TURBO PASCAL 3.0

HANDBUCH

Inhaltsverzeichnis

Anhänge

Bilder

Tabellen

Einführung

Vor Ihnen liegt das Handbuch für das Programm TURBO Pascal in der Form, wie es auf den Betriebssystemen CP/M-80, CP/M-86 und MS-DOS läuft. Obwohl darin viele Beispiele beschrieben werden, ist es nicht als Lehr- oder Textbuch gedacht, und deshalb sollten Sie wenigstens Grundkenntnisse von Pascal besitzen.

Pascal

Pascal ist eine allgemein anwendbare, 'high-level' Programmiersprache, die von Professor Nikolaus Wirth der Techn. Universität Zürich entwickelt und nach Blaise Pascal, dem berühmten Philosophen und Mathematiker aus dem 17. Jahrhundert, benannt wurde.

Diese 1971 veröffentlichte Programmiersprache sollte mittels des strukturierten Programmierens eine systematische Annäherung an die Arbeit mit Computern erlauben. Seitdem wird Pascal auf fast allen Computern in fast allen Anwendungsbereichen genutzt. Heute ist Pascal eine der meistbenutzten, 'high-level' Programmiersprachen, sowohl im Lehr- als auch im professionellen Programmbereich.

TURBO Pascal

TURBO Pascal wurde konzipiert, um allen Anforderungen des Anwenders gerecht zu werden: es unterstützt Studenten im Lernprozeß und bietet dem Programmierer ein extrem effektives Entwicklungssystem, bezogen sowohl auf die Compilierung als auch auf die Ausführungszeit, die ihresgleichen sucht.

TURBO Pascal ist eng an das Standard Pascal von K. Jensen und N. Wirth angelehnt, wie es in dem *Pascal User Manual and Report* beschrieben wird. Die wenigen und geringen Unterschiede werden in Abschnitt F beschrieben.

Hinzu kommen einige Erweiterungen:

Absolute Adressierung der Variablen
Bit/Byte Manipulierung
Direkter Zugriff auf die CPU und die Datenports
Dynamische Strings
Freie Anordnung der Sektionen innerhalb des Deklarierungsteils
Volle Unterstützung des Betriebssystems
Erzeugung von In-line Maschinencode
Include-Dateien
Logische Operationen bei ganzen Zahlen
Programm-Chaining mit allgemeinen Variablen
Overlay-System
Dateien mit direktem Zugriff
Strukturierte Konstanten
Typumwandlungs-Funktionen

Nur IBM PC und kompatible Rechner:

Farben
Graphik
Turtle-Graphik
Fenster
Sound

Dazu kommen viele neue Standardprozeduren und -funktionen, die die Handhabbarkeit von TURBO Pascal erhöhen.

Aufbau des Handbuchs

Da das Handbuch drei unterschiedliche Betriebssysteme (PC-DOS, MS-DOS, CP/M-86 und CP/M-80) behandelt, sollte der Leser folgenden Aufbau im Gedächtnis behalten:

1: Kapitel 1 beschreibt die Installierung und den Gebrauch von TURBO Pascal. Diese Information betrifft alle Betriebssysteme.

2: Der Hauptteil des Handbuchs (Kapitel 2 bis 18) beschreibt die für alle drei Betriebssysteme gültigen Teile von TURBO Pascal, bzw. diejenigen, die in allen drei Versionen gleich sind, also Standard Pascal und viele Erweiterungen. Solange Sie davon nicht abweichen, sind Ihre Programme auf allen drei Betriebssystemen lauffähig.

3: Die Kapitel 19, 20, 21 und 22 beschreiben Punkte, die nicht in den vorherigen Kapiteln beschrieben wurden, da sie sich von Version zu Version unterscheiden. Außerdem werden spezielle Features, Anforderungen und Grenzen jeder einzelnen Implementierung des jeweiligen Betriebssystems besprochen. Insbesondere sollten Sie beachten, daß Kapitel 19 alle Erweiterungen des IBM PC wie Farben, Graphik, Sound, Fenster usw. beschreibt. Um Verwirrung zu vermeiden, sollten Sie nur denjenigen Anhang lesen, der auf Ihr Betriebssystem zutrifft.

Teile von Kapitel 20, 21 und 22 beschreiben hauptsächlich die tiefergehenden Details der Programmierung (wie direkter Zugriff auf den Arbeitsspeicher und die Ports, vom Benutzer geschriebene I/O Treiber,interne Datenformate, usw.). **Es wird vorausgesetzt, daß der Leser Vorkenntnisse besitzt, und es wird kein Versuch gemacht diese Dinge zu vermitteln**. Beachten Sie auf alle Fälle, daß diese Features abhängig vom Betriebsystem sind und Programme, die diese verwenden, somit nicht mehr zwischen verschiedenen Implementierungen übertragbar sind.

Eigentlich brauchen Sie sich überhaupt nicht mit diesen Kapiteln zu befassen, wenn Sie reine Pascal Programme schreiben wollen, oder wenn die Übertragbarkeit zwischen den verschiedenen TURBO Implementierungen für Sie wichtig ist.

4: Kapitel 23 beschreibt TURBO-BCD. Dies ist eine spezielle Version von TURBO Pascal für PC-DOS, MS-DOS und CP/M-86 die binärcodierte, dezimale Arithmetik (BCD) verwendet, um höhere Genauigkeit bei Berechnungen mit reellen Zahlen zu erzielen; besonders nützlich ist dies bei Anwendungen im Geschäftsbereich.

5: Kapitel 24 beschreibt die spezielle 16-Bit TURBO 87 Version, die den 8087 Co-Prozessor für schnellere Berechnungen und erweiterten Bereich reeller Zahlen verwendet.

6: Die Anhänge sind vom Betriebssystem unabhängig und beinhalten Zusammenfassungen der Sprachelemente, der Syntaxdiagramme, Fehlermeldungen, Einzelheiten zur Installationsprozedur, ein alphabetisches Stichwortverzeichnis, usw..

7: Anhang N enthält Antworten auf die Fragen, die beim Arbeiten mit TURBO Pascal am häufigsten auftauchen. Wenn Sie Probleme haben, lesen Sie zuerst hier nach.

Typographie

Das Handbuch ist mit normalen Typen gesetzt. Spezielle Typen werden in folgenden Zusammenhängen benutzt:

Schreibmaschine
Dieser Typ wird verwandt, um Programmbeispiele zu illustrieren und die Bildschirmausgabe darzustellen. Zusätzlich wird der Bildschirm durch einen dünnen Rahmen symbolisiert.

Kursiv
Kursive Schrift hebt Teile des Textes hervor. Besonders vordefinierte Standardbezeichner und Elemente der Syntaxbeschreibungen (siehe unten) sind kursiv gedruckt. Die Bedeutung kursiver Schrift hängt vom Kontext ab.

Fettdruck
Fettdruck wird verwendet, um reservierte Wörter zu kennzeichnen. Er dient weiterhin zur Hervorhebung besonders wichtiger Textstellen.

Beschreibung der Syntax

Die gesamte Syntax von Pascal ist nach den Regeln von Backus-Naur dargestellt und wird ebenso wie Typographie und spezielle Symbole dieser Regeln im Anhang I beschrieben.

Wo es notwendig ist, werden Syntaxbeschreibungen auch ganz spezifisch benutzt, um die Syntax einzelner Sprachelemente zu verdeutlichen, wie z.B. bei der folgenden Syntaxbeschreibung der Funktion *Concat*:

Concat(*St1* , *St2*,| , *StN*|)

Reservierte Wörter werden in **Fettdruck** gesetzt, Standardbezeichner in Groß- und Kleinschreibung und im Text erklärte Elemente in *Kursivschrift* gedruckt.

Die geschweifte Klammer bedeutet in diesem Fall lediglich, daß eine beliebige Anzahl von Strings in dieser Funktion vorkommen können. Die Syntax bedeutet, daß *St1, St2 und StN* vom Typ String sein müssen. Diese Syntaxbeschreibung erklärt also allgemein die Schreibweise der Funktion. In der tatsächlichen Schreibweise könnte die Funktion dann folgendermaßen aussehen (Beispiele):

```
Concat('TURBO','Pascal')
Concat('TU','RBO','Pascal')
Concat('T','U','R','B','O',Name)
```

(Vorausgesetzt, daß *Name* eine Stringvariable ist.)

Anmerkungen

1. Die Benutzung von TURBO Pascal

Dieses Kapitel behandelt die Installierung und den Gebrauch von TURBO Pascal, insbesondere den integrierten Editor.

1.1 .COM und .CMD Dateien

Dateien mit der Bezeichnung .COM sind Programmdateien, die mit CP/M-80 und MS-DOS / PC-DOS ausgeführt werden können. Bei CP/M-86 sind diese mit .CMD bezeichnet. Wann immer im Folgenden .COM auftaucht, muß es für das Betriebssystem CP/M-86 in .CMD umgewandelt werden.

1.2 Bevor Sie beginnen

Bevor Sie beginnen, mit TURBO Pascal zu arbeiten, sollten Sie zu Ihrer eigenen Sicherheit eine Arbeitsdiskette erstellen und das Orginal als Sicherungskopie an einem unzugänglichen Ort aufbewahren. Der Vertrag über das Nutzungsrecht erlaubt Ihnen soviele Kopien anzufertigen, wie Sie wollen, vorausgesetzt, diese sind **für Ihren persönlichen Gebrauch und zur Sicherung**. Benutzen Sie ein Dateikopierprogramm und vergewissern Sie sich, daß alle Dateien überspielt wurden.

1.3 Unbedingt Lesen !!!

TURBO Pascal besitzt einige Compilerbefehle, die die Möglichkeiten der speziellen Laufzeit-Einrichtungen kontrollieren, wie z.B. Indexüberprüfung, Rekursion usw. Bitte beachten Sie, daß die Voreinstellung dieser Compilerbefehle die Ausführungsgeschwindigkeit optimiert und die Codelänge minimiert. So sind eine Reihe der Voreinstellungen ausgeschaltet, bis sie vom Benutzer explizit aktiviert werden. Alle Compilerbefehle und ihre Voreinstellungen werden in Anhang C besprochen. (Bei 16-Bit Versionen ist Rekursion immer möglich; Ausschaltung der Rekursion ist nur bei CP/M-80 möglich.)

1.4 Die Dateien auf der Originaldiskette

Ihre Diskette enthält folgende Dateien:

TURBO.COM Das TURBO Pascal Programm. Wenn Sie das Kommando TURBO eingeben, wird das Programm geladen und steht für Sie bereit.

TURBO.OVR Die Overlay-Datei für TURBO.COM (Nur in der CP/M-80 Version). Sie muß nur dann auf der Diskette sein, wenn Sie .COM Dateien von TURBO aus ausführen wollen.

TURBO.MSG Die Textdatei, die die Fehlermeldungen enthält. Falls Sie keinen Wert auf erklärende Fehlermeldungen beim Compilieren legen, muß sie nicht auf Ihrer Laufzeit-Diskette sein. In diesem Fall werden Fehler nur als Nummern ausgegeben, und das Handbuch kann zur Erklärung herangezogen werden. Da das System automatisch den Fehlerbereich bestimmt, könnten Sie es vielleicht als Vorteil empfinden, ohne diese Datei zu arbeiten. Es spart Ihnen nicht nur auf der Diskette Platz, sondern stellt Ihnen auch 1,5 KByte zusätzlichen Speicher für Ihr Programm zur Verfügung. Diese Datei kann auch editiert werden, falls Sie Fehlermeldungen in eine andere Sprache übersetzen wollen - mehr dazu im Anhang H.

TINST.COM Das Installierungsprogramm. Geben Sie TINST ein, und das Programm führt Sie durch eine menü-gesteuerte Installierungsprozedur. Es ist auf der Laufzeit-Diskette wie auch die folgenden Programme, nicht notwendig.

TINST.DTA Terminal-Installierungsdaten (beim IBM PC nicht vorhanden).

TINST.MSG Meldungen des Installierungs-Programms. Auch diese Datei kann in jede beliebige Sprache übersetzt werden.

.PAS Dateien Programmbeispiele

GRAPH.P Nur bei IBM PC-Versionen. Enthält die **external** Deklarierungen, die für die erweiterte Graphik und Turtle-Graphik-Routinen notwendig sind; diese befinden sich in der Datei GRAPH.BIN. Die Datei muß sich nur auf der Laufzeit-Diskette befinden, wenn Sie Turtle-Graphik verwenden wollen.

GRAPH.BIN Nur bei IBM PC-Versionen. Die Datei enthält die Assembler-Routinen für erweiterte Graphik und Turtlegraphik. Sie muß sich nur auf der Laufzeit-Diskette befinden, wenn Sie erweiterte, oder Turtle-Graphik verwenden wollen.

READ.ME Diese Datei enthält, falls vorhanden, die neuesten Korrekturen und Vorschläge für den Gebrauch des Systems.

Nur *TURBO.COM* **muß** auf ihrer Laufzeit-Diskette sein. Ein voll arbeitsfähiges TURBO Pascal erfordert also nur **30 K** Platz auf der Diskette (37 K für 16-Bit Systeme). *TURBO.OVR* ist nur erforderlich, wenn Sie von dem TURBO-Menü aus Programme ausführen wollen. *TURBO.MSG* ist notwendig, wenn Sie Compiler-Fehlermeldungen mit erklärendem Text haben wollen. Alle *TINST* Dateien sind nur für die Installierungsprozedur nötig, die *GRAPH* Dateien nur bei Verwendung von erweiterter oder Turtle-Graphik. Die *.PAS* Beispieldateien können mit auf die Laufzeit-Diskette übernommen werden, sind aber nicht erforderlich.

1.5 Start des TURBO Pascal

Wenn Sie das System auf Ihre Arbeitsdiskette kopiert haben, geben Sie das Kommando:

TURBO

auf Ihrem Terminal ein. Das System antwortet mit der folgenden Meldung:

```
TURBO Pascal system          Version N.NNX
                             |SYSTEM|

Copyright (C) 1983, 1984 by BORLAND Inc

No terminal selected

Include error messages (Y/N)? ■
```

Bild 1-1: Einschalt-Meldung

N.NNX bezeichnet Ihre Version, |*SYSTEM*| symbolisiert das Betriebssystem (z.B. CP/M-86 auf IBM-PC). Die dritte Zeile gibt über das installierte Terminal Auskunft. Im Moment ist keines installiert, mehr dazu folgt später.

Wenn Sie als Antwort auf die Frage ein **Y** eingeben, wird die Datei mit den Fehlermeldungen in den Speicher eingelesen (vorausgesetzt, sie befindet sich auf der Diskette), gleich darauf erscheint kurz die Meldung Loading TURBO.MSG. Sie können auch **N** eingeben und sich damit 1,5 KByte Speicherplatz sparen. Daraufhin erscheint das Hauptmenü von TURBO Pascal:

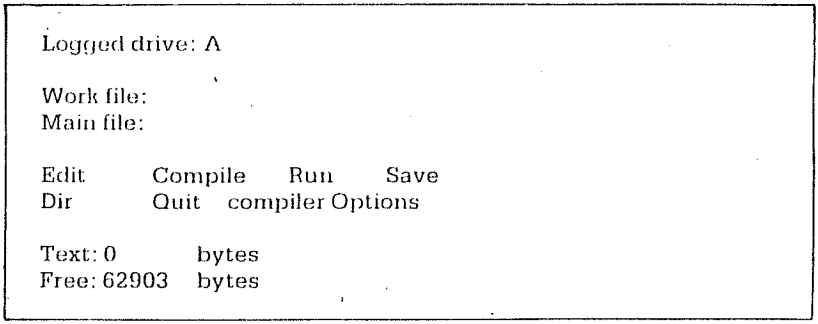

Bild 1-2: Haupt-Menü

Das Menü zeigt die verfügbaren Kommandos. Jedes Einzelne wird in den folgenden Abschnitten genau erklärt. Jedes Kommando wird ausgeführt, indem Sie den entsprechenden Großbuchstaben eingeben (falls Ihr Terminal diese Möglichkeit hat, leuchtet der entsprechende Großbuchstabe im Menü). Drükken Sie nicht <RETURN>, da der Befehl sofort ausgeführt wird. Die Anzeige für das angemeldete (logged) Laufwerk und die Speicherplatzbelegung dienen nur als Beispiel. Die aktuelle Anzeige entspricht Ihrem Computer.

IBM PC Besitzer müssen TURBO Pascal nicht erst installieren und brauchen folglich erst auf Seite 14 weiter zu lesen. Wenn Sie keinen IBM PC besitzen, können Sie TURBO auch ohne Installierung verwenden, falls Sie beabsichtigen, ohne den eingebauten Editor zu arbeiten. Wenn Sie diesen benutzen wollen, geben Sie **Q** ein und verlassen Sie TURBO für eine Minute, um die Installierung durchzuführen.

1.6 Installierung

Tippen Sie TINST, um das Installierungs-Programm zu starten. Alle *TINST* Dateien und die *TURBO.COM* Datei müssen auf dem angemeldeten Laufwerk sein. Es erscheint dieses Menü:

```
TURBO Pascal installation menu.
Choose installation item from the following:

|S|creen installation | |C|ommand installation | |Q|uit

Enter S, C, or Q:
```

Bild 1-3: Installierungs-Hauptmenü

1.6.1 IBM PC Bildschirm-Installierung

Geben Sie **S** ein, um die Bildschirm-Installierung durchzuführen. Es erscheint ein Menü, das Ihnen die Wahl des Bildschirmmodus für den Gebrauch von TURBO ermöglicht (siehe Anhang L für detailierte Hinweise). Wenn Sie Ihre Wahl getroffen haben, erscheint das Hauptmenü wieder und Sie können jetzt mit **C**, der Kommando-Installierung (beschrieben auf den Seiten 350 ff) weitermachen oder die Installierung hier abbrechen, indem Sie **Q** eingeben (**Q** steht für Quit - beenden).

1.6.2 Bildschirm-Installierung bei anderen Computern

Geben Sie **S** (steht für screen) ein, um die Bildschirminstallierung anzuwählen. Ein Menü erscheint, das die meisten handelsüblichen Terminals enthält. Nun können Sie durch Eingabe der entsprechenden Nummer das von Ihnen gewünschte Terminal wählen. Falls Ihr Terminal nicht darunter ist und auch nicht kompatibel zu einem der aufgelisteten ist (beachten Sie, daß das ADM-3A Terminal zu sehr vielen kompatibel ist), müssen Sie die Installierung selbst vornehmen. Dies ist ganz einfach, aber Sie müssen doch das Handbuch Ihres Terminals zu Rate ziehen, um die Fragen, die Ihnen das Menü stellt, beantworten zu können. Schauen Sie sich dazu Anhang L an.

Wenn Sie ein Terminal gewählt haben, werden Sie gefragt, ob Sie die Installierung modifizieren wollen, bevor das Terminal endgültig installiert wird. Dies kann sein, wenn Sie z.B. ein ADM-3A-kompatibles Terminal mit einigen zusätzlichen Features haben. Wählen Sie ADM-3A und fügen Sie die gewünschten Kommandos hinzu, um die speziellen Features zu aktivieren. Wenn Sie mit **Y** antworten, werden Ihnen einige Fragen gestellt. Diese werden in Anhang L beschrieben.

Normalerweise werden Sie mit **N** antworten, was Ihre Zufriedenheit mit der existierenden Terminalinstallierung ausdrückt. Abschließend werden Sie nach der Arbeitsfrequenz Ihrer CPU gefragt. Geben Sie den entsprechenden Wert ein (2, 4, 6, oder 8 MHz, in den meisten Fällen sind es 4 MHz).

Danach erscheint wieder das Menü. Nun können Sie mit **C** der Installierung der Kommandozeichen fortfahren (nächster Abschnitt), oder Sie beenden die Installierung, indem Sie **Q** für Beenden eingeben.

1.6.3 Installierung der Editor-Kommandos

Der integrierte Editor besteht aus einer Reihe von Kommandos, die den Cursor auf dem Bildschirm bewegen, Texte einfügen, löschen usw.. Jede dieser Funktionen kann durch zwei Komanndos, ein erstes und ein zweites, aktiviert werden. Die zweiten Kommandos sind von Borland bereits installiert und entsprechen dem Standard von *WordStar*. Die ersten Kommandos sind für die meisten Systeme undefiniert, sie können leicht nach Ihrem Geschmack, passend für Ihre Tastatur neu festgelegt werden. Beim IBM PC sind die Pfeiltasten und andere Funktionstasten bereits als erste Kommandos installiert, diese sind in Kapitel 19 beschreiben.

In Anhang L finden Sie eine vollständige Beschreibung der Installierung der Editor-Kommandos.

1.7 Das Hauptmenü

Nach der Installierung rufen Sie wieder TURBO Pascal auf, indem Sie das Kommando TURBO eingeben. Ihr Bildschirm sollte nun das Menü darstellen, diesmal mit leuchtenden Anfangsbuchstaben der Kommandos. Falls dies nicht der Fall ist, sollten Sie Ihre Installierungsdaten überprüfen.

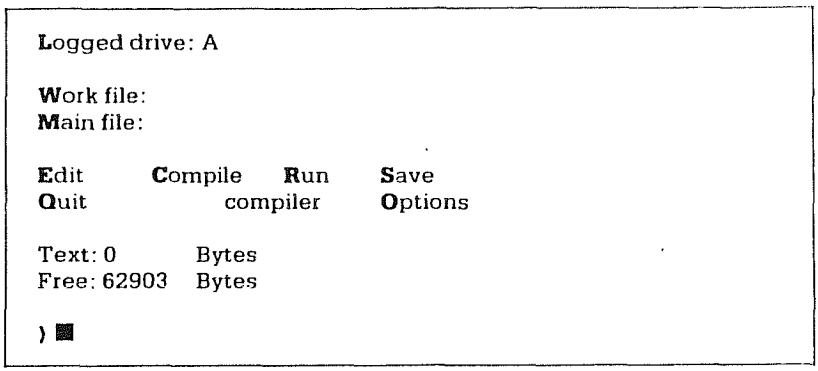

Bild 1-4: Hauptmenü

Wann immer im Folgenden der Darstellungsmodus leuchtend, oderblinkenderwähnt wird, wird natürlich angenommen, daß Ihr Bildschirm diese Bildattribute besitzt, um Text z.B. in verschiedenen Intensitäten, invers, unterstrichen oder sonstwie darzustellen. Wenn dies nicht der Fall ist, überlesen Sie Blinkmodus, da wo er erwähnt wird, einfach.

Das Menü zeigt Ihnen die verfügbaren Kommandos, wenn Sie mit TURBO Pascal arbeiten. Das Kommando wird ausgeführt, wenn Sie den entsprechenden Großbuchstaben eingeben, der leuchtet. Drücken Sie nicht <RETURN>, da das Kommando sofort nach Eingabe des Großbuchstabens ausgeführt wird. Das Menü verschwindet zwar, wenn Sie mit dem System arbeiten, aber es läßt sich leicht wieder auf den Bildschirm holen: Geben Sie einen unerlaubten Befehl ein, z.B. <RETURN> oder <SPACE>.

Die folgenden Abschnitte beschreiben jedes Kommando im Detail.

1.7.1 Die Wahl des angemeldeten Laufwerks

Das **L** Kommando (Logged) wird benutzt, um das Laufwerk zu wechseln. Es erscheint folgende Meldung:

New drive: ■

Sie sollten nun den neuen Laufwerksnamen eingeben, d.h. einen der Buchstaben A bis P, wahlweise gefolgt von einem Doppelpunkt und abgeschlossen mit <RETURN>. Wenn sie den gegewärtigen Wert nicht ändern wollen, geben Sie nur <RETURN> ein. Das **L** Kommando führt einen Disk-Reset aus, auch wenn Sie das Laufwerk nicht gewechselt haben. Deshalb sollte es bei jedem Diskettenwechsel benutzt werden, um Konflikte bei Disketten-Operationen zu vermeiden (nur bei CP/M-80).

Das neue Laufwerk wird nicht sofort im Menü angezeigt, da kein automatischer Update stattfindet. Für die Ausgabe eines neuen Menüs müssen Sie <SPACE> eingeben.

1.7.2 Die Wahl der Arbeitsdatei

Das **W** Kommando wird zur Auswahl der zu bearbeitenden Datei benutzt. Nach seiner Eingabe erscheint auf dem Bildschirm:

Work file name: ■

Sie antworten mit einem legalen Dateinamen, d.h. mit einem Name, der aus 1 bis 8 Buchstaben, optional einem Punkt und möglicherweise dem Dateityp (max. 3 Buchstaben) besteht:

FILENAME.TYP

Wenn Sie einen Namen ohne Punkt und Dateityp eingeben, wird automatisch der Typ *.PAS* generiert. Es ist möglich einen Dateinamen ohne Typ einzugeben, indem hinter den Namen nur der Punkt gesetzt wird.

Beispiele:

PROGRAM	wird zu PROGRAM.PAS
PROGRAM.	wird nicht verändert
PROGRAM.FIL	wird nicht verändert

Die Dateitypen .BAK, .CHN, und .COM/.CMD sollten vermieden werden, da TURBO Pascal diese Namen zu einem bestimmten Zweck verwendet.

Nachdem die Arbeitsdatei angegeben wurde, wird sie von der Diskette eingelesen. Wenn die Datei noch nicht besteht, erscheint als Meldung New file. Wenn Sie eine soeben editierte Datei noch nicht abgespeichert haben, erscheint die Meldung:

Work file X:FILENAME.TYP not saved. Save (Y/N)? ■

Das soll Sie davor warnen, eine neue Datei in den Speicher zu laden und damit diejenige, an der Sie gerade arbeiten, zu überschreiben und zu zerstören. Antworten Sie **Y** zur Speicherung und **N** zur Löschung .

Der neue Name der Arbeitsdatei erscheint im Menü nach dem nächsten Update, d.h. wenn Sie <SPACE> eingeben.

1.7.3 Die Wahl der Hauptdatei

Das **M** Kommando wird benutzt, um eine Hauptdatei zu bestimmen, wenn Sie den Compilerbefehl **$I** verwenden, um Dateien einzubinden. Die Hauptdatei sollte diejenige sein, die die Einbindungsanweisungen enthält. Dann können Sie die Arbeitsdatei als eine von der Hauptdatei unterschiedene definieren. Damit können Sie verschiedene eingebundene Dateien (Include-Dateien) editieren und den Namen der Hauptdatei unverändert lassen.

Wenn die Compilierung gestartet ist, und die Arbeitsdatei nicht die Hauptdatei ist, wird die aktuelle Arbeitsdatei automatisch gespeichert und die Hauptdatei in den Arbeitsspeicher geladen. Wenn während der Compilierung ein Fehler gefunden wurde, wird die Datei, die den Fehler enthält (unabhänig davon, ob es die Hauptdatei, oder eine eingebundene ist) automatisch Arbeitsdatei, die dann editiert werden kann. Wenn der Fehler korrigiert und die Compilierung wieder gestartet ist, wird die korrigierte Arbeitsdatei automatisch gesichert und die Hauptdatei wieder geladen.

Die Benennung der Hauptdatei erfolgt, wie im Vorhergehenden besprochen.

1.7.4 Das Edit-Kommando

Das **E** Kommando ruft den integrierten Editor auf, um die Datei, die als Arbeitsdatei bezeichnet wurde, editieren zu können. Wenn keine Arbeitsdatei angegeben ist, werden Sie aufgefordert, dies zu tun. Das Menü verschwindet und der Editor ist aktiviert. Mehr darüber auf Seite 19 ff.

Während Sie mit TURBO Pascal compilieren können, ohne ein Terminal installiert zu haben, setzt der Gebrauch des Editors die Terminalinstallierung voraus. Vergleichen Sie dazu Seite 12.

1.7.5 Das Compile-Kommando

Das **C** Kommando aktiviert den Compiler. Wenn keine Hauptdatei angegeben ist, wird die Arbeitsdatei compiliert, ansonsten die Hauptdatei. Wurde die Arbeitsdatei editiert, werden Sie gefragt, ob diese abgespeichert werden soll, bevor die Hauptdatei geladen und compiliert wird. Die Compilierung kann jederzeit durch drücken einer Taste unterbrochen werden.

Die Compilierung endet entweder mit einem Programm, das im Arbeitsspeicher verbleibt, oder mit einer .COM, bzw. .CHN Datei. Diese Wahl können Sie im Optionen-Menü des Compilers treffen (beschrieben auf Seite 190 für PC/MS-DOS, Seite 227 für CP/M-86 und Seite 259 für CP/M-80). Die Voreinstellung beläßt das Programm im Arbeitspeicher.

1.7.6 Run-Kommando

Das **R** Kommando aktiviert ein im Arbeitsspeicher befindliches Programm, oder spricht eine Datei mit TURBO Objektcode (eine .COM oder.CMD Datei) an, falls im Optionenmenü des Compilers die Option **C** eingeschaltet ist. Falls schon ein compiliertes Programm im Speicher ist, wird dieses aktiviert. Wenn nicht findet die Compilierung automatisch nach den oben beschriebenen Regeln statt.

1.7.7 Save-Kommando

Das **S** Kommando sichert die aktuelle Arbeitsdatei auf Diskette. Wenn eine alte Version dieser Datei existiert, wird diese ineine .BAK umbenannt und die neue Version abgespeichert.

1.7.8 Das-Directory Kommando

Das **D** Kommando gibt das Dateienverzeichnis (Directory) und die Information über den verbleibenden Speicherplatz auf dem angemeldeten Laufwerk aus. Wenn sie **D** drücken, erhalten Sie folgende Meldung:

```
Dir mask: ■
```

Sie geben entweder einen Laufwerksnamen, einen Laufwerksnamen gefolgt von einem Dateinamen oder eine Maske, die eine der üblichen allgemeinen Symbole * oder ? enthält, ein. Oder Sie drücken lediglich <RETURN>, um die gesamte Directory aufgelistet zu bekommen.

1.7.9 Das Quit-Kommando (Beenden)

Das **Q** Kommando wird benutzt, um das TURBO System zu verlassen. Wenn die Arbeitsdatei nach dem Laden bearbeitet wurde, werden Sie gefragt, ob Sie sie abspeichern wollen, bevor Sie beenden.

1.7.10 Compiler-Optionen

Das **O** Kommando wählt ein Menü an, in dem Sie einen Überblick über einige voreingestellte Werte des Compilers erhalten und diese ändern können. Außerdem bietet es eine hilfreiche Funktion zur Auffindung von Laufzeit-Fehlern in Programmen.

Da sich diese Optionen je nach Betriebssytem unterscheiden, werden sie auf Seite 20, 21 und 22 besprochen.

1.8 Der TURBO Editor

Der integrierte Editor ist ein Bildschirmeditor, der speziell zur Programmtexterstellung geschaffen wurde. Falls Sie mit MicroPro's *WordStar* vertraut sind, benötigen Sie keine weitere Einführung in die Handhabung des Editors, da die Standardfunktionen exakt denen von *WordStar* entsprechen. Es gibt einige kleine Unterschiede, der TURBO Editor besitzt darüberhinaus einige Erweiterungen; diese werden auf Seite 34 besprochen. Sie können, wie auf Seite 13 beschrieben, Ihre eigenen Kommandos, oben in die *WordStar* Kommandoliste installieren; IBM PC Systeme sind bereits mit Pfeilen und eigenen Funktionstasten ausgestattet. Die *WordStar* Kommandos können aber trotzdem benutzt werden.

Die Benutzung des Editors ist sehr einfach: Wenn Sie eine Arbeitsdatei definiert haben und **E** eingeben, verschwindet das Menü und der Editor ist aktiviert. Wenn die Arbeitsdatei im Laufwerk ist, wird sie geladen und die erste Seite des Textes erscheint. Wenn es eine neue Datei ist, ist der Bildschirm ab der *Statuszeile* leer.

Sie verlassen den Editor und kehren durch drücken von **Ctrl-K-D** zum Menü zurück; mehr darüber erfahren Sie später.

Der Text wird, wie auf einer Schreibmaschine, auf der Tastatur eingegeben. Um eine Zeile zu beenden, drücken Sie <RETURN>, <CR> oder <ENTER> (oder wie immer die entsprechende Taste auf Ihrer Tastatur heißt). Wenn Sie Ihren Bildschirm vollgeschrieben haben, wird die oberste Zeile nach oben weggeschoben. Haben Sie keine Angst, damit ist sie nicht verloren. Mit dem jeweiligen Editier-Kommando können Sie im Text auf- und abrollen (scrollen). Dies wird später besprochen.

Schauen wir uns zuerst die *Statuszeile* am oberen Bildschirmrand an.

1.8.1 Die Statuszeile

Die oberste Zeile auf dem Bildschirm ist die Statuszeile. Sie enthält folgende Informationen:

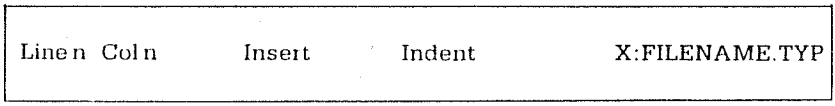

Bild 1-5: Statuszeile des Editors

Line n
Zeigt die Nummer der Zeile an, in der sich der Cursor befindet, vom oberen Bildschirmrand her gezählt.

Col n
Zeigt die Nummer der Spalte an, in der der Cursor steht, von der linken Seite her gezählt.

Insert
Hier wird angezeigt, daß die Zeichen, die über die Tastatur eingegeben werden, an der Cursorposition eingefügt werden. Der bereits existierende Text rechts vom Cursor wird entsprechend der Länge des neuen Textes nach rechts verschoben. Wenn Sie das Kommando *insert mode on/off* (**Ctrl-V** nach Voreinstellung) eingeben, erscheint stattdessen **Overwrite** (überschreiben). Nun wird der Text an der Cursorposition überschrieben und nicht nach rechts verschoben.

Indent
Gibt an, daß die automatische Tabulierfunktion eingeschaltet ist. Mit dem Kommando *auto-indent on/off* (**Ctrl-Q Ctrl-I** nach Voreinstellung) kann ein- oder ausgeschaltet werden.

X:FILENAME.TYP
Laufwerk, Name und Typ der Datei, die editiert werden soll.

1.8.2 Editier-Kommandos

Wie schon erwähnt, wird der Text wie auf einer Schreibmaschine geschrieben. Da Sie aber auf einem Computer arbeiten, werden Ihnen einige Editiermöglichkeiten geboten, die die Textbearbeitung und in diesem Fall das Schreiben von Programmen, sehr vereinfachen.

Der TURBO Editor erlaubt bis zu 45 Editier-Kommandos, die den Cursor bewegen, durch den Text blättern, Textstrings finden und ersetzen usw.. Die Kommandos können in vier Gruppen unterteilt werden:

Kommandos zur Cursor-Steuerung
Kommandos zum Einfügen und Löschen
Block-Kommandos
Weitere Kommandos

Jede dieser Gruppen enthält in sich zusammenhängende Kommandos, die in den folgenden Abschnitten einzeln besprochen werden. Die folgende Tabelle gibt einen Überblick über die vorhandenen Kommandos:

CURSORSTEUERUNGS-KOMMANDOS :

Zeichen links	Oberer Bildschirmrand
Zeihen rechts	Dateianfang
Wort links	Dateiende
Wort rechts	Zeile links
Zeile nach oben	Zeile rechts .
Rollen nach oben	Blockanfang
Rollen nach unten	Blockende
Seite nach oben	Letzte Cursor-Position
Seite nach unten	

EINFÜGEN U. LÖSCHEN-KOMMANDOS:

Einfügen-Modus an/aus	Wort rechts löschen
Zeile einfügen	Zeichen unter Cursor löschen
Löschen bis Zeilenende	Zeile löschen
Zeichen links löschen	

BLOCK-KOMMANDOS:	VERSCHIEDENE KOMMANDOS
Blockanfang markieren	Ende des Editierens
Blockende markieren	Tabulator
einzelnes Wort markieren	Auto Tab an/aus
Block kopieren	Zeile sichern
Block bewegen	Suchen
Block löschen	Suchen/Tauschen
Block v. Diskette lesen	Letztes Finden wiederholen
Block a. Diskette schreiben	Kontrollzeichen-Präfix
Block verdecken/zeigen	

Tabelle 1-2: Editierkommandos

In so einem Fall lernt man am besten, indem man arbeitet. Starten Sie Turbo, bestimmen Sie eines der Programmbeispiele als Arbeitsdatei und geben Sie **E** ein, um editieren zu können. Dann probieren Sie die Kommandos aus, so wie Sie sie lesen.

Bleiben Sie dran, auch wenn Sie es am Anfang schwierig finden. Es ist kein Zufall, daß der Editor *WordStar* kompatibel ist. Die Logik dieser Kommandos, einmal gelernt, wird zu einem Teil Ihrer selbst. Lassen Sie sich das von jemanden gesagt sein, der schon einige MegaBytes Text mit diesem Editor hinter sich hat.

Jede der folgenden Beschreibungen besteht aus einer Kopfzeile, die das Kommando bezeichnet, gefolgt von der Angabe der Tasten, die das Kommando ausführen. Dazwischen ist Platz, so daß sie die Tasten eintragen können, die Sie auf Ihrem Terminal installiert haben. Wenn Sie Pfeiltasten und bestimmte Tasten haben, die speziell für die Textverarbeitung belegt sind (Einfügen, Löschen, usw.), kann es für Sie günstiger sein, diese zu benutzen. Einzelheiten zur Installierung finden Sie auf Seite 13 ff.

Die aufgeführten Beschreibungen gehen von der Benutzung *WordStar* kompatibler Tastaturen aus.

1.8.3 Eine Bemerkung zu den Kontrollzeichen

Alle Kommandos sind so angelegt, daß sie Kontrollzeichen verwenden. Ein Kontrollzeichen ist ein spezielles Zeichen, das von Ihrer Tastatur erzeugt wird, indem Sie die <Ctrl>- oder <CONTROL>-Taste gleichzeitig mit einem Buchstaben von A bis Z drücken.

Die <Ctrl>-Taste arbeitet wie die <Shift>-Taste. Wenn Sie <SHIFT> und ein A gleichzeitig drücken, erscheint auf dem Bildschirm ein A; wenn Sie die <Control>-Taste und A drücken, erhalten Sie ein Kontroll-A (Ctrl-A).

1.8.4 Bevor Sie anfangen: Wie Sie wieder aufhören können

Das Kommando, mit dem Sie den Editor verlassen und wieder in das Hauptmenü zurückkehren können ist auf Seite 30 beschrieben. Aber vielleicht wollen Sie schon jetzt wissen, daß Sie dazu **CTRL-K-D** drücken müssen. Dieses Kommando sichert aber nicht automatisch die Datei; dies muß vom Menü aus mit dem **S**ave Kommando geschehen.

1.8.5 Cursorsteuerungs-Kommandos

1.8.5.1 Kommandos der Grundbewegungen

Das Erste, was man über einen Editor wissen muß, ist, wie man den Cursor auf dem Bildschirm bewegt. Der TURBO Editor benutzt dazu eine Gruppe spezieller Kontrollzeichen, namentlich die Kontrollzeichen **A, S, D, F, E, R, X,** und **C**.

Warum gerade diese? Weil sie sehr nahe neben der **CTRL**-Taste liegen, so daß sie immer beide Tasten mit einer Hand bedienen können. Außerdem ist ihre Position auf der Tastatur so, daß sie ihre Funktion mit ihrer Lage logisch versinnbildlichen. Schauen wir uns die Grundbewegungen |rauf|, |runter|, |links| und |rechts| an:

```
    E
S       D
    X
```

Die Lage der vier Buchstaben zeigt schon optisch an, daß **CTRL-E** den Cursor nach oben, **CTRL-X** nach unten, **CTRL-S** nach links und **CTRL-D** nach rechts bewegt. Versuchen Sie nun den Cursor mit diesen vier Steuerzeichen auf dem Bildschirm hin- und herzubewegen. Wenn ihre Tastatur Wiederhol-Tasten hat, können Sie durch anhaltendes Drücken der **CTRL**- und einer der vier anderen Tasten den Cursor sehr schnell über den Bildschirm bewegen.

Nun schauen wir uns einige Erweiterungen zu diesen Bewegungen an:

```
    E  R
A   S  D  F
    X  C
```

Die Lage von **CTRL-R** neben **CTRL-E** impliziert, daß damit der Cursor nach oben bewegt wird, nur nicht um eine Zeile, sondern um eine ganze Seite. Entsprechend bewegt **CTRL-C** den Cursor um eine ganze Seite nach unten.

Entsprechendes gilt für **CTRL-A** und **Ctrl-F**: **CTRL-A** bewegt den Cursor um ein ganzes Wort nach links, **CTRL-F** um ein ganzes Wort nach rechts.

Die beiden letzten Grundbewegungen steuern den Cursor nicht nur auf dem Bildschirm, sondern lassen den ganzen Bildschirm in der Datei nach oben oder unten rollen:

```
W  E  R
A  S  D  F
Z     X  C
```

CTRL-W rollt in der Datei nach oben (die Zeilen des Bildschirms bewegen sich nach unten) und **CTRL-Z** rollt nach oben (die Zeilen des Bildschirms bewegen sich nach oben).

Zeichen nach links **Ctrl-S**

Bewegt den Cursor ein Zeichen nach links, ohne dieses Zeichen zu verändern. <BACKSPACE> kann die gleiche Funktion übernehmen. Das Kommando führt den Cursor nicht über das Zeilenende hinaus, d. h., wenn der Cursor den linken Rand des Bildschirms erreicht, stoppt er.

Zeichen nach rechts **Ctrl-D**

Bewegt den Cursor ein Zeichen nach rechts, ohne dieses Zeichen zu verändern. Das Kommando führt den Cursor nicht in die nächste Zeile, d.h. wenn der Cursor den rechten Rand des Bildschirms erreicht, beginnt der Text spaltenweise nach links auszuwandern, bis der Cursor die Spalte 128, den äußersten rechten Rand, erreicht, wo er stoppt.

Wort nach links **Ctrl-A**

Bewegt den Cursor zum Wortbeginn nach links. Ein Wort ist als eine Sequenz von Zeichen definiert, die von den Zeichen: I space I, <> , ; . ()|| ^ ' " + - / $ begrenzt wird. Dieses Kommando gilt über das Zeilenende hinaus.

Wort nach rechts **Ctrl-F**

Bewegt den Cursor zum Wortbeginn nach rechts. Zur Definition von Wort siehe oben. Führt den Cursor auch in die nächste Zeile.

Zeile nach oben **Ctrl-E**

Bewegt den Cursor um eine Zeile nach oben. Wenn er die oberste Zeile erreicht hat, rollt der Bildschirm um eine Zeile nach unten.

Zeile nach unten **Ctrl-X**

Bewegt den Cursor eine Zeile nach unten. Wenn der Cursor die unterste Zeile erreicht hat, rollt der Bildschirm um eine Zeile nach oben.

Aufwärts rollen **Ctrl-W**

Rollt den Cursor gegen den Anfang der Datei, jeweils um eine Zeile (d.h. der ganze Bildschirminhalt rollt nach unten). Der Cursor bleibt auf der Zeile, bis diese das untere Ende des Bildschirms erreicht.

Abwärts rollen **Ctrl-Z**

Rollt den Cursor gegen das Ende der Datei, jeweils um eine Zeile (d.h. der ganze Bildschirminhalt rollt nach oben). Der Cursor bleibt auf der Zeile, bis diese den oberen Rand des Bildschirms erreicht.

Seite nach oben **Ctrl-R**

Bewegt den Cursor um eine Seite nach oben, mit einer Überlappung von einer Zeile, d.h. er bewegt sich um eine Bildschirmseite, abzüglich einer Zeile, im Text zurück.

Seite nach unten **Ctrl-C**

Bewegt den Cursor um eine Seite nach unten, mit einer Überlappung von einer Zeile, d.h. er bewegt sich um eine Bildschirmseite, abzüglich einer Zeile, im Text nach vorne.

1.8.5.2 Kommandos für erweiterte Bewegungen

Die eben besprochenen Kommandos erlauben es Ihnen, sich frei im Text zu bewegen, sie sind leicht zu erlernen und zu verstehen. Wenn Sie einige Zeit damit arbeiten, merken Sie wie einfach es ist.

Wenn Sie sie beherrschen, werden Sie den Cursor auch einmal schneller bewegen wollen. Der Editor von TURBO stellt 5 Kommandos zur Verfügung, die es erlauben, sehr schnell zu den äußeren Enden der Zeilen, des Textes und zur letzten Cursorposition zu gelangen.

Diese Kommandos erfordern die Eingabe **zweier** Zeichen, **CTRL-Q** und dann eines der folgenden Kontrollzeichen (**CTRL-**)**S, D, E, X, R** und **C**.Die ersten vier wurden schon vorher besprochen:

E R

S D

X C

d.h: **CTRL-Q-R** bewegt den Cursor zum Beginn, **CTRL-Q-C** zum Ende des Textes, **CTRL-Q-S** zum äußersten linken Ende und **CTRL-Q-D** zum äußersten rechten Ende der Zeile. **CTRL-Q-E** bewegt den Cursor an den oberen Rand des Bildschirms, **CTRL-Q-X** an den unteren Rand des Bildschirms.

Zeile links **Ctrl-Q-S**

Bewegt den Cursor zum äußersten linken Rand des Bildschirms, d.h. auf die Spalte 1.

Zeile rechts **Ctrl-Q-D**

Bewegt den Cursor zum Ende der Zeile, d.h. zu der Stelle, die auf das letzte eingegebene Zeichen dieser Zeile folgt. Daran anschließende Leerzeichen werden, um Platz zu sparen, in allen Fällen weggelassen.

Oberer Bildschirmrand **Ctrl-Q-E**

Bewegt den Cursor an den oberen Bildschirmrand.

Unterer Bildschirmrand **Ctrl-Q-X**

Bewegt den Cursor an den unteren Bildschirmrand.

Textbeginn **Ctrl-Q-R**

Bewegt den Cursor auf das erste Zeichen der Datei.

Textende **Ctrl-Q-C**

Bewegt den Cursor zum letzten Zeichen der Datei.

CTRL-Q zusammen mit einem **B, K** oder **P** ermöglicht es Ihnen, innerhalb der Datei weit zu springen:

Blockanfang **Ctrl-Q-B**

Bewegt den Cursor an die Stelle der *Blockanfang*-Markierung, die mit **Ctrl-K-B** gesetzt wurde (deshalb das **Q-B**). Das Kommando funktioniert auch, wenn der Block nicht dargestellt ist (siehe später *Verdecken/Zeigen*), oder keine *Blockende*-Markierung gesetzt ist.

Blockende **Ctrl-Q-K**

Bewegt den Cursor an die Stelle der *Blockende*-Markierung, die mit **Ctrl-K-K** gesetzt wurde (deshalb das **Q-K**) Das Kommando funktioniert auch, wenn der Block nicht dargestellt ist (siehe später *Verdecken/Zeigen*), oder keine *Blockanfang*-Markierung gesetzt ist.

Letzte Cursorposition **Ctrl-Q-P**

Bewegt den Cursor an seine vorherige Position (**P** soll an Position erinnern). Beispielsweise, um den Cursor nach einem 'Sichern' oder nach einem 'Suchen/Tauschen' auf seine letzte Position zurück zu bewegen.

1.8.6 Einfügen- und Löschen-Kommandos

Mit diesen Kommandos können Sie Zeichen, Wörter und Zeilen einfügen (insert) oder löschen (delete). Sie können in drei Gruppen eingeteilt werden: Eine betrifft die Art der Textbehandlung (Einfügen oder Überschreiben), die zweite Gruppe ist die der einfachen Kommandos, und die dritte, die der erweiterten Kommandos.

Beachten Sie, daß der Editor eine Rücknahmemöglichkeit besitzt, die es Ihnen erlaubt, Veränderungen an einer Zeile rückgängig zu machen, solange der Cursor diese Zeile nicht verlassen hat. Das Kommando dazu ist **CTRL-Q-L** und wird auf Seite 31 genauer beschrieben.

Einfügen-Modus an/aus **Ctrl-V**

Wenn Sie Text eingeben, können Sie zwischen zwei Eingabemodi wählen: *Einfügen* und *Überschreiben*. Der bei Aufruf des Editors voreingestellte Einfügemodus erlaubt es Ihnen, in bestehenden Text neuen einzufügen. Der bestehende Text rechts vom Cursor wird dabei weiter nach rechts verschoben.

Der Modus 'Überschreiben' kann gewählt werden, wenn Sie den alten Text durch neuen ersetzten wollen. Die neu eingegebenen Zeichen ersetzen dabei die, die sich gerade unter dem Cursor befinden.

Zwischen den Modi schalten Sie mit der Eingabe <CTRL>-V hin und her.
Der aktuelle Modus wird in der Statuszeile am oberen Bildschirmrand angezeigt.

Linkes Zeichen löschen **<DEL>**

Bewegt den Cursor um eine Stelle nach links und löscht das dort befindliche Zeichen. Jedes Zeichen rechts vom Cursor rutscht gleichzeitig um eine Stelle nach links. Die Taste <BACKSPACE>, die normalerweise, wie **Ctrl-S**, den Cursor um eine Stelle nach links bewegt, ohne das dortige Zeichen zu verändern, kann auch für dieses Kommando benutzt werden, wenn Sie dies wünschen. Dies ist vorteilhaft, wenn diese Taste für Sie besonders angenehm zu erreichen ist, oder wenn Ihre Tastatur keine <DELETE>-Taste besitzt (diese kann u.U. auch <DEL>, <RUBOUT> oder <RUB> benannt sein). Dieses Kommando führt den Cursor auch in die vorhergehende Zeile. Sie können es also auch verwenden, um das Kommando <NEW LINE>, bzw. <RETURN> zu löschen.

Zeichen unter Cursor löschen **Ctrl-G**

Löscht das Zeichen unter dem Cursor und bewegt alle Zeichen rechts davon um eine Stelle nach links. Es können nur Zeichen innerhalb der Zeile gelöscht werden.

Rechtes Wort löschen **Ctrl-T**

Löscht das Wort rechts vom Cursor. Zur Definition von Wort siehe Seite 24. Befindet sich rechts vom Cursor kein Zeichen mehr in der Zeile, so werden die Weiterschaltung (<RETURN>) und anschließend die Wörter der nächsten Zeile gelöscht.

Zeile einfügen **Ctrl-N**

Fügt an der Cursorposition eine neue Zeile ein, ohne den Cursor zu bewegen.

Zeile löschen **Ctrl-Y**

Löscht die Zeile, in der sich der Cursor befindet und bewegt alle Zeilen darunter um eine Zeile nach oben. Der Cursor bewegt sich zum linken Rand des Bildschirms. Eine gelöschte Zeile ist nicht mehr rekonstruierbar, Sie sollten bei der Anwendung dieses Kommandos also vorsichtig sein.

Löschen bis zum Zeilenende **Ctrl-Q-Y**

Löscht die Zeile von der Cursorposition bis zum Zeilenende.

1.8.7 Block-Kommandos

Alle Block-Kommandos sind erweiterte Kommandos (d.h. Kommandos, die sich in der Standarddefinition aus zwei Kontrollzeichen zusammensetzen). Sie können diese Definitionen auch übergehen, wenn Sie sich momentan ein bißchen verwirrt fühlen. Später, wenn Sie ganze Textteile bewegen, löschen oder kopieren wollen, sollten Sie diesen Abschnitt durcharbeiten.

Nichtsdestotrotz erörtern wir jetzt den Gebrauch von Blöcken.

Ein Textblock ist einfach eine Menge Text, von einem Zeichen bis zu mehreren Seiten. Ein Block wird durch eine Anfangsmarkierung *Begin block* vor dem ersten Zeichen und eine Endmarkierung *End block* nach dem letzten Zeichen des gewünschten Textblocks gekennzeichnet. So markiert, kann er nun kopiert, bewegt, gelöscht oder abgespeichert werden. Es existiert auch ein Kommando, das eine, auf einer Diskette befindliche Datei als Block in den Text lädt. Außerdem gibt es ein spezielles Kommando, das ein einzelnes Wort als Block kennzeichnet.

Blockanfang-Markierung **Ctrl-K-B**

Dieses Kommando markiert den Beginn des Blocks. Die Markierung selbst wird auf dem Bildschirm nicht dargestellt. Der Block wird als markiert erst sichtbar, wenn Ihr Bildschirm die Möglickeit zur Hervorhebung von Zeichen besitzt. Trotz alledem ist der Block intern markiert und damit manipulierbar.

Blockende-Markierung **Ctrl-K-K**

Dieses Kommando markiert das Ende des Blocks. Entsprechend der Markierung für den Blockanfang ist sie nicht sichtbar, wenn nicht beide Markierungen gesetzt sind, oder Ihr Computer keine Möglichkeit hat, Zeichen hervorgehoben darzustellen.

Markierung eines einzelnen Wortes **Ctrl-K-T**

Dieses Kommando markiert ein einzelnes Wort als Block und ersetzt die Block-Anfang/-Endemarkierung, die für ein einzelnes Wort zu umständlich wäre. Wenn der Cursor innerhalb des Worts plaziert ist, wird es markiert. Wenn nicht, ist das Wort links vom Cursor markiert. Zur Definition eines Worts siehe Seite 24.

Block verdecken/zeigen **Ctrl-K-H**

Dieses Kommando schaltet die sichtbare Markierung eines Blocks (hervorgehobener Text) entweder aus oder ein. Blockmanipulations-Kommandos (kopieren, bewegen, löschen und schreiben auf eine Datei) funktionieren nur, wenn der Block dargestellt ist. Blockbezogene Cursorsteuerungs-Kommandos (Springen an Blockanfang/-ende) sind bei verdecktem und angezeigtem Block anwendbar.

Kopieren eines Blocks **Ctrl-K-C**

Dieses Kommando kopiert den vorher markierten Block und plaziert ihn beginnend an der Position des Cursors. Der Block, der kopiert wurde, bleibt unverändert, der neue Block besitzt ebenfalls die Markierungen. Wenn kein Block markiert wurde, findet kein Kopiervorgang statt und es wird auch keine Fehlermeldung ausgegeben.

Versetzen eines Blocks **Ctrl-K-V**

Dieses Kommando versetzt den markierten Block an die Stelle des Cursors. Die ursprüngliche Stelle ist daraufhin leer. Der Block besitzt an seiner neuen Position noch die Markierungen. Wenn kein Block markiert ist, findet keine Versetzung statt und es wird keine Fehlermeldung ausgegeben.

Löschen eines Blocks **Ctrl-K-Y**

Dieses Kommando löscht einen markierten Block. Dieser Vorgang ist nicht rückgängig zu machen, seien Sie also vorsichtig.

Lesen eines Blocks aus einer Datei **Ctrl-K-R**

Dieses Kommando liest eine Datei von Diskette und fügt sie an der aktuellen Cursorposition in den Text ein. Der eingelesene Block ist markiert. Wenn dieses Kommando gegeben wird, werden Sie aufgefordert, den Namen der einzulesenden Datei anzugeben. Jeder legale Dateiname ist möglich, .PAS wird automatisch dazugeschrieben. Eine Datei ohne Typenbezeichnung hat nach dem Namen einen Punkt.

Schreiben eines Blocks in eine Datei **Ctrl-K-W**

Dieses Kommando speichert einen markierten Block als Datei ab. Der Block wird unverändert gelassen, die Markierungen verbleiben an ihrer Stelle. Wenn dieses Kommando eingegeben wird, werden Sie nach dem Namen der Datei gefragt. Wenn die Datei bereits existiert, werden Sie gewarnt, bevor eine Überschreibung stattfindet. Wenn kein Block markiert ist, passiert nichts, es wird auch keine Fehlermeldung ausgegeben. Jeder legale Dateiname ist zulässig, .PAS wird automatisch dazugeschrieben. Eine Datei ohne Typenbezeichnung hat nach dem Namen einen Punkt. Vermeiden Sie den Gebrauch der Dateitypenbezeichnungen .BAK, .CHN, und .COM/.CMD, da diese im TURBO System besondere Verwendungen haben.

1.8.8. Weitere Editier-Kommandos

Der folgende Abschnitt enthält eine Reihe von Kommandos, die logisch nicht in eine der obigen Kategorien fallen, aber trotzdem wichtig sind, insbesondere diese ersten:

Beenden des Editierens **Ctrl-K-D**

Dieses Kommando beendet das Editieren und führt zum Hauptmenü zurück. Das Editieren fand ausschließlich im Arbeitsspeicher statt. Um nun die editierte Datei abzuspeichern, muß entweder das Kommando Save aus dem Hauptmenü verwendet werden, oder es geschieht automatisch in Verbindung mit einer Compilierung oder der Definition einer neuen Arbeitsdatei.

Tabulierung **TAB/Ctrl-I**

Der Editor von TURBO Pascal hat keine festgesetzten Tabulatorpositionen. Stattdessen werden die Tabulierungen automatisch zu Beginn jedes Worts auf der Zeile über dem Cursor gesetzt. Dies ist besonders nützlich beim Editieren von Programmen, wo Sie oft Spalten zueinander in Beziehung stehender Punkte aufbauen wollen, z.B. die Deklarierung von Variablen und ähnliches. Denken Sie daran, daß Pascal Ihnen erlaubt, besonders schöne Sourcetexte zu schreiben. Tun Sie es, nicht aus Purismus, sondern um Ihre Programme leichtverständlich und übersichtlich zu gestalten. Das hilft Ihnen, falls Sie diese nach einiger Zeit verändern müssen.

Automatische Tabulierung an/aus **Ctrl-Q-I**

Die Einrichtung für automatische Tabulierung, falls sie eingeschaltet ist, wiederholt die Spalten der vorhergehenden Zeile, d.h. wenn Sie ein <RETURN> eingeben, geht der Cursor nicht zum Zeilenanfang, sondern zur ersten benutzten Spalte der darüber stehenden Zeile. Wenn Sie eine andere Spalte wünschen, benutzen Sie lediglich eines der Kommandos zur Cursorsteuerung links/rechts. Wenn die automatische Tabulierung eingeschaltet ist, erscheint in der Statuszeile des Editors **Indent**, ansonsten wird nichts angezeigt. In der Voreinstellung ist die Tabulierungsfunktion aktiv.

Zeilensicherung **Ctrl-Q-L**

Dieses Kommando erlaubt alle Änderungen in einer Zeile rückgängig zu machen, *solange Sie die Zeile nicht verlassen haben*. Unabhängig von der Art der Änderung wird diese Zeile in ihrem ursprünglichen Zustand erscheinen, falls Sie in der Zeile geblieben sind. Aus diesem Grund ist bei Verwendung des Lösch-Kommandos (**Ctrl-Y**) *delete line* die Zeile unwiderruflich verloren. Wenn Sie eines Tages auf der Ctrl-Y Taste einschlafen, hilft nur eine lange Arbeitspause.

Finden **Ctrl-Q-F**

Dieses Kommando gibt Ihnen die Möglichkeit, einen String von max. 30 Zeichen zu suchen. Wenn Sie es eingeben, wird die Statuszeile gelöscht und Sie werden aufgefordet, einen Suchstring einzugeben. Machen Sie dies und drükken Sie anschließend <RETURN>. Der Suchstring kann alle Zeichen, auch Kontrollzeichen enthalten. Kontrollzeichen werden in den Suchstring mittels des **Ctrl-P** Präfix eingegeben: Wenn Sie z.B. ein **Crtl-A** eingeben wollen, drükken Sie die <CTRL>-Taste und gleichzeitig erst P dann A. Um in den String das Kommando <RETURN> einzugeben, tippen Sie **CTRL-M CTRL-J**. Beachten Sie, daß **CTRL-A** eine besondere Bedeutung hat, es steht für jedes Zeichen und dient damit quasi als Joker in Suchstrings.

Suchstrings können mit den Kommandos *Zeichen links, Zeichen rechts, Wort links* und *Wort rechts* editiert werden. *Wort rechts* bringt den vorigen Suchstring zurück, der dann editiert werden kann. Die Suche kann mit dem Abbruch-Kommando (**Ctrl-U**) abgebrochen werden.

Wenn Sie einen Suchstring angegeben haben, werden Sie nach den Suchoptionen gefragt. Die folgenden Optionen sind verfügbar:

B Suche von der Cursorposition rückwärts bis zum Beginn des Textes.
G Suche im gesamten Text, unabhängig von der Cursorposition.
n n steht für eine beliebige Zahl, die Sie wählen können. Suche nach dem n-ten Vorkommen des Suchstrings, gezählt von der aktuellen Cursorposiiton.
U Ignorieren von Groß- und Kleinschreibung, d.h. Groß- und Kleinbuchstaben werden gleich behandelt.
W Suche nach ganzen Wörtern. Zeichenfolgen, die in Wörtern eingebettet sind, werden ignoriert.

Beispiele:
W Wenn der Suchstring Zeit ist, wird nur das Wort Zeit gesucht. Das Wort Zeitgeist würde ignoriert werden.
BU Suche rückwärts, unabhängig von Groß- und Kleinschreibung. Bei dem Suchstring Block wird auch blocken und BLOCKADE gefunden.
125 Suche das 125. Vorkommen des Suchstrings.

Nach der Eingabe der Optionen tippen Sie <RETURN>, und die Suche beginnt. Wenn im Text eine Textfolge existiert, die dem Suchstring entspricht, stellt sich der Cursor an das Ende der Kombination. Mit dem Kommando für *Wiederhole letzte Suchfunktion* (**Ctrl-L**) kann die Suche wiederholt werden.

Suchen und Ersetzen **Ctrl-Q-A**

Das Suchen und Ersetzen-Kommando erlaubt Ihnen, eine Zeichenkombination von max. 30 Zeichen zu suchen und durch eine Kombination mit ebenfalls bis zu 30 Zeichen zu ersetzen. Wenn Sie das Kommando eingeben, wird die Statuszeile gelöscht und Sie werden aufgefordert den Suchstring einzugeben. Machen Sie dies und drücken Sie <RETURN>. Zur Eingabe von Kontrollzeichen müssen Sie den **Ctrl-P** Präfix vorausstellen, z.B. Sie geben **Ctrl-A** ein, indem Sie die <CTRL>-Taste und gleichzeitig erst P und dann A drücken. Das Editieren des Strings und der Abbruch der Operation entspricht dem beim 'Finden-Kommando' beschriebenem Vorgehen.

Suchstrings können mit den Kommandos *Zeichen links, Zeichen rechts, Wort links* und *Wort rechts* editiert werden. *Wort rechts* bringt den vorigen Suchstring zurück, der dann editiert werden kann. Die Suche wird mit dem Abbruch-Kommando (**Ctrl-U**) abgebrochen.

Wenn der Suchstring bestimmt ist, werden Sie aufgefordert, den String einzugeben, der diesen ersetzen soll. Bis zu 30 Zeichen sind möglich. Die Eingabe von Kontrollzeichen und das Editieren geht wie oben beschrieben. Beachten Sie aber, daß **Ctrl-A** keine besondere Bedeutung in einem Suchstring hat. Wenn Sie nur <RETURN> eingeben, wird der gefundene String durch nichts ersetzt, d.h. gelöscht.

Schließlich können Sie noch unter folgenden Optionen wählen:

B Suche und ersetze rückwärts, d.h. von der aktuellen Cursorposition bis zum Beginn des Textes.
G Suche im gesamten Text, unabhängig von der Cursorposition.
n n steht für eine beliebige *integer* Zahl. Suche und ersetze n-mal die nächsten vorkommenden, im Suchstring definierten Zeichenkombinationen von der aktuellen Cursorposition ab.
N Ersetze ohne Nachfrage, d.h. die Frage *Replace Y/N (Ersetzen Ja/Nein)* entfällt.
U Ignoriere Groß- und Kleinschreibung.
W Suche und ersetze nur ganze Wörter. Zeichenfolgen, die in Wörter eingebettet sind, werden ignoriert.

Beispiele:
N10 Suche und ersetze die nächsten 10 Kombinationen, ohne nachzufragen.
GWU Finde und ersetze nur ganze Wörter im gesamten Text. Ignoriere Groß- und Kleinschreibung.

Beenden Sie die Wahl der Optionen mit <RETURN>, dann wird die Suche eingeleitet. Wenn die Zeichenfolge gefunden ist, befindet sich der Cursor an deren Ende und Sie werden gefragt, ob eine Ersetzung gewünscht wird: *Replace (Y/N) ?* Die Frage erscheint in der Statuszeile, aber nur wenn die Option **N** nicht gewählt wurde. Sie können an dieser Stelle die gesamte Operation abbrechen, indem Sie das Abbruch-Kommando **Ctrl-U** eingeben. Mit **Ctrl-L** können Sie die Operation wiederholen.

Wiederholen der letzten Suche **Ctrl-L**

Dieses Kommando wiederholt das letzte 'Suchen und Ersetzen'-Kommando so, als ob es ganz neu eingegeben worden wäre.

Kontrollzeichen-Präfix **Ctrl-P**

Der Editor von TURBO erlaubt Ihnen Kontrollzeichen in eine Datei einzugeben, indem Sie bei der Eingabe das Präfix **Ctrl-P** voranstellen. Wenn Sie in einen Text ein Kontrollzeichen eingeben wollen, müssen Sie zuerst **Ctrl-P** und dann das gewünschte Kontrollzeichen eingeben. Kontrollzeichen werden als gedimmte Großbuchstaben (oder invers, je nach Terminal) dargestellt.

Abbruch **Ctrl-U**

Das Kommando **Ctrl-U** erlaubt Ihnen, den Abbruch jeder Operation, wenn eine Eingabe möglich ist, z.B. bei der Abfrage *Replace (Y/N)* des 'Suchen und Ersetzen-Kommandos', ebenso bei der Eingabe eines Suchstrings oder Dateinamens (Block Lesen/Schreiben).

1.9 Der Editor von TURBO und WordStar

Wenn jemand *WordStar* gewöhnt ist, wird er merken, daß sich einige Kommandos von TURBO Pascal und *WordStar* unterscheiden. Obwohl die Kommandos von TURBO nur eine Untermenge bilden, war es notwendig, einige einzuführen, die *WordStar* nicht hat. Die Unterschiede werden in diesem Abschnitt besprochen.

1.9.1 Cursor-Steuerung

Die Kommandos **Ctrl-S**, **D**, **E** und **X** bewegen den Cursor frei auf dem Bildschirm umher und fixieren ihn nicht automatisch auf Spalte 1, wenn die Zeile leer ist. Das heißt nicht, daß Sie mit diesen Kommandos den Bildschirm mit Leerzeichen füllen könnten. Im Gegenteil, führende Leerzeichen werden automatisch gelöscht. Diese Art den Cursor zu bewegen, ist besonders dann nützlich, wenn tabulierte Programmabschnitte editiert werden, z.B. **begin** und **end** Anweisungen, die eingerückt sind.

Ctrl-S und **Ctrl-D** bringen den Cursor nicht in eine neue Zeile. Um dies zu erreichen, müssen Sie die Kommandos **Ctrl-E**, **X**, **A** und **F** benutzen.

1.9.2 Markierung einzelner Wörter

Ctrl-K-T wird benutzt, um einzelne Wörter als Block zu markieren. Dies ist einfacher, als sowohl den Anfang als auch das Ende eines Wortes jeweils einzeln markieren zu müssen.

1.9.3 Beenden des Editierens

Das Kommando **Ctrl-K-D** beendet das Editieren und bringt das Hauptmenü zurück auf den Bildschirm. Anders als in *WordStar*, wird bei TURBO Pascal das Editieren ausschließlich im Arbeitsspeicher durchgeführt, deshalb werden die Dateien auf Diskette nicht verändert. Dies muß ausdrücklich mit dem Sichern-Kommando des Hauptmenüs erfolgen, oder automatisch, in Verbindung mit der Compilierung, bzw. der Definition einer neuen Arbeitsdatei. Das Kommando **Ctrl-K-D** in TURBO hat auch deshalb nichts mit **Ctrl-K-Q** in *WordStar* zu tun, weil in TURBO der editierte Text im Speicher verbleibt, um compiliert oder gesichert zu werden.

1.9.4 Automatische Zeilensicherung

Das Kommando **Ctrl-K-L** erlaubt Änderungen an einer Zeile ungeschehen zu machen, d.h. den Zustand vor dem Editieren wiederherzustellen, *solange der Cursor die Zeile nicht verlassen hat.*

1.9.5 Tabulator

Es gibt keine festgesetzten Tabulatorwerte. Stattdessen werden die Tabulierungen automatisch zu Beginn jedes Wortes der über dem Cursor liegenden Zeile gesetzt, d.h. jedes Wort in der darüberliegenden Zeile entspricht einem Tabulatorwert.

1.9.6 Automatische Tabulierung

Das Kommando **Ctrl-Q-I** schaltet die automatische Tabulierung ein und aus.

Anmerkungen:

2. GRUNDLEGENDE SPRACHELEMENTE

2.1 Grundlegende Symbole

Das wesentliche Vokabular von TURBO Pascal besteht aus Symbolen, die in Buchstaben, Zahlen und Spezialsymbole eingeteilt werden können:

Buchstaben A bis Z, a bis z und _ (Unterstreichung)
Zahlen 0 1 2 3 4 5 6 7 8 9
Spezialsymbole + - * / = ^ < > [ ] ( ) { } . , : ; ' # $

Zwischen Groß- und Kleinschreibung wird keine Unterscheidung getroffen. Bestimmte Operatoren und Begrenzer werden aus zwei Spezialsymbolen gebildet:

Zuweisungs-Operator: :=
Relationaler Operator: <> <= >=
Teilbereichs-Begrenzer: ..
Klammern: (.und.) können anstatt [und] verwendet werden
Kommentare: (* und *) können anstatt { und } verwendet werden

2.2 Reservierte Wörter

Reservierte Wörter sind ein integraler Bestandteil von TURBO Pascal und können nicht neu definiert werden. Sie können also nicht als vom Benutzer definierte Bezeichner verwendet werden. Die reservierten Wörter sind:

```
* absolute    * external      nil          * shl
  and           file          not          * shr
  array         forward     * overlay      * string
  begin         for           of             then
  case          function      or             type
  const         goto          packed         to
  div         * inline        procedure      until
  do            if            program        var
  downto        in            record         while
  else          label         repeat         with
  end           mod           set          * xor
```

Im gesamten Handbuch sind die reservierten Wörter **fett** gedruckt. Die Sterne zeigen reservierte Wörter an, die in Standard Pascal nicht definiert sind.

2.3 Standardbezeichner

TURBO Pascal führt folgende Standard-Bezeichner von vordefinierten Typen, Konstanten, Variablen, Prozeduren und Funktionen. Jeder dieser Bezeichner kann neu definiert werden, was aber bedeutet, daß ihre Möglichkeiten eingeschränkt werden. Solche Veränderungen können auch leicht zu Verwirrung führen. Die folgenden Standard-Bezeichner sind deshalb am besten in ihrer ursprünglichen Definition zu belassen:

FOTO VON ENGL. TABELLE S. 38

Addr	Delay	Length	Release
ArcTan	Delete	Ln	Rename
Assign	EOF	Lo	Reset
Aux	EOLN	LowVideo	Rewrite
AuxInPtr	Erase	Lst	Round
AuxOutPtr	Execute	LstOutPtr	Seek
BlockRead	Exit	Mark	Sin
BlockWrite	Exp	MaxInt	SizeOf
Boolean	False	Mem	SeekEof
BufLen	FilePos	MemAvail	SeekEoln
Byte	FileSize	Move	Sqr
Chain	FillChar	New	Sqrt
Char	Flush	NormVideo	Str
Chr	Frac	Odd	Succ
Close	GetMem	Ord	Swap
ClrEOL	GotoXY	Output	Text
ClrScr	Halt	Pi	Trm
Con	HeapPtr	Port	True
ConInPtr	Hi	Pos	Trunc
ConOutPtr	IOresult	Pred	UpCase
Concat	Input	Ptr	Usr
ConstPtr	InsLine	Random	UsrInPtr
Copy	Insert	Randomize	UsrOutPtr
Cos	Int	Read	Val
CrtExit	Integer	ReadLn	Write
CrtInit	Kbd	Real	WriteLn
DelLine	KeyPressed		

Jede Implementierung von TURBO Pascal enthält ferner eine Reihe spezieller Standardbezeichner, die in den Kapiteln 20, 21 und 22 aufgelistet sind.

Im gesamten Handbuch werden die Standardbezeichner, wie auch alle anderen Bezeichner (siehe Seite 43) mit Klein- und Großbuchstaben geschrieben. Im Text werden sie *kursiv* gedruckt.

2.4 Begrenzer

Sprachelemente müssen wenigstens von einem der folgenden Begrenzer unterbrochen werden: Leerzeichen, neue Zeile oder Kommentar.

2.5 Programmzeilen

Programmzeilen können maximal 127 Zeichen lang sein, darüber hinausgehende Zeichen werden vom Compiler ignoriert. Aus diesem Grund erlaubt der TURBO Editor auch nur 127 Zeichen in einer Zeile, aber Source, die mit einem anderen Editor vorbereitet wurde, könnte längere Zeilen enthalten. In einem solchen Fall wird der Text in den TURBO Editor gelesen, Zeilenendmarkierungen werden automatisch angefügt und es wird eine Warnung ausgegeben.

Anmerkungen:

3. SKALARE STANDARDTYPEN

Ein Datentyp definiert die Art der Werte, die eine Variable annehmen kann. Jede Variable in einem Programm darf einem und nur einem Datentyp zugeordnet sein. Obwohl Datentypen in TURBO Pascal sehr komplex sein können, werden sie alle aus einfachen (unstrukturierten) Typen aufgebaut.

Ein einfacher Typ kann entweder vom Programmierer definiert sein (er wird dann *deklarierter, skalarer Typ* genannt), oder er ist einer der *skalaren Standardtypen*: **integer**, **real**, **boolean**, **char** oder **byte**. Es folgt eine Beschreibung dieser fünf skalaren Standardtypen.

3.1 Integer (ganze Zahlen)

Integers sind ganze Zahlen, die bei TURBO Pascal auf einen Bereich von -32768 bis 32767 begrenzt sind. Ganze Zahlen belegen zwei Byte im Speicher.

Überlauf von arithmetischen Operationen mit ganzen Zahlen wird nicht entdeckt. Beachten Sie besonders, daß Teilergebnisse in ganzzahligen Ausdrükken innerhalb der Bereichsgrenzen für ganze Zahlen liegen müssen. Beispielsweise ergibt der Ausdruck 1000*100/50 nicht 2000, da die Multiplikation zu einem Überlauf führt.

3.2 Byte

Der Typ *Byte* ist ein Teilbereich des Typs *Integer* mit den Grenzen 0 und 255. Bytes sind deshalb mit dem Typ *Integer* kompatibel, d.h. wann immer ein Wert vom Typ *Byte* erwartet wird, kann ein *integer* Wert angegeben werden und umgekehrt, **außer** bei der Übergabe von Parametern. Weiterhin können *Bytes* und *Integers* in Ausdrücken gemischt werden und *Byte* Variable können *integer* Werte zugewiesen bekommen. Eine Variable vom Typ *Byte* belegt ein Byte im Speicher.

3.3 Real (reelle Zahlen)

Der Bereich reeller Zahlen (Datentyp *real*)ist 1E-38 bis 1E+38 mit einer Mantisse mit bis zu 11 signifikanten Stellen. Reelle Zahlen belegen 6 Bytes im Speicher.

Bei einer arithmetischen Operation mit reellen Zahlen verursacht ein Überlauf einen Programmstop und die Anzeige eines Ausführungsfehlers. Eine Unterschreitung der Bereichsgrenze führt zu einem Ergebnis von Null.

Obwohl der Typ **real** zu den skalaren Standardtypen gehört, sollte folgender Unterschied zwischen dem Typ **real** und anderen skalaren Typen beachtet werden:

1) Die Funktionen *Pred* und *Succ* dürfen keine reellzahligen Argumente enthalten.
2) Der Typ *real* darf nicht bei der Indexierung von Arrays verwendet werden.
3) Der Typ *real* kann nicht verwendet werden, um den Grundtyp einer Menge zu definieren.
4) Der Typ *real* kann nicht in kontrollierenden **for** und **case** Anweisungen verwendet werden.
5) Teilbereiche des Typ *real* sind nicht erlaubt.

3.4 Boolean (Bool'sche Wahrheitswerte)

Ein Bool'scher Wahrheitswert kann einen der beiden logischen Werte wahr oder falsch, die durch die Standardbezeichner *True* bzw. *False* bezeichnet sind, annehmen. Diese sind so definiert, daß *True* < *False* ist. Eine *boolean* Variable belegt ein Byte im Speicher.

3.5 Char (alphanumerische Zeichen)

Ein *Char* (alphanumerischer) Wert entspricht einem Zeichen aus der ASCII Zeichenmenge. Die Zeichen sind entsprechend ihrem ASCII Wert geordnet, z.B. 'A' < 'B'. Die ordinalen (ASCII) Werte der Zeichen reichen von 0 bis 255. Eine *Char* Variable belegt ein Byte im Speicher.

4. BENUTZERDEFINIERTE SPRACHELEMENTE

4.1 Bezeichner

Bezeichner (engl: identifier) werden verwendet, um Labels, Konstanten, Typen, Variablen, Prozeduren und Funktionen zu bezeichnen. Ein Bezeichner besteht aus einem Buchstaben oder einer Unterstreichung gefolgt von beliebigen Kombinationen von Buchstaben, Zahlen oder Unterstreichungen. Ein Bezeichner wird in der Länge nur von der Länge der Zeile, die maximal 127 Zeichen betragen kann, begrenzt, und alle Zeichen sind signifikant.

Beispiele:

TURBO	
square	
persons_counted	
BirthDate	
3rdRoot	illegal, da eine Zahl am Anfang steht
Two Words	illegal, da kein Leerzeichen enthalten sein darf

Da TURBO Pascal nicht zwischen Groß- und Kleinschreibung unterscheidet, hat die Verwendung von Groß- und Kleinbuchstaben wie z.B. bei *BirthDate* keine Auswirkung. Diese wird jedoch empfohlen, da sie die Lesbarkeit erhöht. *SehrLangerBezeichner* ist leichter für das menschliche Auge zu lesen, als *SEHRLANGERBEZEICHNER*. Die vermischte Verwendung von Groß- und Kleinschreibung wird im gesamten Handbuch für Bezeichner gebraucht.

4.2 Zahlen

Zahlen sind Konstanten vom Typ *integer* oder *real*. Integer Konstanten sind ganze Zahlen, die in dezimaler oder hexadezimaler Notation ausgedrückt sind. Hexadezimale Konstanten werden dadurch identifiziert, daß ihnen ein Dollarzeichen voransteht: $ABC ist eine hexadezimale Konstante. Der dezimale Integer-Bereich ist - 32768 bis 32767 und der hexadezimale Integer-Bereich ist $0000 bis $FFFF.

Beispiele:

```
1
12345
-1
$123
$ABC
$123G          illegal, da G keine legale hexadezimale Zahl ist
.2345          illegal, da eine ganze Zahl keine Stelle hinter dem
               Komma haben kann
```

Der Bereich reeller Zahlen hat eine Mantisse von 1E-38 bis 1E+38, die 11 signifikante Stellen aufweist. Sie können Exponentenschreibweise verwenden, wobei der Buchstabe E dem Exponentialfaktor vorausgeht und bedeutet '10 mal die Potenz von'. Eine *integer* Konstante ist überall erlaubt, wo eine *real* Konstante stehen kann. Trennungszeichen sind innerhalb von Zahlen nicht erlaubt.

Beispiele:

```
1.0
1234.5678
-0.012
1E6
2E-5
-1.2345678901E+12
1       zulässig, aber es ist keine reelle, sondern eine ganze Zahl
```

4.3 Strings

Eine Stringkonstante ist eine Sequenz von Zeichen, die mit einfachen Anführungszeichen eingefaßt ist:

```
'Dies ist eine String-Konstante'
```

Ein einzelnes Anführungszeichen kann in einem String stehen, indem es doppelt geschrieben wird. Strings, die nur ein einziges Zeichen enthalten, gelten als Standardtyp *Char*. Ein String ist mit einem **array of** *Char* derselben Länge kompatibel. Alle Stringkonstanten sind mit allen **String**-Typen kompatibel.

Beispiele:

```
'TURBO'
'You''ll see'
''''
''''
''
```

Wie man an Beispiel 2 und 3 sieht, wird ein einzelnes Anführungszeichen in einem String als zwei aufeinanderfolgende Anführungszeichen geschrieben. Die vier aufeinanderfolgenden Anführungszeichen in Beispiel 3 stellen einen String dar, der *ein einzelnes* Anführungszeichen enthält.

Das letzte Beispiel - Anführungszeichen, die keine Zeichen umschließen, bedeuten einen *leeren String* - ist nur mit **String**-Typen kompatibel.

4.3.1 Kontrollzeichen

Mit TURBO Pascal können Sie auch Kontrollzeichen in Strings einbetten. Es werden zwei Notationen für Kontrollzeichen unterstützt:

1) Das Symbol #, gefolgt von einer *integer* Konstanten im Bereich 0..255, bezeichnet ein Zeichen des entsprechenden ASCII Werts.
2) Das Symbol ^ gefolgt von einem Zeichen, bezeichnet das entsprechende Kontrollzeichen.

Beispiele:

#10	ASCII 10 dezimal (Zeilen-Vorschub)
#$1B	1ASCII 1B hex (Escape)
^G	Control-G (Bell = Klingel)
^L	Control-L (Formular-Vorschub)
^[	Control- (Escape).

Sequenzen von Kontrollzeichen können zu Strings verkettet werden, indem sie *ohne Trennungszeichen* zwischen den einzelnen Zeichen geschrieben werden:

```
#13#10
#27^U20
^G^G^G^G
```

Die obigen Strings enthalten zwei, drei und vier Zeichen. Kontrollzeichen können auch mit Textstrings gemischt sein:

```
'Waiting for input ! '^G^G^G' Please wake up'
#27'U'
'This is another line of text '^M^J
```

Diese Strings enthalten jeweils 37, 3 und 31 Zeichen.

4.4 Kommentare

Kommentare können überall im Programm eingefügt werden, wo ein Begrenzer erlaubt ist. Ein Kommentar ist von geschweiften Klammern | und | begrenzt, die durch die Symbole (* und *) ersetzt werden können.

Beispiele:
```
|Dies ist ein Kommentar|
(* Dies ist ebenfalls ein Kommentar *)
```

Geschweifte Klammern dürfen nicht geschachtelt werden, (*..*) ebenfalls nicht. Es dürfen aber geschweifte Klammern in (*..*) geschachtelt sein und umgekehrt. Das Äquivalent für gescheifte Klammern ist beim deutschem Zeichensatz 'ä' bzw. 'ü'. Das ermöglicht es Ihnen, ganze Programmteile, auch wenn Sie Kommentare enthalten, als Kommentare zu setzen, d.h. Sie von der Ausführung auszuschließen.

4.5 Compilerbefehle

Eine Reihe der Features des TURBO Pascal Compilers wird durch Compilerbefehle gesteuert. Ein Compilerbefehl wird als Kommentar mit einer speziellen Syntax eingeführt. Das bedeutet, daß überall da, wo ein Kommentar erlaubt ist, auch ein Compilerbefehl stehen kann.

Ein Compilerbefehl besteht aus einer offenen, geschweiften Klammer oder (*, gefolgt von einem Dollarzeichen, unmittelbar darauf folgt ein Compilerbefehls-Buchstabe oder eine Liste von Compilerbefehls-Buchstaben, die durch Kommas getrennt sind. Der Compilerbefehl wird durch eine schließende, geschweifte Klammer bzw. *) abgeschlossen. Die Syntax des Befehls oder der Liste von Befehlen ist unterschiedlich. Sie ist in den entsprechenden Kapiteln beschrieben; eine Zusammenfassung der Compilerbefehle findet sich in Anhang C. Include-Dateien werden in Kapitel 17 behandelt.

Beispiele:
```
|$I|
|$I INCLUDE.FIL|
|$R-,B+,V-|
(*$X-*)
```

Beachten Sie, daß vor und nach dem Dollarzeichen keine Leerzeichen erlaubt sind.

5. PROGRAMMKOPF UND PROGRAMMBLOCK

Ein Pascal Programm besteht aus einem Programmkopf, gefolgt von einem Programmblock. Der Programmblock ist weiter unterteilt in einen Deklarierungsteil, in dem alle im Programm vorkommenden Objekte definiert werden, und einem Ausführungsteil, in dem die Aktionen spezifiziert werden, die mit diesen Objekten ausgeführt werden sollen. Beide werden im Folgenden genau beschrieben.

5.1 Programmkopf

Bei TURBO Pascal ist der Programmkopf nur optional und hat keine Bedeutung für das Programm. Wenn vorhanden, gibt er dem Programm einen Namen und listet wahlweise die Parameter auf, durch die das Programm mit der Umgebung kommuniziert. Die Liste besteht aus einer Reihe von Bezeichnern, die in Klammern stehen und mit Kommas getrennt sind.

Beispiele:
```
program Circles;
program Accountant(Input,Output);
program Writer(Input,Printer);
```

5.2 Deklarierungsteil

Der Deklarierungsteil eines Blocks deklariert alle Bezeichner, die im Anweisungsteil dieses Blocks (und möglicherweise anderer Blöcke innerhalb von diesem) benutzt werden. Der Deklarierungsteil ist in fünf unterschiedliche Abschnitte eingeteilt:

1) Label-Deklarierungsteil
2) Konstanten-Definitonsteil
3) Typen-Definitonsteil
4) Variablen-Deklarierungsteil
5) Prozeduren- / Funktionen-Deklarierungsteil

Während Standard Pascal vorschreibt, daß jeder Abschnitt entweder garnicht oder einmal vorkommen darf und nur in der obigen Reihenfolge, erlaubt TURBO Pascal, daß jeder dieser Abschnitte beliebig oft und in jeder Reihenfolge im Deklarierungsteil auftreten darf.

5.2.1 Label-Deklarierungsteil

Jede Anweisung eines Programms kann mit einem vorangestellten **label** versehen werden, was es ermöglicht, mittels einer **goto** Anweisung direkt zu dieser Anweisung zu verzweigen. Ein Label besteht aus einem Labelnamen, dem ein Komma folgt. Vor Gebrauch muß es in einem Label-Deklarierungsteil deklariert werden. Das reservierte Wort **label** steht am Anfang dieses Teils, es folgt eine Liste der Labelbezeichner, die mit Kommas untereinander getrennt sind und von einem Semikolon abgeschlossen werden.

Beispiel:

```
label  10,    Fehler,    999,    Abbruch;
```

Während Standard Pascal die Label auf Zahlen mit höchstens vier Stellen einschränkt, erlaubt TURBO Pascal, sowohl Zahlen als auch Bezeichner als Label zu verwenden.

5.2.2 Konstanten-Definitionsteil

Der Konstanten-Definitionsteil führt Bezeichner als Synonyme für die Konstantenwerte ein. Das reservierte Wort **const** steht am Anfang des Konstanten-Definitionsteils, es folgt eine Liste der Konstantenzuweisungen, die durch Semikolons getrennt sind. Jede Konstantenzuweisung besteht aus einem Bezeichner, auf den ein Gleicheitszeichen und eine Konstante folgt. Konstanten sind entweder Strings oder Zahlen, die, wie auf den Seiten 43 und 44 beschrieben, definiert sind.

Beispiel:

```
const
Limit = 255;
Max = 1024;
Passwort = 'SESAM';
CursHome = ^|'V';
```

Die folgenden Konstanten sind in TURBO Pascal vordefiniert, d.h. auf sie kann ohne vorherige Definition Bezug genommen werden:

Name:	**Typ und Wert:**
Pi	Real (3.1415926536E+00)
False	Boolean (der Wahrheitswert falsch)
True	Boolean (der Wahrheitswert wahr)
Maxint	Integer (32767)

Wie in Kapitel 13 beschrieben, kann ein Konstanten-Definitionsteil auch typisierte Konstanten definieren.

5.2.3 Typen-Definitionsteil

Ein Datentyp kann in Pascal entweder direkt in dem Variablen-Deklarierungsteil beschrieben sein, oder es kann durch einen Typbezeichner auf ihn Bezug genommen werden. Es stehen mehrere Standarddatentypen zur Verfügung; weiterhin kann ein Programmierer durch die Verwendung der Typendefinition eigene Datentypen erzeugen. Das reservierte Wort **type** steht am Anfang des Typen-Definitionsteils, es folgen eine oder mehrere Zuweisungen, die durch Semikolons getrennt werden. Jede Typzuweisung besteht aus einem Typbezeichner, auf den ein Gleichheitszeichen und ein Typ folgt.

Beispiel:

```
type
Number = Integer;
Day = (mon, tues, wed, thur, fri, sat, sun);
List = array [1..10] of Real;
```

Weitere Beispiele für Typendefinitionen finden sich in den folgenden Abschnitten.

5.2.4 Variablen-Deklarierungsteil

Jede Variable, die in einem Programm auftaucht, muß vor ihrer Verwendung deklariert werden. Die Deklarierung muß textlich einer Verwendung der Variablen vorausgehen, d.h. die Variable muß dem Compiler bekannt sein, bevor sie benutzt werden kann.

Eine Variablendeklarierung besteht aus dem reservierten Wort **var**, darauf folgt(en) ein oder mehrere Bezeichner, die durch Kommas getrennt sind und dann jeweils ein Doppelpunkt und eine **type** Angabe .

Der Geltungsbereich dieser Bezeichner ist der Block, in dem sie definiert sind, und jeder weitere Block innerhalb dieses Blocks. Beachten Sie, daß jeder Block innerhalb eines anderen Blocks, eine andere Variable definieren kann, die denselben Bezeichner verwendet. Diese Variable wird als lokal zu dem Block bezeichnet, in dem sie definiert ist (und in jedem weiteren Block innerhalb dieses Blocks) . Die Variable, die auf dem äußeren Level deklariert wurde (die globale Variable), wird unzugänglich.

Beispiel:

```
var
  Result, Intermediate, SubTotal: Real;
  I, J, X, Y: Integer;
  Accepted, Valid: Boolean;
  Period : Day;
  Buffer: array [0..127] of Byte;
```

5.2.5 Prozedur- und Funktions-Deklarierungsteil

Eine Prozedurdeklarierung dient dazu, eine Prozedur innerhalb einer gegenwärtigen Prozedur oder eines Programms zu definieren (siehe Seite 131). Eine Prozedur wird von einer Prozedur-Anweisung aktiviert (siehe Seite 56). Nach Abschluß der Prozedur, geht die Programmausführung mit der Anweisung weiter, die unmittelbar auf die aufrufende Anweisung folgt.

Eine Funktionsdeklarierung dient dazu, einen Programmteil zu definieren, der einen Wert berechnet und ausgibt (siehe Seite 137). Eine Funktion wird aktiviert, wenn ihr Bezeichner (engl: designator) als Teil eines Ausdrucks angetroffen wird (siehe Seite 54).

5.3 Anweisungsteil

Der Anweisungsteil ist der letzte Teil eines Blocks. Er spezifiziert die vom Programm auszuführenden Aktionen. Der Anweisungsteil hat die Form einer zusammengesetzten Anweisung, der ein Absatz oder ein Semikolon folgt. Eine zusammengesetzte Anweisung besteht aus dem reservierten Wort **begin**, es folgt eine Liste von Anweisungen, getrennt durch Semikolons, und wird durch das reservierte Wort **end** abgeschlossen.

6. AUSDRÜCKE

Ausdrücke (engl: expressions) sind algorithmische Konstrukte, die Regeln für die Berechnung von Werten angeben. Sie bestehen aus Operanden, d.h. Variablen, Konstanten und Funktionsbezeichnern, die mittels Operatoren kombiniert werden.

Dieser Abschnitt beschreibt, wie Ausdrücke aus den skalaren Standardtypen *Integer*, *Real*, *Boolean* und *Char* gebildet werden. Ausdrücke, die die deklarierten, skalaren Typen, **String**-Typen und **Set**-Typen enthalten, werden auf den Seiten 63, 67 und 86 in dieser Reihenfolge beschrieben.

6.1 Operatoren

Operatoren fallen in fünf Kategorien, die hier nach ihrer Priorität geordnet sind:

1) Monadisches Minus (Minus mit nur einem Operanden).
2) **Not** Operator.
3) Multiplikations-Operatoren: *, /, **div, mod, and, shl** und **shr**.
4) Additions-Operatoren: +, -, **or**, und **xor**.
5) Relationale Operatoren: =, <>, <, >, <=, >= und **in**.

Folgen von Operatoren derselben Priorität werden von links nach rechts berechnet. Ausdrücke in Klammern werden zuerst berechnet, unabhängig von vorausgehenden oder nachfolgenden Operatoren.

Wenn beide Operanden eines Multiplikations- oder Additionsoperators vom Typ *integer* sind, dann ist das Ergebnis ebenfalls integer. Wenn einer (oder beide) der Operanden vom Typ *Real* ist, ist auch das Ergebnis vom Typ *Real*.

6.1.1 Monadisches Minus

Das monadische Minus bezeichnet eine Negation seines Operanden, dieser kann vom Typ *Real* oder *Integer* sein.

6.1.2 Not-Operator

Der **not** Operator negiert (kehrt um) den logischen Wert seines bool'schen Operanden:

not True = false
not False = true
TURBO Pascal erlaubt auch die Anwendung des **not** Operators auch auf einen *Integer* Operanden, in diesem Fall findet eine bit-weise Negation statt:

Beispiele:
not 0 = - 1
not -15 = 14
not $2345 = $DCBA

6.1.3 Multiplikations-Operatoren

Operator	Wirkung	Typ des Operanden	Ergebnistyp
*	Multiplikation	Real	Real
*	Multiplikation	Integer	Integer
*	Multiplikation	Real, Integer	Real
/	Division	Real, Integer	Real
/	Division	Integer	Real
/	Division	Real	Real
div	Integer Division	Integer	Integer
mod	Modulus	Integer	Integer
and	arithm. und	Integer	Integer
and	logisches und	Boolean	Boolean
shl	verschieben links	Integer	Integer
shr	verschieben rechts	Integer	Integer

Beispiele:
12 * 34 =408t0
123 / 4 =30.75
123 **div** 4 =30
12 **mod** 5 =2
True **and** False =False
12 **and** 22 =4
2 **shl** 7 =256
256 **shr** 7 =2

6.1.4 Additions-Operatoren

Operator	Wirkung	Typ des Operanden	Ergebnistyp
+	Addition	Real	Real
+	Addition	Integer	Integer
+	Addition	Real, Integer	Real
-	Subtraktion	Real	Real
-	Subtraktion	Real, Integer	Real
-	Subtraktion	Integer	Integer
or	arithm. oder	Integer	Integer
or	logisches oder	Boolean	Boolean
xor	arithm. excl.-oder	Integer	Integer
xor	logisches excl.-oder	Boolean	Boolean

Beispiele:

123+456	=579
456-123.0	=333.0
True **or** False	=Wahr
12 **or** 22	=30
True **xor** False	=True
12 **xor** 22	=26

Relationale Operatoren

Relationale Operatoren gelten für alle skalaren Standardtypen: *Integer*, *Real*, *Boolean*, *Char* und *Byte*. Operanden der Typen *Integer*, *Real* und *Byte* können gemischt werden. Der Ergebnistyp ist immer *Boolean*, d.h. *True* oder *False* (wahr oder unwahr).

=	ist gleich
<>	ungleich
>	größer als
<	kleiner als
>=	größer gleich
<=	kleiner gleich

Beispiele:

a = b	ist wahr, falls a gleich b
a <> b	ist wahr, falls a ungleich b
a > b	ist wahr, falls a größer b
a < b	ist wahr, falls a kleiner b
a >= b	ist wahr, falls a größer gleich b
a <= b	ist wahr, falls a kleiner gleich b

6.2 Funktionsbezeichnung

Die Funktionsbezeichnung (engl:function designator) ist ein Funktionsbezeichner, dem wahlweise eine Parameterliste folgt, die eine oder mehrere Variable oder Ausdrücke, die durch Kommas getrennt sind und von Klammern umschlossen werden, enthält. Das Auftreten einer Funktionsbezeichnung aktiviert die Funktion mit diesem Namen. Wenn die Funktion keine vordefinierte Standardfunktion ist, muß sie vor der Aktivierung erst deklariert werden.

Beispiele:

```
Round(PlotPos)
Writeln(Pi * (Sqr(R)))
(Max(X,Y) < 25) and (Z > Sqrt(X * Y))
Volume(Radius,Height)
```

7. Anweisungen

Der Ausführungsteil definiert die Aktion, die vom Programm (oder Unterprogramm) ausgeführt werden soll, als Folge von Anweisungen (engl: statements). Jede Anweisung spezifiziert einen Teil der Aktion. In diesem Sinne ist Pascal eine sequentielle Programmsprache: Anweisungen werden zeitlich sequentiell abgearbeitet, nie zugleich. Der Anweisungsteil ist umschlossen von den reservierten Wörtern **begin** und **end**, innerhalb dieser sind die Anweisungen durch Semikolons getrennt. Anweisungen können *einfach* oder *strukturiert* sein.

7.1 Einfache Anweisungen

Einfache Anweisungen enthalten keine anderen Anweisungen. Einfache Anweisungen sind die Zuweisungs-, Prozedur-, **goto**- und die leere Anweisung.

7.1.1 Zuweisungs-Anweisung

Die grundlegendste aller Anweisungen ist die Zuweisungs-Anweisung. Sie wird verwendet, um anzugeben, daß ein bestimmter Wert einer bestimmten Variablen zugewiesen werden soll. Eine Zuweisung besteht aus einem Variablenbezeichner, dem ein Zuweisungsoperator := und ein Ausdruck folgt.

Zuweisungen sind zu Variablen beliebigen Typs (außer Dateien) möglich, solange die Variable und der Ausdruck vom selben Typ sind. Als Ausnahme davon kann bei einer *Real* Variablen der Ausdruck *Integer* sein.

Beispiele:

```
Angle := Angle * Pi;
AccessOK := False;
Entry := Answer = PassWord;
SpherVol := 4 * Pi * R * R;
```

7.1.2 Prozedur-Anweisung

Die Prozedur-Anweisung dient dazu, eine zuvor vom Benutzer definierte oder eine vordefinierte Standardprozedur zu aktivieren. Die Anweisung besteht aus einem Prozedurbezeichner, wahlweise gefolgt von einer Parameterliste. Diese Parameterliste ist eine Liste von Variablen oder Ausdrücken, die durch Kommas getrennt und in Klammern eingeschlossen sind. Wenn bei der Ausführung des Programms die Prozedur-Anweisung erreicht wird, wird die Kontrolle auf die bezeichnete Prozedur übertragen, die Werte möglicher Parameter werden ebenfalls auf die Prozedur übertragen. Wenn die Prozedur beendet ist, geht die Programm-
ausführung mit der Anweisung weiter, die auf die Prozedur-Anweisung folgt.

Beispiele:
```
Find(Name,Adresse);
Sort(Adresse);
UpperCase(Text);
UpdateCustFile(CustRecord);
```

7.1.3 Goto-Anweisung

Eine **goto** Anweisung besteht aus dem reservierten Wort **goto**, auf das ein Labelbezeichner folgt. Sie dient dazu, die weitere Verarbeitung an die Stelle im Programmtext zu übergeben, die durch das Label markiert ist. Die folgenden Regeln sollten bei der Verwendung von **goto** Anweisungen beachtet werden.

1) Vor Gebrauch müssen die Labels deklariert werden. Die Deklaration geschieht in einem Label-Deklarierungsteil des Blocks, in dem dieses verwendet wird.
2) Die Reichweite des Labels ist der Block, in dem es deklariert wurde. Deshalb ist es nicht möglich, in oder aus Prozeduren und Funktionen zu springen.

7.1.4 Leere Anweisung

Eine leere Anweisung besteht aus keinen Symbolen und hat keine Wirkung. Sie darf vorkommen, wo immer die Syntax von Pascal eine Anweisung verlangt, aber keine Aktion stattfinden soll.

Beispiele:
```
begin end.
while Answer <> '' do;
repeat until Keypressed; {wait for any key to be hit}
```

7.2 Strukturierte Anweisungen

Strukturierte Anweisungen sind Konstrukte, die aus anderen Anweisungen zusammengesetzt sind. Diese werden sequentiell (zusammengesetzte Anweisung), bedingt (bedingte Anweisung) oder wiederholt (wiederholende Anweisung) ausgeführt. Die Diskussion der **with** Anweisungen wird auf Abschnitt 11.2 verschoben.

7.2.1 Zusammengesetzte Anweisung

Eine zusammengesetzte Anweisung (engl: compound statement) wird benutzt, wenn in einer Situation mehr als eine Anweisung ausgeführt werden soll, in der die Pascal Syntax nur die Spezifikation einer Anweisungen erlaubt. Es besteht aus einer beliebigen Zahl von Anweisungen, die mit Semikolons getrennt sind und von den reservierten Wörtern **begin** und **end** eingeschlossen werden. Die einzelnen Anweisungen der zusammengesetzten Anweisung werden in der Abfolge, in der sie geschrieben sind, ausgeführt.

Beispiel:

```
if Small > Big then
begin
  Tmp := Small;
  Small := Big;
  Big := Tmp;
end;
```

7.2.2 Bedingte Anweisung

Ein bedingte Anweisung (engl: conditional statement) wählt eine ihrer Teilanweisungen zur Ausführung aus.

7.2.2.1 If-Anweisung

Die **if** Anweisung (Entscheidung) spezifiziert, daß eine Anweisung nur dann ausgeführt wird, wenn eine bestimmte Bedingung (Boolean Ausdruck) erfüllt (wahr) ist. Wenn sie nicht erfüllt (falsch) ist, dann wird entweder keine Anweisung, oder die Anweisung, die auf das reservierte Wort **else** folgt, ausgeführt. Beachten Sie, daß **else** kein Semikolon vorangehen darf.

Die syntaktische Zweideutigkeit, die aus folgendem Konstrukt entsteht:

```
if expr1 then
if expr2 then
  stmt1
  else
    stmt2
```

wird beseitigt, indem das Konstrukt folgendermaßen interpretiert wird:

```
if 2 expr1 then
begin
  if expr2 then
    stmt1
  else
    stmt2
end
```

d.h., der **else** Klauselteil gehört generell zur letzten **if** Anweisung, die keinen **else** Teil hat.

Beispiele:

```
if Interest ) 25 then
  Usury := True
else
  TakeLoan := OK;

If (Entry ( 0=) or (Entry ) 100) then
begin
  Write('Range is 1 to 100, please, re-enter: ');
  Read(Entry);
end;
```

7.2.2.2 Case-Anweisung

Die **Case** Anweisung (Auswahl) besteht aus einem Ausdruck (dem Sortierer) und einer Liste von Anweisungen, denen jeweils Case Label vom Typ des Sortierers vorausgehen. Sie gibt an, daß die Anweisung, deren Label dem aktuellen Wert des Sortierers entspricht, ausgeführt werden soll. Wenn kein Case Label den Wert des Sortierers enthält, dann werden entweder keine oder wahlweise die Anweisungen, die dem reservierten Wort **else** folgen, ausgeführt. Die **else** Klausel ist eine Erweiterung von Standard Pascal.

Ein Case Label besteht aus einer beliebigen Zahl von Konstanten oder Teilbereichen, die durch Kommas getrennt sind und denen ein Semikolon folgt. Ein Teilbereich wird als zwei Konstanten geschrieben, die von dem Teilbereichs-Begrenzer '..' getrennt werden. Der Typ der Konstanten muß gleich dem Typ des Sortierers sein. Die Anweisung, die dem Case Label folgt, wird ausgeführt, wenn der Wert des Sortierers gleich einer der Konstanten ist, oder in einem der Teilbereiche liegt.

Gültige Sortierer-Typen sind alle einfachen Typen, d.h. alle skalaren Typen außer reellen Zahlen.

Beispiele:

```
case Operator of
  '+' : Result := Answer + Result;
  '-' : Result := Answer - Result;
  '*' : Result := Answer * Result;
  '/' : Result := Answer / Result;
end;

case Year of
  Min..1939: begin
      Time := PreWorldWar2;
      Writeln('The world at peace..');
      end;
  1946..Max: begin
      Time := PostWorldWar2;
      Writeln('Building a new world');
      end;
  else
  Time := WorldWar2;
  Writeln('We are at war');
end;
```

7.2.3 Wiederholende Anweisungen

Wiederholende Anweisungen (engl: repetitive statements) geben an, daß bestimmte Anweisungen wiederholt ausgeführt werden sollen. Wenn die Zahl der Wiederholungen in vorhinein unbekannt ist, d.h. bevor die Wiederholungen beginnen, ist die **for** Anweisung das geeignete Konstrukt, diese Situation auszudrücken. Andernfalls sollte die **while** oder **repeat** Anweisung verwendet werden.

7.2.3.1 For-Anweisung

Die **for**-Anweisung (Laufanweisung oder Zählschleife) zeigt an, daß die Teilanweisung wiederholt ausgeführt werden soll. Die ansteigenden Werte werden einer Variablen zugewiesen , die *Kontrollvariable* genannt wird. Die Werte können aufsteigend: **to**, oder absteigend: **downto** bis zu dem endgültigen Wert sein.

Die Kontrollvariable, der anfängliche Wert und der endgültige Wert müssen alle vom selben Typ sein. Gültige Typen sind alle einfachen Typen, d.h. alle skalaren Typen außer *Real*. Wenn bei Verwendung der **to** Klausel der anfängliche Wert größer, als der endgültige Wert ist, oder bei Verwendung der **downto** Klausel der anfängliche Wert kleiner, als der endgültige Wert ist, wird der Anweisungsteil nicht ausgeführt.

Beispiele:

```
for I := 2 to 100 do if A[I] > Max then Max := A[I];
for I := 1 to NoOfLines do
begin
  Readln(Line);
  if Length(Line) < Limit then ShortLines := Shortlines + 1
  else
    Longlines := Longlines + 1
end;
```

Beachten Sie, daß eine Teilanweisung einer **for** Anweisung keine Zuweisungen zur Kontrollvariable enthalten darf. Wenn die Wiederholung beendet werden soll, bevor der endgültige Wert erreicht ist, muß eine **goto** Anweisung verwendet werden, obwohl solche Konstrukte nicht empfohlen werden - stattdessen ist es bessere Programmierpraxis, hier eine **while** oder **repeat** Anweisung zu verwenden.

Nach Beendigung einer **for** Anweisung, ist die Kontrollvariable gleich dem endgültigen Wert, außer, die Schleife wurde nicht ausgeführt. In einem solchen Fall würde keine Zuweisung zur Kontrollvariablen erfolgen.

7.2.3.2 While-Anweisung

Der Ausdruck, der die Wiederholung kontrolliert, muß vom Typ *boolean* sein. Die Anweisung wird so lange wiederholt, so lange der Ausdruck *true* (wahr) ist. Ist der Wert schon zu Beginn *false* (falsch), wird die Anweisung überhaupt nicht ausgeführt.

Beispiele:

```
while Size ) 1 do Size := Sqrt(Size);

while ThisMonth do
begin
  ThisMonth := CurMonth = SampleMonth;
  Process;
  {bearbeite dieses Beispiel mit der Process Prozedur}
end;
```

7.2.3.3 Repeat-Anweisung

Der Ausdruck, der die Wiederholung kontrolliert, muß vom Typ *boolean* sein. Die Sequenz der Anweisungen zwischen den reservierten Wörtern **repeat** und **until** wird so oft wiederholt, bis der Ausdruck wahr wird. Im Unterschied zur **while** Anweisung wird die **repeat** Anweisung immer mindestens einmal ausgeführt, da erst am Ende der Schleife die Abbruchbedingung abgefragt wird.

Beispiel:

```
repeat
  Write(^M,'Delete this item? (Y/N)');
  Read(Answer);
until UpCase(Answer) in ['Y','N'];
```

Anmerkungen:

8. Skalare Datentypen und deren Teilbereiche

Der skalare Datentyp ist bei Pascal der grundlegende Datentyp. Er bildet eine endliche, linear angeordnete Reihe von Werten. Obwohl der Standarddatentyp *Real* auch als skalarer Datentyp betrachtet wird, fällt er nicht unter diese Definition. Deshalb können *real* Zahlen nicht immer im gleichen Zusammenhang wie andere skalare Datentypen verwendet werden.

8.1 Skalare Datentypen

Neben den skalaren Standarddatentypen (*Integer, Real, Boolean, Char* und *Bytes*) unterstützt Pascal auch vom Benutzer definierte skalare Datentypen (deklarierte skalare Datentypen). Die Definition eines skalaren Datentypen gibt in linearer Ordnung all seine möglichen Werte an. Die Werte des neuen Datentyps werden durch Bezeichner repräsentiert, die deren Konstanten sind.

Beispiele:

```
type
  Operator = (Plus,Minus,Multi,Divide);
  Day      = (Mon,Tues,Wed,Thur,Fri,Sat,Sun);
  Month    = (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec);
  Card     = (Club,Diamond,Heart,Spade);
```

Variablen des Datentyps *Card* können einen der vier oben angegebenen Werte annehmen. Mit dem skalaren Standarddatentyp *Boolean* sind Sie schon vertraut. Er wird folgendermaßen definiert:

```
type
  Boolean = (False,True);
```

Die Operatoren =, <>, <, >, <= und >= können allen skalaren Datentypen beigeordnet werden, unter der Bedingung, daß beide Operanden vom gleichen Datentyp sind (als Ausnahme können *real* und *integer* Zahlen gemischt werden). Die Operanden werden in der Reihenfolge ihres Auftretens verglichen, so gilt für den Typen *Card* aus obigen Beispiel:

```
Club < Diamond < Heart < Spade
```

Drei Standardfunktionen sind vorhanden, die mit den skalaren Datentypen arbeiten:

Succ(Diamond)	der Nachfolger (successor) von *Diamond* (=Heart)
Pred(Diamond)	der Vorgänger (predecessor) von *Diamond* (=Club)
Ord(Diamond)	die Platznummer (ordinal number) von *Diamond* (=1)

Das erste Element eines Datentyps hat immer die Ordnungszahl, bzw. Platznummer 0. Der Datentyp des Ergebnisses von *Succ* und *Pred* entspricht dem Datentyp des jeweiligen Arguments, von *Ord* ist es immer eine *integer* Zahl.

8.2 Teilbereiche skalarer Datentypen

Ein Datentyp kann auch als Teilbereich eines bereits definierten skalaren Typs definiert werden. Diese werden als Teilbereiche skalarer Datentypen bezeichnet. Die Definition bestimmt lediglich den niedrigsten und den höchsten Wert dieses Teilbereiches. Die erste Konstante bestimmt die untere Grenze und darf nicht größer als die zweite sein, die die obere Grenze bildet. Ein Teilbereich des Typs *Real* ist nicht erlaubt.

Beispiele:

```
type
  HemiSphere    = (North,South,West,East);
  World         = (East,West);
  CompassRange  = 0..360;
  Upper         = 'A'..'Z';
  Lower         = 'a'..'z';
  Degree        = (Celc,Fahr,Ream,Kelv);
  Wine          = (Red,White,Rose,Sparkling);
```

Der Datentyp *World* ist ein Teilbereich des skalaren Datentyps *Hemisphere*. Der Teilbereich von *CompassRange* ist *integer*, und der dazugehörige skalare Typ von *Upper* und *Lower* ist *Char*.

Sie kennen bereits den standardisierten Teilbereich des Datentyps *Byte*, der wie folgt definiert ist:

```
type
  Byte = 0..255;
```

Ein Teilbereich besitzt alle Attribute seines zugeordneten skalaren Datentyps und ist lediglich durch die Menge seiner möglichen Werte begrenzt.

Sie sollten die definierten, skalaren Datentypen und deren Teilbereiche ausgiebig nutzen, da diese die Lesbarkeit von Programmen wesentlich verbessern. Außerdem sind Laufzeit-Prüfungen im Programm enthalten (siehe Seite 65), um die Werte zu überprüfen, die den definierten, skalaren Variablen und deren Teilbereichen zugeordnet sind. Ein weiterer Vorteil liegt in der Minimierung des Speicherplatzbedarfes. TURBO Pascal benötigt für benutzerdefinierte, skalare Datentypen, bzw. deren Teilbereiche, lediglich 1 Byte Speicherplatz, wenn die Gesamtzahl ihrer Elemente 256 nicht übersteigt. Desgleichen besetzen Variable eines Teilbereiches vom Datentyp Integer, Ober- und Untergrenze im Bereich von 0..255, auch nur 1 Byte im Speicher.

8.3 Umwandlung von Datentypen

Mit der Funktion *Ord* können skalare Datentypen dem Wert einer *integer* Zahl zugeordnet werden. Standard Pascal unterstützt diese Umwandlung nicht in die umgekehrte Richtung, d.h. eine *integer* Zahl läßt sich nicht in einen skalaren Wert umwandeln.

Bei TURBO Pascal kann der Wert eines skalaren Datentypen in den Wert eines anderen mit *Retype* umgewandelt werden. Dazu geben Sie den Bezeichner des gewünschten Datentyps ein. Daran anschließend wird ein Parameter in Klammern gesetzt. Der Parameter kann den Wert jedes skalaren Datentyps annehmen, ausgeschlossen sind lediglich *real* Zahlen:

```
Integer(Heart)      = 2
Month(10)           = Nov
HemiSphere(2)       = East
Upper(14)           = 'O'
Degree(3)           = Kelv
Char(78)            = 'N'
Integer('7')        = 55
```

8.4 Überprüfung der Variablengröße

Die Erzeugung von Code, der Überprüfungen der Wertebereiche von Variabeln zuläßt, wird mit dem Compilerbefehl **R** kontrolliert. Die Voreinstellung ist |$R-| d.h. daß keine Prüfungen stattfinden. Wenn einer skalaren Variablen oder einem ihrer Teilbereiche ein Wert zugewiesen wird, wird dieser auf seine Größe geprüft, solange der Compilerbefehl aktiv ist (| $R+ |). Es wird empfohlen, diese Option solange aktiv zu lassen, solange noch Fehler im Programm sind.

Beispiel:

```
program RangeCheck;
type
  Digit = 0..9;
Var
  Dig1,Dig2,Dig3: digit;
begin
  Dig1 := 5;                    {gültig}
  Dig2 := Dig1 + 3;             {gültig, da Dig 1 + 3 ( = 9}
  Dig3 := 47;                   {ungültig, aber ohne Fehlermeldung}
  {$R+}Dig3 := 55;              {ungültig, ergibt einen Laufzeit-Fehler}
  {$R-}Dig3 := 167;             {ungültig,aber ohne Fehlermeldung}
end.
```

9. Strings

TURBO Pascal bietet Ihnen den Datentyp **String**, um Zeichenketten zu verarbeiten. Zeichenketten sind eine Aneinanderreihung von Zeichen. Der Datentyp String ist strukturiert und in vielem dem **Array** (Abschnitt 10) sehr ähnlich. Es gibt allerdings einen großen Unterschied: die Anzahl der Zeichen in einem String, bzw. dessen Länge, kann dynamisch zwischen 0 und einer oberen Grenze variieren, während die Anzahl der Elemente in einem Array feststeht.

9.1 Definition des Strings

Die Definition des Datentyps String muß die obere Grenze der Anzahl der enthaltenen Zeichen, d.h. die Maximallänge angeben. Die Definition besteht aus dem reservierten Wort **string**, dem in eckigen Klammern die Maxixmallänge folgt. Diese muß eine *integer* Konstante zwischen 1 und 255 sein. Strings haben keine voreingestellte Länge, d.h. sie muß immer genau bestimmt werden.

Beispiel:

```
type
  FileName = string[14];
  ScreenLine = string[80];
```

Stringvariablen besetzen den Speicher in der definierten Maximallänge und zusätzlich ein Byte für die aktuelle Länge der Variablen. Die einzelnen Zeichen eines Strings sind, mit 1 beginnend, über die gesamte Länge durchnummeriert.

9.2 Stringausdruck

Strings werden mittels Stringausdrücken bearbeitet. Diese bestehen aus Stringkonstanten, Stringvariablen, Funktionsbezeichnern und Operatoren.

Das Pluszeichen kann Strings verbinden. Die Funktion *Concat* (siehe Seite 71) macht das Gleiche, aber der Operator '+' ist oft einfacher zu handhaben. Sollte die Länge des entstehenden Strings größer als 255 sein, wird eine Laufzeit-Fehlermeldung ausgegeben.

Beispiel:

'TURBO' + 'Pascal'	= 'TURBO Pascal'
'123' + '.' + '456'	= '123.456'
'A' + 'B' + 'C' + 'D'	= 'ABCD'

Die relationalen Operatoren (=, <>, >, <, >=, <=) haben eine geringere Präferenz als der Verbindungsoperator. Wenn die relationalen Operatoren bei Stringoperanden angewendet werden, ist das Ergebnis vom Typ *Boolean* (*True* oder *False*). Beim Vergleich zweier Strings werden die einzelnen Buchstaben von links nach rechts miteinander verglichen. Wenn die Strings verschiedene Länge haben, und der kürzere, bis hin zum letzten Buchstaben, den am Anfang stehenden Zeichen des längeren Strings entspricht, dann wird der kürzere als der kleinere erkannt. Strings sind nur dann gleich, wenn sie sich sowohl im Inhalt als auch in der Länge entsprechen.

Beispiele:

'A' < 'B'	wahr
'A' > 'B'	falsch
'2' < '12'	falsch
'TURBO' = 'TURBO'	wahr
'TURBO ' = 'TURBO'	falsch
'Pascal Compiler' < 'Pascal compiler'	wahr

9.3 Stringzuordnung

Der Zuordnungsoperator weist den Wert eines Stringausdruckes einer Stringvariablen zu.

Beispiel:

```
Age := 'fiftieth';
Line := 'Many happy returns on your ' + Age + ' birthday.';
```

Wenn die angegebene Maximallänge der Stringvariablen überschritten wird, werden die überzähligen Buchstaben verschluckt. Das heißt, wenn die obige Variable mit **string**[5] deklariert wurde, wird die Variable nach der Zuweisung nur die fünf Buchstaben, die links stehen, enthalten: 'fifti'.

9.4 Stringprozeduren

Die folgenden Standardstringprozeduren sind in TURBO Pascal verfügbar:

9.4.1 Löschen

Syntax: Delete (*St, Pos, Num*)

Delete löscht aus einer Stringvariablen (*St*) eine bestimmte Anzahl (*Num*) von Buchstaben, beginnend bei der Position *Pos*. *Pos* und *Num* sind *integer* Ausdrücke. Wenn *Pos* größer als die Länge von *St* ist, wird kein Buchstabe gelöscht. Wenn versucht wird, Buchstaben zu löschen, die sich jenseits des rechten Endes des Strings befinden, d.h. *Pos* und *Num* sind größer als die Länge des Strings, werden nur Buchstaben innerhalb des Strings gelöscht. Wenn *Pos* außerhalb des Bereichs 0..255 ist, wird eine Laufzeit-Fehlermeldung ausgegeben.

Wenn *St* den Wert 'ABCDEFG' hat, dann nimmt *St* unter den folgenden Bedingungen nachstehenden Wert an:

```
Delete(St,2,4)     ergibt den Wert 'AFG'
Delete(St,2,10)    ergibt den Wert 'A'
```

9.4.2 Einfügen

Syntax: Insert (*Obj, Target, Pos*)

Insert fügt den String *Obj* in den String *Target* an der Position *Pos* ein. *Obj* ist ein Stringausdruck, *Target* ist eine Stringvariable und *Pos* ist eine *integer* Zahl. Wenn *Pos* größer als die Länge der Stringvariablen ist, dann wird der Stringausdruck an *Target* angefügt. Wenn das Ergebnis größer als die angegebene Maximallänge von *Target* ist, verschwinden die überzähligen Buchstaben, und *Target* enthält lediglich die am weitesten links stehenden Buchstaben. Ist *Pos* außerhalb des Bereichs 0..255, wird eine Laufzeit-Fehlermeldung ausgegeben.

Wenn *St* den Wert 'ABCDEFG' hat, dann gibt `Insert('XX',St,3)` *St* den Wert 'ABXXCDEFG'

9.4.3 Str

Syntax: Str(*Value,Str*)

Die Prozedur *Str* wandelt den numerischen Wert *Value* in einen String um und speichert das Ergebnis als *St* ab. *Value* ist ein Schreibparameter des Typs *integer* oder *real*, *St* ist eine Stringvariable. Schreibparameter sind Ausdrücke mit speziellen Formatierbefehlen (siehe Seite 111).

Wenn *I* den Wert 1234 hat, gilt:
Str(I:5,St) *St* erhält den Wert ' 1234'

Wenn *X* den Wert 2.5E4 hat, gilt:
Str(X:10:0,St) *St* erhält den Wert ' 2500'

CP/M 80-Benutzer: Eine Funktion, die die *Str* Prozedur benutzt, darf **nie** durch einen Ausdruck in einer *Write* oder *Writeln* Anweisung aufgerufen werden.

9.4.4 Val

Syntax: Val(*St, Var, Code*)

Val wandelt den Stringausdruck *St* in einen *integer* oder *real* Wert um (abhängig vom Datentyp der Variablen *Var*) und speichert diesen Wert in *Var*. *St* muß ein String sein, der einen numerischen Wert ausdrückt, entsprechend den Regeln bei numerischen Konstanten (siehe Seite 43). Weder davor, noch danach sind Leerzeichen erlaubt. *Var* muß eine *integer* oder *real* Variable sein und *Code* eine *integer* Variable sein. Wenn keine Fehler gefunden werden, wird die Variable *Code* auf 0 gesetzt. Ansonsten wird *Code* auf das erste fehlerhafte Zeichen gesetzt und der Wert von *Var* ist undefiniert.

Wenn *St* den Wert '234' hat, gilt:
Val(St,I,Result)
I erhält den Wert '234' und *Result* '0'

Wenn *St* den Wert '12x' hat, gilt:
Val(St,I,Result)
I ist undefiniert und *Result* hat den Wert '3'

Wenn *St* den Wert '2.5E4' hat, und *X* eine *real* Variable ist, gilt:
Val(St,X,Result)
X hat der Wert '2500' und *Result* '0'

CP/M 80-Benutzer: eine Funktion, die die *Var* Prozedur benutzt, darf **nie** durch einen Ausdruck in einer *Write* oder *Writeln* Anweisung aufgerufen werden.

9.5 Stringfunktionen

Folgende Standardstringfunktionen sind in TURBO Pascal anwendbar:

9.5.1 Copy

Syntax: Copy(*St,Pos,Num*)

Copy gibt einen Teilstring eines Strings (*St*) aus, der eine bestimmte Anzahl (*Num*) von Zeichen enthält, gezählt von der Position *Pos*. *St* ist ein Stringausdruck, *Pos* und *Num* sind *integer* Ausdrücke. Wenn der Wert von *Pos* die Länge des Strings übersteigt, wird ein leerer Teilstring ausgegeben. Wenn versucht wird, Zeichen jenseits des Endes des Strings zu erhalten, d.h. der Wert von *Pos* + *Num* übersteigt die Länge des Strings, werden nur die noch innerhalb des Strings befindlichen Zeichen ausgegeben. Wenn *Pos* außerhalb des Bereiches 0..255 ist, erfolgt eine Laufzeit-Fehlermeldung.

Wenn *St* den Wert 'ABCDEFG' hat, gilt:

Copy(St,3,2)	gibt den Wert 'CD' aus
Copy(St,4,10)	gibt den Wert 'DEFG' aus
Copy(St,4,2)	gibt den Wert 'DE' aus

9.5.2 Concat

Syntax: Concat(*St1,St2*[,*StN*])

Die Funktion *Concat* gibt einen Gesamtstring aus, der aus beliebig vielen Einzelstrings in der angegebenen Ordnung (*St1..StN*) zusammengesetzt wird. Ist das Ergebnis größer als 255, wird eine Laufzeit-Fehlermeldung ausgegeben. Wie auf Seite 68 schon besprochen wurde, kann der Operator '+' das Gleiche, u.U. sogar auf einfachere Art und Weise. *Concat* wurde in TURBO Pascal nur aufgenommen, um die Kompatibilität zu anderen Pascalcompilern zu erhalten.

Wenn *St1* den Wert 'TURBO' und *St2* den Wert ' ist am schnellsten' hat, ergibt:

```
Concat(St1,' Pascal',St2)
```

den Wert 'TURBO Pascal ist am schnellsten'.

9.5.3 Length

Syntax: Length(*St*)

Gibt die Länge des Stringausdruckes *St* aus, d.h. die Anzahl der darin enthaltenen Zeichen. Das Ergebnis ist *integer*.

Wenn *St* den Wert '123456789' hat, ergibt:
`Length(St)` den Wert '9'

9.5.4 Pos

Syntax: Pos(*Obj,Target*)

Diese Funktion durchsucht den String *Target* nach dem ersten Vorkommen des Stringausdruckes *Obj*. Das Ergebnis ist *integer* und bezeichnet die Position im Stringausdruck *Target*, die das erste Zeichen von *Obj* innehat. Die Position des ersten Zeichens im String ist '1'. Wird die Zeichenkombination nicht gefunden, liefert *Pos* den Wert '0'.

Wenn *St* den Wert 'ABCDEFG' hat, ergibt:
`Pos('DE',St)` den Wert '4'
`Pos('H',St)` den Wert '0'

9.6 Strings und Zeichen

Der Datentyp **String** und der skalare Standarddatentyp *Char* sind untereinander kompatibel. So kann, wann immer ein Stringwert erwartet wird, auch ein Zeichenwert eingegeben werden und umgekehrt. Außerdem können Strings und Zeichen in Ausdrücken gemischt werden. Wenn ein Zeichen zum Stringwert wird, muß die Länge des Strings als '1' definiert werden, sonst wird eine Laufzeit-Fehlermeldung ausgegeben.

Auf die Zeichen einer Stringvariablen kann mittels Stringindexing zugegriffen werden. Dazu wird der Stringvariablen ein *integer* Indexausdruck in eckigen Klammern angehängt.

Beispiele:
```
Buffer[5]
Line[Length(Line)-1]
Ord(Line[0])
```

Da das erste Zeichen des Strings (bei Index '0') die Länge des Strings enthält, ist `Length(String)` das Gleiche wie `Ord(String[0])`. Wenn der Längenindikator benutzt wird, sollte der Programmierer darauf achten, daß er kürzer als die Maximallänge der Stringvariablen ist. Wenn der Compilerbefehl **R** aktiv ist (`{$R+}`), wird ein Code generiert, der garantiert, daß der Wert eines Stringindexausdrucks nicht die Maximallänge der Stringvariablen übersteigt. Es ist jedoch immer noch möglich einen String-Index zu bilden, der über der aktuellen dynamischen Länge liegt. Die so gelesenen Zeichen sind zufällig, und Zuweisungen jenseits der aktuellen Länge betreffen nicht den momentanen Wert der Stringvariablen.

Anmerkungen:

10. Arrays

Ein Array ist ein strukturierter Datentyp mit einer festgesetzten Anzahl von Komponenten, die alle vom gleichen Typ sind, dem Grundtyp. Auf jede Komponente kann mit Indizes zugegriffen werden. Indizes sind *integer* Ausdrücke, die in eckigen Klammern hinter den Arraybezeichnern stehen. Ihr Datentyp wird Indextyp genannt.

10.1 Arraydefinition

Die Definition eines Arrays besteht aus dem reservierten Wort **array**, dem der Indextyp in eckigen Klammern folgt. Danach steht das reservierte Wort **of**, gefolgt vom Grundtyp.

Beispiele:

```
type
  Day = (Mon,Tue,Wed,Thu,Fri,Sat,Sun);
Var
  WorkHour   : array[1..8] of Integer;
  Week       : array[1..7] of Day;

type
  Players    = (Player1,Player2,Player3,Player4);
  Hand       = (One,Two,Pair,TwoPair,Three,Straight,
                Flush,FullHouse,Four,StraightFlush,RSF);
  LegalBid   = 1..200;
  Bid        = array[Players] of LegalBid;
Var
  Player     : array[Players] of Hand;
  Pot        : Bid;
```

Auf eine Arraykomponente greift man zu, indem ein Index in eckigen Klammern an den Variablenbezeichner des Arrays gehängt wird:

```
Player[Player3] := FullHouse;
Pot[Player3] := 100;
Player[Player4] := Flush;
Pot[Player4] := 50;
```

Da eine Zuweisung zwischen zwei beliebigen Variablen erlaubt ist, solange sie vom gleichen Datentyp sind, können ganze Arrays mit einer einzigen Zuweisungs-Anweisung kopiert werden.

Der Compilerbefehl **R** kontrolliert bei der Codegenerierung, ob die Arrays im zulässigen Bereich liegen. Nach Voreinstellung ist er inaktiv. |R+| verursacht eine Überprüfung aller Indexausdrücke auf die Einhaltung der Grenzen ihres Indextyps.

10.2 Multidimensionale Arrays

Die Komponenten eines Arrays können beliebigen Datentyps sein, d.h. daß die Komponenten auch Arrays sein können. Eine solche Struktur nennt man *multidimensionales* Array.

Beispiel:

```
type
  Card      = (Two,Three,Four,Five,Six,Seven,Eight,Nine,
               Ten,Knight,Queen,King,Ace);
  Suit      = (Hearts,Spade,Clubs,Diamonds);
  AllCards  = array[Suit] of array[1..139] of Card;
Var
  Deck: AllCards;
```

Ein multidimensionaler Array kann auch einfacher definiert werden:

```
type
  AllCards = array[Suit,1..13] of Card;
```

Eine ähnliche Kurzform kann bei der Wahl der Arraykomponenten gewählt werden:

```
Deck[Hearts,10]   entspricht   Deck[Hearts][10]
```

Es ist natürlich auch möglich, multidimensionale Arrays in der Form von vordefinierten Arraytypen zu benutzen.

Beispiel:

```
type
  Pupils    = string[10];
  Class     = array[1..30] of Pupils;
  School    = array[1..100] of Class;
Var
  J,P,Vacant          : Integer;
  ClassA,
  ClassB              : Class;
  NewTownSchool       : School;
```

Nach diesen Definitionen sind alle folgenden Zuweisungen möglich:

```
ClassA[J]:='Peter';
NewTownSchool[5][21]:='Peter Brown';
NewTownSchool[8,J]:=NewTownSchool[7,J];
{Schüler J wechselt Klasse}
ClassA[Vacant]:=ClassB[P];
{Schüler [P] wechselt Klasse und Nr.}
```

Zeichenarrays

Zeichenarrays sind Arrays mit einem Index und Komponenten des skalaren Standarddatentyps *Char*. Zeichenarrays können als Strings mit konstanter Länge gedacht werden.

Bei TURBO Pascal können Zeichenarrays an Stringausdrücken teilnehmen. In diesem Fall wird der Array in einen String der gleichen Länge umgewandelt. So können Arrays auf die gleiche Art und Weise wie Strings verglichen und behandelt, und Stringkonstanten können Zeichenarrays zugewiesen werden, solange sie die gleiche Länge haben. Stringvariable und Werte aus Stringausdrücken können nicht den Zeichenarrays zugewiesen werden.

Vordefinierte Arrays

TURBO Pascal bietet zwei vordefinierte Arrays vom Typ *Byte, Mem* und *Port*, die als direkter Zugang zum CPU-Speicher und zu den Daten-Ports benutzt werden können. Diese werden in den Kapiteln 20, 21 und 22 besprochen.

Anmerkungen:

11. Recordtyp (Satzart)

Ein **record** ist eine Datenstruktur, die aus einer festgelegten Zahl von Komponenten besteht, die Felder genannt werden. Der Zusammenschluß mehrerer Felder als Record wird auch Verbund genannt. Die Felder können aus verschiedenen Datentypen bestehen, und jedes wird mit einem Feldbezeichner (*field identifier*) benannt. Dieser dient der Feldselektion in einem Record.

11.1 Definition des Records

Die Definition des Datentyps Record besteht aus dem reservierten Wort **record**, dem eine Auflistung der einzelnen Felder (*field list*) folgt. Danach steht das reservierte Wort **end**. Die Felderauflistung ist eine Folge von Sätzen (*record sections*), die durch Strichpunkte getrennt werden. Jeder Satz besteht aus einem oder mehreren Bezeichnern, gefolgt von einem Doppelpunkt und einem Datentypbezeichner. So bestimmt jeder Satz den Typ und den Bezeichner für ein oder mehrere Felder.

Beispiele:

```
type
  DaysOfMonth = record
                  Day: 1..31;
                  Month: (Jan,Feb,Mar,Apr,May,Jun,
                          Jul,Aug,Sep,Oct,Nov,Dec);
                  Year: 1900..1999;
                end;
Var
  Birth: Date;
  WorkDay: array[1..5] of date;
```

Day, *Month* und *Year* sind Feldbezeichner. Ein Feldbezeichner ist nur für den Record spezifisch, in dem er definiert wurde. Ein Feld wird durch den Variablenbezeichner und den Feldbezeichner, beide durch einen Punkt getrennt, angesprochen.

Beispiele:

```
Birth.Month := Jun;
Birth.Year := 1950;
WorkDay[Current] := WorkDay[Current-1];
```

Beachten Sie, daß, ähnlich wie bei Arrays, eine Zuweisung zwischen ganzen Records des gleichen Typs möglich ist. Da die einzelnen Komponenten beliebigen Typs sein können, sind deshalb folgende Konstruktionen der Ineinanderschachtelung möglich:

```
type
  Name   = record
             FamilyName: string[32];
             ChristianName: array[1..3] of string[16];
           end;
  Rate   = record
             NormalRate, OverTime,
             NightTime, Weekend: Integer;
           end;
  Date   = record
             Day: 1..31;
             Month: (Jan,Feb,Mar,Apr,May,Jun,
                     Jul,Aug,Sep,Oct,Nov,Dec);
             Year: 1900..1999;
           end;
  Person = record
             ID: Name;
             Time: Date;
           end;
  Wages  = record
             Individual: Person;
             Cost: Rate;
           end;

Var Salary, Fee: Wages;
```

Nach diesem Beispiel wären folgende Zuweisungen möglich:

```
Salary := Fee;
Salary.Cost.Overtime := 950;
Salary.Individual.Time := Fee.Individual.Time;
Salary.Individual.ID.FamilyName := 'Smith';
```

11.2 With Anweisung

Der oben beschriebene Gebrauch von Records führt manchmal zu relativ langen Anweisungen; es wäre einfacher, wenn man auf die einzelnen Felder in einem Record wie auf einfache Variablen zugreifen könnte. Dies ist die Funktion der **with** Anweisung; sie eröffnet einen Record, so daß die Feldbezeichner wie Variablenbezeichner benutzt werden können.

Eine **with** Anweisung besteht aus dem reservierten Wort **with**, gefolgt von einer Auflistung der Recordvariablen, die mit Kommas getrennt sind. Darauf folgt das reservierte Wort **do** und eine Anweisung .

Innerhalb einer **with** Anweisung ist ein Feld lediglich durch den Feldbezeichner bestimmt und nicht durch den Variablenbezeichner des Records:

```
with Salary do
begin
  Individual := NewEmployee;
  Cost := StandardRates;
end;
```

Records können innerhalb einer **with** Anweisung geschachtelt werden, d.h. daß Records von Records wie folgt eröffnet werden können:

```
with Salary, Individual, ID do
begin
  FamilyName := 'Smith';
  ChistianNames[1] := 'James';
end;
```

Dies entspricht:

```
with Salary do with Individual do with ID do
...
```

Die Maximalzahl ineinander verschachtelter **with** Anweisungen hängt von Ihrem Betriebssystem ab und wird in den Kapiteln 20, 21 und 22 besprochen.

11.3 Varianten (variant records)

Die Syntax eines Records erlaubt auch die Verwendung von Varianten, d.h. von alternativen Strukturen, bei denen sich die Recordfelder aus einer unterschiedlichen Anzahl und unterschiedlichen Typen von Komponenten zusammensetzen. Das hängt gewöhnlich vom Wert eines Variantenmarkierfelds (engl: tag-field) ab.

Ein Variantenteil besteht aus dem Variantenmarkierfeld eines zuvor definierten Typs, dessen Wert die jeweilige Variante bestimmt. Ihm folgen Labels, die jedem möglichen Wert des Variantenmarkierfelds entsprechen. Jedes Label steht einer Auflistung der Felder voran, die den entsprechenden Variantentyp definiert.

Angenommen es sei folgender Typ gegeben:

```
Origin = (Citizen, Alien);
```

Hinzu kommen die Datentypen *Name* und *Date*. Nachstehender Record erlaubt nun dem Feld *CitizenShip* verschiedene Strukturen anzunehmen, je nachdem, ob der Wert des Feldes *Citizen* oder *Alien* ist:

```
type
  Person = record
           PersonName: Name;
           BirthDate: Date;
           case Citizenship: Origin of
           Citizen: (BirthPlace: Name);
           Alien:     (CountryOfOrigin: Name;
                       DateOfEntry: Date;
                       PermittedUntil: Date;
                       PortOfEntry: Name);
           end;
```

In dieser Variantendefinition ist das Variantenmarkierfeld ein eigenes Feld, das wie jedes andere Feld behandelt werden kann. Ist *Passenger* eine Variable des Typs *Person* sind folgende Anweisungen durchaus möglich:

```
Passenger.CitizenShip := Citizen;

with Passenger, PersonName do
  if CitizenShip = Alien then writeln(FamilyName);
```

Der feststehende Teil eines Records, d.h. der Teil, der die normalen Felder beinhaltet, muß immer dem Variantenteil voranstehen. Im obigen Beispiel sind *PersonName* und *BirthDate* die feststehenden Felder. Ein Record kann nur einen einzigen Variantenteil haben. Auf jeden Fall muß eine Variante Klammern haben, auch wenn nichts darin steht.

Der Programmierer muß darauf achten, daß die Variantenmarkierfelder tatsächlich vorhanden sind. In TURBO Pascal kann nämlich auch auf das Feld *DateOfEntry* zugegriffen werden, wenn der Wert des Variantenmarkierfelds nicht *Alien* ist. Es ist sogar möglich, alle Feldbezeichner wegzulassen und nur die Bezeichner der Datentypen anzugeben. Derartige Recordvarianten werden freie Verbindungen (free unions) genannt im Unterschied zu solchen mit Variantenmarkierfeld (discriminated unions). Die freien Verbindungen werden nicht sehr häufig angewandt, und unerfahrene Programmierer sollten sie meiden.

Anmerkungen:

12. Mengen

Eine Menge (**set**) ist eine Zusammenfassung mehrerer Objekte desselben Typs, die als Ganzes gedacht werden. Jedes einzelne Objekt einer solchen Menge wird Element (*member, element*) genannt. Einige Beispiele:

1) Alle *integer* Zahlen zwischen 1 und 100

2) Alle Buchstaben des Alphabets

3) Alle Konsonanten des Alphabets

Zwei Mengen sind nur dann gleich, wenn auch ihre Elemente die Gleichen sind. Es gibt in ihnen keine Rangordnung, so daß die Mengen |1,3,5|, |1,5,3| und |5,3,1| gleich sind. Wenn die Elemente einer Menge auch die Elemente einer anderen sind, gilt die erste Menge als in der zweiten enthalten. Bei obigen Beispiel ist 3) in 2) enthalten.

Mit Mengen kann man auf drei verschiedene Arten rechnen (ähnlich der Addition, der Subtraktion und der Multiplikation mit Zahlen):

Die Vereinigung (*union, sum*) zweier Mengen A und B (geschrieben: A + B) ist die Menge, deren Elemente entweder in A oder in B enthalten sind. Die Vereinigung von |1,3,5,7| und |2,3,4| ist |1,2,3,4,5,7|.

Der Durchschnitt (*intersection, product*) zweier Mengen A und B (geschrieben: A * B) ist die Menge, die den beiden Mengen gemeinsam ist. Der Durchschnitt von |1,3,4,5,7| und |2,3,4| ist |3,4|.

Die Differenz oder das Komplement zweier Mengen, das *relative complement* (geschrieben: A - B) ist die Menge der Elemente der zuerst angegebenen Menge, die nicht auch in der zweiten Menge enthalten ist. Die Differenz von |1,3,5,7| und |2,3,4| ist |1,5,7|.

12.1 Mengendefinition

Obwohl es in der Mathematik keine Beschränkungen gibt, welche Elemente in einer Menge sein können, hat Pascal doch einige Restriktionen. Die Elemente einer Menge müssen alle vom gleichen Typ, dem Grundtyp sein, und dieser muß ein einfacher Datentyp sein, d.h. jeder skalare Datentyp, außer dem reellen. Einer Menge steht das reservierte Wort **set of** voran, gefolgt von einem einfachen Datentyp.

Beispiele:

```
type
  DaysOfMonth = set of 0..31;
  WorkWeek = set of Mon..Fri;
  Letter = set of 'A'..'Z';
  AdditiveColors = set of (Red,Green,Blue);
  Characters = set of Char;
```

Bei TURBO Pascal können maximal 256 Elemente in einer Menge enthalten sein, und die Ordinalwerte des Grundtyps müssen zwischen 0 und 255 liegen.

12.2 Mengenausdrücke

Die Werte einer Menge können mit den Werten einer anderen Menge über Mengenausdrücke einer Rechenoperation unterliegen. Mengenausdrücke bestehen aus Mengenkonstanten, Mengenvariablen, den Angaben der Menge und den Mengenoperationen.

12.2.1 Angabe der Menge

Die Mengenangabe besteht aus einem oder mehreren, durch Kommas getrennten und in eckigen Klammern stehenden Elementbestimmungen. Eine Elementbestimmung ist ein Ausdruck, vom gleichen Datentyp wie der Grundtyp der Menge. Oder es ist ein Bereich, der durch zwei solche Ausdrücke, zwischen denen zwei aufeinanderfolgende Punkte stehen, bestimmt wird.

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 zeigt eine *leere Menge*, die, da sie keine Ausdrücke enthält, die ihren Grundtyp angeben würden, zu allen Mengentypen kompatibel ist. Die Menge [1..5] ist äquivalent zu der Menge [1,2,3,4,5] . Wenn X>Y, dann steht [X..Z] für eine leere Menge.

12.2.2 Mengenoperatoren

Die Regeln der Mengenbildung geben den Vorrang der Mengenoperatoren nach den drei verschiedenen Klassen von Operatoren an:

1) * Durchschnitt der Mengen

2) + Vereinigung der Mengen
   - Differenz der Mengen

3) <> Test auf Gleichheit der Mengen
   <> Test auf Ungleichheit der Mengen
   >= Inklusion ('enthält') ist wahr, wenn der zweite Operand im ersten enthalten ist.
   <= Inklusion ('ist enthalten') ist wahr, wenn der erste Operand im zweiten enthalten ist.
   IN Test auf Mitgliedschaft in einer Menge. Der zweite Operand ist eine Menge und der erste Operand ein Ausdruck des gleichen Typs wie der Grundtyp der Menge. Das Ergebnis ist wahr (*true*), wenn der erste Operand ein Element des zweiten ist.

Es gibt keinen Operator für die Exklusion, aber man kann ihn wie folgt programmieren:

```
A * B = ||
```

Mengenausdrücke können zur Vereinfachung komplizierter Tests sehr hilfreich sein:

```
if (Ch='T') or (Ch='U') or (Ch='R') or (Ch='B') or (Ch='O')
```

kann auch klarer ausgedruckt werden:

```
Ch in |'T','U','R','B','O'|
```

Und der Test:

```
if (Ch >= '0') and (Ch <= '9') then...
```

sähe folgendermaßen besser aus:

```
if Ch in |'0'..'9'| then...
```

12.3 Mengenzuweisungen

Werte, die von Mengenausdrücken kommen, werden den Mengenvariablen mittels des Zuweisungsoperators := zugewiesen.

Beispiele:

```
type
  ASCII = set of 0..127;
Var
  NoPrint,Print,AllChars: ASCII;
begin
  AllChars := [0..127];
  NoPrint := [0..31,127];
  Print := AllChars - NoPrint;
end;
```

13. Typisierte Konstanten

Typisierte Kostanten sind eine Besonderheit von TURBO Pascal. Sie können genauso benutzt werden, wie eine Variable des gleichen Typs. Typisierte Konstanten können also als initialisierte Variablen benutzt werden, da der Wert einer typisierten Konstanten definiert ist, während der Wert einer Variablen undefinfiert ist, solange keine Zuweisung stattfand. Man sollte natürlich darauf achten, typisierten Konstanten keinen Wert zuzuweisen, wenn sie tatsächlich als Konstante verwendet werden sollen.

Die Benutzung von typisierten Konstanten verkleinert den Code, wenn sie oft im Programm gebraucht werden, da sie nur einmal im Programmcode auftauchen, während eine untypisierte Konstante, jedesmal wenn sie benutzt wird, im Programm angegeben werden muß.

Typisierte Konstanten sind wie nicht-typisierte (siehe Seite 48) definiert, mit dem Unterschied, daß die Definition nicht nur den Wert, sondern auch den Typ angibt. In der Definition folgen dem Bezeichner der typisierten Konstanten ein Doppelpunkt und der Bezeichner des Datentyps. Danach steht ein Gleichheitszeichen und die aktuelle Konstante.

13.1 Unstrukturierte typisierte Konstanten

Eine unstrukturierte typisierte Konstante ist wie ein skalarer Datentyp definiert:

```
const
  NumbersOfCars: Integer = 1267;
  Interest: Real = 12.67;
  Heading: string[7] = 'SECTION';
  Xon: Char = ^Q;
```

Im Unterschied zu nicht-typisierten können typisierte Konstanten anstelle von Variablen als Variablenparameter einer Prozedur oder Funktion stehen. Da eine typisierte Konstante in Wirklichkeit eine Variable mit konstantem Wert ist, kann sie nicht in der Definition anderer Konstanten oder Typen verwendet werden. Sind *Min* und *Max* typisierte Konstanten, ist folgendes Konstrukt nicht zulässig:

```
const
  Min: Integer = 0;
  Max: Integer = 50;
type
  Range: array[Min..Max] of integer;
```

13.2 Strukturierte typisierte Konstanten

Strukturierte Konstanten umfassen Array-Konstanten, Record-Konstanten und Mengenkonstanten. Sie werden oft für initialisierte Tafeln und Mengen im Testbereich, für Umwandlungen und für Mappingfunktionen benutzt. Die folgenden Abschnitte beschreiben jeden einzelnen Typ im Detail.

13.2.1 Array-Konstanten

Die Definition einer Array-Konstanten besteht aus dem Konstantenbezeichner, einem Doppelpunkt und dem Typenbezeichner eines zuvor definierten Arraytyps, dem ein Gleichheitszeichen und der Wert der Konstanten folgen. Dieser wird durch eine mit Kommas getrennte und in Klammern stehende Menge von Konstanten ausgedruckt.

Beispiel:

```
type
  Status =       (Active,Passive,Waiting);
  StringRep = array[Status] of string[7];
const
  Stat: StringRep = ('active','passive','waiting');
```

Das Beispiel definiert die Array-Konstante *Stat*, die z.B. die Werte des skalaren Datentyps *Status* in die entsprechenden Strings umwandelt. Die Komponenten von *Stat* sind:

```
Stat[Active]    = 'active'
Stat[Passive]   = 'passive'
Stat[Waiting]   = 'waiting'
```

Der Typ einer Komponente einer Array-Konstanten ist beliebig. Ausgeschlossen sind lediglich *File* und *Pointer*. Array-Konstanten aus Zeichen können entweder als einfache Zeichen oder als Strings bestimmt werden. Also kann folgende Definition:

```
const
  Digits: array[0..9] of Char = ('0','1','2','3','4','5','6','7','8','9');
```

auch einfacher ausgedrückt werden:

```
const
  Digits: array[0..9]of Char = '0123456789';
```

13.2.2 Multidimensionale Array-Konstanten

Multidimensionale Array-Konstanten werden definiert, indem die Konstanten jeder einzelnen Definition als eigenständige Mengen in Klammern angegeben werden. Die am weitesten innen stehenden Konstanten korrespondieren dabei zu den am weitesten rechts stehenden Dimensionen.

Beispiel:

```
type
  Cube = array[0..1,0..1,0..1] of integer;
const
  Maze: Cube = (((0,1),(2,3)),((4,5),(6,7)));
begin
  Writeln(Maze[0,0,0],' = 0');
  Writeln(Maze[0,0,1],' = 1');
  Writeln(Maze[0,1,0],' = 2');
  Writeln(Maze[0,1,1],' = 3');
  Writeln(Maze[1,0,0],' = 4');
  Writeln(Maze[1,0,1],' = 5');
  Writeln(Maze[1,1,0],' = 6');
  Writeln(Maze[1,1,1],' = 7');
end.
```

13.2.3 Record-Konstanten

Die Definition einer Record-Konstanten besteht aus einem Konstantenbezeichner, einem Doppelpunkt und dem Typenbezeichner eines zuvor definierten Recordtypen. Darauf folgt ein Gleichheitszeichen und der Wert der Konstanten. Der Wert wird durch eine Liste von Feldkonstanten ausgedrückt, die in runden Klammern steht. Die einzelnen Komponenten sind durch Kommas getrennt.

Beispiel:

```
type
   Point     = record
               X,Y,Z: integer;
                 end;
   OS        = (CPM80,CPM86,MSDOS,UNIX);
   UI        = (CCP,SomethingElse,MenuMaster);
   Computer= record
                 OperatingSystem: array[1..4] of OS;
                 UserInterface: UI;
               end;

const
  Origo: Point = (X:0; Y:0; Z:0);
  SuperComp: Computer =
        (OperatingSystems: (CPM80,CPM86,MSDOS,Unix),
        UserInterface: MenuMaster);
  Planel: array[1..3] of Point =
    ((X:1;Y:4;Z:5),(X:10;Y:-78;Z:45),(X:100;Y:10;Z:-7));
```

Die Feldkonstanten müssen in der gleichen Ordnung angegeben werden, in der sie in der Definition des Recordtypen auftauchen. Wenn ein Record Dateitypenfelder oder Pointertypen enthält, kann die Konstante dieses Recordtypen nicht angegeben werden. Wenn eine Record-Konstante eine Variante enthält, braucht der Programmierer lediglich das Feld der gültigen Variante anzugeben. Enthält die Variante ein Variantenmarkierfeld, muß ihr Wert bestimmt werden.

13.2.4 Mengenkonstanten

Eine Mengenkonstante besteht aus einer oder mehreren angegebenen Elementen, die mit Kommas voneinander getrennt sind und in eckigen Klammern stehen. Ein Element muß eine Konstante oder ein Ausdruck eines Bereiches sein, der aus zwei Konstanten, die mit zwei aufeinanderfolgenden Punkten getrennt sind, besteht.

Beispiel:

```
type
  Up    = set of 'A'..'Z';
  Low = set of 'a'..'z';
const
  UpperCase: Up = ['A'..'Z'];
  Vocals       : Low = ['a','e','i','o','u','y'];
  Delimiter:     set of Char = [''..'/',':'..'?','['..'',']'..'31'];
```

14. Dateitypen

Dateien (engl: files) sind die Kanäle, durch welche, die Programme Daten übertragen können. Eine Datei kann, entweder eine Disketten-Datei sein, in diesem Fall werden die Daten auf irgendeinen magnetischen Datenträger geschrieben und von da gelesen, oder ein logisches Gerät (engl: locigal device) wie etwa die vordefinierten Dateien *Input* und *Output*, die den Standard Ein-/Ausgabekanälen Tastatur und Bildschirm entsprechen.

Eine Datei (*file*) besteht aus einer Folge von Komponenten des gleichen Typs. Die Anzahl der Komponenten in einer Datei (ihre Größe) wird nicht durch die Definition bestimmt. Stattdessen benutzt Pascal einen Zeiger (*file pointer*), der den richtigen Zugriff gewährleistet. Jedesmal, wenn ein Lese- oder Schreibvorgang stattfindet, rückt der Zeiger anschließend zur nächsten Komponente. Da alle Komponenten die gleiche Länge haben, kann die Position einer bestimmten Komponente berechnet werden. Auf diese Weise kann der Pointer auf jede beliebige Komponente, bzw. auf einzelne Dateiblöcke, in der Datei direkt zugreifen (random access).

14.1 Definition der Dateitypen

Eine Datei wird mit dem reservierten Wort **file of** definiert. Ihm folgt der Komponententyp der Datei. Ein Dateibezeichner wird durch die gleichen Worte deklariert. Diesen folgt der Bezeichner eines zuvor definierten Dateityps.

Beispiel:

```
type
  ProductName = string[80];
  Product = file of record
                      Name: ProductName;
                      ItemNumber: Real;
                      InStock: Real;
                      MinStock: Real;
                      Supplier: Integer;
                    end;
Var
  ProductFile: Product;
  ProductNames: file of ProductName;
```

Die Komponenten können beliebigen Typs sein, außer einer Datei selbst, d.h. im obigen Beispiel wäre **file of** *Product* nicht zulässig. Dateivariable dürfen weder in Zuweisungen noch in Ausdrücken auftauchen.

14.2 Dateibearbeitung

Dieser Abschnitt behandelt die einzelnen Prozeduren, die für die Bearbeitung von Dateien vorhanden sind. Der Bezeichner *FilVar* steht im folgenden für einen Bezeichner einer Dateivariablen, wie er oben beschrieben ist.

14.2.1 Assign

Syntax: Assign(*FilVar,Str*)

Str ist ein Stringausdruck, der für jeden legalen Dateinamen steht. Dieser Dateiname wird der Dateivariablen *FilVar* zugewiesen. Alle weiteren Operationen mit *FilVar* arbeiten dann mit der Diskettendatei *Str*. Wenn eine Datei in Bearbeitung ist, darf nie eine Zuweisung erfolgen.

14.2.2 Rewrite

Syntax: Rewrite(*FilVar*)

Damit wird eine neue Diskettendatei geschaffen, die den Namen hat, der der Variablen *FilVar* zugewiesen wurde. Die Datei ist dann zur Bearbeitung vorbereitet, und der Zeiger steht am Anfang, d.h. bei Komponente 0. Jede bestehende Datei mit dem gleichen Namen wird überschrieben. Eine Diskettendatei, die mit *Rewrite* geschaffen wurde, ist zu Beginn leer, d.h. sie enthält keinerlei Elemente.

14.2.3 Reset

Syntax: Reset(*FilVar*)

Die Diskettendatei mit dem Namen, der der Dateivariablen *FilVar* zugewiesen wurde, wird zur Bearbeitung freigegeben und der Zeiger auf den Anfang gesetzt, d.h. zur Komponente 0. *FilVar* muß eine bereits existierende Datei bezeichnen, sonst wird eine I/O-Fehlermeldung ausgegeben.

14.2.4 Read

Syntax: Read(*FilVar, Var*)

Var steht für eine oder mehrere, durch Kommas getrennte Variablen des Komponententyps *FilVar*. Jede einzelne Variable wird aus der Datei eingelesen, und nach jeder Leseoperation wird der Zeiger auf die nächste Komponente gesetzt.

14.2.5 Write

Syntax: Write(*FilVar, Var*)

Var steht für eine oder mehrere, durch Kommas getrennte Variablen des Komponententyps *FilVar*. Jede Variable wird in die Datei geschrieben, und nach jeder Schreiboperation wird der Zeiger auf die nächste Komponente gesetzt.

14.2.6 Seek

Syntax: Seek(*FilVar, n*)

Seek bewegt den Zeiger zu der n-ten Komponente der Datei *FilVar*. *n* ist ein *integer* Ausdruck. Die Position der ersten Komponente ist 0. Um eine Datei zu erweitern, ist es möglich, die Komponente zu suchen, die nach der letzten stehen würde. Die Anweisung

```
Seek(FilVar, FileSize(FilVar));
```

setzt den Zeiger an das Ende der Datei (*FileSize* gibt die Anzahl der Komponenten in der Datei aus. Da die Komponenten von 0 an gezählt werden, ist die ausgegebene Zahl um 1 größer als die Platznummer der letzten Komponente).

14.2.7 Flush

Syntax: Flush(*FilVar*)

Flush leert den internen Puffer des Abschnittes der Datei *FilVar* und sorgt dafür, daß der Inhalt des Puffers auf die Diskette geschrieben wird, wenn seit dem letzten Update der Diskette eine Schreiboperation stattgefunden hat. Außerdem sorgt es dafür, daß auf die nächste Leseoperation auch tatsächlich ein physisches Lesen der Diskettendatei erfolgt. *Flush* sollte bei einer ungeöffneten Datei nicht verwendet werden.

14.2.8 Close

Syntax: Close(*FilVar*)

Die Datei *FilVar* wird geschlossen und die Directory auf den neuesten Stand gebracht. In Multi-User Umgebungen ist es oft notwendig, eine Datei zu schließen, auch wenn von ihr nur gelesen wurde.

14.2.9 Erase

Syntax: Erase(*FilVar*)

Die Datei *FilVar* wird gelöscht. Wenn die Datei noch offen ist, d.h. ein *Rewrite* oder ein *Reset*, aber kein *Close* stattgefunden hat, sollte sie vor der Löschung geschlossen werden.

14.2.10 Rename

Syntax: Rename(*FilVar, Str*)

Die Datei erhält mittels des Stringausdruckes *Str* einen neuen Namen. Die Directory wird auf den neuesten Stand gebracht, und alle zukünftigen Operationen beziehen sich mit *FilVar* auf diese Datei. Ist die Datei offen, sollte *Rename* nicht verwendet werden.

Beachten Sie , daß der Programmierer dafür verantwortlich ist, daß der Dateiname *Str* noch nicht existiert. Wenn doch, entstehen mehrere Dateien des gleichen Namens. Die folgende Funktion gibt den Wert *True* aus, wenn eine Datei gleichen Namens bereits existiert, ansonsten wird *False* ausgegeben:

```
type
  Name=string[66]
:
:
function Exist(FileName: Name): boolean;
Var
  Fil: file;
begin
  Assign(Fil, FileName);
  {$I-}
  Reset(Fil);
  {$I+}
  Exist := (IOresult = 0);
end;
```

14.3 Datei-Standardfunktionen

Folgende Standardfunktionen können bei der Dateibearbeitung verwendet werden:

14.3.1 EOF

Syntax: EOF(*FilVar*)

EOF ist eine Bool'sche Funktion, die den Wert *True* ausgibt, wenn der Zeiger am Ende der Datei steht, d.h. hinter der letzten Komponente. Wenn nicht, wird *False* ausgegeben.

14.3.2 FilePos

Syntax: FilePos(*FilVar*)

Eine *integer* Funktion, die die aktuelle Position des Zeigers ausgibt. Die erste Komponente hat die Position 0.

14.3.3 FileSize

Syntax: FileSize(*FilVar*)

Dies ist eine Funktion, deren Ergebnis vom Typ Integer ist und die die Länge der Datei, ausgedrückt als Zahl der darin enthaltenen Komponenten, ausgibt. Ist *FileSize(FilVar)* = 0, ist die Datei leer.

14.4 Der Gebrauch von Dateien

Bevor eine Datei benutzt wird, muß der Dateivariablen ein Dateiname zugewiesen werden (mit *Assign*). Bevor Input- und/oder Output-Operationen erfolgen, muß die Datei mit *Rewrite* oder *Reset* eröffnet werden. Damit wird der Zeiger auf die erste Komponente der Datei gesetzt, d.h. *FilePos(FilVar)* = 0. Nach einem *Rewrite* ist *FileSize(FilVar)* ebenfalls 0.

Eine Datei kann nur erweitert werden, indem man an das Ende der bereits existierenden Datei Komponenten anfügt. Der Zeiger wird mit folgendem Befehl an das Ende bewegt:

```
Seek(FilVar, FileSize(FilVar));
```

Wenn die I/O-Bearbeitung einer Datei beendet ist, sollte immer die *Close* Prozedur aufgerufen werden. Wenn dies nicht gemacht wird, können, da die Directory nicht auf den neuesten Stand gebracht wurde, Daten verloren gehen.

Das untenstehende Programm erzeugt eine Datei mit dem Namen *PRODUCTS.DTA* und schreibt 100 Records des Typs *Product* hinein. Die Datei ist damit für einen direkten Zugriff (random access) bereit, d.h. Records können an jeder beliebigen Stelle der Datei gelesen und geschrieben werden.

```
program InitProductFile
const
  MaxNumberOfProducts = 100;
type
  ProductName = string[20];
  Product = record
              Name: ProductName;
              ItemNumber: Integer;
              InStock: Real;
              Supplier: Integer;
            end;
Var
  ProductFile: file of Product;
  ProductRec: Product;
  I: Integer;
begin
  Assign(ProductFile, 'PRODUCT.DTA');
  Rewrite(ProductFile); {öffnet die Datei und löscht die Daten!
  with ProductRec do
  begin
    Name := ''; InStock := 0; Supplier := 0;
    for I := 1 to MaxNumberOfProducts do
    begin
      ItemNumber := I;
      Write(ProductFile,ProductRec);
    end;
  end;
  Close(ProductFile);
end;
```

Das folgende Programm demonstriert den Gebrauch von *Seek* bei Random Dateien . Das Programm wird benutzt, um die eben geschaffene Datei *ProductFile* auf den neuesten Stand zu bringen.

```
program UpDateProductFile;
const
  MaxNumberOfProducts = 100;
type
  ProductName = string[20];
  Product = record
              Name: ProductName;
              ItemNumber: Integer;
              InStock: Real;
              Supplier: Integer;
              end;
Var
  ProductFile: file of Product;
  ProductRec: Product;
  I,Pnr: Integer;
begin
  Assign(ProductFile,'PRODUCT.DTA'); Reset(ProductFile);
  ClrScr;
  Write('Enter product number (0=stop)'); Readln(Pnr);
  while Pnr in [1..MaxNumberOfProducts] do
  begin
    Seek(ProductFile,Pnr-1); Read(ProductFile,ProductRec);
    with ProductRec do
    begin
      Write('Enter name of product (',Name:20,') ');
      Readln(Name);
      Write('Enter number in stock (',InStock:20:0,') ');
      Readln(InStock);
      Write('Enter supplier number (',Supplier:20,') ');
      Readln(Supplier);
      Itemnumber:=Pnr;
    end;
      Seek(ProductFile,Pnr-1);
      Write(ProductFile,ProductRec);
      ClrScr; Writeln;
      Write('Enter product number (0=stop) '); Readln(Pnr);
  end;
    Close(ProductFile);
end;
```

14.5 Textdateien

Im Unterschied zu allen anderen Dateiarten, sind Textdateien nicht einfach Folgen von Werten des gleichen Typs. Obwohl die grundlegenden Komponenten von Textdateien Zeichen sind, sind sie in Zeilen unterteilt. Die Zeilen werden durch eine *end-of-line* Markierung (CR/LF) beendet. Außerdem werden Textdateien mit der Markierung *end-of-file* (Ctrl-Z) am Ende gekennzeichnet. Da die Zeilenlängen unterschiedlich sein können, ist die Position einer bestimmten Zeile nicht zu berechnen. Deshalb können Textdateien nur sequentiell bearbeitet werden. Außerdem kann bei einer Textdatei nicht simultan gelesen und geschrieben werden.

14.5.1 Bearbeitung von Textdateien

Eine Variable vom Typ Textdatei wird deklariert, indem man den standardisierten Typenbezeichner *Text* verwendet. Den anschließenden Bearbeitungsvorgängen muß ein Aufruf mit *Assign* vorangehen. Jedem I/O-Vorgang muß entweder ein *Reset* oder ein *Rewrite* vorangehen.

Rewrite wird verwendet, um eine neue Textdatei zu schaffen. Danach dürfen lediglich neue Komponenten an das Ende der Datei angehängt werden. Nach dem Öffnen einer bereits existierenden Datei durch ein *Reset* ist lediglich sequentielles Lesen erlaubt. Wird eine Textdatei geschlossen, wird automatisch eine Markierung an das Ende der Datei geschrieben.

Die Ein- und Ausgabe von Zeichen erfolgt durch die Standardprozeduren *Read* und *Write*. Ganze Zeilen werden mit den Prozeduren *ReadIn*, *WriteIn* und *Eoln* bearbeitet:

ReadLn

Syntax: ReadLn(*FilVar*);

ReadIn(FilVar) springt zum Beginn der nächsten Zeile, d.h. alle Zeichen werden inklusive der nächsten CR/LF-Sequenz übersprungen.

WriteLn

Syntax: WriteLn(*FilVar*);

WriteIn(FilVar) schreibt eine Zeilenmarkierung, d.h. eine CR/LF-Sequenz in die Textdatei.

Eoln

Syntax: Eoln(*FilVar*);

Eoln(FilVar) ist eine Bool'sche Funktion, die den Wert *True* ausgibt, wenn das Ende der in Bearbeitung befindlichen Zeile erreicht ist, d.h. der Zeiger am CR der Zeilenmarkierung CR/LF steht. Wenn EOF(*FilVar*) wahr ist, ist auch Eoln(*FilVar*) wahr.

SeekEoln

Syntax: SeekEoln(*FilVar*);

Ähnlich wie *Eoln*, außer daß es Leerzeichen und Tabulatoren überspringt, bevor es testet, ob eine *end-of-line* Markierung vorhanden ist. Der Ergebnistyp ist *boolean*.

SeekEof

Syntax: SeekEof(*FilVar*);

Ähnlich wie *EOF*, mit der Ausnahme, daß alle Leerzeichen, Tabulatoren, Zeilenende-Markierungen (CR/LF-Sequenzen) übersprungen werden, bevor es testet, ob eine *end-of-file* Markierung vorhanden ist. Der Ergebnistyp ist *boolean*.

Wenn die Funktion *EOF* bei Textdateien angewendet wird, hat sie *True* als Ergebnis, wenn der Dateizeiger auf der *end-of-file* Markierung positioniert ist (dem Zeichen Ctrl-Z, das die Datei abschließt). Die Prozeduren *Seek* und *Flush* und die Funktionen *FilePos* und *FileSize* sind auf Textdateien nicht anwendbar.

Das folgende Beispiel liest eine Textdatei von Diskette und druckt sie auf dem vordefinierten Gerät *Lst*, dem Drucker, aus. Wörter der Datei, die zwischen Ctrl-S stehen, werden unterstrichen:

```
program TextFileDemo
Var
  FilVar: Text;
  Line, ExtraLine: string[255];
  I: Integer;
  UnderLine: Boolean;
  FileName: string[14];
begin
  UnderLine := False;
  Write('Enter name of file to list: ');
  Readln(FileName);
  Assign(FilVar,FileName);
  Reset(FilVar);
  while not Eof(FilVar) do
  begin
    Readln(FilVar,Line);
    I := 1; ExtraLine := '';
    for I := 1 to Length(Line) do
    begin
      if Line[I] () ^S then
      begin
        Write(Lst,Line[I]);
        if UnderLine then ExtraLine := ExtraLine + '-';
        else ExtraLine := ExtraLine + ' ';
      end;
      else UnderLine := not UnderLine;
    end;
    Write(Lst,^M); Writeln(Lst,ExtraLine);
  end;{while not Eof}
end.
```

Erweiterungen der Prozeduren *Read* und *Write*, die die Bearbeitung von Input und Output erleichtern, werden auf Seite 108 besprochen.

14.5.2 Logische Geräteeinheiten

In TURBO Pascal werden externe Geräte, - wie Terminals, Drucker und Modems - wie Textdateien behandelt. Folgende Geräteeinheiten sind möglich:

CON:
Konsole (Console). Der Output wird über das Betriebssystem an das Ausgabegerät gegeben, gewöhnlich an den Bildschirm, der Input erfolgt über das Eingabegerät, gewöhnlich die Tastatur. Der Unterschied zu TRM (Terminal) besteht darin , daß bei CON der Input gepuffert abläuft. Kurz gesagt heißt dies, daß jedes *Read* oder *Readln*, das bei der Bearbeitung einer Textdatei CON: zugewiesen ist, eine ganze Zeile in den Zeilenpuffer eingibt, und daß das Betriebssystem während der Zeileneingabe mit Editieren beschäftigt ist. Weitere Details dazu stehen auf den Seiten 105 und 108.

TRM:
Terminal. Der Output wird normalerweise an den Bildschirm gesendet. Input wird üblicherweise von der Tastatur aufgenommen. Es findet ein Echo der eingegeben Zeichen statt, solange es keine Kontrollzeichen sind. Das einzige Kontrollzeichen mit Echo ist ein Carriage Return (CR). Dessen Echo ist CR/LF (Carriage Return/Line Feed).

KBD:
Tastatur (Keyboard). Üblicherweise steht KBD für Tastatur. Der Input hat kein Echo.

LST:
Lister. Üblicherweise wird damit der Drucker bezeichnet.

Aux:
Alternative (Auxiliary). In PC-/MS-DOS ist dies COM1:; in CP/M ist dies RDR: und PUN:.

USR:
Benutzer (User). Output wird an die Output-Routine des Benutzers gesendet, Input kommt von seiner Input-Routine. Genaueres erfahren Sie auf den Seiten 209, 241, 272.

Auf diese Geräte kann über die vorbezeichneten Dateien zugegriffen werden (siehe Seite 105), oder sie werden, entsprechend dem Verfahren bei Diskettendateien, Dateivariablen zugewiesen. *Close* hat keinerlei Funktion und der Versuch, so eine Datei zu löschen, erzeugt eine I/O-Fehlermeldung.

Die Standardfunktionen *Eof* und *Eoln* arbeiten bei Logischen Geräten anders als bei Diskettendateien. Bei einer Diskettendatei gibt *Eof* den Wert *True* aus, wenn das nächste Zeichen in der Datei ein CTRL-Z, oder wenn das physische Ende der Zeile erreicht ist. *Eoln* gibt den Wert *True* aus, wenn das nächste Zeichen ein Carriage Return oder ein CTRL-Z ist. Die beiden Funktionen sind also 'vorausschauend'.

Da man bei einem Ausgabegerät nicht vorausschauen kann, arbeiten die beiden Funktionen hier mit dem letzten, anstatt mit dem nächsten Zeichen. So gibt *Eof* den Wert *True*, wenn das letzte Zeichen ein CTRL-Z war, und *Eoln* gibt den Wert *True*, wenn das letzte gelesene Zeichen ein Carriage Return oder ein CTRL-Z war. Folgende Tafel gibt einen Überblick über die Arbeit mit *Eof* und *Eoln*:

	bei Dateien	bei logischen Geräten
Eoln ist 'true', wenn:	bei Diskettendateien das **nächste** Zeichen ein CR oder ein CTRL-Z ist, oder wenn EOF zutrifft	bei Ausgabegeräten das **gerade gelesene** Zeichen ein CR, oder ein CTRL-Z war
Eof ist 'true', wenn:	bei Diskettendateien das **nächste** Zeichen ein CTRL-Z, oder wenn das physische Ende der Datei erreicht ist	bei Ausgabegeräten das **gerade gelesene** Zeichen ein CTRL-Z war

Tafel 14-1: Arbeiten mit Eoln und Eof

Entsprechend arbeitet auch die Prozedur *Readln* unterschiedlich. Bei einer Diskettendatei werden alle Zeichen inklusive dem CR/LF gelesen, während bei einem logischen Gerät nur bis zum CR gelesen wird, da das System keine Möglichkeit hat, vorauszuschauen, um das Zeichen dahinter zu lesen.

14.5.3 Standarddateien

Als Alternative zur Zuweisung von Textdateien an Ausgabegeräte, wie es oben beschrieben wurde, bietet TURBO Pascal eine Anzahl vordeklarierter Textdateien, die schon bestimmten logischen Geräten zugewiesen und für die Bearbeitung vorbereitet sind. Auf diese Weise werden dem Programmierer die Prozeduren *Reset/Rewrite* und *Close* erspart. Außerdem verkleinert der Gebrauch dieser Standarddateien den Code:

Input	Die Inputdatei erster Ordnung. Diese Datei ist entweder dem CON-:Gerät zugeordnet oder dem TRM:Gerät (weitere Erläuterungen siehe unten).
Output.	Die Outputdatei erster Ordnung. Diese Datei ist entweder dem CON-:Gerät oder dem TRM:Gerät zugeordnet (weitere Erläuterungen siehe unten).
Con	Der Konsole zugeordnet (CON:).
Trm	Dem Terminal zugeordnet (TRM:).
Kbd	Der Tastatur zugeordnet (KBD:).
Lst	Dem Ausgabegerät (Drucker) zugeordnet (LST:).
Aux	Alternativ verwendbar (AUX:).
Usr	Dem Benutzergerät zugeordnet (USR:).

Beachten Sie, daß der Gebrauch von *Assign*, *Reset*, *Rewrite* und *Close* in diesen Dateien nicht erlaubt ist.

Wenn die Prozedur *Read* ohne Angabe eines Dateinamens verwendet wird, liest sie immer eine Zeile ein, sogar wenn noch Zeichen im Zeilenpuffer zu lesen sind, und ignoriert Ctrl-Z. Der Benutzer muß die Zeile deshalb mit RETURN beenden. Das abschließende RETURN hat kein Echo. Intern wird die Zeile mit einem Ctrl-Z am Ende gespeichert. Wenn also weniger Zeichen in der Eingabezeile angegeben werden, als Parameter in der Parameterliste vorhanden sind, werden alle *Char* Variablen darüber hinaus auf Ctrl-Z gesetzt, *string* Variablen sind dann leer und numerische Variablen bleiben ungeändert.

Der **B** Compilerbefehl wird zur Kontrolle der oben beschriebenen Möglichkeit, des zwangsweisen *Read* benutzt. Der voreingestellte Status ist | $B + |. In diesem Status verursachen Read-Anweisungen ohne Dateivariable immer die Eingabe einer Zeile von der Console. Wenn ein | $B- | Compilerbefehl am Anfang des Programms steht (**vor** dem Deklarierungsteil), wirkt die gekürzte Read-Version, als ob die Standarddatei Input spezifiziert worden wäre; d.h.:

Read(v1,v2,...,vn) entspricht *Read(Input,v1,v2,...,vn)*.

In diesem Fall werden Zeilen nur eingegeben, wenn der Zeilenpuffer geleert wurde. Der | $B- | Befehl folgt der I/O Definition von Standard Pascal, wogegen der voreingestellte | $B+ | Status , der Standard Pascal nicht in jeder Hinsicht entspricht, bessere Kontrolle von Eingabeoperationen erlaubt.

Wenn Eingaben nicht automatisch auf dem Bildschirm angezeigt werden sollen, sollten sie von der Standarddatei *Kbd* aus gemacht werden:

```
Read(Kbd, Var)
```

Da die Standarddateien *Input* und *Output* sehr häufig benutzt werden, werden sie durch die Voreinstellung automatisch gewählt, falls kein Dateityp definiert wird. Die folgende Tabelle zeigt die abgekürzten Textdateioperationen und ihre Entsprechungen:

Write(Ch)	*Write(Output,Ch)*
Read(Ch)	*Read(Input,Ch)*
Writeln	*Writeln(Output)*
Readln	*Readln(Input)*
Eof	*Eof(Input)*
Eoln	*Eoln(Input)*

Das folgende Programm zeigt den Gebrauch der Standarddatei Lst , die dem Ausdruck der Datei ProductFile dient (siehe Beispiel auf Seite 99):

```
program ListProductFile;
const
  MaxNumberOfProducts = 100;
type
  ProductName = string[20];
  Product = record
                Name: ProductName; ItemNumber: Integer;
                InStock: Real;
                Supplier: Integer;
                end;
Var
  ProductFile: file of Product;
  ProductRec: Product; I: Integer;
begin
  Assign(ProductFile,'PRODUCT.DTA'); Reset(ProductFile);
  for I := 1 to MaxNumberOfProducts do
  begin
    Read(ProductFile,ProductRec);
    with ProductRec do
    begin
      if Name <> '' then
        Writeln(Lst,'Item: ',ItemNumber:5,' ', Name:20,
                    ' From: ', Supplier:5,
                    ' Now in stock: ',InStock:0:0);
    end;
  end;
  Close(ProductFile);
end.
```

14.6 Ein- und Ausgabe von Textdateien

Die Ein- und Ausgabe von Daten in lesbarer Form wird durch *Textdateien* gemacht, die auf Seite 101 beschrieben sind. Eine Textdatei kann jedem Gerät zugeordnet werden, z.B. einer Diskettendatei oder einem der üblichen I/O Geräte. Input und Output in Textdateien erfolgen durch die Standard-Operationen *Read*, *Readln*, *Write*, und *Writeln*, die eine eigene Syntax für ihre Parametertabellen haben, um größtmögliche Flexibilität von Ein- und Ausgabe zu gewähren.

Insbesondere können Parameter unterschiedlicher Typen verwendet werden, wobei die I/O-Prozeduren automatisch die Daten auf den Grundtyp *Char* der Textdatei konvertieren.

Wenn der erste Parameter einer I/O-Prozedur ein Variablenbezeichner ist, der für eine Textdatei steht, dann wird I/O diese Datei bearbeiten. Wenn nicht, bearbeitet I/O die Standarddateien *Input* und *Output*.
Weitere Erläuterungen erfolgen auf Seite 105.

14.6.1 Read Prozedur

Die *Read* Prozedur ermöglicht die Eingabe von Buchstaben, Strings und Zahlen. Die Syntax der *Read* Anweisung ist:

Read(Var1,Var2,...,VarN)

oder

Read(FilVar,Var1,Var2,...,VarN)

wobei *Var1*, *Var2*,...,*VarN* Variable vom Typ *Char*, *String*, *Integer* oder *Real* sind. Im ersten Fall sind die Variablen Eingaben von der Standarddatei *Input*, gewöhnlich der Tastatur. Im zweiten Fall sind die Variablen Eingaben von einer Textdatei, die vorher zum *FilVar* erklärt und zum Lesen aufbereitet worden sind.

Read liest ein Zeichen aus der Datei und ordnet dieses Zeichen der Variablen zu, wenn der Variablentyp *Char* ist. Bei einer Diskettendatei ist *Eoln* wahr, wenn das nächste Zeichen ein CR oder ein Ctrl-Z ist, und *Eof* ist wahr, wenn das nächste Zeichen ein Ctrl-Z oder die Kapazität des Mediums erschöpft ist. Ist die Datei eine Ausgabedatei (die Standarddateien *Input* und *Output* eingeschlossen), ist *Eoln* wahr, wenn das gelesene Zeichen ein CR war, oder wenn *Eof* den Wert *True* annimmt. *Eof* ist wahr, wenn das gelesene Zeichen ein Ctrl-Z war.

Read liest bei Variablen vom Typ **string** die Anzahl von Zeichen, die vorher als maximale Länge des Strings definiert wurde, wenn nicht vorher *Eof* oder *Eoln* erreicht ist. *Eoln* ist wahr, wenn das gelesene Zeichen ein CR war, oder wenn *Eof* den Wert *True* annimmt. *Eof* ist wahr, wenn das zuletzt gelesene Zeichen ein Ctrl-Z, oder die Kapazität des Mediums erschöpft ist.

Bei einer numerischen Variablen (*Integer* oder *Real*), erwartet *Read* einen String von Zeichen, der dem Format einer numerischen Konstanten des relevanten Typs entspricht, wie sie auf Seite 43 definiert ist. Leerzeichen, TABs, CRs oder LFs, die einem String vorausgehen, werden übergangen. Der String darf nicht mehr als 30 Zeichen haben, und er muß durch ein Leerzeichen, ein TAB, ein CR oder ein Ctrl-Z begrenzt werden. Wenn der String nicht dem Format entspricht, tritt ein I/O Fehler auf. Andernfalls wird der numerische String in einen Wert des entsprechenden Typs umgewandelt und der Variablen zugeordnet. Wenn von einer Diskettendatei gelesen und der eingegebene String mit einem Leerzeichen oder einem TAB begrenzt wird, beginnt das nächste *Read* oder *Readln* mit dem Zeichen, das unmittelbar auf das Leerzeichen oder TAB folgt. Sowohl für Diskettendateien als auch für Ausgabedateien gilt, daß *Eoln* wahr ist, wenn der String mit CR endet, und daß *Eof* wahr ist, wenn der String mit Ctrl-Z endet.

Ein Sonderfall numerischer Eingabe ist, wenn *Eoln* oder *Eof* am Anfang von *Read* den Wert *True* annimmt (z.B. wenn die Eingabe von der Tastatur nur ein CR ist). In diesem Fall wird der Variablen kein neuer Wert zugeordnet.

Wenn die Konsole als Eingabedatei definiert ist (CON:), oder wenn die Standarddatei *Input* im {$B+} Modus (Voreinstellung) benutzt wird, gelten besondere Regeln für das Lesen von Variablen. Bei einem Aufruf von *Read* oder *Readln* wird eine Zeile von der Konsole eingegeben und in einem Puffer gespeichert. Beim Lesen der Variablen wird dann dieser Puffer als Eingabequelle benutzt. Dadurch wird es möglich, während der Eingabe zu editieren. Die folgenden Editiermöglichkeiten sind verfügbar:

BACKSPACE und **DEL**
Geht um eine Zeichenstelle zurück und löscht das dort befindliche Zeichen. BACKSPACE wird gewöhnlich durch Drücken der Tasten BS oder BACKSPACE oder durch Ctrl-H hervorgerufen. DEL wird gewöhnlich durch eine so bezeichnete Taste aufgerufen, bei manchen Tastaturen erfüllt RUB oder RUBOUT dieselbe Funktion.

Esc und Ctrl-X
Geht an den Zeilenanfang zurück und löscht alle eingegebenen Zeichen.

Ctrl-D
Gibt während des Eingabeprozesses Zeichen für Zeichen die letzte Eingabezeile aus.

Ctrl-R
Gibt die letzte Eingabezeile aus.

RETURN und **Ctrl-M**
Beendet die Eingabezeile und speichert eine Markierung für das Zeilenende (eine CR/LF Sequenz) im Zeilenpuffer. Dies geschieht durch Drücken der Taste RETURN oder ENTER. Das CR/LF erscheint **nicht** auf dem Bildschirm.

Ctrl-Z
Beendet die Eingabezeile und speichert eine *end-of-file* Markierung für das Zeilenende (ein CTRL-Z Zeichen) im Zeilenpuffer.

Intern wird die eingegebene Zeile mit einem Ctrl-Z am Zeilenende gespeichert. Ist die Eingabezeile kürzer als die Zahl der Variablen in der Parameterliste von *Read*, werden die überzähligen *Char* Variablen auf Ctrl-Z gesetzt, *Strings* werden leer und numerische Variablen nicht verändert.

Das Maximum an Zeichen, die von der Konsole in eine Zeile eingegeben werden kann, ist vorgegeben und beträgt 127. Sie können diese Begrenzung noch herabsetzen, indem Sie eine ganze Zahl zwischen 1 und 127 der vordefinierten Variablen *BufLen* zuweisen.

Beispiel:

```
Write('File name (max. 14 chars): '),
BufLen:=14;
Read(FileName);
```

Beachten Sie, daß Zuweisungen zu *BufLen* nur für das unmittelbar folgende *Read* gelten. Danach wird die *BufLen* wieder auf eine Länge von 127 gesetzt.

14.6.2 ReadIn Prozedur

Die *ReadIn* Prozedur ist identisch mit der *Read* Prozedur, mit der Ausnahme, daß der Rest der Zeile wegfällt. Das heißt, alle Zeichen bis zur nächsten CR/LF-Sequenz, diese eingeschlossen (oder das nächste CR auf einem logischen Gerät), fallen weg. Die Syntax dieses Anweisung ist:

ReadIn(Var1,Var2,...,VarN)

oder

ReadIn(FilVar,Var1,Var2,...,VarN)

Nach einem *ReadIn* wird das folgende *Read* oder *ReadIn* am Anfang der nächsten Zeile zu lesen beginnen. *ReadIn* kann auch ohne Parameter aufgerufen werden:

ReadIn

oder

ReadIn(FilVar)

Hier wird der Rest der Zeile gelöscht. Wenn *ReadIn* von der Konsole liest, (der Standarddatei *Input*, oder einer Datei, die CON: zugeordnet ist), dann wird das abschließende CR im Gegensatz zu *Read* als eine CR/LF-Sequenz auf dem Bildschirm gezeigt.

14.6.3 Write Prozedur

Mit der *Write* Prozedur können Zeichen, Strings, Bool'sche Werte und Zahlen ausgegeben werden. Die Syntax der *Write* Anweisung ist wie folgt:

Write(Var1,Var2,...,VarN)

oder

Write(FilVar,Var1,Var2,...VarN)

Dabei sind *Var1*, *Var2*,...,*VarN* Variablen vom Typ *Char*, *String*, *Boolean*, *Integer* oder *Real* , denen wahlweise ein Komma oder ein ganzzahliger Ausdruck folgen soll, mit denen die Größe des Ausgabefeldes bestimmt wird. Im ersten Fall werden die Variablen zur Standarddatei *Output* ausgegeben, gewöhnlich dem Bildschirm. Im zweiten Fall werden die Variablen an die nächstfolgende, zur *FilVar* erklärten, Datei ausgegeben.

Das Format des *write Parameter* hängt vom Variablentyp ab. Es folgt eine Beschreibung der unterschiedlichen Formate , ihrer Bedeutung und ihrer Symbole:

I,m,n
bezeichnet Ausdrücke vom Typ *Integer*,

R
bezeichnet Ausdrücke vom Typ *Real*,

Ch
bezeichnet Ausdrücke vom Typ *Char*,

S
bezeichnet Ausdrücke vom Typ *String*, und

B
bezeichnet Ausdrücke vom Typ *Boolean*.

Ch
Das Zeichen *Ch* wird ausgegeben.

Ch:n
Das Zeichen *Ch* wird in einem *n* Zeichen langen Feld rechtsbündig plaziert. Das restliche Feld wird mit Leerzeichen aufgefüllt.

S
Der String *S* wird ausgegeben. Arrays von Zeichen können ebenfalls ausgegeben werden, wenn sie mit Strings kompatibel sind.

S:n
Der String *S* wird in einem *n* Stellen langen Feld rechtsbündig plaziert. Das restliche Feld wird mit Leerzeichhen aufgefüllt.

B
Abhängig vom Wert von *B* wird entweder das Wort TRUE oder das Wort FALSE ausgegeben.

B:n
Abhängig vom Wert von *B* wird entweder das Wort TRUE oder das Wort FALSE in einem Feld rechtsbündig plaziert , das *n* Zeichen umfaßt.

I
Die Dezimaldarstellung des Wertes von *I* wird ausgegeben.

I:n
Die Dezimaldarstellung des Wertes von *I* wird in einem Feld rechtsbündig plaziert, das *n* Zeichen umfaßt.

R
Die Dezimaldarstellung des Wertes von *R* wird ausgegeben und in ein 18 Stellen langes Feld rechtsbündig plaziert, wobei das Gleitkommaformat verwandt wird. Für R > = 0.0, gilt folgendes Format:

␣␣#.##########E*##

Für R < 0.0, gilt folgendes Format:

␣-#.##########E*##

Dabei steht ␣ für ein Leerzeichen, # steht für eine Zahl und * steht für '+' oder '-'.

R:n
Die Dezimaldarstellung des Wertes von *R* wird in einem *n* Stellen langen Feld rechtsbündig plaziert, wobei das Gleitkommaformat verwandt wird:

Für R > = 0.0, gilt:

blanks#.digitsE*##

Für R < 0.0, gilt:

blanks - #.digitsE*##

Dabei steht *blanks* für ein oder mehrere Leerzeichen, *digits* für 1 bis 10 Zahlen, # steht für eine Zahl und * für '+' oder '-'. Da mindestens eine Stelle hinter dem Komma ausgegeben wird, umfaßt die minimale Feldgröße 7 Zeichen (8 für *R* < 0.0).

R:n:m
Die Dezimaldarstellung des Wertes von *R* wird ausgegeben und in einem *n* Stellen langen Feld rechtsbündig plaziert, wobei das Festkommaformat, mit m Stellen nach dem Dezimalpunkt verwandt wird. m muß im Wertebereich von 0 bis 24 liegen; andernfalls wird das Gleitkommaformat verwandt. Um die Feldgröße *n* zu füllen, gehen der Zahl entsprechend viele Leerzeichen voraus.

14.6.4 Writeln Prozedur

Die *Writeln* Prozedur entspricht der *Write* Prozedur, mit der Ausnahme, daß nach dem letzten Wert eine CR/LF Sequenz ausgegeben wird. Die Syntax der *Writeln* Anweisung ist folgende:

Writeln(Var1,Var2...,VarN)

oder

Writeln(FilVar,Var1,Var2,...,VarN)

Eine *Writeln* Anweisung ohne Write-Parameter gibt eine leere Zeile aus, die aus einer CR/LF Sequenz besteht:

Writeln

oder

Writeln(FilVar)

14.7 Nicht-typisierte Dateien

Nicht-typisierte Dateien sind I/O Kanäle niederer Ordnung, die vorwiegend zum direkten Zugriff auf Diskettendateien mit einer Recordgröße von 128 Bytes benutzt werden.

Bei Ein- und Ausgabevorgängen mit nicht-typisierten Dateien werden die Daten direkt von der Diskettendatei zur Variablen übertragen. So wird der Platz eingespart, den der für typisierte Dateien erforderliche Sektorenpuffer einnimmt. Eine untypisierte Dateivariable nimmt daher weniger Speicherkapazität in Anspruch als typisierte Dateivariablen. Da nicht-typisierte Dateien darüberhinaus kompatibel zu typisierten Dateien sind, ist ihre Verwendung vorzuziehen, wenn eine Dateienvariable nur für *Erase*, *Rename* oder andere Non-I/O Operationen verwendet werden sollen.

Eine nicht-typisierte Datei wird durch das dafür reservierte Wort **file** deklariert:

```
Var
DataFile: file;
```

14.7.1 BlockRead / BlockWrite

Alle Standardprozeduren und -funktionen zur Bearbeitung von Dateien können auch bei nicht-typisierten Dateien verwendet werden, mit den Ausnahmen *Read*, *Write* und *Flush*. *Read* und *Write* werden durch zwei besonders schnelle Übertragungsprozeduren ersetzt: *BlockRead* und *BlockWrite*. Mit folgender Syntax können diese Prozeduren aufgerufen werden:

BlockRead(FilVar, Var,Recs)
BlockWrite(FilVar, Var,Recs)

oder

BlockRead(FilVar, Var, Recs, Result)
BlockWrite(FilVar, Var, Recs, Result)

wobei *FilVar* die Variable einer nicht-typisierten Datei bezeichnet. *Var* bezeichnet eine beliebige Variable und *Recs* stellt einen ganzzahligen Ausdruck dar, mit dem die Anzahl der 128-Byte Records definiert wird, die zwischen Diskettendatei und Variable übertragen werden sollen. Der **fakultative** Parameter *Result* gibt die Zahl der Records an, die tatsächlich übertragen werden.

Die Übertragung beginnt mit dem ersten Byte, das mit der Variablen *Var* besetzt ist. Der Programmierer muß sicherstellen, daß der zur vollständigen Datenübertragung benötigte Raum von der Variablen *Var* freigehalten wird. Ein Aufruf von *BlockRead* oder *BlockWrite* läßt auch den Dateizeiger um *Recs* Records vorrücken.

Soll mit *BlockRead* oder *BlockWrite* eine Datei bearbeitet werden, muß diese zuerst mit *Assign* und *Rewrite* (oder *Reset*) dafür vorbereitet werden. *Rewrite* legt eine neue Datei an, *Reset* macht eine bestehende Datei zugänglich. Nach der Bearbeitung sollte die *Close* Anweisung gegeben werden, um einen eindeutigen Abschluß zu gewährleisten.

Die Standardfunktion *EOF* wird ebenso bei einer typisierten Datei ausgeführt. Das gilt auch für die Standardfunktionen *FilePos* und *FileSize* und die Standardprozedur *Seek*. Hier wird mit einer Komponentengröße von 128 Bytes gearbeitet (der Aufzeichnungsgröße von *BlockRead* und *BlockWrite*).

Das folgende Programm zeigt die Verwendung einer untypisierten Datei. Es liest jede beliebige Diskettendatei und überträgt ihren Inhalt auf jede beliebige andere Diskettendatei:

```
program FileCopy;
const
  BufSize       = 200;
  RecSize       = 128;
var
  Source,
  Destination:    File;
  SourceName,
  DestinationName:string[14];
  Buffer:           array[1..RecSize,1..BufSize]of Byte;
  RecsRead:         Integer;

begin
  Write('Copy from:     ');
  Readln(SourceName);
  Assign(Source, SourceName);
  Reset(Source);
  Write('               To: ');
  Readln(DestinationName);
  Assign(Destination, DestinationName);
  Rewrite(Destination);
  repeat
     BlockRead(Source,Buffer,BufSize,RecsRead);
     BlockWrite(Destination,Buffer,RecsRead);
  until RecsRead = 0;
  close(Destination); Close(Source);
end.
```

14.8 I/O Fehlerroutinen

Der **I** Compilerbefehl wird benutzt, um die Art der I/O Überwachung einzustellen. Voreingestellt ist in |$I+|, d.h. daß jede I/O Operation sofort nach der Ausführungen überprüft wird. I/O Fehler verursachen dann einen Programmabruch und es erscheint eine Fehlermeldung, die die Art des Fehlers anzeigt.

Wenn die automatische I/O Überwachung nicht aktiviert ist, d.h. bei |$I2-|, wird keine Laufzeit-Prüfung vorgenommen. Ein I/O Fehler ruft dann keinen Programmabbruch hervor, jedoch werden alle weiteren I/O Operationen unterbunden, bis die Standardfunktion *IOresult* aufgerufen wird. Diese Funktion stellt den Zustand vor Auftreten des Fehlers wieder her und Eingabe/Ausgabe kann wieder stattfinden. Es obliegt dem Programmierer, den I/O-Fehler zu beheben. Antwortet *IOresult* mit einer Null, zeigt dies den fehlerfreien Ablauf einer Operation an. Jede andere Antwort bedeutet, daß die letzte I/O Operation fehlerhaft war. Im Anhang I sind alle Fehlermeldungen und ihre Codes aufgelistet. **Beachten Sie**, daß bei Aufruf von *IOresult* der Zustand vor dem Auftreten des Fehlers hergestellt wird. Erneute Aufrufe von *IOresult* werden solange die Antwort Null erzeugen, bis der nächste I/O Fehler auftritt.

Die *IOresult* Funktion ist in den Fällen sehr hilfreich, in denen Programmabbrüche als Folge von I/O Fehlern nicht akzeptabel sind, wie im folgenden Beispiel, es wird solange nach einem Dateinamen gefragt, bis der Versuch eine Datei aufzurufen, geglückt ist (d.h. bis ein bekannter Dateiname eingegeben wurde):

```
procedure OpenInFile;
begin
  repeat
    Write('Enter name of input file ');
    Readln(InFileName);
    Assign(InFile, InFileName);
    |$I-| Reset(InFile) |$I+| ;
    OK := (IOresult = 0);
    if not OK then
        Writeln('Cannot find file ',InFileName);
  until OK;
end;
```

Wenn der I Befehl auf passiv Modus steht (!$I-!), sollten folgende Standardprozeduren mittels *IOresult* überprüft werden, um eine richtige Fehlerbehandlung sicher zu stellen:

*Append	Close	Read	Seek
Assign	Erase	ReadLn	Write
BlockRead	Execute	Rename	WriteLn
BlockWrite	Flush	Reset	
Chain	*GetDir	Rewrite	
*ChDir	*MkDir	*RmDir	

* nur PC-DOS/MS-DOS

Anmerkungen:

15. Zeiger-Typen (Pointer)

Die bisher diskutierten Variablen waren *statischer* Natur, d.h. ihre Form und Größe ist vorbestimmt und wird während der gesamten Bearbeitung des Abschnitts, für den sie definiert wurden, aufrechterhalten. Oft erfordern Programme jedoch eine Datenstruktur, die während der Bearbeitung in Form und Größe veränderlich sein sollte. Diesem Zweck dienen *dynamische* Variablen. Sie können bei Bedarf aufgerufen werden und entfallen, wenn sie nicht mehr benötigt werden.

Diese dynamischen Variablen werden nicht wie die statischen mittels einer Variablendeklarierung aufgerufen und sie lassen sich nicht über einen Bezeichner direkt zitieren. Stattdessen wird eine besondere Variable, die die Speicheradresse der Variablen enthält, benutzt, um auf die Variable zu zeigen. Diese besondere Variable heißt *Zeigervariable*.

15.1 Definition von Zeigervariablen

Ein Zeigertyp wird durch das Zeigersymbol ^ definiert, dem der *Typenbezeichner* der dynamischen Variablen folgt, der durch eine Zeigervariable dieses Typs zitiert werden kann.

Im Folgenden wird gezeigt, wie Records mit verwandten Zeigern angelegt werden können. Der Typ *PersonPointer* ist definiert als *Zeiger* von Variablen des Typs *PersonRecord*:

```
type
  PersonPointer = ^PersonRecord;

  PersonRecord = record
                   Name: string[50];
                   Job: string[50];
                   Next: PersonPointer;
                 end;
Var
  FirstPerson, LastPerson, NewPerson: PersonPointer;
```

Die Variablen *NextPerson*, *LastPerson* und *NewPerson* sind jene *Zeigervariablen*, die auf Records vom Typ *PersonRecord* zeigen können. Wie man sieht, kann sich die Typbezeichnung in einer Definition vom Typ *Zeiger* auf eine Bezeichnung beziehen, die noch nicht definiert wurde.

15.2 Zuordnung von Variablen (NEW)

Bevor irgendwelche von diesen Zeigervariablen benutzt werden, muss man natürlich einige Variablen haben, auf die man zeigen kann. Neue Variablen, egal von welchem Typ, werden mit der Standardprozedur *New* bezeichnet. Diese Prozedur hat einen Parameter, der den Variablen, die definiert werden sollen, den Typ zuweist.

Eine neue Variable vom Typ *PersonRecord* wird also folgendermaßen definiert:

```
New(FirstPerson);
```

mit dem Ergebnis, daß *FirstPerson* auf einen dynamisch zugeordneten Record vom Typ *PersonRecord* weist.

Zuweisungen zwischen Zeigervariablen können vorgenommen werden, solange die Zeiger vom gleichen Typ sind. Zeiger vom gleichen Typ können auch mit den logischen Operatoren = und <> untereinander verglichen werden, wobei sich als Ergebnis ein Bool'scher Wahrheitswert ergibt (true bzw. false).

Die Funktion **nil** ist mit allen Typen von Zeigern kompatibel. **nil** zeigt auf keine dynamische Variable und kann Zeigervariablen zugewiesen werden, um die Abwesenheit eines brauchbaren Zeigers anzuzeigen. **nil** kann auch in Vergleichen benutzt werden.

Variable, die durch die Standardprozedur *New* definiert wurden, werden in einer stapelartigen Struktur, *heap* genannt, abgelegt. Das Turbo Pascal System kontrolliert den Heap, indem es einen Heapzeiger erhält, der bei Beginn eines Programms auf die Adresse des ersten freien Bytes im Speicher initialisiert wird. Bei jedem Aufruf von *New*, wird der Heapzeiger an die Spitze des freien Speichers gesetzt, entsprechend der Anzahl der Bytes, die der Größe der neuen dynamischen Variable entspricht.

15.3 Mark und Release

Wenn eine dynamische Variable nicht länger benötigt wird, benutzt man die Standardprozeduren *Mark* und *Release*, um den diesen Variablen zugewiesenen Speicherplatz wieder freizumachen. Die *Mark* Prozedur weist den Wert des Heapzeigers einer Variablen zu. Die Syntax eines Aufrufs von *Mark* ist:

```
Mark(Var);
```

Dabei ist *Var* eine Zeigervariable. Die *Release* Prozedur setzt den Heapzeiger an die in ihrem Argument enthaltene Adresse. Die Syntax lautet:

```
Release(Var);
```

wobei *Var* eine Zeigervariable ist, die zuvor durch *Mark* gesetzt wird. *Release* entfernt dann alle dynamischen Variablen oberhalb dieser Adresse, kann aber nicht den durch Variablen benutzten Platz in der Mitte des Heap freimachen. Wenn Sie das tun möchten, sollten Sie anstatt von *Mark/Release Dispose* (Seite 124) verwenden.

Die Standardfunktion *MemAvail* kann jederzeit benutzt werden, um den auf dem Heap verfügbaren Platz zu bestimmen. Für weitere Hinweise siehe Kapitel 20, 21 und 22.

15.4 Die Benutzung von Zeigern

Angenommen wir haben die Prozedur *New* benutzt, um eine Serie von Records des Typs *PersonRecord* zu schaffen (wie im Beispiel auf der folgenden Seite), und daß das Feld *Next* in jedem Record auf das nächste *PersonRecord* deutet, dann gehen folgende Anweisungen die Liste durch und geben den Inhalt jedes Records aus (*FirstPerson* zeigt auf die erste Person in der Liste)

```
while FirstPerson <> nil do
with FirstPerson^ do
begin
  Writeln(Name,' is a ',Job);
  FirstPerson := Next;
end;
```

FirstPerson^.Name kann als *FirstPerson's.Name* gelesen werden, d.h: als das Feld *Name* auf das im Record mit *FirstPerson* gezeigt wird.

Folgendes Beispiel demonstriert den Gebrauch von Zeigern um eine Liste von Namen und gewünschten Berufen zu erstellen. Die Namen und gewünschten Berufe werden solange gelesen, bis ein Leerzeichen eingegeben wird. Danach wird die Liste ausgedruckt. Anschließend ist der benutzte Speicherplatz wieder frei. Die Zeigervariable *HeapTop* wird nur zur Aufnahme und zum Speichern des Anfangswertes gebraucht. Ihre Definition als ^Integer (Zeiger auf Integer) ist deshalb rein willkürlich.

```
procedure Jobs;
type
  PersonZeiger = ^PersonRecord;

  PersonRecord = record
                   Name: string[50];
                   Job: string[50];
                   Next: PersonZeiger;
                 end;
Var
  HeapTop: ^Integer;
  FirstPerson, LastPerson, NewPerson: PersonZeiger;
  Name: string[50];
begin
  FirstPerson := nil;
  Mark(HeapTop);
  repeat
    Write('Enter name:      ');
    Readln(Name);
    if Name <> '' then
    begin
      New(NewPerson);
      NewPerson^.Name := Name;
      Write('Enter profession: ');
      Readln(NewPerson^.Job);
      Writeln;
      if FirstPerson = nil then
        FirstPerson := NewPerson
      else
        LastPerson^.Next := NewPerson;
        LastPerson := NewPerson;
        LastPerson^.Next := nil;
    end;
  until Name = '';
  Writeln;
  while FirstPerson <> nil do
  with FirstPerson^ do
  begin
      Writeln(Name,' is a ',Job);
      FirstPerson := Next;
  end;
  Release(HeapTop);
end.
```

15.5 Dispose

Anstelle von *Mark/Release* kann die *Dispose* -Prozedur von Standard Pascal benutzt werden, um Speicherplatz im Heap zurückzugewinnen.

Beachten Sie, daß *Dispose* und *Mark/Release* verschiedene Arten der Heapverwaltung verwenden, **und diese nie zugleich benutzt werden dürfen**. Ein Programm kann **entweder** *Dispose*, **oder** *Mark/Release* verwenden, um den Heap zu verwalten. Diese zu mischen verursacht unvorhersagbare Ergebnisse.

Die Syntax ist: Dispose(Var), wobei *Var* eine Zeigervariable ist.

Dispose erlaubt es einen dynamischen Speicher, der von einer spezifischen Zeigervariable genutzt wird, für eine neuerliche Verwendung zurückzugewinnen. Im Gegensatz dazu setzt *Mark* und *Release* den ganzen Heap, von der spezifizierten Zeigervariable an abwärts, frei.

Nehmen wir an, Sie haben eine Reihe von Variablen, die dem Heap zugewiesen wurden. Das folgende Bild zeigt den Inhalt des Heap, und die Wirkung von *Dispose(Var)* und *Mark(Var3)/Release(Var3)*:

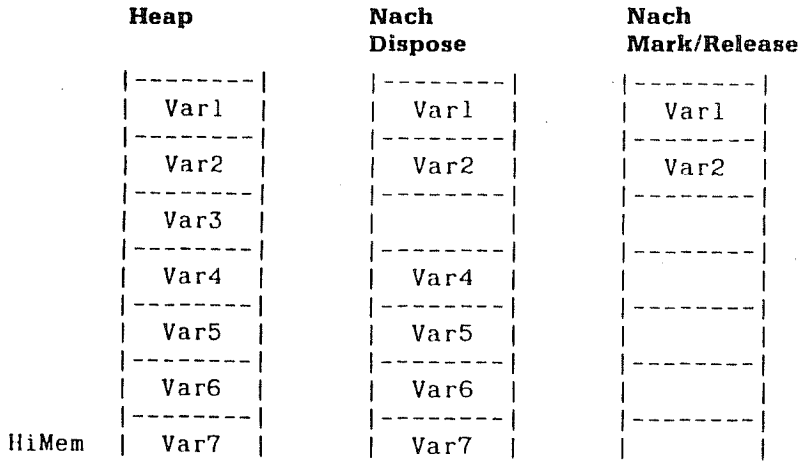

Bild 15-1: Gebrauch von Dispose

Nach der Anwendung von *Dispose* auf eine Zeigervariable, kann der Heap aus einer Reihe von benutzten Speicherelementen und dazwischenliegenden freien Speicherbereichen bestehen. Darauffolgende Aufrufe von *New* verwenden diese, wenn die neue Zeigervariable in diese Stellen paßt.

GetMem

Die Standardprozedur *GetMem* wird benutzt, um auf dem Heap einen bestimmten Platzbedarf zu reservieren. Im Gegensatz zu*New*, wo soviel Platz zugewiesen wird, wie der **Typ** benötigt, auf dessen Argument gezeigt wird, erlaubt *GetMem* dem Programmierer, die Größe des zugewiesenen Platzes zu kontrollieren. *GetMem* wird mit zwei Parametern aufgerufen:

```
GetMem(PVar, I)
```

PVar ist eine beliebige Zeigervariable, und *I* ist ein *integer* Ausdruck, der die Anzahl der Bytes angibt, für die Platz benötigt wird.

5.7 FreeMem

Syntax: FreeMem;

Die *FreeMem* Standardprozedur wird gebraucht, um einen ganzen Block auf dem Heap wieder freizumachen. Es ist also das Gegenstück zu *GetMem*. *FreeMem* wird mit zwei Parametern aufgerufen:

```
FreeMem(PVar,I);
```

wobei *PVar* eine beliebige Zeigervariable ist und *I* ein *integer* Ausdruck, der die Zahl der Bytes angibt, die wieder freizumachen sind. Diese Zahl muß **exakt** der Zahl von Bytes entsprechen, die vorher durch *GetMem* dieser Variablen zugewiesen worden sind.

5.8 MaxAvail

Syntax: MaxAvail;

Die *MaxAvail* Standardfunktion gibt die Größe des größten zusammenhängenden freien Platzes an, die auf dem Heap besteht. Bei 16-Bit Systemen steht dieser Platz in Paragraphen (pro 16 Bytes); bei 8-Bit Systemen in Bytes. Das Ergebnis ist eine ganze Zahl und wenn mehr als 32767 Paragraphen/Bytes verfügbar sind, gibt *MaxAvail* eine negative Zahl aus. Die korrekte Zahl freier Paragraphen/Bytes wird dann durch 65536.0 + *maxAvail* berechnet. Beachten Sie, daß reelle Konstanten verwendet werden müssen, um ein reelles Ergebnis zu erhalten, falls das Ergebnis größer als *MaxInt* ist.

16. Prozeduren und Funktionen

Ein Pascalprogramm besteht aus einem oder mehreren Blöcken, die wieder in Blöcke unterteilt sein können usw.. Ein solcher Block ist eine Prozedur, ein anderer eine Funktion (gemeinhin Unterprogramm genannt). Eine Prozedur ist also ein eigenständiger Teil im Programm, und kann mittels einer Prozeduranweisung von einer beliebigen Stelle im Programm aus aufgerufen werden (siehe Seite 56). Eine Funktion ist dem ziemlich ähnlich, aber sie berechnet einen Wert, wenn ihr Bezeichner während der Ausführung erreicht wird (siehe Seite 54) und gibt diesen dann aus.

16.1 Parameter

Werte können an Prozeduren und Funktionen durch *Parameter* übergeben werden. Dies erlaubt, ein Unterprogramm mit verschiedenen Werten zu fahren und damit auch unterschiedliche Ergebnisse zu bekommen.

Die Prozeduranweisung oder die Funktionsbezeichnung, die das Unterprogramm aufruft, kann eine Liste von Parametern enthalten, die *aktuellen Parameter*. Diese werden an die *formalen Parameter* übergeben, die im *Kopf* des Unterprogrammes bestimmt sind. Die Reihenfolge der Übergabe entspricht der Reihenfolge der Parameterliste. Pascal unterstützt zwei verschiedene Methoden der Parameterübergabe: über den *Wert* und über die *Referenz*, einer Veränderung der formalen Parameter. Dabei ist die Wirkung auf die aktuellen Parameter jeweils unterschiedlich.

Wenn Parameter über den *Wert* übergeben werden, entspricht der formale Parameter einer logischen Variable im Unterprogramm, und Veränderungen der formalen Parameter haben keine Auswirkung auf aktuelle Parameter. Der aktuelle Parameter kann jeder beliebige Ausdrück sein, einschließlich einer Variablen, der vom selben Typ ist, wie der entsprechende formale Parameter. Solche Parameter heißen *Wertparameter* und werden wie im folgenden Beispiel im Unterprogrammkopf deklariert. Dieses und das nächste Beispiel zeigen Prozedurüberschriften; Funktionsüberschriften unterscheiden sich etwas davon und sind auf Seite 137 beschrieben).

```
procedure Example(Num1,Num2: Number; Str1,Str2: Txt);
```

Number und *Txt* sind vorher definierte Typen (z.B. *integer* oder **string**[255]), und *Num1*, *Num2*, *Str1*, *Str2* sind *formale Parameter*, an die der Wert der *aktuellen Parameter* übergeben wird. Die Typen von formalen und aktuellen Parametern müssen übereinstimmen.

Beachten Sie, daß der Typ des Parameters im Parameterteil, so wie ein vorher definierter *Typenbezeichner*, angegeben werden muß. Deshalb ist die Angabe:

```
procedure Select(Model: array[1..500] of Integer);
```

nicht erlaubt. Stattdessen sollte der gewünschte Typ in der **type** Definition des Blocks definiert werden, und der Typenbezeichner sollte dann in der Parametererklärung benutzt werden:

```
type
  Range = array[1..500] of Integer;

procedure Select(Model: Range);
```

Wenn ein Parameter *durch Bezugnahme* übergeben wird, entspricht der formale Parameter tatsächlich während der Ausführung des Unterprogramms dem aktuellen Parameter. Jede Veränderung des formalem Parameters gilt folglich auch für den aktuellen Parameter, der deshalb eine *Variable* sein muß. Parameter, die durch Bezugnahme übergeben werden, werden *Variablenparameter* genannt und wie folgt deklariert:

```
procedure Example(Var Num1,Num2: Number)
```

Wertparameter und Variablenparameter können in derselben Prozedur gemischt werden, entsprechend folgendem Beispiel:

```
procedure Example(Var Num1,Num2: Number; Str1,Str2: Txt);
```

in dem *Num1* und *Num2* Variablenparameter sind und *Str1* und *Str2* Wertparameter.

Alle Adreßberechnungen werden zum Zeitpunkt des Prozeduraufrufs durchgeführt. Wenn eine Variable eine Komponente eines Arrays ist, werden deshalb ihre Indices Überprüft, wenn das Unterprogramm aufgerufen ist.

Beachten Sie, daß *Dateiparameter* immer als Variablenparameter deklariert werden müssen.

Wenn eine große Datenstruktur, wie etwa ein Array, an ein Unterprogramm als ein Parameter übergeben werden soll, spart die Benutzung eines Variablenparameters Zeit und Speicherplatz, da dann nur die Adresse des aktuellen Parameters an das Unterprogramm übergeben wird. Ein Wertparameter würde Speicherplatz und Zeit für eine zusätzliche Kopie der ganzen Datenstruktur benötigen.

16.1.1 Lockerung der Parametertyp-Überprüfung

Im Normalfall müssen bei der Benutzung von Variablenparametern die formalen und aktuellen Parameter exakt übereinstimmen. Unterprogramme, die Variablenparameter des Typs **String** verwenden, laufen nur mit Strings von genau der Länge, wie sie im Unterprogramm definiert ist. Diese Einschränkung kann durch den **V** Compilerbefehl aufgehoben werden. Der voreingestellte aktive Status |$V+| entspricht strenger Typenprüfung, während der passive Status |$V-| die Typenprüfung lockert und es erlaubt, aktuelle Parameter jeglicher Stringlänge zu übergeben, ungeachtet der Länge der formalen Parameter.

Beispiel:

```
program Encoder;
|$V-|
type
  WorkString = string|255|;
Var
  Line1: string|80|;
  Line2: string|100|;
procedure Encode(Var LineToEncode: WorkString);
Var I: Integer;
begin
  for I := 1 to Length(LineToEncode) do
    LineToEncode|I| := Chr(Ord(LineToEncode|I|)-30);
end;
begin
  Line1 := 'This is a secret message';
  Encode(Line1);
  Line2 := 'Here is another (longer) secret message';
  Encode(Line2);
end.
```

16.1.2 Nicht-typisierte Variablenparameter

Wenn der Typ des formalen Parameters nicht definiert ist, d.h. die Typendefinition in dem Parameterteil des Unterprogramm-Kopfes nicht aufgelistet ist, dann wird dieser Parameter *nicht-typisiert* genannt. Deshalb kann der entsprechende aktuelle Parameter beliebigen Typs sein.

Der untypisierte, formale Parameter selbst ist nicht kompatibel mit den anderen Typen und er kann deshalb nur dann benutzt werden, wenn der Datentyp keine Rolle spielt, z.B. als Parameter zu *Addr*, *BlockRead/Write*, *FillChar* oder *Move*, oder als die Adressenspezifikation einer absoluten Variablen.

Die *SwitchVar* Prozedur im folgenden Beispiel demonstriert den Gebrauch von nicht-typisierten Parametern. Es wird der Inhalt von A1 nach A2 und der Inhalt von A2 nach A1 bewegt.

```
procedure SwitchVar(Var A1p,A2p; Size: Integer);
type
  A = array[1..MaxInt] of Byte;
Var
  A1: A absolute A1p;
  A2: A absolute A2p;
  Tmp: Byte;
  Count: Integer;
begin
  for Count := 1 to Size do
  begin
    Tmp := A1[Count];
    A1[Count] := A2[Count];
    A2[Count] := Tmp;
  end;
end;
```

Angenommmen die Angaben lauten:

```
type
  Matrix = array [1..50,1..25] of Real;
Var
  TestMatrix,BestMatrix: Matrix;
```

dann kann man *SwitchVar* verwenden, um die Werte zwischen den beiden Matrixen zu vertauschen:

```
SwitchVar(TestMatrix,BestMatrix, SizeOf(Matrix));
```

16.2 Prozeduren

Eine Prozedur kann entweder vordeklariert ('oder standardisiert') oder vom Programmierer deklariert sein. Vordeklarierte Prozeduren sind Teile des TURBO Pascal Systems und können ohne weitere Angaben aufgerufen werden. Eine vom Benutzer festgelegte Prozedur kann den Namen einer Standardprozedur tragen; aber dann wird diese Standardprozedur unbrauchbar innerhalb des Bereichs der vom Benutzer festgelegten Prozedur.

16.2.1 Prozedurdeklarierung

Die Prozedurdeklarierung besteht aus einem Prozedurkopf, gefolgt von einem Block, der aus einem Deklarierungsteil und einem Anweisungsteil besteht.

Der Prozedurkopf besteht aus dem reservierten Wort **procedure** gefolgt von einem Bezeichner, für den Namen der Prozedur, wahlweise gefolgt von einer formalen Parameterliste, wie auf Seite 127 beschrieben.

Beispiele:
```
procedure LogOn;
procedure Position(X,Y: Integer);
procedure Compute(Var Data: Matrix; Scale: Real);
```

Der Deklarierungsteil einer Prozedur hat die gleiche Form wie der eines Programmes. Alle Bezeichner, die in der formalen Parameterliste und dem Deklarierungsteil deklariert sind, beziehen sich auf die jeweilige Prozedur und die darin eingebundenen Prozeduren. Außerhalb dieses Bezugsrahmens ist der Bezeichner nicht bekannt. Eine Prozedur kann sich auf jede Konstante, Variable, Prozedur oder Funktion beziehen, die in einem anderen Block steht.

Der Anweisungsteil spezifiziert die auszuführende Aktion, wenn die Prozedur aufgerufen wird und hat die Form einer gesamten Befehlszeile (siehe Seite 57). Wenn der Prozedurbezeichner innerhalb des Anweisungsteils selbst benutzt wird, wird die Prozedur rekursiv ausgeführt (**nur CP/M-80 Benutzer**: Beachten Sie, daß der **A** Compilerbefehl !$A-! passiv sein muß. Rekursion siehe auch Anhang C).

Das nächste Beispiel ist ein Programm, das eine Prozedur benutzt und einen Parameter an diese Prozedur übergibt. Wenn der aktuelle Parameter, der an die Prozedur übergeben wird, in manchen Fällen eine Konstante ist (ein einfacher Ausdruck), muß der formale Parameter ein Wertparameter sein.

```
program Box;
Var
  I: Integer;
procedure DrawBox(X1,Y1,X2,Y2: Integer);
  Var I: Integer;
  begin
    GotoXY(X1,Y1);
    for I := X1 to X2 do write('-');
    GotoXY(X1,Y1+1);
    for I := Y1+1 to Y2 do
    begin
      GotoXY(X1,I); Write('!');
      GotoXY(X2,I); Write('!');
    end;
    GotoXY(X1,Y2);
    for I := X1 to X2 do Write('-');
  end; { of procedure DrawBox }
begin
  ClrScr;
  for I := 1 to 5 do DrawBox(I * 4, I *2, 10 * I, 4 * I);
  DrawBox(1,1,80,23);
end.
```

Oft sollen die Veränderungen bei den formalen Parametern in einer Prozedur auch die aktuellen Parameter betreffen. In solchen Fällen werden *Variablenparameter* verwendet, wie im folgenden Beispiel:

```
procedure Switch(Var A,B: Integer);
Var Tmp: Integer;
begin
  Tmp := A; A := B; B := Tmp;
end;
```

Wenn diese Prozedur durch die Anweisung aufgerufen wird:

```
Switch(I,J);
```

werden die Werte von **I** und **J** vertauscht. Wenn der Prozedurkopf in **Switch** wie folgt deklariert war:

```
procedure Switch(A,B: Integer);
```

d.h. mit einem *Wertparameter*, dann würde die Anweisung *Switch(I,J)* *I* und *J* **nicht** vertauschen.

16.2.2 Standardprozeduren

TURBO Pascal kennt eine Reihe von Standardprozeduren. Diese sind:

1) String-Handhabungsprozeduren (beschrieben auf Seite 71 ff),
2) Datei-Handhabungsprozeduren (beschrieben auf Seite 94, 101 und 114),
3) Prozeduren für die Zuweisung von dynamischen Variablen (beschrieben auf Seite 120 und 125) und
4) Eingabe- und Ausgabeprozeduren (beschrieben auf Seite 108 ff)

Darüber hinaus sind folgende Standardprozeduren vorhanden, vorausgesetzt, daß die erforderlichen Kommandos für ihr Terminal installiert wurden (siehe Seite 12 ff):

16.2.2.1 ClrEol

Syntax: ClrEol

Löscht alle Zeichen von der Cursorposition bis ans Ende der Zeile, ohne den Cursor zu bewegen.

16.2.2.2 ClrScr

Syntax: ClrScr

Löscht den Bildschirm und plaziert den Cursor in die linke obere Ecke. Beachten Sie, daß manche Bildschirme auch die Videoeigenschaften zurücksetzen, wenn der Schirm gelöscht wird, was möglicherweise vom Benutzer gesetzte Eigenschaften verändern kann.

16.2.2.3 CrtInit

Syntax: CrtInit

Schickt den *Terminal Initialization String*, der in der Installierungsprozedur definiert ist, zum Bildschirm.

16.2.2.4 CrtExit

Syntax: CrtExit

Schickt den *Terminal Reset String*, der in der Installierungsprozedur definiert ist, an den Bildschirm.

16.2.2.5 Delay (Verzögerung)

Syntax: Delay (*Time*)

Die Prozedur *Delay* erzeugt eine Schleife, die ungefähr soviele Millisekunden läuft, wie im Argument *Time*, das eine ganze Zahl sein muß, angegeben ist. Die exakte Zeit kann in unterschiedlichen Betriebsumgebungen verschieden sein.

16.2.2.6 DelLine

Syntax: DelLine

Löscht die Zeile, in der der Cursor steht und bewegt alle Zeilen unterhalb davon um eine Zeile nach oben.

16.2.2.7 InsLine

Syntax: InsLine

Fügt eine Leerzeile an der Cursorposition ein. Alle Zeilen unterhalb werden eine Zeile nach unten bewegt, die unterste verschwindet vom Bildschirm.

16.2.2.8 GotoXY

Syntax: GotoXY(*Xpos*,*Ypos*)

Bewegt den Cursor auf die Position, die durch die ganzzahligen Ausdrücke *Xpos* (vertikaler Wert, oder *Spalteneinteilung*) und *Ypos*- (horizontaler Wert, oder*Zeileneinteilung*) angegeben werden. Die obere linke Ecke ist die Position (1,1) (home position).

16.2.2.9 Exit

Syntax: Exit
Verläßt den gegenwärtigen Block. Wenn Exit in einem Unterprogramm ausgeführt wird, bewirkt dies das Verlassen des Unterprogramms. Wenn Exit in dem Anweisungsteil eines Programms ausgeführt wird, verursacht es den Abbruch des Programms. Ein Aufruf von *Exit* kann mit einer **goto** Anweisung verglichen werden, die auf eine Adresse, kurz vor dem Ende (**end**) des Blocks zeigt.

162.2.10 Halt

Syntax: Halt

Beendet die Programmausführung und führt zum Betriebssystem zurück.

Bei PC/MS-DOS kann *Halt* wahlweise einen *integer* Parameter übertragen, der den Return-Code des Programms spezifiziert. *Halt* ohne einen Parameter entspricht *Halt(0)*. Der Return-Code kann durch den ursprünglichen Prozess überprüft werden, d.h. Benutzung eines MS-DOS Funktionsaufrufs, oder durch einen *ERRORLEVEL* Test in einer MS-DOS Batchdatei.

16.2.2.11 LowVideo

Syntax: LowVideo

Stellt den Bildschirm auf die Bildeigenschaften ein, die als ' Start of Low Video' in der Installierungsprozedur definiert sind (dunklere Schrift).

16.2.2.12 NormVideo

Syntax: NormVideo

Stellt den Bildschirm auf die Bildeigenschaften ein, die in der Installierungsprozedur als 'Start of Normal Video' definiert sind (Normalschrift).

16.2.2.13 Randomize

Syntax: Randomize

Startet den Zufallsgenerator mit einer Zufallszahl.

16.2.2.14 Move

Syntax: Move(*var1,var2,Num*)

Führt direkt im Speicher eine Kopie einer Anzahl von Bytes aus. *var1* und *var2* sind zwei Variablen beliebigen Typs, *Num* ist ein ganzzahliger Ausdruck. Die Prozedur kopiert einen Block von *Num* Bytes, beginnend beim ersten Byte, von *var1* auf das erste Byte von *var2*. Es gibt keine 'moveright' und 'moveleft' Prozeduren, da *Move* automatisch mögliche Überlappungen während des Move Prozesses handhabt.

16.2.2.15 FillChar

Syntax: FillChar(*Var,Num,Value*)

Füllt einen Speicherbereich mit einem gegebenen Wert. *Var* ist eine Variable beliebigen Typs, *Num* ist ein ganzzahliger Ausdruck und *Value* ein Ausdruck vom Typ *Byte* oder *Char*. *Num* Bytes, beginnend beim ersten durch *Var* belegtem Byte, werden mit dem Wert *Value* aufgefüllt.

16.3 Funktionen

Wie Prozeduren sind Funktionen entweder standardisiert (vordeklariert) oder vom Programmierer deklariert.

16.3.1 Funktionsdeklarierung

Eine Funktionsdeklarierung besteht aus einem *Kopf* und einem *Block*, der aus einem Deklarierungsteil, gefolgt von einem Anweisungsteil, besteht.

Der Funktionskopf entspricht dem Prozedurkopf, außer daß im Kopf der *Typ* des Ergebnisses der Funktion definiert sein muß. Dies geschieht, indem ein Doppelpunkt und ein Typ hinzugefügt wird:

```
function KeyHit: Boolean;
function Compute(Var Value: Sample) : Real;
function Power(X,Y: Real) : Real;
```

Der Ergebnistyp einer Funktion muß skalar (z.B. *Integer*, *Real*, *Boolean*, *Char*, deklarierter, skalarer Typ oder Teilbereich sein), vom Typ **String** oder ein Zeigertyp sein.

Der Deklarierungsteil einer Funktion ist identisch mit dem einer Prozedur.

Der Anweisungsteil einer Funktion ist eine zusammengesetzte Anweisung, wie auf Seite 57 beschrieben. Innerhalb des Anweisungsteils muß mindestens eine Anweisung enthalten sein, die dem Funktionsbezeichner einen Wert zuweist. Die zuletzt ausgeführte Zuweisung bestimmt das Ergebnis der Funktion. Wenn die Funktionsbezeichnung innerhalb des Anweisungsteils selbst steht, wird die Funktion rekursiv aufgerufen (**Nur für CP/M-80 Benutzer:** Beachten Sie, daß der **A** Compiler-Befehl passiv sein muß |$A|. Zu Rekursion siehe Anhang C).

Das folgende Beispiel zeigt den Gebrauch einer Funktion zur Berechnung der Summe einer Zeile aus ganzen Zahlen von **I** bis **J**.

```
function RowSum(I,J: Integer): Integer;
  function SimpleRowSum(S: Integer): Integer;
  begin
    SimpleRowSum := S*(S+1) div 2;
  end;
begin
  RowSum := SimpleRowSum(J)-SimpleRowSum(I-1);
end;
```

Die Funktion *SimpleRowSum* ist in die Funktion *RowSum* eingebettet. *SimpleRowSum* ist daher nur innerhalb des Bereichs von *RowSum* verfügbar.

Das folgende Programm ist das klassische Demonstrationsbeispiel für den Gebrauch einer rekursiven Funktion zur Berechnung des Faktors einer ganzen Zahl:

```
{$A-}
program Factorial;
Var Number: Integer;
function Factorial(Value: Integer) : Real;
begin
  if Value = 0 then Factorial := 1
  else Factorial := Value * Factorial(Value-1);
end;
begin
  Read(Number);
  Writeln(^M,Number,'! = ',Factorial(Number));
end.
```

Beachten Sie, daß der Typ, der in der Definition des Funktionstyps benutzt wird, zuvor als Typenbezeichner spezifiziert sein muß. Deshalb ist :

```
function LowCase(Line: UserLine): string[80];
```

nicht zulässig. Stattdessen sollte ein Typenbezeichner verbunden sein mit dem Typ *string[80]*, und sollte dann dazu benutzt werden, den Ergebnistyp der Funktion zu definieren, z.B.:

```
type
  Str80 = string[80];

function LowCase(Line: UserLine): Str80;
```

Wegen der Implementation der Standardprozeduren *Write* und *Writeln* darf eine Funktion, die die Standardprozeduren *Read,Readln,Write* oder *Writeln* benutzt, **nie** durch einen Ausdruck innerhalb eines *Write* oder *Writeln* Anweisung aufgerufen werden. In 8-Bit Systemen gilt dies auch für die Standardprozeduren *Str* und *Val*.

16.3.2 Standardfunktionen

Die folgenden Standardfunktionen sind in TURBO Pascal implementiert:

1) String-Behandlungsfunktionen (beschrieben auf Seite 71 ff)
2) Datei-Handhabungsfunktionen (beschrieben auf Seite 94 und 101)
3) Zeiger-Funktionen (beschrieben auf Seite 120 und 125)

16.3.2.1 Arithmetische Funktionen

16.3.2.1.1 Abs

Syntax: Abs(*Num*)

Gibt den absoluten Wert von *Num* aus. Das Argument *Num* muß entweder reell oder integer sein, und das Ergebnis ist vom selben Typ wie das Argument.

16.3.2.1.2 ArcTan

Syntax: ArcTan(*Num*)

Gibt den Winkel, dessen Tangente *Num* ist, in Bogenmaß an. Das Argument X muß entweder *real* oder *integer* sein, das Ergebnis ist *real*.

16.3.3.1.3 Cos

Syntax: Cos(*Num*)

Gibt den Cosinus von *Num* aus. Das Ergebnis wird in Bogenmaßausgedrückt und es muß entweder *integer* oder *real* sein. Das Ergebnis ist *real*.

16.3.2.1.4 Exp

Syntax: Exp(*Num*)

Gibt den Exponenten von *Num*, d.h. *e25num26* aus. Das Argument *Num* muß entweder *integer* oder *real* sein, das Ergebnis ist *real*.

16.3.2.1.5 Frac

Syntax: Frac(*Num*)

Gibt den Bruchteil von *Num* aus, d.h. Frac(*Num*) = *Num* -Int(*Num*). Das Argument muß entweder *integer* oder *real* sein, das Ergebnis ist *real*.

16.3.2.1.6 Int

Syntax: Int(*Num*)

Gibt den ganzzahligen Teil von *Num* an, und zwar die größte ganze Zahl kleiner oder gleich Null, falls *Num* > = 0, oder die kleinste ganze Zahl größer gleich *Num*, wenn *Num* < 0. Das Argument *Num* muß entweder *integer* oder *real* sein, das Ergebnis ist *real*.

16.3.2.1.7 Ln

Syntax: Ln(*Num*)

Gibt den natürlichen Logarithmus von *Num* aus. Das Argument *Num* muß entweder *integer* oder *real* sein, das Ergebnis ist *real*.

16.3.2.1.8 Sin

Syntax: Sin(*Num*)

Gibt den Sinus von *Num* an. Das Argument *Num* wird in Bogenmaß ausgedrückt, es muß entweder *integer* oder *real* sein. Das Ergebnis ist *real*.

16.3.2.1.9 Sqr

Syntax: Sqr(*Num*)

Gibt das Quadrat von *Num*, d.h. *Num***Num* aus. Das Argument *Num* muß entweder *integer* oder *integer* sein, das Ergebnis ist vom selben Typ wie das Argument.

16.3.2.1.10 Sqrt

Syntax: Sqrt(*Num*)

Gibt die Wurzel von *Num* aus. Das Argument *Num* muß entweder *integer* oder *real* sein, das Ergebnis ist *real*.

16.3.2.2 Skalar-Funktionen

16.3.2.2.1 Pred

Syntax: Pred(*Num*)

Gibt den Vorgänger von *Num* aus (falls dieser existiert). *Num* ist ein beliebiger skalarer Typ.

16.3.2.2.2 Succ

Syntax: Succ(*Num*)

Gibt den Nachfolger von *Num* aus (falls dieser existiert). *Num* ist ein beliebiger skalarer Typ.

16.3.2.2.3 Odd

Syntax: Odd(*Num*)

Gibt den Bool'schen Wahrheitswert *True* an, wenn *Num* eine ungerade Zahl ist und *False*, wenn *Num* eine gerade Zahl ist. *Num* muß *integer* sein.

16.3.2.3 Transfer-Funktionen

Die Transfer-Funktionen werden gebraucht, um die Werte eines skalaren Typen in die eines anderen umzurechnen. Zusätzlich zu den folgenden Funktionen dient auch die auf Seite 65 beschriebene *retype* Möglichkeit diesem Zweck.

16.3.2.3.1 Chr

Syntax: Chr(*Num*)

Gibt das Zeichen mit dem ordinalen Wert an, der durch den ganzzahligen Ausdruck *Num* gegeben ist. Beispiel: *Chr(65)* gibt das Zeichen 'A' aus.

16.3.2.3.2 Ord

Syntax: Ord(*Var*)

Gibt die ordinale Zahl des Werts von *Var* in der durch den Typ *Var* definierten Menge an. *Ord(Var)* ist gleichbedeutend mit *Integer(Var)* (siehe Seite 65). *Var* kann beliebigen skalaren Typs sein, außer *real*, das Ergebnis ist *integer*.

16.3.2.3.3 Round

Syntax: Round(*Num*)

Rundet den Wert von *Num* wie folgt:

wenn *Num* > = 0, dann ist *Round(Num) = Trunc(Num + 0.5)* und

wenn *Num* < = 0, dann ist *Round(Num) = Trunc(Num - 0.5)*

Num muß *real* sein, das Ergebnis ist *integer*.

16.3.2.3.4 Trunc

Syntax: Trunc(*Num*)

Gibt die größte ganze Zahl kleiner gleich *Num* an, falls *Num > = 0, oder die kleinste ganze Zahl größer gleich Num*, falls *Num < 0. Num* muß *real* sein und das Ergebnis ist *integer*.

16.3.2.4 Verschiedenartige Standardfunktionen

16.3.2.4.1 Hi

Syntax: Hi(*I*)

Das niederwertigere Byte des Ergebnisses enthält das höherwertigere Byte des Werts des ganzzahligen Ausdrucks *I*. Das höherwertige Byte des Ergebnisses ist Null. Das Ergebnis ist *integer*.

16.3.2.4.2 KeyPressed

Syntax: KeyPressed

Gibt den Bool'schen Wahrheitswert *True* aus, falls eine Taste der Konsole gedrückt wurde und *False*, falls keine Taste gedrückt wurde. Das Ergebnis erhält man, indem man über das Betriebssystem den Status der Konsole abfragt.

16.3.2.4.3 Lo

Syntax: Lo(*I*)

Gibt das niederwertigere Byte des Werts des ganzzahligen Ausdrucks *I*, mit dem auf Null gesetzten höherwertigen Byte, aus. Das Ergebnis ist *integer*.

16.3.2.4.4 Random

Syntax: Random

Gibt eine Zufallszahl größer oder gleich Null und kleiner Eins aus. Der Typ ist *real*.

16.3.2.4.5 Random(Num)

Syntax: Random(*Num*)

Gibt eine Zufallszahl größer oder gleich Null und kleiner als *Num* aus.

Num und die Zufallszahl sind beide *integer*.

16.3.2.4.6 ParamCount

Syntax: ParamCount

Diese *integer* Funktion gibt die Zahl der Parameter an, die an das Programm in den Befehlszeilenpuffer übertragen werden. Tabulator- und Leerzeichen dienen als Seperatoren.

16.3.2.4.7 ParamStr

Syntax: ParamStr(*N*)

Diese Stringfunktion gibt den *N*-ten Parameter aus dem Befehlszeilenpuffer an.

16.3.2.4.8 SizeOf

Syntax: SizeOf(*Name*)

Gibt die Zahl der durch die Variable oder den Typ *Name* im Speicher belegten Bytes aus. Das Ergebnis ist *integer*.

16.3.2.4.9 Swap

Syntax: Swap(*Num*)

Die Swap Funktion (Austauschfunktion) tauscht die nieder- und höherwertigen Bytes des ganzzahligen Arguments *Num* aus und gibt den Ergebniswert als ganze Zahl aus.

Beispiel:
Swap($1234) ergibt $3412 (Werte in Hex-Code)

16.3.2.4.10 UpCase

Syntax: UpCase(*ch*)

Gibt das großgeschriebene Äquivalent des Arguments *ch* an, das vom Typ *Char* sein muß. Falls kein großgeschriebenes, äquivalentes Zeichen existiert, wird das Argument unverändert ausgegeben.

16.4 Forward-Referenzen

Ein Unterprogramm ist **forward** deklariert, indem sein Kopf getrennt vom Block spezifiziert ist. Dieser separate Unterprogrammkopf ist exakt wie der normale Kopf, er wird nur mit dem reservierten Wort **forward** abgeschlossen. Der Block folgt später innerhalb desselben Deklarierungsteils. Beachten Sie, daß der Block von einer Kopie des Kopfs eingeleitet wird, die nur den Namen und keine Parameter, Typen, etc. spezifiziert.

Beispiel:

```
program Catch22;
var
  X: Integer;
function Up(Var I: Integer): Integer; foreward;
function Down(Var I: Integer): Integer;
begin
I := I div 2; Writeln(I);
if I <> 1 then I := Up(I);
end;
function Up;
begin
while I mod 2 <> 0 do
begin
  I := I*3+1; Writeln(I);
end;
I := Down(I);
end;
begin
Write('Enter any integer: ');
Readln(X);
X := Up(X);
Write('Ok. Program stopped again. ');
end.
```

Wenn dieses Programm ausgeführt wird und man z.B. *6* eingibt, erhält man als Ergebnis:

```
3
10
5
16
8
4
2
1
Ok. Program stopped again.
```

Das obige Programm ist eine kompliziertere Version des folgenden Programms:

```
program Catch222;
Var
  X: Integer;
begin
  Write('Enter any integer');
  Readln(X);
  while X () 1 do
  begin
    if X mod 2=0 then X : = X div 2 else X:=X*3+1;
    Writeln(X);
  end;
  Write('Ok. Program stopped again.');
end.
```

Vielleicht interessiert es Sie, daß dieses kleine und sehr einfache Programm nicht darauf geprüft werden kann, ob es wirklich für jede ganze Zahl stoppt!

17. Include-Dateien

Die Tatsache, daß der TURBO Editor nur innerhalb des Speichers editiert, begrenzt die Größe des Source. Der **I** Compilerbefehl kann benutzt werden, um diese Einschränkung zu umgehen. Er erlaubt, den Sourcetext in kleinere Stücke aufzuteilen und beim Compilieren wieder zusammenzusetzen. Die Include-Möglichkeit trägt auch zur Programmklarheit bei, da oft benutzte Unterprogramme, wenn sie einmal getestet und fehlerfrei sind, als Dateibibliothek aufbewahrt werden können. Aus dieser lassen sich dann die benötigten Dateien in jedes andere Programm einschließen.

Die Syntax für den Compilerbefehl **I** ist:

|$I filename|

filename kann jeder beliebige, erlaubte Dateiname sein. Leerzeichen werden ignoriert und Kleinbuchstaben in Großbuchstaben umgewandelt. Wenn kein Dateityp angegeben ist, wird der voreingestellte Typ **.PAS** angenommen. Dieser Befehl muß in einer eigenen Zeile gegeben werden.

Beispiele:

```
|$Ifirst.pas|
|$I StdProc|
|$I COMPUTE.MOD|
```

Beachten Sie, daß zwischen dem Dateinamen und der abschließenden Klammer ein Leerzeichen stehen muß, falls die Datei keine Dateitypenbezeichnung hat; sonst wird die Klammer als Teil des Namens betrachtet.

Um den Gebrauch der Include-Möglichkeit zu erläutern, nehmen wir an, daß in ihrer Bibliothek häufig benutzter Prozeduren und Funktionen eine Datei mit Namen *STUPCASE.FUN* enthalten ist. Sie enthält die Funktion *StUpCase*, die mit einem Zeichen oder String als Parameter aufgerufen wird und den Wert dieses Parameters ausgibt, den sie von Kleinbuchstaben in Großbuchstaben umwandelt.

Datei *STUPCASE.FUN:*

```
function StUpCase (St: AnyString): AnyString;
Var I: Integer;
begin
  for I := 1 to Length(St) do
    St[I] := UpCase(St[I]);
  StUpCase := St
end;
```

In jedem zukünftigen Programm, das die Funktion, Strings in Großbuchstaben umzuwandeln, benötigt, müssen sie die Datei nur noch beim Compilieren einfügen, anstatt sie im Source zu duplizieren:

```
program Include Demo;
type
  InData= string[80];
  AnyString= string[255];
Var
  Answer: InData;
{$I STUPCASE.FUN}
begin
  Readln(Answer);
  Writeln(StUpCase(Answer));
end.
```

Diese Methode ist nicht nur einfach und platzsparend; sie erleichert es auch, Programme schneller auf den neuesten Stand zu bringen und sicherer zu machen, da jede Veränderung einer Bibliotheks-Routine
automatisch alle Programme, die diese Routine enthalten, betrifft.

Beachten Sie, daß TURBO Pascal es erlaubt, einzelne Teile des Deklarierungsteils frei zu ordnen, und daß diese auch mehrfach auftreten können. Sie können so z.B. eine Anzahl von Dateien, die verschiedene oft gebrauchte Typendefinitionen enthalten, in ihrer Bibliothek haben und diese bei Bedarf in verschiedene Programme einfügen.

Alle Compilerbefehle außer **B** und **C** sind lokal zu der Datei in der sie vorkommen, d.h. wenn ein Compilerbefehl in einer eingefügten Datei einen veränderten Wert enthält, wird dieser auf den ursprünglichen Wert zurückgesetzt, wenn die Datei zurückgegeben wird. **B** und **C** Befehle sind immer global. Compilerbefehle sind in Anhang C beschrieben.

Include-Dateien können nicht geschachtelt werden, d.h. eine Include-Datei kann keine andere Include-Datei enthalten.

18. Overlay System

Das Overlay System (Overlay = Überlagerung) erlaubt Ihnen, viel größere Programme zu erzeugen, als der Speicher Ihres Computers fassen kann. Die Technik ist, eine Reihe von Unterprogrammen (Prozeduren und Funktionen) in einer oder mehreren Dateien, getrennt von der Hauptdatei, zusammenzufassen, die dann beim Programmlauf automatisch, jeweils eine, in den **selben** Bereich im Speicher geladen werden.

Die folgende Darstellung zeigt ein Programm, das eine Overlaydatei verwendet. Diese enthält fünf in einer **Overlaygruppe** zusammengefaßten Unterprogramme, die den selben Speicherplatz im Hauptprogramm gemeinsam verwenden:

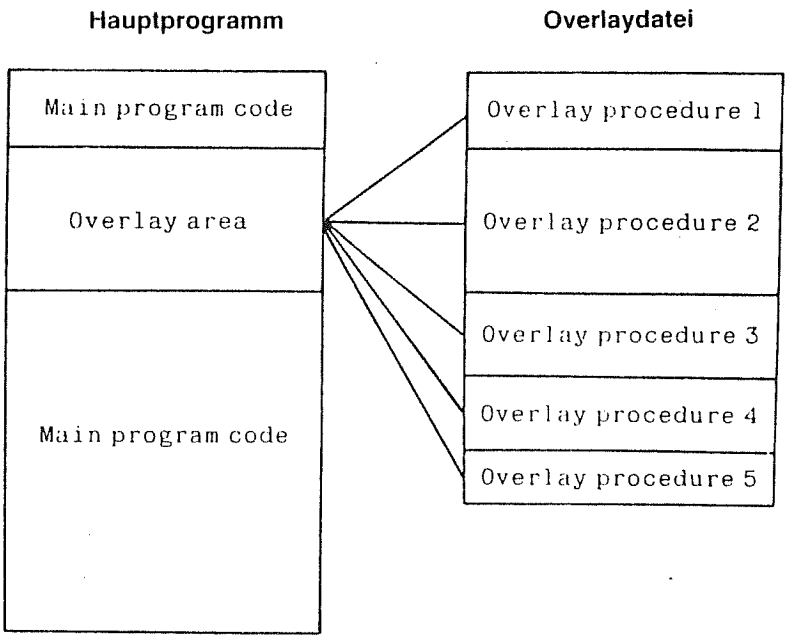

Bild 18-1: Prinzip des Overlay Systems

Wenn eine der Overlayprozeduren aufgerufen wird, wird diese automatisch in den im Hauptprogramm reservierten Overlaybereich geladen. Diese Lücke ist groß genug, um sogar die größten Overlays der Gruppe zu fassen. Der vom Hauptprogramm benötigte Platz wird deshalb, um etwa die Summe aller Unterprogramme in der Gruppe minus des größten reduziert.

Im obigen Beispiel ist Prozedur 2 die größte der fünf Overlayprozeduren und bestimmt deshalb die Größe des Overlaybereichs im Hauptprogramm-CCode.

Wenn diese in den Speicher geladen wird, besetzt sie den ganzen Overlaybereich:

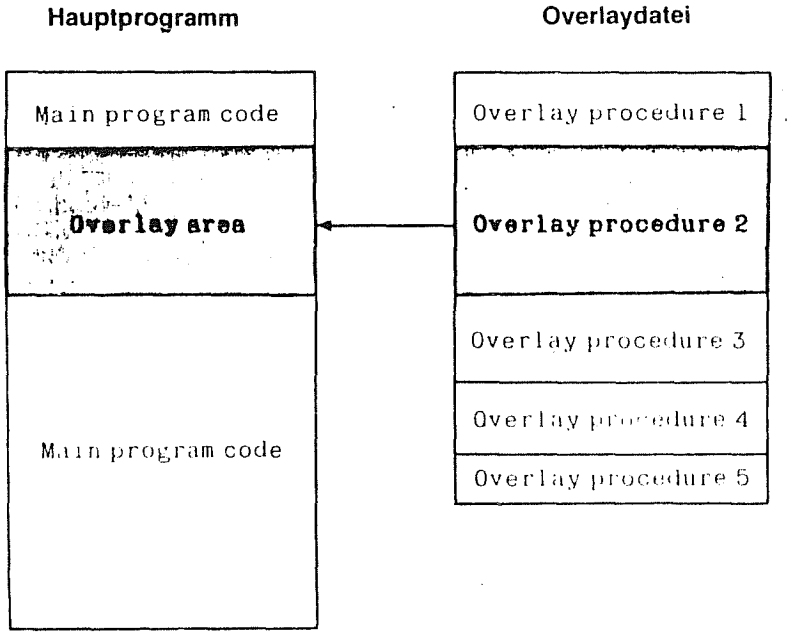

Bild 18-2: Größtes Overlay-Unterprogramm, geladen

Die kleineren Unterprogramme werden in denselben Bereich des Speichers geladen, wobei jedes an der ersten Adresse des Overlaybereichs beginnt. Natürlich belegen sie nur Teile des Overlaybereichs, der Rest bleibt ungenutzt:

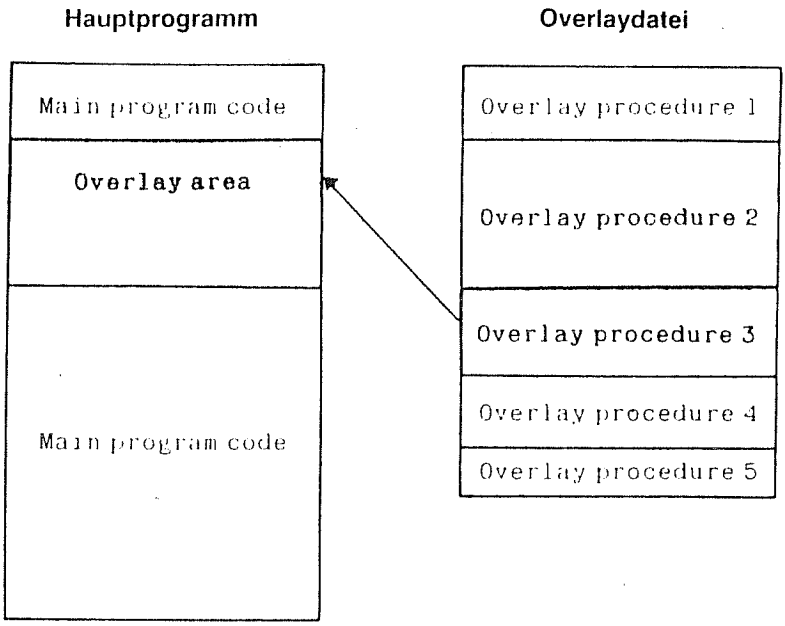

Bild 18-3 Kleinere Overlay-Unterprogramme, geladen

Da die Prozeduren 1, 3, und 5 am selben Platz ausgeführt werden, den die Prozedur 2 verwendet, ist klar, daß sie keinen zusätzlichen Platz im Hauptprogramm benötigen. Es ist ebenfalls klar, daß sich diese Prozeduren nicht gegenseitig aufrufen können, da sie niemals gleichzeitig im Speicher sind.

Es könnten in dieser Overlaygruppe wesentlich mehr Overlayprozeduren sein. Die gesamte Größe der Overlayprozeduren könnte die Größe des Hauptprogramms sogar wesentlich überschreiten und sie würden trotzdem nur den Platz der größten Overlayprozedur benötigen.

Der Preis für den zusätzlichen Platz für Programmmcode ist die zusätzliche Zeit für Diskettenzugriffe, die bei jedem Einlesen einer Prozedur von Diskette verbraucht wird. Bei guter Planung, wie auf Seite 155 beschrieben, ist diese Zeit vernachläßigbar gering.

18.1 Erzeugen von Overlays

Overlay-Unterprogramme werden automatisch erzeugt, einfach indem das reservierte Wort **overlay** bei der Deklaration einer beliebigen Prozedur oder Funktion hinzugefügt wird, z.B.:

```
overlay procedure Initialize;
```

und

```
overlay function TimeOfDay: Time;
```

Wenn der Compiler auf eine solche Deklaration trifft, wird der Code nicht weiter an die Hauptprogrammdatei ausgegeben, sondern an eine gesonderte Overlaydatei. Der Name dieser Datei ist derselbe, wie der des Hauptprogramms, der Typ ist eine Zahl im Bereich 000 bis 099, die die Overlaygruppe bezeichnet.

Aufeinanderfolgende Overlay-Unterprogramme werden zusammengruppiert. Mit anderen Worten, solange Overlay-Unterprogramme nicht durch eine andere Deklarierung getrennt werden, gehören sie zur selben Gruppe und werden in dieselbe Overlaydatei plaziert.

Beispiel 1:

```
overlay procedure Eins;
begin
:
end;

overlay procedure Zwei;
begin
:
end;

overlay procedure Drei;
begin
:
end;
```

Diese drei Overlayprozeduren werden zusammmengruppiert und in dieselbe Overlaydatei plaziert. Wenn sie die erste Gruppe von Unterprogrammen in einem Programm sind, hat die Overlaydatei die Nummer 000.

Die drei Overlayprozeduren im folgenden Beispiel werden in aufeinanderfolgenden Overlaydateien, z.B. *.000* und *.001* plaziert, da die Deklarierung der Nicht-Overlayprozedur *Count* die Overlayprozeduren *Zwei* und *Drei* trennt. Die trennende Deklarierung könnte jede beliebige Deklarierung, z.B. eine Dummy-Deklarierung sein, wenn eine Trennung der Overlaybereiche erzwungen werden soll.

Beispiel 2:

```
overlay procedure Eins;
begin
:
end;

overlay procedure Zwei;
begin
:
end;

procedure Count;
begin
:
end;

overlay procedure Drei;
begin
:
end;
```

Im Hauptprogramm wird für jede Gruppe von Overlay-Unterprogrammen ein getrennter Overlaybereich reserviert. Beispiel 2 würde also die folgenden Dateien erzeugen.

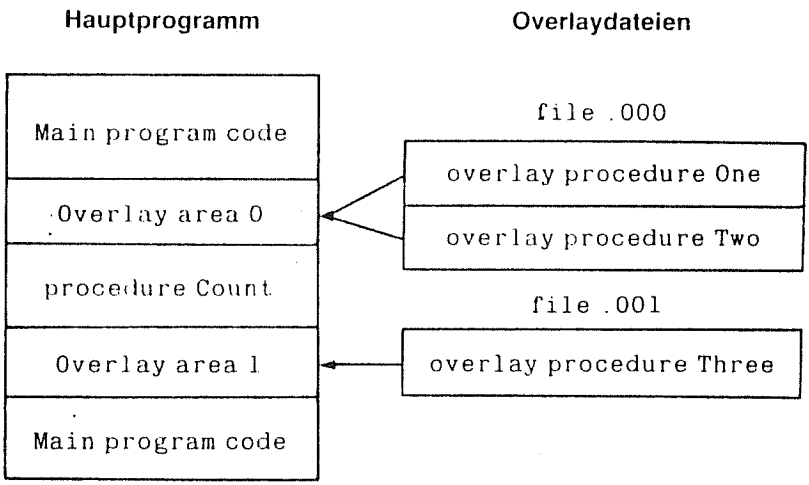

Bild 18-4: Mehrfache Overlaydateien

18.2 Geschachtelte Overlays

Ein Overlay-Unterprogramm kann auch geschachtelt sein. Was bedeutet, daß ein Overlay-Unterprogramm selbst ein Overlay-Unterprogramm enthalten kann, das ein Overlay-Unterprogramm enthalten kann, usw..

Beispiel 3:

```
program OverlayDemo;
:
:
overlay procedure Eins;
begin
:
end;

overlay procedure Zwei;
overlay procedure Drei;
begin
:
end;
begin
:
end;
:
```

In diesem Beispiel werden zwei Overlaydateien erzeugt. Datei .000 enthält Overlayprozedur *Eins* und *Zwei*. Es wird ein Overlaybereich im Hauptprogramm reserviert, der der größten dieser Prozeduren Platz bietet. Overlaydatei .001 enthält die Overlayprozedur *Drei*, die zur Overlayprozedur *Zwei* lokal ist und im Code der Overlayprozedur *Zwei* einen Overlaybereich erzeugt:

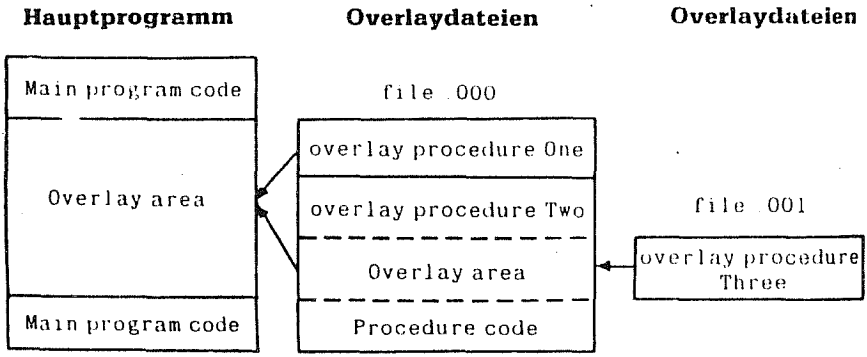

Bild 18-5: Geschachtelte Overlaydateien

18.3 Automatische Overlayverwaltung

Ein Overlay-Unterprogramm wird nur in den Speicher geladen, wenn es aufgerufen wird. Bei jedem Aufruf eines Overlay-Unterprogramms wird geprüft, ob das Unterprogramm schon im Overlaybereich vorhanden ist. Wenn nicht, wird es automatisch von der entsprechenden Overlaydatei eingelesen.

18.4 Plazierung von Overlaydateien

Während der Compilierung werden die Overlaydateien auf dem angemeldeten Laufwerk, d.h. auf demselben Laufwerk wie die Hauptdatei (.Com oder.CMD Datei) plaziert.

Während der Ausführung erwartet das System, daß es die Overlaydateien auf dem angemeldeten Laufwerk findet. Das kann sich wie auf Seite 196 (PC/MS-Dos), auf Seite 233 (CP/M-86) und auf Seite 265 (CP/M-80) beschrieben ändern.

18.5 Effizienter Gebrauch von Overlays

Die Overlaytechnik belastet ein Programm natürlich, da zum Programm zusätzlicher Code zur Verwaltung der Overlays hinzukommt und bei der Programmausführung auf Diskette zugegriffen werden muß. Overlays sollten deshalb mit Bedacht eingesetzt werden.

Um die Ausführungszeit nicht allzusehr zu verlangsamen, sollte ein Overlay-Unterprogramm nicht zu oft aufgerufen werden. Wenn es oft aufgerufen wird, sollte es wenigstens ohne dazwischenliegende Aufrufe anderer Unterprogramme derselben Overlaydatei aufgerufen werden. Damit werden die Diskettenzugriffe auf ein Minimum zu beschränkt. Die zusätzlich benötigte Zeit variiert natürlich je nach der vorhandenen Laufwerks-Konfiguration. Ein 5 1/4 Laufwerk erhöht die Ausführungszeit stark, eine Festplatte verlängert diese schon weniger und eine RAM-Floppy kaum.

Um soviel Platz wie möglich im Hauptprogramm zu sparen, sollte eine Gruppe von Overlays möglichst viele, einzelne Unterprogramme enthalten. Im Interesse der Speicherplatzersparnis ist es umso besser, je mehr Unterprogramme Sie in einer Overlaydatei unterbringen. Der Overlaybereich im Hauptprogramm muß nur dem größten dieser Unterprogramme Platz bieten - den restlichen Unterprogrammen steht dann derselbe Speicherplatz voll zur Verfügung. Die hier erörterten Gesichtspunkte der Speicherplatzersparnis müssen gegen die oben diskutierten Überlegungen bezüglich der Ausführungszeit abgewogen werden.

18.6 Restriktionen bei Overlays

18.6.1 Datenbereich

Overlay-Unterprogramme aus denselben Gruppen verwenden denselben Speicherbereich und können ihn deshalb nicht gleichzeitig belegen. Sie dürfen sich deshalb **nicht** gegenseitig aufrufen. Folglich können sie denselben Datenbereich verwenden, was bei der Benutzung von Overlays weiter Platz spart (Nur CP/M-80).

In Beispiel 1 auf Seite 152 können deshalb die Prozeduren sich nicht gegenseitig aufrufen. In Beispiel 2 kann jedoch Prozedur Eins und Zwei die Overlayprozedur Drei aufrufen und die Overlayprozedur Drei kann jede der zwei anderen aufrufen, da diese sich in getrennten Dateien und folglich in getrennten Overlaybereichen im Hauptprogramm befinden.

18.6.2 Forward-Deklarierung

Overlay-Unterprogramme können nicht **forward** deklariert werden. Diese Restriktion kann leicht umgangen werden, indem man ein normales Unterprogramm **forward** deklariert, das dann das Overlay-Unterprogramm aufruft.

18.6.3 Rekursion

Overlay-Unterprogramme können nicht rekursiv sein. Diese Einschränkung kann umgangen werden, indem ein normales, rekursives Unterprogramm deklariert wird, das dann das Overlay-Unterprogramm aufruft.

18.6.4 Laufzeit-Fehler

Laufzeit-Fehler, die in Overlays entstehen, werden wie gewöhnlich gefunden und durch das Fehler-Behandlungssystem wird eine Adresse ausgegeben. Diese Adresse ist jedoch eine innerhalb des Overlaybereichs und es gibt keine Möglichkeit zu erfahren welches Overlay-Unterprogramm tatsächlich aktiv war, als der Fehler entstand.

Laufzeit-Fehler in Overlays können daher nicht immer ohne weiteres mit der Menüwahl 'Find run-time error' gefunden werden. Durch 'Find run-time error' wird das erste Auftreten des Codes an der spezifizierten Adresse aufgezeigt. Das **kann** natürlich die Fehlerstelle sein, aber der Fehler kann ebensogut in einem späteren Unterprogramm innerhalb derselben Overlaygruppe entstanden sein.

Dies stellt aber keine ernste Beschränkung dar, da die Art und Entstehungsweise häufig daraufhin deutet, in welchem Unterprogramm sich der Fehler ereignete. Um den Fehler präzise zu lokalisieren, müssen Sie das möglicherweise fehlerhafte Unterprogramm als erstes in der Overlay-Gruppe plazieren. Durch 'Find run-time error wird dann der Fehler gefunden.

Am besten ist es, Unterprogramme solange nicht in Overlays zu plazieren, bis diese vollständig überprüft worden sind.

Anmerkungen:

19. IBM PC EXTRAS

Dieses Kapitel betrifft nur die PC-DOS / MS-DOS Versionen. Von den hier beschriebenen Funktionen kann nur beim IBM PC und Kompatiblen erwartet werden, daß sie funktionieren! Wenn Sie Probleme mit einem kompatiblen Rechner haben, dann ist dieser doch nicht so kompatibel, wie Sie dachten.

Kontrolle des Bildschirmmodus
TURBO stellt eine Anzahl von Prozeduren bereit, um die verschiedenen PC Bildschirmmodi zu kontrollieren.

Fenster
Die Fenster-Routinen ermöglichen es Ihnen, einen Teil des Bildschirms als Ihren momentanen Arbeitsbereich zu deklarieren, und damit den Rest des Bildschirms vor Überschreibung zu schützen.

Grund-Graphik
Diese eingebauten Graphikroutinen ermöglichen Ihnen das Drucken von Punkten und Zeichnen von Linien in verschiedenen Farben.

Erweiterte Graphik
Eine Reihe externer Graphikroutinen erweitert noch die Graphikmöglichkeiten. Durch eine einfache Anweisung werden diese Routinen in Ihre Programme aufgenommen.

Turtle - Graphik
Dieselbe externe Assemblerdatei liefert Ihnen die Turtle-Graphikroutinen.

Sound
Es werden Standardprozeduren bereitgestellt, die es Ihnen möglich machen, die PC Soundeigenschaften auf einfache Weise zu nutzen.

Tastatur
Eine Anzahl spezieller Tasten auf dem IBM Terminal sind für den Editor als primäre Befehle installiert. Diese Befehle sind auf Seite 186 aufgeführt und wenn Sie wollen, können Sie noch andere hinzufügen. Die sekundären *WordStar* Befehle sind dennoch verfügbar.

19.1 Kontrolle des Bildschirmmodus

Der IBM PC stellt Ihnen eine Anzahl von Bildschirmmodi zur Verfügung, von denen jeder eine eigene Charakteristik aufweist. Einige stellenZeichen, andere Graphiken dar und alle haben unterschiedliche Möglichkeiten der Farbwiedergabe. TURBO Pascal 3.0 unterstüzt alle Bildschirmformate und ermöglicht ihre bequeme Verwendung.

Die folgenden Bildschirm-Modi sind verfügbar:

TextMode
25 Zeilen mit 40 oder 80 Zeichen

GraphColorMode
320x200 Punkte Farbgraphik

GraphMode
320x200 Punkte schwarz/weiß Graphik (Farbe auf einem RGB Monitor)

HiRes
640x200 Punkte schwarz + einfarbige Graphik

19.1.1 Text-Modi

Im Textmodus zeigt der PC 25 Zeilen mit 40 oder 80 Zeichen. Die Prozedur, um diesen Modus anzufordern, heißt *TextMode* und wird folgendermaßen aufgerufen:

```
TextMode;
TextMode(BW40);     BW40 ist eine integer Konstante mit dem Wert 0
TextMode(BW80);     BW80 ist eine integer Konstante mit dem Wert 2
TextMode(C40);      C40 ist eine integer Konstante mit dem Wert 1
TextMode(C80);      C80 ist eine integer Konstante mit dem Wert 3
```

Das erste Beispiel ohne Parameter fordert den Textmodus an, der zuletzt aktiv war, oder der gegenwärtig aktiv ist. Die nächsten beiden Beispiele aktivieren schwarz/weiß Textmodus mit 40 oder 80 Zeichen pro Zeile. Die letzten beiden Beispiele aktivieren farbigen Textmodus mit 40 oder 80 Zeichen pro Zeile. Aufruf von *TextMode* löscht den Bildschirm.

TextMode sollte vor Beendigung eines Graphikprogramms aufgerufen werden, um das System auf Textmodus zurückzusetzen.

19.1.2 Farb-Modi

In den Farbtext-Modi kann jedes Zeichen in einer der 16 Farben gewählt werden, der Hintergrund kann in einer von acht Farben sein. Die 16 verfügbaren Farben werden durch Zahlen von 0 bis 15 benannt. Um die Sache einfacher zu machen, enthält TURBO Pascal 3.0 16 vordefinierte, *integer* Konstanten, die verwendet werden können, um die Farben namentlich zu bezeichnen:

Dunkle Farben	Helle Farben
0:Black (Schwarz)	**08**:DarkGray (Dunkles Grau)
1:Blue (Blau)	**09**:LightBlue (Helles Blau)
2:Green (Grün)	**10**:LightGreen (Helles Grün)
3:Cyan (Türkis)	**11**:LightCyan (Helles Türkis)
4:Red (Rot)	**12**:LightRed (Helles Rot)
5:Magenta (Lila)	**13**:LightMagenta (Pink)
6:Brown (Braun)	**14**:Yellow (Gelb)
7:LightGray (Helles Grau)	**15**:White (Weiß)

Tabelle 19-1: Textmodus Farbskala

Zeichen können in jeder dieser Farben dargestellt werden, der Hintergrund kann eine der dunklen Farben sein. Manche Monitore erkennen aus irgendeinem Grund das Intensitätssignal, das die acht hellen Farben erzeugt, nicht. Auf solchen Monitoren werden die acht hellen Farben, als ihre dunklen äquivalente dargestellt

19.1.2.1 TextColor

Syntax: TextColor(*Color*);

Durch diese Prozedur wird die **Zeichen**-Farbe gewählt. *Color* ist ein *integer* Ausdruck im Bereich von 0 bis 15, der die Zeichenfarben aus der obigen Tabelle auswählt.

Beispiel:

```
TextColor(1);          wählt blaue Zeichen
TextColor(Yellow);     wählt gelbe Zeichen
```

Die Zeichen können durch Hinzufügen der Zahl 16 zu der Farbnummer zum Blinken gebracht werden. Es gibt dafür auch die vordefinierte Konstante *Blink*:

```
TextColor(Red + Blink);     Rot mit blinkenden Zeichen
```

19.1.2.2 TextBackGround

Syntax: TextBackGround(*Color*);

Durch diese Prozedur wird die **Hintergrund** (engl: background) Farbe gewählt, d.h. die Umgebungsfläche jedes Zeichens; der gesamte Bildschirm besteht aus 25 Zeilen mit 40 oder 80 Zeichen. *Color* ist ein *integer* Ausdruck im Bereich von 0 bis 8, der die Zeichenfarben aus der obigen Tabelle auswählt.

Beispiele:

`TextBackground(4);`	wählt roten Hintergrund
`TextBackground(Magenta)`	wählt lila Hintergrund

19.2 Cursorposition

Im Textmodus erlauben Ihnen zwei neue Funktionen, festzustellen, wo der Cursor auf dem Bildschirm positioniert ist:

19.2.1 WhereX;

Syntax: WhereX;

Diese *integer* Funktion gibt die X-Koordinate der gegenwärtigen Cursorposition aus.

19.2.2 WhereY;

Syntax: WhereY;

Diese *integer* Funktion gibt die Y-Koordinate der gegenwärtigen Cursorposition aus.

19.3 Graphik-Modi

Mit der Standard IBM Graphikkarte, oder einer kompatiblen, erstellt TURBO Graphiken. Es werden drei Graphikmodi unterstützt:

GraphColorMode
320x200 Punkte Farbe

GraphMode
320x200 Punkte schwarz/weiß

HiRes
640x200 Punkte schwarz + eine Farbe

Die obere, linke Ecke des Bildschirms ist die Koordinate 0,0. X Koordinaten erstrecken sich nach rechts, Y Koordinaten nach unten. Zeichnungen werden bei Überschreitung des Bildschirmrandes abgeschnitten; d.h. alles außerhalb des Bildschirms wird ignoriert (außer die Turtle-Graphikprozedur *Wrap* ist in Kraft).

Die Aktivierung eines Graphikmodus löscht den Bildschirm. Die Standardprozedur *ClScr* arbeitet nur im Textmodus, so daß die einzige Möglichkeit einen Bildschirm mit Graphik zu löschen ist, einen Graphikmodus zu aktivieren, dies kann auch der gerade verwendete sein. Bei erweiterter Graphik und Turtle-Graphik gibt es jedoch die Prozedur *ClearScreen*, die das aktive Fenster löscht.

Graphiken können mit Text kombiniert werden. In dem 320 x 200 Modus kann der Bildschirm 40 x 25 Zeichen, in dem 640 x 200 Modus kann er 80 x 25 Zeichen anzeigen.

Die Prozedur *TextMode* sollte vor Beendigung eines Graphikprogramms aufgerufen werden, um das System auf Textmodus zurückzusetzen (siehe auf Seite 160).

19.3.1 GraphColorMode

Syntax: GraphColorMode;

Diese Standardprozedur aktiviert den 320x200 Punkte Farbgraphikbildschirm und erlaubt Ihnen X Koordinaten zwischen 0 und 319, und Y Koordinaten zwischen 0 und 199 anzugeben. Für Zeichnungen können die von einer Palette gewählten Farben (Seite 165 beschrieben) verwendet werden.

19.3.2 GraphMode

Syntax: GraphMode;

Diese Standardprozedur aktiviert den 320 x 200 Punkte schwarz/weiß Graphikbildschirm; Sie können X Koordinaten zwischen 0 und 319, und y Koordinaten zwischen 0 und 199 angeben. Auf einem RGB Monitor, wie dem IBM Color/Graphik Display, stellt dieser Modus auch Farben einer begrenzten Palette dar, wie auf Seite 166 gezeigt.

19.3.3 HiRes

Syntax: HiRes;

Diese Standardprozedur aktiviert den 640 x 200 Punkte hochauflösenden Graphikbildschirm; Sie können X Koordinaten zwischen 0 und 639, und Y Koordinaten zwischen 0 und 199 angeben. Bei hochauflösender Graphik ist der Hintergrund (Bldschirm) immer schwarz und Sie zeichnen in einer Farbe, die Sie durch die *HiResColor* Standardprozedur angeben.

19.3.4 HiResColor

Syntax: HiResColor(*Color*);

Diese Standardprozedur wählt die Zeichenfarbe bei hochauflösender Graphiken. *Color* ist ein *integer* Aausdruck im Bereich von 0 bis 15. Der Hintergrund (Bildschirm) ist immer schwarz. Eine Veränderung von *HiResColor* bewirkt eine Änderung des vorhandenen Bildschirms auf die neue Farbe.

Beispiele:

```
HiResColor(7);          wählt helles Grau
HiResColor(Blue);       wählt Blau
```

Die eine mögliche Farbe kann aus den folgenden 16 Farben gewählt werden:

Dunkle Farben	Helle Farben
0:Black (Schwarz)	08:DarkGray (Dunkles Grau)
1:Blue (Blau)	09:LightBlue (Helles Blau)
2:Green (Grün)	10:LightGreen (Helles Grün)
3:Cyan (Türkis)	11:LightCyan (Helles Türkis)
4:Red (Rot)	12:LightRed (Helles Rot)
5:Magenta (Lila)	13:LightMagenta (Pink)
6:Brown (Braun)	14:Yellow (Gelb)
7:LightGray (Helles Grau)	15:White (Weiß)

Tabelle 19-2: Farbskala der hochauflösenden Graphik

Manche Monitore erkennen das zur Erzeugung der acht hellen Farben benutzte Intensitätssignal nicht. Auf solchen Monitoren werden die hellen Farben durch ihre dunklen Äquivalente dargestellt.

19.3.5 Palette

Syntax: Palette(*N*);

Diese Prozedur aktiviert die Farbpalette, die durch den *integer* Ausdruck *N* indiziert wird; die Nummer der Palette wird mit einen Parameter spezifiziert. Es gibt vier solcher Paletten, die jeweils drei Farben (1-3) enthalten und zusätzlich eine vierte Farbe (0), die immer gleich dem Hintergrund ist (Erläuterung dazu später):

Farbzahl:	0	1	2	3
Palette 0	Hintergrund	Grün	Rot	Braun
Palette 1	Hintergrund	Türkis	Violett	helles Grau
Palette 2	Hintergrund	helles Grün	helles Rot	Gelb
Palette 3	Hintergrund	helles Türkis	Pink	Weiß

Tabelle 19-3: Farbpaletten der Farbgraphik

Die Graphikroutinen verwenden die Farben aus dieser Palette. Sie werden mit einem Parameter, der im Bereich von 0 bis 3 liegt aufgerufen und die tatsächlich benutzte Farbe wird von der aktiven Palette gewählt:

Plot(X,Y,2)	wird ein roter Punkt geplottet, wenn Palette 0 aktiviert ist.
Plot(X,Y,3)	wird ein gelber Punkt geplottet, wenn Palette 2 aktiviert ist.
Plot(X,Y,0)	es wird ein Punkt in den aktiven Farbhintergrund geplottet, wodurch dieser Punkt gelöscht wird.

Wenn eine Zeichnung auf dem Bildschirm ist, erzeugt eine Veränderung der Farbpalette eine Änderung aller Farben des Bildschirms in die Farben der neuen Palette. Gleichzeitig werden nur 3 Farben plus einer Hintergrundfarbe dargestellt.

Der *GraphMode* zeigt vermeintlich nur schwarz/weiß Graphiken an, aber auf einem RGB Monitor, wie dem IBM Color/Graphics Display stellt dieser Modus auch die folgende begrenzte Farbpalette dar:

Farbzahl:	0	1	2	3
Palette 0	Hintergrund	Blau	Rot	helles Grau
Palette 1	Hintergrund	helles Blau	helles Rot	Weiß

Tabelle 19-4: Farbpalette bei S/W Graphik

19.3.6 GraphBackground

Syntax: GraphBackground(*Color*);

Diese Standardprozedur setzt die Hintergrundfarbe, d.h. der ganze Bildschirm kann durch den Aufruf der Standardprozedur *GraphBackground* mit einem *integer* Parameter im Bereich 0 bis 15 auf eine der 16 Farben gesetzt werden:

GraphBackground(0);	der Bildschirm erscheint in Schwarz
GraphBackground(11);	der Bildschirm erscheint in hellem Türkis

Die folgenden Farbnummern und vordefinierten Konstanten stehen zur Verfügung:

Dunkle Farben	Helle Farben
0:Black (Schwarz)	**08**:DarkGray (Dunkles Grau)
1:Blue (Blau)	**09**:LightBlue (Helles Blau)
2:Green (Grün)	**10**:LightGreen (Helles Grün)
3:Cyan (Türkis)	**11**:LightCyan (Helles Türkis)
4:Red (Rot)	**12**:LightRed (Helles Rot)
5:Magenta (Lila)	**13**:LightMagenta (Pink)
6:Brown (Braun)	**14**:Yellow (Gelb)
7:LightGray (Helles Grau)	**15**:White (Weiß)

Tabelle 19-5: Graphik Hintergrundfarbskala

Manche Monitore erkennen das zur Erzeugung der acht hellen Farben verwendete Intensitätssignal nicht. Auf solchen Monitoren werden die acht hellen Farben wie ihre acht dunklen Äquivalente dargestellt.

19.4 Fenster

TURBO Pascal erlaubt Ihnen Fenster (engl: window) überall auf dem Bildschirm zu deklarieren. Wenn Sie in solch ein Fenster schreiben, verhält sich das Fenster genau so, als ob Sie den gesamten Bildschirm benutzen würden und läßt den Rest des Bildschirms unberührt.

19.4.1 Text-Fenster

Die Prozedur *Window* erlaubt Ihnen jeden Bereich des Bildschirms als aktives Fenster zu definieren:

```
Window(X1,Y1,X2,Y2);
```

wobei X1 und Y1 die absoluten Koordinaten der linken oberen Ecke des Fenster sind und X2 und Y2 die absoluten Koordinaten der rechten unteren Ecke. Die minimale Größe der Textfenster ist 2 Spalten und 2 Zeilen.

Das voreingestellte Fenster ist 1,1,80,25 im 80 Spaltenmodus und 1,1,40,25 im 40 Spaltenmodus, d.h. der ganze Bildschirm.

Alle Bildschirmkoordinaten (mit Ausnahme der Fenster-Koordinaten selbst) sind immer relativ zu dem aktiven Fenster. Das heißt, daß die Anweisung:

```
Window(20,8,60,17);
```

den zentralen Bereich Ihres physikalischen Bildschirms als aktives Fenster definiert; die Bildschirmkoordinaten 1,1 (obere linke Ecke) sind nun die obere linke Ecke des Fensters, nicht des physikalischen Bildschirms.

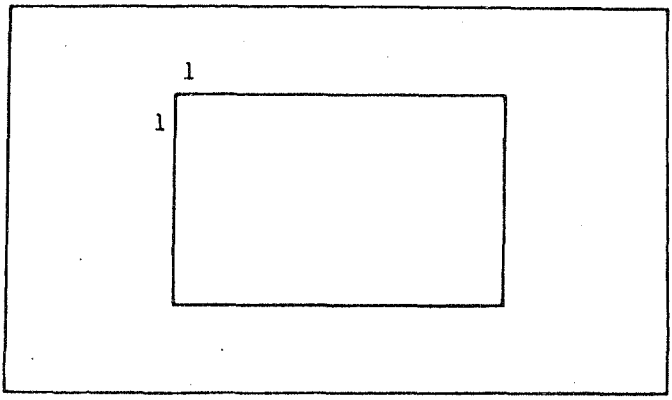

Bild 19-1: Text-Fenster

Der Bildschirm außerhalb des Fensters steht einfach nicht zur Verfügung und das Fenster verhält sich, als ob es der ganze Bildschirm wäre. Sie können einfügen, löschen, Zeilen rollen lassen. Diese rollen nach unten weiter, wenn sie zu lang sind.

19.4.2 Graphik-Fenster

Die Prozedur *GraphWindow* erlaubt Ihnen jeden Bereich des Bildschirms als aktives Fenster in einem beliebigen Graphik-Modus zu definieren:

```
GraphWindow(X1,Y1,X2,Y2);
```

wobei X1 und Y1 die absoluten Koordinaten der linken oberen Ecke des Fensters sind und X2 und Y2 die absoluten Koordinaten der rechten unteren Ecke.

Das voreingestellte Graphik-Fenster ist im 320x200 Punkte Modus

```
GraphWindow(0,0,319,199);
```

und im 640x200 Punkte Modus

```
GraphWindow(0,0,639,199);
```

d.h. der ganze Bildschirm.

Graphiken werden bei Fenstern abgeschnitten, wenn Sie über den Rand hinausgehen: Sie können zwischen zwei Koordinaten außerhalb des Fensters eine Linie zeichnen, und nur der Teil der Linie, der innerhalb des Fensters liegt, wird dargestellt:

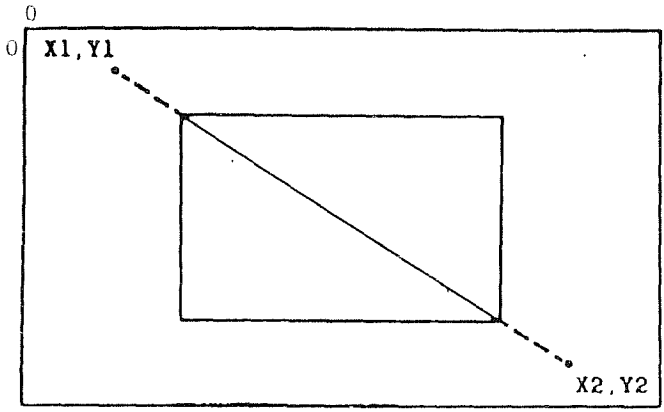

Bild 19-2: Graphik-Fenster

19.5 Grund-Graphik

In jedem der Graphikmodi stellt TURBO Pascal Standardprozeduren bereit, die an den angegebenen Koordinaten Punkte darstellen und zwischen zwei Koordinaten eine Linie zeichnen:

19.5.1 Plot

Syntax: Plot(*X,Y,Color*);

Stellt auf dem Bildschirm an den angegebenen Koordinaten einen Punkt dar, der durch *X und Y* festgelegt ist; die Farbe wird durch *Color* spezifiziert. *X,Y* und *Color* sind *integer* Ausdrücke.

19.5.2 Draw

Syntax: Draw(*X1,Y1,X2,Y2,Color*);

Verbindet auf dem Bildschirm die Koordinaten, die durch *X1,Y1* und *X2,Y2* bestimmt sind, durch eine Linie; die Farbe wird durch *Color* spezifiziert. Alle Parameter sind *integer* Ausdrücke.

19.6 Erweiterte Graphik

TURBO Pascal beinhaltet eine Reihe von Assembler-Routinen, die Sie bei der Compilierung in Ihre TURBO Programme einfügen können. Sie bieten Ihnen die erweiterten Graphik-Kommandos, die im Folgenden beschreiben werden.

Die externen Graphik-Routinen sind in der Datei GRAPH.BIN enthalten. In der Datei GRAPH.P befinden sich die notwendigen **external** Deklarierungen. Die Routinen der erweiterten Graphik werden einfach durch Verwendung folgender Anweisung zum Einbinden der Datei GRAPH.P in Ihr Programm aufgenommen:

```
$I GRAPH.P |
```

19.6.1 ColorTable

Syntax: ColorTable(*C1,C2,C3,C4*);

ColorTable bietet die Funktion von *Palette*, durch die Definition einer Übersetzungstabelle. So kann die gegenwärtige Farbe eines gegebenen Punkts zur Bestimmung der neuen Farbe dieses Punkts dienen, wenn dieser nochmals geschrieben wird. Der voreingestellte Farbtafelwert ist (0,1,2,3), d.h. wenn ein Punkt auf den Bildschirm geschrieben wird, ändert er die bereits existierende Farbe nicht:

Farbe 0 bleibt Farbe 0
Farbe 1 bleibt Farbe 1
Farbe 2 bleibt Farbe 2
Farbe 3 bleibt Farbe 3

Die Farbtafel (3,2,1,0) bewirkt, daß

Farbe 0 zu Farbe 3 wird
Farbe 1 zu Farbe 2 wird
Farbe 2 zu Farbe 1 wird
Farbe 3 zu Farbe 0 wird

d.h. alle Farben werden umgekehrt. Die Prozedur *PutPic* verwendet immer die Farbtafel, alle anderen Zeichen-Prozeduren verwenden die Tafel, falls die Farbangabe -1 auftritt, z.B.:

```
Plot(X,Y,-1);
```

19.6.2 Arc

Syntax: Arc(*X,Y,Winkel,Radius,Farbe*);

Zeichnet einen Bogen von *Winkel* Grad, beginnend an der Position *X,Y* mit dem angegebenen *Radius*. Ist der *Winkel* positiv, dann läuft der Bogen in Uhrzeiger-Richtung, bei negativem Winkel in umgekehrter Richtung. Die Angabe der *Farbe* zwischen 0 und 3 wählt die Stiftfarbe von der Farbpalette (siehe Seite 165), bei Angabe von -1, wird die Farbe von der Farbenübersetzungstafel, die durch die Prozedur *ColorTable* definiert ist (siehe Seite 172), gewählt.

19.6.3 Circle

Syntax: Circle(*X,Y,Radius,Farbe*)

Zeichnet einen Kreis in der durch *Farbe* gegebenen Farbe, dessen Mittelpunkt *X,Y* ist. Der Radius wird durch *Radius* bestimmt.

Der Radius des Kreises ist in horizontaler und vertikaler Achsenrichtung gleich. Im 320 x 200 Graphikmodus wird dadurch ein perfekter Kreis gezeichnet, da der Bildschirm fast linear ist. Im 640 x 200 Modus erscheint der Kreis jedoch als Elipse verzerrt.

Die Angabe der *Farbe* zwischen 0 und 3 wählt die Stiftfarbe von der Farbpalette (siehe Seite 165), bei Angabe von -1, wird die Farbe von der Farbenübersetzungstafel, die durch die Prozedur *ColorTable* definiert ist (siehe Seite 172), gewählt.

19.6.4 GetPic

Syntax: GetPic(*Puffer,X1,Y1,X2,Y2*);

Kopiert den Inhalt einer rechteckigen Fläche, die durch die *integer* Ausdrücke *X1,Y1,X2,Y2* definiert wird, in di Variable *Puffer*, die von beliebigem Typ ist. Die minimal, erforderliche Puffergröße in Bytes zur Speicherung des Bildes läßt sich folgendermaßen berechnen:

320 x 200 Modus:
Puffergröße = ((Breite + 3) div 4) * Höhe * 2 + 6

640 x 200 Modus:
Puffergröße = ((Breite + 7) div 8) * Höhe + 6

wobei:

Breite = abs(x1 - x2) + 1 und Höhe = abs(1 - y2) + 1

Beachten Sie, daß es in der Verantwortlichkeit des Programierers liegt, eine für die ganze Übertragung ausreichende Puffergöße bereitzustellen.

Die ersten 6 Bytes des Puffers bestehen aus einem 3 Wort langem Kopf (3 *integer* Zahlen). Nach der Übertragung enthält das erste Wort eine 2, falls der 320 x 200 Modus, oder eine 1, wenn der 640 x 200 Modus gewählt war. Das zweite Wort enthalt die Breite des Bildes, das dritte seine Höhe. Die restlichen Bytes enthalten Daten. Die Daten der linkesten Pixels werden in den signifikantesten Bytes gespeichert. Am Ende jeder Zeile werden die restlichen Bits des letzten Bytes übersprungen.

9.6.5 PutPic

Synatx: PutPic(*Puffer,X,Y*);
Kopiert den Inhalt von *Puffer* in eine rechteckige Fläche auf de Bildschirm. Die *integer* Ausdrücke *X, Y* definieren die untere, linke Ecke der Bildfläche. *Puffer* ist eine Variable beliebigen Typs, in der zuvor durch *GetPic* ein Bild gespeichert wurde. Jedes Bit im Puffer wird entsprechend der Farbkarte in die richtige Farbe umgewandelt, bevor es auf den Bildschirm geschrieben wird.

19.6.6 GetDotColor

Syntax: GetPoint(*X,Y*);

Diese *integer* Funktion gibt den Farbwert des an der Koordinate *X,Y* befindlichen Punkts aus. Werte von 0 bis 3 können im 320 x 200 Punkte-Modus, und von 0 oder 1 im 640 x 200 Punkte-Modus ausgegeben werden. Wenn *X,Y* außerhalb des Fensters liegt, gibt *GetDotColor* den Wert -1 aus.

19.6.7 FillScreen

Syntax: FillScreen(*Farbe*);

Füllt das ganze aktive Fenster mit der durch den *integer* Ausdruck angegebenen *Farbe*. Wenn Farbe als 0 bis 3 spezifiziert ist, wird sie von der Palette gewählt, wenn sie -1 ist, wird die Farbtafel verwendet. Damit sind dramatische Effekte möglich. Ist die Farbtafel beispielsweise *3, 2, 1, 0* und *FillScreen(-1)* wird das ganze Bild innerhalb des aktiven Fensters invertiert.

19.6.8 FillShape-Prozedur

Syntax: FillShape(*X,Y,FüllFarbe,RandFarbe*);

Füllt eine Fläche von beliebiger Form mit einer durch einen *integer* Ausdruck angegebenen Farbe, die im Bereich 0 bis 3 liegen muß. Die Farbübersetzungstafel wird nicht unterstützt. Die Form muß vollständig von der *RandFarbe* eingeschlossen sein. Ist dies nicht der Fall, läuft die *FüllFarbe* auch auf Bereiche außerhalb der Form über. *X* und *Y* sind die Koordinaten eines Punktes innerhalb des zu füllenden Bildes.

19.6.9 FillPattern

Syntax: FillPattern(*X1,Y1,X2,Y2,Farbe*);

Füllt ein Rechteck, das durch die Koordinaten *X1, Y1, X2, Y2* mit dem Muster, das in der Prozedur *Pattern* definiert wurde. Das Muster wird sowohl horizontal als auch vertikal wiedergegeben, um die ganze Fläche zu füllen.

Bits mit dem Wert 0 bleiben auf dem Bildschirm unverändert, während Bits des Werts 1 als Punkt in der gewählten *Farbe* erscheinen.

19.6.10 Pattern

Syntax: Pattern(*P*);

Definert ein Muster (engl: pattern), das von der Prozedur *FillPattern* verwendet wird. Das Muster ist eine 8 x 8 Matrix, die durch den *P* Parameter definiert wird. Dieser muß den Typ **array**|0..7| **of** *Byte* besitzen. Jedes Byte entspricht einer horizontalen Linie des Musters und jedes Bit einem Pixel. Das folgende Beispiel zeigt ein Muster und den hexadezimalen Wert jeder Zeile der Matrix. Ein Bindestrich entspricht einer binären 0, ein Stern einer binären 1.

```
-*---*--  $44        *-*-*-*-  $AA
*---*---  $88        -*-*-*-*  $55
---*---*  $11        *-*-*-*-  $AA
--*---*-  $22        -*-*-*-*  $55
-*---*--  $44        *-*-*-*-  $AA
*---*---  $88        -*-*-*-*  $55
---*---*  $11        *-*-*-*-  $AA
--*---*-  $22        -*-*-*-*  $55
```

Um das erste Muster zu verwenden, die schrägen Linien, könnten die folgenden typisierten Konstanten deklariert und als Parameter an *Pattern* übergeben werden:

```
const
    Lines: array|0..7| of Byte =
                (($44,$88,$11,$22,$44,$88,$11,$22);
```

Wenn ein Muster von der Prozedur *FillPattern* verwendet wird, erzeugen niedrige Bits (0) keine Veränderung der Bildschirmanzeige, hohe Bits (1) verursachen das Schreiben eines Punkts.

19.7 Turtle-Graphik

Die externe Datei GRAPH.BIN, die die im vorigen Abschnitt besprochenen, erweiterten Graphikroutinen enthält, beinhaltet außerdem die TURBOTurtle-Graphikroutinen. Deshalb steht Ihnen durch das Einfügen der Datei GRAPH.P, die die Graphik-Deklarierungen enthält, auch die Turtle-Graphik zur Verfügung:

```
| $I GRAPH.P |
```

Die Turtle-Graphik wird im Folgenden beschreiben.

TURBO Turtle-Graphik basiert auf dem 'turtle' Konzept das von S. Papert und seinen Mitarbeitern am MIT erarbeitet wurde. Es erlaubt auch denjenigen einen leichten Umgang mit Graphiken, denen das Verständnis von cartesischen Koordinaten schwer fällt. Papert u.a. hatten die Idee, eine Schildkröte (engl: turtle) eine Strecke oder einen Bogen gehen und dadurch eine Linie zeichen zu lassen. Einfache Algorithmen erlauben in diesem System interessantere Bilder zu gestalten, als ähnliche Algorithmen derselben Länge im cartesischen Koordinatensystem.

Wie die anderen Graphikroutinen, arbeiten Turtle-Graphikroutinen innerhalb eines Fensters. Das Fenster nimmt laut Voreinstellung den ganzen Bildschirm ein, kann aber durch die Prozeduren *Window* oder *TurtleWindow* auch verkleinert werden. Die Graphik ist auf die Fensterfläche beschränkt, der restliche Teil des Bildschirm wird nicht überschrieben. Turtle-Graphik und normale Graphik können gleichzeitig in demselben Fenster benutzt werden.

Die TURBO Turtle-Graphik arbeitet mit *Turtle-Koordinaten*. Die Ausgangsposition (engl: *home position*;0,0) der Schildkröte dieses Koordinatensystems ist immer die Mitte des aktiven Fensters. Positive Werte erstrecken sich nach rechts (*X*) und oben (*Y*), negative Werte nach links (*X*) und unten (*Y*):

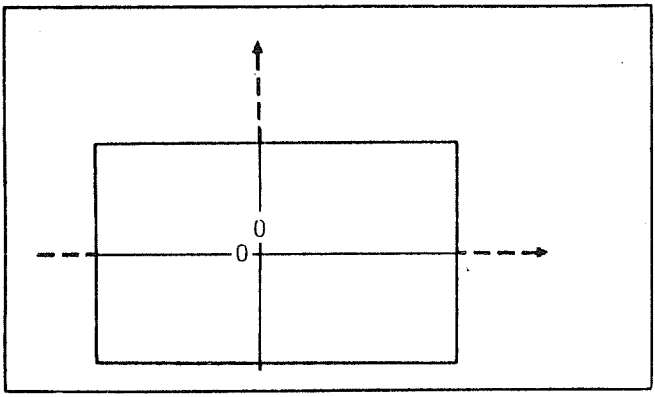

Bild 19 - 3: Turtle-Koordinaten

Der Bereich der Koordinaten des ganzen Bilschirms ist:

320 x 200 Modi: X = -159..0..160, Y = -99..0..100
640 x 200 Modus: X = -319..0..320, Y = -99..0..100

Der aktuelle Bereich wird durch die Größe des aktiven Fensters begrenzt. Koordinaten außerhalb des aktiven Fensters sind zulässig, werden aber ignoriert. Das heißt, daß Zeichnungen an den Rändern des aktiven Fensters abgeschnitten werden.

19.7.1 Back

Syntax: Back(*Dist*);

Bewegt die Schildkröte rückwärts um die *Distanz*, die durch den *integer* Ausdruck *Dist* angegeben ist. Die Bewegung erfolgt von der gegenwärtigen Position der Schildkröte ab, entgegen ihrer momentanen Richtung, dabei wird eine Linie in der aktuellen Stiftfarbe gezeichnet (wenn *Dist* negativ ist, bewegt sich die Schildkröte vorwärts).

19.7.2 ClearScreen

Syntax: ClearScreen;

Diese Prozedur löscht das aktive Fenster und setzt die Schildkröte auf ihre Ausgangsposition.

19.7.3 Forwd

Syntax: Forwd(*Dist*);

Bewegt die Schildkröte um die Distanz vorwärts, die durch den *integer* Ausdruck *Dist* gegeben ist. Die Bewegung erfolgt von ger gegenwärtigen Position der Schildkröte ab, dabei wird eine Linie in der aktuellen Stiftfarbe gezeichnet (wenn *Dist* negativ ist, bewegt sich die Schildkröte rückwärts).

19.7.3 Heading

Syntax: Heading;

Die Funktion *Heading* gibt eine *integer* Zahl im Bereich 0..359 aus, die die Richtung anzeigt, in welcher sich die Schildkröte gerade bewegt. 0 ist aufwärts, zunehmende Winkel entsprechen einer Bewegung in Uhrzeiger-Richtung.

19.7.4 HideTurtle

Syntax: HideTurtle;

Verdecken der Schildkröte, so daß sie nicht auf dem Bildschirm gesehen werden kann. Das entspricht dem Anfangszustand. Sie müssen, um die Schildkröte sichtbar zu machen, erst die Prozedur *ShowTurtle* aufrufen.

19.7.4 Home

Syntax: Home;

Diese Prozedur setzt die Schildkröte auf ihre Ausgangsposition an die Turtle-Koordinaten (0,0) zurück (die Mitte des aktiven Fensters) und läßt sie aufwärts in Richtung 0 zeigen.

19.7.5 NoWrap

Syntax: NoWrap;

Diese Prozedur verhindert, daß die Schildkröte auf der anderen Seite des aktiven Fensters wieder auftaucht, wenn die Grenzen des Fensters überschritten werden *NoWrap* ist die Voreinstellung des Systems.

19.7.6 PenDown

Syntax: PenDown;

Diese Prozedur bewegt den Stift nach unten, so daß die Schildkröte bei einer Bewegung eine Linie zeichnet. Das ist der Anfangsstatus des Stifts.

19.7.7 PenUp

Syntax: PenUp;

Diese Prozedur hebt den Stift an, so daß sich die Schildkröte bewegen kann, ohne zu zeichnen.

19.7.8 SetHeading

Syntax: SetHeading(*Winkel*);

Setzt die Schildkröte in die Richtung, die durch den *integer* Ausdruck *Winkel angegeben wird. 0 bedeutet aufwärts, und zunehmende Winkel entsprechen einer Drehung in Uhrzeiger-Richtung. Wenn Winkel* nicht im Bereich 0..359 liegt, wird er in eine Zahl innerhalb dieses Bereichs umgewandelt.

Es sind vier *integer* Konstanten vordefinert, um das Setzen der Schildkröte in die vier Hauptrichtungen zu erleichtern. *North* = 0 (aufwärts), *East* = 90 (rechts), *South* = 180 (abwärts) und *West* = 270 (links).

19.7.9 SetPenColor

Syntax: SetPenColor(*Farbe*);

Wählt die Farbe des Stifts, das heißt, die Farbe in der bei der Bewegung der Schildkröte gezeichnet wird. *Farbe* ist ein *integer* Ausdruck mit einem Wert zwischen -1 und 3. Wenn *Farbe* zwischen 0 und 3 liegt, wird die Farbe des Stifts von der Farbpalette gewählt (siehe Seite 165), bei -1, von der Farbübersetzungstafel, die in der Prozedur *ColorTabel* definert ist (siehe Seite 172).

19.7.10 SetPosition

Syntax: SetPosition(*X,Y*);

Bewegt die Schildkröte auf die Koordinaten, die durch den *integer* Ausdruck *X* und *Y* gegeben sind, ohne eine Linie zu zeichnen.

19.7.11 ShowTurtle

Syntax: ShowTurtle;

Zeigt die Schildkröte als kleines Dreieck an. Die Schildkröte ist anfangs nicht sichtbar. Damit die Schildkröte sichtbar wird, müssen Sie die Prozedur *ShowTurtle* aufrufen.

19.7.12 TurnLeft

Syntax: TurnLeft(*Winkel*);

Dreht die Schildkröte um *Winkel* Grad von ihrer gegenwärtigen Position. Positive Winkel drehen die Schildkröte nach links, negative Winkel drehen sie nach rechts.

19.7.13 TurnRight

Syntax: TurnRight(*Winkel*);

Dreht die Schildkröte um *Winkel* von ihrer gegenwärtigen Position. Positive Winkel drehen die Schildkröte nach rechts, negative drehen sie nach links.

19.7.13 TurtleWindow

Syntax: TurtleWindow(*X,Y,B,H);*

Die Prozedur TurtleWindow definiert einen Bereich des Bildschirms als aktives Fenster in einem der Graphikmodi. Sie entspricht exakt der Prozedur *Window*. *TurtleWindow* läßt Sie jedoch das Fenster in Turtle-Koordinaten definieren, die bei der Arbeit mit Turtle-Graphik selbstverständlicher sind. *X* und *Y* sind die Bildschirmkoordinaten der Fenstermitte; *W* ist die Fensterbreite und *H* seine Höhe.

Das voreingestellte *TurtleWindow* ist 159,99,320,200 im 320 x 200 Punkte-Modus und 319,99,640,200 im 640 x 200 Punkte-Modus, d.h. der ganze Bildschirm. Wenn das Turtle-Fenster über den physischen Bildschirm hinaus definiert wird, wird es an den Rändern des physischen Bildschirm abgeschnitten.

Turtle-Graphiken werden auf das aktive Fenster beschnitten, das heißt, wenn Sie versuchen die Schildkröte außerhalb des aktiven Fensters zu bewegen, wird sie am Rand gestoppt.

Wenn das Fenster festgelegt ist (entweder durch *TurtleWindow* oder *Window*), wird die Schildkröte auf die Ausgangsposition und -richtung gesetzt. Eine Änderung des Bildschirmmodus bewirkt ein Wiederherstellen der Voreinstellung, d.h. der ganze Bildschirm gilt als Fenster.

Turtle-Graphiken arbeiten mit *Turtle-Koordinaten*. Die Ausgangsposition (0,0) dieses Koordinatensystems ist immer die Mitte des aktiven Fensters. Positive Werte erstrecken sich nach rechts (X) und oben (Y), negative nach links (X) und unten (Y).

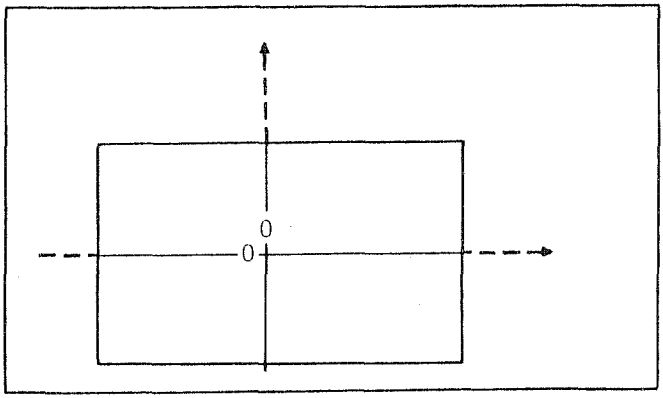

Bild 19 - 4: Turtle-Koordinaten

Der Bereich der Koordinaten des ganzen Bildschirms ist:

320 x 200 Modi: X = -159..0..160, Y = -99..0..100
640 x 200 Modus: X = -319..0..320, Y = -99..0..100

Der aktulle Bereich wird durch die Größe des aktiven Fensters begrenzt. Koordinaten außerhalb des aktiven Fensters sind zulässig, werden aber ignoriert. Das heißt, daß Zeichnungen an den Rändern des aktiven Fensters abgeschnitten werden, alles außerhalb gezeichnete geht verloren.

19.7.14 TurtleThere

Syntax: TurtleThere;

Diese *boolean* Funktion gibt *True* aus, wenn die Schildkröte im aktiven Fenster sichtbar ist (nach einem Aufruf von *ShowTurtle*), sonst gibt sie *False* aus.

19.7.15 TurtleDelay

Syntax: TurtleDelay(*Ms*)
;

Diese Prozedur sorgt für eine Verzögerung um (*Ms*) Millisekunden zwischen jedem Schritt der Schildkröte. Voreinstellung ist keine Verzögerung.

19.7.15 Wrap

Syntax: Wrap;

Nach einem Aufruf dieser Prozedur erscheint die Schildkröte wieder auf der entgegengesetzten Seite des aktiven Fensters, wenn die Fensterränder überschritten werden. Mit *NoWrap* erhalten Sie wieder den normalen Zustand.

19.7.16 Xcor

Syntax: Xcor;

Diese Funktion gibt den *integer* Wert der gegenwärtigen X-Koordinate der Schildkröte aus.

19.7.17 Ycor

Syntax: Ycor;
Diese Funktion gibt den gegenwärtigen Wert der Y-Koordinate der Schildkröte aus.

9.8 Sound

Der Lautsprecher des PC wird durch die Standardprozedur *Sound* angesprochen:

```
Sound(I);
```

wobei *I* ein *integer* Ausdruck ist, der die Frequenz in Hertz angibt. Die spezifizierte Frequenz wird solange ausgesendet, bis der Lautsprecher durch einen Aufruf der *NoSound* Standardprozedur ausgeschaltet wird:

```
NoSound;
```

Das folgende Beispielprogramm sendet einen 440-Hertz Ton eine halbe Sekunde aus:

```
begin
Sound(440);
Delay(500);
NoSound;
end.
```

19.9 Editier-Tasten

Zusätzlich zu den *WordStar* Kommandos wurden die Editiertasten der IBM PC Tastatur als erste Kommandos implementiert. Das bedeutet, daß zwar Ctrl-E, Ctrl-X, Ctrl-S und Ctrl-D immernoch den Cursor aufwärts, abwärts, nach links und rechts bewegen, Sie aber auch die Pfeiltasten der numerischen Tastatur verwenden können. Die folgende Tabelle gibt einen Überblick über die vorhandenen Editiertasten, ihre Funktionen und ihre *WordStar* äquivalenten Kommandos:

Wirkung	**PC-Taste**	**Kommando**
Zeichen n. links	linker Pfeil	Ctrl-S
Zeichen n. rechts	rechter Pfeil	Ctrl-D
Wort links	Ctrl-linker Pfeil	Ctrl-A
Wort rechts	Ctrl-rechter Pfeil	Ctrl-F
Zeile n. oben	aufwärts Pfeil	Ctrl-E
Zeile n. unten	abwärts Pfeil	Ctrl-X
Seite n oben	PgUp	Ctrl-R
Seite n. unten	PgDn	Ctrl-C
Rechst auf d. Zeile	Home	Ctrl-Q Ctrl-S
Links auf d. Zeile	End	Ctrl-Q Ctrl-D
An d. Anfang d. Seite	Ctrl-Home	Ctrl-Q Ctrl-E
An d. Ende d. Seite	Ctrl-End	Ctrl-Q Ctrl-X
An d. Anfang d. Datei	Ctrl-PgUp	Ctrl-Q Ctrl-R
An d. Ende d. Datei	Ctrl-PgDn	Ctrl-Q Ctrl-C
Einfügen-Modus an/aus	Ins	Ctrl-V
Markiere Blockanfang	F7	Ctrl-K Ctrl-B
Markiere Blockende	F8	Ctrl-K Ctrl-K
Tabulator	TAB	Ctrl-I

Tabelle 19 - 6: IBM-PC Editiertasten

Beachten Sie, daß wegen der Kompatibilität der Kommandos zu *WordStar*, manche Funktionstasten in *WordStar* und TURBO unterschiedliche Bedeutungen haben.

20. PC-DOS UND MS-DOS

Dieses Kapitel beschreibt die Features von TURBO Pascal, die nur den PC-DOS und MS-DOS Implementierungen zu eigen sind. Es bietet zwei Arten von Informationen:

1) Dinge, die man für eine effektive Anwendung von TURBO Pascal braucht. Diese finden sich auf den Seiten 187 bis 209.

2) Der Rest des Kapitels beschreibt Dinge, die nur für erfahrene Programmierer von Interesse sind, z.B. Assembler-Routinen, technische Details des Compilers, usw..

20.1 Directory-Baumstruktur

20.1.1 Hauptmenü

Die strukturierten Directories von DOS Version 2.0 werden von TURBO's Hauptmenü unterstützt. Bei der Aktivierung prüft TURBO die DOS-Versionsnummer, falls diese 2.0 oder später ist sieht das Hauptmenü folgendermaßen aus:

```
Logged drive: A
Active directory: X


Work file:
Main file:

Edit        Compile     Run      Save
Dir         Quit    compiler Options

Text:          0 bytes
Free: 62903 bytes

>
```

Bild 20-1 : TURBO Haupt-Menü

Beachten Sie, daß das zusätzliche **A** Kommando Ihnen erlaubt, die aktive Directory zu wechseln. Die Pfadbeschreibung erfolgt genauso wie bei dem DOS Kommando CHDIR. Die gegenwärtig aktive Directory wird nach dem Doppelpunkt angezeigt.

DOS benutzt den Backslash: X um sich auf die Standarddirectoy zu beziehen, wie man im Beispiel sieht. Die Namen der restlichen Directories entsprechen den Konventionen für Dateinamen. Sie dürfen 1 bis 8 Buchstaben lange Namen haben, wahlweise gefolgt von einem Punkt und einer bis zu drei Zeichen langen Dateitypbezeichnung. Jede Subdirectory kann normale Dateien oder weitere Subdirectorys enthalten.

Dateien können in diesem System von Directories durch einen *Pfad*-namen (engl:path name) zusätzlich zum Dateinamen angeprochen werden. Ein Pfadname besteht aus den Namen der Directorys, die zu der Datei führen. Diese werden durch Backslashs (rückwärtige Schrägstriche) getrennt. Die vollständige Bezugnahme auf die Datei INVADERS.PAS in der Directory TURBO lautet deshalb:

```
XTURBOXINVADERS.PAS
```

Der erste Backslash gibt an, daß der Pfad in der Standarddirectory beginnt. Wenn Sie eine andere Directory angemeldet hätten und in die Directory TURBO gelangen wollten, müßten Sie **A** drücken und folgendes eingeben:

```
XTURBO
```

In jeder Subdirectory sehen Sie zwei Einträge beim DIR-Listing: . und .. Der einzelne Punkt dient zur Identifikation dieser Directory als Subdirectory. Die zwei Punkte stehen als Markierung der Stamm-Directory dieser Subdirectory. Die zwei Punkte .. müssen in einem Directory-Pfad verwendet werden. Wenn Sie z.B. eine Subdirectory von TURBO angemeldet haben, und zu TURBO zurück wollen, geben Sie **A** ein und dann die zwei Punkte ...

20.1.2 Directory-Prozeduren

TURBO Pascal bietet die folgenden Prozeduren zur Manipulation der Directory-Baumstruktur von MS-DOS.

20.1.2.1 ChDir

Syntax: ChDir(*St*);

Ändert die aktuelle Directory in den Pfad, der durch den *string* Ausdruck *St* angegeben wird.

20.1.2.2 MkDir

Syntax: MkDir(*St*);

Erzeugt eine neue Subdirectory entsprechend dem Pfad, der in dem *string* Ausdruck *St* angegeben wird. Der letzte Name in dem Pfad darf noch nicht existieren.

20.1.2.3 RmDir

Syntax: RmDir(*St*);

Entfernt die Subdirectory, die durch den als *string* Ausdruck *St* angegebenen Pfad, spezifiziert wurde.

20.1.2.4 GetDir

Syntax: GetDir(*Dr,St*);

Gibt die aktuelle Directory des als *Dr* bezeichneten Laufwerks in der string Variablen *St* aus. *Dr* ist ein *integer* Ausdruck, wobei 0 = A, 1 = B, usw..

20.2 Compiler-Optionen

Das Kommando **O** wählt das folgende Menü an, in dem Sie einige voreingestellte Werte des Compilers sehen und verändern können. Es ist auch beim Finden von Laufzeit-Fehlern hilfreich.

```
compile -)        Memory
                  Com-file
                  cHn-file

commandline Parameter:

Find run-time error   Quit
```

Abbildung 20-2: Optionen-Menü

20.2.1 Memory / Com file / cHn file

Die drei Befehle **M**,**C** und **H** steuern den Compilermodus und die Ablage des erzeugten Object-Codes durch den Compiler. **M**emory ist der voreingestellte Modus. Der Code wird im Speicher erzeugt und behalten. Das Programm kann dann direkt vom Speicher aus durch den **R**un-Befehl ausgeführt werden.

Die **C**om-Datei wird durch Drücken von **C** gewählt. Der Pfeil bewegt sich dann auf diese Zeile. Der Compiler schreibt den Code auf eine Datei mit demselben Namen wie die Arbeitsdatei (oder Hauptdatei, falls angegeben). Der Dateityp ist .COM. Diese Datei enthält den Object-Code und die Pascal Library und kann durch Eintippen des Namens auf der Tastatur aktiviert werden.

Die c**H**ain-Datei wird durch Drücken von **H** gewählt. Der Pfeil bewegt sich dann auf diese Zeile. Der Compiler schreibt den Code auf eine Datei mit dem selben Namen wie die Arbeitsdatei (oder Hauptdatei, falls angegeben). Der Dateityp ist .CHN. Diese Datei enthält den Object-Code, aber keine Pascal Library und muß von einem anderen TURBO Pascal Programm aus mit der *Chain* Prozedur aktiviert werden (siehe Seite 193).

Wenn der **C**om- oder c**H**n Modus gewählt ist, erscheinen vier weitere Zeilen am Bildschirm:

```
minimum cOde segment size:      XXXX paragraphs (max. YYYY)
minimum Data segment size:      XXXX paragraphs (max. YYYY)
mInimum free dynamic memory:    XXXX paragraphs
mAximum free dynamic memory:    XXXX paragraphs
```

Abbildung 20-3: Speicherverwendungs-Menü

Der Gebrauch dieser Befehle wird in den folgenden Abschnitten beschrieben.

20.2.1.1 Minimale Codesegmentgröße

Der **O**-Befehl wird verwendet, um die minimale Größe des Codesegments für eine .COM Datei die *Chain* oder *Execute* benutzt, festzusetzen. Wie auf Seite 193 diskutiert, ändern *Chain* und *Execute* die Basisadresse der Code-, Daten- und Stacksegmente nicht. Die minimal festzulegende Größe richtet sich also nach dem längsten Segment in einer der über *Chain* oder *Execute* aufgerufenen Programme.

Folglich müssen Sie bei der Compilierung eines Hauptprogramms, den Wert der minimalen Codesegmentgröße mindestens auf denselben Wert setzen, den das größte Segment des mit *Chain* zu verkettenden Programms besitzt. Die erforderlichen Werte dazu erhält man durch die Statusanzeige, die jede Compilierung abschließt. Die Werte werden hexadezimal angegeben und spezifizieren die Zahl der Paragraphen; ein Paragraph entspricht 16 Bytes.

20.2.1.2 Minimale Datensegmentgröße

Der **D** Befehl wird verwendet, um die minimale Größe eines Datensegments für eine .Com Datei bei Benutzung von *Chain* oder *Execute* festzulegen. Die Regeln für die Festsetzung der Größe sind dieselben wie oben .

Folglich müssen Sie bei der Compilierung eines Hauptprogramms, den Wert der minimalen Datensegmentgröße mindestens auf denselben Wert setzen, den das größte Segment des mit *Chain* zu verkettenden Programms besitzt. Die erforderlichen Werte dazu erhält man durch die Statusanzeige, die jede Compilierung abschließt. Die Werte werden hexadezimal angegeben und spezifizieren die Zahl der Paragraphen; ein Paragraph entspricht 16 Bytes.

20.2.1.3 Minimaler freier dynamischer Speicher

Dieser Wert gibt die minimale Speichergröße an, die für Stack und Heap benötigt werden. Der Wert wird hexadezimal angegeben und spezifiziert die Zahl der Paragraphen; ein Paragraph entspricht 16 Bytes.

20.2.1.4 Maximaler freier dynamische Speicher

Dieser Wert spezifiziert die maximale Speichergröße, die dem Stack und Heap zugewiesen ist. Es muß in Programmen, die in einer Mehrplatzumgebung laufen, verwendet werden, um sicherzustellen, daß das Programm nicht den ganzen freien Speicher belegt. Der Wert ist hexadezimal und gibt die Zahl der Paragraphen an. Ein Paragraph entspricht 16 Bytes.

20.3 Kommandozeilen-Parameter

Das **P** Kommando erlaubt Ihnen, einen oder mehrere Parameter einzugeben, die an Ihr Programm übergeben werden, wenn Sie es im Memory-Modus laufen lassen. Dies geschieht genauso, als ob Sie diese in der DOS Kommandozeile eingegeben hätten. Die Parameter können mit den Funktionen *ParamCount* und ParamStr angesprochen werden.

20.4 Finden von Laufzeit-Fehlern

Wenn Sie ein Programm laufen lassen, das im Speicher compiliert wurde und es tritt ein Fehler auf, wird der Editor aufgerufen und der Fehler wird automatisch angezeigt. Das ist natürlich nicht möglich, wenn das Programm eine .COM oder .CHN Datei ist. Fehlermeldungen geben den Fehlercode und den Wert des Programmzählers bei Auftreten des Fehlers an, wie z.B.:

```
Run-time error 01, PC=1B56
Program aborted
```

Abbildung 20-4: Fehlermeldung

Um im Quelltext die Stelle zu finden, an der der Fehler aufgetreten ist, ist der Befehl **F** einzugeben. Wenn dann nach der Adresse gefragt wird, gibt man die in der Fehlermeldung genannte Adresse ein:

```
Enter PC: 1B56
```

Abbildung 20-5: Orten des Fehlers

Die Stelle ist nun im Source geortet und so angezeigt, wie der Fehler beim Programmablauf im Speicher aufgetreten war.

Beachten Sie, daß bei Verwendung von Overlays die Lokalisierung von Fehlern im Programm etwas komplizierter sein kann, als auf Seite 196 beschrieben.

20.5 Standardbezeichner

Die folgenden Standardbezeichner gelten nur für die PC/MS-DOS Version:

CSeg	LongFilePos	MemW	PortW
DSeg	LongFileSize	MsDos	SSeg
Intr	LongSeek	Ofs	Seg

20.6 Chain und Execute

TURBO Pascal enthält die zwei Prozeduren *Chain* und *Execute*, die es ermöglichen, daß TURBO Programme andere TURBO Programme aktivieren. Die Syntax der Prozeduraufrufe ist:

Chain(*FilVar*)
Execute(*FilVar*)

Dabei ist *FilVar* eine Dateivariable beliebigen Typs, die zuvor mit der Standardprozedur *Assign* einer Diskettendatei zugewiesen wurde. Wenn die Datei existiert, wird sie in den Speicher geladen und ausgeführt.

Die *Chain* Prozedur wird nur verwendet, um spezielle TURBO Pascal .CHN Dateien zu aktivieren, d.h. Dateien, die mit der cHn Option des Optionenmenüs compiliert wurden (siehe Seite 190) Eine solche Datei enthält nur Programmcode, keine Pascal Library; sie benutzt die schon im Speicher befindliche Pascal Library.

Die *Execute* Prozedur wird benutzt, um eine beliebige TURBO Pascal .COM Datei zu aktivieren.

Wenn die Diskettendatei nicht existiert, tritt ein I/O Fehler auf. Dieser Fehler wird, wie auf Seite 116 beschrieben, behandelt. Wenn der Compilerbefehl passiv ist ({$I-}), geht die Programmausführung mit der Anweisung weiter, die der fehlgeschlagenen *Chain* oder *Execute* Anweisung folgt. Die *I/O result*-Funktion muß vor weiteren I/O Operationen aufgerufen werden.

Daten können vom laufenden Programm aus, auf zu verkettende Programme entweder durch gemeinsame, globale Variablen (*shared global variables*) oder durch absolute Variablen (*absolute address variables*) übertragen werden.

Um Überlappungen zu vermeiden, müssen gemeinsame, globale Variablen als die ersten Variablen in beiden Programmen deklariert werden. Sie müssen in beiden Deklarationen in der gleichen Reihenfolge aufgelistet sein. Weiterhin müssen beide Programme auf die gleiche Größe von Code und Datensegmenten compiliert werden (siehe Seite 191). Wenn diese Bedingungen erfüllt sind, werden die Variablen von beiden Programmen an dieselbe Adresse im Speicher plaziert. Da TURBO Pascal seine Variablen nicht automatisch initialisiert, können sie gemeinsam genutzt werden.

Beispiele:
Programm MAIN.COM:

```
programm Main;
var
  Txt:      string[80];
  CntPrg:     file;

begin
  Write('Enter any text: '); Readln(Txt);
  Assign(CntPrg, 'ChrCount.chn');
  Chain(CntPrg);
end.
```

Programm *CHRCOUNT.CHN:*

```
program ChrCount;
var
  Txt:      string[80];
  NoOfChar,
  NoOfUpc,
  I:          Integer;

begin
  NoOfUpc := 0;
  NoOfChar := Length(Txt);
  for I := 1 to length(Txt) do
    if Txt[I] in ['A'..'Z'] then NoOfUpc := Succ(NoOfUpc);
  Write('No of characters in entry: ', NoOfChar);
  Writeln('. No of upper case characters: ', NoOfUpc,'.');
end.
```

Wenn Sie bestimmen wollen, ob ein TURBO Programm entweder durch eXecute oder direkt von der Befehlszeile aufgerufen wurde, sollten Sie eine **absolute** Variable mit der Basisadresse *Cseg:*$80 verwenden. An dieser Adresse steht das Längenbyte der Befehlszeile und wenn ein Programm von DOS aufgerufen wird, enthält es einen Wert zwischen 0 und 127. Bei der Ausführung eines Programms sollte deshalb das aufrufende Programm diese Variable auf einen höheren Wert als 127 setzen. Wenn Sie dann die Variable in dem aufgerufenen Programm prüfen, bedeutet ein Wert zwischen 0 und 127, daß das Programm von DOS aufgerufen wurde; ein höherer Wert besagt, daß es von einem anderen TURBO Programm aufgerufen wurde.

Die Anwendung von *chain* und *execute* bei TURBO Programmen verändert das Speicher-Layout nicht. Die Basisadresse und die Größe des Codes, der Daten und des Stacksegments werden nicht verändert; *Chain* und *Execute* ersetzen lediglich den Programm-Code in dem Codesegment. 'Fremde' Programme können deshalb von einem TURBO Programm nicht aufgenommen werden.

Es ist unbedingt erforderlich, daß das erste Programm, das eine *Chain* Aweisung ausführt, genügend Speicher für den Code, die Daten und die Stacksegmente zuweist, um das größte .CHN Programm unterzubringen. Dies geschieht, indem man im Optionenmenü den minimalen Platz für Code, Daten und freien Speicher angibt (siehe Seite 190).

Beachten Sie, daß weder *Chain* noch *Execute* in direktem Modus verwendet werden können, d.h. beispielsweise von einem Programm, das mit auf **M**emory geschalteter Compileroption läuft (siehe auch Seite 190).

20.7 Overlays

Während der Ausführung erwartet das System normalerweise seine Overlaydateien im angemeldeten Laufwerk und der aktuellen Directory zu finden. Die Prozedur *OvrPath* kann verwendet werden, um diesen voreingestellten Wert zu ändern.

20.7.1 OvrPath-Prozedur

Syntax: OvrPath(*Path*);

Path stellt einen *string* Ausdruck dar, der einen Subdirectory-Pfad spezifiziert (siehe Seite 188 zur Erklärung von DOS Directory-Pfaden) Beim Aufruf von Overlaydateien, werden die Dateien in der angegebenen Directory erwartet. Wenn einmal eine Overlaydatei in einer Directory eröffnet wurde, wird in Zukunft dieselbe Datei in der gleichen Directory gesucht. Der Pfad kann wahlfrei ein Laufwerk spezifizieren (*A:,B:,etc.*).

Die gegenwärtige Directory wird an einem einzelnen Punkt erkannt. *OvrPath('.')* verursacht also, daß die Overlaydateien in der gegenwärtigen Directory gesucht werden.

Beispiel:

```
program OvrTest;

overlay procedure ProcA;
begin
  WriteLn('Overlay A');
end;

overlay procedure ProcB;
begin
  WriteLn('Overlay B');
end;

procedure Dummy;
begin
  {Dummy Prozedur zur Trennung der Overlays in zwei Gruppen}
end;

overlay procedure ProcC;
begin
  WriteLn('Overlay C');
end;
```

```
begin
  OvrPath('Xsubl');
  ProcA;
  OvrPath('.');
  ProcC;
  OvrPath('Xsubl');
  ProcB;
end.
```

Der erste Aufruf von *OvrPath* spezifiziert Overlays, die in der Subdirectory X*sub1* gesucht werden. Der Aufruf von *ProcA* bewirkt daher, daß die erste Overlaydatei (die die beiden Overlayprozeduren *ProcA* und *ProcB* enthält) in dieser Directory eröffnet wird.

Als nächstes bestimmt die *OvrPath('.')* Anweisung, daß die folgenden Overlays in der gegenwärtigen Directory zu finden sind. Der Aufruf von *ProcC* eröffnet hier die zweite Overlaydatei.

Die folgende *ProcB* Anweisung ruft eine Overlayprozedur in der ersten Overlaydatei auf; um sicherzustellen, daß diese in der X*sub1* Directory gesucht wird, muß vor dem Aufruf die *OvrPath('Xsub1')* Anweisung ausgeführt werden.

20.8 Dateien

20.8.1 Dateinamen

Ein Dateiname in DOS besteht aus einem Pfad mit Dateinamen, die, bis zur gewünschten Directory durch Querstriche getrennt werden, und dem nachfolgenden aktuellen Dateinamen:

Laufwerk:X*Dirname*X...X*Dirname*X*Dateiname*

Falls der Pfad mit einem Querstrich beginnt, wird in der Standarddirectory gestartet, andernfalls im angemeldeten Laufwerk.

Die *Laufwerk*- und Pfad-Spezifikation ist wahlfrei. Wenn sie weggelassen wird, wird angenommen, daß die Datei im angemeldeten Laufwerk ist.

Der *Dateiname* besteht aus einem Namen von acht Buchstaben oder Zahlen, wahlweise mit einem Punkt dahinter, und einem Dateityp aus 1 bis 3 Buchstaben oder Zahlen.

20.8.2 Zahl offener Dateien

Die Zahl von Dateien, die zur selben Zeit offen sein können, wird durch den Compilerbefehl **F** kontrolliert. Die Voreinstellung ist {$F16}, was bedeutet, daß bis zu 16 Dateien gleichzeitig offen sein können. Wenn z.B. ein {$F24} Befehl an den Anfang des Programms (**vor** dem Deklarierungsteil) gesetzt wird, können gleichzeitig bis zu 24 Dateien offen sein. Der Compilerbefehl F begrenzt die Zahl der Dateien, die in einem Programm deklariert werden können nicht; er begrenzt lediglich die Zahl der Dateien, die zur gleichen Zeit offen sein können.

Beachten Sie: Obwohl der Compilerbefehl F benutzt wurde, um ausreichend Dateiplatz anzuweisen, kann dennoch ein 'too many open files' Fehler entstehen, wenn im Betriebssystem nicht genügend Dateipuffer zur Verfügung stehen. Falls das passiert, sollten Sie einen höheren Wert für den 'files=xx' Parameter in der Datei CONFIG.SYS wählen. Der voreingestellte Wert ist gewöhnlich 8. Weitere Einzelheiten darüber erfahren Sie in Ihrer MS-Dos Dokumentation.

20.8.3 Erweiterte Dateigröße

Die folgenden drei zusätzlichen Dateiroutinen erlauben es, einem erweiterten Bereich von Records in DOS Platz zu geben. Diese sind:

die Funktion *LongFileSize*
die Funktion *LongFilePosition* und
die Prozedur *LongSeek*

Sie korrespondieren mit ihren *integer* Äquivalenten *FileSize*, *FilePosition* und *Position*, arbeiten aber mit reellen Zahlen. Die Funktionen geben deshalb Ergebnisse im Typ *Real* aus. Der zweite Parameter der *LongSeek* Prozedur muß ein Ausdruck vom Typ *Real* sein.

20.8.4 File of Byte

In den CP/M Implementierungen muß der Zugriff auf nicht TURBO-Dateien (außer Textdateien) durch untypisierte Dateien geschehen, da die ersten beiden Bytes von TURBO-Dateien immer die Zahl der Komponenten in einer Datei enthalten. Dies ist in DOS Versionen nicht der Fall, eine nicht TURBO-Datei kann deshalb als **file of byte** deklariert werden und es kann direkt mit *Seek*, *Read* und *Write* darauf zugegriffen werden.

20.8.5 Flush-Prozedur

Die Prozedur *Flush* hat im DOS keine Wirkung, da DOS Dateivariable keinen Sektorpuffer verwenden.

20.8.6 Truncate-Prozedur

Syntax: Truncate(*FilVar*);

Diese Prozedur schneidet die Datei, an der gegenwärtigen Postion des Dateizeigers, bestimmt durch *FilVar*, ab; d.h., daß alle Records jenseits des Dateizeigers weggeschnitten werden. *Truncate* bereitet also die anschließende Ausgabe der Datei vor.

20.8.7 Textdateien

20.8.7.1 Puffergröße

Die voreingestellte Puffergröße ist bei Textdateien 128 Bytes. Das ist für die meisten Anwendungen angemessen; Programme, die viele Ein/Ausgaben vornehmen (z.B. Kopierprogramme), profitieren jedoch von einem größeren Puffer, da dadurch die Kopfbewegungen im Laufwerk reduziert werden.

Deshalb ermöglicht Ihnen eine Option die Angabe der Puffergröße, wenn Sie eine Textdatei deklarieren:

```
VAR   TextFile:   Text[$800];
```

deklariert eine Textdateivariable mit einer Puffergröße von 2 KBytes.

20.8.7.2 Append-Prozedur

Syntax: Append(*FilVar*);

Die Diskdatei, die der Dateivariablen *FilVar* zugewiesen wurde, ist geöffnet und der Dateizeiger hat sich zum Dateiende bewegt. Die einzige Operation, die nach *Append* erlaubt ist, ist das Anfügen neuer Komponenten.

20.8.7.3 Flush-Prozedur

Bei Verwendung von Textdateien bewirkt die Prozedur *Flush* eine Entleerung des Dateipuffers.

20.8.7.4 Logische Geräte

Folgende zusätzliche logische Geräte werden bereitgestellt:

INP:
Verweist auf die MS-DOS Standard-Eingabedatei (Standard-Kennzeichnungsnr. 0)

OUT:
Verweist auf die MS-DOS Standard-Ausgabedatei (Standard-Kennzeichnungsnr. 1).

ERR:
Verweist auf die MS-DOS Standard-Fehlerausgabedatei (Standard-Kennzeichnungsnr. 2).

Diese Geräte können mit typisierten oder nicht-typisierten Dateien verwendet werden.

Das MS-DOS Betriebssystem bietet selbst eine Anzahl von logischen Geräten, wie beispielsweise 'CON', 'LST' und 'AUX'. TURBO Pascal behandelt diese Geräte, als ob sie Diskettendateien wären, mit einer Ausnahme: Wenn eine Textdatei durch *Reset, Rewrite* oder *Append* geöffnet ist, fragt TURBO Pascal MS-DOS nach dem Status der Datei. Wenn MS-DOS meldet daß es sich bei der Datei um ein logisches Gerät handelt, macht TURBO Pascal das Puffern, das normalerweise bei Textdateien stattfindet unmöglich, und alle I/O Operationen in der Datei, werden Zeichen für Zeichen ausgeführt.

Die Compileroption D kann diese Überprüfung aufheben. Der voreingestellte Status der Option D ist |$D+|; in diesem Status wird die Geräteüberprüfung durchgeführt. Im Status |$D-| wird keine Überprüfug vorgenommen und alle I/O Operation werden gepuffert. In diesem Fall stellt ein Aufruf der Standardprozedur Flush sicher, daß die Zeichen, die Sie in eine Datei geschrieben haben, auch tatsächlich dort hin gesandt wurden.

20.8.7.5 I/O Umleitung

PC/MS-DOS TURBO Pascal unterstützt die Möglichkeit der I/O Umleitung des MS-DOS Betriebssystems. Kurz gesagt ermöglicht Ihnen die I/O Umleitung, Diskettendateien als Standard-Eingabequelle und/oder als Standard-Ausgabeziel zu benutzen. Weiterhin kann ein Programm, das I/O Umleitung unterstützt als *Filter* in einem *Kanal* (engl: pipe) benutzt werden. Einzelheiten uber I/O Umleitung, Filter und Kanäle sind in der MS-DOS Dokumentation zu finden.

Die I/O Umleitung erfolgt durch die Compilerbefehle **G** (get) und **P** (put). Der Befehl G kontrolliert die Eingabedatei und P die Ausgabedatei. Beide Befehle, G und P verlangen ein Argument, das die Größe des Ein- oder Ausgabepuffers definiert. Die Voreinstellungen sind |$G0| und |$P0|, und damit verweisen *Input* und *Output* entweder auf *Con:* oder *TRM:*

Wenn ein Eingabepuffer ungleich Null definiert wurde, z.B. |$G256|, dann verweist die Standarddatei *Input* auf die MS-DOS Standard-Eingabekennzeichnung. Dementsprechend verweist die Standarddatei *Output* (z.B. |$P1024|) auf die MS-DOS Standard-Ausgabekennzeichnung. Der Compilerbefehl D (siehe Seite 201) ist auf solche *Input-* und *Output*dateien mit einem Puffer ungleich Null anwendbar. Um überhaupt eine Wirkung zu erzielen, müssen die Compilerbefehle P und G am Anfang des Programms d.h. vor dem Deklarationsteil plaziert werden.

Das folgende Programm demonstriert I/O Umleitung. Es liest Zeichen aus der Standarddatei *Input* ein, zählt jedes alphabetische Zeichen (A-Z) und gibt einen Graph der Häufigkeitsverteilung an die Standarddatei *Output* aus:

```
{$G512,P512,D-}
program CharacterFrequencyCounter
const
  Bar       = #223;
var
  Count:      array[65..90] of Real;
  Ch:         Char;
  I,Graph:    Integer;
  Max,
  Total:      Real;
begin
  Max := 0; Total := 0;
  for I := 65 to 90 do Count[I] := 0;
  while not EOF do
  begin
    Read(Ch);
    if Ord(Ch) > 127 then Ch := Chr(Ord(Ch)-128);
    Ch := UpCase(Ch);
    if Ch in ['A'..'Z'] then
    begin
      Count[Ord(Ch)] := Count[Ord(Ch)] +1;
      if Count[Ord(Ch)] > Max then Max := Count[Ord(Ch)];
      Total := Total +1;
    end;
  end;
  WriteLn('       Count       %');
  for I := 65 to 90 do
  begin
    Write(Chr(I),':       ',Count[I]:5:0,
                Count[I]*100/Total:5:0,'  ');
    for Graph := 1 to Round(Count[I]*63/Max) do
      Write(Bar);
    WriteLn;
  end;
  WriteLn('Total', Total:5:0);
end.
```

Wenn das Programm in eine Datei, namens COUNT.COM compiliert wird, dann liest der MS-DOS Befehl

COUNT < TEXT.DOC > CHAR.CNT

die Datei TEXT.DOC und gibt den Graph an die Datei CHAR.CNT aus.

20.9 Absolute Variablen

Variablen können so deklariert werden, daß sie bei einer bestimmten Speicheradresse verbleiben, sie werden dann **absolute** Variablen genannt. Dies wird erreicht, indem man an die Variablendeklaration das reservierte Wort **absolute** anhängt, dem zwei *integer* Konstanten folgen müssen, die den Offset nennen, wo die Variable lokalisiert werden kann:

```
var
Abc: Integer   absolute $0000:$00EE;
Def: Integer   absolute $0000:$00F0;
```

Die erste Konstante spezifiziert eine Segmentbasisadresse, die zweite Konstante das Offset innerhalb dieses Segments. Die Standardbezeichner *CSeg* und *DSeg* können benutzt werden, um Variablen an absolute Adressen innerhalb eines Codesegments (**CSeg**) oder eines Datensegments (**DSeg**) zu plazieren:

```
Special: array[1..CodeSize] absolute CSeg:$05F3;
```

Absolute kann auch zur Überlagerung von Variablen benutzt werden. d.h., daß eine Variable an derselben Adresse beginnen soll wie eine andere Variable. Wenn dem Bezeichner einer Variablen oder eines Parameters **absolute** folgt, beginnt die neue Variable bei der Adresse dieses Variablen-Parameters.

Beispiel:

```
var
  Str: string[32];
  StrLen: Byte absolute Str;
```

Diese Deklaration bedeutet, daß die Variable *StrLen* bei derselben Adresse wie die Variable *Str* beginnen soll, und da das erste Byte einer Stringvariablen die Länge des Strings enthält, wird *StrLen* die Länge von *Str* enthalten. Beachten Sie, daß die Deklaration einer absoluten Variablen nur einen Bezeichner benennen darf.

Weitere Erläuterungen zur Platzzuweisung für Variablen finden Sie auf Seite 216.

20.10 Absolute Adressfunktionen

Mit Hilfe der folgenden Funktionen können Informationen über die Adressen von Programmvariablen und Systemzeigern abgerufen werden.

20.10.1 Addr

Syntax: Addr(*Name*)

Zeigt die Speicheradresse des ersten Bytes einer Variablen mit dem Bezeichner *Name*. Wenn *Name* ein Array ist, kann es indiziert werden, wenn *Name* ein Record ist, können einzelne Felder selektiert werden. Der einzugebende Wert ist ein 32 bit Zeiger, bestehend aus einer Segmentadresse und einem Offset.

20.10.2 Ofs

Syntax: Ofs(*Name*)

Nennt das Offset im Speichersegment, das vom ersten Byte der Variablen, Prozedur oder Funktion mit dem Bezeichner *Name* belegt wird. Wenn *Name* ein Array ist, kann es indiziert werden, wenn Name ein Record ist, können einzelne Felder selektiert werden. Es wird ein *integer* Wert ausgegeben.

20.10.3 Seg

Syntax: Seg(*Name*)

Nennt die Adresse des Segments, das das erste Byte einer Variablen, Prozedur oder Funktion mit der Bezeichner *Name* enthält. Wenn *Name* ein Array ist, kann es indiziert werden, ist *Name* ein Record, können einzelne Felder ausgewählt werden. Es wird ein *integer* Wert ausgegeben.

20.10.4 Cseg

Syntax: Cseg

Nennt die Basisadresse des Codesegments. Es wird ein *integer* Wert ausgegeben.

20.10.5 Dseg

Syntax: Dseg

Nennt die Basisadresse des Datensegments. Es wird ein *integer* Wert ausgegeben.

20.10.6 Sseg

Syntax: Sseg

Nennt die Adresse des Stacksegments. Es wird ein *integer* Wert ausgegeben.

20.11 Vordefinierte Arrays

TURBO bietet vier vordefinierte Arrays vom Typ *Byte* an, mit denen der CPU-Speicher und die Datenports angesprochen werden können. Sie heißen *Mem*, *MemW*, *Port* und *PortW*.

20.11.1 Mem Array

Mit Hilfe der vordefinierten Arrays *Mem* und *MemW* kann der Speicher angesprochen werden.
Jede Komponente des *Mem*-Arrays belegt ein Byte, jede Komponente des Arrays *Wmem* belegt ein Wort (zwei Bytes, zuerst LSB). Der Index muß eine Adresse sein, die die Segmentbasisadresse und ein Offset nennt, die beide durch einen Doppelpunkt getrennt und *integer* sein müssen.

Die folgende Anweisung ordnet den Wert des Bytes in Segment 0000 mit dem Offset $0081 der Variablen *Value* zu:

```
Value := Mem[0000:$0081];
```

während die folgende Anweisung:

```
MemW Seg(Var):Ofs(Var):=Value;
```

den Wert der *integer Variablen Value* dort im Speicher plaziert, wo die ersten zwei Bytes der Variablen *Var* angesiedelt sind.

20.11.2 Port-Array

Port und *PortW* ermöglichen das Ansprechen der Datenports der 8086/88 CPU. Jedes Element des Arrays repräsentiert einen Datenport, wobei der Index den Datenportnummern entspricht. Da Datenports von 16-bit Adressen angewählt werden, ist der Indextyp *integer*. Wenn einer Komponente von *Port* oder *PortW* ein Wert zugeordnet wird, wird dieser an das genannten Port ausgegeben. Bezieht sich ein Ausdruck auf eine Portkomponente, wird deren Wert von dem genannten Port eingelesen. Die Komponenten des Arrays *Port* sind vom Typ *Byte*, die Komponenten von *PortW integer*.

Beispiel:

```
Port'56':=10;
```

Der Gebrauch des Portarrays beschränkt sich auf Zuordnung und Referenz in Ausdrücken, das heißt, daß Komponenten von *Port* und *PortW* nicht als Variablenparameter von Prozeduren und Funktionen benutzt werden können. Darüberhinaus sind keine Operationen zugelassen, die sich auf das gesamte Portarray beziehen (Referenz ohne Index).

20.12 With-Anweisungen

With-Anweisungen können in bis zu maximal 9 Stufen geschachtelt, angegeben werden.

20.13 Hinweise zu Zeigern

20.13.1 MemAvail

Mit der Standardfunktion *MemAvail* ist es jederzeit möglich, zu bestimmen, wieviel Raum im Heap verfügbar ist. Das Ergebnis ist *integer*, es nennt die Zahl der im Heap zur Verfügung stehenden Paragraphen (ein Paragraph umfaßt 16 Bytes).

20.13.2 Zeigerwerte

In bestimmten Fällen kann es von Interesse sein, einer Zeigervariablen einen bestimmten Wert zuzuordnen, **ohne eine andere Zeigervariable zu benutzen**, oder den wirklichen Wert einer Zeigervariablen zu ermitteln.

20.13.2.1 Zuweisung eines Werts zu einem Zeiger

Die Standardfunktion *Ptr* kann dazu benutzt werden, einem Zeiger bestimmte Werte zuzuordnen. Die Funktion gibt einem 32-bit Zeiger aus, der aus einer Segmentadresse und einer im Offset besteht.

Beispiel:

```
Pointer:=Ptr(Cseg,$80);
```

20.13.2.2 Ermittlung des Zeigerwerts

Ein Zeigerwert wird von einer 32-bit Einheit dargestellt und die Standardfunktion *Ord* kann deshalb **nicht** zur Wertermittlung eingesetzt werden. Stattdessen müssen die Funktionen *Ofs* und *Seg* benutzt werden.

Mit der folgenden Anweisung erhält man den Zeigerwert *P* (der aus einer Segmentadresse und dem Offset besteht):

```
SegmentPart:=Seg(P^);
OffsetPart:=Ofs(P^);
```

20.14 DOS-Funktionsaufrufe

Um DOS Systemaufrufe machen zu können, führt TURBO Pascal eine Prozedur *MsDos* ein, die einen Record als Parameter hat.

Details über DOS Systemaufrufe und BIOS Routinen finden sie im Technik-Handbuch des IBM-PC.

Der Parameter zu *MS-DOS* muß vom Typ:

```
record
  AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Integer;
end;
```

sein, oder im anderen Falle:

```
record
  case Integer of
    1: (AX,BX,CX,DX,BP,DI,SI,DS,ES,Flags: Integer);
    2: (AL,AH,BL,BH,CL,CH,DL,DH          : Byte);
  end;
```

Bevor TURBO den DOS Systemaufruf macht, lädt es die Register AX,BX,CX,DX,BP,SI,DI,DS, und ES mit den in den Recordparametern spezifizierten Werten. Wenn DOS die Operation beendet hat, setzt die *MsDos* Prozedur die Register auf den Record zurück und macht so die Erbebnisse von DOS verfügbar.

Das folgende Beispiel zeigt, wie ein MS-DOS Funktionsaufruf benutzt wird, um die Zeit von DOS zu erhalten:

```
procedure Timer(varHour,Min,Sec,Frac:Real);
type
  RegPack = record
              AX,BX,CX,DX,BP,DI,SI,DS,ES,Flags: Integer;
            end;
var
  Regs:                Regpack;
```

```
begin
  with Regs do
  begin
    AX := $2C00;
    MsDos(Regs);
    Hour := hi(CX);
    Min := lo(CX);
    Sec := hi(DX);
    Frac := lo(DX);
end;
      :
      :
      :
end; | procedure Timer |
```

20.15 Benutzergeschriebene I/O Treiber

Für manche Anwendungen ist es für den Programmierer von Nutzen seine eigenen Eingabe- und Ausgabetreiber zu definieren, d.h. Routinen, die Eingabe und Ausgabe von Zeichen auf/von externe/en Geräte/n ausführen. Die folgenden Treiber sind Teil von TURBO und werden von den Standard I/O Treibern benutzt (obwohl sie nicht als Standardprozeduren oder -funktionen verfügbar sind):

```
function  ConSt:boolean;|11|
function  ConIn:Char;|8|
procedure ConOut(Ch:Char);|2|
procedure LstOut(Ch:Char);|5|
procedure AuxOut(Ch:Char);|4|
function  AuxIn:Char;|3|
procedure UsrOut(Ch:Char);|2|
function  UsrIn:Char;|8|
```

Die *ConSt* Routine wird von der Funktion *KeyPressed* aufgerufen, die *ConIn* und *ConOut* Routinen werden von CON:, TRM: und KBD: Geräten benutzt, die *LstOut* Routine von dem LST: Gerät, die *AuxOut* und *AuxIn* Routinen werden vom AUX: Gerät und die *UsrOut* und *UsrIn* Routinen von dem USR: Gerät benutzt.

Laut Voreinstellung sind diese Treiber dem DOS System-Aufrufen zugewiesen, wie in den geschweiften Klammern in der obigen Liste der Treiber gezeigt.

Dies kann vom Programmierer geändert werden, indem die Adresse einer selbstdefinierten Treiberprozedur oder -funktion an eine der folgenden Standardvariablen zugewiesen wird.

Variable	**enthält die Adresse der**
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 Benutzer definierte Treiberprozedur oder -funktion muß den oben gegebenen Definitionen entsprechen, d.h. ein *ConSt* Treiber muß eine *Boolean* Funktion sein, ein *ConIn* Treiber muß eine *char* Funktion sein, usw..

20.16 Externe Unterprogramme

Das reservierte Wort **external** wird gebraucht, um externe Prozeduren und Funktionen aufzurufen, gewöhnlich Prozeduren und Funktionen, die in Assembler geschrieben sind.

Dem reservierten Wort **external** muß eine Stringkonstante folgen, die eine Datei benennt, in der der anzuwendende Assemblercode für die externe Prozedur oder Funktion enthalten sein muß. Der voreingestellte Dateityp ist *.COM*.

Während der Compilierung eines Programms, das externe Prozeduren oder Funktionen enthält, werden die entsprechenden Dateien geladen und in den Objectcode plaziert. Da es nicht möglich ist, in voraus zu wissen, wo genau im Objektcode der externen Code plaziert wird, **muß** dieser Code verschiebbar sein und es dürfen keine Bezüge zum Datensegment bestehen. Darüberhinaus muß der externe Code die Register BP, CS, DS und SS sichern und vor der Ausführung der RET Anweisung wiederherstellen.

Ein externes Unterprogramm hat keinen Block, d.h. keinen Deklarations- und keinen Anweisungsteil. Es wird nur der Programmkopf des Unterprogramms genannt, dem das reservierte Wort **external** unmittelbar folgen muß und der Dateiname, für das Unterprogramm.

Beispiel:
```
procedure DiskReset; external 'DSKRESET';
function IOstatus: boolean; external 'IOSTAT';
```

Eine externe Datei kann für mehrere Unterprogramme Codes enthalten. Das erste Unterprogramm wird wie oben beschrieben deklariert; die weiteren werden deklariert, indem der Bezeichner des ersten Unterprogramms spezifiziert wird, und darauf, in eckige Klammern gestellt, eine *integer* Konstante folgt, die ein Offset benennt. Die Einsprungstelle für jedes Unterprogramm ist die Adresse des ersten Unterprogramms, plus dem Offset.

Beispiel:

```
procedure Com1; external 'SERIAL.BIN';
function Com1Stat: Boolean; external Com1[3];
procedure Com1In: Char; external Com1[6];
procedure Com1Out: Char; external Com1[9];
```

Das obige Beispiel lädt die Datei SERIAL.BIN in das Programm und definiert vier Prozeduren: *Com1*, *Com1Stat*, *Com1In* und *Com1Out*; die Einsprungstelle ist an der Basisadresse des externen Codes, plus 0, 3, 6 und 9. Wenn eine externe Datei mehrere Unterprogramme enthält, ist der erste Teil des Codes typischerweise eine Sprungtabelle, wie in unserem Beispiel angenommen. Dadurch bleibt die Einsprungstelle des Unterprogramms unverändert, falls die externe Datei modifiziert wird.

Parameter können an externe Unterprogramme übergeben werden. Die Syntax ist genauso, wie bei Aufrufen von normalen Prozeduren und Funktionen:

```
procedure Plot(X,Y: Integer); external 'PLOT';
function QuickSort (var List: PartNo); external 'QS';
```

Die Übergabe externer Unterprogramme und Parameter wird auf Seite 221 weiter besprochen.

20.17 In-line Maschinencode

TURBO Pascal bietet die Möglichkeit zu **Inline** Anweisungen, mit Hilfe derer auf einfache Art Assembleranweisungen direkt in den Programmtext eingefügt werden können. Eine Inline-Anweisung setzt sich wie folgt zusammen: aus dem reservierten Wort **Inline** und einer oder mehreren *Codeelementen*, die durch Schrägstriche voneinander getrennt sind und in runde Klammern gesetzt sind.

Ein Codeelement besteht aus ein oder mehreren Datenelementen, getrennt durch die Zeichen plus(+) oder minus(-). Ein Datenelement ist entweder eine *integer* Konstante, ein Variablenbezeicher, ein Prozedurbezeichner, ein Funktionsbezeichner oder eine Kommandozählerreferenz. Eine Kommandozählerreferenz wird als Sternzeichen (*) geschrieben.

Beispiel:

inline (10/$2345/count+1/sort-*+2);

Jedes Codeelement erzeugt ein Byte oder ein Wort (zwei Bytes) Code. Der Byte- oder Wortwert wird berechnet, indem die Werte der Datenelemente, gemäß ihrer Trennungszeichen, entweder addiert oder subtrahiert werden. Der Wert des Variablenbezeichners ist die Adresse (oder Offset) der Variablen. Der Wert eines Prozedur- oder eines Funktionsbezeichners ist die Adresse (oder Offset) der Prozedur oder Funktion. Der Wert einer Kommandozählerreferenz ist die Adresse (oder Offset) des Kommandozählers, d.h. die Adresse, auf der das nächste Byte (Code) erzeugt werden soll.

Ein Codeelement erzeugt einen Code von einem Byte, wenn es nur aus *integer* Konstanten besteht und sein Wert innerhalb des 8-Bit Bereichs (0..255) liegt. Liegt der Wert nicht in diesem Bereich, oder verweist der Code auf Variablen-, Prozedur- oder Funktionsbezeichner, oder enthält das Codeelement eine Kommandozählerreferenz, dann wird ein Code von einem Wort erzeugt (zuerst das niederwertigste Byte).

Die '<' und '>' Zeichen können zur Überschreibung der automatischen Größenwahl, wie oben beschrieben, benutzt werden. Wenn ein Codeelement mit einem '<' Zeichen beginnt, wird lediglich das niederwertigste Byte des Wertes codiert, auch wenn es ein 16-Bit Wert ist. Wenn ein Codeelement mit einem '>' Zeichen beginnt, wird immer ein Wort codiert, auch wenn das höchse Byte null ist.

Beispiel:

inline (<$1234/>$44);

Diese **inline** Anweisung erzeugt einen Code von drei Bytes: $34, $44, $00.

Der Wert eines Variablenbezeichners, der in einer **inline** Anweisung benutzt wird, ist die Offsetadresse der Variablen innerhalb ihres Grundsegments. Das Grundsegment globaler Variablen (das sind im Hauptprogrammblock deklarierte Variablen) ist das Datensegment, das durch das DS Register zugänglich ist. Das Grundsegment lokaler Variablen (das sind innerhalb des gegenwärtigen Unterprogramms deklarierte Variablen) ist das Stacksegment, und in diesem Fall ist das Variablenoffset von dem BP (base page) Register abhängig, dessen Verwendung das Stacksegment automatisch auswählen läßt.

Das Grundsegment typisierter Konstanten ist das Codesegment, das durch das CS Register erreichbar ist. **inline** Anweisungen sollten nicht versuchen auf Variablen zuzugreifen, die weder im Hauptprogramm, noch im gegenwärtigen Unterprogramm deklariert sind.

Das folgende Beispiel einer Inline-Anweisung erzeugt Assemblercode, der alle Zeichen in seinem Stringargument in Großbuchstaben umwandelt.

```
procedure UpperCase(var Strg: Str);
|Str is type String|255||
begin
  inline
     ($C4/$BE/Strg/                 | LES   DI,Strg|BP||
     $26/$8A/$0D/                   | MOV   CL,ES:|DI||
     $FE/$C1/                       | INC   CL|
     $FE/$C9/                       | L1:   DEC   CL|
     $74/$13/                       | JZ    L2|
     $47/                           | INC   DI|
     $26/$80/$3D/$61/               | CMP   ES:BYTE PTR|DI|, 'a'|
     $72/$F5/                       | JB    L1|
     $26/$80/$3D/$7A/               | CPM   ES:BYTE PTR|DI|, 'z'|
     $77/$EF/                       | JA    L1|
     $26/$80/$2D/$20/               | SUB   ES:BYTE PTR|DI|, 20H|
     $EB/$E9);                      | JMP   SHORT L1|
                                    | L2:   |
end;
```

Inline-Anweisungen können im Anweisungsteil eines Blocks beliebig unter andere Anweisungen gemischt werden und sie können alle CPU-Register benutzen. **Beachten Sie** jedoch, daß der Inhalt der Register BP, SP, DS und SS nach Abschluß der Operation derselbe sein muß, wie zu Beginn.

20.18 Interrupt-Handhabung

Eine TURBO Pascal Interruptroutine muß die benutzten Register AX, BX, CX, DX, DI, DS und ES unverändert lassen. Dies geschieht, indem man die folgende Inline-Anweisung der Prozedur voranstellt:

inline ($50/$53/$51/$52/$56/$57/$1E/$06/$FB);

Das letzte Byte ($FB) ist eine STI Instruktion, die weitere Interrupts - erforderlich oder nicht - ermöglicht. Mit der folgenden Inline-Anweisung muß die Prozedur abgeschlossen werden:

inline ($07/$1F/$5F/$5E/$5A/$59/$5B/$58/$8B/$E5/$D5/$CF;

Dies setzt die Register wieder ein und lädt sowohl den Stackzeiger (SP), als auch das BP (base page) Register wieder. Das letzte Byte ($CF) ist eine IRET Instruktion ($CF), die die RET Instruktion, welche vom Compiler erzeugt wird, überschreibt.

Interruptprozeduren sollten keine I/O Operationen verwenden, die die Standardprozeduren und -funktionen von TURBO Pascal benutzen, da ein Zugriff auf das BDOS nicht mehrfach möglich ist. Der Programmierer muß den Interruptvektor initialisieren, mit dem die Interruptroutine aktiviert wird.

20.18.1 Intr Prozedur

Syntax: Intr(*InterruptNo,Result*)

Diese Prozedur ruft die Register und Flags entsprechend ihrer Nennung im Parameter *Result* auf. Der Parameter ist wie folgt typisiert:

```
Result = record
           AX,BX,CX,BP,SI,DI,DS,ES,Flags: Integer;
end;
```

So wird durch den Parameter *interruptNo*, der eine *integer* Konstante sein muß, ein Softwareinterrupt eingeleitet. Wenn die Interruptroutine beendet ist und Ihr Programm weiterabläuft, wird *Result* alle Werte enthalten, die von der Routine ermittelt wurden.

Beachten Sie, daß das Datensegmentregister DS, das für den Zugriff auf globale Variablen benutzt wird, einen falschen Wert erhält, wenn die Interrupt Serviceroutine eingegeben ist. Deshalb kann auf globale Variablen nicht direkt zugegriffen werden. *Typisierte Konstanten* sind jedoch zugänglich, da sie im Codesegment gespeichert sind. EineMöglichkeit auf globale Variablen in einer Interruptroutine zuzugreifen, ist deshalb das Speichern des Wertes von *Dseg* als typisierte Konstante im Hauptprogramm. Auf diese typisierte Konstante kann dann derAnwender des Interrupts zugreifen und sie zur Setzung seines DS Registers verwenden.

20.19 Interne Datenformate

In den folgenden Beschreibungen bedeutet das Symbol @ das Offset des ersten Byte, das von einer Variable gegebenen Typs innerhalb seines Segments besetzt ist. Die Segmentbasisadresse kann bestimmt werden, indem man die Standardfunktion *Seg* benutzt.

Globale Variablen, lokale Variablen und *typisierte Konstanten* besetzten verschiedene Segmente, wie folgt:

Globale Variablen liegen im Datensegment, das Offset bezieht sich auf das DS Register.

Lokale Variablen liegen im Stacksegment, das Offset bezieht sich auf das BP Register.

Typisierte Konstanten liegen im Codesegment, das Offset bezieht sich auf das CS Register.

Alle Variablen sind in ihrem Basissegment enthalten.

20.19.1 Grundtypen von Daten

Die Grundtypen von Daten können in Strukturen (Arrays, Records und Diskettendateien) gruppiert werden, wobei diese Strukturierung ihre internen Formate nicht beeinflußt.

20.19.1.1 Skalare

Die folgenden Skalare werden alle in einem einzigen Byte gespeichert: *Integer* Teilbereiche mit beiden Grenzen im Bereich 0..255, *boolean*, *char* und deklarierte Skalare mit weniger als 256 möglichen Werten. Dieses Byte enthält den ordinalen Wert dieser Variablen.

Die folgenden Skalare werden alle in zwei Byte gespeichert: *Integer*, *integer* Teilbereiche mit einer oder beiden Grenzen außerhalb des Bereichs 0..255 und deklarierte Skalare mit mehr als 256 möglichen Werten. In diesen Bytes ist das niederwertigste Byte zuerst gespeichert.

20.19.1.2 Reelle Zahlen

Reelle Zahlen (Datentyp: *real*) belegen 6 Byte, ein Gleitkommawert hat eine 40-Bit Mantisse und einen 8-Bit Exponenten. Der Exponent wird im ersten Byte gespeichert, die Mantisse in den nächsten fünf Byte, mit dem niederwertigsten Byte als erstem:

@	Exponent
@ +1	LSB der Mantisse
:	
@ +5	MSB der Mantisse

Der Exponent hat binäres Format mit einer Voreinstellung von $80. Daher bedeutet ein Exponent von $84, daß der Wert der Mantisse mit 2^($84-$80) = 2^4=16 potenziert werden muß. Wenn der Exponent null ist, wird der Gleitkommawert als null angesehen.

Den Wert der Mantisse erhält man, indem man die 40-Bit vorzeichenlose ganze Zahl durch 2^40 teilt. Die Mantisse ist immer normalisiert, d.h. das signifikanteste Bit (Bit 7 des fünften Byte) sollte als 1 interpretiert werden. Das Vorzeichen der Mantisse wird in diesem Bit gespeichert, eine 1 zeigt an, daß die Zahl negativ ist, eine 0, daß sie positiv ist.

20.19.1.3 Strings

Ein String besetzt soviele Bytes, wie seine maximale Länge plus eins ist. Das erste Byte enthält die gegenwärtige Länge des String. Die folgenden Bytes enthalten den String, wobei das erste Schriftzeichen an der niedrigsten Adresse gespeichert ist. In der Tabelle unten, bedeutet L die gegenwärtige Länge des String und *Max* bedeutet die maximale Länge:

@	Gegenwärtige Länge (L)
@ +1	erstes Schriftzeichen
@ +2	zweites Schriftzeichen
:	
:	
@ +*L*	letztes Schriftzeichen
@ *L+1*	unbenutzt
:	
@ +*Max*	unbenutzt

20.19.1.4 Mengen

Ein Element einer Menge (engl: set) belegt ein Bit. Die maximale Zahl von Elementen einer Menge ist 256. Eine Mengenvariable belegt nie mehr als 32 Byte (256/8).

Wenn eine Menge weniger als 256 Elemente enthält, sind einige Bits auf Null gesetzt und müssen deshalb nicht gespeichert werden. Im Interesse der Speichereffizienz wäre die beste Art, eine Mengenvariable des gegebenen Typs zu speichern, die nicht-signifikanten Bits wegzuschneiden und die verbleibenden Bits so zu rotieren, daß das erste Element der Menge das erste Bit des ersten Byte belegt. Solche Rotationsoperationen sind sehr langsam, weshalb TURBO einen Kompromiß anwendet: Nur Bytes, die statisch null sind (d h. Bytes, bei denen kein Bit gebraucht wird) werden nicht gespeichert. Diese Methode der Kompression ist sehr schnell und in den meisten Fällen so speichereffizient wie die Rotationsmethode.

Die Zahl der Bytes, die durch eine Mengenvariable belegt sind, wird mit (*Max* **div** 8) - (*Min* **div** 8) + 1 berechnet, wobei *Min* und *Max* untere und obere Grenzen des Grundtyps einer Menge sind. Die Speicheradresse eines spezifischen Elements *E* ist:

MemAddress = (*a* + (*E* **div**8) - (*Min* **div** 8)

und die Bitadresse innerhalb des Byte bei der MemAddress ist:

BitAddress = *E* **mod** 8

E entspricht dem ordinalen Wert des Elements.

20.19.1.5 Zeiger

Ein Zeiger besteht aus vier Bytes, die eine Segmentbasisadresse und ein Offset enthalten. Die beiden niederwertigsten Bytes enthalten das Offset und die beiden höchstwertigsten enthalten die Segmentbasisadresse. Beide sind im umgekehrten Byteformat abgespeichert, d.h. mit dem niederwertigsten Byte zuerst. Der Wert **nil** entspricht zwei Nullwörtern.

20.19.2 Datenstrukturen

Datenstrukturen werden aus den Grundtypen von Daten unter Verwendung verschiedener Strukturierungsmethoden gebildet. Es gibt drei verschiedene Strukturierungsmethoden: Arrays, Records und Diskettendateien. Die Strukturierung von Daten beeinträchtigt in keiner Weise das interne Format der Grundtypen von Daten.

20.19.2.1 Arrays

Die Komponenten mit dem niedrigsten Indexwert werden an der niedrigsten Speicheradresse gelagert. Ein mehrdimensionales Array wird der Wertigkeit nach abgespeichert, d.h. die rechteste Dimension nimmt als erstes zu. Z.B. erhalten Sie bei der Eingabe des Arrays:

Board: **array**[1..8,1..8] **of** Square

das folgende Speicher-Layout seiner Komponenten:

Board[1,1] | *niedrigste Adresse*
Board[1,2]
:
:
Board[1,8]
Board[2,1]
Board[2,2]
:
:
Board[8,8] | höchste Adresse

20.19.2.2 Records

Das erste Feld eines Records wird an der niedrigsten Speicheradresse gespeichert. Wenn der Record aus nicht veränderbaren Elementen zusammengesetzt ist, ist seine Länge durch die Summe der Längen der einzelnen Felder gegeben. Enthält der Record veränderbare Elemente, ergibt sich die Gesamtzahl der von ihm belegten Bytes aus der Summe der Länge der festen Elemente und der Länge des größten seiner veränderlichen Elemente. Jedes veränderliche Element beginnt bei derselben Speicheradresse.

20.19.2.3 Diskettendateien

Diskettendateien unterscheiden sich von anderen Datenstrukturen insofern, als Daten nicht im internen Speicher abgelagert werden, sondern in einer Datei eines externen Mediums. Eine Diskettendatei wird mit Hilfe des File Interface Block (FIB) bearbeitet.

20.19.2.3.1 File Interface Blocks

Folgende Tabelle zeigt das Format eines FIB:

@ +0	Datei-Kennzeichnung (LSB=niedrigstes Byte).
@ +1	Datei-Kennzeichnung (MSB=höchstes Byte).
@ +2	Recordlänge (LSB) oder Flags-Byte.
@ +3	Recordlänge (MSB) oder Zeichenpuffer.
@ +4	Puffer-Offset (LSB).
@ +5	Puffer-Offset (MSB).
@ +6	Puffergröße (LSB).
@ +7	Puffergröße (MSB).
@ +8	Pufferzeiger (LSB).
@ +9	Pufferzeiger (MSB).
@ +10	Pufferende (LSB).
@ +11	Pufferende (MSB).
@ +12	Erstes Byte des Dateipfad.
.	
.	
@ +75	Letztes Byte des Dateipfad.

Wenn die Datei geöffnet ist, enthält das von MS-DOS ausgegebene Wort bei **@ +0 und** @ +1 die 16-Bit Datei-kennzeichnung (bzw. es wird OFFFFH ausgegeben, wenn die Datei geschlossen ist). Für typisierte und nicht-typisierte Dateien enthält das Wort bei @ +2 und @ +3 die Recordlänge in Bytes (null, falls die Datei geschlossen ist); die Bytes von @ +4 bis @ +11 sind unbenutzt.

Bei Textdateien ist das Format der Flags-Byte bei @ +2 wie folgt:

Bit 0..3	Dateityp.
Bit 5	Pre-read Zeichen-Flag.
Bit 6	Output Flag.
Bit 7	Input Flag.

Dateityp 0 kennzeichnet eine Diskettendatei, durch 1 bis 5 werden die logischen I/O Geräte von TURBO Pascal benannt (CON:, KBD:, LST:, AUX:, und USR:). Bit 5 wird gesetzt, wenn der Zeichenpuffer ein Pre-read Zeichen enthält, Bit 6, wenn Output erlaubt ist, und Bit 7, wenn Input erlaubt ist.

Die vier Wörter von @+4 bis @+11 speichern die Offsetadresse des Puffers, seine Größe, das Offset des nächsten zu lesendenden oder zu schreibenden Zeichens und das Offset des ersten Byte nach dem Puffer. Der Puffer liegt immer in demselben Segment wie der FIB und beginnt gewöhnlich bei @+76. Wenn eine Textdatei einem logischen Gerät zugewiesen wird, werden lediglich die Flags-Byte und der Zeichenpuffer benutzt.

Der Dateipfad ist ein ASCII String (ein String, der durch ein Null-Byte beendet wird), von bis zu 63 Zeichen.

20.19.2.3.2 Dateien mit direktem Zugriff

Eine Datei mit direktem Zugriff (engl: random access) besteht aus einer Folge von Records, die alle die gleiche Länge und das gleiche Format besitzen. Um die Dateispeicherkapazität zu optimieren, sind die Records einer Datei direkt aneinandergrenzend.

TURBO behält keine Information über die Länge der Records. Der Programmierer muß deshalb dafür sorgen, daß auf eine Datei mit direktem Zugriff, mit der korrekten Recordlänge zugegriffen wird.

Die Größe, die die Standardfunktion *FileSize* ausgibt, stammt aus der DOS-Directory.

20.19.2.3.3 Textdateien

Die Grundelemente einer Textdatei sind Zeichen, aber weiterhin ist eine Textdatei in *Zeilen* unterteilt. Jede Zeile besteht aus einer beliebigen Zahl von Zeichen, die von einer CR/LF Sequenz (ASCII $0D/$0A) beendet wird. Die Datei wird mit Ctrl-Z (ASCII $1B) abgeschlossen.

20.19.4 Parameter

Parameter werden an Prozeduren und Funktionen mittels des Stack übertragen, der durch SS:SP adressiert ist.

Zu Beginn eines **external** Unterprogramms enthält die Spitze des Stack immer die Rückkehradresse innerhalb des Codesegments (ein Wort). Die Parameter, falls vorhanden, liegen unterhalb der Rückkehradresse, d.h. an höheren Adressen im Stack.

Wenn eine externe Funktion den folgenden Unterprogrammkopf hat:

```
function Magic(var R: Real; S: string5): Integer;
```

dann würde der Stack am Eingang zu *Magic* den folgenden Inhalt haben:

```
( Function result)
( Segment base addrss of R)
( Offset address of R)
( Mantissa of R next 5 bytes )
:
( First character of S)
:
( Last character of S)
( Length of S )
( Return address ) SP
```

Eine externe Unterroutine sollte das Basisseitenregister (Base Page, bzw. BP) sichern und dann den Stackzeiger (Stack Pointer, bzw. SP) in das Basisseitenregister kopieren, um auf Parameter verweisen zu können. Weiterhin sollte die Unterroutine auf dem Stack Platz für den lokalen Arbeitsbereich reservieren. Dies kann kann durch die folgenden Instruktionen erreicht werden:

```
PUSH    BP
MOV     BP,SP
SUB     SP,WORKAREA
```

Die letzten Instruktionen fügen Folgendes zum Stack hinzu:

```
( Return address ) BP
( The saved BP register )
( First Byte of local work area )
:
( Last Byte of local work area ) SP
```

Parameter werden über des BP Registers erreicht.

Die folgende Instruktion lädt die Länge des Strings in das AL Register:

```
MOV   AL,[BP-1]
```

Bevor man eine RET Instruktion ausführt, muß der Stackzeiger und das Basisseitenregister auf die ursprünglichen Werte zurückgesetzt werden.
Wenn der RET ausgeführt wird, können die Parameter entfernt werden, indem man RET einen Parameter gibt, der spezifiziert, wieviele Bytes entfernt werden sollen. Die folgenden Instruktionen sollten deshalb beim Austritt aus einem Unterprogramm benutzt werden:

```
MOV   SP,BP
POP   BP
RET   NoOfBytesToRemove
```

20.19.4.1 Variablenparameter

Ein Variablenparameter (**var**) überträgt zwei Worte auf den Stack, in denen die Basisadresse und das Offset des ersten durch den aktuellen Parameter belegten Bytes angegeben wird.

20.19.4.2 Wertparameter

Bei Wertparametern hängen die auf den Stack übertragenen Daten vom Typ des Parameters ab, wie in den nächsten Abschnitten beschrieben wird.

20.19.4.2.1 Skalare

Integer, *Boolean*, *Char* und deklarierte Skalare (d.h. alle Skalare außer *Real*) werden als ein Wort auf den Stack übertragen. Wenn die Variable bei der Speicherung nur ein Byte belegt, ist das höchste Byte des Parameters null.

20.19.4.2.2 Reelle Zahlen

Eine reelle Zahl wird auf den Stack unter Verwendung von sechs Bytes übertragen.

20.19.4.2.3 Strings

Wenn ein String an der Spitze des Stack ist, enthält das oberste Byte die Länge des String, es folgen dann die Zeichen des String.

20.19.4.2.4 Mengen

Eine Menge (engl: set) belegt immer 32 Bytes im Stack (die Kompression von Mengen wird nur beim Laden und Speichern von Mengen angewendet).

20.19.4.2.5 Zeiger

Ein Zeigerwert wird auf den Stack als zwei Worte übertragen, die die Basisadresse und das Offset einer dynamischen Variablen enthalten. Der Wert **NIL** entspricht zwei Nullwörtern.

20.19.4.2.6 Arrays und Records

Auch wenn sie als Wertparameter verwendet werden, werden *Array*- und *Record*-Parameter tatsächlich nicht auf den Stack übertragen. Stattdessen werden zwei Worte übertragen, die die Basisadresse und das Offset des ersten Byte des Parameters enthalten. Es liegt dann in der Verantwortung der Unterroutine, von dieser Information Gebrauch zu machen und eine lokale Kopie dieser Variablen zu erstellen.

20.19.5 Funktionsergebnisse

Vom Benutzer geschriebene, **externe** Funktionen müssen vor ihrem Rücksprung erst alle Parameter und das Funktionsergebnis vom Stack entfernen.

Vom Benutzer geschriebene, **externe** Funktionen müssen ihre Ergebnisse exakt wie im folgenden angegeben, weitergeben:

Die Werte von skalaren Typen, außer *Real*, müssen in das AX Register ausgegeben werden. Wenn das Ergebnis nur ein Byte ist, dann sollte AH auf null gesetzt werden. *Boolean* Funktionen müssen den Funktionswert ausgeben, indem sie die Z Flag (Z= Falsch, NZ= Wahr) setzen.

Real Zahlen müssen auf den Stack mit dem Exponenten an der niedrigsten Adresse ausgegeben werden. Dies geschieht, wenn die Funktionsergebnisvariable bei der Ausgabe nicht entfernt wird.

Mengen müssen an die Spitze des Stack in Übereinstimmung mit dem auf Seite 223 beschriebenen Format ausgegeben werden. Am Ausgang muß SP auf das Byte zeigen, das die Stringlänge enthält.

Zeigerwerte müssen in DX:AX ausgegeben werden.

20.19.6 Der Heap und die Stacks

Während der Ausführung eines TURBO Pascal Programms sind dem Programm folgende Segmente zugewiesen:

ein Codesegment
ein Datensegment und
ein Stacksegment

Zwei stapelartige Strukturen werden während der Programmausführung verwaltet: der *Heap* und der *Stack*.

Der Heap wird benutzt, um dynamische Variablen zu speichern und wird mit den Standardprozeduren *New,Mark* und *Release* kontrolliert. Am Beginn eines Programms wird der Heapzeiger *HeapPtr* auf eine niedrige Stelle des Speichers im Stacksegment gesetzt. Der Heap wächst dann aufwärts gegen den Stack. Die vordefinierte Variable *HeapPtr* enthält den Wert des Heapzeigers und erlaubt es dem Programmierer, die Position des Heap zu kontrollieren.

Der Stack wird benutzt, lokale Variablen und Zwischenergebnisse bei der Berechnung von Ausdrücken zu speichern und Parameter an Prozeduren und Funktionen zu übertragen. Am Beginn eines Programms ist der Stackzeiger auf die Adresse an der Spitze des Stacksegments gesetzt.

Bei jedem Aufruf der Prozedur *New* und bei der Eingabe einer Prozedur oder Funktion prüft das System ob sich ein Zusammenstoß zwischen Heap und Recursion-Stack ereignet hat. Wenn dies der Fall war, entsteht ein Ausführungsfehler, wenn nicht der Compilerbefehl (|$K-|) passiv gesetzt ist .

20.20 Speicherverwaltung

Bei der Ausführung eines TURBO Programms, werden dem Programm drei Segmente zugewiesen: Ein Codesegment, ein Datensegment und ein Stacksegment.

Codesegment (CS ist das Codesegmentregister):

CS:0000	- CS:00FF	MS-DOS Basisseitenregister.
CS:0100	- CS:EOFR	Laufzeit-Librarycode.
CS:EOFR	- CS:EOFP	Programm Code.
CS:EOFP	- CS:EOFC	Unbenutzt.

Datensegment (DS ist das Datensegmentregister):

DS:0000	- DS:EOFW	Laufzeit-Library-Arbeitsspeicher.
DS:EOFW	- DS:EOFM	Hauptprogramm-Blockvariablen.
DS:EOFM	- DS:EOFD	Unbenutzt.

Die unbenutzten Bereiche zwischen (CS:EOFP-CS:EOFC und DS:EOFM-DS:EOFD) werden nur angewiesen, wenn bei der Compilierung zumindet eine größere COdesegmentgröße, als die erforderliche Größe, spezifiziert wird. Die Größe des Code- und Datensegments kann jeweils 64 K Bytes nicht überschreiten.

Das Stäcksegment ist ein wenig komplizierter, da es größer als 64 K Bytes sein kann. Zu Beginn des Programms wird das Stacksegmentregister (SS) und der Stackzeiger (SP) geladen, so daß SS:SP auf das allerletzte verfügbare Byte des gesamten Segments zeigt. Während der Programmausführung wird SS nie geändert, SP dagegen kann sich abwärts bewegen, bis es den untersten Teil des Segments, oder 0 erreicht hat (entsprechend den 64K Bytes des Stack), wenn das Stacksegment größer als 64 K Bytes ist.

Der Heap wächst von dem unteren Speicher in dem Stacksegment zu dem aktuellen Stack, der in dem oberen Speicher liegt. Immer wenn dem Heap eine Variable zugewiesen wird, wird der Heapzeiger (der eine Doppelwort-Variable ist, die vom TURBO Laufzeitsystem angelegt und verwaltet wird) nach oben bewegt und dann normalisiert, so daß die Offsetadresse immer zwischen $0000 und $000F liegt. Deshalb ist die maximale Größe einer einzelnen Variablen, die dem Heap zugewiesen werden kann, 65521 Bytes (gemäß $1000 weniger $000F). Die Gesamtgröße aller Variablen, die dem Heap zugewiesen werden kann, ist jedoch nur durch den zur Verfügung stehenden Speicherplatz begrenzt. Der Heapzeiger steht dem Programmierer durch den *HeapPtr* Standardbezeichner zur Verfügung. *HeapPtr* ist ein typloser Zeiger, der zu allen Zeigertypen kompatibel ist. Zuweisungen zu *HeapPtr* sollten nur mit äußerster Vorsicht durchgeführt werden.

21. CP/M-86

Dieses Kapitel beschreibt die speziellen Eigenschaften der CP/M-86 Implementation von TURBO Pascal. Es beinhaltet zwei Arten von Informationen:

1) Dinge, die Sie wissen sollten, um effizienten Gebrauch von TURBO-Pascal zu machen. Seite 227 bis 240.

2) Der Rest des Kapitels beschreibt Dinge, die nur für erfahrene Programmierer von Interesse sein dürften, wie z.B. Assemblerroutinen, technische Aspekte des Compilers, etc.

21.1 Compiler-Optionen

Das **O** Kommando wählt das folgende Menü an, in dem Sie einige voreingestellte Werte des Compilers sehen und verändern können. Es ist auch beim Finden von Laufzeit-Fehlern hilfreich.

```
compile -)        Memory
                  Cmd-file
                  cHn-file

commandline Parameter:

Find run-time error     Quit
```

Abbildung 21-1: Optionen Menü

21.1.1 Memory / Cmd-file / cHn-file

Die drei Befehle **M**, **C** und **H** steuern die Verarbeitungsart der Source und die Ablage des erzeugten Object-Codes durch den Compiler. **M**emory ist der voreingestellte Modus. Der Code wird im Arbeitsspeicher erzeugt und behalten. Das Programm kann dann direkt vom Speicher aus durch den **R**un-Befehl ausgeführt werden.

Cmd-file wird durch Eingabe von **C** gewählt und durch den Pfeil angezeigt. Der Compiler schreibt den Code in eine Datei, mit demselben Namen wie die Arbeitsdatei (oder Hauptdatei, falls angegeben), als .CMD-Datei. Diese Datei enthält den Object-Code sowie die Pascal 'runtime library', und kann durch Eingabe des Dateinamens aktiviert werden.

c**H**ain-file wird durch Eingabe von **H** gewählt und durch den Pfeil angezeigt. Der Code wird vom Compiler auf eine Datei mit demselben Namen, wie die Arbeitsdatei (oder Hauptdatei, falls angegeben) als .CHN-Datei geschrieben. Diese Datei enthält den Programmcode, aber nicht die Pascal 'runtime library' und muß von einem anderen TURBO Pascal Programm aus, mit der Prozedur *Chain* aktiviert werden (siehe Seite 231).

Wenn der **C**md oder c**H**n Modus gewählt wird, erscheinen vier zusätzliche Zeilen auf dem Bildschirm:

```
minimum c0de segment size:      XXXX paragraphs (max. YYYY)
minimum Data segment size:      XXXX paragraphs (max. YYYY)
mInimum free dynamic memory:    XXXX paragraphs
mAximum free dynamic memory:    XXXX paragraphs
```

Abbildung 21-2: Speicherverwendungs-Menü

Der Gebrauch dieser Befehle wird in den folgenden Abschnitten beschrieben.

21.1.2 Minimale Codesegmentgröße

Der **O**-Befehl wird verwendet, um die minimale Größe des Codesegments für eine .CMD Datei die *Chain* oder *Execute* benutzt, festzusetzen. Wie auf Seite 231 diskutiert, ändern *Chain* und *Execute* die Basisadresse der Code-, Daten- und Stacksegmente nicht. Die minimal festzulegende Größe richtet sich also nach dem längsten Segment in einer der über *Chain* oder *Execute* aufgerufenen Programme.

Folglich müssen Sie für die Compilierung eines Programms, den Wert der minimalen Codesegmentgröße mindestens auf den gleichen Wert setzen, den das größte Codesegment der zu verkettenden/auszuführenden Programme besitzt. Die erforderlichen Werte erhält man durch die Statusanzeige, die jede Compilierung abschließt. Die Werte werden hexadezimal angegeben und spezifizieren die Zahl der Paragraphen; ein Paragraph entspricht 16 Bytes.

21.1.3 Minimale Größe des Datensegments

Der **D** Befehl wird verwendet, um die minimale Größe eines Datensegments für eine .CMD Datei bei Benutzung von *Chain* oder *Execute* festzulegen. Die Regeln für die Festsetzung der Größe wurden oben diskutiert.

21.1.4 Minimaler freier dynamischer Speicher

Dieser Wert gibt die minimale Speichergröße an, die für Stack und Heap benötigt wird. Der Wert wird hexadezimal angegeben und spezifiziert die Zahl der Paragraphen; ein Paragraph entspricht 16 Bytes.

21.1.5 Maximaler freier dynamische Speicher

Dieser Wert spezifiziert die maximale Speichergröße, die dem Stack und Heap zugewiesen ist. Er muß in Programmen, die in einer Mehrplatzumgebung wie Concurrent CP/M-86 laufen, verwendet werden, um sicherzustellen, daß das Programm nicht den ganzen freien Speicher belegt. Der Wert ist hexadezimal und gibt die Zahl der Paragraphen an. Ein Paragraph entspricht 16 Bytes.

21.1.6 Kommandozeilen-Parameter

Das **P** Kommando erlaubt Ihnen, einen oder mehrere Parameter einzugeben, die an Ihr Programm übergeben werden, wenn Sie es im **M**emory-Modus laufen lassen. Dies geschieht genauso, als ob Sie diese in der DOS Kommandozeile eingegeben hätten. Die Parameter können mit den Funktionen *ParamCount* und *ParamStr* angesprochen werden.

21.1.7 Finden von Laufzeit-Fehlern

Wenn Sie ein Programm laufen lassen, das im Speicher compiliert wurde und es tritt ein Fehler auf, wird der Editor aufgerufen und der Fehler wird automatisch angezeigt. Das ist natürlich nicht möglich, wenn das Programm eine .CMD Datei oder eine .CHN Datei ist. Fehlermeldungen geben den Fehlercode und den Wert des Programmzählers bei Auftreten des Fehlers an, wie z.B.:

```
Run-time error 01, PC=1B56
Program aborted
```

Abbildung 21-3: Fehlermeldung

Um im Source die Stelle zu finden, an der der Fehler aufgetreten ist, ist der Befehl **F** einzugeben. Wenn dann nach der Adresse gefragt wird, gibt man die in der Fehlermeldung genannte Adresse ein:

```
Enter PC: 1B56
```

Abbildung 21-4: Orten des Fehlers

Die Stelle ist nun im Source geortet und so angezeigt, wie der Fehler bei dem Programmablauf im Speicher aufgetreten war.

Beachten Sie, daß in Programmen mit Overlays die Lokalisierung von Fehlern ein wenig komplizierter sein kann, wie auf Seite 156 beschrieben.

21.2 Standardbezeichner

Die folgenden Standardbezeichner sind bei der 16-Bit Version einheitlich:

Bdos	Intr	Ofs	Seg
CSeg	MemW	PortW	SSeg
DSeg			

21.3 Chain und Execute

TURBO Pascal enthält die zwei Prozeduren *Chain* und *Execute*, die es Ihnen erlauben, von einem TURBO Programm aus andere TURBO Programme zu aktivieren. Die Syntax der Prozeduraufrufe ist:

```
Chain(FilVar)
Execute(FilVar)
```

Dabei ist *FilVar* eine Dateivariable beliebigen Typs, die zuvor mit der Standardprozedur *Assign* einer Diskettendatei zugewiesen wurde. Wenn die Datei existiert, wird sie in den Speicher geladen und ausgeführt.

Die *Chain* Prozedur wird nur verwendet, um spezielle TURBO Pascal .CHN Dateien zu aktivieren, d.h. Dateien, die mit der Option cHn-file des Optionenmenü compiliert wurden (siehe Seite 190). Eine solche Datei enthält nur Programmcode, keine Pascal Library; sie benutzt die schon im Speicher befindliche Pascal Library.

Die *Execute* Prozedur wird benutzt, um eine beliebige TURBO Pascal .CMD Datei zu aktivieren.

Wenn die Diskettendatei nicht existiert, tritt ein I/O Fehler auf. Dieser Fehler wird, wie auf Seite 116 beschrieben, behandelt. Wenn der Compilerbefehl passiv ist (|$I-|), geht die Programmausführung mit der Anweisung weiter, die der fehlgeschlagenen *Chain* oder *Execute* Anweisung folgt. Die *I/O result*-Funktion muß vor weiteren I/O Operationen aufgerufen werden.

Daten können vom laufenden Programm aus, auf Chain-Programme entweder durch gemeinsame, globale Variablen (*shared global variables*) oder durch absolute Variablen (*absolute address variables*) übertragen werden.

Um Überlappungen zu vermeiden, müssen gemeinsame, globale Variablen als die ersten Variablen in beiden Programmen deklariert werden. Sie müssen in beiden Deklarationen in der gleichen Reihenfolge aufgelistet sein. Weiterhin müssen beide Programme auf die gleiche Größe von Code und Datensegmenten compiliert werden (siehe Seite 228 und 229). Wenn diese Bedingungen erfüllt sind, werden die Variablen von beiden Programmen an dieselbe Adresse im Speicher plaziert. Da TURBO Pascal seine Variablen nicht automatisch initialisiert, können sie gemeinsam genutzt werden.

Beispiel:
Programm MAIN.CMD:

```
program Main;
var
  Txt:      string[80];
  CntPrg:   file;

begin
  Write('Enter any text: '); Readln(Txt);
  Assign(CntPrg, 'ChrCount.chn');
  Chain(CntPrg);
end.
```

Programm *CHRCOUNT.CHN:*

```
program ChrCount;
var
  Txt: string[80];
  NoOfChar,
  NoOfUpc,
  I:    Integer;

begin
  NoOfUpc := 0;
  NoOfChar := Length(Txt);
  for I := 1 to length(Txt) do
    if Txt[I] in ['A'..'Z'] then NoOfUpc := Succ(NoOfUpc);
  Write('No of characters in entry: ', NoOfChar);
  Writeln('. No of upper case characters: ', NoOfUpc,'.');
end.
```

Wenn Sie bestimmen wollen, ob ein TURBO Programm entweder durch *execute* oder direkt von der Befehlszeile aufgerufen wurde, sollten Sie eine **absolute** Variable mit der Basisadresse *Cseg:*$80 verwenden. Diese Adresse enthält das Längenbyte der Befehlszeile, und wenn ein Programm von DOS aufgerufen wird, enthält es einen Wert zwischen 0 und 127. Bei der Ausführung (eXecute) eines Programms sollte deshalb das aufrufende Programm diese Variable auf einen höheren Wert als 127 setzen. Wenn Sie dann die Variable in dem aufgerufenen Programm prüfen, bedeutet ein Wert zwischen 0 und 127, daß das Programm von CP/M aufgerufen wurde; ein höherer Wert besagt, daß es von einem anderen TURBO Programm aufgerufen wurde.

Die Anwendung von *Chain* und *Execute* bei TURBO Programmen verändert das Speicher-Layout nicht. Die Basisadresse und die Größe des Codes, der Daten und des Stacksegments werden nicht verändert; *Chain* und *Execute* ersetzen lediglich den Programm-Code in dem Codesegment. 'Fremde' Programme können deshalb von einem TURBO Programm nicht aufgenommen werden.

Es ist unbedingt erforderlich, daß das erste Programm, das eine *Chain* Aweisung ausführt, genügend Speicher für den Code, die Daten und die Stacksegmente zuweist, um das größte .CHN Programm unterzubringen. Dies geschieht, indem man im Optionenmenü den minimalen Platz für Code, Daten und freien Speicher angibt (siehe Seite 190).

Beachten Sie, daß weder *Chain* noch *Execute* in direktem Modus verwendet werden können, d.h. beispielsweise von einem Programm, das mit auf Memory geschalteter Compileroption läuft (siehe auch Seite 190).

21.4 Overlays

Während der Ausführung erwartet das System normalerweise seine Overlaydateien im angemeldeten Laufwerk zu finden. Die Prozedur *OvrDrive* kann verwendet werden, um diesen voreingestellten Wert zu ändern. .

21.4 Overlays

Während der Ausführung erwartet das System normalerweise seine Overlaydateien im angemeldeten Laufwerk zu finden. Die Prozedur *OvrDrive* kann verwendet werden, um diesen voreingestellten Wert zu ändern.

21.4.1 OvrDrive-Prozedur

Syntax: OvrDrive(*Drive*);

Drive stellt einen *integer* Ausdruck dar, der ein Laufwerk (0=angemeldetes Laufwerk, 1=A:, 2=B:, etc.) spezifiziert. Bei einem Aufruf von Overlaydateien, werden die Dateien in der spezifizierten Directory erwartet. Wenn einmal eine Overlaydatei in einer Directory eröffnet wurde, wird in Zukunft dieselbe Datei in der gleichen Directory gesucht.

Beispiel:

```
program OvrTest;

overlay procedure ProcA;
begin
  WriteLn('Overlay A');
end;
```

```
overlay procedure ProcB;
begin
  WriteLn('Overlay B');
end;

procedure Dummy;
begin
   Dummy Prozedur zur Trennung der Overlays in zwei Gruppen
end;

overlay procedure ProcC;
begin
  WriteLn('Overlay C');
end;

begin
  OvrDrive(2);
  ProcA;
  OvrDrive(0);
  ProcC;
  OvrDrive(2);
  ProcB;
end.
```

Der erste Aufruf von *OvrDrive* spezifiziert Overlays, die auf dem Laufwerk *B:* gesucht werden. Der Aufruf von *ProcA* bewirkt daher, daß die erste Overlaydatei (die die beiden Overlayprozeduren *ProcA* und *ProcB* enthält) hier eröffnet wird.

Als nächstes bestimmt die *OvrDrive(0) Anweisung, daß die folgenden Overlays auf dem angemeldeten Laufwerk zu finden sind. Der Aufruf von ProcC* eröffnet hier die zweite Overlaydatei.

Die folgende *ProcB* Anweisung ruft eine Overlayprozedur in der ersten Overlaydatei auf; um sicherzustellen, daß diese auf dem Laufwerk *B:* gesucht wird, muß vor dem Aufruf die *OvrDrive(2)* Anweisung ausgeführt werden.

21.5 Dateien

21.5.1 Dateinamen

Ein Dateiname in CP/M-86 besteht aus bis zu acht Buchstaben oder Zahlen, wahlweise mit nachstehendem Punkt und einem Dateityp aus bis zu drei Zahlen oder Buchstaben:

Laufwerk:Name.Typ

21.5.1.1 Nichttypisierte Dateien

Es kann wahlweise ein zweiter Parameter bei *Reset* und *ReWrite* angegeben werden, um die von *BlockRead* und *BlockWrite* zu verwendende Blockgröße zu bestimmen.

Beispiel:
```
Assign(InFile,'INDATA');
Reset(InFile,BlockSize);
```

wobei *BlockSize* ein *integer* Ausdruck ist.

21.5.2 Textdateien

Die Prozeduren *Seek* und *Flush*, sowie die Funktionen *FilePos* und *FileSize* sind nicht auf CP/M Textdateien anwendbar.

21.5.3 Puffergröße

Die voreingestellte Puffergröße ist bei Textdateien 128 Bytes. Das ist für die meisten Anwendungen angemessen; Programme, die viele Ein/Ausgaben vornehmen (z.B. Kopierprogramme), profitieren jedoch von einem größeren Puffer, da dadurch die Kopfbewegungen im Laufwerk reduziert werden.

Deshalb ermöglicht Ihnen eine Option die Angabe der Puffergröße, wenn Sie eine Textdatei deklarieren:

```
VAR
   TextFile:    Text|$1000|;
```

deklariert eine Textdateivariable mit einer Puffergröße von 4K Bytes.

21.6 Absolute Variablen

Variablen können so deklariert werden, daß sie bei einer bestimmten Speicheradresse verbleiben, sie werden dann **absolute** Variablen genannt. Dies wird erreicht, indem man an die Variablendeklaration das reservierte Wort **absolute** anhängt. Darauf müssen zwei *integer* Konstanten folgen, die ein Segment und den Offset nennen, wo die Variable lokalisiert werden kann:

```
var
Abc: Integer absolute $0000:$00EE;
Def: Integer absolute $0000:$00F0;
```

Die erste Konstante spezifiziert eine Segmentbasisadresse, die zweite Konstante das Offset innerhalb dieses Segments. Die Standardbezeichner *CSeg* und *DSeg* können benutzt werden, um Variablen an absoluten Adressen innerhalb eines Codesegments (**CSeg**) oder eines Datensegments (**DSeg**) zu plazieren:

```
Patch: array[1..PatchSize]of byte absolute CSeg:$05F3;
```

Absolute kann auch zur Überlagerung von Variablen benutzt werden, d.h., daß eine Variable bei derselben Adresse beginnen soll wie eine andere Variable. Wenn dem Bezeichner einer Variablen oder eines Parameters **absolute** folgt, beginnt die neue Variable an der Adresse dieses Variablen-Parameters.

Beispiel:

```
var
  Str: string[32];
  StrLen: Byte absolute Str;
```

Diese Deklaration bedeutet, daß die Variable *StrLen* an derselben Adresse wie die Variable *Str* beginnen soll, und da das erste Byte einer Stringvariablen die Länge des Strings enthält, wird *StrLen* die Länge von *Str* enthalten. Beachten Sie, daß die Deklaration einer absoluten Variablen nur einen Bezeichner benennen darf.

Weitere Erläuterungen zur Platzzuweisung für Variablen finden Sie auf Seite 246

21.7 Absolute Adressfunktionen

Mit Hilfe der folgenden Funktionen können Informationen über die Adressen von Programmvariablen und Systemzeigern abgerufen werden.

21.7.1 Addr

Syntax: Addr(*Name*)

Zeigt die Speicheradresse des ersten Bytes einer Variablen mit dem Bezeichner *Name*. Wenn *Name* ein Array ist, kann es indiziert werden, wenn *Name* ein Record ist, können einzelne Felder ausgewählt werden. Der einzugebende Wert ist ein 32 bit Zeiger, bestehend aus einer Segmentadresse und einem Offset.

21.7.2 Ofs

Syntax: Ofs(*Name*)

Nennt das Offset im Speichersegment, das vom ersten Byte der Variablen, Prozedur oder Funktion mit dem Bezeichner *Name* belegt wird. Wenn *Name* ein Array ist, kann es indiziert werden, wenn Name ein Record ist, können einzelne Felder gewählt werden. Es wird ein *integer* Wert ausgegeben.

21.7.3 Seg

Syntax: Seg(*Name*)

Nennt die Adresse des Segments, das das erste Byte einer Variablen, Prozedur oder Funktion mit dem Bezeichner *Name* enthält. Wenn *Name* ein Array ist, kann es indiziert werden, ist *Name* ein Record, können einzelne Felder gewählt werden. Es wird ein *integer* Wert ausgegeben.

21.7.4 Cseg

Syntax: Cseg

Nennt die Basisadresse des Codesegments. Es wird ein *integer* Wert ausgegeben.

21.7.5 Dseg

Syntax: Dseg

Nennt die Basisadresse des Datensegments. Es wird ein *integer* Wert ausgegeben.

21.7.6 Sseg

Syntax: Sseg

Nennt die Adresse des Stacksegments. Es wird ein *integer* Wert ausgegeben.

21.8 Vordefinierte Arrays

TURBO bietet vier vordefinierte Arrays vom Typ *Byte* an, mit denen der CPU-Speicher und die Datenports angesprochen werden können. Sie heißen *Mem*, *MemW*, *Port* und *PortW*.

21.8.1 Mem Array

Mit Hilfe der vordefinierten Arrays *Mem* und *MemW* kann der Speicher angesprochen werden. Jede Komponente des *Mem*-Arrays belegt ein Byte, die Komponenten des Arrays *MemW* ein Wort (zwei Bytes, zuerst LSB). Der Index muß eine Adresse sein, die die Segmentbasisadresse und den Offset nennt, die beide durch einen Doppelpunkt getrennt und *integer* sein müssen.

Die folgende Anweisung ordnet den Wert des Bytes in Segment 0000 mit dem Offset $0081 der Variablen *Value* zu:

```
Value:=Mem[0000:$0081];
```

während die folgende Anweisung:

```
MemW[Seg(Var):Ofs(Var)]:=Value;
```

den Wert der *integer* Variablen *Value* dort im Speicher plaziert, wo die ersten zwei Bytes der Variablen *Var* angesiedelt sind.

21.8.2 Port-Array

Port und *PortW* ermöglichen das Ansprechen der Datenports der 8086/88 CPU. Jedes Element des Arrays repräsentiert einen Datenport, wobei der Index den Datenportnummern entspricht. Da Datenports von 16-bit Adressen angewählt werden, ist der Indextyp *integer*. Wenn einer Komponente von *Port* oder *PortW* ein Wert zugeordnet wird, wird dieser an das genannten Port ausgegeben. Bezieht sich ein Ausdruck auf eine Portkomponente, wird deren Wert von dem genannten Port eingelesen. Die Komponenten des Arrays *Port* sind vom Typ *Byte*, die Komponenten von *PortW integer*.

Beispiel:
```
Port[56]:=10;
```

Der Gebrauch des Portarrays beschränkt sich auf Zuordnung und Referenz in Ausdrücken, das heißt, daß Komponenten von *Port* und *PortW* nicht als Variablenparameter von Prozeduren und Funktionen benutzt werden können. Darüberhinaus sind keine Operationen zugelassen, die sich auf das gesamte Port-array beziehen (Referenz ohne Index).

21.9 With-Anweisung

With-Anweisungen können in bis zu maximal 9 Stufen geschachtelt, angegeben werden.

21.10 Hinweise zu Zeigern

21.10.1 MemAvail

Mit der Standardfunktion *MemAvail* ist es jederzeit möglich, zu bestimmen, wieviel Raum im Heap verfügbar ist. Das Ergebnis ist *integer*, es nennt die Zahl der im Heap zur Verfügung stehenden Paragraphen (ein *Paragraph* umfaßt 16 Bytes).

21.10.2 Zeigerwerte

In bestimmten Fällen kann es von Interesse sein, einer Zeigervariablen einen bestimmten Wert zuzuordnen, *ohne eine andere Zeigervariable zu benutzen*, oder den wirklichen Wert einer Zeigervariablen zu ermitteln.

21.10.2.1 Zuweisung eines Werts zu einem Zeiger

Die Standardfunktion *Ptr* kann dazu benutzt werden, einem Zeiger bestimmte Werte zuzuordnen. Die Funktion gibt einen 32-bit Zeiger aus, der aus einer Segmentadresse und einem Offset besteht.

Beispiel:

```
Pointer:=Ptr(Cseg,$80);
```

21.10.2.2 Ermittlung des Zeigerwerts

Ein Zeigerwert wird als eine 32-bit Einheit dargestellt und die Standardfunktion *Ord* kann deshalb **nicht** zur Wertermittlung eingesetzt werden. Stattdessen müssen die Funktionen *Ofs* und *Seg* benutzt werden.

Mit der folgenden Anweisung erhält man den Zeigerwert *P* (der aus einer Segmentadresse und einem Offset besteht):

```
SegmentPart:=Seg(P^);
OffsetPart:=Ofs(P^);
```

21.11 CP/M-86 Funktionsaufrufe

Um CP/M-86 BDOS Systemaufrufe machen zu können, führt TURBO Pascal eine Prozedur *Bdos* ein, die einen Record als Parameter hat.

Details über BDOS Systemaufrufe und BIOS Routinen finden sie im *CP/M-86 Handbuch von Digital Research*.

Der Parameter zu *Bdos* muß vom Typ:

```
record
  AX,BX,CX,DX,BP,SI,DI,DS,ES,Flags: Integer;
end;
```

sein.

Bevor TURBO einen BDOS Systemaufruf macht, lädt es die Register AX, BX, CX, DX, BP, SI, DI, DS, und ES mit den in den Recordparametern spezifizierten Werten. Wenn BDOS die Operation beendet hat, speichert die *Bdos* Prozedur die Register wieder in dem Record und macht so die Ergebnisse von BDOS verfügbar.

21.12 Benutzergeschriebene I/O Treiber

Für manche Anwendungen ist es für den Programmierer von Nutzen seine eigenen Eingabe- und Ausgabetreiber zu definieren, d.h. Routinen, die Eingabe und Ausgabe von Zeichen externer Geräte ausführen. Die folgenden Treiber sind Teil von TURBO und werden von den Standard I/O Treibern benutzt (obwohl sie nicht als Standardprozeduren oder -funktionen verfügbar sind):

```
function  ConSt:boolean;|6|
function  ConIn:Char;|6|
procedure ConOut(Ch:Char);|6|
procedure LstOut(Ch:Char);|5|
procedure AuxOut(Ch:Char);|4|
function  AuxIn:Char;|3|
procedure UsrOut(Ch:Char);|6|
function  UsrIn:Char;|6|
```

Die *ConSt* Routine wird von der Funktion *KeyPressed* aufgerufen, die *ConIn* und *ConOut* Routinen werden von CON:, TRM: und KBD: Geräten benutzt, die *LstOut* Routine von dem LST: Gerät, die *AuxOut* und *AuxIn* Routinen werden vom AUX: Gerät und die *UsrOut* und *UsrIn* Routinen von dem USR: Gerät benutzt.

Laut Voreinstellung sind diese Treiber dem BDOS System zugewiesen, wie in den geschweiften Klammern in der obigen Liste der Treiber gezeigt.

Dies kann vom Programmierer geändert werden, indem die Adresse einer selbstdefinierten Treiberprozedur oder -funktion an eine der folgenden Standardvariablen zugewiesen wird.

Variable	**enthält die Adresse der**
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 Benutzer definierte Treiberprozedur oder -funktion muß den oben gegebenen Definitionen entsprechen, d.h. ein *ConSt* Treiber muß eine *Boolean* Funktion sein, ein *ConIn* Treiber muß eine *char* Funktion sein, usw..

21.13 Externe Unterprogramme

Das reservierte Wort **external** wird gebraucht, um externe Prozeduren und Funktionen aufzurufen, gewöhnlich Prozeduren und Funktionen, die in Assembler geschrieben sind.

Dem reservierten Wort **external** muß eine Stringkonstante folgen, die eine Datei benennt, in der der auszuführende Assemblercode für die externe Prozedur oder Funktion enthalten sein muß.

Während der Compilierung eines Programms, das externe Prozeduren oder Funktionen enthält, werden die entsprechenden Dateien geladen und in den Objectcode plaziert. Da es nicht möglich ist, im voraus zu wissen, wo genau im Objektcode der externe Code plaziert wird, **muß** dieser Code verschiebbar sein und es dürfen keine Bezüge zum Datensegment bestehen. Darüberhinaus muß der externe Code die Register BP, CS, DS und SS sichern und vor der Ausführung der RET Anweisung wiederherstellen.

Ein externes Unterprogramm hat keinen *Block*, d.h. keinen Deklarations- und keinen Anweisungsteil. Es wird nur der Programmkopf des Unterprogramms genannt, dem das reservierte Wort **external** unmittelbar folgen muß, sowie der Dateiname für das Unterprogramm.

Die Dateitypenbezeichnung ist .CMD; nur das Codesegment einer .CMD Datei wird geladen.

Beispiel:
```
procedure DiskReset; external 'DSKRESET';
function IOstatus: boolean; external 'IOSTAT';
```

Parameter können an externe Unterprogramme übergeben werden. Die Syntax ist genauso, wie bei Aufrufen von normalen Prozeduren und Funktionen:

```
procedure Plot(X,Y: Integer); external 'PLOT';
function QuickSort (var List: PartNo); external 'QS';
```

Die Übergabe externer Unterprogramme und Parameter wird auf Seite 252 weiter besprochen.

21.14 In-line Maschinencode

TURBO Pascal bietet die Möglichkeit **inline** Anweisungen zu verwenden , mit deren Hilfe, auf einfache Art, Assembleranweisungen direkt in den Programmtext eingefügt werden können. Eine Inline-Anweisung setzt sich wie folgt zusammen: aus dem reservierten Wort **inline** und einem oder mehreren *Codeelementen*, die durch Schrägstriche voneinander getrennt sind und in runde Klammern gesetzt sind.

Ein Codeelement besteht aus ein oder mehreren Datenelementen, getrennt durch die Zeichen plus (+) oder minus (-). Ein Datenelement ist entweder eine *integer* Konstante, ein Variablenbezeicher, ein Prozedurbezeichner, ein Funktionsbezeichner oder eine Kommandozählerreferenz. Eine Kommandozählerreferenz wird als Sternzeichen (*) geschrieben.

Beispiel:

Inline (10/$2345/count+1/sort-*+2);

Jedes Codeelement erzeugt eine Byte oder ein Wort (zwei Bytes) Code. Der Byte- oder Wortwert wird berechnet, indem die Werte der Datenelemente, gemäß ihrer Trennungszeichen, entweder addiert oder subtrahiert werden. Der Wert des Variablenbezeichners ist die Adresse (oder Offset) der Variablen. Der Wert eines Prozedur- oder eines Funktionsbezeichners ist die Adresse (oder Offset) der Prozedur oder Funktion. Der Wert einer Kommandozählerreferenz ist die Adresse (oder Offset) des Kommandozählers, d.h. die Adresse, auf der das nächste Byte (Code) erzeugt werden soll.

Ein Codeelement erzeugt ein Byte Code, wenn es nur aus *integer* Konstanten besteht und sein Wert innerhalb des 8-Bit Bereichs (0..255) liegt. Liegt der Wert nicht in diesem Bereich, oder verweist der Code auf Variablen-, Prozedur- oder Funktionsbezeichner, oder enthält das Codeelement eine Kommandozählerreferenz, dann wird ein Wort Code erzeugt (zuerst das niederwertigste Byte).

Die '<' und '>' Zeichen können zur Überschreibung der automatischen Größenwahl, wie oben beschrieben, benutzt werden. Wenn ein Codeelement mit einem '<' Zeichen beginnt, wird lediglich das niederwertigste Byte des Wertes codiert, auch wenn es ein 16-Bit Wert ist. Wenn ein Codeelement mit einem '>' Zeichen beginnt, wird immer ein Wort codiert, auch wenn das höchste Byte null ist.

Beispiel:

inline (<$1234/>$44);

Diese **inline** Anweisung erzeugt drei Bytes Code: $34, $44, $00.

Der Wert eines Variablenbezeichners, der in einer **inline** Anweisung benutzt wird, ist die Offsetadresse der Variablen innerhalb ihres Grundsegments. Das Grundsegment globaler Variablen (das sind im Hauptprogrammblock deklarierte Variablen) ist das Datensegment, das durch das DS Register zugänglich ist. Das Grundsegment lokaler Variablen (das sind innerhalb des gegenwärtigen Unterprogramms deklarierte Variablen) ist das Stacksegment, und in diesem Fall ist das Variablenoffset von dem BP (base page) Register abhängig, dessen Verwendung das Stacksegment automatisch auswählen läßt. Das Grundsegment typisierter Konstanten ist das Codesegment, das durch das CS Register erreichbar ist. **inline** Anweisungen sollten nicht versuchen auf Variablen zuzugreifen, die weder im Hauptprogramm, noch im gegenwärtigen Unterprogramm deklariert sind.
Das folgende Beispiel einer Inline-Anweisung erzeugt Assemblercode, der alle Zeichen in seinem Stringargument in Großbuchstaben umwandelt.

```
procedure UpperCase(var Strg: Str);
{Str is type String[255]}
begin
  inline
    ($C4/$BE/Strg/              { LES   DI,Strg[BP]}
    $26/$8A/$0D/                { MOV   CL,ES:[DI]}
    $FE/$C1/                    { INC   CL}
    $FE/$C9/                    {L1:  DEC   CL}
    $74/$13/                    { JZ    L2}
    $47/                        { INC   DI}
    $26/$80/$3D/$61/            { CMP   ES:BYTE PTR[DI], 'a'}
    $72/$F5/                    { JB    L1}
    $26/$80/$3D/$7A/            { CMP   ES:BYTE PTR[DI], 'z'}
    $77/$EF/                    { JA    L1}
    $26/$80/$2D/$20/            { SUB   ES:BYTE PTR[DI], 20H}
    $EB/$E9);                   { JMP   SHORT L1}
                                {L2:  }
end;
```

Inline-Anweisungen können im Anweisungsteil eines Blocks beliebig unter andere Anweisungen gemischt werden und sie können alle CPU-Register benutzen. **Beachten Sie** jedoch, daß der Inhalt der Register BP, SP, DS und SS nach Abschluß der Operation derselbe sein muß, wie zu Beginn.

21.15 Interrupt-Handhabung

Eine TURBO Pascal Interruptroutine muß die benutzten Register AX, BX, CX, DX, DI, DS und ES unverändert lassen. Dies geschieht, indem man die folgende Inline-Anweisung der Prozedur voranstellt:

inline ($50/$53/$51/$52/$56/$57/$1E/$06/$FB);

Das letzte Byte ($FB) ist eine STI Instruktion, die weitere Interrupts - erforderlich oder nicht - ermöglicht. Mit der folgenden Inline-Anweisung muß die Prozedur abgeschlossen werden:

Inline ($07/$1F/$5F/$5E/$5A/$59/$5B/$58/$8B/$E5/$5D/$CF);

Dies setzt die Register wieder ein und lädt sowohl den Stackzeiger (SP), als auch das BP Register wieder. Das letzte Byte ($CF) ist eine IRET Instruktion ($CF), die die RET Instruktion, welche vom Compiler erzeugt wird, überschreibt.

Interruptprozeduren sollten keine I/O Operationen verwenden, die die Standardprozeduren und -funktionen von TURBO Pascal benutzen, da ein Zugriff auf das BDOS nicht mehrfach möglich ist. Der Programmierer muß den Interruptvektor initialisieren, mit dem die Interruptroutine aktiviert wird.

21.15.1 Intr-Prozedur

Syntax: Intr(*InterruptNo,Result*)

Diese Prozedur ruft die Register und Flags entsprechend ihrer Nennung im Parameter *Result* auf. Der Parameter ist wie folgt typisiert:

```
Result = record
           AX,BX,CX,BP,SI,DI,DS,ES,Flags: Integer;
         end;
```

So wird durch den Parameter *interruptNo*, der eine *integer* Konstante sein muß, ein Softwareinterrupt eingeleitet. Wenn die Interruptroutine beendet ist und Ihr Programm weiterabläuft, wird *Result* alle Werte enthalten, die von der Routine ermittelt wurden.

Beachten Sie, daß das Datensegmentregister DS, das für den Zugriff auf globale Variablen benutzt wird, einen falschen Wert erhält, wenn die Interruptroutine eingegeben ist. Deshalb kann auf globale Variablen nicht direkt zugegriffen werden. *Typisierte Konstanten* sind jedoch zugänglich, da sie in dem Codesegment gespeichert sind. Eine Möglichkeit auf globale Variablen in einer Interruptroutine zuzugreifen, ist deshalb das Speichern des Wertes von *Dseg* als typisierte Konstante im Hauptprogramm. Diese typisierte Konstante kann dann von dem Interrupt Anwender erreicht und zur Setzung seines DS Registers verwendet werden.

21.16 Interne Datenformate

In den folgenden Beschreibungen bedeutet das Symbol *ofs* das Offset des ersten Byte, das von einer Variable gegebenen Typs innerhalb seines Segments besetzt ist. Die Segmentbasisadresse kann bestimmt werden, indem man die Standardfunktion *Seg* benutzt.

Globale Variablen, lokale Variablen und *typisierte Konstanten* besetzen verschiedene Segmente, wie folgt:

Globale Variablen liegen im Datensegment, das Offset bezieht sich auf das DS Register.

Lokale Variablen liegen im Stacksegment, das Offset bezieht sich auf das BP Register.

Typisierte Konstanten liegen im Codesegment, das Offset bezieht sich auf das CS Register.

Alle Variablen sind in ihrem Basissegment enthalten.

21.16.1 Grundtypen von Daten

Die Grundtypen von Daten können in Strukturen (Arrays, Records und Diskettendateien) gruppiert werden, wobei diese Strukturierung ihre internen Formate nicht beeinflußt.

21.16.1.1 Skalare

Die folgenden Skalare werden alle in einem Byte gespeichert: *Integer* Teilbereiche mit beiden Grenzen im Bereich 0..255, *boolean*, *char* und deklarierte Skalare mit weniger als 256 möglichen Werten. Dieses Byte enthält den ordinalen Wert dieser Variablen.

Die folgenden Skalare werden alle in zwei Byte gespeichert: *Integer*, *integer* Teilbereiche mit einer oder beiden Grenzen außerhalb des Bereichs 0..255 und deklarierte Skalare mit mehr als 256 möglichen Werten. In diesen Bytes ist das niederwertigste Byte zuerst gespeichert.

21.16.1.2 Reelle Zahlen

Reelle Zahlen (Datentyp: *real*) belegen 6 Byte, ein Gleitkommawert hat eine 40-Bit Mantisse und einen 8-Bit Exponenten. Der Exponent wird im ersten Byte gespeichert, die Mantisse in den nächsten fünf Byte, mit dem niederwertigsten Byte als ersten:

@	Exponent
@+1	LSB der Mantisse
:	
@+5	MSB der Mantisse

Der Exponent verwendet binäres Format mit einer Voreinstellung von $80. Daher bedeutet ein Exponent von $84, daß der Wert der Mantisse mit 2^ ($84-$80) = 2^4=16 potenziert werden muß. Wenn der Exponent null ist, wird der Gleitkommawert als null angesehen.

Den Wert der Mantisse erhält man, indem man die 40-Bit vorzeichenlose ganze Zahl durch 2^40 teilt. Die Mantisse ist immer normalisiert, d.h. das signifikanteste Bit (Bit 7 des fünften Byte) sollte als 1 interpretiert werden. Das Vorzeichen der Mantisse wird in diesem Bit gespeichert, eine 1 zeigt an, daß die Zahl negativ ist, eine 0, daß sie positiv ist.

21.16.1.3 Strings

Ein String besetzt soviele Bytes, wie seine maximale Länge plus eins ist. Das erste Byte enthält die gegenwärtige Länge des String. Die folgenden Bytes enthalten den String, wobei das erste Zeichen an der niedrigsten Adresse gespeichert ist. In der Tabelle unten, bedeutet **L** die gegenwärtige Länge des String und *Max* bedeutet die maximale Länge:

@	Gegenwärtige Länge (L)
@ +1	erstes Zeichen
@ +2	zweites Zeichen
:	
:	
@ +L	letztes Zeichen
@ L+1:	unbenutzt
:	
@ +Max	unbenutzt

21.16.1.4 Mengen

Ein Element einer Menge (engl: set) belegt ein Bit. Die maximale Zahl von Elementen einer Menge ist 256. Eine Mengenvariable belegt nie mehr als 32 Byte (256/8).

Wenn eine Menge weniger als 256 Elemente enthält, sind einige Bits auf Null gesetzt und müssen deshalb nicht gespeichert werden. Im Interesse der Speichereffizienz wäre die beste Art, eine Mengenvariable des gegebenen Typs zu speichern, die nicht-signifikanten Bits wegzuschneiden und die verbleibenden Bits so zu rotieren, daß das erste Element der Menge das erste Bit des ersten Byte belegt. Solche Rotationsoperationen sind sehr langsam, weshalb TURBO einen Kompromiß anwendet: Nur Bytes, die statisch null sind (d.h. Bytes, bei denen kein Bit gebraucht wird) werden nicht gespeichert. Diese Methode der Kompression ist sehr schnell und in den meisten Fällen so speichereffizient wie die Rotationsmethode.

Die Zahl der Bytes, die durch eine Mengenvariable belegt sind, wird mit (*Max* **div** 8) - (*Min* **div** 8) + 1 berechnet, wobei *Min* und *Max* untere und obere Grenzen des Grundtyps einer Menge sind. Die Speicheradresse eines spezifischen Elements *E* ist:

MemAddress = @ + (*E* **div**8) - (*Min* **div** 8)

und die Bitadresse innerhalb des Byte bei der MemAddress ist:

BitAddress = *E* **mod** 8

E entspricht dem ordinalen Wert des Elements.

21.16.1.5 Zeiger

Ein Zeiger besteht aus vier Bytes, die eine Segmentbasisadresse und ein Offset enthalten. Die beiden niederwertigsten Bytes enthalten das Offset und die beiden höchstwertigsten enthalten die Segmentbasisadresse. Beide sind im umgekehrten Byteformat abgespeichert, d.h. mit dem niederwertigsten Byte zuerst. Der Wert **nil** entspricht zwei Nullwörtern.

21.17 Datenstrukturen

Datenstrukturen werden aus den Grundtypen von Daten unter Verwendung verschiedener Strukturierungsmethoden gebildet. Es gibt drei verschiedene Strukturierungsmethoden: Arrays, Records und Diskettendateien. Die Strukturierung von Daten beeinträchtigt in keiner Weise das interne Format der Grundtypen von Daten.

21.17.1 Arrays

Die Komponenten mit dem niedrigsten Indexwert werden an der niedrigsten Speicheradresse gelagert. Ein mehrdimensionales Array wird der Wertigkeit nach abgespeichert; d.h. die rechteste Dimension nimmt als erstes zu. Z.B. erhalten Sie bei Eingabe des Arrays:

```
Board: array[1..8,1..8] of Square
```

das folgende Speicher-Layout seiner Komponenten:

Board[1,1] niedrigste Adresse
Board[1,2]
:
:
Board[1,8]
Board[2,1]
Board[2,2]
:
:
Board[8,8] höchste Adresse

21.17.2 Records

Das erste Feld eines Records wird an der niedrigsten Speicheradresse gespeichert. Wenn der Record aus nicht veränderbaren Elementen zusammengesetzt ist, ist seine Länge durch die Summe der Längen der einzelnen Felder gegeben. Enthält der Record veränderbare Elemente, ergibt sich die Gesamtzahl der von ihm belegten Bytes aus der Summe der Länge der festen Elemente und der Länge des größten seiner veränderlichen Elemente. Jedes veränderliche Element beginnt bei derselben Speicheradresse.

21.17.3 Diskettendateien

Diskettendateien unterscheiden sich von anderen Datenstrukturen insofern, als Daten nicht im internen Speicher abgelagert werden, sondern in einer Datei eines externen Mediums. Eine Diskettendatei wird mit Hilfe von File Interface Block (FIB) bearbeitet.

21.17.3.1 File Interface Blocks

Folgende Tabelle zeigt das Format eines FIB:

@ +0	Flag - Byte.
@ +1	Zeichenpuffer.
@ +2	Zahl der Records (LSB) oder Pufferoffset (LSB).
@ +3	Zahl der Records (MSB) oder Pufferoffset (MSB).
@ +4	Recordlänge (LSB) oder Puffergröße (LSB).
@ +5	Recordlänge (MSB) oder Puffergröße (MSB).
@ +6	Pufferzeiger (LSB).
@ +7	Pufferzeiger (MSB).
@ +8	Gegenwärtiges Record (LSB) oder Pufferende (LSB).
@ +9	Gegenwärtiges Record (MSB) oder Pufferende (MSB).
@ +10	Unbenutzt.
@ +11	Unbenutzt.
@ +12	Erstes Byte von CP/M FCB.
.	
.	
@ +47	Letztes Byte von CP/M FCB.
@ +48	Erstes Byte des Sektorpuffer.
.	
.	
@ +175	Letztes Byte des Sektorpuffer.

Das Format der Flags-Byte ist bei @+0 wie folgt:

Bit 0..3 Dateityp.
Bit 4 Read-Signal.
Bit 5 Write-Signal oder Pre-read Zeichen-Flag.
Bit 6 Output-Flag.
Bit 7 Input-Flag.

Dateityp 0 kennzeichnet eine Diskettendatei, und durch 1 bis 5 werden die logischen I/O Geräte von TURBO Pascal benannt (CON:, KBD:, LST:, AUX:, und USR:). Bei typisierten Dateien wird Bit 4 gesetzt, wenn der Inhalt des Sektorpuffers undefiniert ist. Bei Textdateien wird Bit 5 gesetzt, wenn der Zeichenpuffer ein Pre-read Zeichen enthält, Bit 6, wenn Output erlaubt ist, und Bit 7, wenn Input erlaubt ist.

Bei typisierten und nicht-typisierten Dateien speichern die vier Wörter von @+2 bis @+9 die Zahl der Records in der Datei, die Recordlänge in Bytes, den Sektorpufferzeiger und die gegenwärtige Recordnummer. Für typisierte Dateien, speichert der Sektorpufferzeiger ein Offset (0..127) im Sektorpuffer bei @+48. Der FIB einer nicht-typisierten Datei hat keinen Sektorpuffer, der Sektorpufferzeiger wird folglich nicht benutzt.

Bei Textdateien speichern die vier Wörter von @+2 bis @+9 die Offsetadresse des Puffers, seine Größe, das Offset des nächsten zu lesendenden oder zu schreibenden Zeichens und das Offset des ersten Byte nach dem Puffer. Der Puffer liegt immer im selben Segment wie der FIB und beginnt gewöhnlich bei @+48. Wenn eine Textdatei einem logischen Gerät zugewiesen wird, werden lediglich die Flags-Byte und der Zeichenpuffer benutzt.

21.17.3.2 Dateien mit direktem Zugriff

Eine Datei mit direktem Zugriff (engl. random access) besteht aus einer Folge von Records, die alle die gleiche Länge und das gleiche Format besitzen. Um die Dateispeicherkapazität zu optimieren, sind die Records einer Datei direkt aneinandergrenzend. Die ersten vier Bytes des ersten Sektors einer Datei enthalten die Zahl der Records in der Datei und die Länge jedes Records in Bytes. Die Speicherung des ersten Records der Datei beginnt mit dem vierten Byte.

Sektor 0, Byte 0: Zahl der Records (LSB)
Sektor 0, Byte 1: Zahl der Records (MSB)
Sektor 0, Byte 2: Recordlänge (LSB)
Sektor 0, Byte 3: Recordlänge (MSB)

21.17.3.3 Textdateien

Die Grundelemente einer Textdatei sind Zeichen, aber weiterhin ist eine Textdatei in *Zeilen* unterteilt. Jede Zeile besteht aus einer beliebigen Zahl von Zeichen, die von einer CR/LF Sequenz (ASCII $OD/$OA) beendet wird. Die Datei wird mit Ctrl-Z (ASCII $1B) abgeschlossen.

21.18 Parameter

Parameter werden an Prozeduren und Funktionen mittels des Stack übertragen, der durch SS:SP adressiert ist.

Zu Beginn eines **external** Unterprogramms enthält die Spitze des Stack immer die Rückkehradresse innerhalb des Codesegments (ein Wort). Die Parameter, falls vorhanden, liegen unterhalb der Rückkehradresse, d.h. an höheren Adressen im Stack.

Wenn eine externe Funktion den folgenden Unterprogrammkopf hat:

```
function Magic(var R: Real; S: string5): Integer;
```

dann hätte der Stack am Eingang zu *Magic* den folgenden Inhalt:

```
( Function result )
( Segment base address of R )
( Offset address of R )
( First character of S )
:
( Last character of S )
( Length of S )
( Return address ) SP
```

Eine externe Unterroutine sollte das Basisseitenregister (Base Page, bzw. BP) sichern und dann den Stackzeiger (Stack Pointer, bzw. SP) in das Basisseitenregister kopieren, um auf Parameter verweisen zu können. Weiterhin sollte die Unterroutine auf dem Stack Platz für den lokalen Arbeitsbereich reservieren. Dies kann kann durch die folgenden Instruktonen erreicht werden:

```
PUSH   BP
MOV    BP, SP
SUB    SP, WORKAREA
```

Die letzten Instruktionen fügen Folgendes zum Stack hinzu:

```
( Return address ) BP
( The saved BP register )
( First Byte of local workarea )
:
( Last Byte of local work area ) SP
```

Parameter werden über des BP Registers erreicht.

Die folgende Instruktion lädt die Länge des Strings in das AL Register:

```
MOV AL,|BP-1|
```

Bevor man eine RET Instruktion ausführt, muß der Stackzeiger und das Basisseitenregister auf die ürsprünglichen Werte zurückgesetzt werden. Wenn der RET ausgeführt wird, können die Parameter entfernt werden, indem man RET einen Parameter angibt, der spezifiziert, wieviele Bytes entfernt werden sollen. Die folgenden Instruktionen sollten deshalb beim Austritt aus einem Unterprogramm benutzt werden:

```
MOV     SP,BP
POP     BP
RET     NoOfBytesToRemove
```

21.18.1 Variablenparameter

Ein Variablenparameter (**var**) überträgt zwei Worte auf den Stack, in denen die Basisadresse und das Offset des ersten durch den aktuellen Parameter belegten Bytes angegeben wird.

21.18.2 Wertparameter

Bei Wertparametern hängen die auf den Stack übertragenen Daten vom Typ des Parameters ab, wie in den nächsten Abschnitten beschrieben wird.

21.18.2.1 Skalare

Integer, *Boolean*, *Char* und deklarierte Skalare (d.h. alle Skalare außer *Real*) werden als ein Wort auf den Stack übertragen. Wenn die Variable nur ein Byte belegt, wenn sie gespeichert wird, ist das höchste Byte des Parameters null.

21.18.2.2 Reelle Zahlen

Eine reelle Zahl wird auf den Stack unter Verwendung von sechs Bytes übertragen.

21.18.2.3 Strings

Wenn ein String an der Spitze des Stack ist, enthält das oberste Byte die Länge des String, es folgen dann die Zeichen des String.

21.18.2.4 Mengen

Eine Menge belegt immer 32 Bytes im Stack (die Kompression von Mengen wird nur beim Laden und Speichern von Mengen angewendet).

21.18.2.5 Zeiger

Ein Zeigerwert wird auf den Stack als zwei Worte übertragen, die die Basisadresse und das Offset einer dynamischen Variablen enthalten. Der Wert **NIL** entspricht zwei Nullwörtern.

21.18.2.6 Arrays und Records

Auch wenn sie als Wertparameter verwendet werden, werden *Array*- und *Record*-Parameter tatsächlich nicht auf den Stack übertragen. Stattdessen werden zwei Worte übertragen, die die Basisadresse und das Offset des ersten Byte des Parameters enthalten. Es liegt dann in der Verantwortung der Unterroutine, von dieser Information Gebrauch zu machen und eine lokale Kopie dieser Variablen zu erstellen.

21.18.3 Funktionsergebnisse

Vom Benutzer geschriebene, **external** Funktionen müssen vor ihrem Rücksprung erst alle Parameter und das Funktionsergebnis vom Stack entfernen.

Vom Benutzer geschriebene, **external** Funktionen müssen ihre Ergebnisse exakt wie im folgenden angegeben, weitergeben:

Die Werte von skalaren Typen, außer *Real*, müssen in das AX Register ausgegeben werden. Wenn das Ergebnis nur ein Byte ist, dann sollte AH auf null gesetzt werden. *Boolean* Funktionen müssen den Funktionswert ausgeben, indem sie die Z Flag (Z= Falsch, NZ= Wahr) setzen.

Real Zahlen müssen auf den Stack mit dem Exponenten an der niedrigsten Adresse ausgegeben werden. Dies geschieht, wenn die Funktionsergebnisvariable bei der Ausgabe nicht entfernt wird.

Mengen müssen an die Spitze des Stack in Übereinstimmung mit dem auf Seite 254 beschriebenen Format ausgegeben werden. Am Ausgang muß SP auf das Byte zeigen, das die Stringlänge enthält.

Zeigerwerte müssen in DX:AX ausgegeben werden.

21.18.4 Der Heap und die Stacks

Während der Ausführung eines TURBO Pascal Programms sind dem Programm folgende Segmente zugewiesen:

ein Codesegment
ein Datensegment und
ein Stacksegment

Zwei stapelartige Strukturen werden während der Programmausführung verwaltet: der *Heap* und der *Stack*.

Der Heap wird benutzt, um dynamische Variablen zu speichern und wird mit den Standardprozeduren *New,Mark* und *Release* kontrolliert. Am Beginn eines Programms wird der Heapzeiger *HeapPtr* auf eine niedrige Stelle des Speichers im Stacksegment gesetzt. Der Heap wächst dann aufwärts gegen den Stack. Die vordefinierte Variable *HeapPtr* enthält den Wert des Heapzeiger und erlaubt es dem Programmierer, die Position des Heap zu kontrollieren.

Der Stack wird benutzt, lokale Variablen und Zwischenergebnisse bei der Berechnung von Ausdrücken zu speichern und Parameter an Prozeduren und Funktionen zu übertragen. Am Beginn eines Programms ist der Stackzeiger auf die Adresse an der Spitze des Stacksegments gesetzt.

Bei jedem Aufruf der Prozedur *New* und bei der Eingabe einer Prozedur oder Funktion prüft das System, ob sich ein Zusammenstoß zwischen Heap und Recursion-Stack ereignet hat. War dies der Fall, entsteht ein Ausführungsfehler, wenn nicht der Compilerbefehl passiv gesetzt ist (|$K-|).

21.19 Speicherverwaltung

Bei der Ausführung eines TURBO Programms, werden drei Segmente dem Programm zugewiesen: Ein Codesegment, ein Datensegment und ein Stacksegment.

Codesegment (CS ist das Codesegmentregister):

CS:0000 - CS:00FR	Laufzeit-Librarycode
CS:EOFR - CS:EOFP	Programmcode
CS:EOFP - CS:EOFC	Unbenutzt.

Datensegment (DS ist das Datensegmentregister):

DS:0000 - DS:00FF	CP/M-86 Basisseitenregister
DS:0100 - DS:EOFW	Laufzeit-Libraryspeicher
DS:EOFW - DS:EOFM	Hauptprogramm-Blockvariablen
DS:EOFM - DS:EOFD	Unbenutzt.!

Die unbenutzten Bereiche zwischen (CS:EOFP-CS:EOFC und DS:EOFM-DS:EOFD) werden nur angewiesen, wenn bei der Compilierung mindestens eine größere COdesegmentgröße, als die erforderliche Größe, spezifiziert wird. Die Größe des Code- und Datensegments kann jeweils 64 K Bytes nicht überschreiten.

Das Stacksegment ist ein wenig komplizierter, da es größer als 64 K Bytes sein kann. Zu Beginn des Programms wird das Stacksegmentregister (SS) und der Stackzeiger (SP) geladen, so daß SS:SP auf das allerletzte verfügbare Byte des gesamten Segments zeigt. Während der Programmausführung wird SS nie geändert, SP dagegen kann sich abwärts bewegen, bis es den untersten Teil des Segments, oder 0 erreicht hat (entsprechend den 64K Bytes des Stack), wenn das Stacksegment größer als 64 K Bytes ist.

Der Heap wächst von dem unteren Speicher in dem Stacksegment zu dem aktuellen Stack, der in dem oberen Speicher liegt. Immer wenn dem Heap eine Variable zugewiesen wird, wird der Heapzeiger (der eine Doppelwort-Variable ist, die vom TURBO Laufzeitsystem angelegt und verwaltet wird) nach oben bewegt und dann normalisiert, so daß die Offsetadresse immer zwischen $0000 und $000F liegt. Deshalb ist die maximale Größe einer einzelnen Variablen, die dem Heap zugewiesen werden kann, 65521 Bytes (gemäß $1000 weniger $000F). Die Gesamtgröße aller Variablen, die dem Heap zugewiesen werden kann, ist jedoch nur durch den zur Verfügung stehenden Speicherplatz begrenzt.

Der Heapzeiger steht dem Programmierer durch den *HeapPtr* Standardbezeichner zur Verfügung. *HeapPtr* ist ein typloser Zeiger, der zu allen Zeigertypen kompatibel ist. Zuweisungen durch *HeapPtr* sollten nur mit äußerster Vorsicht durchgeführt werden.

Anmerkungen

22. CP/M-80

Dieser Anhang beschreibt Eigenschaften von TURBO Pascal, die spezifisch für die 8-Bit Implementation sind. Er enthält zwei Arten von Informationen:

1) Für den effizienten Gebrauch von TURBO Pascal **unbedingt nötige** Informationen. Diese sind auf den Seiten 259 bis 272 beschrieben.

2) Die restlichen Abschnitte beschreiben Einzelheiten, die nur für erfahrene Programmierer interessant sind, z.B. den Aufruf von Assemblerroutinen, technische Aspekte des Compilers, usw..

22.1 eXecute-Kommando

Sie finden in der CP/M-80 Version im Hauptmenü von TURBO ein zusätzliches Kommando: e**X**ecute. Es erlaubt Ihnen andere Programme von Turbo aus laufen zu lassen, z.B. Kopier-, Textverarbeitungsprogramme usw. - einfach alles, was Sie auf Ihrem Betriebssytem laufen lassen können. Wenn Sie **X** eingeben, erscheint die Meldung:

```
Command: ■
```

Sie können daraufhin den Namen jedes beliebigen Programms eingeben, das dann geladen wird und normal abläuft. Am Ende des Programms geht die Kontrolle wieder an TURBO Pascal über. Sie erkennen das an dem Wiedererscheinen des TURBO Prompts **>**.

22.2 Compiler-Optionen

Das **O** Kommando wählt das folgende Menü an, in dem Sie einige voreingestellte Werte des Compilers sehen und verändern können. Es ist auch beim Finden von Laufzeit-Fehlern in Programmen, die in Objektcode-Dateien compiliert sind, hilfreich.

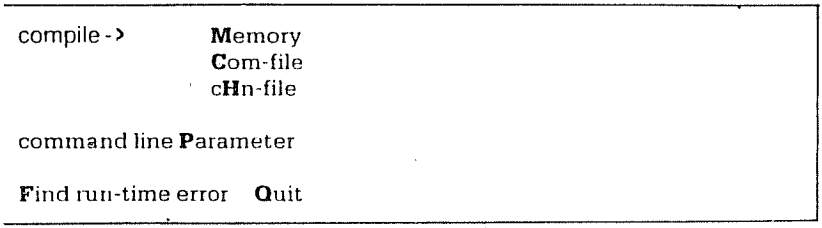

Abbildung 22 - 1: Optionen-Menü

22.2.1 Memory / Com-Datei / cHn-Datei

Die drei Befehle **M**, **C** und **H** steuern die Verarbeitungsart der Source und die Ablage des erzeugten Object-Codes durch den Compiler.

Memory (Arbeitsspeicher) ist der voreingestellte Modus. Der Code wird im Speicher erzeugt und behalten. Das Programm kann dann direkt vom Speicher aus durch den **R**un-Befehl ausgeführt werden.

Com-File wird durch Eingabe **C** gewählt und durch den Pfeil angezeigt. Der Code wird im Falle der Aktivierung auf eine Datei mit demselben Namen wie die Arbeitsdatei (oder Hauptdatei, falls angegeben) als .COM-File geschrieben. Diese Datei enthält den Object-Code und die Pascal 'runtime libary'. Programme, die auf diese Weise compiliert werden, können größer sein als im Speicher compilierte Programme, da der Object-Code selbst keinen Speicherplatz während der Compilierung braucht und bei einer niedrigeren Adresse beginnt.

c**H**ain-file wird gewählt, indem man **H** drückt. Der Pfeil bewegt sich auf die entsprechende Zeile. Nach der Aktivierung wird der Code auf eine Datei mit demselben Namen wie die Arbeitsdatei (oder Hauptdatei, falls angegeben) als .CHN-File geschrieben. Diese Datei enthält den Programmcode, aber nicht die Pascal 'runtime libary' und muß von einem anderen TURBO Pascal Programm aus mit der Prozedur *Chain* aktiviert werden (siehe Seite 263).

Wenn der **C**om oder c**H**n Modus gewählt wird, erweitert sich das Menü um folgende zwei Zeilen:

```
Start address:   XXXX   (min YYYY)
End address:     XXXX.  (max YYYY)
```

Abbildung 22 - 2: Start- und Endadressen

22.2.2 Start-Addresse

Start address gibt die Adresse (in hexadezimal) des ersten Bytes des Codes an. Das ist normalerweise die Endadresse der Pascal Library plus eins, kann aber auch auf eine höhere Adresse verändert werden, falls man Platz reservieren will, z.B. für absolute Variablen, die einer Reihe von verketteten Programmen gemeinsam sind.

Wenn Sie ein **S** eingeben, werden Sie veranlaßt eine neue Startadresse einzugeben. Falls sie nur <RETURN> drücken, wird der kleinste Wert angenommen.

Setzen Sie die Startadresse nicht niedriger als den minimalen Wert, da der Code dann Teile der Pascal Library überschreibt.

22.2.3 End-Adresse

End Address gibt die höchste für das Programm verfügbare Adresse an (hexadezimal). Der Wert in Klammern zeigt die Spitze der TPA auf ihrem Computer, d.h. BDOS minus eins. Die voreingestellte Einstellung ist 700 bis 1000 Bytes weniger, um Platz für den Lader zu lassen, der genau unter BDOS residiert, wenn Programme von TURBO aus ausgeführt werden.

Wenn compilierte Programme in einer anderen Umgebung laufen sollen, kann die Endadresse verändert werden, um sie der TPA-Größe dieses Systems anzupassen. Falls Sie voraussehen, daß ihre Programme auf einer Reihe verschiedener Computer laufen sollen, ist es günstig diesen Wert relativ niedrig zu setzen, z.B. C100 (48K) oder A100 (40k), wenn das Programm unter MP/M laufen soll.

Wenn Sie **E** eingeben, werden Sie aufgefordert eine Endadresse einzugeben. Wenn Sie <RETURN> drücken, wird der voreingestellte Wert angenommen (d.h. die Spitze von TPA weniger 700 oder 1000 Bytes). Geben Sie eine höhere End adress als diese an, können die Programme von TURBO nicht ausgeführt werden, da sie sonst den TURBO Lader überschreiben. Setzen Sie die Endadresse höher als das obere Ende der TPA, überschreiben die resultierenden Programme Teile vom BDOS, wenn sie auf ihrer Maschine laufen.

22.2.4 Kommandozeilen-Parameter

Das **P** Kommando erlaubt Ihnen einen oder mehrere Parameter einzugeben, die an Ihr Programm übergeben werden, wenn es im Memory-Modus läuft. Dies geschieht genauso wie bei der Eingabe von Parametern in der DOS Kommandozeile. Diese Parameter können durch die Funktionen *ParamCount* und *ParamStr* angesprochen werden.

2.2.5 Finden von Laufzeit-Fehlern

Wenn Sie ein im Speicher compiliertes Programm laufen lassen, und es tritt ein Laufzeit-Fehler auf, wird der Editor aufgerufen und der Fehler automatisch aufgezeigt. Dies ist natürlich nicht möglich, wenn das Programm in einer .COM oder .CHN Datei steht. Laufzeit-Fehlermeldungen zeigen dann den Fehlercode und den Wert des Programmzählers zur Zeit des Fehlers an, z.B.:

```
Run-time error 01, PC=1B56
Program aborted
```

Abbildung 22- 3: Laufzeit-Fehlermeldung

Um die Stelle im Source zu finden, an der der Fehler passierte, müssen Sie den **F** Befehl im **O**ptionsmenü eingeben. Wenn die Bereitschaftsmeldung für die Adresse da ist, geben Sie die von der Fehlermeldung angegebene Adresse ein.

```
Enter PC: 1B56
```

Abbildung 22 - 4: Finden eines Laufzeit-Fehlers

Die Stelle im Source wird jetzt gefunden und genauso ausgegeben, als ob der Fehler während eines Programmlaufs im Speicher passiert wäre.

22.3 Standardbezeichner

Die folgenden Standardbezeichner gelten nur für die CP/M-80 Implementation:

```
Bios        Bdos        RecurPtr
BiosHl      BdosHl      StackPtr
```

22.4 Chain und Execute

TURBO PASCAL enthält die zwei Standardprozeduren *Chain* und *Execute*, die es Ihnen erlauben, von einem TURBO Programm aus andere Programme zu aktivieren. Die Syntax des Prozeduraufrufs ist:

```
Chain(FilVar)
Execute(FilVar)
```

wobei *FilVar* eine Dateivariable beliebigen Typs ist, die zuvor mit der Standardprozedur *Assign* einer Diskettendatei zugeordnet wurde. Wenn die Datei existiert, wird sie in den Speicher geladen und ausgeführt.

Die *Chain* Prozedur wird nur benutzt, um spezielle .CHN TURBO Pascal Dateien zu aktivieren, d.h. Dateien, die mit der cHn-Datei Option des Optionenmenüs compiliert wurden (siehe Seite 260). Solche Dateien enthalten nur Object-Code, aber keine Pascal Library mehr. Diese wird schon mit dem Hauptprogramm in den Speicher geladen und an der Startadresse des gegenwärtigen Programms, d.h. der Adresse, die sich beim Compilieren dieses Programms ergibt, ausgeführt. Das .CHN-File wird bei Aufruf aus dem Hauptprogramm in den Speicher geladen und das Hauptprogramm muß deshalb die gleiche Startadresse haben, wie das aufgerufene Programm.

Die *Execute* Prozedur kann zur Ausführung jeder .COM Datei verwendet werden, d.h. jeder Datei, die ausführbaren Object-Code enthält. Dies könnte eine mit der Com Option durch TURBO Pascal erzeugte Datei sein (siehe Seite 260). Die Datei wird an der Adresse $100 geladen und ausgeführt, entsprechend dem CP/M Standard.

Wenn die Diskettendatei nicht existiert, tritt ein I/O Fehler auf. Dieser Fehler wird behandelt wie auf Seite 116 beschrieben. Wenn der I Compilerbefehl passiv ist ({$I-}), wird die Programmausführung mit der Anweisung fortgesetzt, die der mißlungenen *Chain* oder *Execute* Anweisung folgt, und die *IOresult* Funktion muß vor weiteren I/O-Operationen aufgerufen werden.

Daten können vom laufenden Programm zum chained (verketteten) Programm entweder durch gemeinsame globale (*shared global variables*) oder durch absolute Variablen (*absolute address variables*) übertragen werden.

Um sich vor Überlappungen zu schützen, sollten gemeinsame globale Variablen am Anfang beider Programmen deklariert werden, wobei die Reihenfolge in beiden Deklarierungen die gleiche sein muß. Weiterhin müssen beide Programme auf den gleichen Speicherplatz compiliert werden (siehe Seite 261). Wenn diese Bedingungen erfüllt sind, werden die Variablen durch beide Programme an dieselbe Adresse im Speicher plaziert und können gemeinsam genutzt werden, da TURBO Pascal seine Variablen nicht automatisch initialisiert.

Beispiel:
Program *MAIN.COM:*

```
program Main;
var
  Txt:      String[80];
  CntPrg:   file;
begin
  Write('Enter any text: '); Readln(Txt);
  Assign(CntPrg, 'ChrCount.chn');
  Chain(CntPrg);
end.
```

Program *CHRCOUNT.CHN:*

```
programChrCount;
var
  Txt:      String[80];
  NoOfChar,
  NoOfUpc,
  I:        Integer;
begin
  NoOfUpc := 0;
  NoOfChar := Length(Txt);
  for I := 1 to length(Txt) do
    if Txt[I] in['A'..'Z'] then      NoOfUpc:=Succ(NoOfUpc);
  Write('No of characters in entry: ',NoOfChar);
  Writeln('.No of upper case characters: ',NoOfUpc,'.');
end.
```

Wenn Sie wollen, daß ein TURBO Programm feststellt, ob es durch eXecute oder direkt von einer DOS Kommandozeile aufgerufen wurde, sollten Sie eine **absolute** Variable an der Adresse $80 verwenden. Dort befindet sich das Byte, das die Länge der CP/M Kommandozeile enthält. Wenn ein Programm von CP/M aufgerufen wurde, enhält dieses Bit einen Wert zwischen 0 und 127, bei Aufruf durch *eXecute* setzt das aufgerufene Programm die Variable auf einen Wert höher als 127. Wenn Sie die Variable in dem aufgerufenen Programm prüfen, bedeutet ein Wert zwischen 0 und 127, daß das Programm von CP/M, ein höherer Wert, daß es von einem anderen Turbo Program aufgerufen wurde.

Beachten Sie, daß weder *Chain* noch *Execute* im Memory-Modus verwendet werden können (siehe Seite 260).

22.5 Overlays

Während der Ausführung erwartet das System, daß sich Overlay-Dateien auf dem angemeldeten Laufwerk befinden. Die Prozedur *OvrDrive* kann zur Veränderung dieses Werts verwendet werden.

22.5.1 Prozedur OvrDrive

Syntax: OvrDrive(*Laufwerk*);

wobei *Laufwerk* ein *integer* Ausdruck ist, der das Laufwerk bezeichnet (0 = angemeldetes Laufwerk, 1 = A:, 2 = B:, usw.). Bei nachfolgenden Aufrufen von Overlay-Dateien werden die Dateien auf dem angegebenen Laufwerk erwartet. Wenn eine Overlay-Datei auf einem Laufwerk geöffnet wurde, werden weitere Aufrufe derselben Datei auf demselben Laufwerk gesucht.

Beispiel:

```
program OvrTest;

overlay procedure ProcA;
begin
Writeln('Overlay A');
end;

overlay procedure ProcB;
begin
Writeln('Overlay B');
end;
```

```
procedure Dummy;
begin
  { Dummy-Prozedur, um die Overlays in zwei Gruppen zu teilen }
end;

overlay procedure ProcC;
begin
Writeln('Overlay C');
end;

begin
OvrDrive(2);
ProcA;
OvrDrive(0);
ProcC;
OvrDrive(2);
ProcB;
end
.
```

Der erste Aufruf von *OvrDrive* gibt an, daß sich die Overlays auf Laufwerk B: befinden. Der Aufruf von *ProcA* bewirkt deshalb, daß die erste Overlay-Datei (die die Overlay-Prozeduren *ProcA* und *ProcB* enthält) hier geöffnet wird.

Als nächstes gibt die Anweiung *OvrDrive(0) an, daß sich die folgenden Overlays auf dem angemeldeten Laufwerk befinden. Der Aufruf von ProcC* öffnet hier die zweite Overlay-Datei.

Die folgende Anweisung *ProcB* ruft eine Overlay-Prozedur in der ersten Overlay-Datei auf; um sicherzustellen, daß diese auf Laufwerk B: gefunden wird, muß vor diesem Aufruf die Anweisung *OvrDrive(2) ausgeführt werden.*

22.6Dateien

22.6.1 Dateinamen

In CP/M besteht ein Dateiname aus 1 bis 8 Buchstaben oder Zahlen, wahlweise gefolgt von einem Punkt und einer 1 bis 3 Buchstaben (oder Zahlen) langen Dateitypenbezeichnung:

Laufwerk:Name.Typ

22.6.2 Textdateien

Die Prozeduren *Seek* und *Flush* und die Funktionen *FilePos* und FileSize sind auf CP/M Textdateien nicht anwendbar.

22.7 Absolute Variablen

Variablen können deklariert werden, so daß sie an bestimmten Speicheradressen stehen. Sie heißen dann **absolute** Variablen. Dies geschieht, indem man bei der Variablendeklarierung das reservierte Wort **absolute** hinzufügt und eine Adresse als ganzzahlige Konstante angibt.

Beispiel:

```
Var
  IOByte: Byte absolute $0003;
  CmdLine: string[127] absolute $80;
```

Absolute kann auch benutzt werden, um eine Variable an die Spitze einer anderen Variablen zu deklarieren, d.h. daß die Variable an derselben Adresse wie die andere Variable starten soll. Wenn **absolute** vor dem Variablen- (oder Parameter-)Bezeichner steht, startet die neue Variable an der Adresse dieser Variablen (oder dieses Parameters).

Beispiel:

```
var
  Str: string[32];
  StrLen: Byte absolute Str;
```

Die obige Deklarierung gibt an, daß die Variable *StrLen* an derselben Adresse starten soll wie die Variable *Str*, und da das erste Byte einer *String* Variablen die Länge des Strings festlegt, enthält *StrLen* die Länge von *Str*. Beachten Sie, daß nur ein Bezeichner in einer **absolute** Deklarierung angegeben werden kann, d.h. das Konstrukt

```
Ident1, Ident2: Integer absolute $8000
```

ist **unzulässig**. Weitere Details über die Platzzuweisung für Variablen finden Sie in auf den Seiten 278 und 288.

22.8 Addr-Funktion

Syntax: Addr(*name*)

Gibt die Speicheradresse des ersten Bytes des Typen, der Variablen, der Prozedur oder Funktion mit dem Bezeichner *name* aus. Wenn *name* ein Feld (Array) ist, kann er vorgemerkt werden, wenn *name* ein Record ist, können bestimmte Felder ausgewählt werden. Der ausgegebene Wert ist integer.

22.9 Vordefinierte Arrays

TURBO Pascal bietet zwei vordefinierte Arrays vom Typ *Byte*, *Mem* und *Port*, die als direkter Zugang zum CPU-Speicher und zu den Daten-Ports benutzt werden können.

22.9.1 Mem Array

Das vordefinierte Array *Mem* wird benutzt, um auf Speicher zuzugreifen. Jede Komponente des Arrays ist ein *Byte*.Die Indizes entsprechen den Adressen im Speicher. Der Index ist integer. Wenn ein Wert einer Komponente von *Mem* zugewiesen wird, wird er an der durch den Index-Ausdruck gegebenen Adresse gespeichert. Wenn das Feld *Mem* in einem Ausdruck benutzt wird, wird das Byte der im Index angegebenen Adresse verwendet.

Beispiel:

```
Mem[WsCursor] := 2;
Mem[WsCursor+1] := $1B;
Mem[WsCursor+2] := Ord(' ');
IObyte := Mem[3];
Mem[Addr+Offset] := Mem[Addr];
```

22.9.2 Port-Array

Das *Port*-Array wird benutzt, um die Daten-Ports der Z-80 CPU anzusprechen. Jedes Element des Arrays repräsentiert einen Daten-Port, deren Indizes den Port-Nummern entsprechen. Falls Daten-Ports durch 8-Bit Adressen angewählt werden, ist der Index vom Typ *Byte*. Wenn ein Wert einer Komponente von *Port* zugewiesen ist, wird er an den spezifizierten Port ausgegeben. Wenn auf eine Komponente von *Port* in einem Ausdruck Bezug genommen wird, wird ihr Wert von dem spezifizierten Port eingegeben.

Der Gebrauch des Port-Arrays ist beschränkt auf Zuweisung und Bezugnahme in Ausdrücken, d.h. Komponenten von *Port* können nicht als Variablenparameter für Prozeduren und Funktionen dienen. Weiterhin sind Operationen, die auf das ganze *Port*-Array Bezug nehmen, nicht erlaubt (Bezugnahme ohne Index).

22.10 Array-Subscript Optimierung

Der **X** Compilerbefehl erlaubt es dem Programmierer zu wählen, ob die Array-Subskription eher hinsichtlich der Ausführungszeit oder der Codegröße optimiert wird. Der voreingestellte Modus ist aktiv, d.h. {$X+}, was Optimierung der Ausführungszeit zur Folge hat. Wenn der passive Modus gewählt wird, d.h. {$X-}, wird die Codegröße minimiert.

22.11 With-Anweisungen

Die voreingestellte Tiefe der Schachtelung von *With* Anweisungen ist 2. Der **W** Befehl kann verwendet werden, diesen Wert zwischen 1 und 9 zu verändern. Für jeden Block benötigen *With*-Anweisungen zwei Bytes Speicher pro Schachtelungsniveau. Der möglichst sparsame Gebrauch der Schachtelung beeinflußt stark die Größe des Datenbereichs in Programmen mit vielen Unterprogrammen.

22.12 Hinweise zu Zeigern

22.12.1 MemAvail

Die Standardfunktion *MemAvail* kann immer benutzt werden, um den Platz auf dem Heap zu ermitteln. Das Ergebnis ist eine ganze Zahl. Falls mehr als 32767 Bytes zur Verfügung stehen, gibt *MemAvail* eine negative Zahl aus. Die korrekte Zahl der freien Bytes wird dann berechnet, indem man zu 65536.0 *MemAvail* addiert. Achten Sie auf die Verwendung reeller Konstanten, um ein reelles Ergebnis zu erzeugen, falls das Ergebnis größer als *GMaxInt* ist. Speicher-Management wird ausführlicher auf Seite 288 beschrieben.

22.13 Zeiger und ganze Zahlen

Die Standardfunktionen *Ord* und *Ptr* erlauben direkte Kontrolle über die in einem Zeiger enthaltene Adresse. *Ord* gibt die in einem Zeigerargument enthaltene Adresse als ganze Zahl aus. *Ptr* wandelt sein ganzzahliges Argument in einen Zeiger um, der mit allen Zeigertypen kompatibel ist.

22.2 CP/M Funktionsaufrufe

Um CP/M BDOS und BIOS Routinen aufzurufen, hat TURBO Pascal die zwei Standardprozeduren *Bdos*, *Bios* und die vier Standardfunktionen *Bdos*, *BdosHL*, *Bios* und *BiosHL*.

Details über diese Routinen finden Sie im *CP/M Benutzerhandbuch* von Digital Research.

2.13.1 Prozedur und Funktion Bdos

Syntax: Bdos(*Func* |,*Param*|)

Die Prozedur *Bdos* wird verwendet, um CP/M BDOS Routinen aufzurufen. *Func* und *Param* sind *integer* Ausdrücke. *Func* bezeichnet die Nummer der aufgerufenen Routine und wird ins C-Register geladen. *Param* ist optional und bezeichnet einen Parameter, der in das Registerpaar DE geladen wird. BDOS wird auf Adresse 5 aufgerufen.

Die Funktion *Bdos* wird wie die Prozedur aufgerufen und gibt einen *integer* Wert aus, der dem Wert entspricht, der durch das BDOS in das A-Register ausgegeben wird.

2.13.2 Die Funktion BdosHL

Syntax: BdosHL(*Func* |,*Param*|)

Diese Funktion entspricht der Funktion *Bdos*, die gerade besprochen wurde, jedoch wird der Wert über das Registerpaar HL ausgegeben.

2.13.3 Prozedur und Funktion Bios

Syntax: Bios(*Func* |,*Param*|)

Die **Prozedur** *Bios* wird benutzt, um die BIOS Routinen aufzurufen. *Func* und *Param* sind *integer* Ausdrücke. *Func* bezeichnet die Nummer der aufgerufenen Routine, 0 steht für die Routine WBOOT, 1 für CONST usw. Die Adresse der aufgerufenen Routine ist *Func**3 plus die WBOOT-Adresse , die in den Adressen 1 und 2 enthalten ist. *Param* ist optional und bezeichnet einen Parameter, der vor dem Aufruf in das Registerpaar BC geladen wird.

Die **Funktion** *Bios* wird wie die Prozedur aufgerufen und gibt einen *integer* Wert aus, der dem Wert entspricht, der vom BIOS in das A-Register ausgegeben wird.

2.13.4 Die Funktion BiosHL

Syntax: BiosHL(*Func* |,*Param*|)

Diese Funktion entspricht der Funktion *Bios*, jedoch wird das Ergebnis über das Registerpaar HL ausgegeben.

22.14 Benutzergeschriebene I/O Treiber

Bei einigen Anwendungen ist es für den Programmierer praktisch, seine eigenen I/O Treiber zu definieren, um mit externen Geräten kommunizieren zu können. Die folgenden Treiber sind Teil von TURBO und werden durch die Standard I/O Treiber benutzt (obwohl sie nicht als Standardprozeduren oder -funktionen zur Verfügung stehen):

```
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 Routine *ConSt* wird durch die Funktion *KeyPressed* aufgerufen. Die *ConIn* und *ConOut* Routinen können von den CON:, TRM: und KBD: Geräten benutzt werden. Die LstOut-Routine wird von dem LST: Gerät benutzt. Die *AuxOut* und *AuxIn* Routinen werden von dem AUX: Gerät, *UsrIN* und *UsrOut* von dem USR: Gerät benutzt.

Laut Voreinstellung benutzen diese Treiber die entsprechenden BIOS Eingangspunkte des CP/M Betriebssystems, d.h. *ConSt* benutzt CONST, *ConIn* benutzt CONIN, *ConOut* benutzt CONOUT, *LstOut* benutzt LIST, *AuxOut* benutzt PUNCH, *AuxIn* benutzt READER, *UsrOut* benutzt CONOUT und *UsrIn* benutzt CONIN. Dies kann jedoch vom Programmierer verändert werden, indem er eine der folgenden Standardvariablen der Adresse in einer selbst definierten Treiberprozedur oder Treiberfunktion zuweist:

Variable	**enthält die Adresse der**
ConStrPtr	*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 Benutzer definierte Treiberprozedur oder Treiberfunktion muß den oben gegebenen Definitionen entsprechen, d.h. ein *ConSt* Treiber muß eine *Bool'sche* Funktion sein, ein *ConIN* Treiber muß eine *Char* Funktion sein, usw..

22.15 Externe Unterprogramme

Das reservierte Wort **external** wird benutzt, um externe Prozeduren und Funktionen zu deklarieren, typischerweise in Assembler geschriebene Prozeduren und Funktionen.

Ein externes Unterprogramm hat keinen Block, d.h. keinen Deklarierungs- und keinen Anweisungsteil. Es wird nur der Unterprogrammkopf spezifiziert, unmittelbar gefolgt von dem reservierten Wort **external** und einer ganzzahligen Konstanten, die die Speicheradresse des Unterprogramms definiert:

procedure DiskReset; **external** $EC00;
function IOstatus: boolean; **external** $D123

Parameter können an externe Unterprogramme übergeben werden. Die Syntax ist genau dieselbe, wie bei normalen Prozedur- und Funktionsaufrufen:

procedure Plot(X,Y: Integer); **external** $F003;
procedure QuickSort(**var** List: PartNo); **external** $1C00;

Die Übergabe von Parametern an externe Unterprogramme wird auf Seite 283 genauer beschrieben.

22.16 In-line Maschinencode (Assembler)

TURBO Pascal enthält **inline** Anweisungen, die einen sehr bequemen Weg bieten, Maschinencode (Assembler) direkt in den Programmtext einzufügen. Eine *inline*-Anweisung besteht aus dem reservierten Wort **inline**, gefolgt von einer oder mehreren Konstanten, Variablenbezeichnern oder Kommandozählerreferenzen, die durch Schrägstriche getrennt und in Klammern eingeschlossen sind.

Ein Codeelement ist aus einem oder mehreren Datenelementen aufgebaut, die durch (+) oder (-) Zeichen getrennt sind. Ein Datenelement ist entweder eine *integer* Konstante, ein Variablenbezeichner, ein Prozedurbezeichner, ein Funktionsbezeichner oder ein Kommandozählerwert. Ein Kommandozählerwert wird als (*) Stern geschrieben.

Beispiel:
inline (10/$2345/count + 1/sort - * + 2);

Jedes Codeelement erzeugt ein Byte oder ein Wort (zwei Byte) Code. Der Wert eines Bytes oder Wortes wird durch Addition oder Subtraktion der Werte des Datenelements, je nach Trennungszeichen, errechnet. Der Wert eines Variablenbezeichners ist die Adresse (oder das Offset) der Variable. Der Wert eines Prozedur- oder Funktionsbezeichners ist die Adresse (oder das Offset) der Prozedur oder Funktion. Der Wert des Kommandozählers ist die Adresse (oder das Offset) des Kommandozählers, d.h. die Adresse, an der das nächste Byte Code erzeugt wird.

Ein Codeelement erzeugt ein Byte Code, wenn es nur aus einer *integer* Konstanten besteht und wenn der Wert innerhalb des 8-Bitbereichs (0..255) liegt. Wenn der Wert außerhalb des 8-Bitbereichs liegt, oder das Codeelement sich auf einen Variablen-, Prozedur- oder Funktionsbezeichner bezieht oder das Codeelement einen Kommandozählerwert enthält, wird ein Wort Code (das niedrigste Byte steht zuerst) erzeugt.

Die Zeichen < und > können verwendet werden, um die oben beschriebene automatische Größenwahl zu überschreiben. Wenn das Codeelement mit dem Zeichen < beginnt, wird nur das wenigst signifikante Byte des Werts codiert, auch wenn es sich um eine 16-Bit Wert handelt. Wenn das Codeelement mit dem Zeichen > beginnt, wird immer ein Wort codiert, auch wenn das niedrigste Byte 0 ist.

Beispiel:
inline (<$1234/>$44);

Diese **inline** Anweisung erzeugt drei Bytes Code: $34, $44 und $00.

Das folgende Beispiel einer Inline-Anweisung generiert den Maschinencode, der alle Zeichen in ihrem Stringargument in Großbuchstaben umwandelt.

```
procedure UpperCase(var Strg: Str);
{Str is type String[255]}
{$A+}
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/                      { Cs10P  '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);{10                 { L2:   EQU   $ }

end;
```

Inline-Anweisungen können innerhalb des Anweisungsteils eines Blocks jederzeit mit anderen Anweisungen gemischt werden, und sie können alle Register der CPU benutzen. **Beachten Sie**, daß der Inhalt eines Stackzeiger-Registers (SP) am Ein- und Ausgang einer Inline-Routine der gleiche sein muß.

22.17 Interrupt-Handhabung

Das TURBO Pascal Laufzeit-Paket und der vom Compiler erzeugte Code sind beide voll interruptfähig. Interruptroutinen müssen alle benutzten Register zurückgeben.

Falls benötigt, können Interruptroutinen in Pascal geschrieben werden. Solche Prozeduren sollten immer mit dem **A** Compilerbefehl (|$A+|) compiliert werden. Sie dürfen keine Parameter enthalten und müssen selbst sicherstellen, daß alle benutzten Register aufbewahrt werden. Dies geschieht, indem eine **inline** Anweisung mit den notwendigen PUSH-Instruktionen ganz zu Beginnn der Prozedur plaziert wird und eine weitere **inline** Anweisung mit den entsprechenden POP-Instruktionen ganz am Ende. Die letzte Instruktion der **inline** Anweisung sollte eine EI Instruktion ($FB) sein, um weitere Interrupts zu ermöglichen. Wenn daisy chained (verkettete) Interrupts benutzt werden, kann die **inline** Anweisung auch eine RETI Instruktion ($ED,$4D) spezifizieren, die die durch den Compiler erzeugte RET Instruktion überschreibt.

Die generellen Regeln für den Registergebrauch sind, daß ganzzahlige Operationen nur die AF, BC, DE und HL Register verwenden können; andere Operationen können IX und IY und reelle Operationen wechselnde Register verwenden.

Eine Interruptprozedur sollte keine I/O Operation mit Standardprozeduren oder -funktionen von TURBO Pascal benutzen, da diese Routinen nicht rückspringend sind. Dies gilt auch für BDOS-Aufrufe (und gelegentlich für BIOS-Aufrufe, abhängig von der spezifischen CP/M Implementation).

Der Programmierer kann mit den Instruktionen DI oder EI einer **inline**-Anweisung jederzeit im Programm Interrupts außer Kraft setzen oder erlauben.

Wenn Modus 0 (IM 0) oder Modus 1 (IM 1) Interrupts benutzt werden, liegt es in der Verantwortlichkeit des Programmierers, die Restart-Stellen in der Basis-Seite zu initialisieren (Beachten Sie, daß RST 0 nicht verwendet werden kann, da CP/M die Stellen 0 bis 7 benutzt).

Wenn Modus 2 (IM 2) Interrupts verwendet werden, sollte der Programmierer eine initialisierte Sprungtabelle (ein Array von integer Zahlen) an einer absoluten Adresse erzeugen und das Register I durch eine **inline** Anweisung am Anfang des Programms initialisieren.

22.18 Interne Datenformate

In der folgenden Beschreibung bedeutet das Symbol @v die Adresse des ersten Bytes, das von einer Variablen gegebenen Typs belegt ist. Die Standardfunktion *Addr* kann benutzt werden, um diesen Wert für jede Variable zu erhalten.

22.18.1 Standard-Datentypen

Die grundlegenden Datentypen können in Strukturen gruppiert werden (Arrays, Records und Diskettendateien); diese Strukturierung beeinflußt nicht ihr internes Format.

22.18.1.1 Skalare

Die folgenden Skalare werden alle in einem einzigen Byte gespeichert: *Integere* Teilbereiche mit beiden Grenzen im Bereich 0..255, *Boolean*, *Char* und deklarierte Skalare mit weniger als 256 möglichen Werten. Dieses Byte enthält den ordinalen Wert der Variablen.

Die folgenden Skalare sind alle in zwei Bytes gespeichert: *Integer* Zahlen, *integer* Teilbereiche mit einer oder zwei Grenzen außerhalb von 0..255 und deklarierte Skalare mit mehr als 256 möglichen Werten. Diese Bytes enthalten dem 16-Bit-Wert als Zweier-Komplement, wobei das niederwertige Byte zuerst gespeichert wird.

22.18.1.2 Reelle Zahlen

Reelle Zahlen belegen 6 Bytes; bei einem Gleitkommawert sind davon 40 Bit für die Mantisse und 8 Bit für den Exponenten gegeben. Der Exponent wird im ersten Byte gespeichert und die Mantisse in den nächsten fünf Bytes, wobei das niederwertige Byte als erstes gespeichert wird:

@v	Exponent
@v+1	LSB der Mantisse
.	
.	
@v+5	MSB der Mantisse

Der Exponent steht in binärem Format mit einem Offset von $80. Daher zeigt ein Exponent von $84 an, daß der Wert der Mantisse mit 2^($84-$80) = 2^4 = 16 zu potenzieren ist. Wenn der Exponent Null ist, wird der Gleitkommawert übernommen.

Den Wert der Mantisse erhält man, indem man die 40-bit große, vorzeichenlose, *integer* Zahl durch 2^40 teilt. Die Mantisse ist immer normalisiert, d.h. das signifikanteste Bit (Bit 7 des fünften Byte) soll als 1 interpretiert werden. Das Vorzeichen einer Mantisse ist in diesem Bit gespeichert. Eine 1 bedeutet, daß die Zahl negativ, eine 0, daß sie positiv ist.

22.18.1.3 Strings

Ein String belegt pro Zeichen ein Byte. Hinzu kommt ein Byte, das die aktuelle Länge des Strings enthält und als erstes steht. Die folgenden Bytes enthalten die aktuellen Zeichen, wobei das erste Zeichen an der niedrigsten Adresse gespeichert wird. In der folgenden Tabelle bezeichnet L die aktuelle Länge des Strings und *Max* bedeutet die maximale Länge.

(a)	aktuelle Länge (*L*)
(a)+1	erster Buchstabe
(a)+2	zweiter Buchstabe
.	
.	
(a)+L	letzter Buchstabe
(a)+L+1	ungenutzt
.	
.	
(a)+Max	ungenutzt

22.18.1.4 Mengen (sets)

Ein Element einer Menge besetzt ein Bit. Die maximale Zahl von Elementen in einer Menge ist 256. Eine Mengenvariable belegt nie mehr als 32 Bytes (256/8).

Wenn eine Menge weniger als 256 Elemente enthält, sind einige Bits immer Null und müssen nicht gespeichert werden. Im Interesse der Speichereffizienz wäre die beste Art, eine Mengenvariable eines gegeben Typs zu speichern, alle nicht signifikanten Bits wegzuschneiden und die restlichen Bits so umzubauen, daß das erste Element das erste Bit des ersten Bytes belegt. Solche Rotationsoperationen sind jedoch sehr langsam, weshalb TURBO einen Kompromiß eingeht: Nur Bytes, die statisch Null sind (d.h. Bytes, deren Bits nicht benutzt werden) werden nicht gespeichert. Diese Methode der Kompression ist sehr schnell und in den meisten Fällen so speichereffizient wie die Rotationsmethode.

Die Zahl der durch eine Mengenvariable besetzten Bytes wird durch (*Max* **div** 8) - (*Min* **div** 8) + 1 berechnet, wobei *Max* und *Min* die obere und untere Grenze des Grundtyps dieser Menge sind. Die Speicheradresse eines spezifischen Elements *E* ist:

MemAddress = @ + (*E* **div**8) - (*Min* **div** 8)

und die Bitadresse innerhalb des Bytes in MemAddress ist:

BitAddress = *E* **mod** 8

wobei *E* den ordinalen Wert des Elements bezeichnet.

22.18.1.5 File Interface Blocks (FIB)

Die folgende Tabelle enthält das Format eines FIB (Datei-Schnittstellen-Blocks) in TURBO Pascal-80:

@+0	Flag-Byte
@+1	Zeichenpuffer
@+2	Sektorpuffer-Zeiger (LSB)
@+3	Sektorpuffer-Zeiger (MSB)
@+4	Zahl der Records (LSB)
@+5	Zahl der Records (MSB)
@+6	Recordlänge (LSB)
@+7	Recordlänge (MSB)
@+8	Aktueller Record (LSB)
@+9	Aktueller Record (MSB)
@+10	Ungenutzt
@+11	Ungenutzt
@+12	Erstes Byte vom CP/M FCB
.	
.	
@+47	Letztes Byte vom CP/M FCB
@+48	Erstes Byte des Sektorpuffers
.	
.	
@+175	Letztes Byte des Sektorpuffers

Das Format des Flag-Bytes bei @+0 ist:

Bit 0..3	Dateityp
Bit 4	Lese-Signal
Bit 5	Schreibsignal
Bit 6	Ausgabe-Flag
Bit 7	Eingabe-Flag

Dateityp 0 bezeichnet eine Diskettendatei, 1 bis 5 steht für die logischen I/O Geräte von TURBO Pascal (CON:, KBD:, LST:, AUX und USR:). Bei typisierten Dateien ist das Bit 4 gesetzt, falls der Inhalt des Sektorpuffers undefiniert ist. Bit 5 ist gesetzt, falls Daten in den Sektorpuffer geschrieben wurden. Bei Textdateien ist Bit 5 gesetzt, falls der Zeichenpuffer ein *pre-read* Zeichen enthält. Bit 6 ist gesetzt, falls Ausgabe erlaubt ist, Bit 7, falls Eingabe erlaubt ist.

Der Sektorenpufferzeiger speichert einen Offset (0..127) im Sektorpuffer bei $a+48$. Bei typisierten und nicht-typisierten Dateien enthalten die drei Worte von $a+4$ bis $a+9$ die Zahl der Records in der Datei, die Recordlänge in Bytes und die aktuelle Recordnummer. Der FIB einer nicht-typisierten Datei hat keinen Sektorpuffer, deshalb wird auch der Sektorpufferzeiger nicht verwendet.

Wenn eine Textdatei einem logischen Gerät zugewiesen ist, werden nur die Flag-Bytes und der Zeichenpuffer verwendet.

22.18.1.6 Zeiger

Ein Zeiger besteht aus zwei Bytes, die eine 16-Bit Speicheradresse enthalten. Er wird in reversem Byteformat abgespeichert, d.h. das am wenigsten signifikante Byte wird zuerst gespeichert. Der Wert **nil** entspricht dem Wort Null.

22.18.2 Datenstrukturen

Datenstrukturen werden aus den Datengrundtypen unter Verwendung verschiedener Strukturierungsmethoden gebildet. Es existieren drei verschiedene Strukturierungsmethoden: Array, Record und Diskettendatei. Die Strukturierung von Daten beeinträchtigt in keinem Fall das Format der Datengrundtypen.

22.18.2.1 Arrays

Die Komponenten mit den niedrigsten Indexwerten werden an der niedrigsten Speicheradresse gespeichert. Ein multidimensionales Array wird so gespeichert, daß die rechteste Dimension zuerst zunimmt. Gegeben sei z.B. das Feld:

```
Board: array[1..8,1..8] of Square
```

Für seine Elemente ergibt sich folgende Speicheranordnung:

Board[*1,1*] niedrigste Adresse
Board[*1,2*]
:
:
Board[*1,8*]
Board[*2,1*]
Board[*2,2*]
: .
Board[*8,8*] höchste Adresse

22.18.2.2 Records

Das erste Feld (Eintrag) eines Records wird an der niedrigsten Speicheradresse gespeichert. Wenn der Record keine veränderlichen Teile enthält, ist die Länge durch die Summe der Längen der einzelnen Felder gegeben. Wenn der Record veränderliche Teile enthält, ist die Gesamtzahl der belegten Bytes, durch die Länge des festen Teils plus der Länge des längsten veränderlichen Teils gegeben. Alle Varianten eines veränderlichen Teils haben im Speicher die gleiche Anfangsadresse.

22.18.2.3 Diskettendateien

Diskettendateien unterscheiden sich von anderen Datenstrukturen dadurch, daß Daten nicht im interen Speicher, sondern auf einem externen Gerät gespeichert werden. Eine Diskettendatei wird durch einen FIB, wie auf Seite 280 beschrieben, kontrolliert. Generell gibt es zwei verschiedene Typen von Diskettendateien: Dateien mit direkten Zugriff (random access) und Textdateien.

22.18..2.3.1 Dateien mit direktem Zugriff (random access)

Eine Datei mit direktem Zugriff besteht aus einer Sequenz von Sätzen, die alle die gleiche Länge und das gleiche interne Format haben. Um die Dateispeicherkapazität zu optimieren, werden alle Sätze einer Datei unmittelbar aneinander gereiht. Die ersten 4 Bytes des ersten Sektors einer Datei enthalten die Anzahl der Sätze in der Datei und die Satzlänge. Nach diesen 4 Bytes folgen die eigentlichen Dateisätze..

Sektor 0, Byte 0: Zahl der Sätze (LSB)
Sektor 0, Byte 1: Zahl der Sätze (MSB)
Sektor 0, Byte 2: Satzlänge (LSB)
Sektor 0, Byte 3: Satzlänge (MSB)

22.18.2.3.2 Textdateien

Die Grundelemente einer Textdatei sind Zeichen, aber eine Textdatei ist untergliedert in Zeilen. Jede Zeile besteht aus einer beliebigen Anzahl von Zeichen, die von einer CR/LF Sequenz beendet werden (ASCII: $OD/$OA). Eine Datei wird durch Ctrl-Z (ASCII $1B) beendet.

22.19 Parameter

Parameter werden an Prozeduren und Funktionen mittels des Z-80 Stack übergeben. Normalerweise braucht dies den Programmierer nicht zu interessieren, da der von TURBO Pascal erzeugte Maschinencode automatisch Parameter vor einem Aufruf auf einen Stack schiebt (PUSH) und sie bei Beginn eines Unterprogramms holt (POP). Falls ein Programmierer externe Unterprogramme verwenden will, müssen diese selbst die Parameter vom Stack holen (POP).

Am Eingang zu einer **external** Unterroutine enthält der oberste Teil des Stack immer die Rückkehradresse (ein Wort). Die Parameter, falls vorhanden, liegen unter dieser Rückkehradresse, d.h. auf höheren Adressen des Stack. Deshalb muß, um an die Parameter zu kommen, die Unterroutine zuerst die Rückkehradresse holen (POP), dann alle Parameter und die Rückkehradresse schließlich wieder abspeichern, indem sie sie zurück auf den Stack schiebt (PUSH).

22.19.1 Variablen-Parameter

Bei einem Variablen **VAR**-Parameter wird ein Wort auf den Stack übertragen, indem die absolute Speicheradresse des ersten besetzten Bytes des aktuellen Parameters angegeben wird.

22.19.2 Wert-Parameter

Bei einem Wertparameter hängen die auf den Stack übertragenen Daten vom Typ des Parameters ab, wie in den folgenden Abschnitten beschrieben.

22.19.2.1 Skalare

Integer, *Boolean*, *Char* und deklarierte skalare Typen werden als ein Wort auf den Stack übertragen. Wenn die Variable bei der Speicherung nur ein Byte belegt, ist das signifikanteste Byte des Parameters 0. Normalerweise wird ein Wort vom Stack durch eine Instruktion wie POP HL geholt.

22.19.2.2 Reelle Zahlen

Eine reelle Zahl wird mit 6 Bytes auf den Stack übertragen. Wenn diese Bytes mit der Instruktionssequenz

```
POP   HL
POP   DE
POP   BC
```

geholt werden, dann enthält L den Exponenten, H das fünfte (am wenigsten signifikante) Byte der Mantisse, E das vierte, D das dritte, C das zweite und B das erste (signifikanteste) Byte.

22.19.2.3 Strings

Wenn ein String an der obersten Adresse des Stack ist, enthält das Byte, auf das durch SP gezeigt wird, die Länge des Strings. Die Bytes an den Adressen SP+1 bis SP+n (wobei n die Länge des String ist) enthalten den String, wobei das erste Zeichen an der niedrigsten Adresse gespeichert wird. Die folgenden Maschinencode-Instruktionen können benutzt werden, um den String an der Spitze des Stack zu holen (POP) und in *StrBuf* zu speichern.

```
LD      DE,StrBuf
LD      HL,0
LD      B,H
ADD     HL,SP
LD      C,(HL)
INC     BC
LDIR
LD      SP,HL
```

22.19.2.4 Mengen

Eine Menge belegt immer 32 Bytes im Stack (die Mengenkompression wird nur beim Laden und Speichern von Mengen verwendet). Die folgende Maschinecode-Instruktion kann benutzt werden, um die Menge an der Spitze des Stack zu holen (POP) und in *SetBuf* zu speichern.

```
LD      DE,SetBuf
LD      HL,0
ADD     HL,SP
LD      BC,32
LDIR
LD      SP,HL
```

Damit wird das am wenigsten signifikante Byte der Menge an der niedrigsten Adresse in *SetBuf* gespeichert.

22.19.2.5 Zeiger

Ein Zeiger wird als ein Wort auf den Stack übertragen, das die Adresse der dynamischen Variable enthält. Der Wert NIL entspricht dem Wort Null.

22.19.2.6 Arrays und Records

Auch wenn sie als Wert-Parameter verwendet werden, werden Array- und Recordparameter nicht wirklich auf den Stack geschoben (PUSH). Stattdessen wird ein Wort übertragen, das die Adresse des ersten Bytes des Parameters enthält. Es liegt dann in der Verantwortlichkeit der Unterroutine, das Wort zu holen (POP) und es als die Ursprungsadresse in einer Blockkopieroperation zu benutzen.

22.20 Ergebnisse von Funktionen

Vom Benutzer geschriebene **external** Funktionen müssen ihre Ergebnisse exakt in folgender Weise spezifizieren.

Werte eines Skalartyps, außer reelle Zahlen, müssen in das HL Registerpaar ausgegeben werden. Wenn der Typ des Ergebnisses nur ein Byte braucht, dann muß er in das L Register ausgegeben werden, und H muß Null sein.

Reelle Zahlen müssen in die Registerpaare BC, DE und HL ausgegeben werden. B, C, D, E und H müssen die Mantisse (signifikantestes Byte in B) und L muß den Exponenten enthalten.

Strings und **Mengen** müssen an die oberste Adresse des Stack ausgegeben werden in Formaten, wie sie auf Seite 284 beschrieben sind.

Zeigerwerte müssen in das HL Registerpaar ausgegeben werden.

22.21 Der Heap und die Stacks

Wie in den vorigen Abschnitten durch die verschiedenen Layouts des Speichers veranschaulicht wurde, werden drei stapelartige Strukturen während der Ausführung eines Programms aufrechterhalten: Der Heap, der CPU-Stack und der Recursion-Stack.

Der Heap wird benutzt, um dynamische Variablen zu speichern und wird durch die Standardprozeduren *New*, *Mark* und *Release* kontrolliert. Am Beginn eines Programms wird der Heapzeiger *HeapPtr* auf die Adresse am unteren Ende des freien Speichers gesetzt, d.h. auf das erste freie Byte nach dem Objektcode.

Der CPU-Stack wird benutzt, um Zwischenergebnisse bei der Berechnung von Ausdrücken zu speichern und Parameter an Prozeduren und Funktionen zu übergeben. Eine aktive *for* Anweisung benutzt ebenfalls den CPU-Stack und belegt ein Wort. Am Beginn eines Programms ist der CPU-Stackzeiger *StackPtr* auf die Adresse der Spitze des freien Speicher gesetzt.

Der Recursion-Stack wird nur von rekursiven Prozeduren und Funktionen, d.h. mit dem **A** Compilerbefehl auf passiv (!$A-|) compilierten Prozeduren und Funktionen, benutzt. Am Eingang eines rekursiven Unterprogramms kopieren diese ihren Arbeitsspeicher auf den Stack, am Ausgang wird der ganze Arbeitsspeicher in seinen originalen Zustand zurückversetzt. Der voreingestellte Anfangswert von *RecurPtr* bei Beginn eines Programms ist 1K ($400) Bytes unter dem CPU-Stackzeiger.

Wegen dieser Technik dürfen zu einem Unterprogramm lokale Variablen nicht als *var* Parameter in rekursiven Aufrufen verwendet werden.

Die vordefinierten Variablen:

HeapPtr: der Heapzeiger,
RecurPtr: der Recursion-Stackzeiger und
StackPtr: der CPU-Stackzeiger

erlauben dem Programmierer die Position des Heap und Stack zu kontrollieren.

Der Typ dieser Variablen ist *integer*. Beachten Sie, daß *HeapPtr* und *RecurPtr* in dem gleichen Kontext wie andere *integer* Variablen benutzt werden können, wohingegen *StackPtr* nur in Zuweisungen und Ausdrücken verwendet werden kann.

Wenn diese Variablen manipuliert werden, müssen Sie sicherstellen, daß sie auf Adressen mit freiem Speicher zeigen und daß:

HeapPtr < *RecurPtr* < *StackPtr*

ist.

Verletzung dieser Regeln kann zu unvorhersagbaren, vielleicht verhängnisvollen Ergebnissen führen.

Selbstverständlich dürfen Zuweisungen zu Heap- oder Stackzeigern nicht erfolgen, wenn die Stacks oder der Heap einmal in Gebrauch sind.

Bei jedem Aufruf der Prozedur *New* und beim Eintragen einer rekursiven Prozedur oder Funktion, prüft das System, ob *HeapPtr*kleiner als *RecurPtr* ist. Wenn nicht, hat es einen Zusammenstoß zwischen Heap und Stack gegeben, was zu einem Ausführungsfehler führt.

Beachten Sie, daß zu keiner Zeit eine Prüfung vorgenommen wird, die sicherstellt, daß der CPU-Stack nicht in das untere Ende des Recursion-Stacks überlappt. Das könnte passieren, wenn eine rekursive Unterroutine sich selbst 300-400 mal aufruft, was eine sehr unwahrscheinliche Situation ist. Falls, warum auch immer, ein Programm eine solche Schachtelung benötigt, bewegt das folgende Statement, am Beginn eines Programmblocks ausgeführt, den Recursion-Stackzeiger abwärts, um einen größeren CPU-Stack zu erzeugen:

```
RecurPtr := StackPtr + 2 * MaxDepth -512;
```

Dabei ist *MaxDepth* die maximal benötigte Tiefe der Aufrufe für ein rekursives Unterprogramm. Die etwa 512 Bytes extra werden als Grenze gebraucht, um für Parameterübertragungen und Zwischenergebnisse während der Berechnung von Ausdrücken Platz zu schaffen.

22.21 Speicherverwaltung

22.21.1 Speicherkarten

Die folgenden Diagramme illustrieren die Inhalte des Speichers während unterschiedlicher Arbeitsstufen des TURBO Systems. Durchgezogene Linien entsprechen festen Grenzen (d.h. festgelegt durch die Menge an Speicher, Größe Ihres CP/M, Version von TURBO, usw.), gestrichelte Linien entsprechen Grenzen, die während des Laufs festgesetzt werden (z.B. durch die Größe der Source, durch dem Benutzer mögliche Manipulation verschiedener Zeiger, usw.). Die Größe der Segmente im Diagramm stimmen nicht notwendigerweise mit der Menge des tatsächlich verbrauchten Speichers überein.

22.21.1.1 Compilierung im Speicher

Während der Compilierung eines Programms im Speicher (Memory Modus des Compiler Optionsmenü, siehe Seite 288) sieht der Speicher folgendermaßen aus:

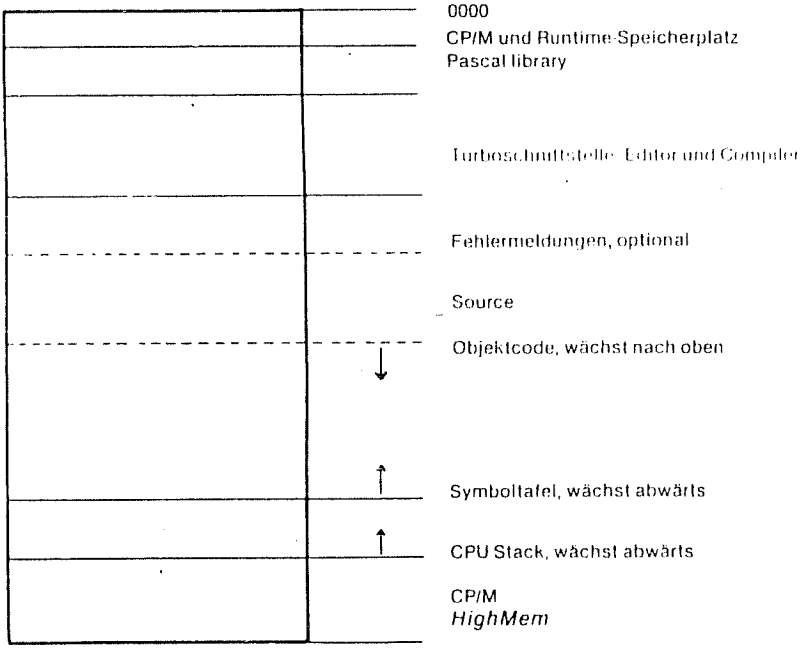

Abbildung 22-5: Speicherbelegung beim Compilieren im Speicher

Wenn die Fehlermeldungen beim Start von TURBO nicht geladen sind, beginnt die Source im Speicher um den Bereich, den die Fehlermeldungen belegt hätten, weiter unten. Wenn der Compiler aufgerufen ist, erzeugt er Objektcode, wobei er vom Ende des Quelltextes aufwärts arbeitet. Der CPU-Stack arbeitet abwärts, von der logischen Spitze des Speichers. Die Symboltafel des Compilers arbeitet abwärts ab einer Adresse, die 1K ($400 Bytes) unter der logischen Spitze des Speichers liegt.

22.21.1.2 Compilierung auf Diskette

Während der Compilierung auf eine .COM oder .CHN Datei (**C**om-Modus oder c**H**n-Modus des Compiler-**O**ptions Menü, siehe Seite 259) sieht der Speicher ziemlich genauso aus, wie bei Compilierung im Speicher, außer daß der erzeugte Objektcode nicht im Speicher verbleibt, sondern in eine Diskettendatei geschrieben wird. Der Code beginnt auch an einer niedrigeren Adresse (gleich nach der Pascal Libary, statt nach der Source). In diesem Modus ist die Compilierung von wesentlich längeren Programmen möglich.

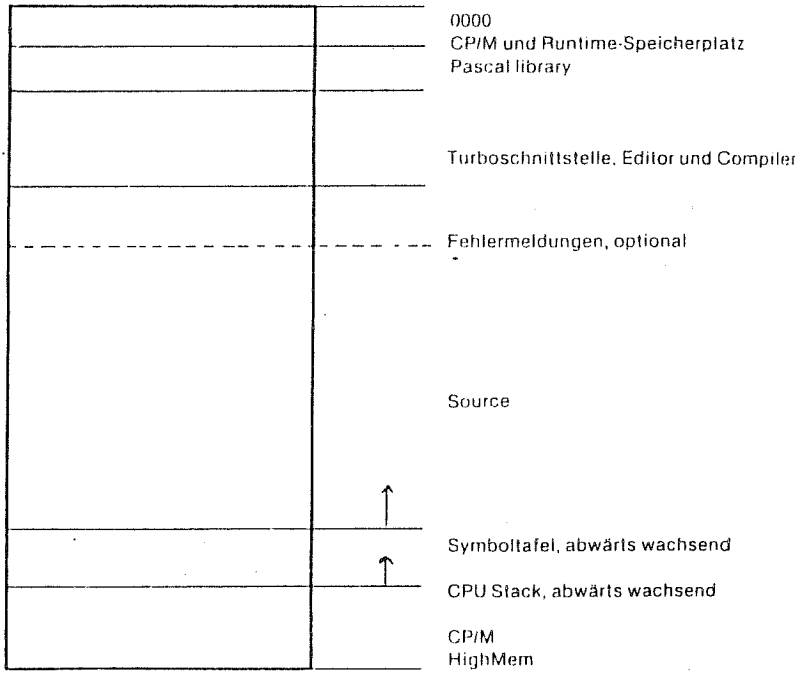

Abbildung 22 - 6: Speicherbelegung beim Compilieren auf eine Datei

22.21.1.3 Ausführung im Speicher

Wenn ein Programm im direkten bzw. im Speichermodus ausgeführt wird (d.h. der **M**emory-Modus im Compiler **O**ptionenmenü gewählt ist, siehe Seite 259) sieht eine Darstellung des Speichers folgendermaßen aus:

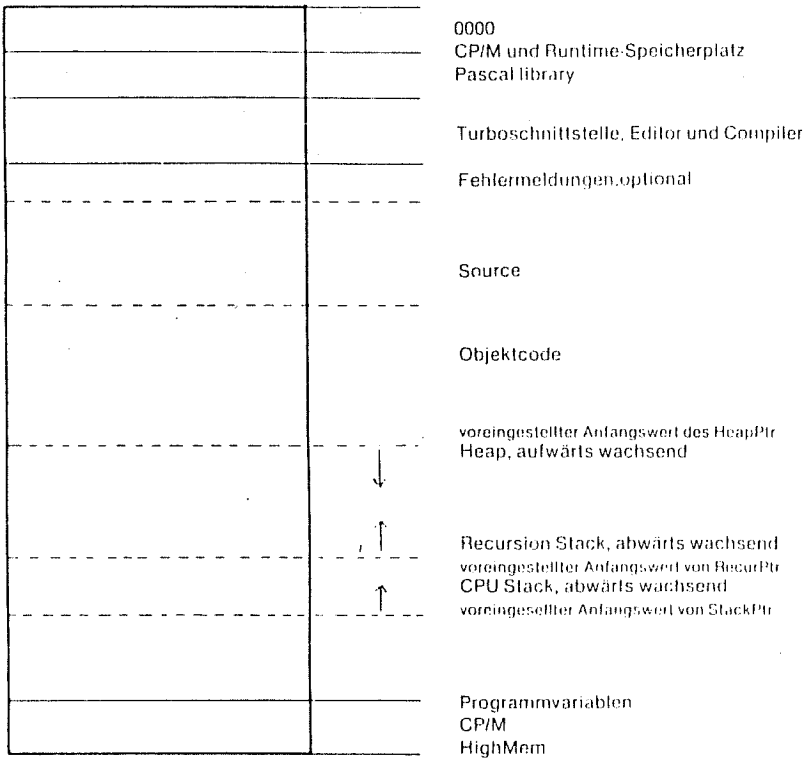

Abbildung 22 - 7: Speicherbelegung bei Ausführung im direktem Modus

Wenn ein Programm compiliert wird, ist das Ende des Objektcode bekannt. Der Heapzeiger *HeapPtr* ist als Voreinstellung auf diesen Wert gesetzt. Der Heap wächst von hier abwärts im Speicher gegen den Recursion-Stack. Die maximale Speichergröße ist BDOS minus eins (angezeigt im Compiler **O**ptions Menü). Programmadressen werden von dieser Adresse an abwärts gespeichert. Das Ende der Variablen ist die Spitze des freien Speichers, die der Anfangswert des CPU-Zeiger *StackPtr* ist. Der CPU-Stack wächst von hier abwärts gegen die Position des Recursion-Stackzeigers *RecurPtr*, $400 Bytes niedriger als *StackPtr*. Der Recursion-Stack wächst von hier abwärts gegen den Heap.

22.21.1.4 Ausführung eines Programmes

Wenn eine Programmdatei ausgeführt wird (entweder durch den **R**un Befehl des **C**omfile Modus im **O**ptionenmenü des Compilers, durch einen e**X**ecute Befehl oder direkt von CP/M aus), sieht eine Darstellung des Speichers folgendermaßen aus:

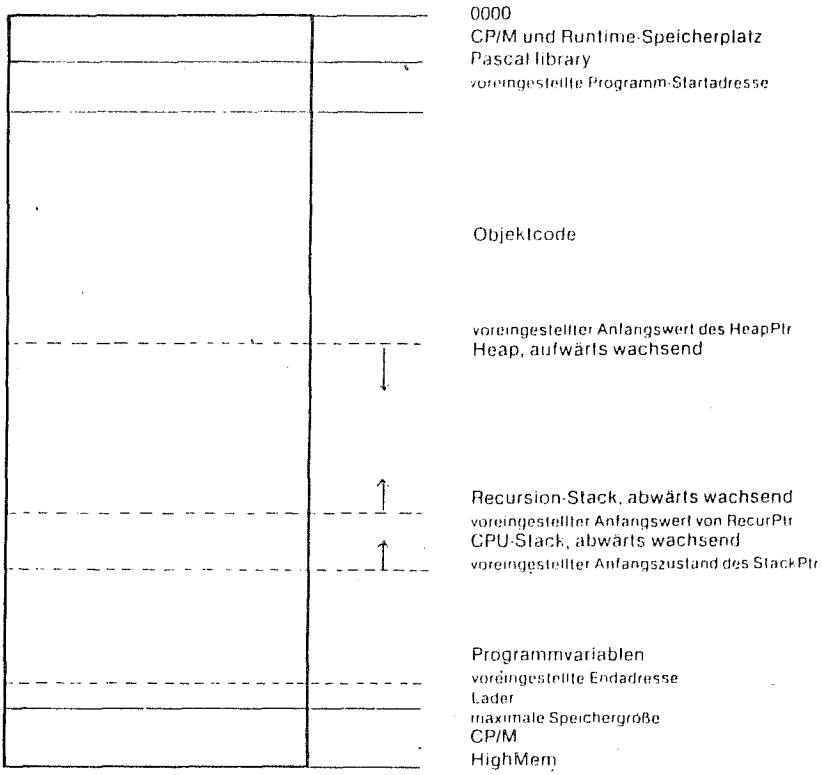

Abbildung 22 - 8: Speicherbelegung bei Ausführung eines Programmes

Diese Karte ähnelt der vorigen, bis auf das Fehlen der TURBO Schnittstelle, des Editors, des Compilers (und möglicher Fehlermeldungen) und der Source.

Die voreingestellte Programmstartadresse (angezeigt im Compiler **O**ptionenmenü) ist das erste freie Byte nach der Pascal Library. Dieser Wert kann mit dem **S**tartaddress-Befehl des Compiler **O**ptionenmenüs beeinflußt werden, z.B. um Platz für **absolute** Variablen und/oder externen Prozeduren zwischen der Library und dem Code zu schaffen. Die maximale Speichergröße ist BDOS minus eins, der voreingestellte Wert ist durch den BDOS Platz auf dem verwendeten Computer festgesetzt.

Wenn Programme für andere Systeme übersetzt werden, sollte darauf geachtet werden, eine Überlappung mit dem BDOS zu vermeiden. Der maximale Speicher kann mit dem **E**nd Address-Befehl des Compiler **O**ptionenmenüs beeinflußt werden. Beachten Sie, daß die voreingestellte Endadressenstelle etwa 700 bis 1000 Bytes unter dem maximalen Speicher liegt. Dieser freie Platz, der genau unter BDOS liegt, ist für den Lader vorgesehen, wenn mit **R**UN oder e**X**ecute aus TURBO ein Programm gestartet werden soll. Nach Programmende lädt dieser Lader den TURBO Editor, den Compiler und mögliche Fehlermeldungen wieder in den Speicher und gibt die Kontrolle an das TURBO-System zurück.

23. TURBO-BCD

TURBO-BCD ist eine spezielle Version von TURBO Pascal, die nicht zum normalen TURBO Pascal Paket gehört. Es verwendet binäre Codierung von dezimalen *real* Zahlen, um eine höhere Genauigkeit zu erreichen, die speziell bei Programmen für den kaufmännischen Bereich erforderlich ist.

Wenn Sie sich für den Kauf von TURBO-BCD interessieren, wenden Sie sich bitte an Heimsoeth Software, München.

Jedes Programm, das mit dem normalen TURBO oder TURBO-87 geschrieben wurde, kann mit TURBO-BCD compiliert und laufen gelassen werden. Der einzige Unterschied liegt in der Verarbeitung von *real* Zahlen und deren Format.

23.1 Dateien auf der TURBO-BCD Originaldiskette

Zusätzlich zu den auf Seite 8 aufgelisteten Dateien enthält die TURBO-BCD Diskette die Datei

TURBOBCD.COM

(TURBOBCD.CMD für CP/M-86). Diese Datei enthält das spezielle TURBO-BCD System. Wenn Sie es mit TINST installieren wollen, müssen Sie davor die Datei vorübergehend in TURBO.COM (oder .CMD) umbenennen.

23.2 BCD Wertebereich

Der Wertebereich von BCD *real* Zahlen des TURBO-BCD ist 1E-63 bis 1E+83 mit 18 signifikanten Stellen.

23.3 Form-Funktion

Syntax: Form(*St,Var1,Var2,...,VarN*);

Die *Form* Funktion ermöglicht bessere numerische und Stringformatierung. *St* ist ein *String* Ausdruck, der das Format des Strings angibt. Er wird im folgenden genauer beschrieben. *Var1, Var2,..., VarN sind Real, Integer* oder *String* Ausdrücke. Das Ergebnis ist ein String derselben Länge wie *St*.

St besteht aus einer Reihe von Feldangaben, wobei jede einzelne zu einem Parameter der Parameterliste gehört. Leerzeichen und andere als die im Folgenden definierten Zeichen dienen als Feldseparatoren und erscheinen ebenso im formatierten Ergebnis:

```
Form('Total: $#,###.##',1234.56) = 'Total: $1234,56'
```

Die Argumente in der Argumentenliste verwenden die Feldangaben in der Ordnung ihres Auftretens:

```
Form('Please @@@@@@ us at (###)### #### '.'phone',089/
264060) = 'Please phone us at (089) 264 060 '
```

Wenn mehr Argumente in der Argumentenliste stehen, als Feldangaben im Formatstring sind, werden die überzähligen Argumente ignoriert. Wenn es weniger Argumente als Feldangaben gibt, werden die überzähligen Feldangaben unverändert ausgegeben:

```
Form('###.##',12.34,43.21) = ' 12.34 '
Form('###.## -##.##',123.4) = '123.40 -##.## '
```

Es gibt zwei Arten von Feldangaben: **numerische** und **string** Felder.

23.3.1 Numerische Felder

Ein numerisches Feld ist eine Folge von einem oder mehreren der folgenden Zeichen:

```
# @ * $ - + , .
```

Jedes andere Zeichen beendet das numerische Feld. Die Zahl wird rechtsbündig innerhalb des Felds ausgegeben, Dezimalzahlen werden gerundet, wenn sie die für Dezimalzahlen angegebene Stellenzahl überschreiten. Wenn die Zahl zu groß ist, um in dem Feld ausgegeben zu werden, werden alle Stellen mit Sternen aufgefüllt.

#
Eine Zeichenstelle. Wenn das numerische Feld keine 0 oder * Zeichen enthält, werden nichtgebrauchte Stellen als Leerzeichen ausgegeben. Wenn das numerische Feld keine Vorzeichenstelle enthält ('-' oder '+' Zeichen) und die Zahl negativ ist, wird ein Minus vor der Zahl ausgegeben.

Beispiele:

```
Form('####',34.567)          = '  35'
Form('###.##',12.345')       = ' 12.35'
Form('####.##',-12.3)        = '-12.30'
Form('###.##',1234.5)        = '***.**'
```

0
Eine Zeichenstelle. Ungebrauchte Stellen werden zwangsweise als Nullen anstatt als Leerzeichen ausgegeben. Das Zeichen 0 muß in dem Feld nur einmal auftauchen um diesen Effekt zu bewirken. Das Vorzeichen der Zahl wird nicht ausgegeben, außer das Feld enthält eine Vorzeichenstelle ('-' oder '+' Zeichen).

Beispiele:

```
Form('0##',9)                = '009'
Form('0000.00',12.345)       = '012.35'
```

*
Eine Zeichenstelle. Ungebrauchte Stellen werden zwangsweise als Sterne anstatt als Leerzeichen ausgegeben. Das Zeichen * muß in dem numerischen Feld nur einmal vorkommen um diesen Effekt zu aktivieren. Das Vorzeichen der Zahl wird nicht ausgegeben, außer das Feld enthält eine Vorzeichenstelle ('-' oder '+' Zeichen).

Beispeile:

```
Form('*##.#8',4.567)         = '**4.57'
Form('****',123)             = '* +123'
```

$

Eine Zeichenstelle. Es wird ein Dollarzeichen '$' jeweils direkt vor der Zahl ausgegeben. Das '$' Zeichen muß nur einmal vorkommen, um diesen Effekt zu aktivieren.

Beispiele:

```
Form('$######.##',123.45)      = '   $123.45'
Form('######.#$',-12.345)      = '    -$12.35'
Form('*$####.##',12.34)        = '***12.34'
```

-

Eine Vorzeichenstelle. Wenn die Zahl negativ ist, wird ein Minus-Zeichen '-' an dieser Stelle ausgegeben; ist die Zahl positiv, wird ein Leerzeichen ausgegeben.

Beispiele:

```
Form('-###.##',-1.2)           = '- 1.20'
Form('-###.,12)                = '  12.00'
Form('*######.##-',-123.45)    = '***123.45-'
```

+

Eine Vorzeichenstelle. Wenn die Zahl positiv ist, wird ein Plus-Zeichen '+' an dieser Stelle ausgegeben; ist die Zahl negativ, wird ein Minus-Zeichen '-' ausgegeben.

Beispiele:

```
Form('+###.##',-1.2)           = '-  1.20'
Form('+###.##,12)              = '+ 12.00'
Form('*$####.###+',12.34)      = '***$12.34+'
```

,

Ein dezimales Komma oder Trennungs-Komma. Der letzte Punkt oder das letzte Komma des numerischen Bilds wird als dezimaler Begrenzer betrachtet.

.

Ein dezimaler Punkt oder Trennungs-Punkt. Der letzte Punkt oder das letzte Komma des numerischen Bilds wird als dezimaler Begrenzer betrachtet.

Beispiele:

```
Form('##,###,###.##',12345.6)  = '      12,345.60'
Form('$#.###.###.##',-12345.6) = '    -$12.345,60'
Form('*$,###,###.##+',12345.6) = '***$12,345.60+'
Form('##,###.##',123456.0)     = '**,***.**'
```

23.3.2 String-Felder

Ein String-Feld ist eine Folge von # oder @ Zeichen. Wenn der Stringparameter länger als das Stringfeld ist, werden nur die ersten Zeichen des Strings ausgegeben.

#
Wenn das Feld nur # Zeichen enthält, wird der String linksbündig ausgegeben.

@
Wenn ein oder mehrere '@' Zeichen in dem Feld vorkommen, wird der String rechtsbündig innerhalb der Länge des Felds ausgegeben.

Beispiele:

```
Form('##########','Pascal')     = 'Pascal'
Form('@@@@@@@@@@','Pascal')     = '     Pascal'
Form('####','TURBO Pascal')     = 'TURBO '
Form('@@@@@@','TURBO Pascal')   = 'TURBO '
```

23.4 Schreiben von reellen Zahlen im BCD-Format

Die *real* Zahlen des BCD-Systems werden in einem zum Standard leicht unterschiedlichen Format geschrieben, das unten beschrieben ist.

R
Die dezimale Darstellung des Wertes von *R* wird als 25 Zeichen langes Feld im Gleitkomma-Format dargestellt. Für *R* >= 0.0 ist das Format:

```
␣␣#.##########################E*##
```

Für *R* < 0.0 ist das Format:

```
␣-#.##########################E*##
```

wobei ␣ für ein Leerzeichen, # für eine Zahl und * für Plus '+' oder Minus '-' steht.

R:n
Die dezimale Darstellung des Werts *R* wird rechtsbündig in einem *n* Zeichen langen Feld im Gleitkomma-Format ausgegeben. Für *R* >= 0.0:

```
Leerzeichen#.ZahlenE*##
```

Für *R* < 0.0:

```
Leerzeichen-#.ZahlenE*##
```

wobei *Leerzeichen* null oder mehr Leerzeichen entspricht, für *Stellen* 1 bis 17 Stellen stehen können, # steht ebenfalls für eine Stelle und * entspricht Plus oder Minus.

23.4.1 Formatiertes Schreiben

Die *Form* Funktion kann als *Schreibparameter* verwendet werden, um formatierte Ausgabe zu erzeugen:

```
Write(Form('The price is $###,###,###.##',Price));
```

23.5 Internes Datenformat

Die BCD *real* Variablen belegen 10 Bytes und bestehen aus einem Gleitkomma-Wert mit einer 18 stelligen binär codierten, dezimalen Mantisse, einem 7-Bit 10'er Exponenten und einem 1-Bit Vorzeichen. Der Exponent und das Vorzeichen werden im ersten Byte und die Mantisse in den nächsten 9 Bytes gespeichert; das niedrigste Byte steht zu Anfang:

a + 0	Exponent und Vorzeichen
a + 1	wenigst signifikantes Byte der Mantisse
.	
.	
a + 9	höchst signifikantes Byte der Mantisse

Das signifikanteste Bit des ersten Byte enthält das Vorzeichen. 0 bedeutet positiv und 1 negativ. Die restlichen sieben Bit enthalten den Exponenten in binärem Format mit einem Offset von $3F. Deshalb bedeutet ein Exponent von $41, daß der Wert der Mantisse mit 10^($41-$3F) = 10^2 = 100 multipliziert werden muß. Wenn das erste Byte Null ist, wird der Gleitkomma-Wert ebenfalls als Null betrachtet. Beginnend mit dem 10-ten Byte, enthält jedes Byte der Mantisse zwei Zahlen im BCD-Format, mit jeweils der signifikantesten Zahl in den oberen vier Bit. Die erste Zahl enthält Zehntel, die zweite Hundertstel usw.. Die Mantisse ist immer normalisiert, d.h. die erste Zahl ist nie 0, außer die ganze Zahl ist 0.

Diese 10-Byte *Reals* sind kompatibel zu den *Reals* von Standard TURBO oder TURBO-87. Das ist auch nur dann problematisch, wenn Sie Programme in verschiedenen Versionen von TURBO entwickeln, die Daten austauschen müssen. Der Trick ist, einfach ein Austauschformat zwischen den Programmen zu vereinbaren, in dem Sie *Reals* beispielsweise im ASCII-Format übertragen.

Anmerkungen:

24. TURBO-87

TURBO-87 ist eine spezielle Version von TURBO Pascal, die den Intel 8087 Mathematik-Prozessor für reellzahlige Arithmetik verwendet, was erhebliche Gewinne bei der Ausführungszeit und der Präzision von Berechnungen erbringt.

Mit TURBO-87 wird jedes in Standard TURBO Pascal geschriebene Programm compiliert und ist lauffähig; der einzige Unterschied liegt in der Verarbeitung und dem Format reeller Zahlen.

Das TURBO-87 Paket enthält den Standard TURBO Pascal Compiler. Sie können also wählen, in welchem Format Sie Ihre Programm erstellen wollen. TURBO-87 Programme laufen **nicht** auf einem Computer ohne den 8087 Prozessor, während der umgekehrte Fall möglich ist.

24.1 Dateien auf der Originaldiskette

Zusätzlich zu den auf Seite 8 des Handbuchs aufgelisteten Dateien enthält die Orginaldiskette die Datei

Turbo-87.COM

(Turbo-87.CMD bei CP/M-86). Diese Datei enthält den speziellen TURBO-87 Compiler. Wenn Sie diesen mit TINST installieren müssen, müssen Sie die Datei vorübergehend in TURBO.COM (.CMD) umbenennen.

24.3 Schreiben von reellen Zahlen im 8087-Format

8087 *Reals* werden in einem vom Standard- leicht abweichenden Format geschrieben, das unten beschrieben ist.

R
Die dezimale Darstellung des Wertes von *R* wird als ein 23 Zeichen langes Feld im Gleitkomma-Format dargestellt. Für *R* >= 0.0 ist das Format:

␣␣#.################E*##

Für *R* < 0.0 ist das Format:

␣␣#.################E*##

wobei ␣ für ein Leerzeichen, # für eine Zahl und * für Plus '+' oder Minus '-' steht.

R:n
Die dezimale Darstellung des Werts *R* wird rechtsbündig in einem *n* Zeichen langen Feld im Gleitkomma-Format ausgegeben. Für *R* >

Leerzeichen#.ZahlenE*##

Für *R* < 0.0:

Leerzeichen-#.ZahlenE*##

wobei *Leerzeichen* null oder mehr Leerzeichen entspricht, für *Stellen* 1 bis 14 Stellen stehen können, # steht ebenfalls für eine Stelle und * entspricht Plus oder Minus.

24.3 Internes Datenformat

Der 8087 Prozessor unterstützt eine Reihe von Datentypen. Davon verwendet TURBO-87 den Typ *long real*. Dies ist eine reelle Zahl mit 16 Stellen Genauigkeit, der 64 Bit im Speicher zur Verfügung stehen. Sie hat einen Bereich von 4.19E-307 bis 1.67E+308.

Diese reellen Zahlen, denen im Speicher 8 Bytes zur Verfügung stehen, sind natürlich nicht mit den reellen Zahlen des Standard TURBO Pascal kompatibel, da diesen nur 6 Bytes zur Verfügung stehen. Dies macht jedoch nur Probleme, wenn Sie Programme sowohl in TURBO Pascal 3.0, als auch in Turbo-87 entwickeln, und Daten ausgetauscht werden müssen. Der Trick, der hier Abhilfe schafft, ist, ein Austauschformat zwischen beiden Programmen zu verwenden, in dem Sie reelle Zahlen, z.B. im ASCII Format, übertragen.

A. Zusammenfassung der Standardprozeduren und Standardfunktionen

Dieser Anhang listet alle in TURBO Pascal verfügbaren Standardprozeduren und -funktionen auf und beschreibt ihre Syntax, ihre Parameter und ihre Typen. Die folgenden Symbole bezeichnen Elemente verschiedenen Typs:

type beliebiger Typ
string beliebiger Stringtyp

file beliebiger Dateityp
scalar beliebiger Skalartyp
pointer beliebiger Zeigertyp

Wo keine Parametertypspezifikation vorhanden ist, bedeutet das, daß die Prozedur oder die Funktion Variablenparameter beliebigen Typs akzeptiert.

A.1 Ein-/Ausgabeprozeduren und -funktionen

Die folgenden Prozeduren benutzten in ihrer Parameterliste nicht die Standardsyntax:

procedure
Read (**var** *F:* **file of** *type;* **var** *v: type);*
Read (**var** *F:* text; **var** *I: Integer);*
Read (**var** *F:* text; **var** *R: Real);*
Read (**var** *F:* text; **var** *C: Char);*
Read (**var** *F:* text; **var** *S: string);*
Readln (**var** *F:* text);
Write (**var** *F:* **file of** *type;* **var** *v:type);*
Write (**var** *F:* text; I: Integer);
Write (**var** *F:* text; R: Real);
Write (**var** *F:* text; B: Boolean);
Write (**var** *F:* text; C: Char);
Write (**var** *F:* text; S: string);
Writeln (**var** *F:* text);

A.2 Arithmetische Funktionen

function
  Abs (*I*: Integer): Integer;
  Abs (*R*: Real): Real;
  ArcTan (*R*: Real): Real;
  Cos (*R*: Real): Real;
  Exp (*R*: Real): Real;
  Frac (*R*: Real): Real;
  Int (*R*: Real): Real;
  Ln (*R*: Real): Real;
  Sin (*R*: Real): Real;
  Sqr (*I*: Integer): Integer;
  Sqr (*R*: Real): Real;
  Sqrt (*R*: Real): Real;

A.3 Skalarfunktionen

function
  Odd (*I*: Integer): Boolean;
  Pred (*X: scalar*): scalar;
  Succ (*X: scalar*): scalar,

A.4 Transferfunktionen

function
  Chr (*I*: Integer): Char;
  Ord (*X: scalar*): Integer;
  Round (*R*: Real): Integer;
  Trunc (*R*: Real): Integer;

A.5 Stringprozeduren und -funktionen

Die *Str* Prozedur benutzt nicht-standardisierte Syntax für ihre numerischen Parameter.

procedure
Delete (**var** *S: string; Pos, Len*: Integer);
Insert (*S: string*; **var** *D: string; Pos* :Integer);
Str (*I*: Integer; **var** *S: string*);
Str (*R*: Real; **var** *S: string*);
Val (*S: string*; **var** *R*: Real;**var** *P*: Integer);
Val (*S: string*; **var***I, P*:Integer);

function
Concat (*S1, S2,..., Sn: string): string*;
Copy (*S: string; Pos, Len:* Integer): *string*;
Length (*S: string*): Integer;
Pos (*Pattern, Source: string*): Integer;

A.6 Datei-Handhabungsroutinen

procedure
Append (**var** *F: file; Name: String);*
Assign (**var** *F: file; Name: string);*
BlockRead (**var** *F:* **file**; **var** *Dest: Type; Num*: Integer);
BlockWrite (**var** *F:* **file**; **var** *Dest: Type; Num:* Integer);
Chain (**var** *F: file*);
Close (**var** *F: file*);
Erase (**var** *F: file*);
Execute (**var** *F :file*);
Rename (**var** *F: file; Name: string*);
Reset (**var** *F: file);*
Rewrite (**var** *F :file*);
Seek (**var** *F:* **file of** *type, Pos:* Integer)

function
Eof (**var** *F: file*): Boolean;
Eoln (**var** *F:* Text): Boolean;
FilePos (**var** *F:* **file of** *type*): Integer;
FilePos (**var** *F:* **file**): Integer;
FileSize (**var** *F:* **file of** *type*): Integer;
FileSize (**var** *F:* **file**): Integer;
SeekEof (**var** *F:* **file**): Boolean;
SeekEoln (**var** *F:* **file**): Boolean;

A.7 Heap Kontrollprozeduren und -funktionen

procedure
Dispose (**var** *P: pointer*);
FreeMem (**var** *P: pointer*, I: integer);
GetMem (**var** *P:pointer; I:* Integer):
Mark (**var** *P: pointer*);
New (**var** *P: pointer*);
Release (**var** *P: pointer*);

function
MaxAvail : Integer;
MemAvail : Integer;
Ord(*P: pointer*): Integer;
Ptr (*I*: Integer):f 2pointer;

A.8 Bildschirm-bezogene Prozeduren

procedure
CrtExit;
CrtInit;
ClrEol;
ClrScr;
DelLine;
GotoXY(*X, Y*: Integer);
InsLine;
LowVideo;
NormVideo;

function
WhereX : Integer; (nur IBM-PC)
WhereY : Integer; (nur IBM-PC)

A.9 Verschiedenartige Prozeduren und Funktionen

procedure

Bdos (func, param: Integer); (nur CP/M-80)
Bios (func, param: Integer); (nur CP/M-80)
ChDir (Path: String); (nur PC/MS-DOS)
GetDir (Path: String);
MkDir (Path: String);
MsDos (Func: Integer, Param: record) (nur PC/MS-DOS)
Delay (*mS*: Integer);
FillChar (**var** dest; length: Integer; data: Char);
FillChar (**var** dest; length: Integer; data: byte);
Halt;
Move (**var** source, dest; length: Integer);
Randomize;
RmDir (Drv: integer; var Path: String); (nur PC/MS-DOS)

function

Addr (**var** *Variable*): Pointer; (PC/MS-DOS)
Addr (**var** Variable): Integer; (CP/M-80)
Addr (<function identifier>): Integer; (CP/M-80)
Addr (<procedure identifier>): Integer; (CP/M-80)
Bdos (*Func, Param*: Integer): Byte;
BdosHL (*Func, Param*: Integer): Integer;
Bios (*Func, Param*: Integer): Byte;
BiosHL (*Func, Param*: Integer): Integer,
Hi (*I*: Integer): Integer;
IOresult : Boolean;
KeyPressed : Boolean;
Lo (*I*: Integer): Integer;
ParamCount: Integer;
ParamStr: (N: Integer): String;
Random (*Range*: Integer): Integer;
Random : Real;
SizeOf (**var** *variable*): Integer;
SizeOf (<type identifier>): Integer;
Swap (*I*: Integer): Integer;
UpCase (*Ch*: Char): Char;

A.10 IBM PC Prozeduren und Funktionen

Die folgenden Prozeduren und Funktionen sind nur in der IBM-PC Version implementiert.

A.10.1 Graphik, Fenster und Sound

procedure
Draw(X1, Y1, X2, Y2, *Color*);
GraphBackground(*Color*: Integer);
GraphColorMode;
GraphMode;
GraphWindow(X1, Y1, X2, Y2, :Integer);
HiRes;
HiResColor(*Color*: Integer);
NoSound;
Palette(*Color*: Integer);
Plot(*X, Y, Color*: Integer);
Sound(*I*: Integer);
TextBackground(*Color*: Integer);
TextColor(*Color*: Integer);
TextMode(*Color*: Integer);
Window(X1, Y1, X2, Y2, : Integer);

function
WhereX: Integer;
WhereY: Integer;

constant

BW40:Integer;	= 0
C40: Integer;	= 1
BW80 Integer;	= 2
C80: Integer;	= 3
Black: Integer;	= 0
Blue: Integer;	= 1
Green: Integer;	= 2
Cyan: Integer;	= 3
Red: Integer;	= 4
Magenta: Integer;	= 5
Brown: Integer;	= 6
LightGray: Integer;	= 7
DarkGray: Integer;	= 8
LightBlue: Integer;	= 9
LightGreen: Integer;	= 10

```
  LightCyan: Integer;      = 11
  LightRed: Integer;       = 12
  LightMagenta: Integer;   = 13
  Yellow: Integer;         = 14
  White: Integer;          = 15
  Blink: Integer;          = 16
```

A.10.2 Erweiterte Graphik

```
procedure
  Arc(X, Y, Winkel, Radius, Farbe: Integer);
  Circle(X, Y,Radius, Farbe: Integer);
  ColorTable(C1, C2, C3, C4: Integer);
  FillScreen(Color: Integer);
  FillShape(X, Y, FüllFarbe, RandFarbe: Integer);
  FillPattern(X1, Y1, X2, Y2, Farbe: Integer);
  GetPic(var Puffer: beliebiger Typ; X1, Y1, X2, Y2: Integer);
  Pattern(P:array[1..7] of Byte);
  PutPic(var Puffer: beliebiger Typ; X, Y: Integer);

function
  GetDot(X, Y: Integer): Integer;
```

A.10.3 Turtle-Graphik

```
procedure
  Back(Dist: Integer);
  ClearScreen;
  Forwd(Dist: Integer);
  HideTurtle;
  Home;
  NoWrap;
  PenDown;
  PenUp;
  SetHeading(Winkel: Integer);
  SetPenColor(Farbe: Integer);
  SetPosition(X, Y: Integer);
  ShowTurtle;
  TurnLeft(Winkel: Integer);
  Turnright(Winkel: Integer);
  TurtleWindow(X, Y, B, H: Integer);
  Wrap;
```

function
Heading: Integer;
Xcor: Integer;
Ycor: Integer;
TurtleThere:Boolean;

constant
North: Integer Konstante = 0;
East: Integer Konstante = 90;
South: Integer Konstante = 180;
West: Integer Konstante = 270;

B. Zusammenfassung der Operatoren

Die folgende Tabelle gibt eine Übersicht über alle Operatoren von TURBO Pascal. Die Operatoren sind nach abnehmender Wichtigkeit gruppiert. Wo der Typ des Operands als *Integer,Real* angegeben ist, ist das Ergebnis wie folgt:

Operand	**Ergebnis**
Integer,Integer	Integer
Real,Real	Real
Real,Integer	Real

Operator	**Wirkung**	**Type des Operand(S)**	**Ergebnistyp**
+ monadisch	Zeichenidentität	Integer, Real	wie Operand
- monadisch	Zeichenumkehrung	Integer, Real	wie Operand
not	Negation	Integer, Boolean	wie Operand
*	Multiplikation	Integer, Real	Integer, Real
	Schnittmenge	jeder Mengentyp	wie Operand
/	Division	Integer, Real	Real
div	Integer Division	Integer	Integer
mod	Modulus	Integer	Integer
and	arithmet. **und**	Integer	Integer
	logisches **und**	Boolean	Boolean
shl	shift nach links	Integer	Integer
shr	shift nach rechts	Integer	Integer
+	Addition	Integer, Real	Integer, Real
	Verkettung	string	string
	Mengenvereinigung	jeder Mengentyp	wie Operand
-	Subtraktion	Integer, Real	Integer, Real
	Mengenschnitt	jeder Mengentyp	wie Operand
or	arithmet. **oder**	Integer	Integer
	logisches **oder**	Boolean	Boolean
xor	arithmet. **xoder**	Integer	Integer
	logisches **xoder**	Boolean	Boolean

Operator	Wirkung	Typ des Operand	Ergebnistyp
=	Gleichheit	jeder Skalartyp	Boolean
	Gleichheit	String	Boolean
	Gleichheit	jeder Mengentyp	Boolean
	Gleichheit	jeder Zeigertyp	Boolean
<>	Ungleichheit	jeder Skalartyp	Boolean
	Ungleichheit	String	Boolean
	Ungleichheit	jeder Mengentyp	Boolean
	Ungleichheit	jeder Zeigertyp	Boolean
>=	größer oder gleich	jeder Skalartyp	Boolean
	größer oder gleich	String	Boolean
	Mengeneinschluß	jeder Mengentyp	Boolean
<=	kleiner oder gleich	jeder Skalartyp	Boolean
	kleiner oder gleich	String	Boolean
	Mengeneinschluß	jeder Mengentyp	Boolean
>	größer als	jeder Skalartyp	Boolean
	größer als	String	Boolean
<	kleiner als	jeder Skalartyp	Boolean
	kleiner als	String	Boolean
in	Mitglied einer Menge	siehe unten	Boolean

Der erste Operand des **in** Operators kann von beliebigem Skalarentyp sein, der zweite Operand muß eine Menge dieses Typs sein.

C. Zusammenfassung der Compilerbefehle

Einige der Eigenschaften des TURBO Pascal Compilers werden durch Compilerbefehle kontrolliert. Ein Compilerbefehl wird als Kommentar mit spezieller Syntax eingeführt, was bedeutet, daß überall wo ein Kommentar erlaubt ist, auch ein Compilerbefehl erlaubt ist.

Ein Compilerbefehl besteht aus einer geschweiften Klammer auf, unmittelbar gefolgt von einem Compilerbefehlsbuchstaben oder einer Liste von Compilerbefehlsbuchstaben, die durch Kommas getrennt sind. Ein Compilerbefehl wird schließlich mit einer geschweiften Klammer abgeschlossen.

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

Beachten Sie, daß keine Zwischenräume vor und nach dem Dollarzeichen erlaubt sind. Ein + Zeichen nach einem Befehl zeigt an, daß die damit verbundene Eigenschaft in Kraft gesetzt wird (aktiv) und ein - Zeichen zeigt, daß sie außer Kraft gesetzt ist (passiv).

Wichtiger Hinweis

Alle Compilerbefehle haben voreingestellte Werte. Diese wurden so gewählt, daß die Ausführungsgeschwindigkeit und die Codegröße optimiert wird. Das bringt mit sich, daß z.B. die Erzeugung von Code für rekursive Prozeduren (nur CP/M-80) und Indexprüfung außer Kraft gesetzt wurden. Prüfen Sie im Zweifelsfall also, ob ihre Programme die benötigten Compilerbefehleinstellungen enthalten!

C.1 Allgemeine Compilerbefehle

E.1.1 B - I/O Modusauswahl

Voreinstellung: B+

Der **B** Befehl kontrolliert die Auswahl des Ein-/Ausgabemodus. Wenn der Modus aktiv ist, |$B+| ist das CON: Gerät (Konsole) den Standarddateien *Input* und *Output* zugewiesen, d.h. dem voreingestellten Eingabe/Ausgabekanal. Im Passiv-Modus, |$B-|, wird das TRM: Gerät (Terminal) benutzt. **Dieser Befehl ist für das ganze Programm gültig** und kann nicht innerhalb des Programms umdefiniert werden. Weitere Details finden Sie auf den Seiten 105 und 108.

C.1.2 C - Control S und C

Voreinstellung: C+

Der **C** Befehl steuert die Interpretation der Kontrollzeichens während der Ein-/Ausgabe durch die Konsole. Im Aktiv-Modus, |$C+|, unterbricht ein Crtl-C als Antwort auf eine *Read* oder *ReadIn* Anweisung die Programmausführung und Crtl-S schaltet die Bildschirmausgabe an und aus. Im Passiv-Modus, |$C-|, werden Kontrollzeichen nicht interpretiert. Der Aktiv-Modus verlangsamt die Bildschirmausgabe etwas, falls ihnen die Ausgabegeschwindigkeit wichtig ist, müßten Sie diesen Befehl ausschalten. **Dieser Befehl ist für das ganze Programm gültig** und kann nicht während des Programms umdefiniert werden.

C.1.3 I - Eingabe/Ausgabefehler-Handhabung

Voreinstellung: I+

Der **I** Befehl kontrolliert die Ein-/Ausgabefehler-Handhabung. Im Aktiv-Modus, |$I+|, werden alle Ein-/Ausgabe-Operationen auf Fehler geprüft. Im Passiv-Modus, |$I-|, liegt es in der Verantwortung des Programmierers, I/O-Fehler durch die Standardfunktion *IOresult* zu prüfen. Für weitere Hinweise siehe Seite 116.

C.1.4 I - Include Dateien

Der **I** Befehl, gefolgt von einem Dateinamen, weist den Compiler an, die Datei mit dem angegebenen Namen in die Compilierung aufzunehmen. Include-Dateien werden in Kapitel 17 genauer beschrieben.

C.1.5 R - Index Bereichsprüfung

Voreinstellung: R -

Der **R** Befehl kontolliert die Indexprüfung zur Laufzeit. Im Aktiv-Modus, |$R+| werden alle Feldindizierungsoperationen darauf geprüft, ob sie innerhalb der definierten Grenzen sind, ebenfalls alle Zuweisungen zu Skalaren und Teilbereichsvariablen. Im Passiv-Modus |$R-| werden keine Prüfungen unternommen und Indexfehler können ein Programm dann durcheinanderbringen. Bei der Programmentwicklung empfiehlt es sich, diesen Befehl zu benutzen. Wenn es dann fehlerfrei ist, wird die Ausführung beschleunigt, indem der Passivmodus (Voreinstellung) eingestellt wird.

C.1.6 V - Var-Parametertyp Prüfung

Voreinstellung: V+

Der **V** Compilerbefehl kontrolliert die Typenprüfung von Strings, die als **var**-Parameter übergeben werden. Im Aktivmodus, |$V+|, wird strenge Typenprüfung durchgeführt, d.h. die Länge der aktuellen und formalen Parameter muß übereinstimmen. Im Passivmodus, |$V-|, erlaubt der Compiler, die Übergabe von aktuellen Parametern auch dann, wenn sie nicht zu der Länge der formalen Parameter passen. Weitere Hinweise finden Sie auf den Seiten 203, 236 und 267.

C.1.7 U - Benutzerunterbrechung

Voreinstellung: U -

Der **U** Compilerbefehl kontrolliert Programmunterbrechungen durch den Benutzer . Im Aktivmodus, |$U+|, kann der Benutzer das Programm jederzeit bei der Ausführung unterbrechen, indem er Crtl-C eingibt. Im Passivmodus, |$U-|, hat diese Eingabe keine Wirkung. Die Aktivierung dieses Befehls vermindert die Ausführungsgeschwindigkeit beträchtlich.

C.3 PC-DOS und MS-DOS Compilerbefehle

Die folgenden Befehle gelten nur für die PC-DOS/MS-DOS Implementationen:

C.3.1 G - Eingabedatei-Puffer

Voreinstellung: G0

Der **G** Befehl (für get) ermöglicht durch die Definition des Standard Eingabedatei-Puffers (*Input*) die I/O-Umlenkung. Wenn die Puffergröße 0 ist (Voreinstellung), entspricht die Eingabedatei (*input file*) *CON:* oder *TRM*. Wenn die Puffergröße ungleich 0 ist (z.B. |$G256|), entspricht sie der Standard MS-DOS Eingabebehandlung.

C.3.2 P - Ausgabedatei-Puffer

Voreinstellung: P0

Der **P** Befehl (für put) ermöglicht durch die Definition des Standard Ausgabedatei-Puffers (*Output*) die I/O-Umlenkung. Wenn die Puffergröße 0 ist (Voreinstellung), entspricht die Ausgabedatei (*output file*) *CON:* oder *TRM*. Wenn die Puffergröße ungleich 0 ist (z.B. |$G512|), entspricht sie der Standard MS-DOS Ausgabebehandlung.

Der **D** Compilerbefehl gilt für solche Ein-/Ausgabe-Pufferdateien, die nicht gleich null sind. Der **P** Befehl muß vor dem Deklarierungsteil stehen.

C.3.3 D - Geräteüberprüfung

Voreinstellung: D+

Wenn eine Textdatei mit *Reset*, *Rewrite* oder *Append* geöffnet wird, fragt TURBO Pascal von MS-DOS den Status der Datei ab. Wenn MS-DOS angibt, daß die Datei ein Gerät ist, setzt TURBO Pascal die Pufferung außer Kraft, die normalerweise bei Textdateien stattfindet. Alle Ein-/Ausgabeoperationen für die Datei erfolgen dann Zeichen für Zeichen.

Der **D** Befehl kann benutzt werden, um diese Überprüfung auszuschalten. Im voreingestellten Status {$D+} erfolgt eine Geräteüberprüfung. Ist {$D-} gesetzt, findet keine Überprüfung statt und alle Ein-/Ausgabeoperationen sind gepuffert. In diesem Fall stellt ein Aufruf der Prozedur *Flush* sicher, daß die Zeichen, die Sie in die Datei geschrieben haben auch wirklich dorthin gesendet wurden.

C.3.4 F - Zahl offener Dateien

Voreinstellung: F16

Der **F** Befehl kontrolliert die Zahl von Dateien, die zugleich geöffnet sein dürfen. Die Voreinstellung ist {$F16}, das heißt, daß bis zu 16 Dateien gleichzeitig offen sein dürfen. Wenn beispielsweise der Befehl {$F24} am Beginn des Programms (**vor** dem Deklarierungsteil) plaziert ist, dürfen bis zu 24 Dateien gleichzeitig offen sein. Der **F** Compilerbefehl begrenzt nicht die Zahl der Dateien, die in einem Programm deklariert werden können; er begrenzt nur die Zahl der Dateien, die gleichzeitig offen sein dürfen.

Auch wenn Sie mit dem F-Befehl genügend Platz für Dateien zur Verfügung gestellt haben, kann die Fehlermeldung too many open files auftreten. Dies geschieht, wenn dem Betriebssystem die Dateipuffer nicht mehr ausreichen. In diesem Fall sollten Sie den Wert für den Parameter *files = xx* in der Datei CONFIG.SYS erhöhen. Die normale Voreinstellung ist 8. Für weitere Details sollten Sie Ihre MS-DOS Dokumentation heranziehen.

C.4 PC-DOS / MS-DOS und CP/M-86 Compilerbefehle

Die folgenden Befehle gelten speziell für CP/M-86 / MS-DOS Implementationen:

C.4.1 K - Stackprüfung

Voreinstellung: K+

Der **K** Befehl kontrolliert die Stacks. Im Aktivmodus, ({$K+}), wird sicherheitshalber geprüft, ob auf dem Stack bei Aufruf eines Unterprogramms für lokale Variablen Platz vorhanden ist. Im Passivmodus ({$K-}) werden keine Prüfungen vorgenommen.

C.5 CP/M-80 Compilerbefehle

Die folgenden Befehle gelten speziell für die CP/M-80 Implementation.

C.5.1 A - Absoluter Code

Voreinstellung: A+

Der **A** Befehl kontrolliert die Erzeugung von absolutem, d.h. nicht rekursivem Code. Im Aktivmodus, (\$A+), wird absoluter Code erzeugt. Im Passivmodus (\$A-) erzeugt der Compiler einen Code, der rekursive Aufrufe erlaubt. Dieser Code benötigt mehr Speicher und ist langsamer in der Ausführung.

C.5.2 W - Schachtelung von With-Anweisungen

Voreinstellung: W2

Der **W** Befehl kontrolliert das Niveau der Schachtelung von With-Anweisungen, d.h. der Zahl von Records, die innerhalb eines Blocks geöffnet werden können. Das W muß unmittelbar von einer Ziffer zwischen 1 und 9 gefolgt sein. Für weitere Hinweise siehe Seite 81.

C.5.3 X - Arrayoptimierung

Voreinstellung: X+

Der **X** Befehl kontrolliert die Arrayoptimierung. Im Aktivmodus, (\$X+), ist die Codeerzeugung auf maximale Geschwindigkeit hin optimiert. Im Passivmodus (\$X-), minimiert der Compiler stattdessen die Codegröße. Dies wird auf Seite 75 weiter erläutert.

D. TURBO und Standard Pascal

Die TURBO Pascalsprache folgt sehr weitgehend Standard Pascal, wie es von Jensen und Wirth in Ihrem **User Manual and Report** definiert ist. Die vorhandenen kleineren Abweichungen sind aus Gründen der Effizienz eingeführt. Diese Unterschiede werden im Folgenden beschrieben. Beachten Sie, daß die Erweiterungen, die TURBO Pascal anbietet, hier nicht erläutert sind.

D.1 Dynamische Variablen

Dynamische Variablen und Zeiger benutzen die Standardprozeduren *New*, *Mark* und *Release*, statt der *New* und *Dispose* Prozeduren, die von Standard Pascal vorgeschlagen werden. Diese Abweichung vom Standard ist vorallem weit effizienter bezüglich der Ausführungsgeschwindigkeit und dem benötigten Code, außerdem bietet sie Kompatibilität mit anderen weitverbreiteten Pascalcompilern (z.B. UCSD Pascal).

Die Prozedur *New* akzeptiert keine unterschiedlichen Recordspezifikationen. Diese Einschränkung kann durch Verwendung der Standardprozedur *GetMem* leicht umgangen werden.

D.2 Rekursion

Nur CP/M-80: Wegen der Art, wie lokale Variablen während der Rekursion behandelt werden, darf eine zu einem Unterprogramm lokale Variable nicht als **var** Parameter in rekursive Aufrufe übergeben werden.

D.3 Get und Put

Die Standardprozeduren *Get* und *Put* sind nicht implementiert. Stattdessen wurden die *Read* und *Write* Prozeduren erweitert, um allen Eingabe-/Ausgabeanforderungen zu genügen. Hierfür gibt es drei Gründe:
Erstens sind *Read* und *Write* bei der Ein-/Ausgabe sehr viel schneller. Zweitens wird der gesamte Platz für Variablen reduziert, da keine Dateipuffervariablen benötigt werden. Drittens sind die *Read* und *Write* Prozeduren wesentlich vielseitiger und leichtverständlicher als *Get* und *Put*.

D.4 Goto-Anweisungen

Eine **goto** Anweisung darf den aktuellen Block nicht verlassen.

D.5 Page-Prozedur

Die Standardprozedur *Page* ist nicht implementiert, da das CP/M Betriebssystem kein Seitenvorschubszeichen definiert.

D.6 Gepackte Variablen

Das reservierte Wort *packed* hat in TURBO Pascal keine Wirkung, aber es ist dennoch erlaubt, weil Packung automatisch vorgenommen wird, wo immer es möglich ist. Aus demselben Grund sind die Standardprozeduren *Pack* und *Unpack* nicht implementiert.

D.7 Prozedurale Parameter

Prozeduren und Funktionen können nicht als Parameter übergeben werden.

E. Compiler-Fehlermeldungen

Es folgt eine Liste von Fehlermeldungen, die Sie vom Compiler bekommen können. Wenn ein Fehler auftritt, gibt der Compiler mindestens immer die Fehlernummer aus. Erklärende Texte werden nur ausgegeben, wenn Sie die Fehlermeldungsdatei auf der TURBO Diskette haben (Antwort **Y** auf die erste Frage beim Start von TURBO).

Viele Fehlermeldungen erklären sich selbst, aber einige benötigen weitere Erklärungen, wie sie im folgenden gegeben werden.

01 ';' erwartet
02 ':' erwartet
03 ',' erwartet
04 '(' erwartet
05 ')' erwartet
06 '=' erwartet
07 ':=' erwartet
08 '[' erwartet
09 ']' erwartet
10 '.' erwartet
11 '..' erwartet
12 BEGIN erwartet
13 DO erwartet
14 END erwartet
15 OF erwartet
16 PROCEDURE oder FUNCTION erwartet
17 THEN erwartet
18 TO oder DOWNTO erwartet
20 Bool'scher Begriff erwartet
21 Datei Variable erwartet
22 Integer Konstante erwartet
23 Integer Ausdruck erwartet
24 Integer Variable erwartet
25 Integer oder reelle Konstante erwartet
26 Integer oder reeller Ausdruck erwartet
27 Integer oder reelle Variable erwartet
28 Zeiger Variable erwartet
29 Record Variable erwartet

30 Einfacher Typ erwartet
einfache Tpyen sind alle skalaren Typen, außer *Reals*.

31 Einfacher Ausdruck erwartet

32 Stringkonstante erwartet

33 Stringausdruck erwartet

34 Stringvariable erwartet

35 Textdatei erwartet

36 Typenbezeichner erwartet

37 Untypisierte Datei erwartet

40 Undefiniertes Label
Ein Anweisung weist auf ein undefiniertes Label hin.

41 Unbekannter Bezeichner oder Syntaxfehler.
Unbekannt: Label, Konstante, Type, Variable, Feldbezeichner, oder Syntaxfehler in der Anweisung.

42 Undefinierter Zeigertyp in vorhergehender Typdefinitionen
Eine vorhergehende Zeigertypdefinition enthält einen Verweis auf einen unbekannten Typenbezeichner.

43 Doppelter Bezeichner oder doppeltes Label
Dieser Bezeichner oder dieses Label wurde schon in dem laufenden Block verwendet.

44 Unpassende Typen
1)Inkompatibler Typ einer Variablen und eines Ausdrucks in einem Zuweisungsstatement. **2)** Inkompatibler Typ von aktuellem und formalem Parameter in einem Unterprogrammaufruf. **3)** Typ des Ausdrucks ist inkompatibel mit dem Indextyp in der Arrayzuweisung. **4)** Die Typen von Operanden in einem Ausdruck sind nicht kompatibel.

45 Konstante außerhalb der Grenze

46 Konstanten und CASE Selektortyp passen nicht zusammen

47 Typ des Operands paßt nicht zum Operator
z.B. 'A' div '2'

48 Ungültiger Ergebnistyp
Gültige Typen sind alle Skalar-, String- und Zeigertypen.

49 Ungültige Stringlänge
Die Länge eines String muß im Bereich 1..255 liegen.

50 Stringkonstantenlänge paßt nicht zum Typ

51 Ungültiger Teilbereichsgrundtyp
Gültige Grundtypen sind alle Skalartypen, außer real.

52 Untere Grenze > obere Grenze
Der ordinale Wert der oberen Grenze muß größer oder gleich dem ordinalen Wert der unteren Grenze sein.

53 Reserviertes Wort
Diese dürfen nicht als Bezeichner verwendet werden.

54 Unerlaubte Zuweisung

55 **Stringkonstante geht über die Zeile hinaus**
Stringkonstanten dürfen sich nicht über die Zeile hinaus erstrecken.

56 **Fehler bei einer** *Integer* Konstanten
Eine *Integer* Konstante stimmt nicht mit der in Abschnitt 4.2 beschriebenen Syntax überein, oder ist nicht innerhalb des *Integer* Bereichs -32768..32767. Ganze reelle Zahlen sollten von einem Dezimalpunkt und einer Null abgeschlossen werden, z.B.123456789.0

57 **Fehler bei einer** *Real* Konstanten
Die Syntax der *Real* Konstanten ist auf Seite 43 definiert.

58 **Unerlaubtes Zeichen in einem Bezeichner**

60 **Konstanten sind hier nicht erlaubt**

61 **Dateien und Zeiger sind hier nicht erlaubt**

62 **Strukturierte Variablen sind hier nicht erlaubt**

63 **Textdateien sind hier nicht erlaubt**

64 **Textdateien und untypisierte Dateien sind hier nicht erlaubt**

65 **Untypisierte Dateien sind hier nicht erlaubt**

66 **Eingabe/Ausgabe ist hier nicht erlaubt**
Variablen diese Typs können nicht ein- oder ausgegeben werden.

67 **Dateien müssen VAR Parameter sein**

68 **Dateikomponenten dürfen keine Dateien sein**
file of file Konstrukte sind nicht erlaubt.

69 **Ungültige Ordnung von Feldern**

70 **Mengengrundtyp außerhalb des zulässigen Bereichs**
Der Grundtyp einer Menge muß ein Skalar mit nicht mehr als 256 möglichen Werten sein, oder ein Teilbereich mit den Grenzen 0..255.

71 **Unerlaubtes GOTO**
Ein **goto** kann nicht auf ein Label innerhalb einer FOR Schleife von außerhalb dieser FOR Schleife hinweisen.

72 **Label nicht innerhalb des gegenwärtigen Blocks**
Eine **goto** Anweisung kann nicht auf ein Label außerhalb des gegenwärtigen Blocks hinweisen.

73 **Undefinierte FOREWARD Prozedur(en)**
Ein Unterprogramm wurde **foreward** deklariert, aber es ist kein Block vorgekommen.

74 **INLINE Fehler**

75 **Unerlaubter Gebrauch von ABSOLUTE**
nur Bezeichner können vor dem Doppelpunkt in einer **absolute** Variablendeklaration auftreten. **2) Absolute** darf in einem Record nicht verwendet werden.

76 **Overlays können nicht FORWARD deklariert werden.**
FORWARD kann nicht in Verbindung mit Overlays verwendet werden.

77 **Im Direkt-Modus sind Overlays nicht erlaubt**
Overlays können nur von Programmen verwendet werden, die auf eine Datei compiliert sind.

90 Datei nicht gefunden
Die angegebene Include-Datei existiert nicht.

91 Unerwartetes Ende der Source
Ihr Programm kann nicht richtig enden. Das Programm hat wahrscheinlich mehr **begin** als **end** Angaben.

92 Es kann keine Overlaydatei gebildet werden.

97 Zuviele geschachtelte WITHs
Benutzen Sie den W Compilerbefehl, um die maximale Zahl von geschachtelten WITH Anweisungen zu erhöhen. Voreinstellung ist 2 (Nur CP/M-80).

98 Speicherüberlauf
Sie versuchen mehr Speicherplatz für Variablen zur Verfügung zu stellen, als vorhanden ist.

99 Compilerüberlauf
Es ist nicht genügend Speicherplatz vorhanden, um das Programm zu compilieren. Dieser Fehler kann auch auftreten, wenn freier Speicherplatz da zu sein scheint; dieser ist jedoch vom Stack und der Symboltafel bei der Compilierung belegt. Teilen Sie ihre Source in kleinere Segmente auf und benutzen Sie Include-Dateien.

F. Laufzeit-Fehlermeldungen

Schwere Laufzeit-Fehler bewirken eine Programmabbruch und die Anzeige folgender Fehlermeldung:

```
Run-time error NN, PC=addr
Program aborted
```

wobei *NN* die Nummer des Laufzeit-Fehlers ist und *addr* die Adresse im Programmcode, bei der der Fehler aufgetreten ist. Die Bedeutung der Nummern wird im Folgenden erklärt. Beachten Sie, daß alle Zahlen hexadezimal sind!

01 Gleitkommaüberlauf.

02 Sie haben versucht, durch Null zu dividieren.

03 Sqrt Argumentfehler.
Sie haben versucht, die Wurzel aus einer negativen Zahl zu ziehen, d.h. Sie haben die Sqrt-Funktion aufgerufen und als Argument eine negative Zahl eingegeben.

04 Ln Argumentfehler.
Sie haben die Ln-Funktion aufgerufen und als Argument Null oder eine negative Zahl eingegeben, d.h. versucht, den Logarithmus von Null oder einer negativen Zahl zu ermitteln.

10 String-Längenfehler.
1) Eine Verkettung von Strings ergab einen String von mehr als 255 Zeichen.**2)** Nur Strings von der Länge 1 können in Buchstabenzeichen umgewandelt werden.

11 Ungültiger Stringindex.
Der Stringindex liegt bei *Copy*, *Delete* oder *Insert* Prozeduren nicht zwischen 1 und 255.

90 Index außerhalb des zulässigen Bereichs.
Der Index eines Arrays liegt nicht im zulässigen Bereich.

91 Skalar oder Teilbereich außerhalb des zulässigen Bereichs
Sie haben einem Skalar oder Teilbereich einen Wert zugeordnet, der nicht im zulässigen Bereich liegt.

92 Außerhalb des integer Bereiches
Sie haben bei *Trunc* oder *Round* einen ganzzahligen Wert eingegeben, der außerhalb des integeren Bereiches von -32767...32767 liegt.

FF Heap/Stackkollision
Sie haben die Standardprozedur *New*, oder ein rekursives Unterprogramm aufgerufen, aber es ist nicht genügend freier Speicherplatz vorhanden.

Anmerkungen:

G. I/O Fehlermeldungen

Ein Fehler während einer Ein-/Ausgabeoperation ist ein I/O Fehler. Wenn die I/O Fehlerroutine in Betrieb ist (der I Compilerbefehl aktiviert ist), verursacht ein I/O Fehler eine Programmunterbrechung und die Anzeige der folgenden Fehlermeldung:

```
I/O error NN, PC=addr
Program aborted
```

wobei *NN* die I/O-Fehlernummer ist und *addr* die Programmadresse, bei der der Fehler aufgetreten ist.

Wenn die automatische I/O Fehlerroutine nicht in Betrieb ist ('$I-'), wird das Programm bei Auftreten eines I/O Fehlers nicht unterbrochen. Stattdessen werden alle weiteren I/O Anweisungen erst dann ausgeführt, wenn das Ergebnis der I/O Operation mit Hilfe der Standardfunktion *IOresult* überprüft worden ist. Wenn versucht wird, eine I/O Operation auszuführen, ohne daß nach einem Fehler *IOresult* aufgerufen wurde, kann dies einen Programmstillstand zur Folge haben.

Es folgt eine Erläuterung aller Laufzeit-Fehlernummern. Beachten Sie, daß es sich dabei um hexadezimale Zahlen handelt!

01 Datei ist nicht vorhanden.
Sie haben bei *Reset*, *Erase*, *Rename*, *Execute* oder *Chain* eine Datei spezifiziert, die nicht vorhanden ist.

02 Lesen der Datei nicht möglich.
1) Sie versuchen von einer Datei zu lesen (mit *Read* oder *Readln*), ohne vorheriges *Reset* oder *Rewrite*. **2)** Sie versuchen, von einer Textdatei zu lesen, (die mit *Rewrite* vorbereitet, aber noch leer ist). **3)** Sie versuchen vom logischen Gerät LST: (Drucker) einzulesen. LST: ist jedoch ein reines Ausgabegerät.

03 Ausgabe in die Datei nicht möglich.
1) Sie versuchen in eine Datei zu schreiben (mit *Write* oder *Writeln*), ohne vorheriges *Reset* oder *Rewrite*. **2)** Sie versuchen in eine Textdatei zu schreiben, die mit *Reset* vorbereitet wurde. **3)** Sie versuchen auf das logische Gerät KBD: (Tastatur) zu schreiben. KBD: ist jedoch ein reines Eingabegerät.

04 Datei nicht offen
Sie versuchen (mit *BlockRead* oder *BlockWrite*) eine Datei zu bearbeiten, ohne vorheriges *Reset* oder *Rewrite*.

10 Fehler im numerischen Format
Der String, der von einer Textdatei in eine numerische Variable eingelesen wurde, entspricht nicht dem richtigen numerischen Format (s. Abs. 4.2).

20 Operation auf einem logischen Gerät nicht zugelassen.
Sie versuchen eine Datei, die einem logischen Gerät zugeordnet ist, mit einer der folgenden Operationen zu bearbeiten: *Erase*, *Rename*, *Execute* oder *Chain*.

21 In Direktmodus nicht zugelassen.
Programme können nicht mit *Execute* oder *Chain* von einem Programm aufgerufen werden, das Direktmodus läuft, (d.h. ein Programm, das mit Compileroption **M**emory mit dem **R**un Befehl aufgerufen wurde).

22 Zuordnung als Standard-Datei nicht zulässig.

90 Unpassende Recordlänge
Die Recordlänge einer Dateivariablen entspricht nicht der Datei, der Sie sie zuzuordnen versuchen.

91 Suchen Sie nach end-of-file.

99 Unerwartetes end-of-file.
1) Beim Lesen von einer Textdatei wurde das physikalische Dateiende vor dem EOF-Zeichen erreicht. **2)** Es wurde versucht, bei einer fest definierten Datei über das end-of-file hinaus zu lesen.

F0 Diskettenschreibfehler
Beim Versuch, eine Datei zu vergrößern, ist die Diskette voll geworden. Dies kann bei den Ausgabeoperationen *Write*, *Writeln*, *BlockWrite* und *Flush* auftreten, aber auch *Read*, *Readln* und *Close* können diesen Fehler verursachen, wenn sie den Schreibpuffer zum Überlaufen bringen.

F1 Directory ist voll
Sie versuchen eine neue Datei neu zu erstellen, aber es ist kein Platz mehr in der Directory.

F2 Dateigrößenüberschreitung
Sie versuchen auf einer definierten Datei auf einen Record über 65535 hinaus zu schreiben.

FF Datei verschwunden
Es wurde ein Versuch unternommen eine Datei zu schließen, die nicht mehr in der Directory vorhanden ist, z.B. weil die Diskette gewechselt wurde.

H. Übersetzung von Fehlermeldungen

Die Fehlermeldungen des Compilers sind in der Datei TURBO.MSG enthalten. Diese Meldungen sind englisch, können aber leicht in jede andere Sprache übersetzt werden.

Die ersten 24 Zeilen der Datei definieren eine Zahl von Textkonstanten, die später in Fehlermeldungszeilen einbezogen werden. Diese Technik reduziert den Disketten- und Speicherplatzbedarf der Fehlermeldungen beträchtlich. Jede Konstante wird durch ein Kontrollzeichen identifiziert, angedeutet durch ein ^ Zeichen in der folgenden Liste. Der Wert jeder Konstanten ist alles das, was in der gleichen Zeile folgt. Alle Zeichen sind signifikant (auch vorausgestellte und folgende Leerzeichen).

Jede verbleibende Zeile enthält eine Fehlermeldung, beginnend mit der Fehlernummer auf die unmittelbar der Fehlermeldungstext folgt. Der Fehlermeldungstext kann aus beliebigen Zeichen bestehen und kann vordefinierte Konstantenbezeichner (Kontrollzeichen) einbeziehen. Anhang E enthält die ausgeschriebenen Fehlermeldungen.

Wenn Sie die Fehlermeldungen übersetzen, ist die Beziehung von Konstanten und Fehlermeldungen wahrscheinlich von der hier aufgelisteten englischen Version abweichend. Beginnen Sie deshalb damit, jede Fehlermeldung auszuschreiben, ohne Rücksicht auf den Gebrauch von Konstanten. Sie können diese Fehlermeldungen verwenden, sie benötigen aber beträchtlichen Speicherplatz.

Wenn alle Meldungen übersetzt sind, sollten Sie soviele allgemeine Bezeichnungen wie möglich finden, dann diese als Konstanten am Anfang der Datei definieren und nur die Konstantenbezeichner in die folgenden Textmeldungen einbeziehen. Sie können soviele Konstanten definieren, wie Sie benötigen. Die einzige Restriktion ist die Zahl der Kontrollzeichen.

Gute Beispiele für den Gebrauch von Konstanten sind die Fehler 25, 26 und 27. Diese sind ausschließlich als Konstantenbezeichner definiert, im ganzen 15; ausgeschrieben würden sie 101 Zeichen benötigen.

Der TURBO Editor kann verwendet werden, um die TURBOMSG.OVR Datei zu editieren. Kontrollzeichen werden mit Ctrl-P Präfix eingegeben, z.B. um Ctrl-A (^A) in eine Datei einzugeben, müssen Sie die <Ctrl> Taste gedrückt halten und dann zuerst P und dann A drücken. Kontrollzeichen erscheinen auf dem Bildschirm (falls dieser über, das entsprechende Videoattribut verfügt) in verminderter Helligkeit.

Beachten Sie, daß der TURBO Editor alle nachfolgenden Leerzeichen löscht. Die Originalmeldungen haben deshalb keine.

H.1 Auflistung der Fehlermeldungsdatei

```
^A are not allowed
^B can not be
^C constant
^D does not
^E expression
^F identifierer
^G file
^H here
^KInteger
^LFile
^NIllegal
^O or
^PUndefined
^Q match
^R real
^SString
^TTextfile
^U out of range
^V variable
^W overflow
^X expected
^Y txpe
^| Invalid
^|pointer
01';'^X
02':'^X
03','^X
04'('^X
05')'^X
06'='^X
07':='^X
08'|'^X
09'|'^X
10'.'^X
11'..'^X
12BEGIN^X
13DO^X
14END^X
```

15OF^X
17THEN^X
18TO^O DOWNTO^X
20Boolean^E^X
21^I^V^X
22^K^C^X
23^K^E^X
24^K^V^X
25^K^O^R^C^X
26^K^O^R^E^X
27^K^O^R^V^X
28Pointer^V^X
29Record^V^X
30Simple^Y^X
31Simple^E^X
32^S^C^X
33^S^E^X
34^S^V^X
35^T^X
36Type^F^X
37Untyped^G^X
40^P label
41Unknown^F^O syntax error
42^P^|^Y in preceding ^Y definitions
43Duplicate^F^O label
44Type mismatch
45^C^U
46^C and CASE selector^Y^D^Q
47Operand^Y(s)^D^Q operator
48^| result^Y
49^| ^S length
50^S^C length^D^Q^Y
51^| subrange base^Y
52Lower bound) upper bound
53Reserved word
54^N assignment
55^S^C exceeds line
56Error in integer^C
57Error in ^R^C
58^N character in^F
60^Cs^A^H
61^Ls and ^|s^A^H
62Structured^Vs^A-41H
63^Ts^A^H

64^Ts and untyped ^Gs^A^H
65Untyped ^Gs^A^H
66I/O ^A
67^Ls must be ^V parameters
68^L components ^B^Gs
69^|^Ordering of fields
70Set base ^Y^U
71^| GOTO
72Label not within current block
73^P FORWARD Procedure(s)
74INLINE error
75^N use of ABSOLUTE
90^L not found
91Unexpected end of source
97Too many nested WITH's
98Memory ^W
99Compiler ^W

I. Turbosyntax

Die Syntax der TURBO Pascal Sprache wird hier in dem als *Backus-Naur Form* bekannten Formalismus dargestellt. Die folgenden Symbole sind Metasymbole, die zum BNF Formalismus gehören. Sie sind nicht Symbole der TURBO Pascal Sprache.

::=	Bedeutet ist definiert als
|	Bedeutet oder
{ }	Die eingeschlossenen Sprachelemente können null oder mehrmals wiederholt werden.

Alle anderen Symbole sind Teil der Sprache. Jedes syntaktische Konstrukt ist kursiv gedruckt, z.B. *block* und *case-element*. Reservierte Wörter stehen in Fettdruck , z.B. **array** und **for**.

actual-parameter ::= *expression* | *variable*
adding-operator ::= + | - | **or** | **xor**
array-constant ::= (*structured-constant* { , *structured-constant* })
array-type ::= **array** [*index-type* { , *index-type* }] **of** *component-type*
array-variable ::= *variable*
assignment-statement ::= *variable* := *expression* |
function-identifier ::= *expression*
base-type ::= *simple-type*
block ::= *declaration-part statement-part*
case-element ::= *case-list* : *statement*
case-label ::= *constant*
case-label-list ::= *case-label* { , *case-label* }
case-list ::= *case-list-element* { , *case-list-element*}
case-list-element ::= *constant* | *constant* .. *constant*
case-statement ::= **case** *expression* **of** *case-element* { ; *case-element* } **end** |
case *expression* **of** *case-element* { ;*case-element* }
otherwise *statement* { ; *statement* } **end**
complemented-factor ::= *signed-factor* | **not** *signed-factor*
component-type ::= *type*
component-variable ::= *indexed-variable* | *field-designator*
compound-statement ::= **begin** *statement* { ; *statement* } **end**
conditional-statement ::= *if-statement* | *case-statement*

constant :: = *unsigned-number* | *sign unsigned-number* | *constant-identifier*
| *sign constant-identifier* | *string*
constant-definition-part :: = **const** *constant-definition*
{ ; *constant-definition* } ;
constant-definition :: = *untyped-constant-definition* |
typed-constant-definition
constant-identifier :: = *identifier*
control-character :: = # *unsigned-integer* | ^ *character*
control-variable :: = *variable-identifier*
declaration-part :: = { *declaration-section* }
declaration-section :: = *label-declaration-part* | *constant-definition-part* |
type-definition-part | *variable-declaration-part* |
procedure-and-function-declaration-part
digit :: = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
digit-sequence :: = *digit* { *digit* }
empty :: =
empty-statement :: = *empty*
entire-variable :: = *variable-identifier* | *typed-constant-identifier*
expression :: = *simple-expression* { *relational-operator simple-expression* }
factor :: = *variable* | *unsigned-constant* | (*expression*) |
function-designator | *set*
field-designator :: = *record-variable* . *field-identifier*
field-identifier :: = *identifier*
field-list :: = *fixed-part* | *fixed-part* ; *variant-part* | *variant-part*
file-identifier :: = *identifier*
file-identifier-list :: = *empty* | (*file-identifier* { , *file-identifier* }
file-type :: = **file of** *type*
final-value :: = *expression*
fixed-part :: = *record-section* { ; *record-section* }
for-list :: = *initial-value* **to** *final-value* | *initial-value* **downto** *final-value*
for-statement :: = **for** *control-variable* : = *for-list* **do** *statement*
formal-parameter-section :: = *parameter-group* | **var** *parameter-group*
function-declaration :: = *function-heading block* ;
function-designator :: = *function-identifier* | *function-identifier*
(*actual-parameter* { , *actual-parameter* })
function-heading :: = **function** *identifier* : *result-type* ; |
function *identifier* (*formal-parameter-section*
{ , *formal-parameter-section* }) : *result-type* ;
function-identifier :: = *identifier*
goto-statement :: = **goto** *label*
hexdigit :: = *digit* | A | B | C | D | E | F
hexdigit-sequence :: = *hexdigit* { *hexdigit* }
identifier :: = letter { *letter-or-digit* }
identifier-list :: = *identifier* { , *identifier* }

if-statement :: = **if** *expression* **then** *statement* [**else** *statement*]
index-type :: = *simple-type*
indexed-variable :: = *array-variable* [*expression* { , *expression* }]
initial-value :: = *expression*
inline-list-element :: = *unsigned-integer* | *constant-identifier* |
variable-identifier | *location-counter-reference*
inline-statement :: = **inline** *inline-list-element* { , *inline-list-element* }
label :: = *letter-or-digit* { *letter-or-digit* }
label-declaration-part :: = **label** *label* { , *label* } ;
letter :: = A | B | C | D | E | F | G | H | I | J | K | L | M |
N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
a | b | c | d | e | f | g | h | i | j | k | l | m |
n | o | p | q | r | s | t | u | v | w | x | y | z | _
letter-or-digit :: = *letter* | *digit*
location-counter-reference :: = * | * *sign constant*
multiplying-operator :: = * | / | **div** | **mod** | **and** | **shl** | **shr**
parameter-group :: = *identifier-list* : *type-identifier*
pointer-type :: = ^ *type-identifier*
pointer-variable :: = *variable*
procedure-and-function-declaration-part :: =
{ *procedure-or-function-declaration* }
procedure-declaration :: = *procedure-heading block* ;
procedure-heading :: = **procedure** *identifier* ; | **procedure** *identifier*
(*formal-parameter-section*
{ ; *formal-parameter-section* }) ;
procedure-or-function-declaration :: = *procedure-declaration* |
function-declaration
procedure-statement :: = *procedure-identifier* | *procedure-identifier*
(*actual-parameter* { , *actual-parameter* })
program-heading :: = *empty* | **program** *program-identifier*
file-identifier-list
program :: = *program-heading block* .
program-identifier :: = *identifier*
record-constant :: = (*record-constant-element*
{ ; *record-constant-element* })
record-constant-element :: = *field-identifier* : *structured-constant*
record-section :: = *empty* | *field-identifier* { , *field-identifier* } : *type*
record-type :: = **record** *field-list* **end**
record-variable :: = *variable*
record-variable-list :: = *record-variable* { , *record-variable* }
referenced-variable :: = *pointer-variable* ^
relational-operator :: = = | < > | < = | > = | < | > | **in**
repeat-statement :: = **repeat** *statement* { ; *statement* } **until** *expression*
repetitive-statement :: = *while-statement* | *repeat-statement* | *for-statement*

result-type :: = *type-identifier*
scalar-type :: = (*identifier* { , *identifier* })
scale-factor :: = *digit-sequence* | *sign digit-sequence*
set :: = [{ *set-element* }]
set-constant :: = [{ *set-constant-element* }]
set-constant-element :: = *constant* | *constant* .. *constant*
set-element :: = *expression* | *expression* .. *expression*
set-type :: = **set of** *base-type*
sign :: = + | -
signed-factor :: = *factor* | *sign factor*
simple-expression :: = *term* { *adding-operator term* }
simple-statement :: = *assignment-statement* | *procedure-statement* |
goto-statement | *inline-statement* | *empty-statement*
simple-type :: = *scalar-type* | *subrange-type* | *type-identifier*
statement :: = *simple-statement* | *structured-statement*
statement-part :: = *compound-statement*
string :: = { *string-element* }
string-element :: = *text-string* | *control-character*
string-type :: = **string** [*constant*]
structured-constant :: = *constant* | *array-constant* | *record-constant* |
set-constant
structured-constant-definition :: = *identifier* : *type* = *structured-constant*
structured-statement :: = *compound-statement* | *conditional-statement* |
repetitive-statement | *with-statement*
structured-type :: = *unpacked-structured-type* |
packed *unpacked-structured-type*
subrange-type :: = *constant* .. *constant*
tag-field :: = *empty* | *field-identifier* :
term :: = *complemented-factor* { *multiplying-operator complemented-factor* }
text-string :: = ' { *character* } '
type-definition :: = *identifier* = *type*
type-definition-part :: = **type** *type-definition* { ; *type-definition* } ;
type-identifier :: = *identifier*
type :: = *simple-type* | *structured-type* | *pointer-type*
typed-constant-identifier :: = *identifier*
unpacked-structured-type :: = *string-type* | *array-type* | *record-type* |
set-type | *file-type*
unsigned-constant :: = *unsigned-number* | *string* | *constant-identifier* | **nil**
unsigned-integer :: = *digit-sequence* | $ *hexdigit-sequence*
unsigned-number :: = *unsigned-integer* | *unsigned-real*
unsigned-real :: = *digit-sequence* . *digit-sequence* |
digit-sequence . *digit-sequence* E *scale-factor* |
digit-sequence E *scale-factor*
untyped-constant-definition :: = *identifier* = *constant*

variable :: = *entire-variable* | *component-variable* | *referenced-variable*
variable-declaration :: = *identifier-list* : *type* |
identifier-list : *type* **absolute** *constant*
variable-declaration-part :: = **var** *variable-declaration*
{ ; *variable-declaration* } ;
variable-identifier : = *identifier*
variant :: = *empty* | *case-label list* : (*field-list*)
variant-part :: = **case** *tag-field type-identifier* **of** *variant* { ; *variant* }
while-statement :: = **while** *expression* **do** *statement*
with-statement :: = **with** *record-variable-list* **do** *statement*

Anmerkungen:

J. ASCII Tabelle

DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR	DEC	HEX	CHAR
0	00	^@ NUL	32	20	SPC	64	40	@	96	60	`
1	01	^A SOH	33	21	!	65	41	A	97	61	a
2	02	^B STX	34	22	"	66	42	B	98	62	b
3	03	^C ETX	35	23	#	67	43	C	99	63	c
4	04	^D EOT	36	24	$	68	44	D	100	64	d
5	05	^E ENQ	37	25	%	69	45	E	101	65	e
6	06	^F ACK	38	26	&	70	46	F	102	66	f
7	07	^G BEL	39	27	'	71	47	G	103	67	g
8	08	^H BS	40	28	(	72	48	H	104	68	h
9	09	^I HT	41	29	)	73	49	I	105	69	i
10	0A	^J LF	42	2A	*	74	4A	J	106	6A	j
11	0B	^K VT	43	2B	+	75	4B	K	107	6B	k
12	0C	^L FF	44	2C	,	76	4C	L	108	6C	l
13	0D	^M CR	45	2D	-	77	4D	M	109	6D	m
14	0E	^N SO	46	2E	.	78	4E	N	110	6E	n
15	0F	^O SI	47	2F	/	79	4F	O	111	6F	o
16	10	^P DLE	48	30	0	80	50	P	112	70	p
17	11	^Q DC1	49	31	1	81	51	Q	113	71	q
18	12	^R DC2	50	32	2	82	52	R	114	72	r
19	13	^S DC3	51	33	3	83	53	S	115	73	s
20	14	^T DC4	52	34	4	84	54	T	116	74	t
21	15	^U NAK	53	35	5	85	55	U	117	75	u
22	16	^V SYN	54	36	6	86	56	V	118	76	v
23	17	^W ETB	55	37	7	87	57	W	119	77	w
24	18	^X CAN	56	38	8	88	58	X	120	78	x
25	19	^Y EM	57	39	9	89	59	Y	121	79	y
26	1A	^Z SUB	58	3A	:	90	5A	Z	122	7A	z
27	1B	^[ESC	59	3B	;	91	5B	[	123	7B	{
28	1C	^\ FS	60	3C	<	92	5C	\	124	7C	|
29	1D	^] GS	61	3D	=	93	5D	]	125	7D	}
30	1E	^^ RS	62	3E	>	94	5E	^	126	7E	~
31	1F	^_ US	63	3F	?	95	5F	_	127	7F	DEL

Anmerkungen:

K. Tastatur-Returncodes

Dieser Anhang enthält eine Liste der Codes, die durch alle möglichen Tastenkombinationen der Tastaur des IBM-PC erzeugt werden, so wie TURBO Pascal sie sieht. Tatsächlich erzeugen Funktionstasten und mit Alt zusammen bedingte Tasten erweiterte Scan-Codes, diese werden von TURBO Pascal aber in EScape-Sequenzen verwandelt.

Um die Escape-Sequenzen zu lesen, sollten Sie Ihre Leseroutine nach ESC suchen lassen und wenn ein ESC gefunden wird, überpüfen, ob sich im Tastaturpuffer ein weiteres Zeichen befindet. Ist das der Fall, wurde ein Escape-Code empfangen, lesen Sie dann das nächste Zeichen und setzen einen Merker, um zu signalisieren, daß dieses ein normales Zeichen ist, das der zweite Teil einer Escape-Sequenz darstellt.

```
if Keypressed then
begin
    Read(Kbd,Ch); | Ch is char |
    if (Ch = #27) and Keypressed then | one more char |
    begin
        Read(Kbd,Ch)
        FuncKey := True; | FuncKey is boolean |
    end
end;
```

Die folgende Tabelle listet die Returncodes als dezimale ASCII-Werte auf. Normale Tasten erzeugen einen einfachen Code; erweiterte Codes erzeugen ein ESC(27) auf das ein oder mehrere Zeichen folgen.

Taste	normal	mit Shift	Ctrl	Alt
F1	27 59	27 84	27 94	27 104
F2	27 60	27 85	27 95	27 105
F3	27 61	27 86	27 96	27 106
F4	27 62	27 87	27 97	27 107
F5	27 63	27 88	27 98	27 108
F6	27 64	27 89	27 99	27 109
F7	27 65	27 90	27 100	27 110
F8	27 66	27 91	27 101	27 111
F9	27 67	27 92	27 102	27 112
F10	27 68	27 93	27 103	27 113

Taste	normal	mit Shift	Ctrl	Alt
LArr	27 75	52	27 115	27 178
RArr	27 77	54	27 116	27 180
UArr	27 72	56	27 160	27 175
DArr	27 80	50	27 164	27 183
Home	27 71	55		27 174
End	27 79	49	27 117	27 182
PgUp	27 73	57	27 132	27 176
PgDn	27 81	51	27 118	27 184
Ins	27 82	48	27 165	27 185
Del	27 83	46	27 166	27 186
Esc	27	27	27	
BackSp	8	8	127	
Tab	9	27 15		
RETURN	13	13	10	
A	97	65	1	27 30
B	98	66	2	27 48
C	99	67	3	27 46
D	100	68	4	27 32
E	101	69	5	27 18
F	102	70	6	27 33
G	103	71	7	27 34
H	104	72	8	27 35
I	105	73	7	27 23
J	106	74	10	27 36
K	107	75	11	27 37
L	108	76	12	27 38
M	109	77	13	27 50
N	110	78	14	27 49
O	111	79	15	27 24
P	112	80	16	27 25
Q	113	81	17	27 16
R	114	82	18	27 19
S	115	83	19	27 31
T	116	84	20	27 20
U	117	85	21	27 22
V	118	86	22	27 47
W	119	87	23	27 17
X	120	88	24	27 45
Y	121	89	25	27 21
Z	122	90	26	27 44

Taste	normal	mit Shift	Ctrl	Alt
[	91	123	27	
\	92	124	28	
]	93	125	29	
'	96	126		
0	48	41		27 129
1	49	33		27 120
2	50	64	27 3	27 121
3	51	35		27 122
4	52	36		27 123
5	53	37		27 124
6	54	94	30	27 125
7	55	38		27 126
8	56	42		27 127
9	57	40		27 128
*	42		27 114	
+	43	43		
-	45	95	31	27 130
=	61	43		27 131
,	44	60		
/	47	63		
;	59	58		

Tabelle K - 1: Tastatur-Returncodes

Anmerkungen:

L. Terminal-Installierung

Bevor Sie TURBO Pascal benutzen, müssen Sie es auf Ihr Terminal installieren, d.h. mit Information über Kontrollzeichen für bestimmte Funktionen versehen. Die Installation ist mit Hilfe des *TINST* Programms leicht durchzuführen. In diesem Kapitel ist das *TINST* Programm beschrieben.

Nachdem Sie eine Arbeitskopie gemacht haben, bewahren Sie bitte Ihre Originaldiskette sicher auf und arbeiten nur auf der Kopie.

Beginnen Sie nun die Installation, indem Sie *TINST* eingeben. Wählen Sie Screen Installation aus dem Hauptmenü. Je nach der Version Ihres TURBO Pascal folgt die Installation so wie es in den nächsten zwei Kapiteln beschrieben ist.

L.1 IBM-PC Bildschirmwahl

Wenn Sie TURBO Pascal ohne Installation verwenden, wird der voreingestellte Bildschirmaufbau verwendet. Sie können diese Voreinstellung durch Auswahl eines anderen Bildschirm-Modus aus diesem Menü verändern:

```
Choose one of the following displays:

0) Default display mode
1) Monochrome display
2) Color display 80x25
3) Color display 40x25
4) b/w display 80x25
5) b/w display 40x25

Which display (enter no. or  X  to exit) :
```

Abbildung L - 1: IBM-PC Bildschirminstallations-Menü

Wenn Sie einen Bildschirm ausgewählt haben, wird dieser für die zukünftigen TURBO-Aufrufe automatisch aktiviert. Verlassen Sie das Menü mit ^x, wird der Bildschirm auf den ursprünglich voreingestellten (Default)-Modus zurückgesetzt.

L.2 Installierung bei Nicht-IBM PC Rechnern

Es erscheint ein Menü, das eine Reihe weitverbreiteter Terminals auflistet. Sie werden aufgefordert einen davon zu wählen, indem Sie seine Nummer eintippen:

```
Choose one of the following terminals:

 01) ADDS 20/25/30              15) Lear-Siegler ADM-31
 02) ADDS 40/60                 16) Liberty
 03) ADDS Viewpoint-1A          17) Morrow MDT-20
 04) ADM 3A                     18) Otrona Attache
 05) Ampex D80                  19) Qume
 06) ANSI                       20) Soroc IQ-120
 07) Apple/graphics             21) Soroc new models
 08) Hazeltine 550              22) Teletext 3000
 09) Hazeltine Esprit           23) Televideo 912/920/925
 10) IBM-PC CCP/M b/w           24) Visual 200
 11) IBM-PC CCP/M color         25) Wyse WY-100/200/300
 12) Kaypro 10                  26) Zenith
 13) Kaypro II and 4            27) None of the above
 14) Lear-Siegler ADM-20        28) Delete a definition

Which terminal ? (Enter no. or ^X to exit):
```

Abbildung L-2: Terminal-Installationsmenü (Nicht-IBM PC)

Wenn Ihr Terminal vorkommt, geben Sie nur die entsprechende Zahl ein und die Installation ist beendet. Bevor die Installation tatsächlich vorgenommen wird, werden Sie gefragt:

```
Do you want to modifiy the definition before installation ?
```

Dies erlaubt Ihnen einen oder mehrere Werte der Installierung, die im Folgenden beschrieben sind, zu modifizieren. Wenn Sie die Definition Ihres Terminals nicht modifizieren wollen, tippen Sie nur **N** ein. Die Installation endet dann mit der Frage nach der Arbeitsgeschwindigkeit der CPU ihres Rechners (siehe letzter Punkt dieses Anhangs).

Wenn Ihr Terminal **nicht** im Menü vorkommt, müssen Sie die erforderlichen Werte selbst definieren. Diese Werte können Sie sehr wahrscheinlich in dem Handbuch Ihres Terminals finden.

Geben Sie die Zahl ein, die bei **None of the above** steht, und beantworten Sie die auf dem Bildschirm erscheinenden Fragen eine nach der anderen.

Im Folgenden ist jeder Befehl, den Sie installieren können, genau beschrieben. Ihr Terminal unterstützt möglicherweise nicht alle Befehle, die installiert werden könnten. Wenn dies der Fall ist, übergehen Sie diesen Befehl einfach, indem Sie auf das entsprechende Stichwort nur ‹RETURN› eingeben. Wenn *Delete line*, *Insert line* oder *Erase to end of line* nicht installiert sind, werden diese Funktionen durch die Software emuliert, was die Bildschirmausgabe etwas verlangsamt.

Befehle können entweder einfach durch Drücken der entsprechenden Tasten eingegeben werden, oder durch Eingabe des dezimalen oder hexadezimalen Wertes des Befehls. Wenn ein Befehl die zwei Zeichen 'ESCAPE' und '=' erfordert, kann:

entweder zuerst die **Esc** Taste und dann = gedrückt werden, die Eingabe wird mit den entsprechenden Labels beantwortet, d.h. ‹ESC› =,

oder Sie geben die dezimalen oder hexadezimalen Werte getrennt durch Leerzeichen ein. Hexadezimalen Werten muß ein Dollarzeichen vorangestellt sein. Geben Sie z.B. 27 61 oder $1B 61 oder $1B $3D ein; die Werte in den Beispielen sind jeweils äquivalent.

Die zwei Methoden können nicht gemischt werden, d.h. wenn Sie ein nicht-numerisches Zeichen eingegeben haben, muß der Rest der Befehle ebenfalls auf diese Weise definiert werden und umgekehrt.

Die Eingabe eines Bindestrichs als erstes Zeichen wird dazu verwendet, um einen Befehl zu löschen. Es erscheint das Wort *Nothing*.

L.2.1 Terminaltyp

Geben Sie den Namen des Terminals ein, das Sie installieren wollen. Wenn Sie *TINST* beenden, werden die Werte gespeichert und der Terminalname erscheint auf der Liste der Terminals. Falls Sie später TURBO Pascal nochmals für das Terminal installieren müssen, können Sie es auf der Liste anwählen.

L.2.2 Soll ein Initialisierungsstring an das Terminal geschickt werden?

Wenn Sie Ihr Terminal initialisieren wollen, wenn Turbo startet (z.B. um Befehle auf Funktionstasten zu programmieren), antworten Sie **Y** auf diese Frage. Wenn nicht, drücken Sie RETURN.

L.2.3 Soll ein Resetstring an das Terminal geschickt werden?

Hier können Sie einen String definieren, der bei Beendigung von TURBO Pascal an das Terminal geschickt wird. Die obige Beschreibung des Initialisierungsbefehls gilt hier ebenfalls.

L.2.4 CURSOR LEAD-IN Befehl

Cursor Lead-in ist eine spezielle Sequenz von Zeichen, die Ihrem Terminal mitteilt, daß die folgenden Zeichen eine Adresse auf dem Bildschirm sind, auf die der Cursor plaziert werden soll. Wenn Sie diesen Befehl definieren, werden Ihnen die folgenden zusätzlichen Fragen gestellt:

L.2.4.1 Soll der CURSOR POSITIONING Befehl zwischen Zeilen- und Spaltenadresse geschickt werden?

Bei einigen Terminals muß der Befehl zwischen den zwei Zahlen, die die Reihen- und Spaltencursoradresse definieren, stehen.

L.2.4.2 Soll der CURSOR POSITIONING Befehl nach der Zeile und Spalte geschickt werden ?

Bei einigen Terminals muß der Befehl nach der Definition der Zeilen- und Spaltenadresse des Cursor stehen.

L.2.4.3 Spalte zuerst?

Bei den meisten Terminals muß die Adresse in folgendem Format stehen: zuerst REIHE, dann SPALTE. Wenn dies bei Ihrem Terminal so ist, antworten Sie mit **N**. Wenn Ihr Terminal zuerst die SPALTE benötigt und dann die REIHE, antworten Sie **Y**.

L.2.4.4 Offset der ersten Zeilennummer

Geben Sie die Zahl ein, die zur Zeilenadresse addiert werden soll.

L.2.4.5 Offset der ersten Spalte

Geben Sie sie Zahl ein, die zur Spaltenadresse addiert werden soll.

L.2.4.6 Binäre Adresse?

Den meisten Terminals muß die Cursoradresse in binärer Form geschickt werden. Falls das auch für Ihr Terminal gilt, geben Sie **Y** ein. Wenn Ihr Terminal die Cursoradresse in ASCII Ziffern erwartet, geben Sie **N** ein. Sie werden dann zusätzlich gefragt:

L.2.4.6.1 2 oder 3 ASCII Ziffern?
Geben Sie die Anzahl von ASCII-Ziffern in der Cursoradresse Ihres Terminals an.

L.2.5 CLEAR SCREEN Befehl
Geben Sie den Befehl ein, der den gesamten Bildschirminhalt löscht (wenn nötig sowohl für den Vorder-, als auch für den Hintergrund).

L.2.6 Führt CLEAR SCREEN auch den Cursor in HOME Position?
Dies ist normalerweise der Fall; wenn dies für Ihr Terminal nicht zutrifft, geben Sie **N** ein und definieren den Cursor HOME Befehl.

L.2.7 DELETE LINE Befehl
Geben Sie den Befehl ein, der die Cursor-Zeile ganz löscht.

L.2.8 INSERT LINE Befehl
Geben Sie den Befehl ein, der an der Cursorposition eine ganze Zeile einfügt.

L.2.9 ERASE TO END OF LINE Befehl
Geben Sie den Befehl ein, der die Zeile von der Cursorposition bis zum rechten Zeilenende löscht.

L.2.10 START OF 'LOW VIDEO' Befehl
Wenn Ihr Terminal über verschiedene Bildintensitäten verfügt, dann geben Sie hier den Befehl ein, der den Bildschirm abdunkelt. Wenn dieser Befehl definiert ist, wird die folgende Frage gestellt:

L.2.10.1 START OF 'NORMAL VIDEO' Befehl
Definieren Sie den Befehl, der den Bildschirm auf |Normalbild| setzt.

L.2.11 Zahl der Zeilen auf Ihrem Bildschirm
Geben Sie die Zahl der horizontalen Zeilen Ihres Terminals an.

L.2.12 Zahl der Spalten auf Ihrem Bildschirm
Geben Sie die Zahl der vertikalen Spaltenpositionen an.

L.2.13 Verzögerung nach CURSOR ADDRESS (0-255ms):
Verzögerung nach CLEAR, DELETE und INSERT (0-255ms):
Verzögerung nach ERASE TO END OF LINE und HIGHLIGHT On/Off (o-255 ms) :
Geben Sie die gewünschte Verzögerung in Millisekunden für die angegebenen Funktionen ein. ‹RETURN› bedeutet Null (keine Verzögerung).

L.2.14 Ist diese Definition korrekt?

Wenn Sie in den Definitionen Fehler gemacht haben, antworten Sie **N**. Sie kehren dann zum Terminalmenü zurück. Die Installationsdaten, die Sie gerade eingegeben haben, sind in der Installationsdatendatei enthalten und erscheinen im Terminalmenü. Die Installation wird aber **nicht** durchgeführt.

Wenn Sie auf diese Frage **Y** eingeben, werden Sie gefragt:

L.2.15 Arbeitsfrequenz Ihres Mikroprozessors in MHz (für Verzögerungen)

Da die weiter oben angegebenen Verzögerungen von der Arbeitsfrequenz Ihrer CPU abhängen, müssen Sie diesen Wert definieren.

Die Installation ist beendet, die Installationsdaten werden auf die Diskette geschrieben und Sie kehren in das höhere Menü zurück (siehe Abschnitt 1.6). Die Installationsdaten sind auch in der Installationsdatendatei gespeichert. Das neue Terminal erscheint in der Terminalwahlliste bei zukünftigen *TINST*-Läufen.

L.1.3 Installierung der Editor-Kommandos

Der integrierte Editor besteht aus einer Reihe von Kommandos, die z.B. den Cursor auf dem Bildschirm bewegen, Texte einfügen, löschen, usw.. Jede dieser Funktionen kann durch eines von zwei Kommandos aktiviert werden: ein erstes und ein zweites Kommando. Die zweiten Kommandos sind von Borland installiert und entsprechen weitgehend den Standard-Kommandos von *WordStar*. Die ersten Kommandos sind für die meisten Systeme undefiniert und können mit dem Installationsprogramm leicht nach Ihren Wünschen definiert werden. IBM PC Systeme werden mit als ersten Kommandos installierten Pfeil- und Funktionstasten geliefert, wie in Kapitel 19 beschrieben.

Wenn Sie **C** für Kommando-Installation eintippen, erscheint das erste Kommando:

```
CURSOR MOVEMENTS:

1: Charakter left     Nothing        -) ■
```

Dies bedeutet, daß kein erstes Kommando installiert ist, um den Cursor ein Zeichen nach links zu bewegen. Wenn Sie ein erstes Kommando zusätzlich zu dem zweiten *WordStar entsprechenden Kommando CTRL-S benutzen wollen (das hier nicht dargestellt ist), haben Sie dazu zwei Mö glichkeiten, dies nach -> einzugeben:*

1) Drücken Sie einfach die Taste, die Sie verwenden wollen. Es kann eine Funktionstaste sein (z.B. die Taste mit dem Pfeil in linker Richtung, wenn sie verfügbar ist), oder eine andere Taste oder Folge von Tasten (max. 4). Das Installierungs-Programm antwortet mit einem Mnemonic für jedes Zeichen, das eingegeben wird. Wenn Sie eine Linkspfeil-Taste haben, die ein <ESCAPE> Zeichen mit einem darauffolgenden **a** überträgt und Sie diese Taste in der oben beschriebenen Situation drücken, erscheint nach der Eingabe auf dem Bildschirm:

```
CURSOR MOVEMENTS:

1:    Charakter left          Nothing -) (ESC) a
```

2) Anstatt die Taste zu drücken, die Sie momentan benutzen wollen, können Sie auch den (die) ASCII-Wert(e) des (der) Zeichens eingeben. Die Werte aus zusammengesetzten Kommandos werden eingegeben, indem ein Leerzeichen dazwischen steht. Die Dezimalwerte werden einfach eingegeben, z.B. 27, den Hexadezimalzeichen steht ein Dollarzeichen voran, z.B. $1B. Dies kann hilfreich sein, um Kommandos zu installieren, die vorerst nicht auf Ihrer Tastatur verfügbar sind, beispielsweise wenn Sie ein neues Terminal installieren wollen, Ihnen aber nur das alte zur Verfügung steht. Das mag selten vorkommen, aber nichtsdestotrotz haben Sie diese Möglichkeit.

In beiden Fällen beenden Sie Ihre Eingabe mit <RETURN>. Beachten Sie, daß die beiden Methoden nicht bei einem einzelnen Kommando miteinander gekoppelt werden können, d.h. wenn Sie begonnen haben, eine Kommandozeichen-Sequenz mit der Tastaur zu definieren, müssen alle Zeichen in diesem Kommando durch Drücken der entsprechenden Tasten eingegeben werden und umgekehrt.

Um ein Kommando von der Liste zu löschen, geben Sie ein '-' (Minus) ein. Ein **B** führt um ein Kommando auf der Liste zurück.

Der Editor akzeptiert insgesamt 45 Kommandos, die Sie alle verändern können. Falls Ihnen bei der Installierung ein Fehler unterläuft, z.B. Sie das selbe Kommando für zwei unterschiedliche Funktionen eingegeben haben, wird eine erklärende Fehlermeldung ausgegeben und Sie müssen Ihren Irrtum korrigieren, bevor Sie die Installierung beenden. Es mag vorkommen, daß ein erstes Kommando mit einem zweiten *WordStar* ähnlichen Kommando unverträglich ist; in diesem Fall wird das zweite Kommando unzugänglich.

Die folgende Tabelle listet die Werte und die Art des jeweiligen Kommandos auf. Etwaige Änderungen können Sie in dem dafür vorgesehenen Platz eintragen.

CURSORSTEUERUNGS-KOMMANDOS

1:	Zeichen nach links	Ctrl-S			
2:	Alternative	Ctrl-H			
3:	Zeichen nach rechts	Ctrl-D			
4:	Wort nach links	Ctrl-A			
5:	Wort nach rechts	Ctrl-F			
6:	Zeile nach oben	Ctrl-E			
7:	Zeile nach unten	Ctrl-X			
8:	Aufwärts rollen	Ctrl-W			
9:	Abwärts rollen	Ctrl-Z			
10:	Seite nach oben	Ctrl-R			
11:	Seite nach unten	Ctrl-C			
12:	Zeile links	Ctrl-Q	Ctrl-S		
13:	Zeile rechts	Ctrl-Q	Ctrl-D		
14:	Oberer Bildschirmrand	Ctrl-Q	Ctrl-E		
15:	Unterer Bildschirmrand	Ctrl-Q	Ctrl-X		
16:	Textbeginn	Ctrl-Q	Ctrl-R		
17:	Textende	Ctrl-Q	Ctrl-C		
18:	Blockanfang			Ctrl-Q	Ctrl-B
19:	Blockende			Ctrl-Q	Ctrl-B
20:	Letzte Cursorposition			Ctrl-Q	Ctrl-P

	EINFÜGEN/LÖSCHEN-KOMMANDOS		
21:	Einfügen Modus an/aus	Ctrl-V	
22:	Zeile einfügen	Ctrl-N	
23:	Zeile löschen	Ctrl-Y	
24:	Löschen bis zum Zeilenende	Ctrl-Q	Ctrl-Y
25:	Rechtes Wort löschen	Ctrl-T	
26:	Zeichen unter Cursor löschen	Ctrl-G	
27:	Linkes Zeichen löschen	<DEL>	
28:	Alternative:	Keine	
	BLOCK-KOMMANDOS		
29:	Blockanfang-Markierung	Ctrl-K Ctrl-B	
30:	Blockende-Markierung	Ctrl-K Ctrl-K	
31:	Markierung eines einzelnen Worts	Ctrl-K Ctrl-T	
32:	Block verdecken/zeigen	Ctrl-K Ctrl-W	
33:	Kopieren eines Blocks	Ctrl-K Ctrl-C	
34:	Versetzen eines Blocks	Ctrl-K Ctrl-V	
35:	Löschen eines Blocks	Ctrl-K Ctrl-Y	
36:	Lesen eines Blocks aus einer Datei	Ctrl-K Ctrl-R	
37:	Schreiben eines Blocks in eine Datei	Ctrl-K Ctrl-W	
	VERSCHIEDENARTIGE EDITIER-KOMMANDOS		
38:	Beenden des Editierens	Ctrl-K Ctrl-D	
39:	Tabulierung	Ctrl-I	
40:	Automatische Tabulierung aus/an	Ctrl-Q Ctrl-I	
41:	Zeilensicherung	Ctrl-Q Ctrl-L	
42:	Finden	Ctrl-Q Ctrl-F	
43:	Finden und Ersetzen	Ctrl-Q Ctrl-A	
44:	Wiederholen der letzten Suche	Ctrl-L	
45:	Kontrollzeichen-Präfix	Ctrl-P	

Tabelle L-3: Werte der Editier-Kommandos

Punkt 2 und 28 erlauben Alternativen bei der Definition der Kommandos für |Linksbewegung um einen Buchstaben| und |Löschung des linken Buchstabens| anzugeben. Normalerweise ist die Alternative zu Ctrl-S <BS> und zu <DEL> gibt es keine Alternative. Um Ihrer Tastatur gerecht zu werden, können Sie das ändern, z.B. durch den Gebrauch von <BS> zusätzlich zu <DEL>, wenn Sie auf der Tastatur leichter zu erreichen ist. Natürlich müssen die alternativen Kommandos eindeutig definiert sein.

Anmerkungen:

CP/M LEITFADEN

Benutzung von TURBO auf einem CP/M System

Wenn Sie Ihren Computer anschalten, werden die ersten Spuren auf Ihrer CP/M Diskette gelesen und eine Kopie des CP/M Betriebssytems wird in den Speicher geladen. Immer wenn Sie Ihren Computer neu booten, erzeugt CP/M eine Liste des zur Verfügung stehenden Diskettenspeicherplatzes für jedes Diskettenlaufwerk. Jedesmal wenn Sie versuchen eine Datei auf der Diskette zu sichern, erfolgt eine
Überprüfung von CP/M, um sicherzustellen, daß die Diskette nicht gewechselt wurde. Falls Sie z.B. die Diskette im Laufwerk A ohne neu zu booten gewechselt haben, gibt CP/M folgende Fehlermeldung aus, wenn versucht wird auf die Diskette zu schreiben:

BDOS ERROR ON A: R/O

Das Betriebssystem erhält die Kontrolle zurück und Ihre Arbeit wurde NICHT gesichert! Das kann das Kopieren von Disketten für den Anfänger etwas verwirrend machen. Wenn CP/M für Sie neu ist, sollten Sie den nachstehenden Anweisungen folgen:

Das Kopieren Ihrer TURBO Diskette

Um von Ihrer TURBO Original-Diskette ein Arbeitskopie zu machen, tun Sie folgendes:

1. Löschen Sie eine Diskette und speichern Sie eine Kopie von CP/M darauf (Im CP/M Handbuch finden Sie mehr dazu). Das ist nun Ihre TURBO **Arbeitsdiskette.**
2. Plazieren Sie diese Diskette im Laufwerk A: Plazieren Sie eine CP/M Diskette mit einer Kopie von PIP.COM im Laufwerk B (PIP.COM ist das Dateikopierprogramm von CP/M, das auf Ihrer CP/M Diskette sein sollte. Das CP/M Handbuch enthält dazu mehr Einzelheiten).
3. Booten Sie den Computer neu. Geben Sie B:PIP ein und drücken Sie ‹RETURN›.
4. Nehmen Sie die Diskette aus dem Laufwerk B: und legen Sie Ihre TURBO Original-Diskette ein.
5. Nun drücken Sie: A:=B:*.*|V| und dann ‹RETURN›.

Sie haben PIP angewiesen alle Dateien von der Diskette im Laufwerk B: auf die Diskette im Laufwerk A: zu kopieren. Falls Fehler entstehen, schauen Sie am besten in Ihrem CP/M Handbuch nach.

Die letzten Zeilen Ihres Bildschirms sollten dann folgendermaßen aussehen:

```
A>B:*.*|Vs09

*A:=B:*.*|V|


COPYING -
FIRSTFILE
:
:
LASTFILE
*
```

6. Drücken Sie <RETURN>, und das PIP Programm wird beendet.

Benutzung Ihrer TURBO Diskette

Lagern Sie Ihre TURBO Original-Diskette an einer sicheren Stelle. Um TURBO Pascal zu benutzen, legen Sie Ihre neue TURBO *Arbeitsdiskette* in Laufwerk A: ein und booten das System neu. Wenn Ihr TURBO für Ihren Computer und Ihr Terminal nicht vorinstalliert ist, müssen Sie TURBO installieren (siehe Seite 12). Wenn Sie das getan haben, tippen Sie

TURBO

und TURBO Pascal startet.

Wenn Sie mit dem Kopieren Ihrer Diskette Schwierigkeiten haben, schauen Sie bitte in Ihr CP/M Benutzerhandbuch oder fragen Ihren Hardware-Händler um Rat.

N.Hilfe!!!

Dieser Anhang enthält eine Reihe der meistgestellten Fragen und die Antworten darauf. Wenn Sie die Antwort auf *Ihre* Fragen hier nicht finden, können Sie jederzeit den Heimsoeth Kundenservice in Anspruch nehmen.

F: Wie benutze ich das System ?
A: Bitte lesen Sie das Handbuch, speziell Kapitel 1. Falls Sie sofort beginnen müssen, tun Sie folgendes:

1. Booten Sie Ihr Betriebssystem
2. Falls Sie keinen IBM PC haben, rufen Sie TINST auf, um TURBO für Ihr Gerät zu installieren
3. Starten Sie TURBO
4. Starten Sie das Programm!

F: Ich habe Probleme mein Terminal zu installieren!
A: Wenn Sie ein Teminal haben, das nicht im Installationsmenü vorhanden ist, müssen Sie ihr Terminal selbst installieren. Allen Terminals ist eine schriftliche Anleitung beigefügt, die Informationen über Codes enthalten, die die Bildschirm I/O steuern. Sie müssen die Fragen in dem Installationsprogramm gemäß der Information des Handbuches zu Ihrer Hardware beantworten. Wir benutzen eine Terminologie, die der Anforderung am besten entspricht. Beachten Sie, daß die meisten Terminals kein Initialisierungs- oder Resetstring erfordern. Diese werden benutzt um auf zusätzliche Features spezieller Terminals zurückzugreifen. Beispielsweise können Sie an einigen Terminals ein Initilisierungsstring dazu verwenden, daß die Tastatur als Cursorblock arbeitet. Sie können bis zu 13 Zeichen in Initialisierungs- oder Resetstrings einsetzen.

F: Ich habe Probleme mit der Diskette. Wie kann ich meine Disketten kopieren?
A: Die meisten Probleme bedeuten nicht gleich, daß Sie eine defekte Diskette haben. Speziell, wenn Sie auf einem CP/M-80 System arbeiten, sollten Sie den kurzen CP/M-80 Leitfaden auf Seite 355 betrachten. Wenn Sie die Directory Ihrer Original-Diskette erhalten, sind die Chancen gut, daß es sich um eine intakte Diskette handelt.

Um eine Sicherungskopie zu machen, sollten Sie ein Datei für Datei Kopierprogramm benutzen, wie *COPY* für PC/MS-DOS, oder *PIP* für CP/M-80/86. Der Grund dafür ist, daß diejenigen von Ihnen, die Diskettenlaufwerke mit doppelte Dichte haben, Schwierigkeiten haben könnten ein DISKCOPY-Programm zu verwenden. Diese Programme erwarten für die Zieldiskette das exakt gleiche Format, wie für die Quelldiskette.

F: Brauche ich den 8087 Co-Prozessor, um Turbo-87 benutzen zu können?
A: Ja, wenn Sie Programme für den 8087 Chip compilieren wollen, muß dieser Chip in Ihrer Anlage vorhanden sein. Der Standard TURBO Compiler ist zusätzlich auch noch auf der Turbo-87 Diskette enthalten, so daß Sie beide Versionen benutzen können.

F: Brauche ich eine spezielle Ausrüstung, um TURBO-BCD zu benutzen?
A: Nein, das BCD System arbeitet jedoch nur auf 16-Bit Implementierungen von Turbo.

F: Brauche ich Turbo, um mit Turbo entwickelte Programme laufen zu lassen?
A: Nein, Turbo kann .COM oder .CMD Dateien anlegen.

F: Wie lege ich .COM oder .CMD Dateien an?
A: Drücken Sie im Hauptmenü O für 'compiler Options' und wählen Sie dann C für .COM oder.CMD Datei.

F: Wo liegen die Grenzen des Compilers im Bezug auf Code und Daten?
A: Der Compiler kann bis zu 64K Code, 64K Daten, 64K Stack und unbegrenzt Heap bearbeiten. Der Objektcode kann jedoch 64K nicht übersteigen.

F: Wo liegt die Grenze des Editors im Bezug auf Speicherplatz?
A: Der Editor kann bis zu 64K zu einem Zeitpunkt bearbeiten. Wenn das nicht genug ist, können Sie Ihren Source in mehr als eine Datei aufteilen, indem Sie den $I Compilerbefehl verwenden. Dies ist in Kapitel 17 erläutert.

F: Was kann ich tun, wenn ich Fehler 99 (Compiler-Überlauf) erhalte?
A: Sie können zwei Dinge tun: Teilen Sie Ihren Code in kleinere Segmnete und benutzen den $I Compilerbefehl (in Kapitel 17 erklärt); oder compilieren Sie auf eine .COM oder.CMD Datei.

F: Was kann ich tun, wenn mein Objektcode länger als 64K wird?
A: Benutzen Sie entweder Chain-Dateien oder Overlays.

F: Wie kann ich von der Tastaur Zeichen einlesen, ohne daß ich RETURN drücken muß?
A: Folgendermaßen: read(Kbd,Ch) wobei *Ch:Char*.

F: Wie lasse ich etwas über den Drucker ausgeben?
A: Versuchen Sie es mit: WriteLn(Lst,...).

F: Wie erhalte ich eine Ausgabe meines Quellencodes über den Drucker?
A: Sie können folgendes Programm benutzen. Wenn Sie eine Ausgabe wünschen, die reservierte Wörter unterstreicht oder fett druckt, die den Seitenvorschub steuert und alle Include-Dateien ausgibt, finden Sie ein solches Programm auf der TURBO Tutor Diskette (einschließlich Source).

```
program TextFileDemo;

var
     TextFile : Text;
     Scratch : String[128]
;

     begin
     Write('File to print: ');              { Get file name }
     Readln(Scratch);

     Assign(TextFile, Scratch);        { Open the file to}
     {$I-}
     Reset(TextFile);
     {$I+};
     if IOresult <> 0 then
     Writeln('Cannot find ', Scratch)       { File not found }
     else                                   { Print the file }
     begin
while not Eof(TextFile) do
begin
     Readln(TextFile, Scratch);             { Read a line }
     Writeln(Lst, Scratch);                 { Print a line }
end; { while }
     Writeln(Lst)                           { Flush printer buffer }
     5end { else }
end.
```

F: Wie erfolgt die Ein- Ausgabe von COM1:
A: so: writeln(AUX,...), nachdem Sie mit MODE in MSDOS den Port angegeben haben, oder dasselbe mit einem ASSIGN-Programm in CP/M gemacht haben. Um vom Port zu lesen geben Sie folgendes an: read(AUX,...). Sie müssen dabei bedenken, daß beim Lesen von AUX kein Puffer automatisch angelegt wird.

F: Wie kann ich eine Funktionstaste lesen?
A: tionstasten erzeugen 'erweiterte Scan-Codes', die von TURBO in 'Escape-Sequenzen' geändert werden, d.h. zwei Zeichen werden von der Tastatur gesendet: Als erstes ein ESC (dezimaler ASCII-Wert 27), und dann irgend ein anderes Zeichen. Sie finden auf Seite 341 alle Werte in einer Tabelle.

Diese erweiterten Codes zu lesen, prüfen Sie nach, ob ein ESC steht und falls ja, schauen Sie, ob noch ein anderes Zeichen im Tastaturpuffer ist. Wenn ja, wurde eine Funktionstaste gedrückt, deshalb lesen Sie das nächste Zeichen und setzen ein Flag, um zu signalisieren, daß das erhaltene kein normales Zeichen, sondern der zweite Teil einer Escape-Sequenze' ist.

```
if Key Pressed then
begin
      Read(Kbd,Ch)                    | ch is char |
      if (ch = #27) and Key Pressed then
      | one more char? |
      begin
        Read(Kbd,Ch)
        FuncKey : = True;             | FuncKey is boolean |
      end
end;
```

F: Ich habe Probleme mit der Dateikennzeichnung. Wie ist die korrekte Reihenfolge der Anweisungen, um eine Datei zu öffnen?
A: Hier sehen Sie die korrekte Art Dateien zu bearbeiten?

Um eine neue Datei zu erzeugen:

```
Assign(FileVar,'NameOf.Fil');
Rewrite(FileVar);
:
:
Close(FileVar);
```

Um eine existierende Datei zu öffnen:

```
Assign(FileVar,'NameOf.Fil');
Reset(FileVar);
:
:
Close(FileVar);
```

F: Warum funktionieren meine rekursiven Prozeduren nicht?
A: Schalten Sie den A Compilerbefehl aus:|$A-| (nur bei CP/M-80)

F: Wie kann ich EOF:und EOLN benutzen ohne eine Dateivariable als Parameter anzugeben.
A: Stellen Sie die gepufferte Eingabe ab:|$B-|

F: Wie stelle ich fest, ob eine Datei auf der Diskette existiert?
A: Benutzen Sie |$I-| und (I+). Die folgende Funktion gibt *True* aus, wenn der Dateiname, der als Parameter übergeben wurde, existiert, andernfalls gibt sie *False* aus:

```
function Exist(FileName: Name) : Boolean;
var
    fil: file;
begin
    assign(Fil, FileName);
    |$I-|
    Reset(Fil);
    |$I+|
    Exist := (IOresult = 0)
end;
```

F: Wie setze ich CTRL-C außer Kraft?
A: Setzen Sie den Compilerbefehl: |$C-|.

F: Ich erhalte die Fehlermeldung Type mismatch bei der Übergabe von Strings als an eine Funktion oder Prozedur.
A: Stellen Sie die Typprüfung von Variablenparametern ab: |$V-|.

F: Wenn ich mein Programm compiliere, erhalte ich bei einer Includedatei den Fehler File not found, obwohl die Datei in der Direktory steht.
A: Bei der Benutzung des Include-Compilerbefehls |$I filename.ext| muß zwischen dem Dateinamen und der abschließenden Klammer ein Leerzeichen sein, wenn die Dateitypbezeichnung nicht drei Zeichen lang ist: |$ISample.F |. Andernfalls wird die Klammer als Teil des Dateinamens interpretiert.

F: Warum verhält sich mein Programm verschieden, wenn ich es mehrmals hintereinander laufen lasse?
A: Wenn Sie Ihr Programm im Memory-Modus durchlaufen lassen und typisierte Konstanten als initialisierte Variablen benutzen, werden diese Konstanten nur unmittelbar nach der Compilierung initialisiert und nicht aber bei jedem Lauf Ihres Programms, da sich typisierte Konstanten im Codesegment liegen. Bei .COM Dateien existiert dieses Problem nicht; falls Sie immer noch verschiedene Ergebnisse erhalten, wenn Sie Arrays und Mengen benutzen, stellen Sie die Wertebereichprüfung an |$R+|.

F: Ich erhalte nicht die Ergebnisse, die ich erhalten sollte, wenn ich *Reals* und *Integers* im selben Ausdruck benutze.

A: Wenn Sie einen *integer* Ausdruck einer *real* Variablen zuweisen, wird der Ausdruck in *Real* umgewandelt. Der Ausdruck selbst wird jedoch als *integer* berechnet und Sie sollten deshalb auf einen mögliche Integer-Überlauf in einem Ausdruck achten. Dies kann zu überraschenden Egebnissen führen. Nehmen Sie als Beispiel:

```
RealVar := 40 * 1000;
```

Zuerst multipliziert der Compiler die *integer* Zahlen 40 und 1000, was zum Ergebnis 40.000 und gleichzeitig zu einem Integer-Uberlauf führt. Tatsächlich führt es zu dem Ergebnis -25536 und wird so *RealVar* zugewiesen. Um dies zu verhindern, benutzen Sie entweder:

```
RealVar := 40.0 * 1000;
```

oder

```
RealVar := 1.0 * IntVar1 * IntVar2;
```

um sicherzustellen, daß der Ausdruck als Real berechnet wird.

F: Wie erhalte ich eine Directory für mein TURBO Programm?

A: Einfache Prozeduren für den Zugriff auf die Directory sind in dem TURBO Tutor Paket enthalten (TURBO Tutor können Sie bei Heimsoeth Software München bestellen).

0. Stichwortverzeichnis