fachhochschule hamburg
Fachbereich Elektrotechnik und Informatik
Berliner Tor 3
20099 Hamburg

Diplomarbeit

Entwicklung eines autonomen, mobilen Roboters zur Kartographie

von Dirk Gehrmann

20. Februar 2001

betreut von:
Prof.Dr.rer.nat. Gunter Klemke
Prof.Dr.rer.nat. Kai von Luck

Danksagung

Zunächst möchte ich Herrn Prof. Dr. Gunter Klemke für die Betreuung, das entgegengebrachte Vertrauen und die Unterstützung danken.

Ich danke auch Herrn Prof. Dr. Kai von Luck für sein Einverständnis, die Arbeit des Zweitgutachters und die zusätzliche Betreuung zu übernehmen.

Mein Dank gilt auch Dipl. Ing. Arno Eikhof für die Entwicklung des Netzwerktreibers und Hilfe bei der Entwicklung rund um das Netzwerk.

Dipl. Ing. Oliver Rodenhagen für Ideen, diverse Elektronikbauteile und die Motoren.

Meinem Vater, Dipl. Ing. Jörn Gehrmann, danke ich für das Korrekturlesen, die hervorragenden mechanischen Zeichnungen im Anhang und meiner ganzen Familie für das Verständnis und die Unterstützung bei dieser Aufgabe.

Dirk Gehrmann

Inhaltsverzeichnis

1. Einleitung

Nicht erst seit der Industrialisierung versucht der Mensch Geräte und Maschinen zu entwickeln, die das Leben erleichtern sollen. Früher waren dies mechanische Lösungen und schon Heron von Alexandria schrieb im 1. Jahrhundert nach Christus über den Bau von Automatentheatern.

Heute können wir Systeme mit mechanischen, elektronischen und Software-Komponenten realisieren. Die damit verbundene Flexibilität der Problemlösungen ist in großem Maße den Computern zu verdanken, die aus der modernen Welt wie wir sie heute kennen nicht mehr wegzudenken sind. Die ständig steigende Miniaturisierung und preisgünstige Verfügbarkeit heutiger Prozessoren macht es möglich auch sehr komplexe Steuerungen in kleinen Geräten unterzubringen.

Diese Möglichkeiten hat sich die Industrie zunutze gemacht und setzt heute millionenfach Roboter zur Produktion ein. In der Automobilindustrie beispielsweise schweißen und lackieren sie Karosserien. Roboter haben gegenüber menschlicher Arbeitskraft unter anderem den großen Vorteil nicht nachlassender Präzision[1].
Aber sie haben auch gravierende Nachteile. So sind sie nur bedingt flexibel, da jede Bewegung ihnen mühsam beigebracht werden muß. Ein Ansatz dies zu verbessern ist, den Roboter zu führen, ihm also die Bewegungen zu „zeigen“ und nachahmen zu lassen. Dies erleichtert zwar die Programmierung erheblich, stellt aber keinen echten Durchbruch dar.

Der nächste fast logische Schritt ist somit, Systeme zu entwickeln, die ihre Entscheidungen selbständig treffen. Die Roboter werden so autonom und auch „intelligent“. Diese sogenannte dritte Generation von Robotern, die zur Zeit Gegenstand vieler Entwicklungen und Forschungen ist, stellt so ein äußerst interessantes Betätigungsfeld dar.

[1] Nicht zuletzt auch ihre Kosten, was die Aktionäre freut.

2. Aufgabenstellung

Ziel dieser Diplomarbeit ist es, einen Roboter zu entwickeln und zu programmieren. Es soll sich um ein möglichst flexibles Systems handeln, das „beliebig“ ausbaubar ist, je nach erwünschter Funktionalität und finanziellem Aufwand. Softwareseitig soll er am Ende in der Lage sein, ein unbekanntes Terrain eigenständig zu erkunden und anhand von Meßwerten eine zwei-dimensionale Karte der Umgebung zu erstellen. Dies alles soll in Echtzeit geschehen.
Zunächst ist es ausreichend, die Daten nur zu sammeln. Eine Interpretation der Kartendaten, in welcher Weise auch immer, ist noch nicht vorgesehen, da dies eine äußerst komplexe Aufgabe darstellt und den Rahmen dieser Arbeit sprengen würde.

Um die Kartographie zu ermöglichen, sind einige Teilaufgaben zu erfüllen:

1. Fertigstellung und Einbau der noch fehlenden Hardware-Module.
2. Es muß ein Betriebssystem für die einzelnen Rechner, die in diesem System zum Einsatz kommen sollen, entwickelt werden.
3. Entwicklung einer geeigneten Kommunikationsform für das Gesamtsystem.
4. Entwicklung der Programme, die dem Roboter später „Leben“ einhauchen. Hierzu zählen:
   a) Programmierung der essentiellen Algorithmen zum Beispiel Treiber zum Ansteuern der Motoren, Meßfunktion für das Sonar und ähnliches.
   b) Entwicklung einer geeigneten Steuerung, mit der das System seine Umgebung erkundet.
   c) Und nicht zuletzt die Algorithmen der eigentlichen Kartographie, also Meßdaten sammeln, klassifizieren und archivieren.

Hier sei bemerkt, daß nicht der gesamte Roboter im Rahmen der Diplomarbeit entstanden ist. Einiges ist bereits vorher entwickelt worden und wird hier, vielmehr der Vollständigkeit halber, beschrieben. Seitens der Hardware sind dies folgende Module:

- Mechanik
- Motorsteuerung
- Sonar
- Hauptrechner
- LC-Display und Tastatur
- Laderechner

Seitens der Software wurde auf nur wenige fertige Algorithmen zurückgegriffen:

- Linien, Ellipsen-Algorithmus sowie der Display-Ansteuerung mit eigenem Zeichensatz für das LC-Display.
- Einem Treiber für das Netzwerk.

Das folgende Kapitel 3 soll zunächst einen Überblick über die Hardware und die Betriebssysteme geben. Danach folgt die Software in Kapitel 4, also die Algorithmen, die in ihrer Gesamtheit die Kartographie ermöglichen und die Ergebnisse. Zum Schluß eine Zusammenfassung und ein kleiner Ausblick in Kapitel 5.

Abbildung 2.1.: Foto des Roboters

3. System

Auf dem Markt sind eine ganze Reihe verschiedener Roboter und Plattformen verfügbar. Doch wer die Wahl hat, hat auch die Qual und eines scheint allen gemeinsam: Entweder ist ihre Ausbaufähigkeit begrenzt, oder es wird teuer. Hinzu kommt die geforderte Flexibilität, die bei den meisten Plattformen stark eingeschränkt scheint. Die Konsequenz daraus ist also eine eigene Plattform zu entwickeln mit all den Vor- und Nachteilen. Dies scheint aber auch eine sehr reizvolle Aufgabe zu sein, da sie interdisziplinäre Lösungen erfordert. Also im mechanischen, elektronischen und Informatik-Bereich.

Bei der Konzeption tritt ein schwerwiegendes Problem zu Tage. Kein Algorithmus, der genutzt werden soll und keine zeitlichen Anforderungen an das System sind von vornherein näher spezifiziert. Schon hier ergibt sich die Schwierigkeit, den Aufbau des Roboters flexibel gestalten zu müssen, um späteren Anforderungen gerecht zu werden. Eine Integration neuer Komponenten aber auch der Austausch veralteter Teile muß auch in späten Entwicklungsstadien möglich sein. Diese Notwendigkeit korreliert mit der in der Aufgabenstellung bereits geforderten Flexibilität voll und ganz.

Mechanisch ist diese Forderung vielleicht nicht in aller Konsequenz erfüllbar. Seitens der Steuerung jedoch drängt sich hier das Konzept des *verteilten Systems* geradezu auf. Die Idee ist, eher kleine, intelligente Komponenten zu entwickeln und sie über ein geeignetes Netzwerk miteinander kommunizieren zu lassen. Die wichtigsten Vorteile gegenüber jedem monolithischem Ansatz, also einem zentralen Steuerrechner, liegen auf der Hand:

- **Flexibilität** durch Austauschbarkeit verbesserter Komponenten.
- **Skalierbare Rechenleistung** durch einfache Anbindung neuer Rechner in das Netzwerk.
- **Erweiterbarkeit** ohne signifikanten Verlust an Rechenleistung im bereits bestehenden Teilsystem, was gerade bei zeitkritischen Prozessen, zum Beispiel Sensorik oder Aktorik, wichtig ist.
- Das Netzwerk als **definierte Schnittstelle** zwischen den einzelnen Modulen.

Doch wie fast immer im Leben, so hat auch diese Medaille eine Kehrseite, die hier nicht verschwiegen werden darf.

- **Hohe Anfangskosten** bis der Kern eines solchen Systems steht und so zumindest ein Teil der gesamten Funktionalität zu Verfügung steht.
- Der **Kommunikationsaufwand** zwischen den Komponenten wächst zwangsläufig und benötigt so immer einen Teil der Rechenleistung.

Diese Nachteile dämpfen die Euphorie natürlich ein wenig. Bei Abwägung der angesprochenen Vor- und Nachteile jedoch erscheint das verteilte System gegenüber einer zentralen Steuerung als das Konzept, das die geforderte Flexibilität am besten unterstützt und wurde deshalb gewählt.

3.1. Mechanik

In der Fachliteratur finden sich einige verschiedene Konfigurationen hinsichtlich der Konstruktion eines Roboters. Sie alle haben zweifelsfrei ihre Vorteile, doch keine Plattform kann sich rühmen auch frei von Nachteilen zu sein. Wie so oft ist die Wahl also irgendwo immer ein Kompromiß.

1. Die Ackermann Steuerung, hinlänglich von Autos bekannt, besticht durch seine relativ einfache Konstruktion, alle nötigen Teile sind im Modellbauhandel verfügbar und müssen nicht selbst angefertigt werden. Leider gestaltet sich das sinnvolle Manövrieren und die Berechnung des zurückgelegten Weges als recht schwierig.

2. Synchro-Drive ist mechanisch sehr aufwendig, da mehrere Räder gleichzeitig in ihrem Winkel verstellt werden müssen. Ansonsten aber eine äußerst interessante Antriebsart.

3. Zwei angetriebene Räder seitlich und ein drittes nachlaufendes Rad hinten ist die vielleicht beliebteste Konfiguration. Sie ist einfach und der Roboter ist äußerst beweglich, jedoch macht das nachlaufende Rad etwas Ärger. Bei einer Drehung muß sich dieses in die Drehrichtung ausrichten. Hierbei kommt es zu einem mehr oder minder schlimmen und unerwünschten Versatz der ganzen Plattform, was auf Dauer der Positionsbestimmung schadet.

4. Kettenantrieb wie von Geländefahrzeugen bekannt. Sie bieten ein hohes Maß an Beweglichkeit. Auch ihre große Auflagefläche am Untergrund spricht für sie, da sie eine effiziente Kraftübertragung gewährleisten und das Überfahren kleinerer Gegenstände, wie etwa herumliegende Stromkabel, kein Problem ist. Zwei Nachteile seien hier jedoch vermerkt. Erstens ist, bedingt durch die große Reibung am Untergrund, ein hoher Energieverbrauch zu erwarten. Und zweitens beeinflußt dieser Reibungskoeffizient den durchfahrenen Winkel bei einer Drehung ganz deutlich. Trotzdem wurde diese Antriebsart, nicht zuletzt aus optischen Gründen[1], gewählt.

Basis der gesamten Konstruktion bildet ein Chassis aus 3mm starken Aluminium. Oberhalb der Plattform findet sich Platz, um die Spannungsversorgung unterzubringen. Die einzelnen Module werden darüber mit ineinander verschraubbaren Abstandshaltern gestapelt, was die Unterbringung beliebig vieler Module erlaubt, je nach erwünschter, maximaler Bauhöhe. Der Platz auf dem Chassis reicht aus, um nebeneinander gleich drei Europlatinen quer einzubauen. Dabei ist zwischen den Platinen immer noch ein Abstand von etwa 15mm, der als Zwischenraum zur Verlegung der Netzwerk-Leitung und Spannungsversorgung der einzelnen Module dient.

[1] Es ist definitiv **kein** Panzer!

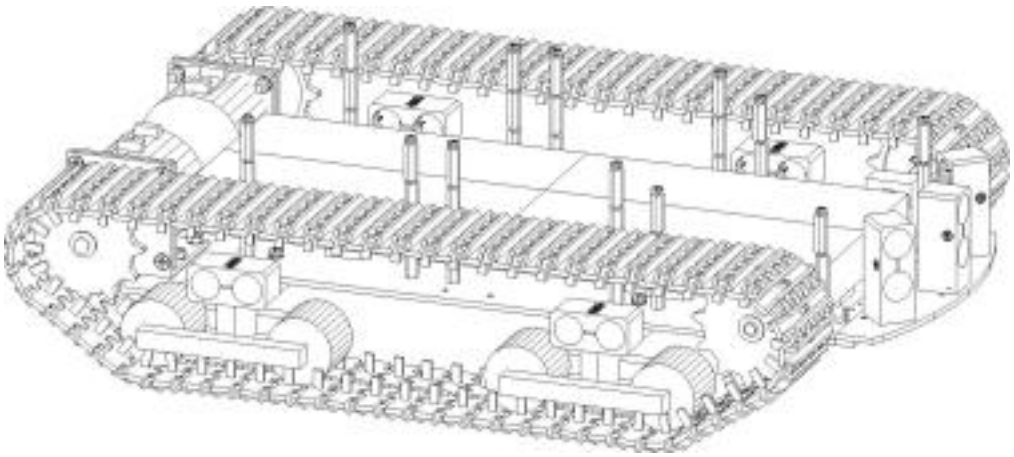


Abbildung 3.1.: Mechanik

Den Antrieb besorgen die beiden Ketten, die von den hinten angeordneten Motoren getrieben werden. Im vorderen Teil sind zwei Leiträder befestigt, die die Ketten führen sollen. Das Gewicht des Roboters nehmen die eigentlichen Laufrollen auf der Unterseite auf. Je zwei dieser Rollen sind auf einer beweglichen Achse an je einem Rollenträger angeordnet. So kann sich die Kette dem Untergrund recht flexibel anpassen und mit der großen Auflagefläche eine wirksame Kraftübertragung gewährleisten. Diese Konstruktion hat den weiteren Vorteil, daß die Motoren nicht mit dem Gewicht des Roboters belastet werden. Mechanische Stöße wirken zunächst auf die massiven Rollen und können die teuren und eher empfindlichen Motoren nicht beschädigen[2].

[2]Das Chassis verträgt ohne weiteres Belastungen von 80kg.

3.2. Basis-Elektronik

Ein System, das sich frei bewegen soll um seine Umwelt zu erkunden, benötigt eine mobile Spannungsversorgung. Durch das Einsatzgebiet des Roboters, nämlich Innenräume, kommen keine Energieträger in Frage die eine Verbrennung benötigen und damit Abgase produzieren, als auch solche, die von äußeren Umwelteinflüssen abhängig sind wie Solarzellen und ähnliches. Brennstoffzellen scheiden aufgrund hoher Kosten und (noch) großem Gewicht aus. Übrig bleiben hier also lediglich Batterien beziehungsweise Akkumulatoren, da letztere wiederaufladbar sind.

Die wohl gebräuchlichsten sind Nickel-Cadmium-Akkus (NiCd) und in zunehmenden Maße auch Nickel-Metall-Hybrid (NiMH) Zellen, die eine um etwa 50% höhere Energiedichte als NiCd-Akkus aufweisen, aber auch sehr viel teurer sind und daher nicht in Frage kommen.
Blei-Gel Akkumulatoren besitzen eine vergleichbare Energiedichte wie NiCd-Akkus, sind jedoch sehr viel preiswerter und in vielen verschiedenen Größen erhältlich. Zudem ist die Wiederaufladung dieser Zellen vergleichsweise unproblematisch, lediglich eine zu starke Entladung muß vermieden werden, da dies die Akkus zerstören würde.

Vier solcher Blei-Gel Akkus mit einer Spannung von je 12V und einer Kapazität von 24Wh wurden eingesetzt. Drei sind in Reihe geschaltet und erzeugen so 36V nur für die Versorgung der Motoren. Der letzte ist für die Steuerelektronik zuständig. Mechanisch finden sie auf dem Chassis unterhalb der Module Platz.

3.2.1. Standard-Rechner

Was die Wahl der Prozessoren angeht, bietet sich hier eine ganze Palette von Systemen der verschiedensten Hersteller an. Von Motorolas 68000 Familie, embedded Personal-Computer über Mikrocontroller bis hin zu digitalen Signal Prozessoren gibt es eine Fülle verschiedener. Doch an die gewählten Prozessoren werden einige Ansprüche gestellt, die natürlich nicht alle erfüllt werden können:

- hohe Rechenleistung
- geringer Energieverbrauch
- simple Anbindung an ein Netzwerk
- Verfügbarkeit von Compilern
- leichte Realisierung der Hardware (Herstellung von Platinen etc.)
- geringer Preis

Die Wahl fiel auf COM20051 Prozessoren der Firma Standard Microsystems Corporation (SMSC). Es handelt sich um 8-Bit Mikrocontroller, die mit der bekannten 8051 Prozessorfamilie[3] von Intel voll kompatibel sind. Sie arbeiten mit einer Taktfrequenz von 16MHz und können bis zu 64KByte Programmspeicher und 64KByte Datenspeicher adressieren. Zusätzlich beinhalten diese Prozessoren einen ARCNET-Controller mit einem *eigenen* Speicher von 1024Byte Größe. Dieser Speicher wird später bei der Kommunikation der Module noch eine zentrale Rolle spielen.

Nicht nur ihr geringer Preis von etwa 25 DM macht diese Prozessoren so interessant, sondern auch die Tatsache, daß der gleiche ARCNET-Controller mit exakt derselben Ansteuerung auch als eigenständiges Bauelement unter der Bezeichnung COM20020 erhältlich ist. Mit dieser Technik ist es nun möglich, (theoretisch) *jeden beliebigen Rechner* an das Netzwerk anzubinden. So kann zum Beispiel ein digitaler Signal-Prozessor in Verbindung mit dem COM20020 ARCNET-Controller den COM20051 Prozessor ersetzen und in das Netz integriert werden. Seitens der Ansteuerung, also in erster Linie der Software, ändert sich (fast) nichts. Das ist ein weiterer Pluspunkt für die geforderte Flexibilität des gesamten Systems. Leider ist die Rechenleistung eines einzelnen COM20051 Prozessors alles andere als üppig.

Für diese Familie von Prozessoren werden C-Compiler zum Beispiel von der Firma Keil angeboten, der sich als quasi-Standard etablieren konnte. Ganz kostenfrei ist für Linux ein C-Compiler namens SDCC im Internet erhältlich. Eine Programmierung mit einer Hochsprache ist also möglich.

Mit diesem Prozessor als Herz wurde ein Standard-Rechner entwickelt, der zunächst in allen Modulen zu Anwendung kommen soll. Abbildung 3.2 gibt sein Blockschaltbild wieder. Der Einsatz eines vielfach eingesetzten und damit auch getesteten Rechners ist unproblematischer als eine ständige Neuentwicklung. Dies senkt den Zeit- und Kosten-Aufwand ganz erheblich.

Normalerweise wird das auszuführende Programm im Programmspeicher abgelegt. Für Daten muß ein extra Datenspeicher zur Verfügung stehen. In diesem Fall sollen EPROMs[4] als Programmspeicher zum Einsatz kommen. Der Inhalt dieser Bausteine ist jedoch vom Prozessor aus nicht änderbar. Sie werden mit einem externen Gerät, einem sogenannten EPROM-Brenner, beschrieben und müssen zum Löschen über einen längeren Zeitraum mit UV-Licht beaufschlagt werden. Um nun aber bei einer Änderung des Programms nicht immer das EPROM wechseln zu müssen, ist es sinnvoll, den Rechner so zu gestalten, daß er zur Laufzeit ein neues Programm aufnehmen und ausführen kann. Im EPROM wird nur ein Betriebssystem untergebracht, das genau dieses Laden und Ausführen von User-Programmen ermöglicht. Diese Funktionalität ist mit den gewählten Rechnern aber nicht ohne weiteres machbar, deshalb soll zum besseren Verständnis

[3] In etwa mit den 68HC11 Prozessoren von Motorola vergleichbar.
[4] Erasable Programmable Read Only Memory

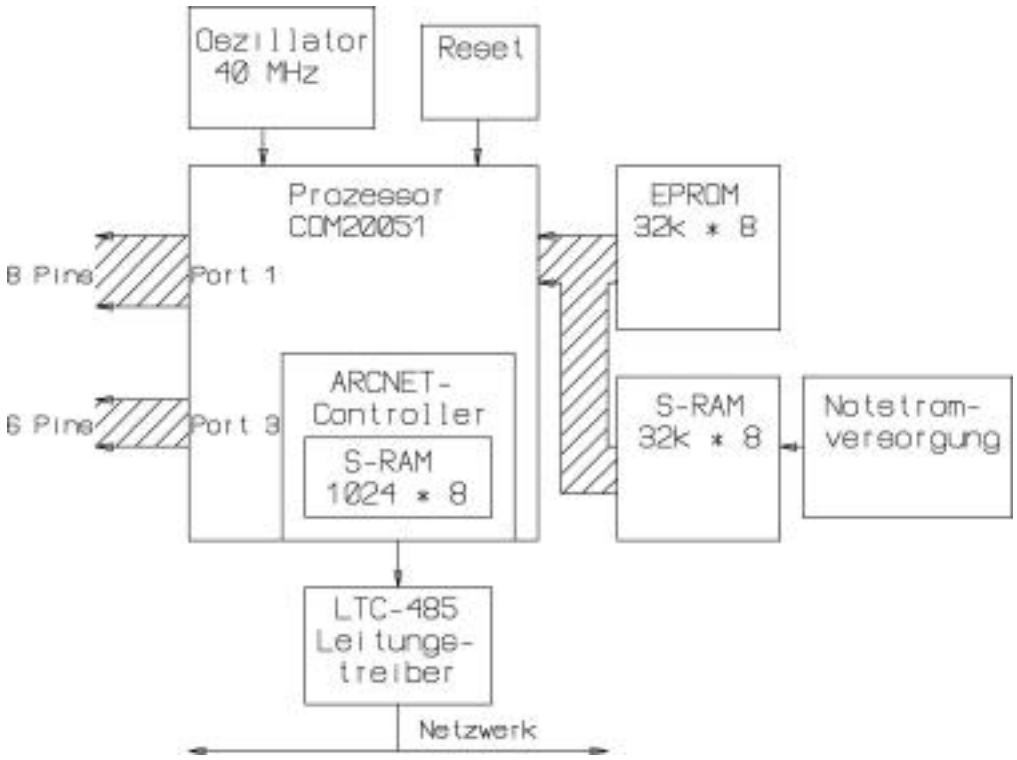


Abbildung 3.2.: Standard-Rechner

die verwendete Speicher-Konfiguration näher erläutert werden.

Wie bereits erwähnt, kann der Prozessor maximal 64KByte Programmspeicher und 64KByte Datenspeicher adressieren, was den Adressen 0x0000 bis 0xFFFF entspricht. Daten- und Programm-Speicher werden mittels eines Prozessorsignals (PSEN Program-Chip-Enable) unterschieden beziehungsweise aktiviert. Diese reine Harvard-Architektur in Abbildung 3.3 hat nun aber den Nachteil, daß das auszuführende User-Programm *komplett* im Programmspeicher stehen muß. Doch genau sein Inhalt läßt sich ja zur Laufzeit nicht ändern.

Eine kleine Zusatzschaltung schafft hier die nötige Abhilfe. Zunächst wird die Größe des Programm- und Daten-Speichers auf jeweils 32KByte begrenzt. Der Programmspeicher wird auf die Adressen 0x0000 bis 0x7FFF gelegt. In diesem Bereich findet das Betriebssystem Platz. Der Datenspeicher kommt auf die Adressen 0x8000 bis 0xFFFF, also die oberen 32KByte. Eine kleine Logikschaltung hebt nun die vorher so strikte Trennung der Speicherbausteine auf. Werden nun Adressen der oberen 32KByte angesprochen (also ab 0x8000) wird *immer* das RAM aktiviert, ganz egal ob es sich um Daten oder um das Holen des nächsten Befehl handelt. Die User-Programme können einfach in den oberen Teil des Speichers und damit im RAM abgelegt werden.

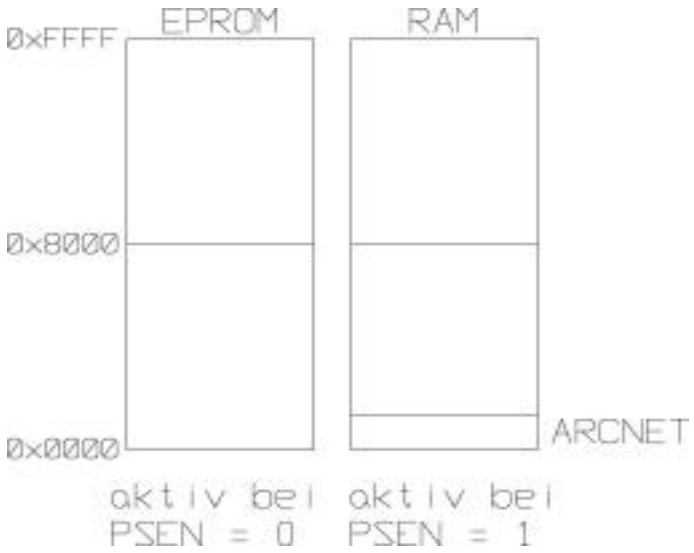


Abbildung 3.3.: Harvard-Architektur

Abbildung 3.4 zeigt die resultierende Speicheraufteilung und eine mögliche Konfiguration. Das Betriebssystem (BS) findet in den unteren Adressen (0x0000 bis 0x7FFF) Platz. Daten des Betriebssystems werden ab Adresse 0x8000, also im RAM abgelegt. Danach folgt ein Bereich, hier beispielsweise ab Adresse 0x8800 bis 0xBFFF, für das User-Programm. Erst dahinter (0xC000 bis 0xFFFF) finden die User-Daten Platz. Um das geladene Programm auszuführen wird nun einfach seine Startadresse, hier 0x8800, vom Betriebssystem angesprungen. Der ARCNET-Controller findet sich an Adresse 0x0000 im RAM-Bereich. Die Voraussetzungen, die seitens der Software nötig sind, um diese Architektur zu nutzen, wird später im Kapitel 3.4 erläutert.

Das RAM wird zusätzlich durch eine „Notstromversorgung“ mit Spannung versorgt. Diese Schaltung stellt sicher, daß die Daten und Programme im RAM auch nach dem Ausschalten des Roboters nicht gelöscht werden. Ein solcher Datenerhalt funktioniert über einen Zeitraum von etwa drei Monaten. Kurzzeitiger Betrieb des Roboters von etwa 5 Minuten innerhalb dieser Frist, lädt die Notstromversorgung wieder voll auf und sichert so die Daten für weitere drei Monate.

Der Oszillator versorgt den Prozessor mit einem 40 MHz Taktsignal. Intern wird dieses Signal direkt an den ARCNET-Controller weitergeleitet und für den Rechner selbst um Faktor 2,5 herunter geteilt, so daß sich seine Arbeitsfrequenz von 16 MHz ergibt.

Zur Anbindung an das Netzwerk benötigt der Prozessor lediglich einen LTC 485 Leitungstreiber der die Prozessor-Signale in ein erforderliches Differenzsignal umsetzt. Als eigentliche Netzwerkverbindung reichen zwei ungeschirmte Leiter aus. ARCNET hat sich in diesem Bereich als recht störunempfindlich und genügsam erwiesen, so daß es hier aus-

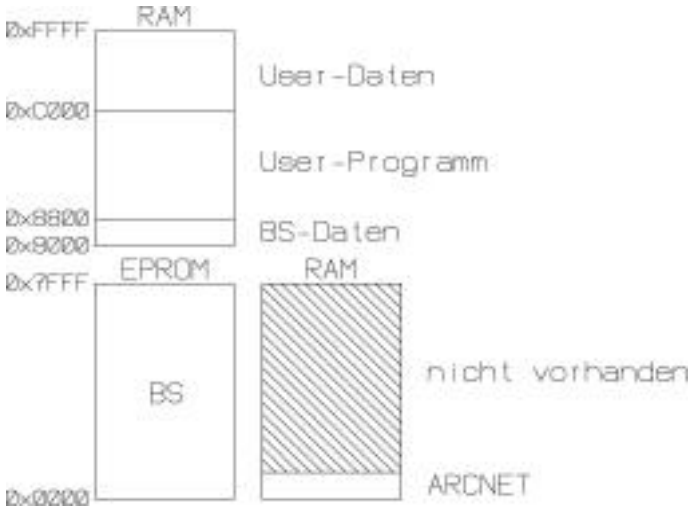


Abbildung 3.4.: Speicheraufteilung

reicht, ein herkömmliches 4-poliges Telefonkabel einzusetzen. Dies ist preisgünstig und leicht verfügbar.

Mechanisch paßt dieser Standard-Rechner auf eine halbe Euro-Platine. Die andere Hälfte steht so für Modul-spezifische Elektronik zu Verfügung, also beispielsweise Leistungselektronik für Motoren, Ansteuerung von Sensoren etc. Zur Ansteuerung stehen zwei Möglichkeiten zur Verfügung:

- 14 freie Pins des Prozessors.
- Einbindung der externen Elektronik in den Speicherbereich des Prozessors.

Der ersten Lösung sollte, wenn möglich, der Vorrang gewährt werden. Dies erübrigt eine sonst nötige Adreß-Dekodierung des Speicherbereichs und in vielen Fällen zeigt sich, daß die 14 Pins ausreichen. Aber es können auch beide Lösungen nebeneinander existieren.

Im folgenden wird dieser Standard-Rechner auch als Rechnereinheit bezeichnet. In den Blockschaltbildern wird auf die, hier noch detaillreiche Darstellung, zugunsten der Übersichtlichkeit verzichtet.

Das Netzwerk, das ja das Rückgrat des gesamten Systems bildet, wurde bisher nicht näher spezifiziert. Dies soll nun im nächsten Abschnitt geschehen.

3.2.2. Netzwerk

Bei dem Netzwerk handelt es sich um ARCNET nach dem ANSI 878.1 Standard (vgl.[18]). Dieses Netz gehört zu den sogenannten Feldbussen und arbeitet mit einem token-passing-Protokoll. Verwendung finden solche Netze vorzugsweise in der Automatisierungstechnik, deren oftmals rauhe Einsatzumgebungen ein hohes Maß an Robustheit und damit Zuverlässig abverlangt. Aber auch ihr relativ geringer Preis spricht für sie und ist ein wichtiges Argument für ihren Einsatz.

Auf eine ausführliche Beschreibung der Funktionsweise wird an dieser Stelle bewußt verzichtet und auf die einschlägige Literatur verwiesen (z.B. [13] oder [19]). Aber einige wichtige Eigenschaften sollen hier kurz skizziert werden:

1. Ein Datenpaket, das sogenannte „token“, wird unter den Rechnern herumgereicht. Der Rechner der das token zur Zeit besitzt, hat das alleinige Senderecht, was Kollisionen verhindert.

2. Jeder Rechner muß eine *eindeutige* Identifikationsnummer bekommen. Anhand dieser ID sortieren sich die Module in aufsteigender Reihenfolge zu einem logischen Ring.

3. Ein Rechner kennt nur seinen Nachfolger, also den mit der nächsthöheren ID. Datenpakete werden von Rechner zu Rechner weitergegeben, bis sie ihren Bestimmungsort erreicht haben.

4. Neue Teilnehmer können sich während der Laufzeit an- und abmelden und werden automatisch in den Ring eingefügt beziehungsweise entfernt.

5. Theoretisch können bis zu 255 Teilnehmer an das Netz angeschlossen werden. Die Wahl der Übertragungsgeschwindigkeit und des Mediums beschränken diese Zahl jedoch ganz erheblich. Für den Roboter wurde zunächst eine Übertragung mit $2{,}5\frac{Mbit}{s}$ und als Medium ein einfaches Telefonkabel gewählt, was die Anzahl der Teilnehmer laut Hersteller auf 32 begrenzt.

6. Alle Datenpakete werden vom ARCNET-Controller mittels einer 16-Bit großen Checksumme auf Richtigkeit hin überprüft und ggf. in sein eigenes RAM eingelagert. Fehlerhafte Daten werden automatisch abgelehnt, was dem Sender sofort mitgeteilt wird. All dies benötigt keinerlei Aktion seitens des Prozessors. Er muß lediglich ein Senden oder Empfangen initiieren und die Daten in beziehungsweise aus dem Controller-RAM heraus kopieren. Die eigentliche Kommunikation erledigt der ARCNET-Controller selbständig.

7. Die Schnittstelle zwischen ARCNET-Controller und Prozessor bilden 16 Register, die im Speicherbereich des Prozessor-RAM liegen. Ein Interrupt des ARCNET-

Controllers meldet dem Prozessor erfolgreiches Senden, Emfangen oder eventuelle Fehler.

8. Die maximale Übertragungsgeschwindigkeit beträgt $5\frac{Mbit}{s}$, aber auch geringere Geschwindigkeiten sind einstellbar bis minimal $3{,}125\frac{kbit}{s}$.

3.3. Module

Damit der Roboter seine Aufgabe erfüllen kann, muß er mit passender Elektronik ausgerüstet sein. Im Sinne eines verteilten Systems und der Idee kleine, intelligente Module einzusetzen, müssen mindestens folgende Bereiche abgedeckt werden, wobei in allen Modulen der Standard-Rechner zur Anwendung kommt:

- Aktorik um den Roboter bewegen zu können (Motorsteuerung).
- Sensorik zur Erfassung der Umwelt (Sonar).
- Hilfselektronik zur Energieversorgung (Laderechner).

Dies entspricht dem absoluten Minimalsystem. Die Steuerung und Kartographie müßten also diese Rechner mit erledigen. Vor dem Hintergrund, daß es sich aber um kleine Rechner mit geringer Rechenleistung handelt und gerade Aktorik und Sensorik zeitkritische Algorithmen benötigen, ist es sinnvoll weitere Module einzuführen. Ganz allgemein soll für jede Aufgabe, die vernünftig teilbar erscheint, ein eigenes Modul eingeführt werden.

- Steuermodul (Hauptrechner).
- Kartographiemodul (Kartenrechner und Coprozessor).
- Schnittstelle zwischen Mensch und Maschine (LC-Display und Tastatur).
- Weitere Sensorik um mehr Daten über die Umwelt zu erhalten (Kompaß).

3.3.1. Motorsteuerung

Eine beliebte Methode für den Antrieb sind zwei normale Gleichstrom-Motoren, wie sie im Modellbau verwendet werden. Solche Motoren benötigen ein Getriebe, um ihre hohe Drehzahl in nutzbare Bereiche zu bringen. Zusätzlich ist eine Elektronik nötig, die die Motorströme regelt und so eine präzise Ansteuerung der Motoren ermöglicht. Eine Odometrie an den Ketten macht die zurückgelegte Strecke des Roboters berechenbar. Alles in allem ist dies aber ein großer Aufwand...

Hier stellt sich die Frage ob es nicht auch einfacher geht und Schrittmotoren bieten sich als Alternative an. Normalerweise werden diese Motoren für Positionierungsaufgaben eingesetzt, für leistungsstarke Antriebe sind sie eigentlich nicht ausgelegt. Ihr Vorteil der schrittweisen und damit berechenbaren Drehung, macht eine Odometrie überflüssig. Die Drehzahl ist direkt über die Schrittgeschwindigkeit, die vom Rechner vorgegeben wird, steuerbar. Ein Getriebe, um die Drehzahl herauf- oder herabzusetzen, erübrigt

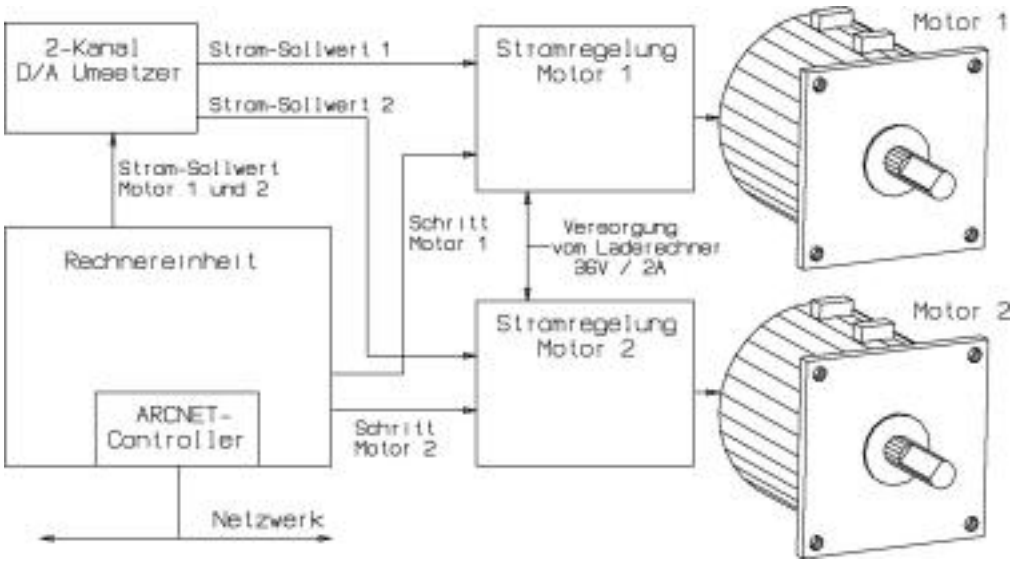


Abbildung 3.5.: Motorsteuerung

sich somit. Es wird nur dann nötig, wenn das erforderliche Drehmoment im direkten Antrieb nicht geliefert werden kann.

Kern der Motorsteuerung namens „ROBMOTOR“ bildet der Standard-Rechner und zwei Stromregelkreise für die Motoren. Der Rechner kann über einen 2-kanaligen 8-Bit D/A-Umsetzer deren Ströme separat vorgeben. Die Regelung ist so kalibriert, daß ein Schritt des D/A-Umsetzer einem effektiven Strom von 20mA entspricht. So ist ein Bereich von 20mA bis 5,1A relativ fein einstellbar. In der Praxis haben sich Ströme von 0,8A bis 1,2A pro Motor als ausreichend erwiesen, da oberhalb dieser Werte das Drehmoment nicht mehr nennenswert ansteigt und ein höherer Stromverbrauch natürlich auch die Laufzeit des Roboters verkürzt. Mit einem mittleren Strom von 1A liegt die Laufzeit bei etwa 45 Minuten.

Zusätzlich zum Strom benötigt die Leistungselektronik noch den Schritt des jeweiligen Motors. Hierbei handelt es sich um eine 4-Bit breite Kombination, die den zu aktivierenden Spulen im Motor entspricht. Es kann zwischen einem Voll- und einem Halbschritt-Betrieb gewählt werden, was bei den eingesetzten Motoren einem Drehwinkel von $1,8°$ beziehungsweise $0,9°$ pro Schritt entspricht. Die Motoren sollten jedoch wegen der genaueren Positionierbarkeit im Halbschritt-Modus angesteuert werden. Diese Betriebsart wird deshalb im weiteren als Berechnungsgrundlage angenommen.

Bei einem Durchmesser des Treibrades inklusive Kette von 69mm errechnet sich pro Umdrehung eine gefahrene Strecke von 216,77mm. Die Drehzahl ergibt sich direkt aus der Anzahl der ausgegebenen Schritte pro Sekunde und die Drehrichtung aus der Reihenfolge

der Schritte.

Ein wichtiger Aspekt sind natürlich die erreichbaren Geschwindigkeiten. Maßgeblichen Einfluß darauf hat das Gewicht des Roboters, also die zu bewegende Last, die momentan bei etwa 9kg liegt (vgl.[16]). Die hier angegebenen Geschwindigkeiten sind somit nur für die derzeitige Konfiguration gültig. Grundsätzlich müssen zunächst zwei Betriebsbereiche unterschieden werden:

1. Start- Stop-Betrieb wird der Bereich genannt, bei dem ein Schrittmotor ohne Schritte zu verlieren „sofort“ seine Solldrehzahl erreicht. Hier liegt der Maximalwert bei $0{,}01\frac{m}{s}$.

2. Höhere Drehzahlen können nicht mehr so einfach erzielt werden. Erst ein definiertes Beschleunigen und Verzögern machen Geschwindigkeiten bis zu $1\frac{m}{s}$ möglich.

Der Benutzer sollte die Ströme auf etwa 1,2A pro Motor begrenzen und nicht auf Dauer überschreiten, da die Motoren sonst überlastet werden könnten und dies im schlimmsten Fall ihre thermische Zerstörung zur Folge hat. Eine zu hohe Ansteuerfrequenz ist dagegen völlig unbedenklich. Die Motoren bleiben höchstens stehen. So ist es dem Benutzer überlassen, welche Drehzahlen, Drehrichtungen und Verbrauchswerte er fahren möchte.

3.3.2. Sonar

Ein System, das sich in realer Umgebung bewegen möchte, muß eine Möglichkeit bekommen, sich ein „Bild“ von ihr zu machen. Es muß also mit einer Sensorik ausgestattet werden, die Informationen über die Umwelt sammelt. Hier bieten sich einige Möglichkeiten an:

- Kontaktschalter
- Infrarotsensoren
- Laserscanner
- Kameras
- Sonar

Kontaktschalter sind die wohl einfachsten und billigsten Sensoren für einen Roboter. Doch sie haben den Nachteil, daß sie nur bei Berührung mit Objekten einen Meßwert ausgeben. Der Roboter muß also ständig mit seiner Umwelt kollidieren, um sich ein Bild

von ihr zu machen. Das sieht in der Praxis zwar lustig aus, ist aber keine sehr elegante Methode.

Infrarot-Sensoren senden Lichtimpulse aus und messen das reflektierte Licht mittels eines Phototransistors. Theoretisch lassen die Laufzeit des Lichtes und die Intensität der Reflektion einen Rückschluß auf die Entfernung zu. Da aber eine Laufzeitmessung des Lichtes wegen der hohen Ausbreitungsgeschwindigkeit nahezu unmöglich ist, bleibt nur die Messung der Reflektionsintensität. Diese ist im Grunde recht einfach machbar, aber die Meßwerte sind stark von der Oberfläche des Objektes abhängig. In einer realen Umgebung jedoch, sind die Reflektionsgrade so stark schwankend, daß verläßliche Meßwerte kaum zu erwarten sind. Lediglich in einer Laborumgebung mit homogenen Oberflächen könnte eine solche Sensorik vernünftig arbeiten. Es gibt allerdings einen Sensor der Firma Sharp, den GP2D02, der anhand des Winkels des reflektierten Lichtes die Entfernung bestimmt und recht brauchbare Ergebnisse liefern soll.

Laserscanner tasten mit einem Laserstrahl ihre Umgebung ab. Sie sind relativ teuer und kommen daher für dieses Projekt leider nicht in Frage.

Eine sehr interessante Variante, die mit dieser Methode arbeitet, und auch preislich interessant werden dürfte, stellt ein neuer Sensor der Firma Siemens dar. Dieser kann ein 1000 Bildpunkte großes 3-dimensionales Bild in einer Millisekunde erfassen. Die Auflösung soll bei etwa 1cm liegen. Zur Zeit scheint dieser Sensor noch nicht auf dem Markt zu sein, aber in Zukunft könnte er eine echte Alternative zu den bisherigen Systemen darstellen (vgl. [23]).

Kameras stellen die wohl interessanteste Sensorik dar. Ob als Einzel- oder Stereo-Kamera bieten sie die wohl umfangreichsten Informationen ihrer Umwelt. Genau hier liegt aber auch ihr „Problem“. Da diese Informationen ausgewertet werden müssen, ist ein erheblicher Rechenaufwand notwendig, der nur mit entsprechend leistungsstarkem Rechner zu bewältigen ist und genau hier wird es teuer. Nicht zuletzt machen die komplexen Algorithmen diese Aufgabe so schwierig.

Sonar (Sound Navigation And Ranging) nutzt Schallwellen, um Richtung und Entfernung von Objekten zu ermitteln. Als Grundlage dieser Berechnung dient die Laufzeit des Schalls. Dazu wird ein Schallimpuls, ein sogenannter **Ping**, ausgesendet. Die Zeit bis zum Eintreffen eines Echos wird gemessen.

Äußerst beliebt sind fertige Sensoreinheiten der Firma Polaroid. Sie arbeiten mit nur einem Sender, der nach dem Abstrahlen des Schallimpulses als Empfänger dient. Während des Sendens kann so für eine bestimmte Zeit keine Messung erfolgen, was einen „blinden“ Bereich zur Folge hat, also einen Bereich, in der das Sonar keine Objekte orten kann. Um dies zu umgehen, wurden Sensoren gewählt, bei denen Sender und Empfänger baulich getrennt sind. Dies ermöglicht eine Messung auch während des Sendens.

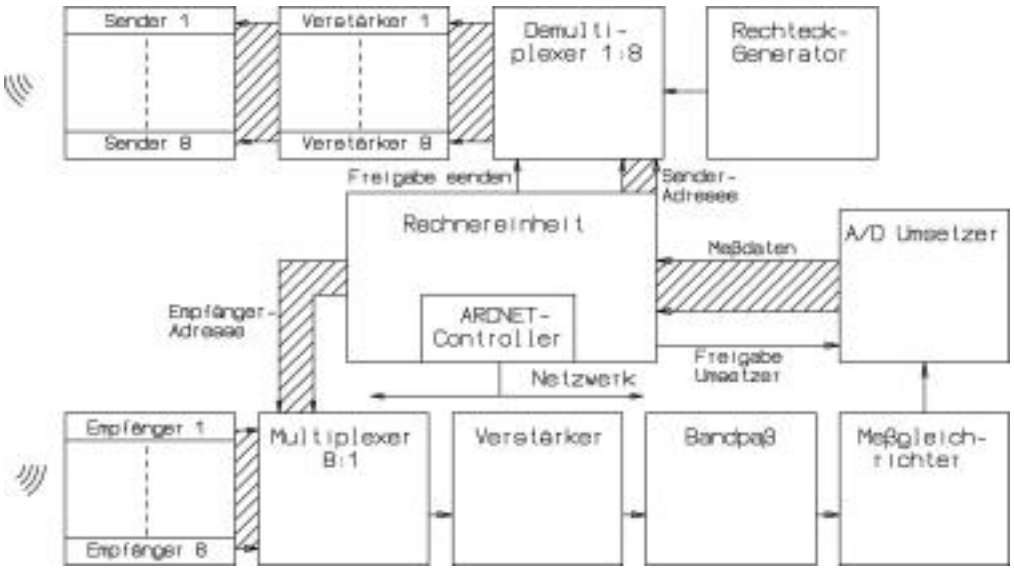


Abbildung 3.6.: Sonar

Zum Einsatz kommen hier Sensoren, die unter anderem bei der Firma Reichelt zu beziehen sind. Sie arbeiten mit einer Frequenz von 41kHz, also im Ultraschallbereich, und sind im Roboter immer als Sender-Empfänger-Paar angeordnet. Insgesamt werden acht solcher Sensor-Paare von der Schaltung unterstützt, was eine Rundumabtastung des Roboters ermöglicht.

Die Schaltung in Abbildung 3.6 ist in drei Teile aufgeteilt:
Die **Sendeeinheit**. Das Signal des Rechteckgenerators, der eine Frequenz von 41 kHz erzeugt, wird über einen 1:8 Demultiplexer zu acht Verstärkern geführt. Diese Verstärker steuern die Ultraschallsender an. Je nach eingestellter Adresse und aktivem Signal „Freigabe senden“ erhält nur einer der Verstärker das Rechtecksignal und versetzt seinen Sender in Schwingung.

Die **Empfangseinheit**. Die Ultraschallsignale werden von den Empfängern aufgenommen und einem 8:1 Multiplexer zugeführt. Je nach eingestellter Adresse am Multiplexer wird nur das Signal eines Empfängers weitergeleitet; und das ist immer der Empfänger, der mit dem aktiven Sender ein Paar bildet. Nach dem Multiplexer wird das Signal verstärkt und mittels eines Bandpasses gefiltert. Nötig ist dies, da die Empfänger auch auf mechanische Schwingungen außerhalb ihrer Arbeitsfrequenz reagieren und diese die Messung unbrauchbar machen würden. Ein Meßgleichrichter, der gleichzeitig als Tiefpaß arbeitet, macht aus den Sinusschwingungen eine Hüllkurve (Abbildung 3.7 zeigt eine solche Hüllkurve gemessen mit einem Oszilloskop). Der nachfolgende A/D-Umsetzer wandelt die Signale in digitale Werte um, die der Rechner nun einlesen und weiter verarbeiten kann.

Die **Rechnereinheit** stellt das zentrale Organ dar. Sie wählt die Adresse des aktiven Sender-Empfänger Pärchens aus, startet mit dem Signal „Freigabe senden“ die Schallabstrahlung und setzt mit dem A/D Umsetzer die Meßwerte in digitale Werte um, die nun zur Auswertung bereit stehen.

Eine Messung geht folgendermaßen vor sich:

1. Einstellen des gewünschten Sender-Empfänger-Paares mittels ihrer Adresse.
2. Starten der Schallabstrahlung mit dem Signal „Freigabe senden“ und gleichzeitiges Starten eines Timers (Stoppuhr).
3. In regelmäßigen Abständen den A/D-Umsetzer aktivieren und den Meßwert einlesen.
4. Ist eine ausreichende Anzahl Schallwellen abgestrahlt worden, das Signal „Freigabe senden“ deaktivieren und so den Sender wieder verstummen lassen.
5. Die Meßwerte des Empfängers werden weiterhin periodisch eingelesen. Wird ein Echo detektiert, zum Beispiel durch Überschreiten eines bestimmten Schallpegels, den Timer stoppen und die abgelaufene Zeit speichern. Abbildung 3.7 zeigt ein typisches Echo eines Objektes im Abstand von etwa 25cm.
6. Der Empfänger bleibt so lange aktiv, bis die maximale Meßweite erreicht wurde, also eine ausreichend lange Zeit verstrichen ist.

Die eigentliche Messung ist nun beendet und mit der bekannten Schallgeschwindigkeit von $330\frac{m}{s}$ in der Luft und dem Wert des Timers, also der verstrichenen Zeit, kann nun die Entfernung berechnet werden:

$$\boxed{s_{abs} = \frac{v_{Schall} * t_{Timer}}{2}} \tag{3.1}$$

Ein besonderes Problem hierbei stellt das sogenannte **falsche Echo** dar. Hierbei handelt es sich um Schallwellen die direkt vom Sender in den Empfänger gelangen, also keine echte Reflektion darstellen. Die Amplitude dieses Signals ist extrem hoch und bedarf normalerweise einer besonderen Behandlung bei der Auswertung der Meßwerte. Eine mechanische Besonderheit der Sensorhalterung schafft hier Abhilfe und reduziert das falsche Echo bis es im allgemeinen Rauschen untergeht. Doch zunächst zu dem physikalischen Effekt, der hier „schamlos“ ausgenutzt wird.

Die Sender bilden durch Interferenz Schallkeulen, also Bereiche in denen sehr hohe Schallenergie beziehungsweise auch extrem niedrige Energie auftritt[5]. Auch die Empfänger

[5]Die Theorie hierzu findet sich z.B. in [1]

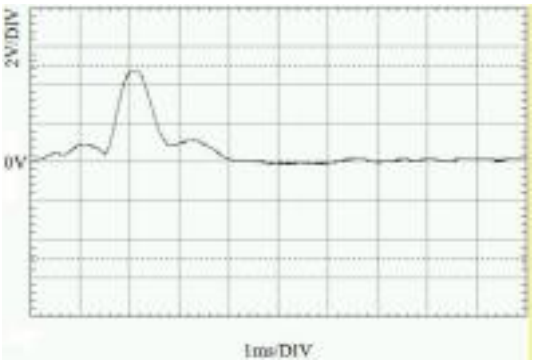
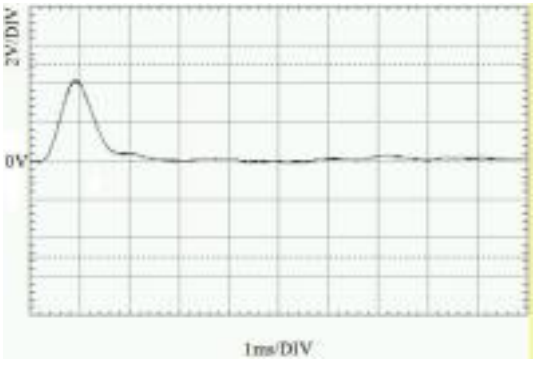
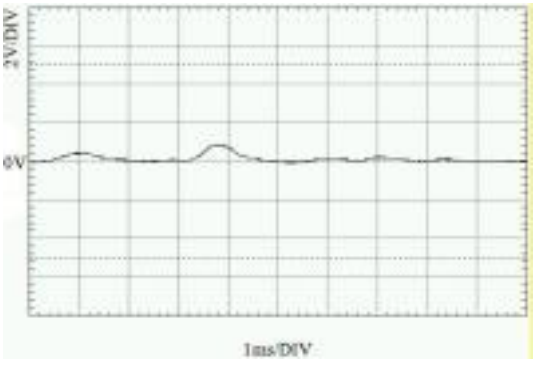


Abbildung 3.7.: Das Echo eines Objektes im Abstand von ca. 25cm

zeigen ein solches Verhalten. Bei ihnen bilden sich Bereiche, in denen sie sehr empfindlich beziehungsweise auch extrem unempfindlich sind. Es zeigt sich, daß die Ausrichtung des Senders zum Empfänger maßgeblich an der Entstehung des falschen Echos beteiligt ist. Durch eine Drehung der beiden und gleichzeitige Messung der Amplitude können die Sensoren so ausgerichtet werden, daß gerade ein Schallminimum des Senders in Richtung des Empfängers weist und dieser in Richtung Sender ein Minimum an Empfindlichkeit aufweist. Dies funktioniert so gut, daß das übrigbleibende falsche Echo keine Rolle mehr spielt. Abbildung 3.8.1 zeigt die Hüllkurve des falschen Echos vor der Kalibrierung und Abbildung 3.8.2 danach. Die Sensoren lassen sich hierzu in ihrer Halterung drehen und nach erfolgter Kalibrierung mit einer Schraube endgültig fixieren.



3.8.1: falsches Echo



3.8.2: eliminiertes falsches Echo durch Kalibrierung

Abbildung 3.8.: Problem des falschen Echos

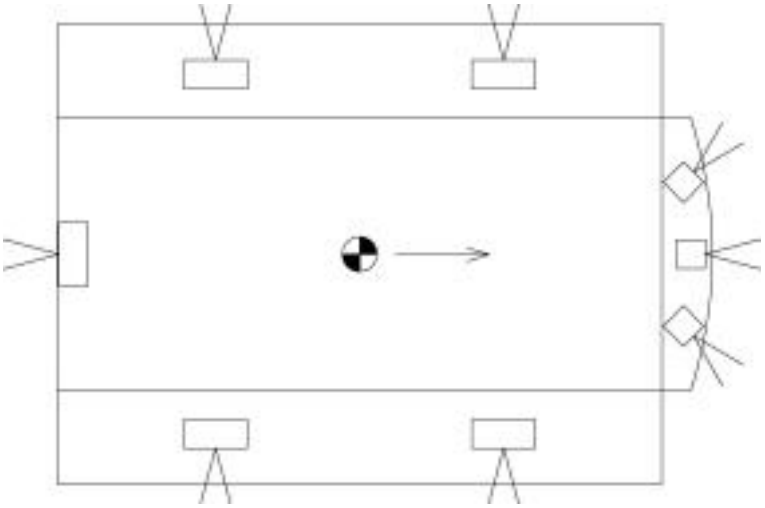


Abbildung 3.9.: Anordnung der Sonar-Sensoren

Mechanisch wurden die Sensoren so angeordnet, daß insgesamt drei den vorderen Bereich abtasten. Zwei messen rechts und links in einem Winkel von je 45°, der dritte weist direkt nach vorne. Eine Sensor mißt direkt nach hinten und je zwei Sensoren tasten die Seiten links und rechts ab. Abbildung 3.9 zeigt den Roboter schematisch von oben. Der Pfeil weist nach vorne.

Das Sonar stellt sich als kostengünstiges, einfach zu entwickelndes und robustes Meßinstrument heraus. Die breiten Meßkeulen der Sensoren von ca. 30° verhindern jedoch eine feine Richtungsauflösung, die wirklich wünschenswert wäre. Hier sind mit mehr Aufwand sicherlich bessere Ergebnisse erreichbar, aber vermutlich ist dann die Wahl eines anderen Sensor-Typs die bessere Alternative. Weiterführende Informationen finden sich in [1], [2] und [4].

3.3.3. Kompaß

Um mehr Daten über die Umwelt zu erhalten, schien es sinnvoll einen weiteren Sensor für diese Aufgabe einzuführen. Dieser Sensor sollte gänzlich anders arbeiten als die bisher eingesetzten, um systembedingte Fehler zu vermeiden. Zudem sollte der Sensor einen Wert liefern, der unabhängig von der Laufzeit des Roboters ist. Hier bietet sich ein Kompaß an.

Kreisel, auch Gyroskop, werden oftmals im Zusammenhang mit Kompanden genannt. Doch eigentlich arbeiten sie nach einem völlig anderen Prinzip. Systembedingt sind sie nicht in der Lage einen *absoluten* Winkel zum Nordpol zu ermitteln. Sie liefern ein Signal,

daß ihrer momentanen Drehgeschwindigkeit entspricht. Der eigentlich erwünschte Winkel wird erst durch eine Kalibrierung und Integration der Meßwerte über die Zeit ermittelt. Doch genau hierbei macht man zwangsläufig einen kumulativen Fehler, der von sich aus nicht mehr zu korrigieren ist. Einfache Kreisel scheiden somit aus und werden deshalb nicht weiter erörtert.

Kreiselkompanden haben den Vorteil, daß sie immer den *geographischen* Nordpol anzeigen und das unabhängig von ihrer Umgebung. Sie nutzen den Effekt, daß sich ein frei drehender Kreisel aufgrund der Erddrehung immer in Nord- Süd-Richtung ausrichtet. Aber auch sie haben Nachteile. Unter ca. 20.000 DM und einer Masse von etwa 15kg scheint kein Kompaß dieser Art erhältlich zu sein. Auch ihr erheblicher Energieverbrauch spricht gegen sie. Diese Kompanden kommen somit leider nicht in Frage.

Magnetische Kompanden dagegen sind klein, billig und sparsam im Verbrauch. Aber sie lassen sich durch magnetische Anomalien ablenken, was ihrer Genauigkeit erheblich schadet. Trotzdem scheinen sie die einzige realisierbare Technik für dieses Projekt zu sein.

Die Himmelsrichtungen werden hier, anders als in der Mathematik, mit rechtsdrehend steigenden Winkeln festgelegt, wobei 0° Norden entspricht. Dies ist die in der Navigation übliche Weise und wird hier und im folgenden durchgängig verwendet. Abbildung 3.11 zeigt eine solche Kompaßrose.

Doch nun zur verwendeten Technik. Abbildung 3.10 zeigt das Blockschaltbild des Kompaß namens „ROBCOMP“.

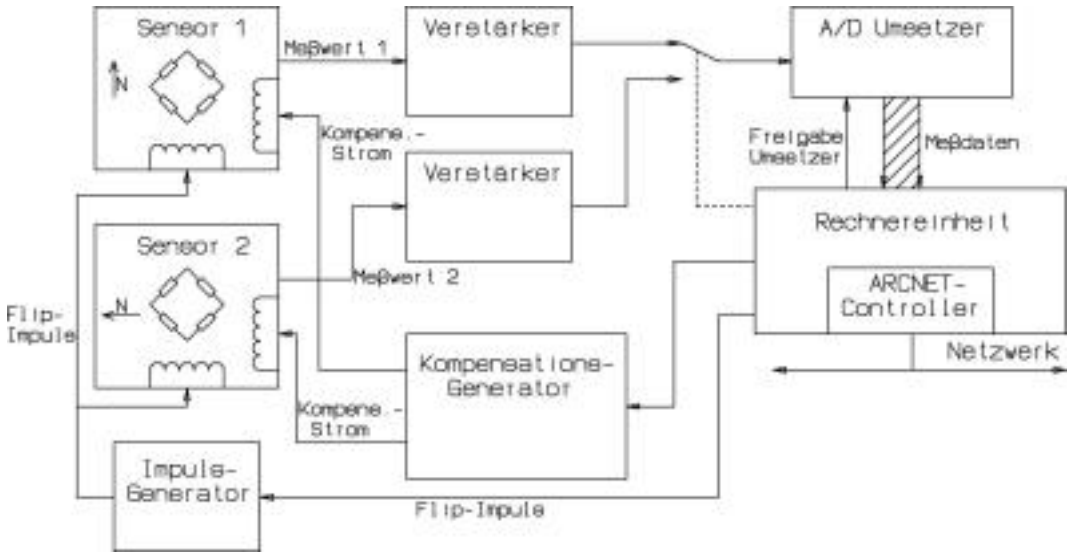


Abbildung 3.10.: Kompaß

Herzstück der Schaltung bilden zwei Magnetfeldsensoren KMZ51 von Philips. Diese Sensoren sind derart empfindlich, daß sie das schwache Erdmagnetfeld, das typischerweise zwischen $38\mu T$ und $63\mu T$ liegt, detektieren können. Intern bestehen sie aus einer Sensorschicht, deren Leitfähigkeit sich proportional zum Winkel des äußeren Magnetfeldes ändert. Mit diesen Sensorschichten ist intern eine Meßbrücke aufgebaut. Bei einer Drehung des Sensors um 360° ist nun eine sinusförmige Differenzspannung meßbar. Mittels der Arcus-Cosinus Funktion kann nun aus dem Meßwert der zugehörige Winkel errechnet werden.

Zusätzlich sind in jedem Sensor zwei Spulen integriert. Die eine, sogenannte „Flip-Spule“, dient zur Ummagnetisierung der Sensorschicht, was wie eine Drehung des Sensors um 180° wirkt und so eine Differenzmessung möglich macht. Mit Hilfe der zweiten „Kompensationsspule“ kann ein zusätzliches Magnetfeld mit beliebiger Richtung und Intensität erzeugt werden. Sie ist nötig, um Offset-Effekte im Sensor zu eliminieren, die durch Temperaturänderungen sowie bei Eintreten in magnetische Anomalien auftreten. Beide Spulen spielen später bei der eigentlichen Messung ganz wesentliche Rollen.

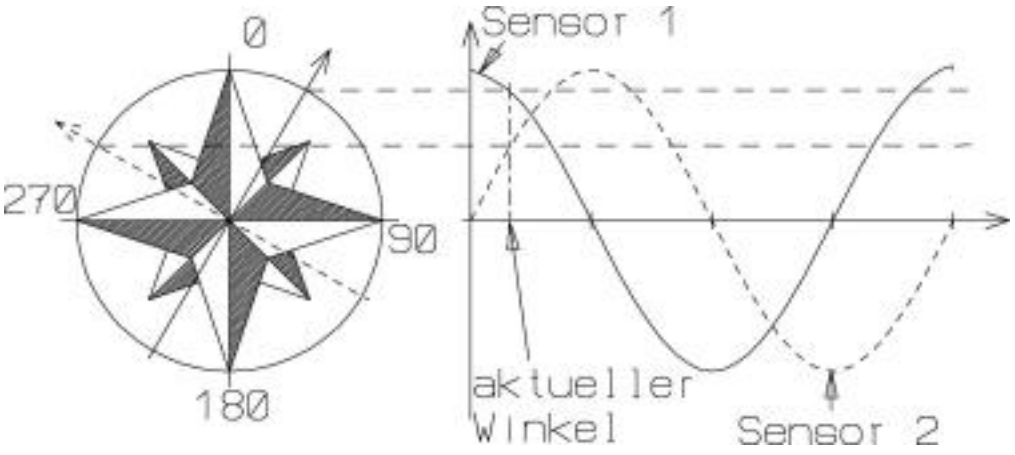


Abbildung 3.11.: Messung mit zwei Sensoren

Daß hier zwei Sensoren zum Einsatz kommen hat einen Grund. Wegen der Periodizität der Sinusspannung ist eine Winkelbestimmung im Vollkreis mit nur einem Sensor unmöglich. Zu einem gemessenen Sensorwert finden sich immer zwei passende Winkel. So würde beispielsweise der Sensorwert 0 bei den Winkeln 90° sowie 270° auftreten (Abbildung 3.11). Außerdem sind im Bereich der Maxima, bedingt durch die geringe Steigung der Funktion, bei einer Winkeländerung nur extrem kleine Spannungsänderungen vorhanden, die nur schwer zu ermitteln und damit natürlich extrem fehlerträchtig sind.

Aus diesen Gründen sind zwei Sensoren nötig, die um 90° versetzt sind. Erstens ist

nun eine Winkelbestimmung möglich, da beide Sensorwerte zusammen den zugehörigen Winkel *eindeutig* identifizieren. Und zweitens bietet sich die Chance, immer nur den Sensorwert zur Berechnung des Winkels heranzuziehen, der sich im Bereich um 0 herum $\pm 45°$ aufhält. Hier hat eine kleine Winkeländerung auch große Spannungsdifferenzen zur Folge. Verläßt der Sensor diesen Bereich (Winkel), wird der andere Sensor ausgewertet.

Nun zur Funktionsweise der eigentlichen Messung anhand eines Sensors. Die Ermittlung des zweiten Sensorwertes erfolgt analog. Zum besseren Verständnis werden als Sensorwerte nicht die echten Spannungswerte angenommen, sondern die reine Sinusfunktion. Weist der Sensor Richtung Norden ergibt das den Wert 1, Süden -1 und Ost- beziehungsweise West-Ausrichtung den Wert 0. Bei diesem Beispiel wird davon ausgegangen, daß der Sensor Richtung Norden schaut.

Zunächst wird der aktuelle Wert mittels A/D-Umsetzer gemessen und gespeichert. Danach wird der Sensor mit der Flip-Spule ummagnetisiert. Er mißt nun den um 180° versetzten Winkel, in diesem Beispiel also Richtung Süden. Dieser Wert wird nun ebenfalls ermittelt und gespeichert. Die gemessenen Werte sollten theoretisch 1 und -1 betragen. Ist dies nicht der Fall ist ein Offset aufgetreten, der aus den Differenzen (Deltas) beider Werte zum Nullpunkt berechenbar ist. Nun muß mit der Kompensationsspule ein Magnetfeld erzeugt werden, daß genau diesem Versatz entspricht und ihn so eliminiert. Danach werden die Messungen wiederholt. Beide Werte sollten nun symmetrisch um Null herum liegen und 1 beziehungsweise -1 betragen. Ist dies der Fall, reicht der erste Wert, um mit der Arcus-Cosinus-Funktion den Winkel zu errechnen (bei arccos(1) ist das genau Norden also 0°), ansonsten muß erneut kompensiert werden.

Dies klingt in der Theorie recht einfach, in der Praxis jedoch gibt es erhebliche Probleme:

- Die Sensorwerte sind nicht so statisch wie es vielleicht zu erwarten wäre. Äußere Wechselfelder schlagen sich direkt im Sensorwert nieder und müssen per Software herausgefiltert werden.
- Die Motoren im Roboter erzeugen, im Vergleich zum Erdmagnetfeld, sehr große statische Magnetfelder. Sie müssen exakt bestimmt und von vornherein kompensiert werden.
- Hinzu kommen die Magnetfelder der Motoren, die bei der Fahrt erzeugt werden.
- Das Magnetfeld der Kompensationsspule beeinflußt die Amplitude der Sensoren. Daher muß ihr Magnetfeld bekannt beziehungsweise berechenbar sein, um dies als Faktor in die Rechnung mit einzubeziehen.
- Die Frequenz, mit der die Sensoren ummagnetisiert werden, hat direkten Einfluß auf das Verhalten der Sensoren. Sie darf nicht zu hoch sein, um nicht zu messen wenn die Spule noch Restenergie besitzt, aber auch nicht zu gering sein, da die Sensoren dann an Empfindlichkeit verlieren.

Hier zeigt sich, daß eine exakte Kalibrierung notwendig ist, um dem Kompaß sinnvolle Meßwerte zu entlocken. Ein früherer Prototyp hat aber gezeigt das ein solches Gerät prinzipiell machbar und sogar eine Winkelauflösung von 1° zu erreichen ist.

Nähere Informationen finden sich in [20] beziehungsweise [21].

3.3.4. Hauptrechner

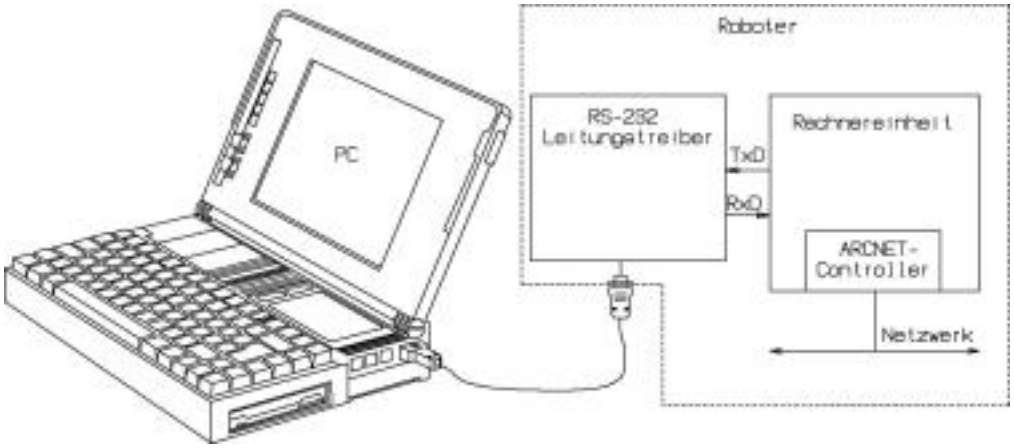


Abbildung 3.12.: Hauptrechner

Der Hauptrechner bildet die Schnittstelle zwischen Roboter und Personal-Computer, auf dem die Software für den Roboter entwickelt wird. Hierzu besitzt „ROBMAIN“ eine galvanisch getrennte RS-232 Schnittstelle, die mit einer Übertragungsrate von $9600\frac{bits}{s}$ arbeitet. Über diese werden die Programme für jedes einzelne Modul herunter geladen.

3.3.5. Coprozessor und Kartenrechner

Coprozessor und Kartenrechner bestehen lediglich aus dem Standard-Rechner und sind auf einer gemeinsamen Platine untergebracht. Da die Rechnereinheit bereits in Kapitel 3.2.1 ausführlich bschrieben ist, wird an dieser Stelle auf eine nähere Beschreibung verzichtet.

Der Kartenrechner soll, nach seinem Namen „ROBMAP“, später die Karte aufnehmen und verwalten. „ROBCOPRO“ dagegen wurde mit integriert, da auf einer Euro-Platine problemlos zwei Standard-Rechner unterzubringen sind und etwas mehr Rechenleistung im System zur Verfügung zu haben nicht schaden kann. Seine eigentliche Aufgabe ist nicht festgelegt sondern er wird Teilaufgaben anderer Module übernehmen, die sonst überlastet wären.

3.3.6. LC-Display und Tastatur

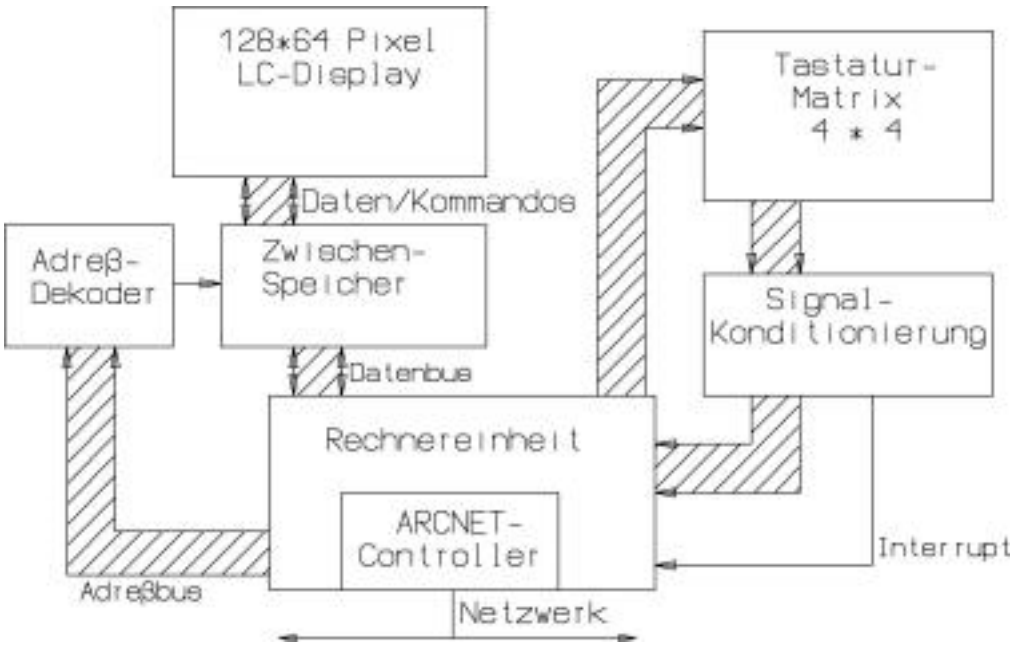


Abbildung 3.13.: Display und Tastatur

Das LC-Display ist als Schnittstelle zwischen Mensch und Maschine gedacht. Der Benutzer soll hier Werte einstellen, abrufen, den Roboter starten und dergleichen mehr können. Hierzu sind zwei Dinge für „ROBLCD“ unbedingt erforderlich:

- Eine **Tastatur**. Da es hier in erster Linie um die Eingabe numerischer Daten und nicht um die eigentliche Programmierung des Roboters geht, reicht eine kleine Tastatur, die die zehn Ziffern und ein paar Zusatztasten umfaßt. Gewählt wurde hier eine $4 * 4$ Tastaturmatrix, also insgesamt 16 Tasten.

  Diese wird mittels der bei jedem Standard-Rechner freien Pins angesteuert. Sobald eine oder mehrere Tasten betätigt werden, meldet dies ein Interrupt dem Prozessor, der nun die betätigten Tasten ermitteln kann.

- Ein **Display** zur Anzeige von Daten. Theoretisch, solange es um die Anzeige einzelner Werte geht, wäre hier ein alphanumerisches Display völlig ausreichend. Da aber die Kartographie im Vordergrund steht, ist es wünschenswert auch die aufgezeichnete Karte anzeigen zu können. Hier scheint ein Grafikdisplay erforderlich. Deshalb wurde ein $128 * 64$ Pixel umfassendes Grafikdisplay integriert.

  Seine Ansteuerung wurde mit einem kleinen Adreßdekoder in den Speicherbereich des RAM integriert, wobei Befehle an die Adresse 0x2000 und Daten an die Adresse

0x4000 geschrieben werden müssen. Der Zwischenspeicher ist erforderlich, weil das gewählte Display die Daten auf dem Bus nicht so schnell übernehmen kann, wie es erforderlich wäre. Der Zwischenspeicher „teilt“ im Grunde den Bustakt auf die Hälfte herunter.

3.3.7. Laderechner

Die Aufgabe des Laderechners ist, das gesamte System mit Energie zu versorgen und die Akkumulatoren vor einer Tiefenentladung zu schützen. Droht diese, kann er den gesamten Roboter ausschalten. Nötig ist dies, da Blei-Akkumulatoren bei Tiefenentladung[6] mit extremem Kapazitätsverlust reagieren. Den ARCNET-Anschluß soll er eigentlich nutzen, um die noch verbleibende Energie in den Akkus abzuschätzen und den anderen Modulen mitzuteilen. Diese können dann entscheiden ob sie noch weiter fahren wollen oder die Ladestation angefahren werden soll. Da diese Ladestation aber zur Zeit noch nicht fertiggestellt ist, wurde auf eine Integration des Laderechners in das Netzwerk verzichtet. Er überwacht also nur die Akkus und schaltet im Notfall alles ab.

Doch nun zur Funktionsweise von „ROBLOAD“ anhand seines Blockschaltbildes in Abbildung 3.14.
Die Spannungen der Akkus ($+12V$ und $+36V$) werden über je ein Hochstromrelais dem Laderechner zugeführt. Zwei Sicherungen mit je 2A Mittelträge sollen allzu große Katastrophen bei Kurzschlüssen abwenden. Beide Spannungen werden als Meßspannungen an einen A/D-Umsetzer geleitet, mit dessen Hilfe der Rechner die aktuellen Akkuspannungen ermitteln kann. Die 36V sind nur zur Versorgung der Motoren gedacht und werden deshalb gleich an die Leistungselektronik der Motorsteuerung weitergeleitet.

Über einen Ein-Taster werden die 12V direkt an die Ansteuerung der Relais geführt. Ziehen diese an, werden auch die beiden DC/DC-Wandler aktiv. Der eine erzeugt +5V für alle im Roboter befindlichen Rechner und der zweite ±15V für analoge Schaltungen wie etwa Operationsverstärker und ähnliches. Der Laderechner kann nun anfangen zu arbeiten und überwacht die beiden Akkuspannungen.

Sind diese „im grünen Bereich“, so aktiviert er ein Selbsthaltesignal, daß die Relais parallel zum Ein-Taster ansteuert. Der Taster kann nun losgelassen werden. Erst wenn der Rechner eine Tiefenentladung einer der Akkus detektiert, nimmt er die Selbsthaltung zurück, wodurch die Relais abfallen und die gesamte Spannungsversorgung zusammenbricht. Der Roboter sollte nun wieder aufgeladen werden.

Der Aus-Taster hat dieselbe Wirkung wie ein Ausschalten des Selbsthaltesignals. Mit seiner Hilfe kann der gesamte Roboter deaktiviert werden.

[6] Bei einem 12V Akku beginnt die Tiefenentladung bei etwa 10,5V.

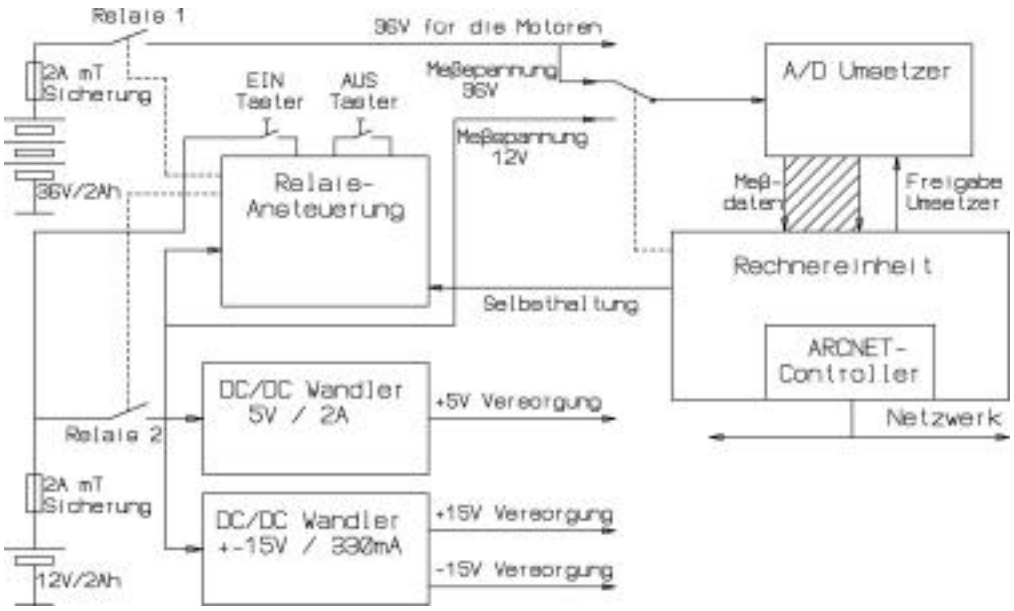


Abbildung 3.14.: Laderechner

Zum Anschluß aller Module stehen folgende Steckverbinder zur Verfügung:

- 17 Pins für 0V
- 12 Pins für +5V
- 6 Pins für +15V
- 6 Pins für -15V

Alle Spannungen für den Roboter werden auf dieser Platine erzeugt und dürfen auch nur hier abgenommen werden.

3.4. Betriebssystem

Bei der Beschreibung des Standard-Rechners in Kapitel 3.2.1 ist das Stichwort Betriebssystem bereits gefallen, wobei es sich hier eher um ein Miniatur-Betriebssystem handelt. Die Hauptmotivation war eine komfortablere Softwareentwicklung. Hierzu soll es folgende Aufgaben erfüllen:

1. Download der User-Programme, wodurch ein Wechsel des Programmspeichers (EPROM) unnötig wird.
2. Komfortable Kommunikationsroutinen für das User-Programm, damit dieses das Netzwerk als abstrakte Schnittstelle ansehen und auf einfache Weise nutzen kann.

Jeder Rechner erhält eine Kopie dieses Betriebssystems, wobei es an die jeweiligen Anforderungen des Moduls angepaßt wird. Diese Änderungen betreffen beispielsweise Interrupts und Zusatzfunktionalitäten, die in anderen Modulen unnötig sind.

Beim Download der User-Programme spielt das Modul „Hauptrechner“ eine zentrale Rolle. Über seine serielle Schnittstelle werden alle User-Programme vom PC in den Roboter geladen. Hierzu wird ein Programm nach dem anderen stückweise in den Hauptrechner verbracht und von dort aus über das Netzwerk an das Ziel-Modul geschickt. Zusätzlich zum eigentlichen User-Programm wird Datum und Zeitpunkt seiner Erstellung mitgeliefert und im Betriebssystem gespeichert. Dieser Zeitstempel entspricht so einer Art Versionsnummer des User-Programms, die später eine wichtige Rolle beim Laden der Programme spielt. Ein einfaches Protokoll sorgt für die Datensicherheit während des Transportes.

Die Kommunikationsroutinen werden im folgenden ausführlich behandelt, da diese in einem verteilten System eine wichtige, wenn nicht sogar die zentrale Rolle spielen.

3.4.1. Kommunikation

Grundsätzlich gibt es zwei verschiedene Arten der Kommunikation:

- Ereignissteuerung
- Zeitsteuerung

Bei der **Ereignissteuerung** werden nur dann Daten versendet, wenn ein Ereignis eingetreten ist. Dies hat den Vorteil, daß Daten nur dann das Netz belasten, wenn es nötig ist. Andererseits sind aber Kollisionen prinzipiell nicht zu vermeiden. Eine ausführlichere

Betrachtung der damit verbundenen Probleme und effektiven Netzwerkauslastung ist in [13] zu finden. An dieser Stelle sei ausdrücklich darauf hingewiesen, daß ARCNET keine Kollisionen auf physikalischer Ebene zuläßt[7]. Auf logischer Ebene jedoch sehr wohl. Ist ein Empfänger nicht bereit, eine Nachricht anzunehmen, zum Beispiel weil er kurz zuvor bereits eine andere erhalten hat, wird die neue Nachricht zurückgewiesen. Hier liegt nun durchaus eine Art Kollision vor, wenn auch auf anderer Ebene. Ein ereignisgesteuerter Treiber hebelt so den wohl größten Vorteil von ARCNET, nämlich die Kollisionsfreiheit, aus.

Ausgedehnte Versuche mit einem solchen Treiber haben sehr schnell die angesprochene Problematik verdeutlicht. Die Rechner waren die meiste Zeit damit beschäftigt zu kontrollieren, ob ihre Nachrichten nicht nur versandt, sondern auch beim Empfänger angekommen waren. Zudem hat sich gezeigt, daß das meiste Datenaufkommen aus *periodischen* Daten besteht, wie den Sonarmeßwerten. Hinzu kommt, daß Daten, die mehrere Rechner benötigen, jedes mal explizit an diese versandt werden müssen. Dies scheint zunächst nur etwas lästig, hat aber zur Folge, daß das Netzwerk durch die vielen Kollisionen und ständigen Sendeversuche chronisch überlastet ist, obwohl der eigentliche Datendurchsatz nicht sehr groß ist. Es mußte also eine Möglichkeit gefunden werden, dies zu verbessern.

Zeitsteuerung scheint hier nun die geeignetere Methode zu sein. Bei ihr werden sie Daten in (festen) Intervallen versendet. Mangels Echtzeituhren in den Modulen wurde auf die festen Intervalle verzichtet, stattdessen werden die Daten periodisch mit größtmöglicher Geschwindigkeit verschickt. Ein sehr interessantes Buch zu diesem Thema ist [12] von H. Kopetz. Die dort vorgebrachten Argumente waren entscheidend für die Wahl dieser Kommunikations-Form. Allerdings wurde nur die Methode der Zeitsteuerung übernommen. Die Art und Weise wie die Daten behandelt werden, stellt eine Eigenentwicklung dar, die direkt auf die Bedürfnisse dieses Projektes zugeschnitten wurde.

Doch nun zur verwendeten Systematik.
Wie bereits im Kapitel über das Netzwerk (Kapitel 3.2.2) beschrieben, ordnen sich die Module zu einem logischen Ring. *Ein* Datenpaket mit einer Größe von 512 Bytes wird in diesem Ring ständig von Rechner zu Rechner weitergereicht. Wie bereits erwähnt, geschieht das mit maximal möglicher Geschwindigkeit. Alle Daten, die versendet werden sollen, sei es von den Betriebssystemen oder den User-Programmen, müssen in dieses Datenpaket geschrieben werden.

An dieser Stelle muß darauf hingewiesen werden, daß dieses Datenpaket im folgenden als „token“ bezeichnet wird. Dies ist *nicht* das gleiche token das der ARCNET-Treiber für das Senderecht der einzelnen Module herumreicht (Kapitel 3.2.2). Da diese Funktionalität ARCNET-Intern ist und automatisch vom ARCNET-Controller erledigt wird, spielt sie hier keine Rolle mehr.

[7] Im Gegensatz zu ETHERNET u.ä..

Die gesamte Kommunikation findet in einer einzigen Interruptfunktion statt, die im Betriebssystem angesiedelt ist. Sie wird immer aktiv wenn das token empfangen wurde, bearbeitet die Daten, Befehle etc. und schickt danach das token *sofort* weiter an das nächste Modul[8]. Die Art und Weise wie das token von dieser Funktion genutzt wird, teilt sich hier in zwei Bereiche auf:

- Betriebssystem-interne Kommunikation
- User-Kommunikation

Beide sollen in den folgenden Kapiteln in dieser Reihenfolge näher beleuchtet werden.

3.4.2. Betriebssystem-Kommunikation

Bei Befehlen der Betriebssysteme oder kurz System-Befehlen wird das token als einfacher Daten-„Stream“ betrachtet. Diese Befehle können prinzipiell jederzeit auftreten, also auch bei aktiven User-Programmen, deshalb haben sie absolute Priorität und werden als erstes abgearbeitet. Die Kommunikation der aktiven User-Programme wird komplett unterbrochen bis kein Modul mehr System-Befehle absetzt.

Das token teilt sich nun wie folgt auf: Die ersten vier Bytes benötigt der ARCNET-Controller für interne Daten wie Sender-ID (SID), Empfänger-ID (DID) und Länge des Paketes und sind hier nicht weiter von Interesse. Die folgenden 508 Bytes stehen also zur Verfügung. Davon werden zunächst nur die nächsten 8 Bytes für System-Befehle genutzt. Danach folgt der Datenbereich für die User-Kommunikation. Da aber einige System-Befehle mehr Platz für Befehls-Daten benötigen, zum Beispiel beim Laden eines User-Programmes, werden diese ab dem zwölften Byte in das token geschrieben. Wie diese doppelte Nutzung ohne die Zerstörung von User-Daten erreicht werden kann, wird noch näher beschrieben.

Die folgende Tabelle zeigt die ersten Bytes des token und ihre Bedeutung bei System-Befehlen:

Byte-Nr	Name	Bedeutung
0	SID	ARCNET-intern: ID des Senders (SID)
1	DID	ARCNET-intern: ID des Empfängers (DID)
2	0	ARCNET-intern: langes Paket
3	4	ARCNET-intern: 512 Bytes übertragen
4	token-counter	laufende token-Nummer (0-255)
5	System-Command	Modul-ID wenn System-Befehl vorliegt, sonst 0
		Fortsetzung auf der nächsten Seite

[8]Das Prinzip der heißen Kartoffel, die keiner lange halten möchte.

Fortsetzung von vorheriger Seite		
Byte-Nr	Name	Bedeutung
6	Destination-ID	für welches Modul ist der Befehl (ID) ?
7	Command-value	welcher Befehl ?
8	Command-length	Datenlänge des Befehls
9	Command-acknowledge	Befehl angekommen (ID wenn ja), sonst 0
10	reserviert	
11	reserviert	
12	Data	Daten für einen Befehl z.B. ein
.	.	Teil eines User-Programms etc.
62	Data	

Der token-counter ist einfach eine laufende Nummer, die bei jedem Umlauf des token vom Hauptrechner inkrementiert wird. Hiermit können die Module erkennen, ob sie ein token verpaßt haben und eventuell darauf reagieren. In das System-Command-Byte trägt ein Modul seine ID ein, falls es einen System-Befehl absetzen möchte. Dann folgt der Befehls-Code sowie seine Länge. Im Command-acknowledge-Byte bestätigt das Empfangsmodul den Erhalt des Befehls indem er einfach seine ID einträgt. Die beiden nächsten Bytes sind für eventuelle Erweiterungen reserviert. Gehören Daten zu einem Befehl, was zum Beispiel beim Laden von ganzen Programmen unerläßlich ist, werden diese ab Byte-Nr 12 geschrieben. Hierzu sichert das sendende Modul zunächst die nächsten 32 Bytes und schreibt dann die Daten hinein. Ist der Befehl angekommen und keine weitere Übertragung gewünscht, restauriert der Sender das token wieder, indem er die vorher gesicherten 32 Bytes wieder in das token zurückschreibt. So wird Datenverlust im User-Datenbereich vorgebeugt.

Den Betriebssystemen stehen eine Reihe von Befehlen zur Verfügung:

Befehl	Wirkung
`ARCNET_SCAN`	Fordert alle angeschlossenen Rechner auf, ihre ID als Liste in das token zu schreiben.
`ARCNET_GET_CONFIG`	Fragt den aktuellen Zeitstempel eines Moduls ab. Dieser trägt ihn in das token ein.
`ARCNET_SEND_CONFIG`	Sendet den neuen Zeitstempel an den Zielrechner.
`ARCNET_DOWNLOAD_DATA`	Sendet ein Datenpaket an einen Zielrechner. Dies ist Teil eines User-Programms.
`ARCNET_DOWNLOAD_TERMINATED`	Teilt dem Zielrechner mit, daß sein Download erfolgreich beendet wurde.
`ARCNET_START_SYSTEM`	Fordert alle Rechner auf ihre User-Programme zu starten.
	Fortsetzung auf der nächsten Seite

Fortsetzung von vorheriger Seite	
Befehl	Wirkung
ARCNET_ERROR	Allgemeine Fehlermeldung.

Weitere Befehle wie ARCNET_STOP_SYSTEM oder das Starten einzelner User-Programme sind zwar schon vorgesehen aber noch nicht vollständig implementiert.

Bei Einschalten des Roboters laufen die Betriebssysteme hoch, die zunächst das Netzwerk initialisieren und sich anmelden. Danach startet ein Rechner, der Hauptrechner, die Kommunikation, indem er das token mit Nullen initialisiert und losschickt. Ab jetzt kreist dieses und nur dieses eine Paket im Netzwerk. Nun können mit den erwähnten ARCNET-Befehlen die Zeitstempel (Versionen) der User-Programme ermittelt werden und eventuell neue Programme geladen werden (Kapitel 3.5 beschreibt den Ablauf des Downloads etwas detaillierter). Ist dies geschehen, werden mittels des Befehls ARCNET_START_SYSTEM alle User-Programme gestartet. Die Betriebssysteme sind nun bis auf die zentrale Interruptfunktion inaktiv. Diese schaltet nun in einen anderen „Modus“ um, denn jetzt wollen die aktiven User-Programme miteinander kommunizieren, was die Behandlung des token verändert.

3.4.3. User-Kommunikation

Der Datenbereich ab Byte Nr 12 steht, wie bereits geschildert, den User-Daten zur Verfügung, was genau 500 Bytes entspricht. Jeder Rechner bekommt nun zwei Bereiche (Kanäle) zur Verfügung. Als ersten Ansatz werden hier maximal 10 Rechner[9] und eine lineare Teilung des freien Bereiches angenommen. Dies hat zur Folge, daß jeder Rechner 50 Bytes beziehungsweise 25 Bytes je Kanal erhält. Ein Kanal ist für Befehlsempfang und einer für Datenausgabe gedacht und werden noch ausführlich behandelt. Das token ist so zu einer Art Postboten geworden, der pro Rechner einen Brief abliefern (den Befehl) und einen Brief, der für ihn bereit gelegt wurde, mitnehmen kann (Datenkanal) und als Postwurfsendung an alle anderen Module verteilt.

Die Datenübertragung zwischen Betriebssystem und User-Programm geschieht über Datenstrukturen. Das User-Programm legt diese an und das Betriebssystem kopiert die Daten in den jeweiligen Kanal beziehungsweise aus dem Kanal in die Struktur. Abbildung 3.15 zeigt das token, daß gerade im Sonar-Modul angekommen ist. Das Betriebssystem wurde durch den Empfang des token aktiv und kopiert nun die Datenstrukturen des User-Programms in die Kanäle und aus ihnen heraus. Mit zwei Semaphoroperationen

```
lock_data(Kanal);
unlock_data(Kanal);
```

[9]Die Betriebssysteme unterstützen zur Zeit nur diese Anzahl.

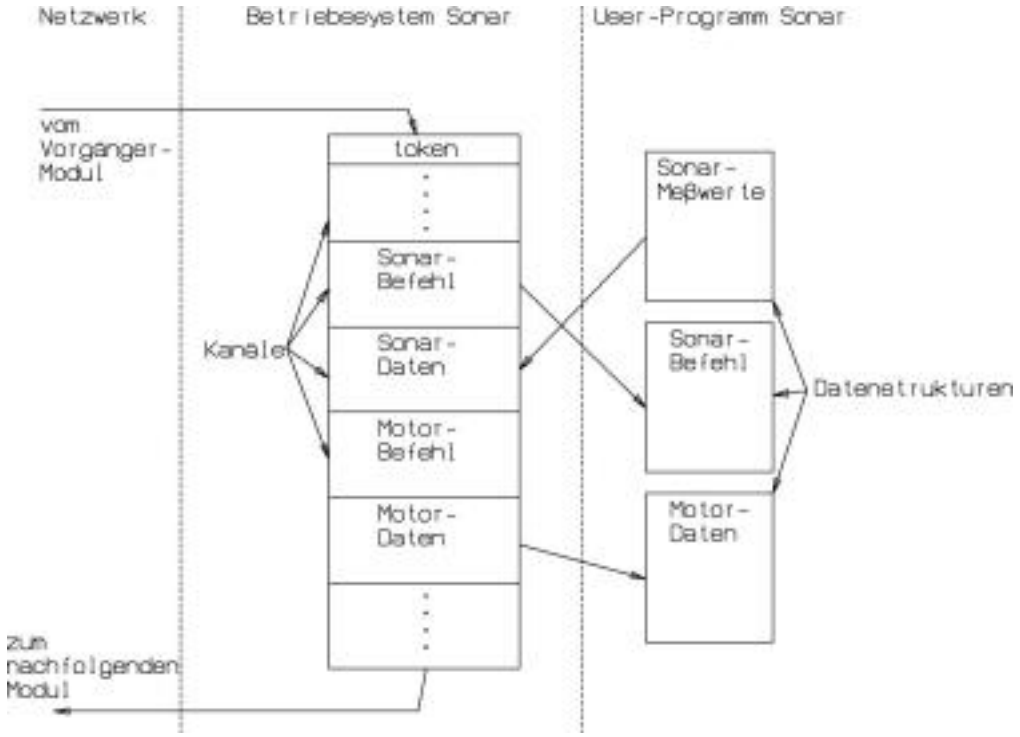


Abbildung 3.15.: Kommunikation mittels token

kann jeder Kanal explizit gesperrt und wieder freigegeben werden. Dies ist notwendig, wenn die Daten einer Struktur einen inneren Zusammenhalt haben und dieser Datensatz gerade vom User gelesen oder beschrieben wird. Hier bestünde sonst die Gefahr, bei Unterbrechung durch das Betriebssystem, nur Datenfragmente zu bearbeiten. Damit diese Kommunikation funktionieren kann, müssen ein paar Voraussetzungen gegeben sein.

Grundvoraussetzungen

Die Betriebssysteme haben zunächst keine Ahnung von der genauen Aufteilung des token, also der Anzahl, Größe und Zuteilung der einzelnen Kanäle. Diese Informationen stehen erst einmal nur den User-Programmen zur Verfügung. Bei ihrer Entwicklung auf dem PC wird eine zentrale Datei namens `PROTOKOL.H` mit eingebunden.

In dieser Datei sind die Kanaladressen, Größen und Namen eingetragen. Die einzelnen Kanäle werden anhand einer eindeutigen Nummer (ID) unterschieden. Ihre symbolischen Namen bestehen immer aus dem Namen des Rechners plus „Kanal-Art“, also „ROBSONAR_DATA“ für den Datenkanal des Sonar-Moduls und „ROBSONAR_COMMAND“

für seinen Befehlskanal usw. Die Adressen bestehen zusätzlich aus dem Anhang „_ADDRESS“, also zum Beispiel „ROBSONAR_DATA_ADDRESS“. Da alle User-Programme die gleiche Datei referenzieren, ist auch die Aufteilung des token für alle gleich.

Dieses Vorgehen hat den entscheidenden Vorteil, daß Kanalparameter auf einfache Weise, nämlich durch Modifikation dieser einen Datei, geändert werden können. Eine Änderung der Betriebssysteme ist nicht erforderlich. So kann sich im Laufe des Entwicklungsprozesses beispielsweise herausstellen, daß die 25 Bytes für die Sonar-Daten nicht ausreichen. Eine Änderung könnte diesem Modul also mehr Platz einräumen[10]. Nach einer Änderung in der Kanalaufteilung müssen allerdings alle User-Programme neu compiliert werden, damit auch alle Programme mit der neuen Aufteilung arbeiten.

Diese Aufteilung muß den Betriebssystemen natürlich mitgeteilt werden, denn sie sollen schließlich später mit den Kanälen arbeiten. Im Folgenden wird also davon ausgegangen, daß User-Programme geschrieben, compiliert, in die Module geladen und mit dem System-Befehl `ARCNET_START_SYSTEM` gestartet wurden. Als erstes wird die nun nötige Initialisierung beschrieben, die jedes User-Programm durchlaufen muß. Dann folgt eine ausführliche Beschreibung des Verhaltens und der Benutzung des Befehls-Kanals, gefolgt von der Beschreibung des Daten-Kanals.

Die Initialisierung

Um einem Betriebssystem die einzelnen Kanalparameter mitzuteilen, stellt es den Befehl

```
set_token_area(Kanal-ID, Kanal-Adresse);
```

zur Verfügung. Er muß für *jeden* Kanal aufgerufen und legt fest welcher Kanal an welcher Adresse (Byte-Nr) zu finden ist. Zusätzlich benötigt das Betriebssystem die Information welcher Kanal der *eigene* Befehlskanal ist, wozu

```
set_own_input_id(Kanal-ID);
```

dient.

Eine vollständige Initialisierung sieht nun am Beispiel des Sonars folgendermaßen aus:

```
/* eigenen Befehlskanal festlegen */
set_own_input_id(ROBSONAR_COMMAND);

/* alle Kanaladressen festlegen */
```

[10] Ein anderer Kanal muß dafür natürlich „abspecken“.

```
set_token_area(ROBMAIN_COMMAND, ROBMAIN_COMMAND_ADDRESS);
set_token_area(ROBMAIN_DATA,    ROBMAIN_DATA_ADDRESS);

set_token_area(ROBSONAR_COMMAND, ROBSONAR_COMMAND_ADDRESS);
set_token_area(ROBSONAR_DATA,    ROBSONAR_DATA_ADDRESS);

set_token_area(ROBMOTOR_COMMAND, ROBMOTOR_COMMAND_ADDRESS);
set_token_area(ROBMOTOR_DATA,    ROBMOTOR_DATA_ADDRESS);

set_token_area(ROBCOMP_COMMAND, ROBCOMP_COMMAND_ADDRESS);
set_token_area(ROBCOMP_DATA,    ROBCOMP_DATA_ADDRESS);

set_token_area(ROBCOPRO_COMMAND, ROBCOPRO_COMMAND_ADDRESS);
set_token_area(ROBCOPRO_DATA,    ROBCOPRO_DATA_ADDRESS);

set_token_area(ROBMAP_COMMAND, ROBMAP_COMMAND_ADDRESS);
set_token_area(ROBMAP_DATA,    ROBMAP_DATA_ADDRESS);

set_token_area(ROBLCD_COMMAND, ROBLCD_COMMAND_ADDRESS);
set_token_area(ROBLCD_DATA,    ROBLCD_DATA_ADDRESS);

set_token_area(ROBLOAD_COMMAND, ROBLOAD_COMMAND_ADDRESS);
set_token_area(ROBLOAD_DATA,    ROBLOAD_DATA_ADDRESS);
```

Die Betriebssysteme wissen nun, welcher Kanal an welcher Adresse zu finden ist und einer Benutzung steht nichts mehr im Wege. Da aber der Befehls- und der Daten-Kanal auf vollständig andere Art und Weise arbeiten, muß ihre Benutzung und Verhaltensweise näher beschrieben werden.

Der Befehlskanal

Über diesen Kanal empfängt ein Rechner in erster Linie Befehle anderer, die er ausführen soll. Er ist nur für sporadisch auftretende Daten gedacht und sollte auch nur so Verwendung finden, da dieser Kanal in seinem Verhalten im Grunde einer Ereignissteuerung entspricht. So könnte zum Beispiel der Hauptrechner der Motorsteuerung den Befehl schicken, nach vorne zu fahren, zu stoppen und ähnliches. Im Postvergleich ähnelt dies einer Art Einschreiben.

Hierzu legt die Motorsteuerung eine Struktur an und teilt seinem Betriebssystem mit, Befehle, die in *seinem* Befehlskanal liegen, in diese Struktur zu schreiben. Er schraubt sozusagen seinen Briefkasten an die Wand.

```
get_data(Kanal-ID, Laenge des Befehls, Adresse der Struktur);
```

Danach muß der Kanal explizit freigegeben werden, da alle Kanäle standardmäßig gesperrt sind. Er öffnet also die Klappe des Briefkastens, da ein Einwurf nur bei geöffneter Klappe funktioniert. Dies erfolgt mit der Funktion

```
unlock_data(Kanal-ID);
```

Mit dem folgenden Funktionsaufruf kann die Motorsteuerung nun überprüfen, ob ein Befehl für ihn gekommen ist, also im Briefkasten nach Post sehen:

```
command_received(Kanal);
```

Wurde ein Befehl empfangen, so sperrt sein Betriebssystem den Kanal automatisch, um ein Überschreiben zu verhindern (die Klappe schließt sich automatisch). Der Kanal muß also zum Empfang eines neuen Befehls mit `unlock_data()` wieder freigegeben werden.

Der Hauptrechner legt seinerseits eine Struktur an, in die er die Befehlsdaten einträgt. Er weist nun sein Betriebssystem an mit der Funktion

```
send_command(Kanal-ID, Laenge des Befehls, Adresse der Struktur);
```

diesen Befehl in den Befehlskanal der Motorsteuerung zu schreiben. Mit den Funktionen

```
command_transmitted(Kanal-ID);
command_acknowledged(Kanal-ID);
```

kann der Hauptrechner überprüfen ob sein Befehl verschickt und auch bestätigt wurde. Dies ist ein wichtiger Unterschied, da diese Bestätigung von den Betriebssystemen übernommen wird, nicht von den User-Programmen (es steht im token, also übernimmt der Postbote diese Aufgabe).
Die Funktion `command_transmitted()` sagt aus, daß der Befehl erfolgreich in das token (den Kanal) geschrieben werden konnte, transportiert wurde und wieder beim Hauptrechner ankam. In diesem Sinne hat der Postbote die Post zur Motorsteuerung mitgenommen.
Die Funktion `command_acknowledged()` dagegen besagt, daß der Befehl zusätzlich vom Betriebssystem der Motorsteuerung in die entsprechende Struktur geschrieben werden konnte. Der Befehl ist also auch auf User-Ebene angekommen, also wirklich im Briefkasten gelandet. Ist diese Prüfung dagegen negativ, bedeutet es, daß der Postbote den Brief zwar mitgenommen hat, aber eine verschlossene Klappe vorfand und folglich den Brief nicht einwerfen konnte.

Zeitlich sieht der Ablauf folgendermaßen aus. Kommt das token beim Sender des Befehls an und ist der gewünschte Befehlskanal frei, so schreibt er den Befehl in den Kanal und schickt das token weiter. Intern wird das erfolgreiche Absetzen des Befehls gespeichert. Die Funktion `command_transmitted()` würde ab jetzt einen Erfolg melden. Der Kanal ist nun belegt, das heißt kein anderer Rechner kann ihn seinerseits belegen und damit überschreiben. Kommt das token nun beim Empfänger an, so versucht sein Betriebssystem den Befehl aus dem token in die User-Struktur zu kopieren. Gelingt ihm dies, so setzt er im Kanal ein acknowledge-Flag und schickt das token weiter. Ansonsten wird das token einfach weitergereicht. Das token kommt nun irgendwann wieder beim Sender an, der nun das acknowledge-Flag überprüft. Ist es gesetzt wird dies für die Funktion `command_acknowledged()` gespeichert. Ist es nicht gesetzt, so fällt auch die Prüfung negativ aus. In jedem Fall wird der Inhalt des Kanals nun vom Sender wieder gelöscht. Er gibt ihn also für die anderen Rechner frei und reicht das token weiter. Auf diese Weise können die Rechner ihre Befehlskanäle nicht gegenseitig blockieren. Im schlimmsten Fall muß ein Rechner $n_{Module} - 2$ token-Umläufe abwarten, bis er den Kanal nutzen kann. Ganz gleich ob das acknowledge-Flag gesetzt ist oder nicht, wird nun kein weiterer Sendeversuch unternommen. Falls dies gewünscht wird, muß das User-Programm diesen neu initiieren (kein echter Postbote hat auf mehrere Versuche Lust).

Wie bereits erwähnt, entspricht dieser Kanal voll und ganz einem ereignisgesteuerten Verhalten, mit all den Vor- und Nachteilen. Gegenseitige Blockaden sind also möglich, da das Freischalten des Kanals Sache des User-Programms ist von den Betriebssystemen nicht beeinflußt werden kann.

Der Datenkanal

Dieser Kanal ist ausschließlich für periodische Daten gedacht. Es sind also genau die Art von Daten, die bei einem ereignisgesteuerten Treiber so viel Ärger machten, da sie am häufigsten auftraten und ständig kollidierten. Der Kanal wird zur Ausgabe von Daten genutzt, die der Rechner dem gesamten Netzwerk zur Verfügung stellen möchte, was einer Art „Broadcast“ entspricht.

Zum Beispiel kann das Sonar in seinem Datenkanal immer seine aktuellen Meßwerte plazieren. Die Motorsteuerung schreibt in seinen Datenkanal die Geschwindigkeit und Richtung, in die zur Zeit gefahren wird und so weiter. Alle Rechner, die sich für diese Daten interessieren, können[11] sich diese aus dem token heraus kopieren, wenn es bei ihnen ist. Hierzu erzeugen sie eine Ziel-Struktur und weisen mit dem Befehl

```
get_data(Kanal-ID, Laenge des Befehls, Adresse der Struktur);
```

[11] Sie *müssen nicht*, ein sehr wichtiger Unterschied zum echten Broadcast.

ihr Betriebssystem an, die Daten dort abzulegen. Da alle Kanäle standardmäßig gesperrt sind, muß der Kanal noch mit

```
unlock_data(Kanal-ID);
```

freigegeben werden. Die Empfänger können sicher sein, immer „frische“ Daten in der Struktur vorzufinden.

Senderseitig gestaltet sich die Nutzung ähnlich einfach. Nach der Erzeugung einer Quell-Struktur sorgt der Befehl

```
store_data(Kanal-ID, Laenge der Daten, Adresse der Struktur);
```

dafür, daß das Betriebssystem beim Empfang des token die Struktur in den Kanal umkopiert. Doch zunächst muß dieser auch mit

```
unlock_data(Kanal-ID);
```

freigegeben werden. Der Sender muß sich jetzt in keiner Weise mehr um den Transport kümmern. All dies übernimmt sein Betriebssystem. Er muß nur noch dafür sorgen, daß immer frische Daten in seiner Struktur stehen und ihre Integrität mit den Semaphoreoperationen sicherstellen. Da jeder alleiniger „Herrscher“ über seinen Datenkanal ist, sind Kollisionen hier unmöglich. Im Vergleich mit dem Postsystem entspricht der Datenkanal so einer Art Postwurfsendung, bei der die Empfänger allerdings steuern können ob sie diese erhalten wollen oder nicht.

Einmal angelegte Kanäle lassen sich mit dem Befehl

```
cancel_data(Kanal-ID);
```

auch wieder löschen.

Diese Art der Kommunikation hat nun einige Nebenbedingungen und Charakteristika, die kurz skizziert werden sollen, wobei diese Liste keinen Anspruch auf Vollständigkeit erhebt:

1. Die Datenkanäle werden zu einer 1:n Beziehung, während Befehlskanäle eine 1:1 Beziehung darstellen.
2. Die Daten für die Datenkanäle sollten von den entsprechenden User-Programmen möglichst schnell auf dem neuesten Stand gehalten werden, da andere Rechner sonst mit veralteten Daten arbeiten.

3. Daten, für die sich mehrere Rechner interessieren, müssen nicht mehr explizit zu jedem versandt werden. Jeder Interessent „bedient sich einfach“ im jeweiligen Datenkanal.

4. Eine Softwareänderung im sendenden Rechner ist obsolet, falls ein neuer Rechner hinzu kommt und sich ebenfalls für diese Daten interessiert.

5. Das Datenpaket stellt eine Art „Gedächtnis“ des gesamten Systems dar.

6. Befehle entsprechen zwar dem Verhalten der Ereignissteuerung, doch da sie zeitgesteuert von Rechner zu Rechner versandt werden, muß eine entsprechende Latenzzeit bis zum Eintreffen einkalkuliert werden.

7. Um die token-Umlaufzeit klein zu halten sollte jeder Rechner nur die Kanäle freischalten, dessen Daten er wirklich benötigt beziehungsweise loswerden möchte. Die Betriebssysteme sind sonst mit sinnlosen Kopiervorgängen beschäftigt und das kostet Zeit, um die das Weiterschicken des token natürlich verzögert wird.

8. Werden einmal freigeschaltete Kanäle überhaupt nicht mehr benötigt, sollten sie mit `cancel_data()` wieder gelöscht werden.

9. Sind die Daten nur temporär nicht nötig, reicht ein Sperren mit `unlock_data()`. Dies spart Zeit im Betriebssystem.

10. Diese token-Umlaufzeit liegt zur Zeit (7 Rechner sind im Netzwerk) bei 30-40 ms, was sich als ausreichend schnell erwiesen hat.

Dies ist, alles in allem, nun eine wesentliche Verbesserung gegenüber dem alten ereignisgesteuerten Treiber. Die *Mehrzahl* der Daten können kollisionsfrei und auf äußerst einfache Weise verschickt und auch empfangen werden, was sich in der Praxis mehr als nur bewährt hat.

3.5. Download-Programm

Um die User-Programme in den Roboter laden zu können, wurde ein Programm für den PC entwickelt, das genau dies leistet. Auf eine vollständige Beschreibung wird hier verzichtet, statt dessen sollen die Arbeitsweise und zur Zeit implementierten Funktionalitäten nur kurz beschrieben werden.

Das Programm heißt `LOAD.EXE` und ist unter DOS beziehungsweise einer entsprechenden Emulation lauffähig. Es wird über Parameter gesteuert, die bei Aufruf angegeben werden müssen. Fehlen diese oder sind falsch, wird automatisch ein kleiner Hilfe-Text ausgegeben. Zudem müssen sich die User-Programme im gleichen Verzeichnis befinden wie das Programm. Hierbei handelt es sich um die HEX-Dateien die der Compiler, respektive Linker generiert und normalerweise zum Programmieren von EPROMs genutzt werden. Diese Dateien müssen im Intel-HEX Format vorliegen.

Die grundlegende Funktionsweise ist nun folgende. Zunächst sucht das Programm auf der seriellen Schnittstelle nach dem Roboter (zur Zeit der Hauptrechner). Meldet sich dieser, wird er veranlaßt, den ganzen Roboter nach angeschlossenen Modulen zu durchsuchen und die einzelnen Konfigurationen zu ermitteln, also die Namen, IDs und Zeitstempel der User-Programme. Die Daten werden nun an das PC-Programm übergeben, das dann nach den neuen User-Programmen sucht. Werden diese gefunden, so wird Programm für Programm paketweise an den Hauptrechner übertragen, der sie über das Netzwerk an die Zielrechner weiterleitet.

Mehrere Funktionen sind zur Zeit implementiert:

- `load -i` steht für inkrementelles Herunterladen. Hierbei werden nur die User-Programme geladen die neuer sind als die aktuell in den Modulen vorhandenen. Sinn macht dies vor dem Hintergrund, daß sich nicht immer alle Programme ändern und so nur die neuen geladen werden müssen, was sehr viel Zeit spart.

- `load -a` steht für „all“ und veranlaßt alle Programme, unabhängig ihres Alters, zu laden. Dies sollte immer dann genutzt werden, wenn eine Zerstörung einzelner User-Programme aufgrund eines Software-Fehlers vermutet wird oder der Entwickler einfach zuviel Zeit hat.

- `load -s <filename>` lädt ein „spezifisches“ Programm in den Roboter.

- `load -start` startet die User-Programme.

- `load -u <filename>` ist eigentlich schon ein Vorgriff auf die Kartographie, darf aber hier der Vollständigkeit halber nicht fehlen und funktioniert nur bei aktiven User-Programmen. Der Roboter schickt seine aufgezeichneten Karten-Daten an

den PC. Dieser speichert die Karte in einem CAD[12]-Format ab. Die Karte kann nun mit einem CAD-Programm betrachtet werden, um zum Beispiel genaue Längen zu ermitteln um sie mit den wahren zu vergleichen, eine Referenz-Karte zu überlagern und ähnliches. Zusätzlich wurde ein weiteres DOS-Programm entwickelt, das eine erste Betrachtung der Daten erlaubt. `GRAPH.EXE` erwartet die Karte als Parameter und diese kann nun vergrößert, verkleinert und bewegt werden. Ein Maßstab, der 1m Länge entspricht, erlaubt eine grobe Größenabschätzung der Karten-Daten.

Ein typischer Entwicklungsprozeß sieht nun folgendermaßen aus:

1. Erstellung eines User-Programms beziehungsweise eine Änderung desselben.
2. Compilieren des User-Programms.
3. Roboter an die serielle Schnittstelle anschließen und einschalten.
4. Laden des neuen Programms mit `load -i` und warten bis dies beendet wurde.
5. Starten des Roboters (der User-Programme) mit `load -start` und Roboter von der seriellen Schnittstelle abtrennen.
6. Testen des Programms also zum Beispiel den Roboter fahren lassen.
7. Nach der Fahrt den Roboter erneut an den PC anschließen und seine aufgezeichneten Daten mit `load -u test.dat` laden und eventuell mit einem CAD-Programm überprüfen.

In der Praxis zeigt sich, daß gerade das inkrementelle Laden von Programmen den Entwicklungsprozeß ganz erheblich beschleunigt. Im Schnitt werden nur zwei bis drei Programme zur Zeit geändert und nur diese ersetzt, was typischerweise eine Ladezeit von etwa 20-40 Sekunden benötigt. Ein vollständiges Laden aller Programme liegt dagegen bei etwa 3-4 Minuten, also eine ganz wesentliche Zeitersparnis.

[12]Computer-Aided-Drawing

4. Kartographie

Die Kartographie soll dem Roboter ein mehr oder minder abstraktes Modell der Umgebung liefern. Diese Abbildung soll später die Grundlage navigatorischer und taktischer Entscheidungen für die Bewegungen des Roboters liefern, zum Beispiel, um bekannten Objekten auszuweichen, gezieltes Anfahren von Positionen und ähnliches zu ermöglichen. Da die dafür notwendige Interpretation dieser Karte zur Zeit noch nicht vorgesehen ist, reicht es aus, die Daten zunächst nur zu sammeln.

Basis dieser Abbildung der Umwelt bilden die Sensordaten des Sonars und die aktuelle Position des Roboters, die er selbst kennen beziehungsweise schätzen muß. Da die Sensorik nicht in der Lage ist mit einem „Blick“ ihre Umgebung, oder zumindest einen größeren Teil von ihr, zu erfassen ist es unabdingbar, die Karte inkrementell zu erstellen. Hierzu muß sich das System durch seine Umwelt bewegen, seine aktuelle Position kennen beziehungsweise schätzen und gleichzeitig die Sensordaten aufzeichnen, klassifizieren, in eine Kartendarstellung umrechnen und letztendlich auch archivieren. All dies soll in Echtzeit geschehen.

4.1. Testumgebung

Einen Roboter zu konstruieren, der mit der gesamten Komplexität unserer Welt zurecht kommt, ist extrem schwierig. Deshalb muß an dieser Stelle auf die Einschränkungen und Zielszenarien, in denen das System eingesetzt werden soll, näher eingegangen werden.

Zunächst wichtige Einschränkungen, ohne die das System zur Zeit keine Chance hätte, zurecht zu kommen[1].

- Einsatz des Roboters nur in Innenräumen. Hier wird in erster Linie an Büroumgebungen oder ähnliches gedacht.
- Der Boden muß eben und glatt sein (PVC, Fliesen oder ähnliches). Dies ist nötig,

[1] Ein großer Teil dieser Einschränkungen sollen in naher Zukunft wegfallen, da sich passende Sensorik bereits in der Entwicklung befindet. Mehr dazu in Kapitel 5.

weil die Schrittmotoren sich als viel zu schwach erwiesen haben.

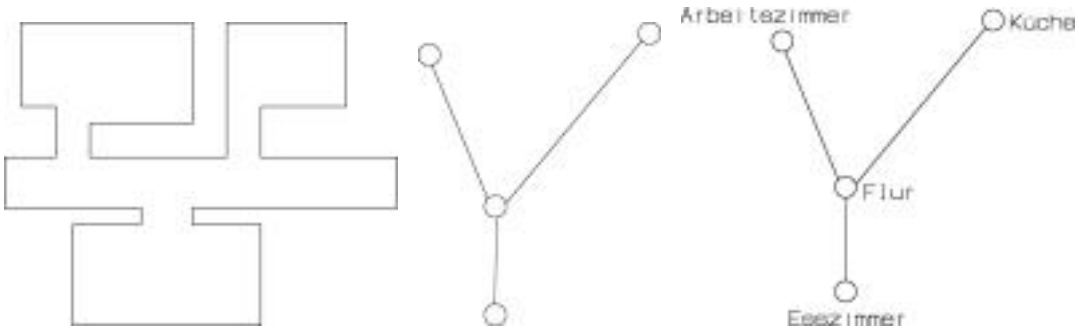
- Der Boden sollte zudem homogen beschaffen sein, also keine Untergrundwechsel.
- Objekte innerhalb des Raumes müssen entweder bis zum Boden reichen, oder so hoch sein, daß der Roboter sie unterfahren kann.
- Bewegte Objekte sind zur Zeit nicht erlaubt.
- Hinabführende Treppen sind nicht zulässig, da sie noch nicht detektiert werden können.

An Material und Formen der Objekte werden keine speziellen Anforderungen gestellt. Das Sonar ist, zumindest im Nahbereich, in der Lage die meisten Materialien zu detektieren. Einbußen hinsichtlich der Meßgenauigkeit müssen aber in Kauf genommen werden, was sich natürlich in der Genauigkeit der Karte niederschlägt. Der aber wohl wichtigste Aspekt ist, daß die Umwelt nicht künstlich an die Bedürfnisse des Roboters angepaßt werden soll, also keine Leiterschleifen, Baken oder ähnliches. Hier werden eher schlechte oder im schlimmsten Fall gar keine Ergebnisse in Kauf genommen.

4.2. Weltmodellierung

Bei der Repräsentation der Umwelt durch eine Karte gibt es verschiedene Möglichkeiten. Generell kommen drei verschiedene Modelle in Betracht, die sich hinsichtlich ihres Abstraktionsgrades unterscheiden (nach [3]):

- Das **geometrische Modell** läßt sich direkt aus den Meßwerten der Sensoren ableiten. Im einfachsten Fall werden Meßwerte der Sensorik, genauer gesagt detektierte Objekte, mittels der aktuellen Roboterposition in globale Koordinaten umgerechnet und als „Punkt“ in die Karte eingetragen. Das Ergebnis ist eine Rasterkarte mit freien und belegten Positionen.
- **Topologische Modelle** fassen Strukturen zusammen und zeigen Beziehungen zwischen ihnen auf. Beispielsweise einzelne Räume und die Verbindungen, also Durchgänge zu anderen Räumen. Hier werden Räume als Knoten dargestellt. Sind zwei Räume direkt untereinander erreichbar, existiert also ein Durchgang, werden die Knoten miteinander verbunden. Diese Art der Karte eignet sich besonders zur Navigation und kann aus dem geometrischen Modell heraus generiert werden.
- Das **semantische Modell** stellt die höchste Abstraktionsebene dar und beinhaltet weitere Umweltinformationen. Hier werden Strukturen und Objekte hinsichtlich ihrer Funktion klassifiziert. Also Büroräume, Korridore, Schränke und Wände unterschieden.



4.1.1: Geometrisches Modell

4.1.2: Topologisches Modell

4.1.3: Semantisches Modell

Abbildung 4.1.: Weltmodelle eines Szenarios

Das geometrische Modell stellt die einfachste und direkteste Form der Umweltmodellierung dar, eignet sich aber nur bedingt zur Weiterverarbeitung, also Interpretation. Da dies zur Zeit aber keine Rolle spielt, scheint sie die geeigneteste Methode zu sein, wobei sich bei der Repräsentation der Daten zwei Möglichkeiten anbieten:

Occupancy-Grid

Diese Methode wurde bereits bei der Vorstellung des geometrischen Modells kurz skizziert. Es handelt sich um ein $n * m$ großes Raster, daß die zu erkundende Umgebung darstellt. Detektiert die Sensorik ein Objekt, wird unter Berücksichtigung der aktuellen Roboterposition, eine globale Objektposition errechnet (X,Y Koordinate) und diese Position im Raster als belegt markiert. Die Einfachheit der Algorithmik hat aber auch einen gravierenden Nachteil. Diese Darstellung ist sehr speicherintensiv. So würde eine Karte mit einer Kantenlänge von 5m*5m und einer Rasterauflösung von 1cm bereits einen Speicherbedarf von 31.25KByte erfordern, was den Speicher des Kartenrechners voll und ganz beanspruchen würde, obwohl der darstellbare Bereich lächerlich klein ist.

Vektoren

Sie bieten eine sehr viel kompaktere Darstellungsform, da ein Vektor nur aus Start- und End-Position in X und Y Richtung besteht. Bei der gleichen Auflösung der Karte von 1cm und einer 16-Bit Integerzahl als Koordinaten wäre so ein Bereich von 655m * 655m darstellbar, wobei je Vektor nur 8 Bytes Speicher erforderlich sind, also bei 32KByte maximalem Speicher könnten theoretisch 4096 Vektoren Platz finden. Diese Parameter scheinen sehr viel besser für das System geeignet zu sein. Allerdings wird dieser Vorteil mit einem erhöhten Rechenaufwand erkauft, da sich die Algorithmik hier aufwendiger gestaltet. Diese Darstellungsart des Karte wird nun im gesamten System übernommen,

also auch zur Navigation.

Der Roboter wird sich in einem absoluten kartesischen Koordinatensystem bewegen, wobei X und Y Position jeweils durch eine vorzeichenbehaftete 16-Bit Integerzahl repräsentiert wird. Der Winkel mit seinem Wertebereich von 0° bis 359° wird, wie in der Navigation üblich, rechtsherum ansteigend berechnet, wobei 0° mit der positiven X-Achse zusammenfällt.

4.3. Algorithmen

Um der Idee des verteiltes System gerecht zu werden und da ein einzelner Prozessor eine sehr geringe Rechenleistung aufweist, ist es notwendig, die Teilaufgaben möglichst gleichmäßig und sinnfällig auf die einzelnen Module zu verteilen. Nur so ist es möglich, ein Gesamtverhalten zu realisieren, daß seine Aufgabe in Echtzeit löst. Im folgenden wird deshalb immer die Suche nach *schnellen* Verfahren im Vordergrund stehen, um die erwünschte Echtzeitfähigkeit nicht zu gefährden. Um ein funktionsfähiges Gesamtsystem zu erhalten sind also folgende Teilaufgaben zu lösen und auf die einzelnen Module zu verteilen:

1. Entwicklung eines Algorithmus zur Ansteuerung und Auswertung der Sonarmeßwerte. Dies ist einzig und allein Aufgabe des Sonar-Moduls.

2. Motoransteuerung und Wegverfolgung. Dies ist Sache der Motorsteuerung. Da sie während des Verfahrens über jede Bewegung informiert ist, ist sie prädestiniert zur Berechnung der aktuellen globalen Position des Roboters im Koordinatensystem.

3. Kompaßalgorithmus zur Ermittlung des aktuellen Winkels bezüglich des Erdmagnetfeldes. Ähnlich wie beim Sonar ist das Kompaßmodul als einziges in der Lage dies wahrzunehmen.

4. Entwicklung einer Steuerung, mit der das System seine Umwelt erkundet, also die Explorationsstrategie beinhaltet. Diese Aufgabe wird dem Hauptrechner zugetan, da er bis auf die Kommunikation mit dem PC noch keine weiteren Arbeiten wahrnimmt.

5. Positionsschätzung und Selbstlokalisation. Dieser Punkt teilt sich ein wenig auf. Während die aktuelle und fortlaufende Positionsschätzung Aufgabe der Motorsteuerung ist, wird die Selbstlokalisation dem Hauptrechner zufallen.

6. Entwicklung einer Methode, die Sensorwerte zu klassifizieren, in Kartendaten umzuwandeln und dies unter Echtzeitaspekten. Dies ist zunächst Aufgabe des Kartenrechners, erfordert aber relativ viel Rechenleistung, wodurch der Kartenrechner überlastet werden könnte. Deshalb scheint auch hier eine Aufteilung notwendig.

Die Klassifizierung und Generierung von Vektoren wird dem bislang untätigen Coprozessor zugedacht. Er wird ständig die Sonarmeßwerte überwachen, gegebenenfalls anhand der aktuellen Position der Motorsteuerung Vektoren erzeugen und sie zur Archivierung an den Kartenrechner übergeben. Dieser wird die Vektoren zunächst nur einlagern und auf Wunsch, zum Beispiel zur Anzeige auf dem LC-Display, die gesamte Karte ausgeben.

7. Nutzung der Mensch-Maschine Schnittstelle für Debug-Zwecke. Hier ist allein das LC-Display sinnvoll und zuständig.

Diese Verteilung der Aufgaben führt zu folgendem Kollaborationsdiagramm in Abbildung 4.2.

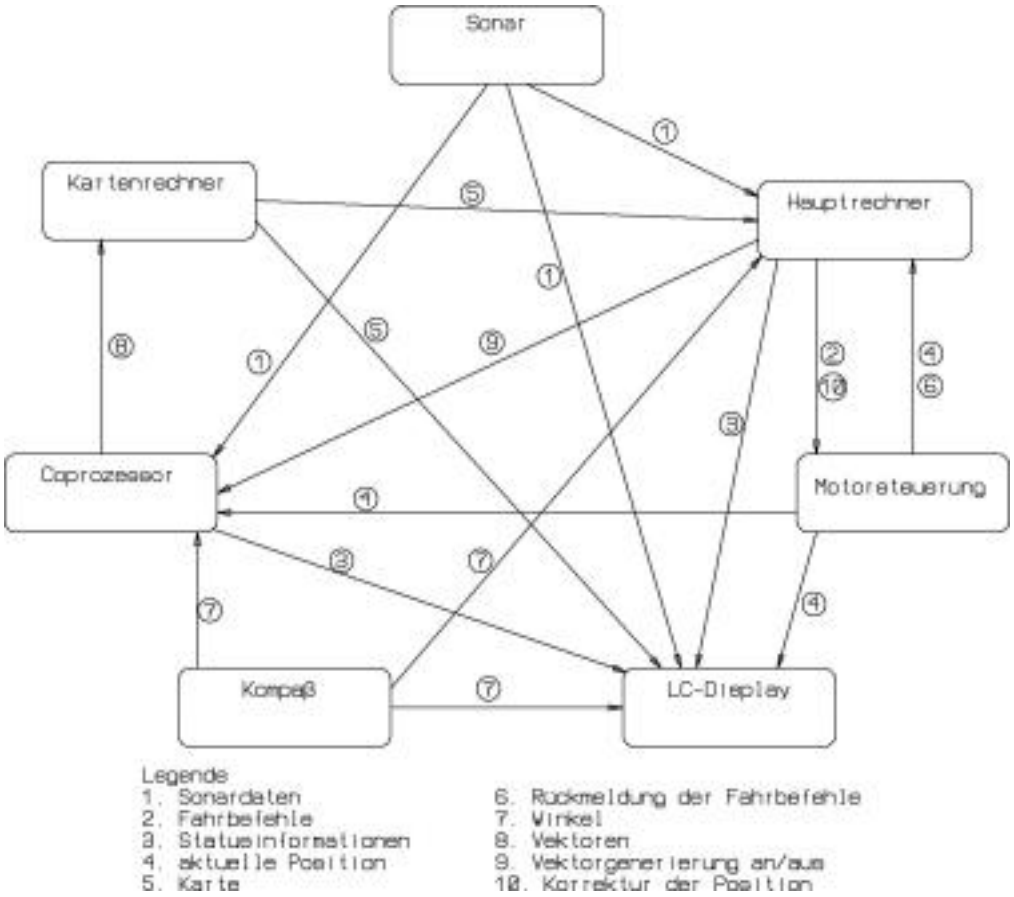


Abbildung 4.2.: Kollaborations-Diagramm

Diese Abbildung darf nicht darüber hinwegtäuschen, daß zum Beispiel Sonardaten von *jedem* Modul gelesen werden können und die hier eingezeichneten Datenflüsse nicht explizitem Versand an nur einige Module entsprechen. Vielmehr soll sie verdeutlichen, welches

Modul welche Informationen benötigt, um die ihm gestellte Aufgabe lösen zu können. Außerdem sind nicht alle Befehlsflüsse berücksichtigt, sondern nur die zur Kartographie essentiellen.

Im folgenden wird nun für jedes Modul eine Analyse der möglichen Lösungen ihrer Aufgaben, ihre Schnittstellen, sowie die eingesetzte Algorithmik näher beschrieben. Aufgrund der Größe des gesamten Projektes wird auf eine vollständige Analyse des Softwaredesigns bewußt verzichtet, da es den Rahmen dieser Arbeit wohl sprengen würde. Vielmehr sollen Architekturmodelle und Pseudocode-Sequenzen die grundsätzliche Arbeitsweise der einzelnen Module schematisch aufzeigen. Gerade der Pseudocode ist stark abstrahiert, da er sonst nicht in vernünftigem Umfang darstellbar wäre.

An dieser Stelle ändert sich die Reihenfolge der Modulbeschreibungen. Zuerst kommen die Module an die Reihe, die keine oder nur wenige Daten anderer benötigen um ihre Aufgabe zu erfüllen. Erst dann folgen die, die mit diesen Daten arbeiten.

4.3.1. Laderechner und Kompaß

Laderechner und Kompaß spielen im weiteren für das Projekt keine signifikante Rolle. Der Laderechner überwacht lediglich die Akkumulatoren und schaltet den Roboter ab, sollte eine Tiefenentladung drohen.

Die grundsätzliche Messung mit dem Kompaß wurde bereits bei der Beschreibung der Module in Kapitel 3.3.3 erläutert. Die bei der Implementierung auftretenden Probleme verhinderten eine Weiterentwicklung, so daß auf die Kompaßwerte zur Zeit leider verzichtet werden muß. Dies stellt eine starke Einschränkung dar, sollten seine Werte doch den Fehler kompensieren, der bei Drehungen des Roboters entsteht.

4.3.2. Sonar

Dem Sonar fällt die Aufgabe zu, mit seinen Sensoren die Umwelt abzutasten und Abstandswerte zwischen Sensor und Objekt bereitzustellen. Hierzu muß, wie in Kapitel 3.3.2 bereits erläutert, von einem Ultraschallsender ein Schallimpuls ausgesandt werden und aus den empfangenen Schallwellen ein Abstandswert ermittelt werden.

Grundsätzlich markiert der Zeitpunkt, an dem die gemessene Amplitude stark ansteigt, den Abstand des Objektes. Der so resultierende Abstand wird mittels der Formel

$$\boxed{s_{abs} = \frac{v_{Schall} * t_{Timer}}{2}} \tag{4.1}$$

errechnet. Bei einer Schallgeschwindigkeit in der Luft von $330\frac{m}{s}$, benötigt der Schall 60μs um 1cm absoluten Abstand zu durchlaufen. Um also eine Sensorauflösung von 1cm zu erreichen, was als ausreichend erscheint, muß mindestens alle 60μs eine Messung erfolgen. Diese Zeitbasis läßt sich softwareseitig mittels eines Timers und zugehöriger Interrupt-Funktion oder über die Laufzeit des Programms definieren. In der Praxis hat sich gezeigt, daß zudem ein dreifaches „oversampling“ die Ergebnisse verbessert. Das heißt, es werden pro Zentimeter nicht nur eine, sondern drei Messungen vorgenommen, was die Zeitbasis auf 20μs verringert. Diese höhere Auflösung wird jedoch nur intern zur Berechnung des Abstandes genutzt. Ausgegeben wird ein Abstandswert mit einer Auflösung von 1cm. Die Realisierung der Meßfunktion mittels Timer erscheint zunächst als die elegantere Lösung, hat aber den Nachteil daß die Einsprungzeit in die Interruptfunktion zwischen 5 und 8 Maschinenzyklen liegt, was maximal 6μs entspricht. Bei einer Zeitbasis von 20μs ist das ein Fehler von 30%, was als inakzeptabel erscheint. Deshalb sollte die Zeitbasis zunächst über die Programmlaufzeit erzeugt werden, was aber eine Sperrung aller Interrupts nötig macht, also auch des Netzwerks. Dies kollidiert hier allerdings mit dem Bestreben, dem Netzwerk die Priorität einzuräumen (Kapitel 3.4.3), da nur ein schnell umlaufendes token die Kommunikation im Gesamtsystem aufrecht erhalten kann. Trotzdem wurde dies als erster Ansatz gewählt.

Ein weiterer wichtiger Punkt ist die Wiederholfrequenz, also Anzahl Messungen pro Sekunde. Da alle Sensoren mit der gleichen Schallfrequenz von 41kHz arbeiten, muß die Messung mit einem Sensor komplett abgeschlossen sein, um den nächsten aktivieren zu können, sonst würden Schallwellen des einen Sensors die Messung des nächsten beeinflussen. Bei einer Meßweite von 255cm benötigt der Schall etwa 15ms. Alleine die reine Meßzeit aller Sensoren liegt somit bei ungefähr 120ms oder 8.3Hz, was an sich schon als wenig erscheint. Die Auswertung der Meßwerte sollte also möglichst schnell gehen, um die Wiederholfrequenz nicht noch weiter zu verringern.

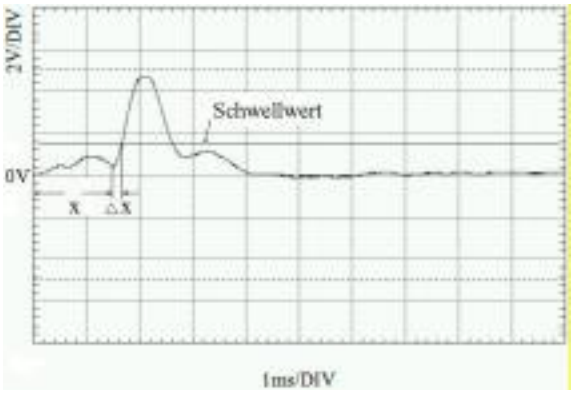
Das Hauptproblem bei Auswertung einer Meßreihe besteht darin, daß die Meßwerte ein Verhalten ähnlich einer Glockenkurve mit geringer Flankensteilheit aufweisen, wohingegen eine hohe Flankensteilheit mit geringer Breite wünschenswert wäre. Drei Faktoren

beeinflussen die Meßwerte maßgeblich:

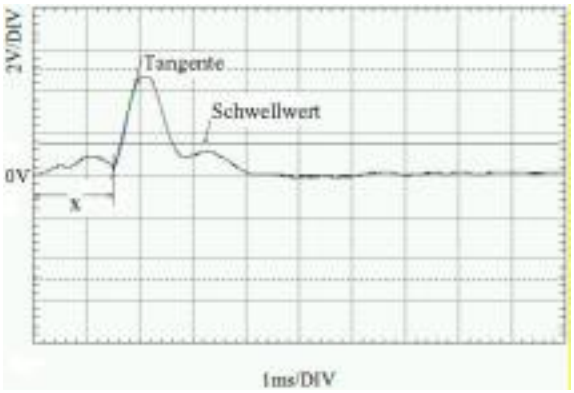
- Die Flankensteilheit nimmt mit zunehmendem Abstand zum Objekt ab aufgrund der geringeren Schallintensität.
- Der Winkel (Form) des Objektes zum Meßsystem ist ebenso wichtig, da ein Teil des Schalls wegreflektiert wird, was ebenfalls die Flanke negativ beeinflußt.
- Die Sensoren besitzen eine Schallkeule an dessen Flanken die Empfindlichkeit stark sinkt. Ein Objekt am Rande dieser Keule reflektiert nur wenig Schall und hat eine entsprechend geringe Amplitude zur Folge.

Es muß somit ein Verfahren entwickelt werden, daß diesem Verhalten Rechnung trägt und diese negativen Parameter möglichst vollständig eliminiert und nebenbei noch schnell ist.

Die wohl einfachste Methode zur Auswertung ist das Schwellwertverfahren. Hier wird ein statischer Schwellwert definiert, dessen Überschreitung einen Kontakt, also ein Objekt markiert. Bild 4.3.1 zeigt eine echte Messung mit einem Objekt in einem Abstand von 25cm.



4.3.1: Schwellwertverfahren



4.3.2: Tangentenverfahren

Abbildung 4.3.: Auswertung der Sonar-Daten

Hier zeigt sich auch die große Schwäche dieser Methode. Der eigentlich zu ermittelnde Abstand liegt bei 1,5ms, was einem Abstand von 25cm entspricht und hier mit x bezeichnet wird. Der Schwellwert wird aber erst nach etwa 1,7ms überschritten, was etwa 28cm entspricht und so ein Δx von 3cm zur Folge hat. Dies ist bedingt durch die geringe Flankensteilheit und bei weiter entfernten Objekten und ungünstigerem Winkel des Objektes wird dieser Effekt immer größer und läßt sich nicht eliminieren. Auch die Modifizierung,

ein Minimum zu suchen, das unmittelbar vor der Schwellwertüberschreitung liegt, führt zu ebenso schlechten Ergebnissen, da es nicht zwingend ein Minimum geben muß.
Als eher brauchbar hat sich eine Teilintegration der Meßwerte erwiesen, wie sie zum Beispiel in [4] vorgeschlagen wird. Die Flanke wird durch die Integration zwar steiler und so das Ergebnis besser, erreicht aber trotzdem nicht die erwünschte Genauigkeit. Allerdings sind dieses Verfahren unschlagbar schnell, da noch während der Messung ausgewertet werden kann.

Als insgesamt günstiger hat sich aber eine Mischung aus Schwellwert und einer Tangentenberechnung erwiesen. Zunächst wird das Signal auf das Überschreiten eines Schwellwertes hin untersucht. Ist dies der Fall, wird eine Tangente auf die Flanke gelegt, deren Schnittpunkt mit der Zeitachse den Zeitpunkt und damit den Abstand zum Objekt markiert. Hier ist es nicht unbedingt erforderlich die gesamte Flanke zur Berechnung heranzuziehen. Es reicht aus, die Flankensteilheit anhand zweier aufeinanderfolgender Meßwerte zu ermitteln. Dieses im Vergleich recht aufwendige Verfahren liefert zwar auch keine „perfekten“ Ergebnisse, hat sich aber als sehr viel genauer erwiesen. Es besitzt aber den Nachteil, daß erst alle Meßwerte aufgezeichnet werden müssen und erst dann eine Auswertung erfolgen kann. Bild 4.3.2 zeigt den Unterschied und Abbildung 4.4 zeigt einen Versuch, bei dem ein Objekt im einem konstanten Abstand am Sonar vorbei geführt wurde und die Abstandswerte, die beide Verfahren *qualitativ* liefern.

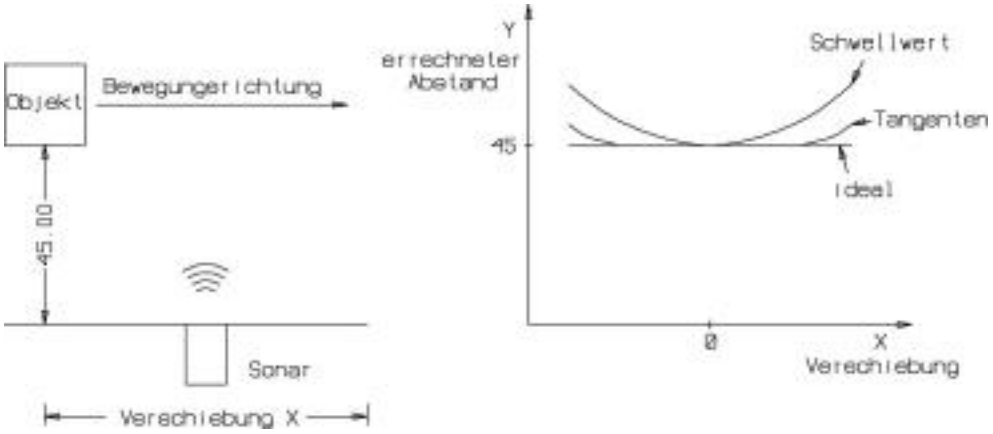


Abbildung 4.4.: Versuch zur seitlichen Verschiebung

In der Praxis hat sich gezeigt, daß höhere Meßauflösungen als 1cm nicht mit vertretbarem Aufwand machbar sind und die maximale Meßweite bei etwa 1,5m liegt. Ein Meßwert läßt sich so als einfaches Byte darstellen, wobei der Wert 255(cm) außerhalb der maximalen Meßweite liegt und als „kein Kontakt“ zu interpretieren ist.

Zusätzlich zur Abstandsermittlung ist es sinnvoll, eine erste Analyse dieser Meßwerte vorzunehmen, wobei zwei Charakteristika direkt aus den Meßwerten gewonnen werden

können:

- **Abstandsbereiche**. Sie sind hilfreich bei der Navigation, zum Beispiel zum Abbruch einer Fahrt aufgrund zu geringen Abstandes zu einem Objekt. Hierzu ist es nicht notwendig, den genauen Meßwert zu kennen, sondern die Information, ob ein bestimmter Abstand unterschritten wurde, reicht hier aus.
- **Verhalten** eines Wertes in Bezug auf vorherige Werte, also Annäherung an ein Objekt, „springen“ von Werten und ähnliches.

Zur Realisierung des ersten Punktes wurden insgesamt fünf verschiedene Abstandsbereiche definiert:

1. `ALERT` 0cm bis 4cm
2. `CLOSE` 4cm bis 12cm
3. `NEAR` 10cm bis 30 cm
4. `MIDDLE` 25cm bis 65cm
5. `FAR` 60cm bis 255cm

Das Verhalten wird in vier Typen unterteilt:

- `LINEAR` bedeutet das der Meßwert eines Sensors gleichmäßig von Messung zu Messung linear steigt oder fällt, wobei ein gewisser Fehler berücksichtigt wird.
- `PROGRESSIV` heißt, daß ein Sensorwert unverhältnismäßig stark fällt. Dieses Verhalten tritt auf, wenn ein Objekt „plötzlich“ im Meßbereich eines Sensors auftritt und der Sensorwert von einem großen Wert auf einen kleinen „springt“. Der Begriff der Progressivität, also eigentlich Vergrößerung oder Erhöhung, ist hier nicht ganz wörtlich zu nehmen, sondern aus Sicht des Roboters im Sinne einer Annäherung zu verstehen.
- `DEGRESSIV`, also das Gegenteil der Progressivität meint eine große Zunahme des Sensorwertes. Hier „verschwindet“ ein Objekt aus dem Meßbereich.
- `NONE` meint, daß der Meßwert keines der vorhergehenden Verhalten aufweist und somit keine Aussage möglich ist.

Netzwerk-Schnittstelle

Befehle : Zur Zeit keine.

Daten : Meßwerte aller acht Sensoren.
Verhalten für jeden Meßwert.
Abstandsbereich für jeden der der acht Sensorwerte.

Architekturmodell

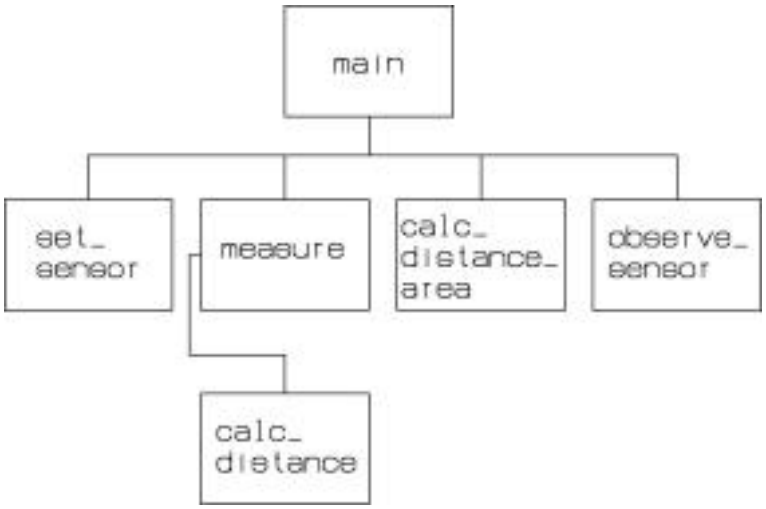


Abbildung 4.5.: Architektur des Sonar

Hauptprogramm

In einer Endlosschleife wird für alle acht Sensoren nacheinander zunächst das Sender-Empfänger-Paar aktiviert. Danach eine Messung durchgeführt, woran sich die Ermittlung des Distanzbereichs anschließt. Zuletzt wird das langfristige Verhalten des Sensors klassifiziert und alle ermittelten Daten in das Netzwerk geschrieben.

```
void main(void)
{
  init();

  while(1) {
    /* Alle acht Sensoren nacheinander durchgehen */
    for(Alle Sensoren (0-7)) {
      set_sensor(sensor_nr);                          /* Sender-Empf.-Paar aktiv. */
      distance = measure(Ping-Laenge, sensor_nr);     /* Messen */
      area     = calc_distance_area(distance);       /* Abstandsbereich ermit. */
      behavior = observe_sensor(sensor_nr,distance);/* Sensorverhalten klass. */
```

```
      /* Daten in das Netz stellen */
      Struktur sperren;          /* lock_data(ROBSONAR_DATA) */
      Messwert eintragen;
      Abstandsbereich eintragen;
      Verhalten eintragen;
      Struktur freigeben;        /*unlock_data(ROBSONAR_DATA);
    }
  }
}
```

Meßprozedur

Diese Funktion nimmt eine Meßreihe eines Sensors auf. Hierzu wir zunächst der Ultraschallsender eingeschaltet. Nun beginnt eine Schleife die 510 mal wiederholt wird und pro Durchlauf 20μs benötigt. In ihr wird immer *ein* Meßwert des A/D-Umsetzers aufgenommen und gespeichert. Zudem wird nach „Ping-Länge“ Durchläufen der Ultraschallsender wieder abgeschaltet. Nach der Aufnahme der Meßreihe wird die Funktion `calc_contact()` aufgerufen, die diese Meßreihe auswertet.

```
unsigned char measure(Ping-Laenge, Sensor-Nummer)
{
  ARCNET-Interrupt sperren;
  Ultraschallsender einschalten;

  for(510 Wiederholungen) {
    if(Ping-Laenge > 0) Ping-Laenge dekrementieren;
    else Sender ausschalten;

    A/D Wandlung starten;
    Datum in Array speichern;
    A/D Wandler ausschalten;
  }
  ARCNET-Interrupt freigeben;
  Distanz = calc_contact();
  return Distanz;
}
```

Auswertung der Meßreihe

Die Auswertung erfolgt in zwei Schritten. Erst einmal wird nach dem Überschreiten eines Schwellwertes gesucht. Dies ist die Kontaktbedingung. Ist ein Wert gefunden, wird die Steigung der Flanke berechnet (Tangente) und der Schnittpunkt mit der X-Achse bestimmt. Das Ergebnis wird durch 3 geteilt, um das oversampling zu eliminieren. Daraus resultiert der eigentliche Abstand zum Objekt.

```
unsigned char calc_contact()
{
  for(Alle 510 Werte) {
    if(Wert[n] > Schwellwert) {
      Steigung bestimmen;
      Schnittpunkt mit X-Achse berechnen (Abstand);
      Abstand durch 3 teilen (oversampling eliminieren);
      return Abstand;
    }
  }
  return 255; /* keine Objekt (Kontakt) */
}
```

Klassifizierung der Meßwerte

Grundlage der Klassifizierung ist die Meßwertdifferenz, also die Änderung des Meßwertes in Bezug auf den vorherigen Wert. Dies entspricht der ersten, beziehungsweise zweiten Ableitung, wobei letztere bei linearem Verhalten Null ist.

```
unsigned char observe_sensor(Sensor-Nr, Distanz)
{
  if(Objekt gefunden) {
    Vorherige deltas sichern;
    Frische deltas berechnen (erste und zweite Ableitung);

    if(Lineares Verhalten (zweite Ableitung = 0)) {
      Sensor verhaelt sich LINEAR;
    }
    else {
      if(Zweite Ableitung > 0) {
        Sensor hat DEGRESSIVES Verhalten (stark entfernend);
      }
      else {
        if(Zweite Ableitung < 0) {
          Sensor hat PROGRESSIVES Verhalten (stark annaehernd);
        }
      }
    }
  }
  else { /* kein Objekt im Messbereich */
    Keine Aussage moeglich (Verhalten = NONE);
  }
  return Verhalten;
}
```

Ein Nachteil dieses einfachen Verfahrens ist das „Springen“ zwischen progressivem und degressivem Verhalten. Ein kleines Beispiel soll dies verdeutlichen:

Messung-Nr	Meßwert	$\dot{x}$	$\ddot{x}$	Klassifizierung	Bemerkung
1	30	-	-	`NONE`	korrekt
2	32	2	-	`NONE`	korrekt
3	34	2	0	`LINEAR`	korrekt
4	50	16	14	`DEGRESSIV`	korrekt
5	50	0	-16	`PROGRESSIV`	falscher Sprung
6	50	0	0	`LINEAR`	korrekt

4.3.3. Motorsteuerung

Die Motorsteuerung treibt den Roboter an. Diese Aufgabe wird somit den Kern der Software bilden. Weiterhin soll dieses Modul die aktuelle Position im Koordinatensystem berechnen und auch während einer Fahrt ausgeben. Daß diese Berechnung genau genommen nur eine Schätzung ist, liegt an der Tatsache, daß jede Bewegung zwar berechnet, aber immer auch fehlerbehaftet ist. Diese Koppelnavigation kann also nur im beschränktem Umfang wirklich präzise Ergebnisse liefern.

Bei Geradeausfahrten liegt die Hauptfehlerquelle bei dem Unterschied zwischen theoretischer Weglänge pro Schritt der Motoren und tatsächlichem Weg. Hier tritt ein kumulativer Längenfehler auf, der aber eher klein ist. Schlimmer ist dagegen der Fehler bei gegenläufiger Ansteuerung der Motoren, also einer Drehung des Roboters. Der durchlaufene Winkel ist stark vom Reibungskoeffizienten des Untergrundes abhängig und nur experimentell ermittelbar.

Der wohl wichtigste Aspekt bei dem Design der Software ist das zeitkritische Verhalten der Schrittmotoren. Gerade bei höheren Geschwindigkeiten benötigen sie eine absolut gleichmäßige Ansteuerung. Eine Verzögerung kann hier das Stehenbleiben der Motoren zur Folge haben und da es keine Rückmeldung zwischen tatsächlicher Bewegung und der gewünschten (berechneten) Bewegung gibt, würde sich der Roboter völlig unkontrolliert verhalten. Die Ansteuerung der Motoren sollte also die Priorität bekommen.

Die sich daraus ergebende Problematik mit dem Netzwerk wurde bereits bei der Beschreibung des Sonar-Algorithmus in Kapitel 4.3.2 angesprochen. Auch hier wird zunächst ein Kompromiß geschlossen. Die Ansteuerung erhält zwar die Priorität und kann somit das Netzwerk ausbremsen, jedoch wird sie auf Geschwindigkeit hin optimiert, um diese Verzögerung im Mikrosekunden-Bereich zu halten. Dies verhindert jedoch rechenintensive Beschleunigungs- und Abbremsvorgänge die, wie in Kapitel 3.3.1 beschrieben, nötig sind, um Geschwindigkeiten über $10\frac{cm}{s}$ zu erreichen. Der Roboter wird somit nur Fahrten im Start-Stop-Betrieb mit dieser Geschwindigkeit als Maximum fahren können. Zudem werden beide Motoren immer gleichzeitig bearbeitet, was die Bewegungsfreiheit des Roboters auf Geradeausfahrten und Drehungen auf der Stelle begrenzt. Zur Berechnung des Schritt-Taktes wird einer der Timer im Prozessor herangezogen. Seine Interruptfunktion errechnet den nächsten Schritt *beider* Motoren und gibt diese Bit-Kombination an die Leistungselektronik weiter.

Um eine möglichst komfortable Kommunikation mit den navigierenden Modulen zu ermöglichen, muß die Motorsteuerung „abstrakte“ Befehle verstehen:

- `MOVE_FORWARD`
- `MOVE_REVERSE`

- TURN_LEFT
- TURN_RIGHT
- STOP

Alle Befehle benötigen zudem die Informationen der zu fahrenden Länge in Millimetern beziehungsweise des Winkels in Grad und der gewünschten Geschwindigkeit. Sie wird in Prozent angegeben, wobei 100% einer Geschwindigkeit von $10\frac{cm}{s}$ entspricht. Ist ein Befehl, bis auf STOP, erfolgreich abgearbeitet worden, sendet das Modul ein „POSITION_REACHED“-Signal an das Modul, das den Fahrbefehl gesendet hat.

Da die Selbstlokalisation Sache des Hauptrechners ist, wird eine Funktionalität benötigt, die eine Korrektur der geschätzten Position ermöglicht. Diese hat den Namen SET_GLOBAL_POSITION und enthält die neue X-, Y-Position und den neuen Winkel.

Der Befehl RECALC_GLIDE_FACTOR dient zur Neuberechnung des Gleitfaktors. Er ist nötig, um eine Umrechnung bei Drehungen von Schritten in Grad zu ermöglichen. Dieser Faktor ist nur experimentell ermittelbar und für Untergründe mit Fliesen liegt er etwa bei 6,5. Mit diesem Standardwert wird zunächst gearbeitet. Zur Neuberechnung benötigt die Motorsteuerung die Information welcher Winkel durchfahren sein sollte und welcher Winkelunterschied besteht. Hieraus wird nun den Gleitfaktor korrigiert:

$$\boxed{\mu_{neu} = \frac{\alpha_{soll}}{\alpha_{soll} + \Delta\alpha} * \mu_{alt}} \tag{4.2}$$

Netzwerk-Schnittstelle

Zur Information der anderen Module über das aktuelle Geschehen legt die Motorsteuerung eine ganze Reihe von Daten auf dem Netzwerk offen. Die wohl wichtigste Information ist die aktuell geschätzte Position im Koordinatensystem.

Befehle : Move-Befehle (STOP, MOVE_FORWARD etc).

SET_GLOBAL_POSITION setzt die Position im Koordinatensystem neu.

RECALC_GLIDE_FACTOR berechnet den Gleitfaktor neu.

Daten : X-, Y-Position und Winkel im Koordinatensystem.
Aktuelle Fahrtrichtung, Geschwindigkeit und gefahrene Länge.

Architekturmodell

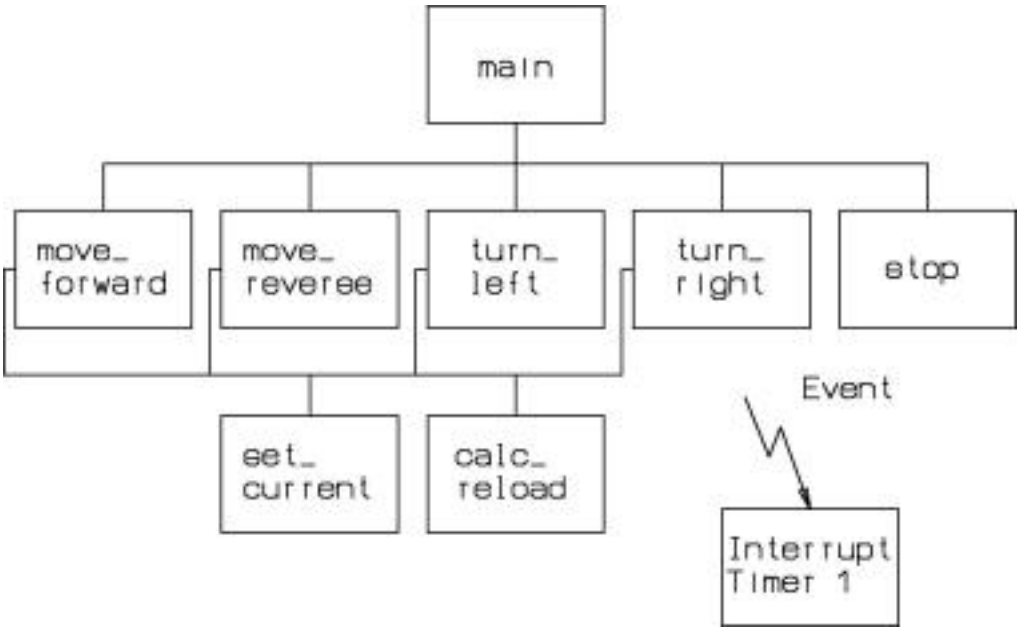


Abbildung 4.6.: Architektur der Motorsteuerung

Hier finden sich die einzelnen Fahrbefehle wieder, sowie die Ansteuerung der Motoren mit der Interruptfunktion. Die Funktion `set_current()` setzt die zulässigen Ströme der Motoren. Der Rückladewert des Timers und damit die Fahrgeschwindigkeit des Roboters berechnet `calc_reload()`.

Hauptprogramm

Sie gestaltet sich relativ einfach und wartet, bis auf ein paar Initialisierungen, nur auf Befehle. Alle Fahrbefehle rufen gleichnamige Funktionen auf, die die weiteren Berechnungen übernehmen.

```
void main()
{
  init();
  Befehlsempfang freischalten;

  while(1) {
    if(Befehl empfangen oder eine Fahrt aufgrund eines neuen Befehls
       abgebrochen wurde) {
      switch(Befehl) {
        case MOVE_FORWARD :
```

```
          move_forward(Geschwindigkeit, Laenge);
          break;

        case MOVE_REVERSE :
          move_reverse(Geschwindigkeit, Laenge);
          break;

        case TURN_LEFT :
          turn_left(Geschwindigkeit, Winkel);
          break;

        case TURN_RIGHT :
          turn_right(Geschwindigkeit, Winkel);
          break;

        case STOP :
          stop();
          break;

        case SET_GLOBAL_POSITION :
          Globale Positionswerte uebernehmen;
          break;

        case RECALC_GLIDE_FACTOR :
          Gleitfaktor korrigieren;
          break;

        default : /* falscher Befehl */
          Befehlsempfang freischalten;
          break;
      }
    }
  }
}
```

Move_Forward

Sie soll stellvertretend für alle Bewegungsfunktionen hier etwas näher erläutert werden.

Nach der Vorbereitung der eigentlichen Bewegung, also Umrechnung von Länge in Schritte, Festlegung der Motorströme und Berechnung der Timer-Geschwindigkeit, wird der Timer gestartet. Seine Interruptfunktion gibt die Schritte aus und wird noch erläutert. Eine while-Schleife wartet auf eine Rückmeldung der Interruptfunktion, die bedeutet, daß alle Schritte abgefahren wurden. Zusätzlich kann die Schleife beendet werden, wenn ein Befehl empfangen wird, der *nicht* `MOVE_FORWARD` lautet, also zum Beispiel `STOP`.

Während des Wartens wird ständig die neue aktuelle Position berechnet und auf dem Netzwerk ausgegeben.

```
unsigned char move_forward(Geschwindigkeit , Laenge)
{
  Umrechnung von Laenge in Schritte;
  Gefahrene Schritte auf Null setzen;
  Drehrichtung Motor links auf REVERSE;
  Drehrichtung Motor rechts auf FORWARD;

  set_current(MAX_DRIVE_CURRENT); /* zulaessiger Strom der Motoren einstellen */
  calc_reload(Geschwindigkeit);   /* Rueckladewert des Timers berechnen */
  Timer starten;

  Aktuelle Bewegung und Geschwindigkeit auf dem Netz ausgeben;
  Befehlsempfang freischalten;

  while(Flag der Interruptfkt noch nicht TRUE) {
    Globale Position berechnen und auf dem Netzwerk ausgeben;

    if(Befehl empfangen) {
      if(Befehl ungleich MOVE_FORWARD) {
        Timer stoppen;
        Motoren ausschalten;
        Globale Position (letzte) berechnen und ausgeben;
        return FALSE;
      }
      else { /* Befehl falsch */
        Befehlsempfang freischalten;
      }
    }
  }
  Timer stoppen;
  Motoren ausschalten;
  Globale Position (letzte) berechnen und ausgeben;
  position_reached(); /* Signal an Sender senden */
  return TRUE;
}
```

Interruptfunktion

Das Herzstück der Motorsteuerung bildet die Interruptfunktion des Timers 1, die den nächsten Schritt beider Motoren ermittelt und ausgibt. Die Bitkombination wird an Port 1 ausgegeben, wobei das untere Nibble die Kombination für den linken Motor

und das oberer für den rechten Motor steht. Rückladewert des Timers und damit die Geschwindigkeit des Roboters, Anzahl zu fahrender Schritte und die Drehrichtung von Motor links und rechts wurden vorher zum Beispiel von der Funktion `MOVE_FORWARD` berechnet und in globalen Variablen abgelegt.

```
void move() interrupt 1
{
  Timer stoppen und nachladen;

  switch(Drehrichtung Motor links) {
    case FORWARD :
        Oberes Nibble berechnen;
      break;
    case REVERSE :
        Oberes Nibble berechnen;
      break;
  }

  switch(Drehrichtung Motor rechts) {
    case FORWARD :
        Unteres Nibble berechnen;
      break;
    case REVERSE :
        Unteres Nibble berechnen;
      break;
  }

  if(noch Schritte zu fahren) {
    Gefahrene Schritte inkrementieren;
    Bitkombination an Port 1 ausgeben;
    Timer wieder starten;
  }
  else {
    Flag setzen;
    Port 1 loeschen;
  }
}
```

4.3.4. Coprozessor

Dem Coprozessor obliegt es, die einzelnen Vektoren zu generieren, die in ihrer Gesamtheit die Karte bilden sollen. Da die Entscheidung, wann Vektoren zu erzeugen sind, nicht bei ihm liegt, sondern Aufgabe der navigierenden Komponenten, muß die Vektorerzeugung Ein und Aus zu schalten sein. Hierzu wurden die Befehle `START_SCAN` und `STOP_SCAN` implementiert. Bleibt die Frage zu klären, auf welche Art und Weise Vektoren zu erzeugen sind, wobei die Echtzeitfähigkeit die wohl wichtigste Rolle hierbei spielt.

Ein möglicher Ansatz ist erst einmal, alle Daten des Sonars bei einer Fahrt zu sammeln und später auszuwerten. Dies bedingt aber einen erheblichen Rechenaufwand, der die Echtzeitfähigkeit gefährdet. Deshalb wurde ein Ansatz entwickelt, der schnelle Ergebnisse liefert. Die Idee ist, das *Verhalten* der Sonarwerte, die das Sonarmodul bereits ausgibt, als Grundlage zu nehmen, wobei die Aufzeichnung zunächst auf die Werte der Seitensonare beschränkt wird und bei Vorwärtsfahrt erfolgen soll.

Grundsätzlich sollen Eckpunkte gesucht werden, die Anfang, respektive Ende einer stetigen Sensorwertfolge sind, was einem linearen Verhalten entspricht. Dies wird in der Literatur als **Edge-Following** bezeichnet. Diese Eckpunkte entsprechen den Anfangs- und End-Punkten der gesuchten Vektoren, die mittels aktueller Position der Motorsteuerung und des eigentlichen Abstandes zum Objekt, in globale Koordinaten umgerechnet werden. Hierzu wird das Verhalten der Seitensonare beobachtet. Wird nun ein lineares Verhalten erkannt, so ist ein Anfangspunkt gefunden, der Meßwert und die globale Position des Roboters werden gespeichert. Das Verhalten wird nun solange beobachtet, bis es nicht mehr linear ist, was dem Finden eines Endpunktes gleichkommt. Auch hier wird die globale Position, sowie Abstandswert gespeichert. Nun werden Anfangs- und End-Punkt in globale Koordinaten umgerechnet, als Vektor zusammengefaßt und an den Kartenrechner übergeben.

Würde dies für alle vier Sensoren einzeln gemacht, würden zwangsläufig doppelte Vektoren auftreten, da die Meßwerte am vorderen Seitensensor mit einer Verzögerung auch am hinteren Seitensensor auftreten. Theoretisch sind diese Werte identisch, was eben auch gleiches Verhalten und somit gleiche Vektoren zur Folge hätte. Aus diesem Grunde wird das Verhalten der Sensoren getrennt nach Seiten zusammengefaßt. Ist nun das Verhalten beider Sensoren linear, so definiert der Meßwert des hinteren Sensors die Anfangsposition eines Vektors. Bei Änderung in nichtlineares Verhalten gibt der Meßwert des vorderen Sensors den Endpunkt an (bei Vorwärtsfahrt muß sich das Verhalten bei ihm zwangsläufig zuerst ändern).

Die Verhaltensweisen (Hypothesen) beider Seitensensoren wird in vier Kategorien eingeteilt:

- `WALL` wird angenommen, wenn *beide* Sensoren lineares Verhalten zeigen.

- `OBSTACLE` wird gewählt, wenn ein Objekt plötzlich am vorderen Sensor auftaucht. Dies korreliert mit einem progressiven Verhalten der Meßwerte.
- `CORNER` wird bei degressivem Verhalten angenommen.
- `NONE` wenn keine Aussage möglich ist.

Zusätzlich gibt ein „Vertrauenswert“ an, wie lange ein solches Verhalten gefunden wurde und ein Distanzbereich, analog zum Distanzbereich eines Sensors, in welcher Distanz sich dieses Objekt befindet. Diese Daten werden für beide Seiten auf dem Netzwerk ausgegeben und andere Module, zum Beispiel der Hauptrechner, können sie nutzen, um ein Wall-Following zu implementieren oder zumindest zu unterstützen.

Das gewählte Verfahren hat den Vorteil, daß es extrem schnell und somit für die Echtzeitverarbeitung bestens geeignet ist. Außerdem wird die rechenintensive Umrechnung der Anfangs und End-Punkte in globale Koordinaten nur einmal bei Entdeckung eines Vektors durchgeführt, was den Rechner nicht allzu sehr belastet.
Diese Art der Datenaufzeichnung bietet aber nur eine grobe Auflösung, da *beide* Sensoren lineares Verhalten zeigen müssen, um Vektoren zu generieren und Objekte von beiden gleichzeitig „gesehen“ werden müssen. Der Abstand der Sensoren gibt hier eine Mindestgröße der Objekte von etwa 10cm vor.

Netzwerk-Schnittstelle

Befehle : `START_SCAN` startet die Generierung von Vektoren.

`STOP_SCAN` stoppt diese wieder.

Daten : Kontaktart links (`WALL` etc.).
Vertrauenswert für links (0 bis 100).

Kontaktart rechts (`WALL` etc.).
Vertrauenswert für rechts (0 bis 100).

Um die Geschwindigkeit des Programms hoch zu halten, übernimmt das Hauptprogramm fast alles. Lediglich die Beobachtung der Sensoren wurde in eine Funktion ausgelagert, weshalb hier ein Architekturmodell unnötig erscheint.

Hauptprogramm

Das Hauptprogramm läßt ständig das Verhalten der Sensoren beider Seiten klassifizieren. Nur bei Vorwärtsfahrt und eingeschaltetem „SCAN“-Modus werden Anfangs- und Endpunkte gesucht und Vektoren generiert. Bei anderen Fahrten werden die Verhaltensweisen lediglich klassifiziert und in das Netzwerk kopiert.

```
void main()
{
  init();
  Befehlsempfang freischalten;

  while(1) {
    if(Befehl empfangen) {
      if(Befehl = START_SCAN) {
        Flag setzen;
      }
      if(Befehl = STOP_SCAN) {
        Flag loeschen;
      }
      Befehlsempfang freischalten;
    }

    switch(Aktuelle Fahrtrichtung) {
      case MOVE_FORWARD :
        if(Mehr als 1cm gefahren) {
          observe_side(Sensor links  vorne, Sensor links  hinten, LEFT);
          observe_side(Sensor rechts vorne, Sensor rechts hinten, RIGHT);

          if(Flag gesetzt) {
            if(Anfangspunkt linke Seite gefunden) {
              Position merken;
            }
            else {
              if(Endpunkt linke Seite gefunden) {
                Globale Position des Vektors berechnen;
                Vektor an Kartenrechner senden;
              }
            }

            if(Anfangspunkt rechte Seite gefunden) {
              Position merken;
            }
            else {
              if(Endpunkt rechte Seite gefunden) {
                Globale Position des Vektors berechnen;
                Vektor an Kartenrechner senden;
              }
            }
          }
        }
```

```
        break;

      case MOVE_REVERSE :
      case TURN_LEFT :
      case TURN_RIGHT :
        observe_side(Sensor links  vorne, Sensor links  hinten, LEFT);
        observe_side(Sensor rechts vorne, Sensor rechts hinten, RIGHT);
        break;
    }
    Verhalten in das Netz kopieren;
  }
}
```

Beobachtung der Seitensensoren

Diese Funktion stellt die Hypothesen für die Seitensensoren auf. Ihr Vorgehen ist analog zur Klassifikation des Einzelsensorverhaltens im Sonar-Modul. Zeigen beide Sensoren gleichartiges Verhalten, so wird dies gespeichert und der Vertrauenswert inkrementiert bis einer der Sensoren ein anderes Verhalten aufweist.

```
unsigned char observe_side(Sensor-Nr vorne, Sensor-Nr hinten, Seite)
{
  if(Beide Sensoren lineares Verhalten) {
    Verhalten = WALL;
    Vertrauenswert inkrementieren;
  }
  else {
    if(Falls einer degressiv und einer linear ist) {
      Verhalten = CORNER;
      Vertrauenswert inkrementieren;
    }
    else {
      if(Falls einer progressiv und einer linear ist) {
        Verhalten = OBSTACLE;
        Vertrauenswert inkrementieren;
      }
      else {
        if(Fall beide irgendein Verhalten) {
          Verhalten = NONE;
          Vertrauenswert inkrementieren;
        }
      }
    }
  }
```

```
  Distanzbereiche beider Sensoren vergleichen und kleinsten
  als Ergebnis speichern;
  return TRUE;
}
```

4.3.5. Kartenrechner

Der Kartenrechner wird die vom Coprozessor erzeugten Vektoren einlagern und auf Verlangen anderer Module die gesamte Karte ausgeben. Da die Verarbeitung der Karte noch nicht erwünscht ist, gestaltet sich eine erste Implementierung sehr einfach. So reicht es hier aus, die Vektoren mittels einer verketteten Liste zu speichern. Aufgrund der (noch) recht bescheidenen Funktionalität wird auf eine nähere Betrachtung verzichtet.

Netzwerk-Schnittstelle

Befehle : `CLEAR_MAP` löscht die gesamte Karte.

`VECTOR` mit den X- und Y-Koordinaten des Start- und End-Punktes, lagert einen neuen Vektor in die Karte ein.

`GET_MAP` fordert die gesamte Karte an. Daraufhin schickt der Kartenrechner einen `VECTOR_COUNTER`-Befehl mit der Anzahl der Vektoren an das anfordernde Modul. Danach folgt ein Vektor nach dem anderen, bis alle verschickt wurden.

Daten : keine

4.3.6. Hauptrechner

So unscheinbar der Hauptrechner seitens der Hardware ist, so wichtig ist er bei der Steuerung des Roboters. Ihm obliegt es, nach einer Strategie durch die Umwelt zu navigieren und sie so sukzessive zu erkunden.

Dies macht folgende Aufgaben nötig:

- Planung der nächsten Fahrt (Zielkoordinate oder einfach nur Fahrtrichtung).
- Erzeugung der dafür erforderlichen Befehlssequenzen.
- Initiierung der Fahrt, also Kommunikation mit der Motorsteuerung.
- Überwachung der Sensordaten, um im einfachsten Fall die Kollisionsfreiheit während der Fahrt zu gewährleisten.

Architekturmodell

Diese Spezifikation macht eine stufenweise Aufteilung sinnvoll. Hierzu wurden zwei Instanzen eingeführt, deren Namensgebung aus [3] entnommen wurde:

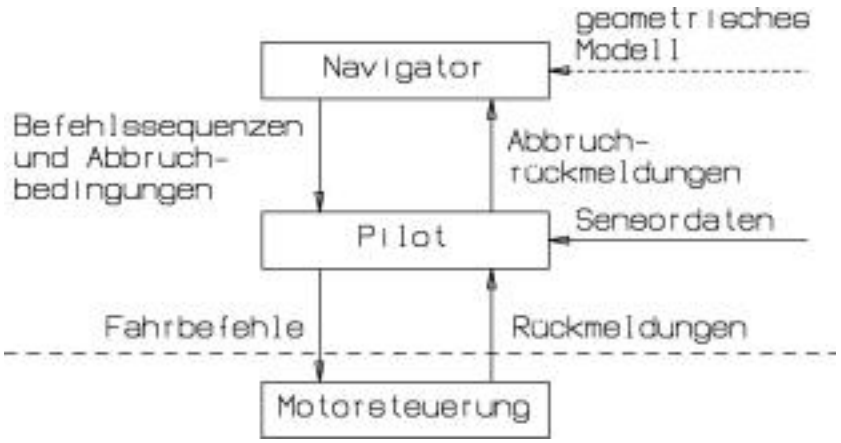


Abbildung 4.7.: Navigator und Pilot

- Der **Navigator** plant den nächsten Explorationsschritt und erzeugt für den Piloten Befehlssequenzen. Diese beinhalten die eigentlichen Fahrbefehle, sowie spezielle Abbruchbedingungen für diese Fahrten.

- Der **Pilot** versucht, diese Sequenzen abzuarbeiten. Zusätzlich überwacht er die Sensordaten, um Kollisionen zu verhindern und auf die Abbruchbedingungen des Navigators zu reagieren.

Diese Unterteilung hat zwei entscheidende Vorteile: Erstens läßt sie sich direkt auf mehrere Prozessoren abbilden, was dem Konzept des verteilten Systems entspricht. Die Kommunikation zwischen Navigator und Pilot, die zur Zeit innerhalb eines Prozessors geschieht, läßt sich ebenso über das Netzwerk realisieren.
Zweitens beinhaltet *nur* der Navigator die eigentliche Strategie. Eine Änderung dieser ist also nur an einer dedizierten Stelle notwendig, was sie extrem einfach und überschaubar macht und auch gilt, wenn in Zukunft zusätzliche Informationen durch Interpretation der Karte hinzu kommen (geometrisches Modell in Abbildung 4.7). Im schlimmsten Fall, nämlich bei Einführung spezieller Abbruchbedingungen, können diese auf einfache Weise im Piloten ergänzt werden. Eine komplettes Redesign ist nicht notwendig. Die Komplexität der Strategie ist also einzig und allein Sache des Nutzers und in erster Linie eine Frage wieviel Zeit er investieren möchte. Auch hier stand die geforderte Flexibilität des Gesamtsystems im Vordergrund.

Da die Entwicklung einer Strategie recht aufwendig ist und ständige Versuche in realer Umgebung mit sich bringt, wurde nicht nur ein Navigator und Pilot, sondern zwei „Pärchen“ implementiert. Der Nutzer kann nun an zwei völlig unterschiedlichen Strategien gleichzeitig arbeiten zum Beispiel eine komplexe Strategie und die zweite zum Testen von deren Unterstrategien. Eine andere Möglichkeit ist, an zwei prinzipiell gleichartigen zu arbeiten, an denen zum Beispiel Abbruchbedingungen oder partiell unterschiedliches Verhalten variiert wird. Die Strategie, die ein günstigeres Verhalten aufzeigt, wird weiter verfolgt, beziehungsweise ihre Parameter in die andere übernommen. Dies kann den Entwicklungsprozeß erheblich beschleunigen, da nicht bei jeder kleinen Variation gleich neue Software geschrieben werden muß.

Weil echtes Multitasking innerhalb eines Prozessors nicht zur Verfügung steht, wurde die Zusammenarbeit mittels kooperativem Multitasking realisiert. Navigator und Pilot wechseln sich ständig ab.

Navigator

Er beinhaltet die eigentliche Strategie, nach der die Umwelt erkundet wird. Sein innerer Aufbau ist zunächst nicht näher festgelegt und kann zum Beispiel mit einem endlichen Automaten realisiert werden, wobei jeder Zustand einem kleinen Teil der Strategie entspricht.

Innerhalb eines Zustands müssen zunächst eventuelle Fahrtabbrüche des Piloten behandelt werden. Erst dann wird, je nach Zustand, ein neuer Befehl oder eine ganze Sequenz von Befehlen für den Piloten erzeugt. Zusätzlich wird der Folgezustand des Navigators

festgelegt, woraufhin die Kontrolle, durch Verlassen der Funktion, an den Piloten übergeben wird.

Der Navigator für eine simple Strategie, die nur versucht Objekten auszuweichen und sich entweder 90° rechts oder 90° links dreht, sieht schematisch folgendermaßen aus:

```
unsigned char navigator_1()
{
  switch(Navigationsschritt)
  {
    case ENTRY :  /* Startschritt und Platz fuer Initialisierungen */
      Naechster Navigationsschritt ist DRIVE_FORWARD;
      break;

    case DRIVE_FORWARD :
      if(Pilot ALERT meldet) {
        Drehe ein Stueck zurueck;
      }
      else {
        Fahre Vorwaerts;
        Abbruch bei Annaeherung im CLOSE-Bereich;
        Vektoren generieren;
      }
      Naechster Navigationsschritt ist TURN;
      break;

    case TURN :
      if(Pilot ALERT meldet) {
        Fahre ein Stueck zurueck;
      }
      else {
       if(Links mehr Platz als rechts) {
          Drehe links 90 Grad;
          Abbruch bei Annaeherung im CLOSE-Bereich;
        }
        else {
          Drehe rechts 90 Grad;
          Abbruch bei Annaeherung im CLOSE-Bereich;
        }
      }
      Naechster Navigationsschritt ist DRIVE_FORWARD;
      break;
  }
  return TRUE;
}
```

Pilot

Er bekommt vom Navigator eine Liste von Befehlen, die er nacheinander abzuarbeiten hat. In einem Befehl stehen neben den Informationen zum eigentlichen Fahren sogenannte Spezialbefehle und Abbruchbedingungen. Die Spezialbefehle reduzieren sich zur Zeit auf das Ein- und Aus-Schalten der Vektorgenerierung im Coprozessor.
Die Abbruchbedingungen hingegen sind beispielsweise Unterschreitungen von Abständen oder das Auffinden einer Ecke. Lediglich eine Bedingung überwacht der Pilot von sich aus, auch ohne spezielle Angabe. Meldet ein Sensor eine Bereichsunterschreitung von 4cm (dies entspricht dem `ALERT`- Bereich), so bricht er auf jeden Fall die Fahrt ab und sperrt alle Fahrtrichtungen, bis auf die entgegengesetzte, die diese Situation hervorgerufen hat. Trat der Fehler also bei einer Vorwärtsfahrt auf, so wird der Pilot beim nächsten Aufruf nur eine Rückwärtsfahrt akzeptieren, um die prekäre Situation nicht noch zu verschlimmern. Es ist dann Aufgabe des Navigators dies zu veranlassen.

Werden alle Befehle ohne Fehler abgearbeitet, so wird die Kontrolle nun wieder dem Navigator überlassen, der die nächste Befehlsequenz erzeugen kann.

```
unsigned char pilot_1()
{
  Ersten Befehl holen;

  if(Vorher ALERT aufgetreten und Befehl falsch) {
    Abbruchparameter speichern;
    return FALSE;
  }
  else {
    while(Noch Befehle abzuarbeiten) {
      switch(Spezialbefehl) {
        case Vektoren generieren :
          Vektorgenerierung starten;
          break;
      }
      Fahrbefehl an Motorsteuerung schicken;
      while(Position noch nicht erreicht) {
        /* Abbruchbedingungen ueberwachen */
        switch(Abbruchbedingungen) {
          case Bedingung_1 :
            if(Bedingung_1 erfuellt){
              Fahrt stoppen;
              Abbruchparameter speichern;
              return FALSE;          /* Fahrt konnte nicht beendet werden */
            }
          case Bedingung_2 :
```

```
                .
                .
        }

        if(Ein Sensor auf ALERT) { /* Fahrt auf jeden Fall abbrechen */
          Fahrt stoppen;
          Abbruchparameter speichern;
          return FALSE;
        }
      }
      /* Ein Befehl ist erfolgreich abgearbeitet worden */

      switch(Spezialbefehl) {
        case Vektoren generieren :
          Vektorgenerierung stoppen;
          break;
      }
      Befehl fertig also naechsten holen;
    }
  }
  return TRUE;
}
```

Dem Benutzer steht so ein System zur Verfügung, dessen Explorationsverhalten auf einfache Art und Weise geändert werden kann. Es können so im Navigator auch vollständig andere Ansätze ausprobiert werden als der hier gezeigte einfache Automat. Zum Beispiel könnten Neuronale Netze oder ähnliches die Navigation oder zumindest einen Teil übernehmen[2], ohne gravierende Änderungen des Gesamtsystems zur Folge zu haben.

[2]Der Spieltrieb spielt hier eine nicht unerhebliche Rolle.

4.3.7. LC-Display

Als Schnittstelle zwischen Mensch und Roboter soll das LC-Display Befehle des Nutzers annehmen und interne Daten des Roboters zu Debug-Zwecken anzeigen. Um die Bedienung des Roboters möglichst einfach zu gestalten, sind alle Eingabe- und Anzeige-Möglichkeiten als Textmenüs realisiert. Die vollständige Menüstruktur ist in Abbildung 4.8 zu sehen. Auf die dazu erforderlichen Algorithmen (zum Beispiel Linienfunktion, Zeichensatz und ähnliches), wird aufgrund ihrer Größe nicht näher eingegangen und einfach als gegeben angenommen.
Jeder umrandete Menüpunkt stellt ein Untermenü dar und wird mittels zugehöriger Zahl auf der Tastatur angewählt, beziehungsweise mit „Escape“ wieder verlassen. Nicht umrandete Punkte sind einfache Befehle.

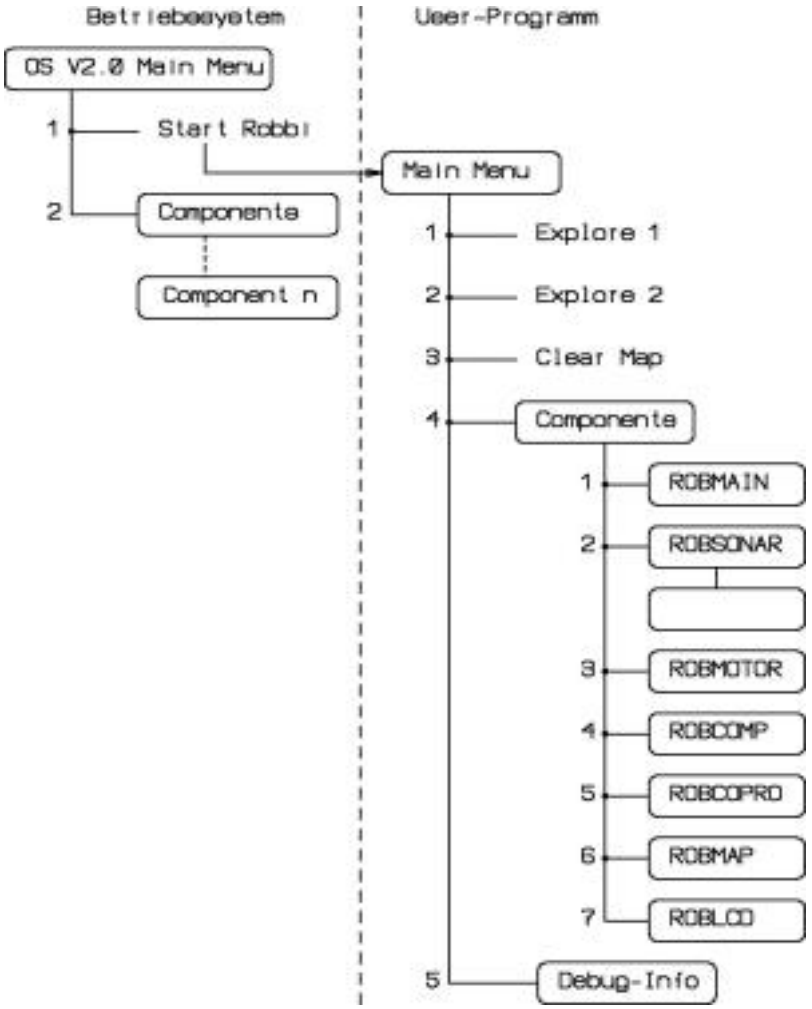


Abbildung 4.8.: Menüstruktur

Nach Einschalten des Systems meldet sich zunächst das Betriebssystem (OS V2.0 Main Menu). Hier kann ausgewählt werden, ob die User-Programme des Roboters gestartet (Start Robbi), oder das Netzwerk nach angeschlossenen Komponenten durchsucht werden soll (Components). Im letzteren Fall werden alle aktiven Rechner nacheinander mit ihrem Namen, ID und Datum der Software angezeigt. Bei „Start Robbi“ werden nun alle User-Programme aktiviert.

Das Hauptmenü „Main Menu“ bietet einige Möglichkeiten:

1. „Explore 1“ startet die erste Strategie.
2. „Explore 2“ fährt mit der zweiten Strategie los.
3. „Clear Map“ löscht alle im Kartenrechner eingetragenen Vektoren.
4. „Components“. Hierunter sind die Daten zu finden, die die einzelnen Komponenten auf dem Netzwerk offenlegen. Hier sind Sonarabstandswerte, aktuelle Position des Systems und nicht zuletzt auch die Karte unter „ROBMAP“ sichtbar. Zunächst muß aber noch die Komponente gewählt werden, deren Daten angezeigt werden sollen.
5. Menüpunkt „Debug-Info“ ist zur Zeit ohne Bedeutung und für Erweiterungen reserviert.

Der für die Aufgabe wichtigste Menüpunkt ist „ROBMAP“. Hier läßt sich die Karte anzeigen, verschieben und in ihrer Größe ändern. Abbildung 4.9 zeigt das LC-Display mit einer echten Karte und die Tastenbelegung zur Anzeige.

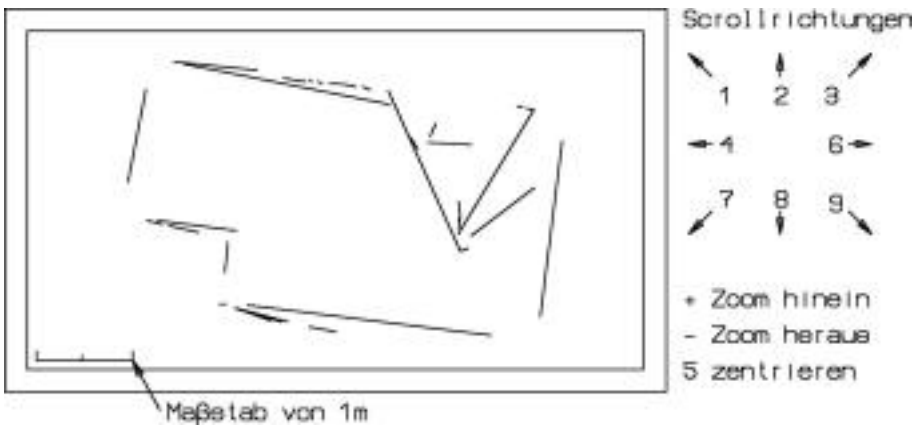


Abbildung 4.9.: Karte auf dem LC-Display

4.4. Ergebnisse

Um erste Ergebnisse zu erlangen, wurde der Roboter zunächst in einem relativ kleinen Szenario eingesetzt, das Abbildung 4.10 zeigt. Daß die Seiten in diesem Szenario alle or-

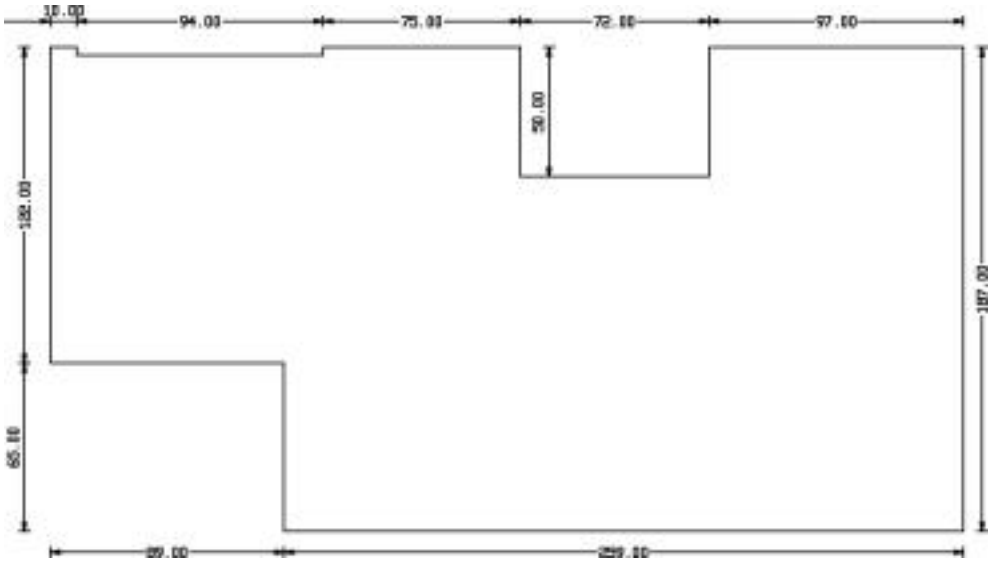


Abbildung 4.10.: Karte des ersten Testszenarios

thogonal sind, hat nur den Hintergrund, die Strategie zunächst einfach halten zu können und trotzdem Ergebnisse zu bekommen.

Als Teststrategie wurde ein einfaches Wall-Following implementiert, das versucht, immer der vom Roboter aus gesehenen linken Wand zu folgen. Im Grunde der Algorithmus zum Verlassen eines Labyrinths. Abbildung 4.11 zeigt ein typisches Ergebnis einer Testfahrt. Die gestrichelten Linien zeigen die überlagerte echte Karte, während die dickeren Linien die vom Roboter aufgezeichneten Vektoren darstellen.

Zunächst fällt auf, daß die Ecken nicht aufgezeichnet werden, was angesichts des Algorithmus zur Vektorgenerierung nicht verwundert, aber für wirklich genaue Karten auf Dauer nicht hinnehmbar ist. Desweiteren ist ein Winkelfehler zu beobachten, der bei längerer Fahrt immer größer wird und schon bei einem so kleinen Szenario recht große Fehler produziert. Hier zeigt sich deutlich die Schwäche der Vektorgenerierung, die mit ihrer Geschwindigkeit, und Einfachheit erkauft wurde. Als Gründe kommen mehrere Effekte in Frage:

- Die Routine zur Erfassung von Vektoren ist noch nicht optimal auf Geschwindigkeit getrimmt. Ihr können somit Endpunkte, beziehungsweise Anfangspunkte von Vektoren verloren gehen. So fehlt in der aufgezeichneten Karte eine ganze Ecke.

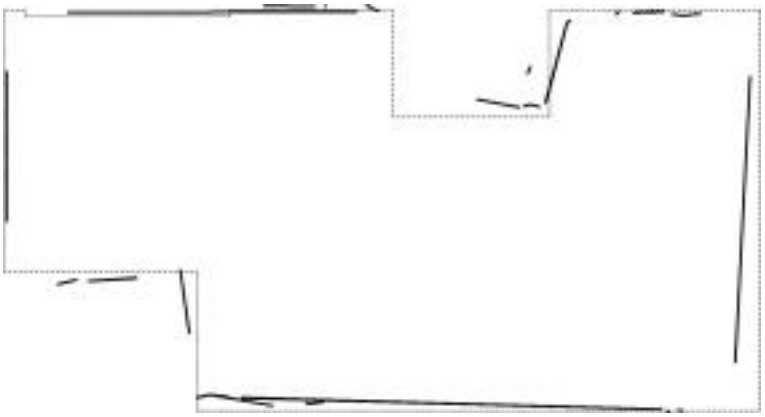


Abbildung 4.11.: Typisches Ergebnis einer Testfahrt

- Die Geschwindigkeit, mit der der Roboter fährt, hat Einfluß auf die Qualität der Vektorgenerierung, was im Grunde mit dem vorangegangenen Punkt zu tun hat.
- Die Ungenauigkeit der Sensoren, gerade an den Flanken ihrer Meßkeulen, verschlimmern die Situation noch zusätzlich. Da ihre Meßwerte bei Auffinden eines Vektoranfangspunktes fehlerbehaftet sind, aber als Grundlage der Koordinatenberechnung dienen, verschiebt sich der gesamte Vektor um diesen Fehler, was den Gesamtvektor unbrauchbar macht.

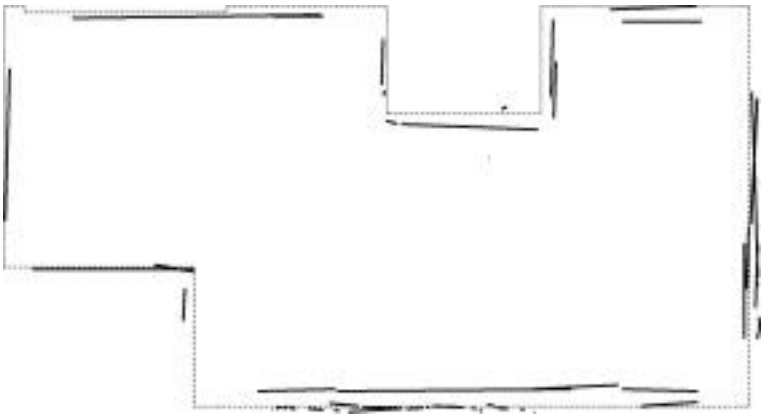


Abbildung 4.12.: Ergebnis mit verbessertem Winkelfehler

Um zumindest den Winkelfehler zu verringern, wurde die Strategie um eine Kalibrierungsfahrt erweitert. Sie versucht am Anfang den Gleitfaktor durch eine 360° Drehung zu bestimmen. Trotz einer leichten Verbesserung der Ergebnisse ist das Problem nicht in den Griff zu bekommen. Das Fehlen der Kompaßwerte, die dies ermöglichen sollten,

macht sich hier deutlich bemerkbar. Abbildung 4.12 zeigt ein etwas verbessertes Ergebnis, aber immer noch mit deutlichen Fehlern. Zudem hat sich bei einer ganzen Reihe von Versuchen gezeigt, daß auch die Kalibrierungsfahrt nicht reproduzierbare Verbesserungen liefert, da auch sie fehlerbehaftet ist.

Die Fehler in den Ergebnissen führen sich auch beim Umfahren eines kreisförmigen Objektes fort. Allerdings haben hier sicherlich die breiten Meßkeulen der Sonare eine erheblich Mitschuld am schlechten Ergebnis.

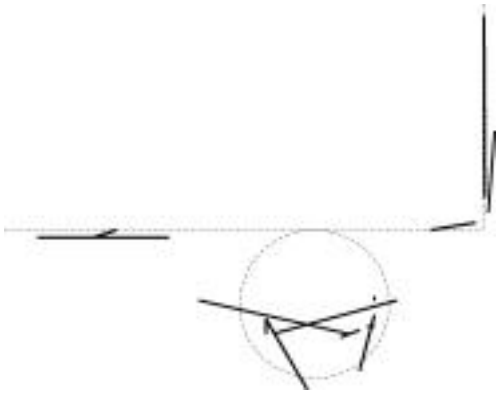


Abbildung 4.13.: Umfahren eines kreisförmigen Objektes

Ihre Stärke kann sie allerdings ausspielen, wenn es um das Abfahren einer schrägen Wand geht. Hier zeigt sich eine zum Teil erstaunliche Präzision.

Abbildung 4.14.: Abfahren einer Schräge

Insgesamt hat das System gezeigt, daß es in der Lage ist zu kartographieren. Hinsichtlich der Genauigkeit allerdings müssen zur Zeit noch einige Unzulänglichkeiten in Kauf genommen werden.

5. Zusammenfassung und Ausblick

Zusammenfassung

Das System hat sich insgesamt als sehr flexibel und ausbaufähig erwiesen und die Erwartungen voll erfüllt. Dies betrifft die Integration neuer Module genauso, wie die Verlagerung einzelner Algorithmen innerhalb des Systems. So wurde beispielsweise die Klassifizierung der Sonardaten, die im ersten Ansatz vom Coprozessor erledigt wurde, in das Sonarmodul verlegt, was sich als unproblematische und schnelle Modifikation erwiesen hat.

Allerdings zeigt das System noch einige „Kinderkrankheiten“, was angesichts der Gesamtkomplexität allerdings nicht weiter verwundert[1]. Zwei Fehler erschweren die Entwicklung maßgeblich:

- Das Download-Programm erzeugt Fehler bei der Übertragung der User-Programme.
- Das Netzwerk des Roboters läuft manchmal nicht sauber hoch. Dies hat zur Folge, daß Module nicht gefunden werden und nach diesem Fehler manchmal User-Programme zerstört werden. Hieraus folgen manchmal Fehler in der Datenaufzeichnung oder völlig unvorhersehbares Verhalten des Roboters.

Auch der Charakter des embedded-Systems und die damit verbundene Schwierigkeit, Daten und damit Fehler sichtbar zu machen, erschwert die Implementierung neuer Algorithmen.

Die Kartographie hat sich insgesamt als sehr große und auch schwierige Aufgabe erwiesen. Die Vektorgenerierung zeigte bereits im vorangegangenen Kapitel deutlich ihre Schwächen. Als erster Ansatz ist sie zumindest in Räumen mit vielen Wänden einigermaßen brauchbar.

Die größten Probleme bereitet die sinnvolle Navigation angesichts der Selbstbeschränkung, die Karte nicht weiter interpretieren zu wollen. Dies ist auf lange Sicht nicht hinnehmbar und unabdingbare Voraussetzung für einigermaßen intelligente Strategien. Ansonsten

[1] Die Software auf dem Roboters besteht aus über 30.000 Zeilen Sourcecode.

muß immer mit lokalen Daten gearbeitet werden, was bereits ein „vorausschauendes“ Ausweichen von Objekten unmöglich macht. Alle bisher erprobten Strategien sind und können somit auch nur bedingt erfolgreich sein. Deshalb wird eine Entwicklung komplexerer Strategien ohne Interpretation der Karte nicht zum Ziel intelligenten Navigierens führen können.
Gerade die Implementierung des Hauptrechners mit Navigator und Pilot, läßt auch jetzt schon Erweiterungen in dieser Richtung zu und erfüllt somit voll und ganz die insgesamt geforderte Flexibilität und Ausbaufähigkeit des Systems.

Ein wichtiger Aspekt ist die Tatsache, daß *keiner* der Rechner zur Zeit überlastet ist. Vor dem Hintergrund der Gesamtleistung des Systems ein recht erstaunliches Ergebnis. Hier wurde eine Überlastung oder zumindest adäquate Belastung viel früher erwartet. Die Idee und Realisierung als verteiltes System hat sich insofern als absolut richtige Entscheidung erwiesen und würde auch wieder gewählt.

Ausblick

In Zukunft soll das System einige Modifikationen erfahren, was als erstes die Motorsteuerung betrifft. Die viel zu schwachen Motoren sollen gegen stärkere und auch sparsamere Gleichstrommotoren mit Inkrementalgebern und Getriebe ausgetauscht werden. Zudem sollte bei einer Neuentwicklung der Module größeres Augenmerk auf die Problematik der EMV[2] gelegt werden.

Dem Roboter soll weitere Sensorik und die Ladestation zur Verfügung gestellt werden, die bereits entwickelt aber noch nicht realisiert wurden.
Eine interessante Sensorik stellen die neuen Computermäuse mit optischer Abtastung dar. Eventuell könnten sie eingesetzt werden, um Informationen über die tatsächliche Bewegung über Grund zu ermitteln.

Seitens der Software sind viele Modifikationen denkbar. So sind in einigen Algorithmen bewußt kleine Unzulänglichkeiten in Kauf genommen worden, um erste Ergebnisse zu ermöglichen. Diese sollten in Zukunft ausgemerzt und der unbedingt erforderliche Kompaß fertiggestellt werden.
Als wohl größte Erweiterung ist natürlich die Interpretation der Karte zu nennen, die das sinnvolle Navigieren unterstützt, beziehungsweise erst möglich macht. Vermutlich wird sich hier aber dieser Kartenrechner als zu leistungsschwach erweisen oder zumindest die Echtzeitfähigkeit einschränken.

Alles in allem aber ein sehr interessantes Projekt, das hinsichtlich der zur Verfügung stehenden Zeit und Mittel, gute Ergebnisse geliefert hat und sicher auch in Zukunft weiter verfolgt wird. Nicht zuletzt, weil die Ressourcen des Roboters zur Zeit noch nicht annähernd ausgereizt wurden.

[2] Elektro-Magnetische-Verträglichkeit

Abbildungsverzeichnis

Literaturverzeichnis

[1] Stefan Strich, Dirk Gehrmann : *Sonar für mobile Roboter*
Studienarbeit, 1998
FH-Hamburg, Fachbereich Elektrotechnik und Informatik

[2] Joseph L. Jones, Anita M. Flynn : *Mobile Roboter*
1. Auflage 1996
Addison Wesley
ISBN: 3-89319-8555-5

[3] T. Knieriemen : *Autonome Mobile Roboter*
Reihe Informatik Band 80
Bibliographisches Institut & F.A. Brockhaus AG
Mannheim 1991
ISBN: 3-411-15031-9

[4] Jens und Uwe Altenburg : *Mobile Roboter*
Hanser Verlag 1999
ISBN: 3-446-21121-7

[5] Serge Zakharian, Patricia Ladewig-Riebler, Stefan Thoer : *Neuronale Netze für Ingenieure*
Vieweg Verlag 1998
ISBN: 3-528-05578-2

[6] Markus Berg, Klaus-Werner Jörg und Jan-Peter Paulick : *3D Ultraschall-Entfernungsmessung mit Pseudo-Random Sequenzen*
AG Robotik & Prozessrechentechnik, Fachbereich Informatik Universität Kaiserslautern
Postfach 3049
67653 Kaiserslautern
email: {berg, joerg}@informatik.uni-kl.de
http://ag-vp-www.informatik.uni-kl.de

[7] Andrew S. Tanenbaum : *Moderne Betriebssysteme*

Carl Hanser Verlag
ISBN: 3-446-18402-3

[8] Buhrmann et.al.: *Physik für Ingenieure*
1. Auflage 1993
Harri Deutsch Verlag
ISBN: 3-8171-1242-4

[9] Heywang, Treiber et.al.: *Physik für Fachhochschulen und technische Berufe*
30. Auflage
Verlag: Verlag Handwerk und Technik
ISBN: 3-582-01113-5

[10] H. Stöcker : *Taschenbuch mathematischer Formeln und moderner Verfahren*
2. Auflage
Verlag Harri Deutsch
ISBN: 3-8171-1256-4

[11] U. Tietze, Ch. Schenk: *Halbleiter Schaltungstechnik*
9. Auflage
Axel Springer Verlag Berlin
ISBN: 3-540-19475-4

[12] H. Kopetz : *Design Principles for Distributed Embedded Applications*
Kluwer Academic Publishers, 1997

[13] Andrew S. Tanenbaum : *Computernetzwerke*
3. Auflage
Prentice Hall, 1998
ISBN: 3-8272-9568-8

[14] Robert Sedgewick : *Algorithmen in C++*
1. Auflage
Addison Wesley, 1992
ISBN: 3-89319-462-2

[15] Andreas Roth : *Das Mikrocontroller Kochbuch*
4. Auflage 1993
IWT Verlag GmbH
ISBN: 3-88322-225-9

[16] Felix Schörlein : *Mit Schrittmotoren steuern, regeln und antreiben*
2. Auflage 1996
Franzis Verlag
ISBN: 3-7723-6723-2

[17] Analog Devices : *Designer's Reference Manual*
Ausgabe von 1996

[18] Standard Microsystems Corporation (SMSC) : *COM20051*
Datenblatt des COM20051 Prozessors
Ausgabe von 1996

[19] Standard Microsystems Corporation (SMSC) : *COM20020*
Datenblatt des COM20020 ARCNET-Controllers
Ausgabe von 1998

[20] Philips : *General Magnetic field sensors*
Datenblatt
12. Juni 1998

[21] Philips : *Magnetic field sensor KMZ51*
Datenblatt
24. März 1998

[22] Conrad : *Datenblatt für LCD-Module*
Datenblatt
Conrad Elektronik GmbH
1998

[23] *Tausend Objektpunkte mit einem „Blick“*
Artikel in der Fachzeitschrift Elektronik
Ausgabe Nr. 15
25.Juli 2000
S.34