Inbetriebnahme eines Power-Cube-Roboterarmes; Entwicklung eines C167-Monitors und des PC-Gegenstücks zur Steuerung der Armgrundfunktionen

Diplomarbeit

Christian Asam

10. März 2003

Inhaltsverzeichnis

Zur Aufgabenstellung

Die ursprüngliche Aufgabenstellung lautete:

> Inbetriebnahme eines Power-Cube-Roboterarms, Entwicklung eines C167-Monitors und eines PC-Gegenstücks zur Steuerung der Armfunktionen

Der Roboter wurde 1996 angeschafft. Bis 2000 wurden lediglich im Rahmen einer Diplomarbeit und einer Studienarbeit einige Funktionen des Fahrwerks ausprobiert. Der Arm war bis 2000 nie Betrieb. Bis zum Beginn der Diplomarbeit hatten wir in den Roboter einen Rechner, ein Funknetz, jeweils eine C167-Mikrokontrollerkarte für die Ansteuerung des Arms und des Fahrwerks und einen Tiefentladeschatz für die Akkus eingebaut. Die große Latenzzeit zwischen Beschaffung und Inbetriebnahme des Roboters hatte jedoch zu einigen weiteren unvorhergesehenen ernsthaften Problemen geführt, die Herr Asam zum Teil im Rahmen seiner Diplomarbeit zusätzlich lösen musste. Teile der ursprünglichen Aufgabenstellung konnten aus technischen Gründen nicht zu Ende geführt werden:

- Zwei der Power-Cube-Module waren defekt. Die alte Elektronik und die alte Software (M3) wurden vom Hersteller nicht mehr aktiv unterstützt, so dass wir ein Elektronik-Up-Date für alle Module nehmen mussten. Das Elektronik-Update hatte die Ansteuerung der Module gegenüber dem Beginn der Diplomarbeit erheblich geändert. Das hatte gewichtigen Einfluss auf die zu entwickelnde Steuersoftware.
- Die serielle Schnittstelle auf dem Roboterrechner, die den Controller angesteuert hat, der wiederum den Arm angesteuert hat, ist gegen Jahresende ausgefallen. Als Work-Around sind wir auf eine kommerzielle PC-CAN-Buskarte umgestiegen und haben damit das Teilprojekt, die Bewegungsteuerung im Mikrokontroller zu kapseln, vorübergehend gestrichen. Als Ersatz wurde eine PC-basierte Ansteuerung vereinbart. Da das Teilprojekt mit dem C167-Monitor später wieder aufgenommen werden soll, soll der Arbeitsstand zum Zeitpunkt des Projektabbruchs dokumentiert werden.
- Trotz der eingebauten Dämpfer und der Verwendung einer Notebook-Platte ist die Festplatte des Roboterrechners ausgefallen. Dadurch musste der Roboterrechner im letzten Monat der Diplomphase auf ein RAM-Disc-System umgestellt werden.

Nach aktuellem Stand der Absprachen soll die Diplomarbeit von Herrn Asam folgende Teile enthalten:

- Eine einführende Beschreibung des Roboters, die für nachfolgende Mitarbeiter am Robotikprojekt als Wegweiser durch die verfügbaren Dokumentationen genutzt werden kann.
- Die aktuelle Hard- und Softwarekonfiguration des Roboterrechners
- Der aktuelle Arbeitsstand des C167-Monitors zum Zeitpunkt des Abbruchs dieses Teilprojekts
- Die entwickelten Softwarekomponenten für die PC-basierte Bewegungssteuerung.

G. Kemnitz

Verzeichnis der im Rahmen dieses Projektes erstellten Dokumentation

Zusätzlich zu dieser Arbeit wurden noch HTML-Dokumentationen erstellt, die auf dem Web-Server `http://techwww.in.tu-clausthal.de` im Dokumentatiosbereich oder auf der dieser Arbeit beiliegenden CD-Rom zu finden sind.

Anleitung für die manuelle Benutzung des Fahrwerks In [10] ist die essentielle Bedienungsanleitung zu finden, die man vor dem Fahren des Fahrwerks gelesen haben sollte um Unfälle zu vermeiden. Ebenso wird der Anschluß an das Ladegerät und die Rechnerstromversorgung beschrieben. Eine Kopie dieser Anleitung sollte auf dem Roboterrechner zu finden sein.

Beschreibung des Arms In [11] ist neben einer allgemeinen Beschreibung des PowerCube-Systems die bei uns vorhandene Konfiguration mit den hier verwendeten Modulparametern aufgeführt.

Beschreibung des Fahrwerks In [12] wird das Fahrwerk zusammen mit den daran vorgenommenen Änderungen beschrieben.

Beschreibung der Besonderheiten der Betriebssysteminstallation In [14] wird die Installation beschrieben und vor allem die Erstellung des RAM-Disk-Systems erläutert.

Beschreibung des Roboter-Webservers [18] beschreibt die Konfiguration des auf dem Roboterrechners laufenden Web-Servers, der für das Starten der Kontrollprozesse genutzt wird und das Steuerungsapplet liefert.

Beschreibung des Kommunikationsprotokolls der Steuerung [16] enthält die für das Verständnis und die Implementierung kompatibler Programmteile nötige Dokumentation des verwendeten Kommunikationsprotokolls.

Beschreibung der Softwarestruktur der Steuerung In [17] wird die allgemeine Struktur des Steuerungssystems beschrieben.

Erweiterungsanleitung für das Hauptkontrollprogramm Die Anleitung [15] gibt schrittweise die zur Erweiterung des Hauptkontrollprogramms um ein Modul nötigen Änderungen am System an.

Anleitung zur Erweiterung eines Moduls um neue Funktionen In [9] wird an einem Beispiel beschrieben, wie ein Modul um neue Funktionen erweitert werden kann.

Dokumentation des C167-Monitors Die Datei [13] enthält die Dokumentation zum C167-Microcontrollermonitorsystem.

1 Der Industrieroboter

In Abbildung 1.1 ist der Roboter zu sehen. Er kann in Arm und Fahrwerk unterteilt werden, die für sich gesehen einzelne Systeme bilden. Im Fahrwerk sind zur Energieversorgung zwei LKW-Akkus montiert, die für einige Stunden mobilen Betrieb ausreichen sollten. Danach sollte er, wie in der Betriebsanleitung [10] beschrieben, an die Ladestation angeschlossen werden. Die genannte Anleitung sollte unbedingt vor Benutzung des Roboters gelesen werden.

1.1 Arm

Der Roboterarm besteht aus Modulen der Firma Amtec, die Motor, Getriebe und dazugehörige Elektronik enthalten. Jedes Modul ist mit mindestens einem Microcontroller ausgestattet, der die über den CAN-Bus eintreffenden Befehle annimmt und umsetzt. Die Microcontroller der Module implementieren die Regler und erlauben eine flexible Nutzung. Bei Bedarf kann man den Bewegungsbereich der Module per Software einschränken. Für Bewegungen stehen drei Modi zur Verfügung:

- Zielposition: Hierbei wird langsam angefahren und kurz vor der Zielposition langsam abgebremst.
- Geschwindigkeit
- Strom: Bei diesem Modus wird der Motorstrom vorgegeben. Hiermit ist indirekt eine Art Kraftvorgabe möglich.

Aufbau des hier vorliegenden Armes

In Abbildung 1.2 ist unser Arm abgebildet. Dabei entsprechen die im Bild eingezeichneten Zahlen den in folgender Aufzählung angegebenen Modulen.

1. PR 90-PowerCube-Rotary-Module mit dem 160:1-Getriebe. Dieses Modul hat eine niedrige Maximalkraft, die Unfälle vermeiden helfen soll. Bei Überlast schaltet sich das Modul ab.
2. PR 90-PowerCube-Rotary-Module mit dem 160:1-Getriebe.
3. PR 70-Rotary-Module mit dem 160:1-Getriebe.
4. PR 70-Rotary-Module mit dem 160:1-Getriebe.

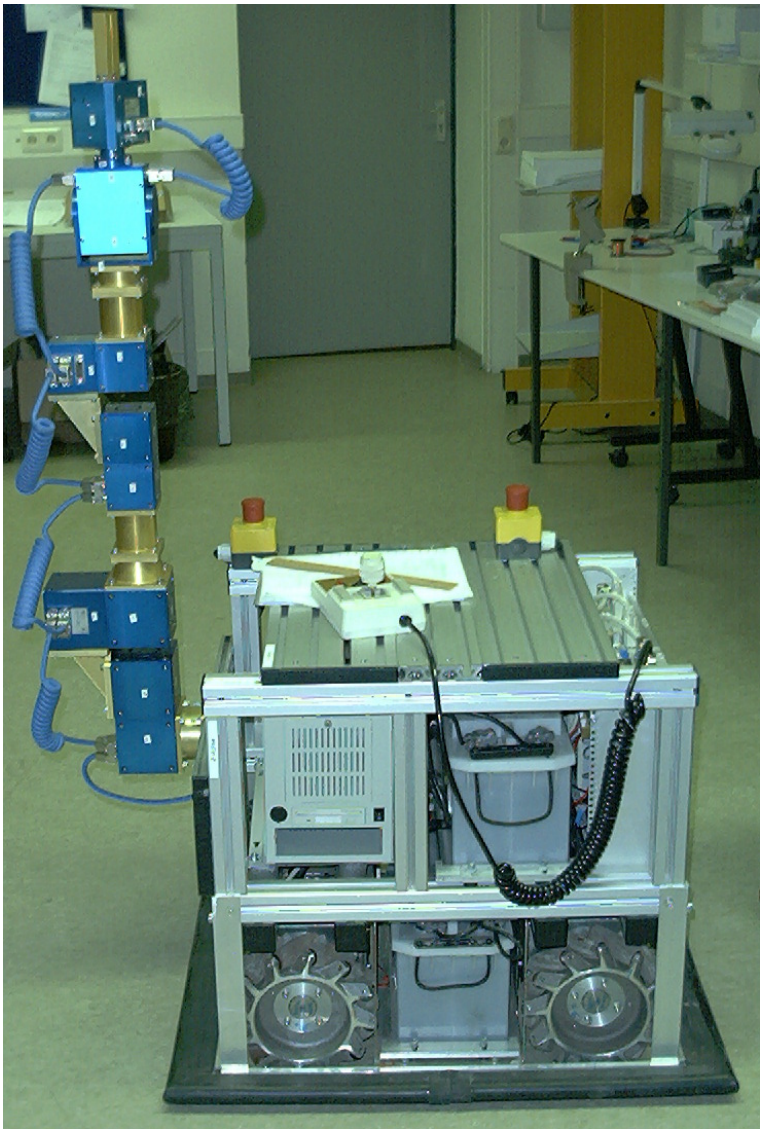

Abbildung 1.1: Der Roboter

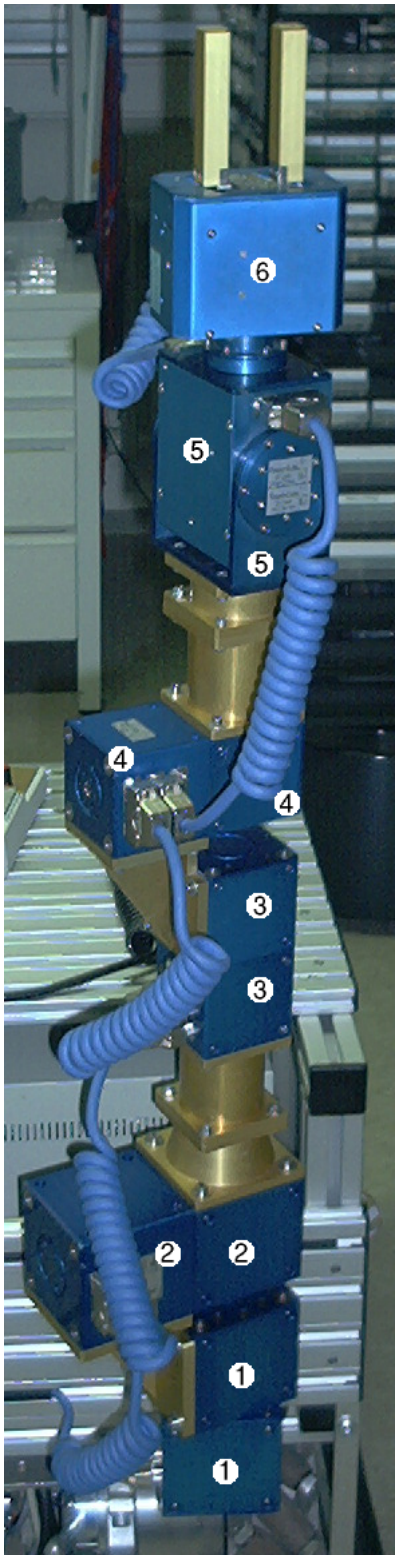

Abbildung 1.2: Aufbau des Arms

5. PW 70-PowerCube Wrist-Modul.

6. PG70-PowerCube Gripper, das zum Greifen von Objekten mit zwei ca. 9 cm langen Aluminiumfingern versehen wurde.

Die Numerierung der Module ist auf der Abbildung in “normaler” Zählweise aufgeführt und zählt reale Module. Die Steuersoftware zählt durchgängig ab 0 logische Module, daher ist für die PR-Module eine 1 abzuziehen und das PW-Modul erscheint als zwei Module. Genauere Angaben sind unter [11] zu finden.

Die PowerCube-Produktfamilie ist auf flexible Nutzung der Module ausgerichtet und daher können die Module vom Anwender zusammen mit Befestigungselementen nahezu beliebig zusammengesetzt werden. Bei unserem Arm haben wir uns für folgende, bei Bedarf änderbare, Konfiguration (von unten nach oben) entschieden:

- Rotation um die Z-Achse
- Rotation um die X-Achse
- Rotation um die Z-Achse
- Rotation um die X-Achse
- Rotation um die Y-Achse
- Rotation um die Z-Achse

Bei der aktuellen Anbringung kann der Arm einen Grossteil des Bereichs vor dem Fahrwerk und seine Ladefläche abdecken und begrenzt seitlich arbeiten und vermutlich eine Last von bis zu 5 kg heben, wobei wir dies noch nicht getestet haben.

1.2 Fahrwerk

Der von der Firma MIAG Fahrzeugbau gelieferte OMNIdrive Research Carrier besteht aus einem Rahmen aus Aluminiumprofilen, in dem die Steuerelektrik und -elektronik, die Energieversorgung und Platz für einen Rechner vorhanden sind. Der Rahmen ist auf einem Fahrwerk mit Mecanum-Antrieb, der beliebige Bewegungen in der Ebene erlaubt, montiert. Die Oberseite des Rahmens ist mit Aluminiumprofilplatten als Ladefläche abgedeckt und der Arm ist mit einer solchen Platte vor dem Rahmen angebracht.

Antriebssystem

Das bei unserem Roboter eingesetzte Mecanum-Antriebssystem besteht aus Rädern, auf denen diagonal angebracht Rollen montiert sind. Dies führt dazu, daß bei Bewegung des Rades eine Komponente in die Bewegungsrichtung ein eine zweite Komponente senkrecht dazu zur Seite auftritt. Ein Radpaar besitzt entgegengesetzt angebrachte Räder, so daß bei Vorwärtsbewegung beider Räder eines Paares sich die seitwärts gerichteten Kräfte aufheben und die Bewegung

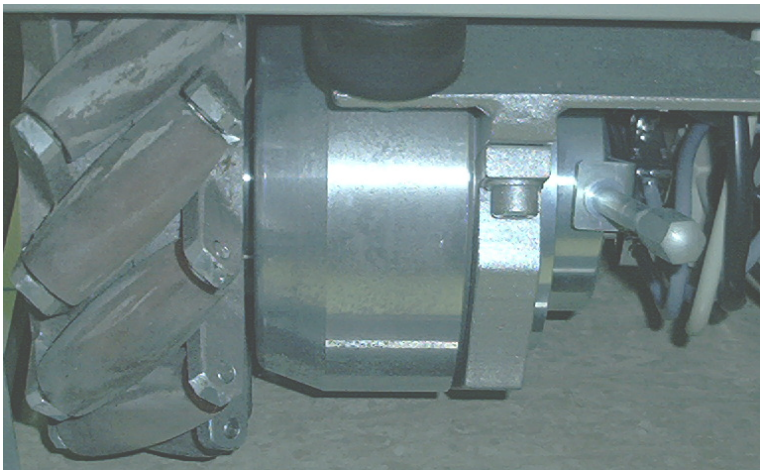

Abbildung 1.3: Mecanum-Rad

nur nach Vorne verläuft. Von diesen Paaren gibt es 2 Stück (und somit 4 Räder), womit alle Fahrtrichtungen möglich werden. Durch Überlagerung der verschiedenen Bewegungsrichtungen ist jede prinzipiell mögliche Fahrtrichtung fahrbar, ebenso Drehungen. Ich zitiere [24, S. 105]:

> Der OC-R stellt ähnlich der Ackermannlenkung eine vierrädrige Konfiguration dar, verwendet jedoch eine besondere Radausführung für den Antrieb, sogenannte MECANUM-Räder. Bei dieser Konstruktion sind auf dem Umfang aller vier Räder des Fahrzeuges Laufrollen aus Hartgummi in einem Winkel von 45° zur Laufrichtung angeordnet. Über separat steuerbare Synchronmotoren für jedes Fahrzeugrad ist eine Bewegung in jede beliebige Richtung in der Ebene möglich.

Zur Kinematik wird in [24, S. 106f] geschrieben:

> Analog zur Definition der Lage eines starren Körpers in der Ebene kann die Position eines mobilen Roboters in seiner Umgebung durch die zwei Ortskoordinaten eines Bezugspunktes auf dem Fahrzeug in einem feststehenden kartesischen Koordinatensystem (Weltkoordinaten) und einen Drehwinkel zur Angabe der Ausrichtung, also durch einen Vektor $\{x, y, \Phi\}$, beschrieben werden. Dieser Vektor wird als Konfiguration des Roboters bezeichnet, als Bezugspunkt dient im allgemeinen der geometrische Mittelpunkt der Grundfläche des Roboters.
>
> Eine Positionsveränderung entspricht der Ausführung sogenannter Grundbewegungen, d.h. Bewegungen entlang der Translationsachsen und um die Rotationsachse des Roboters. Omnidirektionale Fahrzeuge mit MECANUM-Rädern verfügen über Translationsachsen in Längs- und Querrichtung des Fahrzeuges. Die Hochachse durch den geometrischen Mittelpunkt des Fahrgestells stellt die Rotationsachse dar; das Fahrzeug verfügt über frei Freiheitsgrade in der Ebene. Eine Bewegung kann deshalb beschrieben werden durch einen Vektor $\{v_1, v_2, \psi\}$, wobei v_1 im folgenden

die Geschwindigkeit der Translation in Längsrichtung, v_2 die in Querrichtung und ψ die Winkelgeschwindigkeit der Rotation um die Hochachse angibt.

Auf den folgenden Seiten werden die Grundlagen des Mecanum-Systems genauer hergeleitet, ich werde hier nur das für den Betrieb relevante Ergebnis angeben:

$$\begin{Bmatrix} \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \end{Bmatrix} = \begin{bmatrix} 1 & -1 & -(d+s) \\ 1 & 1 & (d+s) \\ 1 & 1 & -(d+s) \\ 1 & -1 & (d+s) \end{bmatrix} \begin{Bmatrix} v_1 \\ v_2 \\ \psi \end{Bmatrix}$$

Variable	Bedeutung
v_1	Vorgabe in Längsrichtung
$v2$	Vorgabe in Querrichtung
ψ	Vorgabe Winkelgeschwindigkeit um Rotationsachse
θ_n	Rotationsgeschwindigkeit des n-ten Rades
d	Abstand des Rades vom Bezugspunkt in Längsrichtung
s	Abstand des Rades vom Bezugspunkt in Querrichtung

1.3 Elektrik und Elektronik

Energieversorgung

Im Fahrwerk des Roboters stellen zwei LKW-Akkus mit je 12V 130Ah die Energieversorgung sicher. Bei diesen Bleiakkus sorgt die Auslegung der Akkus auf den Fahrzeugbetrieb dafür, daß zum einen die Erschütterungen und Bewegungen im Fahrbetrieb kein Problem für die Akkus darstellt und zum anderen auch extrem hohe Spitzenströme entnommen werden können. Kurzzeitig sind Entladeströme von über 500 A möglich, auch wenn dadurch die entnehmbare Energie deutlich geringer als die Nennkapazität ist. Aufgrund des möglichen hohen Kurzschlußstroms ist eine gute Absicherung unabdingbar, um bei einem Kurzschluß den Abbrand des Roboters zu verhindern. Diese Absicherung wurde von der Herstellerfirma durchgeführt, woran wir nichts geändert haben, die aber bei der Nutzung zu berücksichtigen ist. Das Fahrwerk des Roboters hat eine zulässige Dauerleistungsaufnahme von ca. 500 W und eine Spitzenleistungsaufnahme von 2400 W [19, S. 15], der Arm nach [5] eine zulässige Dauerleistungsaufnahme von 960 W und eine Maximalaufnahme von 2880 W womit sich schon abzeichnet, daß die Stromversorgung nicht ganz unproblematisch ist. Im Schaltplan [20, S. 1] ist für den Stellerkreis, an dem mittlerweile auch der Arm angeschlossen ist, eine Absicherung von 60 A angegeben. Damit ergibt sich eine maximale Dauerlast von 1440 W. Sicherungen sind primär für den Schutz der Versorgungsleitungen gegen Überhitzung gedacht, daher ist eine kurzzeitige Überlastung durchaus zulässig, z.B. durch Einschaltströme oder Anfahrströme. Die Absicherung des Kreises erlaubt somit den Betrieb des Fahrwerks ohne Probleme und des Arms unter der Voraussetzung, daß nicht alle Module gleichzeitig mit Maximallast arbeiten. Als Einschränkung ist zu berücksichtigen, daß der gleichzeitige Hochlastbetrieb des Fahrwerks und des Arms nicht möglich ist.

Anpassungen der vom Hersteller gelieferten Elektrik

Zusätzlich zu dem von MIAG gelieferten Fahrwerk, welches für sich schon funktionsfähig ist, mußten noch der Arm und der Roboterrechner versorgt werden. Weiterhin sind im Laufe des Projektes noch einige Probleme aufgefallen, die Änderungen an der Elektrik erforderlich werden liessen:

- Der Arm musste angeschlossen werden
- Der Rechner musste auch bei ausgeschaltetem Fahrwerk versorgt werden
- Ein Schutz der Akkus gegen Tiefentladung (durch den Rechner) war nötig

Anschluß des Armes

Der Arm wurde parallel zu den Stellern hinter den Hauptschütz angeschlossen. Hierdurch wird er automatisch von der Sicherheitsinfrastruktur der Fahrwerks mitgeschaltet und ausreichend versorgt.

Rechnerversorgung und Tiefentladeschutz

In der ursprünglichen Planung des Systems war schon ein Rechner vorhanden, der allerdings ausschließlich über den per Schlüsselschalter einschaltbaren Hauptkreis der Fahrwerkselektrik versorgt wurde. Hierdurch ergab sich aber auch, daß das Fahrwerk bei Rechnerbetrieb (also auch bei reinen Programmiertätigkeiten) aktiv sein mußte und aus der Fahrzeugbatterie gespeist wurde. Das mit dem Fahrwerk gekaufte Ladegerät eignet sich nicht als Netzteil für den Dauerbetrieb des Rechners, da es nach erfolgter Ladung abschaltet und der Rechner dann nur noch von den Akkus gespeist wird. Der Verbrauch des Rechners liegt bei ca. 1.5 A, was aber rechnerisch nach weniger als 4 Tagen zur totalen Entladung der Akkus führt. Zusammen mit der Eigenschaft von Blei-Akkus, nach Tiefentladung nicht mehr ladbar zu sein, ist das Risiko für die Akkus zu groß um nur durch sorgfältige Benutzung den Akkutod zu verhindern.

Als Abhilfe wurde eine zusätzliche Versorgungsbuchse für den Rechner angebracht, über die er von einem Labornetzteil versorgt wird und damit unabhängig von den Akkus betrieben werden kann. Weiterhin wurde eine Schaltung eingebaut, die bei Unterschreiten einer Minimalspannung die Schütze des Fahrwerks abschaltet und damit alle Verbraucher abschaltet. Der aktualisierte Schaltplan ist unter [3] zu finden, der Schaltplan der Tiefentladeschutzschaltung unter [4]. Um die Spannung der Akkus zu überprüfen, wurde ein Digital-Einbauinstrument installiert[6].

Sicherheitssystem

Der Roboter ist mit folgenden Sicherheitseinrichtungen ausgerüstet:

- Not-Aus-Kreis mit zwei Notschaltern
- 2 cm über dem Boden angebrachte Bumperleiste, die das Fahrwerk umgibt

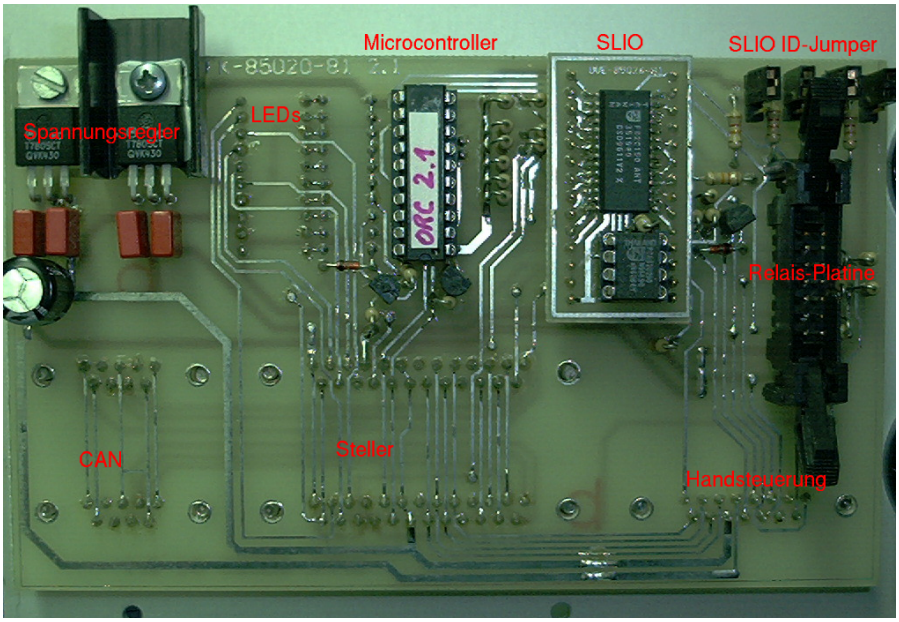

Abbildung 1.4: Platine in der Bedienkonsole

Bei einer Kollision mit einem Hindernis auf Bodenhöhe schaltet sich das Fahrwerk und damit auch der Arm ab. Diese Abschaltung ist manuell überbrückbar, um das Fahrwerk von einem Hindernis wegfahren zu können. Die Notschalter sind nicht überbrückbar.

Für den Arm existiert zur Zeit keine eigene Notabschaltung, die bei Kollisionen auslöst. Die PowerCube-Module haben aber selbst eine Notabschaltung bei zu hoher Kraft, so daß Schäden begrenzt sein dürften. Sollte der Arm einen der Not-Aus-Schalter auslösen, so hat man keine einfachere Möglichkeit ihn wieder vom Schalter wegzubewegen als entweder den Arm oder den Not-Aus-Schalter abzumontieren.

Steller

Der Mecanumantrieb wird von zwei Stellern versorgt, die jeweils für eine Achse zuständig sind. Die Steller sind auf der rechten Seite des Bedienpultes angebracht und werden vom Bedienpult aus gesteuert. Die Steller können entweder analog über den Joystick oder digital über eine serielle Schnittstelle (RS232/5V oder CAN) angesteuert werden. Die Steller speisen beim Bremsen freiwerdende Energie in den Batteriezwischenkreis ein[19, S. 1]. Die Beschreibung der Steller ist unter [19] zu finden.

Steuerungselektronik in der Bedienkonsole

Die Bedienkonsole (links) enthält neben dem Energieversorgungsteil noch die in Abbildung 1.4 abgebildete Platine, an die die Handsteuerung, die CAN-Anschlüsse und die Steller angeschlossen sind. Leider fehlt die genaue Dokumentation dieser Platine, einige grundsätzliche Eigen-

schaften werde ich aber hier beschreiben. Als Kernfunktion scheint die Platine die Verteilung der Signale von der Handsteuerung und dem CAN-Bus zu den Stellern und zur Relaisplatine zu haben. Daneben beherbergt sie noch einen Microcontroller und einen SLIO-Baustein, der prinzipiell nicht mehr als ein per CAN steuerbarer IO-Baustein ist. Da die genaue Funktion des Microcontrollers und die Verdrahtung der Platine mir nicht bekannt ist (entweder wurde keine genaue Dokumentation geliefert oder sie ist verschwunden) können Teilfunktionen des Systems u.U. nicht genutzt werden. In [24, S. 118-119] sind sie Dateneingangssignale[1] des SLIO angegeben:

Bit	Data Input	Bedeutung
15	/Autofahrt	Fahren über den CAN-Bus ist aktiviert
14	Autoladen	Laden per CAN-Bus ist aktiviert
13	/Steller ein	Steller werden mit Spannung versorgt
12	Hand	Schalter Hand/Auto auf Handbetrieb
11	Handfreigabe	Taster Handfreigabe betätigt
10	/Bumper	Schaltleiste wurde betätigt
9	/Notaus	Notausschalter wurde betätigt
8	Auto	Automatikbetrieb freigegeben
7	Bereit_hinten	Steller hinten meldet bereit
6	Bereit_vorne	Steller vorne meldet bereit
5	Freigabe	Freigabe Stellersollwerte
4	Laden	Laden per Handschalter ist aktiviert

Die Datenausgangssignale des SLIO sind:

Bit	Data Output	Bedeutung
15	Freigabe	fahren über CAN-Bus freigegeben
14	Laden	Laden aktivieren

Ein / vor dem Namen eines Signals gibt an, daß das Signal Low-Active ist, ein 0-Pegel also die Aktivität angibt. Für die Daten dieser Tabelle wurde keine Quelle angegeben, daher bleiben einige Unklarheiten bestehen, die vermutlich erst durch Beschaffung weiterer Dokumentation vom Hersteller zu klären sind. Im Zusammenhang mit dem Schaltplan des Fahrwerks [20] macht es keinen Sinn, das “Laden” als das Laden der Akkus zu interpretieren. Daher vermute ich, daß mit dem “Laden” das Laden von Parametern in die Steller gemeint ist, wofür leider keine genaue Dokumentation vorliegt.

[1] In der zitierten Arbeit sind beide hier zitierten Tabellen in einer einzigen Tabelle eingetragen. Da bis auf die ersten beiden Zeilen alle Einträge der letzten beiden Spalten n.b. sind, habe ich die beiden nutzbaren Einträge in eine eigene Tabelle eingetragen.

1.4 CAN-Bus

Der CAN-Bus besitzt, laut dem Entwickler des Standards, folgende Eigenschaften (Zitat von [1]):

- [It] is a high-integrity serial data communications bus for real-time applications
- Operates at data rates of up to 1 Megabits per second
- Has excellent error detection and confinement capabilities
- Was originally developed by Bosch for use in cars
- Is now being used in many other industrial automation and control applications
- Is an international standard: ISO 11898

Die genaueren Eigenschaften werden in [2] erläutert:

> What is CAN ?
>
> CAN is a serial bus system especially suited for networking "intelligent" devices as well as sensors and actuators within a system or subsystem.
>
> The Attributes of CAN
>
> CAN is a serial bus system with multi-master capabilities, that is, all CAN nodes are able to transmit data and several CAN nodes can request the bus simultaneously. The serial bus system with real-time capabilities is the subject of the ISO 11898 international standard and covers the lowest two layers of the ISO/OSI reference model. In CAN networks there is no addressing of subscribers or stations in the conventional sense, but instead, prioritized messages are transmitted.
>
> A transmitter sends a message to all CAN nodes (broadcasting). Each node decides on the basis of the identifier received whether it should process the message or not. The identifier also determines the priority that the message enjoys in competition for bus access.
>
> [...]
>
> One of the outstanding features of the CAN protocol is its high transmission reliability. The CAN controller registers a stations error and evaluates it statistically in order to take appropriate measures. These may extend to disconnecting the CAN node producing the errors.
>
> Each CAN message can transmit from 0 to 8 bytes of user information. Of course, you can transmit longer data information by using segmentation. The maximum transmission rate is specified as 1 Mbit/s. This value applies to networks up to 40 m. For longer distances the data rate must be reduced: for distances up to 500 m a speed of 125 kbit/s is possible, and for transmissions up to 1 km a data rate of 50 kbit/s is permitted.

Ich fasse zusammen:

- Serieller Bus
- Echtzeittauglich durch priorisierte Nachrichten
- Datenrate von 50 kBit/s bis 1 MBit/s
- Gute Fehlerbehandlungsmöglichkeiten
- Für den Einsatz in Fahrzeugen entwickelt und mittlerweile in Industrieanwendungen verbreitet
- Broadcast-Bus, die Empfänger entscheiden anhand des Identifiers einer Nachricht, ob sie sich angesprochen fühlen sollen
- Identifier legt Priorität der Nachricht fest
- Maximal 8 Byte pro Nachricht übertragbar, wenn mehr Daten übertragen werden sollen muß segmentiert werden.

Kollisionen erfolgen zerstörungsfrei, d.h. wenn zwei Stationen gleichzeitig senden wollen wird über den Identifier festgelegt, welche “ihre” Nachricht zuerst senden darf. Im CAN-Bus gibt es durch die Eigenschaften des Mediums dominante und rezessive Zustände, wenn gleichzeitig ein dominanter und ein rezessiver Zustand gesendet wird, setzt sich der dominante Zustand durch. Wenn beim Senden eine Station einen rezessiven Zustand sendet, auf dem Bus aber ein dominanter Zustand vorliegt, beendet die Station das Senden dieser Nachricht und wartet darauf, daß die Station, die gleichzeitig die höherpriorisierte Nachricht zu senden begonnen hat, fertig ist. Danach wird erneut versucht zu senden. In [23, 39] ist dieses beschrieben. [23] enthält den Standard und beschreibt Teil A und B zusammen, in [7] ist Teil A und in [8] Teil B zu finden.

Nach [8, S. 12f] besteht ein Data-Frame (das sind die für uns interessanten Frames) aus folgenden Teilen:

- Start of Frame
- Arbitration Field
- Control Field
- Data Field
- CRC Field
- ACK Field
- End of Frame

Start of Frame besteht aus einem einzelnen dominanten Bit und markiert den Anfang eines Frames. Danach folgt das Arbitration Field, in dem der Nachrichten-Identifier enthalten ist. Durch die frühe Position im Frame kann die Arbitration erfolgen. An dieser Stelle unterscheiden sich auch Standard-Format und Extended-Format, das Standard-Format hat 11 Bits als Identifier, das Extended-Format 29. Auf die genauen Unterschiede gehe ich hier nicht ein, sie sind in [8, S. 12f] nachzulesen. Unsere Systeme arbeiten alle mit dem Standard-Format.

Im Control Field ist der für uns relevante Teil der Data Length Code (DLC), der die Anzahl der übertragenen Bytes angibt. Da 0 bis 8 Bytes übertragen werden können ist der DLC 4 Bit lang.

Im Data Field werden die Datenbytes übertragen, sofern überhaupt welche übertragen werden sollen.

Nach dem Data Field wird noch eine Prüfsumme mitgesandt und dahinter folgt das Acknowledge-Field, in dem alle Stationen, die eine korrekte Prüfsumme für das gerade vorbeikommende Paket berechnet haben das vom Sender gesendete, rezessive Bit ACK-Slot mit einem dominanten Bit überschreiben.

1.4.1 CAN-Bus-Nachrichtenformat der PowerCube-Module

Das PowerCube-CAN-Protokoll trennt den CAN-Nachrichten-ID-Raum in einen Teil für Befehls-Identifier und Moduladressen auf. Die Identifier sind zusammen mit den Moduladressen in die CAN-Nachrichten-ID codiert. Der Identifier MODULEACK hat beispielsweise die Nummer 0x20, wenn man diesen Identifier an ein Modul mit der Modulnummer 3 senden wollte, würde man die Nachrichten-ID 0x23 wählen. Der nächste Identifier ist MODULEREQ mit der Nummer 0x40. Durch den “Abstand” der Befehle ist der Adressraum der Modul-IDs vorgegeben, womit sich 5 Bits für die Moduladressen und 6 Bits für die Identifier ergeben. Um einen Befehl an ein Modul zu senden, wird als CAN-Nachrichten-ID PUT (0xe0)+Modul-ID gewählt und die Command-ID im ersten Datenbyte der Nachricht übertragen. Parameter zum Befehl folgen, sofern es Parameter gibt, als weitere Datenbytes.

Bei Morse3 waren die meisten Befehle als Command-ID im ersten Datenbyte codiert, um die Kompatibilität zu Morse3 bei Modulen mit dem Morse5-Befehlssatz zu wahren, wurden beim Morse5-Protokoll die meisten Befehle als “erweiterte Befehle” hinter die Command-ID SETEXTENDED (0x08) “versteckt”: Als nächsten Parameter kommt der Morse5-Befehl und dahinter dann die Daten.

Befehle

In diesem Abschnitt führe ich einige Befehle auf, die die PowerCube-Module verstehen. Dabei werde ich nicht jeden Befehl aufführen, sondern nur die wichtigsten, die auch im Projekt Verwendung finden. Die detaillierte Liste der Befehle ist ist im Kapitel 7 von [5] zu finden.

Reset

CAN-ID	CMD_PUT+Modul-ID
Command-ID	Reset

Reset löscht den Fehlerstatus eines Moduls. Dies ist erforderlich, um ein Modul nach z.B. einer Überkraftabschaltung wieder zur Mitarbeit zu bewegen.

Home

CAN-ID	CMD_PUT+Modul-ID
Command-ID	Home

Nach dem Einschalten der Module ist eine Referenzpunktfahrt erforderlich damit der Microcontroller im PowerCube die wirkliche Position bestimmen kann und damit den erlaubten Bewegungsbereich überwachen kann.

Bewegungsbefehle

CAN-ID	CMD_PUT+Modul-ID
Command-ID	SetMotion
Parameter-ID	Fxxx_MODE
Parameter 1	Float Byte 1 von 4
Parameter 2	Float Byte 2 von 4
Parameter 3	Float Byte 3 von 4
Parameter 4	Float Byte 4 von 4

Bewegungsbefehle werden in die Command-ID SET_MOTION eingebettet. Als erster Parameter folgt die Bewegungsart, danach die Parameter wie Zielposition. Es gibt Befehle für die Vorgabe von Zielposition, Zielgeschwindigkeit und Strom, weiterhin gibt es Befehle, um nach Möglichkeit einen Zielort in vorgegebener Zeit zu erreichen und die nötige Geschwindigkeit und Beschleunigung vom Modul zu errechnen. Morse3 konnte nur mit Inkrementalgeberwerten arbeiten, bei Morse5 wird überwiegend auf Basis von SI-Einheiten mit Parametern im Float-Format gearbeitet. Bewegungen, die eine Zielposition anfahren, werden bei Morse5 grundsätzlich im Ramp-Modus durchgeführt, der Pos-Modus von Morse3 ist weggefallen. Der Pos-Modus von Morse3 war etwas naiv, das Modul wurde mit der angegebenen Geschwindigkeit bis zum Endpunkt gefahren und dann abgebremst. Dabei wurde allerdings nicht rechtzeitig gebremst und bei entsprechender Geschwindigkeit schwang das Modul über, was die Lageregelung bemerkte und korrigierte. Diese Korrektur war entsprechend überschwingend und es konnte einige Minuten dauern, bis sich der Arm beruhigt hat. Beim Ramp-Modus wird langsam beschleunigt, bis das Modul auf Nenngeschwindigkeit angekommen ist und rechtzeitig vor Erreichen der Solllage langsam abgebremst, so daß ein Überschwingen nicht vorkommt.

Parameterbefehle

CAN-ID	CMD_PUT+Modul-ID
Command-ID	SetExtended
Parameter-ID	Parameter
Parameter 1	Daten
Parameter 2	Daten
Parameter 3	Daten
Parameter 4	Daten
Parameter 5	Daten
Parameter 6	Daten

Als letzte Befehlsklasse gibt es die Parameterbefehle, die mit den Nachrichten-IDs GET_EXTENDED und SET_EXTENDED gelesen und geschrieben werden können. Im Kapitel 7.2.7 von Handbuch.pdf werden die einzelnen Parameter erläutert, grob gesagt sind alle sinnvollen Parameter damit zugänglich. Die Parameter umfassen mechanische Eigenschaften der Module, Default-Werte, Geschwindigkeits- und Beschleunigungsparameter, Strom, aktuelle Position, Digital-IO-Zustand und sogar die Reglerparameter.

Beispiel Als Beispiel wird dem Modul 0x11 der Bewegungsbefehl mit der Zielposition 0.75 aus [5, S. 105] angegeben:

CAN-ID	0xF1	0xe0+0x11
Command-ID	0x0b	SetMotion
Parameter-ID	0x04	FRAMPMODE
Parameter 1	0x00	Pos
Parameter 2	0x00	Pos
Parameter 3	0x40	Pos
Parameter 4	0x3f	Pos

Beispiel für eine einfache Befehlsfolge, um den Arm zu bewegen

- Reset
- Home
- Move

1.4.2 CAN-Ansteuerung des Fahrwerks

Hierzu liegen uns sehr wenig Informationen vor, vor allem fehlt eine zuverlässige Programmieranleitung. Die vorliegenden Quellen sind [19] und [24].

Um das Fahrwerk in Betrieb zu nehmen muß über den SLIO den Stellern die Freigabe erteilt werden. Dazu muß dieser entsprechend des im Datenblatt [21] vorgegebenen Verfahrens kalibriert werden. Danach sind die Ausgänge (siehe [24, S. 118-119]) zu konfigurieren und mit den entsprechenden Werten zu versehen.

Zur Programmierung der Steller sind die genannten Quellen sehr dürftig, daher kann ich hier kein funktionierendes Verfahren angeben, mit dem die Steller angesprochen werden können. Prinzipiell scheinen in der CAN-ID die ersten 6 Bits für die Auswahl einer programmierbaren Funktion zuständig zu sein, das nächste Bit hat die Bedeutung Set/Request und die unteren 4 Bits sind für die Motornummer bzw. Motorgruppe zuständig.

Zur Programmierung der programmierbaren Funktionen über den CAN-Bus bzw. der bereits programmierten Funktionen steht so wenig in der Dokumentation, daß die Steller nur nach Beschaffung genauerer Dokumentation nutzbar sein dürften.

2 Roboter-Rechner

2.1 Hardware

Zur Nutzung als autonomes System ist noch ein Steuerrechner mit ausreichender Flexibilität und Rechenleistung nötig, der die Subsysteme steuert und Erweiterungen (z.B. für die Bilderkennung) aufnehmen kann. Aus Kostengründen bot sich eine PC-basierte Lösung an.

Durch die Vorgaben des Fahrwerks konnte kein normaler PC eingesetzt werden - dafür war kein Platz vorhanden. Daher musste eine Gehäuseform gefunden werden, die am vorgegebenen Platz im Fahrwerk montiert werden konnte. In diesem Gehäuse ist ein 24V-AT-Netzteil, ein 5,25”-Schacht, ein 3,5”-Schacht (Diskettenlaufwerk) und auf einer passiven PCI/ISA-Backplane ein Einplatinen-PC untergebracht. Die ursprünglich im 5,25”-Schacht montierten C167-Controllerbaugruppen wurden entfernt, da sie nicht mehr verwendet werden. Sie stehen jetzt anderen Projekten zur Verfügung und können bei Bedarf wieder montiert werden. Als Ersatz für die C167-Controller wurde eine CAN-PCI-Karte angeschafft, die einen normalen PCI-Platz beansprucht. Ein weiterer PCI-Slot wird von einer Wireless-LAN-Karte (Abbildung 2.2) in einem PCI-Adapter gefüllt. Als Festplatte wurde eine Notebook-Festplatte gewählt (vor allem wegen der mechanischen Unempfindlichkeit), die schwingungsgedämpft montiert ist. Der gesamte Rechner ist auf Silikondämpfern montiert und dadurch unempfindlich gegen die Stöße, die das Fahrwerk während der Bewegung erzeugt. Die Festplatte ist trotz der Dämpfungsmassnahmen nach einigen Monaten ausgefallen.

Auf der linken Seite der Abbildung 2.1 ist die Antenne der WLAN-Karte zu sehen, rechts daneben die beiden CAN-Anschlüsse (der obere ist Bus 1 für das Fahrwerk, der untere Bus 0 für den Arm), dann folgen zwei freie Slots und daneben ist die Karte, auf der der Rechner sitzt. Diese trägt einen Pentium III-Prozessor, 512MByte SD-RAM, einen Chipsatz mit integrierter Graphik und den üblichen PC-Komponenten.

Auf der CPU-Karte (Abbildung 2.3) im PICMG-Format sind von links nach rechts die CPU, RAM-Module, Chipsatz mit Graphik, BIOS, Disk-on-Chip2000-Sockel und diverse ICs (Watchdog, Ethernet usw.) zu finden. Diese Karte steckt in der abgebildeten passiven Backplane (Abbildung 2.4), die nicht viel mehr als die Verbindungen zwischen den Slots und die Anschlüsse für die Stromversorgung enthält.

Die CAN-Karte (Abbildung 2.5) enthält eine PCI-Microcontrollerbus-Bridge (nahezu PCI-ISA), einige MByte SRAM, einen 68331-Microcontroller und zwei CAN-Interfaces.

Im Bild 2.6 ist der Rechner mit eingesteckten Karten abgebildet. Das linke Flachkabel ist der Floppy-Anschluß, das rechte Kabel (das etwas frei rumhängt) ist der Anschluß des CD-Laufwerkes. Unten links auf dem Bild ist der Ort zu sehen, an dem die Festplatte angebracht war, bevor sie ausgefallen ist. Durch den kompakten Aufbau bedingt ist es ziemlich eng im Gehäuse.

Abbildung 2.1: Rückseite des Roboterrechners mit Beschriftung

Abbildung 2.2: WLAN-Karte

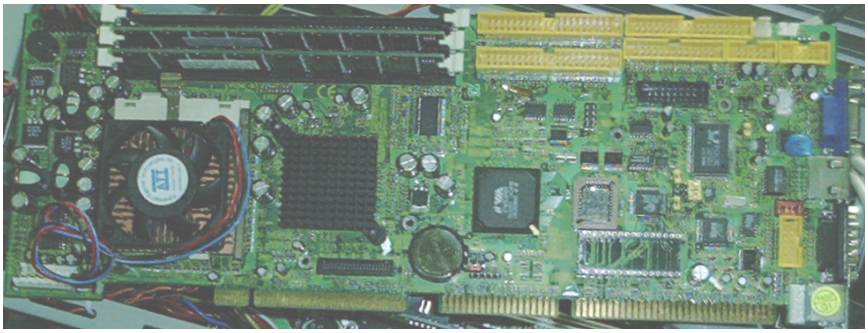

Abbildung 2.3: CPU-Karte

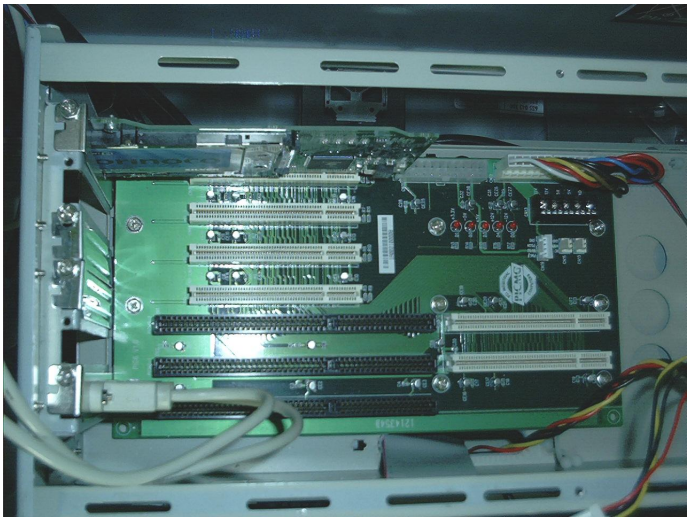

Abbildung 2.4: Backplane mit eingesteckter WLAN-Karte

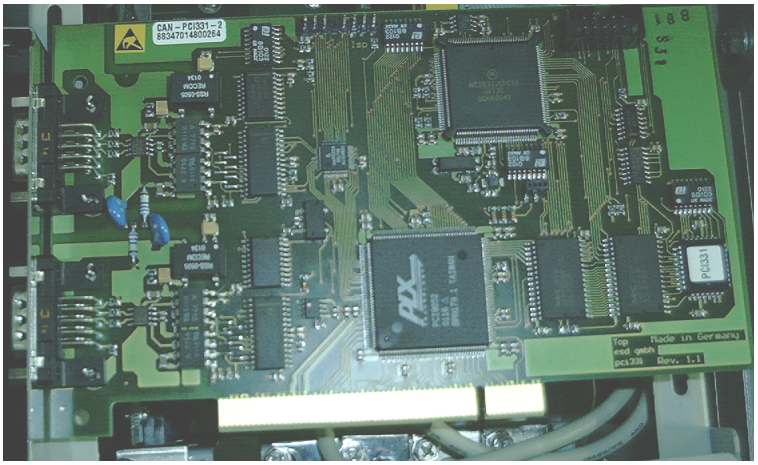

Abbildung 2.5: PCI-CAN-Karte

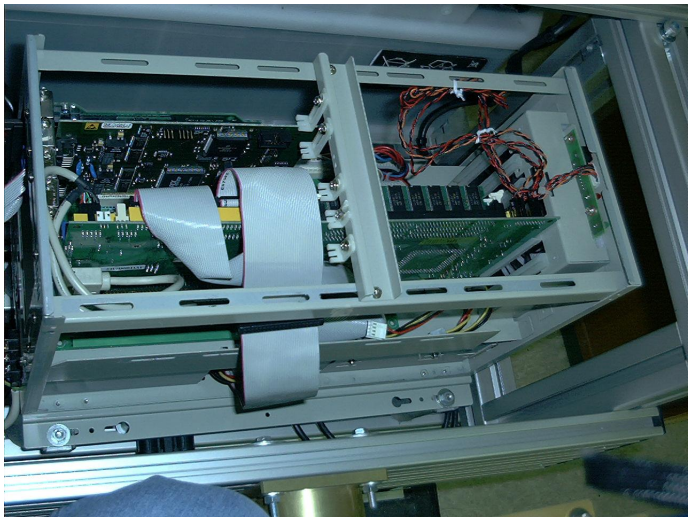

Abbildung 2.6: Rechner mit Karten

Die Kühlung ist aber durch einen großen Lüfter auf der Rechnervorderseite gewährleistet, zumal der Rechner typischerweise nur ca. 35-40W insgesamt aufnimmt.

2.2 Betriebssystem

Durch die mobile, autonome Arbeitsweise ist es unpraktikabel, einen Monitor und Tastatur am Roboter anzubringen. Es ist bedeutend günstiger, das System fernzusteuern, wozu ein handelsübliches Funknetzwerk eingesetzt wird. Aufgrund der Headless-Arbeitsweise bietet sich ein UNIX-artiges System an, da man dort die komplette Fernsteuerbarkeit ohne Zusätze im Betriebssystem verankert hat. In diesem Fall ist das System ein normales Linux-System.

Darauf aufsetzend werden die Softwaremodule so implementiert, daß Standardprotokolle genutzt werden. Es ist vorgesehen, zur Steuerung ausserhalb des Labors beispielsweise ein beliebiges Notebook mit einem Java-fähigen Webbrowser und WLAN-Karte zu verwenden.

Für die Entwicklung ist ein weiterer Linux-Desktop-PC vorgesehen.

Betriebssystem des Roboterrechners

Auf dem Roboterrechner wurde ein einigermaßen normales Debian Gnu/Linux-System installiert. Für die Debian-Distribution spricht die gute Anpassbarkeit und die Möglichkeit, eine kleine Installation durchzuführen. Da der Rechner für den Normalbetrieb keinen Monitor und keine Eingabegeräte erhält, machen die graphischen Administrationswerkzeuge der meisten anderen Distributionen keinen Sinn und die rein textbasierte Administration des Debian-Systems durch Editieren der ASCII-Konfigurationsdateien spart auch die nicht wirklich große Bandbreite des WLANs und ist auch für ein RAM-Disk-Image möglich.

Ursprünglich wurde erwogen, das System aus einer Ramdisk laufen zu lassen und die Festplatte abzuschalten. Dies schien aber durch die ausreichende Dämpfung des Rechners nicht erforderlich zu sein, hat sich aber nach dem Ausfall der Festplatte als erforderlich herausgestellt. Im folgenden Kapitel wird der RAM-Disk-Betrieb erläutert.

Geblieben sind als “Probleme” das Fehlen von Monitor/Tastatur und festem Netzwerkanschluss. Der Rechner musste also so konfiguriert sein, dass er beim Starten automatisch sein Funknetz startet und darüber genutzt werden kann. Die Konfiguration des Funknetztes geschieht an zwei Stellen: Zuerst müssen die Treiber verfügbar sein (bei der Kernelkompilation die passenden Treiber einbinden) und dann muß das Netz richtig konfiguriert werden. Die Funknetzkarte ist eine PCMCIA-Karte, die in einem PCMCIA-PCI-Adapter steckt. Für das Betriebssystem entspricht dies einer Notebookkonfiguration, was zur Folge hat, daß die Treiber der Funknetzkarte erst nach dem Booten des Kernels geladen werden könnnen, also von den Hotplug-Scripten des Systems eingebunden werden. Die Konfigurationsdaten liegen in /etc/pcmcia, relevant sind hier besonders wireless.opts.

Die WLAN-Karte wird vom System als eth1 geführt und benimmt sich wie ein normales Ethernet-Netzwerkinterface.

Besondere Beachtung ist auch den Home-Verzeichnissen zu schenken: Für die Entwicklung ist die Nutzung von per NFS vom Entwicklungsrechner zur Verfügung gestellten Home-Verzeichnissen praktisch, allerdings funktioniert dies nur, solange der Rechner in Reichweite des Funknetzes ist. Sobald der Empfang gestört ist ist mit Problemen zu rechnen, daher bestehen zwei Lösungsansätze:

Beim ersten Ansatz liegt die Robotersoftware lokal auf dem Roboterrechner und das Home-Verzeichnis wird per NFS vom Server bezogen. Dies ist von Systemseite das Einfachste, aber für den Benutzer das umständlichste Vorgehen.

Alternativ kann die automatische Replikation des Entwicklungsrechner-Homes auf den Roboterrechner durch ein verteiltes Dateisystem genutzt werden, wobei die Roboter-Software aus dem Home-Verzeichnis des Entwicklers gestartet würde. Erfolgversprechend scheint das CODA-Dateisystem zu sein, das als verteiltes Dateisystem konzipiert ist und prinzipiell den Anforderungen entspricht (abgesehen von der momentan fehlenden Festplatte). Leider ist es noch nicht wirklich ausgereift und die Installation ist nicht einfach.

Automatisches Kopieren der Robotersoftware nach dem Betriebssystemstart in die RAM-Disk wäre möglich, erhöht aber die Fehleranfälligkeit des Systems und erschwert den mobilen Betrieb des Roboters.

Momentan habe ich mich für die erste Variante entschieden: Die Grundsoftware liegt auf der RAM-Disk und ist somit immer verfügbar. Bei Bedarf kann man auch eine eigene Version laden, dies kann entweder aus dem Home-Verzeichnis des Entwicklers geschehen oder durch überschreiben der mit der RAM-Disk geladenen Version durch eine eigene.

Die folgenden Schritte zur Einrichtung des RAM-Disk-Systems sind in [14] genau beschrieben.

RAM-Disk-Einrichtung

Prinzipiell ist eine RAM-Disk von der Softwareseite nichts anderes als eine sehr schnelle Festplatte, die aber aufgrund der begrenzten RAM-Ausstattung des Rechners etwas klein ausfällt. Da ursprünglich der RAM-Disk-Betrieb fest eingeplant war, wurde der Rechner mit 512MByte RAM angeschafft und damit ist ausreichend Speicher für die RAM-Disk vorhanden.

Als Hauptunterschied zu einer Festplatte verliert die RAM-Disk bei einem Neustart ihren Inhalt und muß neu geladen werden. Der Linux-Kernel unterstützt das Laden von komprimierten RAM-Disks, die im Speicher dekomprimiert werden. Dieses Feature ist hauptsächlich für das Laden von Installern und Notsystemen gedacht, die auf eine Floppy passen müssen, ist aber auch in unserem Anwendungsfall sinnvoll, da von dem eher langsamen Medium Festplatte oder CD-ROM (prinzipiell auch Netzwerk) nur noch ein Drittel der RAM-Disk-Größe geladen werden muß.

Beim Laden des Kernels muß das Image der RAM-Disk mitgeladen werden. Dies übernimmt der Bootloader LILO oder Syslinux/Isolinux, der den Kernel und eine initiale RAM-Disk (initrd) beim Start mitlädt, bevor der Kernel selbst gestartet wird. Der Kernel nimmt die Ram-Disk und entpackt sie, bevor er das Init-System davon startet und normal weitermacht. Danach geht es genau wie bei einem System weiter, das von Festplatte geladen wird.

Um das RAM-Disk-Image zu bauen wird zuerst eine Datei mit ausreichender Größe erstellt. Dies geschieht am einfachsten mit dem Befehl dd.

```
dd if=/dev/zero of=initrd.img bs=1k count=300000
```

kopiert vom Zero-Device (das nur Nullen liefert) 300.000 Blöcke mit der Blockgröße 1 kByte (also ca. 300 MByte) in die Datei initrd.img.

Als nächstes muß man diese Datei formatieren:

```
mke2fs initrd.img
```

erledigt dieses Formatieren (die Frage, ob man wirklich die Datei formatieren will, ist zu bejahen). Als Ergebnis hat man eine Datei, in der ein Dateisystem steckt und noch ganz viel Platz dazu. Um darauf zugreifen zu können, muß man die Datei mounten:

```
mount -oloop initrd.img /mnt
```

Unter Linux gibt es Loop-Devices (sofern man sie im Kernel mit eingebaut hat), die das Mounten einer Datei als Blockdevice erlauben, womit die Datei wie eine Festplatte ansprechbar ist.

In diese Datei kopiert man dann das komplette System, das auf dem Roboter laufen soll. Im Gegensatz zur Festplatte heißt die RAM-Disk nicht /dev/hda (oder so ähnlich), sondern /dev/ram0 (oder so ähnlich), daher muß man noch /etc/fstab *in dem Image* anpassen. Auch hat sich herausgestellt, daß bei dem RAM-Disk-Betrieb /etc/mtab nicht richtig aktualisiert wurde, weshalb es Sinn macht, /etc/mtab gegen /proc/mounts zu linken (was dank des virtuellen Proc-Dateisystems von Linux ohnehin die “aktuelleren” Daten über Mounts enthält, da es die Kernel-Tabellen ins Dateisystem mappt).

Nachdem man die alten Festplatteneinträge aus der /etc/fstab entfernt und mit cd /mnt verlassen hat, kann man es mit

```
umount /mnt
```

unmounten. An dieser Stelle bietet es sich an, einen Dateisystemcheck durchzuführen, damit man nicht ein defektes Dateisystem einpackt. Mit

```
fsck -f initrd.img
```

wird der normale Dateisystemcheck auf das Image angewandt. Danach muß es nur noch eingepackt werden. Ich habe eine Einpackvariante genommen, bei der die Originaldatei nicht gelöscht wird sondern für spätere Änderungen erhalten bleibt:

```
gzip -c initrd.img>initrd.gz
```

Wenn man das Ramdisk-Image soweit hat, muß es nur noch geladen werden. Hierzu gibt es zwei Möglichkeiten: Von Festplatte oder CD-Rom.

Laden der RAM-Disk von Festplatte

Bei dem Laden von Festplatte kopiert man das Image z.B. ins Root-Dateisystem auf einem kleinen Notsystem, das auf der Roboterfestplatte installiert ist (vereinfacht die Installation etwas) und fügt in /etc/lilo.conf die passenden Einträge hinzu. Dazu muß der passende Kernel und die initrd angegeben werden. Leider habe ich keine getestete Version einer entsprechenden lilo.conf greifbar, da sie auf der mittlerweile defekten Roboterrechnerfestplatte liegt (und es keinen Sinn macht, nur für diese Datei einen Rechner umzubauen - initrd=/etc/initrd.gz sollte reichen). Nach dem Aufrufen von lilo kann der Rechner in die RAM-Disk booten. In dieses RAM-Disk-Image sollte man noch

```
/sbin/hdparm -Y /dev/hda
```

in die /etc/init.d/bootmisc.sh eintragen, damit die Festplatte nach dem Laden des Systems abgeschaltet wird.

Laden der RAM-Disk von CD-Rom

Beim Laden von CD-Rom hat man etwas mehr Aufwand bei der Erstellung der CD, dafür aber einige Vorteile und ein geringeres Risiko (es ist nicht möglich, auf dem Roboterrechner die Installation zu beschädigen, auf dem Entwicklungsrechner kann man leicht Backups vom Roboterbetriebssystem anlegen, indem man das Image kopiert).

Für das Booten von der CD wurde Isolinux aus dem Syslinux-Paket ausgewählt, das unter http://syslinux.zytor.com/ zu finden ist.

Isolinux selbst besteht aus einem kleinen Loader (isolinux.bin), der anhand der in isolinux.cfg abgelegten Konfigurationsdaten den Kernel und das RAM-Disk-Image lädt und mit passenden Parametern startet. In der Datei Betriebssysteminstallationen.html ist das genaue Vorgehen beschrieben, das allgemeine Vorgehen (speziell auf Robogate bezogen) ist folgendes:

- su - eingeben, um root zu werden (ein Benutzer hat für cdrecord normalerweise keine ausreichenden Rechte).
- initrd.gz (das RAM-Disk-Image) nach /robo/cdimage/isolinux/ schreiben (siehe oben bei dem gzip-Befehl).
- Im Verzeichnis /robo den Befehl ./mkisoline ausführen. Dieser legt ein ISO9660-Image an.
- Den USB-Brenner an Robogate anschließen und dafür sorgen, daß eine Low-Speed CD-RW eingelegt ist.
- In demselben Verzeichnis den Befehl ./cdrecline ausführen. Nach einer Weile wird die CD ausgeworfen.

Nach diesem Schritt sollte es möglich sein, von der eben gebrannten CD zu starten. Bei der Handhabung von CDs sollte man sehr vorsichtig sein, da gerade die Oberseite extrem empfindlich ist.

2.3 Einbindung der Steuersoftware

Die Software für die Steuerung des Arms besteht aus mehreren Programmen, die einzeln geladen und ggf. neu gestartet werden müssen. Dazu kommt noch häufig das Java-Applet, daß auf dem Benutzerrechner läuft. Um die Nutzung einfach zu gestalten und keine weiteren UNIX-Kenntnisse vorauszusetzen, ist der Start der Programme über ein Web-Interface implementiert. Zur Zeit ist es noch ungesichert, es soll aber mit Zugangsschutz versehen werden.

Um die Software zu starten lädt man die URL `http://robo/Arm.html` und kann dort das Hauptkontrollprogramm *mcp* und das Armsteuerprogramm *arm* starten und beenden.

Das Java-Applet lädt man, indem man entweder einen Browser mit Java 1.4-Unterstützung oder den Appletviewer `http://robo/rcui.html` laden läßt. Ggf. muß der Name robo durch 192.168.100.1 ersetzt werden. Die Verbindung zum Web-Server auf dem Roboterrechner kann nur aus einem Rechner im WLAN erfolgen, dies kann entweder robogate sein oder ein Notebook oder ein anderer Rechner mit WLAN-Karte in Funkreichweite.

2.4 WLAN-Sicherheit

Bei der Netzanbindung und Nutzung des Roboters spielt das Wireless-LAN eine zentrale Rolle. Ohne die drahtlose Steuerung wäre an eine einigermassen einfache Nutzung und Steuerung des Systems nicht zu denken, weshalb man auch in den Angeboten der Roboterbeschaffung noch ein Angebot über eine Funk-Ethernet-Bridge findet, die allerdings ziemlich teuer war. Hilfreich war die mittlerweile sehr starke Verbreitung der WLANs, die dafür gesorgt hat, daß die Kosten für ein WLAN-System mittlerweile eher niedrig sind. Durch die Verbreitung ist aber auch die WLAN-Technik so üblich, daß die Wahrscheinlichkeit hoch ist, daß im nächsten Hörsaal jemand mit Notebook sitzt, der mitzumachen versuchen könnte. Daher muß man sich über die Implikationen des WLANs auf die Systemsicherheit Gedanken machen.

Prinzipiell sehe ich folgende Problemfelder:

- Kostenloser Internetzugang für Passanten und anonymer Internetzugang
- Kompromitierung der Rechner und Vandalismus

Der kostenlose Netzzugang für Passanten ist prinzipiell nicht wirklich problematisch, würde aber unter Umständen dem Institut in Rechnung gestellt und ist daher nicht wünschenswert. Auch könnte es zu Verkehrsbehinderungen auf dem Gehweg vor dem Labor kommen.

Problematischer ist der anonyme Internetzugang. Aufgrund der Tatsache, daß nur die Nähe zum Labor reichen würde um eine Verbindung zu erhalten ist eine Identifikation eines Netznutzers, der ein offenes WLAN missbraucht, nicht möglich. Daher bieten sich solche offenen WLANs geradezu für Leute an, die illegale Tätigkeiten vornehmen wollen. Um nicht mithaftbar zu werden ist ein solcher anonymer Zugriff unbedingt zu verhindern.

Diese beiden Problembereiche sind durch eine einfache Maßnahme in dem hier installierten System blockiert: Das WLAN ist als privates Netz eingerichtet, von dem aus kein Routing ins Internet stattfindet. Um eine Verbindung ins “echte” Netz zu kriegen ist ein Account auf Robogate nötig.

Auf die Sicherheit der Rechner wirkt sich das WLAN nur durch die sehr schwache Sicherung des Übertragungskanals aus. Man kann es mit einem allgemein zugänglichem Ethernet auf Basis von Koaxialkabeln vergleichen, bei dem ein Abgriff frei erreichbar ist. Damit ist ein Abhören der Verbindung leicht möglich. Um damit Zugriff auf die Rechner zu erhalten ist es aber nötig, entweder ein Passwort in Erfahrung zu bringen oder auf ein freigegebenes Verzeichnis, in dem Passwörter stehen, Zugriff zu haben. Durch die ausschließliche Nutzung der Secure Shell (SSH) für Login-Sessions ist das mitloggen eines Passwortes nahezu unmöglich (sofern die SSH nicht einen Fehler hat - sollte das der Fall sein, dann ist das Robotersystem aber das kleinste Problem). Freigegeben sind nur die Homes für Entwickler und Teile des Roboter-Dateisystems (sofern nötig), daher wäre schlimmstenfalls ein Zugriff auf die Home-Verzeichnisse der Entwickler möglich. Davon hätte ein Angreifer nicht wirklich viel. Das Robotersystem selbst ist durch die Nutzung einer RAM-Disk, die von CD-Rom gelesen wird, extrem robust: Selbst wenn das System kompromitiert wäre hätte der Angreifer keine Möglichkeit, etwas im Image zu ändern und alle von ihm gemachten Änderungen wären spätestens mit dem nächsten Neustart des Roboters weg.

Als letztes besteht prinzipiell noch die Möglichkeit, die Kontrolle über den Roboter zu übernehmen. Durch die verschiedenen Hardwaresicherheitsmaßnahmen ist für den Bewegungsbetrieb immer eine Handfreigabe erforderlich, die über das Netz nicht manipuliert werden kann. Daher würde der Versuch, über die WLAN-Verbindung mit dem Roboter Schaden anzurichten, sehr schnell von den Anwesenden gestoppt werden.

Sofern es Probleme mit der Nutzbarkeit durch ein “feindliches” WLAN geben sollte bietet es sich an, die Verbindungen zwischen robogate und robo durch kryptographische Maßnahmen (Tunneln der Verbindung) abzusichern. Dieser Schritt würde allerdings ein wenig Performance kosten (für die Verschlüsselung nötige Rechenzeit und Bandbreite, dazu erhöhte Latenz) und die Nutzung des Systems durch z.B. ein Notebook, das nicht besonders eingerichtet wurde, erschweren. Daher schlage ich vor, solche Maßnahmen erst dann zu implementieren, wenn sie sich als notwendig erweisen.

3 C167-Monitor zur Arm-Ansteuerung

Ursprünglich sollte die Steuerung von Fahrwerk und Arm in je einem C167-Microcontroller gekapselt werden, der über die serielle Schnittstelle an den Roboter-PC angeschlossen ist. Damit hätte man die Möglichkeit gehabt, die sicherheitskritische Steuerung in einem kleinen, überschaubaren System echtzeitfähig unterzubringen. Aufgrund von Hardwareproblemen und des hohen Aufwandes wurde dieser Projektteil für diese Arbeit abgebrochen, soll später aber wieder aufgenommen werden. Der aktuelle Stand dieses Projektteils wird hier dokumentiert.

3.1 Der C167CR

Der C167CR-Microcontroller, der in diesem Projekt eingesetzt wird, ist ein Vertreter der zweiten Generation der Siemens/Infineon C16x-Microcontrollerfamilie. Die Controller besitzen eine (für Microcontroller nicht ganz übliche) von-Neumann-Architektur mit einem 16 MByte umfassenden Adressraum. Für Microcontroller typisch sind Eigenschaften wie die gute Unterstützung von Bit-Befehlen, einer sehr flexiblen Interruptarchitektur mit Priorisierung und flexibler Peripherieeinheiten, die teilweise autonom arbeiten können.

Die Peripherie umfasst beispielsweise:

- 10-Bit A/D-Wandler mit 16 Kanälen
- Zwei 16-Kanal Capture/Compare-Einheiten mit flexiblen PWM-Einheiten
- Eine 4-Kanal-PWM
- Zwei Universaltimereinheiten
- Asynchrone und Synchrone serielle Schnittstelle
- Synchrone serielle Hochgeschwindigkeitsschnittstelle
- Watchdog-Timer
- On Chip CAN-Einheit (in der CR-Variante)
- Insgesamt 111 IO-Leitungen

Genauere Informationen dazu sind in [25] zu finden. Die Beschreibung des Befehlssatzes ist unter [26] zu finden.

3.2 Das Microcontrollerboard

Unsere C167CR-Microcontroller sind auf kitCON-167 Evalutaionsboards aufgelötet, die die einfache Inbetriebnahme der Microcontrollers erlauben. Dazu haben sie neben einem Lochrasterfeld, Stromversorgung und dem Controller noch folgende Komponenten:

- 256 KByte Flash
- 64 KByte SRAM
- Steckerleiste, auf der alle sinnvollen Signale verfügbar sind
- Eine RS232-Schnittstelle
- Eine CAN-Schnittstelle
- Einige LEDs

Nähere Informationen zu den Baugruppen sind unter [22] zu finden.

Ich habe vor einer Weile die Baugruppen mit einer automatischen Reset-Schaltung versehen, die beim Start des Debuggers einen Reset auslöst, womit eine unbeaufsichtigte Nutzung möglich ist. Dazu wird der Pegelwechsel einer Steuerleitung mit Hilfe von Kondensatoren zu einem Puls geformt und von einem FET-Transistor zum Betätigen des Reset-Signals des C167 genutzt.

3.3 Die Entwicklungsumgebung

Zum Programmieren des C167 wird ein schon vor dem Projekt vorhandenes C166-Entwicklungssystem der Firma HighTec EDV-Systeme GmbH `http://www.hightec-rt.com` verwendet. Aufgrund des Alters der uns vorliegenden Version ist es nicht wirklich komfortabel, für den Einsatzzweck aber ausreichend. Es besteht auch eine Einschränkung auf 64k Codegröße, die aber in diesem Projekt nicht relevant sein dürfte. Eine wichtige Einschränkung ist der Debugger/Loader, der den seriellen Port blockiert und nicht einfach nutzbar macht. Das Entwicklungssystem ist ein Port der GNU Toolchain, die man z.B. von Linux-Systemen kennt, für den C16x. Dabei läuft das Entwicklungssystem natürlich nicht auf dem Targetsystem, sondern wird als Crosscompiler und Remote-Debugger genutzt. Die wichtigsten Programme der Toolchain sind:

gcc166

Der verwendete Compiler ist ein reiner C-Compiler, also nicht wie häufig eingesetzt, ein C++-Compiler, der C-Code compiliert. Dieser Umstand tritt vor allem bei Kommentaren zutage, die nicht geschachtelt sein dürfen und nicht mit // eingeleitet werden können. Weiterhin hat man mit sehr schlechten Fehlermeldungen zu kämpfen, vor allem sind die Ortsangaben der Fehler sehr unpräzise, wenn man heutige Compiler gewohnt ist.

gmake166

gmake166 ist die GNU166-spezifische Version des Make-Tools. Eine Besonderheit dieses Make-Tools ist, daß die Makefiles nicht Makefile sondern makefile (man beachte das kleine m) heissen. Ansonsten benimmt sich das gmake166 wie das gewohnte GNU make.

gdb166

gdb166 ist der Port des GNU Debugger, der an den C166 angepasst wurde und als Remote-Debugger genutzt wird. Das Programmverhalten entspricht dem normalen gdb, den man beispielsweise von einem Linux-Entwicklungssystem kennt, wobei die einzige Ausnahme die Remote-Funktion ist. Grundsätzlich lädt der Debugger einen Debug-Kern in den Controller (mittels dessen Bootstrap-Mechanismus) und kopiert anschließend das zu debuggende Programm in den Controller. Danach kann man prinzipiell normal damit arbeiten, allerdings bleibt die serielle Schnittstelle fest in der Hand des Debuggers. Das Senden von Daten vom Microcontroller zum Host ist möglich, die andere Richtung ist aber nicht so einfach, da nicht vorgesehen. Aus diesem Grund wird der zum Entwicklungssystem gehörige Debugger gdb166 von dem Monitorsystem als reiner Programmloader genutzt. Sobald der Monitor gestartet wird, übernimmt er die Kontrolle über den Controller und auf PC-Seite beendet sich der gdb166 selbst. Das den gdb166 aufrufende Programm kann dann wieder die Kontrolle über die serielle Schnittstelle übernehmen und die Steuerung des Monitors beginnen. Alternativ kann man mit einem Terminalemulator (minicom) die Handsteuerung nutzen.

Der Debugger wird von der Datei .gdbinit gesteuert. Diese ist ein Batch-Script, die vom gdb166 automatisch aufgerufen wird. Ein Beispiel:

```
target c16x /dev/ttyS0 38400
load
detach go
quit
```

Die erste Zeile gibt die Art des Zielsystems an, dazu noch den Port, an dem es verbunden ist und die Geschwindigkeit, mit der die Schnittstelle arbeitet. Man kann diese auch weglassen, allerdings muß man dann damit rechnen, daß die Geschwindigkeit des Debuggers nicht mit der Geschwindigkeit des nachher genutzten Kontrollprogramms übereinstimmt und es damit keine Kontrollübernahme geben wird.

Der C167-Controller braucht keinen besonderen Monitor im Flash-ROM des Zielsystems, da er über einen einfachen Bootstrap-Mechanismus verfügt, mit dem die Schnittstellengeschwindigkeit automatisch erkannt wird und einige Bytes über die Schnittstelle in den Speicher geladen und ausgeführt werden können. Auf diesen Mechanismus setzt der gdb166 auf und lädt zuerst seinen Debugger-Code.

Die zweite Zeile lädt den Programmcode (wird dem gdb166 als Aufrufparameter übergeben) und danach wird in der dritten Zeile der Befehl zum Beenden der Debugverbindung und Ausführen des Codes gegeben, wonach das quit der 4. Zeile den gdb166 beendet. Hiernach kann mit der Steuerung durch das Hostprogramm begonnen werden.

Um den gdb166 “normal” zu nutzen, muss man nur die beiden letzten Zeilen auskommentieren (mit einem #), hat dafür aber keine Möglichkeit Eingaben an das Programm zu tätigen. Daher ist diese Betriebsart nicht mit dem von mir geschriebenen Monitorprogramm verträglich (welches dem Debuggercode sofort die Kontrolle entzieht und die komplette Steuerung des Microcontrollers übrnimmt).

3.4 Die entwickelte Software

Auf dem C167-Microcontroller wurde ein Steuerprogramm implementiert, mit dessen Hilfe die Grundsteuerbarkeit direkt gegeben ist (Handsteuerung), aber auch eine direkte Steuerung der Module durch ein PC-basiertes Steuerungssystem möglich ist. Es kommt kein kommerzielles Microcontrollerbetriebssystem zum Einsatz, diese Funktionen werden vom Monitor übernommen. In [13] ist die Software genauer beschrieben.

3.4.1 Grundstruktur

Das Monitorsystem besteht aus Kontrollflußsicht aus einem Initialisierungskontrollfluß und einigen Interruptbehandlungsroutinen. Beim Programmstart werden die meisten Initialisierungsfunktionen aufgerufen und dabei hauptsächlich folgende Aufgaben abgearbeitet:

- Kontrolle über die serielle Schnittstelle erlangen
- Timer- und CAN-Controller initialisieren und mit Interrupthandlern versehen
- Kontrolldatenstrukturen initialisieren
- Menü ausgeben

Nach diesen Initialisierungsroutinen wird nur noch auf Interrupts reagiert.

3.4.2 CAN-Controller

Eine Besonderheit des CAN-Busses ist der Broadcast-Betrieb mit der Festlegung der Nachrichtenart über den Nachrichten-Identifier. Passend dazu haben die CAN-Controller Filter, mit deren Hilfe die Controller eigenständig nur die Nachrichten empfangen, die für sie interessant sind. Diese Nachrichten werden nach Empfang gespeichert und, um den Microcontroller über die neue Nachricht zu informieren, wird ein Interrupt ausgelöst.

Der CAN-Controller des C167 besitzt eine feste Anzahl von Message-Objekten, die einzeln konfiguriert werden können. Die Objekte können als Sende- oder Empfangsobjekt dienen, wobei Empfangsobjekte ihren Nachrichten-Identifier als Filter nutzen. Ein Objekt ist als “catch-all”-Objekt implementiert und ist damit zur Behandlung der übriggebliebenen Nachrichten prädestiniert. Eine typische Nutzungsweise ist, häufig vorkommende Nachrichtentypen von festen Objekten zu behandeln und den Rest, also z.B. Fehlernachrichten, für das catch-all-Objekt übrig zu lassen. Die Objekte können auch in einem Modus betrieben werden, indem sie alle Nachrichten mit “ihrer” ID abfangen und speichern und damit immer nur die neuste Nachricht dieses

Types vorhalten. Dieser Modus ist besonders für Statusmeldungen interessant, bei denen nur der aktuelle Status interessiert, zum Empfang aber keine Rechenzeit verbraucht werden soll, beispielsweise die Statusnachrichten der PowerCube-Module mit M3-Firmware.

Die Nachrichtenobjekte des C167 haben folgenden Aufbau (vereinfacht):

- Message Control Register
- Arbitration-Feld
- Message Configuration Register
- Datenregister (8 Bytes)

Das Message Control Register enthält folgende Einträge[1]:

- RMTPND, Remote Pending; ein Remote-Frame wurde angefordert aber noch nicht versandt.
- TXRQ, Transmit Request; dieses Objekt soll gesendet werden.
- CPUUPD, CPU Update; dieses Sendeobjekt wird gerade von der CPU bearbeitet, der Controller soll es in Ruhe lassen.
- MSGLST, Message Lost; in diesem Empfangsobjekt wurde eine empfangene Nachricht von einer neueren Überschrieben. MSGSLT und CPUUPD sind das selbe Feld.
- NEWDAT, New Data; dieses Objekt enthält neue Daten.
- MSGVAL, Message Valid; dieses Objekt enthält gültige Daten. Ungültig sind die Daten bei Nichtbenutzung oder beim Schreiben.
- TXIE, Transmit Interrupt Enable; dieses Objekt erzeugt Sendeinterrupts.
- RXIE, Receive Interrupt Enable; dieses Objekt erzeugt Empfangsinterrupts.
- INTPND, Interrupt Pending; dieses Objekt hat einen noch nicht zurückgesetzten Interrupt erzeugt.

Im Code des Monitors wird das Message Control Register (MCR) von diversen Funktionen manipuliert, beispielsweise setzt transmit_filled() TXRQ und löscht CPUUPD:

```
void transmit_filled (ubyte ObjNr)
{
  CAN_OBJ[ObjNr].MCR   = 0xe7ff;  /*  set TXRQ,reset CPUUPD */
}
```

[1] Die Einträge haben alle 2 Bit, Der Wert 01 steht für den zurückgesetzten Wert, 10 für den gesetzten Wert beim Lesen. Beim Schreiben wird das Register mit 01 bzw. 10 gelöscht bzw. gesetzt. Wird 11 geschrieben, so wird der Wert nicht geändert.

Damit wird das Objekt ObjNr gesendet. Diese Routinen sind in can.c zu finden.

CAN-Übertragungen erfolgen zu einem großen Teil interruptgesteuert. Nachdem ein CAN-Objekt zur Übertragung freigegeben wurde, kann man per Interrupt über den Versand informiert werden und das nächste Objekt aus der Warteschlange (im Moment noch nicht implementiert) absenden.

Der Empfang von Nachrichten kann auch Interrupts auslösen, die dann z.B. zur Aktualisierung des Modulstatus genutzt werden können. Diese Interrupts werden vom Interrupthandler can_isr() bearbeitet, der in irq_routinen.c zu finden ist. Diese Funktion ist extrem lang und arbeitet alle möglichen CAN-Interruptmöglichkeiten ab. Im Pseudocode sieht die vereinfachte Funktion in etwa so aus:

```
while (noch_ein_interrupt_unbehandelt)
{
  switch (interruptquelle)
    {
      case Status_Change_Interrupt:
        if (Status_Interrupt)
          {
            Abarbeiten;
          }
        if (Receive_Interrupt)
          {
            Abarbeiten;
          }
        if (Bus_Error_Interrupt)
          {
            Abarbeiten;
          }
        break;
      case Error_Interrupt:
        Abarbeiten;
        Flag_löschen;
        break;
      case Message_15_Empfangsinterrupt:
        Abarbeiten;
        MCR=0x5599; /* Objekt zurücksetzen */
        break;
      case Message_1_Interrupt:
        Abarbeiten;
        Status_zurücksetzen;
        break;
...
    }
}
```

Abarbeiten steht bei Empfangsobjekten für die Verarbeitung der Nachricht und bei Sendeobjekten ggf. für das Nachladen der nächsten zu sendenden Nachricht.

Neben dem MCR[2] gibt es noch das Message Configuration Register (MCFG), das drei Funktionen hat:

- Länge der Nachricht
- Richtung (Senden oder Empfangen)
- Art der Nachricht, also Standard (11-bit) oder Extended (29-bit)

Beim Senden einer Nachricht gibt das MCR an, wieviele Bytes gesendet werden und beim Empfang kann damit die Anzahl der empfangenen Zeichen bestimmt werden. Vorsicht, beim Senden einer Nachricht muß man die Länge an zwei Stellen angeben: im MCFG und bei putData(). Vergißt man das MCFG, dann darf man sich über merkwürdige Fehler nicht wundern.

Ein Beispiel für das Senden einer Nachricht:

```
reset_module(ubyte modnum)
{
 ubyte dataarray[16];
     CAN_OBJ[0].MCR=0xf97f;
     CAN_OBJ[0].UAR = myCreateId(CANID_CMDPUT+modnum);
     CAN_OBJ[0].MCFG = 0x18;
     CAN_OBJ[0].MCR = 0xfebf;
     dataarray[0]=CMDCAN_DORESET;
     putData(0,dataarray,1);
     transmit_filled(0);
     waittxrq(0);
}
```

In diesem Beispiel aus der powercube.c wird im ersten Nachrichtenobjekt eine CAN-Nachricht an ein Powercube-Modul geschickt.

In der ersten Zuweisung wird CPUUPD gesetzt, NEWDAT gelöscht und MSGVAL gelöscht.

In der zweiten Zuweisung wird der für dieses Objekt passende Nachrichten-Identifier erzeugt (der Controller hat eine interessante Anordnung der Identifier-Bits im Register, daher sorgt myCreateId() für die korrekte Permutation der Bits).

Danach wird die Länge und Art der Nachricht (Länge=1, Sendeobjekt, Standard-Format) in MCFG geschrieben.

In der vierten Zuweisung wird NEWDAT und MSGVAL gesetzt.

Die fünfte Zuweisung schreibt das zu sendende Byte in den Buffer, von dem 1 Byte von putData() in das Objekt 0 kopiert wird. transmit_filled(0) sendet Objekt 0 und waittxrq(0) wartet darauf, daß das Objekt gesandt worden ist.

[2] Message Control Register

Eine Besonderheit des CAN-Busses ist die Nutzung von Masken. Mit diesen Masken kann beim Vergleich der Nachrichten-ID mit der Objekt-ID nur ein bestimmter Teil der ID berücksichtigt werden. Hiermit können Objekte bei passender Nachrichten-ID-Gestaltung ganze Klassen von Nachrichten abfangen. In diesem Projekt werden die Masken allerings nicht verwendet, daher werden sie bei der Initialisierung des CAN-Controllers komplett auf 1 gesetzt.

3.4.3 Timer

Der C167 verfügt über mehrere Timereinheiten, die Interrupts erzeugen können. Explizit wird im Moment nur ein Timer genutzt um periodisch eine Update-Funktion aufzurufen. Dieser Timer ist auch für die Heartbeat-Funktion zuständig.

Ein weiterer Timer kann für den Versand von periodisch zu sendenden Watchdog-Nachrichten genutzt werden.

3.4.4 Serielle Schnittstelle

Der größte Teil des Monitorprogramms behandelt die serielle Schnittstelle und das dazugehörige User Interface.

Empfang

Einkommende Zeichen (serieller Interrupt) werden entweder an das Menü-System weitergeleitet oder an die Direktsteuerung. Wenn das Menüsystem zuständig ist, dann werden die Zeichen in Abhängigkeit vom Zustand einiger Automaten ausgewertet und abgearbeitet. In der Direktsteuerung werden ankommende Zeichen in einen Ringpuffer geschrieben und eine Abarbeitungsroutine wird aufgerufen, die in Abhängigkeit von den Befehlen und der Anzahl der verfügbaren Zeichen Aktionen ausführt oder noch wartet.

Der Empfangsinterrupthandler in Pseudo-Code:

```
if (Handsteuerung_aktiv)
  {
    main_aut(Empfangnes_Zeichen);
  }
else
  {
    dcon_rxc(Empfangenes Zeichen);
  }
```

Der Handsteuerungsautomat (main_aut()) in Pseudo-Code:

```
void main_aut(char rc)
{
  if (Automatenzustand==0)
    {
      if (rc==’a’)
```

```
        {
          Automatenzustand=1;
          Memory-Inspektor-Menue-Ausgeben;
          Memory-Inspektor-Daten-Ausgeben;
        }
      if (rc==’b’)
        {
          CAN-Reset;
        }
...
    }
  else if (Automatenzustand==1)
    {
      if (subaut_meminsp(rc)==0)
        {
          Automatenzustand=101;
        }
    }
...
  else if (Automatenzustand==101)
    {
      Hauptmenü_ausgeben;
      Automatenzustand=0;
    }
...
}
```

Der Automat der Handsteuerung hat als obere Ebene die Automatenzustände und als nächste Ebene die Behandlung der Eingabezeichen. Damit ergibt sich eine Übergangsfunktion: $(Z_n, E) \rightarrow (Z_{n+1})$, wobei beim Zustandsübergang noch Aktionen ausgeführt werden können, beispielsweise die Ausgabe von Text. Z ist hier der Zustand und E die Eingabe. Die Zustandsmenge existiert nur implizit in Form der Zustandsvariablen, die Eingabemenge sind die Eingabe-Zeichen. In diesem Automaten werden alle unbekannten Zeichen einfach ignoriert. Die Aktionen sind typischerweise die Änderung des Zustandes und die Ausgabe eines neuen Menüs.

Bei der Direktsteuerung ähnelt die Struktur sehr der Struktur der PC-Software. Ein empfangenes Zeichen wird in einen FIFO geschrieben. Anschließend wird anhand des Füllstandes entschieden, ob sich die Entnahme lohnt. In Pseudo-Code sieht dcon_receive() etwa folgendermaßen aus:

```
if (verfuegbare_zeichen>=2)
  {
    switch (erstes_Zeichen_des_FIFO)
      {
        case Ser_SendReset:
```

```
            Entnehme_Zeichen_aus_FIFO;
            Befehlsnummer=erstes_Zeichen_des_FIFO_entnehmen();
            Aktion_ausfuehren;
            break;
...
        }
    }
if (verfuegbare_Zeichen>=3)
....
```

Nachrichten werden nur dann dem FIFO entnommen, wenn für den entsprechenden Befehl genügend Zeichen vorhanden sind, daher werden Befehle nur dann in Erwägung gezogen, wenn genügend Bytes im FIFO liegen.

Senden

Das Senden ist besonders problematisch. Während beim Empfang die Schnittstelle der begrenzende Faktor ist und durch die Automaten abgefangen wird, soll beim Senden der Programmfluß nicht durch die langsame Schnittstelle unterbrochen werden. Die vom Entwichlungssystem bereitgestellte Funktion serOutstring(char*) blockiert solange, bis der String gesendet wurde. Daher war es nötig, eine Sendewarteschlange zu implementieren, die einige Anforderungen erfüllen muß:

- ressourcensparend (kein Umkopieren von statischen Daten)
- keine dynamische Speicherverwaltung
- möglichst mit Hardwareunterstützung abarbeitbar

Die Schnittstelle des C167 nutzt beim Senden ein Register, in das das zu sendende Zeichen geschrieben werden muß. Wenn das Zeichen in das Ausgabeschieberegister kopiert wird, kann ein Interrupt ausgelöst werden, wenn das Zeichen komplett “auf dem Draht” ist, kann ein weiterer Interrupt ausgelöst werden.

Üblicherweise will man keine Pausen bei der Übertragung haben, daher bietet es sich an, um Interruptlatenzen zu maskieren, beim ersten Interrupt (Zeichen in Sendeschieberegister übernommen) schon das nächste Zeichen nachzuladen.

Die Sendeinterruptroutine schreibt das nächste zu sendende Zeichen ins Senderegister und setzt den Pointer, der auf das nächste zu sendende Zeichen zeigt, eine Position weiter. Wenn der aktuelle String fertig gesendet wurde, wird, sofern noch ein String in der Warteschlange steht, der nächste String genommen. Der Code dazu sieht folgendermassen aus:

```
/* ISR for Ser_transmit */
void s0tb_interrupt(void)
{
  if (main_aut_state!=99)
```

```
    {
#ifndef SER_TR_PEC
      if (s_tobesent>0)
        {
          s_tobesent--;
          S0TBUF=*s_curptr++;
        }
      else
#endif
        {
          if (s_curobj!=NULL)
            {
              s_curobj->used=0;
              if (s_curobj->next!=NULL)
                {
                  s_curobj=s_curobj->next;
#ifdef SER_TR_PEC
                  SRCP1=(int)&(s_curobj->data[1]);
                  PECC1=0x0500+s_curobj->size-1;
#else
                  s_curptr=&(s_curobj->data[1]);
                  s_tobesent=s_curobj->size-1;
#endif
                  s_curobj->used=1;
                  S0TBUF=s_curobj->data[0];
                }
              else
                {
                  s_curobj=NULL;
                }
            }
        }
    }
  else
    {
      dcon_txint();
    }
}
```

Der erste Teil (um das erste #ifdef) ist für das Senden innerhalb eines Strings zuständig, der Teil hinter dem ersten #endif bis zum letzten *else* kümmert sich um das Nachladen des Strings. Die #ifdef-Blöcke sind die unten beschriebene Vorbereitung auf den PEC-Betrieb. Sofern die Präporzessorkonstante *SER_TR_PEC* definiert ist, wird der Peripheral Event Contoller (eine Art DMA-Controller) verwendet.

Damit man mehrere Strings nahezu gleichzeitig in eine Warteschlange packen kann, bietet sich eine Struktur mit mehreren Sendeobjekten an, die die entsprechenden Strings und noch ein paar Verwaltungsinformationen speichern.

Ich habe jedes Objekt mit einem eigenen Buffer und einem Zeiger auf den zu sendenden String versehen. Dazu kommt noch ein Zähler für die Anzahl der zu sendenden Zeichen, ein Pointer auf das nächste zu sendende Objekt und zuletzt noch ein Benutzungsstatus, der bei der Suche nach einem freien Sendeobjekt ausgewertet wird. Die genutzte Struktur sieht folgendermaßen aus:

```
struct ser_sendobj
{
  unsigned char size;
  unsigned char used; /* 0 - Unused, 1 - active, 2 - pending */
  struct ser_sendobj* next;
  char* data;
  unsigned char buffer[ser_send_bufsize];
};
```

Beim Senden von zur Laufzeit generierten Strings gibt es ein Probem: Lokale Variablen einer Funktion verlieren nach Rücksprung ihre Gültigkeit. Dies gilt auch für Stringbuffer, die z.B. von

```
char buf[100];
sprintf(buf,”Die Variable d hat den Wert %d\n”,d);
```

genutzt werden. Als Lösung könnte man sich in der Funktion per *malloc* einen String besorgen. Das Problem dabei ist aber, wer ihn wieder freigeben soll. Grundsätzlich jeden String nach Senden freizugeben würde das Senden von statischen Strings, die mit dem Code des Programms erzeugt werden, erschweren. Als Lösung habe ich in jedem Sendeobjekt einen Buffer eingebaut, der zur Laufzeit generierte Strings aufnehmen kann. Um statische Strings nicht kopieren zu müssen, werden grundsätzlich nur die Strings, auf die der Zeiger eines Sendeobjektes zeigt, gesendet. Bei einem zur Laufzeit generierten String muß man den Zeiger auf den Buffer des Strings setzen und hat das Problem ohne besondere Behandlung gelöst.

Das Senden eines zur Laufzeit generierten Strings sieht folgendermassen aus:

```
onr=ser_send_getobj();
sprintf(s_sobj[onr].buffer,"\033\1331A %0d A",4);
ser_send_sendobj(onr);
```

In der ersten Zeile wird ein Sendeobjekt reserviert, in der zweiten Zeile in den Buffer des Sendeobjektes der String geschrieben und in der dritten Zeile das Objekt in die Warteschlange gesteckt. Das Senden eines konstanten Strings ist dagegen sehr einfach:

```
ser_send_str("\033\1332J");
```

Prinzipiell kann der C167 auch, sofern die Daten bestimmten Anforderungen hinsichtlich der Aufbewahrung genügen, ohne Interrupt Strings senden. Dazu gibt es den PEC[3] , der neben einem Zähler für die Anzahl der zu übertragenden Bytes/Worte auch noch einen automatisch in/dekrementierbaren Zeiger enthält. Der PEC kann prinzipiell einen String komplett ohne Interruptaufruf senden und der Interrupt wird nur einmal am Ende des Strings zum Nachladen des nächsten Strings genutzt. In dem hier implementierten System scheitert die Nutzung des PEC an der Unterbringung der Sendeobjekete. Zur Zeit ist das System aber auch so unterfordert, daß die Rechenzeit für den Interruptbetrieb die Umstellung auf PEC-Betrieb nicht rechtfertigt. Das System ist aber konzeptionell darauf vorbereitet und sollte mit geringem Aufwand umstellbar sein.

3.4.5 Hilfsfunktionen für den Betrieb von PowerCube-Modulen

Für die Nutzung der PowerCube-Module wurden einige Hilfsfunktionen geschrieben, die analog zur Morse-Bibliothek (auf dem PC) einen High-Level-Zugriff auf die Module erlaubt. Im Unterschied zur PC-Bibliothek habe ich auf dem Microcontroller Rücksicht auf die Handsteuerung genommen und eine Datenstruktur entwickelt, die alle relevanten Parameter und Zustände speichert. Beispielsweise sendet der Aufruf von

```
powercube_info[modnum].pos=position;
moveramp_module(modnum);
```

den Befehl an das Modul *modnum*, an Position *position* zu gehen. Position ist ein Festkomma-Format. Die Funktionen stehen in powercube.c und powercube_config.c.

[3] Peripheral Event Controller

3.4.6 Besonderheiten der Microcontrollerprogrammierung

In der genutzten Konfiguration werden 64kByte RAM verwendet. Dieser Umstand sollte bei der Programmierung unbedingt berücksichtigt werden, um Speichermangel und die daraus resultierenden Probleme wie Abstürze zu vermeiden. Rekursionen sollten unbedingt vermieden werden, da sie den Stack sehr stark belasten.

Dynamische Speicheralokation ist möglich, es sollte aber unbedingt darauf geachtet werden, daß der Speicher wieder freigegeben wird und nicht "verschwindet". Speicherlecks, die bei PCs mittlerweile üblich sind (man hat ja 256M oder mehr und kann damit leben, den PC einmal am Tag neu zu starten...) kann man sich auf einem Microcontroller nicht leisten, da zum einen sehr wenig Speicher verfügbar ist und zum anderen Abstürze und Fehlfunktionen schlimme Folgen haben können. Daher möglichst schon bei der Programmierung so programmieren, daß riskante und schwer zu debuggende Fehler nicht auftreten können.

Als weiteres Problem der Microcontrollerprogrammierung kommt die Beschränktheit der Hardware hinzu. Ein Microcontroller hat nicht selten keine Hardware für Floating-Point-Berechnungen, etc. Daher sind in einem Integer-Microcontroller Floating-Point-Berechnungen nach Möglichkeit zu vermeiden, über Software sind sie aber möglich.

Im Unterschied zu einem PC hat man auf dem Microcontroller üblicherweise direkten Zugriff auf eine ganze Menge von teilweise sehr mächtigen Peripherieeinheiten, die allerdings sehr stark optimiert sind und damit nicht immer einfach zu verstehen sind. Häufig sind spezielle Befehle und Hardwarefunktionen nur dann verständlich, wenn man die Probleme kennt, für deren Lösung die Hardware entwickelt wurde. Um nicht die ganze Leistungsfähigkeit der Hardware zu verschenken, wird nur selten eine Abstraktionsschicht eingesetzt um die Hardware zu verstecken, daher sind viele auf dem PC einfache Funktionen (scanf("%d",&i);) alles andere als einfach zu implementieren. Dagegen ist z.B. das Einschalten einer I/O-Leitung sehr einfach (P2=1;).

Wenn man anfängt, mit Microcontrollern zu arbeiten, muß man sich zuerst umgewöhnen, danach sind es durchaus "normal" nutzbare Systeme.

4 Ansteuerung des Arms auf dem PC

4.1 Steuerprogramme auf dem Roboter-PC

Als Steuersystem ist ein stark modulares System, bestehend aus einem zentralen Steuerprogramm und mehreren damit kommunizierenden Modulen, implementiert. Als Kommunikationsweg werden normale TCP-Verbindungen genutzt, mit denen sich die Module mit dem Hauptkontrollprogramm verbinden. Nach der Verbindung werden über die TCP-Kanäle Nachrichten zwischen den Modulen und dem Hauptkontrollprogramm ausgetauscht. Durch die Verwendung von Standardschnittstellen ist es problemlos möglich, einzelne Module gegen geänderte Module zu ersetzen oder auch Aufgaben auf andere Rechner auszulagern, wenn die Rechenleistung des mobilen Rechners nicht ausreichen sollte. Das Benutzerschnittstellenmodul wird auf einem anderen Rechner ausgeführt, um die graphischen Ausgaben nicht über das Funk-Netz senden zu müssen. Die Kapselung der Funktionen in einzelne Prozesse bietet als Vorteil die Überschaubarkeit der einzelnen Module und Sicherheit durch die Überprüfbarkeit der definierten Schnittstellen.

Hauptsteuerprogramm

Als zentrales Element des Systems laufen im Hauptsteuerprogramm alle Verbindungen zusammen. Zwischen normalen Modulen gibt es keine Direktverbindungen, wodurch die Austauschbarkeit dieser Module ermöglicht wird. Als zentrales Programm gewährleistet das Hauptkontrollprogramm die Einhaltung der Grundsicherheit, in dem nur bestimmte Kontrollflußwege möglich sind. Das User-Interfacemodul hat beispielsweise keine Möglichkeit, einen direkten Befehl an die Aktoren zu senden, der die Sicherheitsprüfungen umgeht. Als Designvorgabe soll das Hauptsteuerprogramm ziemlich “schlank” und sauber implementiert werden.

Der Sourcecode vom *mcp* unterteilt sich in einen Hauptprogrammteil (mcp.c) und verschiedene Dateien, die die einzelnen Verbindungen behandeln. Die einzelnen Verbindungshandler können die anderen Handler aufrufen und damit kann eine ankommende Nachricht behandelt werden. Nach der Initialisierung der Datenstrukturen und Handler wartet *mcp* auf ankommende Verbindungen und Daten. Kommt eine Verbindung an, wird gefragt, um was für eine Verbindung es sich handelt und entsprechend der Antwort auf den passenden Handler weitergereicht, sofern er frei ist.

Ankommende Daten werden direkt an den entsprechenden Handler weitergeleitet und dort ausgewertet.

Ich werde jetzt die Hauptfunktionalität in Pseudocode angeben:

```
while (1)
{
  Füge alle Deskriptoren in select-Datenstruktur ein;
  select();
  if (Neue Verbindung angekommen)
    {
      Sende Begrüßungsfrage;
      Lege neue Verbindung in Liste der neuen Verbindungen;
    }
  if (Daten für neue Verbindungen angekommen)
    {
      if (Antwortzeichen bekannt)
        {
          if (Slot für Verbindungstyp frei)
            {
              Verbindung in Slot legen;
              Start-Nachricht senden;
            }
          else
            {
              Verbindung trennen;
            }
        }
    }
  if (Daten für Verbindungsslot angekommen)
    {
      connhand_<Modulname>() aufrufen;
    }
  if (neue Sekunde seit letzter Prüfung)
    {
      Heartbeat-Funktion aufrufen;
    }
}
```

Die einzelnen Modulhandler gehen in etwa folgendermassen vor:

```
connhand_<Modulname>(void)
{
  Lese Daten und füge Daten in FIFO ein;
  if (FIFO_Füllstand >= minimale_Befehlslänge)
    {
      Identifiziere Befehl;
      if (FIFO_Füllstand >= Länge_für_identifizierten_Befehl)
        {
```

```
            Decodiere Daten;
            Entferne Befehl aus FIFO;
            Rufe Funktion für identifizierte Daten auf;
          }
        }
    }
```

Die aufgerufene Funktion kann ihrerseits auch wieder Nachrichten senden.

Aus diesem Aufbau folgt, daß die Handlermodule unbedingt sorgfältig zu implementieren sind, da die Verbindungshandler die kritische Stelle des Systems sind.

Arm-Steuermodul

Das Arm-Steuermodul ist für die Kommunikation mit dem Arm zuständig und bildet die Abstraktionsschicht zur Hardware hin. Momentan nutzt es eine auf die von ESD gelieferte CAN-Bibliothek aufsetzende, selbstgeschriebene Bibliothek, die Morse5-kompatibel ist. Ursprünglich war vorgesehen, das Arm-Modul auf eine Bibliothek aufzusetzen, die den Monitor im C167 als intelligenten CAN-Controller nutzt. Dieses Teilprojekt wurde aber aus praktischen Gründen zu dieser Zeit nicht fortgeführt.

Anstelle des auf den Arm zugreifenden Moduls ist über die Arm-Schnittstelle auch der Einsatz eines Arm-Simulators möglich, der dieselbe Softwareinfrastruktur wie das “echte” System hat, aber kein Risiko für Hardware bietet und bei Übungen keine Betreuung braucht.

Das Arm-Modul soll auch als Sicherheitsfunktion prüfen, ob der übergebene Vektor zulässig ist und den Bewegungsbefehl nur an den Arm weiterleiten, wenn keine Kollision des Armes mit dem Fahrwerk zu befürchten ist. Um die Komplexität des Moduls gering zu halten, ist keine Prüfung auf Kollisionen mit Objekten in einem später zu entwickelnden Weltmodell möglich, diese muß aber in ein Berechnungsmodul ausgelagert werden, da sie die Komplexität der Arm-Moduls zu stark erhöhen würde.

Fahrwerksteuermodul

Das Fahrwerksteuermodul übernimmt, analog zum Arm-Steuermodul, die Kommunikation mit dem Fahrwerk. Da es aufgrund der schlechten Dokumentation des Fahrwerks ziemlich zeitaufwendig zu implementieren ist (Reverse-Engineering kostet viel Zeit) und, um die Aufgabenstellung realisierbar zu halten, nicht Teil dieser Arbeit ist, wird es hier nicht näher behandelt. Dieses Modul ist zum aktuellen Zeitpunkt fest im Systemkonzept eingeplant.

Berechnungsmodule

Verschiedene Berechnungsfunktionen, z.B. Kollisionserkennung, Trajektorienberechnung, werden als eigene Programme implementiert, die sich mit dem Hauptsteuerprogramm verbinden. Hierdurch wird es leicht möglich, in Übungen und Projekten eigene Algorithmen und Optimierungen einzubringen ohne das Grundsystem dabei ändern zu müssen.

Sinnvoll erscheint mir, im Laufe des Projektes ein Weltmodell zu implementieren, anhand dessen die auszuführenden Bewegungen geplant und geprüft werden können. Mit einem solchen Modell sollte eine Berücksichtigung der Umwelt des Roboters möglich werden, z.B. um Kollisionen mit Wänden zu vermeiden. Weiterhin wären Hilfsfunktionen wie die Koordinatenumrechnung und Kollisionserkennung für die Entwicklung eigener Module sinnvoll, da dann nicht jedes Modul seine eigenen Routinen implementieren müsste. Diese Module werden, da sie nicht Teil dieser Arbeit sind, nicht näher behandelt.

Benutzerinterface

Das Benutzerinterface ist auch als Modul implementiert. Hiermit sind verschiedene Steuerungsmöglichkeiten denkbar, von einem Java-Applet auf einem Notebook bei Vorführungen bis hin zur Internet-Steuerung (ggf. in Kombination mit einem Simulator). Ein Beispiel-Java-Applet wurde implementiert, mit dem man eine einigermaßen simple Teach-In-Steuerung nutzen kann. Dieses Applet ist bei Erweiterungen des Robotersystems um neue Funktionen so anzupassen, daß die Funktionen nutzbar sind. Sobald die Bewegungsplanung implementiert ist, kann die Vorgabe in Raumkoordinaten geschehen und die Eingabe über einen Spaceball wäre sinnvoll. Zur Zeit kann dieses Interface von einem beliebigen Rechner mit installiertem JRE/JDK, der im WLAN ist, genutzt werden. Der Start des Systems bei der Roboterrechnerbetriebssystembeschreibung beschrieben.

4.2 Schnittstellendokumentation

4.2.1 Modulschnittstellen zum Hauptkontrollprogramm

Die Schnittstelle zum Hauptkontrollprogramm nutzt einen TCP-Stream. Der Client verbindet sich zu dem Hauptkontrollprogramm auf Port 3000[1] und erhält “What do you want to do?” vom Hauptkontrollprogramm. Der Grund für diese “höfliche” Begrüssung ist, daß die Schnittstelle auch von Menschen für Debug- und Kontrollverbindungen genutzt werden kann.

Als Antwort auf die Frage erwartet das Hauptkontrollprogramm ein Zeichen, mit dem es das verbindende Programm identifizieren kann. Im Moment sind folgende Zeichen genutzt:

d Debug: Debugmeldungen der einzelnen Modulverbindungen werden hier ausgegeben

c Control: Telnet-Steuerung der Funktionen

J Java-Control: In Java implementiertes User-Interface

a Arm: Modul, das den Arm steuert

T Test: Dummy-Interface, das als Muster für neu zu schreibende Verbindungstypen genutzt werden kann. Der Code hierfür enthält keine Besonderheiten und sollte daher leicht anpassbar sein.

[1] 3000 wurde wilkürlich gewählt und könnte geändert werden. Bei der Änderung müssten alle Module entsprechend angepasst werden.

Wenn die Verbindung zugeordnet wurde, werden alle Daten an das passende Modul für diese Verbindung weitergeleitet und können damit ausgewertet werden.

Für einen Verbindungstyp kann es üblicherweise maximal eine Verbindung bestehen, daher werden weitere Verbindungen mit dem Typ abgelehnt.

Debugverbindung

In der Debugverbindung ist eine menügesteuerte Abfrage von Systemparametern möglich, weiterhin können Debugmeldungen ausgegeben werden.

In dem Modul werden Eingaben mit Hilfe eines Automaten, dessen Zustandsvariable state_debug ist, ausgewertet. Zustand 0 ist das Hauptmenü, von dort kann mit Eingabe eines ? eine kurze Hilfenachricht abgerufen werden, mit den dort genannten weiteren Zeichen werden andere Zustände ausgewählt, die dann ihrerseits Untermenüs sind. Als Beispiel ist v als Variableninspector angedeutet. Jedes Untermenü sollte die Möglichkeit bieten, wieder zurück ins Hauptmenü zu gelangen, z.B. mit der Eingabe vom Zeichen “q” für Quit.

Momentan dürfte die wichtigste Anwendung die Ausgabe der Debugmeldungen sein. Debug-Nachrichten der Hauptsteuerungsmodule sind einer Prioritätsklasse zugeordnet, mit steigender Klasse werden sie immer wichtiger. In den Handler-Modulen wird eine Debug-Ausgabe folgendermassen durchgeführt:

```
sprintf(debugstr,"arm: receive_char: %d %x %d\n",
r_state,c,receivebuffer_availchars(rcb));
connhand_debug_writemessage(DEBUG_LEVEL_ALL,debugstr);
```

Dabei wird der String debugstr mit der Klasse 7 (Alle Nachrichten) ausgegeben. Die Ausgabe erfolgt nur, wenn eine Debug-Verbindung besteht.

Controlverbindung

Ähnlich der Debugverbindung ist die Kontrollverbindung für die Benutzung durch Menschen gedacht. Anders als die GUI-Kontrolle soll hier aber ein reines Textinterface implementiert werden, mit dem über ein Terminal bzw. eine Telnetverbindung die Steuerung möglich ist.

Wie auch bei der Debugverbindung werden die Eingaben zuerst von einem Automaten (Variable ist state_control) ausgewertet und dieser kann entweder Untermenüs implementieren oder Eingaben direkt auswerten.

Zur Zeit ist vermutlich die interessanteste Funktion das Beenden des Systems durch Eingabe eines ’Q’s.

Java-GUI-Verbindung

Im Gegensatz zu den beiden ersten Verbindungstypen ist die Verbindung zur Java-GUI nicht für die direkte Nutzung durch Menschen gedacht sondern das Protokoll ist auf einfache Lesbarkeit durch Programme ausgelegt. Daher wird hier kein ASCII- sondern ein Byteformat genutzt.

Die Schnittstelle zum Java-GUI hat die Aufgabe, Statusmeldungen zu dem GUI zu transportieren und Befehle vom GUI anzunehmen und passend weiterzuleiten. Einige der Nachrichten

entstehen asynchron (d.h. unabhängig von einem Befehl), daher muß das Protokoll Nachrichten in beide Richtungen ohne vorherige Anforderung zulassen, also z.B. vom UI ausgehend Befehle und vom Hauptkontrollprogramm ausgehend Statusmeldungen. Weiterhin ist ein Heartbeatmechanismus sinnvoll, mit dem bei Unterbrechung der Verbindung sofort geeignete Massnahmen eingeleitet werden können, z.B. eine Bremsung.

Arm-Verbindung

Diese Verbindung ist, wie die GUI-Verbindung, eine Binärverbindung, über die das Arm-Interface angesteuert werden kann. Das für die Java-Verbindung geschriebene gilt analog auch für die Arm-Verbindung.

4.2.2 Nachrichtenformat

Das Protokoll für die Kommunikation zwischen den Modulen ist in [16] dokumentiert. Das Protokoll hat folgenden Aufbau:

- [int32] Nummer des Befehls
- [int32] Befehl
- Daten

Die Nummer des Befehls dient der Zuordnung von Antworten zu Befehlen, Antworten haben dieselbe Nummer.

Die Befehle (in commandnums.h definitert) legen die Aktion und die auf den Befehl folgenden Daten fest. Durch die Nutzung von 32-bit-Zahlen ist der Befehlsraum mit ausreichend Reserven planbar. Momentan habe ich folgende Konventionen festgelegt:

- Module haben einen Befehlsraum von 100000 für sich
- Anfragen liegen in der ersten Hälfte dieses Raumes (0-45999), Antworten auf eine Anfrage liegen im Raum von 50000-99999, Antworten sollten üblicherweise Anfragenummer+50000 sein.
- Aufeinanderfolgende Befehle sollten einen Abstand von 100 haben.

Im Moment sind folgende Befehle im Einsatz:

Befehl	Nummer	Antwortnummer	Funktion
CMD_U_ECHO	0	0	Lebenszeichen geben
CMD_U_SHUTDOWN	1	-	Verbindung beenden
CMD_A_SAV	100000	150000	Arm: Ein Gelenk bewegen
CMD_A_SAGP	100100	150100	Arm: Parameter liefern

Die ersten beiden Zeilen enthalten die Befehle, die sich gerade nicht an die Konventionen halten: Echo wird genauso zurückgesendet und hat nur den Sinn die Kommunikation zu prüfen.

Shutdown wird von einem Modul zum Hauptkontrollprogramm gesandt, um diesem das Beenden des Moduls anzuzeigen. In der Gegenrichtung ist es die Aufforderung an das Modul, sich zu beenden.

Die genaue Dokumentation ist in [16] zu finden, die Parameter sind in commandnums.h definiert (und werden von dort benutzt).

4.2.3 Schnittstelle zum Monitorprogramm auf dem C167-Microcontroller

Zum Monitorprogramm auf dem Microcontroller besteht “nur” eine serielle Verbindung, über die zum einen die Handsteuerung abgewickelt wird, als auch die Direktsteuerung. Daher startet das Monitorprogramm mit der Handsteuerung und die Umschaltung auf Direktsteuerung erfolgt mit einem “x”.

Nach dem “x” versucht die Direktsteuerung mit dem steuernden Programm die Kommunikation aufzunehmen. Wenn diese Kommunikation erfolgreich aufgebaut wurde, ähnelt das Protokoll dem Protokoll der TCP/IP-Verbindungen zwischen *mcp* und den Modulen. Zur Zeit ist noch nicht viel implementiert, da dieser Projektteil zu früh abgebrochen wurde. Die Mechanismen entsprechen ziemlich genau den Mechanismen für die Kommunikation zwischen den PC-Programmen.

Im Verzeichnis Roboter/C166 ist der Source für das Gegenstück zum Monitor zu finden. Dieses Programm lädt den Monitor auf den C167 und startet danach automatisch die Direktsteuerung.

Literaturverzeichnis

[1] Webseite. http://www.can.bosch.com/.

[2] Webseite. http://www.can.bosch.com/content/What_is_CAN.html.

[3] Modifizierter Gesamtschaltplan ORC 650mm. Im Labor im MIAG-Ordner oder gescannt unter Anleitung/ordnerscans/miag-doku_1/kscan_0001.png auf der CD zu finden.

[4] Tiefentladeschutzschaltung. Die Schaltung wurde von Carsten Giesemann gefunden und in modifizierter Form gebaut. Der Schaltplan ist im MIAG-Ordner oder als Scan unter Anleitung/ordnerscans/miag-doku_1/kscan_001d.png auf der CD zu finden.

[5] AMTEC GmbH, Pankstrasse 8-10, 13127 Berlin. *PowerCube Betriebshandbuch*, 1.11 edition, 7 July 2000. Dieses Handbuch ist auf der CD unter Anleitung/HerstellerDokus/PowerCube-Handbuch.pdf zu finden.

[6] Beckmann+Egle Industrieelektronik GmbH, Kirchstrasse 30, 71394 Kernen. *Betriebsanleitung für Digital-Einbauinstrumente der Typen EX2068, EX2069, EX2070 und EX2071*. Die Anleitung ist im MIAG-Ordner zu finden, eine Scan davon unter Anleitung/ordnerscans/miag-doku_1/kscan_001c.png und Anleitung/ordnerscans/miag-doku_1/kscan_001b.png auf der CD.

[7] CAN in Automation, Am Weichselgarten 26, 91058 Erlangen. *CAN Specification 2.0 Part A*. Diesers Dokument steht unter http://www.can-cia.de/can/standardisation/can.standards.html zum Download zur Verfügung und ist auf der CD unter Anleitung/HerstellerDokus/CAN/CAN20A.pdf zu finden.

[8] CAN in Automation, Am Weichselgarten 26, 91058 Erlangen. *CAN Specification 2.0 Part B*. Dieses Dokument steht unter http://www.can-cia.de/can/standardisation/can.standards.html zum Download zur Verfügung und ist auf der CD unter Anleitung/HerstellerDokus/CAN/CAN20B.pdf zu finden.

[9] Christian Asam. Anleitung zur Erweiterung eines Moduls um Funktionen. HTML-Datei, February 2003. Diese Datei ist auf der CD unter Anleitung/Software/MainControl/Funktionserweiterung_Modul.html verfügbar.

[10] Christian Asam. Bedienungsanleitung für das Robotersystem der technischen Informatik. HTML-Datei, January 2003. Die Datei ist auf der CD unter Anleitung/Hardware/anleitung.html zu finden.

[11] Christian Asam. Beschreibung des Arms. HTML-Datei, February 2003. Diese Datei ist auf der CD unter Anleitung/Hardware/Arm.html verfügbar.

[12] Christian Asam. Beschreibung des Fahrwerks. HTML-Datei, January 2003. Diese Datei ist auf der CD unter Anleitung/Hardware/Fahrwerk.html verfügbar.

[13] Christian Asam. Beschreibung des Monitorprogramms für den C167. HTML-Datei, January 2003. Diese Datei ist auf der CD unter Anleitung/Software/Monitor/Monitor.html verfügbar.

[14] Christian Asam. Besonderheiten der Betriebssysteme der beteiligten Rechner. HTML-Datei, February 2003. Diese Datei ist auf der CD unter Anleitung/Software/Betriebssysteme/Betriebssystemeinstallationen.html verfügbar.

[15] Christian Asam. Erweiterungsanleitung für MCP. HTML-Datei, February 2003. Diese Datei ist auf der CD unter Anleitung/Software/MainControl/Erweiterungsanleitung.html verfügbar.

[16] Christian Asam. Protokoll der TCP/IP-Verbindungen. HTML-Datei, February 2003. Diese Datei ist auf der CD unter Anleitung/Software/Struktur/Protokoll.html verfügbar.

[17] Christian Asam. Softwarestruktur des Robotersteuersystems. HTML-Datei, 2003. Diese Datei ist auf der CD unter Anleitung/Software/Struktur/Struktur.html verfügbar.

[18] Christian Asam. Webserver auf dem Roboter. HTML-Datei, February 2003. Diese Datei ist auf der CD unter Anleitung/Software/Betriebssysteme/Roboterwebserver.html verfügbar.

[19] eas. *Beschreibung Antriebsregler MPC 140/36*. Diese Anleitung ist im Labor im MIAG-Ordner zu finden.

[20] jmu. *Gesamtschaltplan ORC 650mm*. MIAG Fahrzeugbau GmbH, 1.3 edition, 18 December 1996.

[21] Philips Semiconducters. *CAN Serial Linked I/O device (SLIO) with digital and analog port functions*, 19 June 1996. Dieses Dokument ist auf der Herstellerwebseite zu finden und steht auf der CD unter Anleitung/HerstellerDokus/82c150.pdf zur Verfügung.

[22] Phytec Meßtechnik GmbH, Robert-Koch-Str. 39, 55129 Mainz. *kitCON-167 Hardware-Manual*, june 1997 edition, June 1997. Dieses Handbuch ist auf der Web-Seite des Herstellers oder auf der CD unter Anleitung/HerstellerDokus/c16x/KitCON167-HardwareManual.pdf zu finden.

[23] Robert Bosch GmbH, Postfach 30 02 40, 70442 Stuttgart. *CAN Specification, Version 2.0*, September 1991. Dieses Dokument steht unter http://www.can.bosch.com/docu/can2spec.pdf zum Download zur Verfügung und ist auf der CD unter Anleitung/HerstellerDokus/CAN/can2spec.pdf zu finden.

[24] Sebastian Wagner. Intelligente Kommunikation durch Einbettung des CAN-Bus zur Bahnpositionierung eines autonomen mobilen Systems. Master's thesis, TU Clausthal, Institut für Informatik, 1999. Diese Diplomarbeit befasst sich mit dem Fahrwerk. Ein Scan ist unter Anleitung/ordnerscans/Sebastian_Wagners_Diplomarbeit auf der CD zu finden.

[25] Siemens/Infineon. *C167 Derivatives User Manual*, 2.0 edition, March 1996. Dieses Handbuch ist auf der Web-Seite des Herstellers oder auf der CD unter Anleitung/HerstellerDokus/c16x/C167_Derivatives.pdf zu finden.

[26] Siemens/Infineon. *Instruction Set Manual for the C16x Family of Siemens 16-Bit CMOS Single-Chip Microcontrollers*, 1.2 edition, December 1997. Dieses Handbuch ist auf der Web-Seite des Herstellers oder auf der CD unter Anleitung/HerstellerDokus/c16x/Instruction_Set_Manual_for_C16x.pdf zu finden.

Hiermit versichere ich diese Arbeit selbständig verfaßt und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Christian Asam