Interner Bericht
DESY F35D-88-02
Februar 1988

ENTWICKLUNG EINER VME-PADAC SCHNITTSTELLE ZUM TRANSPORT VON STRAHL-SIMULATIONSDATEN DES HERA SPEICHERRINGES

von

T. Woeniger

II. Institut f. Experimentalphysik, Universität Hamburg

Entwicklung einer VME-PADAC Schnittstelle zum Transport von Strahl-Simulationsdaten des HERA Speicherringes

Torsten Woeniger
Diplomarbeit
II. Institut für Experimentalphysik
der Universität Hamburg

Februar 1988

A B S T R A C T

Development of a VME-PADAC Interface for the Transfer of Simulation Data from the HERA Accelerator

A multiprocessor system named BOP (Beam Orbit Processor) for beam tracking purposes is described in this work. There is also a description of the data-link from this computer system to the DESY-IBM computer center with a VME-Bus side is written in the computer language C. On the IBM side the software for the communication of VAX-computers with the IBM computer center is used.

In the conclusion is a brief description of the results from the computations done with the computer code RACETRACK on the BOP-system.

Gewidmet
meinen Eltern
die mir die Ausbildung
bis jetzt ermöglicht haben.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Wechselwirkungen zwischen Protonen und Elektronen haben sich in den letzten Jahrzehnten als eine der wichtigsten Beobachtungsgrundlagen zur Untersuchung der Struktur der Materie erwiesen. Das Deutsche Elektronen Synchrotron (DESY) in Hamburg baut deshalb einen großen Elektron-Proton Speicherring mit dem Namen HERA (Hadron-Elektron Ring Anlage). Mit diesem neuen Beschleuniger sollen Wechselwirkungen zwischen Protonen mit einer Energie von 820 GeV und Elektronen einer Energie von 30 GeV untersucht werden. Die wichtigsten technischen Daten dieses Elektron-Proton Speicherringes sind der Tabelle 1 zu entnehmen.
Die Teilchen in diesem Speicherring sollen durch Magnete auf ihrer Bahn gehalten werden, wobei ein großer Speicherring wie HERA aus etwa 2000 solcher Magneten besteht, welche zur Ablenkung und Fokussierung der Teilchen dienen. Die Magneten sind so konstruiert, daß die Teilchen innerhalb eines gewissen Querschnittes senkrecht zur Strahlrohrachse stabile Bahnen durchlaufen. Dieser Bereich wird auch als die Apertur des Speicherringes bezeichnet. Weil die Teilchen über einen Zeitraum von mehreren Stunden in dem Beschleuniger gespeichert werden sollen, ist es sehr wichtig, daß man über die Apertur der Maschine genau Bescheid weiß. Die Berechnung der Apertur eines Speicherringes wie HERA ist ein zu komplexes Problem, als daß es explizit gelöst werden könnte. Deshalb wurden für diesen Zweck Simulationsprogramme geschrieben, mit welchen man den Strahlverlauf durch die Magneten für eine große Zahl von Umläufen numerisch berechnen kann. Eines dieser Programme ist RACETRACK, welches von A.Wrulich [8] entwickelt worden ist.
Die Simulationen, die mit diesen Programmen durchgeführt werden, bestehen u.a. aus Polynomberechnungen, welche mit einer Genauigkeit von 64-Bit erfolgen müssen, um Rundungsfehler zu vermeiden. Ein weiteres Kennzeichen dieser Programme ist, daß der eigentliche Teil, in welchem die Bahnen der Teilchen berechnet werden, relativ kurz ist. Er besteht in der hier verwendeten Version nur aus ein paar hundert Programmzeilen. Aber gerade dieser Programmteil benötigt mit Abstand am meisten Rechenzeit, während die im Zeilenumfang großen Vorbereitungsrechnungen in der Gesamtrechenzeit praktisch nicht ins Gewicht fallen. Ein weiteres Merkmal dieser Programme ist, daß sie ein verhältnismäßig kleines Ein- und Ausgabevolumen besitzen.
Die spezifischen Eigenarten dieser Programme, sowie der große Verbrauch an Rechenzeit auf der damaligen DESY-Rechenanlage (eine IBM 3081) von mehreren Stunden täglich, führten zu der Idee, für diese Programme einen speziellen Rechner zu bauen.
Dieser Spezialrechner wird im folgenden *BOP* (*B*eam *O*rbit *P*rozessor) genannt. Für diesen Rechner wollte man viele Mikroprozessoren zusammen mit selbstgebauten Fließkommarecheneinheiten (*FPU* = *F*loating *P*rocess *U*nit) verwenden. Auf jedem einzelnen Rechnermodul, welches aus einem Mikroprozessorrechner mit Speicher und der *FPU* besteht, sollte dann jeweils ein Simulationsprogramm laufen. Der Vorteil dieses Rechneraufbaus besteht darin, daß man durch den parallelen Einatz von mehrerer dieser Modulen die Rechenkapazität des gesamten Systems vervielfachen kann. Die Steuerung der einzelnen Einheiten sollte ferner durch einen weiteren Rechner geschehen, welcher wiederum mit einem weiteren Rechner, der zur Programmentwicklung dient, verbunden ist.
Alle diese Aspekte wurden beim Bau des *BOP* -Systems auch verwirklicht. Als ich meine Arbeit aufnahm existierte der Rechner bereits fast in der Form, wie es in der Abbildung 2 auf der Seite 16 dargestellt ist. Allerdings waren zu diesem Zeitpunkt erst drei der zehn Modul-

Parameter :	Protonenring	Elektronenring	Einheit
Nominelle Strahlenergie	820	26	GeV
Einschußenergie	40	14	GeV
Anzahl der Wechselwirkungspunkte	4	4	
Luminosität	$1.5 \cdot 10^{31}$	$1.5 \cdot 10^{31}$	$cm^2 s^{-1}$
Umfang	6335	6335	m
Umlauffrequenz	47.31	47.31	kHz
max. Anzahl der Teilchenpakete	220	220	
Teilchenstrom	163	58	mA
Teilchenanzahl pro Paket	$1 \cdot 10^{11}$	$3.5 \cdot 10^{10}$	
horizontale Emittanz (σ_x)	$0.86 \cdot 10^{-8}$	$3.5 \cdot 10^{-8}$	$\pi\ rad$
vertikale Emittanz (σ_z)	$0.43 \cdot 10^{-8}$	$0.69 \cdot 10^{-8}$	$\pi\ rad$
Länge des Teilchenpakets	110	7.8	mm
Energieverlust pro Umlauf	$1.4 \cdot 10^{-10}$	71.9	MeV
Hochfrequenz	208.194	499.6655	MHz
Füllzeit	20	15	min
Stärke der Magneten	4.649	0.1426	T
Vakuum	$< 10^{-8}$	$< 10^{-8}$	mbar

Tabelle 1: Parameter von HERA

rechner, Targets genannt, in das System integriert worden. Auch der Datenweg zur zentralen IBM-Rechenanlage über das PADAC-Netz existierte zu diesem Zeitpunkt noch nicht. Dieses waren beides in dieser Arbeit zu lösende Aufgaben. Der Datenweg über das PADAC-Netz war notwendig, weil der Datentransfer über das DESY-NET für große Datenmengen viel zu lange gedauert hätte. Ansonsten habe ich das gesamte System während meiner Arbeit hardwaremäßig gewartet.
Auf der Softwareseite wurden im Rahmen dieser Arbeit die Programme des *BOP* -Systems geordnet und in das MAKE-Konzept von *UNIX* eingebunden. Weil bei der Datenübertragung zum Programmentwicklungsrechner (PCS-Workstation) über das Q-VME-Bus Interface Probleme auftraten, habe ich diese mit einer von mir geschriebene Software beseitigt. Ein wesentlicher Teil der Arbeit bestand in der Entwicklung von Programmen für den Datentransfer zwischen dem *VME* -Bus des *BOP* -Rechners und der IBM. Um die Funktion des gesamten Systems zu demonstrieren, habe ich dann noch ein Simulationsprogramm auf dem *BOP* installiert, welches auf dem Programm RACETRACK [8] beruht.
Diese Berechnungen waren damit das eigentliche Ziel meiner Arbeit, wobei ich zur graphischen Aufbereitung der Daten die Grafikmöglichkeiten der IBM benutzt habe.
Zum Verständnis des gesamten Systems ist dieser Arbeit ein Kapitel vorangestellt, welches die Hard- und Softwareumgebung beschreibt. Die Beschreibung des *BOP* -Rechnersystems erfolgt dann im Abschnitt 3 und im anschliessenden Abschnitt 4 erläutere ich die Verbindung für den Datentransport zur IBM-Rechenanlage. Die Ergebnisse der Simulationsrechnungen finden sich schließlich im Abschnitt 5.

2 Die Hard- und Softwareumgebung des Strahlrechners

2.1 *UNIX* und *C*

2.1.1 Einleitung

Am Anfang der Entwicklung des von mir verwendeten BOP-Systems gab es mehrere Gründe, welche für das von der Firma PCS angebotene CADMUS-System zur Softwareentwicklung sprachen. Es war damals das einzige käufliche Rechnersystem, welches nicht nur denselben Mikroprozessor wie der *BOP* (MC 68000), sondern auch das Betriebsystem *UNIX* und die Sprache *C* verwendete. Dadurch, daß das Entwicklungssystem denselben Mikroprozessor wie die VME-Bus Rechner des *BOP* -Systems verwendete, wurden etwaige Probleme mit Crossassembler - Software vermieden. Schließlich wurde durch die Wahl des flexiblen Betriebsystems *UNIX* , in der Verbindung mit der Sprache *C* , ein sehr mächtiges Software-Entwicklungssystem eingesetzt, welches im folgenden näher erläutert wird.

2.1.2 Die Programmiersprache *C*

Die Computersprache *C* wurde bereits im Jahre 1972 von Dennis Ritchie in den Bell-Laboratorien (Murray Hill, USA) entwickelt und in den beiden Jahren 1973/1974 von Brian W. Kernigham weiter verbessert.
Bedeutsam für die Entwicklung dieser Sprache war es, daß in der folgenden Zeit das anfangs vollständig in Assembler geschriebene Betriebsystem *UNIX* verbessert und zu 90 Prozent in *C* geschrieben wurde. Weil die erste *C* -Implementation auf einem *UNIX* -Betriebsystem stattfand und insbesondere systemspezifische Aufgaben, wie die Ein- und Ausgabe, in *C* durch Bibliotheks-Routinen übernommen wurden, hat sich ein defacto-Standard herauskristallisiert, welcher Teile der *UNIX* - Bibliothek umfaßt. Da die Bibliothek-Funktionsaufrufe standardisiert sind, ist eine hohe Portabilität erreicht worden. Dieser Standard wird in dem Buch von Kernigham und Ritchie [1] beschrieben.
Hieraus ergibt sich allerdings ein kleiner Nachteil, welcher aber auch bei einigen anderen Sprachen auftritt. Selbst wenn nur wenige Funktionen einer Bibliothek benutzt werden, wird gleich die gesamte Bibliothek an das ausführbare Programm angebunden. Deshalb sind die ausführbaren Programme gerade bei kleinen Quellprogrammen, welche Bibliotheksfunktionen benutzen, überproportional groß. Dieses ist aber angesichts der heutzutage vorhandenen großen Speicher kein schwerwiegendes Problem mehr.
Hinzu kommt, daß die übersetzten *C* -Programme im Vergleich zu anderen Hochsprachen sehr schnell sind. Der Grund hierfür ist, daß der Aufbau von *C* (z.B. die Art der Schleifen) dem der Assemblersprache eines Prozessors sehr stark ähnelt. Zusätzlich existieren einige sehr assemblernahe Befehle (z.B. das Increment) und Optionen, welche dem Compiler bei der Laufzeitoptimierung des Codes helfen (z.B. der Befehl *register*). Der Befehl *register* weist den Compiler an, entsprechend deklarierte Variabeln möglichst in den Registern des Mikroprozessors zu halten.
Eine weitere Eigenart dieser Programmiersprache ist, daß dem Programmierer sehr viele Freiheiten gelassen werden. So können z.B. Variabeln von unterschiedlichen Typen untereinander zugewiesen werden. Diese freizügige Handhabung der Sprachelemente hat sowohl Vor- als auch Nachteile.
Einerseits steht dem Programmierer eine Anzahl von Tricks zur Verfügung, mit denen sich

viele Probleme, welche sonst komplizierte Routinen erfordern würden, quasi nebenbei erledigen lassen. Auf der anderen Seite soll der Umgang mit einer solchen Freiheit sehr wohl geübt sein. Bei einer Programmierung ohne große Disziplin entstehen *C* - Programme , welche sehr schwer zu lesen sind und zu äußerst komplizierten Fehlern neigen. Besonders kritisch sind hierbei Bereichsüberschreitungen von Feldern und Überschreitungen bei Zuweisungen ungleicher Typen. Es existieren zwar Programme, welche eingehendere Syntax-Checks machen als der *C* -Compiler, aber ob auf die von diesen Programmen kommenden Meldungen und Warnungen eingegangen wird, liegt nur am Programmierer.
Wenn der Programmierer aber eine ausreichende Erfahrung und Disziplin besitzt, so steht ihm mit *C* eine extrem leistungsfähige Programmiersprache zur Verfügung. In diesem Fall erlauben die hohe Modularität und die flexible Syntax einen kompakten und einprägsamen Quellcode. Es stehen standardmäßig viele Variabeltypen zur Verfügung, aus welchen sich durch Kombinationen praktisch beliebige Variabelstrukturen bilden lassen. Dadurch kann man die Variabelstrukturen speziellen Problemen anpassen. Der Geltungsbereich dieser Variabeln kann sowohl global als auch lokal sein und sie können bei einem Ausstieg aus einer Funktion ihren Wert behalten oder verlieren. Bei der Variablenübergabe kann entweder der Wert oder der Zeiger einer Variabeln übergeben werden. Damit stehen beide bei höheren Programmiersprachen vorkommende Übergabeverfahren zur Verfügung. Zur weiteren Verbesserung der Übersichtlichkeit trägt auch bei, daß viele Schleifentypen und lange Variablennamen unterstützt werden.
Ein weiteres Merkmal von *C* -Systemen ist die Existenz eines leistungsfähigen Makro-Preprozessors. Mit diesem kann man in einem Quell-Code-File beliebige Zeichenketten durch andere Zeichenketten auszutauschen. Damit besteht die Möglichkeit, anstatt von Konstanten einprägsame Texte zu verwenden, welche in einem Deklarationsteil diesen Konstanten zugewiesen werden. Solche Tabellen brauchen nicht direkt in der Quell-Datei stehen, sondern sie können auch mittels des *#include*-Kommandos in diese Datei gebracht werden. Dasselbe gilt ebenfalls für Funktionen, welche aber auch als Objekt-Files an das Hauptfile gebunden werden können.
Bisher wurden die Hochsprachen-Eigenschaften der Sprache *C* beschrieben. Dabei ist der vielleicht größte Vorteil der Sprache *C* , daß man Zeigern Hardwareadressen zuweisen kann. Erst durch diese Eigenschaft ist es überhaupt möglich gewesen, mit der Sprache *C* ein Betriebsystem wie *UNIX* zu schreiben. Diese Zeiger, welche vollständig in das Konzept der Sprache *C* integiert sind, erlauben es, Probleme aus der Assemblerebene in die Hochsprachenebene von *C* einzubinden und dann mit den Hilfsmitteln dieser komfortablen Hochsprache zu lösen. Die Programmiersprache *C* ist also als bisher einzige Hochsprache in der Lage, die Assemblersprache fast vollständig zu ersetzen. Damit stellt *C* für einen geübten Programmierer die vielleicht leistungsfähigste Entwicklungssprache dar. Eine detailierte Beschreibung der Variabeltypen von *C* ist im Anhang B.1 zu finden.

2.1.3 Das Betriebsystem *UNIX*

Die Geschichte von *UNIX* reicht bis das Jahr 1969 zurück. In diesem Jahr wurde die erste *UNIX* -Version von Ken Thompson bei den Bell-Laboratorien für einen PDP-7 Rechner in Assembler geschrieben.
UNIX geht in seinen Fähigkeiten über ein normales Betriebsystem hinaus. Es ist einerseits zwar ein Betriebsystem, andererseits ist es auch noch eine Art von Werkzeugkasten ("tool

machine") für den Programmierer. Als Betriebsystem ist *UNIX* ein Multi-Tasking-fähiges Mehr-Benutzer- System. Als Werkzeugkasten bietet es zahlreiche leistungsfähige Werkzeuge und gute Möglichkeiten, diese individuellen Ansprüchen anzupassen. Diese Eigenschaften, sowie die portierfähige Implementierung in der höheren Programmiersprache *C* , verhalfen dem System weltweit zum Durchbruch. Das System hat Einzug gehalten in allen Computerklassen: auf Hochleistungs- Mikrocomputern wird *UNIX* heute ebenso selbstverständlich eingesetzt wie auf Minicomputern und Groß-Rechenanlagen.
Das *UNIX* -System besteht im wesentlichen aus zwei logischen Schichten. Die unterste *UNIX* -Schicht wird von dem sogenannten Nucleus oder Kernel gebildet. Der Nucleus verwaltet und steuert die Hardware, er kümmert sich um das Management der Benutzer-Prozesse, ermöglicht die Kommunikation zwischen den Prozessen und führt Aufträge für Benutzer-Prozesse aus. Die Auftragserteilung an den *UNIX* -Nucleus geschieht mittels der Systemaufrufe. Weiterhin ist im Nucleus das Filesystem implementiert, so daß er auch die für die Ein- und Ausgabe verantwortliche Instanz ist.
Über dem *UNIX* -Nucleus liegt die Schicht der Benutzer-Prozesse. Die Benutzer-Prozesse kommunizieren einerseits über das Terminal mit den Benutzern, andererseits erteilen sie Aufträge an den Nucleus. Dabei werden sie von ihm überwacht und kommunizieren auch untereinander mit seiner Hilfe. Hinter den Benutzerprozessen verbergen sich nicht etwa nur benutzereigene Anwenderprogrammme. Vielmehr gehören in diese Kategorie auch alle auf den ersten Blick vermeintlichen "Systemprogramme" wie etwa der *UNIX* -Kommando-Interpreter (Shell); die Filesystem- Dienstprogramme zum Anlegen, Löschen und Kopieren von Files und Directories, kurzum alle Dienstprogramme und *UNIX* -Werkzeuge.
Der Kern des Betriebsystems, der *UNIX* -Nukleus, enthält keine Benutzerschnittstelle im eigentlichen Sinne. Der Nukleus kann nur über Systemaufrufe angesprochen werden und diese lassen sich nur aus einem Programm heraus absetzen. Aus diesem Grund ist zwischen dem Betriebsystem und dem Benutzer eine eigene Programmschicht implementiert, die sich gewissermaßen wie eine Schale um das Betriebsystem legt. Diese Vorstellung stand auch Pate, als die Schnittstelle getauft wurde: ihr Name ist Shell.
Die Shell implementiert eine Kommandoschnittstelle, über die der Benutzer alle Leistungen des *UNIX* -Systems zur Verfügung gestellt bekommt.
Da die Shell kein integraler Bestandteil des Betriebsystems ist, sondern aus der Sicht des Nucleus ein "ganz normales" Benutzerprogramm darstellt, kann es auch modifiziert werden, um etwa den Bedürfnissen spezieller Benutzerkreise gerecht zu werden. Heutzutage existieren für *UNIX* mehrere alternative Schnittstellen, von denen die menue- und windowgesteuerten die komfortabelsten sind. Weil die Shell in der Programmiersprache *C* geschrieben ist, sind Änderungen in dieser sehr einfach durchzuführen.
Zu den vielfältigen leistungsfähigen Merkmalen von *UNIX* gehört auch das hierarchische Dateikonzept. Wenn man dieses System graphisch veranschaulicht, hat es die Form eines (umgekehrten) Baumes. Gebildet wird der Baum von Dateien dreier unterschiedlicher Typen: **Directories, Normal-Files und Spezial-Files** .

Directories Ermöglicht wird diese hierarchische Struktur durch die Directories. Directories sind die Strukturträger des Dateisystems. Sie können weitere Directories enthalten, welche wiederum andere Directories enthalten können usw. Dadurch ergibt sich die baumartige Struktur.

Normal-Dateien Die Normal-Dateien sind Dateien im üblichen Sinne. Sie enthalten ASCII-Texte oder Binärmuster. Diese normalen Dateien bilden die Blätter des Dateisystem-Baumes.
Bemerkenswert ist dabei, daß die Dateien im *UNIX* -System keinerlei Strukturinformationen enthalten. Ebensowenig findet man Dateiattribute oder Verwaltungsinformationen in einer Datei. Dateiattribute und Verwaltungsinformationen, die beispielweise die Eigentumsverhältnisse und Zugriffsrechte für einen Datei beschreiben, gibt es natürlich. Nur sind sie an einer anderen Stelle aufgehoben, nämlich in den sogenannten I-Knoten ("I-Nodes"), das I steht für Information.

Spezial-Dateien Eine Spezial-Datei ist nichts anderes als ein in das hierarchische File-System eingetragenes Ein-/Ausgabe-Gerät. In *UNIX* -Systemen sind Spezial-Files in aller Regel im Directory */dev* untergebracht. Daß die Ein-/Ausgabe-Geräte als Spezial-Files logisch in das hierarchische File-System eingehängt sind, bringt zwei wesentliche konzeptionelle Vorteile mit sich. Zum einen ist der Zugriffsmechanismus auf ein Ein-/Ausgabe-Gerät identisch mit dem Zugriff auf ein normales File. Zum anderen sind für diese Geräte dieselben Zugriffschutzmechanismen wie für normale Files wirksam.

UNIX und *C* sind sicherlich nicht für einen Anfänger geeignet, sondern vielmehr als Werkzeuge für größere Softwareentwicklungen gedacht. Für diese Aufgabe bilden beide zusammen aber ein sehr leistungsfähiges System, dessen wahre Vorteile der Entwickler erst mit der Zeit zu schätzen lernt.
Damit hat sich auch im Nachhinein die Entscheidung für ein Software-Entwicklungssystem mit *UNIX* und *C* als richtig erwiesen.
Ein wichtiges Hilfsmittel für die Erzeugung von großen Programmsystemen, das MAKE-Konzept von *UNIX* wird im Anhang B.2 näher erläutert.

2.2 Allgemeines über Bussysteme

2.2.1 Einführung

Busse sind die Kommunikationsverbindungen zwischen den verschiedenen Teilen eines Computer-Systems. Die meisten Computer enthalten mehrere Busse, wobei jeder einzelne Bus an seine spezielle Aufgabe optimal angepaßt ist. Diese Art der Busse, welche praktisch in jedem Rechner vorkommen, nennt man auch private Busse, weil sie speziell auf die Bedürfnisse eines einzelnen Rechners zugeschnitten sind.
Die anderen Bus-Systeme werden auch als offene Systeme (*public buses*) bezeichnet.
Solche Busse sind mechanisch so aufgebaut, daß über Steckverbindungen weitere Module mit ihnen verbunden werden können. Dieses kann sowohl über Platinen, welche in Sockel gesteckt werden (z.B. IBM PC Bus) oder über Platinen, welche als Einschübe in Crates gesteckt werden (CAMAC, VME-Bus etc.) geschehen.
Die moderneren dieser Bussysteme sind so universell ausgelegt, daß sie praktisch allen Ansprüchen von modernen Computer-Systemen gerecht werden. Damit die an ein solches Bus-System anschließbaren Module untereinander austauschbar (kompatibel) sind, existieren für

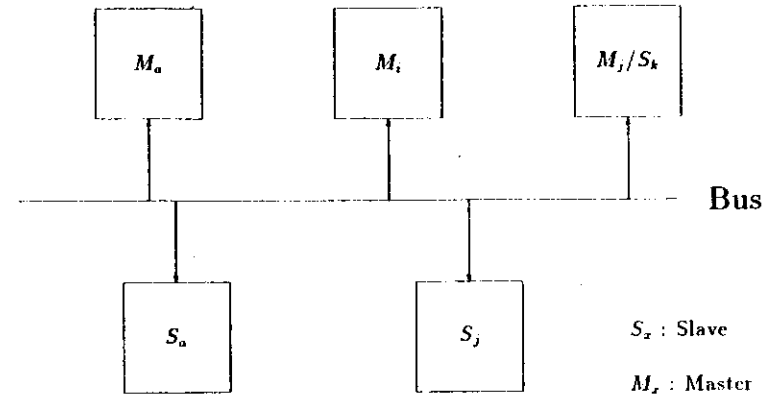


Abbildung 1: Ein Bussystem

solche Busse Normen. Nach diesen muß sich der Entwickler neuer Module richten, wenn seine Einheiten ohne Probleme mit den anderen Modulen des Busses zusammenarbeiten sollen.

2.2.2 Die Auswahllogik in einem Bussystem

Eine typische Kommunikation zwischen zwei Modulen eines Bussystems läuft wie folgt ab.
Wenn in dem Bild 1 das Modul M_i mit dem Modul S_j reden will, so sendet es bestimmte Signale auf den Bus. Sobald das Modul S_j seine Adressen auf dem Bus anliegen sieht, antwortet es dem Modul M_i. Ein Busteilnehmer, welcher wie in diesem Fall das Modul M_i eine Busverbindung einleitet, wird auch als MASTER bezeichnet. Entsprechend wird das Modul, welches auf diese Anforderung antwortet SLAVE genannt.
Es gibt dabei auch Busteilnehmer, welche zeitlich nacheinander sowohl MASTER, als auch SLAVE sein können. Es würde nun ein Durcheinander geben, wenn mehrere MASTER versuchen würden, gleichzeitig auf den Bus zuzugreifen. Deshalb existiert in einem solchen Bus, welcher mehrere MASTER zuläßt, ein Mechanismus der verhindert, daß mehrere MASTER gleichzeitig auf den Bus zugreifen. Ein solcher Mechanismus, der den Buszugriff für andere MASTER blockiert, sobald ein MASTER auf den BUS zugegriffen hat, wird ARBITRATION genannt. Nun ist zu klären, wie die Signale über die Leitungen gelangen.

2.2.3 Parallele und Serielle Busse

Die Leitungen auf einem Bus dienen dazu, Daten und Adressen zu transportieren. Dabei wird der Transport entweder durch eine Synchronisation über ein zentrales Taktsignal oder über weitere Kontrollsignale kontrolliert.

Ferner existieren auf einem Bus noch weitere Signale, welche für die Dekodierung von Prioritäten sowie die ARBITRATION zuständig sind. Bei der Zuordnung der Signale zu den Leitungen gibt es im Prinzip zwei Systeme, nämlich das parallele und das serielle Bus-Konzept. Bei einem parallelen Bus hat jedes Signal seine eigene Leitung, was zu einer schnellen Übertragung führt. Ein serieller Bus hingegen besteht im allgemeinen aus nur zwei Leitungen. Während über die eine Leitung nacheinander alle erforderlichen Signale ablaufen, läuft über die andere Leitung ein Taktsignal, welches die anderen Signale beim Empfänger synchronisiert.
Außerdem gibt es noch Mischformen zwischen diesen beiden Systemen, bei denen z.B. 32 Datensignale zeitlich nacheinander über 16 Datenleitungen gelangen. In einem solchen Fall spricht man von gemultiplexten Signalen.

2.2.4 Interrupts

Die Interrupts dienen dazu, die laufende Arbeit eines Prozessors für Ein- oder Ausgaben zu unterbrechen.
Zwar kann man sich ein einfaches Ein-/Ausgabesystem vorstellen, bei welchem der Prozessor ständig die Peripheriegeräte abfragt, ob irgendetwas für ihn vorliegt, aber gerade mit einer größeren Zahl von Ein- /Ausgabegeräten ist dieses Verfahren ineffizient (sogenanntes *Polling*).
Bei einem interruptgesteuerten System können die Geräte dem Prozessor selber mitteilen, wann er sich um sie kümmern soll. Wenn eine solche Interruptanforderung (Interruptrequest) bei dem Prozessor angekommen ist, wird zuerst der aktuelle Befehl beendet und danach der momentane Prozessorzustand abgespeichert. Anschliessend kann der Prozessor dieser Anforderung nachkommen, indem er die der Anforderung zugeordnete Programmroutine (interrupt service routine) abarbeitet.
Sobald diese Routine beendet ist, wird der alte Prozessorzustand in den Prozessor zurückgeladen und die Ausführung des ursprünglich unterbrochenden Programmes wieder aufgenommen.
In vielen Systemen gibt es auch mehrere Interrupts mit unterschiedlicher Priorität. Hier kann ein Interrupt eine durch einen anderen Interrupt gestartete Programmroutine unterbrechen. Man spricht in diesem Fall davon, daß mehrere Interruptebenen existieren.

2.2.5 DMA

Wenn Daten von außerhalb in den Speicher eines Rechners geladen werden sollen, so kann dieses der Prozessor selbst durchführen. Effektiver ist es jedoch, dieses durch einen sogenannten DMA-Kanal (DMA-Channel) erledigen zu lassen. Ein DMA-Kanal (Direct Memory Access) ist dabei ein Hardware System, welches es ermöglicht, Daten von einem Gerät direkt in den Speicher eines anderen Gerätes zu schreiben. Dabei wird das DMA- System vorübergehend MASTER des Rechnersystems und belastet dabei nicht den Prozessor mit dieser Aufgabe. Während des Transfers muß der Prozessor entweder warten oder er kann seine Arbeit an einem noch freien Bus fortsetzen.
Derartige DMA Kanäle stellen ein sehr leisungsfähiges Konzept dar, um Daten von einem Speicher in einen anderen zu transportieren.

2.2.6 Interface-Systeme

In praktisch vorkommenden Systemen ist mehr als nur ein einziger Bus vorhanden. Z.b. können sich in den an einen Bus angeschlossenen Modulen weitere Busse befinden, welche den Datentransfer innerhalb eines solchen Moduls regeln. D.h. es muß einen Übergang von einem Bus zu einem anderen geben. Einen solchen Übergang zwischen zwei verschiedenen Bussen bezeichnet man allgemein als Interface.
Ein solches Interface muß sowohl physikalisch, als auch logisch die von einem Bus kommenden Signale auf einen anderen Bus umsetzen. Dabei kann es auch passieren, daß die Signale in ihrer zeitlichen Reihenfolge umgesetzt werden müssen. Daraus ist ersichtlich, daß der Aufwand für ein solches Interface sehr davon abhängt, wie ähnlich die Bussysteme einander sind.
Ein Interface im eigentlichen Sinne wird dadurch realisiert, daß man für jeden der beiden vorhandenen Busse ein Register hat. Diese beiden Register sind über ein logisches Netzwerk so miteinander und mit den Bussen verbunden, daß die Signale des einen Registers auf die des anderen Registers übertragen werden. Wenn man bei einem solchen Interface etwas in ein Register schreibt, so erscheint dasselbe auch in dem Register für den anderen Bus. So wurde z.B. das Interface zwischen dem Q- und dem *VME* -Bus bei dem *BOP* -System (siehe B.7) realisiert.
Es gibt auch leistungsfähigere aber aufwendigere Interface-Systeme, um zwei Busse miteinander zu verbinden. Beispiele sind hier ein Dual-Ported Ram (DPR) mit zwei verschiedenen Buszugängen oder ein Interface, welches selbst zu einem DMA in der Lage ist. Auch diese beiden Systeme sind in dem *BOP* -System vorzufinden. Das erste System findet sich dort als die Mailbox, welche einen schnellen Datentransfer zwischen dem *VME* - und dem *VMX* - Bus erlaubt. Das zweite System ist dort als der Link zur IBM anzufinden, welcher nach einem entprechenden Startbefehl die Daten auf der *VME* -Seite per DMA zu oder von der IBM schafft.

2.2.7 Die Leistungsfähigkeit von Bus-Systemen

Das wichtigste Leistungskriterium für ein Bus-System ist die Geschwindigkeit, mit der ein Datentransfer durchgeführt werden kann. Zusätzlich spielen noch Kriterien , ob mehrere Module des Busses MASTER werden können (Multi- Master Fähigkeit), der mechanische Aufbau, sowie die sonstigen Optionen (Anzahl der Interruptebenen usw.) eine wichtige Rolle.
Für die Geschwindigkeit eines Busses ist neben der Anzahl der Signalleitungen die Taktfrequenz, mit der die Übertragung stattfindet, maßgeblich. Für diese Taktfrequenz besteht aber aus physikalischen Gründen eine obere Grenze. Einmal bilden die Leitungen eines Busses ein System von Kapazitäten und Induktivitäten, welche gerade bei hohen Frequenzen den Verlauf der Signale verfälschen. Zum anderen fließen in der Nähe eines solchen Busses durch andere Rechnerbauteile hochfrequente Ströme. Diese koppeln sich in die Busleitungen ein und führen zu zusätzlichen Störungen. Diese Faktoren führen dazu, daß die maximale praktisch zu erreichende Taktfrequenz eines 12 Slot *VME* -Busses bei etwa 50 MHz liegt. Trotzdem ist der *VME* -Bus , den ich im folgenden näher beschreiben werde, einer der leistungsfähigsten derzeit existierenden Bussysteme.

2.3 Der *VME* -Bus

Ursprünglich stammt dieses Bussystem von den vier Firmen Motorola, Mostek, Philips/-Signetics und Thomson CSF. Weil Motorola nicht nur bei der Entwicklung dieses Bussystems beteiligt war, sondern auch den 16-Bit Mikroprozessor MC 68000 entwickelt hat, unterstützt der *VME* -Bus das Bussystem dieses Mikroprozessors besonders gut. Der Mikroprozessor und der *VME* -Bus haben sich in den letzten Jahren zum Industriestandard entwickelt und insbesondere in Europa eine sehr weite Verbreitung erreicht.
Neben seiner großen Leistungsbreite vom 8- Bit Prozessor- bis zum 32-Bit Minirechner- System haben hierzu insbesondere das in Europa sehr verbreitete Europakartenformat, sowie die günstigen und zuverlässigen 96-poligen Steckverbindungen beigetragen. Um mit dem *VME* -Bus über einen 24-bit-Adreß- und 16-bit-Datenbus zu verfügen, wird wenigstens eine Einfach-Europakarte mit einem einzelnen 96-poligen DIN- Stecker benötigt. Wenn man hingegen den vollen 32-Bit-Adreß- und Datenbereich benutzen will, so muß hierfür eine Doppeleuropakarte mit zwei 96-poligen DIN- Steckern eingesetzt werden.
Mit einem solchen Aufbau besitzt der *VME* -Bus dann die folgenden Leistungsmerkmale:

- die Unterstützung von Mikroprozessor-Architekturen bis zu 32-Bit Wortbreite
- die Unterstützung von Multiprozessor-Systemen
- ein Datendurchsatz von bis zu 20 Millionen Byte/s
- ein vollständig asynchrones, multiplexfreies Busprotokoll

Ein Grund für die hohe Leistungsfähigkeit und Modularität dieses Bus-Systems ist dabei sicher auch die Gliederung des *VME* -Busses in vier unabhängige Teilsysteme:

- Der **Daten-Transfer-Bus (DTB)** enthält alle Daten- und Adressleitungen, sowie die zum Datentransfer notwendigen Steuerleitungen.
- Der **Arbitrations-Bus** liefert alle Signale, die zur Steuerung in einem Multi-Master-System notwendig sind.
- Der **Interrupt-Bus** dient zur Behandlung von Unterbrechungsanforderungen.
- Unter dem Begriff **"Versorgungs- und Hilfsleitungen"** werden schießlich alle restlichen Signale, z.B. die Stromversorgung und die Fehlererkennung zusammengefaßt.

Im Anhang B.3 wird das System des *VME* -Busses anhand der Beschreibung seiner einzelnen Unterbusse näher erläutert. Signale bei denen der aktive Zustand gleich der Masse ist, werden in den Texten durch eine über den Namen gezogene *Linie* oder in den Abbildungen durch einen hochgestellten Stern (*) gekennzeichnet.

2.4 *VMX* -Bus

Bei einem Multiprozessorsystemen existieren im allgemeinen sowohl globale als auch lokale Speicher. Die globalen Speicher, auf welche alle Prozessoren zugreifen können, liegen bei einem *VME* -System an dem *VME* -Bus an. Die lokalen Speicher hingegen liegen typischerweise in einem solchen System auf den jeweiligen Prozessorkarten. Nun kann es vorkommen,

daß der lokale Speicher auf den Prozessorkarten nicht ausreicht und erweitert werden muß. Zusätzlicher globaler Speicher kann diesen lokalen Speicher im allgemeinen nicht ersetzen, weil sich durch den zeitlich nacheinander ablaufenden Zugriff auf diesen Speicher dessen effektive Zugriffszeit erhöht (sogenannter *arbitration overhead*). Für diesen Zweck und für zusätzliche Ein- und Ausgaben wurde der *VMX* -Bus geschaffen.
Der *VMX* -Bus ist also als ein zusätzlicher lokaler Bus bei *VME* -Systemen gedacht. Sein Anschluß erfolgt über die noch freien Anschlüsse vom unteren 96-poligen DIN-Stecker, der Doppeleuropakarte (siehe Abb. 37).
Er verfügt über einen 32-Bit Datenpfad mit einem Adressraum von bis zu 16 Megabyte. Die hierfür notwendigen 24-Bit Adressignale werden über 12 Adressleitungen durch 2 *STROBE* -Signale gemultiplexed.
Während der *VMX* -Bus also normalerweise als lokaler und der *VME* -Bus als globaler Bus eingesetzt wird, wurde es bei dem *BOP* -System genau anders herum gemacht. Der Gedanke dabei war, daß der *VME* -Bus durch die nicht gemultiplexten Adressleitungen etwas schneller ist und deshalb dort einzusetzen ist, wo mehr mehr Datentransfers stattfinden. Dieses war bei dem *BOP* -System bei der Verbindung zwischen den CPU-Boards und den Floatingpointkarten gegenüber dem Transfer zur Mailbox der Fall.

2.5 Das PADAC-System

2.5.1 Einführung

Das Wort *PADAC* ist eine Abkürzung für *PA*rallel *D*ata *A*cquisition and *C*ontrol. Es handelt sich dabei um eine von der DESY-Gruppe F58 geschaffene interne Crate- und Bus-Norm, welche unter anderem für Datenverbindungen zur *IBM* benutzt werden kann.
Bei der Entwicklung des *PADAC* -Systems war es das Ziel, eine relativ einfache Busnorm mit mechanisch stabilen Crates und Einschüben zur Verfügung zu haben. Deshalb sind die Module auch relativ niedrig integriert, so daß keine Lüftung notwendig ist. Interrupts und DMA-Übertragungen sind von jeder Crateposition aus möglich und das Timing für die Interrupts und DMAs befindet sich vollständig im Crate-Controller, weshalb die anderen Module einfacher gehalten werden können.
Um eine Verbindung von einem Kleinrechner zur DESY-*IBM* herzustellen, benötigt man als Einschübe in das Crate einen Crate-Controller (z.B. CC1,CC2 oder CC3), ein Interface zur *IBM* -Verbindung (EXPU), sowie, falls es existiert, ein Interface zum Kleinrechner. Im Falle der *IBM* -*VME* - Verbindung befindet sich das letztere im *VME* -Crate.
Das EXPU-Modul besitzt ein internes FIFO (First In First Out), von welchem die Daten in serieller Form zu einem Knotenrechner mit einem PADAC MICRO-Modul gelangen. Die serielle Leitung von dem EXPU-Modul zu dem Knotenrechner kann dabei mehrere hundert Meter lang sein. Dieser regelt den Datentransfer zwischen den verschiedenen EXPU-Modulen und der IBM.
Eine detailierte Beschreibung der beiden Module (CC1 und EXPU) ist im Anhang B.5 zu finden.

2.5.2 Allgemeines zur Kommunikation mit der *IBM*

Wenn die Hardware installiert ist und die Kommunikation mit dem auf dem MICRO laufenden Spiegelprogramm funktioniert, kann man versuchen eine Verbindung mit der *IBM*

herzustellen. Dabei muß man sich bei dieser Kommunikation natürlich an das vorher besprochende Format halten.
Eine detailierte Beschreibung dieser Befehle ist in dem EXP-Manual [16] zu finden, weshalb ich hier nur drei am Anfang besonders wichtige Befehle beschreibe und zwar **CHANGE**, **START < Online-Modul >** und **LOAD < Service-Programm >**.

CHANGE Dieser Befehl bewirkt eine Art von Reset, d.h. durch diesen Befehl wird die Datenverbindung in einen definierten Ausgangszustand gebracht. Falls die Datenverbindung eine gewisse Zeit nicht in Betrieb war, so kommt etwa eine Minute lang nur die Antwort

ONLINE - JOB STARTING.

Wenn alles in Ordnung ist, so wird danach die Meldung

GIVE MESSAGE

gesendet.

START < Online-Modul > Nun kann auf der *IBM* ein sogenannter Online-Job gestartet werden, welcher die für die Benutzerprogramme notwendige Prozeßumgebung schafft. Bei dem von mir benutzten Anschluß lautete der Befehl z.B. START TAL00.
Nun muß die *IBM* solange abgefragt werden, bis die Antwort

START TAL00 REQUEST ACCEPTED

kommt.

LOAD < Service-Programm > Jetzt kann mit diesem Befehl das Benutzermodul geladen und gestartet werden. Das Benutzermodul ist das Benutzerprogramm auf der *IBM* , welches nach gewissen Konventionen geschrieben sein muß und einen bestimmten Satz von JCL-Karten benutzt. Auch bei diesem Befehl muß gewartet werden bis die Antwort

LOAD < Service - Programm > REQUEST ACCEPTED

kommt.

2.5.3 Der IBM-Befehl *ONLINE*

Dieser Befehl stellt ein weiteres Hilfsmittel dar, um festzustellen wie der Zustand der Datenverbindung von der *IBM* aus ist. Dieser Befehl kann auf der *IBM* z.B. unter Newlib mittels der Zeichenfolge ((ONLINE)) ausgeführt werden.
Daraufhin erscheint auf dem *IBM* -Terminal eine Liste aller mit der *IBM* verbundenen EXPU-Module. In dieser Liste wird u.a. angegeben, welcher Online-Job und welches Benutzermodul im Moment für ein bestimmtes EXPU-Modul läuft. Wenn die Datenverbindung gerade aktiv ist, so sind diese Eintragungen in der Tabelle nach links vorgerückt.
Auffallend ist, daß die EXPU-Module als MXX, also als ein M mit einer zweistelligen Nummer bezeichnet werden. Diese Nummer wird binär über sogenannte Dip-Switches in dem EXPU-Modul eingestellt. Dieses sollten aber nur Leute der Gruppe F58 machen, weil diese wissen welche Kennnummer noch nicht belegt ist.

3 Ein Multi-Prozessorsystem für Strahlberechnungen

3.1 Problemstellung

Die Entwicklung dieses Systems, im folgenden *BOP* (*B*eam *O*rbit *P*rozessorsystem) genannt, reicht bis in das Jahr 1983 zurück.
In dieser Zeit wurden intensive Untersuchungen von Strahlinstabilitäten im HERA-Protonenring durchgeführt. Bei solchen Untersuchungen verfolgt man die Bahn der Teilchen numerisch durch die Strahlführungselemente. Die Berechnungen mußten in der Praxis für eine große Anzahl von Teilchen mit verschiedenen Anfangswerten der Bahn und über eine größere Zahl von Umläufen durchgeführt werden.
Solche Programme benötigen im allgemeinen sehr viel Rechenzeit, haben jedoch eine relativ einfache Struktur und ein verhältnismäßig geringes Ein- und Ausgabevolumen. Aus diesen Gründen stellen diese Programme ein potentielles Anwendungsgebiet für Mikroprozessoren dar.
Wegen dieser Überlegung entstand die Idee, für diese Programme einen Spezialrechner zu bauen, welcher bei diesen Programmen in etwa die äquivalente Rechenleistung von einer IBM-3081 hat. Damit sollte die DESY-IBM Rechenanlage von diesen rechenintensiven Programmen entlastet werden. Als Modell und erste Anwendung diente das Programm *RACE-TRACK* von A.Wrulich [8], mit welchem er Untersuchungen zur erforderlichen Apertur der Magnete des HERA-Protonenringes anstellte.
Damit man diese Simulationsberechnung durchführen kann, sind verschiedene Arten von Daten notwendig. Da sind zum einen die Startwerte (x_0, y_0, x'_0, y'_0), welche die Startkoordinaten und die Anfangsgeschwindigkeiten des Teilchens angeben, von welchem die Bahn simuliert werden soll. Dann sind da als zweites die Magnetdaten, welche die Parameter für den Aufbau des Speicherringes darstellen. Hierzu gehören die Werte für die Dipole, Quadrupole, Sextupole so wie der höheren Multipole und der Driftstrecken. Als drittes gibt es noch das ausführbare Programm, durch welches mit den vorherigen Daten die eigentlichen Berechnungen durchführt und die vierte Datenart die Ergebnisse erstellt werden. Die Ergebnisse sind die Koordinaten und die Geschwindigkeiten (x_n, y_n, x'_n, y'_n) eines Teilchens nach jedem Umlauf.
Die Idee des *BOP* -Systems besteht darin, die Bahn von jeweils einem einzelnen Teilchen im Speicherring mit der Hilfe eines Mikroprozessors zu berechnen. Durch den Einsatz von mehreren parallelen Mikroprozessoren kann man im Prinzip beliebig viele Teilchenbahnen parallel berechnen, wodurch man theoretisch eine beliebig hohe effektive Rechengeschwindigkeit (im Vergleich mit einem einzelnen Prozessor) erhält. Jeder der Prozessoren erhält dasselbe Programm und hat dieselben Magnetparameter als Daten. Lediglich die Startwerte der Teilchenbahnen sind für jeden Prozessor verschieden. Die Steuerung aller dieser Prozessoren unterliegt einem weiteren Prozessor, welcher im folgenden Controller genannt wird.
Das Programm, welches die Bahn der Teilchen durch den Speicherring in den einzelnen Mikroprozessoren berechnet, besteht nur aus etwa 200 FORTRAN-Befehlen. In diesem Programm wiederum besteht der rechenaufwendigste Teil aus Polynomberechnungen der Form

$$P_n(x) \quad \sum_{i=0}^{n} a_i x^i \qquad (1)$$

welche nach dem bekannten Horner Schema als

$$P_i = a_i + x\,P_{i+1} \quad \forall\, i \in \{n-1, n-2, \ldots 0\}\ und\ P_n = a_n \qquad (2)$$

geschrieben werden können, was den Rechenaufwand beträchtlich verkleinert. Diese Berechnungen müssen allerdings, wie von P.Wilhelm gezeigt wurde [10], mit doppelter Genauigkeit (64-Bit) erfolgen.
Bereits relativ früh wurde ein *UNIX* -Entwicklungssystem [2] von PCS [11] mit dem Mikroprozessor MC 68000 [12] als Zentraleinheit ausgewählt. Ferner wurde beschlossen, diesen Mikroprozessor mit einer externen, selbst zu bauenden Floating-Point Hardware zu verwenden. Weil die Strahlverfolgungsprogramme praktisch keine Divisionen benötigen, sollte dieser Befehl von der Hardwarearithmetik ausgeschlossen werden, so daß diese Einheit nur die Befehle Addition, Subtraktion und Multiplikation durchführen kann. Während die Divisionen zuerst per Software durchgeführt werden sollten, entstand später die Idee diese auf dem arithmetischen Coprozessor vom Typ NS 16081 auf den *DATA-SUD* Karten durchzuführen.
Als eine mit relativ niedrig integrierter Technik aufgebaute Floating-Point Einheit bereits funktionsfähig war, kamen Mitte 1984 hochintegrierte Floating-Point Chips der Firma Weitek [13] auf den Markt. Wegen des wesentlich einfacheren Aufbaus wurde deshalb eine neue Floating-Point Einheit mit diesen Chips konstruiert.
Auch bei dem Anschluß der Mikroprozessorsysteme an das Entwicklungssystem wurden in der Vorbereitungsphase verschiedene Strategien verfolgt. Während diese Untereinheiten zuerst über Terminalschnittstellen mit dem Host-Rechner verbunden werden sollten, ging man bei späteren Überlegungen davon aus, daß die Mikroprozessoren auf einem gemeinsamen (globalen) Bus zugreifen sollten. Dieser globale Bus sollte dann über ein paralleles Interface mit dem Bus des Entwicklungssystems verbunden werden. Diese Art der Verbindung wurde dann beim endgültigen Entwurf noch einmal erweitert, wie man im nächsten Abschnitt sehen kann, der den endgültigen Aufbau beschreibt.

3.2 Hardware

3.2.1 Übersicht

Der Aufbau des *BOP* -Systems ist in der Abbildung 2 zu sehen. Auf der untersten Ebene befinden sich die 10 Rechner, Target 0 bis Target 9 genannt, welche die eigentlichen Strahlberechnungen durchführen. Sie bestehen aus jeweils zwei Untereinheiten, dem Rechner (68K CPU) und der Fließkommarecheneinheit (FPU), welche durch einen *VME* -Bus miteinander verbunden sind. Von jedem dieser 10 Targets gehen wiederum zwei Leitungssysteme aus. Das eine Leitungssystem enthält Steuerleitungen, welche von der 68K CPU in die Arbitration-Einheit führen. Das andere Leitungssystem ist der VMX-Bus, durch welchen alle 68K CPUs mit der Mailbox verbunden sind. Über diese Mailbox, ein von 2 Seiten adressierbarer Speicher, erfolgt der Austausch sämtlicher Daten zwischen dem an sie angeschlossenen VMX- und VME-Bussen.
Um die Übersicht innerhalb dieses komplexen Rechnersystems zu erhöhen, kann man das System in drei funktionale Ebenen untergliedern. Die eben besprochenen Targets gehören dann zur sogenannten **Targetebene**. Alle Geräte, welche direkt mit dem an der Mailbox befindlichen VME-Bus verbunden sind, werden dann zur **Controllerebene** gerechnet. Diese Ebene hat ihren Namen durch die an diesen Bus angeschlossene 68K CPU(Controller

genannt), welche die an diesen Bus angeschlossenen Geräte kontrolliert, bekommen. Von diesem VME-Bus geht es dann entweder über das VME-Q-Bus Interface zur PCS-Workstation oder über das VME-PADAC Interface zur IBM. Während man die IBM nicht mehr zu dem BOP-Rechnersystem zählen kann, bildet die PCS-Workstation die **Entwicklungssystemebene** des BOP- Systems. Über diese Ebene und die an ihr angeschlossenen Terminals wird außerdem das gesamte BOP-System gesteuert. Über die Terminalschnittstelle läßt sich nämlich von hier aus der Controller steuern, welcher wiederum über die Arbitration-Einheit die Targetebene steuert.

3.2.2 Funktionsweise

Das *BOP* -System ist aus den folgenden drei Ebenen aufgebaut:

Entwicklungssystemebene: PCS-Workstation

Controllerebene:	68-k Controller (DSSECPUA1-Rechner)
	Arbitration-Einheit
	VME-PADAC Interface
	VME-Grafik Karte
	Mailbox (256kB DPR *VME* / *VMX*)
Targetebene:	10 × 68k-Rechnereinheiten
	(DSSECPUA1-Rechner)
	10 Floating-Point Prozessoren

Zur Steuerung und Programmentwicklung des Rechnersystems dient eine *PCS* -Workstation vom Typ QU-68000. Sie besitzt eine 80 MB Festplatte, hat ein 20 MB Streamerlaufwerk und verwendet intern einen Q-Bus.
Nachdem die Tracking-Programme auf diesem Rechner kompiliert und zusätzliche Funktionen in der Form von Programmen an sie angebunden wurden, gelangen sie über das Q-Bus-VME Interface in die Mailbox der Controller-Ebene. Dieser Datentransfer wird auf der Controllerebene durch den 68k-Controller kontrolliert. Sobald das Programm in der Mailbox angelangt ist, können die einzelnen Targetrechner über die Arbitration-Einheit aufgefordert werden, sich das Programm über den *VMX* -Bus abzuholen.
Hierfür wird dem Betriebsystem der Targets (*oper.c & mail.c* in */usr/bop/race*) über ein Signal aus der Arbitration-Box mitgeteilt, daß es sich das Programm oder die Daten mit den Magnetparametern aus der Mailbox abholen soll. Anschließend wird das jetzt im Speicher der Targets befindliche Programm gestartet, welches von dem Controller über die Mailbox die Startwerte anfordert. Diese Startwerte werden dann von Hand auf der PCS eingegeben, von wo aus sie über die Terminalschnittstelle zum Controller und über die Mailbox schließlich zu dem betreffenden Target gelangen. Während die Programme in dem Speicher der 68000-er Target-Karte bleiben, werden die Daten mit den Magnetparametern sofort über den *VME* -Bus des Targets in den statischen Speicher des Floating-Point Prozessors geladen.
Nachdem der Ladevorgang für die Targetrechner abgeschlossen ist, können sie über die Arbitration-Einheit den Befehl bekommen, mit der Ausführung des Programmes zu beginnen.

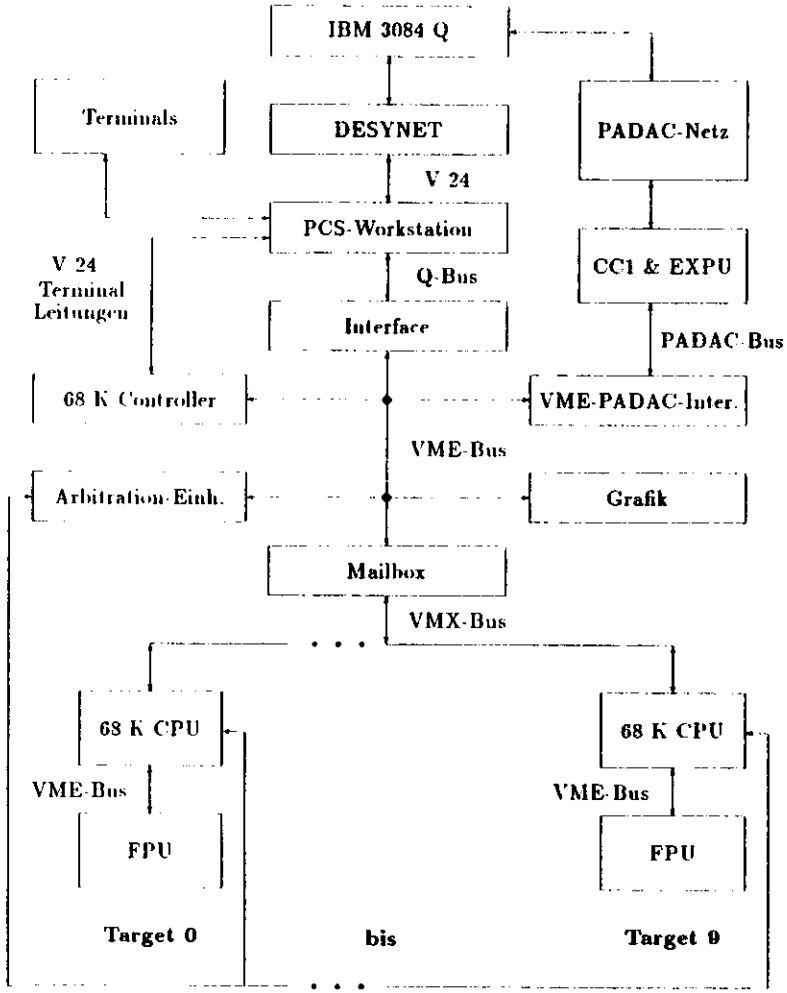


Abbildung 2: Prinzipieller Aufbau des *BOP* -Systems

Sobald ein Target die ersten Ergebnisse berechnet hat, fordern es über die Arbitration-Einheit den *VMX* -Bus für sich an, um die Ergebnisse in die Mailbox zu schreiben. Wenn der *VMX* -Bus frei ist und auch von dem *VME* -Bus kein Zugriff auf die Mailbox vorliegt, erhält das Target die Erlaubnis, die Mailbox benutzen zu dürfen. Allen anderen Rechnern wird jetzt der Zugriff auf die Mailbox solange verwehrt, bis die Arbitration-Einheit durch das Target die Nachricht erhält, daß der *VMX* -Bus wieder freigegegben ist. Nun kann der Bus an ein weiteres Target vergeben werden. Die Ergebnisse nach einem Umlauf sind typischerweise der Ort und die Geschwindigkeit des Teilchens, sowie die Nummer des momentan berechneten Umlaufes.
Falls im Moment gerade kein Targetrechner auf die Mailbox zugreift, so kann dieses der Controller tun. Einen solchen Zugriff auf die Mailbox führt der Controller regelmäßig während einer Simulationsberechnung durch, um sich dort die von den Targets hinterlegten Ergebnisse abzuholen. Die Ergebnisse können dann direkt über die Grafik ausgegeben und entweder über das *PADAC* -*VME* - Interface zur IBM oder über das Q-Bus *VME* -Interface erst einmal zur PCS gebracht werden. Auf der PCS gelangen die Daten mit den Ergebnissen auf eine Winchesterplatte und können von dort, falls der Platz auf dieser Platte nicht ausreicht, auf Streamerkassetten ausgelagert werden.
Daß bereits während eines Programmlaufes die Ergebnisse auf der Grafik der Controllerebene ausgegeben werden können, hat den Vorteil, daß man u.U. bereits bei einer noch laufenden Berechnung entscheiden kann, ob sich eine weitere Auswertung der Daten lohnt.
Zu dem Datentransfer zur IBM ist anzumerken, daß dieser entweder über das DESYNET oder über das von mir installierte *VME* -PADAC Interface geschehen kann. Weil das DESYNET aber etwa 100 mal langsamer als das PADAC-Netz ist, sollte man letzteres vorziehen, selbst wenn das bedeutet, daß während dieses Datentransfers parallel keine Ergebnisdaten zur PCS gebracht werden können.
Bis jetzt wurde das Zusammenwirken der einzelnen Systemkomponenten erläutert. Eine detailierte Beschreibung der einzelnen Ebenen ist hingegen in den Anhang B.6 zu finden.

3.3 Systemsoftware des BOP-Systems

3.3.1 Übersicht über die Systemsoftware

Zu der Systemsoftware des *BOP* -Systems gehören alle Programme, welche direkt oder indirekt dafür sorgen, daß auf dem *BOP* -System Anwendungsprogramme laufen können. Auf dem Entwicklungssystem der PCS-Workstation sind dieses die Programme, welche aus dem Source-Code (Fortran 77, C oder Assembler) auf dem Controller oder den Targetrechnern lauffähige Programme machen.
Hierfür wird im ersten Schritt der Source-Code mittels eines Compilers oder Assemblers in einen auf dem Entwicklungssystem ausführbaren Code umgewandelt. Dieser wird dann disassembliert, so daß ein Textfilterprogramm die Floatingpointbefehle für die PCS gegen die Floatingpointbefehle für die Floatingpoint Recheneinheit der Targets austauschen kann. Anschließend wird dieses neu entstandene File wieder assembliert und in ein Format gebracht (S-Records), welches einen sicheren Transport über die Schnittstellen gestattet. Dieser gesamte Ablauf wird übrigens durch eine Kommandoprozedur (*@run* aus */usr/bop/com*) geregelt.
Auf der Controller- und der Targetebene dienen die Systemprogramme hauptsächlich dazu, den Datenaustausch über die Schnittstellen und die Mailbox zu regeln. Zum anderen sorgen

diese Programme dafür, daß man von Hochsprachen aus die Targetrechner über den Controller steuern kann.
Abgesehen von der Systemsoftware für die parallele Schnittstelle, welche von mir praktisch neu geschrieben worden ist, sind sämtliche weiteren Informationen über die Systemsoftware im Anhang zu finden. Sowohl für die Systemsoftware auf den Targets (Anhang C.1), der Mailbox (Anhang C.2), dem Controller (Anhang C.3), sowie dem Entwicklungssystem (Anhang C.4) sind Details im Anhang zu finden. Dieses gilt auch für eine Beschreibung der Funktionen für die parallele Schnittstelle (siehe Anhang C.3.1).

3.4 Systemsoftware für die parallele Schnittstelle

3.4.1 Ausgangssituation

Wie schon oben beschrieben, besteht die Hardware der parallelen Schnittstelle sowohl aus gekauften, wie auch aus selbstgebauten Teilen. Ähnlich verhält es sich auch mit der Software, genauer gesagt mit dem Device-Treiber, welcher diese Schnittstelle in das *UNIX* - Betriebsystem einbindet. Als die Software gekauft wurde, stellte sich heraus, daß sie nicht funktionierte. Deshalb wurde sie mehrmals, teils mit der Rücksprache von der Firma *PCS* geändert, bis sie schließlich einigermaßen lief.
Es war jetzt zwar möglich Daten von der Entwicklungssystem- zur Controllerebene zu schicken, aber der Datentransfer in die andere Richtung, vom Controller zum Entwicklungssystem, funktionierte praktisch nicht. Dabei war auffällig, daß der Datentransfer, jedesmal nachdem das *UNIX* -System neu hochgefahren worden war, einmal einwandfrei funktionierte.
Während man auf dem *UNIX* -System die normalen *C* - bzw. Fortran- I/O Befehle verwenden konnte, wurden für die Controllerebene eigene Befehle geschaffen, welche in das Mailsystem *mail.c* integriert wurden. Dieses war der Stand, als ich begann mich mit dieser Schnittstelle zu beschäftigen.
Es stellte sich heraus, daß das *UNIX* -Betriebsystem als Puffer ein FIFO (First In First Out) mit einer Länge von bis zu ein paar tausend Worten aufbaut. Dieses FIFO ließ sich nur löschen, indem man das Rechnersystem neu hochfährt, was die obige Beobachtung bestätigt.

3.4.2 Lösungsmöglichkeiten:

Um dieses Problem zu lösen gab es zwei Möglichkeiten. Die erste Möglichkeit wäre gewesen den Devicetreiber für die Schnittstelle neu zu schreiben, bzw. zu versuchen seinen Fehler zu finden. Dieses wäre mit Sicherheit die eleganteste Methode gewesen, aber der Zeitaufwand hierfür war nur schwer abzuschätzen.
Der zweite Ansatz war zwar weniger elegant, aber dafür in seinem zeitlichen Aufwand gut abzuschätzen. Er bestand darin, Kommunikationsbefehle zu erstellen, welche die Eigenarten der Schnittstelle für den Benutzer ausgleichen. Ich entschied mich für die sichere zweite Methode, wobei ich die dafür notwendigen Befehle auf der Controllerebene in das Programm *mail.c* (in */usr/bop/race*) integriert habe. Auf der Entwicklungssystemebene mußte ich eine Bibliothek mit neuen I/O Funktionen für diese Schnittstelle schreiben, welche *pcs_mail.c* heißt und in der Bibliothek */usr/bop/sys* steht.

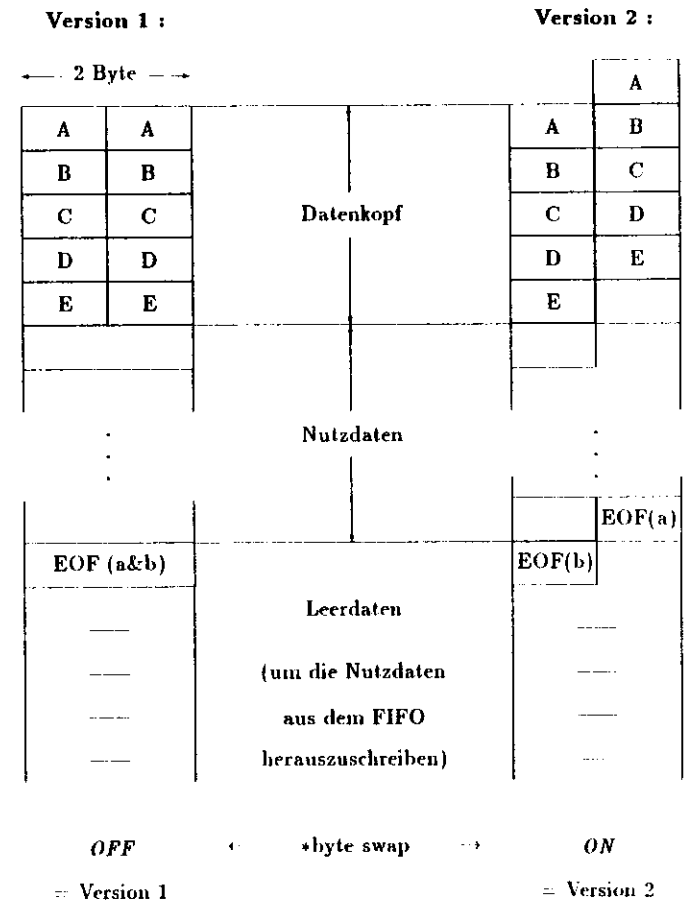


Abbildung 3: Die zwei Versionen des Datenblocks (Prinzip) 68 K Controller → *PCS*

3.4.3 Funktionsprinzip der Lösung

Damit das Empfangsprogramm auf der *PCS* feststellen kann, wann die gesendeten Daten anfangen, wird den Daten vom Sendeprogramm (in *mail.c*) eine Folge von Codewörtern als Kopf vorangestellt. Das Empfangsprogramm (in *pcs_mail.c*) überprüft dabei ständig, ob es gerade den Kopf der Daten einließt oder nicht. Sobald der Datenkopf eingelesen ist, wird erkannt ob die 16-Bit Datenworte um ein Byte versetzt ankommen oder nicht und die Variable *byte swap* auf den entsprechendem Wert gesetzt. Nun können die Daten von den entsprechenden Befehlen (*univ_read*) eingelesen werden. Dieses geschieht solange, bis ein entsprechendes *EOF*-Daten Wort (*End Of File*) eingelesen wird. Den Variablenfeldern, welche von dem Controller gesendet werden, ist immer die Länge der Felder in Bytes vorangestellt und nach der hier verwendeten Konvention bedeutet ein Feld, welches aus Null Bytes besteht, ein *EOF*. Von diesem Punkt an wird der Datentransfer auf der *PCS* an die Routine *dclose* übergeben. Weil die Länge des FIFO unbekannt ist, muß das Sendeprogramm nach den Nutzdaten weiter Datenworte zum Entwicklungssystem schreiben, um das FIFO zu leeren. Die Routine *dclose* auf der *PCS* liest die Daten nach dem *EOF* solange ein, bis sie an einem Kontrollbit erkennt, daß das Programm auf dem Controller keine Daten mehr sendet.
Eine detailierte Beschreibung der Befehle für den Datentransport über die parallele Schnittstelle ist im Anhang C.3.1 zu finden.

4 Eine *VME* - *IBM* Verbindung

4.1 Übersicht

Auf dem DESY-Gelände finden Rechnersysteme, welche auf dem *VME* -Bus basieren eine immer stärkere Verbreitung. Es besteht dabei oftmals der Wunsch von diesen lokalen Rechnern aus, über eine Datenverbindung zum DESY-Rechenzentrum mit seiner IBM-Anlage zu verfügen. Das DESY-Rechenzentrum (IBM 3084Q) ist für die vielen anderen Kleinrechner insbesondere wegen seiner großen Speicherkapazitäten auf Massenspeichern von Interesse. So können diese Kleinrechner die IBM-Anlage als gemeinsamen Datenpool benutzen.
Es existiert zwar auch ein DESY-Internes Netzwerk (DESYNET), was eine einfache Vernetzung von Rechnern gestattet, aber es handelt sich hierbei um ein Terminalnetzwerk. Diese Netzwerke sind dafür gedacht, daß man sich von einem Terminal aus zu verschiedenen Rechnern durchschalten kann. Solche Netzwerke sind nicht für einen schnellen Datentransfer konzipiert und aus diesem Grunde für den Transport großer Datenmengen ungeeignet. Für diese Problemstellung stellt das von der DESY-Gruppe F58 geschaffene PADAC-Netz eine Alternative dar. Die Hardware dieses Systems ist relativ preiswert und es existieren für viele Bussysteme (z.B. Nord-Bus, Q-Bus) Interface. K. Rehlich von der DESY-Gruppe F52 hat nun ein *VME* -PADAC Interface geschaffen, so daß auch Rechner mit diesem Bussystem über das PADAC-Netz miteinander verbunden werden können. Die Logik für dieses Interface befindet sich auf einer Doppeleuropakarte, welche in den *VME* -Bus gesteckt wird (siehe auch Anhang B.7).
Dieser Interfacekarte habe ich mich auch bedient, als das Problem auftauchte, das *BOP* -System mit dem DESY-Rechenzentrum zu verbinden, um große Datenmengen zwischen diesen beiden Rechnern zu transportieren. Schließlich wollte ich zur Auswertung der auf dem *BOP* -System berechneten Ergebnisdaten auch das Grafiksystem der *IBM* benutzen. Neben der Hardware ist für solch eine Datenverbindung aber auch die Software von entscheidener Bedeutung.
Denn ohne die Software hätte man in diesem Fall auf der *VME* -Bus Seite nur vier 16 Bit Register. Die Software muß das Interface initialisieren, die notwendigen Adressen übergeben und die zu transportierenden Daten in ein für das Interface verständliches Format umwandeln. Ferner müssen für eine solche Datenübertragung bestimmte Datenworte in die Steuerregister geschrieben und in diesen Registern stehende Bits ausgelesen werden. Alle diese Aufgaben erledigt die von mir in der Sprache *C* geschriebene Software so, daß dem Benutzer einfache Filefunktionen zur Verfügung stehen.
Außerdem habe ich alle von mir benutzten Konstanten in externe Files ausgelagert, so daß man nur die Werte in diesen Files ändern muß, falls man die Software auf einem anderen System installieren will. Diese gesamte von mir geschriebene Software für die Verbindung wird nächsten Abschnitt näher beschrieben.

4.2 Die Software auf der VME-Seite des Links

4.2.1 Übersicht

Die von mir geschriebenen Programme für die VME-PADAC Schnittstelle lassen sich in vier Kategorien unterteilen. Dieses sind die **include-Files**, die **Funktionsbibliothek**, die **Anwendungsprogramme für das BOP-System** und die **Beispielprogramme**.

4.2.2 include-Files

Wie in dem Abschnitt 2.1.2 erläutert, lassen sich mit der Hilfe von include-Files Tabellen und Listen in das C - Programm einfügen. So befinden sich auch in den vor mir erstellten include-Files fast alle in den Programmen verwendeten Konstanten und Texte.
Um die Übersichtlichkeit zu erhöhen, habe ich insgesamt fünf verschiedene include-Files erzeugt. Das erste dieser Files, **padac1.h** aus der Bibliothek */usr/bop/ibm* verweist nur auf die anderen vier include-Files. D.h., wenn man dieses File in ein bestehendes Programm integriert, so werden automatisch die anderen vier include-Files in das Programm mit eingebunden. Alle anderen include-Files befinden sich in der Bibliothek */usr/bop/include*.

constants.h In dieser Datei stehen alle Konstanten und Texte, welche bei der Installation der VME-PADAC Verbindung auf einem neuen Kleinrechner System nicht verändert werden müssen. Dazu gehören Zahlen, welche einzelnen Bits im EXPU-Modul zugeordnet sind sowie Konstanten, welche die von mir geschriebene Software zur Kennzeichnung bestimter Zustände verwendet. Schließlich stehen in diesem File auch noch die Befehle und die Antworten des auf der IBM laufenden Programmes PCSLINK2 und die von diesem Programm jedem gesendeten Datenblock vorausgestellten zwei 16 Bit Codewörter.

install.h Dieses File enthält die Parameter, welche in der Regel verändert werden müssen, wenn das Programmsystem in einer anderen Umgebung implementiert werden soll. Zum einen sind das alle von dem Interface verwendeten Adressen, d.h. sowohl die Adressen der EXPU-Register, als auch der Bereich auf welchen dieses Interface einen DMA-Zugriff macht. Zusätzlich sind in dieses File auch noch zwei Konstanten zur Fehlersuche sogenannte Debugginghilfen eingebaut, nämlich die Konstanten *TRACEMODUS* und *DESTINATION*. Während die erste Konstante direkt zur Fehlersuche (siehe auch Abschnitt 4.2.3) in den von mir geschriebenen Funktionen verwendet wird, kann man mit der Konstante *DESTINATION* festlegen, ob der Datentransfer mit der IBM oder einem MICRO-Modul des PADAC Systems stattfinden soll. Auf dem MICRO-Modul befindet sich das Spiegelprogramm, welches die empfangenen Daten wieder zurücksendet und somit ebenfalls bei der Fehlersuche hilfreich ist.

def.h In diesem File stehen vom Benutzer gewählte Größen bzw. Kombinationen von Werten aus den beiden vorangegangenen include-Files. Also z.B. die Definition des Write- und des Read-Requests aus der Abbildung 4 sowie die maximalen Größen der in den Funktionen verwendeten Felder.

struc.h In diesem File werden die Variabeltypen definiert, welche die Struktur der EXPU-Register sowie der Datenformate (siehe Abbildung 45) enthalten. Durch diese Definitionen ist selbst mit den etwas komplizierteren Datenformaten eine übersichtlichere Kommunikation möglich.

error.h Diese Datei enthält Informationen, welche zur im Abschnitt 4.2.3 beschriebenen Fehlerbehandlung dienen. Dazu werden in diesem File Variabeltypen definiert, welche diese Fehlerbehandlung unterstützen. Außerdem steht in dieser Datei eine Liste aller bisher eingetragenen Fehlermeldungen und eine Liste aller vorkommender Funktionen. In der Liste der

bisher eingetragenen Fehlermeldungen sind Verweise auf die Liste der Funktion enthalten, wo der Fehler auftritt.

4.2.3 Die Funktionsbibliothek

Die von mir geschriebene Sammlung von Funktionen für den Datenaustausch zwischen einem Kleinrechner und der IBM über das PADAC-System heißt *func.c* und steht in der Bibliothek */usr/bop/ibm*. Die Funktionssammlung ist in der Programmiersprache *C* geschrieben und benutzt ein Programm auf der IBM, welches für den Datentransfer mit VAX- Rechnern geschrieben wurde. Ich habe mich zwar bemüht die Funktionen möglichst übersichtlich zu schreiben und ihre Anwendung in Demonstrationsprogrammen zu erläutern, aber die wichtigsten Funktionen für den Anwender beschreibe ich im folgenden gesondert. Einige der in der Funktionssammlung befindlichen Funktionen werden von mir nicht benutzt, aber es sind durchaus Anwendungen denkbar, wo ihr Einsatz sinnvoll ist.

Fehlerbehandlung im *IBM - VME* Programm In allen Funktionen aus func.c wird eine vorzeichenlose Integervariable mit dem Namen *error_nr* (C-Typ: *unsigned int*) deklariert und an die übergeordnete Programmebene übergeben. Am Anfang jeder dieser Routinen wird der Wert dieser Variablen abgefragt und wenn er nicht gleich Null ist, wird die Routine sofort wieder verlassen. Dieses dient für den folgenden Mechanismus:
Sobald in irgendeinem Unterprogramm ein Fehler auftritt, wird der Variablen *error_nr* ein für diesen Fehler und dieses Unterprogramm spezifischer Wert zugeordnet, welcher von Null verschieden ist. Alle weiteren Routinen werden von da an nicht mehr durchlaufen, sondern bereits kurz nach ihrem Start abgebrochen. Am Schluß des Hauptprogrammes steht ein weiteres Unterprogramm mit dem Namen *error_handler*. An diese Routine wird nur die Variablen *error_nr* übergeben und im Gegensatz zu den anderen Routinen wird sie nur dann durchlaufen, wenn diese Variable einen Wert ungleich Null hat, also irgendwo ein Fehler aufgetreten ist.
Wenn die Routine *error_handler* durchlaufen wird, stellt sie fest um welchen Fehler es sich handelt und gibt sowohl die Art des Fehlers, als auch den Ort seines Auftretens aus. Anschließend sorgt die Routine *error_handler* für einen geordneten Ausstieg aus dem Programm.
Diese Art der Fehlerbehandlung besitzt eine ganze Reihe von Vorteilen:
Weil die Fehlerbehandlung überwiegend zentral in einer einzelnen Routine geschieht, ist der Teil der Fehlerbehandlung, welcher in den einzelnen Unterprogrammen erfolgt sehr einfach. Außerdem werden alle Fehler vom Grundprinzip her einheitlich behandelt, was beides zu einer übersichtlicheren Programmstruktur beiträgt. Auch wenn weitere Fehlermöglichkeiten erkannt werden, ist es sehr einfach diese in das bisherige Konzept einzubinden, selbst eine Erweiterung auf eine komfortablere Fehlerbehandlung ist einfach möglich. Schließlich stehen alle bisher erkannten Fehlersituationen geordnet in einer Liste, so daß sich ein neuer Anwender anhand dieser auf eventuelle Schwierigkeiten des Programmes vorbereiten kann.

Die Debugging Möglichkeiten in diesem Programm Einem C-Compiler ist üblicherweise ein Makro-Preprozessor vorangestellt, welcher in der Lage ist praktisch beliebige Zeichenketten durch andere Zeichenketten auszutauschen. Hierdurch besteht die Möglichkeit Konstanten einprägsame Namen zu geben.

Diese Eigenschaft wurde auch für die Konstante *TRACEMODUS* verwendet. In den ersten Zeilen des Programmes werden ihr die Werte *ON* oder *OFF* zugewiesen. Ansonsten befinden sich in den Programmen kurze Abschnitte, welche nur durchlaufen werden, wenn *TRACEMODUS ON (. 1)* gilt. Diese Abschnitte enthalten Anweisungen, welche bei der Fehlersuche helfen, also z.B. die Ausgabe von Variablenwerten mit der Stelle ihres Auftretens. Hierdurch ist es jederzeit möglich, durch das Verändern einer einzigen Programmzeile und dem anschließenden Neuübersetzen das Programm in einen Testmodus zu bringen.

ibm init

```
ibm init(error nr)
unsigned int *error nr ;
```

Diese Funktion muß bei der Aufnahme des Datenverkehrs als erstes aufgerufen werden. Im einzelnen tut sie folgendes. Zuerst werden notwendige Variablen initialisiert und die Datenstrukturen für den DMA-Transfer im Speicher angelegt. Falls der Datentransfer mit dem MICRO-Modul stattfinden soll, so wird ein entsprechender Teststring losgeschickt. Nach dem Empfang dieser Zeichenkette wird das Programm dann abgebrochen. Wenn der Datentransfer aber mit der IBM stattfinden soll, so wird die Datenverbindung zur IBM in drei Schritten aufgebaut.
Zuerst wird die Datenverbindung in einen definierten Anfangszustand gebracht. Dann wird das Online-Modul (hier TAL00) gestartet und schließlich das entsprechende Service-Programm in diesem Fall PCSLINK2 geladen und anschließend wird diese Routine verlassen.

error_nr Diese Variable ist für das Errorhandling zuständig (siehe auch 4.2.3)

ibm open

```
ibm open(filename,format,modus,i_column,error_nr)
char filename[LETTER]
unsigned int format ;
unsigned int modus ;
int i_column ;
unsigned int *error nr ;
```

Diese Funktion nutzt bereits Eigenschaften des Programmes PCSLINK2 aus, d.h. es kommuniziert ausschließlich mit diesem Programm. Über die Parameter dieser Funktion wird nicht nur festgelegt mit welchem File der Datentransfer auf der IBM stattfinden soll, sondern auch auf welche Art kommuniziert werden soll (Lesen, Schreiben, Druckerausgabe etc.).

filename[LETTER] Mittels dieser Variable wird der vollständige Name einer Datei angegeben (z.B. FXXAAA.LIB(TEST)). Dabei ist darauf zu achten, daß die Datei ein Mitglied einer Library sein muß. Weil in diesem Modus kein Benutzerschutz existiert, sollte man bei Schreibbefehlen sehr vorsichtig sein.

format Dieser Variable können die folgenden in den include-Files definierten Symbole ASCII oder LIBRARY zugeordnet werden. Nur wenn man das Inhaltsverzeichnis einer IBM-Bibliothek einlesen will, muß man hier das Symbol LIBRARY übergeben werden.

modus Durch diese Variable wird festgelegt, ob es sich bei dem Datentransfer um einen Lese- oder Schreibvorgang oder eine Ausgabe auf einen Drucker handelt.

- **READ** Ein File soll von der IBM zum Kleinrechner gelesen werden.
- **WRITE** Ein File soll von dem Kleinrechner auf die IBM geschrieben werden.
- **PRINTINT** Ein File soll von dem Kleinrechner auf dem internen Drucker der IBM ausgegeben werden.
- **PRINTEXT** Ein File soll von dem Kleinrechner auf dem externen Drucker der IBM ausgegeben werden.
- **PRINTL1** Ein File soll von dem Kleinrechner auf dem Laserdrucker L1 ausgegeben werden.
- **PRINTL2** Ein File soll von dem Kleinrechner auf dem Laserdrucker L2 ausgegeben werden.

i_column Hier wird die Anzahl der Spalten eingetragen, mit welcher der Datentransfer stattfinden soll.

error_nr Diese Variable ist für das Errorhandling zuständig (siehe auch 4.2.3)

ibm writeln

```
ibm writeln(text,modus,error_nr)
char text[] ;
char modus ;
unsigned int *error_nr ;
```

Mit diesem Befehl wird die Textzeile, welche sich in der Variablen *text* befindet zur IBM geschrieben. Genauer gesagt wird folgendes gemacht. Die Textzeile wird in dem erforderlichen Format in den Speicherbereich ablegt auf den das PADAC-Modul einen DMA-Zugriff machen kann. Anschließend wird überprüft, ob der in diesem Bereich noch verfügbare Platz für weitere Zeilen ausreicht. Wenn nicht, so wird das PADAC-Modul veranlaßt einen DMA- Zugriff auf diesen Bereich durchzuführen und die dort stehenden Daten zur IBM zu transportieren. Falls dieses geschieht, so wird nach dem DMA-Zugriff in dem Speicher wieder eine leere Datenstruktur angelegt. In diesen jetzt leeren Bereich können bei weiteren Ausführungen dieses Befehles dann weitere Zeilen eines Textes gelangen.

text[ ] In dieses Variabelfeld wird eine Zeile des zur IBM zu sendenen ASCII-Textes geschrieben. Etwaige notwendige Umwandlungen nimmt dabei die Routine selbständig vor. Sollte die Textzeile die maximal erlaubte Länge, definiert als Konstante LETTER im File *def.h*, überschreiten, so wird über die Variable *error nr* eine Fehlerbehandlung eingeleitet.

modus Der Text, welcher zur IBM gesendet wird, kann entweder in eine Datei geschrieben, oder auf einem Drucker ausgegeben werden. Entsprechend muß der Variablen **modus** entweder das Codewort *DATEI* oder *PRINT* zugewiesen werden.
Der Unterschied ist, daß bei der Druckerausgabe dem Text ein Leerzeichen vorangestellt werden muß, damit eine neue Textzeile keinen Seitenvorschub bewirkt.

error_nr Diese Variable ist für das Errorhandling zuständig (siehe auch 4.2.3)

ibm_readln

```
ibm readln(text_line,error_nr)
char text_line[LETTER] ;
unsigned int *error_nr ;
```

Wie der Name dieser Funktion bereits vermuten läßt, bewirkt sie genau das Gegenteil der Funktion *ibm_writeln*. Die Funktion guckt nach, ob in dem Bereich, in welchem das PADAC-Modul per DMA seine Daten ablegt, noch eine Textzeile steht, welche noch nicht ausgegeben wurde. Falls dieses der Fall ist, so wird dieser Text in die Variable *text_line* kopiert. Wenn hingegen alle Textzeilen, welche sich noch im Speicherbereich befinden bereits ausgelesen wurden, so schickt die Funktion eine Aufforderung zur *IBM* , mehr Text zu senden. Hierauf kann die *IBM* auf zwei verschiedene Arten antworten. Entweder sendet sie einen weiteren Textblock oder das Ende des zu übertragenden Files ist erreicht. Im ersten Fall liefert die Funktion die erste Zeile des neuen Textblockes, was entsprechend die nächste Zeile vom gesendeten Text ist. Im anderen Fall hingegen wird dem Funktionsnamen *read_ln* der Wert für das Ende eines Files (EOF) zugeordnet. Hier wird die Eigenschaft von *C* ausgenutzt, daß in dieser Sprache die Funktionsnamen selber Variablen sind, mit denen Werte aus der Funktion übergeben werden können. Dadurch, daß der Anwender den Wert der Funktion bei jedem Lesen abfragt, kann er also feststellen, wann der von der *IBM* gesendete Text zu Ende ist.

text_line[LETTER] In dieses Variabelfeld wird die von der *IBM* gesendete Textzeile getan.

error_nr Diese Variable ist für das Errorhandling zuständig (siehe auch 4.2.3)

ibm_close

```
ibm close(error nr)
unsigned int *error_nr ;
```

Die Funktion *ibm_close* schließt das durch die Funktion *ibm_open* geöffnete File auf der *IBM* . Die Funktion ist so geschrieben, daß sie immer aufgerufen werden kann, also z.B. auch wenn noch nicht der gesamte Text von der *IBM* zum Kleinrechner gesendet worden ist.

error nr Diese Variable ist für das Errorhandling zuständig (siehe auch 4.2.3)

ibm_exit

```
ibm exit(error nr)
unsigned int *error nr ;
```

Durch die Funktion *ibm_exit* wird der Datenverkehr mit dem auf der *IBM* laufenden Service-Programm beendet und die Datenverbindung zur *IBM* abgebrochen.

error_nr Diese Variable ist für das Errorhandling zuständig (siehe auch 4.2.3)

4.2.4 Die Transportsoftware

Übersicht Die Transportsoftware läßt sich grob in zwei verschiedene Gruppen einteilen. Da sind zum einen die Programme, welche speziell für das *BOP* -Rechnersystem geschrieben sind. Diese Programme regeln, sofern sie auf dem Controller laufen, den Datentransfer zwischen der *IBM* und der *PCS* . Mit diesen Programmen wurden die Daten für das *RACETRACK*- Programm transportiert und die Listings der Programme erzeugt.
Die anderen sind Demonstrationsprogramme, welche nur den Datenaustausch zwischen dem Controller und der *IBM* praktizieren. Als Speicher für den DMA-Zugriff wird hierbei, wie in den anderen Fällen auch, die Mailbox benutzt. Diese Funktion hätte aber auch jeder andere *VME* -Speicher mit einer Mindestgröße von 24 kByte erfüllen können. Die Programme veranschaulichen sowohl die einfache Kommunikation mit dem EXPU-Interface, als auch die Anwendung des Funktionspaketes *func.c*.

Anwendungsprogramme Diese Programme bringen entweder ASCII-Texte von der *PCS* auf die Drucker der *IBM* wie *pibm_ex.con* und *pibm_l2.con* oder Files zur *IBM* wie *af_to_ibm.con*. Andere Programme wie *af_to_pcs.con* transportieren Files von der *IBM* zur *PCS* . Alle diese Programme benutzen die Funktionen aus der Bibliothek *func.c*.

Demonstrationsprogramme Diese Programme sollen helfen, wenn man sich mit dem Prinzip der Datenübertragung über das PADAC-Netz vertraut machen will. Außerdem dienen sie dazu, um eine neu installierte VME-IBM Verbindung zu testen. Die Kurzbeschreibungen dieser Programme befinden sich im Anhang D.1.

4.3 Die *IBM* Seite des Links

Dieser Teil der *VME - IBM* Verbindung existierte bereits, bevor ich mit meiner Arbeit begonnen hatte. Ich benutze auf der IBM-Seite das in der Computersprache FORTRAN 77 geschriebene Programm ***VAXLINK1***. Wie es der Name bereits andeutet, wurde es ursprünglich für die Kommunikation zwischen der DESY-IBM und VAX-Rechnern geschrieben. Für diese Datenübertragung wird es auch heute noch überwiegend benutzt. Weil die VAX-Rechner bei einem 16-Bit Wort die oberen 8 mit den unteren 8 Bit vertauschen, man spricht auch von dem Vertauschen von High- und Low-Byte, mußte ich in meinen Programmen die Variable *swap_modus* einführen. Die Variable ist global definiert und taucht deshalb in den Funktionsdeklarationen nicht auf.
Das Programm *VAXLINK1* ist modular aufgebaut und sehr übersichtlich geschrieben, was durch die Verwendung der im EXP-Manual [16] beschriebenen Funktionen erleichtert wurde. Daneben werden Änderungen in diesem Programm auch dadurch erleichtert, daß in der gleichen Bibliothek Kommandoprozeduren stehen, welche ein einfaches neues Erzeugen des lauffähigen Programmes ermöglichen. Dieses habe ich ausgenutzt, als ich in das Programm die Möglichkeit der Ausgabe auf die im DESY befindlichen Laserdrucker eingebaut habe, mit welchen von allen in dieser Arbeit genannten Programmen Listings [1] erzeugt wurden. Die von mir modifizierte Programmversion habe ich dabei *PCSLINK2* genannt. Weil das Programm von vielen Rechnern benutzt wird, wird es auch ständig gepflegt, wodurch sich etwaige Änderungen ergeben können. Deshalb ist es sinnvoll sich vor der Verwendung des

[1] Die Listings können bei Herrn Prof. E. Lohrmann eingesehen werden.

Programmes über seinen aktuellen Stand bei der zuständigen Stelle zu erkundigen.
Im Anhang D.2 ist eine Liste der Befehle, des auf der *IBM* für diesen Datentransfer laufenden Programmes mit Beispielen zu finden. In der Tabelle 6 steht eine kurze Übersicht der Befehle, über welche das Programm *PCSLINK2* verfügt. Diese Liste wird dabei teilweise durch die Eingabe des Befehles HELP ausgegeben.

5 Ergebnisse der Bahnberechnungen

5.1 Überblick

Dieses Kapitel beschäftigt sich mit einer Anwendung aller in den vorherigen Kapitel vorgestellten Hilfsmittel, nämlich mit der rechnergestützten Simulation von Teilchenbahnen im HERA Protonenspeicherring. Solche Berechnungen sind für die Entwicklung eines solchen Speicherringes von entscheidener Bedeutung. So untersucht man u.a., wie groß die Ausdehnung des Strahls senkrecht zur Flugrichtung der Teilchen sein kann, innerhalb derer die Bahnen noch stabil sind (Apertur).
Eines dieser Simulationsprogramme, welches am DESY entwickelt worden ist, ist das Programm RACETRACK [8] von A.Wrulich. In den nächsten Abschnitten erläutere ich zuerst, auf welchen Grundlagen das Programm RACETRACK basiert und wie es in etwa aufgebaut ist. Anschließend zeige ich die graphischen Ergebnisse der durchgeführten Berechnungen.

5.2 Grundlagen des Simulationsprogrammes

Bei RACETRACK handelt es sich um ein Simulationsprogramm, welches die Bewegung der Teilchen in einem Speicherring berechnet. Das Programm basiert auf den folgenden Überlegungen:
Wenn man Magnetfelder, aber keine Beschleunigungsstrecken oder Energieverluste berücksichtigt, so wird die Bewegung eines Teilchens durch die folgenden Differentialgleichungen beschrieben :

$$\frac{d^2x}{ds^2} + K_x(s) * x = \frac{e}{p_0} B_z(x,s,z) \tag{3}$$

$$\frac{d^2z}{ds^2} + K_z(s) * z = -\frac{e}{p_0} B_x(x,s,z) \tag{4}$$

mit

x, s, z horizontale, vertikale und longitudinale Koordinaten in dem rechtshändigen (x, s, z) Koordinatensystem

K_x, K_z Fokussierungsstärken senkrecht zur Strahlachse (durch die Quadrupole)

e Ladung des Teilchens

B_x, B_z Komponenten des Magnetfeldes senkrecht zur Strahlachse (alle restlichen Magnetfelder)

p_0 Impuls des Teilchens

Weil für diese Bewegungsgleichungen im allgemeinen keine analytische Lösung existiert, muß man die Lösung mittels numerischer Methoden finden. Zu diesem Zweck werden die Gleichungen stückweise zu den jeweiligen Anfangsbedingungen des Teilchens gelöst.
Bei den linearen Elementen (Quadrupolen) ist es möglich, über den homogenen Teil der Gleichungen 3 und 4 zu einer einfachen Transformation für die Bahnkoordinaten zu kommen. Während man bei diesen Transformationen einen Matrizenformalismus verwenden kann, muß man bei den nichtlinearen Elementen ein anderes Verfahren anwenden. Bei diesen wird die

sogenannte "dünne Magneten"-Näherung verwendet. Dieses bedeutet, daß für die höheren Multipole der gesamte integrierte Effekt auf einen Magneten der Länge Null reduziert wird:

$$\frac{d^2x}{ds^2} = \frac{e}{p_0}\delta(S_M)\int_{\frac{-l}{2}}^{\frac{l}{2}} B_z(x,s,z)ds$$

$$\frac{d^2z}{ds^2} = -\frac{e}{p_0}\delta(S_M)\int_{\frac{-l}{2}}^{\frac{l}{2}} B_x(x,s,z)ds$$

wobei

l für die Länge des Magneten

S_M für den Ort, an welchem er berücksichtigt wird und

δ für das Dirac'sche Deltafunktional steht.

Wenn man zusätzlich für den integralen Mittelwert des Magnetfeldes

$$\overline{B} = \frac{1}{l}\int_{\frac{-l}{2}}^{\frac{l}{2}} B ds$$

schreibt, so sieht man, daß in dieser Näherung nur die Richtungen $x' = \frac{dx}{ds}$ und $z' = \frac{dz}{ds}$ an der Stelle S_M geändert werden müssen. Diese Änderungen kann man dann in der folgenden Form schreiben :

$$\triangle(x') = \overline{B_z}\ l\ \frac{e}{p_0} \tag{5}$$

$$\triangle(z') = -\overline{B_x}\ l\ \frac{e}{p_0} \tag{6}$$

Nun benutzt man die Multipolentwicklung des Magnetfeldes, welche man zweckmäßigerweise in komplexer Form schreibt:

$$\overline{B_z} + i\overline{B_x} = B_0\sum_n (b_n + ia_n)(x+iz)^{n-1} \tag{7}$$

wobei

b_n, a_n die Multipolkoeffizienten (normal und gedreht)

$n = 1$ die Dipole und

$n = 2$ die Quradupole usw. sind.

Durch die Formeln 5 bis 7 ergeben sich nun die folgenden beiden Beziehungen:

$$\triangle(x') = \frac{e}{p_0}\ l\ B_0\ \Re\{\sum_n (b_n + ia_n)(x+iz)^{n-1}\}$$

$$\triangle(z') = -\frac{e}{p_0}\ l\ B_0\ \Im\{\sum_n (b_n + ia_n)(x+iz)^{n-1}\}$$

Über die man mittels des Zusammenhanges

$$\frac{e}{p_0}B_0 = \frac{1}{\rho}$$

direkt die Formeln für die Richtungsänderung den sogenannten "Kick" eines Teilchens aufgrund der Magnetfeldfehler erhält:

$$\triangle(x') = \frac{l}{\rho}\Re\{\sum_n (b_n + ia_n)(x+iz)^{n-1}\}$$

$$\triangle(z') = \frac{-l}{\rho}\Im\{\sum_n (b_n + ia_n)(x+iz)^{n-1}\}$$

Dabei steht das ρ für den Krümmungsradius der Bahn eines Teilchens (Ladung e), welches sich mit dem Impuls p_0 in einem Magnetfeld der Stärke B_0 bewegt. Wenn man die Dipole bis zur 9. Ordnung (n=9 für 18-Pol) in x- und z-Richtung berücksichtigen will, so muß man jeweils ein Polynom 8.Ordnung in x und z berechnen. Bei dem Programmcode von RACETRACK wurden diese per Hand entwickelt und explizit als Hornerschemata dargestellt, weil dieses die schnellste und genaueste Methode ist.
Wenn man also bei einem Magneten die Abweichungen seines realen Feldes von seinem idealen Magnetfeld berücksichtigen will, so fügt man zu seinem idealen Feld noch ein solches Multipolfeld hinzu. Bei der auf dem *BOP* installierten Fassung wurde dieses Verfahren allerdings nur für die Felder der supraleitenden Magneten in den Kreisbögen angewandt. Diese Magneten weisen nämlich um Größenordnungen stärkere Feldfehler auf als die normal leitenden Magneten in den geraden Strecken.

5.3 Programmablauf

Bei der auf der *IBM* installierten RACETRACK-Version läßt sich der Programmablauf in zwei Teile untergliedern.
Im ersten Teil werden vorbereitende Rechnungen durchgeführt. D.h. es werden alle Parameter für die Simulation der Bewegung eines Teilchens in dem Speicherring berechnet, zu welchen u.a. die Matrizen für die lineare Optik gehören. Weil dieser Teil nur von neuem durchlaufen werden muß, wenn sich an der Struktur der Magnete etwas ändert, wurde in der auf dem *BOP* installierten Version auf diesen Teil verzichtet. Hier werden alle diese vorbereitenden Rechnungen auf der *IBM* durchgeführt und die Parameter für die Simulation als ein Datenblock zum *BOP* gesendet.
Im zweiten Programmteil, welcher auch auf dem *BOP* existiert, wird dann die eigentliche Simulation durchgeführt, bei der die neuen Teilchenkoordinaten nach jedem Magnetelement explizit ausgerechnet werden. Dieser Teil des Programmes ist bereits bei mehr als 10 simulierten Umläufen der zeitaufwendigere Programmteil, weshalb das *BOP* -System schließlich auch für diese Anwendung entwickelt worden ist. Zu der auf dem *BOP* installierten Version des Programmes RACETRACK ist noch anzumerken, daß davon ausgegangen wird, daß alle Teilchen die Sollenergie besitzen und somit auch auch keine Synchrotronschwingungen durchführen.

5.4 Die Implementation des Simulationsprogrammes auf dem Multiprozessorsystem

Als eine Demonstration der Funktion des Systems und eine Anwendung wurde von mir eine in der Computersprache *C* geschriebene Version des Programms RACETRACK eingesetzt, welche fast alle Hilfsmittel des *BOP* -Systems benutzt. In diesem Abschnitt wird deshalb die

Software beschrieben, welche die Berechnungen unterstützt hat. Bis auf die Shellprozedur [2] *DATA_trans* handelt es sich dabei ausschließlich um *C* -Programme.
Bevor die eigentliche Simultionsberechnung stattfinden kann, muß die Parameterliste mit den Magnetdaten des Speicherringes aufbereitet werden. Die hier verwendete Parameterliste lag zuerst als ein Textfile auf der *IBM* vor. Dieses File wurde mit der Hilfe des PADAC-VME Interfaces zur PCS gebracht. Dadurch, daß die Daten als Textfile übertragen wurden, erübrigte sich eine Konvertierung zwischen den Zahlenformaten der IBM und der PCS.
Auf der PCS wurden die Zahlen dieses Textfiles dann in binärer Form in ein sequentielles File geschrieben. Mittels des Programmes *sud.c* schaltete man sich nun von der PCS aus an die Terminalschnittstelle des Controllers. Über den Befehl "push DATA" des auf dem Controller laufenden Programmes *oper.c* wurde dann der Block mit den in binärer Form vorliegenden Zahlen in das statische RAM der Floating Point Unit (FPU) (siehe Abbildung 50) geladen. Anschließend wurde das Programm *emmibop.fpu* in den Programmspeicher der Targets geladen. Dieses Programm ist das lauffähige "Tracking-Programm" von A.Wrulich, welches aus dem Quellprogramm *emmibop.c* compiliert und für die Targets aufbereitet wurde. Jetzt kann man auf dem Controller das Programm *emmicon.con* laden und auf der *PCS* das Programm *rdata* starten. Bei *emmicon.con* handelt sich um das compilierte und für den Controller aufbereitete File des Quellprogrammes *emmicon.c*. Sobald dieses Programm gestartet wird, startet es selbst nacheinander die Targetrechner und übergibt an diese die Startwerte. Die Startwerte sind die Anfangskoordinaten und Anfangsgeschwindigkeiten des Teilchen, sowie die Anzahl der durch den Speicherring zu "trackenden" Umläufe.
Ab jetzt senden die Targets nach jedem berechneten Umlauf die neuen Koordinaten und Geschwindigkeiten des Teilchens zum Controller. Dieser gibt diese Daten mit den Ergebnissen über die Grafik aus und sendet sie gleichzeitig mittels des Interfaces zur *PCS* -Workstation. Auf der *PCS* werden diese Ergebnisdaten dann von dem Programm *rdata* empfangen, welches die Daten in binärer Form sequentiell in ein File geschreibt und zwar für jedes Target getrennt. Dadurch gelangen die Ergebnisdaten, welche von einem Satz Startparameter abstammen auch in ein einzelnes File. Diese Aufgaben werden durch das auf der PCS laufende Programm *rdata* erledigt. Nachdem die Berechnungen erfolgreich beendet worden sind, kann man dann die Datenfiles entweder auf Streamerkassetten zwischenspeichern, oder gleich über die von mir aufgebaute Verbindung zur IBM senden. Auch dieser Datentransfer erfolgt über Textfiles, um Zahlenformatumwandlungen zu vermeiden. Weitere Details über diesen Ablauf sind in dem Anhang D.3 nachzulesen.

5.5 Auswertung

5.5.1 Übersicht

Auf den nachfolgenden Seiten sind die Grafiken zu sehen, welche mit den auf dem *BOP* -System erzeugten Daten erstellt wurden. Die Bilder zeigen die Orts- und Phasenräume von einem Proton am Wechselwirkungspunkt der Halle Ost. Während die Anfangsgeschwindigkeiten (x', z') vertikal zur Strahlachse immer gleich Null gesetzt sind (x'= 0 & z'=0), habe ich den Anfangsamplituden in x- und z-Richtung auch andere Werte zugeordnet. In der Tabelle 2 ist eine Übersicht aller dieser Bilder zu finden, aus der auch ersichtlich ist, wie groß die jeweiligen Startamplituden einer Berechnung gewesen sind. Die Startamplituden

[2] Shell ist die Kommandosprache von *UNIX*

sind übrigens in Vielfachen der Strahlbreite (σ_x, σ_z) angegeben, deren Werte aus der Tabelle 1 (Seite 2) ersichtlich sind. Ferner ist noch zu sagen, daß ich bei den Abbildungen, welche wegen Instabilitäten abgebrochen wurden, die Umlaufzahlen nur genähert angegeben habe.

5.5.2 Darstellung der Ergebnisse

In den ersten beiden Abbildungen 4 und 5 kann man sehr schön das beinahe lineare Verhalten der Maschine bei kleinen Amplituden sehen. Die Startkoordinaten sind dabei so gewählt, daß die Teilchen gerade bei der normalen Strahlbreite der Maschine liegen. So würde man bei einer vollständig linearen Maschine im Ortsraum ein mit den Koordinatenachsen paralleles Rechteck sehen. Durch den Einfluß der nichtlinearen Elemente (Sextupole und Multipole) ist es aber etwas gedreht und die Ränder sind nicht ganz gerade. Ähnlich verhält es sich mit dem Phasenräumen, bei welchen man bei einer linearen Maschine einfach eine Ellipse sehen würde. Hier ist der Rand der Ellipse jedoch zu einem Streifen mit einer regelmäßigen Struktur aufgeweitet (Abb. 5).
In den Abbildungen 6 bis 8 wurde das Teilchen am Anfang nur in der z-Richtung ausgelenkt, d.h. die x-Koordinate ist gleich null. Durch die Kopplung der Koordinaten über die nichtlinearen Elemente kommt aber auch eine Bewegung in die x-Richtung zustande. Anstatt einer Strecke sieht man im Ortsraum (Abb. 6) ein Gebilde, welches in der Richtung der z-Achse nur etwa um den Faktor 3 länger ist als in der Richtung der x-Achse (siehe Maßstab). Die Ortskoordinaten sind in diesem Fall also relativ stark miteinander gekoppelt. Bei dem x-Phasenraum (Abb. 7) hätte man bei einer linearen Maschine keinerlei Bewegung, also nur einen Punkt. In Fall von HERA findet man hier aber eine mit einer regelmäßigen Struktur vollständig ausgefüllte Ellipse vor, welche allerdings in den Halbachsen um den Faktor 3 kleiner ist, als die Ellipse des z-Phasenraumes (Abb. 8). Letztere sieht der Ellipse einer linearen Maschine sehr ähnlich und ihr Rand ist viel weniger stark aufgeweitet als der in der Abbildung 5. Mit dem 1.4-fachen der Strahlbreite bewegt man sich bei diesen Umlaufzahlen ganz offensichtlich noch in dem stabilen Bereich, worauf auch die homogene Verteilung der Punkte bei den bisher aufgetretenden Strukturen schliessen läßt.
Ganz ähnlich verhält es sich mit den Abbildungen 9 bis 11, nur das hier die Anfangsauslenkung in die z-Richtung stattfand. Folglich erhält man vom Prinzip her ähnliche Ergebnisse wie in den Abbildungen 6 bis 8. Nur mit dem Unterschied, daß hier die Verhältnisse von den x- und den z-Koordinaten ausgetauscht sind. Außerdem unterscheiden sich hier die Wertebereiche nicht um den Faktor 3, sondern etwa um den Faktor 20.
Bei den nächsten Abbildungen 12 bis 19 wurde die Startamplitude im Verhältnis zu den Abbildungen 9 bis 11 um mehr als eine Zehnerpotenz erhöht. Erwartungsgemäß ist die Bahn des Teilchens nicht mehr stabil. Bereits bei etwa 23000 Umläufen wächst die Amplitude des Teilchens weit über die normale Dimension des Teilchenstrahles hinaus. Der Ortsraum sieht bei kleinen Umlaufzahlen (Abb. 12) dem der Abbildung 9 sehr ähnlich. Aber mit zunehmenden Umlaufzahlen sieht man in der Figur eine neue regelmäßige Struktur entstehen (Abb. 13 und insbesondere 14) die an die Amplituden bei einer stehenden Welle erinnert. Bei dieser Struktur fällt außerdem auf, daß sie deutlich grober ist, als die Unterstrukturen bei den vorherigen Startwerten. Außerdem beginnt der Rand der Struktur mit zunehmender Umlaufzahl "auszufransen" und die Punkte liegen immer weiter von den Nullpunkten entfernt (Abb. 14 & 15). Ein hierzu äquivalentes Verhalten ist im x-Phasenraum (Abb. 16 - 19) zu sehen. Mit zunehmenden Umlaufzahlen wird der Rand der Ellipse immer breiter und

ihr Rand "franst" zunehmend aus. Dabei zeigen sich mit dem Anwachsen des Randes neue komplizierte Strukturen (siehe insbesondere Abb. 19).
Wenn man die Verhältnisse für die x- und die z-Koordinaten wiederum vertauscht, so fällt auf, daß die Teilchenbahnen bei den hier verwendeten Umlaufzahlen überraschend stabil sind. Zwar erinnern die Strukturen des Strukturen des Orts- (Abb. 20) und des x-Phasenraumes (Abb. 21) nur noch sehr entfernt an die einer linearen Maschine (Rechteck und Ellipse), aber es sind bei ihnen keine Anzeichen für Instabilitäten zu erkennen.
Diese Anzeichen treten jedoch zumindest andeutungsweise auf, wenn man die Startamplitude noch weiter erhöht, wie es bei den Abbildungen 22 und 23 gemacht worden ist. Diese beiden Bilder lassen sich zwar im Prinzip mit den vorherigen vergleichen, aber die inneren Strukturen wirken hier etwas weniger homogen. Außerdem ist der Rand hier nicht mehr so scharf begrenzt, so daß bei größeren Umlaufzahlen Instabilitäten zu erwarten sind. Dieses lassen auch die großen Startamplituden vermuten.
Solche Anzeichen für Instabilitäten sieht man aber auch, wenn man die beiden Startamplituden (x, z) verhältnismäßig groß macht. Der Ortsraum zeigt zwar bei kleinen Umlaufzahlen große Ähnlichkeiten mit einem Rechteck (Abb. 24 und 25), aber es fehlt im Innern eine regelmäßige Struktur und der Rand ist ziemlich unscharf. Dieser Trend setzt sich bei größeren Umlaufzahlen (Abb 26 und 27) noch fort und die durch die Figuren bedeckte Fäche vergrößert sich ständig. Auffällig ist, daß der Rand insgesamt zwar immer unschärfer wird, daß aber die Ecken der Figur mit zunehmender Umlaufzahl immer schärfer begrenzt werden. Der x-Phasenraum läßt ebenso geordnete Strukturen vermissen. So wird der strukturlose Rand der Ellipse nicht nur immer unschärfer, sondern er wandert auch immmer weiter nach innen, bis schließlich das ganze Innere der Ellipse ausgefüllt ist. Obwohl auch die Figur des z-Phasenraumes die Form einer Ellipse hat, zeigt sie doch ein ganz anderes Verhalten. Zuerst, d.h. bei kleineren Umlaufzahlen (Abb. 32 bis 34), läßt sich eine relativ geordnete Struktur im Rand erkennen. Zusätzlich wandert der Rand weiter nach außen, bis es dann bei dem Übergang von Abbildung 34 zu der Abbildung 35 einen Sprung in dieser Entwicklung gibt. Die weiter entfernt liegenden Teile der Figur werden nun völlig ungeordnet und lassen keine Ähnlichkeit mit dem Inneren des Randes erkennen.

Nr der Abb. :	x-Startamplitude	z-Startamplitude	Umlaufzahl
4	1	1	35000
5	1	1	35000
6	0	1.41	35000
7	0	1.41	35000
8	0	1.41	35000
9	1.41	0	35000
10	1.41	0	35000
11	1.41	0	35000
12	15.6	0	3000
13	15.6	0	6000
14	15.6	0	12000
15	15.6	0	$\approx$ 23000
16	15.6	0	3000
17	15.6	0	6000
18	15.6	0	12000
19	15.6	0	$\approx$ 23000
20	0	16.9	40000
21	0	16.9	40000
22	0	21.21	17000
23	0	21.21	17000
24	14	14	3000
25	14	14	6000
26	14	14	12000
27	14	14	$\approx$ 24000
28	14	14	3000
29	14	14	6000
30	14	14	12000
31	14	14	$\approx$ 24000
32	14	14	3000
33	14	14	6000
34	14	14	12000
35	14	14	$\approx$ 24000

Tabelle 2: Anfangswerte der Simulation

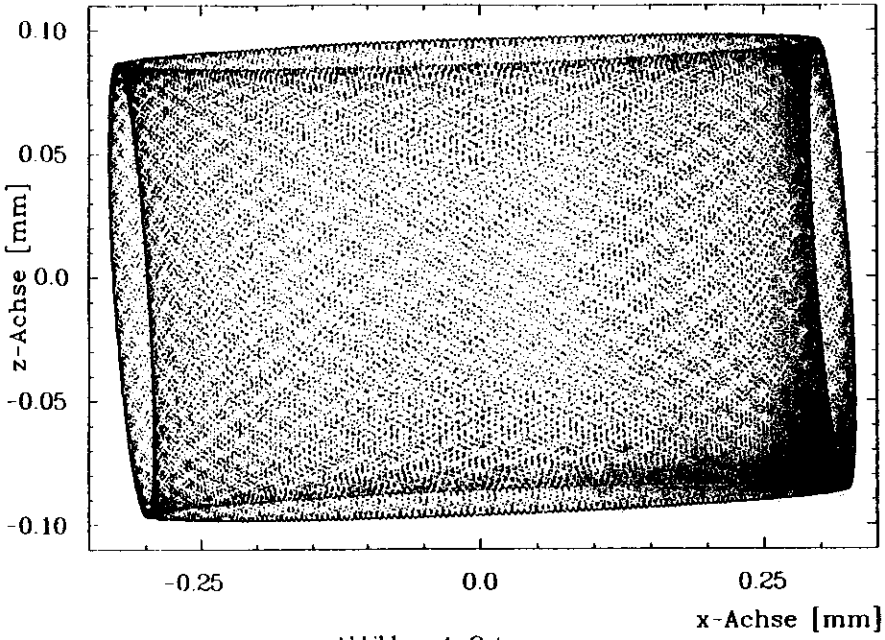


Abbildung 4: Ortsraum

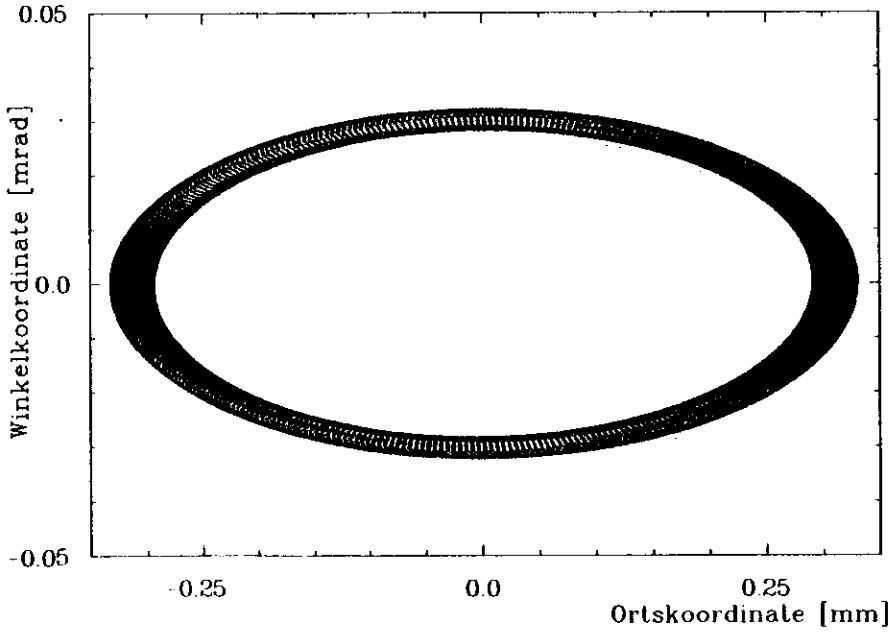


Abbildung 5: x-Phasenraum

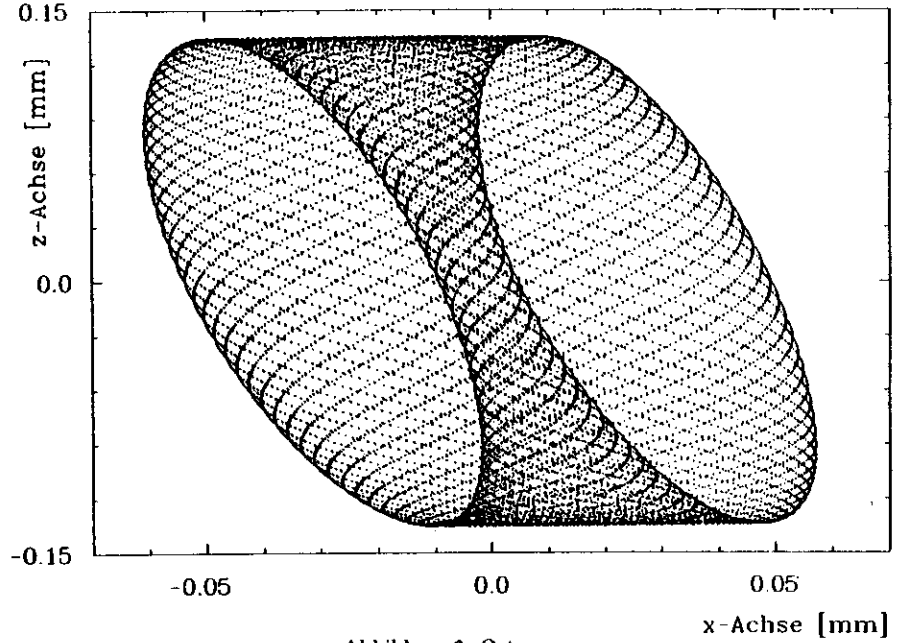


Abbildung 6: Ortsraum

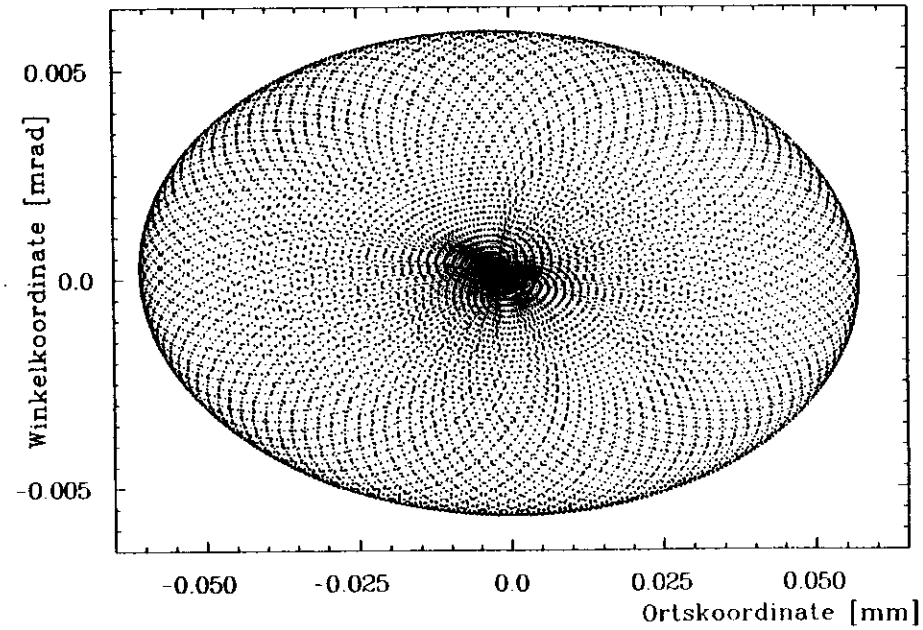


Abbildung 7: x-Phasenraum

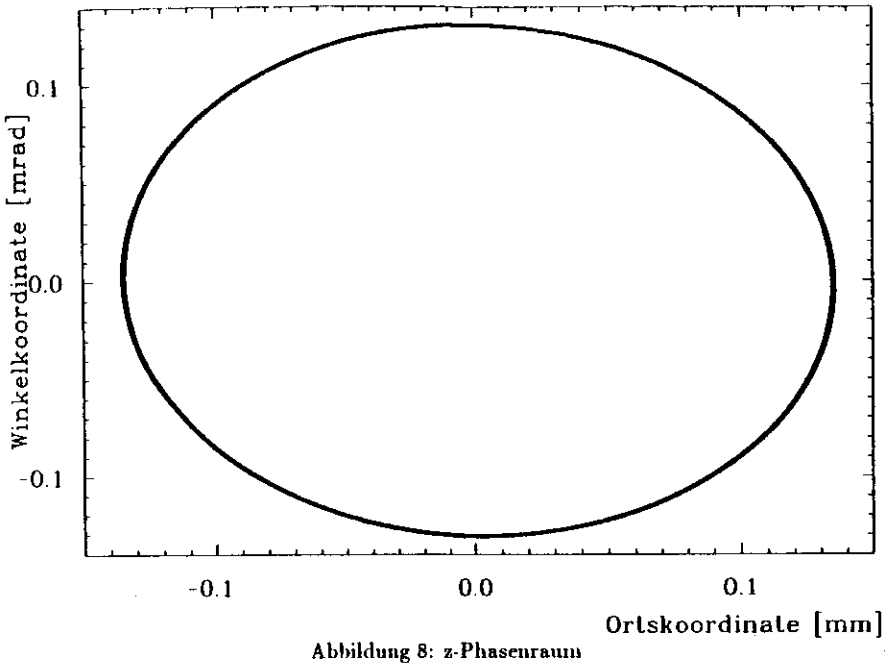


Abbildung 8: z-Phasenraum

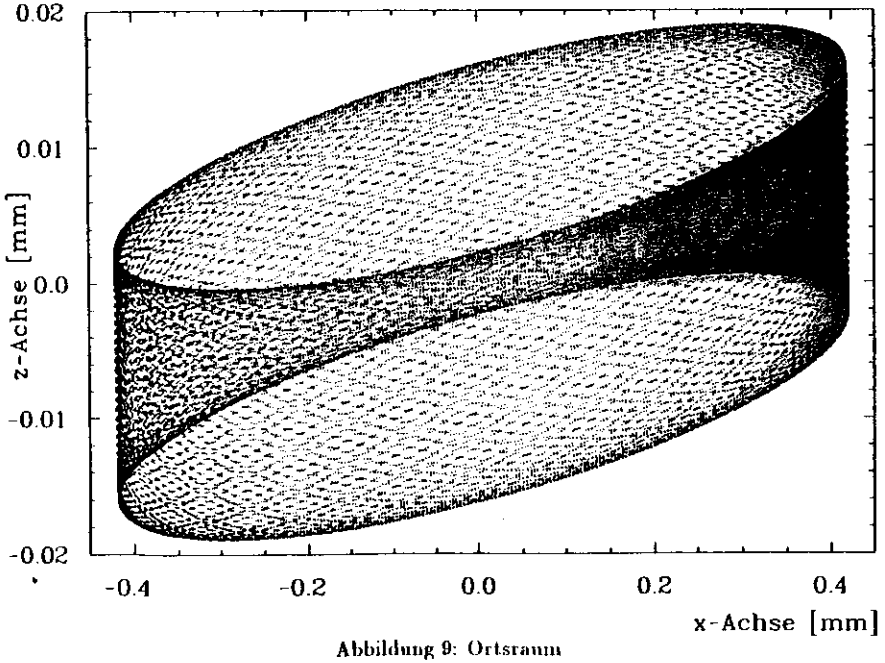


Abbildung 9: Ortsraum

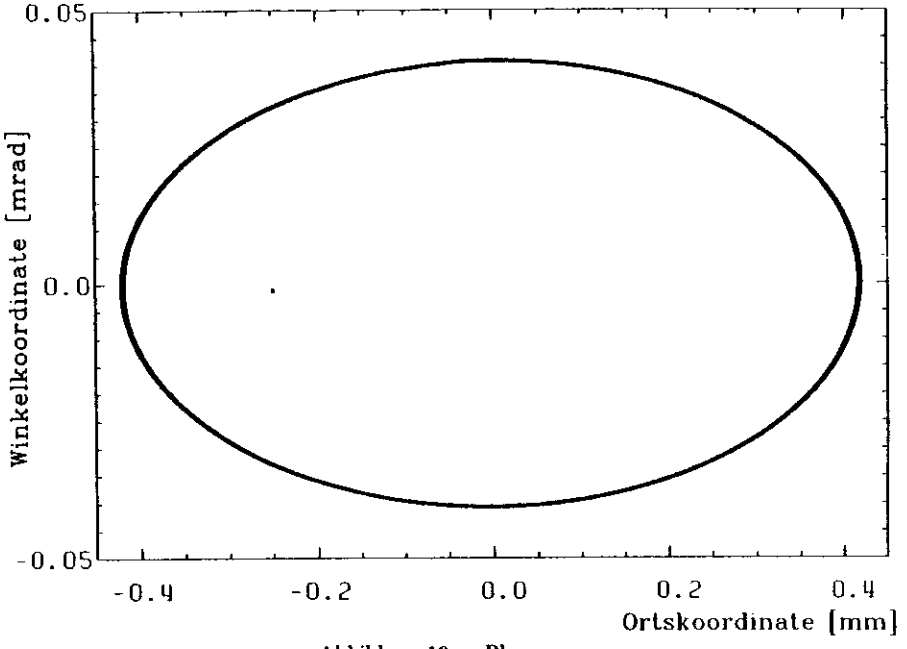


Abbildung 10: x-Phasenraum

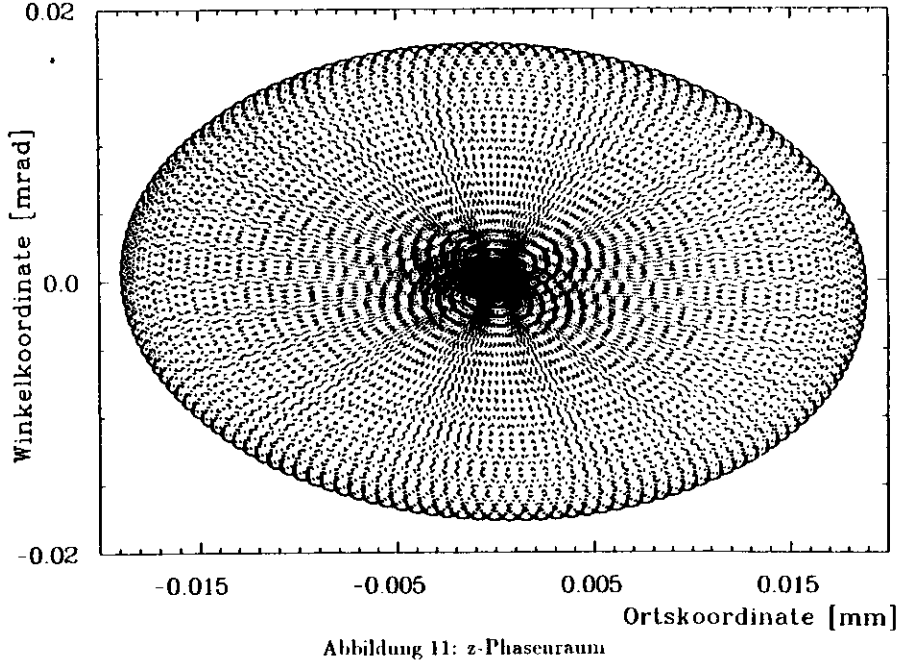


Abbildung 11: z-Phasenraum

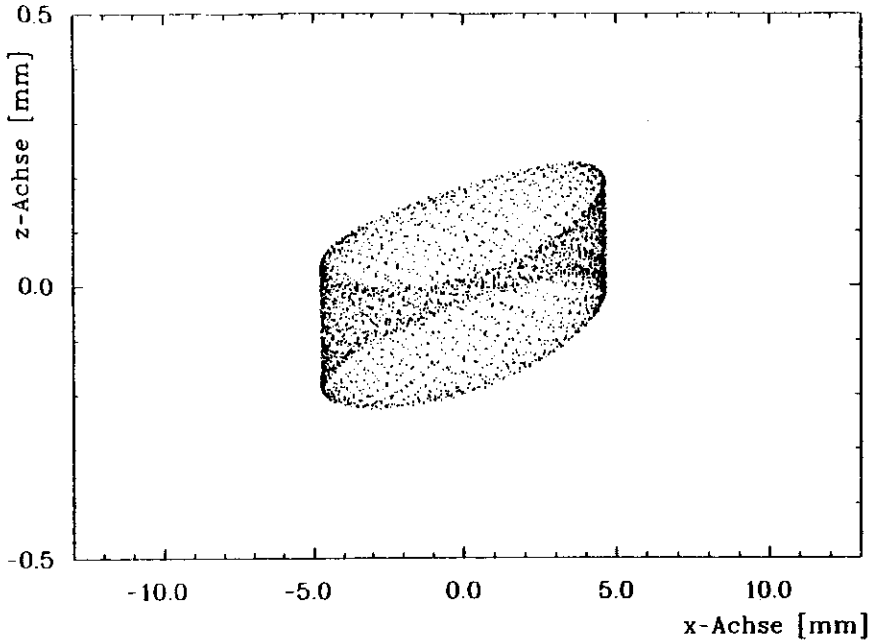


Abbildung 12: Ortsraum

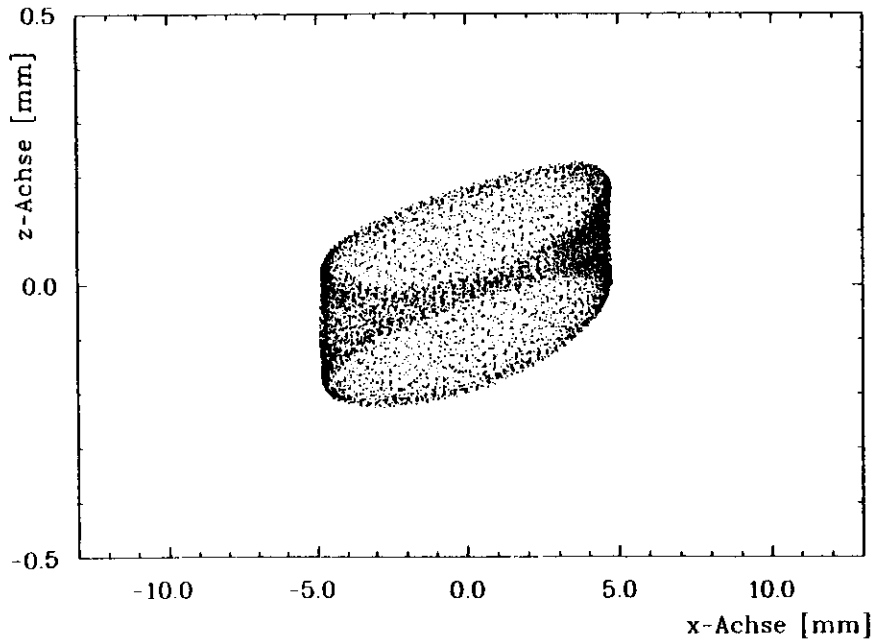


Abbildung 13: Ortsraum

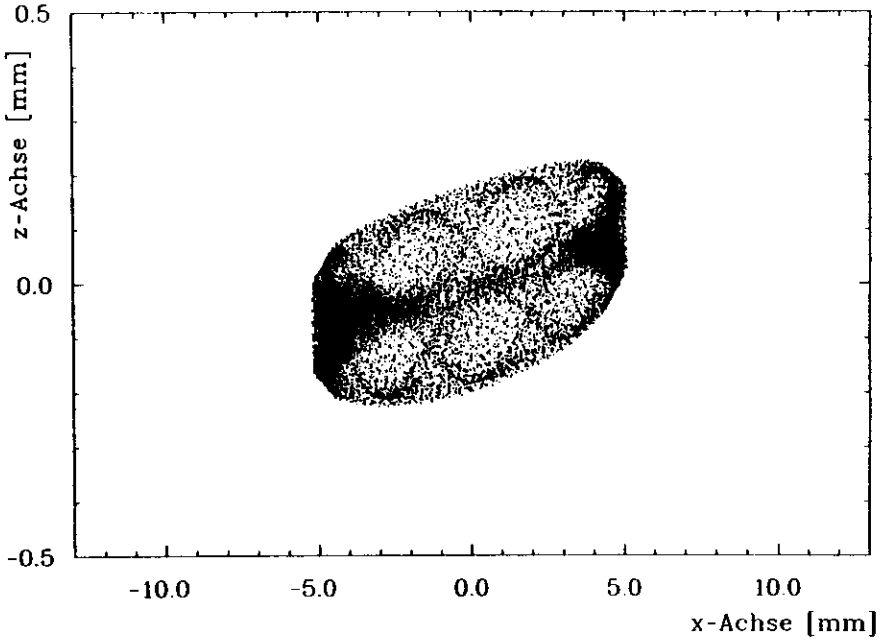


Abbildung 14: Ortsraum

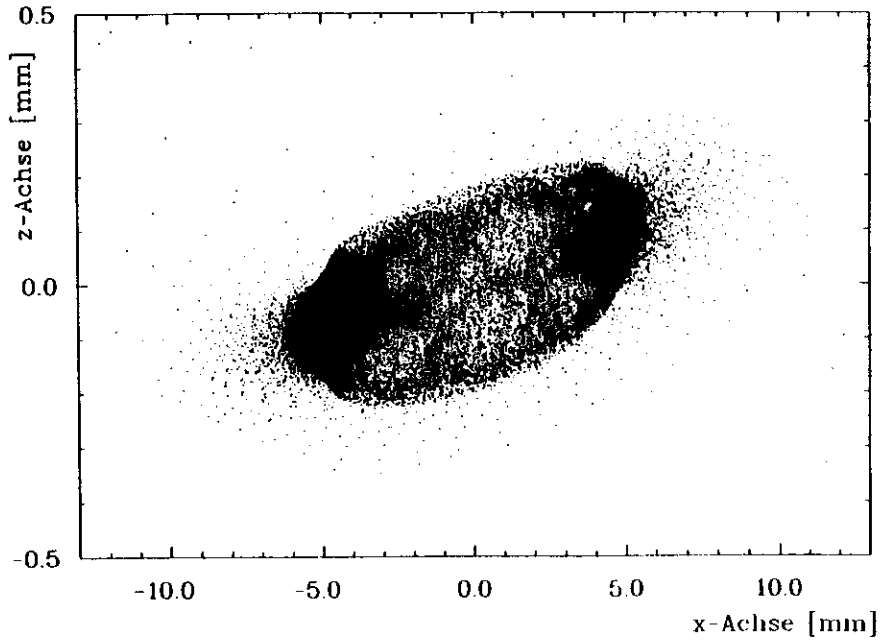


Abbildung 15: Ortsraum

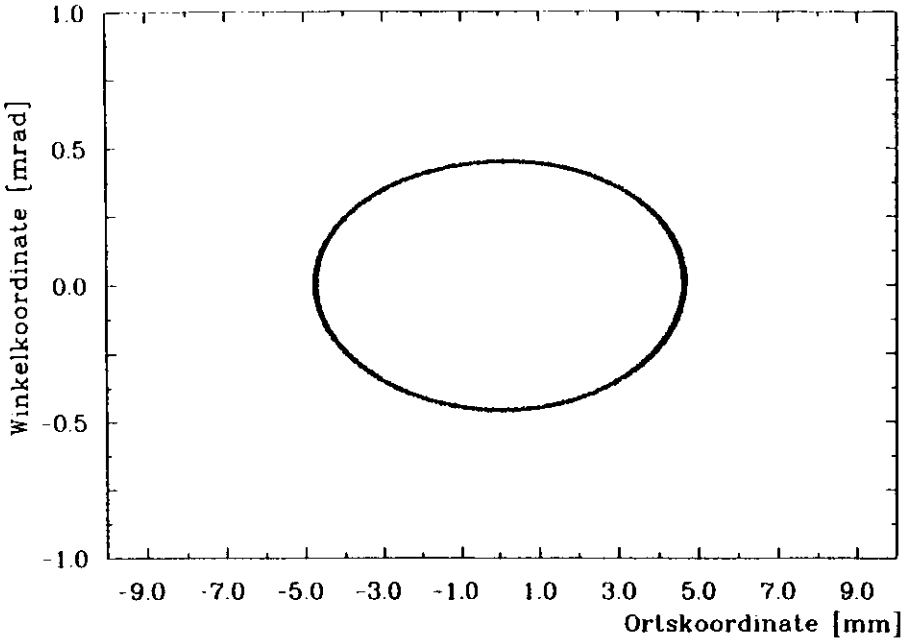


Abbildung 16: x-Phasenraum

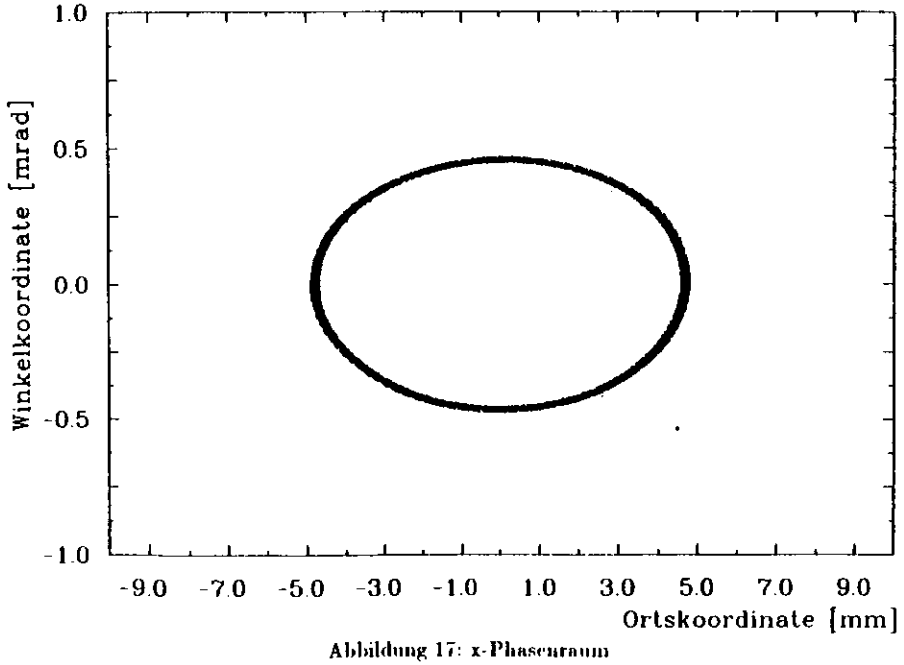


Abbildung 17: x-Phasenraum

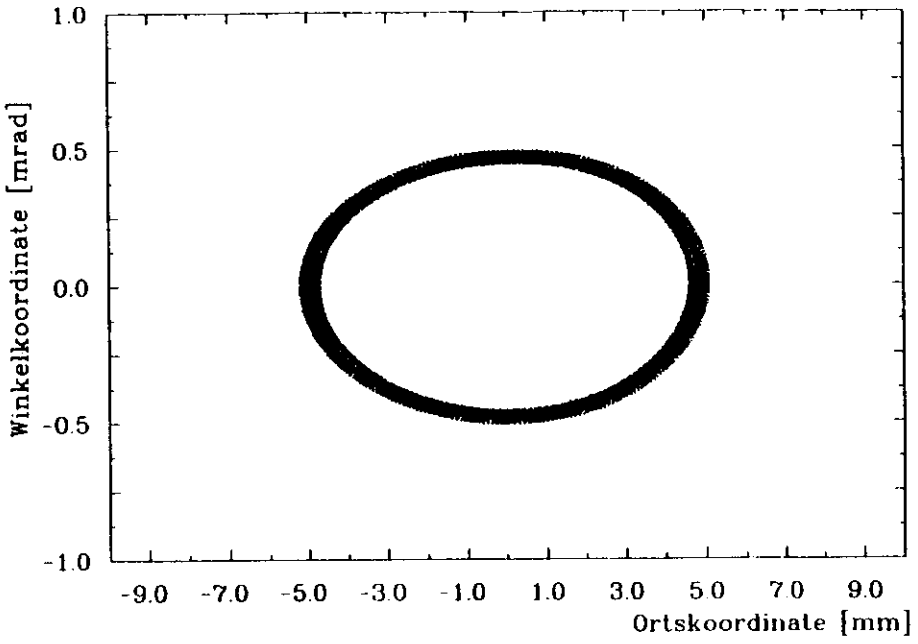


Abbildung 18: x-Phasenraum

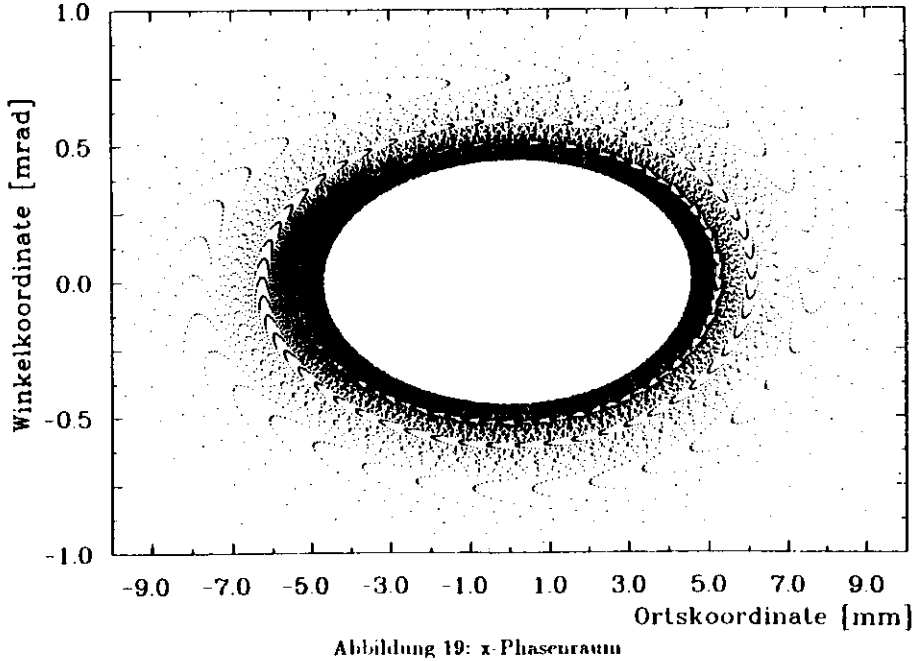


Abbildung 19: x-Phasenraum

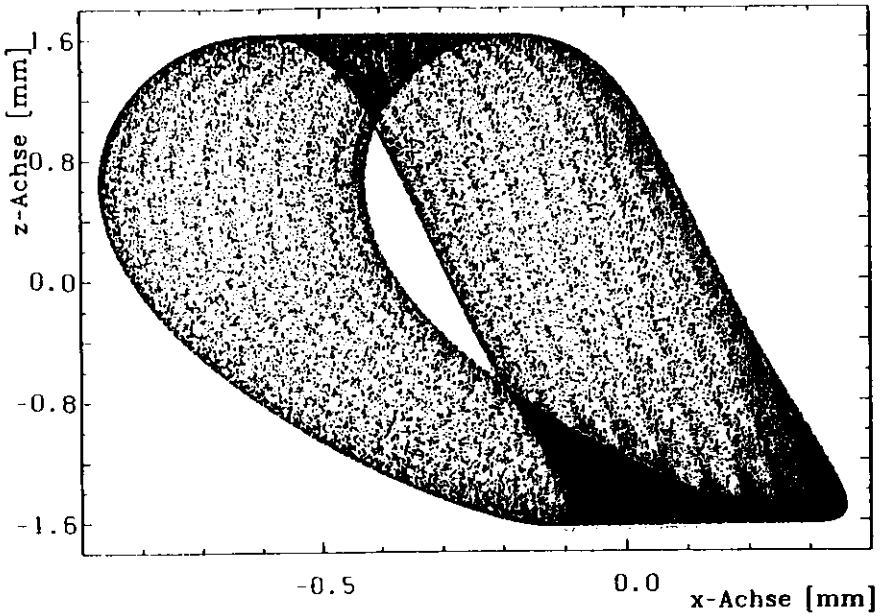


Abbildung 20: Ortsraum

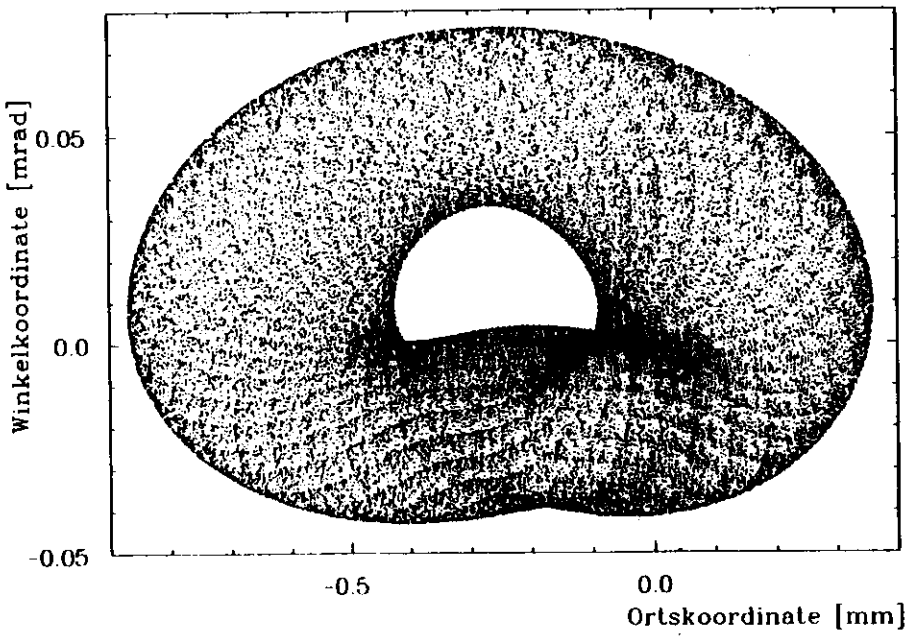


Abbildung 21: x-Phasenraum

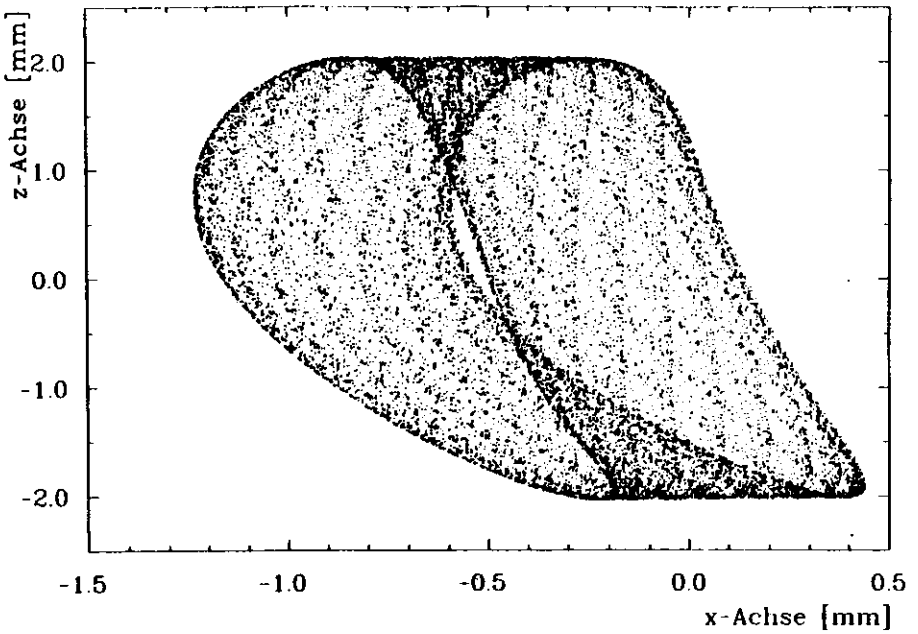


Abbildung 22: Ortsraum

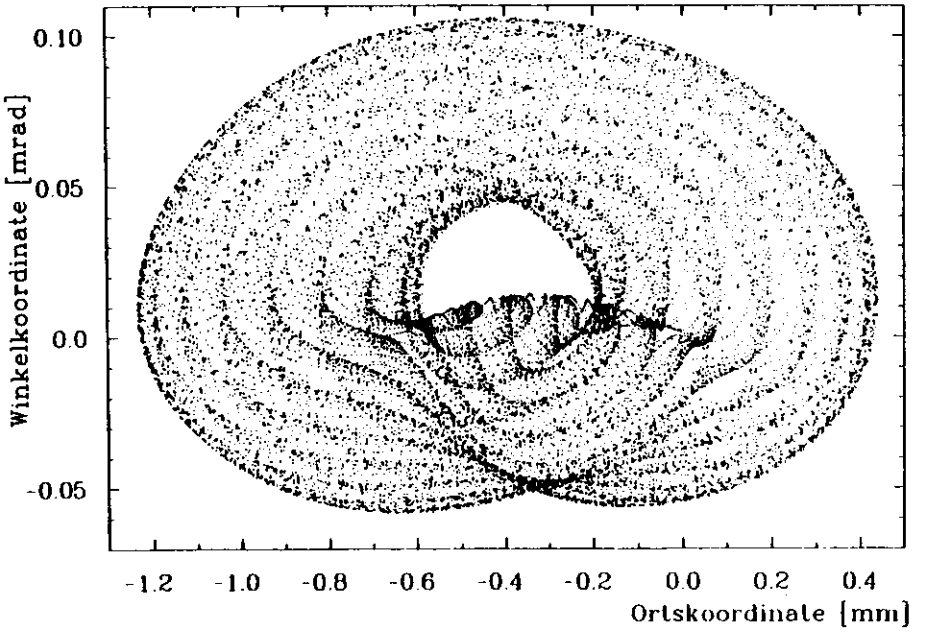


Abbildung 23: x-Phasenraum

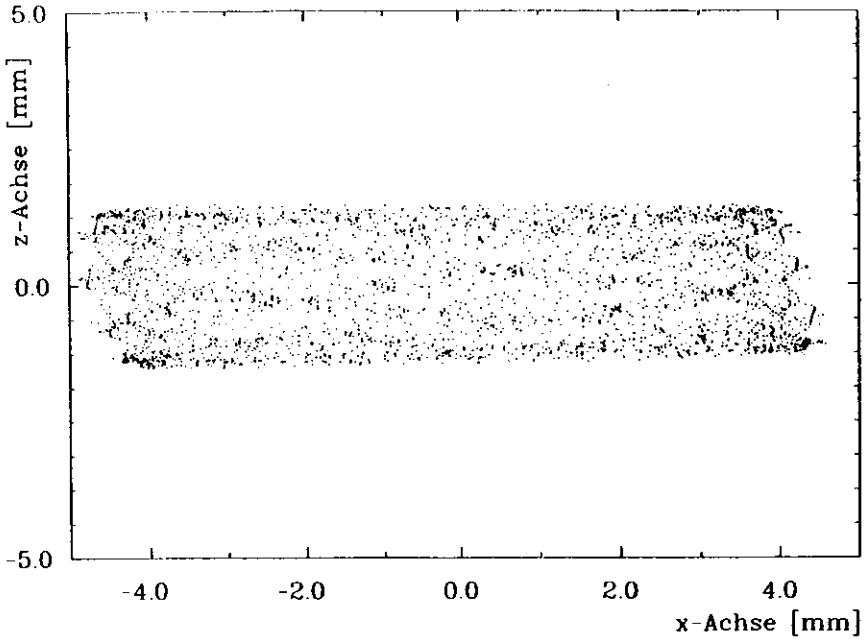


Abbildung 24: Ortsraum

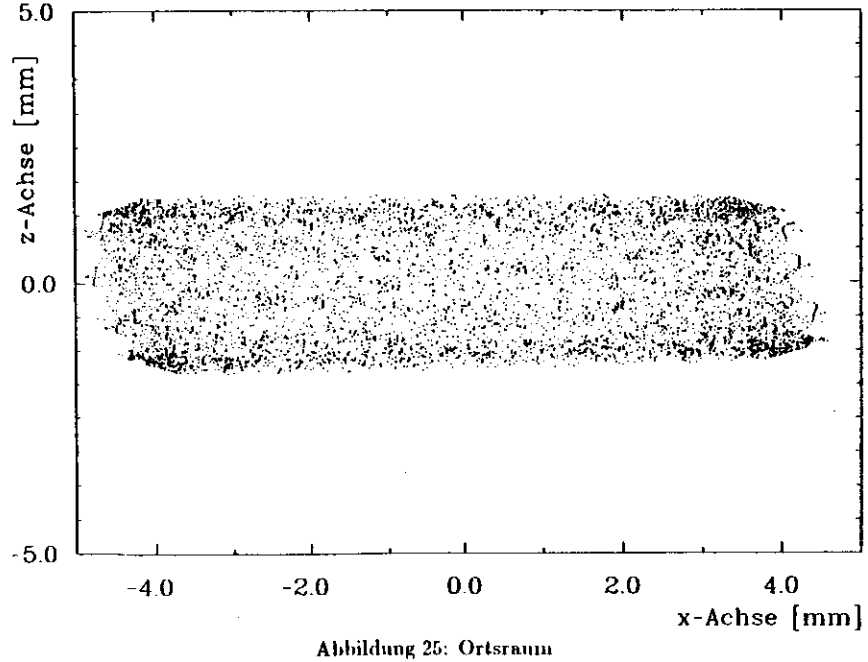


Abbildung 25: Ortsraum

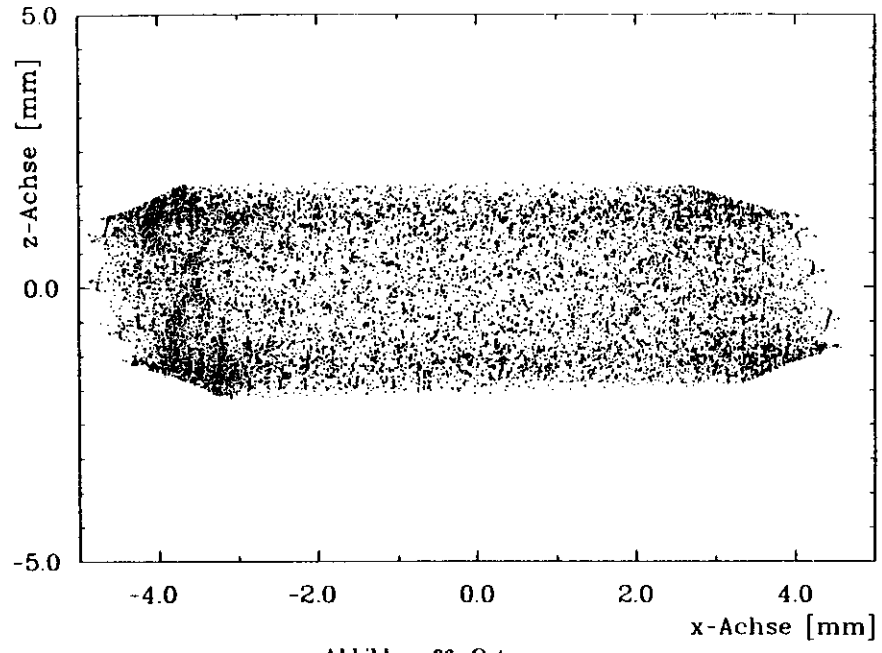


Abbildung 26: Ortsraum

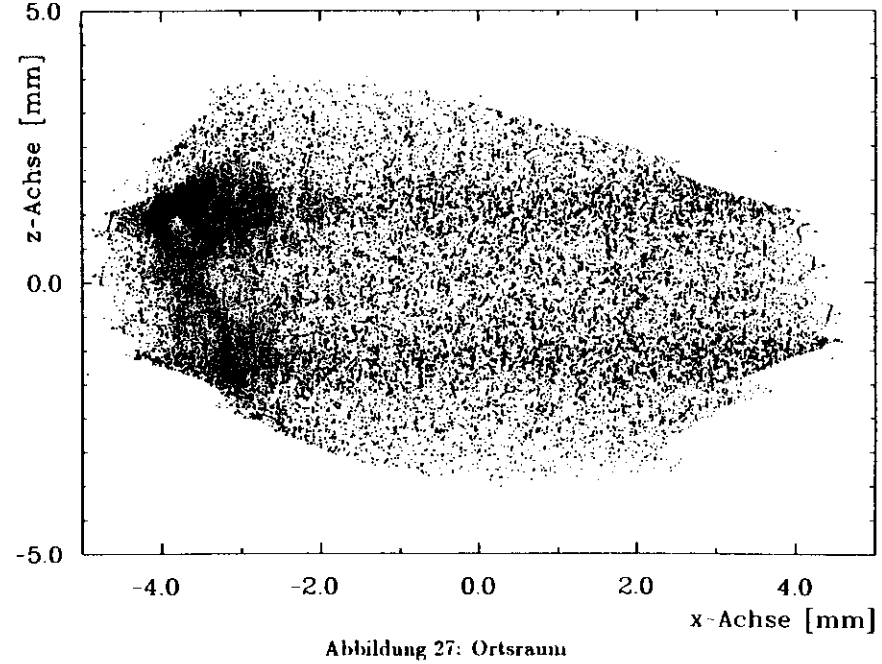


Abbildung 27: Ortsraum

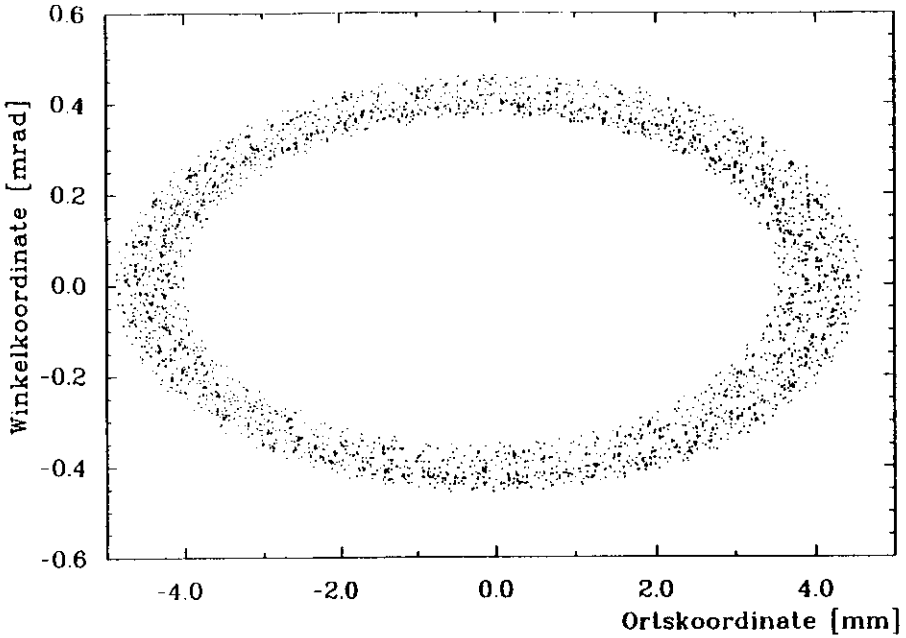


Abbildung 28: x-Phasenraum

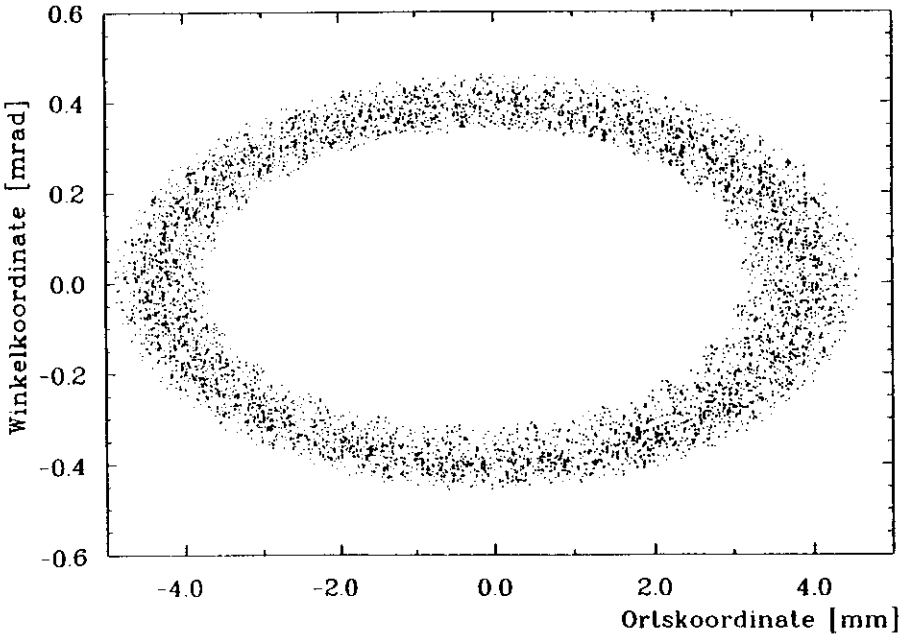


Abbildung 29: x-Phasenraum

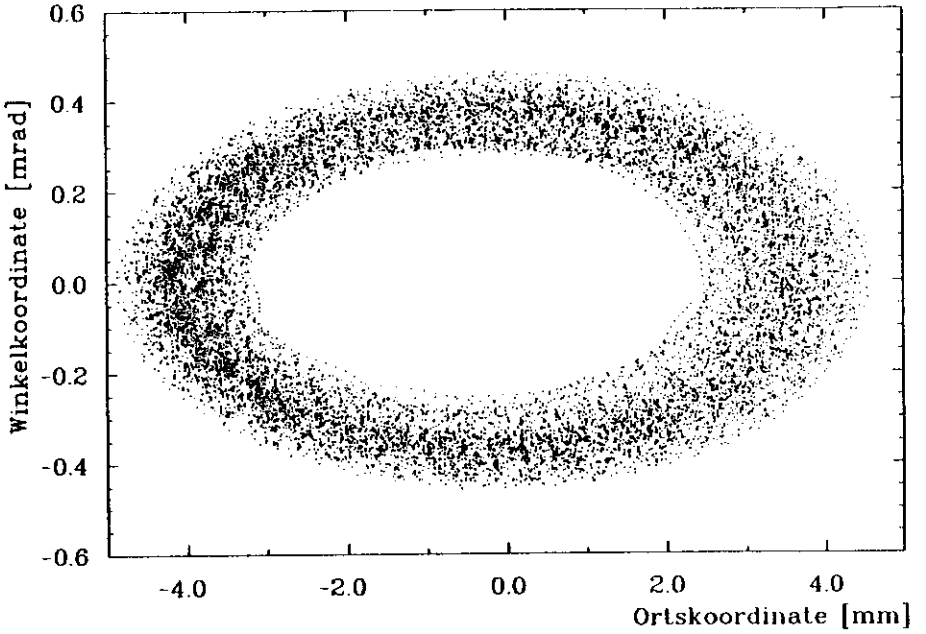


Abbildung 30: x-Phasenraum

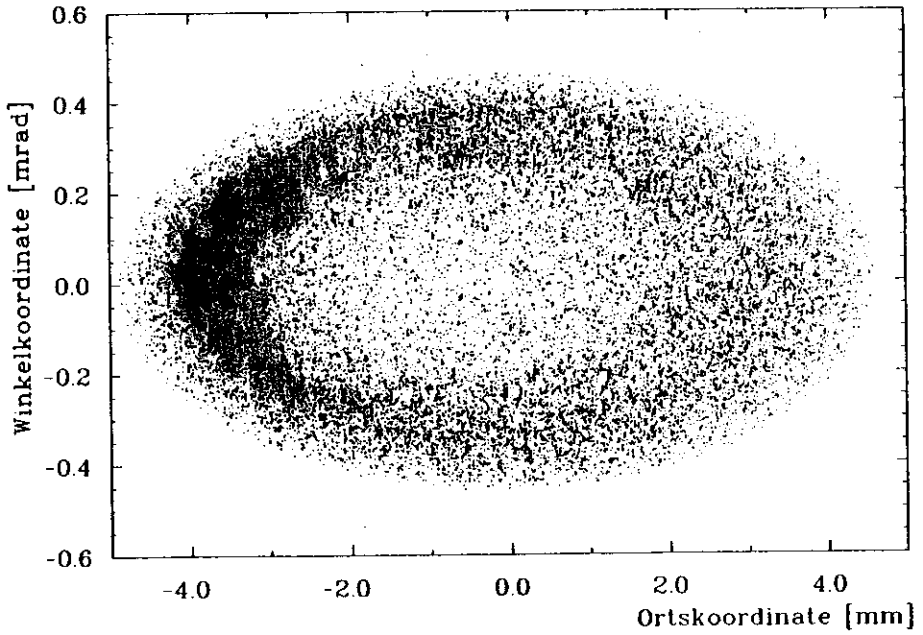


Abbildung 31: x-Phasenraum

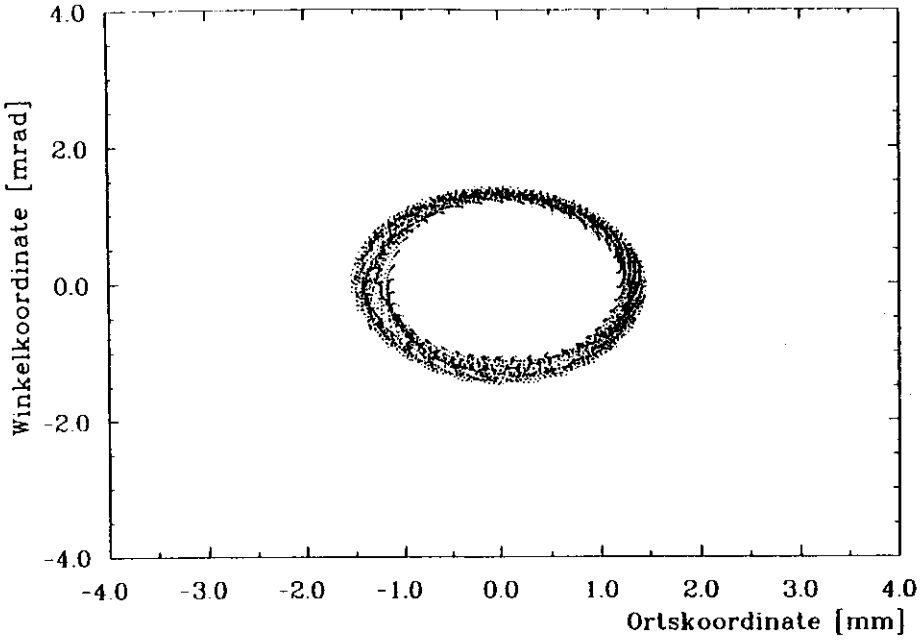


Abbildung 32: z-Phasenraum

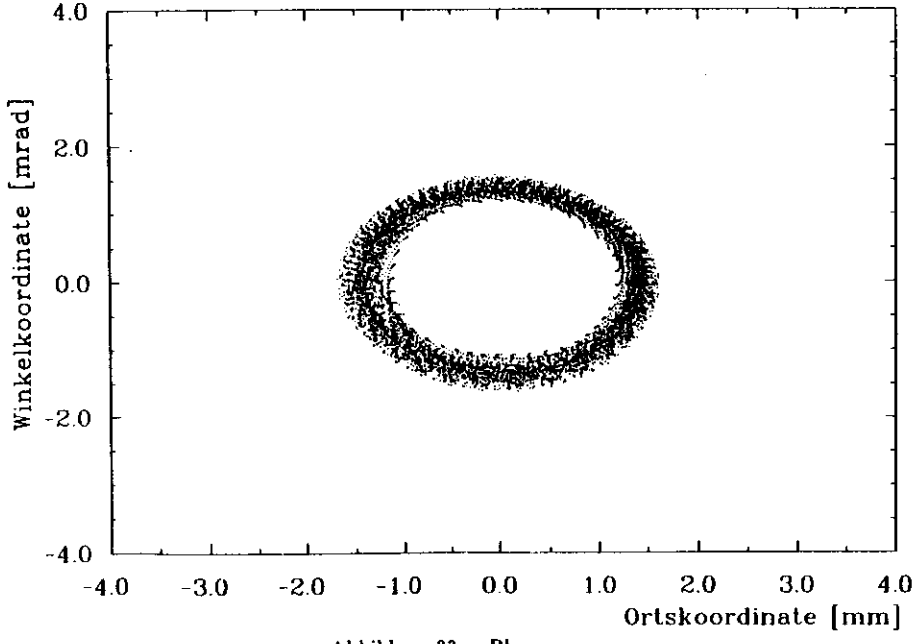


Abbildung 33: z-Phasenraum

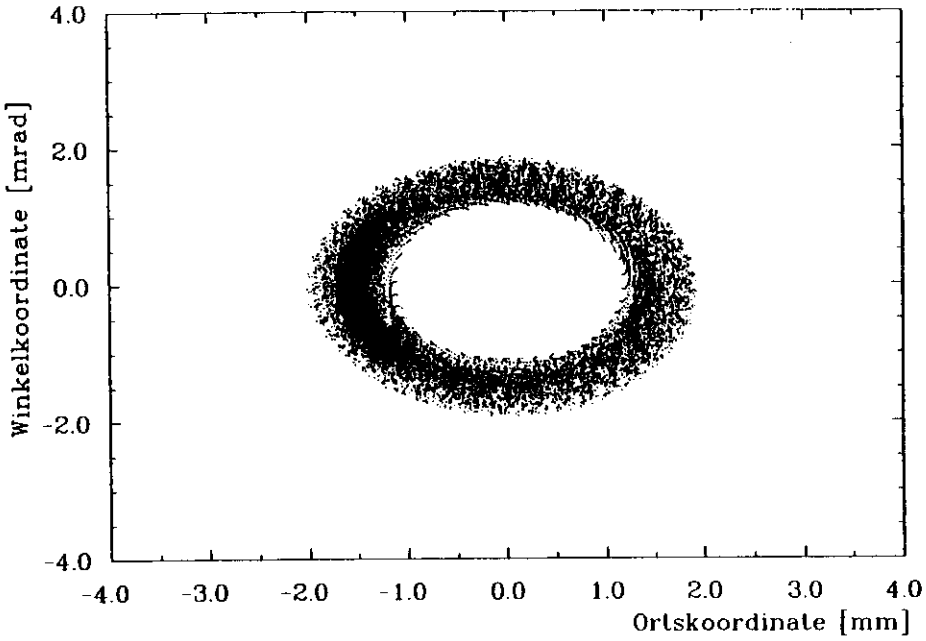


Abbildung 34: z-Phasenraum

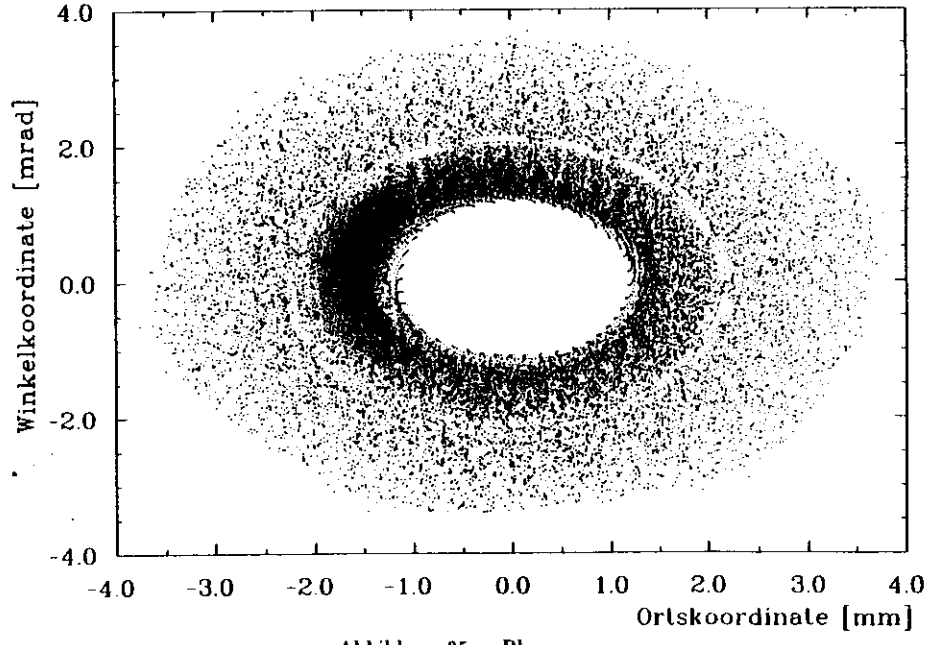


Abbildung 35: z-Phasenraum

6 Zusammenfassung

Das Ziel dieser Arbeit war die Entwicklung eines Rechnersystems, um die Bewegung von Protonen im HERA-Speicherring zu untersuchen.
Dazu wurde die Software für eine Verbindung zwischen dem *VME* -Bus und der DESY-IBM erstellt, wobei das PADAC-Netz als eine DESY interne Busnorm verwendet wurde. Als Entwicklungsumgebung diente ein Multiprozessorsystem, BOP genannt, welches für Protonenstrahlsimulationen entwickelt worden war.
Nachdem die für den Datentransfer notwendigen Programme erstellt worden waren, wurden über die von mir installierte Verbindung Dateien zur *IBM* übertragen. Bei den Daten handelt es sich um Ergebnisse von Simulationen, welche ich auf dem BOP-System mit einer modifizierten Version des Programmcodes *RACETRACK* durchführte. Mit der Datenübertragung wurde die Transferrate von etwa 60 kByte/s, welche von den Entwicklern des PADAC-Systems im EXP-Manual [16] angegeben wurde, erreicht. Es existiert damit eine dokumentierte und in der Hochsprache C geschriebene Software für die Übertragung von Textdateien zwischen dem *VME* -Bus und der DESY-IBM Anlage.
Als Demonstration des Systems wurden Untersuchungen des Bahnverhaltens von Teilchen im HERA Speicherring durchgeführt. Anhand der graphischen Ergebnisse sind die Abweichungen des HERA-Protonenringes von einer linearen Maschine deutlich geworden. Außerdem zeigen die Bilder die Veränderung der Teilchendynamik, wenn der Bereich der Apertur verlassen wird.

Anhang

A Tabellen über den VME-Bus

Hexadezimal Code	Adreß-Modifier 5	4	3	2	1	0	Funktion	Definiert durch
3F	H	H	H	H	H	H	Standard-Supervisory Zugriff, aufsteigend	VMEbus-Spezifikationen
3E	H	H	H	H	H	L	Standard-Supervisory-Programmzugriff	VMEbus-Spezifikationen
3D	H	H	H	H	L	H	Standard-Supervisory-Daten-Zugriff	VMEbus-Spezifikationen
3C	H	H	H	H	L	L	Undefiniert	Reserviert
3B	H	H	H	L	H	H	Nichtprivilegierter Standardzugriff, aufsteigend	VMEbus-Spezifikationen
3A	H	H	H	L	H	L	Nichtprivilegierter Standard-Programmzugriff	VMEbus-Spezifikationen
39	H	H	H	L	L	H	Nichtprivilegierter Datenzugriff	VMEbus-Spezifikationen
38	H	H	H	L	L	L	Undefiniert	Reserviert
30...37	H	H	L	X	X	X	Undefiniert	Reserviert
2F	H	L	H	H	H	H	Undefiniert	Reserviert
2E	H	L	H	H	H	L	Undefiniert	Reserviert
2D	H	L	H	H	L	H	kurzer Supervisory-E/A-Zugriff	VMEbus-Spezifikationen
2C	H	L	H	H	L	L	Undefiniert	Reserviert
2B	H	L	H	L	H	H	Undefiniert	Reserviert
2A	H	L	H	L	H	L	Undefiniert	Reserviert
29	H	L	H	L	L	H	Kurzer nichtprivilegierter E/A-Zugriff	VMEbus-Spezifikationen
28	H	L	H	L	L	L	Undefiniert	Reserviert
20...27	H	L	L	X	X	X	Undefiniert	Reserviert
10...1F	L	H	X	X	X	X	Undefiniert	Benutzer
0F	L	L	H	H	H	H	Erweiterter Supervisory-Zugriff, aufsteigend	VMEbus-Spezifikationen
0E	L	L	H	H	H	L	Erweiterter Supervisory-Programmzugriff	VMEbus-Spezifikationen
0D	L	L	H	H	L	H	Erweiterter Supervisory-Datenzugriff	VMEbus-Spezifikationen
0C	L	L	H	H	L	L	Undefiniert	Reserviert
0B	L	L	H	L	H	H	Erweiterter nichtprivilegierter Zugriff, aufsteigend	VMEbus-Spezifikationen
0A	L	L	H	L	H	L	Erweiterter nichtprivilegierter Programmzugriff	VMEbus-Spezifikationen
09	L	L	H	L	L	H	Erweiterter nichtprivilegierter Datenzugriff	VMEbus-Spezifikationen
08	L	L	H	L	L	L	Undefiniert	Reserviert
00...07	L	L	L	X	X	X	Undefiniert	Reserviert

Abbildung 36: Adressen-Modifier-Codes

	Stecker 1				Stecker 2 (VMXBus)		
	A	B	C		A	B	C
Anschluß-Nr.	Signal-Mnemonik	Signal-Mnemonik	Signal-Mnemonik	Anschluß-Nr.	Signal-Mnemonik	Signal-Mnemonik	Signal-Mnemonik
1	D00	BBSY*	D08	1	DB00	+5 V	DB01
2	D01	BCLR*	D09	2	DB02	GND	DB03
3	D02	ACFAIL*	D10	3	DB04	Reserviert	DB05
4	D03	BG0IN*	D11	4	DB06	A24	DB07
5	D04	BG0OUT*	D12	5	DB08	A25	DB09
6	D05	BG1IN*	D13	6	DB10	A26	DB11
7	D06	BG1OUT*	D14	7	DB12	A27	DB13
8	D07	BG2IN*	D15	8	DB14	A28	DB15
9	GND	BG2OUT*	GND	9	DB16	A29	DB17
10	SYSCLK	BG3IN*	SYSFALL*	10	DB18	A30	DB19
11	GND	BG3OUT*	BERR*	11	DB20	A31	DB21
12	DS1*	BR0*	SYSRESET*	12	DB22	GND	DB23
13	DS0*	BR1*	LWORD*	13	DB24	+5 V	DB25
14	WRITE*	BR2*	AM5*	14	DB26	D16	DB27
15	GND	BR3*	A23	15	DB28	D17	DB29
16	DTACK*	AM0	A22	16	DB30	D18	DB31
17	GND	AM1	A21	17	R/W	D19	STIL0
18	AS*	AM2	A20	18	LOCK*	D20	STIL1
19	GND	AM3	A19	19	LWORD/A12	D21	STIL2
20	IACK*	GND	A18	20	A02/A14	D22	A01/A13
21	IACKIN*	SERCLK	A17	21	A04/A16	D23	A03/A15
22	IACKOUT*	SERDAT	A16	22	A06/A18	GND	A05/A17
23	AM4	GND	A15	23	A08/A20	D24	A07/A19
24	A07	IRQ7*	A14	24	A10/A22	D25	A09/A21
25	A06	IRQ6*	A13	25	STIL3	D26	A11/A23
26	A05	IRQ5*	A12	26	STIL4	D27	LAS*
27	A04	IRQ4	A11	27	STIL5	D28	UAS*
28	A03	IRQ3	A10	28	STIL6	D29	UDS*
29	A02	IRQ*	A09	29	STIL7	D30	LDS*
30	A01	IRQ1*	A08	30	STIL8	D31	DERR*
31	-12 V	+5 V STDBY	+12 V	31	STIL9	GND	ACK*
32	+5 V	+5 V	+5 V	32	SMRQ*	+5 V	SMACK*

VMSbus (Stecker 1, B: GND, SERCLK, SERDAT, GND)

VMX oder Benutzer-E/A (Stecker 2, A und C)

Abbildung 37: Anschlußbelegung beim VME/VMS/VMX-Bus

B Details zur Hard- und Softwareumgebung

B.1 Die Variablentypen von *C*

Allgemeines Um in den Programmbeschreibungen der Abschnitte C.3.1 und 4.2.3 die Original *C* -Deklarationen verwenden zu können, gehe ich an dieser Stelle näher auf die Standardvariabletypen von *C* (siehe auch [1]) ein.
In der Tabelle 3 sind die Standardvariablen des *C* -Compilers auf der QU-68000 zusammengestellt. Generell weichen auch andere *C* -Compiler von diesem Typenumfang nur im geringen Maße ab und größere Unterschiede kann es hier höchstens bei den Wertebereichen geben.
Neben den in der Tabelle 3 stehenden Typennamen kann ihnen jeweils noch die Bezeichnung

Variablentypen von C		
Name (Typ)	Länge in Bit	Eigenschaft
char	8	ganze Zahlen & ASCII Zeichen
unsigned char	8	ganze Zahlen & ASCII Zeichen
int	16	ganze Zahlen
unsigned int	16	ganze Zahlen
long	32	ganze Zahlen & Adressen
unsigned long	32	ganze Zahlen & Adressen
float	32	Fließkommazahlen
double	64	Fließkommazahlen (doppelt genau)

Tabelle 3: Variablentypen von *C* (PCS)

static vorangestellt werden. Hierdurch wird erreicht, daß die Variablen beim Verlassen der Routine ihren Wert nicht verlieren.

Variablenübergabe bei Funktionen Eine Funktion wird in *C* auf die folgende Art definiert :

```
<Typ>Funktionsname ( <Var_1><,...,Var_N>)
<Typ> <*> Var_1 ;
.
.

<Typ> <*>Var_N ;
{ Anweisungen} ;
```

Alle Texte welche in diesen Zeichen <...> dargestellt sind, sind optional.
Die Sprache *C* kennt sowohl die Werte, als auch die Adressen von Variablen. Ebenso können bei der Parameterübergabe von Funktionen sowohl Werte, als auch Adressen übergeben werden. Sollen bei Funktionen nur die Werte übergeben werden, so kann die übergeordnete Ebene zwar die Werte der Funktion, aber die Funktion nicht die Werte der übergeordneten Ebene verändern. Anders ist es, wenn die Adressen übergeben werden, dann erfolgt der

Datenaustausch in beide Richtungen. Dieses wird z.B. bei den Variablen über den Fehlerzustand *error nr* ausgenutzt. Die Übergabe von Adressen und Werten hängt mit dem Stern (*) zusammen, welcher den Variablennamen bei der Deklaration vorangestellt werden kann. Ein Stern kann immer vor einem Ausdruck stehen, wenn dieser eine Adresse (Pointer) darstellt. Mit diesem Stern ist mit dem Ausdruck nicht die Adresse, sondern der Wert, welcher in dieser Adresse steht gemeint. Diesen Zusammenhang will ich noch einmal an einem Beispiel erläutern. Die Deklaration :

*int * zahl*;

bedeutet, daß **zahl* eine Integerzahl und *zahl* die Adresse auf diese Integerzahl ist. Wenn in der Parameterliste hinter dem Funktionsnamen nun einfach *zahl* also eine Adresse steht, so bewirkt eine Veränderung dieser Variablen (**zahl*) in der Funktion auch eine Veränderung der entsprechenden Variable in der übergeordneten Ebene.

B.2 Das *MAKE*-Konzept

Das Programm *MAKE* erlaubt eine weitgehend automatische Generierung von Programmsystemen. Unter einem Programmsystem ist hierbei ein aus mehreren Moduln zusammengesetztes Programm zu verstehen. Das *MAKE*-Konzept ist jedoch so allgemein gehalten, daß auch ein System aus mehreren Programmen oder anders gearteten Dateien damit erstellt werden kann. Für diese Generierung benutzt *MAKE* zwei Informationsquellen:

1. Eine Beschreibung von Abhängigkeiten

2. Das Modifikationsdatum der aus 1. ermittelten Dateien

Die in 1. angegebene Beschreibung muß vom Benutzer selbst erstellt werden. Sie wird durch Abhängigkeiten, die *MAKE* selbst kennt, ergänzt. Die Abhängigkeiten wird man in der Regel in eine Datei schreiben, welche dann von *MAKE* benutzt wird. Wenn beim Aufruf keine Beschreibungsdateien explizit angegeben werden, so sucht *MAKE* in der Datei *makefile* oder *MAKEFILE* des aktuellen Katalogs. Aus diesem Grunde wird statt "*die Beschreibung der Abhängigkeiten*" auch der Begriff *makefile* verwendet. In dieser Datei sind die neu zu erstellenden Dateien, *Zielobjekte* (*genannt*), aufzuführen und zwar zusammen mit den Dateien, von denen sie abhängen und den Kommandos, die die Generierung durchführen. Dieses Ganze kann hierarchisch erfolgen. *MAKE* stellt zur Generierung des Zielsystems fest, welche der zur Erstellung des Systems notwendigen Dateien seit der letzten Erstellung modifiziert wurden und ermittelt aus *makefile*, welche Arbeiten zu tun sind (z.B. welche Teile neu übersetzt und gebunden werden müssen). In der Beschreibung werden 4 Arten von Anweisungen benutzt:

- Makrodefinitionen,
- Definition von Abhängigkeiten,
- Shellkommandos,
- Steueranweisungen.

Makrodefinitionen in *MAKE* Bei einer Makrodefinition wird dem Makronamen die nachfolgende Zeichenkette zugewiesen. Mit der Definition :

name = zeichenkette

wird "$(*name*)" im *makefile* jeweils durch *zeichenkette* ersetzt. Dieses Hilfsmittel Abkürzungen durch Makrodefinitionen einzuführen, ermöglicht insbesondere bei *makefiles* von großen Systemen eine wesentlich verbesserte Übersichtlichkeit.

Beschreibung von Abhängigkeiten Eine Beschreibung der Abhängigkeit der Zielobjekte von Dateien hat die folgende Form :

ziel { ziel 1 ... ziel_n } : { datei 1 ... datei_n} { ; kommandos }
{ <**tab**>kommando_1 { # kommentar}
<**tab**>kommando_2 { # kommentar}
.
.
<**tab**>kommando_n { # kommentar}

Diese Beschreibung ({...} zeigt hier optionale Teile) gibt an, daß die angegebenen Zielobjekte *ziel* bis *ziel_n* von den dem Doppelpunkt folgenden Dateien abhängen. Diesen können, durch Semikolon getrennt, in der gleichen Zeile Kommandos folgen. Wurde eine der aufgeführten Dateien seit der letzten Erstellung des Zielobjektes geändert, so muß das Zielobjekt neu generiert werden. Diese Generierung geschieht durch die nachfolgenden Kommandos. Alle Zeilen, die der Objektdefinition folgen und mit einem Tabulatorzeichen beginnen, werden als solche Kommandos interpretiert und entsprechend der Shell übergeben. Ein Kommentar kann, durch ein " # " eingeleitet, den Kommandos folgen. Er wird jeweils durch das Zeilenende abgeschlossen.
Fehlt die Angabe der Dateien (*datei* bis *datei_n*), so werden die Abhängigkeiten verwendet, die *MAKE* implizit kennt , z.B. hängen Objektdateien von Quelldateien ab.
Es müssen nicht alle Anweisungen der Generierung des Zielobjektes im strengen Sinne dienen, sondern sie können auch nicht mehr benötigte Dateien löschen oder ähnliche Funktionen ausführen.

B.3 Näheres über den VME-Bus

B.3.1 Der Daten-Transfer-Bus (DTB)

Der Daten-Transfer-Bus enthält alle Daten- und Adressleitungen sowie die zum Datentransfer notwendigen Steuerleitungen. Mit ihnen ist es dem *VME*-Bus möglich, einen multiplexfreien und asynchronen Datentransfer durchzuführen. Eine multiplexfreie Datenübertragung bedeutet dabei, daß die Adressen und Daten bei der Übertragung nicht in Blöcke unterteilt, sondern in einem Schritt unzerteilt übertragen werden. Dieser multiplexfreie Datentransfer benötigt zwar mehr Treiberbausteine und Signalleitungen, hat aber dafür eine einfachere logische Struktur und gewährleistet einen wesentlich schnelleren Datentransfer. Bei einem asynchronen Datentransfer quittiert der Empfänger jedes Mal, nachdem er die Daten erhalten hat, den Empfang und fordert damit den Sender auf, weitere Daten zu schicken. Im

Gegensatz dazu existiert beim synchronen Datentransfer ein zentrales Taktsignal, nach welchem sich der Datentransfer richtet. Das heißt, nach der Aufnahme der Datenübertragung erhält der Empfänger mit jedem neuen Taktsignal auch neue Daten, ohne daß er den Erhalt dieser Daten quittiert . Dabei wird der Start und das Ende der Datenübertragung bei beiden Systemen typischerweise durch eine weitere Kontrolleitung angezeigt. Der synchrone Datentransfer ist bei Systemen mit gleich schnellen Komponenten zwar effektiver, aber langsamere Peripheriegeräte oder Speicher können in einem solchen System nicht verwendet werden, ohne daß das Busprotokoll verändert wird. Wie man auch in dem Bild 38 sieht, läuft beim *VME* -Bus eine solche asynchrone Datenübertragung folgendermaßen ab:
Der Sender (Master) legt auf dem Adressbus die von ihm gewünschte Adresse an und teilt dem Empfänger (Slave) durch die Aktivierung des Signals "Address Strobe" (AS) mit, daß die gültige Adresse anliegt. Das Signal "WRITE" zeigt an, ob es sich um eine Schreib- oder Leseoperation handelt. Bei einem Lesezyklus würde dem Slave über die beiden Data-Strobe-Leitungen $\overline{DS0}$ und $\overline{DS1}$ mitgeteilt werden, daß der Master zum Einlesen der Daten bereit ist. Daraufhin würde der adressierte Slave die Daten auf den Datenbus legen und über die Leitung "Data Acknowledge" ($\overline{DTACK}$) mitteilen, daß gültige Daten anliegen. Bei einem Schreibzyklus hingegen werden die Daten vom dem Master auf den Datenbus gelegt und danach die Signalleitungen $\overline{DS0}$ und $\overline{DS1}$ aktiviert. Sobald der Slave dann die Daten aufgenommen hat, signalisiert er dieses durch die Aktivierung von $\overline{DTACK}$. In den beiden Fällen gibt der Master den Bus für den nächsten Zyklus erst dann frei, wenn er das "Quittierungssignal" $\overline{DTACK}$ vom Slave erhalten hat. Der Slave seinerseits gibt den Datenbus erst dann frei (Lesezyklus), wenn der Master die Datenstrobes $\overline{DS0}$ und $\overline{DS1}$ deaktiviert hat. Ist der Slave aufgrund einer fehlerhaften Signalkombiation auf dem Daten-Transfer-Bus nicht in der Lage, der Schreib- oder Lese-Anforderung ordnungsgemäß nachzukommen, sendet er anstatt des Signals $\overline{DTACK}$ das Signal "Bus Error" ($\overline{BERR}$) aus. Der Master kann dann in eine entsprechende Fehlerbehandlungsroutine gehen. Über dieses sogenannte "Handshaking" die Charakteristik des asynchronen Busses ist eine problemlose Zusammenarbeit von Einheiten unterschiedlicher Geschwindigkeit möglich.
Nun sollen nicht immer Daten mit einer Länge von 32-Bit auf einem Adressbereich von 32-Bit übertragen werden, deshalb bestehen für die Daten die beiden Datenstrobes $\overline{DS0}$ und $\overline{DS1}$ sowie das Signal $\overline{LWORD}$. Bei einem Datentransfer, welcher die Länge eines Wortes (16-Bit) hat, werden beide Datenstrobes aktiviert, sobald die Daten an dem Bus anliegen. Dabei steht das Datenstrobe $\overline{DS0}$ für das untere- und das Datenstrobe $\overline{DS1}$ für das obere Byte des Datenwortes. Falls nur ein Byte übertragen wird, wird auch nur das untere Datenstrobe $\overline{DS0}$ aktiviert, und wenn 32- Bits übertragen werden sollen, so wird dieses über das Signal $\overline{LWORD}$ angezeigt.
Was bei den Daten mit den Signalleitungen $\overline{DS0}$, $\overline{DS1}$ und $\overline{LWORD}$ gemacht wird, geschieht bei den Adressen über die sogenannten Adreß-Modifier. Dieses sind die Leitungen *AM0 ...AM5* des Busses. Von den mit diesen Leitungen möglichen 64 verschiedenen Adreßinformationen werden drei für unterschiedliche Adreßbereiche benutzt, und zwar für den vollen 32-Bit-Bereich (Extended), den 24-Bit-Bereich (Standard) sowie den 16-Bit- Bereich (Short). Eine weitere Anwendung der dieser Adreß-Modifier ist zum Beispiel die Unterteilung des Arbeitspeichers in einen geschützten und einen nicht geschützten Bereich, was die Systemsicherheit beträchtlich erhöhen kann. Im Anhang A ist in der Abbildung 36 eine Übersicht über die Funktion der Adreß-Modifier angegeben.

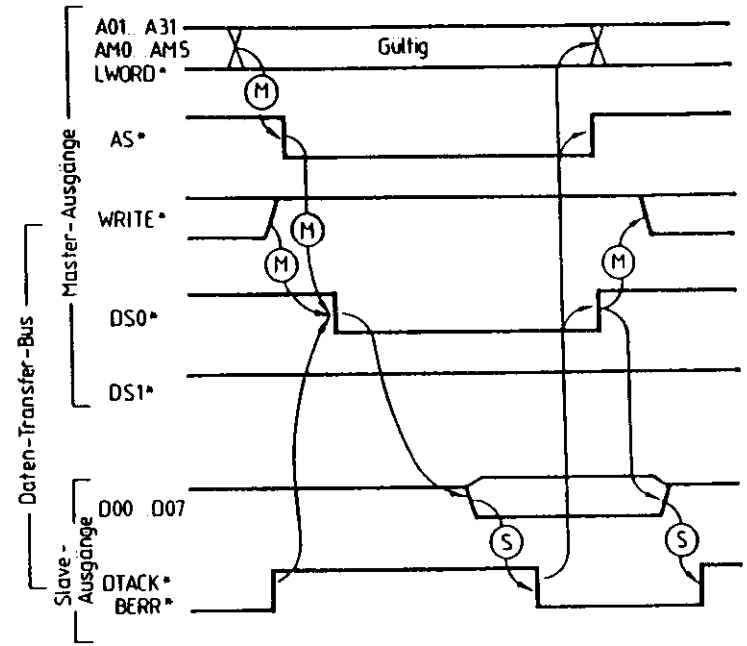


Abbildung 38: Timing eines Byte-Lesezykluses auf dem Datentransfer-Bus. Ein Stern über dem Signalnamen bedeutet, daß der niedrige Pegel (LOW) der aktive Zustand ist, wie auch die Linie über den Signalen im Text. Die mit M(S) bezeichneten Pfeile geben das vom Master (Slave) garantierte Zeitverhalten an.

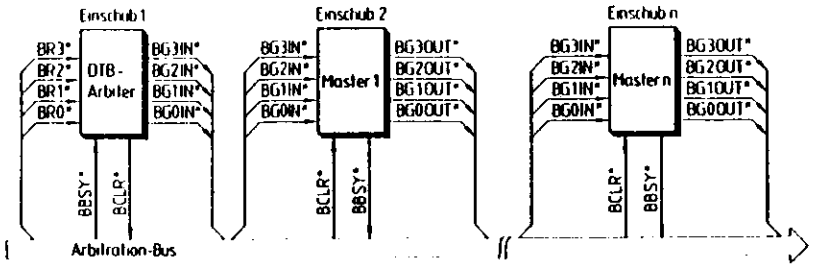


Abbildung 39: VME - Busarbitration

B.3.2 Der Arbitration-Bus für Multiprozessor-Systeme

Unter einem Multiprozessor-System versteht man ein Rechnerkonzept, bei welchem mehrere Teile dieses Rechnersystems auf den Datentransferbus zugreifen und damit zum MASTER dieses Systems werden können. Für die Koordination auf einem solchen Bus ist ein sogenannter Arbiter (Auswähler) verantwortlich. Dieser benötigt, um seine Aufgabe wahrnehmen zu können, verschiedene Leitungen, welche bei dem *VME* -Bus zu dem sogenannten Arbitration-Bus zusammengefaßt sind.

Den Aufbau dieses Busses sieht man auf dem Bild 39. Jeder Master kann über eine der vier Bus-Request-Leitungen $\overline{BR0}$...$\overline{BR3}$ den Bus für sich anfordern, wobei jede dieser Leitungen einer bestimmten Priorität entspricht. Wenn der Arbiter ein Signal über eine dieser Leitungen erhält, so vergleicht er die Busanforderung mit der Priorität des den Bus momentan innehabenden Masters. Ist die Priorität des letzteren höher, so kann der Anforderung nachgekommen werden, wenn der Bus freigegeben wird. Im anderen Fall aktiviert der Arbiter das Signal *Bus Clear* $\overline{BCLR}$, um den gerade aktiven Master zur Busfreigabe aufzufordern. Sobald dieser an einer passenden Stelle zur Unterbrechung seiner Aktivität angekommen ist, signalisiert er dieses, indem er das Signal *Bus-Busy* ($\overline{BBSY}$) deaktiviert. Daraufhin schickt der Arbiter der höchsten anstehenden Priorität eine Busfreigabe (Bus-Grant *BG* 0 ... *BG* 3) und deaktiviert die Signalleitung $\overline{BCLR}$.
Weil auf diese Weise von dem Arbiter nur vier Bus-Master koordiniert werden können, wurde beim *VME* -Bus eine weitere Prioritätsebene mittels der sogenanten *Daisy-Chain*-Technik realisiert:
Hierbei wird eine Signalleitung durch mehrere Module hindurchgeführt. Kommt das Signal bei dem ersten Modul an, so entscheidet dieses, ob es das betreffende Signal verarbeitet oder weiterleitet werden soll. Wird das betreffende Signal verarbeitet bzw. der Anforderung nachgekommen, so erfahren die weiteren Module von diesem Signal nichts. Wenn das Signal von dem ersten Modul hingegen nicht verarbeitet wird, so wird es an das nächste Modul weitergeleitet, welches das Signal ebenfalls verarbeiten oder weiterleiten kann usw. Das dem Arbiter am nächsten befindliche Modul besitzt also die höchste Priorität und diese nimmt

mit wachsender Entfernung vom Arbiter ab.
Bei dem *VME* -Bus besitzt jede der vier Anforderungsebenen $\overline{BR0}$... $\overline{BR3}$ eine eigene Daisy-Chain-Kette. Das entsprechende Bus-Grant-Signal *BG* gelangt also vom Arbiter an den Anschluß $\overline{BGXIN}$ des ersten Masters. Wenn dieser den Bus belegen will, gibt er das *BG* -Signal nicht weiter. Im anderen Fall aktiviert er seine Signalleitung $\overline{BGXOUT}$, welche direkt zu der $\overline{BGXIN}$-Leitung des nächsten Masters führt usw. Dieses System wird in dem Bild 39 durch die in einen Einschub ein- und auslaufenden Leitungen angedeutet. Mittels dieser Daisy- Chain-Struktur können also beliebig viele Master auf dem *VME* -Bus zusammenarbeiten. Es handelt sich also um einen echten Multi-Master-Bus.
Bei dem Aufbau eines *VME* -Bus -Systems muß man auf eine entsprechende Rückwandverdrahtung für die vier Daisy-Chain-Ebenen achten. Wenn sich in einem Einschub kein möglicher Master befindet, dann müssen an diesem Einschub die Eingänge $\overline{BGXIN}$ mit den entsprechenden Ausgängen verbunden werden. Dieses ist notwendig, weil sonst vom Arbiter weiter weg liegende Master nicht über die Bus-Grant-Leitungen mit dem Arbiter verbunden werden können.

B.4 Die Interrupt-Verarbeitung

Der *VME* -Bus erlaubt es, Interrupts eine bestimmte Priorität zuzuordnen. D.h., wenn mehrere Interruptanforderungen zur gleichen Zeit vorliegen, wird diejenige mit der höchsten Priorität zuerst abgearbeitet. Dasjenige Modul, welche die Interruptanforderung von einem anderen Modul bekommt, wird auch Interrupthandler genannt. Wenn sich in dem System nur ein einziger Interrupthandler befindet, läuft die Verarbeitung des Interrupts in den folgenden drei Phasen ab: **Interruptanforderung, Interrupterkennung und Abarbeitung der Interrupt-Service-Routine**

Interruptanforderung : Diese kann über sieben Leitungen, auch Interrupt-Request-Leitungen ($\overline{IRQ1}$...$\overline{IRQ7}$) genannt, erfolgen. Dabei hat die Leitung $\overline{IRQ7}$ die höchste und $\overline{IRQ1}$ die niedrigste Priorität. Sobald der Interrupthandler diese Anforderung erhält, entscheidet er aufgrund einer Bedingung (z.B. Interruptmaske im MC 68000), ob der Interrupt zugelassen wird oder nicht.
Um die Anzahl der Module, welche an die Interrupt-Request- Leitungen angeschlossen werden können, nicht zu begrenzen, wird ein Interrupt-Erkennungssignal über eine Daisy-Chain-Verbindung von Modul zu Modul geführt. Hier wird also wie beim Arbitration-Bus eine zweite Prioritätsebene aufgrund der physikalischen Anordnung der Interruptanfrage aufgebaut.

Interrupterkennung : Zu der Interrupterkennung gehört es, den Interruptrequester zu identifizieren und von ihm die Vektornummer zu erhalten. Über die Vektornummer erkennt der Interrupthandler aus einer Liste, wie er der Interruptanforderung nachzukommen hat. Hierzu muß der Interrupthandler zunächst den Daten-Transfer-Bus anfordern, falls er nicht gerade Master des Daten-Transfer-Busses ist. Wie dieses geschieht ist im oberen Abschnitt beschrieben.

Abarbeitung der Interrupt-Service-Routine : Der Interrupthandler hat nun die Vektornummer von dem Interruptrequester bekommen. Über diese Nummer wird jetzt die

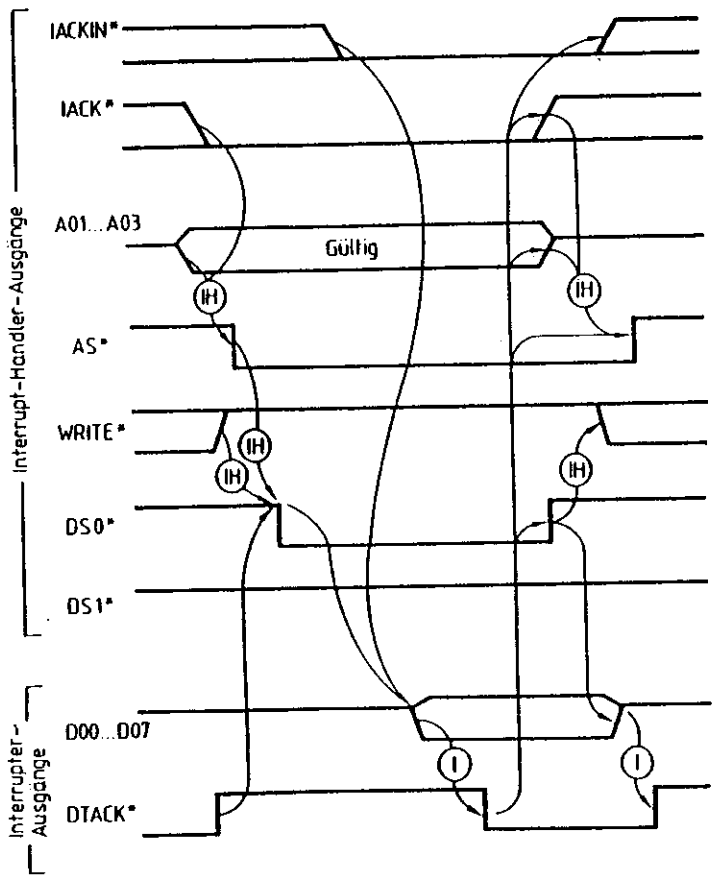


Abbildung 40: Das Zeitverhalten der Interrupterkennung beim VME-Bus. Ein Stern (*) hinter dem Signalnamen bedeutet dabei, daß der niedrige Pegel (Low) der aktive Zustand ist. Die mit IH (I) bezeichneten Pfeile geben das vom Interrupt-Handler (Interrupt-Requester) garantierte Zeitverhalten an.

Adresse der Interrupt-Service-Routine gefunden, von wo sich der Prozessor die nächsten abzuarbeitenden Befehle holt.

Wie der zeitliche Ablauf der Interrupterkennung aussieht, wenn der Interrupthandler den Daten-Transfer-Bus zugeteilt bekommen hat, ist in dem Bild 40 zu sehen. Zuerst wird das Signal *Interrupt-Acknowledge* IACK durch den Interrupthandler aktiviert. Gleichzeitig legt dieser den Interruptanforderungscode in binärer Form auf die niedrigsten drei Adressleitungen *A01* ...*A03* des Adreßbusses. Anschließend folgt der gleiche Ablauf wie bei einem üblichen Lesezyklus auf dem Daten-Transfer-Bus:
Durch die Aktivierung des Adresstrobes wird das Anliegen des Interruptcodes angezeigt. Die inaktive Leitung $\overline{WRITE}$ zeigt einen Lesezyklus an und durch das aktive $\overline{DS0}$-Signal wird dem Interruptanfrager angezeigt, daß das niedrigstwertige Byte auf dem Datenbus gelesen werden soll. Nun wird der Interruptanforderer über die anliegende Adresse adressiert. Er kann erst dann die Vektornummer auf den Datenbus legen, wenn er auch das $\overline{IACKIN}$-Signal empfangen hat. Dieses Signal gelangt über eine Daisy-Chain-Verbindung an den Interruptanforderer. Nachdem der Interrupt-Auslöser seine geforderte Vektornummer auf den Datenbus gelegt hat, aktiviert er das Signal $\overline{DTACK}$. Nun kann der Interrupthandler seine Vektornummer lesen und die Interrupt-Service-Routine starten.

B.4.1 Die Hilfs- und Versorgungssignale

Der *VME*-Bus verfügt über spezielle unabhängige Signale für die Synchronisation, die Initialisierung, den Systemtest und die Fehlerdiagnose. Während die Hilfsignale für einen normalen Datentransfer nicht notwendig sind, leisten sie doch einen wesentlichen Beitrag zur Gesamtleitungsfähigkeit dieses modernen Bussystems. Im einzelnen stellt der *VME*-Bus die folgenden Signalleitungen zur Verfügung:

Systemtakt (SYSCLK) : Dieses ist ein vom Prozessor unabhängiges 16-MHz-Taktsignal, welches in keiner festen Phasenbeziehung zu anderen *VME*-Bus-Signalen steht. Dieses Signal kann beispielsweise für Zähler- und Synchronisationsfunktionen herangezogen werden.

System-Reset (SYSRESET) : Über diese Leitung können alle an dem *VME*-Bus angeschlossenen Module in einen definierten Anfangszustand gebracht werden. Dieses Signal kann sowohl durch einen Taster an der Bedienfront eines Moduls oder automatisch beim Einschalten der Stromversorgung durch ein sogenanntes *Power-Monitor-Modul* geschehen.

System-Test-Leitung (SYSFAIL) : Mittels dieser Leitung kann ein Fehler im System gemeldet werden. Ein Beispiel hierfür sind z.B. Module, welche nach der Aktivierung des System-Resets einen Selbsttest auf der Platine durchführen. Falls sich bei diesem Test ein Fehler zeigt, so wird die System-Test-Leitung nicht deaktiviert und der System-Controller muß entsprechende Maßnahmen ergreifen.

Spannungszusammenbruch (ACFAIL) : Hiermit wird dem System ein Spannungseinbruch gemeldet. Für einen solchen Fall schreiben die *VME*-Bus-Spezifikationen Netzteile vor, welche für den ordnungsgemäßen Abbruch der laufenden Aktivitäten mindestens 4 ms zulassen (siehe auch Abbildung 41).

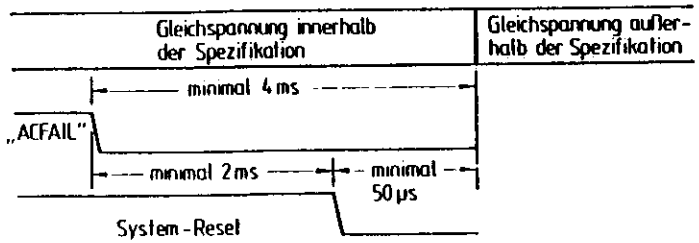


Abbildung 41: Abschaltsequenz beim VME-Bus

Stromversorgung : Bei dem *VME* -Bus existieren außer den Massleitungen vier Verbindungen, welche für die Stomversorgung zuständig sind. Es stehen die folgenden Spannungen zur Verfügung: +5V, +12V, -12V und +5V Notstrom.

B.5 Die Module CC1 und EXPU

Bei dem von mir verwendeten Crate-Controller handelt es sich um den bereits veralteten Typ CC1. Da dieser nur in der Lage ist 512 k Byte im DMA-Transfer zu adressieren, wurde der Adressraum bei neueren Crate-Controllern vergrößert. Während der Crate-Controller hauptsächlich für die Arbitration und als Bustreiber dient, befinden sich im EXPU-Interface vier 16-Bit Register mit denen der Datentransfer gesteuert wird. Diese Register tragen die Bezeichnungen **Status-**, **User-**, **LDA-** und **Pagetable-Register**, wie man in der Abbildung 42 sehen kann.

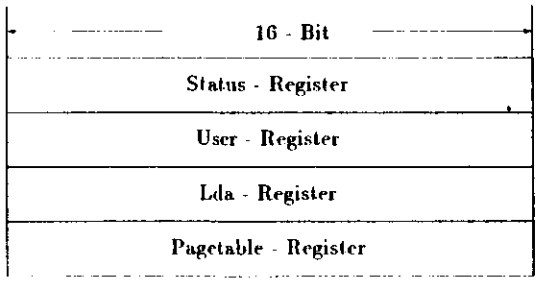


Abbildung 42: Die vier EXPU-Register

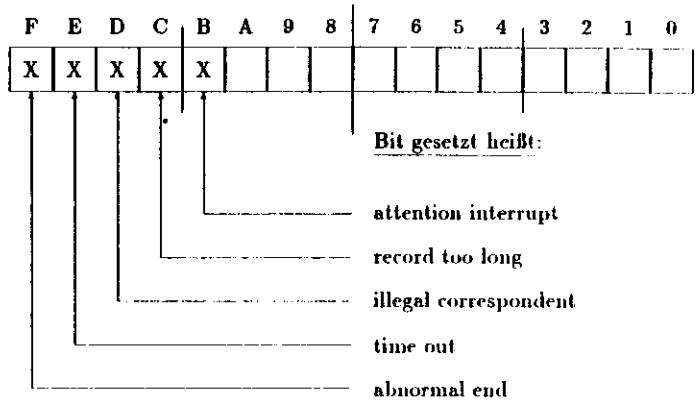


Abbildung 43: Bits im Statusregister

Das Status-Register In diesem Register stehen die Error-Bits, welche über den Erfolg oder den Mißerfolg des Datentransfers Auskunft geben (siehe Abbildung 43). Wenn in allen Bits eine Null steht, so bedeutet dieses, daß der Datentransfer ordnungsgemäß abgelaufen ist. Dementsprechend bedeutet ein gesetztes Bit, daß es bei der Übertragung zu einem Fehler gekommen ist. Dabei kann es auch passieren, daß Kombinationen von diesen Error-Bits auftreten.
Als z.B. einmal die Kennummer, welche auf dem EXPU-Modul über Schalter eingestellt wird, falsch gewählt worden war, ordnete die *IBM* dem Modul einen anderem EXP-Job zu. Der EXP-Job ist das auf der IBM laufende Programm, welches die für die Datentransferprogramme notwendige Prozeßumgebung schafft. Deshalb konnte in diesem Fall keine Kommunikation mit der *IBM* aufgenommen werden und in dem Status-Register stand die Zahl $B000_{16}$. Nun gilt aber :

$$B000_{16} = 8000_{16} + 4000_{16}$$

Nach dem Datentransfer waren also zwei Fehlerbits gesetzt worden. Dabei zeigt das höherwertige Bit an, daß die Datenübertragung nicht normal beendet wurde (abnormal end error). Das niederwertige Bit zeigt in diesem Fall an, daß der Kommunikationspartner nicht in der vorgegebenen Zeit geantwortet hat (timeout error).
Schließlich sollte dieses Register grundsätzlich vor der Aufnahme einer Datenübertragung ausgelesen werden, um einen eventuell noch nicht beantworteten (sogenannten hängenden) Interrupt zu löschen.

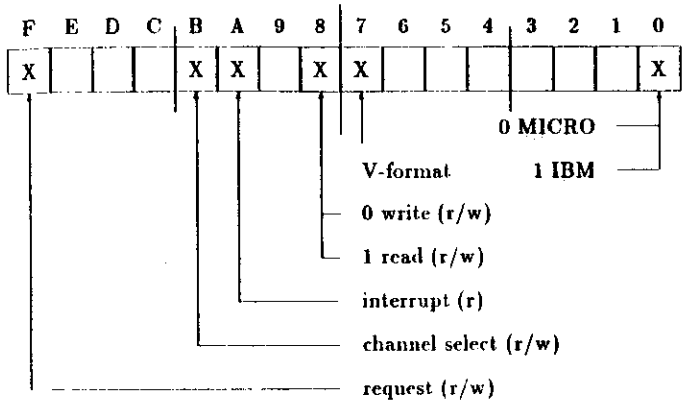


Abbildung 44: Bits im Userregister

Das User-Register Dieses Register enthält die Bits, welche zur Steuerung des Datentransfers notwendig sind, etwa um den Datentransfer zu starten (siehe Abbildung 44).

Bit Nr.0 Wenn dieses Bit eine Null enthält, so findet die Kommunikation nur mit dem sogenannten MICRO-PADAC-Modul statt. Dieses ist ein PADAC-Bus Rechner auf er Basis des Mikroprozessors TMS 9900 von Texas Instruments. Bei dem Versuch einen Kleinrechner über das PADAC-Netz an die IBM anzuschliessen, ist es sinnvoll zuerst mit dem MICRO zu kommunizieren. Auf diesem Rechner läuft nämlich ein Spiegelprogramm, d.h. wenn man Daten zu diesem Rechner schreibt und anschließend gleich wieder einen Lesebefehl gibt, so erhält man, falls man alles richtig gemacht hat, die Daten wieder zurück.
Erst wenn diese Art des Datentransfers einwandfrei funktioniert, sollte man zum nächsten Schritt übergehen und versuchen mit der IBM zu kommunizieren. Es ist dabei auch während des weiteren Betriebes der Datenverbindung sinnvoll, daß man noch einen Transfertest mit dem Spiegelprogramm des MICRO durchführen kann. Falls später einmal eine Störung auftritt, so verfügt man hiermit über ein leistungsfähiges Mittel zur Fehlersuche.

Bit Nr.7 Wenn dieses Bit gesetzt wird, so findet die Kommunikation im sogenannten V-Format statt. Bei diesem Format steht die Anzahl der zu übertragenden 16-Bit Worte (Word Count) nicht in der Adressentafel, sondern am Anfang der Datenblöcke. Dieses Format wird bei den von mir geschriebenen Programmen nicht verwendet.

Bedeutung :	F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
WRITEREQ :	1	0	0	0	1	0	0	0	0	0	0	0	0	0	0	X
READREQ :	1	0	0	0	1	0	0	1	0	0	0	0	0	0	0	X

X: je nach dem an der Kommunikation beteiligten Rechner 0 oder 1

Tabelle 4: Bits für einen Read- oder einen Writerequest

Bit Nr.8 Mit diesem Bit wird festgelegt, ob die Daten vom Kleinrechner zur IBM (Bit=0) oder von der IBM zum Kleinrechner (Bit=1) gesendet werden.

Bit Nr.A_{16} Über dieses Bit kann von der IBM aus auf dem Kleinrechner ein Interrupt ausgelöst werden. Entsprechend ist dieses Bit vom Kleinrechner aus nur zu lesen und kann nicht beschrieben werden (read only). Dieses Bit hat damit nur einen Sinn, wenn der Kleinrechner keine Interrupts versteht. In solch einem Fall kann der Kleinrechner, durch die Abfrage dieses Interrupt-Bits (sogenanntes *Polling*) feststellen, ob der Datentransfer beendet ist.

Bit Nr.B_{16} Durch das Setzen dieses Bits wird das EXPU-Modul angewiesen auf den Speicherbereich einen DMA zu machen und die ersten Daten in sein internes FIFO zu bringen.

Bit Nr.F_{16} Dadurch, daß dieses Bit gesetzt wird, startet man die Datenübertragung vom FIFO des EXPUs zum Kontrollrechner (MICRO-Modul), welcher die Daten dann zur *IBM* weiterleitet. Dieses Bit, welches man zweckmäßigerweise zusammen mit dem Bit Nr.B_{16} setzt, darf nicht vor diesem gesetzt werden.
In der Tabelle 4 sind die Bitkombinationen angegeben, welche einen sogenannten READ- oder WRITE-REQUEST auslösen. Dieses sind die Aufforderungen einen Datensatz von dem Kleinrechner zu einem anderen Rechner oder von dem anderen Rechner zu dem Kleinrechner zu schicken.

Das Lda-Register In dieses Register muß beim Senden die Basisadresse für die zu übertragenden Daten geschrieben werden. Beim Empfangen von Daten hat hier die Adresse zu stehen, wo die von der IBM oder dem MICRO gesendeten Daten abgelegt werden. Wie die Struktur der Daten aussieht, ist in dem Bild 45 dargestellt. Die Adresse in dem Lda-Register weist auf den Anfang des Datenblocks, welcher die Adressen der Daten enthält. Diesen Datenblock, welcher aus einer Gruppe von zwei 16-Bit Worten besteht, bezeichne ich deshalb von nun an als Adressenblock. In dem ersten 16-Bit Wort steht eine Adresse, welche ihrerseits auf den eigentlichen Datenblock hinweist. Das zweite 16-Bit Wort enthält den sogenannten Word-Count (WC). Dieses in die Anzahl der 16-Bit Worte des Datenblockes, auf den die Adresse im ersten Datenwort verweist.
Der Adressenblock kann aus einer ganzen Reihe von diesen Verweisen (Adresse und Word-Count) auf Datenblöcke bestehen. Der Schluß des Adressenblockes wird aus zwei 16-Bit Worten, welche nur Nullen enthalten, gebildet.
Der Datenblock, welcher die vom Kleinrechner zur IBM zu sendenen Daten enthält, beginnt

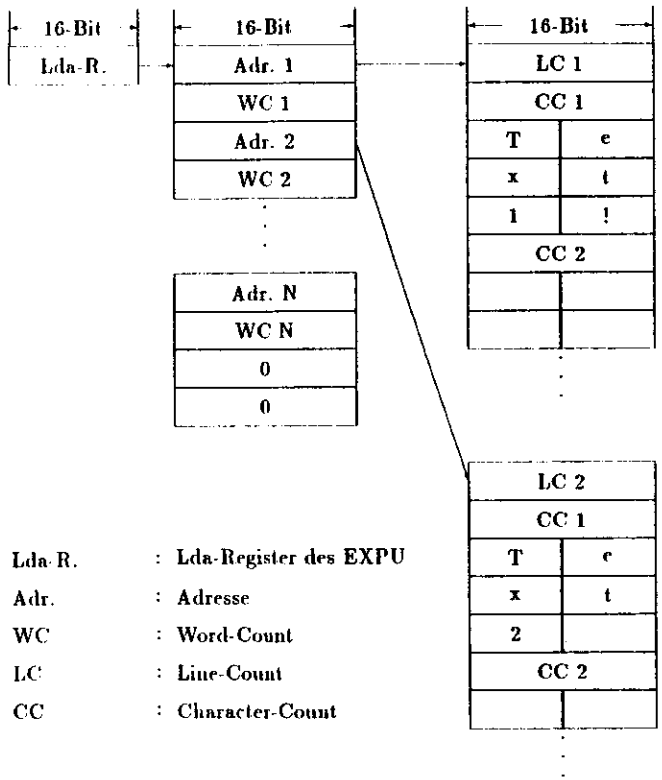


Abbildung 45: Datenformat beim Senden zur *IBM*

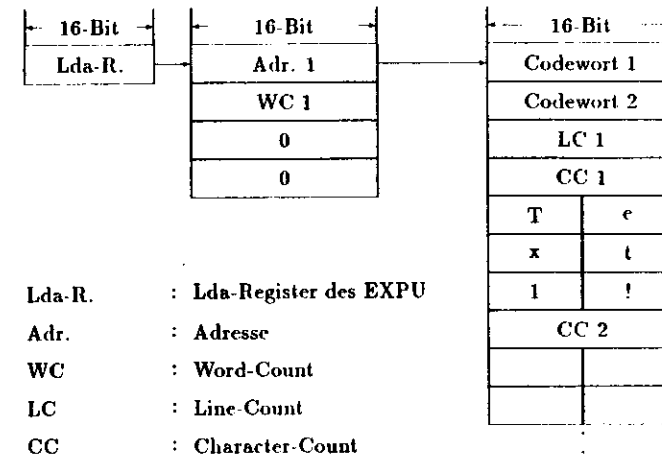


Abbildung 46: Datenformat beim Empfangen von IBM-Daten

mit einem 16-Bit Wort. In diesem steht die Anzahl der in diesem Datenblock vorhandenen Zeilen (sogenannter Line-Count). Nach diesem Line-Count folgen die Blöcke, welche sowohl die Anzahl der Zeichen pro Zeile, als auch die Zeichen der Zeile selbst enthalten. Die Anzahl der pro Zeile vorhandenen Zeichen steht als der sogenannte Character-Count (CC), was ein ebenfalls 16-Bit langes Datenwort ist, den eigentlichen Textzeichen voran. Nach dem Character-Count folgt also der Text, wobei immer zwei Zeichen in einem 16-Bit Wort stehen. Ist die Anzahl der Zeichen in einer Zeile ungerade, so steht in dem letzten 16-Bit Datenwort dieser Zeile nur ein Zeichen.

Der einzige Unterschied zwischen den Datenformaten beim Senden und beim Empfangen besteht darin, daß beim Empfangen den Datenblöcken zwei 16-Bit Codeworte vorangestellt sind (siehe Abb. 46). Diese Codeworte stehen also noch vor dem Line-Count (LC), weshalb in diesem Fall die Adresse im Adressenblock auch auf den Platz des ersten Codewortes zwei Worte vor dem Linecount (LC) verweist. Mit diesen Codeworten werden zusätzliche Informationen übertragen, welche über den Zustand der Datenverbindung Auskunft geben.

Auch bei dem zuletzt beschriebenen Datenformat ist es möglich, daß die Adressentabelle auf eine Reihe von Datenblöcken verweist, wie es beim Senden der Fall war. Beim Empfangen ist es jedoch wegen der unübersichtlichen Struktur in der Regel nicht sinnvoll diese Option verwenden.

Falls die Kommunikation mit dem MICRO-Modul (Spiegelmodus) abläuft, so ändert sich das Format zwar nicht beim Senden vom Kleinrechner zum MICRO (siehe Abbildung 45), aber beim Empfangen der gespiegelten Daten stehen diese ohne Line-Count (LC) und Character-

Datenbits																Anmerkungen:	
F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0		
R	R	R	R	R	R	R	R	N	N	V	V	V	V	V	V	Datenwort	Nr.
0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0401_{16}	1
0	0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0802_{16}	2
0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	1	$0C03_{16}$	3
0	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0	1004_{16}	4

Tabelle 5: Die ersten vier Datenwörter (von 64), um den Pagetable 1:1 zu initialisieren.

Count (CC) direkt ab der Stelle, wo sonst der Line-Count (LC) stehen würde.

Das Pagetable-Register Das Pagetable-Register ist ursprünglich für die Zusammenarbeit zwischen den PADAC-Modulen und NORD10-Rechnern entwickelt worden. Diese NORD-Rechner benutzen für ihre Speicherverwaltung das sogenannte *PAGING*. Bei diesem Verfahren bekommt jeder ablaufende Prozeß eine bestimmte Anzahl von 1 k Byte großen Seiten (*Pages*) zugeteilt. Noch nicht oder nicht mehr benötigte Seiten eines Prozesses werden in den Plattenspeicher ausgelagert und bei Bedarf vom Betriebsystem wieder in den Hauptspeicher transferiert. Mehrere solche Prozesse teilen sich dann einen gemeinsamen Speicher von z.B. 256 k Byte. Während die Adressen eines solchen Prozesses mit je 16 Adressleitungen für Daten und Programme kodiert werden können (virtuelle Adresse), benötigt man, um den gemeinsamen Speicher zu adressieren, 18 Adressleitungen (realer Adressraum). Zusätzlich werden die Adressräume in Einheiten zu je 1024 Byte eingeteilt, d.h. für die Adressierung innerhalb einer solchen Einheit (Page) werden die unteren 10 Adressleitungen (A0...A9) benutzt. Die verbleibenden oberen 6 Adressbits des virtuellen Adressraumes werden nun mittels einer Tabelle dem sogenannten Pagetable auf die 8 oberen Bits des realen Adressraumes abgebildet (siehe Abbildung 48).
Mittels dieser Memory-Management Methode dem sogenannten Paging ist es möglich, daß sehr viele Prozesse ihre gerade benutzten Speicherseiten im gemeinsamen Speicher halten können (siehe Abbildung 47). Dabei muß natürlich darauf geachtet werden, daß die Pagetables für die einzelnen Prozesse so eingestellt sind, daß nicht zwei Prozesse dieselbe Seite im Speicher benutzen.
Wenn die PADAC-Module mit den Rechnern zusammenarbeiten, welche diese Art des Memory-Managements verwenden, braucht nur der Pagetable des EXPU-Moduls mit den aktuellen Adressumrechnungen für seinen Prozeß aufgefüllt werden. Sobald dieses geschehen ist, sind die PADAC-Module vollständig in das Speichersystem des betreffenden Rechners integriert. Dabei müssen die *Pages* welche sich im Hauptspeicher befinden vor dem Auslagern auf eine Festplatte geschützt werden. So gibt es bei dem Betriebsystem *SINTRAN* ein Befehl, welcher das Auslagern der Seiten auf die Festplatte verhindert.
Auch wenn man Rechner mit anders aufgebauten Speicherverwaltungskonzepten und einer anderen Zahl von Adressleitungen an die PADAC-Module anschliessen will z.B. *VME* -Rechner, ist es wichtig, die Funktion des Pagetables zu kennen. Schließlich muß der Pagetable selbst

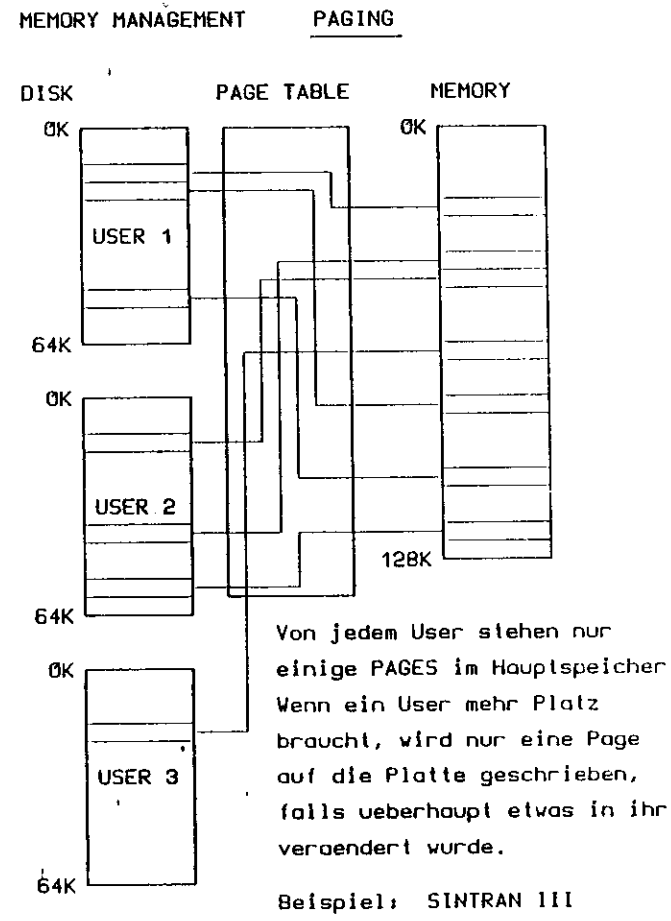


Abbildung 47: Speicherverwaltung bei einem Rechner mit virtuellen Speicher

Die Bits 6 & 7 (NN) werden nicht benutzt.

Abbildung 48: Bits im Pagetable-Register

dann initialisiert werden, wenn die Adressen 1:1 übersetzt werden sollen, d.h. wenn die Adressbits (A0...A15) von virtueller und realer Adresse immer gleich sein sollen.
Das Auffüllen des Pagetables geschieht, indem man in das Pagetable-Register 16-Bit Worte schreibt, wie es aus dem Bild 48 zu ersehen ist. In der Tabelle 5 ist angegeben, wie man den Pagetable laden muß, wenn die Adressen 1:1 übersetzt werden sollen.

B.6 Targetebene

In dem Bild 49 sieht man das Blockdiagramm eines Targetrechners. Ein Targetrechner besteht aus zwei verlängerten Doppeleuropakarten, nämlich der CPU-Karte und der Floating-Point Einheit.
Bei der CPU-Karte handelt es sich um eine DSSECPUA1 [4] Rechnerkarte von DATA-SUD. Sie besitzt die folgenden Eigenschaften:

- 68000-er Mikroprozessor mit einer Taktfrequenz bis maximal 12.5 MHz
- 256 kByte dynamisches RAM (bis 8 MHz ohne Wartezyklen)
- V24-Terminalschnittstelle
- volle Unterstützung des *VME* - und des *VMX* -Busses
- bis zu 64 kByte EPROM auf der Platine
- arithmetischer Coprozessor NS 16081 (er wird für die selten auftretenden Divisionen gebraucht).

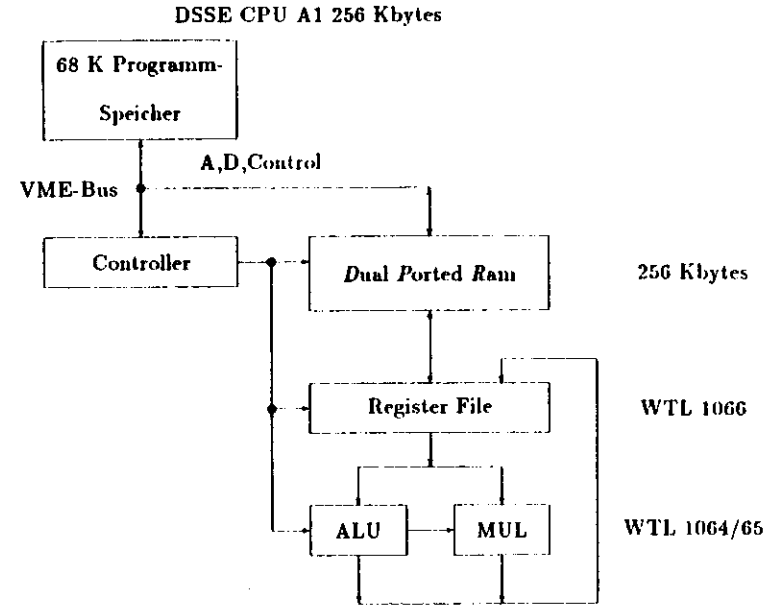


Abbildung 49: Prinzipieller Aufbau eines Targets

- zusätzliche parallele Schnittstelle an der Frontseite (sie wird für den Anschluß an die Arbitration-Einheit benutzt)

Die doppelt genauen Fließkommaberechnungen (Addition, Subtraktion Multiplikation) werden von der Floating-Point Einheit durchgeführt. Bei der Ausführung dieser Befehle muß der 68000-er Prozessor dem Controller der *FPU* nur mitteilen, welche Zahlen sie mit welcher Operation zu verrechnen hat (siehe auch C.1.3).
Die Architektur der *FPU* wird durch die Anforderungen der Weitek-Chips bestimmt. So hat der interne Bus, welcher auf das Dual-Ported Memory zugreift, eine Breite von 64-Bit, während der Zugang von der *VME* Seite nur nur eine Datenbreite von 16-Bit hat. Dank des breiten internen Busses benötigt diese Karte pro Variable nur einen Taktzyklus, um diese zwischen den Registerfile-Chips [13] und dem Dual ported RAM (*DPR*) hin und her zu schieben. Bei den Registerfile-Chips mit der Bezeichnung WTL 1066 handelt es sich um schnelle Zwischenspeicher mit 32 internen Registern je 64-Bit. Sie sind auf die speziellen Anforderungen der Arithmetik-Chips ausgelegt, für welche sie als Zwischenspeicher dienen können und

besitzen die folgenden Leistungsmerkmale:

- 3 schnelle Ein- und 3 schnelle Ausgabekanäle
- Datenein- und Datenausgabe ist alle 60 ns möglich
- Standardgehäuse (144 Pin-Grid)

Mittels dieser Chips können der Multiplizierer (WTL 1065) und die ALU (*Artihmetical and Logical Unit* WTL 1064) binnen zweier Taktzyklen je zwei Operanten in sich hineinladen. Neben dieser hohen Geschwindigkeit beim Zugriff auf die Registerfile-Chips besitzen die beiden Arithmetik-Chips (WTL 1064 & 1065) die folgenden Leistungsmerkmale:

- die Befehle ADD/SUB/MUL/CMP und Absolutwert
- Diese Operationen können im 32- oder 64-Bit IEEE-Format durchgeführt werden.
- Im Pipelinemodus (nicht verwendet) können alle 60 ns 64-Bit Daten eingelesen und 32-Bit berechnete Daten wieder ausgegeben werden.
- Im normalen Modus können alle Operationen außer den Multiplikationen binnen 360 ns durchgeführt werden (Multiplikation 600 ns).
- Beide Chips befinden sich in einem 144 Pin-Grid Gehäuse.

B.7 Controllerebene

Durch die Module in dieser Ebene werden die Targetrechner gesteuert und die Daten zwischen der Grafikkarte, der PCS, der IBM sowie den Targets ausgetauscht.
Physikalisch gesehen besteht diese Ebene aus einem *VME* - Crate mit 6 Doppeleuropakarten und zwar der **Grafik-**, der **Controller-**, der **Arbitration-** und der **Mailboxkarte**, sowie dem **Q-Bus/*VME*** -und dem **PADAC-/*VME*-Interface**.

Graphikkarte Sie wird hauptsächlich dazu benutzt, um während eines Rechenvorganges Daten von den Targets graphisch auszugeben. Damit ist es möglich, bereits während einer Simulation zu entscheiden, ob sich eine weitere Aufbereitung der Daten lohnt oder nicht. Zusätzlich hat es sich als sinnvoll erwiesen, jeweils kurz den Zustand der Targets über die Grafik auszugeben, so daß man mit einem Blick über den Zustand des gesamten Systems unterrichtet ist.

Technische Daten Die verwendete *VME*-Bus Grafikkarte vom Typ DSSE 512CHROM-A8 stammt ebenso wie die verwendeten *VME*-*VMX* -Prozessorkarten von der Firma DATA SUD [5].
Diese Grafikkarte ist in der Lage 512 × 512 Punkte mit 8 verschiedenen Farben darzustellen. Sie wird dabei von einem Grafikprozessor [3] unterstützt, welcher eine sehr schnelle Vektorgrafik erlaubt. Zusätzlich befindet sich auf dieser Karte ein Softwarepaket, welches nicht nur eine Punkt- und Vektorgrafik, sondern auch die komfortable Darstellung des ASCII-Zeichensatzes gestattet.

[3] Typ EF9365 GDP von Thompson-EFCIS

Eine effektive Ein- und Ausgabe der Daten über den *VME* -Bus wird über einen intelligenten Peripherie-Baustein kontrolliert, wobei als eigentliche *VME* -Bus Schnittstelle ein 128 Byte langes *DPR* [4] dient, dessen Basisadresse sich über Jumper auf der Karte einstellen läßt.

Controllerkarte Der Controller ist eine *DATA-SUD* -Karte (vom Typ DSSECPUA1 [4]), wie sie auch bei den Targetmodulen verwendet wird, nur daß diese Karte nicht verlängert worden ist. Sie koordiniert den gesamten Datentransfer in dieser Ebene, wenn man von den *DMA* - Zugriffen durch die PADAC-Module einmal absieht. Aber selbst die Einleitung dieses Datentransfers geschieht durch den Controller.

Arbitration-Einheit Mittels dieses Einschubes ist es möglich, die einzelen Targetmodule zu selektieren. Dieses ist z.B. notwendig, um einem Modul den Zugang zu der Mailbox zu erlauben, oder um ein Target durch ein RESET-Signal wieder in seinen Ausgangszustand zu bringen. Üblicherweise löst man ein solches Problem durch Interrupts verschiedener Priorität, aber in diesem Fall war das nicht möglich, weil die Targets mit der Controllerebene durch den *VMX* -Bus verbunden sind und dieser Bus keine Interrupts gestattet.

Mailbox-Karte Bei der Mailbox handelt es sich um einen Speicher, welcher sowohl einen *VME* - als auch einen *VMX* -Bus Anschluß besitzt. Dabei ist der Zugriff auf diesen Speicher von beiden Bussen aus möglich. Eine Auswahllogik sorgt dafür, daß sobald ein Bus auf diesen Speicher zugreift der Zugriff für den anderen Bus gesperrt ist. Solche Speicher mit zwei Zugängen werden auch als DPR (*Dual Ported Ram* (engl.) = Schreib- und Lesespeicher mit zwei Zugängen) bezeichnet. Sie werden dazu verwendet, zwei Bereiche (hier *VME* -und *VMX* -Bus) schnell und effektiv miteinander zu verbinden. Diese Speicher stellen somit eine Form von Interface dar, welche den Vorteil haben das ein Puffer, nämlich der Speicherbereich, zur Verfügung steht.

Q-Bus/*VME*-Interface Hierbei handelt es sich um ein paralleles Interface, welches die Signale zwischen dem Q- und dem *VME* -Bus so überträgt, daß man auf den beiden Bussen eine 16-Bit breite Speicherstelle für den Datentransfer zur Verfügung hat. In dem *UNIX* -Betriebsystem kann man dieses allerdings erst beim näheren Hinsehen feststellen, weil die parallele Schnittstelle durch ein sogenanntes Device- Treiber Programm in das Ein-/Ausgabekonzept von *UNIX* eingebunden ist.
Während die Q-Bus Seite des Interfaces eine parallele Schnittstelle (Typ DRV11 [6]) ist, wurde die *VME* -Seite des Interfaces von der Gruppe F56 im DESY entwickelt.

PADAC/*VME*-Interface Die gesamte Logik dieses Interfaces befindet sich auf einer *VME* -Doppeleuropakarte, welche mit dem *PADAC* -Bus über zwei 96-polige Kabel verbunden wird. Dieses Interface setzt die Signale des PADAC-Busses so um, daß man von der *VME* -Seite die vier 16-Bit breiten Steuerregister sieht. Die Adresse wo sich diese Register befinden sollen, läßt sich mittels Drahtbrücken auf der Interfacekarte anwählen, genauso wie der Bereich auf den ein DMA-Zugriff gemacht werden kann. Für den DMA-Zugriff muß älteren PADAC-Crate Controllern (z.B. CC1) ein OFFSET wählbar sein, weil diese nur einen Adressraum von 512 k Byte haben.

[4] *DPR* (*Dual Ported Ram*)

An elektronischen Bausteinen befinden sich auf der Karte einige PALs (programmable array logic), um die Signale zu konvertieren, sowie Treiberbausteine, welche die Signale auf die jeweiligen Busse geben. Entwickelt wurde diese Karte von K.Rehlich (DESY F52).

C Details zur Systemsoftware

C.1 Systemsoftware auf den Targets

C.1.1 Steuerungsteil

Weil auf den Targets nur von anderen Rechnern aufbereitete Benutzer-Objektprogramme laufen sollen und Betriebsysteme etwas *CPU* -Zeit für ihre eigene Verwaltung benötigen, wurde in dieser Ebene auf ein Betriebsystem im eigentlichen Sinne verzichtet.
Das einzige was im EPROM-Speicher (siehe Bild 50) vorhanden ist, ist ein DSSEbug-Monitorprogramm, sowie eine kleine Assemblerroutine, mittels der ein Boot-Programm geladen werden kann. Dieses Boot-Programm, es handelt sich hierbei um den Targetteil von *oper.c* aus */usr/bop/sys*, wird über den Befehl boot von dem Controller aus über die Mailbox in das Target geladen.
Sobald dieses geschehen ist, akzeptiert das Target verschiedene Befehle, welche von dem Controller aus durch das Programm *oper.c* gegeben werden können:

push PRGM: Lädt ein Programm aus der Mailbox in den Programmspeicher der 68k-*CPU* des Targetrechners.

push DATA: Lädt einen Datensatz aus der Mailbox in den statischen Speicher der Floating-Point Einheit.

Start: Startet die Ausführung eines eventuell im Speicher des Targets vorhandenen Programmes.

Nach einem Datentransfer wird der augenblickliche Zustand des Targets in die Mailbox eingetragen, so daß der Controller über die Situation des Targets informiert ist.

C.1.2 Befehle in Anwenderprogrammen

Wegen der unüblichen Art der Ein- und Ausgabe bei den Targets (über die Mailbox), mußten hier neue Ein- und Ausgabebefehle geschaffen werden. Sie stehen in dem Programm *mail.c* in der Bibliothek */usr/bop/race*.
Alle diese Befehle wurden in der Sprache *C* geschrieben, anschließend wurde für sie ein FORTRAN-Interface (ebenfalls in *mail.c*) geschaffen. Somit kann man diese Befehle sowohl in *C* als auch in FORTRAN verwenden. In FORTRAN haben sie die folgende Form:

ifd=iopen(100,100) Dieser Befehl öffnet den Datentransfer mit dem Controller über die Mailbox. Dabei besagt die erste 100, daß die Verbindung mit dem Rechner 100 (siehe *addr.h* in */usr/bop/include*) eröffnet wird. Die zweite 100 besagt, daß der Datentransfer zum Schreiben eröffnet wird (siehe ebenfalls *addr.h*). Vom Target aus ist es zwar nur möglich, eine Verbindung zum Controller zu eröffnen, aber diese Kommunikationsbefehle sind auf der Controller- und der Targetebene identisch, so daß immer die allgemeine Form des Befehles angegeben werden muß.

iwrite(ifd,var,n) Dieser Befehl schreibt die Variable oder das Feld *var*, welches aus n-Bytes besteht, in die Mailbox für den Controller.

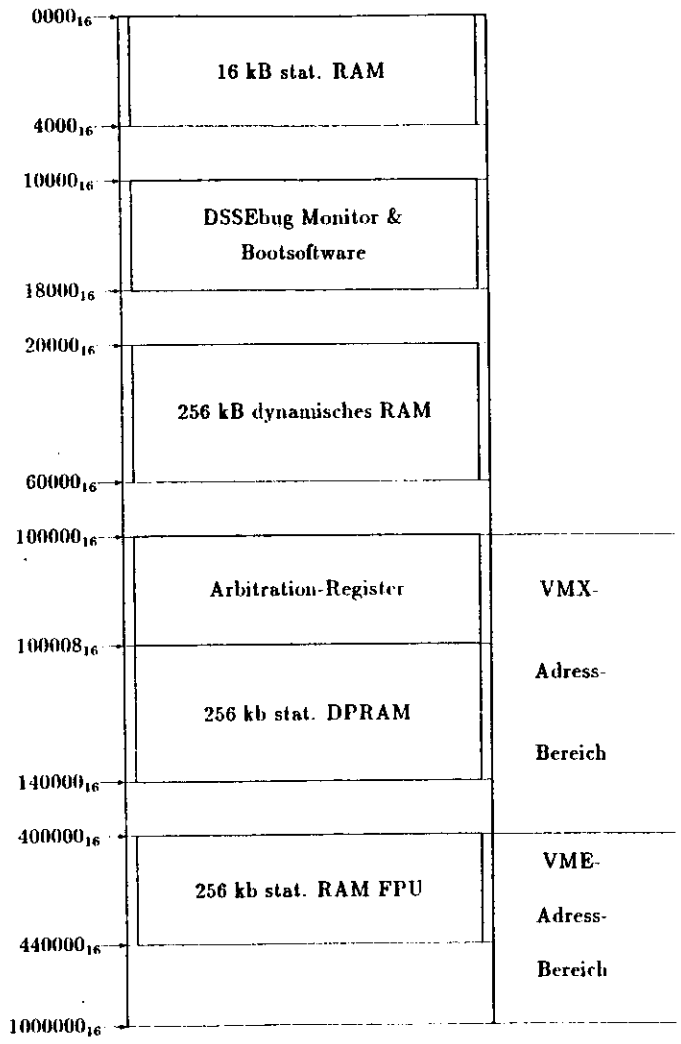


Abbildung 50: Adressraum eines Targets

iread(ifd,var) Durch diesen Befehl wird aus der Mailbox die Variable oder das Feld *var* eingelesen.

iclose(ifd) Dieser Befehl schließt die Verbindung mit dem Controller ab.

Bei diesen Befehlen ist zu bedenken, daß bei ihrer Ausführung durch ein Target auf dem Controller ein Programm laufen muß, welches diesen Befehlen zuarbeitet. Wenn also ein Programm auf einem Target den Befehl *iwrite* ausführt, muß auf dem Controller durch ein entsprechendes Programm demnächst der Befehl iread durchgeführt werden.

C.1.3 Einbindung der Floating-Point Befehle

Damit die auf der *PCS* kompilierten Programme die Floating-Point Einheit ansprechen, müssen die entsprechenden Befehle in den Programmen für die Targets ausgetauscht werden. Dieses geschieht, indem das kompilierte Programm auf der *PCS* disassembliert wird (durch *Cdao.c* oder *F77dao.c* in der Bibliothek */usr/bop/sys*). Anschließend tauscht ein Textfilter (*pas* in der Bibliothek */usr/bop/sys*) die Software-Floating-Point Befehle gegen entsprechende MOVE-Befehle des 68000-er aus. Mittels dieser MOVE-Befehle wird der Controller in der Floating-Point Einheit angewiesen, bestimmte Befehle auf der Karte ablaufen zu lassen. Diese Kodierung durch die MOVE-Befehle ist in der Arbeit von P.Wilhelm [10] nachzulesen.
Anschliessend wird das Textfile durch den Assembler als *UNIX* -Befehl vorliegenden Assembler (*as*) wieder in einen ausführbaren Code umgewandelt. Dieser wird dann durch ein weiteres Programm (*T.srec* oder *H.srec* aus der Bibliothek */usr/bop/sys*) in ein über das Q-Bus-VME Interface zu transportierendes Format (S-Records) gebracht.

C.2 Systemsoftware in der Mailbox

Die Mailbox hat die Aufgabe, für einen schnellen Datenaustausch zwischen der Controller- und der Targetebene zu sorgen. Zwischen diesen beiden Ebenen werden aber nicht nur Programme, sondern sehr viele verschiedene Arten von Daten (Steueranweisungen, Quittierungsmeldungen usw.) ausgetauscht. Um diese Datenströme zu kanalisieren wurde von A.Deuter eine *C* -Struktur (Typ *struc*) geschaffen.
Diese Struktur steht im File *box.h* in der Library */usr/bop/include*. Sie wird in alle Programme eingebunden, welche über die Mailbox mit anderen Programmen kommunizieren. Ihre Anfangsadresse wird dabei in beiden Ebenen so gesetzt, daß der Anfang der Struktur mit dem Anfang der Mailbox übereinstimmt. Diese Struktur läßt sich in fünf verschiedene Bereiche untergliedern :

Register der Arbitration-Box Diese wurden der Einfachheit halber in die Struktur box.h mit hineingezogen, obwohl sie mit der eigentlichen Mailbox nichts zu tun haben und ausschließlich für die Steuerung der Targets dienen. Selbstverständlich lassen sich diese Register auch nur von der Controllerebene aus ansprechen, weil sich die Arbitration-Box in dem *VME* -Bus dieser Ebene befindet.

Boot-Register Das Booten eines Targets ist der Vorgang, bei dem das Betriebsystem für diesen Rechner aus der Mailbox heraus geladen wird. Diese Register enthalten die Adressen dieses Betriebsystems in der Mailbox, sowie die Zieladressen in den Targets.

Mail- und Kontrollregister Neben der Nummer des Targets mit dem momentan kommuniziert wird, befinden sich hier die Register, die für das Protokoll des Datenaustausches zwischen dem Controller und den einzelnen Targets notwendig sind. Also die Register für die Quittierungsmeldungen und die Adressen für die auszutauschenden Datenfelder.

Target-Directory In diesem Bereich befinden sich die Register, wo für jedes Target getrennt die Anfangs- und Endadressen der in ihm befindlichen Daten stehen. Die Daten lassen die dabei in drei verschiedene Segmente untergliedern:
Da ist zuerst das Textsegment, in welchem die Daten für das eigentliche ausführbare Programm stehen.
Das zweite ist das sogenannte Datensegment, in welchem alle Variabeln stehen, welche nicht zum Common-Block gehören. Man nennt diese Art von Variabeln auch manchmal dynamisch, weil sie nach dem Verlassen einer Routine ihren Wert verlieren.
In dem letzten (dritten) Segment stehen schließlich die Variabeln aus dem Common-Block. Weil diese Variabeln ihren Wert beim Verlassen einer Routine nicht verlieren, werden sie auch als statische Variabeln bezeichnet (C-Variabeltyp *static*). In dieses Common-Segment werden u.a. die Maschinenparameter für die Racetrackingprogramme geladen.

Der eigentliche Mailbox-Bereich Dieses ist einfach ein riesiges eindimensionales Feld, welches den restlichen Bereich des Dual Ported Ram der Mailboxkarte umfaßt. Bei dem Datenaustausch zwischen den beiden Ebenen werden die Daten in diesem Bereich vorübergehend abgelegt bis sie abgeholt werden.

C.3 Systemsoftware auf dem Controller

Auf dem Controller läuft in erster Linie das Programm *oper.c*. Dadurch, daß in dieses Programm die Funktionen der Funktionssammlung *mail.c* eingebunden worden sind, ist es in der Lage, vom Controller aus Datentransfers zu den Targets und zur PCS durchzuführen. Ferner können die Targets durch das Programm *oper.c* gesteuert werden. Für diese beiden Anwendungsgebiete stellt das Programm *oper.c* die folgenden Befehle zur Verfügung :

push PRGM: Durch diesen Befehl wird ein Programm aus der Mailbox in den Programmspeicher der 68k-*CPU* geladen.

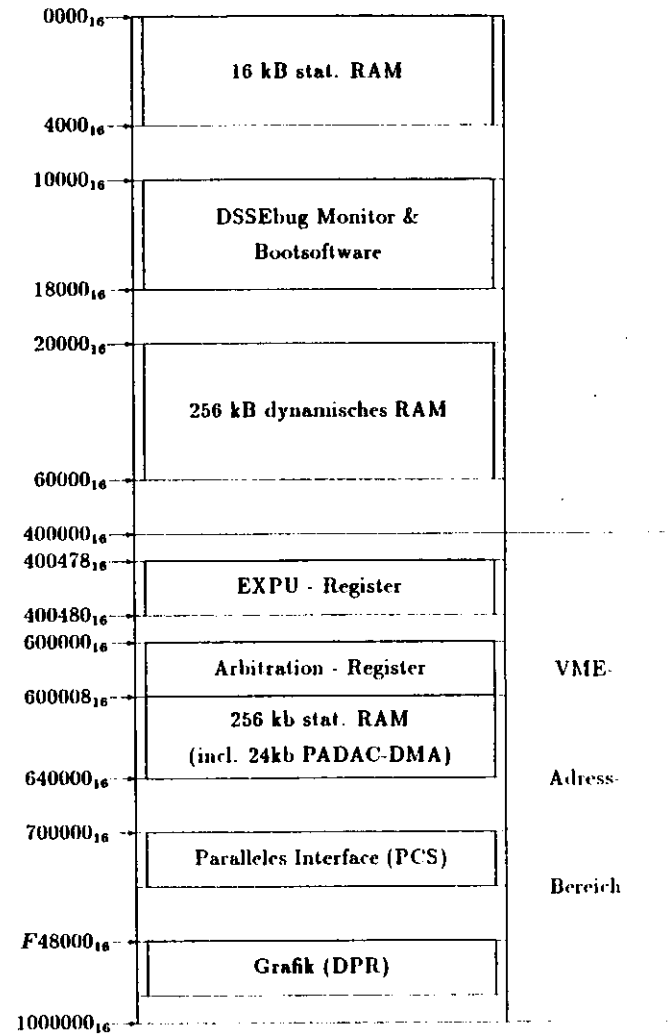
push DATA: Mit diesem Befehl kann man einen Datensatz aus der Mailbox in den statischen Speicher der Floating-Point Einheit laden.

Start: Der Befehl startet die Ausführung eines eventuell im Speicher des Targets vorhandenen Programmes.

HELP : Mit diesem Befehl werden die zur Verfügung stehenden Befehle ausgelistet.

LOAD : Dieser Befehl lädt ein Programm, welches auf der PCS im S-Record Format vorliegt, in den Programmspeicher des Controllers.

INIT : Der Befehl löst über die Steuerleitungen, welche von der Arbitration-Einheit zu den Targets führen, auf allen Targets einen *RESET* aus. Dadurch werden die Targets in einen definierten Anfangszustand gebracht. Außerdem werden



Der VMX-Bus wird hier nicht benutzt.

Abbildung 51: Adressraum des Controllers

die Datenstrukturen in der Mailbox bei der Ausführung dieses Befehles neu angelegt.

BOOT : Dieser Befehl schreibt das Minimalbetriebsystem, welches sich in oper.c befindet, zu den Targets.

DIR : Dieser Befehl listet die Speicherbelegung des Controllers aus. Dazu gehören die Adressen des Betriebsystemes und des auf dem Controller befindlichen Benutzerprogrammes.

STATUS : Durch die Eingabe dieses Befehles wird eine Liste der momentan verfügbaren Targets ausgegeben. In dieser Liste steht nicht nur der momentane Zustand der einzelnen Targets, sondern auch die Adressen der in den Targetspeichern befindlichen Programme und Daten.

GO : Wenn sich auf dem Controller ein Benutzerprogramm befindet, so startet dieser Befehl die Ausführung des Programmes.

C.3.1 Beschreibung der Funktionen von pcs_mail.c

dopen

```
int fd = dopen(byte_swap, modus, error_nr)
char *byte swap ;
int modus ;
unsigned int *error_nr ;
```

Der Befehl **dopen** öffnet von dem *UNIX* -Betriebsystem aus die Verbindung zu dem Controller.

fd : Dieses ist der sogenannte Filedeskriptor. In *C* wird jedem geöffneten File eine Integerzahl eben der Filedeskriptor zugeordnet. In dieser Routine wird der Wert dadurch erhalten, daß der Befehl *open* auf die parallele Schnittstelle durchgeführt wird. Alle weiteren Befehle, welche auf das File zugreifen sollen, bekommen diese Zahl, damit das Betriebsystem die Daten zwischen ihnen und dem File austauschen kann.

***byte swap :** Bei der parallelen Schnittstelle ist es möglich, daß die 16-Bit Worte um ein Byte versetzt ankommen. D.h., daß anstatt den Datenworten :

AC98	B845	C439

die Datenworte

98B8	45C4	*usw.*

eingelesen werden. Wird dieses von der Routine *dopen* anhand des versetzten Datenkopfes (siehe Abb.3) erkannt, so wird die Variable *byte_swap* nicht auf OFF, sondern auf ON gesetzt. Dadurch werden die weiteren Routinen auf die Situation vorbereitet. ON und OFF sind Makrodefinitionen, welche in den ersten Zeilen der Funktionssammlung definiert werden.

modus : Diese Integervariable legt die Richtung des Datentransfers fest. Sie kann sowohl durch die Konstante WRITE (Datenrichtung: PCS → Controller), als auch durch die Konstante READ (Datenrichtung: Controller → PCS) gesetzt werden. READ und WRITE sind Makrodefinitionen, welche aus dem Includefile *addr.h* der Bibliothek */usr/bop/include* stammen.

***error_nr :** Diese Variable ist für das Errorhandling zuständig (siehe auch 4.2.3).

univ_read

```
int nbytes = univ_read(fd, byte_swap, modus, error_nr)
int fd ;
char *byte_swap ;
union
{
unsigned char b[ACHTBYTE*8] ;
char nb[ACHTBYTE*8] ;
unsigned int w[ACHTBYTE*8] ;
int nw[ACHTBYTE*8] ;
long l[ACHTBYTE*8] ;
float f[ACHTBYTE*8] ;
double d[ACHTBYTE*8] ;
}
*buffer ;
unsigned int *error_nr ;
```

Mit dieser Funktion wird eine Gruppe von Bytes von der parallelen Schnittstelle eingelesen. Wieviel Bytes diese Gruppe umfaßt, gibt diese Routine als ihren Funktionswert (Typ: integer) an.

fd : Dieses ist der bereits von *dopen* bekannte Filedeskriptor, welcher die Identifikationsnummer für das geöffnete File darstellt. Er wird benötigt, damit das *UNIX* -Betriebsystem den in dieser Routine verwendeten *read*-Funktionen das richtige File zuweist.

***byte_swap :** Diese Variable erhält ihren Wert wie fd von der Routine dopen.

***buffer :** Diese Variable ist als eine *union* deklariert. Mittels einer *union* ist es möglich mehrere verschiedene Variabeltypen auf die gleichen Speicherplätze zu legen, wie es in der Abbildung 52 dargestellt ist. Der Vorteil dieses Variabeltyps in dieser Funktion liegt darin, daß für die Übergabe sämtlicher Variabeln in eine Richtung nur eine einzige Funktion benötigt wird.

***error_nr :** Diese Variable ist für das Errorhandling zuständig (siehe auch 4.2.3).

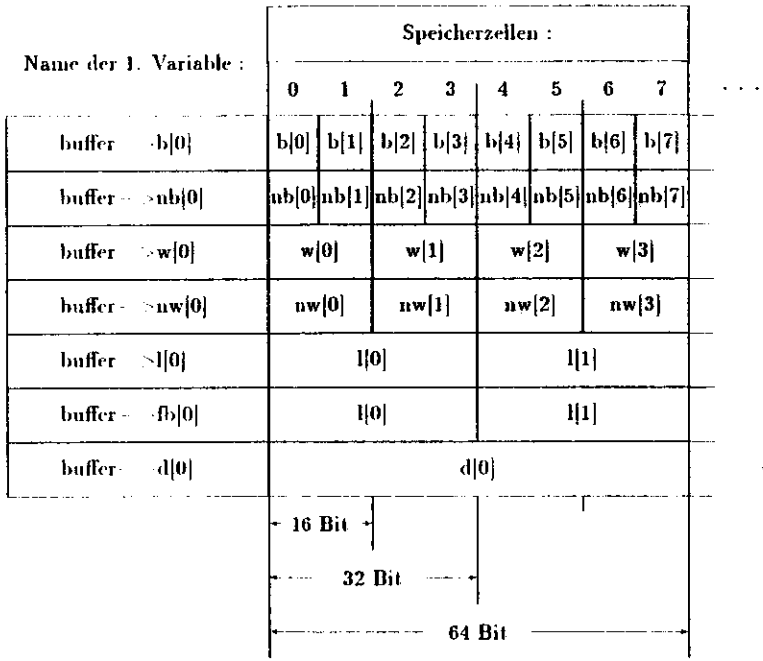


Abbildung 52: Aufbau der Union *buffer*

univ_write

```
univ_write(fd, buffer, nbyte, error_nr)
int fd ;
union
{
unsigned char b[ACHTBYTE*8] ;
char nb[ACHTBYTE*8] ;
unsigned int w[ACHTBYTE*8] ;
int nw[ACHTBYTE*8] ;
long l[ACHTBYTE*8] ;
float f[ACHTBYTE*8] ;
double d[ACHTBYTE*8] ;
}
*buffer ;
unsigned int nbyte ;
unsigned int *error_nr ;
```

Diese Funktion schreibt ein Feld von Variablen, welches eine Länge von *nbytes* Byte hat von der *PCS* zum Controller.

fd : siehe *univ_read*

***buffer :** siehe *univ_read*

nbyte : Dieses ist die Anzahl der Bytes, aus denen das in *buffer* befindliche Variabelfeld besteht.

***error_nr :** siehe 4.2.3

dclose

```
dclose(fd, modus, error_nr)
int fd ;
int modus ;
unsigned int *error_nr ;
```

Für den Fall, bei dem die Daten von der *PCS* zum Controller gesendet werden, schließt diese Funktion das File einfach mit dem *C* -Befehl *close(fd)* ab.
In dem anderen Fall, also wenn die Daten von dem Controller zur *PCS* gesendet werden, liest liest diese Funktion solange Daten von der Schnittstelle, bis der Controller keine Daten mehr in die Schnittstelle schreibt. Anschließend wird das File auf der *PCS* mit dem Befehl *close(fd)* abgeschlossen

fd : siehe *fd* bei *dopen*

modus : Über diese Variable wird der Routine mitgeteilt, in welche Richtung der Datentransfer geht, welcher abgebrochen werden soll.

$$READ : Controller \rightarrow PCS$$

$$WRITE : PCS \rightarrow Controller$$

***error_nr :** Variable zur Fehlerbehandlung, siehe auch 4.2.3

C.4 Systemsoftware auf dem Host

Auf dem *UNIX* -Entwicklungsystem habe ich alle Programme, welche das *BOP* -System betreffen in 7 verschiedene Directories eingeteilt :

/usr/bop/sys : Die Disassembler, sowie Programme die zur Bedienung der parallelen Schnittstelle von der *PCS* aus dienen, stehen in dieser Bibliothek.

/usr/bop/race : Hier befinden sich alle Programme, welche auf der Controller- oder auf der Targetebene laufen sollen (Racetracking- und Steuerprogramme sowie Bibliotheken), sowie alle anderen Programmdateien, welche zu der Erstellung dieser Programme notwendig sind.

/usr/bop/lib : Diese Bibliohek enthält nur das Programm, welches Dateien in ein sicher zu transportierendes Format bringt.

/usr/bop/include : Hier stehen die Tabellen und Datenstrukturen, welche in diverse Programme des *BOP* -Systems eingebunden werden. Insbesondere stehen hier alle für die Programme wichtigen Hardwareparameter des *BOP* -Systems.

/usr/bop/ibm : In diesem Katalog stehen die Programme, welche direkt für die Kommunikation mit der *IBM* zuständig sind, d.h. sowohl die Anwenderprogramme, als auch die Funktionsbibliothek. Nur die Files mit den Systemkonstanten stehen in der Bibliothek */usr/bop/include*.

/usr/bop/utility : In dieser Bibliotek stehen hauptsächlich Textfilter, welche dazu dienen Dateien vor dem Transport zu anderen Rechnern (*IBM*) umzuformen (Tabulatorzeichen zu löschen usw.).

/usr/bop/com : Bei den Programmen die hier stehen, handelt es sich um Kommandoprozeduren, welche zur Erzeugung, Sicherung und zum Transport von Dateien dienen.

D Weiteres zur VME-IBM Verbindung

D.1 Demonstrationsprogramme

demo.ass Dieses Programm ist in der Assemblersprache des MC 68000 geschrieben. Es sendet den Befehl CHANGE zur *IBM* und liest anschließend die Antwort von der *IBM* ein, welche dann in einem Speicherbereich abgelegt wird. Die letzten beiden Befehle (*jsr (a0)* und *trap #15*) sind systemspezifisch und bewirkten bei dem von mir verwendeten Rechner den Sprung in das Monitorprogramm.

demo.c Das Programm tut fast dasselbe, wie das obige Assemblerprogramm. Zusätzlich wird aber noch der Befehl HELP zur *IBM* gesendet, worauf die *IBM* , bzw. der in diesem Fall standardmäßig gestartete EXP-JOB, u.a. mit der Nummer des EXPU-Moduls antworten müßte. Der wesentliche Unterschied zwischen dem obigen und diesem Programm besteht aber darin, daß die Assemblerbefehle in diesem Programm durch Hexadezimalzahlen von der Sprache *C* aus erzeugt werden. In den Kommentarbereichen des Listings sind aber auch noch die Assemblerbefehle angegeben.
Eine zusätzliche Vereinfachung wird in diesem Programm dadurch erzielt, daß die funktionalen Blöcke von diesem Programm in Unterprogramme unterteilt sind. Beide *demo*-Programme benutzen keine weiteren Unterprogramme.

padata.c Mit diesem Programm wird die Benutzung der Funktionssammlung *func.c* demonstriert. Über ein Menue können dabei die verschiedenen Möglichkeiten der Funktionsbibliothek *func.c* ausprobiert werden. Hierzu gehört die Ausgabe von ASCII-Files auf den Druckern der *IBM* sowie das Schreiben dieser Files in Member einer Library auf der *IBM* . Ferner kann man auf dem Kleinrechner das Inhaltsverzeichnis einer Library sowie den Inhalt von Membern dieser Library listen.

pd_inter.c Auch dieses Programm benutzt teilweise Funktionen von *func.c*. Aber nachdem die Verbindung zu dem Kommunikationsprogramm auf der *IBM* hergestellt ist, kann der Benutzer interaktiv mit dem Programm auf der *IBM* kommunizieren. Dieses ist insbesondere dann von Interesse, wenn man Funktionen benutzen will, welche die Funktionssammlung *func.c* noch nicht vorsieht (z.B. den Transport von Daten- und GEP-Files).

D.2 Details zur IBM-Seite des Links

In der Tabelle 6 steht eine kurze Übersicht der Befehle, über welche das Programm *PCSLINK2* verfügt. Diese Liste wird dabei teilweise durch die Eingabe des Befehles HELP ausgegeben.

Allgemeine Bemerkungen Die Filenamen auf der IBM müssen grundsätzlich Mitglied einer Library sein, außerdem steht der *BOP* in dieser Tabelle stellvertretend für beliebige Kleinrechner, welche mit dem Programm kommunizieren. Um die Zeichenkonversion braucht man sich nicht zu kümmern, weil diese Konversion von dem IBM-Format (EBCDIC) auf das Kleinrechnerformat (ASCII) durch das Transferprogramm auf der IBM vorgenommen wird. Im folgenden zeige ich noch anhand von zwei Beipielen, wie man mit den Befehlen der Tabelle

Die Befehle des Programmes PCSLINK2		
Befehl	Datenrichtung	Bedeutung
FROMIBM < *NAME* >	*BOP* ← *IBM*	Name des IBM Files
FROMEXP < *NAME* >	*BOP* → *IBM*	Name des BOP Files
TOIBM < *NAME* >	*BOP* → *IBM*	Name des neuen IBM Files
TOEXP < *NAME* >	*BOP* ← *IBM*	Name des neuen BOP Files
COPY	*BOP* ↔ *IBM*	Datenaustausch von Textfiles
PRINTINT	*BOP* → *IBM*	Fileausgabe auf internen IBM Drucker
PRINTEXT	*BOP* → *IBM*	Fileausgabe auf exteren IBM Drucker
PRINTL1	*BOP* → *IBM*	Fileausgabe auf IBM Laserdrucker 1
PRINTL2	*BOP* → *IBM*	Fileausgabe auf IBM Laserdrucker 2
COPYD	*BOP* ↔ *IBM*	Transfer von Datenfiles
PLOT	*BOP* → *IBM*	Transfer von GEP-Dateien
COLUMN XX	*BOP* ↔ *IBM*	max. Spaltenanzahl bei Textdateien

Tabelle 6: Befehle des Transferprogrammes auf der *IBM*

6 vom Kleinrechner aus eine Verbindung zur *IBM* hergestellt:

Beispiel 1: Mit der folgenden Befehlskombination wird der Datentransfer des *IBM* -Textfiles *NAME1* auf das Kleinrechner Textfile *NAME2* eingeleitet.

```
FROMIBM NAME1
TOEXP NAME2
COPY
COLUMN 80
```

Beipiel 2: Über die folgende Befehlskombination wird das ASCII-File des Kleinrechners NAME1 auf dem externen Drucker der IBM ausgegeben:

```
FROMEXP NAME1
PRINTEXT
```

D.3 Details bei Senden von Datenblöcken zur *IBM*

Für diesen Transfer muß auf dem Controller das Programm ***dat_to_ibm.con*** und auf der PCS die Kommandoprozedur *DATA_trans* laufen. In dieser Prozedur wird zuerst das Programm *DAT_an.c* aufgerufen, welches das betreffende Binärfile in ein Textfile umwandelt. Anschließend tauscht das Programm *e_to_E* die kleinen "e"- Buchstaben in der Exponentialdarstellung durch große "E"-Buchstaben aus. Sodann wird das große Textfile in Blöcke zu je 5000-Zeilen zerlegt, welche dann einzeln durch das Programm *afile_to_con* < *file* > zum

Übersicht der beteiligten Programme :			
Ausführbares File	Quellprogramm	Bibliothek	benutzter Rechner
emmicon.con	emmicon.c	*/usr/bop/race*	Controller
emmibop.fpu	emmibop.c	*/usr/bop/race*	Target
rdata	rdata.c	*/usr/bop/sys*	PCS
DATA_trans	DATA_trans	*/usr/bop/com*	PCS
DAT_an.e	DAT_an.c	*/usr/bop/com*	PCS
e_to_E	e_to_E.c	*/usr/bop/utility*	PCS
afile_to_con	afile_to_con.c	*/usr/bop/sys*	PCS
dat_to_ibm.con	dat_to_ibm.c	*/usr/bop/ibm*	Controller

Tabelle 7: Programme, welche bei der Anwendung auf dem *BOP* beteiligt sind

Controller gesendet werden. Das Zerlegen der Textfiles ist notwendig, weil die Zeilenanzahl von partitionierten Files auf der IBM beschränkt ist.
Sobald die Datenfiles zur Controllerebene gesendet worden sind, werden sie durch das Programm *dat_to_ibm.con* dort empfangen und über das PADAC Netz zur IBM weitergeleitet. Auf der IBM werden diese Daten dann wiederum von einem Programm, in diesem Fall *PCSLINK2* empfangen und in Textfiles abgelegt. Um Speicherplatz zu sparen werden die Textdateien auf der IBM dann wiederum in binäre Files umgewandelt. Diese Dateien können dann beliebig mit den graphischen Hilfsmitteln der IBM ausgegeben werden.

Literatur

[1] Kernighan - Ritchie. *1977 The C Programming Language* Prentice-Hall,Inc.

[2] Jürgen Gulbins. *1985 UNIX - 2.Auflage* Springer-Verlag Berlin Heidelberg New York Tokyo

[3] K.H.Müller, I.Streker-Seeborg. *1984 FORTRAN 77 Programmierungsanleitung* Bibliographisches Institut Mannheim/Wien/Zürich

[4] DSSECPUA1 VME/VMXbus CPU MODULE
September 1984 HARDWARE MANUAL
DATA SUD SYSTEMES S.A.22
rue de Claret, F-34007 Montpellier, France

[5] DSSE 512CHROMA8
September 1984 HARDWARE MANUAL
DATA SUD SYSTEMES S.A.22
rue de Claret, F-34007 Montpellier, France

[6] PIO L11
Technical Information Manual CESI Computer Extension Systems, Incorporated
17511 EL Camino Real, Houston, Texas 77058

[7] Sonderheft Nr 229 der Zeitschrift Elektronik
Der VMEbus
Ein Bussystem für 16/32-Bit-Mikroprozessoren
1986 Franzis-Verlag Gmbh

[8] A.Wrulich, DESY HERA 82/04; DESY Bericht 84-026

[9] A.Wrulich, DESY HERA 85/03; DESY Bericht 85-06

[10] P.Wilhelm, Diplomarbeit, Universität Hamburg 1985

[11] PCS-Periphere Computer Systeme, München

[12] MC68000 16-BIT MICROPROCESSOR *1982 User's Manual (third edition)*

[13] F.Ware, L.Lin, R.Wong, B.Woo und C.Hansen *Electronics, July 12, 1984*

[14] 32 × 32 Six Port Register File WTL 1066 *Weitek Corp. 501 Mercury Drive Sunnyvale, CA 94086, USA*

[15] High Speed 64-bit IEEE Floating Point Multiplier (WTL 1064)
High Speed 64-bit IEEE Floating Point ALU (WTL 1065)
Weitek Corp. 501 Mercury Drive Sunnyvale, CA 94086, USA

[16] D.Moenkemeyer, EXP-Manual *Interner Bericht DESY R1-76/04 Dezember 1976*

[17] A Multiprozessor System For Parallel Proton Tracking
A.Deuter[5], W.Neff, H.Quehl, H.-J. Stuckenberg
und
P.Leu[6], E.Lohrmann, F.Schmidt, P.Wilhelm
Computing in High Energy Physics L.O. Hertzberger and W.Hoogland (Editors)

[5]Jetzt bei SCS Hamburg
[6]Jetzt bei SCS Hamburg

E Danksagung

Bei meiner Diplomarbeit hatte ich mit vielen Beschäftigten des DESY zu tun. Dabei fand ich durchweg ein freundliches Klima vor und mir wurde praktisch immer die erforderliche Unterstützung bereitgestellt. Unter allen diesen Personen möchte ich aber die folgenden hervorheben, welche mich bei meiner Arbeit besonders unterstützt haben :

- Herrn Prof. Dr. Erich Lohrmann, welcher mir nicht nur das Thema gestellt hat und die Manuskripte durchsah, sondern auch eine schnelle Bewältigung des Themas unterstützte. Besonders angenehm war auch, daß er mir bei der Bewältigung des Themas weitgehend freie Hand ließ.

- Herrn Dr. Armin Deuter für seine freundschaftliche und intensive Betreuung, durch welche mir insbesondere in der Anfangsphase die Arbeit sehr erleichtert wurde. Dadurch wurde eine zügige Bewältigung des Themas erst möglich.

- Der gesamten Gruppe F56, namentlich A.Millhouse, M.Möller, W.Neff, H.Quehl und Dr. H.J.Stuckenberg, sowie den in dieser Gruppe beschäftigten Diplomanten und Praktikanten. Die Mitglieder dieser Gruppe haben mich in allen Teilen meiner Arbeit hervorragend unterstützt und standen mir jederzeit zur Seite.

- K. Rehlich und D. Notz, welche mir trotz vieler eigener Aufgaben bei Hard- und Softwareproblemen geholfen haben.

Ferner möchte ich allen Mitdiplomanden und Doktoranden der Gruppen F35 und F1 danken. Sie haben für ein freundliches und angenehmes Arbeitsklima gesorgt, durch welches die Arbeit am DESY für mich sehr angenehm war.

F Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit selbständig unter Verwendung der angegebenen Literatur verfaßt zu haben.

Hamburg, den 2.Februar 1988 Torsten Woeniger