Selbstorganisierendes Orientierungssystem für mobile autonome Roboter

Jugend forscht 2004

von
Johannes Dörr
und
Florian Manteuffel

6. Februar 2004

Inhaltsverzeichnis

Inhaltsverzeichnis 1

1. Einleitung 2
1.1 Einsatz von mobilen Robotern 2
1.2 Problematik und Zielsetzung 2

2. Kartenaufzeichnung 3
2.1 Die Standardmethode 3
2.2 Funktionsweise unseres Verfahrens 3

3. Wegberechnung 4
3.1 Einem Hindernis ausweichen 4
3.2 Programmfluss 4
3.3 Falsche Berechnungen 5
3.4 Erweiterungen 5

4. Positionsbestimmung 6
4.1 Wegmessung 6
4.2 Positionskontrolle 6
4.3 Karte 7
4.4 Installation 8

5. Abläufe und Navigationsmodi 8
5.1 Aufgabenmodus 8
5.2 Erkundungsmodus 8
5.3 Administrationsmodus 9

6. Technische Umsetzung 10
6.1 Aufbau des Roboters 10
6.2 Sekundäre Elektronik 11
6.3 Prozessverteilung und programmiertechnische Implementierung 12
6.4 Kontrollsoftware 13

7. Zusammenfassung 15
7.1 Modell für ein mobiles Robotersystem 15
7.2 Form des Roboters 15
7.3 Ausblick 15

8. Anhang 16
8.1 Begriffserklärung 17
8.2 Kontrollsoftware 18
8.3 Wegberechnung 23
8.4 Schaltpläne 26
8.5 Danksagungen 27
8.6 Literaturangaben 27

1. Einleitung

1.1 Einsatz von mobilen Robotern

Obwohl schon lange sehr intensiv auf dem Gebiet der Robotik geforscht und entwickelt wird, spielen intelligente Roboter heute noch keine große Rolle in unserem alltäglichen Leben. Lediglich Industrieroboter sind beispielsweise in der Autoindustrie von großer Bedeutung, allerdings führen diese nur wiederholt identische Bewegungen aus und sind auf eine bestimmte Aufgabe beschränkt.
Ziel der Robotik ist es jedoch, computergesteuerte, universelle Maschinen zu entwickeln, die je nach Ausgangslage Entscheidungen treffen und mit dem Menschen in Kontakt treten und ihn unterstützen können. Ausschlaggebend hierfür ist unter Anderem ein ausgeprägtes Lernsystem, mit dem sich der Roboter eigenständig seiner Umgebung anpassen und von ihr lernen kann. Außerdem ist bei mobilen Robotern ein Navigationssystem mit Positionsbestimmung nötig, mit dem ein kollisionsfreies Bewegen möglich ist.

Die Anwendungsgebiete solcher mobilen Roboter sind vielfältig. Neben dem Einsatz in Lagerhallen oder bei der Erforschung von Planeten, bei denen direkte Steuerungen auf Grund der langen Datenübertragung nur schwer möglich sind, sind in Zukunft auch künstliche Haushaltshilfen oder selbststeuernde Autos denkbar. Neben der Lösung von speziellen Aufgabenstellungen wie Entnehmen von Bodenproben oder Leeren eines Briefkastens ist das Orientierungsproblem allgegenwärtig. Hat ein mobiler Roboter nicht die Fähigkeit, erstens festzustellen, wo er sich befindet, und zweitens, wie er zu einem anderen Punkt gelangen kann, ist er nicht wirklich einsatzfähig.
Deshalb hat sich die Entwicklung von Navigationssystemen für mobile Roboter zu einer Art Grundlagenforschung auf dem Gebiet der Robotik entwickelt.

1.2 Problematik und Zielsetzung

Ein Orientierungssystem für mobile Roboter muss vielen Anforderungen gerecht werden. Neben der Möglichkeit, Hindernisse zu erkennen, sollte auch eine Kartenfunktion vorhanden sein, mit der nicht befahrbare Regionen abgespeichert werden können, um sie später sofort zu umfahren. Dabei ist eine Rücksichtnahme auf Fehler in der Karte unerlässlich. Ungenaue Messwerte oder nicht mehr vorhandene Hindernisse können den Roboter dazu veranlassen, freien Flächen auszuweichen, obwohl sie problemlos passierbar wären. Die Karte muss also genauso dynamisch wie die Umwelt sein. Es gibt mehrere Lösungen für die oben genannten Probleme. Auf freien Flächen ist per GPS eine absolute Positionsbestimmung möglich, deren Genauigkeit in den nächsten Jahren noch weiter steigen wird. In geschlossenen Räumen hingegen müsste man auf andere, eigene Systeme ausweichen. Hierfür gab es bereits Projekte, die beispielsweise mit Ultraschall eine genaue Bestimmung der Roboterposition ermöglichten.
Auch die Berechnung des kürzesten Weges durch eine Anzahl von Hindernissen ist schon mit mehreren Algorithmen möglich, die besonders durch die Entwicklung von Computerspielen an Bedeutung gewonnen haben. Sehr verbreitet auf diesem Gebiet ist der A*-Algorithmus, der die Karte in ein Raster zerlegt, dessen Felder mit einer gewissen Wahrscheinlichkeit entweder befahrbar oder gesperrt sind.
Leider sind diese Verfahren oftmals nur schwer in einer unpräparierten Umgebung einsetzbar und funktionieren entweder nur auf einem Tisch oder gar nur im Rechner. Außerdem sind die Roboter meistens speziell auf das eine Verfahren ausgerichtet und von ihm abhängig. Auch die Installation eines Roboters sollte nicht nur von speziell ausgebildetem Fachpersonal ausgeführt werden können, sondern sollte im Idealfall vollautomatisch ablaufen. Genau justierte Zusatz-Hardware für die Positionsbestimmung ist deshalb äußerst unpraktisch. Zweitens funktioniert der A*-Algorithmus in Computersimulationen sehr gut, ist aber für die nicht-digitale Umwelt weniger geeignet.
Als Ziel unseres Projektes haben wir uns deshalb gesetzt, ein Robotersystem zu entwickeln, das eine zuverlässige Orientierung auch in unbekannten Räumen ermöglicht, trotzdem äußerst einfach zu installieren ist und eine große Ausbaufähigkeit aufweist. Dazu gehört die Entwicklung eines Kartenmoduls mit Wegberechnungs-Algorithmus, ein Verfahren zur Ermittlung der Eigenposition des Roboters, sowie die technische und mechanische Umsetzung des Roboters, mit der das System getestet, weiterentwickelt und präsentiert werden kann. Zusätzlich ist eine Kontrollsoftware nötig, mit der das Robotersystem mit anderen externen Systemen (z.B. andere Robotereinheiten, elektrische Türen etc.) verknüpft werden kann.

Unsere Arbeit hat nicht das Ziel, ein fertiges System vorzustellen, sondern ein Modell zu entwickeln, das sich aber trotzdem in der Praxis beweisen soll. Analog dazu gliedert sich auch diese schriftliche Arbeit. Die Kapitel 2 bis 5 befassen sich mit dem theoretischen Aufbau so eines Systems, das längere Kapitel 6 erläutert unsere technische Umsetzung.

2. Kartenaufzeichnung

Die Grundlage für eine zielgerichtete Navigation durch mehrere Hindernisse ist eine Karte. Mit Hilfe von Sensorik, mit der Hindernisse erkannt werden können, und Kenntnis der Eigenposition können Regionen als befahrbar oder nicht befahrbar vermerkt werden, sodass später auch weit entfernte Hindernisse, die außerhalb der Sensorreichweite liegen, bei der Navigation berücksichtigt werden können.

2.1 Die Standardmethode

Ein oft verwendetes Prinzip ist das Speichern der Karte in einem zweidimensionalen Datenfeld. Jedes Element dieses Arrays speichert die Eigenschaft eines bestimmten Quadrates der Umgebung. Der große Nachteil ist jedoch die Form und die daraus entstehende Datenmenge.

- Auch leere Bereiche der Karte verbrauchen Speicherplatz, schließlich tragen die einzelnen Elemente mindestens die Information „befahrbar". Karten können demnach auch dann sehr speicherintensiv sein, wenn sie wenige Hindernisse speichern, diese aber weit auseinander liegen.
- Datenfelder müssen dimensioniert werden, wodurch die Größe der Karte und damit der Einsatzbereich des Roboters festgelegt wird. Ein Umdimensionieren zum Ändern der Kartengröße ist zwar in höheren Programmiersprachen durchaus möglich, jedoch meistens aufwändig. Auch der Zuwachs an benötigtem Speicherplatz darf hier nicht vergessen werden.
- Es muss eine Auflösung festgelegt werden, die angibt, in wie viele Elemente die Umgebung unterteilt wird (Maßstab). Eine große Auflösung führt zu einer großen Datenmenge, eine kleine zu Ungenauigkeiten und Fehlern.

2.2 Funktionsweise unseres Verfahrens

Da die Karte unseres Roboters auch große Flächen verwalten können und außerdem höchst flexibel sein soll, haben wir ein neues Speicherverfahren programmiert. Hierbei existiert keine grafische Abbildung der Umwelt, sondern nur eine Ansammlung von Koordinaten, die die Positionen der Hindernisse angeben. Es wird also nur Speicherplatz für Hindernisse verwendet, freie Flächen ergeben sich aus dem Rest.
Ein Hindernis besteht aus mehreren, jedoch möglichst wenigen Points, die die Ecken darstellen. Der Bereich, der von den Points eingeschlossen ist, gilt als unbefahrbar. Die grundlegende Struktur sieht folgendermaßen aus:

```
Map
   Obstacle(1)          // Hindernis-Objekt
      Precision         // Precision-Eigenschaft des Hindernisses
      Point(1)          // Point (Einer der Eckpunkte des Hindernisses)
         x-pos          // x-pos-Eigenschaft des Points
         y-pos          // ...
         a-distance
         b-distance
         x-distance
      Point(2)
      Point(3)
      // weitere Points möglich
   Obstacle(2)          // Hindernis-Objekt
   // weitere Objekte möglich
```

Abb. 2.1: Datenverwaltung der Karte

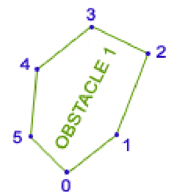


Abb. 2.1: Hindernis mit Points

Die virtuelle Karte enthält eine beliebige Anzahl von Hindernissen. Jedes dieser Hindernisse enthält die Eigenschaft ‚Precision' sowie eine beliebige Anzahl von Points. Alle Points enthalten die Eigenschaften für die x- und y-Koordinaten sowie für weitere Daten, die später erläutert werden, da sie erst für die Wegberechnung wichtig sind.
Um eine möglichst genaue Abbildung der Umwelt zu ermöglichen, erhalten alle Hindernisse eine Precision-Eigenschaft, in die die geschätzte Genauigkeit des Hindernisses gespeichert wird. Später kann der Roboter dann entscheiden, ob er die Hinderniskoordinaten erneut abspeichern will, wenn die neue Messung eine genauere Beschreibung liefert. Dieses Verfahren wird weiter unten detaillierter beschrieben.
Ein Hindernis in der Karte ist also nicht bloß eine Ansammlung von nicht befahrbaren, quadratischen Flächen sondern ein wirklich zusammenhängendes Objekt. Für Erweiterungen des Systems hat dies den Vorteil, dass diese Objekte zusätzliche Eigenschaften (z.B. ‚gültig bis', Art des Hindernisses, etc.) bekommen oder weitere Objekt-Arten hinzugefügt werden können. Solche Objekte können Positionskontrollverfahren kennzeichnen (› siehe Kapitel 4.3), aber auch Hauptstraßen oder elektrische Türen (› siehe Kapitel 3.4) darstellen.
Ein weiterer Vorteil ist, dass hier die Karte an kein Raster gebunden ist und somit alle oben angesprochenen Nachteile der Array-Methode ausgeschaltet sind. Wir haben mit dieser Methode also ein sehr flexibles und ausbaufähiges Speicherverfahren zur Verfügung.

3. Wegberechnung

Um eine Navigation durch die Karte zu ermöglichen, muss ein Wegberechnungsalgorithmus vorhanden sein, der dem Roboter den Weg weist. Allerdings kann die Wegberechnung nur eine Richtfunktion erfüllen. Die Navigation beispielsweise durch eine Tür muss immer von Sensorik überwacht werden, damit Fehlberechnungen nicht in einem Zusammenstoß enden. Auch für den Abstand, in dem ein Hindernis umfahren wird, wird nicht durch die Wegberechnung gesteuert.

3.1 Einem Hindernis ausweichen

Im Idealfall kann das Ziel auf geradem Weg erreicht werden, ohne dass einem Hindernis ausgewichen werden muss. Um zu überprüfen, ob ein Hindernis den direkten Weg unterbricht, existiert für jedes Hindernis eine Funktion, die zurückgibt, ob sich ein Punkt (meistens die Roboterposition) auf seinem Trace befindet. Als Trace kann man sich den Bereich vorstellen, in dem man das Ziel nicht sehen kann, weil das Hindernis die Sicht nimmt (*Abb. 3.1*). Dabei wird überprüft, ob die Verbindungslinie von dem Punkt zum Ziel eine der Verbindungslinien der Points schneidet (rote Linie).

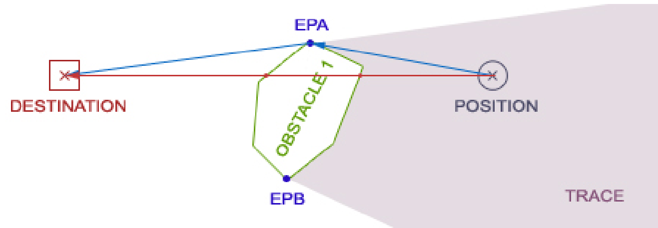


Abb. 3.1: Traces

Grundlage unseres Wegberechnungs-Algorithmus ist die eigentlich recht simple Idee, dass bei dem Umfahren eines Hindernisses einer seiner Points als Ansteuerungspunkt verwendet wird. *Abbildung 3.1* soll dies veranschaulichen. Ein direkter Weg ist nicht möglich (Roboterposition liegt auf Trace), es muss deshalb ein Point des Hindernisses angesteuert werden, von dem man direkt zum Ziel kommt. Diese Points nennen wir Endpoints. Sie werden ermittelt, indem die beiden Points gesucht werden, von denen die restlichen Points alle rechts bzw. alle links liegen. Demnach besitzt ein Hindernis immer zwei Endpoints, bezeichnet mit EPA und EPB, die von der Position des Ziels abhängen.

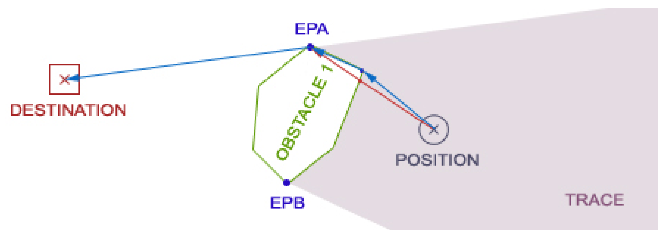


Abb. 3.2: Endpoint nicht direkt erreichbar

Ist der Endpoint des Hindernisses nicht direkt erreichbar (*Abb. 3.2*), muss ein anderer Point gewählt werden, von dem danach der Endpoint angesteuert wird.

3.2 Programmfluss

Die Wegberechnung ist in zwei Teile aufgeteilt (PART A und B). Der erste und komplizierteste Teil der Wegberechnung wird immer dann durchgeführt, wenn die Karte geändert wurde (Hindernis hinzugefügt, neues Ziel definiert usw.). Dabei wird die Karte analysiert und vermessen. Die dabei ermittelten Werte werden in PART B benötigt. PART B liefert dem Roboter während der Fahrt eine Wegbeschreibung, die ihn zum Ziel führt. Damit PART B arbeiten kann, muss PART A abgeschlossen sein, da letzterer Daten berechnet, die von PART B gebraucht werden (*Abb. 3.3*).

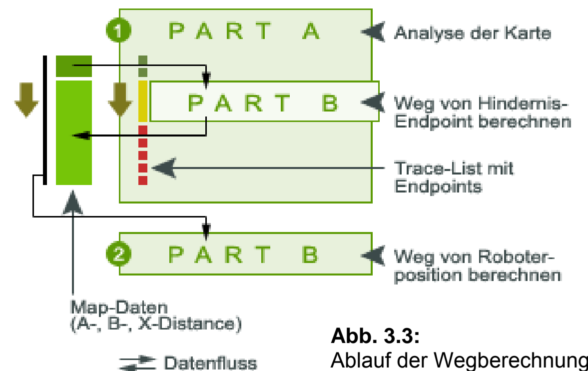


Abb. 3.3: Ablauf der Wegberechnung

PART A beginnt damit, von jedem vorhandenen Hindernis die EPs zu berechnen. Anschließend wird in jedem Hindernis von jedem Point die Entfernung zu EPA und EPB gemessen (*Abb. 3.4*) und in seinen Eigenschaften A- und B-Distance (› siehe Kapitel 2) gespeichert. Diese beiden Eigenschaften der EPs selber sind dementsprechend immer 0.

Jeder Point besitzt auch die Eigenschaft X-Distance. Sie wird allerdings nur bei EPs verwendet und gibt die Entfernung zum Ziel an.

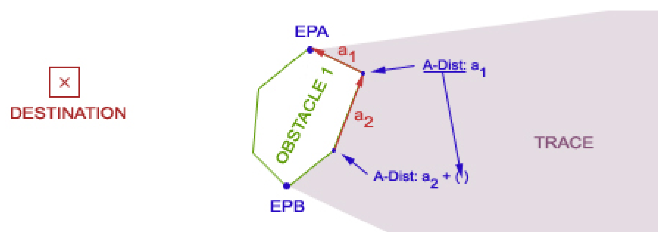


Abb. 3.4: Berechnung von A-Distance

Später kann damit von mehreren Endpoints der mit der kleinsten Entfernung gewählt werden, indem einfach diese Eigenschaften verglichen werden.

Da die meisten EPs keine direkte Sichtverbindung zum Ziel haben, muss schon während PART A eine Wegberechnung ablaufen (› siehe Kapitel 3.1). Es wird also schon während PART A PART B ausgeführt.
Um dies zu ermöglichen, muss sichergestellt werden, dass, obwohl PART A noch nicht vollständig abgeschlossen ist, alle Daten der Points (A-, B- und X-Distance), auf die dabei zugegriffen wird, schon vorhanden sind. Dazu wird eine Liste angelegt, in der alle EPs nach der Anzahl der Traces, in denen sie liegen, sortiert sind. Die blauen Zahlen in der Grafik (*Abb. 3.5*) geben die Position der EPs in dieser Liste wider.

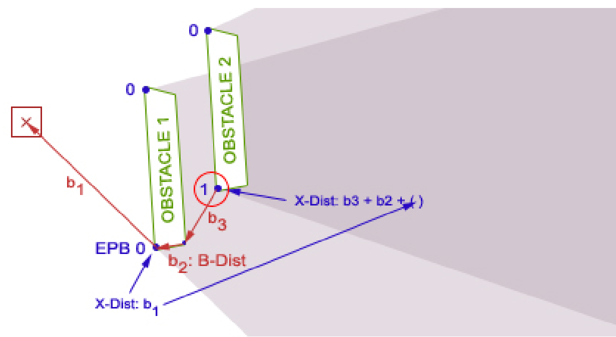


Abb. 3.5: Berechnung von X-Distance

Nun beginnt PART B, von den EPs an Position 0 (0 Traces, also direkter Weg zum Ziel) jeweils die Entfernung zum Ziel zu berechnen und sie als X-Distance abzuspeichern. X-Distance lässt sich hier leicht berechnen (b_1), da keine Hindernisse im Weg sind.
An Position 1 (1 Trace, kein direkter Weg zum Ziel, ein Hindernis im Weg) ist das schon schwieriger. Als Beispiel dafür betrachten wir auf dem Bild (*Abb. 3.5*) den markierten Point (roter Kreis). Um von ihm aus zum Ziel zu kommen, muss einer der EPs angesteuert werden, deren Position in der Liste kleiner als 1 ist. In diesem Beispiel gibt es dafür drei Möglichkeiten, die alle getestet werden. Dies läuft folgendermaßen ab: Es wird bei allen dieser drei Endpoints überprüft, ob sie direkt erreichbar sind. Bei EPB von Hindernis 1 ist dies nicht der Fall. Deshalb wird nun getestet, ob irgendein anderer Point von Hindernis 1 erreichbar ist. Wurde so ein Point gefunden, wird die Entfernung errechnet, die man beim Ansteuern des Ziels über diesen Points zurücklegen müsste. Sie ist die Summe aus der Entfernung zu diesem Point (b_3), B-Dist dieses Points (b_2) und X-Distance von Endpoint B. Dieser Entfernungswert wird mit denen verglichen, die bei der Ansteuerung der anderen Endpoints (bzw. normalen Points, wenn jene nicht direkt ansteuerbar sind) entstehen. Am Ende wird der Point verwendet, der die kleinste Entfernung ergab. Dieser Entfernungswert wird als X-Distance von dem Beispiel-Endpoint, von dem wir ausgegangen sind, gespeichert.
Am Ende von PART A haben alle Points vollständige Eigenschaften, mit denen man feststellen kann, wie weit sie vom Ziel entfernt sind. Möchte man nun mit dem Roboter von einer bestimmten Position zum Ziel fahren, wird PART B aufgerufen, der zurückgibt, welcher Punkt dafür als nächstes angesteuert werden muss.

3.3 Falsche Berechnungen

Unser Wegberechnungs-Algorithmus gibt leider nicht immer sofort den richtigen Weg vor, was allerdings oftmals ohne große Bedeutung ist, da sich die Werte während der Fahrt korrigieren oder im schlimmsten Falle nur zu einem geringfügig längeren Weg führen.
In dem Fall von *Abbildung 3.6* steuert der Roboter zuerst den Endpoint von Hindernis 1 an. Da ein Hindernis nach dem Ausweichen immer am Endpoint verlassen wird, wird der Weg über den Point rechts unten (Hindernis 1, blauer Weg) nicht erkannt und Hindernis 2 angesteuert. Erst wenn der Point links oben (Hindernis 2) direkt erreichbar wird, fährt der Roboter einen Schlenker und nimmt nun den richtigen Weg. Die Konzentration auf die Endpoints hat also auch Nachteile, spart aber Rechenzeit.

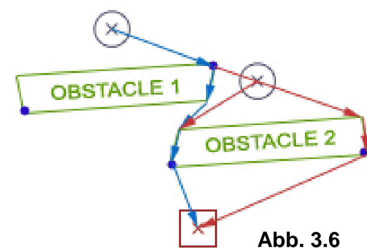


Abb. 3.6

Auch die Berechnung von A- und B-Distance der einzelnen Points kann zu Fehlern führen, wie dieses Beispiel (*Abb 3.7, links*) zeigt. Einbuchtungen werden nämlich nicht berücksichtigt. Das hat zur Folge, dass der Weg länger geschätzt wird, als er wirklich ist. Der Roboter würde in diesem Fall links herum (roter Pfeil, grüner Weg) fahren.
Das Problem ist dabei nicht das Feststellen der Einbuchtungen sondern die Möglichkeit, dass in dieser Einbuchtung ein anderes Hindernis liegt, das den direkten Weg doch versperren würde (*Abb 3.7, rechts*). Da die Berechnung jedoch Hindernis-intern abläuft, kann auf andere Hindernisse keine Rücksicht genommen werden.

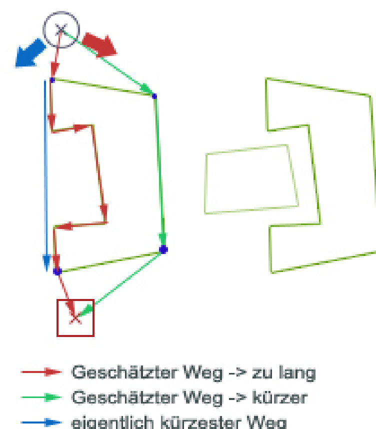


Abb. 3.7

3.4 Erweiterungen

Wie schon in 2.2 beschrieben wurde, ist es ohne weiteres möglich, das Orientierungssystem um neue Objekte zu erweitern. Für die automatische Aufzeichnung der Karte mögen normale Hindernisse vielleicht ausreichen, wenn sich der Roboter jedoch in einer komplizierten Umgebung bewegen soll, führt eine manuelle Hilfe manchmal zu besseren Ergebnissen. Denkbar wären zum Beispiel ‚Hauptstraßen', die dem Roboter Wege empfehlen oder vorschreiben. Soll ein Roboter zum Beispiel mehrere Gebäude überbrücken, wäre es ohne Probleme möglich, ihm zu verbieten, einen anderen Weg als die Straße zu nehmen.

Auch elektrische Türen (die normalerweise nur auf Körperwärme reagieren und deshalb einen Roboter nicht durchlassen würden) könnten eingespeichert werden, sodass automatisch ein Signal abgegeben wird, das die Tür öffnen lässt. Ein entsprechendes Interface dafür müsste natürlich vorhanden sein (› siehe Kapitel 6.5).
Die Liste ließe sich fortsetzen und beweist, wie viele Erweiterungsmöglichkeiten unser System bietet.

4. Positionsbestimmung

Die Verbindung zwischen Karte und wirklicher Umwelt ist die Positionsbestimmung. Die Schwierigkeit bei ihr liegt in den immer auftretenden Ungenauigkeiten, die bei der Aufzeichnung von Hindernissen zu Verschiebungen in der Karte und beim Ansteuern eines Ziels zu falschen Wegberechnungen führen. Apparaturen, mit denen sehr genaue Positionsbestimmungen möglich sind, haben oft den Nachteil, dass sie eine aufwendige und genaue Installation erfordern, besonders dann, wenn sie über mehrere Räume hinweg funktionieren sollen. Einfache Verfahren, wie zum Beispiel Odometrie (Wegmessung durch die Auswertung der Radumdrehungen), sind hingegen meistens zu ungenau.

4.1 Wegmessung

Schon bei Erscheinen der ersten optischen Mäuse vor ein paar Jahren hatten wir die Idee, sie als Sensor für die Wegmessung bei einem mobilen Roboter zu verwenden. Dies hätte nämlich den Vorteil, dass der Weg direkt gemessen wird und nicht durch Schlupf der Reifen oder Ähnlichem abgefälscht werden kann. Außerdem haben Mäuse eine gute Genauigkeit.
Ein Problem ist die geringe Entfernung vom Boden, die der Sensor haben muss. Spätestens ab 5 Millimetern erkennt er in der Regel keine Bewegung mehr. Da ein größerer Abstand jedoch unbedingt nötig ist, damit der Sensor bei unebenen Untergründen nicht auf dem Boden schleift, muss die Optik der Maus mit einer zusätzlichen Linse manipuliert werden. Bei Conrad Elektronik fanden wir die passende Linse, die den Abstand auf 3 bis 4 Zentimeter erweitert.

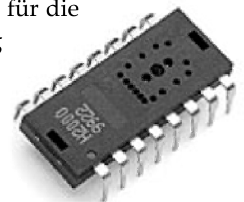


Abb. 4.1: HDNS-2000

Für die Ansteuerung der Maus bieten sich zwei Möglichkeiten an. Entweder wird ein PS/2-Anschluss am Mikrocontroller simuliert, an den die Maus dann wie an einen PC angeschlossen wird, oder man baut den Baustein aus und nimmt die Daten direkt entgegen. Der Chip HDNS-2000 der Firma Agilent Technologies (*Abb. 4.1*) besitzt glücklicherweise nicht nur einen PS/2-, sondern auch zwei Quadratur-Ausgänge, an denen die Bewegungen in X- und Y-Richtung durch Impulszählung abgenommen werden können. Tastenklicks werden dabei zwar ignoriert, aber von uns sowieso nicht gebraucht.
Die Positionierung der optischen Sensoren für eine Positionsbestimmung ist selbstverständlich von großer Bedeutung. *Abbildung 4.2* zeigt zwei Möglichkeiten. Die eine benötigt zwei Sensoren, die neben den Rädern angebracht sind. Sie messen praktisch den Weg, den jedes einzelne Rad zurücklegt. Mit der zweiten Möglichkeit bräuchte man nur einen Sensor. Da wir allerdings bei dem Entwurf einer Auswertungsmethode für diese Anordnung nicht zu einem befriedigendem Ergebnis kamen, entschieden wir uns für die erste Möglichkeit, da diese von der Art der Daten der klassischen Odometrie sehr ähnlich ist und wir aus diesem Grund auf bereits vorhandene Formeln zurückgreifen können.

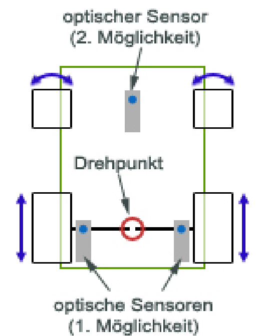


Abb. 4.2: Anbringung der Sensoren

4.2 Positionskontrolle

So genau ein Verfahren zur Wegmessung auch sein mag, solange es auf relativen Messungen beruht, ist es fehleranfällig. Ein absolutes Messverfahren ist also nötig, um die Ungenauigkeiten zeitweise auszugleichen. Da dieses Verfahren möglichst wenig Installationsbedarf haben soll, haben wir uns für diesen Teil unseres Systems die Entwicklung eines optischen Verfahrens vorgenommen, das möglichst dynamisch und vielseitig ist, außerdem nicht zu teuer wird, aber gleichzeitig gute Ergebnisse liefert. Auf der Suche nach geeigneter Elektronik stießen wir auf ein Kamerasystem namens CMUcam, das am Robotics Insitute der Carnegie Mellon Univerity in Pittsburgh entwickelt wurde. Die Platine übernimmt sämtliche Bildauswertungen und ist somit relativ einfach zu verwenden, da die Bildverarbeitung nicht selbst implementiert werden muss, wofür schnellere Rechenleistung nötig wäre, die auf mobilen Robotern immer rar ist.
Für die optische Positionskontrolle haben wir gleich mehrere Ideen entwickelt, um sie ausführlich testen, verbessern und kombinieren zu können.
Bildverarbeitung mit dem Ziel, alltägliche Objekte zu erkennen, ist nur mit sehr viel aufwändigerer Gerätschaft möglich. Unsere Verfahren beschränken sich deshalb darauf, simple, farblich gekennzeichnete Orientierungspunkte zu erkennen. Das Problem hierbei ist, dass die Kamera die entsprechenden Farben gelegentlich auch bei anderen Gegenständen erkennt, die keine Orientierungspunkte sind. Um das zu umgehen, verwenden wir als Farbpunkte retroreflektierende Folien. Diese haben die

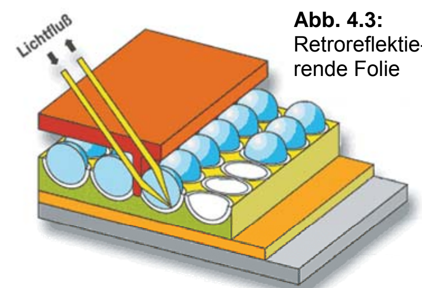
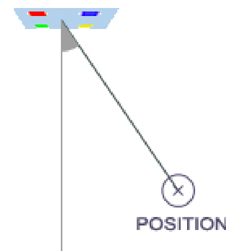


Abb. 4.3: Retroreflektierende Folie

Eigenschaft, auftreffendes Licht genau in die entgegengesetzte Richtung zurückzustrahlen (*Abb. 4.3*). Wenn nun ein LED-Scheinwerfer, der an der Kamera platziert ist, auf die Folie leuchtet, reflektiert sie nur dieses Licht zur Kamera zurück. Dadurch ist das Verfahren gegen Umgebungslicht weitgehend störsicher. Außerdem kann durch eine zweite Aufnahme ohne Scheinwerfer der Punkt überprüft werden, der nun nicht mehr in gleicher Helligkeit zu sehen sein dürfte. Die Orientierungspunkte sind also einfach durch Aufkleben zu montieren und brauchen keine Stromversorgung, was bei einer Lösung mit LEDs der Fall wäre.

4.2.1 Orientierungspunkte

Ein Orientierungspunkt, mit dem die Position des Roboters ermittelt werden kann, muss aus vier Farbfeldern bestehen, deren Abstand voneinander genau definiert sein muss. Hat der Roboter alle Felder im Bild, so kann über die Messung der Abstände auf die Eigenposition geschlossen werden. Die Felder Rot und Grün werden dabei zuerst ausgewertet, danach Blau und Gelb. Bei beiden Paaren wird der Abstand der beiden Felder zueinander gemessen, und der Durchschnittswert dieser beiden Werte lässt auf die Entfernung vom Roboter zum Orientierungspunkt schließen.



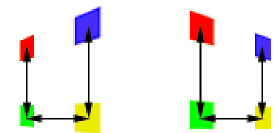


Abb. 4.4: Orientierungspunkte, verschiedene Blickwinkel

Über den Abstand zwischen dem Grünen und dem Gelben Feld wird der (Blick-) Winkel berechnet. Um nun festzustellen, ob der Roboter von links oder von rechts auf den Orientierungspunkt schaut, werden nochmals Rot-Grün und Blau-Gelb überprüft. Ist der Abstand von ersterem kleiner, so steht der Roboter rechts, andernfalls links vom Punkt.
Solche Orientierungspunkte können ohne großen Aufwand an beliebigen Wänden aufgeklebt werden. Der Roboter kann dann an dieser Stelle seine Positionsinformation aktualisieren.

4.2.2 Schneisen

Ein anderes Verfahren ermöglicht zwar keine konkrete Positionsbestimmung, ist allerdings wesentlich unempfindlicher. Es arbeitet nach dem Prinzip eines Leuchtturms und kann den Roboter zu einer Position weisen. Dies ist zum Beispiel bei der Homebase sehr sinnvoll, damit der Roboter auch mit sehr ungenauen Positionsdaten wieder zurückfindet.
Drei Farbfelder werden so aufgebaut, dass im richtigen Winkel die Farbe Gelb und weiter außen Rot bzw. Grün zu sehen ist. Mit Hilfe der Kamera kann der Roboter so genau auf das gelbe Feld zusteuern und die Homebase erreichen. In Verbindung mit schwarzen Linien auf dem Boden kann der Roboter sogar zentimetergenau in die Homebase einfahren. Die Schneise lenkt den Roboter so lange in die richtige Richtung, bis eine Führungslinie erreicht ist, der der Roboter dann mit Hilfe eines Sensors folgt. Sind an der Homebase zum Beispiel spezielle Kontakte angebracht, kann über diese Steuerung gut gewährleistet werden, dass diese beim Einfahren auch wirklich vom Roboter berührt werden.

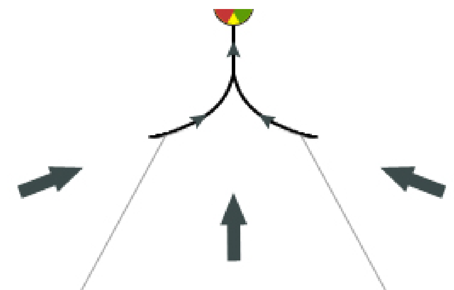


Abb. 4.5: Schneise mit Bodenlinien

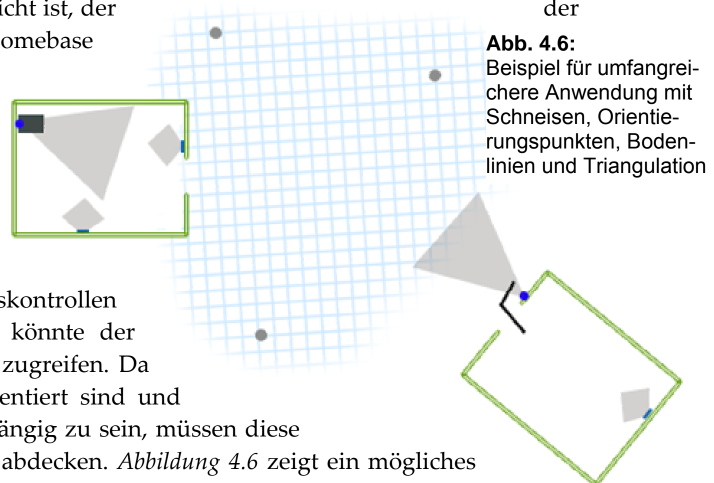


Abb. 4.6: Beispiel für umfangreichere Anwendung mit Schneisen, Orientierungspunkten, Bodenlinien und Triangulation

Es sind noch viele weitere Methoden zur Positionskontrolle denkbar. Während die Wegmessung die Grundlage darstellt, ist das System um beliebig viele verschiedene Positionskontrollen erweiterbar. Neben Orientierungspunkten und Schneisen könnte der Roboter beispielsweise auch auf ein GPS-ähnliches System zugreifen. Da solche Positionskontrollsysteme als Zusatzmodule implementiert sind und der Roboter auf sie bei Bedarf zugreift, ohne von ihnen abhängig zu sein, müssen diese keineswegs die gesamte Fläche des zu befahrenen Bereichs abdecken. *Abbildung 4.6* zeigt ein mögliches Beispiel einer Karte, in der zwei Gebäude überbrückt werden müssen. Innerhalb der Gebäude wird größtenteils mit Orientierungspunkten gearbeitet, zwischen den Gebäuden ist ein Triangulationsverfahren vorgesehen. Außerdem sind zwei Schneisen (Homebase im linken Gebäude; Eingang rechtes Gebäude, mit Bodenspur, auf der der Roboter durch die Tür findet) vorhanden.

4.3 Karte

In den meisten Fällen ist es sinnvoll, die vorhandenen Positionskontrollverfahren bestimmten Regionen in der Karte zuzuordnen. Besonders auf großen Flächen, auf denen manche Verfahren nicht die ganze Fläche abdecken, wäre es gut, dass der Roboter dies weiß und dort nicht vergeblich nach einem Signal sucht. Wie in Kapitel 2 bereits angesprochen wurde, können neue Objekte hierbei eingesetzt werden.
Das Objekt eines Orientierungspunktes speichert 4 Punkte, die ein Feld aufspannen, in dem

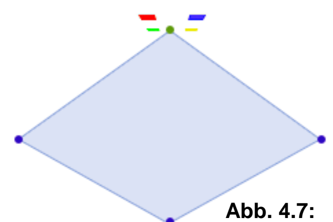


Abb. 4.7: Kartenobjekt eines Orientierungspunktes

eine Positionskontrolle möglich ist. Da die Genauigkeit bei zunehmender Entfernung bzw. bei schrägerem Blickwinkel abnimmt, wird so dem Roboter auch mitgeteilt, wo eine Positionskontrolle sinnvoll ist.

4.4 Installation

Absolute Positionsmessungen oder Positionskontrollen erfordern eine Eintragung in der Karte. Bei der Positionsbestimmung mit Hilfe der Orientierungspunkte ermittelt die Kamera zuerst nur die relative Position. Um jedoch den absoluten Standort in der Karte zu bestimmen, muss natürlich die Position des Orientierungspunktes bekannt sein, die bei der Installation eingespeichert wird. Um diese Installation für den Benutzer so einfach wie möglich zu gestalten, kamen wir auf die Idee, den Roboter diese Aufgabe selbst lösen zu lassen. Der Benutzer muss dem Roboter nur die grobe Position vorgeben, an der dieser dann den Rest erledigt. Das gesamte Verfahren ähnelt der automatischen Speicherung von Hindernissen. Im folgenden Kapitel soll auf die genauen Abläufe näher eingegangen werden.

5. Abläufe und Navigationsmodi

In den letzten drei Kapiteln wurden die Funktionsweisen der wichtigsten Teilsysteme erklärt. Nun geht es darum, wie diese logisch vernetzt sind und zusammen funktionieren.

5.1 Aufgabenmodus

Das gesamte System ist im Prinzip in drei Modi unterteilt, von denen das Verhalten des Roboters stark abhängt. Der Aufgabenmodus ist der erste und wichtigste Modus. Er wird aktiviert, wenn der Roboter ein Ziel vorgegeben bekommt, das er ansteuern soll.

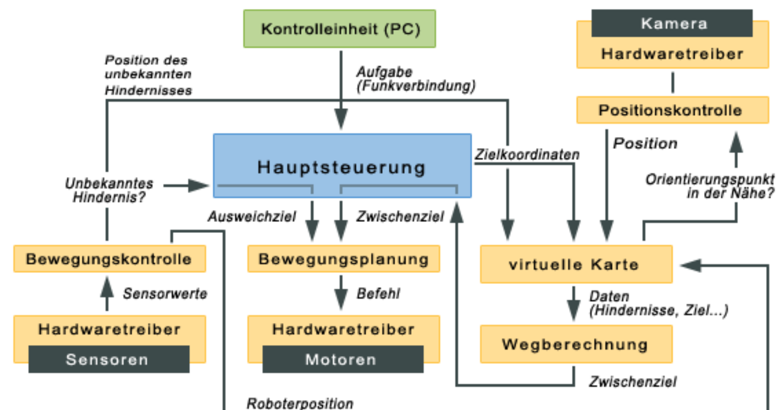


Abb. 5.1: Abläufe im Aufgabenmodus

Die Zielposition wird in die Karte eingespeichert, anschließend startet die Wegberechnung, das erste Zwischenziel anzugeben. Diese Koordinate wird an die Bewegungsplanung weitergereicht, die schließlich den Roboter in Bewegung setzt und den Punkt ansteuert. Parallel dazu werden alle Sensoren abgefragt und die aktuelle Roboterposition ermittelt. Mit diesem Wert wird wieder eine neue Wegberechnung gestartet, deren Ergebnis wiederum in eine neuen Bewegungsplanung eingeht. So wird der Kurs des Roboters laufend überprüft. Mit den Hindernissensoren wird nach unbekannten Hindernissen gesucht, die nicht in der Karte eingespeichert sind und deshalb den Weg versperren. In so einem Fall wird das Zwischenziel für kurze Zeit ignoriert und von der Hauptsteuerung durch ein neues ersetzt. Dabei versuchen die Sensoren, eine freie Lücke auszumachen, durch die dann gefahren wird (*Abb. 5.2*). So probiert der Roboter, dem unbekannten Hindernis auszuweichen. Die Position des unbekannten Hindernisses wird in der Karte als solches abgespeichert. Das Ganze ist natürlich ein kritischer Moment und das System hat mit großen Hindernissen, die schwer zu überschauen sind, Probleme. Aus diesem Grund gibt es den Erkundungsmodus, der weiter unten beschrieben wird.

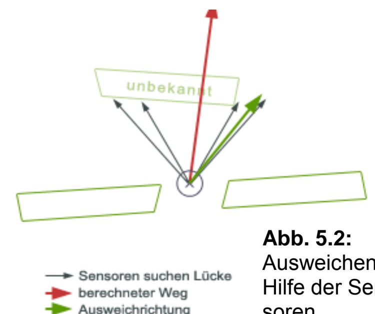


Abb. 5.2: Ausweichen mit Hilfe der Sensoren

Während der Fahrt wird in der Karte nach eingezeichneten (optischen) Orientierungspunkten gesucht, die die Kamera theoretisch sehen müsste. Die Koordinaten eines gefundenen Punktes werden an die Positionskontrolle weitergegeben, die nun mit der Kamera an dieser Stelle nach dem Punkt sucht. Wurde dieser gefunden, hält der Roboter kurz an, damit die Positionsbestimmung durchgeführt werden kann. Die neue Roboterposition wird in der Karte gespeichert und der Roboter fährt weiter. Die relative Positionsbestimmung der Bewegungskontrolle baut nun auf dem neuen Wert auf. Diese Abfolge (Suchen des Orientierungspunktes in der Karte, danach Suchen mit der Kamera an der vermuteten Stelle) ist ganz entscheidend, da an dem Punkt selber optisch nicht erkannt werden kann, um welchen es sich handelt. Aus diesem Grund muss ein ausreichender Abstand zwischen einzelnen Orientierungspunkten vorhanden sein, damit es nicht zu Verwechslungen kommen kann.

5.2 Erkundungsmodus

Das Einspeichern von neuen Hindernissen beim Ansteuern eines Ziels ist sehr umständlich. Einerseits soll der Roboter möglichst schnell zum Ziel gelangen, andererseits muss ein Hindernis in Ruhe vollständig umfahren werden, damit es später korrekt in die Wegberechnung einbezogen werden kann. Aus diesem Grund beschränkt sich der Aufgabenmodus beim Kontakt mit einem neuen

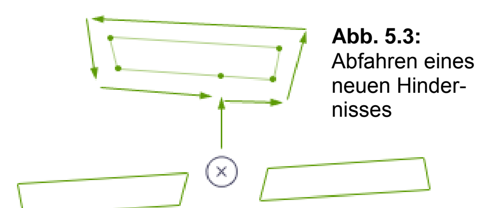


Abb. 5.3: Abfahren eines neuen Hindernisses

Hindernis auf das Eintragen eines Kontrollpunktes, ohne das Hindernis vollständig aufzuzeichnen. Befindet sich das System später wieder im Erkundungsmodus, werden diese Kontrollpunkte angesteuert, wo das Hindernis dann korrekt in die Karte eingezeichnet wird. Der Roboter gibt sich beim Starten des Modus selbst die Aufgabe, einen Kontrollpunkt anzusteuern. Intern wird also wieder in den Aufgabenmodus gewechselt. Ist der Roboter an dem Punkt angekommen, bei dem das neue Hindernis gesichtet wurde, dreht er sich und folgt der Wand mit Hilfe eines IR-Sensors. Dabei versucht er stets, genau parallel zur Wand zu fahren, sodass die Entfernung zu ihr immer konstant bleibt. Ändert sich jedoch die Entfernung, wird der Kurs entsprechend korrigiert. Diese Stellen, die der Sensor als Ecken identifiziert, werden in der Karte als Points gespeichert. Natürlich muss hierbei eine gewisse Toleranz vorhanden sein, damit erstens kleine Fahr-Ungenauigkeiten des Roboters nicht zu Fehlern führen und außerdem die Datenmenge auf dem Mindestmaß gehalten wird.

Ist das Hindernis schließlich vollständig abgespeichert, wird der nächste Kontrollpunkt angesteuert. Auf diese Weise werden neu auftretende Hindernisse mit in die Karte aufgenommen, ohne dass dieser Vorgang den Aufgabenmodus aufhält. Wie gut das System am Ende mit neuen Hindernissen umgehen kann, hängt von der Programmierung des Ausweichverhaltens sowie von der Sensorik, auf die jenes zugreifen kann, ab.

5.3 Administrationsmodus

Der Administrationsmodus ist der Standardmodus, in dem sich der Roboter befindet, wenn er in der Homebase steht und keine Aufgabe ausgeführt wird. Neben einer Zielvorgabe, die das System in den Aufgabenmodus versetzten würde, können ihn auch direkte Anweisungen erreichen. Mit Hilfe eines PC-Kontrollprogramms (› siehe Kapitel 6.6) kann zum Beispiel die Karte vom Roboter runter geladen, auf dem Bildschirm grafisch angezeigt und dort manuell bearbeitet werden. Dies ist in Umgebungen mit komplizierten oder sehr großen Hindernissen hilfreich, da diese von der automatischen Erkennung manchmal nicht korrekt erfasst werden.

Auch die Installation von Positionskontrollsystemen findet im Administrationsmodus statt, der für diese Prozedur eine Art Assistent bietet. Um beispielsweise einen Orientierungspunkt manuell in die Karte einfügen zu können, muss die relative Position zur Homebase bekannt sein, die bei größeren Entfernungen relativ schwer zu ermitteln ist. Damit der Anwender aber möglichst wenig mit der Installation zu tun hat, soll der Roboter diese Aufgabe selbst übernehmen.
Nachdem beispielsweise ein Orientierungspunkt montiert wurde, wird dem Roboter mitgeteilt, in welcher ungefähren Umgebung er zu finden ist. Intern wird dann in den Aufgabenmodus gewechselt und der angegebene Bereich angesteuert. Dort beginnt der Roboter, mit der Kamera den Punkt zu suchen. Wurde er gefunden, wird seine relative Position zum Roboter berechnet und über die absolute Roboterposition in die Karte eingetragen. Da die Position des Punktes in der Karte einer ungenauen Wegmessung des Roboters zu Grunde liegt, muss dieser versuchen, diese Ungenauigkeit möglichst klein zu halten und so oft wie möglich eine Positionskontrolle (z.B. über bereits vorhandene Orientierungspunkte) durchzuführen.

Der Administrationsmodus ermöglicht außerdem ein direktes Fernsteuern des Roboters, was bei der Fehlersuche sehr hilfreich sein kann.

6. Technische Umsetzung

6.1 Aufbau des Roboters

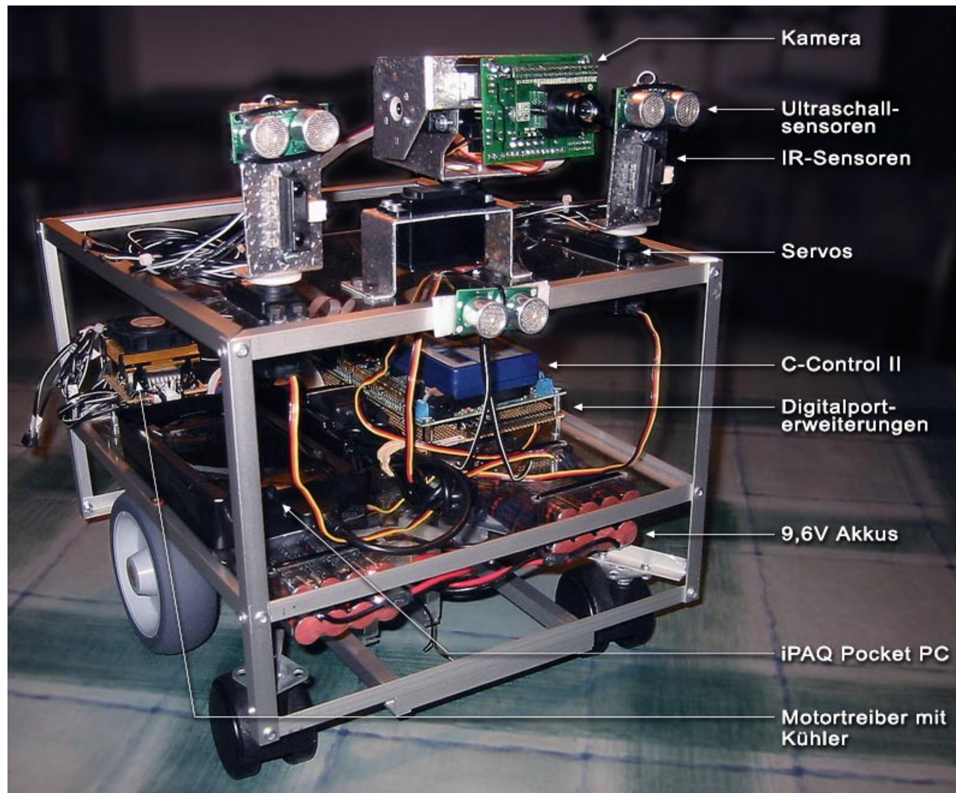


Abb. 6.1

Der Roboter besitzt vier Rädern, von denen die beiden Antriebsräder Vollgummireifen sind, mit denen er Vorzugsweise im Inneren von Gebäudekomplexen fahren kann. Diese haben zwar kein Profil, sind aber trotzdem auch für Außeneinsatzzwecke geeignet. Die beiden anderen Räder, die lediglich Stützfunktion haben, sind Rollen eines Schreibtischstuhls. Die Lenkung erfolgt über unterschiedliche Drehzahlen der Antriebsräder. Dies hat auch den Vorteil, dass auf der Stelle gewendet werden kann, indem die Räder entgegengesetzt gedreht werden.

Als Antrieb dienen vier Gleichstromgetriebemotoren. Wir verwenden für die Antriebsachsen je zwei Motoren, um eine größere Kraftübertragung zu gewährleisten, damit der Roboter unter dem Gewicht der anderen Bauteile (auch bei weiterem Ausbau des Roboters) auch bei größeren Unebenheiten ohne Probleme fahren kann. Durch ein Getriebe mit einer Übersetzung von 1:2 erreicht der Roboter eine Höchstgeschwindigkeit von ungefähr einem Meter pro Sekunde. Zur Geschwindigkeitsregelung werden Motortreiber vom Typ L298N verwendet (siehe 6.2). Diese werden mit einem Lüfter gekühlt, da die ICs besonders bei Belastung schnell heiß werden.

Die gesamte Stromversorgung wird von vier Akkupacks (je NiMH mit je 9,6V – 1100 mAh) geliefert. Dabei sind je zwei Akkus in Reihe geschaltet, woraus eine Betriebsspannung von 19,2 Volt folgt. Die beiden Packs sind dann parallel zueinander geschaltet, so dass wir eine Gesamtleistung von 2,2 Amperestunden besitzen.

Der Roboter verfügt über mehrere Sensoren mit unterschiedlichsten Aufgaben. Die Hinderniserkennung erfolgt mittels drei Ultraschallsensoren (SFR04 von Devantech) und zwei Infrarotsensoren (GP2D12 von Sharp). Während ein Ultraschallsensor direkt nach vorne gerichtet und zentriert befestigt ist, sind je ein Ultraschallmodul und ein Infrarotsensor zu einem Sensorblock zusammengefasst. Beide Sensorblöcke sind beweglich mit einem Drehradius von mehr als 90° angebracht, sodass eine Abtastung nach vorne und zur Seite möglich ist. Der feststehende Ultraschallsensor ist nach vorne gerichtet, um frühzeitig Hindernisse zu erkennen, die zu Kollisionen führen könnten, auch wenn die beweglichen Sensoren gerade nicht nach vorne blicken. Die Ultraschallsensoren sind für die Entfernungsmessung im fernen Bereich zuständig (bis zu drei Meter) und die Infrarotsensoren übernehmen den nahen Bereich von 80 cm – 10 cm. Für den extremen Nahbereich sind an den Kanten des Roboters kleine Tastsensoren angebracht, die bei Kontakt den Roboter sofort stoppen, um Kollisionen mit schwerwiegenden Schäden zu vermeiden. Wir verwenden zwei verschiedene Arten von Abstandmessungssensoren auch aus dem Grund, dass die Infrarotsensoren zwar eine feinere Auflösung besitzen, aber im Gegensatz zu den Ultraschallsensoren keine Glas- oder Spiegelflächen und lichtabsorbierende Flächen erkennen können. Ultraschallsensoren haben jedoch wiederum den Nachteil, dass sie einen zu großen Streuwinkel der Schallwellen besitzen, so dass punktgenaue Messungen nicht möglich sind.

Die Sensorblöcke werden durch Servos (HS-300) horizontal gedreht. Die Servos können etwas mehr als 180 Grad drehen, sodass die Seiten und der vordere Bereich voll abgetastet werden. Die Kamera (CMUcam) ist zentriert am vorderen Rand auf dem Roboter angebracht. Diese lässt sich frei in nahezu jede Richtung bewegen. Dies wird durch eine spezielle Pan-/Tilt-Halterung erreicht, die ebenfalls durch Servos bewegt wird.

Die optischen Wegsensoren sind auf der Höhe der Achsen, neben den Rädern angebracht. Um zu verhindern, dass bei unebenen Untergründen die Sensoren auf dem Boden schleifen, wird durch zusätzliche Linsen der Mindestabstand vom Boden vergrößert.

Die Hauptsteuereinheit, die C-Control II, ist zentral auf dem Roboter untergebracht, damit jedes System ohne große Schwierigkeiten erreicht werden kann. Zur Erweiterung der digitalen Ports der C-Control ist unter dem Application-Board eine Porterweiterungsplatine angebracht, die uns 39 zusätzliche Ports, einzeln über den I²C-Bus ansteuerbar, liefert.

Der Pocket PC ist mit der Plastik-Einlage aus der Originalpackung befestigt, die etwas zugeschnitten wurde.

6.2 Sekundäre Elektronik

6.2.1 Servo-Ansteuerung

Die Servos werden durch den IC „SD20“ angesteuert, der die Befehle für die Ausgabe der Impulse über die I²C-Schnittstelle empfängt. Bei Servos wird eine absolute Position angegeben, auf die sich der Motor drehen soll. Das ist ein großer Unterschied zu Schrittmotoren, die sich jeweils bei einem Impuls nur (wie der Name schon sagt) um einen „Schritt“ drehen.
Dem IC (› Schaltung siehe Kapitel 8.4.1) wird ein Wert zwischen 0 und 255 übergeben, wobei jeder Wert eine bestimmte Position angibt. Der SD20 wandelt diesen Wert nun in einen Impuls mit einer Breite zwischen 1ms und 2ms um, wie es den Spezifikationen von Servos entspricht. Auf diese Weise lassen sich Servos gewöhnlich nur um ca. 90 Grad drehen. Um den Drehbereich zu erweitern, müssen Impulse gesendet werden, die unter 1ms bzw. über 2ms liegen. Dafür kann man den SD20 in den so genannten „Expanded Mode“ schalten. In diesem Modus kann er dementsprechende Impulse erzeugen.
Einen Nachteil hat diese Betriebsart jedoch, denn bei Verwendung besteht die Gefahr, dass der Servo in die Endstellung fährt und dabei unter Umständen beschädigt wird. Deshalb müssen vorher experimentell die mechanischen Obergrenzen der Servos herausgefunden und im Programm berücksichtigt werden.

6.2.2 Ultraschallmodule SRF04

Das Ultraschallmodul SRF04 (*Abb. 6.2*), von einem Triggersignal gestartet, sendet eine Schallwelle aus und wartet, bis das Echo wieder empfangen wird. Um eine Messung zu starten, wird ein Impuls an den Triggereingang gelegt. Der Wandler wird von der Ablaufsteuerung getaktet (ein Signal wird ausgesendet) und der Echo-Ausgang des Moduls auf „High“ gelegt. Das erste hereinkommende Echo schaltet den Echo-Ausgang wieder auf „Low“, sodass ein direkt zur Entfernung des Objektes proportionaler Impuls entsteht.

Abb. 6.2

Manche Controller, wie etwa die C-Control II, sind leider zu langsam um die Länge dieses Ausgangsimpulses (dieser hat eine Lange von 800µs – 18ms) direkt zu messen. Mit einer Schaltung (› siehe Kapitel 8.4.2) kann man aber auch mit einem solchen Controller die genaue Entfernung bestimmen. Die Impulslänge wird hierbei in einen Spannungswert umgewandelt, und kann dann mit einem AD-Port ausgelesen werden.

6.2.3 Infrarotsensoren GP2D12

Die IR-Sensoren GP2D12 (*Abb 6.3*) sind in vieler Hinsicht einfacher zu handhaben. So liefern sie direkt ein analoges Ausgangssignal und es muss kein aufwändiges Taktsignal generiert werden. Bei dem Analogsignal entsprechen 10cm etwa 2,6V und 80cm etwa 0,4V. Um aus diesem Signal nun einen genauen Abstandswert zu erhalten, wäre eine Look-Up-Tabelle sicherlich das einfachste, jedoch auch das programmiertechnisch aufwändigste. So haben wir experimentell eine Formel zur Umrechnung ermittelt.

Abb. 6.3

Auf diese Weise erhielten wir zum Beispiel für den einen Sensor folgende Formel:

$$d = \frac{5915}{X - 13} - 2$$

Hierbei bezeichnet X den Spannungswert, der am AD-Port gemessen (als 10-bit Integer-Wert) wird und d die Entfernung in cm.

6.2.4 Liniensensor

Der Liniensensor besteht aus drei Fotowiderständen. Diese sind in einer Reihe angebracht, horizontal zur Fahrtrichtung. In Verbindung mit einem Spannungsteiler kann über den einzelnen Fotowiderständen die Spannung gemessen werden, die proportional zur gemessenen Helligkeit ist. Mehrere LEDs beleuchten den Untergrund, sodass auch ohne Umgebungslicht eine Bodenlinie gut erkannt werden kann. Die Fotowiderstände sind an jeweils einem AD-Wandler der C-Control angeschlossen. Wir haben auf eine elektronische Schaltung, die die Spannungswerte in ein digitales Signal umwandelt, verzichtet und fragen die Helligkeitswerte direkt ab. Somit können Linien besser erkannt werden. Beispielsweise ist es nicht zwingend, dass eine Bodenlinie eine bestimmte Farbe hat. Vielmehr ist der Farbunterschied ausschlaggebend. Somit kann ebenso eine dunkle Linie auf hellem Boden wie eine helle Linie auf dunklem Boden verfolgt werden.

6.2.5 Motortreiber

Wir verwenden Motortreiber vom Typ L298N. Diese ICs können zwei Motoren mit je 2 Ampere oder einen mit 4 Ampere betreiben. Für die letztere Betriebsart müssen die In- und Outputs jeweils parallel geschaltet werden (› Schaltung siehe Kapitel 8.4.3).
Die Drehrichtung wird mit den beiden Input-Anschlüssen eingestellt. Zum Drehen in die eine Richtung muss der eine Port auf low, der andere auf high gesetzt werden. Die andere Richtung erreicht man mit dem jeweils anderen Pegel. Um den Motor zu bremsen, werden beide Ports auf denselben Pegel gesetzt.
Die Geschwindigkeitsregelung erfolgt einfach über das ständige Ein- und Ausschalten des Motors, was man über den Enable-Pin bewerkstelligen kann. Wird er mit einem PLM-Port verbunden, so lässt sich die Geschwindigkeit gut regeln.

6.2.6 Funkmodul TRX 433

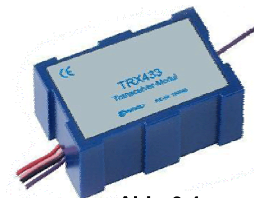


Abb. 6.4

Zur Kommunikation zwischen C-Control und dem Computer an der Homebase verwenden wir zwei 433 MHz Transceiver (Abb. 6.3). Mit diesen Transceiver-Modulen kann eine bidirektionale Funkstrecke aufgebaut werden, die unter optimalen Umständen bis zu 500m Reichweite ermöglicht. Da der TRX 433 den größten Teil der nötigen Formatierung, Codierung und Zeitüberwachung übernimmt, beschränkt sich der Aufwand, Daten zu senden oder zu empfangen, auf wenige kleine Programm-Routinen.
Der Steuercomputer kommuniziert mit dem TRX 433 über drei bidirektionale Ports.

6.3 Prozessverteilung und programmiertechnische Implementierung

Die Verschaltung der Hauptelektronik ist in der rechten Abbildung dargestellt und soll im Folgenden näher erläutert werden.

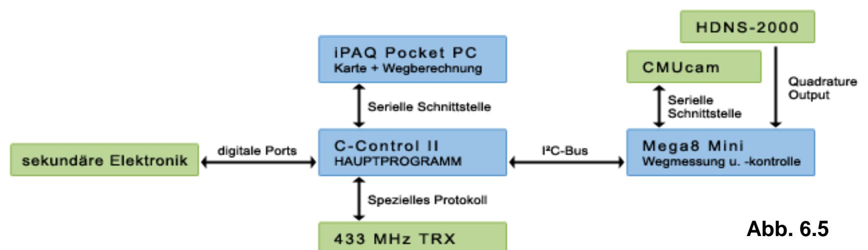


Abb. 6.5

6.3.1 Conrad C-Control II

Die C-Control II (CC2) übernimmt die Hauptsteuerung des Roboters und ist allen anderen Baugruppen übergeordnet. Die grundlegende Verhaltensweise des Roboters sowie die Ansteuerung der sekundären Elektronik (Sensoren und Motoren, › siehe Kapitel 6.4) sind hier implementiert. Über ein Funkmodul ist der Roboter mit dem PC vernetzt. Eingehende Signale werden weiterverarbeitet oder weitergegeben, sodass im Administrationsmodus (› siehe Kapitel 6.3) auch auf Untersysteme zugegriffen werden kann.

6.3.2 iPAQ Pocket PC

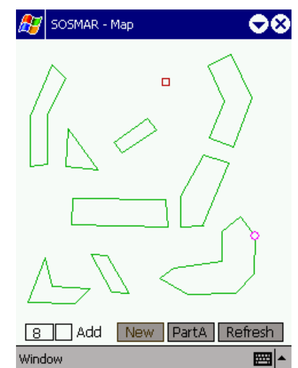


Abb. 6.6

Da die Wegberechnung meistens das rechenaufwendigste Modul bei mobilen Robotern ist, wird diese oft auf einem PC implementiert, der dann per Funk mit dem Roboter kommuniziert. Da unser Roboter aber möglichst unabhängig sein und auch in größeren Gebäuden funktionieren sollte, kam dies für uns nicht in Frage. Notebooks als mobile Rechner sind auch etwas zu groß, aber Pocket PCs, die sich natürlich auch selbst programmieren lassen, scheinen genau die richtige Lösung zu sein. Die Firma AppForge bietet eine Entwicklungsumgebung für Visual Basic an (‚MobileVB'), die unseren Ansprüchen (Klassenmodule, User Defined Types etc.) genügte. (Es gibt auch ein entsprechendes Entwicklungspacket von Microsoft namens ‚eMbedded Visual Basic', allerdings werden die genannten Funktionen davon noch nicht unterstützt) Die am Computer mit Visual Basic entwickelte Wegberechnung konnten wir so ohne größere Probleme auf dem Pocket PC implementieren.
Der Pocket PC kommuniziert mit der CC2 über die serielle Schnittstelle und fungiert praktisch als Slave. Er empfängt Befehle von der CC2 und gibt entsprechende Daten zurück. So können von der CC2 alle Kartenfunktionen gesteuert werden.

Die Programmierung der Karte basiert auf Objektorientierung. Jedes Hindernis ist eine Instanz des Klassenmoduls „tObstacle“, das ein Datenfeld vom Typ Point enthält, in dem die Daten der einzelnen Hindernispunkte gespeichert werden. Außerdem enthält die Klasse grundlegende Funktionen zur Kartenverwaltung.

```
Private Type Point
    x As Long
    Y As Long
    DistanceA As Long
    DistanceB As Long
    DistanceX As Long
End Type

Private Points() As Point
Public Precision As Long
```

Beim Hinzufügen eines neuen Hindernisses wird eine neue Instanz der Klassse erstellt und einem neuen Element des Datenfeldes „Hindernisse“ zugeordnet.

```
Public Type ObstList
    list() As tObstacle
End Type

Private Hindernisse As ObstList
```

(Bei der Portierung auf den PDA traten unerwartete Probleme auf. Beispielsweise unterstützt MobileVB nicht die Übergabe von dynamischen Datenfeldern, weshalb die etwas seltsame Konstruktion mit dem UDT „ObstList“ nötig war.)

6.3.3 ATmega8 Mini

Die CC2 wird von einem kleineren Mikrocontroller unterstützt (Mega8), der die Quadrature-Signale des Maussensors (HDNS-2000) auswertet und die Steuerung der Kamera (CMUcam) übernimmt. Die Daten werden der CC2 per I²C-Bus in Registern zur Verfügung gestellt.
Die gesamte Positionskontrolle, die die Kamera verwendet, ist auf dem Mega8 implementiert. Es bekommt von der CC2 nur Befehle und die nötigen Daten wie zum Beispiel die Position des nächsten Orientierungspunktes, das Suchen des Punktes und die anschließende Positionsbestimmung werden eigenständig ausgeführt.
Die Bewegung der Kamera, die über zwei Servos gesteuert wird, ist etwas umständlich. Die Servos, von denen insgesamt 4 auf dem Roboter verwendet werden, werden alle über einen Ansteuerungs-IC (› siehe 6.4) geregelt. Dieser IC sitzt wie das Mega8 am I²C-Bus. Leider kann eine CC2 nur als Master arbeiten, was bedeutet, dass das Mega8 nur Slave sein darf. Dies bedeutet, dass das Mega8 nicht direkt auf den Servo-IC zugreifen kann, um die Kamera zu bewegen. Aus diesem Grunde ist ein kleiner Umweg nötig. Das Mega8 setzt, um die Kamera zu bewegen, ein Register mit der gewünschten Position, welches die CC2 abfragt und dementsprechend den IC ansteuert.
Wie in Kapitel 4.2.1 beschrieben wurde, besteht ein Orientierungspunkt aus 4 Farben. Bei der Suche nach einem Orientierungspunkt beschränkt sich der Roboter zunächst auf eine Farbe. Wurde diese gefunden, wird die Kamera so ausgerichtet, dass alle 4 Farben vollständig im Kamerabild zu sehen sind. Aus den Positionen der Farbfelder wird dann die Eigenposition berechnet. Diese Prozedur ist leider nur möglich während der Roboter sich nicht bewegt, da die Kamera nicht mehrere Operationen an einem Bild durchführen kann. Mit anderen Worten werden die Positionen der vier Farben nicht in demselben Frame ermittelt, sondern in vier unterschiedlichen. Würde sich die Kamera demnach mit dem Roboter bewegen, wären die Ergebnisse nicht zu gebrauchen. Es gibt jedoch bereits eine neue Version der CMUcam, mit der eine Positionsbestimmung ohne Anzuhalten theoretisch möglich wäre.

6.4 Kontrollsoftware

Robotersysteme, die in Zukunft Einzug in unser alltägliches Leben halten werden, müssen vor allem einen großen Spielraum für Erweiterungen aufweisen. Wie schon in den vorigen Kapiteln angesprochen wurde, bietet unsere Kartenverwaltung eine Fülle von Erweiterungsmöglichkeiten. Um diese jedoch mit externen Systemen verknüpfen zu können, ist eine Software nötig, die eine komfortable Bedienoberfläche sowie eine offene Schnittstelle für Programme anderer Hersteller zur Verfügung stellt. So eine Software kann man sich praktisch wie ein Betriebssystem vorstellen, das eine Verwaltung von Daten und eine Kommunikation zwischen verschiedenen Prozessen ermöglicht. Auf diese Weise können zwei verschiedene Robotersysteme Daten austauschen, obwohl sie gar nicht explizit für den gemeinsamen Betrieb entwickelt wurden. Die Leistungsfähigkeit, die dabei entsteht, ist leicht zu erkennen. Mehrere mobile Roboter können ihre Karten abgleichen und so Informationen über Regionen erhalten, in denen sie sich selbst noch nie befanden. Auch das bereits angesprochene Beispiel der elektrischen Tür ist einfach realisierbar.
Um die Integrationsfähigkeit unseres Systems vorstellen zu können, haben wir eine Software entwickelt, die entsprechende Funktionen bietet.

Damit die Software nicht nur in Verbindung mit unserer speziellen Hardware funktioniert, basiert sie auf einer Plugin-Funktionalität. Hardware-spezifische Plugins (Treiber) können so in das Programm eingebunden werden und mit ihm über eine Schnittstelle kommunizieren. Die Software ist dadurch von der speziellen Hardware

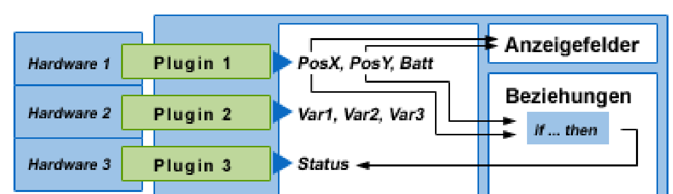


Abb. 6.7

unabhängig und könnte praktisch mit jedem Mikrocontroller kommunizieren, für den ein entsprechendes Plugin vorhanden ist. Beim Initialisieren kann jedes Plugin eine beliebige Anzahl von Variablen registrieren, wie zum Beispiel X- und Y-Position oder Batteriespannung. Diese Variablen sind die einzige Möglichkeit, Daten zwischen Plugin und Kontrollsoftware auszutauschen. Die Software hat jetzt mit anderen Worten nur Kenntnis über die Position und Batteriespannung. Die Datenübertragung zwischen PC und Roboter liegt ganz im Aufgabenbereich des Plugins und ist somit nicht durch die Kontrollsoftware beschränkt und kann genau auf die anzusteuernde Hardware zugeschnitten sein. Das Plugin sendet in einem selbst gewählten Intervall die aktuellen Werte zur Kontrollsoftware.

Die Software bietet einerseits die Möglichkeit, die Werte der Variablen anzuzeigen und zu editieren. Die Anzeigefelder sind übrigens selbst wieder durch Plugins erweiterbar, sodass für eine besondere Variable (zum Beispiel eines Neigungssensors) ein spezielles Feld eingebunden und verwendet werden kann. Diese Möglichkeit ist sehr hilfreich bei der Fehlerdiagnose sowie bei der Kontrolle des Roboters.
Die zweite und wichtigere Funktion ermöglicht das Programmieren von Beziehungen zwischen zwei oder mehreren Variablen. *Abbildung 6.1* zeigt, wie das Beispiel der elektrischen Tür aussehen könnte. „Hardware 1" ist der Roboter, „Hardware 3" die Tür. Die Kontrollsoftware überprüft laufend die Position des Roboters. Befindet er sich in einem festgelegten Bereich, wird die „Status"-Eigenschaft der Tür auf „offen" gesetzt und der Roboter kann passieren.
Die Programmierung der Beziehungen kann in einem Plugin stattfinden. Plugins können nicht nur ihre Variablen der Kontrollsoftware zur Verfügung stellen, sondern auch auf die Variablen anderer Plugins zugreifen. Dies ist aber nur möglich, wenn dem Programmierer des einen Plugins das andere bekannt ist, was selten der Fall sein wird. Aus diesem Grund besitzt die Kontrollsoftware ein Quellcode-Fenster, in dem diese Beziehungen mit der Scriptsprache VBScript selbst programmiert werden können.
Jede Variable bekommt von der Kontrollsoftware ein Ereignis zugeordnet, das auftritt, wenn sich der Wert der Variable ändert. Dem Quellcode können nun Ereignisprozeduren hinzugefügt werden, die beim Auftreten des Ereignisses aufgerufen werden. Der Quellcode für unser Beispiel:

```
Sub Plugin1_PosX_Change()
  PosX = CDK.getData('Plugin 1', 'PosX')
  PosY = CDK.getData('Plugin 1', 'PosY')
  If PosX > 1500 And PosX < 1700 And _
     PosY > 900 And PosY < 1100 Then
    CDK.setData('Plugin 3', 'Status', 'Open')
  Else
    CDK.setData('Plugin 3', 'Status', 'Close')
  End If
End Sub

Sub Plugin1_PosY_Change()
  Call Plugin1_PosX_Change
End Sub
```

Das Beispiel zeigt eine sehr einfache Anwendung. Die Wegberechnung des Roboters ist nicht an dem Öffnen der Tür beteiligt, was gewisse Nachteile hat. Die Tür wird beispielsweise auch dann geöffnet, wenn der Roboter zwar an der Tür vorbei, aber nicht hindurch fahren möchte. Aus diesem Grund wäre auch eine Plugin-Funktionalität auf dem Roboter sinnvoll und durchaus möglich (› siehe Kapitel 3.4), ist im Moment jedoch noch nicht vorhanden.

Die Gestaltung der Benutzeroberfläche ist ebenfalls durch Plugins möglich. Bereits angesprochene Anzeigefelder können programmiert werden, über die ein Darstellen und Editieren der Variablen möglich ist. Die Verknüpfung von Anzeigefeld und Variable geschieht einfach per Drag and Drop. Es können aber auch ganz neue Fenster programmiert werden, die komplizierte Eingaben ermöglichen. Ein Beispiel dafür wäre ein Karten-Plugin, über das die Karte des Roboters angezeigt und editiert werden kann.

Die Plugin-Funktionalität wird über ActiveX-DLLs realisiert. Jedes Plugin wird dabei als DLL kompiliert und in einem vorgesehenen Verzeichnis der Kontrollsoftware abgespeichert. Beim Starten der Software wird die DLL registriert und eingebunden, wobei ihr ein Objekt übergeben wird, über das es auf die Funktionen der Software zugreifen kann. Dieses Objekt (wir nennen es einfach ‚door') ist bildlich vorgestellt die Tür zu einem Raum, in dem sich alle Plugins treffen, um ihre Daten auszutauschen. Das Objekt beinhaltet mehrere Funktionen, die neben dem Lesen und Schreiben von Daten auch das Anzeigen von Meldungen, Steuern des Drag and Drop-Vorgangs etc. beinhaltet. Eine komplette Liste finden Sie im Anhang.
Das door-Objekt ist auch in dem Quellcode zur Programmierung der Beziehungen vorhanden. Im obigen Beispiel ist beispielsweise der Aufruf „`CDK.setData`" zu finden.

Mit so einer Software, die einen Schnittstellenstandard definiert, wäre es möglich, Geräte verschiedener Anbieter, die alle einen entsprechenden Treiber mitliefern, zu verknüpfen.

7. Zusammenfassung

7.1 Modell für ein mobiles Robotersystem

Unser System umreißt die Grundfunktionen, mit denen mobile Robotersysteme in Zukunft in verschiedensten Anwendungsgebieten eingesetzt werden könnten.

- **Kartenverwaltung**, die es dem Roboter ermöglicht, Informationen über seine Umwelt zu speichern und sie im Kontext seiner aktuellen Position bzw. seinem Ziel wieder abzurufen. Das Verfahren sollte um neue Kartenobjekte erweiterbar sein, um den Handlungsspielraum des Roboters nach Bedarf erweitern zu können. Die Speicheranforderungen sollten dabei so gering wie möglich bleiben. › siehe Kapitel 2
- **Wegberechnung**, die eine zielgerichtete Navigation mit Hilfe der Karte erlaubt. Da sie mit der Karte eng verknüpft ist, muss auch sie erweiterbar sein. › siehe Kapitel 3
- **Positionskontrollen**, über die der Roboter bei Bedarf die Eigenposition abgleichen kann. Sie können an nahezu beliebigen Plätzen positioniert werden und müssen sich vom Zugriffsradius nicht überschneiden. › siehe Kapitel 4.2
- Hardware zur **Positionsbestimmung**, die keine externen Geräte benötigt und direkt auf dem Roboter installiert ist. Dadurch ist er vollkommen unabhängig von weisenden Signalen und kann somit auch in „Löchern" der Positionskontrollen navigieren. Damit diese relative Positionsbestimmung nicht zu ungenau wird, müssen Sensoren gewählt werden, die den Weg möglichst direkt (und nicht indirekt, beispielsweise per Umdrehungsmessung der Räder) abnehmen. › siehe Kapitel 4.1
- Sensorik zur **Kollisionsvermeidung**, mit der Hindernisse erkannt und in die Karte eingespeichert werden können. › siehe Kapitel 6.2
- Ein **Kontrollserver**, über den der Roboter mit anderen elektronischen Geräten, zum Beispiel auch mit anderen Robotern, kommunizieren kann. › siehe Kapitel 6.4
- **Selbstorganisation**, wodurch eine aufwendige Installation des Systems entfällt. Hindernisse werden beispielsweise vom Roboter selbstständig in die Karte eingetragen, Positionskontrollen automatisch erkannt etc. › siehe Kapitel 5.2
- **Administrationsmöglichkeit**, mit der trotz aller Automation individuelle Konfigurationen vorgenommen werden können.

7.2 Form des Roboters

In Verbindung mit der Positionsbestimmung fiel uns auf, dass die von uns gewählte Roboterform nicht die günstigste ist, obwohl sie sehr verbreitet ist. Besser geeignet sind so genannte Synchro-Drives. Sie besitzen in der Regel vier Räder, die alle angetrieben und synchronisiert gelenkt werden. Dies hat zur Folge, dass der Roboter seine Orientierung in eine Richtung nie ändert und statt zu Lenken praktisch einfach seitwärts weiterfährt. Eine Positionsbestimmung mit Hilfe eines optischen Sensors (› siehe Kapitel 4.1) wäre hier besonders einfach zu realisieren, da sich der Sensor selber nicht dreht und seine x- und y-Achsen denen der Karte entsprechen. Eine Umrechnung wie bei der Odometrie ist somit nicht nötig. Über einen zweiten Sensor, dessen x-Achse immer in Fahrtrichtung zeigt, könnten Abweichungen festgestellt werden, da sich der y-Wert bei einer perfekten Fahrt nicht verändern sollte. Synchro-Drives sind somit eigentlich bessere Roboterplattformen, wenn relative Wegmessungen durchgeführt werden. Aus diesem Grund haben wir vor, einen neuen Roboter zu konstruieren, um noch bessere Ergebnisse erzielen zu können.

7.3 Ausblick

Obwohl unsere gesamte Software zuverlässig funktioniert, bestehen noch Optimierungsmöglichkeiten. Da beispielsweise die Rechenzeit bei der Wegberechnung mit der Anzahl der Objekte in der Karte ansteigt, wäre eine Funktion wünschenswert, die momentan irrelevante Objekte ausblendet. In einer Karte, die mehrere Räume beinhaltet, könnten somit bei der Navigation innerhalb eines Raums die Hindernisse in den übrigen Räumen ignoriert werden, da sie in der Wegberechnung nicht von Belang sein werden.
Eine Ergänzung der Kartenfunktionen könnte es möglich machen, mehrere Etagen zu verwalten. Momentan ist eine Navigation nur auf einer Ebene möglich.

Die Anwendungsmöglichkeiten eines Robotersystems, ähnlich dem unseren, sind sehr reichhaltig. In unserem Projekt haben wir uns absichtlich nicht auf einen bestimmten Bereich spezialisiert, da wir gerade eine universelle Plattform entwickeln wollten. Mit dieser ist es nun möglich, einen ganz speziellen Aufgabenbereich zu automatisieren. Diese Einrichtung wird sich dadurch auszeichnen, dass sie eine einfache Konfiguration erlaubt und viele Ausbaufähigkeiten besitzt.

Anhang

8. Anhang

8.1 Begriffserklärung

In dieser Liste finden Sie die wichtigsten Begriffe, die wir in der Arbeit eingeführt haben.

Bodenlinien
…sind ein Positionskontrollverfahren und werden zum Beispiel in Verbindung mit Schneisen verwendet. Diese Linien auf dem Boden kann der Roboter mit Hilfe eines Sensors verfolgen.

DistanceA (oder A-Dist)
Diese Eigenschaft eines Hindernispunktes speichert seine Entfernung zum Endpoint A. B-Dist ist analog dazu die Entfernung zu Endpoint B.

DistanceX (oder X-Dist)
…gibt die Entfernung eines Endpoints zum Ziel an.

Endpoint (EP)
…ist ein Hindernispunkt. Jedes Hindernis besitzt zwei solcher Punkte, die von der Position des Ziels abhängen. Man kann sie sich als die äußersten Ecken des Hindernisses vorstellen.

Hindernispunkte
Jedes Hindernis besteht aus mehreren Punkten, die den unbefahrbaren Bereich abstecken.

Kollisionserkennung
Tastsensoren als „Bumper“ können Berührungen des Roboters mit Hindernissen erkennen. Dabei hat also bereits eine Kollision stattgefunden.

Kollisionsvermeidung
Mit spezieller Sensorik, die mit Infrarot oder Ultraschall arbeit, können Hindernisse erkannt werden, ohne dass eine Berührung des Roboters nötig ist.

Kontrollpunkt
Wenn der Roboter im Aufgabenmodus ein Hindernis erkennt, das noch nicht in der Karte eingespeichert ist, setzt er an dieser Stelle in der Karte einen Kontrollpunkt. Im Erkundungsmodus wird dieser Kontrollpunkt wieder angesteuert, um das neue Hindernis einzuspeichern.

Orientierungspunkte
…stellen ein Positionskontrollsystem dar. (› siehe Kapitel 4.2.1)

Positionsbestimmung
…ist eine Methode, mit der der Roboter seine Eigenposition laufend ermitteln kann. Dies funktioniert bei unserem System mit Hilfe zweier optischer Sensoren.

Positionskontrolle
Um die Fehler, die bei der relativen Positionsbestimmung auftreten, zeitweise zu korrigieren, gibt es Positionskontrollsysteme, mit denen die Position absolut bestimmt werden kann.

Schneisen
…stellen ein Positionskontrollsystem dar. (› siehe Kapitel 4.2.2)

Trace
Jedes Hindernis besitzt ein Trace, das die Fläche darstellt, in der das Ziel von dem Hindernis verdeckt wird. Stellt man sich das Ziel als Lampe vor, so wäre das Trace der Schatten des Hindernisses.

Wegberechnung
Algorithmen, die mit Hilfe der gespeicherten Karte einen kollisionsfreien Weg durch die Hindernisse berechnen können.

Zwischenziele
Wegepunkte, die die Wegberechnung zurückgibt.

8.2 Kontrollsoftware

Im Folgenden finden Sie eine genauere Beschreibung unserer Kontrollsoftware. Der Text besteht aus Auszügen mehrerer Statusberichte.

8.2.1 Aufbau

Die Software ermöglicht es in erster Linie, Variablen und damit Daten des Roboters auf dem PC sichtbar und editierbar zu machen, was erstens sehr hilfreich für die Entwicklung unseres Robotersystems ist und zweitens die Übermittlung von Befehlen zum Roboter ermöglicht, was ja trotz aller Autonomie des Roboters immer noch notwendig ist, um dem Roboter zum Beispiel den Zielpunkt zu nennen. Außerdem ist es mit der Software möglich, zwischen mehreren unabhängigen Hardware-Modulen Daten auszutauschen. Somit können beispielsweise zwei verschiedene Roboter miteinander vernetzt werden.
Die Software ist so aufgebaut, dass sie mit jeder beliebigen Roboterelektronik kommunizieren kann. Dies ermöglicht die Plugin- und Treiber-Funktionalität.
Um eine Hardware mit der Software anzusteuern, ist es lediglich nötig, einen Treiber zu programmieren, der Variablenwerte über die Kommunikationsschnittstelle (zum Beispiel der seriellen Schnittstelle) zur Software weiterleitet und umgekehrt Befehle und Daten über die Schnittstelle zum Roboter schickt. Somit ist die Software selber von der Hardware unabhängig und kann mit fast jedem erdenklichen Mikrocontroller verwendet werden.

8.2.2 Treiber und Plugins

Es wird zwischen Treibern und Plugins unterschieden. Treiber stellen die Verbindung zwischen der Soft- und Hardware dar. Plugins hingegen können alle anderen Aufgaben übernehmen, wie zum Beispiel eine Editierung der Roboterkarte. Auch die Anzeigefelder der Displaybar (› siehe nächste Seite) sind spezielle Plugins. Treiber und Plugins werden als ActiveX-DLLs kompiliert und beim Starten der Software registriert und in die Software eingebunden. Ihr Grundgerüst besteht aus einem Klassenmodul, das bestimmte Funktionen und Prozeduren haben muss, da diese von der Steuersoftware aus aufgerufen werden.

Für Plugins sind folgende nötig:

```
Function INIT(ByRef Door As Object, ByRef ProfileName As String) As Boolean
'Initialisierungsfunktion: wird aufgerufen, wenn Plugin
'eingebunden wird (beim Startvorgang des CDKs). Das Plugin bekommt mitgeteil,
'was für ein Name ihm zugeteilt wurde. Außerdem wird Door-Objekt übergeben.

Function INFO() As String
'Gibt die Bezeichnung des Plugins zurück

Sub ACTIVATION()
'Wird ausgeführt, wenn der Menüpunkt des Plugins angeklickt wird
```

Und bei Treibern:

```
Function INIT(ByRef Door As Object, ByRef DriverName As String) As Boolean
'Initialisierungsfunktion: wird aufgerufen, wenn Plugin
'eingebunden wird (beim Startvorgang des CDKs)

Function INFO() As String
'Gibt die Bezeichnung des Treibers zurück

Function START() As String
'Wird aufgerufen, wenn die Interfaces aktiviert werden sollen

Function STOPP() As String
'Wird aufgerufen, wenn die Interfaces deaktiviert werden sollen

Function VARUBOUND() As Integer
'Gibt die Anzahl der Variablen zurück, die der Treiber registrieren will

Function SETVAL(varname, value) As Boolean
'Wird aufgerufen, wenn der User den Wert einer Variable dieses Treibers
'ändert
```

Wird ein Treiber bzw. ein Plugin initialisiert, wird in ihm die INIT-Funktion aufgerufen. Dabei wird ihm eine Referenz auf das Door-Objekt übergeben, über das mit der Software kommuniziert werden kann. Das Objekt beinhaltet folgende Funktionen:

```
Public Sub resetScriptControl()
        Kompiliert den User-Quellcode neu

Public Function MakeLeftBarTable(InsertObject As Form)
        Erstellt einen Table in der linken Leiste (Das macht z.B. der
        Datenbrowser, damit er dort links angezeigt wird)

Public Sub RemoveLeftBarTable(Table As Object)
        Der Table kann auch wieder entfernt werden
```

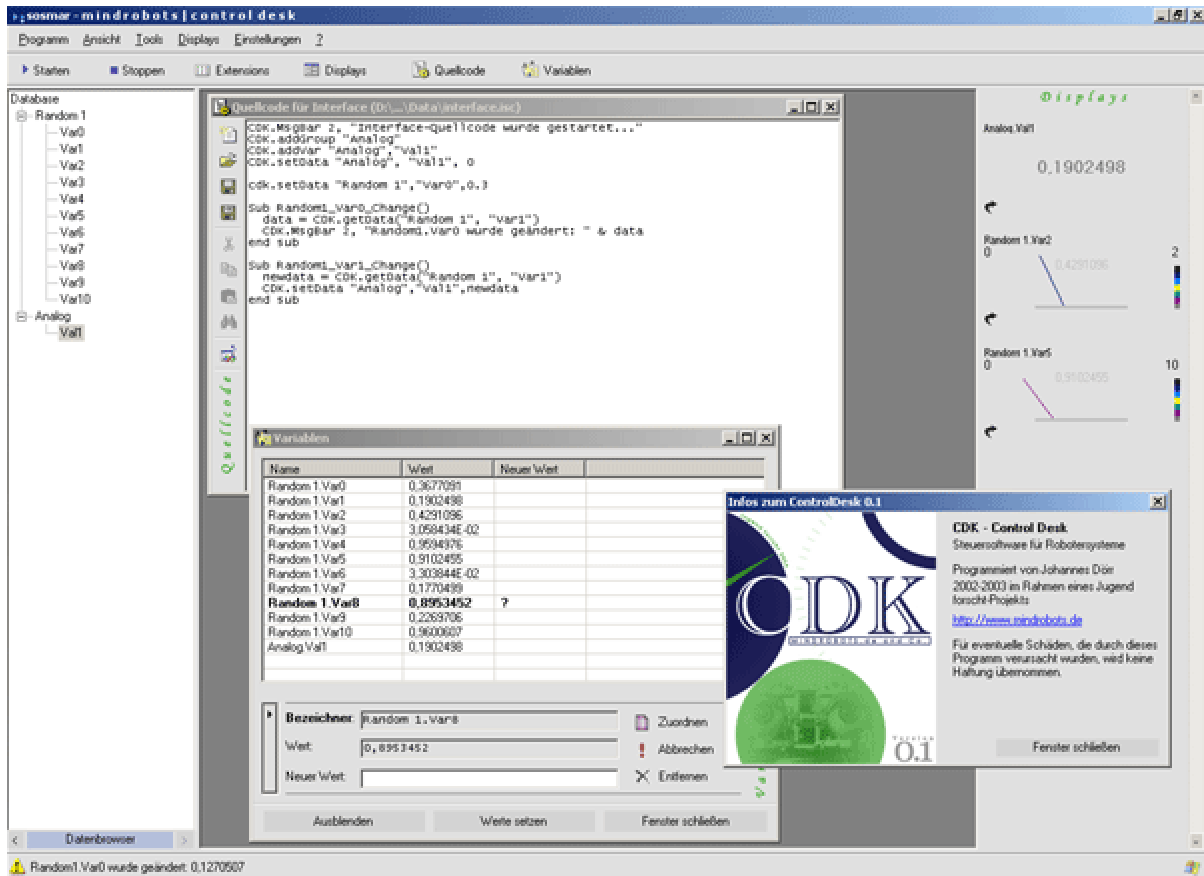
```
Public Sub AddMeToDataChangedEventList(Objekt As Object)
        Klassen, die in dieser Liste eingetragen sind, werden
        informiert, wenn sich etwas in AllData (alle Variablen) ändert.
        Fenster, die also einen Variablenwert anzeigen, können die
        Anzeige somit aktualisieren.

Public Sub RemoveMeFromDataChangedEventList(Objekt As Object)
        Und wieder Löschen des Eintrags

Public Sub MakeMeToDragdropTable(ByRef FormObjekt As Object, ByRef _
        OverObjekt As Object, ByVal DataType As String)
        Hier kann ein DragAndDrop-Zielobjekt angemeldet werden
        (siehe unten)

Public Sub UndoMakeMeToDragdropTable(ByRef FormObjekt As Object, ByRef _
        OverObjekt As Object, ByVal DataType As String)
        Abmelden des Tables

Public Sub MakeMeToChild(FormObjekt As Object)
        Ein Plugin kann mit dieser Funktion erreichen, dass ein Fenster
        von ihm als Child des CDKs angezeigt wird.

Public Sub drag(ByVal Data As Variant, DataType As String)
        Startet den DragAndDrop-Vorgang

Public Sub drop()
        Und beendet ihn wieder. Diese Sub ist im Prinzip völlig sinnlos,
        da das beim lösen der Maustaste automatisch passiert. Aber man
        kann den Vorgang hiermit abbrechen.

Public Function getDragedData() As Variant
        Gibt das Datum zurück, das verschoben wird

Public Function getDragedType() As String
        Gibt den Datentyp des Datums zurück, das verschoben wird

Public Function getDragStatus() As Boolean
        Ist gerade ein DragAndDrop-Vorgang aktiv?

Public Sub MsgBar(ByVal MsgBarStyle As Integer, ByVal Content As String)
        Zeigt eine Nachricht in der Statusleiste an

Public Function addVar(ByVal Group As String, ByVal VarName As String) _
            As Boolean
        Fügt einer Datengruppe in AllData eine Variable hinzu

Public Function addGroup(ByVal GroupName As String) As Boolean
        Fügt zu AllData eine neue Gruppe hinzu

Public Function getData(ByVal Group As String, ByVal Name As String) _
            As Variant
        Gibt den Wert einer Variable in AllData zurück

Public Function setData(ByVal Group As String, ByVal Name As String, _
            ByVal value As Variant) As Boolean
        Setzt den Wert einer Variable in AllData

Public Function getGroupNames(ByVal Index As Integer) As String
        Gibt die Gruppennamen in AllData zurück. Nur sinnvoll in
        Verbindung mit einer For-Schleife

Public Function getVarNames(ByVal GroupIndex As Integer, ByVal VarIndex _
            As Integer) As String
        Gibt die Variablennamen aus einer Gruppe in AllData zurück
        Funktioniert wie vorige

Public Function getVarsUBound(ByVal Group As String) As Integer
        Gibt die Anzahl der Variablen in einer Gruppe zurück

Public Function getGroupsUBound() As Integer
        Gibt die Anzahl der Gruppen in AllData zurück

Public Function getGroupIndexByName(ByVal Name As String) As Integer
        Gegenstück zu getGroupNames.

Public Function getVarIndexByName(ByVal GroupName As String, ByVal _
            VarName As String) As Integer
        Gegenstück zu getVarNames
```

Weiter unten finden Sie Code-Beispiele für einen Treiber.

8.2.3 Benutzeroberfläche

Mit dem Startbildschirm lässt sich ein neues Profil anlegen, in dem die zu verwendende Treiber sowie Fensterpositionen etc. eingespeichert werden. Dieses Profil lässt sich auch in diesem Fenster wieder auswählen und schon hat man alle Daten des entsprechenden Roboters auf dem Bildschirm. Normalerweise reicht ein Profil jedoch aus.

Der Hauptbildschirm:



Links im Bild sieht man den „Datenbrowser". Er enthält alle Variablen nach Gruppen sortiert. Er ist als Plugin implementiert und damit (wie alle anderen Plugins auch) vom eigentlichen System unabhängig kompiliert und greift über die Interface-Klasse mit dem bezeichnenden Namen „Door" auf die Daten zu. Plugins werden normalerweise als ganz normale Fenster angezeigt Beim Starten hat sich der Datenbrowser jedoch mit der Funktion „`MakeLeftBarTable`" in die linke Leiste „gepflanzt". Diese Möglichkeit ist sehr praktisch, da auf diesem Wege oft verwendete Plugins nicht in dem Fenstergewirr untergehen sondern übersichtlich einsortiert werden.

Unten ist eine Message-Bar zu finden, die sich genauso wie die Message-Boxen verwenden lässt. Somit können Meldungen angezeigt werden, die eine geringere Wichtigkeit haben und nicht mit „Ok" bestätigt werden müssen.

Rechts ist die Displaybar zu sehen. Über das Menü lassen sich neue Displays hinzufügen und entfernen, per Drag and Drop werden sie dann mit Variablen verlinkt. Displays sind Plugins, die sich jedoch nicht im Standard-Verzeichnis sondern in einem speziellen Unterordner befinden und beim Start der Software als Display initialisiert werden. Von einem Display können mehrere Instanzen angezeigt werden. In dem Bild oben handelt es sich bei der zweiten und dritten Analoganzeige um zwei Instanzen desselben Displays, die mit zwei verschiedenen Variablen verlinkt sind.

Das Fenster im Hintergrund enthält den Userquellcode. Über diesen Code können der Datenaustausch zwischen Roboter und PC kontrolliert und die Beziehungen zwischen mehreren Variablen festgelegt werden. Ändert sich der Wert einer Variablen, wird hier ihre Ereignisprozedur aufgerufen. Der Quellcode, der in dieser Prozedur steht, wird also immer bei einer Änderung des Variablenwertes ausgeführt und kann darauf reagieren. Somit kann zum

Beispiel gezählt werden, wie oft eine Variable geändert wurde oder einer Variable kann die Differenz von zwei anderen Variablen zugewiesen werden. Die Anwendung hängt somit natürlich von den Daten ab. Ein anderes Beispiel wäre eine Anzeige, die Alarm gibt, wenn die Akkuspannung des Roboters einen bestimmten Wert unterschreitet.

8.2.4 Drag and Drop

Ein wichtiges Ziel des CDKs ist es, eine einfache und komfortable Möglichkeit zu schaffen, Werte von Mikrocontrollern in Echtzeit zu verwalten und zu beobachten. Eine wichtige Funktion hierfür ist Drag and Drop. Sie soll es zum Beispiel ermöglichen, Variablen mit der Maus aus dem Datenbrowser zu nehmen und mit einem Display zu verlinken.

Die Drag and Drop-Funktionen sind auch in der „Door"-Klasse zu finden. Der Datenbrowser startet den Drag-Vorgang, wenn auf ein Element geklickt wird. Der Sub Door.Drag wird einmal das Datum (Einzahl von Daten), nämlich der gesamte Pfad (beispielsweise „Treiber1.Var1"), übergeben. Außerdem muss der Typ angegeben werden. In diesem Fall handelt es sich um „cdk_link", da wir ja eine Variable mit einem Display verlinken wollen.

Das Display wird in der Regel ein Plugin sein, da das CDK selbst keine Displays enthält. Bei der Initialisierung muss es sich als DragAndDrop-Table anmelden. Das heißt, dass es einen bestimmten Typ von Daten annehmen kann. Door.MakeMeToDragdropTable erledigt dies. Neben Anderem wird der Typ übergeben, in diesem Fall wie gesagt „cdk_link". Dadurch wird gleichzeitig verhindert, dass irgendein anderes Datenformat übergeben wird, mit dem das Display nichts anfangen kann.

Wenn man jetzt mit der Maus aus dem Datenbrowser eine Variable „packt", wird neben dem Cursor ein entsprechendes Icon angezeigt, das je nach Datentyp anders ist. Dieses Icon macht mit einem eventuell erscheinenden Kreuz kenntlich, ob das Datum an der momentanen Mausposition abgelegt werden kann.

Plugins können also über „Door" Drag and Drop-Vorgänge steuern. Die verfügbaren Typen sind dabei unbegrenzt. Vom CDK selbst wird nur „cdk_link" verwendet, aber Plugins können untereinander auch andere Typen verwenden. Als Icon wird dann ein Standardlogo verwendet, es sei denn, es werden in das Verzeichnis „Images" zwei entsprechende bmp-Dateien angelegt. Für „cdk_link" liegen dort bereits „cdk_link_i.bmp" und „cdk_link_p.bmp". Das CDK sucht also erst in diesem Verzeichnis nach den Dateien (Typ + „_i" bzw. „_p", stehend für „impossible" und „possible"), bevor es das Standardlogo verwendet.

8.2.5 Beispielcode für einen Treiber

Dieser Beispiel-Quellcode zeigt, wie ein mit Visual Basic programmierter Treiber für eine automatische Tür aussehen könnte.

Klassenmodul: Class1.cls

```
Option Explicit
'==================================================================================
'                         Dieses Klassenmodul ist mit dem CDK
'                      verknüpft und muss unbedingt vorhanden sein
'==================================================================================

'----------------------------------------------------------------------------------
'HINWEIS: Die Köpfe der folgenden Funktionen/Subs dürfen nicht
'         verändert werden, da das CDK auf sie zugreift.
'----------------------------------------------------------------------------------
    Function INIT(Door As Object, DName As String) As Boolean
    'Initialisierungsfunktion: wird aufgerufen, wenn Plugin
    'eingebunden wird (beim Startvorgang des CDKs)
        Set TheDoor = Door
        DriverName = DName
        INIT = True
    End Function

    Function INFO() As String
    'Gibt die Bezeichnung des Plugins zurück - unbedingt einen
    'individuellen und aussagekräftigen Namen wählen!
     INFO = "Schiebetür1"
    End Function

    Function START() As String
    'Wird aufgerufen, wenn die Interfaces aktiviert werden sollen
     Seriell.send(„lamp on") 'Hier wird über die serielle Schnittstelle
                              'eine Signallame eingeschaltet
    End Function

    Function STOPP() As String
    'Wird aufgerufen, wenn die Interfaces deaktiviert werden sollen
     Seriell.send(„lamp off")
    End Function
```

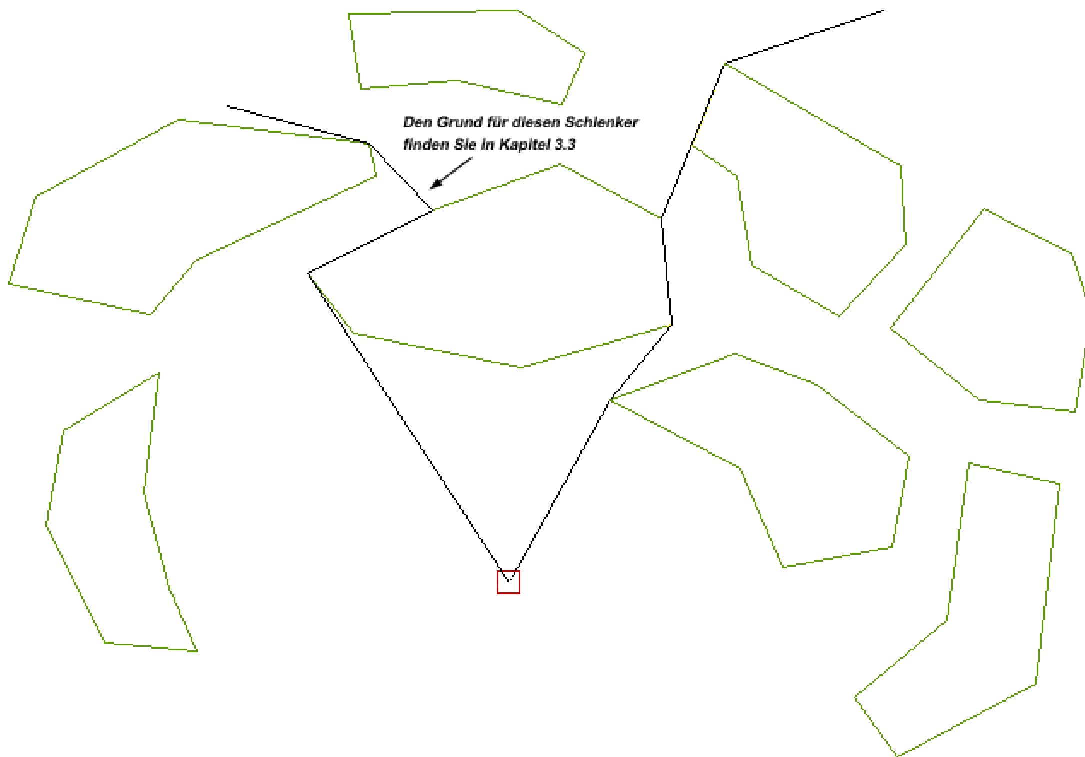
```
    Function VARNAMES(number) As String
    'Gibt die Namen der Variablen zurück, die auf dem Mikrocontroller
    'vorhanden sind.
     If number = 0 Then
      VARNAMES = "Signallampe"
     Else
      VARNAMES = "Status"
     End If
    End Function

    Function VARUBOUND() As Integer
    'Gibt die Anzahl der Variablen zurück, die auf dem Mikrocontroller
    'gespeichert sind.

     VARUBOUND = 1 'Null-basierter Index, also 2 Variablen
    End Function

    Function SETVAL(varname, value) As Boolean
    'Wird aufgerufen, wenn der User den Wert einer Variable dieses Treibers
    'ändern möchte.
     If varname = "Signallampe" Then
      If value = "ON" Then
       Seriell.send(„lamp on")
      Else
       Seriell.send(„lamp off")
      End If
     Else
      If value = "OPEN" Then
       Seriell.send(„door open")
      Else
       Seriell.send(„door close")
      End If
     End If
     SETVAL = True
    End Function
'=================================================================================
```

Modul: Module1.bas

```
Option Explicit
'=================================================================================
'                         Dieses Modul dient der öffentlichen
'                         Bereitstellung der Door-Klasse u.Ä.
'=================================================================================

Public TheDoor As Object
Public DriverName As String
```

Die Variablen dieses Beispieltreibers sind beide nur dazu da, um von der Kontrollsoftware geschrieben zu werden. Es ist jedoch keine Variable vorhanden, deren Wert der Treiber selbst ändert. Ein Beispiel dafür wäre ein Schalter, der an der Tür angebracht ist. Der Treiber könnte den Schalter ständig überprüfen und den Status (also ‚an' oder ‚aus') in eine Variable schreiben. Dazu wäre nur eine Schleife nötig, die bei einer Änderung den neuen Wert mit dem Aufruf „`TheDoor.setData(DriverName, `*`Status`*`)`" an die Kontrollsoftware übergibt.

8.3 Wegberechnung



8.3.1 Endpoint-Berechnung und verschiedene Hindernisformen

Für die Ermittlung der Endpoints eines Hindernisses schrieben wir viele Algorithmen, bis wir die optimale Methode endlich fanden. Betrachtet man das Bild rechts, so liegt die erste Lösung über die Winkel auf der Hand. Man sucht sich einfach die beiden Points heraus, bei denen der Winkel am Ziel maximal ist. Dieses Verfahren hat jedoch den Nachteil, dass der Rechenaufwand nicht linear ist, da jeder Punkt mit allen anderen überprüft werden muss.

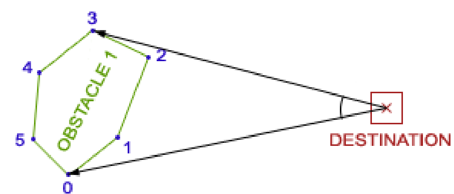


Abb. 3.1

Eine bessere Methode ermöglicht der CCW-Algorithmus von Robert Sedgewick. CCW steht für counterclockwise (engl. entgegen dem Uhrzeigersinn). Mit der Funktion kann überprüft werden, ob einer von drei Punkten links oder rechts neben der Geraden, die durch die beiden anderen Punkte verläuft, liegt. Dies ist für die Lösung unseres Problems sehr hilfreich. Doch auch hier gibt es zwei Varianten.

Bei der ersten wird am Anfang der Point 0 als geschätzter Endpoint (wir betrachten zur Vereinfachung nur Endpoint A, der in diesem Falle Nummer 0 ist) gesetzt. Nun geht die Schleife alle weiteren Punkte durch und überprüft, ob ein Punkt weiter links als Point 0 liegt. Ist dies der Fall, wird der neue Punkt als geschätzter Endpoint gespeichert. Am Ende ist der geschätzte Endpoint der endgültige, da er am weitesten links liegt. Allerdings kommt es bei dieser Vorgehensweise zu Problemen mit speziellen Hindernisformen.

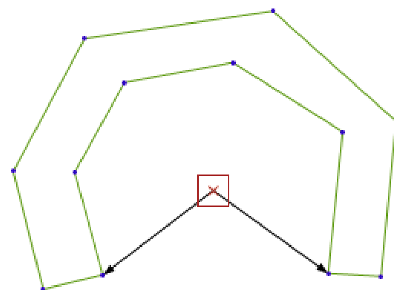


Abb. 3.2

Wir haben zwei Hindernisarten definiert. Die erste Art ist die normale Form und ist in *Abbildung 3.1* zu sehen. Die zweite Art hingegen beschreibt Hindernisse, die das Ziel umschließen (*Abb. 3.2*). Bei solchen Hindernisformen gab es mit der beschriebenen Methode Probleme, weshalb ein weiteres Verfahren programmiert werden musste, das wie folgt funktioniert.

Man denke sich eine Linie vom Ziel zu Punkt 1 (*Abb. 3.1*). Mit der CCW-Funktion wird überprüft, auf welcher Seite der Nachbarpunkt 2 liegt, in diesem Fall rechts. Dasselbe geschieht mit dem anderen Nachbarpunkt 0, er liegt hier links. Es handelt sich also nicht um einen Endpoint. Starten wir jedoch mit Punkt 0, so stellen wir fest, dass beide Nachbarpunkte auf der rechten Seite liegen – das Zeichen für Endpoint A. Bei Punkt 2 liegen die beiden Nachbarpunkte links – Endpoint B.
Natürlich gibt es auch hier wieder ein Problem. Bei ganz besonderen Hindernisformen, die von der Form her wie auf dem zweiten Bild aussehen, das Ziel jedoch außerhalb liegt, kommt es manchmal vor, dass der Endpoint nicht eindeutig ist. Diese falschen Ergebnisse lassen sich jedoch dadurch erkennen, dass sie auf dem Trace des Hindernisses liegen, sie also nicht direkt vom Ziel erreicht werden können.

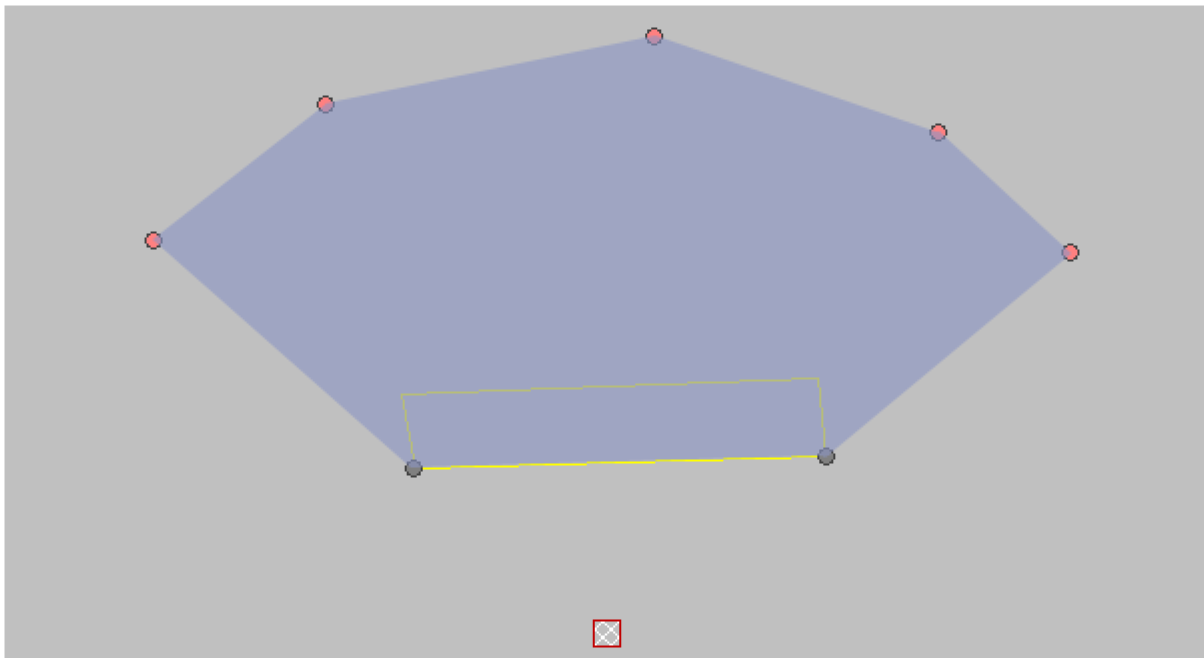
Zur Vereinfachung haben wir hier immer nur von einem Endpoint geredet, ein Hindernis besitzt jedoch logischerweise immer zwei, die Berechnung ist also immer für beide durchzuführen.

8.3.2 Trace-Berechnung

Auch die Trace-Berechnung lässt sich auf verschiedene Art lösen. In diesem Fall ist jedoch die Geschwindigkeit von größter Bedeutung, da der Algorithmus oft ausgeführt werden muss.

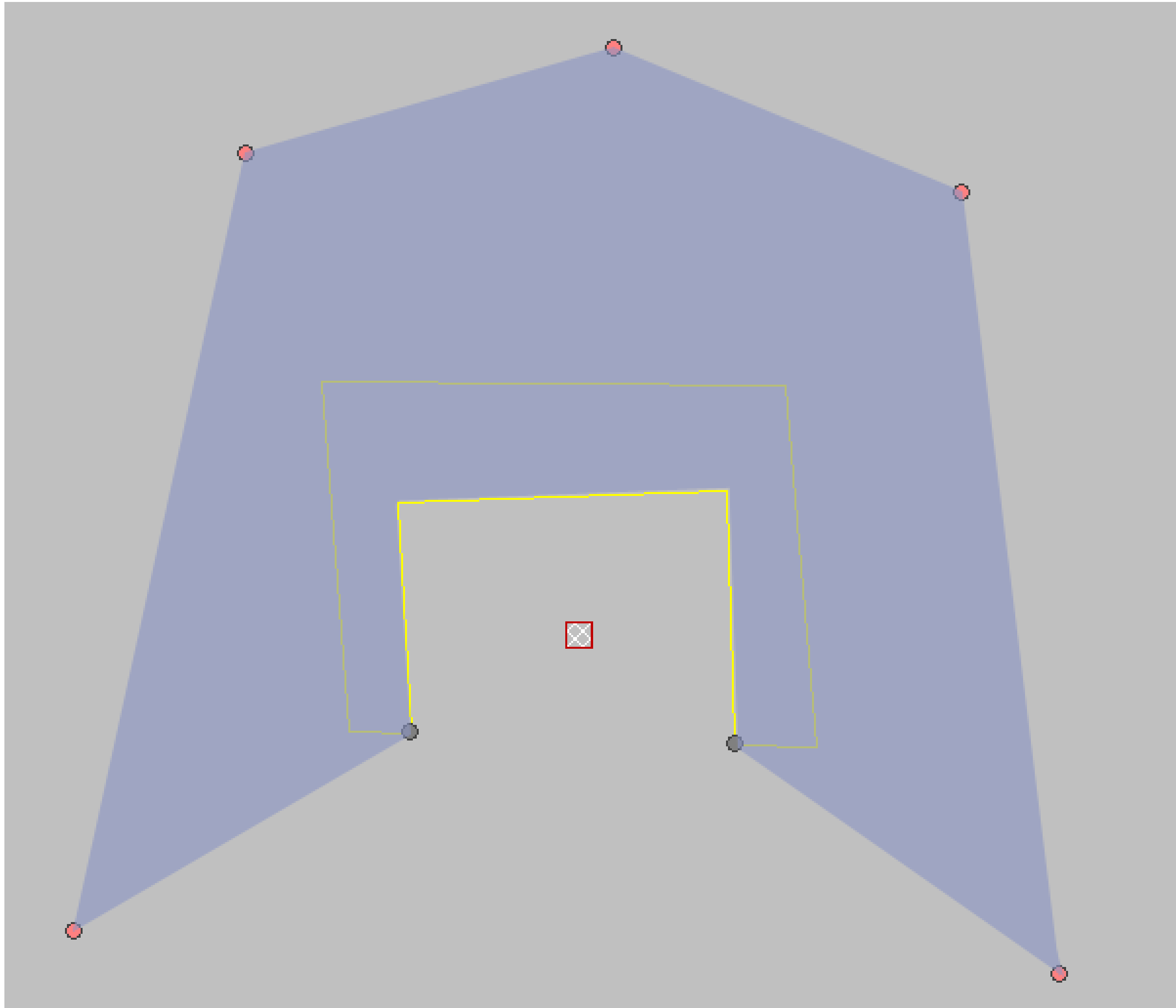
Die erste Version arbeitete mit Winkeln. Dabei wurden die Winkel zwischen der Geraden durch Ziel und jeweils einen Endpoint und der X-Achse errechnet. Der Wert des Winkels zwischen der Geraden durch Ziel und Roboterposition und der X-Achse muss dann zwischen den beiden vorigen Winkeln liegen, wenn sich der Roboter auf dem Trace befindet. Die trigonomischen Funktionen verbrauchen jedoch viel Rechenkapazität, was das Verfahren langsam macht. Außerdem lässt sich nicht erkennen, ob der Roboter bereits hinter dem Hindernis ist oder noch davor. Nur im letzteren Falle befindet er sich auf dem Trace, obwohl die Funktion dies bei beiden behauptet.

Eine neue Idee wurde durch die Inside-Funktion, ebenfalls von Robert Sedgewick, geweckt. Mit der Inside-Funktion lässt sich überprüfen, ob sich ein Punkt innerhalb eines Polygons befindet oder außerhalb liegt.

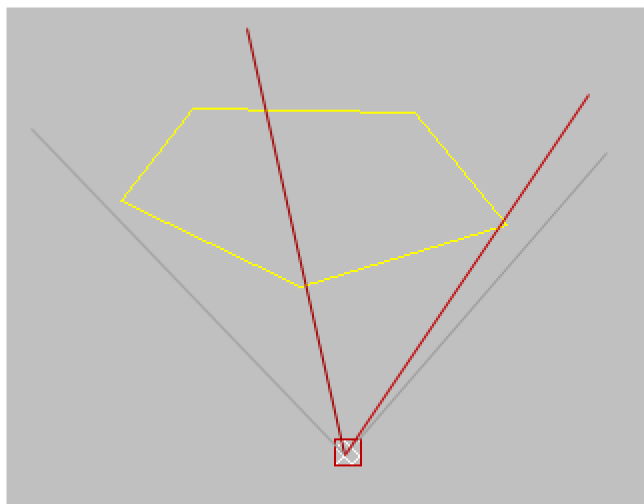


Das Bild oben zeigt diese Version grafisch. Der Blaue Bereich wird von den beiden Endpoints sowie fünf weiteren Punkten aufgespannt. Diese fünf Punkte haben alle die gleiche Entfernung vom Ziel und legen so den maximalen Einflussbereich des Hindernisses in der Karte fest. Ob sich der Roboter nun innerhalb dieser Fläche befindet, ist mit der Inside-Funktion einfach zu erkennen.

Auch die umschließende Art der Hindernisse wird wunderbar unterstützt, wie das nächste Bild zeigt.



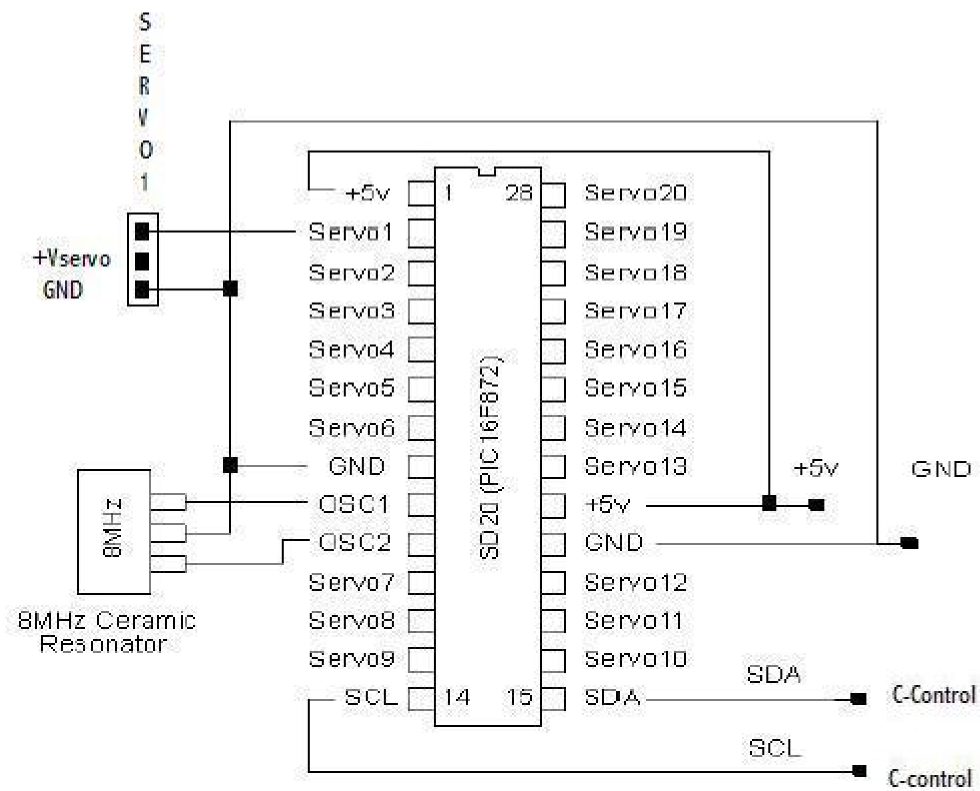
Auch diese Version ist jedoch zu rechenaufwendig und eigentlich auch zu kompliziert im Vergleich zu der nächsten Methode, auf die wir aus unerklärlichen Gründen erst später gekommen sind…



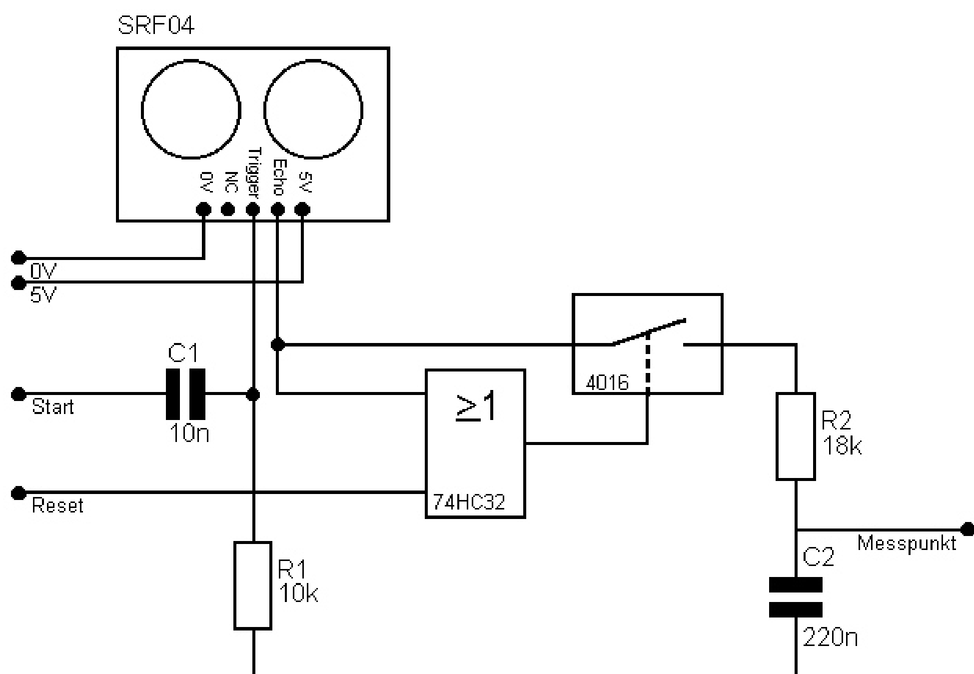
Hierbei wird einfach überprüft, ob die Verbindungslinie vom Roboter zum Ziel eine Verbindungslinie der Hindernispunkte schneidet. Diese Methode läuft wesentlich schneller als ihre Vorgängerversionen und ist außerdem viel einfacher zu implementieren.

8.4. Schaltpläne

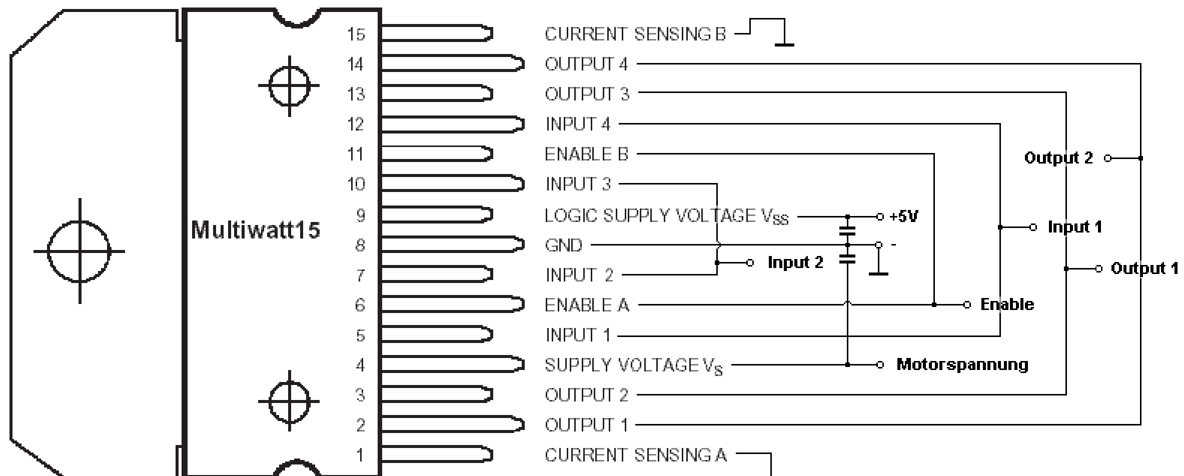
8.4.1 Ansteuerung der Servos mit dem SD20



8.4.2 Abfrage der Ultraschallsensoren (SRF04)



8.4.3 Schaltplan für die Motortreiber



Danksagungen

Teile der Arbeit wurden durch Mittel der Elektronikforschung der Volkswagen AG unterstützt.

Vielen Dank auch an:

- EDV-Beratung & Robotertechnik Jörg Pohl
- Alpha-Werbetechnik
- foil direct Fachhandel für Werbetechnik
- Reinhard Hodde

Literaturverzeichnis

- Robert Sedgewick - Algorithms 2nd Edition, Addison-Wesley Co., 1988
- Robert Sedgewick - Algorithms in C++ 1st Edition, Addison-Wesley Co., 1992
- Gordon McComb: The Robot Builder's Bonaza, 1. Oktober 2000, 2. Auflage

- http://www.roboterwelt.de
- http://www.roboternetz.de
- http://www.robotmaker.de
- http://www.elektronik-projekt.de
- http://www.cc2net.de
- http://www.c-control.de
- http://www.acroname.com/robotics/info/articles/irlinear/irlinear.html