robotron

Bedienungsanleitung und Sprachbeschreibung für das Programmiersystem PASCAL 880/S unter Steuerung des Betriebssystems SCPX

Systemunterlagen-dokumentation	Anwenderdokumentation Beschreibung

Stand 31.08.1986

Bedienungsanleitung und Sprachbeschreibung

fuer das

Programmiersystem P A S C A L 880/S

unter Steuerung des Betriebssystems SCPX

VEB ROBOTRON
Bueromaschinenwerk
"Ernst Thaelmann"
Soemmerda

Die Dokumentation wurde von einem Kollektiv der Sektion Mathematik und Datenverarbeitung der Handelshochschule Leipzig unter Leitung von Prof. Dr. sc. Goldammer ausgearbeitet. Dabei wurden Teile der Dokumentation PASCMP des VEB Kombinat Robotron sowie weitere Materialien verwendet.

Hinweise zur Verbesserung der Dokumentation und zur Fehlerkorrektur richten Sie bitte an:

VEB Kombinat Robotron
Bueromaschinenwerk "Ernst Thaelmann"
Bereich Forschung und Entwicklung
Soemmerda
5230

Inhaltsverzeichnis Seite

1. Einfuehrung

PASCAL 880/S ist ein leistungsfaehiges Programmiersystem fuer 8-Bitrechner. Es ist lauffaehig unter SCPX, bei einem verfuegbaren Speicherbereich (TPA) ab 48 KByte.

Das System besteht aus folgenden Teilen:

1. Systemkern mit Editor, Compiler, Laufzeitbibliothek, Interface (PASCAL.COM), Fehlertextfile (PASCAL.TXT) und einer Restartroutine (PASCAL.RES) zum Wiederladen des Systemkerns nach der Ausfuehrung von Kommandofiles, oder bei Arbeit mit nur einer Diskette.
2. Systemservice mit Routinen zum Loeschen, Umbenennen, Kopieren, Status aendern, Chiffrieren und Verdichten von Files und Quelltext- sowie Moduldiagrammlistern (PLUS.COM). Der Systemservice ist vom Systemkern aus und autonom erreichbar.
3. Hilfsprogramme zum Retten von editiertem Quelltext bei Systemabsturz (PASRETT.COM) sowie zum Installieren von Systemkern und Systemservice (PASINST.COM).

Alle Komponenten (Systemkern, Systemservice und Hilfsprogramme) unterliegen dem Aenderungsdienst und tragen deshalb gesonderte Versionsnummern.
Die von Compiler und Bibliothek verwaltete Sprache PASCAL ist kompatibel zu TURBO-PASCAL /1/. Ausnahmen sind CRTINIT, CRTEXIT, HIGHVIDIO, NORMVIDIO, LOWVIDEO, die auch vom Compiler abgewiesen werden. Die Sprachelemente LABEL/GOTO koennen fuer Lehrzwecke mit PASINST herausgenommen und wieder installiert werden. Der Systemservice ermoeglicht hochwertige Listings mit Einruecken entsprechend der Schachtelungstiefe, Crossreferenzliste fuer selbstdefinierte Bezeichner, paarweises Ausziffern von BEGIN/END sowie eine Schachtelungsuebersicht aller Unterprogramme in Diagrammform. Das erleichtert und beschleunigt die Entwicklungsarbeit erheblich. Fuer Lehrzwecke in Kabinetten koennen Kopieren und Drucken an Codeworte gebunden werden.

Systemkern und Systemservice unterstuetzen module Programmierung durch Include-Files auf PASCAL-Quelltext-Ebene. Dazu wird in den Quelltext des Hauptfiles eine INCLUDE-Direktive {$I <Filename>} /2/ geschrieben. Systemkern und Systemservice unterbrechen an dieser Stelle und verarbeiten den Quelltext des Files <Filename>. Die Include-Technik ermoeglicht so die Entwicklung grosser Programme und eine rationelle Arbeit mit Anwenderbibliotheken.

/1/ TURBO-PASCAL ist ein eingetragenes Warenzeichen der Firma Borland International USA

/2/ Die Schreibweise <Filename> und weitere Notationen in Ziffer 2 und 3 sind ein Vorgriff auf die Metasprache in Ziffer 4.1.

2. Systemkern (Editor, Compiler, Interface)

2.1. Start und Menue

Der Start erfolgt durch die Eingabe des Kommandos PASCAL.

Danach erscheint das Titelbild (zz:Zeilenanzahl), ss:Spaltenanzahl des Monitors):

```
 /-------------------------------------------------\
¦ PASCAL 880/S (c) 01/08/86  VEB ROBOTRON BWS      ¦
¦ Systemkern Version <n>.<m>/SCPX                  ¦
¦ PC robotron 1715 (Monitor zzxss                  ¦
¦                                                  ¦
¦                                                  ¦
¦       Fehlertext laden   (J/N) ?                 ¦
¦                                                  ¦
¦                                                  ¦
 \-------------------------------------------------/
```

Geraetebezeichnung und Bildschirmgroesse sind installierbar (vergl. Ziffer 5.2).
Die Frage ist mit J oder N zu beantworten. Bei J wird die Komponente PASCAL.TXT in den TPA geladen. Sie enthaelt den Fehlertext des Compilers. Durch das Laden wird der Editorpuffer um 1635 Bytes verkleinert. Dafuer wird aber bei auftretenden Fehlern ausser der Fehlernummer auch der Fehlertext ausgegeben.

Beispiel fuer Compilerfehlermeldung:
Fe-Nr.5: Taste ESC-> bei N.
Fe-Nr.5: ')' fehlt. Taste ESC-> bei J.

Bei Programmen mit umfangreichem Quelltext und bei einer TPA-Groesse von 48 K kann mit der Antwort N Platz gespart und ein Ueberlauf verhindert werden. In diesem Falle muss bei der Fehleranalyse auf die Fehlertabelle im Anhang D zurueckgegriffen werden.

Nach Beantwortung der Frage mit N erscheint das Grundmenue (hier fuer 48 KB TPA, Fehlertext nicht geladen):

```
 /-----------------------------------------------\
¦ L)aufwerk: A                                    ¦
¦                                                 ¦
¦ A)ktivfile                                      ¦
¦ H)auptfile                                      ¦
¦                                                 ¦
¦ E)ditor  C)ompiler-O)ptionen T)est              ¦
¦ S)ichern B)eenden  K)ommando D)ir               ¦
¦                                                 ¦
¦ Text :     0 BYTES (7BF6-7BF6)                  ¦
¦ Frei : 20495 BYTES (7BF7-BF06)                  ¦
¦                                                 ¦
¦ >                                               ¦
 \-----------------------------------------------/
```

Ist der Fehlertext geladen, wird der Anfang des Editorpuffers nicht mit $7BF6 sondern mit $825C angezeigt.
Die obere Grenze $BF06 ist betriebssystemabhaengig. Bei 52KB TPA ist sie $CC06.
Als letztes Zeichen erscheint das Promptzeichen >. Durch Eingabe eines Kommandobuchstabens wird die entsprechende Komponente des Systemkerns aufgerufen. Nach Abarbeitung dieser Komponente erscheint wieder das Promptzeichen und eine andere Funktion kann aufgerufen werden.
Wird nach dem Promptzeichen eine Taste gedrueckt, die keinem Kommandobuchstaben entspricht, dann wird der Bildschirm geloescht und das Grundmenue erscheint. Dieses Wiederaufrufen des Grundmenues sollte man grundsaetzlich durchfuehren, falls die aktuellen Werte im Menue gewuenscht werden.

Es gibt folgende Kommandobuchstaben:

L Laufwerkzuweisung.
Dieses Kommando bewirkt zugleich das Ruecksetzen des Diskettensystems.

A Aktivfile laden.
Das File steht danach zum Editieren bereit.

H Hauptfile spezifizieren.
Dieses Kommando muss gegeben werden, wenn das zu compilierende Programm Includefiles enthaelt.

E Editieren.
Eingabe / Veraenderung des Aktivfiles.

C Compiliert.
Compiliert wird das Hauptfile, wenn nicht angegeben, dann das Aktivfile.

T Testlauf.
Ausfuehren des compilierten Programmes, wenn dieses mit den Optionen T oder P (vergl. Nebenmenue) uebersetzt wurde.

D Directory.
Anzeige der Directory im aktuellen Laufwerk sowie des freien Speicherplatzes auf der Diskette.

S Sichern.
Schreiben des Aktivfiles auf Diskette.

O Optionen.
Setzen von Compiler-Optionen ueber ein Nebenmenue.

K Kommando (vergl. Ziffer 2.12.).
Anschluss zum Systemservice mit PLUS

B Beenden.
Rueckkehr zum Laufzeitsystem SCPX.

Die Kommandos werden sofort nach Eingabe ausgefuehrt. Es ist also kein <ET> zu druecken.

2.2. Laufwerkzuweisung

Nach Eingabe von L erscheint die Aufforderung zur Laufwerkzuweisung:

```
Neu(^C):B<ET>
```

Mit B <ET> wird als aktuelles Laufwerk B zugewiesen. Wird nur <ET> gedrueckt, bleibt die alte Zuweisung erhalten.

Bei jeder Ausfuehrung des Kommandos L wird stets auch das Diskettensystem zurueckgesetzt. Das Ruecksetzen des Diskettensystems (^C) nach Diskettenwechsel kann also durch Aufruf des Kommandos L ohne Angabe eines neuen Laufwerkes (nur <ET>) erfolgen.
Die Neuzuweisung eines Laufwerkes wird im Menue nicht sofort angezeigt. Dazu ist nach Erscheinen des Promptzeichens > nochmal <ET> notwendig.

2.3. Aktivfilezuweisung

Nach Eingabe von A erscheint die Aufforderung zur Eingabe des Namens fuer das Aktivfile. Es gelten die bekannten Regeln des SCPX.
Die Eingabe wird mit

```
Aktivfilename:
```

angefordert und mit

```
Laden <Name>
```

quittiert. <Name> schliesst eine eventuelle Laufwerksangabe ein.

Damit wird das File <Name> von dem jeweiligen Laufwerk in den Arbeitsbereich geladen und kann danach mit E oder C weiterbearbeitet werden. Eine Namenserweiterung .PAS erfolgt standardmaessig. Existiert das File nicht, erscheint die Ausschrift:

```
File neu
```

In diesem Falle kann danach mit E das File neu angelegt werden. Falls im Arbeitsbereich ein noch nicht gerettetes, bearbeitetes File steht, wird dies angezeigt durch:

```
Aktivfile <Name> noch sichern (J/N)?:
```

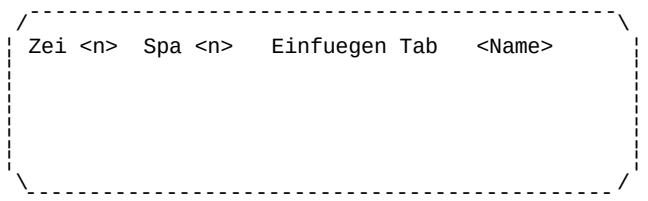
Bei Antwort J wird das alte Aktivfile auf dem jeweiligen Laufwerk abgelegt, und das neue Aktivfile zugewiesen.
Bei Antwort N wird das im Arbeitsbereich (dem Editorpuffer) stehende alte Aktivfile durch das neue ueberschrieben.

2.4. Hauptfilezuweisung

Nach Eingabe von H erscheint die Aufforderung zur Eingabe des Namens fuer das Hauptfile nach den bekannten Regeln fuer SCPX. Die Namenserweiterung .PAS ist wieder Standard.
Die Benutzung eines Hauptfiles wird notwendig, wenn ein PASCAL-Programm Includeanweisungen (vergl. Ziffer 4.1.4.6.) enthaelt. Der Compiler beginnt dann die Uebersetzung mit dem Hauptprogramm und, wenn er auf eine Includeanweisung trifft, laedt das durch die Includeanweisungen benannte File in den Aktivfilebereich. Die Includefiles werden Aktivfiles und an dieser Stelle vom Compiler uebersetzt. Treten bei der Compilierung Fehler auf, kann das fehlerhafte File wie ueblich korrigiert werden. Das Ablegen des geaenderten Files erfolgt automatisch und bei einem erneuten Compilerlauf wird auch das Hauptfile wieder geladen.

2.5. Editieren

Nach Eingabe von E erscheint der Text des Aktivfiles auf dem Bildschirm und der Editor wird gestartet. Der Text kann dann bearbeitet werden.
Existiert noch kein Aktivfilename, so erfolgt die Aufforderung zur Eingabe dieses Namens, danach wird das File geladen oder, wenn es nicht gefunden wurde, die Ausschrift "File neu" ausgegeben und der Editor gestartet. Es erscheint das Bild:



Die erste Zeile bleibt als Statuszeile stets auf dem Bildschirm stehen. Es bedeuten:

Zei <n>	n=Zeilenposition des Cursors
Spa <n>	n=die Spaltenposition des Cursors
Einfuegen	Anzeige Einfuegen.Im Wechsel (CTRL V) mit Ersetzen
Tab	Zeigt automatisches Einruecken an
<Name>	Filename des gerade editierten Textes (eventuell einschliesslich Laufwerksangabe)

In den der Statuszeile folgenden Zeilen kann der Programmtext geschrieben werden. Jede Zeile ist durch <ET> abzuschliessen. In den untenstehenden Kommandos kann das zweite Control-Zeichen durch den entsprechenden normalen Buchstaben ersetzt werden, d.h.,statt ^Q^L kann man auch ^QL druecken.
Fuer das Editieren koennen folgende Kommandos verwendet werden, die weitgehend mit denen von Textverarbeitungssystemen ueberein stimmen:

Cursorbewegungen:

Zeichen links	^S	Wirkt nur bis Zeilenanfang.
Zeichen rechts	^D	Wirkt nur bis Spalte 125.
Wort links	^A	Zum Wortanfang. Worte werden begrenzt durch: <space> {}<>,.()[]^'*+-/$ Wirkt ueber Zeilengrenzen.
Wort rechts	^F	Analog oben.
Zeile hoch	^E	Rollmodus bei Erreichen oberer Zeile.
Zeile tief	^X	Rollmodus bei vorletzter Zeile.
Rollen hoch	^Z	Rollen um eine Zeile zurueck.
Rollen tief	^W	Rollen um eine Zeile vorwaerts.
Blaettern hoch	^R	Rollen um ein Bild zurueck.
Blaettern tief	^C	Rollen um ein Bild vorwaerts.
Zeilenanfang	^Q^S	Nach erster Spalte.
Zeilenende	^Q^D	Nach Spalte hinter dem letzten Zeichen.
Bildanfang	^Q^E	Nach oberster Bildzeile.
Bildende	^Q^X	Nach unterster Bildzeile.

Fileanfang	^Q^R	Nach erster Textzeile des Files.
Fileende	^Q^C	Nach letzter Textzeile des Files.
Blockbeginn	^Q^B	Zum Blockbeginn.
Blockende	^Q^K	Zum Blockende.
Letzte Position	^Q^P	Rueckkehr zur letzten Cursorposition, nach SUCH/ERSETZ-Kommando o.a.

Insert und Delete Kommandos

Einfuege-Modus ein/aus	^V	Einfuegen oder ueberschreiben.
Loeschen links	DEL	Wirkt ueber Zeilengrenzen.
Loeschen Zeichen	^G	Wirkt ueber Zeilengrenzen.
Loeschen Wort rechts	^T	Wirkt ueber Zeilengrenzen.
Zeile einfuegen	^N	Wenn der Cursor sich nicht am Zeilenanfang befindet, wird die Zeile an dieser Stelle getrennt.
Zeile loeschen	^Y	Zeile loeschen, in der Cursor steht.
Zeilenrest loeschen	^Q^Y	Loescht Zeile ab Cursor bis Zeilenende.

Block Kommandos:

Blockbeginn	^K^B	Die gesetzte Marke ist nicht sichtbar.
Blockende	^K^K	Die gesetzte Marke ist nicht sichtbar.
Marken loeschen	^K^H	Blockmarken werden geloescht, unabhaengig von der Cursorstellung.
Wort markieren	^K^T	Markieren des Wortes an Cursorposition oder links von ihm. Wort wird wie Block behandelt
Block kopieren	^K^C	Kopieren an die Cursorposition. Marken wandern mit.
Block verschieben	^K^V	Verschieben an die Cursorposition. Marken wandern mit.
Block loeschen	^K^Y	Achtung! Ein geloeschter Block ist nicht zurueckholbar.
Block lesen	^K^R	Block von einem angeforderten File lesen.
Block schreiben	^K^W	Block in ein angefordertes File schreiben. Marken und Block verbleiben an alter Stelle.

Bei nichtvorhandenem Block haben die Blockkommandos keine Wirkung. Es erfolgt keine Fehlermeldung.

Spezielle Kommandos

Editieren Ende	^K^D	Rueckkehr zum PASCAL-Grundmenue. Es ist zu beachten, dass das Aktivfile noch nicht gesichert ist.
Tab	^I	Die Tabpositionen werden durch die Wortanfaenge der vorhergehenden Zeile bestimmt. Achtung! Im Einfuegemodus werden Tabzeichen eingefuegt!

Tab ein/aus	^Q^I	Bei <ET> springt Cursor in naechster Zeile nicht zur Spalte 1, sondern unter das erste Wort Bei Tab ein wird Tab in Statuszeile angezeigt. Bei Tab aus ist diese Stelle leer.
Ruecksetzzeile	^Q^L	Werden Veraenderungen in einer Zeile durchgefuehrt, so koennen diese alle durch dieses Kommando wieder rueckgaengig gemacht werden, solange dabei der Cursor die Zeile nicht verlassen hat.
Suchen	^Q^F	Die Suchkette kann aus 30 Zeichen bestehen. Sie kann CTRL-Zeichen enthalten und wird durch <ET> beendet. Zeilenende kann durch ^M^J erzeugt werden. In der Suchkette kann ^A als Wildcard (Maskenzeichen) verwendet werden. Zusaetze: B Rueckwaerts suchen G Globales suchen (Anfang bis Ende) n Suchen des n. Auftretens U Ignorieren Gross/Kleinbuchstaben W Nur Worte suchen Zusaetze sind ohne Zwischenraum zu schreiben und mit <ET> zu beenden.
Suchen und Ersetzen	^Q^A	Suchkette und Zusaetze werden wie beim Suchen angegeben. Die Zeichenkette zum Ersetzen kann 30 Zeichen lang sein. Sie kann CTRL-Zeichen enthalten (^A ohne Bedeutung). Abschluss durch <ET>. Zusaetze: B Rueckwaerts suchen G Globales suchen (von Anfang bis Ende) n n-maliges Ersetzen N Ersetzen ohne Fragen: (Ersatz (J/N)?) U Ignorieren Gross/Kleinbuchstaben W Nur Worte suchen Zusaetze ohne Zwischenraum schreiben und mit <ET> beenden.
Suchen wiederholen	^L	Wiederholen des letzten ^Q^F oder ^Q^A Kommandos.
Eingabe CTRL-Zeichen	^P	Im Programmtext koennen CTRL-Zeichen durch durch Voranstellen von ^P eingegeben werden. Beispiel: ^G durch ^P^G.
Abort-Kommando	^U	Jedes Editor-Kommando kann durch ^U sofort abgebrochen werden.

Zur Vereinfachung der Editierung und zur Nutzung der Rechnertasten wird folgende Tastatur-Installation (mit dem Programm INSTSCP bzw. KEYINST) empfohlen:

Tastencode	Belegung	
$82 (INSMOD)	$16 (^V)	Einfuegen ein/aus
$CD (F14)	$0B, $44 (^KD)	Verlassen Editor
$87 (¦<--)	$01 (^A)	Wort nach links
$88 (<--)	$08 (^H)	Zeichen nach links
$9D (<--¦)	$11, $53, $18 (^QS^X)	Anfang naechste Zeile
$8B (↑)	$05 (^E)	Zeile nach oben
$8C (↖)	$12 (^R)	Seite nach oben
$8A (↓)	$18 (^X)	Zeile nach unten
$86 (-->)	$04 (^D)	Zeichen nach rechts
$8E (F15)	$03 (^C)	Seite nach unten
$89 (-->¦)	$06 (^F)	Wert nach rechts

2.6. Compilieren

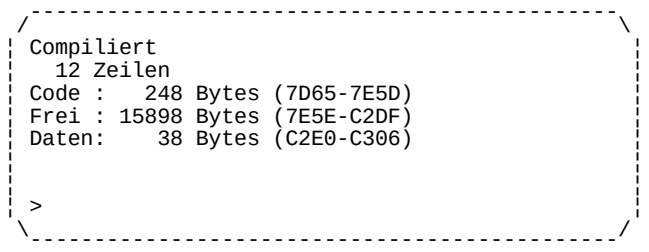
Nach Eingabe von C erfolgt die Compilierung des
- Hauptfiles, wenn es existiert,
- Aktivfiles im anderen Fall.

Existiert noch keine Aktivfile, erfolgt die Aufforderung zur Eingabe des Aktivfilenamens.
Wird das Aktivfile nicht auf der angegebenen Diskette gefunden, erscheint die Mitteilung "File neu" und es wird versucht, das leere File zu uebersetzen. Das Ergebnis ist die Meldung "Fe-Nr.91: Vorzeitiges Ende des Quell-Files. Taste ESC->". Nach dem Druecken der Taste ESC wird der Editor aufgerufen und das neue File kann erstellt werden. Soll dies nicht geschehen, kann der Editorlauf durch ^KD beendet werden. Damit wird das Grundmenue erreicht. Das leere File wird nicht gerettet.

Die Compilierung und die Form des Objektcodes ist abhaengig von verschiedenen Compiler-Optionen. Es sind Standardwerte festgelegt, die eine zeitguenstige Uebersetzung, minimalen Objektcode und schnelle Abarbeitung des uebersetzten Programmes ergeben. Fuer bestimmte Faelle koennen diese Standardwerte veraendert werden (vergl. Ziffer 2.12.). Bei Auftreten eines Fehlers im Quelltext wird die Compilierung abgebrochen in der Form:

```
 /-----------------------------------------------\
¦ >C                                            ¦
¦                                               ¦
¦ Compiliert                                    ¦
¦   10 Zeilen                                   ¦
¦ Fe-Nr.5: ')' fehlt. Taste ESC->               ¦
 \-----------------------------------------------/
```

wobei der Fehlertext nur dann ausgeschrieben wird, wenn PASCAL.TXT geladen wurde (Eingabe J bei "Fehlertext laden (J/N)?" nach dem Starten des Systemkerns). Nach Druecken der Taste ESC wird der Editor gestartet und der Cursor hinter den Fehler positioniert. Es kann sofort editiert und danach erneut compiliert werden. Bei fehlerfreier Uebersetzung erscheint z.B. folgendes Bild:



Es werden die Anzahl der uebersetzten Quelltextzeilen, die Anzahl der Bytes und die Adressen fuer den Programmcode und den Datenbereich und den noch verbleibenden freien Speicherbereich angegeben.
Je nach festgelegter Option (Kommando O) wird die erzeugte File gespeichert

-im TPA (Hauptspeicher) (T) (Standard)
-als COM-File (Programm) (P)
-als CHN-File (Modul) (M)

(vergl. Ziffer 2.12.)

2.7. Testen (Ausfuehren)

Nach Eingabe von T wird das uebersetzte Programm aktiviert und gestartet.
Je nach Compiler-Option bei der Uebersetzung wird das Programm
-aus dem TPA (Hauptspeicher) (T)
-als COM-File von Diskette (P)
aktiviert und gestartet.
Wurde kein uebersetztes Programm gefunden, so uebersetzt automatisch der Compiler das Aktivfile und startet nach erfolgreicher Uebersetzung sofort auch das Programm.
Wurde noch kein Aktivfile angegeben, so wird zur Eingabe des Aktivfilenamens aufgefordert. Existiert auch das Aktivfile selbst noch nicht auf dem Datentraeger, so wird wie bei Kommando C der Editor gestartet.

2.8. Sichern

Nach Eingabe von S wird das Aktivfile ueber das entsprechende Laufwerk auf Diskette geschrieben.
Falls bereits auf der Diskette ein Programm mit gleichem Filenamen existiert, wird dies in ein BAK-File umgewandelt und die neue Version unter dem urspruenglichen Namen eingetragen.
Vor Operationen, die den Editerpuffer zerstoeren (Kommando K,B) wird fuer geaenderte Pufferinhalte das Sichern des Quelltextfiles ueberwacht und gegebenenfalls angeboten.

2.9. Kommandoausfuehrung

Nach Eingabe von K kann ein beliebiges SCPX-Programm oder mit "PLUS" der Systemservice gerufen werden (vergl.Ziffer 3). Endet das SCPX-Programm mit normalem Ausgang zum SCPX, so kehrt die Steuerung unmittelbar zum Systemkern zurueck, d.h. es wird

PASCAL.COM und die benoetigten Files geladen, einschliesslich der zuletzt bearbeiteten Aktivfiles
Dann erscheint der Prompt >. Ein nachfolgendes <ET> bringt das Grundmenue zurueck, es kann aber auch mit jedem anderen Kommando wie T,E,C usw. fortgesetzt werden. Falls zwischenzeitlich die Diskette mit dem Systemkern aus dem betreffenden aktuellen Laufwerk entfernt wurde, wird der Nutzer aufgefordert die Systemdiskette wieder einzulegen und mit <ET> zu starten:

```
 /------------------------------------------------------------\
¦ PASCAL.COM nicht gefunden. Bitte in Laufwerk <x>: und <ET>:  ¦
¦                                                            ¦
 \------------------------------------------------------------/
```

Danach erfolgt das Laden und Starten des Systemkerns wie beschrieben. Aus dieser Verfahrensweise folgt, dass das System bereits mit einem Laufwerk arbeitsfaehig ist.

2.10. Directory-Anzeige

Nach Eingabe eines D erfolgt die Aufforderung zur Eingabe einer Maske, die die auszugebenden Programmnamen steuert.
Die Maske wird in der SCPX ueblichen Weise mit * und ? gebildet und mit <ET> abgeschlossen. Wird nur <ET> eingegeben, dann wird die gesamte Directory des angegebenen Laufwerkes auf dem Bildschirm ausgegeben:

```
 /------------------------------------------------------------\
¦ >D                                                         ¦
¦ Maske: A:*.PAS                                             ¦
¦ HANDEL   PAS : PROBE    PAS : PRO1   PAS : PRO2   PAS      ¦
¦ PRO4     PAS : DRUCK    PAS                               ¦
¦                                                            ¦
¦ Byte  verfuegbar in A: 25K                                 ¦
¦ >                                                          ¦
 \------------------------------------------------------------/
```

Zum Abschluss wird der auf der Diskette noch freie Speicherbereich ausgegeben (hier 25K).

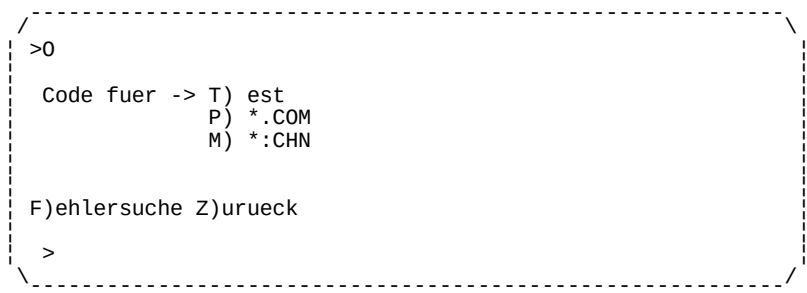
2.11. Beenden

Das Kommando B dient zum Verlassen des PASCAL-Systems und zur Rueckkehr zum SCPX. Es erscheint das SCPX-Prompt X> mit dem aktuellen Laufwerk X. Falls noch ein editiertes geladenes Aktivfile existiert, das nicht gesichert wurde, wird gefragt, ob das geschehen soll.

2.12. Compiler-Optionen

Mit diesem Kommando wird die Form des Ausgabefiles festgelegt oder kann nach einem bei der Abarbeitung eines COM-Files aufgetretenen Fehler dessen Stelle gefunden werden.

Nach der Eingabe des Kommandos O erscheint:



Die Stelle des Pfeiles zeigt das aktuelle Ziel des Compilates. Es kann durch Eingabe von T,P oder M veraendert werden. Mit F wird der Suchprozess zur Auffindung eines Fehlers im Quellprogramm gestartet und mit Z kann zum Grundmenue zurueckgekehrt werden. Start- und Endadresse koennen bei den Optionen P oder M mit S und E geaendert werden.

Die Moeglichkeiten im einzelnen:

T: (Standard) Der Code wird im Hauptspeicher (TPA) erzeugt und das Programm kann durch T(est) gestartet werden.

P: Unter dem Namen des Aktiv- bzw. Hauptfiles wird ein lauffaehiges Programm (COM-File) erzeugt, das den Code und die PASCAL-Bibliothek enthaelt. Gestartet wird das COM-File durch normalen SCPX-Aufruf oder durch die Kommandos T oder K. Diese Programme koennen groesser als bei der Option T sein, da ihre Anfangsadresse im TPA niedriger liegt und bei der Compilierung kein TPA fuer die Speicherung des Programmes benoetigt wird. Nichtgeladener Fehlertext vergroessert auch hier den Arbeitsbereich des Compilers. Es wird die Startadresse und Endadresse des Programmbereiches fuer den Code in COM-File ausgegeben. Sie koennen mit S bzw. E geaendert werden.

M: Unter dem Namen des Aktiv- bzw. Hauptfiles wird ein aufrufbarer Modul (CHN-File) erzeugt. Es enthaelt den Programmcode, aber nicht die PASCAL-Bibliothek. Diese Programme koennen nur von einem anderen Pascalprogramm, das mit der COM-Option uebersetzt wurde, durch die Chain-Prozedur aufgerufen werden. Damit besteht also die Moeglichkeit, auch sehr grosse Pascalprogrammpakete zu erzeugen. Die Start- und Endadresse des Programmbereiches wird fuer den Code im CHN-File ausgegeben. Sie koennen mit S bzw. E geaendert werden.

F: Wurde ein Programm nicht mit der T-Option uebersetzt kann die Stelle eines bei der Abarbeitung auftretenden Programmfehlers nicht automatisch im Quelltext gefunden werden, da zu diesem Zeitpunkt das PASCAL-System nicht ohne weiteres zur Verfuegung steht. Der Abbruch des Programmes erfolgt mit der Fehlermeldung

```
          Laufzeitfehler <nn>, PC = <Adresse>
          Programmabbruch
oder
          E/A-Fehler <nn>, PC = <Adresse>
          Programmabbruch
```

Dabei ist <nn> die Fehlernummer und die Hexadezimalzahl gibt die Fehlerstelle im Code an.

Allgemeine Laufzeitfehler (ausfuehrliche Darstellung im Anhang E)

(nn in der Meldung: Laufzeit Fehler nn, PC = <Adresse>)

nn	Bedeutung
01	Gleitkommaueberlauf
02	Division durch Null
03	Fehler im Argument von SQRT (<Null)
04	Fehler im Argument von LN (<=Null)
10	Falsche STRING-Laenge (auch in Ergibtanweisungen)
11	Fehlerhafter STRING-Index (ausserhalb 1 - 255)
90	Index ausserhalb des zulaessigen Bereiches
91	Ordinaler Typ ausserhalb des Wertebereiches (auch bei Teilbereichstypen)
92	Wert ausserhalb des INTEGER-Bereiches
FF	Halden/Kellerspeicher-Kollision

Ein-/Ausgabe-Laufzeitfehler (Vergl. Anhang F)

(nn in der Meldung: EA-Fehler nn, PC = <Adresse>)

nn	Bedeutung
01	File existiert nicht
02	File fuer Leseoperationen nicht vorbereitet
03	File fuer Schreiboperationen nicht vorbereitet
04	File nicht geoeffnet
10	Fehler im numerischen Format
20	Operation auf logischem Geraet nicht erlaubt
21	Im Direktmodus (Zielauswahl T) nicht erlaubt
22	ASSIGN fuer vordefinierte Filevariablen nicht erlaubt
90	Recordlaenge nicht vertraeglich
91	Position ausserhalb des File
99	Vorzeitiges Fileende
F0	Disketten-Schreibfehler
F1	Directory voll
F2	File zu gross
FF	File nicht unter Kontrolle

Um diese Stelle im Quellprogramm zu finden, ist dieses mit A oder H zu laden. Dann muss das F-Kommando gegeben werden. Es wird die Eingabe der Fehleradresse gefordert:

```
      Stand PC:1B56 <ET>
```

Die Eingabe der Fehleradresse (hier 1B56) fuehrt zum Suchprozess im Quellprogramm. Wurde die Fehlerstelle gefunden, wird dies mitgeteilt und zum Druecken der Taste ESC-> aufgefordert. Danach erscheint das Quellprogramm und der Cursor steht hinter dem Sprachelement, das den Fehler verursachte. Wird bei der Compiliert eine Ueberlagerungsstruktur erzeugt (Overlay), so kann ein Fehler nicht eindeutig lokalisiert werden. Die Adresse wird in dem Programmteil gesucht, der in dem Ueberlagerungsbereich als erstes compiliert wurde.

S: Die Startadresse gibt die Adresse des ersten Bytes vom Code an. Das ist normalerweise die Adresse unmittelbar hinter der Laufzeit-Bibliothek. Diese Adresse kann nach oben erhoeht werden, um Platz z.B. fuer absolute Variablen, die von geketteten Programmen (CHN-Files) verwendet werden, zu schaffen. Nach Eingabe des Kommandos S erscheint die Aufforderung zur Eingabe der Startadresse. Sie wird hexadezimal eingegeben und mit <ET> abgeschlossen. Wird nur <ET> eingegeben, so bleibt die alte Adresse erhalten.

CHAIN-Files und rufende COM-Files sind mit gleicher Startadresse zu uebersetzen.

E: Die Endadresse gibt die hoechste genutzte Adresse des compilierten Programms an. Der Wert in Klammern ist die TPA-Grenze (BDOS-Anfang). Die Eingabe der Endadresse ist hexadezimal. Sie wird mit <ET> abgeschlossen. Wird nur <ET> eingegeben, dann bleibt die alte Adresse erhalten.

Der freie Speicher eines Betriebssystems von z.B. 48K (BDOS=C106) kann noch kleiner werden, wenn Betriebssystemerweiterungen wie Tastenumcodierung und zusaetzliche Ein-/Ausgabetreiber untergeladen werden. Deshalb wird empfohlen, die Endadresse stets niedriger einzustellen (z.B. A000), damit das Programm moeglichst unter allen SCP-Betriebssystemen einschliesslich seinen Erweiterungen lauffaehig ist.

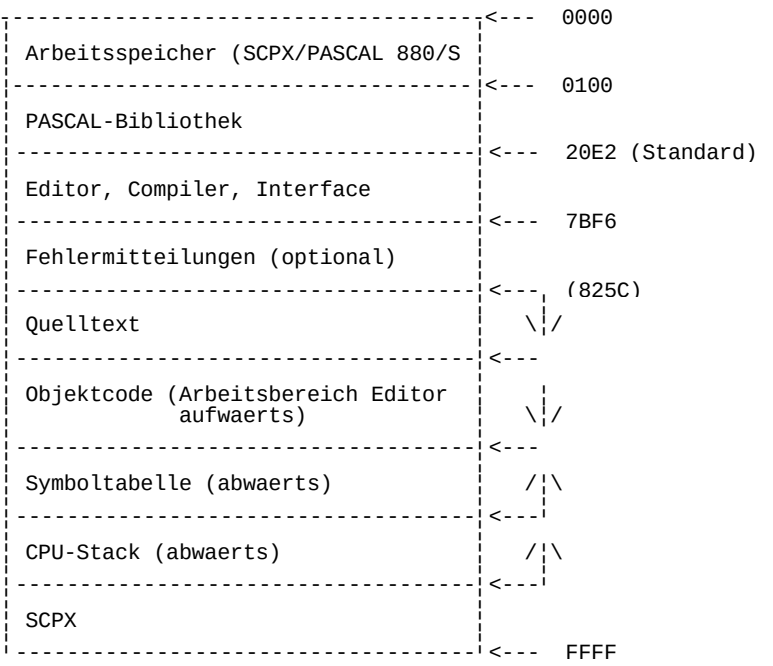
Das Laufzeitsystem des compilierten Programms prueft bei Programmstart die Endadresse gegen die aktuelle TPA-Grenze. Bei Unvertraeglichkeit erscheint die Fehlermeldung "Speicher zu klein, Programmabbruch".

2.13. Hauptspeicherregime

Die folgenden Abbildungen zeigen den Speicherinhalt bei verschiedenen Arbeitsmodi. Einige Grenzen veraendern sich waehrend der Laufzeit (durch die Groesse des Quelltextes, dynamische Variablen,usw.). Die Groesse der Segmente in den Abbildungen haben keinen Bezug zur Groesse der Speicherbereiche.

2.13.1. Compilierung im Speicher

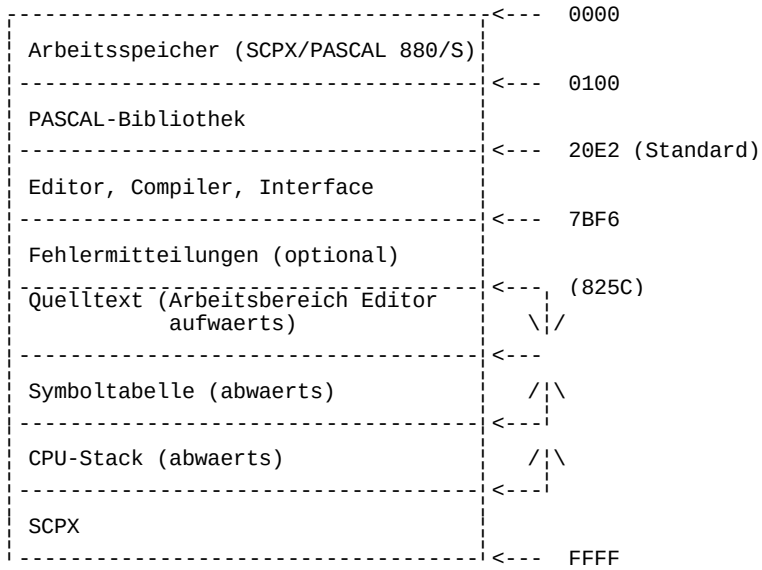
Waehrend der Compilierung eines Programmes im Speicher (Option T) ist der Speicher wie folgt aufgeteilt:



Wurden die Fehlermitteilungen beim Aufruf von PASCAL 880/S nicht geladen, so kann der Quelltext bereits bei der Adresse $7BF6 beginnen. Beim Aufruf des Compilers wird der Objektcode hinter dem aktuellen Quelltext aufwaerts aufgebaut.
Der CPU-Stack arbeitet vom logischen Speicherende abwaerts, und die Symboltabelle des Compilers wird abwaerts von einer Adresse, die 1K($400 Bytes) unterhalb des logischen Speicherendes liegt, aufgebaut.

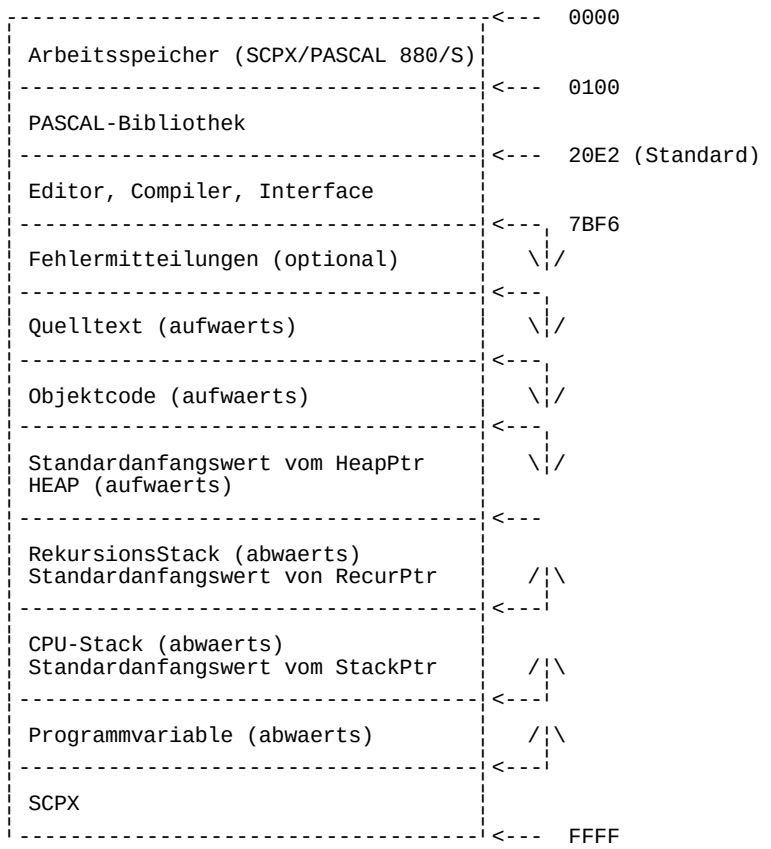
2.13.2. Compilierung auf Diskette

Waehrend der Compilierung zu einem .COM-File oder .CHN-File (Option P oder M) ist die Aufteilung des Speichers anders als bei der Compilierung im Speicher. Der erzeugte Objektcode steht nicht im Speicher, sondern wird ueber einen relativ kleinen Pufferspeicher sofort auf der Diskette abgelegt.
Ausserdem beginnt der Code auf einer niedrigeren Adresse (unmittelbar. hinter der PASCAL-Bibliothek anstatt nach dem Quelltext). Daher koennen in diesem Modus viel groessere Programme uebersetzt werden.


Arbeitsspeicher (SCPX/PASCAL 880/S)
0000
PASCAL-Bibliothek
0100
Editor, Compiler, Interface
20E2 (Standard)
Fehlermitteilungen (optional)
7BF6
Quelltext (Arbeitsbereich Editor aufwaerts)
(825C)
Symboltabelle (abwaerts)
CPU-Stack (abwaerts)
SCPX
FFFF

2.13.3. Ausfuehrung im Speicher

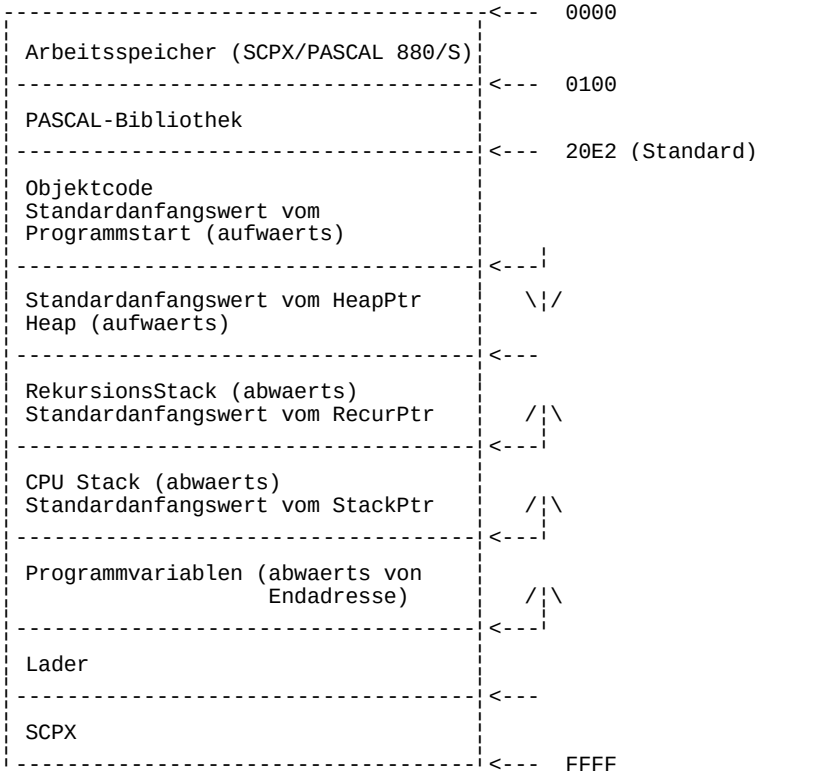
Wenn ein Programm im Direktmodus (Option T) ausgefuehrt wird, sieht die Speicheraufteilung wie folgt aus:



Wenn ein Programm uebersetzt wurde, ist das Ende des Objektcodes bekannt. Standardmaessig wird auf diesen Wert der Heap-Pointer gestellt und der Heap aufwaerts in Richtung Rekursionsstack aufgebaut. Die maximale Speichergroesse ist BDOS-Anfang minus 1 (wird im Compiler-Option-Menue angezeigt). Von dieser Adresse werden die Programmvariablen abwaerts gespeichert in Richtung des Endes vom freien Speicherplatz, auf dem der Wert CPU-Stackpointer STACKPTR gestellt wird. Von hier aus wird der CPU-Stackpointer in Richtung RekursionStack aufgebaut, wobei RECURPTR $400 Bytes niedriger als STACKPTR liegt. Der RekursionStack wird von dieser Stelle in Richtung HEAPPTR (also abwaerts) aufgebaut.

2.13.4 Ausfuehrung eines Programmes als File

Wenn ein Programmfile abgearbeitet wird (entweder ueber K oder direkt von SCPX aus), sieht die Speicheraufteilung wie folgt aus:



Die Standard-Startadresse des Programmes (im Compiler-Option-Menue angegeben) beginnt unmittelbar hinter der PASCAL-Laufzeitbibliothek. Dieser Wert kann mit dem Startadresse-Kommando im Compiler-Option-Menue veraendert werden, um beispielsweise Platz fuer absolute Variablen zu schaffen und/oder externe Prozeduren zwischen Bibliothek und Code einzufuegen. Die maximale Speichergroesse ist BDOS minus 1, und der Standardwert wird durch die Laenge von BDOS im Speicher bestimmt. Wenn Programme fuer andere Rechner uebersetzt werden, muss dies beachtet werden, um Kollisionen mit dem BDOS auf den anderen Rechnern zu vermeiden. Die maximale Speichergroesse kann mit dem E)ndadresse-Kommando im Compiler-Option-Menue veraendert werden. Man sollte beachten, dass die Standard-Endeadresse etwa 700 bis 1000 Byte niedriger liegt als die maximale Speichergroesse, um Platz fuer den Lader zu lassen, der benoetigt wird, wenn das Programm vom Systemkern mittels K-Kommando gestartet werden soll. Dieser Lader holt die PASCAL-Bibliothek und das Interface erneut in den Speicher, nachdem das Programm abgearbeitet wurde. Dadurch kann zum Systemkern zurueckgekehrt werden.

3. Systemservice (Dienstprogramme)

3.1. Start und Menue

Der Start erfolgt ueber das Grundmenue des Systemkerns (Kommandobuchstabe K) oder von SCPX aus durch die Eingabe

```
PLUS
oder
PLUS <Buchstabe>.
```

Nach dem Kommando PLUS erscheint das Servicemenue:

```
 ,-----------------------------------------.
/                                           \
| PASCAL 880/S (c) VEB ROBOTRON BWS         |
| Systemservice Version <n>.<m> /SCPX[<x> KB]
| L)aufwerk:  <x>                           |
|                                           |
| K)opieren   E)rase   R)ename   C)hiffrieren
| D)rucken    M)oduln  S)tatus   V)erdichten
| Z-urueck                                  |
|                                           |
| >                                         |
|                                           |
|                                           |
\                                           /
 `-----------------------------------------'
```

Copyright, Bezeichnung und Versionsnummer und verwaltete TPA-Groesse werden nur bei Neustart sichtbar.
Einer der angegebenen Kommandobuchstaben ist zu waehlen, sonst bleibt das Menue (ohne Kopf) erhalten.
Das kann spaeter (wie im Systemkern) mit <ET> genutzt werden, um sich ueber die aktuellen Parameter zu informieren.
Es bedeuten:

L Laufwerkwechsel. Es besteht die Moeglichkeit zum Ruecksetzen des Diskettensystems und zur Wahl eines anderen aktuellen Laufwerks. Kommunikation und Wirkung sind wie im Grundmenue des Systemkerns.

K Kopieren von Files beliebigen Typs.

E Loeschen von nicht schreibgeschuetzten Files.

R Umbenennen von nicht schreibgeschuetzten Files.

C Chiffrieren eines nichtschreibgeschuetzten Files so, dass das entstehende File ohne Kenntnis des verwendeten Passworts nicht verwendbar ist.
Ist das File bereits chiffriert, wird es dechiffriert.

D Drucken von Textfiles mit wahlweiser, umfassender Druckaufbereitung.
Das Textfile bleibt unveraendert.

M Moduldiagramm drucken.
Das ist eine gegliederte Uebersicht ueber alle Programmeinheiten eines Programms einschliesslich ihrer Schachtelungen.

S Status aendern.
Schreibgeschuetzte Files werden in den Schreibzustand versetzt, nichtschreibgeschuetzte Files erhalten einen Schreibschutz.

V Verdichten von PASCAL-Quelltexten
Das Quelltextfile belegt danach etwa 50% des sonst erforderlichen Speicherplatzes. Es kann in dieser Form nicht compiliert werden. Ist das File bereits verdichtet, wird es in den normalen Zustand zurueckgesetzt.

Z Der Systemservice wird verlassen.
Es wird zum Systemkern zurueckgekehrt. erfolgte der Start von SCPX aus, wird zu diesem zurueckgekehrt.

B Beenden des Systemservice und Rueckkehr zum Laufzeitsystem SCPX

Die Funktion V steht erst ab Version 1.4 zur Verfuegung.

Erfolgte der Start mit

PLUS <Buchstabe>

so wird fuer den Fall, dass <Buchstabe> K, E, R, C, D, M, S, V ist, die jeweilige Funktion ausgefuehrt. Es erscheint kein Servicemenue am Anfang. Ist <Buchstabe> kein gueltiges Kommando (oder Z), wird wie beim Kommando PLUS gestartet.
Alle Kommandos werden sichtbar mit Langtext quittiert. Wurde mit dem Hilfsprogramm PASINST (vergl. Ziffer 5) ein Kopierschutz installiert, meldet sich der Systemservice nach K mit

Kopierschutz. Codewort:

Es ist das generierte Codewort einzugeben und mit <ET> abzuschliessen. Die Eingabe erfolgt ohne Echo auf dem Bildschirm. Ist das Codewort richtig, wird die Kopierroutine gerufen, sonst zum Servicemenue zurueckgekehrt. Ein installierter Printschutz fuehrt nach D zu der Ausschrift

Printschutz. Codewort:

Es ist wie beim Kopierschutz zu verfahren.

Wurde nicht K oder D eingegeben, der Schutz nicht installiert oder das richtige Codewort eingegeben, werden die Files der im aktuellen Laufwerk befindlichen Diskette wie folgt angezeigt (Anzeigemenue, Beispiel):

```
 /-------------------------------------------------------\
¦  1 = PASCAL   COM*   2 = PASCAL  RES*  3 = PASCAL   TXT*  ¦
¦  4 = PLUS     COM    5 = TEST    PAS^  6 = PROBE    SCP   ¦
¦  7 = SORTIERT PAS+                                       ¦
¦                                                         ¦
¦  Auswahl:                                               ¦
¦                                                         ¦
¦                                                         ¦
 \-------------------------------------------------------/
```

Die Namen sind alphabetisch sortiert.
Es kennzeichnen die Zusaetze

*	Schreibschutz
+	verdichtet
^	chiffriert.

Erlaubte Eingaben sind:

a) Zahlen im Bereich der Anzeige (im Beispiel 1 bis 7),
b) Bindestrich im Sinne "bis"
c) beliebig viele Leerzeichen
in der maximalen Laenge von 80 Zeichen.

Erlaubt sind hier zum Beispiel die Eingaben

1-	fuer 1 2 3 4 5 6 7
1 2 4-	fuer 1 2 4 5 6 7
- 5	fuer 1 2 3 4 5

Die Eingabe eines Sterns ist nicht erlaubt. Eine gueltige Eingabe oder eine Abweisung mit nur <ET> werden erzwungen und Fehler angezeigt.
<ET> nach Auswahl fuehrt zurueck ins Servicemenue, das jedoch nicht sofort angezeigt wird.
Die folgenden Funktionen werden fuer die ausgewaehlten Files durchgefuehrt.

3.2. Kopieren

Es erfolgt die Eingabeaufforderung

  Ziellaufwerk:

fuer die Kopie. Die Eingabe wird geprueft auf den Bereich A..F und muss vom aktuellen Laufwerk verschieden sein. Die Fehler werden mit "Kommandofehler" angezeigt. Nur <ET> fuehrt zur Ablehnung der Funktion.
Existiert ein File mit dem Namen der Quelle bereits auf der Diskette im Ziellaufwerk, erfolgt die Ausschrift

  U)eberschreiben, D)oppelt-File, W)eiter:

und der Nutzer muss sich zwischen diesen Moeglichkeiten entscheiden. Eine der Antworten wird erzwungen. Wird D entschieden, erhaelt das kopierte File auf der Zieldiskette die Namenserweiterung DPL. Bei W wird das File uebergangen. U ist nur moeglich, wenn das zu ueberschreibende File nicht schreibgeschuetzt ist. Der Fehler wird mit File <Filename> schreibgeschuetzt mitgeteilt.

Eine erfolgreiche Kopie wird fileweise mit dem Text

  <Ziellaufwerk>: <-- <Quellaufwerk>: <Filename>

quittiert. Sonst erfolgt die Ausschrift

  Laufzeitfehler.

3.3. Erase

Die ausgewaehlten Files im aktuellen Laufwerk werden geloescht, vorausgesetzt sie sind nicht schreibgeschuetzt. Im letzteren Fall werden sie mit

  File <Filename> schreibgeschuetzt.

abgewiesen.

Mit S kann der Schreibschutz aufgehoben werden. Aus Sicherheitsgruenden wird fuer jedes ausgewaehlte File gefragt

  <Filename> loeschen (J/N):

und eine gueltige Antwort erzwungen. Bei N bleibt das File erhalten, bei J wird es geloescht. Das Loeschen wird mit

  <Filename> geloescht.

quittiert.

3.4. Rename

Die ausgewaehlten Files werden umbenannt. Voraussetzung ist, dass sie nicht schreibgeschuetzt sind.
Im letzteren Fall werden sie mit

  File <Filename> schreibgeschuetzt.

abgewiesen.

Je File wird mit

  Neuer Filename:

zur Eingabe des neuen Namens aufgefordert. Der Name darf keine Laufwerksangabe enthalten und nicht bereits auf dieser Diskette vorhanden sein. Bei nur <ET> wird das File uebergangen und der alte Filename bleibt erhalten. Es wird eine gueltige Eingabe erzwungen. Gegebenenfalls erfolgt die Fehlermitteilung

  <Filename> existiert bereits.

Die erfolgreiche Umbenennung wird mit

  <neuer Filename> <-- <alter Filename>

auf dem Bildschirm quittiert. Ist die Umbenennung nicht moeglich, erfolgt die Mitteilung

  Laufzeitfehler.

und das File wird uebergangen.

3.5. Chiffrieren

Die ausgewaehlten Files werden anhand eines vom Nutzer eingegebenen Codewortes verschluesselt, wenn sie bisher im Klartext vorlagen oder entschluesselt, wenn sie bisher verschluesselt waren (Kennzeichnung "^" im Anzeigemenue).
Schreibgeschuetzte Files werden mit

File <Filename> schreibgeschuetzt.

abgewiesen.

Fuer alle Files erfolgt die Anforderung des Schluessels mit

Codewort:

Die maximale Laenge des Codewortes ist 16 Zeichen. Bei der Eingabe erfolgt kein Echo.

Achtung! Fuer die Auswahl des Codewortes beim Chiffrieren und spaeter desselben Codewortes beim Dechiffrieren ist der Anwender selbst verantwortlich. Wird das Codewort vergessen, ist das File verloren. Es empfehlen sich deshalb Codeworte, die sich der Anwender leicht merken kann (Name, Vorname, Geburtstag, Einrichtung). Wurde beim Dechiffrieren ein falsches Codewort benutzt, so ist zunaechst mit dem falschen Codewort wieder zu Chiffrieren und dann mit dem richtigen Codewort zu dechiffrieren.

Nur <ET> fuehrt zur Ablehnung der gesamten Funktion Chiffrieren. Nach Eingabe des Codewortes (Beenden mit <ET>) erfolgt die Ausschrift

Bitte warten.

und das File wird transformiert. Dazu wird aus Sicherheitsgruenden zunaechst ein temporaeres File angelegt. Die Diskette darf also nicht voll und nicht schreibgeschuetzt sein. In diesem Fall erfolgt die Mitteilung

Laufzeitfehler.

Der erfolgreiche Vorgang wird je nach Richtung mit

<Filename> chiffriert.

oder

<Filename> dechiffriert.

quittiert. Der Anwender kann sich auch durch erneutes Rufen des Anzeigemenues davon ueberzeugen. Das File traegt das Kennzeichen "^", bzw. dieses Zeichen ist aufgehoben). Mit dem Editor oder mit TYPE kann ebenfalls der Erfolg kontrolliert werden. Allerdings enthalten chiffrierte Files nicht druckbare Zeichen.

3.6. Drucken

Die ausgewaehlten Textfiles werden ueber Drucker ausgegeben. Der Anwender kann die Form des Druckes wesentlich beeinflussen. Das File selbst bleibt davon unberuehrt. Fuer alle gewaehlten Files erscheint das Menue:

```
/~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~\
|                                                                    |
|  Formatierung:                                                     |
|                                                                    |
|     Z)eilen numerieren                                             |
|     A)bsaetze zusammenhalten                                       |
|     R)eservierte Worte markieren                                   |
|     S)chachtelungstiefe anzeigen                                   |
|     E)inruecken                                                    |
|     I)nclude-Files drucken                                         |
|     C)rossreferenzliste drucken                                    |
|     K)opfzeile ergaenzen                                           |
|     B)EGIN/END ausziffern                                          |
|     Q)uellfilename unterdruecken                                   |
|                                                                    |
|  Auswahl (ZARSEICKBQ / <ET> fuer Z ):                              |
\--------------------------------------------------------------------/
```

Durch Angabe des jeweils gekennzeichneten Buchstabens koennen die Optionen zusammengestellt werden. Falls nur Zeilennumerierung gewuenscht wird, genuegt <ET>.
Werden weitere Optionen gewuenscht, muessen alle Auswahlmoeglichkeiten (also auch Z) angegeben werden, wobei ihre Reihenfolge keine Rolle spielt. Die Optionen bewirken:

Z Zeilen numerieren.
Fuer Include-Files kann statt der durchgaengigen eine gesonderte Zaehlung vorgenommen werden. Deshalb folgt in Verbindung mit der Option 'I)nclude-Files drucken' die Frage: 'Zeilen des Include-File gesondert numerieren (J/N):'.Bei 'N' wird wie bisher weiter gezaehlt, bei 'J' wird die Numerierung des Haupt-Files unterbrochen und das Include-File in der Form [nn:] numeriert. Wurde mit C die Crossreferenz angefordert, kann keine gesonderte Numerierung erfolgen.

A Absaetze zusammenhalten. Diese Option verhindert, dass Programmabschnitte zwischen Leerzeilen durch Seitenvorschub getrennt werden.
In diesem Fall erfolgt der Seitenvorschub bereits nach der vorangestellten Leerzeile.

R Reservierte Worte markieren. Die Markierung erfolgt durch Grossschreibung und Doppeldruck der entsprechenden Worte. Markiert werden Wortsymbole und Standardtypbezeichner.

S Schachtelungstiefe anzeigen. Die Schachtelungstiefe wird vor der entsprechenden Quelltextzeile angegeben.

E Einruecken. Es werden zwei Stellen je Schachtelungstiefe vor dem PASCAL-Text eingefuegt. Die vom Nutzer editierten fuehrenden Leerzeichen werden ignoriert.

I Include-Files drucken. Files, die im Quelltext durch die Include-Compilerdirektive vertreten sind, koennen durch diese Option mitgedruckt werden.
Auf Anforderung (Vergl. Z) koennen die Zeilen gesondert numeriert werden.

C Crossreferenzliste ausgeben. Es erfolgt eine alphabetische Auflistung der im Quelltext gedruckten Variablen, Funktions- und Prozedurbezeichner und die Angabe der Zeilen, in denen sie auftreten.
Eine gesonderte Zeilennumerierung der Include-Files und die Crossreferenzliste schliessen sich aus
Die Crossreferenzliste hat Prioritaet. Uebersteigt eine uebergrosse Crossreferenzliste (bei sehr grossen Programmen und 48 KB TPA) die verfuegbare Hauptspeichergroesse, wird die Funktion C abgeschaltet. Das wird mit dem Text

  Ueberlauf: Crossreferenzliste unterdrueckt.

auf dem Bildschirm angezeigt.

K Kopfzeile ergaenzen. Es erfolgt die Aufforderung:

  Ergaenzung der Kopfzeile (max. 40 Zeichen):

Die Kopfzeile wird auf jede Seite geschrieben. Besonders eignen sich Datum oder Name des Programmierers.

B BEGIN/END ausziffern. Das Ausziffern erfolgt zwischen BEGIN/RECORD/CASE und END in'{ }'. Dabei erhalten die zusammengehoerigen Worte dieselbe Zahl. Bei richtiger Klammerung muesste demzufolge das letzte END die Nummer {1} haben.

Nach beendeter Auswahl erscheint auf dem Bildschirm die Aufforderung

  Bitte Blattanfang einstellen und <ET>:

Nach Eingabe von <ET> beginnt der Listvorgang.
Es kann mit ^C unterbrochen werden.
Vor dem Druck des naechsten Files erfolgt ein Blattvorschub.
Zusaetzlich besteht die Moeglichkeit, ab Version 1.5. im Quelltext durch Lister-Direktiven die Druckausgabe zu beeinflussen.
Die Lister-Direktiven haben die allgemeine Form:

  {.<Kommando>}{, <Kommando>}

wobei die aeusseren geschweiften Klammern mitzuschreiben sind.
Folgende Kommandos sind moeglich:

Kommando	Wirkung	Voreinstellung
SL <nn>	Seitenlaenge	75
RO <nn>	Rand oben	3
RU <nn>	Rand unten	3
RL <nn>	Rand links	0
KV <nn>	Neue Seite bei weniger als n Zeilen	0
NS	Neue Seite	
L+/L-	Listing ein/aus	
A+/A-	Autoblock ein/aus	

Es bewirken

SL <nn>: Veraenderung der Seitenlaenge auf nn Zeilen.
KV <nn>: Durch dieses Kommando kann ein Seitenvorschub an jeder beliebigen Stelle der Seite erreicht werden. Dabei wird mit der Anzahl nn die Mindestanzahl der noch freien Zeilen.
NS : Im Unterschied zu 'SL nn' wird hier die festgelegte Seitenlaenge nicht veraendert, sondern ein Vorschub auf die neue Seite vorgenommen, unabhaengig von der Anzahl der noch freien Zeilen.
L+/L-: Durch 'L-' kann das Drucken unterbrochen und durch 'L+' wieder eingeschaltet werden. Standardeinstellung ist 'L+'. 'L-' unterbricht das Drucken ab der Zeile, die dem Kommando folgt. Durch 'L+' wird der Listvorgang ab der darauffolgenden Zeile fortgefuehrt.
A+/A-: Die Wirkung ist wie A (Absaetze zusammenhalten) im Druckermenue. Eine dort getroffene Festlegung kann mit A+ (ein) und A- (aus) das Zusammenhalten der Absaetze hier noch separat gesteuert werden.

Die Ausfuehrung der Funktion D wird auf dem Bildschirm mit

Bearbeitung --> <Filename>

protokolliert. Fehler werden mitgeteilt.

3.7. Moduln

Fuer die ausgewaehlten Files wird ein Strukturdiagramm der folgenden Form gedruckt:

```
<Filename> <Programmname>
<Filename> <Schachtelungsnomenklatur> <Teilprogrammname>
 .
 .
 .
```

Zum Beispiel:

```
PLUS.PAS    :  PROGRAM PLUS
PLUS.PAS    : 1.  FUNCTION MENUE
ANZIDT.PLS  : 2.  ANZEIGE
ANZIDT.PLS  : 2. 1.  PROCEDURE DIRECTORY
ANZIDT.PLS  : 2. 1. 1.  PROCEDURE SPEICHERN
ANZIDT.PLS  : 2. 2.  PROCEDURE ORDNEN
ANZIDT.PLS  : 3.  PROCEDURE IDENT
 .
 .
 .
```

Include-Files werden mit verarbeitet. Es muss sich nicht um Hauptprogramme handeln.
Am Anfang wird mit

Bitte Blattanfang einstellen und <ET>:

dazu aufgefordert, den Drucker bereitzumachen.

Es folgt dann fuer jedes ausgewaehlte File die Ueberschrift

  Moduldiagramm fuer <Name der Programmeinheit>

Da der Vorgang relativ zeitaufwendig ist, werden die Filenamen der jeweils bearbeiteten Files auf dem Bildschirm mit

  Bearbeitung --> <Filename>

angezeigt.

Moegliche Fehlermitteilungen sind

  Include-File <Filename> nicht gefunden.

und

  Laufzeitfehler

fuer den Datenverkehr mit der Diskette.

3.8. Status

Der Status (Schreib/Lese- oder nur Lese-Status) der ausgewaehlten Files wird in sein Gegenteil geaendert.

Der Vorgang wird mit

<Filename><Schutzzeichen neu> <-- <Filename><Schutzzeichen alt>

quittiert oder durch

  Laufzeitfehler.

die Unmoeglichkeit der Ausfuehrung angezeigt. Das Schutzzeichen ist '*' bei Schreibschutz oder Leerzeichen bei Nicht-Schreibschutz.

3.9. Verdichten

Die ausgewaehlten Files muessen Textfiles sein und die Verdichtung ist nur effektiv, wenn es sich um PASCAL-Quelltext handelt. Die Einsparung betraegt etwa 50%, wird aber erst bei Veraenderung der Anzahl der Aufzeichnungsbloecke mit jeweils 2 (oder 1) KB wirksam. Fuer schreibgeschuetzte Files erfolgt die Ablehnung mit

  File <Filename> schreibgeschuetzt.

Sonst wird die Verdichtung ueber ein temporaeres File vorgenommen. Der dafuer erforderliche Platz muss auf der Diskette vorhanden sein. Das Quellfile wird uebersehrieben. Ist das File bereits verdichtet, erfolgt eine Rueckgewinnung.

Der erfolgreiche Vorgang wird mit

  File <Filename> verdichtet.

oder

  File <Filename> zurueckgewonnen.

quittiert.

Verdichtete Files sind mit Anzeigemenue durch "+" gekennzeichnet, so dass eine Kontrolle moeglich ist. Das verdichtete File enthaelt nichtdruckbare Zeichen.
Die Funktion V steht erst ab Version 1.4. zur Verfuegung.

4. Sprachbeschreibung

4.1. Grundelemente

4.1.1. Beschreibungsform (Metasprache)

Die Beschreibung der Sprache erfolgt in der erweiterten BACKUS-NAUR-Form.

Folgende Symbole werden zur Beschreibung verwendet:

Zeichen/Zeichenfolgen ohne die Klammerung	
<>, ausser ::= und ¦	= Terminalsymbole; die Zeichen sind unverändert in den PASCAL-Quelltext zu Uebernehmen.
<Zeichenfolge>	= Nichtterminalsymbole; an anderer Stelle definiert; durch eine gueltige Konstruktion zu ersetzen.
¦	= Alternative; ein Element muss gewaehlt werden.
{}	= Moegliche Wiederholung einer Konstruktion - kann auch weggelassen werden.
::=	= "..ist definiert durch.."
<leer>	= Leere Symbolfolge.

4.1.2. Grundsymbole

Das Grundvokabular besteht aus Grundsymbolen, die zu folgenden Klassen zusammengefasst sind:

```
<Buchstaben>    ::= 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¦

<Ziffern>       ::= 0¦l¦2¦3:4¦5¦6¦7¦8¦9
                    A¦B¦C¦D¦E¦F       {nur im Hex- Format}

<Sonderzeichen> ::= +¦-¦*¦/¦=¦^¦<¦>¦(¦)¦[¦]¦{¦}¦
                    .¦,¦:¦;¦'¦@¦$¦#¦&¦!¦_¦
```

4.1.3. Morpheme

4.1.3.1. Wortsymbole

Folgende Worte sind fest definiert und duerfen nur fuer die entsprechenden Zwecke verwendet werden. Mit Stern versehene Worte sind nicht in Standard-Pascal enthalten:

*ABSOLUTE	AND	ARRAY	BEGIN	CASE	CONST
DIV	DO	DOWNTO	ELSE	END	*EXTERNAL
FILE	FORWARD	FUNCTION	GOTO	IF	IN
*INLINE	LABEL	MOD	NIL	NOT	OF
OR	OVERLAY	PACKED	PROCEDURE	PROGRAM	RECORD
REPEAT	SET	*SHL	*SHR	*STRING	THEN
TO	TYPE	UNTIL	VAR	WHILE	WITH
*XOR					

4.1.3.2. Standardbezeichner

PASCAL 880/S verwendet eine Anzahl von Standardbezeichnern als Namen fuer Konstanten, Typen, Variablen, Prozeduren und Funktionen. Diese Standardbezeichner duerfen nicht umdefiniert werden. Als Standardbezeichner werden verwendet:

ABS	ADDR	ARCTAN	ASSIGN	AUX	AUXINPTR
AUXOUTPTR	BDOS	BDOSHL	BLOCKREAD	BLOCKWRITE	BIOS
BIOSHL	BOOLEAN	BUFLEN	BYTE	CHAIN	CHAR
CHR	CLOSE	CLREOL	CLRSCR	CON	CONINPTR
CONOUTPTR	CONCAT	CONSTPTR	COPY	COS	DELLINE
DELAY	DELETE	DISPOSE	EOF	EOLN	ERASE
EXIT	EXECUTE	EXP	FALSE	FILEPOS	FILESIZE
FILLCHAR	FLUSH	FRAC	FREEMEM	GETMEM	GOTOXY
HALT	HEAPPTR	HI	IORESULT	INPUT	INSLINE
INSERT	INT	INTEGER	KBD	KEYPRESSED	LENGTH
LN	LO	LST	LSTOUTPTR	MARK	MAXINT
MAXAVAIL	MEM	MEMAVAIL	MOVE	NEW	ODD
ORD	OUTPUT	OVRDRIVE	PI	PARAMCOUNT	PARAMSTR
PORT	POS	PRED	PTR	RANDOM	RANDOMSIZE
READ	READLN	REAL	RECURPTR	RELEASE	RENAME
RESET	REWRITE	ROUND	SEEK	SEEKEOF	SEEKEOLN
SIN	SIZEOF	SQR	SQRT	STACKPTR	STR
SUCC	SWAP	TEXT	TRUC	TRUE	TRUNC
USR	UPCASE	VAL	WRITE	WRITELN	

4.1.3.3. Spezialsymbole

Folgende Spezialsymbole gelten:

Indexklammern:	[]
Ausdrucks- und Funktions-/Prozedur-klammern:	()
Zeigermarkierung:	^
Kommentar- und Direktivklammern:	{}

Operatoren ohne Wortsymbole:

```
Arithmetische
Operatoren:             * ¦ - ¦ + ¦ /
Zuweisungsoperator:     :=
Vergleichsoperatoren:   <> ¦ <= ¦ >= ¦ < ¦ > ¦ =
Teilbereichsbegrenzer:  ..
```

```
Als Transkriptoren sind erlaubt:
Klammern:               (. .) gleichbedeutend mit [ ]
                        { } gleichbedeutend mit { }
```

4.1.3.4. Begrenzer

Sprachelemente muessen durch wenigstens einen der folgenden Begrenzer getrennt werden:

- <space>,
- Zeilenende,
- Kommentar.

4.1.3.5. Zeilenlaenge

Die maximale Laenge einer Programmzeile betraegt 127 Zeichen. Alle weiteren Zeichen werden ignoriert.

4.1.4. Nutzerdefinierte Sprachelemente

4.1.4.1. Bezeichner

Bezeichner werden zur Bezeichnung von Marken, Konstanten, Typen, Variablen, Prozeduren und Funktionen verwendet. Ihre Zuordnung muss in ihrem Gueltigkeitsbereich (z.B. innerhalb einer Prozedur) eindeutig sein.

Syntax:

```
 <Bezeichner>     ::= <Buchstabe>
                      ¦<Buchstabe>{<weitere Zeichen>
 <weitere Zeichen>::= <Buchstabe>
                      ¦<Ziffer>
                      ¦<Unterstreichungszeichen>
```

Ein Bezeichner besteht also aus einem Buchstaben, dem Buchstaben, Ziffern und Unterstreichungsstrich folgen koennen. Die Laenge ist maximal 127 Zeichen, und alle Zeichen sind signifikant. Dadurch sind Programme moeglich, die sich in hohem Masse selbst dokumentieren.

Beispiel:

```
     Pascal
     Preis
     Art_Nummer
     3teWurzel          falsch! Ziffer am Anfang.
     zwei Worte         falsch! Leerzeichen nicht erlaubt.
```

Zwischen grossen und kleinen Buchstaben gibt es keinen Unterschied. So sind

```
ArtikelNummer = ARTIKELNUMMER
```

identisch. Der linke Bezeichner ist leichter lesbar. Wortsymbole duerfen, Standardbezeichner sollten nicht als nutzerdefinierte Bezeicher verwendet werden.
Der Unterstreichungsstrich(_) ist in Bezeichnern zulaessig, wird aber vom Compiler ignoriert (z.B. Art_Nummer entspricht ArtNummer). Bezeichner koennen prinzipiell mit "@" beginnen.
Ein Bezeichner gilt immer als definiert im Block des jeweiligen Deklarationsteils. Im Hauptprogramm definierte Bezeichner sind immer global.

4.1.4.2. Zahlen

Zahlen sind Konstanten der Typen INTEGER, BYTE oder REAL. Integerzahlen sind ganze Zahlen, die dezimal und hexadezimal dargestellt werden koennen. Hexadezimale Integerzahlen werden durch vorangestellte $-Zeichen erklaert.
Integerzahlen haben einen Bereich von -32768 .. +32767.
Hexadezimalzahlen haben einen Bereich von $0000 .. $FFFF.
BYTE ist als Teilbereich 0..255 von INTEGER aufzufassen. Der Bereich der Realzahlen betraegt 1E-38 .. 1E+38 mit 11 signifikanten Ziffern. Die Exponentialdarstellung kann verwendet werden mit E als "mal 10 hoch". Eine Integerkonstante gilt ueberall dort, wo eine Realzahl gueltig ist. Trennzeichen duerfen nicht innerhalb von Zahlen stehen. Fuer Zahlen wird die uebliche Dezimaldarstellung genutzt. Unmittelbar vor einer Dezimalzahl darf ein Vorzeichen stehen.

Syntax:

```
<vzl. Zahl>::=
        <natuerliche Zahl>
        ¦ <vzl. Gleitkommazahl>
<vzl. Gleitkommazahl>::=
        <natuerliche Zahl> . <Ziffernfolge>
        ¦ <natuerliche Zahl> . <Ziffernfolge> E <Exponent>
        ¦ <natuerliche Zahl> E <Exponent>
<Exponent> ::=
        <natuerliche Zahl>
        ¦ <Vorzeichen><natuerliche Zahl>
<natuerliche Zahl>::=
        <Ziffernfolge>
<Ziffernfolge>::=
        <Ziffer> {<Ziffer>}
<Vorzeichen>::= + ¦ -
```

Der den Exponenten einleitende Buchstabe E steht fuer "10 hoch". Zahlen mit Dezimalpunkt haben vor dem Punkt mindestens eine Ziffer.

```
Beispiele:
  5
  62.12E+8
  0.691  (nicht.691)
  $3A
  $12G   Falsch! G keine Hexadezimalzahl.
  $12.3  Falsch! Punkt, keine Hexadezimalzahl.
  -345
  -1.2345678901E+12
  1       erlaubt, ist aber eine Integerkonstante.
```

4.1.4.3. Zeichenketten (Zeichenkettenkonstante)

Zeichenketten sind Folgen von Zeichen, welche in Apostrophe eingeschlossen sind. Zeichenketten sind Daten vom Typ CHAR bzw. STRING.
Syntax:

```
<Zeichenkette>::=
        ' <Zeichen> {<Zeichen>}
        ¦ ''
<Zeichen>::=
        <Buchstabe>
        ¦ <Ziffer>
        ¦ <Sonderzeichen>
```

Sollen innerhalb von Zeichenketten Apostrophe verwendet werden, dann ist das Apostroph zweimal zu schreiben.

```
Beispiele:
  'Zeichenkette '
  'Artikel-Nummer Menge '
  'Mach''s moeglich !'
  '63'
  ''                                    {= leere Zeichenkette}
```

Eine Zeichenkette ist kompatibel mit einem ARRAY OF CHAR gleicher Laenge und mit allen String-Typen gleicher oder groesserer Laenge.

4.1.4.4. CTRL-Steuerzeichen

PASCAL 880/S erlaubt die Verwendung von CTRL-Steuerzeichen als Zeichenketten. Dabei gibt es zwei Moeglichkeiten der Darstellung:

1) als #-Symbol, gefolgt von einer dezimalen oder hexadezimalen Zahl. Damit wird ein Zeichen mit dem entsprechenden Wert des Zeichensatzes definiert.
2) als ^-Symbol, gefolgt von einem Zeichen des Zeichensatzes. Damit wird das entsprechende CTRL-Zeichen definiert.

```
Beispiele:
     #10       entspricht     CTRL-J oder LINE FEED
     #$1B      entspricht     CTRL-[ oder ESCAPE
     ^G        entspricht     CTRL-G oder BELL
```

Folgen von Steuerzeichen koennen ohne Begrenzer aneinandergereiht werden:

Beispiele:
```
  #13#10
  #27^U#20
  ^G^G^G^G
```

Steuerzeichen koennen auch mit anderen Zeichenketten gemischt auftreten:

Beispiele:
```
     'Fehler! '^G^G^G'Bitte, korrigieren! '
```

4.1.4.5. Kommentare

Kommentare dienen der Erlaeuterung von Anweisungen oder Programmteilen. Kommentare sind nur Bestandteil des Quellprogramms, d.h., sie belegen keinen Speicherplatz zur Programmlaufzeit.
Kommentare koennen an jeder beliebigen Stelle im Programm stehen. Sie werden in {...} bzw.(*...*) eingeschlossen und koennen Buchstaben, Ziffern und Sonderzeichen enthalten (ausser }).
Sie sollten benutzt werden zur Kennzeichnung von Programmteilen oder zur Erlaeuterung spezieller, nicht sofort interpretierbarer Befehle.

Beispiel:

```
  .
  Zwischenspeicher := CONOUTPTR;
  CONOUTPTR :=LSTOUTPTR;  (Kanalumschaltung Bildschirm -->
                                                    Drucker}

  writeln('Diese Ausgabe erfolgt ueber Drucker');
  .
  CONOUTPTR := Zwischenspeicher;      {Rueckschaltung}
```

Es koennen in Kommentaren nicht Kommentare mit den gleichen Begrenzern eingeschlossen werden, aber es ist erlaubt, in Kommentaren mit { } Kommentare mit (* *) einzuschliessen und umgekehrt. Damit kann man Quelltexte in Kommentarklammern einschliessen und damit bei der Uebersetzung unberuecksichtigt lassen.

4.1.4.6. Compiler-Direktiven

Die Arbeitsweise des Compilers kann durch Direktiven gesteuert werden. Sie werden in den Quelltext als Kommentare mit einer speziellen Syntax eingefuegt. Die Direktiven koennen ueberall dort im Text stehen, wo Kommentare stehen koennen. Eine Compiler-Direktive besteht aus einer oeffnenden Kommentarklammer der unmittelbar ein $-Zeichen und dann die eigentliche Direktive oder eine Liste solcher Direktiven folgt, die durch Komma untereinander getrennt sind.

Beispiele:
```
  {$I-}
  {$I INCLUDE.PAS}
  {$R-,B+,V-}
  (*$S-*)
```

Achtung! Vor oder nach dem Zeichen $ ist kein Leerzeichen erlaubt. Ein +-Zeichen indiziert eine Aktivierung der Compiler-Direktive und ein --Zeichen indiziert eine Deaktivierung der Compiler-Direktive.

Alle Compiler-Direktiven haben Standardwerte. Diese wurden so ausgewaehlt, dass die Ausfuehrungszeit der Programme schnell und die Programmgroesse minimal ist. Dies bedeutet beispielsweise, dass die Codeerzeugung fuer rekursive Prozeduren und Ueberpruefung der Indexbereiche standardmaessig abgeschaltet ist. Man sollte deshalb genau pruefen, ob in den Programmen die benoetigten Compiler-Direktiven richtig gesetzt wurden.

INCLUDE-Direktive

Die allgemeine Form ist /1/

{$I <Filename>}

Das mit <Filename> bezeichnete File wird geladen und bearbeitet. Existiert es nicht, entsteht ein Laufzeitfehler. Include-Files duerfen nicht selbst wieder Include-Direktiven enthalten. Eine Include-Schachtelung ist also nicht erlaubt.

A-Compiler-Direktive

Standard: A+

Die A-Direktive steuert die Generierung von absolutem, d.h. nicht rekursivem Code. Wenn aktiv {$A+}, so wird ein absoluter Code generiert. Wenn passiv {$A-}, generiert der Compiler einen Code, der rekursive Aufrufe erlaubt. Diese Programme sind groesser und langsamer.

B-Compiler-Direktive

Standard: B+

Die B-Direktive steuert den Eingabe-/Ausgabe-Auswahlmodus. Wenn aktiv {$B+}, wird das CON:Geraet den Standard-Files INPUT und OUTPUT zugewiesen, d.h. dem Standard INPUT-OUTPUT-Kanal. Wenn passiv {$B-}, wird das TRM:Geraet zugewiesen. Diese Direktive ist global zum gesamten Block und kann im Programm nicht umdefiniert werden.

C-Compiler-Direktive

Standard: C+

Die C-Direktive steuert die Interpretation der Steuerzeichen bei Consol-I/O

/1/ In der allgemeinen Schreibweise fuer Direktiven sind geschweifte Klammern Terminalsymbole

I-Compiler-Direktiven

Standard: I+

Die I-Direktive steuert die Behandlung der I/O-Fehler. Wenn aktiv {$I+}, werden alle I/O-Operationen auf Fehler ueberprueft. Wenn passiv {$I-}, ist es notwendig, dass der Programmierer selbst die I/O-Fehler mit der Standardfunktion IORESULT prueft. Folgt der I-Direktive ein Filename, so erkennt der Compiler auf Include-Direktive.

R-Compiler-Direktive

Standard: R-

Die R-Direktive steuert die Indexpruefung zur Laufzeit des Programms. Wenn aktiv {$R+}, werden alle Indexoperationen von ARRAYS geprueft, ob die Indizes in den definierten Grenzen liegen. Alle zugewiesenen Skalar- und Teilbereichs-Variablen werden geprueft, ob sie in den entsprechenden Bereichen liegen. Wenn passiv {$R-}, werden keine Pruefungen durchgefuehrt. Dann koennen Indexfehler zu falschen Programmmablaeufen fuehren. Waehrend der Programmentwicklung sollte man stets diese Direktive aktivieren. Nach Beseitigung aller Fehler kann man dann diese Direktive deaktivieren, um das Programmm schneller zu machen.

U-Compiler-Direktive

Standard: U-

Die U-Direktive steuert Nutzer-Unterbrechungen. Wenn aktiv {$U+}, kann der Nutzer zu jeder Zeit das Programm durch ^C unterbrechen. Wenn passiv {$U-}, hat ^C keine Wirkung. Bei Aktivierung wird die Ausfuehrungszeit etwas verlangsamt. Es empfiehlt sich, waehrend der Programmentwicklung stets auch diese Direktive zu aktivieren.

V-Compiler-Direktive

Standard: V+

Die V-Direktive steuert die Type-Pruefung bei STRING-Variablen-Parametern. Wenn aktiv {$V+}, wird eine genaue Typ-Pruefung durchgefuehrt, d.h., die Laenge der aktuellen und formalen Parameter muss uebereinstimmen. Wenn passiv {$V-}, koennen bei aktuellen und formalen STRING-Parametern die Laengen abweichen.

W-Compiler-Direktive

Standard: W2

Die W-Direktive steuert die Schachtelungstiefe der WITH-Anweisung, d.h. die Anzahl der Records, die in einem Block eroeffnet werden koennen. Dem W muss stets eine Zahl zwischen 1 und 9 folgen.

X-Compiler-Direktive

Standard: X+

Die X-Direktive steuert die Optimierung. Wenn aktiv {$X+}, wird die Code-Generierung fuer ARRAYS hinsichtlich maximaler Geschwindigkeit bestimmt. Wenn passiv {$X-}, minimiert der Compiler die Programmgroesse.

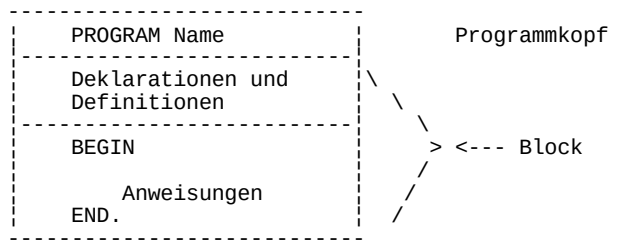
4.2. Programmstruktur/Programmrahmen

Die Programmsprache PASCAL ermoeglicht die modulare Programmierung. Aus diesem Grund sind zusammengehoerige Programmschritte zu Bloecken, Prozeduren oder Funktionen zusammengefasst.
Ein PASCAL-Programm besteht aus dem
- -Programmkopf, dem der
- -Programmblock folgt.

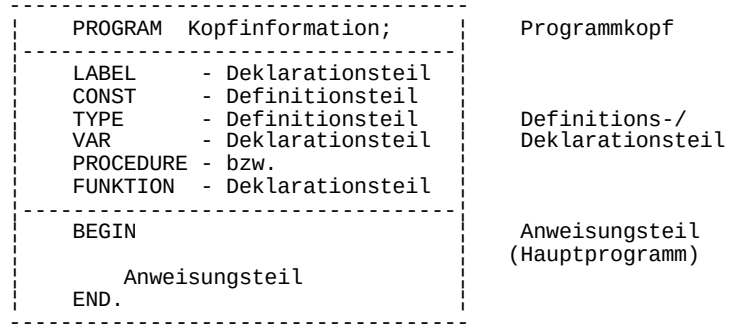
Der Programmblock selbst besteht aus dem
- -Deklarationsteil und dem
- -Anweisungsteil.

Die PASCAL-Syntax verlangt, dass alle Deklarationen bzw. Definitionen am Programmanfang stehen muessen. Im Deklarationsteil werden alle lokalen Objekte des Programms definiert, und im Anweisungsteil stehen alle Aktionen, die mit diesen Objekten ausgefuehrt werden sollen.

Grundaufbau eines PASCAL-Programms:



Im einzelnen besitzt ein PASCAL-Programm folgende Struktur:



*** Programmstruktur/Programmrahmen ***

Syntax:

```
<Programm>::=
        <Programmkopf > <Block> .
<Programmkopf>::=
        PROGRAM <Bezeichner> {(<Programmparameter>)}
<Programmparameter>::=
        <Bezeichner> {,<Bezeichner>}
<Block>::=
        <Markendeklarationsteil>
        <Konstantendefinitionsteil>
        <Typdefinitionsteil>
        <Variablendeklarationsteil>
        <Prozedur—und Funktionsdeklarationsteil>
        <Anweisungsteil>
```

Die Programmparameter beschreiben Files, durch die das Programm mit seiner Umgebung verbunden ist. Werden Sie ausgefuehrt, sind sie im Hauptprogramm zu spezifizieren.

Die Standard-Dateinamen INPUT (Eingabe Tastatur) und OUTPUT (Ausgabe Bildschirm) brauchen nicht definiert werden. Die Angabe im Programmkopf kann entfallen.

Er sollte jedoch aus Gruenden der Uebersichtlichkeit geschrieben werden.

Beispiele:

```
  PROGRAM Test;                              (beide Beispiele )
  PROGRAM Test (INPUT, OUTPUT);              (sind aequivalent)

  PROGRAM Fakt (Drucker, Artikel, Kunde);
```

4.3. Deklarationen und Definitionen

4.3.1. Markendeklaration

Jede Anweisung in einem PASCAL-Programm kann mit einer Marke versehen werden. Die damit gekennzeichneten Anweisungen koennen mit der Sprunganweisung (GOTO) von jeder Stelle des Programmes aus erreicht werden.
Alle Marken (Label) welche in einem Programmblock verwendet werden sollen, muessen deklariert werden.

Syntax:

```
<Markendeklarationsteil>::=
        <leer>
        ¦ LABEL <Marke> {,<Marke>}
<Marke>::=
        <natuerliche Zahl> ¦ <Bezeichner>
```

Bei der Bildung von Markennamen ist sowohl die Verwendung von Zahlen als auch von Bezeichnern erlaubt.

Beispiel:

```
  LABEL Ende,10,20,30;
```

4.3.2. Konstantendefinition

Durch die Konstantendefinition koennen Bezeichnern Werte zugeordnet werden.
Diese Konstantenbezeichner koennen im Programm als Synonym fuer die jeweilige Konstante verwendet werden. Es ist aber auch moeglich, Typenkonstanten zu definieren. Das sind Variablen, die mit CONST einen Anfangswert erhalten.

Die Verwendung der Konstantendefinition hat folgende Vorteile:

- Der Programmtext wird besser lesbar;
- Bei Aenderungen von Konstanten (z.B. Feldgrenzen) muss nur die Definition, nicht aber die Konstante in den einzelnen Anweisungen geaendert werden.
- einfache Anfangswertbelegung fuer Variablen.

Syntax:

```
Konstantendefinition := <leer>
    ¦CONST <Bezeichnerdefinition>{;<Bezeichnerdefinition>
Bezeichnerdefinition::=
    <Konstantendefinition>
    ¦ <Typkonstantendefinition>
Konstantendefinition::=
    <Konstantenbezeichner> = <Konstante>
Konstantenbezeichner::= Bezeichner
Typenkonstantendefinition::=
    <Typkonstantenbezeichner>:<Typ> = <Typkonstante>
Typkonstantenbezeichner::= Bezeichner
Typkonstante::=
    <Konstante>
    ¦(<Typkonstante>{,<Typkonstante>})
    ¦(<Recordteilbezeichner>:<Typkonstante>
                  {;<Recordteilbezeichner>:<Typkonstante>})
    ¦Mengenkonstruktur
```

Bei der Konstantendefinition wird implizit auch der Typ deklariert. Der Wert der Konstanten legt den Typ fest.

Beispiele:
```
  CONST  Datum   = '12.12.85'        { STRING ¦ ARRAY OF CHAR }
         Leerstr = '';               { CHAR ¦ STRING }
         Index   = 130;              { INTEGER }
         Anker   = NIL;              { Zeiger }
```

Vordefinierte Konstante:
```
   PI      =  3.1415926536E+00        {REAL}
   FALSE   =  falsch                  {BOOLEAN}
   TRUE    =  wahr                    {BOOLEAN}
   MAXINT  =  32767                   {INTEGER}
```

Typisierte Konstanten werden ausfuehrlich unter Ziffer 4.3.5. dargestellt.

4.3.3. Datentypen und TYPE-Definition

Jede Variable und Konstante eines PASCAL-Programms besitzt einen Datentyp, welcher die entsprechende Darstellungsform im Speicher und den Wertevorrat spezifiziert.

Grundsaetzlich ist zu unterscheiden zwischen
- Standard-Typen (sind vordefiniert) und
- benutzerdefinierten Typen.

Bei der Definition von Datentypen sind folgende Regeln zu beachten:

(1) Jede Variable kann nur einen Typ besitzen.
(2) Der Typ jeder Variablen muss vor der ersten Verwendung vereinbart werden.
(3) Bei Operationen mit Daten verschiedenen Typs sind Typ- und Zuweisungsvertraeglichkeit zu beachten.

Syntax:

```
<Type>::=
         <einfacher Type>
         ¦ <strukturierter Type>
         ¦ <Zeigertype>
```

4.3.3.1. TYPE-Definition

Die Festlegung des Datentyps erfolgt entweder direkt im Variablendefinitionsteil oder durch einen Typbezeichner. Der Nutzer hat die Moeglichkeit, festgelegte Typbezeichner anzuwenden oder mit Hilfe der Typedefinition eigene festzulegen.
Die Definition von Datentypen erfolgt in folgender Form:

Syntax:

```
<Typdefinitionsteil>::=
        <leer>
        ¦ TYPE <Typdefinition> {;<Typdefinition>}
<Typdefinition>::=
        <Typbezeichner> = <Typ>
```

Eine Definition von Datentypen mit Type hat folgende Vorteile:
- Vereinfachung des Entwurfs eines PASCAL-Programms;
- Einsparung von Schreibaufwand bei Verwendung mehrerer Variablen des gleichen Typs;
- Hilfe beim Verhueten und Suchen von Fehlern;
- Herstellung von Typenvertraeglichkeit fuer Felder;
- Schaffung von Voraussetzungen zum Parameteraustausch mit Unterprogrammen fuer strukturierte Variablen.

4.3.3.2. Einfacher Typ

Syntax:

```
 <einfacher Typ>::= <ordinaler Typ> ¦ REAL

 <ordinaler Typ>::= <ordinaler Standardtyp>
                    ¦ <Aufzaehlungstyp>
                    ¦ <Teilbereichstyp>
```

4.3.3.2.1. Ordinaler Typ

4.3.3.2.1.1. Ordinaler Standardtyp

Der ordinale Standardtyp bezeichnet eine endliche linear geordnete Menge von Werten.
Folgende ordinale Standardtypen sind in PASCAL realisiert:

Syntax:

```
<ordinaler Standardtyp>::=
        CHAR
        ¦ BOOLEAN
        ¦ INTEGER
        ¦ BYTE
```

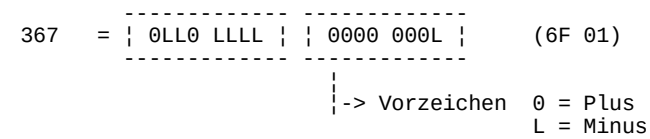
(weitere Einzelheiten zur internen Darstellung enthaelt Anhang F.

Standardtyp	Groesse	Wertebereich
CHAR	1 Byte	Zeichensatz (chr($0)..chr($7F))
BOOLEAN	1 Byte	TRUE ¦ FALSE
INTEGER	2 Byte	-32768 .. +32767 ¦ $0000..$FFFF
BYTE	1 Byte	0..255

CHAR
CHAR definiert einen Datentyp als Elemente des Zeichensatzes. CHAR-Variablen koennen Werte zwischen CHR(0) und CHR(127) annehmen. Ihre Wirkung (Steuerzeichen und druckbare Zeichen) richtet sich nach der Codevereinbarung (vgl. Anhang A).

BOOLEAN
Der Datentyp BOOLEAN repraesentiert Wahrheitswerte, welche durch TRUE ("wahr") und FALSE ("falsch") augedrueckt werden.

INTEGER
Der Datentyp INTEGER definiert eine Untermenge der ganzen Zahlen. Dabei wird zunaechst das nieder-, dann das hoeherwertige Byte abgelegt:



Der Wertebereich umfasst -32768 ... +32767 (MAXINT=32767). INTEGER-Konstanten koennen hexadezimale Zahlen sein (z.B. $3A, $016F). Ihr Wertebereich reicht dann von §0000 bis $FFFF. Ein Ueberlauf zwischen positiven und negativem Bereich wird nicht ueberwacht.

BYTE
Der Datentyp BYTE belegt ein Byte im Speicher und ist zuweisungsvertaeglich zum Typ INTEGER.

4.3.3.2.1.2. Aufzaehlungstyp

Ein Aufzaehlungstyp definiert eine geordnete Menge von Werten durch Aufzaehlung der Bezeichner, die als Konstanten deren Werte ausdruecken.

Syntax:

```
<Aufzaehlungstyp>::=
       (<Bezeichner> {,<Bezeichner>})
```

Beispiele:

```
  TYPE Material = (Grisuten, Baumwolle, Wolle, Polyesterseide)
  TYPE Wochtage = (Montag, Dienstag, Mittwoch, Donnerstag,
                   Freitag, Sonnabend, Sonntag);
```

4.3.3.2.1.3. Teilbereichstyp

Durch die Angabe des kleinsten und des groessten Wertes eines ordinalen Typs kann ein Typ als Teilbereich eines ordinalen Typs definiert werden:

Syntax:

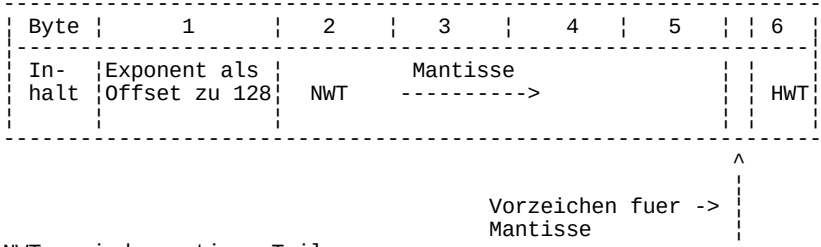
```
<Teilbereichstyp>::=
       <Konstante>..<Konstante>
```

Die erste Konstante legt die untere Grenze fest; ihr Wert darf nicht groesser als die obere Grenze sein.

```
Beispiele:
  TYPE Intzahl   = 1 ..1000;
       Bereich   = -10 .. +10;
       Wochentag = (Montag,Dienstag,Mittwoch,Donnerstag,
                    Freitag,Samstag,Sonntag)
       Werktag   = Montag .. Freitag
 Werktag ist ein Teilbereich des ordinalen Typs Wochentag.
```

4.3.3.2.2. REAL-Typ

Der Datentyp REAL ermoeglicht die Darstellung positiver und negativer gebrochener Zahlen. Eine REAL-Zahl belegt 6 Bytes. Das erste enthaelt den Exponenten und das Vorzeichen, das zweite bis sechste Byte die Mantisse. Das Vorzeichen der Mantisse ist in einem Bit des sechsten Bytes verschluesselt.



NWT - niederwertiger Teil
HWT - hoeherwertiger Teil
Die 6-Byte-REAL-Darstellung sichert eine Genauigkeit von 11 signifikanten Ziffern. Der Wertebereich liegt zwischen 1E-38 und 1E+38. Weitere Einzelheiten zur internen Darstellung enthaelt Anhang F.

4.3.3.3. Strukturierter Typ

Ein strukturierter Typ wird durch die Typen seiner Komponenten und durch die Methode der Strukturierung gekennzeichnet.

Syntax:

```
<strukturierter Typ>::=
        < strukturierter Typ>
        ¦ PACKED < strukturierter Typ>
< strukturierter Typ> ::=
        <Feldtyp>
        ¦ <Recordtyp>
        ¦ <Filetyp>
        ¦ <Mengentyp>
        ¦ <Zeichenkettentyp>
```

PACKED wird vom Compiler akzeptiert, hat aber keine Wirkung. Einzelheiten zur internen Darstellung enthaelt Anhang F.

4.3.3.3.1. Feld-Typ

Ein Feld-Typ ist eine aus einer festen Anzahl von Komponenten bestehende Struktur. Diese Komponenten sind alle vom gleichen Typ. Die Komponenten des Feldes werden durch Indizes angegeben, deren Werte zum ordinalen Typ gehoeren. Sie werden in eckigen Klammern geschrieben und an den Bezeichner des Feldes angehaengt.

Syntax:

```
<Feldtyp>::=
        ARRAY [<Indextyp>{,<Indextyp>}] OF <Typ>
<Indextyp>::=
        <ordinaler Typ>
```

<Typ> ist ein beliebiger Datentyp. Damit gibt es Felder von Feldern, Felder von Felder von Feldern, Felder von Records usw. Ein Feld-Typ heisst n-dimensional, wenn n Indextypen spezifiziert sind.

Beispiele:

```
  TYPE Kette  = ARRAY [Anfang..Ende] OF ARRAY[1..10] OF CHAR;
       Matrix = ARRAY [1..50,1..50] OF REAL;
       Satz   = ARRAY ['a'..'z'] OF BYTE;
       B100   = ARRAY [1..10,1..20] OF 0..99;
  VAR  Tabelle:Kette;
       .
       Tabelle[Anfang][1]:='A';
```

Es besteht die Moeglichkeit, Felder zu kopieren, wenn sie als Ganzes vom gleichen Typ sind, d.h. mit der gleichen Typenvereinbarung eingefuehrt wurden.
Die Pruefung zulaessiger Indexausdruecke ist mit der Compiler-Direktive R moeglich. Standard ist {$R-}, bei gewuenschter Zulaessigkeitspruefung muss {$R+} gesetzt werden (vergl. Ziffer 4.1.4.6.).

4.3.3.3.2. Record-Typ

Ein Record-Typ ist eine Struktur, welche aus einer festen Anzahl Komponenten gleicher oder unterschiedlicher Typen besteht. Fuer jede Komponente wird ein Bezeichner und ein Typ festgelegt. Ein Record-Typ kann mehrere Varianten haben. Dabei kann eine bestimmte Komponente als Kennzeichen verwendet werden, durch deren Wert festgelegt wird, welche Struktur zu einer gegebenen Zeit verwendet werden soll.
Jede Variante wird durch eine Kennzeichenkonstante charakterisiert. Jede dieser Konstanten ist ein Wert des Typs der Kennzeichenvariablen. Der Zugriff zu einer Recordkomponente wird erreicht, indem der Variablenbezeichner mit dem Recordkomponentenbezeichner, getrennt durch einen Punkt, angegeben wird.
Im Fall, dass Datensaetze vom gleichen Typ sind, ist es moeglich, diese einander zuzuweisen und somit einen Datensatz von Datensaetzen zu bilden.

Syntax:

```
<Recordtyp>::=
        RECORD <Recordkomponentenliste> END

<Recordkomponentenliste>::=
        <fester Teil>
        ¦ <fester Teil> ; <varianter Teil>
        ¦ <varianter Teil>

<fester Teil>:: =
        <Recordkomponente> {;<Recordkonponente>}

<Recordkomponente>::=
        <Recordkomponentenbezeichner>
                    {,<Recordkomponentenbezeichner>} : <Typ>
        ¦ <leer>

<varianter Teil>::=
        CASE <Kennzeichenvariable> <ordinaler Typ> OF
        <Variante>{;Variante>}

<Kennzeichenvariable>::=
        <Bezeichner> :<ordinaler Typ>
        ¦ <ordinaler Typ>
Bei fehlenden Bezeichner spricht man von freie Varianten

<Variante>::=
        <Kennzeichenkonstantenliste>:(<Recordkomponentenliste>)
        ¦ <leer>

<Kennzeichenkonstantenliste>::=
        <Konstante> {,<Konstante>}
```

Beispiele:

```
  TYPE Ta = RECORD
              Anr    : STRING [16];
              Bez    : STRING [30];
              Preise : ARRAY [1..5, 1..10] OF REAL;
              Best   : REAL;
              Kz     : CHAR
            END;
       Tdat = RECORD
                Tag : 1..31;
                Mon : 1..12;
                Jhr : INTEGER
              END;
       Tn = RECORD
              Name, Vorname : ARRAY [1..25] OF CHAR;
              Alter         : 0..120;
              Verheiratet   : BOOLEAN
            END;
    Form = (Rechteck,Kreis,Dreieck);
```

```
Tv = RECORD
       x,y     : REAL;
       Flaeche : REAL;
       CASE S  : Form OF
         Dreieck  : (Seite : REAL;
                     Neigung,Wink1,Wink2:Winkel);
         Kreis    : (Radius : REAL);
         Rechteck : (Seite1,Seite2 : REAL)
     END;
               {S ist eine Variable vom Typ Form}
```

4.3.3.3.3. File-Typ

Mit der Definition eines File-Typs wird eine Struktur festgelegt, die aus einer Folge von Komponenten gleichen Typs besteht. Die Anzahl der Komponenten (Groesse des Files) wird durch die Definition nicht festgelegt.

Syntax:

<Filetyp>::= **FILE**
¦**FILE OF** <Typ>
¦**TEXT**

Die erste Definition spezifiziert ein File beliebigen Typs (typlos, ungetypt), die zweite ein Binaerfile und die dritte ein Textfile (TYPE TEXT = FILE OF CHAR).
Fuer Programmverkettungen werden typlose Dateien vereinbart. Eine Konstruktion in der Form TYPE X = FILE OF FILE OF ... ist nicht zulaessig.

Beispiele:

```
  TYPE Arfile = FILE OF Ta;
       Ket1   = FILE;
       Quelle = TEXT;
```

4.3.3.3.4. Mengen-Typ

Unter einer Menge versteht man in PASCAL die Zusammenfassung mehrerer Objekte des gleichen Typs. Zu einer Menge koennen maximal 256 Elemente gehoeren, und die Ordnungswerte des Typs der Objekte liegen folglich im Bereich 0..255.

Syntax:

<Mengentyp>::=
SET OF <ordinaler Typ>

Jedes Element des Satzes wird in einem Bit gespeichert. Ist das jeweilige Element in der Menge enthalten, ist das Bit gesetzt, sonst nicht.

Es werden jeweils soviel Byte reserviert, wie zur Darstellung der Elemente benoetigt werden (maximal 32).

Beispiele:
```
 TYPE Tsp   = (Skat,Halma,Dame,Schach);
 TYPE Spiel = SET OF Tsp;
 TYPE Park  = SET OF (Trabant, Wartburg, Lada, Skoda);
```

Weitere Einzelheiten enthaelt Anhang F.

4.3.3.3.5. Dynamischer Zeichenkettentyp

Syntax:

<Zeichenkettentyp>::=
STRING [<natuerliche Zahl>**]**
¦ **STRING [**<Konstantenbezeichner>**]**

Mit dem Typ STRING wird eine Zeichenkette durch die Angabe der Anzahl maximal moeglicher Zeichen definiert.

```
    STRING [<n>]  = Zeichenkette fuer max.  n Zeichen
                    (n = 1..255)
```

Eine Variable des Typs STRING[<n>] belegt n+1 Byte. Im ersten Byte wird die aktuelle Laenge der Variablen gespeichert. Die einzelnen Zeichen der Zeichenkette sind indizierbar.
Die Speicherung erfolgt in folgender Form:

```
-------------------------------------------------------------
¦ Laenge ¦ 1.Zeichen ¦ 2.Zeichen ¦ 3.Zeichen ¦          ¦ ... ¦
-------------------------------------------------------------
     0          1          2          3          4
```

Wenn die Anzahl der Zeichen einer Kette kleiner ist als die definierte Laenge, dann sind die am Ende im Speicher stehenden Bytes undefiniert (sie werden nicht geloescht).

Beispiel:
```
 TYPE Ts = STRING[20];
```

Weitere Einzelheiten enthaelt Anhang F.

4.3.3.3.6. Standardfelder

Es besteht die Moeglichkeit, zwei Standardfelder vom Typ Byte zu nutzen.
Das Standardfeld MEM wird eingesetzt, um den Zugriff zum Speicher zu realisieren. Der Index jeder Feldkomponente ist identisch mit seiner Adresse im Speicher. Jede Komponente des Feldes ist ein Byte, der Indextyp ist INTEGER. Erfolgt eine Wertzuweisung auf eine Komponente des Feldes MEM, so wird dieser Wert auf der mit dem Indexausdruck spezifizierten Adresse abgelegt.

Beispiele:
```
      Laufwerk := CHR((MEM[4] AND 15 )+65);
      IObyte   := MEM[3];
```

Das Standardfeld PORT wird genutzt, um den Zugriff zu den Datenports zu realisieren. Jede Feldkomponente stellt einen Datenport dar, dessen Adresse dem Wert des Indexausdruckes entspricht. Der Indextyp ist INTEGER. PORT darf nur in Zuweisungen und Ausdruecken verwendet werden; und seine Komponenten duerfen keine Variablenparameter in Unterprogrammen sein.
Standardfelder sollten nur von erfahrenen Programmierern benutzt werden, da unmittelbare, nichtkontrollierbare Eingriffe in das Laufzeitsystem erfolgen koennen.

4.3.3.4. Zeigertyp

Der Zugriff zu dynamischen Variablen erfolgt mit Hilfe des Wertes eines Zeigers. Dieser Zeiger wird waehrend der Erzeugung einer dynamischen Variablen bereitgestellt.
Der Zeigertyp besteht aus einer theoretisch unbegrenzten Menge von Werten, die auf Elemente eines Typs weisen (vergl.4.9. Zeiger und Listen).

Syntax:

```
<Zeigertyp>::=
          ^<Typbezeichner>
```

Beispiel:

```
  TYPE     Zgt      = ^Element;
           Element  = RECORD
                        Wert1 : REAL;
                        Wert2 : REAL;
                        Wert3 : INTEGER;
                        Next  : Zgt
                      END;
```

Anmerkung

Die Variable vom Typ Zgt ist hier ein Zeiger auf ein Objekt vom Typ Element.
Die Bezugnahme auf eine noch nicht definierte Struktur (hier Element) ist in diesem Ausnahmefall moeglich.

4.3.3.5. Typumwandlung und Bereichspruefung

Typumwandlungen werden auf konventionelle Art mit Konvertierungsfunktionen oder mit Retyping ermoeglicht. Bereichspruefungen fuer Skalar- und Teilbereichsvariablen sind mit der Compiler-Direktive {$R+} realisierbar. Standard ist dabei {$R-}, d.h. bei gewuenschter Bereichspruefung muss der Schalter {$R+} im Programmtext gesetzt werden.

4.3.3.5.1. Retyping

Die Typbezeichner CHAR, BYTE, INTEGER und BOOLEAN sowie Typbezeichner des Aufzaehlungstypes sind gleichzeitig als Funktionsbezeichner zur Konvertierung verwendbar.
Hierbei bedeutet das lediglich, dass zum Beispiel
"y = INTEGER('A')" und "y := ord('A')" sowie "x = CHAR(78)" und "x = chr(78)" voellig gleich sind (x ist hier vom Typ CHAR, y vom Typ INTEGER oder BYTE.

Diese Vorgehensweise heisst Retyping.
REAL und STRING sind nicht fuer das Retyping zugelassen.

Beispiele:
```
  TYPE   Monat
=(Jan,Feb,Maerz,April,Mai,Juni,Juli,Aug,Sept,Okt,
                Nov,Dez);
         Farbe =(Rot,Gelb,Gruen);
```

Die Anwendung von Retyping auf die obigen Definitionsbeispiele ermoeglicht (Vergleich jeweils TRUE):

```
         Monat(11)      = Nov
         INTEGER(Gelb)  = 2
                $41     = BYTE ('A')
```

4.3.3.5.2. Pseudofunktionen zur Konvertierung

Die Pseudofunktionen der Konvertierung dienen zur Herstellung der Vertraeglichkeit eines Skalartyps in einem anderen (Pseudo, weil in Wirklichkeit keine Operationen stattfinden).
Pseudofunktionen der Konvertierung sind ORD, PTR und CHR.

ORD-Funktion

ord (<Ausdruck>)

Die Funktion liefert den Ordinalwert (Typ INTEGER) des Ausdrucks.

Beispiele:

```
  write (ord ('A'));        {= 65}
  write (ord (67));         {= 67}
  write (ord(chr(86)));     {= 86}
```

Mit ORD kann auch der INTEGER-Wert von Zeigern festgestellt werden.

CHR-Funktion

chr (<Ausdruck>)

Die Funktion liefert das Zeichen, dessen Ordinalwert dem Wert des Ausdrucks entspricht. Grundlage ist der jeweilige Zeichensatz.

Beispiele:
```
  write(chr(66));                   {= B}
  write(chr('C'));                  {= B}
  write(chr(ord('L')));             {= L}
```

Mit CHR ist auch ein Zugriff auf das dynamische Byte (Index=0) eines STRING moeglich. Es sollten jedoch die STRING-Funktionen-Prozeduren vorgezogen werden.

PTR-Funktion

ptr (<Ausdruck>)

Der Ausdruck muss vom Typ INTEGER sein.
Mit der Pseudofunktion PTR ist es moeglich, die in einem Pointer stehende Adresse direkt zu steuern. PTR konvertiert dabei ein INTEGER-Argument in einen Pointer.

Beispiel:

```
TYPE  Zeiger = ^ INTEGER;
VAR   Pufferzeiger:Zeiger;
.
.
.
Pufferzeiger := ptr($8000);
```

4.3.4. Variablendeklaration und Variablenzugriff

4.3.4.1. Deklaration von Variablen

Alle Variablen, die in PASCAL verwendet werden, muessen deklariert werden.
Bei der Deklaration wird einer Variablen ein Bezeichner und ein Typ zugeordnet.
Waehrend mit der TYPE-Definition nur ein Datentyp beschrieben wird, wird mit der Variablendeklaration Speicherplatz bereitgestellt.

Syntax:

```
<Variablendeklarationsteil>::=
        <leer>
        ¦ VAR <Variablendeklaration> {;<Variablendeklaration>}

<Variablendeklaration>::=
        <Variablenbezeichner>{,<Variablenbezeichner>} : <Typ>
        ¦ <Variablenbezeichner>   :<Typ> ABSOLUTE <Adresse>

<Adresse>::=
          <Konstante>
         ¦<Variablenbezeichner>
```

Absolute Variablen werden durch das Schluesselwort ABSOLUTE gekennzeichnet. Die Variablen werden im Speicher an die durch <Adresse> gekennzeichnete Adresse gelegt.
Diese Adresse sollte ausserhalb des PASCAL-Programms liegen. Der Programmierer ist fuer die Verwaltung selbst verantwortlich.
Beispiele:

```
  VAR  IObyte    :  BYTE ABSOLUTE $0003;
       Cmdzeile  :  STRING[127] ABSOLUTE $80;
```

ABSOLUTE kann auch verwendet werden, um Variablen zu ueberlagern. Die eine Variable beginnt dann auf der gleichen Adresse wie die andere Variable. Dies ist leicht zu erreichen. Folgt in der Variablendefinition dem Wort ABSOLUTE der Bezeichner einer Variablen (oder eines Parameters), dann beginnt die neue Variable auf der Adresse dieser Variablen (oder des Parameters).

Beispiele:

```
          VAR
               Eins      : STRING[22];
               Zwei      : BYTE ABSOLUTE Eins;
```

In diesem Beispiel beginnt Zwei auf der gleichen Adresse wie Eins. Da aber an dieser Stelle die Laenge von Eins steht, enthaelt Zwei die aktuelle Laenge von Eins.
Es ist zu beachten, dass in einer absoluten Deklaration nur **ein** Bezeichner erklaert werden kann! Die folgende Konstruktion ist also nicht erlaubt:

```
Ident1, Ident2 : INTEGER ABSOLUTE $8000;
```

Weitere Beispiele:

```
  VAR     x : REAL;
          y : ARRAY [1..100] OF REAL;
        Art : Ta;
        Mat : ARRAY [1..30, 1..50] OF INTEGER;
        Ans : ARRAY [1..5] OF ARRAY [1..30] OF CHAR;
      Ardat : FILE OF Ta;
       Zgr1 : ^Element;
       Satz : RECORD
                R:REAL;
                I:INTEGER;
                M:ARRAY[1..10, 1..10, 1..100] OF INTEGER;
                B:BOOLEAN
              END;
        Tex : STRING [20];
         Bs : ARRAY[0..23] OF ARRAY[0..79] OF CHAR;
                    ABSOLUTE $F800;
```

4.3.4.2. Variablenzugriff

Der Zugriff zu den Werten der Variablen erfolgt durch ihre Auffuehrung im Programmtext. Es gibt folgende Moeglichkeiten:

Syntax:

```
<Variable>::=
        <Vollstaendige Variable>
        ¦ <indizierte Variable>
        ¦ <Recordkomponentenvariable>
        ¦ <dynamische Variable>
```

Vollstaendige Variable

Der Wert einer Variablen kann durch ihren Bezeichner aufgerufen werden. Es kann sich um einfache, strukturierte Variablen oder Zeiger handeln.

Beispiele:

```
  Satz
  Tex
  x
```

Indizierte Variable

Die Komponente einer n-dimensionalen Feldvariablen wird durch die Angabe der Variablen bezeichnet, der ein n-dimensionaler Index folgt.

Syntax:

```
<Indizierte Variable>::=
        <Feldvariable> [<Index>{,<Index>}]
<Feldvariable>::=
        <Variable>
<Index>::=<ordinaler Ausdruck>
```

Die Typen der Indexausdruecke muessen mit den Indextypen vertraeglich sein, die in der Definition des Feld-Typs vereinbart wurden.

Beispiele:
```
  Mat[5,6]
  y [10]
  y[I+14]
```

Recordkomponentenvariable

Die Komponente einer Recordvariablen wird bezeichnet durch die Angabe der Recordvariablen, gefolgt von dem Bezeichner der Komponente.

Syntax:

```
<Recordkomponentenvariable>::=
        <Recordvariable> . <Recordkomponentenbezeichner>
```

Beispiele:
```
  Art.Anr.
  Satz.r
  Satz.m [i,j,k]
```

Dynamische Variable (Inhalt einer Zeigervariablen)

Syntax:

```
<Dynamische Variable>::=
          <Zeigervariable>^
```

Wenn p eine an den Typ t gebundene Zeigervariable ist, dann bezeichnet p diese Variable und den Wert ihres Zeigers;
p^ bezeichnet die Variable des Typs t, auf die durch p verwiesen wird.

Beispiele:
```
  Zgr1^                              {dynamische Variable}
  Zgr1^.Wert1                        {Aufruf des Wertes einer }
  Zgr2^.Nachf                        {dynamischen Recordkompo-}
                                     {nentenvariablen}
```

Beispiele fuer Variablenzugriff

Nachfolgend wird an einigen Beispielen der Zugriff zu Variablen dargestellt.

Verwendet werden die Beispiele zur TYPE-Definition (Pkt.4.3.3.) und zur Variablen-Deklaration (Pkt. 4.3.4.).

Variablenzugriff	Bereitgestellte Daten	Typ
x	1 * REAL	REAL
y	100 * REAL	ARRAY OF REAL
y [66]	1 * REAL	REAL
Art	1 * 17 BYTE STRING(CHAR)	RECORD bzw. FILE OF RECORD
	1 * 31 BYTE STRING(CHAR)	
	50 * REAL	
	1 * REAL	
	1 * CHAR	
Ar .Preise	50 * REAL	ARRAY
Art.Preise[i,j]	1 * REAL	REAL
Artdat .Kz	1 * CHAR	CHAR
Mat	1500 * INTEGER	ARRAY OF INTEGER
Mat[20,20]	1 * INTEGER	INTEGER
Ans	150 * CHAR	ARRAY OF ARRAY
Ans[2][3]	1 * CHAR	CHAR
Satz	1 * REAL	RECORD
	1 * INTEGER	
	10000 * INTEGER	
	1 * BOOLEAN	
Satz.m	10000 * INTEGER	ARRAY OF INTEGER
Satz.m[I,6,89]	1 * INTEGER	INTEGER
Zgr1^.Wert1	1 * REAL	REAL
Bs[12][8]	1 * CHAR	CHAR
k	1 * INTEGER	INTEGER
Zgr1.	1 * INTEGER	Zeiger

4.3.5. Typisierte Konstante

Eine typisierte Konstante kann wie eine Variable verwendet werden. Sie ist als initialisierte Variable zu betrachten, deren Wert von Anfang an definiert ist. Die Verwendung typisierter Konstanten erspart Laufzeit, da die Anfangsbelegung bereits vom Compiler vorgenommen wird. Typisierte Konstanten werden wie normale Konstanten definiert; sie erhalten nur zusaetzlich auch ihren Typ. Man beachte, dass die definierten Werte nur beim Neustart der COM/CHN-Files zur Verfuegung stehen und dann ihren Wert aendern koennen.
Ein Wiederstart kann bereits andere Werte bringen. Die Syntax ist in Ziffer 4.3.2. definiert.

4.3.5.1. Einfache typisierte Konstante

Eine einfache typisierte Konstante wird wie eine einfache Variable definiert.

Beispiele:
```
  CONST   Anzahl: INTEGER = 1267;
          Zahl: REAL = 12.67;
          Zeichen: CHAR = ^Q;
          Buchstabe:CHAR = #65;
```

Typisierte Konstanten duerfen anstelle einer Variablen als Parameter in Unterprogrammen verwendet werden. Eine typisierte Konstante stellt eine Variable mit einem definierten Wert dar. Sie kann somit nicht in der Definition anderer Konstanten oder Typen verwendet werden.

Beispiel:
```
     CONST
         Unten : INTEGER = 0;
         Oben  : INTEGER = 50;
     TYPE
         Feld: ARRAY [Unten..Oben] OF INTEGER;  {Unzulaessig !}
```

4.3.5.2. Strukturierte typisierte Konstante

Strukturierte typisierte Konstanten sind
Feldkonstanten,
Recordkonstanten und
Mengenkonstanten.

4.3.5.2.1. Typisierte Feldkonstante

Beispiele:
```
     TYPE
          Zustand        = (Kalt,Heiss,Warm);
          Feld           = ARRAY [Zustand] OF STRING[5];
     CONST
          Zust:Feld      =('Kalt','Heiss','Warm');
```

Im Beispiel wird die Feldkonstante Zustand definiert, die genutzt werden kann, um Werte vom Aufzaehlungstyp in ihre entsprechende Stringdarstellung zu konvertieren:

```
     Zustand[Kalt]     = 'Kalt'
     Zustand[Heiss]    = 'Heiss'
     Zustand[Warm]     = 'Warm'
```

Jeder Typ, ausser einem Feld- oder Zeigertyp, stellt einen zulaessigen Komponententyp einer Feldkonstante dar. Bei Characterfeldtypen sind einzelne Zeichen und Zeichenketten erlaubt.

4.3.5.2.2. Mehrdimensionale typisierte Feldkonstante

Bei der Definition einer typisierten mehrdimensionalen Feldkonstante wird jede Dimension in separate Klammernpaare eingeschlossen, die durch Komma voneinander getrennt sind.

Dabei entspricht die innerste Konstante der am weitesten rechts stehenden Dimension.

Beispiele:

```
      TYPE
           Feld = ARRAY[0..1,0..1,0..1] OF INTEGER;
      CONST
           Zahl : Feld = (((0,1),(2,3)),((4,5),(6,7)));
      BEGIN
           writeln (Zahl[0,0,0],' =0');
           writeln (Zahl[0,0,1],' =1');
           writeln (Zahl[0,1,0],' =2');
           writeln (Zahl[0,1,1],' =3');
           writeln (Zahl[1,0,0],' =4');
           writeln (Zahl[1,0,1],' =5');
           writeln (Zahl[1,1,0],' =6');
           writeln (Zahl[1,1,1],' =7')
      END;
```

4.3.5.2.3. Typisierte Recordkonstante

```
Beispiele:
  TYPE
      Zahl       = RECORD
                     a,b,c : INTEGER
                   END;
       Farbe     = (Rot,Gelb,Gruen,Blau);
       Stoff     = (Wolle,Seide,Tweet);
       Kleid     = RECORD
                     Design  : ARRAY[1..4] OF Farbe;
                     Material: Stoff
                   END;
  CONST
       Nummer: Zahl = (a:0, b:0, c:0);
       Modell: Kleid =
                      (Design:(Rot,Gelb,Gruen,Blau);
                       Material:Tweet);
       Matrix: ARRAY[1..3] OF Zahl =
                      ((a:1,  b:4,   c:5),
                       (a:13, b:8,  c:55),
                       (a:200,b:16,  c:-65));
```

Die Feldkonstanten sind in der gleichen Reihenfolge zu definieren, wie sie in der Recorddefinition auftreten. Im Fall, dass ein Datensatz Felder vom File- oder Zeigertyp enthaelt ist es nicht moeglich, typisierte Konstanten fuer diesen Recordtyp zu definieren. Wenn eine Recordkonstante Varianten enthaelt, so ist der Programmierer selbst dafuer verantwortlich, dass nur die Datenfelder der gueltigen Variable spezifiziert werden. Enthaelt die Variable ein Kennzeichnungsfeld, dann muss auch ihr Wert spezifiziert werden.

4.3.5.2.4. Typisierte Mengenkonstante

Eine typisierte Mengenkonstante wird aus einer oder mehreren Elementenspezifikationen, die durch Komma getrennt und in eckigen Klammern eingeschlossen sind, gebildet. Eine Elementenspezifikation kann eine Konstante oder ein Bezeichnerausdruck sein. Er besteht aus zwei Konstanten, getrennt durch zwei Punkte.

Beipiele:

```
TYPE
     Gross= SET OF 'A'..'Z';
     Klein= SET OF 'a'..'z';
CONST
     Grossbuchst :Gross  = ['A'..'Z'];
     Vokale      : Klein = ['a','e','i','o','u'];
     Zeichen     : SET OF CHAR =
                     [' '..'/',':'..'?','['..'`','{'..'~'];
```

4.3.6. Prozedur- und Funktionsdeklaration

Eine Prozedur-Funktionsvereinbarung definiert ein Unterprogramm innerhalb eines Programms oder einer anderen Prozedur/Funktion. Es ist gueltig fuer den gesamten Block, in dessen Vereinbarungsteil sie deklariert wurde. Einzelheiten enthalten die Ziffern 4.6. und 4.7.

4.4. Operatoren und Ausdruecke

4.4.1. Operatoren

Operatoren werden zum Verknuepfen bzw. Vergleichen von Ausdruecken verwendet. Operatoren koennen in sechs Kategorien eingeteilt werden:

1) Minusvorzeichen
2) NOT Operator
3) Multiplikationsoperatoren: *,/,DIV,MOD,AND,SHL,SHR.
4) Additionsoperatoren: +,-,OR,XOR.
5) Vergleichsoperatoren: =,<>,<,>,<=,>=.
6) Mengenoperatoren.

Sind beide Operanden eines Multiplikations- oder Additionsoperators vom Typ INTEGER, dann ist auch das Ergebnis vom Typ INTEGER. Wenn einer oder beide Operatoren vom Typ REAL sind, dann ist auch das Ergebnis vom Typ REAL. Vergleichsoperatoren liefern immer den Typ BOOLEAN. Die Prioritaet der Operatoren wird in Ziffer 4.4.1.7. dargestellt.

4.4.1.1. Minusvorzeichen

Das Minuszeichen bezeichnet die Negation des Operanden, der vom Typ INTEGER oder REAL sein muss.

4.4.1.2. Operator NOT

Der Operator NOT kann auf Operanden vom Typ BOOLEAN angewendet werden und drueckt die Negation aus:

```
NOT TRUE  = FALSE
NOT FALSE = TRUE
```

PASCAL erlaubt auch die Anwendung des Operators NOT auf Operanden vom Typ INTEGER und BYTE. In diesem Falle erfolgt die Negation der einzelnen Bits.

Beispiele:

```
NOT 0     = -1
NOT -15   = 14
NOT $2345 = $DCBA
```

4.4.1.3. Multiplikationsoperatoren

```
<Multiplikationsoperator> ::=
          *
        ¦ /
        ¦ DIV
        ¦ MOD
        ¦ AND
        ¦ SHL
        ¦ SHR
```

Operator	Operation	Typ der Operanden	Typ des Ergebnisses
*	Multiplikation	REAL,REAL	REAL
*	Multiplikation	INTEGER,INTEGER	INTEGER
*	Multiplikation	REAL, INTEGER	REAL
*	Durchschnitt	Mengen	Menge
/	Division	REAL,REAL	REAL
/	Division	REAL, INTEGER	REAL
DIV	Division	INTEGER	INTEGER
MOD	Rest (Modulus)	INTEGER	INTEGER
AND	logisches UND	BOOLEAN	BOOLEAN
AND	arithmetisches UND	INTEGER	INTEGER
SHL	Shift links	INTEGER	INTEGER
SHR	Shift rechts	INTEGER	INTEGER

BYTE gilt als echte Teilmenge von INTEGER. Bei den Operationen muessen sich dann aber BYTE/BYTE gegenueberstehen.

Beispiele:

```
     123*456        =    492 falsch, Ueberlauf der Integerzahl!
     123 DIV 4      =    30
     12 MOD 5       =    2
     TRUE AND FALSE =    FALSE
     12  AND 22     =    4
     2   SHL 7      =    256 {verschiebt das Bitmuster des
                             INTEGER-Typs 2 um 7 Positionen
                             nach links}
     256 SHR 7      =    2
```

Die bitweise AND - Operation zeigt das folgende Schema:

	Zahl	hexadezimal (HWT,NWT)	Bitmuster (HWT,NWT)
1. Operand	12	$0C	0000 1100 0000 1100
2. Operand	22	$16	0001 0110 0001 0110
Ergebnis bei AND	4	$04	0000 0100

4.4.1.4. Additionsoperatoren

Syntax:

<Additionsoperator>::=

+
-
OR
XOR

Operator	Operation	Typ der Operanden	Typ des Ergebnisses
+	Addition	REAL,REAL	REAL
+	Addition	INTEGER,INTEGER	INTEGER
+	Addition	INTEGER,REAL	REAL
+	Vereinigung	Mengen	Menge
-	Subtraktion	REAL,REAL	REAL
-	Subtraktion	INTEGER,INTEGER	INTEGER
-	Subtraktion	INTEGER,REAL	REAL
-	Differenz	Mengen	Menge
OR	logisches ODER	BOOLEAN	BOOLEAN
OR	arithmetisches ODER	INTEGER,INTEGER	INTEGER
XOR	logisches XOR	BOOLEAN,BOOLEAN	BOOLEAN
XOR	arithmetisches XOR	INTEGER,INTEGER	INTEGER

BYTE gilt wieder als Teilbereichstyp 0..255 von INTEGER.

Beispiele:

```
     123 + 456      =    579
     456 - 123.0    =    333.0
     TRUE OR FALSE  =    TRUE
     12 OR 22       =    30
     TRUE XOR FALSE =    TRUE
     12 XOR 22      =    26
```

4.4.1.5. Vergleichsoperatoren

Syntax:

```
<Vergleichsoperator>  ::=
          =
        | <>
        | <
        | <=
        | >
        | >=
        | IN
```

Operator	Operation	Typ der Operanden	Typ d. Ergebnisses
=, <>	gleich, ungleich	einfacher Typ, Mengen, Zeiger, Zeichenketten	BOOLEAN
<, >	kleiner, groesser	einfacher Typ, Zeichenkette	BOOLEAN
<=, >=	kleiner gleich groesser gleich	einfacher Typ Zeichenkette	BOOLEAN
<=	Inklusion "ist enthalten in"	Mengen	BOOLEAN
>=	Inklusion "enthaelt"	Mengen	BOOLEAN
IN	Enthaltensein	ordinaler Typ /Menge	BOOLEAN

Bei Vergleichen von Zeichenketten wird links begonnen, und die Zeichen werden byteweise entsprechend ihrer Ordnung im Zeichensatz verglichen. Kuerzere Zeichenketten werden durch Leerraeume ergaenzt.

Bei Vergleichen der Ordinalwerte von booleanschen Groessen gilt:
FALSE < TRUE
(ord(FALSE) = 0; ord(TRUE) = 1).

4.4.1.6. Mengenoperatoren

Die Mengenoperationen werden entsprechend ihrer Rangfolge in folgende drei Klassen eingeteilt:

1) `*` Mengendurchschnitt.

2) `+` Mengenvereinigung,
   `-` Mengendifferenz.

3) `=` Test auf Gleichheit,
   `<>` Test auf Ungleichheit,
   `>=` Wahr, wenn der zweite Operand im ersten enthalten ist,
   `<=` Wahr, wenn der erste Operand im zweiten enthalten ist,
   `IN` Test auf Mitgliedschaft in einer Menge. Der zweite Operand ist ein Mengentyp und der erste ein Mengenausdruck vom gleichen Typ wie der Basistyp der Menge. Das Ergebnis ist wahr, wenn der erste Operand ein Element des zweiten Operanden ist, andernfalls ist es falsch.

Die Pruefung auf eine leere Durchschnittsmenge kann man in der Form A*B = [] fuer zwei Mengen programmieren, [] kennzeichnet eine leere Menge. Die Relationen < und > sind fuer Mengen nicht definiert.

Bespiele:
```
    x:= [1,2];
    y:= [2,3];

    e:= x * y;      { e = [2] }

    e:= x + y;      { e = [1,2,3] }

    e:= x - y;      { e = [1] }
```

4.4.1.7. Prioritaet

In mehrgliedrigen Ausdruecken werden die einzelnen Operationen entsprechend ihrer Prioritaet ausgefuehrt:

```
  NOT                                    {hoechste Prioritaet}
  Multiplikationsoperatoren
  Additionsoperatoren
  Vergleichsoperatoren                  {niedrigste Prioritaet}
```

Sind in einem Ausdruck mehrere Operatoren gleicher Prioritaet, dann werden diese von links beginnend abgearbeitet.

Die Prioritaet kann durch Setzen von Klammern veraendert werden. Dabei werden Klammern, von links bzw. von innen beginnend, zuerst aufgeloest.

Innerhalb der Klammer gelten wieder die o.g. Regeln.

Beispiele:

```
      5 + 6  * 10        =  65
     (5 + 6) * 10        = 110
     (5* (3+6) -8) +10   =  47
     (5+6) < (3*5)       = TRUE
      NOT (8 > 4)        = FALSE
```

Man beachte, dass in logischen Ausdruecken, z.B. (x>5) AND (y>10), die Klammern notwendig sind, um den durch die Prioritaet sonst entstehenden Typkonflikt zu vermeiden.

4.4.2. Ausdruecke

Ausdruecke sind Konstruktionen, welche die Regeln fuer das Rechnen mit Werten von Variablen und die Erzeugung neuer Werte durch Anwendung von Operatoren ausdruecken.

Ausdruecke bestehen aus Operanden (Variablen, Konstanten), Operatoren und Funktionen.

Syntax:

```
<Ausdruck>::=
        <einfacher Ausdruck>
        ¦ <einfacher Ausdruck><Vergleichsoperator>
                                        <einfacher Ausdruck>

<einfacher Ausdruck> ::
        <Term>
        ¦ <einfacher Ausdruck><Additionsoperator><Term>

<Term>::=
        <Faktor>
        ¦ <Faktor><Multiplikationsoperator><Faktor>

<Faktor>::=
        <Variable>
        ¦ <vzl. Konstante>
        ¦ <Funktionsaufruf>
        ¦ (<Ausdruck>)
        ¦ <NOT Operator><Faktor>
        ¦ <Faktor>
        ¦ <Menge>

<Menge>::=
        [ <Liste der Elemente> ] ¦  []

<Liste der Elemente>::=
        <Element> {,<Element>}
```

```
<Element>::=
        <Ausdruck>
        ¦ <Ausdruck>..<Ausdruck>
```

Ausdruecke, die Elemente der gleichen Menge sind muessen alle vom gleichen Typ sein (= Basistyp der Menge).

[] kennzeichnet eine leere Menge und [x..y] bezeichnet die Menge aller Werte aus dem Intervall x bis y.

Beispiele:

```
     100.76
     x
     x + y[12]
     (x * ART.PREISE[1,4] / 100)
     [Montag, Dienstag, Mittwoch]
     (x < y[2]) AND (ZGR1^.Wert1 <> 0)
     x = 12.3456
```

Wert des Operanden "a"	T	T	F	F
Wert des Operanden "b"	T	F	T	F
NOT a (Negation)	F	F	T	T
NOT b (Negation)	F	T	F	T
a AND b (Konjunktion)	T	F	F	F
a OR b (Disjunktion)	T	T	T	F
a XOR b (Exklusion)	F	T	T	F

4.4.3. Funktionsaufruf

Durch einen Funktionsaufruf wird eine Funktion aktiviert. Der Aufruf besteht aus dem Funktionsbezeichner und einer Liste aktueller Parameter.
Die aktuellen Parameter (Variablen und Ausdruecke) werden fuer die korrespondierenden formalen Parameter substituiert (vergl. Ziffer: 4.6.).

Das Auftreten eines Funktionsaufrufes im Programm bewirkt die Aktivierung der Funktion, durch die sie bezeichnet wird. Wenn die Funktion keine Standardfunktion ist, muss sie vor dem Aufruf definiert sein.

Syntax:

```
<Funktionsaufruf>::=
        <Funktionsbezeichner>
        ¦ <Funktionsbezeichner> (<Parameter>{,<Parameter>})

<Parameter>::=
        <Ausdruck>
```

Funktionen oder Prozeduren sind als aktuelle Parameter nicht erlaubt.

Beispiele:

```
Volumen(Radius,Hoehe)
Durschn (x,y[j])
sin (x)
eof (f)
sqrt(x)
```

4.5. Anweisungen

4.5.1. Uebersicht

Anweisungen beschreiben auszufuehrende Operationen. Sie koennen durch Marken (Label) gekennzeichnet sein, auf die in Sprunganweisungen (GOTO) Bezug genommen wird. Davon sollte aus softwaretechnologischen Gruenden nur im Ausnahmefall Gebrauch gemacht werden.

Syntax:

```
<Anweisung>::=
        <Marke> : <nichtmarkierte Anweisung>
        ¦ <nichtmarkierte Anweisung>

<Marke>::=
        <natuerlicher Zahl> ¦ <Bezeichner>

<nichtmarkierte Anweisung>::=
        <einfache Anweisung>
        ¦ <strukturierte Anweisung>
```

Anweisungen werden durch Semikolon getrennt. Ein Semikolon vor END und UNTIL kann entfallen, da diese Wortsymbole noch zur Anweisung gehoeren. Wird es gesetzt, spezifiziert das Semikolon eine Leeranweisung.

4.5.2. Einfache Anweisungen

Eine einfache Anweisung ist eine Anweisung, in der keine andere Anweisung enthalten ist.

Syntax:

```
<einfache Anweisung>::=
        <Leeranweisung>
        ¦ <Ergibtanweisung>
        ¦ <Prozeduranweisung>
        ¦ <Sprunganweisung>
```

4.5.2.1. Ergibt-Anweisung

Durch die Ergibt-Anweisung wird der rechts von := stehende Ausdruck der Variablen links von := zugewiesen. Innerhalb einer Funktion kann links der Funktionsbezeichner stehen.

Syntax:

```
<Ergibtanweisung>::=
        <Variable> := <Ausdruck>
        ¦ <Funktionsbezeichner> := <Ausdruck>
```

Der Typ der Variablen bzw. der Funktion muss mit dem Typ des Ausdruckes zuweisungsvertraeglich sein.

```
Beispiele:
  x:= Art.Best * Art.Preis [j,8];
  y[66]:= x;
  y[j]:= 47.88;
  Mat [4,41]:= Satz.I;
  Tex:= 'Zuweisung';
  Zgr1^:= Zgr2^;
  Zgr1:= Zgr2;
  Zgr2:= NIL;
  Zgr1^.Wert1:= 234.645;
  x:= x + 10;
  i:= succ(i);
```

4.5.2.2. Prozeduranweisung

Durch eine Prozeduranweisung wird die Aktivierung der Prozedur, die durch den Prozedurbezeichner gekennzeichnet ist, veranlasst.
Die Prozeduranweisung kann eine Liste von aktuellen Parametern enthalten, die fuer die korrespondierenden formalen Parameter substituiert werden. Diese formalen Parameter wurden in der Prozedurvereinbarung deklariert.
Die Korrespondenz ist durch die Stellung der Parameter in den Listen der aktuellen und formalen Parameter gegeben.
Es werden Wert-, Variablen- und nichttypisierte Parameter unterschieden (vergl. Ziffer 4.6.2.2.).

Syntax:

```
<Prozeduranweisung>::=
        <Prozedurbezeichner> (<Parameter>{,<Parameter>})
        ¦ <Prozedurbezeichner>

<Parameter>::=
        <Ausdruck>
```

Funktionen und Prozeduren sind als aktuelle Parameter nicht zugelassen.

```
Beispiele:
  read  (x);
  write ('Bildschirmausgabe');
```

4.5.2.3. Sprunganweisung

Durch die Sprunganweisung wird erreicht, dass die Programmausfuehrung mit der Anweisung fortgesetzt wird, die durch die entsprechende Marke gekennzeichnet ist.

Syntax:

```
<Sprunganweisung>::=
        GOTO <Marke>
```

Der Gueltigkeitsbereich einer Marke ist der Anweisungsteil des Programmtextes, in welchem die Marke deklariert ist.

```
Beispiel:
  PROGRAM xyz;
  LABEL 10;
  VAR i,j: INTEGER;
  BEGIN
  10: read(i);
      j:= i * 21 + 5;
       .
       .
       .
      GOTO 10;
       .
       .
  END.
```

Die Marken gelten nicht in Unterprogrammen des jeweiligen Blockes.

4.5.2.4. Leeranweisung

Die Leeranweisung enthaelt keinerlei Symbole und hat keine Wirkung.

```
<Leeranweisung>::=
        <leer>
```

Eine Leeranweisung kann ueberall im Programm stehen, wo eine Anweisung geschrieben werden kann.

Beispiele:

```
 .
 .
      IF x<0 THEN GOTO Stop;
      writeln('Das Ergebnis ist ',x);
 Stop:END.
```

Die Leeranweisung befindet sich zwischen Doppelpunkt und END.

4.5.3. Strukturierte Anweisungen

Strukturierte Anweisungen sind aus mehreren Anweisungen zusammengesetzte Konstruktionen, die entweder
- nacheinander (Verbundanweisung),
- bedingt (bedingte Anweisungen) oder
- wiederholt (Zyklusanweisungen)

auszufuehren sind.

Syntax:

```
<Strukturierte Anweisung>::=
        <Verbundanweisung>
        ¦ <bedingte Anweisung>
        ¦ <Zyklusanweisung>
        ¦ <White-Anweisung>
```

4.5.3.1. Verbundanweisung

Durch die Verbundanweisung wird eine Folge von Anweisungen zusammengefasst. Die Ausfuehrung der geklammerten Anweisungen erfolgt in der gleichen Reihenfolge wie sie geschrieben sind.

Syntax:

```
<Verbundanweisung>::=
        BEGIN
        <Anweisung> {;<Anweisung>}
        END
```

Eine Verbundanweisung kann ueberall dort geschrieben werden, wo eine Anweisung stehen darf, aber eine Folge von Anweisungen erforderlich ist.

Beispiel:

```
  BEGIN
    x:= 2.678;
    y[i] := x + 2.71;
    i:= i + 1
  END
```

4.5.3.2. Bedingte Anweisungen

Bedingte Anweisungen ermoeglichen die Steuerung der Programmausfuehrung in Abhaengigkeit von Bedingungen.
Die Bedingung, als Alternative (IF-Anweisung) oder als Fallunterscheidung (CASE-Anweisung) formuliert, steuert die Auswahl der auszufuehrenden Anweisung.

Syntax:

```
<bedingte Anweisung>::=
        <CASE-Anweisung>
        ¦ <IF-Anweisung>
```

4.5.3.2.1. IF-Anweisung

Durch die IF-Anweisung wird festgelegt, dass die nach THEN folgende Anweisung nur dann ausgefuehrt wird, wenn der boolesche Ausdruck nach IF den Wert TRUE hat.
Wenn dieser Ausdruck den Wert FALSE annimmt, dann wird die nach ELSE folgende Anweisung abgearbeitet.
Ist kein ELSE-Zweig vorhanden, wird die naechste Anweisung ausgefuehrt.

Syntax:

```
<IF-Anweisung>::=
        IF <Ausdruck> THEN <Anweisung> ELSE <Anweisung>
        ¦ IF <Ausdruck> THEN <Anweisung>
```

Bei geschachtelten IF-Anweisungen ist zu beachten, dass ein ELSE-Zweig immer zur letzten IF-Anweisung der Schachtelung gehoert, die noch nicht durch einen ELSE-Zweig abgeschlossen wurde. Gegebenenfalls ist eine Leeranweisung erforderlich.

Nachfolgende IF-Anweisungen sind aequivalent:

```
  (1)  IF <Ausdruck1> THEN
         IF <Ausdruck2> THEN <Anweisung1>
                        ELSE <Anweisung2>;

  (2)  IF <Ausdruck1> THEN BEGIN
         IF <Ausdruck2> THEN <Anweisung1>
                        ELSE <Anweisung2>
       END;
```

Vor ELSE darf **kein** Semikolon stehen, da sonst die IF-Anweisung vorzeitig abgeschlossen wird.

Beispiele:

```
  (a) IF  x < 2.74 THEN y[i]:=x;
  (b) IF (x > 0) AND (x <= 100) THEN BEGIN
        y[i]:= x;
        x:= 0;
        i:= i+1
      END ELSE writeln('Fehler');
  (c) IF Zgr1^.Nachf <> NIL THEN x:= Zgr1^.Wert1;
```

4.5.3.2.2. CASE-Anweisung

Fuer Programmablaeufe, bei denen unter mehr als zwei Moeglichkeiten zu waehlen ist, steht in PASCAL die CASE-Anweisung zur Verfuegung. Diese Anweisung besteht aus einem Ausdruck (Selektor) und einer Liste von Anweisungen, von denen jede durch eine Liste von Fallkonstanten vom Typ des Selektors markiert ist. Die CASE-Anweisung legt fest, dass die Anweisung ausgefuehrt wird, bei der eine Fallkonstante mit dem Ausdruck (Selektor) uebereinstimmt.

Syntax:

```
<CASE-Anweisung>::=
       CASE <Ausdruck> OF
       <Auswahllistenelement> {;<Auswahllistenelement>}
       ELSE  <Anweisung>
       END
       ¦
       CASE <Ausdruck> OF
       <Auswahllistenelement> {;<Auswahllistenelement>}
       END
<Auswahllistenelement>::=
       <Fallkonstantenliste>: <Anweisung>
       ¦ <leer>
<Fallkonstantenliste:=
       <Fallkonstante> {,<Fallkonstante>}
```

Der Ausdruck muss vom ordinalen Typ sein. Entspricht der Wert des Ausdrucks keiner Fallkonstanten, dann wird die Anweisung nach ELSE (wenn vorhanden), sonst die nach CASE folgende Anweisung ausgefuehrt.

```
Beispiele:
  (a) {Programmauswahl entsprechend eines Programm-
Kennzeichens}
  VAR Programmkennzeichen : CHAR;
  BEGIN
    .
    read (Programmkennzeichen);
    CASE Programmkennzeichen OF
      'D','d' : Datenerfassung;
      'F','f' : Fakturierung;
      'B','b' : Buchung;
      'S','s' : Statistik
      ELSE writeln ('Falsches Programm-Kennzeichen!')
    END;
     .

  (b) {Summierung von Betraegen zur Quartalssumme}
      {(Qs1-Qs4), in Abhaengigkeit von der Monats-Nummer (MNR)}

  VAR Qs1, Qs2, Qs3, Qs4,
      Betrag: REAL;
      Monat: INTEGER;
  BEGIN
    .
    CASE Monat OF
      1,2,3:   Qs1:= Qs1+ Betrag;
      4,5,6:   Qs2:= Qs2+ Betrag;
      7,8,9:   Qs3:= Qs3+ Betrag;
      10,11,12:Qs4:= Qs4+ Betrag
      ELSE writeln ('Ungueltige Monats-Nummer!')
    END;
      .
```

4.5.3.3. Zyklusanweisungen

Zyklusanweisungen ermoeglichen die wiederholte Ausfuehrung von bestimmten Anweisungsfolgen.
Wenn die Anzahl der Wiederholungen vorher bekannt ist, dann ist die FOR-Anweisung die schnellste Konstruktion, um dieses Problem zu programmieren. Andernfalls sollte die REPEAT- bzw. WHILE-Anweisung verwendet werden.

Syntax:

```
<Zyklusanweisung>::=
        <WHILE-Anweisung>
        ¦ <REPEAT-Anweisung>
        ¦ <FOR-Anweisung>
```

4.5.3.3.1. WHILE-Anweisung

Die WHILE-Anweisung dient zum Aufbau von Schleifen, die die Ausfuehrung einer Anweisung bereits abweisen, wenn die Bedingung am Anfang nicht erfuellt ist.

Syntax:

```
<WHILE-Anweisung>::=
        WHILE <Ausdruck> DO <Anweisung>
```

Die Anweisung nach DO wird solange (While) wiederholt wie der boolesche Ausdruck nach WHILE den Wert TRUE liefert.
Bei FALSE wird die Schleife verlassen.
Die Anweisung nach DO wird nicht ausgefuehrt, wenn bereits beim Schleifeneintritt der Ausdruck den Wert FALSE liefert.
Der Ausdruck muss im Schleifenkoerper beeinflusst werden, sonst erfolgt eine unendliche Ausfuehrung der Anweisungen nach DO.

Beispiele:

```
  (a) i:= 1; x:= 0
      WHILE (x < 10000.00) AND (i <= 100) DO BEGIN
        i:= i+1;
        x:= x+y[i]
      END;
  (b) {Streichen fuehrender Leerzeichen in einer Zeichenkette}
      WHILE(pos(' ',Kette)=1)AND(Kette<>'') DO delete (Kette,1,1)
```

Die WHILE-Anweisung ist in der Ausfuehrung langsamer als die FOR- und die REPEAT-Anweisung.

4.5.3.3.2. REPEAT-Anweisung

Mit der REPEAT-Anweisung besteht die Moeglichkeit zur Programmierung von Schleifen, die in jedem Falle mindestens einmal durchlaufen werden.

Syntax:

```
<REPEAT-Anweisung>::=
        REPEAT
          <Anweisung> {;<Anweisung>}
        UNTIL <Ausdruck>
```

Die zwischen REPEAT und UNTIL stehenden Anweisungen werden wiederholt, bis (until) der Ausdruck nach UNTIL den Wert TRUE liefert.
Im Gegensatz zur WHILE-Anweisung wird die REPEAT-Schleife also verlassen, wenn der boolesche Ausdruck den Wert TRUE liefert.
Bei FALSE erfolgt eine weitere Wiederholung.
Die REPEAT-Anweisung wird mindestens einmal ausgefuehrt.
Der Ausdruck muss im Schleifenkoerper beeinflusst werden, sonst erfolgt eine unendliche Ausfuehrung der Anweisungen zwischen REPEAT und UNTIL.
Eine Klammerung von mehreren Anweisungen im Schleifenkoerper durch BEGIN und END ist nicht notwendig.

Beispiel:

```
  {Erzwingen einer gueltigen Antwort}
  REPEAT
     write ('Waehlen Sie (J/N):');
     readln (Antwort);
     Antwort:=upcase(Anwort)
  UNTIL (Antwort='J') OR (Antwort='N');
```

Die REPEAT-Anweisung ist in der Ausfuehrung schneller als die WHILE-, aber langsamer als die FOR-Anweisung.

4.5.3.3.3. FOR-Anweisung

Die FOR-Anweisung wird zur Programmierung von Zaehlschleifen verwendet.

Syntax:

```
<FOR-Anweisung>::=
        FOR <Laufvariable> := <Anfangswert> TO <Endwert>DO
                                                    <Anweisung>
        ¦
        FOR <Laufvariable> := <Anfangswert> DOWNTO <Endwert>DO
                                                    <Anweisung>

<Laufvariable>::=
        <Variable ordinalen Typs>
<Anfangswert>::=
        <Ausdruck vom ordinalen Typ>
<Endwert>::=
        <Ausdruck vom ordinalen Typ>
```

Beim Schleifeneintritt bekommt die Laufvariable den Anfangswert zugewiesen. Vor Ausfuehrung der Anweisung nach DO wird der Wert der Laufvariablen mit dem vorgegebenen Endwert verglichen. Bei Ueberschreitung des Endwertes wird die Schleife verlassen, ansonsten werden die Anweisungen ausgefuehrt.
Nach Ausfuehrung der Anweisung wird die Laufvariable um 1 erhoeht (bei TO) bzw. um 1 verringert (bei DOWNTO).
Ist der Endwert bei Schleifeneintritt bereits ueberschritten (bei TO) bzw. unterschritten (bei DOWNTO), dann wird die Schleifenanweisung nicht ausgefuehrt.
Die Laufvariable, der Anfangswert und der Endwert muessen vom gleichen ordinalen Typ sein. Sie koennen im Schleifenkoerper wertmaessig genutzt, duerfen aber nicht veraendert werden.
Der Wert der Laufvariablen nach vollstaendigem Durchlauf der Schleife bei Schleifenaustritt ist undefiniert. Fuer Laufvariablen duerfen nur lokale Variablen verwendet werden.

Beispiele:

```
  (a) VAR Summe  : ARRAY [1..100] OF REAL;
          Artikel: INTEGER;
        .
        .
        .
      FOR Artikel:=1 TO 100 DO
      writeln ('ART-GRUPPE:',Artikel:3,'=',Summe[Artikel]:8:2);

  (b) VAR Kette : STRING;
           i    : INTEGER;
        .
        .
      writeln (Kette);
      FOR i:= 1 TO length (Kette) DO write ('-');
      writeln;

  (c) FOR j:= 1 TO n DO BEGIN
        x:= 0;
        FOR k:= 1 TO n DO x:= x+y[k]*k;
      END;
```

```
  (d) VAR c: (rot, gelb, gruen, blau);
        .
        .
      FOR c:= rot TO blau DO Proz(c);

  (e) x:= 0; j:= 100;
      FOR i:= j DOWNTO 1 DO BEGIN
        x:= x+y[i];
        IF  x > 1000.0 THEN exit      {vorzeitiger Austritt
      END;
```

4.5.3.4. WITH-Anweisung

Innerhalb der WITH-Anweisung koennen die Recordkomponenten-variablen, die durch die WITH-Klausel spezifiziert sind, allein durch den Recordkomponentenbezeichner angegeben werden, d.h. ohne die Angabe der Recordvariablen voranzustellen.

Syntax:

<WITH-Anweisung>::=
WITH <Recordvariablenliste> **DO** <Anweisung>
<Recordvariablenliste>::=
<Recordvariable> {,<Recordvariable>}

Bei einer Schachtelung von WITH-Anweisungen groesser als zwei Ist ein Compilerschalter erforderlich {($w.)}.

Beispiele:

```
  (a) TYPE Daten=RECORD
                   Adresse : STRING [100];
                   Konto   : STRING [15];
                   Umsatz  : REAL;
                   Datum   : STRING [8]
                 END;
      VAR Kunde: Daten;

  {Die nachfolgenden Anweisungen sind aequivalent:}
  (1) Kunde.Adresse:= 'Lampen-Mueller, 50 Erfurt, Am Hang 4';
      Kunde.Konto  := '4444-46-1100';
      Kunde.Umsatz :=  6000.00;
      Kunde.Datum  := '12.12.84';
  (2) WITH Kunde DO BEGIN
        Adresse:= 'Lampen-Mueller, 50 Erfurt, Am Hang 4');
        Konto  := '4444-46-1100';
        Umsatz :=  6000.00;
        Datum  := '12.12.84'
      END;
```

```
(b) TYPE Person=: RECORD
                    Mitarbeiter= RECORD
                                   Name, Ort, Str: STRING[20];
                                   Gdat: STRING [8]
                                   Verh: BOOLEAN
                                 END;
                .
                .
    VAR Angest, Arb, Lehrl: Person;
    WITH Anges, Mitarb DO BEGIN
      Name:= 'Paul Meyer';
      Ort := '5230 Soemmerda';
      Str := 'Park-Str. 5';
      Gdat:= '12.10.46';
      Verh:= TRUE
    END;
```

4.6. Nutzerdefinierte Prozeduren und Funktionen

4.6.1. Deklaration von Prozeduren und Funktionen

4.6.1.1. Prozedurkopf und -block

Der Prozedurkopf besteht aus dem reservierten Wort PROZEDURE, dem ein Bezeichner folgt, der als Name der Prozedur bezeichnet wird. Gewoehnlich folgt ihm eine formale Parameterliste.

Syntax:

<Prozedurkopf>::=
PROCEDURE <Prozedurbezeichner> **<Parameterliste>**
¦ **PROCEDURE** <Prozedurbezeichner>

<Parameterliste> wird in Ziffer 4.6.2.2. definiert
Der Prozedurblock besteht aus einem Deklarationsteil und einem Anweisungsteil.

Der Deklarationsteil einer Prozedur hat die gleiche Form wie bei einem Programm. Alle in der formalen Parameterliste im Deklarationsteil erklaerten Bezeichner sind lokal zur Prozedur und zu jeder Prozedur in ihr. Dieser Bereich heisst Gueltigkeitsbereich der Bezeichner. Ausserhalb dieses Bereiches sind sie nicht bekannt. Eine Prozedur kann jede in einem zu ihr aeusseren Block definierte Konstante, Type, Variable, Prozedur oder Funktion verwenden.
Der Anweisungteil spezifiziert die Operationen, die ausgefuehrt werden sollen, wenn die Prozedur aufgerufen wird. Er hat die Form einer Verbundanweisung. Sie endet also mit einem Semikolon. Wird der Prozedurbezeichner selbst innerhalb des Anweisungsteiles verwendet, wird die Prozedur rekursiv ausgefuehrt. Es muss dann beachtet werden, dass zu diesem Zeitpunkt bei der Compilierung die A-Compiler-Direktive passiv {$A-} gesetzt ist.

4.6.1.2. Funktionskopf und -block

Der Funktionskopf ist mit dem pPozedurkopf aequivalent, ausser dass der Funktionskopf mit dem reservierten Wort FUNKTION eroeffnet wird und dass auch der Typ des Ergebnisses mit definiert werden muss. Dies wird durch Anfuegung eines Doppelpunktes und eines Types an den Funktionskopf erreicht.

Syntax:

<Funktionskopf>::=
FUNCTION <Funktionsbezeichner><Parameterliste>
:<Ergebnistyp>
¦ **FUNCTION** <Funktionsbezeichner>:<Ergebnistyp>

<Parameterliste wird in Ziffer 4.6.2.2. definiert.

Der Ergebnistyp einer Funktion muss ein einfacher Typ (d.h. INTEGER, REAL, BOOLEAN, CHAR), ein Stringtyp oder ein Zeigertyp sein.

Der Deklarationsteil einer Funktion ist der gleiche wie bei einer Prozedur

Der Anweisungsteil einer Funktion ist eine Verbundanweisung. Innerhalb des Anweisungsteiles muss wenigstens eine Ergibtanweisung auftreten, die dem Funktionsbezeichner einen Wert zuweist. Die letzte dieser Ergibtanweisungen zum Funktionbezeichner ergibt den Wert der Funktion. Wenn der Funktionsbezeichner selbst als Funktionsaufruf im Anweisungsteil der Funktion auftritt, dann wird die Funktion rekursiv aufgerufen. In diesem Falle muss zu diesem Zeitpunkt die A-Compiler-Direktive {$A-} passiv sein.

Bei der Definition eines Funktionstyps ist zu beachten, dass ein als Parameter oder Ergebnistyp verwendeter strukturierter Typ vorher als Typbezeichner erklaert sein muss.
Aus diesem Grunde ist die folgende Konstruktion nicht erlaubt:

```
FUNCTION Kette(Zeile: Linie) : STRING[80];
```

Man muss stattdessen vorher den Typ STRING[80] durch einen Bezeichner erklaeren und mit diesem dann den Typ des Funktionsergebnisses definieren:

```
TYPE
   Str80 = STRING[80];
FUNCTION Kette(Zeile: Linie) : Str80;
```

Wegen der Art der Implementation der Prozeduren WRITE und WRITELN darf eine Funktion, die irgendeine der Standardprozeduren READ, READLN, WRITE oder WRITELN verwendet, niemals durch einen Ausdruck in einer WRITE oder WRITELN-Anweisung aufgerufen werden.

4.6.2. Datenaustausch

4.6.2.1. Blockkonzept

PASCAL ist eine blockorientierte Sprache. Blockorientiert bedeutet, dass alle definierten und deklarierten Objekte, also Konstanten, Typen, Variablen und Unterprogramme in einem gesamten Block gueltig sind, in dem sie vereinbart (eingefuehrt) wurden. Eine Ausnahme bilden lediglich Marken. In den eingelagerten Prozeduren und Funktionen koennen also alle Objekte ohne eigene Deklaration benutzt werden, die im uebergeordneten Block enthalten sind. Solche Objekte nennt man deshalb global.
Eine Kollision wuerde entstehen, wenn im Vereinbarungsteil eines Unterprogramms ein Objekt unter einem Bezeichner deklariert wird, der in der uebergeordneten Programmeinheit bereits benutzt wurde. Entsprechend dem Blockkonzept waere er auch im Unterprogramm noch gueltig. PASCAL legt fest, dass in diesem Falle die (lokale) Deklaration im Unterprogramm die globale Gueltigkeit aufhebt, natuerlich nur lokal fuer den Block dieses Unterprogramms und auch genau nur fuer dieses Objekt.
Man beachte, dass sich die Gueltigkeit entsprechend dem Blockkonzept nur "nach innen", also vom Globalen zum Lokalen hin oeffnet. Die in einem Unterprogramm definierten und deklarierten Objekte - bei Variablen spricht man von lokalen Variablen - sind fuer die uebergeordnete Programmeinheit nicht gueltig, und es kann auch nicht auf sie zugegriffen werden. Das Blockkonzept geht aus dem folgenden Programmausschnitt hervor:

```
PROGRAM Hauptprogramm;
.
TYPE Feld = ARRAY[1..20] OF CHAR;
VAR  x,y  : Feld;
     i,j,k: INTEGER;
PROCEDURE Prozedur;
  VAR z : Feld;
      i : INTEGER;
  {Gueltig auch x,y,j,k}
   .
  END;
  FUNCTION Funktion : Typ;
  VAR a   : Feld;
      j,z : INTEGER;
  {Gueltig auch x,y,i,k}
   .
  END;
BEGIN {Hauptprogramm}
{Gueltig  sind x,y,j,i,k}
.
END.
```

Die Gueltigkeit der globalen Variablen ist als Kommentar eingefuegt.
Das Blockkonzept regelt nicht nur die Gueltigkeit von Bezeichnern innerhalb des Programmtextes. Es legt auch fest, dass lokale Variablen beim Verlassen eines Unterprogramms ihren Wert verlieren.

4.6.2.2. Parameter

Werte koennen den Prozeduren oder Funktionen durch Parameter uebergeben werden. Durch diese Parameter wird ein Substitutionsmechanismus bereitgestellt, der erlaubt, die Logik des Unterprogrammes mit verschiedenen Anfangswerten zu versehen, so dass es entsprechend verschiedene Ergebnisse produziert.
Die Prozeduranweisung oder der Funktionsbezeichner, die das entsprechende Unterprogramm aufrufen, koennen eine Liste von Parametern enthalten, die man als die aktuellen Parameter bezeichnet. Diese werden den formalen Parametern uebergeben, die im Kopf des Unterprogrammes spezifiziert sind. Die Zuordnung der Parameter bei der Uebergabe erfolgt in der Reihenfolge ihres Auftretens in der Parameterliste. PASCAL unterstuetzt zwei unterschiedliche Methoden der Parameteruebergabe: Uebergabe der Parameter durch Uebergabe eines Wertes (Wertuebergabe, Wertparameter) und Uebergabe der Parameter durch Substitution der Adressen (Referenz, Variablenparameter). Hier sind ausserdem typlose Parameter erlaubt.
Die Stacktechnik beim Parameteraustausch ist im Anhang F beschrieben.

Syntax:

```
<Parameterliste>::=
    <formaler Parameter>{;<formaler Parameter>}

<formaler Parameter>::=
        VAR <Segment>
<Segment>::=
      <Parameter>{,<Parameter>}:<Typ>
     ¦ <Parameter>{,<Parameter>}
<Parameter>::= <Variablenbezeichner>
```

4.6.2.2.1. Variablenparameter

Bei Variablenparametern wird die Adresse des aktuellen Parameters an die Prozedur uebergeben (call by reference). Dabei arbeiten Prozedur und rufendes Programm mit der gleichen Variablen, so dass eine Uebermittlung von Ergebnissen moeglich ist, z.B.

```
    PROCEDURE Test (VAR Fehler: BOOLEAN);
```

Kennzeichen fuer Variablenparameter ist das VAR im Segment.

4.6.2.2.2. Wertparameter

Bei Werteparametern wird der Wert des aktuellen Parameters (aus der Prozeduranweisung) in den formalen Parameter der Prozedur uebertragen (call by value).
Eine Rueckgabe von Ergebnissen ist nicht moeglich, z.B.

```
    PROCEDURE Kombination (a: REAL; b: INTEGER);
```

Hier fehlt das VAR im Segment

4.6.2.2.3. Ungetypte Variablenparameter

Ist der Typ eines Parameters nicht definiert, d.h., enthaelt der Parameterteil im Kopf des Unterprogrammes keine Typdefinition, dann wird der Parameter als ungetypt bezeichnet. Der aktuelle Parameter kann dann ein beliebiger Typ sein. Aus diesem Grunde kann man ungetypte formale Parameter nur dort verwenden, wo der Datentyp ohne Bedeutung ist. Dies ist beispielsweise bei den Parametern von ADDR, BLOCKREAD, BLOCKWRITE, FILLCHAR oder MOVE und bei Adress-Spezifikationen von absoluten Variablen der Fall.
Im folgenden Beispiel wird bei der Prozedur SCHALTER die Verwendung ungetypter Parameter demonstriert. Sie uebertraegt den Inhalt der Variablen a1 nach a2 und von a2 nach a1.

```
PROCEDURE Schalter(VAR a1p,a2p; Anzahl : INTEGER);
TYPE
     a = ARRAY[1..Max] OF BYTE;
VAR
     a1      : a ABSOLUTE a1p;
     a2      : a ABSOLUTE a2p;
     Temp    : BYTE;
     Zaehler : INTEGER;
BEGIN
     FOR Zaehler := 1 TO Anzahl DO
     BEGIN
          Temp        := a1[Zaehler];
          a1[Zaehler] := a2[Zaehler];
          a2[Zaehler] := Temp
     END
END;
```

Definiert man:

```
TYPE
     Matrix = ARRAY[1..50,1..25] OF REAL;
VAR
     TestMatrix,BestMatrix : Matrix;
```

dann kann man Schalter zum Austauschen des Inhaltes der beiden Matrizen verwenden. Der Prozeduraufruf lautet dann:

```
SCHALTER(TestMatrix,BestMatrix,Umfang(TestMatrix));
```

4.6.3. FORWARD-Deklaration

Ein Unterprogramm wird vorwaerts deklariert, indem man seinen Kopf separat von seinem Block spezifiziert. Dieser separate Unterprogrammkopf ist exakt der gleiche wie der eines normalen Unterprogrammes, ausser dass er mit dem reservierten Wort FORWARD endet. Der Block selbst folgt spaeter innerhalb des gleichen Deklarationsteiles. Der Block beginnt mit einer Kopie des vorher definierten Kopfes ohne Parameter, Typen,usw., d.h. nur mit dem Namen. Die FORWARD-Deklaration ist nicht fuer OVERLAY-Unterprogramme erlaubt.

Beispiel:

```
PROCEDURE xyz (VAR a:REAL; b:CHAR); FORWARD;
```

4.6.4. EXTERNAL-Deklaration

Das reservierte Wort EXTERNAL wird zur Definition externer Prozeduren und Funktionen verwendet. Typisch ist die Verwendung fuer in Maschinencode geschriebene Prozeduren oder Funktionen.

Ein externes Unterprogramm hat keinen Block, d.h. keinen Deklarationsteil und keinen Anweisungsteil. Es wird nur der Unterprogrammkopf spezifiziert, dem unmittelbar das reservierte Wort EXTERNAL und eine Integerkonstante folgt, die die Adresse des Unterprogramms definiert.

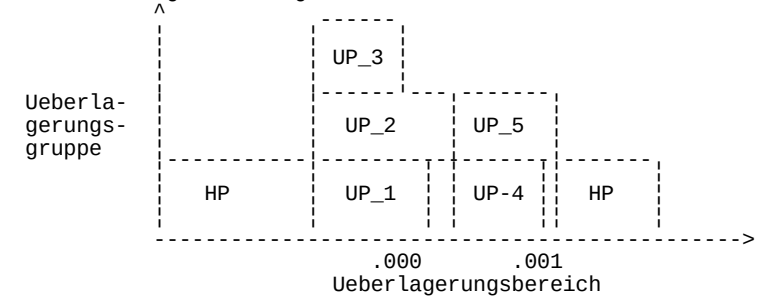
Beispiele:
```
  PROCEDURE DiskReset; EXTERNAL $EC00;
  FUNCTION  Iostatus : BOOLEAN; EXTERNAL $D123;
```

Parameter koennen an externe Unterprogramme uebergeben werden, und die Syntax ist dabei exakt die gleiche wie bei normalen Prozedur- oder Funktionsaufrufen:

```
 PROCEDURE Plot(x,y:INTEGER); EXTERNAL $8003;
 PROCEDURE Sortier(VAR List: Anzahl); EXTERNAL $3000;
```

4.6.5. OVERLAY-Strukturen

Bei Programmen, die die Kapazitaet des Hauptspeichers uebersteigen, kann eine Ueberlagerungsstruktur fuer die Unterprogramme erzeugt werden. Die Hauptspeicherbelegung in einem solchen Falle zeigt das folgende Schema.



Das Prinzip besteht darin, nichtaktive Unterprogramme einer Ueberlagerungsgruppe auf den externen Datenspeicher auszulagern. Erst im Falle ihres Aufrufes werden sie in den Hauptspeicher transportiert. Sie ueberschreiben ein eventuell vorher aktives Unterprogramm derselben Ueberlagerungsgruppe. Der Vorteil besteht darin, dass der Ueberlagerungsbereich fuer eine Ueberlagerungsgruppe nur so gross sein muss wie das groesste Unterprogramm dieser Gruppe. Die moegliche Einsparung an Hauptspeicherkapazitaet kann betraechtlich sein. Auf diese Weise ist es moeglich, grosse Programme auf Rechnern mit relativ kleiner Hauptspeicherkapazitaet auszufuehren.
Die technische Realisierung einer Ueberlagerungsstruktur erfolgt waehrend des einheitlichen Compiler-/Linkervorgangs. Dabei entsteht je Ueberlagerungsgruppe ein Ueberlagerungsfile, das sofort auf einen externen Datentraeger ausgelagert wird. Es enthaelt den Maschinencode aller zur Gruppe gehoerenden Unterprogramme. Die Ueberlagerungsfiles erhalten den Filenamen des Hauptprogramms und eine Nummer 000, 001 ... als Filenamenserweiterung.
Dem PASCAL-System muss mitgeteilt werden, dass eine Ueberlagerungsstruktur erforderlich ist und welche Unterprogramme eine Ueberlagerungsgruppe bilden sollen. Dabei ist zu beachten, dass sich Unterprogramme der gleichen Gruppe niemals untereinander rufen, aktivieren koennen. Sonst gibt es fuer die Bildung der Gruppen noch die folgenden Hinweise:

a) Annaehernd gleich grosse Unterprogramme sollten in eine Gruppe aufgenommen werden. Die Einsparung an Hauptspeicherkapazitaet ist dann besonders gross.

b) Haeufig aktive Programme sollten verschiedenen Gruppen zugeordnet werden. Dadurch wird der Zeitverzug, der durch das staendige Laden der aktivierten Unterprogramme entsteht, geringer.
c) In OVERLAY-Unterprogrammen ist keine Rekursion erlaubt.

Fuer den Aufbau einer Ueberlagerungsstruktur gilt folgender Verfahrensweg. Ein Unterprogramm, das Bestandteil einer Ueberlagerungsstruktur werden soll, erhaelt vor dem Schluesselwort PROCEDURE oder FUNCTION den Zusatz OVERLAY.
Alle aufeinanderfolgenden Unterprogramme mit dem Schluesselwort OVERLAY bilden eine Ueberlagerungsgruppe. Die Gruppe gilt als abgeschlossen, wenn ein folgendes Unterprogramm kein OVERLAY enthaelt. Folgt nach diesem Unterprogramm ohne OVERLAY wieder ein Unterprogramm mit OVERLAY, so wird eine neue Ueberlagerungsgruppe eroeffnet. Da die Reihenfolge der Unterprogrammdeklarationen, gegebenenfalls mit FORWARD, vom Programmierer frei gewaehlt werden kann und auch "leere" (Pseudo-) Unterprogramme deklariert werden koennen, ist auf diese Art eine einfache, aber vollstaendige Mitteilung an das PASCAL 880/S moeglich. OVERLAY-Unterprogramme koennen auch geschachtelt werden. Da solche Programmeinheiten sich dann gegenseitig rufen, wird eine Ueberlagerungsgruppe eroeffnet.

Beispiel:

```
    :
  OVERLAY PROCEDURE UP_1;          {Overlay:  .000}
  BEGIN
    :
  END;
  OVERLAY PROCEDURE UP_2;
  BEGIN
    :
  END;
  OVERLAY PROCEDURE UP_3;
  BEGIN
    :
  END;
  PROCEDURE tab (anz:INTEGER);     {Prozedur zur Trennung}
  BEGIN                            {oder zwei Overlay-Gebiete}
    :
  END;
  OVERLAY PROCEDURE UP_4;          {Overlay:  .001}
  BEGIN
    :
  END;
  OVERLAY PROCEDURE UP_5;
  BEGIN
    :
  END;
  BEGIN
    :                              {Hauptprogramm}
  END.
```

4.7. Standardprozeduren und Standardfunktionen (ohne Filearbeit und Pointer)

Die nachfolgenden Standardprozeduren bzw. -Funktionen werden getrennt nach ihren Anwendungsbereichen dargestellt:
- STRING-Prozeduren und –funktionen
- arithmetische Standardfunktionen
- Skalarfunktionen
- Konvertierungsfunktionen (ohne Pseudofunktionen ORD, CHR, PTR und Retyping)
- bildschirmorientierte Prozeduren
- sonstige Prozeduren und Funktionen.

Nachfolgend werden folgende Abkuerzungen verwendet:

```
  <Quelle>      = Zeichenkette oder ARRAY-OF-CHAR
  <Ziel>        = Zeichenkette
  <Anzahl>      = INTEGER-Ausdruck / BYTE-Ausdruck
  <Kette>       = Zeichenkette(STRING)
  <Zeichen>     = CHAR-Variable/ - Konstante
  <Integer>     = INTEGER-Ausdruck
  <Real>        = REAL-Ausdruck
  <Ordinale>    = Ordinal-Typ (INTEGER,CHAR,BYTE,BOOLEAN)
  <Position>    = INTEGER-Ausdruck / BYTE-Ausdruck
  <x>           = INTEGER-Ausdruck / REAL-Ausdruck
```

4.7.1. STRING-Funktionen und -Prozeduren

CONCAT-Funktion

Syntax:

```
  concat  (<Quelle>{,<Quelle>})
```

Die Funktion CONCAT (Typ STRING) liefert einen STRING, der die Zusammenfuegung der STRING's enthaelt.
Wenn die gesamte Laenge 256 Bytes uebersteigt, entsteht ein Laufzeitfehler. Man kann mit dem "+"-Operator das gleiche erhalten. CONCAT sichert nur die Kompatibilitaet mit anderen Compilern. Die Quellen koennen STRING-Variablen, ARRAY-OF-CHAR-Variablen, STRING-Konstanten oder Zeichen (CHAR) sein.

Beispiel:

```
  PROCEDURE Conc;
  VAR A,B: STRING[30];
  BEGIN
    A:= 'Sprachbeschreibung ';
    B:= '1986';
    writeln (concat (A,'PASCAL ','880/S ','B'))
  END;      {Ausgabe: Sprachbeschreibung PASCAL 880/S 1986}
```

COPY-Funktion

Syntax:

```
  copy  (<Kette>,<Position>,<Anzahl>)
```

Diese Funktion (Typ STRING) liefert aus dem STRING <Kette> ab <Position> einen Teil-STRING in der Laenge <Anzahl>.

Wenn <Position> groesser als Laenge <Kette> ist, besteht das Ergebnis aus der leeren Zeichenkette ''.
Wenn <Position> + <Anzahl> ausserhalb von <Kette> liegt, werden nur die innerhalb von <Kette> liegendenZeichen zurueckgegeben. Liegt <Position> nicht in 1..255, so entsteht ein Laufzeitfehler.

Beispiel:
```
  PROCEDURE Cop;
  VAR A: STRING[80];
  BEGIN
    A:='Zeichenkettenfeld';
    writeln (copy(A,8,5))                    {Ausgabe: kette}
  END;
```

DELETE-Prozedur

Syntax:

  delete (<Kette>,<Position>,<Anzahl>)

In <Kette> werden ab <Position> <Anzahl> Zeichen geloescht (und verdichtet).
Die Parameter <Position> und <Anzahl> sind vom Typ INTEGER.
Wenn <Position> groesser als die Laenge von <Kette> ist, wird kein Zeichen geloescht. Wenn <Position>+<Anzahl> ausserhalb der Zeichenkette liegt, werden nur die Zeichen geloescht, die ab <Position> innerhalb liegen. Liegt <Position> nicht in 1..255, wird ein Laufzeitfehler erzeugt.

Beispiel:
```
  PROCEDURE Del;
  VAR TX: STRING[50];
  BEGIN
    TX:= 'Programmierung in PASCAL';
    delete (TX,9,9);
    writeln (TX)                {Ausgabe: Programm PASCAL}
  END;
```

INSERT-Prozedur

Syntax:

  insert (<Quelle>,<Ziel>,<Position>)

Die Prozedur INSERT fuegt in den Ziel-STRING <Ziel> an der Position <Position> den Quell-STRING <Quelle> ein.
Als Quelle sind Konstanten oder Variablen vom Typ STRING und CHAR zugelassen.
Ist <Position> groesser als die Laenge von <Ziel> wird <Quelle> an <Ziel> angefuegt. Wenn das Ergebnis laenger als die maximale Laenge von <Ziel> ist, werden die ueberstehenden Zeichen abgeschnitten und <Ziel> erhaelt nur die links stehenden. Wenn <Position> ausserhalb von 1..255 liegt, entsteht ein Laufzeitfehler.

```
Beispiel:
  PROCEDURE Ins;
  VAR A: STRING[80];
      B: STRING[20];
  BEGIN
    A:= 'Kombinat ROBOTRON Soemmerda');
    B:= 'Bueromaschinenwerk ';
    insert (B,A,19);
    writeln (A)             {Ausgabe: Kombinat ROBOTRON Buero-}
  END;                                {maschinenwerk Soemmerda}
```

LENGTH-Funktion

Syntax:

length (<Kette>)

Diese Funktion liefert die Laenge des STRING <Kette> als INTEGER-Wert.

Beispiel:

```
  PROCEDURE Len;
  VAR A: STRING[40];
  BEGIN
    A:= 'Leipzig';
    writeln (length(A),'/',length('Soemmerda'));
  END;                                              {Ausgabe: 7/9}
```

POS-Funktion

Syntax:

pos (<Kette>,<Quelle>)

Diese Funktion liefert die Position des 1.Auftretens von <Kette> im STRING <Quelle> als INTEGER-Wert.
Wenn <Kette> nicht in <Quelle> gefunden wird, dann liefert die Funktion den Wert 0.
Fuer <Kette> ist eine Konstante oder Variable vom Typ STRING oder CHAR zugelassen. ARRAY-OF-CHAR wird wie ein String fester Laenge behandelt.

Beispiel:

```
  PROCEDURE Posf;
  VAR A,B: STRING[30];
  BEGIN
    A:= 'Standardfunktion';
    B:= 'fun';
    writeln (pos(B,A),'/', pos('a',A),'/', pos('xy',A))
  END;                                            {Ausgabe: 9/3/0}
```

STR-Prozedur

Syntax:

str (<x>,<Kette>)

Die STR-Prozedur konvertiert den numerischen Wert von <x> (INTEGER bzw. REAL-Typ) in eine Zeichenkette und speichert sie in <Kette> ab. Die Konvertierung kann durch die von writeln/ write bekannten Formatparameter gesteuert werden.

Beispiel:

```
PROCEDURE Strt;
VAR  i:INTEGER;
     j:REAL;
     zk1,zk2:STRING[10];
BEGIN
  i:=1234;
  j:=2.5E4;
  str(i:5,zk1);          {zk1 =' 1234'}
  str(j:10:0,zk2)        {zk2 ='     25000'}
END;
```

VAL-Prozedur

Syntax:

val (<Kette>,<x>,<Code>);

Der STRING-Ausdruck <Kette> muss den Regeln einer numerischen Konstanten genuegen. Weder fuehrende noch nachfolgende Leerzeichen sind erlaubt. Die Prozedur VAL konvertiert die Konstante zu einem Wert vom gleichen Typ wie <x>(INTEGER-/REAL-Typ) und speichert diesen Wert in <x> ab. Wird kein Fehler festgestellt, ist der Wert der Variablen <Code>=0. Andernfalls erhaelt <Code> den Wert der Position des ersten fehlerhaften Zeichens in <Kette> und der Wert von <x> ist undefiniert.

Beispiele:

```
str1 = '234';
val(str1,I,Result);     { I = 234        Result = 0 }

str2:= ' 234';
val(str2,I,Result);     { I = undef.     Result = 1 }

str3:= '2.5E4';
val(str3,r,Result);     { r = 25000      Result = 0 }
```

4.7.2. Arithmetische Funktionen

ABS-Funktion

Syntax:

abs (<x>)

Die Funktion liefert den Absolutwert der INTEGER- oder REAL-Zahl <x>.

Das Ergebnis ist vom gleichen Typ wie das Argument.

Beispiele:
```
  write (abs (-3.21));      {= 3.21}
  write (abs (-127));       {= 127}
  write (abs (18.1299));    {= 18.1299}
```

ARCTAN-Funktion

Syntax:

  arctan (<x>)

Die Funktion liefert den Arcustangens von <x> als reelle Zahl.
Das Argument <x> ist im Bogenmass anzugeben.

Beispiele:
```
  write (arctan (1));               {= 7.85398 E-01  (0.7854)}
  write (arctan (1.222));           {= 8.84977 E-01  (0.8850)}
```

COS-Funktion

Syntax:

  cos (<x>)

Die Funktion liefert den Cosinus von <x> als reelle Zahl.
Das Argument <x> ist im Bogenmass anzugeben.

Beispiele:
```
  write (cos (1));                  {= 5.40302 E-01  (0.5403)}
  write (cos (1.4444));             {= 1.26061 E-01  (0.1261)}
```

EXP-Funktion

Syntax:

  exp (<x>)

Die Funktion liefert die Exponentialfunktion e^x als reelle Zahl.

Beispiele:
```
  write (exp (8));                 {= 2.98094 E+03  (2980.94)}
  write (exp (-12.5555));          {= 3.52547 E-06}
```

FRAC-Funktion

Syntax:

  frac (<Real>)

Die Funktion ermittelt den gebrochenen Teil von <Real>.
Das Ergebnis ist vom Typ REAL.

Beispiel:
```
  write(frac (123.37));                {=0.37}
```

INT-Funktion

Syntax:

int (<Ausdruck>)

Die Funktion ermittelt den ganzen Teil von <Ausdruck>. Ausdruck ist vom Typ INTEGER oder REAL. Das Ergebnis ist je nach Argument vom Typ INTEGER oder REAL.

Beispiele:
```
write(int (5.27));          {= 5}
r:= int(5);                 {wenn r=REAL dann 5.}
```

LN-Funktion

Syntax:

ln (<x>)

Die Funktion liefert den natuerlichen Logarithmus von <x> als reelle Zahl.

Beispiele:
```
write (ln (127));
write (ln (18.5555));
```

SIN-Funktion

Syntax:

sin (<x>)

Die Funktion liefert den Sinus von <x> als reelle Zahl. Das Argument <x> ist im Bogenmass anzugeben.

Beispiele:
```
write (sin (1));
write (sin (1.684));
```

SQR-Funktion

Syntax:

sqr (<x>)

Die Funktion liefert das Quadrat von x. Das Argument kann vom Typ INTEGER oder REAL sein. Das Ergebnis ist gleich dem Typ von x.

Beispiele:
```
write (sqr (9.0000));
write (sqr (-12));
```

SQRT-Funktion

Syntax:

sqrt (<x>)

Die Funktion liefert die Quadratwurzel der Zahl <x> (REAL, INTEGER). Das Ergebnis ist vom Typ REAL.

Beispiel:
```
write (sqrt (100));
```

4.7.3. Skalarfunktionen

PRED-Funktion

Syntax:

pred (<Ordinale>)

Die Funktion pred liefert den Vorgaenger von <Ordinale>.
<Ordinale> ist vom ordinalen Typ.
i:= pred(i) ist schneller als i:= i - 1.

Beispiele:
```
    write(pred('C');             {= B}
    k:=5; write(pred(k));        {= 4}
```

SUCC-Funktion

Syntax:

succ (<Ordinale>)

Die Funktion liefert den Nachfolger von <Ordinale>
<Ordinale> ist vom ordinalen Typ.
i:= succ(i) ist schneller als i:= i + 1.

Beispiele:
```
    write(succ('H'));            { = I}
    k:= 29; write(succ(k));      { = 30}
    k:= succ(k);                 { k = 30}
```

ODD-Funktion

Syntax:

odd (<Integer>)

Die Funktion liefert den booleschen Wert des Ausdrucks <Integer> mod 2 <> 0, d.h., fuer geradzahlige Integer-Werte liefert die Funktion FALSE, fuer gerade Werte TRUE.

```
Beispiel:
  PROCEDURE od;
  VAR i:INTEGER;
  BEGIN
    readln(i);
    IF odd(i) THEN writeln('I = ungerade Zahl')
              ELSE writeln('I = gerade Zahl')
  END;
```

4.7.4. Konvertierungsfunktionen (ohne Pseudofunktionen)

Die Pseudofunktionen der Konvertierung CHAR, ORD, PTR und das Retyping sind in Ziffer 4.3.3.5.2. dargestellt.

ROUND-Funktion

Syntax:

round (<Real>)

Die Funktion liefert die ganzzahlige Rundung (Integer) der reellen Zahl <Real>.
Die Funktion aehnelt der Funktion TRUNC. Das Ergebnis wird hier jedoch auf die naechste ganze Zahl auf- oder abgerundet.
Betraegt der gebrochene Teil genau 0.5, dann wird bei positiven Zahlen auf- und bei negativen Zahlen abgerundet.

```
Beispiele:
  write (round (1.463));                          {= 1}
  write (round (12.864));                         {= 13}
  write (round (-127.3468));                      {= -127}
```

TRUNC-Funktion

Syntax:

trunc (<Real>)

Die Funktion liefert den ganzzahligen Teil (INTEGER) der reellen Zahl <Real>. Der Realteil wird abgeschnitten.

```
Beispiele:
  write (trunc (31.6781));                          {= 31}
  write (trunc (-6.18));                            {= -6}
```

4.7.5. Bildschirmorientierte Prozeduren

CLREOL-Prozedur

Syntax:

clreol

Diese Prozedur loescht alle Zeichen ab Cursorposition bis zum Ende der Zeile, ohne die Cursorposition zu veraendern (Steuerzeichen $16.

CLRSCR-Prozedur

Syntax:

clrscr

Diese Prozedur loescht den Bildschirm und setzt den Cursor in die linke obere Ecke (Steuerzeichen $0C).

DELLINE-Prozedur

Syntax:

delline

Diese Prozedur loescht die Zeile, in der der Cursor steht und schiebt alle darunterstehenden Zeilen um eine Zeile nach oben.

INSLINE-Prozedur

Syntax:

insline

Diese Prozedur fuegt an der Cursorposition eine leere Zeile ein und schiebt alle Zeilen unterhalb um eine Zeile nach unten. Die letzte Zeile wird weggerollt.

GOTOXY-Prozedur

Syntax:

gotoxy (<xpos>,<ypos>)

Diese Prozedur setzt den Cursor an die Position auf dem Bildschirm, die durch die Integeraudruecke <xpos> (horizontaler Wert oder Zeile und <ypos> (vertikaler Wert oder Spalte) angegeben werden. Die linke obere Ecke (Home-Position) ist (1,1).

4.7.6. Sonstige Funktionen und Prozeduren

ADDR-Funktion

Syntax:

addr (<Objekt>)

Die Funktion liefert die Speicheradresse (INTEGER) von <Objekt>.

Das Argument <Objekt> kann sein:
- Variable
- Prozedur
- Funktion

Konstanten sind als Argument nicht erlaubt.

Beispiel:

```
  PROCEDURE Addr_demo (Par:REAL);
  VAR Satz: RECORD  J: INTEGER;
                    B: BOOLEAN
            END;
      Adres: INTEGER;
          R: REAL;
          S: ARRAY[1..100] OF CHAR;
  BEGIN
    writeln (addr (Addr_demo));
    writeln (addr (Par));
    writeln (addr (Satz));
    writeln (addr (Satz.B));
    writeln (addr (S))
  END;
```

DELAY-Prozedur

Syntax:

  delay (<Time>)

Diese Prozedur erzeugt eine Warteschleife, die in ungefaehr soviel Millisekunden durchlaufen wird, wie im Argument angegeben ist. Die exakte Zeit kann wegen der unterschiedlichen Hardware etwas davon abweichen.

CHAIN- und EXECUTE-Prozedur

Syntax:

  chain (<Filevariable>)
  execute (<Filevariable>)

Die Prozeduren CHAIN und EXECUTE erlauben von einem Programm aus die Aktivierung anderer Programmfiles. Eine Verkettung von Programmen macht sich erforderlich, wenn Programme groesser sind, als der verfuegbare Speicherplatz und OVERLAY-Strukturen ungeeignet sind.
<Filevariable> ist die Filevariable eines ungetypten Files. Sie muss vorher mittels ASSIGN einem Diskettenfile zugewiesen sein, aber nicht eroeffnet(RESET / REWRITE) werden.
Die Prozedur CHAIN wird verwendet, um ein CHN-File abzuarbeiten, welches vorher mit der Compiler-Option M compiliert wurde (siehe auch Punkt 2.12).
Das CHN-File wird an die Stelle im Speicher geladen und bei der Adresse gestartet, die das aktuelle Programm hat, d.h. die Adresse, die bei der Uebersetzung des aktuellen Programms angegeben wurde. Das neu gestartete Programm verwendet auch die bereits im Speicher stehende Pascalbibliothek. Aus diesem Grund muessen beide die gleiche Startadresse haben.
Die Prozedur EXECUTE wird verwendet, um ein COM-File abzuarbeiten dass einen abarbeitungsfaehigen Code enthaelt. Existiert das Diskfile nicht, tritt ein E/A-Fehler auf.
Die Programmgroesse hat bei der Verkettung keine Bedeutung, allerdings muessen auszutauschende Daten oberhalb des groessten Programms stehen, wenn eine Datenuebergabe erforderlich ist.

Dieser Datenaustausch kann auf drei Wegen ausgefuehrt werden:
a.) Gemeinsam benutzte globale Variablen (gleicher Vereinbarungsteil notwendig)
b.) Verwendung von absoluten Variablen (ABSOLUTE)
c.) Verwendung von Diskettenfiles.

Beispiel (ohne Datenaustausch):
```
  PROGRAM Eins;
  {Programmierter Start des Programms Zwei}
  VAR  Start    : FILE;
  BEGIN
    ASSIGN(Start,'ZWEI.COM');
    execute(Start)
  END.
```

Eine eventuelle erforderliche Rueckkehr nach CHAIN oder EXECUTE ins rufende Programm muss mit EXECUTE selbst organisiert werden.

FILLCHAR-Prozedur

Syntax:

fillchar (<Ziel>,<Anzahl>,<Zeichen>)

Uebertragung von <Anzahl> gleicher Zeichen <Zeichen> in einen Speicherbereich, beginnend ab dem ersten Byte.
Wenn <Anzahl> groesser ist als die Laenge von <Ziel>, dann werden die nachfolgenden Daten ueberschrieben.
<Zeichen> ist eine Variable oder Konstante vom Typ CHAR. Bei <Zeichen> kleiner 255 ist auch die BYTE-Schreibweise erlaubt.

Beispiel:
```
   PROZEDUR fil;
   VAR Puffer: ARRAY[1..200] OF CHAR;
   BEGIN
     fillchar (Puffer,200,' ');   {in die Variable Puffer werden
   END;                           200 Leerzeichen uebertragen}
```

EXIT-Prozedur

Syntax:

exit

Diese Prozedur dient zum vorzeitigen Beenden einer Programmeinheit (Prozedur, Funktion oder des Hauptprogrammes).
EXIT in einem Hauptprogramm wirkt wie HALT.

```
Beispiel:
   PROCEDURE lesen;
   BEGIN
     assign(f,'DATEN.BAS');
     {$I-}
     reset(f);
     {$I+}
     IF ioresult <> 0 THEN BEGIN
                             writeln ('Filefehler!!!');
                             exit;
                           END;
    .
    .
   END;
```

HALT-Prozedur

Syntax:

halt

Die Prozedur HALT bewirkt den Abbruch der Programmausfuehrung und die Rueckkehr in das Laufzeitsystem.

HI-Funktion

Syntax:

hi (<Integer>)

Das niederwertige Byte des Ergebnisses enthaelt das hoeherwertige Byte des Wertes vom Integerausdruck <INTEGER>. Das hoeherwertige Byte des Ergebnisses ist Null. Das Ergebnis ist vom Typ Integer.

KEYPRESSED-Funktion

Syntax:

keypressed

Die Funktion gibt den Wert TRUE zurueck, wenn eine Taste auf der Konsole gedrueckt wurde. Das Ergebnis wird durch Aufruf der Konsol-Status-Routine des BIOS realisiert.

LO-Funktion

Syntax:

lo (<Integer>)

Die Funktion gibt das niederwertige Byte des Wertes vom Integerausdruck <Integer> zurueck, wobei das hoeherwertige Byte auf Null gesetzt wird. Der Typ des Ergebnisses ist Integer.

OVRDRIVE-Prozedur

Syntax:

ovrdrive (<Laufwerk>)

Laufwerk := 0¦1¦2¦3¦4
Dabei spezifiziert 0 das aktuelle Laufwerk, 1 das Laufwerk A, 2 das Laufwerk B, usw.
Die Prozedur legt das Laufwerk fest, in dem die OVERLAY-Files 000, 001, ... erwartet werden. Da diese Files bei dem jeweiligen Unterprogrammruf fuer OVERLAY-Programme benoetigt werden, muss der Anwender dafuer sorgen, dass zur rechten Zeit das richtige Laufwerk spezifiziert ist.

MOVE-Prozedur

Syntax:

move (<Quelle>,<Ziel>,<Anzahl>)

Diese Prozedur kopiert im Speicher eine bestimmte Anzahl von Bytes <Anzahl> von der Speicherstelle <Quelle> zur Speicherstelle <Ziel>. Hierbei sind <Quelle> und <Ziel> zwei Variablen von beliebigem Typ (auch Zeiger) und <Anzahl> ist ein Integerausdruck.

PARAMCOUNT-Funktion

Syntax:

paramcount

Diese Funktion ermittelt die Anzahl der Kommandozeilenparameter, d.h. die beim Start eines COM-Files durch Leerzeichen getrennt nach dem COM-File-Namen noch ausgegeben werden. Das Ergebnis ist vom Typ INTEGER.

Beispiel: Kommando
>TEST ARTIKEL.DAT 15.9.86 <ET>
paramcount liefert den Wert 2.

PARAMSTR-Funktion

Syntax:

paramstr (<Integer>)

Diese Funktion stellt den <Integer>-ten Kommandozeilenparameter bereit. Das Ergebnis ist von Typ STRING. Im Beispiel fuer PARAMCOUNT gilt:

```
paramstr(1) = 'ARTIKEL.DAT';
paramstr(2) = '15.9.86';
```

RANDOM-Funktion

Syntax:

random

Gibt eine Zufallszahl zurueck, die groesser oder gleich Null und kleiner als und Eins ist. Der Typ ist REAL.

RANDOM(I)-Funktion

Syntax:

random (<Integer>)

Gibt eine Zufallszahl zurueck, die groesser oder gleich Null und kleiner als <Integer> ist. <Integer> und die Zufallszahl sind beide vom Typ INTEGER.

RANDOMSIZE

Syntax:

randomsize

Der Zufallszahlengenerator wird in einen definierten Anfangszustand versetzt.

SIZEOF-Funktion

Syntax:

sizeof (<Variable>)

Die Funktion liefert als INTEGER-Wert die Laenge von <Variable> in Bytes.
Fuer <Variable> ist jeder Variablenbezeichner zugelassen.

Beispiel:

```
PROCEDURE Size;
VAR B: ARRAY[1..10] OF CHAR;
    A: ARRAY[1..15] OF CHAR;
BEGIN
  A:= 'ABCDEFGHIJKLMNO';
  B:= '0123456789';
  writeln (sizeof(A),'/',sizeof(B));         {Ausgabe: 15/10}
  move (B,A,sizeof(B));
  writeln (A);                    {Ausgabe: 0123456789KLMNO}
END.
```

SIZEOF laesst sich auch guenstig mit FILLCHAR und MOVE verbinden.

SWAP-Funktion

Syntax:

swap (<Integer>)

Die Funktion vertauscht vom Wert des Integerausdruckes <Integer> das hoeher- und niederwertige Byte und gibt das Ergebnis als Integerzahl aus. Beispiel swap($1234) gibt $3412 zurueck (Werte zur Verdeutlichung in hexadezimaler Schreibweise).

Beispiel:

```
swap($1234)           { = $3412}
```

UPCASE-Funktion

Syntax:

upcase (<Zeichen>)
Zeichen::= Konstante oder Variable vom Typ ['a'..'z']

Das Ergebnis ist der entsprechende Grossbuchstabe. Liegt <Zeichen> ausserhalb des Bereichs 'a'..'z', ist die Funktion wirkungslos.

Beispiel:

```
VAR T:STRING[20];
    i:INTEGER;
  .
  .
  .
READLN (T);
FOR i:=1 TO LENGTH(T) DO UPCASE(T[i]);
```

4.8. Operationen mit Mengen

Mengenwerte koennen aus anderen Mengenwerte durch Mengenausdruecke berechnet werden. Mengenausdruecke bestehen aus:

- Mengenkonstruktionen
- Mengenoperatoren
- Mengenkonstanten und
- Mengenvariablen

Mengenoperationen wurden in Ziffer 4.4.1.5. dargestellt.

4.8.1. Mengenkonstruktionen

Eine Mengenkonstruktion besteht aus einer oder mehreren Elementenspezifikationen, die durch Komma voneinander getrennt und in eckige Klammern eingeschlossen sind. Eine Elementenspezifikation ist ein Ausdruck vom gleichen Typ wie der Basistyp der Menge. Sie kann auch ein Bereich sein, der durch zwei solcher Ausdruecke dargestellt wird, getrennt durch zwei aufeinanderfolgende Punkte.

Beispiele:

```
['T','U','R','B','O']
['X','Y']
[X..Y]
[1..5]
['A'..'Z','a'..'z','0'..'9']
[1,3..10,12]
[]
```

Das letzte Beispiel stellt die leere Menge dar. Da sie keinen Ausdruck enthaelt, der ihren Basistyp festlegt, ist sie mit allen Mengentypen kompatibel. Die Menge [1..5] ist der Menge [1,2,3,4,5] aaequivalent. Wenn X>Y, dann bezeichnet [X..Y] eine leere Menge.

4.8.2 Mengenzuweisungen

Mengenvariablen wird das Ergebnis von Mengenausdruecken durch das Ergibtzeichen ':=' zugewiesen.

Beispiele:

```
    .
TYPE  Attribut = (braun,grau,karo,beige,rot);
VAR   Farbe:SET OF Attribut;
BEGIN
  Farbe:= [braun];
    .
    .
```

4.9. Zeiger und Listen

Zeiger ermoeglichen eine dynamische Speicherverwaltung

4.9.1. Dynamische Variablen

Dynamische Variablen werden waehrend der Programmausfuehrung geschaffen und wieder vernichtet.

Beispiel:
```
  TYPE  Zeiger   =  ^Struktur;            {^ = Zeigertyp}
                                  {Struktur noch nicht def.}
        Struktur = RECORD
          Nummer   : STRING[6];
          Name     : STRING[40];
          Menge    : INTEGER;
          Naechster: Zeiger
        END;
  VAR  Artikel : Zeiger;
  BEGIN
     .

     {Speicheradresse durch den Programmierer}
     Artikel :=  ptr($8000);
     {Speicheradresse durch das System}
     new(Artikel);
```

Die obige Vereinbarung reserviert zunaechst nur 2 Byte Speicherplatz fuer "Artikel" als Zeiger auf eine Struktur. Die Zeigervariable unterscheidet sich wesentlich von anderen Variablen. Sie enthaelt eine Speicheradresse einer einfachen oder strukturierten Variablen. Der Speicherplatz fuer einfache oder strukturierte Variablen (dynamische Variablen) wird geschaffen, wenn der Zeigervariablen eine (freie) Adresse zugewiesen wird. Das kann direkt oder mit NEW geschehen. Der Zugriff zum Inhalt der Adresse erfolgt mit ^.

4.9.2. Erzeugung und Vernichtung dynamischer Variablen

Mit der Prozedur NEW ist es moeglich, Speicherplatz fuer Variablen vom definierten Typ zu reservieren.

Syntax:

new (<Zeiger>)

Beispiel:
```
  new(Artikel);
```

Artikel zeigt im Beispiel auf einen dynamisch erzeugten Satz vom Typ Struktur. Auf diese dynamische Variable wird wie folgt zugegriffen:

```
  readln(Artikel^.Nummer);
  readln(Artikel^.Name);
```

Der erneute Aufruf von NEW fuehrt zu neuer Speicherplatzreservierung.

Die Gesamtheit des belegten Speicherbereiches, der nicht zusammenhaengend sein muss, wird als Halde(HEAP) bezeichnet.
Der Zeigerwert NIL gehoert jedem Zeigertyp an. Er zeigt auf keine dynamische Variable und wird Zeigervariablen zugewiesen, um anzuzeigen, dass sie keine verwertbare Adresse enthalten.

Die Freigabe des Speicherplatzes von geloeschten Elementen einer Liste erfolgt mit der Prozedur **DISPOSE**.

Syntax:

dispose (<Zeiger>)

Hiermit wird bewirkt, dass der Speicherplatz, auf welchen der <Zeiger> zeigt, fuer weitere Belegungen verwendet werden kann.

Zur Verwaltung dynamischer Variablen werden noch folgende Funktionen bereitgestellt:

memavail
maxavail

Die Funktion MEMAVAIL liefert den fuer dynamische Variablen verfuegbaren Speicherraum als INTEGER-Wert (Anzahl der Bytes).
Die Funktion MAXAVAIL liefert den fuer dynamische Variablen verfuegbaren Speicherraum (ausschliesslich geloeschte dynamische Variablen) als INTEGER-Wert.

4.9.3. Mark und Release

Es gibt statt DISPOSE eine weitere Moeglichkeit zur Freigabe des Speicherplatzes dynamischer Variablen. Das sind die Standardprozeduren MARK und RELEASE.

Syntax:

mark (<Zeiger>)

Mit MARK kann auf einer Zeigervariablen der aktuelle Stand der Halde festgehalten werden, mit dem Ziel der spaeteren Freigabe ab dieser Position.

Syntax:

release (<Zeiger>)

Mit RELEASE kann eine Halde ab der Position freigegeben werden, die vorher mit MARK fixiert wurde. Dabei darf die Pointervariable zwischen den Rufen MARK und RELEASE nicht veraendert werden. Mit RELEASE wird der Zustand wiederhergestellt, der zum Zeitpunkt des vorhergehenden Prozedurrufes MARK existierte.
In einem Programm muss zwischen der Freigabemethode DISPOSE (nach Wirth) und MARK/RELAESE (nach Bowles) gewaehlt werden. Sie sind unvertraeglich.

4.9.4. GETMEM und FREEMEM

Es gibt noch eine weitere Methode der dynamischen Verwaltung von Speicherplatz . NEW repraesentiert stets den Speicherplatz, der fuer die Struktur erforderlich ist, auf den der in der Prozedur angegebene Zeiger zeigt. Das kann hinderlich sein. Deshalb gibt es die Moeglichkeit, die Groesse des dynamisch reservierten Speicherplatzes selbst zu bestimmen.

Syntax:

getmem (<Zeiger>,<Anzahl>)
<Anzahl>::= Ausdruck des Typs INTEGER

<Anzahl> gibt den Speicherplatz in Bytes an. Entsprechend kann der Speicherplatz wieder freigegeben werden.

Syntax:

freemem (<Zeiger>,<Anzahl>)

Unter Nutzung von MAXAVAIL/MEMAVAIL ermoeglicht das eine dem Problem und der Speicherkapazitaet angepasste dynamische Datenverwaltung.

4.9.5. Programmierung dynamischer Listen

(1) Deklaration

```
TYPE Zeiger    = ^Objekt;
     Objekt = RECORD
                Wert: REAL;
                Naechst: Zeiger
              END;
VAR Z,Anf,P,Q: Zeiger;
        Wert1: REAL;
        Wert2: REAL;
```

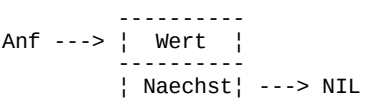
(2) Initialisieren einer dynamischen Liste

```
Anf:= NIL
                          Anf:= NIL       {= leere Liste}
```

Eine Liste hat immer einen Zeiger (z.B. Anf), welcher auf den Anfang der Liste zeigt.

(3) Eintragen eines Listenelements an den Anfang einer Liste:

```
readln (Wert1);
new (Z);
Z^.Wert:= Wert1;
Z^.Naechst:= Anf;               {oder Z^.Naechst:= NIL}
Anf:= Z;
```

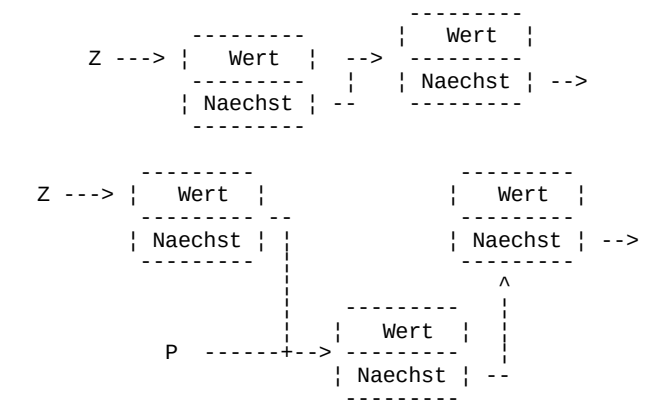


(4) Suchen eines Elements (Wert1) in einer Liste

```
PROCEDURE Such;
VAR Gefunden:Boolean;
BEGIN
 REPEAT
   readln (Wert2);
   Z:= Anf;
   WHILE Z <> NIL DO BEGIN;
     Gefunden:=Z^.Wert = Wert2;
     IF Gefunden THEN Z:= NIL
     ELSE BEGIN
       Q:= Z;
       Z:= Z^.Naechst
     END;
   END;
   IF NOT Gefunden THEN writeln ('Nicht vorhanden!');
 UNTIL Gefunden;
 writeln ('Wert gefunden');
END;
```

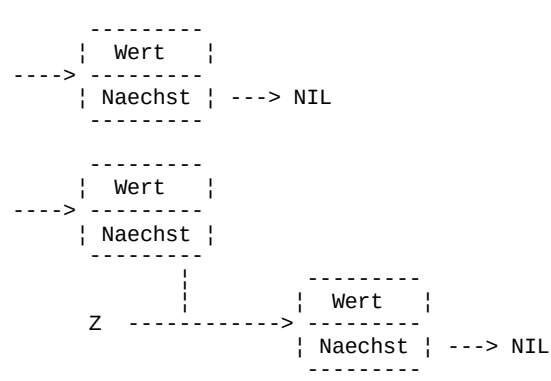
(5) Einfuegen eines neuen Elements in der Mitte der Liste

```
new (P);
readln (P^.Wert);                {neues Element eingeben}
Such;                    {Suchen eines Wertes (Wert2),  }
                         {nach welchem eingefuegt werden}
                         {soll.}
P^.Naechst:= Z^.Naechst;
Z^.Naechst:= P;
```



(6) Anfuegen eines neuen Elements am Ende der Liste

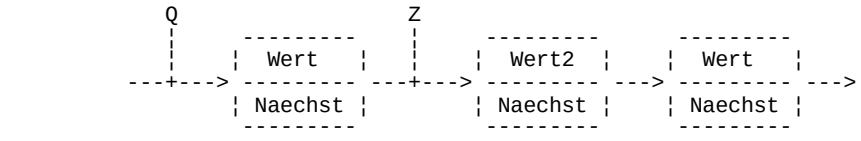
```
new (Z);
readln (Z^.Wert);
P:= Anf;                              {neues Element eingeben}
WHILE P <> NIL DO
      BEGIN Q:= P=;
            P:= Q^Naechst;
      END;
Z^.Naechst:= NIL;
Q^.Naechst:= Z;
```

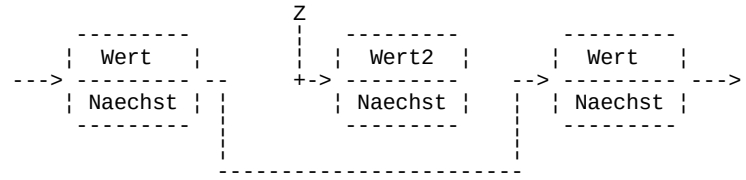


(7) Loeschen eines Elements einer Liste:

```
Such;                          {Eingabe und Suchen des zu}
                               {loeschenden Elements}

Q^.Naechst:= Z^.Naechst;
dispose (Z);
```





4.10. Ein- und Ausgabe von Files

4.10.1. Begriffe

Ein **File** ist ein Datenbestand, welcher aus logisch gegliederten, gleichgrossen Filekomponenten besteht.
Der Zugriff erfolgt ueber einen Zeiger wahlweise direkt oder sequentiell.
PASCAL unterscheidet zwischen Geraete- und Diskettenfiles.
Diskettenfiles werden unter dem vom Nutzer vereinbarten Filenamen (Prozedur ASSIGN) auf Diskette abgelegt.
Der Name eines Diskettenfiles wird durch eine Zeichenkette dargestellt. Fuer <Filename> unter SCPX gilt:

```
<Filename>::=<Basisname>{.<Erweiterung>}
<Basisname>::=
     <Zeichen>{<Zeichen>}
<Erweiterung>::=
     <Zeichen>{<Zeichen>}
<Zeichen>::= Element der Menge ['#'..'&','/','(',')','0'..'9',
             '@'..'Z','a'..'z']
```

Fuer den Basisnamen sind maximal 8, fuer die Erweiterung maximal 3 Zeichen zugelassen.

Es koennen gleichzeitig 16 Files bearbeitet werden.

Durch den **Typ eines Files** (vergl. Ziffer 4.3.3.3.) wird die Groesse und das Format der Filekomponenten spezifiziert.

Folgender Standardtyp ist vordefiniert:

```
 TEXT = FILE OF CHAR.
```

Fuer jedes File wird waehrend der Laufzeit ein **File-Informationsblock** angelegt (vergl. Ziffer F.1.5.). Dieser ist dem Programmierer nicht direkt zugaenglich.

Beispiel fuer die Spezifikation einer Filevariablen:

```
  PROGRAM xyz (Dateix);
  TYPE Dtz: RECORD a: INTEGER;
                   b: REAL;
            END;
  VAR Dateix: FILE OF Dtx;
```

Durch diese Deklaration wird eine interne Filevariable Datei x erzeugt, die genau ein Element (Satz) des externen Diskettenfiles aufnehmen kann (1*INTEGER, 1*REAL).

In PASCAL stehen dem Nutzer folgende Zugriffsroutinen fuer ein File zur Verfuegung:

a) Sequentieller und wahlfreier Zugriff zu Binaerfiles
   - READ
   - WRITE

b) Sequentieller Zugriff zu Textfiles
   - READLN
   - WRITELN

c) Blockweiser Zugriff zu Binaer- und Textfiles
   - BLOCKREAD
   - BLOCKWRITE

Vor einem wahlfreien Zugriff muss das Filefenster mit der Prozedur SEEK positioniert werden.
Ausserdem stehen weitere Funktionen/Prozeduren zur Verfuegung, die die Filearbeit unterstuetzen.
Das Ende einer Textdatei wird durch das Steuerzeichen $1A gekennzeichnet.
Wird dieses Zeichen gelesen, dann liefert die Standardfunktion EOF = TRUE. Binaerfiles enthalten in den ersten 4 Byte Informationen ueber Umfang und Anzahl der Filekomponenten.

4.10.2. Fileoperationen fuer Binaerfiles

ASSIGN

Syntax:

assign(<Filevariable>,<Filename>)

Die Prozedur hat die Aufgabe, der Variablen <Filevariable> einen externen Filenamen zuzuweisen. Die Filevariable kann jeden beliebigen Typ annehmen.
Sollen SCP-Geraetenamen zugewiesen werden, dann muss die <Filevariable> vom Typ TEXT sein.
Ein erneutes Anwenden von ASSIGN auf eine Filevariable, welcher bereits ein physischer Filenamen zugeordnet wurde und mit welcher bereits gearbeitet wurde, ist unzulaessig.

INPUT, OUTPUT, LST, KBD, CON und TRM sind vordefinierte Textfiles, so dass die letzte Spezifikation im Beispiel nur in Ausnahmefaellen erforderlich ist (vergl. Ziffer 4.10.6.).

REWRITE

Syntax:

rewrite (<Filevariable>)

Die REWRITE-Prozedur erzeugt auf Diskette ein File mit dem Namen, welcher mit ASSIGN zugewiesen wurde.
Gleichzeitig wird die Datei fuer Schreiben freigegeben.
Der Filepointer wird dabei auf die Filekomponente mit der Nummer 0 gesetzt. Im Fall, dass auf der Diskette bereits der gleiche physische Name existiert, wird das dazugehoerige File ueberschrieben.
Zu Beginn enthaelt eine mit REWRITE erzeugte File kein Element. Die Funktion EOF ist TRUE.

Beispiel:

```
    PROGRAM Ausgabe;
        .
        .
        .
    BEGIN
     assign (Kunde,'A:Kunden.DAT');
     rewrite (Kunde);
           .
           .
```

RESET

Syntax:

reset (<Filevariable>)

Die RESET-Prozedur eroeffnet ein existierendes File. Bei direktem Zugriff ist die Datei offen zum Lesen und Schreiben. Der Filepointer wird auf die erste Filekomponente (mit der Nummer Null) gesetzt.

Beispiel:

```
  PROGRAM Ausg (Kunde);
  BEGIN
    assign(Kunde, 'A: Kunden.DAT');
    reset (Kunde);
     .
     .
     .
  END.
```

READ

Syntax:

read (<Filevariable>,<Variable>)

Diese Prozedur realisiert das Lesen einer Filekomponente.

Beispiel:

```
  read(Kunde,Name);
```

Es wird, da die Zugriffsanzahl zu Diskettenfiles minimiert wird, nicht bei jedem READ auch wirklich von der Diskette gelesen. Anders ist das, wenn es sich um die Tastatur handelt.

WRITE

Syntax:

write (<Filevariable>,<Variable>)

Diese Prozedur realisiert die Ausgabe der Inhalte der Variablen in die Filekomponenten des Files, welchem <Filevariable> zugeordnet wurde.

Beispiel:
```
   write(Kunde,Name);
```

SEEK

Syntax:

  seek (<Filevariable>,<Nummer>)

  Nummer::= Ausdruck

Das Filefenster des aktuellen Files wird durch diese Prozedur auf die Komponente mit der Nummer <Nummer> 1 eingestellt (die 1. Komponente hat die Nummer 0). Soll das File erweitert werden, so ist es moeglich, die letzte Filekomponente einzustellen (vergl. FILESIZE) und danach WRITE zu benutzen.
Auf diese Weise kann sehr einfach zum direkten Zugriff uebergegangen werden.

Beispiel:
```
   seek (Kunde,20);
```

FLUSH

Syntax:

 flush (<Filevariable>)

Diese Prozedur wird in Multi-User-Systemen benoetigt, in denen mehrere Nutzer zum gleichen Diskettenfile zugreifen. Dabei schreibt die Prozedur flush sofort den Update-Puffer auf die Diskette zurueck und sichert damit, dass die folgende Leseoperation als ein physisches Lesen ausgefuehrt wird. Flush darf niemals auf ein geschlossenes File angewendet werden.

CLOSE

Syntax:

   close (<Filevariable>)

Mit dieser Prozedur wird der Disketten-FCB aktualisiert, indem die entsprechenden Bytes des Speicher-FCB kopiert werden. Das File, welchem die <Filevariablen> zugeordnet wurden, wird geschlossen und der aktuelle Zustand in das Diskettenverzeichnis geschrieben.
Wird die Prozedur close nicht aufgerufen, tritt ein Datenverlust ein.

Beispiel:
```
  assign (Kunde,'Kunde.dat');
  rewrite(Kunde);
      .
      .
      .

  write(Kunde,Name);
  close(Kunde);
```

ERASE

Syntax:

erase (<Filevariable>)

Die Prozedur ERASE loescht das File, welchem <Filevariable> zugeordnet wurde, im Diskettenverzeichnis. Sollte das File mit RESET oder REWRITE bereits eroeffnet sein, muss es vor ERASE mit CLOSE geschlossen werden.

```
Beispiel:
  VAR x: FILE;
  BEGIN
    assign (x,'Beispiel.Dat');
    erase  (x);
```

RENAME

Syntax:

rename (<Filevariable>,<Filename>)

Die Prozedur RENAME wird genutzt, um das <Filevariable> zugeordnete File umzubenennen. Der neue Name wird in das Diskettenverzeichnis eingetragen, und die weiteren Operationen von <Filevariable> werden dann mit diesem File unter dem neuen Namen ausgefuehrt. Nach der Eroeffnung des Files ist das Umbenennen nicht mehr erlaubt.
Es ist zu sichern, dass der neue Filename auf der Diskette nicht bereits exsistiert, um ein Entstehen doppelter Namen in der Directory zu vermeiden. Das kann geprueft werden, wenn die Ein- und Ausgabeueberwachung des Systems mit {$I-} angeschaltet und mit dem neuen Namen eine Eroeffnung durch RESET versucht wird. Ist danach IORESULT gleich Null, so existiert das File bereits.

```
Beispiel:
  VAR x:FILE;
  BEGIN
    assign(x,'Alt.Dat');
    rename(x,'Neu.Dat');
    reset(x);
```

4.10.3. Filefunktionen

EOF

Syntax:

eof (<Filevariable>)

Im Fall, dass der Filepointer das Fileende erreicht hat, liefert die Funktion EOF den Wert TRUE. Andernfalls ist der zurueckgegebene Wert FALSE.

FILEPOS

Syntax:

filepos (<Filevariable>)

Mit dieser Funktion wird der Wert der aktuellen Position des Filefensters als ein INTEGER-Wert zurueckgegeben. Dabei besitzt die erste Komponente den Wert Null.

FILESIZE

Syntax:

filesize (<Filevariable>)

Mit dieser Funktion wird die Groesse des <Filevariable> zugeordneten Files zurueckgegeben. Es wird die Anzahl der Komponenten des Files bestimmt. Ergibt die Funktion den Wert Null, so ist das File leer.

4.10.4. Nutzung von Binaerfiles

Bevor ein File benutzt werden kann, muss der Filevariablen ein physisches File zugeordnet werden. Dies geschieht mit ASSIGN. Vor einer E/A-Operation muss das File durch Aufruf von REWRITE oder RESET eroeffnet werden. Danach zeigt das Filefenster auf die erste Komponente des Files. Nach REWRITE ist stets FILESIZE (filevariable) gleich Null. Eine Diskettenfile kann nur durch Anfuegen von weiteren Komponenten hinter die letzte existierende Komponente des Files erweitert werden. Es ist moeglich, das Filefenster an das Fileende zu positionieren .

Beispiel:
```
   seek(<Filevariable>,filesize<Filevariable>);
```

Nach Beendigung aller Input/Output-Operationen muss das eroeffnete File durch CLOSE geschlossen werden, damit der Disketten-FCB aktualisiert wird.

4.10.5. Textfiles und Textfileoperationen

Textfiles sind Files aus Elementen des gueltigen Zeichensatzes ($0 bis $7f). Nicht alle Bytes repraesentieren dabei druckbare Zeichen. Die Komponenten eines Textfiles sind Zeilen verschiedener Laenge, die durch Steuerzeichen CR/LF getrennt werden.
Eine Textfilevariable wird erklaert, indem man ihr den Standardtypbezeichner TEXT zuweist:

VAR <Filvariable> : **TEXT;**

Zeichenweise E/A-Operationen werden fuer Textfiles mit den Standardprozeduren READ und WRITE ausgefuehrt. Zeilen werden mit den speziellen Textfileoperationen READLN, WRITELN und EOLN behandelt. Es gilt:

readln(<Filevariable>) Springt zum Beginn der naechsten Textzeile,d.h. ueberspringt alle Zeichen bis und einschliesslich der naechsten CR/LF-Folge.

writeln(<Filevariable>) Schreibt die Zeilenendemarke, d.h. die CR/LF-Folge auf das Textfile.

eoln(<Filevariable>) Ist eine Boolesche-Funktion, die den Wert TRUE zurueckgibt, wenn das Ende der aktuellen Zeile erreicht ist, d.h. wenn der Filepointer auf das CR-Zeichen der CR/LF-Folge zeigt. Ist EOF (Filevariable) gleich TRUE, so ist EOLN (Filevariable) auch TRUE.

Wendet man die EOF-Funktion auf ein Textfile an, dann liefert diese Funktion den Wert TRUE, wenn der Filepointer die Fileendemarke CTRL-Z erreicht hat.
Auf Textfiles sind die Funktionen SEEK, FLUSH, FILEPOS und FILESIZE nicht anwendbar, da keine gleichgrossen Filekomponenten existieren.

Beispiele:
```
  VAR Lst:TEXT;
  BEGIN
    assign(Lst,'LST:');
    rewrite(Lst);
    write(Lst,...);
```

Diese Bedeutung von LST ist vordefiniert, so dass auf die Vereinbarung der Filevariablen mit TEXT, ASSIGN und REWRITE verzichtet werden kann (vergl. Ziffer 4.10.7.).

4.10.6. Logische Geraete

Logische Geraete sind in PASCAL externe Geraete wie Terminals, Drucker und Modems. Sie werden wie Textfiles behandelt.

CON: Console. Ausgaben werden an ein Bildschirmgeraet gesendet und Eingaben werden von der Tastatur gelesen. READ oder READLN ueber CON lesen eine ganze Zeile aus dem Zeilenpuffer. Der Operator kann, bis zur Eingabe von CR ueber die ET-Taste, die Editiermoeglichkeiten des Systems fuer Eingaben nutzen.

TRM: Terminal. Ausgaben werden an ein Bildschirmgeraet gesendet und Eingaben werden von der Tastatur gelesen. Eingegebene Zeichen, ausser Controlzeichen, werden als Echo an das Consolausgabegeraet gesendet. Das einzige Controlzeichen, das als Echo gesendet wird, ist das Zeichen CR und zwar in Form der Folge CR/LF.

KBD: Keyboard. Eingaben werden von der Tastatur gelesen. Ein Echo erfolgt nicht.

LST: Listing. Die Ausgaben erfolgen an einen Drucker.
AUX: Auxiliary. Ausgaben werden an den Stanzer gesendet und Eingaben werden vom Leser gelesen. Normalerweise sind beide Lochband- oder Kassetenmagnetbandgeraete.

USR: Usergeraet. Ausgaben gehen an das Nutzerausgabegeraet, und Eingaben werden ueber die Nutzereingaberoutine gelesen.

Es ist moeglich, dass diese logischen Geraete durch vorher definierte Files oder wie ein Diskettenfile einer Filevariablen zugewiesen werden. Bei Zuweisung eines logischen Geraetes zu einem File existiert zwischen REWRITE und RESET kein Unterschied. Die Prozedur CLOSE fuehrt dann keine Funktion aus und ERASE liefert einen E/A-Fehler.

Die Standardfunktionen EOF und EOLN arbeiten bei logischen Geraeten anders als bei Diskettenfiles. Bei einem Diskettenfile liefert EOF den Wert TRUE zurueck, wenn das naechste Zeichen im File das Zeichen CTRL-Z ist. EOLN gibt den Wert TRUE zurueck, wenn das naechste Zeichen CR oder CTRL-Z ist.
Diese beiden Prozeduren sind vorausschauende Routinen.
Wird SEEKEOF oder SEEKEOLN statt EOF/EOLN verwendet, so werden Leerzeichen und Tabulatormarken (und bei SEEKEOF auch CR/LF) uebersprungen.
Bei logischen Geraeten gibt es jedoch keine Moeglichkeit vorauszuschauen, welche Zeichen als naechste kommen werden. Aus diesem Grunde liefern EOF und EOLN bei logischen Geraeten das Ergebnis immer vom letzten behandelten Zeichen und nicht vom naechsten. EOF liefert TRUE, wenn das letzte Zeichen CTRL-Z war, und EOLN liefert TRUE, wenn das letzte Zeichen CR oder CTRL-Z war.

	Diskettenfiles	Logische Geraete
EOLN ist TRUE	wenn aktuelles Zeichen CR und naechstes LF ist oder wenn naechstes Zeichen CRTL-Z ist	wenn aktuelles Zeichen CR oder CTRL-Z ist.
EOF ist TRUE	wenn naechstes Zeichen CTRL-Z ist.	wenn aktuelles Zeichen CTRL-Z ist.

4.10.7. Standardfiles

PASCAL 880/S stellt einige Standardtextfiles zur Verfuegung, die bereits logischen Geraeten zugewiesen sind und unmittelbar genutzt werden koennen. So ist es moeglich, Speicherplatz und den Aufruf von ASSIGN, RESET, REWRITE und CLOSE zu sparen.
Folgende Standardtextfiles sind implementiert:

INPUT	Primaeres Eingabefile. Dieses File ist entweder dem CON- oder TRM-Geraet zugewiesen.
OUTPUT	Primaeres Ausgabefile. Dieses File ist entweder dem CON- oder TRM-Geraet zugewiesen.
CON	Zugewiesen dem Consolgeraet CON:.
TRM	Zugewiesen dem Terminalgeraet TRM:.
KBD	Zugewiesen dem Keybord KBD:.
LST	Zugewiesen dem Listgeraet LST:.
AUX	Zugewiesen dem Auxiliarygeraet AUX:.

USR Zugewiesen dem Usergeraet USR:.

Die Verwendung von RESET, REWRITE und CLOSE ist verboten. Die Zuweisung des logischen Geraetes zu den Standardtextfiles INPUT und OUTPUT erfolgt durch die Compilerdirektive $B.

```
{$B+}  weist  CON: zu,
{$B-}  weist  TRM: zu.
```

Bei Zuweisung von CON: werden die Eingaben gepuffert und koennen in diesem Puffer bei der Eingabe editiert werden. Fuer das Einlesen der Variablen gelten spezielle Regeln. Bei Zuweisung von TRM: ist ein Editieren der Eingaben nicht moeglich. Das Einlesen der Variablen erfolgt aber nach den bekannten Regeln. Bei den Ausgabeoperationen existieren fuer CON: und TRM: keine Unterschiede.
Die Compilerdirektive $B muss vor dem Programmblock stehen und darf als globale Direktive im Programmblock nicht geaendert werden. Wenn in einem Programm sowohl CON- als auch TRM-Geraete verwendet werden, ist die Direktive $B entsprechend dem am haeufigsten verwendeten Geraet zu setzen, und in den anderen E/A-Operationen ist das andere Geraet explizit anzugeben.

Beispiel:

```
  {$B-}
  PROGRAM Lesen/Schreiben(OUTPUT);
  ...
  ...
  ...
  readln(INPUT,Var1);           Lesen von TRM:
  readln(CON,Var2);             Lesen von CON:
```

An den Stellen, wo auf dem Bildschirm kein Echo der Eingabe erscheinen soll, muss man das Standardtextfile KBD zuweisen:

```
read(KBD,Ant);
```

Da die Standardtextfiles INPUT und OUTPUT sehr haeufig verwendet werden, wurde implementiert, dass sie automatisch zugewiesen werden, wenn kein Filebezeichner explizit angegeben wurde. Damit sind die folgenden Textfileoperationen aequivalent:

```
write(x)       write(OUTPUT,x)
read(x)        read(INPUT,x)
writeln        writeln(OUTPUT)
readln         readln(INPUT)
eof            eof(INPUT)
```

Das folgende Programm zeigt die Verwendung des Standardfiles LST:

Beispiel:

```
 .
 .
writeln(LST,'Ausgabe ueber Drucker');
 .
 .
```

4.10.8. Ein- und Ausgabe von Textfiles

Die Ein- und Ausgabe von Daten durch den Menschen in lesbarer Form wird mittels Textfiles, wie in Punkt 4.10.7 beschrieben, ausgefuehrt. Ein Textfile kann einem Diskfile oder einem Standard-E/A-Geraet zugewiesen werden. Die Ein- und Ausgaben werden ausgefuehrt mit den Standardprozeduren READ, READLN, WRITE und WRITELN.
Die Parameter koennen im einzelnen unterschiedliche Typen haben. In diesen Faellen erfolgt eine automatische Datenkonvertierung bei der Ein- und Ausgabe in oder aus den Standart-CHAR Typen des Textfiles.
Ist der erste Parameter einer E/A-Prozedur ein Variablenbezeichner eines Textfiles, dann bezieht sich die Ein- oder Ausgabe auf diese File. Im anderen Fall bezieht sie sich auf das Standardtextfile INPUT oder OUTPUT.

4.10.8.1. READ

Die READ-Prozedur ermoeglicht die Eingabe von Zeichen, Strings und numerischen Daten.

Syntax:

read (<Variable>,{,<Variable>)}
read (<Filevariable>,< Variable>{,<Variable>)}

wobei die <Variable> vom Typ CHAR, STRING,INTEGER oder REAL sein koennen. Die erste Form liest Daten vom Standardfile INPUT. Die zweite Form liest Eingaben vom Textfile <File>, das fuer das Lesen vorbereitet werden muss oder vordefiniert ist. Mit einer Variablen vom Typ CHAR liest READ vom File ein Zeichen und weist dieses der Variablen zu. Im Fall, dass das File ein Diskettenfile ist, wird EOLN TRUE, wenn das naechste Zeichen CR oder CTRL-Z ist. EOF wird TRUE, wenn das naechste Zeichen CTRL-Z ist.
Mit einer Variablen vom Typ STRING liest READ so viele Zeichen wie durch die maximale Laenge des STRING erlaubt sind, es sei denn, EOLN oder EOF wurde vorher erreicht oder der Puffer mit buflen<n> auf n-Zeichen begrenzt.

Mit einer numerischen Variablen (INTEGER oder REAL) erwartet READ eine Zeichenkette, die mit dem Format einer numerischen Konstante des entsprechenden Typs uebereinstimmt. Voranstehende Leerzeichen, HT, CR oder LF werden uebersprungen. Die Zeichenkette darf nicht laenger als 30 Zeichen sein und muss mit einem Leerzeichen, HT, CR oder CTRL-Z beendet sein. Im Fall, dass die Zeichenkette nicht mit dem Format uebereinstimmt, tritt ein E/A-Fehler auf. Im anderen Fall wird die numerische Zeichenkette in den entsprechenden Typ konvertiert und der Variablen zugewiesen. Im Fall, dass von einem Diskfile gelesen wurde und die Eingabezeichenkette mit einem Leerzeichen oder HT endet, dann startet die naechste READ- oder READLN-Operation mit dem Zeichen, das unmittelbar diesem Leerzeichen oder HT folgt. Fuer beide, Diskettenfile oder logischem Geraet, wird EOLN=TRUE, wenn die Zeichenkette mit CR oder CTRL-Z endete. EOF wird TRUE, wenn die Zeichenkette mit CTRL-Z endete. Ein Spezialfall der numerischen Eingabe tritt auf, wenn EOLN oder EOF bereits beim Beginn TRUE wird. In diesem Fall wird der Variablen kein neuer Wert zugewiesen. Die Variable behaelt ihren alten Wert.

Wenn das Eingabefile CON: zugewiesen wurde, oder wenn das Standardfile im {$B+}-Modus verwendet wurde, gelten spezielle Regeln fuer das Lesen der Variablen. Beim Aufruf von READ oder READLN wird die ganze Zeile von der Console in einen Puffer gebracht, und das Einlesen der Variablen erfolgt aus diesem Puffer als Eingabequelle. Dies ermoeglicht das Editieren waehrend der Eingabe. Es bewirken:

Backspace und DEL	Ruecksetzen des Cursors und Loeschen des dort stehenden Zeichens. Backspace wird durch die Taste <-- oder CTRL-H, DEL durch die Taste DEL erzeugt.
CTRL-X	Ruecksetzen des Cursors auf den Eingabebeginn und Loeschen aller eingegebenen Zeichen.

Die ET-Taste beendet die Eingabe; das dabei eingegebene CR wird nicht als Echo auf dem Bildschirm ausgegeben.
Intern wird die Eingabezeile mit einem CTRL-Z am Ende gespeichert. Ist diese Eingabezeile kuerzer als die Variablen in der Parameterliste, werden die restlichen Variablen wie folgt behandelt: bei CHAR wird CTRL-Z eingetragen, bei STRING wird mit Leerzeichen aufgefuellt, und numerische Variablen bleiben unveraendert.

Maximal koennen in eine Eingabezeile 127 Zeichen eingegeben werden. Man kann die Eingabezeile, wie bereits beschrieben, begrenzen. Dazu wird der vordeklarierten Variablen BUFLEN eine INTEGER-Zahl aus dem Bereich 0 bis 127 zugewiesen.

Beispiel:
```
  write('Filename (max. 10 Zeichen):');
  buflen := 10;
  readln(Filename);
```

Es ist zu beachten, dass die Zuweisungen zu BUFLEN nur fuer das unmittelbar darauffolgende READ wirken. Danach wird BUFLEN sofort wieder auf 127 gesetzt.

4.10.8.2. READLN

Der Unterschied zwischen READLN und READ besteht darin, dass nach dem Einlesen der letzten Variablen bei READLN der Rest der Zeile uebersprungen wird.

Syntax:

```
  readln  (<Variable>{,<Variable>})
  readln  (<Filevariableable>,{,<Variable>})
```

Nach einem READLN liest das naechste READ oder READLN vom Beginn der naechsten Zeile. EOLN ist immer FALSE nach READLN, ausser wenn EOF = TRUE ist. Es ist auch moeglich, READLN ohne Parameter aufzurufen.
In diesen Faellen wird die gesamte Zeile uebersprungen. Im Fall, dass READLN von der Console liest, wird im Gegensatz zu READ das beendende CR als Echo in der Form CR/LF-Folge auf den Bildschirm uebertragen.

4.10.8.3. WRITE

Mit WRITE ist die Ausgabe von Zeichen, Strings, booleschen und numerischen Werten moeglich.

Syntax:

write (<Parameter>,{,<Parameter>})
write (<Filevariable>,<Parameter>{,<Parameter>})

Die Parameter sind Variablen vom Typ CHAR, STRING, BOOLEAN, INTEGER oder REAL. Wahlweise folgt diesen Parametern jeweils ein Doppelpunkt und ein INTEGER-Ausdruck, der die Laenge des Ausgabefeldes angibt. In der ersten der oben angegebenen Formen erfolgt die Ausgabe der Variablen durch das Standardfile OUTPUT. Im zweiten Fall werden die Variablen durch die Textfile ausgegeben.

Die Formate der WRITE-Parameter haengen vom Typ der Variablen ab. Im folgenden werden die unterschiedlichen Formate und ihre Eigenschaften beschrieben. Dabei bezeichnen die Symbole:

I,m,n	Ausdruecke vom Typ	INTEGER
R	Ausdruecke vom Typ	REAL
Ch	Ausdruecke vom Typ	CHAR
S	Ausdruecke vom Typ	STRING
B	Ausdruecke vom Typ	BOOLEAN

Formatuebersicht

Ch — Ausgabe des Zeichens Ch.

Ch:n — Ausgabe des Zeichens Ch rechtsbuendig in einem n Zeichen langen Feld, d.h. vor Ch stehen n-1 Leerzeichen.

S — Ausgabe des STRING S. Felder (ARRAYs) koennen ebenfalls ausgegeben werden, wenn sie mit den STRINGs ueberein-stimmen und vom Typ CHAR sind.

S:n — Ausgabe der STRINGs rechtsbuendig in einem n Zeichen langen Feld, d.h. vor S stehen n-length(S) Leerzeichen.

B — Ausgabe des Wortes TRUE oder FALSE.

B:n — Ausgabe des Wortes TRUE oder FALSE rechtsbuendig in einem n Zeichen langen Feld.

I — Ausgabe der Dezimaldarstellung von I.

I:n — Ausgabe der Dezimaldarstellung von I rechtsbuendig in einem n Zeichen langen Feld.

R — Ausgabe der Dezimaldarstellung von R rechtsbuendig in einem 18 Zeichen langen Feld als Gleitkommazahl in der Form:

R >= 0 __x.xxxxxxxxxEtxx
R < 0 _-x.xxxxxxxxxEtxx

Dabei bedeuten die Zeichen _ Leerzeichen, x Ziffern und t entweder + oder -.

R:n Ausgabe der Dezimaldarstellung von R rechtsbuendig in einem n Zeichen langen Feld als Gleitkommazahl in der Form:

```
R >= 0        blanks x. Zahl Etxx
R <  0        blanks-x. Zahl Etxx
```

Dabei bedeuten blanks keine oder mehrere Leerzeichen, Zahl ein bis zehn Ziffern, x eine Ziffer und t entweder + oder -. Nach dem Dezimalpunkt wird mindestens eine Ziffer ausgegeben, d.h. n muss mindestens 7 sein. Ist n groesser als 16, so stehen vor der Zahl Leerzeichen.

R:n:m Ausgabe der Dezimaldarstellung von R rechtsbuendig in einem n Zeichen langen Feld als Festpunktzahl mit m Dezimalziffern. Dabei muss m im Bereich 0 <= m <= 24 liegen, sonst wird Gleitkommaformat verwendet. Das Feld wird vor der Zahl mit Leerzeichen aufgefuellt.

4.10.8.4. WRITELN

Der Unterschied zwischen WRITE und WRITELN besteht darin, dass bei WRITELN nach der letzten Variablen eine CR/LF-Folge ausgegeben wird.

Syntax:

writeln (<Parameter>,{,<Parameter>})
writeln (<Filevariable>,<Parameter>{,<Parameter>})

WRITELN oder WRITELN(Filevariable) bewirkt nur die Ausgabe einer CR/LF-Folge.

4.10.9. Nichtgetypte Files

Nichtgetypte Files sind Kanalein- und -ausgaben auf niedrigstem Niveau. Es werden Saetze zu 128 Bytes verarbeitet.
Eine nichtgetypte Filevariable benoetigt weniger Speicherplatz als eine andere Filevariable, da die Daten bei E/A-Operationen direkt zwischen dem Diskettenfile und der Variablen uebertragen werden, ohne Platz fuer einen Sektorpuffer zu benoetigen.

Beispiel:

```
VAR  Kunde: FILE;
```

Alle Standardfileprozeduren, also auch SEEK, ausser READ, WRITE und FLUSH, sind erlaubt. BLOCKREAD und BLOCKWRITE sind zwei spezielle schnelle Uebertragungsprozeduren, die anstelle von READ und WRITE genutzt werden .

Syntax:

blockread (<Filevariable>,<Variable>,<n>)
blockwrite (<Filevariable>,<Variable>,<n>)

<Filevariable> entspricht dabei dem Variablenbezeichner eines ungetypten Files, <Variable> einer beliebigen Variable und n einem INTEGER-Ausdruck. N gibt die Anzahl der zu uebertragenden 128-Byte-Saetze zwischen Diskettenfile und Speicher an. <Variable> muss dafuer ausreichen. BLOCKREAD und BLOCKWRITE realisieren zusaetzlich die Weiterfuehrung des Filefensters um die entsprechende Anzahl von Saetzen.

Beispiel:

```
   PROGRAM Kopieren;
   VAR Quellfile,Zielfile : FILE;
       Filename           : STRING[12];
       Laufwerk           : CHAR;
       Puffer             : ARRAY[1...1024] OF BYTE;

       BEGIN
         writeln ('Kopieren eines Files');
         write ('Filename: ');readln(Filename);
         assign (Quellfile,Filename);
         reset (Quellfile);
         write ('Ziellaufwerk: ');
         read (KBD,Laufwerk);
         assign (Zielfile,CONCAT(Laufwerk,':',Filename));
         rewrite (Zielfile);
         WHILE NOT eof (Quellfile) DO BEGIN
           blockread (Quellfile,Puffer,8);
           blockwrite (Zielfile,Puffer,8)
         END;
         close(Quellfile);
         close(Zielfile);
       END.
```

4.10.10. Ein- und Ausgabepruefung

E/A-Pruefungen waehrend der Laufzeit eines Programmes sind durch die I-Compilerdirektiven moeglich.
Ist $I+ gesetzt, so werden Fehler in E/A-Operationen durch das Laufzeitsystem SCPX auf die uebliche Weise behandelt. Ist $I- gesetzt, so sind die E/A-Operationen durch den Programmierer zu ueberwachen und Fehler entsprechend zu behandeln. Dazu dient die vordefinierte Funktion IORESULT. Sie liefert nach der E/A-Operation einen Fehlercode vom Typ INTEGER zurueck. Null ist fehlerfrei.

Beispiel:

```
  PROGRAM bsp;
  VAR  Kunde     : FILE;
       Dateiname : STRING[14];
       Test      : BOOLEAN;
  BEGIN
    REPEAT
      write('Eingabe des Namens der Datei:');
      readln(Dateiname);
      assign(Kunde,Dateiname);
      {$I-} reset(Kunde); {$I+}
      Test:= (ioresult = 0);
      IF NOT Test THEN writeln('Datei',Dateiname,'kann  nicht
                                eroeffnet werden!');
    UNTIL Test;
    close(Kunde);
  END.
```

Bei folgenden Standardfunktionen kann es zweckmaessig sein, mit IORESULT die Fehlerbehandlung selbst zu uebernehmen.

```
     BLOCKREAD BLOCKWRITE     CHAIN     CLOSE
     ERASE     EXECUTE        FLUSH     RENAME
     RESET     REWRITE        SEEK
```

4.11. Sonstige Sprachelemente und Besonderheiten

4.11.1. HEAP- und STACK-Manipulationen

Fuer die Speicherung dynamischer Variablen benoetigt man den HEAP(die Halde). Diese wird durch die Standard Prozeduren NEW, DISPOSE, MARK und RELEASE, GETMEM und FREEMEM gesteuert. Zum Programmstart wird HeapPtr auf den Anfang des freien Speicherbereiches, also auf das erste Byte nach dem Objektcode gestellt.
Waehrend der Aufloesung von Ausdruecken und fuer die Uebergabe von Parametern in Prozeduren und Funktionen wird fuer die Speicherung der Zwischenergebnisse der CPU-Stack benoetigt. Auch aktive FOR-Statements benoetigen den Stack und verwenden ein Wort. Der CPU-Stack wird zum Programmbeginn auf das Ende des freien Speicherbereiches gestellt.
Der Rekursions-Stack wird nur fuer rekursive Prozeduren und Funktionen benoetigt, d.h. Prozeduren und Funktionen, die mit der passiven A-Compiler-Direktive {$A-} uebersetzt werden. Bei Eintritt in eine rekursive Prozedur oder Funktion kopiert das Unterprogramm seinen Arbeitsbereich in den Rekursionsstack, und nach Rueckkehr aus der Prozedur wird der gesamte Arbeitsbereich in den urspruenglichen Status zurueckgespeichert. Der Standardanfangswert von RecurPtr zu Programmbeginn liegt 1K($400) Byte unter dem CPU-Stackpointer.
Die vordeklarierten Variablen

HEAPPTR:	Heap-Pointer
RECURPTR:	Rekursions-Stackpointer und
STACKPTR:	CPU-Stackpointer

erlauben dem Programmierer die Position des Heap und des Stacks zu steuern. Der Typ dieser Variablen ist INTEGER.
Man sollte beachten, dass HEAPPTR und RECURPTR im gleichen Kontext wie jede andere Variable verwendet werden koennen, waehrend der STACKPTR nur in Ergibtanweisungen und Ausdruecken verwendet werden darf.
Werden diese Variablen manipuliert, so ist darauf zu achten, dass sie auch im freien Speicher liegen und

HEAPPTR < RECURPTR < STACKPTR

ist. Beachtet man dies nicht, koennen Fehler entstehen. Veraenderungen des Heap und des Stacks duerfen niemals durchgefuehrt werden, wenn sie gerade verwendet werden. Das System prueft bei jedem Aufruf der Prozedur NEW und jedem Eintritt in eine rekursive Prozedur oder Funktion, ob es zu einer Kollision zwischen Heap und Rekursions-Stack kommt, d.h., es wird geprueft, ob HEAPPTR kleiner als RECURPTR ist.
Stimmt das nicht, fand eine Kollision statt, und es wird ein Laufzeitfehler angezeigt.
Zu beachten ist aber, dass niemals geprueft wird, ob der CPU-Stack den Rekursions-Stack ueberschreibt. Dies kann auftreten, wenn ein rekursives Unterprogramm sich selbst etwa 300-400mal aufruft. Wird dies jedoch in einem Programm erwartet, muss das folgende Statement zu Beginn des Programmblockes ausgefuehrt werden. Damit wird der Rekursions-Stackpointer tiefer gelegt und ein groesserer CPU-Stack aufgebaut:

```
RECURPTR:= STACKPTR - 2 * MaxTiefe - 512;
```

MaxTiefe ist die maximale Tiefe der Call-Rufe des rekursiven Unterprogrammes.

Zusaetzlich werden etwa 512 Bytes benoetigt fuer Uebertragung der Parameter und Zwischenergebnisse waehrend der Aufloesung der Ausdrucke.

4.11.2. Optimierung der ARRAY-Indizes

Die X-Compiler-Direktive erlaubt dem Programmierer festzulegen, ob der Aufbau der Array-Indizes bezueglich der Ausfuehrungsgeschwindigkeit oder der Speicherplatzzuordnung optimiert werden soll. Der Standard-Modus ist aktiv {$X+}, das bedeutet Optimierung bezueglich Ausfuehrungsgeschwindigkeit. Bei inaktivem Modus {$X-} wird der Code minimiert.

4.11.3. BDOS/BIOS-Rufe

Fuer Zwecke des Aufrufs von SCPX-Routinen benutzt PASCAL zwei Standardprozeduren: BDOS und BIOS und vier Standardfunktionen: BDOS, BDOSHL, BIOS, BIOSHL.

BDOS-Prozedur und -Funktion

Syntax:

BDOS(<n>)
BDOS(<n>,<p>)
BDOSHL(<n>)

Der Aufruf ist als Funktion oder Prozedur zur Nutzung des BDOS von SCPX moeglich.
n ist die Funktionsnummer, die in das Register C geladen wird, p der Aufrufparameter, der in das Registerpaar DE geladen wird. Die Rueckgabewerte (meist Register A) werden durch Aufruf als Funktion verfuegbar im Register A. Ist die Rueckgabe im Register HL vorgesehen, hat der Aufruf mit BDOSHL zu erfolgen.

BIOS-Prozedur und -Funktion

Syntax:

BIOS(<n>)
BIOS(<n>,<p>)
BIOSHL(<n>)

Der Aufruf ist als Prozedur oder Funktion zur direkten Nutzung der Driverroutinen des Laufzeitsystems SCPX moeglich.
Die Erlaeuterungen zu BDOS gelten sinngemaess.

Die Aufrufparameter fuer BDOS und BIOS enthaelt Angang G.

4.11.4. INLINE-Maschinencode

Das PASCAL-Programmiersystem stellt mit den INLINE-Anweisungen einen sehr brauchbaren Weg zum direkten Einfuegen von Maschinencode in den PASCAL-Programmtext zur Verfuegung.
Eine INLINE-Anweisung besteht aus dem reservierten Wort INLINE, dem eine oder mehrere Konstanten, Variablenbezeichner oder Bezuege zum Speicherplatzzaehler, getrennt durch Schraegstriche

und eingeschlossen in Klammern, folgen.

Syntax:

  Inline-Anweisung ::=

    INLINE (<InlineElement> {/<InlineElement>})

  InlineElement ::=

      <Operationscode>
    ¦ <vzl.g.Zahl>
    ¦ < <vzl.g.Zahl>
    ¦ > <vzl.g.Zahl>
    ¦ <Bezeichner> <Vorzeichen> * <vzl.g.Zahl>
    ¦ <Bezeichner> <Vorzeichen> * <Vorzeichen><vzl.g.Zahl>

  Vorzeichen ::= + ¦ -
  Operationscode::= INLINE-Operationscode lt. Anhang C.

Konstanten als Bezeichner muessen vom Typ INTEGER oder Konstantenbezeichner sein.
Konstanten erzeugen ein Code-Byte, wenn sie im Bereich 0...255 ($00...$FF) sind, andernfalls zwei Byte in dem standardisierten umgekehrten Byte-Format (d.h., das niederwertige steht vor dem hoeherwertigen Byte). Ist ein Zwei-Byte-Wert gefordert, kann ein Ein-Byte-Wert mit > angepasst werden. Ein Zwei-Byte-Wert wird durch < auf einen Ein-Byte-Wert reduziert.
Konstantenbezeichner definieren immer zwei Code-Bytes. Ein Variablenbezeichner erzeugt zwei Code-Bytes (im umgekehrten Byte-Format), die die Speicheradressen der Variablen enthalten. Ein Bezug zum Speicherplatz besteht aus einem Stern, dem ein Offset folgt, das aus einem Plus- oder Minuszeichen mit einer Integerkonstanten besteht. Ein Stern allein erzeugt zwei Code-Bytes (im umgekehrten Byte-Format), die die aktuelle Adresse des Speicherplatzzaehlers enthalten. Wenn dem Stern ein Offset folgt, dann wird dieser addiert oder subtrahiert, bevor die Adresse codiert wird.

Beispiel:
Das folgende Beispiel von INLINE-Anweisungen erzeugt einen Maschinencode, der alle Zeichen in seinem STRING-Argument in Grossbuchstaben umwandelt.

```
PROCEDURE Grossschreibung(Var Strg:Str);{Str ist vom TYPE STRING[255]}
begin
  inline($2A/Strg/          {    LD   HL,(Strg) }
         $46/               {    LD   B,(HL)    }
         $04/               {    INC  B         }
         $05/               {L1: DEC  B         }
         $CA/*+20/          {    JP   Z,L2      }
         $23                {    INC  HL        }
         $7E                {    LD   A,(HL)    }
         $FE/$61/           {    CP   'a'       }
         $DA/*-9/           {    JP   C,L1      }
         $FE/$7B/           {    CP   'z'+1     }
         $D2/*-14/          {    JP   NC,L1     }
         $D6/$20/           {    SUB  20H       }
         $77                {    LD   (HL),A    }
         $C3/*-20/;         {    JP   L1        }
END;                        {L2: EQU  $         }
```

Die Jump-Anweisung wird in diesem Beispiel nur zur Demonstration der Verwendung des Speicherplatzzaehlers benutzt.
INLINE-Anweisungen koennen in einem Teil eines Blockes mit anderen Anweisungen gemischt werden. Sie koennen alle Register der CPU verwenden.
Zu beachten ist jedoch, dass der Inhalt des Stackpointers(SP) beim Ausgang der gleiche wie beim Eintritt sein muss.

4.11.5. Nutzergeschriebene I/O-Driver

Fuer einige Anwendungen ist es fuer den Programmierer praktisch, seine eigenen Ein- und Ausgabedriver zu schreiben, d.h. Routinen, die Ein- und Ausgabe von Zeichen zu und von externen Geraeten liefern. Die folgenden Driver sind Teile des Programmiersystems und werden von Standard-I/O-Drivern verwendet (obgleich sie selbst nicht als Standard-Prozeduren oder Funktionen aufgerufen werden duerfen).

```
FUNCTION   CONST : BOOLEAN;
FUNCTION   CONIN : CHAR;
PROCEDURE  CONOUT  (Ch:CHAR);
PROCEDURE  LSTOUT  (Ch:CHAR);
PROCEDURE  AUXOUT  (Ch:CHAR);
FUNCTION   AUXIN : CHAR;
PROCEDURE  USROUT  (Ch:CHAR);
FUNCTION   USRIN : CHAR;
```

Die CONST-Routine wird durch die Funktion KEYPRESSED aufgerufen. die CONIN- und CONOUT-Routinen werden durch die CON:, TRM:, und KBD-Geraete verwendet; die LSTOUT wird durch das Geraet LST: verwendet; die Routinen AUXOUT und AUXIN werden durch das Geraet AUX: verwendet und die Routinen USROUT und USRIN werden durch das Geraet USR: verwendet.

Standardmaessig verwenden diese Driver die entsprechenden Eintrittspunkte des BIOS, d.h.

CONST	verwendet	CONST
CONIN	verwendet	CONIN
CONOUT	verwendet	CONOUT
LSTOUT	verwendet	LST
AUXOUT	verwendet	PUNCH
AUXIN	verwendet	READER
USROUT	verwendet	CONOUT
USRIN	verwendet	CONIN

Diese Zuordnung kann jedoch vom Programmierer geaendert werden, indem er den folgenden Standardvariablen die Adresse eigener Driver-Prozeduren oder Driver-Funktionen zuweist:

Variable	enthaelt die Adresse der Funktion
CONSTPTR	ConSt Funktion
CONINPTR	ConIn Funktion
CONOUTPTR	ConOut Prozedur
LSTOUTPTR	LstOut Prozedur
AUXOUTPTR	AuxOut Prozedur
AUXINPTR	AuxIn Funktion
USROUTPTR	UsrOut Prozedur
USRINPTR	UsrIn Funktion

Eine vom Nutzer geschriebene Driver-Prozedur oder Driverfunktion muss mit den oben beschriebenen Definitionen uebereinstimmen, d.h. ein CONST-Driver muss eine BOOLEAN-Funktion, ein CONIN-Driver muss eine CHAR-Funktion sein usw.

4.11.6. INTERRUPT-Behandlung

Der durch den Compiler erzeugte Code ist voll interruptfaehig. Wenn notwendig, kann man die Interruptprozeduren in PASCAL schreiben. Eine derartige Prozedur sollte immer mit aktiver A-Compiler Direktive {$A+} uebersetzt werden. Sie darf keine Parameter haben und muss selbst sichern, dass alle Registerwerte erhalten bleiben. Dies wird erreicht durch Setzen von entsprechenden Inline—Push-Anweisungen am Anfang der Prozedur und entsprechenden Inline-Pop—Anweisungen am Ende der Prozedur. Die letzte Inline-Anweisung muss EI sein ($FB), um weitere Interrupts zuzulassen.
Es gibt folgende Regeln fuer die Registerbenutzung:

- Integer-Operationen verwenden nur AF, BC, DE, HL,
- andere Operationen koennen auch IX und IY verwenden
- REAL—Operationen verwenden die alternativen Register (AF', BC', DE', HL', IX', IY')

Eine Interruptprozedur sollte keine I/O—Operation enthalten, die Standardprozeduren und -funktionen verwendet, da sie nicht wieder eintrittsfaehig sind. Aus dem gleichen Grunde duerfen BDOS-Rufe und in einigen Faellen auch BIOS-Rufe (das haengt von der speziellen SCPX—Implementation ab) nicht vom Interrupt-Driver benutzt werden, da auch diese nicht wieder eintrittsfaehig sind.
Der Programmierer kann Interrupts in seinem Programm mittels DI und EI—Inline-Anweisung verbieten und erlauben.
Wenn mod0 (IM0) oder mod1 (IM1) Interrupts verwendet werden, muss der Programmierer fuer die Initialisierung der Restart-Adressen sorgen (man beachte, dass RST0 nicht verwendet werden darf, da SCPX die Adressen 0 bis 7 benutzt).

Wenn mod2 (IM2) Interrupts verwendet werden, muss der Programmierer eine Sprungtabelle generieren (ein Array von Integerzahlen an einer absoluten Adresse) und das Indexregister durch Inline—Statements zu Beginn des Programmes initialisieren (bei BC sollten die freien Adressen fuer die Interruptvektor-Sprungtabellen verwendet werden). Die exakten Adressen sind den entsprechenden Systemunterlagen zu entnehmen.

5. Hilfsprogramme

5.1. Retten eines editierten Aktivfiles nach Systemabsturz

Wird versucht, mit S im Grundmenue des Systemkerns ein editiertes Aktivfile auf die Diskette zu schreiben, ohne dass das Diskettensystem zurueckgesetzt wurde, stuerzt das System ab und das File ist verloren. In diesem Falle oder wenn die Frage nach der erforderlichen Sicherung eines Files irrtuemlich mit N(nein) beantwortet wurde, ist das Retten des Editorpuffers noch moeglich. Dazu dient das Hilfsprogramm

```
PASRETT,
```

das sich als COM-File auf der Systemdiskette befinden sollte. PASRETT ermittelt den Pufferanfang (kann durch Laden oder Nichtladen von PASCAL.TXT verschieden sein) und das Pufferende. Wird es nicht gefunden (z.B. nach Kaltstart oder einem Start von PASRETT), ohne dass vorher editiert wurde, erfolgt die Ausschrift

```
Aktivfile zerstoert.
```

und PASRETT beendet mit

```
Ende PASRETT
```

die Arbeit. Andernfalls versucht das Programm den Filenamen aus dem Hauptspeicher zu lesen. Ist der Name durch Systemeinwirkungen oder den Anwender fehlerhaft, erfolgt die Ausschrift

```
Filename zerstoert.
Neuer Name:
```

Vor dem Namen kann ein Laufwerk angegeben werden. Ist der Name fehlerfrei oder wurde neu eingegeben, setzt PASRETT das Diskettensystem zurueck und versucht auf die Diskette zu schreiben. Ein vorhandenes File gleichen Namens wird dabei ueberschrieben. Kann aus technischen Gruenden (Diskette voll, Laufwerk nicht bereit) nicht geschrieben werden, erfolgt die Ausschrift

```
Ausgabe auf Diskette nicht moeglich.
Anderes Laufwerk oder <ET> nach Diskettenwechsel.
```

Es ist die Diskette zu wechseln und <ET> zu geben oder ein anderer Laufwerksname mit oder ohne Doppelpunkt einzugeben.

PASRETT quittiert die erfolgreiche Arbeit mit

```
File <Laufwerk>: <Filename> mit <n> Byte gesichert
```

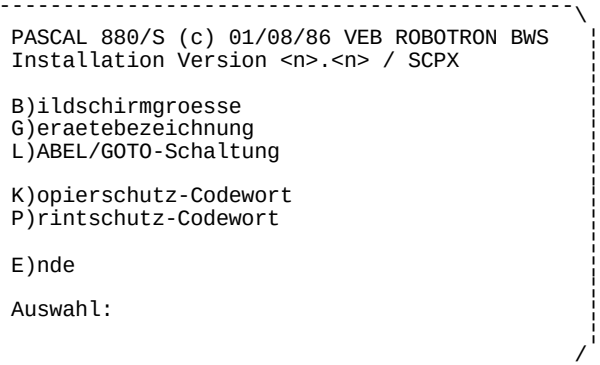
und beendet die Arbeit mit

```
Ende PASRETT.
```

5.2. Installation von Systemkern und Systemservice

Einige Parameter des Programmiersystems PASCAL 880/S lassen sich anwenderspezifisch installieren. Das entsprechende Programm heisst PASINST.COM.

Beim Start von PASINST.COM muessen sich PASCAL.COM und PLUS.COM ohne Schreibschutz auf der Diskette im aktuellen Laufwerk befinden. Die Moeglichkeiten gehen aus dem Eroeffnungsbild hervor.



Es kann immer nur eine Leistung gewaehlt werden. Eine gueltige Antwort wird erzwungen. Die Antworten werden im Langtext quittiert.
B, G, L aendern PASCAL.COM, C, P aendern PLUS.COM.

B Es wird zwischen 16 x 64 und 24 x 80 umgeschaltet. Der gegenwaertige Zustand wird mit

```
Aktueller Zustand: <zz> x <ss>
Aenderung J/N
```

angezeigt und die Bestaetigung der Aenderung verlangt. Bei J wird die erfolgreiche Aenderung mit

```
Neue Bildschirmgroesse <zz> x <ss>
```

quittiert.

G Hier kann eine Geraetebezeichnung mit maximal 20 Zeichen eingegeben werden. Wie bei B wird der aktuelle Zustand angezeigt. Danach erfolgt die Ausschrift

```
Neue Geraetebezeichnung:
```

Nur <ET> fuehrt zur Ablehnung der Funktion. Die erfolgreiche Veraenderung wird wieder angezeigt.

L Mit dieser Funktion kann fuer Ausbildungszwecke beeinflusst werden, ob der Compiler Sprungbefehle zulaesst oder nicht. Wie bei B wird der aktuelle Zustand angezeigt, eine Zustimmung oder Ablehnung zur Aenderung erzwungen und eine erfolgte Aenderung quittiert.

K Die Leistung "Kopieren" des Systemservice kann durch ein Codewort geschuetzt werden. Das Wort

CCCC

(Standard) hebt den Schutz auf.
PASINST antwortet nach C mit der Ausschrift

Aktuelles Codewort: <Codewort>

Ist bisher kein Codewort installiert, wird der Text "ohne" geschrieben. Es erfolgt die Aufforderung

Neues Codewort:

Das Wort ist maximal 8 Zeichen lang. Die Installation des neuen Wortes wird mit

Codewort <Codewort> eingetragen

quittiert.

P Die Leistung "Drucken" des Systemservice kann an ein Codewort gebunden werden. Das Wort

PPPP

(Standard) hebt den Schutz auf. Die Kommunikation ist wie bei C. Die maximale Laenge des Wortes ist ebenfalls 8 Zeichen.

Nach jeder Funktion wird zum Menue zurueckgekehrt. Die Eingabe des E beendet die Arbeit von PASINST. Das wird quittiert.
Die Funktion G steht ab Version 1.2, die Funktion L ab Version 1.3, und die Funktionen C und P ab Version 1.4 zur Verfuegung.

Anhang A: Zeichensatz

1. Steuerzeichen

DEZ	HEX	CTRL-Zeichen	Bezeichnung	Wirkungen (Auswahl) fuer Cursor oder Drucker fuer SCPX(versionsabhaengig)
0	00	^@	NUL	
1	01	^A	SOH	Cursor HOME(Position 1,1)
2	02	^B	STX	(+ $80) Cursor einschalten
3	03	^C	ETX	(+ $80) Cursor ausschalten
4	04	^D	EOT	(+ $80) Zeichendarstellung normal
5	05	^E	ENQ	(+ $80) Zeichendarstellung invers
6	06	^F	ACK	(+ $80) Zeichendarstellung intensiv
7	07	^G	BEL	akustisches u. optisches Signal
8	08	^H	BS	ein Zeichen zurueck
9	09	^I	HT	Tabulatorsprung
10	0A	^J	LF	Zeile nach unten
11	0B	^K	VT	
12	0C	^L	FF	Bildschirm loeschen und $01 oder Blattvorschub
13	0D	^M	CR	Anfang der laufenden Zeile
14	0E	^N	SO	
15	0F	^O	SI	
16	10	^P	DLE	
17	11	^Q	DC1	
18	12	^R	DC2	
19	13	^S	DC3	
20	14	^T	DC4	Loeschen des Bildschirms ab Cursorposition
21	15	^U	NAK	Zeichen nach rechts
22	16	^V	SYN	Loeschen der Zeile ab Cursorposition
23	17	^W	ETB	
24	18	^X	CAN	Loeschen Cursorzeile und $0D
25	19	^Y	EM	
26	1A	^Z	SUB	Zeile nach oben
27	1B	^[	ESC	Erstzeichen Cursorpositionierung
28	1C	^\	FS	
29	1D	^]	GS	
30	1E	^^	RS	
31	1F	^_	US	

2. Druckbare Zeichen

Ziffer	$0	$1	$2	$3	$4	$5	$6	$7	$8	$9	$A	$B	$C	$D	$E	$F
$2		!	"	#	$	%	&	´	(	)	*	+	,	~	.	/
$3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
$4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
$5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
$6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
$7	p	q	r	s	t	u	v	w	x	y	z	{	¦	}	-	DEL

Anhang B: Compilerdirektiven

Compilerdirektiven werden mit {$<Direktive>} an den Beginn einer Quelltextzeile geschrieben.

<Direktive> (Standard erstgenannt)	Wirkung
A+	Keine Rekursion (absoluter Code)
A-	Rekursion zugelassen
B+	Standardfile INPUT gleich CON
B-	Standardfile INPUT gleich TRM
C+	Eingabeinterpretation ^C Programmabbruch ^S Unterbrechung Bildschirmausgabe
C-	CTRL-Zeichen werden nicht interpretiert (beschleunigter Ablauf)
I+	E/A-Fehlerbehandlung durch das Laufzeit-/PASCAL-System
I-	E/A-Fehlerbehandlung ueber IORESULT durch den Programmierer
R-	Ohne Index- und Bereichsueberwachung waehrend der Laufzeit
R+	Index- und Bereichsueberwachung (langsamer Ablauf)
U-	Keine Programmunterbrechung durch den Benutzer waehrend der Laufzeit
U+	Unterbrechung waehrend der Laufzeit mit ^C moeglich(langsamerer Ablauf)
V+	Stringlaenge bei Parameteruebergabe wird geprueft
V-	Stringlaenge kann verschieden sein
Wn	Schachtelungstiefe von WITH-Anweisungen (n = 0..9; Standard 2)
X-	Zugriffsgeschwindigkeit normal
X+	Zugriff auf Felder beschleunigt (hoeherer Speicherplatzbedarf)
I <Name>	File <Name> wird an diese Stelle kopiert (<Name> = Include-Datei)

Anhang C: INLINE-Maschinencode

Uebergangstabelle von mnemonischem Assemblercode zu INLINE-Operationscode (Maschinencode).
Es bedeuten

- n 8 - Bit - Parameter
- nn 16 - Bit - Parameter
- d 8 - Bit - Distanz(mit Vorzeichen)

Die Erlaeuterung des mnemonischen Assemblercodes und die Wirkungen der Befehle auf die Flagregister sind der Literatur zu entnehmen (zum Beispiel Classen, L.;Oefler, U.:Wissensspeicher Mikrorechnerprogrammierung. Berlin: VEB Verlag Technik 1986, S.19ff)

Mnemonischer Code	Operations-code
ADC A,(HL)	$8E
ADC A,(IX+d)	$DD 8Ed
ADC A,(IY+d)	$FD 8Ed
ADC A,A	$8F
ADC A,B	$88
ADC A,C	$89
ADC A,D	$8A
ADC A,E	$8B
ADC A,H	$8C
ADC A,L	$8D
ADC A,n	$CE n
ADC HL,BC	$ED 4A
ADC HL,DE	$ED 5A
ADC HL,HL	$ED 6A
ADC HL,SP	$ED 7A
ADD A,(HL)	$86
ADD A,(IX+d)	$DD 86d
ADD A,(IY+d)	$FD 86d
ADD A,A	$87
ADD A,B	$80
ADD A,C	$81
ADD A,D	$82
ADD A,E	$83
ADD A,H	$84
ADD A,L	$85
ADD A,n	$C6 n
ADD HL,BC	$09
ADD HL,DE	$19
ADD HL,HL	$29
ADD HL,SP	$39
ADD IX,BC	$DD 09
ADD IX,DE	$DD 19
ADD IX,IX	$DD 29
ADD IX,SP	$DD 39
ADD IY,BC	$FD 09
ADD IY,DE	$FD 19
ADD IY,IY	$FD 29
ADD IY,SP	$FD 39
AND (HL)	$A6
AND (IX+d)	$DD A6d
AND (IY+d)	$FD A6d
AND A	$A7

Mnemonischer Code	Operations-code
AND B	$A0
AND C	$A1
AND D	$A2
AND E	$A3
AND H	$A4
AND L	$A5
AND n	$E6 n
BIT 0,(HL)	$CB 46
BIT 0,(IX+d)	$DD CBd46
BIT 0,(IY+d)	$FD CBd46
BIT 0,A	$CB 47
BIT 0,B	$CB 40
BIT 0,C	$CB 41
BIT 0,D	$CB 42
BIT 0,E	$CB 43
BIT 0,H	$CB 44
BIT 0,L	$CB 45
BIT 1,(HL)	$CB 4E
BIT 1,(IX+d)	$DD CBd4E
BIT 1,(IY+d)	$FD CBd4E
BIT 1,A	$CB 4F
BIT 1,B	$CB 48
BIT 1,C	$CB 49
BIT 1,D	$CB 4A
BIT 1,E	$CB 4B
BIT 1,H	$CB 4C
BIT 1,L	$CB 4D
BIT 2,(HL)	$CB 56
BIT 2,(IX+d)	$DD CBd56
BIT 2,(IY+d)	$FD CBd56
BIT 2,A	$CB 57
BIT 2,B	$CB 50
BIT 2,C	$CB 51
BIT 2,D	$CB 52
BIT 2,E	$CB 53
BIT 2,H	$CB 54
BIT 2,L	$CB 55
BIT 3,(HL)	$CB 5E
BIT 3,(IX+d)	$DD CBd5E
BIT 3,(IY+d)	$FD CBd5E
BIT 3,A	$CB 5F
BIT 3,B	$CB 58

Mnemonischer Code	Operations-code
BIT 3,C	$CB 59
BIT 3,D	$CB 5A
BIT 3,E	$CB 5B
BIT 3,H	$CB 5C
BIT 3,L	$CB 5D
BIT 4,(HL)	$CB 66
BIT 4,(IX+d)	$DD CBd66
BIT 4,(IY+d)	$FD CBd66
BIT 4,A	$CB 67
BIT 4,B	$CB 60
BIT 4,C	$CB 61
BIT 4,D	$CB 62
BIT 4,E	$CB 63
BIT 4,H	$CB 64
BIT 4,L	$CB 65
BIT 5,(HL)	$CB 6E
BIT 5,(IX+d)	$DD CBd6E
BIT 5,(IY+d)	$FD CBd6E
BIT 5,A	$CB 6F
BIT 5,B	$CB 68
BIT 5,C	$CB 69
BIT 5,D	$CB 6A
BIT 5,E	$CB 6B
BIT 5,H	$CB 6C
BIT 5,L	$CB 6D
BIT 6,(HL)	$CB 76
BIT 6,(IX+d)	$DD CBd76
BIT 6,(IY+d)	$FD CBd76
BIT 6,A	$CB 77
BIT 6,B	$CB 70
BIT 6,C	$CB 71
BIT 6,D	$CB 72
BIT 6,E	$CB 73
BIT 6,H	$CB 74
BIT 6,L	$CB 75
BIT 7,(HL)	$CB 7E
BIT 7,(IX+d)	$DD CBd7E
BIT 7,(IY+d)	$FD CBd7E
BIT 7,A	$CB 7F
BIT 7,B	$CB 78
BIT 7,C	$CB 79
BIT 7,D	$CB 7A
BIT 7,E	$CB 7B
BIT 7,H	$CB 7C
BIT 7,L	$CB 7D
CALL C,nn	$DC nn
CALL M,nn	$FC nn
CALL NC,nn	$D4 nn
CALL nn	$CD nn
CALL NZ,nn	$C4 nn
CALL P,nn	$F4 nn
CALL PE,nn	$EC nn

Mnemonischer Code	Operations-code
CALL PO,nn	$E4 nn
CALL Z,nn	$CC nn
CCF	$3F
CP (HL)	$BE
CP (IX+d)	$DD BEd
CP (IY+d)	$FD BEd
CP A	$BF
CP B	$B8
CP C	$B9
CP D	$BA
CP E	$BB
CP H	$BC
CP L	$BD
CP n	$FE n
CPD	$ED A9
CPDR	$ED B9
CPI	$ED A1
CPIR	$ED B1
CPL	$2F
DAA	$27
DEC (HL)	$35
DEC (IX+d)	$DD 35d
DEC (IY+d)	$FD 35d
DEC A	$3D
DEC B	$05
DEC BC	$0B
DEC C	$0D
DEC D	$15
DEC DE	$1B
DEC E	$1D
DEC H	$25
DEC HL	$2B
DEC IX	$DD 2B
DEC IY	$FD 2B
DEC L	$2D
DEC SP	$3B
DI	$F3
DJNZ d	$10 d
EI	$FB
EX (SP),HL	$E3
EX (SP),IX	$DD E3
EX (SP),IY	$FD E3
EX AF,AF	$08
EX DE,HL	$EB
EXX	$D9
HALT	$76
IM 0	$ED 46
IM 1	$ED 56
IM 2	$ED 5E
IN A,(C)	$ED 78
IN A,(n)	$DB n
IN B,(C)	$ED 40

Mnemonischer Code	Operations-code
IN C,(C)	$ED 48
IN D,(C)	$ED 50
IN E,(C)	$ED 58
IN H,(C)	$ED 60
IN L,(C)	$ED 68
INC (HL)	$34
INC (IX+d)	$DD 34d
INC (IY+d)	$FD 34d
INC A	$3C
INC B	$04
INC BC	$03
INC C	$0C
INC D	$14
INC DE	$13
INC E	$1C
INC H	$24
INC HL	$23
INC IX	$DD 23
INC IY	$FD 23
INC L	$2C
INC SP	$33
IND	$ED AA
INDR	$ED BA
INI	$ED A2
INIR	$ED B2
JP (HL)	$E9
JP (IX)	$DD E9
JP (IY)	$FD E9
JP C,nn	$DA nn
JP M,nn	$FA nn
JP NC,nn	$D2 nn
JP nn	$C3 nn
JP NZ,nn	$C2 nn
JP P,nn	$F2 nn
JP PE,nn	$EA nn
JP PO,nn	$E2 nn
JP Z,nn	$CA nn
JR C,d	$38 d
JR d	$18 d
JR NC,d	$30 d
JR NZ,d	$20 d
JR Z,d	$28 d
LD (BC),A	$02
LD (DE),A	$12
LD (HL),A	$77
LD (HL),B	$70
LD (HL),C	$71
LD (HL),D	$72
LD (HL),E	$73
LD (HL),H	$74
LD (HL),L	$75
LD (HL),n	$36 n

Mnemonischer Code	Operations-code
LD (IX+d),A	$DD 77d
LD (IX+d),B	$DD 70d
LD (IX+d),C	$DD 71d
LD (IX+d),D	$DD 72d
LD (IX+d),E	$DD 73d
LD (IX+d),H	$DD 74d
LD (IX+d),L	$DD 75d
LD (IX+d),n	$DD 36dn
LD (IY+d),A	$FD 77d
LD (IY+d),B	$FD 70d
LD (IY+d),C	$FD 71d
LD (IY+d),D	$FD 72d
LD (IY+d),E	$FD 73d
LD (IY+d),H	$FD 74d
LD (IY+d),L	$FD 75d
LD (IY+d),n	$FD 36dn
LD (nn),A	$32 nn
LD (nn),BC	$ED 43nn
LD (nn),DE	$ED 53nn
LD (nn),HL	$22 nn
LD (nn),IX	$DD 22nn
LD (nn),IY	$FD 22nn
LD (nn),SP	$ED 73nn
LD A,(BC)	$0A
LD A,(DE)	$1A
LD A,(HL)	$7E
LD A,(IX+d)	$DD 7Ed
LD A,(IY+d)	$FD 7Ed
LD A,(nn)	$3A nn
LD A,A	$7F
LD A,B	$78
LD A,C	$79
LD A,D	$7A
LD A,E	$7B
LD A,H	$7C
LD A,I	$ED 57
LD A,L	$7D
LD A,n	$3E n
LD A,R	$ED 5F
LD B,(HL)	$46
LD B,(IX+d)	$DD 46d
LD B,(IY+d)	$FD 46d
LD B,A	$47
LD B,B	$40
LD B,C	$41
LD B,D	$42
LD B,E	$43
LD B,H	$44
LD B,L	$45
LD B,n	$06 n
LD BC,(nn)	$ED 4Bnn
LD BC,nn	$01 nn

Mnemonischer Code	Operations-code
LD C,(HL)	$4E
LD C,(IX+d)	$DD 4Ed
LD C,(IY+d)	$FD 4Ed
LD C,A	$4F
LD C,B	$48
LD C,C	$49
LD C,D	$4A
LD C,E	$4B
LD C,H	$4C
LD C,L	$4D
LD C,n	$0E n
LD D,(HL)	$56
LD D,(IX+d)	$DD 56d
LD D,(IY+d)	$FD 56d
LD D,A	$57
LD D,B	$50
LD D,C	$51
LD D,D	$52
LD D,E	$53
LD D,H	$54
LD D,L	$55
LD D,n	$16 n
LD DE,(nn)	$ED 5Bnn
LD DE,nn	$11 nn
LD E,(HL)	$5E
LD E,(IX+d)	$DD 5Ed
LD E,(IY+d)	$FD 5Ed
LD E,A	$5F
LD E,B	$58
LD E,C	$59
LD E,D	$5A
LD E,E	$5B
LD E,H	$5C
LD E,L	$5D
LD E,n	$1E n
LD H,(HL)	$66
LD H,(IX+d)	$DD 66d
LD H,(IY+d)	$FD 66d
LD H,A	$67
LD H,B	$60
LD H,C	$61
LD H,D	$62
LD H,E	$63
LD H,H	$64
LD H,L	$65
LD H,n	$26 n
LD HL,(nn)	$2A nn
LD HL,nn	$21 nn
LD I,A	$ED 47
LD IX,(nn)	$DD 2Ann
LD IX,nn	$DD 21nn
LD IY,(nn)	$FD 2Ann

Mnemonischer Code	Operations-code
LD IY,nn	$FD 21nn
LD L,(HL)	$6E
LD L,(IX+d)	$DD 6Ed
LD L,(IY+d)	$FD 6Ed
LD L,A	$6F
LD L,B	$68
LD L,C	$69
LD L,D	$6A
LD L,E	$6B
LD L,H	$6C
LD L,L	$6D
LD L,n	$2E n
LD R,A	$ED 4F
LD SP,(nn)	$ED 7Bnn
LD SP,HL	$F9
LD SP,IX	$DD F9
LD SP,IY	$FD F9
LD SP,nn	$31 nn
LDD	$ED A8
LDDR	$ED B8
LDI	$ED A0
LDIR	$ED B0
NEG	$ED 44
NOP	$00
OR (HL)	$B6
OR (IX+d)	$DD B6d
OR (IY+d)	$FD B6d
OR A	$B7
OR B	$B0
OR C	$B1
OR D	$B2
OR E	$B3
OR H	$B4
OR L	$B5
OR n	$F6 n
OTDR	$ED BB
OTIR	$ED B3
OUT (C),A	$ED 79
OUT (C),B	$ED 41
OUT (C),C	$ED 49
OUT (C),D	$ED 51
OUT (C),E	$ED 59
OUT (C),H	$ED 61
OUT (C),L	$ED 69
OUT (n),A	$D3 n
OUTD	$ED AB
OUTI	$ED A3
POP AF	$F1
POP BC	$C1
POP DE	$D1
POP HL	$E1
POP IX	$DD E1

Mnemonischer Code	Operations-code
POP IY	$FD E1
PUSH AF	$F5
PUSH BC	$C5
PUSH DE	$D5
PUSH HL	$E5
PUSH IX	$DD E5
PUSH IY	$FD E5
RES 0,(HL)	$CB 86
RES 0,(IX+d)	$DD CBd86
RES 0,(IY+d)	$FD CBd86
RES 0,A	$CB 87
RES 0,B	$CB 80
RES 0,C	$CB 81
RES 0,D	$CB 82
RES 0,E	$CB 83
RES 0,H	$CB 84
RES 0,L	$CB 85
RES 1,(HL)	$CB 8E
RES 1,(IX+d)	$DD CBd8E
RES 1,(IY+d)	$FD CBd8E
RES 1,A	$CB 8F
RES 1,B	$CB 88
RES 1,C	$CB 89
RES 1,D	$CB 8A
RES 1,E	$CB 8B
RES 1,H	$CB 8C
RES 1,L	$CB 8D
RES 2,(HL)	$CB 96
RES 2,(IX+d)	$DD CBd96
RES 2,(IY+d)	$FD CBd96
RES 2,A	$CB 97
RES 2,B	$CB 90
RES 2,C	$CB 91
RES 2,D	$CB 92
RES 2,E	$CB 93
RES 2,H	$CB 94
RES 2,L	$CB 95
RES 3,(HL)	$CB 9E
RES 3,(IX+d)	$DD CBd9E
RES 3,(IY+d)	$FD CBd9E
RES 3,A	$CB 9F
RES 3,B	$CB 98
RES 3,C	$CB 99
RES 3,D	$CB 9A
RES 3,E	$CB 9B
RES 3,H	$CB 9C
RES 3,L	$CB 9D
RES 4,(HL)	$CB A6
RES 4,(IX+d)	$DD CBdA6
RES 4,(IY+d)	$FD CBdA6
RES 4,A	$CB A7
RES 4,B	$CB A0

Mnemonischer Code	Operations-code
RES 4,C	$CB A1
RES 4,D	$CB A2
RES 4,E	$CB A3
RES 4,H	$CB A4
RES 4,L	$CB A5
RES 5,(HL)	$CB AE
RES 5,(IX+d)	$DD CBdAE
RES 5,(IY+d)	$FD CBdAE
RES 5,A	$CB AF
RES 5,B	$CB A8
RES 5,C	$CB A9
RES 5,D	$CB AA
RES 5,E	$CB AB
RES 5,H	$CB AC
RES 5,L	$CB AD
RES 6,(HL)	$CB B6
RES 6,(IX+d)	$DD CBdB6
RES 6,(IY+d)	$FD CBdB6
RES 6,A	$CB B7
RES 6,B	$CB B0
RES 6,C	$CB B1
RES 6,D	$CB B2
RES 6,E	$CB B3
RES 6,H	$CB B4
RES 6,L	$CB B5
RES 7,(HL)	$CB BE
RES 7,(IX+d)	$DD CBdBE
RES 7,(IY+d)	$FD CBdBE
RES 7,A	$CB BF
RES 7,B	$CB B8
RES 7,C	$CB B9
RES 7,D	$CB BA
RES 7,E	$CB BB
RES 7,H	$CB BC
RES 7,L	$CB BD
RET	$C9
RET C	$D8
RET M	$F8
RET NC	$D0
RET NZ	$C0
RET P	$F0
RET PE	$E8
RET PO	$E0
RET Z	$C8
RETI	$ED 4D
RETN	$ED 45
RL (HL)	$CB 16
RL (IX+d)	$DD CBd16
RL (IY+d)	$FD CBd16
RL A	$CB 17
RL B	$CB 10
RL C	$CB 11

Mnemonischer Code	Operations-code
RL D	$CB 12
RL E	$CB 13
RL H	$CB 14
RL L	$CB 15
RLA	$17
RLC (HL)	$CB 06
RLC (IX+d)	$DD CBd06
RLC (IY+d)	$FD CBd06
RLC A	$CB 07
RLC B	$CB 00
RLC C	$CB 01
RLC D	$CB 02
RLC E	$CB 03
RLC H	$CB 04
RLC L	$CB 05
RLCA	$07
RLD	$ED 6F
RR (HL)	$CB 1E
RR (IX+d)	$DD CBd1E
RR (IY+d)	$FD CBd1E
RR A	$CB 1F
RR B	$CB 18
RR C	$CB 19
RR D	$CB 1A
RR E	$CB 1B
RR H	$CB 1C
RR L	$CB 1D
RRA	$1F
RRC (HL)	$CB 0E
RRC (IX+d)	$DD CBd0E
RRC (IY+d)	$FD CBd0E
RRC A	$CB 0F
RRC B	$CB 08
RRC C	$CB 09
RRC D	$CB 0A
RRC E	$CB 0B
RRC H	$CB 0C
RRC L	$CB 0D
RRCA	$0F
RRD	$ED 67
RST 0	$C7
RST 8	$CF
RST 10H	$D7
RST 18H	$DF
RST 20H	$E7
RST 28H	$EF
RST 30H	$F7
RST 38H	$FF
SBC A,(HL)	$9F
SBC A,(IX+d)	$DD 9Ed
SBC A,(IY+d)	$FD 9Ed
SBC A,A	$9F

Mnemonischer Code	Operations-code
SBC A,B	$98
SBC A,C	$99
SBC A,D	$9A
SBC A,E	$9B
SBC A,H	$9C
SBC A,L	$9D
SBC A,n	$DE n
SBC HL,BC	$ED 42
SBC HL,DE	$ED 52
SBC HL,HL	$ED 62
SBC HL,SP	$ED 72
SCF	$37
SET 0,(HL)	$CB C6
SET 0,(IX+d)	$DD CBdC6
SET 0,(IY+d)	$FD CBdC6
SET 0,A	$CB C7
SET 0,B	$CB C0
SET 0,C	$CB C1
SET 0,D	$CB C2
SET 0,E	$CB C3
SET 0,H	$CB C4
SET 0,L	$CB C5
SET 1,(HL)	$CB CE
SET 1,(IX+d)	$DD CBdCE
SET 1,(IY+d)	$FD CBdCE
SET 1,A	$CB CF
SET 1,B	$CB C8
SET 1,C	$CB C9
SET 1,D	$CB CA
SET 1,E	$CB CB
SET 1,H	$CB CC
SET 1,L	$CB CD
SET 2,(HL)	$CB D6
SET 2,(IX+d)	$DD CBdD6
SET 2,(IY+d)	$FD CBdD6
SET 2,A	$CB D7
SET 2,B	$CB D0
SET 2,C	$CB D1
SET 2,D	$CB D2
SET 2,E	$CB D3
SET 2,H	$CB D4
SET 2,L	$CB D5
SET 3,(HL)	$CB DE
SET 3,(IX+d)	$DD CBdDE
SET 3,(IY+d)	$FD CBdDE
SET 3,A	$CB DF
SET 3,B	$CB D8
SET 3,C	$CB D9
SET 3,D	$CB DA
SET 3,E	$CB DB
SET 3,H	$CB DC
SET 3,L	$CB DD

Mnemonischer Code	Operations-code
SET 4,(HL)	$CB E6
SET 4,(IX+d)	$DD CBdE6
SET 4,(IY+d)	$FD CBdE6
SET 4,A	$CB E7
SET 4,B	$CB E0
SET 4,C	$CB E1
SET 4,D	$CB E2
SET 4,E	$CB E3
SET 4,H	$CB E4
SET 4,L	$CB E5
SET 5,(HL)	$CB EE
SET 5,(IX+d)	$DD CBdEE
SET 5,(IY+d)	$FD CBdEE
SET 5,A	$CB EF
SET 5,B	$CB E8
SET 5,C	$CB E9
SET 5,D	$CB EA
SET 5,E	$CB EB
SET 5,H	$CB EC
SET 5,L	$CB ED
SET 6,(HL)	$CB F6
SET 6,(IX+d)	$DD CBdF6
SET 6,(IY+d)	$FD CBdF6
SET 6,A	$CB F7
SET 6,B	$CB F0
SET 6,C	$CB F1
SET 6,D	$CB F2
SET 6,E	$CB F3
SET 6,H	$CB F4
SET 6,L	$CB F5
SET 7,(HL)	$CB FE
SET 7,(IX+d)	$DD CBdFE
SET 7,(IY+d)	$FD CBdFE
SET 7,A	$CB FF
SET 7,B	$CB F8
SET 7,C	$CB F9
SET 7,D	$CB FA
SET 7,E	$CB FB
SET 7,H	$CB FC
SET 7,L	$CB FD
SLA (HL)	$CB 26
SLA (IX+d)	$DD CBd26
SLA (IY+d)	$FD CBd26
SLA A	$CB 27
SLA B	$CB 20
SLA C	$CB 21

Mnemonischer Code	Operations-code
SLA D	$CB 22
SLA E	$CB 23
SLA H	$CB 24
SLA L	$CB 25
SRA (HL)	$CB 2E
SRA (IX+d)	$DD CBd2E
SRA (IY+d)	$FD CBd2E
SRA A	$CB 2F
SRA B	$CB 28
SRA C	$CB 29
SRA D	$CB 2A
SRA E	$CB 2B
SRA H	$CB 2C
SRA L	$CB 2D
SRL (HL)	$CB 3E
SRL (IX+d)	$DD CBd3E
SRL (IY+d)	$FD CBd3E
SRL A	$CB 3F
SRL B	$CB 38
SRL C	$CB 39
SRL D	$CB 3A
SRL E	$CB 3B
SRL H	$CB 3C
SRL L	$CB 3D
SUB (HL)	$96
SUB (IX+d)	$DD 96d
SUB (IY+d)	$FD 96d
SUB A	$97
SUB B	$90
SUB C	$91
SUB D	$92
SUB E	$93
SUB H	$94
SUB L	$95
SUB n	$D6 n
XOR (HL)	$AE
XOR (IX+d)	$DD AEd
XOR (IY+d)	$FD AEd
XOR A	$AF
XOR B	$A8
XOR C	$A9
XOR D	$AA
XOR E	$AB
XOR H	$AC
XOR L	$AD
XOR n	$E n

Anhang D: Fehlermeldungen Compiler

Die folgende Liste enthaelt die Fehlermitteilungen des Compilers. Wenn beim Compilieren ein Fehler auftritt, wird immer die Fehlernummer angezeigt. Der Text wird nur ausgegeben, wenn der Fehlertext geladen ist (Anwort "J" auf die erste Frage beim Start des Systemkerns). Die meisten der Fehlermitteilungen erklaeren sich von selbst, aber bei einigen sind noch kurze Erlaeuterungen angefuegt.

01: ';' fehlt

02: ':' fehlt

03: ',' fehlt

04: '(' fehlt

05: ')' fehlt

06: '=' fehlt

07: ':=' fehlt

08: '[' fehlt

09: ']' fehlt

10: '.' fehlt

11: '..' fehlt

12: BEGIN fehlt

13: DO fehlt

14: END fehlt

15: OF fehlt

17: THEN fehlt

18: DO oder DOWNTO fehlt

20: Boolescher Ausdruck erwartet

21: File-Variable erwartet

22: INTEGER/BYTE-Konstante erwartet

23: INTEGER/BYTE-Ausdruck erwartet

24: INTEGER/BYTE-Variable erwartet

25: INTEGER/BYTE- oder REAL-Konstante erwartet

26: INTEGER/BYTE- oder REAL-Ausdruck erwartet

27: INTEGER/BYTE- oder REAL-Variable erwartet

28: Zeiger-Variable erwartet

29: Record-Variable erwartet

30: Einfacher Typ erwartet

31: Einfacher Ausdruck erwartet

32: STRING-Konstante erwartet

33: STRING-Ausdruck erwartet

34: STRING-Variable erwartet

35: Textfile erwartet

36: Typ-Bezeichner erwartet

37: Ungetyptes File erwartet

40: Undefinierte Marke

41: Undefinierter Bezeichner oder Syntaxfehler
(Unbekannter Marken-, Konstanten-, Typ-, Variablen- oder Feldbezeichner oder Syntaxfehler in der Anweisung)

42: Undefinierter Zeigertyp in vorhergehenden Typdefinitionen

43: Doppelter Bezeichner oder doppelte Marke

44: Typ unvertraeglich
1. Inkompatible Typen von Variablen und Ausdruecken in einer Ergibtanweisung.
2. Inkompatible Typen von aktuellen und formalen Parametern bei einem Unterprogrammaufruf.
3. Der Typ eines Ausdruckes ist inkompatibel mit dem Indextyp in einer ARRAY-Ergibtanweisung.
4. Die Typen der Operanden eines Ausdrucks sind inkompatibel.

45: Konstante ausserhalb des zulaessigen Bereiches

46: Konstante und CASE Selektortyp unvertraeglich

47: Operanden- und Operatortyp unvertraeglich

48: Ungueltiger Ergebnis-Typ

49: Unzulaessige STRING-Laenge
(Die Laenge eines STRING muss im Bereich 1..255 liegen.)

50: STRING-Konstantenlaenge unvertraeglich

51: Ungueltiger Teilbereichstyp
(Gueltige Basistypen sind alle Skalartypen, ausser REAL)

52: Untere > obere Grenze

53: Reserviertes Wort
(Das Wort kann nicht als Bezeichner verwendet werden.)

54: Unzulaessige Wertzuweisung

55: STRING-Konstante ueberschreitet Zeile

56: Fehler in einer INTEGER/BYTE-Konstante

57: Fehler in einer REAL-Konstante

58: Unzulaessiges Zeichen im Bezeichner

60: Konstanten sind hier nicht erlaubt

61: Files oder Zeiger sind hier nicht erlaubt

62: Strukturierte Variablen sind hier nicht erlaubt

63: Textfiles sind hier nicht erlaubt

64: Textfiles oder ungetypte Files sind hier nicht erlaubt

65: Ungetypte Files sind hier nicht erlaubt

66: Eingabe/Ausgabe ist hier nicht erlaubt

67: Files erfordern VAR-Parameter

68: Filekomponenten duerfen keine Files sein

69: Unzulaessige Ordnung von Feldern

70: SET-Basistyp ausserhalb des zulaessigen Bereiches
Der Basistyp einer Menge muss ein ordinaler Typ mit nicht mehr als 256 Werten oder ein Teilbereich mit Grenzen im Bereich 0..255 sein.

71: Ungueltiges GOTO
Eine GOTO-Anweisung ausserhalb einer FOR-Schleife kann sich nicht auf eine Marke in dieser FOR-Schleife beziehen. Auch gelten Marken nicht in eingeschlossenen Unterprogrammen.

72: Marke nicht im gleichen Block

73: Undefiniertes FORWARD-Unterprogramm
Ein Unterprogramm wurde FORWARD-deklariert, aber der zugehoerige Programmkoerper existiert nicht.

74: INLINE-Fehler

75: Unzulaessiger Gebrauch von ABSOLUTE
1. Vor dem Doppelpunkt darf nur ein Bezeichner bei der Definition einer absoluten Variablen stehen.
2. Das Wort ABSOLUTE darf nicht innerhalb eines Satzes verwendet werden.

76: OVERLAY und FORWARD unvertraeglich

77: Unzulaessiges OVERLAY im Indirektmodus

90: File nicht gefunden
Die spezifizierte INCLUDE-File existiert nicht.

91: Vorzeitiges Ende des Quell-Files

92: Anlegen des OVERLAY-Files unmoeglich

93: Ungueltige Compilerdirektive

96: Unzulaessiges Schachteln von INCLUDE-Files

97: Zuviele WITH-Schachtelungen
Verwende die W-Compiler-Direktive, um die maximale Zahl der geschachtelten WITH-Anweisungen zu erhoehen. Standard ist zwei.

98: Speicherueberlauf

99: Compilerueberlauf
Es ist nicht genuegend Speicherplatz vorhanden, um das Programm uebersetzen zu koennen. Sie muessen Ihr Programm in kleinere Segmente teilen und INCLUDE-Files verwenden.

Anhang E: Fehlermeldungen Laufzeitsystem

E.1. Allgemeine Laufzeitfehler

Fehler fuehren zur Laufzeit eines Programms zum Abbruch und zur Anzeige der Mitteilung:

Laufzeit- Fehler <nn>, PC = <Adresse>
Programmabbruch

wobei nn die Fehlernummer und <Adresse> die Adresse im Programmcode ist, an der der Fehler auftrat. Diese Stelle kann bei geladenem Quellcode mit F im O-Menue des Systemkerns gesucht werden.

nn	Bedeutung
01	Gleitkommaueberlauf
02	Division durch Null
03	Fehler im Argument von SQRT (< Null)
04	Fehler im Argument von LN (<= Null)
10	Falsche STRING-Laenge (auch in Ergibtanweisungen) (1. Eine STRING-Kettung ergibt einen String, der mehr als 256 Zeichen hat. 2. Nur STRING's der Laenge 1 koennen in ein Zeichen konvertiert werden.)
11	Fehlerhafter STRING-Index (ausserhalb 1 - 255)
90	Index ausserhalb des zulaessigen Bereiches
91	Ordinaler Typ ausserhalb des Wertebereiches (auch bei Teilbereichstypen)
92	Wert ausserhalb des INTEGER-Bereiches
FF	Halden/Kellerspeicher-Kollision Es wurde die Prozedur NEW oder ein rekursives Unterprogramm aufgerufen und es gibt zwischen Heap-Pointer und dem Rekursions-Stack-Pointer keinen freien Speicherplatz mehr.

E.2 Ein/Ausgabe Laufzeitfehler

Tritt waehrend der Laufzeit bei einer Ein- oder Ausgabeoperation ein Fehler auf und ist die Systemueberwachung aktiv (bei aktiver I-Compiler-Direktive), erfolgt ein Programmabbruch mit folgender Fehlermitteilung:

EA-Fehler <nn>, PC = <Adresse>

<nn> ist die EA-Fehlernummer und <Adresse> die Adresse im Programm, an der der Fehler auftrat.
Wenn die EA-Fehlerpruefung passiv ist {$I-}, dann erfolgt kein Programmabbruch. Man kann das Ergebnis der EA-Operation mittels der Funktion IORESULT abfragen und so entsprechende Massnahmen treffen. Ein Fehler kann bei geladenem Quelltext mit F im O - Menue des Systemkerns lokalisiert werden.

nn	Bedeutung
01	Dieses File existiert nicht
02	File fuer Leseoperationen nicht vorbereitet
03	File fuer Schreiboperationen nicht vorbereitet
04	File nicht geoeffnet
10	Fehler im numerischen Format
20	Operation auf logischem Geraet nicht erlaubt
21	Im Direktmodus (Zielauswahl H) nicht erlaubt
22	ASSIGN fuer vordefinierte Filevariablen nicht erlaubt
90	Recordlaenge nicht vertraeglich
91	Position ausserhalb des Files
99	Vorzeitiges Fileende
F0	Disketten-Schreibfehler (Diskette ist voll)
F1	Directory voll
F2	Fileumfang zu gross (Versuch mit WRITE einen 65536.Satz zu schreiben.)
FF	File nicht mehr unter Kontrolle (Versuch ein File mit CLOSE zu schliessen, das nicht mehr in der Directory steht, z.B. durch Diskettenwechsel.)

Anhang F: Interne Datenformate

Im folgenden bezeichnet @ die Adresse des 1. Byte einer Variablen eines entsprechenden Typs. Die Standardfunktion ADDR kann man dazu verwenden, diesen Wert fuer eine beliebige Variable zu erhalten.

F.1. Basis-Datentypen

F.1.1. Skalare

In einem einzigen Byte werden folgende Skalare gespeichert:
- INTEGER-Teilbereiche, wenn beide Grenzen im Bereich 0..255 liegen,
- BOOLEAN,
- CHAR und
- deklarierte ordinale Datentypen mit weniger als 256 moeglichen Werten.

Dieses Byte enthaelt die Ordnungszahl der Variablen.
Folgende ordinale Datentypen werden in zwei Byte gespeichert:
- INTEGER,
- INTEGER-Teilbereiche, wenn mindestens eine der Grenzen nicht im Bereich 0..255 liegt, und
- deklarierte ordinale Datentypen mit mehr als 256 moeglichen Werten.

Diese Bytes enthalten ein Zweierkomplement - 16-Bit-Wert, wobei der niederwertige Teil zuerst gespeichert wird (umgekehrtes Byte-Format).

F.1.2. REAL-Zahlen

REAL-Zahlen belegen 6 Bytes und stellen eine Gleitkommazahl mit 40-Bit-Mantisse und einem 8-Bit-Exponenten zur Basis 2 dar. Im ersten Byte wird der Exponent und in den naechsten 5 Byte die Mantisse gespeichert mit dem niederwertigen Byte zuerst.

```
@       Exponent
@+1     niederwertiger Teil
.
.
.
@+5     hoeherwertiger Teil
```

Der Exponent verwendet ein Binaerformat mit einem Offset von $80, d.h., ein Exponent von $84 bedeutet, die Mantisse ist mit 2^($84-$80) = 2^4 = 16 zu multiplizieren. Der Gleitkommawert ist Null, wenn der Exponent Null ist.
Den Wert erhaelt man, indem man die 40-Bit-Integerzahl (ohne Vorzeichen) durch 2^40 dividiert. Die Mantisse wird immer normalisiert, d.h., das hoechstwertige Bit (Bit 7 des fuenften Byte) wird immer als 1 interpretiert. Das Vorzeichen der Mantisse wird jedoch auch in diesem Bit gespeichert. 1 bedeutet negatives und 0 positives Vorzeichen.

F.1.3. STRING

Ein STRING belegt im Speicher immer 1 Byte mehr als seine angegebene (maximale) Laenge. Das 1. Byte enthaelt die aktuelle
Laenge des STRING; dieses wird auch als dynamisches Byte bezeichnet. Die folgenden Byte enthalten die aktuellen Zeichen des STRING. Das 1. Byte steht auf der niedrigsten Adresse. In der folgenden Tabelle bezeichnet L die aktuelle Laenge und Max die maximale Laenge des STRING:

```
@           aktuelle Laenge: L
@+1         1. Zeichen
@+2         2. Zeichen
.
.
.
@+L         letztes Zeichen
@+L+1       nicht verwendet
.
.
.
@+Max       nicht verwendet
```

Der Bereich ab @+L+1 wird nicht geloescht.

F.1.4. Mengen

Ein Element einer Menge belegt ein Bit. Die maximale Anzahl von Elementen einer Menge betraegt 256; eine Mengevariable belegt also niemals mehr als 32 (=256/8) Bytes.
Die Bytes, deren Bits alle statisch Null sind (d.h. nicht verwendet werden), werden nicht gespeichert.
Die Zahl der Bytes, die von einer Mengenvariablen belegt wird, ist (Max DIV 8) - (Min DIV 8) + 1, wobei Max und Min die oberen und unteren Grenzen des Basistyps der Menge sind. Die Speicheradresse eines speziellen Elementes E ist:

```
Elementeadresse = @ + (E DIV 8) - (Min DIV 8)
```

Die Bit-Adresse innerhalb des Bytes mit der Elementeadresse betraegt

```
BitAdresse = E MOD 8
```

wobei E die Ordnungszahl des Elementes ist.

F.1.5. FILE-Interface-Block

Jede Filevariable besitzt einen ihr zugeordneten File-Interface-Block (FIB). Ein FIB besteht aus 176 Byte und ist in zwei Abschnitte geteilt, den Steuerabschnitt (die ersten 48 Bytes) und den Sektorpuffer (die letzten 128 Bytes). Im Steuerabschnitt stehen verschiedene Informationen ueber das Diskettenfile oder Geraet, das aktuell dem File zugeordnet ist. Der Sektorpuffer wird zur Pufferung der Ein- und Ausgaben vom und zum Diskettenfile verwendet.

Die untenstehende Tabelle gibt das Format des FIB
(LSB = niedrigstes signifikantes Byte,
 MSB = hoechstes signifikantes Byte):

@	Flagbyte
@+1	File Type
@+2	Zeichenpuffer
@+3	Sektorpufferpointer
@+4	Zahl der Records (LSB)
@+5	Zahl der Records (MSB)
@+6	Recordlaenge in Bytes (LSB)
@+7	Recordlaenge in Bytes (MSB)
@+8	aktuelle Record-Nummer (LSB)
@+9	aktuelle Record-Nummer (MSB)
@+10	nicht verwendet
@+11	nicht verwendet
@+12	erstes Byte vom SCP-FCB
.	
.	
.	
@+47	letztes Byte vom SCP-FCB
@+48	erstes Byte des Sektor-Puffers
.	
.	
.	
@+175	letztes Byte des Sektor-Puffers

Das Flagbyte in @ enthaelt vier 1-Bit-Flags, die den aktuellen Status des Files indizieren:

Bit 0	Eingabe-Flag. Gleich 1, wenn Eingabe
Bit 1	Ausgabe-Flag. Gleich 1, wenn Ausgabe
Bit 2	Signalisiert Schreiben. Ist 1, wenn Daten in den Sektorpuffer geschrieben wurden
Bit 3	Signalisiert Lesen. Ist 1, wenn der Inhalt des Sektorpuffers undefiniert wird

Das Filetypbyte in @+1 definiert den Typ des aktuellen Geraetes, dass der Filevariablen zugeordnet wurde. Es koennen die folgenden Werte auftreten:

0	Consol-Geraet (CON:)
1	Terminal-Geraet (TRM:)
2	Keybord-Geraet (KBD:)
3	List-Geraet (LST:)
4	Hilfs-Geraet (AUX:)
5	Nutzer-Geraet (USR:)
6	Diskettenfile

Der Sektorpufferpointer in @+3 enthaelt ein Offset vom 1. Byte des Sektorpuffers. Die folgenden drei Felder werden nur bei Files mit wahlfreiem Zugriff und nichttypisierten Files verwendet. Jedes Feld besteht aus zwei Byte im umgekehrten Byte-Format. Die Bytes @+10 und @+11 werden gewoehnlich nicht verwendet und sind fuer spaetere Erweiterungen vorgesehen.

Die Bytes @+12 und @+47 enthalten den SCP-FILE-Control-Block (FCB). Der letzte Block des FIB ist der Sektorpuffer, der zur Pufferung der Ein- und Ausgaben von und zu Diskfiles verwendet wird.

Wird ein File einem logischen Geraet zugewiesen, sind nur die ersten drei Bytes des FIB signifikant.
Das oben beschriebene FIB-Format wird bei allen definierten Files und Textfiles verwendet. Der FIB eines nichttypisierten Files hat keinen Sektorpuffer, da die Daten direkt zwischen der Variablen und dem Diskettenfile transportiert werden. Daher betraegt die Laenge einer solchen File nur 48 Byte.

F.1.6. Zeiger

Ein Zeiger besteht aus zwei Byte, die eine 16-Bit-Speicheradresse enthalten, die im umgekehrten Byte-Format gespeichert ist, d.h., der niederwertige Adressteil wird zuerst gespeichert. Der Wert NIL entspricht einem Wort mit dem Wert Null.

F.2. Strukturen

Datenstrukturen werden entsprechend den Basistypen aufgebaut, die verschiedene Struktur-Methoden verwenden. Es gibt drei unterschiedliche Strukturmethoden:
- ARRAY,
- RECORD und
- Diskettenfiles.

Die Strukturierung der Daten beeinflusst nicht das interne Format der Datentypen.

F.2.1. ARRAY

Die Komponenten mit der niedrigsten Indexadresse werden auf der niedrigsten Speicheradresse gespeichert. ein mehrdimensionales ARRAY wird so abgespeichert, dass die am weitesten rechts stehende Dimension zuerst aufgebaut wird.

F.2.2. RECORD

Das erste Feld eines Records wird auf der niedrigsten Speicheradresse gespeichert. Die Laenge des Records ist gleich der Summe der Laenge der einzelnen Felder, wenn der RECORD keinen varianten Teil hat. Hat er einen varianten Teil, so wird die Gesamtzahl der belegten Bytes bestimmt durch die Laenge des festen Teils plus Laenge der maximalen Groesse des varianten Teils.
Jeder variante Teil beginnt an der gleichen Speicheradresse.

F.2.3. Diskettenfiles

Diskettenfiles unterscheiden sich von den anderen Datenstrukturen dadurch, dass ihre Daten nicht im internen Speicher, sondern in einem File auf einer externen Diskette gespeichert sind. Ein Diskettenfile wird bei der Uebertragung durch einen File-Interface-Block (FIB) gesteuert (siehe F.1.5.). Im allgemeinen gibt es zwei unterschiedliche Filetypen:
- Binaerfiles und
- Textfiles.

F.2.3.1. Binaerfiles

Ein Binaerfile besteht aus einer Folge von Saetzen gleicher Laenge und gleichem internen Format.
Die Saetze werden kontinuierlich hintereinander gespeichert, um die Filespeicherung zu optimieren.
Die ersten vier Byte des ersten Sektors eines Files enthaelten die Anzahl und die Laenge (in Byte) jedes Records.

Sector 0	Byte 0	Zahl des Records (LSB)
Sector 0	Byte 1	Zahl des Records (MSB)
Sector 0	Byte 2	Recordlaenge (LSB)
Sector 0	Byte 3	Recordlaenge (MSB)

Nach dieser Organisationsinformation beginnt die Nutzinformation, also der erste Record. Auf Binaerfiles kann sequentiell und wahlfrei zugegriffen werden.

F.2.3.2. Textfiles

Die Basiskomponenten eines Textfiles sind Zeichen (CHAR), und ausserdem wird jedes Textfile in Zeilen eingeteilt. Jede Zeile besteht aus einer beliebigen Zahl von Bytes und endet mit einer CR/LF-Folge ($0D/$0A). Das File wird durch das Zeichen CTRL Z ($1A) beendet (EOF).

F.3. Parameter

In Prozeduren und Funktionen werden Parameter durch den CPU-Stack uebertragen. Der generierte Maschinencode bringt automatisch die Parameterwerte mittels PUSH in den Stack und stellt es mittels POP dem Unterprogramm zur Verfuegung.
Doch im Falle der Verwendung externer Unterprogramme muss der Programmierer mittels POP diese Werte selbst dem Unterprogramm aus dem Stack uebergeben.
Zu Beginn eines EXTERNAL-Unterprogrammes enthaelt der Stack immer zuerst die Return-Adresse (ein Wort). Die Parameter, falls welche existieren, stehen im Stack hinter dieser Return-Adresse, d.h. in Richtung hoeherer Adressen. Um diese Parameter zu erhalten, muss mittels POP zunaechst die Return-Adresse, danach die Parameter uebernommen und zum Schluss mittels PUSH die Returnadresse wieder zurueck in den Stack gebracht werden.

F.3.1. Variablenparameter

Bei Variablen-Parametern (VAR) wird ein Wort in den Stack gespeichert, das die absolute Adresse des ersten Bytes des aktuellen Parameters enthaelt.

F.3.2. Wertparameter

Bei Wertparametern haengt die Form der Uebergabe vom Typ der Parameter ab. Dies wird im folgenden beschrieben.

F.3.2.1. Ordinaltyp

INTEGER, BOOLEAN, CHAR und Aufzaehlungstypen werden als ein Wort ueber den Stack uebertragen.
Belegt die Variable nur ein Byte, so ist der hoechstwertige Teil Null. Normalerweise wird fuer die POP-Anweisung "POP HL" verwendet.

F.3.2.2. REAL-Zahlen

REAL-Zahlen benoetigen im Stack 6 Byte. Man verwendet deshalb in der Uebertragung die Anweisungen

```
POP HL
POP DE
POP BC
```

Dabei enthaelt dann L den Exponenten und H das fuenfte (niedrigwertigste) Byte, E das vierte, D das dritte, C das zweite und B das erste (hoechstwertigste) Byte.

F.3.2.3. STRING

Wird ein STRING ueber den Stack uebergeben, so weist SP auf das Byte, welches die Laenge n des STRING's enthaelt. Auf den folgenden Adressen SP+1 bis SP+n stehen die Zeichen des STRING's. Zur Uebernahme kann man die folgenden Instruktionen verwenden, um den STRING aus dem Stack nach Puffer zu uebertragen:

```
LD    DE,Puffer
LD    HL,0
LD    B,H
ADD   HL,SP
LD    C,(HL)
INC   BC
LDIR
LD    SP,HL
```

F.3.2.4. Mengen

Eine Menge belegt stets 32 Byte im Stack (Mengenkomprimierung wird nur beim Laden und Speichern von Mengen verwendet).
Mit den folgenden Instruktionen kann man eine Menge aus dem Stack nach Puffer uebertragen:

```
LD   DE,Puffer
LD   HL,0
ADD  HL,SP
LD   BC,32
LDIR
LD   SP,HL
```

Damit wird das geringwertigste Byte auf die niedrigste Adresse im Puffer gespeichert.

F.3.2.5. Zeiger

Ein Zeigerwert wird ueber den Stack als ein Wort uebertragen. Es enthaelt die Speicheradresse der dynamischen Variablen. Der Wert NIL entspricht Null.

F.3.2.6. ARRAYS und RECORDS

Obgleich ARRAY- und RECORD-Parameter Wertparameter sind, werden sie nicht direkt ueber den Stack uebertragen. Es wird nur ein Wort uebertragen, das die Adresse des ersten Bytes des entsprechenden Parameters enthaelt. Im Unterprogramm ist diese Adresse mittels POP zu uebernehmen und als Quelladresse einer Block-COPY-Operation zu verwenden.

F.4. Funktionsergebnisse

Vom Nutzer geschriebene Funktionen muessen ihren Wert in folgender Weise zurueckgeben:

- Werte von ordinalen Typen werden im HL-Registerpaar uebergeben. Wenn der Typ nur aus einem Byte besteht, wird der Wert in L uebergeben, und H muss Null sein.
- REAL's werden in den Registerpaaren BC, DE, HL uebergeben; dabei enthaelt L den Exponenten und BC, DE und H die Mantisse (das hoechstwertigste Byte in B).
- STRING's und Mengen werden im Stack in dem in F.3.2.3. und F.3.2.4. beschriebenen Format uebergeben.
- Zeigerwerte werden in HL zurueckgegeben.

Anhang G: BDOS / BIOS-Rufe

G.1. BDOS-Funktionen (BDOS-Ruf: $CD/>5{CALL 5})

BDOS-/BIOS-Funktionen zur Nutzung von Komponenten des Laufzeitsystems SCPX (Erlaeuterungen sind den SCP-Systemunterlagen zu entnehmen):

n	Operation	Uebertragener Wert (Register)	Erhaltener Wert
0	Warmstart		
1	Konsoleingabe		Zeichen (A)
2	Konsolausgabe	Zeichen (E)	
3	Lesereingabe		Zeichen (A)
4	Stanzerausgabe	Zeichen (E)	
5	Druckerausgabe	Zeichen (E)	
6	Direkte Konsol-Ein-/Ausgabe	FF (Eingabe) Zeichen (Ausgabe)	0: nicht bereit oder Zeichen (A)
7	I/O-Byte lesen		Byte (A)
8	I/O-Byte setzen	Byte (E)	
9	Puffer drucken	Adresse (DE)	
10	Puffer lesen	Adresse (DE)	
11	Konsolstatus lesen		Byte (A)
13	Laufwerk Grundeinstellung (^C)		
14	Laufwerk anwaehlen	Laufwerk-Nr.(E)	
15	File eroeffnen	FCB-Adresse (DE)	Fehlercode (A)
16	File schliessen	FCB-Adresse (DE)	Fehlercode (A)
17	Erste File suchen	FCB-Adresse (DE)	Fehlercode (A)
18	Naechste Datei suchen		Fehlercode (A)
19	Datei loeschen	FCB-Adresse (DE)	Fehlercode (A)
20	Sequentiell lesen	FCB-Adresse (DE)	Fehlercode (A)
21	Sequentiell schreiben	FCB-Adresse (DE)	Fehlercode (A)
22	Datei erstellen	FCB-Adresse (DE)	Fehlercode (A)
23	Datei umbenennen	FCB-Adresse (DE)	Fehlercode (A)
24	Momentane Laufwerke bestimmen		Vektor (HL)
25	Standardlaufwerk suchen		Laufwerk (A)
26	DMA-Adresse setzen	DMA-Adresse (DE)	
27	Belegtabelle lesen		Vektor (HL)
28	Schreibschutz setzen		
29	R/O-Laufwerke finden		Vektor (HL)
30	Dateiattribute setzen	FCB-Adresse (DE)	
31	Diskettenparameter lesen		Blockadresse (HL)
32	Benutzernummer lesen oder setzen	(E) = $FF Neue Nummer (E)	Benutzer-Nummer (A)
33	Direkt lesen	FCB-Adresse (DE)	Fehlercode (A)
34	Direkt schreiben	FCB-Adresse (DE)	Fehlercode (A)
35	Dateigroesse lesen	FCB-Adresse (DE)	
36	Direkt-Satznummer lesen	FCB-Adresse (DE)	
37	Laufwerk(e) zurueck-setzen	Laufwerkvektor (DE)	

Fehlercode: $FF

G.2. BIOS-Funktionen

(n Funktionsnummer; autonomer Aufruf mit CALL xxxx, wobei xxxx = BIOS-Adresse + n * ; Uebergabe in den Registern C, BC, DE, und Rueckgabe in A oder HL ; RA ist Rueckkehrcode in A - Fehlercode erstgenannt)

n	Wirkung	Aufruf-parameter p	Rueckgabewert
0	Warmstart		
1	Konsolstatus abfragen		$FF/$00 (A)
2	Eingabe (Konsole)		Zeichen (A)
3	Ausgabe (Konsole)	Zeichen (C)	
4	Ausgabe (Drucker)	Zeichen (C)	
5	Ausgabe (Stanzer)	Zeichen (C)	
6	Eingabe (Leser)		Zeichen (A)
7	Spur 0 einstellen		Zeichen (A)
8	Laufwerk selektieren	Nummer 0..(C)	Vektor (HL)
9	Spur auswaehlen	Spur (BC)	
10	Sektor auswaehlen	Sektor (BC)	
11	Pufferadresse setzen	Adresse (BC)	
12	Selektierten Sektor lesen		RA: $01/$00
13	Selektierten Sektor schreiben		RA: $01/$00
14	Druckerstatus abfragen		RA: $FF/$00
15	Logische in physische Sektornummer umrechnen	Sektor (BC) Tabelle (DE)	Nummer (HL)

Sachwortregister

Abschrift erstellt:

Elmar Klinder
Götz Hupe

robotron

VEB Robotron Büromaschinenwerk
„Ernst Thälmann" Sömmerda
Weißenseer Straße 52
Sömmerda
DDR–5230

Exporteur:
Robotron Export-Import
Volkseigener
Außenhandelsbetrieb
der Deutschen
Demokratischen Republik
Allee der Kosmonauten 24
Berlin
DDR–1140

RI 17/87 V/12/6