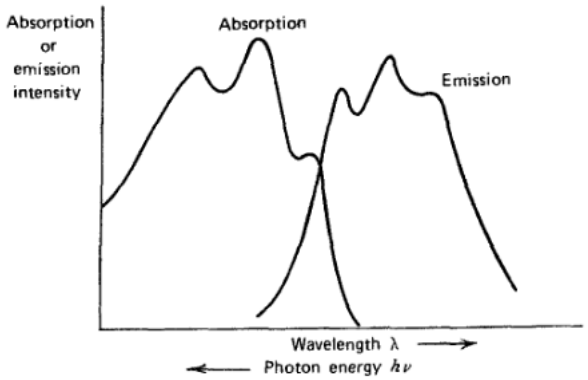


Abb. 2.2: Emissions- und Absorptionsspektrum eines organischen Szintillators: Aufgrund der Verschiebung der beiden Spektren wird nur ein kleiner Teil des vom Szintillator emittierten Lichtes erneut absorbiert. (Quelle: (8), S. 222)

Bei der verzögerten Floureszenz hingegen werden Moleküle, die sich in Triplett Zuständen befinden, durch thermische Anregung erneut in den Zustand S_1 angeregt (vgl. (9), S. 162):

$$T_0 + T_0 \rightarrow S^* + S_0 + \text{Phononen}.$$

Von dort gehen sie durch Floureszenz in den Grundzustand S_0 über.

2.2 Silicon photomultiplier

Silicon photomultiplier (SiPMs) sind Halbleiterdetektoren, die die Lichtpulse der Szintillatoren in einen messbaren Strom umwandeln. SiPMs bestehen dazu aus einer Vielzahl spezieller Photodioden, sog. *avalanche photo diodes* (APDs), und sog. *quenching Widerständen* R_Q.

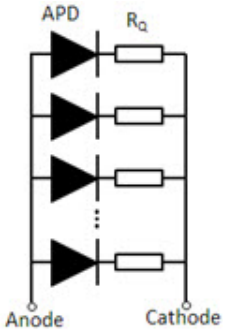


Abb. 2.3: Schematische Darstellung eines SiPMs: Alle APDs sind parallel geschaltet mit einem dazu in Reihe geschalteten Widerstand R_Q. (Quelle: (14))

Die Reihenschaltung aus APD und quenching Widerstand R_Q wird als Mikrozelle bezeichnet. SiPMs können, je nach Größe, zwischen hunderten und mehreren zehntausenden solcher Mikrozellen besitzen. Um nun Strahlung zu detektieren werden die APDs in Sperrrichtung geschaltet. Dies führt zu einer Vergrößerung der Verarmungszone innerhalb der APDs. Im Unterschied zu „klassischen“ Photodioden weisen APDs ein anderes Dotierungsprofil auf, da APDs - im Unterschied zu diesen - die in der Verarmungszone erzeugten Ladungsträger vervielfachen sollen.

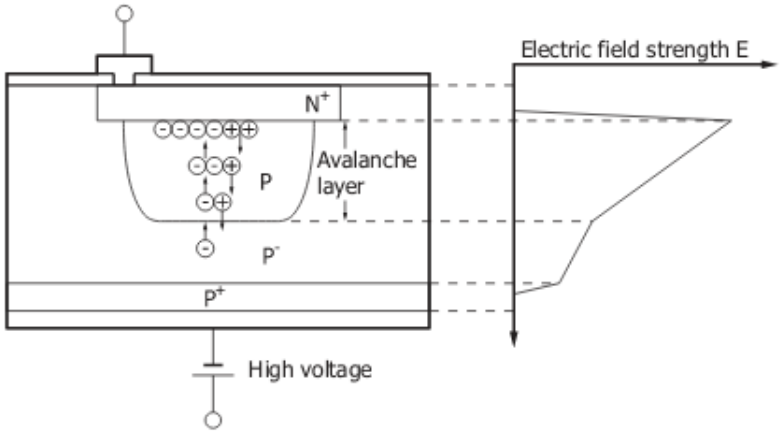


Abb. 2.4: Typisches Dotierungsprofil einer Si-APD mit elektrischer Feldstärkeverteilung (Quelle: (12), S. 3)

Wird ein Photon in der p^- Schicht absorbiert erzeugt es dort Elektronen-Loch Paare. Die so erzeugten Elektronen driften zunächst in die Avalancheegion und anschließend in die Multiplikationszone, welche durch den p und n^+ Übergang erzeugt wird. Aufgrund der hohen elektrischen Feldstärke in der Multiplikationszone werden die Elektronen stark beschleunigt und können bei Kollision mit Gitteratomen weitere Elektronen-Loch-Paare

erzeugen, welche ihrerseits wiederum weitere Elektronen-Loch-Paare erzeugen können. Dieser Effekt wird *Avalanche-Effekt* genannt.

Um die Funktion des Widerstandes R_Q zu verstehen, ist es zunächst hilfreich die verschiedenen Betriebszustände in Abb. 2.5 einer APD zu betrachten.

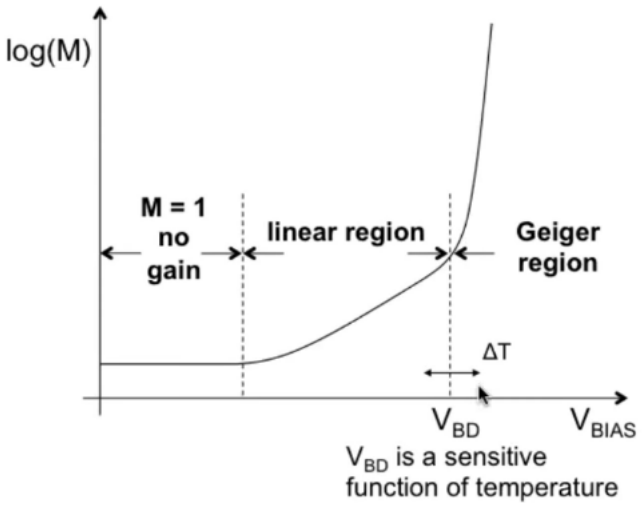


Abb. 2.5: Betriebszustände einer APD in Abhängigkeit der an ihr angelegten Spannung V_{Bias}. M bezeichnet hier die Verstärkung. (Quelle: (13))

Wird eine kleine (Rückwärts) Spannung an die APD gelegt, so findet keine Verstärkung statt ($M = 1$). Die APD verhält sich hier wie eine normale Photodiode. Bei einer weiteren Erhöhung von V_{Bias} wird nun eine Verstärkung $M > 1$ beobachtet und die Abhängigkeit von $log(M)$ ist nahezu linear bis V_{Bias} die *Durchbruchsspannung* (breakdown voltage) V_{BD} erreicht. In dem Intervall, wo die Verstärkung sich nahezu linear verhält, kommt jede Vervielfachung von Ladungsträgern von selbst aus zum Erliegen. Erhöht man die Spannung nun wenige Volt über V_{BD}, befindet sich die APD im Geiger Modus. In diesem Betriebszustand kommt die Ladungsträgervervielfachung nicht mehr von selbst aus zum Erliegen und würde unendlich lange andauern, so dass die APD für eine weitere Detektierung von Photonen nicht länger zur Verfügung stände. Die Funktion des Widerstandes R_Q ist nun, den Strom zum Erliegen zu bringen in die APD in der Geiger Modus zurückzuversetzen.

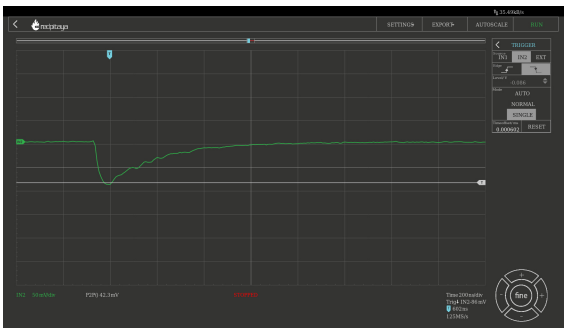


Abb. 2.6: Aufgenommenes SiPM Signal: Zu erkennen sind die unterschiedlichen Anstiegs- und Abstiegszeiten des Signals.

Die Form des Signals lässt sich mit Hilfe von Abb. 2.7 erklären.

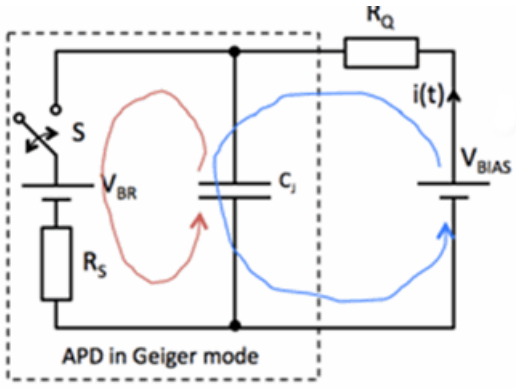


Abb. 2.7: Äquivalenter Schaltkreis einer Mikrozelle: Die APD besteht aus einem Kondensator C_j, der die Kapazität der Verarmungszone repräsentiert, eines Widerstandes R_S, der den Widerstand der Avalanche Region angibt (vgl. Abb. 2.4), einer Spannungsquelle V_{BD} und einem Schalter S. Die Pfeile symbolisieren den fließenden Strom. (Quelle: (14))

Wird ein Photon absorbiert und gelangt das dabei entstandene Elektron in die Avalanche Region schließt sich der Schalter und der Kondensator C_j entlädt sich über den Widerstand R_S, beschrieben durch den roten Pfeil. Dieser Strom ist ein interner der APD und kann nicht gemessen werden. Der interne Stromfluss verursacht nun einen Spannungsabfall über dem Widerstand R_Q und der Strom, der durch den blauen Pfeil symbolisiert wird, beginnt zu fließen. Aus diesem Grund wird ein Strom der Form

$$I(t) \propto 1 - \exp\left(\frac{-t}{R_S \cdot C_j}\right) \tag{2.1}$$

beobachtet (vgl. das fallende Signal Abb. 2.6). Im Minimum der gilt nun, dass Spannungsabfall über R_Q so groß ist, dass über der APD eine Spannung nahe von V_{BR} abfällt.

Für den Strom I gilt im Minimum

$$I = \frac{V_{Bias} - V_{BR}}{R_S + R_Q}. \tag{2.2}$$

Ist der Spannungsabfall über der APD jedoch nahe V_{BR}, öffnet sich Schalter wieder und der Strom, der durch den roten Pfeil gekennzeichnet wird, verschwindet. An diesem Punkt erreicht der Strom im blau gekennzeichneten Kreis sein Minimum und lädt den Kondensator C_j erneut über R_Q auf V_{Bias} auf. Die in Abb. 2.6 steigende Flanke hat deshalb die Form

$$I(t) \propto \exp\left(\frac{-t}{R_Q \cdot C_j}\right). \tag{2.3}$$

3 Entwicklung des Scanners

In diesem Kapitel wird auf den Aufbau und die Entwicklung des Strahlprofilscanners eingegangen. Dabei sollen auch die während der Entwicklung aufgetretenen Schwierigkeiten und deren Lösung behandelt werden.

3.1 Mechanik

Zum Aufbau des Strahlprofilscanners wurden zwei 250 mm lange Aluminiumprofile mit aufgesetzter Laufkatze verwendet. An einem Ende der Aluminiumprofile befindet sich ein Schrittmotor, der die Laufkatze in Bewegung versetzen kann.

Abb. 3.1: Verwendete Aluminiumprofile zum Aufbau des Scanners: Die Laufkatze wird durch den am hinteren Ende sitzenden Schrittmotor in Bewegung versetzt. (Quelle: (10))

An der Laufkatze selbst befindet sich dazu ein Gurt, der über eine Umlenkrolle mit dem Schrittmotor verbunden ist.

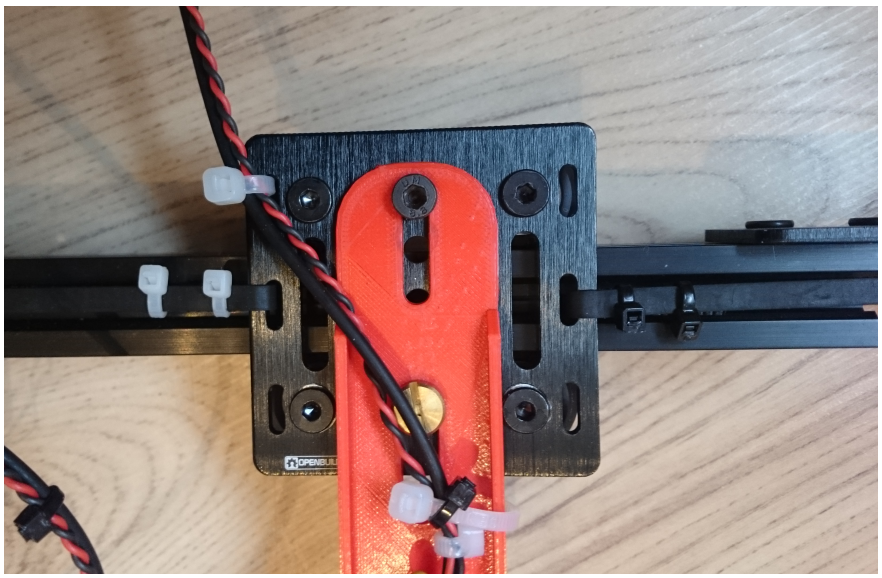


Abb. 3.2: Befestigung des Gurtes an der Laufkatze. Rot: Halterung des Szintillators.

Die zwei Aluminiumprofile wurden nun rechtwinklig zueinander miteinander verbunden. Dazu wurde mit Hilfe des Programms OpenSCAD ein Verbindungselement erstellt und im 3D Drucker gedruckt.

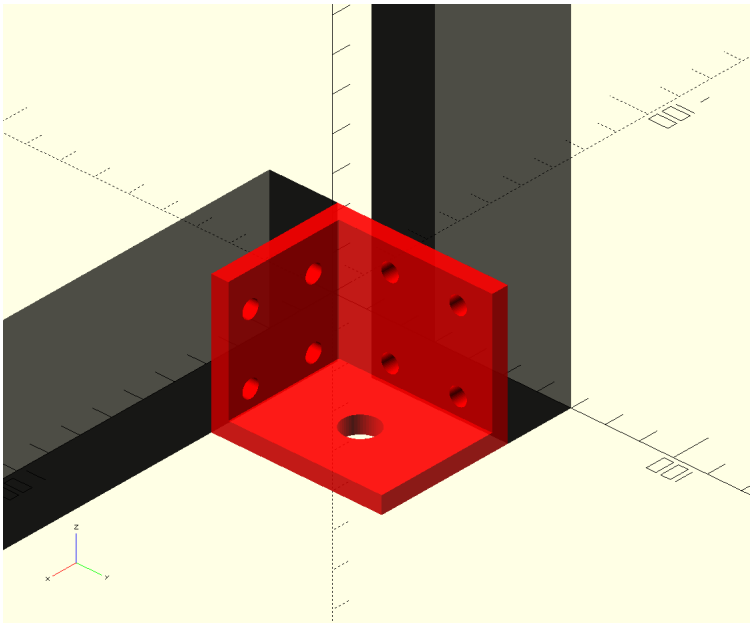


Abb. 3.3: Ansicht des erstellten Verbindungselementes in OpenSCAD.

Zur Halterung der Szintillatoren auf der Laufkatze wurden ebenfalls Halterungen in OpenSCAD erstellt.

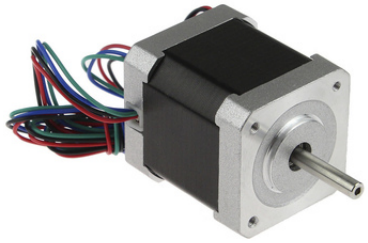


Abb. 3.4: Verwendeter Nema 17 Schrittmotor (Quelle: (7))

Schrittmotoren besitzen gegenüber Gleichstrommotoren ein wesentlich höheres Drehmoment bei geringeren Geschwindigkeiten (vgl. (17), S. 938f) und eignen sich somit besser für Anwendungen, in denen eine präzise Positionierung bei gleichzeitig kleiner Geschwindigkeit notwendig ist. Die verwendeten Schrittmotoren besitzen eine Auflösung von 1.8 ° pro Schritt (vgl. (11), S. 1)

Um die Szintillatoren in eine bekannte Position zu fahren, befindet sich jeweils am Ende eines jeden Aluminiumprofils ein Mikroschalter. Sobald dieser durch das Gegenfahren eines Szintillators geschlossen wird, wird dessen Bewegung gestoppt.

3.2 Hardware

In diesem Abschnitt wird die verwendete Hardware beschrieben.

3.2.1 Allegro A4988 Schrittmotor Treiber IC

Zur Ansteuerung und Kontrolle der Schrittmotoren wurde das IC Allegro A4988 der Firma Polulu verwendet. Abb. 3.5 zeigt das IC sowie die Funktion der jeweiligen Pins.

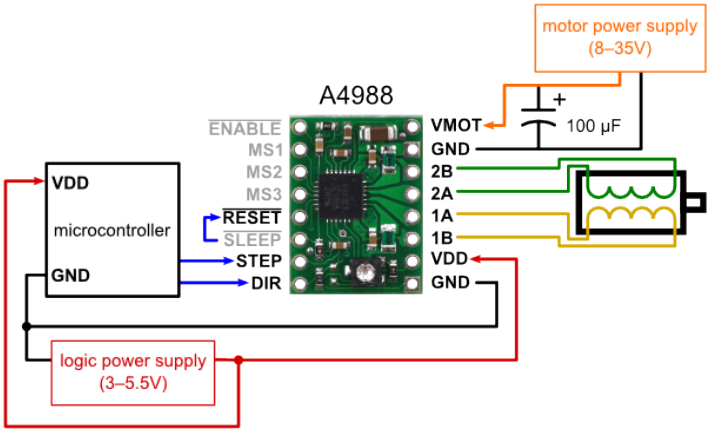


Abb. 3.5: Allegro A4988 IC mit dargestellter Funktion der einzelnen Pins. (Quelle: (15))

Das IC benötigt zwei Versorgungsspannungen, wovon eine für die Logik des ICs (V_{DD}), die andere für die Versorgung der Schrittmotors vorgesehen ist (V_{MOT}). Die Pins $1A$, $1B$, $2A$ und $2B$ werden direkt mit den Schrittmotor verbunden. Da jeweils die Anschlüsse $1A$ und $1B$ bzw. $2A$ und $2B$ des ICs mit einer Phase des Schrittmotors verbunden sind, muss zunächst überprüft werden, welche der vier Anschlüsse des Motors miteinander verbunden sind, was einfachsten mit einer Widerstandsmessung zwischen jeweils zwei der Anschlüsse überprüft werden kann.
Mit Ausnahme der für die Versorgungsspannungen vorgesehenen Pins, erwarten alle übrigen Pins logische Signale. Die folgende Tabelle gibt eine Übersicht der Funktion der einzelnen Pins (vgl. (2), S. 7 - 10).

Tabelle 3.1: Übersichtstabelle zur Funktion der einzelnen Pins.

Pin	Funktion
$\overline{ENABLE}$	Deaktiviert oder aktiviert alle Ausgänge
MSx	Aktiviert *microstepping*
DIR	Setzt die Rotationsrichtung des Motors
STEP	Eine Änderung von logisch 0 auf logisch 1 lässt den Motor einen Schritt machen
$\overline{SLEEP}$	Setzt IC in Schlafmodus und deaktiviert die Ausgänge, Ladepumpe und Stromregler
$\overline{RESET}$	Setzt das IC in einen vordefinierten Zustand zurück

Zur konkreten Ansteuerung des ICs sei auf Unterabschnitt 3.2.3 verwiesen. Die in Abschnitt 3.1 beschriebene Winkelauflösung der Motoren von $1,8\,^\circ$ kann unter Aktivierung der MSx Pins weiter verbessert werden. Werden alle MSx Pins des ICs genutzt, so verbessert sich die Winkelauflösung auf $\frac{1.8}{16}\,^\circ$, jedoch verlängert sich somit auch die Dauer des Scans um das 16-fache. Weitere Zwischenstufen sind möglich und können aus dem Datenblatt entnommen werden. Abb. 3.6 zeigt den fertigen Scanner.

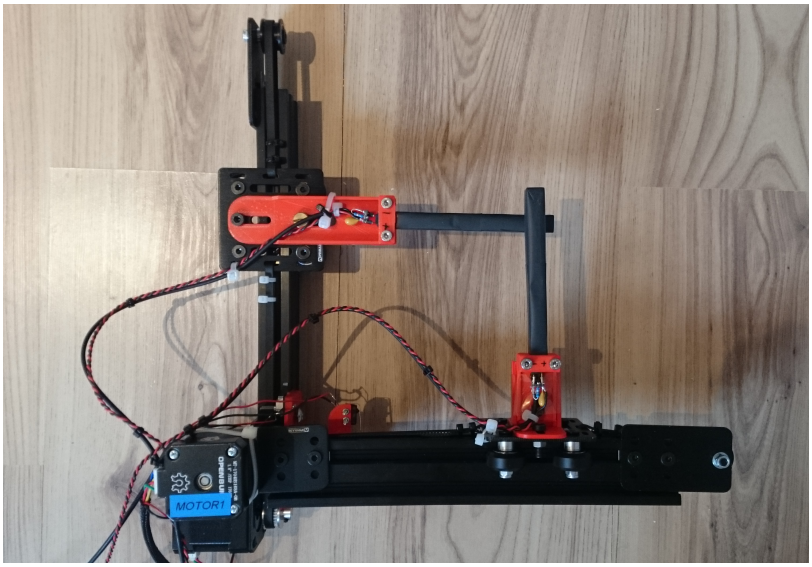
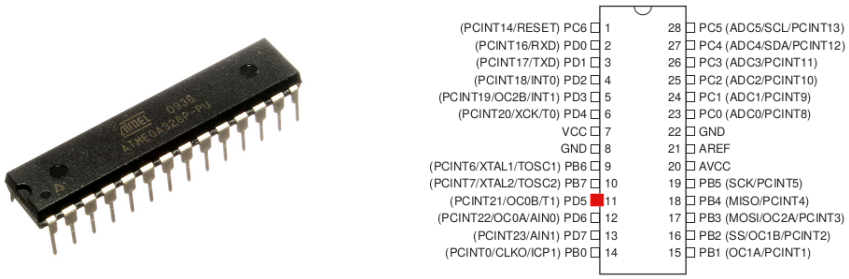


Abb. 3.6: Fertiger Scanner

3.2.2 Counter ICs

Zur Aufnahme der von den SiPMs stammenden Pulse wurden zwei Mikrocontroller vom Typ ATMega328P der Firma Microchip verwendet.



(a) Verwendete Mikrocontroller ATMega328P zum Zählen der Signale (Quelle: (20))

(b) Pin Layout: Rot markiert ist der 16bit Timer/Counter Pin. (Quelle: (4), S. 3)

Abb. 3.7: Bild und Darstellung des Pin Layouts der verwendeten Mikrocontroller zum Zählen der SiPM Pulse.

Das in Abb. 2.6 gezeigte Signal wurde dabei über einen 50 Ω Widerstand am SiPM abgegriffen

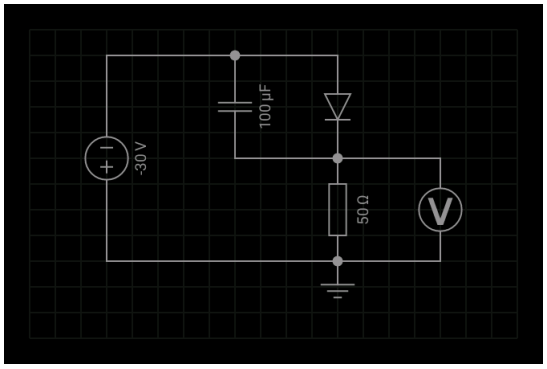


Abb. 3.8: Das gemessene Spannungssignal der SiPMs wurde über einen 50 Ω Widerstand abgegriffen. Dieser ist gerade so gewählt, dass er der Impedanz des verwendeteten LEMO Kabels hin zum Diskriminator entspricht.

Das abgegriffene Signal kann jedoch nicht direkt zum PD5 Pin des Mikrocontrollers geführt werden, welcher mit der Counter Unit des Mikrocontrollers verbunden ist, sondern muss zunächst aufbereitet werden.

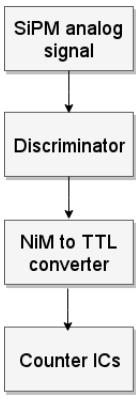


Abb. 3.9: Signalfluss: Die Signale der SiPMs gelangen zunächst in einen Diskriminator und anschließend in einen NIM zu TTL Konverter.

Der Grund zur Wandlung der Signale in TTL liegt im internen Aufbau der Counter Unit begründet.

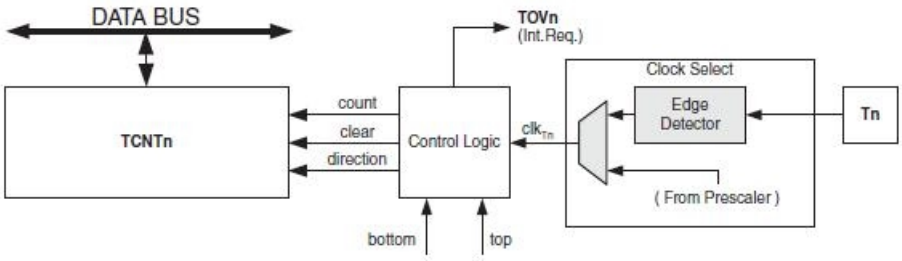
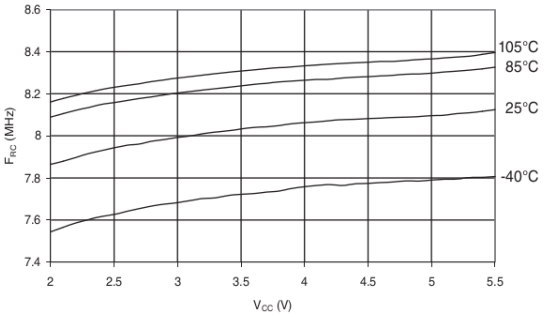


Abb. 3.10: Das Eingangssignal auf Tn (T1) wird an einen Edge Detector weitergeleitet, welcher ein Signal an eine control unit sendet, sofern er triggert.

Soll der Edge Detector einen Trigger auslösen, muss das Signal eine Änderung des logischen Pegels am Pin hervorrufen, was jedoch mit NIM kompatiblen Signalen nicht möglich ist. Gemäß Datenblatt der Mikrocontroller wird garantiert, dass alle Spannungen V zwischen $0.6V_{CC}$ und $V_{CC} + 0.5\,\mathrm{V}$ als logisch 1, und alle Spannungen V zwischen $-0.5\,\mathrm{V}$ und $0.3V_{CC}$ logisch 0 interpretiert werden (vgl. (4), S. 313), wobei $V_{CC} = 5\,\mathrm{V}$ die Versorgungsspannung bezeichnet.

Um möglichst präzise die eintreffenden Pulse zu zählen, ist hardwareseitig ein sehr genauer Taktgeber erforderlich, während softwareseitig die Ausführung des Codes so hoch wie möglich sein sollte. Da die Frequenz des internen RC Oszillators temperaturabhängig ist und dieser nicht kalibriert ist, wurde stattdessen ein externer 20 MHz Quartzkristall verwendet.



(a) Spannungs- und temperaturabhängigkeit des internen RC Oszillators (Quelle: (4), S. 606)

(b) Verwendeter Quartzkristall

Abb. 3.11: Zur Verbesserung der der Zählgenauigkeit wurde der interne RC Oszillator durch einen 20 MHz Quartz ersetzt.

Der Quartzkristall wird dazu an den XTAL1 und XTAL2 Pin des Mikrocontrollers angeschlossen (vgl. Abb. 3.7). Damit der Quartzkristall stabil arbeitet ist es notwendig, zwei externe Kondensatoren mit den XTAL Pins und GND zu verbinden, da die Frequenz des Kristalls abhängig ist von allen Kapazitäten, die parallel mit dem Kristall verbunden sind. All diese parallelen Kapazitäten werden als *Lastkapazität* C_L (load capacitance) bezeichnet. Der verwendete Quartzkristall erwartet gemäß Datenblatt eine Lastkapazität von $C_L = 18\,\mathrm{pF}$. Schätzt man die Streukapazitäten der Schaltung auf $5\,\mathrm{pF} - 10\,\mathrm{pF}$[1], so müssten $8\,\mathrm{pF} - 13\,\mathrm{pF}$ durch die externen Kondensatoren zur Verfügung gestellt werden. Bei symmetrischem Layout, d.h. die beiden Kondensatoren haben die gleiche Kapazität C, ergibt sich deren Gesamtkapazität durch $\frac{1}{2}C$.

Somit kommen Werte von $16\,\mathrm{pF} - 26\,\mathrm{pF}$ in Frage. Für die Praxis hat sich ein Wert von 22 pF als hinreichend genau erwiesen. Zuletzt muss dem Mikrocontroller noch mitgeteilt werden, dass dieser eine externe Clock nutzen soll. Dazu muss man entsprechende Bits[2] im *fuse low byte* (lfuse), *fuse high byte* (hfuse) und im *fuse extended byte* (efuse) setzen, was sich mit dem im Makefile erstellten Target *set_ crystal_fuses* bequem erledigen lässt.

Bei der Erstellung des des Programms in C, welches die Pulse der SiPMs zählen und speichern soll, ist darauf zu achten, dass der Code möglichst schnell läuft, um die eintreffenden Pulse so genau wie möglich zu zählen und um die Zahl der maximal pro Sekunde erfassbaren Pulse so groß wie möglich zu halten.

[1] Dies sind typische Werte gemäß AVR042: AVR Hardware Design Considerations (S. 8)

[2] um herauszufinden, welche das konkret sind bietet sich ein fuse calculator wie z.B. `http://www.engbedded.com/fusecalc/` (Stand: 21.01.2018) an

Das Zählen der Pulse übernimmt hierbei die in Abb. 3.10 dargestellte counter unit des Mikrocontrollers, die das TCNT1 Register mit der Anzahl der gemessenen Pulse automatisch aktualisiert. Prinzipiell wäre es auch möglich, das Zählen und Aktualisieren der Pulse softwareseitig zu lösen, wie das folgende Beispiel illustriert.

Listing 3.1: Beispiel für softwareseitiges Zählen der Pulse

```
1  while(t < t_max)
2  {
3    if(PinStateHasChanged())
4      ++count;
5  }
```

Dieser Ansatz des zyklischen Abfragens, auch als Polling bezeichnet, ist auch mehreren Gründen ungeeigneter als das Zählen über die interne counter unit:

1. Werden Pulse nur gezählt, falls überprüft wird, ob eine logische Pegeländerung des Pins stattgefunden hat. Vor und nach der Überprüfung werden eingehende Pulse nicht registriert. Dies begrenzt die maximal pro Sekunde zählbaren Pulse zusätzlich.

2. Sobald die Zeit t_max erreicht wurde, sollte das Zählen unmittelbar gestoppt werden.

Wird im obigen Beispielcode maximal ein Puls zu viel gezählt, falls sich t > t_max unmittelbar vor der if-Abfrage ergibt, so ändert sich die Situation, falls die interne counter unit das Zählen übernimmt: Durch branching in den Schleifenkopf und Auswertung der Laufbedingung ist es nun möglich, dass mehr als ein zusätzlicher Puls gezählt wird. Die Strategie um das Zählen möglichst genau dann abzubrechen, wenn t_max erreicht ist, besteht darin, die CPU einen Interrupt auslösen zu lassen, das Problem also auch hier durch die Hardware zu lösen. Dazu definiert man eine *interrupt service routine*, die aufgerufen wird, falls t_max erreicht wurde und das Zählen beendet. Der vollständige Code dazu findet sich im Anhang (vgl. Listing 9.2).

Zum Testen der Genauigkeit der Counter ICs wurde mit einen *arbitrary waveform generator* (AWG) ein TTL Signal auf die Eingänge gegeben. Der AWG selbst weist gemäß Datenblatt eine Unsicherheit von ±1 ppm auf die Frequenz auf mit einem jährlichen Drift von weiteren ±1 ppm (vgl. (18), S. 7), was eine Unsicherheit von ±4 ppm liefert.

Tabelle 3.2: Eingestellte Frequenz am AWG gegenüber der von den Mikrocontrollern gemessenen Frequenz

Eingestellte Frequenz / Hz	Gemessene Frequenz / Hz
10^2	101
10^3	1000
10^4	10039
10^5	100390
10^6	1003914
$6 \cdot 10^6$	6007492

3.2.3 Steuerung & Kommunikation der Komponenten

Zur Ansteuerung der Schrittmotoren und zur Konfiguration und zum Auslesen der Counter ICs wurde ein Raspberry Pi 2 verwendet. Hierbei handelt es sich um einen Einplatinencomputer, der über seine *general purpose input/output* (GPIO) Schnittstelle mit peripheren Geräten verbunden werden kann.

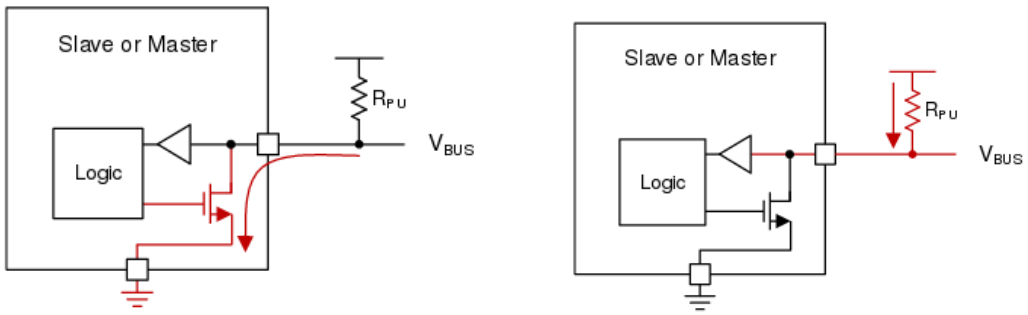
Abb. 3.12: Ein Raspberry Pi 2 übernimmt die Ansteuerung der Schrittmotoren und der Counter ICs (Quelle: (16)) .

Erwarten die in Abb. 3.5 beschrieben ICs zur Schrittmotorsteuerung nur logische Signale an ihren Eingängen, so ist die Situation bei den Counter ICs ungleich komplizierter, denn zwischen Raspberry Pi und diesen müssen bidirektional Daten ausgetauscht werden, um in der einen Richtung die gemessenen Pulse an den Raspberry und in der anderen Richtung die Messzeit an die Mikrocontroller zu übermitteln. Zur Kommunikation zwischen den Mikrocontrollern und dem Raspberry Pi wurde der I^2C Datenbus

verwendet.

I²C Datenbus

Der I²C Datenbus ist ein bidirektionaler Datenbus, der einen *Master* benutzt (Raspberry Pi), um mit *Slaves* (ATMega328P Mikrocontroller) zu kommunizieren. Um zwischen verschiedenen Slaves unterscheiden zu können, besitzen diese verschiedene Adressen.
Zur Kommunikation werden zwei Leitungen, die Taktleitung *serial clock line* (SCL) und die Datenleitung *serial data line* (SDA) benötigt, die beide über Pullup Widerstände mit der Versorgungsspannung des Raspberry Pis von 3.3 V verbunden sind. Die beiden Leitungen werden als *open drain* Leitungen betrieben.



(a) Busleitung wird auf logisch 0 gezogen

(b) Busleitung wird „losgelassen“ und durch den Pullup R_{PU} auf logisch 1 gehalten

Abb. 3.13: Darstellung der open drain Betriebsweise am I²C Bus (Quelle: (19), S. 2f)

Aufgrund der open drain Betriebsweise ist es somit nicht möglich, dass Geräte gleichzeitig eine logische 1 und eine logisch 0 senden. Im Fall, dass der Master Daten an den Slave sendet folgt die Kommunikation dabei folgendem Schema:

1. Der Master sendet eine Start Bedingung mit anschließender 7 Bit Slave Adresse. Das achte Bit signalisiert, ob der Master Daten zum Slave senden soll oder Daten vom Slave empfangen soll.
2. Der Master sendet eine beliebige Anzahl an Bytes an den Slave.
3. Der Master sendet eine Stop Bedingung an den Slave.

Soll der Slave hingegen Daten an den Master senden, so sendet der Master zusätzlich nach 1. Schritt noch die Adresse desjenigen Registers im Slave, aus dem die Daten gelesen werden sollen. Dann sendet der Slave die entsprechenden Daten zum Master und

der Master beendet die Übertragung mit der Stop Bedingung. Die Start und Stop Bedingungen sind in Abb. 3.14 dargestellt.

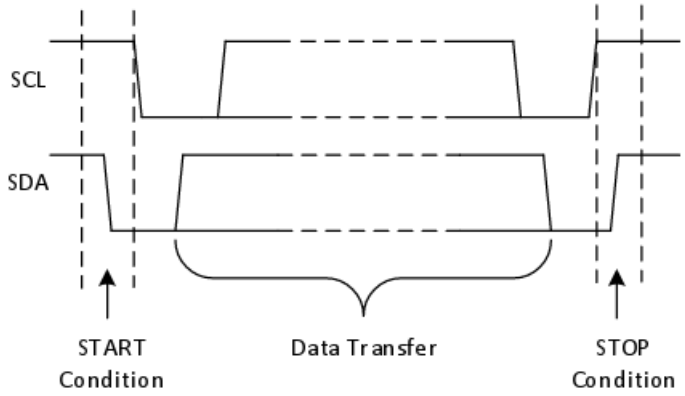


Abb. 3.14: Start und Stop Bedingung: Die Start Bedingung wird eingeleitet durch eine Änderung von logisch 1 auf logisch 0 von SDA während SCL auf logisch 1 ist. Ändert sich SDA von logisch 0 auf logisch 1 während SCL auf logisch 1 ist, so stellt dies die Stop Bedingung dar (Quelle: (19), S. 4).

Abb. 3.15 zeigt beispielhaft eine Übertragung eines einzelnen Bytes.

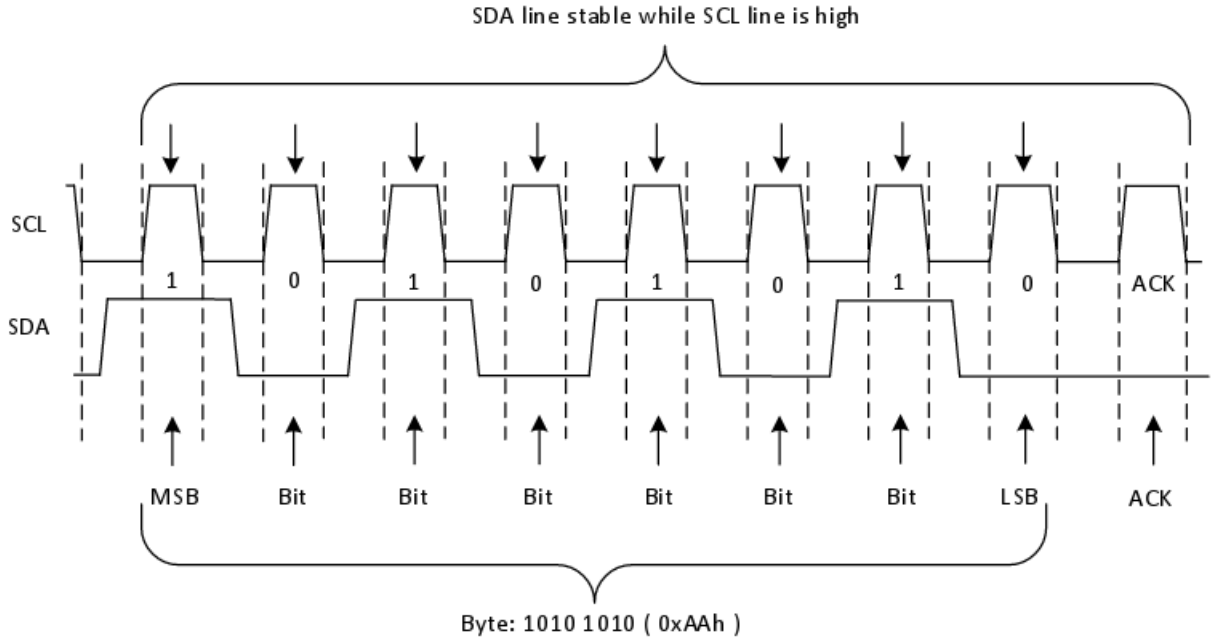


Abb. 3.15: Beispiel für die Übertragung des Bytes 10101010: Das übertragene Byte kann ermittelt werden, indem man das logische level von SDA ermittelt, während SCL auf logic high ist (Quelle: (19), S. 5).

Nach der Übertragung eines Bytes signalisiert der Empfänger dem Sender, ob das übertragene Byte erfolgreich oder nicht erfolgreich übertragen wurde durch das Setzen des *(not)acknowledge* Bits ((N)ACK). Slaves ist es darüber hinaus erlaubt sog. *clock stretching* zu betreiben: Dabei halten die Slaves SCL auf logisch 0, um die Kommunikation zwischen Master und Slaves zu verlangsamen.

BCM2835 hardware bug

Während der Testphase der I²C Kommunikation kam es in unregelmäßigen Abständen zu fehlerhaften Übertragungen, die weder durch eine Überprüfung des Codes noch durch Setzen einer anderen Übertragungsgeschwindigkeit im Gerätebaum[3] des entsprechenden Chips BCM2835 behoben werden konnten. Die Ursache der fehlerhaften Übertragungen liegen bei dem für die Kommunikation zuständigen Chip BCM2835: Dieser unterstützt das im I²C Standard definierte clock stretching ausschließlich unmittelbar nach der ACK Phase der Kommunikation mit der zusätzlichen Voraussetzung, dass die Taktleitung SCL um mehr als 0.5 Taktzyklen auf logic low gehalten wird. [4]

Im SCL Signal finden sich daraufhin zu kurze Taktzyklen wieder, was mit dem Oszilloskop verifiziert werden konnte.

[3] Gerätebäume (device trees) sind Baum Datenstrukturen mit Knoten, die Parameter für Hardware beinhalten, wie beispielsweise das Taktgeschwindigkeit im Falle von I²C

[4] Quelle: `https://elinux.org/BCM2835_datasheet_errata#p35_I2C_clock_stretching` (Stand: 14.01.2018)

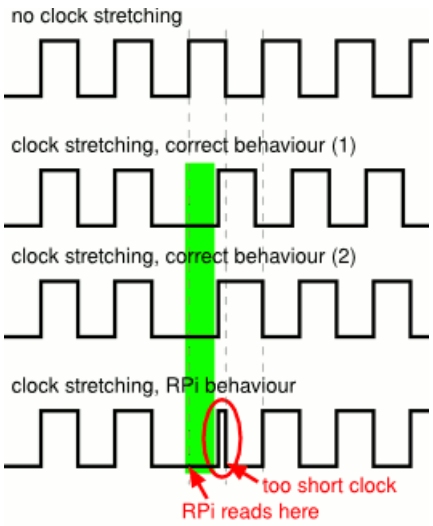


Abb. 3.16: Darstellung der fehlerhaften clock stretching Verhaltens des Broadcom BCM2835 chips: Das in der untersten Grafik dargestellte SCL Signal zeigt einen zu kurzen Taktzyklus, was zu einer fehlerhaften Datenübertragung führt (Quelle: (1)).

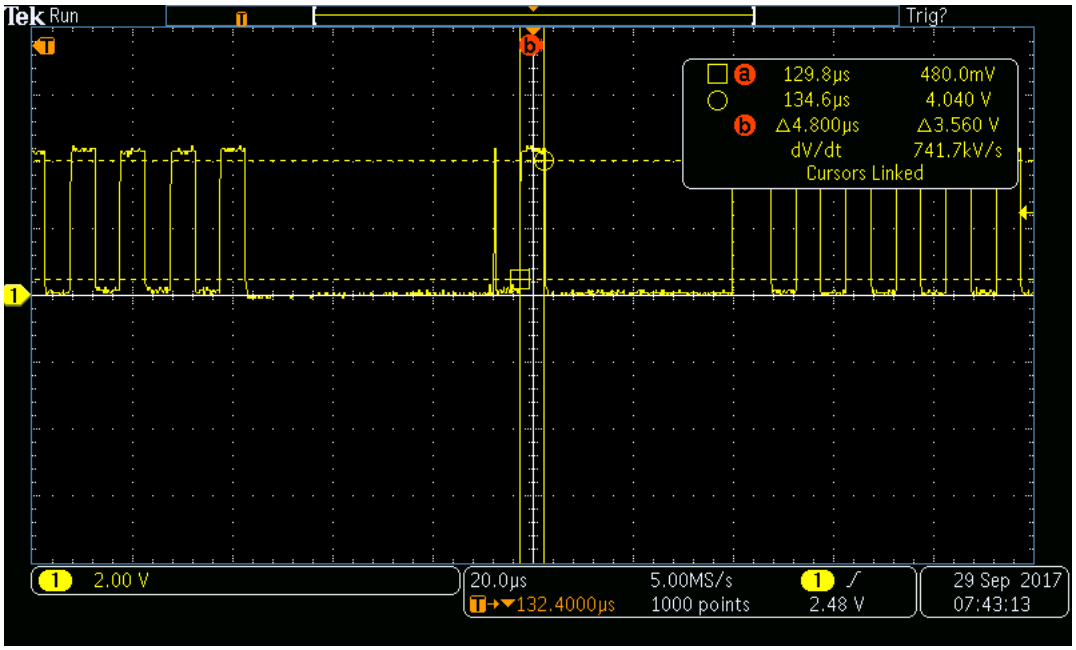


Abb. 3.17: Messung des I²C SCL Signals mit dem Oszilloskop: unmittelbar vor den vertikalen Cursorn zeigt sich ein zu kurzer Taktzyklus, der zu einer fehlerhaften Datenübertragung führt.

Um dieses Verhalten zu verhindern, wurde die I²C Kommunikation softwareseitig mit Hilfe der in Abschnitt 3.4 beschriebenen Bibiliothek *pigpio* implementiert. Hierbei wird

die I^2C Kommunikation über *bit banging* implementiert. Bit banging beschreibt eine Technik, bei der man Hardware Schnittstellen mit Hilfe von Software emuliert und somit deren Verhalten simuliert.

Weitere Probleme bei der Entwicklung

Beim Aufbau eines Prototypen der Schaltung auf einem Steckbrett kam es in Folge von zu langen Kabeln zu Interferenzen innerhalb des SCL Signals führenden Kabels. Wurde ein Signal mit dem AWG auf einen der Mikrocontroller gegeben, so zeigten sich deutliche Spannungsspitzen auf dem SCL Signal.

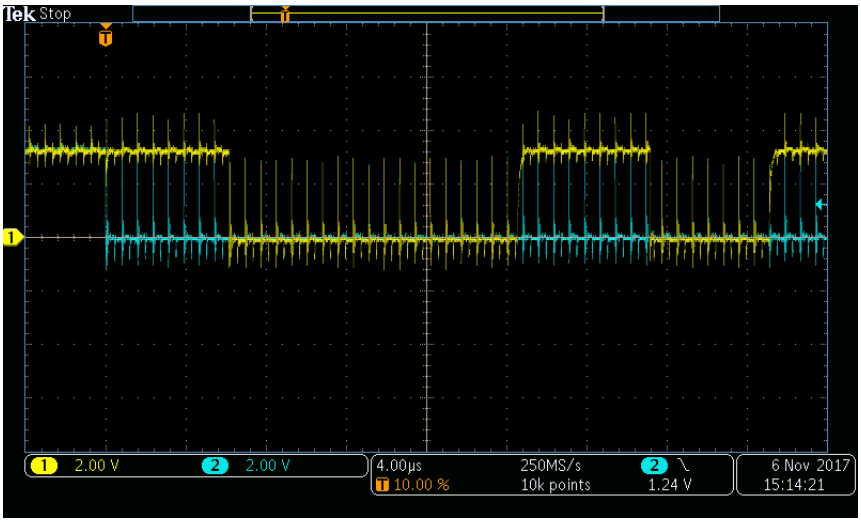


Abb. 3.18: Durch Verwendung zu langer Kabel wurde das SCL Signal verfälscht, sobald ein Signal auf einen der Mikrocontroller gegeben wurde.

Durch das Löten der Schaltung auf eine Streifenrasterplatine konnte ein solches Problem nicht erneut beobachtet werden. Jedoch kam es hier durch Verwendung eines 6-adrigen Kabels zu einem anderen Problem: Das durch die Schrittmotor Treiber ICs ausgegebene Steuersignal für die Motoren interferierte mit dem Kabel, das mit den Endschaltern verbunden war. Dieses Kabel ist auch über einen internen Pullup Widerstand des Raspberry Pis mit 3.3 V verbunden und sollte, im Fall eines geöffneten bzw. geschlossenen Schalters, einen Spannungspegel von 3.3 V bzw. 0 V aufweisen.

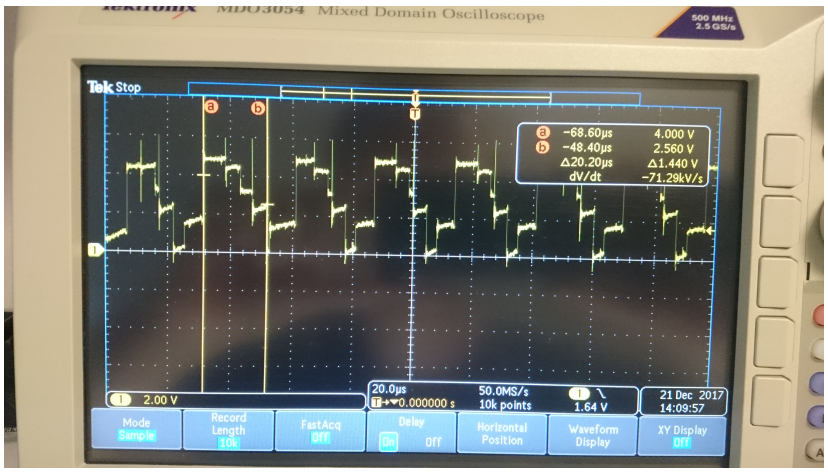
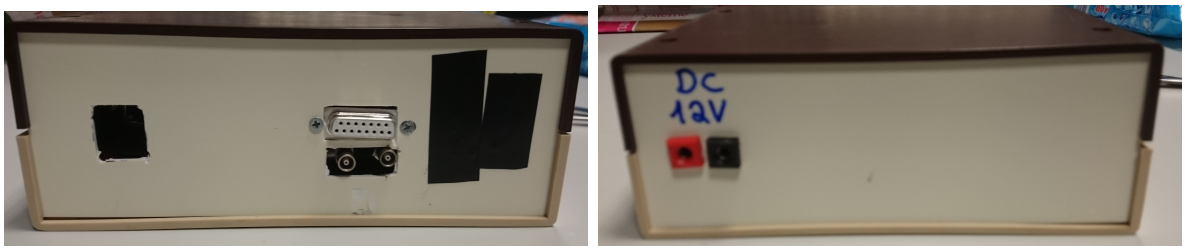


Abb. 3.19: Gemessenes Signal an der mit einem der Endschalter verbundenen Leitung: Diese sollte nur zwei mögliche Signalpegel von 0 V oder 3.3 V zeigen. Durch Reflexionen innerhalb des Kabels findet sich dort jedoch das von dem Schrittmotor Treiber IC ausgegebene Signal wieder.

Um die Signalintegrität wiederherzustellen wurden deshalb zwei 100 nF Bypass Kondensatoren von den Leitungen, die mit den Endschaltern verbunden sind, mit Ground verbunden, um die höher frequenten Anteile des Signals herauszufiltern.

3.3 Gehäuseanfertigung

Zusätzlich zur Schaltung, bestehend aus zwei Mikrocontrollern, zwei Schrittmotor Treiber ICs und einem *Abwärtsregler*, wurde auch der Raspberry Pi auf der Streifenrasterplatine befestigt. Zum Erfassen der Signale wurden weiter zwei 50 Ω terminierte LEMO Buchsen auf der Platine befestigt. Um ein falsches Anschließen des Scanners an das Gehäuse zu verhindern, wird die Verbindung über einen D-Sub Stecker bereitgestellt. Auf der Rückseite befinden sich zwei Bananenbuchsen für die 12 V Spannungsversorgung der Schrittmotoren und des Raspberry Pis. Letztere wird intern über den Abwärtsregler von 12 V auf 5 V heruntergeregelt.



(a) Frontansicht des Gehäuses: links Ethernetanschluss des Raspberry Pis. Mitte: LEMO Buchsen als Signaleingang

(b) Rückansicht des Gehäuses mit Buchsen zur Spannungsversorgung

Abb. 3.20: Gehäuseansichten

3.4 Software

In diesem Abschnitt werden die verwendeten Bibliotheken zur Erstellung der Kontrollprogramms vorgestellt.

3.4.1 pigpio

Die pigpio Bibliothek (`http://abyz.me.uk/rpi/pigpio/cif.html` Stand: 14.01.2018) ist eine in C geschrieben Bibliothek, die es erlaubt, die GPIOs des Raspberry Pis zu kontrollieren. Die wichtigsten verwendeten Funktionen der Bibliothek, die zur Erstellung des Programms verwendet wurden sind das Lesen und das Schreiben einzelner GPIOs über die Funktionen *gpioRead* und *gpioWrite*, sowie die über bit banging implementierte I^2C Kommunikation, zu der die Funktion *bbI2CZip* eine Schnittstelle darstellt.

3.4.2 I^2C Bibliothek ATMega328p

Zur grundlegenden Implementierung des I^2C Protokolls auf den Mikrocontrollern wurde die in `https://github.com/thegouger/avr-i2c-slave` (Stand: 14.01.2018) zur Verfügung gestellte Bibliothek genutzt. Ergänzt wurden die in **??** angegebenen Funktionen *I2C_requested()* und *I2C_received()*, sowie die Funktionen *setSeed()* und *sendFreq()*.

4 Steuerungsprogramm

Das Steuerungsprogramm des Scanners ist ein in C geschriebenes Konsolenprogramm. Im den folgenden Abschnitten wird auf die wesentlichen Bestandteile des Programms eingegangen. Der vollständige Quellcode befindet sich im Anhang.

4.1 Messparameter & Konfigurationsdatei Datei

Beim Start des Programms wird eine Konfigurationsdatei eingelesen und überprüft, ob alle relevanten Messparameter korrekt gesetzt wurden. Sollte dies nicht der Fall sein, beendet sich das Programm und zeigt an, wo sich der Fehler innerhalb der Konfigurationsdatei befindet (vgl. Abb. 4.1). Die Konfigurationsdatei muss sich dazu im aktuellen Verzeichnis des Programms befinden und mit *properties.conf* bezeichnet werden.

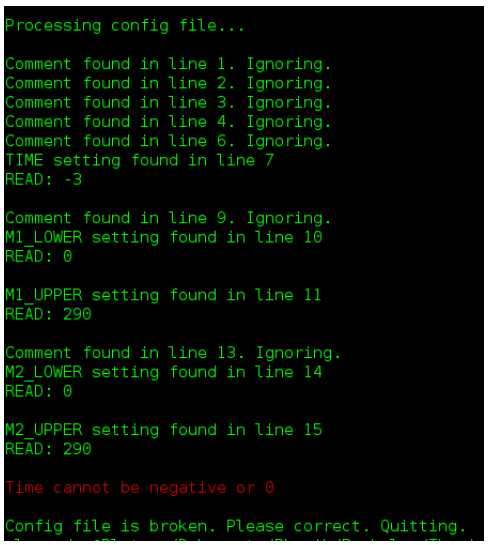


Abb. 4.1: Beispiel für falsch gesetzter Messparameter in der Konfigurationsdatei

Als Messparameter sind folgende Variablen gültig:

1. TIME: Definiert die Messzeit pro Position. Aufgrund der internen Programmierung der Mikrocontroller gibt TIME nicht die tatsächliche Zeit an, sondern ist ein Vielfaches von 1.28 ms und kann Werte von 1 bis 255 annehmen.

2. mN_lower: Für $N = 1, 2$ gibt dieser Wert die untere Position der Szintillatoren 1 bzw. 2 im Scanbereich an.

3. mN_upper: Für $N = 1, 2$ gibt dieser Wert die obere Position der Szintillatoren 1 bzw. 2 im Scanbereich an.

4.2 Hauptprogramm

Abb. 4.2 zeigt das Hauptmenü des Programms.

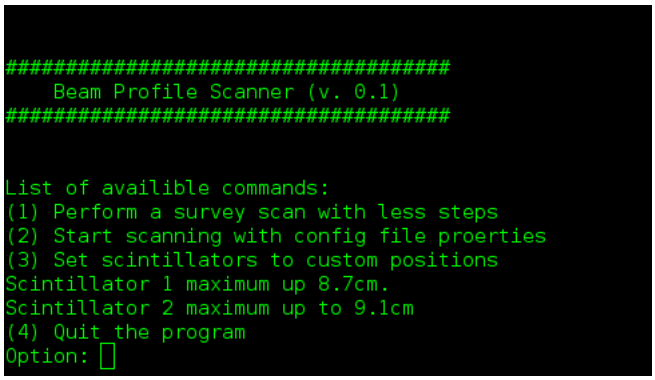


Abb. 4.2: Hauptmenü des Kontrollprogramms

- (1) Startet einen Übersichtsscan. Bei diesem werden die Szintillatoren mit einer größeren Schrittweite zum Ende des maximalen Scanbereiches bewegt.

- (2) Startet einen Scan, mit den in der Konfigurationsdatei definierten Messparametern. Anschließend werden die gemessenen Pulse in einer Datei mit Namen *data.dat* gespeichert. Die Datei speichert dabei in der ersten Spalte die Position (relativ zum Startpunkt mN_lower) des ersten Szintillators, in der zweiten Spalte die des zweiten. In der dritten Spalte befinden sich die durch den Mikrocontroller gezählten Pulse des ersten Szintillators, in der vierten Spalte die des zweiten Szintillators.

- (3) Bewegt beide Szintillatoren in die vom Benutzer angegebene Position.

- (4) Beendet das Programm.

Die aufgenommenen Daten lassen sich mit dem im Anhang zu findenen *root* Skript visualisieren.

5 Test

Um den Scanner zu testen wurde eine Bismuth Quelle in der Mitte des Scanbereiches hinter dem Scanner fixiert und der maximale Scanbereich, von etwa 9 cm × 9 cm, eingestellt.

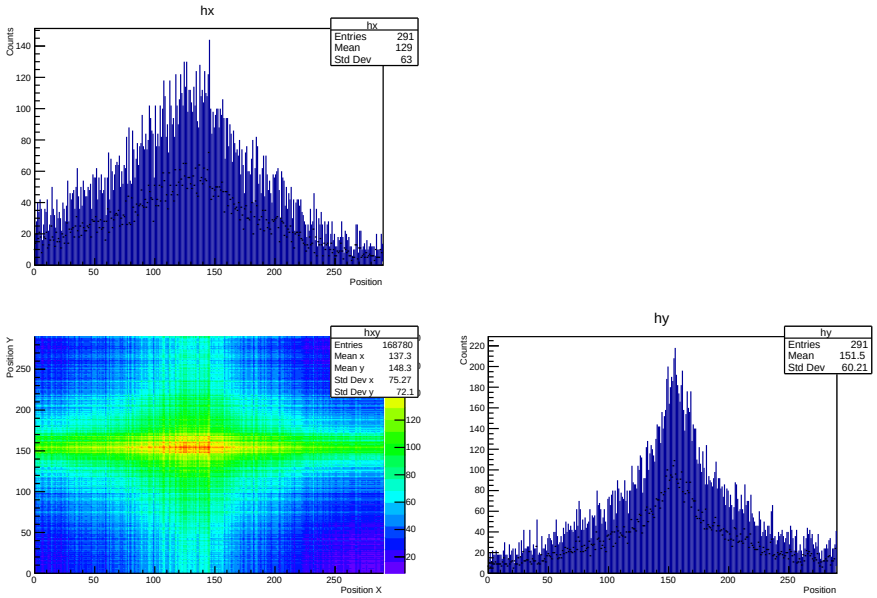


Abb. 5.1: Aufgenommenes Bild des Strahlprofilscanners mit einer im Mittelpunkt des Scanbereiches zentrierten Bismuth Quelle. Das Histogramm hx stellt die gemessenen Pulse des Szintillators dar, der sich in horizontaler Richtung bewegte, während hy die Pulse des Szintillators darstellt, der sich in vertikaler Richtung bewegte.

In den jeweiligen Histogrammen sind verschiedene maximale Zählraten erkennbar und das Maximum in Histogramm hx weist ein unschärferes Maximum in der Zählrate auf als das Histogramm hy, was darauf zurückzuführen ist, dass der Szintillator in y Richtung der Quelle am nächsten war.

Aufgrund eines mechanischen Defekts eines Mikroschalters, konnte der Scanner während der Strahlzeit im Dezember 2017 nicht getestet werden. Auf Basis des Tests mit der Bismuth Quelle sollte der Scanner jedoch auch vergleichbare Resultate liefern.

6 Zusammenfassung

Das Ziel der vorliegenden Arbeit war die Entwicklung eines Strahlprofilscanners, auf Basis zweier steuerbaren Szintillatoren sowie die Entwicklung des dazugehörigen Datenaufnahmesystems, welches aus einem Raspberry Pi sowie zwei Mikrocontrollern besteht.

Zu Beginn wurde ein Programm für die Mikrocontroller in C geschrieben, das es erlaubt, diese als Pulszähler bis zu einer maximalen Frequenz von 6 MHz zu nutzen. Mit Hilfe eines kalibrierten arbitrary wave generators (AWG) wurden im Anschluss für verschiedene Frequenzen die systematischen Offsets des Mikrocontrollers ermittelt, mit dem Ergebnis, dass die in einer Sekunde vom Mikrocontroller gemessenen Pulse um maximal 1% von den ausgegeben Pulsen des AWG abweichen.

Im zweiten Schritt mussten die so ermittelten Pulse an den Raspberry Pi übertragen werden. Dazu wurde die Kommunikation zwischen Raspberry Pi und den Mikrocontrollern mittels I^2C realisiert. Während des Tests auf eine fehlerfreie Datenübertragung, kam es in unregelmäßigen Abständen zu fehlerhaften Übertragungen, bei denen das least significant bit (LSB) des zu übertragenen Bytes verloren ging. Nach sehr intensiver Überprüfung des für die Kommunikation zuständigen Codes sowohl auf Seiten der Mikrocontroller, als auch auf Seite des Raspberry Pis, konnte ein Programmierfehler ausgeschlossen werden. Durch eine Internetrecherche konnte der Fehler schließlich auf einen Hardware Bug, des für die I^2C zuständigen Chips des Raspberry Pis, zurückgeführt werden. Aus diesem Grund musste auf die Nutzung des Chips verzichtet werde und die I^2C Schnittstelle softwareseitig über bit banging simuliert werden. Des Weiteren musste die Schaltung schon während des Testens auf eine Platine gelötet werden, da der Aufbau auf einem Steckbrett zu Spannungsspitzen in der I^2C Taktleitung führte.

Nachdem die Elektronik zur Datenaufnahme sowie die Datenübertragung erfolgreich getestet werden konnten, konnte der Strahprofilscanner aufgebaut werden. Dazu wurden zwei Aluminiumprofile, mit Hilfe eines im 3D Drucker hergestellten Verbindungsstückes, rechtwinklig zueinander verbunden und die Szintillatoren auf den Laufkatzen fixiert. Zur Steuerung des Strahlprofilscanners wurde ein Programm entwickelt, das eine Konfigu-

rationsdatei einliest und anschließend einen Scan, mit der in der Konfigurationsdatei definierten Messparametern, startet.

Bei einem ersten Test mit einer radioaktiven Quelle konnte der intendierte Verwendungszweck erfolgreich bestätigt werden: Hierbei konnte die Position der Quelle im Scanbereich aus dem Histogramm ermittelt werden.

7 Abkürzungsverzeichnis

APD Avalanche Photodiode

AWG Arbitrary Waveform Generator

EDM Elektrisches Dipolmoment

GPIO General Purpose Input Output

IC Integrated Circuit

I^{2}C Inter-Integrated Circuit

JEDI Jülich Electric Dipole moment Investigations

LSB Least significant bit

MSB Most significant bit

NIM Nuclear Instrumentation Standard

ppm parts per million

SCL Serial Clock

SDA Serial Data

SiPM Silicon photomultiplier

TTL Transistor-Transistor-Logik

8 Danksagung

Bedanken möchte ich mich bei Herrn Prof. Jörg Pretz, der meine Bachelorarbeit betreut und begutachtet hat und mir die Möglichkeit eröffnet hat, dieses Projekt im Forschungszentrum Jülich zu realisieren.

Außerdem möchte ich mich bei Dr. Irakli Keshelashvili und Fabian Müller für ihre Bereitschaft bedanken, mir die nötigen physikalischen Hintergründe erklärt zu haben und mir bei technischen Problemen immer hilfreich zur Seite gestanden zu haben.

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Bachelorarbeit mit dem Titel *Entwicklung eines Strahlprofilscanners für JEDI* selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen

Düren, den 31. Januar 2018

. . . . . .

(Alexander Krampe)

9 Anhang

Listing 9.1: Makefile zum flashen des Codes auf den ATMega328p

```
MCU   = attiny84
F_CPU = 20000000UL
BAUD  = 9600UL

## A directory for common include files and the simple USART library.
## If you move either the current folder or the Library folder, you'll
##  need to change this path to match.
LIBDIR = ~/Dokumente/AVRProgramming/AVR-Programming-master/AVR-Programming-Library

##########------------------------------------------------------##########
##########                 Programmer Defaults                  ##########
##########          Set up once, then forget about it           ##########
##########        (Can override.  See bottom of file.)          ##########
##########------------------------------------------------------##########

PROGRAMMER_TYPE = usbasp
# extra arguments to avrdude: baud rate, chip type, -F flag, etc.
PROGRAMMER_ARGS =

##########------------------------------------------------------##########
##########                  Program Locations                   ##########
##########     Won't need to change if they're in your PATH     ##########
##########------------------------------------------------------##########

CC = avr-gcc
OBJCOPY = avr-objcopy
OBJDUMP = avr-objdump
AVRSIZE = avr-size
AVRDUDE = avrdude

##########------------------------------------------------------##########
##########                   Makefile Magic!                    ##########
##########         Summary:                                     ##########
##########             We want a .hex file                      ##########
##########        Compile source files into .elf                ##########
##########        Convert .elf file into .hex                   ##########
##########        You shouldn't need to edit below.             ##########
##########------------------------------------------------------##########

## The name of your project (without the .c)
# TARGET = blinkLED
## Or name it automatically after the enclosing directory
TARGET = $(lastword $(subst /, ,$(CURDIR)))
```

```
45 # Object files: will find all .c/.h files in current directory
46 #  and in LIBDIR.  If you have any other (sub-)directories with code,
47 #  you can add them in to SOURCES below in the wildcard statement.
48 SOURCES=$(wildcard *.c $(LIBDIR)/*.c)
49 OBJECTS=$(SOURCES:.c=.o)
50 HEADERS=$(SOURCES:.c=.h)

52 ## Compilation options, type man avr-gcc if you're curious.
53 CPPFLAGS = -DF_CPU=$(F_CPU) -DBAUD=$(BAUD) -I. -I$(LIBDIR)
54 CFLAGS = -Os -g -std=gnu99 -Wall
55 ## Use short (8-bit) data types
56 CFLAGS += -funsigned-char -funsigned-bitfields -fpack-struct -fshort-enums
57 ## Splits up object files per function
58 CFLAGS += -ffunction-sections -fdata-sections
59 LDFLAGS = -Wl,-Map,$(TARGET).map
60 ## Optional, but often ends up with smaller code
61 LDFLAGS += -Wl,--gc-sections
62 ## Relax shrinks code even more, but makes disassembly messy
63 ## LDFLAGS += -Wl,--relax
64 ## LDFLAGS += -Wl,-u,vfprintf -lprintf_flt -lm  ## for floating-point printf
65 ## LDFLAGS += -Wl,-u,vfprintf -lprintf_min      ## for smaller printf
66 TARGET_ARCH = -mmcu=$(MCU)

68 ## Explicit pattern rules:
69 ##  To make .o files from .c files
70 %.o: %.c $(HEADERS) Makefile
71    $(CC) $(CFLAGS) $(CPPFLAGS) $(TARGET_ARCH) -c -o $@ $<;

73 $(TARGET).elf: $(OBJECTS)
74   $(CC) $(LDFLAGS) $(TARGET_ARCH) $^ $(LDLIBS) -o $@

76 %.hex: %.elf
77    $(OBJCOPY) -j .text -j .data -O ihex $< $@

79 %.eeprom: %.elf
80   $(OBJCOPY) -j .eeprom --change-section-lma .eeprom=0 -O ihex $< $@

82 %.lst: %.elf
83   $(OBJDUMP) -S $< > $@

85 ## These targets don't have files named after them
86 .PHONY: all disassemble disasm eeprom size clean squeaky_clean flash fuses

88 all: $(TARGET).hex

90 debug:
91   @echo
92   @echo "Source files:"   $(SOURCES)
93   @echo "MCU, F_CPU, BAUD:"  $(MCU), $(F_CPU), $(BAUD)
94   @echo

96 # Optionally create listing file from .elf
```

```
97  # This creates approximate assembly-language equivalent of your code.
98  # Useful for debugging time-sensitive bits,
99  # or making sure the compiler does what you want.
100 disassemble: $(TARGET).lst

102 disasm: disassemble

104 # Optionally show how big the resulting program is
105 size:  $(TARGET).elf
106   $(AVRSIZE) -C --mcu=$(MCU) $(TARGET).elf

108 clean:
109   rm -f $(TARGET).elf $(TARGET).hex $(TARGET).obj \
110   $(TARGET).o $(TARGET).d $(TARGET).eep $(TARGET).lst \
111   $(TARGET).lss $(TARGET).sym $(TARGET).map $(TARGET)~ \
112   $(TARGET).eeprom

114 squeaky_clean:
115   rm -f *.elf *.hex *.obj *.o *.d *.eep *.lst *.lss *.sym *.map *~ *.eeprom

117 ##########------------------------------------------------------##########
118 ##########              Programmer-specific details             ##########
119 ##########           Flashing code to AVR using avrdude         ##########
120 ##########------------------------------------------------------##########

122 flash: $(TARGET).hex
123   $(AVRDUDE) -c $(PROGRAMMER_TYPE) -p $(MCU) $(PROGRAMMER_ARGS) -U flash:w:$<

125 ## An alias
126 program: flash

128 flash_eeprom: $(TARGET).eeprom
129   $(AVRDUDE) -c $(PROGRAMMER_TYPE) -p $(MCU) $(PROGRAMMER_ARGS) -U eeprom:w:$<

131 avrdude_terminal:
132   $(AVRDUDE) -c $(PROGRAMMER_TYPE) -p $(MCU) $(PROGRAMMER_ARGS) -nt

134 ## If you've got multiple programmers that you use,
135 ## you can define them here so that it's easy to switch.
136 ## To invoke, use something like `make flash_arduinoISP`
137 flash_usbtiny: PROGRAMMER_TYPE = usbtiny
138 flash_usbtiny: PROGRAMMER_ARGS =  # USBTiny works with no further arguments
139 flash_usbtiny: flash

141 flash_usbasp: PROGRAMMER_TYPE = usbasp
142 flash_usbasp: PROGRAMMER_ARGS =  # USBasp works with no further arguments
143 flash_usbasp: flash

145 flash_arduinoISP: PROGRAMMER_TYPE = avrisp
146 flash_arduinoISP: PROGRAMMER_ARGS = -b 19200 -P /dev/ttyACM0
147 ## (for windows) flash_arduinoISP: PROGRAMMER_ARGS = -b 19200 -P com5
148 flash_arduinoISP: flash
```

```
150 flash_109: PROGRAMMER_TYPE = avr109
151 flash_109: PROGRAMMER_ARGS = -b 9600 -P /dev/ttyUSB0
152 flash_109: flash

154 ##########------------------------------------------------------##########
155 ##########      Fuse settings and suitable defaults             ##########
156 ##########------------------------------------------------------##########

158 ## Default for ATtiny84
159 LFUSE = 0x62
160 HFUSE = 0xdf
161 EFUSE = 0xff

163 ## Generic
164 FUSE_STRING = -U lfuse:w:$(LFUSE):m -U hfuse:w:$(HFUSE):m -U efuse:w:$(EFUSE):m

166 fuses:
167   $(AVRDUDE) -c $(PROGRAMMER_TYPE) -p $(MCU) \
168               $(PROGRAMMER_ARGS) $(FUSE_STRING)
169 show_fuses:
170   $(AVRDUDE) -c $(PROGRAMMER_TYPE) -p $(MCU) $(PROGRAMMER_ARGS) -nv

172 ## Called with no extra definitions, sets to defaults
173 set_default_fuses:  FUSE_STRING = -U lfuse:w:$(LFUSE):m -U hfuse:w:$(HFUSE):m -U efuse:w:$(EFUSE):
174 set_default_fuses:  fuses

176 ## Set the fuse byte for full-speed mode
177 ## Note: can also be set in firmware for modern chips
178 set_fast_fuse: LFUSE = 0xE2
179 set_fast_fuse: FUSE_STRING = -U lfuse:w:$(LFUSE):m
180 set_fast_fuse: fuses

182 ## Set the EESAVE fuse byte to preserve EEPROM across flashes
183 set_eeprom_save_fuse: HFUSE = 0xD7
184 set_eeprom_save_fuse: FUSE_STRING = -U hfuse:w:$(HFUSE):m
185 set_eeprom_save_fuse: fuses

187 ## Clear the EESAVE fuse byte
188 clear_eeprom_save_fuse: FUSE_STRING = -U hfuse:w:$(HFUSE):m
189 clear_eeprom_save_fuse: fuses

191 ## Set fuses for suitable external crystal
192 set_crystal_fuses: LFUSE = 0xe2
193 set_crystal_fuses: HFUSE = 0xdf
194 set_crystal_fuses: EFUSE = 0xff
195 set_crystal_fuses: fuses
```

Listing 9.2: Programm zum Zählen der Pulse

```
1 #include<avr/io.h>
2 #include<avr/interrupt.h>
3 #include<util/delay.h>
4 #include "USART.h"
```

```
5  #include <string.h>
6  #include "I2CSlave.h"
7  #define SLAVE_ADDRESS 0x04



11 //Global variables
12 uint32_t f_freq;
13 volatile unsigned char f_ready;
14 volatile unsigned char f_mlt;
15 volatile uint16_t f_tics;
16 volatile uint8_t begin = 1;
17 volatile uint16_t f_period;
18 volatile uint8_t seed = 0;
19 char seedFlag = 0;
20 uint32_t _f;
21 uint32_t test;
22 char buffer[32];
23 char numBuf[20];
24 //I2C variable to hold data
25 volatile uint8_t data;
26 int i = 0;

28 void start(uint16_t ms) {
29     TIMSK0 &=~(1<<TOIE0);       // disable Timer0  //disable  millis and delay
30     f_period=ms;

32     TCCR1A=0;                  // reset timer/counter1 control register A
33     TCCR1B=0;                  // reset timer/counter1 control register A
34     TCNT1=0;                 // counter value = 0

36     TCCR1B |=  (1<<CS10) ;// External clock source on T1 pin. Clock on rising edge.
37     TCCR1B |=  (1<<CS11) ;
38     TCCR1B |=  (1<<CS12) ;

40     // timer2 setup / is used for frequency measurement gatetime generation
41     TCCR2A=0;
42     TCCR2B=0;


45     //timer 2 prescaler to 256
46     TCCR2B |= (1 << CS21);
47     TCCR2B |= (1 << CS22);
48     //set timer2 to CTC Mode with OCR2A is top counter value
49     TCCR2A &= ~(1<<WGM20) ;
50     TCCR2A |=  (1<<WGM21) ;
51     TCCR2A &= ~(1<<WGM22) ;
52     OCR2A = 99;              // 99 Ticks = 1.28ms

54     f_ready=0;                  // reset period measure flag
55     f_tics=0;                   // reset interrupt counter
56     GTCCR = (1<<PSRASY);        // reset presacler counting
57     TCNT2=0;                    // timer2=0
```

```
58      TCNT1=0;                        // Counter1 = 0

60      TIMSK2 |=(1<<OCIE2A);           // enable Timer2 Interrupt

62                                      // External clock source on T1 pin. Clock on rising edge.
63      TCCR1B |= (1<<CS12) | (1<<CS11) | (1<<CS10);        //   start counting now
64  }


67  ISR(TIMER2_COMPA_vect) {
68     if (f_tics >= f_period) {
69          TCCR1B = TCCR1B & ~7;           // Gate Off  / Counter T1 stopped
70          TIMSK2 &= ~(1<<OCIE2A);          // disable Timer2 Interrupt
71          TIMSK0 |=(1<<TOIE0);            // enable Timer0 again // millis and delay
72          f_ready=1;                 // set global flag for end count period

74                                           // calculate now frequeny value
75          //f_freq=0x10000 * f_mlt;  // mult #overflows by 65636
76          f_freq=f_mlt * 0x10000;
77          f_freq += TCNT1;           // add counter1 value
78          f_mlt=0;

80      }
81      f_tics++;                  // count number of interrupt events
82      if (TIFR1 & 1) {                   // if Timer/Counter 1 overflow flag
83          f_mlt++;                   // count number of Counter1 overflows
84          TIFR1 =(1<<TOV1);              // clear Timer/Counter 1 overflow flag
85      }
86  }

88  void setSeed(){
89      seed = (uint8_t)data;
90      seedFlag = 0;
91      I2C_transmitByte('A'); //successfully set seed for gate time
92  }

94  void sendFreq(){
95      char byte = 0;
96      byte |= (_f >>  8*i++); //magic do not touch
97      I2C_transmitByte(byte);
98      if(i ==4){
99          i = 0;
100     }
101 }

103 void I2C_received(uint8_t received){
104     data = received;
105 }

107 void I2C_requested(){
108     if(!seedFlag){
109         switch(data){
110       //print statements are for Debug purpose
```

```
111             case 'F': printString("Switch case F"); sendFreq(); break;
112             case 'S': printString("Switch case S");seedFlag = 1; break;
113             case 'B': printString("In switch B"); I2C_transmitByte('S');
114                 begin = 0;  break;
115             default: break;
116         }
117     }
118     else
119         setSeed();

121 }

123 int main(void)
124 {
125     initUSART();
126     I2C_init(SLAVE_ADDRESS);
127     I2C_setCallbacks(I2C_received, I2C_requested);
128     while(seed == 0);
129     while(1){
130         while(begin); //this is needed fpor synchro stepper - counter
131         sei();
132         start(seed);
133         while(f_ready == 0);
134         _f = f_freq ;
135         begin = 1; //stop. No new counts until stepper is in new position.
136     }

138     return 0;
139 }
```

Listing 9.3: I2C.h

```
1  #ifndef I2C_SLAVE_H
2  #define I2C_SLAVE_H

4  #include <avr/interrupt.h>
5  #include <stdint.h>

7  void I2C_init(uint8_t address);
8  void I2C_stop(void);
9  void I2C_setCallbacks(void (*recv)(uint8_t), void (*req)());

11 inline void __attribute__((always_inline)) I2C_transmitByte(uint8_t data)
12 {
13   TWDR = data;
14 }

16 ISR(TWI_vect);

18 #endif
```

Listing 9.4: I2C.c

```
1  #include <util/twi.h>
2  #include <avr/interrupt.h>

4  #include "I2CSlave.h"

6  static void (*I2C_recv)(uint8_t);
7  static void (*I2C_req)();

9  void I2C_setCallbacks(void (*recv)(uint8_t), void (*req)())
10 {
11   I2C_recv = recv;
12   I2C_req = req;
13 }

15 void I2C_init(uint8_t address)
16 {
17   cli();
18   // load address into TWI address register
19   TWAR = address << 1;
20   // set the TWCR to enable address matching and enable TWI, clear TWINT, enable TWI interrupt
21   TWCR = (1<<TWIE) | (1<<TWEA) | (1<<TWINT) | (1<<TWEN);
22   sei();
23 }

25 void I2C_stop(void)
26 {
27   // clear acknowledge and enable bits
28   cli();
29   TWCR = 0;
30   TWAR = 0;
31   sei();
32 }

34 ISR(TWI_vect)
35 {
36   switch(TW_STATUS)
37   {
38     case TW_SR_DATA_ACK:
39       // received data from master, call the receive callback
40       I2C_recv(TWDR);
41       TWCR = (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
42       break;
43     case TW_ST_SLA_ACK:
44       // master is requesting data, call the request callback
45       I2C_req();
46       TWCR = (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
47       break;
48     case TW_ST_DATA_ACK:
49       // master is requesting data, call the request callback
50       I2C_req();
51       TWCR = (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
52       break;
53     case TW_BUS_ERROR:
```

```
54        // some sort of erroneous state, prepare TWI to be readdressed
55        TWCR = 0;
56        TWCR = (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
57        break;
58      default:
59        TWCR = (1<<TWIE) | (1<<TWINT) | (1<<TWEA) | (1<<TWEN);
60        break;
61    }
62  }
```

Listing 9.5: scanner.c

```
1  #include "measurement.h"
2  #include "calibration.h"
3  #include <stdlib.h>
4  #include <string.h>
5  #include <ctype.h>
6  #define KRED  "\x1B[31m"
7  #define KNRM  "\x1B[0m"
8  int isBlank (char const * line)
9  {
10   char * ch;
11   int is_blank = -1;

13   // Iterate through each character.
14   for (ch = line; *ch != '\0'; ++ch)
15   {
16     if (!isspace(*ch))
17     {
18       // Found a non-whitespace character.
19       is_blank = 0;
20       break;
21     }
22    }

24   return is_blank;
25 }


28 int getParameter(const char* line)
29 {
30   /*
31   look for '=' in line  after that look for the first non blank.
32   save all of this non blank chars in input_buffer up to the next occuring blank
33   cast this buffer to int if possible.
34   */
35   unsigned k = 0;
36   while(line[k] != '=')
37     ++k;

39   while(isspace(line[++k]));
40   char input_buffer[8];
41   unsigned l = 0;
```

```
43    while(!isspace(line[k]))
44      input_buffer[l++] = line[k++];

46    input_buffer[l] = '\0';
47    printf("READ: %s\r\n\r\n", input_buffer);
48    //Now cast to int and check if this is even possible
49    return atoi(input_buffer);
50  }

52  bool processConfigFile(FILE* f, Measurement* m)
53  {
54    printf("\r\n\r\nProcessing config file...\r\n\r\n");
55    char line_buffer[512];
56    bool time_set = 0, m1_lower_set = 0, m1_upper_set = 0, m2_lower_set = 0, m2_upper_set = 0, bad_l
57    int time, m1_lower, m1_upper, m2_lower, m2_upper;
58    unsigned line_number = 1;
59    while(fgets(line_buffer, 512, f) != NULL)
60    {
61      if(strstr(line_buffer, "#"))
62          printf("Comment found in line %i. Ignoring.\r\n", line_number);

64      else if(strstr(line_buffer, "TIME") != NULL)
65      {
66        printf("TIME setting found in line %i\r\n", line_number);
67        time = getParameter(line_buffer);
68        time_set = 1;
69      }
70      else if(strstr(line_buffer, "M1_LOWER") != NULL)
71      {
72        printf("M1_LOWER setting found in line %i\r\n", line_number);
73        m1_lower = getParameter(line_buffer);
74        m1_lower_set = 1;
75      }
76      else if(strstr(line_buffer, "M1_UPPER") != NULL)
77      {
78        printf("M1_UPPER setting found in line %i\r\n", line_number);
79        m1_upper = getParameter(line_buffer);
80        m1_upper_set = 1;
81      }
82      else if(strstr(line_buffer, "M2_LOWER") != NULL)
83      {
84        printf("M2_LOWER setting found in line %i\r\n", line_number);
85        m2_lower = getParameter(line_buffer);
86        m2_lower_set = 1;
87      }
88      else if(strstr(line_buffer, "M2_UPPER") != NULL)
89      {
90        printf("M2_UPPER setting found in line %i\r\n", line_number);
91        m2_upper = getParameter(line_buffer);
92        m2_upper_set = 1;
93      }
94      else if(!isBlank(line_buffer))
```

```
95      {
96        printf("\r\nCannot understand line %i. Whether comment nor measurement parameter\r\n\r\n", l
97        bad_line = 1;
98      }
99      ++line_number;
100   }

102   if(!time_set)
103   {
104     printf("\r\nNo TIME setting found. Quitting.\r\n");
105     return false;
106   }
107   if(!m1_lower_set)
108   {
109     printf("\r\nNo M1_LOWER setting found. Quitting.\r\n");
110     return false;
111   }
112   if(!m2_lower_set)
113   {
114     printf("\r\nNo M2_LOWER setting found. Quitting.\r\n");
115     return false;
116   }
117   if(!m1_upper_set)
118   {
119     printf("\r\nNo M1_UPPER setting found. Quitting.\r\n");
120     return false;
121   }
122   if(!m2_upper_set)
123   {
124     printf("\r\nNo M2_UPPER setting found. Quitting.\r\n");
125     return false;
126   }
127   if(bad_line)
128     return false;

130   //check if values make sense
131   if(time <= 0)
132   {
133     printf("%sTime cannot be negative or 0\r\n", KRED);
134     printf("%s", KNRM);
135     return false;
136   }
137   if(m1_lower >= m1_upper || m1_lower < 0 || m1_upper < 0)
138   {
139     printf("%sCheck M1 values!\r\n", KRED);
140     printf("%s", KNRM);
141     return false;
142   }
143   if(m2_lower >= m2_upper || m2_lower < 0 || m2_upper < 0)
144   {
145     printf("%sCheck M2 values!\r\n", KRED);
146     printf("%s", KNRM);
147     return false;
```

```
148     }

150     m -> gatetime = time;
151     m -> gatetime_set = 1;
152     m -> m1_lower = m1_lower;
153     m -> m1_upper = m1_upper;
154     m -> m2_lower = m2_lower;
155     m -> m2_upper = m2_upper;
156     m -> step_set = 1;
157     printf("\r\n\r\n##################################\r\n\r\n");
158     printf("The following parameters will be used\r\n");
159     printf("Time: %i\r\n\r\n", time);
160     printf("M1 start position: %i\r\n\r\n", m1_lower);
161     printf("M1 end position: %i\r\n\r\n", m1_upper);
162     printf("M2 start position: %i \r\n\r\n", m2_lower);
163     printf("M2 end position: %i \r\n\r\n", m2_upper);
164     printf("################################\r\n\r\n");

166     return true;
167   }



171   int initI2C()
172   {
173     if(gpioInitialise() < 0)
174       printf("Error in initializing!\n");


177     //Use standard GPIO pins 2 and 3 for I2C but in bit banged mode. 100000Hz speed.
178     if(bbI2COpen(2,3,100000))
179       printf("Error establishing bit banging\n");

181     //This return value 'pi' is needed to perform "PWM" on the pins which drives the stepper
182           int pi = pigpio_start(NULL, NULL);
183     return pi;
184   }

186   void show_commands()
187   {
188     printf("\r\n\r\n\r\n\r\n");
189     printf("#####################################\r\n");
190     printf("    Beam Profile Scanner (v. 0.1)    \r\n");
191     printf("#####################################\r\n");
192     printf("\r\n\r\n");
193     printf("List of availible commands:\r\n");
194     printf("(1) Perform a survey scan with less steps\r\n");
195     printf("(2) Start scanning with conig file proerties\r\n");
196     printf("(3) Set scintillators to custom positions\r\nScintillator 1 maximum up 8.7cm.\r\nScintil
197     printf("(4) Quit the program\r\n");
198   }
```

```
201 int main(int argc, char** argv)
202 {
203   int option;
204   char buf[10];
205   char quit = 0;
206   Measurement m;

208   FILE* cfg = fopen("properties.conf", "r");
209   bool good_file;

211   if(cfg == NULL)
212   {
213     printf("\r\nCannot open config file. Check if there is a file named 'properties.conf' in progr
214     return EXIT_FAILURE;

216   }
217   else
218   {
219     good_file = processConfigFile(cfg, &m);
220     if(!good_file)
221     {
222       printf("\r\nConfig file is broken. Please correct. Quitting.\r\n");
223       return EXIT_FAILURE;
224     }
225   }

227   int pi = initI2C();
228   gpioSetPullUpDown(5, PI_PUD_UP);
229   gpioSetPullUpDown(6, PI_PUD_UP);
230   show_commands();
231   while(1)
232   {
233     printf("Option: ");
234     fgets(buf ,sizeof(buf), stdin);
235     int check = sscanf(buf, "%d", &option);
236     char input= (check != 1 || ((option != 1) && (option != 2) && (option != 3) && (option != 4)))

238     if(input)
239     {
240       printf("Incorrect input\r\n");
241       continue;
242     }

244     switch(option)
245     {
246       case 1: step_motor2(pi); break;
247       case 2: start_scan(&m, pi); break;
248       case 3: set_to_custom_position(&m, pi); break;
249       case 4: quit = 1; break;
250       default: break;
251     }
252     if(quit)
253       break;
```

```
255     }

257     fclose(cfg);
258     bbI2CClose(2);
259     pigpio_stop(pi);
260     gpioTerminate();
261     return 0;
262   }
```

Listing 9.6: measurement.h

```
1   #ifndef MEASUREMENT_H
2   #define MEASUREMENT_H
3   #define _BSD_SOURCE
4   #include <unistd.h>
5   #include <stdbool.h>
6   #include <pigpio.h>
7   #include <pigpiod_if2.h>
8   #include <stdio.h>

10  typedef struct
11  {
12    bool gatetime_set;
13    bool step_set;
14    int m1_lower, m1_upper, m2_lower, m2_upper;
15    int gatetime;
16  } Measurement;

18  void set_position(Measurement*);
19  void set_gatetime(Measurement*);
20  void start_scan(Measurement*, int);
21  void set_step_number(Measurement*);
22  void raster_m1(int);
23  void raster_m2(int);
24  void get_counts(Measurement*);
25  void transfer_gatetime(Measurement*);
26  void start_counter1(Measurement*);
27  void start_counter2(Measurement*);
28  void set_to_initial_position(Measurement*, int);
29  void set_to_custom_position(Measurement*, int);
30  #endif
```

Listing 9.7: measurement.c

```
1   #include "measurement.h"
2   #include <stdlib.h>
3   #include <errno.h>
4   #include <stdio.h>
5   #define sDEBUG
6   #define max(a,b)  ({ __typeof__ (a) _a = (a);  __typeof__ (b) _b = (b); _a >= _b ? _a : _b; })
```

```
8  void writeData(unsigned* data_x, unsigned* data_y, unsigned len_x, unsigned len_y)
9  {
10   FILE* data = fopen("data.dat", "w");
11   if(data == NULL)
12     printf("\r\nError in writing data to file\r\n");


15   //only valid if you have a square that you are scanning
16   for(int i = 0; i < len_y; ++i)
17   {
18     fprintf(data, "%i %i %i %i", i, i, data_x[i], data_y[i]);
19     fprintf(data, "%s", "\r\n");
20   }

22   fclose(data);


25 }

27 void raster_m1(int pi)
28 {
29   /**
30     * generates a pulse with 400 mu s high
31     * and 400 mu s logic low to drive motor1.
32     * Note: Values < 400 do not seem to work properly
33   */
34   gpio_write(pi, 4, 1);
35   usleep(400);
36   gpio_write(pi, 4, 0);
37   usleep(400);
38 }


41 void raster_m2(int pi)
42 {
43   //see raster_m1
44   gpio_write(pi, 10, 1);
45   usleep(400);
46   gpio_write(pi, 10, 0);
47   usleep(400);
48 }

50 void set_to_custom_position(Measurement* m, int pi)
51 {
52   //enable the ICs
53   gpio_write(pi, 17, 1);
54   gpio_write(pi, 22, 1);
55   char buf[10];
56   double x, y;
57   printf("Enter coordinate for scintillator 1: ");
58   fgets(buf ,sizeof(buf), stdin);
59   int check = sscanf(buf, "%lf", &x);
60   if(check != 1 || x < 0 || x > 8.7)
```

```
61    {
62      printf("Wrong input\r\n");
63      gpio_write(pi, 17, 0);
64      gpio_write(pi, 22, 0);
65      return;
66    }
67    printf("\r\nEnter coordinate for scinitillator 2: ");
68    fgets(buf, sizeof(buf), stdin);
69    check = sscanf(buf, "%lf", &y);
70    if(check !=1 || y < 0 || y > 9.1)
71    {
72      printf("Wrong input\r\n");
73      gpio_write(pi, 17, 0);
74      gpio_write(pi, 22, 0);
75      return;
76    }

78    //now home the scintillators
79    set_to_initial_position(m, pi);
80    //move scintillator 1
81    //set direction first
82    gpio_write(pi, 27, 0);

84    /**
85      * 290 is the total number of pulses for scintillator 1
86      * 8.7 is its maximum travel distance. So 290/8.7*x is
87      * a linear map that maps pulses to cm.
88    */
89    for(int k = 0; k < (int)290.0/8.7*x; ++k)
90    {
91      gpio_write(pi, 4, 1);
92      usleep(400);
93      gpio_write(pi, 4, 0);
94      usleep(400);
95      usleep(1000);
96    }

98    //move scintillator 2
99    //set direction first then move

101   gpio_write(pi, 9, 1);
102   for(int k = 0; k < (int)305.0/9.1*y; ++k)
103   {
104     gpio_write(pi, 10, 1);
105     usleep(400);
106     gpio_write(pi, 10, 0);
107     usleep(400);
108     usleep(1000);
109   }

111   //turn off the ICs
112   gpio_write(pi, 17, 0);
113   gpio_write(pi, 22, 0);
```

```
114 }

116 void set_to_initial_position(Measurement *m, int pi)
117 {
118   gpio_write(pi, 17, 1); //wake up the IC from sleep
119   gpio_write(pi, 22, 1); //wake up IC from sleep

121   //according to the datasheet 1ms is needed for the charge pump
122   usleep(1000);
123   gpio_write(pi, 27, 1); //sets the direction correct to home fpr motor 1
124   while(gpioRead(5) != 0)
125   {
126     #ifdef DEBUG
127     printf("\r\nPin 5 reads %i \r\n", gpioRead(5));
128     #endif
129     gpio_write(pi, 4, 1);
130     usleep(400);
131     gpio_write(pi, 4, 0);
132     usleep(400);
133     usleep(1000);
134   }

136   /*This describes the case if the user does not want to start the measurement from the lowest pos
137   if(m -> m1_lower != 0)
138   {
139     gpio_write(pi, 27, 0);
140     //make m1_lower steps
141     for(int i = 0; i < m -> m1_lower; ++i)
142     {
143       gpio_write(pi, 4, 1);
144       usleep(400);
145       gpio_write(pi, 4, 0);
146       usleep(400);
147       usleep(1000);
148     }
149   }


152   gpio_write(pi, 9, 0); //sets direction to home for motor 2
153   while(gpioRead(6) != 0)
154   {
155     #ifdef DEBUG
156     printf("Pin 6 reads %i\r\n", gpioRead(6));
157     #endif
158     gpio_write(pi, 10, 1);
159     usleep(400);
160     gpio_write(pi, 10, 0);
161     usleep(400);
162     usleep(1000);
163   }

165   if(m -> m2_lower != 0)
166   {
```

```
167       gpio_write(pi, 9 ,1);
168       //make m2_lower steps
169       for(int i = 0; i < m -> m2_lower; ++i)
170       {
171         gpio_write(pi, 10, 1);
172         usleep(400);
173         gpio_write(pi, 10, 0);
174         usleep(400);
175         usleep(1000);
176       }
177     }
178     gpio_write(pi, 17, 0); //set to sleep again
179     gpio_write(pi, 22, 0); //set to sleep again
180   }

182   void transfer_gatetime(Measurement* m)
183   {
184     //transfer time to the first counter IC (0x04)
185     char receive[4];
186     char cmd_buf_seed[] = {4, 0x04, 2, 7, 1, 'S', 3, 2,6, 1,3,0};
187     bbI2CZip(2, cmd_buf_seed, sizeof(cmd_buf_seed), receive, sizeof(receive));
188     //set new Gate Time now
189     char cmd_buf_gate_time [] = {4, 0x04,2,7, 1, (char)m -> gatetime, 3, 2, 6, 1, 3, 0};
190     bbI2CZip(2, cmd_buf_gate_time, sizeof(cmd_buf_gate_time), receive, sizeof(receive));

192     //transfer time to the second counter IC (0x05)
193     char cmd_buf_seed2[] = {4, 0x05, 2, 7, 1, 'S', 3, 2,6, 1,3,0};
194     bbI2CZip(2, cmd_buf_seed2, sizeof(cmd_buf_seed2), receive, sizeof(receive));
195     //set new Gate Time now
196     char cmd_buf_gate_time2 [] = {4, 0x05, 2, 7, 1, (char)m -> gatetime, 3, 2, 6, 1, 3, 0};
197     bbI2CZip(2, cmd_buf_gate_time2, sizeof(cmd_buf_gate_time2), receive, sizeof(receive));

199   }

201   void start_count(Measurement* m)
202   {
203     /*This function starts both counters*/
204     char receive[4];
205     char cmd_buf_count[]={4,0x04, 2, 7, 1, 'B', 3, 2, 6, 1, 3,0};
206     bbI2CZip(2, cmd_buf_count, sizeof(cmd_buf_count), receive, sizeof(receive));
207     char cmd_buf_count2[]={4,0x05, 2, 7, 1, 'B', 3, 2, 6, 1, 3,0};
208     bbI2CZip(2, cmd_buf_count2, sizeof(cmd_buf_count2), receive, sizeof(receive));

210     //sleep at least the gatetime because the measuremnt is not ready yet before
211     usleep(1.28*1000*m->gatetime);
212   }

214   void start_counter1(Measurement* m)
215   {
216     char receive[4];
217     char cmd_buf_count[]={4,0x04, 2, 7, 1, 'B', 3, 2, 6, 1, 3,0};
218     bbI2CZip(2, cmd_buf_count, sizeof(cmd_buf_count), receive, sizeof(receive));
219     //sleep at least the gatetime because the measuremnt is not ready yet before
```

```
220   usleep(1.28*1000*m->gatetime);
221 }

223 void start_counter2(Measurement* m)
224 {
225   char receive[4];
226   char cmd_buf_count2[]={4,0x05, 2, 7, 1, 'B', 3, 2, 6, 1, 3,0};
227   bbI2CZip(2, cmd_buf_count2, sizeof(cmd_buf_count2), receive, sizeof(receive));
228   //sleep at least the gatetime because the measuremnt is not ready yet before
229   usleep(1.28*1000*m->gatetime);
230 }

232 unsigned int read_counts_x()
233 {
234   /*
235     cryptic command to read frequency
236     4: after 4 it expects the I2C adress
237     2: start command
238     7: command that indicate you will send smething followd by the number of bytes (here 1)
239     'F' the byte sent.
240     3: means stop. So here it means stop sending
241     2: start command
242     6: receive data followd by the number of received bytes (here 4 a uint_32)
243     3: stop receiving
244     0: end of command
245   */
246   char cmd_buf[] = {4, 0x04, 2, 7 ,1, 'F', 3, 2, 6, 4, 3, 0};
247   char receive[4]; // this holds the 4 bit integer aka the frequency
248   unsigned int freq = 0;
249   //this command advices the avr to send the frequency to the pi
250   bbI2CZip(2, cmd_buf, sizeof(cmd_buf), receive, sizeof(receive));
251   //set the received bytes properly together
252   freq |= receive[3];
253   freq |= freq << 8;
254   freq |= receive[2];
255   freq |= freq << 8;
256   freq |= receive[1];
257   freq |= freq << 8;
258   freq |= receive[0];
259   #ifdef DEBUG
260   printf("Counts: %i\n", freq);
261   #endif
262   return freq;
263 }


266 unsigned int read_counts_y()
267 {
268   /*
269     cryptic command to read frequency
270     4: after 4 it expects the I2C adress
271     2: start command (after that it expects a comma and then the actual command)
272     7: command that indicate you will send something followd by the number of bytes (here 1)
```

```
273       'F' the byte sent.
274       3: means stop. So here it means stop sending
275       2: start command
276       6: receive data followd by the number of received bytes (here 4 a uint_32)
277       3: stop receiving
278       0: end of command
279     */
280     char cmd_buf[] = {4, 0x05, 2, 7 ,1, 'F', 3, 2, 6, 4, 3, 0};
281     char receive[4]; // this holds the 4 byte integer aka the frequency
282     unsigned int freq = 0;
283     //this command advices the avr to send the frequency to the pi
284     bbI2CZip(2, cmd_buf, sizeof(cmd_buf), receive, sizeof(receive));
285     //set the received bytes properly together
286     freq |= receive[3];
287     freq |= freq << 8;
288     freq |= receive[2];
289     freq |= freq << 8;
290     freq |= receive[1];
291     freq |= freq << 8;
292     freq |= receive[0];
293     #ifdef DEBUG
294     printf("Counts: %i\n", freq);
295     #endif
296     return freq;
297 }



301 void start_scan(Measurement* m, int pi)
302 {
303     transfer_gatetime(m);
304     unsigned interval_m1 = (m -> m1_upper - m-> m1_lower); //total positions for motor 1
305     unsigned interval_m2 = (m -> m2_upper - m -> m2_lower); // total positions for motor 2

307     /*
308       allocate memory for data:
309       interval_m1 is the total number
310       of positions of scinitillator1
311       interval_m2 analog
312      */
313     errno = 0;
314     unsigned int* countPtr_x = malloc(interval_m1*sizeof(int));
315     if(errno != 0)
316     {
317       perror("Allocation error\n");
318       exit(EXIT_FAILURE);
319     }
320     unsigned int* countPtr_y = malloc(interval_m2*sizeof(int));
321     if(errno != 0)
322     {
323       perror("Allocation error\n");
324       exit(EXIT_FAILURE);
325     }
```

```
327   //before the measurement starts the scintilators have to be in their home positions

329   set_to_initial_position(m, pi);

331   gpio_write(pi, 27, 0); //set the correct direction for M1
332   gpio_write(pi, 9, 1); //sets the correct direction for M2

334   int m1 = 0, m2 = 0, i = 0;


337   //This was modified for th cosmic myoubn measurement.
338   //Sleep mode for ICs was changed to be in the loop
339   /*start rastering in the specified intervals*/
340   /*
341   while(m1 < interval_m1 || m2 < interval_m2)
342   {
343     if(m1 < interval_m1)
344     {
345       start_counter1(m);
346       countPtr_x[i] = read_counts_x();
347       raster_m1(pi);
348     }
349     if(m2 < interval_m2)
350     {
351       start_counter2(m);
352       countPtr_y[i++] = read_counts_y();
353       raster_m2(pi);
354     }
355     ++m1;
356     ++m2;
357   }
358   */
359   gpio_write(pi, 17, 1); //wake up IC
360   usleep(1000); //again: this is needed for charge pump see datasheet
361   while(m1 < interval_m1)
362   {
363     start_counter1(m);
364     countPtr_x[i++] = read_counts_x();
365     raster_m1(pi);
366     ++m1;

368   }

370   gpio_write(pi, 17, 0);

372   i = 0;
373   gpio_write(pi, 22, 1);
374   usleep(1000);
375   while(m2 < interval_m2)
376   {
377     start_counter2(m);
378     countPtr_y[i++] = read_counts_y();
```

```
379       raster_m2(pi);
380       ++m2;
381     }
382     gpio_write(pi, 22, 0);

384     //write data into a file
385     writeData(countPtr_x, countPtr_y, interval_m1, interval_m2);



389     free(countPtr_x);
390     free(countPtr_y);
391     gpio_write(pi, 17, 0); //set the IC to sleep
392     gpio_write(pi, 22, 0); //set the IC to sleep
393   }
```

Listing 9.8: properties.conf

```
1  # This is a config file that contains default parameters for
2  # 1. measurement time
3  # 2. scan area
4  # Change only to reasonable values. Program will check if the values
5  #here are nice and will report stupid values and will not start scanning.

7  # Note that TIME is NOT the actual time per positon.
8  #You can calculate the actual time by multiplying TIME with 1.28ms (so TIME = 100 is 128ms)
9  TIME = 255

11 # M1's (motor 1) largest scan interval is [0, 290]
12 M1_LOWER = 0
13 M1_UPPER = 290

15 # M2's (motor 2) largest scan interval is [0, 290]
16 M2_LOWER = 0
17 M2_UPPER = 290
```

Listing 9.9: plot.C

```
2  //
3  // root 'plot.C("left_up.dat")'
4  //


7  void plot(char szName[]="pechblende.dat") {

9    const int kNBins = 290;
10   int xx, yy, xc, yc;

12   TH1D *hx = new TH1D("hx","hx",kNBins, 0, kNBins);
13   TH1D *hy = new TH1D("hy","hy",kNBins, 0, kNBins);
```

```
15    TH2D *hxy = new TH2D("hxy", "", kNBins, 0, kNBins, kNBins, 0, kNBins);


18    ifstream fin(szName);
19    while(fin.good()) {
20      fin >> xx >> yy >> xc >> yc;

22      hx->Fill(xx, xc);
23      hy->Fill(yy, yc);

25      for(int i=0; i<kNBins; i++) {
26        hxy->Fill(xx, i, xc);
27        hxy->Fill(i, yy, yc);
28      }

30      cout << Form("%i %i %i %i\n", xx, yy, xc, yc);
31    }
32    fin.close();

34    // drawing all
35    //
36    TCanvas *cMain = new TCanvas();
37    cMain->Divide(2,2);

39    cMain->cd(1);
40    hx->Draw();
41    hx -> SetXTitle("Position");
42    hx -> SetYTitle("Counts");
43    cMain -> Modified();
44    cMain->cd(3);
45    hxy->Draw("colz");

47    cMain->cd(4);
48    hy->Draw();
49    hy -> SetXTitle("Position");
50    hy -> SetYTitle("Counts");
51    //hx -> SetXTitle("Position");
52    //hx -> SetYTitle("Counts");
53    hxy -> SetXTitle("Position X");
54    hxy -> SetYTitle("Position Y");
55    cMain -> Modified();

57    // writing output file with input name plus extension
58    //
59    cMain->SaveAs( Form("%s.root", szName) );
60    cMain->Print(  Form("%s.pdf",  szName) );

62  }
```

Literaturverzeichnis

1. Advamotion. (o. J.). *Clock Stretching Bug.* `http://www.advamation.com/knowhow/raspberrypi/rpi-i2c-bug.png/`. ([Online; zugegriffen 18.01.2018])

2. Allegro-Microsystem. (2014). *A4988: DMOS Microstepping Driver with Translator And Overcurrent Protection.* Allegro Microsystem, LLC.

3. Atmel. (2008). *AVR042: AVR Hardware Design Considerations.* Autor.

4. Atmel. (2015). *ATMEL 8-bit microcontroller with 4/8/16/32KBYTES in-system programmable flash Datasheet.* Autor.

5. Balazs, C. (2014, November). Baryogenesis: A small review of the big picture. *ArXiv e-prints*.

6. Demtröder, W. (2017). *Experimentalphysik 4 (5. Auflage).* Springer-Verlag GmbH Berlin Heidelberg.

7. Inventables. (o. J.). *Stepper Motor - NEMA 17.* `https://dzevsq2emy08i.cloudfront.net/paperclip/technology_image_uploaded_images/23984/default/Nema%2017.jpg?1372864563/`. ([Online; zugegriffen 18.01.2018])

8. Knoll, G. F. (2000). *Radiation Detecton and Measurement.* John Wiley & Sons.

9. Leo, W. R. (1994). *Techniques for Nuclear and Particle Physics.* Springer-Verlag Berlin Heidelberg GmbH.

10. Openbuildspartstore. (o. J.). *V-Slot® NEMA 17 Linear Actuator Bundle (Belt Driven).* `http://cdn8.bigcommerce.com/s-itwgldve/images/stencil/1280x1280/products/247/2471/vslot_actuator_belt_s2_w_1__01551.1509817421.jpg?c=2/`. ([Online; zugegriffen 18.01.2018])

11. PBC-Linear. (o. J.). *Stepper Motor NEMA 17.* `http://www.pbclinear.com/Download/DataSheet/Stepper-Motor-Support-Document.pdf/`. PBC Linear. ([Online; zugegriffen 30.01.2018])

12. Piatek, S. (o. J.). *Opto-semiconductor handbook Chapter 03 Si APD, MPPC .* http://www.hamamatsu.com/resources/pdf/ssd/e03_handbook_si_apd_mppc.pdf/. ([Online; zugegriffen 18.01.2018])

13. Piatek, S. (2017a). *Silicon Photomultiplier: Theory and Practice.* http://www.hamamatsu.com/us/en/community/optical_sensors/articles/sipm_theory_and_practice/index.html/. ([Online; zugegriffen 18.01.2018])

14. Piatek, S. (2017b). *A technical guide to silicon photomultipliers (SiPM).* https://www.hamamatsu.com/us/en/community/optical_sensors/articles/technical_guide_to_silicon_photomultipliers_sipm/index.html#/. ([Online; zugegriffen 18.01.2018])

15. Polulu. (o. J.). *A4988 Stepper Motor Driver Carrier.* https://a.pololu-files.com/picture/0J3360.600.png?d94ef1356fab28463db67ff0619afadf/. ([Online; zugegriffen 18.01.2018])

16. Raspberry-Pi-Foundation. (o. J.). *Raspberry Pi 2.* https://www.raspberrypi.org/app/uploads/2017/05/Raspberry-Pi-Model-B-462x322.jpg/. ([Online; zugegriffen 18.01.2018])

17. Scherz, P. & Monk, S. (2016). *Practical Electronics for Inventors (4. Auflage)*. Mac Graw Hill Education.

18. Tektronix. (o. J.). Arbitrary/Function Generators AFG3000C Series Datasheet.

19. Valdez, J. & Becker, J. (2015). *Understanding the I2C bus.* Texas Instruments.

20. Wikipedia. (o. J.). *ATMega 328P.* https://upload.wikimedia.org/wikipedia/commons/thumb/0/0c/ATMEGA328P-PU.jpg/1024px-ATMEGA328P-PU.jpg/. ([Online; zugegriffen 18.01.2018])

21. Zurek, M. (2017). *Searches for Electric Dipole Moments (EDM) at Storage Rings.* http://collaborations.fz-juelich.de/ikp/jedi/public_files/usual_event/MZSeminarEDM_v3.pdf. ([Online; zugegriffen 30.01.2018])