Slow Control und Datenauslese für den HiSCORE-Detektor

Diplomarbeit

von Robert Eichler

Institut für Experimentalphysik
Universität Hamburg

Hamburg,6. Februar 2011

Gutachter:
1. Prof. Dr. Dieter Horns
2. Prof. Dr. Caren Hagner

Kurzfassung

Im Rahmen dieser Arbeit sollen Elemente der Stationssteuerung für den in Entwicklung befindlichen HiSCORE-Detektor erstellt werden. Hierzu werden zunächst die Erfordernisse für die Steuerung der mechanischen Komponenten sowie die Aufnahme der Telemetriedaten festgelegt. Diese werden dann mit dem programmierbaren Arduino Mikrocontroller umgesetzt. Zudem muss die Integration des Arduino und der zugehörigen Hardware in die Station geplant werden.
Weiterhin muss das Programm, welches die von den Messinstrumenten eintreffenden Daten verwaltet, modifiziert werden, um einem externen Benutzer Kontrolle über die Datennahme zu gewähren. Zudem muss die Speichermethode des Programms optimiert werden, damit möglichst viele Ereignisse in kurzer Zeit gelesen werden können.
Am Ende der Arbeit steht das Erstellen einer Benutzerschnittstelle, die einem Benutzer an der Zentralstation von HiSCORE erlaubt, auf jede einzelne Station zuzugreifen, die mechanischen Systeme zu kontrollieren, die Telemetriedaten abzurufen, sowie die Datennahme zu steuern.

Abstract

Within this thesis the elements of the station control for the HiSCORE detector, which is currently being developed, are to be built. As a start, the requirements for the controlling of the mechanichal components as well as the telemetry data taking are defined. These will then be implemented using the programmable Arduino mocrocontroller. In addition, the integration of the Arduino and associated hardware will be planned.
Furthermore, the program used to administrate the data received from the measuring instruments has to be modified to grant control over data acquisition to an external user. In addition, the method of data saving has to be optimized to increase the number of events that can be processed in a certain amount of time.
The last thing to be done within this thesis is the construction of an interface, allowing a user at HiSCORE's central station to access every single station, control its mechanical systems, call for telemetry data and control data acquisition.

Inhaltsverzeichnis

Kapitel 1

Einleitung und Hintergrund

1.1 Experimentelle Astroteilchenphysik

Die experimentelle Astroteilchenphysik befasst sich mit der Beobachtung hochenergetischer Teilchen kosmischen Ursprungs. Dazu gehören u.A. die 1912 von Victor Hess entdeckte sogenannte kosmische Strahlung [28], kosmische Neutrinos und Photonen. Die kosmische Strahlung ist für diese Arbeit von zentraler Bedeutung, daher soll sie im Folgenden näher betrachtet werden.

Kosmische Strahlung Die kosmische Strahlung, die permanent auf die Erdatmosphäre trifft, lässt sich in zwei Kategorien einteilen, die geladene kosmische Strahlung und die ungeladene γ-Strahlung. Die geladene kosmische Strahlung besteht zu ca. 98 % aus Hadronen und zu ca. 2 % aus Leptonen [12]. Die hadronische Komponente wiederum besteht zum größten Teil aus Protonen.
Der Ursprung der hochenergetischen kosmischen Strahlung ist noch nicht entgültig geklärt, man weiß jedoch, dass der Teilchenfluss einem Potenzgesetz folgend mit der Energie abnimmt (siehe Abb. 1.1). Man nimmt allerdings an, dass Teilchen oberhalb eines Energiebereiches von 10^{17} bis 10^{18} eV einen extragalaktischen Ursprung haben [30].

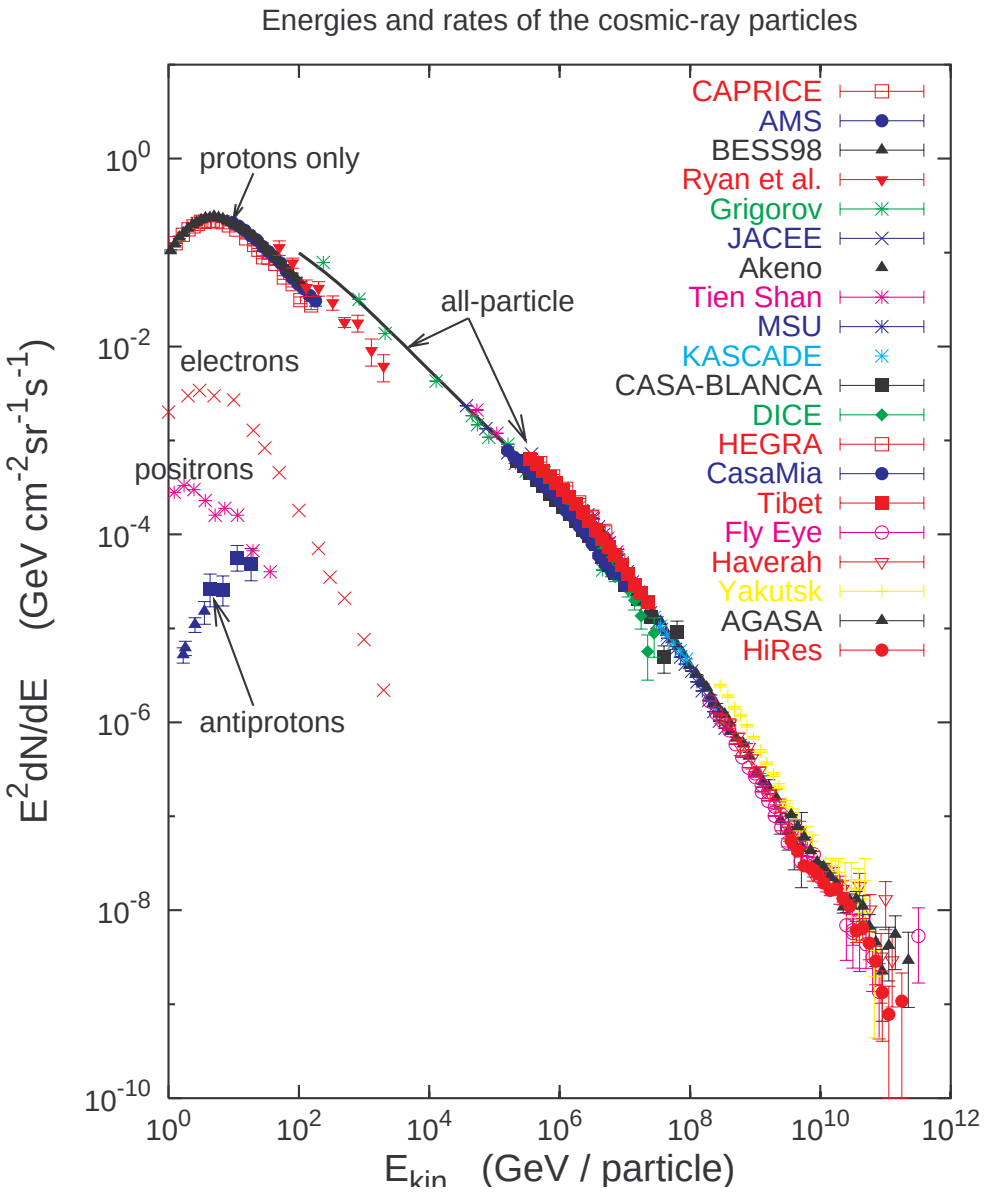

Abbildung 1.1: Das Spektrum der kosmischen Strahlung (aus [30])

Der HiSCORE-Detektor dient vorwiegend zur Detektion hochenergetischer Photonen, der sogenannten γ-Astronomie. Auf diese soll nun genauer eingegangen werden.

γ-Astronomie Eines der Hauptziele der γ-Astronomie ist es, die genauen Erzeugungsmechanismen und Quellen der hochenergetischen Photonen zu finden, vermutet wird bisher nur, dass die γ-Strahlung nicht thermischen Ursprungs ist, da die dafür nötigen Temperaturen und damit auch die Beschleunigung im bekannten Universum nicht erreicht werden.
Eine mögliche Erklärung für die Entstehung hochenergetischer Photonen bieten Prozesse mit geladener kosmischer Strahlung. Dazu zählt die Synchrotron-Strahlung, die unter An-

derem von hochenergetischen geladenen Leptonen oder Hadronen in stellaren Magnetfeldern abgegeben wird. Dieses Phänomen ist allerdings nur für Photonen im Röntgenbereich dominant [38]. Im γ-Bereich gibt es unterschiedliche Erzeugungsprozesse für Leptonen und Hadronen.
Der dominierende hadronische Erzeugungsprozess für hochenergetische Photonen ist die $\gamma\gamma$-Produktion durch neutrale Pionen, die aus Reaktionen hochenergetischer Protonen mit interstellarer Materie stammen (siehe z.B. [13], [15]). Die Erzeugung hochenergetischer Photonen in leptonischen Prozessen findet größtenteils über den inversen Comptoneffekt, die Streuung hochenergetischer Leptonen an Photonen des kosmischen Mikrowellenhintergrundes (CMB), statt (siehe z.B. [40]).
Aktuell werden hauptsächlich γ-Quellen im *High Energy* (HE) (30 MeV - 30 GeV) und *Very High Energy* (VHE) -Bereich (100 GeV - 30 TeV) beobachtet. Zur Erkundung des HE-Bereichs verwendet man sattelitengestützte Experimente (u.A. COS-B [37], EGRET [34], AGILE [41] oder aktuell FERMI [11]). Zur Beobachtung des VHE-Bereichs sind Sattelitenexperimente aufgrund ihrer nutzlastbedingt geringen Fläche bei hohen Energien nur bedingt aussagekräftig, da der Teilchenfluss bei hohen Energien zu gering wird (siehe vorheriger Paragraph). Daher bedient man sich zur Erkundung des VHE-Bereichs bodengestützter Experimente (z.B. HEGRA [9], HESS [31], HAWC [39], CTA [27], MAGIC [10]).
Das Problem vor dem die aktuellen Experimente bei der Suche nach der Quelle der γ-Strahlung stehen ist, dass das Spektrum in den bisher zugänglchen Energiebereichen mehrdeutig ist. Die Daten lassen keinen Schluss zu, ob die Photonen aus leptonischen oder hadronischen Prozessen stammen.
Da der Wirkungsquerschnitt des Comptoneffekts für große Energien (>10 TeV) stark abfällt, man spricht vom Klein-Nishina-Regime, ließe sich sich durch eine genaue Messung des Photon-Spektrums im *Ultra High Energy* (UHE) -Bereich jenseits von 30 TeV der Erzeugungsprozess der kosmischen γ-Strahlung bestimmen. Kennt man diesen, kann man auch die Quellen der Ausgangsteilchen und deren Beschleunigungsmechanismus erkennen.
Die meisten Teilchen aus der kosmischen Strahlung können nicht direkt mit Detektoren gemessen werden. Stattdessen werden Teleskope benutzt, die sowohl Hadronen als auch Leptonen und Photonen über ihre Interaktion mit der Atmosphäre beobachten: die Luftschauer.

Luftschauer Trifft kosmische Strahlung auf die Erdatmosphäre, so entstehen durch Interaktion der Teilchen mit den Atomkernen in der Atmosphäre weitere Partikel, die ihrerseits wieder mit der Atmosphäre interagieren. Auf diese Weise entstehen Teilchenkaskaden, die sogenannten Luftschauer. Hierbei unterscheidet man zwischen elektromagnetischen und hadronischen Schauern. Die elektromagnetischen Schauer werden von Photonen oder Elektronen ausgelöst und pflanzen sich als Kaskade von Photonen, Elektronen und Positronen fort. Hierbei erzeugen die Photonen in einem als Paarbildung bekannten Prozess e^+e^--Paare. Die Elektronen und Positronen wiederum geben jedes Mal wenn sie eine, von der Dichte der Atmosphäre abhängige, Weglänge X_0 zurückgelegt haben Bremsstrahlung in Form eines Photons ab. Die Kaskade läuft nicht weiter, wenn die Energie der abgegebenen Photonen unter den für die Paarbildung nötigen Wert sinkt [25].
Ein hadronischer Schauer ist in seiner Morphologie Komplexer, wobei vereinfacht angenommen wird, dass ein Kern mit Ordnungszahl Z und Energie E sich wie Z Protonen mit der Energie E/Z verhält. In der Atmosphäre entstehen durch die starke Wechselwirkung hauptsächlich Pionen, wobei aufgrund der Isospinsymmetrie $\pi^\pm$ und π^0 gleich häufig vorkommen.

Für hohe Energien dominiert für die Pionen die Erzeugung neuer hadronischer Schauer, da hier die mittlere freie Weglänge der Pionen klein gegen die Zerfallslänge $\lambda_\tau = \gamma\beta c\tau$ ist[1]. Mit sinkender Energie wird die Wahrscheinlichkeit, dass ein Pion zerfällt, bevor es interagieren kann, stetig größer, bis der Zerfall zum dominierenden Prozess innerhalb des Schauers wird. Die neutralen Pionen zerfallen bevorzugt in Photonenpaare, die ihrerseits bei ausreichender Energie elektromagnetische Schauer erzeugen. Geladene Pionen zerfallen hingegen hauptsächlich in Myonen und Neutrinos. Die Neutrinos gelangen größtenteils ohne weitere Wechselwirkung mit der Atmosphäre zum Boden, während die Myonen zum Teil in Elektronen bzw. Positronen und Neutrinos zerfallen.
Die Schauerteilchen emittieren in der Atmosphäre sogenanntes Cherenkov-Licht, wenn ihre Geschwindigkeit größer als die Lichtgeschwindigkeit c' $= \frac{c}{n}$ im sie umgebenden Medium mit Brechungsindex n ist, wobei der Winkel ϕ des abgestrahlten Lichtes zur Bewegungsrichtung des erzeugenden Teilchens von dessen Geschwindigkeit und dem Brechungsindex abhängt: $\cos(\phi) = \frac{1}{\beta n}$.
Mit den rekonstruierten Schauerparametern, wie Energie oder Tiefe, lässt sich das erzeugende Teilchen identifizieren [29]. Die in der Kaskade vorhandenen Partikel werden bei den meisten Experimenten nicht selbst eingefangen, sondern über das von ihnen in der Erdatmosphäre erzeugte Cherenkov-Licht beobachtet. Dieses bildet aufgrund der hohen Energie der Ausgangsteilchen einen Kegel mit einem sehr geringen Öffnungswinkel um die Bewegungsrichtung des ursprünglichen Teilchens. Allerdigs sind auch die meisten bisherigen Cherenkov-Teleskope in ihrem Energiebereich limitiert, da aufgrund des schnell abfallenden Teilchenflusses bei hohen Energien eine Detektion oberhalb einer Teilchenenergie von 100 TeV nur mit einer sehr viel größeren Detektorfläche als bisher vorhanden möglich ist.

1.2 HiSCORE

Um das Energieregime von 100 TeV bis 1 EeV für geladene kosmische Strahlung, bzw. 10 TeV bis zum PeV-Bereich für γ untersuchen zu können, muss also ein Detektor gebaut werden, der auch bei kleinem Teilchenfluss genug Daten für eine statistische Auswertung liefern kann. Hierfür wird HiSCORE (**H**undred***i** **S**quare-km **C**osmic **OR**igin **E**xplorer) entwickelt, dessen große Fläche von hundert km^2 den geringen Fluss hochenergetischer Teilchen kompensieren soll (siehe Abb. 1.2 und Abb. 1.3).

[1] τ ist die Lebensdauer der Pionen

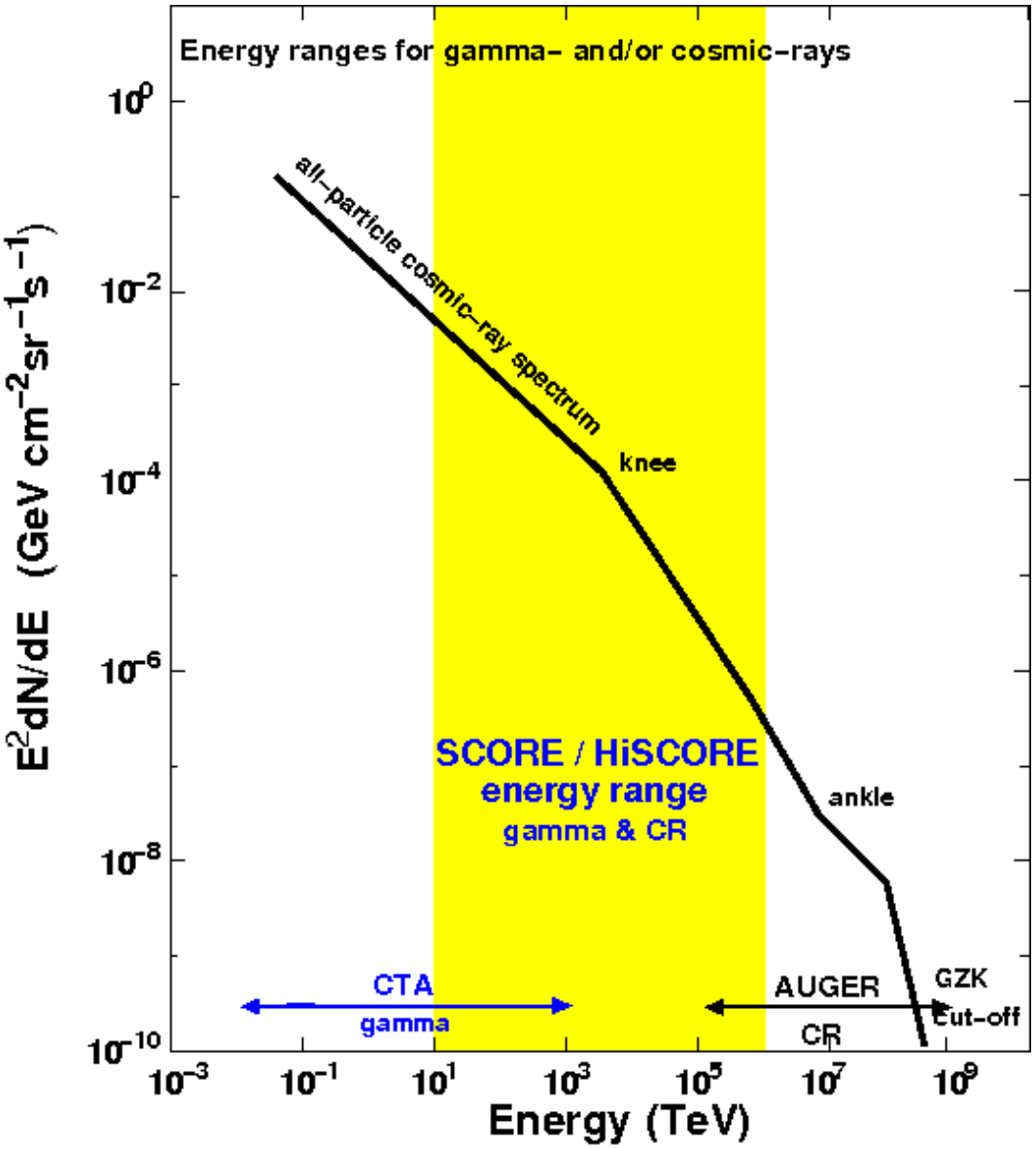

Abbildung 1.2: Observationsbereich verschiedener Teleskope (Darstellun von M. Tluczykont, basierend auf [30])

Abb. 1.2 zeigt den Observationsbereich und Abb. 1.3 die Flusssensitivität[2] verschiedener Teleskope. Um die Sensitivität von HiSCORE abzuschätzen wurde die effektive Triggerfläche für γ-Strahlen und Protonen berechnet. Dazu wurden auf das Detektorarray niedergehende Schauer simuliert und das Verhältnis zwischen den Anzahlen der getriggerten und der simulierten Ereignisse geteilt durch die Detektorfläche bestimmt [42]. Hieraus wurde die Anzahl der Signal- und Hintergrundereignisse berechnet. Die Sensitivität für Punktquellen wurde

[2]Die Flusssensitivität ist der minimale Energiefluss, den ein Teleskop mit einer Signifikanz von $5\,\sigma$ über dem Hintergrund auflösen kann

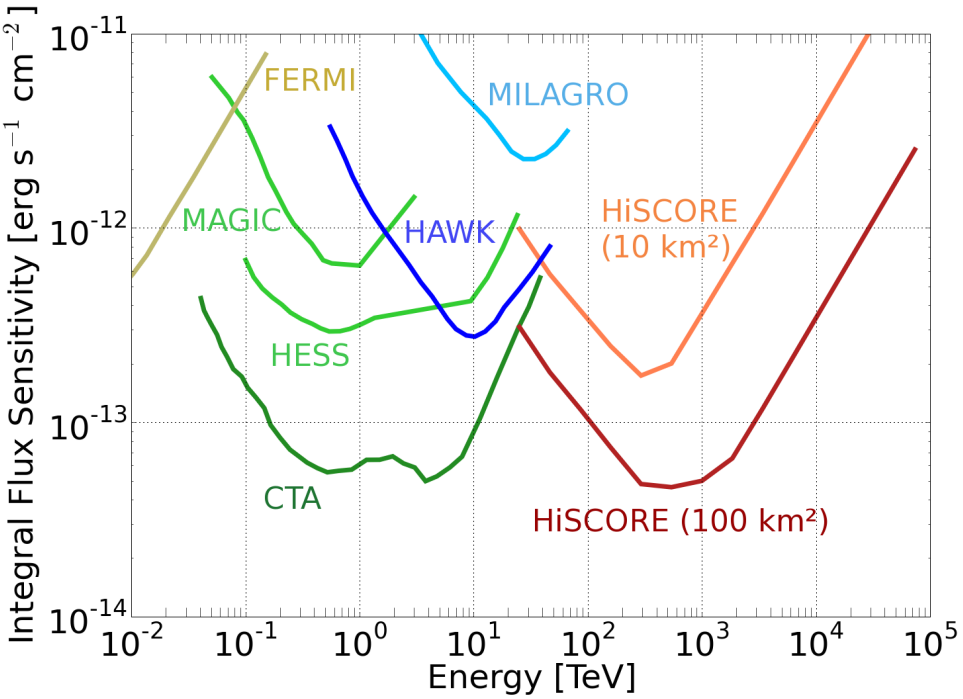

Abbildung 1.3: Sensitivitätskurven verschiedener Teleskope (Aus: [22])

dann unter der Bedingung on 50 γ-Ereignissen mit einer Signifikanz von $5\,\sigma$ innerhalb von 5 Jahren abgeschätzt ([43], [23]).

1.2.1 Das Detektorkonzept

Der Detektor besteht aus vielen einzelnen Stationen, die in einem Gitter mit Stationsabstand von etwa 100 bis 200 m zueinander haben sollen ([42]). Auf diese Weise deckt man eine große Fläche ab, behält aber trotzdem eine gute Auflösung der Cherenkov-Kegel einzelner Ereignisse bei. Es werden sowohl die Ankunftszeitverteilung als auch die Energiemenge des Cherenkov-Lichts gemessen, um den Schauerkernort und die Energie des ursprünglichen Teilchens rekonstruieren zu können. Diese Technik , bei der die Form des Luftschauers direkt aufgenommen wird, bezeichnet man als nicht abbildende Cherenkov Technik. Bei der abbildenden Cherenkov Technik (u.A. HESS [31]) wird das einfallende Licht mit speziellen Spiegeln auf eine hochauflösende Kamera geworfen um aus diesen Bildern die Schauerrichtung zu rekonstruieren. Die Vorteile der nicht abbildenden Technik für die γ-Astronomie liegen u.A. in der gegenüber der abbildenden Technik um den Faktor 100 geringere Anzahl an Auslesekanälen und dem großen Gesichtsfeld[43], [42].
Eine einzelne Station verfügt über 4 Photomultiplier (PMTs), deren Lichtausbeute durch sogenannte Winston-Cones, trichterförmige Lichtsammler[3], noch gesteigert wird. Der Grund, warum hier nicht nur ein einziger Photomultiplier benutzt wird, ist, dass so falsche Sig-

[3]Höhe: 52 cm, Durchmesser: 40 cm (oben), 20 cm (unten)

nale besser unterdrückt werden können, da man nur Ereignisse weiterverfolgt, bei denen mehrere PMTs in einer Station ansprechen. Einen Auswahlmechanismus, der die zu verarbeitende Datenmenge verringern soll, indem er schon während der Datennahme Ereignisse, die nicht von Interesse sind, anhand äußerer Kriterien, d.h. ohne die Ereignisdaten zu betrachten, ausschließt, bezeichnet man als Trigger. Die Bedingung, dass mehrere PMTs gleichzeitig ansprechen müssen, gehört bei HiSCORE zur ersten Triggerstufe. Es wird ein Diskriminator eingesetzt, der mit den aufsummierten Spannungspulsen aller PMTs einer Station gespeist wird. Überschreitet das eingespeiste Signal einen bestimmten Schwellenwert, dargestellt von der Anzahl der erzeugten Photoelektronen, für eine bestimmte Zeit, so wird ein Triggersignal gesendet und das Ereignis erreicht die zweite Triggerstufe.
Auf der zweiten Stufe wird verglichen, ob mehrere benachbarte Stationen innerhalb eines bestimmten Zeitfensters ein Signal verzeichnet haben, da es sich dann nur um ein Cherenkov-Ereignis und nicht um eine Fluktuation des Nachthimmelsleuchtens handeln kann.

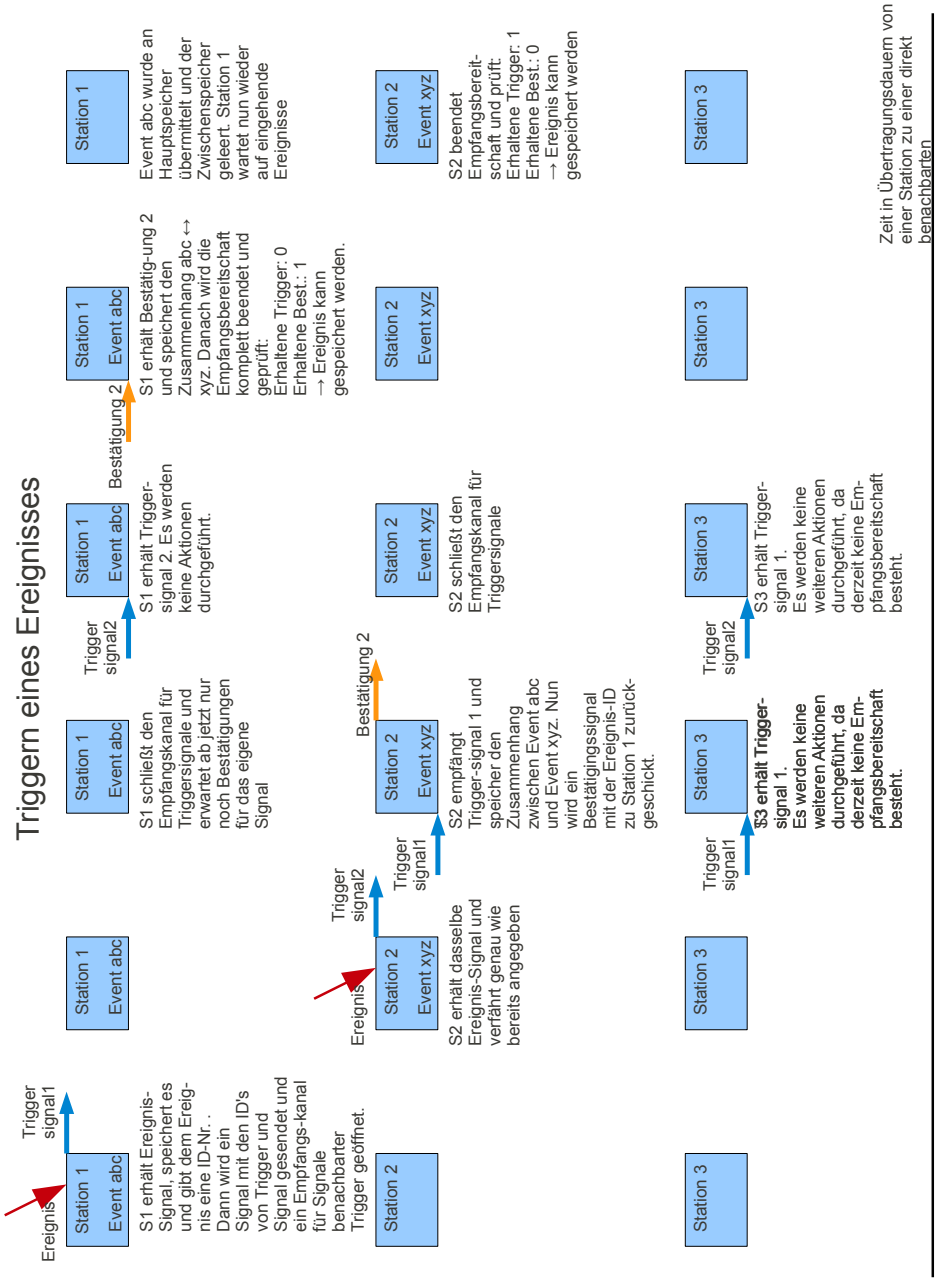

Abbildung 1.4: Illustration des Triggerkonzepts mit direkter Kommunikation der Stationen untereinander

Es wurde die Möglichkeit geprüft, mit dem in jede Station eingebauten Arduino-Mikrocontroller (Kap.2.2), der mittels eines speziellen Moduls, dem sog. XBEE-Modul, mit Arduinos in anderen Stationen in einem drahtlosen Netzwerk verbunden wird. Es zeigte sich jedoch, dass die Kommunikation der XBEEs untereinander zusammen mit der Verarbeitungsgeschwindigkeit der Arduinos bei einer erwarteten Ereignisrate von 1000 Hz bei weitem nicht ausreicht.

Zudem entstehen bei hoher Datenrate viele Fehler in der Kommunikation [18].
Zur genauen Arbeitsweise der zweiten Triggerstufe gab es (u.A. im Rahmen von [18]) verschiedene Überlegungen. Es ist vorgesehen, mit einer geeigneten Hardware ein drahtloses Netzwerk zwischen den Stationen einzurichten, über das ein Austausch von Triggersignalen stattfindet. Die genauen Kommunikationswege hierfür sind jedoch noch nicht festgelegt. Zum einen ist ein Servermodell denkbar, in dem die Stationen nicht direkt miteinander kommunizieren, sondern ihre Triggersignale und möglicherweise auch ihre Ereignisdaten an einen Server schicken, der dann die Triggersignale mehrerer Stationen vergleicht und über das Speichern der Daten entscheidet. Um die Totzeit gering zu halten, kann es hierbei mehrere Server innerhalb des Detektorarrays geben, von denen jeder nur mit einer kleinen Anzahl von benachbarten Stationen verbunden ist. Die Server sollen wiederum auch untereinander kommunizieren, um Triggersignale besser auflösen zu können. Ein zweites mögliches Triggermodell ist in Abb. 1.4 gezeigt. Hierbei kommunizieren die Stationen direkt miteinander, die Triggerentscheidung wird von jeder Station einzeln getroffen. Fällt diese positiv aus, so werden die Daten an eine Zentralstation geschickt.
Um die Triggerrate des Detektors[4] vorhersagen zu können, wurden Simulationen des zu erwartenden γ-Flusses für den Detektor durchgeführt. Hierbei geht es sowohl um den Fluss hochenergetischer kosmischer Strahlung als auch um das Nachthimmelsleuchten (**N**ight **S**ky **B**ackground).
Der NSB stellt ein Problem für das Experiment dar, da die vom Himmelsleuchten ausgelösten Spannungspulse die tatsächlichen Cherenkov-Signale in den Stationen überlagern können. Die vorher bereits angesprochene erste Triggerstufe soll dazu dienen den NSB auszuschließen. Hierzu müssen die genauen Triggerbedingungen so gewählt werden, dass die Triggerrate für den NSB möglichst gering ist, ohne jedoch zu viele Cherenkov-Signale zu verlieren. Abb. 1.5 zeigt die Triggerrate einer Station in Abhängigkeit von der Energieschwelle für verschiedene Pulslängen.

[4] Gemeint ist die Frequenz mit der Ereignisse eintreffen, die die Triggerbedingungen erfüllen

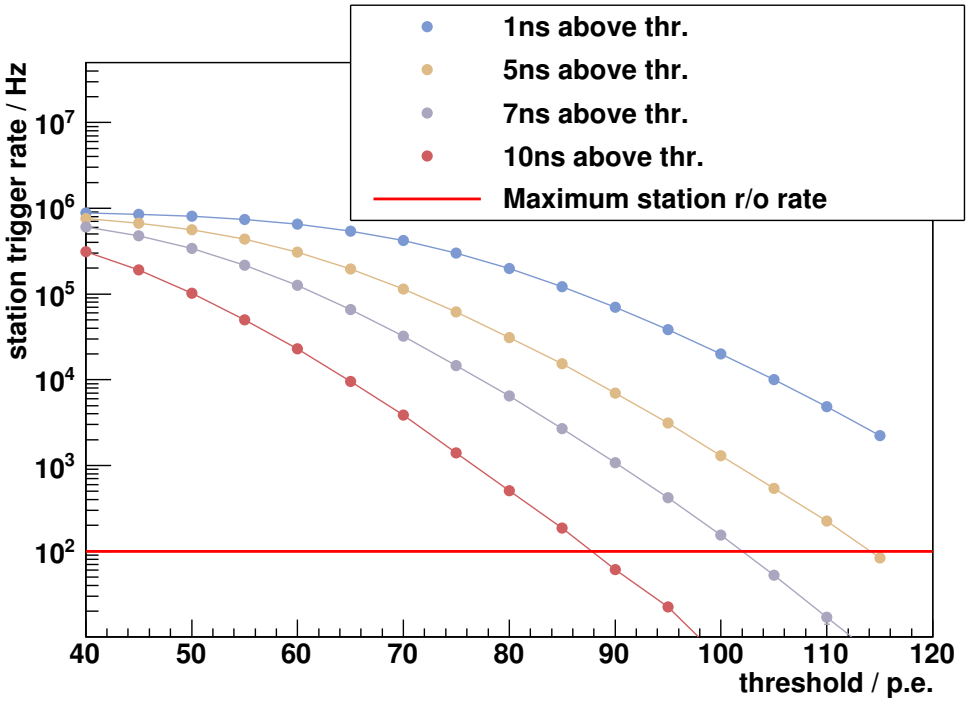

Abbildung 1.5: Triggerrate des Nachthimmelsleuchtens in einer Station (Simulation und Graphik von Dr. Martin Tluczykont)

Bisher wird beabsichtigt, eine Pulsbreite von 7 ns und eine Schwelle von 100 bis 110 Photoelektronen zu benutzen. Unter diesen Annahmen liegt die erwartete Triggerrate für hadronische Ereignisse bei etwa 20 Hz und für myonische Ereignisse bei unter 40 Hz. Diese Werte stammen aus den aktuellen Simulationen von Dr. Martin Tluczykont (bisher nicht veröffentlicht). Vergleicht man diese simulierten Werte mit den in Abb. 1.5 gezeigten Triggerraten für den NSB, so zeigt sich, dass die Gesamttriggerrate durch das Nachthimmelsleuchten dominiert ist.

1.2.2 Das Stationskonzept von HiSCORE

Wie bereits in der vorangegangenen Sektion erklärt, besteht der HiSCORE-Detektor aus identischen, einzeln stehenden und unabhängig voneinander steuerbaren Stationen. Die Steuerung soll innerhalb eines drahtlosen Netzwerks erfolgen, das die Stationen untereinander und mit der zentralen Kontrollstation verbindet.
Jede Station wird von einer 12 V Spannungsquelle, wie einer Batterie, einer externen Zuleitung oder einem Solarpanel, versorgt [42] [5]. Die bereits erwähnten Photomultiplier benötigen eine Betriebsspannung[6] von 2000 V, welche von einem Gerät der Firma ISEG, Prod. Nr. PHQ9352 aus der 12 V Betriebsspannung generiert wird. An die Photomultiplier ist zur Datenauslese ein DRS4-Board (siehe Kap. 3.1.2) angeschlossen, welches über ein USB-Kabel mit einem GuruPlug-Rechner (siehe Kap. 3.1.1) verbunden ist, der von der Batterie gespeist wird.

[5] nachfolgend wird hier von einer Batterie ausgegangen
[6] nachfolgend auch als Hochspannung oder HV bezeichnet

Zur Kontrolle der internen Systeme der Station dient ein Arduino-Mikrocontroller (siehe Kap. 2.2). Dieser ist über seinen seriellen Port mit einer USB-Schnittstelle des GuruPlug verbunden und erhält so seine Betriebsspannung, zudem werden über die Verbindung die vom Benutzer gegebenen Anweisungen an den Arduino weitergereicht. Neben der Einstellung des HV-Gerätes soll der Arduino u.A. zur Einstellung der Triggerschwelle und zur Überwachung der HV und Batteriespannung, sowie des Stroms im HV-System der Station eingesetzt werden. Weiterhin sind an den Mikrocontroller verschiedene Sensoren angeschlossen, die zum Aufzeichnen von Umweltdaten dienen (Kap. 2.2).
Ein Aluminiumgehäuse soll die empfindlichen Geräte vor der Witterung schützen und außerdem die Station lichtdicht abschließen. Das Gehäuse besitzt einen beweglichen Deckel, der mit einem Motor geöffnet und geschlossen und von einem Elektromagneten in geöffneter Position festgehalten werden kann. Sowohl der Motor als auch der Magnet werden von der Batterie gespeist und vom Arduino über eine Relaisschaltung gesteuert. Weiterhin sind am Deckel Sensoren angebracht, durch die der Arduino dessen Öffnungszustand überwachen soll. Eine Veranschaulichung des grundlegenden Aufbaus der Systeme ist in Abb. 1.6 zu sehen

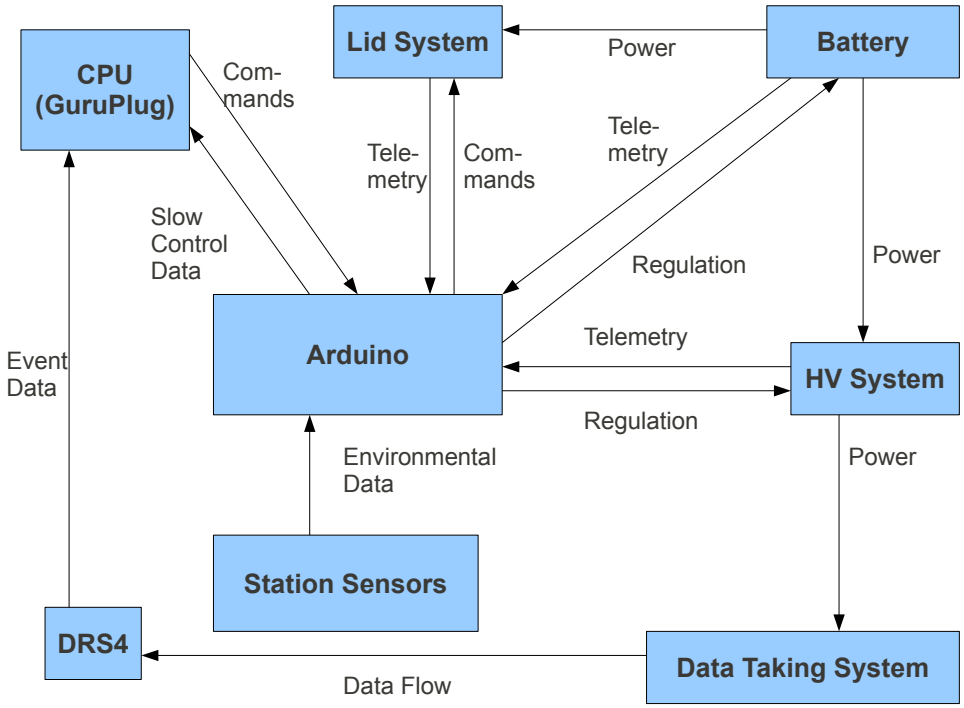

Abbildung 1.6: Schematische Darstellung der Stationssysteme und der Kommunikationswege zwischen den einzelnen Komponenten

Für eine Beschreibung der Station siehe auch [42].

1.3 Ziele dieser Arbeit

Im Rahmen dieser Diplomarbeit soll folgendes erreicht werden:

- Festlegung der Erfordernisse der Stationssteuerung
- Umsetzung der Stationssteuerung in einem Programm
- Test der Stationssteuerung
- Planung der Integration der Steuerelemente in die Station
- Implementation der nötigen Mechanismen für eine externe Kontrolle der Datennahme
- Optimieren der Datenspeicherung
- Erstellen einer Schnittstelle, die einem Benutzer an der Zentralstation Zugriff auf die Kontrollen des gesamten Detektorarrays gestattet

Kapitel 2

Aufbau einer Slow Control für HiSCORE

2.1 Aufgaben

Unter dem Begriff *Slow Control* versteht man diejenigen Teile der Steuerung des Detektors, die im experimentellen Rahmen wenig zeitkritisch sind. Während die Messelektronik oder der Trigger eine minimale Verzögerung haben dürfen, um effizient arbeiten zu können, kann die Steuerung der mechanischen Komponenten oder der Stromversorgung eine Reaktionszeit $\mathcal{O}$(s) haben. Weiterhin gehört zur Slow Control auch das Erfassen von Telemetriedaten sowie Umweltdaten, die für das Funktionieren des Detektors wichtig sein können. Im Rahmen dieser Arbeit sollen folgende Elemente realisiert und in die Station integriert werden (siehe auch schematische Darstellung in Abb. 2.1):

- Sensoren/Eingaben:
  - Temperatursensor (Bereich 0 bis 100 °C)
  - Lichtstärke
  - Aktivierungsstatus des HV-Systems
  - Spannung im HV-System (Bereich: 0 bis 2000 V, Genauigkeit: 9,8 V)
  - Strom im HV-System (Bereich: 0 bis 200 μA, Genauigkeit: 0,98 μA)
  - Batteriespannung (Bereich: 0 bis 12 V, Genauigkeit: 0,01 V)
  - relative Luftfeuchtigkeit
  - Schließsensor des Deckels der Station
- Ausgaben:
  - Steuerung des Motors zum Öffnen und Schließen des Deckels
  - Steuerung des Elektromagneten zum Fixieren des Deckels
  - Ein- und Ausschalten der HV
  - Regulierung der HV
  - Regulierung der Diskriminatorschwelle für die Photomultiplier

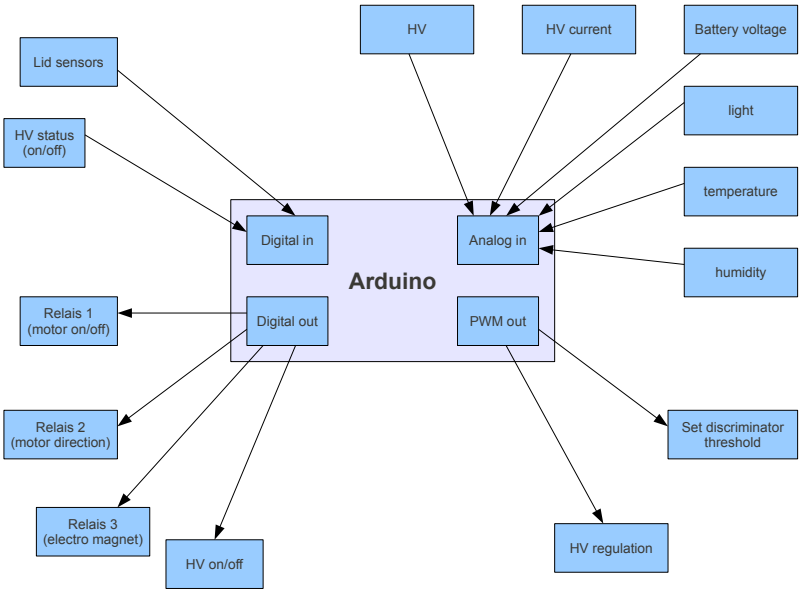

Abbildung 2.1: Schematische Darstellung der Slow Control Ein- und Ausgaben

2.2 Benutzte Hardware

2.2.1 Der Arduino-Mikrocontroller

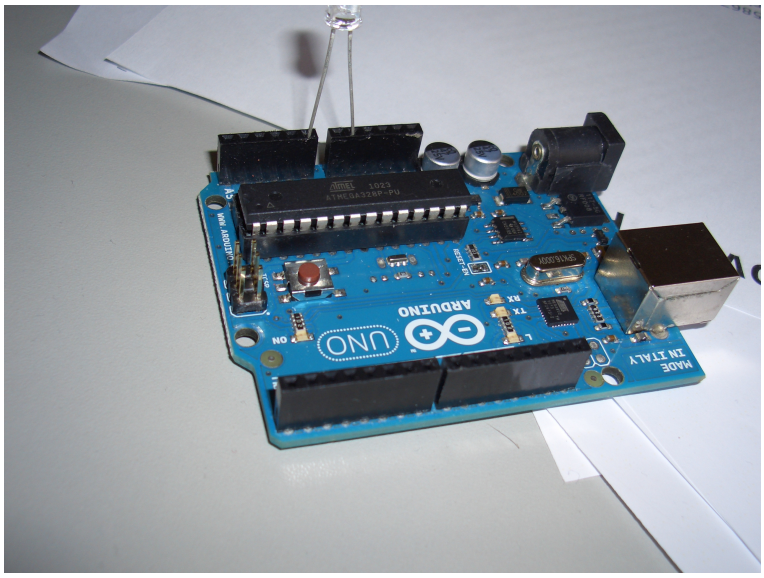

Abbildung 2.2: Der *Arduino UNO*

Das zentrale Element der Slow Control ist der Arduino Mikrocontroller (siehe Abb. 2.2). Dieser hat eine maximale Rechengeschwindigkeit von etwa 20 MHz und besitzt 32 kB inter-

nen Flash-Speicher. Das Arduino-Board hat 13 digitale Ein- und Ausgänge, sowie 6 analoge Eingänge, so genannte *Pins*. Die maximale Eingangsspannung für die digitalen und analogen *Pins* liegt bei 5 Volt, was auch die Ausgangsspannung der digitalen *Pins* ist. Die analogen *Pins* lesen eine eingegebene Spannung in insgesamt 1024 Stufen. Da die Maximalspannung bei 5 V liegt, beträgt die Messgenauigkeit also ca. 4,89 mV. Die digitalen *Pins* werden sowohl für die Ein- als auch für die Ausgabe von Signalen benutzt. Zum einen ist es möglich, per *digitalWrite* ein simples binäres Ein/Aus Signal zu geben, um z.B. LEDs zu betreiben. Zum anderen verfügen einige der *Pins* auch über die Möglichkeit zur Pulsweitenmodulation (PWM): Die Spannung wechselt in einer Rechteckwelle zwischen 0 und 5 Volt, wobei der effektive Ausgabewert durch das Verhältnis zwischen den Zeiten bei 0 und 5 innerhalb einer konstanten Zeitspanne (Zyklusdauer) gesteuert wird. Beispielsweise führt eine Verteilung von 1:1 zu einer Ausgabe von 50 Prozent des Maximalwertes, während ein Verhältnis von 1:3 75 Prozent generiert.
Für die Programmierung und die Tests wurde hier das Modell *Arduino duemilianove* benutzt, das aber vom Hersteller inzwischen durch den funktionsgleichen *Arduino UNO* ersetzt worden ist.
Der *Arduino duemilianove* wurde auch von Jan Drewes im Rahmen von dessen Bachelorarbeit (Kalibration einer CCD-Kamera) verwendet, um eine LED-Matrix als Referenzpunkt für die Kameras zu steuern. Die Programme hierfür wurden vom Autor dieser Arbeit verfasst (siehe Anhang A.4).

2.2.2 Das XBee-Modul

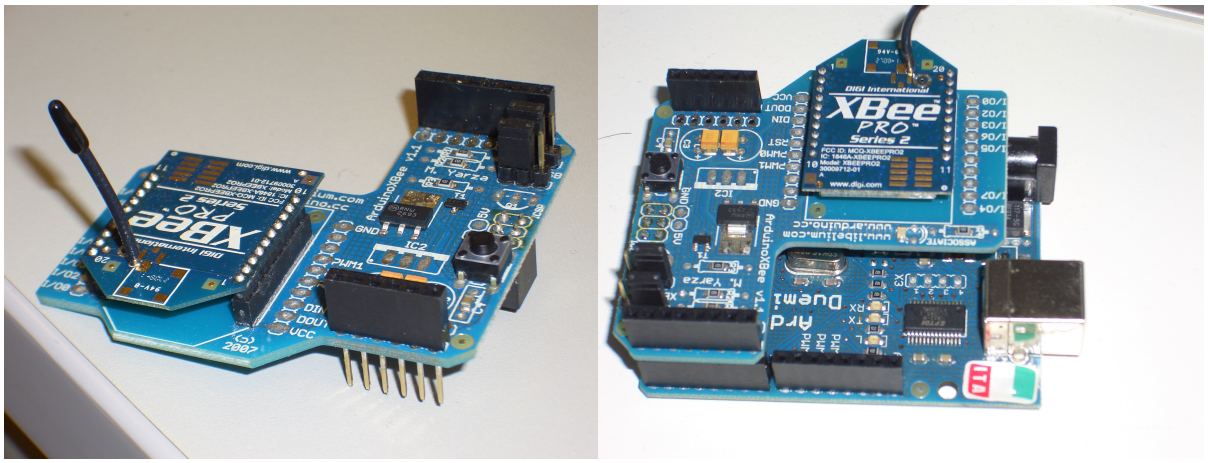

Abbildung 2.3: Das XBee-Modul

XBee-Module können zur Kommunikation zwischen Arduino-Boards eingesetzt werden. Um sie einzusetzen, muss man sie auf ein spezielles Bauteil stecken, den sogenannten XBee-Shield, welches dann wiederum auf das Arduino-Board gesteckt wird. Die hier benutzten Module gehören zur *Series 2* (genauer: XBee Pro Series2).
Anstatt wie ihre Vorgänger einfach nur miteinander zu kommunizieren bauen sie ein komplettes Netzwerk auf, ein sog. *Mesh Network*, in dem die Module nicht nur Daten senden und empfangen sondern auch automatisch an andere Netzwerkmitglieder weiterleiten. Die Struktur eines solchen Netzwerkes ist komplizierter, da ein Modul die Aufgabe des *Koordinators*

übernehmen muss, während die anderen als *Router* oder *End Device* definiert sind. Der *Koordinator* initialisiert bei Inbetriebnahme ein neues Netzwerk und erlaubt anderen Modulen den Beitritt, ein *Router* kann ebenfalls anderen Modulen erlauben, in das Netzwerk einzutreten. Beide Typen können nicht nur Daten senden und empfangen, sondern sie auch weiterleiten. Nicht so das *End Device*, dieses kann nur senden und empfangen. Die Netzwerkarchitektur ist in Abb. 2.4 dargestellt.

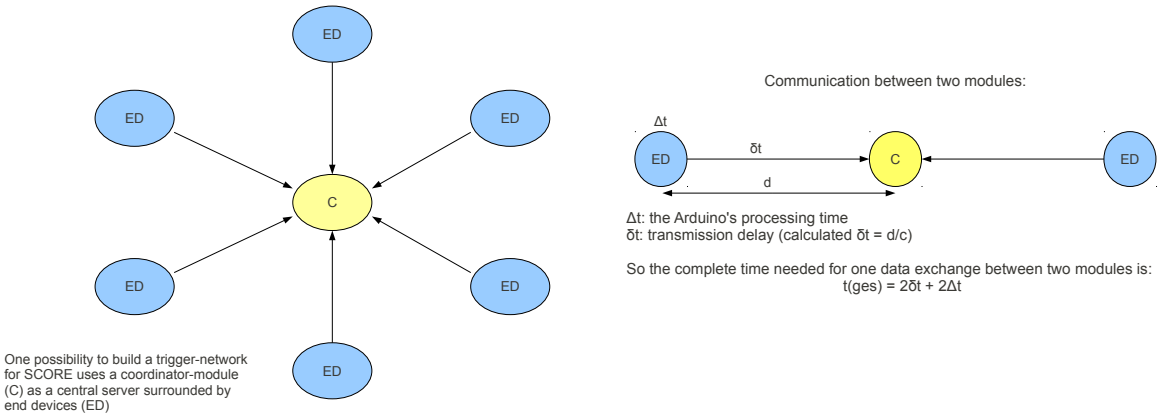

Abbildung 2.4: Darstellung der Architektur eines *Mesh Networks*

Der Typ eines Moduls wird durch dessen Firmware festgelegt. Um diese zu installieren muss das Modul auf einem Arduino-Board angebracht werden, das sich entweder im *Reset*-Modus befindet (Drahtbrücke zwischen *Reset-* und *Ground-Pins*) oder dessen Prozessor entfernt wurde. Dann schließt man es an den Rechner an und startet das Programm XCTU, welches zur Konfiguration genutzt wird. Hier kann man nun alle Eigenschaften des Moduls, wie etwa die Übertragungsrate, die Zieladresse oder die individuelle Seriennummer ablesen. Diese Eigenschaften können auch größtenteils geändert werden, wobei es zu beachten gilt, dass in der Grundeinstellung die Zieladresse so gewählt ist, dass das ganze Netzwerk angesprochen wird. Wichtig ist auch die Einstellung der Firmware, welche am Ende der Versionsbezeichnung steht, denn die unterschiedlichen Firmware-Typen interagieren auf verschiedene Weise mit dem Arduino. In der transparenten AT-Einstellung werden alle Parameter vor Inbetriebnahme gewählt und bleiben konstant, solange das Modul läuft. AT steht für *attention* und ist ein Standardbefehlssatz zum Konfigurieren von Modems und anderen Netzwerkgeräten (siehe `http://docs.kde.org/development/de/kdenetwork/kppp/appendix-hayes-commands.html`) Die Kommunikation der Arduinos über das Netzwerk findet hierbei über die normale serielle Schnittstelle statt, es sind also keine speziellen Befehle nötig. In der API (*Application Programming Interface*)-Einstellung hingegen kommunizieren das Arduino-Board und das XBee-Modul direkt miteinander, denn Sende- und Lesebefehle müssen mit allen nötigen Parametern, wie z.B. der Zieladresse, im Programmcode enthalten sein. Hierfür existiert eine Bibliotheksdatei, welche in das Programm importiert werden kann. Die AT-Einstellung ist einfacher zu Handhaben und für simplere Aufgaben besser geeignet, da der Code sehr kurz gehalten werden kann, während die API-Einstellung eine vielfältigere Anwendung des XBee-Moduls ermöglicht. Die Einfachheit der AT-Einstellung ist für die durchgeführten Tests von großem Vorteil, daher wird im Folgenden diese Variante genutzt.

2.2.3 Andere Stationskomponenten

Die HV-Spannung wird vom Gerät der Firma ISEG in einen Input von 0 bis 5 Volt umgewandelt, der im Maßstab dem Wert der Hochspannung von 0 bis 2000 Volt entspricht, während von der Batteriespannung mittels eines Spannungsteilers ein Drittel, also maximal 4 Volt an den Arduino abgegeben wird. Damit von außen festgestellt werden kann, ob das ISEG-Gerät eingeschaltet ist, gibt dieses, solange die Hochspanning aktiviert ist, auf einem separaten Ausgang ein Spannungssignal von 5 Volt aus. Zudem wird auch die Regelung der HV-Spannung mit einer analogen Eingabe im Bereich von 0 bis 5 V an das ISEG-Gerät vorgenommen. Alle Signale vom ISEG-Gerät werden in den Arduino eingespeist.
Die Messung von Temperaturen ist mit dem Arduino verhältnismäßig einfach durchzuführen. Es muss ein Temperatursensor mit einem der analogen *Pins* verbunden und dann eine passende Arbeitsspannung von 5 Volt, die vom Arduino geliefert wird, angelegt werden. Die Spannung, welche aus dem Stromkreis abgegriffen und an den Arduino abgegeben wird, richtet sich dann nach der Temperatur in der Umgebung des Sensors. Da hier meist mit Ausgaben über die serielle Schnittstelle gearbeitet wird, ist es natürlich nötig, die gelesenen Spannungswerte zu übersetzen, um einen Temperaturwert zu erhalten. Die Übersetzungsfunktion ist hierbei modellabhängig, da unterschiedliche Temperaturfühler natürlich auch unterschiedliche Skalen, also °C pro mV, und unterschiedliche Offsets, also Ströme bei 0 °C, haben können. Die ersten Messungen wurden mit einem LM35 Thermowiderstand durchgeführt. Dieser hat einen Offset von 0 und Spannungssteigerung von 10 mV pro °C. Die Messgenauigkeit liegt demzufolge bei ca. 0,5 °C. Dies genügt für die Zwecke von HiSCORE, da Fehlfunktionen oder andere Schäden,

wie z.B. Kabelbrände, eine weit größere Schwankung in der Temperatur innerhalb der Station verursachen würden.
Die Messung der Lichtstärke muss im Bereich der Slow Control weniger exakt, dafür allerdings verlässlicher sein als die der Temperatur, denn der Zweck dieser Messung liegt nicht im Aufzeichnen von Wetterdaten, sondern als Sicherheitskriterium für die Datennahme. Man kann nur dann mit dem HiSCORE Detektor sinnvolle Daten nehmen, wenn sicher ist, dass das Cherenkov-Licht aus den Luftschauern nicht von anderen Lichtquellen überlagert wird. Liegt die Lichtintensität also über einem bestimmten Schwellwert, so ist davon auszugehen, dass der Lichtuntergrund, z.B. durch den Mond oder künstliche Beleuchtung, zu stark ist. Die Funktionsweise dieser Messung unterscheidet sich leicht von der Temperaturmessung. Beim verwendeten Lichtsensor handelt es sich um eine Fotodiode, was bedeutet, dass hier die Änderung der Leitfähigkeit mit der Lichtintensität zu einer Änderung der Leitfähigkeit der Diode führt. Der Strom fließt durch die Diode direkt von der Spannungsquelle zu analogen *Pin*.
Für die Versuche wurde eine SFH203-Fotodiode verwendet. Diese hat bei normalem Tageslicht und einer Arbeitsspannung von 5 V einen Fotostrom von ca. 80 μA (laut Datenblatt), der Sensitivitätsbereich[1] der Diode reicht von 400 bis 1100 nm und der Dunkelstrom soll weniger als 5 nA betragen. Die Photodiode wird mit einem analogen *Pin* und der Erdung verbunden, um so die durch den Photostrom entstehende Spannung zu messen. Alternativ wäre es auch möglich, Strom anzulegen und den größer werdenden Photowiderstand der Diode anzuzeigen.
Als Öffnungs- Schließsensor dienen Magnetschalter. Diese sind jeweils an einer Seite mit einer 5 V Spannungsquelle und an der anderen mit einem digitalen *Pin* des Arduino verbunden und werden am Deckel der Station angebracht. Gleichzeitig sind am Rahmen Magnete angebracht, die jeweils einen der Magnetschalter aktivieren, wenn der Deckel vollständig geöffnet oder geschlossen ist. Wird ein Schalter aktiviert, schließt sich sein Stromkreis und der Arduino registriert eine Spannung am entsprechenden *Pin*. Bisher konnte nicht getestet werden, ob Magnet oder Magnetschalter die Photoelektronen in den PMTs stärker als das Erdmagnetfeld ablenken. Sollte sich dies jedoch als wahr erweisen, so müssen um die Magnetschalter und den Magneten Isolierungen angebracht werden.
Da der Arduino nicht die erforderliche Spannung zur Verfügung stellen kann, um den Motor direkt zu betreiben, wird er stattdessen benutzt um die 12 Volt Spannungsversorgung des Motors durch die Batterie zu steuern. Da sich der Motor in beide Richtungen drehen soll, muss zudem ein System integriert werden, mit dem dies möglich ist. Dazu werden zwei Relais eingesetzt, von denen eines den Stromkreis schließt, während das andere die Richtung des Stromflusses und so die Drehrichtung des Motors kontrolliert.
Auch der Elektromagnet am Deckel wird mittels eines Relais angesteuert.
Um den Arduino innerhalb der Messstation zusätzlich zu schützen, soll er in ein eigenes Gehäuse eingebaut werden. Deshalb wurde zum anschließen aller Hardwarekomponenten eine spezielle Platine entworfen, die die *Pins* des Arduino auf ein Flachbandkabel legt, denn so ist es möglich, einzelne Teile zu wechseln, ohne jedes Mal den Arduino auszubauen. Das Flachbandkabel muss nur, genau wie der USB-Port des Arduino, durch eine kleine Öffnung im Gehäuse nach außen geführt werden. Eine Schaltskizze der Platine ist in Abb. 2.5 zu sehen.

[1] Bezogen auf die Wellenlänge des zu messenden Lichts

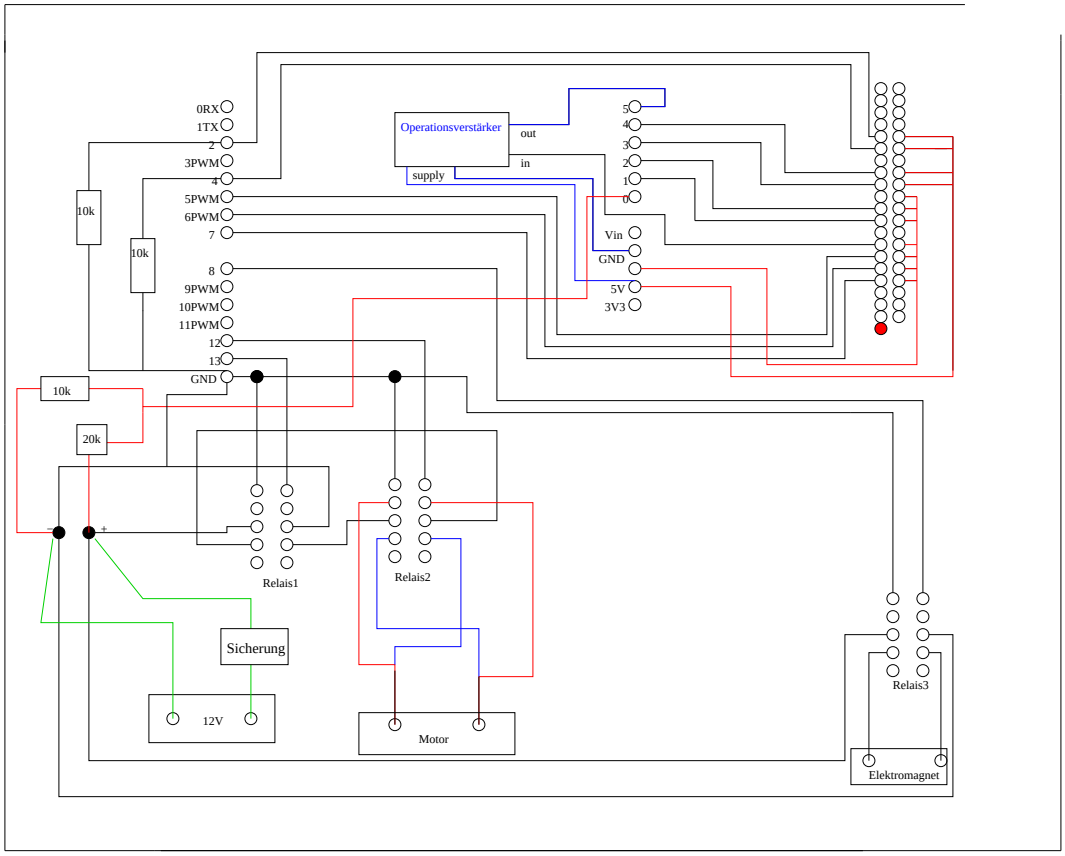

Abbildung 2.5: Skizze der Slow Control Platine

2.3 Slow Control Hauptprogramm

Das Hauptprogramm läuft nach seiner Initialisierung auf dem Mikrocontroller in einer Endlosschleife ca. 12 mal pro Sekunde bis der Strom abgeschaltet, bzw. der Arduino vom Computer (PlugPC) getrennt wird. Als erstes wird die serielle Konsole abgefragt. Wenn eine Eingabe vorhanden ist, wird sie mit einer Liste der vordefinierten Kommandos, z.B. für die in Sektion 2.1 angegebenen Ausgaben oder die Kalibration der Inputs, verglichen. Die Kommandos sind wegen der Eigenarten des Arduino bewusst einfach gehalten und bestehen zumeist nur aus einem einzelnen Buchstaben, dem bei analogen Einstellungen noch die eingegebene Zahl hinzugefügt wird. Zum Vergleich der Kommandos wird eine $switch-case$ Anweisung benutzt (siehe Anhang A.3).
Die Eingabe eines neuen Diskriminatorschwellen und das Ein- und Ausschalten des Magneten hängen von keinem weiteren äußeren Parameter ab und werden vom Programm sofort vorgenommen. Der Magnet bezieht, wie der Motor, seine 12 Volt Betriebsspannung, die vom Arduino mittels eines Relais ein- und ausgeschaltet werden kann, von der Batterie. Befehle jedoch, die die Deckelsteuerung und das HV-System betreffen (Beispiel in Code-Zeile 1 bis 4), werden vom Programm zunächst in Befehlsvariablen gespeichert, um dann weiterbearbeitet zu werden.

```
1   case 'h':                    // input says "HV on"
```

```
2    hvon = 1;
3    hvoff = 0;
4    break;
```

Erhält das Programm die Anfrage zum Senden der Slow Control Daten, so liest es die entsprechenden Input *Pins* aus und schreibt in den internen Speicher einen String mit folgender Zusammensetzung: HV in Volt, Batteriespannung, Strom, Temperatur in ℃, Deckelstatus, HV Status, *canopen* (s.u.), *canpower* (s.u.) und die Feuchtigkeit. Die einzelnen Variablen sind dabei durch das $ Zeichen voneinander getrennt, so dass sie später wieder separiert werden können (Auszug in Zeile 5 bis 9).

```
5   input = analogRead(temperaturePin); // read out temperature
6   intwo = float(input);
7   temp = ((intwo*0.004888))*100;
.....................................
8   Serial.print(temp);
9   Serial.print("$");
```

Nach dem Abfragen der Befehle bearbeitet das Programm die Deckelsteuerung und die HV Kontrolle.

Deckelsteuerung Der Motor, mit dem der Deckel der Messstation geöffnet wird, wird, wie im vorigen Abschnitt erwähnt, vom Arduino mit zwei Relais angesteuert. Zu dem Relais, das den Stromkreis schließen soll, gehört die Variable *lidpower* mit dem Wert 1 (ein) oder 0 (aus). Die Variable *open* bestimmt die Drehrichtung des Motors. Ist $open == 0$), so wird das zweite Relais nicht beschaltet und der Stromkreis des Motors so geschlossen, dass sich der Deckel schließt, bei $open == 1$ wird er geöffnet.
Als erstes wird die Photodiode ausgelesen und der eingegangene Wert in der Variable *light* gespeichert. Anschließend stellt das Programm den Öffnungszustand des Deckels fest (geöffnet, geschlossen oder undefiniert) und speichert diesen in der Variablen *lidstate* ('o', 'c' oder 'n'). Nun vergleicht das Programm die gerade gemessene Lichtintensität mit dem Grenzwert für die Deckelsteuerung. Ist die Intensität niedriger und die Hochspannung eingeschaltet, so stellt das Programm fest, dass der Deckel geöffnet werden darf, die Variable *canopen* wird auf "y"gesetzt, anderenfalls setzt das Program die Variable auf "n". Sollte ein Befehl zum öffnen des Deckels vorliegen ($open == 1$, $lidpower == 1$), obwohl dies nicht erlaubt ist, so wird er vom Programm an dieser Stelle rückgängig gemacht ($open = 0$, $lidpower = 0$). Anschließend wird, wenn die Intensität über dem Grenzwert liegt und der Deckel nicht geschlossen ist ein Befehl zum Schließen erteilt ($lidpower = 1$). Erst nach diesen Abgleichen wird das Programm die Befehle zur Deckelsteuerung ausführen. Vor dem Einschalten des Motors in einer Richtung wird allerdings noch überprüft, ob dieser innerhalb der letzten 500 ms in entgegengesetzter Richtung geschaltet war, um zu verhindern, dass das Material überbeansprucht wird.
Nachstehend eine Zusammenfassung der logischen Bedingungen in der Deckelsteuerung:

Kommando	HV-Status	Licht	Deckelstatus	Aktion
open	Ein	< Schwellenwert	geschlossen	öffne Deckel
open	Ein	> Schwellenwert	geschlossen	nichts
open	Aus	beliebig	geschlossen	nichts
open	Ein	< Schwellenwert	offen	nichts
beliebig	Ein	> Schwellenwert	offen	schließe Deckel
close	Ein	beliebig	offen	schließe Deckel
close	Ein	beliebig	geschlossen	nichts
beliebig	Aus	beliebig	offen	schließe Deckel

HV-System Als nächstes wird der Wert von *light* mit dem Grenzwert für das HV-System verglichen, welcher etwas über dem für die Deckelsteuerung liegt[2]. Auch hier wird, wenn die Lichtintensität den Grenzwert übersteigt, jeder Befehl zum Einschalten der Spannungsversorgung rückgängig gemacht und ggf. ein Kommando zum Abschalten erteilt. Mit letzterem geht auch der Befehl einher, den Sollwert der Spannung auf Null zu setzen. Dann wird vom Programm der digitale Eingang abgefragt, in den das Statussignal des ISEG-Gerätes eingegeben wird. Das Ergebnis dieser Abfrage wird in der bool-Variablen[3] hvHigh gespeichert, wobei *true* bedeutet, dass die HV eingeschaltet ist. Nach diesen Abgleichen beginnt das Programm mit dem Regulieren der Spannungsstärke. Hierzu wird die Variable *hvset*, die den aktuellen Wert der Spannung beinhaltet, mit der Variable *hvwrite* verglichen, die den zu setzenden Wert enthält. Dieser wird bei aus der Eingabe x des Benutzers wie folgt berechnet: $hvwrite = \frac{x*255}{calib}$. Der Kalibrationsfaktor ist hier, genau wie bei der HV-Messung, eine Variable und enthält den Spannungswert, der vom HV-Gerät bei einer Eingabe von 5 Volt ausgegeben wird. Stimmen diese nicht überein, so wird der Wert von *hvset* alle 200 ms um 13 Skalenpunkte (etwa 100 V) erhöht oder gesenkt, je nachdem ob *hvwrite* höher oder niedriger als *hvset* ist. Nachdem etwaige Regulierungen vorgenommen wurden wird der aktuelle Wert von *hvset* vom Arduino als PWM-Signal an das HV-Gerät ausgegeben.
Die logischen Zusammenhänge der HV-Kontrolle seien hier noch einmal zusammengefasst:

Input	HV Status	Gemessener Lichtwert	Aktion
hv on	Aus	< Schwellwert	schalte HV ein
hv on	Aus	> Schwellwert	schalte HV nicht ein
hv on	Ein	< Schwellwert	nichts
beliebig	Ein	> Schwellwert	deaktiviere HV
hv off	Ein	beliebig	deaktiviere HV
hv off	Aus	beliebig	nichts

Zur Veranschaulichung des Programmablaufs siehe Abb. 2.6

[2]So ist es möglich die Messinstrumente vor Beginn des Zeitfensters für die Datennahme hochgefahren werden, damit keine Messzeit verlorengeht

[3]Ein bool'scher Ausdruck kann nur die Werte *true*, also 'wahr', oder *false*, also 'unwahr' annehmen

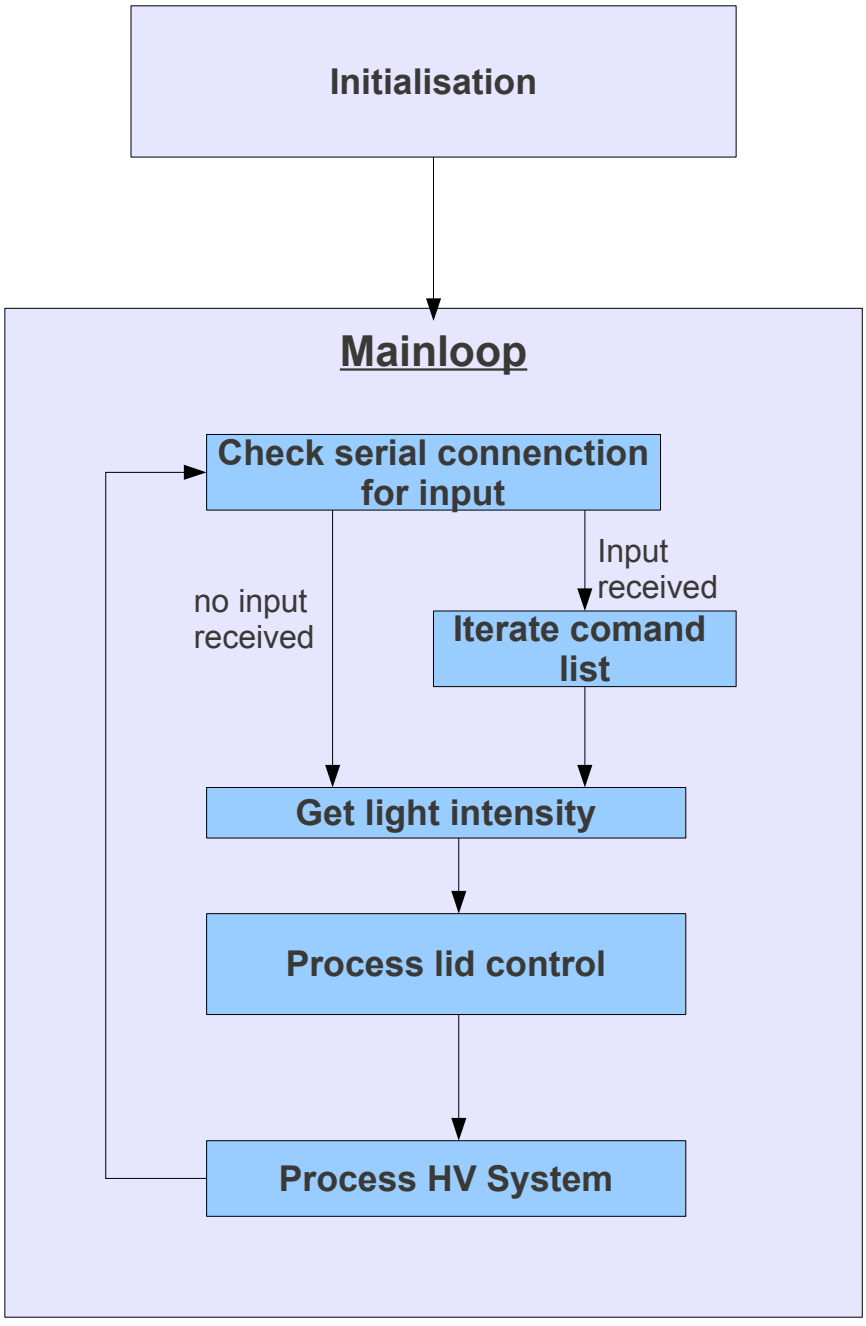

Abbildung 2.6: Zeitlicher Ablauf des Slow Control Hauptprogramms

2.4 Testmessungen

2.4.1 Temperaturmessungen

Die ersten Temperaturmessungen wurden in einem Raum an der Nordseite von DESY-Gebäude 68 durchgeführt. Der Aufbau stand hierbei in unmittelbarer Nähe der Fenster.

Nachdem erwiesen war, dass der Messaufbau wunschgemäß funktionierte, wurden einige Langzeitmessungen gestartet. Eine dreitägige Messung der Innentemperatur im Büro sollte zeigen, ob das Messprogramm und die Elektronik über einen längeren Zeitraum stabil bleiben können.

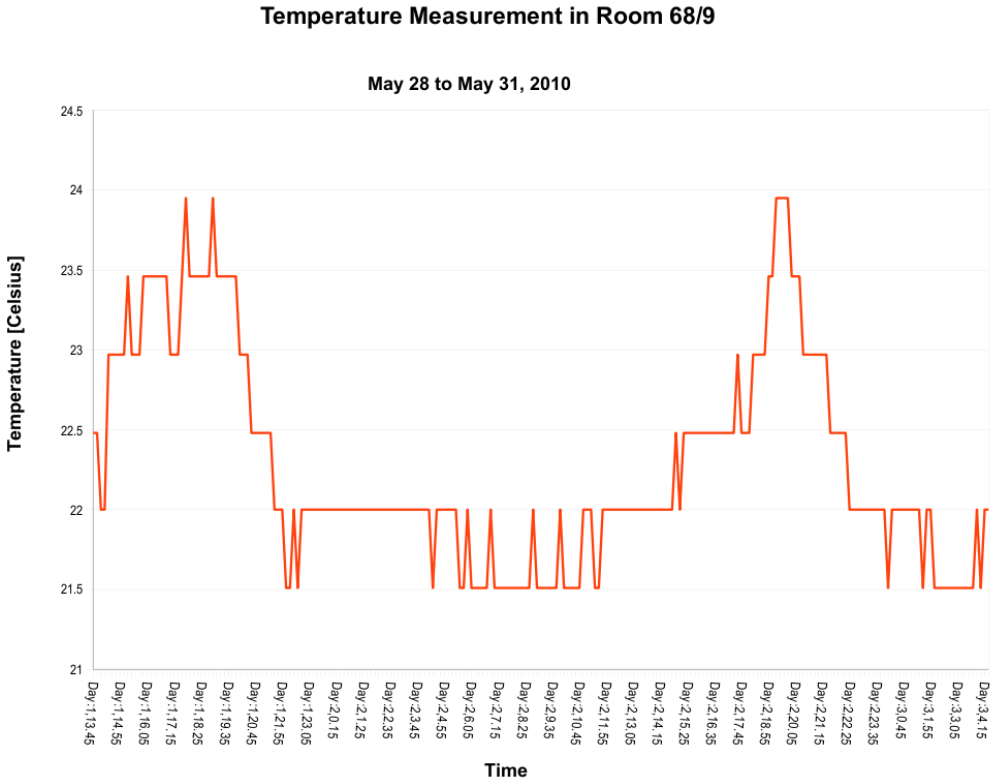

Abbildung 2.7: Verlauf der dreitägigen Messung

Wie man in Abb. 2.7 sieht lief die Messung stabil und zuverlässig, die einzigen Schwankungen, die auftreten, sind normale, tageszeitabhängige Temperaturänderungen. Als echter Test sollte nun eine Messung der Außentemperatur dienen. Da der Aufbau gegenüber den Umwelteinflüssen zu anfällig wäre, bestand die Veränderung im Vergleich zum vorherigen Versuch lediglich darin, den Temperaturfühler mittels einer Verlängerung so nahe wie möglich an ein geöffnetes Fenster zu bringen. Diese Anordnung wurde zunächst über nacht stehen gelassen.

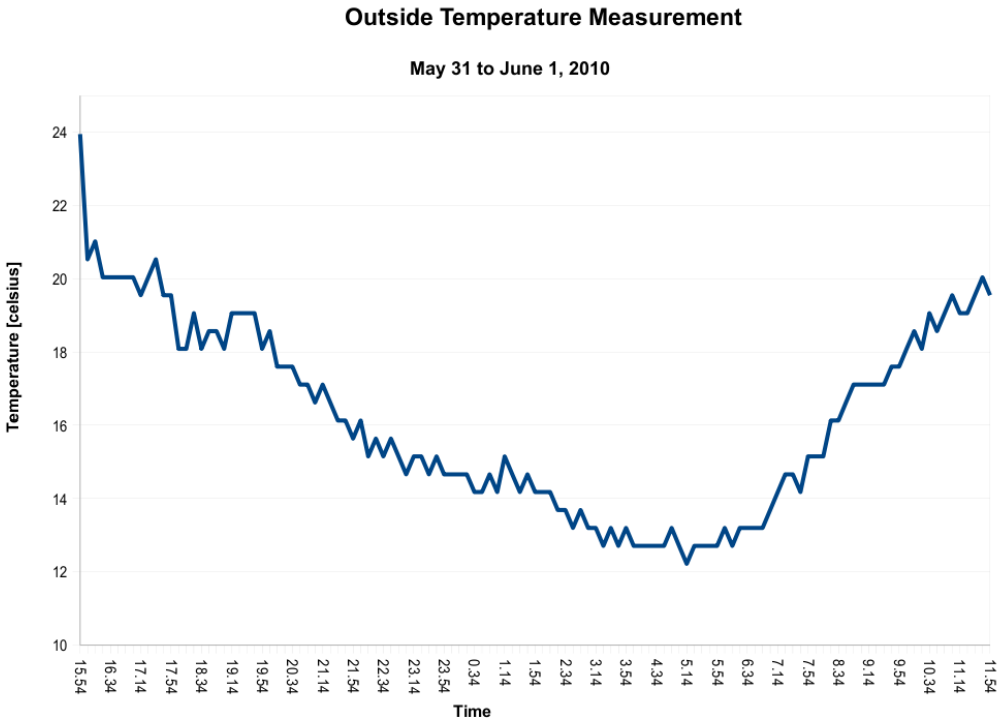

Abbildung 2.8: Ergebnis der ersten Nachtmessung

In Abb. 2.8 zeigt sich eine größere Differenz zwischen den Extremwerten als bei der ersten Messung. Auch bei diesem Versuch ist die Elektronik stabil geblieben und das Programm arbeitete zuverlässig.
Sowohl in Abb. 2.7 als auch in Abb. 2.8 liegen die Maximaltemperaturen am späten Nachmittag und frühen Abend. Dies liegt am wahrscheinlichsten in der Ausrichtung des Aufbaus nach Norden begründet liegen. Hierdurch war der Thermowiderstand dem zu diesen Tageszeiten seitlich von Westen enfallenden Sonnenlicht ausgesetzt.

Da nicht jede Messstation per Kabel mit einem separaten Computer verbunden wird, müssen die Daten drahtlos an die Zentralstation übertragen werden. Bisher wurden die Daten einfach nach ihrer Bearbeitung über die serielle Schnittstelle ausgegeben, während es bei einer drahtlosen Übertragung allerdings nötig ist, die vom messenden Modul gesendeten Daten durch den Empfänger korrekt zu interpretieren und wiederzugeben. Das Messprogramm kann hierbei nahezu identisch mit dem Desktop-Versuch sein, da über die drahtlose Verbindung serielle Ausgaben gesendet werden. Die Schwierigkeit liegt hier im Leseprogramm, da der Ardunino nicht in der Lage ist, Eingaben, die länger als ein Byte sind, zu lesen. Die drahtlose Verbindung zwischen Messaufbau und Zentralstation wurde mittels zweier Xbee-Module hergestellt
Bei einer weiteren Außentemperaturmessung wurde das Arduino-Board mit einer 9 Volt-Batterie betrieben, um eine größere Mobilität und Flexibilität bei der Wahl des Messortes zu erhalten. Zusätzlich zur Zuverlässigkeit der Messapparate war es hier auch wichtig zu ermitteln, wie lange der Arduino nur mit einer herkömmlichen Batterie betrieben werden kann. Der Messaufbau wurde an einer nach Süden und Osten hin licht- und windgeschützten Stelle angebracht.

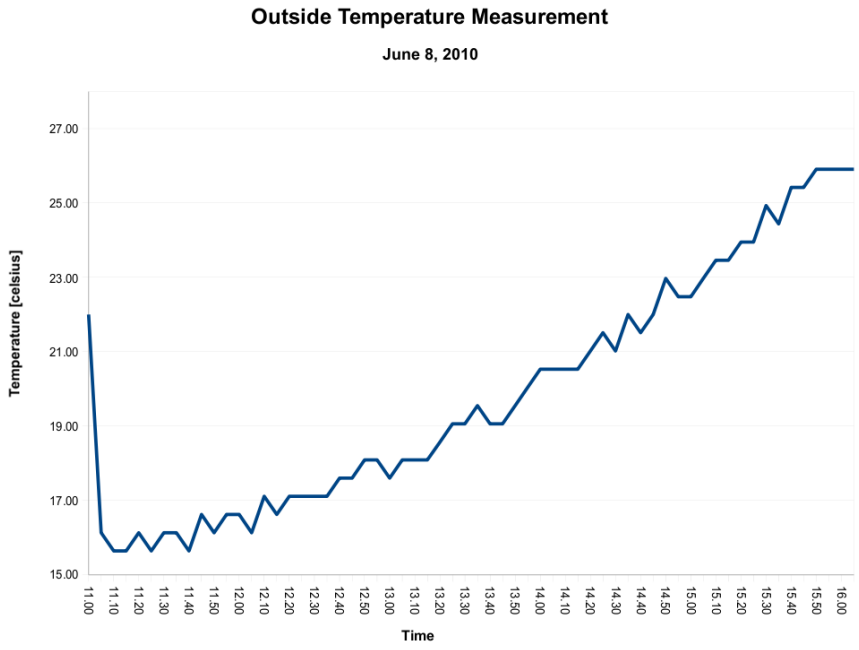

Abbildung 2.9: Vollständige Darstellung der batteriebetriebenen Messung

Die Temperaturmessung war erfolgreich, wie in Abb. 2.9 zu sehen ist. Allerdings sieht man, dass eine handelsübliche Alkaline 9 Volt Batterie mit einer Ladung von ca. 550 mAh den Aufbau aus Temperaturfühler, Arduino und XBee-Modul nur für etwa 6 Stunden mit Strom versorgen kann. Dies entspricht einem Strom von etwa 100 mA.

2.4.2 Lichtmessungen

Die Testmessung ging über einen Zeitraum von ca. zwei Tagen und sollte die generellen Funktionsparameter der Photodiode aufzeigen. Auch hier wurde der Aufbau unmittelbar an einem nach Norden blickenden Fenster aufgestellt. Da vom Hersteller keine Angaben zum genauen Verhältnis von Lichtintensität zur Photospannung vorliegen, werden die Messergebnisse hier in Skalenelementen (*Bins*) des Arduino angegeben.

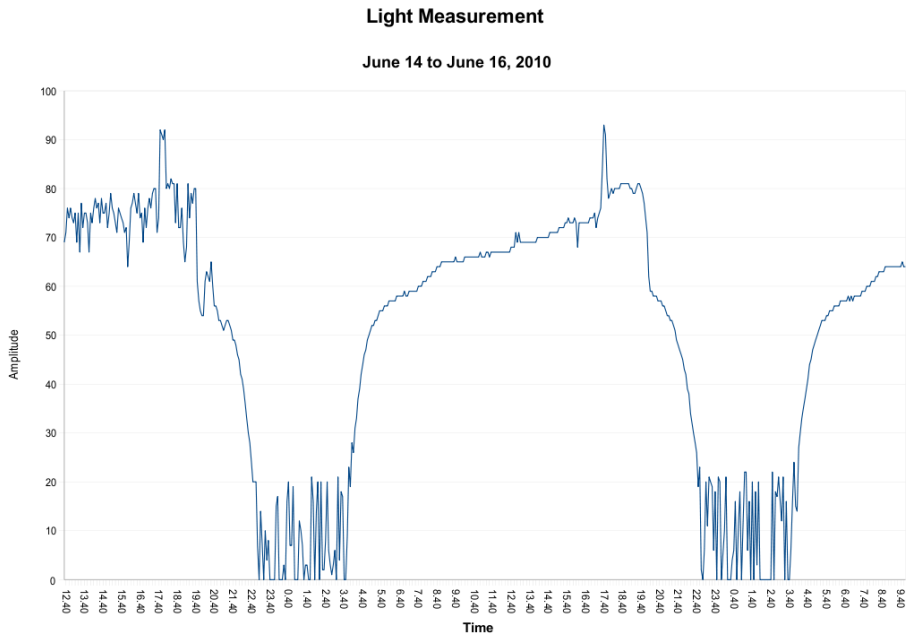

Abbildung 2.10: Testmessung mit Photodiode

Man kann in Abb. 2.10 sehen, dass die Lichtstärke zu Tageszeiten einigermaßen gleich verläuft, wobei alle Messtage auch ähnliche Wetterbedingungen hatten. Zu Nachtzeiten allerdings schwanken die Messwerte sehr stark.
Um zu ermitteln, ob dieses Rauschen auf schwankende Lichtverhältnisse, hauptsächlich durch künstliche Lichtquellen, verursacht wurde, wurde eine weitere Testmessung gestartet. Hierfür wurde die Photodiode gegen äußere Lichtquellen abgeschirmt und die Messung fand in einem komplett abgedunkelten Raum statt.
In Abb. 2.11 sind die Ergebnisse dieser Messung im zeitlichen Ablauf und als Histogramm dargestellt, in dem die Häufigkeit des Auftretens jedes Skalenwertes bei einer Gesamtzahl von 1000 genommenen Messwerten gezeigt wird.

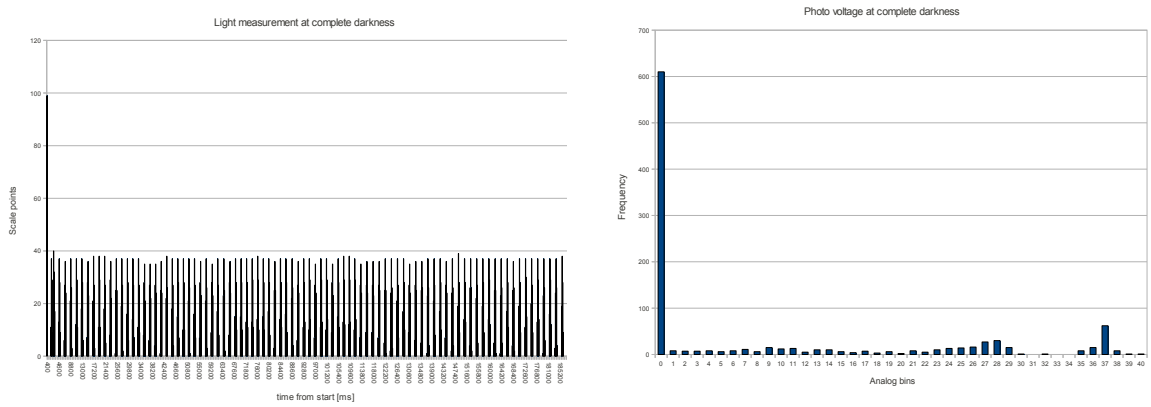

Abbildung 2.11: Ergebnisse zur Messung des Rauschens der Photodiode

Die Messung zeigt, dass nur etwa 60 % der genommenen Messpunkte einen Wert von 0 haben und Photospannungen von bis zu 200 mV (entspricht etwa 40 Skalenpunkten) auftreten, auch zeitliche Form des Rauschens ist mit der vorherigen Messung vergleichbar. Damit sind künstliche Lichtquellen als Quelle des Rauschens weitgehend ausgeschlossen. Es ist zu vermuten, dass diese Schwankungen entweder durch den Arduino selbst verursacht werden, oder durch thermische Prozesse in der Photodiode entstehen.

2.4.3 Latenzzeitmessungen an den XBee-Modulen

Um zu testen, ob die XBee-Module auch für zeitkritische Zwecke wie den Einsatz in der zweiten Triggerstufe geeignet sind, mussten deren Kommunikationsgeschwindigkeit und Zuverlässigkeit geprüft werden. Hierfür wurde ein Aufbau aus zwei mit XBee-Modulen bestückten Arduinos benutzt. Der erste, mit einem Computer verbunde Arduino schickte serielle Eingaben verschiedener Länge an den anderen, der nach Empfang eine Antwort zurückschickte. Der erste Arduino gab dann die Zeit, die zwischen dem Beginn der Übertragung und dem Empfang der Antwort vergangen war an den Computer aus und schickte nach Ablauf eine vordefinierten Zeitspanne (*Delay*) die nächste Eingabe. Es wurden hierbei Delays von 250, 500, 750, 1000, 1500 und 2000 ms getestet, wobei jeweils 120 Messwerte genommen wurden.
Diese Messung wurde für Eingaben mit einer Länge von 1, 5, 10 und 15 Byte getestet wobei

sowohl das Schreiben der Eingabe als ganzer *String* als auch das Senden jedes Bytes als einzelne Zeile beobachtet wurden. Die Übertragungsdauern wurden für Netzwerkgeschwindigkeiten von 9600 und 19200 Baud (Übertragungsschritte/Sekunde) gemessen.
Neben der Feststellung der durchschnittlichen Kommunikationsdauer wurde auch die Anzahl der gestörten Übertragungen, bei denen die Übertragungsdauer mehrere Sekunden betrugen, bestimmt[4]. In Abb.2.12 ist die mittlere Kommunikationsdauer als Funktion der Eingabelänge, gemittelt über alle Delays, gezeigt.
Erwartungsgemäß sind die Übertragungen bei 19200 Baud schneller als ben 9600. Dass die

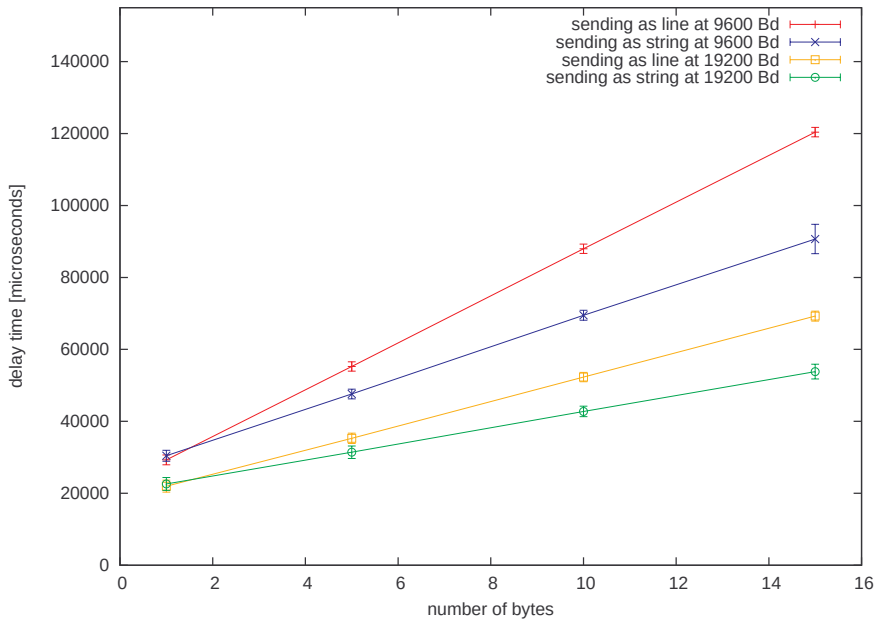

Abbildung 2.12: Darstellung der Durchschnittlichen Kommunikationsdauer

Übertragung in Form ganzer Strings schneller ist, ist darauf zurückzuführen, dass beim Senden in einzelnen Zeilen nach jedem Byte zusätzlich ein Kommando zum Zeilenumbruch geschickt wird. Durch dieses zusätzliche Zeichen wird die zu sendende Datenmenge stark vergrößert.
Der Anteil der gestörten Übertragungen, abhängig vom Delay, ist in Abb. 2.13, 2.14, 2.15 und 2.16 zu sehen, wobei über alle Eingabelängen aufsummiert wurde.

[4]für die Bestimmung der Durchschnittszeit wurden fehlerhafte Übertragungen nicht berücksichtigt

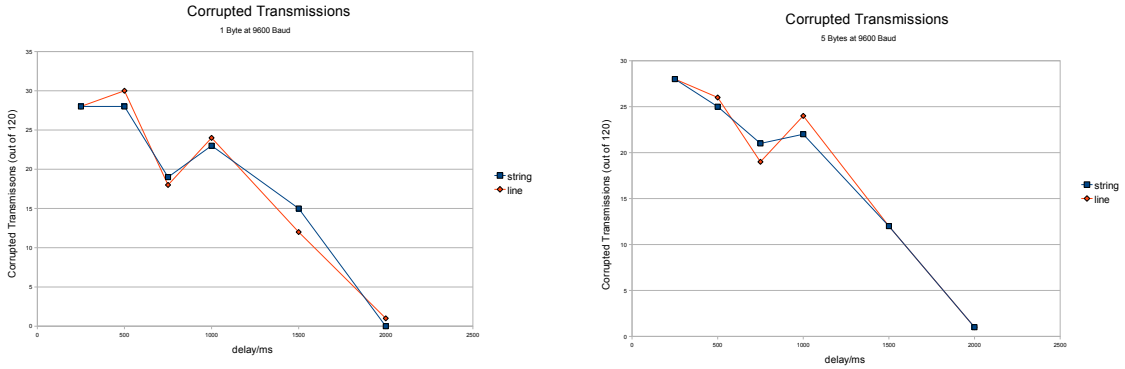

Abbildung 2.13: Anzahl der fehlerhaften Übertragungen für 1 Byte (links) und 5 Bytes (rechts) bei 9600 Baud

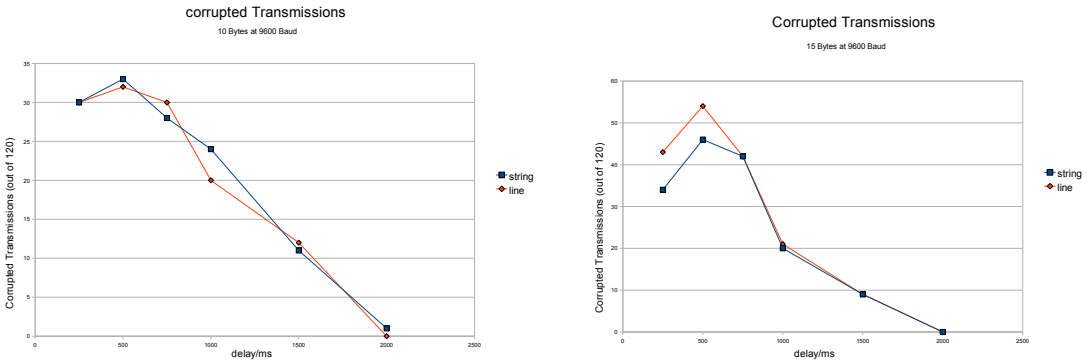

Abbildung 2.14: Anzahl der fehlerhaften Übertragungen für 10 Bytes (links) und 15 Bytes (rechts) bei 9600 Baud

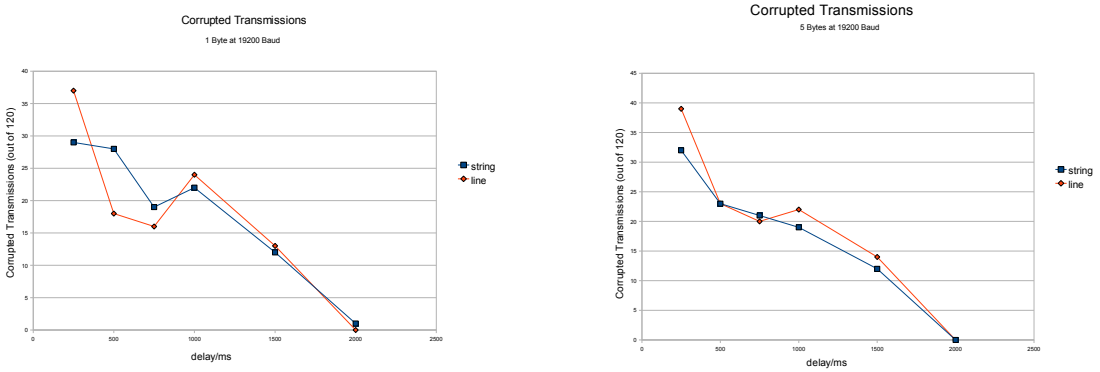

Abbildung 2.15: Anzahl der fehlerhaften Übertragungen für 1 Byte (links) und 5 Bytes (rechts) bei 19200 Baud

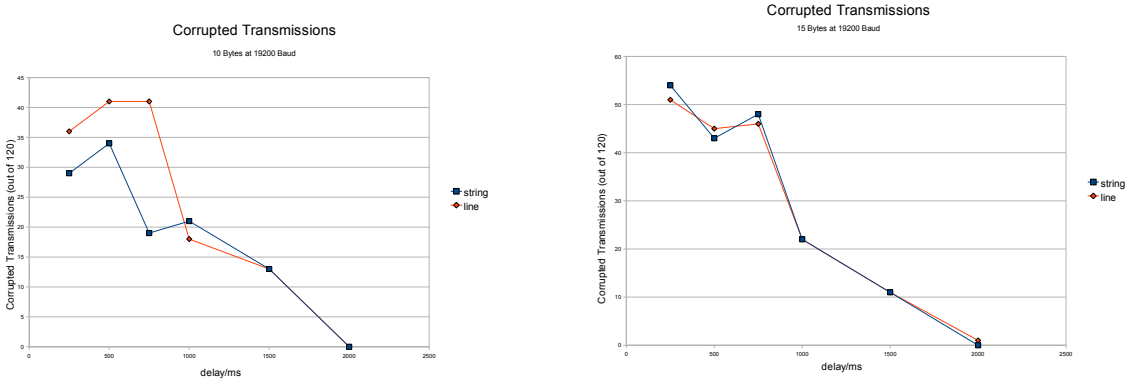

Abbildung 2.16: Anzahl der fehlerhaften Übertragungen für 10 Bytes (links) und 15 Bytes (rechts) bei 19200 Baud)

Erst ab einer Verzögerung von zwei Sekunden zwischen zwei Übertragungen funktioniert die Kommunikation fehlerfrei. Dies und die hohe Übertragungsdauer bei großen Eingaben machen das XBee-Modul für zeitkritische Operationen ungeeignet. Für weitere Graphiken und Ergebnisse siehe Anhang A und [18].

2.4.4 Weitere grundlegende Funktionstests

Neben den bereits genannten Testmessungen musste auch geklärt werden, ob der Arduino in der Lage ist, die Ausgaben für die Slow Control korrekt vornehmen kann.
Um zu ermitteln, ob der Arduino in der Lage ist, ein Relais zu schalten, wurde ein Relais mit einem seiner beiden Schaltkontakte an einen digitalen *Pin* und mit dem anderen an den *GND-Pin* des Arduino angeschlossen. Danach wurde die Leitfähigkeit auf beiden Möglichen Schaltrichtungen des Relais bei ein- und bei ausgeschalteter Spannung gemessen, um festzustellen, ob sie sich ordnungsgemäß öffnen und schließen lassen. Dieser Versuch war erfolgreich, es ist also möglich, mit dem Arduino eine Relaisschaltung aufzubauen.
Die analoge Ausgabe des Arduino wurde mit Hilfe einer Leuchtdiode getestet, die an einen Pulsweitenmodulations-*Pin* angeschlossen wurde. Es wurden zwei Programme für diesen Test benutzt, eines, das die Ausgabespannung kontinuierlich steigert und beim Maximum wieder auf 0 zurückspringt und ein zweites, das einen Eingabewert aus der seriellen Konsole liest und die Ausgabespannung entsprechend ändert. In beiden Fällen sollte die Helligkeit der Leuchtdiode (relativ zum Maximum) als Anhaltspunkt fur die wunschgemäße Funktion des Aufbaus dienen. Diese schien (subjektiv) gewährleistet zu sein, genauere Ergebnisse wird erst ein Test mit dem ISEG-Gerät liefern.

Kapitel 3

Kontrolle der Datennahme

Im Rahmen dieser Diplomarbeit soll die Datennahme von HiSCORE mit einem DRS4-Board, das an einen PlugPC angeschlossen ist, implementiert werden. Es müssen Möglichkeiten für eine externe Kontrolle der Datennahme eingebaut werden und das Speichersystem des Datennahmeprogramms muss optimiert werden, damit Daten mit einer möglichst geringen Totzeit gespeichert werden können.
Weiterhin müssen alle hierfür am bereits bestehenden Datennahmeprogramm vorgenommenen Änderungen getestet werden.

3.1 Verwendete Komponenten

3.1.1 PlugPC

Abbildung 3.1: Der GuruPlug

Bei einem PlugPC handelt es sich um eine Art Miniatur-Computer. Er ist in der Lage, dieselben Rechenoperationen und Programme auszuführen wie ein normaler Computer. Im

Rahmen dieser Diplomarbeit kam der GuruPlug Server Plus von Globalscale Technologies zum Einsatz. Dieser hat eine Prozesorgeschwindikeit von 1,2 GHZ, 512 GB internen Flash-Speicher, sowie 512 MB RAM. In der Standardeinstellung öffnet der Rechner einen W-LAN Access-Point. Das vorinstallierte Betriebssystem ist Debian, eine Unix-Distribution mit Linux-Kernel. PlugPcs werden meistens in Netzwerken eingesetzt, z.B. als File- oder Printserver, da sie diese Aufgaben genauso ausführen können, wie ein Desktoprechner, bei Kosten von 100 bis 160 €.
Der User kann sich über ein Netzwerk mit dem Computer in Verbindung setzen, indem er ein ssh-Login als root durchführt. Alternativ ist es möglich, das so genannte JTAG-Modul zu benutzen, eine zusätzliche Hardware-Komponente, die man als serielle Schnittstelle zum PlugPC benutzen kann. So kann man mit einem seriellen Terminal-Programm ebenfalls ein Login auf dem PlugPC durchführen, zudem ist die serielle Verbindung der einzige Weg, bei Bedarf eine neue Firmware auf den GuruPlug zu spielen.

3.1.2 Das DRS4 Board

Abbildung 3.2: Das DRS4-Board (Bild: PSI [2])

Das DRS4 (***D**omino **S**crambling* **C**hip) Board wird vom Paul Scherrer Institut (PSI) hergestellt und dient der Digitalisierung analoger Spannungssignale. Der auf dem Board befindliche DRS Chip ist in der Lage, Signale aus 8 analogen Kanälen mit einer maximalen Geschwindigkeit von 6 Milliarden Signalen pro Sekunde zu verarbeiten.

Installation Am Beginn der Arbeit an der Datenauslese stand zunächst das Konfigurieren der Hardware. Da auf den GuruPlugs wie bereits erwähnt keinerlei Anwendersoftware vorhanden ist, war es also nötig, folgende Installationen vorzunehmen:

- C++ Compiler
- make (Standardprogramm zur Programminstallationnstallation unter Linux)
- libusb (notwendig zum Betrieb der USB-Anschlüsse)

Zum Auslesen des DRS4-Chips wird vom Hersteller ein Programmpaket bereitgestellt, das neben den Auswertungsprogrammen auch die verschiedenen speziellen Bibliotheksdateien und Klassendefinitionen enthält, die zur Auswertung benötigt werden [2].
Das Board verfügt über eine *onboard-Trigger*, der es dem Board erlaubt, auch ohne extern eingespeistes Triggersignal eine Ereignisauswahl zu treffen.

drs_exam Um die Auslese des DRS4-Boards auf dem GuruPlug betreiben zu können, musste allerdings das Makefile des Programmpakets modifiziert werden (siehe Anhang B), da der PlugPC nicht in der Lage ist, die standardmäßig enthaltenen graphischen Komponenten zu verarbeiten. Nach der Entfernung der entsprechenden Elemente blieb nur noch das Ausleseprogramm drs_exam übrig. Dieses sucht nach dem Aufruf standardmäßig die USB-Ports des Computers nach DRS4-Boards ab und beginnt, sofern eines vorhanden ist, sofort mit der Datennahme. Das Programm wartet auf ein vorher definiertes Triggersignal[1] (Definition: Zeile 10 bis 13, Abfrage des Signals: Zeile 16) und liest, wenn ein solches eintrifft die vom Chip auf den einzelnen Kanälen separat übertragenen Daten aus, speichert sie für jeden in einem eigenen Array, dem *wavearray*, in mV ab und generiert zusätzlich noch ein Array, das den zu den Spannungswerten gehörenden Zeitindex enthält (Zeile 18 bis 20). Zu einem Ereignis gehören hierbei 1024 solcher Gruppen aus Zeitindex und Spannungspulsen. Der Zeitindex ist im Chip fest vorgegeben, Der Abstand zwischen zwei genommenen Spannungshöhen beträgt ca. 0,191 ns, jedes Ereignis hat also dieselbe Länge.

```
10      b->EnableTrigger(0, 1);            // lemo off, analog trigger on
11      b->SetTriggerSource(0);            // use CH1 as source
12      b->SetTriggerLevel(-0.05, true);   // 0.05 V, negative edge
13      b->SetTriggerDelay(0);             // zero trigger delay

     ....................................................
14      /* wait for trigger */
15      printf("Waiting for trigger...");
16      while (b->IsBusy());

       /* read all waveforms */
17      b->TransferWaves(0, 8);

      /* read time (X) array in ns */
18      b->GetTime(0, time_array);

      /* decode waveform (Y) array first channel in mV */
19      b->GetWave(0, 0, wave_array[0]);

      /* decode waveform (Y) array second channel in mV
         Note: On the evaluation board input #1 is connected to channel
         0 and 1 of the DRS chip, input #2 is connected to channel 2 and
         3 and so on. So to get the input #2 we have to read DRS channel
         #2, not #1 */
```

[1] bei diesem Trigger handelt es sich um die erste Triggerstufe

```
20      b->GetWave(0, 2, wave_array[1]);
```

Danach werden die Daten in einen String umgewandelt und in eine Textdatei geschrieben (Zeile 21 bis 24).

```
21  f = fopen("data.txt", "w");
       ..........
 /* Save waveform: X=time_array[i], Yn=wave_array[n][i] */
22  fprintf(f, "Event #%d: t y1 y2\n", j);
23  for (i=0 ; i<1024 ; i++)
24     fprintf(f, "%1.2f %1.2f %1.2f\n", time_array[i], wave_array[0][i],
               wave_array[1][i]);
```

Die Spannungswerte werden hierbei als Fließkommazahl ($\%f$) mit einer Genauigkeit von zwei Nachkommastellen gespeichert.
Das Programm ist standardmäßig auf einen Testmodus eingestellt: nachdem das Board initialisiert wurde beginnt es mit einer for-Schleife, die von 0 bis 9 läuft. Innerhalb der for-Schleife wird nun zunächst das Aufnehmen eines Ereignisses vorbereitet, indem der Chip bereinigt und das Board gestartet wird. Hat das Programm dann ein Triggersignal erhalten, wird das Ereignis wie vorangehend beschrieben aufgenommen. Dieser Vorgang geschieht nun einmal für jeden Index, es werden also genau zehn Ereignisse aufgenommen.

3.2 Anpassung des Datennahmeprogramms für die HiSCORE-Station

Für die Anwendung bei HiSCORE muss für das Programm ein *Datennahme-Modus* erstellt werden, der es dem Benutzer erlaubt, jederzeit eine beliebig lange Datennahme zu starten, unterbrechen oder beenden. Das Programm wird auf dem PlugPC in jeder einzelnen Station installiert und soll für jede Station separat kontrollierbar sein.
Die genannte Anpassung ist unkompliziert vornehmbar. Zuerst wird statt der von 0 bis 9 laufende for-Schleife eine while-Schleife eingeführt, deren Bedingung immer erfüllt ist, also z.B. while(1). Auf diese Weise läuft das Programm bis es manuell unterbrochen oder der Computer abgeschaltet wird. Diese wird nachfolgend als *innere while-Schleife* bezeichnet.
Anschließend wird das Speichersystem angepasst. Im Testmodus hat das Programm alle aufgenommenen Ereignisse in eine einzige Datei geschrieben, deren Größe durch die vorbestimmte Anzahl an Events festgelegt ist. Dies ist hier nicht der Fall, da die Datennahme über einen unbestimmt langen Zeitraum erfolgen soll. Somit hat auch die Datei eine unbestimmte Größe, was sowohl bei der Speicherung der Daten als auch bei deren Auswertung (aufgrund mangelnder Übersicht) zu Schwierigkeiten führen kann. Außerdem wäre es unmöglich, die Eventdaten während der Laufzeit des Programms zu lesen, da auf eine Datei, in der aktuell geschrieben wird, nicht von außen zugegriffen werden kann. Auch wäre bei einem unvorhergesehenen Fehler, wie etwa einem Systemabsturz, die komplette Datei unbrauchbar. Also wird stattdessen eine Reihe von Speicherdateien mit einem fortlaufenden Index und einer fest

definierten Größe, z.B. 1000 Events[2], eingeführt.

```
25            i = j % 1000; //j == number of events written
            /* if the last data file has been closed after 1000
               events were written or the program has just been
               started, open a new one */
26            if (i == 0){
27               sprintf(filname,"/root/meas_data/data_%d.bin", filcount);
28           f.open(filname, ios::out | ios::binary);
29            }
                .............
            /* if 1000 events have been written to the current
               data file, close it */
30            if (i == 0 && f.is_open()){
31               f.close();
32               filcount = filcount + 1;
33            }
```

Nach dem Schreiben eines Ereignisses prüft das Programm, ob die Gesamtzahl der Ereignisse ganzzahlig durch 1000 teilbar ist (Zeile 30 bis 33) und schließt, wenn dem so ist, die aktuelle Ereignisdatei und erhöht den Dateiindex. Zu Beginn eines neuen Ereigniszyklus wird diese Prüfung noch einmal durchgeführt und, wenn nötig, eine neue Ereignisdatei.

Mit den zuletzt beschriebenen Änderungen ist das Programm nunmehr in der Lage, zeitlich unbeschränkt Daten zu sammeln, solange genügend Speicher zur Verfügung steht. Als nächstes muss nun eine Möglichkeit zur Kommunikation des Programms mit anderen Prozessen geschaffen werden, denn die Datennahme wird nicht direkt per Benutzereingabe kontrolliert, sondern läuft ferngesteuert. Da die anderen Prozesse ebenfalls auf dem PlugPC laufen, funktioniert die Kommunikation einfach über ASCII Dateien, die im Speicher des GuruPlug unter fest definierten Dateipfaden abgelegt werden. In der Datei */root/commands.txt* werden vom Benutzer Befehle für das DAQ Programm geschrieben.

Erste Zeile	Zweite Zeile	Bedeutung
command	start	Beginne Datennahme
command	stop	Unterbreche Datennahme
command	quit	Beende das Programm
enter	<float>	neuer Threshold

Diese Datei muss während der Laufzeit des Programms regelmäßig abgefragt werden (*Polling*). Eine Abfrage während jedes Event-Zyklus wäre bei hohen Frequenzen ein viel zu großer Zeitaufwand und würde somit das Programm empfindlich verlangsamen. Daher wird die Datei statt dessen in regelmäßigen Zeitabständen abgefragt. Ein Wert von zwei Sekunden ist hier vollkommen ausreichend, da so weder eine übermäßig lange Reaktionszeit auftritt, noch wird das Programm durch das Öffnen und ggf. Lesen der Datei zu stark verlangsamt.

[2] Ereignis j befindet sich hierbei immer in der Datei j/1000 bzw. j/Dateigröße

Hierzu wird in der innere while(1) Schleife eine Zeitabfrage integriert, die, ist das Limit von zwei Sekunden überschritten, ein *break* Kommando aktiviert. Nach dem Ende der while-Schleife wird dann der Lesebefehl für die Kommandodatei ausgeführt. Damit das Programm allerdings nicht terminiert nachdem es die Datei gelesen hat, muss eine weitere, übergeordnete Endlosschleife eingebaut werden. Diese zweite (äußere) while-Schleife fragt die Variable a ab, die mit dem Wert 1 initialisiert wird. Solange a == 1 gilt, läuft das Programm weiter, nimmt also zuerst zwei Sekunden lang Daten und sucht dann nach Benutzerkommandos. Erst wenn es in der Kommandodatei den Befehl "quit"vorfindet, wird a auf 0 gesetzt und das Programm terminiert, nachdem es alle anhängenden Prozesse beendet hat (siehe auch Flussdiagramm in Abb. 3.3).
Da das Programm letztendlich aber nur auf einen expliziten Befehl hin mit der Datennahme beginnen soll, muss noch eine weitere übergeordnete Bedingung eingesetzt werden. Zu diesem Zweck wird in dem Prozess die bool-Variable *active* eingeführt. *active* wird mit dem Wert *false* initiiert und jedes Mal direkt nach Beginn der äußeren while-Schleife abgefragt. Ist der Wert von *active* gleich *true*, so führt das Programm die innere while-Schleife aus. Anderenfalls wartet es einfach zwei Sekunden ab, bevor es die Kommandodatei erneut liest.

Da die Funktion des Programms von außen überwacht werden soll, muss noch ein Weg implementiert werden, die Telemetrie der Datennahme für den Benutzer und auch für andere Prozesse sichtbar zu machen. Da bei einer Fernsteuerung des Programms eine Anzeige im Terminal nur bedingt sinnvoll wäre, wird auch hier auf die Speicherung der Daten als Textdatei zurückgegriffen. Die Fragen, die das Programm dem Benutzer beantworten soll, sind:

1. Wurde beim Starten des Programms ein DRS-Board gefunden?
2. Findet aktuell eine Datennahme statt?
3. Wie viele Ereignisse werden pro Sekunde verarbeitet?

Um diese Daten zugänglich zu machen wurden (in derselben Reihenfolge) folgende Ausgaben eingeführt:

1. Wenn nach dem Aufruf ein Board gefunden wurde, wird die Datei */root/found_board.txt* generiert. Diese Datei wird beim Beenden des Programms wieder gelöscht, um eine Fehlinformation zu verhindern.
2. Erhält das Programm den Befehl, die Datennahme zu starten, so wird die Datei */root/active.txt* erstellt. Bei einem Stop-Befehl wird diese Datei augenblicklich gelöscht.
3. Die Ereignisrate wird alle zwei Sekunden, direkt vor dem Lesen der Befehlsdatei, vom Programm berechnet und in die Datei */root/meas_data/evcount.txt* geschrieben.

In Abb. 3.3 ist der Programmfluss vereinfacht dargestellt.

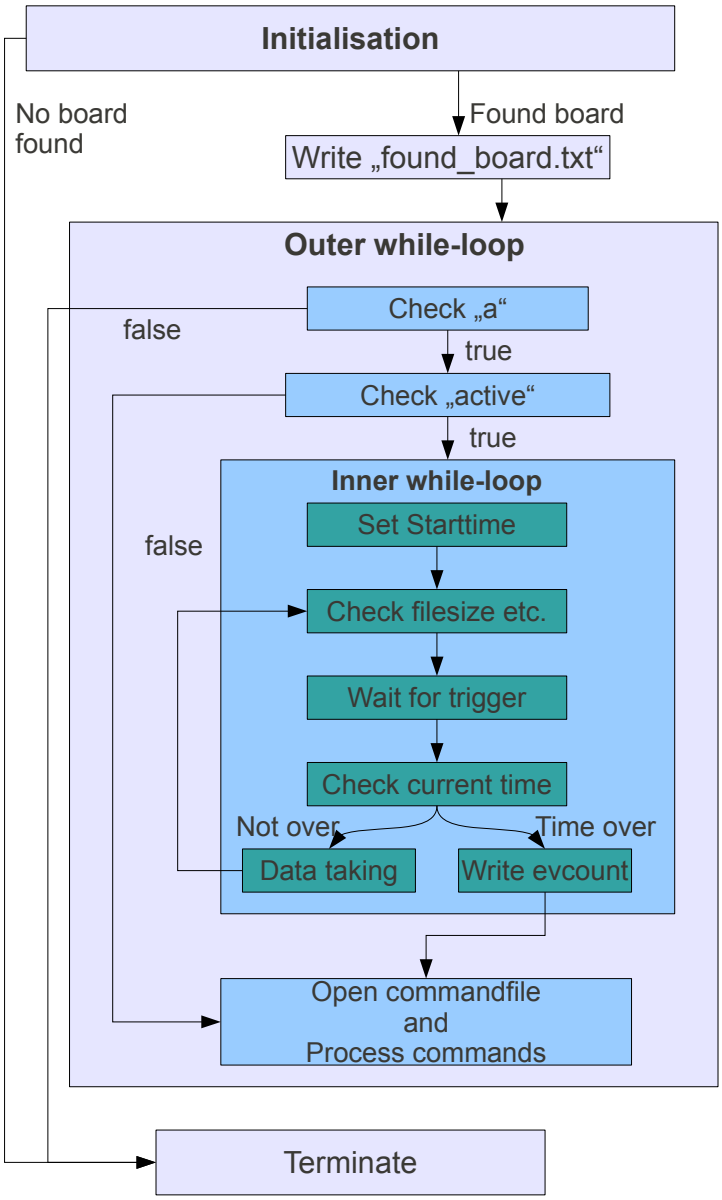

Abbildung 3.3: Ablaufdiagramm des DAQ-Programms

3.3 Tests

Der nächste Schritt ist ein praktischer Test des Programms. Das DRS4-Board wurde an einen Pulsgenerator angeschlossen, welcher Spannungspulse mit einer vorgegebenen Höhe, Breite und Frequenz erzeugte. Hierbei sollte bei bekannter Pulsfrequenz die Ereignisrate des Programms beobachtet werden, um festzustellen, bis zu welcher Pulsfrequenz das Programm alle eingehenden Signale aufzeichnet. Wie in Sektion 1.2.1 dargelegt ist der Signaleingang vom NSB dominiert, während die eigentliche Ereignisrate geringer ist. Daher muss ein möglichst

hoher Anteil der eingehenden Signale aufgezeichnet werden.

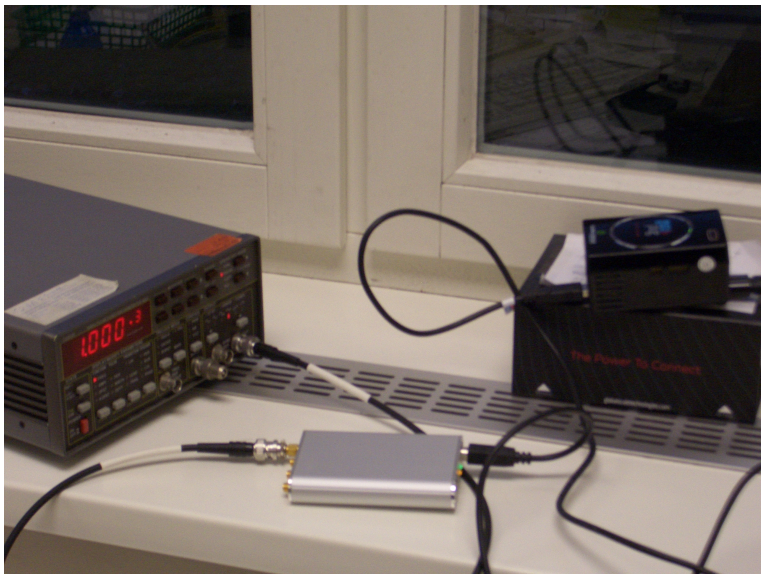

Abbildung 3.4: Der Aufbau zur Messung der Ereignisrate

Mit einem Programm, das eingehende Triggersignale zählt, soll die maximal mögliche Datenrate des Programms auf dem PlugPC bestimmt werden. Nach Ablauf einer bestimmten Zeit wird der Zählerwert zu Beginn des Messzeitraums vom aktuellen Wert subtrahiert und durch die Zeitspanne dividiert, um die Frequenz zu erhalten, welche dann ausgegeben wird. Es werden weder Wellenformen abgefragt, noch Daten gespeichert. Es wurden Rechteckpulse mit einer negativen Amplitude von 400 mV und einer Breite von > 300 ns benutzt. Pro Pulsfrequenz wurden 10 Messwerte genommen und der statistische Mittelwert mit Standardabweichung berechnet.

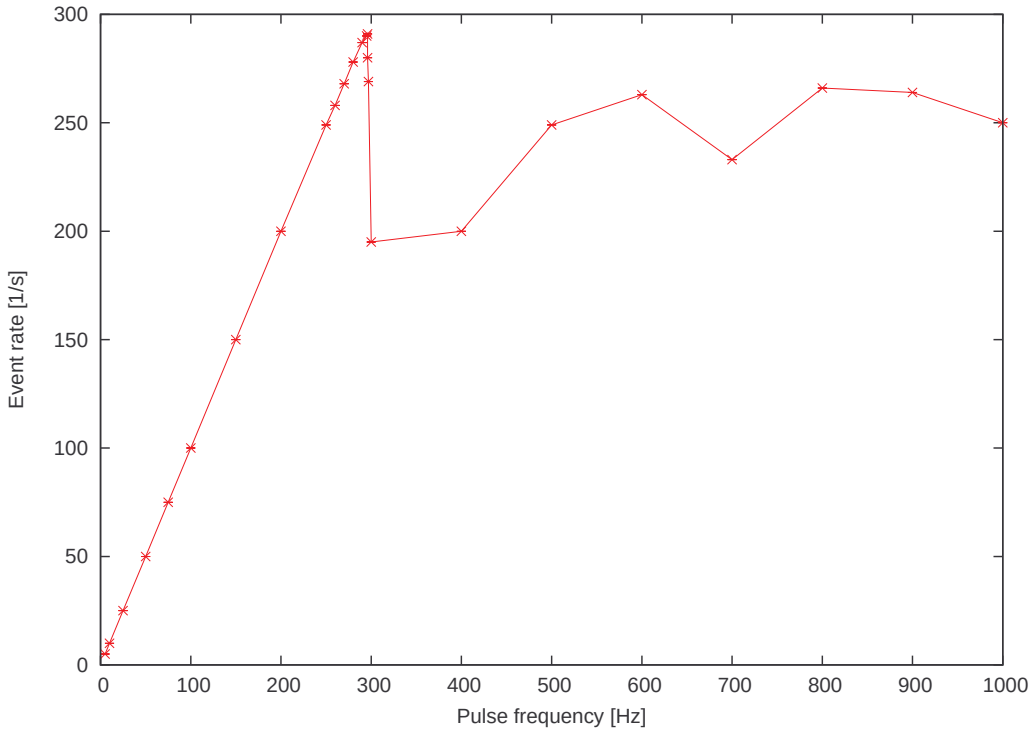

Abbildung 3.5: Ereignisrate gegen Pulsfrequenz ohne Datenspeicherung

In Abb 3.5 ist die oben angesprochene reine Frequenzmessung aufgezeigt. Man sieht, dass die PlugPCs keinesfalls mehr als 300 Hz an Daten verarbeiten können. Oberhalb von 190 Hz treten alle Messwerte mehrfach auf, erlauben also keinen Rückschluss auf die ursprüngliche Pulsfrequenz mehr zu. Daher sind Datenraten über diesem Wert nur noch eingeschränkt verwertbar.
Ein weiterer Versuch sollte zeigen, wie stark die Datenrate nur vom Schreiben der Ereignisdaten beeinflusst wird. Hierbei wurde das vollständige Datennahmeprogramm verwendet, nur die Zeilen, die das Schreiben der Wavearrays in die Textdatei beinhalten, wurden auskommentiert.

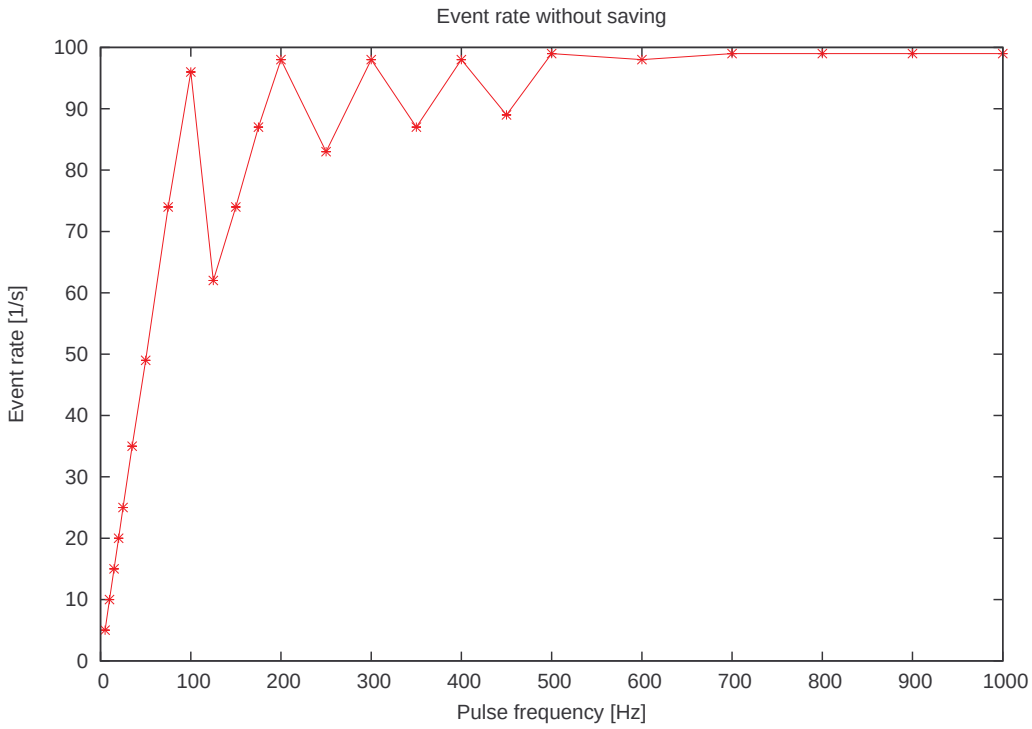

Abbildung 3.6: Ereignisrate gegen Pulsfrequenz ohne Schreiben der Daten

Am Ergebnis dieser Messung, gezeigt in Abb.3.6, sieht man, dass die höchstmögliche Datenrate der aktuellen Konfiguration 100 Hz beträgt, nach Optimierung des Speichersystems sollte die erreichte maximale Datenrate diesem Wert also möglichst nahe kommen.
Bei beiden vorangehend genannten Messungen war der aufgetretene Messfehler zu gering um in diesem Maßstab dargestellt werden zu können.
Die nach dem ersten Maximum in beiden Versuchen auftretenden Wellenformen scheinen eine Resonanzerscheinung zu sein. Bei bestimmten Frequenzen trifft, nachdem die durch einen Lese- und Schreibvorgang eintretende Totzeit verstrichen ist, sofort ein neues Ereignis ein, während bei anderen kurz vor dem Ende der Totzeit ein Ereignis eintrifft, was die Wartezeit des Programms verlängert.
Im Folgenden werden nun die einzelnen Verbesserungsschritte für das Speichersystem beschrieben, beginnend mit der Ausgangskonfiguration, bis hin zur momentan besten Einstellung. Hierzu wurde das oben beschriebene Programm mit einem kleinen Interface kombiniert, das im Abstand von fünf Sekunden den Inhalt der Datei *evcount.txt* liest und ausgibt.

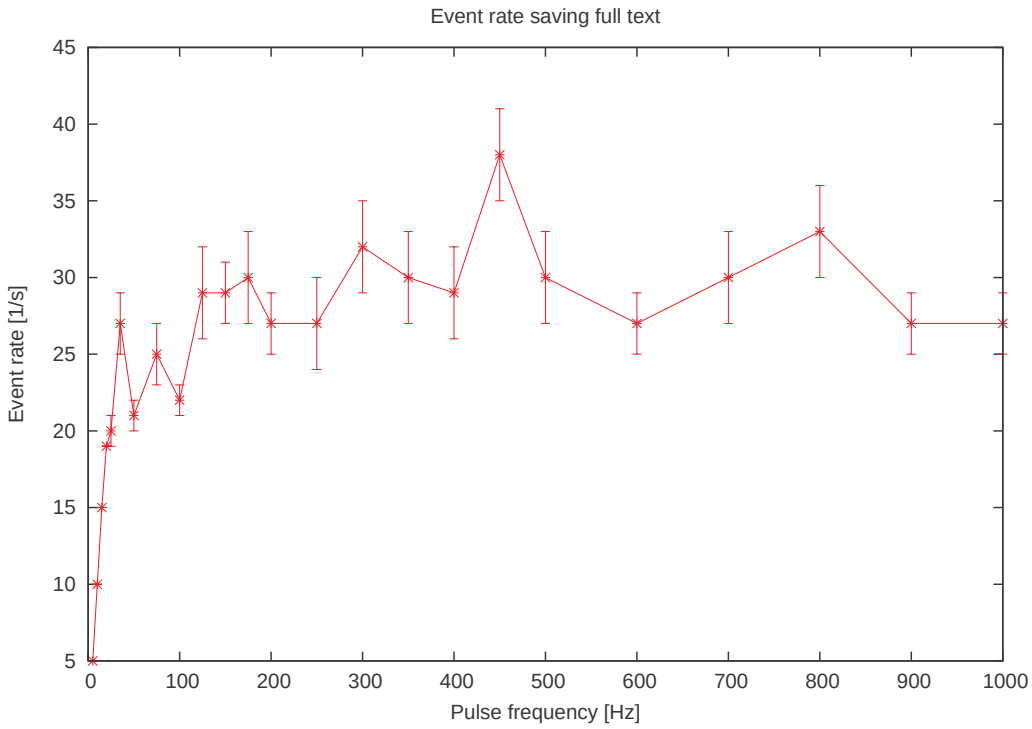

Abbildung 3.7: Ereignisrate gegen Pulsfrequenz bei Speicherung aller Daten

Die in Abb 3.7 aufgezeichnete Ereignisrate hat als Maximum einen Wert von 38,05 Hz und ist nur bis etwa 20 Hz eindeutig auswertbar. Dies ist für die Datennahme bei HiSCORE zu wenig (siehe Kap. 1.2.1), es muss ein Maximum von etwa 100 Hz angestrebt werden.
Um die Verarbeitungsgeschwindigkeit zu verbessern muss das Programmpaket untersucht und nicht notwendige oder zu umständliche Prozesse müssen entfernt oder verkürzt werden. Im Rahmen dieser Diplomarbeit beschränkten sich diese Verbesserungen auf das Programm drs_exam.

Verbesserungen an der Datennahme Als Hauptursache für die langsame Verarbeitung wurde das Speichermanagement des PlugPCs beim Schreiben der Ereignisdaten ausgemacht, denn alle anderen Operationen werden in drs_exam nur ein Mal oder zumindest in (vom Standpunkt der Datenverarbeitung aus gesehen) großen Zeitabständen durchgeführt. Also musste als erstes die Speicherdatei verkleinert werden. Hierzu wurde das Zeitarray entfernt, denn da hier nur die Zeit seit dem Beginn des Ereignisses angezeigt wird und die Zeitabstände der einzelnen Messpunkte ohnehin konstant sind, ist es für die spätere Auswertung nicht nötig, diese Daten zu behalten. Weiterhin wurde die Genauigkeit, mit der die Spannungswerte aufgezeichnet werden, auf eine ganze Zahl reduziert (siehe Zeile 36), da es ausreicht, die Höhe des Spannungspulses auf 1 mV genau zu kennen.

Die Größe einer Speicherdatei beträgt jetzt mit ca. 6 MB nur noch knapp ein Drittel des ursprünglichen Wertes. Dies führt zu einer enormen Erhöhung der Ereignisrate. Der Spitzenwert beträgt hier über 50 Hz und die höchste zuverlässig auswertbare Ereignisrate liegt nun bei etwa 35 Hz, also um 75 % höher als vorher (vergleiche Abb. 3.8, blaue Kurve, mit Abb. 3.7).

```
34  for (i=0 ; i<1024 ; i++){
35     char tt[8];
36     sprintf(tt, "%1.0f %1.0f\n", wave_array[0][i], wave_array[1][i]);
37     f << tt;
38  }
```

Eine mögliche Erklärung für die geringe Datenrate ist auch, dass die langsame Speicherung der Daten in der Struktur des internen Speichers der PlugPCs begründet liegt. Es sollte also getestet werden, wie sich die Datenrate des Programms bei der Benutzung anderer Speichermedien verhält. Da der GuruPlug u.a. über einen Slot für MicroSD Karten verfügt, wurde zunächst mit einer solchen gearbeitet.

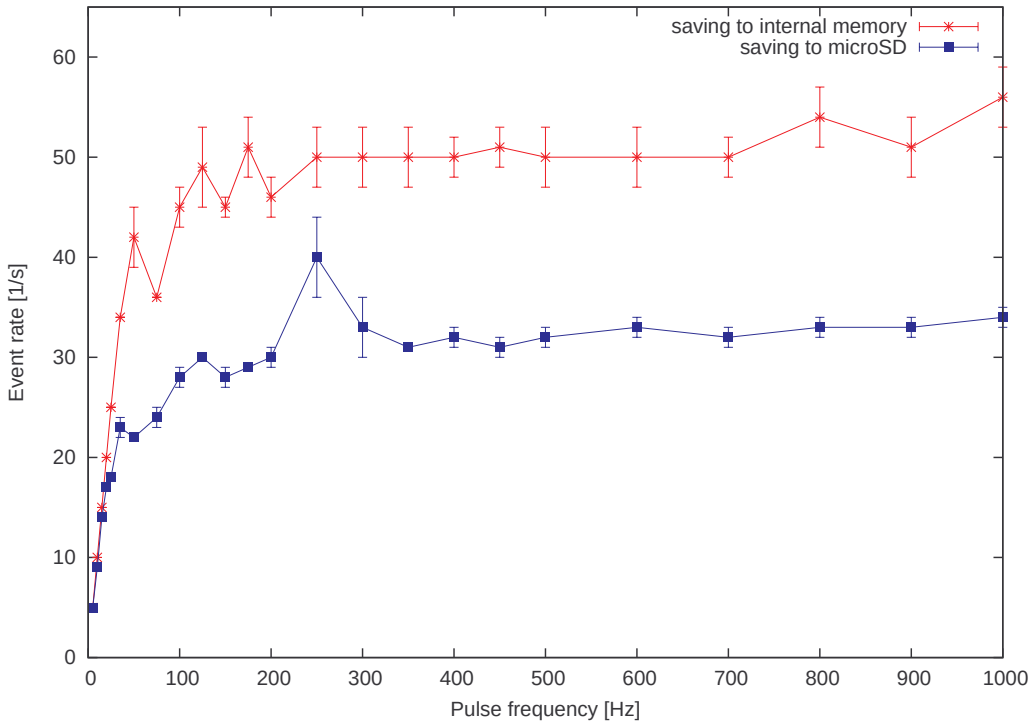

Abbildung 3.8: Ereignisrate gegen Pulsfrequenz für Speicherung auf verschiedenen Medien

Die erreichbare Ereignisrate ist für die Speicherung auf einer MicroSD Karte um über 20 Hz kleiner als bei der internen Speicherung (Abb. 3.8). Die MicroSD Karte ist also nicht als Speichermedium für HiSCORE geeignet. Bis auf weiteres wird also die Datenspeicherung im internen Speicher der PlugPCs durchgeführt.
Die Spannungswerte wurden bisher vom Programm trotz der verkürzten Darstellung immer noch als Fließkommazahlen behandelt und geschrieben, was einen unnötigen Speicherbedarf mit sich brachte. Der nächste logische Schritt war also, ein sog. *casting* an den wave-arrays durchzuführen, d.h. sie vor dem Schreiben in einen anderen Datentyp umzuwandeln. Hierfür

wurde der Typ *short* gewählt, da er mit 16 Bit, gegenüber 32 Bit bei einer Fließkommazahl, einen sehr kleinen Speicherbedarf hat.

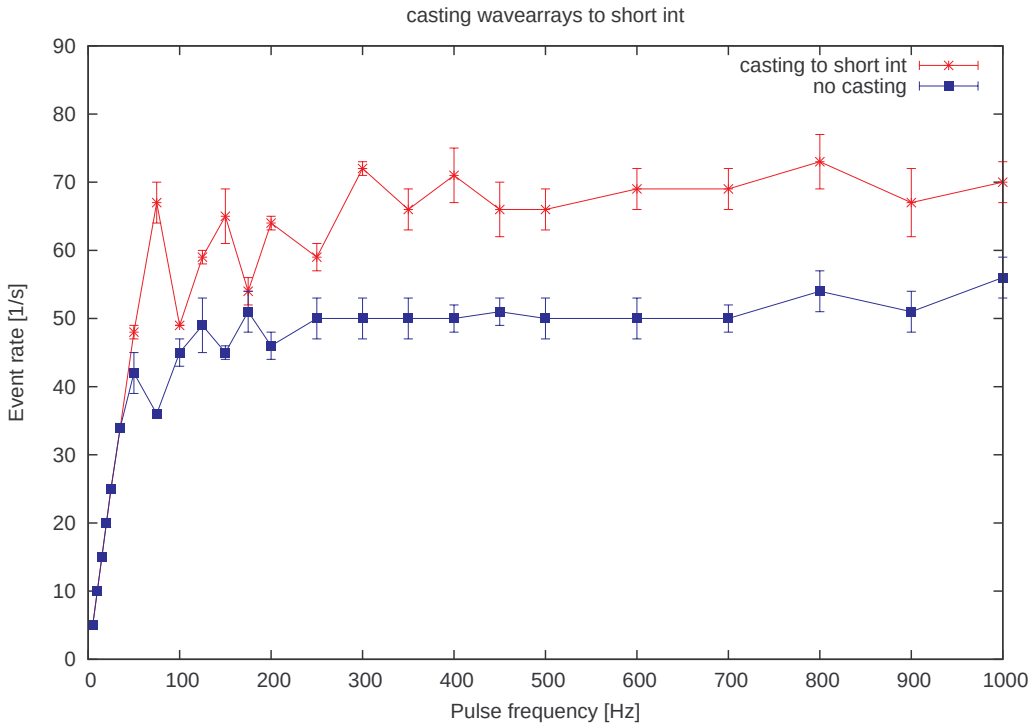

Abbildung 3.9: Vergleich zwischen der Datenspeicherung mit und ohne Casting

Man sieht in Abb. 3.9, dass allein durch die Veränderung des Zahlentyps eine Verbesserung von 15 Hz eintritt.
Eine weitere wichtige Überlegung im Bezug auf die Geschwindigkeit der Datennahme ist die Art des Triggers. In den bisherigen Versionen des Programms wurde ein Software-Trigger verwendet. Das Programm entscheidet also selbst, ob ein Ereignis die für eine Aufnahme erforderliche Intensität hat. Man kann diesen Rechenschritt aber auch durch einen externen (Hardware-)Trigger in Form eines Diskriminators ersetzen. Ein Vergleich der Datenraten für Soft- und Hardwaretrigger ist in Abb. 3.10 zu sehen.

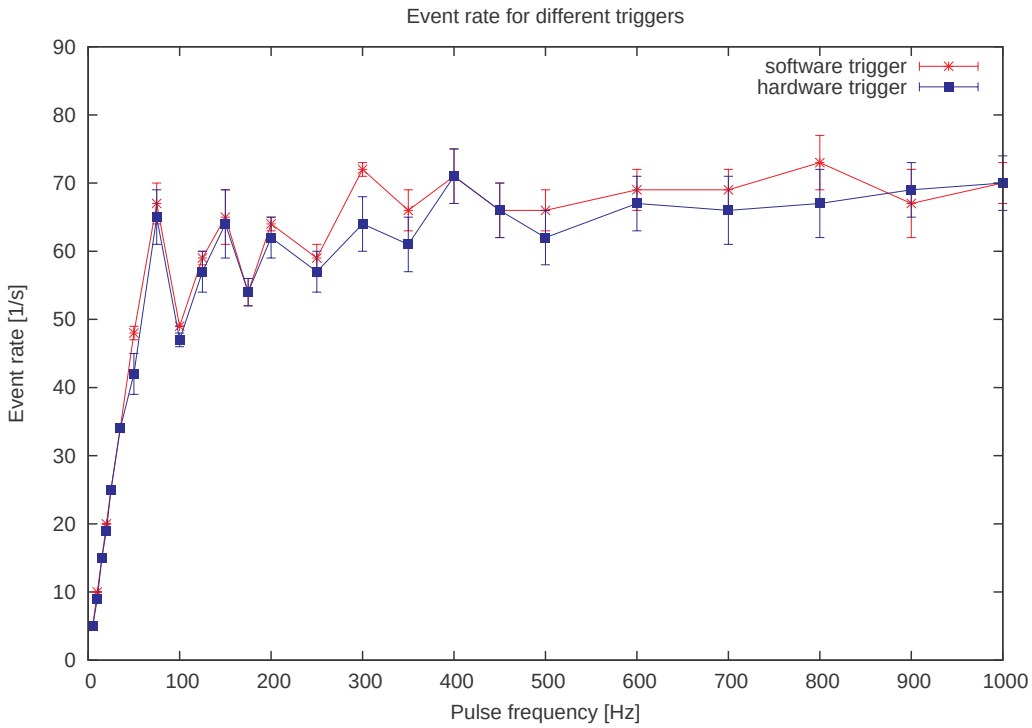

Abbildung 3.10: Vergleich zwischen der Soft- und Hardwaretrigger

Es ist kein signifikanter Unterschied zwischen beiden Triggerformen zu erkennen. Die Art des benutzten Triggers ist also beliebig wählbar.
Eine weitere Möglichkeit zur Steigerung der Schreibgeschwindigkeit besteht in der Speicherung als Binärdatei. Gleichzeitig sollte sich das Schreiben auf die Zahlenwerte aus den beiden *wavearrays* beschränken. Die Stringelemente, also die Ereignisnummer, das Leerzeichen und das Kommando zum Zeilenwechsel, können weggelassen werden. Dies ist möglich, da diese keine wichtigen Daten enthalten, sondern nur der Übersichtlichkeit für den Leser dienen (siehe Zeile 39 bis).
Die einzelnen Ereignisse innerhalb einer Datei können getrennt werden, da bekannt ist, dass für jedes Ereignis 1024 Zeilen mit je zwei Zahlen geschrieben werden, also insgesamt 2048 *Short Integer*-Elemente. Die Nummern der Ereignisse lassen sich aus der Indexnummer der Speicherdatei erschließen (siehe Fußnote 2). Ein Vergleich zwischen der eben beschriebenen Methode und dem Speichern als *short* in einer Textdatei ist in Abb. 3.11 zu sehen.

```
39        for (i=0 ; i<1024 ; i++){
40          short wav0 = (short)wave_array[0][i];
41          fwrite(&wav0,sizeof wav0, 1 ,f);
42          short wav1 = (short)wave_array[1][i];
43          fwrite(&wav1,sizeof wav1, 1 ,f);
44        }
```

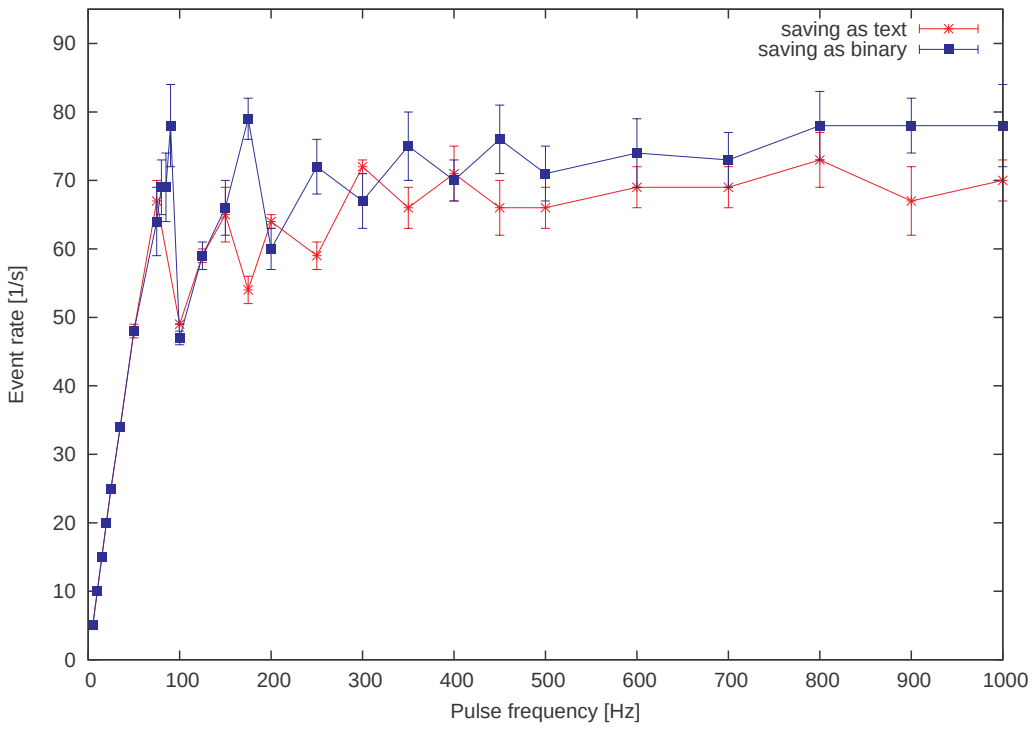

Abbildung 3.11: Vergleich zwischen Text- und Binärspeicherung

Mit dieser Optimierung wurde nunmehr eine Datenrate von über 80 Hz erreicht und die Dateigröße beträgt 3,9 MB. Eine weitere Verbesserung des Speichersystems ist nicht mehr möglich, da nur noch absolut notwendige und unverzichtbare Daten gespeichert werden, die nicht weiter komprimiert werden können.
Im Einzelnen hatten die verschiedenen Messungen folgende Maximalwerte[3].

Modus	maximale Datenrate [1/s]	Fehler [1/s]
Speicherung als voller String	33,65	3,62
Speicherung als kurzer String	57,85	2,93
Speicherung als kurzer String auf MSD	40,5	4,73
Speicherung mit Casting	72,25	1,94
Speicherung mit Casting und externem Trigger	71,1	4,79
Speicherung als Binärdatei	63,55	5,07
Speicherung als verkürzte Binärdatei	78,75	5,04

Unabhängig vom Programm drs_exam sollten auch die anderen Bestandteile des Programmpaketes optimiert werden, denn obwohl die Zahlen als *short* gespeichert werden, werden sie bis dahin intern weiter als *float* behandelt. Eine diesbezügliche Modifikation der Bibliotheksdatei *Drs.h* und der Klasse *Drs.cpp* könnte ohne Zweifel zu einer weiteren Beschleunigung führen. Auch ist es denkbar, dass die Datenspeicherung auf einer an den eSATA-Anschluss des PlugPCs angeschlossenen externen Festplatte zu einer Verbesserung der Datenrate führt.

[3] zuzüglich einer systematischen Ungenauigkeit von 0,01 s^{-1}

Kapitel 4

Hauptsteuerung

4.1 Grundidee

Bisher wurde nur die Funktionsweise der einzelnen Programme betrachtet, nicht jedoch ihre Benutzung in der fertigen HiSCORE Station. Es muss ein Weg geschaffen werden, auf dem die einzelnen Programme über die eingeführten Schnittstellen mit dem Benutzer und ggf. auch untereinander kommunizieren können. Da die einzelnen Stationen nicht über eigene Terminals oder andere externe Schnittstellen verfügen werden und es ohnehin ein nicht zu bewältigender Aufwand wäre, jede Station direkt vor Ort zu bedienen, muss eine Fernsteuerung implementiert werden.
Um den durch die Kommunikation entstehenden Rechenaufwand sowohl für die Zentralstation als auch für die einzelnen Detektorstationen zu minimieren, ist es am besten, ein einzelnes Programm als zentrales Interface auf dem PlugPC in Kombination mit einer grafischen Benutzeroberfläche (engl. **G**raphical **U**ser **I**nterface) auf der Zentralstation zu verwenden. Das Interface bedient die Netzwerkschnittstelle zur Zentralstation und leitet eintreffende Befehle in der richtigen Form an die anderen Prozesse weiter. Über die GUI kann hierbei eine für den Benutzer leicht verständliche Befehlseingabe implementiert werden. Gleichzeitig können die eintreffenden Telemetriedaten übersichtlich präsentiert werden. Eine Illustration der Kommunikationswege ist in Abb. 4.1 gezeigt.

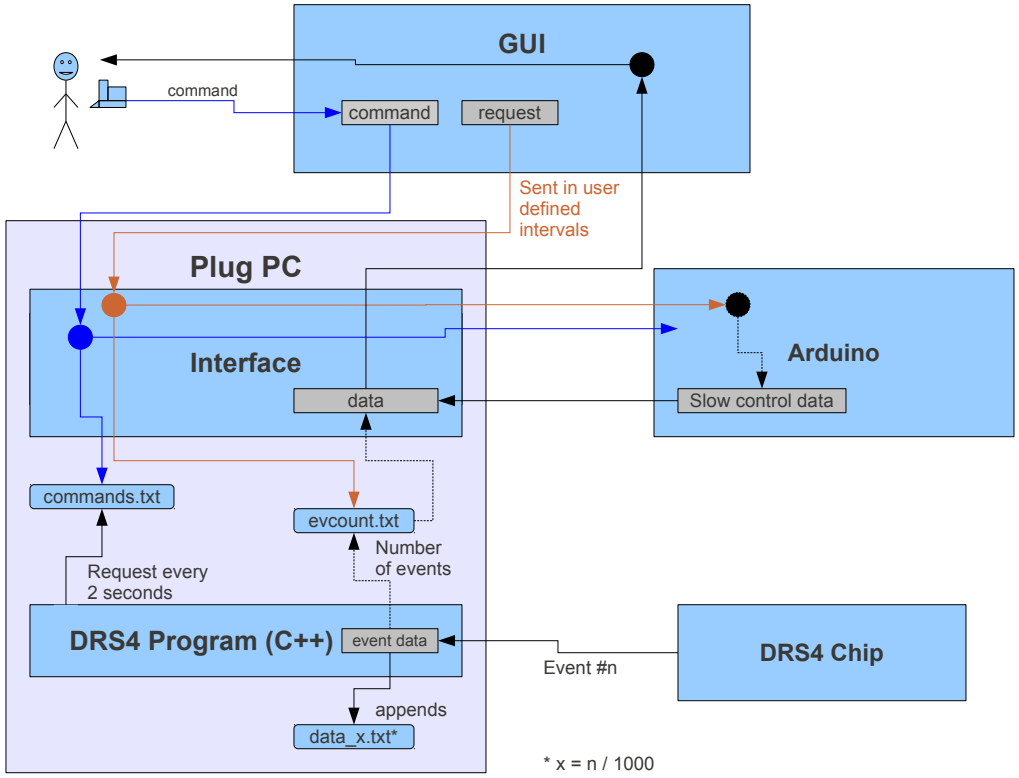

Abbildung 4.1: Darstellung des Funktionsprinzips der Stationssteuerung

Bei HiSCORE werden alle Stationen und der Zentralrechner mit demselben drahtlosen Netzwerk verbunden sein, über das der Datenaustausch stattfinden wird. Da auf allen Computern Linux-basierte Betriebssysteme laufen, bieten sich hier einfache Wege, um die Kommunikation umzusetzen. Für die Kommunikation des GUI Programms auf dem Zentralrechner mit dem Interface auf den PlugPCs wurden mehrere Lösungen in Betracht gezogen. Eine Lösung ist,dass von der Zentralstation ASCII-Dateien mit Befehlen an die einzelnen Messtationen geschickt werden, welche dann vom Interface gelesen werden. Gleichzeitig legt das Interface seine Daten ebenfalls auf dem PlugPC ab, wo sie vom GUI-Programm gelesen werden können. Der Nachteil dieses Systems ist, dass es zu Abstürzen oder Lesefehlern kommen könnte, wenn Interface und GUI gleichzeitig auf dieselbe Datei zuzugreifen versuchen.
Stattdessen findet die Kommunikation zwischen GUI und Interface mittels einer TCP/IP-Verbindung statt. Die Kommunikationspartner sprechen einander im Netzwerk mit ihrer jeweiligen Netzwerkidentität an, was in diesem Fall ihre IP-Adresse ist, die zwecks eindeutiger Identifikation der einzelnen Stationen jedem PlugPC statisch zugewiesen wird. Die Daten werden in Form von Strings ausgetauscht.
Da sich mit der *socket*-Bibliothek eine gut zugängliche TCP/IP Schnittstelle bietet und gleichzeitig mit TKinter eine simple Bibliothek zum Erstellen graphischer Benutzeroberflächen existiert, wurde für das Interface und die GUI Python als passende Programmiersprache ausgewählt.

Bei einer Kommunikation mittels des *socket*-Moduls muss eines der beiden Programme als *host* und eines als *client* definiert sein. Das *host* Programm wartet nach dem Start darauf, dass sich ein *client* Programm anmeldet, so dass beide mit ihrer Arbeit beginnen können. Während dem *host* für die TCP/IP-Verbindung nur ein Kanal zugewiesen werden muss (siehe Zeile 45/46), braucht der *client* zusätzlich noch die Netzwerkadresse eines *hosts* um sich mit diesem verbinden zu können (siehe Zeile 47/48). Welches von beiden Programmen welche Rolle erhält spielt nur bei der Aktivierungsreihenfolge eine Rolle, da sich der *client* sofort wieder abschaltet, bzw. einen Fehler produziert, wenn kein *host* zu finden ist.
Die Definition eines *hosts* funktioniert folgendermaßen:

```
45   host = ''     # this program is used as server for the TCP/IP connection
46   port = 50007   # the port chosen mustn't be used by other processes
```

Ein *client* hingegen wird so definiert:

```
47   host='192.168.1.1'   #designated host(PlugPC)
48   port=50007           #designated port
```

4.2 Funktionsweise des Hauptinterfaces

Wie bereits vorher erwähnt, ist es die Hauptaufgabe des Interfaces, die von Seiten der GUI übermittelten Befehle über eine serielle Verbindung an das auf dem Arduino laufende Slow Control Hauptprogramm und das Datennahmeprogramm, das zeitgleich mit dem Interface auf dem PlugPC läuft, weiter zu geben und seinerseits die von beiden Prozessen gesendeten Telemetriedaten zur Zentralstation zu schicken.
Das Interface Programm für den PlugPC wurde im Rahmen der TCP/IP Kommunikation als *host* definiert. Hinzu kommt noch, dass auf diese Weise der Zentralrechner austauschbar bleibt, was spätere Tests mit mehreren PlugPCs und vor allem das finale HiSCORE Setup sehr erleichtert.
Das Programm öffnet nach dem Start einen neuen socket *host* und wartet dann auf die Verbindungsanfrage eines *client* Programms. Wenn eine solche eintrifft und akzeptiert wird, baut das Interface die Verbindung auf und fährt mit der Initialisierung fort. Da das Interface in jeder Station eigenständig laufen soll, benötigt jeder PlugPC im Netzwerk einen anderen Kanal für seine Kommunikation. Das GUI-Programm wird allein auf dem Zentralrechner laufen und soll zu jedem einzelnen *host* im Netzwerk eine eigene *socket-client*-Verbindung aufbauen. Weiterhin wird die serielle Schnittstelle definiert, über die das Interface mit dem Arduino kommunizieren soll. Es ist wichtig, dass die hier eingestellte Datenrate für die Kommunikation mit der im Slow Control Hauptprogramm für die serielle Schnittstelle festgelegten übereinstimmt, da sonst kein sinnvoller Datenaustausch zwischen Arduino und PlugPC stattfinden kann. Zudem wird die Variable a mit dem Wert 1 initialisiert. Solange $a==1$ gilt, wird nachfolgend eine while-Schleife ausgeführt, die nötig ist, damit das Programm unbegrenzt lange laufen kann (siehe Beschreibung des Programms zur Datennahme).
Nachdem die Initialisierung abgeschlossen ist, beginnt das Programm mit der while-Schleife, die solange ausgeführt wird, bis das Programm den Befehl *quit* erhält. Da ein Aufruf der *recv*-Methode das Programm anhält, bis über die *socket*-Verbindung ein String eintrifft, werden

alle Funktionen des Programmes über Eingaben gesteuert, denn eingabeunabhängige Operationen sind nicht möglich.
Ein eingehender String wird in der Variablen *income* abgelegt (Zeile 49), welche daraufhin per *split*-Anweisung zerlegt und im Array *incoming* gespeichert wird (Zeile 50), dessen erstes Element das Programm dann mit vordefinierten Befehlen vergleicht (Zeile 51 bis 64). Handelt es sich um einen Befehl für das Datenausleseprogramm, dann öffnet das Interface die Datei *commands.txt*, die auch vom Datennahmeprogramm gelesen wird (siehe Kap. 3.2), und schreibt die entsprechenden Befehlszeilen (Zeile 52 bis 59). Hier kommt auch das zweite Element von *incoming* zum Tragen, welches bei einem Befehl zur Änderung der Diskriminatorschwelle den neuen Wert beinhaltet, sonst aber leer ist (Zeile 65).

```
49    income=str(income)
50    incoming=income.split("$")
 ..................................
51    if incoming[0]=="quit":
52        a=0
53        if (os.path.isfile("/root/commands.txt")):
54            writefile=open("/root/commands.txt", "a")  # if the file exists:
                                                          append command
55        else:
56            writefile=open("/root/commands.txt", "w")  # else: create file
57        wlines="command\nquit\n"                       # write command
58        writefile.write(wlines)
59        writefile.close()
60    elif incoming[0]=="start":
.....................................
61        wlines="command\nstart\n"
.....................................
62    elif incoming[0]=="stop":
.....................................
63        wlines="command\nstop\n"
.....................................
64    elif incoming[0]=="enter":
.....................................
65        wlines="setthr\n"+incoming[1]+"\n"
```

Wenn es sich bei dem Kommando hingegen um eine Abfrage der Telemetrie handelt, so schickt das Interface zunächst eine Anforderung an den Arduino (siehe Kapitel 2) und speichert den zurückgegebenen String in der Variablen *reader* (Zeile 66 bis 69). Dann wird der Status der Datennahme ermittelt. Das Interface fragt im Speicher die Datei *found_board.txt* ab. Diesewird vom Programm *drs_exam* geschrieben, wenn es bei der Initialisierung ein DRS4-Board vorfindet. Ist kein Board angeschlossen oder läuft das Programm *drs_exam* nicht, so ist diese Datei nicht vorhanden und die Variable *activedt*, die den Status der Datennahme beinhaltet, wird auf den Wert 2 gesetzt. Ansonsten wird als nächstes die Datei *active.txt* gesucht, die von *drs_exam* geschrieben wird, wenn es mit der Datennahme beginnt. Wenn diese nicht existiert, erhält *activedt* den Wert 0. Existieren hingegen beide Dateien auf dem Rechner, so

bekommt *activedt* den Wert 1 (Zeile 70 bis 85) und das Interface fragt die Ereignisrate der Datennahme ab. Hierzu wird versucht, die Datei *evcount.txt* zu öffnen. Ist die Datei nicht vorhanden, d.h. es findet keine Datennahme statt oder es ist ein Fehler aufgetreten, so wird die Variable *evcount* auf 0 gesetzt (Zeile 78 und 79). Anderenfalls wird die Datei gelesen, ihr Inhalt in der Variablen *evcount* gespeichert (Zeile 74 bis 76) und anschließend gelöscht (Zeile 77), weil sonst auch nach dem Anhalten der Datennahme noch die zuletzt gemessene Ereignisrate angezeigt würde. Findet keine Datennahme statt, so wird *evcount* generell auf 0 gesetzt (Zeile 81 und 82). Das Interface kombiniert nun *reader*, *activedt* und *evcount* zu einem einzigen String, der über *socket* an das GUI Programm geschickt wird (Zeile 86 und 87).

```
66    elif incoming[0] == "q":
67      ser.open()
68      ser.write(income)
69      reader=str(ser.readline())
70      if (os.path.isfile("/root/found_board.txt")):
71          if (os.path.isfile("/root/active.txt")):
72              activedt="1"
73              if (os.path.isfile("/root/meas_data/evcount.txt")):
74                  evcountfile=open("/root/meas_data/evcount.txt","r")
75                  evcount=str(evcountfile.readline())
76                  evcountfile.close()
77                  os.remove(evcountfile)   # delete data after reading
78              else:
79                  evcount = "0"         # else: event rate is 0
80          else:
81              activedt="0"
82              evcount = "0"
83      else:
84          activedt='2'        # else: report missing board
85          evcount = '0'
86     linetowrite=evcount+"$"+reader+"$"+str(activedt)
                    # write all telemetry data to a string

87      conn.send(linetowrite)            # send it to the GUI
88      ser.close()
```

Handelt es sich beim über die *socket*-Verbindung empfangenen String um keines der oben genannten Kommandos, so wird er vom Interface als Befehl an den Arduino eingestuft und einfach an die serielle Schnittstelle gesendet.

```
89   else:
90      ser.open()
91      ser.write(income)
92      dummy6=str(ser.readline())
      # the Arduino will automatically send an empty line
```

```
93      ser.close()
```

4.3 Slow Control GUI

Die GUI der Slow Control dient als Schnittstelle für den Benutzer in der Zentralstation. Ihre Aufgabe ist es einerseits, die vom Benutzer eingegebenen Befehle per *socket* an die einzelnen Stationen weiterzugeben, und andererseits die von den Stationen übertragenen Daten übersichtlich zu präsentieren. Da wie im vorigen Abschnitt bereits erwähnt, eine *recv* Anfrage auf die *socket* Schnittstelle das Programm bis zum Eintreffen eines Datenstroms anhält, wurde entschieden, dass die GUI von der Zentralstation aus in festen, vom Benutzer einstellbaren Intervallen (Standard: 5 s) eine Anfrage nach Telemetriedaten an die Messstation(en) schickt und nur dann die *recv*-Methode aufruft. Die empfangenen Datenstrings sollen zusätzlich zur Anzeige in einen Array mit änderbarer Größe (Standard: 100) geschrieben werden, welcher nach dem Einfügen des letzten Strings in eine Speicherdatei geschrieben und dann geleert wird.
Im Rahmen der *socket*-Kommunikation wurde das GUI-Programm als *client* definiert, was bedeutet, dass es seine Funktion erst dann aufnimmt, wenn es mit dem Interface-Programm einer Messstation verbunden ist. Für die graphische Komponente wird die vorher bereits erwähnte *TKinter*-Bibliothek von Python benutzt [5], die noch durch die *Python Mega Widgets* (*PMW*) ergänzt wird. *TKinter* erlaubt es unter anderem, eine Schaltfläche mit Buttons und Textfenstern für Ein- und Ausgaben zu belegen. Weiterhin können auch Anzeigen verschiedener Art integriert werden, um anzuzeigen, welcher von mehreren vordefinierten Zuständen momentan für eine Variable zutrifft (z.B. der Öffnungszustand des Stationsdeckels). *PMW* ist eine Bibliothek, die es erlaubt in einem *TKinter*-Programm verschiedene Funktionen innerhalb desselben Fensters auf unterschiedliche Register (*Tabs*) zu legen, was die Übersicht verbessert. Für eine Ansicht der GUI siehe Abb. 4.2
Nach dem Aufbau der *socket*-Verbindung startet die Definition der Klasse *SlowControl*, die alle Funktionsparameter der GUI beinhaltet. Hierbei wird als erstes die Initialisierungsfunktion *__init__* definiert, die die genauen Positionen, äußeren Eigenschaften und ggf. die zugewiesenen Kommandos der Textfelder, Buttons, etc. auf der graphische Oberfläche, sowie die Variablendefinitionen enthält (Zeile 94 bis 115).

```
94   class SlowControl():

95      def __init__(self, master):

.....................................................................................
96          self.timestamp=time.time()
97          self.interv=3.67    # this is the interval in which the GUI will
                                  send data requests to the PlugPC
                                (default: 3.67s + ~1,33s delay =~5s)
98          self.income=(0,0,0,0,0,0,0,0,0,0,0)
99          self.readout=2000  # readout calibration: the HV-value
                               corresponding to 5V input to the
                               Arduino (default 2000V)
```

```
100         self.entry=2000  # entry calibration: the HV-value
                               corresponding to 5V output from
                               the Arduino (default 2000V)
101         self.arlen=100  # number of data packages that are saved into
                              the same file (default 100)
102         self.lastset=time.time()
103         self.currentset=time.time()
104         self.evpersec=0     # events/second measured by the DRS4 chip
105         self.takingdata=0   # 1 if DRS4 is taking data, 0 if not
106         self.lidstate='c'  # opening status of lids:
                                'c'=closed,'o'=open,'n'=nn
107         self.hvstate='l'   # HV status: 'l'=low, 'h'=high
108         self.canopen='n'   # n if opening the lids is prohibited by
                                too high light intensity,
                                y if opening is allowed
109         self.canpower='n'  # n if powering the HV system is prohibited
                                by too high light intensity,
                                y if powering is allowed
110         self.magstate='l'   # l if e-magnet is off, h if it is on
111         self.dtthresh=-0.05 # threshold for peak height in DRS4
                                data taking, default=-50mV
112         self.i=0        # internal counter for the save files
113         self.savearr=[]
114         self.lastline="abcdefghijklmn"

115         self.scoreloop()       # run function 'scoreloop'
```

Am Ende der Initialisierung wird die Funktion *scoreloop* Aufgerufen (Zeile 115), die das Herz des GUI-Programms darstellt, denn neben dem sog. *mainloop*, einer Endlosschleife, die in allen *TKinter*-Programmen enthalten sein muss, da das Programm sonst sofort nach dem Start wieder beendet würde, wird hier noch eine weitere Schleife benötigt, um die Telemetriedaten der Messstationen abzufragen und anschließend die Anzeige zu aktualisieren. Ebendiese Aufgaben werden von *scoreloop* übernommen.

```
116   def scoreloop(self):

117      if (time.time()-self.timestamp) >= self.interv:
               # check if enough time has passed since the
                     last query for telemetry data

118          s.send("q")      # send a query to the PlugPC
119          print "q"
120          self.incoming=s.recv(1024)    # wait for data string
121          self.currentset=time.time()
122          self.income=str(self.incoming).split("$")
```

```
                    # split the received string into its components
.....................................................................................
123              self.lidstate=str(self.income[5])
                        # assign the array's elements to the
                          corresponding variables
124              self.hvstate=str(self.income[6])
125              self.canopen=str(self.income[7])
.....................................................
                # insert the variables into the corresponding data fields
126              self.lastline1=string.join(self.income, "$")
127              self.svln=strftime("%a,%d_%b_%Y_%H:%M:%S",localtime())
                                +"$"+self.lastline1 + "\n"
128              self.savearr.append(self.svln)
129              self.timestamp=time.time()

130     if len(self.savearr)>=self.arlen:
             # when the savearray has reached its maximum length, save
               the data to a new file and empty the savearray
131          self.filename='slowcont'+str(self.i)
132          self.FILE=open(self.filename, "w")
133          self.FILE.writelines(self.savearr)
134          self.savearr=[]
135          self.i=self.i+1
136          self.FILE.close()
.....................................................................................
137      if self.canopen=='y': # read 'canopen' variable and draw an
                               oval in the color defined for that value
138          self.canv1_1_4_1.delete(self.circ1_1_4_1)
139          self.circ1_1_4_1=self.canv1_1_4_1.create_oval
                                  (5,3,35,33, fill='green')
.....................................................................................
140      if self.lidstate=='o':    # read 'lidstate' variable and change
                                    the status figure on canv1_1_4_2
141          self.canv1_1_4_2.delete(self.lidline)
142          self.lidline=self.canv1_1_4_2.create_line(0,5,19,5,
                                  fill="red", width=2)
                                  #if lid is complitely open
.....................................................................................
143      self.frame1.after(500, self.scoreloop)
                           # wait for 500 ms and then run the
                             'scoreloop'-function again
```

Als erstes überprüft es, ob seit dem letzten Abfragen der Stationstelemetrie mindestens die in der Variablen *interv* gespeicherte Intervallzeit vergangen ist. Dazu vergleicht es den Zeitpunkt der letzten Abfrage, gespeichert in der Variablen *timestamp*, mit der aktuellen Zeit. Ist es Zeit für eine Telemetrieabfrage, wird der Buchstabe q (für "query") über *socket* gesendet

und aunschließend die Antwort des Stationsinterfaces per *recv* entgegengenommen (Zeile 120 bis 121). Der erhaltene String wird, analog zum Interface, erst als ganzes gespeichert (Variable *incoming*) und dann für die Auswertung zerlegt (Array *income*) (Zeile 122 und 123). Die genaue Zusammensetzung ist wie folgt:

Index	Inhalt	
0	Datenrate	siehe Kap. 3
1	HV-Spannung	siehe Kap. 2
2	Batteriespannung	siehe Kap. 2
3	Strom	siehe Kap. 2
4	Temperatur	siehe Kap. 2
5	Deckelstatus	siehe Kap. 2
6	HV-Status	siehe Kap. 2
7	Deckelfreigabe	siehe Kap. 2
8	HV-Freigabe	siehe Kap. 2
9	Magnet-Status	siehe Kap. 2
10	Feuchtigkeit (momentan nicht benutzt)	siehe Kap. 2
11	Status der Datennahme	siehe Kap. 3

Anschließend werden alle numerischen Anzeigewerte, also die Indizes 0 bis 4, die aktuell eingestellte Größe einer Speicherdatei, die aktuelle Intervalllänge, der Schwellwert der Datennahme sowie die Ein- und Ausgabeskalierung des HV-Systems, die hier in den Variablen *entry* und *readout* gespeichert sind, in die dafür vorgesehenen Textfelder geschrieben, dann wird *timestamp* auf die aktuelle Zeit gesetzt und der String *incoming* wird zusammen mit der Angabe von Datum und Uhrzeit an den Speicherarray angehängt (Zeile 127 bis 128). Danach wird die aktuelle Länge des Arrays geprüft und wenn sie den vorgegebenen Wert erreicht hat, wird der Array in eine Speicherdatei geschrieben und geleert (Zeile 130 bis 136). Die Speicherdateien erhalten eine fortlaufende Nummerierung um die chronologische Einordnung zu erleichtern.
Nach dem Speichern fährt das Programm mit der Aktualisierung der Anzeigen fort. Diejenigen Daten, die nur vordefinierte Werte annehmen können, werden auf der graphischen Oberfläche durch Bilder dargestellt. Der Deckelstatus wird durch eine stilisierte Darstellung eines Stationsgehäuses angezeigt, dessen Deckel sich in der entsprechenden Position befindet (Zeile 140 bis 142), wogegen die anderen Variablen, also die Statusangaben und die Freigaben, durch stilisierte Leuchten (farbige Kreise) dargestellt werden. Die Leuchten sind grün, wenn das entsprechende System aktiv oder eine Aktivierung möglich ist, rot, wenn das System inaktiv oder eine Aktivierung nicht gestattet ist, oder schwarz, wenn ein Fehler aufgetreten ist (beispielsweise Zeile 137 bis 139).
Neben den in *scoreloop* enthaltenen, zyklisch auszuführenden Aktualisierungen der graphischen Anzeigen und internen Variablen, ist das GUI-Programm auch für die Übermittlung von Befehlen für Slow Control und Datennahme an den PlugPC zuständig. Daher können die Aktualisierungen nicht in einer *for*- oder *while*-Schleife stattfinden, denn diese würde bis zu ihrer Beendigung keine anderen Prozesse zulassen, die Übermittlung also verzögern. Stattdessen wird die *after*-Methode benutzt: Nachdem alle Aktualisierungen abgeschlossen sind, erhält das GUI-Programm die Anweisung, *scoreloop* nach Ablauf von 500 ms noch einmal aufzurufen (Zeile 143).
Es besteht auch die Möglichkeit, statt der if-Abfrage am Beginn von*scoreloop* die *after*-

Methode mit *timestamp* aufzurufen. Dann sollte jedoch die Aktualisierung der Anzeige von GUI-intern gespeicherten Variablen, wie den Kalibrationswerten der HV, aus *scoreloop* ausgegliedert und stattdessen in die entsprechenden Übertragungskommandos mit eingebaut werden. So würde sichergestellt, dass die entsprechenden Anzeigen möglichst schnell aktualisiert werden.
Neben der Definition von *scoreloop* enthält das Programm auch die Definitionen der Methoden, die in *__init__* den einzelnen Buttons zugewiesen wurden. Diese bestehen üblicherweise allerdings nur in der Anweisung, einen Befehl über *socket* zu senden und ggf. vorher ein bestimmtes Eingabefeld zu prüfen, wenn ein Zahlenwert eingegeben werden soll (verdeutlicht anhand der HV-Steuerung in Zeile 144 bis 149).

```
144  def hvon(self):    # send command to power HV
145      s.send("h")
146      print 'h'

147  def hvoff(self):   # send command to switch HV off
148      s.send("l")
149      print 'l'
```

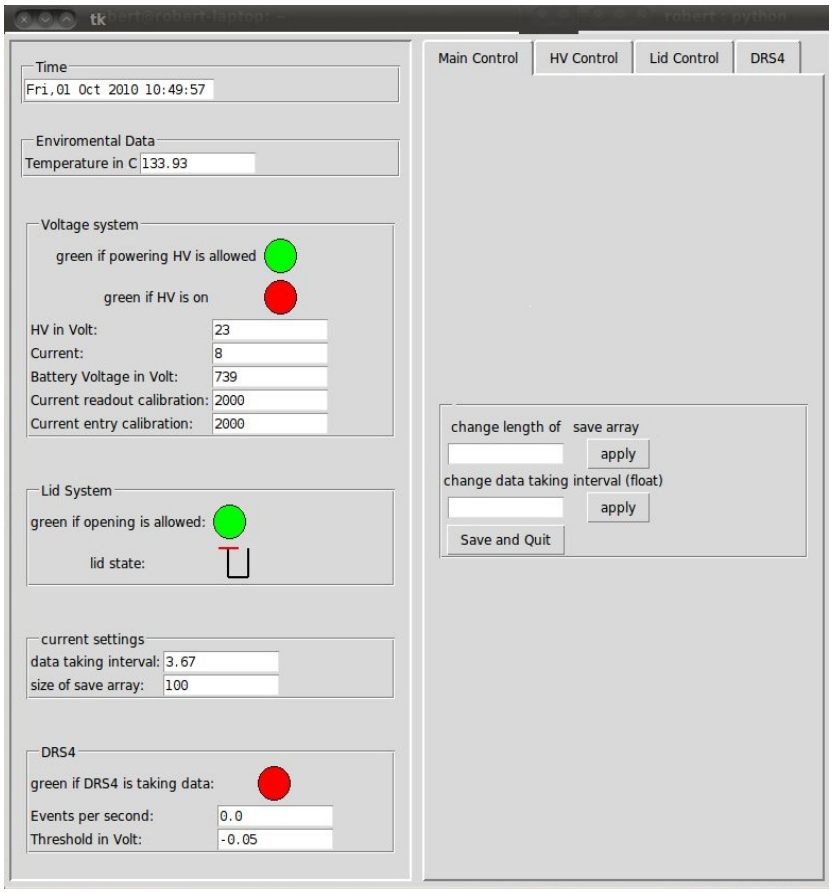

Abbildung 4.2: Ansicht der GUI

4.4 Status der Hauptsteuerung

Sowohl das Interface für den PlugPC als auch das GUI-Programm für die Zentralstation sind bereits einsatzfähig und es wurden bereits einige Testläufe durchgeführt. Der Testaufbau beinhaltete auch ein DRS4-Board und einen Arduino-Mikrocontroller, auf dem das in Kap. 2 Hauptprogramm lief. Da jedoch ein großer Teil der für die Slow Control benötigten Hardware, einschließlich der bereits beschriebenen Platine, nicht zur Verfügung stand, sind die hierbei erhaltenen Slow Control Telemetriedaten sinn- und zusammenhangslos, auch konten einige Steuerelemente nicht getestet werden. Die Zusammenarbeit der vier Programme funktionier-

te im feststellbaren Rahmen fehlerfrei.
Für die Benutzung bei HiSCORE ist das Interface für den PlugPC bereit, das GUI-Programm jedoch muss modifiziert werden, um in einem größeren Netzwerk mit mehreren Stationen gleichzeitig kommunizieren zu können. Weiterhin muss ein Datenbanksystem aufgebaut werden, um bei einer Station vorgenommene Neueinstellungen, wie z.B. eine Veränderung der Auslesekalibration des HV-Systems oder eine Veränderung des Schwellenwertes für die Datennahme, lokal zu speichern und bei jedem Neustart automatisch wieder einzulesen.

Kapitel 5

Zusammenfassung und Ausblick

Die momentan zur Verfügung stehenden Programme ermöglichen den Betrieb einer einzelnen HiSCORE-Station. Das Slow Control Hauptprogramm ist bereits für die in Kap. 1.2.2 genannte Hardware konfiguriert und es werden in Kürze erste Tests mit dem HV-Gerät der Firma ISEG durchgeführt.
Das Datennahme-Programm läuft auf dem PlugPC bereits, so dass problemlos mit der Auslese des DRS-Chips begonnen werden kann. Auch die Kontrolle durch einen externen Benutzer konnte erfolgreich implementiert werden. Das Programm muss für den Gebrauch einer neueren Generation des DRS4-Boards noch leicht angepasst werden. Wichtig ist auch eine Erhöhung der Obergrenze für die Datenrate, da mit einer Ereignisrate von etwa 100 Hz zu rechnen ist. Die Anpassung des Ausleseprogramms hat das Schreiben der Ereignisse bereits um beinahe 100 % beschleunigt. Die am Ende von Kap. 3 angesprochenen Modifikationen sind sehr umfangreich und sollten in Zusammenarbeit mit Mitarbeitern des PSI durchgeführt werden.
Das Hauptinterface läuft auf den PlugPCs und ist in der Lage mit den Programmen für Slow Control und Datennahme zu kommunizieren. Außer der Speicherung der Kalibrationsdaten sind ist hier keine Notwendigkeit zu weiteren Modifikation zu erkennen.
Um die GUI mit mehreren Stationen zu verbinden, muss ein Netzwerk mit mehreren PlugPCs aufgebaut werden, wofür diese umprogrammiert werden müssen. Erste Test dazu haben bereits begonnen. Die Konstruktion eines ersten Stationsprototypen ist bereits in einem fortgeschrittenen Stadium. Mit Ausnahme der Slow Control Platine stehen bereits alle Komponenten zur Verfügung. Der in den Prototyp zu integrierende GuruPlug wurde entsprechend modifiziert, um an die internen 12 Volt Spannungsversorgung der Station angeschlossen werden zu können. Ist die Station fertiggestellt, so soll im Institut für Experimentalphysik der Universität Hamburg mit umfangreichen Tests begonnen werden.
Desweiteren sind die Planungen für einen Testaufbau (auch als *Engineering Array* bezeichnet) mit 25 Stationen in Russland in Zusammenarbeit mit den Mitarbeitern des TUNKA-Experiments [16] bereits weit fortgeschritten.

Literaturverzeichnis

[1] *Arduino homepage*, `http://www.arduino.cc`.

[2] *Drs chip documentation*, `http://drs.web.psi.ch/evaluation/`.

[3] *effbot.org*, `http://www.effbot.org`.

[4] *An introduction to python: File i/o*, `http://www.penzilla.net/tutorials/python/fileio/`.

[5] *An introduction to tkinter*, `http://www.pythonware.com/library/tkinter/introduction/`.

[6] *Plugwiki*, `http://www.plugcomputer.org/plugwiki/index.php/GuruPlug`.

[7] Attila Abramowski, *Zeitabhängige absorption von sehr hochenergetischen photonen aus der richtung des galaktischen zentrums*, Master's thesis, Hamburg, 2008.

[8] F. Aharonian and HESS Collaboration, *Upper limits from hess agn observations in 2005-2007*, November 2007.

[9] F. A. Aharonian, *The project of the hegra imaging cherenkov telescope system: Status and motivation*, Towards a Major Atmospheric Cherenkov Detektor (R.C. Lamb, ed.), vol. II, 1993, Calgary.

[10] J. Albert, E. Aliu, H. Anderhub, P. Antoranz, A. Armada, M. Asensio, C. Baixeras, J. A. Barrio, M. Bartelt, H. Bartko, D. Bastieri, R. Bavikadi, W. Bednarek, K. Berger, C. Bigongiari, A. Biland, E. Bisesi, R. K. Bock, T. Bretz, I. Britvitch, M. Camara, A. Chilingarian, S. Ciprini, J. A. Coarasa, S. Commichau, J. L. Contreras, J. Cortina, V. Curtef, V. Danielyan, F. Dazzi, A. De Angelis, R. de los Reyes, B. De Lotto, E. Domingo-Santamaría, D. Dorner, M. Doro, M. Errando, M. Fagiolini, D. Ferenc, E. Fernández, R. Firpo, J. Flix, M. V. Fonseca, L. Font, N. Galante, M. Garczarczyk, M. Gaug, M. Giller, F. Goebel, D. Hakobyan, M. Hayashida, T. Hengstebeck, D. Höhne, J. Hose, P. Jacon, O. Kalekin, D. Kranich, A. Laille, T. Lenisa, P. Liebing, E. Lindfors, F. Longo, J. López, M. López, E. Lorenz, F. Lucarelli, P. Majumdar, G. Maneva, K. Mannheim, M. Mariotti, M. Martínez, K. Mase, D. Mazin, M. Merck, M. Meucci, M. Meyer, J. M. Miranda, R. Mirzoyan, S. Mizobuchi, A. Moralejo, K. Nilsson, E. Oña-Wilhelmi, R. Orduña, N. Otte, I. Oya, D. Paneque, R. Paoletti, M. Pasanen, D. Pascoli, F. Pauss, N. Pavel, R. Pegna, L. Peruzzo, A. Piccioli, E. Prandini, J. Rico, W. Rhode, B. Riegel, M. Rissi, A. Robert, S. Rügamer, A. Saggion, A. Sánchez, P. Sartori, V. Scalzotto, R. Schmitt, T. Schweizer, M. Shayduk, K. Shinozaki, S. N. Shore, N. Sidro, A. Sillanpää, D. Sobczynska, A. Stamerra, A. Stepanian, L. S. Stark, L. Takalo, P. Temnikov,

D. Tescaro, M. Teshima, N. Tonello, A. Torres, D. F. Torres, N. Turini, H. Vankov, A. Vardanyan, V. Vitale, R. M. Wagner, T. Wibig, W. Wittek, and J. Zapatero, *Observation of Gamma Rays from the Galactic Center with the MAGIC Telescope*, apjl **638** (2006), L101–L104.

[11] A.Morselli, B.Canadas, and V.Vitale on behalf of the Fermi LAT collaboration collaboration collaboration collaboration, *The indirect search for dark matter from the centre of the galaxy with the fermi lat*, December 2010.

[12] T. Antoni and KASCADE, *Kascade measurements of energy spectra for elemental groups of cosmic rays: Results and open problems*, Astropart.Phys. **24** (2005), 1–25.

[13] J. Arons, *Relativistic Shocks and the Excitation of the Crab Nebula*, Millisecond Pulsars. A Decade of Surprise (A. S. Fruchter, M. Tavani, & D. C. Backer, ed.), Astronomical Society of the Pacific Conference Series, vol. 72, 1995, pp. 257–+.

[14] J. Bienlein and R. Wiesendanger, *Einführung in die struktur der materie*, Teubner, 2003.

[15] S. V. Bogovalov and F. A. Aharonian, *Very-high-energy gamma radiation associated with the unshocked wind of the Crab pulsar*, mnras **313** (2000), 504–514.

[16] D. V. Chernov, E. E. Korosteleva, L. A. Kuzmichev, V. V. Prosin, I. V. Yashin, N. M. Budnev, O. A. Gress, T. I. Gress, L. V. Pankov, Y. V. Parfenov, Y. A. Semeney, B. K. Lubsandorzhiev, P. G. Pokhil, T. Schmidt, C. Spiering, and R. Wischnewski, *Primary Energy Spectrum and Mass Composition Determined with the Tunka EAS Cherenkov Array*, International Journal of Modern Physics A **20** (2005), 6799–6801.

[17] Fermi-LAT Collaboration, *Fermi large area telescope measurements of the diffuse gamma-ray emission at intermediate galactic latitudes*, Phys.Rev.Lett. **103** (2009), 251101.

[18] R. Eichler, *Drahtloses netzwerk, slow control und level 2 trigger für den score-detektor*, 2010.

[19] F. Aharonian et al., *The crab nebula and pulsar between 500 gev and 80 tev: Observations with the hegra stereoscopic air cherenkov telescopes*, Astrophys.J.614:897-913,2004 (2004).

[20] E. Fermi, *On the origin of cosmic radiation*, Physical Review **75** (1949), no. 8, 1169–1174.

[21] Milton Virgilio Fernandes, *Untersuchung hochenergetischer gamma-strahlung von westerlund 1 und weiteren galaktischen sternhaufen*, Master's thesis, Universität Hamburg, 2009.

[22] D. Hampf, M. Tluczykont, and D. Horns, *Simulation of expected performance for the proposed gamma-ray detector hiscore*, TEXAS 2010 Proceedings, 2011.

[23] Daniel Hampf, Martin Tluczykont, and Dieter Horns, *Event reconstruction with the proposed large area cherenkov air shower detector score*, September 2009.

[24] D. Heck, J. Knapp, J. N. Capdevielle, G. Schatz, and T. Thouw, *CORSIKA: a Monte Carlo code to simulate extensive air showers.*, February 1998.

[25] W. Heitler, *Quantum theory of radiation*, 1954.

[26] Volkmar Henke, *Studie zur auswertung der ankunftszeitverteilung des Čerenkov-lichts ausgedehnter luftschauer*, Master's thesis, Universität Hamburg, 1994, Diplomarbeit.

[27] G. Hermann, W. Hofmann, T. Schweizer, and et al., *Cherenkov Telescope Array: The next-generation ground-based gamma-ray observatory*, International Cosmic Ray Conference, International Cosmic Ray Conference, vol. 3, 2008, pp. 1313–1316.

[28] V. Hess, *Observation of penetrating radiation of seven balloon flights*, Physikalische Zeitschrift **13** (1912), 1084.

[29] A. M. Hillas, *Cerenkov light images of EAS produced by primary gamma*, International Cosmic Ray Conference (F. C. Jones, ed.), International Cosmic Ray Conference, vol. 3, August 1985, pp. 445–448.

[30] ———, *Cosmic Rays: Recent Progress and some Current Questions*, ArXiv Astrophysics e-prints (2006).

[31] J. Hinton, *Very high energy gamma-ray astronomy with H.E.S.S.*, Journal of Physics Conference Series **110** (2008), no. 6, 062011–+.

[32] D. Horns, *High-(energy)-lights – the very high energy gamma-ray sky*, September 2008.

[33] M. Martinez, *Towards the ground-based gamma-ray observatory CTA*, American Institute of Physics Conference Series (F. A. Aharonian, W. Hofmann, and F. Rieger, eds.), American Institute of Physics Conference Series, vol. 1085, December 2008, pp. 824–829.

[34] R. Mukherjee, *Egret (gev) blazars*, AIP Conf.Proc. **558** (2001), 324–337.

[35] D. E. Nagle, T. K. Gaisser, and R. J. Protheroe, *Extensive air showers associated with discrete astrophysical sources*, Annual Review of Nuclear and Particle Science **38** (1988), 609–657.

[36] M. Odendahl, J. Finn, and A. Wenger, *Arduino - physical computing für bastler, designer & geeks*, O'Reilly, 2009.

[37] L. Scarsi, K. Bennett, G. F. Bignami, G. Boella, R. Buccheri, W. Hermsen, L. Koch, H. A. Mayer-Hasselwander, J. A. Paul, and E. Pfeffermann, *The COS-B experiment and mission*, Recent Advances in Gamma-Ray Astronomy (R. D. Wills & B. Battrick, ed.), ESA Special Publication, vol. 124, July 1977, pp. 3–11.

[38] I. S. Shklovskii, *On the Nature of the Optical Emission from the Crab Nebula.*, azh **34** (1957), 706–+.

[39] G. Sinnis, *HAWC: A Next Generation VHE All-Sky Telescope*, High Energy Gamma-Ray Astronomy (F. A. Aharonian, H. J. Völk, and D. Horns, eds.), American Institute of Physics Conference Series, vol. 745, February 2005, pp. 234–245.

[40] T. Tanimori, K. Sakurazawa, S. A. Dazeley, P. G. Edwards, T. Hara, Y. Hayami, S. Kamei, T. Kifune, T. Konishi, Y. Matsubara, T. Matsuoka, Y. Mizumoto, A. Masaike, M. Mori, H. Muraishi, Y. Muraki, T. Naito, S. Oda, S. Ogio, T. Osaki, J. R. Patterson, M. D. Roberts, G. P. Rowell, A. Suzuki, R. Suzuki, T. Sako, T. Tamura, G. J. Thornton, R. Susukita, S. Yanagita, T. Yoshida, and T. Yoshikoshi, *Detection of Gamma Rays of up to 50 TeV from the Crab Nebula*, apjl **492** (1998), L33+.

[41] M. Tavani, G. Barbiellini, A. Argan, F. Boffelli, A. Bulgarelli, P. Caraveo, P. W. Cattaneo, A. W. Chen, V. Cocco, E. Costa, F. D'Ammando, E. Del Monte, G. de Paris, G. Di Cocco, G. di Persio, I. Donnarumma, Y. Evangelista, M. Feroci, A. Ferrari, M. Fiorini, F. Fornari, F. Fuschino, T. Froysland, M. Frutti, M. Galli, F. Gianotti, A. Giuliani, C. Labanti, I. Lapshov, F. Lazzarotto, F. Liello, P. Lipari, F. Longo, E. Mattaini, M. Marisaldi, M. Mastropietro, A. Mauri, F. Mauri, S. Mereghetti, E. Morelli, A. Morselli, L. Pacciani, A. Pellizzoni, F. Perotti, G. Piano, P. Picozza, C. Pontoni, G. Porrovecchio, M. Prest, G. Pucella, M. Rapisarda, A. Rappoldi, E. Rossi, A. Rubini, P. Soffitta, A. Traci, M. Trifoglio, A. Trois, E. Vallazza, S. Vercellone, V. Vittorini, A. Zambra, D. Zanello, C. Pittori, B. Preger, P. Santolamazza, F. Verrecchia, P. Giommi, S. Colafrancesco, A. Antonelli, S. Cutini, D. Gasparrini, S. Stellato, G. Fanari, R. Primavera, F. Tamburelli, F. Viola, G. Guarrera, L. Salotti, F. D'Amico, E. Marchetti, M. Crisconio, P. Sabatini, G. Annoni, S. Alia, A. Longoni, R. Sanquerin, M. Battilana, P. Concari, E. Dessimone, R. Grossi, A. Parise, F. Monzani, E. Artina, R. Pavesi, G. Marseguerra, L. Nicolini, L. Scandelli, L. Soli, V. Vettorello, E. Zardetto, A. Bonati, L. Maltecca, E. D'Alba, M. Patané, G. Babini, F. Onorati, L. Acquaroli, M. Angelucci, B. Morelli, C. Agostara, M. Cerone, A. Michetti, P. Tempesta, S. D'Eramo, F. Rocca, F. Giannini, G. Borghi, B. Garavelli, M. Conte, M. Balasini, I. Ferrario, M. Vanotti, E. Collavo, and M. Giacomazzo, *The AGILE Mission*, aap **502** (2009), 995–1013.

[42] M. Tluczykont, D. Hampf, D. Horns, T. Kneiske, R. Eichler, R. Nachtigall, and G. Rowell, *The ground-based wide-angle gamma-ray and cosmic-ray experiment hiscore*, Cospar 2010 Proceedings, 2011.

[43] M. Tluczykont, T. Kneiske, D. Hampf, and D. Horns, *Gamma-ray and cosmic ray astrophysics from 10 tev to 1 eev with the large-area (>10 km^2) air-shower detector score*, September 2009.

Abbildungsverzeichnis

Anhang A

Slow Control

A.1 Temperaturmessung

Die finalen Programme für die Temperaturmessung mit dem LM35 Temperaturfühler sehen wie folgt aus:

Das Leseprogramm Temperature_Read_LM35_XBee

```
int temperaturePin = 4;
int input = 0;
float intwo;
float temp = 0;
int i = 0;
int day = 1;
int hour = 15;
int minute = 15;
int currPin = 11;

void setup(){
  Serial.begin(9600);
  delay(10000);
  pinMode(currPin, OUTPUT);
  digitalWrite(currPin, HIGH);
}


void loop(){

  input = analogRead(temperaturePin);
  intwo = float(input);
  temp = ((intwo*0.004888))*100;
  //diese Zeile übersetzt analogen Input
  //in numerische Werte
```

```
  if(minute >= 60){
    hour = hour + 1;
    minute = minute%60;
  }

  if(hour == 24){
    hour = 0;
    day++;
  }
  Serial.print("Day:");
  Serial.print(day);
  Serial.print(",");
  if (hour < 10){
    Serial.print(0);
  }
  Serial.print(hour);
  Serial.print(".");
  if(minute < 10){
    Serial.print(0);
  }
  Serial.print(minute);
  Serial.print(" ");
  Serial.print(temp);
  minute = minute+5;

  i = (i+1)%10;
   delay(300000);
}
```

Das Programm text_in_out_1

```
char received[20];

void setup(){
  Serial.begin(9600);
}




void loop(){
  if(Serial.available() > 0){
    Serial.println("                ");
//diese Zeile ist wichtig, da ohne sie
```

```
        //jedes Byte im Array als einzelner
        //String behandelt würde
    for (int i = 0; i < 20 ;i++){
      if (Serial.available() > 0){
        received[i] = Serial.read();
      }
      else{
       received[i] = 0;
      }

    }
      Serial.write(received);
  }
}
```

A.2 Latenzzeitmessungen

Messung für 19200 Baud Übertragungsgeschwindigkeit.

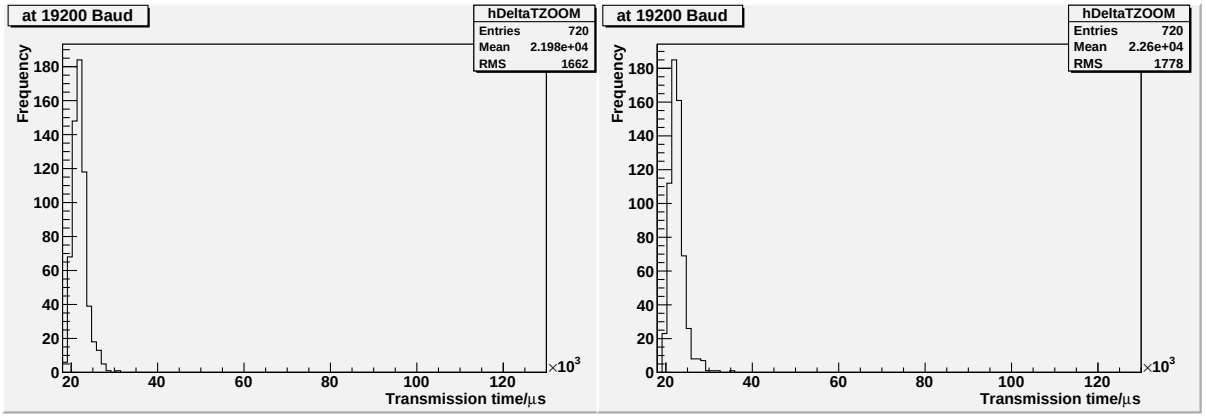

Abbildung A.1: Zeitverteilung für 1 Byte, gesendet als neue Zeile (links) und als String (rechts)

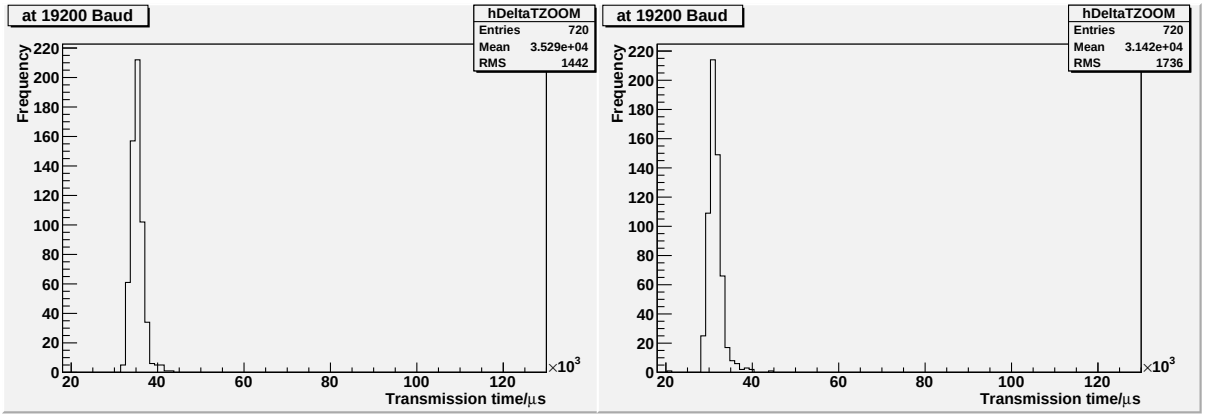

Abbildung A.2: Zeitverteilung für 5 Bytes, gesendet in mehreren Zeilen (links) und als String (rechts)

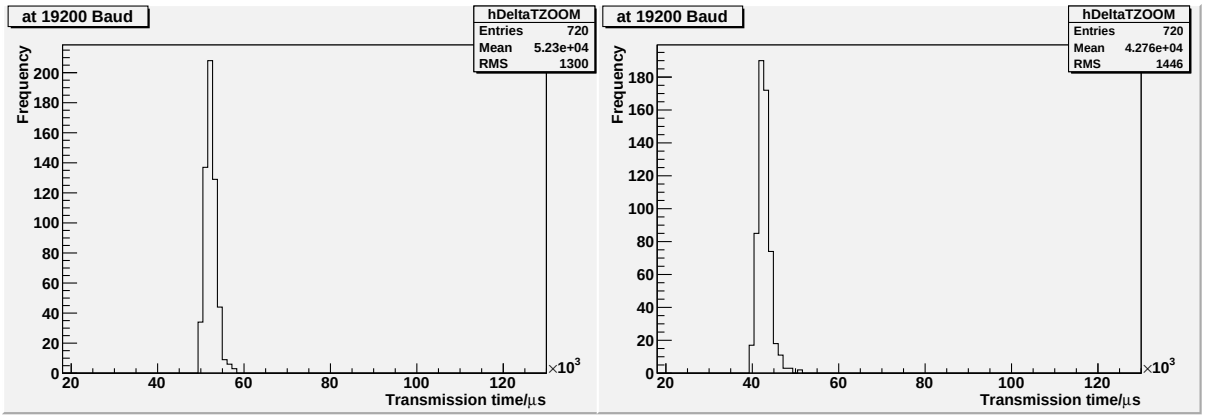

Abbildung A.3: Zeitverteilung für 10 Bytes, gesendet als neue Zeile (links) und als String (rechts)

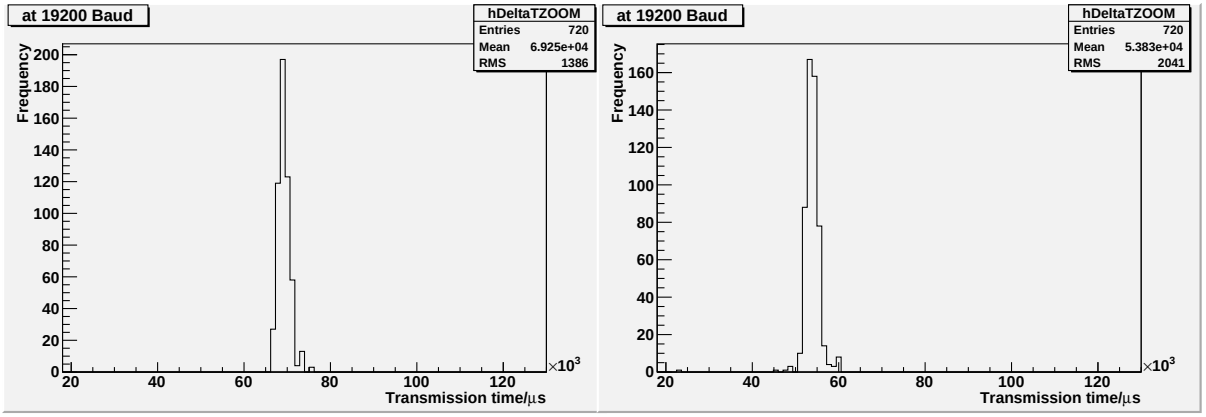

Abbildung A.4: Zeitverteilung für 15 Bytes, gesendet als neue Zeile (links) und als String (rechts)

Messung für 9600 Baud Übertragungsgeschwindigkeit.

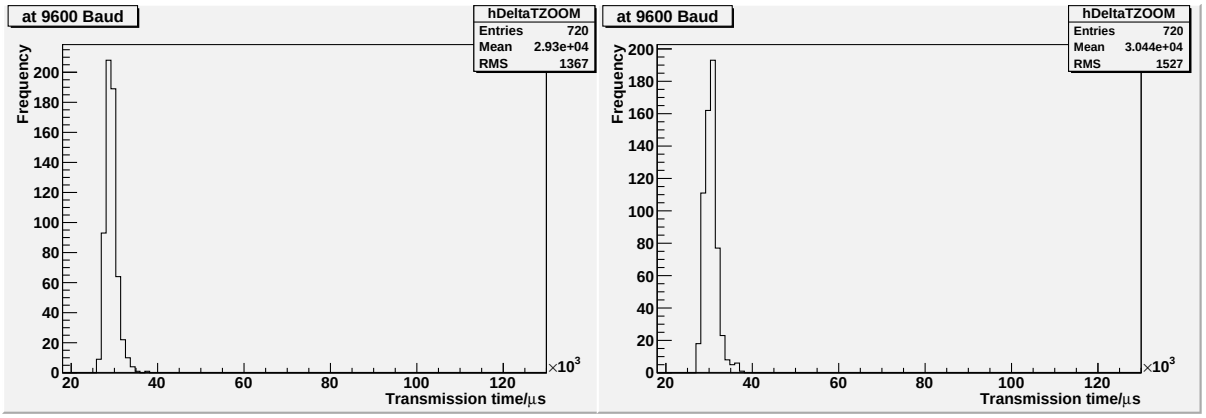

Abbildung A.5: Zeitverteilung für 1 Byte, gesendet als neue Zeile (links) und als String (rechts)

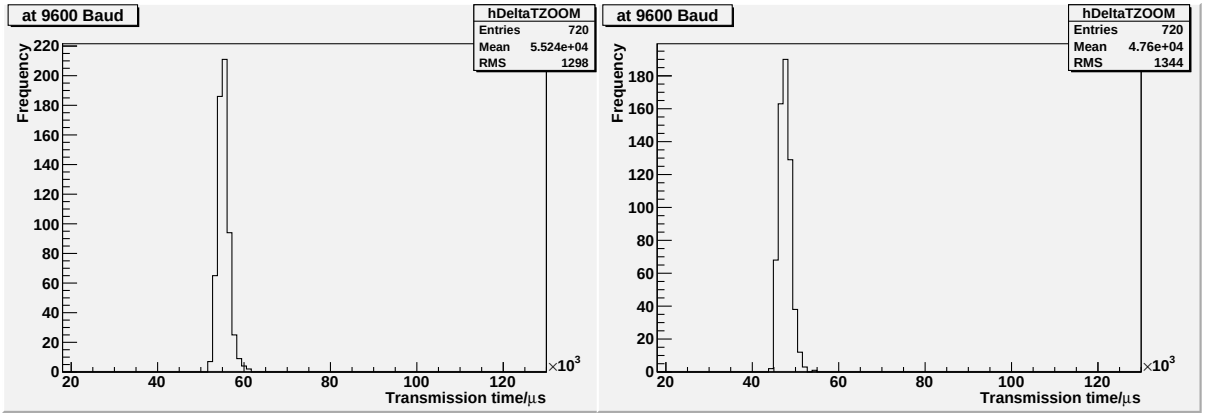

Abbildung A.6: Zeitverteilung für 5 Bytes, gesendet als einzelne Zeilen (links) und als String (rechts)

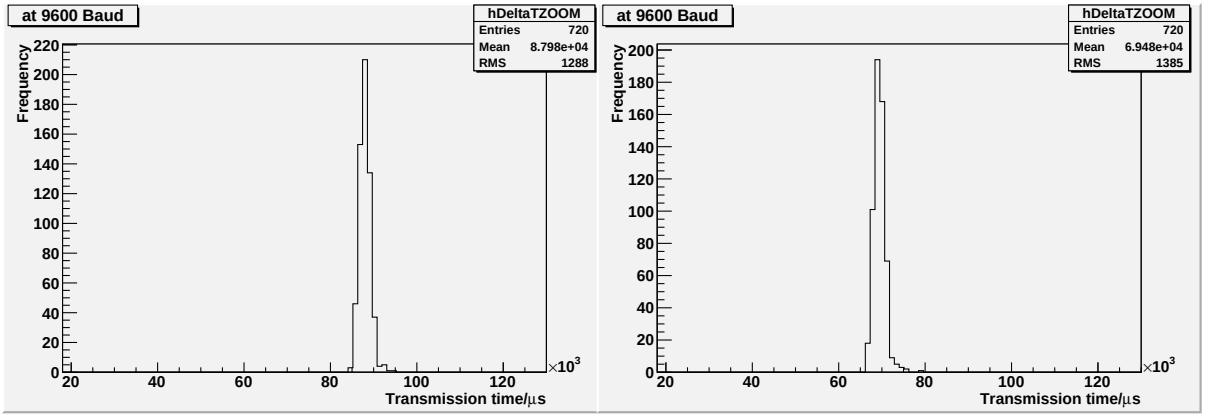

Abbildung A.7: Zeitverteilung für 10 Bytes, gesendet als einzelne Zeilen (links) und als String (rechts)

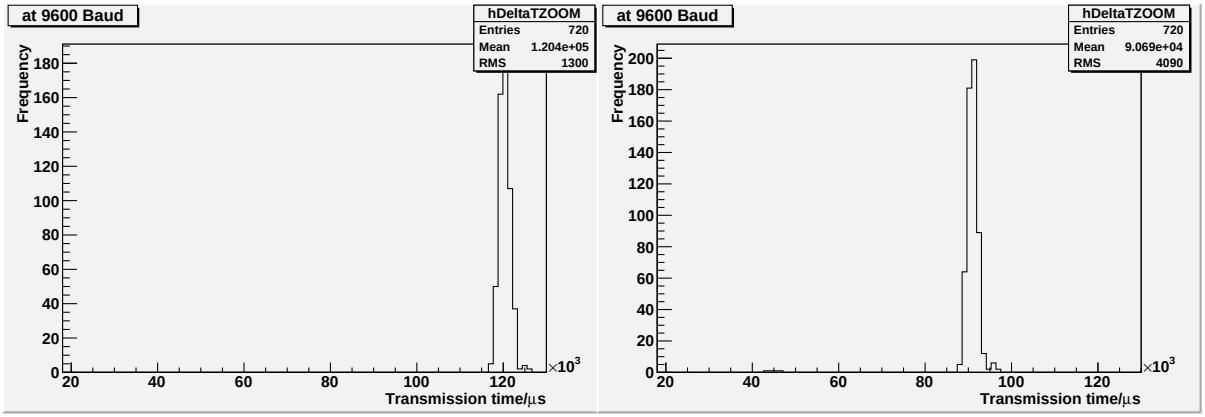

Abbildung A.8: Zeitverteilung für 15 Bytes, gesendet als einzelne Zeilen (links) und als String (rechts)

A.3 Slow Control Hauptprogramm

```
int temperaturePin = 3;
int input = 0;
float intwo;
float temp = 0;

int voltPin = 1;
int volt = 0;
int hvPin = 7;
int regPin = 6;
char volts[5];
int hvstrength = 0;
int hvwrite;
int hv;

int currPin = 2;
int curr = 0;
int readcurr = 0;

int batPin = 0;
int batval = 0;
int bat = 0;

int lightPin = 5;

int light = 0;
int income;
int lidpower = 0;
int open = 0;
int openPin = 12;
```

```
int closePin = 13;
int thresh = 100;
         // maximum light strength (values between 0 and 1023)
int lidPinC = 2;
       // the digital Pin the close sensor is connected to
int lidPinO = 4;
boolean lidClosed;
boolean lidOpen;
char lidstate = 'c';
char canopen = 'n';
unsigned long optime;

unsigned long hvtime;
int hvon = 0;
int hvoff = 0;
boolean hvHigh = false;
int hvthresh = 110;        // maximum light strength for hv
int hvset = 0;
int hvaim;
int hvstatePin = 3;
char hvstate = 'l';
char canpower = 'n';
int light2;

int interv = 1000;                  // interval of data taking
char timeread[4];
int calib1 = 2000;  //hv read out calibration (default: 2000 V = 1023)
int calib2 = 2000;   //hv write calibration (default: 2000 V = 255)

int threshPin = 5;
int setthresh;
int threshwrite;

int magnetPin = 8;
char magstate = 'l';

int humidPin = 4;
int humid = 0;

boolean donothing = true;

void setup(){
  Serial.begin(9600);
  pinMode(hvPin,OUTPUT);
  pinMode(regPin,OUTPUT);
  pinMode(openPin, OUTPUT);
  pinMode(closePin, OUTPUT);
```

```
  pinMode(lidPinO,INPUT);
  pinMode(lidPinC,INPUT);
  pinMode(threshPin, OUTPUT);
  pinMode(magnetPin, OUTPUT);
  pinMode(hvstatePin, INPUT);
}


void loop(){

   if(Serial.available() > 0){
       Serial.println("                                                    ");


    for(int i = 0; i < 4; i++){
      if(Serial.available() > 0){
        volts[i] = Serial.read();
      }
      else{
        volts[i] = 'a';
      }
    }

    switch(volts[0]){
      case 'h':           // input says "HV on"
        hvon = 1;
        hvoff = 0;
        break;
      case 'l':           // input says "HV off"
        hvoff = 1;
        hvon = 0;
        break;
      case 'o':      // input says "open lids!"
        if(lidpower==1 && open==0){
          optime=millis();
        }
        lidpower = 1;
        open = 1;
        break;
      case 'c':          // input says "close lids!"
        if(open == 1){
          optime = millis();
        }
        open = 0;
        lidpower = 1;
        break;
      case 'v':      // Change calibration for HV Readout
```

```
  for(int i = 0; i <4; i++){
    timeread[i] = volts[i+1];
  }
  calib1 = atoi(timeread);
  break;
case 'w':      // Change calibration for HV writing
  for(int i = 0; i <4; i++){
    timeread[i] = volts[i+1];
  }
  calib2 = atoi(timeread);
  break;
case 't':        // Change light threshold for lid control
   for(int i = 0; i <4; i++){
    timeread[i] = volts[i+1];
  }
  thresh = atoi(timeread);
  break;
case 'm':       // Change light threshold for HV control
   for(int i = 0; i <4; i++){
    timeread[i] = volts[i+1];
  }
  hvthresh = atoi(timeread);
  break;
case 'q':         // query for slow control data
  input = analogRead(temperaturePin); // read out temperature
  intwo = float(input);
  temp = ((intwo*0.004888))*100;
  volt = analogRead(voltPin);      // read out voltage
  hv = (volt * calib1)/1023;
  batval = analogRead(batPin);
  bat = (batval * 15) / 1023;
  readcurr = analogRead(currPin);
  curr = (readcurr * 200)/1023;
  humid = analogRead(humidPin);


  Serial.print(hv);
  Serial.print("$");
  Serial.print(bat);
  Serial.print("$");
  Serial.print(curr);
  Serial.print("$");
  Serial.print(temp);
  Serial.print("$");
  Serial.print(lidstate);
  Serial.print("$");
  Serial.print(hvstate);
```

```
      Serial.print("$");
      Serial.print(canopen);
      Serial.print("$");
      Serial.print(canpower);
      Serial.print("$");
      Serial.print(magstate);
      Serial.print("$");
      Serial.print(humid);
      Serial.print("$");
      break;
    case 's':              // write new threshold
       for(int i = 0; i <4; i++){
        timeread[i] = volts[i+1];
      }
      setthresh = atoi(timeread);
      threshwrite = 51 * setthresh;
      analogWrite(threshPin, threshwrite);

      break;
    case 'e':
       digitalWrite(magnetPin, HIGH);
       magstate='h';
       break;
    case 'f':
       digitalWrite(magnetPin, LOW);
       magstate='l';
       break;
    case 'a':
         // else: set HV
      if(hvHigh){
       hvstrength = atoi(volts);
       if (hvstrength <= 2000){
         hvwrite = (hvstrength * 255)/calib2;
       }
      }
      break;

     default:
      donothing = true;
  }


}

light = analogRead(lightPin);     // check light strength
```

```
if (digitalRead(lidPinO) == HIGH){
                    // check if lid is completely closed
  lidOpen = true;
  lidClosed = false;
  lidstate = 'o';
}
else{
  if (digitalRead(lidPinC) == HIGH){
              // check if lid is completely open
    lidOpen = false;
    lidClosed = true;
    lidstate = 'c';
  }
  else{
    lidOpen = false;
    lidClosed = false;
    lidstate = 'n';
  }
}

 if (light < thresh && hvHigh){
   canopen = 'y';
 }
 else{
   canopen = 'n';
   if (open == 1){
      open = 0;
      lidpower = 0;
      optime = millis();
   }
   if (lidpower == 0 && (optime - millis()) > 500 && !lidClosed){
      lidpower = 1;
   }
 }


if (lidOpen && open == 1){
                  // if the command says 'open' and the lid is open
  open = 0;    // don't open
  lidpower=0;
}

if (lidClosed && lidpower == 1 && open == 0){
                 // if the command says 'close' and the lid
                           is already closed
  lidpower = 0;              // don't close
}
```

```
if(open == 1 && lidpower == 1 && (optime - millis()) > 500){
  digitalWrite(openPin, HIGH);
  digitalWrite(closePin, HIGH);  // if the command says open, power motor

}
else{
  if(lidpower == 1 && open == 0 && (optime - millis()) > 500){
    digitalWrite(magnetPin, LOW);   // if command says close power motor
    magstate='l';
    digitalWrite(openPin, LOW);
    digitalWrite(closePin, HIGH);

  }
  else{
    digitalWrite(openPin, LOW);  // else power off
    digitalWrite(closePin, LOW);

  }

}

light2 = analogRead(lightPin);     // check light strength


if(light2 < hvthresh){
  canpower = 'y';
}
else{
  canpower = 'n';
}

if(canpower=='n' && hvHigh){
  hvoff = 1;
  hvon = 0;
}

if(hvHigh && hvon == 1){
  hvon = 0;
}

if(!hvHigh && hvoff == 1){
  hvoff = 0;
}

if(hvon == 1){
```

```
  digitalWrite(hvPin, HIGH);
}

if(hvoff == 1){
  digitalWrite(hvPin, LOW);
  hvwrite = 0;
}

if(digitalRead(hvstatePin)==HIGH){
  hvHigh = true;
}
else{
  hvHigh = false;
}

if (hvHigh){
  hvstate = 'h';
}
else{
  hvstate = 'l';
}


if(hvwrite > hvset){       // slowly increase HV until desired value is reached
  hvaim = hvwrite- hvset;
      if((hvaim % 13)!=0){
        hvset = hvset + (hvaim % 13);
        hvaim = hvaim - (hvaim % 13);
        hvtime = millis();
      }
      else{
       if(millis()-hvtime >= 200){
          hvset = hvset + 13;
          hvaim = hvaim - 13;
          hvtime = millis();
       }
      }
    analogWrite(regPin, hvset);
  }
else{
  if(hvwrite <  hvset){
    hvaim = hvset - hvwrite;
    if((hvaim % 13) !=0){
      hvset = hvset - (hvaim % 13);
      hvaim = hvaim -(hvaim % 13);
      hvtime = millis();
    }
```

```
      else{
        if((millis()-hvtime) >= 200){
          hvset = hvset -13;
          hvaim = hvaim -13;
          hvtime = millis();
        }
      }
    }
    analogWrite(regPin, hvset);
  }

  if((analogRead(voltPin)/4)!=hvset){
    analogWrite(regPin, hvset);
  }

}
```

A.4 LED-Matrix

```
 uint8_t rows[7] = {1, 2, 3, 4, 5, 6, 7};
uint8_t columns[5] = {8, 9, 10, 11, 12};
uint8_t matrix[7][5] = {
{0, 0, 0, 0, 0},
{0, 0, 0, 0, 0},
{0, 0, 0, 0, 0},
{0, 0, 0, 0, 0},
{0, 0, 0, 0, 0},
{0, 0, 0, 0, 0},
{0, 0, 0, 0, 0}
};
int input[2];
int row;
int col;

void setup(){
  Serial.begin(9600);
   for(int i = 0; i < 7; i++){
    pinMode(rows[i],OUTPUT);
  }
  for(int i = 0; i < 5; i++){
    pinMode(columns[i], OUTPUT);
  }
  for (int k = 0; k < 7; k++){
    digitalWrite(rows[k], HIGH);
  }
}
```

```
void loop(){
  if (Serial.available() > 0){
   for(int y = 0; y < 7; y++){
      digitalWrite(rows[y], 1);
    }
  for(int x = 0; x < 5; x++){
    digitalWrite(columns[x],0);
  }
  for (int c = 0; c < 2; c++){
    input[c] = Serial.read();
  }
  row = input[0]-49 ;
  col = input[1]-49;

  }



  digitalWrite(rows[row], 0);
  digitalWrite(columns[col],1);
}
```

Anhang B

DAQ-Programme

B.1 das veränderte Makefile

```
 ############################################################
#
#  Makefile for drsosc, drscl and drs_exam
#  executables under linux
#
#  Requires wxWidgets 2.8.9 or newer
#
############################################################


CFLAGS        = -g -O2 -Wall -Wuninitialized
                -fno-strict-aliasing -Iinclude
                -DOS_LINUX -DHAVE_LIBUSB
LIBS          = -lpthread -lutil -lusb


CPP_OBJ       = DRS.o ConfigDialog.o DOFrame.o DOScreen.o
                 DRSOsc.o MeasureDialog.o Measurement.o
                 Osci.o AboutDialog.o
OBJECTS       = musbstd.o mxml.o strlcpy.o

all: drs_exam


drs_exam: $(OBJECTS) DRS.o drs_exam.o
$(CXX) $(CFLAGS) $(OBJECTS) DRS.o drs_exam.o -o
          drs_exam $(LIBS)

main.o: src/main.cpp include/mxml.h include/DRS.h
$(CXX) $(CFLAGS) $(WXFLAGS) -c $<
```

```
drs_exam.o: src/drs_exam.cpp include/mxml.h include/DRS.h
$(CXX) $(CFLAGS) $(WXFLAGS) -c $<

$(CPP_OBJ): %.o: src/%.cpp include/%.h include/mxml.h include/DRS.h
$(CXX) $(CFLAGS) $(WXFLAGS) -c $<

$(OBJECTS): %.o: src/%.c include/mxml.h include/DRS.h
$(CXX) $(CFLAGS) -c $<

clean:
rm -f *.o
```

B.2 das Programm drs_exam_modif_cast_bin_2

```
/********************************************************************\

  Name:         drs_exam.cpp
  Created by:   Stefan Ritt

  Contents:     Simple example application to read out a DRS4
                evaluation board

  $Id: drs_exam.cpp 13482 2009-05-26 06:48:58Z ritt@PSI.CH $

\********************************************************************/


/* modified for use at HiSCORE by Robert Eichler */

#include <math.h>

#ifdef _MSC_VER

#include <windows.h>

#elif defined(OS_LINUX)

#define O_BINARY 0

#include <unistd.h>
#include <ctype.h>
#include <sys/ioctl.h>
#include <errno.h>
```

```
#define DIR_SEPARATOR '/'

#endif

//#define debugversion

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <iostream>
#include <fstream>
#include <string>
#include <sstream>
#include <time.h>



#include "strlcpy.h"
#include "DRS.h"

using namespace std;

/*------------------------------------------------------------------*/

int main()
{
   int nBoards;
   DRS *drs;
   DRSBoard *b;
   float time_array[1024];
   float wave_array[8][1024];
   FILE  *f = NULL;
   char buff1[100000];

   int i = 0;
   int a = 1;
   int filcount = 0;
   int j = 0;
   char filname[80];
   string line = "initial";
   float linval = 0;
   string line2;
   bool active = false;
   time_t starttime;
   time_t currenttime;
   int difference;
```

```
int oldevents = 0;
int newevents = 0;
float freq;

/* do initial scan */
drs = new DRS();

/* show any found board(s) */
for (i=0 ; i<drs->GetNumberOfBoards() ; i++) {
   b = drs->GetBoard(i);
   #ifdef debugversion
      printf("Found DRS4 evaluation board, serial #%d,
             firmware revision %d\n", b->GetBoardSerialNumber(),
             b->GetFirmwareVersion());
   #endif
}

/* exit if no board found */
nBoards = drs->GetNumberOfBoards();
if (nBoards == 0) {
   #ifdef debugversion
      printf("No DRS4 evaluation board found\n");
   #endif
   return 0;
}
else {
   ofstream fbfound ("/root/found_board.txt");
          // write a file to inform the Python interface
          // that a DRS board is connected to the PlugPC
   char foundb[10];
   sprintf(foundb, "board found\n");
   fbfound << foundb;
   fbfound.close();
}

/* continue working with first board only */
b = drs->GetBoard(0);

/* initialize board */
b->Init();

/* set sampling frequency */
b->SetFrequency(5);

/* enable transparent mode needed for analog trigger */
b->SetTranspMode(1);
```

```
/* set input range to -0.5V ... +0.5V */
b->SetInputRange(0);

/* use following line to set range to 0..1V */
//b->SetInputRange(0.5);

/* use following line to disable hardware trigger */
//b->EnableTrigger(0, 0);

/* use following line to enable external hardware trigger (Lemo) */
//b->EnableTrigger(1, 0);

/* use following lines to enable hardware trigger
   on CH1 at 50 mV negative edge */
b->EnableTrigger(0, 1);              // lemo off, analog trigger on
b->SetTriggerSource(0);              // use CH1 as source
b->SetTriggerLevel(-0.05, true);     // -0.05 V, negative edge
b->SetTriggerDelay(0);               // zero trigger delay


/* repeat while a = 1 */
while (a == 1) {

   /* if the program has been told to do event sampling*/
   if (active){
      /* open file to save waveforms */

      starttime = time(NULL);
                  // set the time when data taking starts
      #ifdef debugversion
         printf("setting starttime\n");
      #endif

      /* repeat until there's an internal break-command */
      while(1){
         i = j % 1000;
         /* if the last data file has been closed after 1000 events were
            written or the program has just been started, open a new one */
         if (i == 0){
            sprintf(filname,"/root/meas_data/data_%d.bin", filcount);
            #ifdef debugversion
               printf("opening %s\n",filname);
            #endif
        f=fopen(filname, "wb");
            setbuf(f, buff1);
         }
         currenttime = time(NULL); // set current time
```

```
difference = currenttime - starttime;
    // calculate total running time
    // of the current event sampling

/* clear chip (necessary for DRS4 to reduce noise) */
b->ClearCycle();

/* start board (activate domino wave) */
b->StartDomino();

/* wait for trigger */
#ifdef debugversion
   printf("Waiting for trigger...");
#endif
while (b->IsBusy()){
   currenttime = time(NULL);
   difference = currenttime - starttime;
   /* if at least 2 seconds have passed since the start
        of event sampling, abort the while(1) loop*/
   if(difference >= 2){
      break;
   }
   else{
   }

}

currenttime = time(NULL);
difference = currenttime - starttime;
/* if at least 2 seconds have passed since the start of
  event sampling */
if (difference >= 2){
   #ifdef debugversion
      printf("time elapsed\n");

      printf("checking for logfile\n");
   #endif
   fstream fcheck ;
   fcheck.open("/root/meas_data/evcount.txt");
                        //check for 'evcount' file
   if(fcheck.is_open()){
      #ifdef debugversion
         printf("logfile existing\n");
      #endif
      fcheck.close();
      remove("/root/meas_data/evcount.txt");
                       // if it exists: delete it
```

```
   }
   else{
      #ifdef debugversion
         printf("no logfile\n");
      #endif
   }

   ofstream fthree ("/root/meas_data/evcount.txt");
   #ifdef debugversion
      printf("creating logfile\n");
   #endif
   newevents = j - oldevents;
   freq = float(newevents)/2;  // calculate current event rate
   char evcount[20];
   sprintf(evcount, "%f\n",freq);
   fthree << evcount;      // write rate to file 'evcount.txt'
   fthree.close();
   #ifdef debugversion
      printf("logfile written\n");
   #endif
   oldevents = j;

   break;           // abort the while loop
}
else{
}

/* read all waveforms */
b->TransferWaves(0, 8);

/* read time (X) array in ns */
b->GetTime(0, time_array);

/* decode waveform (Y) array first channel in mV */
b->GetWave(0, 0, wave_array[0]);

/* decode waveform (Y) array second channel in mV
   Note: On the evaluation board input #1 is connected
   to channel 0 and 1 of the DRS chip, input #2 is
   connected to channel 2 and 3 and so on. So to
   get the input #2 we have to read DRS channel #2,
   not #1 */
b->GetWave(0, 2, wave_array[1]);

/* Save waveform: X=time_array[i], Yn=wave_array[n][i] */
//fprintf(f, "Event #%d: t y1 y2\n", j);
for (i=0 ; i<1024 ; i++){
```

```
            short wav0 = (short)wave_array[0][i];
            fwrite(&wav0,sizeof wav0, 1 ,f);
            short wav1 = (short)wave_array[1][i];
            fwrite(&wav1,sizeof wav1, 1 ,f);
         }

         /* print some progress indication */
         #ifdef debugversion
            printf("\rEvent #%d read successfully\n", j);
         #endif

         j++;
         i = j % 1000;

         /*if (i == 1){
            a = 0;
            break;
         }*/

         /* if 1000 events have been written to the
            current data file, close it */
         if (i == 0){
            fclose(f);
            #ifdef debugversion
               printf("closing file\n");
            #endif
            filcount = filcount + 1;

         }
         else{
         }

      }

   }

   /* if there's no event sampling, do nothing*/
   else{
      sleep(2);
   }


   /* open file 'commands.txt', which is written by
      the Python interface when it receives commands
      for DAQ */
   #ifdef debugversion
      printf("trying to open ftwo\n");
```

```
#endif
ifstream ftwo ("/root/commands.txt");
#ifdef debugversion
   printf("opening ftwo\n");
#endif
if (ftwo.is_open()){
   #ifdef debugversion
      printf("ftwo open\n");
   #endif
   while (! ftwo.eof()){
      #ifdef debugversion
         printf("reading ftwo command\n");
      #endif
      getline(ftwo,line);
      #ifdef debugversion
         printf("ftwo command read\n");
      #endif
      line2 = string(line);

      /* if there's a normal command in the file*/
      if (line2.compare("command")==0){
         getline(ftwo, line);
         line2 = string(line);

         if (line2.compare("quit")==0){
            a = 0;
                 // set a to 0 to abort while(a==1)
                 //loop and quit the program
         }
         else if(line2.compare("start")==0){
            active = true;          // start event sampling
            ofstream fact ("/root/active.txt");
                        // create file 'acive.txt' to inform
                        // the Python interface that event
                        // sampling starts
            #ifdef debugversion
               printf("starting\n");
            #endif
            char activa[15];
            sprintf(activa, "starting");
            fact << activa;
            fact.close();
         }
         else if(line2.compare("stop")==0){
            active = false;    // stop event sampling
            fstream fcheck2 ;
            fcheck2.open("/root/active.txt");
```

```
            if(fcheck2.is_open()){
               #ifdef debugversion
                  printf("stopping\n");
               #endif
               fcheck2.close();
               remove("/root/active.txt");
                    // remove file 'active.txt' to inform the
                    // Python interface that there's no event
                    // sampling
            }
            else{
               #ifdef debugversion
                  printf("not active\n");
               #endif
            }

         }
         else{
         }
      }
      /* if there's a new threshold value, just apply it */
      else if (line2.compare("setthr")==0){
         #ifdef debugversion
            printf("setting threshold\n");
         #endif
         getline(ftwo, line);
         line2 = string(line);
         stringstream sstr;
         sstr << line2;
         sstr >> linval;
         #ifdef debugversion
            printf("threshold set to %f \n", linval);
         #endif
         b->SetTriggerLevel(linval, true);
      }
      else{
         #ifdef debugversion
            printf("no command\n");
         #endif
      }

   }
   ftwo.close();
   remove("/root/commands.txt");
}

else{
```

```
      }
   }

   /* before quitting the program, remove 'evcount.txt' and
      'found_board.txt' to prevent that the Python interface
      gets any wrong information after the next reboot*/

   fstream fchecknumb ;
   fchecknumb.open("/root/meas_data/evcount.txt");
   if(fchecknumb.is_open()){
      #ifdef debugversion
         printf("countfile existing\n");
      #endif
      fchecknumb.close();
      remove("/root/meas_data/evcount.txt");
   }
   else{
      #ifdef debugversion
         printf("no countfile\n");
      #endif
   }



   fstream fcheckbf ;
   fcheckbf.open("/root/found_board.txt");
   if(fcheckbf.is_open()){
      #ifdef debugversion
         printf("found_board existing\n");
      #endif
      fcheckbf.close();
      remove("/root/found_board.txt");
   }
   else{
      #ifdef debugversion
         printf("no board found\n");
      #endif
   }


   /* delete DRS object -> close USB connection */
   delete drs;
}
```

Anhang C

GUI-Komponenten

C.1 Interface

```
''' This program is an interface that will run
    on a PlugPC and directly communicate with
    the Arduino and the C++ program for DAQ
    It will receive its orders from a Python GUI
    on the central station computer '''


import socket

from string import *

import serial

import os

import glob

host = ''     # this program is used as server for the TCP/IP connection
port = 50007   # the port chosen mustn't be used by other processes
               # (should be >50000)

s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, port))
s.listen(1)     # the program is waiting for a client to connect

conn, addr = s.accept()

ser = serial.Serial('/dev/ttyACM0', 9600, timeout = 100)
            # the connection for Arduino

a=1             # set runtime variable a to 1
```

```
evcount=0

activedt=0

while a:      # as long as a is true (a==1) the program will be running
    income=conn.recv(1024)  # the program stays halted until it receives
                            # a message from the GUI

    ''' The string received via TCP/IP will be split up and the components
        are put into an array
        The array's first component will contain the main command while
        the second component holds an additional argument if needed '''

    income=str(income)
    incoming=income.split("$")


    ''' After reading the primary command, the program will open the file
        'commands.txt'
    This file contains all commands that are destined for the DAQ program'''

    if incoming[0]=="quit":
        a=0
        if (os.path.isfile("/root/commands.txt")):
            writefile=open("/root/commands.txt", "a")
                        # if the file exists: new command
        else:
            writefile=open("/root/commands.txt", "w") # else: create the file
        wlines="command\nquit\n"                      # write the command
        writefile.write(wlines)
        writefile.close()
    elif incoming[0]=="start":
        if (os.path.isfile("/root/commands.txt")):
            writefile=open("/root/commands.txt", "a")
        else:
            writefile=open("/root/commands.txt", "w")
        wlines="command\nstart\n"
        writefile.write(wlines)
        writefile.close()
    elif incoming[0]=="stop":
        if (os.path.isfile("/root/commands.txt")):
            writefile=open("/root/commands.txt", "a")
        else:
            writefile=open("/root/commands.txt", "w")
        wlines="command\nstop\n"
        writefile.write(wlines)
```

```
        writefile.close()
    elif incoming[0]=="enter":
        if (os.path.isfile("/root/commands.txt")):
            writefile=open("/root/commands.txt", "a")
        else:
            writefile=open("/root/commands.txt", "w")
        wlines="setthr\n"+incoming[1]+"\n"
                        # if the command is to set a new threshold:
                    # send the float stored in the array's second component
        writefile.write(wlines)
        writefile.close()

    ''' If the command is a query for telemetry data, the interface will
        transmit it to the Arduino and then save the incoming slow control
        data as a string.
        Additionally, the interface will access the files where the
        DAQ program saves its telemetry data '''
    elif incoming[0] == "q":
        print "query received"
        ser.open()
        ser.write('q')
        print "sending to arduino"
        reader=str(ser.readline())
        print "got data"
        print reader
if (os.path.isfile("/root/found_board.txt")): # if theres a DRS4 board:
            if (os.path.isfile("/root/active.txt")):
                            # check if it is currently sampling events
                activedt="1"
                if (os.path.isfile("/root/meas_data/evcount.txt")):
                    evcountfile=open("/root/meas_data/evcount.txt","r")
                        # if this file exists: open it and
                        # read current event rate
                    evcount=str(evcountfile.readline())
                    evcountfile.close()
                    os.remove(evcountfile)   # delete data after reading
                else:
                    evcount = "0"  # else: event rate is 0

            else:
                activedt="0"
                evcount = "0"
else:
    activedt='2'         # else: report missing board
            evcount = '0'
        linetowrite=evcount+"$"+reader+"$"+str(activedt)
                            # write telemetry data to string
```

```
        print linetowrite
        conn.send(linetowrite)            # send it to the GUI
        ser.close()


    else:
        ser.open()
        ser.write(str(income))
        dummy6=str(ser.readline())
                  # the Arduino will automatically send an empty line
        ser.close()
s.close()
conn.close()
```

C.2 GUI

```
from string import *
import time
from time import gmtime, strftime, localtime
from Tkinter import *
import Pmw
import socket
import string

host='192.168.1.1'   #designated host(PlugPC)
port=50007           #designated port
s=socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((host, port))   #connect to server

v=0

root=Tk()

class SlowControl():

    def __init__(self, master):


        ''' notebook1 will contain textfields to display
            current setting and telemetry data transmitted
            by detector stations'''

        self.frame1=Frame(master, width=1000, height=1000)
        self.frame1.pack(side=LEFT, fill='both')
```

```
self.notebook1=Pmw.NoteBook(self.frame1, tabpos=None)
self.notebook1.pack(fill='both', expand=1, padx=5, pady=5)

self.page1_1=self.notebook1.add('data')
                            # add tab 'data' to notebook1
#self.notebook1.tab('data').focus_set()
   # only necessary if notebook1 has multiple
   # tabs (tabpos=None removed)

self.group1_1_1=Pmw.Group(self.page1_1, tag_text='Time')
              #add group 'Time' which contains the timefield
self.group1_1_1.pack(fill='x', expand=1, padx=5, pady=1)

self.text1_1_1_1=Text(self.group1_1_1.interior(), height=1, width=25)
self.text1_1_1_1.grid(row=0, column=0)


self.group1_1_2=Pmw.Group(self.page1_1, tag_text='Enviromental Data')
     # this group contains data, such as temperature and maybe humidity
self.group1_1_2.pack(fill='x', expand=1, padx=5, pady=5)

self.label1_1_2_1=Label(self.group1_1_2.interior(),
                          text='Temperature in C')
self.label1_1_2_1.grid(row=0, column=0)

self.text1_1_2_1=Text(self.group1_1_2.interior(), height=1, width=15)
self.text1_1_2_1.grid(row=0, column=1)


self.group1_1_3=Pmw.Group(self.page1_1, tag_text='Voltage system')
                       # everything abot the HV system stands here
self.group1_1_3.pack(fill='x', expand='yes', padx=10, pady=10)

self.frame1_1_3a=Frame(self.group1_1_3.interior())
self.frame1_1_3a.grid(row=0, column=0)

self.label1_1_3a_1=Label(self.frame1_1_3a,
                        text='green if powering HV is allowed')
self.label1_1_3a_1.grid(row=0, column=0)

self.canv1_1_3a_1=Canvas(self.frame1_1_3a, height=35, width=40)
self.canv1_1_3a_1.grid(row=0, column=1)
self.circ1_1_3a_1=self.canv1_1_3a_1.create_oval
                                     (5, 3, 35, 33, fill='black')

self.label1_1_3a_2=Label(self.frame1_1_3a,
                           text='green if HV is on')
```

```
self.label1_1_3a_2.grid(row=1, column=0)

self.canv1_1_3a_2=Canvas(self.frame1_1_3a, height=35, width=40)
self.canv1_1_3a_2.grid(row=1, column=1)
self.circ1_1_3a_2=self.canv1_1_3a_2.create_oval
                              (5, 3, 35, 33, fill='black')


self.frame1_1_3=Frame(self.group1_1_3.interior(), width=100)
self.frame1_1_3.grid(row=1, column=0)

self.label1_1_3_1=Label(self.frame1_1_3,text='HV in Volt:')
self.label1_1_3_1.grid(row=1, column=0, sticky=W)

self.text1_1_3_1=Text(self.frame1_1_3, height=1, width=15)
self.text1_1_3_1.grid(row=1, column=1)

self.label1_1_3_2=Label(self.frame1_1_3, text='Current in uA:')
self.label1_1_3_2.grid(row=2, column=0, sticky=W)

self.text1_1_3_2=Text(self.frame1_1_3, height=1, width=15)
self.text1_1_3_2.grid(row=2, column=1)

self.label1_1_3_3=Label(self.frame1_1_3,
                        text='Battery Voltage in Volt:')
self.label1_1_3_3.grid(row=3, column=0, sticky=W)

self.text1_1_3_3=Text(self.frame1_1_3, height=1, width=15)
self.text1_1_3_3.grid(row=3, column=1)

self.label1_1_3_4=Label(self.frame1_1_3,
                        text='Current readout calibration:')
self.label1_1_3_4.grid(row=4, column=0, sticky=W)

self.text1_1_3_4=Text(self.frame1_1_3, height=1, width=15)
self.text1_1_3_4.grid(row=4, column=1)

self.label1_1_3_5=Label(self.frame1_1_3,
                        text='Current entry calibration:')
self.label1_1_3_5.grid(row=5, column=0, sticky=W)

self.text1_1_3_5=Text(self.frame1_1_3, height=1, width=15)
self.text1_1_3_5.grid(row=5, column=1)

self.group1_1_4=Pmw.Group(self.page1_1, tag_text='Lid System')
                          # group for lid system telemetry
self.group1_1_4.pack(fill='x', expand=1, padx=10, pady=10)
```

```
self.frame1_1_4=Frame(self.group1_1_4.interior())
self.frame1_1_4.grid(row=0, column=0)

self.label1_1_4_1=Label(self.frame1_1_4,
                        text='green if opening is allowed:')
self.label1_1_4_1.grid(row=0, column=0)

self.canv1_1_4_1=Canvas(self.frame1_1_4, height=35, width=40)
self.canv1_1_4_1.grid(row=0, column=1)
self.circ1_1_4_1=self.canv1_1_4_1.create_oval
                                  (5,3,35,33, fill='black')

self.label1_1_4_2=Label(self.frame1_1_4, text='lid state:')
self.label1_1_4_2.grid(row=1, column=0)

self.canv1_1_4_2=Canvas(self.frame1_1_4, height=35, width=40)
self.canv1_1_4_2.grid(row=1, column=1)

self.line1_1_4_2_1=self.canv1_1_4_2.create_line
                (19, 31,38,31, fill='black', width=2)

self.line1_1_4_2_2=self.canv1_1_4_2.create_line
                (19, 31,19,5, fill='black', width=2)

self.line1_1_4_2_3=self.canv1_1_4_2.create_line
                 (38,31 ,38,5, fill='black', width=2)

self.lidline=self.canv1_1_4_2.create_line
                 (19,5,38,5, fill="red", width=2)
self.group1_1_5=Pmw.Group(self.page1_1,
                          tag_text='current settings')
                   # general settings for Slow Control
self.group1_1_5.pack(fill='x', expand=1, padx=10, pady=10)

self.frame1_1_5=Frame(self.group1_1_5.interior())
self.frame1_1_5.grid(row=0, column=0)

self.label1_1_5_1=Label(self.frame1_1_5,
                        text='data taking interval:')
self.label1_1_5_1.grid(row=0, column=0, sticky=W)

self.text1_1_5_1=Text(self.frame1_1_5, height=1, width=15)
self.text1_1_5_1.grid(row=0, column=1)

self.label1_1_5_2=Label(self.frame1_1_5,
                        text='size of save array:')
```

```
self.label1_1_5_2.grid(row=1, column=0, sticky=W)

self.text1_1_5_2=Text(self.frame1_1_5, height=1,width=15)
self.text1_1_5_2.grid(row=1, column=1)


self.group1_1_6=Pmw.Group(self.page1_1, tag_text='DRS4')
                       # DAQ settings and telemetry
self.group1_1_6.pack(fill='x', expand=1, padx=10, pady=10)

self.frame1_1_6=Frame(self.group1_1_6.interior())
self.frame1_1_6.grid(row=0, column=0)

self.label1_1_6_1a=Label(self.frame1_1_6,
                         text='green if DRS4 is taking data:')
self.label1_1_6_1a.grid(row=0, column=0)

self.canv1_1_6_1=Canvas(self.frame1_1_6, height=35, width=40)
self.canv1_1_6_1.grid(row=0, column=1)
self.circ1_1_6_1=self.canv1_1_6_1.create_oval
                       (5,3,35,33, fill='black')


self.label1_1_6_1=Label(self.frame1_1_6,
                         text='Events per second:')
self.label1_1_6_1.grid(row=1, column=0, sticky=W)

self.text1_1_6_1=Text(self.frame1_1_6, height=1, width=15)
self.text1_1_6_1.grid(row=1, column=1)

self.label1_1_6_2=Label(self.frame1_1_6,
                         text='Threshold in Volt:')
self.label1_1_6_2.grid(row=2, column=0, sticky=W)

self.text1_1_6_2=Text(self.frame1_1_6, height=1, width=15)
self.text1_1_6_2.grid(row=2, column=1)


''' notebook2 has multiple tabs which contain
    the controls to change Slow Control and
    DAQ settings '''

self.frame2=Frame(master)
self.frame2.pack(side=RIGHT, fill='both')

self.notebook2=Pmw.NoteBook(self.frame2)
self.notebook2.pack(fill='both', expand=1, padx=5, pady=5)
```

```
self.page2_1=self.notebook2.add('Main Control')
    # the 'Main Control' tab contains
    # all the GUI's main settings
self.notebook2.tab('Main Control').focus_set()

self.group2_1_1=Pmw.Group(self.page2_1)
self.group2_1_1.pack(fill='x', expand=1, padx=10, pady=10)

self.frame2_1_1=Frame(self.group2_1_1.interior())
self.frame2_1_1.grid(row=0, column=0)

self.label2_1_1_1a=Label(self.frame2_1_1,
                         text='change length of')
self.label2_1_1_1a.grid(row=0, column=0)

self.label2_1_1_1b=Label(self.frame2_1_1, text='save array')
self.label2_1_1_1b.grid(row=0, column=1, sticky=W)

self.text2_1_1_1=Text(self.frame2_1_1, height=1, width=15)
self.text2_1_1_1.grid(row=1, column=0)

self.lenbut=Button(self.frame2_1_1, text='apply',
                   command=self.chanlen)
self.lenbut.grid(row=1, column=1)

self.label2_1_1_2a=Label(self.frame2_1_1,
                         text='change data taking')
self.label2_1_1_2a.grid(row=2, column=0)

self.label2_1_1_2b=Label(self.frame2_1_1,
                         text='interval (float)')
self.label2_1_1_2b.grid(row=2, column=1, sticky=W)

self.text2_1_1_2=Text(self.frame2_1_1, height=1, width=15)
self.text2_1_1_2.grid(row=3, column=0)

self.intervbutton=Button(self.frame2_1_1, text='apply',
                         command=self.applyinterv)
self.intervbutton.grid(row=3, column=1)

self.quitbutton=Button(self.frame2_1_1, text='Save and Quit',
                       command=root.quit)
self.quitbutton.bind("<Button-1>", self.saves)
self.quitbutton.grid(row=4, column=0)
```

```
self.page2_2=self.notebook2.add('HV Control')

self.group2_2_1=Pmw.Group(self.page2_2,
                          tag_text='general control')
self.group2_2_1.pack(fill='x', expand=1, padx=10, pady=10)

self.frame2_2_1=Frame(self.group2_2_1.interior())
self.frame2_2_1.grid(row=0, column=0)

self.hvonbutton=Button(self.frame2_2_1, text='HV on',
                       command=self.hvon)
self.hvonbutton.grid(row=0, column=0)

self.hvoffbutton=Button(self.frame2_2_1, text='HV off',
                        command=self.hvoff)
self.hvoffbutton.grid(row=0, column=1)

self.label2_2_1_1=Label(self.frame2_2_1,
                        text='change HV via Scale:')
self.label2_2_1_1.grid(row=1, column=0, sticky=W)

self.hvscale=Scale(self.frame2_2_1, from_=0,
                   to=2000, showvalue='yes',
                           orient='horizontal')
self.hvscale.grid(row=1, column=1, sticky=W)

self.scalebutton=Button(self.frame2_2_1, text='apply',
                        command=self.applyscale)
self.scalebutton.grid(row=1, column=2)

self.label2_2_1_2=Label(self.frame2_2_1,
                       text='change HV via entry:')
self.label2_2_1_2.grid(row=2, column=0, sticky=W)

self.text2_2_1_2=Text(self.frame2_2_1, height=1, width=15)
self.text2_2_1_2.grid(row=2, column=1)

self.hvbutton=Button(self.frame2_2_1, text='apply',
                     command=self.applyhv)
self.hvbutton.grid(row=2, column=2)


self.group2_2_2=Pmw.Group(self.page2_2, tag_text='calibration')
self.group2_2_2.pack(fill='x', expand=1, padx=10, pady=10)

self.frame2_2_2=Frame(self.group2_2_2.interior())
self.frame2_2_2.grid(row=0, column=0)
```

```
self.label2_2_2_1a=Label(self.frame2_2_2,
                         text='Calibrate read out')
self.label2_2_2_1a.grid(row=0, column=0)

self.label2_2_2_1b=Label(self.frame2_2_2,
                 text='(Voltage corresponding to 5V input)')
self.label2_2_2_1b.grid(row=0, column=1)

self.text2_2_2_1=Text(self.frame2_2_2, height=1, width=15)
self.text2_2_2_1.grid(row=1, column=0)

self.hvreadbutton=Button(self.frame2_2_2, text='apply',
                          command=self.applyreadout)
self.hvreadbutton.grid(row=1, column=1)

self.label2_2_2_2a=Label(self.frame2_2_2,
                         text='calibrate entry')
self.label2_2_2_2a.grid(row=2, column=0)

self.label2_2_2_2b=Label(self.frame2_2_2,
                 text='(Voltage corresponding to 5V output)')
self.label2_2_2_2b.grid(row=2, column=1)

self.text2_2_2_2=Text(self.frame2_2_2, height=1, width=15)
self.text2_2_2_2.grid(row=3, column=0)

self.hventrybutton=Button(self.frame2_2_2, text='apply',
                            command=self.applyentry)
self.hventrybutton.grid(row=3, column=1)

self.label2_2_2_3a=Label(self.frame2_2_2, text='set light threshold')
self.label2_2_2_3a.grid(row=4, column=0)

self.text2_2_2_3=Text(self.frame2_2_2, height=1, width=15)
self.text2_2_2_3.grid(row=5, column=0)

self.hvthreshbutton=Button(self.frame2_2_2, text='apply',
                           command=self.sethvthr)
self.hvthreshbutton.grid(row=5, column=1)


self.page2_3=self.notebook2.add('Lid Control')

self.group2_3_1=Pmw.Group(self.page2_3, tag_text='Motor Control')
self.group2_3_1.pack(fill='x', expand=1, padx=10, pady=10)
```

```
self.frame2_3_1=Frame(self.group2_3_1.interior())
self.frame2_3_1.grid(row=0, column=0)

self.openbutton=Button(self.frame2_3_1, text='Open',
                       command=self.openlid, width=15)
self.openbutton.grid(row=0, column=0)

self.closebutton=Button(self.frame2_3_1, text='Close',
                       command=self.closelid, width=15)
self.closebutton.grid(row=0, column=1)


self.group2_3_2=Pmw.Group(self.page2_3, tag_text='Magnet Control')
self.group2_3_2.pack(fill='x', expand=1, padx=10, pady=10)

self.frame2_3_2=Frame(self.group2_3_2.interior())
self.frame2_3_2.grid(row=0, column=0)

self.magonbutton=Button(self.frame2_3_2, text='On',
                        command=self.magon, width=15)
self.magonbutton.grid(row=0, column=0)

self.magoffbutton=Button(self.frame2_3_2, text='Off',
                         command=self.magoff, width=15)
self.magoffbutton.grid(row=0, column=1)


self.group2_3_3=Pmw.Group(self.page2_3, tag_text='Calibration')
self.group2_3_3.pack(fill='x', expand=1, padx=10, pady=10)

self.frame2_3_3=Frame(self.group2_3_3.interior())
self.frame2_3_3.grid(row=0, column=0)

self.label2_3_3_1a=Label(self.frame2_3_3,
                          text='set new light threshold')
self.label2_3_3_1a.grid(row=0, column=0)

self.text2_3_3_1=Text(self.frame2_3_3, height=1, width=15)
self.text2_3_3_1.grid(row=1, column=0)

self.thrbutton=Button(self.frame2_3_3, text='apply',
                                    command=self.setthr)
self.thrbutton.grid(row=1, column=1)


self.page2_4=self.notebook2.add('DRS4')
```

```
self.group2_4_1=Pmw.Group(self.page2_4)
self.group2_4_1.pack(fill='x', expand=1, padx=10, pady=10)

self.frame2_4_1=Frame(self.group2_4_1.interior())
self.frame2_4_1.grid(row=0, column=0)

self.startbutton=Button(self.frame2_4_1, text='Start',
                        command=self.startdt)
self.startbutton.grid(row=0, column=0)

self.stopbutton=Button(self.frame2_4_1, text='Stop',
                       command=self.stopdt)
self.stopbutton.grid(row=0, column=1)

self.label2_4_1_1a=Label(self.frame2_4_1, text='Set new threshold')
self.label2_4_1_1a.grid(row=1, column=0)

self.label2_4_1_1b=Label(self.frame2_4_1, text='in Volt')
self.label2_4_1_1b.grid(row=1, column=1)

self.text2_4_1_1=Text(self.frame2_4_1, height=1, width=15)
self.text2_4_1_1.grid(row=2, column=0)

self.dtthreshbutton=Button(self.frame2_4_1, text='apply',
                           command=self.enterdtthresh)
self.dtthreshbutton.grid(row=2, column=1)


self.timestamp=time.time()
self.interv=3.67  # this is the interval in which the GUI will send
                  # data requests to the PlugPC
                  # (default: 3.67s + ~1,33s delay =~5s)
self.income=(0,0,0,0,0,0,0,0,0,0,0)
self.readout=2000 # readout calibration: the HV-value corresponding
                  # to 5V input to the Arduino (default 2000V)
self.entry=2000   # entry calibration: the HV-value corresponding
                  # to 5V output from the Arduino (default 2000V)
self.arlen=100    # number of data packages that are saved into
                  # the same file (default 100)
self.lastset=time.time()
self.currentset=time.time()
self.evpersec=0    # events per second measured by the DRS4 chip
self.takingdata=0  # 1 if DRS4 is taking data, 0 if not
self.lidstate='c'  # opening status of lid covers:
                    # 'c'=closed, 'o'=open, 'n'=nn
self.hvstate='l'    # HV status: 'l'=low, 'h'=high
self.canopen='n'  # n if opening the lids is prohibited by too high
```

```
                    # light intensity, y if opening is allowed
    self.canpower='n' # n if powering the HV system is prohibited by
                    # too high light intensity,
                     # y if powering is allowed
    self.magstate='l'  # l if electric magnet is off, h if it is on
    self.dtthresh=-0.05 # threshold for peak height in
                     # DRS4 data taking, default=-50mV
    self.i=0            # internal counter for the save files
    self.savearr=[]
    self.lastline="abcdefghijklmn"

    self.scoreloop()    # run function 'scoreloop'

def scoreloop(self):

    if (time.time()-self.timestamp) >= self.interv:
                     # check if enough time has passed since
                     # the last query for telemetry data

        s.send("q")     # send a query to the PlugPC
        print "q"
        self.incoming=s.recv(1024)    # wait for data string
        self.currentset=time.time()
        self.income=str(self.incoming).split("$")
               # split the received string into its components
        if len(self.income) == 13:
            self.lidstate=str(self.income[5])
                            # assign the array's elements
                            # to the corresponding variables
            self.hvstate=str(self.income[6])
            self.canopen=str(self.income[7])
            self.canpower=str(self.income[8])
            self.magstate=str(self.income[9])
            self.takingdata=int(self.income[12])
            self.text1_1_1_1.delete(1.0, END)
            self.text1_1_1_1.insert(END,
                      strftime("%a,%d %b %Y %H:%M:%S",localtime()))
           # insert the variables into the corresponding data fields
            self.text1_1_2_1.delete(1.0, END)
            self.text1_1_2_1.insert(END, self.income[4])
            self.text1_1_3_1.delete(1.0, END)
            self.text1_1_3_1.insert(END, self.income[1])
            self.text1_1_3_2.delete(1.0, END)
            self.text1_1_3_2.insert(END, self.income[2])
            self.text1_1_3_3.delete(1.0, END)
            self.text1_1_3_3.insert(END, self.income[3])
            self.text1_1_3_4.delete(1.0, END)
```

```
            self.text1_1_3_4.insert(END, self.readout)
            self.text1_1_3_5.delete(1.0, END)
            self.text1_1_3_5.insert(END, self.entry)
            self.intervread=self.interv+1.33
            self.text1_1_5_1.delete(1.0, END)
            self.text1_1_5_1.insert(END, self.intervread)
            self.text1_1_5_2.delete(1.0, END)
            self.text1_1_5_2.insert(END, self.arlen)
            self.evpersec=int(self.income[0])
            self.text1_1_6_1.delete(1.0, END)
            self.text1_1_6_1.insert(END, self.evpersec)
            self.lastset=self.currentset
            self.text1_1_6_2.delete(1.0, END)
            self.text1_1_6_2.insert(END, self.dtthresh)
            self.income.pop(11)
            self.lastline1=string.join(self.income, "$")
            self.svln=strftime("%a,%d_%b_%Y_%H:%M:%S",localtime())+"$"
                                        +self.lastline1 + "\n"
            self.savearr.append(self.svln)
            self.timestamp=time.time()

    if len(self.savearr)>=self.arlen:  # when the savearray has
                                       # reached its maximum length,
                                       # save the data to a new file
                                       # and empty the savearray
        self.filename='slowcont'+str(self.i)
        self.FILE=open(self.filename, "w")
        self.FILE.writelines(self.savearr)
        self.savearr=[]
        self.i=self.i+1
        self.FILE.close()

    if self.canpower=='y':  # read 'canpower' variable and draw an oval
                            # in the color defined for that value
        self.canv1_1_3a_1.delete(self.circ1_1_3a_1)
        self.circ1_1_3a_1=self.canv1_1_3a_1.create_oval
                                (5,3,35,33, fill='green')
    elif self.canpower=='n':
        self.canv1_1_3a_1.delete(self.circ1_1_3a_1)
        self.circ1_1_3a_1=self.canv1_1_3a_1.create_oval
                                    (5,3,35,33, fill='red')
    else:
        self.canv1_1_3a_1.delete(self.circ1_1_3a_1)
        self.circ1_1_3a_1=self.canv1_1_3a_1.create_oval
                                    (5,3,35,33, fill='black')

    if self.hvstate=='h': # read 'hvstate' variable and draw an oval
```

```
                    # in the color defined for that value
    self.canv1_1_3a_2.delete(self.circ1_1_3a_2)
    self.circ1_1_3a_2=self.canv1_1_3a_2.create_oval
                                  (5,3,35,33, fill='green')
elif self.hvstate=='l':
    self.canv1_1_3a_2.delete(self.circ1_1_3a_2)
    self.circ1_1_3a_2=self.canv1_1_3a_2.create_oval
                                   (5,3,35,33, fill='red')
else:
    self.canv1_1_3a_2.delete(self.circ1_1_3a_2)
    self.circ1_1_3a_2=self.canv1_1_3a_2.create_oval
                                    (5,3,35,33, fill='black')

if self.canopen=='y':     # read 'canopen' variable and draw an
                    #  oval in the color defined for that value
    self.canv1_1_4_1.delete(self.circ1_1_4_1)
    self.circ1_1_4_1=self.canv1_1_4_1.create_oval
                                  (5,3,35,33, fill='green')
elif self.canpower=='n':
    self.canv1_1_4_1.delete(self.circ1_1_4_1)
    self.circ1_1_4_1=self.canv1_1_4_1.create_oval
                                   (5,3,35,33, fill='red')
else:
    self.canv1_1_4_1.delete(self.circ1_1_4_1)
    self.circ1_1_4_1=self.canv1_1_4_1.create_oval
                                    (5,3,35,33, fill='black')


if self.lidstate=='o':  # read 'lidstate' variable and change the
                        # status figure on canv1_1_4_2
    self.canv1_1_4_2.delete(self.lidline)
    self.lidline=self.canv1_1_4_2.create_line
                                    (0,5,19,5,fill="red", width=2)
                                  #if lid is complitely open
elif self.lidstate=='n':
    self.canv1_1_4_2.delete(self.lidline)
    self.lidline=self.canv1_1_4_2.create_line
                                    (10,5,29,5,fill="red", width=2)
                                  #if it's undefined (n.n.)
elif self.lidstate=='c':
    self.canv1_1_4_2.delete(self.lidline)
    self.lidline=self.canv1_1_4_2.create_line
                                   (19,5,38,5,fill="red", width=2)
                                   #if it's closed
else:
    self.canv1_1_4_2.delete(self.lidline)   #error: blank
```

```
    if self.takingdata==1:    # read 'takingdata' variable and draw an
                              # oval in the color defined for that value
        self.canv1_1_6_1.delete(self.circ1_1_6_1)
        self.circ1_1_6_1=self.canv1_1_6_1.create_oval
                                        (5,3,35,33, fill='green')
    elif self.takingdata==0:
        self.canv1_1_6_1.delete(self.circ1_1_6_1)
        self.circ1_1_6_1=self.canv1_1_6_1.create_oval
                                        (5,3,35,33, fill='red')
    else :
        self.canv1_1_6_1.delete(self.circ1_1_6_1)
        self.circ1_1_6_1=self.canv1_1_6_1.create_oval
                                        (5,3,35,33, fill='black')




    self.frame1.after(500, self.scoreloop)
                        # wait for 500 ms and then run 'scoreloop' again

def chanlen(self):    # apply change to the length of the savearray
    self.arlen=atoi(self.text2_1_1_1.get(1.0, END))
    print self.arlen
    self.text2_1_1_1.delete(1.0, END)

def applyinterv(self):  # apply entered change to the data taking interval
    self.interv=(float(self.text2_1_1_2.get(1.0, END))-1.33)
    print 'interv'+str(self.interv)
    self.text2_1_1_2.delete(1.0, END)

def saves(self, event):      # save current savearray and close socket
                             # connection to the PlugPC
    self.filename='slowcont'+str(self.i)
    self.FILE=open(self.filename, "w")
    self.FILE.writelines(self.savearr)
    self.savearr=[]
    self.i=self.i+1
    s.send("quit")
    s.close()

def hvon(self):                    # send command to power HV
    s.send("h")
    print 'h'

def hvoff(self):                 # send command to switch HV off
```

```
        s.send("l")
        print 'l'

    def applyscale(self):       # set HV to a value entered on the scale
        self.sethv=self.hvscale.get()
        print 'a'+str(self.sethv)
        s.send("a"+str(self.sethv))

    def applyhv(self):       # set the scale to show the
                             # value entered and set HV to it
        self.v=str(self.text2_2_1_2.get(1.0,END))
        self.text2_2_1_2.delete(1.0, END)
        self.hvscale.set(self.v)
        self.sethv=self.hvscale.get()
        print 'a'+str(self.sethv)
        s.send("a"+str(self.sethv))

    ''' note: the program will set the scale and then take the scale
        value to send it to the PlugPC this looks complicated but it
        is necessary because otherwise the GUI would set the number
        entered into the text field followed by an empty line which
        the script on the PlugPC might take as another change and set
        HV to 0'''

    def applyreadout(self):              # apply new readout calibration
        self.readout=str(self.text2_2_2_1.get(1.0, END))
        self.text2_2_2_1.delete(1.0, END)
        print 'v'+str(self.readout)
        s.send ("v"+self.readout)

    def applyentry(self):                # apply new entrycalibration
        self.entry=str(self.text2_2_2_2.get(1.0, END))
        self.text2_2_2_2.delete(1.0, END)
        print 'w'+str(self.entry)
        s.send("w"+str(self.entry))

    def sethvthr(self):       # set new light threshold for the HV system
        self.hvthresh=str(self.text2_2_2_3.get(1.0, END))
        self.text2_2_2_3.delete(1.0, END)
        print 'm'+str(self.hvthresh)
        s.send("m"+str(self.hvthresh))

    def openlid(self):               # open the lid
        print 'o'
        s.send("o")

    def closelid(self):        # close the lid
```

```
        print 'c'
        s.send("c")

    def magon(self):          # switch magnet on
        print 'e'
        s.send("e")

    def magoff(self):       # switch magnet off
        print 'f'
        s.send("f")

    def setthr(self):      # set new light threshold for the lid system
        self.lidthresh = str(self.text2_3_3_1.get(1.0, END))
        self.text2_3_3_1.delete(1.0, END)
        print 't'+str(self.lidthresh)
        s.send("t"+str(self.lidthresh))

    def startdt(self):   # start event sampling with DRS4
        print 'start'
        s.send("start")

    def stopdt(self):   # stop event sampling
        print 'stop'
        s.send("stop")

    def enterdtthresh(self): # enter new threshold for event sampling
        self.dtthresh=str(self.text2_4_1_1.get(1.0, END))
        s.send("enter$"+self.dtthresh)
        self.text2_4_1_1.delete(1.0, END)
        print 'enter '+self.dtthresh

sloco=SlowControl(root)
root.mainloop()
```

Danksagung

Ich möchte meinen Eltern danken, ohne deren fortwährende Unterstützung es mir nicht möglich gewesen wäre, mein Studium erfolgreich zu beenden. Auch meinem Bruder sei für seinen moralischen Beistand gedankt.

Weiterhin möchte ich Herrn Professor Dr. Dieter Horns dafür danken, dass er mir die Möglichkeit gab, diese Diplomarbeit in seiner Arbeitsgruppe abzulegen. Für seine Anleitung und Hilfe, sowie das wiederholte Korrektulesen meiner Diplomarbeit möchte ich mich auch bei Herrn Dr. Martin Tluczykont bedanken, ohne den diese Arbeit nicht möglich gewesen wäre. Mein Dank gilt auch den anderen Mitgliedern der HiSCORE-Gruppe: Daniel Hampf, Rayk Nachtigall und Dr. Tanja Keiske.

Zu guter letzt möchte ich mich bei allen anderen Mitgliedern der Arbeitsgruppe für experimentelle Astroteilchenphysik bedanken (ohne besondere Reihenfolge): Dr. Martin Raue, Milton Fernandes, Manuel Meyer, Andreas Maurer, Nelly Nguyen, Max Kastendieck, Prof. Dr. Götz Heinzelmann, Hannes Zechlin, Björn Opitz, Attila Abramowski, Jan Dreyling-Eschweiler, Wiebke Schubotz, Alexander Gewering-Peine und allen anderen, die ich eventuell vergessen habe, namentlich zu nennen.

Natürlich möchte ich mich auch bei der IT-Abteilung des DESY bedanken. Mein besonderer Dank gilt heirbei Herrn Rainer Peter Feller, der mir immer wieder mit Rat und Tat zur Seite stand und mir bei der Arbeit mit den PlugPCs eine große Unterstützung war.

Erklärung

Hiermit versichere ich, dass ich die Diplomarbeit selbständig und lediglich unter Benutzung der angegebenen Quellen und Hilfsmittel verfasst habe. Mit einer universitätsinternen Veröffentlichung bin ich einverstanden.

Hamburg, den Robert Eichler