ATARI®

ATARI PROGRAMM

DXG 20102
2 Disketten

PASCAL

(c) 1983 Jegliche Rechte vorbehalten
ATARI ELEKTRONIK - Vertriebsges. mbH

Re-Edit by GoodByteXL, Juli 2Ø2Ø

ATARI PROGRAMM

DXG 2Ø1Ø2
2 Disketten

PASCAL

(c) 1983 Jegliche Rechte vorbehalten
ATARI ELEKTRONIK - Vertriebsges. mbH

INHALTSVERZEICHNIS

Kapitel 1: ATARI PASCAL, Einführung und Überblick 1

1.1 Manualüberblick 2
1.2 Systemüberblick 3
1.3 Systemanforderungen 4
1.4 Laufzeitanforderungen 4
1.5 Inhalt der gelieferten Pascaldisketten 4

Kapitel 2: Bedienung des Pascalsprachsystems 6

2.1 Compilieren, Binden und Starten des Beispielprogramms 7
2.1.1 Compilieren des Beispielprogramms 7
2.1.2 Binden des Beispielprogramms 9
2.1.3 Starten des Beispielprogramms 10

2.2 Compilerbedienung 11
2.2.1 Aufruf und Filenamen 11
2.2.1.1 DOS und QUIT Optionen 11
2.2.1.2 Compilierung 11
2.2.2 Compilierungsdaten 12
2.2.3 Compilerschalter (-Optionen) 12
2.2.3.1 Erzeugung der Einstiegspunktrecords(E) 12
2.2.3.2 Aufnahme von Files (I) 13
2.2.3.3 Strenge Typ- und Portabilitätsüberwachung (T,W) 13
2.2.3.4 Bereichsüberprüfung während der Laufzeit (R) 13
2.2.3.5 Ausnahmeüberprüfung während der Laufzeit (X) 14
2.2.3.6 Listingkontrollen (L,P) 14
2.2.3.7 Zusammenfassung der Compilerschalter 14
2.2.4 Eingebaute Routinen und einschließen von Files 15
2.2.5 Fehlermeldungen 15
2.2.6 Zeilennummern 15

2.3 Linkerbedienung 16
2.3.1 Aufruf und Kommandos 16
2.3.2 Linkeroptionen 16
2.3.2.1 Suche nach benötigten Laufzeitbibliotheksmodulen (/S) 16
2.3.2.2 Memory Map (/M) 17
2.3.2.3 Load Map (/L) und erweitertes Load Map (/E) . 17
2.3.2.4 Programm- (/P) und Datenursprung (/D) 17
2.3.2.5 Fortsetzung von Zeilen (/C) 18
2.3.2.6 Linker-Kommandofile (/F) 18
2.3.2.7 Zusammenfassung der Linkeroptionen 18
2.3.2.8 Anforderungen relokatibler Files 18
2.3.2.9 Linkerfehlermeldungen 19
2.3.2.10 Eigenschaften der linkfähigen Module 19

2.4 Starten des Objektprogramms 20

2.5 ATARI Programm-/Texteditor 20
2.5.1 Start des ATARI Programm-/Texteditors 20

Kapitel 3: Erweiterungen des ATARI Pascalsprachsystems 21

3.1 Modulare Compilation 22

3.2 Datenabspeicherung und Parameterübergabe 25
3.2.1 Datenabspeicherung 25
3.2.2 Parameterübergabe 26

3.3 Programmaufteilung und -verkettung 28

3.4 Eingebaute Prozeduren und Parameter 3Ø
3.4.1 MOVE, MOVERIGHT, MOVELEFT 3Ø
3.4.2 EXIT 32
3.4.3 TSTBIT, SETBIT, CLRBIT 33
3.4.4 SHR, SHL 34
3.4.5 HI, LO, SWAP 35
3.4.6 ADDR 36
3.4.7 SIZEOF 37
3.4.8 FILLCHAR 37
3.4.9 LENGTH 38
3.4.1Ø CONCAT 38
3.4.11 COPY 39
3.4.12 POS 39
3.4.13 DELETE 4Ø
3.4.14 INSERT 41
3.4.15 ASSIGN 41
3.4.16 WNB, GNB 42
3.4.17 BLOCKREAD, BLOCKWRITE 42
3.4.18 OPEN 43
3.4.19 CLOSE, CLOSEDEL 43
3.4.2Ø PURGE 43
3.4.21 IORESULT 44
3.4.22 MEMAVAIL, MAXAVAIL 44
3.4.23 Schneller Überblick über alle eingebauten Prozeduren und Parameter (innerhalb jeder Gruppe alphabetisch geordnet.) 45

3.5 Nichtstandardgemäßer Datenzugriff 46
3.5.1 Absolute Variable 46

3.6 INLINE 46
3.6.1 Syntax 46
3.6.2 Anwendungen 46

3.7 Grafik- und Soundroutinen 47
3.7.1 Bildschirmtypen 48
3.7.2 Variable 48
3.7.3 Grafikprozeduren und -funktionen 49
3.7.3.1 Initialisierungsprozedur 49
3.7.3.2 Grafikprozedur 49
3.7.3.3 Textmodusprozedur 49
3.7.3.4 Setcolorprozedur 49
3.7.3.5 Colorprozedur 5Ø
3.7.3.6 Plotprozedur 5Ø
3.7.3.7 Locateprozedur 5Ø
3.7.3.8 Positionprozedur 5Ø
3.7.3.9 Drawtoprozedur 51
3.7.3.1Ø Fillprozedur 51
3.7.4 Soundprozeduren und -funktionen 51
3.7.4.1 Soundprozedur 51

3.7.4.2 Soundoffprozedur 52
3.7.5 Controllerfunktionen 52
3.7.5.1 Paddle 52
3.7.5.1.1 Paddlefunktion 52
3.7.5.1.2 Triggerfunktion 52
3.7.5.2 Joysticks 52
3.7.5.2.1 Stickfunktion 52
3.7.5.2.2 Triggerfunktion 52

Kapitel 4: Laufzeitfehlerbehandlung 53
4.1 Bereichsüberprüfung 54
4.2 Ausnahmeüberwachung 54
4.3 Benutzerdefinierte Fehlerbehandlung 54
4.4 Schwerwiegende Fehler 54

Kapitel 5: Struktur und Format eines Pascalprogramms .. 55

5.1 Datentypen 55
5.1.1 CHAR 55
5.1.2 BOOLEAN 55
5.1.3 INTEGER 56
5.1.4 REAL 56
5.1.5 BYTE 56
5.1.6 WORD 56
5.1.7 STRING 56
5.1.7.1 Definition 56
5.1.7.2 Zuweisung 57
5.1.7.3 Vergleiche 58
5.1.7.4 Lesen und Schreiben von Strings 59
5.1.8 SET 60

Kapitel 6: Kompatibilität 61

6.1 Inkompatibilitäten zu UCSD Pascal 62
6.2 Im ATARI PASCAL Sprachsystem zusätzlich zur Verfügung stehende Eigenschaften des ATARI PASCAL 63

Kapitel 7: Sprachdefinition 65

7.1 Einführung 65
7.2 Zusammenfassung der Sprache ATARI PASCAL 65
7.3 Notation, Terminologie und Vokabular 67
7.4 Bezeichner, Zahlen und Strings 67
7.5 Konstantendefinitionen 68
7.6 Datentypdefinitionen 68
7.6.1 Einfache Typen 68
7.6.1.1 Skalartypen 68
7.6.1.2 Standardtypen 69
7.6.1.3 Unterbereichstypen 69
7.6.2 Strukturierte Typen 69
7.6.2.1 Arraytypen 70
7.6.2.2 Recordtypen 70
7.6.2.3 Settypen 70
7.6.2.4 Filetypen 70
7.6.3 Zeigertypen 71
7.6.4 Typ- und Zuweisungskompatibilität 71

7.7 Deklaration und Bezeichnung von Variablen ... 73
7.7.1 Ganze Variable 73
7.7.2 Komponentenvariable 73
7.7.2.1 Indizierte Variable 73
7.7.2.2 Feldbezeichner 74
7.7.2.3 Filebuffer 74
7.7.3 Bezugsvariable 74

7.8 Ausdrücke 75
7.8.1 Operatoren 75
7.8.1.1 Der Operator NOT 75
7.8.1.2 Multiplikationsoperatoren 75
7.8.1.3 Additionsoperatoren 75
7.8.1.4 Vergleichsoperatoren 76
7.8.2 Funktionsbezeichner 76

7.9 Statements 76
7.9.1 Einfache Statements 76
7.9.1.1 Zuweisungsstatements 76
7.9.1.2 Prozedurstatements 77
7.9.1.3 GOTO Statements 77
7.9.2 Strukturierte Statements 77
7.9.2.1 Zusammengesetzte Statements 77
7.9.2.2 Bedingungstatements 77
7.9.2.2.1 IF Statements 77
7.9.2.2.2 CASE Statements 77
7.9.2.3 Wiederholungsstatements 78
7.9.2.3.1 WHILE Statements 78
7.9.2.3.2 REPEAT Statements 78
7.9.2.3.3 FOR Statements 78
7.9.2.4 WITH Statements 78

7.1Ø Prozedurdeklarationen 79
7.1Ø.1 Standardprozeduren 81
7.1Ø.1.1 Filemanagementprozeduren 81
7.1Ø.1.2 Dynamische Datenzuweisungsprozeduren 81
7.1Ø.1.3 Datentransferprozeduren 81
7.1Ø.2 FORWARD 81
7.1Ø.3 Angleichbare Arrays 82

7.11 Funktionsdeklarationen 83
7.11.1 Standardfunktionen 83
7.11.1.1 Arithmetische Funktionen 83
7.11.1.2 Boolesche Funktionen 83
7.11.1.3 Übertragungsfunktionen 83
7.11.1.4 Weitere Standardfunktionen 83

7.12 Ein- und Ausgabe 84
7.12.1 Die Prozedur READ 84
7.12.2 Die Prozedur READLN 84
7.12.3 Die Prozedur WRITE 84
7.12.4 Die Prozedur WRITELN 84
7.12.5 Zusätzliche Prozeduren 85

7.13 Programme 85

Anhang A: Sprachsyntaxbeschreibung 86

Anhang B: Reservierte Wörter 93

Anhang C: Fehlermeldungen 94

Anhang D: ATARI PASCAL Fileein- und ausgabe 1Ø2

Anhang E: Bibliografie 117

Anhang F: Demoprogramm für Player/Missilegrafik 118

Anhang G: Hilfreiche Hinweise 125

Stichwortverzeichnis 126

Abbildungen:

Abb. 1-1: Schema der ATARI PASCAL Operationen 3

Abb. D-1: Fileein- und ausgabe 1Ø5

Abb. D-2: Textfiles 113

Abb. D-3: Druckerausgabe und Zahlenformatierung 115

VORWORT

Was ist PASCAL ?

Pascal wurde von Niklaus Wirth entwickelt, um einen systematischen Ansatz der Computerprogrammierung und der Problemlösung zu erleichtern. Diese stark strukturierte Programmiersprache eignet sich für professionelle Softwareentwickler und stellt somit ein ausgezeichnetes Werkzeug für Programmentwicklung und -wartung dar.

Zweck dieses Handbuches

Dieses Referenz- und Bedienungshandbuch definiert die Spracheigenschaften von ATARI-Pascal und soll Ihnen helfen, diese Eigenschaften zu nutzen. Es setzt die Kenntnis von Jensens und Wirths 'Pascal User Manual and Report' und/oder des ISO Standard Konzepts DPS/7185 voraus. Die standardmäßigen Pascaleigenschaften, in denen sich ATARI PASCAL von denen im Standard oder in Jensens und Wirths 'Report' unterscheidet, sind hier beschrieben. Dieses Handbuch enthält auch Informationen über die Bedienung von Compiler und Linker, eine Beschreibung der in ATARI PASCAL implementierten Datentypen und eine Zusammenfassung der eingebauten Eigenschaften zusammen mit Beispielen für ihre Benutzung.

Zielgruppe

Dieses Handbuch ist speziell für fortgeschrittene Programmierer geschrieben, die mit PASCAL und dem ATARI Homecomputersystem vertraut sind.

Gebrauch dieses Handbuches

Wir empfehlen, mit der Einführung und dem Überblick (Kapitel 1) zu beginnen. Fahren Sie dann mit Kapitel 2 fort, das die Bedienung des Systems beschreibt, Empfehlungen für Sicherungskopien gibt und ein Beispielprogramm für den Anfang enthält.
Der Rest dieses Handbuches ist rein formal und sollte, je nach Bedarf als Nachschlagewerk verwendet werden.

Produktbetrachtungen

Das ATARI PASCAL Sprachsystem wurde für den Gebrauch durch erfahrene Programmentwickler gemacht. Die Schritte, die erforderlich sind, um ein ATARI PASCAL Programm zu compilieren, sind zeitraubend. Speicherbeschränkungen, Diskettenkapazität und Zugriffszeit werden die Produktleistung beeinflussen.

Falls Probleme auftreten

Alle dokumentierten und an 'The ATARI Programm Exchange' eingesandten Probleme werden bearbeitet und bei zukünftigen Änderungen dieses Produkts beachtet werden.

KAPITEL 1: ATARI PASCAL EINFÜHRUNG UND ÜBERBLICK

Dieses Handbuch beschreibt das ATARI PASCAL Sprachsystem.

ATARI PASCAL ist ein Pseudocodecompiler, der dem ISO Standardkonzept entspricht, indem er Variant Records, Sets, Type- und Textfiles, Prozeduren und Funktionen als Parameter, Sprünge zum Verlassen einer Prozedur, angleichbare Arrays und Programmparameter einschließt. Zusätzlich sind folgende Erweiterungen implementiert:

Skalartypen: Byte, Word, String
Operatoren für Integer: & (and), !, / (or), ? (not)
else im case-statement
leere Strings
absolute Variable
externe Prozeduren

Vordefinierte Prozeduren und Funktionen:

Grafik-, Ton- und Controllerdefinitionen
Reelle und transzendente Funktionen
Bewegungs- und Füllprozeduren
Bit- und Bytemanipulationen
Filemanipulationen
Heapmanagementhilfen
Stringmanipulationen
Adressen- und Sizeoffunktionen

Modulare Compilationsmöglichkeiten

Zusätzlich sorgt die Runtimefehlerbehandlung für Überprüfung auf Division durch Null, Heap- und Stringoverflow, Bereichsüberprüfung und benutzerdefinierte Fehlerroutinen.

ATARI PASCAL ist für Datenverarbeitungsanwendungen bestimmt und besteht aus Compiler-, Editor- , Linker-, Geschäfts- und Wartungspaketen. Es ist für das Arbeiten mit dem ATARI DOS 2.ØS geschrieben und mit dem ATARI Programm- und Texteditor (Wz) kompatibel.

Dieses Kapitel bietet einen Überblick über dieses Handbuch, das System, Compilierungs- und Laufzeitsystemanforderungen und beschreibt die Disketteneinteilung.

Da über die Sprache PASCAL viele Bücher erhältlich sind, ist dieses Handbuch nicht als Lehrbuch, sondern eher als Nachschlagewerk zu verstehen und als detaillierte Beschreibung der Erweiterungen und Zusätze, die ATARI PASCAL einzigartig machen. Schlagen Sie in der Bibliografie nach, um weitere Hinweise auf Lehrmaterial zu erhalten.

1.1 MANUALÜBERBLICK

Im Folgenden wird ein kurzer Überblick über den Inhalt aller Kapitel dieses Handbuches gegeben.

Kapitel 1: Dieses Kapitel bietet eine Einführung und skizziert die Eigenschaften von ATARI PASCAL und einen Überblick über das System zusammen mit einer Definition der Systemanforderungen.

Kapitel 2: Hier fangen Sie richtig an. Es werden die Compiler- und Linkeroptionen beschrieben mit allen Schritten, die notwendig sind, ein Programm zu compilieren, zu binden und zu starten.

Anm.: 'Binden' und das aus dem englischen Sprachgebrauch bekannte 'Linken' bzw. 'Linker' werden im Folgenden synonym gebraucht.

Kapitel 3: Dieses Kapitel beschreibt die Erweiterungen des ATARI PASCALs, z.B. modulare Compilation, vordefinierte Prozeduren, Grafik- und Soundmöglichkeiten.

Kapitel 4: Dieses Kapitel bietet eine kurze Zusammenfassung der Laufzeitfehlerbehandlung.

Kapitel 5: Dieses Kapitel beschreibt die Struktur eines Programms, das durch den Compiler erzeugt wird. Außerdem wird die Datenspeicherung behandelt.

Kapitel 6: Ein kurzer Vergleich zwischen ATARI- UND UCSD-PASCAL.

Kapitel 7: Dieses Kapitel definiert die Spracheigenschaften von ATARI-PASCAL.

Anhang A : Eine komplette Beschreibung der Sprachsyntax.

Anhang B : Liste der reservierten Wörter.

Anhang C : Eine komplette Beschreibung der Compilerfehlermeldungen.

Anhang D : ATARI-PASCAL File Input/Output.

Anhang E : Bibliografie zusätzlicher Literaturvorschläge.

Anhang F : Spieler/Raketen Demoprogramm.

Anhang G : Hilfreiche Hinweise

1.2 SYSTEMÜBERBLICK

Das ATARI PASCAL Sprachsystem beinhaltet den Pascalmonitor, Compiler, Linker, Laufzeitunterprogrammbibliothek und den Interpreter.

Abb. 1-1 zeigt die Beziehungen zwischen diesen Teilen. Der ATARI Programm-Texteditor wurde eingeschlossen, um seine Beziehung zum ATARI PASCAL Sprachsystem zu zeigen.

```
+------------+
I Texteditor I
+------------+
      ¦
      ¦
      ¦
      ¦
  Quelldatei
      ¦
  +-------+
  I Com-  I  <--> vorübergehende Arbeitsdatei
  I piler I  <--> Fehlermeldungsdatei
  +-------+
    ¦
    ¦ ------------>   relokatible   Laufzeit-
    ¦                   Datei       bibliothek
Protokollfile             ¦            ¦
                          ¦            ¦
                         +--------------+
                         I    Linker    I
                         +--------------+
                                ¦
                                ¦
                          ausführbares
                             Programm
                                ¦
                                ¦
                         +-------------+
                         I Interpreter I
                         +-------------+
```

Abb. 1-1 Schema der ATARI PASCAL Operation.

Anm.: Um dem englischen Sprachgebrauch zu entsprechen, werden im Folgenden 'relokatibel' und 'verschiebbar' synonym verwendet.

Der ATARI Programm-Texteditor wird verwendet, um ein PASCAL-Quellprogramm zu erstellen und zu verändern. Der Compiler übersetzt das Quellprogramm in relokatiblen Maschinencode. Über den Linker verbindet der Benutzer diesen Code mit der Laufzeitbibliothek zum ausführbaren Objektcode.

1.3 SYSTEMANFORDERUNGEN

Das ATARI PASCAL Sprachsystem benötigt den ATARI mit 48 KByte RAM und zwei ATARI Diskettenlaufwerke. Der ATARI 8Ø-Zeichendrucker ist optimal.

ATARI PASCAL benötigt auch den ATARI Programm-Texteditor. Während der Benutzung des ATARI PASCALs sollte kein ROM Modul in den Modulschacht eingesteckt sein.

1.4 LAUFZEITANFORDERUNGEN

Das ATARI PASCAL Sprachsystem erzeugt Programme, die eine Anzahl Laufzeitunterprogramme benutzen, die der Bibliothek PASLIB entnommen werden, und andere relokatible Module. Diese Laufzeitroutinen erledigen Operationen, wie Multiplikation und Division, sowie Datei I/O Übergabe an das Operationssystem.

1.5 INHALT DER ATARI PASCAL DISKETTEN

Das ATARI PASCAL Sprachsystem wird auf Disketten geliefert, die zum ATARI Laufwerk kompatibel sind. Das System besteht aus zwei Disketten, die Objekt-, Quell- und relokatible Dateien enthalten.

Im Folgenden die Namen der Dateien zusammen mit einer kurzen Beschreibung der Inhalte.

Diskette 1

Name	Inhalt
DOS.SYS	ATARI Disk Operating System
DUP.SYS	ATARI Disk Operating System
PASCAL	Interpreter für die Ausführung aller PASCAL Objektprogramme
MON	PASCAL Monitor, der von PASCAL geladen wird. Er sorgt für die Ausgabe des Menüs, um die gewünschte Operation aufzurufen (compile, link, edit oder run).
LINK	PASCAL Linker, erzeugt aus relokatiblen Files (.ERL) und der Laufzeitbibliothek Objektfiles (.COM).
LINK.OVL	Zweiter Teil des PASCAL Linkers.
PASLIB.ERL	Laufzeitunterprogrammbibliothek in relokatibler Form, sollte immer zuletzt eingebunden werden.
FPLIB.ERL	Laufzeitunterstützungsroutinen für Fließkommaarithmetik und transzendente Funktionen.

GRSND.ERL	Laufzeitunterstützungsroutinen für Grafik-, Sound- und Controllerfunktionen.
CALC.PAS	Quelldatei für das PASCAL-Demoprogramm.

DISKETTE 2

Name	Inhalt
PHØ	Phase Ø des PASCAL-Compilers, wird benutzt, für einen Syntaxüberblick und die Erzeugung eines Tokenfiles.
PH1	Phase 1 des PASCAL-Compilers, wird benutzt, um die permanenten Symboltabellen zu erzeugen und die Benutzersymbole aufzubauen.
PH2	Phase 2 des PASCAL-Compilers, enthält die Initialisierung der Codeerzeugung.
PH3	Phase 3 des PASCAL-Compilers erzeugt die relokatible Objektcodedatei.
PH4	Phase 4 des PASCAL-Compilers, beendet die Objektcodeerzeugung.
ERRORS.TXT	enthält ATASCII Texte für Fehlermeldungen.
GSPROCS	einschließbares File, enthält Grafik-, Ton- und Controllerdefinitionen.
FLTPROCS	einschließbares File, enthält Funktionsdeklarationen für reelle Zahlen und transzendente Funktionen.
MOVES	einschließbares File, enthält Deklarationen für Charakterarrays.
BITPROCS	einschließbares File, enthält Deklarationen für Bitmanipulationsroutinen.
HEAPSTUF	einschließbares File, enthält Deklarationen für Heapprozeduren.
DSKPROCS	einschließbares File, enthält Dateimanipulationen.
STDPROCS	einschließbares File , enthält PASCAL-Standardroutinen und Fließkommaroutinen.
ISOPROCS	einschließbares File, enhält ISO PASCAL-Standardroutinen ohne Fließkommaroutinen.
STRPROCS	einschließbares File, enthält Funktionen und Prozeduren für die Stringverarbeitung.

KAPITEL 2: BEDIENUNG DES PASCALSPRACHSYSTEMS

Dieses Kapitel beschreibt die Bedienung des ATARI PASCAL Sprachsystems, das auf den PASCAL/Linker und PASCAL-Compilerdisketten gespeichert ist. Es behandelt folgende Themen:

Abschnitt 1 zeigt durch schrittweise Instruktionen, wie man ein Programm compiliert, bindet und startet.

Abschnitt 2 beschreibt den Compiler und seine Optionen.

Abschnitt 3 beschreibt den Linker und seine Optionen.

Abschnitt 4 beschreibt den Start eines Objektprogramms.

Abschnitt 5 beschreibt den ATARI Programm-Texteditor.

2.1 COMPILIEREN, BINDEN UND STARTEN DES BEISPIELPROGRAMMS

Machen Sie vor dem Compilieren und Binden des Beispielprogramms eine Sicherungskopie aller in diesem Paket enthaltenen Disketten.

2.1.1 COMPILIEREN DES BEISPIELPROGRAMMS

Schritt eins:

Legen Sie die PASCAL/LINKER Diskette in Laufwerk 1 und starten Sie das Diskettenbetriebssystem DOS 2.ØS. Benutzen Sie dann Option C, um das Beispielrechnungsprogramm 'CALC.PAS' auf eine leere Diskette in Laufwerk 2 zu kopieren. Benutzen Sie jetzt Option L, um das File 'PASCAL' von Laufwerk 1 zu laden.

Darauf erscheint das PASCAL Menü:

```
                ATARI PASCAL
           Version 1.Ø : 1-Mar-82
             (c) 1982 by ATARI

    E>dit                  C>ompile
    L>ink                  R>un
    D>os                   Q>uit

    Enter Letter and RETURN
```

Schritt zwei:

Antworten Sie auf das auf dem Schirm angezeigte Menü mit dem Kommando 'C' RETURN, um die Compilation zu beginnen. Wenn Sie nach dem Namen des Quellfiles gefragt werden, antworten Sie mit 'D2:CALC.PAS' RETURN.

Der Monitor wird Sie dann nach den Namen für Token- und Codefile fragen. Antworten Sie für beides mit RETURN.

Dann wird die Meldung 'Change D1 to Compiler Disk' ausgegeben. Legen Sie jetzt den Pascalcompiler (Diskette 2) in Laufwerk 1, vergewissern Sie sich, daß das Beispielprogramm 'CALC.PAS' in Laufwerk 2 liegt und drücken Sie RETURN.

Darauf wird der Compiler in den Speicher geladen und fordert Sie auf, ein Gerät für das Protokoll zu wählen. Antworten Sie 'P:' (printer) für Drucker, 'E:' (screen), oder RETURN (kein Protokoll).

Der Compiler fährt mit der Meldung der folgenden Compilationsstatistik fort:

```
Loading Compiler
----------------------------
       ATARI Pascal
    Version 1.Ø -1-Mar-82
     (c) 1982 by ATARI
----------------------------

Syntax Scan

Creating: D2:CALC.TOK
Listing file , P: or E:
RETURN for none

File does not contain line numbers
<    Ø>.............
Including Text from File: D1:STDPROCS
<   14>.................
<   32>.............................
<   64>.............................
<   96>.............................
<  128>.............................
End of Phase Ø           (Syntax Überprüfung/Tokengeneration)
Source lines processed: 132
Loading Phase I
Open as input: D2:CALC.TOK
Open as output: D2:CALC.ERL
Available Memory: 4387     (Gesamtplatz für Symboltabelle)
User table space: 3264     (nach vordefinierten Symbolen)
Version 1.Ø Phase 1
#####                    (ein # für jedes Programmsegment)
Remaining Memory 21ØØ    (nach Benutzersymbolen)
Version 1.Ø, Phase 2
SUBREAL           18
ADDREAL           43
TF                64    (dezimale Verschiebung vom Beginn)
CALC             119
MENU             915
CALCULAT
External:        TRUNC
External:        SQRT
External:        SIN
External:        ROUND
External:        OUTPUT
External:        LN
External:        INPUT
External:        EXP
External:        COS
External:        ARCTAN

Lines :          13Ø
Errors:            Ø
Code  :         1737
Data  :           64
REPLACE D1 THEN              (legen Sie Diskette 1 PASCAL/
Type RETURN to continue       LINKER in Laufwerk 1 und
                              drücken Sie RETURN)
```

Minuten später

Das System wird Sie auffordern, die Diskette in Laufwerk 1 auszutauschen und für die Fortsetzung RETURN zu drücken. Nehmen Sie nun die Compilerdiskette aus Laufwerk 1, legen Sie dort die PASCAL/LINKER-Diskette ein und drücken Sie RETURN.

Die Compilierung wird dann beendet und das PASCAL-Menü ausgegeben.

Bemerkung: Wenn der Compiler die Compilation nicht beendet, überprüfen Sie, ob die Disketten in den richtigen Laufwerken sind. Wenn ja, versuchen Sie die Eingabe SYSTEM RESET. Wenn diese beiden Versuche scheitern , bleibt als einzige Möglichkeit, den Rechner aus- und wieder einzuschalten.

2.1.2 BINDEN DES BEISPIELPROGRAMMS

Schritt eins:

Um eine relokatible, d. h. eine nicht an feste Speicheradressen gebundene Objektdatei zu erzeugen, antworten Sie auf das PASCAL-Menü mit der Eingabe 'L' RETURN, um den Linkerlauf zu starten. Jetzt wird das Folgende angezeigt:

```
Loading Linker
when Linker prompts with '*' enter
your .ERL file names separated by
commas ending with PASLIB/S

Then type RETURN

LINKER V1.0
```

Wenn Sie durch ein Asterisk (*) nach dem Filenamen gefragt werden, brauchen Sie Erweiterung (.ERL) nicht zu benutzen, müssen aber das Laufwerk angeben "D2:".

Die Pascalbibliotheksroutinen müssen dann zusammen mit Ihrem Programm gebunden werden.

Antworten Sie auf die Frage nach dem Filenamen mit:

```
D2:CALC,FPLIB,PASLIB/S RETURN
```

Bemerkung: Dieses Programm kann als Beispiel für die Benutzung der Fließkommabibliotheksroutinen (FPLIB) dienen.

Der Linker wird dann die folgenden Werte und die Meldung 'LINK COMPLETE TYPE RETURN' ausgeben.

```
D2:CALC.ERL          <48A7H>
D1:FPLIB.ERL         <2FFAH>
D1:PASLIB.ERL        <1F50H>

UNDEFINED SYMBOLS

-- NO UNDEFINED SYMBOLS --
```

```
11405 bytes written to D2:CALC.COM

Total Data    :   00BEH bytes
Total Code    :   2BCEH bytes
Remaining     :   1442H bytes

Link complete type RETURN
```

Drücken Sie nun RETURN und das PASCAL-Menü wird erscheinen.

2.1.3 STARTEN DES BEISPIELPROGRAMMS

Um das Beispielprogramm zu starten, antworten Sie auf das PASCAL-Menü mit dem Kommando 'R' RETURN um das Objektprogramm zu starten.

Sie werden nach dem Filenamen gefragt werden und sollten mit Folgendem antworten:

```
D2:CALC.COM
```

Das Berechnungsprogramm beginnt seine Ausführung mit der Meldung 'Enter first Operand?'. Versuchen Sie als Beispiel 5.5 zu 99.256 zu addieren. Antworten Sie zuerst mit '5.5' RETURN. Die Meldung 'R1 = 5.500E+00' sollte erscheinen, gefolgt von 'Enter second Operand?'. Antworten Sie mit '99.256' RETURN. Die Meldung 'R2 = 9.92560E+01' sollte erscheinen gefolgt von 'Enter Operator' und einer Liste von Operatoren. Antworten Sie mit '+' RETURN. Das Ergebnis '104.756' sollte darauf angezeigt werden. Drücken Sie nun ESCAPE, um zum DOS zurückzukehren.

Sie haben nun die Compilation, das Binden und Ausführen Ihres ersten ATARI PASCAL Programms vollendet.

2.2 COMPILERBEDIENUNG

2.2.1 AUFRUF UND FILENAMEN

Das ATARI PASCAL Sprachsystem wird unter dem ATARI Diskettenbetriebssystem (DOS 2.ØS) ausgeführt. Um den Compiler zu starten, legen Sie den PASCAL/LINKER (Diskette 1) in Laufwerk 1 und laden ('LOAD') das File mit dem Namen PASCAL aus dem DOS Menü . Dieses File ist der Pascalinterpreter, der automatisch den Pascalmonitor mit dem Namen MON aufruft. Der Monitor zeigt dann folgendes Menü:

```
          ATARI PASCAL
      Version 1.Ø : 1-Mar-82
        (c) 1982 by ATARI

   E>dit              C>ompile
   L>ink              R>un
   D>os               Q>uit

   Enter Letter and RETURN
```

Wählen Sie die gewünschte Funktion und geben Sie ihren ersten Buchstaben, gefolgt von RETURN ein.

2.2.1.1 DOS UND QUIT OPTIONEN

Die 'DOS' und 'QUIT' Operationen gestatten Ihnen, das PASCAL-Menü zu verlassen und zum ATARI Diskettenbetriebssystem zurückzukehren.

2.2.1.2 COMPILIERUNG

Wenn Sie 'c' für 'Compile' wählen, wird Sie der Monitor nach drei Filenamen fragen und dann den Compiler laden. Die erste Frage gilt dem Quellfilenamen . Antworten Sie mit dem Filenamenprefix (D2:), dem Eingabefilenamen und der Erweiterung (.PAS). Dann fragt der Compiler nach den Namen für Token- und Codefiles. Falls genug Platz auf der Diskette ist, die das Quellfile enthält, können Sie durch einfaches Drücken von RETURN auf diese Fragen antworten. Wenn nicht genug Platz vorhanden ist, können Sie bestimmen, daß die Files auf verschiedenen Disketten abgelegt werden, indem Sie die gewünschten Filenamen vollqualifizierend angeben . Bemerkung: Keines der Compilerfiles darf auf Cassette abgelegt sein.

Darauf wird die Meldung 'Change D1 to Compiler Disk' ausgegeben. Legen Sie nun den Pascalcompiler (Diskette 2) in Laufwerk 1, die Diskette, die Ihr Quellprogramm enthält in Laufwerk 2 und drücken Sie RETURN. ATARI PASCAL erzeugt dann ein relokatibles File (name).ERL, das durch den PASCAL-Linker mit den Routinen der Laufzeitbibliothek zu einem ausführbaren Programm gebunden werden muß.

2.2.2 COMPILIERUNGSDATEN

Der ATARI PASCAL-Compiler wird während der ersten beiden Phasen (Phase Ø und Phase 1) der Compilation in Abständen Zeichen ausgeben.

Ein Punkt (.) wird auf dem Schirm für die Syntax jeder in Phase Ø überprüften Zeile angezeigt. Zu Beginn der Phase 1 wird der zur Verfügung stehende Speicherplatz angezeigt. Das ist die Anzahl der freien Speicherbytes (dezimal) vor Erzeugung der Symboltabelle. Ungefähr 1 K Byte der Symboltabelle wird durch vordefinierte Bezeichner verbraucht. Sobald eine Funktion oder Prozedur erkannt ist, wird ein No. Zeichen (#) auf dem Schirm angezeigt. Am Ende der ersten Phase wird die Anzahl der noch freien Speicherbytes dezimal ausgegeben.

Phase 2 generiert den Objektcode. Wenn eine Prozedur erkannt ist, wird der Name der Prozedur ausgegeben, so daß Sie wissen, wie weit der Compiler mit der Compilation ist. Die Linkeroption /M (Map) listet die absoluten Adressen der Funktionen in jedem Modul. Nach dem Ende werden folgende Zeilen ausgegeben:

```
Lines : Anzahl der compilierten Quellzeilen (dezimal).
Errors: Anzahl der gefundenen Fehler.
Code  : Länge des generierten Codes (dezimal).
Data  : Länge des Datenbereichs (dezimal).
```

2.2.3 COMPILERSCHALTER (-OPTIONEN)

Ein Compilerschalter kann in das Quellprogramm aufgenommen werden, um den Compiler anzuweisen, gewisse Operationen auszuführen oder zu unterlassen. Das Format dieses Schaltbefehls ist (*$____*), mit '_' als Platzhalter für den aktuellen Schaltbefehl. Der Compiler akzeptiert keine führenden Leerzeichen vor dem Schlüsselwort oder Leerzeichen in bzw. nach dem Namen; führende Leerzeichen werden übersprungen z. B. (*$E +*) ist dasselbe wie (*$E+*), aber (*$ E+*) wird ignoriert.

Beispiele:

```
(*$E+*)
(*$P*)
(*$I D:USERFILE.LIB*)
```

2.2.3.1 ERZEUGUNG DER EINSTIEGSPUNKTRECORDS (E)

$E+ und $E- kontrollieren die Erzeugung von Einstiegspunktrecords im relokatiblem File. $E+ führt dazu, daß alle globalen Variablen, alle Prozeduren und Funktionen als Einstiegspunkte zur Verfügung stehen (d. h. daß auf sie durch EXTERNAL-Deklarationen in anderen Modulen zugegriffen werden kann. $E- unterdrückt die Erzeugung dieser Records und führt somit dazu, daß alle diese Variablen, Funktionen und Prozeduren logisch lokal sind. Von

vornherein befindet sich der Schalter in der $E+ Stellung und kann je nach Bedarf ein- oder ausgeschaltet werden.

2.2.3.2 AUFNAHME VON FILES (I = INCLUDE)

$I<filename> veranlaßt den Compiler das genannte File in die Sequenz des Quellprogramms aufzunehmen. Die Angabe des Filenamens schließt die Laufwerknummer und die Erweiterung im Standardformat ein.

Folgendes Format gilt:

```
(*$IDn:XXXXXXX*)

   oder

(*$IDn:XXXXXXX.PAS*)
```

mit n:Laufwerknummer,XXXXXXX:Name des aufzunehmenden Files.

Durch Gebrauch dieser Beispiele für Standard Fileaufnahmeprozeduren können Sie Ihre eigenen Files erzeugen, die während der Compilation benutzt werden.

2.2.3.3 STRENGE TYP- UND PORTABILITÄTSPRÜFUNG (T,W)

$T+, $T-, $W+ und $W- kontrollieren die strenge Typüberprüfung und die Warnung vor Nichtübertragbarkeit. Diese Einrichtungen sind fest aneinander gekoppelt (d. h. Die strenge Typüberprüfung impliziert die Warnung vor Nichtübertragbarkeit und umgekehrt). Von vornherein ist der Zustand der Schalter $T- ($W-), in dem die Typüberprüfung gelockert ist und keine Warnungen erzeugt werden. Während des Quellcodes können diese Einrichtungen je nach Wunsch ein- und ausgeschaltet werden. Der Gebrauch einer nicht standardmäßigen Logik und/oder eingebauter Routinen führt zur Erzeugung der Fehlermeldung 5ØØ. Dieser Fehler ist nicht schwerwiegend, dient aber als Warnung für den Programmierer.

Der Code, während dessen Compilation der Fehler 5ØØ auftaucht, wird trotzdem korrekt ausgeführt.

2.2.3.4 BEREICHSÜBERPRÜFUNG WÄHREND DER LAUFZEIT (R)

$R+ und $R- kontrollieren während der Compilation die Erzeugung des Codes, der Arrayindizierung und Abspeicherung in Unterbereichstypen durchführt. Ohne Definition ist der Schalter im Zustand $R- (aus) und kann während des Quellcodes je nach Wunsch ein- und ausgeschaltet werden.

2.2.3.5 AUSNAHMEÜBERPRÜFUNG WÄHREND DER LAUFZEIT (X)

$X+ und $X- kontrollieren während der Compilation die Erzeugung des Codes, der die Überprüfung und Handhabung sogenannter 'Ausnahmen' durchführt. Diese Ausnahmen sind:

Division durch Null
Stringoverflow/abtrennung
Heapoverflow

Die Systemphilosophie unter der ATARI PASCAL arbeitet, sieht vor, daß Division durch Null und Stringoverflow in einer 'vernünftigen' Weise behandelt werden, wenn die Ausnahmekontrolle abgeschaltet ist. Division durch Null ergibt den Maximalwert für den Datentyp und Stringoverflow führt zu einer Abtrennung statt zu einer Beeinflussung angrenzender Speichergebiete. Ohne Definition ist der Schalter im Zustand $X- (aus) und kann während des Quellcodes je nach Wunsch ein- und ausgeschaltet werden. Siehe Kapitel 4 für mehr Information über Laufzeitfehlerbehandlung und Optionen.

2.2.3.6 LISTINGKONTROLLEN (L,P)

Die $P, $L+ und $L- Schalter kontrollieren das durch den ersten Pass des Compilers erzeugte Listing. $P führt zur Einfügung eines Vorschubcharakters (chr(12)) in das .PRN-File. $L+ und $L- werden benutzt um das Listing während des Quellcodes ein- und auszuschalten und können nach Belieben im Quellprogramm platziert werden.

2.2.3.7 ZUSAMMENFASSUNG DER COMPILERSCHALTER

Im Folgenden eine zusammenfassende Auflistung aller zur Verfügung stehenden Compilerschalter:

Schalter		Ausgangs-stellung
$E +/-	kontrolliert Einstiegspunkterzeugung	$E+
$I<name>	schließt ein bekanntes Quellfile mit in die Eingabe ein (z. B. (*$1 XXX.LIB*)	
$R +/-	kontrolliert den Bereichsprüfungscode	$R-
$T +/- $W +/-	kontrollieren die strenge Typüberprüfung und die Erzeugung von Warnungen	$T- $W-
$X +/-	kontrolliert den Ausnahmeüberwachungscode	$X-
$P	fügt einen Vorschubcharakter in das .PRN-File ein	
$L +/-	kontrolliert das Quellcodelisting	$L+

2.2.4 EINGEBAUTE ROUTINEN UND EINSCHLIESSEN VON FILES

Der ATARI PASCAL-Compiler enthält nur die Logik, die für die Definition 'magischer' vordefinierter Prozeduren, Funktionen und Variablen notwendig ist. Das sind Routinen wie z.B. READ, WRITE, ADDR, SIZEOF usw., die inline Codeerzeugung durch den Compiler oder Unterstützung für eine variable Anzahl von Parametern erfordern.

Alle anderen Routinen benutzen ein spezielles Schlüsselwort, 'PREDEFINED', und zwei spezielle Typen, 'ANYTYPE' und 'ANYFILE'. Ihr Quellprogramm muß Deklarationen für diese Routinen enthalten. Das wird normalerweise durch den Gebrauch des $I-Compilertoggles erledigt, um STDPROCS oder ähnliche Files einzubinden. STDPROCS enthält Deklarationen für Prozeduren und Funktionen, die durch den ISO-Standard für PASCAL definiert sind. Zusätzliche Files enthalten Deklarationen für Prozeduren und Funktionen, die über den ISO-Standard hinausgehen, wie z. B. Stringroutinen, ASSIGN, IORESULT usw. Sie können STDPROCS und andere Files so editieren, daß Sie nur noch die für ein gegebenes Programm notwendigen Routinen enthalten.

Diese Methode, eingebaute Routinen zu definieren, ist vorhanden, weil der ATARI 800 Homecomputer für alle Deklarationen und benutzerdefinierten Symbole,die bei der Compilation eines großen Programms auftreten, nur einen begrenzten Speicherraum hat.

2.2.5 FEHLERMELDUNGEN

Die Compilationsfehler sind in der gleichen Reihenfolge und Bedeutung wie in Jensen und Wirths 'User Manual and Report' durchnummeriert. Die Fehlermeldungen, eine kurze Erklärung und mögliche Ursachen sind in Anhang C erklärt.

Z. B.: Error 407, Symbol Table Overflow

Tritt in Phase 1 auf, wenn in der Symboltabelle nicht mehr genug Platz für das zuletzt aufgetretene Symbol ist. Dieser Fehler kann behoben werden, indem man das Programm in mehrere Module aufspaltet.

2.2.6 ZEILENNUMMERN

ATARI PASCAL läßt Zeilennummern zu. Wenn Zeilennummern gewünscht werden, muß die erste Zeile der Programmquelldatei einen numerischen Wert enthalten. Es wird dann angenommen, daß alle Zeilen Nummern enthalten und die Zeilennummern können von beliebiger Länge sein , und es sollte darauf geachtet werden, daß der Compiler sie ignoriert.

2.3 LINKERBEDIENUNG

2.3.1 AUFRUF UND KOMMANDOS

LINK wird benutzt, um den Linker vom Monitor aus aufzurufen. Geben Sie auf das PASCAL-Menü 'L' RETURN und der Linker wird geladen. Der Linker fragt dann den Benutzer nach den, durch Kommas getrennten Namen des Hauptprogramms und der einzubindenden Module.

CALC, FPLIB/S,PASLIB/S

Das Ergebnis wird auf der gleichen Diskette abgelegt wie das Hauptprogramm, falls Sie nicht vor dem Hauptprogrammnamen einen Filenamen gefolgt von einem Gleichheitszeichen spezifizieren, z. B.

D2:CALC=CALC,FPLIB,PASLIB/S (CALC.COM wird auf Laufwerk 2 abgelegt)

Das oben angegebene Kommando wird eines der Demoprogramme mit dem Laufzeitpaket binden. Die zu bindenden Files können von einer Laufwerksnummer angeführt werden.

D2:CALC,D1:FPLIB,D1:PASLIB/S

2.3.2 LINKEROPTIONEN

Der Linker läßt zu, daß Sie eine Anzahl Schalter auf jeden in der Liste auftauchenden Filenamen folgen lassen. Jeder Schalter wird von einem '/' angeführt. Auf die /P und /D Schalter folgt ein Parameter.

2.3.2.1 SUCHE NACH BENÖTIGTEN LAUFZEITBIBLIOTHEKSMODULEN (/S)

Die oben angeführten Beispiele zeigen den Gebrauch des /S-Schalters, der den Linker anweist, die benannte relokatible Datei, PASLIB, abzusuchen und nur die benötigten Module zu entnehmen. Der /S-Schalter entnimmt nur Module den Bibliotheken, nicht aber Funktionen und Prozeduren einzeln compilierter Module. Wie in den Beispielen gezeigt, muß er in der Linkerkommandozeile auf den Namen der Laufzeitbibliothek folgen. PASLIB ist eine speziell konstruierte, durchsuchbare Datei. Andere .ERL- Dateien sind nicht durchsuchbar, falls nicht ausdrücklich darauf hingewiesen wird. (Dies gilt auch für die von Benutzern erzeugten Files.) Die Reihenfolge der Module innerhalb einer Bibliothek ist wichtig.

Jede durchsuchbare Datei muß Routinen in der korrekten Reihenfolge enthalten und mit /S gekennzeichnet sein, um die Suche einzuleiten. Falls sie nicht durch /S gekennzeichnet ist, wird der Inhalt der ganzen Bibliothek geladen.

2.3.2.2 MEMORY MAP (/M)

Ein auf den letzten Namen in der Parameterliste folgendes /M erzeugt ein Memory Map auf dem Schirm.

2.3.2.3 LOAD MAP (/L) UND ERWEITERTES LOAD MAP (/E)

Ein dem letzten Modulnamen folgendes /L weist den Linker an, Modulcode- und Datenadressen anzuzeigen, sobald sie eingebunden werden. Ein dem letzten Modulnamen folgendes /E modifiziert /M und /L dahingehend, daß alle Routinen, auch die mit $, ? und @ beginnenden, für die Laufzeitbibliothek reservierten Namen angezeigt werden.

2.3.2.4 PROGRAMM (/P) UND DATEN (/D) URSPRUNG

Um die Verschiebung von Programm und Daten zu unterstützen, bietet der Linker die /P und /D Schalter. Der /P-Schalter kontrolliert die Adressen des Objektbereichs (ROM) und der D/Schalter die des Datenbereiches (RAM). Die Syntax ist: /P:nnnn oder /D:nnnn mit nnnn als Hexadezimalzahl im Bereich von ØØØØ....FFFF.

Weiterhin wird der Linker nach der Eingabe von /D keine Daten im .COM-File speichern. Das ist eine Möglichkeit, den Datenspeicherbereich für Programme auf der Diskette zu reduzieren, weil nur der Code von der Diskette geladen wird und keine uninitialisierten Datenbereiche. Beachten Sie, daß beim Gebrauch dieser Möglichkeit lokale Fileoperationen nicht gewährleistet werden können, da das System darauf angewiesen ist, daß der Linker den Datenbereich löscht um diese Option zu ermöglichen.

Auch wird bei der Benutzung von /D im Linkprozeß mehr Speicherplatz gewonnen, da beim Binden Daten und Code nicht vermischt werden. Dieser Schalter ist die erste Möglichkeit, einer durch den Linker angezeigten 'out of memory'-Meldung auszuweichen.

Der Gebrauch des /P und /D-Schalters veranlaßt den Linker nicht am Anfang des .COM-Files Platz zu lassen. Die Philosophie des Linkers ist, daß Sie bei Benutzung des /P-Schalters das Programm zur Ausführung auf ein anderes System übertragen wollen. Das bedeutet, daß nach der Eingabe /P:8ØØØ das erste Byte des .COM-Files in der Adresse 8ØØØH liegt ohne 32K Byte Nullen vor diesem ersten Byte. Weiterhin wird der Linker nach der Eingabe von /D keine Daten im .COM-File speichern. Das ist eine Möglichkeit, den Datenspeicherbereich für Programme auf der Diskette zu reduzieren, weil nur der Code von der Diskette geladen wird und keine uninitialisierten Datenbereiche.

Die /P und /D-Schalter werden nach dem letzten Filenamen in beliebiger Reihenfolge angegeben.

2.3.2.5 FORTSETZUNG VON ZEILEN (/C)

Wenn eine Kommandozeile fortgesetzt werden muß, geben Sie nach dem letzten Zeichen der Zeile /C ein, bevor Sie RETURN drücken.

2.3.2.6 LINKER-KOMMANDOFILE (/F)

Der Linker läßt zu, daß Sie Daten in ein File eingeben, die der Linker dann wie Filenamen abarbeitet. Sie geben ein File mit der Erweiterung .CMD an und darauffolgend /F (z. B. CFILES/F). Der Linker wird dieses File lesen und die Namen verarbeiten, als ob sie über die Tastatur eingegeben worden wären. Falls das File mehr als eine Zeile enthält, müssen Sie nach jeder Zeile /C benutzen.

Wenn Sie für weitere Eingaben zur Tastatur zurückkehren wollen, geben Sie /C in der letzten Zeile des .CMD-Files ein. Daten, die nach /F in der Kommandozeile stehen, werden ignoriert. Ein .CMD-File darf in keiner Zeile ein /F enthalten.

2.3.2.7 ZUSAMMENFASSUNG DER LINKEROPTIONEN

/S	Durchsuche die vorher genannte Bibliothek nur nach den benötigten Routinen.
/L	Auflistung der Module, sobald sie gebunden werden.
/M	Auflistung aller Einstiegspunkte in Tabellenform.
/E	Zusätzliche Auflistung der mit $, ? oder @ beginnenden Einstiegspunkte.
/P:nnnn	Verschiebe den Objektcode nach nnnnH.
/D:nnnn	Verschiebe den Datenbereich nach nnnnH.
/F	Entnimm die zu verarbeitenden Filenamen dem vorher genannten .CMD-File.
/C	Fortsetzung von Zeilen.

2.3.2.8 ANFORDERUNGEN RELOKATIBLER FILES

Die Disketten enthalten einige .ERL-Files, die in Ihr Programm eingebunden werden müssen. Welche Files das sind, hängt von der Gruppe der Routinen ab, die der Compiler benötigt und damit von Ihrem Programm. Im Folgenden eine Liste aller Files und der Routinen, die sie enthalten. Falls Sie eine dieser Routinen als undefinierter Verweis gemeldet bekommen, binden Sie das entsprechende relokatible File ein, um diese Meldung zu verhindern.

FPLIB	Reelle Fließkommazahlen @XOP, @RRL, @WRL (durchsuchbar)
PASLIB	Vergleiche, I/O, Arithmetikunterstützung usw.
GRSND	Grafik-, Sound- und Controllerunterstützung

2.3.2.9 LINKERFEHLERMELDUNGEN

Im Linkprozeß auftauchende Fehlermeldungen sind in der Regel selbsterklärend, wie z. B. 'unable to open input file: XXXXXXXX' und 'Duplicate Symbol- XXXXXXX'. Duplicate Symbol bedeutet, daß eine Laufzeitroutine oder -variable und eine Benutzerroutine oder -variable denselben Namen haben. Undefinierte Verweise deuten darauf hin, daß das benötigte relokatible File nicht mit eingebunden wurde. Sehen Sie im vorhergehenden Paragraphen unter Anforderungen relokatibler Files nach.

Wenn Ihnen während des Linkprozesses der Speicherplatz ausgeht, können Sie mit der /D-Option die Daten aus dem Codebereich entfernen . Unter Umständen müssen Sie die /D-Option mit einem sehr hohen Wert anwenden, um die Größe des Codes herauszufinden und darauf einen erneuten Linkerlauf starten, diesmal mit einem Parameter für /D, der wenig über der letzten Codeadresse liegt, so daß Raum für weiteren Code zur Verfügung steht.

2.3.2.1Ø EIGENSCHAFTEN DER LINKFÄHIGEN MODULE

Der Linker kann ATARI PASCAL Hauptprogramme, ATARI PASCAL-Module und Assemblermodule, die mit einem passenden Assembler erstellt worden sind, binden.

2.4 STARTEN DES OBJEKTPROGRAMMS

Sobald das Quellprogramm erfolgreich compiliert und mit den benötigten Laufzeitmodulen gebunden ist, können Sie das Programm starten.

Wenn Sie aus dem Pascalmenü 'R' für Run gewählt haben, werden Sie nach dem Namen des zu startenden Files gefragt werden, z. B.:

    D2:CALC.COM

Das Objektprogramm wird dann in den Speicher geladen und ausgeführt.

2.5 ATARI PROGRAMM-/TEXTEDITOR

Der ATARI Programm-Texteditor ist ein vielseitiges Werkzeug und kann benutzt werden, ATARI PASCAL-Quellprogramme zu erstellen und zu verändern.

2.5.1 START DES ATARI PROGRAMM-/TEXTEDITORS

Das PASCAL-Menü bietet die Option, den ATARI Programm-Texteditor aufzurufen. Ohne weitere Angaben wird dafür Laufwerk 2 angenommen. Bevor Sie diese Option aufrufen, müssen Sie folgende Änderungen vornehmen:

1) Kopieren Sie MEDIT von der gelieferten Diskette auf eine freie Diskette in Laufwerk 2.

2) Laden Sie D2:MEDIT vom DOS Menü mit der Option IN, um den Start zu verhindern (Dies erfordert für den Moment MEM.SAV, das später gelöscht werden kann.).

3) Speichern Sie es vom DOS wie folgt zurück: D2:MEDIT/A,26ØØ,26Ø1.

Diese Append-Operation kann dazu benutzt werden, jedes Assemblerfile von PASCAL aus zu starten. Das File muß mit der Startadresse und der Startadresse+1 angehängt werden. Falls das File aus vielen unzusammenhängenden Modulen besteht, vergewissern Sie sich, daß die angehängte Startadresse dem Laufzeiteinstiegspunkt entspricht.

KAPITEL 3: ERWEITERUNGEN DES ATARI PASCALSPRACHSYSTEMS

Dieses Kapitel beschreibt die Funktion und den Gebrauch der ATARI PASCAL Erweiterungen.

Es deckt die folgenden Gebiete ab:

3.1 Modulare Compilation

3.2 Datenabspeicherung und Parameterübergabe

3.3 Programmsegmentierung und -verkettung

3.4 Eingebaute Prozeduren

3.5 Nichtstandardmäßiger Datenzugriff

3.6 Einbau von Assemblercode

3.7 Grafik- und Sounderweiterung

3.1 MODULARE COMPILATION

ATARI PASCAL unterstützt ein flexibles, modulares Compilationssystem. Programme können solange in einern Stück entwickelt werden, bis sie für Bedienung oder Compilation zu groß werden und dann in mehrere Module zerlegt werden. Das ATARI Compilationssystem erlaubt Zugriff auf jede Prozedur und jede Variable jedes Moduls von jedem anderen Modul aus. Ein Compilerschalter gestattet es, jede gewünschte Gruppe von Prozeduren oder Variablen zu 'verstecken' (d. h. nur lokal zu deklarieren). Lesen Sie in Abschnitt 2.2.3.1 unter Bedienung des $E-Schalters nach.

Die Struktur eines Moduls ist der eines Programms ähnlich. Es beginnt mit dem reservierten Wort 'MODULE', gefolgt von einem Bezeichner und einem Semikolon (z. B. MODULE TEST1;) und endet mit dem reservierten Wort 'MODEND' gefolgt von einem Punkt (z. B. MODEND.) . Dazwischen können Sie, wie in einem Programm, Label, Konstanten, Typen, Variablen, Prozeduren und Funktionen deklarieren. Anders als im Programm gibt es kein BEGIN..END-Abschnitt nach Prozedur- und Funktionsdeklarationen, sondern nur das Wort MODEND, gefolgt von einem Punkt.

z. B.:

```
MODULE MOD1;

<label,const,type,var Deklarationen)

<procedure/function Deklarationen und Blöcke >

MODEND.
```

Um auf Variablen, Prozeduren und Funktionen in anderen Modulen oder im Hauptprogramm zugreifen zu können, wurde ein neues reserviertes Wort, EXTERNAL, eingeführt, das zu folgenden Zwecken gebraucht wird.

Erstens kann das Wort EXTERNAL zwischen Doppelpunkt und Typzuweisung in der GLOBALvariablendeklaration benutzt werden, um anzudeuten, daß dieser Variablenliste eigentlich in diesem Modul kein Speicherplatz zugewiesen wird, sondern in einem anderen Modul. Für in dieser Weise deklarierte Variablen wird kein Speicherplatz eingerichtet.

z. B.:

```
I, J, K, : EXTERNAL INTEGER;(* in einem anderen Modul *)
R:         EXTERNAL RECORD  (* auch in einem anderen
                             Modul *)
                    ......  (* irgendwelche Felder *)
                    END;
```

Sie sind für die passende Typzuweisung verantwortlich, da Compiler und Linker keine Möglichkeit haben die Typverträglichkeit zu überprüfen.

Außerdem wird das Wort EXTERNAL benutzt, um Prozeduren und Funktionen zu deklarieren, die in anderen Modulen existieren. Diese Deklarationen müssen im Modul/Programm vor der ersten normalen Prozedur- oder Funktionsdeklaration erscheinen.

EXTERNAL darf nur auf der globalen (äußersten) Ebene eines Programms oder Moduls verwendet werden.

Genau wie bei Variablendeklarationen fordert ATARI PASCAL von Ihnen, daß Sie sich der Übereinstimmung der Parameter in Typ und Anzahl vergewissern und daß bei Funktionen der zurückgegebene Typ passend ist, da Compiler und Linker keine Möglichkeit haben, die Typverträglichkeit modulübergreifend zu überprüfen. Durch EXTERNAL deklarierte Routinen dürfen keine Funktionen oder Prozeduren als Parameter haben.

Beachten Sie, daß ATARI PASCAL die Signifikanz der externen Namen von acht auf sieben Zeichen beschränkt. Begrenzen Sie die Zugriffsmöglichkeit eventuell verwendeter Assemblerprogramme auf sechs Zeichen.

Im Folgenden sind ein Programm- und Modulskelett aufgelistet. Das Hauptprogramm bezieht sich auf Variable und Unterprogramme im Modul, und das Modul bezieht sich auf Variable und Unterprogramme im Hauptprogramm. Die einzelnen Unterschiede zwischen einem Hauptprogramm und einem Modul sind die 16 Bytes des Headercodes und der auf Prozeduren und Funktionen folgende Programmblock.

Beispiel für ein Hauptprogramm

```
PROGRAM EXTERNAL_DEMO;

<label,const,type declarations>

VAR

    I,J INTEGER ;     (* durch andere Module erreichbar *)

    K,L EXTERNAL INTEGER;   (* woanders lokalisiert *)

EXTERNAL PROCEDURE SORT (VAR Q:LIST;LEN:INTEGER);

EXTERNAL FUNCTION IOTEST: INTEGER;

PROCEDURE PROC1;
BEGIN
   IF IOTEST = 1 THEN
      (* normalerweise Aufruf einer externen Funktion *)
   .....
END;

BEGIN
   SORT(....) ;
      (* normalerweise Aufruf einer externen Prozedur *)
END.
```

Beispiel für ein Modul (Beachten Sie, daß dies separate Files sind)

```
MODULE MODULE_DEMO;

<label,const,type declarations>

VAR

  I,J:EXTERNAL INTEGER; (* Zugriff auf das Hauptprogramm *)

  K,L:INTEGER;          (* hier im Modul definiert *)

EXTERNAL PROCEDURE PROC1;(* Zugriff auf das Hauptprogramm *)

PROCEDURE SORT ( ... );  (* hier im Modul definiert)
    ......

FUNCTION IOTEST:INTEGER; (* hier im Modul definiert *)

(andere Prozeduren und Funktionen)

MODEND.
```

3.2 DATENABSPEICHERUNG UND PARAMETERÜBERGABE

3.2.1 DATENABSPEICHERUNG

Neben dem Aufruf der Variablen durch ihren Namen müssen Sie wissen, wie die Variablen im Speicher abgelegt werden. Abschnitt 5.1 behandelt Speicherzuweisung und Format jedes skalaren, eingebauten Datentyps. Variable, die hauptsächlich im GLOBAL Datenbereich abgelegt werden, werden hier behandelt. Variable in einer Bezeichnerliste, denen ein Typ folgt (z. B. A,B,C : INTEGER) , sind in umgekehrter Reihenfolge angeordnet (z. B. C zuerst, gefolgt von B und A).

Beispiel:

```
A     : INTEGER;
B     : CHAR;
I,J,K : BYTE;
L     : INTEGER;
```

Speichereinteilung:

```
  +Ø A LSB (last significant byte: niedrigstwertiges Byte)
  +1 A MSB (most significant byte: höchstwertiges Byte)
  +2 B
  +3 K
  +4 J
  +5 I
  +6 L LSB
  +7 L MSB
```

Strukturierte Datentypen, ARRAYs, RECORDs und SETs, bedürfen einer weiteren Erklärung. ARRAYs sind zeilenweise abgespeichert. Z. B. wird A:ARRAY [1..3,1..3] OF CHAR folgendermaßen abgespeichert:

```
  +Ø A [1,1]
  +1 A [1,2]
  +2 A [1,3]

  +3 A [2,1]
  +4 A [2,2]
  +5 A [2,3]

  +6 A [3,1]
  +7 A [3,2]
  +8 A [3,3]
```

Logisch gesehen ist dies ein eindimensionales Array von Vektoren. In ATARI PASCAL sind alle Arrays logisch eindimensionale Arrays irgendeines Typs.

RECORDs werden in der gleichen Weise wie globale Variablen abgespeichert.

SETs werden immer als 32-byte-Ausdruck gespeichert. Jedes Element des SETs ist als Bit gespeichert. SETs sind

byteorientiert und das niederwertige Bit jedes Bytes ist das erste Bit in dem Byte dieses SETs. Im Folgenden ist das SET 'A'..'Z':

Bytenummer:

```
ØØ Ø1 Ø2 Ø3 Ø4 Ø5 Ø6 Ø7 Ø8 Ø9 ØA ØB ØC ØD ØE ØF 1Ø ... 1F
-- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- ... --
ØØ ØØ ØØ ØØ ØØ ØØ ØØ ØØ FE FF FF Ø7 ØØ ØØ ØØ ØØ ØØ ... ØØ
```

Das erste Bit ist Bit 64 ($41) und liegt in Byte 8, Bit 1.
Das letzte Bit ist 9Ø und liegt in Byte 11, Bit 2.
In diesem Fall stellt Bit Ø das niedrigstwertige Bit im Byte dar.

3.2.2 PARAMETERÜBERGABE

Beim Aufruf einer Assemblerroutine vom ATARI PASCAL aus, oder beim Aufruf einer ATARI PASCAL-Routine durch eine Assemblerroutine werden Parameter auf dem Stack übergeben. Der Parameterübergabestack unterscheidet sich vom 65Ø2 Hardwarestack. Dieser Softwarestack befindet sich in den Adressen $6ØØ bis $6FF. Das Hardwareregister X muß während der Ausführung einer Assemblerroutine gerettet und aktualisiert werden und dient als Pointer auf den Softwarestack. Sie können den Beginn des Stacks durch LDA $6ØØ,X usw. laden. Beim Eintritt in die Routine enthält der Beginn des Hardwarestacks die Rückkehradresse. Auf dem Softwarestack liegt in umgekehrter Reihenfolge die Deklaration (A, B :INTEGER;C :CHAR), also C über B und B über A. Jeder Parameter erfordert mindestens einen 16-bit-Wort Stackplatz. Ein Charakter oder eine boolsche Variable wird als 16-bit-Wort übergeben, dessen höherwertiges Byte ØØ ist. VAR-Parameter werden über ihre Adresse übergeben. Die Adresse zeigt auf das Byte der Variablen mit der niedrigsten Speicheradresse.

Nichtskalare Parameter (außer SETs) werden immer per Adresse übergeben. Falls es sich bei dem Parameter um einen aktuellen Wert handelt, wird der Code, um das Datum zu übertragen, durch den Compiler in einer PASCAL-Routine erzeugt. SET-Parameter werden mit ihrem Wert auf den Stack übergeben und der Interpreter wird benutzt um sie abzuspeichern.

Das folgende Beispiel zeigt eine typische Parameterliste beim Eintritt in eine Prozedur:

```
PROCEDURE DEMO (I,J:INTEGER;VAR Q:STRING;C,D:CHAR);
```

Stackbelegung beim Eintritt ($6ØØ,X):

+Ø D
+1 Byte ØØ
+2 C
+3 Byte ØØ
+4 Adresse des aktuellen Strings
+5 Adresse des aktuellen Strings
+6 J (LSB)
+7 J (MSB)
+8 I (LSB)
+9 I (MSB)

Vor der Rückkehr zur aufrufenden Routine muß die Assemblerroutine alle Parameter vom Softwarestack entfernen.

SETs werden auf dem Stack mit dem niedrigstwertigen Byte zuunterst (hohe Adresse) gespeichert.

Funktionswerte werden auf dem Stack zurückgegeben. Sie werden vor der Rückkehr logisch unterhalb der Rückkehradresse gespeichert. Daher verbleiben sie auf dem Stack, nachdem in das aufrufende Programm wieder eingetreten worden ist. Assemblerroutinen dürfen nur die Skalartypen INTEGER, REAL, BOOLEAN und CHAR zurückgeben.

3.3 PROGRAMMAUFTEILUNG UND -VERKETTUNG

Von Zeit zu Zeit überschreiten Programme den zur Verfügung stehenden Speicherplatz und genau so oft ist es wünschenswert, Programme für Compilation und Wartung aufzuteilen.

ATARI PASCAL stellt einen Verkettungsmechanismus zur Verfügung, durch den Programme die Kontrolle an andere Programme übergeben können.

Sie müssen ein typfreies File deklarieren (FILE;) und die ASSIGN und RESET-Prozeduren zur Initialisierung anwenden. Sie können dann die CHAIN-Prozedur aufrufen und die Filevariable als einzelnen Parameter übergeben. Das Laufzeitbibliotheksprogramm wird dann die benötigten Funktionen zum Laden des Files durch Benutzung der RESET-Funktion durchführen. Die Programmgröße spielt keine Rolle. Ein kleines Programm können Sie an ein großes hängen und umgekehrt. Wenn Sie eine Verbindung zwischen den verketteten Programmen wünschen, können Sie zwischen zwei Wegen wählen: gemeinsam benutzte globale Variable oder Variable vom Typ ABSOLUTE.

Wenn Sie die erste Methode benutzen, müssen Sie sicherstellen, daß zumindest die erste Sektion der globalen Variablen in den beiden Programmen, die miteinander in Verbindung stehen sollen, übereinstimmt. Die übrigen globalen Variablen müssen nicht die gleichen sein und die Deklaration externer Variablen im globalen Abschnitt hat keinen Einfluß auf diese Speicherverteilung. Zusammen mit den passenden Deklarationen müssen Sie die /D-Option (siehe 2.3.2.4) des Linkers benutzen, um die Variablen aller Programme, die miteinander kommunizieren sollen, an den gleichen Stellen zu plazieren.

Um die zweite Methode zu benutzen, definieren Sie in der Regel einen Record, der als Kommunikationsbereich dient, und legen diesen Record in jedem Modul an absoluter Adresse ab. Diese Methode erfordert nicht die Benutzung der /D-Option des Linkers, aber einen Überblick über die Speicherbenutzung des Programms und des Systems.

Im Folgenden finden Sie zwei Beispielprogramme, die über die Benutzung absoluter Variablen miteinander kommunizieren. Das erste Programm verkettet sich mit dem zweiten, das wiederum die Ergebnisse des ersten Programms ausdruckt.

Beispiel:

```
PROGRAM PROG1;

TYPE
 COMMAREA = RECORD
             I,J,K : INTEGER
            END;

VAR
  GLOBALS : ABOLUTE [$8ØØØ] COMMAREA;
  CHAINFIL: FILE;

BEGIN  (* Hauptprogramm*)
  WITH GLOBALS DO
    BEGIN
      I:=3;
      J:=3;
      K:=I*J;
    END;

  ASSIGN (CHAINFIL,'D1:PROG2.COM');
  RESET (CHAINFIL);
  IF IORESULT <> Ø THEN
    BEGIN
      WRITELN ('UNABLE TO OPEN D1:PROG2.COM');
      EXIT
    END;

  CHAIN (CHAINFIL)
END. (* Ende von Prog 1 *)

(* Programm #2 für Verkettungsdemo *)

PROGAM PROG2;

TYPE
  COMMAREA = RECORD
               I,J,K : INTEGER
             END;

VAR
  GLOBALS : ABSOLUTE [$8ØØØ] COMMAREA;

BEGIN (* Start Programm #2 *)
  WITH GLOBALS DO
    WRITELN ('RESULT OF ',I,' TIMES',J,' IS =', K)
END. (* Nach Abschluß Rückkehr zum Betriebssystem *)
```

3.4 EINGEBAUTE PROZEDUREN UND PARAMETER

Dieser Abschnitt beschreibt eingebaute Prozeduren und Parameter. Syntax und Parameter jeder Routine werden beschrieben und ein Beispielprogramm zeigt den Gebrauch der Funktion. Abschnitt 3.4.23 gibt einen schnellen Überblick über alle eingebauten Prozeduren und Funktionen.

3.4.1 MOVE, MOVERIGHT, MOVELEFT

```
PROCEDURE MOVE      (SOURCE,DESTINATION,NUM BYTES)
PROCEDURE MOVELEFT  (SOURCE,DESTINATION,NUM-BYTES)
PROCEDURE MOVERIGHT (SOURCE,DESTINATION,NUM=BYTES)
```

Diese Prozeduren bewegen die in NUM BYTES Bytes spezifizierte Anzahl von SOURCE nach DESTINATION.MOVE ist ein Synonym für MOVELEFT. MOVELEFT bewegt vom linken Ende der SOURCE zum linken Ende der DESTINATION.

MOVERIGHT bewegt vom rechten Ende der SOURCE zum rechten Ende der DESTINATION(die Parameter, die an MOVERIGHT übergeben werden, spezifizieren das linke Ende von SOURCE und DESTINATION).

Sie können MOVELEFT und MOVERIGHT benutzen, um Bytes von einer Datenstruktur zu einer anderen zu transportieren oder um Daten in derselben Datenstruktur im Kreis zu bewegen. Die Bewegung erfolgt auf der Byte-Ebene, so daß der Datentyp unbeachtet bleibt. MOVERIGHT ist sinnvoll , um Daten von niedrigen Indices eines Arrays zu hohen Indices zu bewegen. Ohne diese Prozedur wäre eine FOR-Schleife von Nöten, die jedes Datum aufnehmen und an höheren Adressen wieder ablegen müßte. MOVERIGHT ist also sehr viel schneller. MOVERIGHT ist ideal für eine Zeicheneinsetzroutine, die Raum für Zeichen in einem Buffer schaffen soll.

MOVELEFT ist sinnvoll, um Bytes von einem Array zu einem anderen zu schaffen, Zeichen aus einem Buffer zu entfernen oder Daten von einer Struktur in eine andere zu bringen.

SOURCE und DESTINATION können von beliebigem Variablentyp sein und brauchen nicht vom gleichen Typ zu sein. Sie dürfen auch Zeiger auf Variable oder als Zeiger benutzte Integers sein, aber keine benannten oder Literalkonstanten. Die Byteanzahl muß ein Integerausdruck größer oder gleich null sein.

Beachten Sie folgende Probleme:

1. Da keine Überprüfung daraufhin erfolgt, ob die Anzahl der Bytes größer ist als die DESTINATION, wird ein Überlauf in die an DESTINATION angrenzenden Speicherbereiche stattfinden, falls DESTINATION nicht groß genug ist, um die übertragenen Bytes aufzunehmen.

2. Die Bewegung von null Bytes hat keine Auswirkung.

3. Eine Typüberprüfung wird nicht vorgenommen.

Beispiel:

```
PROCEDURE
CONST
  STRINGSZ = 8Ø;
VAR
  BUFFER : STRING[STRINGSZ];
  LINE : STRING;

PROCEDURE INSRT(VAR DEST:STRING;INDEX:INTEGER;VAR SOURCE:
                STRING);
BEGIN
  IF LENGTH(SOURCE) <= ( = STRINGSZ - LENGTH(DEST) THEN
    BEGIN
      MOVERIGHT(DEST[INDEX],DEST[INDEX+LENGTH (SOURCE)],
                LENGTH(DEST)-INDEX+1);
      MOVELEFT(SOURCE[1],DEST[INDEX],LENGTH(SOURCE));
               DEST(Ø):=CHAR(ORD(DEST[Ø]+LENGTH(SOURCE))
    END;
END;

BEGIN
  WRITELN('MOVE_DEMO......');
  BUFFER:='Judy J. Smith/ 335 Drive/ Lovely, Ca. 95666';
  WRITELN(BUFFER);
  LINE:='Roland' ;
  INSRT(BUFFER,POS('5',BUFFER)+2,LINE);
  WRITELN(BUFFER);
END;
```

Ausgabe dieses Beispielprogramms:

```
MOVE DEMO......
Judy J. Smith/ 335 Drive/ Lovely, Ca. 95666
Judy J. Smith/ 335 Roland Drive / Lovely, Ca. 95666
```

3.4.2 EXIT

PROCEDURE EXIT

EXIT ist ein Äquivalent zu RETURN in FORTRAN oder BASIC. Es führt zum Verlassen der laufenden Prozedur/Funktion oder des Hauptprogramms. Wenn EXIT in einer INTERRUPT-Prozedur verwendet wird, werden die Register wieder hergestellt und weitere Interrupts zugelassen, bevor die Prozedur verlassen wird. EXIT folgt als Statement normalerweise auf einen Vergleich.

Beispiel:

```
PROCEDURE EXITTEST;
 (* Verlassen der laufenden Funktion oder des Hauptpro-
    gramms. *)

  PROCEDURE EXITPROC (BOOL:BOOLEAN);

  BEGIN
    IF BOOL THEN
      BEGIN
        WRITELN('EXITING EXITPROC');
        EXIT;
      END;
    WRITELN('STILL IN EXITPROC, ABOUT TO LEAVE NORMALLY');
  END;

BEGIN
  WRITELN(EXITTEST.......');
  EXITPROC(TRUE);
  WRITELN('IN EXITTEST AFTER 1ST CALL TO EXITPROC');
  EXITPROC(FALSE);
  WRITELN('IN EXITTEST AFTER 2ND CALL TO EXITPROC');
  EXIT;
  WRITELN('THIS LINE WILL NEVER BE PRINTED');
END;
```

Ausgabe:

```
EXITTEST.......
EXITING EXITPROC
IN EXITTEST AFTER 1ST CALL TO EXITPROC
STILL IN EXITPROC, ABOUT TO LEAVE NORMALLY
IN ECITTEST AFTER 2ND CALL TO EXITPROC
```

3.4.3 TSTBIT, SETBIT, CLRBIT

```
FUNCTION TSTBIT (BASIC_VAR,BIT NUM) : BOOLEAN;
PROCEDURE SETBIT (VAR BASIC_VAR,BIT NUM);
PROCEDURE CLRBIT (VAR BASIC_VAR,BIT=NUM);
```

TSTBIT ergibt TRUE, wenn das durch BIT NUM spezifizierte Bit in BASIC VAR 1 ist, und ergibt FALSE, wenn das Bit Ø ist. SETBIT setzt das spezifizierte Bit im Parameter. CLRBIT löscht das spezifizierte Bit im Parameter.

BASIC VAR ist eine beliebige 8 oder 16-bit Variable, wie z. B. integer, char, byte, word oder boolean. BIT_NUM kann Werte von Ø bis 15 annehmen und zählt von rechts an. Der Versuch, Bit 1Ø einer 8-bit Variablen zu setzen, verursacht keinen Fehler, hat aber auch keinen Einfluß auf das Ergebnis.

Diese Prozeduren sind nützlich, um Warteschleifen zu erzeugen, oder um eingehende Daten durch Umschalten von Bits je nach Bedarf zu verändern. Eine andere Anwendung ist die Manipulation eines bitorientierten Bildschirms.

Beispiel:

```
PROCEDURE TST_SET_CLR_BITS;

VAR
  I : INTEGER;
BEGIN
  WRITELN('TST_SET_CLR_BITS.......');
  I := Ø;
  SETBIT(I,5);
  IF I = 32 THEN
    IF TSTBIT(I,5) THEN
      WRITELN('I=',I);
  CLRBIT(I,5);
  IF I = Ø THEN
    IF NOT (TSTBIT(I,5)) THEN
      WRITELN('=',I);
END;
```

Ausgabe:

```
TST_SET_CLR_BITS .......
I=32
I=G
```

3.4.4 SHR, SHL

```
FUNCTION SHR(BASIC_VAR,NUM):INTEGER;
FUNCTION SHL(BASIC_VAR,NUM):INTEGER;
```

SHR schiebt BASIC_VAR um NUM Bits nach rechts und setzt dafür Ø ein. SHL schiebt BASIC_VAR um NUM Bits nach links und setzt dafür Ø ein. BASIC_VAR ist eine 8 oder 16-bit Variable, NUM ein Integerausdruck.

Der Einsatz von SHR und SHL ist offensichtlich. Angenommen, ein 1Ø-bit Wert ist von zwei verschiedenen Ports zu erhalten. Sie können die Werte mit SHL einlesen.

```
VAR
  PORT1 : ABSOLUTE [$DØØØ] BYTE;
  PORT2 : ABSOLUTE [$D232] BYTE;

X := SHL(PORT1 & $1F,3) ! (PORT2 & $1F);
```

Das Beispielprogramm liest port1, blendet aus dem erhaltenen Byte die drei hohen Bits aus und schiebt das Ergebnis nach rechts. Dann wird das Ergebnis mit der Eingabe von port2 ODER-verknüpft, die auch maskiert wurde.

Das folgende Beispiel zeigt das erwartete Resultat der Ausführung der beiden Funktionen.

Beispiel:

```
PROCEDURE SHIFT_DEMO;
VAR I : INTEGER;
BEGIN
  WRITELN('SHIFT DEMO ....... ');
  I := 4;
  WRITELN('I=',I);
  WRITELN('SHR(I,2)=',SHR(I,2));
  WRITELN('SHL(I,4)=',SHL(I,4));
END;
```

Ausgabe:

```
SHIFT DEMO .......
I=4
SHR(I,2)=1
SHL(I,4)=64
```

3.4.5 HI, LO, SWAP

```
FUNCTION   HI(BASIC_VAR) : INTEGER;
FUNCTION   LO(BASIC_VAR) : INTEGER;
FUNCTION SWAP(BASIC_VAR) : INTEGER;
```

HI ergibt die 8 hohen Bits von BASIC_VAR (8 oder 16-bit Variable) in 8 niedrigen Bits des Ergebnisses. LO ergibt die 8 niedrigen Bits, die 8 hohen Bits werden auf Ø gesetzt. SWAP ergibt die 8 hohen Bits von BASIC_VAR in den 8 niedrigen Bits des Ergebnisses und die 8 niedrigen Bits von BASIC_VAR in den 8 hohen Bits des Ergebnisses. Die Übergabe einer 8-bit Variablen ergibt Ø und die Übergabe von 8 Bits an LO verändert nichts.

Diese Funktionen steigern ATARI PASCALs Möglichkeiten, die I/O-Ports zu lesen oder zu beschreiben. Wenn ein 16-bit Datenwort auf einen Port geschickt wird, der nur 8 Bit zur Zeit verarbeiten kann, können Sie LO und HI benutzen, um das niedrige Byte, gefolgt vom hohen zu senden. Auf ähnliche Weise kann ein 16-bit Wort von einem 8-bit Port gelesen werden, indem man die ersten 8 Bit in die höheren 8 Bit 'swapped':

```
    VAR
      PORT6 : ABSOLUTE [$D234] BYTE;

  PORT6 := LO(B);
  PORT6 := HI(B);
  B := SWAP(PORT6) ! PORT6;
```

Das folgende Beispiel zeigt die zu erwartenden Resultate dieser Funktionen:

Beispiel:

```
PROCEDURE HI_LO_SWAP;
VAR
  HL : INTEGER;
BEGIN
  WRITELN('HI_LO_SWAP.......');
  HL := $1Ø4;
  WRITELN('HL=',HL);
  IF HI(HL) = 1 THEN
    WRITELN('HI(HL)=',HI(HL));
  IF LO(HL) = 4 THEN
    WRITELN('LO(HL)=',LO(HL));
  IF SWAP(HL) = $Ø4Ø1 THEN
    WRITELN('SWAP(HL)=',SWAP(HL));
END;
```

Ausgabe:

```
HI_LO_SWAP.......
HL=26Ø
HI(HL)=1
LO(HL)=4
SWAP(HL)=1Ø25
```

3.4.6 ADDR

FUNCTION ADDR(VARIABLE REFERENCE) : INTEGER;

ADDR ergibt die Adresse der genannten Variablen. Als Argumente der Funktion sind Prozedur/Funktionsnamen, indizierte Variable und Recordfelder zugelassen. Nicht zugelassen sind Konstante, benutzerdefinierte Typen oder andere Daten, die weder Code- noch Datenplatz belegen.

Diese Funktion wird benutzt, um bestimmte Adressen aufzufinden:durch INLINE zur Compilierzeit erzeugte Tabellen, die Adressen von Datenstrukturen, die für MOVE-Operationen benötigt werden usw.

Beispiel:

```
PROCEDURE ADDR_DEMO(PARAM : INTEGER);
VAR
  REC : RECORD
          J : INTEGER;
          BOOL : BOOLEAN;
      END;
  ADDRESS : INTEGER;
  R : REAL;
  S1 : ARRAY[1..1Ø] OF CHAR;

BEGIN
  WRITELN('ADDR_DEMO.....');
  WRITELN('ADDR(ADDR_DEMO)=',ADDR(ADDR_DEMO));
  WRITELN('ADDR(PARAM)=',ADDR(PARAM));
  WRITELN('ADDR(REC)=',ADDR(REC));
  WRITELN('ADDR(REC.J)=',ADDR(REC.J));
  WRITELN('ADDR(ADDRESS)=',ADDR(ADDRESS));
  WRITELN('ADDR(R)=',ADDR(R));
  WRITELN('ADDR(S1)=',ADDR(S1));
END;
```

Die Ausgabe hängt vom System ab.

3.4.7 SIZEOF

FUNCTION SIZEOF(Variablen- oder Typname) : INTEGER;

SIZEOF gibt die Größe des genannten Parameters in Byte an. Es wird in MOVE-Routinen benutzt, um die Anzahl der zu verschiebenden Bytes zu erhalten . Durch SIZEOF brauchen Sie während der Programmentwicklung keine Konstanten verändern. Jede Variable ist als Parameter zugelassen: Charakter, Arrays, Records usw. oder jeder benutzerdefinierte Typ.

Beispiel:

```
PROCEDURE SIZE_DEMO;

VAR
  B : ARRAY[1..1Ø] OF CHAR;
  A : ARRAY[1..15] OF CHAR;
BEGIN
  WRITELN('SIZE_DEMO.......');
  A:='***************';
  B:='Ø123456789';
  WRITELN('SIZEOF(A)=',SIZEOF(A),' SIZEOF(B)=',SIZEOF(B));
  MOVE(B,A,SIZEOF(B));
  WRITELN('A= ',A);
END;
```

Ausgabe:

```
SIZEOF(A)=15 SIZEOF(B)=1Ø
A= Ø123456789*****
```

3.4.8 FILLCHAR

PROCEDURE FILLCHAR (DESTINATION ,LENGTH,CHARACTER)

Diese Prozedur füllt DESTINATION(ein packed Array of char) mit der Anzahl CHARACTER, die durch LENGTH spezifiziert wird. DESTINATION ist ein packed Array of Charakters. Es darf indiziert sein. LENGTH ist ein Integerausdruck. Falls LENGTH größer als DESTINATION ist, werden angrenzende Daten- oder Codebereiche überschrieben. Auch bei einem negativen Wert können angrenzende Speicherbereiche überschrieben werden. CHARACTER ist eine Konstante oder Variable vom Typ char.

Der Sinn von FILLCHAR ist, eine schnelle Methode zum Füllen großer Datenstrukturen mit dem gleichen Datum zu bieten. Z. B. kann das Löschen von Puffern mit FILLCHAR erreicht werden.

Beispiel:

```
PROCEDURE FILL_DEMO;
VAR
  BUFFER : PACKED ARRAY [1..256] OF CHAR;
BEGIN
  FILLCHAR(BUFFER,256,''); (* Löschen der Buffer *)
END;
```

3.4.9 LENGTH

```
FUNCTION LENGTH(5TRING) : INTEGER;
```

Diese Funktion ergibt die Länge eines 5tring als Integer-wert.

Beispiel:

```
PROCEDURE LENGTH_DEMO;
VAR
  S1 : 5TRING [4Ø];
BEGIN
  S1 := 'This String is 33 characters long';
  WRITELN('LENGTH OF',S1,'=',LENGTH(S1));
  WRITELN(LENGTH OF EMPTY STRING=',LENGTH(''));
END;
```

Ausgabe:

```
LENGTH OF This String is 33 characters long=33
LENGTH OF EMPTY STRING = Ø
```

3.4.1Ø CONCAT

```
FUNCTION CONCAT(SOURCE1,SOURCE2,...,SOURCEn) : STRING
```

Diese Funktion ergibt einen String, in dem alle in der Parameterliste angegebenen Quellen verknüpft sind. Bei den Quellen kann es sich um Stringvariablen, -literale oder um Character handeln. Eine Quelle der Länge Ø kann ohne Probleme verkettet werden. Falls die Gesamtlänge aller Quellen 256 überschreitet, wird der String nach 256 Bytes abgeschnitten. Sehen Sie im nächsten 5chritt unter Copy nach, der die Einschränkungen bei der Verwendung von CONCAT und COPY behandelt.

Beispiel:

```
PROCEDURE CONCAT_DEMO;
VAR
  S1,S2 : STRING;
BEGIN
  S1:=' left link, right link';
  S2:=' root root root';
  WRITELN(S1,'/',S2);
  S1 := CONCAT (S1,' ',S2,'!!!!!!');
  WRITELN(S1);
END;
```

Ausgabe:

```
left link, right link/root root root
left link, right link root root root !!!!!!
```

3.4.11 COPY

FUNCTION COPY(SOURCE,LOCATION,NUM_BYTE) : STRING;

COPY entnimmt aus dem String SOURCE einen Teilstring, der bei LOCATION beginnt und NUM_BYTE lang ist. SOURCE muß ein String sein. LOCATION und NUM_BYTE sind Integerausdrücke. Falls LOCATION außerhalb der Grenzen oder negativ ist, entsteht kein Fehler. Falls NUM_BYTE negativ oder NUM_BYTE plus LOCATION länger als SOURCE ist, wird der String abgeschnitten.

Beispiel:

```
PROCEDURE COPY_DEMO;
BEGIN
  LONG_STR := 'Hi from Cardiff-by-the-Sea';
  WRITELN(COPY(LONG_STR,9,LENGTH(LONG_STR)-9+1));
END;
```

Ausgabe:

```
Cardiff-by-the-Sea
```

Bemerkung:

COPY und CONCAT sind Funktionen, die Pseudostrings ergeben. Sie haben nur einen statisch zugewiesenen Buffer für das Ergebnis. Deshalb wird, wenn diese Funktionen mehr als einmal in einem Ausdruck aufgerufen werden, dem Wert jedes Aufrufs dieser Funktion der Wert des letzten Aufrufs zugewiesen. Zum Beispiel ergibt

'IF(CONCAT(A,STRING1))=(CONCAT(A,STRING2))'

immer den Wert 'true', da das Ergebnis der Operation CONCAT(A,STRING1) durch das Ergebnis der Operation CONCAT(A,STRING2) ersetzt wird.
'WRITELN(COPY(STRING1,1,4),COPY(STRING1,5,4)) schreibt zwei gleiche, vier Zeichen lange Teilstrings aus STRING1.

3.4.12 POS

FUNCTION POS(PATTERN,SOURCE) : INTEGER;

Diese Funktion ergibt einen Integerwert für das erste Auftauchen von PATTERN in SOURCE. Falls PATTERN nicht gefunden wird, ist das Ergebnis Ø. SOURCE ist ein String und PATTERN kann String, Character oder Literal sein.

Beispiel:

```
PROCEDURE POS_DEMO;
VAR
  STR,PATTERN : STRING;
  CH : CHAR;
BEGIN
  STR := 'ABCDEFGHIJKLMNO';
  PATTERN := 'FGHIJ';
  CH := 'B';
  WRITELN('pos of ',PATTERN,' in ',STR,' is ',
          POS(PATTERN,STR));
  WRITELN('pos of ',CH,' in ',STR,' is ',POS(CH,STR));
  WRITELN('pos of ''z'' in ',STR,' is ',POS('z',STR));
END;
```

Ausgabe:

```
pos of FGHIJ in ABCDEFGHIJKLMNO is 6
pos of B in ABCDEFGHIJKLMNO is 2
pos of 'z' in ABCDEFGHIJKLMNO is Ø
```

3.4.13 DELETE

PROCEDURE DELETE(TARGET,INDEX,SIZE);

Diese Prozedur wird benutzt, um SIZE Characters aus TARGET zu entfernen, beginnend bei dem durch INDEX bestimmten Byte. TARGET ist ein String, INDEX und SIZE sind Integer-Ausdrücke. Auf SIZE mit dem Wert Ø folgt keinerlei Aktion. Ein negativer Wert hat ernste Fehler zur Folge. Falls INDEX plus SIZE größer als TARGET oder TARGET leer ist, können die Daten und umliegender Speicherraum zerstört werden.

Beispiel:

```
PROCEDURE DELETE_DEMO;
VAR
  LONG_STR : STRING;
BEGIN
  LONG_STR := '      get rid of the leading blanks';
  WRITELN(LONG_STR);
  DELETE(LONG_STR,1,POS('g',LONG_STR)-1);
  WRITELN(LONG_STR);
END;
```

Ausgabe:

```
      get rid of the leading blanks
get rid of the leading blanks
```

3.4.14 INSERT

PROCEDURE INSERT (SOURCE,DESTINATION,INDEX);

Diese Prozedur wird benutzt, um SOURCE an der Stelle in DESTINATION einzusetzen, die durch INDEX angegeben wird. DESTINATION ist ein String. SOURCE kann vom Typ Character oder String oder ein Literal oder eine Variable sein. INDEX ist ein Integer-Ausdruck. SOURCE kann leer sein. Falls INDEX außerhalb der Grenzen liegt oder DESTINATION leer ist, werden die Daten zerstört. Falls das Einsetzten von SOURCE in DESTINATION verursacht, daß DESTINATION länger als erlaubt wird, wird DESTINATION abgeschnitten.

Beispiel:

```
PROCEDURE INSERT_DEMO;
VAR
  LONG_STR : STRING;
  S1 : STRING [1Ø];
BEGIN
  LONG_STR := 'Remember May 9';
  S1 := 'Mother's Day,';
  INSERT(S1,LONG_STR,1Ø);
  WRITELN(LONG_STR);
  INSERT('to celebrate',LONG_STR,1Ø);
  WRITELN (LONG_STR);
END;
```

Ausgabe:

```
Remember Mother's Day, May 9
Remember to celebrate Mother's Day, May 9
```

3.4.15 ASSIGN

PROCEDURE ASSIGN(FILE,NAME);

Benutzen Sie diese Prozedur, um vor einem RESET oder REWRITE-Befehl einer Filevariablen den Namen eines externen Files zuzuordnen. FILE ist der Filename, NAME ist ein Literal- oder variabler String, der den Namen des zu erzeugenden Files enthält. FILE muß vom Typ TEXT sein, um die unten aufgeführten speziellen Gerätenamen benutzen zu können.

Beachten Sie, daß Standard PASCAL lokale Files definiert. ATARI PASCAL implementiert diese Möglichkeit, indem es zeitweilige Filenamen in der Form PASTMP xx benutzt. 'xx' startet mit dem Wert Ø zu Beginn jedes Programms und wird dann fortlaufend zugewiesen. Falls ein REWRITE für ein externes File nicht von einem ASSIGN eingeleitet wurde, wird ihm trotzdem vor seiner Erzeugung ein vorübergehender Filename zugeordnet.

NAME ist meistens ein Diskettenfilename im Format Dn:Filename.ext, kann aber auch ein spezieller Gerätename sein.

Gerätenamen:

E: Screeneditor (Ein/Ausgabe)
S: Screenmonitor (Ausgabe)
K: Tastatur (Eingabe)
P: Drucker (Ausgabe)

Bemerkung: Cassettenfiles (C:) werden von ATARI PASCAL nicht unterstützt.

Beispiele für ASSIGN-Anwendungen:

```
ASSIGN(PRINTFILE,'P:');
ASSIGN(F,'D2:MY28Ø.ØVL');
ASSIGN(KEYBOARD,'K:');
ASSIGN(CRT,'S:');
```

Bemerkung: Nach ASSIGN(CRT,'S:') müssen Sie REWRITE benutzen, da das ASSIGN-Kommando das File nicht öffnet.

3.4.16 WNB,GNB

```
FUNCTION GNB(FILEVAR:FILE OF PAOC) : CHAR;
FUNCTION WNB(FILEVAR:FILE OF CHAR;CH:CHAR) : BOOLEAN;
```

Diese Funktionen gestatten Ihnen, mit hoher Geschwindigkeit auf einzelne Bytes eines Files zuzugreifen. PAOC ist jeder Typ, der ein packed Array of Char ist. Die Größe des Arrays ist im Bereich 128..4Ø95 optimal.

GNB läßt Sie ein File byteweise lesen und liefert ein Ergebnis vom Typ Char. EOF nimmt den Wert 'true' an, sobald das physikalische Fileende erreicht ist, hat aber keinen Bezug zu den Daten im File.

WNB läßt Sie byteweise in ein File schreiben. Es erfordert ein File und ein Zeichen, das geschrieben werden soll. Die Funktion hat ein boolsches Ergebnis, das den Wert 'true' annimmt, sobald während des Schreibens auf das File ein Fehler auftritt. Die Bytes, die geschrieben werden, unterliegen keinerlei Interpretation.

GNB und WNB werden im Gegensatz zu F^, GET/PUT Kombinationen benutzt, da sie wesentlich schneller sind.

3.4.17 BLOCKREAD, BLOCKWRITE

```
BLOCKREAD (F:FILEVAR;BUF:ANY;VAR IOR:INTEGER;SZ,RB:INTEGER);
BLOCKWRITE(F:FILEVAR;BUF:ANY;VAR IOR:INTEGER;SZ,RB:INTEGER);
```

Diese Prozeduren werden für direkten Diskettenzugriff benutzt. FILEVAR ist ein typfreies File(FILE;). BUF ist eine beliebige Variable, die groß genug ist, um die Daten aufzunehmen. IOR ist eine Integervariable, die die Ergebnisse vom DOS aufnimmt. SZ ist die Anzahl der zu übertragenden Bytes und RB sollte immer auf Ø gesetzt sein.

Die Daten werden in Form der spezifizierten Anzahl von Bytes je nach Prozedur der BUF Variablen entnommen oder in sie hineingeschrieben.

3.4.18 OPEN

```
PROCEDURE OPEN(FILE,TITLE,RESULT);
```

Durch die OPEN-Prozedur wird die Flexibilität von ATARI PASCAL erhöht. FILE ist eine Filevariable beliebigen Typs. TITLE ist ein String, der den Filenamen enthält. RESULT ist ein VAR INTEGER Parameter der nach der Rückkehr von OPEN den gleichen Wert wie IORESULT hat. Zur gleichen Zeit können maximal drei Files eröffnet werden, E:, S:, K:-Files nicht eingeschlossen.

Die OPEN-Prozedur entspricht in der Ausführung der Sequenz ASSIGN(FILE,TITLE), RESET(FILE) und RESULT:=IORESULT.

Beispiele:

```
OPEN(INFILE,'D:FNAME.DAT',RESULT);
```

3.4.19 CLOSE, CLOSEDEL

```
PROCEDURE CLOSE   (FILE,RESULT);
PROCEDURE CLOSEDEL(FILE,RESULT);
```

Die CLOSE und CLOSEDEL-Prozeduren werden für das Schließen von Files bwz. Schließen mit Löschen benutzt. Die Close-Prozedur muß aufgerufen werden, um sicherzustellen, daß sämtliche in das File mit welcher Methode auch immer geschriebenen Daten vom Filebuffer zur Diskette geschafft werden. Die CLOSEDEL-Prozedur wird normalerweise benutzt, um zeitweilige Files nach ihrem Gebrauch zu löschen. FILE und RESULT verhalten sich wie in OPEN (siehe Abschnitt 3.4.18).

Files werden implizit geschlossen, wenn auf ein offenes File RESET angewendet wird.

Die Anwendung der CLOSE-Prozedur wird im Anhang unter File-Ein/Ausgabe beschrieben.

3.4.2Ø PURGE

```
PROCEDURE PURGE(FILE);
```

Die PURGE-Prozedur wird benutzt, um ein File zu löschen, dessen Name in einem String gespeichert ist. Sie müssen durch ASSIGN diesen Namen dem File zuordnen und können dann PURGE ausführen.

Beispiel:

```
ASSIGN(F,'D2:BADFILE.BAD');
PURGE(F);               (* D2:BADFILE.BAD wird gelöscht*)
```

3.4.21 IORESULT

FUNCTION IORESULT : INTEGER;

Nach jeder I/O-Operation wird der Wert der IORESULT-Funktion durch Laufzeitbibliotheksroutinen aktualisiert. Die generelle Regel für den ATARI 800 Homecomputer ist, daß ein Wert ungleich null auf einen Fehler hinweist und null den fehlerfreien Ausgang einer Operation kennzeichnet.

Beispiel:

```
ASSIGN(F,'D2:HELLO');
RESET(F);

IF IORESULT <> Ø THEN
  WRITELN('C:HELLO IS NOT PRESENT');
```

3.4.22 MEMAVAIL; MAXAVAIL

FUNCTION MEMAVAIL : INTEGER;
FUNCTION MAXAVAIL : INTEGER;

Die Funktionen MEMAVAIL und MAXAVAIL werden in ATARI PASCAL in Verbindung mit NEW und DISPOSE benutzt, um den HEAP-Speicherbereich zu verwalten. Die MEMAVAIL-Funktion ergibt zu jeder Zeit ohne Rücksicht auf die Fragmentierung den größten zur Verfügung stehenden Gesamtspeicherplatz. Die MAXAVAIL-Funktion wird zunächst unbenötigte Daten löschen ('Garbage Collection') und dann den größten zur Verfügung stehenden Block melden. MAXAVAIL kann benutzt werden, um eine Garbage Collection zu erzwingen, bevor ein zeitkritisches Programmsegment durchlaufen wird.

Das ATARI PASCAL-System entspricht völlig den im PASCAL Standard definierten NEW und DISPOSE-Mechanismen. Der HEAP-Bereich wächst vom unteren Ende des Datenbereichs aus, und der Stackrahmen (für Rekursion) wächst vom oberen Ende des Speichers aus abwärts.

3.4.23 SCHNELLER ÜBERBLICK ÜBER ALLE EINGEBAUTEN PROZEDUREN UND PARAMETER (INNERHALB JEDER GRUPPE ALPHABETISCH GEORDNET.)

Stringmanipulationen

```
PROCEDURE FILLCHAR(DESTINATION,LENGTH,CHARAKTER);
PROCEDURE MOVELEFT(SOURCE,DESTINATION,NUM BYTES);
PROCEDURE MOVERIGHT(SOURCE,DESTINATION,NUM_BYTES);
```

Bit-/Bytemanipulationen

```
PROCEDURE CLRBIT(BASIC_VAR,BIT_NUM);
FUNCTION HI(BASIC_VAR) : INTEGER;
FUNCTION LO(BASIC_VAR) : INTEGER;
PROCEDURE SETBIT(BASIC_VAR,BIT NUM);
FUNCTION SHL(BASIC_VAR,NUM) : INTEGER;
FUNCTION SHR(BASIC_VAR,NUM) : INTEGER;
FUNCTION SWAP(BASIC_VAR) : INTEGER;
FUNCTION TSTBIT(BASIC_VAR,BIT_NUM) : BOOLEAN;
```

Stringverarbeitung

```
FUNCTION CONCAT(SOURCE1,SOURCE2,...,SOURCEn) : STRING;
FUNCTION COPY(SOURCE,LOCATION,NUM_BYTES) : STRING;
PROCEDURE DELETE(TARGET,INDEX,SIZE);
PROCEDURE INSERT (SOURCE,DESTINATION,INDEX);
FUNCTION LENGTH(STRING) : INTEGER;
FUNCTION POS (PATTERN,SOURCE) : INTEGER;
```

Filehandling

```
PROCEDURE ASSIGN(FILE,NAME);
PROCEDURE BLOCKREAD(FILE,BUF,IOR,NUMBYTES,RELBLK);
PROCEDURE BLOCKWRITE (FILE,BUF,IOR,NUMBYTES,RELBLN);
PROCEDURE CLOSE(FILE,RESULT);
PROCEDURE CLOSEDEL(FILE,RESULT);
FUNCTION GNB(FILE) : CHAR;
PROCEDURE IORESULT : INTEGER;
PROCEDURE OPEN(FILE,TITLE,RESULT);
PROCEDURE PURGE(FILE);
FUNCTION WNB(FILE,CHAR) : BOOLEAN;
```

Verschiedene Routinen

```
FUNCTION ADDR(VARIABLE REFERENCE) : INTEGER;
PROCEDURE EXIT;
FUNCTION MEMAVAIL : INTEGER;
FUNCTION MAXAVAIL : INTEGER;
FUNCTION SIZEOF(VARIABLE OR TYPE NAME) : INTEGER;
```

3.5 NICHTSTANDARDGEMÄSSER DATENZUGRIFF

3.5.1 ABSOLUTE VARIABLE

<absolute var> ::=ABSOLUTE [<constant >] <var>

ABSOLUTE Variablen können deklariert werden, wenn Sie ihre Adressen bereits zur Compilierungszeit kennen. Sie deklarieren absolute Variablen ,indem Sie in der VAR-Deklaration eine spezielle Syntax verwenden. Absoluten Variablen wird durch den Compiler kein Platz in Ihrem Datensegment zugewiesen, d. h. Sie sind dafür verantwortlich, daß die absoluten Variablen nicht mit den durch den Compiler lokalisierten Variablen kollidieren.

Bemerkung: Stringvariablen dürfen nicht unterhalb der Adresse $1ØØ angelegt werden.

Beispiele:

```
I:      ABSOLUTE [$8ØØØ] INTEGER;
SCREEN: ABSOLUTE [$CØØØ] ARRAY[Ø..15] OF ARRAY[Ø..63]
        OF CHAR;
```

3.6 INLINE

ATARI PASCAL verfügt über eine weitere nützliche Funktion, die INLINE genannt wird. Diese Eigenschaft ermöglicht es, daß Sie Daten mitten in eine ATARI PASCAL Prozedur einsetzen. Auf diese Weise können Sie kurzen Maschinen- oder P-Code oder Konstantentabellen in ein ATARI PASCAL-Programm einsetzen.

3.6.1 SYNTAX

Die Syntax der INLINE-Funktion hat große Ähnlichkeit mit der Syntax eines PASCAL-Prozeduraufrufs. Auf das Wort INLINE folgt'(' und eine beliebige Anzahl von Argumenten, die durch '/' getrennt werden. Die Liste wird durch ')' beendet. Die Argumente zwischen den Schrägstrichen müssen Konstante oder Variable sein, die sich zu Konstanten auswerten lassen . Diese Konstanten können vom Typ CHAR, BOOLEAN, STRING, INTEGER oder REAL sein. Beachten Sie, daß ein in ' eingeschlossener String kein Byte erzeugt, das die Länge angibt, sondern lediglich das Datum für den String darstellt.

Literalkonstanten des Typs Integer werden als ein Byte zugewiesen, wenn ihr Wert im Bereich Ø bis 255 liegt. Benannte oder deklarierte Integerkonstanten werden immer in zwei Bytes zugewiesen.

3.6.2 ANWENDUNGEN

Die INLINE-Funktion kann benutzt werden, um Code einzuschieben oder Compilertabellen aufzubauen. Die folgenden beiden Abschnitte zeigen ein Beispiel für jede Anwendung.

Dieser Programmabschnitt zeigt, wie die INLINE-Funktion benutzt werden kann, um eine Compilierzeittabelle aufzubauen.

Beispiel:

```
PROGRAM DEMO_INLINE;

TYPE
  IDFIELD = ARRAY[1..4] OF ARRAY[1..1Ø] OF CHAR;

VAR
  TPTR : ^IDFIELD;

PROCEDURE TABLE;
BEGIN
  INLINE(      'ATARI       '/
               'HOME        '/
               'COMPUTER    '/
               'SYSTEMS...  ');
END;

BEGIN (* Hauptprogramm *)
TPTR := ADDR(TABLE)+5         (* +5 nur für P-Code *)
WRITELN (TPTR^[3]);           (* sollte 'COMPUTER' schrei-
END.                           ben *)
```

3.7 GRAFIK- UND SOUNDROUTINEN

Das Grafik-, Sound- und Controllerpaket besteht aus einem einschließbaren File, GSPROCS, und einem PASCAL-Modul, GRSND.ERL. Das einschließbare File definiert die im PASCAL-Modul zur Verfügung stehenden Einstiegspunkte. Das PASCAL-Modul muß in Ihr Programm eingebunden werden. Um das Paket zu benutzen, lassen Sie auf die Deklaration der globalen Variablen Ihres Programms die Eingabe (*$ID:GSPROCS*) folgen und führen INITGRAPHICS als ersten Befehl in Ihrem Hauptprogramm aus.

Beispiel:

```
PROGRAM GRSND;

LABEL
 ...;
CONST
 ...;
TYPE
 ...;
VAR
 ...;

(* Schließen Sie Grafik- und Tondefinitionen ein *)

(*$ID :GSPROCS*)

(* Lokale Prozeduren *)
```

```
PROCEDURE XXXX;
  BEGIN
    .....;
  END;

PROCEDURE YYYY;
  BEGIN
    .....;
  END;

(* Hauptprogramm *)

BEGIN
  INITGRAPHICS(5) (* Initialisierung des Grafikpaketes mit
                     maximal 5 Grafikmodi *)
  .....;
END.
```

Die folgenden Abschnitte beschreiben jede zur Verfügung stehende Funktion des Grafik- und Soundpaketes.

3.7.1 BILDSCHIRMTYPEN

TYPEs:

```
SCRN_TYPE = (SPLIT_SCREEN,FULL_SCREEN);
CLEAR_TYPE = (CLEAR_SCREEN,DO_NOT_CLEAR_SCREEN);
```

Diese bildschirmbezogenen Typen werden von der GRAPHICS-Prozedur benutzt, um den Grafikmodus zu definieren und ob der Schirm während der Grafikprozedur gelöscht wird oder nicht.

3.7.2 VARIABLE

VARs:

```
SCRNFILE : EXTERNAL TEXT;
GRRESULT : EXTERNAL INTEGER;
```

SCRNFILE kann benutzt werden, um auf dem Schirm Standard-Pascal-I/O durchzuführen, z. B.:

```
WRITE(SCRNFILE,'A');
```

Diese Variable wird das 'A' an den Schirm schicken und es dort, vom momentanen Modus abhängig, irgendwie darstellen. Beachten Sie, daß diese Technik normalerweise in Grafikmodus 1 oder 2 angewendet wird. Benutzen Sie für die anderen Grafikmodi die im Folgenden beschriebenen Prozeduren.

GRRESULT wird benutzt, um zu erkennen, ob während der Grafikroutinen ein Fehler aufgetaucht ist. Im Folgenden sind die Prozeduren und Funktionen aufgelistet, die GRRESULT verändern.

```
INITGRAPHICS  GRRESULT = Ø OK,255 = Fehler
GRAPHICS      GRRESULT = Ø OK,255 = Fehler
PLOT          GRRESULT = Ergebnis des XIO-Aufrufs
LOCATE        GRRESULT = Ergebnis des XIO-Aufrufs
FILL          GRRESULT = Ergebnis des XIO-Aufrufs
DRAWTO        GRRESULT = Ergebnis des XIO-Aufrufs
```

3.7.3 GRAFIKPROZEDUREN UND -FUNKTIONEN

3.7.3.1 INITIALISIERUNGSPROZEDUREN

PROCEDURE INITGRAPHICS(MAX_MODE:INTEGER);

INITGRAPHICS muß der erste Befehl jedes Programms sein, das das Grafik- und Soundmodul benutzt. Es gibt einen Parameter:

MAX_MODE Die maximale Anzahl der durch das Programm benutzten Modi, ein Wert zwischen Ø und 9.

Falls ein Fehler auftaucht, hat GRRESULT den Wert 255, anderenfalls den Wert Ø.

3.7.3.2 GRAFIKPROZEDUR

PROCEDURE GRAPHICS(MODE:INTEGER; SCREEN:SCRN_TYPE;CLEAR: CLEAR_TYPE);

GRAPHICS führt die gleichen Funktionen durch wie das GRAPHICS-Statement in ATARI-BASIC, hat aber anstelle von einem Parameter drei.

MODE gewünschter Grafikmode im Bereich von Ø bis MAX_MODE

SCREEN FULL_SCREEN oder SPLIT_SCREEN

CLEAR CLEAR_SCREEN oder DO_NOT_CLEAR_SCREEN

Falls ein Fehler auftaucht, hat GRRESULT den Wert 255, andernfalls den Wert Ø.

3.7.3.3 TEXTMODUSPROZEDUR

PROCEDURE TEXTMODE;

TEXTMODE schließt 'S:' und öffnet 'E:'. GRRESULT bleibt unverändert.

3.7.3.4 SETCOLORPROZEDUR

PROCEDURE SETCOLOR(REGISTER,HUE,LUMINANCE:INTEGER);

SETCOLOR erfüllt die gleichen Funktionen wie das SETCOLOR Statement in ATARI BASIC. GRRESULT bleibt unverändert.

REGISTER Ein Wert zwischen Ø und 4. Schlagen Sie im ATARI Bedienungshandbuch in Abschnitt 9 unter SETCOLOR nach.

HUE Ein Wert zwischen Ø und 15. Schlagen Sie im ATARI Bedienungshandbuch in Abschnitt 9 unter SETCOLOR nach.

LUMINANCE Ein Wert zwischen Ø und 14. Schlagen Sie im ATARI Bedienungshandbuch im Abschnitt 9 unter SETCOLOR nach.

3.7.3.5 COLOR PROZEDUR

PROCEDURE COLOR (COLOR_VALUE:INTEGER);

COLOR führt die gleiche Funktion durch wie der COLOR-Befehl in BASIC.

COLOR_VALUE Ein Wert zwischen Ø und 255. Schlagen Sie im ATARI 4ØØ/8ØØ Bedienungshandbuch in Abschnitt 9 unter COLOR nach.

3.7.3.6 PLOT PROZEDUR

PROCEDURE PLOT(X,Y:INTEGER);

PLOT führt die gleiche Funktion durch wie der PLOT-Befehl in BASIC. Er erzeugt einen Punkt in der gewählten Farbe an der Bildschirmposition x,y.

x horizontale Bildschirmkoordinate.
y vertikale Bildschirmkoordinate.

GRRESULT hat den Wert des XIO PUT Zeichenaufrufs.

3.7.3.7 LOCATE PROZEDUR

FUNCTION LOCATE(X,Y:INTEGER) : INTEGER;

LOCATE führt die gleiche Funktion durch wieder LOCATE-Befehl in ATARI-BASIC. Er ergibt den Wert des Farbpunktes an der Bildschirmposition x,y.

x horizontale Bildschirmkoordinate.
y vertikale Bildschirmkoordinate.

GRRESULT hat den Wert des XIO GET Zeichenaufrufs.

3.7.3.8 POSITION PROZEDUR

PROCEDURE POSITION(X,Y:INTEGER);

POSITION führt die gleiche Funktion durch wie der POSI-TION-Befehl in ATARI-BASIC. Er bewegt den unsichtbaren Cursor zur Position x,y. Beachten Sie, daß der Cursor

nicht vor der nächsten I/O-Operation bewegt wird.

- x horizontale Bildschirmkoordinate.
- y vertikale Bildschirmkoordinate.

3.7.3.9 DRAWTO PROZEDUR

```
PROCEDURE DRAWTO(X,Y:INTEGER);
```

DRAWTO führt die gleiche Funktion durch wie der DRAWTO-Befehl in ATARI-BASIC. Er zeichnet in der gewählten Farbe von der momentanen Grafikposition eine Linie zur Position x,y.

- x horizontale Bildschirmkoordinate.
- y vertikale Bildschirmkoordinate.

GRRESULT hat den Wert des XIO DRAWTO Zeichenaufrufs.

3.7.3.1Ø FILL PROZEDUR

```
PROCEDURE FILL(X,Y:INTEGER);
```

Fill führt die gleiche Funktion aus wie der XIO 18 Aufruf in ATARI BASIC. Der Unterschied besteht darin, daß er an der Position x,y einen Bildpunkt setzt, um den Cursor nach Beendigung von FILL an die Position x,y bringen zu können.

- x horizontale Bildschirmkoordinate.
- y vertikale Bildschirmkoordinate.

GRRESULT hat den Wert des XIO FILL Zeichenaufrufs.

3.7.4 SOUNDPROZEDUREN UND -FUNKTIONEN

3.7.4.1 SOUND PROZEDUR

```
PROCEDURE SOUND(VOICE,PITCH,DISTORTION,VOLUME:INTEGER);
```

SOUND führt die gleiche Funktion durch wie der SOUND-Befehl in ATARI BASIC. Er schaltet den durch VOICE spezifizierten Tonkanal mit dem durch PITCH, DISTORTION, VOLUME spezifizierten Ton ein.

VOICE	Einer der vier Tonkanäle Ø bis 3.
PITCH	Ein Wert zwischen Ø und 255. Schlagen Sie im ATARI-BASIC Manual im Abschnitt 1Ø unter SOUND nach.
DISTORTION	Ein Wert zwischen Ø und 14. Schlagen Sie im ATARI-BASIC Manual im Abschnitt 1Ø unter SOUND nach.
VOLUME	Ein Wert zwischen Ø und 15. Ø bedeutet ausgeschaltet, 15 ist die maximale Lautstärke.

3.7.4.2 SOUNDOFF PROZEDUR

PROCEDURE SOUNDOFF;

SOUNDOFF schaltet den Ton auf allen Kanälen ab.

3.7.5 CONTROLLER FUNKTIONEN

3.7.5.1 PADDLES

3.7.5.1.1 PADDLE FUNKTION

FUNCTION PADDLE(PDLNUM:INTEGER) : INTEGER;

PADDLE führt die gleiche Funktion durch wie der PADDLE-Befehl in ATARI-BASIC. Er liefert als Ergebnis den laufenden Wert eines der acht Paddle.

PDLNUM ist die Nummer des aufgerufenen Paddles, ein Wert zwischen Ø und 7.

3.7.5.1.2 TRIGGER FUNKTION

FUNCTION PTRIG (PDLNUM: INTEGER) : INTEGER;

PTRIG führt die gleiche Funktion durch wie der PTRIG-Befehl in ATARI-BASIC. Er liefert als Ergebnis den laufenden Triggerwert eines der acht Paddle.

PDLNUM ist die Nummer des aufgerufenen Paddles, ein Wert zwischen Ø und 7.

3.7.5.2 JOYSTICKS

3.7.5.2.1 STICK FUNKTION

FUNCTION STICK(STKNUM:INTEGER) : INTEGER;

STICK führt die gleiche Funktion durch wie der STICK-Befehl in ATARI-BASIC. Er liefert als Ergebnis den laufenden Wert eines der vier Joysticks.

STKNUM ist die Nummer des aufgerufenen Joysticks, ein Wert zwischen Ø und 3.

3.7.5.2.2 STRIG FUNKTION

FUNCTION STRIG(STKNUM:INTEGER) : INTEGER;

STRIG führt die gleiche Funktion durch wie der STRIG-Befehl in ATARI-BASIC. Er liefert als Ergebnis den laufenden Triggerwert eines der vier Joysticks.

STKNUM ist die Nummer des aufgerufenen Joysticks, ein Wert zwischen Ø und 3.

KAPITEL 4: LAUFZEITFEHLERBEHANDLUNG

ATARI PASCAL bietet zwei Typen der Laufzeitüberwachung: Bereichs- und Ausnahmeüberwachung.

Die Bereichsüberwachung wird auf Arrayindices und Unterbereichszuweisungen angewendet. Ohne besondere Maßnahmen ist diese Überwachung außer Funktion gesetzt. Sie können sie aber in jeder gewünschten Passage Ihres Programms durch Benutzung der $R und $X-Schalter (siehe Abschnitt 2.2.3.4, 2.2.3.5) aktivieren. Diese Abschnitte behandeln die Implementation dieser Mechanismen und zeigen Ihnen, wie Sie diese Mechanismen zu einer nichtstandardmäßigen Laufzeitfehlerbehandlung vorteilhaft einsetzen können.

Die generelle Vorgehensweise ist, daß Fehlerüberwachung und -routinen boolsche Flags setzen. Diese boolschen Flags werden zusammen mit einem Fehlercode auf den Stack geladen und darauf die Fehlerroutine @ERR für diese beiden Parameter aufgerufen. Die @ERR-Routine wird dann den boolschen Parameter testen. Wenn er den Wert 'false' hat, ist kein Fehler aufgetreten, die @ERR-Funktion kehrt zum compilierten Code zurück und die Ausführung wird fortgesetzt. Falls er den Wert 'true' hat, wird die @ERR-Routine eine Fehlermeldung ausgeben und Ihnen die Wahl über Abbruch oder Fortsetzung des Programms lassen.

Im Folgenden sind die Fehlernummern , die an die @ERR-Funktion übergeben werden, aufgelistet:

Wert	Bedeutung
1	Überprüfung auf Division durch Ø
2	Überprüfung auf Heapoverflow
3	Überprüfung auf Stringoverflow
4	Bereichsüberprüfung

4.1 BEREICHSÜBERPRÜFUNG

Wenn die Bereichsüberprüfung aktiviert ist, erzeugt der Compiler für jeden Arrayindex und jede Unterbereichszuweisung einen Aufruf der @CHK Routine. Die @CHK-Routine hinterläßt einen boolschen Wert auf dem Stack und der Compiler erzeugt einen @ERR-Aufruf nach dem @CHK-Aufruf. Falls die Bereichsüberwachung abgeschaltet ist und ein Index außerhalb des gültigen Bereichs liegt, werden unvorhersehbare Ergebnisse auftauchen. Der Wert der Unterbereichszuweisung wird auf der Byteebene abgeschnitten.

4.2 AUSNAHMEÜBERWACHUNG

Wenn die Ausnahmeüberwachung aktiviert ist, wird der Compiler die Fehlerflags (Division durch Ø, Stringoverflow, Heapoverflow) je nach Bedarf laden und die @ERR-Routine nach jeder Operation, die diese Flags setzen kann, aufrufen. Wenn die Ausnahmeüberwachung abgeschaltet ist, versuchen sich die Laufzeitroutinen nach Möglichkeit in einem benutzerfreundlichen Verhalten: Eine Division durch Null ergibt den größtmöglichen Wert, Heapoverflow hat keine Auswirkung und Stringoverflow resultiert in einer Abschneidung des Strings.

4.3 Benutzerdefinierte Fehlerbehandlung

Sie können Ihre eigene @ERR- Routine an Stelle der des Systems verwenden. Sie sollten diese Routine folgendermaßen deklarieren:

```
PROCEDURE @ERR(ERROR:BOOLEAN; ERRNUM:INTEGER);
```

Diese Routine wird, wie oben erwähnt, immer dann aufgerufen, wenn eine Fehlerüberprüfung notwendig geworden ist. Sie sollte die Fehlervariable überprüfen und falls diese den Wert 'false' hat, beendet werden. Falls die Fehlervariable den Wert 'true' hat, können Sie selber über eine angemessene Verfahrensweise entscheiden. Die Werte für ERRNUM finden Sie in Abschnitt 9.Ø.

4.4 SCHWERWIEGENDE FEHLER

Meldungen schwerwiegender Fehler können für Redigierzwecke entschlüsselt werden, können aber verwirrend sein. Der Fehler kann erst in die PASCAL-Fehlermeldung und dann in die standardmäßige ATARI Fehlermeldung übersetzt werden. Das folgende Beispiel illustriert den Übersetzungsprozeß:

```
     Fatal Error 64.88  --->  Pascal Error . ATARI Error
```

Umsetzung von hex nach dez:

```
       64 ---> 1ØØ  ,  88 ---> 136
Basis    16       1Ø     16       1Ø
```

Der PASCAL-Fehler 1ØØ bezieht sich auf einen Betriebssystemfehler. In diesem Beispiel werden wir unter der ATARI Fehlermeldung 136 finden, daß unser Fehler einem EOF gleichkommt.

Eine Aufzählung der vordefinierten, schwerwiegenden PASCAL-Fehler:

64: Fehler während einer Verkettung
65: fehlerhafter Pseudocode
66: fehlerhafter Pseudocode
67: undefinierter Pseudooperationscode
68: Stackoverflow (das Programm ist zu komplex)

KAPITEL 5: STRUKTUR- UND FORMAT EINES PASCALPROGRAMMS

Dieses Kapitel beschreibt die Datentypen und ihre Abspeicherung. Außerdem wird die Verwendung von Strings behandelt, sowie eine Beschreibung der Darstellung eines .COM-Files im Speicher unter DOS 2.ØS.

5.1 DATENTYPEN

Dieser Abschnitt beschreibt die Implementierung der Datentypen in ATARI PASCAL. Diese Tabelle faßt alle Datentypen zusammen:

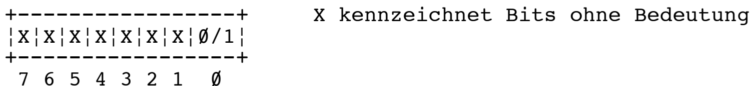
Datentyp	Größe	Bereich
CHAR	1 8-bit-byte	Ø..255
BOOLEAN	1 8-bit-byte	false..true
INTEGER	1 8-bit-byte	Ø..255
INTEGER	2 8-bit-bytes	-32768..32767
BYTE	1 8-bit-byte	Ø..255
WORD	2 8-bit-bytes	Ø..65535
FLOATING REAL	4 8-bit-bytes	1ØE-98..1ØE+98
STRING	1..256 bytes	--------------
SET	32 8-bit-bytes	Ø..255

5.1.1 CHAR

Der Datentyp CHAR ist durch den Gebrauch eines Bytes pro Zeichen implementiert. Das reservierte Wort PACKED kann auf Char-Arrays angewendet werden. Char-Variable existieren im Bereich von chr(Ø) bis chr(255). Wenn eine Char-Variable auf dem Stack abgespeichert wird, ist sie 16 Bit lang, das höhere Byte ist auf Ø gesetzt. Diese Vorgehensweise erlaubt Funktionen ORD, ODD, CHR und WRD eine Zusammenarbeit.

5.1.2 BOOLEAN

Der Datentyp BOOLEAN ist durch den Gebrauch eines Bytes pro boolscher Variabler implementiert. Wenn sie auf den Stack geschoben werden, werden 8 Bits mit dem Wert Ø mit abgespeichert, um Kompatibilität mit den eingebauten Operatoren und Routinen zu erreichen. Das reservierte Wort PACKED ist zugelassen, komprimiert die Datenstruktur aber nicht unter ein Byte pro Element (das geschieht sowohl mit als auch ohne PACKED Anweisung). ORD(TRUE)=ØØØ1 und ORD(FALSE)=ØØØØ. Die boolschen Operatoren AND, OR und NOT arbeiten nur mit EIN-Byte-Operanden. Sehen Sie unter & und ! nach, um Informationen über boolsche 16-Bit Operatoren zu erhalten.



```
+-----------------+
¦X¦X¦X¦X¦X¦X¦X¦Ø/1¦
+-----------------+
 7 6 5 4 3 2 1  Ø
```

X kennzeichnet Bits ohne Bedeutung

5.1.3 INTEGER

Der Datentyp INTEGER ist durch den Gebrauch von zwei 8-Bit Bytes für jede Integervariable implementiert. MAXINT = 32767 und INTEGERs können im Bereich -32768..32767 liegen. Die Deklaration eines Integerunterbereichtyps zwischen 0 und 255 benötigt dem gegenüber nur 1 Byte Speicherplatz. Integerkonstanten dürfen Hexadezimalzahlen sein, wenn sie von einem Dollarzeichen angeführt werden (z. B. $0F3B).

5.1.4 REAL

Die Implementierung des Typs REAL erfolgt in ATARI PASCAL genauso wie in ATARI BASIC. Sechs Datenbytes sind erforderlich, um eine Fließkommazahl zu implementieren. Das erste Byte ist folgendermaßen aufgeteilt: Bit 7 enthält das Vorzeichen der Mantisse, Bit 6 das Vorzeichen des Exponenten und die folgenden 6 Bits den Exponenten selber. Die Basis des Exponenten ist 100. Die übrigen 5 Bytes enthalten die BCD-codierte Mantisse. Die Genauigkeit liegt bei ca. 8 Stellen.

```
     +---------------------------------------///-----+
low  |Mantissen- Exponenten- Exponent | ms |    | ls | high
mem  |vorzeichen vorzeichen   6 bits  |    |    |    | mem
     +---------------------------------------///-----+
```

ms   = höchstwertige bits
ls   = niedrigstwertige bits
low mem   = niedrige Speicheradressen
high mem = hohe Speicheradressen

5.1.5 BYTE

Der Datentyp BYTE belegt ein einzelnes Byte. Er ist kompatibel mit den Typen INTEGER und CHAR. Diese Kompatibilität kann bei der Manipulation von Kontrollzeichen und der Handhabung von Characterarithmetik sehr nützlich sein. Character und Integer können dem Typ BYTE zugewiesen werden.

5.1.6 WORD

WORD ist eine vorzeichenlose, dem Prozessor eigene Variable. Alle Arithmetik- und Vergleichsoperationen, die auf Ausdrücke des Typs WORD angewendet werden, sind vorzeichenlos.

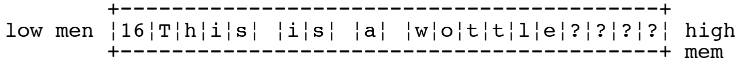
5.1.7 STRING

5.1.7.1 DEFINITION

Der vordefinierte Typ String ist einem packed Array of char ähnlich, in dem Byte 0 die dynamische Länge des Strings und Byte 1..n die Zeichen enthalten. String dürfen maximal 255 Zeichen lang sein. Die vorgegebene Länge

beträgt 80 Zeichen, kann aber durch die Deklaration einer STRING-Variablen geändert werden (s. umseitiges Beispiel).

Der String 'This is a Wottle' ist 16 Zeichen lang. Das folgende Diagramm zeigt, wie diese Zeichen in einem für maximal 20 Zeichen deklariertem String abgelegt werden.



low mem = niedrige Speicheradressen
high mem = hohe Speicheradressen

Falls die Anzahl der Zeichen eines Strings kleiner als die deklarierte Länge ist, sind die restlichen Zeichen undefiniert. Beachten Sie, daß somit die Länge im ersten Byte gespeichert ist und daß die erforderliche Gesamtanzahl für die Speicherung des Strings 17 Bytes beträgt.

Beispiel:

```
VAR
  LONG_STR:      STRING;       (*kann maximal 80 Zeichen
                                 enthalten *)
  SHORT_STR:     STRING [10];  (*kann maximal 10 Zeichen
                                 enthalten *)
  VERY_LONG_STR: STRING [255]; (*kann maximal 255 Zeichen
                                 enthalten, die erlaubte
                                 Höchstzahl *)
```

5.1.7.2 ZUWEISUNG

Die Zuweisung an einen String kann über das Zuordnungsstatement (:=), durch das Einlesen in eine Stringvariable über READ, READLN oder über die vordefinierten Stringfunktionen und -prozeduren erfolgen.

Beispiel:

```
PROCEDURE ASSIGN;
VAR
  LONG_STR  : STRING;
  SHORT_STR : STRING [12];
BEGIN
  LONG_STR := 'This string may contain as many as eighty
             characters';
  WRITELN(LONG_STR);

  WRITE('Type in a string 10 characters or less :');
  READLN(SHORT_STR);
  WRITELN(SHORT_STR);

  SHORT_STR := COPY(LONG_STR,1,11);
  WRITELN('COPY(LONG_STR..)=',SHORT_STR);
END;
```

Ausgabe:

```
This string may contain as many as eighty characters
Type in a string 10 characters or less : {123456} (Eingabe)
123456
COPY(LONG_STR..)=This is string m
```

Auf die einzelnen Zeichen einer Stringvariablen wird wie auf ein Array of char zugegriffen. Daher ist die normale Arrayindizierung über Konstante, Variable und Ausdrücke auch für Zuordnung und Zugriff auf die einzelnen Bytes innerhalb eines Strings erlaubt. Zugriff auf den String in seiner gesamten deklarierten Länge ist zulässig und verursacht auch dann keinen Laufzeitfehler, wenn ein Zugriff auf einen Teil des Strings durchgeführt wird, der jenseits seiner momentanen dynamischen Länge liegt. Wenn der String eine aktuelle Länge von 20 Zeichen hat und auf 30 Zeichen deklariert wurde, darf auf STRING [25] zugegriffen werden.

Beispiel:

```
PROCEDURE ACCESS;
VAR
  I:INTEGER;
BEGIN
  I:=15;
  LONG_STR:='123456789abcdef';
  WRITELN(LONG_STR);
  WRITELN(LONG_STR[6],LONG_STR[I-5]);
  LONG_STR[16] := '*';
  WRITELN(LONG_STR[16]);
  WRITELN(LONG_STR); (* gibt immer noch nur 15 Zeichen
                       aus *)
END;
```

Ausgabe:

```
123456789abcdef
6a
*
123456789abcdef
```

5.1.7.3 VERGLEICHE

Vergleiche sind zwischen zwei Variablen des Typs STRING (ohne Beachtung ihrer Länge) oder zwischen einer Stringvariablen und einem Literalstring zugelassen. Ein Literalstring ist eine Zeichenfolge zwischen einzelnen Anführungszeichen. Vergleiche dürfen auch zwischen einem String und einem einzelnen Zeichen erfolgen. Der Compiler formt das Zeichen zu einem String um, indem er den CONCAT-Buffer benutzt; deshalb ist der Vergleich zwischen dem Ergebnis einer CONCAT-Funktion und einem Zeichen sinnlos, da dieser Vergleich immer 'true' liefert.

Beispiel:

```
PROCEDURE COMPARE;

VAR
  S1,S2 : STRING [1Ø];
  CH1   : CHAR;

BEGIN
  S1 :='Ø12345678';
  S2 :='222345678';

IF S1 < S2 THEN
  WRITELN(S1,'is less than ',S2);

S1 := 'alpha beta';
IF S1 = 'alpha beta ' THEN
  WRITELN('trailing blanks don''t matter')
ELSE
  WRITELN('trailing blanks count');
IF S1 ='    alpha beta' THEN
  WRITELN('blanks in front don''t matter')
ELSE
  WRITELN('blanks in front do matter');
IF S1 = 'alpha beta' THEN
  WRITELN (S1,' = ', S1);
S1 := 'Z';
CH1 := 'Z';
IF S1 = CH1 THEN
  WRITELN('string and chars may be compared');
END;
```

Ausgabe:

```
Ø12345678 is less than 222345678
trailing blanks don't matter
blanks in front do matter
alpha beta = alpha beta
strings and chars may be compared
```

5.1.7.4 LESEN UND SCHREIBEN VON STRINGS

Strings können durch Benutzung der WRITE- oder WRITELN-Prozedur in Textfiles geschrieben werden. WRITELN führt zu einem auf den String folgenden Wagenrücklauf und Papiervorschub. Das Lesen der Strings erfolgt stets über READLN, da die Strings mit einem Wagenrücklauf und Papiervorschub beendet werden. READ funktioniert nicht, da die End-of-line-Zeichen falsch verarbeitet werden. Tabulatoren werden ausgewertet, wenn sie in eine Variable des Typs String eingelesen werden.

5.1.8 SET

Der Datentyp SET wird immer als 32-byte Datum gespeichert. Jedes Element der Menge wird als ein Bit abgespeichert. Das niederwertige Bit jeden Bytes ist das erste Bit in dem Byte der Menge. In der Abbildung ist die Menge 'A'..'Z' (Bit 65 .. 122) dargestellt.

Bytenummer	00	01	02	03	04	05	06	07	08	09	DA	0B	0C	...	1F
Inhalt	00	00	00	00	00	00	00	00	FE	FF	FF	07	00	...	00

KAPITEL 6: KOMPATIBLILITÄT

PASCAL ist in weit erheblicherem Maße standardisiert als BASIC. Fast jede Sprachversion basiert auf der im 'Pascal User Manual and Report' gegebenen Definition (Kathleen Jensen und Niklaus Wirth, Springer-Verlag 1974). Das PASCAL-Sprachsystem ist eine Übermenge des in diesem Buch beschriebenen PASCALs. Außerdem folgt ATARI PASCAL einem neueren Standard, hauptsächlich dem ISO-Standard (International Standards Organization, ähnlich ANSI). Es wird angenommen, daß sich zukünftige PASCAL-Versionen an diesem Standard orientieren werden und wir werden uns bemühen, ihn zu erreichen. ATARI hat die Wichtigkeit der Kompatibilität durch seine Erfahrung mit ATARI BASIC erkannt. Eine PASCAL-Version, die sich nach dem neu entwickelten ISO Standard richtet, ist ein positiver Schritt in Richtung Kompatibilität.

Ein mögliches Kompatibilitätsproblem kann der Umstand aufwerfen, daß ATARI PASCAL nicht gänzlich zu UCSD Pascal kompatibel ist. UCSD Pascal hat auf kleinen Rechnern eine beachtliche Popularität erlangt. Auch wenn es zutrifft, daß ATARI PASCAL nicht gänzlich zu UCSD Pascal kompatibel ist, sollte doch bedacht werden, daß beide Versionen um einen gemeinsamen Kern herum geschrieben wurden - Pascal, wie es von Jensen und Wirth definiert wurde. Die Unterschiede, obwohl vorhanden, sind nicht so gravierend wie in verschiedenen BASIC-Dialekten. Außerdem rechtfertigt die Überordnung des PASCAL-Sprachsystems die auftretenden Inkompatibilitäten.

Ein kurzer Vergleich der differierenden Eigenschaften der beiden PASCAL-Dialekte folgt. Einige Passagen dieses Vergleichs geraten notwendigerweise etwas technisch, da viele Unterschiede nur tief in den Details der Sprache zu finden sind.

6.1 INKOMPATIBILITÄTEN ZU UCSD PASCAL

1. Der vordefinierte Typ INTERACTIVE ist nur in UCSD Pascal verfügbar. Auf dem ATARI Computer ist jedes mit der Konsole verbundene File automatisch interaktiv, so daß dieser Typ nicht gebraucht wird und nur unnötige Unordnung in die Sprache bringen würde.

2. Die vordefinierte Prozedur SEEK ist nur in UCSD Pascal verfügbar.

3. UCSD Pascal benutzt UNITs für die Implementierung der modularen Compilation. Sie sind leicht zu verstehen, aber wesentlich restriktiver als ATARIs Implementation der modularen Compilation.

4. UCSD Pascal bietet SEGMENT-Prozeduren, um Overlays von Diskette zu erlauben. ATARI benutzt Standard DOS Methoden, um Overlays aufzurufen.

5. SETS können in UCSD Pascal wesentlich größer sein. Sie sind in ATARI PASCAL wesentlich schneller. Die ATARI PASCAL-Implementation entspricht mehr dem Geist des Jensen-Wirth Standards.

6. UCSD Pascal schließt eine Kompression gepackter Strukturen auf der Bit-Ebene ein. Diese Eigenschaft geht auf Kosten der Interpretergröße und der Ausführungszeit (besonders bei einer 65Ø2-Maschine, die weder dividieren noch multiplizieren kann).

7. UCSD Pascal verfügt über das Konstrukt EXIT <Prozedurname>, das in ATARI PASCAL nicht vorhanden ist, obwohl ATARI PASCAL ein EXIT ohne den Prozedurnamen vorsieht. Viele Pascalpuristen sind der Meinung, daß dies ein unstrukturiertes Konstrukt ist und somit der Sprachphilosophie entgegensteht.

8. UCSD Pascal schließt den Typ LONG INTEGER ein, der in ATARI PASCAL nicht zur Verfügung steht.

9. Einige Eigenschaften von UCSD Pascal sind betriebssystemabhängig, z. B. lange Filenamen und unit I/O (ähnlich XIO). Sie sind in ATARI PASCAL nicht implementiert.

6.2 IM ATARI PASCAL SPRACHSYSTEM ZUSÄTZLICH ZUR VERFÜGUNG STEHENDE EIGENSCHAFTEN

1. Das ATARI PASCAL ist ein komplettes ISO Standard Pascal. Einige der nicht in UCSD Pascal eingeschlossenen Eigenschaften sind Handhabung angleichbarer Arrays, Prozeduren und Funktionen als Parameter, lokale Files, PACK- und UNPACK-Prozeduren, READ und WRITE für Files, die nicht vom Typ Text sind, WRITE und WRITELN für boolsche Ausdrücke und das Herausspringen in eine umgebende Prozedur.

2. Der in ATARI PASCAL implementierte Pseudocode ist für die 65Ø2-CPU optimiert worden.

3. ATARI PASCAL benutzt das gleiche Betriebssystem wie alle anderen ATARI Programme. ATARI PASCAL und ATARI BASIC haben das gleiche Format, und Datenfiles können von beiden Sprachen gelesen werden . Sie müssen sich nicht der Mühe unterziehen, zwei unterschiedliche, inkompatible Betriebssysteme, wie in UCSD Pascal, zu lernen. Außerdem bietet ATARI PASCAL einen ATARI BASIC ähnlichen I/O-Zugriff. XIO, Grafik, Sound und die bekannten Geräte sind alle implementiert.

4. UCSD begrenzt die Anzahl der Prozedursegmente auf sechs und behindert damit die Entwicklung umfangreicher Anwendungen. ATARI PASCAL läßt die Entwicklung umfangreicher Anwendungen zu.

5. ATARI PASCAL bietet für Realzahlen 9 oder 1Ø gültige Stellen. UCSD Pascal liefert nur 6.5 gültige Stellen.

6. ATARI PASCAL gestattet dem Programmierer, Fehler aufzufangen und so den Programmabbruch zu verhindern.

7. ATARI PASCAL bietet Schutz beim Einlesen eines Strings. Falls der String für die aufnehmende Variable zu lang ist, sorgt ATARI PASCAL für eine Abtrennung. UCSD PASCAL überschreibt die dem String folgenden Speicherplätze und führt zu undefinierten Programmfehlern.

8. ATARI PASCAL hat ein durch ELSE erweitertes CASE-Statement. Falls der auswählende Ausdruck nicht zur Ausführung eines Statements führt, wird das ELSE-Statement ausgeführt. Die Ausführung eines ähnlich unpassenden CASE-Statements in UCSD Pascal führt zu undefinierten Ergebnissen.

9. Die modulare Compilation ist im ATARI PASCAL wesentlich flexibler. Lokale statische Variable, externe Prozeduren und Funktionen und der Gebrauch externer globaler Variablen fehlen in UCSD Pascal völlig.

10. ATARI PASCAL verfügt über den eingebauten Datentyp BYTE. Dieser Datentyp vermeidet den verwirrenden Gebrauch varianter CASE-Records, wenn Character als Integers manipuliert werden.

11. ATARI PASCAL verfügt über den Datentyp WORD. Als vorzeichenloser Typ ist er für Adressenberechnung und Maschinenprogrammierung sehr nützlich.

12. UCSD Pascal verfügt nicht über vollkommene Kompatibilität zwischen Strings und Characters. Strings und Characters sind in ATARI PASCAL total kompatibel.

13. Für systemabhängige Anwendungen läßt ATARI PASCAL eine Lockerung der Typüberwachunsregeln zu. Diese Lockerung läßt I/O auf der Maschinenebene und Speichermanipulation ohne die verwirrende Verwendung varianter CASE-Records im Programm zu.

14. ATARI PASCAL verfügt über die eingebauten Bitmanipulationsroutinen TSTBIT, SETBIT, CLRBIT, SHL und SHR. Jede Bitmanipulation in UCSD Pascal muß mit verwirrenden und maschinenabhängigen, varianten CASE-Records durchgeführt werden.

15. Sowohl in UCSD Pascal als auch in ATARI PASCAL sind die PUT/GET File I/O-Routinen recht langsam. ATARI PASCAL verfügt für byteweisen I/O über die Hochgeschwindigkeitsroutinen WND und GNB.

16. ATARI PASCAL implementiert die NEW und DISPOSE-Prozeduren vollständig, einschließlich einer Aufspaltungsverwaltung und dem erneuten Gebrauch durch DISPOSE gelöschter Speicherbereiche. UCSD Pascal verfügt über eine wesentlich restriktivere Version dieser Prozeduren. Diese Eigenschaft ist wichtig für alle Programme, die mit dynamischer Datenverwaltung arbeiten.

17. ATARI PASCAL erlaubt uneingeschränkten Gebrauch von Files. UCSD Pascal gestattet nicht den Gebrauch lokaler Files, Files in Records oder Arrays of files.

18. ATARI PASCAL schließt die Funktion ADDR ein. Diese Funktion ergibt die Adresse jeder Variablen, Prozedur oder Funktion. Diese Funktion ist bei der maschinenabhängigen Programmierung nützlich.

19. ATARI PASCAL verfügt über die INLINE-Funktion, die zur Erzeugung während der Compilierzeit konstanter Daten benutzt werden kann. Diese Funktion macht die Initialisierung überflüssig und erhöht damit die Ausführungszeit und vermindert die Größe des Codes.

20. ATARI PASCAL erlaubt die Ausgabe jeder Zahlenbasis von 2 bis 16.

21. ATARI PASCAL erlaubt die Eingabe von Dezimal- oder Hexadezimalzahlen.

22. ATARI PASCAL hat nicht die Parameterliste irgendeiner ISO-Standardroutine (insbesondere RESET oder REWRITE) erweitert. Für den Zugriff auf externe Files wurde eine neue Prozedur (ASSIGN) aufgenommen.

KAPITEL 7: SPRACHDEFINITION

7.1 EINFÜHRUNG

Kapitel 7 definiert die Eigenschaften von ATARI PASCAL, die jeder Implementierung des Compilers gemeinsam sind. Es wird vorausgesetzt, daß Sie mit Jensen und Wirths 'Report' und/oder mit ISO-Standardauszügen (OPS/7185) vertraut sind. ATARI PASCAL Eigenschaften, die von denen des ISO-Standards oder Jensen und Wirths 'Report' abweichen, sind im jeweiligen Abschnitt beschrieben. Jeder Abschnitt benutzt BNF (Backus Normal Form) Syntax als Bezugsnahrne. Die komplette BNF Beschreibung der Sprache wird in einem der Anhänge gegeben. Jeder Abschnitt bezieht sich auf Wirths 'Report'.

7.2 ZUSAMMENFASSUNG DER SPRACHE ATARI PASCAL

Eigenschaften des ISO Pascals beinhalten die Datentypen REAL, INTEGER, CHAR, BOOLEAN, mehrdimensionale ARRAYS, benutzerdefinierte RECORDs, POINTERtypen, Filevariablen, benutzerdefinierte Typen und Konstanten, und SETs (in dieser Version mit einem Basistyp von 256 Einbitelementen implementiert), Aufzählungs- und Unterbereichstypen.

In ISO Pascal sind ebenfalls PROCEDUREs, FUNCTIONs und PROGRAMs eingeschlossen. Die Übergabe von Prozeduren und Funktionen als Parameter an Pascalroutinen sind genauso Teil des ISO Standards wie angleichbare Arrays. Arrays gleichen Index- und Elementtyps, aber verschiedener Größe, dürfen an die gleiche Prozedur übergeben werden. Externe Parameter im PROGRAM-Statement werden auf der Syntaxebene unterstützt.

TYPED und TEXT-Files werden wie im Standard definiert durch die Pascalroutinen RESET, REWRITE, GET, PUT, READLN und WRITELN unterstützt. INPUT und OUTPUT sind als I/O ohne Vereinbarung definiert.

Alle ISO Statements, einschließlich WITH, REPEAT..UNTIL, CASE, WHILE- und FOR-Schleifen, IF..THEN..ELSE und GOTO werden unterstützt.

PACK und UNPACK werden unterstüzt, beeinflussen das Programm aber nicht (sämtliche Datenstrukturen sind bereits auf der Byteebene komprimiert). NEW und DISPOSE sind implementiert; sie verwalten die Zuweisung des HEAP-Bereichs.

Modulare Compilation ist eine Erweiterung des ATARI PASCAL Compilers. Variable und/oder Routinen können global oder lokal sein und können von jedem Modul oder dem Hauptprogramm erreicht werden.

Die erweiterten Datentypen STRING, BYTE und WORD sind im ATARI PASCAL Compiler implementiert. Der Stringtyp schließt ein die Länge angebendes Byte ein, auf das die höchstzulässige Anzahl Bytes folgt. Es werden Stringroutinen angeboten, um Zeichen einzusetzen (INSERT), sie zu lö-

schen (DELETE), die Position eines Zeichens aufzufinden (POS), einen Teilstring in einen anderen String zu kopieren (COPY) und zwei oder mehrere Strings und/oder Zeichen zu verketten (CONCAT). BYTE ist ein Ein-Byte-Datum für die Darstellung von Zahlen von Ø..255. Ein WORD-Datum erfordert zwei Bytes für die 8-Bit-CPU.

Es sind zusätzliche Prozeduren für das Diskettenfilemanagement implementiert. Ein Diskettenfile kann einem internen File zugeordnet werden und geschlossen oder gelöscht werden.

Die Manipulation von Bit und Bytes wird durch die Routinen TEST, SET, CLEAR, SHIFT LEFT und RIGHT vorgenommen. HI und LO ergeben Teile einer Variablen, SWAP tauscht hohes und niedriges Byte einer Variablen aus.

Verschiedene Routinen ermöglichen es, auf Programmdaten zuzugreifen: die Adresse einer Variablen oder Routine, Umfang einer Variablen oder eines Typs, die Verschiebung einer gegebenen Byteanzahl von einer Speicherstelle zur anderen oder die Auffüllung eines Datums mit einem bestimmten Zeichen. Außerdem kann zu jeder Zeit auf den Umfang des HEAP-Bereiches zugegriffen werden. Die Garbage Collection im HEAP-Bereich wird unterstützt.

Logische Operatoren für nicht-boolsche Typen sind implementiert.

Hexadezimale Literalzeichen müssen mit dem Dollarzeichen ($) versehen werden.

Eingeschlossene Files werden unterstützt.

Das ELSE-Statement im CASE-Statement wird angeboten.

Die Programmverkettung wird unterstützt. Das Verketten erfolgt dadurch, daß der neue Programmcode den alten vollständig ersetzt. Der HEAP-Bereich kann während einer CHAIN-Operation erhalten werden.

7.3 NOTATION, TERMINOLOGIE UND VOKABULAR

```
<letter>  :: = A ¦ B ¦ C ¦ D ¦ E ¦ F ¦ G ¦ H ¦ I ¦ J ¦
               K ¦ L ¦ M ¦ N ¦ O ¦ P ¦ Q ¦ R ¦ S ¦ T ¦
               U ¦ V ¦ W ¦ X ¦ Y ¦ Z ¦ a ¦ b ¦ c ¦ d ¦
               e ¦ f ¦ g ¦ h ¦ i ¦ j ¦ k ¦ l ¦ m ¦ n ¦
               o ¦ p ¦ q ¦ r ¦ s ¦ t ¦ u ¦ v ¦ w ¦ x ¦
               y ¦ z ¦ @ ¦

<digit>   :: = Ø ¦ 1 ¦ 2 ¦ 3 ¦ 4 ¦ 5 ¦ 6 ¦ 7 ¦ 8 ¦ 9 ¦
               A ¦ B ¦ C ¦ D ¦ E ¦ F ¦ (nur in Hexzahlen
                                        erlaubt)

<special symbol> ::= (reservierte Worte sind im Anhang
                      aufgelistet)

                    + ¦  - ¦ * ¦ / ¦ = ¦ <> ¦ < ¦ > ¦
                   <= ¦ >= ¦ ( ¦ ) ¦ [ ¦  ] ¦ ^ ¦
                   := ¦  . ¦ , ¦ ; ¦ : ¦  ' ¦

                      (die folgenden sind zusätzlich oder
                       Substitute)
                   (. ¦ .) ¦ \ ¦ ? ¦ ! ¦   ¦ ¦ $ ¦ &

                   (. ist gleich [
                   .) ist gleich ]
                   ? und'\ sind Synonyme
                   ! und ¦ sind Synonyme
                   &
```

Erweiterungen:

Das Symbol '@' ist ein zusätzlich erlaubtes Sonderzeichen. Es wurde aufgenommen, da die Laufzeitroutinen es als Sonderzeichen für das erste Zeichen ihres Namens benutzen. Die Aufnahme '@' verhindert Kollisionen mit benutzerdefinierten Namen, obwohl es dem Benutzer gestattet ist, diese Routinen aufzurufen (siehe 7.4).

Ein Kommentar, der mit (* beginnt, muß mit *) enden.
<Kommentar>::= (* <beliebige Zeichen> *)

7.4 BEZEICHNER, ZAHLEN UND STRINGS

```
<identifier>       ::= <letter> {<letter or digit or underscore>}
<letter or digit> ::= <letter) ¦ <digit> ¦ _

<digit sequence>  ::= <digit> {<digit>}
<unsigned integer>::= $<digit sequence>¦<digit sequence>

<unsigned real>   ::= <unsigned integer> . <digit sequence>¦
                      <unsigned integer> . <digit sequence>
                       E<scale factor>                     ¦
                      <unsigned integer>E<scale factor>
<unsigned number> ::= <unsigned integer> ¦ <unsigned real>
<scale factor>    ::= <unsigned integer> ¦ <sign><unsigned
                                                 integer>
<sign>            ::= + ¦ -
<String>          ::= <character > {<character>} ' ¦ ''
```

Alle Bezeichner sind bis zu acht Zeichen signifikant. Externe Bezeichner sind, je nach Gebrauch, bis zu sechs oder sieben Zeichen signifikant. Das Unterstreichungszeichen (_) ist zwischen Buchstaben und Ziffern in Bezeichnern zulässig, wird aber vom Compiler ignoriert (d. h. A_B ist das gleiche wie AB). Bezeichner dürfen mit '@' beginnen, um externe Laufzeitroutinen in einem Pascalprogramm zu deklarieren. Der Benutzer wird aber im Allgemeinen angewiesen, '@' zu vermeiden, um Kollisionen mit Laufzeitroutinen zu verhindern.

Zahlen dürfen dezimal oder hexadezimal sein. Ein '$' am Anfang einer Integerzahl führt dazu, daß der Compiler sie als Hexadezimalzahl interpretiert. Das Symbol <digit> enthält nun auch 'a', 'b', 'c', 'd', 'e' und 'f'. Sie dürfen in Groß- oder Kleinschrift eingegeben werden.

7.5 KONSTANTENDEFINITIONEN

```
<constant identifier>  ::= <identifier)
<constant>             ::= <unsigned number>              |
                           <sign> <unsigned number>       |
                           <constant identifier>          |
                           <sign> <constant identifier>   |
                           <string>
<constant definition>  ::= <identifier> = <constant>
```

Zusätzlich zu allen in Standardpascal zur Verfügung stehenden Konstantendeklarationen, bietet ATARI PASCAL die Deklaration einer Leerstringkonstanten.

Beispiel:

```
nullstr:=''
```

7.6 DATENTYPDEFINITIONEN

```
<type>            ::= <simple type>       |
                      <structured type>   |
                      <pointer type>
<type definition> ::= <identifier> = <type>
```

7.6.1 EINFACHE TYPEN

```
<simple type>     ::= <scalar type>    |
                       <subrange type> |
                       <type identifier>
<type identifier> ::= <identifier>
```

7.6.1.1 SKALARTYPEN

```
<scalar type> ::= ( <identifier {, <identifier>} )
```

7.6.1.2 STANDARDTYPEN

Die folgenden Typen sind Standardtypen in ATARI PASCAL:

INTEGER
REAL
BOOLEAN
CHAR

BYTE
WORD
STRING

Drei zusätzliche Standardtypen existieren in ATARI PASCAL. Sehen Sie im Anhang für Informationen über Darstellung und Gebrauch aller Standard- und strukturierten Typen nach.

STRING: packed Array [Ø..n] of char
Byte Ø gibt die dynamische Länge an, Byte 1..n enthalten die aktuellen Zeichen

BYTE : Unterbereich Ø..255, kompatibel mit CHAR

WORD : vorzeichenloses Maschinenwort

7.6.1.3 UNTERBEREICHSTYPEN

```
<subrange type) ::= <constant>..<constant>
```

7.6.2 STRUKTURIERTE TYPEN

```
<structured type> ::= <unpacked structured type>        ¦
                      PACKED <unpacked structured type>

<unpacked structured type> ::= <array type>  ¦
                               <record type> ¦
                               <file type>   ¦
                               <set type>
```

Das reservierte Wort PACKED wird durch den ATARI PASCAL-Compiler erkannt und wie folgt behandelt:

Alle Strukturen werden auf der Byte-Ebene komprimiert, auch wenn das reservierte Wort PACKED nicht gefunden wird.

7.6.2.1 ARRAYTYPEN

```
<array type>         ::= <normal array>  ¦
                         <string array>
<string array>       ::= STRING <max length>
<max length>         ::= [ <intconst> ]  ¦
                         <empty>
<intconst>           ::= <unsigned integer>  ¦
                         <int const id>
<int const id>       ::= <identifier>
<normal array>       :: = ARRAY [ < index type> {, <index
                          type>} ] OF
                          <component type>
<index type>         ::= <simple type>
<component type>     ::= <type>
```

Variable vom Typ STRING haben eine vordefinierte Länge von 81 Bytes (80 nutzbare Zeichen). Eine andere Länge kann auf das Wort STRING folgend in eckigen Klammern angegeben werden. Die Länge muß als Literal oder als Konstante definiert werden, z. B. STRING[5] oder STRING[xyz] (mit xyz als Konstante (xyz=10)).

Literal bzw. Konstante geben die Länge des Datenanteils an (ein zusätzliches Byte wird für die Angabe der Länge hinzugefügt.

7.6.2.2 RECORDTYPEN

```
<record type>    ::= RECORD <field list> END
<field list>     ::= <fixed part>                      ¦
                     <fixed part> ; <:variant part>    ¦
                     <variant part>
<fixed part>     ::= <record section> {; <record section>}
<record section> ::= <field identifier> {,<field identifier>}:
                     <type> ¦ <empty>
<variant part>   ::= CASE <tag field) <type identifier> OF
                     <variant> {;<variant>}
<variant>        ::= <case label list> : (<field list>)  ¦
                     <empty>
<case label list>::= <case label> {,<case label>}
<case label>     ::= <constant>
<tag field>      ::= <identifier> : ¦ <empty>
```

7.6.2.3 SETTYPEN

```
<set type>     ::= SET OF <base type>
<base type>    ::= <simple type>
```

Der Maximalbereich für den Basistyp ist 0..255.
Z. B. ist SET[0..10000] nicht zulässig.
SET of Char oder SET of 0..255 ist zulässig, aber SET of 0..256 nicht.

7.6.2.4 FILETYPEN

```
<file type>  ::= file {of <type>}
```

Typfreie Files sind erlaubt. Sie werden für Verkettung und bei der Verwendung von BLOCKREAD- und BLOCKWRITE-Prozeduren

benutzt. Seien Sie äußerst vorsichtig, wenn Sie typfreie Files benutzen.

Wenn Sie ein ASCII-Zeichenfile einlesen wollen und den Gebrauch impliziter Umwandlung für Integer und Reals wünschen, benutzen Sie den vordefinierten Typ TEXT. TEXT ist dasselbe wie FILE OF CHAR. TEXT impliziert die Umwandlung in READ- und WRITE-Prozeduraufrufen und kann auch mit READLN und WRITELN benutzt werden. Ein File des Types TEXT wird in folgender Weise deklariert:
'VAR F:TEXT'. Die falsche Weise, ein TEXT-File zu deklarieren ist: 'VAR F:FILE OF TEXT'. Sehen Sie im Anhang unter PASCAL-Filebehandlung nach.

7.6.3 ZEIGERTYPEN

<pointer type > ::= ^ <type identifier>

Pointertypen sind identisch mit dem Standard, ausgenommen das eine schwache Typüberwachung existiert, wenn die gelockerte Typüberwachung des Compilers eingeschaltet ist. In diesem Fall sind als Pointer definierte Zeiger und WORD-typen in jedem Fall kompatibel.

7.6.4 TYP- UND ZUWEISUNGSKOMPATIBILITÄT

Die häufigste Frage in bezug auf Standardpascal betrifft Fehlermeldungen über Typenprobleme des Compilers. Typen müssen identisch sein, wenn die Variable an einen VAR Parameter übergeben wird. Für Ausdrucks- und Zuweisungs-Statements müssen die Typen kompatibel sein. Um den Unterschied zwischen identisch und kompatibel zu verstehen, denken Sie sich die Typen als Zeiger auf Compilierzeit-Records. Wenn Sie einen Typ (z. B. T = ARRAY [1..1Ø] OF INTEGER) deklarieren, dann zeigt alles, was Sie als Typ T deklarieren, auf den Record, der den Typ T beschreibt. Wenn Sie andererseits zwei Variable folgendermaßen deklarieren:

```
VAR
  A1  :  ARRAY [1 .. 1Ø] OF INTEGER;
  A2  :  ARRAY [1 .. 1Ø] OF INTEGER;
```

sind sie nicht identisch. Der Compiler hat für jeden Typ eine neuen Record angelegt und daher zeigen A1 und A2 zur Compilierzeit nicht auf denselben Record. Im Allgemeinen gilt die Regel, daß die Typen nicht identisch sind, wenn Sie keine Typdefinition auffinden können.

CHR, ORD und WORD sind Typenumwandlungsoperatoren, die keinen Code erzeugen, aber dem Compiler mitteilen, daß das folgende 8-Bit-Datum als Typ CHAR, INTEGER oder WORD zu betrachten ist. Diese Operatoren dürfen in Ausdrücken und in Parameterlisten,außer in Zusammenhang mit VAR Parametern, benutzt werden.

Im Folgenden ein Auszug aus dem ISO Standard (IPS/7185), der beim American National Standards Institute erhält-

lich ist. Die ISO Standarddefinition kompatibler Typen ist wie folgt:

Die Typen T1 und T2 sind als kompatibel zu bezeichnen, wenn eine der folgenden vier Aussagen zutrifft:

(a). T1 und T2 sind gleichen Typs.

(b). T1 ist ein Unterbereich von T2, oder T2 ist ein Unterbereich von T1, oder T1 und T2 sind Unterbereiche desselben Übertyps.

(c). T1 und T2 sind per Bezeichnung gepackt oder beide per Bezeichnung ungepackt.

(d). T1 und T2 sind Stringtypen gleicher Komponentenanzahl (*1.).

Zuweisungskompatibilität. Ein Wert des Typs T2 soll als zuweisungskompatibel zu einem Typ T1 bezeichnet werden, wenn eine der folgenden fünf Aussagen zutrifft:

(a). T1 und T2 sind gleichen Typs, der weder ein Filetyp noch ein strukturierter Typ mit einer Filekomponente ist (diese Regel ist rekursiv anzuwenden).

(b). T1 ist ein REAL und T2 ein INTEGER.

(c). T1 und T2 sind kompatible Ordinaltypen und der Wert von T2 ist im geschlossenen Intervall durch T1 spezifiziert (*2.).

(d). T1 und T2 sind kompatible SETtypen und alle Werte von T2 sind im geschlossenen Intervall durch den Basistyp von T1 spezifiziert.

(e). T1 und T2 sind kompatible Stringtypen (*1.).

Überall, wo die Regeln der Zuweisungskompatibilität angewendet werden, gilt:

(a). Es gilt als Fehler, wenn T1 und T2 kompatible Ordinaltypen (*2.) sind, und der Wert von T2 nicht im geschlossenen Intervall durch T1 spezifiziert ist.

(b). Es gilt als Fehler, wenn T1 und T2 kompatible SETtypen sind, und irgendein Wert von T2 nicht im geschlossenen Intervall durch den Basistyp von T1 spezifiziert ist.

Fußnoten:

*1. Im ISO-Pascal sind Stringtypen Arrays of Char.

*2. Ordinaltypen sind benannte Zahlenbereiche oder Aufzählungen.

7.7 DEKLARATION UND BEZEICHNUNG VON VARIABLENEN

```
<variable>          ::= <var>            ¦
                        <external var> ¦
                        <absolute var>

<external var>      ::= <EXTERNAL <var>

<absolute var>      ::= ABSOLUTE [<constant>]<var>
<var>               ::= <entire variable>     ¦
                        <component variable) ¦
                        <referenced variable)
```

ABSOLUTE Variablen können Sie deklarieren, wenn Sie ihre Adresse während der Compilierzeit kennen. Sie deklarieren Variable als absolut, indem Sie in der VAR-Deklaration eine besondere Syntax verwenden. Absoluten Variablen wird durch den Compiler kein Platz in ihrem Datensegment zugewiesen, d. h. Sie sind dafür verantwortlich, daß sie nicht mit durch den Compiler abgelegten Variablen kollidieren. Beachten Sie, daß STRING-Variable nicht unterhalb der Adresse $1ØØ existieren dürfen. Durch diesen Umstand können die Laufzeitroutinen den Unterschied zwischen einer Stringadresse und einem Character auf dem Stackbeginn feststellen. Das Highbyte eines Characters auf dem Stack hat den Wert Ø. Zwischen dem Doppelpunkt und dem Variablentyp setzen Sie das Schlüsselwort ABSOLUTE und darauffolgend die Adresse der Variablen in Klammern.

Beispiele:

```
I:       ABSOLUTE [$8ØØ] INTEGER;
SCREEN:  ABSOLUTE [$CØØØ] ARRAY [Ø..15] OF ARRAY [Ø..63]
         OF CHAR;
```

7.7.1 GANZE VARIABLE

```
<entire variable>      ::= <variable identifier>
<variable identifier>  ::= <identifier>
```

7.7.2 KOMPONENTENVARIABLE

```
<component variable>  ::= <indexed variable> ¦
                          <field designator> ¦
                          <file buffer>
```

7.7.2.1 INDIZIERTE VARIABLE

```
<indexed variable> ::= <array variable> [<expression> {,
                       <expression>}]
<array variable>   ::= <variable>
```

Stringvariable sind für Indizierungszwecke wie packed arrays of char zu behandeln. Der zulässige Bereich beträgt Ø..maxlength. Der vordefinierte Wert für maxlength ist 8Ø.

7.7.2.2 FELDBEZEICHNER

```
<field designator> ::= <record variable>.<field identifier>
<record variable>  ::= <variable>
<field identifier> ::= <identifier>
```

7.7.2.3 FILEBUFFER

```
<file buffer>   ::= <file variable>^
<file variable> ::= <variable>
```

7.7.3 BEZUGSVARIABLE

```
<referenced variable> ::= <pointer variable>^
<pointer variable>    ::= <variable>
```

7.8 AUSDRÜCKE

```
<unsigned constant> ::= <unsigned number>                     ¦
                        <string>                              ¦
                        NIL                                   ¦
                        <constant identifier>
<factor>            ::= <variable>                            ¦
                        <unsigned constant>                   ¦
                        <function designator>                 ¦
                        (<expression>)                        ¦
                        <logical NOT operator> <factor>       ¦
<set>               ::= [<element list>]
<element list>      ::= <element> {,<element>}                ¦
                        <empty>
<element>           ::= <expression>                          ¦
                        <expression> .. <expression>
<term>              ::= <factor> <multipliying operator>
                        <factor>
<simple expression) ::= <term>                                ¦
                        <simple expression> <adding
                         operator> <term>                     ¦
                        <operator> <term>
<expression>        ::= <simple expression>                   ¦
                        <simple expression> <relational
                         operator> <simple expression>
```

Zur Kategorie der 16-Bit Additionsoperatoren gehören &, ! (auch ¦), \ (auch oder ?), die AND, OR bzw. das Einerkomplement NOT darstellen. Sie haben die gleiche Prioritätsfolge wie ihre äquivalenten boolschen Operatoren und akzeptieren jeden Operandentyp, dessen Größe <=2 Byte ist.

7.8.1 OPERATOREN

7.8.1.1 DER OPERATOR NOT

```
<logical NOT operator>  ::= NOT ¦ \ ¦ ?
```

\ und ? sind NOT-Operatoren für nicht-boolsche Operanden.

7.8.1.2 MULTIPLIKATIONSOPERATOREN

```
<multiplying operator>  ::=  * ¦ / ¦ DIV ¦ MOD ¦ AND ¦ &
```

& ist ein AND-Operator für nicht-boolsche Operanden.

7.8.1.3 ADDITIONSOPERATOREN

```
<adding operator > ::= + ¦ - ¦ OR ¦ ¦ ¦ !
```

! (oder ¦) ist ein OR-Operator für nicht-boolsche Operanden.

7.8.1.4 VERGLEICHSOPERATOREN

```
<relational operator> ::=  = ¦ <> ¦ < ¦ > ¦ <= ¦ >= ¦ IN
```

7.8.2 FUNKTIONSBEZEICHNER

```
<function designator>  ::= <function identifier>          ¦
                           <function identifier> (<parm>
                            {,<parm>})
<function identifier>  ::= <identifier>
```

7.9 STATEMENTS

```
<statement>            ::= <label> : <unlabelled statement> ¦
                           <unlabelled statement)
<unlabelled statement) ::= <simple statement>            ¦
                           <structured statement>
<label>                ::= <unsigned integer>
```

7.9.1 EINFACHE STATEMENTS

```
<simple statements>  ::= <assignment statement>   ¦
                         <procedure statement>    ¦
                         <goto statement>         ¦
                         <empty statement>
<empty statement>    ::= <empty>
```

7.9.1.1 ZUWEISUNGSSTATEMENTS

```
<assignment statement> ::= <variable> := <expression>    ¦
                           <function identifier> := <expression>
```

Zu der Liste an Ausnahmen zur Zuweisungskompatibilität kommt hinzu:

1. Integerausdrücke dürfen Pointertypvariablen zugewiesen werden.

Beispiel:

```
TYPE X = RECORD
           (* Felddeklarationen *)
         END;
VAR P : ^X;
    I : INTEGER;
................
P: =I+1;
```

2. Ausdrücke des Typs CHAR dürfen Variablen vom Typ STRING zugewiesen werden.

3. Variable vom Typ CHAR und Literalcharactern dürfen Variablen des Typs BYTE zugewiesen werden.

4. Ausdrücke, die vom Typ WORD ausgewertet werden, dürfen Pointervariablen zugewiesen werden.

5. Ausdrücke, die zum Typ INTEGER ausgewertet werden, dürfen Variablen des Types WORD zugewiesen werden.

7.9.1.2 PROCEDURE STATEMENTS

```
<procedure statements>  ::= <procedure identifier>
                                 (<parm> {,<parm>})     ¦
                            <procedure identifier>
<procedure identifier>  ::= <identifier>
<parm>                  ::= <procedure identifier> ¦
                            <function identifier>  ¦
                            <expression>           ¦
                            <variable>
```

Die maximale Parameteranzahl für Prozeduren und Funktionen beträgt fünfzig.

7.9.1.3 GOTO Statements

```
<goto statements> ::= goto<label>
```

7.9.2 STRUKTURIERTE STATEMENTS

```
<structured statements> ::= <repetitive statement>   ¦
                            <conditional statement>  ¦
                            <compound statement>     ¦
                            <with statement>
```

7.9.2.1 ZUSAMMENGESETZTE STATEMENTS

```
<compound statements> ::= BEGIN< statement> {;(statement)} END
```

7.9.2.2 BEDINGUNGSSTATEMENTS

```
<conditional statement> ::= <case statement> ¦ <if statement>
```

7.9.2.2.1 IF STATEMENTS

```
<if statement> ::= IF <expression> THEN <statement> ELSE
                                        <statement> ¦
                   IF <expression>THEN < statement>
```

7.9.2.2.2 CASE STATEMENTS

```
<case statement> ::= CASE <expression> OF
                     <case list>{,<case list>}
                     {ELSE <statement>}
                     END
<case list>      ::= <label list > : <statements>  ¦
                     <empty>
<label list>     ::= <case label> {,<case label>}
<case label>     ::= <non-real short scalar constant>
```

ATARI PASCAL implementiert in das CASE-Statement ein ELSE-Statement. Zusätzlich wird der Programmfluß, falls keiner der Selektoren der Labelliste zutrifft, 'durchfallen'.

Der Standard definiert diese Bedingung als Fehler.

Beispiel:

```
CASE CH OF
  'A' : WRITELN('A');
  'Q' : WRITELN('Q');
  ELSE
    WRITELN('NOT A OR Q');
END
```

7.9.2.3 WIEDERHOLUNGSSTATEMENTS

```
<repetitive statement> ::= <repeat statement>  ¦
                           <while statement>   ¦
                           <for statement>
```

7.9.2.3.1 WHILE STATEMENTS

```
<while statement> ::= WHILE <expression> DO <statement>
```

7.9.2.3.2 REPEAT STATEMENTS

```
<repeat statement> ::= REPEAT <statement> {,<statement)} UNTIL
                       <expression>
```

7.9.3.2.3 FOR STATEMENTS

```
<for statement> ::= FOR <ctrlvar> := <for list> DO <statement>
<for list>      ::= <expression> DOWNTO <expression> ¦
                    <expression) TO <expression>
<ctrlvar>       ::= <variable>
```

7.9.2.4 WITH STATEMENTS

```
<with statement> ::= WITH <record variable list> DO <statement>
<record variable list> ::= <record variable> {,<record variable>}
```

Beachten Sie, daß der ISO Standard von dem durch Jensen und Wirth definierten PASCAL insofern abweicht, daß er nur lokale Variable als FOR-Schleifenkontrollvariable zuläßt. Das verhindert Programmfehler, die durch unangemessenen Gebrauch von globalen Variablen als FOR-Schleifenkontrollvariable entstehen, sobald sie fünffach geschachtelt sind.

Sie sind auf 16 FOR und/oder WITH-Statements in einer einzelnen Prozedur/Funktion beschränkt. Diese Beschränkung ermöglicht, daß der Compiler eine feste Anzahl zeitweiser Datenplätze (16 Worte) im Datensegment der Prozedur/Funktion zuweisen kann.

7.1Ø PROZEDURDEKLARATIONEN

```
<procedure declaration>    ::= EXTERNAL <procedure heading> ¦
                               <procedure heading> <block>

<block>                    ::= <label declaration part>
                               <constant definition part>
                               <type definition part>
                               <variable declaration part>
                               <procfunc declaration part>
                               <statement part>

<procedure heading>        ::= PROCEDURE <identifier> <parmlist>; ¦
                               PROCEDURE <identifier>;

<parmlist>                 ::= (<fparm> {,<fparm>})

<fparm>                    ::= <procedure heading> ¦
                               <function heading>  ¦
                               VAR <parm group>    ¦
                               <parm group>

<parm group)               ::=<identifier> {,<identifier>} :
                                <type identifier>               ¦
                              <identifier> {,<identifier>} :
                                <conformant array>

<conformant array>         ::= ARRAY [<indxtyp> {,<indxtyp>} ] OF
                                 <conarray2>

<conarray2>                ::= <type identifier>  ¦
                               <conformant array>

<indxtyp                   ::= <identifier>..<identifier> :
                                 <ordtypid>

<ordtypid>                 ::= <scalar type identifier>    ¦
                               <subrange type identifier>

<scalar type identifier>   ::= <identifier>

<subrange type identifier> ::= <identifier>

<label declaration part>   ::= <empty)    ¦
                               LABEL <label> {,<label>} ;

<constant definition part> ::= <empty>    ¦
                               CONST
                                 <constant definition>
                                 {;<constant definition>};

<type definition part>     ::= <empty>    ¦
                               TYPE
                                 <type definition>
                                 {;<type definition>};
```

```
<variable declaration part>:: = <empty>    ¦
                                VAR
                                  <variable declaration>
                                  {;<variable declaration>};

<procfunc part>               ::= { <proc or func> ; }

<proc or func>                ::= <procedure declaration>    ¦
                                  <function declaration>

<statement part>              ::= <compound statement>
```

7.1Ø.1 STANDARDPROZEDUREN

Im Folgenden eine Liste der in ATARI PASCAL eingebauten Prozeduren. Schlagen Sie in Kapitel 3 für den Gebrauch und die Parameter nach. Diese Prozeduren sind in einem Rahmen um das Programm vordefiniert. Daher hat jede benutzerdefinierte Routine gleichen Namens eine höhere Priorität.

NEW	DISPOSE	EXIT	INSERT
DELETE	COPY	CONCAT	FILLCHAR
MOVELEFT	MOVERIGHT	CLRBIT	HI
LO	SETBIT	SHL	SHR
SWAP	TSTBIT	LENGTH	POS
ADDR	MOVE	MAXAVAIL	MEMAVAIL
SIZEOF			

7.1Ø.1.1 FILEMANAGEMENTPROZEDUREN

Alle standardmäßigen Filemanagementprozeduren sind eingeschlossen. Zusätzlich gibt es die Prozedur ASSIGN(F,string) mit F als File und String als Literalstring oder Stringvariable. ASSIGN weist einen externen Filenamen, der in String spezifiziert ist, dem File F zu. Es wird vor einem RESET oder REWRITE benutzt. Schlagen Sie für weitere Details in Abschnitt 3.4.15 nach.

Im Folgenden eine Liste der Filemanagementprozeduren:

GET	PUT	RESET	REWRITE
ASSIGN	CLOSE	CLOSEDEL	PURGE
OPEN	BLOCKREAD	BLOCKWRITE	READ
CHAIN	PAGE	IORESULT	
GNB	WNB	WRITELN	
WRITE	READLN		

7.1Ø.1.2 DYNAMISCHE DATENZUWEISUNGSPROZEDUREN

NEW   DISPOSE

Zusätzlich zu NEW und DISPOSE sind MEMAVAIL und MAXAVAIL vorhanden.

7.1Ø.1.3 DATENTRANSFERPROZEDUREN

PACK (A,I,Z)   UNPACK (Z,A,I)

7.1Ø.2 FORWARD

Forwardprozedurdeklarationen sind in ATARI PASCAL implementiert . Es wird jedoch empfohlen, sie nur dann zu benutzen, wenn strenge Pascalübereinstimmung gefordert ist. Der Dreiphasencompiler macht Forwarddeklarationen unnötig.

7.10.3 ANGLEICHBARE ARRAYS

Beachten Sie, daß der ISO Standard ein Schema angleichbarer Arrays für die Übergabe Arrays ähnlicher Struktur (z. B. gleiche Anzahl der Dimensionen, kompatible Indextypen und gleicher Elementtyp) aber unterschiedlicher oberer und unterer Grenzen aufgenommen hat. Jetzt können Sie z. B. ein Array, das auf 1..10, und ein Array, das auf 2..50 dimensioniert ist, an eine Prozedur übergeben, die ein Array erwartet. Sie definieren das Array als VAR Parameter und während der Arraydeklaration definieren Sie Variable, die die Ober- und Untergrenze des Arrays aufnehmen. Diese Daten für Ober- und Untergrenze werden während der Laufzeit durch den erzeugten Code gefüllt. Um Arrays auf diese Weise zu übergeben, muß der Indextyp zum Typ der Grenzen des angleichbaren Arrays kompatibel sein.

Im Folgenden ein Beispiel für die Übergabe zweier Arrays an eine Prozedur, das die Inhalte der Arrays über das File OUTPUT anzeigt.

```
PROGRAM DEMOCON;

TYPE
  NATURAL = 0..MAXINT; (* für die Deklaration der an-
                          gleichbaren Arrays *)

VAR
  Al : ARRAY [1..10] OF INTEGER;
  A2 : ARRAY [1..20] OF INTEGER;

PROCEDURE DISPLAYIT (

  VAR AR1:ARRAY  [LOWBOUND..HIGHBOUND:NATURAL] OF INTEGER;
                   );

(* Diese Deklaration definiert drei Variable:

    AR1       : Das übergebene Array
    LOWBOUND  : Untergrenze von AR1 (Zur Laufzeit übergeben)
    HIGHBOUND : Obergrenze von AR1 (Zur Laufzeit übergeben)

*)

VAR
  I : NATURAL;
(* kompatibel mit dem Indextyp des angleichbaren Arrays *)

BEGIN
  FOR I:=LOWBOUND TO HIGHBOUND DO
    WRITELN('INPUT ARRAY[',I,']=',AR1[I])
END;

BEGIN (* Hauptprogramm *)

  DISPLAYIT(A1); (* Aufruf von DISPLAYIT, explizite Übergabe
                    von Al und implizite Übergabe von 1..10 *)
```

```
  DISPLAYIT(A2); (* Aufruf von DISPLAYIT, explizite Übergabe
                    von A2 und implizite Übergabe von 2..2Ø *)
END.
```

7.11 FUNKTIONSDEKLARATIONEN

```
<function decl>    ::= EXTERNAL <function heading> ¦
                       <function heading> <block>

<function heading> :: = FUNCTION <identifier> <parmlist> : <result type>;
                        FUNCTION<identifier> : <result type>;

<result type>      ::= <type identifier>
```

7.11.1 STANDARDFUNKTIONEN

Im Folgenden eine Liste der Namen der vorhandenen Standardfunktionen

ABS	SQR	SIN	COS
EXP	LN	SQRT	ARCTAN
ODD	TRUNC	ROUND	ORD
WRD	CHR	SUCC	PRED
EOLN	EOF	IORESULT	MEMAVAIL
MAXAVAIL	ADDR	SIZEOF	POS
LENGTH			

7.11.1.1 ARITHMETISCHE FUNKTIONEN

7.11.1.2 BOOLSCHE FUNKTIONEN

7.11.1.3 ÜBERTRAGUNGSFUNKTIONEN

WRD (x): Der Wert x (eine Variable oder Ausdruck) wird als der WORD- (vorzeichenlose Integerzahl) Wert von x behandelt. Integerliteralkonstanten sind nicht vom Typ WORD. Daher ist W:=3 unzulässig, wenn W vom Typ WORD ist. Sie müssen W:=WRD(3) verwenden.

7.11.1.4 WEITERE STANDARDFUNKTIONEN

Filebehandlung: (F ist eine Filevariable, siehe im Anhang unter Files)

PUT(F) GET(F) RESET(F) REWRITE(F) PAGE(F) EOF(F) EOLN(F)

Dynamische Zuweisung: (Tn ist ein variabler Recordselector, P ein Zeiger)

NEW (P) NEW(P,T1,T2,...,Tn) DISPOSE(P) DISPOSE (P,T1,T2,...Tn)

Datenübertragungsfunktionen: (siehe Jensen und Wirth, S. 1Ø6 für eine komplette Beschreibung.)

PACK(A,I,Z) UNPACK(Z,A,I)

Arithmetische Funktionen

ABS(X) OR ABS(I) - ergibt den Typ des Arguments
SQR(X) OR SQR(I) - ergibt bei Übergabe von Integer Integer usw.

Übertragungsfunktionen (SC ist ein nichtreeller, kurzer Skalartyp)

Zur Compilierzeit implementiert und nicht codeerzeugend:

ODD(SC):BOOLEAN ORD(SC):INTEGER CHR(SC):CHAR WRD(SC):WORD

Zur Compilierzeit implementiert und codeerzeugend:

SUCC(jeder Skalartyp außer REAL)
PRED(jeder Skalartyp außer REAL)

7.12 EIN- UND AUSGABE

ATARI PASCAL unterstützt alle Standardpascal I/O Funktionen.

7.12.1 DIE PROZEDUR READ

Das Einlesen in Unterbereiche ist implementiert, aber auch wenn die Bereichsprüfung eingeschaltet ist, findet keine Bereichsüberprüfung statt.

7.12.2 DIE PROZEDUR READLN

```
<readcall>         ::= <read or readln> {( {<filevar>,} {<varlist>} )}

< read or readln> ::= READ ¦ READLN

(filevar>          ::= <variable>

<varlist>          ::= <variable> {,(variable)}
```

7.12.3 DIE PROZEDUR WRITE

7.12.4 DIE PROZEDUR WRITELN

```
<writecall>         ::= <write or writeln> {( {<filevar> ,} {exprlist})}

<write or writeln)  ::= WRITE ¦ WRITELN

<exprlist>          ::= <wexpr> {,<wexpr>}

<wexpr>             ::= <expression> {: <width expr>" {:<dec expr>}}

<width expr>        ::= <expression>

<dec expr>          ::= <expression>
```

Um Integerzahlen mit einer anderen Basis als zehn auszugeben, können Sie für <dec expr> eine negative Dezimalzahl einsetzen.

Beispiel:

```
WRITE (I:15:-16) (* I wird zur Basis 16 ausgedruckt *)
```

Sie dürfen im WRITE oder WRITELN-Statement keine Funktion als Parameter verwenden, die Input- oder Outputoperationen durchführt. Das schließt Zugriffsroutinen wie z. B. GNB ein. Die Filezeiger werden durch die Leseroutinen verändert, so daß die Ausgabe auf das Eingabefile erfolgen würde.

7.12.5 ZUSÄTZLICHE PROZEDUREN

(siehe auch Abschnitt 7.1Ø.1.1)

Die Ein- und Ausgabe einer Variablen des Typs WORD erfolgt nicht mit den Standardprozeduren READ und WRITE. Dafür werden zwei neue Prozeduren, READHEX und WRITEHEX, eingeführt.

Diese neuen Prozeduren erlauben hexadezimale Ein- und Ausgabe jedes beliebigen Ein-, Zwei- oder Vierbytetyps, wie z. B. Integer, Char, Byteunterbereich, Aufzählungstyp, Word und Long Integer. Siehe auch Kapitel 3.4 unter ATARI PASCAL Erweiterungen.

7.13 PROGRAMME

```
<program>         ::= <program heading> <block >.  ¦
                      <module heading>
                        <label declaration part>
                        <constant definition part>
                        <type definition part>
                        <variable declaration part>
                        <profunc declaration part>
                        MODEND.

<program heading> ::= PROGRAM <identifier> (<prog parms>);

<module heading>  ::= MODULE <identifier>;

<prog parms>      ::= <identifier> {,<identifier>}
```

Abgesehen von der Aufnahme der Module ist diese Definition identisch mit dem Standard, siehe auch Kapitel 3.

ANHANG A: SPRACHSYNTAXBESCHREIBUNG

```
<letter>  ::= A ¦ B ¦ C ¦ D ¦ E ¦ F ¦ G ¦ H ¦ I ¦ J ¦
              K ¦ L ¦ M ¦ N ¦ O ¦ P ¦ Q ¦ R ¦ S ¦ T ¦
              U ¦ V ¦ W ¦ X ¦ Y ¦ Z ¦ a ¦ b ¦ c ¦ d ¦
              e ¦ f ¦ g ¦ h ¦ i ¦ j ¦ k ¦ l ¦ m ¦ n ¦
              o ¦ p ¦ q ¦ r ¦ s ¦ t ¦ u ¦ v ¦ w ¦ x ¦
              y ¦ z ¦ @

<digit>   ::= Ø ¦ 1 ¦ 2 ¦ 3 ¦ 4 ¦ 5 ¦ 6 ¦ 7 ¦ 8 ¦ 9 ¦
              A ¦ B ¦ C ¦ D ¦ E ¦ F   (nur in Hexzahlen
                                       erlaubt)

<special symbol> ::= (reservierte Worte sind im Anhang B aufgelistet)

                     +  ¦ -  ¦ * ¦ / ¦ = ¦ <> ¦ < ¦ > ¦
                     <= ¦ >= ¦ ( ¦ ) ¦ [ ¦ ]  ¦ ^ ¦
                     := ¦ .  ¦ , ¦ ; ¦ : ¦ '  ¦

(die folgenden sind zusätzlich oder Substitute)

                     (. ¦ .) ¦ \ ¦ ? ¦ ! ¦ ¦ ¦ $ ¦ & ¦

                     (. ist gleich [
                     .) ist gleich ]
                     ? und \ sind Synonyme
                     ! und ¦ sind Synonyme
                     &

<identifier>        ::= <letter>{(letter or digit or underscore>}

<letter or digit>   ::= <letter> ¦ <digit> ¦ _

<digit sequence>    ::= <digit> {<digit>}

<unsigned integer>  ::= $ <digit sequence> ¦
                        <digit sequence>

<unsigned real>     ::= <unsigned integer> . <digit sequence> ¦
                        <unsigned integer> . <digit sequence>
                        E <scale factor>                      ¦
                        <unsigned integer> E <scale factor>

<unsigned number>   ::= <unsigned integer> ¦ <unsigned real>

<scale factor)      ::= <unsigned integer> ¦ <sign><unsigned integer>

<sign>              ::= + ¦ -

<string>            ::= '<character> {<character>} ¦ ''

<constant identifier> ::= <identifier>

<constant>          ::= <unsigned number>         ¦
                        <sign><unsigned number>   ¦
                        <constant identifier>     ¦
                        <sign><constant identifier>  ¦
                        <string>

<constant definition> ::= <identifier> = <constant>
```

```
<type>                ::= <simple type>      ¦
                          <structured type>  ¦
                          <pointer type>

<type definition>     ::= <identifier> = <type>

<simple type>         ::= <scalar type>     ¦
                          <subrange type>   ¦
                          <type identifier>

<type identifier>     ::= <identifier>

<scalar type>         ::= ( <identifier> {, <identifier>} )

<subrange type>       ::= <constant> .. <constant>

<structured type >    ::= <unpacked structured type>  ¦
                          PACKED <unpacked structured type>

<unpacked structured type> ::= <array type>   ¦
                               <record type>  ¦
                               <set type>     ¦
                               <file type>

<array type>          ::= <normal array> ¦
                          <string array>

<string array>        ::= STRING <max length>

<max length>          :: = [<intconst>] ¦
                           <empty>

<intconst>            ::= <unsigned integer> ¦
                          <int const id>

<int const id>        ::= <identifier>

<normal array)        ::= ARRAY [<index type> {,<index type>}] OF
                          <component type>

<index type>          ::= <simple type>

<component type>      :: = <type>

<record type>         ::= RECORD <field list> END

<field list>          ::= <fixed part>                        ¦
                          <fixed part> ; <variant part>       ¦
                          <variant part>

<fixed part>          ::= <record section> {,<record section>}

<record section)      ::= <field identifier> {,<field
                           identifier>} : <type>          ¦
                          <empty>

<variant part>        ::= CASE <tag field> <type identifier> OF
                          <variant> {;<variant>}
```

```
<variant>            ::= <case label list> : (<field list>) |
                         <empty>

<case label list>    ::= <case label> {,<case label>}

<case label>         ::= <constant>

<tag field>          ::= <identifier> :  |
                         <empty>

<set type>           ::= SET OF <base type>

<base type>          ::= <simple type>

<file type>          ::= file {of <type>}

<variable>           ::= <var>            |
                         <external var>   |
                         <absolute var>

<external var>       ::= EXTERNAL <var>

<absolute var>       ::= ABSOLUTE [<constant>] <var>

<var>                ::= <entire variable>     |
                         <component variable>  |
                         <referenced variable>

     Deklaration einer Variablen des Typs STRING:

     <identifier> {,<identifier>} : STRING {[<constant>]}

<entire variable>     ::= <variable identifier>

<variable identifier> ::= <identifier>

<component variable>  ::= <indexed variable>  |
                          <field designator>  |
                          <file buffer>

<indexed variable> ::= <array variable> [<expression> {,<expression>}]

<array variable>   ::= <variable>

<field designator> ::= <record variable> . <field identifier>

<record variable>  ::= <variable>

<field identifier> ::= <identifier>

<file buffer>      ::= <file variable> ^

<file variable>    ::= <variable>

<referenced variable> ::= <pointer variable> ^

<pointer variable> ::= <variable>

<unsigned constant> ::= <unsigned number> |
                        <string>          |
                        NIL               |
                        <constant identifier>
```

```
<factor>             ::= <variable>                 ¦
                         <unsigned constant>        ¦
                         <function designator>      ¦
                         ( <expression> )           ¦
                         <logical NOT operator> < factor>
                         <set>

<set>                ::= [ <element list> ]

<element list>       ::= <element> {,<element>}     ¦
                         <empty>

<element>            ::= <expression>               ¦
                         <expression>..<expression>

<term>               ::= <factor> <multiplying operator> <factor>

<simple expression> ::= <term>                                      ¦
<                        <simple expression> <adding operator> <term> ¦
                         <adding operator> <term>

<expression>         ::= <simple expression>              ¦
                         <simple expression> <relational
                              operator> <simple expression>

<logical NOT operator> ::= NOT ¦ \ ¦ ?
```

\ und ? sind NOT-Operatoren für nicht-boolsche Operanden.

```
<multiplying operator> ::= * ¦ / ¦ DIV ¦ MOD ¦ AND ¦ &
```

& ist ein AND-Operator für nicht-boolsche Operanden.

```
<adding operator>    ::= + ¦ - ¦ OR ¦ ¦ ¦ !
```

(! oder ¦) ist ein OR-Operator für nicht-boolsche Operanden.

```
<relational operator> ::= = ¦ <> ¦ < ¦ <= ¦ > ¦ >= ¦ IN

<function designator> ::= <function identifier>    ¦
                          <function identifier> (<parm> {,<parm>})

<function identifier> ::= <identifier>

<statement>           ::= <label> : <unlabelled statement>  ¦
                          <unlabelled statement>

<unlabelled statement>::= <simple statement>           ¦
                          <structured statement>

<label>               :: = <unsigned integer>

<simple statement>    ::= <assignment statement>     ¦
                          <procedure statement>      ¦
                          <goto statement>           ¦
                          <empty statement>

<empty statement>     ::= <empty>

<assignment statement>::= <variable> := <expression>     ¦
(function identifier):=(expression)
```

```
<procedure statement>  ::= <procedure identifier> (<parm> {,<parm>}) ¦
                           <procedure identifier>

<procedure identifier> ::= <identifier>

<parm>                 ::= <procedure identifier>  ¦
                           <function identifier>   ¦
                           <expression>            ¦
                           <variable>

<goto statement>       ::= GOTO <label>

<structured statement) ::= <repetitive statement>  ¦
                           <conditional statement> ¦
                           <compound statement>    ¦
                           <with statement>

<compound statement>   ::= BEGIN <statement> {;<statement>} END

<conditional statement>::= <case statement>  ¦
                            <if statement>

<if statement> ::= IF < expression) THEN <statement> ELSE <statement> ¦
                   IF <expression> THEN <statement>

<case statement>       ::= CASE <expression> OF
                           <case list> {,<case list>}
                           {ELSE <statement>}
                           END

<case list>            ::= <label list> : <statement>  ¦
                           <empty>

<label list>           ::= <case label> {,<case label>}

<repetitive statement> ::= <repeat statement>   ¦
                           <while statement>    ¦
                           <for statement>

<while statement>      ::= WHILE <expression> DO <statement>

<repeat statement>     ::= REPEAT <statement> {,<statement>} UNTIL
                           <expression>

<for statement>        ::= FOR <ctrlvar> := <for list> DO <statement>

<for list>             ::= <expression> DOWNTO <expression> ¦
                           <expression> TO <expression>

<ctrlvar>              ::= <variable>

<with statement>       ::= WITH <record variable list> DO <statement>

<record variable list> ::= <record variable> {,<record variable>}

<procedure declaration>::= EXTERNAL <procedure heading>  ¦
                           <procedure heading> <block>

<block>                ::= <label declaration part>
                           <constant definition part>
                           <type definition part>
                           <variable declaration part>
                           <profunc declaration part>
                           <statement part>
```

```
<procedure heading>    ::= PROCEDURE <identifier> <parmlist>; ¦
                           PROCEDURE <identifier>;
                           PROCEDURE INTERRUPT [<constant>];

<parmlist>             ::= (<fparm> {,<fparm>})

<fparm>                ::= <procedure heading>  ¦
                           <function heading>   ¦
                           VAR <parm group>     ¦
                           <parm group>

<parm group>           ::= <identifier> {,<identifier>} :
                              <type identifier)          ¦
                           <identifier> {,<identifier>} :
                              <conformant array>

<conformant array>     ::= ARRAY [ <indxtyp> {,<indxtyp>} ] OF
                           <conarray2>

<conarray2>            ::= <type identifier>    ¦
                           <conformant array>

<indxtyp>              ::= <identifier> .. <identifier> : <ordtypid>

<ordtypid>             ::= <scalar type identifier>    ¦
                           <subrange type identifier>

<label declaration part>  ::= <empty>   ¦
                              LABEL <label> {,<label>};

<constant definition part>::=<empty>    ¦

                             CONST
                                <constant definition>
                                {;<constant definition>};

<type definition part>    ::= <empty>   ¦
                              TYPE
                                 <type definition>
                                 {;<type definition>};

<variable declaration part> :: = <empty>   ¦
                                 VAR
                                    <variable declaration>

<indxtyp>    ::= <identifier> .. <identifier> : <ordtypid>

<ordtypid>   ::= <scalar type identifier>  ¦
                 <subrange type identifier>

<label declaration part>  ::= <empty>   ¦
                              LABEL <label> {,<label>};

<constant definition part> ::= <empty>   ¦
                               CONST
                                  <constant definition>
                                  {;<constant definition>};

<type defintion part>      ::= <empty>   ¦
                               TYPE
                                  <type definition>
                                  {;<type definition>};
```

```
<variable declaratian part> ::= <empty>   ¦
                                VAR
                                   <variable declaration>
                                   {;<variable declaration >};

<profunc part>              ::= {<proc or func>;}

<proc or func>              ::= <procedure declaration>  ¦
                                <function declaration>

<statement part>            ::= <compound statement>

<function decl>             ::= EXTERNAL <function heading> ¦
                                <function heading> <block>

<function heading> ::= FUNCTION <identifier><parmlist>:<result type>; ¦
                       FUNCTION <identifier>:<result type>;

<result type>      ::= <type identifier>

<readcall> ::= <read or readln> {( {<filevar> ,} {<varlist>} )}

<read or readln>            ::= READ ¦ READLN

<filevar>                   ::= <variable>

<varlist>                   ::= <variable> {,<variable>}

<writecall> ::= <write or writeln> {( {<filevar> ,} {exprlist} )}

<write or writeln>          ::= WRITE  ¦  WRITELN

<exprlist>                  ::= <wexpr> {,<wexpr>}

<wexpr>     ::= <expression> {:<width expr> {:<dec expr>}}

<width expr>                ::= <expression>

<dec expr>                  ::= <expression>

<program>                ::= <program heading> <block>.  ¦
                             <module heading>
                             <label declaration part>
                             <constant definition part>
                             <type definition part>
                             <variable declaration part>
                             <profunc declaration part>
                             MODEND.

<program heading> ::= PROGRAM <identifier (<prog parms>);

<module heading>  ::= MODULE <identifier>;

<prog parms>      ::= <identifier> {,<identifier>}
```

ANHANG B: RESERVIERTE WÖRTER

Im Folgenden die in ATARI PASCAL reservierten Worte:

AND	DOWNTO	GOTO	NOT	RECORD	UNTIL
ARRAY	ELSE	IF	OF	REPEAT	VAR
BEGIN	END	IN	OR	SET	WHILE
CONST	FILE	LABEL	PACKED	THEN	WITH
CASE	FOR	MOD	PROCEDURE	TO	
DO	FUNCTION	NIL	PROGRAM	TYPE	

ATARI PASCAL verfügt darüber hinaus über folgende reservierte Worte:

ABSOLUTE   EXTERNAL   PREDEFINED

ANHANG C: FEHLERMELDUNGEN

Recursion stack overflow: Die Größe des Auswertungsstacks kollidiert mit der Symboltabelle. Versuchen Sie, die Größe der Symbol tabelle zu reduzieren und Ausdrücke zu vereinfachen.

1: Error is simple Type
Selbsterklärend.

2: Identifier expected
Selbsterklärend.

3: 'PROGRAM' expected
Selbsterklärend.

4: ')' expected
Selbsterklärend.

5: ':' expected
unter Umständen wird '=' in einer Variablendeklaration verwendet.

6: Illegal symbol (possibly missing ';' on line above)
Das aufgetretene Symbol ist an dieser Stelle in der Syntax unzulässig.

7: Error in parameter list
Syntaktischer Fehler in der Parameterlistendeklaration.

8: 'OF' expected
Selbserklärend.

9: '(' expected
Selbsterklärend.

10: Error in type
Syntaktischer Fehler in der TYPE-Deklaration.

11: '[' expected
Selbsterklärend.

12: ']' expected
Selbsterklärend.

13: 'END' expected
Alle Prozeduren, Funktionen und Blöcke müssen durch 'END' abgeschlossen werden. Suchen Sie nach unpassenden BEGIN/END Folgen.

14: ';' expected (possibly on line above)
Eine Trennung der Statements durch ';' ist erforderlich.

15: Integer expected
Selbsterklärend.

16: '=' expected
Unter Umständen ':' in einer TYPE oder CONST-Deklaration verwendet.

17: BEGIN expected
Selbsterklärend.

18: Error in declaration part
Typisch ist ein illegaler Rückzug auf einen Typ einer Pointerdeklaration.

19: Error in <field-list>
Syntaktischer Fehler in einer Recorddeklaration.

20: '.' expected
Selbsterklärend.

21: '*' expected
Selbsterklärend.

50: Error in constant
Syntaktischer Fehler in einer Literalkonstanten.

51: ':=' expected
Selbsterklärend.

52: 'THEN' expected
Selbsterklärend.

53: 'UNTIL' expected
Kann durch unpassende BEGIN/END-Folgen entstehen.

54: 'DO' expected
Syntaktische Fehler.

55: 'TO' or 'DOWNTO' expected in FOR statement
Selbsterklärend.

56: 'IF' expected
Selbsterklärend.

57: 'FILE' expected
Unter Umständen ein Fehler in einer Type-Deklaration

58: Error in <factor> (bad expression)
Syntaktischer Fehler in einem Ausdruck auf der Faktorebene.

59: Error in variable
Syntaktischer Fehler in einem Ausdruck auf der Variablenebene.

99: MODEND expected
Jedes Modul muß mit MODEND enden.

101: Identifier declared twice
Der Name steht bereits in der sichtbaren Symboltabelle.

102: Low bound exceeds high bound
Für Unterbereiche muß die Untergrenze kleiner oder gleich der Obergrenze sein.

103: Identifier is not of the appropriate class
Ein Variablenname, der als Typ benutzt wird oder ein Typ, der als Variable benutzt wird usw., kann diesen Fehler verursachen.

104: Undeclared identifier
Der spezifizierte Bezeichner ist nicht in der sichtbaren Symboltabelle.

105: Sign not allowed
Vorzeichen sind in nicht-Integer oder nicht-Realkonstanten verboten.

106: Number expected
Dieser Fehler kann oft durch eine totale Verwirrung des Compilers durch einen Ausdruck entstehen, wenn er auf Zahlen überprüft, nachdem er alle anderen Möglichkeiten ausgeschöpft hat.

107: Incompatible subrange types
Z. B. 'A'..'Z' nicht kompatibel zu 'Ø'..'9'.

108: File not allowed here
Vergleiche und Zuordnungen von Files sind nicht zugelassen.

109: Type must not be real
Selbsterklärend.

110: <tagfield> type must be scalar or subrange
Selbsterklärend.

111: Incompatible with <tagfield> part
Die Auswahlvariable in einem varianten CASE-Record ist nicht kompatibel zu dem <tagfield>-Typ.

112: Index type must not be real
Ein Array darf mit reellen Dimensionen deklariert werden.

113: Index must be scalar or subrange
Selbsterklärend.

114: Base type must not be real
Der Basistyp eines SETs muß ein Skalar- oder Unterbereichstyp sein.

115: Base type must be scalar or subrange
Selbsterklärend.

116: Error in type of standard procedure parameter
Selbsterklärend.

117: Unsatisfied forward reference
Ein im Voraus deklarierter Pointer ist nicht definiert worden.

118: Forward reference type identifier in variable declaration
Sie haben versucht, eine Variable als Pointer eines noch undeklarierten Typs zu definieren.

119: Respicified parameters not OK for a forward declared procedure
Selbsterklärend.

12Ø: Function result type must be scalar, subrange or pointer
Eine Funktion wurde deklariert, deren Ergebnis vom Typ String oder einem anderen nicht skalaren Typ ist. Dies ist nicht erlaubt.

121: File value parameter not allowed
Files müssen als VAR Parameter übergeben werden.

122: A forward declared function's result can't be respecified.
Ein im Voraus deklarierter Ergebnistyp kann nicht erneut spezifiziert werden.

123: Missing result type in function declaration
Selbsterklärend.

125: Error in type of standard procedure parameter
Dieser Fehler wird oft dadurch verursacht, daß die Parameter für die eingebauten Prozeduren nicht in der richtigen Reihenfolge angegeben werden, oder man versucht hat, Pointer oder Aufzählungstypen usw. ein- oder auszugeben.

126: Number of parameters does not agree with declaration
Selbsterklärend.

127: Illegal parameter substitution
Der Typ des Parameters paßt nicht exakt zum korrespondierenden formalen Parameter.

128: Result type does not agree with declaration
Wenn Typen einem Funktionsergebnis zugeordnet werden, müssen die Typen kompatibel sein.

129: Type conflict of operands
Selbsterklärend.

13Ø: Expression is not of the set type
Selbsterklärend.

131: Test on equality allowed only
SETs dürfen nur auf Gleichheit getestet werden.

133: File comparison not allowed
Filekontrollblöcke dürfen nicht getestet werden, da sie Mehrfachfelder enthalten, die dem Benutzer nicht zugänglich sind.

134: Illegal type or operand(s)
Die Operanden passen nicht zu dem verwendeten Operator.

135: Type of operand must be Boolean
Die Operanden passen nicht zu dem verwendeten Operator.

136: Set element type must be scalar or subrange
Selbsterklärend.

137: Set element types must be compatible
Selbsterklärend.

138: Type of variable is not array
In einer nicht-Arrayvariablen wurde ein Index verwendet.

139: Index is not compatible with declaration
Auf ein Array wurde mit einem unpassenden Indextyp zugegriffen.

140: Type of variable is not record
Auf eine Variable, die nicht vom Typ Record ist, wurde mit '.' oder mit dem 'WITH'-Statement zugegriffen.

141: Type of variable must be file or pointer
Tritt auf, wenn ein '^' auf eine Variable folgt, die nicht vom Typ File oder Pointer ist.

142: Illegal parameter solution
Selbsterklärend.

143: Illegal type of loop control variable
Schleifenkontrollvariable dürfen nur lokale nichtreale Skalartypen sein.

144: Illegal type of expression
Der Ausdruck, der die Auswahl in einem CASE-Statement durchführt, darf nur ein nichtrealer Skalartyp sein.

145: Type conflict
Caselabel und auswählender Ausdruck sind verschiedenen Typs.

146: Assignment of files not allowed
Selbsterklärend.

147: Label type incompatible with selecting expression
Caselabel und auswählender Ausdruck sind verschiedenen Typs.

148: Subrange bounds must be scalar
Selbsterklärend.

149: Index type must be integer
Selbsterklärend.

150: Assignment to standard function is not allowed
Selbsterklärend.

151: Assignment to formal function is not allowed
Selbsterklärend.

152: No such field in this record
Selbsterklärend.

153: Type error in read
Selbsterklärend.

154: Actual parameter must be a variable
Es wurde versucht, einen Ausdruck als einen VAR Parameter zu übergeben.

155: Control variable cannot be formal or non-local
Die Kontrollvariable einer FOR-Schleife muß lokal sein.

156: Multidefined case label
Selbsterklärend.

157: Too many cases in case statement
Tritt auf, wenn die für das Casestatement erzeugte Sprungtabelle ihre Grenzen überschreitet.

158: No such variant in this record
Selbsterklärend.

159: Real or string tagfields not allowed
Selbsterklärend.

160: Previous declaration was not forward

161: Again forward declared

162: Parameter size must be constant

163: Missing variant in declaration
Es wurde versucht, NEW/DISPOSE auf eine nicht existierende Variante anzuwenden.

164: Substitution of standard procedure/function not allowed

165: Multidefined label
Mehrere Statements tragen das gleiche Label.

166: Multideclared label
Das gleich Label wurde mehrfach deklariert.

167: Undeclared label
Ein verwendetes Label wurde nicht deklariert.

168: Undefined label
Ein deklariertes Label wurde nicht verwendet.

169: Error in base set

170: Value parameter expected

171: Standard file was re-declared

172: Undeclared external file

174: Pascal function or procedure expected
Selbsterklärend.

183: External declaration not allowed at this nesting level
Selbsterklärend.

187: Attempt to open library unsuccessful
Selbsterklärend.

191: No private files
Files dürfen nur in dem Abschnitt der globalen Variablen eines Programms oder Moduls deklariert werden, da sie statisch angeordnet werden müssen.

193: Not enough room for this operation
Selbsterklärend.

194: Comment must appear at top of the program

201: Error in real number - digit expected
Selbsterklärend.

202: String constant must not exceed source line

203: Integer constant exceeds range
Integerkonstanten sind auf den Bereich -32768..32767 beschränkt.

250: Too many scopes of nested identifiers
Zur Compilierzeit ist die Verschachtelungstiefe auf 15 Ebenen beschränkt. Das schließt WITH und Prozedurverschachtelungen ein.

251: Too many nested procedures or functions
Zur Ausführungszeit ist die Verschachtelungstiefe auf 15 Ebenen beschränkt.

253: Procedure too long
Die Erzeugung eines Prozedurcodes hat die Größe des internen Prozedurbuffers überschritten. Verkleinern Sie die Prozedur und versuchen Sie es erneut.

259: Expression too complicated
Ihr Ausdruck ist zu kompliziert (z. B. zu viele rekursive Aufrufe für die Compilierung notwendig). Vereinfachen Sie den Ausdruck durch den Gebrauch zeitweiliger Variabler.

397: Too many FOR or WITH statements in a procedure
In einer einzelnen Prozedur sind nur 16 FOR und/oder WITH Statements erlaubt (im rekursiven Modus).

400: Illegal character in text
Außerhalb eines in Anführungszeichen gesetzten Strings wurde ein nicht zu Pascal gehöriges Sonderzeichen gefunden.

4Ø1: Unexpected end of input
'END.' wurde vor der Rückkehr auf die äußere Ebene angetroffen.

4Ø2: Error in writing code file, not enough room
Selbsterklärend.

4Ø3: Error in reading include file
Selbsterklärend.

4Ø4: Error in writing list file, not enough room
Selbsterklärend.

4Ø5: Call not allowed in separate procedure
Selbsterklärend.

4Ø6: Include file not legal
Selbsterklärend.

4Ø7: Symbol table overflow

497: Error in closing code file
Beim Schließen des ERL.-Files trat ein Fehler auf. Schaffen Sie auf der Zieldiskette mehr Platz und versuchen Sie es erneut.

5ØØ: Eine nichtstandardmäßige Operation wurde benutzt während die T+ oder W+ Optionen aktiviert waren. Dieser Fehler ist nicht schwerwiegend und die Meldung dient nur zur Information.

ANHANG D: ATARI PASCAL FILEEIN- UND AUSGABE

Die Abschnitte in diesem Anhang beschreiben ATARI Pascal-files und ihren Gebrauch. Da das Arbeiten mit Beispielen den effektivsten Weg der Beschreibung dieser Konzepte darstellt, wurden für jeden Bereich der Filebehandlung Programmbeispiele aufgenommen.

- Der erste Abschnitt beschreibt die benutzten Ausdrücke wie z. B. 'File', 'Fenstervariable' und 'TEXT'.

- Der zweite Abschnitt zeigt mit Beispielen den Gebrauch aller Fileoperationen. Diese schließen ASSIGN, REWRITE, RESET, Zugriffsoperationen auf sequentielle Files, CLOSE, usw. ein.

- Der dritte Abschnitt beschreibt Pascal TEXT Files. Beispielprogramme zeigen den Gebrauch der eingebauten booleschen Funktionen EOLN und EOF, READLN, WRITELN, formatiertes I/O und die Ansteuerung des Druckers.

- Der vierte Abschnitt beschreibt den Gebrauch seltener benutzter Fileoperationen.

1. DEFINTIONEN

Die hier eingeschlossenen Ausdrücke und Definitionen sind für die logische Behandlung des Filekonzepts während des Durchlesens angeordnet.

FILE

Ein File besteht aus Daten, die als logische, gleich große Elemente angeordnet sind, einem endlosen Array sehr ähnlich, auf das mit einem Zeiger zugegriffen wird. Größe und Anordnung der Daten wird durch Ihr Programm kontrolliert. Ein File wird i.A. auf einem sekundären Speichermedium abgelegt. Diese Dokumentation nimmt als sekundäres Speichermedium eine Diskette an. Sie können ein File beschreiben oder auslesen, indem Sie die durch ATARI PASCAL angebotenen Fileoperationen benutzen. Auf die Daten im File kann sequentiell (auf Record 1 wird vor Record 2 zugegriffen, auf Record 2 wird vor Record 3 zugegriffen usw.) oder direkt zugegriffen werden.

FILENAME

Filename ist der Name des Files auf der Diskette. Es ist der Name,der im Directorylisting des Speichermediums auftaucht. In ATARI PASCAL wird der Filename im Programm durch einen String dargestellt (eine dynamisch Folge von ASCII Zeichen). Z. B. ist 'D2:TEST.PAS' das Literalstringformat des Filenames für das File, das auf Laufwerk 'D2' mit dem Namen 'TEST' und der Erweiterung '.PAS' liegt.

TYPE

Der Typ des Files definiert Größe und Format der jeweiligen Fileelemente, den kleinsten Elementen, auf die zuge-

griffen werden kann. Zum Beispiel kann man sich ein File des Typs Integer (2*8-bit-bytes) folgendermaßen vorstellen:

```
+--------+--------+--------+--------+--------+--------+
¦00001000¦00000000¦00100001¦00000000¦00000001¦00000000¦
+--------+--------+--------+--------+--------+--------+
¦     Record 1    ¦     Record 2    ¦     Record 3    ¦
```

Dieses File enthält die Integerzahlen 8, 33 und 1 (in diesem Beispiel umgekehrt abgespeichert). Das kleinste aufrufbare Element ist zwei Bytes lang. Beachten Sie die Erklärungen für typfreie oder Bytefiles, wenn Sie dieses File nicht als Integerfile behandeln wollen. Files können alle skalaren Pascalstandardtypen aufweisen: BOOLEAN, INTEGER, CHAR oder REAL. Sie können aber auch die strukturierten Typen STRING, Array oder Record aufweisen. Der vordefinierte Typ TEXT wird für ASCII-Files benutzt. Textfiles sind FILEs OF CHAR ähnlich, sind aber in Zeilen unterteilt. Außerdem werden in sie geschriebene Zahlen nach ASCII umgesetzt (und können formatiert sein) und aus ihnen gelesene Zahlen werden binär codiert. Eine Zeile ist i. A. eine Folge von Zeichen, die durch ein End-of-Line-Zeichen, normalerweise das CR/LF-Zeichen, abgeschlossen wird. Im Unterschied zu einem FILE OF CHAR akzeptiert ein Textfile auch PACKED ARRAYs[1..N] OF CHAR, ARRAYs[1..N] OF CHAR und STRINGS als Daten (das Schreiben eines ungepackten Arrays ist kein ISO-Standard). Ein boolescher Wert wird während des Schreibens in die ASCII-Folge 'true' oder 'false' übersetzt, die Umkehrung dieser Operation tritt nicht ein. Schlagen Sie für weitere Informationen über typbehaftete und Textfiles im Abschnitt Fileoperationen nach.

Ein nicht dem ISO-Standard gemäßes Konzept der Filebetrachtung sind die typfreien Files. Dieses Konzept wird für die s chne lle Blockein- und ausgabe benutzt (ganze Sektoren werden geschrieben oder gelesen), ohne daß die Art der im File enthaltenen Date n beachtet wird.

FILE INFORMATION BLOCK (FIB)

Der FIB-Block enthält die für die Laufzeitroutinen notwendigen Informationen, um auf das Diskettenfile Operationen auszuführen. Der Filename, der Filetyp, Zugriffsart (Lesen oder Schreiben), End-of-File und End-of-Line-Flags und ein Diskettenbuffer von der Größe eines Diskettenblocks sind einige der Informationen, die im FIB gehalten werden.

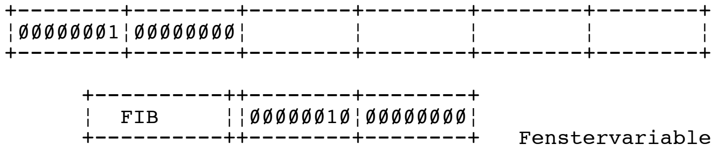
WINDOW VARIABLE oder WINDOW POINTER

Die Fenstervariable ist ein Buffer von der Größe eines Fileelements und wird in ATARI PASCAL direkt hinter dem FIB angeordnet. Man kann sich vorstellen, daß sie sich am File entlangbewegt und als Fenster für das Element handelt, das beschrieben oder gelesen werden soll. Aus diesem Grund wird sie als Zeiger auf das aufgerufene File betrachtet. Sie wird mit 'F1' bezeichnet, 'F' ist der Name der Filevariablen. Um das File zu lesen, wird das aufrufbare Element in die Fenstervariable gebracht. Um das File zu be-

schreiben, muß das Datum von der Fenstervariablen in das File gebracht werden.

FILE VARIABLE

Die Filevariable besteht aus einem FIB und einer Fenstervariablen. Sie ist das aktuelle Datum, das durch den Compiler abgelegt wird, und auf das in einem Pascalprogramm Bezug genommen wird. Ein Beispiel wird deutlicher machen, was Filevariable, FIB und Fenstervariable sind. Das Statement 'VAR F:FILE OF INTEGER;' veranlaßt den Compiler, eine Filevariable mit ihrem eigenen FIB zusammen mit einer Fenstervariablen (2 Bytes), die eine 15-Bit Zahl aufnehmen kann, im Datenbereich zu erzeugen. Die Fenstervariable wird mit F^ bezeichnet. Nehmen Sie an, 'I' sei eine Integerzahl mit dem Wert 2 im gleichen Programm. Nehmen Sie weiter an, daß das File, wie unten gezeigt, bereits im ersten Element den Wert 1 enthält:



Um den Inhalt von I in das File zu schreiben, muß die Fenstervariable 2 enthalten (F^ :=I bringt den Inhalt von I in die Fenstervariable) und die Fenstervariable über das zweite Element des Files 'geschoben' werden. Mit dem Kommando PUT(F), beschrieben im Operationsabschnitt, wird die Zahl 2 in das File geschrieben.

2. GRUNDLEGENDE FILEOPERATIONEN

Beispielprogramm und Erklärungen demonstrieren den Gebrauch der Fileoperationen in ATARI PASCAL. Sie werden sehen, wie Sie Files eröffnen, erzeugen, lesen, schreiben, löschen und schließen können. Erklärt wird auch der Gebrauch typbehafteter und Textfiles, die Filestatusfunktionen IORESULT, EOF und EOLN und die Zuweisung einer Fenstervariablen.

Abb. D-1 zeigt ein Programm mit dem Namen WRITE_READ_-FILE_DEMO, das ein typbehaftetes File auf Diskette erzeugt, Daten in das File schreibt, das File schließt, das File erneut eröffnet und die Daten zurückliest. Zur Ausführung werden die Prozeduren ASSIGN, REWRITE, RESET, IO-RESULT, PUT, GET und CLOSE benutzt. WRITE wird benutzt, um die Ergebnisse auf dem Bildschirm anzuzeigen. Die Ausgabe wird in WRITEFILE, die Eingabe in READFILE erledigt. Erzeugen, Eröffnen und Schließen des Files werden im Hauptblock des Programms durchgeführt.

Die WRITELN-Statements in den Zeilen 37, 43, 46 und 49 schreiben die an sie übergebenen Strings auf das implizit vordefinierte Ausgabefile (die Konsole). Diese Prozedur wird zusammen mit READLN später im Zusammenhang mit Textfiles behandelt.

Beachten Sie zuerst die Form, in der OUTFILE deklariert wird. Nach der Deklaration ist es vom Typ CHFILE, der in der Typdeklaration als FILE OF CHAR (Zeilen 3 und 4) definiert ist. Dies geschieht, weil das File als Parameter an die Routinen WRITEFILE und READFILE übergeben wird, und eine Parameterliste keinen neuen Typ deklarieren kann. Zum Beispiel ist die folgende Parameterliste in Pascal nicht zulässig, da nur Typbezeichner in der Parameterliste zugelassen sind:

```
PROCEDURE WRITEFILE ( VAR F : FILE OF CHAR) ;
```

```
1     Ø     PROGRAM WRITE_READ_FILE_DEMO;
2     Ø
3     Ø     TYPE
4     1       CHFILE = FILE OF CHAR;
5     1     VAR
6     1       OUTFILE   : CHFILE;
7     1       RESULT    : INTEGER;
8     1       FILENAME  :  STRING [16];
9     1
1Ø    1       PROCEDURE WRITEFILE(VAR F : CHFILE);
11    1       VAR CH : CHAR;
12    2       BEGIN
13    2         FOR CH := 'Ø' TO '9' DO
14    2           BEGIN
15    3             F^:= CH;(*CHR(I + ORD('Ø'));*)
16    3             PUT(F)
17    3           END;
18    2       END;
19    1
2Ø    1       PROCEDURE READFILE(VAR F : CHFILE);
21    1       VAR I  : INTEGER;
22    2           CH : CHAR;
23    2       BEGIN
24    2         FOR I :=Ø TO 9 DO
25    2           BEGIN
26    3             CH:=F^;
27    3             GET (F);
28    3             WRITELN(CH);
29    3           END;
3Ø    2       END;
31    1
32    1     BEGIN
33    1       FILENAME := 'TEST.DAT';
34    1       ASSIGN(OUTFILE,FILENAME);
35    1       REWRITE(OUTFILE);
36    1       IF IORESULT <> Ø THEN
37    1         WRITELN('Error creating ' ,FILENAME)
38    1       ELSE
39    1         BEGIN
4Ø    2           WRITEFILE(OUTFILE);
41    2           CLOSE(OUTFILE,RESULT);
42    2           IF RESULT <> Ø THEN
43    2             WRITELN('Error closing ',FILENAME)
44    2           ELSE
45    2             BEGIN
46    3               WRITELN('Successful close of ',FILENAME);
47    3               RESET(OUTFILE);
48    3               IF IORESULT <> Ø THEN
```

```
49    3                WRITELN('Cannot open ', FILENAME)
5Ø    3              ELSE
51    3                READFILE(OUTFILE)
52    3            END;
53    2          END;
54    1    END.
```

Abbildung D-1 Fileein- und ausgabe

PROCEDURE ASSIGN(VAR F : FILE VARIABLE;STR : STRING);

Zweck: Zuordnung einer Filevariablen F zu einem in STR genannten, externen Diskettenfile.

ASSIGN ist die erste auszuführende Fileoperation in Zeile 34. Diese Prozedur verbindet eine Filevariable (OUTFILE) mit einem externen Diskettenfile, das in FILENAME genannt wird (hier 'TEST.DAT'). Der an ASSIGN übergebene String wird im FIB abgelegt und der Name ausgewertet. Nach der Ausführung der ASSIGN Prozedur ist die an ASSIGN übergebene Filevariable dem im Namensparameter genannten Diskettenfile zugeordnet, bis auf die Filevariable eine erneute ASSIGN Prozedur angewendet wird.

PROCEDURE REWRITE(VAR F : FILE VARIABLE);

Zweck: Erzeugung eines Diskettenfiles mit dem im FIB angegebenen Namen (der entweder durch ein vorhergegangenes ASSIGN Statement gefüllt wurde oder leer ist; falls er leer ist, wird ein zeitweiliges File erzeugt).

Die REWRITE Prozedur wird in Zeile 35, Abb. D-1 aufgerufen. Die Ausführung dieser Prozedur veranlaßt die Erzeugung eines Files dessen Name im FIB von F enthalten ist. Da Files gleichen Namens gelöscht werden, wenden Sie REWRITE niemals auf Files an, die noch brauchbare Daten enthalten. In diesem Beispiel wird das Diskettenfile 'TEST.DAT' genannt und auf der implizit vordefinierten Diskette abgelegt (in dem an ASSIGN übergebenen Filenamen wurde keine andere Diskette spezifiziert).

Falls kein ASSIGN erfolgt ist, ist das Namensfeld in FIB leer und es wird ein zeitweiliges File mit dem Namen 'PASTMPØØ.$$$' angelegt. Zeitweilige Files werden normalerweise als 'Notizblock' und für Daten, die nach der Programmausführung nicht mehr gebraucht werden, benutzt. Die Ziffern an den letzten beiden Stellen des Namens werden benutzt, um jedem zeitweiligen File einen eindeutigen Namen zu geben.

Die EOF-Funktion und die EOLN-Funktion ergeben 'true', da OUTFILE ein Ausgabefile ist. OUTFILE ist nur für sequentielles Schreiben geöffnet und bereit, Daten in sein erstes Element aufzunehmen. Falls die Operation nicht erfolgreich ist, ergibt die IORESULT-Funktion in diesem Fall einen Wert ungleich null (s. Zeile 36).

FUNCTION IORESULT : INTEGER;

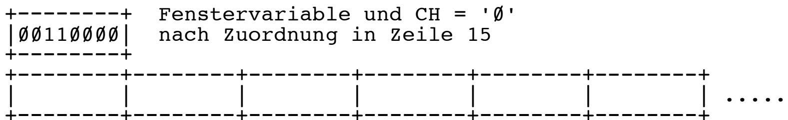
Zweck: Ausgabe eines Integerwertes, der den Status einer Fileoperation kennzeichnet.

Der Wert dieser Funktion wird nach jeder Ein- oder Ausgabeoperation gesetzt und kann zu jeder Zeit überprüft werden. Beachten Sie, daß in Abb. D-1 die Funktion nach jeder Fileoperation in den Zeilen 36, 42 und 46 aufgerufen wird. Sie wird hier benutzt, um das Programm zu stoppen, falls eine der Fileoperationen nicht wie vorgesehen gearbeitet hat. Beachten Sie, daß 'WRITE(IORESULT)' nicht möglich ist, da IORESULT nach jeder I/O-Operation auf Ø gesetzt wird. Die Bedeutung der für IORESULT möglichen Werte finden Sie in Kapitel 3.

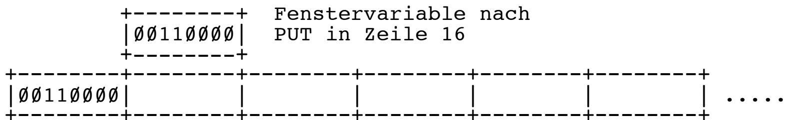
PROCEDURE PUT(VAR F : FILE VARIABLE);

Zweck: Übertragung des Inhalts der Fenstervariablen in das nächstmögliche Element des Files.

Die Prozedur WRITEFILE, Beginn Zeile 9 Abb. D-1, schreibt die Zeichen 'Ø' bis '9' in das File 'TEST.DAT'. Die PUT Prozedur veranlaßt, daß die Daten in das File geschrieben werden. Vor jedem Aufruf der PUT Prozedur erfolgt wie in Zeile 15 eine Zuordnung an die Fenstervariable. Im Folgenden ein Schema des Ablaufs:



File vor der Ausführung des ersten PUT Statements.



File nach der Ausführung des ersten PUT in der FOR-Schleife (Abb. D-1 Zeile 13 - 17).

PROCEDURE WRITE;
PROCEDURE WRITE(expression,...,expression);
PROCEDURE WRITE (VAR F:FILE VARIABLE,expression,...,expression);

Zweck: Kurzform für 'F^:= Datum;PUT(F);'. Außerdem werden Zahlen nach ASCII umgesetzt, wenn Fein Textfile ist.

Expression schließt die Inhalte von Variablen, Strings, Arr ayelementen, Konstante n und Ausdrücken ein. Falls keine Filevariable spezifiziert ist, wird das implizit vordefinierte Ausgabefile angenommen. Die WRITE Prozedur führt auf das File die gleichen Operationen aus wie die Zeilen 15 und 16. Sie bewirkt eine Zuweisung, auf die eine PUT Operation folgt und stellt nur eine Kurz-

form dar. GET und PUT wurden eingeschlossen, da der ISO Standard sie erfordert und in einige Pascalversionen, z. B. UCSD Pascal, WRITE nur auf Textfiles angewendet werden kann.

PROCEDURE CLOSE(VAR F : FILE VARIABLE; RESULT : INTEGER);

Zweck: Räumen des Buffers in dem zu F gehörigen FIB, so daß alle Daten auf die Diskette geschrieben werden.

Das nächste nach der Rückkehr von WRITEFILE auszuführenden Statement ist Zeile 41, in der das File geschlossen wird. CLOSE muß ausgeführt werden, um sicherzugehen, daß die nach 'TEST.DAT' geschriebenen Daten wirklich auf Diskette gespeichert werden. Bis zu diesem Zeitpunkt wurden die Daten in den im Speicher befindlichen Buffer geschrieben und müssen nun durch Räumen des Buffer auf die Diskette gebracht werden. RESULT ist der vom Betriebssystem zurückgegebenen Wert, der anzeigt, ob die CLOSE Operation fehlerfrei abgelaufen ist. Es wurde als Parameter aufgenommen, um Kompatibilität mit früheren Compilerversionen aufrechtzuerhalten. In diesem Programm bedeutet ein Wert ungleich null, daß während des Schließens ein Fehler aufgetreten ist. Null kennzeichnet ein erfolgreiches Schließen.

PROCEDURE RESET(VAR F : FILE VARIABLE);

Zweck: Ein existierendes File wird zum Lesen geöffnet. Die Fenstervariable wird an den Anfang des Files gesetzt.

Nach der Überprüfung von RESULT wird in Zeile 47 die Prozedur RESET aufgerufen. RESET öffnet ein existierendes File zum Lesen und setzt die Fenstervariable an den Anfang des Files. F wird dem ersten Element von F zugeordnet. Falls F bereits eröffnet war, führt RESET zu einem CLOSE. EOF und EOLN ergeben 'false'. Wenn RESET auf ein nichtexistierendes File angewendet wird, ergibt IORESULT einen Wert ungleich null. Jeder andere Wert zeigt einen Erfolg an. Im Beispielprogramm wird OUTFILE durch ein RESET geöffnet, so daß es gelesen werden kann. Im Folgenden finden Sie eine Darstellung des Files und der Fenstervariablen nach der Ausführung des RESETs in Zeile 47. Beachten Sie, daß RESET bei Files wie OUTFILE, die nach ihrem Typ nicht auf die Computerconsole führen, ein initiierendes GET impliziert, das das erste Element des Files (hier den ASCII-Wert für Ø) in die Fenstervariable einliest.

```
+--------+  Fenstervariable (OUTFILE^)
|ØØ11ØØØØ|  nach RESET in Zeile 47
+--------+
+--------+--------+--------+--------+--------+--------+
|ØØ11ØØØØ|ØØ11ØØØ1|ØØ11ØØ1Ø|ØØ11ØØ11|ØØ11Ø1ØØ|ØØ11Ø1Ø1| .....
+--------+--------+--------+--------+--------+--------+
```

Das initiierende GET wird nicht auf typfreie oder Consolenfiles angewendet. Sie würden vor jeder Programmausführung ein Zeichen eingeben müssen, da die GET Prozedur auf ein Zeichen wartet.

PROCEDURE GET(VAR F : FILE VARIABLES);

Zweck: Das momentan aufrufbare Element in die Fenstervariable bringen und die Fenstervariable vorwärtsbewegen.

Wenn die Überprüfung der RESET Prozedur erfolgreich ist, wird in Zeile 51 READFILE aufgerufen. Diese Prozedur liest jedes an sie übergebene Element des Files (hier sind die Elemente Zeichen) und schreibt sie auf den Schirm. READFILE beginnt in Zeile 2Ø. Die Arbeit geschieht in einer FOR-Schleife in den Zeilen 24 bis 29.

Die GET Prozedur schiebt die Fenstervariable ein Element vorwärts und bringt den Inhalt der angezeigten Filestelle in die Fenstervariable. Falls kein nächstes Element existiert, nimmt EOF den Wert 'true' an. Schlagen Sie im Abschnitt 3 über Textfiles nach, um weitere Informationen über GET und Textfiles zu erhalten. Das folgende Diagramm zeigt die Vorgänge innerhalb der FOR-Schleife in den Zeilen 26 und 27 während des ersten Durchlaufs.

```
+--------+  Fenstervariable (OUTFILE)
|ØØ11ØØØØ|  nach Zeile 26
+--------+
+--------+--------+--------+--------+--------+--------+
|ØØ11ØØØØ|ØØ11ØØØ1|ØØ11ØØ1Ø|ØØ11ØØ11|ØØ11Ø1ØØ|ØØ11Ø1Ø1|
+--------+--------+--------+--------+--------+--------+
```

Nach der Ausführung von Zeile 26 enthält CH den ASCII-Wert für 'Ø' (ØØ11ØØØØ). Nach der Ausführung von Zeile 27 ist die Fenstervariable weitergeschoben worden.

```
         +--------+
         |ØØ11ØØØ1|  Fenstervariable nach GET in Zeile 27
         +--------+
+--------+--------+--------+--------+--------+--------+
|ØØ11ØØØØ|ØØ11ØØØ1|ØØ11ØØ1Ø|ØØ11ØØ11|ØØ11Ø1ØØ|ØØ11Ø1Ø1|
+--------+--------+--------+--------+--------+--------+
```

Zeile 28 schreibt den Inhalt von CH auf das implizit vordefinierte Ausgabefile, die Konsole. Die Prozedur READFILE schreibt die Zeichen 'Ø' bis '9' in einer Spalte auf den Schirm. Nach RESET für einen sequentiellen Aufruf ist ein CLOSE nicht notwendig, da das File bereits auf Diskette existiert und nicht verändert worden ist. Falls auf OUTFILE direkt zugegriffen wird, kann ein CLOSE notwendig sein.

PROCEDURE READ(data,data,...,data);
PROCEDURE READ(VAR F : FILE VARIABLE, data,data,...,data);

Zweck: Wenn sie mit Files verwendet wird, die nicht auf die Konsole zugreifen, benutzen Sie 'data := F^; GET(F);' für jedes zu lesende Datum. Falls F nicht angegeben ist, wird das implizit vordefinierte Eingabefile benutzt. Schlagen Sie für Informationen über Umwandlungen im Abschnitt Textfiles nach.

Die READ Prozedur ist das Gleiche,wie eine Zuordnung und ein folgendes GET. Wenn im vorliegenden Beispiel READ anstelle von GET benutzt worden wäre, würde der FOR-Schleifenblock so aussehen:

```
FOR I:=Ø TO 9 DO
  BEGIN
    READ(CH);
    WRITELN(CH);
  END;
```

Das überlesen eines End-of-File Zeichens von der Konsole führt zu einem Systemabsturz.

3. TEXTFILES

Definition

Ein Textfile ist ein File für ASCII-Zeichen, das in Zeilen unterteilt ist. Eine Zeile ist eine Folge von Zeichen, die durch ein nicht druckbares End-of-Line Zeichen beendet wird, normalerweise ein Carriage-Return- und ein Line-Feed-Zeichen. Es ist einem File of Char ähnlich, mit dem Unterschied, daß Zahlen beim Lesen oder Schreiben umgewandelt werden. Außerdem können Strings aus dem Textfile gelesen und Variable vom Typ BOOLEAN, STRING und PACKED ARRAY in das Textfile geschrieben werden. Für Zeichenein- und Ausgabe kann das Textfile über GET und PUT (die keine Zahlenumwandlung vornehmen) erreicht werden, mit READ und WRITE, die früher in diesem Abschnitt definiert wurden und mit READLN und WRITELN, die in Abb. D-2 benutzt und in diesem Abschnitt definiert werden.

Das Format eines Textfiles im Speicher besteht aus einem FIB und einer 1-Byte Fenstervariablen. Auf der Diskette sieht ein Textfile wie in der unten gezeigten Abbildung aus, in der '>' einen Carriage Return, '/' einen Linefeed und '#' ein End-of-File-Zeichen darstellt.

```
+-------------------------------------------------------------------+
Eine Zeile>/Dies ist die nächste Zeile>/Dies ist die letzte Zeile>/#
+-------------------------------------------------------------------+
```

```
FUNCTION EOLN : BOOLEAN;
FUNCTION EOLN(VAR F : TEXT) : BOOLEAN;
```

Zweck: Darstellung des Filestatus durch Übergabe des Wertes 'true', sobald die Fenstervariable über dem End-of-Line-Zeichen steht. Falls kein File angegeben ist, wird das implizit vordefinierte Eingabefile angenommen.

Die Funktion ergibt 'true', wenn durch ein READ das letzte gültige Zeichen einer Zeile aus einem Diskettentextfile gelesen wurde. Wegen der dem READ-Befehl gleichbedeutenden Folge 'CH := F^;GET(F);' (für nicht-Konsolenfiles), wird die Fenstervariable direkt nach dem Lesen des letzten Zeichens über das End-of-Line Zeichen gesetzt. Deswegen ergibt EOLN für nicht-Konsolentextfiles 'true', sobald das letzte Zeichen eingelesen wurde. Außerdem wird ein

Leerzeichen anstelle des End-of-Line Zeichens übergeben. Die oben angegebene Folge wird für Konsolenfiles umgedreht (READ besteht aus einem GET-Aufruf, dem eine Zuweisung an die Fenstervariable folgt). Beim Gebrauch von Konsolenfiles ergibt EOLN 'true', sobald das CR/LF Zeichen gelesen wurde. Bei Diskettentextfiles ergibt EOLN nach dem Lesen des CR/LF Zeichen 'true'. Trotzdem wird ein Leerzeichen mit dem Zeichen übergeben.

```
FUNCTION EOF : BOOLEAN;
FUNCTION EOF(VAR F : TEXT) : BOOLEAN;
```

Zweck: Darstellung des Filezustandes durch Übergabe des Wertes 'true', sobald die Fenstervariable über dem End-of-File-Zeichen steht. Falls kein File angegeben ist, wird das implizit vordefinierte Eingabefile angenommen.

EOF ist eine Funktion, die 'true' ergibt, sobald das End-of-File Zeichen gelesen wird. Es ist EOLN darin ähnlich, daß das letzte gelesene Zeichen bei nicht-Konsolenfiles EOF gleich 'true' setzt. Bei Konsolenfiles ist EOF nur 'true', wenn das End-of-File Zeichen eingegeben wird. Das Überlesen eines End-of-File-Zeichens von der Konsole wird nicht unterstützt (das System kann abstürzen). Überlesen eines End-of-File-Zeichens von Diskette wird nicht unterstützt. Ein Leerzeichen wird übergeben, wenn EOF 'true' geworden ist. Beachten Sie, daß bei nicht-Textfiles EOF u. U. nicht 'true' wird, da die gültigen Daten den letzten Sektor des Files nicht ausfüllen.

Abb. D-2 zeigt ein Programm, das Daten in ein Textfile schreibt und sie zur Darstellung auf dem Ausgabegerät zurück liest. Die Prozedur WRITE-DATA schreibt in das Textfile und die Prozedur READDATA liest die Information, die in dem File gespeichert ist, aus. Das Programm ist in das Hauptprogramm und zwei Prozeduren aufgeteilt, um die Nützlichkeit der Aufteilung des Codes in Blöcke zu zeigen, die bestimmte Aufgaben erfüllen. Diese Methode macht das Programm lesbarer und erleichtert die Fehlersuche.

Das File wird in Zeile 3 deklariert. Beachten Sie, daß die Deklaration nicht 'VAR F:FILE OF TEXT' lautet. TEXT wird als Spezialfall eines FILEs OF CHAR behandelt, FILE OF TEXT ergibt ein FILE OF FILE OF CHAR (unsinnig).

Die Programmausführung beginnt in Zeile 25 mit einem ASSIGN Aufruf. Zeile 25 bis 29 erzeugen ein Textfile namens TEXT.TST auf dem angesprochenen Laufwerk. Falls die Fileerzeugung fehlerfrei läuft, werden in Zeile 31 und 32 die Beispieldaten initialisiert und es folgt ein Aufruf der WRITEDATA Routine. WRITEDATA benutzt die WRITELN Prozedur, die nur bei Textfiles verwendet wird.

```
PROCEDURE WRITE;
PROCEDURE WRITELN;
PROCEDURE WRITELN(expr,expr,...,expr);
PROCEDURE WRITELN(F);
PROCEDURE WRITELN(F,expr,expr,...expr);
```

Zweck: Schaffe die Daten in das File, das mit dem File F verbunden ist, und beende die Ausgabe mit einem End-of-Line-Zeichen. Falls kein File angegeben ist, erfolgt die Ausgabe auf das OUTPUT File. Ein WRITELN ohne jeden Ausdruck schreibt lediglich eine CR/LF. Die WRITE Prozedur entspricht eher einer Umwandlungsprozedur als einem PUT-Ersatz.

Diese Prozedur übergibt die an sie übergebenen Daten an das genannte File und schreibt ein End-of-Line-Zeichen hinter das letzte geschriebene Datum. Falls kein File angegeben ist, werden die Daten auf das implizit vordefinierte Ausgabefile geschrieben. Literal- und benannte Konstante, Integers, Reals, Unterbereiche, Aufzählungstypen, Booleans, Strings und packed Arrays of Char sind als Daten zulässig, aber nicht strukturierte Typen wie z. B. Records. Numerische Daten werden in ASCII umgewandelt, Strings werden als Arrays of char behandelt (das die Länge angebende Byte wird nicht auf das File geschrieben).

Formatierte Ausgabe

In Abb. D-2 besorgen drei Zeilen des Blocks WRITEDATE (9, 1Ø und 11) die eigentliche Ausgabe. Zeile 9 schickt den Inhalt der Stringvariablen S, gefolgt von CR/LF, an das Textfile F. Zeile 1Ø formatiert den Inhalt von I in einem Feld von vier Leerzeichen und sendet diese formatierte Ausgabe zum File F. Der Realliteral in Zeile 11 wird in einem Feld von neun Leerzeichen formatiert, von denen vier rechts vom Dezimalkomma stehen. Diese formatierte Zahl wird in das File F geschrieben. Das Feldformat kann für jeden Datentyp spezifiziert werden. Für alle Typen außer REAL wird nur die Feldlänge, nicht die Anzahl der Dezimalstellen angegeben. Die Daten stehen im Feld rechtsbündig. Falls die Zahl größer ist als 6.5 signifikante Stellen darstellen können, erfolgt die Ausgabe in exponentieller Notation. Das geschieht auch, wenn die Feldlänge zu klein ist, um die Zahl darzustellen.Weitere Informationen finden Sie in einem Pascallehrbuch oder probieren Sie es einfach aus.

Mit den gleichen Ergebnissen konnte der WRITEDATE Block auch so aussehen:

```
WRITELN(F,S);
WRITELN(F,I : 4,45.6789 : 9 : 4);
```

Die Kontrolle wird an das Hauptprogramm zurückgegeben und Zeile 34 ausgeführt. Falls CLOSE fehlerfrei ist, öffnet RESET in Zeile 34 das File F (das immer noch mit TEXT.TST auf der Diskette verbunden ist) und setzt die Fenstervariable auf den Anfang, um das Lesen der Daten vorzubereiten. Auf ein erfolgreiches RESET folgt READDATA, um die in TEXT.TST gespeicherten Informationen zurückzulesen und sie auf dem Bildschirm darzustellen.

```
Statement Nest   Source Statement
1          Ø       PROGRAM TEXT1Ø_DEMO;
2          Ø
3          Ø       VAR F:TEXT;
4          1           I:INTEGER;
5          1           S:STRING;
6          1
7          1         PROCEDURE WRITEDATA;
8          1         BEGIN
9          2           WRITELN(F,S);
1Ø         2           WRITE(F,I:4);
11         2           WRITELN(F,45.6789:9:4);
12         2         END;
13         1
14         1         PROCEDURE READDATA;
15         1         VAR R : REAL;
16         2         BEGIN
17         2           READLN(F,S);
18         2           READ(F,I);
19         2           READ(F,R);
2Ø         2           WRITELN(S);
21         2           WRITELN(I:4,' ',R:9:4);
22         2         END;
23         1
24         1       BEGIN
25         1         ASSIGN(F,'TEXT.TST');
26         1         REWRITE(F);
27         1         IF IORESULT <> Ø THEN
28         1           WRITELN('Error creating')
29         1         ELSE
3Ø         1           BEGIN
31         2             I := 35;
32         2             S := 'THIS IS A STRING';
33         2             WRITEDATA;
34         2             CLOSE(F,I);
35         2             IF IORESULT <> Ø THEN
36         2               WRITELN('Error closing')
37         2             ELSE
38         2               BEGIN
39         3                 RESET(F);
4Ø         3                 IF IORESULT <> Ø THEN
41         3                   WRITELN('Error opening')
42         3                 ELSE
43         3                   READDATA
44         3               END;
45         2           END;
46         1       END.
46         Ø       ----------------------------
46         Ø       Normal end of input reached.
```

Abbildung D-2 Textfiles

PROCEDURE READ;
PROCEDURE READLN;
PROCEDURE READLN(F);
PROCEDURE READ(F,variable,variable,...,variable)

Zweck: Einlesen des zugeordneten Files in die aufgelisteten Variablen. In allen Fällen erfolgt ein Lesen bis ein End-of-Line-Zeichen aufgefunden wird, ungelesene Daten werden übersprungen und das Lesen am Beginn der nächsten Zeile fortgesetzt. READ ist undefiniert, so daß es REALs, BOOLEANs und INTEGERs umwandelt.

READLN besitzt wie WRITELN eine optimale Filevariable und eine beliebige Anzahl Variabler als Parameter, um Daten aus dem File zu empfangen. Falls keine Filevariable spezifiziert ist, wird der Input aus dem implizit vordefinierten Eingabefile, der Tastatur, entnommen. Die Daten in der Parameterliste sind vom gleichen Typ wie die aus dem File zu lesenden. Allerdings wird keine Typüberprüfung vorgenommen, so daß es Ihre Aufgabe ist, eine mit dem Format Ihres Files kompatible Parameterliste aufzubauen. Alle Zahlen werden bei der Eingabe umgewandelt, jedoch geht die Formatierung verloren. Zahlen müssen voneinander und von anderen Datentypen durch ein Leerzeichen oder ein CR/LF getrennt werden.

READLN erkennt das End-of-Line-Zeichen, überträgt es aber nicht. Der Ablauf besteht darin, Daten zu lesen, bis ein End-of-Line-Zeichen erkannt wird und dann die Fenstervariable an den Anfang der nächsten Zeile zu setzen. Die Daten in 'TEXT.TST'würden folgendermaßen aussehen:

```
THIS IS A STRING>/
  35  45.6789>/#
```

Nachdem Sie den String aus der ersten Zeile gelesen haben, müssen Sie, um die Integerzahl 35 zu lesen, READ und nicht READLN benutzen. Falls READLN benutzt würde, würde 35 korrekt gelesen, da das erste Leerzeichen die Nummer beendet. Allerdings würde die Fenstervariable über die Realzahl hinweg an das Fileende gesetzt werden. Falls Sie dann versuchten, die Realzahl zu lesen, würden Sie nur ein EOF erhalten, und sich wundern, was mit Ihrer Realzahl passiert ist, von der Sie wissen, daß sie irgendwo abgelegt wurde.

Strings müssen mit READLN gelesen werden, da sie mit einem End-of-Line-Zeichen beendet werden. Falls das Datum im File 'THIS IS A STRING 35>/' lauten würde, ergäbe sich als Wert für S die ganze Zeile, einschließlich dem ASCII-Wert 35.

Zeilen 2Ø und 21 schreiben die Daten auf die Computerkonsole in dem gleichen Format, wie sie in dem File gespeichert sind.

Nach der Ausführung von READDATA ist das Programm beendet. Ein CLOSE ist nicht notwendig, da die Daten im File 'TEXT.TST' seit dem letzten CLOSE dieses Files nicht verändert worden sind.

Ansteuerung des Druckers

Die Ansteuerung des Druckers ist, wie in Abb. D-3 gezeigt, sehr einfach. Eine Filevariable vom Typ TEXT wird in Zeile 5, Abb. D-3, deklariert. Diese Filevariable wird in Zeile 11 dem Drucker zugeordnet . Der an ASSIGN übergebene Filename 'P:' bedeutet, daß F mit dem Drucker verbunden ist und alle Daten, die auf dieses File geschrieben werden, zum Drucker umzuleiten sind. REWRITE wird aufgerufen, um das Druckerfile zum Schreiben zu öffnen. Beachten Sie, daß ein CLOSE nicht notwendig ist, da alle Daten bereits geschrieben worden sind und der Buffer bereits geräumt wurde. Zeilen 23 und 25 benutzen Standardpascal Formatierungsanweisungen. Zeile 23 schreibt R in ein sieben Zeichen langes Feld, von denen drei Stellen rechts vom Dezimalkomma stehen sollen.

```
Statement   Nest     Source Statement

1            Ø         PROGRAM PRINTER;
2            Ø         (* Schreibe Daten und Text auf den
                          Drucker *)
3            Ø
4            Ø         VAR
5            1           F:TEXT;
6            1           I: INTEGER;
7            1           S:STRING;
8            1           R:REAL;
9            1
1Ø           1         BEGIN
11           1           ASSIGN(F,'P:');
12           1           REWRITE(F);
13           1           IF IORESULT <> Ø THEN
14           1             WRITELN('Error rewriting file')
15           1           ELSE
16           1             BEGIN
17           2               S:='THIS LINE IS A STRING';
18           2               I := 55;
19           2               R := 3.141563;
2Ø           2               WRITE(F,S);
21           2               WRITE(F,I);
22           2               WRITELN(F);
23           2               WRITELN(F,R:7:3);
24           2               WRITELN(F,I,R);
25           2               WRITELN(F,I:4,R:7:3);
26           2               WRITELN(F);
27           2               WRITELN(F,'THIS IS THE END')
28           2             END
29           1         END.
29           Ø         ---------------------------
29           Ø         Normal End of Input reached
```

Abb. D-3 Druckerausgabe und Zahlenformatierung

4. Verschiedene Fileoperationen

Für die folgenden Routinen wird kein Beispielprogramm gezeigt.

PROCEDURE OPEN(F:FILE VARIABLE,TITLE:STRING;VAR RESULT:INTEGER);

Zweck: Identisch mit der Folge 'ASSIGN(F,TITLE);RESET (F):'.

PROCEDURE CLOSEDEL(F:FILE VARIABLE;VAR RESULT:INTEGER);

Zweck: Schließen und Löschen des Files F. Mit zeitweiligen Files verwendet. Identisch mit der Folge CLOSE, PURGE.

PROCEDURE PURGE(F : FILE VARIABLE)

Zweck: Das File, das mit F verbunden ist, wird von der Diskette gelöscht. Ein ASSIGN muß irgendwann vor dem PURGE-Aufruf ausgeführt worden sein, damit der FIB den Namen des zu löschenden Files enthält. Bei einigen Betriebssystemen kann es notwendig sein, das File vorher zu schließen, damit diese Funktion korrekt arbeiten kann. In diesem Fall ist CLOSEDEL eine nützliche Prozedur.

ANHANG E: BIBLIOGRAFIE

Grogono, Peter, Programming in Pascal,
Addison-Wesley, Reading, Massachusetts, 1978
Eine gute Einführung für das Selbststudium.

Wilson, I.R. und Addyman, A.M.,
A Practical Introduction to Pascal
Springer-Verlag, New York, 1979
Ein Lehrbuch für Fortgeschrittene.

Jensen, Kathleen und Wirth, Niklaus,
Pascal User Manual and Report
Springer-Verlag, New York, 1974
Erste Pascaldefinition, sehr gut als Nachschlagewerk zu benutzen.

"Draft Proposal ISO/DP 7185; Programming Languages - Pascal"
Eine präzise Sprachdefinition, nicht für den Anfänger geeignet.
Erhältlich durch American National Standards Institute,
International Sales Department
1430 Broadway, New York, New York 10018

Findley, William und Watt, David A.,
PASCAL: An Introduction to methodical Programming
Computer Science Press, Potomac, Maryland 1978

Conway, Richard; Gries, David; Zimmerman, E. Carl,
A Primer on Pascal
Winthrop Publishers, Cambridge, Massachusetts, 1976

Miller, Alan R.,
Pascal Programs for Scientists and Engineers
Sybex, Inc., Berkeley, CA., 1981

De Re Atari, "A Guide to effective Programming"
APX-90008 *)

ATARI Disk Operating System II Reference Manual
C016347

ATARI BASIC Reference Manual
C015307

*) Deutsche Fassung erhältlich bei ABBUC e.V. (abbuc.de)

ANHANG F: DEMO PROGRAMM FÜR PLAYER/MISSILEGRAFIK

Das folgende Demoprogramm kann mit dem ATARI Programm-Texteditor eingegeben werden, als Beispiel für modulare Compilation und den Gebrauch der eingebauten Grafik- und Tonprozeduren. Compilieren Sie jedes dieser Module (PMDEMO, PMMIS, PEEKPOKE, PMSND) einzeln. Binden Sie dann diese Module mit der Grafik- und Tonbibliothek (GRSND). Wenn sich der Linker mit dem Sternchen meldet, geben Sie Folgendes ein:

```
D2:PMDEMO,D2:PMMIS,D2:PMSND,D2:PEEKPOKE,GRSND,PASLIB/S
```

Sobald die Module gebunden sind, können Sie das Programm mit dem RUN-Kommando starten. Ein Joystick ist erforderlich, um den Spieler zu bewegen und die Rakete abzufeuern.

```
PROGRAM PLAYER/MISSILE (INPUT,OUTPUT);

(*
Dieses in Pascal geschriebene Programm demonstriert die
Grafikmöglichkeiten von ATARI PASCAL. Es basiert auf dem
in BASIC geschriebenen Player/Missile Demoprogramm. Es
wurde eine Fehlerüberprüfung implementiert, so daß der
Spieler keinen Systemabsturz verursacht, wenn er den Bild-
schirmbereich verläßt. Der Spieler wird erst dann auf dem
sichtbaren Schirm positioniert, wenn durch eine Joystick-
eingabe seine Position den Wert einer sichtbaren Bild-
schirmposition annimmt. Außerdem wird eine sichtbare Ra-
kete abgefeuert, sobald der Spieler den Knopf auf dem
Joystick betätigt. Geräusche für die Bewegung des Spielers
und der Rakete wurden ebenfalls implementiert.

Vier Module müssen separat compiliert und dann zum lauf-
fähigen Objektfile gebunden werden. Diese Module sind
PMSOUND(D2:PMSND.PAS), PEEKPOKE(D2:PEEKPOKE.PAS),
PMMISSILE(D2:PMMIS.PAS), und das Demoprogramm
(D2:PMDEMO.PAS).

Das ausführbare File heißt D2:PMDEMO.COM und kann durch
die 'R'-Operation des Pascalmonitors gestartet werden. Ein
Joystick ist für die Programmausführung notwendig. Der
Player antwortet auf den Joystick mit horizontalen,
vertikalen und diagonalen Bewegungen. Die Rakete wird
durch Niederdrücken ,des Joystickknopfes abgefeuert. Spie-
ler und Rakete können sich gleichzeitig bewegen.
*)
```

```
TYPE
  SCRN TYPE=(FULL SCREEN,SPLIT SCREEN);
  CLEAR_TYPE=(CLEAR_SCREEN,DO_NOT_CLEAR_SCREEN);

VAR
  PMBASE,     (*Spieler/Raketenbasisadresse*)
  X,          (*horizontale Spieler/Raketenposition*)
  Y,          (*vertikale Spielerposition*)
  MISY,       (*vertikale Raketenposition*)
  A:INTEGER;
  FIRED:BOOLEAN;  (*Flag; nach Abschuß der Rakete 'true'
                   gesetzt, auf 'false' gesetzt sobald
                   die Rakete den Schirm verlassen hat *)

EXTERNAL PROCEDURE INITGRAPHICS(MAX_MODE:INTEGER);

EXTERNAL PROCEDURE GRAPHICS(MODE:INTEGER;SCREEN:SCRN_TYPE;
CLEAR:CLEAR_TYPE);

EXTERNAL PROCEDURE SETCOLOR(REGISTER,HUE,LUMINANCE:
INTEGER);

EXTERNAL PROCEDURE SOUND(VOICE,PITCH,DISTORTION,VOLUME:
INTEGER);

EXTERNAL FUNCTION STICK(STKNUM:INTEGER):INTEGER;

EXTERNAL FUNCTION STRIG(STKNUM:INTEGER):INTEGER;

EXTERNAL PROCEDURE MAKENOISE; (*Im Modul PMSOUND*)

EXTERNAL PROCEDURE BIGBANG; (*Im Modul PMMISSILE *)

EXTERNAL PROCEDURE MOVEMISSILE; (*Im Modul PMMISSILE*)

EXTERNAL PROCEDURE PCKEBYTE(ADDR,VAL:INTEGER); (*Im Modul
                                                 PEEKPOKE*)

EXTERNAL FUNCTION PEEKBYTE(ADDR:INTEGER):INTEGER;
                                    (* Im Modul PEEKPOKE*)

PROCEDURE SETPLAYER;
(* SETPLAYER initialisiert den Spieler, indem sie zuerst
den Spielerabschnitt im Speicher löscht und dann die
korrekten Werte in den Speicher schreibt, so daß der
Spieler die unten gezeigte Form annimmt. *)

VAR I:INTEGER;
BEGIN
    (*Löschen des Player Datenbereichs*)
  FOR I:=PMBASE+512 TO PMBASE+64Ø DO POKEBYTE(I,Ø);
  POKEBYTE(7Ø4,1Ø8);  (*Der Spieler wird in Purpur dar-
                        gestellt*)
    (*Initialisierung des Speichers mit der Raketengröße
      und der Spielerform *)

  I:=PMBASE+512 +Y;
  POKEBYTE(I,153);  (*SPIELERFIGUR*)
  I:=I+1;
  POKEBYTE(I,189);  (*            *)
  I:=I+1;           (*            *)
  POKEBYTE(I,255);  (*            *)
```

```
  I:=I+1;            (*            *)
  POKEBYTE(I,189);   (*            *)
  I:=I+1;
  POKEBYTE(I,153)
END;

PROCEDURE MOVERIGHT;
  (* MOVERIGHT bewegt den Spieler nach rechts, indem sie
     das Horizontalpositionsregister inkrementiert *)

BEGIN
  IF X<214 THEN BEGIN  (* 2 Pixel nach rechts schieben*)
    X:=X+1;            (* Inkrementierung *)
    (*neuer Wert wird in das Horizontalregister gepokt*)
    POKEBYTE(53248,X)
  END  (* anderenfalls keine Aktion, da der Spieler
          gerade rechts den Schirm verlassen hat *)
END;

PROCEDURE MOVELEFT
  (* MOVELEFT bewegt den Spieler nach links, indem sie das
     Horizontalpositionsregister dekrementieren *)

BEGIN
  IF X>4Ø THEN BEGIN  (* 2 Pixel nach links schieben *)
    X:=X-1;           (*Dekrementierung*)
    (* Neuer Wert wird in das Horizontalregister gepokt*)
    POKEBYTE(53248,X)
  END  (* anderenfalls keine Aktion, da der Spieler
          gerade links den Schirm verlassen hat *)
END;

PROCEDURE MOVEUP;
  (* MOVEUP bewegt den Spieler auf dem Schirm hinauf indem
     sie die Figur im Speicherbereich hinaufbewegt *)

VAR I:INTEGER;
BEGIN
  IF Y>1 THEN BEGIN  (* bewege den Spieler im Speicher
                        und auf dem Schirm eine Einheit
                        nach oben *)
    FOR I:=Ø TO 6 DO POKEBYTE(PMBASE+522+Y+I,
                                PEEKBYTE(PMBASE+512+Y+I));
    Y:=Y-1  (* Der Spieler hat sich eine Einheit nach
               oben bewegt *)
  END  (* anderenfalls keine Aktion, da sich der Spieler
          gerade über den oberen Bildschirmrand hinaus be-
          wegt hat *)
END;

PROCEDURE MOVEDOWN;
  (* MOVEDOWN bewegt den Spieler auf dem Schirm hinunter
     indem sie die Figur im Speicherbereich hinunterbewegt *)

VAR I:INTEGER;
BEGIN
  IF Y<12Ø THEN BEGIN
    (* bewege den Spieler im Speicher und auf dem Schirm
       eine Einheit nach unten *)
    FOR I:=6 DOWNTO Ø DO POKEBYTE(PMBASE+512+Y+I,
                               PEEKBYTE (PMBASE+511+Y+I));
```

```
    Y:=Y+1  (* Der Spieler hat sich eine Einheit nach
               unten bewegt *)
  END  (* anderenfalls keine Aktion, da sich der Spieler
          gerade über den unteren Bildschirmrand hinaus
          bewegt hat *)
END;

BEGIN  (* Hauptprogramm *)
  INITGRAPHICS(Ø);
  GRAPHICS(Ø,FULL_SCREEN,CLEAR_SCREEN);  (* Schirm löschen *)
  POKEBYTE(755,1);  (* Cursor ausgeschaltet *)
  SETCOLOR(2,Ø,Ø);  (* Hintergrundfarbe schwarz *)
  X:=12Ø;           (* Horizontalkoordinate des Spielers *)
  Y:=48;            (* Vertikalkoordinate des Spielers *)
  A:=PEEKBYTE(1Ø6)-8;
  POKEBYTE(54279,A);  (* Basisadresse für Spieler und
                         Rakete gesetzt *)
  PMBASE:=256*A;      (* Spieler/Raketenadresse gesetzt *)
  POKEBYTE(559,46);   (* DMACTL im OS gesetzt *)
  POKEBYTE(53277,3);  (* GRACTL gesetzt - ermöglicht DMA
                         für Spieler und Rakete auf ihre
                         Speicherregister *)
  POKEBYTE(53248,X);  (* Horizontalposition des Spielers
                         gesetzt *)
  SETPLAYER;          (* Spieler im Speicher gelöscht und
                         neu gesetzt*)

  (* Bewegung und Raketenabschuß *)
  FIRED:=FALSE;  (* Abschußflag initialisiert *)
  WHILE 4>2 DO BEGIN
    A:=STICK(Ø);
    IF A<>15 THEN MAKENOISE;  (* Erzeugung des Bewegungs-
                                 geräusches *)
    (* Bewegung *)
    IF A=5 THEN BEGIN
      MOVERIGHT;
      MOVEDOWN
    END ELSE IF A=6 THEN BEGIN
      MOVERIGHT;
      MOVEUP
    END ELSE IF A=7 THEN MOVERIGHT
    ELSE IF A=9 THEN BEGIN
      MOVELEFT;
      MOVEDOWN
    END ELSE IF A=1Ø THEN BEGIN
      MOVELEFT;
      MOVEUP
    END ELSE IF A=11 THEN MOVELEFT
    ELSE IF A=13 THEN MOVEDOWN
    ELSE IF A=14 THEN MOVEUP
    ELSE IF A=15 THEN SOUND(Ø,182,2,Ø);
      (* Der Spieler bewegt sich nicht, macht also auch
         keine Geräusche *)
    IF FIRED THEN MOVEMISSILE  (* Rakete weiterbewegen *)
    ELSE IF STRIG(Ø)=Ø THEN BIGBANG; (* Rakete abfeuern *)
  END;  (* WHILE *)
END.
```

```
MODULE PMMISSILE
  (* Die Routinen dieses Moduls handhaben Abschuß und Flug
     der Rakete für das Demoprogramm *)

VAR PMBASE,X,Y,MISY:EXTERNAL INTEGER;
    FIRED:EXTERNAL BOOLEAN;

EXTERNAL FUNCTION PEEKBYTE(ADDR:INTEGER):INTEGER;

EXTERNAL PROCEDURE POKEBYTE(ADDR,VAL:INTEGER);

EXTERNAL PROCEDURE SOUND(VOICE,PITCH,DISTORTION,VOLUME:
INTEGER);

PROCEDURE MOVEMISSILE;
  (* MOVEMISSILE wird von der Prozedur BIGBANG aufgerufen,
     wenn die Rakete abgeschossen wird und später vom
     Hauptprogramm, um die Bahn der Rakete festzusetzen.
     Das Hauptprogramm ruft MOVEMISSILE auf, bis die
     Rakete den Schirm verlassen hat und das Abschußflag
     zurückgesetzt ist *)

VAR I:INTEGER;

BEGIN
  IF MISY>5 THEN BEGIN
    FOR I:=Ø TO 1 DO POKEBYTE(PMBASE+383+MISY+I,PEEKBYTE
                                     (PMBASE+384+MISY+I));
    (* Die Rakete wird um ein Feld im Speicher weiterbe-
       wegt *)

    MISY:=MISY-1  (* Die Rakete hat sich eine Einheit wei-
                     terbewegt *)
  END;
IF MISY<=5 THEN FIRED:=FALSE  (* Die Rakete hat den Schirm
                                 verlassen,also wird das
                                 Abschußflag zurückge-
                                 setzt *)
END;

PROCEDURE BIGBANG;
  (* BIGBANG startet die Rakete, sobald der Spieler den
     Abschußknopf drückt. Die Rakete wird gestartet und
     beginnt ihren Flug *)

VAR I: INTEGER;

BEGIN
  FOR I:=PMBASE+384 TO PMBASE+512 DO POKEBYTE(I,Ø);
    (* Missile-Bereich im Speicher löschen *)
  SOUND(3,46,12,14);   (* Beginn des Abschußgeräusches *)
  POKEBYTE(5326Ø,Ø);   (* normale Raketengröße gesetzt *)
  POKEBYTE(53252,X+3); (* Horizontalposition der Rakete
                          gleich der Horizontalposition
                          des Spielers gesetzt *)
  MISY:=Y-1;           (* Vertikalposition der Rakete
                          direkt über die Vertikalposition
                          des Spielers gesetzt *)
  I:=PMBASE+384+MISY;
  POKEBYTE(I,3);       (* Raketenfigur in den Speicher
                          gebracht *)
  FIRED:=TRUE;         (* gesetztes Abschußflag zeigt an,
                          daß eine Rakete abgefeuert wurde *)
```

```
  MOVEMISSILE;          (* Anfang der Raketenbahn *)
  SOUND(3,46,12,Ø)      (* Ende des Abschußgeräusches *)
END;

MODEND.

MODULE PMSOUND;

(* Dieses Modul enthält die Prozedur MAKENOISE, die die
   Tonerzeugung für die Spielerbewegung kontrolliert. Die-
   se Prozedur hat ihr eigenes Modul erhalten *)

EXTERNAL PROCEDURE SOUND (VOICE,PITCH,DISTORTION,VOLUME:
INTEGER);

  PROCEDURE MAKENOISE;

    (* Erzeugung des Maschinengeräusches, wenn der Spieler
       sich bewegt *)

  BEGIN
    SOUND(Ø,182,2,6)
  END;

MODEND.

MODULE PEEKPOKE;

  (* Dieses Modul enthält die aus BASIC bekannten Peek-
     und Pokeprozeduren *)

  PROCEDURE POKEBYTE(ADDR,VAL:INTEGER);

  (* POKEBYTE bietet dem Pascalbenutzer eine dem BASIC-Be-
       fehl ähnliche Methode, Speicherinhalte zu ändern.
     Eintritt: POKE(ADDR,VAL); (Beispielaufruf)
               ADDR = Adresse des Speicherplatzes
               VAL  = abzuspeichernder Wert
     Ausgang:  Der Inhalt des Speicherplatzes ADDR ist
               gleich VAL.
     Veränderungen: ADDR(Adresse)
     Aufrufe: - keine - *)

  VAR
    PTR: ^CHAR;   (* Zeiger auf die zu verändernde Adresse *)
  BEGIN
    PTR:=ADDR;    (* PTR zeigt jetzt auf die gewünschte
                     Adresse *)
    PTR^:=CHR(VAL)  (* Abspeichern des Wertes VAL in die
                       durch PTR angezeigte Adresse *)
  END;

FUNCTION PEEKBYTE(ADDR:INTEGER):INTEGER;

  (* PEEKBYTE bietet dem Pascalbenutzer eine dem BASIC-Be-
     fehl PEEK ähnliche Methode, die Inhalte von Speicher-
     plätzen zu überprüfen.
     Eintritt: INTEGERVARIABLE:=PEEKBYTE(ADDR); (Beispiel-
                                                  aufruf)
               ADDR = Adresse des zu überprüfenden Spei-
                      cherplatzes
```

```
    Ausgang: PEEKBYTE = Inhalt des durch ADDR angegebenen
                        Speicherplatzes.
    Veränderungen: Integervariable in der aufrufenden
                   Routine.
    Aufrufe: - keine - *)

  VAR
    PTR:^CHAR;  (* Zeiger auf die zu überprüfende Adresse *)

  BEGIN
    PTR:=ADDR;  (* Der Zeiger wird mit der gewünschten
                   Adresse geladen *)
    PEEKBYTE:=ORD(PTR^)  (* PEEKBYTE ergibt den Inhalt
                            der durch PTR spezifizierten
                            Adresse *)
  END;

MODEND.
```

ANHANG G: HILFREICHE HINWEISE

Im Folgenden einige ausgewählte Hinweise, die sich beim Gebrauch des ATARI PASCAL Sprachsystems als nützlich erweisen können.

1. Die Compilation von Pascalprogrammen, die Fließkommazahlen (REALS) benutzen, erfordert, daß die Einschlußfiles FLTPROCS oder STDPROCS im Deklarationsblock des Quellprogramms angegeben werden. Zusätzlich müssen FPLIB und PASLIB in Ihr compiliertes Quellprogramm eingebunden werden. Eine Unterlassung wird zu einem Fehler während des Compiler- und/oder des Linkerlaufes führen. Betrachten Sie das Demoprogramm CALC als Beispiel.

2. Nur die ersten acht Zeichen eines Bezeichners sind signifikant.

3. CLOSEDEL kann auf jedes File angewendet werden, seien Sie also vorsichtig. Sie könnten versehentlich etwas löschen, was Sie gar nicht löschen wollten.

4. Während Standardprozeduren in den Compiler eingebaut sind, benötigen andere für Deklarationszwecke das korrekte Einschlußfile. Überprüfen Sie die Files und entscheiden Sie über ihren Gebrauch. Diese Einschlußfiles können unter DOS mit der COPY-Option auf dem Drucker aufgelistet werden.

5. Das reservierte Wort "PREDIFINED" erlaubt gewissen Prozeduren, Teil des das Programm umgebenden Rahmens zu werden. Außerdem wird jeder Fileparameter in Form zweier Parameter übergeben, so wie es die Laufzeitroutinen verlangen.

STICHWORTVERZEICHNIS

ABSOLUTE Variable 28,46
ADDR 36
AND und 16-Bit-Variable 75
Anforderungen
  Laufzeit 4
  System 4
Angleichbare Arrays 82
ARCTAN 83
ARRAY als Prozedurparameter 82
      Abspeicherung 25
ASSIGN 41,1Ø6
Ausgabe, formatiert 112
Ausnahmeüberprüfung s. Compilerschalter

BCD REAL Zahlen 56
Benutzertabelle 8
Bereichsüberwachung s. Laufzeit
Bezeichner
  mit '@' 67,68
  externe Signifikanz 22
  legale Pascal- 67
Bit- und Bytemanipulation 33,75
BLOCKREAD 42
BLOCKWRITE 42
BOOLEAN 55
Bytemanipulation s. Bit- und Bytemanipulation

CALC.PAS 7
CHAR 55
CHR 55,71,84
CLOSE 43,1Ø8
CLOSEDEL 43,116
CLRBIT 33
Compilerschalter
  $E Einstiegspunktkontrolle 12
  $P/$L Listkontrolle 14
  $R Laufzeitbereichskontrolle 13
  $X Laufzeitausnahmeüberwachung 14
  $I Quellcodeeinschlußmechanismus 13
  $T/$W strenge/gelockerte Typüberwachung 13
  Zusammenfassung 14
  Syntax 12
Compiler
  Ausgabe eines '#' 8,12
  Ausgabe eines '_' 8,12
  verfügbarer Speicher 8
  Informationsausgabe während der Compilierzeit 7,12
  Ausführung 7,11
  Operationsbeschreibung 11
  Phase 1 12,15
  Phase 2 12
  übrigbleibender Speicher 8
  Beispielausgabe 7
  Separate Compilation 22
  schrittweise Instruktionen 7
  Systemanforderungen 4
  Raum für die Benutzertabelle 8

CONCAT 38
COPY 39

Datenspeicherung 55
Datentypen
  Gültige Bereiche 55
  Größe 55
  BOOLEAN 55
  BYTE 56
  CHAR 55
  INTEGER 56
  REAL 56
  SET 56
  WORD 56
DELETE 40
Diskette, gelieferte
  Inhalt 4
Drucker
  Zuweisung 41
  Ansteuerungsbeispiel 115
  Ansteuerung 115
  Minimalkonfiguration 3

End-of-File 103,110
Eingebaute Funktionen
  Zusammenfassung 45
  ADDR 36
  ASSIGN 41
  BLOCKREAD 42
  BLOCKWRITE 42
  CLOSE 43
  CLOSEDEL 43
  CLRBIT 33
  CONCAT 38
  COPY 39
  DELETE 40
  EXIT 32
  FILLCHAR 37
  GNB 42
  HI 35
  INSERT 41
  IORESULT 44
  LENGTH 38
  LO 35
  MAXAVAIL 44
  MEMAVAIL 44
  MOVE 30
  MOVELEFT 30
  MOVERIGHT 30
  OPEN 43
  POS 39
  SETBIT 33
  SHL 34
  SHR 34
  SIZEOF 37
  SWAP 35
  TSTBIT 33
  WNB 42
  PURGE 43
Einschlußfiles 4,5,8,13,125

EOF 83,1Ø8
EOLN 83,1Ø8
EXIT 32
Erweiterungen des ISO Standards s. ISO-Standarderweite-
rungen
Erweiterungen, Zusammenfassung der 81
EXTERNAL
  Eintrittspunktsymbole 12
  Modulare Compilation 22
  Prozeduren/Funktionen 22
  Variable 23
  Routinen als Parameter 22

Fehlermeldungen
  Typkonflikt 71
  Zusammenfassung 15,94
Fenstervariable s. Files
FIB s. Fileinformationsblock
Fileinformationsblock 1Ø3
Filevariable 1Ø3
  Zulassung typfreier Files 71
Filename
  Definition 1Ø2
  interne und externe Zuordnung 41
  Compilereingabe 7,12
  Linkereingabe 9,16
Files
  ASCII Text 71
  ASSIGN Prozedur 41
  Zuordnung externer Namen an Files 81
  Eingebaute Prozeduren 81
  Verkettung 123
  Schließen 43,1Ø8
  Erzeugung 1Ø6
  Definition 1Ø2
  Löschen 43
  Geräte E:,S:,K:,P: 42
  Fehlerbehandlung 44
  Beispiele 1Ø5
  Schnelle Byteroutinen 42
  Formatierte Ausgabe 112
  Hexadezimale Ausgabe 85
  Implizite Umwandlungen 71
  Lokale Files 41
  Lokale Files und /D Schalter des Linkers 17
  Eröffnen (s. auch RESET) 43
  Vordefinierter Typ TEXT 7Ø
  Direkter Filezugriff 42
  Druckerausgabe 41,12
  Zeitweilige Files s. lokale Files
  Text 11Ø,113
  Typfreie Files 7Ø
  Fenstervariable 1Ø4,1Ø7,1Ø8,1Ø9
  Druckeransteuerung 115
FILLCHAR 37
Fließkommazahlen REAL 56
Formatierte Ausgabe 84,112
FORWARD 81
FPLIB.ERL 4,9,16,56,125

GET 1Ø8
GNB 42
GOTO 77
GRSND.ERL 5,118

Heapmanagement
  ISO Standard 117
  MEMAVAIL,MAXAVAIL 44
  Parameter 83
Hexadezimalzahlen 85
HI 35

I/O s. Files
INLINE
  Beispiele 47
  Syntax 46
INSERT 41
INTEGER 56
IORESULT 44,1Ø7,113,115
ISO Standarderweiterungen
  Absolute Variable 73
  Zusätze zur Zuweisungskompatibilität 76
  BNF Syntaxbeschreibung für ATARI PASCAL 86
  Eingebaute Prozeduren und Funktionen 3Ø
  Verkettung 28
  Kurze Liste der ATARI PASCAL Fähigkeiten 1
  ELSE in Zusammenhang mit CASE-Statements 77
  Externe Prozeduren 79
  INLINE 46
  Modulare Compilation 22
  Leere Strings 67
  Operatoren 75
  WRD-Typ Übertragungsfunktion 83
ISO Standard
  Zuweisungskompatibilität 71
  FOR-Schleifen; Abweichungen von Jensen und Wirth 78
  Von ATARI benutzter Auszug 1
  Erweiterungen für angleichbare Arrays 82
  Zusammenfassung der Eigenschaften 65
  Typkompatibilität 71

Kommentare 33
Kompatibilität zu UCSD Pascal 67

LENGTH 38
Linker
  /D Schalter, Verkettung 28
  Eigenschaften kompatibler Module 19
  /F Schalter, Kommandofiles 18
  /D Schalter, Datenursprung 16
Auswirkungen der /P und /D Schalter auf den Inhalt des .COM-Files 17
Auswirkungen des /D Schalters auf lokale Files 17
  /E Schalter, Erweiterung des Mappings 17
  Speicherplatzeinsparungen 16
  Eingabefilenamen 16
  Aufruf 17
  /S Schalter, Bibliothekssuche 16

/L Schalter, load mapping 17
/P Schalter, Programmursprung 17
Beispiel 9
Beispielausgabe 17
Platzersparnis durch den /D Schalter 17
Zusammenfassung 18
Schalter 19
LINK 16
Listing 7
LO 35
Lokale Files s. Files

MEMAVAIL 44
MAXAVAIL 44
Modulare Compilation
$E Schalter 22
EXTERNAL 22
Beispiel 22
Überblick 22
Syntax 22

MOVE 30
MOVELEFT 30
MOVERIGHT 30

NOT und 16-Bit-Variable 75

ODD 55,83
OPEN 43,108,116
Operatoren
AND 55,75
und 16-Bit-Variable 75
NOT 55,75
OR 55,75
Schalteroptionen
Compiler 14
Linker 18
OR und 16-Bit-Variable 75
ORD 55,56,72,83

PACKED 55,69
PASLIB 9,125
PASLIB.ERL 71
Pointer 71
Portabilität 13
POS 39
Programmbeispiele
CHAIN Demo 29
DEMOCON (angleichbare Arrays) 82
DEMO_INLINE 47
External_Demo (modulare Compilation) 23
PRINTER 115
Prozedur ACCESS (Strings) 58
Prozedur ADDR DEMO 36
Prozedur ASSIGN (Strings) 57
Prozedur COMPARE (Strings) 56
Prozedur CONCAT_DEMO 38
Prozedur COPY_DEMO 39
Prozedur DELETE_DEMO 40
Prozedur EXITTEST 32

Prozedur FILL_DEMO 37
Prozedur INSERT_DEMO 41
Prozedur MOVE_DEMO 3Ø
Prozedur POS_DEMO 39
Prozedur SHIFT_DEMO 34
Prozedur SIZE_DEMO 37
Prozedur TST_SET_CLR_BITS 33
Prozedur TEXT1Ø_DEMO 113
Prozedur WRITE_READ_FILE_DEMO 1Ø5
PURGE 43,116
PUT 1Ø4

READ 1Ø9
READLN 114
REAL
BCD 56
Fließkomma 56
RECORD
Abspeicherung 25
Meldung über verbleibende Speicher 8
Reservierte Worte 93
RESET 1Ø8
REWRITE 1Ø6
Skalartypen
Abspeicherung 25
SET 26
SETBIT 33
SHL 34
SHR 34
SIZEOF 37
STRING 11Ø
STRING Details der Implementierung 56
STRING
Zugriff 59
READLN 114
Zuordnung 56
Vergleich 58
CONCAT 38
COPY 39
Vordefinierte Länge 7Ø
Definition 56,75
Explizite Längendeklaration 7Ø
Leerstring 68
Laufzeitfehler 53
Gebrauch als Array of Char 73
Strings
DELETE 4Ø
INSERT 41
LENGTH 38
POS 39
SWAP 35
Symbole
Bezeichnersignifikanz 68
Gebrauch des '@' 67
Gebrauch von Hexadezimalliteralen 68
Gebrauch des '_' 68
TEXT Files
Definition 11Ø
TSTBIT 33

Typüberwachungsschalter 13
Typkonfliktschalter 7Ø
Typen
  ABSOLUTE Zusatz für Variable 46
  Datenimplementation 55
  Erweiterungen 68
  Filetypen 7Ø
  PACKED Implementation 69
  Zeiger 71
  Vordefinierte Typen 69
  Bereich des SET Typs 7Ø
  Beschränkungen für ABSOLUTE Strings 46
Verkettung
  Beispiel 29
  Heapmanagement 28
  Instruktionen 28
  Kommunikation durch absolute Variable 28
  Kommunikation durch globale Variable 28

WNB 43
WORD 56,69
WRITE 1Ø7
WRITELN mit Textfiles 111
Zeiger 71
Zeile 11Ø
Zeilennummer 15
Zuweisungskompatibilität 71

ATARI®

A Warner Communications Company

ATARI-Elektronik Vertriebsgesellschaft mbH
Postfach 60 01 · Bebelallee 10 · 2000 Hamburg 60

Jegliche Rechte vorbehalten.
Vermietung, Verleih, Vervielfältigung
und öffentliche Aufführung verboten.