UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Realisierung einer Stewart-Plattform

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science
im Studiengang Informatik

vorgelegt von

Daniel Mc Stay

Erstgutachter:	Prof. Dr. Dieter Zöbel Institut für Softwaretechnik
Zweitgutachter:	Dr. Merten Joost Institut für integrierte Naturwissenschaften, Abteilung Physik
Drittgutachter:	Alexander Hug Institut für Wirtschafts- und Verwaltungsinformatik

Koblenz, im Oktober 2012

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Parallelmanipulatoren, welche den Stewartmechanismus nutzen, ermöglichen die präzise Ausführung von Aufgaben in einem begrenzten Arbeitsraum. Durch die Nutzung von sechs Freiheitsgraden wird eine hohe Flexibilität der Positionierung erreicht. Die robuste Konstruktion sorgt zudem für ein sehr gutes Verhältnis von Gewicht zu Nutzlast. Diese Bachelorarbeit befasst sich mit der Entwicklung einer flexiblen Softwarelösung zur Ansteuerung einer Stewartplattform. Dies umfasst ein Modell der Plattform, welches zu Testzwecken dient. Es werden zunächst die mathematischen Grundlagen der Inversen Kinematik erarbeitet aufbauend auf einem zuvor definierten Bewegungsmodell. Es folgt die Entwicklung einer generischen Architektur zur Übermittlung und Auswertung von Steuerkommandos vom PC. Die Implementierung geschieht in C und wird in verschiedene Module aufgeteilt, welche jeweils einen Aufgabenbereich der Positionskontrolle oder der Hardwarekommunikation abdecken. Es wird zudem eine graphische Nutzeroberfläche vorgestellt, über die man die Position der Plattform manuell verändern kann. Eine automatische Ansteuerung wird im folgenden Anwendungsbeispiel beschrieben, wo die Plattform mit frequentiellen Beschleunigungswerten einer Achterbahnsimulation beliefert wird.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

1.1 Beschreibung und Anwendungsgebiete

1965 beschrieb D. Stewart in seinem Paper eine parallel-kinematische Konstruktion mit sechs Freiheitsgraden, deren Bewegung über die Kombination von sechs Aktoren kontrolliert werden kann. Von Stewart als Flugsimulator vorgeschlagen, wurde bereits 1947 von Gough ein ähnliches Konzept vorgestellt, das als Reifenprüfstand diente. Dieses Konstrukt bekam folglich den Namen Stewart-Gough-Plattform oder kurz Stewart-Plattform (SP). Die Abbildungen 1.1 und 1.2 zeigen einen Vergleich der ersten Entwürfe.

Die Stewart-Plattform besteht aus einer stationären Plattform (Basis) und einer mobilen Plattform. Verbunden werden diese über sechs Verbindungsstreben (Beine), welche an Kugelgelenken befestigt sind. Es existieren mehrere Varianten der SP, im späteren Verlauf wird sich ausschließlich mit der 6-6-Konstruktion beschäftigt, das heißt, es existieren je sechs Verbindungspunkte an der Basis sowie der mobilen Plattform. Im Gegensatz dazu stehen beispielsweise die 3-3- oder 6-3-Varianten mit einer unterschiedlichen Anzahl an Verbindungspunkten der beiden Plattformen. In der Originalkonstruktion bestehen die Beine aus in der Länge änderbaren Linearaktoren. Die Position und Orientierung der mobilen Plattform wird durch eine Kombination der sechs Beinlängen erreicht. Anzumerken ist hierbei, dass sich die Längen nicht individuell beziehungsweise unabhängig voneinander ändern lassen, sondern nur innerhalb des Rahmens, den die Konstruktion und die Bewegung erlauben. Eine solche Bewegung besitzt sechs Frei-

heitsgrade oder Degrees of Freedom (DoF), welche sich in drei Rotations-DoF (roll, pitch, yaw) zur Orientierung und drei Translations-DoF (horizontal, vertical, lateral) zur Positionierung aufteilen. Die SP gehört wie bereits erwähnt zu den

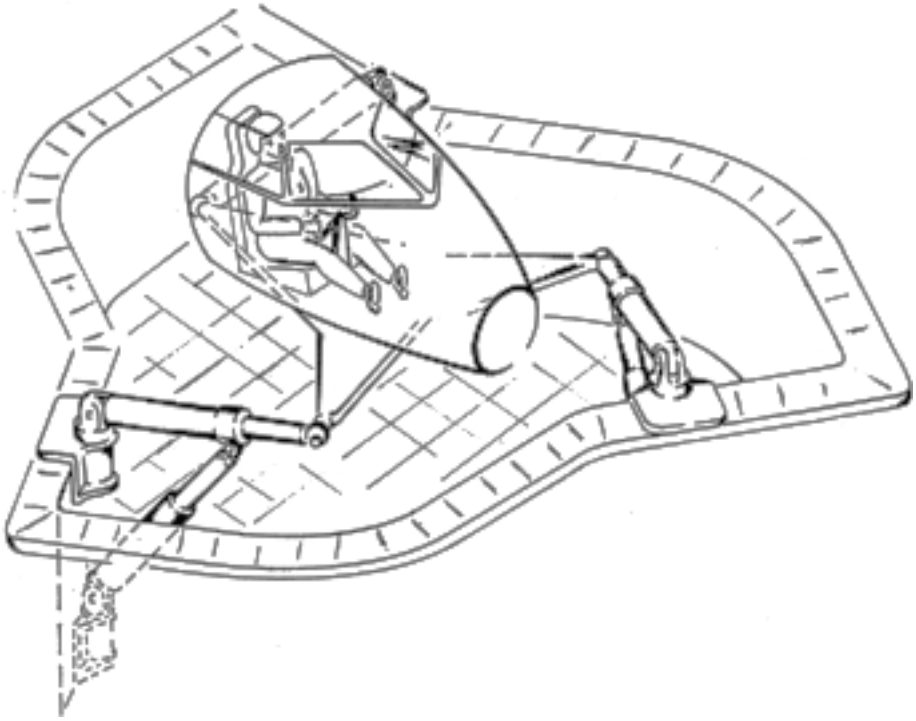


Abbildung 1.1: Der originale 6-3-Entwurf als Flugsimulator von D. Stewart aus seinem 1965 erschienenen Paper

parallelen Mechanismen, denen gegenüber die seriellen stehen. Parallelmanipulatoren haben eine sehr robuste mechanische Kontruktion, denn bedingt durch die große Anzahl an Verbindungspunkten besitzt die Plattform eine hohe Struktursteifigkeit. Zudem wird durch die Verteilung der Last auf alle sechs Aktoren ein sehr gutes Verhältnis von Lastaufnahme zu Eigengewicht erreicht, was die sichere Bewegung hoher Nutzlasten ermöglicht. Die beweglichen Teile (Beine, Linaeraktoren, mobile Plattform) besitzen eine geringe Masse und erlauben so eine schnelle und effiziente Positionierung. Ein serieller Manipulator muss zusätzlich zu seinem eigenen Gewicht das aller folgenden Aktoren tragen, was zum einen hohe Kraftausübungen, zum anderen hohe Geschwindigkeiten, vor allem unter starken Belastungen, unmöglich macht. Ein weiterer Vorteil, der durch die parallele Anordnung entsteht, ist die Beschränkung des Positionierungsfehlers der Plattform. Bei serielle Mechanismen akkumulieren sich die Positionsfehler der

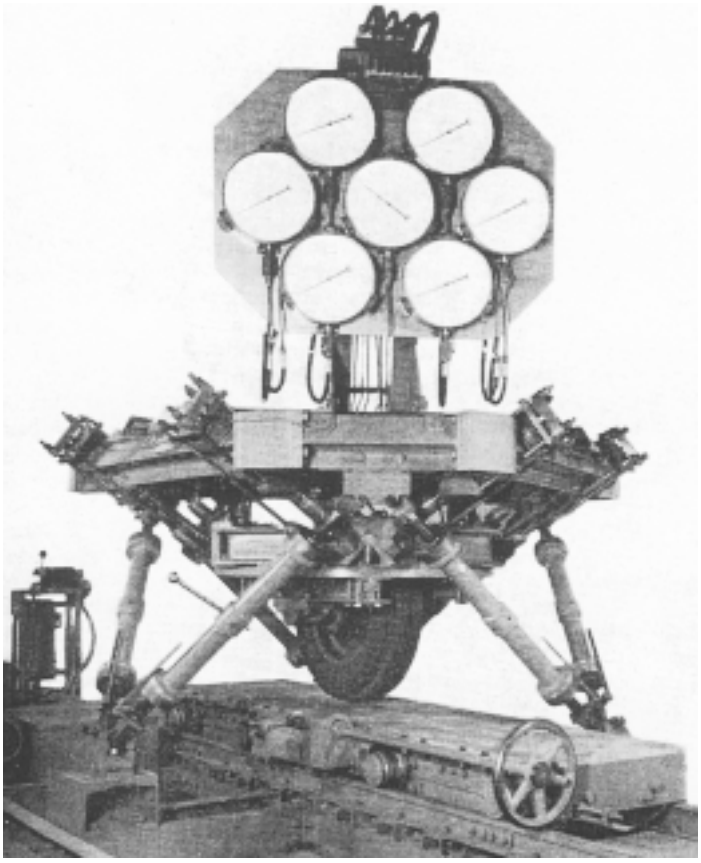


Abbildung 1.2: Gough's Reifentestmaschine kommt dem in dieser Arbeit vorgestellten 6-6-Ansatz näher als die originale Stewart-Plattform

einzelnen Aktoren über eine Sequenz von Verbindungen, was diese unbrauchbar für präzise Aufgaben macht. Der Positionsfehler einer parallelen Konstruktion ist beschränkt durch den durchschnittlichen Fehler der einzelnen Beinpositionen. Einzelne Fehler addieren sich nicht auf, wodurch eine sehr hohe Präzision erreicht wird, die selbst Bewegungen im Nanometer-Bereich ermöglicht. Ein Nachteil, der sich aus dem Aufbau ergibt, ist der relativ geringe Arbeitsraum, welcher beschränkt wird durch die maximalen Beinlängen sowie die Dimension und den maximalen Winkel der Kugelgelenke.
Das Hauptproblem der SP ist die Komplexität der Bewegungskontrolle. Das Problem, die Beinlänge bei gegebener Position der mobilen Plattform zu finden, nennt sich Inverse Kinematik. Das Gegenteil, die Position über die gegebenen Beinlängen zu ermitteln, nennt sich Vorwärtskinematik. Bei Parallelmanipulatoren ist die Inverse Kinematik wenig komplex, da sie durch die Berechnung

der Beinlängen der einzelnen Aktoren erfolgt, wobei Position und Orientierung der mobilen Plattform und daraus folgend die Verbindungspunkte der Beine bekannt sind. Die Vorwärtskinematik ist uneindeutig und erfordert die Lösung vieler nicht-linearer Gleichungen. Im Zusammenhang mit der Stewart-Plattform ist die Inverse Kinematik ohnehin bedeutender, denn zumeist werden bereits bestehende Positionierungsdaten genutzt um die Plattform auszurichten. Ein weiterer Vorteil, der sich aus der parallelen Anordnung ergibt, ist, dass sich die Beinlängen unabhängig voneinander berechnen lassen. Dies ermöglicht eine parallele Bestimmung der Aktorkonfigurationen, was wiederum der Geschwindigkeit der Konstruktion dienlich ist.

Die zuvor beschriebenen Eigenschaften eröffnen der SP weitreichende Einsatzgebiete, zum Beispiel in der Industrie, der Robotik, der Medizin, der Flugausbildung sowie der Astronomie und Raumfahrt.
In der Industrie können mithilfe der SP große Lasten sehr schnell und effizient bewegt werden. Ein Beispiel hierfür ist der Robocrane [Rob12], eine Variation der SP, bei der die mobile Plattform an sechs Kabeln hängt. Damit ist es möglich, Werkzeuge und Material auf einer Baustelle schnell von einem Ort zum nächsten zu transportieren, beispielsweise im Schiff- oder beim Brückenbau.
Auch im medizinischen Bereich findet die Technik immer mehr Einsatzgebiete. So wurden bereits präzise chirurgische Operationen mithilfe des SP-Mechanismus durchgeführt. Ein weiteres Beispiel ist der Taylor Spatial Frame [Tay12], eine Vorrichtung zur Stabilisierung von Knochen.
Der erste Entwurf von Stewart sah die Konstruktion als Flugsimulator vor, als was sie auch heute noch genutzt wird. Durch die sechs Freiheitsgrade bietet die SP eine realistische Simulation von Fahr- beziehungsweise Flugverhalten und der Bewegung im Raum. Flugsimulatoren, welche sechs Freiheitsgrade nutzen, werden auch Full Flight Simulator [Flu12] genannt.
Das Astronomische Institut der Universität Bochum entwickelte eigens für ihre Zwecke ein Hexapodteleskop [Tel12], bei dem die Linse durch den Stewart-Mechanismus ausgerichtet werden kann. Die Robustheit der Konstruktion sowie die bereits angeführte Präzision der Positionierung waren hierbei die entscheidenden Qualitätsmerkmale.
Die Vielfältigkeit der Einsatzgebiete zieht sich bis in die Raumfahrt mit dem patentierten Low Impact Docking System der NASA [Nas12], einem Kopplungsme-

chanismus für Raumfahrzeuge, das auf der SP basiert.
Unter Einsatz von entsprechender Sensorik kann man mit der SP zudem Stabilisierungsoperationen ausführen, etwa um den Wellengang auf einem Schiff auszugleichen, sollte man eine stabile Umgebung zur Ausführung sensibler Aufgaben benötigen.

1.2 Motivation und Ziel

Wie in der Einführung bereits beschrieben, bietet die Stewart-Plattform zahlreiche Anwendungsmöglichkeiten. Durch die Flexibilität der Konstruktion kann der Mechanismus in einer Vielzahl von universitären Projekten eingesetzt werden. Vornehmlich Simulatoren und Echtzeitanwendungen (Ausgleich von Bewegungen) sowie Robotik-Anwendungen (präzise Positionierung von Bestandteilen) können um diese Technologie erweitert werden und davon profitieren.
Das Ziel dieser Bachelorarbeit ist es, eine generische Softwarelösung in C zu erarbeiten, welche die nutzerfreundliche Ansteuerung einer Stewartplattform ermöglicht. Dazu wird ein lauffähiges Modell der Plattform benötigt, welches über die serielle Schnittstelle eines Computers angesteuert werden kann. Im Rahmen der Arbeit wird vom ursprünglichen Konzept der Linearaktoren abgewichen und es werden stattdessen Drehservomotoren genutzt. Durch Modularisierung der Software mit austauschbaren Hardwarekomponenten wird von der Hardware abstrahiert und eine Aktoren- und Plattform-unabhängige Ansteuerung ermöglicht.
Der Plattformprototyp, dessen Struktur und Bestandteile im Kapitel [Aufbau der Plattform] beschrieben werden, wird gesteuert über einen AtMega16 Mikrocontroller. Statt Hydraulikstäben wurden preisgünstigere Modellbauservomotoren als Aktoren gewählt. Weitere elektronische Komponenten umfassen zwei Platinen, welche das technische Framework des Prozessors sowie der Aktoren bilden. Zu jeder Hardwarekomponente werden (Schalt-)Pläne bereitgestellt und erläutert. Anhand dieser Hardware-Voraussetzungen wird die später bestimmte generische Architektur angepasst und spezifiziert.
Dazu müssen jedoch zunächst im Kapitel [Entwurf der Softwarearchitektur] die mathematischen und die logischen Grundlagen der Plattformbewegung erarbeitet werden. Die mathematischen Grundlagen enthalten zum einen das Bewegungsmodell und die Repräsentation der mobilen Plattform, zum anderen die

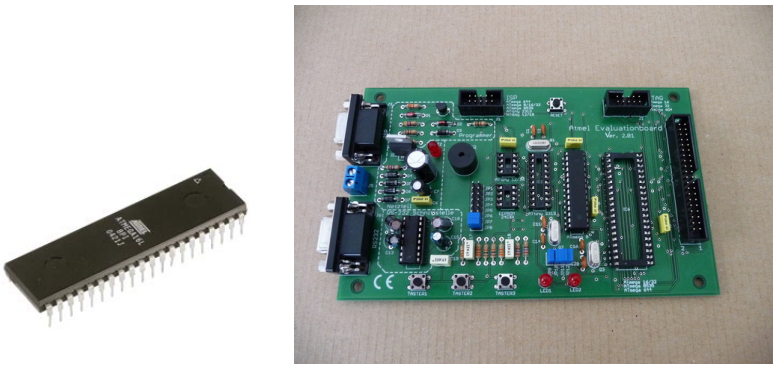
Lösung der Inversen Kinematik, welche die Grundlage der Positionierung bildet. Dabei werden bereits erste Anpassungen der mathematischen Algorithmen vorgenommen, um dem Plattform-Prototyp zu entsprechen. Es folgen die logischen Grundlagen mit einer Spezifizierung der Softwarearchitektur, welche die Interaktion der internen Komponenten anhand von mehreren Modellsichten beschreibt. Dieser Teil ist generisch gehalten und sieht eine Modularisierung in austauschbare Peripherie- und unveränderliche Kern-Komponenten vor.
Die Implementation der Architektur erfolgt im darauffolgenden Kapitel [Programmierung], wo das zuvor erarbeitete Klassenmodell in Module der Programmiersprache C überführt wird. Mit Hilfe der mathematischen und logischen Grundlagen des vorangegangenen Kapitels werden die einzelnen Methoden der Module ausgearbeitet. Die Methoden und Klassen sowie ihre Funktionalität werden dabei im Detail beschrieben. Den Abschluss des Kapitels bildet die Beschreibung einer graphische Benutzeroberfläche, welche in Object-Pascal geschrieben wurde und dem Nutzer eine einfache Möglichkeit bieten soll die Plattform zu bewegen.
Den Abschluss des Hauptteils bildet ein Anwendungsbeispiel, in welchem eine Achterbahnsoftware eines vergangenen Projektpraktikums erweitert wird, um Beschleunigungsdaten an die Plattform zu schicken. Dies ist eine der einfachsten Arten der Echtzeitsimulation von Bewegungen, da nur drei von sechs möglichen Transformationen genutzt werden. Es folgt eine Auswertung der Anwendung, in der überprüft wird, inwiefern die Plattform Echtzeitanforderungen unter Belastung genügte.
Der Schluss der Arbeit umfasst zum einen eine Zusammenfassung des Geleisteten, zum anderen einen Ausblick, in dem weitere Ideen zur Qualitätsverbesserung und Erweiterung der Funktionalität der Plattform gegeben werden.

Kapitel 2

Aufbau der Plattform

Im Rahmen dieser Arbeit wurde zu Testzwecken ein eigenes Modell der Stewart-Plattform erstellt. Dieses umfasst sowohl den Rahmen der Plattform als auch die Elektronik und das Framework von Mikrocontroller und Aktorik.

2.1 Mikrocontroller und Board



(a) Der AtMega16 (b) Das Evaluationsboard

Abbildung 2.1: Technische Bestandteile der Plattform

Als Prozessor wird ein AtMega16 der Firma Atmel genutzt. Dieser besitzt 16kB programmierbaren Flashspeicher und vier Ports (A,B,C und D) mit jeweils acht Ein-/Ausgängen. Zusätzlich besitzt der Mikrocontroller nützliche Timer- sowie Uartfunktionen, welche später der Ansteuerung der Hardware dienen. Ange-

bracht ist der AVR auf einem Evaluationsboard von Pollin, welches eine Steckvorrichtung zum Auslesen der einzelnen Ports bereitstellt.

2.2 Servomotoren und Platine

Als Aktoren wurden aus Kostengründen Modelcraft Servomotoren von Pollin gewählt, siehe Abbildung 2.2. Servomotoren besitzen drei Eingänge, darunter

Abbildung 2.2: Modelcraft Standard-Servo RS 2 mit einem Konfigurationsraum von 90°.

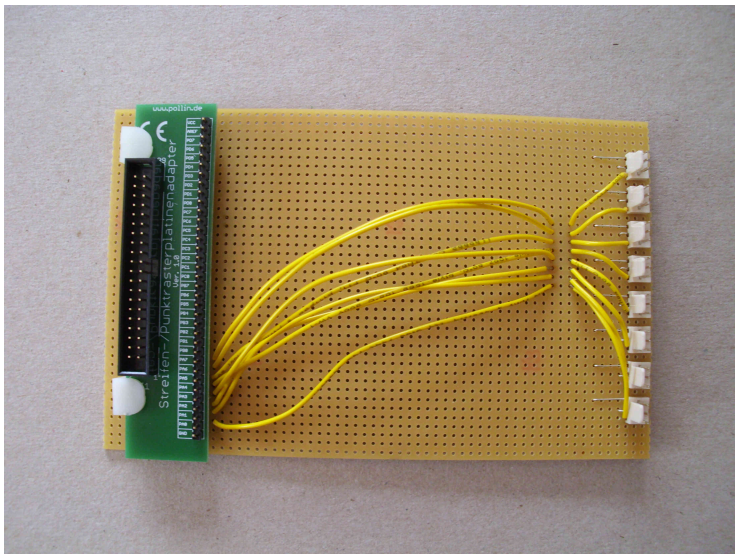


Abbildung 2.3: Steuerplatine mit Streifen-/Punktrasterplatinenadapter (links) und Steckvorrichtungen (rechts)

eine Versorgungsspannung von 5V bis 9V sowie einen Massepegel. Die Ausrichtung wird durch den dritten Eingang, einem pulsweitenmodulierten Signal bestimmt (siehe Kapitel [Befehlsübertragung]). Angebracht werden die Motoren in

Paaren pro Seite unter der Basisplattform. Jeder zweite ist dabei entgegengesetzt zu seinem Nebenmotor ausgerichtet; genaueres dazu im Unterkapitel [Rahmen der Plattform].

Zur Ansteuerung der Servomotoren wurde eine eigene Platine (Abbildung 2.3)

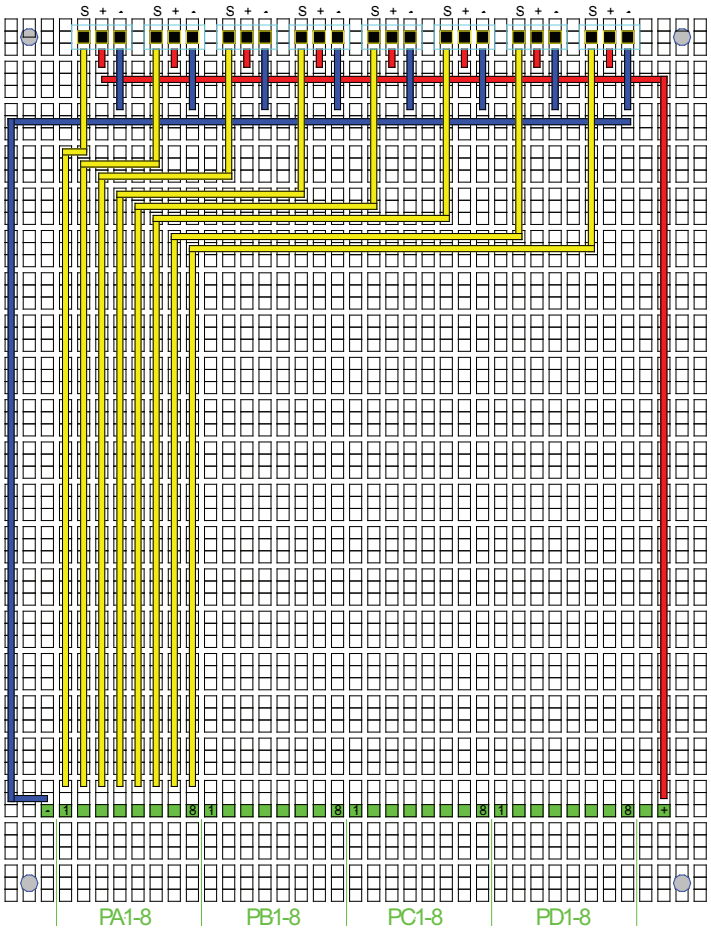


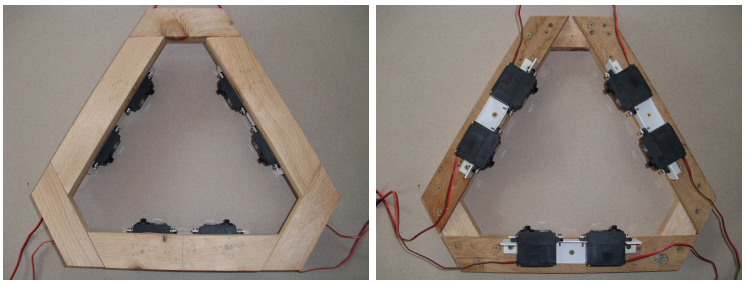
Abbildung 2.4: Input: Streifen-/Punktrasterplatinenadapter (grün), Output: Steckvorrichtungen (schwarz), Unterseite: Spannungs- (rot) und Masse-Leitung (blau), Oberseite: Spannungs- (rot) und Massebrücken (blau) zu den Steckern sowie Kabel für Signalübertragung (gelb)

entworfen, welche Signale des Evaluationsboards entgegen nimmt und diese als Output an die Motoren weiterleitet. Jeder Motor wird an einen Stecker angeschlossen und dadurch mit Spannung, Masse und einem Impuls versorgt. Abgegriffen werden diese Signale an einem Streifen-/Punktrasterplatinenadapter [Pol12], der über einen 40-poligen Flachbandstecker mit dem Evaluationsboard

verbunden ist.
Abbildung 2.4 zeigt den Plan der Platine. Von dem angesprochenen Adapter werden sowohl die interne Versorgungsspannung (+) als auch Massepegel (-) des AtMega abgegriffen und über zwei separate Leitungen an die Stecker weitergeleitet. Als Signalinput der Pulsweitenmodulation ist der PortA (PA1-8) des Mikrocontrollers vorgesehen, von dem aus acht Kabel acht verschiedene Stecker mit einem Impuls (S) versorgen.

2.3 Rahmen der Plattform

Der Rahmen der Plattform besteht aus drei Hauptbestandteilen, der mobilen und der festen Plattform sowie der Verbindungsstreben. Beide Plattformen bestehen aus einer Holzkonstruktion, wobei die obere Plattform merklich kleiner als die untere ist. Die Verbindungsstreben (Beine) bestehen aus Metall und haben an jedem Ende je ein Kugelgelenk angebracht. Die Basisplattform besteht aus sechs



(a) Obere Seite der Basis (b) Untere Seite der Basis

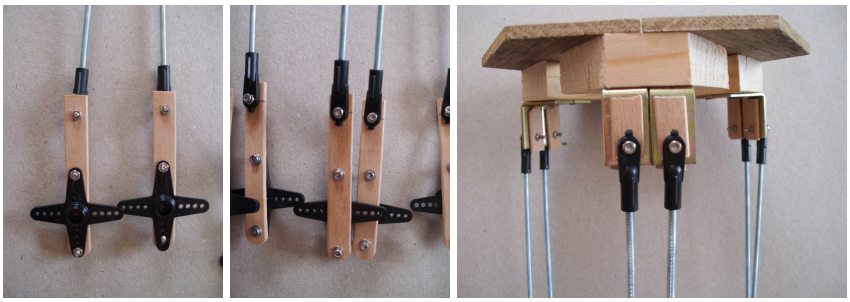
Abbildung 2.5: Beidseitige Ansicht des Modells der Basisplattform

trapezförmigen Einzelteilen, welche zusammengeklebt und -geschraubt wurden (Abbildung 2.5). Die mobile Plattform besteht aus einer oberen Platte und drei ebenfalls trapezförmigen Eckstücken darunter, an denen je zwei metallene Winkel angebracht sind (Abbildung 2.7(c)). Jeder Winkel besitzt sowohl auf Vorder- als auch Rückseite je ein Zwischenstück aus Holz, an welchen später die Kugelgelenke der Beine festgeschraubt werden (Abbildung 2.6(b) und 2.7(c)). Die Befestigung der Servomotoren an der unteren Plattform wurde durch Bildung einer Einlassung aus Plastikwinkeln erreicht (Abbildung 2.5(b) und 2.6(a)).
Wie in der Abbildung 2.6(a) zu sehen ist, sind die Servomotoren einer Seite paar-

(a) Gesamtansicht der Motoren und Beine.

(b) Der Aufsatz des Modells.

Abbildung 2.6: Die mobile Plattform und deren Befestigung auf der Basis.



(a) Hinterseite des Servosterns

(b) Vorderseite des Servosterns

(c) Die Befestigung der Beine an den Winkeln über Holzverbindungen.

Abbildung 2.7: Modifizierte Motorenaufsätze und -befestigung

weise entgegengesetzt angeordnet angebracht. Zur Erweiterung des Radius des Drehaufsatzes (Stern) kamen kleine flache Holzstäbe zum Einsatz (Abbildung 2.7(a) und 2.7(b)). Die Verbindungsstreben werden jeweils mithilfe eines Kugelgelenks (Größe M2) auf diesem Aufsatz festgeschraubt. Das Beinpaar einer langen Basisseite führt zu den Winkeln einer Eckseite der mobilen Plattform (Abbildung 2.9).

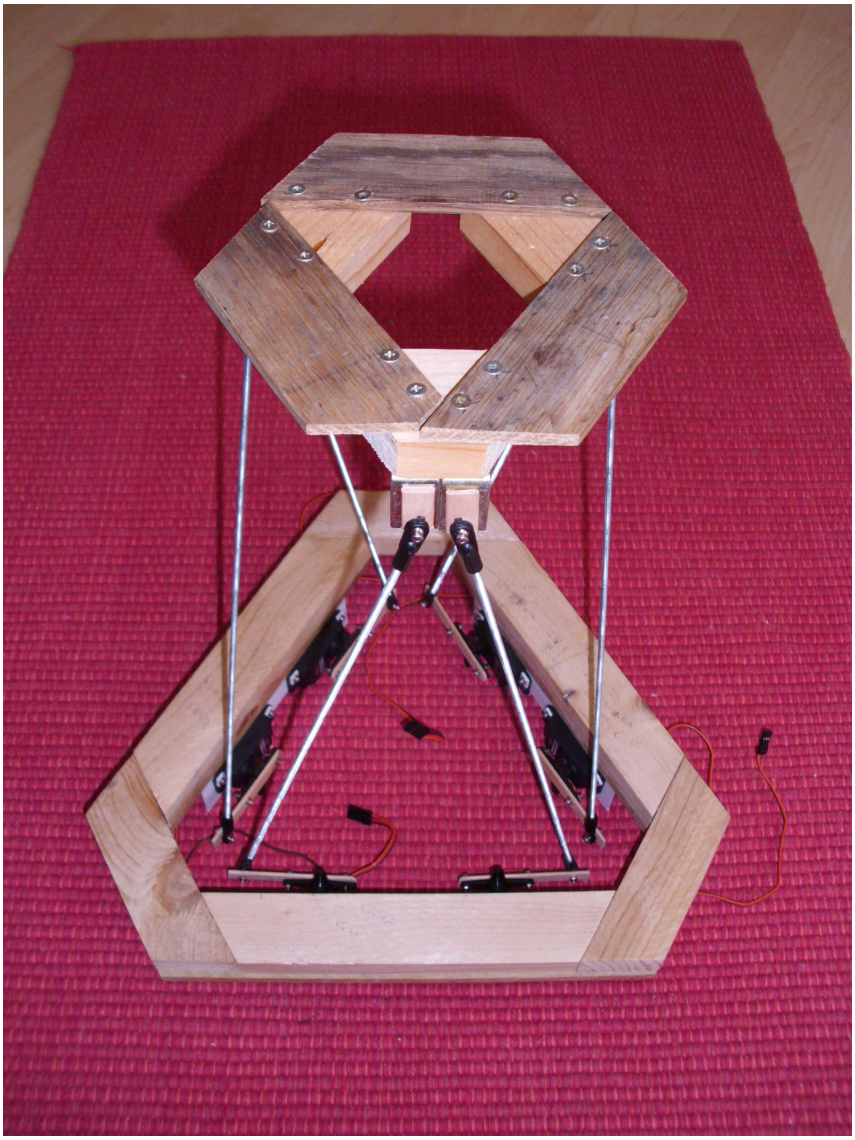


Abbildung 2.8: Frontalansicht des Plattformmodells.

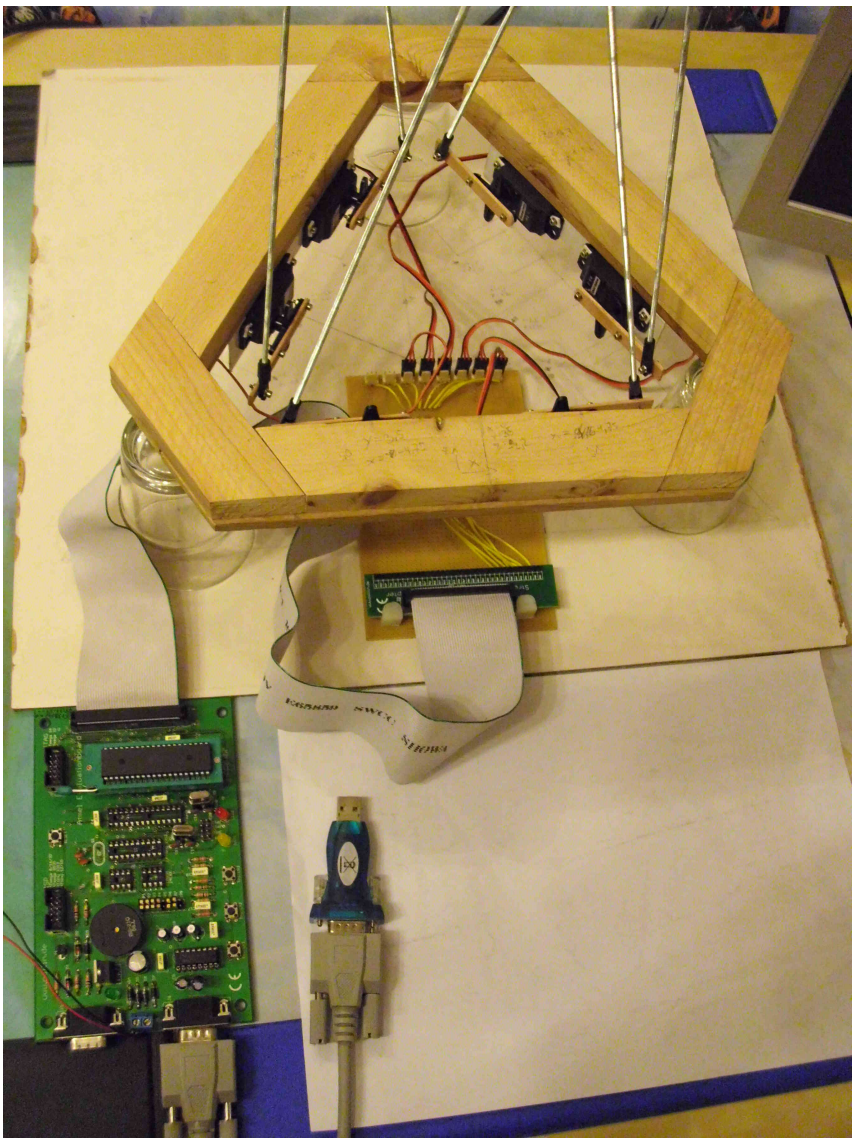


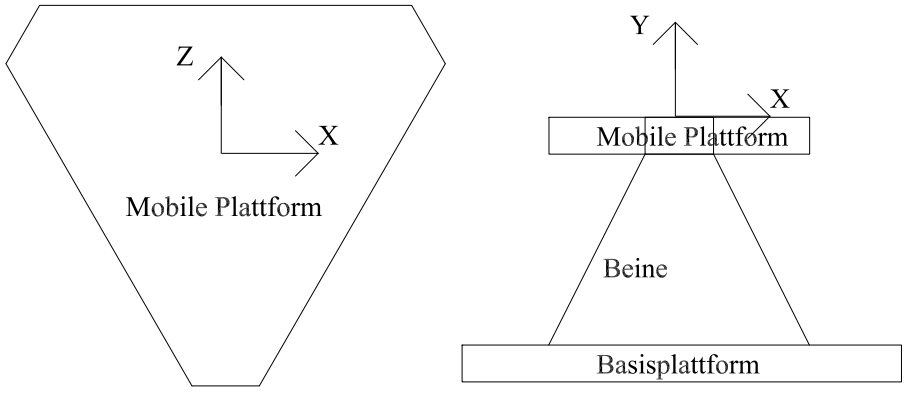
Abbildung 2.9: Die Servomotoren werden über die Steckvorrichtungen mit der Übertragungsplatine verbunden. Das Evaluationsboard wird über die serielle Schnittstelle (USB-Seriell-Adapter) an den PC angeschlossen. Als Brücke zwischen den beiden Platinen dient ein Flachbandstecker.

Kapitel 3

Entwurf der Softwarearchitektur

3.1 Bewegungsmodell

Im Folgenden wird zunächst das klassische Bewegungsmodell der Plattform betrachtet, welches den Einsatz von Linear-Aktoren voraussetzt. Das bedeutet, dass der Basisbefestigungspunkt der Streben statisch und die Beinlänge variabel ist. Dieses Modell vermittelt die allgemeine Herangehensweise für die Repräsentation der Positionsinformationen und wird im darauffolgenden Kapitel [Inverse Kinematik] an die Hardware-spezifischen Anforderungen angepasst. Um voll-



(a) Ansicht von oben (b) Frontalansicht

Abbildung 3.1: Das Koordinatensystem im Zentrum der mobilen Plattform aus verschiedenen Perspektiven.

kommene Kontrolle über die Ausrichtung der Plattform zu erhalten, muss für jeden Punkt stets bekannt sein, wo er sich im Raum befindet. Grundlage für die Aufzeichnung von Bewegungen im Raum ist ein globales Koordinatensystem, dessen Ursprung O_0 sich im Zentrum der mobilen Plattform (in Ruhelage) befindet (Abbildung 3.1). Hierbei handelt es sich um ein Linkssystem. Die x-Achse $\overrightarrow{x} = (1,0,0)^T$ zeigt nach rechts parallel zur gegenüber liegenden Plattformseite, die y-Achse $\overrightarrow{y} = (0,1,0)^T$ nach oben und die z-Achse $\overrightarrow{z} = (0,0,1)^T$ nach vorne in den Raum. Eine Längeneinheit entspricht dabei einem Millimeter.

Den Verbindungspunkten der freien Plattform sowie der Basis werden wie in Ab-

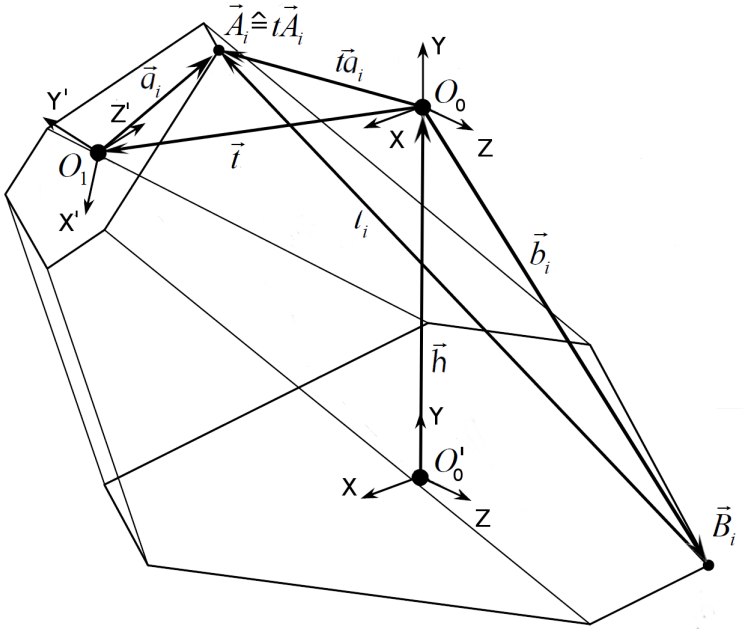


Abbildung 3.2: Das klassische Plattformmodell mit Linearaktoren. Das Koordinatensystem mit Ursprung O_0' entspricht dem um die Plattformhöhe $\overrightarrow{h}$ verschobenen System O_0 und dient der Veranschaulichung dessen Position im Raum.

bildung 3.2 beschrieben Koordinatenvektoren zugeordnet. Während die Punkte $\overrightarrow{B_i}$ (i $\in$ {1 .. 6}) der Basis (in globalen Koordinaten, $\overrightarrow{b_i}$) unveränderlich sind, hängen die Eckpunkte der Plattform im Folgenden von dem betrachteten Koordinatensystem ab. Die äquivalenten Punkte $\overrightarrow{tA_i}$ und $\overrightarrow{A_i}$ entsprechen zum einen den transformierten Punkten im Kontext des O_0-Systems (Koordinatenvektor $\overrightarrow{ta_i}$), zum anderen den Ruhepunkten in Bezug auf das O_1-System (Koordinatenvek-

tor $\overrightarrow{a_i}$), welches das transformierte Koordinatensystem bezeichnet. Gegeben sind zunächst nur die Koordinaten der Ruhepunkte $\overrightarrow{A_i}$, welche in die transformierten $\overrightarrow{tA_i}$ überführt werden müssen. Diese Überführung geschieht standardmäßig durch eine Koordinatentransformation [Koo12]. Dazu müssen sowohl die Rotationsmatrix R als auch der Translationsvektor $\overrightarrow{t}$ bekannt sein, die das System O_0 nach O_1 transformieren.
Die Plattform besitzt sechs Freiheitsgrade und daraus folgend sechs verschiedene Arten der Transformation, denen das Ruhesystem, also die mobile Plattform, unterworfen werden kann. Diese werden unterschieden in je drei Rotationen (pitch, yaw, roll) und Translationen (horizontal, vertical, lateral). Als Rotations- beziehungsweise Translations-Achsen dienen die Koordinatenachsen von O_0. Die Transformationen werden in zwei verschiedenen Strukturen gespeichert, Translationen in einem Vektor, Rotationen in einer Matrix. Diese beschreiben genau eine Position der Plattform im Raum, denn sie enthalten jeweils alle Transformationsinformationen relativ zur Ruhelage des Systems. Es werden zunächst die Koordinatenachsen entsprechend der Rotationsmatrix gedreht und daraufhin der Ursprung um den Translationsvektor verschoben. Für jeden Eckpunkt ergibt sich nach [Koo12] folgende Gleichung:

$$\overrightarrow{tA_i} = R * \overrightarrow{A_i} - \overrightarrow{t} \tag{3.1}$$

Der Translationsvektors $\overrightarrow{t}$ beschreibt pro Zeile die Verschiebung in Richtung der entsprechenden Koordinatenachse. Abbildung 3.2 zeigt $\overrightarrow{t}$ als den Vektor zwischen den Koordinatenursprüngen O_0 und O_1 in jeweils globalen Koordinaten.

$$\overrightarrow{t} = \begin{pmatrix} O_{1,x} - O_{0,x} \\ O_{1,y} - O_{0,y} \\ O_{1,z} - O_{0,z} \end{pmatrix} = \begin{pmatrix} O_{1,x} \\ O_{1,y} \\ O_{1,z} \end{pmatrix} \tag{3.2}$$

Die vollständige Rotationsmatrix *R* beschreibt entsprechend die Unterschiede der Achsenausrichtung der beiden Koordinatensysteme in Abhängigkeit von drei Raumwinkeln. Sie ist eine Akkumulation der einzelnen Rotationsmatrizen, welche jeweils die Drehung um eine der Achsen beschreiben.

$$R = R_x(\alpha) * R_y(\beta) * R_z(\gamma) \tag{3.3}$$

Dabei bezeichnen α, β und γ die Drehwinkel gegen den Uhrzeigersinn um die spezifizierte Achse. Die einzelnen Matrizen haben folgende Struktur:

Rotationsmatrix der x-Achse:

$$R_x(\alpha) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & cos\alpha & sin\alpha \\ 0 & -sin\alpha & cos\alpha \end{pmatrix} \tag{3.4}$$

Rotationsmatrix der y-Achse:

$$R_y(\beta) = \begin{pmatrix} cos\beta & 0 & -sin\beta \\ 0 & 1 & 0 \\ sin\beta & 0 & cos\beta \end{pmatrix} \tag{3.5}$$

Rotationsmatrix der z-Achse:

$$R_z(\gamma) = \begin{pmatrix} cos\gamma & sin\gamma & 0 \\ -sin\gamma & cos\gamma & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{3.6}$$

Dieses Bewegungsmodell ist ein sehr oft genutztes zur Beschreibung von Transformationen im Raum. Es bildet die Grundlage der Ansteuerung der Plattform und der Positionskontrolle. Ein Vorteil dieser Form der Beschreibung ist die Möglichkeit, eine Transformation zunächst zwischenzuspeichern, ohne die Auswirkungen direkt an die Hardware übermitteln zu müssen, was im Unterkapitel [Befehlsauswertung] ausgenutzt wird.

Den Fokus des nächsten Kapitels bildet die Überführung der Transformationen des Bewegungsmodells in Befehle für die Servomotoren. Dazu wird die angepasste inverse Kinematik der Struktur analysiert.

3.2 Inverse Kinematik

Die Inverse Kinematik einer mechanischen Konstruktion beschreibt das Problem, zu einer gegebenen Position im Arbeitsraum die aktuelle Konfiguration der Aktoren zu bestimmen. Bei seriellen Mechanismen ist die Lösung der Inversen Kinematik uneindeutig und komplex, bei parallelen Mechanismen wie der Stewart-Plattform ist das Gegenteil der Fall. Die zu bestimmenden Konfigurationen der Aktoren sind bei der klassischen Stewart Plattform die Beinlängen der einzelnen Verbindungsstreben. Die Berechnung dieser Längen ist sehr einfach durch das im vorangegangen Kapitel beschriebene Bewegungsmodell. Durch den Einsatz eines kartesischen Koordinatensystems und die Aufzeichnung der Transformationen sind die Positionen der Eckpunkte der mobilen sowie der stationären Plattform im Raum stets bekannt. Die Länge des Vektors zwischen beiden Punkten ergibt die gesuchte Beinlänge l_i.

$$l_i = |\overrightarrow{tA_i} - \overrightarrow{B_i}| \tag{3.7}$$

für alle i $\in$ {1,...,6}

Durch den Einsatz von Servomotoren ändern sich jedoch die Voraussetzungen und der Konfigurationsraum der Aktoren. Wurde zuvor ein linearer Konfigurationsraum mit Längenwerten angenommen, ändert sich dieser im Rahmen des erstellten Modells in einen Konfigurationsraum von Drehwinkeln mit verschiedenen Ausrichtungen der Servomotoren. Die genutzten Motoren besitzen (wie im Kapitel [Aufbau der Plattform] beschrieben) einen sternförmigen Aufsatz mit einer hölzernen Verlängerung zu einer Seite, an deren Enden die Drehgelenke der Streben angebracht werden. Dadurch werden die zuvor stationären Eckpunkte der Basis zu dynamischen, vom Drehwinkel abhängigen Punkten. Im Gegenzug bleiben die Beinlängen L konstant. Die Verbindungspunkte der Plattformen entsprechen den jeweiligen Positionen der Kugelgelenke der Beine im Raum. Abbildung 3.3 beschreibt den groben Aufbau der Konstruktion.
Den zuvor zu bestimmenden Beinlängen entsprechen nun die Abstände der Drehpunkte der Servos $\overrightarrow{S_i}$ von den transformierten Punkten $\overrightarrow{tA_i}$. Gesucht ist jedoch entsprechend der Änderung des Konfigurationsraums der einzustellende Winkel statt der Beinlänge eines jeden Aktors. Es wird also im Vergleich zu dem aus Kapitel [Bewegungsmodell] gefolgerten Ansatz 3.7 ein solcher benötigt, der eine

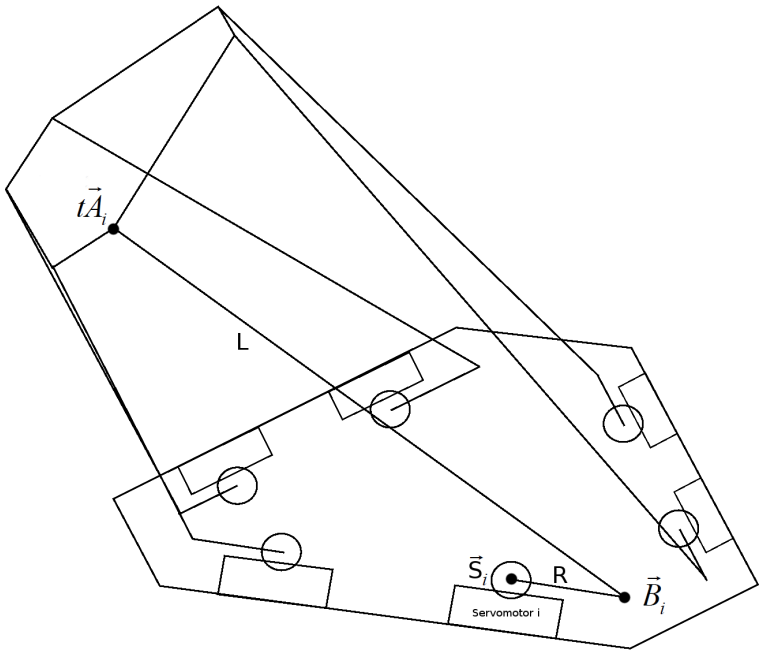


Abbildung 3.3: Das geänderte Plattformmodell unter Einsatz von Servomotoren. Die Position und Ausrichtung der Koordinatensysteme sind unverändert, wurden aber zur Übersicht nicht eingezeichnet. Alle Punktkoordinaten beziehen sich auf das globale Koordinatensystem.

eindeutige Abhängigkeit des Drehwinkels von der Punktposition beinhaltet. Dieser neue Ansatz baut auf einige im Folgenden vorgestellten Vorüberlegungen auf.

Die einzelnen transformierten Eckpunkte befinden sich zunächst noch im globalen Koordinatensystem O_0. Da jeder Servomotor eine individuelle Ausrichtung besitzt mit den unterschiedlichen Drehpunktkoordinaten S_i als Referenzen, wird die Anwendung eines einheitlichen Algorithmus verkompliziert. Für jeden Eckpunkt sind daher Vorabtransformationen von Nöten, bevor die Berechnung des Winkels erfolgen kann. Daher wird zu Beginn für jeden Servomotor ein lokales Koordinatensystem definiert wie in Abbildung 3.4 beschrieben. Der Ursprung der verschiedenen lokalen Systeme liegt im jeweiligen Drehpunkt des Motors. Durch diese Maßnahme wird Symmetrie zwischen den einzelnen Motoren erreicht, denn die Drehpunkte der Motoren besitzen so dieselben Koordinaten in ihren jeweiligen Systemen ($(0,0,0)^T$) und dienen in der Folge als identische Referenzpunkte der zugehörigen Ecken $\overrightarrow{tA_i}$. Um einen Punkt vom globalen in eines der lokalen Systeme zu transformieren, benötigt man wie aus Kapitel [Be-

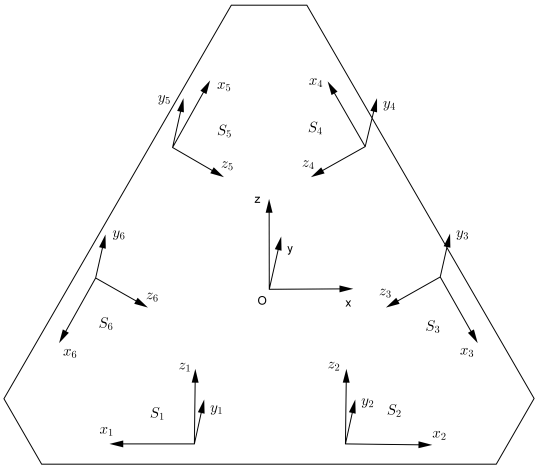


Abbildung 3.4: Jeder Motor erhält sein eigenes lokales Koordinatensystem. Die y-Achsen zeigen jeweils nach oben.

wegungsmodell] bekannt sowohl eine entsprechende Rotationsmatrix als auch einen Translationsvektor. Die folgende Tabelle zeigt die Transformationen, die notwendig sind, um aus dem globalen das entsprechende lokale Koordinatensystem zu erhalten.

Betrachteter Eckpunkt	tA_1	tA_2	tA_3	tA_4	tA_5	tA_6
Rotation um y-Achse	$0°$	$0°$	$120°$	$120°$	$-120°$	$-120°$
Translation um	S_1	S_2	S_3	S_4	S_5	S_6
Gespiegelte x-Achse	ja	nein	ja	nein	ja	nein

Die dazugehörigen Transformationsstrukturen sehen entsprechend 3.2 und 3.5 folgendermaßen aus:

$$\overrightarrow{t_i} = \overrightarrow{S_i} \qquad R_{y,i}(\alpha) = \begin{pmatrix} cos\alpha & 0 & -sin\alpha \\ 0 & 1 & 0 \\ sin\alpha & 0 & cos\alpha \end{pmatrix} \tag{3.8}$$

mit α als Drehwinkel aus obiger Tabelle.

Da sich alle Transformationen auf das globale Koordinatensystem beziehen, wird zunächst eine Überführung von Servo- in globale Koordinaten als Ansatz verwendet, entsprechend 3.1:

$$\overrightarrow{tA_i} = R_{y,i}(\alpha) * \overrightarrow{tA'_i} + \overrightarrow{t_i}$$

mit $\overrightarrow{tA'_i}$ als Eckpunkt in Servokoordinaten.

Dieser Ansatz kann nach $\overrightarrow{tA'_i}$ freigestellt werden:

$$\overrightarrow{tA'_i} = R^{-1}_{y,i}(\alpha) * (\overrightarrow{tA_i} - \overrightarrow{t_i}) \tag{3.9}$$

mit $R^{-1}_{y,i}(\alpha) = R_{y,i}(-\alpha)$ als inverser Rotationsmatrix.

Da die beiden Servomotoren auf jeder Seite der Basis jeweils entgegen gerichtet sind, muss die Hälfte der Koordinatensysteme in der x-Achse ebenfalls entgegen gerichtet sein, um die Symmetrie zu gewährleisten. Jeder ungerade nummerierte Motor hat deshalb eine an der yz-Ebene gespiegelte x-Achse, was einer Umkehr des Vorzeichens der x-Koordinate entspricht.

Die folgende Herleitung des Winkel-Algorithmus nutzt die bereits transformierten Werte der einzelnen Punkte, zur Einfachheit ebenfalls mit $\overrightarrow{tA}$ bezeichnet. Da die Positionsbestimmung der Eckpunkte und damit auch die Konfigurationen der Servomotoren im Rahmen der Beschränkungen der Plattform unabhängig voneinander erfolgt, kann die Bestimmung der Winkel ebenfalls unabhängig erfolgen.
Gegeben seien (zur Einfachheit ohne Indizes) der transformierte Punkt $\overrightarrow{tA}$ einer Ecke der mobilen Plattform sowie der dynamische Basispunkt B an der Spitze des Motorradius, dessen Koordinaten sich entsprechend des Drehwinkels ändern. Weiterhin bekannt sind die konstanten Längen des Beines L sowie des Drehradius R wie in Abbildung 3.5 gezeigt. Ziel ist es, für jeden Servomotor den Drehwinkel zu bestimmen, der für eine korrekte Positionierung des zugehörigen Eckpunktes sorgt. Wie in Abbildung 3.6 zu sehen lassen sich die Koordinaten von $\overrightarrow{B}$ mithilfe von trigonometrischen Funktionen berechnen, welche den Winkel θ

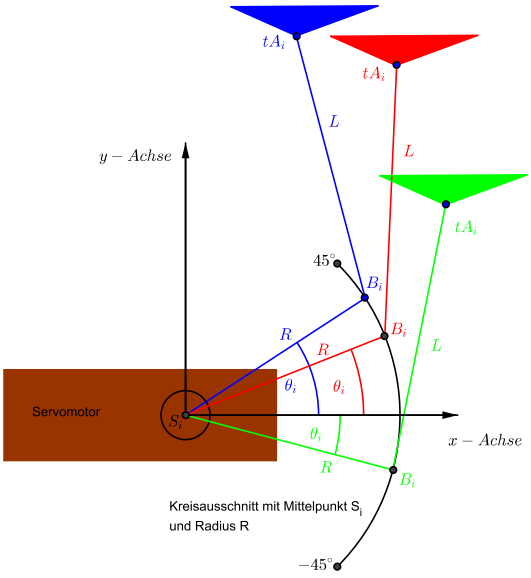


Abbildung 3.5: Die Frontalansicht des Koordinatensystems eines ungerade nummerierten Servomotors zeigt den Zusammenhang zwischen Auslenkung des Motors und den anderen angepassten Größen. Die gerade nummerierten Motoren haben eine entgegengesetzte x-Achse.

als Parameter nehmen. Die Verbindungspunkte haben folglich die Struktur:

$$\overrightarrow{tA} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \qquad \overrightarrow{B} = \begin{pmatrix} R * cos\theta \\ R * sin\theta \\ 0 \end{pmatrix}$$

Da sowohl beide Endpunkte der Strebe als auch deren Länge gegeben ist, ergibt sich folgender Ansatz:

$$L = |\overrightarrow{tA} - \overrightarrow{B}| = | \begin{pmatrix} x - R * cos\theta \\ y - R * sin\theta \\ z - 0 \end{pmatrix} | = \sqrt{(x - R * cos\theta)^2 + (y - R * sin\theta)^2 + z^2}$$

Diese Gleichung soll nach dem gesuchten Winkel θ freigestellt werden.

$$L^2 = x^2 - 2Rx * cos\theta + R^2 cos^2\theta + y^2 - 2Ry * sin\theta + R^2 sin^2\theta + z^2$$
$$L^2 = (x^2 + y^2 + z^2) + R^2(cos^2\theta + sin^2\theta) - 2R(x * cos\theta - y * sin\theta)$$

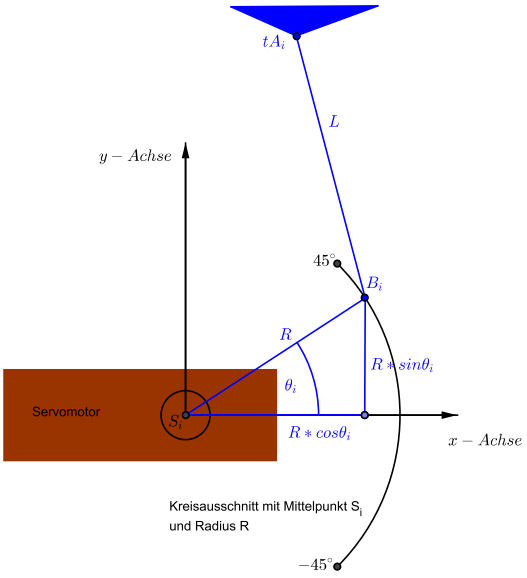


Abbildung 3.6: Der Basispunkt B liegt nicht mehr starr auf der unteren Plattform, sondern ist abhängig von der Motorenausrichtung.

Diese Gleichung kann über den trigonometrischen Pythagoras vereinfacht werden. Zudem entspricht der Ausdruck in der ersten Klammer der quadrierten Länge des betrachteten Eckpunktvektors.

$$L^2 = |\overrightarrow{tA}|^2 + R^2 - 2R(x * cos\theta - y * sin\theta)$$

Eine Freistellung nach dem gesuchten Winkel stößt dabei mit folgender Gleichung zunächst an seine Grenzen:

$$x * cos\theta - y * sin\theta = (|\overrightarrow{tA}|^2 + R^2 - L^2)/2R \tag{3.10}$$

Der gesuchte Winkel ist in zwei verschiedenen trigonometrischen Konstrukten enthalten, wodurch die Lösung der Gleichung nicht trivial zu bestimmen ist. Der folgende Lösungsweg nutzt die trigonometrische Identität der Linearkombination von Sinuswellen verschiedener Phasen [Sin12], um die Zahl der von θ abhängigen Parameter zu verringern. Es gilt:

$$a * sin(x) + b * sin(x) = \sqrt{a^2 + b^2} * sin(x + \phi)$$

mit

$$\phi = \begin{cases} arcsin(b/\sqrt{a^2+b^2}) & if a \geq 0, \\ \pi - arcsin(b/\sqrt{a^2+b^2}) & if a < 0, \end{cases}$$

oder vereinfacht beschrieben über die mathematische Hilfsfunktion atan2:

$$\phi = atan2(b, a)$$

Übertragen auf die vorherige Winkel-Gleichung 3.10 :

$$\sqrt{x^2+y^2} * sin(\theta + atan2(x, y)) = (|A|^2 + R^2 - L^2)/2R$$

Diese Gleichung enthält den Drehwinkel nur noch in einer einzigen trigonometrischen Funktion, lässt sich demnach nach θ freistellen.

$$sin(\theta + atan2(x, y)) = (|\overrightarrow{tA}|^2 + R^2 - L^2)/(2R\sqrt{x^2+y^2})$$
$$\theta + atan2(x, y) = arcsin((|\overrightarrow{tA}|^2 + R^2 - L^2)/(2R\sqrt{x^2+y^2}))$$

$$\theta = arcsin((|\overrightarrow{tA}|^2 + R^2 - L^2)/(2R\sqrt{x^2+y^2})) - atan2(x, y) \tag{3.11}$$

Die einzigen Parameter dieser Gleichung sind die Koordinaten des betrachteten Eckpunktes. Folglich kann der Algorithmus zum Bestimmen des Drehwinkels auf jeden Punkt unabhängig von den anderen angewandt werden.

3.3 Funktionale Anforderungen

Nun, da das Bewegungsmodell den Hardwarevoraussetzungen angepasst und die mathematischen Grundlagen ausgearbeitet wurden, gilt es zunächst, Anforderungen an die Architektur der Plattformsoftware festzulegen.

Flexibilität: Die Architektur soll von der genutzten Hardware und den damit verbundenen Kommunikationsmethoden abstrahieren. Dies wird durch eine Modularisierung erreicht und der damit verbundenen klaren Trennung der

einzelnen Aufgaben jedes Moduls. Daraus folgt, dass die Hardware-abhängigen Module austauschbar sind und die Hardware-unabhängigen nicht in ihrer Arbeit beeinflussen.

Skalierbarkeit: Die Architektur soll ohne Probleme um zusätzliche Funktionen erweitert werden können. Jede Erweiterung kann durch ein zusätzliches Modul realisiert werden, welches Funktionen bereits bestehender Module nutzen kann.

Benutzbarkeit: Die Architektur soll einfach zu handhaben sein durch kurze, leicht verständliche Befehle. Dabei soll es möglich sein, die Plattform manuell durch eine GUI anzusteuern.

Echtzeitfähigkeit: Im Falle einer Ansteuerung durch automatisch generierte Befehle technischer Geräte oder externer Software soll die Plattform Echtzeiteigenschaften zeigen. Das heißt, sie soll in angemessener Zeit eine Bewegung ausführen, sofern der Befehl dazu gegeben wurde. Bei kontinuierlicher Übertragung ist eine hohe Reaktivität zu gewährleisten, sodass ein Befehl rechtzeitig vor Erhalt des darauffolgenden ausgewertet wird.

3.4 Befehlsübertragung

Die Befehlsübertragung umfasst sowohl die Übermittlung von Nutzereingaben vom PC an den Mikrocontroller als auch die Übertragung von Signalen vom Mikrocontroller an die Motoren. Der AtMega16 stellt dabei mit der Uart sowie dem internen Timer alle benötigten Funktionalitäten bereit. Die im Folgenden vorgestellten Methoden sind an das vorhandene Modell angepasst und können je nach Änderung der Hardware ausgetauscht werden. Über die Uart wird mit der seriellen Schnittstelle des PC kommuniziert, wohingegen der Timer periodische Signale an die einzelnen Servomotoren schickt.

Das *Uartmodul* dient der Kommunikation mit der seriellen Schnittstelle des PCs. Die Uart-Funktionen des AtMega werden dabei gekapselt als Sende- und Empfangsmethoden zur Verfügung gestellt. Damit sich die Plattform bewegen kann,

müssen ihr entsprechende Befehle übermittelt werden. Anzumerken ist, dass nur die Übertragung eines einzelnen Zeichens auf einmal unterstützt wird. Folgende Zeichen werden derzeit unterstützt:

Zeichen	Bedeutung
r	Transformationsart auf Rotation setzen
t	Transformationsart auf Translation setzen
x	Transformationsachse auf X-Achse setzen
y	Transformationsachse auf Y-Achse setzen
z	Transformationsachse auf Z-Achse setzen
a	relative Trans.: Transformation aufaddieren Aufzeichnungsmodus: Speicherung in der Bewegungshistorie
s	absolute Trans.: Transformation setzen Aufzeichnungsmodus: Speicherung in der Bewegungshistorie
m	Plattform-Bewegung ausführen Aufzeichnungsmodus: startet neue Bewegung in der Bewegungshistorie
b	Bewegungsschritt zurück
w	Transformationen zurücksetzen
e	Plattform auf Startposition zurücksetzen
i	Aufzeichnungsmodus aktivieren
o	Aufzeichnungsmodus deaktivieren
c	Kreisbewegung einschalten/fortführen
v	Kreisbewegung stoppen
-	Vorzeichen des Transformationswerts negativ setzen
+	Vorzeichen des Transformationswerts positiv setzen
#	Setzt Transformationswert bestehend aus Betrag und Vorzeichen Erlaubt die Nutzung mehrziffriger Zahlenwerte
sonst	Rechnet Ascii-Zeichenwert in Integerwert um Hängt eine Ziffer (0-9) an die Betragsvariable an

Im Folgenden wird zwischen verschiedenen Arten der Ansteuerung unterschieden.
Beim Zurücksetzen der Plattform (e,w), Widerrufen einer Bewegung (b) oder der Kreisbahn (c,v) handelt es sich um eine indirekte Ansteuerung, da die Positi-

on hier nicht direkt vom Nutzer, sondern über einen Methodenaufruf bestimmt wird. Eine indirekte Steuerung ist sehr simpel und benötigt nur die Übertragung des Befehls zum Aufruf dieser Funktionen.
Die direkte Ansteuerung (*a*,*s*,*m*) ist komplexer, da hier direkt auf den Transformationsvektoren gearbeitet wird und Variablen manuell gesetzt werden müssen (*r* bis *z*, - bis *default*). Befehle zur direkten Ansteuerung können als zusammenhängender Block versendet werden. Nach Befehlen, die einen Methodenaufruf zur Folge haben (indirekte Ansteuerung sowie einfacher Bewegungsbefehl *n/m*), sollte eine Sendepause erfolgen, damit die entsprechende Methode in Ruhe abgearbeitet werden kann und keine Konflikte mit nachfolgenden Aufrufen der ISR auftreten. Die direkte Art der Ansteuerung geschieht über relative oder absolute Transformationskommandos. Bei ersteren bezieht sich die übergebene Transformation auf die jeweils letzte Position der Plattform, bei letzteren auf die Ruhelage. Eine valide Transformation ergibt sich aus einer Übergabe von Typ (t,r), Achse (x,y,z) und Transformationswert ($\mathbb{Z}$#), welcher eine beliebige ganze Zahl sein darf, an ein globales Transformationsobjekt. Eine Transformation darf erst hinzugefügt werden (a für relative, s für absolute Positionierung), sobald alle drei Felder dieses Objekts initialisiert wurden. Um die Plattform schließlich über *m/n* zu bewegen, muss mindestens eine Transformation hinzugefügt worden sein. Hierauf wird in den Kapiteln [Transformationen] und [Bewegungen] detaillierter eingegangen.
Das im Folgenden vorgestellte Beispiel gehört zur direkten Ansteuerung über unmittelbare Positionierungsbefehle. Eine Befehlsfolge einer Transformation mit anschließender Bewegung kann folgendermaßen aussehen: $rx-20\#am$, wobei $rx-20\#$ den Transformationsblock bezeichnet, der nacheinander ein Zeichen zur Art der Transformation (r für Rotation), zur betrachteten Achse (x für X-Achse) sowie einen Wert ($-20\#$ für $-20°$) an das aktuelle Transformationsobjekt übergibt. Der Werteblock beschreibt dabei entweder eine Länge in Millimetern oder einen Winkel in Grad, wobei $\#$ das Ende der übergebenen Zahl angibt. Prinzipiell ist die Reihenfolge der Befehle des Transformationsblocks austauschbar, jedoch sollte der Werteblock nicht aufgespalten werden. Die Befehlsfolge am wiederum fügt die zuvor spezifizierte Transformation dem entsprechenden Feld des internen Transformationsvektors hinzu (a) und bewegt die Plattform entsprechend dieser Struktur (m).

Nachdem die Grundlagen der Ansteuerung vom PC geklärt wurden, wird im Folgenden erläutert wie die Auswertungen dieser Befehle an die Aktorik weitergeleitet werden.
Das *Aktormodul* bildet die Schnittstelle zu den Motoren. Es enthält die Verwaltungsstrukturen des Timers, über die die Servoausrichtungen geändert werden können. Diese Ausrichtungen werden zuvor mit Hilfe der mathematischen Algorithmen der Inversen Kinematik berechnet.
Ein Timer bietet die Möglichkeit, in regelmäßigen Abständen Aktionen zu veranlassen. In diesem Fall wird er zum Generieren einer Pulsweitenmodulation (PWM) genutzt, die an den Signalinput der Servomotoren übertragen wird. Die Dauer eines Impulses, zumeist zwischen 1ms und 2ms, bestimmt den Auslenkwinkel.
Es wird ein gewisses Maß an Präzision benötigt, um diese Impulse in Timerticks zu repräsentieren. Die Wahl fiel daher auf den Timer2 des AtMega16, einen 8 Bit-Timer, der die Möglichkeit bietet, das Timerregister mit einem festen Wert zu vergleichen statt auf einen normalen Überlauf zu warten. Dieser Vergleichswert wird später auf einen Wert entsprechend dem Impuls an den Servomotor gesetzt. Standardmäßig inkrementiert der Timer das Timerregister um eins pro Takt und führt eine Unterbrechungsbehandlung aus, sobald das Register überläuft (Overflow), was in diesem Fall dem Erreichen des Vergleichswerts entspricht. Daraufhin wird das Register zurückgesetzt. Um die Zahl der Überläufe zu kontrollieren wird ein sogenannter Prescaler benötigt, welcher angibt, wie viele Takte gewartet wird bis der Timer inkrementiert wird. Die Nutzung des Timers erfordert neben dem Prescaler den Prozessortakt, welcher standardmäßig 8 Millionen Ticks/Sekunde beträgt. Da ein Servomotor einen Konfigurationsraum zwischen 1ms und 2ms besitzt, ist es ausreichend für den Timer diese Zeitspanne abbilden zu können. Ein 8-Bit-Timer zählt bis maximal 255, das heißt um eine möglichst hohe Winkelauflösung der Motoren zu erreichen, muss dem maximalen Zeitwert von 2ms ein Wert möglichst nahe am maximalen Tickwert 255 zugeordnet werden. Die Hälfte, eine Millisekunde, erhält folglich einen Wert nahe 127. Daraus ergibt

sich für den Prescaler folgende Berechnung:

$$\begin{aligned} HalfMaxTicks_{Millisekunde} &\geq F\_CPU/1000/PRESCALER \\ 127 &\geq 8.000/PRESCALER \\ PRESCALER &\geq 8.000/127 = 62,992 \end{aligned}$$

Um die maximale Präzision zu erreichen wird für den Prescaler mit 64 die kleinstmögliche Zweierpotenz gewählt. Detailliertere Beschreibungen der Funktionsweise des Timer2 finden sich im Datenblatt des AtMega16 [Atm12].

3.5 Befehlsauswertung

Zum Teilgebiet der Befehlsauswertung gehören alle Hardware-unabhängigen Tätigkeiten, die sich auf die Positionskontrolle entsprechend des Bewegungsmodells beziehen sowie die Weiterleitung der Positionsinformationen an die Motorenschnittstelle. Die Position der mobilen Plattform wird dabei wie aus Kapitel [Bewegungsmodell] bekannt als globale Transformation gespeichert, hierfür wird eine eigene Struktur, genannt *Transformationsvektor* bereitgestellt, welche sowohl Translationen als auch Rotationswinkel pro Achse enthält. Den Kern der Funktionalität der Plattform bilden zwei elementare Steuerkommandos, über welche zum einen die innere Logik, zum anderen die Schnittstelle mit der Aktorik angesprochen werden können. Abbildung 3.7 zeigt ein Aktivitätsdiagramm mit elementarer Funktionalität. Die grobe Architektur setzt eine Übermittlung der zuvor vorgestellten Steuerkommandos voraus, um die Bewegungen der Plattform beeinflussen zu können. Eine ausgeführte Bewegung besteht aus einer Ansammlung mehrerer Transformationen. Sowohl relative (zur letzten Plattformposition) als auch absolute Transformationen (zur Ruhelage) können über den Befehl *Transformation hinzufügen* an die Plattform übertragen werden. Der globale Transformationsvektor wird daraufhin aktualisiert. Der Befehl *Bewegen* sorgt für eine Anwendung der von diesem Vektor abhängigen Transformationsstrukturen ($\overrightarrow{t}$ und R) auf die Punktmenge. Auf Grundlage der neuen Eckpunkt-Koordinaten wird daraufhin der Winkel des zugehörigen Servomotors berechnet und ein entsprechendes Signal an die Motoren gesendet. Diese beiden Kommandos bilden die Grundlage jeder Bewegung, welche über direkte Ansteuerungsbefehle aus-

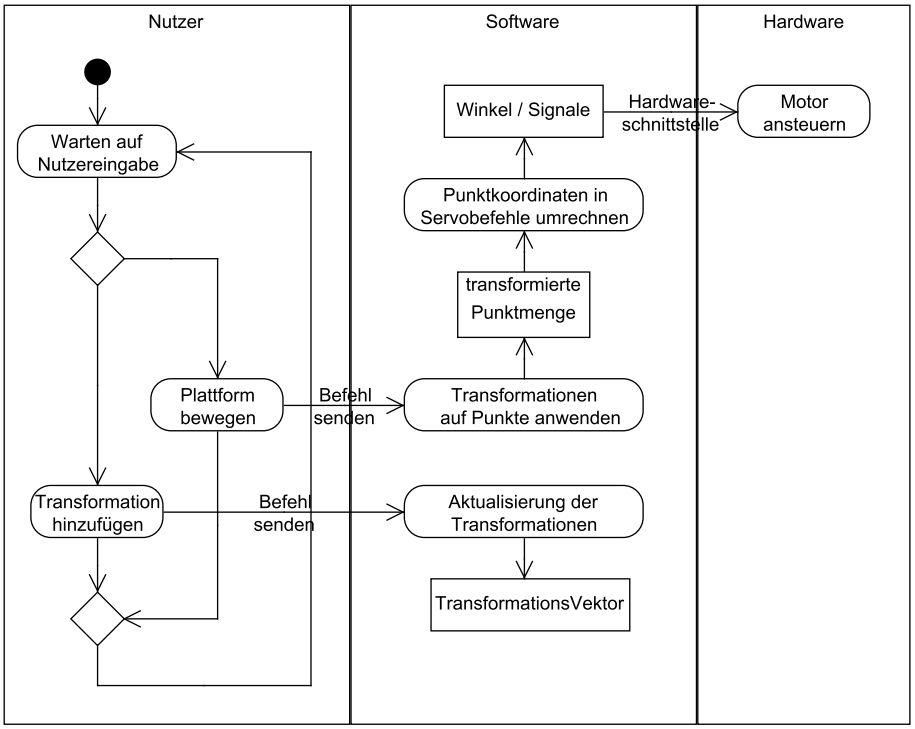


Abbildung 3.7: Der Algorithmus mit elementaren Funktionen zur Bewegung der Plattform

geführt wird.
Jede Form der indirekten Ansteuerung lässt sich auf elementarer Ebene über Manipulationen des Transformationsvektors sowie Bewegungsbefehle beschreiben. Damit besitzen komplexere Funktionen ein einheitliches Aufbauprinzip. Abbildung 3.8 zeigt eine Erweiterung des vorherigen Aktivitätsdiagramms. Hier wurden die komplexen Benutzerkommandos *Schritt zurück* und *Position zurücksetzen* hinzugefügt, welche es erlauben, die letzte Bewegung der Plattform zu widerrufen beziehungsweise die Plattform wieder in ihre Nullstellung zu bewegen. Diese werden im Folgenden kurz beschrieben.
Da entsprechend dem Bewegungsmodell nur die Gesamtheit aller Transformationen gespeichert wird, muss das Modell für den *Schritt zurück*-Befehl um eine Struktur erweitert werden, die einzelne Bewegungen speichert, das *Bewegungsobjekt* sowie den *Bewegungszeiger* auf das aktuelle Objekt. Aufeinander folgende Transformationen werden als Teilstruktur der aktuellen Bewegung zusammen-

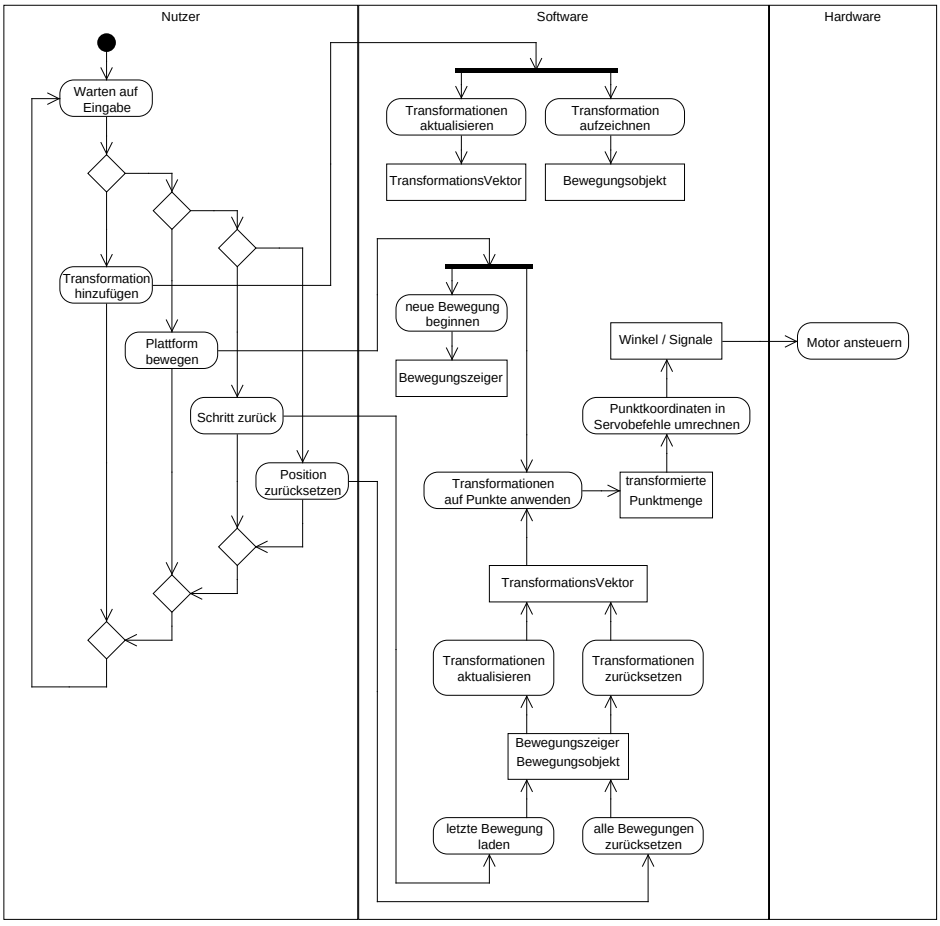


Abbildung 3.8: Das vormalige Aktivitätsdiagramm, um weitere (Bewegungs-)Funktionen ergänzt

gefasst. Dazu wird jeweils nach dem Hinzufügen einer Transformation das derzeitige Bewegungsobjekt aktualisiert. Wird die Plattform bewegt (*Plattform bewegen*), kann zusätzlich ein Abschluss der letzten und der Beginn einer neuen Bewegungs-Teilstruktur erfolgen. Möchte der Nutzer die Plattform auf die vormalige Position bewegen (*Schritt zurück*), so müssen zunächst alle Transformationen der letzten Bewegung (sofern vorhanden) geladen werden (abrufbar über den Zeiger auf das vormalige Objekt). Daraufhin werden die bereits bekannten Methoden genutzt, um Transformationen anzuwenden und die Plattform neu zu positionieren (ein einfacher Bewegungsbefehl). Der Befehl *Position zurücksetzen* geht ähnlich vor, nur dass hierbei vor der erneuten Positionierung sowohl Trans-

formationsvektor als auch Bewegungsobjekte auf ihren Initialzustand zurückgesetzt, mit anderen Worten gelöscht werden. Im Hinblick auf die Einheitlichkeit der Bewegungsroutinen reicht auch hier ein einfacher Bewegungsbefehl zur Neupositionierung.

3.6 Objektmodell

Das im Folgenden vorgestellte Objektmodell umfasst die verschiedenen Module, welche jeweils eine gekapselte Teilaufgabe der Softwarearchitektur enthalten. Die Module beschreiben ausschließlich öffentliche Variablen und Methoden, welche als Richtlinie für die Implementierung im nächsten Kapitel dienen. Sie entsprechen den Headern der späteren C-Module. Der Sourcecode wird zudem durch private Variablen und statische Hilfsmethoden erweitert, welche in diesem Objektmodell nicht enthalten sind. Die Programmierung umfasst des Weiteren Initialisierungsmethoden für einzelne Module, welche hier aus Übersichtsgründen weggelassen wurden.

Den Rahmen des Diagramms in Abbildung 3.9 bildet die Hardwareperipherie an beiden Enden des Algorithmus, zum einen der PC, zum anderen die einzelnen Motoren. Zu jeder Peripheriekomponente existiert eine Schnittstelle um Nachrichten zu empfangen und zu versenden. Die Kommunikation zum PC wird wie bereits erwähnt über die serielle Schnittstelle (Uart) realisiert. Dem Mikrocontroller können einzelne Befehlsfolgen gesendet werden, welche ausgewertet und später an das Aktormodul als Pulsweitenmodulation weitergereicht werden. Die Auswertung erfolgt wie im vorangegangenen Kapitel beschrieben in zwei Hardware-unabhängigen Kernmodulen, genannt Transformation und Bewegung. Zudem existiert ein Konstanten- bzw. Parametermodul, welches alle relevanten Konstanten und Felder des Plattform-Modells bereitstellt.

Relevant für das Objekt-Modell sind zunächst die Beziehungen der Module untereinander. Das Kommunikationsmodul, welches später die Uart-Funktionen enthält, soll in der Lage sein, die beiden Kern-Module direkt anzusprechen. Dem Nutzer wird so die Möglichkeit gegeben, sowohl Transformationen als auch Bewegungen auszuführen. Das Transformationsmodul dient dabei neben der Speicherung und Verwaltung der Transformationen auch der darauf aufbauenden

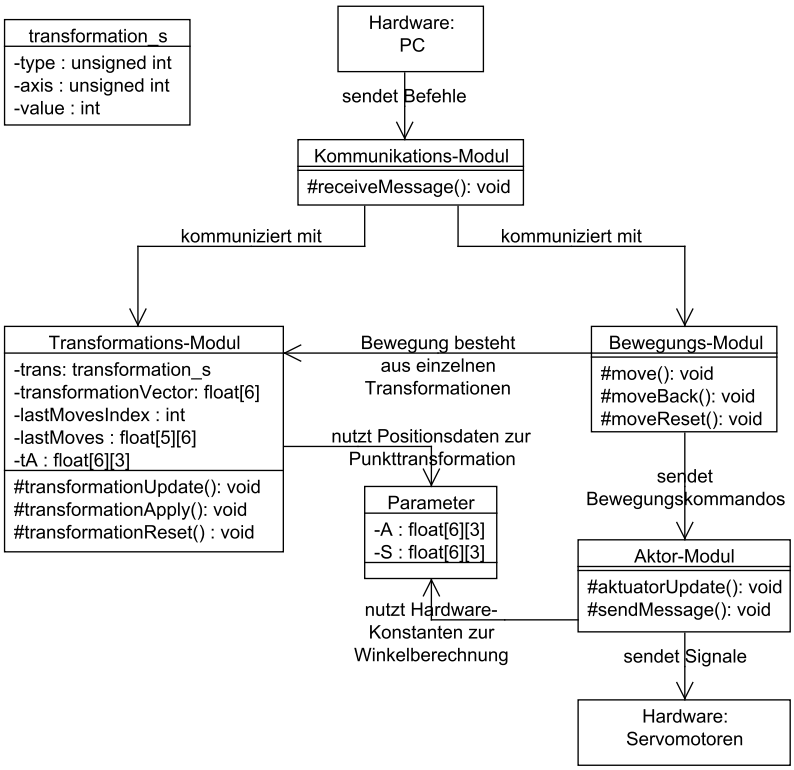


Abbildung 3.9: Ein UML-Diagramm zu Abbildung 3.8, welches alle beteiligten Module der Architektur beschreibt. Verschiedene Methoden werden in der Implementierung um Optionsparameter erweitert.

Berechnung der transformierten Punktkoordinaten. Das Aktormodul besitzt die Aufgabe, aus gegebenen Koordinaten ein Steuersignal für die Motoren zu generieren. Das Bewegungsmodul wiederum ist ein Mittelsmann und sorgt für die Zusammenarbeit und Koordinierung der beiden erstgenannten Module. Der Zugriff auf die Aktorik ist dabei exklusiv, was bedeutet, dass die Motoren nur gekapselt über das Bewegungsmodul angesprochen werden können. Aktorik sowie Transformationen nutzen zudem das Konstanten-Modul für ihre Algorithmen zur Umrechnung der Programmlogik auf die Physik der Plattform. Dieses Modul ist abhängig vom genutzten Plattform-Modell und kann wie das Aktormodul ausgetauscht werden, sofern sich an der Hardware etwas ändern sollte. Dies garantiert, dass die Kernfunktionalität auf jedem System zum Einsatz kommen kann, ohne aufwendige Änderungen am Code vornehmen zu müssen.

Kapitel 4

Programmierung

Der Großteil dieses Kapitels umfasst die Implementierung des zuvor ausgearbeiteten Objektmodells in C. Die einzelnen Methoden der Module enthalten dabei Funktionalitäten im Sinne der Ansteuerungslogik (siehe [Befehlsauswertung] und [Befehlsübertragung]) sowie der mathematischen Algorithmen (siehe [Inverse Kinematik]) der letzten Kapitel. Die graphische Ansteuerung erfolgt über eine Nutzeroberfläche, welche in Object-Pascal geschrieben wurde.

Zur Programmierung auf dem AtMega16 wurde die Entwicklungsumgebung *AVR Studio 4* und zum Flashen des Mikrocontrollers die Freeware *PonyProg 2000* genutzt. Die GUI wurde in der *Lazarus*-IDE entwickelt.

4.1 Mathematisches Hilfsmodul

Diese Sektion beschreibt das mathematische Hilfsmodul, welches nützliche trigonometrische Funktionen umfasst. Das Modul stellt eine Interpolationsmethode bereit, um die Sinus- und Cosinus-Funktion zu approximieren. Die Implementation eigener trigonometrischer Funktionen bietet gegenüber der Standard-C-Bibliothek $math.h$ erhebliche Geschwindigkeitsvorteile und wird dieser, wenn möglich, vorgezogen.

```
1  #ifndef _mathSP_h
2  #define _mathSP_h
3
4  #include <avr/io.h>
5  #include <stdlib.h>
6
7  typedef enum trigonometric
8  {
9          SINUS, COSINUS
10 } trigo_e;
11
12 float interpolate (trigo_e func, int angle);
13
14 #endif
```

Der Header *mathSP.h* enthält wiederverwendbare mathematische Konstrukte und Funktionen. Der Enumerations-Typ $trigo\_e$ enthält die beiden Identifizierer SINUS und COSINUS. Der Interpolations-Methode wird ein Identifizierer sowie ein ganzzahliger Winkel in Grad übergeben. Durch die Angabe des Funktionstyps wird die Methode generisch. Der Rückgabewert ist der auf drei Nachkommastellen genaue trigonometrische Funktionswert der angegebenen Funktion.

Winkel	0°	10°	20°	30°	40°	50°	60°	70°	80°	90°
Sinus	0	0.174	0.342	0.5	0.643	0.766	0.866	0.94	0.985	1
Cosinus	1	0.985	0.94	0.866	0.766	0.643	0.5	0.342	0.174	0

In der Source-Datei *matheSP.c* werden zu Beginn zwei konstante Felder $sinus$ und $cosinus$ deklariert, welche je zehn vorab berechnete Werte der Sinus- beziehungsweise Cosinus-Funktion enthalten. Die Werte reichen von $0°$ bis $90°$ in je $10°$-Schritten pro Feldeintrag.

```
1  float interpolate (trigo_e func, int angle)
2  {
3  float sign = 1.0;
4  float topvalue, botvalue;
5
6  if(angle < 0)
7  {
8          if(func==SINUS) sign = -1.0;
9          angle *= -1;
10 }
11
12 int bot = angle / 10;
13 int top = bot + 1;
14
15 if(func==SINUS)
16 {
17         topvalue = sinus[top];
18         botvalue = sinus[bot];
```

```
19  }
20
21  if(func==COSINUS)
22  {
23          topvalue = cosinus[top];
24          botvalue = cosinus[bot];
25  }
26
27  float diff = topvalue - botvalue;
28  float base = botvalue;
29  float t = (float)(angle % 10)/10;
30
31  return sign*(base + t * diff);
32  }
```

In der Implementierung der öffentlichen Interpolations-Methode werden zwei benachbarte Elemente von einem der beiden Felder ausgelesen und daraufhin wird zwischen diesen interpoliert. Zunächst erfolgt eine Überprüfung des Vorzeichens des übergebenen Winkels (Z.6-10). Im Intervall $[0°; 90°]$ beeinflusst dieses nur die Resultate der Sinusfunktion. Ein negativer Winkel bedeutet ein negatives Vorzeichen ($sign$). Der (positive) Eingabewinkel wird über eine Ganzzahldivision auf die nächste Zehnerstelle nach unten (bot) beziehungsweise oben (top) gerundet (Z.12-13). Diese Rundungen dienen als Indizes des Sinus- beziehungsweise Cosinus-Arrays (Z.15-25). Zwischen zwei benachbarten Werten eines der Arrays wird wie folgt interpoliert:

$$Val_{interpoliert} = Val_{bot} + t * (Val_{top} - Val_{bot})$$

wobei *t* den Interpolationsiterator bezeichnet. In diesem Fall erhält man ihn durch eine einfache Modulo 10 Operation samt float-Division über den Eingabe-Winkel (Z.29). Der Rückgabewert berücksichtigt zudem das zuvor bestimmte Vorzeichen (Z.31).

4.2 Peripherie - Hardware-Schnittstellen

Die Peripheriekomponenten befassen sich mit der Kommunikation mit der Hardware. Sie sind austauschbar und speziell an die Hardware anzupassen. Sie umfassen zum einen die Ansteuerung des Mikrocontrollers von einem technischen Gerät mit serieller Schnittstelle (hier der PC), zum anderen die Ansteuerung der Aktoren der Plattform (hier die Servomotoren). Zusätzlich existiert ein Modul,

welches die Ausmaße und Abmessungen der Bestandteile des Modells bereitstellt.

4.2.1 Hardwareparameter

```
1  #ifndef _parameter_h
2  #define _parameter_h
3
4  #define LEGLENGTH 325.0
5  #define MOTORRADIUS 40.0
6
7  extern const float A[6][3];
8  extern const float S[6][3];
9
10 #endif
```

Das Modul $parameter.h$ enthält zwei Plattform-spezifische Konstanten, welche Beinlänge ($LEGLENGTH$) und Drehradius der Motoren ($MOTORRADIUS$) definieren. Sämtliche genutzte Längenwerte werden in Millimetern angegeben. Des Weiteren werden zwei konstante externe Felder bereitgestellt, welche zum einen die Koordinaten der Eckpunkte der mobilen Plattform in Ruhelage (A) sowie der Drehpunkte der Motoren (S) enthalten, gespeichert in der Reihenfolge der Motorennummerierung aus [Inverse Kinematik]:

Punkt	A_1	A_2	A_3	A_4	A_5	A_6	S_1	S_2	S_3	S_4	S_5	S_6
x	-7.5	7.5	65	58	-58	-65	-46	46	80	35	-35	-80
y	0	0	0	0	0	0	-315	-315	-315	-315	-315	-315
z	-71	-71	29.5	42.5	42.5	29.5	-66.5	-66.5	-8	73.5	73.5	-8

4.2.2 Serielle Schnittstelle

```
1  #ifndef _uart_h
2  #define _uart_h
3
4  #include <stdio.h>
5  #include <avr/interrupt.h>
6  #include <avr/io.h>
7  #include "transformation.h"
8  #include "movement.h"
9
10 #define BAUD 9600UL
11 #define UBRR F_CPU/16/BAUD-0.5
12
13 void uartInit(unsigned int ubrr);
14
15 #endif
```

Zur Übertragung an den Computer werden zunächst folgende Konstanten benötigt: die (vorinitialisierte) interne Taktfrequenz der CPU F_CPU und die genutzte Baudrate $BAUD$. Der AtMega16 unterstützt verschiedene Taktraten, wobei diese standardmäßig auf 8 Millionen Takte pro Sekunde eingestellt ist. Diese Einstellung wurde beibehalten, da sie günstig für die Nutzung des Timers ist (siehe [Aktor-Schnittstelle]). Die Baudrate kann variiert werden und wurde auf 9600 festgelegt. Das Feld $UBRR$ ist von den beiden vorangegangenen Konstanten abhängig und bildet den Wert des Baudratenregisters der Uart. Der *uart.h*-Header stellt zudem in Erweiterung des Objektmodells die externe Funktion *uartInit* zur Verfügung, um die Uart-Register zu initialisieren.

```
1  static void uartSend(char data)
2  {
3  while(!( UCSRA & (1<<UDRE)));
4  UDR = data;
5  }
6
7  static unsigned char uartRecv()
8  {
9  while(!(UCSRA & (1<<RXC)));
10 return UDR;
11 }
12
13 void uartInit(unsigned int ubrr)
14 {
15 UBRRH = (unsigned char)(ubrr>>8);
16 UBRRL = (unsigned char)ubrr;
17 UCSRB = (1<<RXEN)|(1<<TXEN);
18 UCSRC = (1<<URSEL)|(3<<UCSZ0);
19 }
```

Die Sourcedatei enthält zusätzlich die privaten Hilfsmethoden *uartRecv* und *uartSend* zur Empfangen und Senden eines Zeichens. Die Empfangsmethode wartet den Empfang eines Zeichen ab und gibt dieses zurück, die Sendemethode setzt das Übertragungsregister mit dem entsprechenden Asciiwert des Zeichens und wartet bis es gesendet wurde. In der *uartInit*-Methode wird den Uart-Registern die genutzte Baudrate mitgeteilt (Z.15-16) sowie eine beidseitige Übertragung mit 8 Bit (Z.17-18) aktiviert. Detailliertere Informationen zur Uart finden sich im Datenblatt des AtMega16 [Atm12].

```
1  ISR(USART_RXC_vect)
2  {
3  char c = uartRecv();
4  switch(c)
5  {
6          case 'r':
7          trans.type = ROT;
8          break;
9          case 't':
10         trans.type = TRA;
11         break;
12         case 'x':
13         trans.axis = X;
14         break;
15         case 'y':
16         trans.axis = Y;
17         break;
18         case 'z':
19         trans.axis = Z;
20         break;
21         case 'a':
22         transformationUpdate(ADD, bRecord);
23         break;
24         case 's':
25         transformationUpdate(SET, bRecord);
26         break;
27         case 'm':
28         move(bRecord);
29         break;
30         case 'b':
31         moveBack();
32         break;
33         case 'w':
34         transformationReset();
35         break;
36         case 'e':
37         moveReset();
38         break;
39         case 'i':
40         bRecord = 1;
41         break;
42         case 'o':
43         bRecord = 0;
44         break;
45         case 'c':
46         bCircle = 1;
47         break;
48         case 'v':
49         bCircle = 0;
50         break;
51         case '-':
52         sign = -1;
53         break;
54         case '+':
55         sign = 1;
56         break;
57         case '#':
58         trans.value = (float)(sign*absValue);
59         sign = 1;
60         absValue = 0;
61         break;
62         default:
63         absValue = 10*absValue + (c-48);
64 }
65 return;
66 }
```

Die Unterbrechungsroutine (ISR) wird bei einem Uart-Interrupt aufgerufen und wartet den Empfang eines Zeichens ab. Dieses Zeichen vom Typ *char* wird in der Folge ausgewertet und eine entsprechende Methode ausgeführt beziehungsweise eine Variable gesetzt. Die ISR entspricht der *receiveMessage*-Methode des Objektmodells.
Die Variablen *bCircle* und *bRecord* legen fest, ob eine Kreisbewegung im Gange ist, beziehungsweise ob Bewegungen und Transformationen aufgezeichnet werden sollen. Das Feld *trans* des Transformationsmoduls umfasst Variablen für die Transformationsart, -achse sowie -wert. Da nur ein einziges Zeichen auf einmal empfangen werden kann, werden die beiden globalen Variablen $absValue$ und $sign$ genutzt, um den Zahlenbereich der Transformationswerte von [0;9] auf die ganzen Zahlen zu erweitern (Z.51-63).

4.2.3 Aktor-Schnittstelle

```
1  #ifndef _aktuator_h
2  #define _aktuator_h
3
4  #define F_CPU 8000000UL
5
6  #include <avr/io.h>
7  #include <avr/interrupt.h>
8  #include <math.h>
9  #include "parameter.h"
10 #include "mathSP.h"
11
12 #define S_PORT PORTA
13 #define S_DDR DDRA
14
15 #define PRESCALER 64
16 #define PRESCALER_BITS (1<<CS22)
17 #define ONE_MS (F_CPU/PRESCALER/1000)
18 #define MID ONE_MS/2
19 #define TICKS_PER_DEGREE (float)(ONE_MS/90.0)
20
21 #define NO_SERVOS 8
22
23 void timerInit();
24 void aktuatorUpdate();
25
26 #endif
```

Der *aktuator.h*-Header stellt zwei Methoden zur Verfügung, einerseits zur Initialisierung des Timers $timerInit$, andererseits zur Aktualisierung der Auslenkwinkel der Motoren $aktuatorUpdate$. Letztere ist eine Methode, welche aus den transformierten Punkten die entsprechende Position des Motors (repräsentiert durch Timerticks bis zum Überlauf) nach dem Winkel-Algorithmus aus Kapitel [Inverse

Kinematik] berechnet. Nach der Initialisierung beginnt der Timer seinen Endloszyklus.
Die vom Timer generierten Impulse werden an den Ausgabeport geschickt, definiert über die Felder S_PORT und S_DDR. Für den *PRESCALER* ergibt sich wie in Kapitel [Befehlsübertragung] ein Wert von 64, dessen Registerwert im Makro $PRESCALER\_BITS$ gespeichert wird. Der Wert für eine Millisekunde ONE_MS entspricht im Umkehrschluss 125 Ticks. MID bezeichnet die Anzahl der Ticks der relativen Mittelstellung der Motoren (0.5ms). Die absolute Mittelstellung (1.5ms) entspricht $ONE\_MS + MID$ in Timerticks.
Die Konstante $TICKS\_PER\_DEGREE$ wird zur Umrechnung der berechneten Winkel in Timerticks benötigt. Da der Servomotor einen Arbeitsraum von $90°$ bei einer Konfigurationsspanne von einer Millisekunde (1ms bis 2ms) in Timerticks besitzt, ergibt sich diese Konstante aus dem Quotienten dieser beiden Werte.
Die letzte Konstante $SERVOCOUNT$ setzt die maximale Anzahl der Servomotoren. Da 8 Ausgabepins auf Port A zur Verfügung stehen, wurde die Maximalzahl dementsprechend gewählt. Die Anzahl der angesteuerten Motoren ist allein von dieser Konstante abhängig. Dies ermöglicht es, zwei weitere Motorenanschlüsse in Reserve zu haben, ohne Änderungen am Code vornehmen zu müssen.

```
1 volatile uint8_t timerTicks[NR_SERVOS] =
2         {MID, MID, MID, MID, MID, MID, MID, MID};
3
4 uint8_t servoPosition[NR_SERVOS] =
5         {1<<PC0, 1<<PC1, 1<<PC2, 1<<PC3, 1<<PC4, 1<<PC5, 1<<PC6, 1<<PC7};
```

Das globale Feld $timerTicks$ enthält die aktuellen Servowinkel in Timerticks. Initialisiert wird das Feld mit der relativen Mittelstellung der Motoren. Statt eines Wertebereichs von [1;2] werden Werte aus [0;1] gespeichert, die absoluten Ticks erhält man durch Addition der Konstante ONE_MS. Aus dem Kapitel [Bewegungsmodell] ist bekannt, dass die Berechnung der Beinlängen zwar unabhängig, aber die Änderung dieser und damit der Winkel nicht unabhängig voneinander erfolgen kann (siehe [Beschreibung und Anwendungsgebiete]). Daraus folgt, dass alle Elemente dieses Feldes nur gemeinsam über die Methode $aktuatorUpdate$ aktualisiert werden können.
Die Source-Datei enthält ein weiteres privates Feld, welches die konstanten binären Positionswerte der Ausgabepins der einzelnen Servomotoren enthält (*servoPosition*).

```
1 void timerInit()
2 {
3 S_DDR = 0xFF;
4 S_PORT = 0x00;
5
6 OCR2 = ONE_MS + timerTicks[0];
7 TIMSK |= (1<<OCIE2);
8 TCCR2 = (1<<WGM21) | PRESCALER_BITS;
9 }
```

Die Methode $timerInit$ deklariert zunächst den S_PORT als Ausgang. Daraufhin wird das Vergleichsregister $OCR2$ auf den absoluten Tickwert des ersten Servomotors gesetzt (Z.6). Die folgenden Register dienen der Definition des Prescaler und Aktivierung des Compare Interrupt (Z.7-8, siehe Timerdokumentation [Tim12]).

```
1 ISR(TIMER2_COMP_vect)
2 {
3 static uint8_t i = 0;
4
5 S_PORT &= ~servoPosition[i];
6 if (++i >= NO_SERVOS) i = 0;
7 S_PORT |= servoPosition[i];
8
9 OCR2 = ONE_MS + timerTicks[i];
10 }
```

Die ISR wird bei einem Comparematch (Erreichen des Vergleichswerts) des Timers aufgerufen. Sie erfüllt die Aufgabe der *sendMessage*-Methode des Objektmodells. Die Ansteuerung der Motoren geschieht durch zyklische Verwaltung der $timerTicks$- und $servoPosition$-Felder (Z.6). Ein statischer Index (*i*) zeigt auf das jeweils aktuelle Element beider Listen. Mit jedem Aufruf der ISR wird dieser Index erhöht. Dabei wird der letzte Ausgabepin auf low (Ende des letzten Impulses, Z.5) und der aktuelle auf high (Start des nächsten, Z.7) gesetzt. Um die Länge des Impulses zu bestimmen, wird das Vergleichsregister auf die Timerticks des aktuellen Winkelwerts gesetzt (Z.9). Diese Prozedur findet permanent im Hintergrund statt. Die bei einer PWM übliche Pause von bis zu 20ms zwischen zwei nachfolgenden Impulsen desselben Motors wird durch die sequenzielle Ansteuerung simuliert. Bei 8 Runden pro Zyklus ergibt sich eine Pause von 8ms bis 16ms, welche ausreichend für einen stabilen Motorbetrieb ist.

```
1  void aktuatorUpdate()
2  {
3  globalToLocalCoordinates();
4
5  float P[3] = {0,0,0};
6  int angle;
7
8  for(int i = 0; i < 6; i++)
9  {
10         P[0] = tA[i][0];
11         P[1] = tA[i][1];
12         P[2] = tA[i][2];
13         angle = angleUpdate(P);
14
15         if(i%2 == 0)
16         timerTicks[i] = ONE_MS - angleToTicks(angle);
17
18         if(i%2 == 1)
19         timerTicks[i] = angleToTicks(angle);
20 }
21 }
```

Die Methode $aktuatorUpdate$ wendet den in [Inverse Kinematik] vorgestellten Algorithmus zur Winkelberechnung nacheinander auf alle transformierten Eckpunkte an. Voraussetzung für diesen Algorithmus war die Überführung der Eckpunkte vom Welt- in das entsprechende lokale Servo-Koordinatensystem jedes Motors (wie in [Inverse Kinematik] beschrieben). Entsprechend wird zunächst die Methode $globalToLocalCoordinates$ aufgerufen, welche im Verlauf dieses Kapitels näher erklärt wird. Danach wird über die transformierten Punkte tA, einem 6x3-Feld des Transformationsmoduls, iteriert und eine lokale Kopie des aktuell betrachteten Punktes an die im Folgenden ebenfalls noch zu erklärende Funktion $angleUpdate$ übergeben (Z.10-13). Diese Funktion enthält den eigentlichen Winkel-Algorithmus und liefert einen Winkel zwischen $0°$ und $90°$ zurück (im Gegensatz zu Abbildung 3.5 mit einem Intervall von $[-45°;45°]$). Die Aktualisierung des globalen Feldes $timerTicks$ unterscheidet zwischen den Ausrichtungen der Servomotoren (Z.15-19). Der Impuls des ersten Motors bei $0°$ beispielsweise entspräche 2ms, beim zweiten jedoch 1ms, bei $90°$ genau umgekehrt. Jeder zweite Motor besitzt wie in [Inverse Kinematik] beschrieben eine entgegengesetzte Ausrichtung und Drehrichtung, sodass die Hälfte der Winkelwerte nach Umrechnung in Timerticks ($angleToTicks$) angepasst werden müssen.

```
1  static void globalToLocalCoordinates()
2  {
3  float t1, t2;
4
5  float s[6] = { 0, 0, -0.866, -0.866,  0.866,  0.866 };
6  float c[6] = { 1, 1, -0.5  , -0.5  , -0.5  , -0.5   };
7
8  for(int i=0; i<6; i++)
9          for(int j=0; j<3; j++)
10                 tA[i][j] -= S[i][j];
11
12 for(int k=0; k<6; k++)
13 {
14 t1 = tA[k][0];
15 t2 = tA[k][2];
16 tA[k][0] = c[k]*t1 - s[k]*t2;
17 tA[k][2] = s[k]*t1 + c[k]*t2;
18 }
19
20 for(int k=0; k<6; k+=2)
21         tA[k][0] *= -1;
22 }
```

Die Methode $globalToLocalCoordinates$ nutzt die Punktmengen A und S des Parameter-Moduls zur Überführung der transformierten Eckpunkte vom globalen in die einzelnen lokalen Koordinatensysteme entsprechend Formel 3.9. Die neue lokale Punktmenge wird dabei in dem globalen 6x3-Feld *tA* des Transformationsmoduls gespeichert, sodass keine Rücktransformation von Nöten ist. Zunächst werden die Punkte in ihre jeweils neuen Ursprünge S_i verschoben (Z.8-10) und um die Y-Achse entsprechend Tabelle 3.2 rotiert (Z.12-18). Die beiden Felder s und c enthalten dazu vorberechnet die benötigten trigonometrischen Funktionswerte. Schlussendlich wird jeder zweite Punkt an der YZ-Ebene gespiegelt (Z.20-21).

```
1  static int angleUpdate (float* P)
2  {
3  float angle = 0;
4  float squareLength = P[0]*P[0]+P[1]*P[1]+P[2]*P[2];
5  float c =
6          (MOTORRADIUS*MOTORRADIUS + squareLength
7                  - LEGLENGTH*LEGLENGTH)/2/MOTORRADIUS;
8
9  angle = asin(c/(float)sqrt(P[0]*P[0]+P[1]*P[1]))
10                 - atan2(P[0],P[1]);
11
12 angle = angle*180/3.1415 + 45.5;
13
14 return (int)angle;
15 }
```

Die Methode $angleUpdate$ ist eine triviale Übernahme der Winkelberechnung aus Formel 3.11. Der einzige Unterschied ist die Verschiebung des Wertebereichs von $[-45°; 45°]$ nach $[0°; 90°]$.

```
1 static uint8_t angleToTicks (int angle)
2 {
3 return (uint8_t)(TICKS_PER_DEGREE * angle + 0.5);
4 }
```

Der aktualisierte Winkel muss daraufhin wie bereits beschrieben in Ticks umgerechnet werden. Die Methode $angleToTicks$ nutzt dazu die Hardwarekonstante $TICKS\_PER\_DEGREE$ und sorgt für ein korrektes Aufrunden des Integer-Rückgabewerts.

4.3 Softwarekern - Interne Logik

Die Kernmodule bilden die Hardware-unabhängige Basis der Plattformansteuerung und müssen im Gegensatz zur Peripherie nicht an die Hardware angepasst werden. Jede modulare Erweiterung der Bewegungsformen der Plattform baut auf die im Folgenden vorgestellten Komponenten auf. Diese beinhalten die interne Transformationslogik sowie die Übertragung einer Bewegung auf das Plattformmodell. Die Methoden der Kernmodule arbeiten dabei fast ausschließlich auf globalen Strukturen und nehmen folglich bis auf wenige Ausnahmen keine Parameter entgegen.

4.3.1 Transformationen

Das Transformationsmodul baut stark auf dem in den Grundlagen erläuterten Bewegungsmodell auf. Eine Bewegung besteht aus einer Akkumulation der sechs Basis-Transformationen, die durch dieses Modul in einem Array (Transformationsvektor) verwaltet werden, wobei die ersten drei Elemente den Translationen, die zweiten den Rotationswinkeln gegen den Uhrzeigersinn jeweils einer der drei Koordinatenachsen entsprechen. Das Hinzufügen einer Transformation durch den Nutzer ist leichtgewichtig, da zunächst alle übergebenen Werte auf das entsprechende Feldelement aufaddiert werden. Ein Aufruf zur Berechnung

der transformierten Koordinaten der Eckpunkte erfolgt erst im Bewegungsmodul nach Übergabe eines Bewegungsbefehls. Diese Art der Verwaltung ist günstig bei Nutzung aller sechs Freiheitsgrade, da nur eine einzige zentrale Berechnung der Punkte erfolgt.

```
1  #ifndef _transformation_h
2  #define _transformation_h
3
4  #include <avr/io.h>
5  #include "parameter.h"
6  #include "mathSP.h"
7
8  #define ADD 1
9  #define SET 2
10
11 #define X 0
12 #define Y 1
13 #define Z 2
14 #define TRA 0
15 #define ROT 3
16
17 #define RECORDEDMOVES 5
18
19 typedef struct transformation
20 {
21         unsigned int type;
22         unsigned int axis;
23         int value;
24 } transformation_s;
25
26 extern transformation_s trans;
27 extern volatile float tA[6][3];
28 extern volatile float transformationVector[6];
29
30 extern volatile int lastMovesIndex;
31 extern volatile int lastMoves[RECORDEDMOVES+1][6];
32
33 void transformationInit();
34 void transformationUpdate(int mode, int rec);
35 void transformationApply();
36 void transformationReset();
37
38 #endif
```

Der Header $transformation.h$ definiert zu Beginn zwei Konstanten, die verschiedene Modi der folgenden Methoden beschreiben. *ADD* und *SET* beziehen sich auf relative beziehungsweise absolute Transformationen. Die folgenden Konstanten wurden zum vereinfachten Zugriff auf den Transformationsvektor definiert. *X*, *Y* und *Z* entsprechen den Koordinatenachsen und *TRA* beziehungsweise *ROT* dem Transformationstyp. Die Addition einer der Typkonstanten auf einen Feldindex gibt an auf welche Transformationsart, die Addition einer Achsenkonstante auf welche Achse zugegriffen werden soll. Eine Addition zweier dieser Konstanten ergibt folglich den Index des anzusprechenden Elements. Der Index $TRA+X$

entspräche beispielsweise dem Index des Translationswerts in Richtung der X-Achse. *RECORDEDMOVES* wiederum enthält die Größe der Bewegungshistorie, welche im Laufe dieses Kapitels näher erläutert wird. Eine einzelne Transformation wird über den hier definierten strukturierten Typ *transformation_s*, welcher Typ, Achse und Wert (Millimeter beziehungsweise Grad) enthält, eindeutig beschrieben.
Das Modul verwaltet zudem eine Vielzahl an globalen Feldern und Variablen, welche durch Funktionsaufrufe der Kernmodule verändert werden können. Die aktuelle Transformation wird im Feld $trans$ gespeichert, im folgenden auch als Transformationsobjekt bezeichnet, das über Nutzereingaben (siehe [Serielle Schnittstelle]) gesetzt wird. Das Feld tA ist wie bereits angedeutet im Kapitel [Aktor-Schnittstelle] für die transformierten Punkte reserviert. Die Buchführung über alle Transformationen der mobilen Plattform geschieht im globalen Feld *transformationVector*. Da dieses Array jedoch nur die Gesamtheit aller Transformationen relativ zur Ruhelage enthält, existiert ein zusätzliches Feld $lastMoves$ zur Aufzeichnung der letzten ausgeführten Bewegungen der Plattform. Eine Bewegung besteht dabei aus allen Unterschieden der Transformationsvektoren zwischen zwei Positionierungen. Dieses Feld wird als zyklische Liste verwaltet. Zur Navigation dient ein Index, der auf das nächste Element der Liste zeigt. Das Feld enthält dabei ein Element mehr als die definierte maximale Anzahl der zu speichernden Bewegungen. Es wird stets auf dem Element des Feldes gearbeitet, welches die aktuelle auszuführende Bewegung enthält, dem Arbeitselement, wohingegen die restlichen Elemente die bereits ausgeführten Bewegungen beschreiben, die Historie.
Der Header stellt eine Vielzahl an Methoden zur Transformationskontrolle bereit. $TransformationInit$ dient der externen Initialisierung der Felder des Moduls. *TransformationUpdate* wertet die in der Übergangsvariablen gespeicherte Transformation $trans$ aus und addiert diese auf beziehungsweise setzt das entsprechende Element des Transformationsvektors je nach Modus. Ist der zweite Parameter entsprechend gesetzt, werden die übergebenen Transformationen zudem aufgezeichnet und dem aktuellen Bewegungsobjekt hinzugefügt. *TransformationApply* nutzt die Transformationsvektoren, um schlussendlich die Punkte der Ruhelage in ihre neuen Koordinaten zu überführen. Zur Rücksetzung des Transformationsvektors steht die Methode $transformationReset$ bereit.

```
1  void transformationInit()
2  {
3  for(int i = 0; i<RECORDEDMOVES+1; i++)
4          for(int j = 0; j<6; j++)
5                  lastMoves[i][j] = 0;
6
7  for(int j = 0; j<6; j++)
8  {
9          for(int i = 0; i<3; i++) tA[j][i] = 0.0;
10         transformationVector[j] = 0.0;
11 }
12 }
```

Hier werden essentiell nur die globalen Felder null-initialisiert. Ein Aufruf dieser Methode ist notwendig, um die transformierten Eckpunkte berechnen, Transformationen hinzufügen und eine Historie der Bewegungen erstellen zu können.

```
1  void transformationUpdate(int mode, int rec)
2  {
3  uint8_t i = trans.type + trans.axis;
4
5  if(mode == ADD)
6          transformationVector[i] += trans.value;
7  if(mode == SET)
8          transformationVector[i] = trans.value;
9
10 if (rec) transformationRecord(mode);
11 }
```

Die Funktion *transformationUpdate* bietet einfache Verwaltungsstrukturen zur Auswertung der gerade anzuwendenden Transformation ($trans$) und nimmt zwei Parameter entgegen. Der Parameter $mode$ bestimmt, ob eine Transformation hinzugefügt (*ADD*, relativ) oder gesetzt (*SET*, absolut) wird. Der Index des betreffenden Elements ergibt sich aus der Addition von Typ- und Achsenvariable, welche je eine der zuvor definierten Konstanten enthalten (Z.3). Der Parameter *rec* bestimmt, ob eine Aufzeichnung der Transformation im aktuellen Bewegungsobjekt erfolgen soll. Der aufzurufenden Methode *transformationRecord* wird dabei der Modus übergeben.

```
1  void transformationRecord(int mode)
2  {
3  uint8_t i = trans.type + trans.axis;
4
5  if (mode==ADD)
6          lastMoves[lastMovesIndex][i] += trans.value;
7  if (mode==SET)
8          lastMoves[lastMovesIndex][i] = trans.value;
9  }
```

In der Methode $transformationRecord$ wird wiederum anhand des Modus zwischen relativen und absoluten Transformationen unterschieden. Im Folgenden wird das Feld $lastMoves$ um eine Transformation erweitert innerhalb des Unterelements, auf das $lastMovesIndex$ zeigt, der aktuellen Bewegung. Die Art der Übergabe der Transformation ist dabei äquivalent zu der aus *transformationUpdate*.

```
1  void transformationReset()
2  {
3  for(int i = 0; i<6; i++)
4  {
5          transformationVector[i] = 0.0;
6  }
7  }
```

Eine Möglichkeit zur Zurücksetzung aller bisherigen Transformationen bietet die Methode $transformationReset$, in welcher der Inhalt des Transformationsvektors gelöscht wird.

```
1  void transformationApply()
2  {
3  for(int i = 0; i<6; i++)
4          for(int j = 0; j<3; j++)
5                  tA[i][j] = 0.0;
6
7  rotationsApply();
8
9  for(int i = 0; i<6; i++)
10         for(int j = 0; j<3; j++)
11                 tA[i][j] -= transformationVector[j];
12 }
```

Endgültig auf die Punktmenge A angewandt werden die gespeicherten Transformationen in der Methode $transformationApply$ (nach 3.1). Zu Beginn der Methode wird das für die transformierten Punkte reservierte Feld tA als Null-Vektor initialisiert (Z.3-5). Nacheinander werden auf die Punktmenge Rotationen ($rotationsApply$) sowie Translationen (Z.9-11, vergleiche 3.2) angewandt.

```
1  static void rotationsApply()
2  {
3  float s,c;
4
5  float r1[3][3] = { {1,0,0}, {0,1,0}, {0,0,1} };
6  float r2[3][3] = { {1,0,0}, {0,1,0}, {0,0,1} };
7  float r3[3][3] = { {1,0,0}, {0,1,0}, {0,0,1} };
8  float r4[3][3] = { {0,0,0}, {0,0,0}, {0,0,0} };
9  float r5[3][3] = { {0,0,0}, {0,0,0}, {0,0,0} };
10
11 s = interpolate(SINUS, transformationVector[3]);
12 c = interpolate(COSINUS, transformationVector[3]);
13
14 r1[0][0] = 1.0;
15 r1[1][1] = c;
16 r1[1][2] = s;
17 r1[2][1] = -s;
18 r1[2][2] = c;
19
20 s = interpolate(SINUS, transformationVector[4]);
21 c = interpolate(COSINUS, transformationVector[4]);
22
23 r2[0][0] = c;
24 r2[0][2] = -s;
25 r2[1][1] = 1.0;
26 r2[2][0] = s;
27 r2[2][2] = c;
28
29 s = interpolate(SINUS, transformationVector[5]);
30 c = interpolate(COSINUS, transformationVector[5]);
31
32 r3[0][0] = c;
33 r3[0][1] = s;
34 r3[1][0] = -s;
35 r3[1][1] = c;
36 r3[2][2] = 1.0;
37
38 for(int i = 0; i<3; i++)
39         for(int j = 0; j<3; j++)
40                 for(int k = 0; k<3; k++)
41                         r4[i][j] += r2[i][k]*r1[k][j];
42
43 for(int i = 0; i<3; i++)
44         for(int j = 0; j<3; j++)
45                 for(int k = 0; k<3; k++)
46                         r5[i][j] += r3[i][k]*r4[k][j];
47
48 for(int i = 0; i<3; i++)
49         for(int j = 0; j<6; j++)
50                 for(int k = 0; k<3; k++)
51                         tA[j][i] += r5[i][k]*A[j][k];
52 }
```

In der Hilfsmethode $rotationsApply$ werden die drei Rotationsmatrizen der verschiedenen Achsen entsprechend der gespeicherten Rotationswinkel des Transformationsvektors initialisiert (Z.5-36, vergleiche 3.3). Nacheinander werden die Matrizen von links an die Ergebnismenge beziehungsweise die Ruhepunkte multipliziert und die resultierende Punktmenge schließlich im Feld tA gespeichert (Z.38-51).

4.3.2 Bewegungen

Das Bewegungsmodul bildet das Bindeglied zwischen der Anwendung von Transformationen und deren Ausführung. Es übermittelt die Informationen der akkumulierten Rotationen und Translationen an die Aktorik, in diesem Fall die Servomotoren.

```
1  #ifndef _movement_h
2  #define _movement_h
3
4  #define F_CPU 8000000UL
5
6  #include <avr/io.h>
7  #include <util/delay.h>
8  #include "aktuator.h"
9  #include "transformation.h"
10 #include "mathSP.h"
11
12 extern volatile uint8_t bCircle;
13
14 void move(int rec);
15 void moveBack();
16 void circle();
17 void moveReset();
18
19 #endif
```

Der Header *movement.h* stellt Methoden bereit, die entweder eine einzelne Bewegung oder eine Bewegungsfolge steuern. Eine einzelne Bewegung entsprechend der Transformationsvektoren wird über $move$ ausgeführt. Die Methode $moveBack$ ermöglicht es, einen einzelnen Bewegungsschritt zu widerrufen. Sie bedient sich dabei des Aufzeichnungsfeldes des Transformation-Moduls. Die Plattform kann durch den Aufruf von $moveReset$ in ihre Startposition zurückbewegt werden. Die Methode $circle$ ist eine Beispielmethode einer komplexeren Bewegungsform, welche die Plattform in eine Kreisbewegung versetzen kann, gesteuert über das globale Flag $bCircle$, einer boolschen Variablen zum Beginnen oder Stoppen der Kreisbewegung.

```
1 void move(int rec)
2 {
3 if (rec) moveRecord();
4 transformationApply();
5 aktuatorUpdate();
6 }
```

Eine Bewegungsausführung beinhaltet stets eine Neuberechnung der Eckpunktkoordinaten ($transformationApply$) sowie eine Aktualisierung der Motorenaus-

richtung ($aktuatorUpdate$). Über den Parameter *rec* ($= 1$) kann festgelegt werden, ob ein neues Bewegungsobjekt initialisiert werden soll. Hierzu wird wenn nötig die Methode *moveRecord* aufgerufen. Der Übergabewert für eine einfache Bewegung ohne Aufzeichnung ist dementsprechend 0.

```
1  void moveRecord()
2  {
3  lastMovesIndex++;
4  if(lastMovesIndex > RECORDEDMOVES) lastMovesIndex = 0;
5  for(int i=0; i<6; i++) lastMoves[lastMovesIndex][i] = 0;
6  }
```

Die Ausführung der $moveRecord$-Funktion schließt die letzte Bewegung ab und startet eine neue. Dazu wird der Zeiger auf das Aufzeichnungsfeld inkrementiert und der Inhalt des neu ausgewählten Elements gelöscht, um so eine neue Bewegung zu initialisieren. Nach der Neuinitialisierung besteht das Feld *lastMoves* aus den maximal 5 gespeicherten letzten Bewegungen sowie dem aktuellen leeren Element zur Aufzeichnung folgender Transformationen.

```
1  void moveBack()
2  {
3  lastMovesIndex--;
4  if(lastMovesIndex < 0) lastMovesIndex = RECORDEDMOVES;
5
6  for(int i=0; i<6; i++)
7  {
8          transformationVector[i] -= lastMoves[lastMovesIndex][i];
9          lastMoves[lastMovesIndex][i] = 0.0;
10 }
11
12 move(0);
13 }
```

Die Methode $moveBack$ bewegt den Zeiger auf das letzte aktualisierte Element des Aufzeichnungsfelds (Z.3-4). Die darin enthaltenen Transformationen werden direkt auf die entsprechenden Vektoren angewandt (Z.8) und der aktuelle Feldeintrag gelöscht (Z.9), da die Bewegung widerrufen wurde. Schließlich wird ein nicht aufzeichnender Bewegungsbefehl gegeben.

```
1  void moveReset()
2  {
3  transformationReset();
4  lastmovesReset();
5  methodReset();
6  move(0);
7  }
```

Die $moveReset$-Methode setzt nacheinander Transformationen (Z.3), die Bewegungshistorie (Z.4) sowie Methoden (Z.5) auf ihren Anfangszustand zurück und aktualisiert die Motorenposition entsprechend.

```
1  void lastmovesReset()
2  {
3  lastMovesIndex = 0;
4  for(int i=0; i<RECORDEDMOVES+1; i++)
5          for(int j=0; j<6; j++)
6                  lastMoves[i][j] = 0.0;
7  }
```

In *lastmovesReset* werden sowohl Historie als auch der Zeiger zurückgesetzt.

```
1  void methodReset()
2  {
3  bReset = 1;
4  circle();
5  bReset = 0;
6  }
```

In der Funktion $methodReset$ werden diejenigen Methoden aufgerufen, welche statische Variablen enthalten. Diese werden durch einen Aufruf nach Setzen des Flags $bReset$ reinitialisiert. Alternativ können hier auch globale Variablen geändert werden.

```
1  void circle()
2  {
3  static float angle = 0.0;
4  static int i = 120;
5
6  if(bReset)
7  {
8          angle = 0.0;
9          i = 120;
10         return;
11 }
12
13 lastmovesReset();
14
15 float rad = 40;
16 float steps = 120.0;
17 float inc = 6.2832/steps;
18
19 while(i--)
20 {
21         transformationVector[0] = rad*sin(angle);
22         transformationVector[2] = rad*cos(angle);
23
24         move(0);
25
26         _delay_ms(15);
27
```

```
28          angle = (steps-i) * inc;
29
30          if(bCircle==0) break;
31      }
32
33      if(i==0) i = (int)steps;
34      }
```

Die Methode $circle$ versetzt die Plattform in eine unterbrech- und wieder aufnehmbare Kreisbewegung im Uhrzeigersinn. Diese ist so ausgelegt, dass sie einen festen Radius besitzt und ihre letzte Position vor der Unterbrechung wieder anfahren kann. Dadurch haben folgende Bewegungen keine Auswirkung auf die Kreisbahn.
Sowohl der Winkel ($angle$) der Auslenkung als auch der Index (i) der Teilbewegungen werden als statische Variablen verwaltet (Z.3-4). Wurde zuvor die Methode über $methodReset$ in ihren Ursprungszustand zurückversetzt (Flag $bReset$ gesetzt), so werden die statischen Variablen reinitialisiert und die Methode wird umgehend wieder verlassen (Z.6-11).
Vor Beginn der Kreisbewegung wird die Historie der vorherigen Bewegungen gelöscht ($lastmovesReset$), da diese nach Beginn der Kreisbahn keinen Wert mehr besitzen. Die Kreisbewegung setzt sich aus 120 gleichförmigen linearen Einzelbewegungen zusammen ($steps$), welche durch trigonometrische Funktionen berechnet werden. Der Radius der Kreisbewegung (rad) beträgt dabei 40 Millimeter. In einer Schleife werden im Folgenden nacheinander die Positionen der Plattform nach jeder Einzelbewegung berechnet, abhängig vom aktuellen Winkel der Ausrichtung. Genutzt wurden dazu die Funktionen der *math.h*, welche bessere Resultate als die der *mathSP.h* boten. Entsprechend der berechneten Werte werden die horizontalen Translationen im Transformationsvektor gesetzt (Z.21-22) und ein Bewegungsbefehl gegeben (Z.24). Zur Berechnung des aktuellen Auslenkungswinkels ist schließlich ein Winkelinkrement inc notwendig, welches mit der Größe des Schleifenindex skaliert (Z.28). Dieser wird dabei zyklisch verwaltet, damit die Plattform nicht nach einem Umlauf den Betrieb aufgibt (Z.33). Die Geschwindigkeit der Kreisbewegung kann durch die Delay-Funktion des AVR beeinflusst werden (Z.26). Sobald das Flag $bCircle$ nicht mehr gesetzt ist wird die Schleife abgebrochen und die Methode verlassen (Z.30).

4.4 Hauptprogramm

```
1  #define F_CPU 8000000UL
2
3  #include <avr/io.h>
4  #include <util/delay.h>
5  #include <avr/interrupt.h>
6  #include "aktuator.h"
7  #include "transformation.h"
8  #include "movement.h"
9  #include "uart.h"
10 #include "mathSP.h"
11
12 int main(void)
13 {
14         transformationInit();
15         timerInit();
16         uartInit(UBRR);
17
18         sei();
19         _delay_ms( 1000 );
20         UCSRB |= (1<<RXCIE);
21
22     while(1)
23     {
24                 if(bCircle) circle();
25     }
26 }
```

Im Hauptprogramm erfolgt zunächst ein Initialisierungsaufruf der verschiedenen Module (Z.14-16). Zusätzlich werden Interrupts aktiviert (Z.18) und nach einer Wartezeit von einer Sekunde durch Setzen eines bestimmtes Flags dafür gesorgt, dass eine Unterbrechung bei Empfang eines Zeichens ausgelöst wird (Z.20). In der while-Schleife folgt die Abfrage von verschiedenen Flags, woraufhin komplexe Sequenzen von Bewegungen aufgerufen werden, welche ansonsten die ISR der seriellen Schnittstelle blockieren würden. Bislang wird ausschließlich die Kreisbewegung auf diese Weise angesteuert (Z.24). Die Liste der Bewegungen kann jedoch um weitere Flags und Aufrufe erweitert werden. Da der gesamte Lebenszyklus des Programms auf Unterbrechungsbehandlungen aufbaut, ist hier nichts weiter zu erledigen und die Plattform kann durch Anschluss an den PC angesteuert werden. Das folgende Kapitel befasst sich mit der Vereinfachung der manuellen Ansteuerung über eine graphische Nutzeroberfläche.

4.5 Graphische Benutzeroberfläche

Die graphische Nutzeroberfläche nutzt die Lazarus-Entwicklungsumgebung, welche die Programmierung in Object-Pascal ermöglicht. Das Ziel dieses Programms ist eine intuitive Ansteuerung der Plattform über Schieberegler. Statt relativer Positionierungskommandos kann dadurch sofort die absolute Position eingestellt werden. Die Option *Connect* richtet die Verbindung zur seriellen Schnittstelle

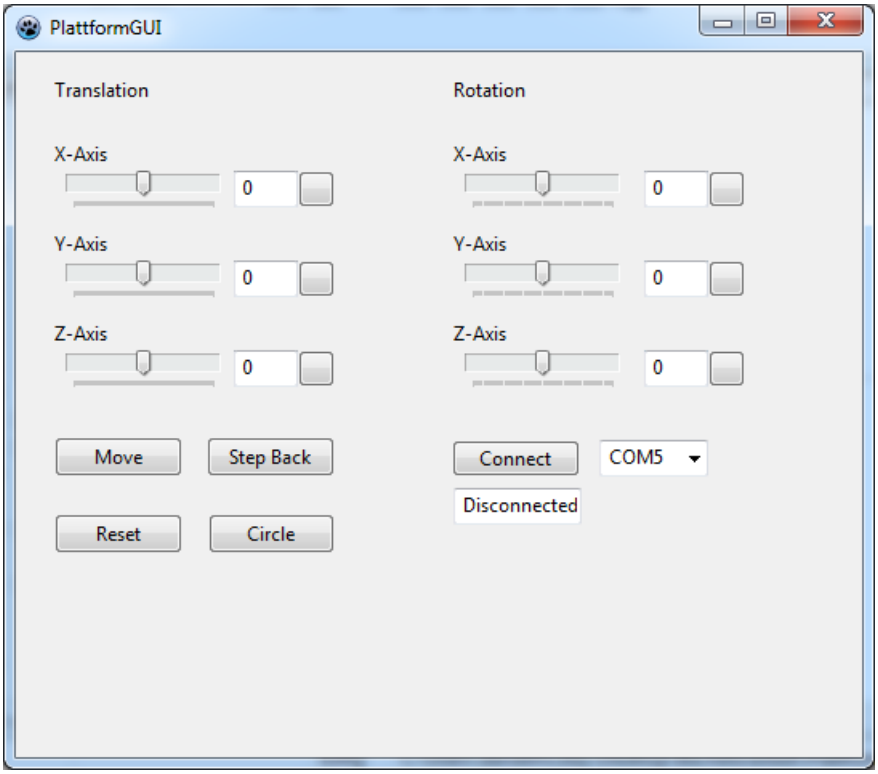


Abbildung 4.1: Die graphische Nutzeroberfläche stellt die grundlegenden Funktionen der Plattformansteuerung bereit. Statt einer relativen realisiert dieser Ansatz eine absolute Positionierung.

ein, wobei im nebenstehenden Feld der gewünschte Comport ausgewählt werden kann. Um diese wieder zu trennen, muss der Knopf ein zweites Mal betätigt werden. Nach dem Eingehen der Verbindung benötigt die Plattform wenige Sekunden zum Initialisieren, andernfalls kann es bei der Kommunikation zu Initialisierungsfehlern kommen und Befehle werden nicht korrekt empfangen und verwertet.

Je drei Schieberegler für Translationen und Rotationen stehen für die absolute

Positionskontrolle zur Verfügung. Dabei ist es dem Anwender überlassen, den Regler zu nutzen oder manuell eine Position in das nebenstehende Textfeld einzugeben. Beide Elemente sind intern verknüpft und ändern sich entsprechend des anderen. Eine Betätigung des Textfeldbuttons schickt einen Transformationsbefehl entsprechend des ausgewählten Reglers sowie des nebenstehenden Wertes an die Plattform. Über $Move$ wird ein Bewegungsbefehl gesendet, *Step Back* widerruft die letzte Bewegung (bis zu 5 mal) und $Reset$ setzt die Plattform in Nullstellung zurück. Der Togglebutton $Circle$ ermöglicht die (Wieder-)Aufnahme beziehungsweise das Pausieren einer Kreisbewegung mit festem Radius.

Die Kommunikation erfolgt wie bereits in den vorangegangenen Kapiteln erwähnt über die serielle Schnittstelle, deren Ansteuerung durch das Synaser-Paket der Lazarusumgebung erfolgt. Da der Code zu einem großen Teil aus Sendemethoden besteht, welche äquivalent im Aufbau sind, werden hier nur zwei der wichtigsten Prozeduren, die Ansteuerung der seriellen Schnittstelle sowie die Positionskontrolle, näher erläutert.

```
1  procedure TPlattformGUI.connectButtonChange(Sender: TObject);
2  var
3    portnr : string;
4    parity : char;
5    baud,bits,stop : integer;
6    softflow,hardflow : boolean;
7    data : char;
8  begin
9    if(connectButton.Checked = True) then
10   begin
11        connectButton.Caption := 'Disconnect';
12        ser := TBlockSerial.Create;
13        portnr := ComboBox1.Text;
14        baud := 9600;
15        bits := 8;
16        parity := 'N';
17        stop := SB1;
18        softflow := false;
19        hardflow := false;
20
21        ser.Connect(portnr);
22        sleep(250);
23        ser.Config(baud,bits,parity,stop,softflow,hardflow);
24        sleep(250);
25
26        if ser.LastError = 0 then
27        begin
28             Edit1.Text := 'Connected';
29             connected := True;
30             data := 'i';
31             ser.SendString(data);
32        end
33        else Edit1.Text := 'Error';
34   end
```

```
35
36    else
37    begin
38          connectButton.Caption := 'Connect';
39          ser.Free;
40          sleep(1000);
41          Edit1.Text := 'Disconnected';
42          connected := False;
43    end;
44
45  end;
```

Diese Methode öffnet die serielle Schnittstelle und speichert die Referenz darauf in einer globalen Variablen des Typs TBlockSerial (*ser*). Sobald der Togglebutton *Connect* gedrückt wird, wird die Variable initialisiert (Z.12) und die zur Kommunikation mit der Schnittstelle nötigen Parameter gesetzt (Z.13-19) und hinzugefügt (Z.21-24). Dies umfasst zum einen die Portnummer (COM-Port), der über die Auswahlbox ausgelesen werden kann, zum anderen die Baudrate und andere wichtige Bits. War der Verbindungsaufbau erfolgreich, wird das zugehörige Textfeld aktualisiert und der Verbindungsstatus (*connect*) auf True gesetzt, zudem wird der Befehl zur Aktivierung der Aufzeichnung von Bewegungen versendet (Z.26-33). Wird der Togglebutton ein weiteres Mal genutzt, wird die Schnittstelle wieder freigegeben und der Verbindungsstatus auf False zurückgesetzt (Z.37-42).

```
1   procedure TPlattformGUI.EditButton1Click(Sender: TObject);
2   var
3      data : String;
4      val : integer;
5   begin
6     if connected then
7     begin
8      transXBar.Position := StrToInt(translateX.Text);
9      val := StrToInt(translateX.Text);
10
11     if(val < 0) then data := 't' + 'x' + '-'
12     else data := 't' + 'x';
13
14     ser.SendString(data);
15     sleep(100);
16     data := IntToStr(val);
17     ser.SendString(data);
18     sleep(100);
19     data := '#';
20     ser.SendString(data);
21     sleep(100);
22     data := 's';
23     ser.SendString(data);
24    end;
25  end;
```

Die hier betrachtete Methode wird aufgerufen bei Betätigung des Textfeldbuttons der einzelnen Transformationsregler. Um einen fehlerfreien Betrieb zu ge-

währleisten wird zunächst überprüft, ob derzeit eine Verbindung besteht (Z.6). Daraufhin wird der Wert (*val*) der jeweiligen Transformation ausgelesen (Z.9) und auf sein Vorzeichen überprüft (Z.11-12). Zuvor wird die Reglerposition entsprechend aktualisiert (Z.8). Die Nutzeroberfläche ermöglicht eine Angabe der absoluten Position und Ausrichtung der Plattform. Nacheinander werden daher Transformationsart, Wert und Zahlenende sowie der Befehl (s) zum Setzen der Transformation über einen Aufruf der Schnittstellenvariablen ($ser.SendString$) an die Plattform gesendet (Z.14-23). Zwischen einzelnen Sendeoperationen wurden zur sicheren Übertragung Sleep-Kommandos (Z.15,18,21) eingebaut, damit die Plattform genügend Zeit hat einzelne Befehle auszuwerten.

Kapitel 5

Anwendungsbeispiel Achterbahn

Bislang wurde die Plattform nur anhand von manuellen Steuerkommandos bewegt. Dieses Kapitel befasst sich mit der automatischen und frequenziellen Ansteuerung über eine externe Software. Die Wahl fiel auf die Erweiterung einer in C geschriebenen Achterbahnsoftware (siehe Abbildung 5.1), welche im Rahmen eines Projektpraktikums durchgeführt wurde. Diese ermöglicht die Visualisierung von Beschleunigungswerten während einer Achterbahnfahrt wie links oben in Abbildung 5.2 zu sehen. Unterstützt werden Beschleunigungen um alle drei Raumachsen. Entsprechend werden für eine simple Lösung nur maximal drei Transformationen der Plattform benötigt. Ziel ist es im Folgenden, die Achterbahnsoftware um eine Funktion zum Senden der Werte zu erweitern. Den Schluss bildet eine kurze Evaluierung der Ergebnisse und damit der Echtzeiteigenschaft der erarbeiteten Lösung.

5.1 Übertragungsmodell

Die berechneten Kräfte der Achterbahn werden durch Vielfache der Erdbeschleunigung g repräsentiert. Diese Werte reichen von -8 bis 8. Zur Übertragung auf die Plattformbewegung stehen sechs Freiheitsgrade zur Verfügung, welche genutzt werden müssen um die Intensität der Kräfte sinnvoll zu repräsentieren. Die im Folgenden gewählten Transformationswerte gewährleisten einen flüssigen Betrieb der Plattform, können aber je nach Wunsch geändert werden.
Die Kräfte auf der Horizontalen (Plattformachsen X und Z) sollen dabei durch Rotationen simuliert werden. Beschleunigungen größer als $1g$ (bzw. kleiner als

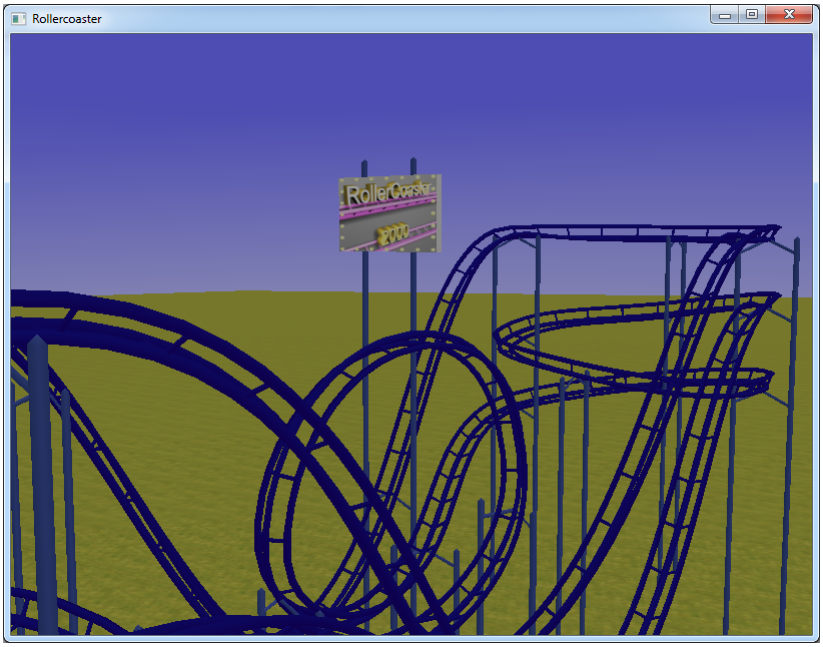


Abbildung 5.1: Die Achterbahnsoftware ermöglicht die Simulation einer Rundfahrt in einem der Achterbahnwagen.

$-1g$) entsprechen dabei der Maximalauslenkung. Da der Arbeitsraum der Plattform für Rotationen etwa in [-20;20] liegt wurde eine Auslenkung von $12°$ in beide Richtungen als Maximalwert bestimmt, damit überlagerte Transformationen einen gewissen Puffer besitzen. G-Kräfte aus [-1.0; 1.0]g müssen zunächst auf den Arbeitsraum der Plattform umgerechnet werden. Die folgende Tabelle zeigt die nötige Umwandlung der Kräfte in Transformationswerte und die Achsen, um die rotiert werden sollen.

g-Vielfache / Kraftrichtung	< -1.0	$\in [-1.0; 0]$	$\in]0; 1.0]$	> 1.0
x-Achse	Z-Rot 12	Z-Rot Value * 12	Z-Rot -Value * 12	Z-Rot -12
y-Achse	X-Rot -12	X-Rot -Value * 12	X-Rot Value * 12	X-Rot 12

Zusätzlich zu den horizontalen Beschleunigungen werden Kraftänderungen um mehr als $1g$ auf der Vertikalen durch Translationen beschrieben. Hierzu wird ein

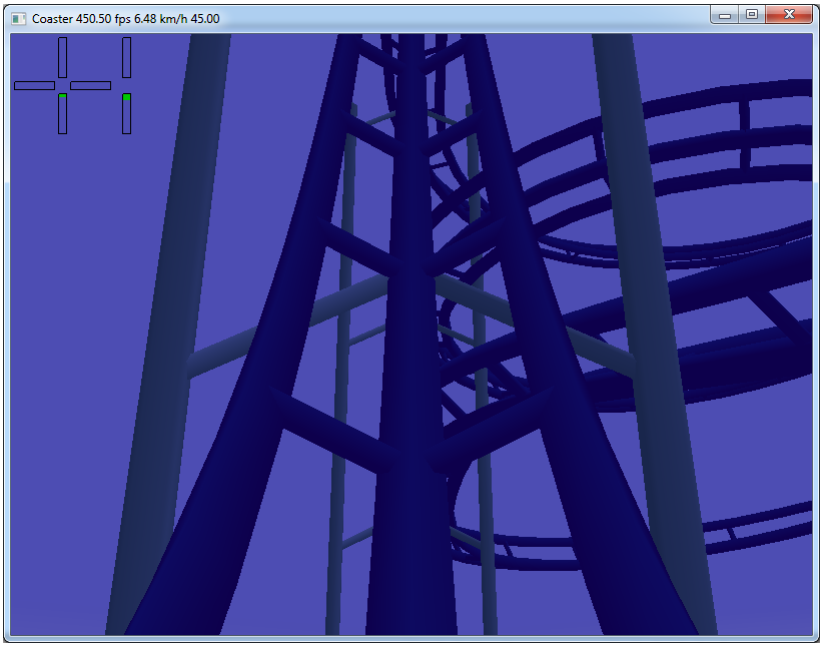


Abbildung 5.2: Es wird eine Rundfahrt in einem der Wagen simuliert, wobei links oben die wirkenden Kräfte angezeigt werden. Das Fadenkreuz zeigt die horizontalen, der Doppelbalken daneben die vertikalen Kräfte. Der Füllgrad der Balken beschreibt den Betrag der jeweiligen Kraft.

Translationsbefehl in Richtung der y-Achse der Plattform übermittelt. Das Vorzeichen des Translationswerts bzw. die Richtung ändert sich entsprechend des Vorzeichens der Differenz zweier aufeinanderfolgender Werte. Der konstante Betrag der Translation wurde auf 10 Millimeter gesetzt.
Die Umwandlung findet schließlich innerhalb der Rendermethode eines Frames der Achterbahnsoftware statt (zu finden in roller.c), da hier mehr Ressourcen zur Verfügung stehen um eine schnelle Berechnung zu gewährleisten. Diese wird im Folgenden vorgestellt.

5.2 Programmanpassung

Voraussetzung für die Übertragung ist die Kommunikation über die serielle Schnittstelle, welche über das C-Modul serial.h [Ser12a] erreicht wird. Dieses ermöglicht die Initialisierung der seriellen Schnittstelle in einem separaten Objekt sowie das zugehörige Schließen und bietet sowohl Schreib- als auch Lesefunktionen.

```
1  int main(int argc, char* argv[])
2  {
3          comPort = open_com(5, 9600, 8, 1, 0, 0);
4          ...
5          comPort = uart_close(comPort);
6          return 0;
7  }
```

Damit die Übertragung gelingt muss zunächst die serielle Schnittstelle im Hauptprogramm geöffnet werden (Z.3). Der Rückgabewert der entsprechenden Funktion ist die ID der Schnittstelle, welche im globalen Feld *comPort* der roller.c-Datei gespeichert wird. Dies erlaubt einen globalen Zugriff auf das Schnittstellenobjekt. Vor Beendigung des Programms wird die Schnittstelle wieder geschlossen (Z.5).

Die Ansteuerung der Plattform findet wie schon erwähnt in der Rendermethode statt, genannt *DrawGLScene*. Dort wird die Methode *force_draw* aufgerufen, welche die zuvor berechneten Kräfte visualisiert. In dieser Methode soll im Folgenden auch die Übertragung an die serielle Schnittstelle stattfinden.

```
1  void force_draw(void){
2          ...
3
4          static int framesPerUpdate = 0;
5          char xVal[2], yVal[2], zVal[2];
6          int xValInt, yValInt, zValInt;
7          char* numEnd = "#";
8          char* negSign = "-";
9          char* posSign = "+";
10         char* transSet = "s";
11         char* movePlatform = "m";
12         char* resetTransformation = "w";
13         char *xAxis = "x", *yAxis = "y", *zAxis = "z";
14         char *rotType = "r", *transType = "t";
15
16         if (framesPerUpdate==100) uwrite(comPort, resetTransformation, 1);
17
18         ...
19         // Beschleunigung nach links
20         if (resultforce.x < 0.0f) ...
21         // Beschleunigung nach rechts
22         if (resultforce.x >= 0.0f) ...
23         // Beschleunigung nach vorne
24         if (resultforce.y >= 0.0f) ...
25         // Beschleunigung nach hinten
26         if (resultforce.y < 0.0f) ...
27         // Beschleunigung nach unten
28         if (resultforce.z < 0.0f) ...
29         // Beschleunigung nach oben
30         if (resultforce.z >= 0.0f) ...
31         ...
32
33         if (framesPerUpdate==100) uwrite(comPort, movePlatform, 1);
34 }
```

Das hier angegebene Codefragment zeigt eine starke Vereinfachung der Methode. Zu Beginn werden einige Characterpointer definiert, welche die reservierten Zeichen der zu sendenden Befehle beinhalten (Z.7-14) oder als Variablen für die Transformationswerte dienen (Z.5-6). Die Zählvariable *framesPerUpdate* wird benötigt, um die Frequenz der Aktualisierungen der Plattformposition zu steuern. Alle 100 Frames ist eine Zurücksetzung der vorangegangenen Transformationen notwendig (Z.16) bevor eine Neuberechnung der Kräfte erfolgen kann. Ein Sendebefehl umfasst die Übergabe der Schnittstellen-ID, der zu übertragenden Zeichen sowie deren Anzahl. In den folgenden If-Abfragen (Z.19-30) werden zunächst die Richtungen und Beträge der einzelnen Kräfte ausgewertet, um entsprechende Visualisierungen durchzuführen. Diese Kontrollstrukturen sollen im Folgenden um Sendemethoden des Uartmoduls erweitert werden, um der Plattform die aktuellen Beschleunigungswerte mitzuteilen. Schlussendlich erfolgt der Befehl zur Bewegung ebenfalls alle 100 Frames (Z.33).

```
1  if (resultforce.x < 0.0f)
2  {
3          float temp = fabs(resultforce.x);
4
5          if(framesPerUpdate==100)
6          {
7          uwrite(comPort, rotType, 1);
8          uwrite(comPort, zAxis, 1);
9
10         if(temp <= 1.0f) zValInt = (int)(temp * 12.0);
11         else zValInt = 12;
12
13         zVal[0] = (char)(zValInt/10 + 48);
14         zVal[1] = (char)(zValInt%10 + 48);
15
16         uwrite(comPort, posSign, 1);
17         uwrite(comPort, zVal, sizeof(zVal));
18         uwrite(comPort, numEnd, 1);
19         uwrite(comPort, transSet, 1);
20         }
21
22         ...
23 }
```

Hier ist ein Codefragment zu sehen, welches die Auswertung der Kräfte in X-Richtung der Achterbahn beschreibt. Dieses ist im Aufbau äquivalent zur entsprechenden Auswertung bezüglich der Y-Richtung. Die Variable *platformcounter* sorgt dafür, dass die Ausrichtung lediglich alle 100 Frames aktualisiert wird. Die Richtungen der Kräfte entsprechen den Rotationsachsen aus der Tabelle des vorangegangenen Unterkapitels. Wie dort bereits erklärt, erfolgt eine Unterscheidung von Kraftbeträgen größer und kleiner als eins. Erstere erfordern eine Um-

rechnung der Beträge (Z.10), bei letzteren wird der Maximalwert (in Grad) gesetzt (Z.11). Statt der üblichen Methode *snprintf* zur Umwandlung von *int* nach *char** wird ein Typecast verwendet, der immense Geschwindigkeitsvorteile gegenüber den *printf*-Schreibmethoden bietet (Z.13-14). Dadurch werden Inkonsistenzen beim anschließenden Senden des Transformationswertes vermieden. Die gesamte Übertragung an die Plattform entspricht dem direkten Ansteuerungsmodell aus Kapitel [Serielle Schnittstelle] (Z.7-8,16-19). Der Zeile *uwrite(comPort, posSign, 1)* bei positiven Kräften entspricht der Befehl *uwrite(comPort, negSign, 1)* für negative Kräfte.

```
1  if(framesPerUpdate==100)
2  {
3  float changerate = resultforce.z - oldY;
4
5  if(fabs(changerate) >= 1.0f)
6  {
7  uwrite(comPort, yAxis, 1);
8  uwrite(comPort, transType, 1);
9
10 yValInt = 10;
11 yVal[0] = (char)(yValInt/10 + 48);
12 yVal[1] = (char)(yValInt%10 + 48);
13
14 if(changerate < 0) uwrite(comPort, negSign, 1);
15 else uwrite(comPort, posSign, 1);
16
17 uwrite(comPort, yVal, sizeof(yVal));
18 uwrite(comPort, numEnd, 1);
19 uwrite(comPort, transSet, 1);
20 }
21
22 oldY = resultforce.z;
23 }
```

Dieses Codefragment behandelt starke Kraftänderungen (*changerate*) auf der Vertikalen, also der Z-Achse der Achterbahn, welche mit Hilfe einer statischen Variablen berechnet werden. Die Abfrage wurde nicht in die Standardvisualisierung der Beschleunigungen integriert, da es sich hierbei um die Reaktion auf Änderungen und nicht auf absolute Werte handelt. Sie ist im Aufbau äquivalent zu denen der Rotationen bis auf die Tatsachen, dass es sich um Translationen handelt (Z.8) und der Transformationswert konstant ist (Z.10). Das Vorzeichen der Änderung bestimmt die Richtung der Bewegung (Z.14-15). Alle 100 Frames wird daraufhin unabhängig von der Änderungsrate der aktuelle Beschleunigungswert für die nächste Überprüfung gespeichert (Z.22).

Diese Anpassungen der Achterbahnsoftware genügen um die Plattform automa-

tisch ansteuern zu können. Die Plattformsoftware muss nicht erweitert werden. Das folgende Unterkapitel befasst sich mit dem Test der Software sowie der Auswertung der Ergebnisse.

5.3 Test und Auswertung

Der erste Testlauf verlief zufriedenstellend. Die Neigung der Plattform war konsistent zu den gezeigten Kräften. Die Bewegungen in der Vertikalen bei starken Beschleunigungsschwankungen wurden ebenfalls gut umgesetzt. Die Überlagerung mehrerer Transformationen bot keine Probleme, was zum Großteil an der Wahl der maximalen Transformationswerte lag. Bei Erhöhung dieser kam es vor, dass die Plattform ihren Arbeitsraum zu verlassen versuchte, was zu inkonsistentem Verhalten führte.
Die Geschwindigkeit der Aktualisierung (alle 100 Frames) indes änderte sich abhängig von der Anzahl der Bilder pro Sekunde, welche zwischen 30 und 700 schwankte. So war die Plattform bei hohen Bildraten besonders reaktiv, bei niedrigen jedoch weniger. Dieser Wert musste wie bei den Transformationswerten sinnvoll gewählt werden. Eine Aktualisierung in jedem einzelnen Frame (*framesPerUpdate* = 1) war beispielsweise nicht erfolgreich, da es anscheinend zu einer fehlerhaften, weil zu schnellen Übertragung der Befehle kam. Aufgrund dessen zeigte die Plattform ein unvorhersehbares Verhalten, sodass ein sofortiger Abbruch von Nöten war um Schaden am Modell zu verhindern. Die besten und flüssigsten Resultate erzielte ein Wert zwischen 100 und 200, was bei einer geschätzten Durchschnittsbildrate von 500 ca. $2,5 - 5$ Updates pro Sekunde entspricht. Diese Werte sind für den Prototypen durchaus akzeptabel.
Die Plattform zeigte zusammenfassend ein beschränktes Maß an Echtzeitfähigkeit, da die Hardware (noch) nicht höchsten Ansprüchen genügt und die Software keine stabile Bildrate liefert. Die Befehlsübertragung jedoch erwies sich als stabil und konsistent, sofern die Frequenz der Übertragung ein gewisses Maß nicht überschritt. Die automatische Ansteuerung unterliegt damit weiteren Randbedingungen zusätzlich zu Problemen beim Verlassen des Arbeitsraums (wie bei der manuellen Ansteuerung). Im folgenden Kapitel kommen einige Vorschläge zur Sprache, wie man die beschränkenden Eigenschaften des Modells weiter verbessern kann.

Kapitel 6

Schluss

6.1 Fazit und Ausblick

Das Ziel dieser Bachelorarbeit war eine generische Softwarelösung, welche von der Hardware des Plattformmodells abstrahiert. Dazu wurde zu Testzwecken zunächst ein eigenes Modell konstruiert (Kapitel [Aufbau der Plattform]), auf welches die Software später angepasst werden konnte. Dieses Modell umfasste einen Mikrocontroller als Prozessor, Servomotoren als Aktorik sowie eine Verbindung zur seriellen Schnittstelle des PCs. Der Prototyp wies im Folgenden eine im Rahmen der Qualität des vorgegebenen Materials hohe Robustheit und ausreichende Präzision auf.
Vor der Implementierung der Software wurden einige Vorüberlegungen in Form der zu erarbeitenden Softwarearchitektur und aller zugehörigen mathematischen Algorithmen benötigt, um eine sinnvolle Art der Positionskrontrolle zu ermöglichen. Das vorgestellte Bewegungsmodell sah eine Speicherung der Transformationen vor, um den Überblick über die aktuelle Position zu behalten. Die Grundlage der Ansteuerung der Aktoren bildete die Lösung der Inversen Kinematik der Stewartplattform. Durch die Anpassungen in der Aktorik wurde diese um einiges komplexer als die des Originales, ohne jedoch die Eindeutigkeit der Lösung zu verlieren. Statt einfacher Längenberechnungen mussten hier komplizierte trigonometrische Berechnungen durchgeführt werden.
Um Befehle vom PC zu senden, die am Modell als Änderung der Motorenausrichtung sichtbar werden, wurde in der Folge eine Softwarearchitektur entwickelt. Die Kommunikation mit der Hardware wurde dabei über Funktionen des Mi-

krocontrollers realisiert. Vom PC ließen sich so komplexe Befehlsfolgen über die serielle Schnittstelle übermitteln, über die die Plattform bewegt werden konnte. Aus verschiedenen Modellsichten wurde daraufhin die interne Auswertung der Befehle erläutert. Dabei werden zunächst alle Transformationen gespeichert, die Berechnung der transformierten Punkte erfolgt erst bei Aktualisierung der Motorenposition. Dies diente der Vereinheitlichung der Ansteuerung und der Reduzierung der Rechenzeit, da stets nur eine einzige Berechnung der neuen Koordinaten erfolgt. Der Fokus dieses Kapitels lag auf der Erfüllung mehrerer Qualitätsanforderungen, um die Architektur leicht erweiterbar und an verschiedene Arten der Hardware adaptierbar zu machen. Dies bedeutete die Aufteilung der Komponenten in Module, welche verschiedene Zuständigkeiten und Abhängigkeiten untereinander besaßen. Es wird unterschieden zwischen austauschbaren Peripherie- oder Hardwarekomponenten, welche Informationen zu Aktorik und Plattformmodell enthielten, und unveränderlichen Kernkomponenten, die die interne Logik entsprechend des vorangegangenen Kapitels enthielten.
Die anschließende Programmierung umfasste die Übertragung der ausgearbeiteten Objektklassen in C-Module. Die gesamte Ansteuerung der Hardware basierte dabei auf zwei Unterbrechungsroutinen für sowohl Aktorik als auch serielle Schnittstelle. Dem Nutzer wurde damit die Möglichkeit gegeben, direkte Positionierungsbefehle an die Plattform zu senden beziehungsweise eine automatisierte Bewegung zu initialisieren. Es wurde im Hinblick auf die Schnelligkeit der Plattform dafür gesorgt, dass die transformierten Eckpunkte der mobilen Plattform erst nach einem Bewegungsbefehl, entsprechend der zuvor gespeicherten Transformationen, berechnet wurden. Die Reaktivität der Plattform war dabei zufriedenstellend und Ausfälle der Funktionalität wurden nur nach Eingabe invalider Positionierungsbefehle beobachtet, was eventuell noch verbessert werden kann.
Den Abschluss der Programmierung bildete die Vorstellung der entwickelten GUI, die eine einfache manuelle Ansteuerungsmöglichkeit bot. Statt durch komplexe Befehlsfolgen konnte die Plattform über einen simplen Knopfdruck bewegt werden, was vor allem Debuggingzwecken, aber auch der anschaulichen Präsentation der Bewegungsfunktionen diente.
Das folgende Anwendungsbeispiel hingegen diente der Veranschaulichung einer anderen Ansteuerungsart. Statt direkter Nutzereingaben wurden regelmäßige Positionsdaten von einer Software an die Plattform geschickt. Dieses Beispiel zeigte, dass die Software eine ansprechende Geschwindigkeit bot, um die

Simulation einer Bewegung in Echtzeit zu gewährleisten. Es zeigte jedoch ebenso einige Probleme auf, die die Bewegung der Plattform einschränkten. Eine zu hochfrequente Verarbeitung der Positionsbefehle führte zu unerwartetem Verhalten, sodass die Frequenz der Übertragung sinnvoll gewählt werden musste um zum einen Funktionalität, zum anderen Reaktivität zu gewährleisten. Ein weiteres Problem bildete der beschränkte Arbeitsraum bedingt durch den geringen Konfigurationsraum der Motoren. Nichtsdestotrotz war das Bestehen des Anwendungstests ein Zeichen, dass das Ziel dieser Arbeit erreicht wurde.

Die angesprochenen Probleme können als Ansatzpunkt für künftige Verbesserungen der Plattform dienen, sowohl hardware- als auch softwaretechnisch.
Da das Modell nur einen ersten Prototyp darstellt, liegt es nahe, zunächst die Hardware zu überarbeiten. Die genutzten Servomotoren beispielsweise kommen aus dem Modellbau, besitzen einen geringen Konfigurationsraum, eine statische Stellgeschwindigkeit und geringe Präzision, aufgrund der analogen Signalverarbeitung. Viele dieser "Mängel" fielen in der Testphase, aufgrund der beschränkten Anwendungsmöglichkeiten, jedoch nicht ins Gewicht, sodass das aktuelle Modell ausreichte. Im Hinblick auf zukünftige Anwendungen sollten diese Probleme jedoch behoben werden, etwa durch den Einsatz von Digitalservomotoren oder Hydraulik. Die Robustheit der Konstruktion ist ein weiterer Ansatzpunkt. Bei dem im Rahmen dieser Arbeit erarbeiteten Modell handelt es sich um ein reines Testmodell, welches lediglich der Demonstration von Bewegungen und dem Debugging des Codes dienen sollte. Es wäre wünschenswert, in Zukunft ein Modell zu konstruieren, welches komplexere Aufgaben im Sinne der in der Einleitung aufgezählten Anwendungsgebiete ausführen könnte.
Erweiterungen und Verbesserungen der Software sind der zweite wichtige Ansatzpunkt, beispielsweise die Ansteuerung der Plattform. Bislang wurde diese entweder durch Nutzereingaben oder automatisch generierte Befehle durchgeführt. Eine neue Idee zur Ansteuerung umfasst die Nutzung eines Motioncontrollers wie einer 3D-Maus, welche analog zur Plattform 6 Freiheitsgrade besitzt. Die Bewegung der Maus würde dabei eins zu eins auf die Plattform übertragen. Interessant wäre überdies eine Interaktion mit entsprechender Sensorik, etwa wie in der Einleitung beschrieben zum Ausgleich von Bewegungen. Ein zweiter Mikrocontroller würde dabei die Signale auswerten und an den Prozessor der Plattform übertragen. Ein Kernproblem des Anwendungsbeispiels war, dass bei Verlassen

des Arbeitsraums der Plattform ein unvorhergesehenes Verhalten auftrat. Hierzu könnten weitere Beschränkungen und Fehlerbehandlungen in die Software implementiert werden. Da die Berechnung des Arbeitsraumes äußerst komplex und abhängig von der aktuellen Ausrichtung der Plattform ist, hätte das jedoch den Rahmen dieser Arbeit gesprengt.

Alles in allem handelt es sich bei der erarbeiteten Softwarelösung nur um die Basis für zukünftige Projekte, die den Stewart-Mechanismus nutzen. Durch die Flexibilität der Implementierung wurde sichergestellt, dass sowohl Erweiterungen als auch Qualitätsverbesserungen ohne Probleme vonstattengehen können.

Literaturverzeichnis

[Atm12] Datenblatt des AtMega16. URL: http://www.atmel.com/Images/doc2466.pdf , [Stand: 19. August 2012].

[Flu12] Full Flight Simulator (6 DoF). URL: http://en.wikipedia.org/w/index.php?title=Full_flight_simulator&oldid=506624626 , [Stand: 18. August 2012].

[GW62] V. E. Gough and S. G. Whitehal. Universal tyre test machine. *Ninth international automobile technical congress*, page 122, 1962.

[JJ02] Domagoj Jakobovic and Leonardo Jelenkovic. The forward and inverse kinematics problems for stewart parallel mechanisms. page 7, 2002.

[Koo12] Überführung von Punkten von einem Koordinatensystem in ein anderes. URL: http://de.wikipedia.org/w/index.php?title=Koordinatentransformation&oldid=101755691 , [Stand: 31. August 2012].

[Nas12] NASA Low Impact Docking System. URL: http://en.wikipedia.org/w/index.php?title=NASA_Docking_System&oldid=498033988 , [Stand: 18. August 2012].

[Pol12] Technische Bestandteile. URL: http://www.pollin.de , [Stand: 8. August 2012]. Atmel Evaluationsboard Version 2.0.1, MicroController AtMega16 sowie Streifen-/Punktrasterplatinen-Adapter.

[Rob12] Robocrane. URL: http://en.wikipedia.org/w/index.php?title=Robocrane&oldid=459029496 , [Stand: 18. August 2012].

[Ser12a] Modul zur Ansteuerung der Seriellen Schnittstelle in C. URL: http://userpages.uni-koblenz.de/~physik/informatik/hwp/flash/ , [Stand: 18. August 2012].

[Ser12b] Conrad Servo. URL: http://www.conrad.de , [Stand: 8. August 2012]. Modelcraft Standard Servo RS2 mit Gleitlagergetriebe.

[Sin12] Linearkombination von Sinuswellen verschiedener Phasen. URL: http://en.wikipedia.org/w/index.php?title=List_of_trigonometric_identities&oldid=508088705 , [Stand: 19. August 2012].

[Ste65] D. Stewart. A platform with six degrees of freedom. *Proceedings of the Institution of Mechanical Engineers*, 180:372, 1965.

[Tay12] Taylor Spacial Frame. URL: http://en.wikipedia.org/w/index.php?title=Taylor_Spatial_Frame&oldid=491666819 , [Stand: 18. August 2012].

[Tel12] Hexapod Teleskop in Chile, vormals Bochum. URL: http://de.wikipedia.org/w/index.php?title=Hexapod-Teleskop&oldid=86831486 , [Stand: 18. August 2012].

[Tim12] Kurzdokumentation der AVR internen Timer. URL: http://www.mikrocontroller.net/articles/AVR-Tutorial:_Timer , [Stand: 19. August 2012].