Technische Universität Darmstadt
Fachbereich Informatik
Prof. Dr.-Ing. S. A. Huss

Entwicklung eines Mechatronik-Frameworks für das Projekt Läufer

Christian Rüdiger 954596
Christoph Reichenbach 861922
Markus Weimer 946371

8. Januar 2003

Verantwortlichkeit der Autoren für die Einzelnen Kapitel:

Markus Weimer Kapitel 1,2, 3 und 6

Christoph Reichenbach Kapitel 4

Christian Rüdiger Kapitel 5, die Appendices

Die vorliegende Arbeit wurde im Rahmen des Projektes Läufer erstellt. Weitergehende Informationen zum Projekt finden sich auf dessen Homepage:
http://www.projekt-laeufer.de

Zusicherung

Ich versichere, meine oben genannten Anteile an dieser Arbeit ohne unzulässige fremde Hilfe und nur unter Verwendung der in der Arbeit angegebenen Literatur und Hilfsmittel angefertigt zu haben.

Darmstadt, den 8. Januar 2003 Markus Weimer

Ich versichere, meine oben genannten Anteile an dieser Arbeit ohne unzulässige fremde Hilfe und nur unter Verwendung der in der Arbeit angegebenen Literatur und Hilfsmittel angefertigt zu haben.

Darmstadt, den 8. Januar 2003 Christian Rüdiger

Ich versichere, meine oben genannten Anteile an dieser Arbeit ohne unzulässige fremde Hilfe und nur unter Verwendung der in der Arbeit angegebenen Literatur und Hilfsmittel angefertigt zu haben.

Darmstadt, den 8. Januar 2003 Christoph Reichenbach

Zusammenfassung

Im Rahmen des Projektes Läufer wurde mit dieser Arbeit ein mechatronisches System für den Läufer entwickelt. Dieses System schliesst drei Komponenten ein: Das Fahrerinformationssystem, eine Platine samt Software für die Entwicklung mechatronischer Systeme sowie eine auf CAN-Bus basierende Kommunikationsstruktur.

Die vorliegende Arbeit beschreibt die dabei getroffenen Design- und Produktentscheidungen vor dem Projektkontext. Im ersten Kapitel soll zu diesem Zweck zunächst ein Überblick über das Projekt Läufer gegeben werden. Darauf folgt im zweiten Kapitel eine sich aus dem Projekt ergebende Anforderungsdarstellund an die Mechatronik. Diese beiden Kapitel wurden in enger Zusammenarbeit mit den Projektleitern des Läufers erstellt.

Die folgenden drei Kapitel beschreiben die drei Komponenten des mechatronischen Systems. Dabei wird von der Anwenderschnittstelle über die Kommunikation der Geräte bis zu den Geräten selbst vorgegangen.

Im letzten Kapitel wird dann eine kurze Zusammenfassung und ein Ausblick gegeben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Das Projekt Läufer

von: Markus Weimer

Inhaltsangabe

Diese Kapitel soll einen Überblick über den Projektkontext dieser Arbeit und den sich daraus ergebenden Anforderungen geben.

1.1 Überblick

Das Projekt Läufer wurde im Sommer 1998 von Maschinenbaustudenten der Technischen Universität Darmstadt und Industriedesignstudenten der Fachhochschule Darmstadt am Fachgebiet Maschinenelemente und Konstruktionslehre (MuK) der Technischen Universität Darmstadt ins Leben gerufen.

Ziele der beteiligten Studenten, Lehrenden und Industriepartner sind die Entwicklung und Realisierung eines zweisitzigen Reisefahrzeugs, das überwiegend durch Muskelkraft angetrieben wird und mit dem durchschnittlich trainierte Menschen eine Reisegeschwindigkeit von 40 km/h erreichen können. Das interdisziplinäre Projekt will ein innovatives Produkt realisieren und eine zeitgemäße, offene Form des Studiums umsetzen. Die Motivation für die Beteiligten ergibt sich hierbei aus dem Projektziel, und nicht aus dem Leistungsnachweis.

Das studentische Team organisiert sich bezüglich der Arbeitsverteilung, dem internen Informationsmanagement und der Akquisition von unterstützenden Unternehmen und Universitätsinstituten selbst. Gerade letzteres zeigt, daß die grundlegenden Ideen dieses Projekts von vielen begeistert aufgenommen und unterstützt werden.

Abbildung 1.1: Die Projektleiter im Gespräch

1.2 Randbedingungen

Das Projekt Läufer wird von Studenten neben dem Studium geleitet und ausgeführt. Hieraus entstehen Randbedingungen, die die Arbeit am Projekt nicht unwesentlich beeinflussen. Im folgenden sollen auf zwei aus Sicht der Universitäten besonders interessante Randbedingungen eingegangen werden: Das Projektstudium sowie die verteilte Produktentwicklung.

1.2.1 Projektstudium

Das Projekt Läufer wurde als studentisches Pilotprojekt am Fachgebiet Maschinenelemente und Konstruktion (kurz: MuK) an der TU Darmstadt ins Leben gerufen. Professoren und wissenschaftliche Mitarbeiter, die den Studierenden sonst als Lehrende gegenübertreten, nehmen im Projekt die Rolle eines Teamcoaches ein. Sie lernen dadurch ihrerseits, auch einmal weniger zu wissen als ihre Studenten.

Die Studenten arbeiten in eigener Verantwortung auf ein selbst gestecktes Ziel hin. Dazu erschließen sie sich Fachwissen, organisieren Zusammenarbeit, managen Informationen, aquirieren Know-How und gewinnen die Unterstützung externer Partner.

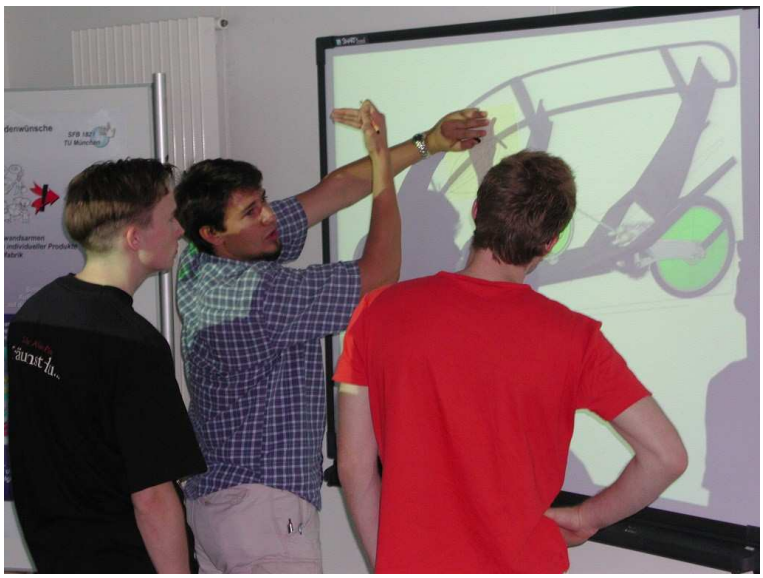

Abbildung 1.2: Diskussion vor einem sog. Smartboard

1.2.2 Verteilte Produktentwicklung

Im Projekt Läufer werden Einsatzmöglichkeiten der modernen Informations- und Kommunikationstechnologie untersucht, und es wird aufgezeigt, welche neuen Perspektiven und Potentiale bestehen, um Vorgehensweisen und Prozesse der Produktentwicklung verteilt an mehreren Standorten durchzuführen.

In einer Zusammenarbeit mit dem Lehrstuhl für Produktentwicklung an der Technischen Universität München (Prof. Lindemann) werden sowohl organisatorische und methodische Voraussetzungen für eine erfolgreiche Gestaltung verteilter Entwicklungsprozesse als auch Möglichkeiten zur technischen Durchführung solcher Prozesse herausgearbeitet. Das studentische Team in München ist für die Entwicklung der peripheren Komponenten verantwortlich. Diese sollen das in dieser Arbeit entworfene Kommunikations- und Steuerungssystem des Läufers einsetzen.

1.3 Schwerpunkte

Ein Projekt wie der Läufer setzt im Rahmen seiner Entstehung und seiner Entwicklung Schwerpunkte, die es von anderen unterscheiden und es einzigartig machnen. Das Projekt Läufer setzt die folgenden Schwerpunkte:

- Design
- Antrieb
- Aerodynamik
- Ergonomie
- Fahrwerk
- Peripherie

Im folgenden sollen diese Schwerpunkte kurz vorgestellt werden.

1.3.1 Design

Das Design des Läufers erzeugt ein Spannungsfeld zwischen den dynamischen Linien von Tragstruktur und Fahrwerk und der weichen Linienführung von Dach und Verkleidung. Die Form folgt der Funktion und mehr noch, sie unterstreicht die Funktion. Bedienelemente und mechanische Knotenpunkte werden bewußt betont. Die dominanten Tretringe identifizieren den Läufer als muskelgetriebenes Fahrzeug.

Maßgeblich für das Design ist die Abkehr vom modularen System „Fahrrad". Die Entscheidung gegen eine fahrradtypische Gesamtstruktur ermöglicht die gestalterische Integration der umfangreichen Funktionalitäten und führt letztlich auch zu einer Reduzierung der Teilevielfalt.

Die Farbkombination aus hellgrün, schwarz und silber symbolisiert die Realisierung eines ökologisch vertretbaren Produktkonzepts mit Hilfe einer hochwertigen, industriellen Fertigung und zeigt, dass beides einander nicht ausschließen muss.

Abbildung 1.3 zeigt die Hinterradkonstruktion sowie die ungewöhnlichen Tretringe in einer 3D-CAD[1] Ansicht.

[1] CAD = Computer Aided Design

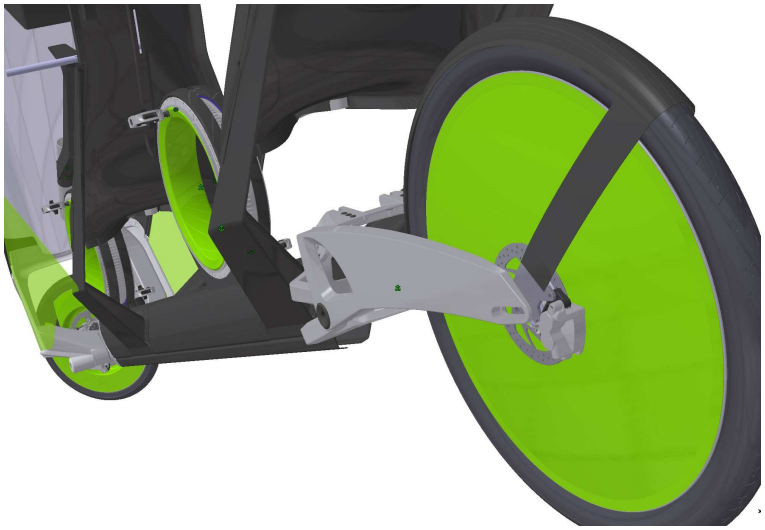

Abbildung 1.3: Designdetails des Läufer

1.3.2 Antrieb

Mit herkömmlichen Tandemfahrrädern erreichen durchschnittlich trainierte Personen eine Durchschnittsgeschwindigkeit von rund 35 km/h. Unter gleichen Bedingungen strebt das Projekt 40 km/h an. Nicht als rechnerisch ermittelten Wert, sondern als konstant gehaltene Reisegeschwindigkeit, realisiert durch einen leistungsverzweigten Hybridantrieb. Von den Tretringen läuft ein Zahnriehmen in ein Zwischengetriebe. Von hier treibt ein zweiter die Hinterradnabe an. Die Speedhub von Rohloff ist eine 14-stufige Schaltungsnabe mit einer Spreizung von 500%, die damit den gewünschten Geschwindigkeitsbereich abdeckt.

Abbildung 1.4 zeigt eine aus dem verwendeten CAD-System gewonnene Schemazeichnung des Antriebsstrangs.

Das Zwischengetriebe, ein Planetengetriebe, verfügt über drei elektronisch geregelte Elektromaschinen als Energiewandler. Die Regelung ermöglicht eine Überlagerung von Drehzahlen im Planetengetriebe mit dem Effekt, daß die Schaltstufen der Hinterradnabe ausgeglichen werden: Eine stufenlose Schaltung mit im wesentlichen mechanischer Übertragung! Außerdem werden die Elektromaschinen als Generatoren betrieben, um etwa Bremsenergie zurück zu gewinnen.

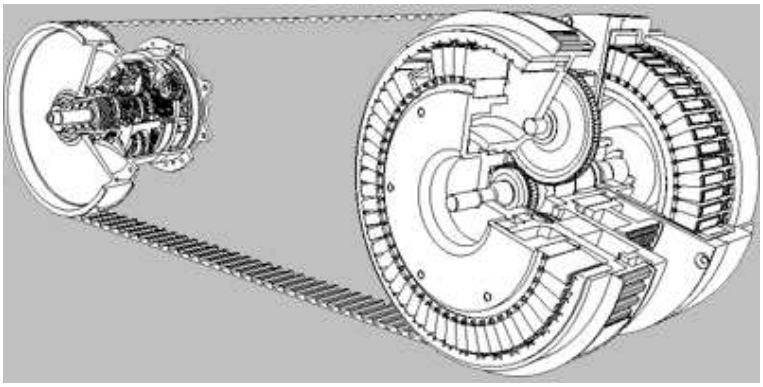

Abbildung 1.4: Antriebsstrang des Läufers

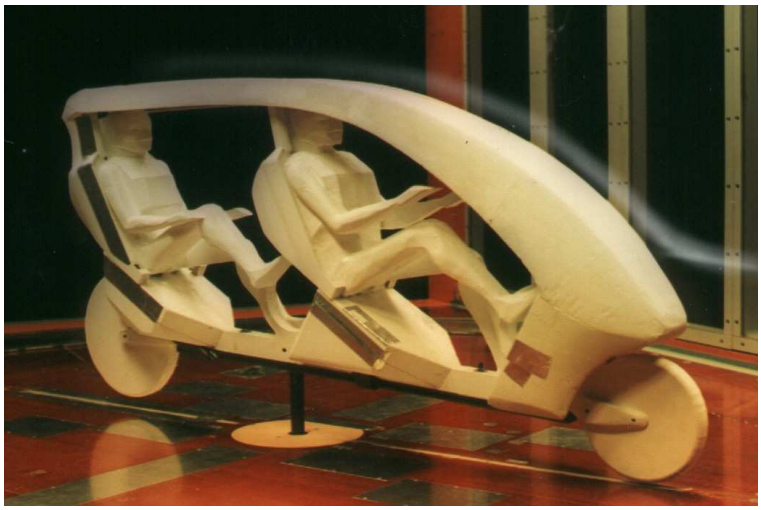

Abbildung 1.5: Ein Modell des Läufers im Windkanal

1.3.3 Aerodynamik

Die Effektivität muskelgetriebener Fahrzeuge kann durch aerodynamische Verbesserungen massiv gesteigert werden. Der Luftwiderstand ist in der dritten Potenz abhängig von der Geschwindigkeit. Das bedeutet, daß er bereits bei 25 km/h weit größer ist als die übrigen Fahrwiderstände wie Lagerreibung und Rollreibung der Reifen. Der Luftwiderstand ist das Produkt aus der Stirnfläche und dem Luftwiderstandsbeiwert cw. Ein herkömmliches Fahrrad hat etwa einen cw-Wert von 1,0, ein Tandem 1,1 und ein Liegerad etwa 0,8. Geschlossene Liegedreiräder erreichen dagegen einen cw-Wert von bis zu 0,11.

Dieses Potential muß mit den Anforderungen eines Reisefahrzeuges

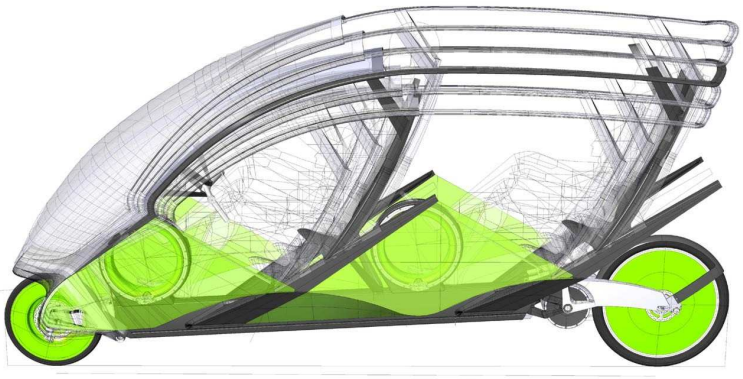

Abbildung 1.6: Die Verstellbarkeit des Läufers

in Einklang gebracht werden. Eine Teilverkleidung mit großen seitlichen Öffnungen läßt die Fahrer ihre Umwelt näher erleben. Aus der Sicht der Aerodynamik steht dies im Widerspruch zu einer optimalen Strömungsführung um das Fahrzeug. Aufgabe der Optimierungen im Windkanal der TU Darmstadt (siehe Abbildung 1.5 und der anschließenden numerischen Berechnungen mit der Simulationssoftware Powerflow muß es also sein, unter Beibehaltung der äußeren Erscheinung eine Verringerung des Fahrwiderstandes und ein wettergeschütztes Reisen zu ermöglichen.

1.3.4 Ergonomie

Bei einsitzigen Fahrzeugen kann auf eine Größenanpassung des Läufers gegebenenfalls verzichtet werden. Hochwertige Fahrradrahmen gibt es in vielen verschiedenen Größen, einige werden auf Maß gefertigt. Bei einem Tandemfahrzeug ist eine flexible Größenanpassung der Fahrerpositionen unabdingbar, weil hier die Maße der beiden Fahrer auch in Relation zueinander berücksichtigt werden müssen. Außerdem wird ein Reisefahrzeug in vielen Fällen nicht immer von den selben Personen genutzt.

Als hoch belastete Baugruppe bleibt der Antriebsstrang fest mit der tragenden Struktur verbunden. Stattdessen werden die Sitze verschoben. Die Sitzverstellung entlang einer Schrägen sorgt dafür, daß für jeden Fahrer nicht nur der richtige Abstand zu den Pedalen, sondern auch die richtige Sitzhöhe über Boden gewährleistet ist (Siehe Abbildung 1.6). So können Fahrer jeder Größe im Sitzen mit beiden Füße den Boden erreichen und das stehende Fahrzeug stabilisieren.

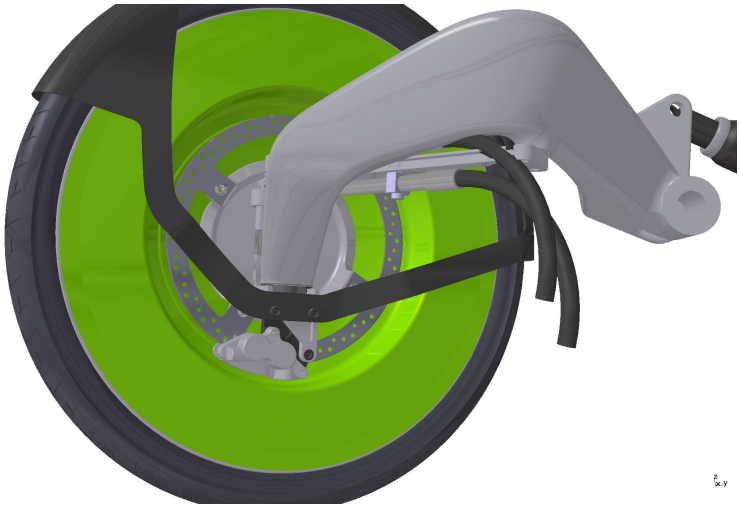

Abbildung 1.7: CAD-Konstruktion der Vorderradschwinge

1.3.5 Fahrwerk

Die Tragstruktur besteht aus selbsttragenden CFK-Hohlkörpern, die aus Halbschalen zusammengefügt werden. Auf diese Weise können die Grundelemente ohne große Nacharbeit mit Hilfe einfacher Werkzeugformen gefertigt werden. Die Laschen, entlang derer die Schalen verklebt werden, erfüllen zusätzliche Funktionen, etwa als Systemschiene für die Sitzverstellung oder als Befestigungslasche für Verkleidungsteile. Alle Komponenten des Hauptrahmens werden miteinander fest verklebt.

Aus den gleichen Bauteilen lassen sich Rahmen sowohl für den Einals auch für den Zweisitzer aufbauen. Die Kohlefaserteile weisen im Vergleich zu herkömmlichen Liegeradrahmen große eingeschlossene Volumina auf. Dadurch wird dem vergleichsweise langen Fahrzeug ausreichend Steifigkeit verliehen.

Um eine einfache Verstellbarkeit des Cockpits zu garantieren, wird die Lenkung mit einem flexiblen Aktorsystem betätigt (siehe dazu auch Abbildung 1.7.

1.3.6 Peripherie

Der Anspruch, ein reisetaugliches Fahrzeug zu entwickeln, steht und fällt mit der Qualität sämtlicher Teilfunktionalitäten. Bei der Entwicklung des Läufers wird daher besonderer Wert darauf gelegt, daß innovative Ansätze in Antrieb und Ergonomie in allen peripheren Komponenten ihre Fortsetzung finden.

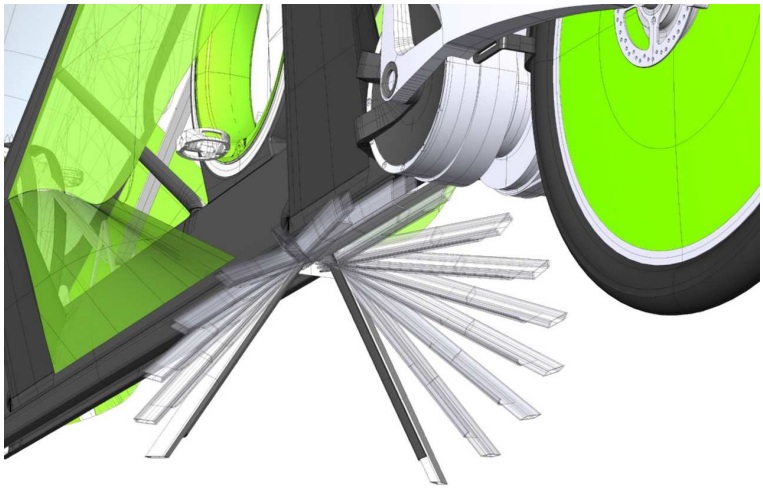

Abbildung 1.8: Der Mechatronische Ständer in einer Funktionsanalyse

In dieser Philosophie entstehen ein mechatronischer Fahrzeugständer (siehe Abbildung 1.8), ein intelligentes Beleuchtungssystem, ein System zur Scheibenreinigung, Gepäckraum und Verkleidungsteile. Dabei verfolgen die Studenten den Anspruch, alle Komponenten in das Design des Läufers zu integrieren: Endlich ein Reiserad ohne Schlauchschellen, Klebeband und Kabelbinder.

Kapitel 2

Aufgabenstellung

von: Markus Weimer

Inhaltsangabe

2.1 Problemstellung

Nach den im vorangegangenen Kapitel beschriebenen großen Erfolgen des Projektes im Bereich der bisherigen Schwerpunkte stellte sich für das Projekt die Frage nach der Steuerung der vielen geplanten Komponenten des Läufers.

Steuerung meint in diesem Zusammenhang sowohl die automatische Steuerung und Regelung der Komponenten als auch die Einflußnahme des Fahrers auf diesen Regelungsprozess. Die Einflußnahme des Fahrers kann z.B. durch das Betätigen von Schaltern und Knöpfen, aber auch durch die Art und Weise, wie er in die Pedale tritt, zum Ausdruck kommen. Das zu entwickelnde System muß also sowohl offensichtliche Interaktion mit dem Fahrer als auch eine umfassende Sensorik unterstützen.

Die Problemstellung des Projektes geht jedoch über diese Anforderungen an die Funktionalität eines zu entwickelnden Systems hinaus. Im Projekt sind zusätzliche Randbedingungen zu beachten. So stand zu Beginn dieser Arbeit noch nicht fest, welche Komponenten denn nun tatsächlich ihren Weg in die Produktion der geplanten zehn Fahrzeuge finden wird. Auch war nicht auszuschließen, daß sich zu den bereits angedachten Systemen weitere gesellen werden. Eine solche Lage fordert eine große Flexibilität des im Rahmen dieser Arbeit entworfenen und entwickelten Systems.

Eine weitere spezifische Problemstellung ist die Zusammenarbeit der verschiedenen Fachrichtungen. Im Bereich der Zusammenarbeit zwischen Maschinenbau und Informatik reicht es eben nicht, eine gemeinsame Sprache zu finden, wie in [Ort02] für die Zusammenarbeit zwischen Wirtschaftswissenschaftlern und der Informatik gefordert.

Vielmehr ist es im Rahmen einer solchen Zusammenarbeit zusätzlich erforderlich, eine Schnittstelle zu definieren. Diese Schnittstelle wird von der Informatik zur Verfügung gestellt und auf dieser Schnittstelle können dann die anderen beteiligten Fachgebiete ihr individuelles Wissen einbringen. So ist es wohl wenig sinnvoll, wenn ein Informatiker die Entscheidungen über die zukünftige Fahrdynamik des Läufers trifft. Der Maschinenbauer, bei dem diese Entscheidung natürlich sinnvoll angesiedelt ist, benötigt zur Umsetzung solcher Entscheidungen allerdings Zugriff auf die wesentlichen Systeme des Läufers.

Vor dem Hintergrund all dieser Überlegungen ist in enger Zusammenarbeit mit den anderen am Projekt beteiligten der folgende Entwicklungsauftrag entworfen worden.

2.2 Entwicklungsauftrag

Die im vorangegangenen Abschnitt beschriebene Problemstellung läßt sich am sinnvollsten mittels eines mechatronischen Systems bearbeiten. [DEF MECHATRONIK]

Nur mit Hilfe der programmierbaren Elektronik läßt sich die große Flexibilität und Erweiterbarkeit praktisch umsetzen. Auch bietet eine Mechatronik die Möglichkeit, eine (Software-)Schnittstelle anzubieten, auf der die anderen Projektteilnehmer dann arbeiten können.

Eine Gliederung des Auftrags hat die folgenden Arbeitsgebiete ergeben, die auch in Abbildung 2.1 noch einmal gezeigt werden:

Fahrerinterface Der Läufer wird wie im vorangegangenen Kapitel beschrieben über ein sehr weit entwickeltes Antriebssystem verfügen, daß dem Benutzer angepaßt werden kann. Diese Anpassung soll in einem bestimmten Rahmen auch durch den Nutzer während des Betriebes möglich sein. Der Läufer braucht folglich eine Möglichkeit, den Fahrer mit dem Antriebssystem interagieren zu lassen. Diese Möglichkeit kann man dann auch direkt nutzen, um alle weiteren Informationspräsentationen für und Interaktionen mit dem Fahrer zu realisieren. Details zur Implementierung dieses im weiteren ,,Fahrerinterface" genannten Systems finden sich in Kapitel 3.

Kommunikation der mechatronischen Komponenten Die verschiedenen mechatronischen Komponenten des Läufers sollen kommunizieren können. Zum einen mit dem Fahrer über das Fahrerinterface, aber zum anderen auch untereinander, um z.B. Gefahrensituationen erkennen zu können. Details zur Kommunikation innerhalb des Läufers werden in Kapitel 4 näher ausgeführt.

Mechatronische Komponenten Im Bereich der mechatronischen Komponenten soll den Entwicklern aus dem Projekt ein Rahmen geschaffen werden, den sie dann ausfüllen können. Dieser Rahmen ist eine Platine, die allen möglichen Einsatzszenarien innerhalb des Läufers gerecht wird. Ergänzt wird die Entwicklung der Platine durch durch die Bereitstellung einer Code-Bibliothek. durch die der Zugriff auf die Funktionen der Platine weitestgehend vereinfacht wird. Diesen Rahmen aus Soft- und Hardware können die anderen Projektteilnehmer dann mit Ihrem spezifischen Wissen ausfüllen, indem sie z.B. Eingaben aus den Sensoren mit Aktionen der Platine verknüpfen. Details zu diesem Teil des mechatronischen Systems werden in Kapitel 5 beleuchtet.

Zusammengefaßt läßt sich somit sagen, daß eine Art Framework für die spätere Entwicklung des mechatronischen Systems des Läufers erstellt werden soll. Dies folgt der guten Tradition des Projektes, an sinnvollen Stellen klare Schnittstellen zwischen den einzelnen Projektgruppen zu definieren.

Im Falle der Mechatronik läßt sich dies sehr gut veranschaulichen: Es ist sicherlich sinnvoll, die Entwicklung und Programmierung der Kommunikationsprotokolle durch Informatiker durchführen zu lassen. Es macht allerdings keinen Sinn, den selben Informatiker auch damit zu betrauen, *konkrete* Meldungen über z.B. die aktuelle Geschwindigkeit, zu interpretieren. Dieses Wissen um die Fahrdynamik des Läufers ist die Domäne der Maschinenbauer, denen eine geeignete Schnittstelle angeboten werden muß, um darauf aufsetzend die Fahrzeuglogik zu entwickeln. Ein weiteres Beispiel ist das Fahrerinterface: Eines der Kernmitglieder des Projektes ist Designer, so daß es nur natürlich ist, bei der Gestaltung des Benutzerinterfaces auf seine Expertise zurückzugreifen. Nur muß auch dort erst einmal der Punkt erreicht werden, an dem er beginnen kann zu arbeiten.

Das Erreichen dieses Punktes, ab dem die beteiligten Personen aus dem Läufer-Projekt arbeiten können, ist das Hauptziel dieser Arbeit.[1]

2.3 Rahmenbedingungen

Die Rahmenbedingungen der Entwicklung des mechatronischen Systems für den Läufer ergeben sich aus den Zielen des Projektes insgesamt sowie dem geplanten weiteren Projektverlauf. Im einzelnen sind dies die folgenden Anforderungen:

Design: Der Läufer lebt von seinem außergewöhnlichen Design. In dieses muß sich folglich eine Mechatronik mit all ihren Komponenten integrieren. Dies hat offensichtliche Implikationen beim Fahrer-Interface aber auch weitaus subtilere bei der Wahl des Kommunikationssystems und dessen Verkabelung. Denn dicke Kabelbäume, die für den Benutzer sichtbar im Fahrzeug verlegt sind, sind im Läufer ganz offensichtlich fehl am Platz.

Energieeffizienz: In einem Fahrzeug, dessen gesamtes Energiesystem aus den Muskeln der Fahrer gespeist wird, ist der sinnvolle Umgang mit

[1] Eine kurze Einführung in die Interdisziplinäre Arbeit des Teams findet sich u.A. in [And02]

der Energie natürlich sehr wichtig. Auch dies hat Implikationen im gesamten Mechatronik-System, da jedes elektronische System elektrische Energie verbraucht.

Attraktivität für Industriepartner: Das Projekt Läufer finanziert sich zu weiten Teilen aus Industriepartnern. Um das vom Projekt angestrebte Ziel einer Kleinserie von zehn Fahrzeugen erreichen zu können, muß dies auch für die mechatronischen Komponenten gelten. Auch wenn diese im Vergleich zu den mechanischen Komponenten des Läufers recht preiswert sind, übersteigt ihr Wert doch mit Sicherheit die normalen Finanzmittel eines studentischen Projektes. Bei den Entscheidungen, die im Rahmen der Entwicklung der Mechatronik zu treffen sind, muß dieser Gesichtspunkt also ebenfalls berücksichtigt werden.

Verfügbarkeit: Die verwendeten Produkte und Tools sollen auf jeden Fall in großen Stückzahlen verfügbar sein, da sie direkt in die Fertigung der Nullserie übernommen werden sollen.

Handhabbarkeit durch Ingenieure: Das entwickelte Mechatronik-Framework soll durch die Ingenieure des Projektes Läufer genutzt werden, um ihre mechatronischen Anwendungen zu realisieren. Bei den Schnittstellen, die diese Entwickler verwenden sollen, ist also darauf zu achten, sie verständlich und kompakt zu halten. Außerdem ist eine regelmäßige Rücksprache mit den Ingenieuren geboten, um ihre Anforderungen möglichst exakt zu erfüllen.

Erweiterbarkeit: Zum Zeitpunkt des Erstellens dieser Arbeit steht noch nicht fest, welche mechatronischen Systeme der Läufer erhalten soll. Es muß also sichergestellt sein, daß das entwickelte Framework erweiterbar ausgelegt ist.

Sicherheit und Fehlertoleranz: Diese Anforderung ist nahezu selbsterklärend. Beim mechatronischen System eines Fahrzeugs spielt die Sicherheit natürlich eine sehr große Rolle. Gemeint ist hier nicht unbedingt die Sicherheit vor Manipulation durch dritte, wie in anderen Bereichen der Informatik, sondern vielmehr ein durchgängiges Failsafe Design in allen Komponenten.

Fehler dürfen folglich nicht als Ausnahme betrachtet werden, deren gelegentliches auftreten man verschmerzen kann, sondern ihr Auftreten muß eingeplant werden und alle denkbaren Fehler müssen sicher abgefangen werden können. Dies gilt auch dann, wenn die

gesamte Mechatronik ausfallen sollte. Es muß infolgedessen ein mechanisches Notsystem für die wichtigsten Funktionen vorhanden sein. Ein solches System ist aber sehr Projekt-spezifisch und übersteigt diese Studienarbeit. Vielmehr ist es Aufgabe der Entwickler der Mechatronik, also der Autoren dieser Arbeit, die Ingenieure auf die verbleibenden Fehlermöglichkeiten aufmerksam zu machen, damit diese geeignete Notfallsysteme entwickeln können.

2.4 Aufteilung der Gesamtaufgabe in Einzelaufgaben

Die weiter oben schon beschriebene und auch in Abbildung 2.1 dargestellte Einteilung des Auftrags folgt nicht nur technischen Überlegungen. Ebenfalls berücksichtigt wurde eine sinnvolle Aufteilung der Arbeit auf die drei Autoren.

Besonders wichtig ist in diesem Zusammenhang der Zuschnitt des Bereichs „Kommunikation"; Dieser wurde so gewählt, obwohl damit eine Implementierung auf beiden Seiten (Platine und PDA) einhergeht. Grund hierfür ist, daß eine Entwicklung von Kommunikation nur in einer Hand funktionieren kann. Zu dieser Überlegung führten intensive Gespräche auch mit unseren Industriepartnern. Eine Aufteilung dieses Bereichs anhand der Kommunikationsteilnehmer bereitet viel größeren Aufwand, da sich die auf beiden Seiten beteiligten viel häufiger abgleichen müssen. Zusätzlich steigt die Gefahr von Missverständnissen.

Die Bereiche wurden schliesslich unter den Autoren wie folgt verteilt:

Fahrerinterface Markus Weimer

Kommunikation Christoph Reichenbach

Mechatronische Komponenten Christian Rüdiger

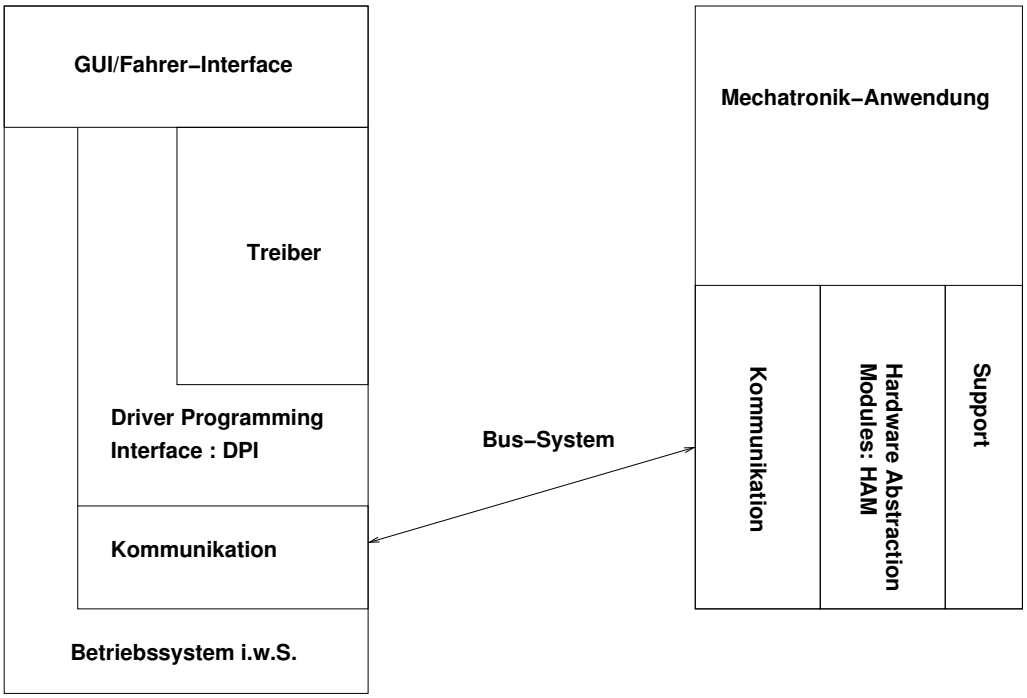

Abbildung 2.1: Schema des zu entwickelnden Gesamtsystems

Kapitel 3

Fahrerinterface

von: Markus Weimer

Inhaltsangabe

3.1 Aufgabenstellung

Abbildung 3.1 zeigt eine Seitenansicht des Läufers. Dabei wurden die derzeit geplanten mechatronischen Komponenten markiert.
Die einzelnen Komponenten sind:

1. Antriebsstrang
2. Beleuchtungssystem
3. Fahrerinformations-System
4. Ausfahrbare Fahrzeugstütze

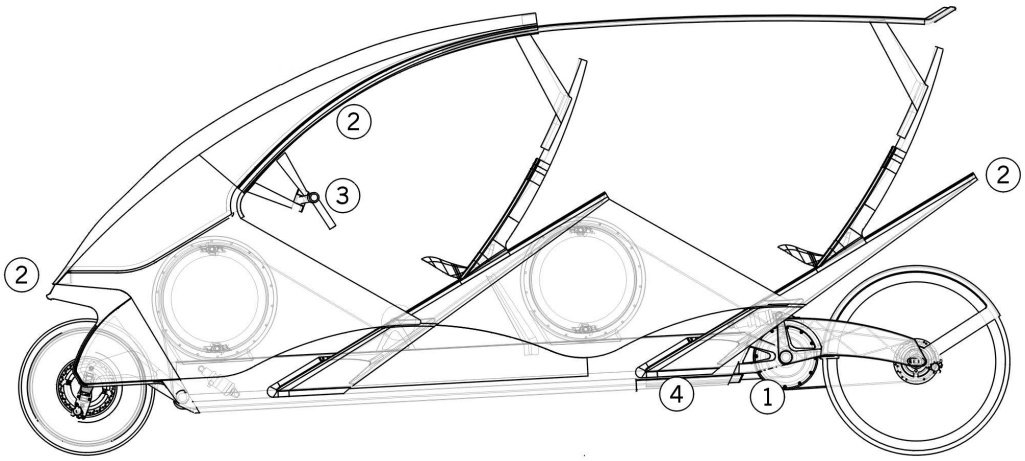

Abbildung 3.1: Seitenansicht des Läufers

Für die Zukunft sind weitere Komponenten möglich und auch eingeplant. Konkret angedacht wurde z.B. eine Anbindung eines GPS[1]-Systems über den CAN[2]-Bus. Die mechatronischen Komponenten des Läufers verfügen über etliche Parameter, die der Fahrer beeinflussen möchte. Als Beispiel sei hier der Antriebsstrang genannt. Es soll dem Fahrer z.B. ermöglicht werden, bei vollen Batterien und kurzer Strecke alleine elektrisch zu fahren, und die Pedale ohne Widerstand zu benutzen.

Außerdem gibt es Informationen über den aktuellen Zustand des Fahrzeuges, die an den Fahrer weitergeleitet werden müssen. Dies sind zum einen Daten, wie man Sie auch vom normalen PKW kennt, wie z.B. die

[1]Global Positioning System
[2]Controller Area Network, vgl. Kapitel 4

aktuelle Geschwindigkeit, zum anderen kommen beim Läufer spezielle Informationen aus dem Antriebsstrang hinzu: Wie schnell kann man bei der aktuell in den Pedalen abgegebenen Energie fahren? Oder, wie voll sind die Batterien? Die beschriebenen Informationen müssen dem Fahrer in einer Art und Weise zur Verfügung gestellt werden können, die der Fahrsituation angemessen ist.

Zusätzlich ergaben Gespräche mit den Entwicklern mechatronischer Komponenten, daß aufwendige Berechnungen, die sich aus den Wünschen des Fahrers ergeben, in einer relativ komfortablen Hochsprache wie C/C++ programmierbar sein sollten, anstatt vergleichsweise umständlich in Assembler auf den Embedded Platinen.

Es ergeben sich somit die folgenden Anforderungen an das Fahrerinformationssystem des Läufers:

Genügend Rechenleistung: Die geforderten Berechnungen sollen komfortabel in Hochsprachen durchgeführt werden können.

Display: Die Informationen für den Fahrer müssen dargestellt werden. Aufgabe des Displays ist es, das Armaturenbrett eines PKW zu ersetzen.

Interaktion: Um die teilweise sehr umfassenden Einstellungsmöglichkeiten des Fahrzeugs ausnutzen zu können.

Die beschriebenen Aufgaben lassen sich in hervorragender Weise von aktuellen Taschencomputern, sogenannten PDAs[3] erfüllen. Exemplare der Gerätekategorie PDA, insbesondere die neuerer Bauart, verfügen über genügend Rechenleistung, um die Aufgaben innerhalb des Läufers bewältigen zu können. Zusätzlich sind die mobilen Kleinrechner sehr stark auf den sorgsamen Umgang mit Energie optimiert, was in einem Fahrzeug, daß seine Energie ausschließlich aus der Muskelkraft seiner Fahrer bezieht, natürlich von besonderem Interesse ist. Außerdem sind sie in großen Stückzahlen kommerziell verfügbar, was den Preis für den einzelnen PDA stetig sinken läßt.

Aufgabe im Bereich des Fahrerinformationssystems ist es also einerseits, einen geeigneten PDA auszuwählen und den gewählten andererseits durch das Erstellen zusätzlicher Software für das Projekt nutzbar zu machen.

Die zu erstellende Software ist vor allem ein Kommunikationssystem zwischen PDA und den im Fahrzeug verteilten Komponenten. Dies ist

[3]PDA: Personal Digital Assistent, zu Deutsch etwa: Persönlicher Digitaler Assistent.

nötig, da im PDA-Bereich zwar die aus dem PC-Markt bekannten Schnittstellen und Kommunikationsprotokolle zur Verfügung stehen[4]. Im Bereich der mechatronischen Systeme stehen die aus der PC-Technik stammenden Schnittstellen jedoch im Allgemeinen nicht zur Verfügung. Vielmehr dominieren Protokolle wie CAN und SPI[5].

Für den Entwickler einer mechatronischen Komponente des Läufers muß die Kommunikation möglichst einfach zu handhaben sein. Eine solche Forderung ergibt sich aus den Überlegungen in Kapitel 2.1. Die entwickelte Software muß diesem Umstand Rechnung tragen, indem sie dem Entwickler eine möglichst einfache und intuitive Programmierschnittstelle zur Verfügung stellt.

Neben den bisher erläuterten Anforderungen der Entwickler mechatronischer Komponenten soll aber gerade bei der Auswahl des PDAs der spätere Nutzer des Läufers und seine Wünsche Berücksichtigung finden. Der potentielle Läuferfahrer bzw. die potentielle Fahrerin wünscht sich von einem solchen Gerät einen Mehrfachnutzen. Es soll also sowohl als „Bordcomputer" des Läufers fungieren können, als auch ein vollwertiger PDA sein.

Im folgenden soll zunächst die Entscheidung für einen speziellen PDA transparent gemacht werden. Daraufhin wird die Kommunikationsschnittstelle für die Entwickler mechatronischer Komponenten für den Läufer beschrieben.

Im Bereich der Projektarbeit ergeben sich natürlich auch Anforderungen, die sich nicht direkt aus theoretischen Erwägungen ableiten lassen können. Viele Anforderungen werden erst und immer wieder im Gespräch mit den Projektteilnehmern aus anderen Fachgebieten, wie z.B. dem Maschinenbau oder dem Industriedesign, deutlich. Ergebnisse dieser Anforderungen sind unter anderem der so genannte DebugTreiber, auf den am Ende des aktuellen Kapitels exemplarisch für die interdisziplinäre Zusammenarbeit innerhalb des Projektes eingegangen werden soll.

[4]Beispiele hierfür sind die serielle Schnittstelle, die Infrarottechnik IrDA sowie USB
[5]siehe Kapitel 5

3.2 Auswahl des PDA

3.2.1 Anforderungen

Die Anforderungen an den PDA ergeben sich aus den Anforderungen an die gesamte Mechatronik sowie an das Fahrerinformationssystem im speziellen. Sie lassen sich in Anforderungen an die PDA Systemsoftware und Anforderungen an die PDA-Hardware gliedern, was im folgenden geschehen soll. Eine Gliederung nach Betriebssystem ist hilfreich, da sich auch der PDA-Markt sehr gut nach der Systemsoftware der PDAs unterteilen läßt. Weiter unten wird dies zur Vorauswahl des PDAs ausgenutzt.

Anforderungen an die Software

Multitasking: Hierbei verstehen wir unter ,,Multitasking" die Fähigkeit des PDA, mehrere Programme unabhängig voneinander auszuführen. Die Anforderung nach Multitasking ergibt sich unmittelbar aus der Forderung an die gesamte Mechatronik, erweiterbar zu sein. Ein PDA ohne Multitasking würde allerdings bei jeder Erweiterung des Systems einen Eingriff in die sicherheitsrelevante Fahrsoftware des Läufers erfordern, was z.B. beim Einfügen eines GPS[6] basierten Navigationssystems nicht wünschenswert ist. Verfügt der PDA hingegen über Multitasking, kann eine Erweiterung auch durch ein weiteres Programm geschehen, daß parallel zu den anderen läuft.

GUI-Flexibilität: Das Projekt Läufer geht in vielen Bereichen einen designorientierten Weg, der nicht unerheblich zu dem großen Erfolg des Läufers in der Öffentlichkeit sowie beim Aufbau von Industrie-Partnerschaften beigetragen hat. Auch im Bereich der graphischen Interaktion mit dem System soll ein designorientierter Weg möglich sein, um den Läufer als ,,rundes Produkt" entwickeln zu können.

Bei der Auswahl eines PDA samt zugehöriger Software ist demgemäß darauf zu achten, daß man im Bereich der Gestaltung des GUI[7] grösstmögliche Freiheiten hat.

Programmierung: Die Programmierung des PDA sollte einfach sein, wobei mit ,,einfach" im Kontext des Läufers gemeint ist, daß die Entwicklung für den PDA sich möglichst wenig von der Entwicklung

[6] GPS=Global Positioning System
[7] GUI=Graphical User Interface, zu Deutsch: Graphische Benutzer Schnittstelle

für einen PC unterscheiden sollte. Das ist sinnvoll und wünschenswert, da die späteren Nutzer unserer Software wahrscheinlich schon über Erfahrungen in der Programmierung von PCs verfügen und deren Lernaufwand ja so niedrig wie möglich zu halten ist.

Außerdem ermöglicht eine solche Vorgehensweise idealerweise die Entwicklung der Software auf einem PC, um sie dann später ohne große Umstände einfach auf den PDA übertragen zu können. Insbesondere vor dem Projekthintergrund und der verteilten Produktentwicklung kann ein so ausgelegtes Vorgehen erhebliche Kosten für die Beschaffung von PDAs für die einzelnen Entwicklerteams in Darmstadt und München einsparen.

Dokumentation und Support: Da die Entwicklung der Software über die vorliegende Studienarbeit hinaus von weiteren Entwicklern fortgesetzt werden soll, ist eine gute Dokumentation und ein umfassender Support der verwendeten Werkzeuge, Bibliotheken und Compiler mehr als wünschenswert.

Aus den selben Gründen ist es nötig, auf eine möglichst „langlebige" Plattform zu setzen, die sich nicht innerhalb des Projektes ständig verändert oder für deren eingesetzte Version man in Zukunft womöglich keine Entwicklungstools mehr erwerben kann.

Anforderungen an die Hardware

An die Hardware eines PDA für den Einsatz in einem Fahrzeug stellen sich besondere Anforderungen, die im folgenden erläutert werden sollen.

Display: Beim Display eines PDA für den Einsatz im Läufer gibt es zwei wesentliche Aspekte zu beachten: Zum einen sollte es sich um ein Farbdisplay handeln und zum anderen sollte es vor allem unter all den Lichtbedingungen, in denen der Läufer unterwegs sein wird, gut ablesbar sein.

Die Forderung nach einem Farbdisplay ergibt sich aus der Design-Orientierung des Projektes sowie der Anforderung nach GUI-Flexibilität. Desweiteren spricht für ein Farbdisplay, daß man mit ihm Warnhinweise und normale Nachrichten des Systems auf dem Display farblich kodieren kann, wie man dies von anderen Geräten kennt. So lassen sich durch Farbe gerade im Fahrbetrieb wichtige Informationen hervorheben, was z.B. beim PKW mit seinen *roten* Warnleuchten ebenfalls geschieht.

Die Anzeigeeigenschaften des Display bei sehr extremen Lichtverhältnissen sind für den Läufer entscheidend. Das Display muß bei allen Fahrsituationen möglichst gut ablesbar sein, um den Fahrer über den aktuellen Zustand seines Fahrzeuges zu informieren. Zu den kritischen Lichtverhältnissen gehört dabei weniger die Nacht, da alle modernen PDAs über eine leistungsfähige Beleuchtung verfügen. Wesentlich wichtiger ist das Verhalten während der Dämmerung und bei starkem Sonnenschein. Beides stellt einige Display-Konzepte bei heute marktüblichen PDAs vor große Probleme.

Als Beispiel sei hier die inverse Hintergrundbeleuchtung der PDAs des Herstellers Palm genannt, die zwar bei absoluter Dunkelheit hervorragende Dienste leistet, bei Dämmerung aber zu Unlesbarkeit des Displays führt.

Optik: Ein ebenfalls nicht zu unterschätzender, wenn auch nur subjektiv zu beurteilender, Bereich ist die optische Erscheinung des PDA. Die Optik spielt aus technischer Sicht natürlich nur eine untergeordnete Rolle, wird jedoch im Projektkontext bedeutsam. Erkennen läßt sich die Bedeutung des Designs für den Läufer an dem Umstand, daß einige Elemente des Läufers nur unter ästhetischen Gesichtspunkten zu Stande gekommen sind. Da sich unsere Arbeit in das Projekt integriert, müssen wir dem Gesichtspunkt Optik ebenfalls Bedeutung zumessen.

Ein objektiv nachprüfbarer Bereich der Optik ist die Bauform. Für den Einsatz im Läufer kommen nur sogenannte Stift-PDAs in Frage, bei denen das Display hochkant steht und die Eingabe über einen Stift erfolgt. Unter den bekannten um am Markt vertretenen Bauformen läßt sich ein Hochkant-Design am besten in das Fahrzeug-Cockpit des Läufers integrieren.

Anschlußmöglichkeiten: Da das Projekt Läufer eines mit außergewöhnlicher Dynamik ist, ist die Anschlussvielfalt an einem PDA im Läufer ein wesentliches Argument, um sich spätere Erweiterungen nicht durch einen zu beschränkten PDA zu verbauen.

Die Mindestanforderung im Schnittstellen-Bereich ist eine RS232[8] Schnittstelle, da über die serielle Schnittstelle die Verbindung zu einer unserer Platinen hergestellt wird. Die Platine am anderen Ende

[8] Im PC-Bereich auch häufig nur „serielle Schnittstelle" genannt.

der RS232-Schnittstelle verbindet dann den PDA mit dem CAN-Bus. Zu Details der Kommunikation im Läufer siehe Kapitel 4.

3.2.2 Auswahl der Systemsoftware

Der PDA Markt läßt sich, wenn man die diversen geschlossenen Systeme außer Acht läßt, nach den Betriebssystemen PalmOS, WindowsCE und Linux aufgliedern. Die ebenfalls noch angebotenen PDAs der Firma PSION finden hier keine Beachtung, da deren Vermarktung und Weiterentwicklung mittlerweile eingestellt wurde. Außerdem paßt deren Tastatur-Orientiertes Design nicht in das Konzept des Cockpit-Entwurfs für den Läufer.

Die Einteilung nach Betriebssystem lieferte eine erste Kategorisierung des PDA-Marktes. Zu jedem System sind relativ viele verschiedene PDAs am Markt, wodurch die Entscheidung für ein bestimmtes Betriebssystem die erste zu treffende war.

Übersicht über PDA-Betriebssysteme

Im folgenden sollen die einzelnen Systeme samt ihrer für das Projekt relevanten Eigenschaften vorgestellt werden, um sich für eins zu entscheiden, daß die software-seitigen Voraussetzungen für einen Einsatz im Läufer mitbringt.

PalmOS: Das System „PalmOS" wurde von der Firma 3Com entwickelt und zunächst ausschließlich auf deren PDAs, den sogenannten „PalmPilots" eingesetzt. Mittlerweile hat es PalmOS zur Marktführerschaft gebracht und ist auch auf den Geräten von Herstellern wie Sony, Handspring und anderen zu finden.

Folglich ist es verständlich, daß als erstes Geräte mit dem System des Marktführers ins Auge gefaßt wurden. Bei den Geräten kann man aufgrund der großen Verbreitung mit einer guten Akzeptanz seitens der Nutzer des Läufers rechnen. Auch wird ein eventueller Kunde mit hoher Wahrscheinlichkeit schon über ein solches Gerät verfügen.

Ein „Palm IIIx" wurde dann auch Basis eines ersten Versuchs der Mechatronik, der im Rahmen einer Studienarbeit von Jörn Schlingensiepen[Sch00] stattfand. Im Rahmen der Arbeit wurde die grundsätzliche Möglichkeit der Anbindung eines solchen Gerätes an den CANBus untersucht und auch bestätigt.

Allerdings verfügt PalmOS in seiner jetzigen Version nicht über Multitasking, was einen Einsatz im Läufer nicht ermöglicht. Die Erweiterbarkeit wird also nicht in dem Maße sicherstellt, wie es im Läufer erforderlich ist.

Im Bereich der Graphischen Gestaltung des Fahrerinterface erwiesen sich die verfügbaren Geräte mit PalmOS als recht eingeschränkt, da sie weder über eine anpaßbare Bibliothek von Oberflächenelementen noch über genügend Rechenleistung verfügen, um die GUI selbst zu „malen". Bei einem System wie PalmOS ist das jedoch kein Design-Fehler, sondern schlägt sich z.B. in der überwiegend konstanten und guten Bedienbarkeit der Software für die PalmOS-PDAs nieder. Die an sich gute Eigenschaft wirkt sich aber vor dem Hintergrund unserer Aufgabe negativ aus.

Die Programmierung eines PalmOS-PDAs erfolgt mit speziellen Tools. Allerdings haben sich Geräte mit PalmOS im Laufe der Zeit so weit verbreitet, daß mehrere alternative Programmierumgebungen und Compiler zur Verfügung stehen. Auch hat die hohe Verbreitung zu einer guten und frei verfügbaren Dokumentation der Programmierschnittstelle und -Praktiken für das Betriebssystem PalmOS geführt.

WindowsCE / Pocket PC: PDAs mit dem System aus dem Hause Microsoft zeichnen sich durch Ihre leistungsfähige Hardware aus. So ist es bei WindowsCE-Geräten nicht selten, einen Prozessor mit deutlich mehr als 200MHz Taktfrequenz vorzufinden.

Auch verfügt Windows CE über Multitasking, eine permanent im Hintergrund laufende Fahrsoftware ist also mit dem System realisierbar.

Die graphische Oberfläche ist sehr an die Desktop-Versionen von Windows angelehnt. Die Flexibilität bei der Gestaltung einer Oberfläche für die Läufersoftware ist auch unter Berücksichtigung der Einschränkung durch die Festlegung auf eine Entwicklungsumgebung schon alleine deshalb groß, weil die Geräte über so große Leistungsreserven verfügen. Im Prinzip wird sogar eine komplett „gemalte" Oberfläche ermöglicht, auch wenn die dann recht viel Speicher verbraucht.

Die Programmierung des Systems erfolgt mittels der bekannten Microsoft-Tools wie z.B. Visual C++, welche schon alleine aufgrund ihrer Verbreitung bestens dokumentiert sind. Allerdings ist WindowsCE genauso wie PalmOS ein reines PDA-System, so daß die

Entwicklung für diese PDAs mit den speziellen Tools erfolgen muß und zum Testen der Software ein Emulator zum Einsatz kommt.

Linux: Linux ist im gesamten PDA und Embedded-Bereich ein Neuling, allerdings mit steigender Verbreitung. Die steigende Verbreitung ist vor allem auf die große Flexibilität des Systems zurückzuführen.

Linux verfügt natürlich auch auf dem PDA über echtes Multitasking. Schließlich kommt der selbe Kernel und die selbe Systemsoftware zum Einsatz wie auf den anderen Plattformen, auf denen es eine Implementierung des freien Betriebssystems gibt.

Die Gleichheit in der Software bringt Linux auf dem PDA eine große Flexibilität, da direkt die gesamte Software, die es für Linux gibt, verfügbar ist. Betroffen hiervon sind insbesondere die Bibliotheken für Entwickler. So existieren z.B. mehrere GUI-Bibliotheken, die noch dazu auf verschiedenen Wegen mit der Hardware kommunizieren können.

Aus der identischen Software auf PC/Workstation und PDA folgt aber auch, daß man Software für den PDA unter Berücksichtigung einiger Randbedingungen auf dem PC entwickeln und testen kann. Dabei kann auf einen Emulator der PDA-Hardware verzichtet werden. Hierdurch wird es ermöglicht, daß Entwickler Software auf Ihrem PC entwickeln und mit diesem testen, die dann später durch ein Neuübersetzen [9] auf den PDA portiert werden kann.

Die große Flexibilität zieht allerdings auch einen entscheidenden Nachteil nach sich: Nahezu jeder Entwickler für Linux-basierte Embedded Systems arbeitet mit seinem eigenen Set an Tools, Bibliotheken und Compilern. Daraus folgt, daß es keine große Entwicklergemeinde gibt, die alle mit den selben Tools arbeiten, was zu einer Unübersichtlichkeit der Dokumentation führt. Die Unübersichtlichkeit wird allerdings durch die Gleichheit des Systems zu dem auf PCs teilweise wieder ausgeglichen.

Entscheidung

Tabelle 3.1 zeigt nochmals eine Übersicht der Systeme mit ihren individuellen Eigenschaften.

PalmOS ist wegen der fehlenden Multitasking-Fähigkeit für unsere Aufgabe ungeeignet. Somit mußte die Entscheidung zwischen Linux und Windows CE getroffen werden.

[9] Diesen Vorgang nennt man im Allgemeinen „crosscompiling"

Anforderung	*PalmOS*	*WindowsCE*	*Linux*
Multitasking	○	●	●
GUI-Flexibilität	○	+	++
Programmierung	○	+	++
Dokumentation und Support	++	+	+

Tabelle 3.1: Übersicht über die verschiedenen PDA-Betriebssysteme. ○: Nicht vorhanden / schlecht; +: Gut; ++: sehr gut

Wir haben uns letztendlich für Linux entschieden, um uns die größte Flexibilität zu sichern. Betroffen hiervon ist nicht nur die oben schon ausgeführte Auswahl an GUI-Bibliotheken, sondern auch die Möglichkeit im Falle der Notwendigkeit selbst am Betriebssystem Hand anlegen zu können. Da wir ein klassisches Embedded System durch einen handelsüblichen PDA ersetzten, war zu Beginn nicht klar, welche Probleme dabei auftreten können. Und mit einem unabänderlichen Betriebssystem wären solche Probleme eventuell nicht lösbar.

Zusätzlich ist es mit einem Linux-PDA möglich, Software auf PCs zu entwickeln und zu testen, ohne daß dazu ein PDA oder gar ein Emulator nötig ist.

3.2.3 Auswahl der Hardware

Im folgenden sollen kurz die PDAs beschrieben werden, die für das Projekt in Frage kamen, und die in folge dessen auch näher geprüft wurden.

Da der Markt für Linux-basierte PDAs sehr in Bewegung ist, kann und soll hier keine vollständige und insbesondere keine aktuelle Übersicht gegeben werden. Ein guter Anlaufpunkt für eine solche ist die Homepage von Linuxdevices[10].

Agenda VR3: Diesem PDA, der in Abbildung 3.2 zu sehen ist, gebührt die Ehre des „first-to-market"-Gerätes im Bereich Linux-PDAs, da es Agenda Computing als erstes gelang, einen endkundentauglichen PDA mit Linux als Betriebssystem in die Läden zu bringen.

Leider gab es zum Zeitpunkt der Entscheidung noch keine Farb-Variante dieses Geräts. Farbe ist aber in einer Design-orientierten Entwicklung wie dem Läufer ein absolutes Muß.

[10] http://www.linuxdevices.com

Abbildung 3.2: Der Agenda VR3

G.Mate Yopy: Abbildung 3.3 zeigt den G.Mate Yopy, der lange Zeit als Mythos durch die diversen News-Seiten und -Foren des Internet geisterte. Ihm wurde am ehesten zugetraut, der erste brauchbare PDA mit Linux als Betriebssystem zu sein. Die Hoffnung der Linux-Enthusiasten wurde nicht zuletzt durch die Präsentation des Yopy auf der CeBIT 2000 durch den damaligen G.Mate-Partner Samsung genährt.

Abbildung 3.3: Der G.Mate Yopy

Nachdem Samsung jedoch aus dem Projekt ausgestiegen ist, ist es ruhig um den Yopy geworden. In Deutschland war kein solches Gerät auszumachen. Somit kam der Yopy leider nicht für das Projekt in Frage, zumal seine Zukunft nach wie vor ungeklärt ist und somit eine Versorgung der Nullserie des Läufers mit Geräten nicht sicher war.

Auf der CeBIT 2002 wurde der Yopy wieder gezeigt, jedoch nun mit einem völlig neuen mechanischen Aufbau, der an ein klappbares Mobiltelefon erinnert. Eine solche Form paßt nicht zu den Anforderungen an einen PDA im Läufer, weshalb sich die Entscheidung gegen diesen PDA auch im Nachhinein als richtig erwiesen hat.

VTech Helio: Siehe Abbildung 3.4. Hierbei handelt es sich um einen PDA der Firma VTech, der mit dem proprietären VT-OS als Betriebssystem betrieben wird. VTech unterstützt allerdings auch die Entwicklung einer Linux-Distribution für den Helio.

Der PDA macht von seiner äußeren Erscheinung mehr den Eindruck eines Spielgerätes für Kinder, was ja auch der Haupt-Markt des An-

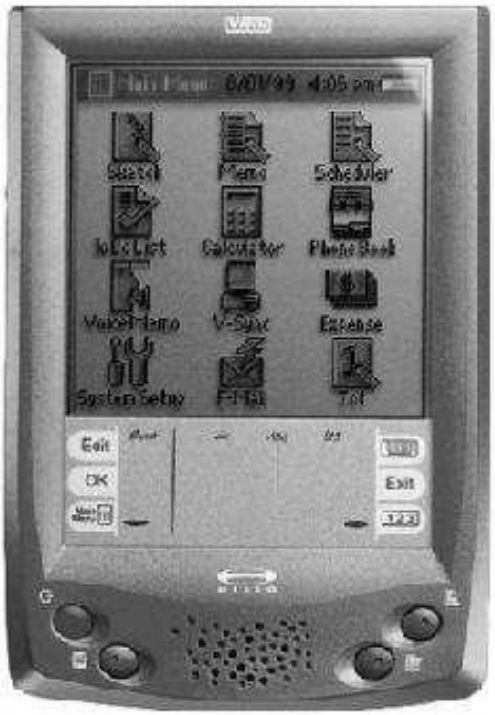

Abbildung 3.4: Der VTech Helio

bieters VTech ist. Der PDA verfügt außerdem nicht über ein Farb-Display, so daß er für das Projekt ebenfalls nicht in Frage kam.

Casio Cassiopeia: Die PDAs dieser Serie sind überaus leistungsfähig. Sie verfügen über einen Prozessor auf Basis der MIPS-Architektur bei 125MHz-150MHz, bis zu 32MB RAM und Farbdisplays mit einer Auflösung von 320x240 Pixel. Auf der Seite der Hardware spricht also vieles für den Cassiopeia. Abbildung 3.5 zeigt den Cassiopeia.

Ebenfalls verfügbar ist eine Implementierung des Linux Kernels für diese Geräte. Allerdings besitzen die Geräte dieser Serie kein wiederbeschreibbares Flash-ROM, sondern nur ein ROM für das Betriebssystem, so daß ein Start von Linux immer über spezielles WindowsCE Programm namens „CyaCE" erfolgen muß. Im Rahmen einer *Produkt*entwicklung ist es jedoch nicht wünschenswert, daß sich ein potentieller Nutzer zu sehr mit den technischem Details des Fahrzeuges auseinandersetzen muß.

Ein weiteres, nicht zu unterschätzendes Argument gegen die Geräte dieser Serie ist, daß sie von einem Asiatischen Hersteller vertrieben

Abbildung 3.5: Der Casio Cassiopeia

werden, und diese sich bisher als eher unwillige Sponsoren des Projektes erwiesen haben.

Compaq IPAQ: Der Compaq IPAQ (siehe auch Abbildung 3.6) ist von der Hardware her dem Sharp-PDA sehr ähnlich, der einzige Unterschied liegt in der standardmäßig ausgelieferten Software. Der IPAQ kommt standardmäßig mit Windows CE, während der Sharp mit Linux ausgestattet wird.

Für den IPAQ existiert allerdings eine sehr weit fortgeschrittene Portierung des Linux Kernels und etlicher Programme, so daß mittlerweile sogar schon mehrere Distributionen um die Gunst des IPAQ-Besitzers buhlen. Siehe dazu auch Kapitel 3.3.2

Die Hardware des IPAQ H3660 stellt wohl derzeit das technisch machbare im PDA-Umfeld dar. Das Gerät bietet Leistungen, die ein Entwickeln auf normalen Desktop-Rechnern ermöglicht, ohne das

Abbildung 3.6: Der Compaq IPAQ mit der Demo-Software für die CeBIT 2002

Unbehagen, die Zielplattform könnte eine bestimmte Funktionalität aus Gründen der dort vorhandenen Ressourcen nicht zur Verfügung stellen.

Die Hardware des IPAQ im Einzelnen:

CPU: StrongARM 206MHz

RAM: 64MB

Flash: 16MB interner Flash-Speicher

Display: 240x320 12Bit reflektives Farbdisplay

Schnittstellen: RS-232, USB, IRDA, Analog Sound

Erweiterungen: Es besteht die Möglichkeit, den IPAQ um folgende Slots zu erweitern: CompactFlash, 1x PCMCIA oder 2x PCMCIA. Hierzu wird der IPAQ in seine eigene Erweiterung, ein so genanntes „Jacket" hineingesteckt, was Veränderungen im

Formfaktor zur Folge hat. Die Jackets gibt es in Ausführungen mit den genannten Steckplatz-Konfiguration.

Sharp Zaurus: Dieser PDA verfügt über ähnliche technische Daten wie der Compaq IPAQ. Wie dieser verfügt er über einen StrongARM 206MHz Prozessor, 64MB RAM und 16MB Flash. Auch das Display ist ähnlich dem des Compaq. Abbildung 3.7 zeigt dieses Gerät.

Abbildung 3.7: Der Sharp Zaurus

Allerdings wird der zu Beginn der vorliegenden Arbeit noch nicht mit einem Namen versehene PDA nicht mit Windows CE, sondern mit Linux als Betriebssystem ausgeliefert. Die Software entspricht weitgehend der, die wir auf dem IPAQ einsetzen. Es kommt also ein Linux Kernel 2.4.X mit QTopia als graphischer Oberfläche zum Einsatz.

Leider war das Gerät zu Beginn unserer Arbeit noch nicht verfügbar, so daß wir das von all seinen Daten her ideale Gerät leider nicht verwenden konnten. Auf der CeBIT 2002 wurden erste Kontakte mit

Sharp geknüpft, um ein eventuelles Wechseln auf den Sharp Zaurus vorzubereiten.

Entscheidung

Anforderung	*Display*	*Optik*	*Anschlußmöglichkeiten*	*Verfügbarkeit*
VR3	–	0	0	+
Yopy	+	+	0	–
Helio	–	–	0	+
Cassiopeia	+	+	+	–
Zaurus	++	+	+	(-)
IPAQ	++	++	+	+

Tabelle 3.2: Übersicht über die verschiedenen Linux-fähigen PDAs

Tabelle 3.2 zeigt nochmals eine Übersicht über die verschiedenen PDAs mit ihren individuellen Eigenschaften. Die Wahl fiel letztlich auf einen Compaq IPAQ, da das Gerät nach den Informationen, die zur Zeit der Entscheidung im WWW zur Verfügung standen, über die weitestgehende Unterstützung unter Linux verfügt. Ein weiteres wesentliches Argument für dieses Gerät ist das reflektive Display. Dadurch wird eine Nutzung als zentrale Anzeige in einem Fahrzeug überhaupt erst möglich. Durch das reflektive Display wird die Anzeige des IPAQ bei stärkerer Sonneneinstrahlung automatisch heller, so daß im Gegensatz zu z.B. Notebooks ein Ablesen auch im Sommer bei großer Helligkeit möglich ist.

Leider war es zu Beginn der vorliegenden Arbeit nicht möglich, den Sharp Zaurus zu berücksichtigen, da er zu diesem Zeitpunkt nicht zur Verfügung stand. Im weiteren Projektverlauf wird allerdings versucht, auf den Zaurus umzusteigen, da er über ähnliche Leistungen wie der IPAQ verfügt, jedoch direkt mit Linux ausgeliefert wird. Die Verwendung eines Standard-PDA macht vor dem Hintergrund einer *Produkt*entwicklung natürlich Sinn.

3.2.4 Zusammenfassung

Wie weiter oben beschrieben wurde die Entscheidung zugunsten von Linux gefällt; Linux auf dem PDA ist sehr PC-ähnlich zu programmieren und verfügt über Multitasking. Außerdem ist es im Bereich der GUI sehr

flexibel und es ist nicht unbedingt ein PDA nötig, um Komponenten für die Läufer-Software zu entwickeln.

Für den IPAQ wurde sich entschieden, weil er die beste Wahl nach dem noch nicht verfügbaren Sharp Zaurus darstellt. Das Gerät verfügte als erstes am Markt über ein *reflektives* TFT-Display[11], was den Einsatz in einem offenen Fahrzeug überhaupt erst möglich macht.

[11] TFT=Thin Film Transistor

3.3 Auswahl der Linux-Distribution

3.3.1 Anforderungen

Für den IPAQ standen zum Zeitpunkt der Entscheidung mehrere Linux-Distributionen zur Auswahl. Es mußte sich also innerhalb des Projektes für eine entschieden werden. Dabei waren die Anforderungen der Entwickler mechatronischer Komponenten ebenso zu berücksichtigen wie die späteren Nutzer des Läufers.

Für Entwickler ist es am wichtigsten, daß ihnen die Entwicklung für das System möglichst leicht gemacht wird. Dies schließt eine einfach nutzbare Programmierschnittstelle ebenso ein wie gute Werkzeuge zum Erstellen der Graphischen Benutzerschnittstelle. Aber auch ein gutes Management der Softwareinstallation auf dem Gerät ist dem Entwickler wichtig.

Für den Anwender ist der Mehrfachnutzen des Gerätes entscheidend. Es soll zusätzlich noch als ,,normaler" PDA nutzbar bleiben. Dazu gehört eine ausgereifte Auswahl an Anwendungen zur Termin- und Adreßverwaltung. Aber natürlich auch ein umfangreiches Softwareangebot, was auch die Möglichkeit bietet, neue Versionen der installierten Software einfach einspielen zu können. Dies gilt insbesondere für die Systemsoftware, also die Distribution an sich, aber auch für die für den Läufer speziell angefertigte Software.

3.3.2 Die Alternativen

Compaq selbst unterstützt maßgeblich die Portierung des Linux-Kernels, ist aber mittlerweile aus der Entwicklung einer eigenen darauf aufsetzenden Distribution ausgestiegen, da mehrere aus der Sicht Compaqs bessere Alternativen zur Verfügung stehen. Zur Entstehungszeit der vorliegenden Ausarbeitung befindet sich Linux auf PDAs und insbesondere auf dem IPAQ unter dem Einfluß starker Entwicklertätigkeit, so daß sich viele Distributionen und Programme noch im Zustand ,,Projekt" befinden und erst nach und nach zu ,,Produkten" werden. Das prominenteste ,,Produkt" dürfte der Sharp Zaurus sein, dessen Software weitgehend mit der freien Oberfläche OPIE übereinstimmt.

Im folgenden sollen die einzelnen Projekte, die sich um Linux auf dem IPAQ bemühen, vorgestellt werden. Wie auch schon bei der Übersicht über aktuelle Linux-PDAs (siehe 3.2.3) kann und soll hier kein vollständiger Überblick über die Projekte gegeben werden. Die Projekte, die hier genannt werden, sind die, die im Rahmen dieser Studienarbeit näher untersucht wurden.

Die einzelnen Projekte sind:

PocketLinux: PocketLinux[Tec] kann man wohl mit Fug und Recht als das ambitionierteste Projekt bezeichnen. PocketLinux setzt sich aus einem Linux-Kernel, eine Java Virtual Machine[12] sowie einer XML[13]-basierten GUI zusammen. Leider hat ein kurzer Test der Version für x86-Linux zu der Erkenntnis geführt, daß die Kombination Java+XML+PDA wohl (noch) nicht performat genug ist, um die Ansprüche des Projekts Läufer an das Ansprechverhalten der Software zu erfüllen.

Mittlerweile wurde die Entwicklung von PocketLinux auch eingestellt. Die Entwicklung der Java VM geht allerdings weiter und wird als eigenständige JVM für Embedded Devices entwickelt.

Familiar Linux: Familiar[Pro] ist die ,,Brot-Und-Butter" Distribution von Linux auf dem IPAQ. In ihr sind auch die früheren Compaq-Distributionen aufgegangen.

Familiar verfügt über ein Debian[14]-ähnliches Paketsystem, wodurch die Installation weiterer Software sehr einfach möglich ist. Jede verfügbare Software ist hierzu in Pakete organisiert, die womöglich untereinander durch Beziehungen verbunden sind. Die Pakete werden dazu automatisch aus dem Internet geladen und eventuelle Abhängigkeiten zu anderen Softwarepaketen werden aufgelöst. Dadurch werden dem Anwender die von anderen Linux-Systemen bekannten Probleme mit Abhängigkeiten zwischen Softwarepaketen erspart. So führt z.B. die Installation eines graphischen Programms automatisch zur Installation der benötigten Umgebung. Für das Paketsystem der Distribution Familiar existieren verschiedene graphische Frontends, so daß die Handhabung der Softwareverwaltung für den Anwender sehr einfach möglich ist.

In der Standardinstallation verwendet Familiar das von UNIX-Workstations bekannte XWindow System in der Version 11 (kurz X11) als graphische Schnittstelle, es existieren aber auch andere graphische Oberflächen. Die Portierung bestehender Anwendungen ist recht einfach, da X11 als Fenstersystem auf nahezu jeder auf UNIX oder seinen Artverwandten basierenden Workstation Verwendung

[12] Kurz: Java VM bzw. JVM

[13] eXtensible Markup Language

[14] Debian: eine auf vielen Plattformen verfügbare, vollständig auf freier Software basierende Linux-Distribution

findet. Allerdings sind diese Anwendungen in der Regel nicht auf den kleinen Bildschirm des PDA hin optimiert, da an UNIX Workstations historisch gesehen schon immer sehr große Monitore mit großer Auflösung üblich waren. Ebenfalls verfügbar sind die ersten X11-basierte Anwendungen, die speziell für Familiar entwickelt wurden. Die Anwendungen decken den Bereich der Organizer-Software ab und berücksichtigen natürlich die spezielle Hardware, auf der sie laufen sollen.

Intimate: Das Ziel des Projektes „Intimate" ist es, ein komplettes Debian-Linux auf den IPAQ zu bringen. Hierfür wird mehr Speicherplatz benötigt, als der IPAQ in seiner Standard-Version zur Verfügung stellt. Deshalb setzt die Installation von Intimate zwingend das Vorhandensein einer externen Speichermöglichkeit voraus. Meistens wird wohl ein IBM Microdrive zum Einsatz kommen, das 1GB an Speicher im Formfaktor einer CompactFlash-Karte bietet.

Intimate entschädigt für den geschilderten Aufwand mit einer Softwareauswahl, die (fast) auf dem Niveau von Linux auf Intel-Rechnern liegt. Insbesondere sind die bekannten Pakte wie Gnome, KDE, Mozilla, Emacs und andere verfügbar. Die Anwendungen leiden jedoch unter dem kleinen Display des PDA, das über eine Auflösung von 240x320 Pixel verfügt.

Für den Läufer ist allerdings eine weitgehende Verfügbarkeit von Desktop- und Server-Anwendungen nicht unbedingt erforderlich, ja sogar hinderlich, da es im Bereich der XWindow-Anwendungen derzeit noch keine wirklich zufriedenstellende Software für den PDA-Einsatz gibt. Die benötigte Festplatte braucht überdies recht viel Strom, wodurch sich Intimate für die gegebene Anwendung als ungeeignet erwies.

QTOPIA: QTOPIA ist keine eigene Distribution, sondern ein Set von PDA-Anwendungen die auf der QT-Bibliothek des Norwegischen Herstellers Trolltech basieren.

Die Bibliothek wurde in der Version QT/Embedded (QTE) speziell an die Bedürfnisse von PDAs angepaßt. QTE operiert direkt auf dem Linux Framebuffer und nicht auf dem X Window System. Dadurch spart es Speicherplatz und Rechenzeit, verliert aber den Vorteil von X11, nämlich die Netzwerkfunktionalität.

QTE ist je nach Art der Kompilierung voll Sourcecode-kompatibel zu den anderen QT-Varianten für Windows, XWindow, MacOS, etc.

Um aus einer bestehenden QT-Anwendung für z.B. das XWindow System eine Anwendung für den PDA zu machen, Bedarf es keiner Änderungen, vorausgesetzt die GUI paßt auf den Bildschirm des PDA. Daraus folgt eine ungemeine Vereinfachung der Programmierung.

QT genießt außerdem den Ruf, über eine sehr gute Dokumentation zu verfügen, was den Entwicklern mechatronischer Komponenten sicher zugute kommen sollte.

QTOPIA stellt nun eine komplette PDA-Umgebung auf Basis der Klassenbibliothek QT zur Verfügung. QTOPIA schließt von Rahmenbedingungen, wie verschiedenen Formen der Texteingabe (Handschrift, Virtuelle Tastatur, T9[15] etc.) ein. Zusätzlich umfaßt das PDA System ebenfalls einen ordentlichen Grundstock an Anwendungen, die man auf einem PDA erwartet. Dazu gehören neben der PIM[16]-Suite auch ein Betrachter für Dateien von Tabellenkalkulationen, ein Webbrowser sowie Email-Client und nicht zuletzt eine Anzahl an Spielen.

QTOPIA findet z.B. auf dem Sharp Zaurus Verwendung, was für die Produktreife der Software spricht.

3.3.3 Die Entscheidung

Für die Verwendung im Läufer wurde QTOPIA auf Familiar gewählt, da dies unter den gegebenen Umständen die fortgeschrittenste Wahl darstellte. Familiar mit X11 kam trotz der technischen Überlegenheit des XWindow Systems nicht zum Zuge, da es an guten PDA Anwendungen für die Workstation-Grafikschnittstelle fehlte. QTOPIA ist explizit für den Einsatz auf PDAs entwickelt und stellt darüber hinaus sogar eine sehr gute Programmierschnittstelle zur Verfügung.

Auch die Aussicht auf kommerziell verfügbare PDAs mit QTOPIA beeinflußte die Entscheidung maßgeblich, genauso wie die Qualität anderer auf QT aufsetzender Softwarepakete wie z.B. KDE.

[15] Bei T9 werden wie beim Handy die Worte nach einigen wenigen Zeichen erkannt
[16] Personal Information Manager

3.4 Treiberprogrammier-Schnittstelle

Neben der im Kapitel 5 beschriebenen Programmierumgebung auf den Platinen benötigt der Entwickler mechatronischer Komponenten auch eine einfache Möglichkeit, um vom PDA aus einfach auf die in Kapitel 4 datailiert beschriebenen Kommunikationswege im Läufer zugreifen zu können. Genau diese Möglichkeit soll ihm die Treiberprogrammier-Schnittstelle[17] bieten.

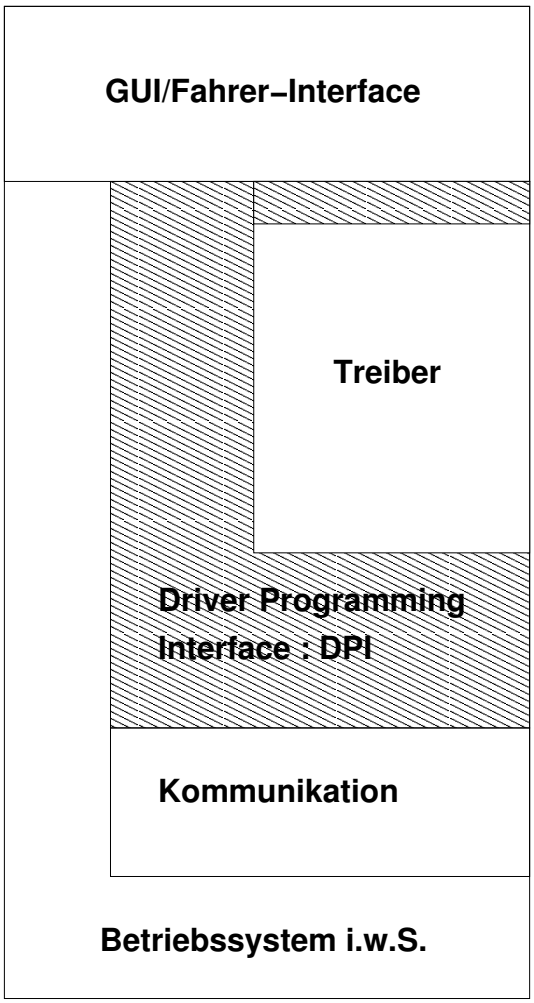

Abbildung 3.8: Die durch das DPI abzudeckenden Schnittstellen (schraffiert)

Abbildung 3.8 zeigt die grobe Architektur des DPI. Alle schraffiert dar-

[17] Im folgenden DPI für *Driver Programming Interface* in Anlehnung an API abgekürzt

gestellten Bereiche werden durch das DPI abgedeckt. Der Entwickler einer mechatronischen Komponente programmiert zu seinem Gerät lediglich einen *Treiber*, der die durch das DPI implementierten Kommunikationsmöglichkeiten nutzt. Dadurch wird der Entwickler davon befreit, sich mit den internen Strukturen des DPI auseinanderzusetzen.

Wie aus Abbildung 3.8 ebenfalls hervorgeht, hat das DPI aber eine zweite wichtige Aufgabe: Koordination der Kommunikation zwischen GUI[18] und dem Treiber für das Gerät. Diese Aufgabe des DPI ergibt sich unmittelbar aus der Natur des Projektes: Es ist nicht abzusehen, welche mechatronischen Komponenten im „fertigen" Läufer zu finden sein werden. Folglich kann die Erstellung der GUI nicht Teil dieser Arbeit sein, die sich ja nicht bis zum Ende des Projektes Läufer hinaus ausdehnt. In Folge dessen muß die Entwicklung der GUI außerhalb dieser Arbeit beim Design des DPI Berücksichtigung finden.

Auf diese beiden Aufgabenfelder des DPI Treiber $\Leftrightarrow$ CAN-Bus sowie Treiber $\Leftrightarrow$ GUI soll im folgenden detailliert eingegangen werden. Dem voran stehen einige Anforderungen, die sich aus den schon weiter oben in Kapitel 2 auf Seite 19ff beschriebenen Anforderungen an das Gesamtsystem herleiten.

3.4.1 Anforderungen

Bei der Entwicklung des DPI müssen die für das Projekt spezifischen Randbedingungen Beachtung finden: Zu diesen Randbedingungen zählt insbesondere , daß mehrere Entwickler parallel und wahrscheinlich ohne Kenntnis voneinander an Komponenten für den Läufer arbeiten bzw. arbeiten werden. Außerdem war zu Beginn der vorliegenden Arbeit noch nicht vollständig sicher, über welche mechatronischen Systeme der Läufer mal verfügen wird. Trotz des unsicheren Umfeldes müssen die gesamten Software- und Hardware-Komponenten am Ende der Entwicklung an einer Stelle zentral zusammengeführt und kontrolliert werden können.

Es ist folglich notwendig, die Entwickler so gut es möglich ist voneinander unabhängig zu machen, trotzdem aber nicht zu verhindern, deren Arbeit am Ende in einem Gesamtsystem zusammen zu führen.

Im Projektumfeld kann man sich nie sicher sein, daß die derzeit verwendete Software und Hardware auch konstant bleibt. Aus diesem Grund sollte das DPI möglichst wenige Annahmen über die Plattform, auf der es läuft, treffen.

[18] GUI = Graphical User Interface

3.4.2 High-Level Sicht auf die Kommunikation

Hier soll nun im folgenden auf die Anwender- bzw. Entwicklersicht auf die Kommunikation vom PDA aus eingegangen werden soweit dies für die Erläuterung des DPI hilfreich ist. Für die näheren Details der Kommunikationsprotokolle im Läufer sei auf Kapitel 4 verwiesen.

Um die Entwickler mechatronischer Komponenten für den Läufer voneinander unabhängig zu machen, ist eine Kommunikationsstruktur notwendig, die dem durch die Bereitstellung virtueller privater Verbindungen zwischen PDA Software und Platine Rechnung trägt.

Die Grundidee hinter der Kommunikation im Läufer ist die der objektorientierten Programmierung (OOP). Wir verstehen das physische Gerät und auch seinen Treiber als *Objekte*, die sich gegenseitig *Nachrichten* schicken. Diese Nachrichten sind im Läufer allerdings nicht so umfangreich selbst bestimmbar wie z.B. in einer objektorientierten Programmiersprache, wo sie durch Methoden modelliert werden können. Vielmehr werden im Läufer wirklich und letztlich sogar physisch Nachrichten[19] versendet.

Die Objekte, also Treiber und physisches Gerät haben einen Status, sind also Statusmaschinen. Ein physisches Gerät und sein Treiber stellen nun gekoppelte bzw. kommunizierende Statusmaschinen dar. Wenn sich der Status des einen *signifikant* ändert, teilt es diesen Statusübergang dem jeweils anderen über das Versenden einer Nachricht mit. *Signifikant* meint in diesem Zusammenhang, daß nur solche Statusübergänge dem anderen mitgeteilt werden, die für diesen auch von Bedeutung sind. Dieser Zusammenhang soll nun anhand eines Beispiels erläutert werden:

Als Beispiel für die Klassifikation von Übergängen siehe Bild 3.9. Dort wird ein einfaches Übergangsdiagramm für einen Blinker dargestellt. Die fett gezeichneten Übergänge werden dabei extern, also durch eine Nachricht des Treibers ausgelöst. Alle anderen Übergänge sind intern, werden also vom Gerät selber initiiert.

Wenn nun das Gerät z.B. eine Fehlfunktion feststellt, geht es in den Zustand $Z2$ über. Dieser Übergang ist signifikant, da der Treiber im PDA davon unterrichtet werden muß. Nicht signifikant sind die Übergänge zwischen $Z1A$ und $Z1B$, da diese den inneren Zustand des physischen Gerätes wiederspiegeln. Diese mit dem Treiber auszutauschen ist nicht nur unnötig, sondern auch gefährlich. Wenn nämlich diese inneren Zustände vom Bus abhängen, ist ein sicherer Betrieb des Fahrzeugs ohne Bus nicht mehr gewährleistet. Im Falle eines Blinkers scheint dies nicht kri-

[19] Konkret: Instanzen der Klasse `Message`

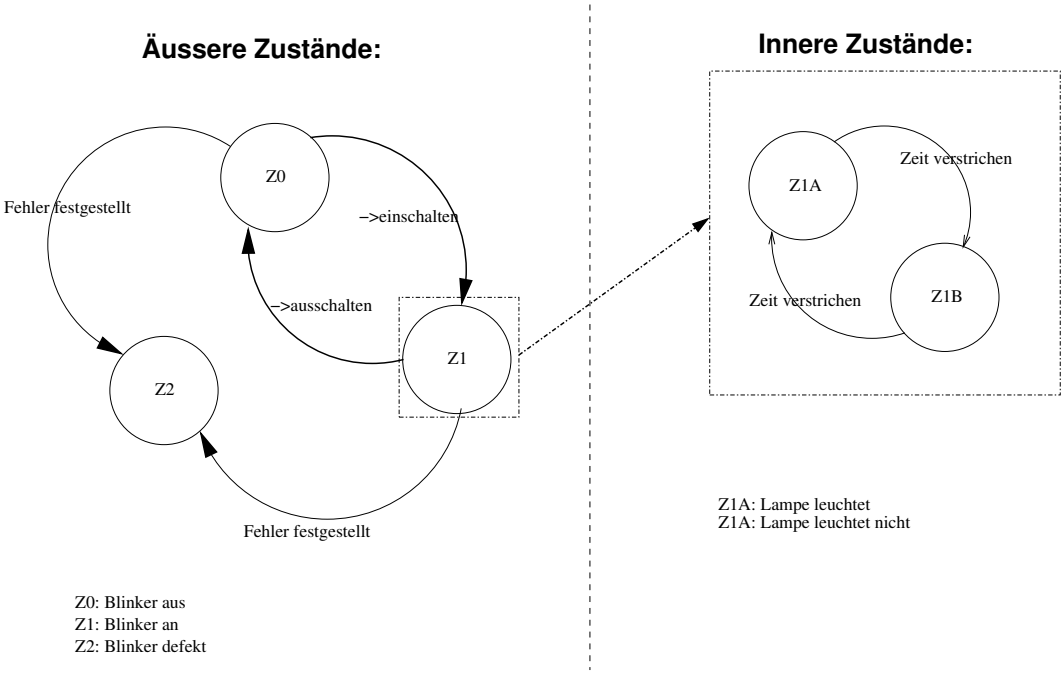

Abbildung 3.9: Übergangsdiagramm eines modellhaften Blinkers. Fette Linien bedeuten Übergänge, die vom Treiber aus ausgelöst wurden.

tisch zu sein, aber das mechatronische System des Läufers wird auch z.B. den Antrieb enthalten. Aus diesem Grund gilt für das Projekt der Grundsatz: *Kein sicherheitsrelevanter Regelkreis auf dem Bus!*

Wie werden aber nun diese Übergänge zwischen den beiden Statusmaschinen ausgetauscht? Grundbausteine der Kommunikation im Läufer sind wie weiter oben angemerkt so genannte Nachrichten[20], die von einem Busteilnehmer zum anderen gesendet werden. Jeder Busteilnehmer erhält dazu eine im System eindeutige Nummer, die sogenannte ID[21]. Über die Komponenten-ID wird die Verbindung zwischen einem realen Gerät, also z.B. einem Scheibenwischer und dem entsprechenden Treiber auf dem PDA hergestellt. Die Kommunikationsstruktur des Läufers macht das Routing der Nachrichten über den CANBus für den Entwickler transparent, d.h. er muß sich lediglich darum kümmern, daß sich sein Gerät und der dazugehörige Treiber unter der selben ID beim System anmelden. Prinzipiell ist es für den Entwickler auf dem PDA sogar unerheblich, ob überhaupt ein CAN-Bus verwendet wird. Man könnte sich z.B. einen ,,Treiber" für die Anzeige des PDA vorstellen, bei dem dann gar kein phy-

[20] im folgenden auch: Message
[21] ID: für IDentifier

sikalischer Bus mehr zum Einsatz kommt.

Die eigentliche Kommunikation erfolgt dann wie schon gesagt in Form von Nachrichten, die zwischen dem Treiber und dem zu ihm gehörenden Gerät ausgetauscht werden. Bei den Nachrichten handelt es sich immer um einen Befehl (8Bit) mit optionalen Parametern (15Bytes). Dieses Paradigma ähnelt wieder der OOP, ist allerdings eingeschränkter: Jedes Gerät kann lediglich maximal 256 externe Statusübergänge entgegennehmen. Durch Gespräche mit den anderen Projektteilnehmern wurde jedoch sichergestellt, daß diese Zahl ausreichend dimensioniert ist.

Dieser Aufbau ermöglicht es jedem Entwickler, für sein Gerät einen eigenen Satz an Befehlen zu definieren, ohne daß der Befehlsatz jedes Gerätes den anderen Teilnehmern im System bekannt sein muß. Die Forderung nach Unabhängigkeit der Entwickler voneinander wird durch dieses Design erfüllt.

Details zur physikalischen Kommunikation und der Verarbeitung der Nachrichten auf den Microcontrollern finden sich in Kapitel 4.

3.4.3 Implementierung

Überblick

Ziel der Entwicklung des DPI ist es, den Entwurf darauf aufsetzender Komponenten so einfach wie möglich zu machen, da die Entwickler der Komponenten sich neben der Programmierung auch noch anderen Teilen ihrer Komponente widmen müssen. Die auf der entwickelten Bibliothek aufsetzenden Module sind genauer:

Treiber Die Treiber stellen ein Interface zu den einzelnen mechatronischen Komponenten her. Dazu sollen sie das Gerät durch eine Klasse repräsentieren, die als Methoden die speziellen Fähigkeiten des Gerätes exportiert. Hierzu ist es nötig, den Entwicklern durch das DPI ein Interface zum CANBus zur Verfügung zu stellen. Außerdem müssen die Treiber auf Nachrichten sowohl von der GUI als auch von ihrem Gerät am CANBus reagieren können.

GUI Die GUI soll später das Interface zum Fahrer darstellen. In ihr werden alle Fäden zusammengeführt, die als Methoden aus den einzelnen Klassen laufen. Dazu muß es für den Entwickler der GUI einfach sein, auf die Methoden der Treiber zuzugreifen und deren Statusmeldungen zu erhalten.

Die genannten Module sind nicht Teil der vorliegenden Studienarbeit, da ihre Implementierung im Falle der Treiber am besten durch den Entwickler der mechatronischen Komponenten durchgeführt wird. Nur der Entwickler hat das Detailwissen, um die Treiber dem mechatronischen Gerät angemessen zu entwickeln. Im Falle der GUI kann diese erst dann erstellt werden, wenn alle Fahrzeugkomponenten feststehen, da in ihr auch die Fahrzeuglogik festgehalten wird. Um die Fahrzeuglogik zu implementieren, bedarf es genauerer Kenntnisse der Fahrdynamik des Läufers als die Autoren haben können, schon aufgrund ihrer Fachrichtung. Ziel dieser Arbeit ist es, die Kommunikation im Läufer für die spätere Verwendung zugänglich zu machen.

Aufgabe ist es also, zwei Schnittstellen zu definieren. Im einzelnen sind es die Anbindung der GUI an die Treiber sowie die Kommunikation zwischen Treiber und CANBus. Der Beschreibung dieser beiden Schnittstellen sei jedoch ein kurzer Einschub über die Wahl der Programmiersprache vorangestellt:

Verwendete Programmiersprache

Die Wahl der verwendeten Programmiersprache ergibt sich zum einen aus der derzeit verwendeten Plattform und zum anderen aus dem weiter oben beschriebenen grundsätzlicheren Designüberlegungen. Aus diesen ergibt sich unmittelbar die Verwendung einer Sprache, die den Objektorientierten Entwurf unterstützt, da dann die Kommunikation im Läufer auf natürliche Weise implementiert werden kann. Auf dem Linux-PDA sind einige Sprachen verfügbar, die diesen Entwurf unterstützen. Aufgrund des Vorwissens der Beteiligten kamen Java und C++ in Frage. Da die gesamte GUI jedoch wahrscheinlich auf QT aufsetzen wird, und QT selbst in C++ geschrieben ist, haben wir uns für C++ entschieden.

Das weiter oben beschriebene Grundkonzept sowie die folgenden Implemenierungsdetails nutzen jedoch bewußt keine Sprachmittel, die C++ kennt und Java nicht. Es soll damit der Weg offengehalten werden, die Kommunikationsstruktur im Läufer einfach auf andere Systeme zu übertragen. So wäre es z.B. mittelfristig denkbar, den PDA durch ein Java-Fähiges Handy zu ersetzen. Und genau diese interessanten Weiterentwicklungsmöglichkeiten sollen nicht verbaut werden. Eine Ausnahme von dieser Regel stellt die Anbindung der GUI an die Treiber dar, da der Bereich GUI auf mobilen Endgeräten auf absehbare Zeit wohl nicht portabel in den Griff zu bekommen ist.

Anbindung der GUI an die Treiber

Die GUI muß Befehle an die Treiber geben können. Das läßt sich recht einfach dadurch erreichen, daß die Treiber als lokale Variablen in der GUI Vorliegen. Die GUI wird also zum Hauptbestandteil des Software-Systems des Läufers.

Die umgekehrte Kommunikation gestaltet sich etwas schwieriger. Bei der Anbindung der GUI an die Treiber kann man grundsätzlich Verschiedene Wege gehen:

Callback Zum einen kann man allen Treibern bei der Initialisierung einen Pointer auf die GUI übergeben. Dabei muß aber jeder Treiber die GUI und ihre Methoden schon kennen. Da diese aber nicht Teil der vorliegenden Arbeit sein kann, ist die verbreitete Methode des Callback für unsere Anwendung leider nicht sehr geeignet.

Eine Alternative hierzu wäre es, dem Treiber ein vermittelndes Objekt zu übergeben, das eine Methode enthält, die einen String entgegennimmt. Hierdurch kann man die gewünschte Flexibilität erreichen, da ein späterer Autor der GUI mittels Stringvergleichen erkennen kann, was für ein Ereignis aufgetreten ist. Stringvergleiche sind allerdings nicht sehr effizient und auch im Läufer gar nicht nötig, wie im folgenden Text deutlich wird.

Polling Die GUI könnte auch die Treiber in regelmäßigen Abständen Pollen, d.h. eine bestimmte im Framework zu definierende Methode der Treiber-Objekte aufrufen. Über einen solchen Mechanismus würde dann jeder Treiber Rechenzeit erhalten und könnte eingehende Nachrichten bearbeiten.

Die Methode des Polling bietet die gewünschte Flexibilität, da die GUI nur noch die Treiber kennen muß, aber nicht umgekehrt. Allerdings ist es offensichtlich, daß ein solches Vorgehen starke Probleme hinsichtlich der Leistung hat. Zum einen wird so keine bedarfsgerechte Verteilung der CPU Leistung erreicht und zum anderen kostet das unnötige Aufrufen von Methoden, die derzeit nichts zu tun haben, unnötig Rechenzeit.

Signals und Slots Für die GUI kommt das Produkt „QTOPIA" der Firma Trolltech zum Einsatz, folglich stand eine weitere Möglichkeit zur Verfügung, die Kommunikation zwischen Treiber und GUI zu realisieren. Die auf QTOPIA basierende Lösung ist im Gegensatz zu den anderen Ansätzen nicht zu den Standard-Techniken zu zählen, wes-

halb hier eine Erläuterung des bei Trolltech entwickelten Verfahrens gegeben werden soll.

QT verfügt über einen Mechanismus, der mit sogenannten „Signals" und „Slots" arbeitet. Signale und Slots können eine beliebige Anzahl Argumente beliebigen Typs übermitteln und sind überdies typsicher.

Ein Objekt verschickt Signale, durch welche Slots von verknüpften (connected) Objekten aufgerufen werden. Am besten läßt sich dies an einem kleinen Beispiel zeigen:

```
class Foo : public QObject
 {
  Q_OBJECT
  public:
    Foo();
    int  value() const { return val; }
  public slots:
    void setValue( int );
  signals:
    void valueChanged( int );
  private:
    int  val;
 };
```

Die Ausdrücke „Q_OBJECT", „slots" und „signals" werden vom Meta Object Compiler (moc) benötigt. Dieser Compiler erzeugt ein neues C++-Sourcefile, das Code enthält, der das Objekt initialisiert. Dieses File muß compiliert und zu den anderen Objekt-Files gelinkt werden. Die genannten Ausdrücke werden vom Präprozessor entfernt bzw. so verändert, daß der C++-Compiler den Code problemlos übersetzen kann.

Die Implementation von Foo::setValue():

```
void Foo::setValue( int v )
 {
   if ( v != val ) {
    val = v;
    emit valueChanged(v);
   }
 }
```

Ein weiterer moc-Ausdruck ist „emit", womit ein Signal versendet wird. Nun werden zwei Instanzen von Foo miteinander verbunden:

```
Foo a, b;
connect(&a, SIGNAL(valueChanged(int)), &b,
        SLOT(setValue(int)));
b.setValue( 110 );
a.setValue( 88 );
b.value();          // gibt 88 zurück
```

Durch Aufrufen von a.setValue() verschickt a ein Signal, worauf der damit verbundene Slot b.setValue() aufgerufen wird.

Durch den Signal/Slot-Mechanismus von QT können zwei Objekte zusammenarbeiten, ohne daß sie sich gegenseitig kennen (es muß nur jemand da sein, der sie verknüpft). Ein solcher Mechanismus erleichtert die Programmierung von GUIs ungemein, da in einem neuen Widget[22] oft Standardkomponenten wie Pushbuttons etc. eingebunden werden müssen. QT übernimmt für den Programmierer die Verwaltung solcher Standardkomponenten, sie müssen nur dynamisch alloziert (mit new) und connected werden.

Genau das ist aber auch bei der Kommunikation im Läufer mehr als nützlich: Die Treiber können unabhängig von der GUI entwickelt werden und auch in eventuellen Nachfolgeprojekten Verwendung finden. Sie stellen ihre Funktionalität als Slot zur Verfügung und emittieren Signale, wenn sie der GUI etwas mitteilen wollen. Wohin die Signale dann geleitet werden, ist Sache desjenigen, der den Treiber verwenden wird.

Die beschriebene große Flexibilität gab den Ausschlag für eine Entscheidung zugunsten eines solchen Ansatzes. Informationen über QT finden sich unter [Tro]. Der einzige Nachteil der beschriebenen Lösung ist die Abhängigkeit von QT, was bei der Nutzung von QT als GUI für den PDA aber kein großes Hindernis darstellt.

Zugriff der Treiber auf den CANBus

Für den Zugriff auf den CANBus kann man ebenfalls die sehr mächtigen Strukturen von QTOPIA nutzen. In einem solchen Fall sendet das Framework ein Signal, daß eine neue Nachricht vorliegt. Über einen Verteiler-

[22] Element einer graphischen Oberfläche

mechanismus wird die Nachricht dann als Signal an einen Slot des entsprechenden Treibers gesendet.

Im Läufer wird ein solcher Ansatz allerdings nicht verfolgt. Statt dessen erben alle Treiber von einer gemeinsamen Basisklasse, die die Funktionalität des Sendens und Empfangens von Nachrichten zur Verfügung stellt. Das Empfangen ist dabei eine abstrakte virtuelle Methode, die von jedem Treiber einzeln implementiert werden muß. Hierzu haben die folgenden beiden Überlegungen geführt:

Zum einen den Teil der Bibliothek, der nicht direkt mit der GUI zusammenhängt, möglichst portabel zu gestalten. Das bedeutet auch, daß man sich in diesem Bereich nicht von QT's Präprozessor abhängig machen kann.

Zum anderen sind die anzubietenden Funktionalitäten in dem Teil des Frameworks wesentlich übersichtlicher und auch schon im Laufe der vorliegenden Arbeit bekannt: Das Senden und Empfangen von Nachrichten. Die genaue Kenntnis der anzubietenden Funktionen nicht zu nutzen würde unnötige Komplexität im Programm hervorrufen und die Bibliothek nur unnötig vergrößern.

Weg einer Nachricht

Um die Funktionsweise faßbar zu machen, soll hier ein kleiner Überblick über den Weg einer Nachricht durch das System gegeben werden. Betrachtet wird hier natürlich nur der Weg im PDA, wohingegen in Kapitel 4 ab Seite 65 auf die Details des Protokolls eingegangen wird.

Vom Benutzer zum Gerät: Nachdem der Benutzer am PDA eine physische oder virtuelle Taste betätigt hat, ruft die GUI die entsprechende Funktion des passenden Treibers auf. Dieser entscheidet dann, ob es sich um eine signifikante Statusänderung handelt. Falls nein, ändert er lediglich seinen internen Status bzw. tut nichts.

Falls es sich jedoch um eine signifikante Statusänderung handelt, erstellt der Treiber eine neue Instanz der Klasse Message, die er mit den passenden Werten für Befehl und Parameter füllt. Das Message-Objekt schickt er dann mittels der geerbten Funktion send() auf die Reise zum Gerät. An dieser Stelle ist für den Treiberentwickler die Arbeit getan und das DPI übernimmt die restliche Arbeit:

Zunächst wird überprüft, ob die Nachricht überhaupt den richtigen Absender trägt. Falls ja, wird die Nachricht gesperrt, d.h. es ist nicht

mehr möglich, Änderungen daran vorzunehmen. Dies hat Optimierungsgründe, da diese Nachricht dann in den weiter unten liegenden Schichten in einer Queue gespeichert wird und dort evtl. einige Zeit verbleibt. Da der Absender ja noch einen Pointer darauf hat, könnte es dadurch zu unangenehmen Problemen kommen.

Von dort wird die Nachricht an den MessageDispatcher weitergereicht, der sie lediglich an den Busmanager übergibt. Dieser stellt dann die Schnittstelle zum physikalischen Netzwerk dar.

Vom Gerät bis zum Treiber: Tritt auf Seiten des Gerätes ein signifikanter Statusübergang ein, sendet es mit den in Kapitel 4 beschriebenen Mitteln eine Nachricht an den Treiber. Diese Nachricht wird auf der Ebene des BusManager in ein Message-Objekt mit passenden Einträgen verwandelt und an den MessageDispatcher übergeben. Dieser prüft einige Annahmen und stellt die Nachricht dem passenden Treiber mittels dessen receive-Funktion zu. Ab diesem Zeitpunkt übernimmt der Treiber wieder die Regie und reagiert auf die Nachricht.

Details

Auf Details zur Implementierung soll an dieser Stelle unter Hinweis auf die im Anhang vorhandene API[23]-Referenz verzichtet werden.

[23] API: Application Programming Interface

3.5 Entwickler-Tool: DebugTreiber

Im Rahmen der Gespräche mit den ebenfalls am Projekt Läufer beteiligten Maschinenbau-Studenten kam immer mehr zum Vorschein, daß für die Entwicklung der mechatronischen Teile noch zusätzliche Werkzeuge benötigt werden. Diese sind nicht unbedingt durch die Aufgabenstellung abgedeckt, wurden aber im Laufe des Projektes nachgefragt. Neben einigen kleineren Werkzeugen wie z.B. einem Farbmischer, der es dem Designer erleichtert, das Farbschema für die PDA-GUI zu erstellen ist dies vor allem der so genannte DebugTreiber:

3.5.1 Zweck des Tools

So ist es für eine zügige Entwicklung von Komponenten nicht vertretbar, die Hardware parallel zum Treiber entwickeln zu müssen.

Ein Entwickler z.B. eines Blinkers sollte die Möglichkeit erhalten, seine Hardware zu testen, ohne tatsächlich schon den Treiber dazu entwickelt zu haben. Der Entwickler weiß, welche ID[24]sein Gerät hat und welche Operationen es versteht. Folglich kann er die korrekte Funktionsweise durch manuelle Eingaben und Beobachten der Reaktion seines Gerätes überprüfen. Beobachten kann er entweder Änderungen am Gerät selber (z.B.: Lampe blinkt) oder aber Rückmeldungen des Gerätes über den CAN-Bus. Um den Hardware-Entwicklern die Arbeit zu erleichtern, ermöglicht der DebugTreiber nun genau das: Manuelles Senden und Empfangen von Nachrichten an ein bzw. von einem bestimmten Gerät.

Da wir bei der Kommunikation zwischen Gerät und Treiber (siehe 3.4.2) ein hinreichend generisches Verfahren einsetzen, war es leicht möglich, ein Programm zu entwickeln, das einen solchen „Handbetrieb" eines Gerätes ermöglicht.

3.5.2 Funktionalitätsübersicht

Senden

Abbildung 3.10 zeigt die GUI zum Senden einer Message an ein Gerät am CAN-Bus. Wie in 3.4.2 beschrieben setzt sich das Protokoll aus Operationen und eventuellen Parametern zusammen.

[24]Diese jeder Komponente eindeutig zugeordnete Nummer muß allerdings für jedes mechatronische Gesamtsystem zentral vergeben werden, da ein globaler Addressraum, wie ihn z.B. Ethernet bietet, nicht zur verfügung steht.

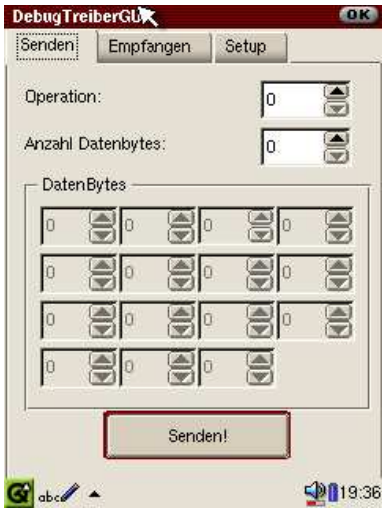

Abbildung 3.10: Screenshot Senden

Die Messages kann man hier (numerisch) zusammenstellen, um dem unter „Setup" eingestellten Gerät eine Message zu senden. Die GUI sollte weitgehend selbsterklärend sein. Sie verhindert, daß man Parameter eingibt, die nicht gesendet werden, indem man vorher auswählen muß, wieviele Parameter es denn werden sollen. Einstellen kann dies ein Benutzer mit dem Eingabefeld „Anzahl Datenbytes".

Ein solches Tool kann (und will) allerdings nicht verhindern, daß einem Gerät Befehle gesendet werden, die es nicht verstehen kann. Der Entwickler kann und will das selbst überprüfen, um auch das Verhalten seines Systems bei falschen Eingabedaten zu testen.

Empfangen

Abbildung 3.11 zeigt die GUI zum Empfangen von Messages vom CAN-Bus. Das Tool speichert eingehende Messages, die unter der im „Setup" eingestellten ID eintreffen.

Mit dem Feld „Übertragung" kann man auswählen, welche der gespeicherten Messages man betrachten möchte. Dieses Feld zeigt standardmäßig immer die Nummer der letzten empfangenen Message an.

Der Wechsel kann jedoch recht häufig sehr schnell von statten gehen. Aus diesem Grund ist es möglich, den Prozeß zu pausieren. Die Pause kann der Benutzer mittels des Kontrollfeldes „Pause" auslösen. Einkommende Nachrichten werden bei aktivierter Pause weiterhin im Hintergrund archiviert.

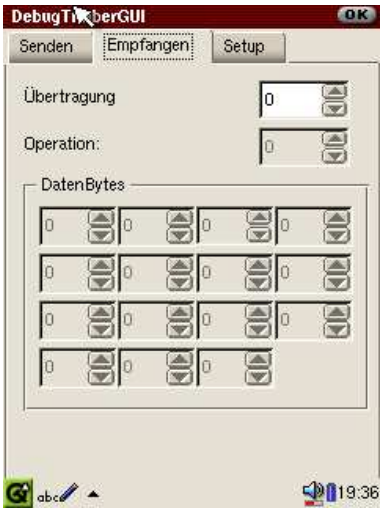

Abbildung 3.11: Screenshot Empfangen

Setup

Abbildung 3.12 zeigt die GUI zum Einrichten des Debug-Tools. Hier kann man die gewünschte ID des Gerätes einstellen, für das sich das Tool als Treiber registriert. Alternativ kann man hier auswählen, einen sogenannten „Broadcast" zu erstellen, der dann an alle Busteilnehmer gesendet wird.

Abbildung 3.12: Screenshot Setup

3.6 Fazit

Im Rahmen dieses Teilbereichs der Mechatronik-Entwicklung für das Projekt Läufer wurde der Rahmen für das Fahrerinformationssystem des Läufers entworfen und implementiert. Der Rahmen wird dann von den Maschinenbauern mit ihrer spezifischen Fachkenntnis ausgefüllt werden.

Daß das Framework hinreichend Leistungsfähig ist, bewies die zügige Entwicklung der Demo für die CeBIT 2002, bei der der Läufer und seine Mechatronik ausgestellt wurde. Abbildung 3.13 zeigt die hierzu entwickelte GUI. Auch bei dieser Entwicklung hat sich die Wahl von Linux als PDA-System bewährt, da umfangreicher Support seitens der Entwickler auf den entsprechenden Mailinglisten gegeben wurde, ohne den eine solch schnelle Umsetzung sicher nicht möglich gewesen wäre.

Abbildung 3.13: Der Compaq IPAQ mit der Läufer-Software für die CeBIT 2002

Kapitel 4

Kommunikation

von: Christoph Reichenbach

Inhaltsangabe

4.1 Einleitung

Der Läufer ist kein Monolith. Das wendige Fahrzeug kann kaum mit einem statischen Felsbrocken verglichen werden, sondern ist in seinem Aufbau modular. Doch die Module sind nicht in isolierten Welten; Kommunikation, Datenübertragung zwischen ihnen ist sogar essentiell– Blinker und Scheibenwischer können nicht, oder jedenfalls bestenfalls in beschränktem Umfang, über ihre Zustände entscheiden, zugleich kann und will man diese einzelnen Elemente nicht unmittelbar mit dem Fahrer verdrahten, der bei der Fahrt meist besseres zu tun hat, als Blinker über DIP-Schalter zu konfigurieren.

Dieser Abschnitt widmet sich der Kommunikation zwischen den Elementen des Läufers, insbesondere zwischen der zentralen Steuereinheit, hier durch einen PDA realisiert, und ihren Klienten. Verschiedene Möglichkeiten der Konstruktion eines Datenübertragungssystems werden beschrieben und bewertet werden. Am Schluß erfolgt eine Beschreibung der gewählten Kommunikations- und Gerätesteuerungsinfrastruktur.

4.2 Vorüberlegung

Bevor wir uns in die faszinierende Welt der Datenübertragung stürzen, sollten wir für dieses Projekt eine Grundsatzentscheidung treffen: Wollen wir ein möglichst zentral gesteuertes Netzwerk verwenden, in dem das Maximum an Entscheidungen vom PDA getroffen wird, oder diese Entscheidungen, soweit möglich, den zu verwenden vorgesehenen SX-Platinen überlassen?

Ein Vorteil einer zentralistisch angelegten Steuerung scheint, jedenfalls bei einem Reisefahrzeug, nur schwer erkennbar; der übliche Grund für solche Konstruktionen, die Minimierung von Redundanzen, greift in einem Netz von konzeptionell ohnehin separaten Einheiten nicht. Somit sollten wir, um den Datenübertragungsaufwand zu minimieren, nur mehr Konfigurationsinformationen vom PDA an die Peripherie, und umgekehrt nur wirklich auf dem PDA notwendige Informationen versenden; letztere lassen sich zusammenfassend als für den Fahrer gedachte Angaben– Geschwindigkeit, Akkumulatorstand– und zur Weiterleitung an andere Geräte vorgesehene Informationen wie Zustände von als Peripherie angeschlossenen Lichtschaltern oder Blinker-Hebeln beschreiben[1](s. 3.4.2).

[1]Das Vorkommen der letztgenannten Informationen ist nicht zwingend erforderlich, falls die Peripherie unmittelbar miteinander kommunizieren kann, z.B. durch eine Direktverbindung oder in einem Multi-Master-Bussysstem.

4.3 Anforderungen an die Kommunikation

Um von einer “guten” oder “schlechten” Lösung unseres Kommunikationsproblemes sprechen zu können, stellt sich uns somit zunächst die Aufgabe, die zur Bemessung der gesuchten Qualität dienenden Kriterien niederzuschreiben und auszuformulieren, nicht zuletzt gegeneinander abzuwägen, denn eine vollständige Ordnung muß sein, jedenfalls für ein eindeutiges Urteil.

Daß diese Abwägung nicht einfach sein muß, ist einleuchtend, doch führen wir sie trotz dieser Vorüberlegung zunächst aus, um eine Beurteilungsbasis zu bilden. Was also sind jene Meßlatten, die wir unsrer Kommunikationslösung anlegen wollen?

4.3.1 Korrektheit

Zunächst wollen wir die folgende Forderung stellen:

Alle übertragenen Daten müssen korrekt sein.

Diese Forderung bedarf einer Konkretisierung, die wir für einzelne Übertragungen wie folgt gestalten:

1. Wenn ein Datum den Empfänger erreicht, muß es zuvor versandt worden sein
2. Wenn ein Datum den Empfänger mehrfach erreicht, muß seine Semantik idempotent sein
3. Wenn ein Datum den Empfänger nicht erreicht, muß der Sender davon erfahren
4. Nichtrrivialität

Von diesen vier Forderungen ist insbesondere die Nummer 2 erläuterungsbedürftig. In Kommunikationssystemen kommt es erfahrungsgemäß vor, daß versandte Pakete aufgrund von unangemessen ausgelösten Fehlerbehandlungen (üblicherweise, wenn die Bestätigung der Gegenseite einem unbemerkten Übertragunsfehler zum Opfer fiel) wiederholt versandt werden. Unsere obige Forderung ist auf zwei Weisen erfüllbar, nämlich zum Einen, indem verhindert wird, daß der Empfänger die Botschaft mehrfach erhält, oder zum Anderen, indem auf einer höhergelegenen Interpretationsebene (auf die somit diese Fehlerbehandlung verschoben wird) der mehrfache Empfang der gleichen Botschaft– sofern nicht durch eine andere Botschaft unterbrochen– ohne Folgen bleibt.

Erkennung von Nichtübertragungen

Die erste Forderung ist in der Praxis auf eine sehr hohe Protokollebene verschiebbar, da wir Bidirektionalität des unterliegenden Kommunikationsmechanismusses fordern müssen, indem wir für jede versandte Anfrage eine explizite Empfangsbestätigung verlangen (und zugleich uns versichern, daß Empfangsbestátigungen eindeutig sind). Zwar kann in der Praxis auch die Bestätigung einer Übertragung verloren gehen, obwohl die Übertragung selbst erfolgreich war, aber eben aus diesem Grund stellten wir die dritte Forderung.

Nichttrivialität

Die ersten drei Forderungen sind durch ein triviales Modell bereits erfüllbar, einen *wissentlichen Nichtverschicker*, der kein Datum versendet und den Sender davon in Kenntnis setzt (was die erste Forderung unmittelbar und die beiden anderen trivial erfüllt). Die Forderung, daß überhaupt Daten übertragen werden können, müssen wir somit auch ausdrücken.

Diese vier Forderungen übernehmen wir nun als Grundforderungen. Da wir jenseits der Wunderbaren Welt der theoretischen Informatik operieren, werden sie natürlich nur "in den allermeisten Fällen" erfüllbar sein, da die (verschwindend geringe) Wahrscheinlichkeit eines Bitfehlers auf bearbeitenden Prozessoren jenseits realistischer Kontrollmöglichkeiten liegt, aber diese Aussage bezieht sich natürlich auf alle Forderungen, die wir im Verlauf der Anforderungsanalyse stellen werden. Zur Erfüllung dieser speziellen Forderungen jedoch wollen wir so geringen Spielraum wie möglich lassen.

4.3.2 Hinreichende Datenübertragungsrate

Die Geschwindigkeit, mit der Daten übertragen werden, muß den Anforderungen des Projektes genügen.

Diese Anforderung ist, in einer derartigen Formulierung, von kaum zu übertreffender Ungenauigkeit, doch eine präzise Formulierung der benötigten Datenübertragungsrate scheint in der gegebenen Situation kaum möglich, insbesondere, da die Menge der anzuschließenden Geräte nicht fest gegeben ist und somit nur eine Abschätzung gemacht werden kann, wieviel diese dann schließlich an "laufenden" Informationen benötigen beziehungsweise versenden.

Unsere Vorüberlegung jedoch, den Geräten größtmögliche Autonomie zu gewähren (also die Kommunikation auf das nötige Minimum zu redu-

zieren), legt nahe, daß die Bandbreite überschaubar bleiben kann; betrachten wir dazu als typischen Extremfall ein oft sendendes Peripheriegerät (Aufgrund der Autonomie wird die Kontrolleinheit eher selten Daten versenden, da deren Versand erst vom– im Vergleich sehr trägen– Benutzer angestoßen werden muß).

Schnelles Tachyometer

Typische Geschwindigkeitsmesser in ähnlichen Fahrzeugen werden oft nicht häufiger als zwei Mal pro Sekunde mit neuen Informationen versorgt. Ein auf der Zeitachse höher auflösendes Tachyometer könnte jedoch z.B. mit der Fernseh-typischen Rate von 25 Hz diese Informationen versenden; höhere Übertragungsraten sind erfahrungsgemäss mit nur geringem praktischem Gewinn verbunden. Bei 24 Bit Datenlast (8 Bit Kennziffer, 16 Bit Information) wären dies folglich, in diesem Beispiel, 600 bps für dieses eine Gerät.

Auf eine glatte Zahl aufrundend (und sehr grob nach oben abschätzend) erhalten wir z.B. 1024, für eine geschätzte Obergrenze von 16 Geräten also 16384 bps im auf Datenlast eingeschränkten Übertragungsverkehr– Kontrolldaten ignorierend– als theoretisches Minimum. Das ist, aus heutiger Sicht, nicht sehr viel, was aber, nach der Vorüberlegung, unsren Erwartungen grob entspricht.

Bandbreite alleine reicht natürlich nicht unbedingt; die Erzählung vom mit CD-ROMs beladenen Güterzug als König der Bandbreite ist geläufig genug, um an eine andere häufige Forderung der Datenübertragung zu erinnern.

4.3.3 Geringe Latenzzeiten und Prioritisierung von Nachrichten

Die Zeit, die zwischen dem Versand und dem Eintreffen eines Datenpaketes vergeht, sollte den gestellten Anforderungen genügen.

Erneut eine eher vage Forderung, die an unsere Argumentation zur nicht-Zentralisiertheit erinnert. Wie 'minimal' die Verzögerung sein sollte, ist an praktischen Beispielen wohl– abgesehen von offensichtlichen Beschränkungen– am ehesten an einer doppelten Übertragung zwischen einem Schalter und einer anderen Peripherie, beispielsweise einem Scheinwerfer, erkennbar; hier machen sich hohe Latenzzeiten zwar nicht in einer gefährdenden, aber doch für den Fahrer ersichtlichen und irritierenden Weise bemerkbar. Bei einer nicht direkten Verdrahtung müßte also eine

Übertragung zweier Botschaften innerhalb kurzer Zeit versandt werden können, bei direkter Kommunikationsmöglichkeit zwischen Schalter und Licht verringert sich dies auf eine einzelne Botschaft.

Eine gute Latenzzeit für diesen Kommunikationsschritt wäre, wie im letzten Kriterium erwähnt, 0.04s; praktisch ausreichend wäre jedoch auch eine schlechtere Zeit von 0.1s, da, wie erwähnt, zu guter Letzt doch "nur" wieder das Empfinden des Fahrers bedient wird. Im Falle sicherheitskritischer Kommunikation jedoch, wie z.B. busgesteuerten Bremsen, sind im Allgemeinen wesentlich geringere Zeiten nötig; dies impliziert die Möglichkeit einer Prioritisierung von Nachrichten anhand des angesteuerten Gerätes. Da sicherheitskritische Systeme im Läufer entweder autark (Getriebe) oder nicht busgesteuert (Bremsen) arbeiten, werden wir für diese an dieser Stelle keine konkreten Forderungen einbringen.

Gerade an diesem Beispiel sieht man natürlich eine zuvor schon oft implizit erwähnte notwendige Anforderung, die als nächstes ausgeführt werden soll.

4.3.4 Bidirektionalität

Kommunikation zwischen Peripherie und PDA sollte in beide Richtungen möglich sein.

Diese Forderung ergibt sich unmittelbar aus der Notwendigkeit, Informationen, wie zum Beispiel die aktuelle Reisegeschwindigkeit, aus Peripheriegeräten auszulesen, verbunden mit der Notwendigkeit, Geräte auch regulieren zu können[2].

4.3.5 Unterstützung für mehr als ein angeschlossenes Gerät

Es sollte möglich sein, mehr als ein Peripheriegerät mit dem PDA kommunizieren zu lassen.

Eine Ermangelung an Bidirektionalität oder Mehr-Client-Funktionalität wäre zwar allgemein
durch Verwendung mehrerer paralleler Kommunikationssysteme ausgleichbar, eine solche Konstruktion ist aufgrund des Mehraufwandes an Hardware und der grundsätzlich anderen Ansteuerung jedoch separat zu handhaben. Die angeklungene Forderung nach einem eher geringen

[2]Es ist praktisch nicht haltbar, die Geräte in Ein- und Ausgabegeräte zu separieren; als Beispiel sei ein Scheinwerfer genannt, der ein Durchbrennen der von ihm gesteuerten Glühbirne erkennen und dem PDA mitteilen kann.

Hardware-Aufwand läßt sich im Übrigen noch verallgemeinern, wie wir im nächsten Punkt sehen werden.

Nach Diskussion mit dem Läufer-Team wurde die Notwendigkeit eine Unterstützung von mehr als zehn separaten Geräten als nicht gegeben eingestuft. Aus Gründen zukünftiger Erweiterbarkeit wäre eine Unterstützung für eine größere Zahl wünschenswert; da diese jedoch nicht näher spezifiziert werden kann, wurde, um die Anzahl der möglichen Protokolle an dieser Stelle nicht unrealistisch einzuschränken, von mindestens 16 Geräten ausgegangen.

4.3.6 Geringe Ressourcenanforderungen

Das verwendete Kommunikationssystem soll "möglichst wenig" Hardware, Strom, und Prozessorleistung der beteiligten Kommunikationssysteme benötigen.

Eine präzisere Angabe jenseits von offensichtlichen Schranken des Machbaren sind hier nicht möglich; dies ist ein eher komparatives Kriterium.

4.3.7 Praktische Realisierbarkeit mit gegebenen Mitteln

Das Kommunikationssystem muß finanziell und technisch realistisch implementierbar sein.

Insbesondere ist, rein komparativ gesehen, eine Unterstützung seitens der Sponsoren, sowohl materiell als auch mit Informationen und praktischer Unterstützung ein relevantes und jedenfalls im Zweifelsfall den Ausschlag für oder gegen die "endgültige" Wahl einer entsprechenden Technologie gebendes Kriterium. Eine wesentliche Einschränkung ergab sich zudem durch den gewählten Controller, der im Läufer verwendeten Mehrzweck-Platinen, der nur eine sehr beschränkte Menge an Speicherplatz für Programme bietet.

4.3.8 Grundvoraussetzungen

Um nichts unnötig auszuschließen, wollen wir unsere Minimalforderungen derart formulieren, daß der Ausgleich fehlender Eigenschaften auf einer höheren Ebene prinzipiell erlaubt sein soll, endgültiger Ausschluß einer Technologie würde also nur erfolgen, wenn sie das Erfüllen eines der Kriterien (in einem angemessenen Maß) prinzipiell ausschließen würde.

Wir fordern von gewählten Technologien minimal, daß sie nicht ausschließen, daß folgende Kriterien erfüllt werden:

1. **Korrektheit**
2. **Hinreichende Datenübertragungsrate**: Mindestens 16384 bps für Daten alleine
3. **Geringe Latenzzeit**: Maximal 0.05s pro Botschaft oder 0.1s für zwei Botschaften[3]
4. **Prioritisierung von Nachrichten**
5. **Bidirektionalität**
6. **Unterstützung für mindestens 16 Peripheriegeräte**
7. **Geringe Ressourcenanforderungen**: Strombedarf auf Platine abdeckbar, eventuelle Kabel maximal handelsübliche Koaxialkabel an Dicke, sonstige Hardware sollte maximal soviel Masse haben wie der Rest der SX-Platine
8. **Praktische Realisierbarkeit**

Mit diesen acht Kriterien werden wir nun die verschiedenen Möglichkeiten, die sich uns zur Kommunikation bieten, beurteilen.

[3]Diese beiden Kriterien sind sehr unterschiedlich, da z.B. über Priorisierungsmechanismen unter Umständen sichergestellt werden kann, daß Botschaften, die eingingen, jedoch weiterversandt werden müssen, eine geringere Latenzzeit haben. Auf der anderen Seite jedoch ist es– bei einer Latenz von konstant 0.05s– durch zusätzlichen Bearbeitungsaufwand möglich, daß dennoch ein wenig mehr als 0.1s benötigt wird. Diese Zeit betrachten wir jedoch als unerheblich und verwenden somit 0.05s als ein einfacher zu handhabendes Kriterium.

4.4 Alternativen der Kommunikationshardware

Wir werden nun eine Reihe von Alternativen durchgehen, die uns, verbunden mit verschiedenen Vor- und Nachteilen, Kommunikation zwischen PDA und Peripherie erlauben. Im Falle einiger dieser Technologien könnten die benötigten Anschlüsse zwar unmittelbar auf die SX-Board aufgelötet werden, eine Kommunikation mit dem PDA würde jedoch eine Zwischenstation mit einem als Proxy agierenden Programm (oder entpsrechender Hardware) benötigen. Da auf den genannten Platinen ohnehin ein serieller Anschluß vorhanden ist, können sie, bei entsprechender Programmierung, unmittelbar als ein solcher dienen (sofern die verwendete Kommunikationsmethode der Forderung nach **Bidirektionalität** genügt).

4.4.1 Einfache Hartverdrahtung

Die einfachsten Dinge sind oft auch die besten, daher wollen wir zunächst die Direktverbindung des PDAs mit der Peripherie über direkte Kabelverbindungen betrachten.

Diese Lösung erfüllt zunächst unsere Minimalanforderungen, erlaubt beliebig viele Peripheriegeräte und ist an Bandbreite und Latenz nur durch den Systembus des SX-Controllers begrenzt. Doch schon hier stellt sich die erste Frage: Wie verbindet man die Controller direkt an ihrem Systembus? Ein Arbitrierungsmechanismus müßte herbei, um den Verlust von Übertragungen zu verhindern[4]. Aber damit haben wir schon wesentlich mehr als eine einfache Hartverdrahtung; wenn wir jedoch unser Modell beibehalten wollen, bliebe uns nur, die Peripherie direkt mit den Eingängen der SX-Ports zu verbinden. Dies würde natürlich die Anzahl der anschließbaren Peripheriegeräte unmittelbar beschränken– die notwendigen 16 Geräte sollten jedoch kein prinzipielles Problem darstellen.

Eine direkte Verbindung erzwingt jedoch einen unnötig komplexen Kabelbaum (wenn zwei Peripheriegeräte direkt nebeneinander liegen, ist es nicht ausreichend, sie miteinander zu verbinden– jedes von ihnen muß einzeln verkabelt sein). Eine derartige Konstruktion würde Masse am Läufer kosten und wäre fehleranfällig; dennoch verbleibt die Hartverdrahtung als eine, wenn auch nicht optimale, Möglichkeit. Ihr Hauptvorteil, die hohe Geschwindigkeit und geringe Latenz (die durch zwischengeschaltete Verstärker jedoch geringfügig leiden dürfte) ist im Projektrah-

[4] Dritte Forderung der Korrektheit

men zwar interessant, aber nicht fundamental ausschlaggebend.

4.4.2 Drahtlose Kommunikation

Die Zusammenfassung aller drahtlosen Kommunikationsmittel in einer einzigen Sektion legt das Urteil über diese natürlich bereits nahe, daher will ich es ohne Umschweife ausformulieren.

Drahtlose Kommunikation benötigt mehr Leistung als hartverdrahtete Kommunikation und ist störungsanfälliger; da sie kein geschlossenes System erlaubt (jedenfalls nicht ohne Kryptographie, die jenseits der Schranken der praktischen Realisierbarkeit lag), ist somit auch die Erfüllung der Forderungen der Korrektheit nur schwerlich vorstellbar.

4.4.3 Actuator Sensor Interface (AS-i)

Das 1993 entworfene Actuator Sensor Interface des AS-i Konsortiums ist ein Master-Slave-System mit bis zu 31 Slaves, das sich insbesondere durch eine sehr geringe Komplexität auszeichnet. Tatsächlich erfüllt es alle Anforderungen, die von uns an ein solches Netzwerk gestellt werden; sein einziger wesentlicher Nachteil ist die Beschränkung der Größe des Ein- und Ausgangskanals auf je 4 Bit, was zu einer Verwendung von fragmentierter Datenübertragung zwingt.
Da wir jedoch zu spät von seiner Existenz erfuhren und nicht unmittelbar mit Unterstützung seitens der Sponsoren rechnen konnten, muß seine Bedeutung auf eine Aufführung in dieser Ausarbeitung beschränkt bleiben.

4.4.4 Controller Area Network (CAN)

Das Controller Area Network, 1986 von der Robert Bosch AG zur Multi-Master-Kommunikation im Auftrag von Mercedes-Benz entwickelt, erfreut sich heutzutage einer wachsenden Beliebtheit auch bei anderen KFZ-Herstellern wie BMW, Porsche, Fiat etc., wird aber auch außerhalb der Automobilbranche genutzt.

Das CAN-Protokoll ist ein reines Kommunikationsprotokoll (welches auch vielfach bereits in Hardware implementiert wurde), bezieht sich also nicht auf die verwendeten physikalischen Transfermedien und die darübergelegene “Objektschicht” (wie die darüberliegenden Schichten gem. OSI-Modell von der CAN-Spezifikation genannt werden). Zur Zeit

existieren zwei relevante, im gleichen Netz ineroperable Protokollversionen:

CAN 1.2

Version 1.2 des Protokolls stellt Mechanismen zur Bus-Arbitrierung, Erkennung und Korrektur fehlerhafter Datenübertragungen und einen vorgesehenen Adressraum von 11 Bits für angeschlossene Geräte zur Verfügung; aufgrund einer protokollbedingten Einschränkung ist jedoch "nur" eine Adressierung von 2032 ($= 2^{11} - 16$) Busteilnehmern möglich.
Ein zur unmittelbaren Datenübertragung verwendetes CAN-Frame (der Version 1.2) maximaler Größe beinhaltet 8 Bytes an Daten und belegt (inkl. Inter-frame space, dem Abstand zweier Frames) 111 Bit.
Das Protokoll selbst läßt die genaue Hardware-Implementierung offen, gängig sind jedoch Raten von 1 Mb/s bei minimalem Verdrahtungsaufwand, was die Minimalanforderungen mehr als abdeckt.

CAN 2.0

Diese Protokollversion unterscheidet sich von CAN 1.2 lediglich in der optionalen Verfügbarkeit von 29-Bit-Bezeichnern[5].

4.4.5 Vehicle Area Network (VAN)

Dieses Netzwerk, in direkter Konkurrenz zum CAN, wurde in Frankreich entwickelt, scheint sich jedoch keiner vergleichbaren Popularität zu erfreuen. Ebenfalls als Multi-Master-fähiges Protokoll entworfen, wurde es 1994 ISO-standardisiert, hat jedoch keine (uns relevant erscheinenden) markanten Vorteile gegenüber seinem Konkurrenten.

4.4.6 Konklusion

Schlussendlich fiel die Wahl auf das CAN-Protokoll, nicht nur aufgrund der guten Abdeckung der Anforderungen, sondern auch und insbesondere wegen der Unterstützung bei dessen Verwendung, die seitens der Sponsoren des Läufer-Projektes geleistet wurde und des allgemeinen Interesses an einem in der Industrie derart verbreiteten Protokoll.

[5]Die Einführung dieser Erweiterung begründet sich historisch aus von der American Society of Automative Engineers (SAE) gestellten Anforderungen

4.5 Übrige Anforderungen und Umsetzung

4.5.1 Bereits durch CAN abgedeckte Anforderungen

Wir untersuchen zunächst, welche der konkreten gestellten Anforderungen bereits durch CAN selbst abgedeckt werden:

1. **Korrektheit**: Wir erinnern uns an die vier geforderten Eigenschaften:
   (a) *Wenn ein Datum den Empfänger erreicht, muß es zuvor versandt worden sein*: Wird trivial von CAN erfüllt.
   (b) *Wenn ein Datum den Empfänger mehrfach erreicht, muß seine Semantik idempotent sein*: Die Prämisse gilt in CAN nicht, somit muß die Konklusion nicht erfüllt werden.
   (c) *Wenn ein Datum den Empfänger nicht erreicht, muß der Sender davon erfahren*: Diese Implikation wird von CAN nicht selbst erfüllt; ein erweiterndes Protokoll müßte sie sich zur Aufgabe setzen. In unmittelbarem Zusammenhang damit ergibt sich ein praktisch relevantes Problem: Kurzzeitige Leitunsstörungen können den Verlust einzelner Botschaften zur Folge haben, ohne, daß die Kommunkation unbedingt zusammengebrochen wäre. Diese Situation erfordert Neuversendungen der alten Daten, wird jedoch nicht von CAN abgedeckt, da dieses Protokoll keine empfangenen Sendungen quittiert.
   (d) Datenübertragung muß nichttrivial sein
2. **Hinreichende Transferrate**: Marktübliche Implementierungen liefern 1 Mb/s; bei Wahl eines hinreichend intelligenten Protokolles sollte die gewünschte Mindest-Transferrate somit zu erreichen sein: Eine solche Basis-Übertragungsrate liefert, bei zwischen 111 (1×8) und 440 (8×1) Bits für 8 Byte, also zwischen knapp 580000 und 145000 bps alleine für Daten [6], weit mehr, als gefordert.
3. **Geringe Latenzzeiten**: Die Latenzen in CAN sind, wie allgemein in Bus-Systemen üblich, minimal. Bei schlechtest möglichen Frames liegt die Latenz unter 0.00012s, was darüberliegenden Protokollen viel Freiraum einräumt.
4. **Bidirektionalität**: CAN bietet, als Multi-Master-System, verschiedenste Möglichkeiten multidirektionaler Kommunikation.

[6] falls keine Fehlerbehandlung nötig ist

5. **Unterstützung mehr als eines Clients**: Wie angedeutet, sieht CAN 1.2 eine Adressierung von bis zu 2032 Clients vor; tatsächlich könnte durch spätere Auswahl auf den Empfangenen Daten diese Menge beliebig erweitert werden (was im Rahmen dieses Projektes jedoch einer Begründbarkeit entbehren würde).

6. **Geringe Ressourcenanforderungen**: Die aufgeführten Bedingungen (Subesektion 4.3.8) werden von der handelsüblichen Hardware uneingeschränkt erfüllt.

7. **Praktische Realisierbarkeit**: Bei Verwendung dieser Technologie erhielten wir Unterstützung durch unsre Sponsoren (von denen sogar eine entsprechende Empfehlung ausgesprochen worden war).

4.5.2 Verbleibende Anforderungen

Die wesentliche verbleibende Anforderung ist somit die dritte Forderung der Korrektheit, die eine Notifikation des Senders im Falle einer fehlgeschlagenen Übertragung voraussetzt; weitere Eigenschaften, wie ein verallgemeinertes Initialisierungsprotokoll erscheinen zwar ebenfalls hilfreich, können jedoch auf einer höheren Ebene nachgereicht werden.

4.6 Alternativen für das Kommunikationsprotokoll

Die Wahl des Bus-Systems (für das wir, auf externe Empfehlung hin, CAN-Controller vom Typ MCP2510 der Firma Microchip Technology Inc. verwenden), war somit gefallen. Diese liefern die bereits zuvor für diverse Abschätzungen verwendete Übertragungsrate von 1 Mb/s.

Es stand jedoch noch die Wahl des auf CAN aufsetzenden Datenprotokolles offen; zwar lieferte CAN selbst bereits, wie beobachtet, die notwendigsten Eigenschaften zur Datenübertragung, doch Übertragung auf reinem CAN ist sehr primitiv, erlaubt nur, Datenpakete "an alle interessierten Zuhörer" zur Verfügung zu stellen[7]. Zur gerichteten, sequentiellen Datenübertragung und zum Geräte-Management bedarf es weiterer Konventionen, in Form von Datenübertragungsprotokollen.
Betrachten wir einige der üblichen Möglichkeiten zur Implementierung eines Kommunikationsprotokolles:

4.6.1 Smart Distributed System (SDS)

Das Smart Distributed System [SDS], von Honeywell Inc. entwickelt, ist ein primär zur Automatisierung von Produktionssystemen gedachtes Komplettpaket von Sensoren und Aktuatoren, das bis zu 64 physikalische Einheiten miteinander verknüpfen kann. Aufgrund seiner Proprietät, des geringen Bekanntheitsgrades in Europa und der wenig fahrzeugtechnischen Orientierung wirdmeten wir diesem System keine besondere Aufmerksamkeit.

4.6.2 CANopen

Das CANopen-Protokoll [CAN] ist ein Protokoll, das sich durch Reichtum an Features und Umfang auszeichnet und seit 1995 von der internationalen *Can in Automation*- Gruppe (CiA) gepflegt wird. CANopen bietet verschiedenste Kommunikationsmodelle und selbst erweiterte Funktionalität wie die automatische Konfiguration von Busteilnehmern oder die Synchronisierung von Zeitmessungsgeräten oder synchrone Client-Server-Kommunikation "auf Antrag" über einen dedizierten Server (den sogenannten Service-Data-Object-Server).
Zentral für dieses Protokoll ist der Begriff des "Object dictionary", eines

[7] Die zuvor erwähnten "Adressierungsmechanismen" sind somit nur Konventionen des Auslesens.

zentralen Repositoriums, dessen Inhalt durch alle Kommunikationsteilnehmer, ihre konfigurierbaren Felder und deren Typen[8] gebildet wird.
Schon angesichts dieser selbstbezüglichen Eigenschaften wird die grosse Mächtigkeit dieses Protokolles klar, zugleich jedoch ist erkennbar, daß es weit mehr bietet als für diesen Bereich nötig und, insbesondere, bei einer manuellen Implementierung auf unsrer beschränkten Hardware unnötigen Overhead in Daten- und Programmspeicher verursachen würde.

4.6.3 DeviceNet

DeviceNet [Dev], ein mit SDS vergleichbares Protokoll der *Open DeviceNet Vendor Association*, ist, wie dieses, primär zu Automationszwecken gedacht. Die Spezifikation dieses Protokolles ist jedoch nur auf Bestellung gegen Entgelt erhältlich; da der Einsatz von DeviceNet-spezifischen Hardwarekomponenten keinen relevanten Vorteil zu versprechen schien, wurde von einer genaueren Evaluierung von DeviceNet abgesehen.

4.6.4 Eigenentwicklung

Zuletzt blieb noch die Möglichkeit eines selbst entworfenen, auf CAN aufsetzenden Protokolles. Diese hatte zwar zunächst den unmittelbaren Nachteil, noch nicht entworfen worden zu sein, in Anbetracht der Nichtverfügbarkeit existierender Implementierungen der anderen genannten Protokolle für die von uns gewählte Plattform jedoch andererseits den Vorteil, mit moderaten Entwurfsaufwand den Implementierungsaufwand den
Bedürfnissen des Läufers gegenüber angemessen gestaltbar zu sein. Die wesentlichste Schwachstelle dieser Vorgehensweise war die höhere Fehleranfälligkeit einer Eigenimplementierung.

4.6.5 Fazit

Schlußendlich fiel unsre Entscheidung schon sehr früh auf eine eigene Entwicklung, da der Aufwand für eine vollständige Implementierung bereits existierender Protokolle (soweit ihre Spezifikationen überhaupt öffentlich verfügbar waren) unseres Ermessens nach wesentlich höher gewesen wäre, wohingegen die Verwendung einer eingeschränkten Version eines

[8] mit einem sehr eingeschränkten, maschinell orientierten Typbegriff, wie in Laufzeitsystemen verbreitet

dieser Protokolle keine merklichen Vorteile gegenüber einem eigenen solchen erwirkt hätte. Eine (entsprechend fehlerträchtige) Eigenimplementierung wäre in jedem Fall nötig, auch, da nach unserem Kenntnisstand keine frei verfügbare Implementierungen der genannten Protokolle für den SX-Prozessor verfügbar sind.

4.7 Das Läufer-Protokoll für den CAN-Bus

Die Aufgaben des resultierenden Protokolls müssen wir nun wieder auf die bereits zuvor verwendeten Protokollkriterien zurückführen. Hierbei sind die Eigenheiten des CAN-Protokolls zu beachten; die Protokolle haben also nicht nur zur Aufgabe, unsere Korrektheitsforderungen zu erfüllen, sondern auch, eine gute Anbindung an CAN zu bieten– wobei es sich von diesem natürlich aus Gründen eines klaren Designs zumindest in seiner Spezifikation hinreichend weit abtrennen sollte.

4.7.1 Grundsätzliche Protokollstruktur

Um die Komplexität eines solchen Protokolles zu reduzieren, beschlossen wir eine Aufspaltung in zwei Protokolle, in denen eines die grundsätzliche Kommunikation und das andere einfaches Session-Management mit Timeouts spezifizieren würde. Trotz des zu erwartenden Overheads schien uns dieses Design wartbarer als ein komplett integriertes solches.

Eine zunächst noch offene Frage war, ob Multi-Mastering verwendet werden sollte, eine von CAN angebotene Möglichkeit. Der offensichtliche Vorteil war die Verringerung von Latenzzeiten von Meldungen von der Peripherie an den PDA, der Nachteil eine höhere Komplexität und mögliche Blockierungen des Busses durch ungezügelten prioritisierten Schreibverkehr auf dem Bus. Da die Blockierung des Busses jedoch in einem stark eingeschränkten System allgemein gut zu isolieren ist (und der tatsächliche Gebrauch des Multi-Masterings für viele Geräte unnötig ist), schien eine Verwendung dieser Möglichkeit zur schnellen Meldung von Fehlerzustandsübergängen sinnvoll (wenn auch wir Protokollseitig die Kommunikationsmöglichkeiten aus Gründen der Fehlererkennung einschränkten).

4.7.2 Struktur des Netzes

Aus theoretischer Sicht ist das Netz von sternförmiger Topologie, mit dem PDA als Zentrum; diese Vereinfachung des Protokolls schien uns hilfreich, um die vollständige Trennung (und somit Isolation von Fehlern) auf einzelne Geräte zu beschränken. Die tatsächliche Verdrahtung hingegen kann natürlich beliebig im Rahmen des von CAN Ermöglichten erfolgen.

4.7.3 Die beiden Protokollschichten

Die beiden Protokolle, mit Namen *LLO* und *LLZ* ("Läufer Layer One" bzw. "Zero"), nehmen also zueinander signifikant andere Aufgaben wahr:

Aufgaben des LLO-Protokolls

Die Aufgaben des LLO-Protokolls umfassen die folgenden:

1. Adressierung der Klienten
2. Verpackung der zu übertragenden Daten in Pakete des Data Link/ Network Layer (speziell CAN)
3. Busmastering

Aufgaben des LLZ-Protokolls

Das LLZ-Protokoll übernimmt folgende Teilaufgaben:

1. Session-Management
2. Erfüllung des ersten Korrektheitskriteriums

Eine "Sitzung" im für uns relevanten Sinn erstreckt sich hierbei von Aktivierung bis Deaktivierung des Läufers.

Der sichere Betriebszustand

Beiden Protokollen ist ein Konzept zu Eigen, das den Namen "sicherer Betriebszustand" trägt, sich auf Klienten im Bus bezieht und individuell für jedes angeschlossene Gerät definiert werden muß. Dieser sichere Betriebszustand ist von angeschlossenen Geräten im Fehlerfall einzunehmen, insbesondere, wenn die Kommunikation nach außen abreißt.

4.8 Der Bezug zum OSI-Schichtenmodell

Als kurzer Einschub soll hier noch einmal auf das OSI-Schichtenmodell eingegangen werden, das zwar keinen leitenden Einfluß auf das Protokolldesign hatte, wohl aber zur nachträglichen Klassifizierung verwendet werden kann.
In dem verwendeten Kommunikationssystem werden die Ebenen wie folgt belegt:

- **Schicht 1: Physical Layer**: Diese Schicht findet sich in Verdrahtung, dem MCP2510, dem SX-Microcontroller, und den dazwischenliegenden Verbindungen.
- **Schicht 2: Data Link Layer**: Die Fehlererkennung dieser Schicht ist im CAN-Protokoll spezifiziert, findet sich entsprechend physikalisch auf dem MCP2510.
- **Schicht 3: Network Layer**: Ansteuerung der Komponenten wird, im Rahmen der CAN-spezifizierten Adresskonventionen, von einer Implementierung des LLO-Protokolls realisiert. Die wesentliche Arbeit dabei wird durch Filtermechanismen im MCP2510 implementiert.
- **Schicht 4: Transport Layer**: Die Zerteilung in einzelne Pakete führt die LLO-Implementierung durch.
- **Schicht 5: Session Layer**: Die Registrierung von Busteilnehmern und Timeout-Checks bei der Kommunikation werden von LLZ realisiert.
- **Schicht 6: Presentation Layer**: Nach- bzw. Vorbearbeitung der übersandten Daten wird von Geräten und Treibern individuell gelöst und ist nicht Teil der hier beschriebenen Kommunikationsschicht.
- **Schicht 7: Application Layer**: Die tatsächliche Durchreichung der Informationen an den Anwender, ihre optische Aufbereitung, automatische Beantwortung oder, allgemeiner, Versendung, findet sich in den anderen Teilen dieser Ausarbeitung beschrieben.

4.9 Das LLO-Protokoll

Das Protokoll geht davon aus, dass versandte Daten "ganz oder gar nicht" ankommen, gemäß unseren Korrektheitsforderungen (die ja zu zwei Dritteln schon von CAN erfüllt wurden). In diesem Rahmen stellt es zusätzliche Sicherheitsmechanismen und ein Multi-Master-Protokoll, das sich auf CAN implementieren läßt[9].

4.9.1 Struktur des Protokolls

Das LLO-Protokoll ist ein prinzipiell Multi-Masterfähiges Protokoll, das die Nutzung der Möglichkeiten von Multi-Master- Sendungen jedoch auf asynchrone Notifikationen einschränkt. Somit existiert ein tatsächlicher Master in diesem Netzwerk, der als Controller bezeichnet wird; er erhält zugleich eine der 32 im Bus verfügbaren Adressen (1) fest zugewiesen. Eine weitere Adresse ist für explizite Broadcasts (z.B. für eine totale Systemabschaltung im Falle der Entfernung des den Controller steuernden PDAs) reserviert, was 30 Adressen für Busteilnehmer läßt.
Gerichtete und Broadcast-Adressierung sind somit möglich; die Länge der übertragenen Daten ist zudem durch LLO nicht eingeschränkt.

Fassen wir einmal kurz die vom Protokoll vorgesehenen Befehle mit ihren relativen Prioritäten zusammen:

Kürzel	Priorität	Sender	Bedeutung
SYS	3	Alle	Fehlermeldungen, Fehlerbehandlung
ACK	3	Alle	Bestätigung nach Übertragung
KAL	3	Controller	Keep-Alive-Signal (s. 4.9.1)
STX	2	Controller	Controller $\rightarrow$ Client- Transfer
RTX	1	Client	Client $\rightarrow$ Controller- Transfer
NOP	0	Controller	Busfreigabe

Betrachten wir nun die wesentlichen Eigenschaften des Protokolls:

Sicherheitsmechanismen gegen Datenverlust

Um den Verlust von Datenpaketen wenigstens in den häufigsten Fällen zu bemerken, wird mit jeder Übertragung der Inhalt eines 4-Bit-Sequenzzählers versandt, der (bis auf SYS-, KAL- und NOP-Nachrichten) zudem bei jeder nicht-Broadcast-Botschaft um eins erhöht wird. Stellt nun

[9]Die vollständige Protokollspezifikation findet sich als Anhang (Sektion B).

der Controller oder der Client beim Empfang einer Botschaft ein Nicht-Übereinstimmen der Sequenznummern fest, kann er davon ausgehen, einige Botschaften verloren zu haben und diese neu anfordern.
Diese Methode greift offensichtlich nicht immer; bei einer hypothetisch normalverteilten Datenverlustrate von p ist die Wahrscheinlichkeit, genau 16 Nachrichten hintereinander zu verlieren, jedoch genau $p^{15} * (1-p)$, im Falle von Vielfachen von 16 (also allen Möglichkeiten, in denen ein Datenverlust nicht bemerkt würde) somit, für die Aufsummierung der Wahrscheinlichkeiten des Verlustes an der $n$ten Stelle

$$p_n = p^{16n+15} * (1-p)$$

und im Limes

$$\begin{aligned} \lim_{n\to\inf} \sum_{k=0}^{n} p_k &= (1-p) * p^{15} * \lim_{n\to\inf} \sum_{k=0}^{n} p^{16n} \\ &= (1-p) * p^{15} * \lim_{n\to\inf} \frac{p^{16(n+1)} - 1}{p^{16} - 1} \\ &= -\frac{(1-p) * p^{15}}{p^{16} - 1} \end{aligned} \tag{4.1}$$

Entsprechend ist bei einem "realistischen" normalverteilten Datenverlustwert die Wahrscheinlichkeit für einen kompletten Ausfall dieses Mechanismusses vernachlässigbar; selbst bei einem Störfaktor von 0.1 läge die Wahrscheinlichkeit noch bei unter 10^{-15}, sogar ein Störfaktor von 0.9 könnte nur in knapp 2.6% der Fälle einen Totalausfall dieses Sicherungsmechanismusses bewirken:

p	**Datenverlustwahrscheinlichkeit**
0.001	$0.999 * 10^{-45}$
0.01	$0.99 * 10^{-30}$
0.1	$0.9 * 10^{-15}$
0.15	$0.37221 * 10^{-12}$
0.2	$0.262144 * 10^{-10}$

Tatsächlich treten in der Praxis jedoch oft sequentielle Ausfälle auf, in denen ein vorheriger Ausfall die Wahrscheinlichkeit für einen erneuten Ausfall erhöht. Die Berechnung der Ausfallwahrscheinlichkeit hierfür ist aufwendiger, im Allgemeinfall jedoch aufgrund des stochastischen Zusammenhanges aller relevanten Zufallsvariablen nicht möglich.

Für allgemein geringe Fehlerwahrscheinlichkeiten ist eine solche Steigerung der Fehlerwahrscheinlichkeit aufgrund eines vorhergehenden

Fehlers mit einer Verschlechterung des Verhaltens verbunden, die mit größeren Sequenznummermengen besser aufgefangen werden könnte.

Keep-Alive

Dies alles ist im Falle einer kompletten Unterbrechung der Datenleitung jedoch wenig hilfreich. In diesem Fall kommt das Keepalive-Signal zum Tragen, das in regelmäßigen Abständen vom Controller auf den Bus geschrieben wird; wenn nicht mindestens ein solches `KAL` innerhalb von 0.3s empfangen wurde, sind Klienten dazu angehalten, sich selbst in den sicheren Betriebszustand zurückzufahren.

Busfreigabe und Client-zu-Controller-Botschaften

Da alle vom Controller verwendeten direkten Steuerbefehle Priorität über Client-Befehle haben (mit Ausnahme von `ACK` und `SYS`, die von Clients im Rahmen von vom Controller angestoßenen Kommunikationssequenzen gesandt werden dürfen, sowie des schweren Systemfehlers `SYS:EGLBL`, der alle Systeme in den sicheren Betriebszustand zwingt), muß der Bus vom Controller explizit freigegeben werden. Dies kann alternativ durch das Nicht-Versenden von Nachrichten oder dem Versand eines minimal prioritisierten `NOP`-Befehls (zu Debug-Zwecken) geschehen. Eine solche explizite Freigabe erfolgt, wenn der Controller selbst keine anstehenden Botschaften hat, ansonsten in regelmäßigen Intervallen.

`RTX`- Übertragung läuft analog zur `STX`-Übertragung von Controller zu Client, verwendet jedoch einen eigenen Sequenzzähler[10].
`RTX`-Botschaften sind die einzigen Botschaften die, auf vier Prioritisierungsstufen, noch einmal untereinander an Wichtigkeit unterscheiden können. Eine genaue Spezifikation der Verwendung dieser Prioritisierung wurde im Rahmen dieser Studienarbeit nicht durchgeführt und den endgültig die Geräte programmierenden Mitentwicklern überlassen.

4.9.2 Umsetzung und Tests

Das LLO-Protokoll wurde in C zu Testzwecken, aber auch zur seriellen Kommunikation implementiert. Für den SX-Chip wurde aus terminlichen Gründen nur ein vereinfachtes Protokoll mit der vorgesehenen endgültigen API implementiert; die vollständige Protokollimplementierung muß-

[10]Die historische Begründung dafür liegt im Wunsch, Vollduplex effizient unterstützen zu können; dies war relevant für die Verwendung von LLO über die serielle Schnittstelle.

te (ebenfalls aus terminlichen Gründen) auf Mitte Mai 2003 verschoben werden.

4.9.3 Einsatz im Proxy

Zur Kommunikation zwischen PDA und dem Rest des Busses muß ein "Proxy" verwendet werden, eine dedizierte Platine, die der Vermittlung zwischen CAN-Bus und der seriellen Schnittstelle des PDAs dient. Die Kommunikation zwischen Proxy und PDA findet über die serielle Schnittstelle statt, die beide Geräte optional auf Vollduplex betreiben können. Hierbei werden LLO/LLZ-Nachrichten vermittels eines einfachen seriellen Protokolls (SLP, beschrieben in C) übertragen, das der Trennung von Blöcken dient.
Der Proxy selbst nimmt im Netz noch eine weitere Aufgabe wahr: Er Überprüft das Vorhandensein des PDAs und sendet einen Broadcast der Form `SYS:EGLBL` im Falle des Ausbleibens von Antworten des PDAs (Systemabsturz, Ausfall, oder Entfernung des Gerätes), um im Falle der Ausschaltung der zentralen Kontrollinstanz die angeschlossene Peripherie in den sicheren Betriebszustand zu zwingen. Zudem bestätigt er selbst `RTX`-Sendungen mit `ACK`, um die Implementierung von Timeout-Behandlung auf der Peripherie unnötig zu machen.

4.10 Das LLZ-Protokoll

Das LLZ-Protokoll ist ein zustandsbasiertes Protokoll, das eine feste Maximallänge von Botschaften vorsieht (16 Bytes, mit einer Minimallänge von einem Byte) und eine sehr spezifische Semantik erzwingt. Zudem implementiert es den letzten noch fehlenden Aspekt des Korrektheitskriteriums, die Erkennung der Nichterreichung ihres Ziels durch eine Nachricht[11].

4.10.1 Struktur des Protokolls

LLZ definiert Kommunikation zwischen genau zwei Teilnehmern, von denen einer dedizierter Sender und der andere Empfänger ist (somit überläßt es die Handhabung größerer Mengen von Teilnehmern dem darunterliegenden LLO-Protokoll). Beide können Kommunikation initiieren, aber nur der Sender kann eine Antwort "erzwingen" lassen; zudem ist er auch derjenige, der den Zustand des Empfängers kontrolliert.
Das LLZ-Protokoll wird in mehreren Instanzen verwendet, wobei maximal eine pro Kanal des unterliegenden Protokolles angeboten wird (bei LLO werden aufgrund der maximal 30 verwendeten nicht-Controller-Adressen also höchstens 30 Instanzen des Protokolls zugleich ausgeführt).

Zustände der LLZ-Empfänger

Bevor es jedoch zu einer Ausführung der Datenübertragungen des Protokolles kommt findet die Initialisierung der Busteilnehmer statt, bei der der Sender zugleich (möglichst durch Verwendung von Broadcasts eines unterliegenden Protokolles) überprüft, wieviele Protokollinstanzen er zu verwenden hat, auf welchen der Kanäle des unterliegenden Protokolles also Empfänger zu erwarten sind.

Clients befinden sich initial im `PRE-INIT`- Zustand, der ihnen jegliche Kommunikation auf dem Bus, mit Ausnahme der Beantwortung von Initialisierungssignalen (`INI`) verbietet. Sobald sie ein solches Signal erhalten, gehen sie kurzfristig in einen weiteren Zustand (`INIT`), in dem sie eine Überprüfung der Protokollversionsnummer durchführen und, falls kein Fehler auftrat (den sie an den Sender kundtun würden, um Sendern, die mehrere Protokollversionen unterstützen, eine entsprechende Suche zu ermöglichen), in den `RUNNING`-Zustand über, der ihren normalen Operationsmodus darstellt. Dabei wird ein weiteres Signal an den Sender, `IOK`,

[11] Die vollständige Protokollspezifikation findet sich als Anhang (Sektion A).

versandt.
Auf Befehl des Senders hin schalten sie sich (eventuell auch schon aus dem PRE-INIT-Zustand) in den DOWN-Modus, der dem sicheren Betriebszustand entspricht.

Befehlstabelle

Bei der Bestätigung der erfolgreichen Initialisierung vermittels des IOK-Blockes wird eine Tabelle der von diesem Gerät unterstützten Befehle mitverschickt; diese Tabelle ist bitweise kodiert, wobei in 16 Bytes Raum für 128 Befehle geschaffen wurde. Eine spezifische Wahl der Befehlscodes wurde im Rahmen dieser Studienarbeit nur zu Testzwecken durchgeführt und wird späteren Implementierern überlassen, um nach genauerer Auswahl der zu verwendenden Peripherie eine Abstimmung deren Prioritäten untereinander den mit der tatsächlichen Betriebsweise vertrauteren Entwicklern zu überlassen.
Der Vorteil einer derart versandten Tabelle ist eine eingeschränkte "Plug-and-Play"- Nachrüstbarkeit der Peripherie, sowie die Möglichkeit für Treiber seitens des PDAs[12], mittels der Befehlstabelle unmittelbar die Verfügbarkeit von Funktionen, die in einer echten Teilmenge der Menge der Versionen der Peripherie implementiert wurde, überprüfen, und somit ihre Handlungen oder die von ihrer API angebotene Funktionenmenge entsprechend ausrichten. Zudem erlaubt sie eine frühzeitige PDA-seitige Erkennung nicht erlaubter Befehle; dort ist es leichter, Entwickler über diesen Fehler zu informieren[13].

Semantik der Befehle

Wie im Protokoll spezifiziert, ist die Semantik der Befehle eine Semantik der *absoluten Zustandstransformation*, in der für einen Befehl p mit einem festen Befehlscode und n Parametern gilt, daß für beliebige $\overline{x} = x_1, \ldots, x_n$, $\overline{x}' = x'_1, \ldots, x'_n$ die Ausführung von $p(\overline{x})$ nach $p(\overline{x}')$ den gleichen internen Zustand wie die Ausführung von $p(\overline{x})$ ohne $p(\overline{x}')$ erzeugt. Die Motivation für diese (die Ausdrucksfähigkeit prinzipiell einschränkende) Designentscheidung ist wie folgt:

[12] welche ihre Klienten üblicherweise auf einem bestimmten Kommunikationskanal erwarten

[13] Ein SX-Assembler-seitiger Präprozessor zur korrekten Erstellung dieser Befehlstabelle wäre hier ein nützliches Werkzeug, eine entsprechende Implementierung steht jedoch noch aus, da die SX-Entwicklungsumgebung integriert ist und somit keine einfache Einbildung in einen Buildprozess erlaubt.

1. Senderseitig ist es damit möglich, eine konstante Maximalgröße für die Nachrichtenqueue zu definieren

2. Das Designprinzip des doppelten Zustandsautomaten wird erzwungen, da alle "relevanten" Zustände nun PDA-seitig dupliziert werden. Dies nimmt den Entwicklern Designfreiraum, ohne ihnen Funktionalität vorzuenthalten, was uns aufgrund der Wahl der endgültigen Entwickler sinnvoll erschien.

3. Im Fehlerfall können neuere Botschaften ältere auch Senderseitig überschreiben und unnötigen Neuversand verhindern (nach explizitem Übertragungsabbruch)

Timeouts

Datenübertragungen in LLZ nehmen an, daß sie vollständig und hinreichend korrekt durchgeführt werden oder jedenfalls dem Übertragenden eine Nachricht zukommt, wenn bei der Übertragung ein nicht korrigierbarer Fehler auftrat (im LLO-Protokoll ist das bei massivsten Datenverlusten und fehlschlagender Fehlerbehandlung der Fall). Von der korrekten Überprüfung der Beantwortung von Botschaften geht es jedoch nicht aus, daher wird bei jeder Versendung eines Befehls b mit beliebigen Parametern zum Zeitpunkt t ein Tupel (b, t) gespeichert, wobei eine (partielle) Zuordnung $\mathit{sent}\ b \mapsto t$ resultiert, also jedem b ein eindeutiges t (jeweils das aktuellste Datum, da, wie gefordert, gleiche Befehle sich "gegenseitig überschreiben") zugeordnet ist.
Mit dieser Information wird nun in regelmäßigen Intervallen nach Timeouts von Antworten gesucht und somit der Rest des Korrektheitskriteriums erfüllt[14].

4.10.2 Umsetzung und Tests

Die Implementierung der Timeouts beschränkt sich auf den PDA; auf der Ebene der SX-Controller, also der Clients, ist im Allgemeinen kein ausreichender Speicherplatz für eine Timeout-Fehlerbehandlung; Timeouts treten zudem meist in Situationen auf, in denen der Client vom Netz abgetrennt ist, ein Fall, der jedoch bereits durch das periodische Keepalive-Signal des LLO-Busses behandelt wird.

[14]Tatsächlich wird natürlich manchmal umsonst Alarm geschlagen, da unter Umständen nur die Bestätigung verloren gint, aber dies haben wir nicht ausgeschlossen

Die Implementierung und Anbindung auf Hochsprachenebene wurde abgeschlossen, auf SX-Ebene ist jedoch zum Zeitpunkt des Schreibens nur ein Fragment der Implementierung verfügbar.
In der ersten Implementierung ist die Beschränkung von Befehlen zumindest SX-seitig auf solche mit bis zu 5 Bytes an Parametern vorgesehen; Notifikationen (`NTF`) und Kommandos ohne Antworten (`CMD`) werden nicht verwendet werden.

4.11 Fazit

Im Rahmen des Projektes wurde eine für die Zwecke des Läufers geeignete Kommunikationsstruktur entworfen und getestet, eine vollständige Implementierung auch auf den Mikrocontrollern war aus verschiedenen Gründen bis zur Fertigstellung dieser Ausfertigung nicht mehr möglich und muß im Anschluß erfolgen. Zum Ausgleich wird anstelle von LLZ zur Zeit ein vereinfachtes Protokoll (MLP, ebenfalls eine Eigenentwicklung) eingesetzt. Die endgültige Entwicklung wird nach Abschluß der M.Sc. Thesis des verantwortlichen Entwicklers nachgereicht.

4.12 Ausblick

In zukünftiger Arbeit planen wir, die Implementierung des LLZ-Protokolles abzuschließen. Danach hoffen wir, eine vereinfachte Programmierunmgebung für das synchrone Zustandsmodell entwerfen und implementieren zu können sowie eine Kaskadierung des LLZ-Protokolles zur Vergrößerung des Adressraumes zu spezifizieren und, bei Bedarf, zu implementieren. Parallel dazu planen wir Benchmarks, um den hinreichenden Durchsatz unserer Implementierung zu demonstrieren.

4.13 Zusammenfassung

In diesem Abschnitt spezifizierten und analysierten wir die Anforderungen für ein Kommunikationssystem für den Läufer. Wir verglichen eine geeignete Auswahl von verfügbaren und potentiellen eigenen Implementierungen miteinander, anhand dieser Kriterien, und erläuterten die Eigenschaften der gewählten Kombination, des CAN-Busses mit einem zweischichtigen Protokoll, das wir selbst entworfen hatten. Wir schlossen mit eine Diskussion der Protokollschichten.

Kapitel 5

Embedded Platinen

von: Christian Rüdiger

Inhaltsangabe

5.1 Aufgabenstellung

Herzstück dieser Arbeit ist die Implementierung einer „intelligenten" Platine. Sie stellt die Schnittstelle zwischen dem Kommunikations-Framework, wie es in den vorangegangenen Kapiteln beschrieben wurde, und den etwaigen mechanischen und elektromechanischen Applikationen der Ingenieure des Projekts dar. Um die Inhalte dieser Arbeit zu erläutern gilt es den Begriff des mechatronischen Systems zu erklären.

Als mechatronisches System wird gemeinhin ein funktionell und räumlich integriertes mechanisch elektrisches System bezeichnet, in dem Sensoren Informationen aufnehmen, Prozessoren diese Informationen auswerten und Aktoren geziehlt genutzt werden, um z.B. Kräfte oder Bewegungen zu erzeugen, die auf das System und seine Umgebung zurückwirken.

Vor diesem diesem Hintergrund steht den Ingenieuren im Projekt mit der Platine ein Bausatz zur Entwicklung eines mechatronischen Systems zur Verfügung. Auf der Platine enthalten ist ein Prozessor, Sensoren und Aktoren sowie Anschlüsse für weitere Sensorik und Aktorik. Abbildung 5.1 stellt die Platine im Kontext eines mechatronischen Systems dar. Ziel ist es also ein Werkzeig zu erstellen mittels dem es Ingenieuren möglich ist, schnell und einfach mechatronische Applikationen zu erstellen und in das Gesamtsystem Läufer zu integrieren.

5.2 Anforderungen an die Peripherieelektronik

Die Mechatronik im Projekt Läufer muß einer Reihe projektspezifischer Anforderungen standhalten. Was bedeuten also die eingangs geschilderten Zielsetzungen für die Peripherieelektronik?

Da sich der genaue Funktionsumfang aller mechatronischen Komponenten für die Zukunft nicht abschätzen läßt und neue Komponenten jederzeit einfach einzugliedern sein sollen, muß das System leicht erweiterbar sein. Für Maschinenbauingenieure muß das System mit vertretbarem Aufwand bedienbar sein. Für den Benutzer ist der Effekt und nicht der Weg interessant. Die begrenzten Ressourcen des autonomen Systems Läufer erfordern außerdem eine sparsame Nutzung von Energie. Um die Fahrer nicht durch Fehlfunktionen oder Ausfall eines Gerätes zu gefährden muß das System betriebssicher sein. Noch vor der Inbetriebnahme gilt es, die Hardware vor Fehlbedienung durch den Ingenieure zu schützen. Das heißt sowohl in der physikalischen Auslegung als auch in der Programmierung müssen mögliche Extreme in der Nutzung des Fahrzeugs

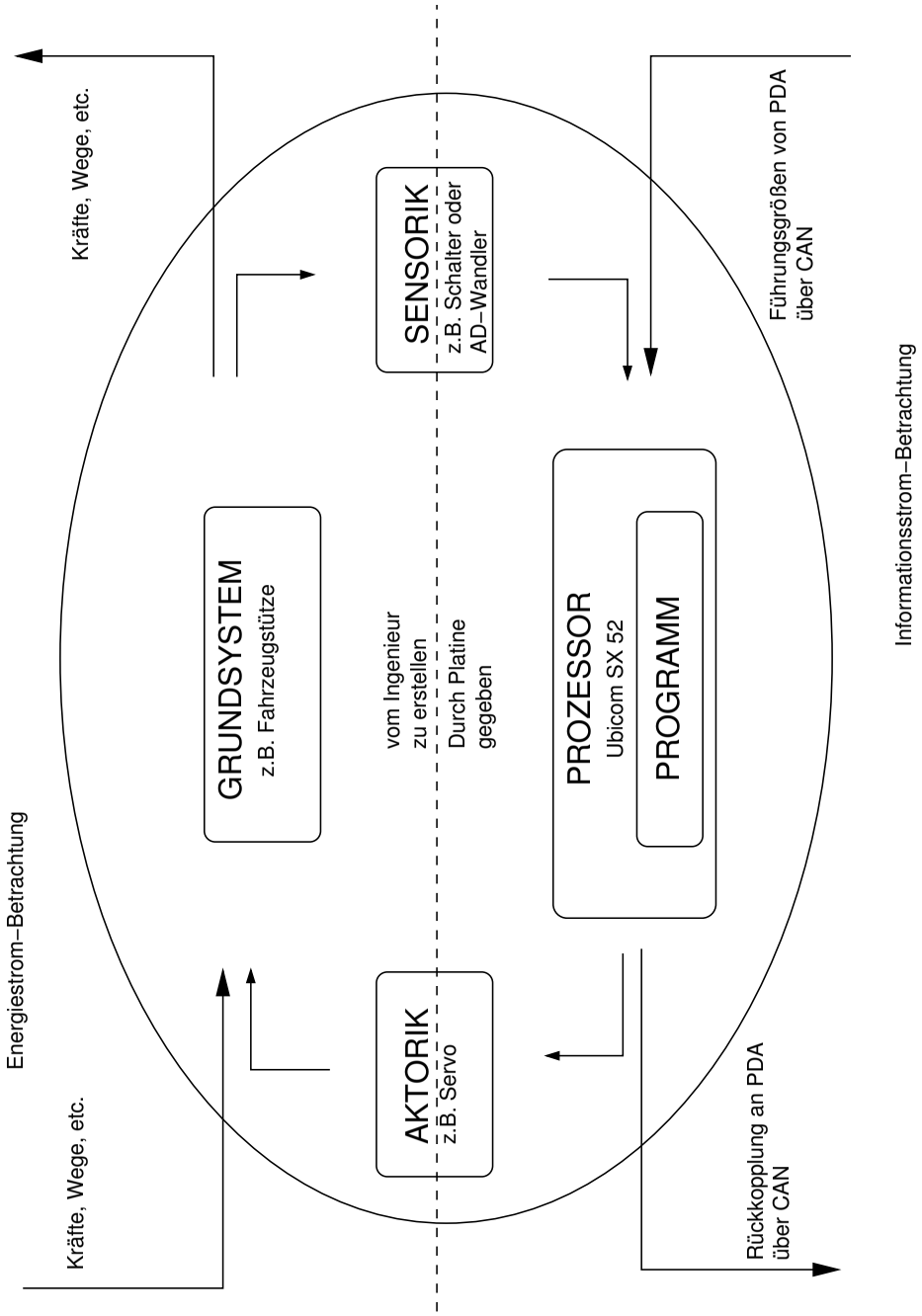

Abbildung 5.1: Grundstruktur eines mechatronischen Systems im Läufer (nach [Wal95])

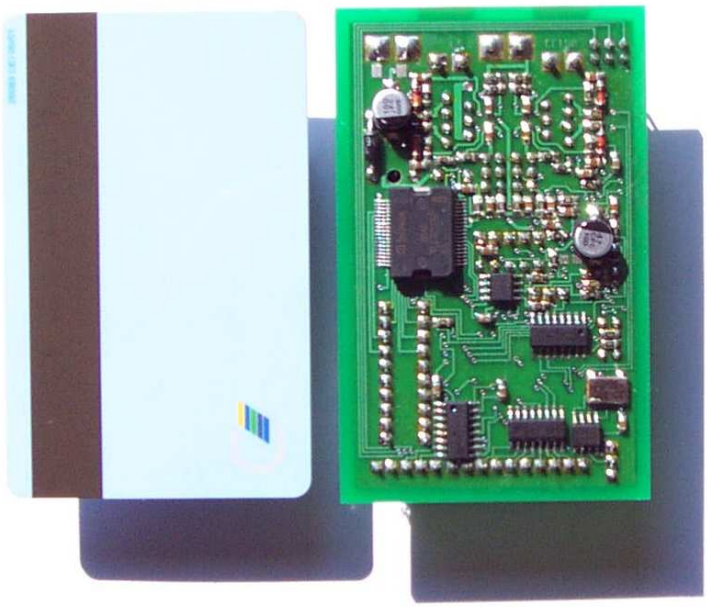

Abbildung 5.2: Grössenvergleich Platine Scheckkarte

berücksichtigt werden. Bei der Zusammenstellung der Platine muß außerdem die Verfügbarkeit der Bauelemente sichergestellt sein. Wie in den restlichen Teilen des Läufers zollen wir auch hier dem Design Tribut: die Platine sollte so wenig Raum wie möglich einnehmen, um der Gestaltung des Fahrzeugs größtmöglichen Freiraum zu gewähren.

Nicht zuletzt muss auch der Sicherheit Rechnung getragen werden. Sicherheit in diesem Zusammenhang heißt: Sicherheit vor Strom- und Spannungsspitzen zur Betriebszeit und Sicherheit der Komponenten vor Fehlbedienung durch den Entwickler.

5.3 Die Platine

Die Platine durchlief zwei Iterationsstufen der Entwicklung. Die erste, kleinere Platine enthielt noch keinen ausgereiften SPI-Bus[1] (der später erwähnte Multiplexer war noch nicht vorgesehen), die Low-Side Switches waren nicht integriert und die angeschlossene Peripherie wurde durch einen kleineren Microcontroller (der SX28) gesteuert. Der kleinere Microcontroller unterscheidet sich unter anderem in der Anzahl der I/O-Ports, der niedrigeren Taktfrequenz sowie einem kleineren Arbeits- und Programmspeicher von der aktuell verwendet Version. Mehr dazu in Kapitel 5.4.1.

Die zwei erstellten Prototypen wurden Bald durch eine neue vierlagige scheckkartengroße Trägerplatine ersetzt. Der zweite Revision berücksichtigt eine Reihe weiterer Ideen zur Annäherung an das Ziel einer multifunktionalen Mechatronikapplikation für Maschinenbauingenieure. Die Dimensionierung der passiven Bauelemente erlaubt eine Anpassung an andere Versorgungsspannungen als die momentanen 12 Volt. Ein Spannungswandler kann Versorgungsspannungen von bis zu 48 Volt auf die für CMOS üblichen 5 Volt herunterregeln. Die Platine entstand in ständiger Kooperation mit den Maschinenbaustudenten des Projekts und der Firma Cetron. Um den Anforderungen der Ingenieure gerecht zu werden, waren immer wieder Erörterungen von möglichen Aufgabenfeldern der Platine mit den Maschinenbauern nötig.

Im Folgenden werden einige Aspekte der Platine näher betrachtet.

5.3.1 Kosten

Die Platine enthält Standardbauelemente wie sie zum Teil auch in der Automobilindustrie verwendet werden. Die Kosten für die IC's belaufen sich auf ca 58,- Euro (eine genaue Aufstellung ergibt sich aus Tabelle 5.2). Die passiven Bauelemente belaufen sich auf ca 20,- Euro (LEDs, Kondensatoren, Spule, Dioden und Widerstände). Zusammen mit der vierlagigen Platte ergibt das einen Preis von ca. 110,- Euro ohne Bestückungskosten. Das Layout erlaubt jedoch eine Teilbestückung die sich auf die für ein spezifisches Peripheriegerät notwendigen Bauelemente beschränkt. Ein weiterer wichtiger Punkt bei der Kostenbestimmung war die Verwendbarkeit und das Zusammenspiel der Bauelemente in der gewünschten Zusammenstellung.

Die genauen Kosten der Platine hängen nicht unwesentlich von der

[1] siehe dazu Kapitel 44

Wahl der Komponentendistributoren ab. Preisunterschiede von bis zu 400 % (z.B. bei blauen LEDs) sind keine Seltenheit. Eine sorgfältiger Preisvergleich der möglichen Lieferanten ist wichtiger Bestandteil der Kostenbestimmung (und letztendlich des Einkaufs) gewesen.

5.3.2 Ausfallsicherheit zur Betriebszeit

Komponenten mit Anschlussmöglichkeiten auf der Platine vertragen meist kurzfristige Strom und Spannungsspitzen und enthalten gegebenenfalls Zener-Dioden. Näheres dazu steht in der Beschreibung der einzelnen Komponenten in Kapitel 5.4.2. Die Platien selbst enthält eine besonders dünne Leiterbahn, die, sollte der Entwickler doch einmal eine zu hohe Spannung anlegen, als Schmelzsicherung dient. So werden die teuren IC's vor Schaden bewahrt.

5.3.3 Design

Um sich in das Design des Läufers zu integrieren sind sowohl die Erweiterbarkeit als auch die simple Größe der Platine von Wichtigkeit. Die Größe der Platine beträgt ca. $4,5 * 8,6cm$, hat also etwa die Größe einer Scheckkarte (siehe Abbildung 5.2).

5.4 Wahl der Komponenten

5.4.1 Microcontroller

Der Markt für Microcontroller stellt eine große Auswahl an verschiedensten Konfigurationen zur Verfügung. Um bei der großen Auswahl zügig zu einer guten Entscheidung zu kommen gilt es, die Menge der Kriterien frühzeitig einzuschränken.

Wichtigste Kriterien bei der Auswahl waren dementsprechend: Preis, Anzahl I/O-Ports, Taktfrequenz (um möglichst viel Freiheit bei der Entwicklung neuer Peripherie zu haben), Qualität erhältlicher Entwicklungsumgebungen, Energieverbrauch, Größe des Programm- und Arbeitsspeichers sowie vorhandene Interrupt-Möglichkeiten.

Um den Preis gering zu halten empfahl sich eine 8-Bit Architektur. Im 8-Bit Bereich sind Produkte folgender Hersteller näher in Betracht gezogen worden: Infineon/Siemens (C500-Familie), Motorola (68HC11, 68HC05, 68HC08), Microchip (PIC16Fx), Ubicom (SX).

- Infineons SABC500-Familie bietet mit zum Teil integrierten RS232-Bauelementen eine reiche Auswahl für Kommunikationsanwendungen. Mit bis zu 24MHz sind die Microcontroller für die vorgesehene Anwendungen schnell genug. Der Preis von 9 bis 14 Euro pro Microcontroller ist vergleichbar hoch. Desweiteren müssen für den SABC515 die Programme in externen Speicherbausteinen abgelegt werden, die unnötig viel Platz in Anspruch nehmen.

- Die Microcontrollerreihen 68HC11, 68HC05, 68HC08 von Motorola wird bereits vielfach in der Autoindustrie genutzt. Die CISC-Architektur machen die Implementierung von Kommunikationsanwendungen schwierig, da das auszählen des exakten Timings mit unterschiedlich langen Instruktionen den Aufwand unnötig vergrössert. Motorola gleicht dies durch eine reiche Auswahl an Modellen mit integrierter Kommunikationslogik aus. Für die 68HC11Fx Familie sind mehrere Entwicklungsumgebungen, und reichlich Beispielapplikationen erhältlich. Die betrachteten Microcontroller wiesen jedoch höchstens einen interruptfähigen Pin auf, um Interrupts durch externe Geräte auszulösen. Desweiteren existiert kein Stromsparmodus. Auch die Preise von bis zu 14 Euro sprechen gegen den 68HC11x.

- Mikrochip ist in den USA für die PIC-Microcontroller bekannt. PIC sind schnelle RISC Microcontroller die sich gut für Kommunikationsanwendungen eignen. Für die Läufer-Platine wurde die PIC16F87x-Familie unter die Lupe genommen, da sie unter den interruptfähigen 8-Bit Controllern in der gewünschte Preisklasse liegt (5 - 7 Euro). Auch hier ist eine große Anzahl an Quellen für die Programmierung sowie eine Auswahl an Entwicklungsumgebungen erhältlich. Der PIC16F87x ist mit 14 interruptfähigen Pins unter seinen Konkurrenten am besten ausgestattet. Die sehr dürftige Ausstattung mit Programmspeicher schränkt die Eignung für die Mechatronikplatine jedoch wieder ein.

- Mit dem SX52 von Ubicom steht zusätzlich ein PIC-kompatibler Microcontroller zur Auswahl. Mit acht interruptfähigen PIN's ist der SX52 nicht ganz so gut ausgestattet wie der PIC16F87x, reicht aber für die vorgegebenen Zwecke aus. Den Mangel an Programmspeicher teilt der SX52 mit vier Kilobyte nicht mit seinem Konkurrenten. Sein Preis ist mit ca. 7 Euro besser als der des Motorolas und

gleichauf mit dem PIC-Controller. Durch die PIC-Kompatibilität stehen auch für den SX52 viele Ressourcen offen. Mit bis zu 100 MHz ist der SX52 der schnellste Microcontroller seiner Klasse.

Letztendlich ausschlaggebend für die Wahl des SX52, war die erhältliche Entwicklungsumgebung der Firma Parallax. Die Software ermöglicht eine schrittweise Verfolgung des Programmablaufs, und spiegelt dabei die Arbeitsspeicherbelegung auf den Monitor. Näheres dazu im Kapitel 5.6.1.

Ein tabellarischer Vergleich der Microcontroller findet sich nochmals in Tabelle 42.

Die erste Wahl im Detail

Der SX52[Inc00c] (52 steht für die Anzahl an Pins) birgt acht Seiten zu 512 12-Bit Worten Programmspeicher für den Programmcode, und 16 Bänke a 16 Register (8-Bit Register) sowie eine Spezialbank zur Konfiguration. Das heißt es stehen 8 Kilobyte[2] für Programmcode und 2^8 Register zu Laufzeit zur Verfügung. Für Unterprogrammaufrufe steht ein Stapel mit einer Tiefe von acht Einträgen zur Verfügung. Der Programmspeicher entspricht physikalisch einem Flashrom, sodaß zum Beispiel weiterentwickelte Versionen auch im nachhinein eingespielt werden können. Diese Möglichkeit erleichtert auch die Entwicklung insofern, als das Programme nicht erst in simulierter Umgebung getestet werden müssen, sondern sofort im Einsatzumfeld erprobt werden können, ohne große Kosten zu verursachen.

Der SX52 hat fünf 8-Bit Ports, die sowohl als Input als auch als Output nutzbar sind. Es existiert ein Stromsparmodus (Sleep-Mode) der zur Laufzeit an- und durch externe Quellen wieder abschalbar ist. Der Microcontroller hat eine Interruptfunktion, die ebenfalls zur Laufzeit aktivier- und deaktivierbar ist. Einer der fünf Ports kann als Interrupt oder Weck-Port konfiguriert werden. Das heißt ein Signal an diesem Port löst einen Interrupt aus bzw. ,,weckt" den Microcontroller aus dem Stromsparmodus.

5.4.2 Peripherie

Im folgenden werden die genutzten Bauelemente und Funktionalitäten der Platine beschrieben.

[2]Eine Instruktion benötigt 12 Bit, daher gilt für den Programmspeicher das ein Byte 12 Bit hat

Microcontroller	*SX52*	*68HCxx*[3]	*PIC16F87x*
Architektur	RISC	CISC	RISC
Arbeitsspeicher (in Bytes)	256	1000	256
Programmspeicher	$4kB$	$20kB$	368 words
Anzahl I/O Pins	5×8	$7 \times 8 + 1 \times 8$ input only	5×8
Taktfrequenz	$50MHz$	$4MHz$	$20MHz$
Interrupt intern/extern	ja / ja 8 Pins	ja / ja 1 Pin	ja / ja 14 Pins
Serielle Schnittstelle	nein	ja	nein
Strom im Sparmodus	$1\mu A$	/	$5\mu A$
ca. Preis	7 Euro	14 Euro	7Euro

Tabelle 5.1: Vergleich der drei favorisierten Microcontroller

Kommunikation Neben dem SPI-Bus zur internen Kommunikation stehen eine serielle Schnittstelle nach RS232-Standard und, wie bereits im vorangegangenem Kapitel 4 beschrieben, ein CAN-Controller[Inc00b] für die externe Kommunikation zur Verfügung. Die serielle Schnittstelle ist direkt über einen Wandlerbaustein (MAX202CSE) mit dem Microcontroller verbunden. Durch die Anbindung des RXD- (RXD = Receive Data) und CTS-Signals (CTS = Clear To Send) an den interruptfähigen Port des Microcontrollers ist eine Aktivierung der Platine via serieller Schnittstelle möglich.

Digital-Analog-Wandler Der am SPI-Bus angeschlossene Digital-Analog-Wandler LTC1446[Inc96] von Linear Technology dekodiert eine 2×12 Bit lange Nachricht und wandelt sie in zwei analoge Signale. Die Ausgabe ist eine lineare Funktion des Betrags der binären Eingangszahlen, so daß 000hex für Null Volt und FFFhex für zwölf Volt stehen.

Analog-Digital-Wandler Der LTC 1594[Inc00a] von Linear Technology enthält einen 4-Kanal Multiplexer über den einer von vier Eingängen an den Analog-Digital-Wandler geführt werden kann. Der LTC 1594 ist ebenso wie der Digital-Analog-Wandler über den SPI-Bus an den Microcontroller angeschlossen. Zur Konfiguration werden anfangs vier Bit übertragen die zum einen als Aktivierung und zum anderen für die Auswahl des Multiplexereingangs dienen. Wenn ein Eingang ausgewählt ist, dann codiert der Analog-Digital-Wandler die anliegende Spannung bis maximal 12 Volt in 12 Bit. Die Abtastrate beträgt 16,8 kHz.

Low-Side-Switch Mit dem Siemens Smart Octal Low Side Switch TLE 6230 GP [AG99] stehen über den SPI-Bus zwei Acht-Bit Schalter zur Verfügung. Vier der Acht Anschlüsse sind PWM[4]-bar (Mehr über die Puls Weit Modulation in Kapitel 5.7). Die notwendigen Signale für die PWM gehen direkt vom Microcontroller aus (also ohne den Umweg über den SPI-Bus). Der TLE hat einen Kurzschlußschutz integriert.

Serial Peripheral Interface (SPI) Die große Anzahl an Bauelementen macht eine sparsame Nutzung der I/O-Ports am Mikroprozessor nötig. Zu diesem Zweck sind der CAN-Bus-Chip, zwei Low-Side-Switches, ein Analog-Digital-Wandler und ein Digital-Analog-Wandler über ein SPI-Bus-System mit dem Microcontroller verbunden. Jeder dieser Bausteine hat einen Chipselect-Eingang der bei einer logischen Null den jeweiligen Baustein aktiviert. Der SPI-Bus wickelt die Kommunikation der Komponenten über zwei Leitungen SI (SPI-Input) und SO (SPI-Output) seriell ab. Die Kommunikation läuft immer zwischen einem Master (hier der SX52) und dem durch die jeweilige Chip-Select-Leitung ausgewählten Baustein ab. Der Master übernimmt auch die Taktung über eine separate CLK-Leitung.

In dieser Applikation wird das jeweilige Chip-Select über einen Multiplexer[Inc99] gesetzt. Diese Lösung spart nicht nur wertvolle Pins am Port des Microcontroller, sie garantiert auch, das nur genau ein Gerät auf den Datentransfer reagiert. Dies macht die Applikation also ein Stück sicherer.

Brücken Für verbrauchsstarke Gleichstromgeräte sind zwei H-Brücken der Firma Infineon auf der Platine enthalten. Der TLE 5206-2[Tec99] kann bis zu 40 Volt Spannung bei fünf Ampere liefern. Er verträgt dabei kurzzeitige Stromspitzen von bis zu sechs Ampere. Die aus zwei Bausteinen resultierenden vier Anschlüsse, können für vier in eine Richtung steuerbare Motoren oder zwei in beide Richtungen steuerbare Motoren verwendet werden. Eine weitere Anwendung wäre der Anschluss von Servomotoren.

Strommessung an der Brücke Über Jumper läßt sich eine Rückkopplung der Brückenausgänge mit den Eingängen des Analog-Digital-Wandlers herstellen. Der Strom an der Brücke wird über einen Widerstand geführt, die abfallende Spannung mit einem Operations-

[4]PWM: Puls Weit Modulation oder engl. Pulse Width Modulation

verstärker verstärkt und dann im Analog-Digital-Wandler gemessen. Das heißt, es existiert eine direkte Möglichkeit zur Messung der Leistungsaufnahme der an der Brücke angeschlossenen Verbraucher (z.B. Motoren).

Digitale Eingänge Mit dem ULN2004A gibt es über 7 Anschlüsse einen direkten Zugang zum Microcontroller. Der sieben Darlington Schaltungen enthaltende Baustein von STMicroelectronics akzeptiert eine Eingangsspannung von fünf bis zu acht Volt, je nach Eingangsstrom. Drei der Eingänge werden an den interruptfähigen Port des Microcontrollers durchgeschaltet. Dadurch ergibt sich eine weitere Möglichkeit die Platine von außerhalb zu „wecken".

Speicher Mit dem FM24C64[Cor02] steht ein 64 Kilobyte großer seriell bedienbarer Speicher zur Verfügung. Der Speicher ist unterteilt in 8.192 $\times$ 8 Bits, so daß jeweils acht Bit seriell in den Speicher geschrieben werden können. Die Speicherzelle werden also mit einer 13-stelligen Binärzahl adressiert ($2^{13} = 8.192$). Der FM24C64 ist über ein sogenanntes „Two-wire Interface" an den SX angeschlossen. Dabei generiert der SX als Master das Taktsignal für die serielle Übertragung.

LED Um ein optisches Feedback (zum Beispiel für den Fall einer Fehlfunktion) für den Programmierer zu ermöglichen, sind zwei LEDs[5] an den Microcontroller angeschlossen. Eine davon ist blau, die andere grün.

[5]LED = Light Emitting Diode

Tabelle 5.2: Zusammenfassung der verwendeten IC's

Komponente	*Hersteller-bezeich-nung*	*Hersteller*	*ca. Preis in Euro*
Brücken	TLE 5206-2	Infineon	4.33
Spannungswandler	LM2594	National Semiconductor	5,09
Operationsverstärker	LM324	National Semiconductor	0,31
Darlington Array	ULN2004A	STMicroelectronics	0,66
Microcontroller	SX52	Ubicom	7
Low Side Switch	TLE 6230 GP	Infineon	3.79
RS232	Max202CWE	Dallas Semiconductor	3,26
CAN-Controller	MCP2510	Microchip	4.25
Analog-Digital-Wandler	LTC 1594	Linear Technology	11,30
Digital-Analog-Wandler	LTC 1446	Linear Technology	12,60
Multiplexer	MM74HC138	Fairchild	0,36

Die Preise sind den Katalogen von RS-Components[RC], Farnell[Far] und Bürklin[Bür] entnommen.

5.5 Anforderungen an die Assemblerprogramme

Im folgenden werden die Auswirkungen der Vorgaben auf die Assembler-Programme des Microcontrollers beschrieben.

5.5.1 Bedienbarkeit

Was bedeutet Bedienbarkeit im Bezug auf die Programme des Microcontrollers?

Ziel ist es, dem Entwickler eine Art Bibliothek zur Ansteuerung der einzelnen Platinenkomponenten anzubieten. Folgende Punkte sind dabei mit Rücksicht auf die Assemblerumgebung weitest möglich zu beachten:

- Die Programmpakete für die einzelnen Platinenkomponenten müssen als Module verfügbar sein, die gegebenenfalls vom Entwickler eingebunden werden können. Nicht benötigte Module belegen dann keinen Speicherplatz.

- Die einzelnen Module müssen sich den Arbeitsspeicher so teilen, das Keines das jeweils Andere stört bzw. Inkonsistenzen zur Laufzeit hervorruft. Das heißt, daß eine strikte Arbeitsspeichertrennung vorhanden sein muß.

- Es müssen Konventionen für den Austausch von Daten zwischen den Modulen entwickelt werden.

- Der Programmspeicher muß möglichst effizient von den Modulen genutzt werden.

- Innerhalb der Module sollte es keine weiten Verzweigungen durch Unterprogrammaufrufe geben, um dem Entwickler (und damit dem Nutzer der Module) möglichst viel Freiraum auf dem Unterprogramm-Stapel zu gewähren. Der Unterprogrammstapel hält maximal acht Einträge.

- Konstante Einstellungen die sich aus der Architektur der Platine ergeben sollten dem Entwickler durch ,,sprechende" Konstantennamen einfach handhabbar gemacht werden.

5.5.2 Sicherheit

Bei der Softwareentwicklung der Platinen müssen eventuelle Fehlfunktionen berücksichtigt und adäquat behandelt werden, ohne die Kommunikationsebene zu stören. Desweiteren muß es einen sicheren Zustand geben in den die Platine bei etwaiger Fehlfunktion versetzt wird um Kurzschluß, unnötigem Leistungsverbrauch oder ähnlichem vorzubeugen.

5.5.3 Kommunikation

Die Modularität des Systems bringt eine Vielzahl von Kommunikationsanforderungen mit sich. So müssen die Einzelnen Platinen in Kontakt mit der Steuereinheit treten. Auf der Platine muß der reibungslose Austausch zwischen den Programmmodulen sichergestellt sein, ohne Inkonsistenzen hervorzurufen. Auf Hardwareebene werden einzelne Komponenten über ein serielles Protokoll angesprochen (SPI). Und nicht zuletzt muß der Microcontroller die zeitkritische serielle Schnittstelle betreiben (sofern das Modul eingebunden ist).

5.5.4 Erweiterbarkeit

Um auch zukünftige Weiterentwicklungen der Peripherie berücksichtigen zu können, muß die Programmierung der Komponenten einer klaren Konvention folgen. So können auch neue Implementierungen diesem Muster folgen und der Entwickler hat eine geringe Einarbeitungszeit. Eine Schnittstelle im Sinne einer Hochsprache ist hier nicht von Nöten (und auch schwer realisierbar), da die geringe Stapeltiefe und der begrenzte Programmspeicher ohnehin eine geringere Verschachtelungstiefe der Unterprogramme erzwingen.

Abbildung 5.3: Der SX-Key

5.6 Konzeption

5.6.1 Entwicklungsumgebung

Wie bereits oben beschrieben war einer der Gründe für die Wahl des SX52 seine gute Handhabbarkeit bei der Entwicklung. Dies hängt mit der erhältlichen Entwicklungsumgebung[Inc98] der Firma Parallax zusammen. Die Entwicklungsumgebung besteht aus der SX-Software (SX.exe) und dem SX-Key (siehe Abbildung 5.3).

Mit Hilfe des SX-Key spielt man die Software in den Programmspeicher des SX52 ein. Er stellt außerdem, im entsprechenden Modus betrieben, die Debugfunktionen für die SX-Software bereit (Spiegeln und Ändern der Register und Zähler, sowie externe Taktung). Der SX-Key ist über die serielle Schnittstelle an den PC angeschlossen. Der Anschluß an den Microcontroller erfolgt über eigens dafür vorgesehene Anschlußpins auf der Platine.

Die SX.exe stellt sich nach dem Start als Editor dar. Der Assembler-Interpreter und die Debugfunktionen lassen sich über einen Menüpunkt in der Kopfleiste ausführen.

Die Debugfunktion ermöglicht es, die Ausführung von Programmen schrittweise zu verfolgen und dabei die Registerinhalte zu überwachen. Desweiteren können bestimmte Register oder gar Bits formatiert (als binäre, hexadezimale oder dezimale Zahl in einem zusätzlichen Fenster) angezeigt werden (Abbildung 5.4) . Bei jeder Änderung durch einen Be-

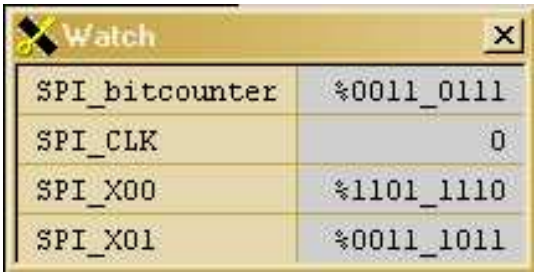

Abbildung 5.4: Formatierte Anzeige von Registern

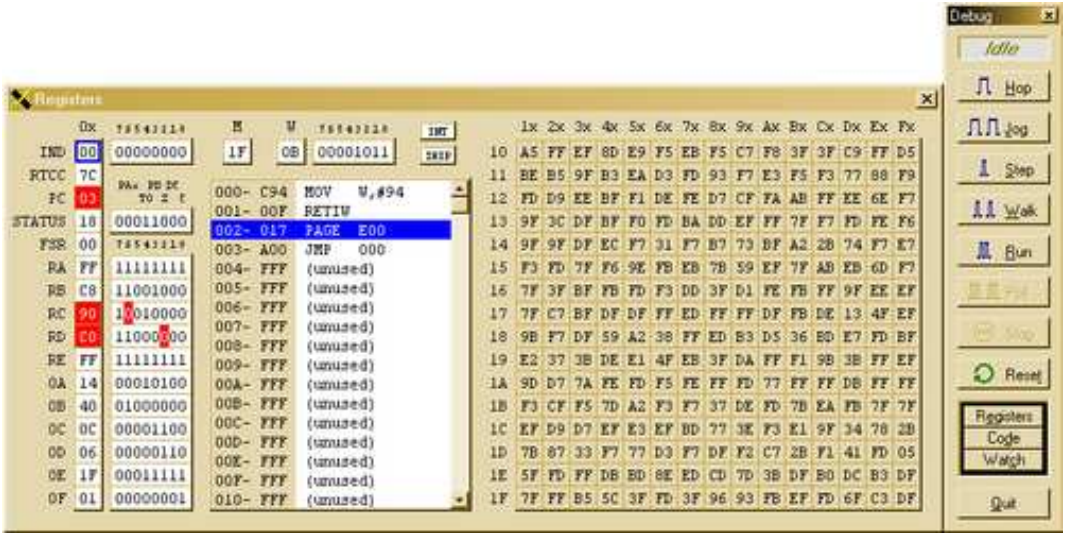

Abbildung 5.5: Registerfenster mit Debugleiste

fehl werden die betroffenen Register rot umrandet. Auf diese Art läßt sich schnell feststellen, ob das Programm in den korrekten Bereichen des Arbeitsspeichers arbeitet. Ebenfalls zur Laufzeit kann man durch anklicken des entsprechenden Registers die Registerinhalte ändern. Abbildung 5.5 zeigt das Registerfenster der Entwicklungsumgebung im Debugmodus. In der Tabelle sind alle Register aufgezählt. Zur Linken sieht man die Inhalte der globalen Register sowohl in binärer als auch in hexadezimaler Form. Desweiteren sind der Programmzähler (PC) und der Akkumulator (W) am oberen Rand zu sehen.

5.6.2 Funktionsweise eines SX52 Assemblerprogramms

Im folgenden wird die Funktionsweise eines SX52 Assemblerprogramms grob umrissen, um ein besseres Verständnis der folgenden Erklärungen zu

ermöglichen.

Die Programme des SX52 sind in einem 64 Kilobyte großem Programmspeicher untergebracht. Jedes Byte hat eine Länge von zwölf Bit. Ein Byte repräsentiert eine Instruktion für den Microcontroller. Dabei teilt sich ein solches Byte je nach Befehl in Operationscode und Argumente. Es handelt sich beim SX52 um eine RISC-Architektur, das heißt der Anteil des Operationscodes an einer Instruktion ist (fast) immer gleich. Die Adresse der aktuell auszuführende Zeile entnimmt man dem Programmzähler (zu engl. Programm Counter, abgekürzt: PC).

Ein Programm beginnt an der höchsten Stelle des Speichers, also an der Stelle FFF (Hexadezimal). Dort wird die Einsprungadresse des Hauptprogramms eingelesen und in den Programmzähler geschrieben. Die Einsprungadresse wird auch als „reset"[6] bezeichnet. Der Befehl an der aktuellen Stelle wird eingelesen und, nachdem der Programmzähler um Eins iteriert wurde, ausgewertet.

Ein solches Programm könnte nun bis zur höchsten Stelle durchlaufen und dann wieder von der Stelle FFF (hexadezimal) aus zurückspringen. In der Praxis wird das jedoch selten so gehandhabt. Folgende Ereignisse können den eben beschriebenen Kreislauf unterbrechen:

Rollover-Interrupt Der Rollover-Interrupt[7] unterbricht das Programm, schreibt den aktuellen Inhalt des Programmzählers, sowie den Akkumulator und die Register FSR [8] und STATUS[9] in eigens für die Interruptbehandlung vorgesehene Register. Danach werden die Befehle ab der Adresse 000hex ausgeführt bis das Ende des Interrupts durch einen speziellen Rücksprungbefehl angezeigt wird (hier: RETIW oder RETI). Diese Befehle sorgen nun dafür, daß der Programmzähler, der Akkumulator und die Register FSR und STATUS wieder auf die zuvor gespeicherten Werte zurückgesetzt werden. Die Befehle von der Adresse 000hex bis zum Rücksprungbefehl werden im folgenden als Interrupt-Routine bezeichnet. Während der Ausführung einer Interrupt-Routine wird kein weiterer Interrupt erkannt.

SLEEP-Befehl Durch den SLEEP-Befehl wird die Clock ausgeschaltet und

[6] engl.reset: wiederherstellen. In diesem Kontext soll der Anfangszustand des Programms wieder hergestellt werden

[7] engl.roll over: überschlagen, vorne überkippen; engl. interrupt: die Unterbrechung

[8] FSR: File Select Register, enthält den Zeiger auf die Aktuelle Registerbank

[9] STATUS: enthält den arithmetischen Status der ALU, sowie die Page Select Bits. Mehr dazu in Abschnitt 46

der Microcontroller nur noch minimal mit Strom versorgt. Das Programm hält an.

Multi-Input-Wakeup-Interrupt Der Multi-Input-Wakeup-Interrupt wird durch ein Signal am Port B ausgelöst. Er bewirkt entweder das ,,aufwachen" des Microcontrollers, wenn dieser im SLEEP-Modus war, oder er startet die Interrupt-Routine.

Endlosschleife Man kann das Rrreichen der höchsten Speicherstelle auch durch eine eigene Endlosschleife verhindern, die während des Programmablaufes nicht verlassen wird.

Neustart durch ab- und einschalten der Versorgungsspannung Natürlich kann man den Microcontroller auch aus- und wieder anschalten um das Programm zu unterbrechen. Man führt also einen Hardware Reset durch.

Während des Programmablaufes werden Daten eingelesen, bearbeitet und ausgegeben. Dies geschieht in den Registerbänken. Man Unterscheidet im Fall des SX52 zwischen globalen und zu adressierenden Registerbänken zu je 16 Byte (1 Byte = 8 Bit). Die globalen Register sind jederzeit adressierbar. Sie enthalten unter anderen das STATUS-Register, das FSR-Register und die Port-Register. Im FSR (engl. File Select Register, etwa Dateiauswahlregister) stehen die ersten vier Bit der zu adressierenden Registerbank. Um eine Bank zu adressieren schreibt man entweder die entsprechende Adresse direkt in das FSR-Register oder nutzt den BANK-Befehl gefolgt von einer drei-Bit Nummer, das vierte Bit im FSR wird dabei immer auf Null gesetzt. Um jetzt trotzdem die ungeraden Registerbänke adressieren zu können muß der Programmierer das entsprechende Bit im FSR-Register setzten. Diese umständliche Handhabe ist auf die Kompatibilität mit dem SX32 zurückzuführen, der nur acht Registerbänke zur Verfügung stellt.

Unterprogrammaufrufe

Unterprogrammaufrufe werden mittels CALL-Befehl realisiert. Der SX52 enthält einen bis zu acht Einträge haltenden Stapel um Unterprogramme verschachtelt aufzurufen. Beim Aufruf eines Unterprogramms wird die Rücksprungadresse auf den Stack gelegt und der Programmzähler auf den Wert des CALL-Arguments gesetzt. Das Argument kann jedoch nur einen Adreßraum von 256 Byte ansprechen (Argumentlänge: acht Bit), drei der restlichen vier Bit für eine vollständige Programmspeicheradresse stehen

im STATUS Register. Das vierte Bit wird automatisch auf Null gesetzt. Für die Programmierung bedeutet dies, daß mit einem Unterprogrammaufruf innerhalb einer Programmspeicherseite nur Unterprogramme in der unteren Hälfte aufgerufen werden können. Im Abschnitt 5.6.4 wird ein Konzept beschrieben, das diese Einschränkung geschickt umgeht.

Arbeitsregister

Der SX52 nutzt in der Summe 17 Registerbänke zur Verarbeitung von Daten zur Laufzeit. Die Register teilen sich in eine globale Bank und 16 indirekt adressierbare Bänke auf (siehe Abbildung 5.6). Jede Bank enthält 16 Register. Für die Adressierung der globalen Register sind keine Vorkehrungen nötig. Um ein solches Register zu adressieren benutzt man im Quelltext einfach die absolute Adresse des Registers (direkte Adressierung). Um die 16 ,,normalen" Registerbänke zu referenzieren, muß zuvor die entsprechende Registerbank über das FSR angegeben werden. Erst dann kann man die 16 Register der entsprechenden Bank wählen. Das heißt, das nur die unteren vier Bit von Relevanz sind. Um eine solche (indirekte) Registeradressierung anzukündigen, müssen die ersten vier Bits (MSB[10]) eine 1 darstellen. Steht also im Bit Nummer vier einer Registerreferrenz eine 1, so werden die Bits 0 bis 3 für die Adressierung eines globalen Registers genutzt, sonst wird ein Register aus der im FSR spezifizierten Bank entnommen.

```
; W bezeichnet den Akkumulator
MOV  $1A,W ; schreibt von W in das globale Register $1A
BANK $60   ; stellt FSR auf Bank sechs
SETB FSR.4 ; stellt FSR auf Bank sieben
MOV  $0A,W ; schreibt von W in das Register Nr.7A
```

In der Summe stehen dem Benutzer 256 Register plus 15 globale Register zur Verfügung. Von den globalen Registern sind die ersten fünf Register für spezielle Aufgaben vorgesehen (unter anderen FSR und STATUS-Register),dann folgen die fünf I/O-Ports die auch als Register nutzbar sind, zuletzt verbleiben noch sechs ,,freie" Register.

5.6.3 Assembler-Direktiven

Der zum SX52 gelieferte Assembler stellt eine Reihe von Direktiven zur Verfügung, die, klug genutzt, die Programmierung vereinfachen. Eine Direktive wird nicht zur Laufzeit, sondern bereits vor der Programmierung

[10]Most Significent Bits: höchstwertige Bits

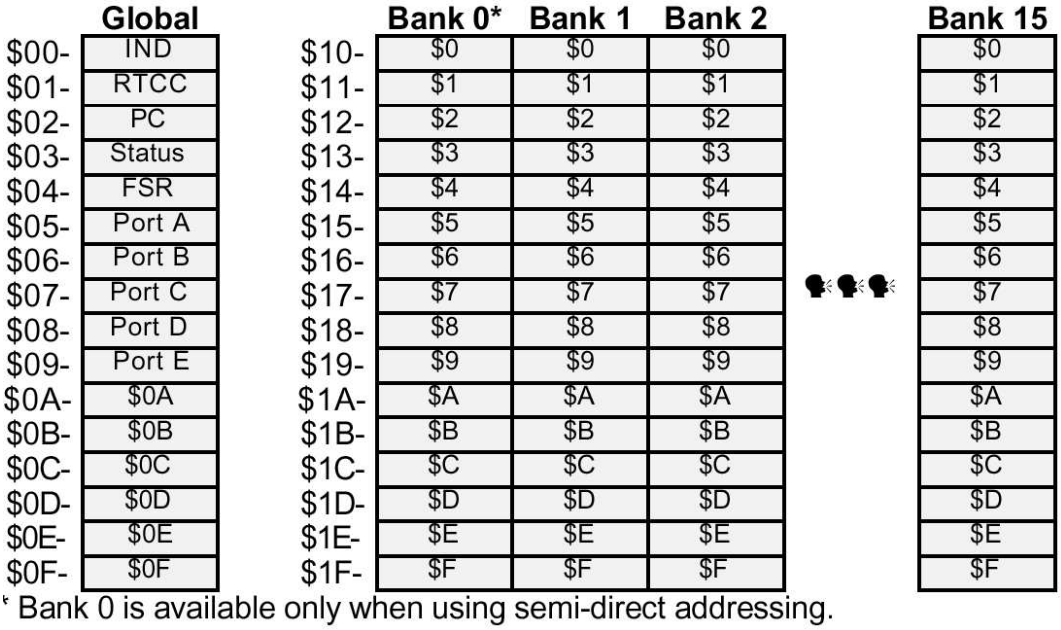

Abbildung 5.6: Aufbau der Registerbänke im SX52

des Programmspeichers ausgeführt. Sie dient zum Beispiel zur Organisation des Programmcodes, zur Benennung von Variablen und Konstanten, oder zur arithmetischen Überprüfung einiger Programmkomponenten. Im Folgenden werden einige der Direktiven beschrieben. Ausserdem wird gezeigt, wie sie genutzt werden können, um die zuvor formulierten Ziele in der Implementierung zu erreichen.

EQU-Direktive

Mit der EQU-Direktive werden Konstanten oder Registern Namen zugeordnet, die dann beim Programmieren statt der Zahl bzw. Registernummer genutzt werden kann.

```
LED EQU     RB.5        ; 6.Bit auf Port B wird mit LED benannt
FOO EQU     $0a         ; Register 0a wird mit FOO benannt
BAR EQU     #%0000_0011 ; Konstante BAR hat den Wert drei
```

ORG-Direktive

Die ORG-Direktive dient der genauen Plazierung von Programmcode im Programmspeicher. Man definiert eine Adresse zwischen 000_{hex} und

$FFFhex$, an welcher der Programmcode dann im Programmspeicher stehen soll .

Beispiel:

```
ORG  #000   ;der folgende Code wird ab Adresse 000 eingefügt
# Es folgt die Interrupt-Routine
...

ORG  #200   ;der folgende Code steht ab Adresse 200 (hex)
MOV  $01, #2
...
```

DS-Direktive

Die DS-Direktive benennt einen Programmspeicherbereich. Die Größe des Bereichs in Byte wird als Argument übergeben. Kombiniert man diesen Befehl mit der ORG-Direktive, so kann man auch den Ort der Reservierung festlegen.

```
ORG #20
FOO DS  1 ;reserviert ein Byte unter FOO
BAR DS  2 ;reserviert zwei Byte unter BAR
```

MACRO-Direktive

Ein nützliches Werkzeug bei der Speicherorganisation ist die MACRO-Direktive. Programmcode, der in ein Macro eingebunden wird läßt sich leicht aus dem Programm entfernen und wieder einsetzen. Diese Eigenschaft ermöglicht es, Programmteile genau zu plazieren und so Adressierungsbeschränkungen zu umgehen. Mehr dazu in Kapitel 5.6.4

```
MACRO   FOO ; Macro namens FOO

MOV M,#BAR  ; Programmcode

ENDM ; Ende dees Macros
.
.
.
ORG 200
FOO     ; Macro FOO wird bei 200 (hex) eingefügt
```

$-Direktive

Das Dollarzeichen liefert die Adresse der aktuellen Bank zurück.

```
FOO EQU $   ;FOO enthält die aktuelle Adresse
```

5.6.4 Assembler-Module

Ein Ingenieur, der die Aufgabe erhält ein SX-Assemblerprogramm zu schreiben, wird mit einigen gefährlichen Schwierigkeiten konfrontiert. Um potentielle Fehler beispielweise mit Unterprogrammaufrufen. Speicherbelegung, Registerzugriff etc. zu vermeiden, bietet sich die im Rahmen dieser Arbeit entwickelte Modulstruktur an.

Bei der Konzeption der Module galt es, die potentiellen Schwierigkeiten mit Unterprogrammaufruf, Verteilung der Programmteile auf dem Speicher, sowie den Zugriff auf Register für den Entwickler möglichst einfach handhabbar zu machen. Um dies zu erreichen eignet sich eine Modulstruktur die durch geschicktes Einsetzen der Makro-Direktiven die oben genannten Schwierigkeiten umgeht.

Modulstruktur

Man unterteilt jedes Modul in verschiedene Segmente. Begreift man nun ein Assemblerprogramm als Automat mit den Zuständen Initialisierung, Hauptfunktion und Interrupt (siehe Abbildung 5.7) , so repräsentieren grob betrachtet die einzelnen Segmente die Zustände des Automaten.

Zusätzlich bedarf es für die Datenorganisation noch eines CALL- und eines DATA-Segments. Im folgenden die Segmente im Einzelnen.

CODE-Segment: Im CODE-Segment befinden sich die eigentlichen Routinen eines Moduls. Jede Routine beginnt dabei mit einer Einsprungmarke und endet mit einem Rücksprungbefehl für Unterprogramme. Der Rücksprungbefehl setzt die durch einen CALL-Befehl gespeicherten Werte in STATUS-, FSR-Register und Programmzähler wieder ein, und führt das Programm von dort an weiter aus.

CALL-Segment: Das Call-Segment dient dazu die 256-Byte Grenze des CALL-Befehls zu umgehen. Dieses Segment muß jeweils in den unteren 256-Byte einer Programmspeicherseite liegen. Im CALL-Segment werden Funktionsnamen mit den entsprechenden Funktionen im CODE-Segment verknüpft. Der Trick liegt dabei in der Verwendung des JMP-Befehls. JMP springt an die im Argument über-

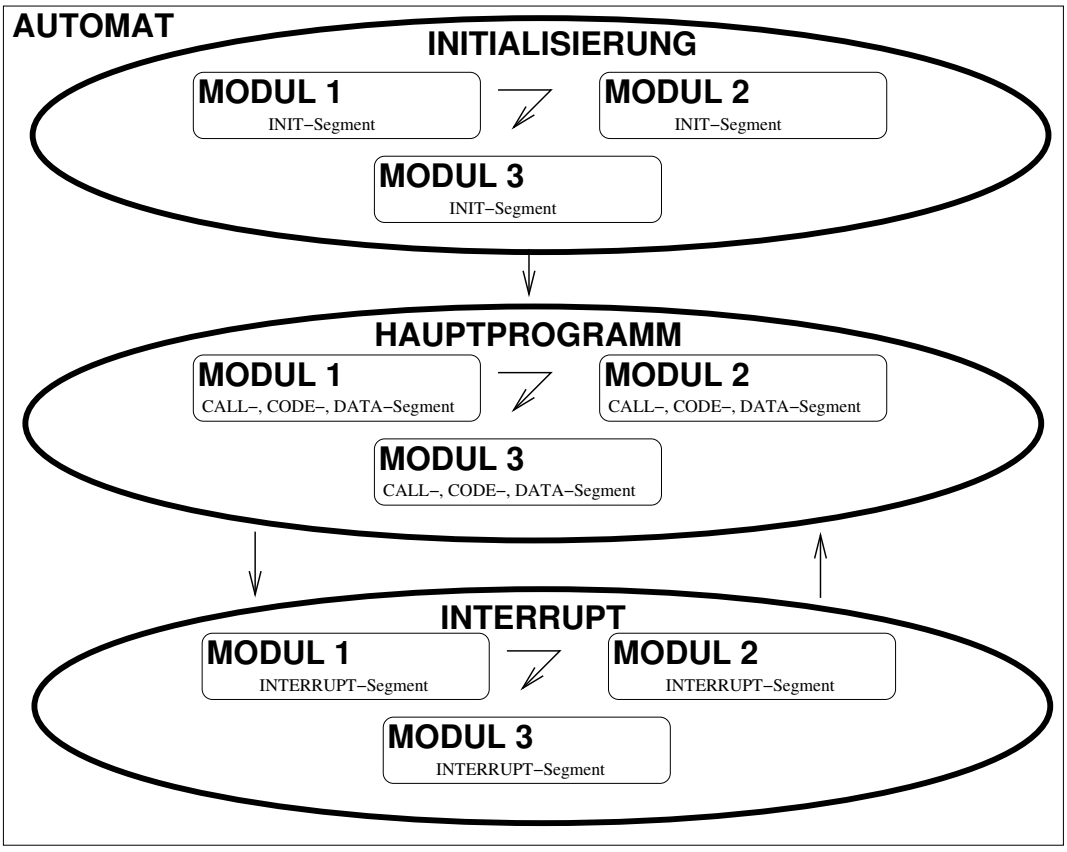

Abbildung 5.7: Ein Assemblerprogramm als Automat

gebene Stelle im Programmspeicher. Da das Argument des JMP-Befehls neun Bit lang ist, können die vollen 512 Byte einer Speicherseite referenziert werden. Das CALL-Segment ermöglicht also einen Unterprogrammaufruf ohne die 256-Byte Beschränkung bedenken zu müssen. Desweiteren fügt der Assembler noch einen PAGE-Befehl ein wenn dem CALL ein @ vorangestellt wird, so daß auch die Beschränkung auf die aktuelle Seite des Speichers wegfällt.

DATA-Segment: Wie bereits oben beschrieben stehen dem Programmierer 256 Register auf verschiedenen Bänken zur Verfügung. Um mit den Registeradressierungen nicht durcheinander zu kommen, empfiehlt es sich pro Modul nur auf einer Bank zu arbeiten. Die Makros ermöglichen es bei der Programmierung auf bewußte Registeradressierung zu verzichten, indem man nur noch mit Variablennamen hantiert.

Man definiert also ein Makro in dem Platz für maximal 16 Register reserviert wird. Bei der Reservierung wird die Speicherstelle zwangsweise benannt, sodaß beim programmieren, wie bereits gesagt, nur noch die Variablen für den reservierten Platz benutzt werden können. Um dann das Makro einer Registerbank zuzuordnen, wird das Makro durch die ORG-Direktive an der entsprechenden Stelle eingefügt. Die erste Instruktion im Makro weist dann einer Variablen die aktuelle Adresse zu, so daß der Entwickler mit der Kombination BANK-Befehl und eben genannte Variablen die entsprechende Modulbank referenzieren kann ohne schon bei der Implementierung wissen zu müssen, welche Bank er eigentlich nutzt.

```
MACRO     FOO   ; Macro namens FOO
MEINEBANK $     ; Die aktuelle Bank wird referenziert
VAR1      DS1   ; Ein Register reservieren
VAR2      DS1   ; ...
...
ENDM

ORG  $20
FOO     ; Macro FOO wird bei 20 (hex) eingefügt
                ; enspricht der dritten Bank
```

INTERRUPT-Segment: Benötigt man für ein Modul auch Routinen die durch einen Interrupt ausgelöst werden, so schreibt man den da-

zu nötigen Code in ein Makro. Dieses Makro wird als Interrupt-Segment beschrieben. Die Interrupsegmente schreibt man dann hintereinander an die Adresse 000 (hexadezimal). In den Interruptsegmenten darf kein Interrupt-Rücksprungbefehl (RETI oder RETIW) stehen. Der muß am Ende der Makroliste stehen, so daß alle Segmente vor Beendigung der Interruptroutine ausgeführt werden.

INIT-Segment: Jedes Modul benötigt auch einen Bereich im Initialbereich des Gesamtprogramms zur Initialisierung seiner Variablen. Dazu definiert man zu jedem Modul ein INIT-Macro.

Zusammengefaßt könnte ein Programm nach den oben beschriebenen Konventionen dann wie folgt aussehen:

```
;M O D U L E
; Modul_1

; DATASEGMENT
    MACRO   DATA_MODUL_1
     MEINEBANK $     ; Die aktuelle Bank wird referenziert
     VAR1      DS1   ; Ein Register reservieren
     VAR2      DS1   ; ...
    ENDM

; INTERRUPTSEGMENT
    MACRO   INTERR_MODUL_1
    ;Anweisungen die bei Interrupt ausgeführt werden sollen
    ENDM

; INITSEGMENT
    MACRO   INIT_MODUL_1
    BANK MODUL_BANK_1  ; Auf Modulspezifische Bank wechseln
    ;Initialisierung der Variablen
    ENDM

; CALLSEGMENT
    MACRO   CALL_MODUL_1
     routine_1
     JMP _routine_1
    ENDM

; CODESEGMENT
   MACRO   CODE_MODUL_1
   _routine_1 ;Sprungmarke für CALL-Segment
```

```
   ; der eigenliche Code
   ...
   ENDM

; Modul_2
...
; CALLSEGMENT
    MACRO   CALL_MODUL_2
     routine_2
     JMP _routine_2
    ENDM

; CODESEGMENT
   MACRO   CODE_MODUL_2
   _routine_2 ;Sprungmarke für CALL-Segment
   ; der eigenliche Code
   ...
   ENDM
...
; Modul_3
...

;P R O G R A M M
Hier wird jetzt der Programmcode auf den Speicher verteilt

ORG $000
;Interrupt
INTERR_MODUL_1
INTERR_MODUL_2
INTERR_MODUL_3
RETI ;aus Interrupt herausspringen

ORG $500
main        ;anfängliche Einsprungmarke
;Initialisierung
;Interrupt abschalten
...
;allgemeine Konfiguration
...
INIT_MODUL_1
INIT_MODUL_2
INIT_MODUL_3
;Interrupt wieder einschalten
```

```
mainprog   ; Hauptteil des Programms
...
@CALL routine_1
...
@CALL routine_2
...
JMP mainprog

ORG $600
CALL_MODUL_1
CALL_MODUL_2
CODE_MODUL_1
CODE_MODUL_2

ORG $800
CALL_MODUL_3
CODE_MODUL_3
```

Datenaustausch zwischen den Modulen

Um den Austausch von Daten sicherzustellen, werden zwei Bänke reserviert. Jedes Modul entnimmt dort seine Eingabewerte und legt die Ergebnisse dort wieder ab.

5.7 Implementierungen

Im Folgenden werden beispielhaft Module für einige Bauelemente beschrieben.

Serielle Schnittstelle: Die serielle Datenübertragung ist in der Norm RS232 festgelegt. Der Standard definiert die nötigen Physikalischen Größen, sowie die Verkabelung. Das beschriebene Modul implementiert eine vereinfachte Version (d.h. es werden nicht alle Kabel genutzt), die für unsere Zwecke vollkommen ausreichend ist. Danach können zwei Computer über die Datenleitungen TxD (Transmit Data) und RxD (Receive Data) miteinander kommunizieren. Ein Byte wird in unserem Fall im 8n1 Modus gesendet. Das heißt es werden acht Bits, sowie ein vorangestelltes Startbit und ein nachgestelltes Stopbit gesendet. Eine Fehlerüberprüfung (Parity) findet bei der Implementierung in diesem Fall nicht statt. Es handelt sich bei der RS232 um eine asynchrone Übertragung die, im Gegensatz zur synchronen Übertragung, ohne separates Taktsignal erfolgt. Ein Empfänger muß das eintreffende Signal mit dem mehrfachen der möglichen Taktfrequenz abtasten und so das Taktsignal rückgewinnen. Um sicherzustellen, daß

ein Eingangssignal registriert wird, wählt man als Abtastrate das vierfache der Zielfrequenz ab. Um beispielsweise bei einer Zielfrequenz F von $115.200Baud$ eine Änderung der eingehenden Flanke festzustellen, wird der Eingangszustand mit $460,8kHz$ abgetastet. Um ein genaues Timing zu realisieren nutzt die Implementierung den Rollover Interrupt. Der Rollover Interrupt löst alle 256 Zyklen einen Interrupt aus. Dies entspricht bei einer Taktfrequenz T von $50MHz$ einem Takt von $195.312,5Hz$. Reduziert man nun die Anzahl der Zyklen bis zum nächsten Interrupt, so läßt sich die Interrupt-Frequenz regulieren. Der Befehl RETIW macht das möglich. RETIW setzt den Interruptzähler auf den im Akkumulator W übergebenen Wert. So läßt sich mit 108 Zyklen pro Interrupt, bei einer Zyklenlänge von $20ns$, ein Takt von ungefähr $115.200Baud$ realisieren (vierfache Abtastrate). In diesem Beispiel wird davon ausgegangen, das im Interrupt keine weiteren Verzögerungen auftreten. Wenn jedoch mehrere Module mit Interrupt-Segmenten genutzt werden, so muß die Verzögerung beim setzen des neuen Rollover Interrupt berücksichtigt werden. Für eine Anzahl n von Zyklen im Interrupt ergibt sich der zu übergebende Wert w für der Akkumulator wie folgt:

$$w = \left\lfloor \frac{T}{4 * F} \right\rfloor - n$$

mit $w \gg 0$.

$w \gg 0$ deutet darauf hin das die Interruptroutine nicht zu groß ausfallen sollte, da sonst die Ausführung sonstiger Programme zu kurz kommt. Nutzt man also das Modul, so muß zuvor eine Baudrate für die serielle Übertragung festgelegt werden, und im Interrupt-Segment eingetragen zu werden. Für die Platine ist eine Baudrate größer als 115.200 nicht empfehlenswert, da sonst die eigentlichen Programmteile nicht mehr ausreichend Zeit zur Ausführung bekommen. Mehr zur seriellen Schnittstelle sowie ihren Nutzungsvarianten findet sich in [Sem].

Puls Weit Modulation: Mittels Puls-Weit-Modulation läßt sich an digital angesteuerten Geräten analoges Verhalten erzeugen. Dabei wird die Trägheit der jeweiligen elektromechanischen Gerätes ausgenutzt um die digitalen Impulse zu einer monoton steigenden Funktion zu „wandeln". Man unterteilt ein Zeitintervall I äquidistant in n kleinere Zeitintervalle z_i mit $i = 0, ..., n$. Geht man davon aus, daß ein Ausgang zwei Zustände low und $high$ einnehmen kann, und das jedes Zeitintervall z_i mit dem Zustand low beginnt. In jedem Zeitintervall z_i wählt man dann ein t_i mit $0 < t_i < |z_i|$ ab welchem der Wert des Ausgangs auf $high$ geht. Die Größe t_i bestimmt dann die Intensität mit der das am Ausgang angeschlossene Gerät betrieben wird. Im folgenden wird t_i auch Pulsweite genannt. Ein angeschlossener Gleichstrommotor zum Beispiel dreht sich bei $t_i = 1/2z_i$ nur halb so schnell als er es bei durchgehendem Stromfluß tun würde. Ändert man nun

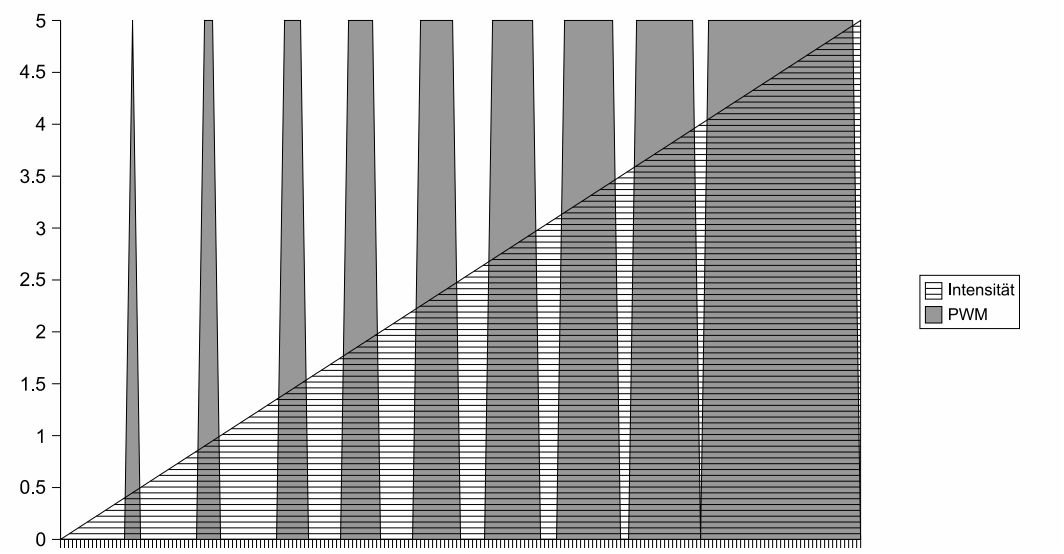

Abbildung 5.8: Puls Weit Modulation einer linearen Funktion.

t_i zur Laufzeit in die eine oder andere Richtung, so erzeugt man Dimm-Effekte. Ein Beispiel ist die Funktion $t_i = i * z_i/n$, die $t - i$ mit jedem Schritt an n annähert und damit von low nach $high$ übergeht. Ein Beispiel findet sich in Abbildung 5.8:

Die fallende Flanke des PWM-Signals markiert hier jeweils das Ende eines Intervalls z_i. Die dunklen Flächen markieren die Plusweiten t_i. Durch zunehmende Pulsweiten wird die lineare Funktion simuliert.

Auf der Platine sind die Brücken und die PWM-baren Low-Side-Switchs für die Puls-Weit-Modulation vorgesehen. Ein z_i entspricht 256 Interrupts. In jedem Interrupt wird ein Zähler iteriert bis dieser überläuft. Bei jedem Übelauf wird der Zustand auf low bzw $high$ gesetzt (je nachdem ob von Aus nach An oder umgekehrt moduliert wird). Im Hauptprogramm (CODE-Segment) wird dann der Zähler mit einer Variable verglichen, die die Pulsweite bestimmt. Je nach Anwendung kann man nun die Pulsweite verändern, indem man sie inkrementiert oder dekrementiert bzw. unterschiedlich initialisiert.

Serial Peripheral Interface: Um eines der Geräte über den SPI-Bus zu adressieren, muß das betreffende Gerät per Chip-Select aktiviert werden. Wie in 44 beschrieben erfolgt die Aktivierung der einzelnen Geräte über einen Multiplexer, der über die Pins drei bis fünf am Port D an den SX52 angeschlossen ist. Die folgenden Konstanten repräsentieren die Pinbelegungen des Port D für die einzelnen Geräte:

```
SPI_TLE1        EQU %0000_0000 ; Low-Side-Switch
```

```
SPI_TLE2        EQU %0000_1000 ; Low-Side-Switch
SPI_ADC         EQU %0001_0000 ; Analog-Digital-Wandler
SPI_DAC         EQU %0001_1000 ; Digital-Analog-Wandler
SPI_CAN         EQU %0010_0000 ; CAN-Controller
SPI_RESET       EQU %0010_1000 ; RESET aller RESET-fähigen Geräte
SPI_NOTCON      EQU %0011_0000 ; zu keinem Gerät verbunden
```

Das Bitmuster SPI_NOTCON kann genutzt werden um sicher zu gehen, das kein Gerät angewählt ist, so daß man bei der Programmierung nicht auf die Pinbelegungen auf Port D achten muß. Wenn das Gerät ausgewählt ist, erfolgt gerätespezifisch die Übertragung verschiedener Bitmuster. Im Detail werden die Geräte wie folgt behandelt:

TLE: Es werden zwei 8-Bit Wörter in den Low Side Switch übertragen. Das erste Wort Konfiguriert die Switches für die einzelnen Pins, das zweite aktiviert die Einstellung dann gegebenenfalls.

ADC: Noch bevor der Analog-Digital-Wandler per Chip-Select aktiviert wird, werden vier Bit zum auswählen des zu wandelnden Eingangs übertragen, erst nachdem Chip-Select auf low gesenkt wurde, wird nach kurzer Verzögerung das Ergebnis an den Microcontroller geschickt.

DAC: Zwei $\times$ 12 Bit werden in das Shift-Register des Digital-Analog-Wandlers übertragen und dann, beim Wechsel des Chip-Select Signals von low nach $high$, aktiviert. Das heißt die analogen Interpretation der 12-Bit-Muster liegen in Volt an den Ausgängen des Digital-Analog-Wandlers an.

CAN: Der CAN-Controller wird über einen Instruktionensatz von fünf Befehlen gesteuert. Ab der fallenden Flanke des $\bar{CS}$ Signals erwartet der Controller eine Befehlseingabe auf jeder steigenden Flanke der CLock-Leitung. Allen Befehlen gemein ist, daß in den ersten acht Bit der Befehlscode steht und dann, sofern nötig, die Adresse des betreffenden Registers im Pufferspeicher des CAN-Controllers. Zuletzt folgen die Argumente des Befehls, bzw eventuelle Rückgebewerte werden gelesen. Gesendet werden die im Pufferspeicher gesetzten Einstellungen durch den Befehl Request to Send (RTS).

Speicher Der Speicher ist über eigene Anschlüsse an den Microcontroller angeschlossen. Über eine serielle Datenleitung erfolgt der Datentransfer. Für jede Operation (ob schreiben oder lesen) , wird das Gerät über die Bits 1 bis 3 der ersten 8 Bit adressiert. In den Bits 7 bis 4 steht die Slave ID (hier 1010). Das nullte Bit zeigt an, ob eine Lese- (1) oder Schreibe-Operation (0) anschließt. Nach der Adressierung sendet der Master (hier der Microcontroller) beliebig viele acht-Bit Worte und erhält nach jedem Word eine Bestäti-

gung (engl. acknowledge) vom Speicherbaustein. Bei der Lese-Operation unterscheidet man zwischen sequentiellem Lesen und selektivem Lesen.

Beim sequentiellen Lesen werden vom Master dem oben beschriebenen Adressierungsbyte zwei Byte mit der Speicheradresse des gesuchten Wortes übertragen. Ab diesem Zeitpunkt beginnt der FM24C64 mit dem Senden des Worts aus der adressierten Speicheradresse. Nach jedem Byte wird wieder eine Bestätigung gesendet und das im Speicher folgende Wort kann sequentiell ausgelesen werden ohne erneut adressieren zu müssen.

Beim selektiven Lesen beginnt der Master wie bei einer Schreibeoperation (LSB = 0). Er schreibt das Adressierungsbyte und spezifiziert mit den folgenden zwei acht-Bit Worten die Speicheradresse. Nachdem der FM24C64 den Eingang der Speicheradresse bestätigt hat setzt der Master eine Start-Signal ab. Dadurch wird der Schreibmodus abgebrochen. Wenn der Master jetzt direckt im Anschluß das Adressierungsbyte mit gesetzten Lesebit sendet geht der FM24C64 in den sequentiellen Lesemodus über.

5.8 Fazit

Das Konzept der multifunktionalen Platinen, die potentielle Periepeheriegeräte steuern ist aufgegangen. Das dies nicht Münchner Projektteams im Mai 2002 gezeigt. Bereits jetzt sind Maschinenbauer in München dabei die Steuerung eines Lichtsystems und einer ausfahrbaren fahrzeugstütze mit Hilfe der Platine zu realisieren. Auch in Zukunft wird die Platine in München in anderen mechatronischen Projekten genutzt werden. Ebenso hat die Arbeit gezeigt wie wichtig der Austausch zwischen den Teammitgliedern und den beteiligten Disziplinen für eine adäquate Anforderungslösung ist. Während der Entwicklung wurden immer wieder Rücksprachen mit den Maschinenbauern geführt um die Implementierung an deren Bedürfnisse anzupassen.

Dennoch würde ich, könnte ich dieses Projekt mit meinem jetzigen Wissen erneut beginnen, meine Prioritäten bei der Wahl des Microcontrollers ändern. Schwerpunkte wären dann nicht mehr Preis und Geschwindigkeit, sondern C-Programmierbarkeit (also ob ein zumindest kostengünstiger C-Compiler für den Microcontroller erhältlich ist) und integrierte Kommunikationslogik. Die zeitkritischen Elemente wie serielle Schnittstelle und CAN wären dann nicht mehr zu implementieren, womit die hohe Geschwindigkeit des Microcontrollers hinfällig wird. Desweiteren könnte man auf PDA und Platinenseite die gleiche API zur Verfügung stellen.

Persönlich habe ich durch diese Studienarbeit viel über die Elektotechnik und ihre Tücken gelernt. Die Arbeit im Team, d. h. Terminabsprachen, Meilensteine der Arbeit und ihr Abgleichen mit den anderen Gebieten des Projekts (insbesondere vor der CeBit) sowie der fachliche Austausch untereinander zählen zu den reichsten Erfahrungen aus dieser Arbeit.

5.9 Ausblick

Die Grundlagen für die Nutzung der Platine sind geschaffen. Um den Gebrauch noch komfortabler zu gestalten ist es wünschenswert, eine didaktisch ausgereifte Anleitung auf Basis der bereits gehaltenen Schulung zu erstellen. Darüber hinaus ist eine ausführliche technische Dokumentation wünschenswert.

Kapitel 6

Fazit

von: Markus Weimer

Inhaltsangabe

Insgesamt wurde ein Mechatronik-Framework entwickelt, das den gestellten Anforderungen weitgehend gerecht wird. Teile des Frameworks, wie die Platine und die darauf aufsetzende Software sind bereits an der TU München im Einsatz, wo mechatronische Komponenten für den Läufer entwickelt werden. Andere Teile wie Kommunikation und das Fahrerinterface können derzeit noch nicht eingesetzt werden, da noch keine mechatronischen Komponenten zur Verfügung stehen. Deren Entwicklungsende wird den Zeitraum der vorliegenden Ausarbeitung überschreiten.

Das Autoren-Team wird das Projekt Läufer weiter begleiten, um auch die Anwendung der im Rahmen dieser Arbeit erstellten Software zu unterstützen.

Im Rahmen dieser Arbeit wurden allerdings nicht nur die im Hauptteil beschriebenen Entwicklungen betrieben, sondern auch Erfahrungen in Bereichen gesammelt, die nur einem Projekt offenstehen. Diese sollen im folgenden auch ihren Platz in dieser Ausarbeitung finden.

Gewinnung von Partnern Das Projekt Läufer setzt in seiner gesammten Entwicklung sehr stark auf die Zusammenarbeit mit externen Partnern aus Industrie und Forschung. Dies stellt zum einen die praktische Umsetzbarkeit der angedachten und dem Partner vorgestellten Entwicklung sicher, ermöglicht aber andererseits auch die Nutzung externen Know-Hows.

So reisten zwei der Autoren[1] gemeinsam nach Bern, um an der dortigen FH deren Lösung für eine Mechatronik in einem muskelgetriebenen Fahrzeug kennenzulernen. Aus den dortigen Gesprächen folgten maßgebliche Design-Entscheidungen für die Kommunikationsstruktur des Läufers. Im frühen Stadium des Projektes reisten die selben Personen nach Saarbrücken, um dort mit Spezialisten der Firma HighTec über die Möglichkeiten einer Verbindung zwischen PDA und CANBus zu sprechen.

Später besuchten wiederum zwei der Verfasser dieser Arbeit[2] die Firma SHARP in Hamburg. In einem Gespräch mit den dort für mobile Endgeräte zuständigen Managern gelang es, SHARP als weiteren Industriepartner zu gewinnen. Dies war und ist für das Projekt als ganzes ein wichtiger Erfolg, da SHARP evtl. die Versorgung des Projektes mit PDAs für die Nullserie leisten wird. Bei Kosten um die 600Euro je PDA ist dies ein nicht unerheblicher Posten bei der Frage, ob die Nullserie von zehn Fahrzeugen realisierbar ist.

Interdisziplinäre Zusammenarbeit Auch dies ist eine der Stärken des Projektes. Das Mechatronik-Team stieß als bisher letztes zum Läufer-Team, in dem sich bis zu diesem Zeitpunkt hauptsächlich Maschinenbauer und Designer befanden. Anders als in anderen Bereichen wird im Projekt die Unterschiedlichkeit nicht als Handicap, sondern als Chance verstanden. Jedes

[1] Markus Weimer und Christoph Reichenbach
[2] Markus Weimer und Christian Rüdiger

Mitglied des Projektes kann sich der Unterstützung durch andere in deren jeweiligem Fachgebiet sicher sein.

So erstellte z.B. der Designer des Projektes Farbschema und Layoutvorlagen für die CeBIT-Demo (s.u.). Andererseits erstellte Markus Weimer für die Webseite des Projektes einige interaktive Elemente[3] und gab Hinweise zur technischen Realisierung der Webseite. Diese Zusammenarbeit der unterschiedlichen Fakultäten hat auch im Bereich der Mechatronik Lösungen ermöglicht, die ohne Sie nicht denkbar gewesen wären.

Messen Für die Industriepartner des Läufers ist es natürlich von besonderem Interesse, sich mit dem Läufer der interessierten Öffentlichkeit zu präsentieren. Dies ist vor dem Hintergrund zu sehen, daß einiger der Partner im Läufer mit Technologien arbeiten, die so noch nicht kommerziell erhältlich sind, oder nur einem sehr kleinen Kreis von Spezialisten bekannt sind. Auf Messen können sich die Industriepartner mit dem Läufer einen wahren Publikumsmagneten auf den Stand stellen, um an Ihm die eigene Technologie zeigen zu können.

Für das Projekt stellen Messen zum einen eine Herausforderung dar, da der Läufer zum Beginn einer solchen in einem präsentablen Zustand sein muß. Zum anderen bergen die Messen aber auch sehr große Chancen in sich, Kontakte zu weiteren interessanten Partnern aus Industrie, Forschung und Politik zu knüpfen.

Das Mechatronik-Team hatte zusammen mit dem Projekt Läufer die Gelegenheit, sich auf der CeBIT 2002 auf dem Stand von IBM zu präsentieren. Während dieser Messe wurden erste Kontakte zu SHARP (s.o.) und auch zur C't (s.u.) aufgebaut. SHARP hat mittlerweile auch einen PDA für die Entwicklung zur Verfügung gestellt.

Medien Ein Fahrzeug wie der Läufer erregt natürlich ein gewisses Interesse der Öffentlichkeit und damit ebenfalls das der Medien. Im Rahmen des Projektes sind so schon einige Veröffentlichungen der Projektarbeit zustande gekommen.

Das Mechatronik-Team hatte im Rahmen des Projektes ebenfalls die Gelegenheit, sich einem breiteren Publikum zu präsentieren. So z.B. in [And02]. Auch wurden Kontakte zum Computermagazin c't geknüpft.

Insgesamt hat das Projekt dieser Arbeit einen spannenden und interessanten Rahmen gestellt, auch wenn es durch seinen aktiven Charakter oftmals das Ziel dieser Arbeit neu definierte. Es bleibt zu hoffen, das die Universitätsausbildung in Deutschland zukünftig mehr Studenten die Chance auf ein solches Umfeld bieten kann.

[3] z.B. ein Kontakt-Formular

Kapitel 7

Danksagungen

Inhaltsangabe

Im Rahmen unserer Arbeit wurden wir durch einige Firmen und Privatpersonen unterstützt, denen wir hier gerne danken möchten:

SHARP Deutschland Für die Unterstützung unserer Entwicklung durch zwei Sharp Zaurus PDAs.

CE.TRON Gesellschaft für Hard- und Softwaretechnik Dort besonders Andreas May für ein Offenes Ohr bei der Entwicklung der Platine und der darauf aufsetzenden Software.

IBM Für die Möglichkeit, auf der CeBIT 2002 auszustellen sowie die Bereitstellung von Hardware.

Literaturverzeichnis

[AG99] AG, SIEMENS: *TLE 6330 Smart Octal Low-Side Switch*, 1999

[And02] ANDERL, Reichenbach W.: Embedded Devices in der Produkt... In: *Forschung Aktuell* (2002), S. 50ff

[Bür] BÜRKLIN, Elektronik: Online. – http://www.buerklin.com

[CAN] *CiA: CANopen*. Online. – http://www.can-cia.de/canopen

[Cor02] CORPORATION, Ramtron I.: *FM24C64 64kB FRAM serial Memory*, 2002

[Dev] *CiA: DeviceNet*. Online. – http://www.can-cia.de/devicenet/

[Far] FARNELL, Electronics: Online. – http://www.farnell.com

[Inc96] INC., Linear T.: *LTC 1446 Dual 12-Bit Rail-to-Rail Micropower DAC's*, 1996

[Inc98] INC., Parallax: *SX-Key/Blitz Development system Manual 1.1*, 1998

[Inc99] INC., Fairchild S.: *MM74HC138 3-To-8-Line Decoder*, 1999

[Inc00a] INC., Linear T.: *LTC 1594 4-channel Micropower Sampling 12-Bit serial I/O A/D-Converter*, 2000

[Inc00b] INC., Microchip T.: *MCP2510 Standalone CAN Controller with SPI Interface*, 2000

[Inc00c] INC., Ubicom: *SX Family User's Manual Rev 3.1*, 2000

[Ort02] ORTNER, Prof. Dr. E.: Sprachingenieurwesen - Empfehlung zur inhaltlichen Weiterentwicklung der (Wirtschafts-)Informatik. In: *Informatik-Spektrum* (2002), S. 39ff

[Pro] PROJECT, The F.: Online. – http://familliar.handhelds.org

[RC] RS-COMPONENTS: Online. – http://www.RS-Components.com

[Sch00] SCHLINGENSIEPEN, Jörn, Technische Universität Darmstadt, Studienarbeit, 2000

[SDS] *Smart Distributed System*. Online. – http://content.honeywell.com/sensing/prodinfo/sds/

[Sem] SEMICONDUCTORS, Dallas: Fundamentals of RS-232 serial communications. In: *Application Note 83*

[Tec] TECHNOLOGIES, Transvirtual: Online. – http://www.pocketlinux.org

[Tec99] TECHNOLOGIES, Infineon: *datasheet 5-A H-Bridge for DC-Motor Applications*, 1999

[Tro] TROLLTECH: Online. – http://www.trolltech.com

[Wal95] WALLASCHEK: Grundstruktur mechatronischer Systeme. In: *VDI-Berichte 1215* (1995)

Anhang A

Protokoll-Layer 1: LLO-Protokoll, Version 0

Das Läufer Layer 1- Protokoll hat die Aufgabe, eine zielgerichtete, sichere Übertragung von potentiell beliebigen Datenmengen in einem Broadcast-Netzwerk mit bis zu 30 Kommunikationsteilnehmern und potentiell begrenzter Paketgröße durchzuführen. Sicher in diesem Zusammenhang bedeutet, daß es überprüft, ob ein Rezipient die für ihn gedachte Botschaft erhalten hat, und, falls nicht, eine vernünftige Anzahl von Neuübertragungen versucht, nach denen es den Rezipienten als vom Netz abgetrennt registriert. Das vorliegende Netz ist dabei sternförmig aufgebaut.
Die hier befindliche Spezifikation bezieht sich auf Version 0 des Läufer Layer One (LLO)-Protokolls.

A.1 Kommunikation

Das Protokoll sieht ein Multi-Master-Netzwerk als Basis vor, mit einem Master in der Funktion des LLO-Controllers. Das Netzwerk muß entweder drei Klassen von Nachrichten-Prioritäten, oder voll-Duplex-Kommunikation unterstützen.
Im Kontrast zum LLO-Controller (kurz: Controller) werden die anderen Busteilnehmer als LLO-Clients (kurz: Clients) bezeichnet, auch, wenn diese Bezeichnung ihre Möglichkeiten nicht voll würdigt.

In dieser Spezifikation wird Bezug auf Timeouts genommen. Diese Timeouts werden nicht explizit spezifiziert, mit Ausnahme des Keep-Alive-Timeouts, um Implementierern die Wahl Bus-spezifische angemessene Timeouts zu erlauben.

A.1.1 Clients

Jedem Client ist eine eindeutige Identifikationsnummer (Port) zugeordnet, die im Addreßbereich des Protokolls liegt (0-31 in dieser Version) und nicht 0 oder 1 ist (1 ist für den Controller reserviert).
Eine Menge von Hardware ist ein Client genau dann, wenn sie sich an einem Port P != 0 als Client im Sinne der Protokollspezifikation verhält. Aus dieser Definition folgt, daß mehr als ein physikalischer Empfänger als Client agieren kann; die Protokollimplementierungen auf den Clients haben in diesem Fall Vorkehrungen dafür zu treffen, daß ihr Verhalten auch im Fehlerfall dieser Spezifikation entspricht.

Jedem physikalichen Element eines Clients ist ein sicherer Betriebszustand zugeordnet. Sobald dieses Element feststellt, daß es von den Übertragungen des Netzes abgetrennt wurde, muß es sämtliche Kommunikation beenden und den sicheren Betriebszustand einnehmen.
Ein Verlassen des sicheren Betriebszustandes ist nur durch Zurücksetzen des Gerätes auf einer untergeordneten Ebene möglich (z.B. Hardware-Reset).

A.1.2 Controller zu Client-Übertragungen

Synchrone Übertragungen

Eine Übertragung von einem Controller zu einem Client erfolgt unangekündigt mit einer Sequenz von STX-Paketen (Synchronous Transfer); aus Sicht des Masters:

```
← STX(id, more) size: data
→ ACK(id, more)
```

Graphisch dargestellt:

010IIIII	X00MSSSS	LLLLLLLL	··· SIZE bytes ···

011IIIII	X00MSSSS

Wobei die Bedeutungen wie folgt sind:

Name	Kürzel	Bedeutung
STX	010	Synchronous Transfer: Operationscode
ACK	011	Acknowledgement: Operationscode
more	M	Flag, welches indiziert, ob noch mehr Daten in diesem Kommunikationsstrom folgen
size	L	Menge an folgenden Datenbytes
id	I	Identität des Clients (Client-Kanals)
seqnr	S	Sequenznummer (s. A.1.4)
reserviert	X	Reserviertes Bit
data		Sequenz von *size* Bytes an Daten

Wenn bei einer Übertragung das *more*-Flag gesetzt ist, müssen die Daten der nächstes Übertragung ans Ende der vorherigen Übertragung angehängt werden. Zwischen Controller und Client findet maximal eine Übertragung gleichzeitig statt; somit ist gewährleistet, daß Pakete einer Übertragung eindeutig zugeordnet werden können.

Ein Ausbleiben des Acknowledgements für eine bestimmte, implementierungs- abhängige Zeit wird als Timeout gewertet; der Sender führt in diesem Fall eine Neuübertragung durch.
Nach einer angemessenen Anzahl von versuchten Neuübertragungen wird schliesslich eine Fehlermeldung an die aussendende Stelle ausgegeben oder (falls keine solche Stelle existiert) intern behandelt werden.

Broadcast-Übertragung

Broadcast-Übertratungen entsprechen Übertragungen, bei denen die Zieladdresse gleich 0 ist.

A.1.3 Client zu Controller- Übetragungen

Diese Übertragungen sind synchron und werden mit der `RTX`-Operation durchgeführt:

```
→  RTX(id, more) size:data
←  ACK(id, more)
```

Die Übetragungen finden statt, sobald der Controller explizit mit einer NOP-Anweisung den Bus freigibt oder pausiert. In diesem Fall werden R-Übertragungen von dem ausgewiesenen Sender akzeptiert, bis eine Übertragung nicht mehr das 'more'-Flag gesetzt hat oder ein unbehebbarer Fehler auftrat. Jede erfolgreiche Übertragung wird mit einem `ACK` bestätigt.

A.1.4 Fehlererkennung und -behandlung

Das LLO-Protokoll geht von einem selbst fehlererkennenden Protokoll aus, daher testet es nur auf Fehler, die eine Netz-Trennung zwischen Controller und Client indizieren.

Sequenznummern

Jede synchrone und asynchrone Übertragung erhöht einen Sequenzzähler sowohl auf der Seite des Clients wie auch des Controllers; dieser Zähler hat den initialen Zustand 0 und zählt aufwärts, in Schritten der Grösse 1, Modulo 16. Sein Zustand wird in jeder Übertragung mitgesandt und danach erhöht, mit der expliziten Ausnahme von `SYS`-, `KAL`- und `NOP`-Übertragung (bei denen er nur mitgesandt, aber nicht erhöht wird) sowie Broadcasts[1].
Für `STX` und `RTX` werden hierbei zwei getrennte Zähler verwendet, um bei gepipelinetem Versandt eine eindeutige Zuordnungsfähigkeit der Nummern zu gewährleisten.

Übertragungen im Fehlerfall

Falls bei einer Übertragung ein Fehlerfall diagnostiziert wird, wird diese Übertragung entweder entweder erneut durchgeführt (wenn der Fehler vom Sender diagnostiziert wurde), oder es wird– anstelle eines `ACK` (synchron) oder statt zu schweigen (asynchron)– ein `SYS` zurückgesandt:

```
→  RTX(id, more) seq=10, size:data
←  ACK(id, more) seq=11
→  RTX(id, more) seq=14, size:data /* FEHLER */
←  SYS(id, more) seq=12, INVSQ /* Fehlersignalisierung */
```

In diesem Fall antwortet der Sender entweder ebenfalls mit `SYS` (Abbruch der Übertragung), oder er wiederholt die Übertragung an der vom erhaltenen `SYS` indizierten Stelle. Ein Abbruch erfolgt genau dann, wenn sich die angezeigte Sequenznummer des Empfängers auf eine vorhergehende Botschaft bezieht, oder wenn die Differenz der Sequenznummern modulo 16 grösser oder gleich 8 ist. Der Empfänger synchronisiert im Abbruchsfall seine Sequenznummer mit der des Senders. Dieser Fehlerfall kann unter folgenden Bedingungen auftreten (unter Voraussetzung der Korrektheit des Controllers):

a. Ein Client verwendet eine inkorrekte Protokollimplementierung

b. Ein Client hat mehrere asynchrone Pakete verpaßt

[1] Da die Sequenznummern von Broadcasts nicht definierbar sind. Der Nichterfolg eines Broadcasts kann somit nur durch Timeout der Bestätigung erkannt werden. Dies bedeutet, dass die Empfangsreihenfolge für Broadcasts nicht garantiert ist und reduziert deren Nutzen auf globale Fehlermeldungen.

c. Ein Client besteht aus mehreren physischen Einheiten, von denen eine echte nichtleere Teilmenge einen Teil der Übertragungen verpaßt hat.

Für den Fall (a) kann die Korrektheit des Empfangs im Allgemeinen nicht gewährleistet werden. Fall (b) ist durch Definition der inhärenten Unsicherheit asynchroner Pakete abgedeckt. Kritisch ist somit Fall (c): Die empfangenden physikalischen Einheiten müssen erkennen, daß eine Geschwistereinheit einen Fehler erkannt hat, und sich ruhig verhalten. Im Falle eines Übertragungsabbruches müssen auch die Einheiten, die die Nachricht korrekt erhalten haben, die aktuelle Übertragung abbrechen, um Konsistenz zu gewährleisten.

SYN-Request

Ein `SYN`-Request ist eine synchrone `STX`-Übertragung, die eine size von 0 hat. Sie dient der Synchronisierung der Sequenznummern und ist wie eine normale Übertragung zu behandeln, soweit das Protokoll betroffen ist.

Nach spätestens 14 asynchronen Paketen (Broadcasts) in Folge muß ein Sender ein `SYN`-Paket schicken, um die Synchronität der Sequenzzähler zu gewährleisten.

Systemcodes

Bei einem SYS werden Systemcodes (meist Fehlercodes) mitversandt. Sie sind definiert wie folgt:

NR	ID	Bedeutung	Kommunikation
1	`EINTN`	Interner Fehler	Abbruch
2	`EINSQ`	Unpassende Sequenznummer	Retry oder `ABORT`
3	`ABORT`	Übertragung Ungültig, Abbruch	Abbruch
4	`CONTN`	Übertragung ab hier fortgesetzt	Fortsetzung
5	`EOVFL`	Nachricht zu lang: Buffer-Überlauf	Abbruch
6	`EINVR`	Ungültige Version	Abbruch
7	`EGLBL`	Globales Versagen	Sicherer Betriebszustand
8	`ENRDY`	Gerät nicht bereit	Retry nach Verzögerung

Ein SYS:EGLBL kann als Broadcast vom Controller eingesetzt werden, um alle Geräte in den sicheren Betriebszustand zurückzufahren (und damit den Bus effektiv zu deaktivieren).

Timeout

Bei synchronen Versendungen wird ggf. in einem spezifizierten Zeitraum kein `ACK` vom Empfänger erhalten. In diesem Fall wird das Paket erneut versandt

A.2 Keep-Alive

Mindestens einmal pro 100 ms muß der Controller für ein Keep-Alive-Signal (`KAL`) auf dem Bus sorgen. Bleibt dieses Signal für mehr als 300 ms aus, so müssen sich alle Clients in einen sicheren Betriebszustand zurückfahren.

A.3 Kodierung der Übertragungspakete

Dieses Kapitel dokumentiert die Standardkodierung der Pakete; implementierende Systeme können eine eigene, angemessenere Kodierung verwenden.

Graphisch präsentiert sich die Kodierung wie folgt:

OOOIIIII	XVVMSSSS	LLLLLLLL	··· SIZE bytes ···

wobei manche Befehle nur zwei Bytes verwenden und die Bedeutung der Abkürzungen wie folgt ist:

Kürzel	Bedeutung
O	Synchronous Transfer: Operationscode
I	Identität des Clients (Client-Kanals)
X	Reserviertes Bit
V	Versionsnummer
M	Flag, welches indiziert, ob noch mehr Daten in diesem Kommunikationsstrom folgen
S	Sequenznummer (s. A.1.4)
L	Menge an folgenden Datenbytes

A.3.1 Erstes Präfixbyte

Das erste Byte des Paketes (Präfixbyte) setzt sich zusammen wie folgt:

Bits	Bedeutung
7-5	Operation
4-0	Rezipient

A.3.2 Operationen

Nummer	ID
0	NOP
1	STX
2	RTX
3	ACK
4	SYS
5	KAL

A.3.3 Zweites Präfixbyte

Bits	Bedeutung
7	Reserviert
6,5	Version (hier immer 0)
4	Das “mehr”-Flag
3-0	Sequenznummer

A.3.4 STX, RTX: Übetragungspakete

In diesen Paketen folgt ein Byte mit der Anzahl der zu übertragenden Datenbytes, gefolgt von entsprechend vielen Daten.

A.3.5 ACK, NOP und KAL

Diese Pakete sind nach den Präfixbytes leer.

A.3.6 SYS

Auf dieses Paket folgt genau ein Byte mit dem Identifikator der System- Message.

A.4 Appendix A: Implementierung auf dem CAN-Bus

Bei einer Implementierung auf dem CAN-Bus (LLO/CAN) wird das komplette erste Präfixbyte im Identifyer dargestellt:

Bits	Bedeutung
10-8	LLO/CAN-Operation
7-5	Priorität für RTX-Botschaften, sonst 011
4-0	ID des Rezipienten

A.4.1 A.1 Codierung der LLO/CAN-Operationen

Bit 10	Bit 9	Bit 8	ID
0	0	0	SYS
0	0	1	ACK
0	1	0	KAL
1	0	0	STX
1	1	0	RTX
1	1	1	NOP

Somit gilt: $\{\texttt{SYS}, \texttt{ACK}, \texttt{KAL}\} > \{\texttt{STX}\} > \{\texttt{RTX}\} > \{\texttt{NOP}\}$, wobei A > B bedeutet, daß alle Elemente von A eine höhere Priorität als ein beliebiges Element aus B haben.

Bei STX und RTX wird die Längenangabe im dafür von CAN vorgesehenen Bereich gespeichert, nicht als zusätzliches Byte. Das Protokoll verwendet im optionalen Datenbereich also genau ein Byte.

Anhang B

Protokoll-Layer 0: LLZ-Protokoll, Version 0

Alle folgenden Aussagen beziehen sich auf Version 0 des Protokolls, mit Ausnahme explizit gekennzeichneter Stellen.

Das LLZ (Läufer Layer Zero)- Protokoll realisiert eine asynchrone und synchrone Kommunikation über einen existierenden sicheren Transferkanal. 'Sicher' hat hierbei die Bedeutung, daß die erfolgreiche Übertragung mit hoher Sicherheit "garantiert" wird; kommt die Übertragung nicht zustande, tritt der Garantiefall ein und es muß ein Fehler auf der darunterliegenden Protokollebene ausgelöst werden. Das Protokoll unterscheidet zwischen Empfänger- und Senderstationen, die verschiedene Rechte und Pflichten haben.
Zudem ist das Protokoll Zustandsbehaftet.

Im folgenden bezeichnet $\prec X$ das Ereignis, daß die Botschaft X empfangen wurde, und $X \succ$ das vollständige und erfolgreiche Versenden der Botschaft X, ebenso wie $\rightarrow X$ (Senden) und $\leftarrow X$ (Empfangen).

B.1 Kommunikation

Jegliche Kommunikation, mit Ausnahme einer Notifikation, wird von einem Sender ausgelöst.

B.1.1 Synchrone Kommunikation

Operationen und Semantik

Für jedes Gerät ist eine Menge von Operationen definiert. Diese Operationen dürfen durch eine bestimmte Anzahl von Bytes parameterisiert sein. Für jede mit $\overline{x}$ parametrisierte Operation $p(\overline{x})$ gilt, daß die Semantik des Empfangs von $p\overline{x}'$ nach $p\overline{x}$ gleich der Semantik des Empfangs von $p\overline{x}'$ ist, also, daß alle Operationen nur Zustand *setzen* und *keine relativen Modifikationen* an ihm durchführen.

Request

Eine synchrone Kommunikation beginnt mit einem Request, der vom Sender ausgelöst wird. Die Notation hierfür ist (aus Sender-Sicht)

→ `REQ(n) <OP> [Parameter]*`

z.B.

→ `REQ(2) SET-STATE 7`

(n ist 1 plus die Anzahl der Parameter). Die Operation ist ein 7-Bit-Wert (das MSB ist reserviert); die Anzahl der Parameter ist beschränkt auf 15. Auf einen Request muß ein Response folgen. Protokollimplementierungen müssen also eventuell vorher eingehende Notifikationen zurückstellen.

Response

Aus Sender-Sicht ist die Notation hierfür wie folgt:

← `RES(n) <OP> [Value]*`

Wobei `<OP>` eine Wiederholung des Operationscodes aus dem Request ist.

Kommando

Statt einem Request kann auch ein Kommando (`CMD` statt `REQ`) abgesetzt werden. Der einzige Unterschied ist, daß keine Response hierauf generiert wird.

B.1.2 Asynchrone Notifikation

Der Empfänger kann asynchron Notifikationen absetzen. Diese Notifikationen sind wie Responses aufgebaut, haben als Transmissionsoperation (s. 1.3) jedoch `NTF` statt `RES`. Insbesondere beinhalten sie auch eine Operation. Notifikationen dürfen nur bei außergewönlichen Situationen verwendet werden. Es bleibt Senderimplementierungen vorbehalten, Empfänger, die Notifikationen unangemessen mißbrauchen, per `SDN` zu deaktivieren.

Jeder Einheit ist jedoch eine Notifikation pro Sekunde zuzugestehen.

B.1.3 Spezifikation der Transmissionspakete

Alle Übertragungen sind Byte-basiert. Ein Transmissionspaket mit einem Payload der Länge n+1 sieht aus wie folgt:

| prfx | dat0 | dat1 | . . . | datn |

I.e. ein 8-Bit-Präfix, gefolgt von n+1 Datenbytes.

Der Präfix hat folgende Form:

Bit	ID	Bedeutung
7-5	TOP	Transmissionsoperation (siehe B.1.4)
4	–	reserviert (muß 0 sein)
3-0	LEN/ ERC/ VNR	LEN: Anzahl der folgenden Bytes an Daten minus 1. Falls TOP den Wert ERR hat, ist dies der Error- code; es folgen keine weiteren Daten. Falls TOP den Wert INI oder IOK hat, ist hier die Versionsnummer des Protokolls codiert.

B.1.4 Transmissionsoperationen

Code	ID	Semantik
0	SDN	Der Empfänger soll einen sicheren Betriebs- Zustand einnehmen und sämtliche Kommunikation unterbinden (siehe Sektion B.3)
1	IOK	Initialisierungsantwort (Protokollversion in VNR)
2	REQ	Sender-Operation: Synchrone Anfrage
3	RES	Empfänger-Operation: Synchrone Antwort
4	CMD	Sender-Operation: Asynchrones Kommando
5	NTF	Empfänger-Operation: Asynchrone Notifikation
6	INI	Initialisierungsoperation, siehe Sektion B.3
7	ERR	Protokollfehler, siehe B.1.4

Protokollfehler

Falls ein Sender eine ungültige Übertragung absetzt, muß der Empfänger ihn darüber benachrichtigen. Zu diesem Zweck ist TOP ERR definiert.

Die folgenden Werte für ERC sind definiert:

ERC	**Bedeutung**
0	Sonstiger Fehler (catch-all-code)
1	Version nicht unterstützt
2	Falsche Rolle: Sender versuchte, `RES`, `NTF` oder `IOK` zu senden
3	Schon initialisiert: `INI` gesendet, obwohl der Empfänger bereits initialisiert ist
4	Fehler verschickt: Der Sender sandte ein `ERR` zum Empfänger
5	Paket korrupt: Ein reserviertes Bit ist falsch gesetzt
6-15	– *reserviert* –

B.2 Payload

LLZ v0 überträgt Datenpakete einer Größe von 1 bis 16 Bytes. Das erste Byte, als "Operation" bezeichnet, muß immer vorhanden sein.

B.3 Lebenslauf von Kommunikationsteilnehmern

Der Lebenslauf von Sendern wird in dieser Spezifikation nicht behandelt. Empfänger jedoch werden von Sendern initialisiert und, bei Bedarf, deaktiviert; die entsprechenden Zustände und Übergänge finden sich in B.3.1.

B.3.1 Empfänger

Das LLZ-Protokoll definiert folgende Zustände eines Empfängers:

Code	**Name**	**Erklärung**
S0	`PRE-INIT`	Initialer Zustand. Gerät darf keine Fehlercodes senden.
S1	`INIT`	Gerät hat INI erhalten, noch nicht beantwortet
S2	`RUNNING`	Gerät hat IOK geantwortet und ist im normalen Betriebszustand
S3	`DOWN`	Gerät hat SDN erhalten und den sicheren Betriebszustand eingenommen, es darf keine Fehlercodes mehr senden.

In Zuständen **S0**, **S1** und **S3** werden alle Messages ignoriert, die nicht zu einem Zustandsübergang führen.

B.3.2 Shutdown

Beim Erreichen des `DOWN`-Zustandes sollte das Kommunikationsprotokoll, soweit möglich, das zugrundeliegende System informieren, so daß dies einen gesicherten "Failsafe"-Betriebszustand einnehmen kann.

B.3.3 Initialisierung des Empfängers

Die korrekte Sequenz zur Initialisierung eines Empfängers verläuft aus Sicht des SENDERS wie folgt:

← `INI(VNR=0)`
→ `IOK(VNR=0) ops[16]`

Folgt keine Antwort, so muß der Sender annehmen, daß der Empfänger nicht existiert.

Statt IOK kann auch `ERR(1)` (Version nicht Unterstützt) gesendet werden, siehe B.3.3.

Das Datenfeld ops[16] beinhaltet eine Liste aller vom Gerät unterstützten Operationen, codiert in 128 Bits. Der Test auf Vorhandensein der Operation x (für x > 7) ist also in Hochsprachensemantik wie folgt:

```
have-op(x) := ((x >> 3) < n) /* Byte is defined */
          && (ops[x >> 3] & (1 << (x & 7)))
```

Dies erlaubt die Deklaration der Operationen 0-127. Die Deklaration dient Debugzwecken und ist in dieser Protokollversion nicht verbindlich.

Versionsunterschiede

Falls die Protokollversion des Senders vom Empfänger nicht unterstützt wird, muß er mit `ERR(1)` antworten. Der Sender kann dann andere Protokollversionen durchprobieren, oder mit `SDN` das System deaktivieren.

Anhang C

Serielle Protokollschicht: LSP-Protokoll, Version 0

Alle folgenden Aussagen beziehen sich auf Version 0 des Protokolls, mit Ausnahme explizit gekennzeichneter Stellen.

Das LSP-Protokoll transferiert "namenlose" Datenblöcke unspezifizierter Länge über eine serielle Verbindung. Es spezifiziert "Blocks", die die atomaren Einheiten der Übertragung darstellen (und deren Semantik darüberliegenden Protokollschichten überläßt) und stellt sicher, daß für eine fehlerfreie Leitung die eintreffenden Blöcke eindeutig erkannt werden können.

Zur Übertragung wird auf 8-Bit-Zeichen zurückgegriffen.

C.1 Zeichenvorrat

Code	Name	Bedeutung
`0x00--0xef`		Freie Zeichen
`0xf0`	`SOB`	Blockstart
`0xf1`	`EOB`	Blockende
`0xf2`	`ESC`	Escape
`0xf3--0xf7`	–	*reserviert*
`0xf8--0xff`		Freie Zeichen

Der Zeichenvorrat teilt sich in die Menge der Freien Zeichen, und in die Menge der Kontrollzeichen {`0xf0 ...0xf7`}.

C.2 Block

Ein *Block* ist eine Sequenz von Zeichen aus dem Zeichenvorrat, deren erstes Zeichen ein `SOB` und deren letztes Zeichen ein `EOB` ist, und für die gilt, daß sie an-

sonsten keine weiteren Vorkommnisse von Kontrollzeichen mit Ausnahme von `ESC` wie in C.3 angegeben.

Blocks werden in der Reihenfolge, in der sie definiert sind, sequentiell uebertragen und empfangen.

C.3 Nachrichtenkodierung

Um eine beliebige Folge auf dem kompletten Zeichenvorrat in einen Block zu kodieren, wird vorgegangen wie folgt:

C.3.1 Blockbeginn

Zu Beginn der Folge wird ein `SOB` angegeben.

C.3.2 Elemente des Zeichenvorrates

Für alle Zeichen aus der Sequenz wird in der gegebenen Reihenfolge folgendes in den Block eingetragen:

- `b`, falls das zu übertragende Zeichen `b` und ein Element der Freien Zeichen ist.
- `ESC` $\mathtt{b}'$, falls das übertragende Zeichen `b` kein Element der Freien Zeichen ist und $\mathtt{b}' = b \mod \mathtt{0x10}$.

C.3.3 Blockende

Nachdem alle Elemente der Folge eingetragen wurden, wird `EOB` eingetragen.

C.4 Nachrichtendekodierung

Um einen Block in die urspruengliche Folge zu dekodieren, wird invers zu C.3 vorgegangen. Bei allen Zeichen, die einem `ESC` folgen, werden dabei die höchstwertigen vier Bit gesetzt, um das ursprüngliche Zeichen wiederherzustellen.