Projekt „Haptical Interfaces“

Unterprojekt „Force-Feedback-Trackball“

Lehrstuhl für Virtuelle Realität
Fakultät Medien
Bauhaus-Universität Weimar

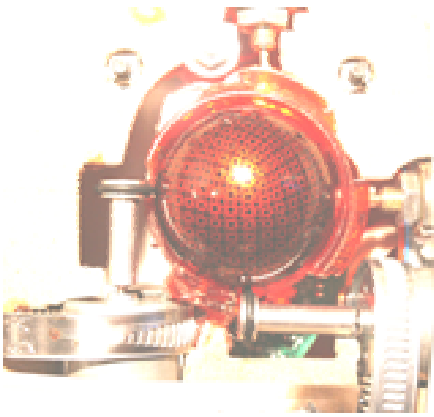
Projektbetreuer:

Prof. Dr. Bernd Fröhlich
Diplom-Inf. (FH) Jan Hochstrate

Projektteilnehmer:

Benjamin Brombach (benjamin.brombach@medien.uni-weimar.de)
Onno Haak (onno.haak@medien.uni-weimar.de)

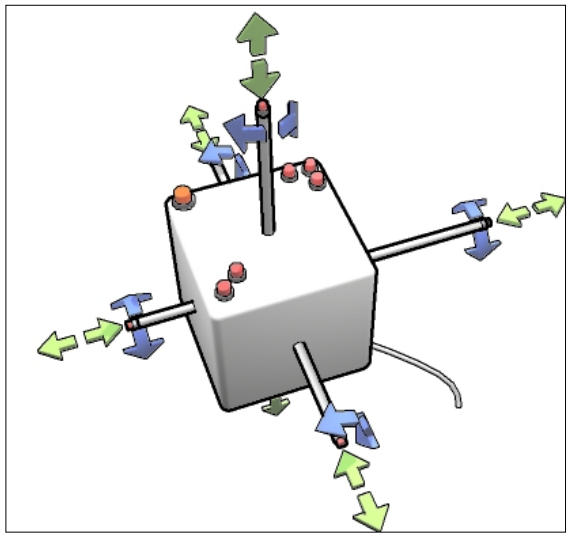
Inhalt



1. Einleitung

1.1. Intention des Projekts

Ziel des Forschungsprojekts im Wintersemester 2003/2004 war das Kennenlernen von Mikrokontrollern und das Erlernen der Grundzüge der Mikrocontrollerprogrammierung. Dies ist durchaus Neuland im Bereich der Virtuellen Realität an der Fakultät Medien der Bauhaus-Universität Weimar, deren Teilgebiet die Interaktion mit virtuellen Umgebungen ist.
Mikrokontroller kommen in vielfältigster Weise zum Einsatz. Es gibt kaum einen Bereich, in dem sie nicht vorkommen: Heimelektronik, Automobilbau, Spielzeugindustrie etc. Das Einsatzgebiet ist dabei so umfangreich, wie die Aufgaben, die der Mikrokontroller übernimmt. Er kann eine Vielzahl von verschiedensten Sensordaten auswerten, die Daten per Analog-Digital-Wandler bzw. Digital-Analog-Wandler umwandeln und über entsprechende Ausgänge weiterleiten, wo diese zum Beispiel Aktoren ansteuern können
Zu Beginn des Projekts wurden die Möglichkeiten untersucht, die moderne Mikrokontroller bieten und wie diese zum Einsatz gebracht werden können. Im Speziellen wurde der ATmega32 der Firma Atmel verwendet, da dieser uns zur Verfügung stand und in den späteren Teilprojekten involviert wurde.
Darüber hinaus wurde der Umgang mit den Geräten über die Programmierung in Assembler und C in Übungen erlernt. Abschließend wurden in kleineren Gruppen Ideen entwickelt, in denen haptische Eingabegeräte durch Mikrokontroller gesteuert werden sollen.
Der Mikrokontroller hat dabei unterschiedliche Aufgaben zu erfüllen. Er soll möglichst die Eingaben des Benutzers verarbeiten, die über unterschiedliche Sensoren eingelesen werden. Gleichzeitig soll er die Reaktionen des Gerätes auf eine Eingabe des Benutzers oder des Systems kontrollieren, in dem er entsprechende Aktoren ansteuert. Des Weiteren soll er eine Schnittstelle anbieten, die die Kommunikation zwischen Computer und Gerät sicherstellt.



CubicMouse: Beispiel für ein 12 DOF Eingabegerät.

1.2. Eingabe-/Ausgabegeräte

Die Bedienung eines Computers (im weitesten Sinne) setzt das Vorhandensein von Eingabegeräten („Input-Devices“) voraus, um mit dem System interagieren zu können. Eingabegeräte wandeln dazu relevante Information vom Benutzer in Daten um, mit denen der Computer arbeiten kann. Das Drücken einer Tastaturtaste liefert den entsprechenden Code an den Computer, die Bewegung der Maus erzeugt relative Ortskoordinaten usw.
Es gibt dabei eine Vielzahl von unterschiedlichen Geräten, die je nach Einsatzgebiet Vorteile mit sich bringen. Man unterscheidet dabei zwischen relativen Informationen („Locators“, z.B. Maus), absoluten Informationen („Valuators“, z.B. Lautstärkeregler) und diskreten Informationen („Choice Devices“, z.B. Tastatur). Handelt es sich bei den Benutzerinformationen um Ortskoordinaten, kann es sich um unterschiedliche Freiheitsgrade handeln, den

so genannten „Degrees of Freedom“ (DOF). Während z.B. eine Maus nur 2 Freiheitsgrade besitzt, gibt es auch Eingabegeräte mit 6 Freiheitsgraden (zusätzlich zu Translation im 3D-Raum noch Rotation um die 3 Achsen) und mehr. Die Eingabegeräte stellen im Idealfall eine Abstraktion/Approximation der computerseitigen Aufgabenstellung dar.
Um die Konsistenz innerhalb des Systems zu bewahren, muss der Benutzer ein Feedback zu seinen Aktionen erhalten. Dies geschieht hauptsächlich auf audiovisuellem Weg, d.h. dem Benutzer werden auf einem Ausgabegerät (visuell z.B. Monitor, akustisch z.B. Lautsprecher) computerspezifische Daten „anschaulich“ dargeboten. Ein haptisches Feedback ist dabei eher die Ausnahme.
Es gibt allerdings auch Mischformen von Ein- und Ausgabegeräten. So ist z.B. ein Touch-Screen gleichzeitig Eingabegerät und visuelles Ausgabegerät. Die Gruppe der Force-Feedback-Geräte vereinen Eingabegerät und das seltene haptische Ausgabegerät. Der Benutzer kann mit diesen Geräten entweder eine Reaktion auf seine Aktion wahrnehmen oder eine Aktion des Systems spürbar wahrnehmen. Haptische Wahrnehmung beinhaltet dabei Druck-, Temperatur- und Texturwahrnehmung. Feedback kann also durch autonome Krafteinwirkung auf das Gerät, durch Temperaturveränderung und durch Textur-Variation erzeugt werden.
Die Vorteile liegen auf der Hand. Der Benutzer erlangt seine Informationen nicht nur auf visuellem oder akustischem Wege, sondern auch auf haptischem Wege. Durch das Ansprechen eines weiteren Sinnes, dem Tastsinn, wird die Aufmerksamkeit des Benutzers gesteigert. Des Weiteren kann er das System (in unserem Fall die virtuelle Umgebung) viel intuitiver bedienen, da das Feedback auch direkt seine Eingabe beeinflusst.

1.3. Idee eines Force-Feedback-Trackballs

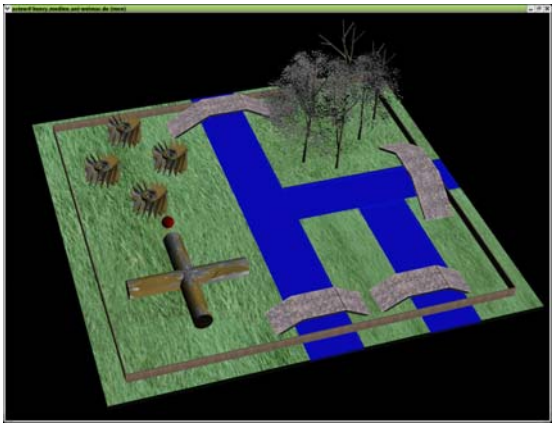
Um die im vorherigen Abschnitt angesprochenen Überlegungen zu realisieren, sollte ein experimentelles Beispielgerät entwickelt werden. Wir überlegten uns dazu, wie man bestehende einfache Eingabegeräte um eine „haptische Dimension“ erweitern könnte. Theoretisch ist es möglich alle vorhandenen Eingabegeräte um Force-Feedback zu erweitern. So könnte man mit Sperren oder Motoren die Eingabemöglichkeit des Nutzers beschränken, z.B. bestimmte Tasten sperren, die Auslenkung eines Joysticks variieren, Geräte mit Vibrationsmotoren ergänzen, usw.
Wir entschieden uns für ein weit verbreitetes Eingabegerät, nämlich einen Trackball (2 DOF), und ergänzten ihn mit Aktoren. Der Force-Feedback-Trackball sollte sich aktiv bewegen, beschleunigen und bremsen oder blockieren können. Die aktive Bewegung des Trackballs sollte mit Hilfe von zwei Motoren, die über die Achse mit dem eigentlichen Trackball verbunden sind, in alle Richtungen möglich sein. Das Blockieren sollte mit Hilfe zweier Zugrelais entlang der 2 Achsen möglich sein.
Weitere Möglichkeiten wären gewesen, den Trackball mit einem Vibrationsmotor und einer Wärmeplatte auszustatten.

Der bestehende Microsoft Trackball „Explorer“ soll um eine „haptische Dimension“ erweitert werden.

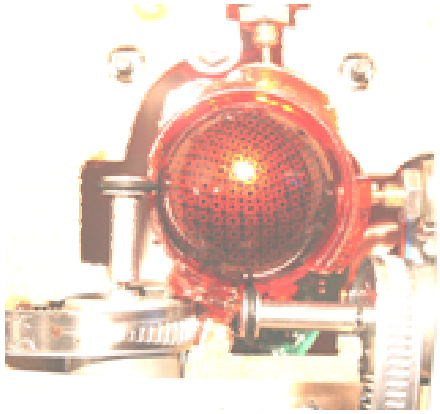
1.4. Anwendungsbereich

Primär sollte der Anwendungsbereich in der Interaktion mit virtuellen Umgebungen liegen. Mit einem haptischen Feedback kann die komplexe Szenerie besser erkundet werden und die Interaktion geht im wahrsten Sinne des Wortes leichter von der Hand.
Tritt zum Beispiel der Fall auf, dass ein vom Benutzer gesteuertes Objekt mit einem anderen kollidiert, so könnten die Zugrelais die entsprechende Richtung blockieren und ein weiteres Bewegen des Trackballs verhindern. Dazu wird das jeweilige Zugrelais ausgelenkt

und bildet mit dem gegenüberliegenden Motor eine Art Fixationsachse, was wiederum die Bewegung nur um diese Achse ermöglicht. Der physikalische Vorgang der Kollision kann damit simuliert werden. Wenn beide Zugrelais angesteuert werden, wird der Trackball komplett blockiert.
Des Weiteren könnte eine Bewegungsunterstützung bzw. -verringerung durch die Motoren erfolgen. Man stelle sich zum Beispiel vor, ein virtueller Ball wird auf einer Schräge positioniert und rollt diese daraufhin physikalisch korrekt hinab. Die Motoren könnten anhand der errechneten Richtung und Geschwindigkeit die Bewegung des virtuellen auf den realen Ball (Trackball) abbilden. Ebenfalls könnte ein Objekt Strömungen ausgesetzt sein. Je nachdem, ob es sich um Kraft entgegen oder gleich der Bewegungsrichtung des Objekts handelt, können die Motoren die Bewegung des Trackballs durch den Benutzer erschweren oder verstärken (unterstützen).
Bezogen auf virtuelle Umgebungen kann mit Hilfe des Force-Feedback-Trackballs die Bewegung/Interaktion in dieser besser vermittelt werden. Der Benutzer spürt, wenn das virtuelle Objekt sich auf einer Schräge befindet, mit anderen Objekten kollidiert, in Gruben/Löcher geraten ist oder eine Bewegung vom System ausgeht. Unterschiedliche Materialeigenschaften können ebenfalls simuliert werden. Die Bewegung in flüssiger Umgebung ist schwieriger als in Luft, auf Öl geht es leichter als auf warmem Asphalt usw.
Der sekundäre Anwendungsfall wäre die Interaktion in Betriebsystemen. Gerade die weit verbreiteten Trackballs und Mäuse (die grundsätzlich auf dem gleichen Prinzip beruhen) würden damit ein enorm großes Anwendungsgebiet erschließen. So könnte die Bewegung der Maus an den Desktopbereich gebunden sein, d.h. eine „Kollision“ mit dem Bildschirmrand oder einem bestimmten Bereich, der die Aufmerksamkeit des Benutzers erfordert, könnte zum Sperren des Trackballs führen. Bei Problemlösungen könnte der Mauszeiger durch die Bewegungen der Motoren zu den wichtigeren Bereichen führen (Fenster, Menüs, Knöpfe, ...). Wichtige Ereignisse wie Fehlermeldungen, Email-Nachrichten oder Terminerinnerung würden durch Auslösen des Vibrationsmotors haptisch signalisiert werden. Die Vielzahl der Möglichkeiten könnte auch zur Unterstützung der Bedienung des Betriebssystems durch Menschen mit Imperfektionen, wie beispielsweise Blinden, dienen.

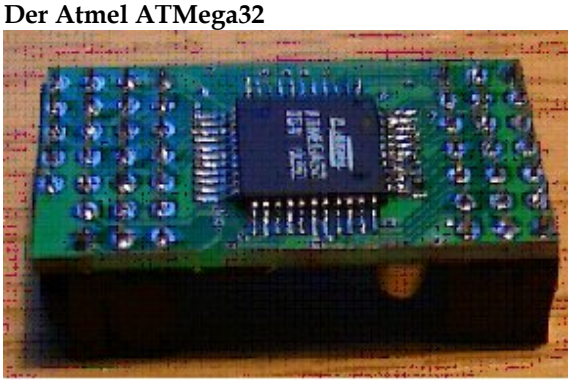


Die Interaktion ist nicht nur in virtuellen Umgebungen ein wichtiger Aspekt.



2. Der Mikrocontroller

Der Atmel ATMega32



2.1. Einführung

Ein Mikrocontroller/Mikrochip (in Zukunft MC) ist eine Art Mini-Computer, d.h. er kann nicht nur Berechnungen durchführen, wie eine CPU (Central Processing Unit, das Herz eines PC's), sondern verfügt auch über eigenen Arbeits-, Programm- und ggf. Datenspeicher.

Er besitzt einige Ein- und Ausgänge, die so-genannten Pins, mit teilweise festgelegtem, teilweise frei wählbarem Verwendungszweck.

Normalerweise wird er mit circa 5V versorgt und besitzt eine Reset-Funktion, d.h. das Programm kann neu gestartet werden. Wie man

Ein Programm erstellt und auf den MC lädt, erfahren Sie im Abschnitt „Die Mikrocontroller-Programmierung".

Die Ausstattung kann sehr unterschiedlich sein. Die Ausstattung des von uns verwendeten Mikrocontrollers ist sehr groß und gibt einen guten Überblick über den Funktionsumfang, den ein MC haben kann.

Im Folgenden werden die einzelnen Features dieses MC aufgelistet und kommentiert.

2.2. Der Atmel ATMega32 – Features

- 2.7-5.5 Volt Stromversorgung
- 8 MHz Taktfrequenz
- 32 Kilobyte Flash-Programmspeicher
- 2 Kilobyte SRAM-Arbeitsspeicher
- 1 Kilobyte EEPROM-Datenspeicher
- 8 Bit Wortbreite
- RISC-Architektur
- 32 allgemein verwendbare Register
- zahlreiche Spezialregister
- Schnittstellen:
    - U(S)ART
    - Serielles Interface
    - SPI-Interface
- Counter
- ADC
- Analog-Comparator
- Interrupts
- Sleep Modes

- Watchdog-Timer
- Pins
- Programming Lock
- JTAG Debug-Interface
- ISP (In System Programming)
- 4 PWM-Channels

- 2.7-5.5 Volt Stromversorgung

An zwei Pins werden Masse und Spannung angelegt. Der Chip benötigt je nach Stromverbrauch der Peripherie an den anderen Pins unterschiedlich viel Spannung. Prinzipiell sollte die Spannung von 2.7 Volt an soweit hochgeregelt werden, bis ein zuverlässiges Arbeiten des Mikrocontrollers eintritt.

Verpolung oder zu hohe Spannungen führen zur Zerstörung des MCs, weshalb ein Spannungsregler und ein Verpolschutz (Diode) vorgeschaltet werden sollten.

- 8 MHz Taktfrequenz

Die Taktfrequenz kann bei Bedarf heruntergeregelt werden (auf 4, 2 oder 1 Mhz).

Andere MCs besitzen üblicherweise Frequenzen im gleichen Bereich. Die Befehle sind oft sehr schnell, d.h. benötigen oft nur einen Zyklus zur Ausführung.

Es können der interne Quarzoszillator oder ein externer Oszillator verwandt werden. So ist eine Synchronisation mit anderen Geräten möglich.

Die eigentliche Taktfrequenz steigt mit der anliegenden Spannung und variiert individuell leicht aufgrund von Produktionsungenauigkeiten.

Ein umfangreiches Kapitel zu diesem Thema ist im Datenblatt enthalten, das auch die Berechnung der Parameter für die Kommunikationsschnittstellen sowie die Arbeitsweise der Taktung und Problembehebungen in diesem Bereich detailiert erläutert.

- 32 Kilobyte Flash-Programmspeicher

Der MC hat (wie die meisten Anderen) eine sogenannte Harvard-Architektur, d.h. Datenspeicher und Programmspeicher sind getrennt. Der Programmspeicher ist laut Datenblatt bis zu 10.000 mal programmierbar, was in der Praxis deutlich ausreicht.

Es existieren Befehle zur Selbstmodifikation und der Speicher kann auch in den PC eingelesen werden. Zusätzlich können über die Programmierschnittstelle eigenständig Programmaktualisierungen bezogen werden. Der fertig programmierte Mikrocontroller kann mit sogenannten Sperrbits vor einer Reprogrammierung oder Löschung geschützt werden.

Es existiert ein spezieller Bootsektor, ebenso wie Interruptadressen. An diese Stellen werden Befehle für die entsprechenden Aufgaben gesetzt.

- 2 Kilobyte SRAM-Arbeitsspeicher

Statischer Arbeisspeicher ist sehr schnell.

- 1 Kilobyte EEPROM-Datenspeicher

Electrically Erasable ROM kann im Gegensatz zu normalem EPROM auch mit einem Computer selbst gelöscht werden und es sind keine speziellen Geräte notwendig.

Es kann eine Datei mit Initialwerten erstellt werden, die anschließend in das EEPROM geladen wird. Hier können Daten permanent gespeichert werden (z.B. Nutzereinstellungen).

Der Zugriff liegt zwischen 1.9 – 3.8 Millisekunden, weshalb sparsam hiermit umgegangen werden sollte.

- 8 Bit Wortbreite

Arbeits-, Daten- und Programmspeicher sowie alle Register arbeiten mit acht Bit Wortbreite, so dass z.B. 1024 Adressen für den Datenspeicher verfügbar sind. Des Weiteren hat das zur Folge, dass 16-Bit-Operationen (oder noch Höherwertige) stets in 8-Bit-Operationen

aufgespalten werden müssen. Beispielsweise müssen die höherwertigen und niederwertigen 8-Bit eines 16-Bit-Wertes stets separat gelesen/geschrieben werden.

- RISC-Architektur

Die „Reduced Instruction Set Computer"-Architektur ist die Folge der CISC-Architektur („Complex ..."). Die CISC-Architektur besaß einen grossen Befehlssatz (500 Befehle und mehr). Oft galt das „10-90"-Prinzip, d.h. praktisch wurden 10 % aller Befehle zu einem Anteil von 90 % benutzt, während 90 % der Befehle zumeist redundant waren und nur in 10 % aller Fälle benutzt wurden. Die RISC-Architektur verwendet deutlich weniger Befehle, in unserem Falle 130.

- 32 allgemein verwendbare Register

Die Register befinden sich direkt in der CPU und sind sehr schnell. Hier können Daten gehalten werden, auf denen häufig operiert wird.

- zahlreiche Spezialregister

Hierbei handelt es sich um Register mit speziellen Aufgaben.

Ohne Anspruch auf Vollständigkeit ist dieses:

- Speicher für die Operanden der arithmetischen Operationen
- Speicher für die empfangenen/zu sendenden Daten der Schnittstellen
- zahlreiche Flagregister für Einstellungen des MCs

  Diese Flagregister können je nach Verwendungszweck gesetzt und/oder gelesen werden. Eine gute Referenz im Programm AVRStudio enthalten.

  Beispielsweise wird mit Hilfe der Flagregister bestimmt, welche Pins als Ausgänge und welche als Eingänge benutzt werden, die Pegel der Ausgangsregister, die Taktung des Chips, das Ein-/Ausschalten der einzelnen Interrupts, Fehlerzustände, den Status von Operationen (Flag wird gesetzt, wenn Übertragung fertig o.ä.) und vieles mehr.

- Schnittstellen:

Es sind drei verschiedene Schnittstellen nutzbar. Je nach Schnittstellen des Kommunikationspartners ist die freie Wahl nicht immer gegeben.

- U(S)ART

  Das Universal (Synchronous) Asynchronous Receiver Transmitter Interface benötigt zwei Pins, einen zum Senden und einen zum Empfang von Daten. Es können bis zu 1 Mbit/s erreicht werden, jedoch sinkt mit höherer Bandbreite die Zuverlässigkeit der korrekten Übertragung. Die wichtigsten Eigenschaften des USART sind:

  - synchroner und asynchroner Modus
  - Vollduplex (gleichzeitiges Senden und Empfangen)
  - Frame-Error-Detection
  - Prüfbits
  - Multi-Prozessor-Kommunikation (Addressierung der Teilnehmer)
  - Master- oder Slave-Taktung im synchronen Modus
  - Interruptsteuerung möglich
  - Pufferüberlauferkennung

  Das Datenblatt enthält nicht zu Unrecht ein umfangreiches Kapitel zum USART, welches bei Problemen unbedingt herangezogen werden sollte.

  Die synchrone Kommunikation mit Computern ist PC-seitig leider oft nicht möglich.

- Serielles Interface (two-wire serial interface)

Hiermit werden hauptsächlich angeschlossene ICs (Integrated Circuit – digitale Schaltungen, wie z.B. Datenspeicher, Logikbausteine oder anderen MCs) angesprochen. Es wird ein Pin zur Taktung (maximal 400 kHz) und Einer zum Senden/Empfangen von Daten benötigt.

Wichtigste Features:

  - synchrone Kommunikation im Master- oder Slave-Modus
  - Halbduplex (Bidirektionale Kommunikation, aber kein Gleichzeitiges Senden und Empfangen)
  - 7-Bit Addressierung (127 Teilnehmer)
  - Collision Detection
  - mögliche Interruptsteuerung
  - maximale Übertragungsrate ist theoretisch 400 kByte/Sekunde

- SPI-Interface (Serial Peripheral Interface)

Hierbei handelt es sich um ein Highspeed-Interface zur Kommunikation mit anderen ICs, welches drei Pins benötigt.

Wichtigste Features:

  - Taktrate bis zu 4 MHz
  - Vollduplex (gleichzeitiges Senden und Empfangen)
  - synchrone Kommunikation im Master- oder Slave-Modus
  - Collision Detection
  - mögliche Interruptsteuerung
  - Wecken aus dem Ruhezustand (s.u.)

- Counter

Es existieren drei unabhängige Counter, einer davon mit 16 Bit. Die Zählgeschwindigkeit kann eingestellt werden (Frequenz/[2,4,8,16]). Beim Erreichen eines vorgebbaren Wertes sowie beim Erreichen des Maximalwertes wird ein Interrupt erzeugt. Ebenfalls kann eingestellt werden, ob alternierend auf- und abwärts oder stets aufwärts gezählt wird und ob der Counter beim Erreichen des Referenzwertes zurückgesetzt wird.

- ADC

Der Analog-Digital-Konverter kann eine 8- oder 10-Bit Parallelwandlung an 8 verschiedenen Pins (nicht gleichzeitig) vornehmen.

Es kann eine Samplerate von 15 kS/s (8000 Samples pro Sekunde, 8 Bit) bzw. 76 kS/s erreicht werden.

Gewandelt werden können Spannungen zwischen 0 Volt und der Versorgungsspannung.

Für ein genaues Ergebnis (+- 2 Bit) sollten zur Zeit der Wandlung keine anderen Aktionen an dem Port geschehen. Außerdem ist bei einer instabilen Versorgungsspannung ein interner geregelter Referenzwert von 2.56 Volt vorhanden.

- Analog-Comparator

Mit dem Analog-Komparator kann ein Interrupt ausgelöst werden, falls eine an dem dafür vorgesehenen Pin anliegende Spannung einen anzugebenden Referenzwert überschreitet.

- Interrupts

Damit der MC nicht im Pollingbetrieb ständig alle möglichen Ereignisflags abfragen muss, um Ereignisse zu detektieren, sondern sich ggf. sogar „schlafen legen“ kann, existieren zahlreiche Interrupts.

Die wichtigsten Interrupt-Typen sind:

- Daten angekommen/gesendet für die einzelnen Interfaces
- Fehlerzustand eingetreten (diverse)
- ADC fertig
- Analog-Comparator detektiert Spannung
- Software-Interrupts
- Timer-Interrupt (regelmässig)

Die Interrupts können individuell oder global ein- und ausgeschaltet und ggf. konfiguriert werden. Tritt ein Interrupt auf, so kann automatisch eine beliebige Aktion ausgeführt werden.

- Sleep Modes

Da MCs häufig in Geräten eingesetzt werden, die nur über begrenzte Energiereservern verfügen (z.B. Batteriebetriebene Geräte, beispielsweise Discman), existieren sechs verschiedene Schlaf-Modi, welche verschiedene Features des MCs zwecks Stromersparnis ausschalten. Der MC kann per Interrupt wieder „geweckt" werden. Die Einschaltdauer hängt vom verwendeten Schlafmodus ab.

Die verschiedenen ausgeschalteten Features und die jeweilige Weckdauer können dem Datenblatt entnommen werden.

- Watchdog-Timer

Im Falle eines Deadlocks sorgt der Watchdog-Timer, wenn gewünscht, dafür, daß der MC neu gestartet wird. So können kritische Situationen vermieden werden.

- Pins

Die Pins sind die Anschlüsse des MCs, über die er mit Strom versorgt wird, mit den Peripheriegeräten kommuniziert sowie Ereignisse detektiert und Informationen von Sensoren bezieht o.ä.

Manche Pins haben spezielle Einsatzzwecke, können aber stets als genereller Digital Ein- oder Ausgang genutzt werden, falls der jeweilige spezielle Einsatzzweck (beispielsweise USART) nicht benötigt wird.

- JTAG Debug-Interface

Ist der MC nicht an ein Evaluationsboard angeschlossen (s.u., Mikrocontrollerprogrammierung), so kann die Funktionsweise des Programms nicht oder nur schlecht kontrolliert werden. Der Zustand interner Register kann sowieso nicht direkt kontrolliert werden. Deshalb existiert ein sogenanntes JTAG-Interface, mit dem diese Werte abgefragt werden können und der MC im Step-Modus, also Schrittweise, ausgeführt werden kann.

- ISP (In System Programming)

Damit für die Programmierung des MCs nicht dauerhaft eine oder mehrere Pins reserviert werden müssen, wird die sogenannte ISP-Schnittstelle verwandt. Diese ermöglicht den Anschluss der Programmierschnittstelle an zwei Pins, die weiterhin anderweitig eingesetzt werden können. Im Falle der Programmierung wird in den Programmiermodus geschaltet, danach wieder in den Betriebsmodus und die Pins können ihren eigentlichen Zweck verfolgen.

- 4 PWM-Channels

Da der MC nicht über analoge Ausgänge verfügt, behilft man sich des Tricks der Pulsweitenmodulation (PWM). Will man beispielsweise eine Analogspannung von 3 Volt an einen Ausgang legen, so legt man bei angenommenen 5 Volt Versorgungsspannung in 3 von 5 Zeitintervallen Spannung an den entsprechenden Ausgang. So entsteht ein (nicht ideales) Rechtecksignal. Daß anschliessend einem oder mehreren Kondensatoren in Reihe geglättet werden kann, falls notwendig. Hierfür

existiert ein spezieller PWM-Pin, ebenso wie spezielle PWM-Kanäle. Allerdings kann diese Vorgehensweise auch mit den oben erwähnten Timern realisiert werden.

Weitere Informationen sind auf der Homepage der Firma Atmel verfügbar.

2.3. Die Mikrocontroller-Programmierung

Hier erfahren Sie, was Sie benötigen, um einen Mikrocontroller zu programmieren.

Ausserdem wird erklärt, mit welchen Programmiersprachen Sie das tun können und welche Bibliotheken verfügbar sind.

Es werden die Unterschiede zur herkömmlichen (High-Level-)Programmierung und die einzelnen Schritte bis zum fertig programmierten Mikrocontroller erläutert, nicht aber konkrete Programme vorgestellt.

Im Anhang sind verschiedene in Assembler und C geschriebene Beispielprogramme vorhanden. Diese sprechen meistens die auf dem von der Firma Albotronic bereitgestellten Evaluationsboard (s.u.) vorhandenen Peripheriegeräte an, so daß dieses idealerweise zum testen benutzt werden sollte.

Um einen Mikrocontroller zu programmieren, benötigen Sie folgende Dinge:

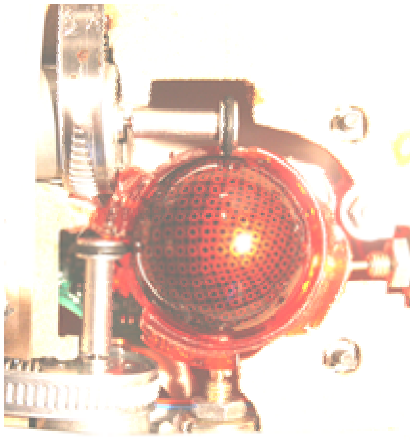
1. **Mikrocontroller**

2. **Stromversorgung**

Der Mikrocontroller muss mit Strom versorgt werden, damit er funktioniert.

An welche beiden Pins Masse und Strom angeschlossen werden und in welchem Bereich die Spannung liegen sollte (beim Atmel ATMega32 5-8 Volt), ist dem Datenblatt zu entnehmen. Dafür kann ein handelsüblicher Trafo verwandt werden.

Um eine geregelte Versorgungsspannung zu gewährleisten, ist allerdings eine Schaltung zu empfehlen, die aus wenigen Bauteilen sehr einfach aufgebaut werden kann und im Tutorial auf www.mikrocontroller.net im Abschnitt 1 („benötigte Ausrüstung“) gut beschrieben ist. Ebenfalls sollte ein Verpolschutz in Erwägung



gezogen werden, den bei Verpolung wird der MC zerstört !

Das lässt sich einfach mit einer Diode realisieren. Eine Diode lässt Strom nur in einer Richtung durch und sperrt die Andere.

3. Compiler

Das Programm für den MC wird wie ein gewöhnliches Programm auf dem PC geschrieben und anschließend mit einem speziellen Compiler kompiliert, der den Befehlssatz des Mikrocontrollers kennt.

In welchen Sprachen Sie das Programm schreiben können, hängt natürlich von den verfügbaren Compilern ab.

In der Regel können Sie einen Chip-spezifischen Assembler- und/oder C-Dialekt verwenden, wie in unserem Fall.

Die Firma Atmel stellt die Simulationsumgebung AVRStudio kostenlos zur Verfügung (s.u.), mit der Sie in Assembler programmieren können.

Ausserdem ist die ebenfalls kostenlose Programmierumgebung ***WinAVR*** verfügbar, welche sowohl Assembler als auch C übersetzen kann und eine C-Bibliothek bereitstellt.

Eventuell gibt es weitere Compiler im Internet, jedoch sind o.g. ausreichend.

4. Download-Programm

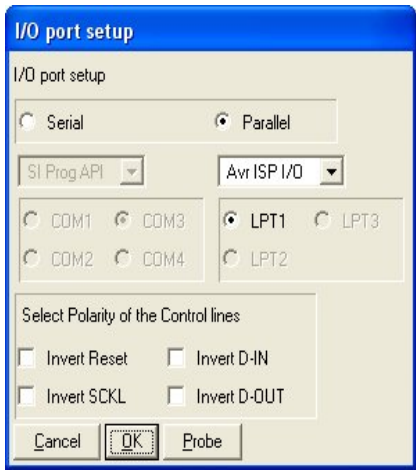
Um das kompilierte Programm auf den MC zu laden, benötigen Sie ein spezielles Programm,

welches das für Sie erledigt. Wir haben das frei verfügbare Programm ***ponyprog*** verwandt, weitere sind verfügbar.

Bevor die Programmübertragung vorgenommen werden kann, müssen in der Regel noch der angeschlossene Chip und der verwendete Port (z.B. Parallelport 1) sowie einige angaben zur Übertragungsweise gemacht werden). Anschliessend muss eine Kalibrierungsfunktion aufgerufen werden, Zuallerletzt kann die Verbindung getestet werden. Das Programm wird übertragen, indem das fertige Programm (*.hex oder .bin) in das Programm geladen und anschliessend übertragen wird. Die Verifizierung kann in der Regel abgebrochen werden.

Im Falle von *Ponyprog* gestaltet sich das Vorgehen wie folgt (ACHTUNG: Es funktioniert bisher nur die aktuelle Version 2.06c Beta !):

1. Wählen Sie im Menü *Device – AVRMicro – ATMega32* aus.
2. Wählen Sie im Menü *Setup - Interface Setup* und nehmen Sie die Einstellungen wie dargestelltvor:



3. Drücken Sie auf ***Probe.*** Sie bekommen eine Erfolgs- oder Fehlermeldung. Im Falle einer Fehlermeldung ist ihre Schnittstelle nicht oder fehlerhaft angeschlossen. Überprüfen Sie alle Verbindungen. Auch die Stromversorgung der MC muss eingeschaltet sein. Kontrollieren Sie die relevanten Jumper des Evaluationsboards.
4. Vor der ersten Benutzung eines jeden Mikrocontrollers bzw. nach einem Austausch sollten Sie stets *Setup – Calibration* ausführen.
5. Anschliessend wählen Sie die gewünschte Datei über das *File*-Menü aus und laden es mit Ctrl-P in den MC herunter.

5. Verbindung Computer-Mikrocontroller

Damit das Programm auf den MC geladen werden kann, muss der Computer mit dem MC verbunden werden.

Dafür gibt es prinzipiell verschiedene Möglichkeiten:

1. Sie benutzen ein Evaluationsboard (s.u.). In diesem Falle verbinden Sie das Board über seine Programmierschnittstelle (in der Regel die parallele Schnittstelle) mit dem Computer.
2. Sie verbinden den Computer manuell mit dem MC über eine Schnittstelle Ihrer Wahl (das Download-Programm (4) muss diese Schnittstelle unterstützen!), z.B. über die parallele Schnittstelle/den Druckerport (LPT)). In diesem Fall müssen Sie sich die Verbindung selberbauen. Ein einfacher Schaltplan ist ebenfalls auf www.mikrocontroller.net im Abschnitt 1 verfügbar. Ebenfalls ist dort eine Bestellmöglichkeit für einen billigen fertigen Adapter angegeben.
3. Sie bestellen ein ISP-Programmiermodul bei www.atmel.com, was allerdings sehr teuer ist.

   Der Mikrocontroller wird stets über die sogenannte ISP-Schnittstelle (In-System-Programming) verbunden.

Diese ermöglicht die Programmierung ohne die Aufopferung von Pins. Dabei können die Pins, welche für die ISP-Schnittstelle benutzt werden, frei benutzt werden, wenn der MC nicht reprogrammiert wird.

Sie sollten für das fertige Gerät eine dauerhafte ISP-Schnittstelle einrichten, falls der MC am fertigen Gerät eventuell neu programmiert werden muss bzw. wenn das Gerät sich noch in der Testphase befindet und Sie ggf. den MC debuggen müssen.

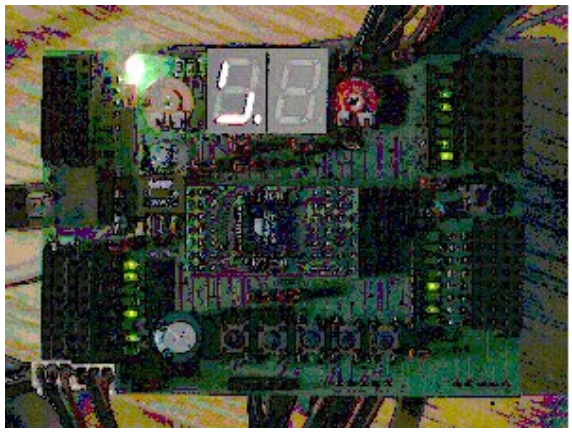
6. Evaluationsboard (optional)

Um ihre Applikation zu testen, ohne den MC in die geplante Umgebung eingebaut zu haben, können Sie sich ein Evaluationsboard kaufen oder selber bauen. Sehr beliebt ist hier das STK500, welches bei vielen bekannten Elektronik-Vertrieben zu erwerben gibt. Für uns hat die Firma Albotronic eigens Evaluationsboards hergestellt.

Der MC wird in das Evaluationsboard eingesetzt und stellt in der Regel eine ISP-Schnittstelle sowie ein USART-Interface oder andere Interfaces bereit, sowie LEDs, welche die Spannungen der einzelnen Pins anzeigen (0 oder 5 Volt), ein 7-Segment-Display, Taster, Potentiometer eine RGB-LED o.ä.

So können Sie die MC-Umgebung simulieren und die Reaktionen testen.

Hier finden sie eine Beschreibung des von uns verwendeten Evaluationsboards.



Das Evaluationsboard der Firma Albotronic.

7. Hilfsbibliothek (optional)

Die Entwicklungsumgebung WinAVR (avr-libc) und Andere stellen eine C-Bibliothek bereit, welche häufig verwendete Features bereitstellt, die sonst selbst programmiert werden müssen. Beispiele für Features sind mathematische Funktionen, Stringbehandlung, ein einfaches Dateisystem, Interrupthandling, Datentypen, Sleep Modi, Datenspeicherzugriff usw. Eine Bibliotheksreferenz ist im Anhang referenziert.

8. Simulationsumgebung (optional)

Atmel stellt die Simulationsumgebung AVRStudio bereit, welche das Testen von Programmen für alle möglichen Atmel-MCs auch ohne das Vorhandensein Dieser ermöglicht, indem sie sie emuliert. Es können interaktiv Spannungen an die einzelnen Pins

gelegt werden und die Reaktionen beobachtet werden.

Aber es kann auch mit einem angeschlossenen MC über die ISP- oder JTAG-Debug-Schnittstelle kommuniziert werden und einzelne Befehle im können Step-Modus ausgeführt werden.

Leider kann hier bisher nur in Assembler programmiert werden.

9. JTAG-Debug-Konsole (optional)

Atmel stellt eine JTAG-Konsole bereit, mit welcher der MC gedebugt werden kann. Allerdings ist Diese sehr teuer.

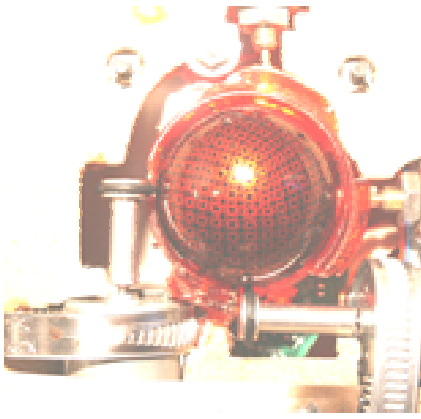
10. Kommunikationsinterface (optional)

Soll der MC mit anderen Devices, z.B. mit dem PC, über eines der drei Schnittstellen, z.B. über das USART, kommunizieren, so muss auch diese Schnittstelle gebaut werden. Eine Bauanleitung für eine USART-Schnittstelle steht ebenfalls auf www.mikrocontroller.net bereit.

11. weitere Bauelemente (optional)

Sollen weitere Befehlgeber (z.B. Taster) oder Befehlsempfänger (z.B. Display) an den MC angeschlossen werden, so muss auch dieses bewerkstelligt werden. Im Tutorial von www.mikrocontroller.net wird der Betrieb eines 7-Segment-Displays gut beschrieben. Eine Einführung zum Anschluss von Peripherie aller Art finden Sie hier.

Verweise zu Codebeispielen, Tutorials, Befehlsreferenzen und Programmiertips finden Sie im Anhang.



3. Die Hardware

Eine besondere Herausforderung dieses Projektes war sicherlich die Hardware.

Die Natur hat ihre eigenen Gesetze und diese müssen bei der Auswahl der Hardware und bei der Interaktion der Komponenten untereinander sowie der Interaktion der Hardware mit der Software berücksichtigt werden.

Der erste Abschnitt beschreibt unser Vorgehen und wichtige Problemstellungen beim Bau des Force-Feedback-Trackballs.

Im zweiten Abschnitt werden die einzelnen Bauelemente aufgelistet und anschließend eingehend einer Diskussion unterzogen, soweit notwendig.

Der dritte Abschnitt beschreibt den konkreten Aufbau des Gerätes und gibt einige wichtige Hinweise und Tipps zum Betrieb.

Abschliessend werden im vierten Abschnitt die Erfahrungen und Probleme sowie wünschenswerte Verbesserungen der Hardware dargestellt.

3.1. Der Bau des Force-Feedback Trackballs

Zunächst mussten wir herausfinden, welche Bauelemente wir benötigen, um den Trackball zu realisieren, was in unserem Falle sicherlich relativ klar war.

Anschliessend stellte sich die Frage, welche Eigenschaften diese Bauelemente haben sollten (z.B. wie stark die Motoren sein müssen). Dafür zogen wir unsere im Grundstudium gewonnenen Kenntnisse der Elektrotechnik und der Mechanik heran, um theoretische Berechnungen der benötigten Bauteilcharakteristiken anzustellen.

Die Theorie allein reicht hier jedoch nicht aus. Die von einem Nutzer zur Bedienung eines Trackballs aufgewendete Kraft oder die ideale Geschwindigkeit und Kraft der Motoren lassen sich nicht berechnen, sondern nur empirisch herausfinden.

Da wir über wenig praktische Erfahrung mit Motoren und Relais verfügten, mussten wir also unsere diesbezüglichen Einschätzungen in die Berechnungen einfliessen lassen und einen gewissen Spielraum einbeziehen, so daß wir schliesslich mehrere Elemente mit leicht unterschiedlichen Kennwerten beschafften, um Diese dann auszutesten.

Damit wären wir beim nächsten Punkt:

Wir wussten nun, welche Eigenschaften unsere Bauteile in etwa haben sollten, aber wir wussten nicht, wo wir sie beschaffen sollten oder wie wir sie selber bauen. Wir wussten nicht einmal, ob Bauteile mit diesen Eigenschaften überhaupt zu annehmbaren Kosten realisierbar waren. Wir studierten somit die Kataloge einiger bekannter Versandhäuser, um geeignete Bauelemente zu finden. Oft mussten Kompromisse eingegangen werden, was Bauteileigenschaften betrifft, so zum Beispiel zwischen Kraft und Größe eines Motors.

Ebenfalls spielte die Bezahlbarkeit der Bauteile oft eine Rolle, da schnell nie erträumte Dimensionen erreicht werden. Oft weisen bestimmte Bauelemente Vorteile auf, die durch gravierende Nachteile wieder zunichte gemacht werden.

Das Bestellen zahlreicher Bauelemente zum Zwecke des Experimentierens war aufgrund zu hoher Kosten ebenfalls nur eingeschränkt möglich.

Ein weiterer Punkt bei der Auswahl der Elemente war die Kompatibilität der verschiedenen Elemente untereinander. Ein Motor darf zum Beispiel nicht mehr Strom

verbrauchen, als ein von uns benutzter Motortreiber liefern kann.

Nachdem die Bauteile ausgesucht waren, mussten sie zusammengebaut werden, was ebenfalls keine leichte Aufgabe war. Es mussten eine geeignete Anordnung sowie die Möglichkeit des nachträglichen Austausches von Komponenten und ggf. Justierungsvorrichtungen berücksichtigt werden.

Abschließend sei bemerkt, dass wir im Grundstudium zwar einige, die Hardware betreffende theoretische Grundkenntnisse vermittelt bekamen (Mechanik, Elektrotechnik etc.), dies aber nicht zum Studienschwerpunkt gehört. Fundiertere Grundkenntnisse und mehr praktische Erfahrungen wären hier hilfreich gewesen. Eine Zusammenarbeit mit entsprechenden Fachleuten wäre ebenfalls denkbar gewesen. Dr. Schatter hat in der Abschlusspräsentation erwähnt, dass es einige in diesem Bereich hochqualifizierte Leute an Thüringer Hochschulen gibt.

3.2. Die Komponenten

Um unseren Force-Feedback-Trackball zu realisieren, benötigten wir somit folgende Bauteile:

- **Trackball (bzw. Maus mit integriertem Trackball)**
- **2 Motoren**
- **2 Antriebsrädchen**
- **2 Antriebsgummis**
- **2 Zugrelais/Hubmagneten**
- **2 Taster**
- **1 Mikrocontroller (Atmel ATMega32)**
- **Evaluationsboard**
- **UART-Adapter**
- **serielles Kabel**
- **Programmieradapter**
- **Parallelkabel (Downloadkabel)**
- **2 Motortreiberplatinen**
- **Lötlitze (Kabel) & Steckverbinder**
- **4 Federn**
- **2 Rohrschellen**
- **Chassis**
- **2 Transformatoren zur Stromversorgung**

Im folgenden werden die einzelnen Bauelemente diskutiert:

- **Trackball (bzw. Maus mit integriertem Trackball)**

Grundlage unseres Trackballs sollte ein geeigneter kommerzieller Trackball sein.

Da unser Studienschwerpunkt nicht auf der Hardware liegt, ist das akzeptabel, auch wenn der Eigenbau mit eines durch eine Lichtschranke abgetasteten Trackballs denkbar gewesen wäre.

Der Trackball sollte idealerweise folgende Eigenschaften aufweisen:

- eine genügend grosse Kugel, die die Anbringung der Motoren und Relais zulässt
- eine optische Abtastung aufgrund der Platzersparnis (und der höheren Präzision) gegenüber einem mit Rädchen und Lichtschranke abgetasteten Trackball

Diese Eigenschaften erfüllt der *Microsoft Optical Trackball*, eine Produktbeschreibung liegt im Anhang vor.

Der eindeutige Nachteil der optischen Abtastung war allerdings, dass uns die Methode der Auswertung der durch die Kamera gewonnenen Abtastdaten nicht zugänglich war, was eine direkte Verarbeitung auf dem Mikrocontroller nicht möglich machte.

Auch die Auflösung des USB-Protokolls, welches von dem On-Board-Chip-erzeugt wurde, ist durch den Microcontroller nicht

auflösbar, da das sehr komplexe Protokoll zuviel Rechenzeit und Implementationsaufwand in Anspruch nehmen würde.

Stattdessen mussten wir die Daten über den USB-Port des PCs auslesen und über das UART an den Microcontroller weiterleiten.

- **2 Motoren**

Die schwierigste Entscheidung war die Auswahl der Motoren. Die Motoren sollten dabei idealerweise folgende Eigenschaften aufweisen:

- Bidirektionalität
- ausreichendes Drehmoment
- genügend hohe Reaktionsgeschwindigkeit
- akzeptable Grösse
- akzeptabler Preis
- kein sofortiges Durchbrennen beim Blockieren des Motors
- möglichst Gleichstromversorgung
- möglichst nicht zu hoher Stromverbrauch
- Möglichkeit des Positionstrackings (Motorstellung ermitteln/Schrittweite angeben)

Es stellte sich zunächst heraus, daß es äusserst unterschiedliche Motoren mit verschiedenen Characteristiken, Preisen und Anwendungszwecken gibt. Jedoch kamen für unsere Zwecke lediglich zwei Motortypen in Frage, welche auch die Einzigen in dem von uns geforderten Leistungsbereich waren.

DC-Motoren und Schrittmotoren waren diese beiden Typen.

Schrittmotoren haben den Vorteil, dass sie trotz geringer Baugrösse sehr stark sind und digital Angesteuert werden, jedoch mit externem Strom versorgt werden. „Digital angesteuert“, bedeutet, dass ich dem Motor mitteile, um dass er sich nun um einen Schritt vorwärts oder rückwärts bewegen soll. Leider werden diese Vorteile durch einige Nachteile kompensiert:

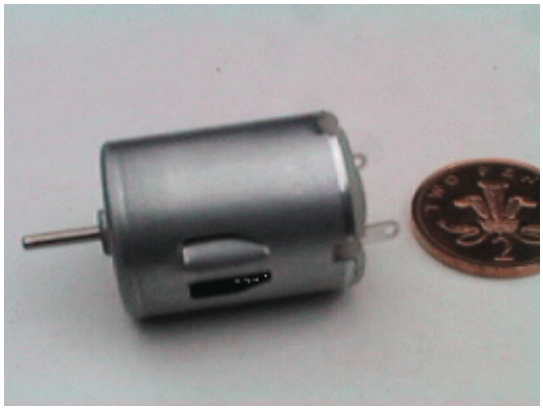
Das Schrittweise drehen sorgt für ein ruckhaftes Drehen und eine relativ geringe Drehzahl. Dennoch bleibt auszutesten, ob sich diese Motoren nicht dennoch eignen.

Wir haben uns in unserem Projekt für DC-Motoren entschieden (DC = direct current = Gleichstrom). Diese schienen zunächst die beste Wahl zu sein, auch wenn sich im Laufe des Projekts herrausstellte, dass einige Eigenschaften besser sein könnten. Sie sind relativ billig und die Bidirektionalität wird einfach durch Änderung der Stromrichtung erreicht. Wie der Name schon sagt, beziehen sie Gleichstrom. Das Positionstracking bzw. die Angabe der Schrittweite entfallen zwar, jedoch hat sich im nach hinein herausgestellt, dass dieses ohnehin überflüssig gewesen wäre. Die Grösse, Kraft, Verbrauch und Reaktionsgeschwindigkeit (Trägheits- bzw. Massenabhängig) hängen natürlich voneinander ab. Die Reaktionsgeschwindigkeit war akzeptabel. Wir fassten diese nicht in Werte, sondern testeten die Motoren praktisch. Die Grösse der Motoren nimmt leider mit der zunehmenden Kraft stark zu.

*Beispiel: Ein DC-Motor mit den ungefähren Massen: 2*2*3 cm kann ca. 100mNm Kraft erzeugen, während ein 1Nm-Motor bereits ca. 5*5*8 cm einnehmen und 500g Gewicht haben kann.*

Die schliesslich von uns bestellten zwei verschiedenen Motorentypen erwiesen sich als leider etwas zu schwach. Stärkere Motoren werden aber meistens zu gross und schwer sein. Auch die Palette der am Markt erhältlichen Motoren und die geringe Bandbreite verfügbarer Leistungen verwunderten uns.

Die Motoren wiesen in der Praxis sehr gute Stabilität im blockierten Zustand auf, d.h. Sie brannten nie durch. Der Stromverbrauch lag in dem von den Motortreibern unterstützten Bereich von 700 Milliampere, auch wenn die theoretischen Werte höher waren.



Im Folgenden wird ein kleiner Exkurs in die Mechanik vorgenommen:

Drehzahl, Drehmoment und Leistung:

Um diese Werte zu ermitteln, ist einiges (physikalisches) Grundwissen notwendig. Sollten Sie mit der Beziehung zwischen Kraft, Leistung und Drehzahl eines Motors vertraut sein, können Sie die folgenden Erläuterungen überspringen.

Das Drehmoment eines Motors ist seine Kraft und wird in Newtonmeter angegeben.

Ein Kilopond ist die Gewichtskraft, welche die Masse von einem Kilogramm auf der Erdoberfläche erzeugt:

1 kP = 1 kg x g (Gravitationskonstante der Erdoberfläche = 9.81)

1 kP = 9.81 N (Newton)

Ein Newton entspricht also in etwa der Gewichtskraft von 100 Gramm.

Bei Motoren wird die Kraft in Bezug auf eine bestimmte Drehzahl angegeben. Jeder Motor besitzt seine eigene Charakteristik, was das Verhältnis von Drehzahl und Kraft betrifft.

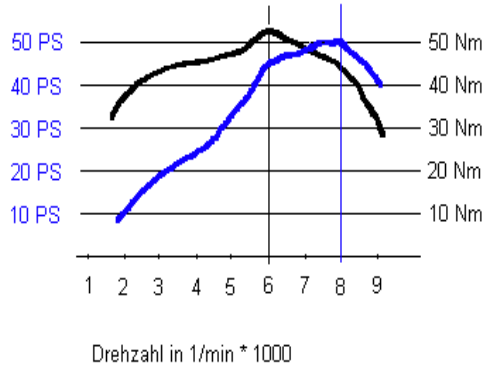
Jeder Motor hat bei einer bestimmten Drehzahl eine maximale Kraft. Diese Kraft wird im Übrigen zur Bestimmung der Leistung eines Motors herangezogen:

Leistung = Drehzahl x Kraft

Zur Erinnerung: Die Leistung wird in Pferdestärken, Watt oder Newtonmeter/Sekunde angegeben.

1 Nm/s = 1 Watt und 1,35 PS = 1 kW

Um eine qualifizierte Aussage über einen Motor treffen zu können, müssen zwei Werte des oben genannten Dreisatzes angegeben werden. Sollten mehrere Drehzahlen und Drehmomente angegeben sein, so müssen die Werte „Drehzahl bei Leistungsanpassung“ und „Drehmoment bei Leistungsanpassung“ gewählt werden. Das sind die Werte, welche miteinander multipliziert den grössten Wert, d.h. die grösste Leistung ergeben. Das ist nicht notwendigerweise die Drehzahl mit dem höchsten Drehmoment, sondern in den meisten Fällen eine etwas höhere Drehzahl. Eine charakteristische Kennlinie ist im folgenden dargestellt:



Ein solches Diagramm sollte ebenfalls in einem guten Datenblatt enthalten sein (z.B. könnte die Leistung im oberen und unteren Drehzahlbereich stark fallen, was sicherlich niemandem gefällt).

Um eine Vorstellung von der Bedeutung des Dreisatzes zu bekommen, sei ein anschauliches Beispiel angeführt:

Es existieren zwei Autos mit der gleichen Leistung. Das erste Auto besitzt ein grosses Drehmoment bei Leistungsanpassung, das Andere eine hohe Drehzahl.

Das wird sich wie folgt auswirken:

Ist Auto 1 schwer beladen, wird es weiterhin gut beschleunigen und auch nahe an die bisherige Maximalgeschwindigkeit kommen; das Auto 2 wird hingegen sehr schlecht beschleunigen. Das Auto 2 wird Dank der höheren Drehzahl bei angenommener gleicher Getriebeübersetzung sehr viel schneller fahren, wenn es nur gering beladen ist.

Für unsere Motoren bedeutet das:

Das Drehmoment ist der entscheidende Kennwert für uns, denn es ist wichtig, daß der Ball bei einer Berührung durch die Hand des Nutzers nicht sofort stehenbleibt, aber auch nicht übermässig stark ist.

Desweiteren sollte das Drehmoment für alle Drehzahlen möglichst gleich sein, so daß bei einer Verdopplung der (Steuer-)Spannung auch eine Verdopplung der Drehzahl eintritt. Diese Überlegungen nutzen uns angesichts der bereits erwähnten geringen Bandbreite an angebotenen Motoren herzlich wenig.

Allerdings benötigen wir diese Werte, um die Rotationsgeschwindigkeit und -kraft von den Motoren auf den Ball umzurechnen.

Die dahinterstehende Rechnung wird als bekannt angenommen. Kurzgesagt, es geht um die passende Übersetzung.

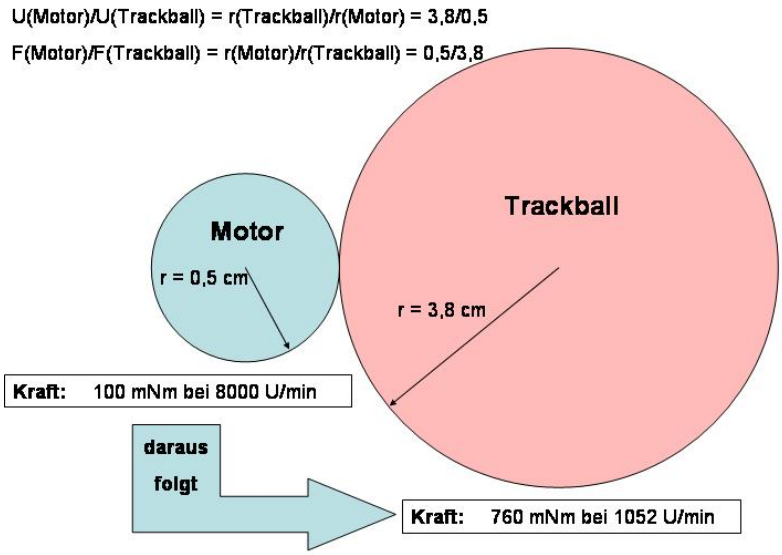
Die von uns verwendeten Motoren besitzen ungefähr 100 mNm Drehmoment. Wir müssen hier nicht allzu genau sein, da es sich lediglich um Richtwerte handelt, denn bei der durchschnittlichen Handkraft handelt es sich lediglich um eine Abschätzung.

Eine Kraft von 380 mNm entspricht der Kraft, die benötigt wird, um 76 g an einer Achse mit einem Meter Radius bzw. 2 kg an 3,8 cm mit der angegebenen Drehzahl zu bewegen. Das ist in etwa die empirisch von uns ermittelte Handkraft. Im nachhinein hat sich herausgestellt, dass die Handkraft etwas grösser ist.

Desweiteren hat sich gezeigt, daß diese maximale Kraft bei maximaler Drehkraft nicht allein entscheidend ist, sondern die Kraftkurve im Verhältnis zu Drehzahl, wie oben gezeigt, ebenfalls wichtig ist. Leider sinkt die Kraft relativ stark bei kleineren Drehzahlen, so daß beim Abbremsen oder Gegensteuern des Balles mittels Handkraft deutlich sinkt.

Wir benötigen eher geringere Drehzahlen (1052 U/min = 17 U/s) bei höherem Drehmoment. Bei diesen Motoren kann das Drehmoment leider nicht ganz durch niedrigere Drehzahlen kompensiert werden.

Für diesen Zweck eine kleine Skizze:



- **2 Antriebsrädchen**

Wider Erwartung existieren keine standardisierten Achsendurchmesser oder Aufsätze, wie z.B. Antriebs- oder Zahnräder. Diese werden in der Regel individuell Produziert und sind deshalb so gut wie nicht zu finden. Deshalb mussten auch wir uns die Antriebsrädchen selbst bauen. Diese mussten absolut genau gearbeitet werden, d.h. insbesondere unwuchtfrei sein. Ausserdem musste die Verbindung mit dem Motor derart sein, daß die Rädchen möglichst austauschbar sind.

Nach diversen Versuchen der manuellen Herstellung dieser Rädchen aus Plastik oder Holz war klar, daß ohne maschinelle Hilfe keine unwuchtfreien Rädchen herstellbar waren.

Diese maschinelle Hilfe bekamen wir durch die Drehbank der Metallwerkstatt der Fakultät Gestaltung. Der Metallarbeiter Herr Spitze erklärte sich freundlicherweise bereit, die Rädchen aus Aluminium für uns anzufertigen. Die Anfertigungsskizze ist im Anhang vorzufinden.

Um die Rädchen fest auf die Motorachsen aufzusetzen, boten sich grundsätzlich vier Möglichkeiten:

- festkleben (von uns realisiert mittels 2-Komponenten-Kleber)
- Anpassen der Durchmesser von Motor und Rädchen, so dass durch den Druck der Anpassgenauigkeit ein Durchrutschen auch bei der zu erwartenden Maximalbelastung nicht eintritt. Das hat leider bei uns im ersten Anlauf nicht funktioniert und wir wollten Herrn Spitze nicht noch zu einer Neuanfertigung auffordern.
- Angeschliffene Motorachse mit entsprechendem Gegenstück am Rädchen. Diese Möglichkeit war technisch nicht realisierbar.
- Gewinde auf der Motorachse. Das war ebenfalls technisch nicht realisierbar.

- **2 Antriebsgummis**

Die Oberfläche der Rädchen muss geeignet sein, den Trackball zuverlässig anzutreiben.

Dafür müssen folgende Eigenschaften gewährleistet sein:

- kein Durchrutschen
- keine Zerstörung des Trackballs
- keine Deformation der Rädchenoberfläche (das käme einer Unwucht gleich)

Ein Gummiring schien diese Eigenschaften zu erfüllen, jedoch gestaltete sich die Suche nach einem solchen Ring als nicht einfach. Wir hatten erwartet, geeignete Ringe im Modellbauhandel, der Spielzeugabteilung oder in Elektronik- oder Mechanikteile-Grosshandel oder im Baumarkt zu finden; vergeblich.

Wir hatten schließlich Erfolg bei einem Weimarer Handwerksgeschäft, das zahlreiche Kleinteile und somit auch Dichtungsgummis kleinen Durchmessers anbot.

- **2 Zugrelais/Hubmagneten**

Ziel dieser Relais sollte es sein, je eine der beiden Achsen des Trackballs gelegentlich zu sperren, wie der Animation zu entnehmen ist.

Weitere Möglichkeiten, das zu realisieren könnte die Verwendung von Permanentmagneten, Motoren, Federn und Einrastmechanismen, auch in Kombinationen sein. Wer schon einmal einen Walkman o.ä. von innen angesehen hat, weiss, dass hier viele Möglichkeiten vorhanden sind, weshalb ich nicht näher auf diese Entscheidung eingehe. Es sei nur soviel gesagt: Die Hubmagneten sind am einfachsten einzubauen und bieten bei akzeptabler Größe akzeptable Kraft und akzeptablen Stromverbrauch.

Es gab zwei Hauptprobleme:

- Die den Trackball berührende Fläche sollte zuverlässig mit einem gegenüberliegenden Gegenpart eine Achse bilden, um welche sich der Trackball leicht drehen kann. Der Durchmesser der Fläche durfte also ebenso wie der Druck auf den Ball nicht zu gross sein, denn sonst wäre der Ball unbeweglich. Ist die Fläche zu klein (z.B. Nadelspitze), wird der Trackball deformiert. Ist die Kraft zu klein, kann die Zuverlässigkeit der Achse nicht garantiert werden (Durchrutschen möglich).
- Die Kraft des Zugrelais nimmt mit dem Abstand des Elektromagneten mit dem Zugstift quadratisch ab. Hier gilt das coulombsche Gesetz zur Anziehungskraft zweier Punktladungen. Da die Zugrelais als Spule mit einem Stift in ihrer Mitte, der durch das Magnetfeld in eine Richtung gezogen wird, realisiert sind, kann durch unterschiedliche Maße von Spulen- bzw.

Stiftlänge, -dicke, -material und -position eine leicht unterschiedliche Kraft bei gegebenem Abstand vorliegen, was aber das Problem prinzipiell nicht löst. Praktisch reagiert ein solches Zugrelais wie ein Magnet oder ein Schalter: Ist es nicht geschlossen, besitzt der Stift fast keine Kraft, ist er jedoch einmal nahe der Schliessposition, nimmt die Kraft quadratisch zum Abstand zu und der Stift „rastet ein“ und ist auch nicht mehr einfach zu öffnen.

Diese Eigenschaften erfordern ein sehr genaues Justieren des Relais.

Die wichtigsten Eigenschaften des Relais waren:

- ausreichende Kraft
- genügend hohe Schalthäufigkeit
- akzeptable Grösse (nicht zu gross)
- nicht zu hoher Stromverbrauch
- akzeptabler Preis

Wir fanden nicht viele Zugrelais vor, jedoch relativ schnell ein fast geeignetes Relais.

Die Kraft des Relais beträgt bei Maximalstrom (5-9 V) in geschlossenen Zustand ca. 1 Newton. Das entspricht ungefähr der Kraft von 100 Gramm Gewicht auf der Erdoberfläche. In der Praxis hat sich diese Kraft als ein wenig, jedoch nicht ZU schwach herausgestellt.

Eine Schalthäufigkeit von 33 Zyklen pro Minute war ebenfalls akzeptabel.

Die Grösse war akzeptabel (ca. 2*2*5 cm). Das Problem der erhältlichen Relais war, daß mit zunehmender Kraft auch sehr schnell die Größe der Bauteile ansteigt, was zu einem inakzeptabel grossen und schweren Gesamtgerät führt.

Der Preis war mit ca. 12 € pro Stück ebenfalls akzeptabel und der theoretisch zu erwartende Stromverbrauch von ca. 1 Ampere wurde bei weitem nicht erreicht (nur ca. 300 A), so daß hier ebenfalls keine Bedenken vorlagen.

Eine Produktbeschreibung und ein Datenblatt liegen im Anhang vor.

- **2 Taster**

Da wir für den Force-Feedback-Trackball ein eigenes Chassis herstellten, mussten die Tasten des Trackballs durch eigene Taster ersetzt werden.

Taster sind bei den gängigen Elektronikgrosshändlern in verschiedenen Ausführungen verfügbar.

- **1 Mikrocontroller (Atmel ATMega32)**

Der MC wird im Kapitel „Der Mikrocontroller“ eingehend besprochen.

Wir haben uns von vorneherein für diesen MC entschieden, da er keine oder nur wenige Wünsche offenlässt.

Die Aufgaben des MC werden im Kapitel „Die Software“ besprochen.

- **Evaluationsboard**

Wir haben den MC nicht direkt in das Gerät eingebaut, sondern Kommunizieren über das Board. Die Gründe hierfür sind die besseren Testmöglichkeiten sowie besonders die Bereitstellung verschiedener von uns benötigter Bauelemente, die wir nicht ohne erheblichen Mehraufwand hätten selbstbauen können, was auch nicht zum Forschungsschwerpunkt gehört hätte. Diese Bauelemente waren die Open-Drain-Ausgänge des Evaluationsboard, welche die für die Relais benötigten Ströme liefern, sowie die Motortreiber, welche die für die Motoren benötigten Ströme liefern, sowie die Anschlussmöglichkeit des UART-Adapters und der Programmierschnittstelle.

Ein Funktionieren des Gerätes ohne Evaluationsboard ist natürlich der erstrebenswerte Endzustand. Wie die Versorgung von Bauteilen mit grösseren Strömen bei gleichzeitiger Steuerung dieser Ströme durch den MC möglich ist, erfahren Sie in diesem Vortrag.

Wie Sie das UART und die ISP-Programmierschnittstelle direkt an den MC anschliessen, wird im Kapitel „Der

Mikrocontroller“ besprochen, ebenso wie das Evaluationsboard selbst.

- **UART-Adapter**

Der UART-Adapter wurde uns von Markus Borst der Firma Albotronic bereitgestellt und kann an das ebenfalls von ihm bereitgestellte Evaluationsboard mit einem seriellen Kabel und somit den Computer mit dem MC zwecks Kommunikation verbinden. Wie man einen solchen Adapter selbst baut, erfahren Sie im Abschnitt „Der Mikrocontroller“.

- **serielles Kabel**

Das serielle Kabel wird benötigt, um den Computer mit dem UART-Adapter zu verbinden, so daß Applikationen mit dem Trackball kommunizieren können.

- **Programmieradapter**

Der Programmieradapter verbindet lediglich das Parallelkabel mit dem Evaluationsboard und wurde ebenfalls von Marcus Borst bereitgestellt. Wie Sie sich einen solchen Adapter selber bauen, erfahren Sie im Abschnitt „Der Microcontroller“.

- **Parallelkabel (Downloadkabel)**

Das Parallelkabel verbindet den Computer mit dem Programmieradapter und somit mit dem MC, so daß der MC programmiert werden kann.

- **2 Motortreiberplatinen**

Das Evaluationsboard von Marcus Borst stellt zwei Anschlüsse für ebenfalls von ihm bereitgestellte Motortreiberplatinen mit 700 mA bzw. 2 A Stromversorgung für den Anschluss von Motoren mit entsprechend hohem Stromverbrauch zur Verfügung.

Schwächere Motoren wären nicht in Kauf zu nehmen gewesen.

- **Lötlitze (Kabel) & Steckverbinder**

Lötlitze sind einfache dünne Stromkabel zum führen von niedrigen Strömen (z.B. Steuerspannungen). Sie verbinden das Evaluationsboard mit den Peripheriegeräten.

- **4 Federn**

Die Federn haben wir für das Zurücksetzen der Zugrelais nach dem Einschaltzustand sowie für die Justierung des Motordrucks auf den Trackball verwandt (siehe Photo).

Diese Justierungsmöglichkeit ist in der Entwicklungsphase wichtig. Bei zu hohem Druck reicht die Kraft der Motoren nicht mehr aus, den Ball zu drehen. Der Ball wird eingeklemmt. Bei zu niedrigem Druck rutschen die Antriebsrädchen durch.

Die Federn müssen geeignete Kraft und Masse aufweisen. Federsets sind bei Modellbauhändlern oder Elektronik- & Mechanikgrosshändlern erhältlich, aber oft sehr teuer.

- **2 Rohrschellen**

Die Rohrschellen haben wir ebenfalls für die Justierung des Motordrucks auf den Trackball verwandt. (siehe Photo).

- **Chassis**

Die Bauelemente waren natürlich nicht in dem Originalchassis des Trackballs unterzubringen, weshalb wir einen grosszügiges Chassis gebaut haben, besser gesagt einen Holzkasten. Es war eine nicht zu unterschätzende Bastelei, alle Bauteile, insbesondere natürlich die Motoren und Relais mit Justierungsmöglichkeit in einem Chassis unterzubringen.

- **2 Transformatoren zur Stromversorgung**

Hier können handelsübliche Transformatoren verwandt werden. Sie müssen über geeignete Steckverbindungen zum UART-Adapter bzw. zum Evaluation-Board verfügen, was üblicherweise durch ein beigefügtes Adapterset gewährleistet wird. Ausserdem müssen Polung und Spannung im Bereich von circa 5-9 Volt sowie die Polung regelbar sein, was ebenfalls absolut üblich ist und somit kein Problem darstellen sollte.

Die Trafos sollten die geforderte Menge Strom liefen können, welche sich aus dem maximalen Stromverbrauch der einzelnen Komponenten berechnet. Das wären für unseren Anwendungsfall in etwa 2*300 mA für die beiden Relais und 2*600 mA für die Motoren bei angenommenen 9V Stromversorgung, also insgesamt gute 2 Ampere.

Die Qualität der Trafos entscheidet außerdem über die Konstanz des gelieferten Stroms bei unterschiedlichen Abnahmemengen. Beim Einschalten des zweiten Motors darf beispielsweise nicht die Stromversorgung des ersten Motors geschwächt werden.

3.3. Aufbau und Schaltplan des Force-Feedback Trackballs

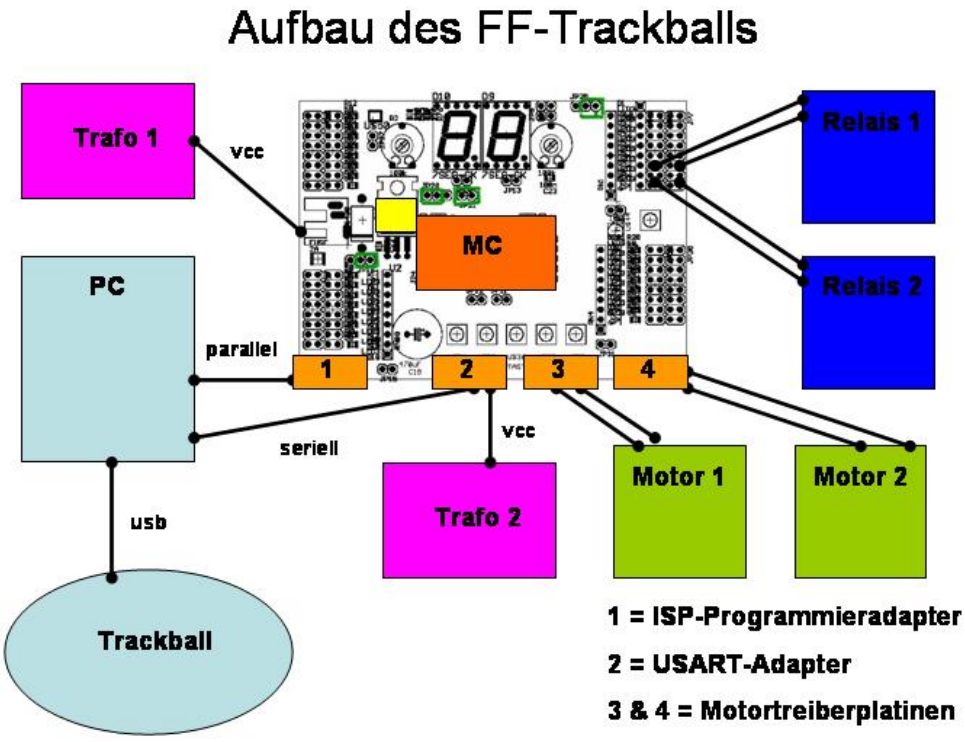
Hinweis: Beachten Sie bitte UNBEDINGT die unter ACHTUNG aufgeführten Bemerkungen, um eine Beschädigung der Hardware zu vermeiden !

- Das Maus-Tracking läuft wie gewöhnlich über USB, d.h. das USB-Kabel wird an den Computer angeschlossen.
- Die Zugrelais brauchen theoretisch fast 1 Ampere! Aber: Praktisch haben wir nur ca. 250 Ampere gemessen. Deshalb können wir Diese an die Open-Drain-Ausgänge (Einen an die mögliche 300 mA-Summe und Einen an den separaten 700 mA-Ausgang) schalten:

  Zugrelais an Pins C5 & C7

  Es wird jeweils ein Kabel an Masse gelegt und das Andere an Spannung, also an den äussersten Pin (Spannung) und an den zweiten Pin von innen (Masse). Sehen Sie sich die Skizze unten an, falls Sie Zweifel haben. Die Polung ist beliebig.

  Welches Relais an Welchen der beiden Pins



angeschlossen werden muss, wurde noch nicht festgelegt und kann einfach durch ausprobieren herausgefunden werden.

- Die Motoren werden an die Motortreiber-platinen angeschlossen, welche wiederum an das Evaluation Board angeschlossen werden. Die Polung wurde noch nicht festgelegt und muss herausgefunden werden (also die Drehrichtung der Motoren)!
- Das Evaluationsboard muss natürlich mit Strom versorgt werden, und kann mit 5-12 Volt betrieben werden. Als Ideal hat sich bei uns ca. 9V Versorgung herausgestellt.

- Jumpersettings:
  - Die grün eingerahmten Jumper müssen wie eingezeichnet gesetzt werden.
  - nicht markierte Jumper können beliebig gesetzt werden
  - beachten Sie allerdings, daß zusätzlicher Stromverbrauch (z.B. LEDs) die Ausgangsspannung anderer Komponenten senkt !

- Das UART wird über die RS232-Adapterplatine angeschlossen und benötigt eine separate Stromversorgung, welche auch mit entsprechend gesetztem Jumper 19 das Board gleich mitversorgen kann. Wir haben jedoch die Erfahrung gemacht, dass diese Variante unzuverlässige oder keine Datenübertragung mit sich bringt.

  Am Computer wird das UART mit der seriellen Schnittstelle verbunden. Wie Sie sich eine RS232-Adapterplatine selbst bauen können, erfahren Sie auf www.mikrocontroller.net, aber es liegt auch ein Schaltplan im Anhang vor. Durch die Nutzung des UART werden die Pins PC0 und PC1 belegt und können nicht anderweitig verwandt werden.
- Der Download des Programms für den Mikrocontroller erfolgt über den Anschluss JP32 auf der Skizze. Wie Sie sich eine ISP-Schnittstelle selbst bauen können, erfahren Sie auf www.mikrocontroller.net, aber es liegt auch ein Schaltplan im Anhang vor.

  Ebenfalls sind solche Schnittstellen erwerblich. Das Download-Kabel schließen Sie am besten an den Parallelport an.

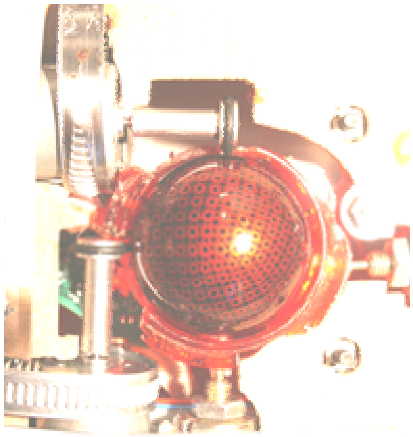
Wichtige Hinweise – bitte unbedingt vor Inbetriebnahme lesen!

- Es muss beachtet werden, dass der angeschlossene Trafo die geforderte Gesamtleistung erbringen kann (Ampere), so dass Dieser nicht zerstört wird.
- Die Versorgungsspannung des Trafos muss eventuell erhöht werden, d.h. bis zu 9V. Sonst bekommen die Motoren nicht genug Strom und reagieren nicht. Dem Chip ist auf dem Evaluationsboard ein Spannungsregler vorgeschaltet, so daß dieser nicht überlastet werden kann. Der Spannungsregler (auf Skizze gelb eingefärbt) sollte jedoch hin und wieder auf Hitze oder Geruchsbildung geprüft werden.
- Das gilt ebenfalls für die Motortreiber-Platinen, auch wenn Sie durch eine 700mA- bzw. 2 A- Sicherung geschützt sind.
- Das gilt weiterhin für die Motoren selbst, welche nicht über längere Zeit mit (mehr als) Maximalleistung betrieben werden sollten.
- Es kann an die Ausgänge der Motortreiber-Platinen aufgrund von Produktionsungenauigkeiten unterschiedlich viel Spannung geliefert werden, weshalb man Dieses per Software ausgleichen sollte. Das sollte natürlich nur beim Prototypen passieren (Ausgleich von Hardwareschwächen durch Software).
- Die Motoren dürfen nicht über einen längeren Zeitpunkt blockieren, da sie sonst zerstört werden. Für diesen Zweck gibt es spezielle Motoren mit sogenannter "Rutschkupplung", die aber hier nicht verwendet werden.
- Die Zugrelais dürfen nicht überlastet werden, was in unserem Anwendungsfall aber sowieso unmöglich ist, da die Open-Drain-Ausgänge viel zu wenig Strom abgeben können.

- Die Open-Drain-Ausgänge dürfen nicht überlastet werden. Das heist Pin C7 darf nicht mehr als 500 Milliampere und die restlichen Pins von Port C dürfen insgesamt nicht mehr als 300 Milliampere abgeben.
- Grundsätzlich gilt: Im Zweifelsfalle messen Sie mit einem Multimeter nach !
- Folgende weitere Pins sind belegt und können nicht anderweitig belegt werden:

  C3/C4 - Motoren an/aus

  D2/D4 - Motoren Richtung



3.4. Erfahrungen, Probleme, Verbesserungen

In diesem Kapitel werden die wichtigsten Erfahrungen und Probleme dargestellt und Verbesserungsmöglichkeiten aufgezeigt.

Man sollte über die folgenden Qualifikationen verfügen, wenn man sich an ein solches Projekt heranwagt:

- **Fachwissen** – Wie man am Beispiel der Motoren sieht, muss elementares, aber fundiertes Wissen der Mechanik und der Elektrotechnik vorhanden sein. Auch müssen die wesentlichen Eigenschaften von Bauteilen und ihre Eignung den Produktbeschreibungen und Datenblättern entnommen werden können. Oft sind diese Unterlagen auf ein Fachpublikum ausgelegt.

- **Kreativität** – Es gibt nicht nur einen Weg, ein Gerät zusammenzubauen, sondern Viele. Hier muss man sich etwas einfallen lassen, wenn es darum geht, ein Bauteil zu beschaffen oder herzustellen. Oft sind Diese nicht in erwarteter Weise am Markt vorhanden. Es gibt auch nicht immer eine einfache Lösung zur Realisierung eines Geräteteils. Das ist der Grund, warum Firmen in ihren Entwicklungsabteilungen sehr viel selbst entwickeln und nicht bei Drittanbietern einkaufen. Die Elemente müssen eben auch rein räumlich passen.

  Als kleiner Vorgeschmack nur eine kleine Liste der von uns besuchten Lokalitäten:

  - Kieferorthopäde – Beschaffung geeigneter Gummiringe
  - Modellbauladen
  - Spielzeugladen
  - Baumarkt
  - Ausbau alter Geräte vom Sperrmüll
  - Bastelladen
  - Metallwerkstatt der Fakultät Gestaltung

Für die Suche nach geeigneten Komponenten sollte genügend Zeit eingeplant werden.

- **Experimentierfreudigkeit & Geld** – Oft kann man nur durch Experimentieren herausfinden, welches konkrete Bauteil sich für einen bestimmten Zweck eignet. Ein gutes Beispiel ist die bereits erwähnte Motorcharakteristik. Günstigstenfalls werden verschiedene Bauteile selben Typs bestellt oder gebastelt und getestet.

- **handwerkliches Geschick** – Es ist Geduld für den Einbau der Geräteteile gefordert und es sollte stets eine mögliche Kalibrierung oder Justierung bedacht werden, ebenso wie der Austausch von Elementen.

- **vorrausschauendes Handeln** – Es ist ärgerlich, wenn das ganze Chassis neu gebaut werden muss (was viel umständlicher sein kann, als man es sich zunächst vorstellt), nur weil ein anderer Motor verwendet oder sogar nur getestet werden soll. Ausserdem ist zu beachten, daß der Projektbetrieb mit einem durchgebrannten Motor ausfällt, bis ein neuer Motor eintrifft. Es sollten also Ersatzbauteile im Vorhinein besorgt werden. Es sollte sorgfältig darauf geachtet werden, daß alle benötigten Teile bestellt werden, da das Nachbestellen billiger Einzelteile oft aufgrund von Mindestbestellwerten nicht möglich ist. Auch sollte die Verantwortlichkeit für die Bauteilverwaltung klar sein.

Probleme:

Die bauteilspezifischen Probleme wurden schon in den jeweiligen Abschnitten besprochen.

Hier werden Probleme angesprochen, die im Zusammenhang mit dem Zusammenwirken der einzelnen Hardwarekomponenten zu tun haben, angesprochen.

- Das grösste Problem war mit Sicherheit das präzise Justieren von Elementen bei gleichzeitiger Berücksichtigung von Justierungsvorrichtungen und der Möglichkeit des Austausches von Elementen gleichen Typs oder sogar des Einbaus alternativer Elemente (z.B. anderer Motortyp). Es hat sehr lange gedauert, das Chassis zu bauen. Ein Neubau wäre zeitlich nicht verantwortbar gewesen, ebenso wie die Zerstörung und somit Nachbestellung anderer Bauteile, welche Aufwendig wieder eingebaut werden müssen.

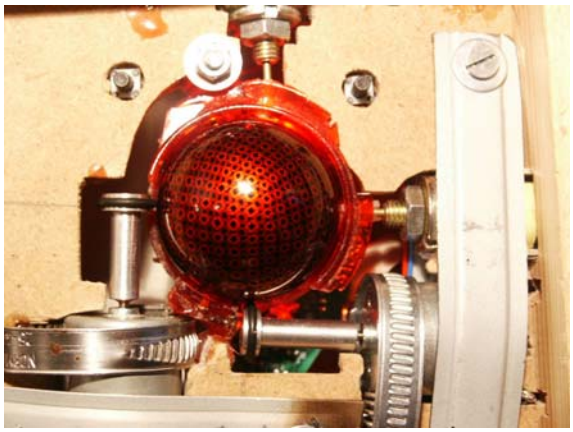
Beispiele für geforderte Präzision sind:

- Die Motoren müssen mit den jeweils gegenüberliegenden Relais eine Achse bilden, die exakt durch den Mittelpunkt des Trackballs geht
- Der Anpressdruck der Motoren an den Trackball muss sehr fein kalibrierbar sein.
- Das Gleiche gilt für die Relais.
- Aufgrund dessen muss das Grundgerüst, welches den Trackball und seine Fassung hält, sehr starr und stabil sein.

- Im Laufe der Zeit verschleißen einzelne Bauteile durch häufigen Ein- & Ausbau o.ä. Zum Beispiel mussten Kabel oft nachgelötet werden. Auch das Plastikchassis der Motoren litt unter der Hitze der Lötvorgänge. Schraubengewinde leiern aus etc.
- Da das von der Firma Albotronic bereitgestellte Evaluationsboard ebenso wie seine Zusatzkomponenten sich erst im prototypischen Stadium befanden und

- manuell hergestellt wurden, waren auch hier sowohl Design-, als auch Verarbeitungsfehler nicht auszuschließen. Die Steckverbindungen z.B. des UART oder der Motortreiberplatinen erwiesen sich oft als unzuverlässig.

Unser Trackball in der Aufsicht



Verbesserungsvorschläge:

Hinweis: Da uns die Kommunikation über das UART das Genick gebrochen hat (siehe Abschnitt „Software – Probleme“), können wir nur begrenzt Aussagen über die Einsatztauglichkeit des Gerätes machen. Dennoch gibt es einige Verbesserungsvorschläge.

- Es sollten weiterhin geeignetere Motoren gesucht und getestet werden. Eine Anfrage an Dr. Schatter könnte neue Kontakte zu Fachleuten eröffnen.
- Das gleiche gilt für Relais.
- Der Microcontroller könnte die Trackingdaten des Trackballs (vor oder nach der Konvertierung in USB-Daten) direkt einlesen, anstatt sie über den PC auszulesen. Auch wäre ein eigenes optisches Trackingverfahren denkbar.
- Da wir uns noch im prototypischen Stadium befinden, ist ein abschliessendes Chassis-Design angebracht.

4. Designprinzipien

4.1. Die erste Idee

Unser erstes Implementationsdesign basierte auf dem Konzept einer sogenannten „state machine“.
Der Trackball sollte sich also stets in einem Zustand befinden, aus welchem er die Ansteuerung der Aktoren, d.h. Motoren und Relais berechnete.
Beispielsweise könnte sich der Ball auf einer Schräge mit 5 Prozent Gefälle entlang der Längsachse und 3 Prozent Gefälle entlang der Querachse befinden. Entsprechend würden die Motoren den Ball permanent in Richtung des Gefälles bewegen. Der Nutzer muss dann den Ball den virtuellen Berg hinaufschieben.

Grund für dieses Design waren zwei Tatsachen:

1. In der Natur bzw. in der VR befindet sich ein Gegenstand ebenfalls zu einem bestimmten Zeitpunkt stets in einem bestimmten Zustand, welcher durch eine Menge von Eigenschaften des Objekts (bspw. Gewicht, Material, Geschwindigkeit, Richtung, Spin etc.) und eine Menge von äusseren Einflüssen (bspw. Wind, Untergrundbeschaffenheit, Gefälle) beschrieben wird. Eine Anpassung an dieses Prinzip mit der stückweisen Erweiterung neuer Features zur realistischeren Abbildung der Natur schien erfolgversprechend.
2. Durch dieses Prinzip kann viel Kommunikation zwischen Applikation und Eingabegerät gespart werden, da lediglich Zustandswechsel mitgeteilt werden müssen.

Allerdings hat sich im Laufe der Implementation herausgestellt, dass dieses Design eklatante Nachteile aufweist:

Wohl entscheidenster Nachteil für die Verwerfung des Designs war die Tatsache, dass diese „state machine“ nichts anderes ist, als die Implementation einer weiteren, separaten VR auf dem Mikrocontroller. Die VR des Computers teilt der MC-VR dann lediglich einen Zustandswechsel mit, woraufhin autarke Eigenberechnungen zur Ermittlung der notwendigen Aktorenansteuerung erfolgen.

Das Problem hieran ist, dass die Ergebnisse beider VRs nur sehr unwahrscheinlich und schon gar nicht garantiert gleiche Resultate ergeben, speziell nicht im Zeitbereich. Es kann also nicht die gleiche Verhaltensweise für beide VRs und deren Objekte (Bälle) garantiert werden.

Ein einfaches Beispiel:

Wenn der Ball auf einer Ebene eine bestimmte Geschwindigkeit in eine bestimmte Richtung besitzt und nun ausrollt, kann die Zeit bis zum endgültigen Stillstand in beiden VRs variieren. Es gibt zahlreiche andere Beispiele.

Dieses Problem ist nur durch eine Art Synchronisation der VRs zu erreichen. Einen ähnlichen Ansatz hat unsere zweite Idee.

Ein weiteres Problem ist die Komplexität der VR. Es ist anzunehmen, dass die VR des Rechners relativ komplex ist, sofern es sich um eine professionelle VR handelt.
Soll diese in eine MC-VR abgebildet werden, so erfordert das ähnliche Komplexität, was sich in viel Rechenzeit, viel Programmspeicher und viel Implementationsaufwand bei hoher

Fehleranfälligkeit niederschlägt.
Angesichts der Tatsache, dass hier im Prinzip alle Berechnungen der ursprünglichen VR nur ein zweites Mal ausgeführt werden, der MC sehr langsam ist (8 MHz in unserem Falle) und nur wenig Programmspeicher bereitstellt (2 KByte), stellt sich die Frage, ob das wirklich die beste Lösung ist.
In der Praxis hat sich herausgestellt, dass der Programmieraufwand und der benötigte Programmspeicherplatz dieses Designprinzip als abwegig erscheinen lassen.

Die Häufigkeit der Zustandsänderungen lässt an der Einsparung der Kommunikation zwischen VR und Eingabegerät gegenüber dem zweiten Ansatz zweifeln.
Lässt man den Ball beispielsweise in einer Glasschüssel o.ä. hin- und herrollen, so hat man die Wahl, ständig den Zustand „Steigung" zu Ändern (hohe Kommunikationslast), oder aber die MC-VR ihre eigenen Berechnungen durchführen zu lassen, wobei die Synchronisation recht sicher verloren gehen wird, d.h. in diesem Falle wird der Trackball schneller oder langsamer hin- und herrollen.

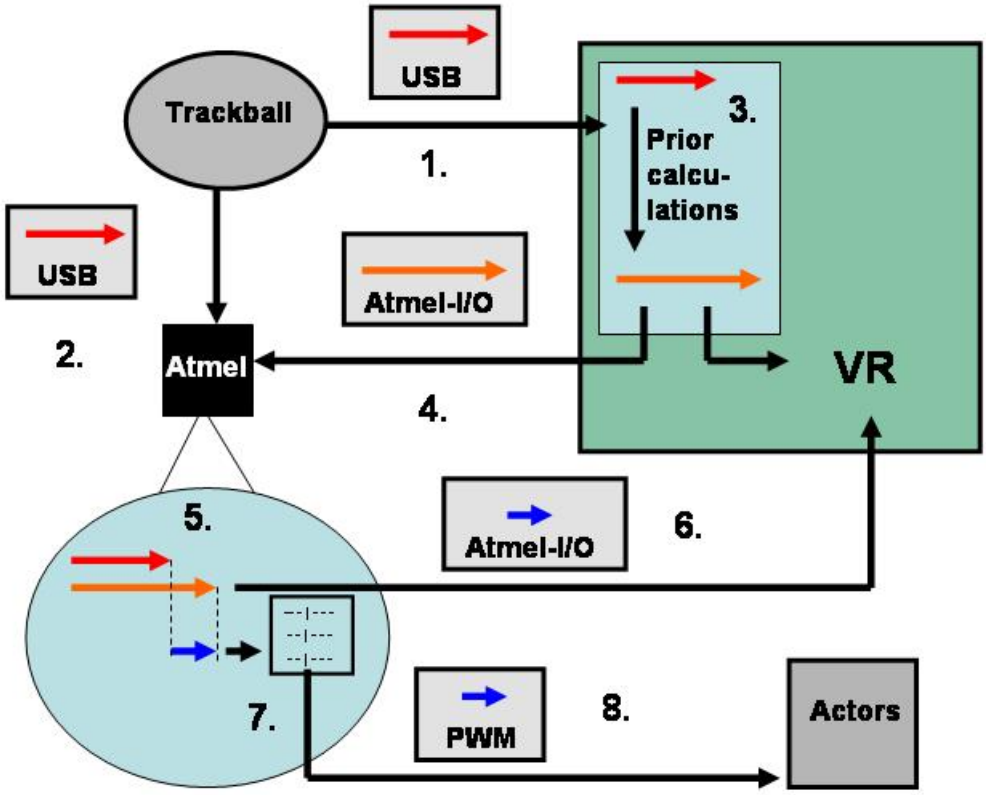
Dieses Design wurde im Verlaufe des Projektes aus oben genannten Gründen verworfen.
Eine nicht fertiggestellte Mikrocontrollerapplikation, die im Übrigen nicht in 2K Programmspeicher passen wird, ist dennoch im Quellcode-Paket enthalten.
Im Anhang liegt ausserdem das zugehörige Kommunikationsprotokoll vor.

4.2. Die bessere Idee

Die bessere Idee schien nach den vorangegangenen Überlegungen und Erfahrungen Folgende zu sein:

Es gibt nur eine VR, die alle Berechnungen durchführt und ständig Informationen an das FF-Gerät sendet, damit Dieses seine Bewegungen mit denen des virtuellen Objektes (Balles) synchronisieren kann.

Eine Skizze, die im Folgenden kommentiert wird, soll das verdeutlichen:



Die Schritte werden der Nummerierung nach ausgeführt:

- Der MC wird ständig darüber informiert, welchen Bewegungsvektor (also Betrag und Richtung der Bewegung) der Trackball gerade produziert, unabhängig davon, woher die Kraft gerade kommt (Hand, Motoren). (2.) Das geschieht zur Zeit noch über einen PC-seitigen Loopback (siehe Abschnitt „Die Software – Der Universal Serial Bus (USB)“) und geschieht in Zukunft idealerweise geräteintern, was allerdings für diese Erläuterungen keine Konsequenzen hat.
- Dieselben Informationen werden natürlich auf gewohnte Weise an den Computer bzw. die VR gesendet. (1.)
- Die VR verarbeitet die eingehenden Informationen anschliessend. (3.) Beispielsweise könnte der virtuelle Ball noch ausrollen, wenn der User den Trackball gar nicht mehr bewegt. In diesem Fall würde der orange Vektor aus dem roten generiert werden und mit der Zeit immer kleiner werden, bis der Ball steht.
- Dieser berechnete Vektor wird dann zum Einen wie gewohnt in der VR benutzt (Ball bewegen), zum Anderen wird er über die serielle Schnittstelle (UART) im von uns entworfenen FF-Trackball-Kommunikationsprotokoll-Format zum MC gesendet (4.), und dort verarbeitet(5.):
- Die VR teilt dem Trackball auf diese Weise mit, wie sich das virtuelle Objekt (Ball) gerade bewegt. Der Trackball passt nun seine Bewegung an, indem er aus der Differenz des von ihm selbst produzierten Vektors und dem der VR berechnet, welchen Vektor seine Motoren hinzufügen müssen, damit die Vektoren (produzierter und der der VR) gleich sind.
- Da die Motoren über Pulsweitenmodulation angesteuert werden, verstehen sie natürlich das USB-Protokoll nicht, weshalb Dieses in die PWM transformiert werden muss.
  Da das Verhältnis von eingesetzter PWM-Stufe und daraus resultierendem USB-Vektor für jede Stufe individuell ist, kann hier keine Funktion zur Berechnung dienen. Stattdessen wird in einer Tabelle festgehalten, bei welcher PWM-Stufe, welcher USB-Vektor zu erwarten ist. Das wird für die beiden Achsen (d.h. Motoren) getrennt ermittelt. Nähere Informationen dazu folgen im Abschnitt „Die Software – Die Microcontrollerprogrammierung“. (7.)
  Es wird also anhand der Tabelle ermittelt, welche PWM-Stufe an die Motoren jeweils zu senden ist und die Motoren werden angesteuert. (8.)
- Damit ist es allerdings noch nicht getan, denn nun kommen wir zu einem force-feedback-typischen Problem:

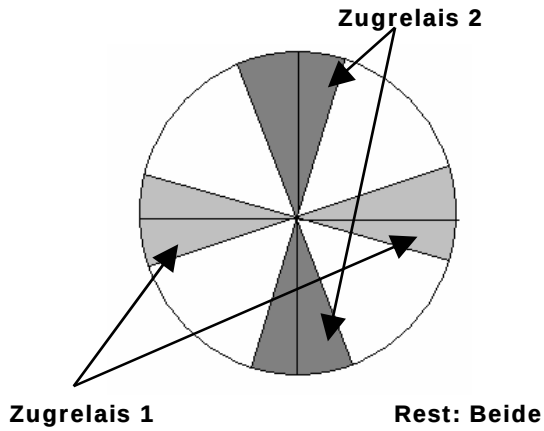
Die Rückkopplung

Der durch die Motoren produzierte Vektor veranlasst den virtuellen Ball zur Bewegung. In der Vorberechnung wird die Bewegung eventuell verstärkt, so dass ein größerer Vektor gleicher Richtung an den Trackball gesendet wird. Der Trackball versucht, mittels Motoreneinsatzes, den von sich produzierten Vektor anzugleichen und veranlasst den virtuellen Ball zu stärkerer Bewegung, woraufhin dieser den Trackball wiederum zu stärkerer Bewegung veranlasst usw.
Dieses Phänomen dürfte Ihnen spätestens jetzt als Rückkopplung bekannt sein.
Wir können dieses Problem beheben, indem Wir die VR den VON DEN MOTOREN produzierten Vektor ignorieren lassen.
Das tun wir, indem wir der VR permanent diesen Vektor mitteilen, d.h. als Antwort auf den zuvor von der VR gesendeten Vektor. (6.)
Ob das in der Praxis funktioniert, ist fraglich, denn die gesendeten Werte sind lediglich bei einer Kalibration ermittelte empirische Werte. Tatsächlich kann der durch die Motoren ermittelte USB-Vektor leicht abweichen, was eine Rückkopplung ermöglicht und den Ball somit zu einem perpetuum mobile machen kann. Hier müssen weitere Maßnahmen ergriffen werden, falls es weiterhin dazu kommt.
Ein bekanntes Phänomen soll die Schwere dieser Problematik verdeutlichen:
Wenn die Motoren den Ball mit sehr geringer Kraft antreiben, so, dass sie es gerade nicht schaffen, ihn zu bewegen, so genügt in der Regel ein kleiner Fingerstoss, um den Ball „ins Rollen zu bringen“, wie man so schön sagt. Ab diesem Zeitpunkt wird sich der Ball kontinuierlich bewegen,

bis ich ihn wieder stoppe.
Wir erhalten hier also zwei verschiedene USB-Vektoren für dieselbe PWM-Stufe (bzw. einmal überhaupt keinen USB-Vektor). Das ist natürlich absolut inakzeptabel.
Dieses Problem ist noch nicht geklärt und Bedarf dringend einer Behebung !

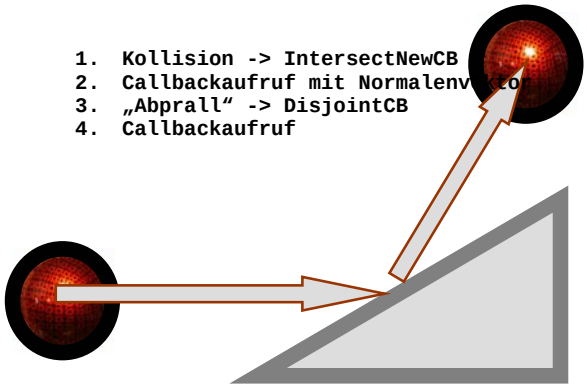
4.3. Kollision

Für die Kollision des Balles mit einem anderen Objekt sind die Relais zuständig; sie können die Bewegung des Trackballs entlang einer Achse oder auch völlig unterbinden.
Bei der Kollision gab es für uns zwei mögliche Ansätze. Im ersten Ansatz würde man ein physikalisches Objekt in der Simulation aus mehreren Objekten zusammen setzen, um zu wissen von wo aus die Kollision stattgefunden hat. Dies hätte zur Folge, dass man alle Körper in 5 Gruppen unterteilen würde (4 für die "normalen" Körper und 1 für den "schrägen" Körper, da die Reaktion, nämlich das Ansteuern beider Zugrelais, die gleiche ist) und jeder Gruppe eine entsprechende Callback-Funktion zuweisen würde. Dies hätte aber zur Folge, dass selbst einfache Quader aus mehreren Teilen bestehen würden. Dies könnte man durch zusätzliche Klassen lösen, die aus den Basiskörpern "komplexe" Strukturen zusammensetzen.
Der zweite Ansatz ist dabei durch aus generischer und wurde aus diesem Grund auch von uns implementiert. Es existiert dabei nur ein Callback für jegliche Art von Objekt, d.h. kollidiert der Ball mit einem Objekt, wird immer die gleiche Callback-Funktion aufgerufen. Man muss die Objekte also nicht in Gruppen einteilen und spezielle Funktionen schreiben.
Kommt es zu einer Kollision, wird an die Callback-Funktion, die im Feld "IntersectNewCB" des entsprechenden Objekts der Klasse "fpMECollisionTracker" steht, der Kollisionsvektor übergeben. Die Möglichkeit, diesen Kollisionsvektor auslesen zu können, musste (wie schon erwähnt) für uns noch in Vortex4Avango implementiert werden. Der Vektor wird dann an den Mikrokontroller übergeben (siehe Kommunikation in Avango) um dort die entsprechende Aktion zu berechnen. Aus den x-Werten und y-Werten (der Kollisionsvektor hat zusätzlich einen z-Wert) wird auf dem Mikrocontroller der Winkel in einem imaginären zweidimensionalen kartesischen Koordinatensystem berechnet. Je nachdem, wie groß der errechnete Wert ist, befindet er sich in einem von 8 Bereichen (siehe Grafik). In diesen Bereichen werden jeweils bestimmte Zugrelais aktiviert (durch Fallunterscheidung ermittelt).



Beispiel 1: Ist der ermittelte Winkel 20° befindet er sich in dem Bereich von 0° bis 22,5° und von 337,5° bis 360°. Die Kollision in der Simulation hat also auf horizontaler Ebene (je nach Betrachtungsweise) stattgefunden, wenn man sich die Szene von oben anschaut. Das hat dann entsprechend zur Folge, dass das Zugrelais, welches für das horizontale (ebenfalls je nach Betrachtungsweise) Blockierung des Trackballs zuständig ist, angesteuert wird.

Beispiel 2: Ist der ermittelte Winkel 30° befindet er sich in den Bereichen, die weder für "horizontale" Blockierung noch für "vertikale" Blockierung zuständig sind. Da die beiden ersten Bedingungen nicht zutreffen, werden beide Zugrelais aktiviert, da es sich dann um eine "diagonale Kollision handeln muss.



Befindet sich der Ball also zum Beispiel an einer Wand (bereits mit ihr kollidiert, berührt sie aber weiterhin), ist das entsprechende Zugrelais aktiviert. Eine Bewegung des Trackballs ist nur noch um die blockierte Achse möglich. Ist die Berührung mit der Wand beendet wird der "DisjointCB" aufgerufen und das jeweilige Relais deaktiviert. Sind beide Zugrelais aktiviert ist ja theoretisch keine Bewegung möglich, da beide Achsen blockiert sind. Allerdings ist der Spielraum groß genug um eine minimale Bewegung des Trackballs zu ermöglichen, was wiederum zur Folge hat, dass sich der virtuelle Ball in der Simulation bewegt.

4.4. Bewegung

Die Bewegung des Trackballs soll durch die Motoren erfolgen, wenn die Bewegung des virtuellen Balls sich von der Bewegung des Trackballs unterscheidet.

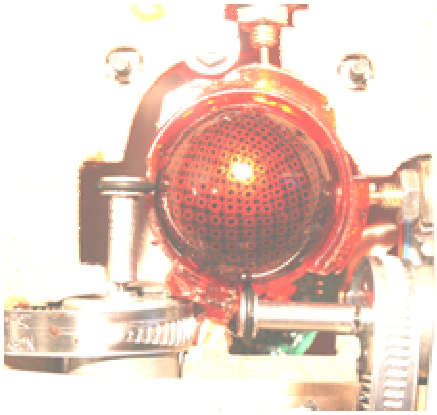
Beispiel 1: Bewegt man den virtuellen Ball durch den Trackball und lässt diesen dann los, bewegt sich der Ball in der Simulation weiter, da auf diesen noch Kräfte wirken, solange bis die Bewegungsdämpfung (durch Reibung) den Ball zum Stehen gebracht hat. In diesem Fall soll der Trackball mit der gleichen Geschwindigkeit und der gleichen Richtung durch die Motoren ähnlich dem "Ausrollen" des virtuellen Balls bewegt werden.

Beispiel 2: Wir bewegen den Ball eine Schräge hinauf. Der Bewegungsvektor des Trackings bzw. der Kraftvektor auf den Ball ist dann proportional größer als die eigentliche Bewegung des virtuellen Balls (wenn sie auf einer Ebene stattfinden würde). In diesem Fall sorgt die Differenz beider Vektoren für eine Gegenbewegung der Motoren.

Die Bewegungskomponente konnte leider nicht evaluiert werden, aber das Modell wäre wie folgt: Es erfolgt kontinuierlich ein Abgleich von Kraftvektor und dem eigentlichen Bewegungsvektor des Balls. Eine Bewegung des Trackballs ergibt einen Kraftvektor auf den virtuellen Ball. Bewegt man den Trackball nicht mehr und der Ball bewegt sich jedoch weiter, ist die Differenz zwischen den beiden Vektoren ungleich Null (bzw. einem Toleranzwert). Der sich ergebende Differenzvektor wird in entsprechende Motorbewegung umgesetzt. Eine vorherige Kalibierung der Werte ist notwendig. So weiß man, welche USB-Werte welche Bewegungswerte ergeben sollten.
Allerdings gibt es dabei ein Problem. Rollt zum Beispiel der Ball eine Schräge hinunter, ohne das der Trackball bewegt wird, ergibt dies eine Bewegung des Trackballs durch die Motoren in die entsprechende Richtung. Die Bewegung des Trackballs ergibt durch das Tracking allerdings wieder einen USB-Vektor in die gleiche Richtung. Die Folge wäre ein zusätzlicher Kraftvektor auf den Ball, wobei die Gefahr bestehen könnte, dass sich das System "aufschaukelt". Dies hätte sich aber in Tests ergeben und hätte entsprechend angepasst werden können. Es muss also stets der durch die Motoren erzeugte Kraftvektor von der interpretierenden VR ignoriert werden. Praktisch erfolgt dass, indem die VR dem Trackball kontinuierlich mitteilt, welchen USB-Vektor das System gerne vom Trackball bekommen würde. Der Trackball berechnet dann die Differenz zwischen dem gewünschten und dem tatsächlich erzeugten USB-Vektor und gleicht die Bewegung durch die Motoren an. Der durch die Motoren erzeugte USB-Vektor wird an die VR gemeldet, so daß dieser Vektor ignoriert werden kann und so ein Aufschaukeln des Systems vermieden wird.

4.4. Simulation von Grenzen

Hier soll kurz ein besonderes Phänomen erläutert werden, welches in unserem Anwendungsfalle Probleme bei der Simulation von Grenzen machen könnte, aber in anderen Fällen sicher wieder zum Vorschein kommen wird.
Wenn eine Grenze simuliert werden, beispielsweise eine Strasse, auf der der Ball rollt, von welcher er rechts und links nicht herunterrollen kann, so wirkt nach dem von uns bekannten physikalischen Prinzip der Gegenkraft stets dieselbe Kraft von der Grenze auf den Ball, wie umgekehrt.
Das bedeutet praktisch:
Wenn der Ball nicht gegen die Grenze gedrückt, also mit der Hand in die „verbotene“ Richtung bewegt wird, gibt es auch keine Gegenkraft.
Wird er in diese Richtung bewegt, gibt es eine Gegenkraft.
Wer diese letzten beiden Sätze scharfäugig gelesen hat, wird das Problem schon erkannt haben: Bewegung.
Ich kann eine Kraft nur durch die daraus resultierende Bewegung detektieren !
Das bedeutet, dass der Trackball auf eine Bewegung in eine „verbotene“ Richtung erst reagieren kann, wenn diese schon eingetreten ist und eventuell reagiert er noch darauf, wenn die Kraft gar nicht mehr vorhanden ist.
Es werden also nur „weiche“ Grenzen simuliert werden können, abhängig von der Reaktionsgeschwindigkeit des Gesamtsystems.



5. Die Software

5.1. Übersicht

Die von uns erstellte Software setzt sich im wesentlichen aus drei verschiedenen Komponenten zusammen:

- Mikrocontrollerapplikation
- beispielhafte VR-Umgebung
- Kommunikationsschnittstelle (VRPN)

Bei der Kommunikationsschnittstelle VRPN handelt es sich nicht um eine Notwendigkeit. Allerdings ist hierdurch eine Kommunikation (auch zeitgleich mehrerer) beliebiger Programme mit diesem Gerätetreiber auf Basis eines Protokolls möglich, so dass Diesen die Arbeit der Kommunikationsverwaltung abgenommen wird.

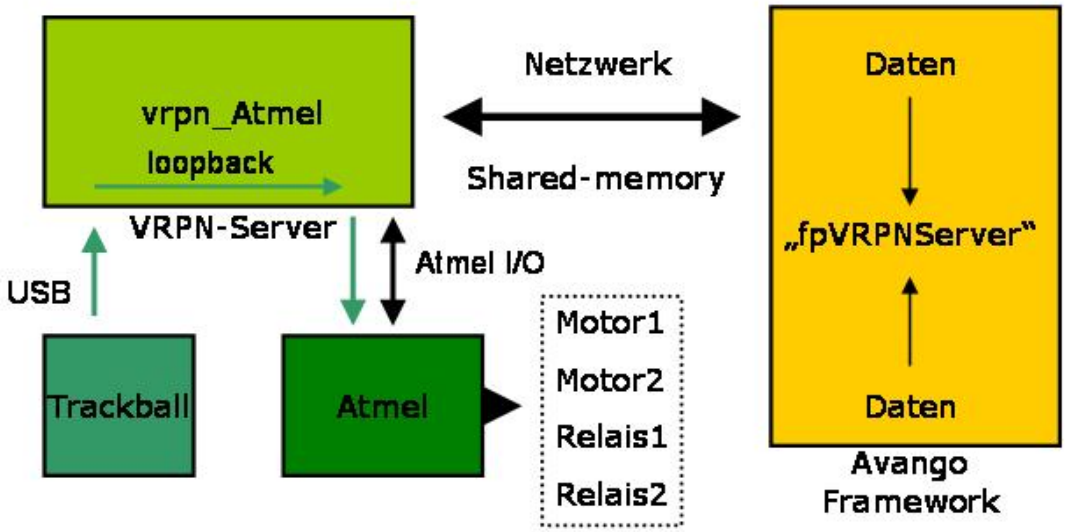
Ausserdem wird hierdurch eine rechnertransparente Kommunikation möglich, d.h. Eingabegerät und VR können an verschiedenen Rechnern betrieben werden. Das beruht auf einer Client-Server-Architektur dieses Treibers.

In der nebenstehenden Grafik ist die Kommunikation der einzelnen Komponenten dargestellt.
Im Folgenden werden die einzelnen Komponenten betrachtet.
Die Kommunikation mit den jeweiligen Komponenten wird dort ebenfalls besprochen.

Im Unterabschnitt „Der Universal Serial Bus (USB)“ wird ausserdem der in VRPN integrierte Loopback (grüne Pfeile in der nebenstehenden Grafik) besprochen, welcher die vom Trackball erzeugten USB-Daten an den Mikrocontroller zur Auswertung weiterleitet.

Das Zusammenspiel der einzelnen Softwarekomponenten

Die Software-Kommunikation



5.2. Anwendung in Avango und Vortex

Für den primären Anwendungsfall sollte eine Beispielanwendung für die Evaluierung des Gerätes in Avango erfolgen. Wir haben uns dazu entschlossen, eine Land-schaft nachzubilden, in der ein Ball manövriert werden kann. Eine Bewegung des Trackballs durch den Nutzer hat eine Bewegung des Balls in der Simulation zur Folge. In dieser recht ein-fachen Landschaft mit Bäumen und Brücken können Kollision und Bewegung gut umgesetzt werden. Die Logik ist relativ simpel und für den Nutzer (da sehr nah an der Realität) gut ver-ständlich. Die Verwendung des Balls ist eben-falls eine sehr gute Metapher. Nach der Divise "What you see ist what you get" (WYSIWYG) ist die Bedienung des Balls in der Simulation mit Hilfe des Trackballs sehr intuitiv.

Da wir bestimmte Ereignisse wie Kollision oder physikalisch korrekte Bewegung benötigten, hatten wir die Wahl, diese Routinen entweder selbst in Avango zu programmieren oder auf bereits bestehende Module zurückzugreifen. Eine dieser möglichen Module war die Physik-Engine Vortex. Diese bot uns alle notwendigen Routinen wie Kollision etc., also eine komplette, physikalisch korrekt simulierte virtuelle Umgebung. Während der Laufzeit wird - mit speziellen Algorithmen für die jeweiligen Körper – dauerhaft die Kollision aller physikalischen Objekte überwacht und die Bewegung einzelner Objekte berechnet.

```
;;Laden des Packets
(load-and-init-class "fpVortexPack")

;;Definierden der physikalischen Welt
(define phy-world (make-instance-by-name
"fpMEWorld"))
;; Anziehungskraft
(fp-set-value phy-world 'Gravity (make-vec3
0.0 0.0 -100))
;; Zeit muss vergehen
(fp-connect-from phy-world 'TimeIn time-
sensor 'Time)
...
;; Zeitintervalle nach jeder
Simulationsberechnung
(fp-set-value phy-world 'TimeStep 0.02)
(fp-set-value phy-world 'UseTimeStep 1)
;; die Simulation starten
(fp-set-value phy-world 'Running 1)
(fp-set-value phy-world 'Name "myworld")
;; Kontakt- und Reibungsparameter
(define contact-para #f)
(set! contact-para (-> phy-world 'create-
contact-params 0 0))
...
(fp-set-value contact-para 'FrictionType 2)
(fp-set-value contact-para 'Friction 1000)
(fp-set-value contact-para 'Restitution 1)
...
```

Objekte bestehen in dem Programm aus einem physikalischen Körper (Quader, Zylinder oder Kugel) und einer grafischen Repräsentation. Die physikalischen Objekte werden von der entsprechenden Klasse abgeleitet (phy-cube, phy-cylinder, phy-sphere), die vorher geladen werden müssen.

```
(load "scheme/phy-sphere.scm")
(load "scheme/phy-cube.scm")
(load "scheme/phy-cylinder.scm")
```

Avango ist ein Framework für interaktive verteilte virtuelle Umgebunden in Echtzeit und wurde am Frauenhofer Institut für Medienkommunikation entwickelt. Avango bietet eine Vielzahl an möglichen Eingabegeräten mit unterschiedlichen Freiheitsgraden (DOF) an und ebenfalls eine große Vielzahl ein Ausgabegeräten, vom einfachen Monitor über Stereoprojektionen bis zu „Cave“-Anwendungen. Es setzt auf OpenPerformer auf und ist in C++ programmiert (also objektorientiert), bietet aber zusätzlich ein Scripting Interface in der interpretierten Sprache Scheme an, wodurch noch während der Laufzeit Modifikationen an der Szene durchgeführt werden können („Rapid Prototyping“ möglich). Die räumliche objektorientierte Beschreibung der virtuellen Welt erfolgt in Avango durch einen so genannten Szenengraph, d.h. die einzelnen Objekte mit ihren speziellen Eigenschaften (repräsentiert durch „Fields“) werden als Knoten hierarchisch in einen azyklischen Graph einsortiert. Der Szenengraph spielt auch eine tragende Rolle in Bezug auf die Verteilung. Der Graph ist allen verteilten Teilnehmern zugänglich und wird über das Netzwerk versendet und konsistent gehalten, d.h. jeder Client hat eine lokale Kopie des Szenengraphen. Veränderungen in der lokalen Kopie eines Clienten werden dann per „Updatemessage“ über das Netzwerk an alle teilnehmenden Clienten versendet, so dass sie ihre lokale Version des Szenengraphs aktualisieren können. Die Verteilung erfolgt über ein „Peer2Peer“-Netzwerk.

Hat man ein Objekt erstellt, können bestimmte Methoden (je nach Klasse) auf dem Objekt ausgeführt werden. Dazu gehören Transformation des physikalischen Objekts und der Repräsentation:

```
;;Objekt der Klasse Phy-Cylinder
(define tree7 (make-instance phy-cylinder))
(define tree7_model_list (list 0 0 -3 .1 .1
.1 0 0 90))
;;Physikalisches Objekt transformieren und
Gewicht setzen
(with-instance tree7(::make-physical 8 10 3
```

So wird mit allen Objekten in der Anwendung vorgegangen. Bei den Bäumen kann man sehen, dass physikalisches Objekt und grafische Repräsentation nicht unbedingt überein stimmen müssen. Für die Kollisionsberechnung ist die Approxiamtion jedoch ausreichend.
Die Bewegung des Balls erfolgt noch getrennt von dem Mikrokontroller, nämlich einfach über USB direkt von dem Device für das optische Tracking. Dazu werden die USB-Daten ausgelesen. Dazu ist es notwending den entsprechenden Daemon zu starten und einen Trigger an den "Time-Sensor" zu hängen, damit ständig die USB-Werte ausgelesen werden. Aus diesen Werten wird dann ein Vektor berechnet. Dieser Vektor wirkt dann als skalierter Kraftvektor auf den Ball in der Simulation, der sich entsprechend des Trackballs bewegen soll. Da Kraft auf den Ball wirkt, bewegt sich dieser auch weiter wenn der Trackball keine Werte liefert, so lange, bis der physikalisch simulierte Ball " ausgerollt ist:

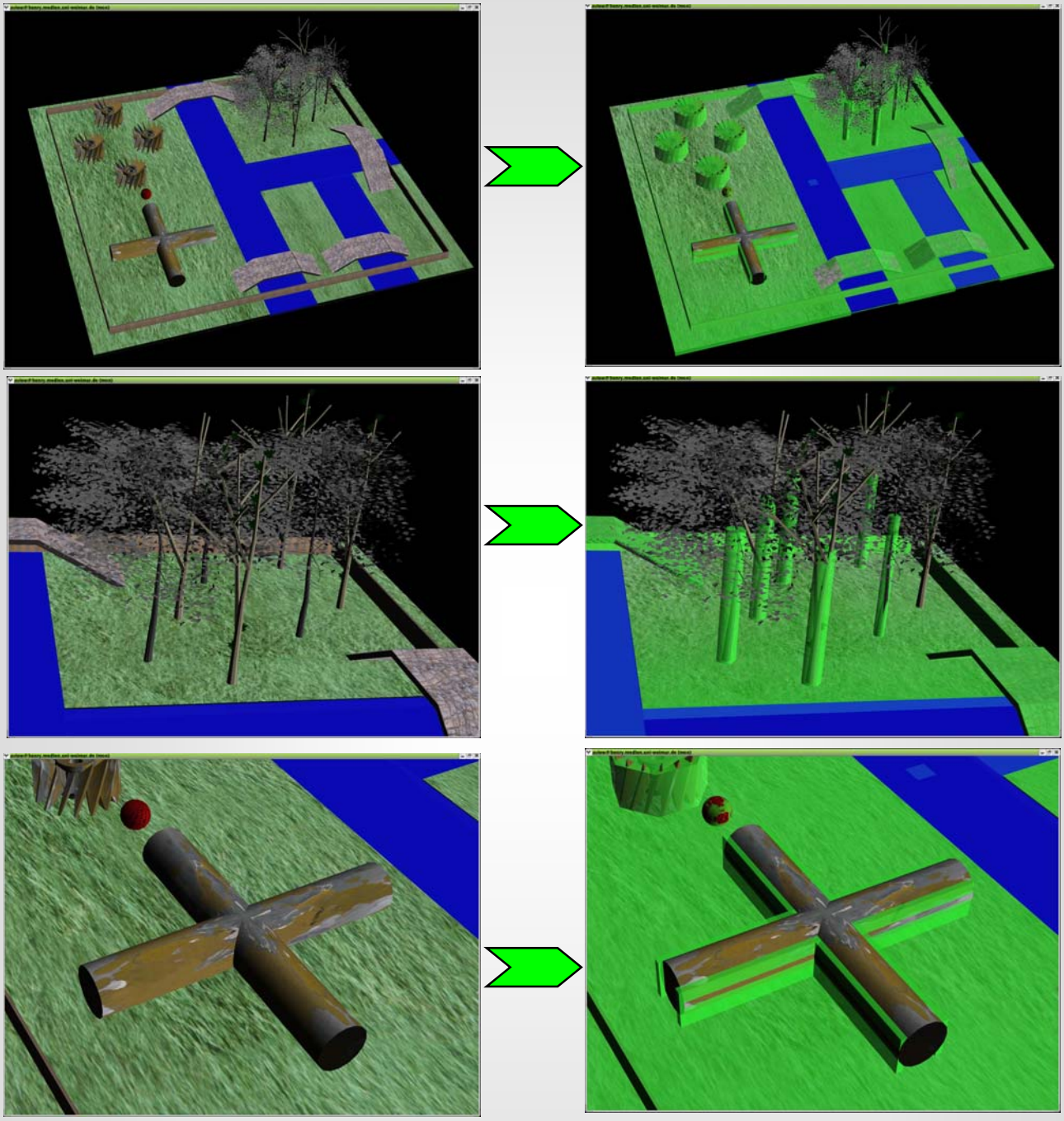
```
(define movevec2)
(define movevec2_old (make-vec2 0 0))

(define button0 0)(define button1 0)(define button2 0)(define button3 0)

(define device-service (make-instance-by-name "fpDeviceService"))
(-> device-service 'register-service "DeviceService")
(-> device-service 'connect-daemon)

(define dev0 (make-instance-by-name "fpADSensor"))
(fp-set-value dev0 'Station  "usb-station1")

;; Callback-Funktion fuer space-trigger --> liest andauernd Werte  von dev0 aus und uebt mit
diesen Kraft auf "ball1" aus
(define (dev0-cb trigger)
 (set! movevec2 (sub-vec2 movevec2_old (make-vec2 (fp-get-value dev0 'Value0)(fp-get-value dev0
'Value1))))
 (set! movevec2_old (make-vec2 (fp-get-value dev0 'Value0)(fp-get-value dev0 'Value1)))

 ;;(set! button0 (fp-get-value dev0 'Contact0))
 ;;(set! button1 (fp-get-value dev0 'Contact1))
 ;;(set! button2 (fp-get-value dev0 'Contact2))
 ;;(set! button3 (fp-get-value dev0 'Contact3))

 ;; Feintuning notwendig, u.U. mit log bzw.exp
 (with-instance ball1(::add-power(* bewegungsfaktor (vec2-ref movevec2 0))(* -1 (* bewegungsfaktor
(vec2-ref movevec2 1))) 0 0 0 0))
)
(define space-trigger (make-instance-by-name "fpTriggerCB"))
(fp-connect-from space-trigger 'Input time-sensor 'Time)
(fp-set-value space-trigger 'Callback dev0-cb)
```

Vortex ist eine vollständige Physik-Engine für Echtzeitsimulationen. Vortex stellt dazu Bibliotheken für die Dynamik und Kollisionserkennung von Festkörpern zur Verfügung. Darüber hinaus bietet es auch die Visualisierung durch OpenGl und DirectX an. Die Einbindung in andere Frameworks wie zum Beispiel Avango ist durch den stetigen Abgleich von physikalischer Simulation und grafische Repräsentation ohne weiteres möglich. Vortex arbeitet sowohl mit „primitiven" Objekten („Sphere", „Box", „Cones", …) als auch mit komplexen Objekten („ConvexMesh", …) zur Berechnung von Kollision. Die Dynamikberechnungen erfolgt mit Hilfe von Positions-, Orientierungs- und Geschwindigkeitseigenschaften der einzelnen Objekte auf so genannten primitiven „Rigid Bodies".

Objektrepräsentation in Vortex4Avango

Grafische Objektrepräsentation mit komplexen 3D-Modellen

Physikalische Repräsentation mit einfachen Körpern



In der Applikation besteht ein Objekt aus der sichtbaren Geometrie und dem nicht sichtbaren Körper. Die Geometrie dient der Visualisierung des Objekts, also das, was man eigentlich sieht. Dazu wird mit Hilfe des „fpLoadFile“-Knoten ein beliebiges Modell geladen, nach den Bedürfnissen transformiert und der Szene hinzugefügt. Der Körper ist die physikalische Repräsentation innerhalb der Simulation. Nur dieser Teil wird von der Vortex-Engine benutzt, um Kollision, Bewegung etc. des Objekts zu berechnen. Im Idealfall sind Geometrie und Körper deckungsgleich. Oft genügt es allerdings, Vereinfachungen der Geometrie zu nutzen, um die Engine nicht mit Berechnungen zu überlasten.
Die Bewegung von Objekten in physikalisch korrekt simulierten Umgebungen erfolgt nicht durch die Transformation der Geometrie, sondern durch das Wirken von spezifischen Kräften auf den Körper. Die Geometrie folgt dann dem Körper.

In Bezug auf Kollision ist es mit Vortex möglich, so genannte „Callback“-Funktionen für Objektpaare zu definieren, die aufgerufen werden, wenn die Kollision der jeweiligen zwei Objekte stattfindet. Dazu definiert man einen Knoten vom Typ "fpMECollisionTracker". Dieser Knoten hat u.a. folgende Felder:

1. 'Model1 -> 1.Objekt des Objektpaars
2. 'Model2 -> 2.Objekt
3. 'Active -> 1 = Aktiviert, 0 = Deaktiviert
4. 'IntersectNewCB
-> Definiert Callback-Funktion, die aufgerufen wird, WENN die beiden "Models" kollidieren
5. 'IntersectCB
-> Definiert Callback-Funktion, die aufgerufen wird, SOLANGE die beiden "Models" in Kontakt zu einander haben
6. 'DisjointCB
-> Definiert Callback-Funktion, die aufgerufen wird, NACHDEM sie „kollidiert haben“

In unserem Fall wird der Ball mit allen in den jeweiligen Listen befindlichen Objekten zu einem Objektpaar verbunden und die Felder "IntersectNewCB" und "DisjointCB" mit entsprechenden Funktionen belegt. In dem "IntersectNewCB"-Feld steht eine Funktion, die an den Atmel die Information sendet, dass eine Kollision zwischen dem Ball und einem beliebigen Objekt stattgefunden hat.
In Vortex selber kann bei der Kollision der Kollisionsvektor ausgegeben werden. In Vortex4Avango ("gewrapptes" Modul) stand die Möglichkeit nicht zur Verfügung, wurde aber auf unseren Wunsch hin implementiert. Mit Hilfe des Kollisions-vektors, der an den Mikroprozessor geschickt wird, kann das oder die richtigen Zugrelais ausgefahren werden (siehe Funktionsweise).

```
...

(let activate_ballcontact_new ((counter 0))
       (unless (> counter 26)
                     (begin (define ballcontact (make-instance-by-name
"fpMECollisionTracker"))
;;Die physikalische Welt bestimmen, in der sich der Tracker befindet
(fp-set-value ballcontact 'World phy-world)
;; Setzt die Models (not bodies), die getrackt werden sollen
(fp-set-value ballcontact 'Model1 (with-instance ball1(::get-model)))
(fp-set-value ballcontact 'Model2 (list-ref CollisionObjectList counter))
(fp-set-value ballcontact 'Active 1)
(fp-set-value ballcontact 'CanTrigger 1)
(fp-set-value ballcontact 'IntersectNewCB
                           (lambda (self) (BallCollisionNewCB (fp-get-value self
'ContactPos)(fp-get-value self 'ContactNormal))))
       (activate_ballcontact_new (+ counter 1)))))
...

;;Laden des Mikroporzessor-Knotens, da kein Avango-Kern-Objekt
(load-and-init-class "fpVRPNSensor")
;;Objekt erzeugen
(define atmel (make-instance-by-name "fpVRPNSensor"))
;;Aktiviert und Öffnet die Verbindung des VRPN-Sensors
(-> atmel 'enable "atmel0@127.0.0.1")
;;Funktion übermittlet Liste mit Werten an den Mikroprozessor
(define (set-atmel)
       (-> (-> atmel 'channels) 'set-value atmel_list))

...
```

5.3 Kommunikation mit dem Mikroprozessor aus Avango

Ist die Kollision zwischen Ball und Objekt beendet, wird die Funktion aus dem Feld "DisjointCB" aufgerufen, die dem Mikroprozessor mitteilt, dass die Zugrelais wieder eingefahren werden (im Prinzip werden sie nicht eingefahren, sondern die Stromzufuhr unterbrochen und von Federn zurück-gezogen). Da im Falle einer kurzen Kollision, die beiden Funktionen schnell hintereinander aufgerufen werden, käme es aufgrund der Verzögerung in den Bauteilen nicht zu einer Reaktion. Aus diesem Grund gibt es IMMER eine minimale Aktion auf das Ereignis "IntersectNewCB".
Um diese Informationen zu übermitteln, ist es notwending ein Objekt von "fpVRPNSensor" zu erstellen (siehe oben), was eine Art Interface zum Atmel darstellt. Damit die Klasse zur Verfügung steht, muss es extra geladen werden und die Datei "fpVRPNSensor.so" im gleichen Pfad sein, wie das Start-Script. Über ein Objekt von "fpVRPNSensor" erfolgt die Kommunikation mit dem "VRPN-Server", einer zusätzlichen Applikation, die die Daten dann an den Mikroprozessor weitersendet. Dieses Objekt wird als Knoten in den Szenengraph eingeordnet aber nicht visualisiert. Es hat wie andere Knoten auch diverse Felder:

1. 'enable
-> Aktiviert die VRPN-Verbindung zu einem Server
2. 'disable
-> Deaktiviert die Verbindung
3. 'set_channel
-> Setzt den Wert für einen bestimmten Kanal
4. 'get_channel
-> Liest den Wert eines bestimmten Kanals aus
5. 'add_callback
-> Definiert Callback-Funktion für einen bestimmten Kanal

Das Gerät wird dabei durch eine Anzahl an „Kanälen“ präsentiert (variiert von Gerät zu Gerät). Jeder Kanal stellt unserem Fall ein Register auf dem Mikrokontroller da und enthält einen Wert. Ein Kanal je nach Einstellung nur gelesen, nur geschrieben und gelesen und beschrieben werden. Die Bewegungssteuerung der Motoren befand sich noch in einem sehr experiementellen Stadium. Um die Bewegung des Balls innerhalb der Simulation zu ermitteln, wird in einer Schleife (genau gesagt mit einem "fpTriggerCB" der mit dem Timesensor verbunden ist und die ganze Zeit eine Funktion aufruft) die Position des Balls ermittelt und zusammen mit der vorherigen Position der Bewegungsvektor ermittelt. Dieser Bewegungssensor sollte mit dem USB-Vektor verglichen werden und die daraus resultierende Differenz in Motorbewegung umgewandelt werden.

```
(define (Loop x)
;; Daten schicken
;; Möglichkeit 1: Absolute Position schicken, Bewegung auf Microcontroler berechnen
;; Möglichkeit 2: Relative Position hier berechnen, Bewegungsvektor schicken
;; Berechung des Bewegungsvektors
(set! move-vec (make-vec2     (- (with-instance ball1(::get-x))(vec2-ref temp-vec 0))
                              (- (with-instance ball1(::get-y))(vec2-ref temp-vec 1))))
;; Alte Position speichern
(set! temp-vec (with-instance ball1(::get-pos))))

;; Funktion Loop an Trigger haengen
(define timeCB (make-instance-by-name "fpTriggerCB"))
(fp-set-value timeCB 'Callback Loop)

;; Trigger an das Feld von time-sensor haengen
(define (startLoopPos)
        ;; timeCB calls Loop
        (fp-connect-from timeCB 'Input time-sensor 'Time)
)

(define (stopLoopPos)
       (fp-disconnect-from timeCB 'Input time-sensor 'Time)
)

;;startet den loop
(startLoopPos)
```

5.4. Die Mikrocontrollerapplikation

Die Mikrocontrollerapplikation hat im Wesentlichen zwei Aufgaben zu erfüllen:

- Bereitstellung eines Kommunikationsprotokolls zum empfangen von Befehlen und zum senden/empfangen von Informationen.
- Ansteuerung der je zwei Motoren und Relais mit unterschiedlichen Spannungen.

Die wichtigsten zu übermittelnden Befehle/Informationen sind:

- Übermittlung von Trackball-USB-Daten (Richtung und Bewegung des Trackballs)
- Übermittlung der Bewegung des VR-Objekts
- einige Kalibrationsbefehle zur Justierung der Kräfte und Dynamiken
- Möglichkeit der direkten Steuerung von Motoren und Relais, um eine low-level-Nutzung zu ermöglichen

Einige weitere Befehle ermöglichen die Abfrage von Werten sowie einige nützliche Funktionen, wie z.B. die Simulation eines Stosses.

Das Kommunikationsprotokoll liegt im Anhang vor und sollte nach dem Lesen dieses Abschnitts keinerlei Fragen aufwerfen.

Die Aufgaben wurden realisiert, indem die Hauptprogrammschleife im Pollingbetrieb ständig die UART-Schnittstelle nach eingehenden Befehlen abfragt und beim Empfang eines Befehls eine befehlsinterpretierende Routine aufruft, welche wiederum die entsprechende befehlsverarbeitende Routine aufruft und/oder gegebenenfalls angeforderte Informationen zurücksendet. Einige Initialisierungsroutinen werden zu Beginn ausgeführt.
Im Folgenden wird eine Übersicht über die vorhandenen Quelldateien und Funktionen gegeben.
Anschliessend werden die Funktionen je nach Bedarf genauer erläutert. Sie können zum besseren Verständnis dieses Abschnittes den hoffentlich zufriedenstellend kommentierten Quellcode heranziehen.

Übersicht der Quellcodedateien:

- **ff_trackball_interface.h**
  Hier sind die möglichen Befehle und deren Parameter des Kommunikationsprotokolls in Form von Makros und Enumerations aufgeführt, so dass eine klare Trennung von Schnittstelle und Programm vorliegt. Das Interface wird in das Hauptprogramm eingebunden.
- **main.c – das Hauptprogramm**

Hinweis: Die folgenden beiden kleinen „Hilfsbibliotheken“ wurden von meinem Projektpartner Robert Lenhardt geschrieben und anschliessend von mir um Fehlerbehandlung und die Wahl zwischen blocking- und nonblocking-Modus erweitert.

- **uart.c/uart.h**
  Stellt eine einfache Schnittstelle zum senden und empfangen von Bytes über das UART-Interface bereit.
  Die Daten werden werden in einen Ringbuffer geschrieben und warten dann aufs Senden bzw. gelesen werden.
  Es werden Empfangs- und Sendefehler sowie volle Buffer erkannt und angezeigt.
  Die Daten können im nonblocking- bzw. blocking-Modus gesendet/empfangen werden, d.h. es der Funktionsaufruf kehrt wahlweise mit ggf. einem Fehlerstatus sofort zurück, oder wartet, bis eine korrekte Übertragung stattgefunden hat.
- packet.h/packet.c
  Stellt eine Schnittstelle zur Übertragung von Datenpaketen im von uns genutzten Übertragungsformat bereit. Es können der Befehlswert und zugehöriger Parameterwert gelesen oder gesetzt werden, aber auch fertige Pakete können übergeben/gelesen werden.
  Es werden die gleichen Features wie von uart.h/uart.c unterstützt, da Diese von packet.h/packet.c genutzt werden.

Hinweis: Die einzelnen Funktionen der Bibliotheken zur UART-Kommunikation werden nicht näher erläutert, da die Headerdateien und

Funktionsrümpfe sehr einfach sind und gut kommentiert wurden.
Die Bibliotheken müssen im Pollingbetrieb aufgerufen werden, sind also passiv und erzeugen keine Interrupts für eingehende Daten, Fehler oder fertiggestellte Sendeoperationen.
Prinzipiell werden verschiedene Flags gesetzt, welche das UART einschalten und konfigurieren (Anzahl Stopbits, Anzahl Bits pro Paket, Baudrate, Senden und/oder Empfangen) diverse UART-Interrupts ein- bzw. ausschalten (Datenempfang/-senden fertig).
Wichtig ist hier, dass der „senden fertig"-Interrupt sich selbst ausschaltet, nachdem er die Daten fertig übertragen hat, damit er nicht permanent aufgerufen wird, wenn gar keine Daten gesendet werden sollen, denn das behindert den normalen Programmfluss.

- siehe UCSRB &= ~ _BV (UDRIE);

Übersicht der Routinen des Hauptprogramms:

Hinweis: Alle globalen und lokalen Variablen sind kommentiert, sofern erforderlich, und werden hier nur im Bedarfsfalle erläutert.
Alle Makros und Enumerationen, für die Kommunikationsschnittstelle sind in *ff_trackball_interface.h* definiert.

Initialisierung :

init_timer() – Timer konfigurieren, erzeugt die beiden Interrupts zum ein- & ausschalten der PWM
init_IOs() – Ausgänge zum ein- & ausschalten der Motoren & Relais, sowie zum Richtungswechsel der Motoren einschalten
init_interrupts() – Interrupts global sowie die beiden Timer-Interrupts (Overflow = Register komplett hochgezählt; Compare Match = Register bis zum Vergleichswert hochgezählt) einschalten
init_globals() – Kalibrationsdaten aus dem Datenspeicher in die entsprechenden Variablen lesen
read_map() – Mapping USB-Vektor → PWM-Stufe vom Datenspeicher in die entsprechenden Variablen lesen
init() – Aufruf aller o.g. Initialisierungsroutinen

Interrupt-Routinen:

SIGNAL(SIG_OUTPUT_COMPARE0) – Timer Compare Match Interrupt für Motor- & Relaisansteuerung
SIGNAL(SIG_OVERFLOW0) – Timer Overflow Interrupt für Motor- & Relaisansteuerung
SIGNAL(SIG_OVERFLOW1) – Timer Overflow Interrupt des zweiten Timers zum Ermitteln der vergangenen Zeit für die wait()-Routine
SIGNAL(SIG_OUTPUT_COMPARE1A) – Timer Compare Match Interrupt zum Ermitteln der vergangenen Zeit für die wait()-Routine

Befehlsinterpretation:

int *limit_data(from, to, *data) – Überprüfen des Wertebereichs der mit Befehlen eingehenden Parameterwerte
interpret(command, data) – Interpretation eingehender Befehle und Aufruf der befehlsausführenden Routinen; ggf. setzen zu setzender Variablen und Senden von Rückgabewerten
int main() – Programmhauptschleife; fragt die Kommunikationsschnittstelle im Pollingbetrieb (Endlosschleife) nach eingehenden Befehlen ab und reicht Diese gegebenenfalls an die interpret()-Routine weiter

Befehlsausführung:

pull_relais(relais) – Anziehen eines oder beider Relais (low-level Befehl)
release_relais(relais) – Loslassen eines oder beider Relais
run_motor_1(power_as_percent) – Motor 1 mit ***percent*** Prozent Kraft einschalten
run_motor_2(power_as_percent) – Motor 2 mit ***percent*** Prozent Kraft einschalten
bump(direction) – Simulation eines Stosses aus Richtung *direction*
get_value(address, which) – Ermittlung des Wertes ***which*** zur Rückgabe über das Interface an den Computer

reset(which) – Rücksetzen einer oder mehrerer Kalibrationsvariablen
collision(x) – Simulation einer Kollision aus einer bestimmten Richtung mittels der Relais
collision_end() – Beenden einer Kollision
vector align_vector(x_ref, y_ref) – Anpassen der Bewegung des Trackballs an den übergebenen Vektor. Rückgabe des durch die Motoren erzeugten Vektors. (siehe Abschnitt „Designprinzipen“)
map_calibration() – Erstellen des Mappings USB-Vektor → PWM-Stufe; idealerweise einmalig vor Inbetriebnahme oder physischer Justierung des Trackballs
wait(ms) – ***ms*** Millisekunden warten

Erläuterungen zu Routinen des Hauptprogramms:

Enumerationen für Befehlsparameter

Die in ff_trackball_interface.h definierten Enumerationen dienen dazu, die mögliche Übergabe ungültiger Befehlsparameter zu verhindern.

Die Initialisierung mit Flagregistern

Microcontrollereinstellungen, wie z.B. das Ein- und Ausschalten von Interrupts oder das setzen der Baudrate oder der Anzahl von Stopbits für die UART-Schnittstelle und zahlreiche weitere Einstellungen werden durch das Setzen von Flags, d.h. Bits in Spezialregistern vorgenommen.
Damit man sich nicht die Adressen dieser Register und der korrekten Bitposition innerhalb der Register merken muss, werden für die gängigen Programmiersprachen (C, Assembler) normalerweise Makros in Headerdateien bereitgestellt. Für C beispielsweise sind die Makros für die Register und Bitpositionen in der Datei avr/io*.h (z.B. iom32.h) enthalten.

So wird beispielsweise aus
```
0x39 = 4; /* drittes Bit (4 = 2³) in Register 0x39 setzen */
```
durch
```
#define TIMSK = 0x39;
#define TOIE1 = 2;
```
einfach
```
/* Timer 0 Overflow Interrupt einschalten */
TIMSK = 1 << TOIE1;
```

Eine Erläuterung der hier gesetzten Flags ist hier überflüssig, da die Bedeutung jeweils im Quellcode kommentiert ist und im Programm AvrStudio nachgesehen werden kann.
Da sich stets acht Flags in einem Register befinden, muss darauf geachtet werden, dass beim Setzen eines Flags die verbleibenden Flags nicht versehentlich geändert werden.
Das geschieht am besten mit dem Bit-Oder-Operator |

Das oben genannte Beispiel würde beispielsweise alle übrigen Timer Interrupts, die nämlich mit dem selben Flagregister ein- und ausgeschaltet werden, ausgeschaltet!

```
TIMSK = 1 << TOIE1;
```

Hierdurch wird das niederwertigste Bit des Registers gesetzt. Anschliessend wird es um TOIE1=2 Positionen nach links verschoben.
War das Register vorher teilweise gesetzt, z.B. auf 1001 1001 (d.h. einige Interrupts eingeschaltet), so wird daraus zunächst 0000 0001, und anschliessend 0000 0100 !

Verwendet man den Oder-Operator, so bleiben die alten Werte unverändert:

```
TIMSK |= 1 << TOIE1; /* 1001 1001 --> 1001 1001 | 0000 0100 = 1001 1101 */
```

Es können auch mehrere Interrupts ein- und ausgeschaltet werden:

```
TIMSK = 1 << TOIE0 | 0 << OCIE0 | 1 << TOIE1;
```

Begrenzung der Eingangsparameter

Eingangsparameter, die nicht durch Enumerationen beschrieben werden können, müssen überprüft werden.
Das geschieht nötigenfalls durch die Routine ***limit_data(from, to, data)***. Der Wert ***data*** wird beim Herausfallen aus dem gegebenen Wertebereich [***from***, ***to***] auf den ihm am nächsten gelegenen Wert gesetzt, also entweder auf ***from*** oder auf ***to***.

Befehlsinterpretation durch Setzen von Variablen

Interrupts unterbrechen den gewöhnlichen Programmablauf, um die in der entsprechenden Interrupt-Routine notierten Befehle auszuführen und anschließend die Kontrolle an das normale Programm zurückzugeben.
Aus diesem Grunde genügt es häufig, im normalen Programm eine globale Variable zu setzen, die anschließend von einer Interrupt-Routine ausgelesen wird.
Hiervon macht die Befehlsinterpretationsroutine häufig Gebrauch.
Beispiel:
Das Kommando ***SetMaxMotor1Power*** wird aufgerufen.
Die ***interpret()***-Routine setzt darauf

```
_maximum_motor_1_power = data;
```

Anschliessend wissen die Timer-Interrupt Routinen, wie gross die maximale Motorkraft nun ist.

Kalibrierung

Es gibt verschiedene Werte, die experimentell eingestellt werden müssen, da zum Einen Hardwareungenauigkeiten im Prototypenstadium des Gerätes auftreten können, wie zum Beispiel die ideale Anzugskraft der Relais, zum Anderen einige Masse die experimentell ermittelt werden müssen, beispielsweise die maximale Motorkraft, die für den Standard-User verträglich ist.
Es existieren verschiedene Kalibrationskommandos zu diesem Zwecke.
Diese Kalibration durch Software sollte nach dem Prototypenstadium möglichst nur noch vorhanden sein, sofern sie von Nutzen für den User ist und von Diesem selbst in einfacher Weise vorgenommen werden kann.

Die Kalibrierung der USB→ PWM-Map

Um USB-Vektoren in eine geeignete PWM-Stufe für die Motoren umzusetzen, müssen wir zunächst einmal die Wertetabelle erstellen, welche anschließend im permanenten Datenspeicher des MCs abgelegt wird.
Die Aktualisierung der Tabelle muss nach jeder Justierung der Motoren erneut vorgenommen werden, idealerweise ist das einmalig nach der Produktion des Gerätes.

Das wird durch die Funktion ***map_calibration()*** wie folgt realisiert:
Es werden separate Tabellen für die beiden Motoren geschrieben, indem jeder Motor einige Zeit (drei Sekunden) auf einer bestimmten PWM-Stufe laufen gelassen wird, so dass eine stabile Geschwindigkeit angenommen werden kann. Der hierdurch für die entsprechende Achse erzeugte USB-Vektor wird in die Tabelle an die entsprechende Position geschrieben.

Hier existiert nun noch ein wieder einmal durch mögliche Hardwareungenauigkeiten hervorgerufenes Problem:
Was ist, wenn die Motoren auf die jeweils andere Achse übersprechen?
Beispielsweise dreht der Motor der Längsachse den Trackball nicht exakt entlang der Längsachse, sondern um einige Grad verdreht. In diesem Falle wird ein Teil der Bewegung entlang der Querachse sein. Für diesen Fall wird der Einfluss auf die jeweils andere Achse notiert und muss bei Nutzung der Tabelle berücksichtigt werden.
Er berechnet sich aus:
Vektor fremde Achse/Vektor eigene Achse.

Die Funktion ***alignVektor()*** nimmt die in "Designprinzipien" erläuterte Vektorangleichung wie folgt vor:

1. Der Anpassungsvektor wird berechnet:

```
// Abziehen des bisher durch die Motoren
erzeugten Vektoranteils durch auslesen der
Tabelle
temp_v.x = usb_v.x -
eeprom_read_byte((uint8_t *) EE_BEGIN +
                                _motor1_power);
temp_v.y = usb_v.y -
eeprom_read_byte((uint8_t *) EE_BEGIN + 255 +
                                _motor2_power);

// Berechnen des Anpassungsvektors:
Berechnung der Differenz zwischen gefordertem
und durch
// Handkraft hervorgerufenen Vektors
_return_v.x = x_ref - temp_v.x;
_return_v.y = y_ref - temp_v.y;
```

2. Ermitteln der x- und y-PWM-Stufe und ihrer Einflüsse auf die jeweils andere Achse:

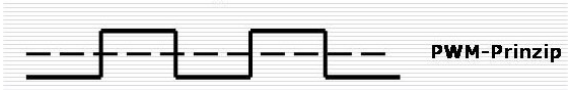
```
do { // gleiches geschieht für y
      address = (uint8_t *) (EE_MAP1_BEGIN +
      _minimum_motor_1_power + i++);
      if (...in range...) {
        usbx_expect = eeprom_read_byte(
        (uint8_t *) address);
      }
} while (usbx_expect < _return_v.x);

// Einfluss aktuellen Wertes auf x-Achse
influence_x =(((float) usbx_expect *
(float)eeprom_read_word((uint16_t *)
EE_Y_FACTOR_MOTOR_1);
```

3. Anpassung der gefundenen Werte, bis der letzte ermittelte Wert dem neuen ermittelten Wert entspricht:

```
do { /* das gleiche wie unter Punkt 2,
diesmal aber mit Anpassung */
      do { // gleiches geschieht für y
        if (...in range...) {
        usbx_expect = eeprom_read_byte(
                (uint8_t *) address +
                (uint8_t) influence_y);
        }
      } while (usbx_expect < _return_v.x);

      // der Einfluss ist nun:
      influence_x +=(((float) usbx_expect -
        (float) usbx_last_expect)) *
        (float) eeprom_read_word(
        (uint16_t *) EE_Y_FACTOR_MOTOR_1);
      ...
} while (
(usbx_expect != usbx_last_expect) ||
(usby_expect != usby_last_expect));
```

Erzeugung der Pulsweitenmodulation

Die Pulsweitenmodulation wird mit Hilfe eines Timers erzeugt.
Der Timer inkrementiert pro Taktzyklus um eins. Ist sein Zählregister voll, d.h. bis 256 hochgezählt (8-Bit), so erzeugt dieser einen sogenannten „Overflow Interrupt“ und beginnt (in unserem Falle, andere Konfigurationen existieren) von vorne zu zählen.
Ein weiterer Interrupt, der sogenannte „Compare Match Interrupt“, wird jedesmal aufgerufen, wenn der Inhalt des Timerregisters mit dem eines Vergleichsregisters übereinstimmt, welches beliebig gesetzt werden kann.
Schaltet man einen Motor mit dem Overflow Interrupt ein und mit dem Compare Match Interrupt aus, so kann man mit dem Setzen des Vergleichsregisters das Verhältnis von eingeschalteten zu ausgeschalteten Zuständen und damit die Kraft des Motors steuern.



Die maximale Stromversorgung des Motors ist die Ausgangsspannung des Pins, an dem der Motor anliegt. Allerdings kann man, wie im Falle der von uns verwendeten Motortreiber, den Motor mit einer externen Stromquelle versorgen und den Ausgang des Microcontrollers als Steuerspannung verwenden.

Ein Beispiel:

Enthält das Vergleichsregister den Wert 128, so wird nach dem halben Hochzählvorgang des Timers der Motor wieder ausgeschaltet und erst wieder eingeschaltet, wenn der Timer von neuem beginnt (Overflow Interrupt).
Der Motor liefe also bei fünfzig Prozent.

Unsere PWM geht ein klein wenig anders von statten:
Da wir vier Aktoren (2 Motoren und 2 Relais) mit unabhängigen Kräften gleichzeitig ansteuern müssen, setzen wir das Vergleichsregister auf 255, d.h. auf den maximalen Wert. Somit liefen alle Aktoren bei 100 Prozent.
Die Aktoren werden bei jedem Compare Match ausgeschaltet, so daß ein Aktor nach dem Einschalten ohne erneutes Einschalten maximal einen Zyklus läuft:

```
SIGNAL(SIG_OUTPUT_COMPARE0)
{
      // turn off everything
      PORTC = 0;
}
```

Der Overflow Interrupt zählt eine statische Variable ständig hoch, genau wie ein Timer, und schaltet nun die einzelnen Aktoren nur ein, wenn die ihnen zugeordneten Power-Werte kleiner als dieser Zähler sind.
Ist die Power von Motor 1 beispielsweise auf 128 gestellt (möglich sind Werte von 0 bis 255), so wird der Motor eingeschaltet, bis der Zähler 128 erreicht, danach nicht mehr.

```
SIGNAL(SIG_OVERFLOW0) {
      _t &= 255;
      _t++;
      if (_t <= _motor1_power) {
        PORTC |= 8;
      ...
      }
}
```

Die Auflösung ist aufgrund dieser Methode natürlich um das 255-fache gesunken, ist aber dennoch akzeptabel.

INTERRUPT vs SIGNAL

Es stellt sich die Frage, ob Interrupts von Interrupts unterbrochen werden dürfen.

Sollte ein Timer-Interrupt beispielsweise sich selbst unterbrechen, so wird das System früher oder später abstürzen, da der Stack-Pointer für Routinen irgendwann überläuft. Wenigstens wird eine Verzögerung der Programmabarbeitung eintreten.
Werden während der Abarbeitung eines Interrupts andere Interrupts ignoriert, so gehen möglicherweise wichtige Informationen verloren, beispielsweise das Eintreffen eines Datenpaketes über eine Schnittstelle.

Diese Tatsachen sollten beim Einsatz von Interrupts stets bedacht werden.

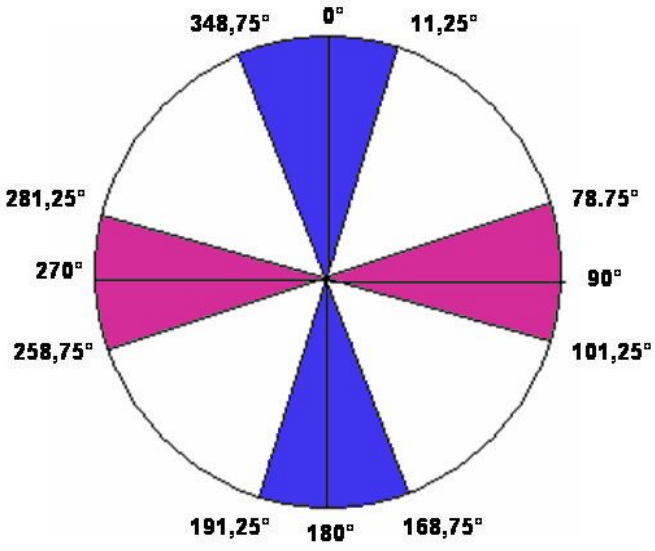
Die mittels ***INTERRUPT()*** registrierten Routinen können von anderen Interrupts unterbrochen werden, mit ***SIGNAL()*** registrierte nicht.
Wir haben uns entschieden, daß Interrupts nicht unterbrochen werden können, da so eine korrekte Pulsweitenmodulation garantiert werden kann. Die Timer-Interrupts müssen sehr wenige Aufgaben erfüllen und werden sich daher nur in Einzelfällen überschneiden.
Das UART wird im Polling-Betrieb abgefragt und erzeugt somit keine Interrupts.

Hinweis: Werden die Interrupt-Routinen falsch bezeichnet, so wird dies nicht erkannt !
Das bedeutet, daß die entsprechenden Routinen in einem solchen Falle niemals ausgeführt werden.

Die Kollisionssimulation

Eine Kollision wird durch zwei Funktionen repräsentiert: Eine Funktion stellt den Kollisionszustand her, die Andere (***end_collision()***) beendet diesen wieder.
Im Zustand der Kollision kann sich das Objekt nicht in Richtung der Kollision fortbewegen, allerdings in die entgegengesetzte Richtung. Der Trackball detektiert sehr feine Bewegungen, so dass diese Bewegung wahrgenommen werden kann und ein Beenden des Sperrzustands möglich ist.
Der Funktion wird der normierte x-Vektor des normierten Einheitsvektors der Kollisionsrichtung übergeben.

Im Falle eines Vektors x=2, y=1 wird
x= x/Wurzel(x^2+y^2) = 1/2.234 = 0.447



Sperren des Relais an der Querachse Sperren des Relais an der Längsachse

So wird ein zu übergebender Parameter gespart. Die Kollisionsbereiche sind wie folgt eingeteilt:

Deshalb ist es nicht notwendig, zu wissen, ob der y-Vektor ein positives Vorzeichen hat oder nicht.

Die bump()-Routine

Diese Routine soll das Rollen des Trackballs über eine Schwelle, z.B. einen Fensterrahmen, simulieren. Es lässt sich eine Richtung angeben. Das geschieht, indem die Motoren für 0,1 Sekunden eine Bewegung in die angegebene Richtung mit voller Kraft realisieren. Anschliessend wird der alte Status wieder hergestellt.

Warten

Eine scheinbar so triviale Aufgabe wie das Nichtstun muss sogar in C erst mit ein wenig Einfaltsreichtum implementiert werden, insbesondere, wenn eine ganz bestimmte, eventuell längere Wartezeit gefragt ist.

Uns stehen folgende Informationen zur Verfügung:

- Einziges Zeitmaß ist die Taktfrequenz des Oszillators. Uns ist bekannt, daß wir mit acht MHz Taktfrequenz arbeiten.
- Benutzen wir nun einen Timer, so inkrementiert dieser um einen Schritt pro Takt.
- Bei dem hier eingesetzten 16-Bit Timer taucht alle 65536 Schritte ein Interrupt auf.
- Wir können ein Flag setzen, so daß der Timer nur alle 1024 Takte inkrementiert.
- Somit erhalten wir einen Interrupt alle 1/(8 MHz/1024) = circa 0.000128 Sekunden.

Die wait-Routine() erfüllt ihre Aufgabe nun wie folgt:

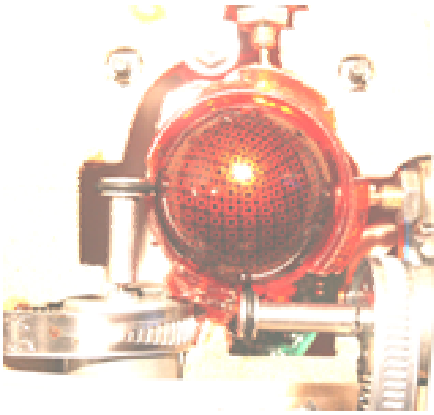
Bei der Initialisierung wird der Timer 1 eingeschaltet und inkrementiert eine globale Variable ***_waiting_timer_cycles*** jedesmal, wenn ein Interrupt auftritt.
Wir wissen, daß das alle 0.000128 Sekunden passiert.

wait() setzt ***_waiting_timer_cycles*** auf 0;

wait() ermittelt den aktuellen Wert des Timers 1.

wait() wartet, bis ***_waiting_timer_cycles*** so weit inkrementiert wurde, daß dieser Wert und der aktuelle Timerwert den Ablauf der gewünschten Zeitperiode angeben.

Die maximale Wartezeit ist durch den maximalen Wert von ***_waiting_timer_cycles*** begrenzt. Das sind in diesem Falle 65536*0.000128 = circa 8.3 Sekunden.



5.5. Die Kommunikation über VRPN

Was ist VRPN ?

VRPN bedeutet "Virtual Reality Peripheral Network". Hierbei handelt es sich um eine open-source Applikation, die VRs beliebige Eingabegeräte auf Basis einer Client-Server Architektur rechnertransparent zur Verfügung stellt.
Dabei wird auf jedem Rechner, der ein oder mehrere Eingabegeräte zur Verfügung stellt, ein Server bereitgestellt, mit dem sich beliebige Clients, d.h. VRs verbinden können, um mit dem jeweiligen Eingabegerät zu kommunizieren.
Je nach Eingabegerät (Maus, Tastatur, Joystick, ...) sind verschiedene Kommunikationsschnittstellen vorhanden.
Das wird durch eine C++-Implementation realisiert, welche die verschiedenen Gerätetypen von einem "Basisgerät" ableitet.
Die clientseitige Implementation basiert auf einer Menge von Klassen, die in die VR-Applikation eingebaut werden müssen und so eine Kommunikationsschnittstelle bereitstellen.

Der Lehrstuhl der virtuellen Realität hat sich dafür entschieden, VRPN zu nutzen, um eine einfache Kommunikation mit dem Microcontroller und anderen (teilweise an unserer Fakultät entwickelten) Eingabegeräten zu nutzen.
Für den Microcontroller mussten eigens Klassen abgeleitet werden, welche das von der Firma Albotronic für uns entworfene Protokoll zur Kommunikation mit dem Atmel ATMega32 (oder beliebigen anderen Microcontrollern) über die serielle Schnittstelle 1 per UART verstehen.
Diese Arbeit nahm uns unser Komilitone Christian Lessig ab.
Allerdings erfolgten unsererseits einige Anpassungen, weshalb nun eine Übersicht über die für uns relevanten Klassen und Funktionen erfolgt, damit ein Einstieg in die eventuelle Weiterentwicklung gegeben ist.

Der Quellcode ist im Quellcodepaket im Verzeichnis */**vrpn**…* enthalten.

Alle folgenden Verzeichnis- und Dateiangaben gehen von hier aus.

Die Atmel-Kommunikation über VRPN

Für die Kommunikation mit dem Atmel wurden dem Verzeichnis ***/vrpn*** einige Dateien hinzugefügt.
Die Klasse ***vrpn_Atmel*** ist die abgeleitete Klasse, welche das Interface für den Mikrocontroller mittels ihrer Funktion ***mainloop()*** bereitstellt, wie das alle anderen Klassen, die eine solche Schnittstelle für ein bestimmtes Eingabegerät bereitstellen, auch tun müssen.
Desweiteren bilden die Dateien mit Namen ***vrpn_atmellib**** eine kleine Bibliothek, welche verschiedene von ***vrpn_Atmel*** genutzte Funktionen bereitstellt. Die Bibliothek wird in der Headerdatei ***vrpn_atmellib.h*** zusammengefasst.

Im Folgenden wird die Funktionsweise der ***mainloop()***-Routine besprochen.
Anschliessend wird eine Übersicht über die darunterliegende Bibliothek und deren Funktionsweise gegeben.

Die mainloop()-Routine

Diese Routine wird im Polling-Betrieb von der Hauptroutine des Servers aufgerufen und erhält die vom Client gesendeten Daten (in unserem Falle also Befehle an den MC) über shared memory.
Die Aufgabe ***der mainloop()***-Routine ist es also, den shared memory auf den Neueingang von Daten zu überprüfen und ggf. Befehle zu senden. Beide Implementationen greifen auf den shared memory zu, welcher aus einem Array besteht. Der Index des Arrays repräsentiert den Befehl, der Inhalt den Parameter.
Ist ***o_channel[0]*** beispielsweise mit dem Wert 5 belegt, so wird der Befehl ***AlignXByte*** mit dem Paramterwert 5 auf dem MC des Trackballs ausgeführt.

Der wichtigste Unterschied zwischen der Implementation von Christian Lessig und der Anpassung von uns betrifft die Kommunikation mit dem MC:
Die ursprüngliche Implementation sendet jeden

neuen Befehl an den MC und fordert eine Befehlsbestätigung durch Rückgabe eines Paketes des gleichen Befehls an. Alle Befehle, welche nicht gesendet werden sollen, werden dennoch zur Rückgabe eines Wertes aufgefordert. Es werden also pro Pollingschleife doppelt so viele Befehlspakete (Befehl + Antwort bzw. Wertaktualisierung) übertragen, wie Befehle vorhanden sind.

Das ist für unseren Fall nicht notwendig gewesen, weshalb wir lediglich wirklich von der Client-VR empfangene Befehle senden und auch keine Antwort erwarten, es sei denn, es wird explizit ein Rückgabewert verlangt, was durch den Befehl festgelegt wird.

Hier nun ein Auszug aus der aktualisierten mainloop()-Routine:

```
// Überprüfung aller Befehle (hier: channels)
for(int i=0; i<o_num_channel; ++i) {
      // Ist ein neuer Befehl eingegangen ?
      if (o_channel[i] != channel[i]) {
        ...
      // Ja --> Senden...
      /* ...eines Befehls, der keine Antwort
      gibt... */
      if (_channel_mode[i] ==
              VRPN_ATMELLIB_MODE_WO) {
        setRegister(serial_fd, i, (unsigned
                      char) o_channel[i]);
        ...
      }
      /* ...eines Befehls, der eine Antwort
        gibt... */
      else {
        temp = getRegister(serial_fd, i,
               (unsigned char) o_channel[i]);
               ...
        }
        o_channel[i] = temp;
      }
      // melde Befehl als gesendet
      channel[i] = o_channel[i];
  }
}
```

Zu beachten ist, daß ein Befehl nicht zweimal hintereinander mit demselben Parameterwert ausgeführt werden kann, da zu sendende Befehle ermittelt werden, indem geprüft wird, ob der zugehörige Parameter geändert wurde:

```
      if (o_channel[i] != channel[i])
```

Das ist natürlich nicht sehr komfortabel und sollte anders implementiert werden, d.h. beispielsweise durch einen zusätzlichen Array mit Flags.

Die wichtigsten Routinen der VRPN-Microcontrollerbibliothek sind:

- ***setRegister()*** und ***getRegister()***
- ***setCmd()*** und ***getCmd()***

Die ****Cmd()*-Funktionen werden von den ****Register()*-Funktionen benutzt.
Die ***set*()*-Funktionen packen ein Datenpaket zusammen und schicken es ab.
Die ***get*()*-Funktionen erwarten zusätzlich eine Antwort, die in einem akzeptablen Zeitrahmen eintreffen muss. Eine Antwort ist ein Paket mit gleichem Befehlswert. Der Parameterwert kann verschieden sein.

Aufgaben der einzelnen Bibliotheksdateien:

errno - Fehlermakrodefinitionen
helper - Hilfsfunktionen zum packen/extrahieren eines Datenpaketes
tester - teste Variablen auf Gültigkeit(sbereiche), i.e. Handles
openclose - uart initialisieren
iobasic - ***getCmd()*** und ***SetCmd()***
register - ***setRegister()*** und ***getRegister()***

Fehlererkennung durch VRPN

Es wird erkannt und angezeigt, ob beim Start des Servers und der Verbindung mit dem MC irgendetwas nicht geklappt hat. Auch nicht erfolgreich gesendete Befehle werden angezeigt. Da jedoch keine Empfangsquittung erfolgt, kann die korrekte Verarbeitung nicht garantiert werden. Ein nicht gesendeter Befehl wird ausserdem durch die folgende Pollingschleife erneut gesendet.
Sollte der MC jedoch beispielsweise während des Betriebs vom Computer gelöst werden, so wird das nicht bemerkt.
Der Avango-VR-Client meldet, falls der Server nicht mehr reagiert und versucht selbsttätig, die Verbindung wieder herzustellen.

Achtung: Wenn die VR zwei Befehle innerhalb eines Pollingzyklusses von VRPN abgibt, so werden die Befehle hier entsprechend ihrer Reihenfolge im Array ausgeführt, nicht in der des Setzens ! Das ist zwar oft nicht relevant und

wird vermutlich auch nur selten eintreten, ist aber zu berücksichtigen.

Die Kompilierung von VRPN

Um den VRPN-Quellcode nach Änderungen neu zu kompilieren, müssen Sie folgende Schritte durchführen:
Fügen Sie den Variablen ***SLIB_FILES*** und ***SLIB_INCLUDES*** in der Datei ***Makefile*** im Verzeichnis ***/vrpn*** ggf. von Ihnen neu erstellte Quellcodedateien hinzu.
Führen Sie die make-Datei im Verzeichnis ***/vrpn*** aus. (Einfach "make" in die Betriebssystemshell eingeben.)
Führen Sie die make-Datei im Verzeichnis ***/vrpn/server_src*** aus.
Anschliessend werden Sie die ausführbare Datei ***vrpn_server*** im Verzeichnis ***/vrpn/server_src/pc_linux*** finden.

5.6. Der Universal Serial Bus (USB)

Da wir das optische Tracking des Trackballs nicht direkt abgreifen konnten und auch das USB-Protokoll mit dem MC auszulesen kaum möglich ist, haben wir uns entschieden, die Daten am USB-Port des Computers abzugreifen und über VRPN dem Microcontroller ständig mitzuteilen, indem wir entsprechende Aufrufe in die im vorhergehenden Abschnitt besprochene ***mainloop()***-Funktion einbauen:

```
// auslesen der Mausbewegungen
hid_event = _hid_reader.get_events();

if (hid_event != NULL) {
      // Bewegung entlang der x-Achse
      if (hid_event->first != 0) {
        setRegister(serial_fd, SetXValueHByte
                        , (char) (0x0000FF00 &
                        hid_event->first>>8);
        setRegister(serial_fd, SetXValueLByte
                        ,  (char) (0x000000FF
                        & hid_event->first);
  }
      ... ebenso für y-Achse
}
```

Die hier aufgerufenen Funktionen befinden sich in der Klasse ***/vrpn/hid_reader***. Die Klassen sind ausserdem inclusive eines Testprogramms im Verzeichnis ***/hid_reader*** des sourcecode-Paketes enthalten.
Eine Instanz dieser Klasse sucht bei der Initialisierung eigenständig den Trackball, indem sie alle verfügbaren USB-Ports abhört und die Gerätekennummer überprüft. Wird der von uns verwendete Microsoft Trackball verwendet, wird dieser erkannt.
Anschliessend werden pro Aufruf von ***get_events()*** die aktuellsten MAUSBEWEGUNGEN zurückgegeben, d.h. andere Events (z.B. Maustasten~) werden nicht zurückgegeben.
Es werden pro Sekunde weniger als insgesamt 250 Events erzeugt, so daß VRPN keine Schwierigkeiten haben sollte,
Diese zu verarbeiten.

Zu erwähnen ist noch, daß das UART im nonblocking-Modus ausgeführt werden muss (und wird), damit ***mainloop()*** unverzüglich fortfahren kann, falls kein Event aufgetreten ist.

Unix-Derivate stellen zum Auslesen von USB-Ports Strukturen in Klassen bereit, beispielsweise:

```
struct input_devinfo {
  uint16_t bustype;
  uint16_t vendor;
  uint16_t product;
  uint16_t version;
};
```

Diese Strukturen können instantiiert und anschliessend ausgelesen werden. So sind alle über den USB-Bus erhältlichen Informationen sehr einfach lesbar und sendbar. Einige kleine Beispielprogramme stellt der wissenschaftliche Mitarbeiter Jan Springer unserer Fakultät bereit (Verzeichnis ***/source/ev_family*** bzw. ***/bin/ev_family***). Sie zeigen, wie einfach mit dem USB-Bus gearbeitet werden kann (so einfach, daß hier keine weiteren Erklärungen notwendig sind).

5.7. Stand der Dinge und toDo's

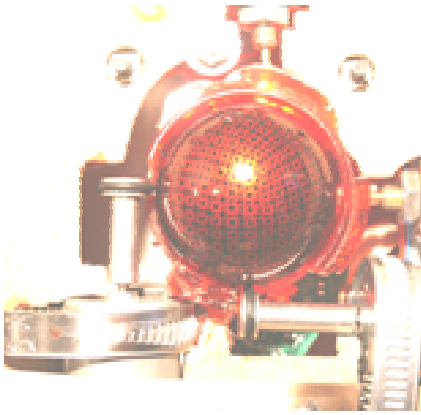
- Wir hatten unerklärliche Probleme mit der Kommunikation über UART, d.h. die Bytes wurden bei der Übertragung permanent verfälscht. Wir überprüften und tauschten alle Hardware und verstellten alle möglichen Parameter der UART-Schnittstelle (Baudrate, Stopbits uvm.). Leider gelang uns die Kommunikation nach zweiwöchiger Beschäftigung mit diesem „Problemkind" dennoch nicht. Da das UART esssentieller Bestandteil unseres Gerätes ist, konnten wir alle davon abhängigen Aufgaben, wie das Testen der Mikrocontrollerapplikation und der Hardwarefunktionalitäten nur eingeschränkt durchführen.
  Eine Wiederaufnahme des Projektes im neuen Semester war leider nicht möglich, da zum einen unser Mikrocontroller anderweitig verwendet worden ist und zum anderen prototypenbedingte Fehleranfälligkeiten des Evaluationsboards von uns nicht behoben werden konnten (diverse Wackelkontakte).
- Informationen über das UART können controllerseitig umfassend im Datenblatt nachgesehen werden, computerseitig existieren verschiedene man-pages unter Unix zu diesem Thema. Zum Einstieg genügen die ***termios***-Pages.
- Aus diesem Grunde ist der Quellcode des MCs nicht hundertprozentig getestet.
- Die Kalibrationsbefehle können nach der Herstellung einer einwandfreien Hardware beseitigt werden.
- Der VRPN-Server und die Avango-Applikation laufen zuverlässig und bedürfen zunächst keinerlei Verbesserung.
- Das USB-PWM-Mapping des MCs wird zur Zeit permanent aus dem EEPROM gelesen. Sollte das zu langsam sein, so ist über ein initiales Einlesen in den Speicher nachzudenken.
- Sollte eine Performanceverbesserung angestrebt werden, so sollte das Paper ***efficient C Coding for AVR*** gelesen werden. Da ein MC in allen belangen kleinere Dimensionen besitzt, ist hier eine Optimierung grundsätzlich bedeutsamer. Hier ist vor allem die Entscheidung zwischen Variablenarten entscheidend (lokal, global, statisch usw.)
- Ein MC-seitiger Pufferüberlauf des UART wird dem PC zur Zeit nicht mitgeteilt. Das sollte geschehen, damit der PC entsprechend neu senden kann.
- Eine von VRPN unabhängige Implementierung des Treibers als einfache Bibliothek oder daemon-Prozess wäre denkbar.
  Eine Bibliothek ähnlich der in VRPN verwendeten Klassen liegt im Verzeichnis ***/source/atmellib*** vor und wurde von Christian Lessig (ein Komilitone) implementiert.
- Bisher existieren nur Linux-kompatible Treiber. Eine Realisierung unter Windows-Systemen steht noch aus.

6. Installation

Um das komplette System zu betreiben, müssen Sie die folgenden Schritte in der angegebenen Reihenfolge durchführen:

- Schliessen Sie die Hardware an, wie im Abschnitt "Die Hardware - Aufbau ..." beschrieben.
- Konfigurieren Sie nach einer Änderung des Schnittstellenprotokolls des MCs stets die Datei ***vrpn.cfg*** ganz am Ende.
  Für jeden Befehl wird angegeben, ob es sich um einen Schreib- und/oder Lesebefehl handelt. Sie finden die Datei vorkonfiguriert im Verzeichnis ***/vrpn*** des Programm- und Quellcodepaketes
- Starten Sie den VRPN-Server, indem Sie die Datei ***vrpn_server*** aus dem */vrpn*-Verzeichnis des Quellcodepaketes ausführen.
  Die Datei vrpn.cfg muss sich im gleichen Ordner befinden.
- Starten Sie die Avango-VR-Beispielapplikation, indem Sie die Datei ***start-aview-mon*** aus dem Verzeichnis ***/avango_example*** des Quellcodepaketes starten. Die Datei ***libfpVRPNSensor.so*** muss sich im selben Verzeichnis befinden und Sie finden sie auch dort.

Hinweis: Das System läuft nur unter Linux/Unix mit einer Avango-Installation. Sollten Sie keine Avango-Installation besitzen, so können Sie diese hier bekommen. Avango ist ein VR-Framework und wurde vom Frauenhofer Institut für Medienkommunikation entwickelt. Die Erstellung von VRs mittels Avango wird hier nicht vermittelt und Bedarf einiger Einarbeitungszeit. Allerdings dient diese VR nur als Beispiel. Sollten Sie den MC mit einer anderen VR nutzen können, so müssen Sie diese zu einem VRPN-Client erweitern. Wie das geht, erfahren Sie auf http://www.cs.unc.edu/Research/vrpn/



7. Downloads

7.1. Quellcodes

Download (zip)

Inhalt der zip-Datei:

- **atmel** – Die MC-Applikation
- **atmel_basic_protocol** – Implementiert das Standard-Protokoll auf dem MC (von Marcus Borst, Albotronic)
- **atmel_use_examples** – einfache, kommentierte Programmbeispiele in C und Assembler
- **atmellib_c** – von Christian Lessig; stellt einfachen Zugriff vom Computer auf die Befehlsschnittstelle des Atmel Standardprotokolls zur Verfügung.
- **avr_includes** – die Schnittstelle der avr-libc Hilfsbibliothek zum anschauen.
- **ev_family** – Einige Programme zum Auslesen von USB-Ports. Nützlich und einfach.
- **ff_trackball_test_...** – kann benutzt warden, um die Relais und Motoren mit den Tastern des Evaluationsboards zu steuern
- **hid_reader** – Liest die Bewegungen (USB-Daten) des Trackballs aus und gibt sie zurück. Inclusive Beispielprogramm.
- **uart_library_for_atmel** – von Robert Lenhardt, erweitert von Onno Haak; Bereitstellung einer einfachen Schnittstelle zur Nutzung des UART auf dem ATMega32
- **vrpn_(ff_optimized_version)** – von Christian Lessig erweiterte und von Onno Haak angepasste Version des VRPN

7.2. Binärdateien/Programme

Download (zip)

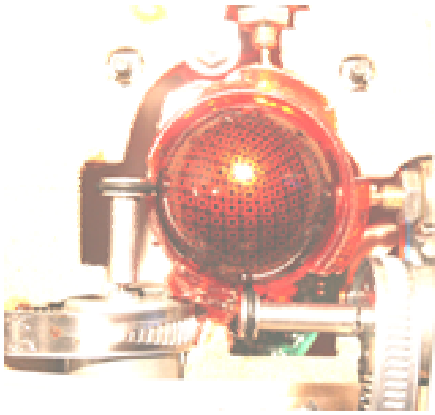
Die Binärdateien sind ausschliesslich aus den Quellcodes aus 7.1 entstanden, weshalb hier nur nötigenfalls auf einzelne Ordner/Dateien eingegangen wird.

Inhalt der zip-Datei:

- *atmel_use_examples*
  - *asm_examples*
- *avango_example*
  - *start-aview-mon* – ausführbare Datei
  - *libfpVRPNSensor.so* – muss sich hier befinden (shared obj)
  - weitere
- *ev_family*
- *trackball*
  - *daimler.bin* – MC-Applikation von Albotronic
  - *ff_trackball.bin* – MC-Applikation von uns
- *vrpn*
  - *vrpn_server* – ausführbare Datei
  - *vrpn.cfg* – Konfigurationsdatei (muss im gleichen Verzeichnis wie *vrpn_server* sein)
  - *libfpVRPNSensor.so* – s.o.

7.3. Präsentationen

- Zwischenpräsentation der Projektgruppe „haptical Interfaces“
- Zwischenpräsentation des FF-Trackballs
- Präsentation des FF-Trackballs
- Ein Vortrag zum Anschluss von Peripherie an Mikrocontroller



8. Quellen/Links

8.1. Informationen zum Atmel ATMega32

- Datenblatt (PDF)
- Produktbeschreibung (PDF)
- Befehlssatz (PDF)

8.2. Software zur Mikrocontrollerprogrammierung

- AvrStudio – Diese Entwicklungsumgebung wird von der Firma Atmel kostenlos bereitgestellt und bietet die Emulation/Simulation, aber auch den echten Betrieb und Debugging aller Atmel-MCs in Assembler. Hier sind ebenfalls viele weitere Tools von Atmel zur MC-Entwicklung verfügbar.

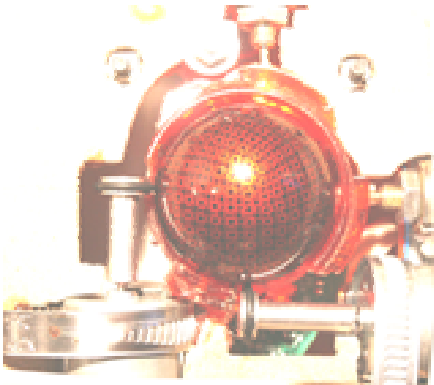
- WinAVR – Ein Set von Entwicklungstools für die Programmierung von AVR-MCs unter Windows MS-DOS in C oder Assembler. Ein Kompiler und die avr-libc Hilfsbibliothek sind die Kernfeatures.
- avr-gcc – Ein Compiler zur MC-Programmierung unter Unix/Linux. Features entsprechen im wesentlichen denen von WinAVR. NOCHMACHEN gucken wasses is.

- Ponyprog – zum Downloaden des kompilierten MC-Programms auf den MC. Versionen für Windows und Unix-Derivate.

8.3. Sonstige

- www.atmel.com – Atmel bietet umfassende Informationen und Hilfestellungen zu all seinen Mikrocontrollern und anderen Bauelementen an.
- http://www.atmel.com/dyn/products/tools.asp?family_id=607 - Der ATMega32
- http://www.atmel.com/dyn/products/tools.asp?family_id=607 - Hier finden Sie viele Programmierbeispiele und Erläuterungen zu verschiedenen Problemstellungen als PDF incl. Quellcode vor.
- http://www.mikrocontroller.net – Auf dieser Seite finden Sie ein gutes Tutorial zur Programmierung von Mikrocontrollern sowie vieles weitere zum Thema MC-Programmierung (deutsche Seite).
- http://www.avrfreaks.net/ - sehr kompetente Entwicklerseite
- http://www.microcontroller.com - Entwicklerseite für AVR-Leute
- http://r.webring.com/hub?ring=avr – Ein AVR-Webring
- Efficient C Coding for AVR (PDF von Atmel)
- http://www.frogmouth.net/hid-doco/linux-hid.html - ein gutes USB-Tutorial
- www.elektronik-kompendium.de - Diese Seite behandelt elektronische Bauteile und Schaltungen aller Art von einfach bis komplex.
- Elektronikbauteile uvm.:
    - www.farnell.de
    - www.conrad.de
    - www.huebner-elektronik.de
- Internetseite des Lehrstuhls für Virtuelle Realitäten an der Bauhaus-Universität http://www.uni-weimar.de/medien/vr

- Vortex
    - http://www.cm-labs.com



9. Anhang/technische Informationen

- Atmel-Schnittstellen
  - Allgemeines Protokoll (Marcus Borst, Albotronic)
  - FFTB-Protokoll v1 (verworfen)
  - FFTB-Protokoll v2
- USB-Protokoll des Trackballs
- VRPN
  - VRPN-Avango-Interface (Christian Lessig)
  - Anwendungsvorlage
- Produktbeschreibungen
  - Motoren
  - Relais
  - Federn
- Datenblätter
  - Motoren
  - Relais
  - Federn
- UART-Schaltplan
- ISP-Schaltplan
- Fertigungsskizze der Motorrädchen