ETH zürich

ETH Library

Strukturen und Konzepte zur Multiprogrammierung und ihre Anwendung auf ein System für Datenstationen (Hexapus)

Doctoral Thesis

Author(s):
Sandmayr, Helmut

Publication date:
1975

Permanent link:
https://doi.org/https://doi.org/10.3929/ethz-a-000204376

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Diss. ETH 5537

Strukturen und Konzepte zur Multiprogrammierung und ihre Anwendung auf ein System für Datenstationen (Hexapus)

ABHANDLUNG

zur Erlangung
des Titels eines Doktors der Mathematik
der
EIDGENÖSSISCHEN TECHNISCHEN HOCHSCHULE
ZÜRICH

vorgelegt von

HELMUT SANDMAYR
dipl. Math. ETH
geboren am 12. April 1947
aus Deutschland

Angenommen auf Antrag von

Prof. Dr. N. Wirth, Referent
Prof. Dr. M. Engeli, Korreferent

Juris Druck + Verlag Zürich
1975

ISBN 3 260 03954 6

Herrn Prof. Dr. N. Wirth danke ich herzlich für die Anregung zur Wahl dieses Themas, die Betreuung der Durchführung und die aus vielen Diskussionen erwachsenen Anregungen.

Herrn Prof. Dr. M. Engeli danke ich für einige wertvolle Hinweise und die freundliche Uebernahme des Korreferates.

Mein Dank gilt auch den Mitarbeitern des Rechenzentrums, W. Bächi und S. Fäsch, die die im Zentralrechner für den Anschluss der Datenstationen notwendigen Programme geschrieben haben.

Dem Direktor des Rechenzentrums, Herrn A. Schai, und meinem Kollegen E. Marmier danke ich herzlich für ihr Interesse und ihre stete Diskussionsbereitschaft.

Inhaltsverzeichnis

	Seite
1. Einleitung	3
1.1. Problemkreis Betriebssystem	5
1.2. Begriffe und Bezeichnungen	6
1.3. Die Problemstellung des Systems HEXAPUS	8
1.4. Maschinenorientierte oder höhere Programmiersprache?	9
1.5. Methodik der Implementation	11
2. Spezifikation der Implementationssprache	14
2.1. Ziel und Zweck der neuen Konstruktionen	14
2.2. Der Prozess	16
2.3. Der Modul	17
2.4. Der geschützte Modul	22
2.5. Die Queue	24
2.6. Steuerung peripherer Geräte: Der Driver Modul	27
2.7. Sonstige Erweiterungen und Einschränkungen	29
3. Eine Implementation der Sprachelemente	31
3.1. Statische Auflösung der Konstruktionen	31
3.1.1. Prozesse und Prozeduren	31
3.1.2. Der Modul, der geschützte Modul und die Queue	33
3.1.3. Periphere Geräte	35
3.2. Die Laufzeitunterstützung: der NUCLEUS	36
3.2.1. Die Prozessorverwaltung	37
3.2.2. Routinen für die Synchronisation von Prozessen	45
4. Das System HEXAPUS	52
4.1. Struktur des Systems und der Information	52
4.1.1. Zuordnung peripherer Geräte zu Datenströmen	55
4.1.2. File-Struktur und Verschlüsselung der Information	57
4.2. Implementationsstrategie	59
4.2.1. Die Pufferbehandlung	61
4.2.2. Steuerung und Kontrolle der Verbindungen	66
4.2.3. Steuerung von Ein- und Ausgabeprozessen	71
4.2.4. Die allgemeine Struktur der Ein- und Ausgabeprozesse	74

Seite

4.3. Behandlung der Linienströme 76
4.3.1. Linienkontrolle 81
4.3.2. Struktur der externen und internen Linienprozesse 84
4.4. Interpretation und Ausführung von Befehlen 88

5. Steuerung peripherer Geräte 90
5.1. Allgemeine Struktur von Ein- und Ausgaberoutinen 90
5.2. Konsolendriver 93
5.3. Liniendriver 97
5.3.1. Das Linienprotokoll 98
5.3.2. Starten und Aufrechterhalten der Verbindung 100

6. Erfahrungen und Folgerungen 103
6.1. Trennung der Aspekte 103
6.2. Bemerkungen und Zahlen zur Implementation 104
6.3. Ergänzungen und mögliche Erweiterungen der Sprache 107
6.4. Schlussfolgerungen 108

Literaturverzeichnis 111

Anhang A. Syntaxdiagramme 114
B. Auszug aus der Bedienungsanleitung für eine Datenstation 120
C. Vollständiges Programm des Liniendrivers 127
D. Struktur des Programms HEXAPUS für eine Datenstation 131

Programming of computers
for their own sake is never
a goal, but always a means.

W.L. van der Poel

1. Einleitung

Rechenanlagen werden heute immer häufiger und für immer neue Anwendungen eingesetzt. Bei stets wachsender technischer Leistungsfähigkeit stellt sich das Problem der wirtschaftlichen Auslastung, das durch die Verwendung von Betriebssystemen gelöst werden soll. Dies sind Programme, die mehreren Benutzern die gleichzeitige Verwendung von Teilen einer Anlage erlauben, ausgehend von der Erkenntnis, dass ein Programm im allgemeinen nicht alle Komponenten gleichmässig auslastet.

Mit der technischen Entwicklung und den wachsenden Ansprüchen der Benutzer wächst auch die Vielfalt, Komplexität und Grösse dieser Systeme, und zwar schneller als ihre Effizienz und Zuverlässigkeit. Bei der Entwicklung der meisten dieser Systeme werden Sprachen verwendet, die volle Allgemeinheit bei der Programmierung einer Anlage und viele Möglichkeiten zu lokaler Optimierung der Programme erlauben. Diese Allgemeinheit bringt aber gleichzeitig grösseres Risiko mit sich als Methoden und Sprachen, die sich auf gewisse Strukturen und damit vielleicht auch gewisse Anwendungen beschränken, die aber eher zu übersichtlicheren und zuverlässigeren Systemen führen.

Ziel dieser Arbeit ist die Entwicklung eines modularen, portablen Systems, das die Verwendung von verschiedenen Kleinrechnern als Datenstationen für stapelweise Verarbeitung erlaubt. Dabei wird eine möglichst einfache Erweiterung von PASCAL [Wirth 71] und ein zugehöriges System zur Multiprogrammierung entwickelt, wobei besonders auf folgende Punkte geachtet wird:

1) Die Entwicklung und Beschreibung des Systems für die Datenstationen soll so weit als möglich in der erweiterten Sprache erfolgen, d.h.

2) Die Behandlung peripherer Geräte soll in der Sprache explizit möglich sein.

3) Es werden syntektische Strukturen definiert, die die zusammenhängende Darstellung von Teilaspekten (Moduln) erlauben, wie sie bei der Entwicklung eines Systems behandelt werden.

4) Die Sprache soll eine klare Unterteilung des Systems in
   a) maschinenabhängige
   b) peripherieabhängige
   c) systemspezifische

   Teile erlauben, in Teile also, die bei Uebergang auf andere Rechner oder andere Peripherie zu ändern sind, und Teile, die nur von den Spezifikationen des Systems abhängen.

Die so entwickelte Sprache kann dann Ausgangspunkt für die Implementation komplexerer Systeme sein, die ihrerseits die Einführung weiterer Konstruktionen oder Strukturen bedingen. Eine solche, an der Praxis orientierte Entwicklung einer Sprache für die Systemprogrammierung ist als Gegenstück zur Definition einer "vollständigen" Sprache zu sehen, die alle 'erdenkbaren' Konstruktionen enthält und in einem Compiler festhält, bevor die Notwendigkeit der Konstruktionen an einsatzfähigen Systemen überprüft wird.

In dieser Arbeit wird nun ausgehend von Arbeiten von P. Brinch Hansen [73, 74] und C.A.R. Hoare [73] eine Erweiterung von PASCAL definiert und die Entwicklung des Systems für die Datenstationen beschrieben, das als abgeschlossenes Programm den Kunden des Rechenzentrums der ETH Zürich zum Betrieb von Kleinrechnern als Datenstationen zur Verfügung steht.

1.1. Problemkreis Betriebssystem

Betriebssysteme bestehen aus einer Menge von Programmen, die als Zwischenglied zwischen Benutzer und Rechenanlage stehen und dabei gewisse Forderungen erfüllen sollen, wie z.B. gleichmässige Auslastung der verschiedenen Komponenten einer Anlage. Man kann das Betriebssystem als die Umgebung betrachten, in der die Benutzerprogramme schliesslich ausgeführt werden, d.h. sie simulieren eine Maschine, für die der Benutzer sein Programm schreibt, eine virtuelle Maschine.

Aus dieser Zwischenstellung von Betriebssystemen ergeben sich folgende Forderungen:

(1) Optimalität und Wirtschaftlichkeit: Das System soll eine gute Auslastung der Anlage erlauben; dabei ist zu berücksichtigen, dass der Verwaltungsaufwand für das System selbst nicht zu der nach aussen sichtbaren Leistung zählt.

(2) Sicherheit: Das System soll zuverlässig und korrekt arbeiten, da es Grundlage zur Berechnung von Resultaten ist, d.h. es soll soweit wie möglich vor Fehlern von aussen (technische Anlagen, Mitbenutzer) schützen, wie auch selbst absolut zuverlässig sein.

(3) Flexibilität und Modularität: Das System soll leicht an technische oder betriebliche Aenderungen angepasst werden können. Dabei ist Punkt (2) zu beachten, wobei insbesondere von einer Aenderung nicht betroffene Teile in ihrem Verhalten nicht beeinflusst werden sollen.

Diese Forderungen wird man in modifizierter Form an jedes Programm stellen, wobei je nach Anwendung der eine oder andere Punkt stärker betont wird. Welches sind nun spezifische Punkte des Betriebssystems?

(a) Sie sind i.a. gross und komplex.
Das bedingt, dass meistens mehrere Personen an Entwurf und Erstellung über längere Zeit beteiligt sind, d.h. ein System

ist aus Stücken verschiedener Programmierer zusammengesetzt. Durch die Verschiedenartigkeit der Komponenten und der Ansprüche, die das System zu erfüllen hat, umfasst es viele Algorithmen und Vorgänge, die zusammen mit der Art ihrer Verknüpfung letztlich die Leistungsfähigkeit einer ganzen Anlage bestimmen.

(b) Sie sind weitverbreitet.
Gleiche Systeme werden für viele Anlagen verwendet, und zwar nicht unter der Kontrolle ihrer Autoren; sie haben vielmehr viele 'Kunden', die meistens sogar ihre eigenen Modifikationen anbringen wollen.

(c) Sie werden über lange Zeit verwendet.
Systeme sind im allgemeinen nie fertig. Sie werden nicht wie "experimentelle" Programme für die kurzfristige Behandlung eines Problems entwickelt, sie bilden vielmehr die Grundlage des Betriebs einer Rechenanlage und müssen immer verfügbar sein. Unterhalt und Anpassung sind wesentlich; da beim Entwurf eines Systems nicht alle möglichen Anwendungen voraussehbar sind (z.B. technische Neuerungen, neue Zusammenstellung von Komponenten), muss das System ausbaufähig sein, auch für andere Programmierer als die ursprünglichen Autoren selbst.

Von einem Betriebssystem werden also Zuverlässigkeit, Wirtschaftlichkeit und Modularität gefordert unter erschwerenden Bedingungen wie Grösse, Komplexität, Ausbaufähigkeit und Kommunikation zwischen Entwicklungsgruppe und verschiedenen "Generationen" von Unterhaltsgruppen. Dabei ist das dynamische Verhalten der im System enthaltenen Programme von grösserem Interesse als statische Gesichtspunkte.

1.2. Begriffe und Bezeichnungen

Es ist üblich, die sequentielle Ausführung einer Folge von Operationen, eines Programmes, als sequentiellen Prozess oder einfach Prozess zu bezeichnen: "... what happens during such a program execution is called 'a sequential process'" [Dijkstra 68].

Die Operationen werden dabei von Prozessoren ausgeführt, die im Mehrprogrammbetrieb einem Prozess i.a. nicht fest zugeordnet sind, da gleichzeitig oft mehr Prozesse ablaufen als Prozessoren zur Verfügung stehen und da vielleicht gewisse Operationen nur von einem bestimmten Prozessor ausgeführt werden können. Die Zuordnung der realen Prozessoren zu den Prozessen geschieht nach einem zu spezifizierenden Algorithmus, dem Scheduling-Algorithmus, für den ein Prozess eine Verwaltungseinheit bildet, der er einen eigenen Pseudo-Prozessor, eine virtuelle Maschine, zur Verfügung stellt.

Von der Geschwindigkeit des Prozessablaufes soll nur angenommen werden, dass sie positiv ist [Dijkstra 71], d.h. dass der Scheduling-Algorithmus vorhandene Prozessoren den verschiedenen Prozessen "gleichmässig" zur Verfügung stellt.

Prozesse arbeiten auf Objekten, die entweder Variablen sind oder durch solche repräsentiert werden; diese Objekte können lokal oder global zu einem Prozess sein, d.h. entweder hat durch die syntaktische Struktur der Variablen-Vereinbarung nur der eine Prozess zu diesen Objekten Zugriff, oder es können auch andere Prozesse zugreifen. Prozesse, deren Ablauf sich zeitlich überschneidet, heissen gekoppelt, wenn sie gemeinsame globale Objekte verwenden, sonst sind sie unabhängig.

Koppelung oder Synchronisation von Prozessen kann aus verschiedenen Gründen notwendig sein:

1. Gegenseitiger Ausschluss (mutual exclusion):
   Für eine Folge von Operationen soll einem Prozess die exklusive Verwendung von globalen Objekten ermöglicht werden, während deren Ausführung z.B. gewisse Relationen für die globalen Objekte nicht erfüllt sind, und daher kein weiterer Prozess für die Dauer dieser Operationen zu diesen Objekten zugreifen soll. Eine Sequenz solcher Operationen heisst kritischer Bereich.

2. Austausch von Zeitsignalen:
   Damit soll die explizite gegenseitige Synchronisation von Prozessen ermöglicht werden.

3. Austausch von Daten:
   Bei der Kooperation von Prozessen zur Behandlung einer gewissen Aufgabe müssen Prozesse gegenseitige Daten und Information austauschen können.

Variable und Operationen, die der Synchronisation von Prozessen dienen, heissen Synchronisations-Variablen bzw. -Operationen.

Verknüpft mit der Synchronisation ist das Problem des Deadlocks: das ist ein Zustand, in dem zwei oder mehrere Prozesse auf einen Zustand oder ein Signal warten, die nur einer der anderen wartenden Prozesse herbeiführen bzw. auslösen könnte.

1.3. Die Problemstellung des Systems HEXAPUS

Das System HEXAPUS soll den Anschluss kleiner Rechenanlagen verschiedener Typen (HP 2100, PDP-11, usw.) an einen zentralen Rechner (CDC 6500/6400) und ihre Verwendung als mittelschnelle Datenstationen zur stapelweisen Verarbeitung erlauben. Dabei sind folgende Bedingungen zu erfüllen:

1. An die Satellitenrechner sollen alle Arten von peripheren Geräten angeschlossen werden können. (Normalstation: Kartenleser, Drucker, Lochstreifenleser und -stanzer.)

2. Die Datenstationen sollen ohne speziellen Operateur betrieben werden, d.h. jeder Benutzer bedient die Anlage selbst, was daher mit minimalem Aufwand ("narrensicher") möglich sein muss.

3. Informationsfluss soll möglich sein zwischen:
   Eingabegerät --> Zentralrechner (Inputqueue)
   Zentralrechner (Outputqueue) --> Ausgabegerät
   Eingabegerät --> Ausgabegerät (lokal)

4. Mehrere Geräte sollen gleichzeitig mit dem Zentralrechner Information austauschen können.

5. Information soll vom Kartenleser und anderen Eingabegeräten zu einem File zusammengefasst werden können (z.B. Jobkarten mit Daten auf Lochstreifen).

Um bereits vorhandene Anlagen mit minimalen Erweiterungen verwenden zu können, sind folgende technische Gegebenheiten zu berücksichtigen:

6. Die verschiedenen Rechner sollen mit 8k 16-Bit-Worten Vordergrundspeicher, ohne Hintergrundspeicher und möglichst ohne Uhr auskommen.

7. Der Anschluss an die Zentralanlage erfolgt synchron mit 2400 bd über ein Terminalsystem (VENUS) mit halb-duplex Leitung. Uebergang zu höheren Geschwindigkeiten und Vollduplexbetrieb soll leicht möglich sein.

1.4. Maschinenorientierte oder höhere Programmiersprache?

Die Entwicklung höherer Programmiersprachen (general purpose languages) führte zu Sprachen, die die Definition von hierarchischen Daten- und Programmablaufstrukturen erlauben, wobei nur abstrakte Eigenschaften dieser Strukturen, nicht aber ihre Darstellung in einer Rechenanlage zur Behandlung eines Problems in einem Programm verwendet werden. In diese Kategorie gehören Sprachen wie SIMULA 67 [Dahl 70], ALGOL 68 und PASCAL [Wirth 71]. Verknüpft mit diesen höheren Sprachen sind die Begriffe "strukturierte Programmierung" [Dijkstra 72], "strukturiertes Programm", "schrittweise Programmverfeinerung" [Wirth 71b], "modulare Zerlegung" [Parnas 72] und "action clusters" [Naur 69, 74], die Methoden zur Erstellung zuverlässiger, systematischer Programme bezeichnen.

Diese höheren Sprachen, die auf verschiedenen Anlagen zur Verfügung stehen sollen, sind weitgehend maschinenunabhängig konzipiert und erlauben nur einen sehr beschränkten Zugriff zu physikalischen Komponenten einer Rechenanlage, der durch ein ausgearbeitetes, aber

starres Laufzeitsystem kontrolliert wird.

Im Gegensatz dazu enthalten die seit 1967 in zunehmender Zahl entstehenden maschinenorientierten Sprachen Operationen und Datenstrukturen, die direkten Zugriff zu Komponenten einer Rechenanlage erlauben, bzw. sich direkt auf Maschinenstrukturen abbilden lassen; hingegen unterscheiden sich schon bei PL360 [Wirth 68], dem ersten Vertreter dieser Klasse, die Programmablaufstrukturen nicht wesentlich von denen höherer Sprachen. Ein auf die Hardware ausgerichtetes elementares Typenkonzept und die Möglichkeiten des direkten Zugriffs zu Komponenten der Rechenanlage, wie Registern oder peripheren Geräten, bieten die Freiheit, auch in der Sprache nicht vorgesehene Strukturen zu programmieren.

Versucht man nun, für die Systemprogrammierung eine Sprache aus einer der beiden Klassen um notwendige Möglichkeiten von Sprachen der anderen Klasse und zusätzliche Strukturen zur Darstellung von Konzepten der Multiprogrammierung zu erweitern, so sprechen folgende Punkte für die Wahl einer höheren Sprache als Ausgangspunkt:

1. Sicherheit und Zuverlässigkeit.
   Durch die Freiheiten maschinenorientierter Sprachen wird gleichzeitig dem Compiler die Möglichkeit zu weitgehenden Fehlertests entzogen, z.B. durch fehlende Redundanz im Programmtext (Variablenvereinbarung), durch fehlendes oder durch automatische Konversionen abgeschwächtes Typenkonzept, freie Verwendung von Adressvariablen (Pointer), usw.; Fehler sollten aber so früh wie möglich, am besten bereits bei der Uebersetzung, erkannt werden können, denn gerade bei der Multiprogrammierung macht die zeitabhängige Interaktion gleichzeitig ablaufender Prozesse das Finden von Fehlern durch systematisches Testen nahezu unmöglich (vgl. [Brinch Hansen 73]).

2. Portabilität.
   Durch geeignete Wahl der Konstruktionen ist es bei höheren Sprachen möglich, notwendige Maschinenabhängigkeit von Programmen zuzulassen, sie aber durch die Art der Struktur explizit und lokal zusammenzufassen. Die Abbildung maschinenunabhängiger Algorithmen und Strukturen in den ausführbaren

maschinenabhängigen Code wird dem Compiler überlassen (siehe auch 4.).

3. Dokumentation, Unterhalt und Verständlichkeit.
   Programme sind leichter zu dokumentieren, besser verständlich und somit auch leichter und zuverlässiger zu unterhalten, wenn Konzepte nicht zusammen mit maschinenabhängiger Information dargestellt sind.

Ein möglicher Nachteil dieser Methode ist:

4. Ineffizienz bezüglich Zeit und Platz.
   Um diesen Nachteil, der durch ineffiziente Abbildung eines Programmes auf Maschinencode durch einen Compiler entsteht, nicht allzu stark ins Gewicht fallen zu lassen, ist eine solche höhere Sprache zu wählen, für die (bezüglich generiertem Code) effiziente Compiler erstellt werden können, so dass der Programmierer nicht verleitet wird, in der Sprache vorhandene Strukturen aus Effizienzgründen schliesslich selber auszuprogrammieren oder zu umgehen.

1.5. Methodik der Implementation

Das ganze System HEXAPUS besteht primär aus zwei Teilen, einem Teil im Zentralrechner als Verbindungsglied zwischen der Linie und dem System des Zentralrechners und einem zweiten Teil im Rechner der Datenstation.

Der Anschluss der Datenstation an die Zentralanlage soll über das Terminalsystem VENUS erfolgen [Schicker 75]. Dieses System verfügt bereits über einen Driver zur Bedienung verschiedenartiger Konsolen mit verschiedenen Uebertragungsgeschwindigkeiten (110bd-2400bd) sowie über Routinen zum Einfügen eines Jobs in die Eingabeschlange oder Holen aus der Ausgabeschlange.

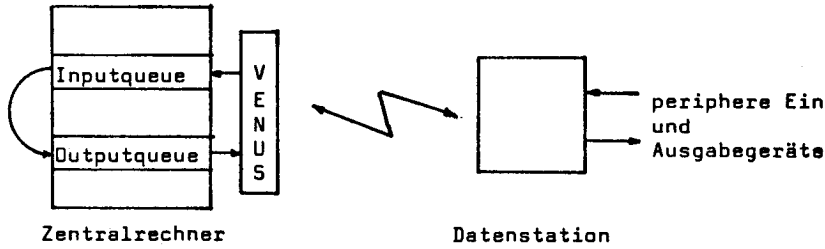


Abb. 1.1: Verbindung Datenstation - Zentralrechner

Dieses System ist folgendermassen zu erweitern:

a) im Driver: zur Behandlung eines zu spezifizierenden Protokolls.

b) um Programmteile zur Aufrechterhaltung der Verbindung und Aufbereitung der Information vor bzw. nach der Uebermittlung, sowie um Aufrufe von Systemprogrammen zur Ausführung von Befehlen, die von der Datenstation übermittelt wurden.

Diese die Zentralanlage betreffenden Arbeiten wurden von Mitarbeitern des Rechenzentrums der ETH nach üblichen Methoden der Systemprogrammierung in Assembler und maschinenorientierter Sprache ausgeführt und werden im Rahmen dieser Arbeit nicht weiter behandelt.

Für die <u>Datenstationen</u> sind verschiedene bereits vorhandene Kleinrechner (HP 2100, PDP 11, ...) vorgesehen, für die ein <u>einheitliches</u> System zu entwickeln ist. Daher wird auf verschiedene bereits vorhandene Systeme auf diesen Rechnern nicht eingegangen, sondern ein vollständiges System entwickelt werden; diese Entwicklung, die in dieser Arbeit skizziert wird, soll auf Grund der Ueberlegungen von Abschnitt 1.4 in einer höheren, maschinenunabhängigen Programmiersprache erfolgen. Diese Implementationssprache ist geeignet zu definieren, so dass sie eine möglichst vollständige Entwicklung und Darstellung des Kleinrechnersystems erlaubt; insbesondere sollen auch Möglichkeiten zur expliziten Behandlung peripherer Geräte zur Verfügung gestellt werden.

Ein Compiler für die Sprache ist im Rahmen dieser Arbeit nicht vorgesehen, eine mögliche Implementation wird jedoch skizziert. Die für das System entwickelten Programme sind daher von Hand in die Assembler-Sprache der verschiedenen Rechner zu übersetzen.

2. Spezifikation der Implementationssprache

Entsprechend den Erörterungen des letzten Abschnittes soll das System HEXAPUS in einer erweiterten höheren Sprache geschrieben werden; als Grundlage dient hierbei PASCAL [Wirth 71a] und die Erweiterung CONCURRENT PASCAL [Brinch Hansen 74], die bereits Konzepte zur Multiprogrammierung enthält, die aber den Zugriff zu peripheren Geräten auf vordefinierte Prozeduren zur Ein- oder Ausgabe eines Blocks von Daten beschränkt. Brinch Hansen begründet seine Wahl für PASCAL als Ausgangssprache folgendermassen:

> "PASCAL permits hierarchal structuring of data and program, extensive error checking at compile time, and production of efficient machine code on present computers. It combines the clarity needed for teaching the subject with the efficiency required for designing operating systems."

Ausgehend von der in [Jensen und Wirth 74] gegebenen Definition von PASCAL sollen in diesem Kapitel nur die Erweiterungen und Aenderungen aufgeführt werden; die Erweiterungen sollen so gewählt werden, dass die Sprache einfach bleibt [Wirth 74] und die oben zitierten Eigenschaften auch für sie erhalten bleiben.

Da die Wahl von Konstruktionen sicherlich einen Einfluss auf die Art und Weise hat, in der ein Problem angegangen und schliesslich programmiert wird, bleibt die Definition an gewissen Punkten allgemein und überlässt die Einschränkung auf einen Problemkreis einer entsprechenden Implementation.

2.1. Ziel und Zweck der neuen Konstruktionen

Die Entwicklung von Konzepten zur Lösung eines Problems (Behandlung einer Situation) ist ein kreativer Vorgang, der oft Einsichten erfordert, die erst in einem späteren Stadium bei der Behandlung des Problems gewonnen werden. Daher ist ein Softwareprojekt - zumindest bei der Behandlung eines neuen Problems - ein iterativer

Prozess, der in den verschiedenen Entwicklungsstadien Neukonstruktion und Ueberarbeitung erfordert.

Jedes so entwickelte Konzept entspricht notwendigerweise einem begrenzten Aspekt des Problems. Kann man nun diesen Teil auch in einem syntaktisch abgeschlossenen Programmstück darstellen, und setzt man die Lösung aus solchen Stücken zusammen, so werden die Lösungen eher zuverlässig und leichter adaptierbar, da sich Teilaspekte bei konzentrierter Bearbeitung leichter erfassen und bearbeiten lassen. Die Zusammensetzung solcher Teile zu grösseren Einheiten verlangt keine Kenntnis über deren innere Struktur, während andererseits die Korrektheit eines Teiles ohne Kenntnis der Einbettung in das Gesamtsystem nachprüfbar ist.

Um die Sprache möglichst einfach zu halten, soll keine System-Sprache definiert werden, die sich für Compilerbau, Betriebssysteme und Echtzeitsysteme gleichermassen eignet. Primäres Ziel ist vielmehr eine Sprache zur vollständigen Entwicklung und Beschreibung eines Betriebssystemes; aus PASCAL werden daher nur die dazu notwendigen Konstruktionen und Konzepte übernommen. Denn während bei der Entwicklung von Compilern das Gewicht mehr auf der Behandlung von Zeichenfolgen, dem Einsatz von Registern und der Ausnutzung spezifischer Instruktionen liegt, wird bei Betriebssystemen ein ausgeprägter Mechanismus für die Behandlung von Ein- und Ausgabe und von Interrupts benötigt; bei Echtzeitproblemen schliesslich liegt die Betonung noch stärker auf dem zeitlichen Verhalten der Operationen und Prozesse.

Bei geeigneter Wahl der neuen Sprachkonstruktionen sollte dabei die Effizienz des resultierenden Codes nicht (wesentlich) beeinflusst werden, während der Gesamtaufwand für Entwicklung, Test und Unterhalt wesentlich kleiner werden sollte; so erlaubt z.B. der gegenseitige Ausschluss aller Operationen auf einer für mehrere Prozesse gemeinsamen Variablen die Analyse der Wirkung dieser Operationen mit Methoden (vgl. [Hoare 73]) der rein sequentiellen Programmierung.

2.2. Der Prozess

"Starting point of a theory of operating systems must be a sequential process - a succession of events that occur one at a time. This is the way our machine work; this is way we think." [Brinch Hansen 73].

Der sequentielle Prozess, die Verwaltungseinheit in einem System zur Multiprogrammierung, wird entsprechend der im folgenden Diagramm dargestellten Syntax vereinbart:

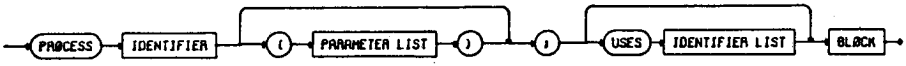


Abb. 2.1: Prozess

Um die Verflechtung eines Prozesses mit dem ihn umgebenden Programm explizit werden zu lassen, sind alle im Prozess verwendeten globalen Objekte und Prozeduren (oder die sie enthaltenden Moduln) im Programmkopf in der durch uses eingeleiteten Liste aufzuführen. Neben den Abschnitten zur Beschreibung der Daten bzw. Algorithmen dient diese Liste (zusammen mit der Parameterliste) zur expliziten Spezifikation der Umgebung, in der der Prozess ablaufen soll.

Prozesse können nur in einem Block der Stufe Ø vereinbart werden, d.h. es gibt keine zu Prozessen oder Prozeduren lokalen Prozesse. Ein Prozess wird initialisiert und gestartet durch eine Anweisung der Form

```
start procsname(...)
```

und existiert bis nach der Ausführung seiner letzten Anweisung. Da es keine zu Prozessen lokale Prozesse gibt und somit durch den Schutz der Blockstruktur kein anderer Prozess Zugriff zu den lokalen Variablen eines Prozesses hat, werden auch die Variablen eines Prozesses nach dessen Beendigung nicht mehr benötigt (und der Speicherplatz kann je nach Implementation wieder anderweitig verwendet werden). Mehrfaches Starten eines Prozesses ist zulässig

und führt zur Ausführung mehrerer gleicher Prozesse.

Variable, zu denen mehrere Prozesse (über ihre explizite oder implizite Parameterliste) Zugriff haben (globale Variable oder Variable in einem globalen Modul), müssen in einem geschützten Modul vereinbart werden oder von einem dort vereinbarten Typ sein (siehe 2.4).

Bemerkung: Durch diese allgemeine Definition soll die Struktur eines Prozesses getrennt bleiben von der Instanz eines Prozesses; diese Trennung kann für effizientere Implementationen für spezielle Anwendungen durchaus eingeschränkt werden, indem etwa mehrfaches Starten eines Prozesses nicht zugelassen wird und somit zu jeder Zeit höchstens eine Instanz eines Prozesses aktiv ist, oder indem syntaktische Angaben über die Maximalzahl gleichzeitig aktiver Instanzen eines Prozesses verlangt werden.

Die von Brinch Hansen in [74] eingeführte Notation mit der Struktur eines Prozesses als Typ und einer Instanz als Variable wurde nicht übernommen, da sie zu begrifflichen Schwierigkeiten führt, wenn etwa nur solche "Variable" Parameter eines Prozesses sein können, die nicht Variable im herkömmlichen Sinn sind (nämlich Prozess und Monitor).

2.3. Der Modul

Durch den Modul soll neben der Blockstruktur eine weitere Möglichkeit geschaffen werden, um logisch zusammenhängende Typen oder Objekte und Operationen auch syntaktisch zusammenfassen und Objekte vor unkontrolliertem Zugriff von ausserhalb des Moduls schützen zu können. Ein Modul kann als eine Zusammenfassung von Anweisungen zu "action clusters" [Naur 69,74] betrachtet werden, wobei dann das Zusammenfügen dieser Einheiten einem Compiler überlassen wird.

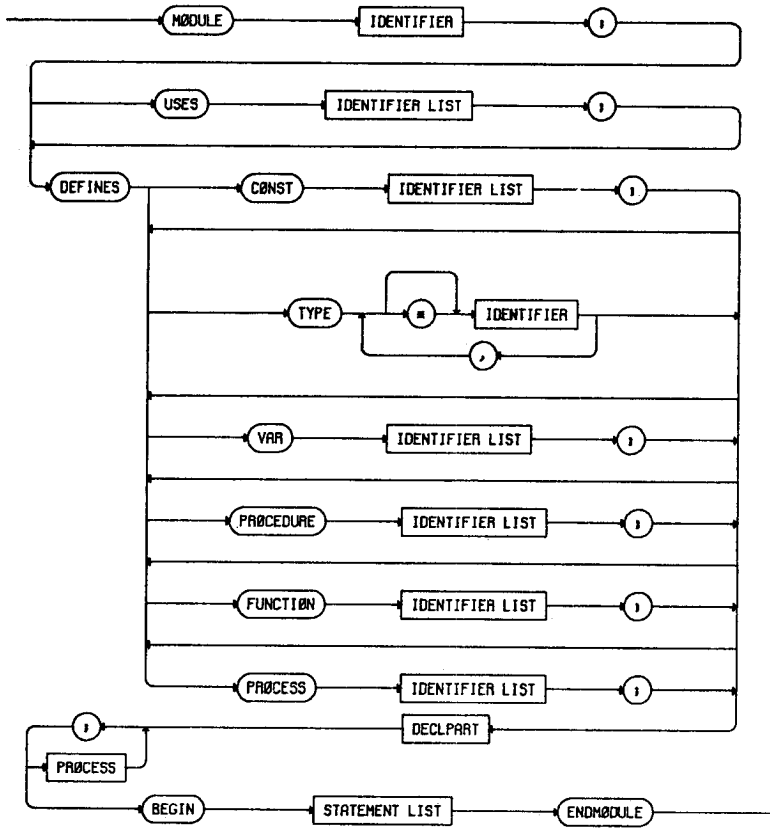


Abb. 2.2: Einfacher Modul

Ein Modul bildet keinen neuen Block wie ein Prozess oder eine Prozedur, er ist vielmehr eine syntaktische Klammer. Wie beim Prozess sind auch beim Modul alle global (zum Modul) vereinbarten und im Modul verwendeten Objekte und Prozeduren in einer Liste aufzuführen. Alle im Deklarationsteil des Moduls definierten Namen, die nicht in der durch <u>defines</u> eingeleiteten Liste der extern verwendbaren Namen aufgeführt sind, sind <u>lokal</u> zum Modul, d.h. sie sind ausserhalb des Moduls nicht bekannt.

Der Gültigkeitsbereich der extern verwendbaren Namen wird durch den Modul nicht beeinflusst, wohl aber die Art ihrer Verwendbarkeit:

1. Die Struktur von Typen ist ausserhalb des Moduls nur bekannt, wenn der Name eines Typs in der Liste der extern verwendbaren Namen mit einem Stern (*) gekennzeichnet ist; dann kann zu der Struktur des Typs zum Lesen zugegriffen werden.

2. Variable können ausserhalb des Moduls nur gelesen werden.

3. Konstante, Prozeduren und Funktionen unterliegen keinen Einschränkungen.

Die scharfe Begrenzung der Struktur eines Objektes auf einen Modul führt zu einer schwachen Verknüpfung des Moduls mit dem ihn umgebenden Programm und zu entsprechender Freiheit bei der Wahl der Darstellung von Strukturen innerhalb des Moduls, d.h. alle Aenderungen, die die Eigenschaften der nach aussen gegebenen Typen, Objekte und Operationen nicht beeinflussen, haben auch keinen Einfluss auf das umgebende Programm.

Ein Modul mit seinen inneren Objekten existiert solange wie der ihn umgebende Block. Die einen Modul initialisierenden Anweisungen werden implizit zu Beginn des den Modul enthaltenden Blocks oder Moduls ausgeführt.

Folgendes Beispiel zeigt einen Modul, der die Generierung und den Austausch von dynamischen Puffern kontrolliert, wobei ein Compiler prüfen kann, dass ausserhalb des Moduls keine Kopien von Zeigern gemacht werden, keine neuen "buffer" generiert und ihre Adressen einer Variablen vom Typ buffref (durch new(b)) zugewiesen werden, da beides die Zuweisung eines Wertes an eine nur zum Lesen freigegebene Variable bedingt.

```
type buffer = ...;

module bufferhandling;
  defines type *buffref;
          var nob;
          procedure gete, pute, initb, swap;

  const N = ...;
  type buffref= ↑buffer;
```

```
var nob: integer; {number of buffers in system}

procedure initb(var b: buffref);
begin b := nil end {initb};

procedure gete(var b: buffref);
begin if b = nil then {assign empty buffer}
      begin new(b); nob := nob + 1
      end
end {gete};

procedure pute(var b: buffref);
begin if b <> nil then
      begin dispose(b); nob := nob - 1; b := nil
      end
end {pute};

procedure swap(var a,b: buffref);
  var x: buffref;
begin x := a; a := b; b := x end {swap};

begin nob := 0
endmodule {bufferhandling}
```

Bemerkung: In Analogie zur Initialisierungsanweisung des Moduls könnte man eine Abschlussanweisung implizit am Ende des den Modul enthaltenden Blocks oder Moduls aufrufen, in der gewisse Abschlussoperationen ausgeführt werden, etwa die Ausgabe von Protokollinformation oder Weitergabe von gesammelter Information. Beim obigen Beispiel könnte etwa geprüft werden, ob am Ende nob=0, d.h. ob alle Puffer zurückgegeben wurden.

Gegenüberstellung von Class- und Modulstruktur

Eine Class-Struktur, wie sie in [P. Brinch Hansen 74] und [Hoare 74] definiert wird, lässt sich durch einen Modul folgendermassen beschreiben:

```
type T =

  class
    V: "Datenstruktur";
    "lokale Prozeduren";
    "public-Prozeduren P1..Pn"

  begin "initialisiere V"
  end {T}
```

```
module M:
  defines type T:
    procedure P1, ... Pn, Pinit;

  type T = "Datenstruktur";
  "lokale Prozeduren";
  "Prozeduren P1..Pn";
  procedure Pinit(var V: T);
  begin "initialisiere V" end;
endmodule {M}
```

Die durch den Class-Typ T vereinbarten Variablen werden implizit initialisiert (im Gegensatz zu Variablen mit anderen Typstrukturen), während beim Modul die Variablen vom Typ T explizit durch Aufruf der Prozedur Pinit initialisiert werden.

Eine Class-Struktur definiert immer nur einen Typ (mit zugehörigen Operationen), während durch einen Modul auch mehrere Typen mit entsprechenden Operationen, auch Operationen auf Variablen verschiedener Typen, eingeführt werden können.

Ein Modul, der nur einen Typ T definiert, kann durch zwei Class-Strukturen dargestellt werden, eine zur Definition von T und der Operationen auf T, die zweite zur Aufnahme der in den lokalen Variablen des Moduls gespeicherten Information:

```
module M:
  defines type T:
          procedure P1,...,Pn;
  type T = "Datenstruktur";
  var D: "Daten über Variable
          vom Typ T";
  "lokale Prozeduren";
  "Prozeduren P1,..Pn"
begin "initialisiere D"
endmodule {M}
```

```
D: class
     L: "Daten über Variable
         vom Typ T"
     "Operationen auf diesen
      Daten"
   begin "initialisiere D"
   end {D};
T = class extern D;
     V: "Datenstruktur";
     "lokale Prozeduren";
     "Prozeduren P1,...,Pn"
   end {T}
```

Die eigentlich lokal zum Modul M gehörende Information D muss

ausserhalb der entsprechenden Class-Struktur T vereinbart werden, da sonst für jede Variable (Instanz) vom Class-Typ T ein Feld L vereinbart wird.

Im Unterschied zur Class-Struktur werden im Modul Typen und Operationen syntaktisch verknüpft, ohne dass die Operationen in den Typ eingebettet werden. Diese Einbettung ermöglicht bei der Class-Struktur die implizite Initialisierung von Variablen, während die Trennung beim Modul die Verschachtelung von Moduln zur Darstellung lokaler Beziehungen ermöglicht; diese Verschachtelung ist bei Class-Strukturen nicht möglich, da dort mit jeder Instanz der äusseren Struktur auch eine Instanz der inneren Struktur geschaffen wird.

2.4. Der geschützte Modul

Der gegenseitige Ausschluss von Prozessen zur Ausführung einer oder mehrerer Operationen auf globalen Objekten wird durch folgende Erweiterung der Moduldefinition ermöglicht:

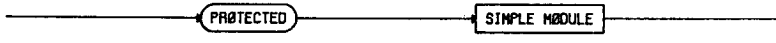


Abb. 2.3: Geschützter Modul

Da zu Variablen, die in einem Modul vereinbart oder von einem in einem Modul definierten Typ sind, nur über Modulprozeduren zugegriffen werden kann, ist es zur Verhinderung "gleichzeitigen" Zugriffs verschiedener Prozesse auf solche Variablen sicherlich hinreichend, wenn bei der Ausführung zu jedem Zeitpunkt höchstens eine Operation auf einer solchen Modulvariablen oder auf allen Variablen von einem in einem Modul definierten Typ ausgeführt werden kann.

Die Verknüpfung des Schutzes mit Operationen auf allen Variablen von im gleichen Modul definierten Typen bedeutet eine gewisse Einschränkung der Parallelität, die aber auf Anlagen mit nur einem

Prozessor und damit für die vorliegende Anwendung unwesentlich ist, während die Definition und die von einem Compiler auszuführenden Tests wesentlich einfacher sind als z.B. bei der von Brinch Hansen [74] gegebenen Definition des Monitors. Eine Verallgemeinerung wäre die Verknüpfung des Schutzes mit Objekten von einem in einem Modul zu definierenden Typ entsprechend den "shared variables" in [Brinch Hansen 73], wobei hier der exklusive Zugriff auf minimale kritische Bereiche beschränkt werden kann; durch die Beschränkung der Definition von geschützten Typen auf den Modul würden auch bei diesem Konzept alle kritischen Bereiche notwendigerweise in <u>einem</u> Modul zusammengefasst, würden aber nicht mehr mit Prozeduren zusammenfallen, wie dies beim Monitor bzw. beim geschützten Modul der Fall ist.

Durch den geschützten Modul können einzelne zu Prozeduren oder Funktionen zusammengefasste Operationen unter exklusivem Zugriff auf gewissen Daten oder Systemkomponenten (z.B. Drucker) ausgeführt werden. Wechselt eine Folge solcher Operationen mit Operationen auf lokalen Variablen eines Prozesses ab, werden zwei Moduln benötigt, wenn die Daten oder Systemkomponenten auch während der lokalen Operation reserviert bleiben sollen. Das folgende Beispiel zeigt Routinen zum Drucken eines Files, das nicht als Ganzes einem Driver übergeben werden kann, das aber ohne Einschub anderer Information gedruckt werden soll:

```
protected module printer;
  defines procedure printerrequest, printerrelease;
  ....
endmodule {printer};

protected module print;
  defines procedure printheader, printblock, printendfile;
  ....
endmodule {print}
```

Dabei müssen sich alle Prozesse an folgende, syntaktisch nicht prüfbare Aufrufsequenz halten:

```
...
printerrequest;
"Folge von Anweisungen mit
 printheader, printblock und printendfile";
printerrelease;
...
```

Wenn sich alle Prozesse an diese Sequenz halten, ist der Schutz des Moduls print eigentlich überflüssig; eine einfache zweite Erweiterung des Moduls beschreibt diese Situation besser (und ermöglicht effizienteren Code):

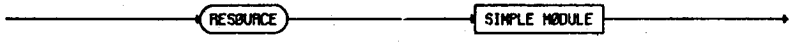


Abb. 2.4: Resource Modul

Die Anweisung der Form

<u>with</u> "module identifier" <u>do</u> S

erlaubt dabei einem Prozess P in der Anweisung S den exklusiven Zugriff zu den Objekten und Operationen eines Moduls, ohne dass jede Modul-Operation den Schutzmechanismus aktiviert, der mit der <u>with</u>-Anweisung verknüpft werden könnte. Diese zweite Art von geschützten Modulen wird im weiteren nicht verwendet werden, da die Problemstellung des Systems HEXAPUS eine feste Zuordnung von peripheren Geräten zu Prozessen erlaubt.

2.5. <u>Die Queue</u>

Neben dem gegenseitigen Ausschluss, der durch das Konzept des geschützten Moduls garantiert wird, soll die Queue die explizite Synchronisation von Prozessen ermöglichen. Die hier definierte Queue entspricht genau der von Hoare [73] eingeführten <u>condition</u>. Sie kann innerhalb eines geschützten Moduls vereinbart werden durch

```
var q : queue
```

und es können auf ihr folgende Operationen ausgeführt werden:

1. initq: Zur einmaligen Initialisierung.

2. enterq: Zur Unterbrechung der Ausführung einer Modulprozedur bei Nichterfülltsein einer gewissen Bedingung; der geschützte Modul wird dadurch zur Ausführung einer nächsten Prozedur freigegeben.

3. startq: Zur sofortigen Wiederaufnahme eines in der Queue wartenden Prozesses, wozu der startq aufrufende Prozess unterbrochen wird und dem wiedergestarteten Prozess den Modul freigibt.
   Auf eine leere Queue hat startq keine Wirkung.

Eine Queue mit so spezifizierten Operationen darf nicht ausserhalb eines geschützten Moduls definiert werden, wenn das relative zeitliche Verhalten von Prozessen für die richtige Kooperation keine Rolle spielen soll! Dies soll am Beispiel der Kooperation eines Verbraucherprozesses mit einem Erzeugerprozess gezeigt werden, die über einen einelementigen Puffer gekoppelt seien:

```
type item = ...;
 ...
module buffer;
  defines procedure get, put;

  var b: item; empty: Boolean;
      qin, qout : queue;

  procedure put(i: item);
  begin if not empty then {buffer full} enterq(qin);
    b:= i; empty:= false; startq(qout)
  end {put};

  procedure get(var i: item);
  begin if empty then {buffer empty} enterq(qout);
    i:= b; empty:= true; startq(qin)
```

```
    end {get};

  begin initq(qin); initq(qout); empty:= true
  endmodule {buffer}
```

Diese Routinen arbeiten nach folgender Strategie:

1. empty = true <=> Puffer leer.

2. Der Verbraucher wartet bei leerem Puffer in der Queue qout.

3. Der Erzeuger wartet bei vollem Puffer in der Queue qin.

4. Das Aendern des Pufferzustandes wird dem eventuell wartenden Prozess durch Aufruf von startq mit dem entsprechenden Parameter angezeigt.

Wenn die Bedingungen 1. bis 4. erfüllt sind, geht keine Information verloren, es wird keine "erfunden", und es kann kein Deadlock eintreten.

Bei der oben gegebenen Definition der Queue-Operationen sind die Aussagen 2. und 3. für den ungeschützten Modul buffer nicht richtig, denn wenn nach Feststellen eines leeren Puffers aber vor Betreten der Queue qout der "schnellere" Erzeuger ein Element liefert und signalisiert, geht das Signal verloren, da qout noch leer ist, und der Verbraucher wartet schliesslich in qout, obwohl der Puffer nicht leer ist. Beim Liefern des nächsten Elementes wird der Erzeuger wegen des noch vollen Puffers in qin warten.

Diese Deadlock-Situation kann bei einem geschützten Modul nicht eintreten, da wegen des gegenseitigen Ausschlusses die Aussagen 2. und 3. richtig sind.

Bemerkung: Bei Definition der Queue entsprechend dem binären Semaphore [Dijkstra 68], d.h. bei Speicherung eines startq im Falle einer leeren Queue, wären in diesem Beispiel mit 2 Prozessen 2. und 3. auch im ungeschützten Modul erfüllt; es gilt dann:

2'. Puffer leer <=> empty <u>or</u> signaled(qout)
Puffer voll <=> <u>not</u> empty <u>or</u> signaled(qin)

2.6. Steuerung peripherer Geräte: Der Driver Modul

Bei der direkten Steuerung peripherer Geräte ist neben dem Zugriff zu festen, den peripheren Geräten zugeordneten Puffern (Interfaces) auch das wirkliche zeitliche Verhalten eines Programmes von Bedeutung, es genügt hier nicht mehr das Ablaufen eines Prozesses mit positiver Geschwindigkeit zu garantieren. Bei einem Kartenleser ohne Kartenpuffer oder einer synchronen Verbindung fällt Information in gewissen vom peripheren Gerät abhängigen Zeitabständen an und muss bis zum Eintreffen der nächsten Information verarbeitet sein! Daher werden für solche zeitkritischen Instruktionsfolgen eines Prozesses je nach Implementation gewisse Operationen mit höherer Priorität auszuführen sein, gewisse syntaktische Einschränkungen werden aufgehoben oder zusätzlich eingeführt, und die entsprechende Verantwortung für die richtige Verwendung liegt dann beim Programmierer. Direkter Zugriff zu peripheren Geräten und die Ausführung zeitkritischer Operationen (queue-Operationen ausserhalb eines geschützten Moduls) kann nur in Moduln erfolgen, die durch die Präposition <u>driver</u> gekennzeichnet sind:

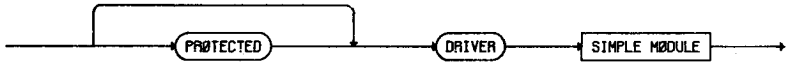


Abb. 2.5: Driver Modul

Um den Zugriff zu den Kommunikationspuffern peripherer Geräte zu ermöglichen, sollen vordefinierte Adressen PG1, PG2, ..., PGn nach abnehmender Priorität Referenzierung der entsprechenden Puffer erlauben. Die Zuweisung und Strukturbeschreibung dieser Puffer erfolgt durch folgende nur in Driver-Moduln erlaubte Vereinbarung:

```
device <identifier> = PGi : record "structure of
                                    communication area"
                                end
```

Die Anweisung

<u>under</u> <device identifier> <u>control</u> S

erlaubt in der Anweisung S die ausschliessliche Verwendung der Operationen

<u>procedure</u> start(<device identifier>)
<u>procedure</u> continue(<device identifier>)
<u>function</u> completed(<device identifier>): Boolean

Durch <u>start</u> wird das periphere Gerät aktiviert, durch <u>continue</u> wird der weitere Empfang von Daten ermöglicht, wenn auf ein start mehrere Informationseinheiten geliefert werden. Bei beiden Operationen wird der sie ausführende Prozess unterbrochen, bis das periphere Gerät den vollständigen bzw. teilweisen Abschluss seiner Aktion signalisiert. In Ausnahmesituationen kann auch ein unbeteiligter Prozess (nicht <u>under</u> <device identifier> <u>control</u>) mit der Operation

reset(<device identifier>)

die Aktion des peripheren Gerätes abbrechen und den unterbrochenen Prozess reaktivieren, damit dieser die Kontrolle für weitere Massnahmen erhält, durch die z.B. nicht mehr benötigte Systemeinheiten für die Verwendung durch andere Prozesse freigegeben werden. Mit der Funktion <u>completed</u> kann der betroffene Prozess feststellen, ob die Aktion des peripheren Gerätes normal abgeschlossen oder durch reset abgebrochen wurde.

Da in Prozessen oder Prozeduren zur Steuerung peripherer Geräte das zeitliche Verhalten ohnehin berücksichtigt werden muss, wird hier die Verwendung des Typs queue auch ausserhalb geschützter Modulen erlaubt. Verschiedene Prozesse dürfen hier auch zu globalen Variablen beliebigen (auch nicht in geschützten Modulen definierten) Typs zugreifen.

2.7. Sonstige Erweiterungen und Einschränkungen

(1) Um die durch Modul und Prozedur gewonnene Modularität auch ausnutzen zu können, wird bei der Vereinbarung der formalen Parameter einer Prozedur neben Typennamen auch die Struktur eines Array zugelassen, bei der der Indextyp nicht spezifiziert ist und als Subrange des Typs integer angenommen wird; die Dimension des Array und der Typ der Komponenten wird dagegen angegeben:

```
procedure p(buff: array [*] of item; low,up: integer);
...
```

(2) Die Gleichheit von Typen wird folgendermassen definiert:

1. Namenlose Typen mit gleicher Struktur sind gleich.

2. Benennte Typen sind gleich, wenn sie in einer Typdefinition gleichgesetzt wurden und gleichen Einschränkungen unterliegen (etwa durch Definition in einem Modul). Benannte und unbenannte Typen werden auch bei gleicher Struktur nicht als gleich betrachtet.

Damit wird erreicht, dass, wenn für einen Typ ein Name eingeführt wurde, alle weiteren Verwendungen sich auf diese eine Definition beziehen. Für die Definition von Typen in einem Modul folgt aus 2.:

```
type T = ...;
 ...

module m;
  defines type A; ...;

  type A = T;
   ...
  {im Modul m sind die Typen A und T gleich}

endmodule {m};

{ A <> T }
```

Damit wird erreicht, dass der Modul über Elemente von einem im Modul definierten Typ völlige Kontrolle hat; so kann z.B. syntaktisch geprüft werden, dass von Zeigern von in einem Modul definierten Typ ausserhalb des Moduls keine Kopien gemacht werden.

(3) Die repeat-Anweisung ist nach einem Vorschlag von Wirth [75] auf folgende Form erweitert worden:

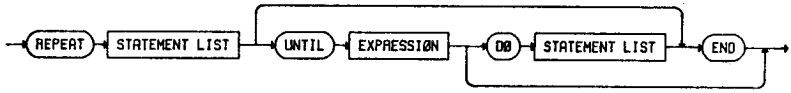


Abb. 2.6: repeat-Anweisung

Damit kann die Schleifen-Abbruchbedingung an irgendeiner Stelle zwischen den zu wiederholenden Anweisungen stehen und Fehlen dieser Anweisung ergibt einen (nicht abbrechenden) Zyklus (für zyklische Prozesse).

(4) Folgende in PASCAL vorhandenen Möglichkeiten werden nicht übernommen:

(a) Rekursion (und forward-Vereinbarungen von Prozeduren)
(b) Files
(c) Formale Prozeduren und Funktionen
(d) Der Typ real und zugehörige Operationen.

3. Eine Implementation der Sprachelemente

Sprachelemente wie Prozess, Modul usw. werden zur Ausführung auf gewisse Speicher- und Ablaufstrukturen abgebildet, die hier in statische und dynamische Komponenten getrennt dargestellt werden sollen. Die statische Abbildung erfolgt durch die Uebersetzung, während das dynamische Verhalten zur Ausführungszeit durch gewisse, in einem Nucleus zusammengefasste Routinen kontrolliert wird.

3.1. Statische Auflösung der Konstruktionen

Die statische Auflösung der Konstruktionen ist Aufgabe eines Compilers. Eine Konstruktion kann dabei Anlass geben zu

1. Eintragungen in Compilertabellen
2. Reservation von Speicherplatz
3. Erzeugung von Code.

Ausgehend von der Compilation von PASCAL-Programmen, wie sie von Wirth [71c] und Ammann [75] beschrieben wurden, soll hier die bei der Implementation von HEXAPUS verwendete Abbildung beschrieben werden, gleichzeitig als Skizze einer möglichen Compilererweiterung.

3.1.1. Prozesse und Prozeduren

Wie einem rein sequentiellen PASCAL-Programm wird jedem Prozess für seine Variablen und zur Aufnahme dynamisch anfallender Information (etwa beim Aufruf von Prozeduren in einem geschützten Modul) ein Stack zugeordnet. Bei der hier vorliegenden Implementation ohne Rekursion kann dabei die maximale Länge des Stacks für jeden Prozess bereits bei der Uebersetzung bestimmt werden.

Schliesst man ausserdem die nach Definition von Abschnitt 2.2 mögliche dynamische Generierung von Prozessen aus, d.h. verlangt

man bei der Compilation die Angabe der Maximalzahl gleichzeitig aktiver gleicher Prozesse, so kann die vollständige Speicherplatzzuordnung bereits bei der Uebersetzung bzw. beim Laden des Systems vorgenommen werden.

Start und Ende des Ablaufs eines Prozesses werden dem die Prozesse verwaltenden Nucleus durch den Aufruf der Prozeduren initprocs und termprocs angezeigt. Abb. 3.1 zeigt die bei der Uebersetzung eines Prozesses notwendigen Aktionen.

Struktur	Tabellen	Speicher	Code
...			
<u>process</u> P(...):		Prozess-	
	Eintragung	deskriptor	
"Vereinbarungen"	lokaler		
	Vereinbarungen	Parameter	
<u>begin</u>		+ lokale	
"Anweisungen"		Variable	Code(Anweisungen)
<u>end</u> {P};	Löschen der		termprocs
...	lokalen		
	Vereinbarungen		
...			
start P;		begrenzter	initprocs(P)
		Stack	

Abb. 3.1: Uebersetzung der Prozess-Struktur

Prozeduren werden zwar nicht rekursiv verwendet, können aber je nach Art der in ihnen vorkommenden Operationen und der damit verknüpften Prozessorzuordnung (siehe 3.2) von mehreren Prozessen aufgerufen werden, bevor die Aktionen für einen Prozess abgeschlossn sind, d.h. Prozeduren, die von verschiedenen Prozessen gleichzeitig aufgerufen werden können, müssen reentrant übersetzt werden; ihre lokalen Variablen werden auf dem Stack des die Prozedur aufrufenden Prozesses gespeichert.

Struktur	Art der Prozedur reentrant	sonst
procedure q(...);		
var ...		Reservation eines festen Speicherplatzes
begin	Platzreservation für lokale Variable auf dem Stack des aufrufenden Prozesses	
end {q};	Freigabe dieses Platzes	

Abb. 3.2: Speicherzuordnung für lokale Prozedurvariable

Je nach Art der Prozedur muss auch die Parameterübergabe auf dem Stack, in den Registern oder an einer festen Adresse geregelt werden.

3.1.2. Der Modul, der geschützte Modul und die Queue

Den in einem Modul vereinbarten Variablen wird bei der Uebersetzung ein fester Speicherplatz zugeordnet, Prozeduren werden entsprechend der Beschreibung in Abschnitt 3.1.1 behandelt.

```
...
module m;

"Vereinbarung innerer und nach aussen weiterzu-
 gebender Typen, Variablen und Prozeduren"

{1} begin {2}
      "Anweisungsfolge zur Initialisierung des Moduls"
    endmodule {3};
...
```

```
begin {4}
    ...
end
```

Abb. 3.3: Modulstruktur

Nach der Analyse des Moduls ({3}) können die Eintragungen in den verschiedenen Compilertabellen für innere Vereinbarungen gelöscht werden, Eintragungen nach aussen gehender Typen sind so zu kennzeichnen, dass die Verwendung von Variablen dieser Typen entsprechend der Definition in 2.3 und 2.4 kontrolliert werden kann, d.h. dass ihre Struktur nicht mehr zugreifbar ist, bzw. diese Variablen nur gelesen werden können.

Zu Beginn der Anweisungsfolge des den Modul umfassenden Blocks oder Moduls ({4}) wird ein Aufruf der zu einer anonymen Prozedur zusammengefassten Initialisierungsanweisung {1} generiert.

Beim <u>geschützten Modul</u> wird zusätzlich implizit eine innere Variable als Deskriptor vereinbart, für deren Initialisierung zu Beginn der Initialisierungsanweisung des geschützten Moduls {2} der entsprechende Code erzeugt wird (Aufruf der Nucleusprozedur initprot). Als erste bzw. letzte Anweisung einer von aussen aufrufbaren Modul-Prozedur werden implizit die Nucleus-Routinen enterprot bzw. return aufgerufen, die mittels Deskriptor dafür sorgen, dass zu jeder Zeit höchstens ein Prozess bei der Ausführung einer Prozedur in einem geschützten Modul ist.

Um diese Routinen einfach halten zu können, dürfen bei dieser Implementation von aussen aufrufbare Modulprozeduren innerhalb des Moduls nicht aufgerufen werden.

Bei Maschinen mit einem Prozessor sind solche Prozeduren automatisch geschützt, wenn während ihrer Ausführung der Prozessor keinem anderen Prozess zugewiesen wird, der Prozeduren im gleichen geschützten Modul ausführen will. Eine zusätzliche Schutzoperation wäre also nur notwendig, wenn in einem kritischen Bereich eine Prozedur in einem anderen geschützten Modul aufgerufen oder eine Queue-Operation ausgeführt würde.

Durch eine solche optimierende Methode werden die Schutzoperationen eng mit dem Algorithmus der Prozessorzuteilung verknüpft, die Wahl des Algorithmus wird eingeschränkt und kann ohne erneute Uebersetzung des ganzen Systems nicht geändert werden. Um eine klare, nur durch die angegebenen Routinen definierte Verbindung zwischen Nucleus und restlichem System zu erhalten, wird auf diese Optimierung verzichtet.

Einer Queue-Variablen wird wie jeder anderen Variablen ein Speicher zugewiesen, der zur Aufnahme einer Referenz zu den in der Queue wartenden Prozesse dient (siehe 3.2.2); die Operationen initq, enterq und startq werden direkt als Aufrufe entsprechender Nucleus-Prozeduren übersetzt.

3.1.3. Periphere Geräte

Die Vereinbarung peripherer Geräte (device ...) entspricht der Vereinbarung einer Record-Variablen, wobei aber die Adresse der Variablen durch die Angabe einer konfigurationsabhängigen vordefinierten Adress PGi (vgl. 2.6) bestimmt ist.

Die Anweisung

under dvc control S (*)

wird unter Verwendung von Nucleus-Routinen switchon und switchoff aufgelöst zu:

switchon(dvc); S; switchoff(dvc)

während die Operationen start(dvc), continue(dvc), completed(dvc) und reset(dvc) direkt als Nucleus-Routinen zur Verfügung stehen. Um die Implementation dieser Routinen möglichst einfach und effizient halten zu können, wird die Anweisung (*) folgenden Einschränkungen unterworfen:

1. Anweisungen der Art (*) dürfen nicht verschachtelt werden.

2. In der Anweisung S dürfen pause, enterq, startq und Prozeduren

eines anderen Moduls nicht aufgerufen werden.

Bedingung 1. bedeutet insbesondere auch, dass in S keine Prozeduren in einem mit der Präposition driver versehenen Modul aufgerufen werden dürfen!

3.2. Die Laufzeitunterstützung: Der Nucleus

Der Nucleus ist eine Sammlung von Routinen, die im Abschnitt 3.1 bereits erwähnt wurden und die für das dynamisch richtige Verhalten der eingeführten Konstruktionen verantwortlich sind. Ausserdem werden im Nucleus die Prozessoren einer beliebigen Zahl von Prozessen so zugewiesen, dass jeder Prozess einen eigenen virtuellen Prozessor hat, oder umgekehrt werden Folgen sequentiell auszuführender Operationen der verschiedenen Prozesse auf Folgen von Operationen abgebildet, die von jeweils einem Prozessor auszuführen sind. Da alle Rechner, die als Datenstation eingesetzt werden sollen, nur über einen zentralen Prozessor verfügen, soll im weiteren nur dieser Fall behandelt werden.

Die Beschreibung der im Nucleus zusammengefassten Routinen erfolgt in einer freien Erweiterung der in Kapitel 2 definierten Sprache. Informell werden verwendet:

```
type natural = <positive integer>;
     address = <address range>;

statement: goto <expression of type address>

function returnaddr(<proc or funct identifier>): address;
  delivers return address of last activation

function addr(<label or proc entry>): address;
  delivers address of label or procedure

procedure halt(i: natural);
  stops processor with state indication i
```

```
procedure push(x: <any type (represented in one word)>);
begin sp:= sp + 1; {increment stack pointer}
  sp↑:= x
end

procedure pop(var x: <any type (represented in one word)>);
begin x:= sp↑; sp:= sp-1 end

operations: integer + and - on pointer variables
```

3.2.1. Die Prozessorverwaltung

Auf dieser untersten Ebene der Prozessorzuordnung sollen alle Prozesse gleichberechtigt sein, d.h. jedem Prozess wird der Reihe nach der Prozessor für eine gewisse Zeit oder Sequenz von Operationen zugeordnet. Da jedem Prozess ein virtueller Prozessor mit einem Satz von Registern zur Verfügung stehen soll, müssen die Werte dieser Register bei jeder Unterbrechung eines Prozesses gerettet und bei erneuter Zuteilung des Prozessors wieder geladen werden. Geschieht die Unterbrechung nicht zeit- sondern strukturabhängig, d.h. wird der Prozessor nicht für ein bestimmtes Zeitintervall sondern für eine bestimmte Sequenz von Instruktionen zugeteilt, so kann der Punkt des Unterbruchs so gewählt werden, dass nur eine minimale Anzahl von Registern eine Bedeutung hat und nur deren Werte zu retten sind. Die Festlegung dieser Punkte kann dem Compiler übertragen werden; sie können auch mit Nucleus-Routinen verknüpft werden, die in einem Prozess explizit oder implizit durch die Auflösung von Sprachkonstruktionen aufgerufen werden. Zum Teil sind diese Punkte ohnehin mit Synchronisationsoperationen verknüpft, denn will z.B. ein Prozess eine Operation in einem geschützten Modul ausführen, in dem bereits ein anderer Prozess ist, so muss er auf die Freigabe des Moduls warten und kann den Prozessor abgeben.

Für das System HEXAPUS, das möglichst ohne Uhr auskommen soll (siehe 1.3), wird die zweite Methode gewählt, wobei die Zuordnung des Prozessors zu den Prozessen nur in bestimmten Nucleus-Routinen geändert wird.

Neben dieser normalen Zuordnung des Prozessors ist dabei noch die Zuordnung des Prozessors an bestimmte Prozesse durch Interrupt, d.h. zu gewissen Zeitpunkten, zu berücksichtigen; diese wird in Abschnitt 3.2.1.3 behandelt.

3.2.1.1. Zustände eines Prozesses und Prozess-Deskriptor. Prozesse, die im Nucleus verwaltet werden, können entsprechend den in 3.1 spezifizierten Operationen in einem der folgenden Zuständen (procsstate) sein:

1. ready: können bei Zuteilung des Prozessors weiterarbeiten.

2. irctrl: (interrupt control) können bei Eintreffen eines Interruptes eines bestimmten peripheren Gerätes weiterarbeiten.

3. delayed: verzichten für eine gewisse "Zeit" auf die Zuteilung des Prozessors, könnten sonst aber weiterarbeiten.

4. queuing: warten auf Signalisierung eines bestimmten Zustandes von einem anderen Prozess.

Uebergang von einem in den anderen Zustand geschieht durch bereits erwähnte Operationen, die entweder direkt im Programm oder durch die Auflösung entsprechender Konstruktionen aufgerufen werden:

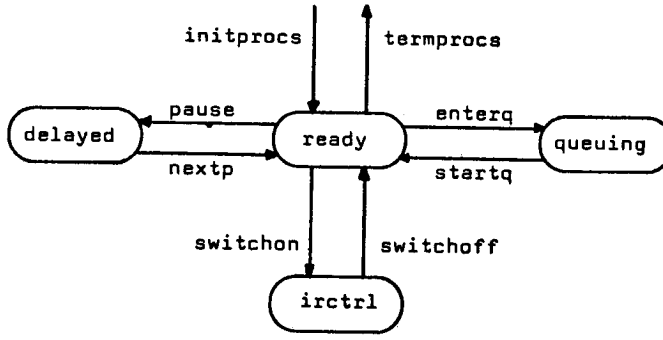


Abb. 3.4: Zustände eines Prozesses und Uebergangsoperationen

Diese Zustände werden zusammen mit weiterer zur Verwaltung eines

Prozesses notwendigen Information in einem Deskriptor folgenden Typs gespeichert:

```
type procsdescr = record state: procsstate;
                         time: natural;
                         nexti: address;
                         LMP, LSP: address;
                         next, nextpd: ↑procsdescr;
                         firstinstruction, stacklimit: address
                       end
```

Diese Deskriptoren können in einem Array oder einer Liste im Nucleus vereinbart werden, was aber bei der Entwicklung eines Systems (die bekanntlich selten abgeschlossen ist) und der damit verbundenen veränderlichen Zahl von Prozessen entweder häufige Compilation des Nucleus, grosszügig dimensionierte Tabellen oder dynamische Speicherverwaltung bedingt. Bei der vorliegenden Implementation belegen diese Deskriptoren als Stackmarke die ersten Plätze des Stacks eines Prozesses, für den der Platz bei der Compilation reserviert werden kann.

Die Nucleus-Routinen haben Zugriff zu diesen Deskriptoren, die (durch die Prozedur initprocs) in einer zirkulären Liste (mittels nextpd) verhängt sind. Der gerade aktive Prozess, d.h. der Prozess, dem gerade der Prozessor zugeordnet ist, wird durch einen Zeiger (crnt) identifiziert.

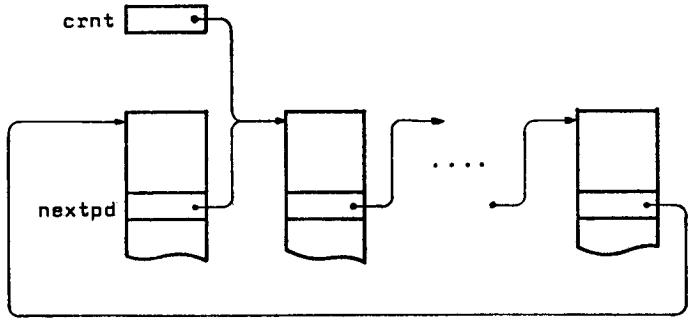


Abb. 3.5: Verkettete Liste der Prozessdeskriptoren

3.2.1.2. Normale Prozessorzuteilung. Durch die normale Zuordnung wird der Prozessor einem Prozess mit state = ready zugewiesen; beim Wechsel der Zuordnung sind im Deskriptor des bisher aktiven Prozesses die Adresse der für diesen Prozess als nächste auszuführende Instruktion (nexti) sowie die Werte der auf den Stack dieses Prozesses zeigenden globalen Systemvariablen (nach Möglichkeit Register) MP und SP zu speichern (LMP, LSP). SP zeigt auf die letzte Eintragung im Stack des durch crnt als aktiv gekennzeichneten Prozesses, MP zeigt auf die letzte Stackmarke (siehe 3.2.2.1) in diesem Stack. Der Prozessor wird dann (durch nextp) dem in der zirkulären Liste der Prozessdeskriptoren nächsten Prozess mit state = ready zugewiesen. Bei diesem Durchsuchen der Liste der Deskriptoren wird bei Prozessen, die im Zustand delayed gefunden werden, die Wartezeit (time) um 1 vermindert und bei Ablauf der Zeit ihr Zustand geändert (ready).

```
procedure nextp(ni: address);
begin crnt↑.nexti:= ni;
entry nextr;
  crnt↑.lmp:= mp; crnt↑.lsp:= sp;
  repeat
    repeat crnt:= crnt↑.nextpd
    until crnt↑.state in [ready,delayed];
    if crnt↑.state = delayed then
      with crnt↑ do
      begin time:= time-1;
        if time = 0 then state:= ready
      end
  until crnt↑.state = ready;
  mp:= crnt↑.lmp; sp:= crnt↑.lsp;
  goto crnt↑.nexti
end {nextp};

procedure pause(n: integer);
begin
  with crnt↑ do
    if n>0 then
    begin time:= n; state:= delayed end;
  nextp(returnaddr(pause))
end {pause}
```

3.2.1.3. Prozessorzuteilung durch Interrupt. Prozesse, die Operationen auf peripheren Geräten ausführen und auf den Abschluss dieser Operationen warten, sollen beim Eintreffen eines Interruptes möglichst schnell über den Prozessor verfügen können, um zeitkritische Operationen als Antwort auf den Interrupt ausführen zu können und/oder um das periphere Gerät wieder starten zu können. Damit die normale Prozessorzuteilung nicht zeitabhängig wird und damit bei einem Interrupt der Prozessor dem wartenden Prozess mit minimalem Verwaltungsaufwand zugewiesen werden kann, wird die Prozessorzuteilung bei Interrupt der normalen Prozessorverwaltung überlagert, d.h. bei einem Interrupt wird dem Prozess, dem der Prozessor gerade zugeteilt ist (crnt↑), der Prozessor entzogen, um einige Operationen des auf den Interrupt wartenden Prozesses auszuführen (vgl. cycle stealing), ohne dass crnt geändert wird.

Die beim Unterbruch zu rettende Information und die Referenz zu dem auf den Interrupt wartenden Prozess werden in einem für jedes periphere Gerät vorhandenen Device-Deskriptor untergebracht:

```
dvcsdescr = record pd: ↑procsdescr {waiting for interrupt};
                iret: address {of interrupted process};
                sret: address {of process waiting on interrupt};
                abnormal: Boolean;
                r: array [1..nr] of word {register values}
            end
```

Die hier dargestellte Struktur des Deskriptors stellt eine abstrakte Zusammenfassung der für jedes Gerät benötigten Daten dar. Bei der Implementation werden vom jeweiligen Rechner gegebene Möglichkeiten ausgenutzt, wie z.B. das Abspeichern des Instruktionszählers des unterbrochenen Prozesses an einer festen Adresse.

Kündigt ein Prozess durch switchon(dvc) die Verwendung eines peripheren Gerätes an, so wird sein Zustand geändert zu state = irctrl und im Deskriptor des peripheren Gerätes wird eine Referenz zu diesem Prozess eingetragen (pd); dieser Prozess behält dann den Prozessor bis zum ersten Aufruf von start(dvc) (Phasen 1 und 2 in Abb. 3.6), wodurch das periphere Gerät gestartet wird und der Prozess in die Kontrolle des Interruptsystems geht; der Prozessor

geht dabei durch die normale Prozessorverwaltung (nextp) an einen Prozess mit dem Zustand ready, da durch switchon eine entsprechende Adresse in iret eingetragen wurde.

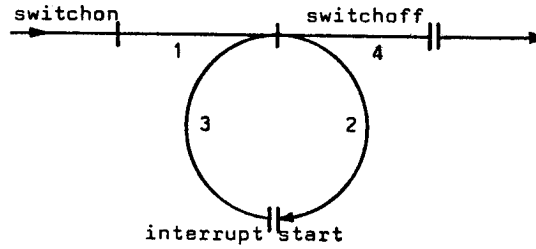


Abb. 3.6: Die vier Phasen unter Interrupt-Kontrolle

Bei weiteren Aufrufen von start(dvc) oder continue(dvc), die dann eine Sequenz von auf einen Interrupt folgenden Operationen abschliessen (Phasen 3 und 2), geht der Prozessor an den unterbrochenen Prozess zurück, nachdem die Register wieder auf die Werte gesetzt worden sind, die sie im Moment des Unterbruchs hatten; die Adresse, an der dieser Prozess unterbrochen wurde, wurde durch die Interrupt-Routine (interrupt(dvc)) zusammen mit den Werten der Register in iret und r gespeichert.

Die letzte auf einen Interrupt folgende Sequenz von Operationen (Phasen 3 und 4) wird durch die Routine switchoff(dvc) abgeschlossen, die den Prozess wieder der normalen Prozessorzuteilung unterstellt (state:= ready), das Interruptsystem für das periphere Gerät abstellt und dann den Prozessor wieder dem unterbrochenen Prozess zurückgibt.

Die Routine reset(dvc) schliesslich schaltet das Interruptsystem für das periphere Gerät dvc aus, prüft, ob ein Prozess auf einen Interrupt wartet, und löst ihn gegebenenfalls durch Aufruf der Interrupt-Routine aus, wobei dann im Deskriptor ein entsprechendes Flag (abnormal) gesetzt wird.

```
type peripheral = (PG1, ... , PGn);

var dvcs: array [peripheral] of dvcdescr;
 ...
procedure switchon(dvc: peripheral);
begin crnt↑.state:= irctrl;
  dvcs[dvc].pd:= crnt;
  dvcs[dvc].iret:= address(entry nextr)
    {initialize call of nextp at entry nextr upon first
     activation of peripheral device}
end {swichon};

procedure switchoff(dvc: peripheral);
begin dvcs[dvc].pd↑.state:= ready;
  dvcs[dvc].pd↑.nexti:= returnaddr(switchoff);
  "registers":= dvcs[dvc].r;
  goto dvcs[dvc].iret
end {switchoff};

procedure start(dvc: peripheral);
begin dvcs[dvc].sret:= returnaddr(start);
  "registers":= dvcs[dvc].r;
  "start device and enable interrupt";
  goto dvcs[dvc].iret;
end {start};

procedure continue(dvc: peripheral);
begin dvcs[dvc].sret:= returnaddr(continue);
  "registers":= dvcs[dvc].r; "enable interrupt";
  goto dvcs[dvc].iret
end {continue};

procedure interrupt(dvc: peripheral; abnorml: Boolean);
begin
  with dvcs[dvc] do
  begin r:= "registers"; abnormal:= abnorml;
    iret:= returnaddr(interrupt);
    goto sret
  end
end {interrupt};
```

```
function completed(dvc: peripheral): Boolean;
begin completed:= dvcs[dvc].abnormal
end {completed};

procedure reset(dvc: peripheral);
begin "disable interrupt and stop dvc";
  interrupt(dvc, true)
end {reset}

{the interrupt locations are to be
 initialized to 'interrupt(dvc,false)'}
```

Bemerkung: Je nach Art des Interruptsystems eines Rechners werden die Routinen start, continue, interrupt, usw. nicht mit dem Parameter dvc implementiert, sondern für jedes Gerät als separate Routine oder als Macro.

3.2.1.4. Bedingungen für die Korrektheit dieser Prozessorzuordnung. Für den aus dem normalen und dem interruptabängigen Teil zusammengesetzten Algorithmus ist zu zeigen, dass jedem Prozess ein virtueller Prozessor zugewiesen wird, d.h. dass ihm für gewisse Zeitintervalle oder Folgen von Operationen der Prozessor zur Verfügung gestellt wird, wenn er ihn benötigt (starvation excluded).

Durch die normale Prozessorzuordnung wird jedem Prozess der Reihe nach der Prozessor zugeteilt, wenn er ihn benötigt (state=ready) und wenn der Vorgänger ihn freigibt. Es ist also dafür Sorge zu tragen, dass kein einzelner Prozess den Prozessor monopolisiert, d.h. ihn höchstens für Operationen von Prozessen unter Kontrolle peripherer Geräte (mit höherer Priorität) abgibt. Dazu genügen die Bedingungen:

(1) Synchronisationsoperationen garantieren den Wechsel der Prozessorzuteilung, und zwar sequentiell.

(2) In jeder nicht a priori endlichen Schleife (for-Anweisung) muss (nach endlich vielen Durchläufen) entweder eine Synchronisationsoperation vorkommen oder pause aufgerufen werden.

Bedingung (1) richtet sich an die Implementation der Nucleus-Routinen für die Synchronisation von Prozessen, während (2) von einem Compiler oder Programmierer einzuhalten ist.

Die streng sequentielle Zuordnung des Prozessors an die Prozesse wird durchbrochen durch die interruptabhängige Zuordnung, durch die es möglich wäre, dass verschiedene Prozesse unter peripherer Kontrolle den Prozessor monopolisieren. Dies wäre der Fall, wenn immer während oder nach Ausführung einer auf einen Interrupt folgenden Sequenz von Befehlen bereits wieder ein Interrupt für einen anderen Prozess anfällt, da dann der Prozessor mit höherer Priorität zugeordnet wird.

Bedingung (3) ist hinreichend (aber nicht notwendig) für die Verhinderung dieser Art der Monopolisierung:

(3) Ein Prozess darf unter peripherer Kontrolle keine unbeschränkte Schleife (Zyklus) haben.

Denn unter der Annahme, der Prozessor würde von nicht zyklischen Prozessen unter peripherer Kontrolle monopolisiert, würde einer dieser Prozesse nach dem anderen wieder der normalen Prozessorverwaltung unterstellt, während wegen der Monopolisierung kein weiterer Prozess von der normalen in die interruptabhängige Zuordnung wechseln könnte. Somit wäre schliesslich kein Prozess mehr unter peripherer Kontrolle, und der Prozessor würde im Widerspruch zur Annahme Prozessen durch die normale Prozessorverwaltung zugewiesen.

Die Trennung der Prozessorzuordnung in einen zeitunabhängigen Teil (nextp) und einen zeit- bzw. interruptabhängigen Teil bewirkt, dass für die Ausführung <u>keiner</u> der bisher beschriebenen Routinen das Interrupt-System ausgeschaltet werden muss.

3.2.2. Routinen für die Synchronisation von Prozessen

Die Routinen für die Synchronisation müssen garantieren, dass jeweils höchstens ein Prozess eine Prozedur eines geschützten Moduls ausführt und somit ausschliesslichen Zugriff auf die dort

vereinbarten Variablen hat, während eventuell weitere Prozesse bis zur Freigabe des geschützten Moduls in einer Queue warten müssen. Diesem Schutzmechanismus überlagert sind die Routinen der Queue-Operationen, durch die Prozesse sich gegenseitig synchronisieren können.

Queue und Deskriptor für einen geschützten Modul haben folgende Strukturen:

```
type  queue = record  first,last: ↑procsdescr
              end;

      protdescr = record  free: Boolean;
                          pqueu: queue
                  end
```

Variable dieser Typen werden initialisiert durch:

```
procedure initq(var q: queue);
begin q.first:= nil
end {initq};

procedure initprot(var ptd: protdescr);
begin initq(ptd.pqueu); ptd.free:= true
end {initprot}
```

Eine Queue referenziert eine über das Feld next im Prozessdeskriptor verkettete Liste von Prozessen; first zeigt auf den ersten, last auf den letzten Deskriptor einer solchen Kette. Angehängt wird ein (Prozess-) Deskriptor durch die Routine append:

```
procedure append(var q: queue);
begin crnt↑.state:= queuing; crnt↑.next:= nil;
  with q do
  begin
    if first=nil then first:= crnt else last↑.next:= crnt;
    last:= crnt;
  end
end {append}
```

3.2.2.1. Gegenseitiger Ausschluss. Da die Routinen enterprot und return als erste und letzte Operation einer Prozedur in einem geschützten Modul ausgeführt werden, müssen die Rücksprungadresse und die Parameter reentrant behandelt werden, da mehrere Prozesse "gleichzeitig" eine Routine in einem geschützten Modul aufrufen und die erste Operation enterprot ausführen können. enterprot rettet daher die Rückkehradresse und (Referenz zu den) Parameter(n) auf den Stack, bevor der Zustand des Schutzes mittels Deskriptor geprüft wird und ein Prozess entweder die Prozedur ausführen kann und den Modul blockiert (free:= false) oder in der zum Deskriptor gehörenden Queue (pqueu) wartet, bis durch die zum Nucleus lokale Routine leaveprot der eine Modulprozedur ausführende Prozess den Modul dem nächsten (wartenden) Prozess freigibt.

```
procedure enterprot(ptd: ↑protdescr; vars,retaddr: address);
begin push(retaddr); push(mp); push(vars); mp:= sp;
  push(ptd);
  if sp>crnt↑.stacklimit then halt(2);   {stack overflow}
  if ptd↑.free then ptd↑.free:= false else
  begin append(ptd↑.pqueu);
    nextp(returnaddr(enterprot));
  end
end {enterprot}
```

Durch den Aufruf einer Prozedur in einem geschützten Modul wird auf den Stack des aufrufenden Prozesses durch enterprot also folgende Stackmarke geschrieben:

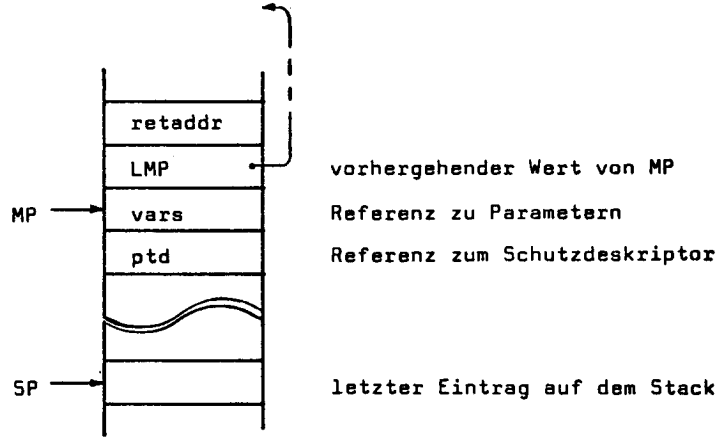


Abb. 3.7: Stackmarke durch Betreten eines geschützten Moduls

```
procedure leaveprot;
begin
  if mp<>crnt then {process in protected module}
    with (mp+1)↑ do
      if first = nil then (mp+1)↑.free:= true else
      begin first↑.state:= ready;
        first:= first↑.next
      end;
end {leaveprot}

procedure return;
  var retaddr: address;
begin leaveprot; sp:= mp-1; pop(mp); pop(retaddr);
  nextp(retaddr)
end {return}
```

Es ist nun noch zu zeigen, dass mit diesen Routinen und der gewählten Prozessorzuordnung nicht zwei Prozesse gleichzeitig eine Prozedur im gleichen Modul ausführen können, dass jeder Prozess in einen geschützten Modul gelangen kann, und dass die Prozessorverwaltung nicht gestört wird.

1. Die Routinen zum Schutz des Moduls werden von dem einen

Prozessor streng sequentiell ausgeführt (Bedingung (2) in 3.1.3), d.h. Operationen auf Deskriptoren werden nur unter normaler Prozessorzuteilung (nextp) ausgeführt und enthalten keine kritischen Bereiche.

2. Falls free = true, wartet kein Prozess auf Ausführung einer Modulprozedur, ein Prozess kann sofort eine Prozedur ausführen. Bei Freiwerden des Moduls kann direkt der nächste (am längsten) wartende Prozess eine Modulprozedur ausführen, da der Modul bei wartenden Prozessen nach Abschluss einer Prozedur nicht als frei gekennzeichnet wird und somit unabhängig von der weiteren Prozessorzuordnung kein anderer Prozess die bereits wartenden Prozesse überholen kann.

3. Die Routinen enterprot und return beeinflussen die Prozessorzuteilung nur durch Aufruf von nextp und erfüllen gleichzeitig die Bedingung (1) von 3.2.1.4, indem bei jeder Ausführung einer Prozedur in einem geschützten Modul mindestens beim Verlassen (return) durch nextp der Prozessor einem anderen Prozess zugewiesen wird.

3.2.2.2. Queue-Operationen. Entsprechend ihrer Definition in 2.5 sind die Queue-Operationen enterq und startq der Implementation des geschützten Moduls anzupassen.

Durch enterq ist der Deskriptor des aktiven Prozesses an eine Queue anzuhängen, der geschützte Modul ist freizugeben und der Prozessor einem anderen Prozess zuzuteilen. Mit den bereits beschriebenen Routinen append, leaveprot und nextp ergibt sich

```
procedure enterq(var q: queue);
begin
  append(q);
  leaveprot;
  nextp(returnaddress(enterq));
end {enterq}
```

Startq unterbricht im Falle einer nicht-leeren Queue den aktiven Prozess, setzt ihn an den Anfang der Queue im Modulschutz-Deskriptor und überlässt den Modul dem ersten in der Queue

wartenden Prozess. Dieser wird in den Zustand ready gebracht und dann durch den normalen Prozessorzuteilungs-Algorithmus aktiviert:

```
procedure startq(var q: queue);
  {a reference to the protection descriptor is part of the
   last stack mark of the active process}
  var wp: ↑procsdescr;
begin
  if q.first <> nil then
  begin wp:= q.first; q.first:= wp↑.next;
    with crnt↑ do
    begin next:= (mp+1)↑.pqueu.first;
      (mp+1)↑.pqueu.first:= crnt;
      if wp↑.next= nil then (mp+1)↑.pqueu.last:= crnt;
      state:= queuing
    end;
    wp↑.state:= ready;
    nextp(returnaddr(startq))
  end
end {startq}
```

Bemerkung: Falls die Vereinbarung und Verwendung von Queue-Variablen auch ausserhalb geschützter Moduln erlaubt ist, sind die Routinen enterq und startq entsprechend zu modifizieren.

Durch diese Implementation von startq und enterq ist sichergestellt, dass die Prozesse in der Reihenfolge aus einer Queue ausgelöst werden, in der sie in die Queue gelangten, d.h. ein Prozess kann in der Queue nicht von anderen Prozessen überholt werden und trotz entsprechend häufiger Aufrufe von startq beliebig lange in einer Queue warten müssen.

Das Einschieben eines Prozesses an den Anfang von pqueu im Modul-Deskriptor beeinflusst weder das Verhalten der Operationen enterprot und leaveprot noch die Argumentation im letzten Abschnitt über deren korrektes Verhalten, da durch startq kein zusätzlicher, sondern ein sich bereits im geschützten Modul befindender Prozess in die Queue eingeschoben wird.

Bemerkung: Wenn startq die letzte Operation einer Prozedur in einem geschützten Modul ist, könnte man dieses Einschieben des Prozesses in pqueu vermeiden, da bei anschliessendem Freiwerden des Moduls dieser Prozess dann den Modul nur noch betritt, um ihn wieder zu verlassen.

Bei automatischer Uebersetzung könnte die Prozedur startq um einen Parameter erweitert werden, durch den der Compiler angeben kann, welcher Fall vorliegt. Die Beschränkung, die Prozedur startq nur als letzte Anweisung einer Prozedur in einem geschützten Modul zuzulassen [Brinch Hansen 74], führt zu einer aufwendigen Syntaxanalyse, da die Einhaltung dieser Bedingung unbedingt geprüft werden müsste.

4. Das System HEXAPUS

Aus der in Abschnitt 1.3 gegebenen Problemstellung werden zunächst die Strukturen der Information und des Informationsflusses abgeleitet, dann ihre Implementation mittels der in Kapitel 2 angegebenen Sprache dargelegt, bevor schliesslich im nächsten Kapitel die Behandlung der peripheren Geräte beschrieben wird.

Die Darlegung des Systems HEXAPUS für die Datenstation erfolgt "top-down", d.h. umfassendere Strukturen werden immer weiter verfeinert und schliesslich in der in Kapitel 2 definierten Sprache dargestellt; zu verfeinernde Anweisungen werden dabei in Programmskizzen in Anführungszeichen gesetzt und eventuell in weiteren Programmstücken ausgeführt.

4.1. Struktur des Systems und der Information

Jedes Ein- oder Ausgabegerät am Kleinrechner behandelt einen Strom von Daten, der in einzelne Files gegliedert ist; diese Files stellen die Informationseinheit dar, die durch das System von einem Eingabegerät zu einem Ausgabegerät (im bezug auf die Datenstation) übermittelt werden kann, d.h.

> Ein Eingabestrom E[i] ist eine Sequenz von Files (von irgendwelchen Zeichen), die vom System (der Datenstation) nach bestimmten Spezifikationen eingelesen und bearbeitet wird; die einzelnen Files verlassen das System als Teil eines Ausgabestromes A[j].

Beginnt in einem Eingabestrom ein neues File, so benötigt das System die Angabe eines Ausgabestromes A[z] und eventuell Spezifikationen, wie die Information des Files zu interpretieren bzw. umzuformen ist.

Im allgemeinen ist diese Steuer-Information nicht Teil der als File zusammengefassten Information und kann bei Speichermedien wie

Bänder und Platten auch nur umständlich (oder mit Hilfe eines Systems) zusammen mit der Imformation eingelesen werden. Neben den Eingabeströmen E[i] existiert daher noch ein Befehlsstrom B, der die für das System notwendigen Spezifikationen als Folge von Befehlen liefert. Da diese Befehle sich auf verschiedene Ströme beziehen können, muss jeder Befehl mindestens einen Strom als Adressaten enthalten und bezieht sich auf das gerade behandelte oder, falls keines vorhanden, auf das nächste einzulesende File, d.h. die Zuordnung von Befehl zu File ist zeitabhängig. Ein Meldungsstrom M enthält alle Meldungen von den verschiedenen Strömen an den Benutzer.

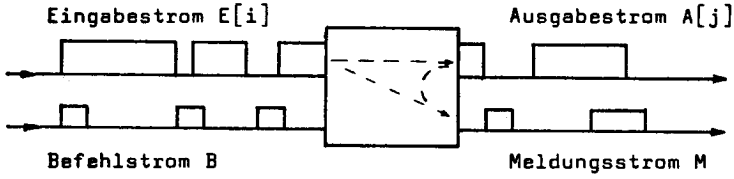


Abb. 4.1: Die vier Arten von Strömen

Mit Eingabeströmen

CRD vom Kartenleser, RPT vom Lochstreifenleser
und LI1, ... , LIn von der Linie

und Ausgabeströmen auf

Drucker (PRT), Lochstreifenstanzer (PPT) und
Linie (LO1, ..., LOm)

entspricht diese Darstellung den Forderungen (1), (3) und (4) der Problemstellung, wobei zusätzliche periphere Geräte zu weiteren Ein- oder Ausgabeströmen führen würden. Die Unterscheidung in Forderung (3) zwischen lokalen und externen Verbindungen entfällt auf dieser Stufe, da jede Kombination von Ein- und Ausgabestrom zugelassen werden kann. Der gleichzeitige Informationsaustausch zwischen Zentralrechner und mehreren peripheren Geräten wird durch die Existenz verschiedener Linienströme ermöglicht.

Um Information bzw. Spezifikationen für diese Liniendatenströme und

auf dem Zentralrechner auszuführende Befehle aus dem Befehlsstrom B zwischen Datenstation und Zentralrechner austauschen zu können, werden zwei weitere Linienströme eingeführt:

LOØ der Linienbefehlsstrom und
LIØ der Linienmeldungsstrom.

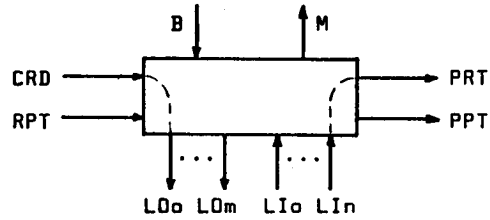


Abb. 4.2: Datenströme einer Normalstation

Abb. 4.2 zeigt Daten-, Befehls- und Meldungsströme für eine Normalstation, wobei das Rechteck für einen Mechanismus zur Kontrolle der Informationsübertragung von Eingabe- zu Ausgabeströmen und zur Steuerung des Datenflusses der einzelnen Ströme steht.

Der Forderung (2) der Problemstellung nach einfacher Bedienung einer Datenstation kann hier insoweit Rechnung getragen werden, als die zur stapelweisen Verarbeitung der Datenstation notwendigen (in Abb. 4.2 gestrichelten) Verbindungen vom System ohne zusätzliche Eingriffe des Benutzers initialisiert werden.

Forderung (5) nach Einflechten von Information eines Dateneingabestromes in den Eingabestrom CRD bedingt eine Erweiterung des in Abb. 4.2 dargestellten Systems von Datenströmen. Ein interner (Ausgabe-) Strom CRDO ermöglicht die Verbindung irgendeines Eingabestromes zum Strom CRD und das Mischen dieser Ströme zu einem (Eingabe-) Strom CRD'; in der Normalstation wird von dieser Möglichkeit nur zur Verknüpfung von RPT und CRD Gebrauch gemacht:

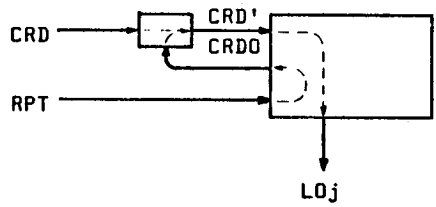


Abb. 4.3: Mischen zweier Eingabeströme

Bemerkung: Im weiteren Verlauf der Arbeit wird zwischen CRD und CRD' nur dann unterschieden, wenn das Problem des Mischens behandelt wird.

4.1.1. Zuordnung peripherer Geräte zu Datenströmen

Im allgemeinen wird jedes periphere Gerät entweder Daten für einen Eingabestrom liefern oder Daten eines Ausgabestromes ausgeben. Bei drei Geräten ist diese einfache Zuordnung nicht möglich: bei der Konsole (1), beim Kartenlesers (2) und bei der Linie (3).

(1) Ueber die Konsole werden Befehle und Meldungen ein- bzw. ausgegeben.

Für jedes von einem peripheren Gerät her einzulesende File muss zu Beginn bekannt sein, zu welchem Ausgabestrom es zu übermitteln ist. Diese Angaben werden dem Befehlsstrom entnommen. Bei peripheren Geräten, bei denen pro File eine Einheit eines Speichermediums einzulegen ist (z.B. Streifen oder Band), können diese Befehle gut für jedes File von der Konsole eingegeben werden und dienen gleichzeitig als Quittung für das Einlegen.

(2) Beim Kartenleser können mehrere Files (Kartenpakete) eingelegt werden und sollen im Normalfall der Stapelverarbeitung ohne Befehl einem geeigneten Linienausgabestrom zugewiesen werden. Um diesen Normalfall von einer speziellen Verbindung ohne Blockieren des Kartenlesers bis zur Eingabe eines Befehls von der Konsole her unterscheiden zu können, wird ein Kartenfile für eine spezielle Verbindung um eine erste Befehlskarte mit den entsprechenden Angaben erweitert.

Durch solche in ein Kartenpaket eingeschobene Befehlskarten kann dann auch die Verknüpfung des Eingabestromes CRD über den Ausgabestrom CRDO mit einem anderen Eingabestrom gesteuert werden.

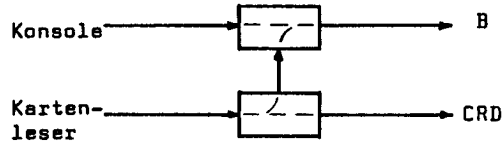


Abb. 4.4: Relation von Kartenleser und Konsole zu den Strömen CRD und B

(3) Die Linienströme sind entsprechend Punkt (7) der Problemstellung über <u>eine</u> Leitung zu übertragen, d.h. die (internen) Linienein- und -ausgabeströme sind auf je einen in einer Richtung zu übertragenden (externen) Datenstrom (LI und LO) abzubilden. Die Information dieser Ströme LI und LO kann dann entweder abwechselnd über eine halb-duplex oder gleichzeitig über eine voll-duplex-Leitung übermittelt werden.

Um die "<u>gleichzeitige</u>" Uebermittlung von Files mehrerer Datenströme in einer Richtung zu ermöglichen, werden die Files von der Sendestation jeweils in Blöcke zerlegt und es werden Blöcke der verschiedenen Ströme (nach verschiedenen Prioritäten) abwechselnd übertragen. Die Empfängerstation setzt aus diesen Blöcken wieder die ursprünglichen Files zusammen; Linienbefehle und -meldungen sollen dabei mit höchster Priorität übermittelt werden.

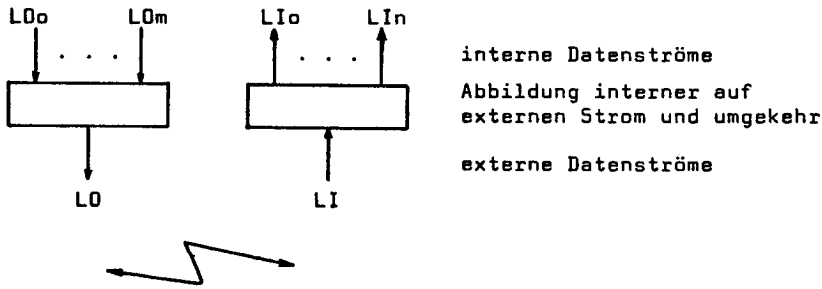


Abb. 4.5: Zusammenfassung der Linienströme zur Uebermittlung

Für eine Datenstation ergibt sich somit unter Berücksichtigung peripherer Geräte und Datenströme die in Abb. 4.6 gezeigte Struktur.

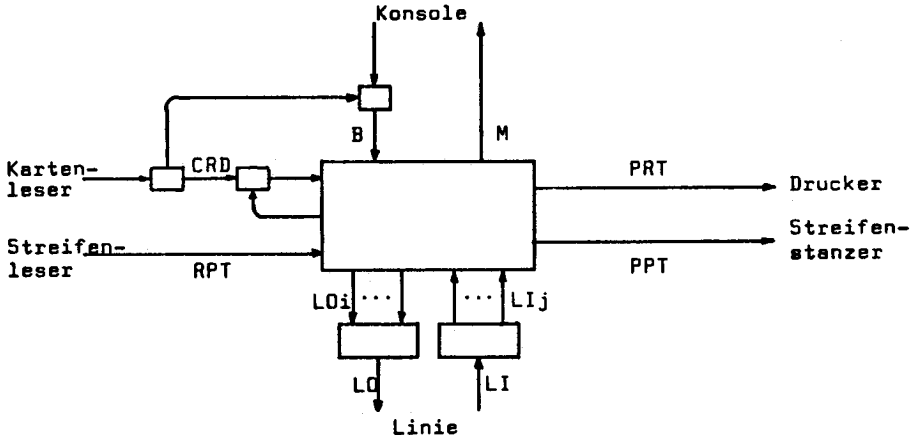


Abb. 4.6: Struktur der Ströme in einer Datenstation

4.1.2. File-Struktur und Verschlüsselung der Information

Da Files hauptsächlich auf der Zentralanlage verarbeitet bzw. erstellt werden, wird für das System HEXAPUS direkt die CDC-File-Struktur übernommen; diese umfasst:

1. Mehrstufen-Files,
2. Files als Komponenten von Mehrstufen-Files,
3. Segmente als Komponenten von Files,

und für Text

4. Zeilen.

Das File als Komponente eines Datenstromes entspricht bei dieser Bezeichnung dem Mehrstufen-File.

Bei der Information wird zwischen binärer Information und Text unterschieden, wobei ein Segment nur Information einer der beiden

Arten enthalten kann.

Information wird von den peripheren Geräten im allgemeinen mit jeweils verschiedener Verschlüsselung geliefert bzw. verlangt; um diese geräteabhängige Verschlüsselunge aber so lokal wie möglich zu halten, wird die Information aller Datenströme vom Einlesen bis zur Ausgabe oder Verarbeitung in ASCII-Code dargestellt, d.h. eventuell notwendige Code-Transformationen werden bei der Ein- oder Ausgabe ausgeführt. Binäre Information wird dabei auf die druckbaren Zeichen (> 37 oktal) abgebildet, damit die ASCII-Steuerzeichen eindeutig für die Darstellung der Struktur und Steuerung der Uebermittlung von Information über die Linie zur Verfügung stehen. Um binäre Segmente von anderen unterscheiden zu können, wird jedes Segment um ein die Art spezifizierendes Steuerzeichen erweitert; da der Zentralrechner zwei Darstellungen für Text erlaubt (ASCII- und interner Display-Code), kann ausserdem die im Zentralrechner gewünschte Darstellung angegeben werden:

AD: ASCII-codiertes Segment, die Information ist im Zentralrechner auf internen Code abzubilden.

AA: ASCII-codiertes Segment; Codierung ist im Zentralrechner beizubehalten.

BB: Binär codiertes Segment.

Die File-Strukturen werden durch zusätzliche Zeichen für den Abschluss einer Zeile, eines Segmentes, Files oder Mehrstufen-Files angegeben:

EOI: (=34B) Abschluss eines Mehrstufen-Files (beendet auch File, Segment und Zeile).

EOF: (=35B) Abschluss eines Files (beendet auch Segment und Zeile).

EOR: (=36B) Abschluss eines Segmentes (beendet auch Zeile).

CR,LF: Abschluss einer Zeile.

4.2. Implementationsstrategie

In diesem und den folgenden Abschnitten sollen Algorithmen zur Behandlung und Kontrolle von Datenströmen sowie für die Uebermittlung von Information von Eingabe- zu Ausgabeströmen in der in Kapitel 2 definierten Sprache entwickelt und dargestellt werden.

Die dynamische Behandlung zur Ein- bzw. Ausgabe von Datenströmen und die dazu erforderliche Kontrolle der peripheren Geräte soll dabei durch Prozesse beschrieben werden. Im einfachsten Fall, z.B. dem des Streifenlesers, behandelt ein Prozess das periphere Gerät und liefert einen Eingabestrom. Beim Kartenleser liefert dagegen der Prozess zur Behandlung des Kartenlesers den Datenstrom CRD und gleichzeitig auch Teile des Befehlsstromes B (s. Abb. 4.4). Wird im gleichen Prozess auch noch der Ausgabestrom CRDO behandelt, was ohne Effizienzverlust möglich ist, da während der Uebernahme von Information aus CRDO in den Strom CRD' der Fluss des Datenstromes CRD und damit der Kartenleser selbst unterbrochen wird, so behandelt dieser Prozess Teile von zwei Eingabeströmen und einen Ausgabestrom:

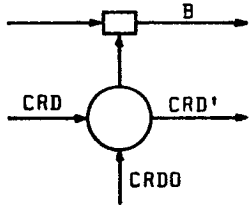


Abb. 4.7: Kartenleseprozess und behandelte Ströme

Wird jeder Datenstrom von einem Prozess vollständig erfasst, so kann die Uebertragung eines Files von einem Eingabe- zu einem Ausgabestrom durch eine gewisse Synchronisation der entsprechenden Prozesse implementiert werden. Behandelt ein Prozess mehrere Ströme, so wird durch einen Unterbruch für die Synchronisation für einen Strom auch der Fluss der anderen, durch den gleichen Prozess behandelten Ströme unterbrochen.

Bei der Behandlung der internen Liniendatenströme ist diese

Kopplung, d.h. die Behandlung durch einen Prozess, sicher zu vermeiden, da sonst durch eine Synchronisationsoperation die Uebertragung von Information über die Linie völlig blockiert wird. Wird also jeder interne Linienstrom durch einen eigenen Prozess im System vertreten und behandelt ein zusätzlicher Prozess die Abbildung der Linienströme auf einen externen Strom, so ergibt sich unter Verwendung geeigneter Moduln die Abb. 4.5 entsprechende Abb. 4.8:

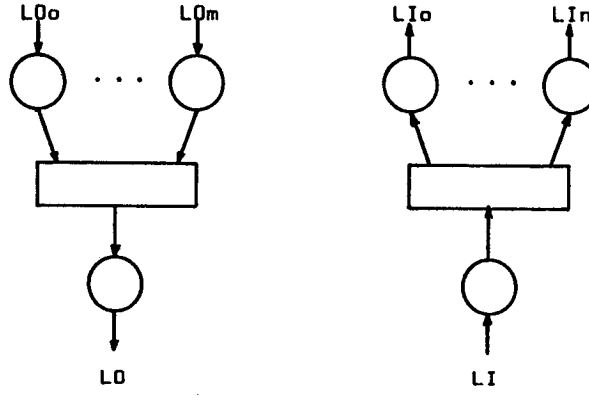


Abb. 4.8: Linienprozesse und Linienströme

Für das gesamte System gilt damit:

jeder Datenstrom wird durch <u>einen</u> Prozess, den <u>zugeordneten</u> Prozess, dargestellt

und es ergibt sich folgende, Abb. 4.6 entsprechende Struktur:

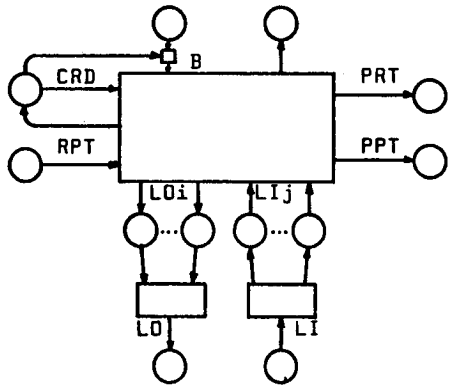


Abb. 4.9: Prozesse und Ströme einer Datenstation

Bei der Uebertragung von Information von einem Eingabe- zu einem Ausgabestrom liegt für die die Ströme behandelnden Prozesse die typische Erzeuger-Verbraucher-Situation vor, bei der die beiden Prozesse über einen Puffer gekoppelt sind; die Operationen zum Ablegen und Holen von Information enthalten kritische Bereiche, da dieser Puffer zu beiden Prozessen global sein muss.

Beim hier behandelten Problem erfolgt die Verknüpfung zweier Prozesse nicht statisch, sondern dynamisch für die Länge der Uebertragung eines oder mehrerer Files; die Puffer können dabei den Ein- <u>oder</u> den Ausgabeprozessen fest zugeordnet werden. Wenn pro Ein- <u>oder</u> Ausgabeprozess ein Puffer existiert, muss zu Beginn der Uebertragung nur noch jeweils dem anderen Prozess der Puffer des Partner-Prozesses zugewiesen werden.

4.2.1. <u>Die Pufferbehandlung</u>

Bei der Strukturierung und Organisation der Puffer für die Uebermittlung eines Files von Eingabe- zu Ausgabeprozess sind verschiedene Punkte zu berücksichtigen, insbesondere wenn wenig Speicherplatz für Puffer zur Verfügung steht:

(1) Je grösser der Puffer für eine Verbindung, umso schwächer ist die Kopplung der beteiligten Prozesse.

(2) Der reservierte Platz für die Puffer nicht aktiver Verbindungen soll möglichst klein sein, damit aktiven Verbindungen möglichst viel Pufferplatz zur Verfügung steht.

(3) Je seltener ein Prozess in einem kritischen Bereich ist, umso seltener muss der andere Prozess warten, bevor er seinen kritischen Bereich durchlaufen kann.

(4) Der Verwaltungsaufwand pro Zeichen, bedingt durch Unterbrechen und Wiederstarten des schnelleren Prozesses bei leerem bzw. vollem Puffer, soll möglichst gering sein.

Bedingungen (1) und (2) lassen sich erfüllen, wenn man entweder

1. weniger Puffer zur Verfügung stellt als gleichzeitig Verbindungen möglich sind. Damit werden die Puffer grösser, müssen aber beiden Prozessen dynamisch zugeteilt werden und bei Unterbrechung einer Verbindung bleibt der Pufferplatz blockiert.

oder

2. die Grösse der Puffer nicht konstant hält und aktiven Verbindungen mehr Speicher für die Pufferung von Information zur Verfügung stellt.

Bedingungen (3) und (4) legen nahe, Zeichen nicht einzeln sondern in Blöcken zum und vom Puffer zu übertragen. Zusammen mit 2. empfiehlt sich folgende Struktur für die Pufferbehandlung:

(a) Information wird in Blöcken von höchstens N Zeichen übertragen. Kleinere Blöcke können am Ende eines Files oder in Ausnahmesituationen auftreten.

(b) Die Speicherplätze für Blöcke von N Zeichen (<u>Blockpuffer</u>) werden <u>dynamisch</u> verwaltet, d.h. sie werden den aktiven Verbindungen nach Bedarf zur Verfügung gestellt.

(c) Der einer Verbindung fest zugeordnete Puffer hat eine konstante Länge und enthält nur Zeiger zu Blockpuffern. Nicht benötigte Blockpuffer werden zentral verwaltet (buffer pool).

Durch die Einführung der Blockpuffer wird nur ein kleiner Teil des für Puffer zur Verfügung stehenden Platzes fest für Verbindungspuffer verwendet, der grössere Teil steht in Form von Blockpuffern für die aktiven Verbindungen zur Verfügung (2), die somit so stark entkoppelt (1) werden, wie es die Gesamtbelastung des Systems zulässt. Bei Unterbrechung einer Verbindung stehen freiwerdende Blockpuffer den restlichen aktiven Verbindungen wieder zur Verfügung. Die kritischen Bereiche in den Operationen für das Ablegen bzw. Abholen von Information (3) werden nur je einmal pro N Zeichen durchlaufen und auch das Unterbrechen bzw. Wiederstarten des schnelleren Prozesses (4) erfolgt höchstens einmal pro Block.

Bei diesem Puffermechanismus ergibt sich folgender Fluss von Blockpuffern:

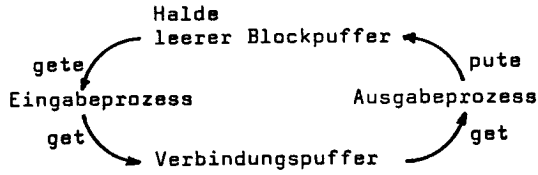


Abb. 4.10: Fluss von Blockpuffern

Damit z.B. beim Blockieren eines Ausgabeprozesses nicht alle Blockpuffer gefüllt in den zugehörigen Verbindungspuffer abgelegt werden und somit der Informationsfluss des ganzen Systems blockiert wird, ist eine Möglichkeit zur Beschränkung der Grösse der Verbindungspuffer vorzusehen; die Wahl dieser Grösse hängt von der Struktur des Systems und den dort vorgesehenen Möglichkeiten zum Abbrechen einer Verbindung ab und soll daher hier noch nicht behandelt werden.

Folgender Programmauszug enthält die bisher entwickelten Strukturen und garantiert durch die Art der Vereinbarungen, dass kein Prozess eine lokale Kopie eines Zeigers zu einem Blockpuffer behält, wenn dieser einem anderen Prozess weitergegeben wird:

```
type bit8 = 0..377B;
     block = record cnt: 0..N; next: ↑block;
                info: array [1..N] of bit8
             end;
```

```
...

module bufferhandling;
  defines type *pblock, buffer;
          procedure initbuffer, get, put, gete, pute, swap;

  const M = ...  {= max. length of buffer > 2};
  type pblock= ↑block;
       buffer= record inx, outx: 0..M;
                 blgth: 2..M: wq: queue;
                 b: array [0..M-1] of pblock
               end;

  protected module emptyblocks;
    defines procedure gete,pute;

    var emptyb, pb: pblock; web: queue;

    procedure gete(var b: pblock);
    begin
      if emptyb= nil then enterq(web);
      "assign to b block from emptyb"
    end {gete};

    procedure pute(var b: pblock);
    begin b↑.next:= emptyb; emptyb:= b; b:= nil;
      startq(web)
    end {pute};

  begin emptyb:= nil;
    while "space" do
    begin new(pb); pb↑.next:= emptyb; emptyb:= pb
    end
  endmodule {emptyblocks};

  protected module getput;
    defines procedure get, put;

    procedure get(var bf: buffer; var pb: pblock);
    begin with bf do
```

```
      begin if inx= outx then {buffer empty} enterq(wq);
        pb:= b[outx]; outx:= (outx+1) mod blgth;
        startq(wq) {producer might be waiting}
      end
    end {get};

    procedure put(var bf: buffer; var pb: pblock);
    begin with bf do
      begin if (inx+1) mod blgth = outx
              then {buffer full} enterq(wq);
        b[inx]:= pb; pb:= nil; inx:= (inx+1) mod blgth;
        startq(wq) {consumer might be waiting}
      end
    end {put};

  endmodule {getput};

  procedure initbuffer (var bf: buffer; lgth: integer);
  begin with bf do
    begin inx,outx:= 0; blgth:= "max"(2, "min"(M, lgth))
    end
  end {initbuffer};

endmodule {bufferhandling}
```

Bei dieser Darstellung der Routinen wird nicht verhindert, dass eventuell (leere) Pufferblöcke verlorengehen, z.B. durch aufeinanderfolgende Aufrufe von gets. Die Modulprozeduren garantieren aber, dass

1. keine Verbindung aus modulinternen Gründen blockiert wird; jeder Prozess, der auf einen Block oder Platz für einen Block wartet, wird wieder gestartet, sobald der entsprechende Zustand eintritt.

2. keine Information verloren geht. Die Modulroutinen greifen zur Komponente info des Typs block nicht zu, und die Behandlung der Indizes inx und outx garantiert, dass im Modul keine Informationsblöcke verlorengehen. Für den Zustand des Puffers

gilt

inx = outx <=> Puffer ist leer,

d.h. dieser Zustand darf beim Einfüllen (put) nicht erreicht werden, da sonst die ganze Information verloren ginge. Ausserdem garantiert die Initialisierung, dass ein Puffer mindestens für zwei Blöcke Platz bietet und damit nicht Verbraucher- und Erzeugerprozess in der Queue wq warten.

3. durch die rein sequentielle Struktur des zirkulären Puffers die Reihenfolge der Information erhalten bleibt.

Bemerkung: Bei diesem Algorithmus zur Behandlung des zirkulären Puffers kann beim Modul bufferhandling bei beliebigem Algorithmus der Prozessorzuteilung und bei nur zwei Prozessen, die zu einem Puffer zugreifen, auf den Schutz des Moduls getput verzichtet werden, wenn die Queue wie ein binäres Semaphore implementiert ist (vgl. 2.5), da put nur die Variablen inx und b[inx], get nur outx verändert, und outx von put, inx und b[outx] von get nur gelesen werden.

4.2.2. Steuerung und Kontrolle der Verbindungen

Zwei Ströme bzw. die ihnen zugeordneten Prozesse E und A heissen verknüpft, wenn ein File des Eingabestromes von E nach A zum entsprechenden Ausgabestrom übermittelt wird oder übermittelt werden kann. Eine Verknüpfung ist aktiv, wenn die Prozesse E und A durch einen Puffer verbunden sind.

Verknüpfungen von Strömen bzw. Prozessen werden durch Befehle aus dem Strom B hergestellt bzw. getrennt, während die Aktivierung einer Verbindung von den beteiligten Prozessen E und A kontrolliert wird. Ein Eingabestrom kann höchstens mit einem Ausgabestrom verknüpft sein, da ein nächstes File nur einen Zielstrom haben kann, während ein Ausgabestrom gleichzeitig mit mehreren Eingabeströmen verknüpft sein kann, wenn Files von verschiedenen Strömen auf einem peripheren Gerät auszugeben sind; für einen Ausgabestrom kann aber zu jedem Zeitpunkt höchstens eine Verbindung

aktiv sein. Damit gibt es für Ströme folgende Zustände (und Uebergänge):

```
        >-(connect)    ->            >-(openconnection) ->
free                        connected                          active
        <-(disconnect)-<             <-(closeconnection)-<
```

Abb. 4.11: Mögliche Zustände der Ströme

Dabei gilt:

1. connect: Ein- und Ausgabestrom (bzw. Prozesse) werden verknüpft. Für Eingabeprozesse, die an einer aktiven Verbindung beteiligt sind, wird dieser Befehl nicht zugelassen, da sonst die zeitabhängige Zuordnung von Befehlen zu Files, die mit Ausnahme des Kartenlesers über zwei unabhängige Ströme ins System gelangen, nicht eindeutig möglich ist.

   Eine bestehende, nichtaktive Verbindung wird implizit getrennt.

2. disconnect: Eine nichtaktive Verbindung wird sofort getrennt, eine aktive erst nach Abschluss der Uebermittlung.

3. openconnection: Ein Eingabeprozess verlangt die Aktivierung einer Verbindung: bei aktivem Ausgabeprozess muss er (in einer Queue von Eingabeprozessen) warten.

4. closeconnection: Ein Ausgabeprozess schliesst eine aktive Verbindung ab und ist zur Aufnahme einer nächsten bereit.

Die Trennung von Herstellen und Aktivieren einer Verbindung erlaubt die wiederholte Aktivierung einer einmal vom Benützer hergestellten Verbindung ohne zusätzliche Eingriffe. Somit ist es z.B. möglich eine einmal hergestellte Verbindung eines Linieneingabestromes mit

dem Drucker für jedes übermittelte File zu aktivieren.

Um in das Schema dieser Befehle die Zuordnung eines Puffers einfach einbauen zu können, wird jedem Ausgabestrom ein Puffer fest zugeordnet, und dem Eingabestrom bzw. -prozess wird durch die Prozedur openconnection der Name des Ausgabeprozesses geliefert, der gleichzeitig Index zum Puffer des entsprechenden Ausgabestromes ist.

Dieser Algorithmus zur Kontrolle und Steuerung der Prozesse ist im Modul connectcontrol mit folgender Struktur implementiert:

```
...
type procsname= (idummy,cardi,ppti,lineNi, ..., line1i,
                 line1o, ..., lineMo,
                 cardo, print, ppto, odummy);
     iprocsname= idummy..line1i;
     oprocsname= line1o..odummy;

...
protected module connectcontrol;
  defines procedure connect, disconnect,
                    openconnection, closeconnection,
                    getconnection, getspecification;

  type procstyp= (iproc, oproc);
       procsdescriptor=
            packed record active, todisconnect: Boolean;
              conn: procsname;
              pspec: integer; {specification for
                               information processing}
              wprcs: queue;
              case procstyp of
                iproc: (ospec: integer); {spec for
                                          outputprocess}
                oproc: ( )
              end
            end;
  var pcstable: array [procsname] of procsdescriptor;
```

```
procedure connect(ip: iprocsname; op: oprocsname;
                  ispec, ospec: integer; var done: Boolean);
begin if "input process not active" then
      begin "mark connection and insert specifications
             in descriptor of input process";
        done:= true
      end else done:= false
end {connect};

procedure disconnect(ip: iprocsname);
begin with pcstable[ip] do
  if active then todisconnect:= true
            else conn:= idummy
end {disconnect};

procedure openconnection(ip: iprocsname; var op: oprocsname;
                                          var spec: integer);
  {spec= specification for information processing}
begin
  if "ip not connected or last connection still active"
      then enterq(wprcs);
  "mark inputprocess active, assign outputprocess and spec";
  {start connection:}
  if "connected outputstream active" then
    enterq(pcstable[pcstable[ip].conn].wprcs);
  "mark outputprocess active and copy specification
   in output process descriptor"
end {openconnection};

procedure closeconnection(op: oprocsname);
  var ip: iprocsname;
begin "deactivate input and output process";
  ip:= pcstable[op].conn;
  if "input process to-disconnect"
    then pcstable[ip].conn:= idummy
    else startq(pcstable[ip].wprcs);
  startq("queue of processes waiting on output process")
end {closeconnection};
```

```
    procedure getconnection (p1: procsname; var p2: procsname);
    begin p2:= "process connected to p1"
    end {getconnection};

    procedure getspecification (p: procsname; var spec: integer);
    begin spec:= "specification for process p"
    end {getspecification};

  begin for p:= idummy to odummy do "initialize processdescriptor"
  endmodule {connectctrl}
```

Für Eingabe- und Ausgabe-Prozesse ergibt sich damit folgende Struktur:

```
   ...
  var carea: array [line1o..odummy] of buffer;
                    {communication area}
   ...

  process input;
    ...
  begin
    repeat "read first block of information";
      openconnection("ipname", op, spec);
      repeat put(carea[op], "information block");
      until "end of file"
      do "read block of information" end
    end {of cycle}
  end {input};

  process output;
    ...
  begin
    repeat
      repeat get(carea["opname"], "information block");
        "write block of information"
      until "end of file";
      closeconnection("opname")
    end {of cycle}
  end {output}
```

Der Aufbau der Modulstrukturen zusammen mit dieser Struktur der Prozesse garantiert, dass jeweils höchstens ein Eingabeprozess Information zu einem Ausgabeprozess übermittelt; syntaktisch nicht erfasst wird ein Eingabeprozess, der nicht nach obigem Schema abläuft, sondern z.B. direkt Information in den Puffer eines Ausgabestromes übermittelt.

4.2.3. Steuerung von Ein- und Ausgabeprozessen

Ein- und Ausgabeprozesse erhalten bzw. liefern ihre Daten auf periphere Geräte, die durch Fehler ausfallen können oder die für weitere Arbeit neu zu initialisieren sind (z.B. Papier nachfüllen). Auch kann ein File falsch zusammengestellt sein oder durch einen Bedienungsfehler einem falschen Ausgabegerät zugewiesen werden, und eine Fortsetzung der Uebermittlung sinnlos werden lassen.

Für diese Fälle ist ein Eingriff in den Ablauf einer Verbindung über die Konsole und den Befehlsstrom B vorgesehen (siehe Anhang B); zur Ausführung dieser Befehle werden intern folgende Operationen zur Verfügung gestellt:

1. stopeoj(p): Der Prozess p ist nach Abschluss der gerade aktiven Verbindung anzuhalten.

2. stop(p): Der Prozess p ist "sofort" zu unterbrechen.

3. go(p): Die durch stopeoj und stop gesetzten Haltebedingungen werden aufgehoben, der unterbrochene Prozess wird wieder gestartet.

4. terminate(p): Eine gerade aktive Verbindung, an der der Prozess p beteiligt ist, wird abgebrochen, der Rest des Files vom Eingabestrom wird ignoriert.

Da der Ablauf eines Prozesses nach der Definition in 2.2 von einem anderen Prozess nicht direkt beeinflusst werden kann, muss für einen Befehl eine Anzeige gesetzt werden, die jeder Prozess regelmässig abzufragen hat und entsprechend darauf reagieren muss. Dazu sind folgende Funktionen und Prozeduren vorgesehen:

1. condset(p,c): Zum Abfragen einer bestimmten Anzeige c; diese Möglichkeit erlaubt einem Prozess, noch lokale Aktionen vorzunehmen oder sogar die Anzeige dem verknüpften Prozess weiterzugeben, bevor er z.B. auf ein "go" wartet.

2. wait(p,c): Zum Warten, wenn die Anzeige c für den Prozess p gesetzt ist.

3. terminate(p,q): Quittiert den Abbruch der Verbindung und löscht die Anzeige.

Diese Routinen zusammen mit den Variablen für die Anzeigen sind im folgenden Modul runcontrol zusammengefasst:

```
module runcontrol;
  defines type *rcond;
          procedure wait, stop, stopeoj, terminate, go;
          function condset;

  type rcond= (none, stp, seoj, term);
       wcond= none..seoj;
       rstatedescr= record s: set of rcond;
                           wgo: queue
                    end;
  var rs: array [procsname] of rstatedescr;

  function condset(p: procsname; r: rcond): Boolean;
  begin condset:= r in rs[p].s end {condset};

  protected module changerunstate;
    uses rs;
    defines procedure wait, stop, stopeoj, terminate, go;

    procedure wait(p: procsname; c: wcond);
    begin if c in rs[p].s then enterq(rs[p].wgo)
    end {wait};

    ...
```

```
    procedure terminate(p:procsname; done: Boolean);
    begin if done then rs[p]:= rs[p].s - [term]
                  else rs[p]:= rs[p].s + [term]
    end {terminate};

    procedure go(p: procsname);
    begin rs[p].s:= rs[p].s - [stp,seoj];
      startq(rs[p].wgo)
    end {go};

  endmodule {changerunstate};

begin "initialize rs"
endmodule {runcontrol}
```

Bemerkung 1: Dieser Modul runcontrol entspricht nicht der im zweiten Kapitel gegebenen Sprachdefinition, da durch die Funktion condset mehrere Prozesse zu rs (zum Lesen) zugreifen können; dabei werden aber nur Werte gelesen, unter Umständen zwar veraltete, wenn "gleichzeitig" eine Prozedur des lokalen Moduls changerunstate ausgeführt wird, aber das führt höchstens zu einer verzögerten Reaktion auf eine Anzeige.

Es ist an Hand weiterer Anwendungen zu ermitteln, ob ungeschütztes Lesen von globalen Variablen (bei eventuell gleichzeitigem Aendern durch andere Prozesse) syntaktisch erlaubt werden soll.

Bemerkung 2: Für diese vielen kleinen Prozeduren und Funktionen, die bedingt sind durch das syntaktische Zusammenfassen von Typen bzw. Variablen und Operationen, sollte die Sprache erweitert werden, damit ein Compiler speziell gekennzeichnete Prozeduren als offene Subroutinen (Macros) übersetzt.

4.2.4. Die allgemeine Struktur der Ein- und Ausgabeprozesse

Mit den drei Modulen bufferhandling, connectcontrol und runcontrol stehen nun alle Typen, Variablen und Operationen zur Verfügung, um den Austausch von Information zwischen Datenströmen entsprechend

Abb. 4.2 zu kontrollieren. Die in 4.2.2. skizzierte Struktur der Ein- und Ausgabeprozesse soll nun unter Berücksichtigung der Ablaufkontrolle verfeinert werden: für die Ein- und Ausgabe von Blöcken von Information werden hier Prozeduren

```
readperipheral(block, length, ..)
```

und

```
writeperipheral(block, count, ..)
```

vorausgesetzt als Verbindung zu den Programmteilen, die die Steuerung der peripheren Geräte übernehmen. Da diese Prozeduren vom Typ der Geräte abhängen und nicht HEXAPUS- spezifisch sind, sollen sie im fünften Kapitel getrennt behandelt werden.

```
process input;
  uses carea, readperipheral,
       bufferhandling, connectcontrol, runcontrol;

  var ...; b: pbuffer; {vgl. 4.2.1., bufferhandling}

begin "initialize local variables";
  repeat gete(b); readperipheral(b↑.info, b↑.cnt, ...);
    openconnection(ip,op,spec);
    repeat eof:= b↑.info[b↑.cnt]= EOI;
      wait(ip,stp); {wait if stop condition set}
      if condset(term) then {terminate connection}
      begin b↑.info[1]:= EOI; b↑.cnt:= 1;
        put(carea[op],8);
        if not eof then {skip rest of information}
        begin gete(b);
          repeat readperipheral(b↑.info, b↑.cnt, ...);
          until "end of file";
          pute(b)
        end; eof:= true; terminate(ip,true)
      end else
      begin "transform information corresponding spec";
        put(carea[op],b)
      end;
    until eof do gete(b) end;
    wait(ip,seoj); {wait if stop-end-of-job condition set}
```

```
    end {of cycle}
  end {input};


  process output;
    uses carea, writeperipheral,
         bufferhandling, connectcontrol, runcontrol;

    var ...; b: pbuffer;

  begin "initialize local variables";
    repeat get(carea[op],b); getspecification(op,spec);
      repeat wait(op); eof:= b↑.info[b↑.cnt]= EOI;
        "transform information corresponding spec";
        if condset(term) then
        begin "ignore information until end of file";
          eof:= true
        end else writeperipheral(b↑.info, b↑.cnt);
        pute(b)
      until eof do get(carea[op],b);
      closeconnection(op); wait(op,seoj)
    end {of cycle}
  end {output}
```

Die Abbruchbedingung kann bei einem Ausgabeprozess ignoriert werden, wenn die Eingabe des Abbruchbefehls bewirkt, dass bei beiden beteiligten Prozessen die entsprechende Anzeige gesetzt wird, und wenn jeder Eingabeprozess die Uebermittlung von Information durch einen Block mit EOI so schnell wie möglich abschliesst. Wenn die Verarbeitung der restlichen bereits im Puffer gespeicherten Information besondere Nebenwirkungen hat, wie z.B. dauerndes Springen auf eine neue Seite bei der Ausgabe auf einem Drucker, wird jedoch auch der Ausgabeprozess die Abbruchanzeige berücksichtigen.

Bei diesen allgemeinen Strukturen scheint es überflüssig, die Ablaufkontrolle separat zu behandeln, und sie nicht mit den Pufferoperationen put und get zu verknüpfen. Spezielle Prozesse benötigen aber die getrennte Kontrolle, etwa um auf Unterbruch-

bedingungen speziell reagieren zu können. Zum Beispiel soll beim Abbruch einer Verbindung (Zentralrechner-) Linie-Drucker die Ausgabe sofort unterbrochen werden und die Information des Files bis zum letzten Segment übersprungen werden, während das letzte Segment, das Protokoll eines Jobs, wieder gedruckt werden soll.

Die Umwandlung der Information kann zu einer Ausdehnung bzw. Kompression der Daten führen und somit das Umkopieren in andere Puffer erfordern; dies kann im Ein- oder Ausgabeprozess direkt als Teil der zu verfeinernden Anweisung "transform information corresponding spec" erfolgen. Eine andere Möglichkeit wäre die Aufteilung des Prozesses in einen Ein- oder Ausgabeprozess und einen Umwandlungsprozess:

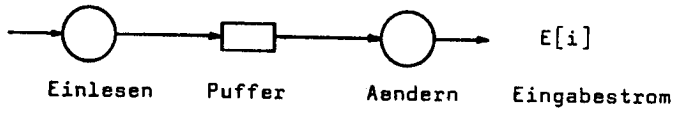


Abb. 4.12: Aufspaltung des Eingabeprozesses

Diese zweite Möglichkeit wurde bei der vorliegenden Implementation nicht gewählt, da unter den in 1.3 gestellten Bedingungen der Platz und nicht die Zeit der kritische Faktor ist.

4.3. Behandlung der Linienströme

In Abschnitt 4.1 wurde bereits skizziert, wie Files für verschiedene Linienströme verzahnt zu übermitteln sind, um gleichzeitig mehrere Datenströme über eine Linie führen zu können (vgl. Abb. 4.5 und 4.8). Die effektive Behandlung der Leitung als peripheres Gerät geschieht in einem Driver-Modul, der Prozeduren readline und writeline zur Ein- und Ausgabe von Datenblöcken zur Verfügung stellt (s. 5.3). Hier soll die Verbindung der lokalen Datenströme des Systems HEXAPUS mit den Ein- und Ausgabefiles der Zentralanlage beschrieben werden.

Um eine einfache Abbildung mehrerer Datenströme auf einen Strom durch den Sender und eine einfache Aufteilung des einen Stromes auf die ursprünglichen Ströme durch die Empfängerstation zu ermöglichen, werden in einem Block nur Daten für einen Strom übermittelt, zusammen mit Information, die angibt, zu welchem Datenstrom dieser Block gehört.

Um die beiden Anlagen bzw. die beiden an der Uebertragung der Information beteiligten Prozesse nicht zu stark zu koppeln und den späteren Uebergang zu höheren Uebertragungsraten und parallele Uebertragung zu erleichtern, ist eine gewisse Asynchronität zwischen diesen Sender- und Empfängerprozessen erforderlich, d.h. eine Station soll mehrere Blöcke absenden können, bevor sie auf eine Quittung der Gegenseite warten muss. Dazu werden die Informationsblöcke pro Station und Richtung laufend durchnumeriert (modulo 16), und (positive oder negative) Quittungen beziehen sich auf diese Nummern.

Negative Quittungen erlauben der Datenstation bereits empfangene Information abzulehnen, wenn sonst z.B. die Linie für andere Ströme blockiert würde. Bei einer negativen Quittung werden alle Blöcke mit späteren Nummern als der zurückgewiesenen ignoriert und sind von der Zentralstation nochmals zu übermitteln; damit wird verhindert, dass in der Datenstation Speicher blockiert wird, um die Informationsblöcke wieder in die richtige Reihenfolge zu bringen. Diese Möglichkeit der negativen Quittung bedingt die Speicherung der Information in der Sendestation bis zum Empfang einer positiven Quittung vom Empfänger. Da die Datenstation mit kleinem Speicher und ohne Hintergrundspeicher auskommen soll, die Zentralanlage aber über genügend (Hintergrund-) Speicher für die eintreffende Information verfügt, ist die Möglichkeit der negativen Quittung auf die Datenstation beschränkt.

Beiden Empfängern muss es dagegen möglich sein, die Uebertragung für einen bestimmten Strom zu unterbrechen, wenn stationsinterne Gründe dies erfordern.

Da der Informationsaustausch des Zentralrechners mit der Datenstation nach der Problemdefinition nur über die Jobeingabeschlange und Jobausgabeschlange möglich ist und da im Zentralrechner für die

Eingabe und für jede Art von Ausgabegerät eine eigene Ausgabeschlange unterhalten wird, beziehen sich die Stromdefinitionen zwischen Datenstation und Zentralrechner auf diese verschiedenen Sequenzen von Files.

Der mit jedem Informationsblock übermittelte Vorspann hat folgende Struktur:

1. Nummer des Stromes, zu dem die Information gehört (Ø-7).

2. Sequenznummer des abgehenden Informationsblocks (Ø-17B).

3. Sequenznummer des zuletzt zu quittierenden Blocks, eventuell mit negativer Quittung von der Datenstation.

4. Ein Zustands-Bit für jeden der acht möglichen Linienströme:
   jede Station kann damit für ihre Eingabeströme angeben, ob Information übermittelt werden kann, für ihre Ausgabeströme, ob Information vorhanden ist.

(Die 'binäre' Information des Vorspanns wird durch Addition von 4ØB in den Bereich der druckbaren ASCII-Zeichen abgebildet.)

Auf jedem zwischen Zentralrechner und Datenstation definierten Strom kann nur Information in einer Richtung übertragen werden; ausgenommen ist der sogenannte <u>Nullstrom</u>, der immer definiert ist und über den Befehle (interner Strom LOØ) und Meldungen (LIØ) zwischen den Stationen ausgetauscht werden.

Wird jeder Datenstrom durch einen Prozess dargestellt und wird für die zentral zu verwaltende Information ein geschützter Modul eingeführt, so ergibt sich für die Behandlung der Linie folgende Struktur:

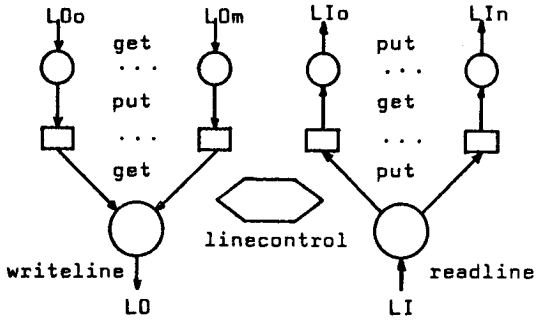


Abb. 4.13: Behandlung der Linienströme

Von dieser Struktur sind folgende Bedingungen zu erfüllen:

1. LI und LO, die über einen Modul linectrl gekoppelt sind, dürfen nie in einen Zustand gelangen, in dem sie den anderen Prozess und damit die Linie blockieren.

2. Diese ganze Struktur muss sich aus <u>jedem</u> Zustand selbst wieder initialisieren können, wenn vom Zentralrechner (bei Deadstart oder bei gewissen Uebermittlungsfehlern (siehe 5.3.2)) eine neue Verbindung gestartet wird.

3. Diese Einheit muss dafür sorgen, dass beim Starten der Datenstation auch das Zentralsystem richtig initialisiert wird.

4. Bei Linienunterbruch oder Ueberbelastung der Datenstation muss es möglich sein, die Verbindung zwischen LI und irgendeinem LIj abzubrechen und dadurch auch den mit LIj verbundenen lokalen Ausgabeprozess freizugeben.

Die in Abb. 4.13 gezeigten Puffer, Prozesse und der Modul linecontrol sind in einem Modul line zu einer syntaktischen Einheit zusammengefasst:

```
modul line;
  uses bufferhandling, connectcontrol, runcontrol, ...;
  defines procedure cmdlno0;
```

```
type lxrange = (LInx, ..., LI0x, LO0x, ..., LOmx);
     lixrange = LInx .. LI0x;
     loxrange = LO0x .. LOmx;
var ...
  lcarea: array [lxrange] of buffer;

procedure cmdlno0( ... );
begin : "set up line commands from parameters: LO0";
  put(lcarea[LO0x], "command block")
end {cmdlno0};

protected module linecontrol;
  "control cooperation of line processes (4.3.1)"
endmodule {lincontrol};

process receive;
  uses lcarea, readline, bufferhandling, linecontrol;
begin "process data stream LI (4.3.2)"
end {receive};

process transmit;
  uses lcarea, writeline, bufferhandling, linecontrol;
begin "process data stream LO (4.3.2)"
end {transmit};

process lineo(lonx: loxrange; op: oprocsname);
  uses lcarea, carea,
       bufferhandling, connectcontrol, runcontrol;
begin "process data stream LO[lonx] (lonx <> LO0x) (4.3.2)"
end {lineo};

process linei(linx: lixrange; ip: iprocsname);
  uses lcarea, carea, linecontrol,
       bufferhandling, connectcontrol, runcontrol;
begin "process data stream LI[linx] (linx <> LI0x) (4.3.2)"
end {linei};
```

```
    process linei0;
      uses lcarea, bufferhandling, writetty;
    begin "process messages (LI0)"
    end {linei0};

  begin "initialize module variables";
    start transmit, receive,
          lineo(LO1x,line1o), ..., lineo(LOmx,linemo),
          linei(LI0x,line1i), ..., linei(LInx,lineni),
          linei0;
  endmodule {line}
```

4.3.1. Linienkontrolle

Beim Informationsaustausch zwischen Datenstation und Zentralanlage müssen LI und LO Information über den Zustand der Verbindung austauschen:

1. LO darf nur DIFF (= 2) Informationsblöcke absenden, ohne auf eine Quittung von der Zentralanlage (über LI) zu warten.

2. LO muss der Zentralanlage Information über den Zustand der Ströme LI1, ..., LIn übermitteln, d.h. über deren Bereitschaft zur Entgegennahme von Information.

3. Nach Vereinbarung eines Linienausgabestromes muss LO die Quittung der Zentralanlage abwarten, bevor Information für diesen Strom übermittelt werden darf.

Diese Information, die jeweils im Vorspann übermittelt wird, soll zusammen mit der Zuordnung der mit der Zentralanlage vereinbarten Datenströme zu den internen Strömen LI0 ... LIn und LO0 ... LOm bzw. den entsprechenden Prozessen linei(...) und lineo(...) in dem geschützten Modul linecontrol verwaltet werden. In diesem Modul definierte Prozeduren sollen daher den von receive empfangenen Vorspann auswerten und prüfen sowie den von transmit zu sendenden Vorspann aufsetzen. Weitere Prozeduren erlauben das Unterbrechen der Uebermittlung von Daten über einen externen Strom und geben Auskunft über den Zustand einzelner Ströme und Verbindungen:

```
protected module linecontrol;
  defines type *transmstate, *loxset;
          procedure startconnection,getstate,transmpossible,
                    send,recvd,repeattransm,
                    canceledstream,stopstream,gostream;

  const DIFF = 2;
  type loxset = set of loxrange;
       transmstate = (none, idle, restart);
  var ...

  procedure startconnection;
  begin "initialize all modul variables
         for a new connection"
  end {startconnection};

  procedure getstate(var cond: transmstate;
                     var activeostreams: loxset);
  begin
    if newconnection then
    begin newconnection:= false; cond:= restart
    end else
      if "not all received blocks acknowledged"
        then cond:= idle else cond:= none;
    activeostreams:= "set of all active outputstreams"
  end {getstate};

  function transmpossible: integer;
  begin transmpossible:= "number of blocks which may be sent
                          without waiting for a quit;
                          < 0 iff transmission is blocked"
  end {transmpossible};

  procedure send(var bf: linebuf; lx: loxrange);
    {setup protocol and wait for appropriate
     quit if necessary}
  begin "increment send sequence number (mod 16)";
    if "DIFF quits outstanding" then enterq(waitforquit);
    "set up protocol in bf[1] - bf[5];
     add 40B to the 'binary' representation of
```

```
   protocol information.
    1. Insert stream[lx].
    2. Send sequence number.
    3. Quit last block received (and accepted);
       if block is to be rejected, set reject bit.
    4. Insert stream status."
  if "buffer contains command for commandstream" then
    if "command= define stream"
      then "mark stream defined"
      else if "command= cancel stream"
             then "mark stream to be canceled"
end {send};

procedure accept(ok: Boolean);
begin if ok then "increment sequence no. of next
                  block to be received";
  if "quit for a yet unquit block has been received" then
    startq(waitforquit)
end {accept};

procedure recvd(bf: linebuf; var lx: lixrange;
                var info: Boolean);
begin lx:= internalstream[linebuf[1]-40B];
  if "block no. <> expected no." then
    if "(block no.- expected no.)" mod 16 = 1 then
        "ignore block"
            {block may be sent after a rejected block}
      else
      begin message("out of sequence");
        "reset expected sequence no."
      end;
  "store quit no. and status of linestreams";
  if "streams which are to be canceled not active"
    then "clear definition";
  if "destination stream not active" then "ignore block";
  if "block to be ignored" then
  begin info:= false; accept (false)
  end else info:= true
end {recvd};
```

```
    procedure repeattransm(rep: Boolean);
    begin if rep then "block is to be rejected";
      accept(not rep)
    end {repeattransm};

    function canceledstream(lx: lixrange): Boolean;
    begin canceledstream:= "stream not defined"
    end {canceledstream};

    procedure stopstream(lx: lixrange);
    begin "mark stream inactive" end {stopstream};

    procedure gostream(lx: lixrange);
    begin "mark stream active" end {gostream};

  begin "initialize tables and variables used
         before first call of startconnection"
  endmodule {linecontrol}
```

4.3.2. Struktur der externen und internen Linienprozesse

Durch einen speziellen Informationsblock wird über den Strom LI die Verbindung von der Zentralanlage her aufgenommen bzw. nach einem Zusammenbruch der Linie oder bei Initialisierung des Systems der Zentralanlage wieder gestartet. Der den Strom LI behandelnde Prozess "receive" veranlasst beim Empfang dieses Blocks die Initialisierung des Moduls linecontrol, nachdem zuvor eventuell noch nicht abgeschlossene Files der Ströme LI1, ..., LIn durch einen Block mit EOI abgeschlossen wurden.

Der Prozess receive überträgt auf dem Strom LI empfangene Information auf den im Vorspann spezifizierten internen Strom LIj. Dabei wird die Information aus einem Linienblockpuffer umgepackt in die (kleineren) Blockpuffer vom Typ block, wobei gleichzeitig die für die Uebertragung erfolgte Komprimierung von Folgen gleicher Zeichen wieder aufgelöst wird. Bei dieser Uebertragung kann receive in zwei Fällen blockiert werden:

1. Es stehen keine leeren Puffer zur Verfügung, receive wartet

durch Ausführung von gete in der Queue web auf einen leeren Blockpuffer.

2. Ein gefüllter Blockpuffer hat in einem Verbindungspuffer keinen Platz, durch put gelangt receive in die Queue wq.

Beide Fälle können eintreten, wenn genügend viele Ausgabegeräte gestoppt werden, und führen in der Folge zu einem Blockieren des den Strom LO behandelnden Prozesses transmit, wenn dieser spätestens nach Uebermittlung von DIFF weiteren Informationsblöcken auf eine Quittung (über den Strom LI) warten muss; somit könnten durch Anhalten der Ausgabegeräte schliesslich die Eingabegeräte gestoppt werden!

Um diese Situation zu umgehen, sind die Prozeduren gete, put und aus analogen Gründen für das Verhalten des Prozesses transmit auch get zu erweitern. Der sie aufrufende Prozess soll angeben können, ob er bei Nichtvorhandensein von Blockpuffern oder Platz zum Speichern eines Blockes in der entsprechenden Queue warten will oder ob die Prozedur nicht ausgeführt werden soll, damit der Prozess andere Aktionen ausführen kann.

Der Prozess receive kann somit in den Fällen (1) und (2) speichern, wieviel Information des Linienblockpuffers bereits verarbeitet wurde, kann dann den ganzen empfangenen Block zurückweisen (repeattransm(true)) und die restliche Information bei erneutem Empfang des Blocks verarbeiten. Damit diese mehrmalige Uebermittlung nicht unnötig oft vorkommt, versucht receive aber solange einen leeren Puffer zu bekommen oder einen Block abzulegen, bis der Prozess transmit ohne Quittung nicht mehr weiterarbeiten kann; dieser Zustand wird durch die Funktion transmpossible im Modul linecontrol abgefragt.

Damit ergibt sich für receive folgende Struktur:

```
process receive;
  ...
begin "initialize local variables";
  repeat readline(bf,cnt) until "starting block";
  repeat "initialize state descriptions for streams LIj";
```

```
      startconnection; {initialize module linecontrol}
      "send a block with EOI to terminate files on
       all streams LIj";
      repeat readline(bf,cnt)
      until "starting block" or "end of connection"
      do recvd(bf,linx,info);
        if info then
        begin "load variables from state descr. of stream linx";
          if linx= LI0 then
            "test if idle block, message or end of connection";
          if info then
            "copy and expand information from
             bf into blockbuffers (putch)";
        500:"store state of copying in state description
             of stream linx"
        end
      end
    until "end of connection"
  end {receive}
```

Dabei wird die Information durch folgende lokale Prozedur in die Blockbuffer übertragen:

```
  procedure putch;
  begin
    if "block full" then
    begin repeat put(lcarea[linx],b,true) {conditional put}
          until (b=nil) or (transmpossible < 0);
          if b<>nil then goto 500;
          repeat gete(b,true)
          until (b<>nil) or (transmpossible < 0);
          if b=nil then goto 500;
          inx:= 0
    end;
    inx:= inx+1; b↑.info[inx]:= bf[out]
  end {putch}
```

Der Prozess receive sendet bei der Initialisierung einen Befehl zum Abbruch der Verbindung an die Zentralanlage, um die Initialisierung einer neuen Verbindung von der Zentralanlage aus auch dann zu erzwingen, wenn die Datenstation neu gestartet wird, für die Prozesse der Zentralanlage aber noch eine Verbindung besteht.

Wenn durch die Prozedur getstate angezeigt wird, dass von der Zentralanlage eine Verbindung neu oder erneut gestartet wurde, wird ein spezieller Block mit der Identifikation der Datenstation gesendet, die Teil des Namens jedes Files in der Zentralanlage ist, das von der Datenstation empfangen wurde und nach Verarbeitung wieder dorthin zurückzusenden ist. Diese Identifikation muss von der Datenstation gesendet werden, da bei Telefonanschlüssen die Station von der Zentralanlage nicht über den Dateneingang identifiziert werden kann.

Die in Abb. 4.2 eingezeichneten lokalen Verbindungen werden dann hergestellt und die entsprechenden Linienströme über den Linienbefehlstrom definiert. Anschliessend wird Information eines internen Linienstromes auf den externen Linienstrom übertragen und soweit möglich gepackt. Ein Linienblockpuffer wird übertragen, wenn er voll ist oder, bei langsam eintreffender Information, wenn die Funktion transmpossible anzeigt, dass eine Uebertragung möglich ist; dabei wird zum Holen von Informationsblöcken die bereits erwähnte Modifikation der Prozedur get im Modul bufferhandling verwendet.

```
process transmit;
  ...
begin "initialize local variables"; writeline(stopconn, ...);
  repeat
    repeat
      repeat getstate(cond,activeostreams)
      until activeostreams <> [ ]; {connection started}
      if cond= restart then
        "initialize default connections and
         define corresponding streams"
                        else
        "try to get (conditionally) information from
         one of the active output streams"
```

```
      until (b<>nil) or (cond=idle);
      if b<>nil then {information to send}
        "copy and comprime information into lineblockbuffer
         until buffer full or no more information available
         and transmission possible"
      else "send idle in order to quit received information";
      send(bf,lonx); writeline(bf,cnt)
    end
  end {transmit}
```

Die Struktur der die internen Linienströme behandelnden Prozesse entspricht der in Abschnitt 4.2.4 beschriebenen. Bei den internen Linienprozessen wird Information aber nicht direkt vom peripheren Gerät gelesen bzw. an periphere Geräte abgegeben, sondern sie wird über Verbindungspuffer mit den Prozessen receive bzw. transmit ausgetauscht. Wird ein Linieneingabestrom LIj unterbrochen, so hat der entsprechende Prozess diesen Unterbruch an den Modul linecontrol mit der Prozedur stopstream zu melden, damit bis zum Ende des Unterbruchs für diesen Strom keine Information mehr akzeptiert bzw. durch entsprechendes Setzen des Stromstatusbits keine Information von der Zentralanlage mehr gesendet wird.

4.4. Interpretation und Ausführung von Befehlen

Die zur Steuerung und Kontrolle des Systems vorgesehenen Befehle sind im Anhang B in einem Auszug aus der Bedienungsanleitung zusammengefasst. Befehle können vom Benutzer über Konsole oder Karten ans System gerichtet werden. Die entsprechenden Prozesse rufen eine Prozedur "interpret" im Modul cmdinterpreter, der alle Tabellen und Routinen enthält, die zur Entschlüsselung der eingegebenen Befehle notwendig sind; von diesen Roetinen werden entsprechend den eingegebenen Befehlen die von den anderen Modulen zur Ablaufkontrolle zur Verfügung gestellten Routinen aufgerufen.

Abb 4.14 zeigt die Behandlung dieser Steuerbefehle im Rahmen der bisher behandelten Struktur.

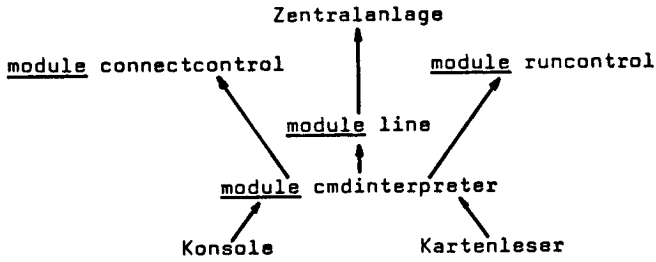


Abb. 4.14: Wirkung der Steuerbefehle

5. Steuerung peripherer Geräte

Während im Nucleus der Einsatz des Prozessors kontrolliert wird, werden die peripheren Geräte in den Driver-Routinen zur Ein- und Ausgabe von Zeichen kontrolliert. Zu einer möglichst guten (lokal optimierten) Auslastung dieser Geräte ist die Zeit zwischen Fertigmeldung des Gerätes und erneutem Starten klein zu halten, d.h. eingelesene Zeichen sollten 'sofort' verarbeitet werden, bzw. das als nächstes auszugebende Zeichen sollte schnell zugreifbar sein. Daher werden Routinen zur Verfügung gestellt, die Blöcke von N Zeichen einlesen bzw. ausgeben.

5.1. Allgemeine Struktur von Ein- und Ausgaberoutinen

Unter Verwendung der in 2.6 definierten Sprachkonstruktionen lässt sich die Struktur einer Eingaberoutine in einem Driver Modul folgendermassen darstellen:

```
driver module inputdevice;
  defines procedure input;

  type statustype = ...;
       infotype = ...;
  device dvc = PGi: packed record status: statustype;
                                   info: infotype
                    end;
   function f(info: infotype): bit8;
      "transform and/or check information"

  procedure input(var b: buffer; var n: length;
                  var error: Boolean);
    {upon entry: n= buffer length
     upon exit:  n= no. of characters read}
    var i: integer;

  begin while dvc.status <> "ready" do pause(M);
```

```
      i:= 0;
      under dvc control
        repeat start(dvc); i:= i+1; b[i]:= f(info)
        until (i=n) or (info = "termination value")
                    or "status indicates error";
        n:= i; error:= "error condition in status"
    end {input};

  endmodule {inputdevice}
```

Als Parameter sind ein Puffer und dessen Länge mitzugeben, Information, Umfang der Information und eventuelles Auftreten eines Fehlerzustandes werden durch die Routine geliefert.

Ausgaberoutinen lassen sich folgendermassen beschreiben:

```
  driver module outputdevice;
    defines procedure output;

    type statustype = ...;
         infotype = ...;
    device dvc = PGj: packed record status: statustype;
                                    info: infotype
                      end;

    function f(i: bit8): infotype;
      "transform and/or check information"

    procedure output(var b: buffer; n: length;
                     var error: Boolean);
    begin i:= 0;
          under dvc control
            while not "status indicates error" and (i<n) do
            begin i:= i+1; info:= f(b[i]); start(dvc)
            end;
          error:= "error condition in status"
    end {output};

  endmodule {outputdevice}
```

Je nach Art der Darstellung der Information auf einem Speichermedium können auch Parität oder Checksummen in diesen Routinen geprüft werden und z.B. bei Fehlern zu einem erneuten Einleseversuch führen.

Ein komplexeres Beispiel ist die Steuerung eines Druckers, dessen Kette bei längerer Nichtverwendung des Druckers unter Programmkontrolle auszuschalten ist:

```
module beltprinter;
  defines procedure writeprt;

  protected driver module prtctrl;
    defines procedure testpr, writeprt;

    const N = ...;
    device prt= PGi: packed record ...
                                      belton: Boolean;
                                      ch: bit8
                        end;
    var freecnt: 0..N; prtq: queue;

    procedure testpr;
    begin freecnt:= freecnt + 1;
      if freecnt= N then
      begin
        under prt control "stop beltmotor";
        enterq(prtq)
      end
    end {testpr};

    procedure writeprt(var b: buffer; cnt: length);
      ...
    begin
      under prt control
      begin if not belton then "start beltmotor";
        "print information"
      end;
      freecnt:= 0; startq(prtq)
    end {writeprt};
```

```
    begin freecnt:= N-1;
    endmodule {prtctrl};

    process timebelt;
      const C= ...;
    begin repeat testpr; pause(C) end
    end {timebelt};

  begin start timebelt
  endmodule {beltprinter}
```

Konsole und halb-duplex-Leitung sollen in den nächsten Abschnitten gesondert behandelt werden, da hier mehrere Prozesse beteiligt sind (Eingabe, Ausgabe und Operator bzw. Zentralrechner) und damit neue Gesichtspunkte berücksichtigt werden müssen.

5.2. Konsolendriver

Die Routinen zur Behandlung von Ein- und Ausgabe auf einer Konsole sollen hier den schwierigeren Fall der über eine halb-duplex-Leitung angeschlossenen Konsole behandeln; sie sollen so konzipiert sein, dass virtuell zwei Konsolen zur Verfügung stehen, eine zum Lesen und eine zum Schreiben.

Lesen von der Konsole soll entweder direkt möglich sein oder bedingt, d.h. das System wartet auf keinen Befehl, ist aber zur Entgegennahme eines Befehls bereit. Bedingtes Lesen kann für eine Ausgabe unterbrochen werden, während bei direktem Lesen die Konsole erst nach einer Eingabe wieder freigegeben wird. Da eine Eingabe durch einen Benutzer vom System nicht erzwungen werden kann, soll durch Ausgabe eines Doppelpunktes wenigstens angezeigt werden, dass eine Eingabe erwartet wird und die Konsole für die Ausgabe von Meldungen blockiert ist.

Bedingtes Lesen wird verwendet, wenn eine Eingabe zwar möglich ist, von einem Prozess aber nicht unbedingt benötigt wird. Dazu wird das

Lesen von der Konsole gestartet und entweder durch Eingabe eines Zeichens abgeschlossen oder durch Aufruf der Prozedur reset beendet, wenn ein anderer Prozess die Konsole zur Ausgabe einer Meldung benötigt.

Um nicht jedes versehentliche Eintippen als Anfang eines Befehles zu behandeln, soll erst ein bestimmtes Zeichen vom Zustand des bedingten Lesens in den des direkten Lesens führen, d.h. dieses Zeichen entspricht dem bei manchen Konsolen vorhandenen Knopf für einen "Konsolinterrupt".

Es gibt somit vier mögliche Zustände der Konsole und drei Operationen, für die die Konsole verlangt werden kann:

```
protected module ttyctrl;
  uses tty;
  defines type *ttyreq;
          procedure requestty,returntty;

  type ttystate= (free, condread, read, write);
       ttyreq  = condread..write;
  var wtty: queue; state: ttystate;

  procedure requestty(req: ttyreq);
  begin
    if state <> free then
      if state <> condread then enterq(wtty) else
        if req <> read then
        begin if req= write then reset(tty);
          enterq(wtty)
        end;
    state:= req
  end {requestty};

  procedure releasetty;
  begin state:= free; startq(wtty)
  end {releasetty};

begin initq(wtty); state:= free
endmodule {ttyctrl}
```

Durch requestty wird bei einem Schreibrequest ein eventuell bedingt lesender Prozess durch reset(tty) unterbrochen; dieser Prozess hat dann selbst die Konsole freizugeben und erneut auf ihre Verfügbarkeit zu warten. Im Zustand condread wird die Konsole sofort für das direkte Lesen freigegeben, da bei korrekter Verwendung der Konsolenoperationen readtty und writetty jeweils nur ein Prozess lesen darf.

```
driver module teletype;
  defines procedure readtty, writetty;

  const del= ...; lf= ...;
  device tty= PGi: packed record ...
                                  ch: bit8
                 end;

  protected module ttyctrl;
    ...
  endmodule  {ttyctrl};

  procedure readtty(var b: buffer; var cnt: length;
                    onreq: Boolean);
    var tyin: integer; req: Boolean;
  begin
    if onreq then
      repeat requestty(condread);
        under tty control
        begin "set tty to input mode";
          repeat start(tty)
          until (ch=del) or not completed(tty);
          req:= ch= del
        end
      until req do releasetty end;
    requestty(read);
    under tty control
    begin "set tty to output mode";
      ch:= ord(':'); start(tty); {input expected}
      "set tty to input mode"; tyin:= 0;
      repeat start(tty); tyin:= tyin+1; b[tyin]:= ch;
      until (tyin= cnt) or (ch= lf)
```

```
    end;
    releasetty; cnt:= tyin
  end {readtty};

  procedure writetty(b:buffer; cnt: integer);
    var tyout: integer;
  begin requestty(write);
    under tty control
    begin "set tty to output mode";
      for tyout:= 1 to cnt do
      begin ch:= b[tyout]; start(tty) end
    end;
    releasetty
  end {writetty}

endmodule {teletype}
```

Nicht berücksichtigt wird in dieser Darstellung das Löschen einer fehlerhaften Zeile und das Abbrechen der Ausgabe auf die Konsole (sofern die Technik der Konsole dies erlaubt).

Bemerkung: Beim Anschluss einer Konsole über eine duplex-Leitung stehen vom Rechner her gesehen zwei periphere Geräte zur Verfügung, die Tatstatur für die Eingabe und der Schreibkopf für die Ausgabe von einzelnen Zeichen. Damit entfällt die Notwendigkeit des bedingten Lesens, da Ein- und Ausgabe getrennt möglich sind.

Die Ausgabe eingetippter Zeichen ist aber mit der Ausgabe von Meldungen (im einfachsten Fall zeichenweise) zu synchronisieren. Dies geschieht entweder in der Konsole selbst, wenn eingetippte Zeichen im "home loop" ausgegeben werden, oder in der entsprechenden Ausgaberoutine.

Sollen Meldungen und Befehle auf dem Protokoll aber nicht beliebig verstückelt werden, muss die Konsole wie im behandelten Beispiel (ohne den Zustand condread) kontrolliert werden. Sollen ganze Meldungen eine Eingabe beliebig unterbrechen können, genügt die Synchronisation der Ausgabe von Meldungen mit der Ausgabe einzelner eingetippter Zeichen.

5.3. Liniendriver

Zur Uebermittlung bzw. zum Empfang von Information über die Linie werden zwei Routinen readline und writeline zur Verfügung gestellt; diese sollen so in eine Struktur eingebettet werden, dass ausserhalb dieser Struktur zumindest virtuell eine duplex-Leitung zur Verfügung steht, unabhängig von der technischen Realisierung.

Um bei einer halb-duplex-Leitung den beiden beteiligten Systemen die Möglichkeit zu geben, Information zu senden, wenn sie anfällt, muss die Verbindung periodisch der einen und anderen Seite zum Senden zur Verfügung gestellt werden. Jede Seite hat dann zu entscheiden, ob Information zu übermitteln ist, und kann die Information gegebenenfalls senden; diese Kontrolle der Verbindung muss unabhängig von Aufrufen der Routinen readline und writeline geschehen, d.h. der Liniendriver muss einen selbständigen Prozess enthalten, der diese Steuerungsfunktion übernimmt. Auf der Satellitenseite ist dies der Prozess LIP (Line Interface Process) und im Zentralrechner der Projess TP (Terminal Process).

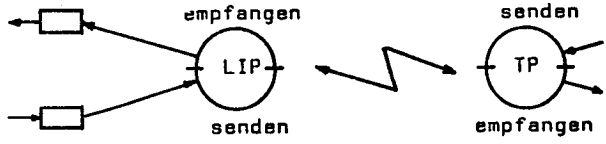


Abb. 5.1: Synchrone Verbindung zwischen den Linienprozessen in der Datenstation und im Zentralrechner

Diese Prozesse LIP und TP verwalten nun die Verbindung und werden durch den Austausch von Information über den gegenseitigen Zustand nach einem bestimmten Protokoll synchronisiert und sorgen für die korrekte Uebermittlung von eigentlicher Information.

In den nächsten Abschnitten wird dieses Protokoll und das Zusammenspiel der Prozesse LIP und TP beschrieben; im Anhang C ist das vollständige Programm des Moduls line aufgeführt, der die Routinen readline, writeline und den Prozess LIP enthält.

5.3.1. Das Linienprotokoll

Alle Zeichen werden in ASCII-Verschlüsselung mit ungerader Parität über die Linie gesendet. Meldungen zwischen den Prozessen LIP und TP sind dabei nach folgender Struktur aufgebaut und werden synchron übertragen:

1. 4* SYNC	Zeichen zur Synchronisation der an der Uebertragung beteiligten Hardware (Interface, Multiplexer)
2. SOH	Beginn der Meldung (start of header).
3. SITE	Hat für diese Verbindung keine Bedeutung. Existiert aus Kompatibilitätsgründen für TP, der auch andere Arten von Datenstationen bedient.
4. STATION	Kann zwei Werte annehmen (141B, 161B); für aufeinanderfolgende Meldungen eines Prozesses wird jeweils der Wert geändert.
5. MSG	Nachricht für den Empfänger-Prozess (s. unten).
6. information	In Abhängigkeit des Wertes von MSG können hier bis zu 480 Zeichen eigentlicher Information übermittelt werden. Die Werte der Zeichen (ohne Parität) müssen grösser sein als 6, damit sie von Steuerzeichen des Protokolls (SOH, EOM) unterscheidbar sind.
7. EOM	Abschluss der Meldung (end of message).
8. checksum	Checksumme (ausschliessendes oder) über alle von SYNC verschiedenen Zeichen.

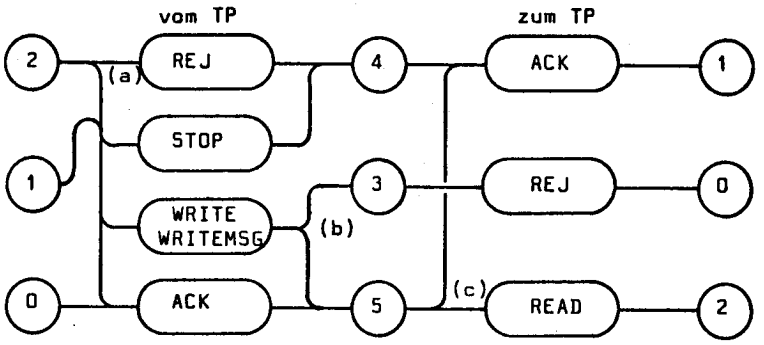
In dieser Sequenz dürfen zwischen 1. und 7. beliebig viele Synchronisationszeichen (SYNC) eingestreut sein und sind beim Empfang zu ignorieren.

Die Positionen 1.-3., 7. und 8. sind feste Bestandteile jeder Meldung und sollen nicht weiter erörtert werden.

Durch den Wert von MSG können die Prozesse angeben, ob in der gleichen Meldung Information übertragen wird, für die der Empfänger einen Puffer benötigt, sie können damit auch bereits übermittelte Information quittieren oder zurückweisen:

MSG	Bedeutung:
WRITE READ	In Position 6. folgt Information; READ und WRITE unterscheiden dabei nur die Richtung der Uebertragung.
WRITEMSG	In Position 6. folgt eine Meldung vom Operateur der Zentralanlage, die auf der Konsole ausgegeben werden soll.
ACK	Quittiert zuletzt empfangene Meldung und besagt, dass vom Sender keine Information zu übermitteln ist.
REJ	Quittiert wie ACK den richtigen Empfang der letzten Meldung, die dort in Position 6. enthaltene Information wurde aber aus lokalen Gründen nicht angenommen (z.B. kein Puffer frei).
STOP	Dient wie ACK als Quittung, die Zentralanlage unterbricht damit die Verbindung für eine gewisse Zeit.

Diese Werte können entsprechend ihrer Bedeutung nicht in beliebiger Reihenfolge übermittelt werden. Es sind aus der Sicht des LIP folgende Sequenzen möglich:



Verzweigungen sind abhängig von
(a) der empfangenen Meldung
(b) der Bereitschaft, Information anzunehmen
(c) dem Vorhandensein zu übermittelnder Information

Abb. 5.2: Mögliche Folgen von Meldungen und entsprechende Zustände des LIP

Die Kreise in Abb. 5.2 bezeichnen die sechs möglichen Zustände des LIP, die Verbindungen stellen mögliche Uebergänge dar.

5.3.2. Starten und Aufrechterhalten der Verbindung

Die beiden Prozesse LIP und TP sind so konzipiert, dass sie beim Starten der beiden Systeme auf dem Satelliten- und Zentralrechner die Verbindung selbständig aufnehmen, sie bei Unterbrechungen und bei erneutem Start einer Anlage wieder aufnehmen; dabei ist soweit wie möglich sicherzustellen, dass keine Information verändert wird, insbesondere darf keine Information verloren gehen oder "erfunden" werden.

Die aktivere Rolle bei Aufnahme und Aufrechterhaltung der Verbindung hat der zentrale TP. LIP wird zu Beginn sowie beim Zusammenbruch der Verbindung auf den Zustand 1 initialisiert, d.h. LIP wartet auf eine Meldung vom TP, sendet dann eine Antwort, usw.

Um zu vermeiden, dass bei Verlust einer Meldung beide Prozesse auf Nachricht des anderen warten, ist beim TP die Zeit des Wartens auf

eine Nachricht vom LIP begrenzt; hat nach Ablauf dieser Zeitspanne die Uebermittlung einer Meldung vom LIP noch nicht begonnen, so schaltet der TP die Leitung um und wiederholt seine letzte Meldung.

Ausserdem wiederholen beide Prozesse ihre letzte Meldung, falls sie einen Informationsblock mit einem nicht erwarteten Wert von STATION erhalten; die Zahl der Wiederholungen einer Meldung ist für den TP begrenzt; wird das Maximum erreicht, bricht TP die Verbindung ab und startet sie nach einiger Zeit erneut. Um die beiden Prozesse bezüglich des Wertes von STATION richtig zu synchronisieren, startet TP jedesmal mit einem anderen Wert, d.h. erwartet LIP irgendeinen Wert und startet TP mit dem falschen, so bricht TP nach einigen Wiederholungen der ersten Meldung die Verbindung wieder ab und startet mit dem anderen Wert.

Folgende Zusammenfassungen zeigen mögliche Fehler und die entsprechenden Reaktionen von Empfänger und Sender:

1. Uebertragung TP --> LIP

Fehler	LIP Empfänger	TP Sender
-	wird nach Abschluss der Meldung zum Sender, ändert Wert von STATION	sendet ganze Meldung
Uebertragungs-fehler (Zeit, Parität oder Checksumme)	ignoriert die Meldung, überliest evtl. folgenden Rest. Wiederholt die letzte Meldung (alter Wert von STATION)	"
anderer Wert von STATION als er-wartet	ignoriert Rest der Meldung (die als bereits empfangen betrachtet wird) und wiederholt die letzte Meldung (alter Wert von STATION)	"
Leitungsunter-bruch	initialisiert Empfang	"

2. Uebertragung LIP --> TP

Fehler	TP Empfänger	LIP Sender
-	wird nach Abschluss der Meldung zum Sender, ändert Wert von STATION nicht	wird nach Ausgabe des letzten Zeichens zum Empfänger
Uebertragungs-fehler	ignoriert Meldung sendet letzte Meldung (ändert Wert von STATION)	"
anderer Wert von STATION als erwartet	wie Uebertragungs-fehler	"
Leitungsunterbruch	bricht nach Erreichen einer Zeitgrenze Verbindung ab.	initialisiert Empfang
Uebertragung startet nicht (Zeit)	wie Uebertragungs-fehler	ist noch auf Empfang oder initialisiert Empfang bei Leitungs-unterbruch

Aus dieser Zusammenfassung geht hervor, dass LIP nach richtigem Empfang einer Meldung den empfangenen Wert von STATION ändert und zurücksendet, während TP den empfangenen Wert zurücksendet und im Fall eines Fehlers STATION ändert, da LIP bei korrektem Empfang von TP den erhaltenen Wert erwartet.

TP und LIP reagieren auf die einzelnen Fehlersituationen verschieden, da die verschiedene Hardware den Prozessen die Fehler nicht gleichartig anzeigt.

6. Erfahrungen und Folgerungen

6.1. Trennung der Aspekte

Die Verwendung einer höheren Sprache zum Entwurf und zur Implementation führte zu einer natürlichen und klaren Trennung der für die Ausführung des gesamten Systems notwendigen Programme in drei Gruppen:

1. Nucleus: umfasst Routinen zur Behandlung maschinenabhängiger Probleme, wie Prozessorverwaltung und Interrupt-Struktur.

2. Driver-Routinen: zur Behandlung peripherieabhängiger und echt zeitabhängiger Aspekte.

3. System: umfasst die Routinen, die aufbauend auf Nucleus und Driver-Routinen die durch die Problemstellung gegebene Aufgabe behandeln.

Diese erste Unterteilung erlaubt die konzentrierte Behandlung eines Problems, auf dessen Lösung dann die Lösung weiterer Probleme aufgebaut werden kann, ohne dass Einzelheiten der ersten Lösung verwendet werden. So erlaubt z.B. die Behandlung der Interrupt-Struktur und Zurverfügungstellung geeigneter Routinen im Nucleus bei der Entwicklung eines Programms zum Betrieb eines peripheren Gerätes die Beschränkung auf wirklich geräteabhängige Probleme.

Entsprechend der Darlegung der einzelnen Programmgruppen in den voranstehenden Kapiteln sind diese selbst wieder unterteilt in Einheiten zur Behandlung eines Aspekts des in dieser Gruppe zu behandelnden Problems. Die Darstellung dieser logischen Einheiten in syntaktischen Programmeinheiten (Modul, Prozess) macht diese Strukturen auch nach der Entwicklung im Programm selbst deutlich.

Diese Modularität erleichtert nicht nur die Entwicklung, sondern auch das Testen von Programmen. Algorithmen und Konzepte sind

leicht auszutauschen oder zu erweitern, wenn das Verhalten des Systems unbefriedigend ist oder eine bessere Lösung für ein Problem gefunden wird. So wurden bei der im Rahmen dieses Projekts ausgeführten und von Hand in Assembler codierten Implementation die Routinen zur Behandlung des Interrupts völlig geändert, ohne zu irgendeinem Fehler zu führen.

6.2. Bemerkungen und Zahlen zur Implementation

Die Programme für die drei Gruppen Nucleus, Driver-Routinen und System wurden unabhängig voneinander entwickelt und bis auf die in Kapitel 2 definierte Sprache verfeinert; aus dieser Darstellung erfolgte von Hand die Uebersetzung in Assembler-Sprache für einen Rechner HP 2115. Dabei wurde die Struktur des Ausgangsprogramms beibehalten und im wesentlichen nicht mehr, zum Teil sogar weniger optimiert, als ein guter Compiler optimieren könnte. Das übersetzte Programm enthält keinerlei Kommentar über die Struktur des Programms in PASCAL-Notation; dadurch wurde bei jeder Aenderung (gezwungenermassen) auf das PASCAL-Programm zurückgegriffen und die Aenderung gegebenenfalls auch dort nachgetragen. Das Finden entsprechender Strukturen in beiden Programmen bereitete dabei in keiner Richtung Schwierigkeiten.

Optimiert wurden:

1.
```
if c1 then
begin ...
      if c2 then S2
            else S
end else S
```

wobei nicht zweimal der Code für S generiert wurde.

2. Prozeduren wurden nur dann reentrant implementiert, wenn es wirklich nötig war, da dies auf einer Maschine ohne Index-Register (HP 2115) sehr umständlich ist. (Bei grösseren Programmstücken mit lokalen Variablen und Parametern wird der

Code um den Faktor 2.5 länger.)

3. Die Abbildung der Programme auf den in Seiten unterteilten Speicher.

Nicht optimiert wurden:

1. Modul-Prozeduren wurden im allgemeinen, auch wenn sie nur eine Zuweisung enthielten, nicht als offene Routinen (Macros) codiert, da dies bei Aenderungen in einem Modul zu Aenderungen im ganzen Assembler-Programm mit der entsprechenden Fehlerwahrscheinlichkeit geführt hätte.

2. Die Verwendung von Registern wurde nur über sehr kurze Folgen von Anweisungen optimiert, um bei Modifikationen die Wahrscheinlichkeit für sekundäre Fehler klein zu halten.

Der Umfang aller für eine Normalstation benötigten Programme beträgt:

	PASCAL-Listing (Zeilen)	HP-2100 Assembler (16-bit Worte)	
Nucleus	240	397	
Driver	360	767	(ohne Puffer)
System	1210	4117	
Puffer		freier Platz bis zu 8k	

Im Driver- und Systemteil gibt es 13 Prozesse mit einer durchschnittlichen Länge von 53 Zeilen, 12 Module, die weder lokale Prozesse noch lokale Module enthalten, mit einer durchschnittlichen Länge von 60 Zeilen, d.h. das System setzt sich aus nicht zu vielen gut überschaubaren Einheiten zusammen. Vier weitere Module mit einer durchschnittlichen Länge von 180 Zeilen fassen andere Module und Prozesse mit zugehörigen Variablen zu Einheiten zusammen, zu denen von aussen kein Zugriff (Modul line) oder nur über spezielle Routinen (Modul lineio) möglich sein soll.

Nach verschiedenen Testversionen von Nucleus und einfachen Erzeuger-Verbraucher-Systemen wurden alle Programme in einem Zug

aus der erweiterten PASCAL-Notation in Assembler übersetzt (6 Wochen) und anschliessend getestet (5 Wochen + 3 Wochen Liniendriver).

Der Liniendriver wurde vom restlichen System getrennt getestet und zwar gleichzeitig mit der Erweiterung des Liniendrivers im Terminalsystem der Zentralanlage. Das stark zeitabhängige Verhalten sowie die Eigenschaft dieser Programme, sich aus einem falschen Zustand selbst wieder in einen richtigen Zustand zu führen, erschwerte das Testen dieser Programme; die PASCAL-Version des Drivers enthielt drei Fehler, einen Fehler in der Behandlung der Linieninterfacekarte durch Fehlinterpretation des Manuals, einen logischen Fehler, durch den die Checksumme verloren ging, wenn sie den Wert des Synchronisationszeichens hatte. Durch die Uebersetzung kam ein dritter Fehler bei der Berechnung des Checksumme hinzu. Die meiste Zeit wurde aber benötigt, um herauszufinden, dass die Modems falsch geschaltet waren.

Die gesamte Testzeit umfasst nicht nur das Testen der Programme für die Datenstation, sondern auch einen Teil des Testens der für HEXAPUS erforderlichen Programme auf der Zentralanlage; diese Programme, die Teil eines Terminalsystems sind, wurden in Batchsimulation des Terminalsystems getestet, wobei aber nicht alle Situationen erfasst werden konnten. Fehler im Gesamtsystem mussten also von der Datenstation aus lokalisiert werden.

Beim Testen des übersetzten Programms erwiesen sich die Modularität und die Existenz eines Listings in PASCAL-Notation als äusserst nützlich:

1. Durch den gleichartigen Aufruf der Driver-Routinen konnten Datenströme leicht auf verschiedenen peripheren Geräten ausgegeben werden, bzw. auf höherem Niveau konnten gleiche Files über verschiedene Ströme ausgegeben und verglichen werden.

2. Durch das Listing in PASCAL-Notation konnten an Hand der Strukturen Fehler leicht lokalisiert werden. Bei logischen Fehlern wurde die Korrektur unter Berücksichtigung der Gesamtstruktur ausgeführt.

3. Das Suchen eines Fehlers beim Testen geschah selten am Rechner oder nur zum Einkreisen eines Fehlers. Durch paralleles Studium von PASCAL- und Assemblerlisting konnten die Fehler wesentlich schneller gefunden werden als bei einem früheren Projekt auf dem gleichen Kleinrechner [Sandmayr 71], da etwa 85% aller Fehler bei der Uebersetzung gemacht wurden.

6.3. Ergänzungen und mögliche Erweiterungen der Sprache

Obwohl beim Projekt HEXAPUS für die in Kapitel 2 definierte Sprache kein Compiler geschrieben wurde, erfolgten jedoch Entwicklung und Korrekturen nur an Hand eines in dieser Sprache geschriebenen Programmes. Aus den dabei gewonnen Erfahrungen empfehlen sich folgende Ergänzungen der Sprache:

1. Konstanten- oder Read-only-Parameter statt oder zusätzlich zu value-Parametern; allgemein sollte zwischen der Implementation der Parameterübergabe und der Schreib- und Leseerlaubnis der Parameter unterschieden werden.

2. Typstrukturen mit Feldern, deren Wert immer oder nach einmaliger Initialisierung konstant sind und entsprechend spezifiziert werden können.

3. Kennzeichnung eines zyklischen Prozess und Beschränkung der Schleife repeat ... end auf solche Prozesse.

4. Konzept zur impliziten Initialisierung von Variablen.

Diese Ergänzungen erlauben dem Programmierer die Angabe seiner Absichten und erhöhen insbesondere bei Verwendung eines Compilers für die Uebersetzung die Sicherheit. So bieten Konstanten- oder read-only- Parameter die Möglichkeit Arrays (durch eine Adresse) effizient als Parameter zu übergeben und bringen zum Ausdruck, dass dieser Array in der Prozedur nicht verändert werden soll, während sonst aus Effizienzgründen - im Gegensatz zur Intention - eine Uebergabe als Variablenparameter vereinbart wird.

Die in Kapitel 2 definierte Sprache lässt sich relativ einfach zu einer Sprache für Echtzeitprogrammierung erweitern, indem man eine Systemzeit und weitere zur Behandlung von Echtzeitproblemen spezifische Konstruktionen einführt, wie

a) Starten von Prozessen in regelmässigen Abständen oder zu (nach) bestimmten Zeiten.

b) Prioritäten für Prozesse entweder allgemein oder beim Warten in einer Schlange, damit auf die Ablaufgeschwindigkeit zeitkritischer Prozesse Einfluss genommen werden kann.

Wesentlich ist auch hier die Reduktion der in einer gewissen, vorgegebenen Zeit auszuführenden Anweisungen auf ein Minimum, so dass für die Ausführung der meisten Anweisungen parallel ablaufender Prozesse die relative Geschwindigkeit keine Rolle spielt und diese Prozesse explizit synchronisiert werden [Dijkstra 71]. Die zeitkritischen Anweisungen sind bei der hier definierten Sprache nur in Driver Moduln möglich, also im Zusammenhang mit Operationen auf peripheren Geräten, durch die ja Zeitabhängigkeit eingeführt wird. Die starken implementationsabhängigen Einschränkungen für Operationen unter peripherer Kontrolle (vgl. 3.1.3) könnten - falls wirklich notwendig - für komplexere Echtzeitanwendungen aufgehoben oder abgeschwächt werden.

6.4. Schlussfolgerungen

Mit der Entwicklung dieses Systems zur Verwendung von Kleinrechnern als Datenstation wurde gezeigt, dass in einer geeignet erweiterten höheren Programmiersprache auf natürliche Art ein modulares portables System geschrieben werden kann (siehe Anhang D). Die dazu eingeführten Erweiterungen dienen drei Zielen:

1) Der Prozess ermöglicht syntaktische Darstellung einer Folge von Operationen, die unabhängig von oder gekoppelt mit anderen Prozessen auszuführen ist.

2) Der Modul ermöglicht eine syntaktische Klammerung von Typen und Operationen auf diesen Typen, wobei aber im Gegensatz zur Class-(Monitor-) Struktur [Dahl 70, Hoare 73] die Operationen nicht Teil des Typs sind.

   Eine einfache Erweiterung, der geschützte Modul, erlaubt den gegenseitigen Ausschluss von Prozessen für die Ausführung kritischer Bereiche und die Queue ermöglicht zusammen mit dem geschützten Modul die explizite Synchronisation von Prozessen.

3) Die Vereinbarung von Schnittstellen für den Zugriff zu peripheren Geräten (device-Vereinbarung) und entsprechender elementarer Operationen ermöglicht die explizite Behandlung peripherer Geräte, ohne die Struktur der Sprache und den damit verbundenen Schutz zu zerstören, wie dies etwa bei Einführen des Zugriffes oder Zuweisungen zu festen Adressen oder Registern oder Einflechten von Anweisungen in einer maschinenorientierten Sprache der Fall wäre.

Weitere Anwendungen müssen zeigen, ob und wie diese Erweiterungen zu modifizieren sind. Ergänzungen sind zu erwarten für die Behandlung

a) komplexerer Rechner, für die der maschinenabhängige Teil grösser wird und selbst so weit wie möglich in einer höheren Sprache geschrieben werden soll.

b) komplexrer peripherer Geräte, für die evtl. neben (oder als Teil) der device-Vereinbarung vordefinierte Funktionen notwendig sind.

c) von Systemen, in denen Daten (Code) als Programme weiterzuverarbeiten sind.

Alle Aenderungen sollten aber die Sprache einfach lassen und einem Compiler erlauben, alle Einschränkungen mit vertretbarem Aufwand zu prüfen. Dann können sehr viele Fehler bereits bei der Uebersetzung abgefangen werden und man erhält zuverlässigere Systeme bei kürzerer Testdauer.

Selbst bei Fehlen eines Compilers empfiehlt sich der Weg über die höhere Programmiersprache, deren Strukturen und Konstruktionen den menschlichen Gedankengängen eher entsprechen als maschinennahe Notationen; Konstruktion, Verifikation und Modifikation sind auf der höheren Stufe einfacher, Zusammenhänge sind besser überschaubar. Für die Konstruktion grosser Systeme oder mehrerer kleiner für einen Rechner sollte aber die Implementation eines Compilers vorgesehen werden, da die meisten Fehler bei der Uebersetzung (85%) gemacht werden. Ausserdem gibt es nach der Uebersetzung in Assembler eine gewisse Sperre, Teile nochmals zu ändern, für die eine bessere Lösung gefunden wurde.

Zur Durchführung eines grossen, mehrere Programmierer umfassenden Projektes werden neben Sprache und Compiler aber noch weitere Hilfsmittel benötigt, um die durch die dann erforderliche teilweise Compilation eingeführten Fehlermöglichkeiten klein zu halten. Als Einheiten für diese teilweise Compilation bieten sich der Modul und der Prozess an, deren Umgebung für den Compiler aber zusätzlich zu spezifizieren ist. Diese bereits in anderen Programmteilen vorhandene Spezifikation sollte nicht vom Programmierer wiederholt werden müssen, sondern dem Compiler in einer "Programmbank" zur Verfügung gestellt werden. Diese Programmbank könnte neben (verschiedenen Versionen von) Programmteilen auch bereits für den Compiler direkt verwendbare Tabellen mit Attributen der im Programmteil definierten und verwendeten Objekte enthalten. Neben dem Quellprogramm würde der Compiler dann Eintragungen dieser Programmbank zu Code und neuen Eintragungen in der Programmbank verarbeiten. Damit lässt sich die durch die Verwendung höherer Sprachen gewonnene Sicherheit auf grosse, in Teilen zu bearbeitende Systeme übertragen.

Literaturverzeichnis

Ammann U.: Die Entwicklung eines PASCAL-Compilers nach der Methode des strukturierten Programmierens. ETH Disseration Nr. 5456, (1975)

Brinch Hansen, P.: Operating System Principles. Prentice-Hall, Englewood Cliffs, N.J., 1973

Brinch Hansen, P.: Concurrent Pascal. A Programming Language for Operating System Design. Information Science Technical Report No. 10, California Institute of Technology, Pasadena, Calif., April 1974

Dahl, O.-J., Myrhaug, B., Nygaard, K.: Simula 67, Common Base Language. Revised Edition. Norwegian Computing Center, 1970

Dahl, O.-J., Dijkstra, E.W., Hoare, C.A.R.: Structured Programming. A.P.I.C. Studies in Data Processing, No. 8, Academic Press, London and New York, 1972

Dijkstra, E.W.: Cooperating Sequential Processes. In: Programming Languages (Ed. F.Genuys) Academic Press, London, 1968, 43-112

Dijkstra, E.W.: The Structure of the "THE"-Multiprogramming System. Comm. ACM 11. (May 1968b), 341-346

Dijkstra, E.W.: Hierachical Ordering of Sequential Processes. Acta Informatica 1 (1971), 115

Hoare, C.A.R.: Proof of Correctness of Data Representations. Acta Informatica 1 (1972), 271-281

Hoare, C.A.R.: Monitors: An Operating System Structuring Concept. Stan-CS-73-401, November 1973

Hoare, C.A.R.: A Structured Paging System.
Comp. J. 16 (1973b), 209-215

Hoare, C.A.R.: Types and Classes.
Vortrag im Kolloquium in Informatik, ETHZ (Mai 1974)

Jensen, K., Wirth, N.: PASCAL, User Manual and Report.
Springer-Verlag, Berlin, Heidelberg, New York, 1974

Naur, P.: Programming by Action Clusters.
BIT 9, No. 3 (1969), 250-258

Naur, P.: Concise Survey of Computer Methods.
Studentlitteratur Lund, Akademisk Forlag København, (1974)

Parnas, D.L.: On the Criteria Used in Decomposing Systems into Modules. Comm. of ACM 15, No. 12 (1972), 1053-1058

Sandmayr, H.: Entwurf und Implementation einer Programmiersprache.
Diplomarbeit am Institut f. Informatik der ETH, 1971

Schicker, P.: VENUS, an Interactive Subsystem.
ETH Dissertation 1975

Wirth, N.: PL 360 - A Programming Language for the 360 Computers. Journ. of ACM 15 (1968), 37-74

Wirth, N.: On Multiprogramming, Machine Coding, and Computer Organizations. Comm. of ACM 12, No. 9 (1969), 489-498

Wirth, N.: The Programming Language PASCAL. Acta Informatica, 1, 35-63, 1971a

Wirth, N.: Programm Development by Stepwise Refinement.
Comm. ACM 14, (April 1971b), 221-227

Wirth, N.: The Design of a PASCAL Compiler.
Software-Practice and Experience 1, No. 4 (1971c), 309-334

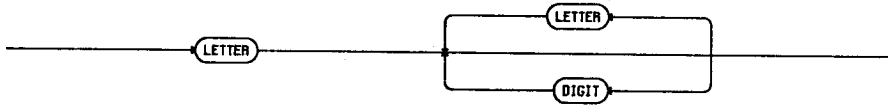
Wirth, N.: On the Design of Programming Languages.
IFIP Congress 74, North-Holland Publishing Company,
Amsterdam (1974), 386-393

Wirth, N.: Internal Working Paper. (1975)

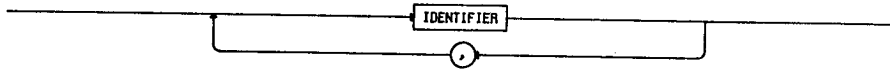
Wulf, W.A., Shaw, M.: Global Variable Considered Harmfull.
SIGPLAN Notices 8, No. 2 (1973), 28-34

Anhang A: Syntaxdiagramme der Implementationssprache

IDENTIFIER

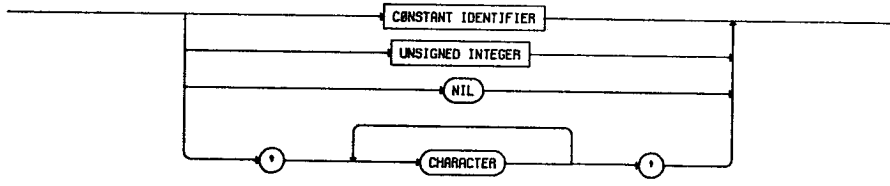


IDENTIFIER LIST

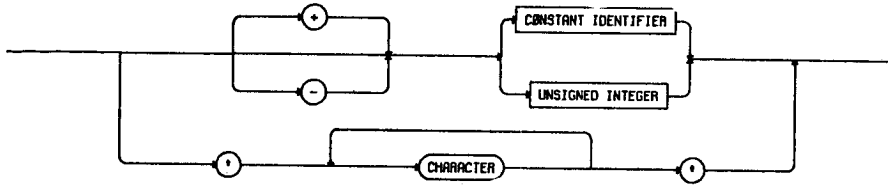


UNSIGNED INTEGER

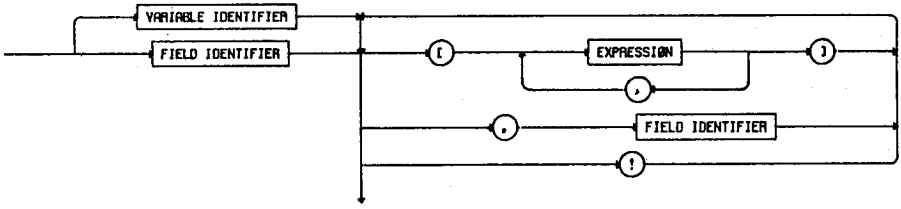
UNSIGNED CØNSTANT



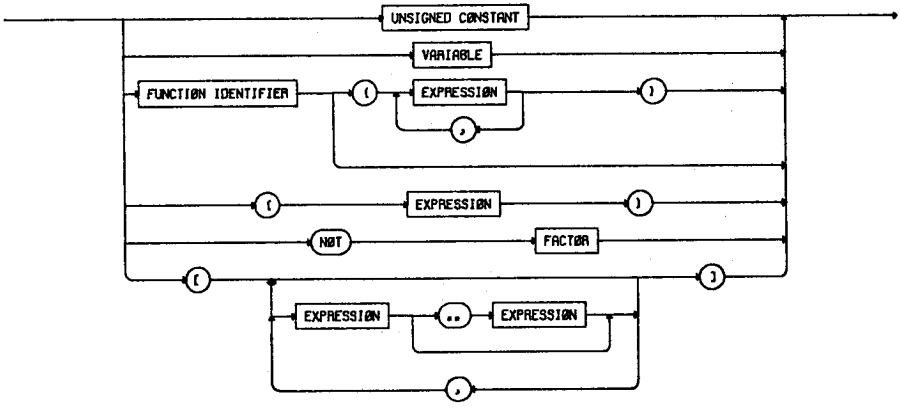
CØNSTANT



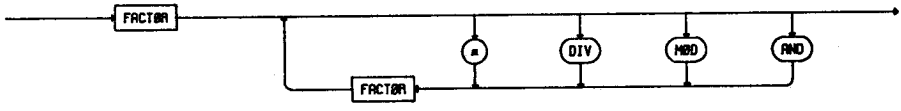
VARIABLE



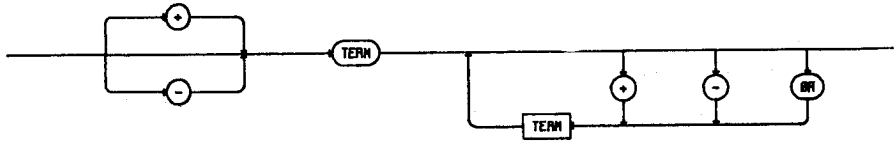
FACTØR



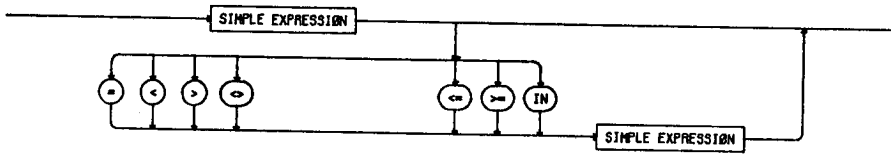
TERM



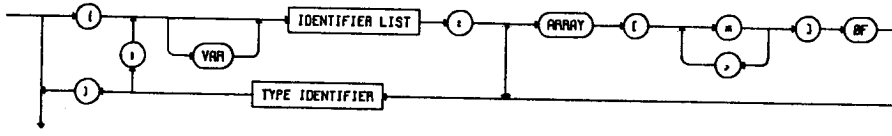
SIMPLE EXPRESSIØN



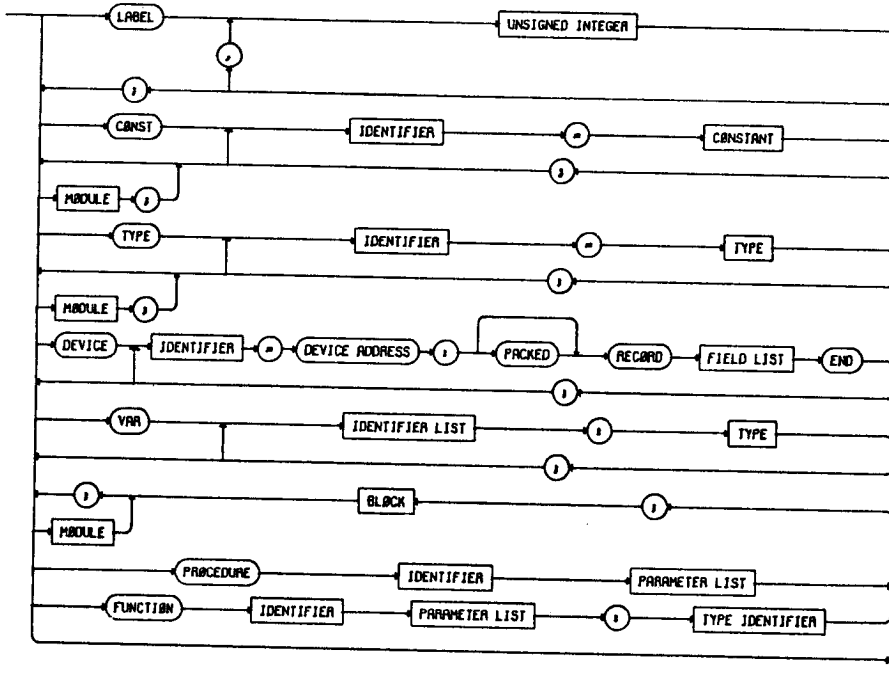
EXPRESSIØN



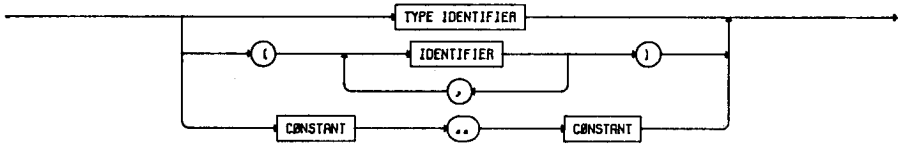
PARAMETER LIST



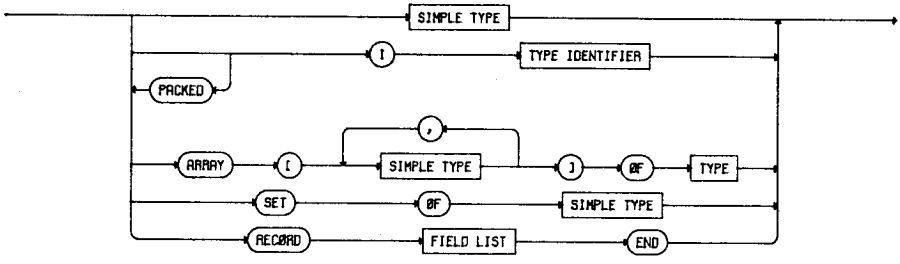
DECLARATIØN PART



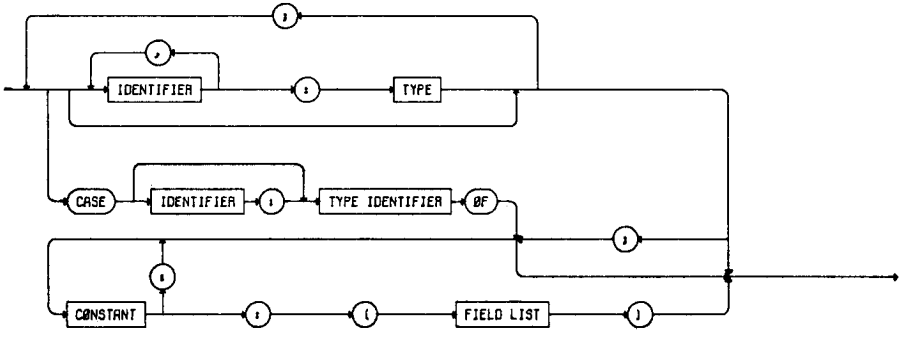
SIMPLE TYPE



TYPE

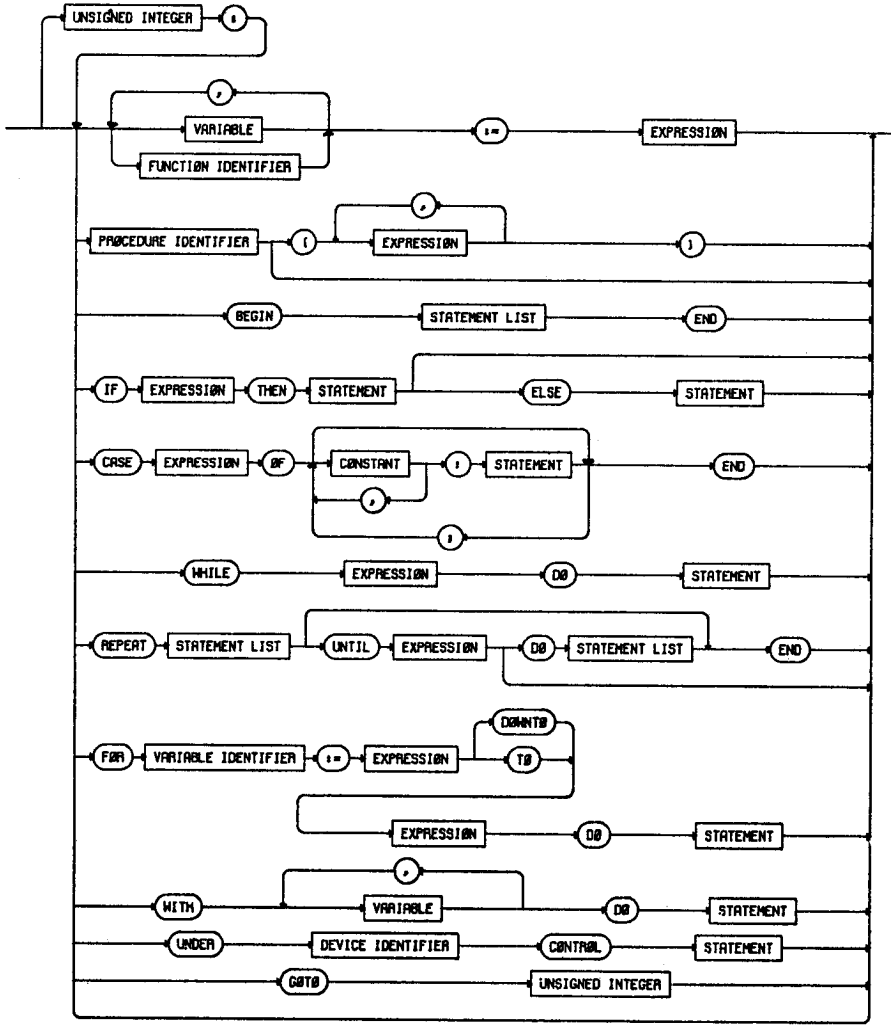


FIELD LIST

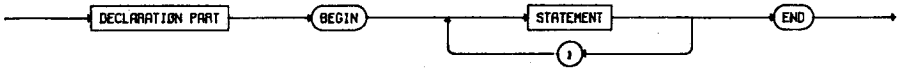


STATEMENT LIST

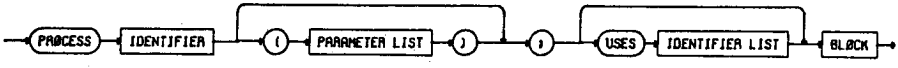
STATEMENT



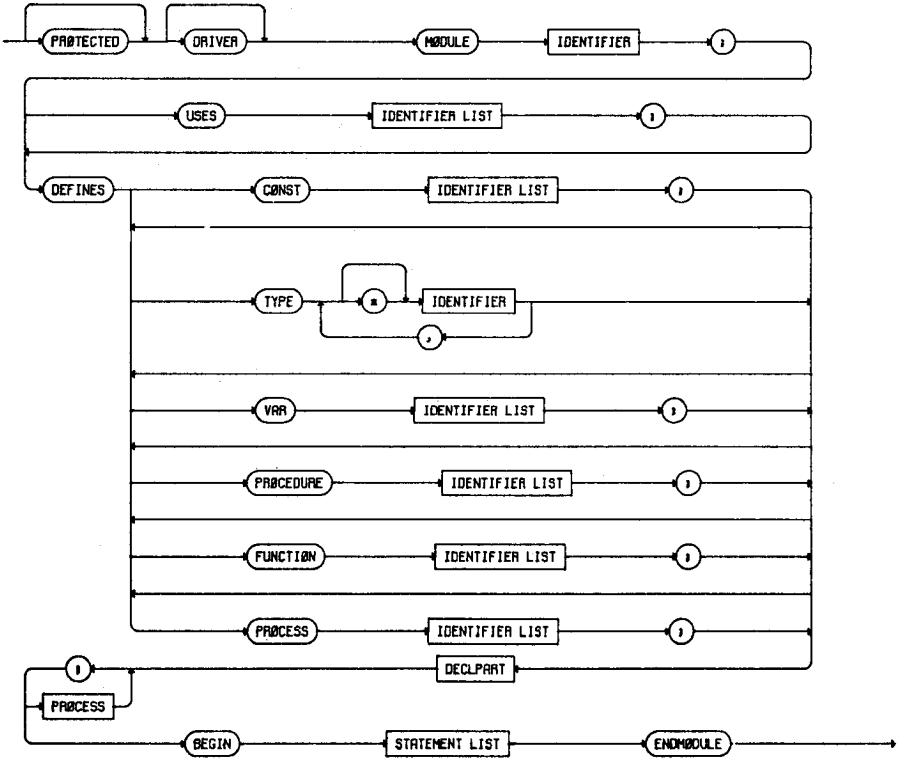
BLØCK



PRØCESS



MØDULE



Anhang B: Auszug aus der Bedienungsanleitung für eine Datenstation

In der implementierten HEXAPUS-Version existieren folgende Prozesse:

Output-Prozesse

DMY Der Dummy Output Prozess dient als Zielprozess, wenn man irgendwelche Daten zu Testzwecken nur einlesen will (z.B. für Paritätstests) und keinen Output benötigt.

PPT Punch Paper Tape ermöglicht die Ausgabe von Information im ASCII-Code mit gerader (E) bzw ungerader (O) Parität, sowie von binär kodierter Information. Die Zeichen .EOR. und .EOF. werden in Folgen von je 40 Leerzeichen aufgelöst.

Parameter:

O ASCII mit ungerader Parität
E ASCII mit gerader Parität

PRT Der Print-Prozess listet ein File auf dem Printer. Das erste Zeichen einer Zeile wird als Steuerzeichen interpretiert, wenn nicht der Parameter U gesetzt wird.

Die ASCII- Zeichen haben folgende Steuerfunktionen:

'+': keine neue Zeile beginnen
'0': neue Zeile nach einschalten einer Leerzeile
'-': neue Zeile nach einschalten von zwei Leerzeilen
'2': Sprung auf die letzte Zeile einer Seite
'1': Sprung auf erste Zeile einer neuen Seite

Alle anderen Zeichen bewirken normale Zeilenfortschaltung.

LO1 Der Linienoutputprozess ermöglicht die Verbindung eines Datenstromes zur Inputqueue der Zentralanlage.

Input-Prozesse

CRD Der Karteninputprozess ermöglicht das Einlesen von Karten, die in einem der CDC-Formate gelocht sind. Eine spezielle Karte (.EOR.- Karte ab Kolonne 1 überlocht mit P,x., wobei x ein für den Prozess RPT definierter Parameter ist (A|E|O|B)) erlaubt das Einfügen von Lochstreifeninformation in das von Karten her einzulesende File.

Wird ein Streifen erwartet, so wird die Meldung 'CRD PT' auf der Konsole ausgegeben und der Kartenleser wartet bis zum Ende eines Streifens. Nach dem Einlegen des Streifens in den Leser ist der Streifenlese-Prozess mit 'GO,RPT.' zu starten. Das Streifenende ergibt im Inputfile weder ein .EOR. noch ein .EOF. .

Die Verknüpfungen des Kartenlesers können nicht von der Konsole her gesteuert werden, da Verknüpfungen für ein ganzes File, d.h. Kartendeck, gelten und gleichzeitig mehrere Decks im Kartenleser bereitliegen können. Eine spezielle Steuerkarte ermöglicht die lokalen Verknüpfungen des Kartenlesers (.EOR.-Karte überlocht mit CONCT,CRD,op. , wobei op der Name irgendeines Outputprozesses ist). Fehlt diese Karte am Anfang eines Kartenpaketes, so wird der Kartenleser mit dem Prozess LO1 verbunden.

RPT Read-Paper-Tape ermöglicht das Einlesen von binären und ASCII kodierten Streifen. ASCII-Streifen können ausserdem auf gerade (E) oder ungerade (O) Parität geprüft werden. Bei ASCII - Streifen ist zu beachten, dass gewisse Steuerzeichen (mit Verschlüsselung kleiner als 40b) für die interne Verarbeitung vom System verwendet werden und daher bei Eingabe (34b, 35b, 36b) bzw. Uebermittlung (< 10b) ignoriert werden.

Das Ende eines Streifens wird durch (mindestens) 20 Nullzeichen (NUL) angegeben und wird auch als EOI aufgefasst, wenn nicht der Streifen als Teil eines Kartenfiles eingelesen wird.

Parameter:

A ASCII Streifen (Default)
E ASCII Streifen mit gerader Parität
O ASCII Streifen mit ungerader Parität
B binär kodierter Streifen

LI1, LI2, LI3 Die Linieninputprozesse ermöglichen die Verbindung von der Outputqueue der Zentralanlage zu den peripheren Geräten des Satelliten. Diese Prozesse werden mit abnehmender Priorität von der Zentralanlage mit Information bedient. Sie sind mit der Zentralanlage über je einen Datenstrom verknüpft, über den Files mit einem bei der Definition des Datenstromes festgelegten Dispositioncode übermittelt werden.

Implementierte Befehle

In der folgenden Zusammenfassung sind fakultative Parameter in geschweifte Klammern gesetzt. IP, OP und P stehen für einen Input-, Output- oder beliebigen (Strom-) Prozessnamen. DC steht in den Befehlen für einen der implementierten Dispositioncodes:

PR, P1, P2 (print, , Print ASCII)
LA, LB (Lochstreifen ASCII bzw. binär)
SC (scratch)

JN steht für Jobname, der in folgenden Formaten akzeptiert wird:

b Buchstabe; z Ziffer: {b}zzzz{bb}

PR steht für Priorität. Es wird eine Oktalzahl erwartet, von der die vier letzten Ziffern akzeptiert werden.

CNCT,IP{(b)},OP{(b)}{,DC,JN}. CNCT verbindet und startet zwei Prozesse. Ein bereits aktiver Inputprozess kann nicht verknüpft werden. Dagegen können meherere Inputprozesse gleichzeitig mit einem (auch gerade aktiven) Outprozess verknüpft werden. Die Verbindungen werden dann nacheinander aktiviert.

b steht für einen möglichen Parameter des zu verknüpfenden Prozesses (siehe Prozessdefinition).

Falls IP ein Linieninputprozess ist, werden die Parameter DC und JN akzeptiert, und es wird ein entsprechender Datenstrom definiert. Fehlt DC, so wird ein Strom für die ganze Gruppe von Dispositioncodes definiert, die zum entsprechenden Outputprozess gehören (PRT: Ø5, PPT: Ø4). JN ermöglicht das selektive Oeffnen eines Stromes für nur eine Kontonummer.

DSCT,P. DSCT löst die Verbindung zwischen dem Prozess P und dem mit P verknüpften Prozess. Wenn die Verbindung gerade

aktiv ist, wird sie erst nach Abschluss der Uebermittlung gelöst.

Ist einer der involvierten Ströme ein Linieninputstrom, so wird ausserdem die Stromdefinition aufgehoben.

WAIT,P. WAIT dient zur momentanen Unterbrechung eines Prozesses (,d.h. einer Unterbrechung zu Beginn eines nächsten Buffers). Die Wirkung ist nicht für alle Prozesse definiert.

WEOJ,P. WEOJ stoppt einen Prozess nach beendeter Verarbeitung eines Files. Bei PPT und RPT wird dieser Befehl vom System für jedes File implizit gegeben, damit ein verarbeiteter Streifen entnommen bzw. ein neuer eingelegt werden kann.

GO,P. GO startet einen unterbrochenen Prozess, d.h. es hebt die durch WAIT oder WEOJ gesetzte Sperre auf.

TERM,P. TERM beendet die Arbeit an einem gerade aktiven File sofort. Der noch nicht verarbeitete Teil des Files geht verloren. Bei Printlistings wird aber noch das Dayfile gedruckt.

Ist P ein Linieninputstrom, so wird der Rest des Files auf alle Fälle unterdrückt und der Outputprozess steht für die nächste Verbindung zur Verfügung. Damit kann ein Outputprozess verfügbar gemacht werden, wenn die Verbindung zur Zentralanlage zusammengebrochen ist.

Die folgende Gruppe umfasst die Befehle, die in der Zentralanlage auszuführen sind. Sie sind daher nur sinnvoll, wenn der angegebene Prozess selbst oder der mit ihm verknüpfte ein Linienprozess ist.

REW,P. REW bewirkt, dass ein File das gerade von der Zentralanlage her übermittelt wird, wieder in die Outputqueue gewiesen wird.

BS,P,N. Der Arbeitspunkt an dem gerade bearbeiteten File wird auf der Zentralanlage um N PRU's zurückgesetzt.

LIST,{,JN{,DC}}. LIST liefert die Namen aller zugreifbaren Files, die mit dem evtl. angegebenen Dispositioncode und entsprechenden Jobnamen in der Zentralanlage vorhanden sind und zum Satelliten gehören.

Fehlen DC und JN, so wird eine Liste aller Jobs gesendet, die zum Satelliten gehören. (Diese Liste kann sehr lang sein.)

ABT,JN. Falls ein Job mit dem Jobnamen JN in der Inputqueue oder an einem Kontrollpunkt existiert, wird er eliminiert.

EVCT,JN,DC. Ein Job mit Jobnamen JN und Dispositions-Code DC wird aus der Outputqueue gestrichen.

ENPR,JN,DC,PR. Der durch DC und JN spezifizierte Job in der Oetputqueue erhält die Priorität PR.

DVT,JN,DC,b. Der durch DC und JN spezifizierte Job wird auf die durch den Buchstaben b charakterisierte Anlage geschickt. JN muss dazu aber der eigenen Anlage zugewiesen sein.

EXIT. EXIT bewirkt sofortigen Abbruch der Verbindung zur Zentralanlage. Files die gerade übermittelt werden, können dabei verloren gehen.

Zur erneuten Aufnahme der Verbindung muss der Satellit neu gestartet werden, während lokale Arbeiten fortgesetzt werden können.

Meldungen auf die Konsole

Neben der Startmeldung kommen vom Venusmodul weitere Meldungen, z.B. nach dem Einlesen eines Jobs (Jobname) und nach der Ausgabe eines Files (Jobname RELEASED). Ausserdem können Meldungen erscheinen, die auf einen Eingabe- oder System- Fehler hinweisen.

Lokale Meldungen

Jede lokale Meldung wird eingeleitet durch den Namen des Prozesses, der die Meldung veranlasste. Fehlt dieser Namen, so bezieht sich die Meldung auf eine Konsoleingabe. Der Rest der Meldung bedeutet:

CH Checksumfehler beim Einlesen von binären Karten.

CM Illegaler Befehl (command) wurde eingegeben.

CN Es wurde versucht, einen aktiven Inputprozess zu verknüpfen (connect).

DC Ein illegaler Dispositioncode wurde eingegeben.

IL Eingelesener ASCII-Streifen enhält Zeichen 34B,35B oder 36B, die ignoriert wurden.

LF (line full) Befehl konnte nicht übermittelt werden.

NP Der Printer wird benötigt. (need printer)

PA Paritätsfehler bei Lochstreifeninput.

PC Illegale Punchkombination (auf einer Karte).

PN Ein illegaler Prozessname wurde eingegeben.

PT Lochstreifen einlegen und Prozess RPT starten.

RT In einem Inputprozess wurde versucht, binäre und kodierte Information zu mischen. (record type)

SQ Sequenzfehler beim Einlesen binärer Karten.

SY Syntaxfehler bei der Eingabe eines Befehles.

TE (transmission error). Informationsblock mit falscher Sequenznummer wurde empfangen. Ausgabe evtl. unvollständig.

TL (tape low) Lochstreifenstanzer nachfüllen. Nach Eingabe von 'GO,PPT' arbeitet der Streifenstanzer weiter.

Beispiele:

```
PRT NP
CRD RT
    CN
```

Anhang C: Vollständiges Programm des Liniendrivers

```
driver module lineio;
  defines type *lbufptr;
          procedure readline, writeline;

  type lbufptr= ↑array [1..480] of bit8;
  device lni = PG2: record ich: bit16 end;
         lno = PG3: record och: bit16 end;
  var acceptedobuf, filledibuf: queue;
      inbuf, outbuf: lbufptr;
      inx: integer;
      outcnt: integer;

   process lip;
     label 1,2,3;
     const soh= 1; site= 160b; sync= 26b; read= 23b; ack= 6;
           write= 21b; opmsg- 22b; stop=24b; eom= 3; rej= 30b;
     var station: integer {141b,161b}; ch, msg: bit8;
         reject: Boolean; lsend: (acc, idle, sent);
         state: 0..5; sum, i: integer;
         answer: array [0..2] of bit8;

     procedure waiteom;
     begin sum:= -1; repeat nextch(ch) until ch=eom
     end {waiteom};

     procedure nextch(var ch: bit8);
     begin repeat start(lni)
           until not "interface in hunt for sync mode";
       if "modem or line down" then goto 1 {leave ircrtl};
       if "parity error" then ch:= 0 else
       begin ch:= ich mod 200b; sum:= sum xor ch end
     end {nextch};

     procedure send(ch:bit7);
     begin
         och:= ch; sum:= sum xor ch;
         start(lno);
```

```
      if "line down" then
      begin lsend:= idle; goto 3 end
  end {send};

begin answer:= (rej, ack, read); station:= 141b;
  state:= 1; lsend:= idle;
  repeat "switchoff transmit interface";
  1:repeat "initialize receive interface and modem";
      while not "ready modem" do pause(5);
      under lni control
      begin sum:= 0; reject:= false;
        "set interface in hunt for sync mode";
        nextch(ch);
        if ch=soh then
        begin nextch(ch);
          if ch=site then
          begin nextch(ch);
            if ch=station then
            begin nextch(msg);
              if msg in [rej,write,opmsg,ack,stop] then
              begin nextch(ch);
                if msg<>rej then
                begin if lsend <> idle then lsend:= acc;
                  state:= 1
                end;
                if msg in [write,opmsg] then
                begin
                  if inbuf = nil then reject:= true else
                  begin inx:= 1;
                    while not (ch in [0,eom])
                            end (inx <= 480) do
                    begin inbuf↑[inx]:= ch; inx:= inx+1;
                      nextch(ch)
                    end;
                    reject:= ch=0;
                  end;
                  if reject then
                    while ch <> eom do nextch(ch)
                end;
                if ch<>eom then waiteom
```

```
            end else waiteom
          end else waiteom
        end else waiteom
      end else waiteom;
      "switch interface to process sync characters";
      {checksum to be read may be equal to sync:}
        nextch(ch);
      "switchoff receive interface"
    end;
    if sum <> 177b then goto 2:  {repeat transmission}
    if lsend = acc then
    begin lsend:= idle; startq(acceptedobuf)
    end;
    case msg of
      rej:   if state=2 then state:= 4;
      write,opmsg:
             if state <> 0 then
               if not reject then
               begin if msg=opmsg then inbuf↑[0]:= 36b;
                 startq(filledibuf); state:= 5;
               end else state:= 3;
      stop:  if state<>0 then state:= 4;
      ack:   state:= 5
    end
  until state>=3;
  state:= state - 3;
  station:= 322b - station;   {161b <-> 141b}
2:"initialize send interface and modem";
  while not "ready modem" do pause(5);
  under lno control
  begin for i:= 1 to 4 do send(sync);
    sum:= 177b; send(soh); send(site);
    send(station);
    if (state=2) and (outbuf=nil) then state:= 1;
    msg:= answer[state]; send(msg);
    if msg = read then
    begin
      for i:= 1 to outcnt do
        if (outbuf[i] mod 200b) > 6 then
          send(outbuf[i] mod 200b);
```

```
            lsend:= sent
          end else lsend:= idle;
          send(eom); send(sum);
      3:end
      end
    end {lip};

    procedure readline(b:lbufptr; var cnt: integer);
    begin inbuf:= b; enterq(filledibuf);
      inbuf:= nil; cnt:= inx-1
    end {readline};

    procedure writeline(b: lbufptr; cnt: integer);
    begin
      if cnt > 0 then
      begin outcnt:= cnt; outbuf:= b;
        enterq(acceptedobuf); outbuf:= nil
      end
    end {writeline};

begin  initq(acceptedobuf); initq(filledibuf);
  inbuf, outbuf:= nil; start lip
endmodule {lineio}.
```

Anhang D: Struktur des Programms HEXAPUS für eine Datenstation

Die folgende Programmskizze soll einen zusammenfassenden Ueberblick über die Beziehungen zwischen den in den Kapiteln 4 und 5 besprochenen Programmkomponenten vermitteln; dazu ist von jeder Komponente der Vereinbarungskopf, der die Beziehung zu den umgebenden Programmteilen beschreibt, mit einer die Funktion(en) der Komponente erläuternden Anweisung versehen. Ein Kommentar verweist jeweils auf den Abschnitt der Arbeit, in der der entsprechende Programmteil behandelt wird.

```
module HEXAPUS;

  driver module peripheral;                    {vgl. Kapitel 5}
    {existiert für jedes periphere Gerät}
    defines readperipheral, {and/or} writeperipheral;
    "Prozeduren zur Ein- und/oder Ausgabe auf einem
     peripheren Gerät"
  endmodule {peripheral};

  module bufferhandling;                       { 4.2.1 }
    defines type bufferqueue, *pbuffer;
            procedure initbufferqueue, initpb, swap,
                      get, put, gete, pute;
    "Typen für Block- und Verbindungspuffer;
     Prozeduren zur Initialisierung und zum Austausch
     von Puffern"
  endmodule {bufferhandling};

  var carea: array ["outputprocesses"] of bufferqueue;
    {Verbindungspuffer zum Austausch von Informationsblöcken}

  module connectcontrol;                       { 4.2.2 }
    defines procedure connect, disconnect,
                      openconnection, closeconnection,
                      getconnection, getspecification;
    "Tabellen und Prozeduren zur Steuerung der Verknüpfung
     von Datenströmen bzw. entsprechender Prozesse"
  endmodule {connectcontrol};
```

```
module runcontrol;                                  { 4.2.3 }
  defines type *rstate;
          procedure wait, condset,
                    terminate, stop, stopeoj, go;
  "Tabellen und Prozeduren zur Steuerung des Ablaufs der
   den Datenströmen entsprechenden Prozessen"
endmodule {runcontrol};

protected module tty;
  uses writetty;
  defines procedure puttty, msg;
  "Prozeduren zur Ausgabe von Meldungen auf der Konsole"
endmodule {tty};

module line;                                        { 4.3 }
  uses carea, lineio, tty,
       bufferhandling, connectcontrol, runcontrol;
  defines procedure cmdlnoØ;
  "Prozeduren und Prozesse zur Verwaltung interner Linienströme,
   zur Abbildung interner Linienströme auf einen externen
   und umgekehrt"
endmodule {line};

process output;                                     { 4.2.4 }
  uses writeperipheral, carea, tty,
       bufferhandling, connectcontrol, runcontrol;
  "für jeden (nicht Linien-) Ausgabestrom existiert ein
   solcher Prozess zur Verwaltung dieses Stromes und
   für die notwendigen Umformungen der Darstellung
   der Information"
end {input};

protected module cmdinterpreter;                    { 4.4 }
  uses cmdlnoØ, tty, connectcontrol, runcontrol;
  defines interpret;
  "Tabellen und Prozeduren zur Interpretation und Weitergabe
   von Befehlen"
endmodule {cmdinterpreter};
```

```
  process input;                                    { 4.2.4 }
    uses readperipheral, carea,
         {cardinput uses also: cmdinterpreter}
         bufferhandling, connectcontrol, runcontrol;
    "für jeden (nicht Linien-) Eingabestrom existiert ein
     solcher Prozess zur Verwaltung des Stromes und der
     eventuell notwendigen Umformung der Information auf
     ASCII-Code"
  end {input};

  process readcmd;
    uses readtty, cmdinterpreter;
    "liest bedingt Befehle von der Konsole und lässt sie
     im Modul cmdinterpreter ausführen"
  end {readcmd};

begin for "all outputprocesses" do
        initbufferqueue(carea["process"]);
  start "inputprocesses", "outputprocesses", readcmd
endmodule {HEXAPUS}.
```

Lebenslauf

Ich wurde am 12. April 1947 in Konstanz (BRD) geboren. Von 1953 bis 1957 besuchte ich in Laufenburg/Baden die Volksschule, daran anschliessend in Säckingen das Scheffel-Gymnasium, das ich 1966 mit dem Abitur abschloss. An der Albert-Ludwigs-Universität in Freiburg/Brsg. war ich von 1966 bis 1969 für Mathematik immatrikuliert und wurde 1968 auf Vorschlag von Prof. Dr. K. Kirchgässner ins Cusanuswerk, eine deutsche Studienstiftung, aufgenommen. Nach weiteren vier Semestern an der Eidgenössischen Technischen Hochschule Zürich erwarb ich im Oktober 1971 das Diplom als Mathematiker.

Hier wurde ich dann Assistent II am Institut für Informatik und war an verschiedenen Entwicklungsprojekten beteiligt. Seit Mitte 1973 arbeitete ich am Projekt HEXAPUS und führte unter der Leitung von Prof. Dr. N. Wirth die vorliegende Arbeit aus.

Seit April 1975 bin ich Dozent für Mathematik und Informatik am Neu-Technikum Buchs.