Technische Informatik

Grafik

VEB Verlag Technik Berlin

Grafik
dBASE III

Technische Informatik

Herausgegeben von

Dr.-Ing. Ludwig Claßen

Dr.-Ing. Dieter Nedo

Dr. rer. nat. Gerhard Paulin

dBASE III

Dr. sc. Wilfried Grafik

VEB VERLAG TECHNIK BERLIN

Grafik, Wilfried :
dBase III / Wilfried Grafik. - 1. Aufl.
- Berlin : Verl. Technik, 1989. -
240 S. : 31 Bilder, 15 Taf.
ISBN 3-341-00703-2

ISBN 3-341-00703-2
ISSN 0863-0860 Technische Informatik

1. Auflage

Lizenz 201• 370/85/89
Printed in the German Democratic Republic
Druck und buchbinderische Verarbeitung:
Druckerei "Neues Deutschland", Berlin
Lektor: HS-Ing. Hans G. Schüler
Einbandgestaltung: Gabriele Schwesinger
LSV 3043 • VT 3/6010-1
Bestellnummer: 554 093 2
02400

Vorwort

8- und 16-Bit-Rechner gehören heute zur alltäglichen Arbeitsumgebung für viele von uns. Fast monatlich erscheinen neue Produkte der Hard- und Software, die es rechtfertigen, die Realisierbarkeit bestimmter Probleme auf diesen Rechnern zu überprüfen. Wachsende interne und externe Speicherkapazitäten, erhöhte Geschwindigkeiten, bessere Algorithmen und problemnahe Nutzerschnittstellen mit grafischer und Farbunterstützung sind die Visitenkarten dieser Technik. Sie lassen diese Entwicklung mit einer Geschwindigkeit voranschreiten, die vor fünf Jahren kaum absehbar war.

Immer mehr Probleme führen dabei auf Fragen aus dem Gebiet der Datenbanken. Da Festplatten von 20 bis 50 MByte praktisch verfügbar sind, lassen sich durchaus Datenbanken mittlerer Größe organisieren. Zum Vergleich: An Universalrechnern der siebziger Jahre waren Wechselplatten von 7,5 MByte üblich.

Für den potentiellen und den konkreten Anwender gilt es jetzt zu entscheiden, unter welchen Voraussetzungen er sein Problem als Datenbank auf einem 16-Bit-Rechner realisieren kann. Was kann die Hardware? Was kann dBASE III? Wie finde ich den Einstieg in den Entwurf und die Handhabung von Datenbanken? Was geschieht, wenn ich die Grenzen von dBASE erreiche? Wie kann ich sie erweitern, wie umgehen? Wie geht es von dBASE aus weiter? Kann ich meine Datenbank auch auf zukünftigen Rechnern und Systemen verwenden? Und für die direkte Arbeit mit dBASE möchte man wissen: Wie finde ich schnell und präzise Informationen über einzelne Sprachelemente von dBASE? All diese Fragen werden in diesem Buch behandelt.

Das Buch wendet sich an den Einsteiger und soll ihm bei der späteren Arbeit eine Unterstützung als Nachschlagewerk sein. Es werden kaum Vorkenntnisse über Computer oder Programmiersprachen vorausgesetzt. Alle benutzten Begriffe werden erklärt, sobald sie über das übliche Wissen hinausgehen. Es ist selbstverständlich, daß z.B. dBASE II-Kenner über viele Dinge leicht hinweglesen werden. Das sollte nicht der Maßstab sein. Probieren Sie die Beispiele an Ihrem Rechner! Die scheinbar schwierigen Passagen werden Sie schätzen lernen, wenn Sie Details suchen oder später nachschlagen wollen.

Dieses Buch ist dadurch für Nutzer in der Industrie, in der Ausbildung und in den verschiedenen Leitungsebenen geeignet, praktisch überall dort, wo es darauf ankommt, Daten zu Informationen zusammenzustellen.

Man muß damit rechnen, daß dBASE III sehr schnell weiterentwickelt wird und daß noch mehr Programme geschaffen werden, die sich um dBASE III ranken. Es ist unmöglich, all diese Systeme im einzelnen zu kennen, und es ist auch nicht sinnvoll, sie alle benutzen zu wollen. Konzentrieren Sie sich auf Ihre Aufgaben und wählen Sie die dafür geeigneten Werkzeuge. dBASE III wird für 16-Bit-Rechner lange im Mittelpunkt stehen! Es wäre auch gar nicht gut, wenn alle Tochterprogramme von dBASE und dBASE selbst vereint wären. Wir wollen

schließlich auch noch Daten auf unserer Festplatte unterbringen!

Mein besonderer Dank gilt Herrn Dr. G. Paulin, der mir in allen Etappen der Erstellung dieses Buches mit wertvollen Ratschlägen zur Seite stand. Den Herren J. Reichenbach und H. G. Schüler vom Verlag Technik danke ich für ihre Unterstützung und die gute Zusammenarbeit in verlagstechnischen Dingen. Nicht zuletzt ein Dankeschön an meine Frau für die sorgfältige Durchsicht mehrerer Manuskriptversionen. Erwähnt werden sollen auch die Hinweise von Kollegen, Partnern der Industrie und Teilnehmern an Vorträgen, die ebenfalls zur Abrundung der Darstellungen dieses Buches beitrugen.

Wilfried Grafik

Inhaltsverzeichnis

1. Einführung

Dieses Buch behandelt das Datenbanksystem dBASE III. dBASE III wurde für 16-Bit-Mikrorechner entwickelt, speziell für IBM-PC-kompatible Rechner unter dem Betriebssystem MS-DOS. Der Vorgänger von dBASE III, dBASE II, ist das erfolgreichste Datenbanksystem auf 8-Bit-Mikrorechnern. Seine Lauffähigkeit auf allen CP/M-fähigen Rechnern mit einer Standardperipherie war ein wichtiger Fakt dafür, daß dBASE II eine weite nationale und internationale Verbreitung erreichte.

Betrachtet man den Einsatz von 8-Bit-Mikrorechnern, ggf. mit Ausnahme der OEM-Rechner, so entfallen etwa 90% der Einsatzfälle auf drei Aufgabenklassen: Datenbanken, Textverarbeitung und Tabellenberechnung. Die Klasse der Aufgaben, die auf Datenbanken führen, nimmt davon schätzungsweise über 30% ein. Ein guter Grund dafür, Grundlagen, Entwurf, Implementation und Unterhaltung von Datenbanken genauer zu betrachten. Darin besteht das Anliegen des vorliegenden Buches.

Die 16-Bit-Mikrorechner mit ihren wesentlich größeren externen Speichern und ihrer höheren Verarbeitungsgeschwindigkeit bieten die Möglichkeit, Datenbanken mittlerer Größe zu verarbeiten, für die die Kapazität der 8-Bit-Rechner nicht ausreichte. dBASE III ist nicht nur als eine Weiterentwicklung von dBASE II anzusehen. Es berücksichtigt in seiner Gesamtkonzeption die charakteristischen Eigenschaften der 16-Bit-Mikrorechner. dBASE III reflektiert diese Eigenschaften in einer fortgeschrittenen Dialogführung im Direktmodus (full screen mode), in einer Berücksichtigung eines hierarchischen Dateikonzepts und in einer Vielzahl erweiterter Parametergrenzen gegenüber dBASE II. Betrachten wir den Einsatz der 16-Bit-Computer, so zeigt sich, daß sich der Trend zur Nutzung von Datenbanksystemen, speziell von dBASE, auch auf diesen Rechnern fortsetzt.

Der Behandlung von dBASE III wurde gegenüber dBASE II in diesem Buch der Vorzug gegeben, da es das modernere System ist, welches der Leistungsfähigkeit der 16-Bit-Rechner angepaßt ist und in einigen Ansätzen auch vorwärts weist. Wir sind uns dabei sehr wohl der Tatsache bewußt, daß sehr viele Anwender Datenbanken in dBASE II besitzen und diese weiter nutzen möchten. Ein Übergang von dBASE II zu dBASE III ist möglich (Abschn. 17.4). Ferner denken wir an die Anwender, die auch auf 16-Bit-Mikrorechnern vorerst nur dBASE II zur Verfügung haben. Aus diesen Gründen beschreiben wir dBASE III und weisen an wichtigen Stellen auf Unterschiede zu dBASE II hin. Eine Einführung in dBASE II findet man bei Eggerichs [Egg86.1], [Egg86.2]. Beide Systeme werden z.B. in [Cas85] und [Cas87] behandelt.

Alle angeführten Beispiele wurden auf einem IBM-PC-kompatiblen Rechner mit dBASE III, Version 1.10, unter MS-DOS 3.20 getestet. Kommandofolgen, sequentielle Ausgaben von dBASE, Bildschirminhalte, z.B. Bildschirmmasken, und Ausgaben im Direktmodus sind direkte Kopien vom Rechner, ohne daß sie nochmals von Hand eingegeben wurden.

Durch das ganze Buch führt ein Beispiel, welches durch seine Nähe

zum täglichen Leben die Probleme der Datenbanken veranschaulichen soll, die wesentlichen Prinzipien widerspiegelt und in seinen Detailangaben (Adressen, Preise, Namen usw.) frei erfunden ist.

Dieses Buch wendet sich an den **Einsteiger**. Sollten Sie sich dazu zählen, so beginnen Sie mit den Abschnitten 2 und 3. Überspringen Sie ggf. den Abschnitt 2.1 und setzen Sie dann mit den Abschnitten 5 bis 11 fort. Abschnitt 4 beschreibt die in dBASE gültigen Ausdrücke. Sie werden sowohl Vorwärts- als auch Rückwärtsverweise auf diesen Abschnitt finden. Folgen Sie ihnen, wenn Sie sich nicht sicher sind, was im konkreten Fall ein gültiger Ausdruck ist. Abschnitt 4 ist nur dann zum sequentiellen Lesen geeignet, wenn man tiefer in das Wesen der Ausdrücke eindringen will. Für ein erstes Verstehen von dBASE reicht eine intuitive Vorstellung von Ausdrücken. Beide Aspekte zusammen genommen rechtfertigen die Stellung dieses Abschnitts.
Sie sollten dann mit den Abschnitten 13 und 16 fortfahren. Sind Sie bis dahin vorgedrungen und haben Sie einige Beispiele am Rechner nachvollzogen, so sollten Sie Ihr erstes kleines Übungprojekt in Angriff nehmen. Danach gehören Sie nicht mehr zu den Einsteigern. Möchten Sie sich sehr gründlich mit dBASE beschäftigen, so lesen Sie dieses Buch sequentiell.

Dieses Buch wendet sich an die Nutzer von dBASE II. Sind Sie ein **Umsteiger**, so sollten Sie trotzdem mit dem Abschnitt 2.6 beginnen und danach ab Abschnitt 10 fortfahren. Die Abschnitte 17.1 und 18 eignen sich dabei weniger zum sequentiellen Lesen als zum "Durchblättern". Interne Eigenschaften von dBASE sind in den Abschnitten 19 und 20 enthalten.

Dieses Buch soll als **Nachschlagewerk** benutzbar sein. Dazu sind alle Funktionen in alphabetischer Reihenfolge im Abschnitt 4.7 zusammengestellt. Eine inhaltliche Zusammenstellung aller Funktionen befindet sich im Anhang 1. Die SET-Kommandos sind im Abschnitt 17.1 alphabetisch geordnet, und alle Kommandos findet man, ebenfalls alphabetisch geordnet, im Abschnitt 18. Man wird sich die Details des internen Aufbaus (Abschn. 19) und die Ansteuerung des Druckers aus dBASE heraus nicht merken können. Auch in diesem Sinne soll dieses Buch zum Nachschlagen geeignet sein.

Da das Buch eine Einführung und zum Nachschlagen geeignet sein soll und aus weiteren inhaltlichen Gründen sind Vorwärtsverweise erforderlich und gerechtfertigt. Die angegebenen Beispiele sind gleichzeitig als Übung gedacht, ohne daß sie im Text als solche ausgewiesen werden. Versuchen Sie, ob Sie zu den gleichen Ergebnissen kommen! Die Beispiele zeigen sowohl erwartete als auch unerwartete Ergebnisse. Da dBASE sehr sparsam mit Fehlermitteilungen ist, sind kritische Teile eines Programms erst "im Kleinen" zu testen. Auf einige Probleme weisen wir in diesem Buch hin, andere werden immer vom konkreten Problem abhängig sein, wieder andere treten erst nach einer Langzeitnutzung zutage.
Viel Spaß also mit dBASE III !

2. Grundlagen

In diesem Abschnitt beschäftigen wir uns mit einigen grundlegenden Begriffen und Eigenschaften von Datenbanken. Abschnitt 2.1 behandelt die Stellung von dBASE unter den Datenbankmodellen. Danach schränken wir uns ein und erklären den Begriff "Datenbank", wie er in dBASE verstanden wird. Vorher weisen wir auf den besonders hohen Wert hin, den eine funktionsfähige Datenbank verkörpert (Abschn. 2.2). Ferner werden Hinweise gegeben, die es gestatten, zu überprüfen, ob dBASE für ein vorliegendes Problem ein geeignetes Werkzeug ist.

2.1. Datenbanken und Datenbankmodelle

Höhere Programmiersprachen sind seit dem Anfang der fünfziger Jahre bekannt, z. B. FORTRAN. Es entstand die Notwendigkeit, die Compiler selbst, Nutzerprogramme, Quelltexte und Programmdaten auf Speichermedien abzulegen, die direkt (on line) mit dem Rechner verbunden waren. Es entstanden Filesysteme. Mit der Zunahme der nichtnumerischen Datenverarbeitung fielen immer mehr Daten an, die nach einem einheitlichen Verwaltungsapparat verlangten, der über die Fähigkeiten eines Filesystems hinausging. Verkörpert ein Filesystem eine Sammlung von Daten, so sind Programme Algorithmen zur Verarbeitung von Daten. Ein Datenbanksystem ist eine Sammlung von Datenstrukturen, Daten und dafür typischen Zugriffsalgorithmen. Datenbanksysteme gestatten eine vom konkreten Anwendungsfall unabhängige Beschreibung von Daten und Zugriffsalgorithmen; sie unterstützen Auswahl- und Suchoperationen und die Verknüpfung von Datensätzen nach vorgegebenen Kriterien. Sie weisen eine gewisse Stabilität gegenüber Änderungen auf. Teilaufgaben davon lassen sich auch in höheren Programmiersprachen formulieren. In höheren Programmiersprachen (Pascal, Modula, C) fehlen jedoch die typischen, relativ komplexen Sprachelemente für die Datenbankoperationen. Ein Datenbanksystem ist also mehr als Sprachen und Files. Schlagwortartig kann es durch folgende Eigenschaften charakterisiert werden:

1. Daten und Datenstrukturen stehen im Vordergrund.
2. Es werden große Datenmengen behandelt.
3. Es sind komplexe Operationen implementiert, die für die Verarbeitung dieser Daten typisch sind.
4. Der prozedurale Aspekt der Datenverarbeitung tritt etwas zurück.
5. Dafür wird zusätzlich eine Adressierung von Objekten durch ihren Inhalt vorgenommen.

Datenbanksysteme entstanden durch den Bedarf in praktischen Bereichen. Sie entwickelten sich schrittweise aus der Bewältigung immer größerer Datenmengen mit Programmiersprachen, z.B. COBOL. Während theoretische Betrachtungen zu Programmiersprachen und Filesystemen den ersten praktischen Einsätzen unmittelbar folgten (formale Defini-

tion der Syntax von ALGOL 60), galten theoretische Untersuchungen zu Datenbanken bis zum Ende der sechziger Jahre als nicht attraktiv. So sind viele der im Einsatz befindlichen Datenbanksysteme (DBMS - data base management system) mehr durch praktische Belange und nachfolgende Standardisierungen geprägt, als daß sie eine Implementation auf theoretischer Grundlage darstellen.

Eines der ersten Datenbanksysteme, das System IMS, war Mitte der sechziger Jahre einsatzbereit. Es basiert auf einem hierarchischen Datenmodell. Solche Datenmodelle zeichnen sich dadurch aus, daß Objekte in Klassen unterteilt werden, die dann in weitere Klassen unterteilt werden, bis schließlich unteilbare Objekte, die je nach Anwendungsfall definiert werden, die unterste Ebene bilden. So lassen sich z.B. "Getränke" in "alkoholische" und "alkoholfreie" Getränke unterteilen. "Alkoholfreie Getränke" können wiederum unterteilt werden in "Getränke auf Milchbasis", "Säfte", "kohlensäurehaltige Getränke", "Getränke mit anregenden Stoffen" und "sonstige". "Getränke mit anregenden Stoffen" sind z.B. Kaffee, schwarzer Tee usw. bis zu den einzelnen Produkten. Eine sinnvolle Fragestellung könnte dann darin bestehen, alle Getränke zu ermitteln, die sich für ein Kinderfest eignen.

Unser Beispiel, unwesentlich erweitert, liefert das folgende Bild, wobei die Hierarchieebenen durch Einrückungen dargestellt werden:

```
Getränke
    alkoholische Getränke
    alkoholfreie Getränke
        Getränke auf Milchbasis
            Vollmilch
            Fruchtmilch
                Banane
                Himbeere
                Aprikose
            H-Milch
        Säfte
        kohlensäurehaltige Getränke
        Getränke mit anregenden Stoffen
        sonstige Getränke
```

Die Anwendung hierarchischer Datenbanksysteme führte 1969 zu einem Standardisierungsvorschlag, den eine Gruppe (DBTG - data base task group) der Konferenz über Datenbanksprachen (CODASYL - conference on data system languages) unterbreitete. Nach mehreren Veränderungen und Erweiterungen wurde dieser Standard vielen Herstellern in den siebziger Jahren zur Grundlage ihrer Implementationen.

Ein weiteres Datenmodell für Datenbanken ist das Netzwerkmodell. Es wird in einigen Veröffentlichungen gar nicht erwähnt bzw. als eine Erweiterung des hierarchischen Datenmodells betrachtet. Der Kerngedanke besteht darin, daß, im Gegensatz zum hierarchischen Modell, eine Klasse mehrere Vorgänger haben kann. Versetzen wir uns in das oben angeführte Beispiel zurück, so gehört "Cola" sicher zu den "kohlesäurehaltigen Getränken". Genauso wäre "Cola" aber auch zu den

"Getränken mit anregenden Stoffen" zu rechnen. "Cola" muß deshalb von beiden Klassen aus erreichbar sein. Weitere Beispiele dieser Art lassen sich finden. Zeichnet man sich den entstandenen Graphen auf, so verwandelt sich die hierarchische Klassenordnung aus dem obigen Beispiel in einen vermaschten Graphen. Die Hierarchie ist hier zwar noch erhalten, doch es lassen sich Beispiele denken, in denen sich keine Hierarchie des gesamten Graphen mehr erkennen läßt.

1970 publizierte E. F. Codd seine Gedanken zum relationalen Datenmodell [Cod70]. 1975 bis 1977 wurde ein praxistaugliches relationales Datenbanksystem mit der Bezeichnung "R" und der zugehörigen Datenbanksprache SQL implementiert. In den Jahren danach erfolgte auch für die relationalen Datenbanken eine Standardisierung der zugehörigen Datenbanksprache (ANSI X3H2), die im wesentlichen die Prinzipien der Sprache SQL übernahm [Här87].

Die Existenz konkurrierender Datenmodelle und diesbezügliche Standardisierungen sowie die Notwendigkeit, immer größere Datenmengen verwalten zu können, hoben die Datenbankprobleme von den Problemen der Programmiersprachen ab. Ende der sechziger Jahre wurden Untersuchungen zu Datenbankproblemen als ernsthafte wissenschaftliche Probleme anerkannt. Eine theoretische Durchdringung setzte ein, die sich bis heute stürmisch entwickelt [Wed83]. Eine Diskussion aktueller theoretischer Entwicklungen auf diesem Gebiet findet man z. B. bei Härder [Här87].

Vergleichen wir das relationalen Datenmodell von Codd und dBASE. Ein Datenbankfile in dBASE ist einer Relation bei Codd gleichzusetzen. Ein Satz eines Datenbankfiles ist im relationalen Datenmodell ein Tupel.

Sprachen für relationale Datenbanksysteme zeigen einen Trend zum Deskriptiven und eine Abschwächung des prozeduralen Aspekts von Sprachen. Gelegentlich bezeichnet man Datenbanksprachen deshalb auch als Srachen der vierten Generation. Sprachen der dritten Generation sind danach die prozedurorientierten höheren Programmiersprachen (Pascal, Modula, C usw.). Sprachen der fünften Generation (Prolog) zeichnen sich durch Sprachelemente aus, die ein sog. logisches Programmieren gestatten.

Wir wollen die theoretischen Grundlagen der Datenbanksysteme in diesem Buch nicht weiter betrachten. Wir verzichten ebenfalls auf eine theoretische Interpretation von dBASE und wenden uns technischen, technologischen und anwendungsorientierten Problemen von dBASE zu. Die vorangegangenen Ausführungen dienen uns dabei als Orientierungshilfe für die Einordnung von dBASE in die Welt der Datenbanksysteme.

2.2. Der Wert einer Datenbank

Wir stellen diese Betrachtungen an den Anfang unserer Untersuchungen, um die Bedeutungen von Datenbanken in allen Stadien ihrer Exi-

stenz (Entwurf, Implementation, Datenerfassung, Datenauswertung, Datenpflege usw.) zu betonen. Wir folgen dabei den Überlegungen von Zehnder [Zeh83]. Nehmen wir an, daß ein Informationssystem für eine große Bibliothek zu erstellen sei. Drei wesentliche Komplexe bestimmen die Kosten, wobei wir von der konkreten Währung abstrahieren können.

1. Die Anschaffung des Rechners einschließlich Betriebssystem und Datenbanksystem kostet ca. 2 Mill. Es wird eine Lebenszeit von höchstens 10 Jahren geschätzt.

2. Die zugehörigen Anwenderprogramme wurden in einer Zeit von 5 Jahren entwickelt und kosteten ca. 6 Mill. Rechnet man mit einer möglichen Übernahme der Programme auf die nachfolgende Hardware, so erscheint eine Lebenszeit von 15 bis 20 Jahren real.

3. Die Daten selbst werden fortlaufend von Mitarbeitern der Bibliothek eingegeben, wofür ca. 3 Mill. im Jahr geschätzt werden. Bedenkt man, daß ein Bibliothekskatalog praktisch für immer besteht, so erscheint eine Lebenszeit von wenigstens 50 Jahren als durchaus angemessen. Rechnen wir eine Amortisationszeit von 10 Jahren, so ist ein Wert von ca. 30 Mill. dafür anzusetzen.

Fassen wir diese Angaben in einer Tafel zusammen:

Gegenstand	Lebenszeit (in Jahren)	relativer Wert
Rechner mit Betriebssystem und Datenbanksystem	<= 10	1
Anwendersoftware	<= 20	3
Datenbasis	>= 50	15

Es ist seit vielen Jahren bekannt, daß für ein rechnergestütztes Projekt ca. 70 bis 90% der Gesamtkosten für die Software aufgewendet werden müssen. Bedenkt man, daß mit dem Rechner ein Betriebssystem und ein Datenbanksystem gekauft wurden, so überraschen die Werte in den ersten beiden Zeilen nicht. Insgesamt ziehen wir aus dieser Betrachtung die folgenden Schlußfolgerungen:

1. Die Anwendersoftware muß unbedingt auf die nachfolgenden Rechner transportierbar und in die neuen Betriebssysteme einbettbar sein. Für die Implementation der Anwenderprogramme kommen damit nur gängige und zukunftsorientierte Sprachen in Frage.

2. Die Logik der Datenbasis - hier der Bibliothekskatalog - muß wohldurchdacht sein. Sie darf nicht von einem exotischen, gegenwärtig gerade verfügbaren Datenbanksystem abhängen. Sie muß flexibel in ihrem Inhalt und ihrer Struktur sein.

3. Die Daten müssen mit äußerster Sorgfalt erfaßt werden, da sich Erfassungsfehler bis ans Ende der Existenz der Datenbasis (unbemerkt) auswirken können. (Ein falsch geschriebener Name wäre dabei das kleinere Übel.)

2.3. Datenbanken in dBASE

dBASE ist ein relationales Datenbanksystem, d.h., es basiert auf dem relationalen Datenmodell. Alle Informationen über ein Objekt sind, in grober Vereinfachung, eine Zeile in einer Tabelle. Einen Eintrag über ein Objekt nennen wir Datensatz (record) oder einfach Satz. Die Spalten der Tabelle kennzeichnen die einzelnen Fakten, aus denen sich die Objekte zusammensetzen. Ein Element einer Spalte und einer Zeile ist ein Feld. Ein Datensatz setzt sich also aus Feldern zusammen. Alle Datensätze zusammen bilden eine Tabelle, die in Form einer Datei (eines Files) gespeichert wird. Ein derartiges File nennen wir Datenbankfile (im Unterschied zur Datenbank, zu der meist mehrere Files gehören). Somit ist dBASE für Probleme geeignet, die sich in Tabellenform darstellen lassen.

Betrachten wir eine Tabelle, wie sie in konventioneller Form auf einem Blatt Papier dargestellt wird, z. B. eine Liste beliebiger Produkte. Jede Spalte wird durch eine Überschrift bezeichnet, z.B.

```
Artikel    Anzahl    Preis    Lieferdatum    Bemerkungen
```

Diese Bezeichnungen gestatten einen eindeutigen Bezug auf eine Spalte. Wählen wir dazu eine bestimmte Zeile (Datensatz) aus, so bezeichnen die Spaltenüberschriften bezüglich dieser Zeile jeweils ein Feld des Datensatzes. Diese Eigenschaft werden wir nutzen, um uns auf den Inhalt eines Feldes in einem Datensatz zu beziehen. Die Spalten erhalten Namen, und ein Dateizeiger verweist auf den aktuellen Satz.

Offenbar benötigt man unterschiedlich breite Spalten. Unter "Artikel" wird ein kurzer Text stehen von vielleicht 20 oder 25 Zeichen. Für die Anzahl reichen 5 Zeichen, ein Datum benötigt 8 Zeichen usw. Genauso, wie wir den für eine Spalte erforderlichen Platz auf dem Papier freihalten, müssen wir auch dBASE die maximale Breite einer Spalte (eines Feldes) angeben.

Die Angaben in der Spalte "Artikel" und in der Spalte "Anzahl" unterscheiden sich aber nicht nur durch ihre Breite. Während "Artikel" eine Folge von Zeichen, ein String, ist, kann z.B. aus

```
Anzahl * Preis
```

der Gesamtpreis berechnet werden. "Anzahl" und "Preis" sind numerische Werte, d.h. Angaben, mit denen Berechnungen durchgeführt werden. Wir bezeichnen solche Felder als numerische Felder, während Felder, in denen nur Zeichenketten eingetragen werden, String-Felder sind. Aber auch "Anzahl" und "Preis" unterscheiden sich voneinander. "An-

zahl" ist ein ganzzahliger Wert, während Preise mit 2 Stellen nach dem Komma notiert werden oder ggf. auch mit drei Stellen nach dem Komma, z.B. für Erzeugnisse, die einen geringen Preis haben und in großen Stückzahlen verkauft werden (Steine, Ziegel usw.) oder falls bei großen Objekten in Millionen Mark gerechnet wird. Numerische Felder können Dezimalstellen haben. Ihre Breite bestimmt sich aus der Anzahl der Ziffer plus eins für den Dezimalpunkt, falls vorhanden.

"Lieferdatum" enthält auch Ziffern, unterscheidet sich aber dennoch von numerischen Feldern, da mit einem Datum keine numerischen Operationen durchgeführt werden können. Es ist zwar sinnvoll, zu einem Datum z.B. 50 Tage zu addieren, aber die Addition eines Datums mit einem anderen Datum oder die Multiplikation eines Datums mit einer Anzahl ergeben keinen Sinn. Wegen dieser letzten Eigenschaft wurden Datumsangaben in dBASE II als Zeichenketten behandelt. dBASE III berücksichtigt einige Besonderheiten des Datums gegenüber Zeichenketten und gestattet daher Felder vom Typ Datum (date). Um Fehler bei der Datenerfassung und bei den Operationen in der Datenbank zu reduzieren, wird jeder Spalte, und damit jedem Feld, ein Typ zugeordnet. dBASE III kennt die Typen String (character), numerisch (numeric), Datum (date), logisch (logic) und Memo (memo). Die beiden zuletzt genannten Typen werden wir später betrachten.

Nachdem wir die Spalten näher untersucht haben, kehren wir zur Auswahl der Zeile in einer Tabelle zurück. Sie erfolgt in der Praxis auf zwei Wegen. Im ersten Fall wissen wir, die wievielte Zeile wir suchen, etwa die erste oder die letzte Zeile. Viel häufiger suchen wir aber nach einem bestimmten Erzeugnis. Wir wollen beispielweise wissen, wieviel eine Kurbelwelle kostet, d.h., wir suchen in der Tabelle eine Zeile, in der unter "Artikel" "Kurbelwelle" eingetragen ist, und sehen dann in dieser Zeile unter "Preis" nach. Was heißt das für dBASE? Eine Tabelle ist in einem File gespeichert. Die Datensätze werden, beginnend mit eins, durchnumeriert. Jeder Satz kann über seine Nummer erreicht werden. Im zweiten Fall muß im File ein Feld "Artikel" mit dem Inhalt "Kurbelwelle" gesucht werden. In diesen Fall erfolgt die Adressierung des Datensatzes nicht durch eine Nummer oder einen Namen, sondern durch den Inhalt eines Feldes. Diese Art der Adressierung über den Inhalt eines Objektes nennt man auch assoziative Adressierung. Für die assoziative Adressierung muß jedes Objekt sich selbst identifizieren, d.h., es muß im Datensatz ein Feld geben, das diesen Satz eindeutig kennzeichnet. Häufig wird verlangt, daß die Fakten, die ein Objekt beschreiben, keine Redundanz aufweisen oder noch schärfer, daß ein Satz keine Informationen enthält, die aus anderen Feldern gewonnen werden können. Wir werden diese Forderung in den folgenden Abschnitten genauer untersuchen.

dBASE besitzt für jedes Datenbankfile einen Dateizeiger. Er verweist auf eine Zeile in der Tabelle. Dieser ausgewählte Datensatz heißt aktueller Datensatz. Der Dateizeiger wird für den Nutzer als Nummer des aktuellen Datensatzes sichtbar (Funktion RECNO()).

2.4. dBASE II oder dBASE III?

Diese Frage muß eindeutig zugunsten von dBASE III beantwortet werden. Der 16-Bit-Rechner ist gegenwärtig der Arbeitsplatzcomputer für kleine und mittlere Betriebe bzw. Probleme. Leistungsfähigkeit, Preis und persönliche Verfügbarkeit scheinen hier ein Optimum gefunden zu haben. dBASE III ist auf diese Rechnerklasse abgestimmt, dBASE II nicht. dBASE III besitzt außerdem vorwärtsweisende Eigenschaften und Ansätze, die in dBASE II nicht vorhanden sind. Im Sinne einer Orientierung nach vorn sollte man sich deshalb mit dBASE III beschäftigen.

Wir sind uns dabei sehr wohl der Tatsache bewußt, daß viele Anwender Datenbanken in dBASE II besitzen und diese weiter nutzen möchten. Ein Übergang von dBASE II zu dBASE III ist möglich (Abschn. 17.4). Ohne auf Details einzugehen, können die Vorzüge von dBASE III gegenüber seinem Vorgänger wie folgt charakterisiert werden: dBASE III bedient einen größeren Speicher, läßt größere Datenbankfiles zu, hat wesentlich höhere Grenzwerte, unterstützt die Arbeit im Direktmodus besser, hat einen erweiterten Satz von Standardfunktionen, eine erweiterte Menge von Auswahlbedingungen, die durch ein Konfigurationsfile bei jedem Start von dBASE für den entsprechenden Anwendungsfall gesetzt werden können, unterstützt den Typ Datum und reduziert den Nachteil von Datensätzen konstanter Länge durch den Typ Memo.

Außerdem existieren Nachfolgesysteme, die auf dBASE III aufbauen und weitere Eigenschaften der modernen 16-Bit-Technik unterstützen (Abschn. 20).

2.5. Datenbanken und Files

Aus dem Abschnitt 2.3 wissen wir, daß die eigentlichen Daten, die sogenannte Datenbasis, in einem oder mehreren Datenbankfiles gespeichert werden. Jeder Filename besteht aus acht Zeichen, die gewöhnlich vom Nutzer gewählt werden, und einer Erweiterung (extension). Diese Erweiterung des Filenamens sagt etwas über den Typ des Files aus. In dBASE III existieren feste Bezeichnungen für die verwendeten Filetypen. Tafel 2.1 gibt einen Überblick über die von dBASE II und dBASE III verwendeten Filetypen.

Datenbankfiles kennen wir bereits. Sie enthalten die eigentlichen Daten und bilden die Datenbasis.

Memo-Files sind Hilfsfiles zu einem Datenbankfile. Sie können einerseits nur mit diesem zusammen verwendet werden und müssen andererseits vorhanden sein, wenn ein Datenbankfile Informationen in Memo-Feldern enthält.

Indexfiles geben eine logische Ordnung in einem Datenbankfile wieder. Dem Begriff der logischen Ordnung steht die physische Anordnung der Sätze im Datenbankfile gegenüber, d.h. die Reihenfolge, in

der die Datensätze abgespeichert sind. Eine logische Ordnung wird durch die Betrachtung eines Datenbankfiles nach verschiedenen Ordnungskriterien festgelegt. So kann eine Produktliste nach Erzeugnissen geordnet betrachtet werden oder nach dem Hersteller, dem Preis usw. Das wird erreicht, indem das Datenbankfile mit einer entsprechenden Indexdatei in Benutzung genommen wird, die vorher erzeugt wurde (Abschn. 8.2).

Tafel 2.1. Filetypen in dBASE II und dBASE III

Bezeichnung des Files	Filetyp in dBASE III	Filetyp in dBASE II
Datenbankfile	.dbf	.dbf
File für Memo-Texte	.dbt	---
Indexfile	.ndx	.ndx
Memory-Files	.mem	.mem
Kommandofiles	.prg	.cmd
Formatfiles	.fmt	.fmt
Format von Adreßaufklebern	.lbl	---
Berichtsformate	.frm	.frm
Textfiles	.txt	.txt

Kommandofiles enthalten Folgen von dBASE-Kommandos. Sie sind als normale Textfiles mit dem dBASE-internen Editor (MODIFY COMMAND) oder einem anderen Editor (Abschn. 12) erstellt worden (Abschn. 11).

Memory-Files dienen der Speicherung von maximal 256 Memory-Variablen. Sie werden während der Arbeit mit dBASE erzeugt (SAVE) und bewahren den Zustand des Memory-Bereichs (oder Teile davon) für spätere Verwendungen auf (RESTORE).

Formatfiles enthalten Information über den Aufbau von Bildschirmmasken für die Datenerfassung, Korrektur und Ausgabe. Der Inhalt eines Formatfiles ist mit dem Schema eines Formulars oder einer Karteikarte vergleichbar (Abschn. 13.2).

Label-Files bestimmen Form und Inhalt von Briefadressen (Abschn. 16).

Berichtsformate (Report-Files) bestimmen Form und Inhalt von Berichten (Abschn. 10).

Textfiles dienen u.a. als Schnittstellen für den Datenaustausch von dBASE mit anderen Programmen. Enthalten diese Textfiles Sonderzeichen, so ist darauf zu achten, daß sie in den verschiedenen Systemen geeignet verarbeitet werden. Probleme können schon durch die unterschiedliche Behandlung des Tabulatorzeichens und der Umlaute entstehen. dBASE nimmt z.B. auf Textfiles Bezug, wenn in COPY oder APPEND das Schlüsselwort SDF (standard data format) enthalten ist (Abschn. 6.1.3).

2.6. Anforderungen und Grenzen

dBASE III stellt folgende Anforderungen an die Hardware:

1. einen IBM-PC-kompatiblen Rechner mit
   - 256 KByte-RAM,
   - einem MS-DOS-Betriebssystem der Version 2.0 oder höher,
   - zwei Floppy-Laufwerken mit einer Mindestkapazität von 360 KByte, d.h. 40 Spuren doppelseitig (besser eine Festplatte von 20 MByte oder mehr) und,
2. je nach Bedarf, einen Drucker.

Tafel 2.2 liefert eine Zusammenstellung der Grenzen von dBASE II und dBASE III so, wie sie durch das Softwareprodukt standardmäßig gegeben sind. Einige Grenzen können durch Angaben im Konfigurationsfile (Abschn. 17.2) verändert werden. Selbstverständlich setzen die Hardware und das Betriebssystem ebenfalls Grenzen. So nutzt eine theoretisch mögliche Anzahl von Datensätzen je Datenbankfile von einer Billion wenig, wenn nur Diskettenlaufwerke für 360 KByte zur Verfügung stehen.

Es ist hier zu bemerken, daß dBASE bei Verletzung der angegebenen Grenzen sehr unterschiedlich reagiert. Die Einhaltung einiger Grenzen wird nicht getestet. Die Wirkung ist dann undefiniert. So wurde z.B. die Erfahrung mit dBASE II, Version 2.4, gemacht, daß Indexausdrücke mit mehr als 99 Zeichen sehr wohl möglich sind, die Reaktion von dBASE aber undefiniert ist. In anderen Fällen, z.B. bei Verletzung der Formate, werden die Angaben zwar berücksichtigt, es können aber verstümmelte Werte entstehen. In diesen Fällen wird ohne Fehlermeldung weitergearbeitet. Der Nutzer sollte deshalb selbst auf die Einhaltung dieser Grenzen achten und die Einhaltung privat vorgegebener Grenzen (Formate) ggf. selbst testen.

2.7. Wie groß kann eine Datenbank sein?

Die Länge eines Datensatzes berechnet sich aus der Summe der Längen der einzelnen Felder plus 1. Dieses zusätzliche Byte wird für die Löschmarkierung benötigt. Der benötigte Speicherplatz für alle Datensätze ergibt sich dann durch Multiplikation. Das Datenbankfile ist jedoch etwas länger. Es enthält in einem Kopf Informationen über die Felder, ihren Namen, Typ und Länge.

Die Strukturen der Datenbankfiles von dBASE II und dBASE III sind unterschiedlich. Deshalb können dBASE II-Datenbankfiles nicht ohne weiteres von dBASE III verarbeitet werden. dBASE III arbeitet mit einer variablen Kopflänge. Genauere Angaben dazu sind im Abschnitt 19.1 zu finden.

Uns geht es hier um eine Abschätzung der benötigten Plattenkapazität für eine Datenbank. Wir bringen deshalb für den Kopf des Datenbankfiles 500 bis 1000 Byte in Anzatz, was bei großen Datenbanken vernachlässigt werden kann.

Tabelle 2.2 Grenzen von dBASE II und dBASE III

Gruppe/Eigenschaft	dBASE III	dBASE II
Datenbankfiles		
Anzahl der Datensätze	<= 10^{12}	<= 65535
Bytes je File	<= $2*10^{12}$	
Felder je Datensatz	<= 128	<= 32
Bytes je Datensatz	<= 4000	<= 1000
gleichzeitig aktive .dbf-Files	<= 10	<= 2
Fileoperationen		
gleichzeitig eröffnete Files	<= 15	<= 16
aktive Indexfiles je Datenbankfile	<= 7	<= 5
aktive Formatfiles je Datenbankfile	<= 1	<= 1
Feldtypen		
Zeichen je String-Feld	<= 254	<= 254
Bytes je Datumsfeld	8	—
Bytes je numerisches Feld	<= 19	<= 10
Bytes je logisches Feld	1	1
Bytes je Memo-Feld im .dbf-File	10	—
Bytes je Memo-Feld im .dbt-File	4096	—
Memory-Variable		
Anz. der aktiven Memory-Var.	<= 256	<= 64
Bytes für Memory-Var. gesamt	<= 6000	<= 1536
Numerische Eigenschaften		
Anzahl der Standardfunktionen	37	18
kleinste positive Zahl	10^{-307}	10^{-63}
größte Zahl	10^{+308}	$1.8*10^{+63}$
Rechengenauigkeit	15 Stellen	10 Stellen
Anz. der Ausdrücke in SUM	> 32	<= 5
Sonstiges		
Länge eines Indexschlüssels	<= 100	<= 98
Kopf eines Berichtes	<= 4*60	<= 254
Felder im Bericht	<= 24	<= 24
Breite eines Berichts	<= 999 Zeichen	—

Für einen Zeitschriftenkatalog für 1000 Zeitschriften, wobei jeder Datensatz 500 Byte benötigt, ergibt sich z.B. ein Plattenbedarf von ca. 500 KByte. Das läßt sich auf einer Diskette, bei der 80 Spuren doppelseitig beschrieben sind, speichern (Kapazität 720 KByte). Allerdings hat man dann noch keine Indexfiles, keine Kommandofiles, keine Formatfiles usw. Die Größe dieser Files läßt sich schlecht abschätzen. Sie liegt nach Erfahrungswerten zwischen 20% und 150%, bezogen auf die Länge des Datenbankfiles. Die sehr unterschiedlichen Größen sind durch die Anzahl der anderen, für die Datenbank benötig-

ten Files bedingt. Je komfortabler das Anwendersystem und seine Nutzeroberflächen, desto mehr Speicherplatz muß man dafür veranschlagen. So sind Indexfiles i. allg. erheblich kürzer als das Datenbankfile, zu dem sie gehören. Ihre Größe hängt wesentlich von der Länge des Indexschlüssels ab. Legt man zu einem Datenbankfile mehrere Indexdateien mit langem Schlüssel an, so kann im realen Fall eine Indexdatei bereits mehr Platz beanspruchen als das Datenbankfile selbst. Aus Gründen, die wir später noch betrachten werden, muß ein Datenbankfile gleichzeitig mit seinen Indexfiles on-line sein können. Für die Arbeit ohne Festplatte heißt das: Datenbankfile und alle Indexfiles müssen auf zwei, drei oder vier Disketten Platz finden, je nachdem, wieviel Laufwerke vorhanden sind.

Files können nicht über mehrere Disketten gehen! Bei 360 KByte-Laufwerken wird diese Grenze bei realen Projekten sehr bald erreicht. Eine Orientierung auf Festplatten ist zwar nicht notwendig, aber grundsätzlich zu empfehlen.

Stellen wir noch einige Hochrechnungen für Datenbankfiles an. Die Beschreibung einer Person, z.B. für den Briefverkehr oder für die Organisation einer Tagung, läßt sich in 150 bis 200 Byte unterbringen. 5 Sätze benötigen dann 1 KByte. Auf einer Diskette mit 360 KByte Speicherkapazität sind das Angaben zu 5 * 360 = 1800 Personen. Das ist sicher für viele Belange mehr als ausreichend.

Eine Verwaltung von Datenträgern mit dBASE benötigt 65 Byte je Datensatz, das sind ca. 15 Datensätze je 1 KByte. Auf einer Diskette mit 360 KByte läßt sich ein Datenbankfile für 15 * 360 = 5400 Datenträger speichern. Auch das scheint keine wirkliche Grenze zu sein. Wir stellen also fest, daß praxisrelevante Datenbanken kleiner und mittlerer Größe von ihrem Speicherplatzbedarf her auf 8- und 16-Bit-Rechnern realisierbar sind. Mit der Verwendung von Festplatten kommen wir in den Bereich mittlerer Datenbanken.

Es ist aber nicht ratsam, einen Datenträger mit einem Datenbankfile zu füllen. Wir empfehlen folgende Faustregel: Ergibt die Hochrechnung für eine komplette Datenbank einen Speicherplatzbedarf von mehr als 50% der verfügbaren On-line-Kapazität, so ist die Realisierbarkeit auf dieser Hardware zumindest fraglich. Datenbanken unterliegen einem natürlichen Wachstum, welches verschiedene Ursachen haben kann. Das erfordert freie Speicherkapaziät.

2.8. Typische Aufgabenstellungen für ein Datenbanksystem

Typische Operationen über Datenbanken beziehen sich auf das gesamte Datenbankfile bzw. auf große Teile davon. Nehmen wir an, es liege eine Datenbank vor, in der Erzeugnisse gespeichert sind. Sinnvolle Aufgabestellungen wären dann:

Sortieren der Datensätze in alphabetischer Ordnung nach den Erzeugnissen.

In dieser sortierten Datenbank erzeugen wir eine logische

Ordnung (Indexdatei), die die Erzeugnisse nach ihren Werten (Preisen) sortiert und bei gleichen Preisen die alphabetische Reihenfolge einhält.

Wir lassen uns eine alphabetisch geordnete Liste all der Erzeugnisse ausgeben, die vor dem 1.1.88 geliefert wurden.

Man drucke eine Inventurliste für alle Erzeugnisse mit einem Gesamtwert über 100 Mark.

Man ermittle den Gesamtwert aller erfaßten Erzeugnisse.

Für andere Datenbanken wäre z.B. das Drucken von Briefadressen oder die Auslösung von Bestellungen beim Unterschreiten einer Mindestanzahl im Lager eine sinnvolle Aufgabenstellung. Auf jeden Fall sehen wir daraus, daß dBASE stupide Routinearbeiten erledigen kann, die uns heute noch von produktiver Arbeit abhalten und die wir zudem nur unvollständig ausführen können. dBASE liefert uns aber auch Zusammenstellungen und Berichte nach bestimmten Gesichtspunkten in einer Geschwindigkeit, die von Hand nie erreicht wird. dBASE unterstützt in dieser Hinsicht die Entscheidungsfindung in Leitungsprozessen.

Erstaunlich ist darüber hinaus, daß viele der oben formulierten Aufgaben mit einem einzigen dBASE-Kommando veranlaßt werden können. Das ist das Typische an Datenbanksprachen: die Komplexität ihrer Operationen. Wie lange würde man für die Formulierung der gleichen Algorithmen in Pascal oder Basic brauchen?

2.9. Hilfen bei der Arbeit mit dBASE

Zum dBASE-System gehört ein umfangreiches Help-File. Durch die Eingabe des Kommandos HELP oder durch Drücken der Funktionstaste F1 werden Texte auf dem Bildschirm ausgegeben, die Teile von dBASE auf unterschiedlichen Niveaus erläutern. Wird nur HELP eingegeben, so gelangt man über mehrere Menübilder bis zu dem gesuchten Hilfstext.

Außerdem besteht die Möglichkeit, nach HELP ein Schlüsselwort anzugeben. Zulässige Schlüsselwörter sind die einleitenden Wörter aller Kommandos und die Schlüsselwörter FUNCTION und EXPRESSION. Das nach HELP angegebene Schlüsselwort bestimmt, auf welchem Niveau die Hilfe ansetzt. Je genauer man nach Hilfe fragen kann, desto eher erhält man sie. Es wird kein Schlüsselwort zurückgewiesen. Ein unbekanntes Schlüsselwort wird ignoriert. Das Kommando wirkt dann wie HELP ohne Schlüsselwort. Es erscheint das Bild 2.1 auf dem Bildschirm. Die Auswahlmöglichkeit "1 - Erste Schritte" ist invers dargestellt, was wir durch die Schraffur andeuten wollen. Diesen Balkenkursor bewegt man durch die Kursorpositionierungstasten (↑, ↓) und wählt durch Drücken der Enter-Taste, hier durch ↵ angedeutet, das nächste Hilfsmenü.

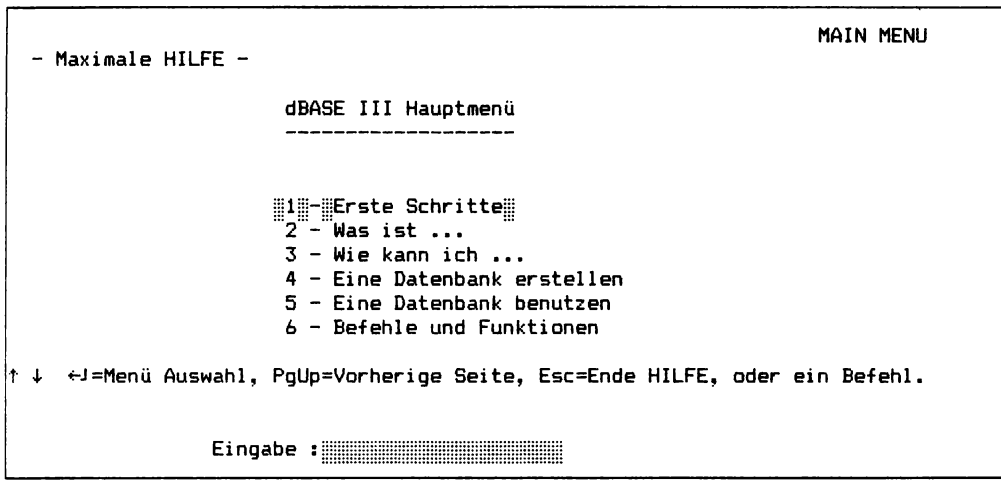


Bild 2.1. HELP-Menü

Eine neue Art der Hilfestellung im Vergleich zu dBASE II wird durch das ASSIST-Kommando angeboten. Hierbei wird ein Kommando im Dialog aufgebaut. ASSIST schafft dazu im Direktmodus eine Arbeitsumgebung, die Hilfstexte enthält und durch weitere Angebote die Anzahl der einzugebenden Zeichen reduziert. Wir erläutern das an einem Beispiel. Nach Eingabe von ASSIST wird ein Bild aufgebaut, das die Benutzung der Kursortasten beschreibt. Mit **Enter** kommen wir zum nächsten Menü, das Bild 2.2 wiedergibt.

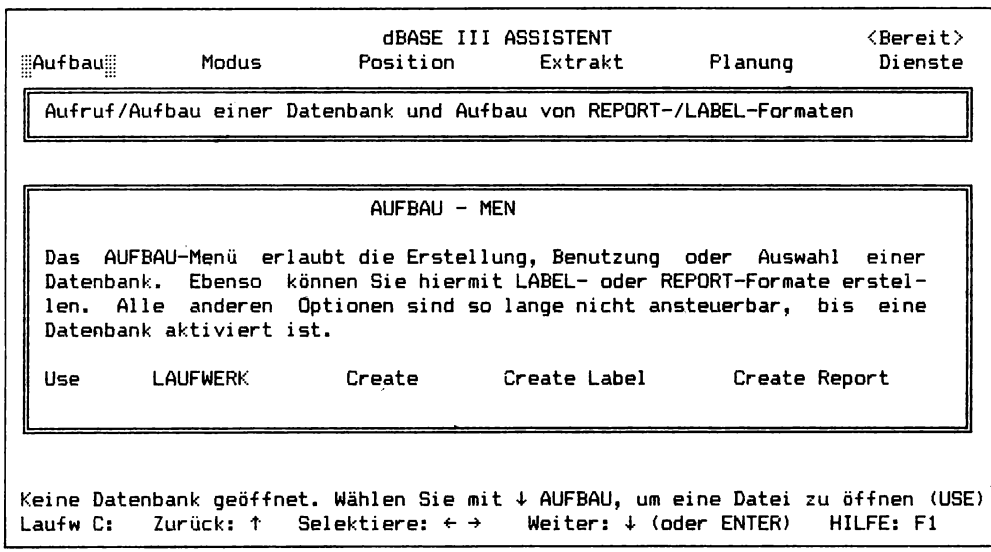


Bild 2.2. ASSIST: Auswahl einer Kommandogruppe

Im oberen Kästchen von Bild 2.2 erscheint eine kurze Information zu der Kommandogruppe, die durch den inversen Balkenkursor der zweiten Bildzeile ausgewählt wird. Hier ist es der Aufbau von Dateien. Nach einiger Zeit erscheint dann das untere Kästchen mit näheren Erläuterungen. Wählt man durch die Kursortasten eine andere Befehlsgruppe, so verändern sich die Informationen entsprechend. Wir wählen in unserem Beispiel das erste Angebot, den Aufbau von Dateien, durch Drücken der **Enter**-Taste. Es erscheint Bild 2.3 mit Erläuterungen zum USE-Kommando. In der dritten Zeile von unten wird schrittweise das Kommando erstellt. Wir haben USE gewählt, also erscheint diese Zeichenfolge nach "Befehl:".

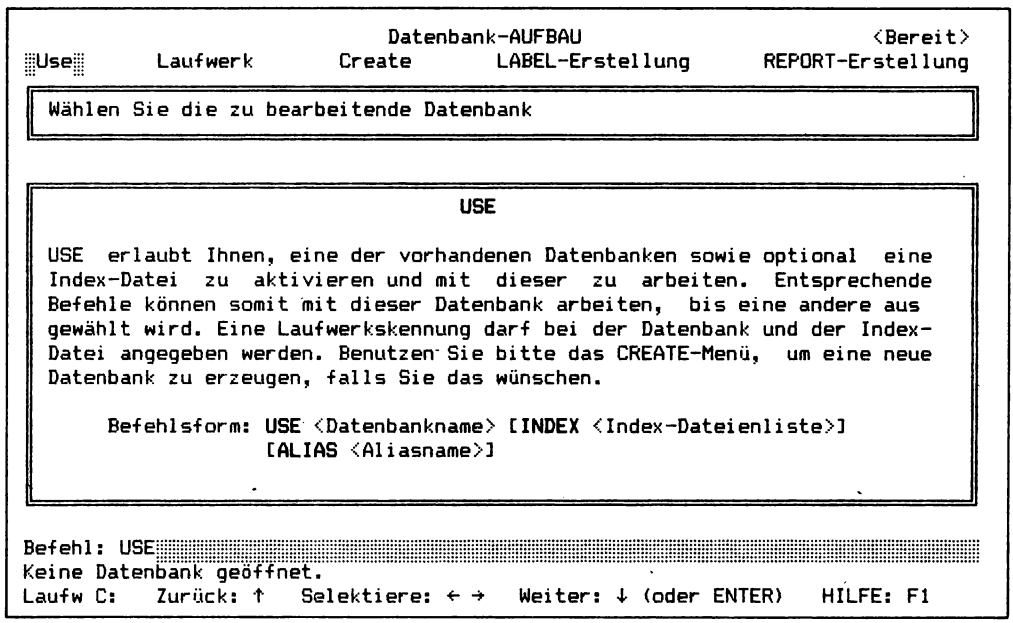


Bild 2.3. ASSIST: das USE-Kommando

Wir müssen jetzt ein Datenbankfile auswählen, welches wir benutzen wollen. Drückt man die **Enter**-Taste, so kann man zuerst das Gerät auswählen, auf dem das zu benutzende Datenbankfile steht. Danach werden alle Datenbankfiles auf diesem Gerät und im aktuellen Verzeichnis angezeigt. Bild 2.4 gibt diese Situation wieder. Jetzt steht ein Balkenkursor zur Verfügung, mit dem man über die Filenamen wandern kann. Steht dieser Kursor auf dem gewünschten Filenamen, so wird mit **Enter** dieses File ausgewählt. Das Kommando in der dritten Zeile von unten wird vervollständigt. Auf diese Weise können wir fast jedes Kommando im Dialog schrittweise erstellen. Man braucht also für die ersten Versuche in dBASE die Kommandos gar nicht im einzelnen zu kennen. Durch Drücken der Funktionstaste F1 können weitere Hilfen angefordert werden.

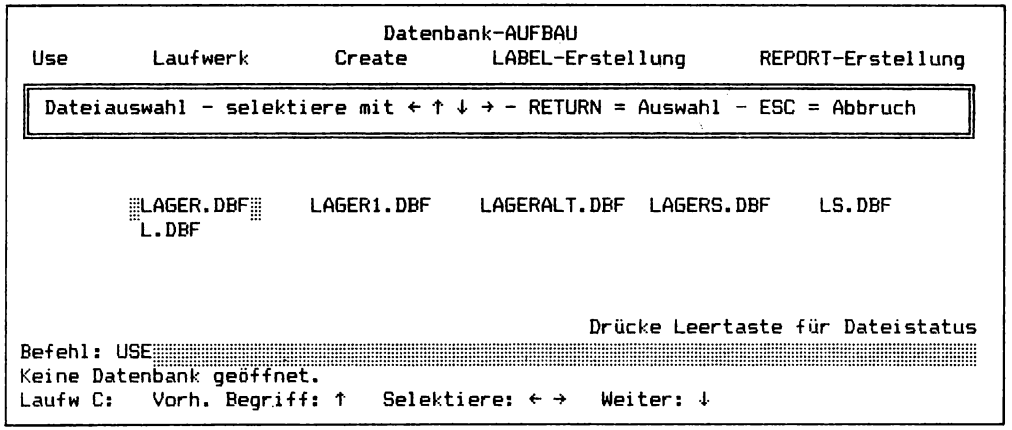


Bild 2.4. ASSIST: Auswahl eines Datenbankfiles

Zwei Bemerkungen sind anzufügen. Erstens sind über ASSIST nicht alle Kommandos in ihrer vollen Form erreichbar. Zweitens ist für den geübten Nutzer die Arbeit mit ASSIST zu langsam. Die Eingabe einer Kommandozeile ist wesentlich schneller als der ständige Wechsel der Menübilder und die Auswahl einzelner Komponenten durch Kursorbewegungen und **Enter**.

2.10. Einige praktische Hinweise

Wir wollen die bereits gewonnenen Erkenntnisse zusammenfassen und für den ersten Einstieg auf einige Dinge hinweisen, die beim Entwurf einer Datenbank beachtet werden sollten.

Bevor Sie an den Computer gehen, sollten Sie die Logik Ihrer Datenbank gründlich durchdenken: Wieviel Tabellen (Datenbankfiles) werden gebraucht? Was sind unteilbare Fakten, die dann zu Feldern werden? Wie werden daraus die gewünschten Informationen zusammengesetzt? Welche Abhängigkeiten bestehen zwischen den Datenbankfiles? Was soll mit den Daten gemacht werden, was sind die Ziele?

Die interne Struktur eines Datenbankfiles und die Form der Datenerfassung und der Berichte sind grundsätzlich verschieden. Für die Datenerfassung und für Berichte wird jeweils eine Nutzeroberfläche geschaffen, die den Bedürfnissen und Gewohnheiten der Nutzer angepaßt ist. So muß die Datenerfassung möglichst in einer Form erfolgen, die dem Erfasser angepaßt ist. Die eingegebenen Daten können direkt in einem Datenbankfile gespeichert werden, oder sie füllen nur einzelne Felder eines Datenbankfiles, oder sie werden auf mehrere Datenbankfiles verteilt. Filenamen, Feldnamen und interne Formate können, im Interesse des Nutzers, vor ihm verborgen werden. Durch die Verwendung von Kommando und Formatfiles lassen sich Nutzeroberflächen schaffen, die keine Kenntnisse über die dBASE-Kommandos voraussetzen. Die Er-

stellung solcher Oberflächen und ihre Wartung erfordern jedoch interne Kenntnisse über das konkrete Projekt und über dBASE als Sprache.

Für Berichte gilt sinngemäß das gleiche. Die Form eines Berichts hat mit der internen Form eines Datenbankfiles nichts zu tun. Wichtig ist, daß sich die Daten des Berichts aus den Angaben in der Datenbank ermitteln lassen. Bild 2.5 stellt diesen Sachverhalt schematisch dar.

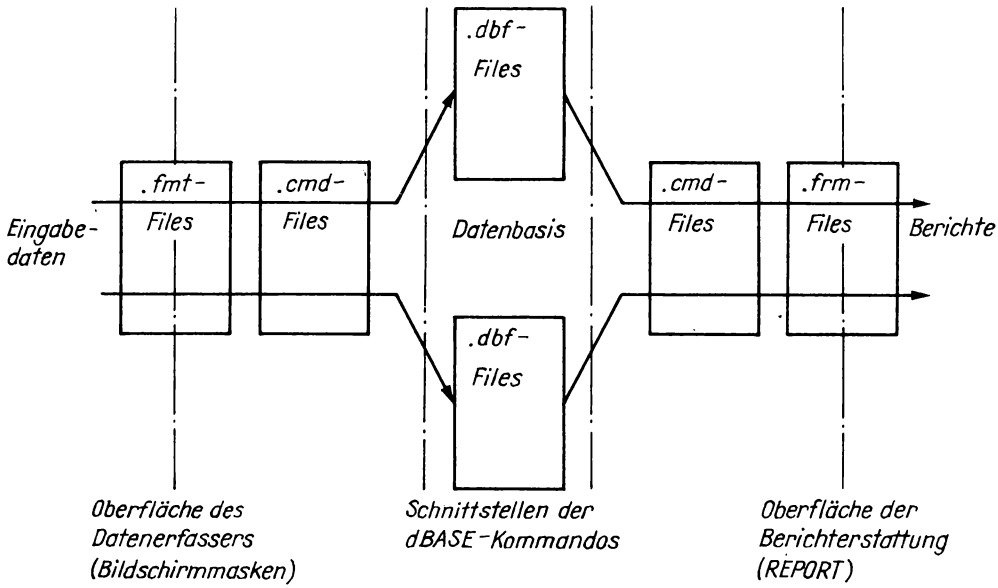


Bild 2.5. Nutzeroberflächen in dBASE

In einer Datenbank sollten keine redundanten Daten gespeichert werden! Benötigen Sie in einer Datenbank von einer Person den Namen, Vornamen, die Personenkennzahl und die Adresse, so dürfen Sie diese Angaben nur einmal erfassen. Wird in einer anderen Datenbank ebenfalls die Adresse benötigt, so ist auf das entsprechende, bereits vorhandene Datenbankfile zuzugreifen, das die Adresse enthält. Warum ist das so wichtig? Die Daten leben, sie sind Veränderungen unterworfen. Zieht z.B eine Person um, so ist die Adresse nur einmal zu ändern. Das ist nicht nur eine Frage der Bequemlichkeit, es erhöht auch die Konsistenz des Datenbestands wesentlich. Logische Fehler, wie eine falsche Adresse, sind sonst schwer aufzuspüren.

Ein Datenbankfile sollte nicht zu viele Felder enthalten. Es wird dadurch unübersichtlich und träge in der Bearbeitung. Eine Teilung in mehrere Datenbankfiles (vertikale Teilung) bringt oft noch den Vorteil mit sich, daß insgesamt Speicherplatz gespart wird. Das ist immer dann der Fall, wenn Gruppen von Sätzen in einigen Feldern die gleiche Information enthalten. So ist es z.B. nicht zweckmäßig, in ein Datenbankfile einer Lagerhaltung die Adressen aller Hersteller

aufzunehmen. Es ist sehr wahrscheinlich, daß mehrere Teile vom gleichen Hersteller bezogen werden. Die Adressen bilden deshalb ein extra Datenbankfile, welches mit dem Datenbankfile der Namen verbunden ist.

Werden Datenbankfiles zu lang, so ist eine horizontale Teilung möglich. Das Prinzip ist mit der Einteilung eines großen Lexikons in mehrere Bände vergleichbar. So können z.B. die Ersatzteile nach ihrer Bezeichnung geordnet werden. In das erste Datenbankfile kommen die Teile mit den Anfangsbuchstaben A bis J, in das zweite Datenbankfile die mit K bis M beginnenden usw.

3. Besichtigen einer Datenbank – die Kommandostruktur

Für diesen Abschnitt versetzen wir uns in die Lage eines Erstnutzers. Wir nehmen an, daß eine Datenbank auf unserem Rechner existiert. Wir wollen sie besichtigen und dabei grundlegende Prinzipien der Handhabung von dBASE kennenlernen. Zur Demonstration dieser Prinzipien benutzen wir das Kommando LIST. Es dient der Ausgabe verschiedener Informationen auf dem Bildschirm. Dazu sind vorerst keine weiteren Kenntnisse über das Betriebssystem und sonstige Randbedingungen erforderlich. Auf dem Rechner muß sich ein dBASE III-System befinden, welches in den Version 1.00 und 1.10 aus je vier Files besteht:

DBASE.EXE,
DBASE.OVL,
HELP.DBS und
ASSIST.HLP

Diese vier Files benötigen 321 (V. 1.00) bzw. 355 KByte (V. 1.10), sind also auf einer normalen Diskette (360 KByte) speicherbar. Wird keine Festplatte verwendet, so entstehen bei ernsthaften Anwendungen allerdings bald Speicherplatzschwierigkeiten. HELP.DBS und ASSIST.HLP sind nicht unbedingt erforderlich. Fehlen sie, so sind die Help-Texte und die Erläuterungen zum Assistenten nicht verfügbar.

3.1. Starten von dBASE

Nach dem Starten des Betriebssystems besichtigen wir mit dem Kommando

```
dir
```

das Verzeichnis (directory). Befinden sich die Files des dBASE-Systems in diesem Verzeichnis, so kann dBASE unmittelbar gestartet werden. Ansonsten sehen wir im ausgegebenen Verzeichnis weitere Unterverzeichnisse (subdirectories). Wir wählen das Verzeichnis als aktuelles aus (cd - change directory), welches die Files des dBASE-Systems enthält. Sei der Name dieses Verzeichnisses 'dbase3', so erreichen wir das durch Eingabe von

```
cd dbase3
```

auf der Kommandoebene von MS-DOS. Nach Eingabe von 'dbase' erscheint eine Anfangsmeldung von dBASE. Von diesem Augenblick an bewegen wir uns völlig im dBASE-System. Anforderungen an das dBASE-System werden durch Kommandos gestellt. dBASE kennzeichnet seine Bereitschaft, Kom-

mandos entgegenzunehmen, durch ein Anforderungszeichen (Prompt). Das Prompt von dBASE ist der Punkt '.'. Möchte man an dieser Stelle bereits einen Text zur Erläuterung des Systems auf dem Bildschirm sehen, so drückt man die Taste F1. Die Wirkung der Hilfen wurde im Abschnitt 2.9 beschrieben.

Ein Kommando ist eine Zeichenfolge, die eine bestimmte Aktion des Systems, also einen Algorithmus, bezeichnet. Gibt der Nutzer ein Kommando ein, so veranlaßt er damit die Ausführung dieser Aktion. Kommandos bestehen aus fest definierten Zeichenfolgen, sog. Schlüsselwörtern, und frei wählbaren Zeichenfolgen, die Bezeichnungen von Feldern, Variablen usw. darstellen. LIST ist ein solches Schlüsselwort. Zur besseren Unterscheidung von anderen Bezeichnern schreiben wir Schlüsselwörter groß. Auch in Kommandofiles sollte man Schlüsselwörter groß schreiben! Klein- und großgeschriebene Schlüsselwörter sind zwar identisch, die Lesbarkeit des Kommandofiles (Programms) erhöht sich aber dadurch wesentlich.

dBASE arbeitet interpretativ, d.h., ein eingegebenes Kommando wird analysiert, auf Korrektheit geprüft und sofort ausgeführt. Ist es vollständig ausgeführt, so kann das nächste Kommando eingegeben werden usw. Es ist die gleiche Vorgehensweise wie bei allen Interpretern, z.B. bei einem Basic-Interpreter. Typische Schritte, wie sie bei Übersetzern (Compilern) auftreten (Übersetzen, Linken, Laden), sind daher bei dBASE nicht erforderlich. Besichtigen wir also ein Datenbankfile.

3.2. Benutzen einer Datenbank

Zuerst verschaffen wir uns eine Information über die Datenbankfiles in unserem Filesystem. Das Kommando

```
LIST FILES
```

liefert uns

```
. LIST FILES
Datenbanken     # Sätze      Letze Änderung  Größe
AUSRUEST.DBF        59       31.07.87       15360
PRODUKTE.DBF        15       30.04.87         793
LAGER.DBF            9       21.10.87         799
  16922 BYTES in     3 Dateien.
```

Wir entscheiden uns für das Datenbankfile 'lager.dbf'. Dazu müssen wir dieses Datenbankfile in Benutzung nehmen:

```
USE lager
```

lager.dbf wird damit zum aktiven Datenbankfile. Die Fileerweiterung (extension) '.dbf' wird durch das dBASE-System selbständig ergänzt.

Da keine weiteren Files (Indexfiles, Formatfiles usw.) benötigt werden, ist das in diesem Fall auch unsere Datenbank. Wir werden später sehen, daß durch 'USE lager' zwei Files benutzt (eröffnet) werden: das File 'lager.dbf' und das zugehörige File für die Memo-Texte 'lager.dbt'.

3.3. Struktur eines Datenbankfiles

Die Struktur unseres Datenbankfiles (unserer Tabelle) erscheint durch das Kommando LIST STRUCTURE auf dem Bildschirm.

```
. LIST STRUCTURE
Datenbankstruktur       - C:LAGER.dbf
Anzahl der Datensätze   -         9
Letztes Änderungsdatum - 21.10.87
Feld   Feldname    Typ         Länge    Dez
    1  ARTIKEL     Zeichen        15
    2  ANZAHL      Numerisch       4
    3  PREIS       Numerisch       8      2
    4  MINIMUM     Numerisch       2
    5  LIEFERUNG   Datum           8
    6  HERSTELLER  Zeichen        12
    7  BEM         MEMO           10
** Gesamt **                      60
```

Wir sehen drei Felder vom Typ numerisch, wobei das Feld mit der Bezeichnung 'Preis' in seiner Gesamtbreite von 8 Zeichen zwei Dezimalstellen enthält. Zwei Felder vom Typ String (character, Zeichen) treten auf, die, genauso wie die numerischen Felder, eine Länge haben, die bei der Erzeugung des Datenbankfiles (CREATE) gewählt werden kann. Im Gegensatz dazu haben Felder der Typen Datum und Memo eine durch dBASE bestimmte Länge (8 bzw. 10 Zeichen). Der einzige Typ, der nicht vorkommt, ist der Typ logisch. Er hat ebenfalls eine durch dBASE vorgegebene Länge von einem Zeichen. Er wird selten benutzt, da oft inhaltsreichere Aussagen mit einem Zeichenfeld der Länge 1 getroffen werden können.

3.4. Inhalt einer Datenbank

Betrachten wir nun den Inhalt dieser Datenbank. Das Kommando LIST liefert uns diese Informationen:

```
. LIST
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER    BEM
      1  Flachschieber       112     2.40      15 29.02.88  C&C           MEMO
      2  Kurbelwelle I        16   360.00       3 12.05.88  Achse&Welle   MEMO
      3  Kurbelwelle II       12   325.00       3 12.05.88  Achse&Welle   MEMO
      4  Formschlauch         41     1.55       4 28.07.88  Gummi&Sohn    MEMO
      5  Kühlluftgehäuse       8    16.60       2 23.11.88  VEB Guß       MEMO
      6  Luftleitblech        14     5.40       2 10.11.88  Blech&Co      MEMO
      7  Kopfdichtungen      116     0.15      30 17.08.88  Dicht&Fest    MEMO
      8  Krümmerdichtung      88     0.14      30 18.09.88  Dicht&Fest    MEMO
      9  Fußdichtung         122     0.15      30 07.06.88  Dicht&Fest    MEMO
```

Wir sehen in einer Kopfzeile die Feldbezeichnungen und darunter, in den Spalten, die Inhalte der einzelnen Zeilen. Texte (Feldinhalte vom Typ Zeichen) werden linksbündig geschrieben, numerische Größen rechtsbündig. Die Inhalte der Memo-Felder werden nicht angezeigt. Das ist durchaus sinnvoll, da diese Texte lang sein können und dann die Tabellenform stören. Möchte man sie dennoch anzeigen, so muß das Feld, hier 'Bem', explizit aufgeführt werden.

Felder vom Typ Datum werden linksbündig und in deutscher Form angezeigt. In der ersten Spalte erscheint außerdem die Nummer des betreffenden Datensatzes. Durch das Schlüsselwort OFF kann diese Anzeige ausgeschaltet werden.

3.5. Gültigkeitsbereich eines Kommandos

Durch Eingabe des Kommandos LIST werden alle Datensätze ausgegeben, d.h., LIST bezieht sich auf alle Datensätze, oder wir sagen auch, der Gültigkeitsbereich von LIST besteht aus allen Datensätzen des Datenbankfiles. Offenbar sind auch andere Gültigkeitsbereiche für Kommandos sinnvoll. So ähnelt das Kommando DISPLAY dem Kommando LIST mit den Unterschieden, daß der Gültigkeitsbereich von DISPLAY der aktuelle Datensatz ist. Werden bei DISPLAY mehrere Datensätze ausgegeben, so stoppt die Bildschirmausgabe nach 20 Datensätzen. Bei LIST kann sie nur durch ^S (**Ctrl** und S drücken) gestoppt werden.

Außer den Standardannahmen für den Gültigkeitsbereich von Kommandos ist es auch sinnvoll, im Kommando Angaben über den Gültigkeitsbereich zuzulassen. Befinden wir uns mit dem Dateizeiger am Anfang der Datenbank, so kann die Anzeige der nächsten drei Sätze schon ausreichend sein, um einen Überblick über den Inhalt der Datenbank zu bekommen.

```
. LIST NEXT 3
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER    BEM
      1  Flachschieber       112     2.40      15 29.02.88  C&C           MEMO
      2  Kurbelwelle I        16   360.00       3 12.05.88  Achse&Welle   MEMO
      3  Kurbelwelle II       12   325.00       3 12.05.88  Achse&Welle   MEMO
```

Genauso kann man den Inhalt eines bestimmten Datensatzes anzeigen lassen, auch wenn der Dateizeiger nicht auf diesen Satz zeigt.

```
. LIST RECORD 3
 Satz # ARTIKEL         ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
      3 Kurbelwelle II      12   325.00       3 12.05.88 Achse&Welle  MEMO
```

Das Schlüsselwort ALL bestimmt alle Datensätze als Gültigkeitsbereich. So ist LIST ALL gleichwertig mit LIST.
Mögliche Angaben für einen Gültigkeitsbereich sind

ALL	für alle Datensätze
NEXT i	für die nächsten i Datensätze ab Dateizeiger
RECORD i	für den i-ten Datensatz.

Ein wenig formaler und dafür auch kürzer formulieren wir diesen Sachverhalt so:

```
<gb> ::= ALL | NEXT <anzahl> | RECORD <nummer>
```

<gb> steht für Gültigkeitsbereich. Die spitzen Klammern <> besagen, daß diese Stelle durch etwas zu ersetzen ist, was inhaltlich durch die eingeschlossene Zeichenfolge zum Ausdruck gebracht wird, also für <anzahl> eine Anzahl von Sätzen, für <nummer> die Nummer eines Datensatzes usw. Symbole in spitzen Klammern bezeichnet man auch als Metasysmbole.
Um ein gültiges Kommando zu erhalten, müssen Metasymbole so lange ersetzt werden, bis keine Metasymbole in der Zeichenkette mehr auftreten. Wodurch man <gb> ersetzen kann, gibt die obige Zeile an. Sie ist eine syntaktische Regel und für uns somit eine Ersetzungsregel.
Das Zeichen '::=' liest man als 'ist definiert durch'.
Der senkrechte Strich '|' trennt Alternativen voneinander.
Somit haben wir uns mit der Erklärung des Begriffs "Gültigkeitsbereich" gleichzeitig eine Form für die Notation der Syntax von dBASE-Kommandos erarbeitet und ihre Interpretation festgelegt.

3.6. Anzeigen ausgewählter Felder

Kehren wir zurück zur Besichtigung unserer Datenbank. Ist die Tabelle zu breit, d.h., sind zu viele Spalten vorhanden oder sind die Spalten zu breit, so entsteht der Wunsch, nur ausgewählte Felder anzuzeigen. Man kann das erreichen, indem man eine <feldliste> angibt. Eine Feldliste ist definiert durch eine Folge von Feldbezeichnungen, die durch Komma getrennt sind.

```
<feldliste> ::= <feld> {,<feld> ...}
```

Die geschweiften Klammern bezeichnen Teile, die beliebig oft wie-

derholt werden können.

```
. LIST NEXT 3 artikel, anzahl, preis, bem
 Satz #  artikel          anzahl    preis bem
      1  Flachschieber       112     2.40 Monatliche Lieferung.
      2  Kurbelwelle I        16   360.00 Neue Ausführung ohne Simmerring.
      3  Kurbelwelle II       12   325.00 Erfordert extra Simmerring.
```

Wir sehen jetzt, daß auch die Inhalte der Memo-Felder ausgegeben werden. Statt einer Feldliste kann auf der gleichen Position sogar eine Ausdrucksliste stehen.

<ausdr_liste> ::= <ausdruck> {,<ausdruck> ...}

Ein sinnvoller Ausdruck wäre in diesem Fall die Berechnung des Gesamtpreises, also 'preis * anzahl'. Übrigens kann man auch gleich die Gesamtsumme bilden (Funktion SUM).

```
. LIST artikel, preis, anzahl, preis*anzahl
 Satz #  artikel            preis anzahl  preis*anzahl
      1  Flachschieber        2.40    112       268.80
      2  Kurbelwelle I      360.00     16      5760.00
      3  Kurbelwelle II     325.00     12      3900.00
      4  Formschlauch         1.55     41        63.55
      5  Kühlluftgehäuse     16.60      8       132.80
      6  Luftleitblech        5.40     14        75.60
      7  Kopfdichtungen       0.15    116        17.40
      8  Krümmerdichtung      0.14     88        12.32
      9  Fußdichtung          0.15    122        18.30
. SUM(preis*anzahl)
      9 Sätze summiert
   (preis*anzahl)
         10248.77
```

3.7. Auswahlbedingungen

Liegt eine Datenbank vor, so lassen sich Zusammenstellungen nach bestimmten Gesichtspunkten vornehmen. Man formuliert eine Bedingung, nach der aus dem Gültigkeitsbereich (<gb>) des Kommandos all die Datensätze ausgewählt werden, auf die diese Bedingung zutrifft. Möchte man z.B. alle Angaben über Kurbelwellen ausgeben lassen, so lautet die Bedingung

```
artikel = "Kurbelwelle".
```

'Artikel' ist ein Feld vom Typ String. "Kurbelwelle" ist eine Zeichenkette und als solche ebenfalls vom Typ String. Verglichen

werden beide Zeichenketten so lange, bis eine der beiden Zeichenketten erschöpft ist. Waren dann alle verglichenen Zeichen gleich, so sind die Zeichenketten gleich, unabhängig von ihrer Länge und von irgendeinem Rest. Eine sinnvolle Anwendung besteht darin, sich alle Artikel ausgeben zu lassen, die mit "K" beginnen.

```
. LIST FOR artikel="K"
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
      2  Kurbelwelle I        16   360.00       3 12.05.88  Achse&Welle  MEMO
      3  Kurbelwelle II       12   325.00       3 12.05.88  Achse&Welle  MEMO
      5  Kühlluftgehäuse       8    16.60       2 23.11.88  VEB Guß      MEMO
      7  Kopfdichtungen      116     0.15      30 17.08.88  Dicht&Fest   MEMO
      8  Krümmerdichtung      88     0.14      30 18.09.88  Dicht&Fest   MEMO
```

Große und kleine Buchstaben sind beim Vergleich signifikant! Eine Auswahlbedingung

```
artikel="k"
```

trifft auf keinen Datensatz zu; denn kein Artikel beginnt mit einem kleinen "k".

Auch Kombinationen zwischen den angegebenen Kommandoformen sind möglich:

```
. GOTO TOP
. LIST NEXT 5 artikel, preis FOR (artikel="K") .AND. (anzahl<16)
Satz #  artikel           preis
      3  Kurbelwelle I    325.00
      5  Kühlluftgehäuse   16.60
```

Die Auswahlbedingung besteht in diesem Fall nicht nur aus einem Vergleich, sondern aus zwei Vergleichen, die durch den logischen Operator .AND. verknüpft sind. Operatoren werden in Punkte eingeschlossen, um sie von gewählten Bezeichnungen unterscheiden zu können.

3.8. Notation von Kommandos

Ein Kommando in dBASE beginnt mit einem Schlüsselwort oder einem speziellen Zeichen (?,@). Die Schlüsselwörter sind so gewählt, daß sie eindeutig die Kommandos unterscheiden. Ein Schlüsselwort muß angegeben werden. Die nachfolgenden Bestandteile können wahlweise hinzugefügt werden. Eckige Klammern ([...]) schließen Teile ein, die wahlweise angegeben werden können.

Die Reihenfolge der Angaben nach dem Schlüsselwort ist beliebig. Voraussetzung ist jedoch, daß die einzelnen Bestandteile voneinander

unterscheidbar sind. Deshalb empfiehlt es sich, keine Bezeichnungen zu verwenden, die mit Schlüsselwörtern von dBASE übereinstimmen. In der Notation der Syntax der Kommandos kann die Möglichkeit der Vertauschung der Reihenfolge nicht wiedergegeben werden.

Einige Kommandos lassen nach dem einleitenden Schlüsselwort mehrere, wesentlich verschiedene Möglichkeiten der Fortsetzung zu. Um die Syntax übersichtlich zu halten, geben wir in diesem Fall mehrere Definitionszeilen an.

Ein Beispiel für alle aufgeführten Möglichkeiten liefert das Kommando LIST. Die Bedeutung der hier nicht erläuterten Formen des LIST-Kommandos ist im Abschnitt 18 zu finden.

LIST [**OFF**] [<gb>] [**FOR**|**WHILE** <bedingung>] [<ausdr_liste>]
Ausgabe der aktiven Datenbank auf dem Bildschirm.

LIST MEMORY [**TO PRINT**]
Anzeige der gültigen Speichervariablen.

LIST STATUS [**TO PRINT**]
Anzeige des Zustands der aktiven Datenbanken einschließlich Indexfiles, Alternate-Files und Systemparametern.

LIST STRUCTURE [**TO PRINT**]
Anzeige der Struktur des aktiven Datenbankfiles.

3.9. Bewegungen durch ein Datenbankfile

Im Abschnitt 3.7 haben wir das Kommando GOTO (oder GO) benutzt. GOTO TOP bewirkt die Positionierung des Dateizeigers auf den ersten Datensatz. Würde diese Anweisung in dem dortigen Beispiel fehlen, so würde kein Datensatz ausgegeben werden. Die Ursache dafür liegt darin, daß durch LIST der Dateizeiger bewegt wird. Nach der Ausführung des Kommandos LIST zeigt der Dateizeiger auf den Datensatz, der dem letzten des Gültigkeitsbereichs folgt, also in diesem Fall auf den zehnten Datensatz (9 Datensätze sind nur in der Datei). Von diesem Datensatz existiert aber kein Nachfolger. Das Kommando ist damit korrekt abgearbeitet. Möchte man die ersten fünf Datensätze sehen, so muß GOTO TOP eingegeben werden. Da die Bewegung des Dateizeigers im Kommando nicht angegeben wird, sprechen wir von einer impliziten Veränderung.

Nehmen wir eine Datenbank in Benutzung (USE), so zeigt der Dateizeiger auf den ersten Datensatz. Eine explizite Positionierung des Dateizeigers ist ebenfalls möglich. Fünf Kommandos sind dafür zuständig: GOTO, SKIP, LOCATE, FIND und SEEK. Die beiden letzten werden wir später besprechen (Abschn. 9.2). Mit GOTO kann der Dateizeiger auf den Anfang (TOP), das Ende (BOTTOM) oder auf einen Datensatz mit vorgegebener Nummer positioniert werden ([RECORD] i). Möchte man sich vom aktuellen Datensatz aus i Datensätze vorwärts oder rückwärts bewegen, so wird

SKIP i

verwendet. Das entspricht einer direkten Adressierung der Datensätze. Ist i positiv, so erfolgt eine Vorwärtspositionierung. Bei negativen Werten wird der Dateizeiger rückwärts bewegt.

LOCATE, FIND und SEEK positionieren den Dateizeiger assoziativ, d.h. nach dem Inhalt eines Feldes oder mehrerer Felder.

Wir kennen somit fünf Möglichkeiten, uns mit dem Dateizeiger durch unsere Datenbank zu bewegen:

1. Eröffnen der Datei;
2. durch die Abarbeitung von Kommandos;
3. direkt durch GOTO;
4. direkt und relativ zum Stand des Dateizeigers (SKIP);
5. assoziativ nach dem Inhalt der Sätze (LOCATE, FIND, SEEK).

Den Wert des Dateizeigers liefert die Funktion RECNO() als Nummer des Datensatzes in der Datei. Möchte man lediglich sehen, wo man sich befindet, so liefert z.B. DISPLAY diese Information.

4. Ausdrücke in dBASE

Im Abschnitt 3 sind wir verschiedenen Ausdrücken begegnet. Wir berechneten den Wert aller Exemplare eines Produkts durch den Ausdruck

```
preis * anzahl
```

und ermittelten den gesamten Wert über alle Produkte durch

```
SUM(preis * anzahl).
```

Der letzte Ausdruck enthält den Aufruf der Funktion SUM. Welche Funktionen durch dBASE bereitgestellt werden, behandeln wir im Abschnitt 4.7. Im ersten Ausdruck werden zwei numerische Größen miteinander multipliziert. Derartige Ausdrücke nennen wir arithmetische Ausdrücke.

Aus den Bedingungen, die zu einem Kommando gehören (FOR <bedingung>), kennen wir Vergleiche (Abschn. 4.4) und die Verknüpfung mehrerer Vergleiche, was zu logischen Ausdrücken führt.

In dem Vergleich

```
produkt = "K"
```

ist "K" eine Zeichenkette, die aus einem Zeichen besteht. Welche Operationen für Zeichenketten zugelassen sind, behandeln wir im Abschnitt 4.5.

Wir betrachten in diesem Abschnitt die grundlegenden Operationen, die auf Konstanten, Felder und Variablen anwendbar sind. Sie sind auch in verschiedenen Programmiersprachen enthalten und sind Operationen "im Kleinen". Damit soll der Unterschied zu den komplexen, datenbanktypischen Operationen hervorgehoben werden, wie sie z.B. in den Kommandos SORT, APPEND, INDEX, JOIN, UPDATE usw. zu finden sind. Dennoch sind diese Operationen "im Kleinen" auf die Belange der Datenbankarbeit zugeschnitten. Insofern unterscheiden sie sich von den Operationen und Standardfunktionen höherer Programmiersprachen wie Pascal, C und Modula, aber auch von den Operationen anderer Anwendersysteme, wie z.B. SuperCalc.

4.1. Feldbezeichner, Datentypen und Konstanten

Feldbezeichner sind Zeichenfolgen von höchstens zehn Zeichen Länge, die aus Buchstaben, Ziffern und dem Unterstreichungsstrich "_" bestehen. Sie beginnen mit einem Buchstaben. Alle kleinen Buchstaben werden bei der Eingabe in große Buchstaben umgewandelt. Das gilt auch für alle Eingaben auf Kommandoebene, d.h., große und kleine Buchstaben werden auf Kommandoebene nicht unterschieden. Sie werden dagegen in Zeichenketten und im Inhalt einer Datenbank als verschiedene

Zeichen behandelt. Zur besseren Lesbarkeit der Kommandos schreiben wir daher Schlüsselwörter groß und selbstgewählte Bezeichner klein. Falsche Zeichen in einem Bezeichner werden nicht angenommen. Der Kursor verbleibt auf der Position. Es erscheint kein Echo der gedrückten Taste auf dem Bildschirm. Der Unterstreichungsstrich ist zur Auflockerung der Bezeichner gedacht.

Ein Feldbezeichner bezeichnet ein Feld eines Datenbankfiles, oder, wenn wir an eine Tabelle denken, so bezeichnet ein Feldbezeichner eine Spalte.

Die möglichen Typen sind in Tafel 4.1 in Deutsch und Englisch zusammengestellt. Sie wurden bereits im Abschnitt 2 beschrieben.

Tafel 4.1. Datentypen in dBASE

Typ (englisch)	dBASE III	dBASE II
numeric	numerisch	numerisch
character	String	String
logic	logisch	logisch
date	Datum	—
memo	Memo	—

Konstanten vom numerischen Typ sind Ziffernfolgen, die einen Punkt enthalten können und denen ein Vorzeichen vorangestellt werden kann. Ist die eingegebene Konstante zu lang, so wird ihr Wert in den letzten Stellen verfälscht, bzw. es wird ein Formatfehler angezeigt (Folge von Sternen). Die nachfolgenden Beispiele demonstrieren diesen Sachverhalt. "?" ist ein Kommando für die Ausgabe. Es kann wie in Basic als "PRINT" gelesen werden oder als "Was ist ...".

```
* Exakte Wiedergabe
. ? 123456789012345
123456789012345
* Die letzten beiden Ziffern sind verfälscht
. ? 1234567890123456789
1234567890123456768
* Formatüberlauf
  ? 1234567890.123456789
  **********
* exakte Wiedergabe
. ? 123456.1234567890
123456.1234567890
* Formatüberlauf
  ? 12345.123456789012345
  **********
```

dBASE kennt nur dezimale Konstanten, also keine hexadezimalen, oktalen oder binären Konstanten.

Konstanten vom Typ String sind Folgen von ASCII-Zeichen, die durch die Paare " ", ' ' oder [ ] begrenzt werden. Das Anfangszeichen muß mit dem Endezeichen korrespondieren. Das gibt uns die Möglichkeit,

alle Zeichen in Zeichenketten aufzunehmen. "]" ist eine Zeichenkette von einem Zeichen Länge, die aus dem Zeichen] besteht, ["'] enthält die zwei Zeichen "' usw.

Probleme können bei der Arbeit mit Zeichen entstehen, die einen internen Kode zwischen 128 und 255 haben, z.B. Umlaute der deutschen Sprache. Einige dBASE-Versionen, so die englische Version 1.00, schneiden das achte Bit jedes Zeichens weg, wodurch Steuerzeichen entstehen. Das führt zu unerwünschten Effekten und unleserlichen Texten. Die deutsche Version 1.10 akzeptiert auch Kodes über 127, d.h. auch Umlaute.

Logische Konstanten werden in den verschiedenen dBASE-Versionen leider sehr unterschiedlich und inkonsequent behandelt. Die deutschsprachige Version 1.10 akzeptiert auf der Kommandoebene die logischen Konstanten .y., .Y., .t. und .T. für wahr (true) und .n., .N., .f. und .F. für falsch (false). Um logische Konstanten von anderen Bezeichnern unterscheiden zu können, werden sie in Punkte eingeschlossen.

Ein logisches Feld in einem Datenbankfile hat die Länge 1. Bei der Eingabe oder Korrektur werden die Punkte weggelassen, da keine Verwechslungen mit Bezeichnern auftreten können. Im Direktmodus (APPEND, BROWSE, CHANGE, EDIT) werden die Zeichen j, J, t und T für wahr und n, N, f und F für falsch akzeptiert. Im Datenbankfile werden z.B. bei den Kommandos EDIT und BROWSE die eingegebenen Zeichen in großen Buchstaben wiedergegeben. Beim Kommando LIST dagegen wird für alle logischen Werte 'wahr' .T. und für 'falsch' .F. ausgegeben. Beim Kommando INPUT werden in der deutschsprachigen Version nicht die Konstanten .j. und .J. akzeptiert!

Wurde ein Datenbankfile mit der englischen Version erstellt, so akzeptiert die deutschsprachige Version auch die logischen Konstanten y und Y als Werte von Feldern. Ein ähnliches Durcheinander findet man bei der englischen Version 1.00.

> Alle untersuchten Versionen akzeptieren die logischen Konstanten t, T, f und F bzw. in Punkte eingeschlossen auf der Kommandoebene. Will man logische Konstanten benutzen, so sollte man nur diese nehmen!

Es gibt keine Konstanten vom Typ Datum. Lediglich bei der Datenerfassung oder -korrektur im Direktmodus kann ein Datum als Folge von Ziffern eingegeben werden. Ein Datum wird in der Form tt.mm.jj eingegeben, z.B. 29.02.88 (deutsche Version). In den englischen Versionen entsprechend mm.tt.jj. Bei der Eingabe wird getestet, ob es sich um ein gültiges Datum handelt. So wird z.B. der 29.02.88 akzeptiert, während der 29.02.87 zurückgewiesen wird (Abschn. 4.7).

Man kann das aktuelle Datum durch die Funktion DATE() erhalten und damit Operationen ausführen (Abschn. 4.6). Die Konvertierung einer Zeichenkette in ein Datum ist durch Anwendung der Funktion CTOD() (character to date) möglich:

```
CTOD("29.02.88")
```

liefert das gewünschte Datum.

Beim Typ Memo hat es keinen Sinn, von Konstanten zu sprechen. Der Inhalt eines Memo-Feldes ist ein Text, der mit dem Editor im Direktmodus bearbeitet und in einigen Kommandos ausgegeben werden kann. Eine Zuweisung von Texten zu Memo-Feldern ist auf der Kommandoebene nicht möglich.

4.2. Speichervariable

Speichervariable (Memory-Variable) sind Objekte mit den folgenden Eigenschaften:

1. Sie können zu einem Zeitpunkt jeweils einen Wert aufnehmen.

2. Speichervariablen werden durch ihre Benutzung vereinbart. Der ihnen zugewiesene Wert bestimmt ihren Typ und ggf. ihr Format (Länge, Anzahl der Dezimalstellen).

3. Sie haben einen Typ, der gleich dem Typ des zugewiesenen Wertes ist.

4. Ihr Typ kann durch die Zuweisung eines anderen Wertes geändert werden. Es bestehen also nicht die Möglichkeiten der Typkontrolle, wie sie aus höheren Programmiersprachen bekannt sind.

5. Zulässige Typen für Speichervariable sind: numerisch, String (maximal 254 Zeichen), logisch und Datum.

6. Sie werden durch einen Bezeichner benannt. Dieser ist der Name der Variablen. Die Konventionen für die Bildung von Bezeichnern sind die gleichen wie für Felder. Werden für Felder und Speichervariable gleiche Namen gewählt, so haben auf Positionen, auf denen Felder und Speichervariable auftreten können, Felder den Vorrang. Betrachten wir dazu das folgende Beispiel:

```
* Das Datenbankfile 'lager.dbf' enthält ein Feld 'Artikel'.
. USE lager
* Eine Ergibtanweisung wirkt nur auf Speichervariable.
* Es wird eine neue Speichervariable erzeugt.
. artikel = 2.718
2.718
. * Wir geben das Feld Artikel und die Speichervariable Artikel aus.
. DISPLAY artikel, m->artikel
 Satz #  artikel             m->artikel
      1  Flachschieber            2.718
```

Der Buchstabe m, für Memory-Variable, kann der Bezeichnung einer Speichervariablen wie ein Aliasname vorangestellt werden.
Werden Felder oder Variable wie Schlüsselwörter bezeichnet, so haben auf Positionen, auf denen eine Auswahl getroffen werden

muß, Schlüsselwörter den Vorrang vor Feldern und Speichervariablen.
Programme sind jedoch sicherer und besser lesbar, wenn unterschiedliche Bezeichner verwendet werden. Es wird empfohlen, Namen von Speichervariablen mit dem Buchstaben m beginnen zu lassen.

7. Speichervariable werden im Speicher gehalten, solange nicht durch CLEAR MEMORY oder RELEASE etwas anderes veranlaßt wird. SAVE MEMORY bewirkt das Speichern aller oder ausgewählter Speichervariablen in ein File (.mem). Dieses File kann über RESTORE wieder eingelesen werden.

8. Alle zu einem Zeitpunkt bekannten Speichervariablen heißen aktiv. Sie sind intern in einem Speicherbereich untergebracht, der auf 6000 Byte begrenzt ist. Im Konfigurationsfile kann diese Grenze erhöht werden, ohne daß jedoch die Anzahl der möglichen Speichervariablen dadurch erhöht werden kann. Durch Auslagerung in ein File (SAVE MEMORY) und spätere Reaktivierung (RESTORE) kann dieser Engpaß umgangen werden.

9. Die Lebensdauer von Speichervariablen hängt von ihrer Erzeugung ab. Sie ist nicht an irgendein (aktives) Datenbankfile gebunden. Somit eignen sich Speichervariable als Zählgrößen, zur Speicherung von Zwischenergebnissen, für die Eingabe und spätere Verteilung in mehrere Datenbankfiles, aber auch zum Austausch oder zur Vereinigung von Informationen aus mehreren Datenbankfiles.

Wird eine Speichervariable von der Tastatur aus erzeugt, also nicht in einem Kommandofile (.cmd), so wird sie beim Verlassen von dBASE (QUIT) oder durch ein RELEASE-Kommando freigegeben. USE-Kommandos haben darauf keinen Einfluß. Wird sie in einem Kommandofile erzeugt, so wird sie beim Verlassen dieses Kommandofiles vernichtet. Beim Wiedereintritt in das Kommandofile ist ihr alter Wert verloren. Wird sie in einem Kommandofile erzeugt und als PUBLIC erklärt, so besteht sie bis zum Verlassen von dBASE. Wird sie in einer Prozedur als PRIVATE deklariert, so ist sie außerhalb dieser Prozedur nicht sichtbar. Gleiche Bezeichner für Speichervariable in verschiedenen Prozeduren sind somit möglich. Hier besteht ein wesentlicher Unterschied zu dBASE II, der auch bei der Übertragung von dBASE II zu dBASE III zu Fehlern führt (Abschn. 17.4). In dBASE II ist die Lebenszeit einer Speichervariablen durch die Beendigung von dBASE oder durch RELEASE bestimmt.

Wertzuweisungen zu Speichervariablen erfolgen grundsätzlich anders als Wertzuweisungen zu Feldern eines Datenbankfiles. Die bekannte Ergibtanweisung existiert für Speichervariable (STORE oder =), aber nicht für Felder. Weitere Kommandos und Schlüsselwörter, die sich auf Speichervariable beziehen, sind ACCEPT, AVERAGE, COUNT, CLEAR, INPUT, PARAMETERS, PRIVATE, PUBLIC, RELEASE, RESTORE, SAVE, STORE, SUM und WAIT. Wertzuweisungen zu Feldern erfolgen im Direktmodus durch APPEND, BROWSE, CHANGE, EDIT oder im Kommandomodus durch REPLACE oder das @-Kommando.

Die Verwendung von Speichervariablen in Kommandofiles wird im Abschnitt 11 behandelt.

4.3. Arithmetische Ausdrücke

Die einfachste Form eines arithmetischen Ausdrucks sind Konstanten, Felder oder Speichervariable. Sie können durch arithmetische Operatoren verknüpft werden. So entstehen kompliziertere arithmetische Ausdrücke, die ihrerseits wieder durch arithmetische Operatoren verknüpft werden können usw. Die Operatoren sind in Tafel 4.2 zusammengestellt. Die höchste Priorität haben die Operatoren mit der höchsten Prioritätsziffer.

Tafel 4.2. Arithmetische Operatoren

Operator	Operation	Beispiel	Priorität
+ und -	Vorzeichen	-12.34	8
^ oder **	Potenzieren	a ^ b	7
*	Multiplikation	a * b	6
/	Division	a / b	6
+	Addition	a + b	5
-	Subtraktion	a - b	5

Durch runde Klammern kann die Reihenfolge der Berechnung eines Ausdrucks verändert werden. Diese arithmetischen Ausdrücke entsprechen unserer allgemeinen Vorstellung von Ausdrücken. So ist es z.B. möglich, den Inhalt von zwei Feldern zu addieren, zu multiplizieren usw. Über die Reihenfolge der Abarbeitung eines Ausdrucks ist nichts ausgesagt.

4.4. Vergleiche und logische Ausdrücke

Ausdrücke der Typen numerisch und String können miteinander durch Vergleichsoperatoren verglichen werden. Das Ergebnis ist ein logischer Wert. Tafel 4.3 faßt die Vergleichsoperatoren zusammen. Eine Aneinanderkettung von Vergleichen ist nicht möglich. Will man testen, ob eine Varible i zwischen 1 und n liegt (mathematisch: 1 <= i <= n), so ist in dBASE dafür zu schreiben

```
(1 <= i) .AND. (i <= n)
```

Jeder Vergleich liefert einen logischen Wert. Beide logische Werte werden durch die Operation "logisches Und" verknüpft (Operator .AND.). Die Klammern sind in diesem Fall nicht erforderlich, erhöhen aber die Lesbarkeit des Programms.

Tafel 4.3. Vergleichsoperatoren

Operator	Bedeutung	Beispiel	Priorität
=	gleich	produkt = "K"	4
<> oder #	ungleich	produkt # "K"	4
<	kleiner	produkt < "K"	4
>	größer	produkt > "K"	4
<=	kleiner gleich	produkt <= "K"	4
>=	größer gleich	produkt >= "K"	4

Die Vorrangfolge der Vergleichsoperatoren untereinander ist ohne Bedeutung, da nur jeweils zwei Größen verglichen werden können. Die Priorität 4 wurde hier vergeben, um die Vorrangordnung aller Operationen untereinander zum Ausdruck zu bringen. Die logischen Operatoren sind in Tafel 4.4 zusammengestellt. a und b sind logische Ausdrücke, also im einfachsten Fall logische Felder.

Tafel 4.4. Logische Operatoren

Operator	Operation	Beispiel	Priorität
.NOT.	Negation	.NOT. EOF()	3
.AND.	logisches Und	a .AND. b	2
.OR.	logisches Oder	a .OR. b	1

Sei i ein numerisches Feld, dann ist

```
1 <= i+4 .AND. i+4 <= 100
```

ein gültiger logischer Ausdruck. Zuerst werden die arithmetischen Ausdrücke ausgewertet. Arithmetische Operatoren besitzen die höchste Priorität. Danach werden die Vergleichsoperatoren ausgewertet und dann die logischen Operatoren. Die Prioritätsziffern in den letzten drei Tafeln geben diese Priorität untereinander wieder. Deutlicher wäre für den obigen Ausdruck die Schreibweise

```
(1 <= i+4) .AND. (i+4 <= 100)
```

Logische Ausdrücke werden sehr häufig in einer FOR- oder WHILE-Bedingung benutzt. Sie bestimmen dann, auf welche Datensätze sich das betreffende Kommando bezieht.

4.5. Operationen mit Zeichenketten

Operationen mit Zeichenketten sind in dBASE weitaus wichtiger als in gewöhnlichen höheren Programmiersprachen. Sie bilden die Grundlage

für viele sortierte Zusammenstellungen, für die Suche nach bestimmten Datensätzen, für die Verkettung von Zeichenketten zu lesbaren Texten usw.

Vergleiche von Zeichenketten werden durchgeführt, indem ein von dBASE intern vergebener Kode benutzt wird. Der Vergleich von zwei Zeichenketten erfolgt zeichenweise von links nach rechts, bis eine der beiden Zeichenketten erschöpft ist. Trifft dann die Relation zu (größer, kleiner usw.), so ist der Wert des Vergleichs wahr (.T. für true). Wurde bereits vorher festgestellt, daß die Relation nicht gelten kann, so wird der Vergleich abgebrochen. Der Wert des Vergleichs ist dann falsch (.F. für false).

Die interne Zeichenordnung von dBASE bewirkt, daß alle Ziffern, alle großen Buchstaben und alle kleinen Buchstaben in sich geordnet sind. Alle Ziffern sind kleiner als alle großen Buchstaben, und diese sind wiederum - paradoxerweise - kleiner als alle kleinen Buchstaben. Durch Anwendung der Funktion UPPER kann trotzdem die gewohnte Ordnung hergestellt werden.

Aufmerksamkeit sollte man den Umlauten und anderen Sonderzeichen schenken. Während die englische Version 1.00 keine Umlaute akzeptiert, können sie in der deutschsprachigen Version 1.10 im Text, in den Feldern und in Bezeichnungen von Variablen und Feldern benutzt werden. Beim Sortieren werden Umlaute nach dem zugehörigen Vokal eingeordnet, also ä nach a, ö nach O usw. Der Buchstabe ß wird zwischen s und t eingeordnet, ist in Feldbezeichnern aber nicht zugelassen. Das ist ein wesentlicher Unterschied zu den bekannten Versionen von dBASE II, auch denen, die auf 16-Bit-Rechnern laufen, und zu anderen dBASE III-Versionen. Es entsteht eine für die deutschsprachige Version von dBASE III spezifische Anordnung der Zeichen. Tafel 4.5 zeigt ein sortiertes Datenbankfile, in dem gängige Zeichen, einschließlich einiger Zeichen des erweiterten ASCII-Zeichensatzes, enthalten sind. Sortiert wurde nach den ASCII-Zeichen. Zur Kontrolle ist der numerische Wert der ASCII-Kodes angegeben (Funktion ASC).

Zeichenketten können durch die Operationen "+" und "-" verknüpft werden.

"+" fügt die Zeichenketten aneinander. Leerzeichen bleiben erhalten. Die Länge der Ergebniszeichenkette ist die Summe der Längen der einzelnen Zeichenketten.

"-" fügt Zeichenketten aneinander, wobei Leerzeichen am Ende der ersten Zeichenkette an das Ende der zweiten Zeichenkette angefügt werden.

Die Operationen werden von links nach rechts ausgeführt. Ein Beispiel soll die Wirkung veranschaulichen. Gegeben sei ein Datenbankfile mit zwei Feldern "name" und "vorname" vom Typ String und von der Breite 10. Wir wollen erreichen, daß Name und Vorname durch Komma getrennt und mit einem Leerzeichen nach dem Komma eine Zeichenkette bilden.

Tafel 4.5. Sortierreihenfolge der ASCII-Zeichen (deutsche dBASE-Version 1.10)

Satz #	zeichen	ASC(zeichen)	Satz #	zeichen	ASC(zeichen)
1	<leerzeichen>	32	28	{	123
2	0	48	29	}	125
3	9	57	30	<	60
4	A	65	31	>	62
5	Ä	142	32	=	61
6	B	66	33	-	45
7	O	79	34	%	37
8	Ö	153	35	*	42
9	U	85	36		46
10	Ü	154	37	,	44
11	Z	90	38	;	59
12	a	97	39	:	58
13	ä	132	40	?	63
14	b	98	41	!	33
15	o	111	42	/	47
16	ö	148	43	\	92
17	p	112	44	¦	124
18	s	115	45	@	64
19	ß	225	46	&	38
20	t	116	47	#	35
21	u	117	48	$	36
22	ü	129	49	$	36
23	z	122	50	"	34
24	(	40	51		96
25	)	41	52		39
26	[	91	53		94
27	]	93	54	~	126
			55	_	95

```
* Hier stören die Leerzeichen nach dem Namen.
. ? name + vorname
Kant       Immanuel
* Durch '-' werden alle Leerzeichen nach dem Namen entfernt ...
. ? Kant - name - vorname
KantImmanuel
* ... und nach dem zweiten Operanden eingefügt.
. ? name - ", " + vorname
Kant,        Immanuel
* Hier stehen wieder alle Leerzeichen am Ende der Zeichenkette.
. ? name - ", " - vorname
Kant,Immanuel
* TRIM schneidet die Leerzeichen am Ende einer Zeichenkette ab.
. ? TRIM(name) + ", " + vorname
Kant, Immanuel
```

Neben den Vergleichen und der Verkettung von Zeichenketten stellt

dBASE Funktionen für die Arbeit mit Zeichenketten bereit. Sie werden im Abschnitt 4.7 behandelt.

4.6. Operationen mit Größen vom Typ Datum

Für Größen vom Typ Datum sind die Operationen "+" und "-" erklärt.

"+" addiert zu einem Datum eine ganze Zahl und liefert einen Wert vom Typ Datum. Dabei werden Überträge vom Tag zum Monat und vom Monat zum Jahr berücksichtigt.

"-" subtrahiert von einem Datum eine ganze Zahl, die als Anzahl von Tagen interpretiert wird, und liefert einen Wert vom Typ Datum.

"-" subtrahiert außerdem von einem Datum ein anderes Datum und liefert als numerischen Wert die Anzahl der Tage.

Nachfolgend einige Beispiele. "datum" und "datum2" sind Speichervariablen vom Typ Datum.

```
. ? datum
07.11.87
. ? datum + 60
06.01.88
. ? datum - 365
07.11.86
. ? datum - datum2
      -30
. ? datum2
07.12.87
* CTOD konvertiert eine Zeichenkette in ein Datum.
* Es wird die Differenz in Tagen berechnet.
. ? datum - CTOD("28.07.48")
    14346
* Die Eingabe anderer Jahrhunderte ist nur über die Funktion
* CTOD möglich.
* Geburtstag von Einstein
. m_geb_ein = CTOD("14.03.1879")
14.03.79
* Sterbetag
. m_gest_ein = CTOD("18.04.55")
18.04.55
* Wie alt wurde Einstein?
. ? YEAR(m_gest_ein) - YEAR(m_geb_ein)
   76
. * Ein Datum vor dem Jahr 100 kann nicht eingegeben werden
. m_d1 = CTOD("11.11.0009")
11.11.09
. ? YEAR(m_d1)
 1909
```

```
* ... aber die Jahreszahl der Schlacht im Teuteburger Wald
* läßt sich berechnen.
. m_d2 = CTOD("01.01.100")
01.01.00
. ? m_d3 = m_d2 - 365*91
23.01.09
  ? YEAR(m_d3)
    9
* Unter Berücksichtigung der Schaltjahre
. m_d4 = m_d2 - 365*91 - INT(91/4)
01.01.09
. m_* Ein Datum v.u.Z. kann nicht verwaltet werden.
. m_d5 = m_d4 - 365*20
./.02.//
* Aber Vergleiche funktionieren noch
. ? m_d5   m_d4
.T.
```

4.7. Funktionen

In diesem Abschnitt besprechen wir die in dBASE III implementierten Funktionen im Detail. Dabei sind Vorgriffe auf die folgenden Abschnitte unumgänglich. Sie werden durch Verweise angegeben. Wir wählen hier eine alphabetische Ordnung der Funktionen. Eine thematische Auflistung unter verschiedenen Gesichtspunkten findet man im Anhang 1.

Für die Notation erinnern wir uns an unsere Festlegungen aus dem Abschnitt 2. Die Namen der Funktionen werden groß geschrieben, Metasymbole in spitze Klammern eingeschlossen und klein geschrieben. Unter Syntax wird die Schreibweise angegeben und danach ihre Wirkung erklärt, ggf. durch Beispiele unterstützt. Hat eine Funktion keine Parameter, so wird das durch ein leeres Klammerpaar nach dem Namen angegeben. Funktionen ohne Parameter sind dadurch von anderen Bezeichnern unterscheidbar.

& — Makroersetzung

Syntax: &<mem_var>

Die Speichervariable <mem_var> muß vom Typ String sein. Der Inhalt von <mem_var> wird für &<mem_var> als Zeichenkette in den Text eingefügt. "&" unterscheidet sich von den anderen Funktionen dadurch, daß die Makroersetzung vor jeder anderen Aktion ausgeführt wird, insbesondere vor der Interpretation einer Kommandozeile. Dadurch lassen sich Kommandozeilen während der Ausführung modifizieren.

Der Inhalt der Speichervariablen m_datei sei "myfile". Er kann z.B. durch eine Eingabe von der Tastatur (ACCEPT, INPUT) eingegeben

worden sein.

```
USE &m_datei
```

bewirkt, daß zuerst &m_datei substituiert wird. Damit wirkt das Kommando, als stünde dort

```
USE myfile
```

Das Datenbankfile "myfile.dbf" und ggf. das Memo-File "myfile.dbt" werden in Benutzung genommen.

Makroersetzung ist auch innerhalb von Zeichenketten möglich:

```
. m_datei = "myfile"
myfile
. m_text = "Mein Datenbankfile ist &m_datei"
Mein Datenbankfile ist myfile
. m_text = "Mein Datenbankfile ist &m_datei..dbf"
Mein Datenbankfile ist myfile.dbf
```

ASC — Funktion

Syntax: ASC(<string>)

ASC liefert den dezimalen Wert des ersten Zeichens der Zeichenkette <string>.

```
. ? ASC("L")
  76
. ? ASC("Lagerverwaltung")
  76
. m_datei = "myfile"
myfile
* 109  ist  der dezimale Wert des Zeichens "m" von  myfile
. ? ASC(("&m_datei")
 109
* myfile ist keine Zeichenkette
. ? ASC(&m_datei)
Variable nicht gefunden
             ?
? ASC(myfile)
Wünschen Sie HILFE? (J/N) Nein
```

AT —Funktion

Syntax: AT(<string_ausdr_1>,<string_ausdr_2>)

AT sucht, von links beginnend, die Zeichenkette 1 in der Zeichenkette 2. Die Zeichenkette 1 heißt auch Teilzeichenkette (substring). AT liefert die Position des ersten Zeichens der Teilzeichenkette in der Zeichenkette 2. Ist die Zeichenkette 1 nicht in der Zeichenkette

2 enthalten, so liefert AT den Wert 0.

```
* Das  sechste Zeichen in "Lagerhaltung" ist "h"
. ? AT("halt","Lagerhaltung")
        6
* Die  Zeichenkette "2a" beginnt ebenfalls ab Position 6
*  in der Zeichenkette "123452abc"
. ? AT("2"+"a", "123452"+"abc")
        6
. m_text = "Mein Datenbankfile ist &m_datei..dbf"
Mein Datenbankfile ist myfile.dbf
* my beginnt auf der Position 24 in der Variablen m_text
. ? AT( "my",m_text)
       24
* Es wird der Wert 0 geliefert,  da die Zeichenkette "MY" (groß
geschrieben!) nicht in der zweiten Zeichenkette enthalten ist
. ? AT("MY",m_text)
        0
```

BOF — Funktion

Syntax: BOF()

BOF (begin of file) wird auf das aktive Datenbankfile des ausgewählten (aktiven) Arbeitsbereichs angewendet. Sie liefert einen logischen Wert. Wahr (.T.) wird geliefert, wenn versucht wird, den Dateizeiger vor dem Anfang des Files zu positionieren, sonst falsch (.F.). BOF kann vorteilhaft benutzt werden, wenn eine Datei rückwärts durchschritten wird.

```
. USE lager
* Unmittelbar nach dem Eröffnen liefert BOF() falsch.
* Der Dateizeiger zeigt auf den ersten Satz (RECNO).
. ? BOF(), RECNO()
.F.          1
* Wir versuchen einen Satz zurückzugehen.
. SKIP  -1
Satz Nr.       1
* Jetzt liefert BOF() wahr. Der Dateizeiger ist unverändert.
. ? BOF(), RECNO()
.T.          1
```

CDOW — Funktion

Syntax: CDOW(<dat_ausdr>)

CDOW (character day of week) liefert den Namen des Wochentages des Datums. <dat_ausdr> ist ein Datumsausdruck.

```
* Aus einer Zeichenkette bilden wir ein Datum
* und ermitteln von diesem den Wochentag
```

```
. ? CDOW(CTOD("24.12.88"))
Samstag
. ? "Heute ist "+CDOW(DATE())
Heute ist Sonntag
```

CHR — Funktion

Syntax: CHR(<cardinal_byte>)

<cardinal_byte> ist eine positive ganze Zahl i, die in einem Byte speicherbar ist (0 < i <= 255). CHR liefert diesen Wert in einem Byte. Damit können in einen Text Zeichen eingefügt werden (Sonderzeichen oder Steuerzeichen), für die auf der Tastatur keine Taste vorhanden ist. Steuerzeichen für den Drucker können so ausgegeben werden. Der Wechsel der Schriftart und fast alle programmierbaren Funktionen des angeschlossenen Druckers sind dadurch erreichbar (Abschn. 17.6). Ein Byte mit dem Wert 00h läßt sich so nicht erzeugen.

```
* 48 = 30h ist das Zeichen 0
. ? CHR(48)
0
* 7 entspricht ^G und bewirkt die Ausgabe eines
* akustischen Signals
. ? "Nicht einschlafen "+ CHR(7)
Nicht einschlafen
```

CMONTH — Funktion

Syntax: CMONTH(<dat_ausdr>)

CMONTH liefert den Namen des Monats des angegebenen Datums. <dat_ausdr> ist ein Datumsausdruck.

```
. ? CMONTH(DATE())
November
. ? "In 1000 Tagen haben wir den Wochentag " +;
 "CDOW(DATE()+1000) + " und den Monat " + CMONTH(DATE()+1000)
In 1000 Tagen haben wir den Wochentag Samstag und den Monat August
```

COL — Funktion

Syntax: COL()

COL liefert die Spaltennummer des Kursors auf dem Bildschirm. COL ist für die Steuerung der Ausgabe im Direktmodus auf dem Bildschirm nützlich. Sie wird häufig in @-Kommandos eingesetzt.

```
@ 10, COL()+5 SAY "Hallo!"
```

In Zeile 10 wird ab aktuelle Spalte + 5 das Wort "Hallo!" ausgegeben.

CTOD — Funktion

Syntax: CTOD(<string_ausdr>)

CTOD (character to date) konvertiert die angegebene Zeichenkette in einen Wert vom Typ Datum. Um eine sinnvolle Konvertierung zu erreichen, sollte die Zeichenkette ein gültiges Datum darstellen. Als Trennzeichen zwischen Tag und Monat und Monat und Jahr werden Zeichen akzeptiert, die keine Ziffer sind. Sie werden in der Version 1.10 als Trennzeichen verworfen und durch Punkte ersetzt. CTOD konvertiert nahezu jede Zeichenkette. Dadurch werden auch sinnlose Datumsangaben erzeugt. Standardmäßig gibt es keine Möglichkeit, zu prüfen, ob ein gültiges Datum erzeugt wurde.

```
. m_date = CTOD("31.12.99")
31.12.99
* TYPE liefert den Typ, hier D für Datum
.  ? TYPE ("m_date")
D
* Der 1.1.2000 ergibt sich durch Addition von 1
. ? m_date + 1
01.01.00
* Es können Stringausdrücke gebildet werden
. ? CDOW(m_date) + ", der ", m_date
Freitag, der  31.12.99
* ... mit anderen Trennzeichen
. ? CTOD("24x12x87")
24.12.87
* Wird ein ungültiges Datum eingegeben, so erfolgt eine Umwandlung.
* Manchmal sind sie sinnvoll ...
. ? CTOD("30.02.88")
01.03.88
* ... und manchmal nicht.
. ? CTOD("311287")
12.01.55
* CTOD ist der einzige Weg, andere Jahrhunderte einzugeben
. m = CTOD("17.8.1875")
17.08.75
* YEAR liefert das Jahr vierstellig
. ? YEAR(m)
1875
```

DATE — Funktion

Syntax: DATE()

DATE liefert das Systemdatum. Je nach Version wird es in der deutschen (tt.mm.jj) oder in der englischen Form (mm/tt/jj) ange-

zeigt.

```
. ? DATE()
08.11.87
. ? DATE() + 1000
04.08.90
```

DAY — Funktion

Syntax: DAY(<dat_ausdr>)

DAY liefert den numerischen Wert des Tages des angegebenen Datums, relativ zum Monatsanfang.

```
. ? DAY(DATE())
  8
```

DELETED — Funktion

Syntax: DELETED()

DELETED testet, ob der aktuelle Datensatz als gestrichen markiert ist. Das Testergebnis wird als logischer Wert geliefert.

```
. USE lager
* Der  erste  Datensatz des Datenbankfiles lager.dbf ist nicht  gestrichen
. ? DELETED(), RECNO()
.F.          1
```

DOW — Funktion

Syntax: DOW(<dat_ausdr>)

DOW liefert als numerischen Wert den Wochentag. Sonntag liefert 1, Montag 2 usw.

```
* Heute ist Sonntag, der 8.11.1987
. ? DATE()
08.11.87
* Sonntag ist der erste Tag der Woche
. ? DOW(DATE())
 1
. ? CDOW(DATE())
Sonntag
```

DTOC — Funktion

Syntax: DTOC(<dat_ausdr>)

DTOC konvertiert das angegebene Datum in eine Zeichenkette.

```
. ? .DATE()
05.12.87
* Das Datum wird als Zeichenkette der Speichervariablen zugewiesen
. m_cdat = DTOC(DATE())
05.12.87
* Der Inhalt von m_cdat hat den Typ string (character)
. ? TYPE("m_cdat")
C
. ? DTOC(DATE() + 100)
14.03.88
```

EOF — Funktion

Syntax: EOF()

EOF (end of file) wird auf das aktive Datenbankfile des ausgewählten (aktiven) Arbeitsbereichs angewendet. Sie liefert einen logischen Wert. Wahr (.T.) wird geliefert, wenn versucht wird, den Dateizeiger über das Ende der Datei hinaus zu positionieren, sonst falsch (.F.). EOF kann vorteilhaft in DO-WHILE-Kommandos benutzt werden.

```
. USE lager
* Positioniert man auf den letzten Datensatz, so liefert EOF() falsch
. GO BOTTOM
. ? EOF(), RECNO()
.F.          9
. SKIP
* Es wird Satznummer 10 geliefert, obwohl ein solcher Satz nicht existiert
Satz Nr.     10
. ? EOF(), RECNO()
.T.          10
```

EXP — Funktion

Syntax: EXP(<ausdr>)

<ausdr> ist ein numerischer Ausdruck. Dieser ist der Exponent, mit dem e (2.7182818) potenziert wird. EXP liefert den numerischen Wert $e^{<ausdr>}$.

```
* Standardmäßig werden zwei Dezimalstellen ausgegeben
. ? EXP(1)
      2.72
. ? EXP(1.000)
     2.718
```

FILE — Funktion

Syntax: FILE(<string_ausdr>)

<string_ausdr> ist ein Ausdruck vom Typ String. FILE prüft, ob die berechnete Zeichenkette ein Filename im Fileverzeichnis ist, und liefert als Resultat einen logischen Wert, d.h. wahr (.T.), wenn das File vorhanden ist. Alle kleinen Buchstaben werden in große Buchstaben umgewandelt.

```
. ? FILE("lager.dbf")
.T.
. ? FILE("LAGER.dbt")
.T.
* Auch Pfadnamen und die Angabe eines Laufwerks sind möglich
. ? FILE("\sfd\abs4.txt")
.T.
```

INT — Funktion

Syntax: INT(<ausdr>

<ausdr> ist ein numerischer Ausdruck. INT liefert den ganzzahligen Anteil von <ausdr> als numerischen Wert.

```
. ? INT(12.99)
 12
```

LEN — Funktion

Syntax: LEN(<string_ausdr>)

<string_ausdr> ist ein Ausdruck vom Typ String. LEN liefert als numerischen Wert die Anzahl der Zeichen in der berechneten Zeichenkette.

```
. USE lager
. DISPLAY artikel
 Satz #  artikel
      1  Flachschieber
* Die Breite des Feldes "artikel" wurde mit 15 Zeichen definiert
. ? LEN(artikel)
        15
* TRIM schneidet die Leerzeichen am Ende der Zeichenkette ab
. ? LEN(TRIM(artikel))
        13
. m_datei = "myfile"
myfile
* Die Substitution der Makrovariablen  erfolgt vor der Bestimmung der Länge.
* Die resultierende Zeichenkette besteht  in diesem Fall aus 35 Zeichen.
. ? LEN("Mein Datenbankfile heißt &m_datei..dbf")
        35
```

LOG — Funktion

Syntax: LOG(<ausdr>)

<ausdr> ist ein numerischer Ausdruck. LOG liefert den natürlichen Logarithmus des durch <ausdr> bestimmten Wertes.

```
. ? LOG(2.71828)
   1.00000
. ? LOG(2*5*2**10)
      9.23
```

LOWER — Funktion

Syntax: LOWER(<string_ausdr>)

<string_ausdr> ist ein Ausdruck vom Typ String. LOWER liefert als Ergebnis eine Zeichenkette, die aus <string_ausdr> berechnet wurde, wobei alle großen Buchstaben durch kleine Buchstaben ersetzt sind.

```
 ? LOWER('Ehm-Welk-Straße 57')
ehm-welk-straße 57
```

MONTH — Funktion

Syntax: MONTH(<datum>)

MONTH liefert einen numerischen Wert, der die Nummer des Monats des angegebenen Datums ist.

```
. ? DATE()
02.12.87
* Dezember ist der zwölfte Monat
. ? MONTH(DATE())
 12
```

PCOL — Funktion

Syntax: PCOL()

PCOL (printer column) bezieht sich auf die Position des nächsten zu druckenden Zeichens auf dem Drucker. PCOL liefert einen numerischen Wert, der die Nummer der Spalte ist, in der das nächste Zeichen gedruckt wird.

Diese Funktion wird vorzugsweise verwendet, um bei der direkt positionierten Ausgabe auf dem Drucker (@-Kommando) die nächste Spalte zu berechnen.

```
. SET DEVICE TO PRINTER
. @ 10,PCOL()+5 SAY "Ausgabe in Zeile 10, aktuelle Spalte plus 5"
```

Es gibt die hinter SAY stehende Zeichenkette auf dem Drucker aus (durch SET DEVICE TO PRINTER). Der erste Buchstabe der Zeichenkette steht in Zeile 10 und in der aktuellen Spalte plus fünf.

PROW — Funktion

Syntax: PROW()

PROW (printer row) bezieht sich auf die Position des nächsten zu druckenden Zeichens auf dem Drucker. PROW liefert einen numerischen Wert, der die Nummer der Zeile ist, in der das nächste Zeichen gedruckt wird.

Diese Funktion wird vorzugsweise verwendet, um bei der direkt positionierten Ausgabe auf dem Drucker (@-Kommando) die nächste Zeile zu berechnen.

```
. SET DEVICE TO PRINTER
. @ PROW()+10,5 SAY "Ausgabe aktuelle Zeile plus 10, Spalte 5"
```

Beginnt man eine direkt positionierte Ausgabe auf dem Drucker relativ zur aktuellen Zeile, also durch PROW()+1, so wird vor dem Druck kein Seitenvorschub ausgeführt.

RECNO — Funktion

Syntax: RECNO()

RECNO liefert die Nummer des Datensatzes, auf den der Dateizeiger zeigt, als numerischen Wert. Ist in dem aktiven Arbeitsbereich kein Datenbankfile in Benutzung, so liefert RECNO() den Wert 0.

```
. ? RECNO()
        0
. USE lager
. SKIP 2
Satz Nr.      3
. ? RECNO()
        3
```

ROUND — Funktion

Syntax: ROUND(<ausdr>,<n>)

Der numerische Ausdruck <ausdr> wird berechnet und der so ermittelte Wert auf n Stellen nach dem Komma gerundet. Für <n> sind auch negative Werte zugelassen, die Stellen vor dem Dezimalpunkt angeben.

```
. ? ROUND(2.718281828459,5)
 2.718280000000
. ? ROUND(2.71828,2)
 2.72000
* Negative Werte für n bewirken, daß auf n Stellen vor dem
* Dezimalpunkt gerundet wird
. ? ROUND(100 * 2.7182818,-1)
   270.0000000
```

ROW — Funktion

Syntax: ROW()

ROW liefert einen numerischen Wert, der die Nummer der Zeile angibt, in der sich der Bildschirmkursor befindet. Die Verwendung ähnelt der Funktion PROW, nur daß sich diese auf die Druckposition bezieht.

SPACE — Funktion

Syntax: SPACE(<ausdr>)

SPACE liefert als Ergebnis eine Zeichenkette, die aus Leerzeichen besteht. Die Anzahl der Leerzeichen in der Zeichenkette wird durch den Wert des numerischen Ausdrucks bestimmt. SPACE ist günstig einsetzbar, wenn z.B. vor der Eingabe mit dem @-Kommando eine Speichervariable von n Zeichen Länge definiert werden muß. Außerdem erspart sie das Abzählen mehrerer Leerzeichen.

SQRT — Funktion

Syntax: SQRT(<ausdr>)

SQRT liefert die Quadratwurzel des numerischen Ausdrucks <ausdr>. Die Genauigkeit des Ergebnisses kann durch die Genauigkeit des Wertes von <ausdr> erhöht werden. Das Ergebnis wird mindestens mit zwei Stellen nach dem Komma dargestellt.

```
. ? SQRT(2.00)
     1.41
. ? SQRT(2)
     1.41
. ? SQRT(4.000000)
  2.000000
. ? SQRT(-1)
***Ausführungsfehler bei SQRT() : negativer Wert
**********
```

STR — Funktion

Syntax: STR(<ausdr>[,<länge>[,<dez>]])

STR konvertiert den Wert des numerischen Ausdrucks <ausdr> in eine Zeichenkette. Alle drei Parameter sind numerische Ausdrücke. <länge> bestimmt die Länge dieser Zeichenkette einschließlich des Dezimalpunktes und der Dezimalstellen. <dez> gibt die Stellen nach dem Dezimalpunkt in der Zeichenkette an. Diese Angaben entsprechen einer Formatspezifikation für die zu erzeugende Zeichenkette.

Ist das angegebene Format zu klein, so wird ein Formatüberlauf durch Ausgabe von Sternen angezeigt. Sind keine Dezimalstellen angegeben, so wird eine Zeichenkette erzeugt, die eine ganze Zahl darstellt. Das Ergebnis ist, entsprechend der Formatangabe, gerundet.

```
* Es wird STR(x,10,0) angenommen
. ? STR(2.71828)
         3
* Es wird STR(x,5,0) angenommen
. ? STR(2.71828,5)
    3
. ? STR(2.71828,5,2)
 2.72
```

SUBSTR — Funktion

Syntax: SUBSTR(<string_ausdr>,<anfangs_pos>[,<anzahl>])

SUBSTR (substring) liefert als Ergebnis eine Zeichenkette. Sie ist ein Teil der Zeichenkette, die durch Berechnung des Stringausdrucks ermittelt wird. <anfangs_pos> und <anzahl> sind numerische Ausdrücke. <anfangs_pos> muß eine positive ganze Zahl liefern, die die Anfangsposition der zu bildenden Teilzeichenkette bestimmt. Beginnend mit diesem Zeichen, gehören fortlaufend so viele Zeichen zur Teilzeichenkette, wie es durch <anzahl> bestimmt wird. Fehlt <anzahl>, so gehören alle Zeichen, beginnend bei der Anfangsposition, bis zum Ende von <string_ausdr> zur Teilzeichenkette. Die gleiche Wirkung hat ein zu großer Wert für <anzahl>. Liegt die Anfangsposition außerhalb der Zeichenkette, so wird eine leere Zeichenkette, deren Länge 0 ist, als Ergebnis geliefert.

```
* Eine Zeichenkette wird einer Speichervariablen zugewiesen
. m_adr = "H1032 Budapest"
H1032 Budapest
* Die Postleitzahl wird herausgezogen ...
. ? SUBSTR(m_adr,2,4)
1032
* ... und der Ort
. ? SUBSTR(m_adr,7,8)
Budapest
```

TIME — Funktion

Syntax: TIME()

TIME liefert die Systemzeit als Zeichenkette. Die Angabe erfolgt in der Form hh:mm:ss, wobei hh für Stunden, mm für Minuten und ss für Sekunden steht.

Vergleicht man die Funktionen TIME und DATE, so ist festzustellen, daß zu TIME keine Konvertierungsfunktionen oder Operationen existieren. Wollen wir die zwischen zwei Zeitpunkten vergangene Zeit ermitteln, so müssen wir umfangreiche Konvertierungen, ausgehend von Zeichenketten, ausführen.

```
. m_start = TIME()
15:50:10
.  m_end = TIME()
16:01:25
* Die folgende Subtraktion liefert nicht die Zeitdifferenz,
* sondern die Verknüpfung von Zeichenketten
. m_end - m_start
15:50:1016:01:25
```

Möchte man die Zeitdifferenz in Sekunden berechnen, so ist das folgende Kommandofile dafür geeignet:

```
* Kommandofile 'time1.prg' zur Berechnung der Zeitdifferenz
* m_end - m_start in Sekunden.
* Umwandlung der Endzeit in numerische Werte
* Numerischer Wert für die Stunden
m_he = VAL(SUBSTR(m_end,1,2))
* Numerischer Wert für die Minuten
m_me = VAL(SUBSTR(m_end,4,2))
* Numerischer Wert für die Sekunden
m_se = VAL(SUBSTR(m_end,7,2))

* Umwandlung der Anfangszeit in numerische Werte
m_hs = VAL(SUBSTR(m_start,1,2))
m_ms = VAL(SUBSTR(m_start,4,2))
m_ss = VAL(SUBSTR(m_start,7,2))
* Berechnung der Zeitdifferenz in Sekunden
m_diff  = ((m_he * 60 + m_me) * 60 + m_se) - ((m_hs * 60 + m_ms) * 60 + m_ss)
```

Der Aufruf dieses Kommandofiles kann nach Bereitstellung der Anfangs- und der Endzeit erfolgen. Eine Möglichkeit dafür ist im folgenden angegeben:

```
* Speichern der Startzeit in einer Speichervariablen
. m_start = TIME()
15:50:10
* Speichern der Endzeit
.  m_end = TIME()
16:01:25
```

```
* Aufruf des Kommandofiles. Die berechneten Zwischenwerte
* werden auf dem Bildschirm ausgegeben.
. DO time1
16.00
 1.00
25.00
15.00
50.00
10.00
                675.00
```

Die Zeitdifferenz beträgt 675 Sekunden. Die Angabe in Sekunden ist jedoch für größere Zeitdifferenzen schlecht einzuschätzen. Hier möchte man die gewohnte Angabe in Stunden, Minuten und Sekunden haben.

```
* Kommandofile 'time2.prg' zur Berechnung der Zeitdifferenz
* m_end - m_start in der Form hh:mm:ss
* Berechnung der Endzeit als numerische Werte
m_he = VAL(SUBSTR(m_end,1,2))
m_me = VAL(SUBSTR(m_end,4,2))
m_se = VAL(SUBSTR(m_end,7,2))

* Berechnung der Startzeit als numerische Werte
m_hs = VAL(SUBSTR(m_start,1,2))
m_ms = VAL(SUBSTR(m_start,4,2))
m_ss = VAL(SUBSTR(m_start,7,2))

IF m_se < m_ss
* übertrag Sekunden --> Minuten
   m_se = m_se + 60
   m_ms = m_ms + 1
ENDIF
* Zeichenkette für die Sekundendifferenz
m_s = STR(m_se - m_ss,2,0)

IF m_me < m_ms
* übertrag Minuten -->Stunden
  m_me = m_me + 60
  m_hs = m_hs + 1
ENDIF
* Zeichenkette für die Minutendifferenz
m_m = STR(m_me - m_ms,2,0)
If m_he < m_hs
* übertrag bei Tageswechsel. Zeitdifferenzen über
* zwei Tageswechsel werden  nicht behandelt.
   m_he = m_he + 24
ENDIF
* Zeichenkette für die Stundendifferenz
m_h = STR(m_he - m_hs,2,0)
* Zusammensetzen der Ergebniszeichenkette
m_tdiff = m_h + ":" + m_m + ":" + m_s
RETURN
```

Der Aufruf dieses Kommandofiles nach Festlegung des Anfangs- und des Endwertes liefert die folgenden Ergebnisse:

```
* Speichern der Startzeit in einer Speichervariablen
. m_start = TIME()
15:50:10
* Speichern der Endzeit
.  m_end = TIME()
16:01:25
* Aufruf des Kommandofiles. Die berechneten Zwischenwerte
* werden auf dem Bildschirm ausgegeben.
. DO time2
16.00
 1.00
25.00
15.00
50.00
10.00
15
        61.00
        16.00
11
 0
 0:11:15
```

Die Zeitdifferenz beträgt 11 Minuten und 15 Sekunden. Die Ausgabe der berechneten Zwischenwerte kann durch SET TALK OFF unterdrückt werden.

TRIM — Funktion

Syntax: TRIM(<string_ausdr>)

TRIM schneidet die Leerzeichen am Ende der Zeichenkette ab. Besaß die Zeichenkette solche Leerzeichen am Ende, so wird sie dadurch verkürzt.

Betrachten wir z.B. ein Datenbankfile, in dem die Felder "Name" und "Vorname" vorkommen und je 20 Zeichen breit sind. Wir möchten den Namen gefolgt von einem Komma, einem Leerzeichen und den Vornamen ausgeben (vgl. Beispiel in Abschn. 4.5).

```
. ? TRIM(name) + ", " + vorname
Kant, Immanuel
```

TYPE — Funktion

Syntax: TYPE("<ausdr>")

TYPE analysiert den Ausdruck und liefert dessen Typ als ein Zei-

chen. In Tafel 4.6 ist die Zuordnung angegeben. Der Ausdruck muß durch Zeichenkettenbegrenzer (", ' oder []) eingeschlossen sein. Wird als Resultat von TYPE "U" geliefert, so ist der angegebene Ausdruck kein gültiger dBASE-Ausdruck. Durch diese Eigenschaft ersetzt TYPE die Funktion TEST aus dBASE II. Besteht der getestete Ausdruck nur aus einer Variablen, so ist das gleichwertig mit der Aussage, daß diese Variable nicht existiert. Man kann so in einem Kommandofile testen, ob eine Variable existiert.

Das folgende Beispiel gibt alle möglichen Fälle wieder. Wir benutzen dazu unser Datenbankfile 'lager'.

```
. USE lager
. ? TYPE("artikel")
C
. ? TYPE("preis")
N
. ? TYPE("bem")
M
. ? TYPE("lieferung")
D
. ? TYPE(".T.")
L
```

Tafel 4.6. Durch TYPE gelieferte Typzeichen

Geliefertes Zeichen	Typ des Ausdrucks
N	numerisch
C	string (character)
L	logisch
D	Datum
M	Memo
U	undefiniert

UPPER — Funktion

Syntax: UPPER(<string_ausdr>)

<string_ausdr> liefert eine Zeichenkette. UPPER konvertiert alle kleinen Buchstaben der Zeichenkette in große Buchstaben. Das Ergebnis ist vom Typ String.

```
. ? UPPER("Ehm-Welk-Straße 57")
 EHM-WELK-STRAßE 57
```

VAL — Funktion

Syntax: VAL(<string_ausdr>)

<string_ausdr> liefert eine Zeichenkette. VAL konvertiert die

angegebene Zeichenkette in einen numerischen Wert. VAL betrachtet die Zeichenkette von links beginnend. Ein Vorzeichen kann wahlweise angegeben werden. Dann wird eine Folge von Ziffern erwartet, der ein Dezimalpunkt folgen kann, und danach können wieder Ziffern folgen. Die Genauigkeit beträgt mindestens zwei Dezimalstellen. Ist die Zahl genauer angegeben, so bleibt ihre Genauigkeit intern erhalten. Bei arithmetischen Operationen wird die volle Genauigkeit berücksichtigt und, in Abhängigkeit von den beteiligten Operanden, auch ausgegeben. Bildet der Anfang der Zeichenkette keine gültige Zahl, so liefert VAL den Wert 0.

```
. m_e = "2.718"
2.718
* Hier erfolgt die Ausgabe mit zwei Dezimalstellen (Standard)
. ? VAL(m_e)
    2.72
. ? 1.000 + VAL(m_e)
     3.718
* Der Wert der Zeichenkette "m_e" ist 0
. ? VAL("m_e")
  0.00
. ? VAL("2")
2.00
. ? VAL("-3.14")
   -3.14
. ? VAL("+22/7.")
    22.00
```

YEAR — Funktion

Syntax: YEAR(<dat_ausdr>)

<dat_ausdr> wird berechnet und liefert einen Wert vom Typ Datum. Mögliche Ausdrücke sind Speichervariable oder Felder vom Typ Datum, der Aufruf der Funktion DATE sowie Additionen oder Subtraktionen mit diesen Datumswerten (vgl. Abschn. 4.6). YEAR liefert von einem solchen Datum die Jahreszahl als vierstelligen numerischen Wert.

```
. m_year = YEAR(DATE())
 1987
. ? TYPE("m_year")
N
. ? YEAR(DATE() + 1000)
 1990
```

5. Erstellen eines Datenbankfiles

Nachdem wir im Abschnitt 3 eine bereits existierende Datenbank besichtigt haben, stellen wir uns hier die Frage: Wie ist eine Datenbank anzulegen? Diese Frage ist keineswegs gleichwertig mit der Frage nach einem Kommando. Im Abschnitt 2.10 gaben wir einige grundlegende praktische Hinweise, die bei der Erstellung einer Datenbank zu beachten sind. Wir wollen diese Untersuchungen hier vertiefen. Zuerst zwei grundsätzliche Hinweise:

1. Bevor man an ein größeres Datenbankprojekt geht, sollte man an einem kleineren Projekt ernsthafte Übungen durchführen. Die Erfahrung zeigt, daß dadurch die Entscheidungsfähigkeit für viele Fragestellungen des zweiten Schritts, einschließlich des vorausschauenden Erkennens der Probleme, wesentlich wächst.

2. Bevor die Arbeit am Computer beginnt, muß man sich über die Zielstellung im klaren sein. Was wollen wir eigentlich erreichen? Welche Anforderungen an Zusammenstellungen, Auszügen, Berichten usw. bestehen jetzt? Welche weiteren Aufgaben könnten in Zukunft entstehen?

Der letzte Punkt ist durchaus nicht so trivial, wie er auf den ersten Blick erscheinen mag. Sehr viele Programmierer von dBASE erstellen dBASE-Programme für Endnutzer. Oft entstehen viele Wünsche der Auftraggeber erst während der Entwicklungsphase. Deshalb sollte man bei der Konzipierung den Blick des Auftraggebers für die bestehenden Möglichkeiten weiten, dagegen aber eine uferlose Ausdehnung des Projekts durch das Fixieren der Zielstellung verhindern.

Ausgehend von der inhaltlichen Zielstellung ist die Struktur der Daten abzuleiten. Wieviel Tabellen lege ich an? Was sind die elementaren Informationen, aus denen ich später alle gewünschten Informationen zusammenstellen kann? Wieviel Spalten hat meine Tabelle? Wie breit müssen die Spalten sein, und von welchem Typ sind die Informationen? Wieviel Tabellen sind sinnvoll? Wie sollen die Daten erfaßt werden? Bestehen Unterschiede zwischen der Ersterfassung und der laufenden Erfassung? Nach welchen Ordnungskriterien soll die Datenbank betrachtet werden?

Mitunter fällt die Entscheidung zwischen den Typen String und numerisch schwer. Nicht alle Felder, die nur Ziffern enthalten, sind vom numerischen Typ, z.B. Telefonnummern. Als Faustregel gilt hier: Können mit einer Größe sinnvoll arithmetische Operationen durchgeführt werden, so ist der Typ numerisch zu wählen. Die Addition von Kontonummern, Postleitzahlen, Personenkennzahlen u.ä. ergibt sicher keinen Sinn. Das sind Größen vom Typ String. Die Anzahl der Kinder einer Person sollte hingegen eine numerische Größe sein, auch wenn vorerst keine arithmetischen Operationen geplant sind. Es ist nämlich denkbar, daß irgendwann die Anzahl aller Kinder ermittelt werden soll.

Hat man sich trotz sorgfältiger Analyse des Problems doch für den

falschen Typ oder die falsche Spaltenbreite entschieden, so sind im Einzelfall die Konvertierungsfunktionen anwendbar (Abschn. 4.7 und Anhang 1). Handelt es sich um einen prinzipiellen Fehler, so können durch eine Strukturänderung des Datenbankfiles (MODIFY STRUCTURE), die eine Typänderung von Feldern beinhaltet, bzw. eine Veränderung der Spaltenbreite die Daten konvertiert werden.

Ist die logische Struktur der Daten ermittelt, so sind die benötigten Algorithmen festzulegen. Sie werden später als Kommandofiles realisiert. Wir fragen uns also: Welche Operationen sind gewünscht? Lassen sich alle benötigten Informationen aus der Datenbank gewinnen? Das beinhaltet einen Konsistenztest für die vorher getroffenen Festlegungen über die Daten.

Die Güte einer Datenbank erweist sich im wesentlichen darin, wie sie den Änderungen (Gerätetechnik, Datenbanksystem) und den sich ändernden Wünschen der Nutzer gewachsen ist. Wir sagen: Sie muß flexibel sein. Damit ist, mehr als in den bekannten Programmiersprachen, eher eine klare und verständliche Strategie gefragt als eine trickreiche Programmierung.

Haben wir diese Entwurfsphase erfolgreich absolviert, so können wir unser Konzept in dBASE realisieren.

5.1. Definition der Struktur

Wir wollen das Datenbankfile anlegen, welches wir im Abschnitt 3 bereits im Vorgriff benutzt haben. Der Kopf der zu realisierenden Tabelle wird in konventioneller Form im Bild 5.1 wiedergegeben.

Artikel	*Anzahl*	*Preis*	*Min.*	*Lieferung*	*Hersteller*	*Bem.*
15 Zeichen	*4*	*8,2*	*2*	*8*	*12*	*ca. 50*
c	*n*	*n*	*n*	*d*	*c*	*text*

Bild 5.1. Festlegen des Tabellenkopfes in gewöhnlicher Form

Definition der Struktur einer Datenbank heißt in dBASE, gerade diesen Tabellenkopf festzulegen. Wir haben dabei für jede Tabellenspalte nur drei Angaben zu machen: Wie lautet die Überschrift der Spalte? Das wird der Name dieses Feldes. Von welchem Typ sind die Inhalte dieser Spalte? Wie breit ist die Spalte? Haben wir als Typ numerisch gewählt, so können wir eine Anzahl von Stellen nach dem Dezimalpunkt festlegen. Bild 5.1 enthält alle erforderlichen Infor-

mationen.

CREATE ist das Kommando, mit dem die Struktur eines Datenbankfiles definiert wird. Man beachte, daß man sich durch HELP oder ASSIST von dBASE beim Auffinden des richtigen Kommandos helfen lassen kann! Geben wir CREATE ohne einen Filenamen an, so fordert CREATE diesen an:

```
. CREATE
Name der neuen Datei eingeben: lager
```

Danach geht die sequentielle Bedienung des Bildschirms in den Direktmodus (full screen mode) über. Es erscheint Bild 5.2.

```
C:lager.dbf                                    Restliche BYTES :   4000
                                               Felder definiert:      0

 CURSOR : <-- -->   | Einfügen        | Löschen         | Feld auf    :  ↑
  Zeich.:  ← →      | Zeich.: Ins     | Zeich.: Del     | Feld ab     :  ↓
  Wort  :Home End   | Feld  : ^N      | Wort  : ^Y      | Ende/Sichern: ^End
  Spalte:  ^← ^→    | HILFE : F1      | Feld  : ^U      | Abbruch     :  Esc

    Feld Name   Typ       Länge  Dez          Feld Name              Länge  Dez
 1  ▒▒▒▒▒▒▒▒▒▒  Zeich_Fld  ▒▒▒▒  ▒▒▒▒

 Namen beginnen mit Buchstaben; sonst Buchstabe, Zahl oder _ Zeichen
```

Bild 5.2. CREATE, Anfangssituation

Das Hilfsmenü ist in diesem Fall eingeschaltet. Der Kursor steht im ersten Feld. Feldbezeichner dürfen maximal aus 10 Zeichen bestehen. Die Eingabe eines Feldbezeichners wird mit **Enter** beendet. Danach springt der Kursor auf das Feld 'Typ'. 'Zeich_Fld' bezeichnet hier den Typ String. Man kann jetzt einen der Anfangsbuchstaben der Typbezeichner (c, n, l, d oder m) angeben. Da sich alle Typbezeichner im ersten Buchstaben unterscheiden, kann der Typ nach Eingabe eines Zeichens selbständig erkannt werden. dBASE ergänzt den Typbezeichner entsprechend. Befindet sich der Kursor im Feld 'Typ', so kann man sich durch Betätigen der Leertaste nacheinander alle Typen anbieten lassen. Wird der gewünschte Typ angezeigt, so bestätigt man diese Angabe mit **Enter**. Der Kursor springt zum Feld 'Länge'. Für Felder vom Typ Datum oder Memo steht die Länge fest. Der Kursor geht sofort zur nächsten Zeile über (Definition des nächsten Feldes). Ist der Typ numerisch, so wird der Kursor nach Eingabe der Länge auf das Feld 'Dez' (Dezimalstellen) positioniert. Ist ein Feld vollständig definiert, so springt der Kursor auf den Anfang des nächsten Feldes. Die Definition wird beendet, indem kein Feldbezeichner eingegeben wird oder durch **^End**.

Der Bildschirminhalt nach der Definition aller Felder ist im

Bild 5.3 wiedergegeben.

```
C:lager.dbf                                    Restliche BYTES :   3941
                                               Felder definiert:      7

 CURSOR : <-- -->   Einfügen      Löschen      Feld auf    :  ↑
  Zeich.:   ← →     Zeich.: Ins   Zeich.: Del  Feld ab     :  ↓
  Wort  :Home End   Feld  : ^N    Wort  : ^Y   Ende/Sichern: ^End
  Spalte:  ^← ^→    HILFE : F1    Feld  : ^U   Abbruch     :  Esc

     Feld Name   Typ        Länge  Dez          Feld Name   Typ        Länge  Dez
  1  ARTIKEL     Zeich_Fld    15
  2  ANZAHL      Numerisch     4    0
  3  PREIS       Numerisch     8    2
  4  MINIMUM     Numerisch     2    0
  5  LIEFERUNG   Datum         8
  6  HERSTELLER  Zeich_Fld    12
  7  BEM         MEMO         10
  8  ▒▒▒▒▒▒▒▒▒▒  Zeich_Fld   ▒▒

Namen beginnen mit Buchstaben; sonst Buchstabe, Zahl oder _ Zeichen
```

Bild 5.3. CREATE nach der Definition der Felder

Bei Bedarf erfolgt die Definition der Tabelle in zwei breiten Spalten. Die entstandene Tabellenstruktur kann man sich über LIST STRUCTURE ansehen. Sie ist damit vollständig definiert. Wir können Daten eingeben.

5.2. Eingabe der Daten

Beendet man die Definition der Struktur, so fragt dBASE, ob man sofort Daten eingeben möchte. Wird die Frage mit 'j' oder 'J' (für ja) beantwortet, so erscheint das Bild 5.4, das im Direktmodus bedient werden kann, auf dem Bildschirm. Die Doppelpunkte sind als Begrenzer der Eingabefelder ausgewählt worden (SET DELIMITER ON). Standardmäßig werden diese Felder ohne Begrenzer und invers dargestellt. Ihre Farbe kann durch SET COLOR verändert werden.

Der Kursor kann nur innerhalb dieser Felder bewegt werden. Wird das Ende eines Feldes erreicht, so springt der Kursor auf das erste Zeichen des nächsten Feldes. Felder, die nicht vollständig gefüllt werden, können mit **Enter** verlassen werden. Bleibt ein Feld frei, so wird ein Standardwert angenommen. Das sind für String-Felder Leerzeichen, für numerische Felder der Wert 0; für logische Felder erscheint ein Fragezeichen "?", das als Wahrheitswert falsch interpretiert

wird; Felder vom Typ Datum werden mit Leerzeichen gefüllt, und Memo-Felder verweisen auf keinen Text.

Wird das Ende des letzten Feldes erreicht, so erscheint ein leeres Formular für den nächsten Datensatz.

Bemerkt man bei der Eingabe Fehler, so kann der Kursor frei innerhalb der Felder und über Feldgrenzen hinweg (aber immer innerhalb der Doppelpunkte) bewegt werden. **PgUp** führt uns zum vorherigen Formular, **PgDn** zum nächsten Formular.

Die Eingabe der Daten wird durch ein leeres erstes Feld oder durch **^End** beendet. Es besteht jederzeit die Möglichkeit, Daten zu korrigieren (EDIT, BROWSE), Sätze zu streichen (DELETE), Sätze einzufügen (INSERT) oder neue Sätze anzufügen (APPEND). Trotzdem sollte die Dateneingabe sehr sorgfältig erfolgen, da einige Fehler u.U. sehr schwer auffindbar sind.

Das auf dem Bildschirm dargestellte Formular für die Eingabe von Daten ist eine Standardform, die aus den Angaben der Struktur der Tabelle ermittelt wird. Man kann sich vorstellen, daß ein Bildschirminhalt eine Karteikarte darstellt. Ein Übergang zum nächsten (**PgDn**) oder zum vorherigen (**PgUp**) Datensatz kann wie das Blättern in einem Karteikasten verstanden werden. Um diese Eingabebilder noch mehr gewöhnlichen Karteikarten ähnlich zu machen, hat man die Möglichkeit, Bildschirmmasken zu definieren. Näheres dazu findet man im Abschnitt 13.2.

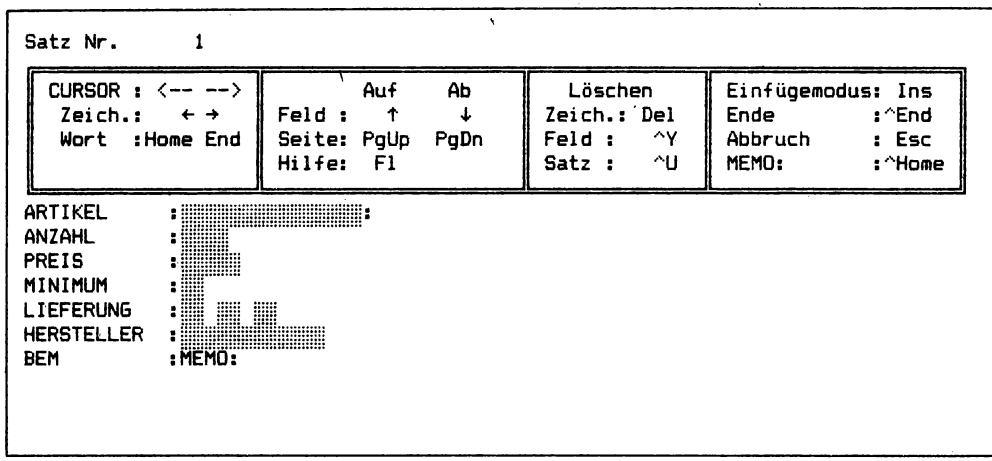


Bild 5.4. Bildschirminhalt bei der Eingabe im Direktmodus

6. Ergänzen und Verändern von Daten

Aus den verschiedensten Gründen ist es erforderlich, die Inhalte der Datensätze zu verändern, zu ergänzen, neue Sätze hinzuzufügen oder überflüssige zu streichen. All diese Änderungen beziehen sich auf den Inhalt des Datenbankfiles, nicht auf seine Struktur. Änderungen der Struktur betrachten wir im nächsten Abschnitt.

6.1. Anfügen von Sätzen

Neue Datensätze können von der Tastatur aus angefügt werden, oder sie sind bereits in einem anderen File gespeichert. Ein solches File, aus dem Sätze an eine Datenbank angefügt werden, kann ein anderes Datenbankfile oder ein Textfile sein. Die verschiedenen Formen des Kommandos APPEND sind dafür geeignet.

6.1.1. Anfügen von der Tastatur

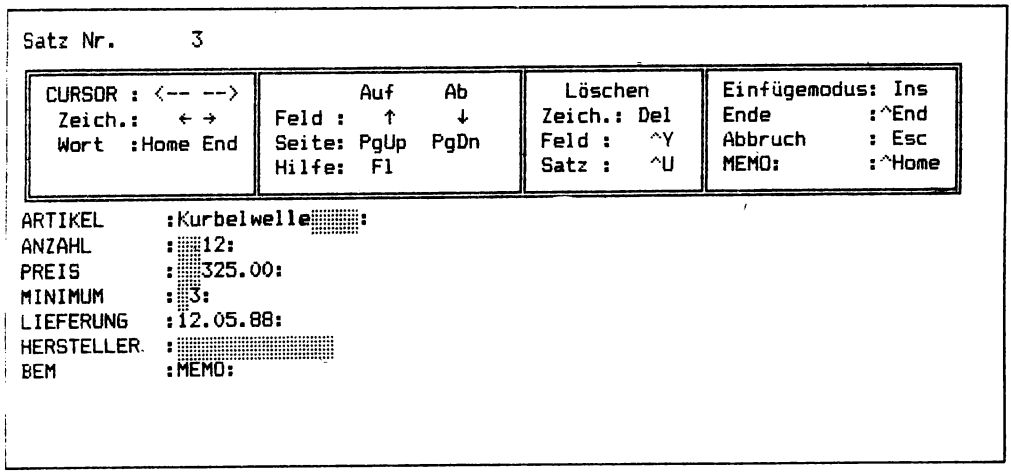


Bild 6.1. Bildschirminhalt beim Anfügen eines Datensatzes von der Tastatur

Zum Anfügen von Datensätzen an das Ende des Datenbankfiles von der Tastatur aus benutzt man das Kommando APPEND ohne weitere Parameter. Die Bildschirmbedienung geht vom sequentiellen in den Direktmodus über. Es erscheint ein leeres Formular mit der Nummer des nächsten freien Satzes. Die Eingabe der Daten erfolgt genauso wie im Ab-

schnitt 5 beschrieben, d.h., der Kursor kann nur innerhalb der Doppelpunkte (im Normalfall sind die Eingabebereiche invers hervorgehoben, dafür fehlen die Doppelpunkte), aber dort frei positioniert werden. Korrekturen innerhalb eines Satzes und in den vorangehenden bzw. den nachfolgenden Sätzen sind möglich. dBASE geht dann selbständig in den Edit-Modus über (Abschn. 6.3). Bild 6.1 gibt die Situation wieder, wie sie beim Anfügen des dritten Satzes und nach der Eingabe einiger Werte vorliegt. Aus dem Hilfsmenü ist die Verwendung der Kursorpositionierungstasten ersichtlich.

Die Eingabe neuer Datensätze wird durch ^**End** oder durch Betätigen der Taste **Enter** auf dem ersten Zeichen des ersten Feldes eines leeren Datensatzes beendet. dBASE kehrt zurück in die sequentielle Bedienung des Bildschirms.

6.1.2. Anfügen eines Datenbankfiles

Nehmen wir an, daß zwei Tabellen (Datenbankfiles) gleicher Struktur vorliegen. Wir möchten die Datensätze des einen Datenbankfiles an das andere Datenbankfile anfügen. Das entspricht anschaulich dem Fall, daß beide Tabellen aneinandergeklebt werden. Da die Struktur gleich ist, treten keinerlei Probleme auf. In dBASE benutzt man dazu die folgende Kommandofolge:

```
* Erstes Datenbankfile in Benutzung nehmen
.USE datei1
* Anfügen aller Datensätze des zweiten Datenbankfiles
.APPEND FROM datei2
```

Was geschieht aber, wenn die Felder unterschiedliche Breite haben, wenn die Typen oder Namen nicht übereinstimmen oder wenn Felder in der Zieldatenbank nicht existieren, die in dem anzufügenden Datenbankfile vorkommen und umgekehrt? Das entstehende Format wird in jedem Fall durch das Zieldatenbankfile bestimmt. Nehmen wir zuerst an, daß die Namen übereinstimmen.

Unterschiedliche Feldbreiten

Nur numerische Felder und Stringfelder können unterschiedliche Breite haben. Ist ein Stringzielfeld breiter, so wird das entsprechende Quellfeld durch Leerzeichen nach rechts ergänzt. Ist ein Stringzielfeld schmaler, so werden nur soviel Zeichen des Quellfeldes von links beginnend übernommen, wie im Zielfeld abgelegt werden können. Es erfolgt keine Warnung, falls Zeichen dabei verschwinden, die keine Leerzeichen sind.

Sind zwei Felder vom numerischen Typ und ist das Zielfeld breiter, so werden die Werte des Zielfeldes übernommen. Führende Nullen werden nicht angezeigt. Ist das Zielfeld schmaler und kann ein Quellwert im Zielfeld nicht dargestellt werden, so wird das Zielfeld mit Sternen '****' gefüllt (Formatüberlauf). Es erfolgt keine Fehlermitteilung. Beim Zugriff auf ein solches Feld wird der Wert 0 geliefert. Unter-

scheiden sich die numerischen Felder in der Anzahl ihrer Dezimalstellen, so erfolgt eine Rundung bzw. es werden Nullen als Dezimalstellen angefügt.

Unterschiedliche Namen

Existiert in dem Zieldatenbankfile ein Feld, welches in dem Quellfile nicht vorkommt, so wird für jeden Satz, der angefügt wird, sein Inhalt initialisiert. Für ein numerisches Feld wird der Wert 0 eingetragen (für String- und Datumsfelder sind das Leerzeichen) für ein logisches Feld wird ein Fragezeichen eingetragen und als Wert 'falsch' interpretiert, und Memo-Felder verweisen auf keinen Text.

Existiert ein Feld in der Quelldatei, aber nicht in der Zieldatei, so werden dessen Werte nicht übernommen. Ob ein Feld existiert oder nicht, wird an gleichen Feldnamen erkannt. Die Reihenfolge der Felder in den Datenbankfiles ist dabei unwichtig.

Unterschiedliche Typen

Existieren in beiden Dateien Felder mit gleichem Namen, aber von unterschiedlichem Typ, so findet eine Typanpassung statt. Zusätzlich zur Typanpassung wirken die zuvor beschriebenen Anpassungsregeln. Offenbar sind nicht alle Kombinationen von Ziel- und Quelltypen sinnvoll. dBASE akzeptiert aber jede Kombination. Die wichtigsten Typkonvertierungen sind in der Tafel 6.1 angegeben. Den Typ Memo können wir dabei außer acht lassen, denn er ist zu keinem anderen Typ zuweisungskompatibel.

Sehen wir uns diese Eigenschaften von dBASE an einem Beispiel an. Die Zieldatei sei unser Datenbankfile lager.dbf. Es sei mit drei Sätzen gefüllt. Außerdem existiert ein Datenbankfile lager2.dbf, dessen Datensätze wir an das File 'lager.dbf' anfugen möchten. Die Struktur der Zieldatei ist uns bekannt. Die Struktur von lager2.dbf besichtigen wir durch

```
. LIST STRUCTURE
Datenbankstruktur       - C:lager2.dbf
Anzahl der Datensätze   -        2
Letztes Änderungsdatum - 06.12.87
Feld   Feldname    Typ         Länge    Dez
    1  ARTIKEL     Zeichen       18
    2  ANZAHL      Zeichen        5
    3  PREIS       Numerisch      6      2
    4  LIEFERUNG   Zeichen        8
** Gesamt **                     38
```

Es fehlen die Felder 'Minimum', 'Hersteller' und 'Bem'. Außerdem stimmen einige Typen und Breiten nicht mit denen im Datenbankfile 'lager.dbf' überein. So hat 'Artikel' hier 18 Zeichen, sonst 15. 'Anzahl' ist vom Typ String und hat die Breite 5, sonst vom Typ numerisch mit der Breite 4. 'Preis' hat die Breite 6, sonst 8. 'Lieferung ist hier vom Typ String, sonst vom Typ Datum.

Tafel 6.1. Typanpassungen in dBASE III

Zieltyp	Quelltyp	Wirkung
numerisch	String	Die Zeichenkette wird von links beginnend als Zahl interpretiert. Der ermittelte Wert wird dem Zielfeld zugewiesen.
numerisch	Datum	Das Datum wird wie eine Zeichenkette behandelt.
numerisch	logisch	Liefern den Wert 0.
String	numerisch	Die Darstellung des Wertes wird dem Zielfeld als Zeichenkette zugewiesen, d.h., es wird implizit die Funktion STR angewendet.
String	logisch	Die Buchstaben T für wahr, F für falsch und ein Leerzeichen für einen undefinierten Wert, (?) werden dem Zielfeld zugewiesen.
String	Datum	Das Datum wird als Zeichenkette interpretiert.
datum	String	Stellen die ersten acht Zeichen der Zeichenkette ein gültiges Datum dar, so wird es der Datumsvariablen zugewiesen. Andernfalls ist das Zielfeld leer.
datum	sonst	Alle anderen Kombinationen ergeben keinen Sinn und liefern ein leeres Datum.
logisch	String	Ist das erste Zeichen ein T, t, F oder f, so wird der entsprechende Wahrheitswert gebildet. In allen anderen Fällen reagieren die getesteten Versionen unterschiedlich. So liefert die deutsche dBASE-Version 1.10 für Y und y den Wahrheitswert wahr, für J und j hingegen falsch.
logisch	sonst	Die anderen Kombinationen ergeben keinen Sinn. Sie liefern undefinierte Werte.

Die zwei anzufügenden Datensätze haben den folgenden Inhalt:

```
. LIST
 Satz #  ARTIKEL          ANZAHL  PREIS LIEFERUNG
      1  Luftleitblech    14        5.40 10.11.88
      2  Fußdichtung      122       0.15 18.09.88
```

Um ein anderes File an ein Datenbankfile anfügen zu können, muß dieses Datenbankfile aktiv sein. Deshalb nehmen wir 'lager' in Benutzung und fügen dann das Datenbankfile 'lager2' durch APPEND an:

```
. USE lager
. APPEND FROM lager2
      2 Sätze addiert
```

Das Datenbankfile 'lager' enthält danach die folgenden Datensätze:

```
. LIST
Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
     1  Flachschieber        112     2.40      15 29.02.88 C&C          MEMO
     2  Kurbelwelle I         16   360.00       3 12.05.88 Achse&Welle  MEMO
     3  Kurbelwelle II        12   325.00       3 12.05.88              MEMO
     4  Luftleitblech         14     5.40         10.11.88              MEMO
     5  Fußdichtung          122     0.15         18.09.88              MEMO
```

Wir beobachten die Formatanpassungen. Die Felder 'Minimum' und 'Hersteller' werden mit Leerzeichen initialisiert. 'Bem' bleibt ebenfalls leer. Memo-Felder können nur im Direktmodus von der Tastatur aus gefüllt oder verändert werden. In allen Fällen fand eine Anpassung der Breite statt. Bei 'Anzahl' erfolgte eine Anpassung an den Typ numerisch (sichtbar dadurch, daß bei numerischen Größen die Werte rechtsbündig stehen). Für das Feld 'Lieferung' wurde eine Konvertierung in den Typ Datum vorgenommen.

6.1.3. Anfügen eines Textfiles

Mitunter möchte man in ein Datenbankfile Informationen aufnehmen, die bereits vorher und nicht durch ein Datenbanksystem erfaßt wurden. Wir nehmen an, daß die folgende Tabelle mit einem Textverarbeitungssystem erstellt und im File 'teile.txt' gespeichert wurde:

Ersatzteil	Anz.	Preis
Luftleitblech	20	5.4
Kopfdichtungen	55	0.15
Formschlauch	8	1.55

Die Tabelle besteht aus drei Spalten, die in ihrer Breite mit den ersten drei Spalten unseres Datenbankfiles 'lager.dbf' übereinstimmen (15, 4 und 8 Zeichen). Wir entfernen den Tabellenkopf einschließlich der Unterstreichung, z.B. mit einem Texteditor, und fügen dieses File an das Datenbankfile 'lager.dbf'an, welches wieder nur drei Sätze enthalten soll. Das darauffolgende LIST-Kommando zeigt uns das Resultat.

```
. USE lager
. APPEND FROM teile SDF
      3 Sätze addiert
```

```
. LIST
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
      1  Flachschieber        112     2.40      15 29.02.88  C&C          MEMO
      2  Kurbelwelle I         16   360.00.      3 12.05.88  Achse&Welle  MEMO
      3  Kurbelwelle II        12   325.00       3 12.05.88               MEMO
      4  Luftleitblech         20     5.4                                MEMO
      5   Kopfdichtungen       55     0.15                               MEMO
      6  Formschlauch          8     1.55                               MEMO
```

Zuerst bemerken wir, daß die in der Texttabelle gewählten Überschriften ohne Bedeutung sind. Einen Unterschied in der Kommandosyntax finden wir in dem Wort SDF (standard data format). Es bewirkt, daß das nach FROM angegebene File, wenn keine Namenserweiterung angegeben ist, als Textfile mit der Namenserweiterung .txt betrachtet wird.

Es werden drei neue Datensätze angefügt. Sie enthalten je eine Zeile unserer Tabelle. Allgemein gilt: Aus jeder Zeile des Textfiles entsteht ein Datensatz. Die Struktur unseres Datenbankfiles wurde wie eine Schablone über den Text gelegt. Die ersten 15 Zeichen einer Zeile werden dem ersten Feld zugewiesen. Die nachfolgenden vier Zeichen werden positionsgerecht im Feld 'Anzahl' dargestellt. Die folgenden acht Zeichen werden in gleicher Weise dem Feld 'Preis' zugeordnet. Nur wenn die Position exakt eingehalten wird und die Zeichenketten als Zahlen interpretierbar sind, enthalten numerische Felder den richtigen Wert.

Die nachfolgenden Felder werden initialisiert, da eine Zeile des Textfiles 'teile.txt' nicht mehr Zeichen enthält.

Beim Anfügen von Textfiles ist zu beachten, daß in der Texttabelle die Spalten streng eingehalten werden müssen. In unserem Beispiel sehen wir im zweiten Satz des Textfiles ein Leerzeichen am Anfang der Zeile. Es wird in das Datenbankfile übernommen. Beim Besichtigen der Datenbank fällt dieses Leerzeichen lediglich als 'Unschönheit' ins Auge. Es hat für die Datenbank jedoch erhebliche Konsequenzen. Nehmen wir an, wir möchten eine Zusammenstellung aller Artikel haben, die mit "Kopf" beginnen.

```
.LIST FOR artikel="Kopf"
```

liefert uns für dieses Datenbankfile keinen Datensatz, denn es gilt:

```
"Kopf" <> " Kopf"
```

Das wirkt sich auch beim Sortieren, Indizieren, kurz, bei allen Operationen, die auf dem Zeichenkettenvergleich beruhen, aus.

In der dritten Zeile steht der Preis nicht rechtsbündig. Wir sehen, daß auch im Feld 'Preis' der Wert nicht rechtsbündig steht. Es wird jedoch der korrekte Wert geliefert. Das gleiche gilt für das Feld 'Anzahl' des sechsten Satzes.

```
.GO 6
.? preis
   1.55
. ? anzahl
   8
```

Im vierten Satz fehlt die zweite Dezimalstelle, d.h., sie wurde beim Anhängen nicht ergänzt.

```
.GO 4
.? preis
   5.40
```

Betrachten wir nun den Fall, daß eine Tabelle als Textfile vorliegt, die Spalten exakt ausgerichtet sind, jedoch in ihrer Anordnung, Breite und Anzahl nicht denen des Datenbankfiles entsprechen. Das Ziel besteht darin, diese Texttabelle an unser Datenbankfile 'lager.dbf' anzufügen. Gegeben sei die folgende Tabelle:

Nr.	Ersatzteil	Preis	Anz.
1.	Luftleitblech	5.40	20
2.	Kopfdichtungen	0.15	55
3.	Formschlauch	1.55	8

Die laufende Nummer nimmt vier Zeichen ein. Für die Bezeichnung des Ersatzteils werden 18 Zeichen benutzt, für Preis sind 10 Zeichen mit zwei Dezimalstellen reserviert und für die Anzahl fünf Zeichen. Das Anfügen der Texttabelle an unser Datenbankfile durch APPEND würde nicht das gewünschte Resultat liefern. Wir beschreiten deshalb den folgenden Weg:

1. Kopie der Texttabelle erstellen und die Überschrift einschließlich Unterstreichung löschen.

2. Erstellen eines Datenbankfiles mit genau der gleichen Struktur, wie sie das Textfile aufweist, und den Feldnamen des Datenbankfiles 'lager.dbf', soweit erforderlich.

3. Anfügen des Textfiles an dieses leere Datenbankfile.

4. Anfügen dieses (Hilfs-)Datenbankfiles 'teile.dbf' an das Datenbankfile 'lager.dbf'.

Für den ersten Schritt wird ein beliebiger Texteditor benutzt. Über CREATE wird ein Datenbankfile mit dem Namen 'teile.dbf' erstellt, das die folgende Struktur hat:

```
. LIST STRUCTURE
Datenbankstruktur      - C:teile.dbf
Anzahl der Datensätze  -        0
Letztes Änderungsdatum - 12.12.87
Feld   Feldname    Typ          Länge    Dez
   1   NR          Zeichen         4
   2   ARTIKEL     Zeichen        18
   3   PREIS       Numerisch      10      2
   4   ANZAHL      Numerisch       5
** Gesamt **                      38
```

Der dritte Schritt wird wie folgt realisiert und das Ergebnis besichtigt:

```
. APPEND FROM teile2 SDF
     3 Sätze addiert
. LIST
 Satz #  NR   ARTIKEL                PREIS ANZAHL
      1   1.  Luftleitblech          5.40     20
      2   2.  Kopfdichtungen         0.15     55
      3   3.  Formschlauch           1.55      8
```

Das Feld 'Nr' interessiert uns nicht weiter. Sein Inhalt wird nicht übernommen. Da wir immer an das aktive Datenbankfile etwas anfügen können, nehmen wir 'lager.dbf' in Benutzung.

```
. USE lager
* Es wird das Datenbankfile 'teile.dbf' angefügt, da SDF nicht angegeben wurde.
. APPEND FROM teile
     3 Sätze addiert
* Besichtigen des entstandenen Datenbankfiles
. LIST
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
      1  Flachschieber       112     2.40      15 29.02.88  C&C          MEMO
      2  Kurbelwelle I        16   360.00       3 12.05.88  Achse&Welle  MEMO
      3  Kurbelwelle II       12   325.00       3 12.05.88               MEMO
      4  Luftleitblech        20     5.40                                MEMO
      5  Kopfdichtungen       55     0.15                                MEMO
      6  Formschlauch          8     1.55                                MEMO
```

Die in diesem Abschnitt beschriebenen Verfahren geben uns die Möglichkeit, nahezu beliebige Tabellen, die im Textformat vorliegen und spaltengerecht geschrieben sind, an ein Datenbankfile anzufügen. Solche Textfiles können von der Tastatur aus eingegeben worden sein, sie können aber auch von einem anderen Programm erzeugt worden sein, etwa als Ausgabe eines Pascal-Programms oder als Ausgabe von SuperCalc, MultiPlan oder einem anderen Datenbanksystem.

Beim Kommando COPY werden wir sehen, daß wir eine Datenbank auch in ein Textfile ausgeben können. Wir halten deshalb fest:

Textfiles stellen die Schnittstelle für den Datenaustausch mit anderen Systemen dar.

6.2. Einfügen von Sätzen

Sind in einem Datenbankfile die Sätze nach einem bestimmten Kriterium sortiert, so möchte man diese Ordnung auch beim Hinzufügen von Sätzen erhalten. Zwei Wege sind gebräuchlich, die abhängig von der Größe des Datenbankfiles und der Anzahl der einzufügenden Sätze gewählt werden. Ist die Anzahl der einzufügenden Sätze im Verhältnis zur Anzahl der bereits existierenden Sätze groß, so wird man die neuen Sätze über APPEND anfügen und danach die Ordnung durch SORT wiederherstellen. Werden nur wenige Sätze eingefügt, so ist es sinnvoll, diese gleich an der richtigen Stelle einzufügen. Dazu benutzt man das Kommando INSERT.

Der Dateizeiger zeige auf den Satz mit der Nummer n. INSERT bewirkt, daß ein neuer Datensatz nach dem Satz mit der Nummer n eingefügt wird. Der Dateizeiger zeigt auf diesen neuen Satz. Ein leeres Formular wird auf dem Bildschirm angezeigt. Für diesen Datensatz können die Daten genauso wie bei APPEND im Direktmodus eingegeben werden. Ist die Eingabe für diesen einen Satz beendet, wechselt die Eingabe vom Direktmodus in den sequentiellen Modus. Ein neues Kommando kann eingegeben werden.

INSERT BEFORE

fügt einen neuen Datensatz vor dem aktuellen Datensatz ein. Ansonsten ist die Wirkung die gleiche wie bei INSERT.

Möchte man einen leeren Datensatz einfügen und nicht in den Direktmodus übergehen, so kann man das durch

INSERT BLANK

erreichen. Für die Arbeit in Kommandodateien kann das sehr nützlich sein. Die Daten können z.B. durch REPLACE eingefügt werden.

6.3. Korrektur des Inhalts

Die Korrektur des Inhalts eines Datensatzes ist durch das Kommando EDIT möglich. EDIT geht in den Direktmodus der Bildschirmbedienung über. Es entsteht genau das gleiche Bild, wie wir es vom Kommando APPEND kennen (Bild 6.1). Der Kursor ist innerhalb der Datenfelder, hier durch Doppelpunkte eingeschlossen, frei beweglich. Er kann von Feld zu Feld bewegt werden. Ein "Blättern in den Karteikarten" ist durch die Tasten **PgUp** (rückwärts) und **PgDn** (vorwärts) möglich. Die Bedeutung der Steuertasten ist aus dem Hilfsmenü des Bildes 6.1 ersichtlich.

Gibt man nur EDIT (ohne Parameter) ein, so wird der aktuelle Datensatz zur Korrektur angeboten. EDIT n bietet den Datensatz mit der Nummer n an. Erreicht man bei der Korrektur das Ende der Datenbank, so kehrt man vom Direktmodus in den sequentiellen Modus zurück. Alle Änderungen in den Datensätzen sind gültig. Die Wirkung ist die

gleiche wie bei der Eingabe von ^**End**.

EDIT empfiehlt sich außer zur Korrektur auch zur Besichtigung von Datensätzen. Das ist insbesondere dann vorteilhaft, wenn die Länge des Datensatzes die Länge einer Bildschirmzeile überschreitet. LIST und DISPLAY setzen zu lange Datensätze auf den folgenden Bildschirmzeilen fort. Dadurch wird die Anzeige sehr unübersichtlich. EDIT bietet hingegen immer das gleiche Bild in Form einer Karteikarte.

6.4. Schnellkorrektur

Die Inhalte der Datensätze können auch durch das Kommando BROWSE im Direktmodus korrigiert werden. BROWSE stellt die gesamte Datenbank in Tabellenform dar, d.h., die Felder werden nebeneinander angeordnet und die Sätze untereinander. Mit dem Kursor können wir uns frei über alle Zeichen, Felder und Datensätze bewegen. Die Bedeutung der Kursortasten ist im Hilfsmenü des Bildes 6.2 wiedergegeben.

Satz Nr. 1 lager

CURSOR : <-- -->	Auf Ab	Löschen	Einfügemodus: Ins
Zeich.: ← →	Satz : ↑ ↓	Zeich.: Del	Ende : ^End
Feld :Home End	Seite: PgUp PgDn	Feld : ^Y	Abbruch : Esc
Spalte: ^← ^→	HILFE: F1	Satz : ^U	Optionen : ^Home

ARTIKEL--------	ANZAHL	PREIS---	MINIMUM	LIEFERUNG	HERSTELLER--	BEM-
Flachschieber	112	2.40	15	29.02.88	C&C	MEMO
Kurbelwelle I	16	360.00	3	12.05.88	Achse&Welle	MEMO
Kurbelwelle II	12	325.00	3	12.05.88	Achse&Welle	MEMO
Formschlauch	41	1.55	4	28.07.88	Gummi&Sohn	MEMO
Kühlluftgehäuse	8	16.60	2	23.11.88	VEB Guß	MEMO
Luftleitblech	14	5.40	2	10.11.88	Blech&Co	MEMO
Kopfdichtungen	116	0.15	30	17.08.88	Dicht&Fest	MEMO
Krümmerdichtung	88	0.14	30	18.09.88	Dicht&Fest	MEMO
Fußdichtung	122	0.15	30	07.06.88	Dicht&Fest	MEMO

Bild 6.2. Bildschirminhalt bei der Schnellkorrektur (BROWSE)

Nach Eingabe des Kommandos BROWSE steht der Kursor auf dem ersten Zeichen des ersten Feldes. Der Datensatz, in dem sich der Kursor befindet, jetzt eine Zeile, wird invers hervorgehoben.

Bei größeren Datenbanken kann der Bildschirm nur einen Ausschnitt der Tabelle wiedergeben. Durch gleichzeitiges Drücken der Tasten **Ctrl** und der Kursortaste nach rechts (→) rückt der Bildschirm ein Feld nach rechts, durch ^← entsprechend um ein Feld nach links, wenn sich dort jeweils noch ein Feld befindet. ←, →, Home und End bewegen nur den Kursor, verschieben aber nicht den angezeigten Bildschirmausschnitt.

Im Unterschied zum EDIT-Menü bemerken wir ein Angebot, durch ^**Home** Optionen auswählen zu können. Bild 6.3 zeigt den Bildschirm nach

Eingabe von ^**Home**.

```
 Ende          Anfang          Stoppen          Satz Nr.          Sperren

 CURSOR : <-- -->     Auf   Ab     Löschen      Einfügemodus:  Ins
  Zeich.:   ← →      Satz :  ↑    ↓  Zeich.: Del  Ende       : ^End
  Feld  :Home End    Seite: PgUp  PgDn Feld  : ^Y  Abbruch    : Esc
  Spalte:  ^← ^→     HILFE: F1         Satz  : ^U  Optionen   : ^Home

ARTIKEL-------- ANZAHL PREIS--- MINIMUM LIEFERUNG HERSTELLER-- BEM-
Flachschieber▒▒▒  ▒▒112 ▒▒▒▒2.40    ▒▒15 29.02.88 C&C▒▒▒▒▒▒▒▒ MEMO
Kurbelwelle I      16   360.00       3 12.05.88 Achse&Welle  MEMO
Kurbelwelle II     12   325.00       3 12.05.88 Achse&Welle  MEMO
Formschlauch       41     1.55       4 28.07.88 Gummi&Sohn   MEMO
Kühlluftgehäuse     8    16.60       2 23.11.88 VEB Guß      MEMO
Luftleitblech      14     5.40       2 10.11.88 Blech&Co     MEMO
Kopfdichtungen    116     0.15      30 17.08.88 Dicht&Fest   MEMO
Krümmerdichtung    88     0.14      30 18.09.88 Dicht&Fest   MEMO
Fußdichtung       122     0.15      30 07.06.88 Dicht&Fest   MEMO


  Nur ein Feld editierbar
```

Bild 6.3. Optionen von BROWSE

Die in der ersten Bildschirmzeile angegebenen Wörter sind die möglichen Optionen. In der Anfangsstellung ist das erste Wort invers dargestellt (oder entsprechend farbig). Durch Betätigen der Leertaste bewegen wir uns, ggf. zyklisch, bis zur gewünschten Option. In der unteren Bildschirmzeile wird eine kurze Erläuterung der Option gegeben. Bild 6.3 zeigt die Auswahl der Option 'Stoppen'. Mit ihr kann genau ein Feld festgelegt werden, das korrigiert werden kann. Nur dieses Feld erscheint dann invers. Der Kursor kann nicht aus diesem Feld herausbewegt werden.

Die Option 'Sperren' bewirkt die Bestimmung eines Feldes (Spalte) als linker Rand. Über diesen linken Rand hinaus kann der Kursor nicht weiter nach links bewegt werden. Diese Spalte bleibt ständig auf dem Bildschirm sichtbar. Nur der verbleibende rechte Teil des Bildschirms wird beim Verschieben des Schirms verändert. Die anderen Optionen dienen der Positionierung des Kursors.

BROWSE eignet sich, wie EDIT, zum Besichtigen einer Datenbank.

6.5. Verändern von Feldern in mehreren Sätzen

CHANGE gestattet es, den Inhalt von Feldern eines Datensatzes zu verändern. Die Syntax von CHANGE ist

CHANGE [<gb>] [**FIELDS** <feldliste>] [**FOR/WHILE** <bedingung>]

Wir erläutern die Wirkung an einem Beispiel. Ausgangspunkt ist unser Datenbankfile 'lager.dbf'. Nehmen wir an, es sind Preisveränderungen eingetreten, aber nur für solche Ersatzteile, die bisher weniger als 10.00 Mark kosteten. All diese Teile sind zu besichtigen, und der neue Wert ist von der Tastatur einzugeben.

```
CHANGE FOR preis < 10
```

führt uns in den Direktmodus. Der Bildschirminhalt gleicht dem von EDIT und APPEND. Es werden alle Felder angezeigt, falls keine Feldliste angegeben ist, oder nur die in der Feldliste aufgeführten. Der Kursor ist frei beweglich. Alle Daten können verändert werden.

APPEND (ohne FROM), BROWSE, CHANGE und EDIT sind für die Eingabe und Änderung von Daten von der Tastatur aus gedacht. Sie setzen die Anwesenheit am Bildschirm voraus. Schreibt man ein dBASE-Programm, so benötigt man im Kommandofile Anweisungen, die die Ersetzung von Feldern bewirken. REPLACE ist dafür gedacht.

Eine Ergibtanweisung, wie sie aus anderen Programmiersprachen bekannt ist, gibt es in dBASE nicht. Mit STORE oder "=" kann man nur Speichervariablen Werte zuweisen! Für REPLACE ist folgende Syntax festgelegt:

REPLACE [<gb>] <feld_1> **WITH** <ausdr_1> [,<feld_2> **WITH** <ausdr_2>,...]
[FOR/WHILE <bedingung>]

Stellen wir uns die folgende Aufgabe: Für unser Ersatzteillager ist zu prüfen, wie sich der Wert aller vorhandenen Waren verändert, wenn alle Ersatzteile, die weniger als 10.00 Mark kosten, um ein Prozent im Preis erhöht werden.

```
. USE lager
* Ermitteln des Wertes vor der Preisveränderung
. SUM anzahl * preis
      9 Sätze summiert
   anzahl * preis
         10248.77
* Erhöhen der ausgewählten Preise um 1%
. REPLACE ALL preis WITH preis * 1.01 FOR preis < 10
      6 Sätze ersetzt
* Ermittlung des neuen Wertes
. SUM anzahl * preis
      9 Sätze summiert
   anzahl * preis
         10252.53
```

REPLACE kann auch dann vorteilhaft benutzt werden, wenn in verschiedenen Sätzen das gleiche Feld mit einer längeren Information zu füllen ist. Wir nennen dieses Feld 'feld'. Man wählt dann eine Kurzform dieser Information, die sie von den Informationen dieses Feldes in allen Datensätzen unterscheidet. Diese Kurzform sei 'abk'. Die Ersetzung durch die eigentliche, längere Information kann durch

```
REPLACE ALL feld WITH "eigentliche Information";
```

```
FOR feld = "abk"
```

erfolgen. Das Semikolon am Ende der ersten Kommandozeile zeigt an, daß das Kommando auf der nachfolgenden Zeile fortgesetzt wird.

6.6. Streichen von Sätzen

dBASE unterscheidet zwischen dem logischen Streichen eines Satzes und dessen physischer Vernichtung. Das logische Streichen bewirkt, daß der betreffende Datensatz nur als gestrichen markiert wird, physisch bleibt er erhalten. Es wird eine Löschmarkierung gesetzt. Intern wird dazu das erste Byte eines Datensatzes verwendet. Deshalb ist die Länge eines Datensatzes gleich der Summe der Länge aller Felder plus eins.

Als gelöscht markierte Datensätze können in den Gültigkeitsbereich eines Kommandos einbezogen oder ausgeschlossen werden. Das Kommando

SET DELETED ON/OFF

steuert die Behandlung als gelöscht markierter Sätze. Ist ON gesetzt, so werden als gelöscht markierte Sätze von den meisten Operationen ausgeschlossen. Die Standardeinstellung ist OFF. Hier ist eine gewisse Vorsicht geboten, da man bei Kommandos wie SUM, COUNT usw. nicht ohne weiteres merkt, ob als gelöscht markierte Sätze einbezogen werden und so eine verdeckte Fehlerquelle vorliegt. Insbesondere können als gelöscht markierte Sätze auch korrigiert werden. Das ein Satz als gelöscht markiert ist, wird bei den Kommandos zur Besichtigung einer Datenbank (EDIT, BROWSE, LIST, DISPLAY usw.) angezeigt.

Ein Satz wird über das Kommando DELETE gestrichen oder im Direktmodus durch Eingabe von ^U. Die Löschmarkierung kann im Direktmodus durch nochmalige Eingabe von ^U wieder entfernt werden. Im sequentiellen Kommandomodus wird das Kommando RECALL dafür verwendet.

Als gelöscht markierte Sätze werden erst durch das Kommando PACK physisch vernichtet. PACK bewirkt ein Umspeichern des Datenbankfiles auf dem Datenträger.

7. Strukturänderungen

Die Struktur eines Datenbankfiles ist durch die Anzahl der Felder, ihren Namen, ihren Typ und ihre Breite gegeben. Jede Veränderung einer oder mehrerer dieser Angaben ist eine Strukturänderung. In diesem Abschnitt wird also die Veränderung der äußeren Form der Tabelle betrachtet, während im Abschnitt 6 die Veränderung des Inhalts einer Tabelle behandelt wurde.

Strukturänderungen sollten die Ausnahme sein. Welche Bedeutung die Wahl der Struktur eines Datenbankfiles für das gesamte Projekt hat, wurde im Abschnitt 5 behandelt.

Selbstverständlich beeinflußt die Struktur einer Tabelle auch die in ihr eingetragenen Daten. Was geschieht mit den Daten, wenn die Struktur geändert wird? Wird eine Formatanpassung durchgeführt? Was geschieht mit den Daten gestrichener Felder? Werden neu hinzugefügte Felder initialisiert? Wie wirken sich Änderungen im Feldnamen aus? Diese Fragen werden wir in diesem Abschnitt beantworten.

Drei wesentliche Unterschiede zu dBASE II sind hervorzuheben.

1. Beim Modifizieren der Struktur werden die Daten des modifizierten Datenbankfiles übernommen. Bei dBASE II werden sie vernichtet. Dadurch erübrigt sich in dBASE III ein Kopieren der Struktur des zu ändernden Datenbankfiles und die sich daran anschließenden Operationen: Wechseln des aktiven Datenbankfiles, Modifizieren der Struktur, Anfügen der Sätze des alten Datenbankfiles, Löschen des alten Datenbankfiles, Umbenennen des neuen Datenbankfiles.

2. Ändert man den Feldnamen, so können die Daten dieses Feldes übernommen werden. Das ist in dBASE II nicht möglich.

3. Beim Modifizieren der Struktur wird eine Sicherheitskopie des Datenbankfiles (backup file) angelegt. Sie hat die Namenserweiterung '.bak'.

Die Strukturänderung des aktiven Datenbankfiles kann durch das Kommando

```
MODIFY STRUCTURE
```

eingeleitet werden. Es erfolgt ein Übergang vom sequentiellen Modus der Bildschirmbedienung zum Direktmodus.

Nehmen wir an, eine alte Version unseres Datenbankfiles für die Lagerhaltung ist im File 'lageralt.dbf' gespeichert. Wir besichtigen die Struktur durch die folgende Kommandofolge:

```
. USE lageralt
. LIST STRUCTURE
Datenbankstruktur       - C:lageralt.dbf
Anzahl der Datensätze   -        3
Letztes Änderungsdatum - 19.12.87
Feld   Feldname    Typ         Länge    Dez
    1  ARTIKEL     Zeichen       15
    2  ANZAHL      Numerisch      4
    3  PREIS       Numerisch      8      2
    4  LIEFERUNG   Datum           8
    5  HERSTELLER  Zeichen       12
** Gesamt **                     48
```

Um daraus die Struktur unseres Datenbankfiles 'lager.dbf' zu erzeugen, müssen wir zwischen dem dritten und vierten Feld das Feld 'Minimum' einfügen und ein neues Feld 'Bem' am Ende anfügen. Bild 7.1 zeigt die Situation vor dem Einfügen des Feldes 'Minimum'. Der Platz für das Feld ist bereits geschaffen. Die Eintragungen für Namen, Typ und Breite müssen noch vorgenommen werden. Um das freie Feld zu erzeugen, wurde der Kursor (Leuchtbalken) auf das vierte Feld 'Lieferung' positioniert und ^N eingegeben.

```
C:lageralt.dbf                                  Restliche BYTES :   3953
                                                Felder definiert:      5

 CURSOR : <-- -->    Einfügen      Löschen      Feld auf    :  ↑
  Zeich.:   ← →     Zeich.: Ins   Zeich.: Del  Feld ab     :  ↓
  Wort  :Home End   Feld  : ^N    Wort  : ^Y   Ende/Sichern: ^End
  Spalte:  ^← ^→    HILFE : F1    Feld  : ^U   Abbruch     :  Esc

    Feld Name    Typ        Länge  Dez          Feld Name   Typ       Länge  Dez
 1  ARTIKEL      Zeich_Fld    15
 2  ANZAHL       Numerisch     4     0
 3  PREIS        Numerisch     8     2
 4  ▒▒▒▒▒▒▒▒▒▒▒  Zeich_Fld  ▒▒▒   ▒▒▒
 5  LIEFERUNG    Datum         8
 6  HERSTELLER   Zeich_Fld    12

 Namen beginnen mit Buchstaben; sonst Buchstabe, Zahl oder _ Zeichen
```

Bild 7.1. MODIFY STRUCTURE

Aus dem Hilfsmenü des Bildes 7.1 sind die möglichen Funktionen ersichtlich. Der Kursor kann auf alle Angaben eines Feldes positioniert werden. Die Änderung erfolgt dann durch Eingabe der neuen Werte auf den entsprechenden Positionen. ^End beendet die Strukturänderung. Die Daten des Datenbankfiles werden übernommen. Dabei findet eine physische Umspeicherung der Datensätze statt.

Wurde ein Feld gestrichen, so werden die Daten dieses Feldes ebenfalls gestrichen. Wurde die Spaltenbreite verändert, so findet eine Anpassung an die neue Breite statt. Ein möglicher Formatüberlauf führt bei Zeichenketten zum Abschneiden der rechten Zeichen und ist u.U. nicht sichtbar. Ein Formatüberlauf bei numerischen Feldern liefert Sterne in den Feldern, jedoch keine Fehlermitteilung. Die Breite von logischen, Datum- und Memo-Feldern kann nicht verändert werden.

Wurde der Typ eines Feldes verändert, so werden die Daten übernommen. Es findet eine Typkonvertierung statt. Bezüglich der Typkonvertierung und der Formatanpassung gilt das im Abschnitt 6.1.2 (APPEND) Gesagte. Wird der Typ eines Memo-feldes geändert, z.B. in den Typ String, so werden die Daten nicht übernommen.

Wurde der Name eines Feldes verändert, so wird nach der Eingabe von ^End gefragt, ob die Daten der Felder übernommen werden sollen, deren Name geändert wurde.

> **Achtung!** Werden mit der Namensänderung weitere Strukturänderungen durchgeführt, so erscheint diese Anfrage nicht. Namensänderungen sollten deshalb getrennt von anderen Strukturänderungen durchgeführt werden, wenn die Daten der veränderten Felder übernommen werden sollen. Die doppelte Umspeicherung des Datenbankfile muß man dabei in Kauf nehmen.

Werden neue Felder hinzugefügt, so ist ihr Inhalt undefiniert. Tests mit verschiedenen dBASE-Versionen ergaben, daß man nicht davon ausgehen kann, daß die Inhalte der neu angelegten Felder initialisiert werden. Über EDIT, BROWSE, REPLACE oder CHANGE können ihnen Werte zugewiesen werden.

Wird MODIFY STRUCTURE aufgerufen, während ein Datenbankfile mit Indexfiles in Benutzung ist, so ist das Datenbankfile nach Beendigung der Strukturkorrektur ohne Indexfiles in Benutzung. Es wird keine selbständige Korrektur der Indexfiles durchgeführt, d.h., alle Indexfiles müssen neu erstellt werden. Hier liegt eine verdeckte Fehlerquelle, da die veralteten Indexfiles weiterhin bestehen, ihre Paßfähigkeit zum Datenbankfile aber nicht von dBASE kontrolliert wird.

8. Sortieren und Indizieren

Beide Operationen dienen dazu, eine Ordnung zwischen den Sätzen eines Datenbankfiles herzustellen. Sortiert wird nach den Inhalten eines oder mehrerer Felder in aufsteigender (/A - ascending) oder absteigender (/D - descending) Ordnung. Dabei werden die Datensätze physisch umgespeichert. Sie erhalten neue Satznummern. Es entsteht ein neues, sortiertes Datenbankfile und ein neues .dbt-File, während die alten Files erhalten bleiben.

Beim Indizieren werden die Sätze des Datenbankfiles nicht umgespeichert. Es wird eine logische Ordnung nach einem Ausdruck, dem sog. Indexschlüssel oder Schlüsselausdruck, erzeugt. Hauptbestandteile dieses Ausdrucks sind Felder des Datenbankfiles. Es entsteht ein zusätzliches File, welches neben Verwaltungsinformationen im wesentlichen zwei Angaben enthält: die Werte der berechneten Schlüsselausdrücke, die entsprechend ihrer Ordnung aufgefunden werden können, und einen Verweis zum entsprechenden physischen Satz im Datenbankfile. Greift man auf ein Datenbankfile über ein Indexfile zu, so erscheinen die Sätze entsprechend der Reihenfolge des Indexfiles logisch geordnet.

Wann wird sortiert und wann wird indiziert?

Ein Datenbankfile sollte sortiert nach den wichtigsten Feldern vorliegen. Beim Erfassen der Daten und bei großen Veränderungen oder Ergänzungen braucht man nicht auf diese Reihenfolge zu achten. Sind alle Daten erfaßt, so wird das Datenbankfile sortiert. Der Zeitaufwand für das Sortieren ist bei größeren Datenbankfiles nicht zu unterschätzen. Der Nutzer muß selbst entscheiden, ob ein sortiertes File erforderlich ist oder ob er durch einen geschickten Umgang mit der Datenbank häufiges Sortieren vermeidet. In jedem Fall sollte man die stupide Arbeit, wie z.B. das Sortieren, dem Computer überlassen und sie nicht aus "Effektivitätsgründen" dem Endnutzer anlasten.

Werden wenige Datensätze ergänzt, so kann man über INSERT die Ordnung aufrechterhalten. Da dBASE III gegenüber dBASE II wesentlich bessere Sortieralgorithmen benutzt, ist das Sortieren bei weitem nicht mehr so zeitaufwendig.

Einige praktische Hinweise zum Sortieren:

1. Ein Datenbankfile darf nie auf sich selbst sortiert werden, d.h., das aktive Datenbankfile und das Zielfile müssen verschiedene Namen haben.

2. Streichen Sie das unsortierte Datenbankfile. Denken Sie dabei an unseren Grundsatz, Daten nie doppelt aufzubewahren! Verschiedene Datenbankfiles des gleichen Datenbestandes, nur nach verschiedenen Kriterien sortiert, nehmen nicht nur viel Speicherplatz weg, sie sind eine Gefahr für die Korrektheit unserer Daten, denn Änderungen in allen Datenbankfiles steht man auf die Dauer nicht durch. Für solche Zwecke sind Indexfiles zu benutzen.

3. Wurden auf der Grundlage des unsortierten Datenbankfiles Indexfiles erzeugt, so müssen diese Indexfiles ebenfalls neu erstellt werden. Dadurch verbietet sich ein häufiges Sortieren.

4. Wird die Datenbank von mehreren Nutzern oder Programmen benutzt, so dürfen durch das Sortieren die Anschlußbedingungen nicht verletzt werden. Das betrifft insbesondere die Filenamen und die Paßfähigkeit von Datenbankfile und Indexfiles.

Soll ein Datenbankfile nach verschiedenen Kriterien geordnet betrachtet werden, so verwendet man dafür Indexfiles. Es werden so viele Indexfiles erzeugt, wie Ordnungen gewünscht sind. Dieser Vorgang heißt indizieren. Die markantesten Vorteile gegenüber dem Sortieren sind:

1. Es wird immer mit dem gleichen physischen Datenbankfile gearbeitet. Dadurch müssen Korrekturen an den Daten nur einmal durchgeführt werden und sind dann unter allen Ordnungen erreichbar. Korrekturen an den Daten bewirken gleichzeitig die Überarbeitung der aktiven Indexfiles, so daß die Ordnungen erhalten bleiben. Es liegt keine Datendopplung vor.

2. Indexfiles sind i. allg. kleiner als das Datenbankfile. Ihre Größe hängt von der Länge des Indexschlüssels und von der Reihenfolge ab, in der die Schlüssel beim Indizieren erkannt werden. Beispiele zeigen, daß Indexfiles auch größer werden können als das Datenbankfile. Der Platz, den sie auf dem Datenträger einnehmen, ist deshalb nicht zu vernachlässigen.

3. Der Zugriff zu bestimmten Datensätzen über den Indexschlüssel ist schneller als sequentielles Suchen in einem Datenbankfile (vgl. LOCATE, FIND, SEEK). Es entspricht einem binären Suchen mit anschließendem Direktzugriff.

4. Werden Korrekturen am Datenbankfile durchgeführt, ohne daß die zugehörigen Indexdateien aktiv sind, so muß der Nutzer selbst für die Paßfähigkeit des Datenbankfiles zu allen seinen Indexfiles sorgen. dBASE besitzt keine Möglichkeit, diese Paßfähigkeit zu erkennen. Nimmt man ein Datenbankfile mit einer falschen oder veralteten Indexdatei in Benutzung, so kann das zum Datenverlust führen!

8.1. Sortieren

Ein Datenbankfile sei aktiv. Es kann durch das Kommando SORT sortiert werden.

SORT [<gb>] **TO** <neues_file> **ON** <feld> [**/A**] [**/D**]
[,<feld2] [**/A**] [**/D**] ...] [**FOR/WHILE**<bedingung>]

Fehlt die Angabe von /D (descending - absteigend), so wird in aufsteigender Reihenfolge sortiert. Die Sortierreihenfolge kann für jedes aufgeführte Feld extra festgelegt werden. So ist es möglich, unser Datenbankfile 'lager.dbf' aufsteigend nach dem Feld Artikel zu ordnen und innerhalb gleicher Artikel nach fallender Anzahl.

```
. SORT TO lagers ON artikel, anzahl/D
  100% sortiert          9 Sätze sortiert ... Kopiert Textdatei.
. USE lagers
. LIST
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER    BEM
      1  Flachschieber        112     2.40      15 29.02.88  C&C           MEMO
      2  Formschlauch          41     1.55       4 28.07.88  Gummi&Sohn    MEMO
      3  Fußdichtung          122     0.15      30 07.06.88  Dicht&Fest    MEMO
      4  Kopfdichtungen       116     0.15      30 17.08.88  Dicht&Fest    MEMO
      5  Krümmerdichtung       88     0.14      30 18.09.88  Dicht&Fest    MEMO
      6  Kühlluftgehäuse        8    16.60       2 23.11.88  VEB Guß       MEMO
      7  Kurbelwelle I         16   360.00       3 12.05.88  Achse&Welle   MEMO
      8  Kurbelwelle II        12   325.00       3 12.05.88  Achse&Welle   MEMO
      9  Luftleitblech         14     5.40       2 10.11.88  Blech&Co      MEMO
```

Die Felder, nach denen sortiert wird, können von unterschiedlichem Typ sein. Nach Memo-Feldern kann nicht sortiert werden. Sind mehrere Felder angegeben, so bestimmt das erste Feld die Ordnung. Bei gleichen Inhalten des ersten Feldes bestimmt das zweite Feld die Ordnung usw. Wird durch die aufgeführten Felder keine Reihenfolge bestimmt, so wird die Reihenfolge des Datenbankfiles, das sortiert wird, benutzt. Im Unterschied zum Indizieren können bei SORT keine Ausdrücke angegeben werden!

Gültigkeitsbereich und Bedingung können benutzt werden, um einen sortierten Auszug aus dem bestehenden Datenbankfile zu erzeugen. Ihre Anwendung ist wegen der Dopplung der Daten fraglich. In dBASE II wurden sortierte Auszüge aus Datenbankfiles benutzt, um die Erstellung von Berichten vorzubereiten. Besser ist jedoch der Weg über indizierte Datenbanken und das REPORT-Kommando.

Wollen wir z.B. einen sortierten Auszug unseres Datenbankfiles 'lager.dbf' erstellen, in dem nur die ersten fünf Sätze enthalten sind, und von diesen auch nur solche, deren Hersteller mit einem Zeichen <= "C" beginnen, so leistet die Kommandofolge das Verlangte:

```
. USE lager
. SORT NEXT 5 ON artikel , anzahl TO lagers2 FOR hersteller <= "C"
  100% sortiert          3 Sätze sortiert ... Kopiert Textdatei.
. USE lagers2
. LIST
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER    BEM
      1  Flachschieber        112     2.40      15 29.02.88  C&C           MEMO
      2  Kurbelwelle I         16   360.00       3 12.05.88  Achse&Welle   MEMO
      3  Kurbelwelle II        12   325.00       3 12.05.88  Achse&Welle   MEMO
```

8.2. Indizieren

Das aktive Datenbankfile wird durch das folgende Kommando indiziert:

INDEX ON <ausdr> **TO** <filename>

Es entsteht ein Indexfile mit dem Namen <filename>.ndx. Es empfiehlt sich, die Namen aller Files, die zu einer Datenbank gehören, so zu benennen, daß sie in den ersten beiden Zeichen übereinstimmen. Das erleichtert bei Wartungsarbeiten, die Zusammengehörigkeit der Files zu erkennen.

Der Ausdruck wird als Schlüsselausdruck bezeichnet. Sollen Zeichenketten, numerische Felder und Datumsfelder in einem Ausdruck verknüpft werden, so sind die Konvertierungsfunktionen (Anhang 1) anzuwenden. Ein Schlüsselausdruck ist nur dann sinnvoll, wenn in ihm mindestens ein Feldname vorkommt. Indiziert werden kann nur in aufsteigender Reihenfolge. Bei SORT ist auch die Angabe von /D möglich.

Zulässige Typen für einen Indexausdruck sind String, numerisch und Datum, wobei Indexschlüssel vom Typ Datum in den getesteten Versionen nicht über FIND und SEEK gefunden wurden.

Als Beispiel betrachten wir unser ursprüngliches Datenbankfile 'lager.dbf'. Es soll nach dem Feld 'Lieferung' indiziert werden. Im Unterschied zu dBASE II wird die Datenbank sofort nach dem Indizieren mit diesem Indexfile in Benutzung genommen. Wir sehen das an der folgenden Kommandofolge:

```
. INDEX ON lieferung TO lalief
     9 Sätze indiziert
. LIST
```

Satz #	ARTIKEL	ANZAHL	PREIS	MINIMUM	LIEFERUNG	HERSTELLER	BEM
1	Flachschieber	112	2.40	15	29.02.88	C&C	MEMO
2	Kurbelwelle I	16	360.00	3	12.05.88	Achse&Welle	MEMO
3	Kurbelwelle II	12	325.00	3	12.05.88	Achse&Welle	MEMO
9	Fußdichtung	122	0.15	30	07.06.88	Dicht&Fest	MEMO
4	Formschlauch	41	1.55	4	28.07.88	Gummi&Sohn	MEMO
7	Kopfdichtungen	116	0.15	30	17.08.88	Dicht&Fest	MEMO
8	Krümmerdichtung	88	0.14	30	18.09.88	Dicht&Fest	MEMO
6	Luftleitblech	14	5.40	2	10.11.88	Blech&Co	MEMO
5	Kühlluftgehäuse	8	16.60	2	23.11.88	VEB Guß	MEMO

Darüber hinaus hat man die Möglichkeit, auf ein Datenbankfile über ein Indexfile zuzugreifen:

```
USE lager INDEX lalief
```

oder bei aktivem Datenbankfile 'lager.dbf' durch

```
SET INDEX TO lalief
```

Bewegt man sich mit dem Dateizeiger durch eine indizierte Datenbank, so bewegt sich der Dateizeiger entsprechend der logischen

Ordnung. In unserem Beispiel würde er sich vom Satz 4 zum Satz 7, dann zum Satz 8 usw. bewegen.

Wurde vor dem Indizieren das Kommando SET UNIQUE ON gegeben, so werden nur Datensätze mit unterschiedlichem Indexschlüssel in die Indexdatei aufgenommen (vgl. SET UNIQUE, Abschn. 17.1).

8.3. Mehrere Indexdateien

USE und SET INDEX lassen es zu, bis zu sieben Indexfiles anzugeben. Diese Indexfiles nennt man die aktiven Indexfiles. Das erste Indexfile ist dabei der Hauptindex oder Hauptschlüssel. Dieses Indexfile bestimmt die logische Ordnung, unter der das Datenbankfile betrachtet wird. Die Angabe der anderen Indexfiles dient lediglich dazu, zu sichern, daß bei Korrekturen der Daten auch diese Indexfiles korrigiert werden. Sie haben keinen Einfluß auf die Ordnung, in der die Sätze angezeigt werden.

Sind größere Korrekturen der Daten einer indizierten Datenbank durchzuführen, so wird die wiederholte, von dBASE selbständig durchgeführte Korrektur der Indexfiles zeitaufwendig. Man sollte dann die Indexfiles vor der Korrektur aus der Benutzung nehmen (USE <file> oder CLOSE INDEX), die Korrektur durchführen, dann mit SET INDEX TO die Indexfiles wieder in Benutzung nehmen und über REINDEX aktualisieren. Man spart dadurch u.a. auch die erneute Eingabe der Schlüsselausdrücke ein.

9. Auswahl von Informationen

Der Abschnitt 8 gibt uns die Möglichkeit, unser Datenbankfile unter verschiedenen Gesichtspunkten geordnet zu betrachten. Viele Kommandos (LIST, DISPLAY, REPORT, UPDATE, JOIN u.a.) nehmen auf diese Ordnung direkt oder indirekt Bezug. Wir wollen in diesem Abschnitt die verschiedenen Möglichkeiten betrachten, Informationen in unserer Datenbank aufzufinden.

9.1. Zusammenstellungen

Ein großer Vorteil der Verwendung von Datenbanken besteht darin, Zusammenstellungen von Informationen nach einer bestimmten Ordnung vorzunehmen, wobei die relevanten Sätze einer bestimmten Bedingung genügen. Wählen wir als Beispiel dafür das Kommando LIST und unser Datenbankfile 'lager.dbf', welches indiziert nach dem Feld 'Lieferung' benutzt wird. Damit ist die Ordnung gegeben: das Lieferdatum. Wir suchen nun alle Lieferungen, geordnet nach aufsteigendem Datum, die von der Firma 'Dicht&Fest' kommen.

```
.USE lager INDEX lalief
. LIST FOR hersteller="Dicht&Fest"
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
      9  Fußdichtung         122     0.15      30 07.06.88  Dicht&Fest   MEMO
      7  Kopfdichtungen      116     0.15      30 17.08.88  Dicht&Fest   MEMO
      8  Krümmerdichtung      88     0.14      30 18.09.88  Dicht&Fest   MEMO
```

Möchte man solche einfachen Zusammenstellungen drucken, so bieten sich vier Möglichkeiten an. Die primitivste Lösung besteht darin, über ^P eine Hardkopie einzuschalten und später wieder über ^P auszuschalten. Unter einer Hardkopie versteht man die Ausgabe aller sequentiell auf dem Bildschirm erscheinenden Texte auf dem Drucker. Kaum besser ist die zweite Variante. Über

```
SET PRINT ON
```

wird das gleiche erreicht wie durch ^P. Durch

```
SET PRINT OFF
```

wird die Ausgabe auf den Drucker abgeschaltet. Beide Varianten sind nur für einen schnellen Überblick und für den privaten Gebrauch akzeptabel. Sollen die Zusammenstellungen an andere Personen weitergegeben werden, so wird eine Verbindung oder Ergänzung zu anderen Texten notwendig sein. Mit

```
SET ALTERNATE TO <file>
SET ALTERNATE ON
```

wird ein File mit dem Namen 'file.txt' erzeugt und zum Schreiben eröffnet. Alle sequentiellen Ausgaben, die auf dem Bildschirm erscheinen, werden auch in dieses File geschrieben. Das Textfile kann später mit einem Textverarbeitungssystem bearbeitet werden. Übrigens: Alle Beispiele sequentieller Ausgaben von dBASE in diesem Buch wurden auf diese Weise aufgenommen.

Die vierte Möglichkeit besteht in der Anwendung des COPY-Kommandos mit dem Schlüsselwort SDF (standard data format, hier: Textformat).

```
COPY TO <file> SDF
```

erzeugt ein gewöhnliches Textfile.
Wie man anspruchsvollere Berichte erstellt, werden wir im Abschnitt 11 betrachten.

9.2. Suchen eines Datensatzes

Wie der Dateizeiger im Datenbankfile bewegt wird, haben wir im Abschnitt 3.9 besprochen. Hier wollen wir die sog. assoziative Adressierung betrachten. Dabei wird der Dateizeiger entsprechend dem Inhalt eines Feldes auf den Datensatz positioniert. Die Satznummer ist für die Auffindung des Datensatzes ohne Bedeutung.

LOCATE [<gb>] **FOR** <bedingung>]

LOCATE beginnt am Anfang des Gültigkeitsbereichs einen Satz zu suchen, auf den die Bedingung zutrifft. Der Dateizeiger wird auf den ersten Satz des Gültigkeitsbereichs positioniert, der die Bedingung erfüllt. Zu beachten ist die Wirkung von LOCATE, wenn kein derartiger Satz gefunden wird. Wurde kein Gültigkeitsbereich angegeben, so liefert RECNO() die Nummer des letzen physischen Datensatzes plus eins und EOF() liefert wahr (.T.). Ist ein Gültigkeitsbereich angegeben und wurde kein Datensatz gefunden, so liefert RECNO() die Nummer des letzten Satzes des Gültigkeitsbereichs, wenn dadurch nicht das Fileende erreicht wurde. Es bleibt dem Nutzer überlassen, zu testen, ob wirklich ein Satz gefunden wurde.

> **Achtung!** LOCATE besitzt bezüglich seiner Reaktion bei erfolgloser Suche eine veränderte Semantik gegenüber dem LOCATE-Kommando in dBASE II.

Die FOR-Bedingung bestimmt den zu suchenden Satz. Nehmen wir an, es wurde durch LOCATE ein Satz gefunden. Den nächsten Satz, der der gleichen Bedingung genügt, erreicht man durch CONTINUE. Ist die Suche durch CONTINUE nicht erfolgreich, so gilt das für LOCATE Gesagte. CONTINUE übernimmt den Gültigkeitsbereich und die Auswahlbedingung vom vorangehenden LOCATE-Kommando. Ein Beispiel soll das verdeutlichen.

```
. USE lagers
* Der Dateizeiger soll auf den ersten Satz innerhalb der nächsten
* 5 Sätze positioniert werden, bei dem der eingetragene Preis
* größer als eine Mark ist.
. LOCATE NEXT 5 FOR preis > 1
Satz =       1
. DISPLAY
 Satz # ARTIKEL         ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
      1 Flachschieber      112     2.40      15 29.02.88 C&C          MEMO
* Wir gehen zum nächsten Satz, der diesen Bedingungen genügt.
. CONTINUE
Satz =       2
. DISPLAY
 Satz # ARTIKEL         ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
      2 Formschlauch       41     1.55       4 28.07.88 Gummi&Sohn   MEMO
. CONTINUE
Ende des LOCATE-Bereiches
* ..., d.h., es gibt keinen weiteren derartigen Satz
. ? RECNO(), EOF()
        5 .F.
```

Betrachten wir nun ein Datenbankfile, welches mit einem Indexfile in Benutzung ist, z.B. 'lager.dbf' indiziert nach den Herstellern. FIND und SEEK positionieren den Dateizeiger auf den in logischer Folge ersten Datensatz des Datenbankfiles, dessen Indexschlüssel dem Wert des im Kommando angegebenen Ausdrucks gleich ist. Da alle Datensätze, die diesen Schlüssel haben, falls es mehrere gibt, logisch hintereinander angeordnet sind, gelangt man durch SKIP zum nächsten Datensatz. FIND und SEEK beginnen stets mit dem ersten Satz in logischer Ordnung, unabhängig vom Stand des Dateizeigers.

Werden FIND oder SEEK erfolglos beendet, so ist EOF() = .T., und RECNO() liefert die Nummer des physisch letzten Satzes plus eins.

FIND und SEEK unterscheiden sich nur in der Angabe ihres Ausdrucks. Diesbezüglich besteht eine Analogie zu den Eingabeanweisungen INPUT und ACCEPT. FIND erlaubt die Angabe eines Stringausdrucks auch ohne Zeichenkettenbegrenzer, dafür aber keine numerischen Werte. SEEK akzeptiert numerische Ausdrücke und Stringausdrücke. Letztere müssen in Zeichenkettenbegrenzer eingeschlossen werden. Das hat Konsequenzen, wenn nach Werten gesucht werden soll, die Inhalte von Speichervariablen sind.

SEEK <mem_var>

sucht nach einem Schlüssel, dessen Wert gleich dem Inhalt der Speichervariablen ist.

FIND <mem_var>

sucht nach einem Schlüssel, der gleich der Bezeichnung der Speichervariablen ist, was natürlich nicht gemeint ist. Die Suche nach einem Schlüssel, der gleich dem Inhalt der Speichervariablen ist, wird durch

FIND &<mem_var>

erreicht. Der Inhalt der Speichervariablen wird durch Makrosubstitution erreicht.

Die Syntax beider Kommandos ist durch

FIND <string_ausdr>

und

SEEK <ausdr>

gegeben. Wir bemerken, daß kein Gültigkeitsbereich und keine FOR-/WHILE-Bedingung angegeben werden können. Hier besteht ein wesentlicher Unterschied zum LOCATE-Kommando! Die Bedingung im LOCATE-Kommando kann aus einem beliebigen Ausdruck bestehen, sich also auf beliebige Felder beziehen. FIND und SEEK suchen nur nach übereinstimmenden Schlüsseln. Mehr Angaben sind im Indexfile nicht enthalten. Es ist nicht möglich, Teilzeichenketten aus dem Schlüssel herauszuziehen und danach mit FIND oder SEEK zu suchen. Auch Funktionen, z.B. UPPER(), sind nicht anwendbar. Wissen wir z.B. nur noch, daß in einer Firmenbezeichnung "Fest" vorkam, so würden wir keinen derartigen Satz mit FIND oder SEEK finden, denn die vollständige Bezeichnung ist "Dicht&Fest". Nach "Dicht" können wir dagegen suchen, denn in dBASE gelten zwei Zeichenketten als gleich, wenn die kürzere mit dem Anfang der längeren Zeichenkette gleich ist. Das trifft hier zu. Es sei denn, wir haben SET EXACT ON eingestellt.

LOCATE kann auch in einer Datenbank angewendet werden, die mit Indexfiles in Benutzung ist. LOCATE sucht auch dann sequentiell und folgt in seiner Suche der logischen Ordnung der Sätze. Das verlangsamt das Auffinden von Sätzen im Vergleich zu FIND und SEEK. FIND und SEEK suchen über das Indexfile nach einem Schlüssel. Bei LOCATE können die Bedingungen aus allen Feldern und nicht nur aus dem Schlüssel gebildet werden! Das Suchverfahren, welches für FIND und SEEK angewendet wird, ist ein modifiziertes binäres Suchen in einem balancierten Baum und dadurch sehr schnell. In der Literatur findet man maximale Suchzeiten von zwei, höchstens jedoch drei Sekunden für jeden beliebigen Datensatz. Tests mit einer Datenbank mit 28000 Sätzen konnten diese Aussagen nicht widerlegen.

Betrachten wir die soeben getroffenen Feststellungen an einem Beispiel.

```
. USE lager
* Wir indizieren unser Datenbankfile 'lager' nach dem Hersteller.
. INDEX ON hersteller TO laherst
     9 Sätze indiziert
* Nach dem Indizieren ist das Datenbankfile mit der soeben
* erstellten Indexdatei aktiv.
```

```
. LIST
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
      2  Kurbelwelle I        16   360.00       3 12.05.88  Achse&Welle  MEMO
      3  Kurbelwelle II       12   325.00       3 12.05.88  Achse&Welle  MEMO
      6  Luftleitblech        14     5.40       2 10.11.88  Blech&Co     MEMO
      1  Flachschieber       112     2.40      15 29.02.88  C&C          MEMO
      7  Kopfdichtungen      116     0.15      30 17.08.88  Dicht&Fest   MEMO
      8  Krümmerdichtung      88     0.14      30 18.09.88  Dicht&Fest   MEMO
      9  Fußdichtung         122     0.15      30 07.06.88  Dicht&Fest   MEMO
      4  Formschlauch         41     1.55       4 28.07.88  Gummi&Sohn   MEMO
      5  Kühlluftgehäuse       8    16.60       2 23.11.88  VEB Guß      MEMO
* Wir positionieren den Dateizeiger auf den ersten Artikel,
* der von der Firma "Dicht&Fest" hergestellt wird.
. FIND Dicht
. DISPLAY
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
      7  Kopfdichtungen      116     0.15      30 17.08.88  Dicht&Fest   MEMO
* Wir suchen nach einem nicht vorhandenen Schlüssel
. FIND unbekannt
Nicht gefunden
* ... und prüfen die aktuelle Satznummer.
. ? RECNO(), EOF()
        10 .T.
* Der Anfang des Herstellernamens wird einer Speichervariablen zugewiesen.
. m_herst = "Blech"
Blech
* Ein Satz mit dem Namen "m_herst" existiert nicht.
. FIND m_herst
Nicht gefunden
. FIND &m_herst
. DISPLAY
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
      6  Luftleitblech        14     5.40       2 10.11.88  Blech&Co     MEMO
* Bei SEEK ist die Makroersetzung nicht erforderlich.
. SEEK m_herst
. DISPLAY
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
      6  Luftleitblech        14     5.40       2 10.11.88  Blech&Co     MEMO
```

Achtung! Tests ergaben, daß nach Feldern vom Typ Datum zwar indiziert, jedoch nicht über FIND oder SEEK positioniert werden kann.

```
. USE lager INDEX lalief
. FIND 28.07.88
Nicht gefunden
. FIND "28.07.88"
Nicht gefunden
. FIND CTOD("28.07.88")
Nicht gefunden
```

Die Wirkung von LOCATE in einer indizierten Datenbank zeigen die

folgenden Kommandos:

```
* Wir suchen einen Artikel, der mit "Kur" beginnt.
* Man beachte, daß 'Artikel' nicht der Schlüssel ist!
. LOCATE FOR artikel="Kur"
Satz =       2
. DISPLAY
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
      2  Kurbelwelle          16   360.00       3 12.05.88  Achse&Welle  MEMO
* LOCATE beginnt immer am Anfang des Gültigkeitsbereichs und folgt der logischen
* Ordnung.
. LOCATE FOR anzahl > 112
Satz =       9
. DISPLAY
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
      9  Fußdichtung         122     0.15      30 07.06.88  Dicht&Fest   MEMO
* Der physisch erste Satz, der dieser Bedingung genügt, ist Satz 7.
```

Ein implizites FIND oder SEEK wird bei verbundenen Datenbankfiles durchgeführt. Diese Eigenschaft von dBASE wird im Zusammenhang mit der Arbeit mit mehreren Datenbankfiles im Abschnitt 14 behandelt.

10. Berichte

heBerichte

Im Abschnitt 9.1 haben wir bereits Zusammenstellungen nach bestimmten Gesichtspunkten behandelt. Es wurden Möglichkeiten aufgezeigt, Informationen aus der Datenbank in Textfiles auszugeben. SET ALTERNATE ... und COPY ... SDF waren zwei einfache Varianten. Im Abschnitt 13 wird durch Benutzung von Ausgabeanweisungen eine weitere Möglichkeit aufgezeigt. Hier beschäftigen wir uns mit Berichten, die Tabellenform haben und die wir der Einfachheit halber nur Berichte (REPORT) nennen.

Grundlage der Betrachtungen ist die Feststellung, daß die äußere Form von Berichten weitgehend von der internen Struktur der Datenbankfiles unabhängig ist. Bedingung ist lediglich, daß sich alle auszugebenden Informationen aus den in der Datenbank gespeicherten Fakten zusammensetzen lassen. Die Datenbank kann sich über mehrere Datenbankfiles einschließlich Indexfiles usw. erstrecken.

Es ist selbstverständlich, daß man derartige Berichte durch normale Texte ergänzen oder erläutern möchte. Deshalb besteht die Möglichkeit, Berichte in Textfiles auszugeben, die dann mit einem beliebigen Texteditor ergänzt werden können.

dBASE unterscheidet drei Schritte der Arbeit mit Berichten:

1. die Definition ihrer Struktur (Form),
2. die eigentliche Erzeugung des Berichts in einer vorgegebenen Form und
3. die Korrektur der Struktur eines Berichts.

10.1. Wie soll der Bericht aussehen?

Hierbei legt man die äußere Berichtsform fest - Seitenüberschriften, Seitennumerierung, Länge einer Druckseite, Breite einer Druckzeile, Überschriften der einzelnen Spalten, die Formate ihrer Inhalte, und man bestimmt, für welche Tabellenspalten mit numerischen Werten Zwischensummen und Endsummen gebildet werden sollen. Sehen wir uns das an einem Beispiel an.

Wir benutzen unser Datenbankfile 'lager.dbf' und das Datenbankfile mit den zugehörigen Adressen 'lageradr.dbf'. Gesucht ist eine allerdings sehr vereinfachte Bestandsliste, die Aufschluß über die vorhandenen Ersatzteile, ihre Anzahl, ihren Einzelpreis, den Gesamtwert und den Hersteller einschließlich dessen Standorts gibt. Die Vorbereitungen zur Definition der Berichtsform treffen wir durch folgende Kommandos:

```
. USE lager
. SELECT 2
* Das Datenbankfile mit den Adressen wird, geordnet nach Herstellern,
```

```
* im Arbeitsbereich 2 in Benutzung genommen.
. USE lageradr INDEX laadrh
* Der Arbeitsbereich, in dem lager in Benutzung ist, wird aktiv
. SELECT lager
. CREATE REPORT
REPORT-Dateiname eingeben: labest
```

10.1.1. Festlegen der Form

Die Syntax der Kommandos zum Erzeugen einer Berichtsstruktur ist

CREATE REPORT <filename>

Wird kein Filename angegeben, so fordert dBASE ihn an. Die Bildschirmbedienung geht in den Direktmodus über. Es erscheint Bild 10.1.

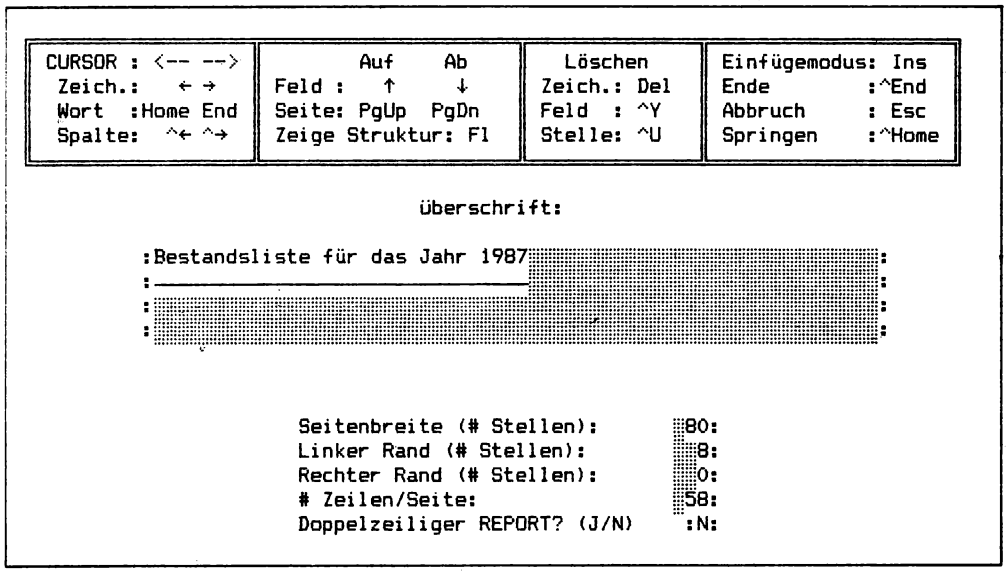


Bild 10.1. CREATE REPORT, Definition der Struktur einer Seite

Die Bereiche für die Eingabe sind wieder durch Doppelpunkte gekennzeichnet bzw. durch Schraffur hervorgehoben. Innerhalb dieser Bereiche ist der Kursor frei beweglich. Beenden wir die Eingabe für das letzte Feld, so erscheint das nächste Menü (Bild 10.2). Es gibt uns die Möglichkeit, Festlegungen über Zwischensummen zu treffen. Wir betrachten diesen Fall später. Für den ersten Versuch wurde ohne Zwischensummen gearbeitet.

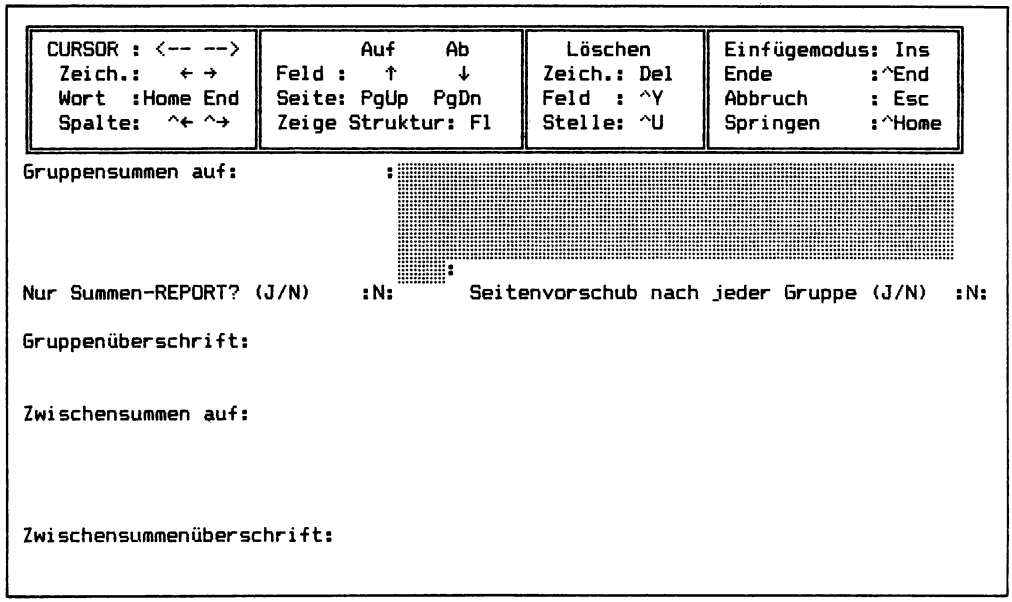


Bild 10.2. CREATE REPORT, Definition der Zwischensummen

Verlassen wir das letzte Feld, so wird danach für jede Spalte des Berichts ein Menü angeboten. Ein derartiges Menü teilt sich in drei Teile. Im oberen Drittel des Bildschirms finden wir das übliche Hilfsmenü. Danach folgen in 6 Zeilen Statusangaben, die den gegenwärtigen Stand der Definition wiedergeben. Die erste Zeile gibt Auskunft über die Nummer der Spalte, in der wir uns befinden, und über die noch verfügbaren Stellen in einer Zeile. In der Statuszeile darunter wird durch >>>>>> der linke Rand angedeutet und durch ---------- der verbleibene Rest in einer Zeile. Die bisher definierten Spaltenüberschriften werden dazwischen wiedergegeben. Maximal vier Zeilen sind für eine Spaltenüberschrift möglich. Große und kleine Buchstaben sind signifikant. Die Wiedergabe in der Statuszeile entspricht fast der Form, wie sie im Bericht erscheint. Einzige Ausnahme ist die Positionierung der Überschriften. Sie werden in der Statuszeile grundsätzlich linksbündig angezeigt, während sie im Bericht für Textspalten linksbündig und für numerische Spalten rechtsbündig erscheinen (vgl. Bild 10.4). Darunter erscheinen die Feldinhalte in Form von Formatzeichen.

In der unteren Bildhälfte kann der Inhalt der Spalte in Form von Ausdrücken definiert werden. Die vier Zeilen für den Feldkopf bestimmen die Überschrift der Spalte. Die Anzahl der Dezimalstellen wird aus der Breite der im Ausdruck verwendeten Felder berechnet. Sie kann für numerische Werte nicht verkleinert werden. Ist eine Zeichenkette länger als die Spaltenbreite, so wird sie auf der nächsten Zeile fortgesetzt. Es ist zu prüfen, ob die dabei von dBASE gewählte Trennung den Vorstellungen des Nutzers entspricht.

Für numerische Werte kann man die Bildung einer Gesamtsumme am Ende des Berichts veranlassen, indem wir in das letzte Feld (Gesamt) ein 'J' eintragen. Bild 10.3 gibt die Situation nach der Definition der vierten Spalte des Berichts wieder.

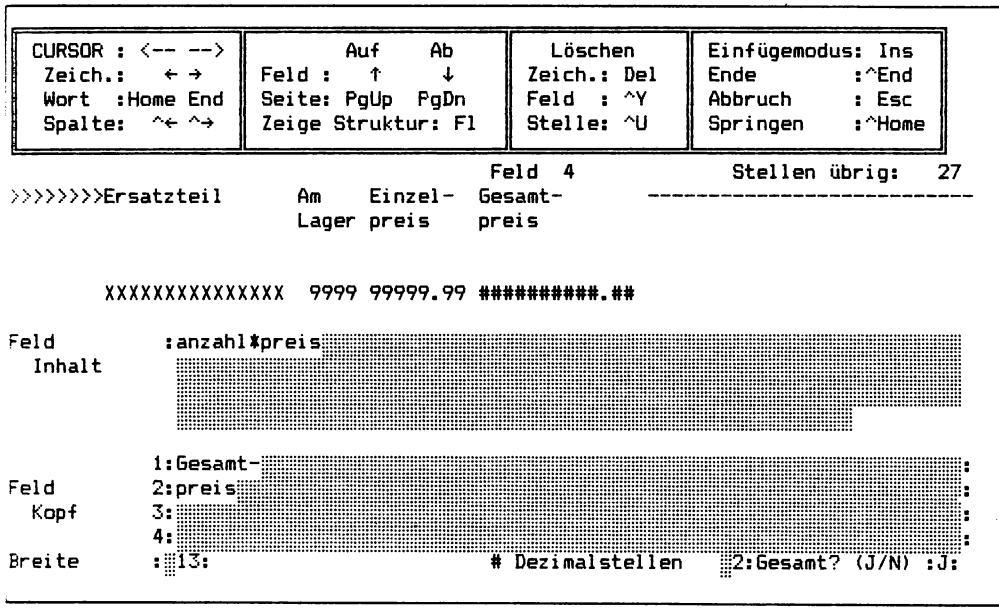


Bild 10.3. CREATE REPORT, Definition der Struktur einer Spalte

Beenden wir die Definition der Struktur durch ^**End**, so steht uns ein File mit dem Namen 'labest.frm' zur Verfügung, welches alle getroffenen Festlegungen enthält. Die Namenserweiterung (extension) '.frm' kennzeichnet Formatfiles für Berichte.

Wir bemerken noch, daß in der letzten Spalte des Berichts der Ort des Herstellers eingetragen werden soll. Da sich diese Angabe in dem Datenbankfile befindet, welches in einem anderen Arbeitsbereich in Benutzung ist, muß dem Feldbezeichner der Aliasname (Dateiname) vorangestellt werden.

```
lageradr->ort
```

Das wird von dBASE nur akzeptiert, wenn das Datenbankfile 'lageradr.dbf' in Benutzung ist. Ansonsten wird ein ungültiger Ausdruck erkannt.

Alle Definitionen über die Struktur des Berichts konnten ohne den konkreten Inhalt der benutzten Datenbankfiles getroffen werden. Die Struktur der Datenbankfiles wurde hingegen verwendet. Ihre Länge ist für die Struktur des Berichts ohne Bedeutung.

Der Bericht selbst wird durch das Kommando REPORT erzeugt.

REPORT FORM <filename> [<gb>] [**FOR** <bedingung>] [**PLAIN**]
[**HEADING** <zeichenkette>] [**NOEJECT**] [**TO PRINT**]
[**TO FILE** <filename>]

Die Wirkung der einzelnen Schlüsselwörter erläutern wir an Beispielen. Dazu setzen wir die Existenz des Formatfiles 'labest.frm' voraus. Geben wir jetzt

```
REPORT FORM labest
```

ein, so wird ein Bericht auf dem Bildschirm ausgegeben, in dem für alle Produkte der gleiche Ort erscheint. Warum? Die Datenbankfiles 'lager' und 'lageradr' sind zwar in Benutzung, sie sind aber nicht verbunden. Das erreicht man durch

```
SET RELATION TO hersteller INTO lageradr
```

Man beachte ferner, daß für das Verbinden zweier Datenbankfiles die Benutzung des zweiten Datenbankfiles mit einer Indexdatei, indiziert nach dem Verbindungsfeld, erforderlich ist (Abschn. 14). Durch

```
USE lageradr INDEX laadrh
```

haben wir diese Voraussetzung erfüllt. Unser Bericht hat dann die folgende Form:

```
. REPORT FORM labest
     Seitennr.     1
     28.12.87
                        Bestandsliste für das Jahr 1987
                        ---------------------------------

Ersatzteil          Am Lager  Einzel-preis  Gesamt-preis  Hersteller    Zu beziehen aus

Flachschieber          112         2.40        268.80     C&C           Rheinsberg
Kurbelwelle I           16       360.00       5760.00     Achse&Welle   Annaburg
Kurbelwelle II          12       325.00       3900.00     Achse&Welle   Annaburg
Formschlauch            41         1.55         63.55     Gummi&Sohn    Cottbus
Kühlluftgehäuse          8        16.60        132.80     VEB Guß       Berlin
Luftleitblech           14         5.40         75.60     Blech&Co.     Dresden
Kopfdichtungen         116         0.15         17.40     Dicht&Fest    Berlin
Krümmerdichtung         88         0.14         12.32     Dicht&Fest    Berlin
Fußdichtung            122         0.15         18.30     Dicht&Fest    Berlin
*** Gesamt ***                                 10248.77
```

Bei längeren Berichten wird, wie aus den Angaben im Bild 10.1 zu ersehen ist, eine Seitenformatierung erzeugt. Die Seiten werden fortlaufend numeriert. Das Datum wird hinzugefügt, und die Seiten- und Spaltenüberschriften werden auf jeder Seite wiederholt. Am Ende einer Seite werden keine Zwischensummen gebildet. Die zweite Seite hat den

im Bild 10.4 angegebenen Kopf:

```
Seitennr.      2
28.12.87
                        Bestandsliste für das Jahr 1987
                        ═══════════════════════════════

Ersatzteil          Am Lager  Einzel-preis  Gesamt-preis  Hersteller  Zu beziehen aus
```

Bild 10.4. Kopf einer Berichtsseite

Betrachten wir nun weitere Möglichkeiten des Kommandos REPORT. Der Gültigkeitsbereich <gb> und die FOR-Bedingung beschränken die Wirkung des Kommandos auf bestimmte Sätze.

TO FILE <filename> läßt den Bericht auf dem Bildschirm erscheinen und legt ihn in gleicher Form in das Textfile <filename> ab. Analog bewirkt **TO PRINT** die Ausgabe auf dem Drucker. TO PRINT und TO FILE können gleichzeitig angegeben werden. Die Ausgabe erfolgt dann auf dem Drucker, ins File und auf dem Bildschirm.

PLAIN unterdrückt die Ausgabe der Seitennummer und des Datums. Die folgende Kommandofolge macht das deutlich.

```
. REPORT FORM labest PLAIN
                        Bestandsliste für das Jahr 1987
                        ═══════════════════════════════

Ersatzteil          Am Lager  Einzel-preis  Gesamt-preis  Hersteller  Zu beziehen aus

Flachschieber            112          2.40        268.80  C&C         Rheinsberg
Kurbelwelle I             16        360.00       5760.00  Achse&Welle Annaburg
Kurbelwelle II            12        325.00       3900.00  Achse&Welle Annaburg
Formschlauch              41          1.55         63.55  Gummi&Sohn  Cottbus
Kühlluftgehäuse            8         16.60        132.80  VEB Guß     Berlin
Luftleitblech             14          5.40         75.60  Blech&Co.   Dresden
Kopfdichtungen           116          0.15         17.40  Dicht&Fest  Berlin
Krümmerdichtung           88          0.14         12.32  Dicht&Fest  Berlin
Fußdichtung              122          0.15         18.30  Dicht&Fest  Berlin
*** Gesamt ***                                  10248.77
```

HEADING <string> fügt eine zusätzliche Überschrift, vergleichbar mit einer Kopfzeile in Texten, ein:

```
. REPORT FORM labest HEADING "(Vereinfachte Form)"
      Seitennr.     1          (Vereinfachte Form)
      28.12.87
                          Bestandsliste für das Jahr 1987
                          ___________________________________

      Ersatzteil          Am  Einzel-      Gesamt- Hersteller   Zu beziehen
                        Lager   preis        preis               aus


      Flachschieber      112     2.40       268.80 C&C          Rheinsberg
      Kurbelwelle I       16   360.00      5760.00 Achse&Welle  Annaburg
      ... usw.
```

NOEJECT verhindert, daß vor dem Beginn des Drucks ein Seitenvorschub ausgeführt wird.

Memo-Felder können ebenfalls in Berichte einbezogen werden. Dazu verändern wir unser Formatfile derart, daß die letzte Spalte des bisherigen Berichts durch den Inhalt des Memo-Feldes "Bem" ersetzt wird. Als Standardbreite der Spalte wird 50 angeboten. Wir verringern die Breite auf 12. Es ergibt sich der folgende Bericht:

```
. REPORT FORM labest
      Seitennr.     1
      28.12.87
                          Bestandsliste für das Jahr 1987
                          ___________________________________

      Ersatzteil          Am  Einzel-      Gesamt- Hersteller   Bemerkungen
                        Lager   preis        preis


      Flachschieber      112     2.40       268.80 C&C          Monatliche
                                                                Lieferung.
      Kurbelwelle I       16   360.00      5760.00 Achse&Welle  Neue
                                                                Ausführung
                                                                ohne
                                                                Simmerring.
      Kurbelwelle II      12   325.00      3900.00 Achse&Welle  Erfordert
                                                                extra
                                                                Simmerring.
      Formschlauch        41     1.55        63.55 Gummi&Sohn
      Kühlluftgehäuse      8    16.60       132.80 VEB Guß
      Luftleitblech       14     5.40        75.60 Blech&Co.
      Kopfdichtungen     116     0.15        17.40 Dicht&Fest    Monatliche
                                                                Lieferung.
      Krümmerdichtung     88     0.14        12.32 Dicht&Fest
      Fußdichtung        122     0.15        18.30 Dicht&Fest
      *** Gesamt ***
                                           10248.77
```

Wir sehen, daß längere Texte auf mehrere Zeilen spaltengerecht

verteilt werden. Ein Zeilenwechsel erfolgt zwischen zwei Wörtern.

10.1.2. Sortierte Berichte und Zwischensummen

Möchte man einen Bericht nach einem bestimmten Kriterium sortiert haben, so ist das Datenbankfile entsprechend geordnet in Benutzung zu nehmen. In unserem Beispiel sind die oben angegebenen Berichte unsortiert. Nehmen wir unser Datenbankfile 'lager.dbf' nach dem Feld 'Artikel' indiziert in Benutzung, so erhalten wir durch REPORT einen Bericht, der nach den Ersatzteilen alphabetisch geordnet ist.

```
. index on artikel to laartikel
     9 Sätze indiziert
. REPORT FORM labest
       Seitennr.      1
       28.12.87
                        Bestandsliste für das Jahr 1987
                        ═══════════════════════════════

       Ersatzteil          Am Lager  Einzel-preis  Gesamt-preis  Hersteller      Zu beziehen aus

       Flachschieber            112          2.40        268.80  C&C             Rheinsberg
       Formschlauch              41          1.55         63.55  Gummi&Sohn      Cottbus
       Fußdichtung              122          0.15         18.30  Dicht&Fest      Berlin
       Kopfdichtungen           116          0.15         17.40  Dicht&Fest      Berlin
       Krümmerdichtung           88          0.14         12.32  Dicht&Fest      Berlin
       Kühlluftgehäuse            8         16.60        132.80  VEB Guß         Berlin
       Kurbelwelle I             16        360.00       5760.00  Achse&Welle     Annaburg
       Kurbelwelle II            12        325.00       3900.00  Achse&Welle     Annaburg
       Luftleitblech             14          5.40         75.60  Blech&Co.       Dresden
      *** Gesamt ***
                                                       10248.77
```

Zwischensummen können für gleiche Werte eines Feldes gebildet werden. Dazu muß das Datenbankfile geordnet nach diesem File in Benutzung genommen werden. In unserem Beispiel ist eine Zwischensummenbildung getrennt nach Herstellern denkbar. Dazu indizieren wir 'lager' nach dem Hersteller und nehmen 'lager' so geordnet in Benutzung.

```
. INDEX ON hersteller TO lagerher
     9 Sätze indiziert
. MODIFY REPORT labest
```

Die Zwischensummenbildung wird durch das zweite Menü von REPORT gesteuert (Bild 10.2).

MODIFY REPORT leitet die Korrektur der Berichtsstruktur ein. Die Bildschirmsteuerung geht in den Direktmodus über. Es werden alle

Menüs, die wir von der Erstellung der Berichtsstruktur kennen, zur Modifikation angeboten. Bild 10.5 zeigt die veränderten Werte.

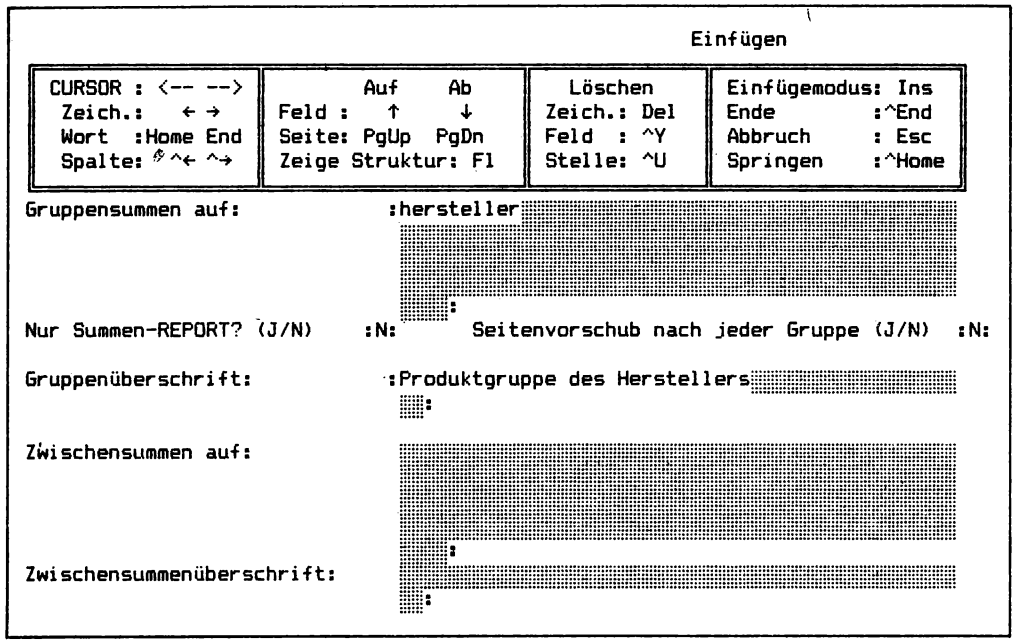


Bild 10.5. CREATE REPORT mit Zwischensummen

"Gruppensummen auf:" verlangt die Angabe des Feldes, nach dem Zwischensummen gebildet werden sollen. Die nach "Gruppenüberschrift:" eingegebene Zeichenkette wird durch den Inhalt des Feldes, nach dem Zwischensummen gebildet werden, ergänzt. Der Begriff "Zwischensumme" wird hierfür verwendet, da er bereits in dBASE II für genau diesen Sachverhalt festgelegt wurde. dBASE III bietet die Möglichkeit, innerhalb der Objekte, über die eine Zwischensumme gebildet wird (in unserem Menü als "Gruppe" bezeichnet), weitere Zwischensummen zu bilden. Die beiden unteren Felder im Bild 10.5 müssen dann entsprechend gefüllt werden. Voraussetzung dafür ist, daß innerhalb der bereits bestehenden Ordnung (hier "Hersteller") eine weitere Ordnung, besteht, nach der die Gruppenbildung vorgenommen werden kann. Wir verfolgen diesen Weg im Beispiel nicht weiter. Den Bericht mit einfachen Zwischensummen gibt Bild 10.6 wieder.

Wir sehen an diesem Bericht, daß der Hersteller in der Gruppenüberschrift und in der fünften Spalte vorkommt. Diese doppelte Angabe kann durch Streichen der fünften Berichtsspalte vermieden werden. Hersteller und Ort sind fest miteinander verbunden. Gelingt es, auch den Ort in die Gruppenüberschrift zu bringen, so kann auch die sechste Spalte des Berichts entfallen. Wir erreichen das, indem wir folgenden Ausdruck nach "Gruppensummen auf:" eingeben:

```
TRIM(hersteller) + ' in   + lageradr->ort

. REPORT FORM labest
      Seitennr.     1
      28.12.87
                        Bestandsliste für das Jahr 1987
                        ------------------------------

      Ersatzteil          Am  Einzel-      Gesamt-  Hersteller   Zu beziehen
                       Lager   preis        preis                 aus


      ** Produktgruppe des Herstellers Achse&Welle
       Kurbelwelle I      16   360.00      5760.00 Achse&Welle   Annaburg
       Kurbelwelle II     12   325.00      3900.00 Achse&Welle   Annaburg
      ** Zwischensumme **
                                          9660.00

      ** Produktgruppe des Herstellers Blech&Co.
       Luftleitblech      14     5.40        75.60 Blech&Co.     Dresden
      ** Zwischensumme **
                                            75.60

      ** Produktgruppe des Herstellers C&C
       Flachschieber     112     2.40       268.80 C&C           Rheinsberg
      ** Zwischensumme **
                                           268.80

      ** Produktgruppe des Herstellers Dicht&Fest
       Kopfdichtungen    116     0.15        17.40 Dicht&Fest    Berlin
       Krümmerdichtung    88     0.14        12.32 Dicht&Fest    Berlin
       Fußdichtung       122     0.15        18.30 Dicht&Fest    Berlin
      ** Zwischensumme **
                                            48.02

      ** Produktgruppe des Herstellers Gummi&Sohn
       Formschlauch       41     1.55        63.55 Gummi&Sohn    Cottbus
      ** Zwischensumme **
                                            63.55

      ** Produktgruppe des Herstellers VEB Guß
       Kühlluftgehäuse     8    16.60       132.80 VEB Guß       Berlin
      ** Zwischensumme **
                                           132.80
      *** Gesamt ***
                                         10248.77
```

Bild 10.6. Bericht mit Zwischensummen

10.1.3. Andere Berichtsformen

Wir kennen jetzt die Möglichkeiten, die uns REPORT für die Erstellung von Berichten bietet. Man muß sich dabei vor Augen halten, daß es sich um eine Standardform für Berichte handelt, die natürlich nicht allen Sonderwünschen gerecht werden kann. Kommen wir zu dem Schluß, daß diese Form für unsere Zwecke nicht ausreicht, so sollte man als nächstes überlegen, ob mit geringfügigen textlichen Veränderungen an diesem Bericht eine akzeptable Form erreicht werden kann. In diesem Fall empfiehlt sich

```
REPORT FORM      TO FILE
```

und die anschließende Überarbeitung mit einem Texteditor.

Reicht das immer noch nicht, so helfen nur eigene Kommandofiles, die Ein- und Ausgabekommandos (Abschn. 13) enthalten. Der Aufwand zum Erstellen und Testen von Kommandofiles ist jedoch erheblich höher als die Benutzung von REPORT, so daß unter diesem Gesichtspunkt über mögliche Kompromisse in der Berichtsform nachzudenken ist.

Reicht die Papierbreite anscheinend nicht aus, um alle Angaben (Spalten) unterzubringen, so prüfe man zuerst die Fähigkeiten seines Druckers. Gängige Drucker für A4-Papier (hochkant) können auch in engeren Zeichenabständen drucken. Durch Einstellung am Drucker (DIP-Schalter oder Bedientasten) oder durch Programmierung lassen sich Zeilen mit 96, 136, 156 Zeichen und noch mehr darstellen. Wir bringen dadurch pro Zeile mehr Informationen unter, als auf einer konventionellen Schreibmaschine mit A3-Wagen darstellbar ist. Ein Drucker für A3-Papier verdoppelt die Anzahl der möglichen Zeichen je Zeile noch einmal. Die Tabellenbreite dürfte damit kein Problem mehr sein. dBASE III läßt Berichte bis zu 999 Zeichen je Zeile mit höchstens 24 Berichtsspalten zu. Insbesondere die 999 Zeichen je Zeile stellen keine wirkliche Einschränkung dar. Außerdem bleibt uns immer noch die Möglichkeit, Berichte vertikal zu teilen, d.h., wir übernehmen einige Berichtsspalten in einen zweiten oder sogar dritten Bericht.

10.2. Modifizieren von Berichten

Zum Modifizieren (Editieren) von Berichten steht das Kommando

```
MODIFY REPORT <filename>
```

zur Verfügung. Es bietet die gleichen Menüs in genau der gleichen Reihenfolge an wie CREATE REPORT (Abschn. 10.1.1). Im Abschnitt 10.1.2 haben wir ein bereits existierendes Formatfile modifiziert. Die möglichen Steuerkommandos sind den Hilfsmenüs der Bilder 10.1 bis 10.5 zu entnehmen. Es können alle Angaben modifiziert werden, die in den Menübildern angeboten werden, und man kann neue Berichtsspalten hinzufügen.

Es ist jedoch nicht möglich, Berichtsspalten zu streichen oder in

der Mitte einzufügen! Auch die letzte Berichtsspalte kann nicht gestrichen werden.

Es sei noch auf einen Unterschied zwischen dBASE II und dBASE III hingewiesen: Formatfiles (.frm) für Berichte sind in dBASE III, im Unterschied zu dBASE II, keine Textfiles. Sie können insbesondere nicht durch einen Texteditor korrigiert werden, wodurch das Kommando MODIFY REPORT erforderlich wird. In dBASE II sind Formatfiles Textfiles, die mit einem Editor korrigiert werden können. Dadurch ist es in dBASE II kein Problem, Berichtsspalten zu streichen oder in der Mitte hinzuzufügen.

11. Kommandofiles

Wir haben in den vorangegangenen Abschnitten gesehen, daß einige dBASE-Kommandos sehr komplexe Operationen beinhalten. Das ist auch ein charakteristisches Merkmal von Datenbanksystemen. Dennoch lassen sich nicht alle gewünschten Resultate durch die Eingabe eines Kommandos erreichen. Es sind Folgen von Kommandos erforderlich. Wird eine Wiederholung der Operation gewünscht, so muß, jedenfalls bis jetzt, die Kommandofolge erneut eingegeben werden. Das ist aus mehreren Gründen sehr unpraktisch. Deshalb möchte man Kommandofolgen speichern und benennen, um über den so gewählten Namen ihre Abarbeitung zu veranlassen. Gespeichert werden sie in Kommandofiles.

Im Abschnitt 4.7 haben wir bei der Besprechung der Funktion TIME() solche Kommandofolgen bereits angegeben. Wir erwähnten die Möglichkeit, sie in Kommandofiles zu speichern, ohne diesen Begriff genauer zu erklären. Das soll in diesem Abschnitt geschehen.

Alle Kommandos, die man von der Tastatur aus im Dialog eingeben kann, lassen sich in einem Textfile speichern. Ein Textfile, dessen Zeilen aus Kommandos bestehen, heißt Kommandofile. Der Name des Kommandofiles bezeichnet dann die Kommandofolge. Er hat die Namenserweiterung '.prg' (in dBASE II auch '.cmd'). Die Ausführung der Kommandofolge wird über den Namen des Kommandofiles, hier <filename>, veranlaßt. Man sagt auch: Das Kommandofile wird aufgerufen. Dazu dient das Kommando

```
DO <filename>
```

Diese Möglichkeiten besitzt auch dBASE II. In dBASE III geht man davon aus, daß für die Abarbeitung einer Kommandofolge diese kein separates File bilden muß. Es reicht aus, wenn sie über einen Namen erreichbar ist. Solche Folgen von dBASE-Kommandos bezeichnet man als Prozeduren. Durch das Kommando PROCEDURE wird ihnen ein Name zugeordnet. Über diesen Namen werden sie aufgerufen:

```
DO <proc_name>
```

Im Normalfall sind mehrere Prozeduren in einem File gespeichert. Ein solches Kommandofile heißt Prozedurfile. Prozeduren wurden im Abschnitt 11.2 behandelt.

Wir vereinbaren für die folgenden Erläuterungen, daß wir unter einem Kommandofile auch ein Prozedurfile verstehen. Ist eine Unterscheidung erforderlich, so wird sie im Text besonders hervorgehoben.

Kommandofiles können mit einem beliebigen Texteditor erstellt werden, speziell mit dem dBASE-internen Editor (Abschn. 12). Wird ein privater Editor verwendet, so ist darauf zu achten, daß er keine Zeichen (sichtbar oder unsichtbar) in den Text ablegt, die bei der Abarbeitung des Kommandofiles zu Fehlern führen. Die unterschiedliche Behandlung des Tabulatorzeichens kann u.U. schon eine Fehlerquelle sein.

11.1. Kommandos in Kommandofiles

Für Kommandofiles reichen die im Dialog benutzten dBASE-Kommandos nicht aus. Wir benötigen zusätzliche Kommandos für Programmverzweigungen, für Schleifen, für den Aufruf weiterer Kommandofiles, und wir brauchen Kommandos für die Ein- und Ausgabe usw. Mit diesen für Kommandofiles typischen Kommandos beschäftigt sich dieser Abschnitt. Für die Notation der Syntax beziehen wir uns auf die im Abschnitt 3 getroffenen Festlegungen.

Sind Kommandos in einem File gespeichert, so besteht kein Unterschied zu Anweisungen einer Programmiersprache. "Kommando" und "Anweisung" werden in diesem Sinne als gleichwertig angesehen.

Mit dem Aufstellen von Kommandofiles programmieren wir in dBASE. dBASE wird damit in den Rang einer Programmiersprache erhoben. Eine Sammlung von dBASE-Kommandofiles, die evtl. nur aus einem Kommandofile besteht, ist ein dBASE-Programm. Auch Kommandofiles werden, wie einzelne Kommandos, interpretativ abgearbeitet. Es entfallen dadurch die bei der Übersetzung traditionellen Schritte: Übersetzen - Verbinden - Laden.

> Eine Vorbemerkung: dBASE kennt keine Sprünge! Das fördert das Denken in Programmstrukturen. GOTO oder GO positioniert den Dateizeiger.

Programmverzweigungen

Hierzu zählen zwei Kommandos. Das IF-Kommando, auch als bedingte Anweisung bezeichnet, dient der Verzweigung der Programmsteuerung in Abhängigkeit von einer Bedingung. Die Syntax des IF-Kommandos ist gegeben durch

```
IF <bedingung>
     <kommandofolge_1>
[ELSE
     <kommandofolge_2>]
ENDIF
```

Die Bedingung wird berechnet. Sie liefert einen logischen Wert (wahr oder falsch, .T. oder .F.). Liefert sie "wahr", so wird die Kommandofolge 1 abgearbeitet, sonst die Kommandofolge 2. In beiden Fällen wird mit dem nach ENDIF folgenden Kommando fortgesetzt. Fehlt ELSE und die Kommandofolge 2, so wird mit dem Kommando nach ENDIF fortgesetzt. Wurde die Kommandofolge 1 abgearbeitet, so wird mit dem Kommando nach ENDIF fortgesetzt. Die Kommandofolge 2 wird in diesem Fall nicht abgearbeitet. Beide Kommandofolgen stellen parallele Wege dar. Die Programmsteuerung kann entweder den einen oder den anderen Weg verfolgen, aber nie beide gleichzeitig. Bild 11.1 gibt den Verlauf der Programmsteuerung für ein IF-Kommando an. Betrachten wir die Wirkung eines IF-Kommandos an einem einfachen Beispiel. Das Kommandofile 'ifbsp.prg' enthält nur ein IF-Kommando. Über eine Speicherva-

riable 'm_schalter' vom Typ 'logisch' wird der Weg ausgewählt.

```
* Kommandofile 'ifbsp.prg' zum Test des IF-Kommandos
IF m_schalter
  ? "THEN-Teil abgearbeitet"
ELSE
  ? "ELSE-Teil abgearbeitet"
ENDIF
RETURN
* Ende des Kommandofiles
```

Betrachten wir die folgenden Aufrufe:

```
. m_schalter = .t.
.T.
. DO ifbsp
THEN-Teil abgearbeitet
. m_schalter = .f.
.F.
. DO ifbsp
ELSE-Teil abgearbeitet
```

Die Schlüsselwörter IF, ELSE und ENDIF sollten die ersten in einer Zeile sein. Einrückungen sind möglich und empfehlenswert, da sie die Lesbarkeit des Programms wesentlich erhöhen können.

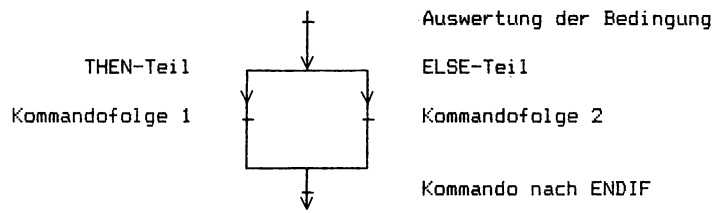


Bild 11.1. IF-Kommando

Eine zweite Form der Programmverzweigung ist das DO CASE-Kommando:

```
DO CASE
   CASE <bedingung> <kommandofolge>
   {[CASE <bedingung> <kommandofolge>]...}
   [OTHERWISE] <kommandofolge>
ENDCASE
```

Die erste Bedingung wird berechnet. Liefert sie "wahr", so wird die danach stehende Kommandofolge abgearbeitet. Liefert die Bedingung "falsch", so wird die danach folgende Bedingung berechnet usw. Wurde eine wahre Bedingung erkannt und deren Kommandofolge abgearbeitet, so wird mit dem Kommando nach ENDCASE fortgesetzt. Trifft keine Bedingung zu, so wird die Kommandofolge nach OTHERWISE abgearbeitet. Fehlt OTHERWISE, so wird sofort mit dem Kommando nach ENDCASE fortgesetzt.

Bild 11.2 gibt den Verlauf der Programmsteuerung in einem DO CASE-Kommando wieder. Ein Beispiel für die Anwendung eines DO CASE-Kommandos befindet sich im Abschnitt 13.1.

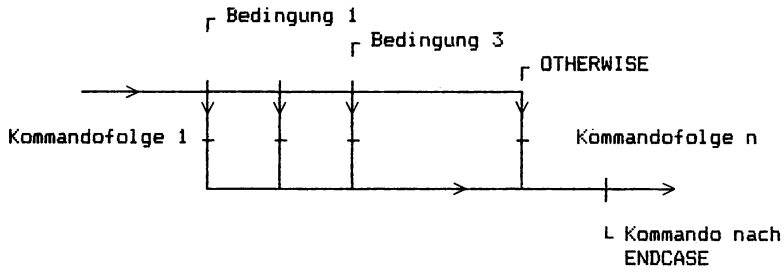


Bild 11.2. DO CASE-Kommando

Zyklusanweisungen

Zyklusanweisungen - hier spricht man besser von Anweisungen als von Kommandos - beschreiben eine Kommandofolge, die in Abhängigkeit von einer Steuergröße wiederholt wird. Die Steuergröße geht in die Bedingung ein. Die Syntax ist

```
DO WHILE <bedingung>
     <kommandofolge>
ENDDO
```

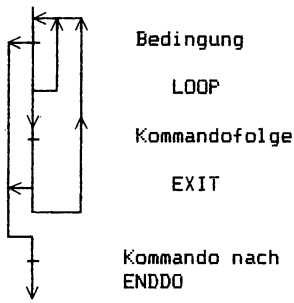


Bild 11.3. DO WHILE-Kommando

Die Bedingung wird ausgewertet. Liefert sie den Wert "wahr", so wird die Kommandofolge ausgeführt. ENDDO kehrt zu der Zeile zurück, in der das zugehörige DO WHILE steht. Die Bedingung wird erneut ausgewertet usw. Liefert die Auswertung der Bedingung "falsch", so wird mit dem Kommando fortgesetzt, das nach ENDDO steht. Die Kommandofolge wird also so lange wiederholt, wie die Bedingung erfüllt ("wahr") ist. Bild 11.3 zeigt den Verlauf der Programmsteuerung in einem DO WHILE-Kommando.

Programmverzweigungen sind innerhalb von DO WHILE-Kommandos durch die Kommandos LOOP und EXIT möglich. Sie haben nur innerhalb von DO WHILE-Kommandos einen Sinn.

Das Kommando **LOOP** bewirkt, daß die Programmsteuerung an die innerste DO WHILE-Zeile übergeben wird, oder, anders ausgedrückt, es erfolgt ein Rücksprung zur innersten DO-WHILE-Zeile. Dadurch werden die zwischen LOOP und ENDDO stehenden Kommandos in diesem Zyklus nicht abgearbeitet. Die Bedingung wird erneut geprüft usw.

EXIT bewirkt die Übergabe der Programmsteuerung an das Kommando, welches auf das innerste ENDDO folgt. Anders ausgedrückt: Es erfolgt ein Sprung zu dem Kommando, das auf das innerste ENDDO folgt. Der Zyklus ist beendet.

LOOP und EXIT können beliebig oft in einem DO WHILE-Kommando stehen. Bild 11.3 zeigt schematisch die Wirkung beider Kommandos.

Zwei weitere Kommandos sind nur in Kommandofiles sinnvoll: RETURN und CANCEL.

RETURN in einer Prozedur bewirkt die Rückkehr der Programmsteuerung zu dem Kommando, welches dem aufrufenden DO-Kommando folgt. Das Prozedurfile wird nicht abgeschlossen. RETURN in einem Kommandofile bewirkt das Verlassen dieses Kommandofiles. Es wird abgeschlossen. Die Programmsteuerung kehrt zu dem Kommando zurück, welches dem aufrufenden DO-Kommando folgt.

CANCEL bewirkt die Übergabe der Programmsteuerung zur Dialogebene von dBASE, d.h., auf dem Bildschirm wird das Prompt (gewöhnlich ein Punkt) ausgegeben. Das nächste Kommando muß von der Tastatur eingegeben werden.

Alle in diesem Abschnitt besprochenen Kommandos können ineinander verschachtelt sein. Für die Einhaltung der Struktur ist der Nutzer selbst verantwortlich. Bedingt durch die interpretative Arbeitsweise, können syntaktische Fehler in Kommandofiles erst zur Abarbeitungszeit gefunden werden.

Beispiel:

Betrachten wir eine mögliche Kommandofolge mit verschachtelten strukturierten Kommandos. 'Artikel' und 'Preis' seien Felder des aktiven Datenbankfiles. 'i' ist eine Speichervariable, die einen Zählwert enthält. Da der Wert von i von einem Anfangswert an eine Folge von Werten durchläuft, spricht man auch von einer Laufvariablen. 'm_max' ist ebenfalls eine Speichervariable, die in unserem Fall einen Endwert für die Laufvariable i enthält.

Um die Beschreibung zu erleichtern, wurden die Zeilen numeriert. In dBASE sind Kommentare nach einem Kommando und in der gleichen Zeile nicht zugelassen. Kommentarzeilen werden durch einen Stern '*' oder durch das Schlüsselwort NOTE eingeleitet.

Betrachten wir den folgenden Ausschnitt aus einem Kommandofile:

```
* Das zugehörige ENDDO steht in Zeile 13.
* Der Zyklus wird so lange durchlaufen, bis das Ende des
* aktiven Datenbankfiles in diesem Arbeitsbereich
* erreicht ist.
```

```
DO WHILE NOT EOF()                                          (1)
   ...
   * Das zugehörige ENDIF steht in Zeile 11.
   IF artikel = "K"                                         (2)
      ...
      * Dieses EXIT führt zum Kommando nach Zeile 13.
      EXIT                                                  (3)
   ELSE                                                     (4)
      * Dieses DO WHILE-Kommando liegt im ELSE-Teil
      * des IF-Kommandos (Zeilen 2 bis 11).
      * Das zugehörige ENDDO steht in Zeile 10.
      DO WHILE i < m_max                                    (5)
         ...
         * Dieses IF-Kommando hat keinen ELSE-Teil.
         * Das zugehörige ENDIF steht in Zeile 8.
         IF preis < 100                                     (6)
            ...
         * Dieses LOOP führt zum innersten DO WHILE,
         * also zur Zeile 5.
            LOOP                                            (7)
         ENDIF                                              (8)
         i = i + 1                                          (9)
      * Die Programmsteuerung gelangt zur Zeile 5.
      ENDDO                                                 (10)
   * Das zugehörige IF steht in Zeile 2.
   ENDIF                                                    (11)
   ...                                                      (12)
   SKIP
* Die Programmsteuerung gelangt zur Zeile 1.
ENDDO                                                       (13)
```

Zu den Kommandos, die nur in Kommandofiles sinnvoll sind, gehören auch die Kommandos für die Ein- und Ausgabe. Sie werden im Abschnitt 13 behandelt.

11.2. Kommandofiles und Prozeduren

Die Begriffe wurden bereits am Anfang des Abschnitts 11 definiert. Hier wollen wir Definition und Aufruf von Prozeduren und Kommandofiles betrachten.

Kommandofiles werden definiert, indem man sie als normale Textfiles erstellt. Die Textzeilen müssen gültige dBASE-Kommandos sein. Ihr Aufruf erfolgt durch

```
DO <filename>
```

Der Aufruf eines Kommandofiles hat zur Folge, daß dieses File eröffnet wird. Rufen Kommandofiles einander auf, so bedeutet das, daß mehrere Files gleichzeitig eröffnet sein müssen. Aus Abschnitt 2.7 wissen wir, daß die Anzahl der gleichzeitig eröffneten Files be-

schränkt ist. Schlußfolgert man daraus, daß möglichst große Kommandofiles vorteilhaft sind, so ist das ein Trugschluß. Demgegenüber steht nämlich das Bestreben, ein Programm in "handliche" und überschaubare Prozeduren zu zerlegen, die in ihrer Textlänge zwei Seiten nicht übersteigen sollten. dBASE III löst den Widerspruch dadurch, daß ein Kommandofile mehrere Prozeduren enthalten kann.

Eine Prozedur, vorerst ohne Parameter, wird deklariert durch

```
PROCEDURE <proc_name>
   <kommandofolge>
```

Die Kommandofolge wird durch ein folgendes PROCEDURE oder das Ende des Prozedurfiles begrenzt. Mehrere RETURN-Kommandos sind möglich. Sie bewirken die Rückgabe der Programmsteuerung. <proc_name> ist ein Name, der aus höchstens acht Zeichen besteht. Er ist der Name der Prozedur und bezeichnet damit die nachfolgende Kommandofolge. Durch

```
DO <proc_name>
```

wird diese Kommandofolge aufgerufen. In einem Prozedurfile können maximal 32 Prozeduren enthalten sein.

Im Gegensatz zum Aufruf eines Kommandofiles ist beim Aufruf einer Prozedur nicht klar, in welchem File sie zu finden ist. Es wird deshalb vereinbart, daß das File, in dem eine aufzurufende Prozedur gesucht wird, durch

```
SET PROCEDURE TO <filename>
```

vorher festgelegt wird. Durch SET PROCEDURE TO wird das Prozedurfile eröffnet. Ein evtl. vorher eröffnetes Prozedurfile wird abgeschlossen. Damit entfällt das Eröffnen beim Aufruf jeder Prozedur, und die Abarbeitung wird beschleunigt. Zu einem Zeitpunkt kann immer nur ein Prozedurfile eröffnet sein. Daraus folgt, daß in einem Prozedurfile kein neues Prozedurfile eröffnet werden kann. Wir können aber in einem Kommandofile ein Prozedurfile eröffnen, Prozeduren aus diesem Prozedurfile aufrufen und ins Kommandofile zurückkehren. Dort kann das Prozedurfile durch

```
CLOSE PROCEDURE
```

abgeschlossen werden. Ein neues Prozedurfile kann eröffnet, Prozeduren können aufgerufen werden usw.

Die Prozeduren eines Prozedurfiles können sich gegenseitig aufrufen. Vorwärtsbezüge sind möglich, d.h., eine Prozedur kann aufgerufen werden, obwohl sie im Text des Prozedurfiles erst später definiert wird. Wir zeigen das an einem Beispiel. Gegeben sei das folgende Kommandofile 'procfile.prg'.

```
* Prozedurfile 'procfile.prg'
* Am Anfang können Kommandos stehen, die zu keiner Prozedur
* gehören. Sie werden abgearbeitet, indem das File über
* seinen Filenamen aufgerufen wird.
? "Das ist der Hauptteil"
```

```
* Die Prozeduren dieses Files sind auch aus seinem Hauptteil
* nur erreichbar, wenn vorher SET PROCEDURE     abgearbeitet wurde.
RETURN
* Dieses Prozedurfile enthält zwei Prozeduren p1 und p2.
PROCEDURE p1
   ? "Ich bin Prozedur p1"
   * Aufruf von p2 aus dem gleichen Prozedurfile
   * p2 wird im Text später definiert
   DO p2
RETURN
* Ende des Prozedurfiles 'procfile.prg'

PROCEDURE p2
   ? "Ich bin Prozedur p2"
RETURN
* Ende des Prozedurfiles
```

Dann ist folgender Dialog möglich:

```
. DO procfile
Das ist der Hauptteil
. SET PROCEDURE TO procfile
. DO p2
Ich bin Prozedur p2
. DO p1
Ich bin Prozedur p1
Ich bin Prozedur p2
* p3 ist nicht im eröffneten Prozedurfile ...
. DO p3
Syntaxfehler
      ^
      ?
DO p3
Wünschen Sie HILFE? (J/N) Nein
. SET PROCEDURE TO pf2
* ... aber in einem Prozedurfile 'pf2.prg'
. DO p3
Ich bin Prozedur p3
```

Betrachten wir noch den Aufruf von Prozeduren, die in verschiedenen Prozedurfiles stehen, aus einem Kommandofile. Das Kommandofile sei 'main.prg'. Es ruft Prozeduren aus dem Prozedurfile 'procfile.prg' auf. Außerdem wird ein Prozedurfile 'proc2.prg' mit der Prozedur 'pr1' verwendet. Da 'pr1' mit CANCEL beendet wird, kehrt die Programmsteuerung nicht zum Kommando "? 'main'" zurück. Die Ausschrift 'DO abgebrochen' wird bei der Abarbeitung von CANCEL erzeugt. Die Programmsteuerung kehrt zur Dialogebene von dBASE zurück.

```
* Kommandofile 'main.prg'
SET PROCEDURE TO procfile
DO p2
DO p1
* pr1 steht im Prozedurfile 'proc2.prg'
SET PROCEDURE TO proc2
```

```
DO pr1
? 'main'
RETURN
* Ende des Kommandofiles 'main.prg'

* Prozedurfile 'proc2.prg'
PROCEDURE pr1
? 'Ich bin Prozedur pr1'
CANCEL
RETURN
* Ende des Prozedurfiles 'proc2.prg'
```

Die Abarbeitung erfolgt durch

```
. DO main
Ich bin Prozedur p2
Ich bin Prozedur p1
Ich bin Prozedur p2
Ich bin Prozedur pr1
DO abgebrochen
```

Zwei Bemerkungen für Kenner anderer Programmiersprachen:

1. Tests ergaben, daß auch rekursive Prozeduraufrufe möglich sind. Sie sind in ihrer Wirkung jedoch nicht dokumentiert und sollten deshalb nicht verwendet werden. Ein Prozeduraufruf heißt rekursiv, wenn eine Prozedur sich selbst, evtl. über mehrere andere Prozeduren, wieder aufruft.
2. Aus den angegebenen Regeln zur Definition einer Prozedur folgt, daß verschachtelte Prozeduren nicht möglich sind, d.h., eine Prozedur kann nicht lokal zu einer anderen Prozedur sein.

11.3. Lokale Variable

dBASE III unterscheidet lokale und globale Speichervariable. Eine Variable heißt global, wenn sie auf der Dialogebene von dBASE, also durch Eingabe von der Tastatur, deklariert ist. Eine Variable heißt lokal, wenn sie innerhalb eines Kommandofiles deklariert ist. Wird eine Variable in einem Kommandofile als PUBLIC deklariert, so wird sie dadurch zu einer globalen Variablen. Wird eine lokale Variable als PRIVATE deklariert, so ist sie in dem Prozedurfile lokal. Eine globale Variable mit gleichem Namen bleibt unverändert erhalten.

Lokale Variable werden eingeführt, um die Wirkung von Ergibtanweisungen (STORE- und =-Kommando) zu beschränken und so unbeabsichtigte Veränderungen zu reduzieren. Insbesondere für größere Programme hat sich dieses Konzept in vielen Programmiersprachen bewährt. dBASE II kennt keine lokalen Variablen.

Wir erläutern die Arbeit mit lokalen und globalen Variablen an Beispielen. Verwendet wird das Prozedurfile 'vars.prg'.

```
* Prozedurfile 'vars.prg'
* m_im_file ist lokal
m_im_file = 1
PROCEDURE p1
* m_p1 ist lokal
m_p1 = 1
? "m_p1 in p1= ",m_p1
DO p2
RETURN

PROCEDURE p2
* p2 benutzt m_p1 aus p1. Ist m_p1 immer sichtbar für p2 ?
? "m_p1 in p2= ",m_p1
RETURN

PROCEDURE p3
* m_im_file ist nie sichtbar in p3, denn p3 wird nicht vom
* Hauptteil aufgerufen.
? "m_im_file in p3: ", m_im_file
RETURN

PROCEDURE p4
* Eine globale Variable ist sichtbar
? "m_global in p4: ", m_global
RETURN
* Ende des Prozedurfiles 'vars.prg'
```

Wir weisen der Speichervariablen m_global einen Wert von der Tastatur aus zu und deklarieren sie damit als global.

```
. m_global = 2
2
* Das Prozedurfile wird eröffnet
. SET PROCEDURE TO vars
* Die globale Variable ist sichtbar
. DO p4
m_global in p4:      2
* Damit der Hauptteil von 'vars' aufgerufen werden kann, muß
* 'vars' als Prozedurfile abgeschlossen werden.
. CLOSE PROCEDURE
* m_im_file wird der Wert 1 zugewiesen
. DO vars
1
. SET PROCEDURE TO vars
. DO p3
Variable nicht gefunden
                                    ?
? "m_im_file in p3: ", m_im_file
Aufgerufen von - C:vars.prg
Abbruch der Programmdatei erwünscht? (J/N) Ja
DO abgebrochen
* Wir prüfen die Sichtbarkeit von m_p1 und m_p2
. DO p1
```

```
1
m_p1 in p1=          1
m_p1 in p2=          1
* Wird p2 von p1 aufgerufen, so kennt p2 auch den Wert von m_p1.
* Eine Variable ist also nicht lokal in einer Prozedur, sondern
* in einem File.
* Wird p2 durch ein Kommando von der Tastatur aus aufgerufen,
* so ist m_p1 in p2 nicht bekannt.
* Ursache: Eine Variable wird in dBASE durch eine Wertzuweisung
* deklariert!
. DO p2
Variable nicht gefunden
                     ?
? "m_p1 in p2= ",m_p1
Aufgerufen von - C:vars.prg
Abbruch der Programmdatei erwünscht? (J/N) Ja
DO abgebrochen
```

Lokale Variable werden mit der ersten Wertzuweisung erzeugt. Sie werden vernichtet, wenn die Programmsteuerung das zugehörige Prozedurfile verläßt. Sie bestehen also nicht so lange, wie das Prozedurfile eröffnet ist. Folglich können Zwischenergebnisse nicht in lokalen Variablen gespeichert werden, wenn zwischendurch das Prozedurfile verlassen wird. In diesem Fall muß eine globale Variable benutzt werden.

SAVE und RESTORE zum Auslagern von Speichervariablen in ein .mem-File bzw. zum Rückspeichern ausgelagerter Variabler sollten nicht in Prozeduren verwendet werden. Die Sichtbarkeit der so rückgespeicherten Variablen ist nicht definiert.

Wenden wir uns der Einschränkung der Sichtbarkeit durch das Kommando PRIVATE zu. Die Syntax ist

PRIVATE [ALL [LIKE/EXCEPT <maske>]] [<speichervar_liste>]

Die Wirkung wurde bereits beschrieben. Ist ALL abgegeben, so werden alle Variablen dieser Prozedur als privat erklärt. Ist ALL LIKE angegeben, so sind alle Variablen der Prozedur privat, auf die die Maske zutrifft. Ist ALL EXCEPT angegeben, so sind alle Variablen der Prozedur privat, auf die die Maske nicht zutrifft. Ist eine Liste von Speichervariablen angegeben, so sind sie in dort aufgeführten Variablen privat.

Wir betrachten die Wirkung von PRIVATE an einem Beispiel. Gegeben sei das Prozedurfile 'vars2.prg':

```
* Prozedurfile 'vars2.prg'
PROCEDURE p5
PRIVATE m_p5
PUBLIC  m_p6
m_p6 = 1
m_p5 = 2
DO p6
```

```
* m_p5 ist sichtbar
? "m_p5 in p5= ", m_p5
RETURN

PROCEDURE p6
* m_p5 ist sichtbar
? "m_p5 in p6= ", m_p5
RETURN
* Ende des Kommandofiles 'vars2.prg'
```

Die Wirkung sehen wir durch die folgenden Kommandos:

```
* Der globalen Variablen m_p5 wird eine Zeichenkette zugewiesen
. m_p5 = "Ina"
Ina
. SET PROCEDURE TO vars2
* In der Prozedur p5 wird der privaten Variablen m_p5 der Wert 2
* zugewiesen
. DO p5
1
2
m_p5 in p6=          2
m_p5 in p5=          2
* Der Wert der globalen Variablen m_p5 ist erhalten
. ? m_p5
Ina
* m_p6 ist durch PUBLIC global
. ? m_p6
        1
```

Laufvariablen und alle Variablen mit lokaler Bedeutung sollten als PRIVATE erklärt werden, um evtl. existierende globale Variable mit dem gleichen Namen zu schützen. Fehlt PRIVATE, so sind dadurch entstehende semantische Fehler nur schwer zu finden.

11.4. Parameter

Prozeduren können Parameter haben. Parameter sind Werte oder Variable, die an eine Prozedur übergeben werden (Eingabeparameter) bzw. in denen eine Prozedur Ergebnisse zurückgibt (Ausgabeparameter). Prinzipiell können solche Werte auch in globalen Variablen übergeben werden. Die Erfahrung mit Programmiersprachen lehrt jedoch, daß Programme, die mit vielen globalen Größen arbeiten, unübersichtlich sind. Nur solche Größen sollten global benutzt werden, die im gesamten Programm eine globale Bedeutung haben. Es hängt also von der Logik des Programms ab, ob eine Größe global oder lokal benutzt werden sollte.

Bei der Definition einer Prozedur sind ihre Parameter als Bezeichner anzugeben. Das Kommando PARAMETERS wird dazu benutzt:

PARAMETERS <par_liste>

Die Parameterliste besteht aus Bezeichnungen, die durch Komma getrennt sind. Sie sind nicht an einen Typ gebunden. Es ist sogar möglich, daß die gleiche Prozedur mit Parametern unterschiedlichen Typs aufgerufen wird. PARAMETERS muß das erste ausführbare Kommando einer Prozedur sein. Da diese Bezeichner nur in der Definition der Prozedur verwendet werden, bei der Abarbeitung aber stets durch aktuelle Werte oder Variablen ersetzt werden, heißen sie formale Parameter. Die beim Aufruf anzugebenden Parameter heißen dagegen aktuelle Parameter. Sie werden im DO-Kommando angegeben:

DO <proc_name> [**WITH** <par_liste>]

Die Zuordnung der aktuellen zu den formalen Parametern erfolgt nach ihrer Stellung in den Parameterlisten. Der erste aktuelle Parameter wird dem ersten formalen Parameter zugeordnet usw. Ist der aktuelle Parameter eine Speichervariable, so wird innerhalb der gerufenen Prozedur mit dieser Speichervariablen gearbeitet. Die Verwendung von Speichervariablen als aktuelle Parameter ist eine Möglichkeit, Ergebnisse einer Prozedur an das rufende Programm zurückzugeben. Alle anderen aktuellen Parameter werden als Ausdruck behandelt. Ihr Wert wird berechnet und der Prozedur übergeben.

Betrachten wir dazu ein Beispiel. Benutzt wird das Prozedurfile 'pars.prg'.

```
* Prozedurfile 'pars.prg'
* Prozedur mit 2 Eingabeparametern a, b
* und einem Ausgabeparameter result.
* Es werden zwei Werte a und b addiert. Das Ergebnis wird im
* Parameter result zurückgegeben.
PROCEDURE add
PARAMETERS a, b, result
  result = a + b
RETURN
* Ende des Prozedurfiles 'pars.prg'
```

Diese Prozedur wird nachfolgend mit Größen der Typen numerisch, String und Datum aufgerufen. Sollen Variable als aktuelle Parameter übergeben werden, so müssen diese Variablen vorher deklariert sein. Wir erreichen das durch Wertzuweisungen.

```
* Deklarieren von 3 Variablen
* Die Ausgabe der berechneten Werte wird unterdrückt
. SET TALK OFF
. m_n = 0
. m_s = ""
. m_d = DATE()
* Eröffnen des Prozedurfiles 'pars.prg'
. SET PROCEDURE TO pars
* Addieren von zwei numerischen Werten
. DO add WITH 4+2,6,m_n
* Das Ergebnis steht in m_n
```

```
. ? m_n
        12
* Addieren von zwei Zeichenketten ...
. DO add WITH "Parameter", "übergabe", m_s
* ... und besichtigen des Ergebnisses
. ? m_s
Parameterübergabe
* Addieren einer Anzahl von Tagen zu einem Datum
. DO add WITH DATE(), 100, m_d
. ? DATE(), m_d
18.02.88 28.05.88
* Der Aufruf einer Prozedur mit zu wenigen oder zu vielen Parametern
* führt zu einer Fehlermeldung
. DO add WITH 4,3
Anzahl der Parameter ungültig.
                  ?
PARAMETERS a, b, result
Wünschen Sie HILFE? (J/N) Nein
* Die gleiche Variable kann in einem Aufruf nicht Ein- und Ausgabe-
* parameter sein
. DO add WITH m_n, 1, m_n
Variable nicht gefunden
                        ?
DO add WITH m_n, 1, m_n
Wünschen Sie HILFE? (J/N) Nein
* Eine bereits deklarierte Variable kann einen anderen Typ erhalten, ...
. DO add WITH 2,4,m_s
. ? m_s
         6
* ... sie muß aber vorher deklariert sein
. DO add WITH LOG(2.71828), 0, m_neu
Variable nicht gefunden
                                  ?
DO add WITH EXP(2.71828), 0, m_neu
Wünschen Sie HILFE? (J/N) Nein
* Die Typverträglichkeit der Parameter untereinander wird bei der
* Ausführung der Kommandos geprüft, in denen sie benutzt werden
. DO add WITH "Donnerstag, den ", DATE(), m_s
Ungültiges Datenformat
             ?
  result = a + b
Aufgerufen von - C:pars.prg
Abbruch der Programmdatei erwünscht? (J/N) Ja
DO abgebrochen
```

12. dBASE und Editoren

12.1. Der interne Editor

dBASE benötigt gewisse Möglichkeiten der Textverarbeitung für die Bearbeitung von

Memo-Feldern,
Kommandofiles und
Formatfiles (Bildschirmmasken).

Standardmäßig wird für alle drei Aufgaben ein interner dBASE-Editor verwendet. Er ist für die Bearbeitung kleiner Texte gedacht, die keine Dokumente sind. Die maximale Textlänge, die bearbeitet werden kann, beträgt 4 KByte. Da dieser Editor nicht für die Bearbeitung von Dokumenten gedacht ist, fehlen solche Eigenschaften wie Randausgleich, Wortumbruch, Seitenformatierung, Trennhilfen, Kopf- und Fußzeilen usw., wie sie z.B. aus der Arbeit im Dokumentenmodus in WordStar oder einem ähnlichen Textverarbeitungssystem bekannt sind. Das ist aber kein Nachteil, da in dBASE nicht vorrangig Dokumente bearbeitet werden sollen. Im Vordergrund steht die Bearbeitung von Programmtexten.

Der Vorteil der Benutzung eines Editors aus dBASE heraus besteht darin, daß alle Einstellungen in dBASE erhalten bleiben. Das betrifft die in Benutzung befindlichen Datenbank- und Indexfiles, die wahlweisen Einstellungen über SET usw.

Nachteilig ist hingegen, daß die Bedienung des internen Editors etwas von der anderer Editoren abweicht. Bei dBASE III hat man sich im Vergleich zu dBASE II der allgemein üblichen Bedienung genähert, indem die acht Kursorpositionierungstasten (4 Pfeile, **Home**, **End**, **PgUp**, **PgDn**) in üblicher Weise benutzt werden. Außerdem sind die aus dem dBASE II-Editor bekannten Steuerfunktionen gültig. Die Funktionen im einzelnen sind im Anhang 2 angegeben.

```
MODIFY COMMAND <filename>
```

leitet das Editieren des Files <filename> ein. Ist keine Namenserweiterung angegeben, so wird '.prg' angenommen. Wollen wir ein Textfile mit einer anderen Namenserweiterung bearbeiten, so muß diese angegeben werden. Die Bildschirmbedienung geht in den Direktmodus über.

Während der Arbeit mit dem internen Editor stehen keine Hilfen zur Verfügung, die eine Auskunft über die möglichen Steuerfunktionen geben.

Aus all diesen Gründen ist es wünschenswert, daß man in dBASE verbleibt, aber trotzdem "seinen privaten" Editor benutzt. dBASE unterstützt diese Arbeitsweise durch die Konfigurierung von dBASE beim Aufruf. Dazu wird ein File 'config.db' verwendet. Im Abschnitt 17.2 werden wir näher auf die Möglichkeiten der Konfigurierung von dBASE eingehen.

12.2. Einbinden eines privaten Editors

Hierbei ist es erforderlich, zwischen dem Editieren von Memo-Feldern und dem Editieren von ganzen Files (Kommandofiles, Formatfiles, sonstige Textfiles) zu unterscheiden. Ein privater Editor für die Bearbeitung von Files wird in dBASE eingebunden, indem in das Konfigurationsfile 'config.db' die Zeile

```
TEDIT = <editor>
```

aufgenommen wird. <editor> ist dabei der Name, der auch sonst beim Aufruf des Editors auf der Betriebssystemebene angegeben wird. Die Fileerweiterung '.com', '.exe' oder auch '.bat' kann fehlen. Wird MODIFY COMMAND aufgerufen, so meldet sich dieser Editor mit all seinen bekannten Eigenschaften.

```
TEDIT = ne
```

bindet z.B. den NORTON-Editor in dBASE ein.

Die Beziehungen des eingebundenen Editors zu seiner Umgebung sind dabei zu berücksichtigen. Dazu gehört entscheidend die Arbeit in hierarchischen Fileverzeichnissen. Benutzt der Editor Überlagerungen? Über welchen Pfad werden sie gefunden? Akzeptiert der Editor Pfadausdrücke im Dateinamen usw.?

Bindet man z.B. WordStar ein, so ist zu beachten, daß WordStar seine Überlagerungen als Datenfiles sucht, also nicht über den mit PATH im MS-DOS eingestellten Suchpfad für Programmdateien, sondern über den Suchpfad für Nicht-Programmdateien. Dieser wird über das MS-DOS-Kommando APPEND (ab Vers. 3.20) eingestellt. Das wiederum macht den Aufruf einer Kommandodatei (.bat) des MS-DOS erforderlich. WordStar meldet sich dann mit seinem Anfangsmenü. Hier muß die Funktion zum Editieren (meist N) ausgewählt und der Dateiname noch einmal eingegeben werden. WordStar findet zu dBASE zurück. Die Einstellungen von dBASE bleiben erhalten.

Insgesamt erscheint es günstiger, einen kleinen und leistungsfähigen Editor für Programmdateien einzubinden, z.B. den NORTON-Editor, als einen umfangreichen Texteditor, von dem man doch nur wenige Funktionen in dBASE benötigt.

Zu beachten ist ferner, daß Texteditoren bestimmte Sonderzeichen einfügen bzw. nicht akzeptieren. So speichert WordStar z.B. die Zeichen des erweiterten Zeichensatzes, wozu auch die Umlaute gehören, in einer für WordStar spezifischen Form ab (physisch als drei Zeichen). Das gilt auch für den N-Modus (non document mode)!

Hat man den NORTON-Editor eingebunden, so muß man sich damit abfinden, daß dieser keine Umlaute verträgt. Das war für dBASE eine Eigenschaft, die uns bei der Wahl zwischen der deutschen und der englischen Version die deutsche bevorzugen ließ. An der Behandlung der Umlaute in den Feldern und als Feldbezeichnungen ändert sich dadurch natürlich nichts.

Die Behandlung der Tabulatoren ist in den Editoren ebenfalls

unterschiedlich. So speichert z.B. der NORTON-Editor einen Tabulator als ein Zeichen, während andere Editoren, so auch der interne dBASE-Editor, Tabulatoren durch Leerzeichen ausdehnen.

Wir sehen, daß Texte entstehen können, die u.U. mit anderen Editoren nicht richtig verarbeitet werden. Für den Austausch von Datenbankfiles zwischen verschiedenen dBASE-Versionen mit unterschiedlichen Texteditoren ist das zu berücksichtigen.

Besser noch als das Einbinden eines privaten Editors erscheint die Nutzung des dBASE-Editors und die wahlweise Nutzung anderer Editoren, ohne dBASE neu zu starten. Man nutzt von jedem Editor nur die Vorteile. Das wird erreicht, indem der gewünschte Editor über das Kommando RUN geladen wird (Abschn. 17.3).

```
RUN ws
```

ruft WordStar aus dBASE heraus auf. Alle Einstellungen von dBASE bleiben erhalten. Das gewünschte File wird bearbeitet. Jetzt sind z.B. auch die Blockoperationen aus WordStar möglich. Bezüglich der Zeichendarstellung und der Sonder- und Steuerzeichen gilt das oben Gesagte.

Betrachten wir nun die Bearbeitung von Memo-Feldern. Durch

```
WP = <editor>
```

im Konfigurationsfile 'config.db' bindet man einen privaten Editor für die Behandlung von Memo-Feldern ein. Prinzipiell gilt das gleiche wie beim Einbinden eines privaten Editors für MODIFY COMMAND. Zu beachten ist jedoch, daß hier immer nur Teile eines Files, des '.dbt'-Files, bearbeitet werden, und das in einer Weise, die von dBASE bestimmt wird. So kann WordStar für diesen Zweck z.B. nicht verwendet werden. Die Programmsteuerung gelangt zwar zu WordStar, wir befinden uns aber im Anfangsmenü und müssen jetzt über die Funktion N ein File bearbeiten. Wir wollen aber nur Teile eines Files editieren. Das ist so nicht möglich.

Möglich ist hingegen das Einbinden des NORTON-Editors. Auch hier liegt in der Verantwortung des Nutzers, die Verträglichkeit der benutzten Editoren zu überprüfen.

Der für das Editieren von Memo-Feldern eingebundene Editor wird aufgerufen, indem bei APPEND, EDIT oder einem ähnlichen Kommando der Kursor auf dem Memo-Feld steht und ^Home eingegeben wird. Die Rückkehr aus dem Editor zu dBASE erfolgt durch das vom Editor verlangte Kommando für die Beendigung seiner Arbeit.

13. Eingabe und Ausgabe

Wir wissen bisher, daß man einfache Zusammenstellungen als Kopie der sequentiellen Bildschirmausgabe (hard copy) auf dem Drucker ausgeben kann. ^P, auf der Kommandoebene von dBASE eingegeben, leitet die Kopie ein und schaltet sie auch wieder ab. Bildschirminhalte können durch Drücken der Taste **PrtSc** (print screen) gedruckt werden. Textfiles lassen sich durch SET ALTERNATE und COPY TO ... SDF erzeugen. Mit Berichten in Tabellenform haben wir uns im Abschnitt 10 beschäftigt.

Eingabemöglichkeiten sind uns ebenfalls bekannt. Durch APPEND, BROWSE, EDIT, CHANGE können im Direktmodus von der Tastatur aus Daten eingegeben und korrigiert werden. APPEND FROM, UPDATE und JOIN sind weitere Möglichkeiten.

Hier betrachten wir die Kommandos, die im Normalfall in einem Kommandofile stehen und Daten ausgeben und als Eingaben von der Tastatur verlangen. Sie dienen in ihrer Gesamtheit dazu, eine Nutzeroberfläche zu schaffen, die dem Anwenderproblem näher liegt als die Kommandoebene von dBASE.

13.1. Sequentielle Eingabe und Ausgabe

Betrachten wir zuerst die sequentielle Ausgabe. Drei Kommandos stellt dBASE dafür zur Verfügung:

```
? <ausgabeliste>
?? <ausgabeliste>
TEXT
  ...
ENDTEXT
```

Durch

```
SET PRINT ON
```

werden all diese Ausgaben auf dem Drucker und auf dem Bildschirm ausgegeben.

"?" benutzten wir in den ersten Abschnitten im Sinne von "Was ist ...?". Basic-Nutzer kennen das Fragezeichen als verkürzte Schreibweise für das Schlüsselwort PRINT. Die gleiche Deutung erweist sich für dBASE als günstig.

"?" berechnet die in der Ausgabeliste aufgeführten Ausdrücke, gibt ein CR und LF (carriage return, line feed) und danach die berechneten Werte im Standardformat aus. "??" wirkt wie "?", unterdrückt aber den Zeilenwechsel. Standardformat heißt bei Feldern das durch die

Struktur des Datenbankfiles gegebene Format. Bei Speichervariablen vom numerischen Typ ist es das voreingestellte Format, ggf. in 'config.db' festgelegt. In der Ausdrucksliste sind beliebige Ausdrücke von dBASE zugelassen (Abschn. 4).

```
TEXT
  <text>
ENDTEXT
```

ist nur in Kommandofiles gestattet. Der Text wird unverändert ausgegeben. Es wird auch keine Makroersetzung im Text durchgeführt (im Gegensatz zu "?", wo eine Makroersetzung stattfindet). Der Text wird im Kommandofile ohne Zeichenkettenbegrenzer geschrieben. TEXT und ENDTEXT müssen die ersten Wörter einer Zeile sein. Zeichen, die nach TEXT in der gleichen Zeilen stehen, werden bei der Ausgabe ignoriert.

Die folgende Kommandofolge zeigt die Wirkung:

```
* Wir benutzen unser Datenbankfile ...
. USE lager
* ... und definieren zusätzlich eine logische Variable.
. * logische Variable
. m_l = .t.
.T.
. ? artikel+":", "Lieferung am", lieferung, ". Einzelpreis=",preis
Flachschieber  : Lieferung am 29.02.88 . Einzelpreis=     2.40
* Ausgabe logischer Werte
. ? m_l, preis > 5.00
.T. .F.
* Ausgabe von Memo-Feldern
. ? bem
Monatliche Lieferung.
* Wir definieren Stringvariable
. m_s = "mein File"
mein File
* In "?" wird eine Makroersetzung durchgeführt
. ? "Das ist &m_s.."
Das ist mein File.
* 'text.prg' ruft TEXT  ... ENDTEXT auf.
* Es erfolgt keine Makroersetzung.
. DO text
  Das ist &m_s.
  Die Eingabe von Texten, die aus mehreren Zeilen bestehen,
  erledigt man besser  mit TEXT und ENDTEXT als mit dem
  Kommando "?".
```

Das Kommandofile 'text.prg' enthält:

```
TEXT
  Das ist &m_s.
  Die Eingabe von Texten, die aus mehreren Zeilen bestehen,
  erledigt man besser  mit TEXT und ENDTEXT als mit dem
  Kommando "?".
ENDTEXT
```

Für die sequentielle Eingabe von Werten für Speichervariable stehen uns die Kommandos ACCEPT, INPUT und WAIT zur Verfügung.

ACCEPT [<prompt>] **TO** <mem_var>

INPUT [<prompt>] **TO** <mem_var>

Man beachte, daß durch ACCEPT und INPUT keine Werte an Felder eines Datenbankfiles zugewiesen werden können!

ACCEPT akzeptiert nur Werte vom Typ String, also Zeichenketten. Dafür müssen diese Zeichenketten nicht in Zeichenkettenbegrenzer eingeschlossen werden. Wird also eine Zeichenkette erwartet, so sollte ACCEPT verwendet werden.

Ist hingegen nicht sicher, von welchem Typ der eingegebene Wert ist, so sollte man INPUT benutzen. INPUT akzeptiert auch Zeichenketten. Sie müssen allerdings in Begrenzer eingeschlossen sein.

Die angegebene Speichervariable wird mit einer Eingabeanweisung immer neu deklariert. Ihr Typ richtet sich dann nach dem eingegebenen Wert. Ist die Speichervariable bereits definiert, so wird sie ohne irgendeine Warnung neu deklariert. Ihr alter Wert ist damit verloren. Hierin liegt eine große Gefahr! Eine implizite Typkonvertierung kann deshalb nicht erfolgen. So ist es auch nicht möglich, Konstanten vom Typ Datum einzugeben. Der Typ Memo scheidet ohnehin aus, da Speichervariablen nicht vom Typ Memo sein können.

Wir zeigen das an einer Kommandofolge:

```
. INPUT "Bitte Zahl eingeben: " TO m_n
Bitte Zahl eingeben: 12.456
* Wir testen den Wert ...
. ? m_n
       12.456
* ... und den Typ.
. ? TYPE("m_n")
N
. ACCEPT "Bitte Ziffernfolge eingeben: " TO m_s
Bitte Ziffernfolge eingeben: 12.456
* Der Wert ist vom Typ String. Deshalb wird die Zeichenkette
* auch linksbündig ausgegeben.
. ? m_s
12.456
. ? TYPE("m_s")
C
* m_s wird neu deklariert durch INPUT.
. INPUT "Bitte Zahl eingeben: " TO m_s
Bitte Zahl eingeben: 122.7818
. ? m_s
        2.7818
. ? TYPE("m_s")
N
* Das gleiche geschieht mit einer Variablen vom Typ Datum.
. m_d = DATE()
03.01.88
```

```
. ACCEPT "Bitte Datum eingeben: " TO m_d
Bitte Datum eingeben: 23.11.88
. ? m_d
23.11.88
. ? TYPE ("m_d")
C
* Aber eine Konvertierung ist natürlich möglich.
. ? CTOD(m_d) + 100
03.03.89
* Sollen logische Werte eingegeben werden, so müssen sie in Punkte
* eingeschlossen sein
. input 'Wert:' TO m_l
Wert:T
Variable nicht gefunden
Wert:.T.
* .T., .t., .Y., .y., .F., .f., .N. und .n. werden akzeptiert
. input 'Wert:' TO m_l
Wert:.y.
* .J. wird auch in der deutschsprachigen Version nicht akzeptiert
. input 'Wert:' TO m_l
Wert:.J.
Syntaxfehler
Wert:.y.
```

WAIT wartet auf eine Eingabe, oft nur auf das Drücken einer Taste. Ist eine Speichervariable angegeben, so wird das Zeichen, welches der Taste entspricht, zum Wert der Speichervariablen. Sie hat den Typ String und die Länge 1. Wird eine Taste gedrückt, die kein druckbares Zeichen darstellt, z.B. **Enter**, so wird der Wert 00h in der Speichervariablen abgelegt. Die Syntax von WAIT ist

WAIT [<prompt>] [TO <mem_var>]

Fehlt die Anforderungszeichenkette <prompt>, so wird eine Standardmitteilung ausgegeben. WAIT ohne Speichervariable wird häufig verwendet, um die Bildschirmausgabe zu stoppen und dem Nutzer die Gelegenheit zu geben, den ausgegebenen Text zu lesen. Hat er dies getan, so bestätigt er das durch Drücken einer beliebigen Taste.

WAIT mit Speichervariable ist günstig einsetzbar, wenn eine Auswahl aus mehreren Angeboten durch die Eingabe eines Zeichens getroffen werden kann. Wir betrachten dazu das folgende Beispiel. Gegeben sei das Kommandofile 'wait.prg':

```
* Kommandofile 'wait.prg'
SET TALK OFF
* Diese Variable steuert den Zyklus
m_zyklus = "t"
DO WHILE m_zyklus = "t"

   TEXT
        0: Ende der Bearbeitung
```

```
            1: Ausgabe der Ersatzteile alphabetisch geordnet
            2: Ausgabe der Ersatzteile einschl. der Adressen
               der Hersteller
      ENDTEXT

   WAIT "Bitte geben Sie die Ziffer der gewünschten Funktion ein:
      TO m_wahl
      DO CASE
         CASE m_wahl = "0"
           ? "Fall 0"
           m_zyklus = "f"
        CASE m_wahl = "1"
           ? "Fall 1"
        CASE m_wahl = "2"
           ? "Fall 2"
         OTHERWISE
           ? "Bitte Eingabe wiederholen
      ENDCASE
   ENDDO
   RETURN
   * Ende des Kommandofiles 'wait.prg'
```

Statt der Ausschriften für die einzelnen Fälle stehen sonst natürlich die entsprechenden Kommandofolgen. Eine mögliche Benutzung ist

```
. DO wait
       0: Ende der Bearbeitung
       1: Ausgabe der Ersatzteile alphabetisch geordnet
       2: Ausgabe der Ersatzteile einschl. der Adressen
          der Hersteller

Bitte geben Sie die Ziffer der gewünschten Funktion ein: 2
Fall 2
```

Dieses Menü wird wiederholt, bis durch Eingabe einer Null der Zyklus und das Kommandofile verlassen werden.

13.2. Eingabe und Ausgabe im Direktmodus

Das Ziel besteht darin, auf dem Bildschirm ein stehendes Bild zu erzeugen, in dem sich Anforderungszeichenkette und Eingabebereiche abwechseln. Innerhalb des gesamten Bildschirms ist der Kursor in den Eingabefeldern - und nur dort - frei beweglich. Eine Bildschirmzeile kann mehrere Anforderungen und Eingabefelder enthalten. Die eingegebenen Werte können Speichervariablen und Feldern eines Datenbankfiles zugewiesen werden. Wir erreichen dadurch

1. einen Bildschirmaufbau, der Karteikarten ähnelt,
2. daß nur die Daten angefordert werden, die der Nutzer verändern darf,

3. daß die Anforderungstexte unabhängig von den Feldnamen des Datenbankfiles werden,
4. daß der Nutzer die meisten dBASE-Kommandos gar nicht kennen muß. Er muß nicht einmal die Struktur der Datenbank kennen. Wir geben ihm nur die Werkzeuge in die Hand, die er für seine Zwecke, z.B. Datenerfassung, benötigt.

Mit einem Wort: Wir schaffen eine Benutzeroberfläche, die der Problemstellung adäquat ist. Verschiedene Nutzer können mit verschiedenen Nutzeroberflächen, aber der gleichen Datenbank arbeiten. Jeder Endnutzer benötigt nur Kenntnisse über sein Gebiet.

Wichtigstes Kommando für diese Arbeitsweise ist das @-Kommando:

```
@ <zeile>,<spalte> [CLEAR]
@ <zeile>,<spalte> [SAY <ausdr> [PICTURE <klausel>]]
          [[GET <var> [PICTURE <klausel>]
          [RANGE <ausdr>,<ausdr>]]
```

<zeile> und <spalte> geben die Zeile bzw. Spalte auf dem Bildschirm oder Drucker an. Beide Werte können durch Ausdrücke gegeben sein. Bedingt durch die Gerätetechnik sind Grenzen für diese Werte gesetzt. Für einen normalen Bildschirm mit 24 Zeilen und 80 Zeichen je Zeile sind das:

```
0 <= <zeile>  <= 23        und
0 <= <spalte> <= 79
```

Beide Werte bestimmen die Position des ersten auszugebenen Zeichens.

Die auszugebende Zeichenfolge wird durch die Berechnung des Ausdrucks nach SAY ermittelt. Ausdrücke aller Typen sind hier möglich. Für die Ausgabe werden sie, wie üblich, in eine Zeichenkette konvertiert.

Die Klauseln hinter PICTURE bestimmen das Aus- bzw. Eingabeformat. Es gibt Formatfunktionen und Formatzeichen. Sie sind im Abschnitt 18 zusammengestellt.

Die Ausdrücke hinter RANGE sind eine untere und obere Grenze. Bei der Eingabe wird geprüft, ob der eingegebene Wert innerhalb dieser Grenzen liegt. Dieser Grenzentest ist für Größen der Typen Datum und numerisch möglich.

<var> nach dem Schlüsselwort GET kann eine Speichervariable oder ein Feld sein.

CLEAR löscht den Bildschirm rechts von <spalte> und unterhalb von <zeile>. Fehlen alle wahlfreien Angaben, so wird die ausgewählte Zeile ab der angegebenen Spalte gelöscht.

Untersuchen wir zuerst die direkte Ausgabe auf dem Bildschirm.

```
* Wir benutzen unser Datenbankfile 'lager.dbf'
. USE lager
* In die nächste Zeile soll der Preis ab Position 10 ausgegeben werden
.cp 2
```

```
. @ ROW()+1, 10 SAY preis
              2.40
* Die Formatangabe wirkt hier so wie erwartet.
. @ ROW()+1, 10 SAY preis PICTURE "###.#"
            2.4
* Eine Erweiterung der Anzahl der Dezimalstellen ist nicht möglich.
* Der Wert wird ohne Fehlermeldung abgeschnitten.
. @ ROW()+1, 10 SAY preis PICTURE ".###"
           .40
* Auch Zusammensetzungen sind möglich.
. @ ROW()+1, 5 SAY "Ein "+artikel+"kostet "+STR(preis,6,2)+" Mark"
    Ein Flachschieber  kostet   2.40 Mark
* Auch Felder verbundener Datenbankfiles können ausgegeben werden.
. SELECT 2
. USE lageradr INDEX laadrh
. SELECT lager
. SET RELATION TO hersteller INTO lageradr
. @ ROW()+1, 10 SAY lageradr->ort + lageradr->str
          Rheinsberg  M.-Henning-Str. 7
```

Versuchen wir nun Werte einzulesen:

```
* Wir schalten die Ausgabe der berechneten Werte aus
. SET TALK OFF
* Einem Feld kann durch @ ein Wert zugewiesen werden.
* Das Feld bestimmt das Format, falls es nicht durch PICTURE
* weiter eingeschränkt wird.
. @ ROW()+1,10 SAY "Neuer Preis: " GET preis
          Neuer Preis:  :▒▒▒▒2.45:
* Nach dem READ springt der Kursor auf das erste Zeichen des Eingabefeldes.
. READ
* Der neue Wert wurde eingegeben.
* Wir versuchen einen Wert in eine Speichervariable einzulesen.
. @ ROW()+1, 10 SAY "Neuer Preis: " GET m_preis
Variable nicht gefunden
                                                ?
. @ ROW()+1, 10 SAY "Neuer Preis: " GET m_preis
 Wünschen Sie HILFE? (J/N) Nein
* Diese Ausschrift erscheint, da m_preis noch nicht definiert ist,
* d.h., jede in GET benutzte Variable muß vorher definiert sein.
. m_preis = 0.0
. @ ROW()+1, 10 SAY "Neuer Preis: " GET m_preis
          Neuer Preis: :▒▒▒▒▒▒▒▒▒▒▒▒0.0:
* Der Typ von m_preis ist durch die Zuweisung des Wertes
* 0.0 festgelegt worden. Er kann durch GET nicht verändert werden.
* Man beachte den Unterschied zu ACCEPT und INPUT!
* Auch eine Formatangabe nach GET kann zwar das Format weiter
* beschränken, aber nicht erweitern.
. @ ROW()+1, 10 SAY "Neuer Preis: " GET m_preis PICTURE "###.##"
          Neuer Preis:  :▒▒2.4:
* Die Zuweisung zu Feldern eines verbundenen Datenbankfiles
```

```
* ist nicht möglich
. SELECT 2
. USE lageradr INDEX laadrh
. SELECT lager
. SET RELATION TO hersteller INTO lageradr
. @ ROW()+1, 10 GET lageradr->plz
Variable nicht gefunden
                                    ?
@ ROW()+1, 10 GET lageradr->plz
Wünschen Sie HILFE? (J/N) Nein
```

Das @-Kommando gibt uns auch die Möglichkeit, Bildschirmmasken zu definieren. Sie wirken dann auf alle Befehle, die Eingaben im Direktmodus beinhalten, also auf APPEND, EDIT usw. Bildschirmmasken bestehen aus einer Folge von @-Kommandos, die in einem Formatfile mit der Namenserweiterung '.fmt' gespeichert werden. Wir erläutern die Wirkung an einem Beispiel.

Wir möchten einen Erfassungsbeleg für Ersatzteile erstellen, der nur einige Daten enthält und andere vor dem Erfasser verbirgt und der die Form einer möglichen Karteikarte hat. Dazu erstellen wir das Formatfile 'labeleg.fmt':

```
* Formatfile 'labeleg.fmt'
* CLEAR löscht den Bildschirm
@ 0,0 CLEAR
.cp 4
@ 4, 20 SAY "╔═══════════════════════════════════════╗"
@ 5, 20 SAY "╟ Erfassungsbeleg für neue Ersatzteile,╢"
@ 6, 20 SAY "╟ die bisher noch nicht erfaßt sind    ╢"
@ 7, 20 SAY "╚═══════════════════════════════════════╝"
@ 9, 8  SAY "Neues Ersatzteil: " GET artikel PICTURE "XXXXXXXXXXXXXX"
@ 11,8  SAY "Anzahl der eingegangenen Teile: " GET anzahl PICTURE "####"
@ 11,50 SAY "Einzelpreis: " GET preis PICTURE "####.##"
.cp 3
@ 15,20 SAY "Angaben zum Hersteller"
@ 16,20 SAY "————————————————————"
@ 18,20 SAY "Name der Firma: " GET hersteller
* Straße, Ort und PLZ sind in einem anderen Datenbankfile
* und müssen deshalb erst in Speichervariablen erfaßt und später
* durch REPLACE dort eingetragen werden.
@ 20,20 SAY "Straße:          " GET m_str PICTURE "XXXXXXXXXXXXXXXXXXXX"
@ 21,20 SAY "Stadt/Ort:       " GET m_ort PICTURE "XXXXXXXXXXXX"
@ 22,20 SAY "Postleitzahl:    " GET m_plz PICTURE "9999"
* Ende des Formatfiles
```

Wir geben folgende Kommandos ein:

```
. USE lager
* Die Speichervariablen müssen mit dem richtigen Typ
* und in der richtigen Länge definiert werden
. m_str = SPACE(20)
. m_ort = SPACE(12)
```

```
. m_plz = "
.* Doppelpunkte sollen die Eingabefelder begrenzen (Standard)
. SET DELIMITER ON
.* Wird eine Eingabe im Direktmodus verlangt, so soll 'labeleg.fmt'
.* als Formatfile benutzt werden
. SET FORMAT TO labeleg
. EDIT 1
```

Zum Editieren wird ein Bildschirminhalt angeboten, wie er im Bild 13.1 wiedergegeben ist.

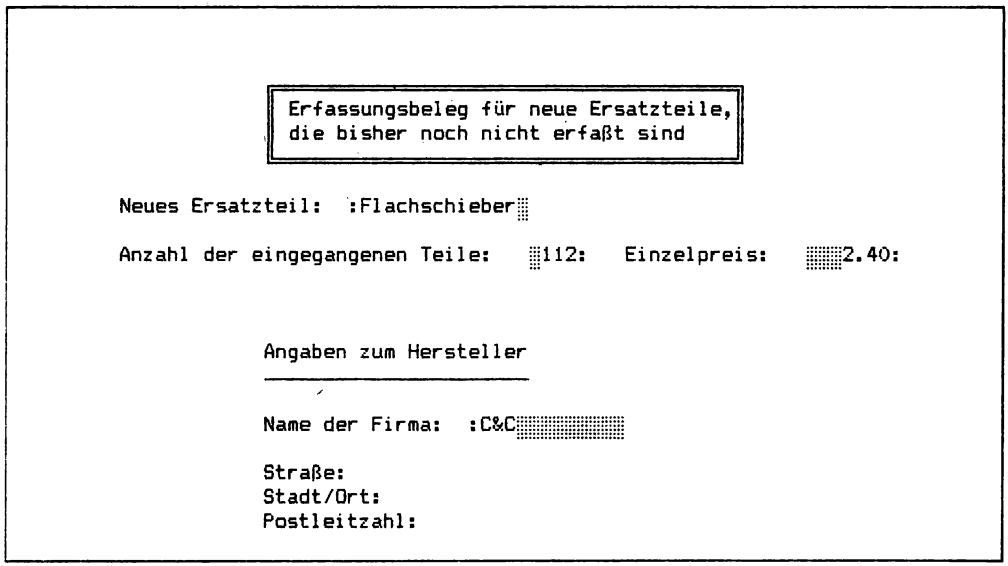


Bild 13.1. Eine private Bildschirmmaske

Wir sehen an diesem Beispiel, daß auch Sonderzeichen in der Maske enthalten sein können, wie Ränder, Schraffur u.ä. Die Bedienung ist ansonsten genauso wie bei der Benutzung von EDIT mit der Standard-bildschirmmaske. Für die Erstellung von Bildschirmmasken gibt es ein separates Programm 'dFORMAT' (Abschn. 20.3).

14. Arbeit mit mehreren Datenbankfiles

dBASE III gestattet die Benutzung von maximal 10 Arbeitsbereichen. Die Arbeitsbereiche werden mit den Zahlen 1 bis 10 oder mit den Buchstaben A bis J bezeichnet. Wird dBASE gestartet, so befinden wir uns im ersten Arbeitsbereich. Durch

```
SELECT i
```

oder

```
SELECT b,
```

wobei i die Nummer eines Arbeitsbereichs bzw. b der Buchstabe eines Arbeitsbereichs ist, wechselt man in den so bezeichneten Arbeitsbereich.

Der Arbeitsbereich, in dem wir uns befinden, heißt aktiv. In jedem Arbeitsbereich kann ein Datenbankfile mit seinen Indexfiles, Formatfiles usw. in Benutzung sein, wobei die in Abschnitt 2.7 angegebenen Grenzen, insbesondere für die gleichzeitig eröffneten Files, nicht überschritten werden dürfen.

Jeder Arbeitsbereich hat einen separaten Dateizeiger, der durch den Wechsel zwischen den Arbeitsbereichen und die explizite oder implizite Bewegung des Dateizeigers in anderen Arbeitsbereichen nicht beeinflußt wird. Eine Ausnahme bilden logisch verbundene Datenbankfiles, auf die wir am Ende dieses Abschnitts genauer eingehen.

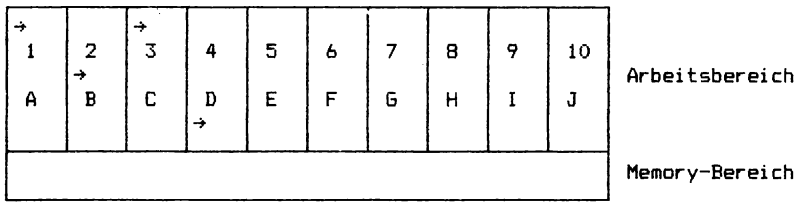


Bild 14.1. Arbeitsbereiche und Speichervariable

Bild 14.1 zeigt in symbolischer Form die zehn Arbeitsbereiche mit angedeuteten separaten Dateizeigern und den Bereich der Speichervariablen. Von jedem Arbeitsbereich aus kann auf die Speichervariablen zugegriffen werden. Es existiert nur ein Bereich für die Speichervariablen. Ihre Eigenschaften (global - lokal, deklariert - nicht deklariert, geladen - nicht geladen) hängen nicht von den in den Arbeitsbereichen in Benutzung befindlichen Datenbankfiles ab, sondern

von der Struktur der Kommandofiles und der Reihenfolge der Kommandoabarbeitung. Ihre Existenz (Deklaration, Auslagerung, Einlagerung und Vernichtung) wird durch Kommandos bestimmt.

Solange in einem Arbeitsbereich ein Datenbankfile in Benutzung ist, kann man sich auf diesen Arbeitsbereich durch den Namen dieses Datenbankfiles beziehen. Dieser Name ist dann ein sog. Aliasname für diesen Bereich. Das ist sehr vorteilhaft, da man ohnehin vergißt, in welchem Arbeitsbereich welches Datenbankfile in Benutzung ist. Im USE-Kommando kann auch ein anderer Aliasname für ein Datenbankfile vereinbart werden. Die Zahlen und Buchstaben für die Arbeitsbereiche benötigt man nur bei der ersten Auswahl eines Arbeitsbereichs, wenn noch kein Datenbankfile in Benutzung ist.

Man bezieht sich auf ein Feld des aktiven Datenbankfiles durch seinen Namen. Auf diese Weise sind Wertzuweisungen (REPLACE, CHANGE) und Zugriffe zum Wert möglich. Auf den Inhalt eines Feldes in einem anderen Arbeitsbereich greift man zu, indem dem Feldnamen der entsprechende Aliasname vorangestellt wird. Das ist der Name des Datenbankfiles, zu dem dieses Feld gehört. Beide Angaben werden durch die beiden Zeichen '->' verbunden.

Sei z.B. das Datenbankfile 'lageradr.dbf' im Arbeitsbereich 2 in Benutzung. 'lageradr' enthält ein Feld 'Ort'. Aus einem anderen Arbeitsbereich greift man auf dieses Feld zu durch

```
lageradr->ort
```

Während der Arbeit, auch innerhalb eines Kommandofiles, können in einem Arbeitsbereich nacheinander mehrere Datenbankfiles in Benutzung genommen werden. Die zehn möglichen Arbeitsbereiche erscheinen nur als eine theoretische Grenze. dBASE II läßt nur zwei Arbeitsbereiche zu, was in keinem bekannten praktischen Anwendungsfall eine ernsthafte Beschränkung war. Die vielen Arbeitsbereiche von dBASE III bieten gegenüber dBASE II den Vorteil, daß die Positionierung innerhalb des Datenbankfiles erhalten bleibt. Somit entfällt das Abschließen der alten Datei, das Eröffnen der neuen Datei und die Positionierung, wie es in dBASE II erforderlich ist.

Betrachten wir als ein Beispiel der Arbeit mit mehreren Arbeitsbereichen das logische Verbinden von Datenbankfiles. Wir haben im Abschnitt 5 betont, daß Daten, die eigentlich zusammengehören, vorteilhaft in verschiedenen Files abgelegt werden können. Das hat offenbar Auswirkungen auf die Struktur unserer gesamten Datenbank und spart mitunter Speicherplatz.

Stellen wir uns vor, daß Briefe an die Hersteller unserer Ersatzteile zu schicken sind. Im Datenbankfile 'lager.dbf' haben wir nur die Firmenbezeichnung eingetragen. Zu einer Adresse gehören aber noch Ort, Straße, Postleitzahl, ggf. eine Abteilung und vielleicht noch weitere Angaben. Nehmen wir all diese Angaben in unser Datenbankfile 'lager' auf, so erhöht sich die Länge eines Datensatzes erheblich, was eigentlich nicht notwendig ist. Wir sehen, daß gleiche Hersteller mehrfach auftreten. Es ist also günstiger, d.h. platzsparender, wenn man sich eine separate Adreßliste anlegt. So würde man in der konventionellen Arbeitsweise, also ohne dBASE, auch verfahren. Es gibt dann

einen eindeutigen Bezug zwischen einem Satz in der Datei 'lager', z.B. die Firmenbezeichnung, und einem Satz in der Adreßdatei. Ein Feld der Adreßdatei enthält ebenfalls die Firmenbezeichnung. Man sucht also anhand einer Firmenbezeichnung, die man der Datei 'lager' entnimmt, einen Satz in der Adreßdatei, der die gleiche Firmenbezeichnung enthält. Man sagt in diesem Fall, daß beide Datenbankfiles über das Feld 'Hersteller' logisch verbunden sind.

Für das Suchen von Datensätzen kennen wir bereits die Kommandos LOCATE, FIND und SEEK (Abschn. 9), wobei FIND und SEEK über Indexdateien schneller sind als LOCATE. Das logische Verbinden zweier Datenbankfiles realisiert intern ein FIND-Kommando.

Bewegt sich der Nutzer mit dem Dateizeiger durch das Datenbankfile 'lager', so wird sofort der Dateizeiger im Datenbankfile für die Adressen 'lageradr.dbf' auf den zugehörigen Datensatz gestellt. Syntaktisch wird diese logische Verbindung zweier Datenbankfiles über ein SET-Kommando hergestellt:

SET RELATION TO <index_wert> **INTO** <alias>

Betrachten wir die verbal beschriebene Wirkung in Form von dBASE-Kommandos. Die Struktur des Adreßfiles ist

```
. USE lageradr
. LIST STRUCTURE
Datenbankstruktur      - C:lageradr.dbf
Anzahl der Datensätze  -        6
Letztes Änderungsdatum - 23.12.87
Feld   Feldname    Typ         Länge    Dez
    1  ID          Zeichen        3
    2  HERSTELLER  Zeichen       12
    3  ABTEILUNG   Zeichen       12
    4  PLZ         Zeichen        4
    5  ORT         Zeichen       12
    6  STR         Zeichen       20
** Gesamt **                     64
```

Es sind die folgenden Adressen gespeichert:

```
LIST
Satz #  ID  HERSTELLER   ABTEILUNG     PLZ  ORT          STR
     1  A&W Achse&Welle               7902 Annaburg     Schulstraße 111
     2  B&C Blech&Co.    Auslieferung 8036 Dresden      Herzberger Straße 12
     3  C&C C&C                       1955 Rheinsberg   M.-Henning-Str. 7
     4  D&F Dicht&Fest   Absatz       1092 Berlin       Suermondtstr. 43
     5  G&S Gummi&Sohn   Absatz       7500 Cottbus      Straße der Jugend 17
     6  Guß VEB Guß                   1140 Berlin       Kienbergstr. 55
```

Unter diesen Voraussetzungen wollen wir beide Datenbankfiles miteinander verbinden und in einem LIST-Kommando Felder beider Datenbanken auslisten.

```
. USE lager
. SELECT B
. USE lageradr
* Indizieren der Adressen nach dem Hersteller. Es muß indiziert werden,
* obwohl das Datenbankfile bereits nach Artikeln sortiert ist, da FIND
* den Zugriff über ein Indexfile realisiert.
. INDEX ON hersteller TO laadrh
     6 Sätze indiziert
. SELECT lager
* Verbinde die Datenbankfiles über das Feld 'Hersteller' in dem
* Datenbankfile 'lageradr'.
. SET RELATION TO hersteller INTO lageradr

. LIST artikel, hersteller, lageradr->ort, lageradr->str
 Satz #  artikel          hersteller   lageradr->ort lageradr->str
      1  Flachschieber    C&C          Rheinsberg    M.-Henning-Str. 7
      2  Kurbelwelle I    Achse&Welle  Annaburg      Schulstraße 111
      3  Kurbelwelle II   Achse&Welle  Annaburg      Schulstraße 111
      4  Formschlauch     Gummi&Sohn   Cottbus       Straße der Jugend 17
      5  Kühlluftgehäuse  VEB Guß      Berlin        Kienbergstr. 55
      6  Luftleitblech    Blech&Co
      7  Kopfdichtungen   Dicht&Fest   Berlin        Suermondtstr. 43
      8  Krümmerdichtung  Dicht&Fest   Berlin        Suermondtstr. 43
      9  Fußdichtung      Dicht&Fest   Berlin        Suermondtstr. 43
```

Die letzte Ausgabe erweckt den Eindruck, als ob alle Daten in einem File stehen. Das war unser Ziel.

Warum wird die Adresse der Firma "Blech&Co" nicht gefunden? Das liegt daran, daß die Inhalte der Felder unterschiedlich sind. Einmal haben wir geschrieben "Blech&Co" und im Adreßfile "Blech&Co." - ein Unterschied, der bei der traditionellen Bearbeitung solcher Daten völlig unerheblich ist, hier aber eine große Wirkung hat. Die Erfahrung lehrt, daß es viele Irrtümer dieser Art gibt, die darauf beruhen, daß entweder keine Konventionen getroffen oder sie nicht eingehalten werden: AdW oder ADW, UdSSR oder UDSSR, "Goethestraße" oder "Goethestr." usw. Die Benutzung kürzerer Bezeichnungen würde hier helfen. Deshalb haben wir das Feld 'ID' aufgenommen. Nimmt man statt 'Hersteller' 'ID' in das Datenbankfile 'lager' auf, so spart man sogar noch Speicherplatz. Auf der anderen Seite muß man sich die gewählten Abkürzungen merken, und sie müssen eindeutig sein.

15. Weitere komplexe Operationen

Wir wissen bereits, daß komplexe Operationen für Datenbanksysteme charakteristisch sind. In den vorangegangenen Abschnitten haben wir mehrere davon kennengelernt. Hier wollen wir uns mit drei Kommandos beschäftigen, die für die Wartung einer Datenbank nützlich sind.

15.1. UPDATE

Der Sinn des UPDATE-Kommandos besteht darin, eine Korrektur der Daten eines Datenbankfiles vorzunehmen, wobei die Korrekturdaten selbst aus einem anderen Datenbankfile kommen. Es ist das einzige Kommando, welches dies gestattet. Für die Korrektur einzelner Felder kann das Kommando REPLACE verwendet werden.

Die Syntax von UPDATE ist

```
UPDATE [RANDOM] ON <schlüssel> FROM <alias>
     REPLACE <feld> WITH <ausdr> [,<feld2> WITH <ausdr2>...]
```

Betrachten wir dazu das folgende Beispiel. In unserem Materiallager ist eine Lieferung angekommen. Die eingegangenen Teile wurden in einem Datenbankfile mit dem Namen 'eingang.dbf' erfaßt (oder das File wurde, wie ein Lieferschein, mit übergeben). Wir möchten unsere Datei für die Lagerhaltung 'lager.dbf' dahingehend aktualisieren, daß wir zu den bereits vorhandenen Teilen die Anzahl der neuen Teile addieren.

Der Idealzustand wäre erreicht, wenn die Strukturen beider Datenbankfiles gleich wären. Das ist in der Praxis trotz Absprachen und Festlegungen nur schwer erreichbar und für dBASE auch nicht erforderlich. Wichtig ist, daß beide Datenbankfiles je ein Feld mit dem gleichen Namen und vom gleichen Typ haben. Anhand des Inhalts dieses Feldes wird bestimmt, welcher Satz mit welchem Wert korrigiert wird. Eine eindeutige Bezeichnung in den Feldern ist deshalb wichtig. Katalognummern, Schlüsselnummern, Kundennummern und Personenkennzahlen sind Beispiele für mögliche eindeutige Bezeichnungen aus verschiedenen Bereichen. Wir wählen, der besseren Lesbarkeit wegen, das Feld 'Artikel' für diesen Zweck.

Die Anzahl wird hier im Feld mit dem Namen 'Menge' eingetragen. 'Menge' hat den gleichen Typ wie 'Anzahl', ist aber nur drei Zeichen breit ('Anzahl' vier Zeichen). Eine Formatanpassung wird durchgeführt. Auch eine Typkonvertierung ist unter Anwendung der Konvertierungsfunktionen möglich.

'Preis' stimmt in allen Angaben mit dem Feld 'Preis' aus dem Datenbankfile 'lager.dbf' überein, wird aber überhaupt nicht verarbeitet. Unter 'Herst' verbirgt sich der Hersteller. Auch er ist in diesem Fall ohne Bedeutung.

Das Datenbankfile für die eingegangenen Teile, im folgenden Korrekturfile genannt, hat die folgende Struktur:

```
. USE eingang
. LIST STRUCTURE
Datenbankstruktur       - C:eingang.dbf
Anzahl der Datensätze   -        5
Letztes Änderungsdatum - 23.12.87
Feld   Feldname     Typ         Länge    Dez
    1  ARTIKEL      Zeichen       15
    2  MENGE        Numerisch      3
    3  PREIS        Numerisch      8      2
    4  HERST        Zeichen       12
** Gesamt **                      39
```

Den Inhalt des Korrekturfiles besichtigen wir mit dem Kommando LIST:

```
. LIST
 Satz #  ARTIKEL          MENGE    PREIS HERST
      1  Kurbelwelle II       2   325.00 Achse&Welle
      2  Formschlauch        20     1.55 Gummi&Sohn
      3  Kopfdichtungen     100     0.15 Dicht&Fest
      4  Krümmerdichtung    100     0.00 Dicht&Fest
      5  Fußdichtung         50     0.15
```

Jedes der beiden Datenbankfiles wird in einem Arbeitsbereich in Benutzung genommen. Das zu korrigierende Datenbankfile, im folgenden Zielfile genannt, muß immer geordnet (sortiert oder indiziert) nach dem verbindenden Feld in Benutzung genommen werden. Das Zielfile muß im aktiven Bereich in Benutzung sein, das Korrekturfile in einem anderen Arbeitsbereich. Man erreicht das durch die folgende Kommandofolge: .

```
. USE lager
  INDEX ON artikel TO laartikel
      9 Sätze indiziert
. SELECT 2
  INDEX ON artikel TO eiartikel
      5 Sätze indiziert
. SELECT lager
```

Beschreiben wir zuerst die Wirkung ohne Verwendung des Schlüsselwortes RANDOM. Wird RANDOM nicht verwendet, so muß auch das Korrekturfile geordnet in Benutzung genommen werden. UPDATE-Kommando und Ausführung sehen wie folgt aus:

```
. UPDATE ON artikel FROM eingang REPLACE anzahl WITH anzahl + eingang->menge
      5 Sätze aktualisiert
```

Wir besichtigen das korrigierte Datenbankfile durch das Kommando LIST:

```
. LIST
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER   BEM
      1  Flachschieber        112     2.40      15 29.02.88 C&C         MEMO
      4  Formschlauch          61     1.55       4 28.07.88 Gummi&Sohn  MEMO
      9  Fußdichtung          172     0.15      30 07.06.88 Dicht&Fest  MEMO
      7  Kopfdichtungen       216     0.15      30 17.08.88 Dicht&Fest  MEMO
      8  Krümmerdichtung      188     0.14      30 18.09.88 Dicht&Fest  MEMO
      5  Kühlluftgehäuse        8    16.60       2 23.11.88 VEB Guß     MEMO
      2  Kurbelwelle I         16   360.00       3 12.05.88 Achse&Welle MEMO
      3  Kurbelwelle II        14   325.00       3 12.05.88 Achse&Welle MEMO
      6  Luftleitblech         14     5.40       2 10.11.88 Blech&Co    MEMO
```

Das obige UPDATE-Kommando hat im einzelnen die folgende Wirkung. Der Dateizeiger wird in jedem Arbeitsbereich auf den ersten Datensatz positioniert. Sind die Inhalte der aktuellen Verbindungsfelder, hier 'Artikel', gleich, so wird die Korrektur ausgeführt. Danach wird der Dateizeiger im Korrekturfile auf den nächsten Datensatz gestellt (SKIP). Die Inhalte beider Verbindungsfelder werden verglichen. Bei Gleichheit wird die Korrektur ausgeführt usf. Sind die Inhalte der Verbindungsfelder nicht gleich, so wird der Dateizeiger in dem Datenbankfile um einen Satz vorwärts gestellt (SKIP), das den kleineren Vergleichswert enthielt. Der Vergleich wird durchgeführt usw. Die Operation wird beendet, wenn bei einem der beiden Datenbankfiles das Fileende erreicht wird.

Daraus ergibt sich, daß Korrektursätze, solche aus 'eingang', ohne Wirkung bleiben, wenn sie sich auf Artikel beziehen, die im ersten Datenbankfile nicht vorkommen. Sie werden einfach übergangen. Das hat den Vorteil, daß das gleiche Korrekturfile mehrfach zur Korrektur verschiedener Datenbankfiles benutzt werden kann. Haben wir z.B. verschiedene Datenbankfiles für die Baugruppen Motor, Getriebe, Karosserie, Elektrik usw. und sind alle Eingänge in einem Korrekturfile gespeichert, so kann UPDATE für diese Datenbankfiles nacheinander durchgeführt werden.

Beziehen sich mehrere Korrektursätze auf den gleichen Artikel, so werden sie alle berücksichtigt.

Ist ein Artikel mehrfach im Zielfile enthalten, so wird nur der erste Datensatz korrigiert. Das gilt auch für den Fall, daß mehrere Korrektursätze für diesen Artikel im Korrekturfile enthalten sind.

Sind die Datenbankfiles nicht geordnet, so wirkt UPDATE genau wie oben beschrieben. Inhaltlich geschieht aber nicht das Gewünschte. Bei größeren Datenbankfiles sind auch diese Fehler nur schwer auffindbar, da scheinbar alles richtig verläuft.

Wird RANDOM verwendet, so muß das Korrekturfile nicht geordnet in Benutzung sein. Das Zielfile muß indiziert (nicht sortiert!) nach dem Vergleichsfeld in Benutzung sein. UPDATE positioniert dann den Dateizeiger auf den ersten Datensatz des Korrekturfiles. Im Zielfile wird über ein intern durchgeführtes FIND-Kommando der Satz gesucht, der den gleichen Schlüssel wie der Satz im Korrekturfile hat. Die Korrektur wird durchgeführt. Im Korrekturfile wird zum nächsten Datensatz übergegangen (SKIP). Es wird wieder ein FIND-Kommando ausgeführt usw. Das Kommando wird beendet, wenn das Ende des Korrekturfiles erreicht

ist. Erfolglose FIND-Kommandos stören nicht.

Bezüglich doppelter Sätze gelten die gleichen Bemerkungen, wie sie für das UPDATE-Kommando ohne das Schlüsselwort RANDOM angegeben wurden.

Ist das Zielfile nicht indiziert in Benutzung, so erfolgt eine Fehlermitteilung. Daraus folgt, daß die Verwendung von RANDOM wesentlich sicherer ist.

Wir mußten feststellen, daß bei gleichen Artikeln im Zielfile nur der erste Datensatz korrigiert wurde. In diesem Fall ist zu prüfen, ob die Korrektur die beabsichtigte Wirkung hat. Ob Sätze in einem Datenbankfile mehrfach vorkommen, läßt sich durch die Kommandofolge

```
SET UNIQUE ON
INDEX ON artikel TO dummy
```

feststellen. Sind alle Artikel verschieden, so werden genausoviel Sätze indiziert, wie in dem Datenbankfile enthalten sind (vgl. SET UNIQUE, Abschn. 17.1). Sollen gleiche Artikel zusammengefaßt werden, so kann dafür das Kommando TOTAL (Abschn. 15.3) verwendet werden.

Scheinbar gleiche Vergleichsfelder können z.B. dadurch entstehen, daß 'Artikel' in 'eingang' schmaler ist als 'Artikel' in 'lager'. Hat es z.B. nur eine Breite von acht Zeichen, so sind die beiden Sorten Kurbelwellen nicht mehr unterscheidbar. In diesem Fall scheint die Reaktion des UPDATE-Kommandos die gleiche zu sein. Fünf Sätze werden korrigiert. Diesmal wird allerdings die Anzahl für Kurbelwelle I erhöht.

Das alles lehrt uns, daß UPDATE zwar sehr schön ist, wenn alle Daten korrekt sind, für kompliziertere Fälle aber nicht genügend Sicherheit bietet. Bei größeren Datenbanken sollte man immer einen sicheren Weg wählen. Eine Datenbank durchzusehen, um den Erfolg eines UPDATE-Kommandos zu kontrollieren, ist unzumutbar. Die Lösung der Aufgabe über ein Kommandofile ist deshalb zu bevorzugen.

Wird ein neues Teil geliefert, das bisher noch nicht in der Datenbank 'lager' vorhanden ist, so wird es durch UPDATE auch nicht übernommen. Nach erfolgtem UPDATE ist schwer zu überblicken, welche Sätze des Korrekturfiles berücksichtigt wurden. Die nicht berücksichtigten Sätze könnten neue Ersatzteile sein, sie können aber auch auf Grund fehlerhafter Vergleichsfelder nicht verarbeitet worden sein. Man möchte jedenfalls sehen können, welche Sätze des Korrekturfiles noch nicht verarbeitet wurden. Dazu bieten wir das folgende Kommandofile 'updmark.prg' (update and mark, korrigiere und markiere) an. Es führt eine Korrektur durch und markiert die verarbeiteten Sätze im Korrekturfile als gestrichen. Gibt man nach dem Aufruf von 'updmark' SET DELETED ON ein, so liefert ein nachfolgendes LIST alle nicht berücksichtigten Sätze. Sind das tatsächlich neue Ersatzteile, die in unser Datenbankfile 'lager.dbf' aufgenommen werden sollen, so erreicht man das durch

```
SET DELETED ON
APPEND FROM eingang
```

```
* Kommandofile 'updmark' zum Korrigieren und Markieren
* der Sätze, deren Werte bei UPDATE berücksichtigt werden.
* Vor.: Zu korrigierendes Datenbankfile unter alias
*          'lager' in Benutzung
*       Korrekturfile unter alias 'eingang' in Benutzung
*       Beide Files nach 'artikel' geordnet
SET EXACT ON
SET TALK OFF
SELECT eingang
GO TOP
* eof-Flag (end of file) für das Korrekturfile
m_eof_eing = EOF()
SELECT lager
GO TOP
* Solange in keinem der Files das Ende erreicht ist
DO WHILE .NOT. (EOF() .OR. m_eof_eing)
  IF artikel = eingang->artikel
    * Gleiche Artikel gefunden, also Korrektur
    REPLACE anzahl WITH anzahl + eingang->menge
    SELECT eingang
    * Wir markieren den verarbeiteten Satz als gestrichen
    DELETE
    * ... und gehen zum nächsten Satz im Korrekturfile
    SKIP
    m_eof_eing = EOF()
    SELECT lager
  ELSE
    IF artikel < eingang->artikel
      * Im Zielfile vorwärts schreiten
      SKIP
    ELSE
      * Im Korrekturfile vorwärts schreiten
      SELECT eingang
      SKIP
      m_eof_eing = EOF()
      SELECT lager
    ENDIF
  ENDIF
ENDDO
SET TALK ON
SET EXACT OFF
RETURN
* Ende des Kommandofiles 'updmark.prg'
```

15.2. JOIN

JOIN vereinigt zwei Datenbankfiles und erzeugt dabei ein drittes, neues Datenbankfile. Das neue Datenbankfile setzt sich in seiner Struktur und in seinem Inhalt aus den beiden anderen Datenbankfiles zusammen.

JOIN WITH <alias> **TO** <filename> **FOR** <bedingung> [**FIELDS** <feldliste>]

Im aktiven Arbeitsbereich ist das erste Datenbankfile in Benutzung. Das zweite Datenbankfile ist in einem anderen Arbeitsbereich in Benutzung. Es wird durch den Aliasnamen angegeben. Das zu erzeugende Datenbankfile wird sofort in ein File <filename> geschrieben. Es muß vorher nicht existieren und belegt keinen Arbeitsbereich.
Die Feldliste bestimmt die Struktur des neuen, dritten Datenbankfiles. Ist keine Feldliste angegeben, so ergibt sich die Struktur des neuen Datenbankfiles aus den Feldern des ersten Datenbankfiles und den Feldern des zweiten Datenbankfiles.

Bei der Ausführung von JOIN werden beide Dateizeiger auf den ersten Datensatz jedes Datenbankfiles gestellt. Die FOR-Bedingung wird ausgewertet. Ist sie erfüllt, so wird ein neuer Datensatz an das Datenbankfile <filename> angefügt, wobei Struktur und Inhalt sich wie oben beschrieben ergeben. Der Dateizeiger im zweiten Datenbankfile wird einen Datensatz vorwärts gestellt. Die FOR-Bedingung wird geprüft usw.
Ist das Ende des zweiten Datenbankfiles erreicht, so wird der Dateizeiger des ersten Datenbankfile einen Satz vorwärts gestellt. Der Dateizeiger des zweiten Datenbankfiles wird auf den ersten Datensatz gestellt. Die FOR-Bedingung wird ausgewertet usw., bis das Ende des ersten und des zweiten Datenbankfiles erreicht sind.

Einige Bemerkungen zu diesem Kommando:

1. Besteht das erste Datenbankfile aus m, das zweite aus n Sätzen und trifft die FOR-Bedingung auf jeden Datensatz zu, so hat das erzeugte Datenbankfile m*n Sätze! Man sollte sehr sorgfältig prüfen, ob das wirklich erforderlich und gewünscht ist.

2. Einer unserer Grundsätze besteht darin, Informationen nicht zu doppeln. Es ist die erklärte Wirkung von JOIN, Informationen zu doppeln, indem gleiche Feldinhalte in mehrere Datensätze aufgenommen werden. Es sei denn, die FOR-Bedingung trifft nur für jeden Datensatz des ersten Datenbankfiles auf genau einen Datensatz des zweiten Datenbankfiles zu.

3. Trifft die FOR-Bedingung nur auf wenige Datensätze zu, so wird dennoch in m*n Sätzen ihre Gültigkeit überprüft. Das bedeutet viele überflüssige Operationen und viele Bewegungen durch die Files, also auch viele Zugriffe auf den externen Datenspeicher. JOIN ist deshalb zeitaufwendig!

4. Alle Informationen, die man über JOIN in ein Datenbankfile aufnimmt, lassen sich auch durch andere, einfachere und überschaubare Kommandos in Zusammenstellungen oder Berichten vereinen. Man beachte, daß die äußere Form von Zusammenstellungen und Berichten nichts mit der internen Struktur der Datenbankfiles zu tun hat. Bedingung ist, daß sich alle auszugebenden Informationen aus den in der Datenbank gespeicherten Fakten zusammensetzen

lassen! JOIN wird oft dort benutzt, wo dieser Unterschied nicht gesehen wird.

5. Findet sich ein Anwendungsfall für JOIN, so heißt das schließlich, daß die Struktur der Datenbank nicht gut genug konzipiert wurde (Abschn. 5). Wir haben mehrfach herausgearbeitet, daß die Separierung der Informationen in mehrere Datenbankfiles aus verschiedenen Gründen vorteilhaft ist. Diese Tendenz wurde auch in dBASE III erkannt, indem 10 Arbeitsbereiche zugelassen werden. JOIN unterstützt gerade die entgegengesetzte Richtung.

6. Als wichtiges Kriterium für die Güte einer Datenbank hatten wir herausgearbeitet: Eine Datenbank ist so gut, wie sie den wechselnden Anforderungen standhalten kann. Es können deshalb, selbst bei allersorgfältigster Vorbereitung, Wechsel in der Strategie erforderlich sein. Dann ist JOIN vorteilhaft einsetzbar; denn die Ausgangsdaten sind zuverlässig, die Operation wird nur einmal durchgeführt, und die Quellfiles können danach vernichtet werden, so daß keine Dopplung von Daten auftritt.

7. Viele Anwendungen von JOIN in dBASE II dienen der Vorbereitung von Standardberichten (REPORT). Man sollte in dBASE III unter Anwendung von SET RELATION TO versuchen, die Berichte ohne eine vorherige Vereinigung der Datenbankfiles zu erzeugen.

8. JOIN ist sehr komplex und in seiner inhaltlich richtigen Ausführung nur schwer kontrollierbar. Bei unsicheren Daten ist eine Lösung über ein Kommandofile zu bevorzugen.

15.3. TOTAL

TOTAL dient dem Verdichten von Datenbankfiles. Es können die Sätze eines Datenbankfiles zusammengefaßt werden, die in einem beliebig gewählten Feld gleiche Inhalte haben.

TOTAL [<gb>] **ON** <schlüssel> **TO** <filename>
[**FIELDS** <feldliste>] [**FOR/WHILE** <bedingung>]

Der Standard für den Gültigkeitsbereich ist ALL. Durch die Auswahlbedingung kann die Wirkung auf bestimmte Sätze weiter eingeschränkt werden.

<schlüssel> muß ein Feldname sein. Das Datenbankfile ist nach diesem Feld geordnet (indiziert oder sortiert) in Benutzung. Es wird ein neues Datenbankfile <filename> erzeugt, welches die gleiche Struktur hat wie das aktive Datenbankfile. Aus allen Datensätzen des aktiven Datenbankfiles, die gleiche Schlüsselfelder haben, entsteht ein Satz im neuen File. Für alle Felder der Typen String, Datum und numerisch werden die Inhalte des ersten Datensatzes übernommen. Ist keine Feldliste angegeben, so werden Summen über alle numerischen Felder gebildet, die den gleichen Schlüssel haben. Die Ergebnisse

werden in den neuen Datensatz des Zielfiles eingetragen. Ist eine Feldliste angegeben, so erfolgt die Summenbildung nur über die angegebenen Felder. Für alle anderen numerischen Felder wird ebenfalls der Wert des ersten Satzes übernommen.

Memo-Felder werden durch TOTAL nicht übernommen!

Betrachten wir die Wirkung an einem Beispiel. Wir wählen ein verkürztes Datenbankfile für die Lagerhaltung 'lagerk.dbf', in dem einige Artikel doppelt vorkommen. Ziel ist es, dieses File so zu verdichten, daß jeder Artikel nur einmal vorkommt, d.h., 'artikel' wird das Schlüsselfeld, so daß die Anzahl addiert wird, aber alle anderen Feldinhalte aus dem ersten Satz (sie sind in allen Sätzen gleich) übernommen werden.

```
. USE lagerk
* Das zu verdichtende Datenbankfile enthält
. LIST
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER    BEM
      1  Flachschieber       112     2.40      15 29.02.88  C&C           MEMO
      2  Formschlauch         41     1.55       4 28.07.88  Gummi&Sohn    MEMO
      3  Formschlauch        122     1.55      30 28.07.88  Gummi&Sohn    MEMO
      4  Kopfdichtungen      116     0.15      30 17.08.88  Dicht&Fest    MEMO
      5  Kopfdichtungen       10     0.15       5 18.09.88  Dicht&Fest    MEMO
      6  Kühlluftgehäuse       8    16.60       2 23.11.88  VEB Guß       MEMO
. TOTAL ON artikel TO lagerv FIELD anzahl
      6 Sätze gesamt
      4 Sätze erzeugt
. USE lagerv
* Wir sehen uns das verdichtete Datenbankfile an
. LIST
 Satz #  ARTIKEL          ANZAHL    PREIS MINIMUM LIEFERUNG HERSTELLER
      1  Flachschieber       112     2.40      15 29.02.88  C&C
      2  Formschlauch        163     1.55       4 28.07.88  Gummi&Sohn
      3  Kopfdichtungen      126     0.15      30 17.08.88  Dicht&Fest
      4  Kühlluftgehäuse       8    16.60       2 23.11.88  VEB Guß
```

Wir sehen, daß das Memo-Feld fehlt. Außerdem stimmten die Sätze nicht in allen übrigen Feldern überein. Man sieht, daß nur die Daten der ersten Sätze übernommen werden. Es erfolgt keine Warnung. Werden bei der Summenbildung Werte berechnet, die im Zielfeld aufgrund der Spaltenbreite nicht mehr gespeichert werden können, so erfolgt eine Fehlerausschrift: "Numerischer Überlauf (Daten sind verloren)".

16. Briefadressen

Als Erweiterung gegenüber dBASE II bietet dBASE III die Möglichkeit, Briefadressen nach einem Standardformat zu erzeugen. Für die Definition solcher Adressen existieren die Kommandos CREATE LABEL und MODIFY LABEL.

CREATE LABEL <filename>

MODIFY LABEL <filename>

Beide Kommandos führen zu denselben Menüs. dBASE geht in den Direktmodus der Bildschirmbedienung über und bietet nacheinander zwei Menüs an. Zuerst wird die Form der Briefadresse definiert. Bild 16.1 gibt die möglichen Angaben wieder. Durch die eingeschalteten Hilfsmenüs sind gleichzeitig die Steuerfunktionen sichtbar.

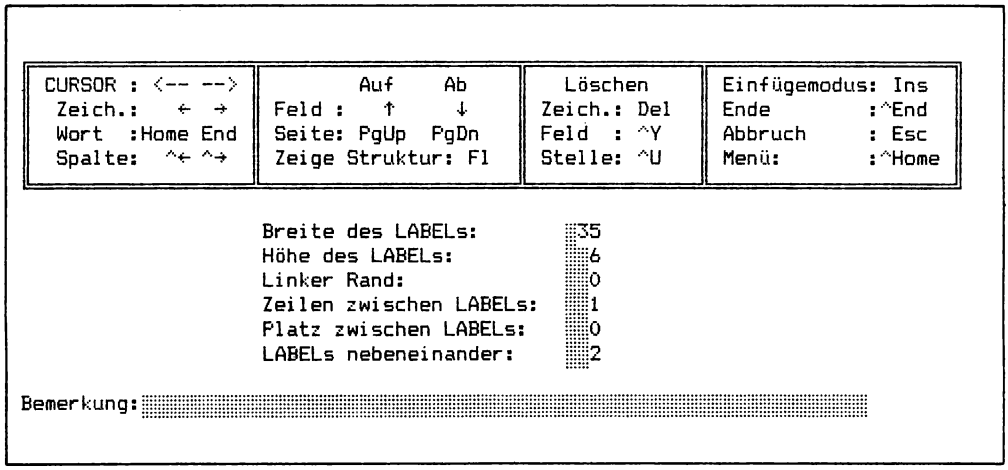


Bild 16.1. CREATE und MODIFY LABEL, Definition der Struktur

Wird hinter "Bemerkung:" ein Text eingegeben, so dient dieser lediglich als Kommentar. Er wird nicht gedruckt, ist aber bei jedem MODIFY LABEL wieder sichtbar.

Bis auf die Höhe, d.h. die Anzahl der Zeilen der Adresse, sind die im Bild gezeigten Werte die Standardvorgaben. Wir stellen uns das Ziel, Briefadressen für die Hersteller der Ersatzteile zu drucken. Form und Inhalt der Adressen werden in einem .lbl-File 'lalabel.lbl' abgelegt. Dazu benötigen wir eigentlich nur das Datenbankfile 'lageradr.dbf'. Wollen wir jedoch das Drucken einer Adresse vom Artikel abhängig machen, so ist es angebracht, beide Datenbanken zu verbinden (vgl. Abschn. 14). Das erreichen wir durch folgende Kommandos:

```
. USE lager
. SELECT 2
. USE lageradr INDEX laadrh
. SELECT lager
. SET RELATION TO hersteller INTO lageradr
```

Unter diesen Voraussetzungen erscheint als zweites Menü das Bild 16.2. In unserer Darstellung sind bereits die Ausdrücke eingegeben, die die Adresse bilden. Jeder Ausdruck muß vom Typ String sein.

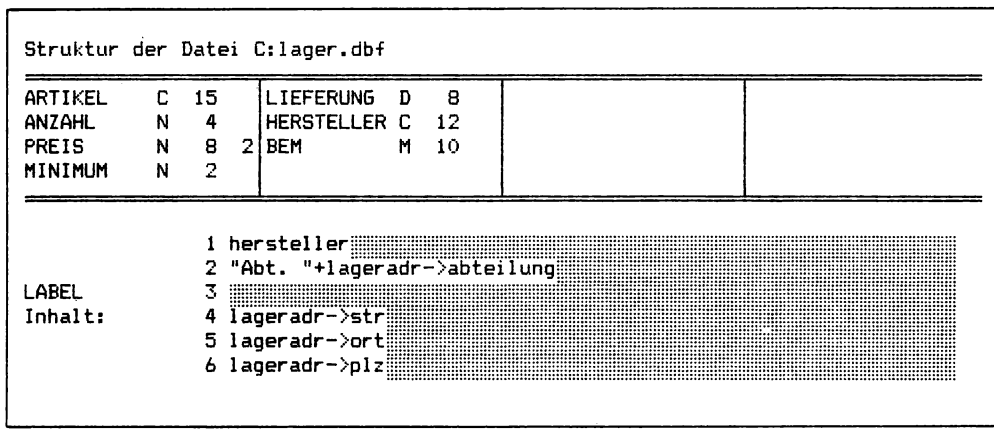


Bild 16.2. CREATE und MODIFY LABEL, Definition des Inhalts

Wir sehen, daß wir Felder des Datenbankfiles (1. Zeile) und Ausdrücke angeben können (2. Zeile). Auch Felder eines Datenbankfiles aus einem anderen Arbeitsbereich können benutzt werden (4. bis 6. Zeile). Drucken wir die Adresse des Herstellers, der Krümmerdichtungen herstellt:

```
. LABEL FORM lalabel FOR artikel = "Krümmer"
Dicht&Fest
Abt. Absatz
Suermondtstr. 43
Berlin
1092
```

In der dritten Zeile hatten wir eine Leerzeile gelassen, um eine Form der Adresse zu erzeugen, wie sie im Duden vorgeschlagen wird. Die Leerzeile (3. Zeile) wird beim Drucken gestrichen. Selbst wenn wir Leerzeichen einfügen, werden diese ignoriert.

Sehen wir uns die Adresse an, die für die Firma C&C ausgegeben wird:

```
. LABEL FORM lalabel FOR hersteller = "C&C"
C&C
Abt.
M.-Henning-Str. 7
Rheinsberg
1955
```

Hier stört uns das Rudiment "Abt." der zweiten Zeile. Es gibt keine Möglichkeit, Zeilen in Abhängigkeit von bestimmten Bedingungen zu unterdrücken, wie es z.B. in MailMerge, einer Komponente von WordStar, beim Einfügen variabler Texte möglich ist.

Versuchen wir, die Postleitzahl gesperrt zu drucken. Wir müssen dazu mit der Funktion SUBSTR jedes Zeichen als Teilzeichenkette herausziehen und jeweils ein Leerzeichen einfügen. Dieser Ausdruck wird sehr lang. In unserem Beispiel kommen wir bis zur zweiten Ziffer. Dann ist die Zeile voll. Eine Fortsetzung ist nicht möglich. Bleibt noch die Möglichkeit, die benutzten Bezeichner zu verkürzen. An der Länge der Funktionsbezeichnung SUBSTR(,,) können wir nichts ändern. Wir können aber einen anderen, kürzeren Aliasnamen für das Datenbankfile 'lageradr.dbf' einführen, indem wir dieses File wie folgt in Benutzung nehmen:

```
USE lageradr ALIAS l.
```

Eine weitere Variante besteht darin, Zwischenergebnisse Speichervariablen zuzuweisen, etwa

```
p = lageradr->plz
```

Dann ergibt sich der Ausdruck für die gesperrt zu druckende Postleitzahl als

```
SUBSTR(p,1,1)+' '+SUBSTR(p,2,1)+' '+SUBSTR(p,3,1)+' '+SUBSTR(p,4,1)
```

Das sind immer noch mehr als 64 Zeichen. Außerdem kann so immer nur eine Adresse gedruckt werden, da die Wertzuweisung zur Speicherplatzvariablen nicht in die Definition des .lbl-Files eingeht. Zum Drucken mehrerer Adressen ist also ein Kommandofile erforderlich.

Insgesamt muß man sagen, daß LABEL für einfache Briefadressen ausreichend ist. Möchte man jedoch eine spezielle Form haben, so hilft nur ein Kommandofile.

17. dBASE in seiner Umgebung

Im Abschnitt 2.6 haben wir die Anforderungen besprochen, die dBASE III an die Hard- und Software stellt. Trotzdem sind damit noch viele Unterschiede zwischen den Rechnern möglich. Sie haben unterschiedlich große interne Speicher; sie haben verschiedene externe Speichermedien (Festplatte, Diskette), und sie benutzen, selbst bei gleichen Betriebssystemversionen, unterschiedliche Voreinstellungen. Solche Voreinstellungen im Betriebssystem MS-DOS sind z.B. der Suchpfad, auf dem nach Files gesucht wird, unterschiedliche Pufferanzahlen für die Ein- und Ausgabe usw. Diese Größen sind in jedem konkreten System standardmäßig festgelegt. Für den Anfang braucht man sich deshalb um diese Einstellungen nicht zu kümmern. Möchte man seinen Computer möglichst effektiv nutzen, so kann sich im Laufe der Zeit, z.B. mit dem Wachsen der Datenbank und der Anzahl der Kommandofiles, eine Veränderung dieser Voreinstellung erforderlich machen.

Hinzu kommt, daß dBASE selbst für alle Optionen Voreinstellungen hat. Auf diese Voreinstellungen von dBASE haben wir uns in den vergangenen Abschnitten immer bezogen. Diese Voreinstellungen können während der Arbeit mit dBASE mehrfach geändert werden. Ferner ist es möglich, Voreinstellungen beim Laden von dBASE zu verändern. Welche Optionen zu einem Zeitpunkt gültig sind, kann man im wesentlichen mit dem Kommando

```
LIST STATUS [TO PRINT]
```

sichtbar machen.

Eine günstige Voreinstellung von dBASE kann die Eingabe vieler Kommandos ersparen und die Arbeit für den Nutzer angenehmer gestalten.

Wie diese Voreinstellungen auf die Bedürfnisse zugeschnitten werden können und wie die Zusammenhänge mit dem Betriebssystem sind, behandeln wir in diesem Abschnitt.

17.1. SET-Kommandos

SET-Kommandos dienen der Auswahl bestimmter Optionen von dBASE. Sie legen für einige Kommandos Parameter fest, schalten auf bestimmte Arbeitsweisen um, schalten Dienstleistungen ein und aus, oder sie legen die Arbeitsweise von dBASE insgesamt fest. Mehrere Formen eines SET-Kommandos sind mitunter zur Handhabung einer Option erforderlich. So wird z.B. durch

```
SET ALTERNATE TO <file>
```

festgelegt, daß ein Textfile mit dem Namen <file> angelegt wird,

welches wir als Alternate-File bezeichnen. Es wird ein leeres File ins Fileverzeichnis aufgenommen. Erst durch

SET ALTERNATE ON

erreicht man, daß alle Ausgaben, die danach sequentiell auf dem Bildschirm erscheinen, im Alternate-File abgelegt werden.

dBASE III kennt 33 SET-Kommandos, die wir im folgenden in alphabetischer Reihenfolge besprechen. Ihre Wirkung wird durch Beispiele erläutert, soweit das erforderlich ist. Durch Verweise auf andere Kommandos werden weitere Zusammenhänge aufgezeigt. Die Standardeinstellungen sind bei Schaltern (ON/OFF) durch Unterstreichung hervorgehoben. So bedeutet ON/OFF, daß OFF der Standard ist.

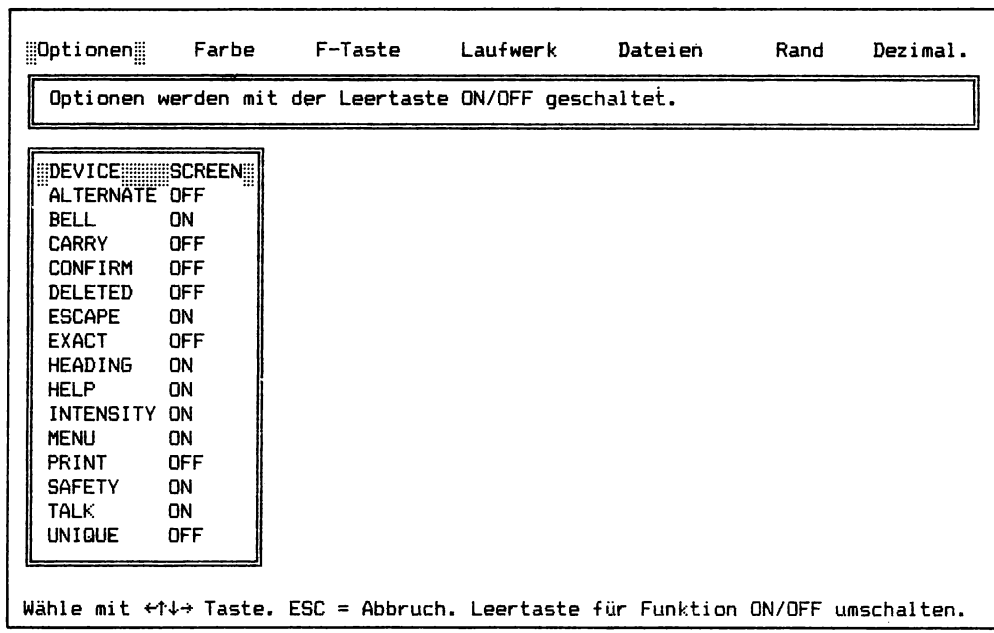


Bild 17.1. Setzen von Optionen im Direktmodus

SET

Syntax: **SET**

SET dient zur Einstellung von Optionen im Direktmodus. Auf dem Bildschirm erscheint Bild 17.1. Die erste Bildschirmzeile zeigt die Komplexe an, aus denen eine Auswahl gestroffen werden kann. In diesem Bild ist das Setzen von Optionen ausgewählt. Das Kästchen am linken Rand des Bildschirms wird aufgebaut, wenn die **Enter**-Taste betätigt wird. Es wird "heruntergezogen". Derartige Menüs heißen Pull-down-

Menüs. Positionieren wir den Kursor der oberen Zeile auf "Farbe", so verschwindet das Menükästchen für die Optionen. Man sagt: Es wird geschlossen. Drücken wir die **Enter**-Taste, so erscheint ein neues Pull-down-Menü mit den Auswahlmöglichkeiten für Farben. Bild 17.2 zeigt diese Situation.

Mit den senkrechten Pfeilen bewegt man sich in diesen Pull-down-Menüs und wählt die gewünschte Einstellung durch Drücken der Leertaste aus. "F-Taste" gestattet die Belegung der Funktionstasten.

Alle Auswahlmöglichkeiten des SET-Kommandos im Direktmodus kann man auch im Kommandomodus als ein SET-Kommando eingeben. Wir verzichten deshalb an dieser Stelle auf eine eingehende Beschreibung aller Möglichkeiten.

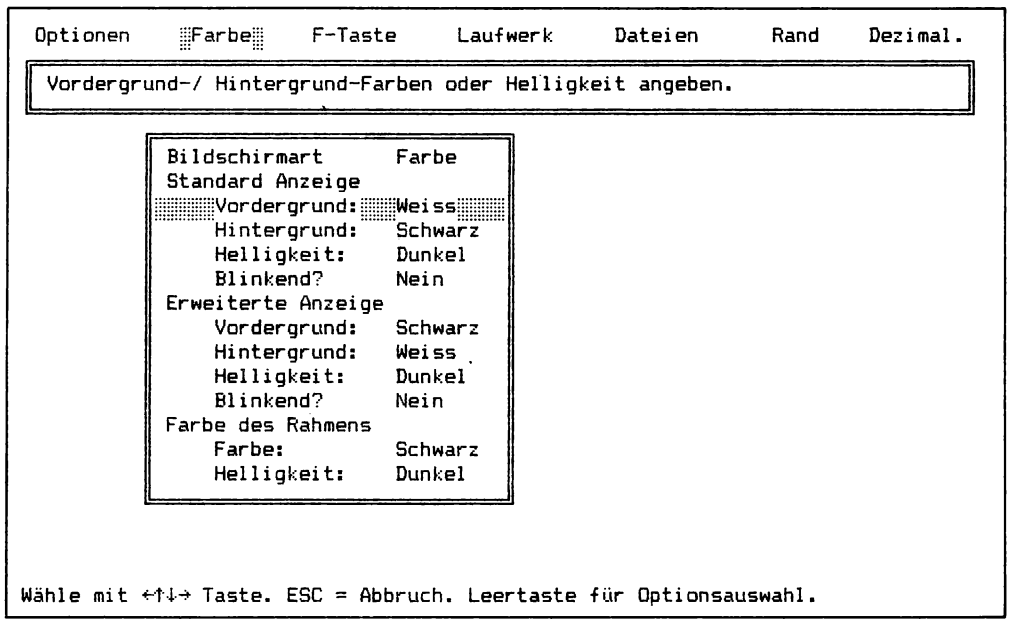


Bild 17.2. Farbauswahl im Direktmodus

SET ALTERNATE

Syntax: SET ALTERNATE TO <file>
SET ALTERNATE ON/OFF

SET ALTERNATE TO erzeugt ein Textfile mit dem Namen <file>. Dieses File nennen wir Alternate-File. Es ist nach SET ALTERNATE leer und nicht eröffnet. Existiert bereits ein File mit dem gleichen Namen, so wird es ohne Anfrage überschrieben; SET SAFETY hat hierauf keinen Einfluß. Damit besteht keine Möglichkeit, Alternate-Files aus vorigen dBASE-Sitzungen fortzusetzen. Mehrere Alternate-Files können aber z.B. mit einem Texteditor zusammengesetzt werden.

SET ALTERNATE ON eröffnet das Alternate-File. Alle sequentiellen Ausgaben, die danach auf dem Bildschirm erscheinen, werden auch im

Alternate-File gespeichert. Die Ausgabe erfolgt so lange, bis eins der folgenden Kommandos eingegeben wird: SET ALTERNATE OFF, CLOSE ALTERNATE oder QUIT.

SET ALTERNATE OFF beendet die Ausgabe in das Alternate-File. Sie kann später, aber ohne zwischendurch dBASE zu verlassen, durch SET ALTERNATE ON fortgesetzt werden.

Wird ein Alternate-File mit CLOSE ALTERNATE abgeschlossen, so kann es danach nicht mehr fortgesetzt werden. SET ALTERNATE ON wird gesetzt, führt somit nicht zu einer Fehlerausschrift, kann aber nicht wirksam werden, da kein Alternate-File mehr bekannt ist.

Ist ALTERNATE ON wirksam, so bewirkt der Aufruf eines Kommandos, welches den Bildschirm im Direktmodus bedient, daß das Kommando selbst noch in das Alternate-File aufgenommen wird. Danach führt dBASE selbständig ein SET ALTERNATE OFF aus. Bei der Rückkehr aus dem Direktmodus fügt dBASE, ebenfalls selbständig, ein SET ALTERNATE ON ein.

Verläßt man dBASE durch QUIT, so wird das Alternate-File ordnungsgemäß abgeschlossen.

Siehe auch: CLOSE ALTERNATE.

SET BELL

Syntax: SET BELL ON/OFF

SET BELL ON bewirkt, daß ein akustisches Signal ausgegeben wird, wenn einer von zwei Fällen eintritt.

1. Bei der Eingabe von Daten für Felder im Direktmodus wird ein Feld durch die Eingabe von Zeichen verlassen, d.h., es wurde das letzte Zeichen des Feldes eingegeben. Ist dann CONFIRM OFF eingestellt, so springt der Kursor auf das erste Zeichen des nächsten Feldes.

2. Bei der Eingabe von Daten für Felder im Direktmodus wird ein Zeichen eingegeben, welches nicht dem Format entspricht, z.B. ein Buchstabe für ein numerisches Feld.

Siehe auch: SET CONFIRM.

SET CARRY

Syntax: SET CARRY ON/OFF

CARRY ON bewirkt, daß bei den Kommandos APPEND und INSERT alle Daten des vorherigen Satzes in den neuen Satz übernommen werden. Treten ähnliche Daten in aufeinanderfolgenden Sätzen auf, so können durch CARRY ON Eingaben gespart werden. Die veränderten Daten werden, wie normal bei APPEND und INSERT, im Direktmodus überschrieben. SET CARRY wirkt auch bei der Verwendung privater Bildschirmmasken (Abschn. 13.2).

SET COLOR

Syntax: SET COLOR TO [<standard>] [,<erweitert>] [,<rand>]

SET COLOR TO gestattet die Auswahl von Farben und Attributen des Bildschirms. <standard> und <erweitert> sind jeweils Paare von Farben. Acht Farben stehen zur Auswahl. Sie werden durch Ziffern oder Buchstabenkombinationen gemäß Tafel 17.1 bezeichnet. Jedem Farbpaar kann ein Pluszeichen '+' und ein '*' vorangestellt sein. '+' bewirkt eine intensive Darstellung auf dem Bildschirm, und '*' läßt die nachfolgenden Ausgaben blinken. Die Farbpaare werden in der Form

<vordergrundfarbe> / <hintergrundfarbe>

angegeben.

<rand> ist eine einzelne Farbangabe. Sie bestimmt die Farbe des Bildschirmrandes, beeinflußt aber auch die Ausgabe innerhalb der umrandeten Bildschirmfläche.

Alle Angaben führen zur Ausgabe von Steuerinformationen. Sie sind u.U. auch nach dem Beenden von dBASE noch wirksam. Durch

SET COLOR TO

werden alle Farben auf den Standard zurückgesetzt.

<standard> ist die Auswahl für alle Ausgaben auf dem Bildschirm. Weiß auf Schwarz, d.h. 7/0, sind die Standardannahmen dafür.

<erweitert> bestimmt die Darstellung der Eingabebereiche für Variable in Bildschirmmasken. Hiervon sind sowohl die Standardmasken als auch die privaten Bildschirmmasken betroffen. Die Standardannahmen sind schwarze Zeichen auf weißem Grund (inverse Darstellung), d.h. 0/7. Mit SET INTENSITY OFF kann die unterschiedliche Darstellung von <standard> und <erweitert> aufgehoben und einheitlich auf die für <standard> eingegebenen Werte gesetzt werden.

Der Rand <rand> ist standardmäßig schwarz, d.h. 0. Eine Veränderung wirkt sofort auf den ganzen Rand und bleibt auch nach dem Verlassen von dBASE wirksam.

Die gesamte Standardeinstellung ist gegeben durch

SET COLOR TO 7/0,0/7,0

Beispiel: Wir verändern unser Beispiel aus dem Abschnitt 13.2. Dort hatten wir eine private Bildschirmmaske für die Eingabe definiert. Sie bewirkt u.a., daß bei der Datenerfassung im oberen Drittel des Bildschirms ein doppelt umrandeter Text erscheint. Wir wollen die Umrandung rot, intensiv und blinkend erscheinen lassen, während der Text im Innern des Kästchens intensiv weiß erscheint und alle anderen Texte normal. Die Lösung kann dann nicht mehr in einem Formatfile bestehen, da Formatfiles nur @-Kommandos und Kommentare enthalten dürfen. Alle Kommandos des Formatfiles nehmen wir in ein Kommandofile. Die Lösung ist durch die folgende Kommandofolge gegeben

```
...
* Es muß nur das neue Attribut für <standard> und
* dort nur für den Vordergrund angegeben werden.
SET COLOR TO *+4
@ 4, 20 SAY "╔═══════════════════════════════════════╗"
@ 5, 20 SAY "║                                       ║"
@ 6, 20 SAY "║                                       ║"
@ 7, 20 SAY "╚═══════════════════════════════════════╝"
* Rücksetzen blinken
SET COLOR TO +4
@ 5, 22 SAY "Erfassungsbeleg für neue Ersatzteile,"
@ 6, 22 SAY "die bisher noch nicht erfaßt sind    "
* Rücksetzen aller Attribute
SET COLOR TO
* Die folgenden Ausgaben erscheinen in der normalen Darstellung.
```

Durch derartige Hervorhebungen kann der Nutzer z.B. auf Fehler oder besonders wichtige Dinge besser aufmerksam gemacht werden als durch einfache Texte. Das entspricht dem Standard der 16-Bit-Technik.

Tafel 17.1. Farbtafel für SET COLOR

Farbe	Buchstabenkode	Ziffernkode	Farbe	Buchstabenkode	Ziffernkode
Schwarz	blank	0	Rot	R	4
Blau	B	1	Magenta	BR	5
Grün	G	2	Gelb	GR	6
Cyan	BG	3	Weiß	W	7

SET CONFIRM

Syntax: SET CONFIRM ON/OFF

SET CONFIRM ON bewirkt, daß bei der Eingabe im Direktmodus (APPEND, EDIT, CHANGE usw.) der Kursor nicht zum Anfang des nächsten Feldes springt, wenn das Ende eines Feldes erreicht ist. Er verharrt auf dem letzten Zeichen dieses Feldes. Erst durch Eingabe von **Enter** geht er in das nächste Feld. Je nach Gewohnheit kann dadurch in Kombination mit SET BELL eine höhere Sicherheit bei der Datenerfassung ereicht werden.

SET CONSOLE

Syntax: SET CONSOLE ON/OFF

CONSOLE OFF unterdrückt alle Ausgaben auf dem Bildschirm. Eingaben

von der Tastatur sind weiter möglich. Es erscheint kein Echo auf dem Bildschirm. Die Ausgabe von Fehlermitteilungen auf dem Bildschirm wird durch CONSOLE OFF nicht unterdrückt. Dieses Kommando kann verwendet werden, um Paßwörter einzulesen, die auf dem Bildschirm nicht sichtbar sein sollen.

SET DEBUG

Syntax: SET DEBUG ON/OFF

DEBUG ist eins von vier SET-Kommandos, die zum Test von Kommandofiles implementiert wurden. Die anderen drei sind SET ECHO, SET TALK und SET STEP. DEBUG wirkt nur, wenn ECHO ON ist. Dann bewirkt DEBUG ON, daß die im Kommandofile stehenden Kommandos auf dem Drucker ausgegeben werden. Ist der Drucker nicht bereit, so wird ein Fehler auf dem Gerät PRN gemeldet. Reagiert man darauf mit Eingabe von A (für Abbruch, abort), so wird dBASE verlassen. Dieses Kommando ist deshalb mit äußerster Vorsicht anzuwenden!
Siehe auch: SET ECHO, SET STEP, SET TALK.

SET DECIMALS

Syntax: SET DECIMALS TO <ausdr>

<ausdr> liefert bei der Berechnung einen ganzzahligen Wert. SET DECIMALS bestimmt die Anzahl an Dezimalstellen, die für einen numerischen Wert ausgegeben werden.

SET DECIMALS wirkt nur auf Werte, die durch Division entstehen, und für Ergebnisse der Funktionen SQRT(), LOG() und EXP(). Alle anderen Operationen sind davon nicht beeinflußt.

Die Anzahl an Dezimalstellen im Ergebnis einer Addition oder Subtraktion ist die Anzahl an Dezimalstellen, die der Operand mit der größten Genauigkeit hat.

Das Ergebnis einer Multiplikation wird mit so vielen Dezimalstellen angegeben, wie die Summe der Dezimalstellen beider Multiplikanden angibt.

Es werden erst dann alle Werte mit genau der Stellenzahl ausgegeben, wenn SET FIXED ON gegeben wurde.

Standard ist SET DECIMALS TO 2.
Siehe auch: SET FIXED, Definition von Feldern, Ein- und Ausgabeformate (PICTURE).

SET DEFAULT

Syntax: SET DEFAULT TO <gerät>

<gerät> ist eine Gerätebezeichnung. SET DEFAULT bewirkt, daß das angegebene Gerät zum Bezugslaufwerk wird. Das hat zur Folge, daß dBASE alle Files (Datenbankfiles, Indexfiles, Kommandofiles, Formatfiles usw.) auf diesem Gerät sucht.

Jedem Filenamen kann außerdem eine Gerätebezeichnung vorangestellt werden.

Gibt man für <gerät> einen Pfadnamen mit oder ohne Gerätebezeichnung an, so wird dieser ignoriert. Es erfolgt keine Fehlermitteilung. LIST STATUS liefert u.a. den aktuellen Pfad.

SET DEFAULT beeinflußt nicht die Suche nach Überlagerungen von dBASE. Sie werden auf dem Gerät und in dem Verzeichnis gesucht, in dem das File 'dbase.exe' steht.

SET DEFAULT wirkt nur innerhalb von dBASE. Nach QUIT gelten die Festlegungen, die beim Aufruf von dBASE gültig waren.

Standard ist das Gerät, von dem aus dBASE aufgerufen wurde. Das ist nicht notwendig das Gerät, auf dem das File 'dbase.exe' steht.
Siehe auch: SET PATH.

SET DELETED

Syntax: SET DELETED ON/OFF

Gilt DELETED ON, so werden als gestrichen markierte Datensätze in die meisten Kommandos nicht mit einbezogen. Es gibt jedoch auch Kommandos oder Teile davon, die als gestrichen markierte Sätze selbst bei DELETED ON nicht ignorieren. Das gilt für 'RECORD n' und 'NEXT n' im Gültigkeitsbereich jedes Kommandos. Desgleichen wirkt INDEX und REINDEX immer auf alle Datensätze, unabhängig von ihrer Löschmarkierung. Dadurch wird beim Aufheben der Löschmarkierung kein erneutes Indizieren notwendig.

Beispiel: Listen wir mit LIST Datensätze auf dem Bildschirm aus, so sind als gestrichen markierte Sätze durch einen Stern '*' gekennzeichnet. Man kann durch einfache Betrachtung sofort sehen, ob gestrichene Sätze einbezogen werden oder nicht. Anders sieht es hingegen bei Kommandos wie SUM aus. Hier ist aus den gebildeten Summen nicht ohne weiteres zu ersehen, ob als gestrichen markierte Sätze einbezogen wurden.
Siehe auch: DELETE, DELETED(), EDIT, PACK, RECALL.

SET DELIMITER

Syntax: SET DELIMITER TO <string>
SET DELIMITER TO DEFAULT
SET DELIMITER ON/OFF

DELIMITER bestimmt die Ausgabe von Begrenzern für Eingabebereiche im Direktmodus (APPEND, EDIT, CHANGE usw.). Durch SET DELIMITER TO <string> werden diese Begrenzer bestimmt. Ein oder zwei Zeichen sind für <string> möglich. Besteht er aus einem Zeichen, so begrenzt es den Eingabebereich an beiden Seiten. Besteht er aus zwei Zeichen, so wird das erste Zeichen als vorderes Begrenzungszeichen, das zweite Zeichen als hinteres Begrenzungszeichen verwendet.

DELIMITER ON schaltet die Anzeige der Begrenzer ein.

Standardbegrenzer ist der Doppelpunkt. Diese Einstellung wird durch SET DELIMITER TO DEFAULT erreicht.

Diese SET-Kommandos beeinflussen nicht die Darstellung im Innern des Eingabebereichs, d.h., die durch SET COLOR gewählten Einstellungen bleiben davon unberührt.
Siehe auch: @, APPEND, CHANGE, EDIT, INSERT, READ.

SET DEVICE

Syntax: SET DEVICE TO PRINTER/SCREEN

SET DEVICE TO PRINTER bewirkt, daß alle Ausgaben, die durch ein @-Kommando erzeugt werden, zum Drucker umgelenkt werden. Sie erscheinen nicht auf dem Bildschirm. Der GET-Teil eines solchen Kommandos wird ignoriert. Soll dabei auf eine Zeile gedruckt werden, die vor der aktuellen Zeile liegt, so wird ein Seitenvorschub ausgeführt, d.h., eine rückwärts gerichtete Ausgabe auf den Drucker ist nicht möglich.
SET DEVICE TO SCREEN ist der Standard.
Siehe auch: @, SET FORMAT, SET PRINT.

SET ECHO

Syntax: SET ECHO ON/OFF

SET ECHO ist eins der vier SET-Kommandos, die den Test von dBASE-Kommandofiles unterstützen. ECHO ON bewirkt, daß die abgearbeiteten Kommandos auf dem Bildschirm (SET DEBUG OFF) oder auf dem Drucker (SET DEBUG ON) ausgegeben werden. Mit SET STEP ON kann ein schrittweiser Betrieb erreicht werden.
Siehe auch: SET DEBUG, SET STEP, SET TALK.

SET ESCAPE

Syntax: SET ESCAPE ON/OFF

ESCAPE ON bewirkt, daß jedes Kommando durch Drücken der **Escape**-Taste abgebrochen werden kann. Das nächste Kommando wird von der Tastatur angefordert.

SET EXACT

Syntax: SET EXACT ON/OFF

SET EXACT beeinflußt den Vergleich von Zeichenketten. Gilt EXACT ON, so sind nur solche Zeichenketten gleich, die, neben der Gleichheit der Zeichen von links nach rechts betrachtet, auch die gleiche Länge haben.

Beispiel: Wir nehmen unser Datenbankfile 'lager.dbf' in Benutzung. Da EXACT OFF Standard ist, liefert das Kommando

```
LIST FOR artikel = "Kurbel"
```

eine Liste aller Kurbelwellen. Wird vorher SET EXACT ON gesetzt, so würde das gleiche Kommando keinen Artikel auslisten, denn es gibt keinen Artikel mit der Bezeichnung "Kurbel" (Abschn. 4.5).

SET FILTER

Syntax: SET FILTER TO [<bedingung>]

SET FILTER TO setzt eine Auswahlbedingung, die für alle Kommandos gültig ist und die nur auf den Arbeitsbereich wirkt, in dem sie abgearbeitet wird. Zusätzlich können in den Kommandos weitere Auswahlbedingungen (FOR/WHILE <bedingung>) gegeben werden. Beide Bedingungen werden durch die logische Operation .AND. (Abschn. 4.4) verknüpft. Ein Kommando wird nur dann ausgeführt, wenn beide Auswahlbedingungen erfüllt sind. In jedem Arbeitsbereich kann ein separater Filter festgelegt werden.

SET FORMAT TO ohne Bedingung setzt den Filter für diesen Arbeitsbereich auf den Standard zurück.

Standardmäßig gilt SET FILTER TO .T., d.h., der Filter bewirkt keine Einschränkung.

Beispiel: In dem Datenbankfile 'lager.dbf' sollen Sätze ausgelistet und später bearbeitet werden. Die Kommandos sollen sich nur auf Artikel der Firma C&C beziehen.

```
SET FILTER TO hersteller = "C&C"
```

erspart uns die Eingabe der Auswahlbedingung nach jedem Kommando (Abschn. 3.7).

SET FIXED

Syntax: SET FIXED ON/OFF

FIXED ON bewirkt, daß alle numerischen Werte mit genau so vielen Dezimalstellen ausgegeben werden, wie in SET DECIMALS festgelegt wurde.

Gilt FIXED OFF, so wird die Anzahl der auszugebenden Dezimalstellen so bestimmt, wie es im Kommando SET DECIMALS definiert wurde.

Ausnahmen sind die Funktionen RECNO(), DOW() und LEN(), die immer ganzzahlige Werte liefern.

Siehe auch: SET DECIMALS.

SET FORMAT

Syntax: SET FORMAT TO <file>

Das File muß ein Formatfile sein, d.h., es hat die Namenserweite-

rung '.fmt' und enthält nur @-Kommandos und Kommentare. Die Kommandos des Formatfiles bestimmen den Bildschirmaufbau bei den Kommandos APPEND, CHANGE, EDIT und INSERT. Durch SET FORMAT TO wird eine private Bildschirmmaske in Benutzung genommen. Das Formatfile bestimmt die Bildschirmmaske bis zum nächsten SET FORMAT TO oder bis zum CLOSE FORMAT. In jedem Arbeitsbereich kann ein separates Formatfile aktiv sein.
Siehe auch: @, APPEND, CHANGE, CLOSE FORMAT, EDIT, INSERT, READ, SET DEVICE (Abschn. 13.2).

SET FUNCTION

Syntax: SET FUNCTION <ausdr> TO <string_ausdr>

SET FUNCTION dient dazu, die Funktionstasten mit bestimmten Zeichenketten zu belegen. Es existieren die Funktionstasten F1 bis F10. <ausdr> liefert bei Berechnung einen ganzzahligen Wert, der die Nummer der Funktionstaste bestimmt. Deren Belegung wird definiert durch die Zeichenkette <string_ausdr>. Drückt man danach die entsprechende Funktionstaste, so erscheint die Zeichenkette auf dem Bildschirm.

Die Zeichenkette darf höchstens aus 30 Zeichen bestehen. Ein Semikolon in der Zeichenkette wirkt beim Betätigen der Funktionstaste wie ein **Enter**. Fehlt das Semikolon am Ende der Zeichenkette, so kann der Nutzer die Zeichenkette ergänzen. Das ist dann sinnvoll, wenn die Zeichenkette aus einem Kommando besteht, das manchmal durch Parameter ergänzt werden muß. Schließt die Zeichenkette mit einem Semikolon ab, so wird beim Drücken der Funktionstaste die Zeichenkette mit **Enter** abgeschlossen. Ist die vor dem Semikolon stehende Zeichenkette wieder ein Kommando, so wird dieses Kommando sofort ausgeführt. Enthält die Zeichenkette mehrere Kommandos, die jeweils durch ein Semikolon getrennt sind, so werden diese Kommandos hintereinander ausgeführt.

Funktionstasten sind auch innerhalb von MODIFY COMMAND wirksam, wenn der dBASE-interne Editor verwendet wird. Die durch die Funktionstaste definierte Zeichenkette wird dann in den Text des editierten Files aufgenommen. Ein Semikolon wirkt auch hier wie ein **Enter**. Das kann vorteilhaft benutzt werden, wenn ähnliche Zeichenketten mehrfach in einem File vorkommen. Belegen wir z. B. die Funktionstaste F2 mit der Zeichenkette "@ ROW(),m_spalte SAY GET", so erhält man beim Drücken von F2 dieses Gerüst eines @-Kommandos. Im Editor kann man nun mit dem frei beweglichen Kursor diesen Text modifizieren.

Wurde ein privater Editor für MODIFY COMMAND eingebunden (Abschn. 12.2), so gilt dessen Belegung der Funktionstasten, bis das Editieren abgeschlossen wird.

Die Funktionstaste F1 kann nicht privat definiert werden. Die Standardbelegungen der Funktionstasten sind in Tafel 17.2 wiedergegeben.

Beispiele: Die Belegung der Funktionstaste F8 soll dahingehend verändert werden, daß "DISPLAY OFF" ohne ein abschließendes **Enter** erscheint:

SET FUNCTION 8 TO "DISPLAY OFF"

Man kann durch Eingabe von **Enter** das Kommando so abschicken, oder man kann OFF erst streichen, oder man kann das Kommando durch eine Feldliste und eine Auswahlbedingung erweitern.

Tafel 17.2. Standardbelegung der Funktionstasten

Funktions-taste	Zeichenkette	Funktions-taste	Zeichenkette
F1	help;	F6	display status;
F2	assist;	F7	display memory;
F3	list;	F8	display;
F4	dir;	F9	append;
F5	display structure;	F10	edit;

SET HEADING

Syntax: SET HEADING ON/OFF

HEADING ON bewirkt, daß bei den Kommandos AVERAGE, DISPLAY, LIST und SUM die Feldbezeichner über den entsprechenden Spalten ausgegeben werden. Die Breite, in der eine Spalte dann ausgegeben wird, ist entweder die Breite, so wie sie aus dem Datenbankfile ermittelt wird, oder die Breite der Feldbezeichnung. Es wird der größere Wert von beiden genommen. Dadurch entsteht eine vernünftig lesbare Tabellenform mit Überschriften. Allerdings können die Spalten dadurch auch breiter werden (vgl. Abschn. 3.4). Durch HEADING OFF wird dann nur soviel Platz in einer Zeile benötigt, wie es die Darstellung der Werte verlangt. Zwischenräume zwischen den Spalten werden immer eingefügt. Die Anzeige der Satznummer kann durch das Schlüsselwort OFF in den Kommandos unterdrückt werden.

SET HELP

Syntax: SET HELP ON/OFF

HELP ON bewirkt, daß nach dem Erkennen eines falschen Kommandos eine Anfrage erscheint: "Wünschen Sie HILFE (J/N)?". Wünscht man Hilfe, so erscheint ein Hilfstext zu dem Teil des Kommandos, der als falsch erkannt wurde. Die falsche Kommandozeile wird nicht zur Korrektur angeboten, wie es in einigen Versionen von dBASE II der Fall ist. Die Hilfstexte beziehen sich immer auf das Kommando, in dem der Fehler auftrat. Gibt man z.B. einen falschen Ausdruck im LIST-Kommando ein, so erscheint ein Hilfstext zu LIST und nicht, wie es sinnvoll wäre, zur syntaktischen Einheit Ausdruck.

SET INDEX

Syntax: SET INDEX TO [<fileliste>]

<fileliste> ist eine Liste von Indexfiles. Das erste Indexfile der Liste wird zum Hauptindex und bestimmt die logische Ordnung, nach der das Datenbankfile erscheint. Der Dateizeiger wird auf den ersten Datensatz entsprechend dieser logischen Ordnung gestellt. Alle in der Liste aufgeführten Indexfiles sind damit in Benutzung. Alle in Benutzung befindlichen Indexfiles werden bei Korrekturen am Datenbankfile, insbesondere bei solchen Korrekturen, die den Indexschlüssel verändern, selbständig korrigiert. Das ist der einzige Sinn für die Benutzung mehrerer Indexfiles.

USE gestattet ebenfalls die Angabe von Indexfiles. So ist

```
.USE lager INDEX laherst, laartikel
```

gleichwertig mit

```
USE lager
SET INDEX TO laherst, laartikel
```

'laherst.ndx' und 'laartikel.dbf' seien hier Indexfiles zu unserem Datenbankfile 'lager.dbf'. Die Ordnung nach dem Hersteller, gegeben durch 'laherst.ndx', ist der Hauptschlüssel. Alle Sätze erscheinen nach dem Hersteller geordnet.

SET INDEX kann mehrfach gegeben werden, ohne daß das Datenbankfile abgeschlossen wird. SET INDEX TO schließt alle Indexfiles ab.

In komplexen Kommandos (UPDATE, REPLACE) oder bei vielen Korrekturen erfolgt eine ständige Korrektur der in Benutzung befindlichen Indexfiles (vgl. Abschn. 8). Die Feldliste darf höchstens sieben Filenamen enthalten.

Siehe auch: CLOSE, INDEX, REINDEX, USE.

SET INTENSITY

Syntax: SET INTENSITY ON/OFF

INTENSITY ON bewirkt, daß Eingabebereiche in Standard- und privaten Bildschirmmasken invers dargestellt werden. INTENSITY OFF läßt die Eingabebereiche mit schwarzem Untergrund erscheinen. Ist außerdem SET DELIMITER OFF gegeben, was Standard ist, so ist die Größe der Eingabebereiche in den Bildschirmmasken nicht sichtbar.

SET INTENSITY hat nichts zu tun mit der Intensität der Darstellung, wie sie in SET COLOR TO durch eine Pluszeichen '+' beeinflußt werden kann.

SET INTENSITY schaltet zwischen den beiden Klassen von Bildschirmattributen <standard> und <erweitert> um, wie sie durch SET COLOR definiert werden können. Gilt INTENSITY ON, so können die Attribute für <standard> und <erweitert> verschieden sein. Ist INTENSITY OFF eingestellt, so werden die Attribute für <erweitert> auch für <stan-

dard> verwendet.
Siehe auch: SET COLOR.

SET MARGINE

Syntax: SET MARGINE TO <ausdr>

<ausdr> liefert einen ganzzahligen Wert. Dieser bestimmt den linken Rand bei der Ausgabe auf dem Drucker. Ausgaben auf dem Bildschirm werden davon nicht beeinflußt.
Standard ist die Position 0 für den linken Rand.

SET MENUS

Syntax: SET MENUS ON/OFF

MENUS ON bewirkt, daß bei allen Kommandos, die den Bildschirm im Direktmodus bedienen, eine Hilfestellung für die wichtigsten Steuerfunktionen im oberen Teil des Bildschirms angezeigt wird. In diesen Kommandos kann durch die Funktionstaste F1 diese Anzeige beliebig ein- und ausgeschaltet werden. In den meisten Bildern dieses Buches ist diese Hilfestellung enthalten.
Diese Einstellung hat keinen Einfluß auf Hilfen in MODIFY COMMAND oder bei der Eingabe von Memo-Feldern. Hier gelten die Hilfen, die der für diesen Zweck eingebundene Editor anbietet. Der Standardeditor von dBASE enthält keine Hilfsmenüs.

SET PATH

Syntax: SET PATH TO [<pfad>]

PATH definiert eine Liste von Suchpfaden, die beschritten werden, wenn ein File nicht im aktuellen Verzeichnis (directory) gefunden wird. Ein Pfad ist hierbei eine Folge von Filenamen, die durch einen Rückstrich (back slash "\") voneinander getrennt sind. Der letzte Filename dieser Folge muß ein Filename der geforderten Art sein, z.B. ein Datenbank- oder Indexfilename. Die davor stehenden Filenamen bezeichnen Verzeichnisse. Beginnt ein Pfad mit einem Rückstrich "\", so beginnt der Pfad im Wurzelverzeichnis (root). Von Bedeutung sind Pfade bei der Arbeit mit Festplatten. Die Pfade von dBASE entsprechen den Pfaden im Betriebssystem MS-DOS, machen aber keine Unterschiede zwischen Pfaden für Programme und Daten. Man findet über diesen Pfad sowohl Datenbankfiles, Indexfiles usw. als auch Kommandofiles. Mehrere Pfade werden durch Semikolon oder Komma voneinander getrennt.
SET PATH wirkt nur in solchen Kommandos, die vorhandene Files benutzen. Diese können dann auch modifiziert werden. Soll ein neues File in einem anderen Verzeichnis abgelegt werden, so muß dem Filenamen der vollständige Pfadname vorangestellt werden.

Beispiel: Befinden wir uns im Verzeichnis 'kfz' und rufen wir dort

dBASE auf, so wird dBASE auf dem Pfad gesucht, der durch PATH im Betriebssystem (nicht in dBASE!) definiert ist. dBASE findet dort auch all seine Überlagerungen und Hilfsfiles. In dBASE geben wir:

```
SET PATH TO \privat;\dbase3;d:\listen
USE lager
```

'lager.dbf' wird zuerst im aktuellen Verzeichnis 'kfz' gesucht. Ist es dort nicht vorhanden, so wird es im Verzeichnis 'privat' gesucht, welches ein Unterverzeichnis des Wurzelverzeichnisses (root) ist (kenntlich durch "\" am Anfang). Wird es dort auch nicht gefunden, dann wird im Verzeichnis 'dbase3' gesucht. Alle Suchen erfolgten bisher auf dem Gerät, von dem aus dBASE aufgerufen wurde. Als nächstes wird dann auf dem Gerät 'd:' im Verzeichnis 'listen' gesucht.

Der Suchpfad, der im Betriebssystem eingestellt ist, spielt für das Auffinden der Nutzerfiles in dBASE keine Rolle.

Ein Pfad kann mit einer Gerätebezeichnung beginnen. Pfade werden in der Reihenfolge beschritten, in der sie in der Pfadliste auftreten.

SET PATH überschreibt alle bisherigen Suchpfade. Ohne Pfadliste wird als Suchpfad das aktuelle Verzeichnis festgelegt.

Nach dem Verlassen von dBASE gelten die Pfaddefinitionen des Betriebssystems, wie sie über PATH und ggf. über APPEND (vgl. Kommandobeschreibung des Betriebssystems, z.B. [Smo87]) festgelegt wurden.

Standard für den Suchpfad ist das aktuelle Verzeichnis.

Siehe auch: SET DEFAULT.

SET PRINT

Syntax: SET PRINT ON/OFF

PRINT ON gibt alle Ausgaben, die nicht mit dem @-Kommando erzeugt werden, auch auf dem Drucker aus. Gilt SET CONSOLE ON und SET PRINT ON, so erfolgt die Ausgabe auf dem Bildschirm und auf dem Drucker. Ist kein Drucker verfügbar, so führt das zu folgender Mitteilung:

```
. SET PRINT ON
Schreiben.
Schreibfehler  Einheit PRN
A(bbruch), W(iederholen), I(gnorierern)?
```

Ignorieren führt immer zu der gleichen Ausschrift, <Esc> ebenfalls. Den Drucker bereit machen und 'w' eingeben ist die beste Lösung. 'a', für Abbruch, läßt dBASE "bis auf die Betriebssystemebene abstürzen". Alle Tests ergaben, daß die benutzten Files hinterher ordnungsgemäß abgeschlossen waren. Da dieser Fall vom Hersteller nicht dokumentiert ist, sollte man sich nicht darauf verlassen. Werden die Files nicht abgeschlossen, so kann das zum Datenverlust führen. Wir empfehlen deshalb,

1. dieses Kommando nicht in ein Kommandofile zu stecken und

2. besser ein Alternate-File anzulegen, welches hinterher mit einem Editor überarbeitet und dann gedruckt werden kann.

Siehe auch: SET ALTERNATE, SET CONSOLE, SET DEVICE, SET ESCAPE.

SET PROCEDURE

Syntax: SET PROCEDURE TO [<proc_file>]

<proc_file> ist der Name eines Prozedurfiles, in dem mehrere Prozeduren enthalten sein können. SET PROCEDURE TO <proc_file> legt fest, daß bei nachfolgenden Kommandos DO <proc> im Prozedurfile <proc_file> nach der Prozedur <proc> gesucht wird. Höchstens ein Prozedurfile kann zu einem Zeitpunkt eröffnet sein. Dieses File bleibt bis zum nächsten SET PROCEDURE oder CLOSE PROCEDURE eröffnet. SET PROCEDURE TO ohne Filenamen schließt ein möglicherweise offenes Prozedurfile ab.

Maximal 32 Prozeduren können in einem Prozedurfile enthalten sein. Siehe auch: DO, PARAMETERS, Abschnitt 11.2, 11.3.

SET RELATION

Syntax: SET RELATION TO [<feld> INTO <alias>]

SET RELATION verbindet zwei aktive Datenbankfiles über ein Schlüsselfeld <feld> in zwei Arbeitsbereichen. Der zweite Arbeitsbereich wird durch den Aliasnamen festgelegt. Eine ausführliche Besprechung enthält Abschnitt 14.

SET SAFETY

Syntax: SET SAFETY ON/OFF

SAFETY ON bewirkt, daß beim Streichen und Überschreiben von Files mit bestimmten Kommandos eine Anfrage gerichtet wird:

```
. USE lager
. COPY TO x
x.dbf bereits vorhanden, überschreiben erwünscht? (J/N)
```

Diese Anfrage erscheint aber nicht in allen Fällen, in denen ein File überschrieben wird. So überschreibt SET ALTERNATE ein bereits vorhandenes File ohne Anfrage. Anfragen erfolgen bei COPY, COPY FILE, CREATE, INDEX, JOIN, SAVE, SORT, TOTAL, UPDATE, ZAP.

SET STEP

Syntax: SET STEP ON/OFF

SET STEP ist eins der vier Kommandos zum Test von Kommandofiles.

Es bewirkt, daß nach der Abarbeitung jedes Kommandos, auch wenn es in einem Kommandofile steht, der Nutzer die Steuerung erhält. Mit **Esc** kann er das Kommandofile beenden oder mit einer beliebigen anderen Taste das nächste Kommando abarbeiten lassen.
Siehe auch: SET DEBUG, SET ECHO, SET TALK.

SET TALK

Syntax: SET TALK ON/OFF

SET TALK ON bewirkt, daß das Resultat jeder Berechnung auf dem Bildschirm ausgegeben wird. Das ist bei der Eingabe der Kommandos am Bildschirm vorteilhaft, da man das Ergebnis ohne ein zusätzliches Kommando sofort sehen kann. Bei der Abarbeitung von Kommandofiles ist das lästig, da man i.allg. nicht mehr weiß, woher der berechnete Wert stammt. SET TALK OFF ist deshalb meist eins der ersten Kommandos, die in einem Kommandofile stehen (Abschn. 4.7).

SET UNIQUE

Syntax: SET UNIQUE ON/OFF

SET UNIQUE setzt Optionen für das Indizieren (INDEX). Es bewirkt, daß beim Indizieren nur verschiedene Schlüsselinhalte in das Indexfile aufgenommen werden (ON). Man hat über ein solches Indexfile nur zu solchen Sätzen Zugriff, die sich in ihren Schlüsseln unterscheiden. Wird ein Datensatz geändert, so muß ein derartiges Indexfile nicht korrekt korrigiert werden.

Beispiele: Wir benutzen das Datenbankfile 'lager.dbf' und interessieren uns für alle Hersteller, wobei jeder Hersteller genau einmal aufgelistet werden soll.

```
  USE lager
. SET UNIQUE ON
* In unserem Fall gibt es nur 6 verschiedene Hersteller.
  INDEX ON hersteller TO laherun
      6 Sätze indiziert
. LIST hersteller
 Satz #  hersteller
      2  Achse&Welle
      6  Blech&Co.
      1  C&C
      7  Dicht&Fest
      4  Gummi&Sohn
      5  VEB Guß
```

Im zweiten Beispiel wollen wir feststellen, ob kein Artikel doppelt aufgenommen wurde:

```
. SET UNIQUE ON
. INDEX ON artikel TO laartikel
     9 Sätze indiziert
```

Ist die Anzahl der indizierten Sätze gleich der Anzahl der Sätze in der Datenbank, so ist jeder Schlüssel genau einmal vorhanden. Andernfalls könnte eine Zusammenfassung der Datensätze mit gleichem Schlüssel sinnvoll sein (TOTAL, Abschn. 15.3).
Siehe auch: FIND, INDEX, REINDEX, SET INDEX, USE.

17.2. Das Konfigurationsfile

Die Voreinstellungen für bestimmte Optionen von dBASE können beim Laden von dBASE festgelegt werden. Einige dieser Optionen können bei der Arbeit nach Belieben verändert werden, andere sind nur beim Laden von dBASE wählbar. Diesen Vorgang bezeichnet man als Konfigurieren. Der Unterschied zum Installieren besteht darin, daß beim Installieren ein Programm für eine bestimmte Umgebung (Bildschirm, Drucker, Tastatur, Betriebssystem) einmal angepaßt wird und dann bei der Benutzung nur noch geladen wird. Zum Installieren benötigt man ein spezielles Programm, das Installationsprogramm. dBASE II muß für einen Rechner installiert werden.

Wir kennen auch für dBASE III eine Installation, die aber hier einen ganz anderen Sinn hat. Dabei erfolgt keine Anpassung an die Umgebung. dBASE III wird konfiguriert. Das ist eine Technik, die von vielen Programmen unter MS-DOS benutzt wird. Ein Programm benutzt dabei ein Datenfile (Textfile), in dem die Angaben über die zu setzenden Optionen angegeben sind. Das Betriebssystem selbst wertet beim Laden - auch als Booten, Urladen oder Kaltstart bezeichnet - ein File 'config.sys' aus. dBASE III sucht entsprechend dem eingestellten Suchpfad nach einem File 'config.db' und entnimmt die zu setzenden Optionen daraus.

Jede Option steht in einer Zeile. Wir listen zuerst die Optionen auf, zu denen die entsprechenden SET-Kommandos existieren. Eine detaillierte Beschreibung erübrigt sich damit (vgl. Abschn. 17.1).

```
ALTERNATE = <file>
* Wirkt wie SET ALTERNATE TO <file> und
* SET ALTERNATE ON, d.h., das Alternate-File wird definiert und
* zur Ausgabe eröffnet.
BELL      = ON/OFF
CARRY     = ON/OFF
COLOR     = [<standard>] [,<erweitert>] [,<rand>]
COMMAND   = <kommando>
* Hier kann ein beliebiges Kommando stehen. So auch der Aufruf
* eines Kommandofiles, so daß sofort die Nutzeroberfläche
* sichtbar gemacht werden kann.
* Bem.: Beim Aufruf von dBASE ist ein Kommandofilename als
```

```
* Parameter wirkungslos.
CONFIRM    = ON/OFF
CONSOLE    = ON/OFF
DEBUG      = ON/OFF
DECIMALS   = ON/OFF
DEFAULT    = ON/OFF
DELETED    = ON/OFF
DELIMITERS= <string>
DELIMITERS= ON/OFF
DEVICE     = SCREEN/PRINTER
ECHO       = ON/OFF
ESCAPE     = ON/OFF
EXACT      = ON/OFF
* In der folgenden Zeile ist für <i> eine Zahl zwischen
* 2 und 10 einzusetzen. Es wird die Belegung der entsprechenden
* Funktionstaste definiert.
F<i>       = <string>
* Für <n> ist eine Zahl zwischen 35 und 1023 einzusetzen.
* Sie bestimmt die Anzahl der @-Kommandos, die zu einem
* Zeitpunkt abgearbeitet wurden, aber noch kein zugehöriges READ
* erfolgte (hängende GETS, pending GETS).
GETS       = <n>
HEADINGS   = ON/OFF
HELP       = ON/OFF
INTENSITY  = ON/OFF
MARGINE    = ON/OFF
MENUS      = ON/OFF
PATH       = <pfadliste>
PRINT      = ON/OFF
SAFETY     = ON/OFF
STEP       = ON/OFF
TALK       = ON/OFF
UNIQUE     = ON/OFF
```

Die folgenden Angaben des Konfigurationsfiles haben keine Entsprechungen als Kommandos:

```
Syntax:   MVARSIZ    = <zahl>
```

<zahl> ist eine ganze Zahl mit 1 <= <zahl> <= 31. Sie legt die Größe des Bereichs für die Speichervariablen in KByte fest.

```
Syntax:   PROMPT     = <string>
```

<string> ist eine Zeichenkette, die auch aus mehr als einem Zeichen bestehen kann. Zeichenkettenbegrenzer sind nicht erforderlich.

```
Syntax:   TEDIT      = <file>
```

<file> ist der Name eines Texteditors. Dieser wird immer dann aufgerufen, wenn MODIFY COMMAND abgearbeitet wird. Es ist vom Nutzer zu gewährleisten, daß der so angeschlossene Editor seine Überlagerungen findet. Insbesondere bei WordStar kann sich ein '.bat'-File mit

einer APPEND-Anweisung erforderlich machen.

Nach dem Verlassen des Editors kehrt die Steuerung zu dBASE zurück. Es sind alle Einstellungen von dBASE erhalten, wie sie vor dem Aufruf von MODIFY COMMAND gültig waren.

Syntax: WP = <file>

Auch hier ist <file> der Name eines Texteditors (WP - word processor), d.h. des entsprechenden '.com'- oder '.exe'-Files. Er wird aufgerufen, wenn ein Memo-Feld editiert wird. Die Organisation des '.dbt'-Files, in dem die Memo-Felder abgelegt werden, verbleibt bei dBASE. Das hat zur Folge, daß der Editor so aufrufbar sein muß, daß das '.dbt'-File sofort in Benutzung ist. Die vorliegenden Versionen von WordStar (Version 3.40) sind dafür nicht geeignet. Weitere Hinweise befinden sich im Abschnitt 12.2.

Durch die Konfigurierung besitzt der Nutzer eine effektive Möglichkeit, das gleiche dBASE-System für die einzelnen Arbeitsaufgaben an seine Belange anzupassen. Für verschiedene Aufgaben können unterschiedliche Einstellungen durch verschiedene Konfigurationsfiles vorgenommen werden. Das wird dadurch erreicht, daß dBASE ein Konfigurationsfile auf dem vorgegebenen Pfad sucht. Startet man dBASE von einem anderen Verzeichnis aus, so kann ein anderes Konfigurationsfile die Voreinstellung bestimmen bei völlig gleichen Files des dBASE-Systems.

17.3. dBASE, MS-DOS und Kooperation mit anderen Programmen

Betrachten wir nun die Parameter des Betriebssystems MS-DOS, die die Arbeit von dBASE beeinflussen. Zwei Parameter sind wichtig:

```
buffers = 20        und
files = 20
```

Diese Angaben befinden sich im Konfigurationsfile 'config.sys', welches beim Laden des Betriebssystems ausgewertet wird. 'files' legt die obere Grenze für die gleichzeitig eröffneten Files und Ein- und Ausgabegeräte fest. Wird dieser Wert zu klein gewählt, so kann die in dBASE mögliche maximale Anzahl von gleichzeitig eröffneten Files (15, Abschn. 2.7) dadurch weiter begrenzt werden. Ein Wert von 20 wird allgemein empfohlen.

'buffers' legt die Anzahl der im System verfügbaren Ein- und Ausgabepuffer fest. Für jeden Puffer werden 528 Byte des Hauptspeichers belegt, die von Anwenderprogrammen dann nicht mehr genutzt werden können. Ist nur der minimale Speicher von 256 KByte verfügbar, so steht bei zu vielen Puffern zuwenig Speicherplatz als Arbeitsbereich für dBASE zur Verfügung. Wählt man zu wenige Puffer, so müssen zu oft Daten zwischen den Datenträgern und dem Speicher transportiert werden. Der Zugriff zur Platte beschränkt dann die Geschwindigkeit des Systems. Nimmt man zu viele Puffer, dann wird die interne Puffer-

verwaltung zeitaufwendiger. Die Geschwindigkeit des Prozessors zusammen mit dem verkleinerten Arbeitsbereich für die Anwenderprogramme wird dann zum Engpaß. Messungen haben ergeben [Cas87] [Smo87], daß 10 bis 20 Puffer günstig sind.

Sollte bei der Arbeit mit dBASE diese Grenze erreicht werden, so sind die Angaben im File 'config.sys', welches ein Textfile ist, entsprechend zu ändern

Welche Möglichkeiten bietet dBASE, um

1. Daten mit anderen Systemen auszutauschen und

2. mit anderen Programmen zusammenzuarbeiten?

Die erste Frage beantworten wir in den folgenden beiden Abschnitten. Der zweiten Frage wollen wir uns jetzt zuwenden.

Unter der Zusammenarbeit mit anderen Programmen verstehen wir den Aufruf eines Programms aus dBASE heraus. Von dBASE aus betrachtet, sind solche Programme extern. Man möchte also in einem externen Programm Dinge erledigen, die mit dBASE nicht oder nur schwerfällig gelöst werden können. Anschließend möchte man nach dBASE zurückkehren und genau die gleichen Bedingungen vorfinden, wie sie beim Aufruf des externen Programms existierten. Ein externes Programm kann in einer beliebigen Programmiersprache geschrieben sein.

Glaubt man ohne einen bestimmten Assemblerbefehl nicht auskommen zu können, so sollte man die Lösung in dieser Reihenfolge suchen:

1. Läßt sich vielleicht das gleiche aus dBASE heraus erledigen? Im Abschnitt 17.6 geben wir dafür Beispiele.

2. Kann man die notwendigen Assemblerteile sinnvoll aus einer Programmiersprache heraus erzeugen?

3. Erst wenn 1. und 2. begründet verneint wurden, sollte man ein separates Programm in Assemblersprache schreiben und dieses als externes Programm aufrufen.

Der Aufruf eines externen Programms aus dBASE heraus erfolgt durch

RUN <kommandozeile>

<kommandozeile> ist dabei eine Kommandozeile des MS-DOS. Es besteht kein Unterschied zwischen dem Aufruf von Dienstprogrammen oder privat erstellten Programmen. RUN lädt das Programm in den freien Speicher und führt es aus. Ist nicht genug freier Speicherplatz verfügbar, so wird eine Fehlernachricht ausgegeben. Es ist sogar möglich, ein Kommandofile (.bat-File) des Systems aufzurufen. Das gerufene Programm findet seine Überlagerungen entsprechend dem im System eingestellten Suchpfad. So ist es z.B. möglich, aus dBASE heraus ein Textverarbeitungssystem aufzurufen, z.B. WordStar. Das hat gegenüber dem Einbinden eines privaten Editors über das File 'config.db' (Abschn. 12.2) den Vorteil, daß beim Aufruf über RUN mehrere Editoren

zur Verfügung stehen. Einfache Arbeiten werden mit dem dBASE-internen Editor erledigt, umfangreichere, wie z.B. Blockoperationen, mit einem komfortableren Editor.

Zur Veranschaulichung dieses Prinzips schreiben wir das folgende Pascal-Programm:

```
PROGRAM farproc;
 (* Dieses Programm wird von dBASE aus aufgerufen.
    Statt der einfachen Ausschrift könnte es auf beliebige
    dBASE-Files zugreifen. *)

 BEGIN
   Writeln;
   Writeln('Hallo, hier ist das externe Programm!')
 END.
```

Es wird mit dem Turbo-Pascal-Compiler in ein .com-File mit dem Namen 'farproc.com' übersetzt. Aufruf und Wirkung sind:

```
. RUN farproc
Hallo, hier ist das externe Programm!
```

Enthält die nach RUN angegebene Kommandozeile nach dem Programmnamen noch weitere Angaben, so werden diese wie bei Aufrufen auf der Ebene des MS-DOS behandelt. Rufen wir z.B.

```
RUN ne x.txt
```

auf, so wird der Norton-Editor geladen. Dieser eröffnet das File 'x.txt' zur Bearbeitung. Für die Übergabe von Parametern, die keine Files sind, ist dieser Mechanismus nicht tauglich.

Welche Möglichkeiten der Parameterübergabe bestehen? Der sicherste Weg besteht darin, die Parameter in einem Textfile (COPY TO ... SDF) oder in einem Datenbankfile abzulegen und von der Programmiersprache aus auf dieses File zuzugreifen. Die folgenden Abschnitte beschreiben den Zugriff.

Dann besteht die Möglichkeit, kurze Informationen im Speicher abzulegen. Dieser Weg ist sehr gefährlich, da man einen Platz im Speicher finden muß, der garantiert frei ist und dessen absolute Adresse man kennt. Die Funktionen PEEK und POKE können dann verwendet werden, um die Parameter byteweise unter einer absoluten Adresse abzulegen bzw. von dort zu lesen. PEEK und POKE sind für dBASE III nicht dokumentiert, werden in den Hilfsmenüs nicht angezeigt, wirken aber dennoch. Da POKE sehr gefährlich ist und zur Zerstörung des Systems führen kann, sind PEEK und POKE in diesem Buch nicht in die Liste der Kommandos aufgenommen worden.

```
PEEK(<absolute_adr>)
```

liefert den Inhalt des Bytes mit der absoluten Adresse <absolute _adr>. Die Angabe einer segmentierten Adresse ist nicht möglich.

```
POKE(<absolute_adr>, <bytefolge>)
```

<bytefolge> ist eine Folge von Werten zwischen 0 und 255, die jeweils durch ein Komma getrennt sind. Die berechneten Werte werden in angegebener Reihenfolge ab der Adresse <absolute_adr> im Speicher abgelegt.

Welche Speicherplätze dafür genutzt werden können, hängt auch von der Version des Betriebssystems ab. Ein sicherer Platz ist der Bereich im Interruptvektor, der für den Anschluß privater Serviceroutinen gedacht ist. Wir machen aber darauf aufmerksam, daß bei einer solchen Parameterübergabe Schwierigkeiten beim Übergang zu anderen Systemen entstehen können. Es handelt sich nicht um eine "saubere" Lösung. Wir empfehlen deshalb den sicheren Weg über ein File.

dBASE III akzeptiert auch die aus dBASE II bekannten Kommandos

```
CALL <speichervar>
LOAD <hexfile>          und
SET CALL TO <absolute_adr>
```

Diese Kommandos sind unter dBASE III nicht dokumentiert, und ihre Wirkung ist damit nicht definiert! Wir empfehlen hier, das <hexfile> (File im Intel-hex-Format) in ein externes Programm umzuformen und dieses dann über RUN aufzurufen. Wir raten dringend davon ab, Befehle binär kodiert in den Speicher zu schreiben und dann mit einem CALL-Kommando abzuarbeiten! Dieser Weg wird gelegentlich in dBASE II vorgeschlagen, wird jedoch in komplexen Betriebssystemen immer problematischer und gefährlicher. Hier ist RUN eine bessere Lösung.

17.4. Überführung von dBASE II nach dBASE III

Hat man dBASE II-Datenbanken, so möchte man diese nach dBASE III übernehmen. Dazu existiert ein Konvertierungsprogramm dCONVERT. Es bietet in seinem Hauptmenü die Konvertierung aller Typen von dBASE II-Files an. Kommandofiles müssen vorher die Erweiterung '.prg' bekommen. Ebenso müssen Files umbenannt werden, die zwar unter CP/M gültige Filenamen sind, unter MS-DOS aber nicht gestattet sind. Das gilt für Sonderzeichen in Filenamen. Normale Filenamen, die aus Buchstaben und Ziffern bestehen, werden problemlos akzeptiert.

dBASE II-Datenbankfiles werden in dBASE III-Datenbankfile konvertiert. Diese Konvertierung ist notwendig, da die interne Struktur der Datenbankfiles verschieden ist. Damit ist gesichert, daß die Daten nicht neu erfaßt werden müssen.

Kommandofiles werden konvertiert. Dabei kann nicht garantiert werden, daß die Bedeutung der Kommandos immer richtig übernommen wird. Auf einige mögliche Fehler wird hingewiesen. Diese hängen vor allem mit dem Test auf das Erkennen des Fileendes (EOF()) zusammen.

Eine zweite Gruppe von Fehlern, die nicht erkannt werden, bezieht

sich auf die Benutzung von Speichervariablen in Kommandofiles. Diese sind in dBASE III lokal (Abschn. 11). dBASE II kennt nur globale Variable. Diese Fehler werden behoben, indem man diese Variablen in dem Kommandofile, in dem sie definiert werden, als PUBLIC erklärt.

Eine dritte Gruppe von Fehlern ist bei der Konvertierung nicht erkennbar. Sie betrifft die Wirkung mehrerer Kommandos untereinander und liefert auch bei der Abarbeitung keine Fehlermeldungen. Diese Fehler hängen wesentlich vom Programmierstil ab. Wir weisen hier auf die veränderte Semantik des Kommandos LOCATE hin. Ein erfolgloses LOCATE liefert in dBASE III eine Satznummer, die um eins größer ist als die Nummer des letzten existierenden Satzes. In dBASE II liefert LOCATE im gleichen Fall die Nummer des letzten Satzes. Je nachdem wie das Mißlingen des LOCATE-Kommandos unter dBASE II getestet wurde, ist die Konvertierung korrekt oder nicht.

Files für Speichervariable (.mem). Formatfile (.fmt und .frt) werden konvertiert.

Indexfiles werden nicht konvertiert. dCONVERT erzeugt bei der Überführung eines Indexfiles ein Kommandofile mit der Namenserweiterung '.rx'. Dieses ist unter dBASE III abzuarbeiten.

Insgesamt ist festzustellen, daß durch ein derartiges Programm der Übergang von dBASE II nach dBASE III gesichert ist. Eine erneute Datenerfassung ist nicht möglich und wäre auch nicht akzeptabel. Wir müssen allerdings die erzeugten Kommandofiles genau untersuchen. Hier bewahrheitet sich wieder der Satz, daß ein Programm nur einmal geschrieben, aber vielleicht 50mal gelesen wird, und das noch von anderen Personen.

17.5. Datenaustausch zwischen dBASE und anderen Systemen

Wie sieht es nun mit dem Übergang von dBASE III in andere Systeme aus? Wir denken dabei natürlich zuerst an UNIX-ähnliche Systeme. Der Lösungsweg gilt aber für alle zukünftigen Systeme.

Zuerst ist das Problem des Transports von Textfiles in das Zielsystem zu lösen. Da dieses Problem nichts mit dBASE zu tun hat, sondern generell gelöst werden muß, können wir darauf vertrauen, daß eine solche Lösung verfügbar ist, bzw. wir beginnen sie zu schaffen. Tatsächlich ist der Datentransport von Textfiles zwischen den Betriebssystemen CP/M-80, CP/M-86, MS-DOS, XENIX, VENIX und WEGA (XENIX, VENIX und WEGA sind UNIX-ähnliche Systeme) gelöst.

Der Transport von Textfiles sei also möglich. Dann können wir zwei Wege beschreiten:

1. Wir können aus dBASE heraus das Datenbankfile als ein Textfile ausgeben (COPY TO ... SDF). Kann das Datenbanksystem auf dem Zielrechner Textfiles einlesen, wie z.B. dBASE durch APPEND FROM SDF, so definieren wir die Struktur des Datenbankfiles und

lesen es ein.

2. Kann das Zielsystem keine Textfiles in Datenbankfiles umformen, so muß der interne Aufbau der Datenbankfiles des Zielsystems bekannt sein. In diesem Fall sollte eine Konvertierung von Textfiles in Datenbankfiles geschrieben werden.

Von akzeptablen Systemen kann man erwarten, daß der erste Weg gangbar ist. In jedem Fall sollte die Lösung über Textfiles führen!

Kommandofile und Bildschirmmasken sind Textfiles. Ob sie im Zielsystem einen Sinn ergeben, ist zu überprüfen. Alle anderen Files müssen im Zielsystem neu erstellt werden.

17.6. Direkter Zugriff auf die Hardware

Wir sprechen von einem direkten Zugriff auf die Hardware von einem Programm aus, wenn durch Kommandos unmittelbar die Gegebenheiten der Hardware ausgenutzt werden. Mittelbar ist die Hardware natürlich an der Ausführung jedes Kommandos beteiligt. dBASE enthält aber auch einige Kommandos, die die Hardware unmittelbar ansteuern, ohne daß dieser hardwarenahe Charakter dem Nutzer bewußt wird. Ein typisches Beispiel dafür ist das Kommando SET COLOR (Abschn. 17.1). Es gibt Steuerinformationen an den Bildschirm aus bzw. bewirkt die Ausgabe bestimmter Bildschirmattribute zusammen mit dem auszugebenden Text. Der Text kann auch Zeichen des erweiterten Zeichensatzes enthalten, so daß z.B. auch Umrandungen und Tabellen in einer Form dargestellt werden können, die den Leistungen der 16-Bit-Technik angemessen ist. Im Abschnitt 13.2 haben wir Beispiele dafür angegeben.

In diesem Abschnitt wollen wir uns als Beispiel für einen konkreten Anwendungsfall der hardwarenahen Programmierung mit der direkten Ansteuerung eines Druckers beschäftigen. Möchte man die Hardware ansteuern, so kann man natürlich nicht erwarten, daß man ohne konkrete Kenntnisse der Hardware auskommt. Wir erklären die hardwarenahe Programmierung unter dBASE so, daß keine Vorkenntnisse über die Hardare des Druckers erforderlich sind.

Zu einem IBM-PC-kompatiblen Rechner gehört ein Drucker, der die IBM-Drucker-Steuerfolgen versteht. Die Drucker LX86 und FX1000 können z.B. dazu benutzt werden. Die Steuerfolgen sind fest vorgegeben und im Zweifelsfall der Beschreibung des Druckers zu entnehmen. Ist ein solcher Drucker vorhanden, so wird man sehr schnell merken, daß er viel mehr kann, als wir durch die normalen Kommandos von dBASE ausnutzen können. Es ist auch, im Gegensatz zur 8-Bit-Technik, nicht mehr damit getan, Bildschirmausgaben einfach auf den Drucker umzulenken. Das geht nur in den einfachsten Fällen gut. Versuchen wir z.B. die Bildschirmmaske aus dem Abschnitt 13.2 auf den Drucker umzulenken, so entsteht nicht annähernd das gleiche Bild. Der Drucker kann z.B. nicht um Zeilen zurücksetzen, sehr wohl aber innerhalb einer

Zeile.

Ein normaler Drucker kennt mehrere Schriftarten, kann Pseudografikzeichen drucken und hat einen frei programmierbaren Zeichensatz. All diese Möglichkeiten und noch mehr können aus dBASE heraus unmittelbar genutzt werden. Wir zeigen das im folgenden an einem Beispiel.

Wir möchten für unsere Datenbank mit den Datenbankfiles 'lager.dbf' und 'lageradr.dbf' Karteikarten drucken, die üblichen Karteikarten ähnlich sind. Sie sollen die Beschreibung des Inhalts einer Angabe in einer kleineren Schriftart enthalten und den eigentlichen Inhalt in Briefqualität (NLQ, near letter quality). Nach einer unterstrichenen Überschrift wollen wir den Artikel, das Datum der nächsten Lieferung, die Firma und deren Ort ausgeben. Die letzten drei Größen sollen dabei spaltengerecht auf der rechten Seite untereinander stehen.

Wir lösen diese Aufgabe zuerst mit der sequentiellen Ausgabe (?-Kommando) und danach mit der direkten Ausgabe (@-Kommando).

17.6.1. Sonderdruck mit sequentieller Ausgabe

Für die sequentielle Ausgabe stehen uns die Kommandos '?' und '??' zur Verfügung. Auf welchem Gerät die Ausgaben erscheinen, wird durch die Kommandos SET PRINT ON/OFF und SET CONSOLE ON/OFF angegeben. Wir wählen

```
SET PRINT ON
SET CONSOLE OFF
```

Es empfielt sich, die Ausgabe auf den Bildschirm abzuschalten, da alle Steuerzeichen für den Drucker sonst als Zeichen aus dem erweiterten Zeichensatz des Bildschirms dargestellt werden.

Beide Kommandos geben nichts an den Drucker aus. Ebenso kann man mit SET COLOR keine Ausgaben auf den Drucker lenken.

Wir definieren uns Speichervariable, die die benötigten Steuerfolgen für den Drucker enthalten. Die Funktion CHR hilft uns hierbei wesentlich. Sie liefert Zeichen für Argumente i mit

```
1 <= i <= 255.
```

Leider liefert CHR(0) kein Zeichen. Es wird kein Byte an den Drucker übertragen.

Die Steuerfolge für die Umstellung der Druckart auf den Indexdruck ist

```
<Esc> "S" 1 oder hexadezimal   1B 53 01
```

Das Druckbild von Indizes sieht besser aus, wenn es in Schmalschrift erzeugt wird. Ein Steuerbyte ist dafür erforderlich:

```
15          oder hexadezimal  0F
```

Arbeiten wir das Kommando

```
? CHR(27), "S", CHR(1), CHR(15)
```

ab, so wird folgende Bytefolge (hexadezimal) an den Drucker gesendet:

```
0D 0A 1B 20 53 20 01 20 0F
```

Der vorangestellte Zeilenwechsel (0D 0A) stört nicht. Er kann durch Verwendung des ??-Kommandos unterdrückt werden. Schädlich sind jedoch die Leerzeichen (20), die nach jedem Zeichen eingefügt werden. Es werden dadurch völlig andere Steuerfolgen an den Drucker ausgegeben. Ursache dafür ist die Aufzählung der Steuerzeichen im ?-Kommando. Arbeiten wir

```
?? CHR(27) + "S" + CHR(1) + CHR(15)
```

ab, so wird die gewünschte Steuerfolge übertragen. Da wir sie mehrfach benötigen, weisen wir sie einer Speichervariablen 'm_klein' zu

```
m_klein = CHR(27) + "S" + CHR(1) + CHR(15)
```

und schreiben später

```
?? m_klein
```

Arbeiten wir diese Kommandos ab, so wird die Steuerfolge zweimal übertragen: das erstemal bei der Berechnung von m_klein und das zweitemal bei der Abarbeitung des ??-Kommandos. Durch

```
SET TALK OFF
```

können wir die Ausgabe bei der Berechnung von m_klein unterdrücken.

Da wir ständig zwischen Normaldruck und Schmaldruck im Indexmodus hin- und herschalten müssen, definieren wir uns eine Variable 'm_normal', die die Steuerfolge enthält. Das Rückschalten aus dem Indexmodus in den Normalmodus erfolgt durch

```
<Esc> "T"       oder hexadezimal    1B 54
```

Auf normale Schreibdichte wird durch

```
18              oder hexadezimal    12
```

zurückgeschaltet. Wir erhalten:

```
m_normal = CHR(27) + "T" + CHR(18)
```

Die Aufgabe wird durch das folgende Kommandofile gelöst. Vorausgesetzt ist dabei, daß beide Datenbankfiles 'lager.dbf' und 'lageradr.dbf' in Benutzung und verbunden sind über das Feld 'Hersteller'. Das wird erreicht durch

```
USE lager
SELECT 2
USE lageradr INDEX laadrh
SELECT lager
SET RELATION TO hersteller INTO lageradr
```

Unter diesen Voraussetzungen kann das Kommandofile 'karte1.prg' aufgerufen werden.

```
* Kommandofile 'karte1.prg' zum Drucken einer Karteikarte.
* Die Ausgabe erfolgt sequentiell.
SET TALK OFF
SET CONSOLE OFF
SET PRINT ON
* Definition der Steuerfolgen
* Indexmodus, Schmalschrift
m_klein = CHR(27) + "S" + CHR(1) + CHR(15)
* Rückschalten vom Indexmodus auf Normalmodus, Normalschrift
m_normal = CHR(27) + "T" + CHR(18)
* Zwei Leerzeilen und überschrift. Die durchgehende Unterstreichung
* wird durch Zeichen aus dem erweiterten Zeichensatz realisiert.
?
?
? "                    Karteikarte für Ersatzteile"
? "                    ___________________________"
?
?
* 'Artikel:' wird ab Position 10 ausgegeben
?? SPACE(10) + m_klein + "Artikel:" + m_normal
* Nach zwei weiteren Leerzeichen wird der Inhalt des Feldes
* 'artikel' ausgegeben.
?? SPACE(2) + artikel
* Nach fünf weiteren Leerzeichen die Bezeichnung des nächsten
* Ausgabebereichs 'Nächste Lieferung:' in Indexschreibweise
?? SPACE(5) + m_klein + "Nächste Lieferung:" + m_normal
* Das Datum wird nach zwei weiteren Leerzeichen ausgegeben,
* nachdem es in eine Zeichenkette konvertiert wurde.
?? SPACE(2) + DTOC(lieferung)
* Beenden dieser Zeile und eine Leerzeile.
?
?
* 'Firma:' soll linksbündig unter 'Nächste Lieferung:' stehen.
* Das Auszählen der Zeichen nutzt wenig, da der Zeichenabstand des
* Druckers mit der Schriftbreite verändert wird. Wir sind so vorge-
* gangen, daß wir ihn nach einem Probedruck ausgemessen haben.
?? SPACE(37) + m_klein + "Firma:" + m_normal
?? SPACE(9) + hersteller
?
?
?? SPACE(37) + m_klein + "in:" + m_normal
?? SPACE(11) + lageradr->ort
?
SET PRINT OFF
```

```
SET CONSOLE ON
SET TALK ON
RETURN
```

Die gedruckte Karteikarte wird im Bild 17.3 wiedergegeben. Wir sehen, daß die Ausgaben nicht exakt spaltengerecht sind. Das liegt daran, daß in den Zeilen unterschiedlich viele Zeichen in Schmalschrift und in Normalschrift ausgegeben werden. Kann man diese Zeichenanzahlen für jede Zeile und jede Schriftart gleich halten, so entsteht ein spaltengerechter Druck. Wir geben diese Lösung hier nicht an, sondern verweisen auf den nächsten Abschnitt.

Möchten wir eine Umrandung mit drucken, so sind die Zeichen aus dem Sonderzeichensatz kein Problem. Die exakte Einhaltung der Spalten muß dann unbedingt garantiert werden. Die angegebene Lösung ist dafür nicht geeignet.

Karteikarte für Ersatzteile

Flachschieber 29.02.88

C&C

in Rheinsberg

Bild 17.3. Im sequentiellen Modus gedruckte Karteikarte

17.6.2. Sonderdruck mit direkt positionierter Ausgabe

Wir setzen die Aufgabe aus dem vorangegangenen Abschnitt unter den gleichen Voraussetzungen fort. Das Ziel besteht jetzt darin, eine doppelt umrandete Karteikarte mit den gleichen Informationen zu drucken. Dazu positionieren wir die Ausgabe auf dem Drucker direkt unter Ausnutzung des @-Kommandos. Zu berücksichtigen ist, daß wir keine Rückwärtspositionierung von Zeilen ausführen können. Das würde zu einem Seitenvorschub führen. Wir nutzen aber wesentlich die Eigenschaft, innerhalb einer Zeile frei positionieren zu können.

Das @-Kommando wirkt so, daß eine Positionierung um n Zeilen vorwärts, relativ zur aktuellen Position des Druckkopfes, zur Ausgabe von n Paaren cr und lf (carriage return und line feed, hexadezimal 0D 0A) führt. Verändert man den Zeilenabstand während der Ausgabe, so treten die gleichen Probleme auf, wie wir sie für die Spaltenposition durch den Schmaldruck erhalten.

Ein Fortschreiten um n Zeichen in einer Zeile wird durch die Ausgabe von n Leerzeichen realisiert. Bei unterschiedlichen Schriftbreiten entstehen wieder Probleme, die Texte spaltengerecht zu posi-

tionieren.
Setzen wir in einer Zeile zurück, so beginnt die Zählung der Spaltenposition wieder von vorn. Wir können also eine Zeile nacheinander und jeweils von vorn beginnend in den gewünschten Schriftarten beschreiben.

Die Ausgaben des @-Kommandos werden nicht durch SET PRINT auf den Drucker umgelenkt, sondern durch

SET DEVICE TO PRINTER

Außerdem wollen wir den Drucker am Anfang zurücksetzen. Wir sind damit sicher, daß die Zeilenzählung für den Drucker (Funktion PROW()) garantiert am Seitenanfang beginnt. Dadurch, daß wir die Zeile der ersten Ausgabe mit PROW() bestimmen, verhindern wir, daß ein Seitenvorschub am Anfang erfolgt. Will man wirklich Karteikarten drucken oder mit Einzelblättern arbeiten, so ist das Unterdrücken des Seitenvorschubs wichtig. Das Rücksetzen (Initialisieren) des Druckers wird durch die Steuerfolge

<Esc> "@" oder hexadezimal 1B 40

erreicht.

Wir wollen außerdem in Briefqualität (NLQ) drucken. Beim sequentiellen Druck der Karteikarte konnten wir NLQ vorher auf dem Drucker einstellen, brauchten also keine Steuerfolge dafür. Hier geht das nicht, da wir den Drucker aus dem Kommandofile heraus initialisieren. Damit wird auch die am Drucker eingestellte Briefqualität auf die Schriftart 'Pica' zurückgesetzt Die Steuerfolge

<Esc> "x" 1 oder hexadezimal 1B 78 01

ist für die Umschaltung auf NLQ-Qualität zuständig.

Druckt man durchgehende senkrechte Striche, so 'flattern' diese, wenn der Drucker vor- und rückwärts (bidirektional) druckt. Es ist deshalb der unidirektionale Druck einzuschalten:

<Esc> "U" 1 oder hexadezimal 1B 55 01

Das folgende Kommandofile löst die gestellte Aufgabe.

```
* Kommandofile 'karte2.prg'
* Drucken einer Karteikarte mit privater Druckersteuerung
* und Umrandung
SET TALK OFF
SET CONSOLE OFF
* Ausgabe des @-Kommandos auf dem Drucker
SET DEVICE TO PRINTER
* Drucker initialisieren
m_reset = CHR(27) + "@"
* Indexmodus ein, Schmalschrift ein
```

```
m_klein = CHR(27) + "S" + CHR(1) + CHR(15)
* Unidirektionaldruck ein
m_uni = CHR(27) + "U" + CHR(1)
* NLQ ein
m_nlq = CHR(27) + "x" + CHR(1)
* Indexmodus aus, Schmalschrift aus, d.h. auf NLQ zurück
m_normal = CHR(27) + "T" + CHR(18)
* Aktuelle Druckerzeile bestimmen
m_zeile = PROW()
@  m_zeile  ,0  SAY m_reset + m_uni + m_nlq
* Die Ausgabe kann nicht, wie in den nachfolgenden Zeilen,
* in Spalte 8 beginnen, da die ausgegebenen Steuerzeichen
* von dBASE wie normale Zeichen gezählt werden. Sie werden dann
* nicht mehr als Leerzeichen ausgegeben und müssen daher hier addiert
* werden.
@ m_zeile,8+2+3+3 SAY"╔═══════════════════════════════════════════════════════════╗"
m_zeile = m_zeile + 1
@  m_zeile  ,8  SAY "╟"
@  m_zeile  ,23 SAY "Karteikarte für Ersatzteile"
@  m_zeile  ,62 SAY "╟"
@  m_zeile+1,8  SAY "╟───────────────────────────────────────────────────────────╢"
@  m_zeile+2,8  SAY "╟"
@  m_zeile+2,62 SAY "╟"

* In dieser Zeile stört der Wechsel der Schriftarten nicht.
@  m_zeile+3,10 SAY m_klein + "Artikel:" + m_normal
@  m_zeile+3,17 SAY artikel
@  m_zeile+3,37 SAY m_klein + "Nächste Lieferung:" + m_normal
@  m_zeile+3,50 SAY DTOC(lieferung)
@  m_zeile+3,8  SAY "╟"
@  m_zeile+3,62 SAY "╟"
@  m_zeile+4,8  SAY "╟"
@  m_zeile+4,62 SAY "╟"

* Zuerst wird in Schmalschrift gedruckt, ...
@  m_zeile+5,37 SAY m_klein + "Firma:" + m_normal
* ... dann in Normalschrift.
@  m_zeile+5,8  SAY "╟"
@  m_zeile+5,50 SAY hersteller
@  m_zeile+5,62 SAY "╟"
@  m_zeile+6,8  SAY "╟"
@  m_zeile+6,62 SAY "╟"
@  m_zeile+7,37 SAY m_klein + "in:" + m_normal
@  m_zeile+7,8  SAY "╟"
@  m_zeile+7,50 SAY lageradr->ort
@  m_zeile+7,62 SAY "╟"
@  m_zeile+8,8  SAY "╟"
@  m_zeile+8,62 SAY "╟"
@  m_zeile+9,8  SAY "╚═══════════════════════════════════════════════════════════╝"
@  m_zeile+10,1 SAY m_reset + CHR(13)
SET DEVICE TO SCREEN
SET CONSOLE ON
```

```
SET TALK ON
CLOSE PRINT
RETURN
```

Die Abarbeitung liefert, wenn die Datenbankfiles 'lager.dbf' und 'lageradr.dbf' wie oben beschrieben in Benutzung sind, das Bild 17.4.

Selbstverständlich ist das nur ein Beispiel. Deshalb ist unsere Karteikarte so klein geraten. Jetzt müßte man sehen, welche Fähigkeiten der Drucker besitzt. Man kann nahezu beliebige Ausdrucke nach diesem Verfahren erzeugen. Es sollte aber nur dann angewendet werden, wenn wirklich wenige und kurze Steuerfolgen auszugeben sind. Sonst sollte man separate Programme schreiben, die den Drucker programmieren, und diese entweder nacheinander oder aus dBASE heraus aufrufen. Wir geben auch dafür je ein Beispiel an.

```
+-----------------------------------------------+
|          Karteikarte für Ersatzteile          |
|                                               |
| Artikel: Flachschieber          29.02.88      |
|                                 C&C           |
|                                 Rheinsberg    |
+-----------------------------------------------+
```

Bild 17.4. Im direkten Modus gedruckte Karteikarte

Wir wollen noch erklären, warum nur 'nahezu' alle Steuerfolgen ausgegeben werden können: CHR(0) liefert kein Zeichen an den Drucker! So hätten wir z.B. Schwierigkeiten, wollten wir auf unserer Karteikarte die Bezeichnungen der Inhalte, wie z.B. 'Artikel:', in Exponentenschreibweise ausgeben. Die Steuerfolge für das Einstellen der Exponentenschreibweise ist

<Esc> "S" 0 oder hexadezimal 1B 53 00

Es scheitert also an der Übertragung des Nullbytes an den Drucker. Hier ist gegenwärtig noch keine vernünftige Lösung in Sicht. Die nachfolgend beschriebene Technik kann aber auch für diesen Fall angewendet werden.

Die Definition eines privaten Zeichens für den Drucker in NLQ-Qualität ist eine Folge von mehr als 50 Byte! Wir nehmen an, daß mehrere derartige Zeichen in einem Programm 'privat.com' für den Drucker definiert und verfügbar gemacht werden. Bevor wir dBASE starten, schalten wir den Drucker ein und rufen, auf der Ebene des Betriebssystems, das Programm 'privat' auf. Geben wir aus dBASE heraus nicht selbst andere Steueranweisungen an den Drucker, so wird diese Voreinstellung für alle Ausgaben wirksam. dBASE selbst gibt

keine Daten an den Drucker zur Initialisierung aus. Wir ersparen uns somit lange und in dBASE unverständliche Ausgaben zur Programmierung des Druckers. Die Programme werden auf der Ebene des Betriebssystems nacheinander aufgerufen.

Wir können aber auch nach dem Start von dBASE das Kommando

```
RUN privat
```

eingeben. Das Programm 'privat' wird ausgeführt. Danach kehrt die Programmsteuerung zu dBASE zurück. Alle Einstellungen von dBASE sind so erhalten, wie sie vor dem RUN-Kommando vorlagen. Das verstehen wir unter dem Aufruf eines Programms aus dBASE heraus. In welcher Sprache das Programm 'privat' geschrieben ist, ist ohne Bedeutung.

Offenbar hat man so ein Werkzeug in der Hand, die Hardware möglichst weitgehend auszunutzen, die Details aber vor dem Nutzer dadurch zu verbergen, daß der Aufruf fremder Programme oder Kommandofiles in eine Nutzeroberfläche einbezogen wird. Der Endnutzer merkt nichts von der hardwarenahen Programmierung, die er benutzt.

Zum Schluß dieses Abschnitts möchten wir deutlich betonen, daß dBASE nicht dazu da ist, diffizile Aufgaben der hardwarenahen Programmierung zu lösen. Wenn aber durch die Ausgabe von wenigen Steuerbytes eine wesentliche Qualitätssteigerung möglich ist, dann kann dBASE auch dafür eingesetzt werden!

18. Alle Kommandos in alphabetischer Reihenfolge

In diesem Abschnitt besprechen wir die Kommandos von dBASE III in alphabetischer Reihenfolge. Er ist daher mehr zum Nachschlagen geeignet, läßt sich aber auch verwenden, um einen schnellen Überblick zu bekommen. Die vorangegangenen Abschnitte trugen mehr einführenden Charakter. So findet man grundlegende Ausführungen über die Kommandostruktur im Abschnitt 3.

Aus der alphabetischen Ordnung der Kommandos haben wir die Funktionen (Abschn. 4.7) und die SET-Kommandos (Abschn. 17.1) herausgenommen. Das ist sowohl inhaltlich als auch ihrem Umfang nach gerechtfertigt. Ferner sind solche Kommandos nur sehr kurz beschrieben, die in den vorangegangenen Abschnitten ausführlich besprochen wurden. In diesem Fall ist ein Verweis auf den entsprechenden Abschnitt angefügt.

Wir verzichten auf eine vollständige formale Beschreibung der Syntax der dBASE-Kommandos, da dadurch dem Nutzer der Zugang erschwert und der Sachverhalt für unsere Zwecke nicht klarer würde. Dennoch ist ein minimaler Aufwand für die Beschreibung von Zeichenketten, die gültige Kommandos von dBASE bilden, unumgänglich. Wir vereinbaren deshalb folgende Notationen:

<> Spitze Klammern schließen Symbole ein, die noch zu ersetzen sind, damit ein gültiges Kommando entsteht. Sie heißen Metasymbole. Sie sind inhaltlich durch das zu ersetzen, was in den spitzen Klammern steht. So ist <zeile> durch eine Zeilennummer zu ersetzen, <ausdr_liste> durch eine Liste von Ausdrücken usw. In den spitzen Klammern werden ggf. Abkürzungen verwendet. Ferner sind Begriffe in spitzen Klammern klein geschrieben, um Verwechslungen mit dem Text zu vermeiden.

[] Eckige Klammern schließen Teile ein, die wahlweise weggelassen oder aufgeführt werden können.

kennzeichnet eine Alternative, d.h., die vor und hinter dem "/" stehenden Begriffe können wahlweise benutzt werden. So besagt

FOR/WHILE <bedingung>

daß

FOR <bedingung>, z.B. FOR a < b,

und

WHILE <bedingung>, z.B. WHILE a < b,

gültige Ersetzungen sind.

{} beschreibt eine wahlweise Wiederholung des davorstehenden Begriffs. So besagt

{**CASE** <bedingung> <kommandofolge>...}

in der Definition des DO CASE-Kommandos, daß

CASE <bedingung> <kommandofolge>
beliebig oft wiederholt werden kann.

Schlüsselwörter von dBASE-Kommandos werden hier, zur besseren Unterscheidung von anderen Begriffen, fett gedruckt und in großen Buchstaben geschrieben.

?/??

Syntax: ?/?? [<ausdr_liste>]

Die Ausdrucksliste wird berechnet und auf dem Bildschirm ausgegeben. ? bewirkt einen Zeilenwechsel vor der Ausgabe des ersten Wertes (Abschn. 13.1).
Siehe auch: SET CONSOLE, SET PRINT, SET TALK.

@

Syntax: **@** <zeile>,<spalte> [**SAY** <ausdr> [**PICTURE** <klausel>]]
[[**GET** <var> [**PICTURE** <klausel>]]
[**RANGE** <ausdr>,<ausdr>]] [**CLEAR**]

Tafel 18.1. Formatfunktionen

Funktionszeichen	Wirkt bei der Eingabe	Wirkt bei der Ausgabe	Wirkt auf die Typen N	C	D	Bedeutung
A	x					nur Buchstaben
B						linksbündig
C			x			CR wird nach positiven Werten ausgegeben (credit)
D				x		deutsches Datumsformat
E	x					engl. Datumsformat
R				x		Zeichen im Format, die keine Formatzeichen sind, werden nicht übernommen
X						negative Zahlen werden ohne Vorzeichen, dfür mit nachfolgendem DB (debit) ausgegeben
Z						der Wert 0 wird als Leerzeichen ausgegeben
(						negative Werte werden in runde Klammern eingeschlossen
						kleine Buchstaben werden in große Buchstaben umgewandelt

Es erfolgt eine Ausgabe auf dem Bildschirm ab der angegebenen

Zeile und Spalte. Ausgegeben wird der Ausdruck nach SAY in dem Format, welches durch das Format des Wertes und die nachfolgende PICTURE-Klausel bestimmt wird. Für die nach GET stehende Variable können Werte eingegeben werden, die durch die nachfolgende PICTURE-Klausel überprüft werden. Auch hier beeinflußt das Format der zu lesenden Variablen die Darstellung der Eingabebereiche. RANGE bestimmt den Bereich, in dem der eingegebene Wert liegen darf (Abschn. 13.2).

Bezüglich der Formatzeichen, die in einer PICTURE-Klausel auftreten können, besitzt dBASE III wesentlich erweiterte Möglichkeiten gegenüber dBASE II. Man unterscheidet Formatfunktionen (functions) und Formatzeichen (template symbols). Beide Gruppen unterscheiden sich in ihrer Wirkung bei der Ein- oder Ausgabe und in ihrer Wirkung auf verschiedene Typen. Betrachten wir zuerst in Tafel 18.1 die Formatfunktionen. Die Formatzeichen sind in Tafel 18.2 zusammengefaßt.

Klauseln werden in Zeichenkettenbegrenzer eingeschlossen. Als Klauseln lassen sich Folgen von Formatzeichen wie in dBASE II angeben. Außerdem kann eine Klausel auch eine Formatfunktion oder eine Folge von Formatfunktionen sein. Eine Formatfunktion wird durch ein at-Zeichen '@' eingeleitet. Danach folgen ohne Leerzeichen die Formatfunktionen. So enthält die Klausel '@BCX' die Formatfunktionen 'B', 'C' und 'X'. Die Wirkung besteht darin, daß Zahlen linksbündig ausgegeben werden. Positiven Werten werden die Buchstaben CD (für Guthaben, credit) nachgestellt. Negative Werte werden ohne Vorzeichen und mit nachgestelltem DB (für Schuld, debit) ausgegeben.

In einer Klausel können Formatfunktionen und Formatzeichen auftreten. Die Klausel beginnt dann mit den Formatfunktionen, gefolgt von Formatzeichen. Beide Teile werden durch ein Leerzeichen voneinander getrennt. Wir erklären das an einigen Beispielen. Dazu betrachten wir das folgende Kommandofile:

```
* Kommandofile 'formate.prg'
* Wir definieren einige Speichervariablen
c = "Ab1C-de"
n1 = -12.34
n2 = 123456.789
d = DATE()
SET DELIMITER ON
* Löschen sicherheitshalber alle hängenden GETS
CLEAR GETS
* ... und löschen den Bildschirm
@ 0,0 CLEAR

* Test der Ausgabeformate
@ ROW()+1,10 SAY 'Test der Ausgabeformate'
* N wirkt nicht bei der Ausgabe
@ ROW()+1,10 SAY c  PICTURE 'NNNNNNN'
* Diese Formate sind sinnvoll
@ ROW()+1,10 SAY n1 PICTURE '99999.999'
@ ROW()+1,10 SAY n1 PICTURE '**,***.**'
@ ROW()+1,10 SAY n2 PICTURE '999,999.99'
@ ROW()+2,10 SAY 'Test der Ausgabefunktionen'
```

```
* Test der Ausgabefunktionen
* '!' wirkt, Formatzeichen A aber nicht, denn Ziffern
* werden auch ausgegeben
@ ROW()+1,10 SAY c  PICTURE '@! AAAAAAA'
@ ROW()+1,10 SAY n1 PICTURE '@CX 999999.9'
@ ROW()+2,10 SAY 'Test der Eingabeformate'

* Test der Eingabeformate
* Hier tritt die erwartete Wirkung ein
@ ROW()+1,10 SAY "->" GET c  PICTURE '!!!!!!!!'
* Der alte Wert von n1 wird im Eingabebereich angeboten.
* Sterne werden statt Leerzeichen vorangestellt
@ ROW()+1,10 SAY "->" GET n1 PICTURE '******.*'
* Eine vierstellige Jahresangabe kann so nicht erreicht werden
@ ROW()+1,10 SAY "->" GET d  PICTURE 'tt.mm.jjjj'
@ ROW()+2,10 SAY 'Test der Eingabefunktionen'

* Test der Eingabefunktionen
* Diese Klausel ist widersprüchlich.
@ ROW()+1,10 SAY "->" GET c  PICTURE '@! NNNNN.NN'
* Das Datum wird im deutschen Format angeboten und eingelesen
@ ROW()+1,10 SAY "->" GET d  PICTURE '@D mm.dd.yy'
* Das Datum wird im englischen Format angeboten und eingelesen
@ ROW()+1,10 SAY "->" GET d  PICTURE '@E 99.99.99'
READ
RETURN
* Ende des Kommandofiles 'formate.prg'
```

Die Abarbeitung dieses Kommandofiles zeigt vor der Eingabe das folgende Bild:

```
Test der Ausgabeformate
Ab1C-de
  -12.34
***-12.34
123,456.78

Test der Ausgabefunktionen
AB1C-DE
    12.3 DB

Test der Eingabeformate
-> :AB1C-DE:
-> :***-12.3:
-> :20.02.88:

Test der Eingabefunktionen
-> :AB1C-DE:
-> :20.02.88:
-> :02.20.88:
```

Der Kursor steht auf dem ersten Zeichen des ersten Eingabefeldes.

Wir geben jetzt Werte ein. Nachfolgend ist der Bildschirminhalt der Eingabebereiche nach der Eingabe angegeben, wobei in den Zeilen auf der rechten Seite die Kommentare nachträglich ergänzt wurden:

```
Test der Eingabeformate
-> :DBASE 2:              kleine Buchstaben werden umgewandelt
-> :*****2.7:
-> :10.11.88:             die Jahreszahl ist nur zweistellig

Test der Eingabefunktionen
-> :DBASE 3:              die Formatzeichen 'N' wirken nicht
-> :23.11.88:             nur deutsche Datumsform möglich
-> :01.06.88:             nur engl. Datumsform möglich
```

Die Werte der letzten beiden Eingabebereiche werden einer Variablen vom Typ Datum zugewiesen. Dadurch wird die Einhaltung der Datumsgrenzen überprüft. Würden, bei gleichen Formaten (Klauseln), die Werte einer Stringvariablen zugewiesen werden, so erfolgt diese Überprüfung nicht.

Tafel 18.2. Formatzeichen

Formatzeichen	Wirkt bei der Eingabe	Wirkt bei der Ausgabe	Bedeutung
9			für String dürfen hier nur Ziffern stehen für numerische Größen sind Ziffern und Vorzeichen erlaubt
#			nur Ziffern, Vorzeichen und Leerzeichen
A			nur Buchstaben, einschl. Umlaute, außer ß
L			nur logische Werte
N	x		nur Ziffern und Buchstaben, einschl. Umlaute und ß
X	x		alle Zeichen
!	x		alle Zeichen zugelassen; kleine Buchstaben werden in große umgeformt
$			bei der Ausgabe numerischer Werte werden für führende Nullen $-Zeichen gedruckt Eingabe: $-Zeichen werden angezeigt
*			bei der Ausgabe numerischer Werte werden für führende Nullen Sterne gedruckt Eingabe: Sterne werden angezeigt Dezimalpunkt

dBASE überprüft nicht, ob Formatfunktionen, Formatzeichen und die Werte eine sinnvolle Kombination bilden. Vorrangregeln sind nicht erklärt.

Formate können auch Zeichen enthalten, die keine Formatzeichen sind. Je nach Formatzeichen und Typ des Wertes wirken sie unterschiedlich. In den obigen Beispielen enthalten die Formate '**,***.**', '999,999.99' und 'tt.mm.jjjj' z.B. solche Zeichen.
Siehe auch: CLEAR GETS, SET DEVICE, SET TALK.

ACCEPT

Syntax: **ACCEPT** [<prompt>] **TO** <num_var>

ACCEPT bewirkt die Eingabe einer Zeichenkette von der Tastatur in eine Speichervariable. Die Anforderungszeichenkette (Prompt) wird vorher ausgegeben (Abschn. 13.1). Der Unterschied zu INPUT besteht darin, daß einzugebende Zeichenketten nicht in Begrenzer (', " oder []) eingeschlossen werden. ACCEPT behandelt auch eingegebene Zahlen als Zeichenketten.
Siehe auch: @, INPUT, WAIT.

APPEND

Syntax: **APPEND [BLANK]**
APPEND FROM <filename> [**FOR/WHILE** <bedingung>]
[**SDF/DELIMITED [WITH BLANK/**<zeichen>]]

Alle Formen des APPEND-Kommandos dienen dem Anfügen von Datensätzen an das aktive Datenbankfile (Abschn. 6.1).

APPEND bewirkt den Übergang in den Direktmodus. Die Daten werden von der Tastatur eingegeben.

APPEND BLANK fügt einen leeren Datensatz an das aktive Datenbankfile an. Das ist dann sinnvoll, wenn die einzutragenden Daten über private Bildschirmmasken eingelesen und erst in Speichervariablen zwischengespeichert werden.

APPEND FROM fügt Daten aus einem File an. Fehlt SDF, so muß das anzufügende File ein Datenbankfile sein. Ist SDF angegeben, so ist es ein Textfile. Ist DELIMITED angegeben, so ist es ein Textfile, in dem die einzelnen Felder durch Komma voneinander getrennt sind (Standard). Durch WITH BLANK wird ein Leerzeichen als Trennzeichen zwischen Feldern definiert und durch WITH <zeichen> das angegebene Zeichen.
Siehe auch: EDIT, SET CARRY, SET FORMAT.

ASSIST

Syntax: **ASSIST**

ASSIST ist eine menügesteuerte Programmierunterstützung (Abschnitt 2.10). Es sind nicht alle Kommandos und nicht alle möglichen syntaktischen Formen in ASSIST enthalten.

AVERAGE

Syntax: **AVERAGE** <ausdr_liste> [<gb>] [**FOR/WHILE** <bedingung>]
[**TO** <mem_var_liste>]

AVERAGE berechnet den Mittelwert aller numerischen Felder des aktiven Datenbankfiles über alle Sätze. Ist eine Ausdrucksliste ange-

geben, so wird das arithmetische Mittel nur über diese Ausdrücke gebildet. Die so berechneten Mittelwerte können einer Liste von Speichervariablen zugewiesen werden. Gültigkeitsbereich <gb> und die Auswahlbedingung schränken die Wirkung von AVERAGE auf die Sätze ein, die im Gültigkeitsbereich liegen und die Bedingung erfüllen.

AVERAGE ist seinem Charakter nach eigentlich eine Funktion, wird in dBASE aber als Kommando betrachtet. Ähnliches gilt für SUM und COUNT. Es ist nicht möglich, den Mittelwert über einer Anzahl von Speichervariablen oder anderen Werten zu berechnen, die nicht in einem Datenbankfile stehen.

Beispiel: Wir benutzen unser Datenbankfile 'lager' und bilden den mittleren Preis über alle Ersatzteile und über die Gesamtwerte der Ersatzteile, die am Lager sind.

```
. USE lager
. AVERAGE preis, preis*anzahl
     9 Sätze gemittelt
  preis  preis*anzahl
  79.04       1138.75
```

Siehe auch: COUNT, SUM.

BROWSE

Syntax: **BROWSE** [**FIELDS** <feldliste>]

BROWSE dient der Schnellkorrektur der Feldinhalte des aktiven Datenbankfiles im Direktmodus (Abschn. 6.4). Ist eine Feldliste angegeben, so werden nur die angegebenen Felder zur Korrektur angeboten.
Siehe auch: APPEND, CHANGE, EDIT.

CANCEL

Syntax: **CANCEL**

CANCEL bricht die Ausführung eines Kommandofiles ab und fordert das nächste Kommando von der Tastatur an. Dabei werden alle eröffneten Kommandofiles abgeschlossen.
Siehe auch: CLOSE, RETURN.

CHANGE

Syntax: **CHANGE** [<gb>] [**FIELDS** <feldliste>] [**FOR/WHILE** <bedingung>]

CHANGE gestattet die Korrektur der Feldinhalte im Direktmodus (Abschn. 6.5). Es werden nur die Felder zur Korrektur angeboten, die in der Feldliste aufgeführt sind. Fehlt die Feldliste, so werden alle Felder angeboten. Die Reihenfolge der Felder auf dem Bildschirm wird durch die Reihenfolge in der Feldliste bestimmt.
Siehe auch: APPEND, BROWSE, EDIT, SET FORMAT.

CLEAR

Syntax: **CLEAR**
CLEAR ALL
CLEAR GETS
CLEAR MEMORY

CLEAR löscht den Bildschirm und positioniert den Kursor in die linke obere Ecke (Position 0,0). In Verbindung mit dem @-Kommando kann auch nur ein Teil des Bildschirms gelöscht werden.

CLEAR ALL schließt alle eröffneten Datenbankfiles ab, gibt alle Speichervariablen frei (vernichtet sie) und wählt den Arbeitsbereich 1 (A) als aktiven Arbeitsbereich. Sind mit dem Datenbankfile Indexfiles, Formatfiles oder ein .dbt-File (für Memo-Felder) eröffnet, so werden auch diese durch CLEAR ALL abgeschlossen. Ein eröffnetes Prozedurfile wird durch CLEAR ALL nicht abgeschlossen. Mit CLEAR ALL kann dBASE, auch aus einer Prozedur heraus, in einen definierten Anfangszustand versetzt werden. Die Voreinstellungen aus dem Konfigurationsfile 'config.db' (Abschn. 17.2) werden dadurch weder hergestellt noch beeinflußt.

CLEAR GETS löscht alle Eingabeanforderungen, die über GET angefordert, aber noch nicht befriedigt wurden (hängende, pending GETS). Es dient der Initialisierung bei der Eingabe im Direktmodus. Die Datenerfassung im Direktmodus sollte so erfolgen, daß vor dem Aufbau einer Bildschirmmaske evtl. hängende GETS durch CLEAR GETS gelöscht werden. CLEAR GETS löscht nicht die Werte, die eingegeben wurden oder werden.

CLEAR MEMORY vernichtet alle Speichervariablen. Es ist gleichwertig mit RELEASE ALL.
Siehe auch: @, CLOSE, RELEASE.

CLOSE

Syntax: **CLOSE** <filetyp>

CLOSE schließt alle eröffneten Files des angegebenen Typs. Gültige Schlüsselwörter für Filetypen sind

ALTERNATE	- schließt ein Alternate-File
DATABASES	- schließt alle Datenbankfiles einschließlich .dbt-Files, Indexfiles und Formatfiles
FORMAT	- schließt das offene Formatfile im aktiven Arbeitsbereich
INDEX	- schließt alle Indexfiles im aktiven Arbeitsbereich
PROCEDURE	- schließt das offene Prozedurfile

Beispiel: Ein Prozedurfile 'myfile.prg' mit mehreren Prozeduren sei eröffnet (SET PROCEDURE TO myfile). Beim Test haben sich Fehler herausgestellt, so daß 'myfile' korrigiert werden muß. Bevor wir die

Korrektur im Textfile durchführen können (MODIFY COMMAND myfile), muß das Prozedurfile abgeschlossen werden (CLOSE PROCEDURE). Andernfalls kann 'myfile.prg' nicht zum Editieren eröffnet werden, da es schon eröffnet ist, nämlich als Prozedurfile.
Siehe auch: CLEAR, SET ALTERNATE, SET FORMAT, SET INDEX, SET PROCEDURE, USE.

CONTINUE

Syntax: **CONTINUE**

Vor der Ausführung eines CONTINUE-Kommandos muß ein LOCATE abgearbeitet worden sein. CONTINUE wirkt wie ein LOCATE - mit der einzigen Ausnahme, daß die Suche mit dem Datensatz beginnt, der in logischer Reihenfolge dem aktuellen Datensatz folgt. CONTINUE übernimmt den Gültigkeitsbereich und die FOR-Bedingung des zuletzt ausgeführten LOCATE-Kommandos und positioniert den Dateizeiger auf den ersten Datensatz in logischer Reihenfolge, der die FOR-Bedingung erfüllt (Abschn. 9.2). Die logische Reihenfolge ist hier hervorgehoben, da LOCATE und CONTINUE auch auf Datenbanken angewendet werden können, die mit Indexfiles in Benutzung sind. Die Suche erfolgt dann zwar nicht über das Indexfile, wie bei FIND und SEEK, das Indexfile bestimmt aber die logische Reihenfolge der Sätze.
Siehe auch: FIND, LOCATE, SEEK, SET INDEX, USE.

COPY

Syntax:
```
COPY FILE <quell_filename> TO <ziel_filename>
COPY STRUCTURE EXTENDED
COPY STRUCTURE TO <filename> [FIELDS <feldliste>]
COPY TO <filename> [<gb>] [FIELDS <feldliste>]
   [FOR/WHILE <bedingung>]
   [SDF/DELIMITED [WITH BLANK/<begrenzer>]]
```

COPY FILE kopiert ein beliebiges File. Dieses File darf nicht eröffnet sein. Beiden Filenamen können ein Pfad und eine Gerätebezeichnung vorangestellt sein.

COPY STRUCTURE EXTENDED erzeugt ein Datenbankfile, welches vier Felder enthält. Jedes Feld entspricht einer Angabe aus der Strukturdefinition eines Datenbankfiles. Es werden so viele Sätze in diesem Zielfile erzeugt, wie dieses Datenbankfile Felder enthält. Jeder Datensatz enthält die Angaben eines Feldes. Wir zeigen das an unserem Datenbankfile 'lager.dbf'.

```
. USE lager
. COPY STRUCTURE EXTENDED TO lagerex
* Wir besichtigen das Resultat
. USE lagerex
```

```
. LIST STRUCTURE
Datenbankstruktur       - C:lagerex.dbf
Anzahl der Datensätze   -        7
Letztes Änderungsdatum - 25.01.88
Feld   Feldname    Typ         Länge    Dez
    1  FIELD_NAME  Zeichen       10
    2  FIELD_TYPE  Zeichen        1
    3  FIELD_LEN   Numerisch      3
    4  FIELD_DEC   Numerisch      3
** Gesamt **                     18
. LIST
 Satz #  FIELD_NAME FIELD_TYPE FIELD_LEN FIELD_DEC
      1  ARTIKEL    C                15         0
      2  ANZAHL     N                 4         0
      3  PREIS      N                 8         2
      4  MINIMUM    N                 2         0
      5  LIEFERUNG  D                 8         0
      6  HERSTELLER C                12         0
      7  BEM        M                10         0
```

Solche Strukturfiles können im CREATE-Kommando verwendet werden.

COPY STRUCTURE TO erstellt ein neues Datenbankfile, welches die gleiche Struktur hat wie das aktive Datenbankfile, aber keine Sätze enthält. Dieses Kommando hat im Vergleich zu dBASE II an Bedeutung verloren, da MODIFY STRUCTURE in dBASE III die alten Datensätze in das neue Datenbankfile übernimmt.

COPY TO kopiert alle Felder des aktiven Datenbankfiles in ein neues File. Ist eine Feldliste angegeben, so werden nur die aufgeführten Felder kopiert. Die Struktur des Zielfiles ergibt sich aus der Struktur der Felder in der Feldliste einschließlich ihrer Reihenfolge. Die Standardannahme für den Gültigkeitsbereich <gb> ist ALL. Die Auswahlbedinguung bewirkt, daß nur die Sätze kopiert werden, für die die Bedingung erfüllt ist. Fehlt SDF, so ist das Zielfile ein Datenbankfile. Ist SDF angegeben, so wird ein Textfile erzeugt, in dem die Feldinhalte spaltengerecht untereinander stehen. DELIMITED erzeugt ein Textfile, in dem die Feldinhalte durch ein Komma (Standard), ein Leerzeichen (BLANK) oder durch einen beliebigen anderen Begrenzer (<zeichen>) voneinander getrennt sind. Enthalten die Zeilen solcher Textfiles Adressen, so können sie z.B. unter WordStar mit MailMerge günstig weiterverwendet werden.

COPY TO ... SDF ist der am meisten genutzte Weg, um Daten von dBASE in andere Systeme zu transportieren.

COPY ist die einzige Möglichkeit, Memo-Files zu komprimieren!

Siehe auch: APPEND, CREATE, SET DELETED, SET SAFETY.

COUNT

Syntax: **COUNT** [<gb>] [**FOR/WHILE** <bedingung>] [**TO** <mem_var>]

COUNT zählt die Sätze im angegebenen Gültigkeitsbereich (Standard ist ALL), die der Auswahlbedingung genügen, und speichert diesen Wert in einer Speichervariablen.

Beispiel: Will man die Anzahl der Sätze eines Datenbankfiles ermitteln, so liefert COUNT (ohne weitere Angaben) den gewünschten Wert. Ist das Datenbankfile ohne Indexfile (!) in Benutzung, so liefert die Kommandofolge

```
GO BOTTOM
? RECNO()
```

das gleiche Resultat wie COUNT und ist schneller.

CREATE

Syntax: **CREATE** <filename> [**FROM** <strukturfile>]
CREATE LABEL <filename>
CREATE REPORT <filename>

CREATE leitet die Definition der Struktur eines Datenbankfiles im Direktmodus ein. Ist die Definition vollständig und beendet, so können nach einer Anfrage Daten eingegeben werden (Abschn. 5).

Wurde ein Strukturfile angegeben, so muß dieses den gleichen Aufbau haben wie ein Datenbankfile, welches durch COPY STRUCTURE EXTENDED erzeugt wird. Diese Variante gibt uns die Möglichkeit, ein neues Datenbankfile einer bestimmten Art zu erzeugen, ohne in den Direktmodus überzugehen und den Nutzer mit der Strukturdefinition zu belästigen. Für die Nutzung von dBASE von einer Nutzeroberfläche aus ist das wichtig.

CREATE LABEL leitet die Definition der Struktur und des Inhalts eines Formatfiles (.lbl) für Briefadressen ein (Abschn. 16). Möchte man ein .lbl-File erstellen, so kann dafür auch MODIFY LABEL verwendet werden.

CREATE REPORT leitet die Definition von Struktur und Inhalt eines Formatfiles (.frm) für Standardberichte ein (Abschn. 10). Möchte man ein .frm-File erstellen, so kann dafür auch MODIFY REPORT verwendet werden.

Siehe auch: COPY, LABEL, MODIFY LABEL, MODIFY REPORT, MODIFY STRUCTURE, REPORT.

DELETE

Syntax: **DELETE** [<gb>] [**FOR/WHILE** <bedingung>]

DELETE markiert die durch den Gültigkeitsbereich und die Auswahlbedingung bestimmten Sätze als gestrichen. So markierte Sätze werden

bei der sequentiellen Ausgabe durch einen Stern ('*') vor dem ersten Feld gekennzeichnet. Im Direktmodus (BROWSE, EDIT) erscheint in der obersten Bildschirmzeile '*Gelöscht*'. ^U schaltet im Direktmodus die Löschmarkierung ein und aus. Die Standardannahme für den Gültigkeitsbereich ist der aktuelle Satz. Die Löschmarkierung wird im Datensatz selbst vermerkt (Abschn. 19.1), ist also unabhängig von der Benutzung der Datenbank.

Gilt SET DELETED ON (Standard ist OFF), so werden als gestrichen markierte Sätze nicht in Operationen einbezogen.

Als gestrichen markierte Sätze werden durch PACK physisch vernichtet.

Siehe auch: BROWSE, DELETED(), EDIT, PACK, RECALL, SET DELETED.

DIR

Syntax: **DIR** [<gerät>:] [<pfad>\] [<filegruppe>]

DIR zeigt die ausgewählten Filenamen auf dem Bildschirm an.

DISPLAY

Syntax: **DISPLAY** [<gb>] [<ausdr_liste>] [**FOR/WHILE** <bedingung>]
[**OFF**] [**TO PRINT**]

DISPLAY MEMORY [**TO PRINT**]
DISPLAY STATUS [**TO PRINT**]
DISPLAY STRUCTURE [**TO PRINT**]

Alle Varianten des DISPLAY-Kommandos haben die gleiche Wirkung wie die entsprechenden LIST-Kommandos - mit zwei Unterschieden: Die Standardannahme für den Gültigkeitsbereich ist der aktuelle Datensatz, und DISPLAY stoppt die Ausgabe auf dem Bildschirm nach jeweils 20 Zeilen. Die Ausgabe kann durch Drücken einer Taste fortgesetzt werden (Abschn. 3).

DO

Syntax: **DO** <filename> [**WITH** <parameter_liste>]
DO <proc_name> [**WITH** <parameter_liste>]

DO ruft ein Kommandofile oder eine Prozedur zur Abarbeitung auf. Durch RETURN in der aufgerufenen Kommandofolge kann die Programmsteuerung wieder zum Kommando nach DO zurückgegeben werden.

Wird ein Kommandofile aufgerufen, so wird es eröffnet. Wird eine Prozedur aufgerufen, so muß das Prozedurfile, in dem sie steht, vorher durch SET PROCEDURE TO eröffnet worden sein.

Wird ein Kommandofile verlassen, so wird es dadurch abgeschlossen. Ein Prozedurfile bleibt eröffnet, bis es durch CLOSE PROCEDURE abgeschlossen wird (Abschn. 11). CLEAR ALL läßt ein Kommandofile eröffnet.

Siehe auch: CLEAR ALL, PARAMETERS, PRIVATE, PROCEDURE, PUBLIC, SET PROCEDURE.

DO CASE

Syntax: **DO CASE**
CASE <bedingung> <kommandofolge>
{[**CASE** <bedingung> <kommandofolge>]...}
[**OTHERWISE** <kommandofolge>]
ENDCASE

DO CASE ist eine strukturierte Anweisung zur Übergabe der Programmsteuerung an den ersten Fall, dessen Bedingung erfüllt ist (TRUE liefert). DO CASE ist nur im Kommandofile sinnvoll und spart dort verschachtelte IF-Kommandos (Abschn. 11).
Siehe auch: IF.

DO WHILE

Syntax: **DO WHILE** <bedingung>
[<kommandofolge>]
[**LOOP/EXIT**]
[<kommandofolge>]
ENDDO

DO WHILE ist eine strukturierte Anweisung zur zyklischen Abarbeitung einer Kommandofolge. Die Bedingung hinter DO WHILE wird ausgewertet. Ist sie erfüllt, so wird die Anweisungsfolge abgearbeitet. Wird ENDDO erreicht, so kehrt die Programmsteuerung zur letzten abgearbeiteten DO WHILE-Zeile zurück. Der Ausdruck wird erneut berechnet usw. (Abschn. 11). DO WHILE ist nur im Kommandofile sinnvoll.
Die Kommandofolge kann beliebig viele LOOP- und EXIT-Kommandos enthalten.
Siehe auch: LOOP, EXIT.

EDIT

Syntax: **EDIT** [[**RECORD**] <n>]

EDIT stellt die Daten des ausgewählten Satzes zur Korrektur im Direktmodus bereit. Wird nur EDIT eingegeben, so wird der aktuelle Satz angeboten (Abschn. 6.3).
Siehe auch: APPEND, BROWSE, CHANGE, REPLACE, SET FORMAT TO.

EJECT

Syntax: **EJECT**

EJECT bewirkt einen Vorschub des Papiers auf dem Drucker zum

Anfang der neuen Seite. An den Drucker wird das Byte 0Ch (ff form feed) ausgegeben. Wird EJECT ausgegeben, so wartet dBASE so lange, bis ein Drucker angeschlossen und bereit ist. Auch Escape <Esc> hilft hier nicht! Ist kein Drucker verfügbar, so ist das System damit blockiert. Ein Neustart des Systems kann zum Datenverlust in den eröffneten Files führen! Der Ausführung eines EJECT-Kommandos in einem Kommandofile sollte deshalb immer eine Anfrage an den Nutzer vorausgehen.

ERASE

Syntax: **ERASE** <filename>

ERASE löscht das angegebene File. Es können nur solche Files gelöscht werden, die nicht eröffnet sind.

EXIT

Syntax: **EXIT**

EXIT kann nur innerhalb eines DO WHILE-Zyklus ausgeführt werden. Es bewirkt die übergabe der Programmsteuerung an das Kommando, das im Kommandofile auf das nächste ENDDO folgt. Anders ausgedrückt: Es erfolgt ein Sprung zu dem Kommando, welches auf den DO WHILE-Zyklus folgt (Abschn. 11).
Siehe auch: DO WHILE, LOOP.

FIND

Syntax: **FIND** <string_ausdr>

FIND sucht in einem Datenbankfile, welches indiziert in Benutzung ist, den ersten Datensatz in logischer Folge, dessen Indexschlüssel gleich dem Wert des Stringausdrucks ist (Abschn. 9.2).
Siehe auch: CLEAR ALL, CLOSE INDEX, INDEX, LOCATE, SEEK, SET INDEX, USE.

GO/GOTO

Syntax: **GO/GOTO BOTTOM/TOP/**<ausdr>

Der Dateizeiger wird auf den angegebenen Satz positioniert (Abschn. 3.9).
Siehe auch: FIND, LOCATE, SEEK, USE.

HELP

Syntax: **HELP** [<schlüsselwort>]

Es erscheinen Textinformationen zu dem ausgewählten Schlüsselwort bzw. zur Arbeit mit dBASE (Abschn. 2.10).

IF

Syntax:
```
IF <bedingung>
     <kommandofolge_1>
[ELSE
     <kommandofolge_2>]
ENDIF
```

IF bewirkt eine Verzweigung der Programmsteuerung in einem Kommandofile. Ist die Bedingung erfüllt, so wird die Kommandofolge 1 abgearbeitet, sonst Kommandofolge 2 (Abschn. 11).
Siehe auch: DO CASE.

INDEX

Syntax: **INDEX ON** <ausdr> **TO** <filename>

INDEX indiziert das aktive Datenbankfile nach dem angegebenen Ausdruck. Es entsteht ein Indexfile (.ndx-File). Nach dem Indizieren ist das Datenbankfile mit diesem Indexfile aktiv (Abschn. 8.2).
Siehe auch: CLOSE, FIND, REINDEX, SEEK, SET INDEX, SET UNIQUE, USE.

INPUT

Syntax: **INPUT** [<prompt>] **TO** <mem_var>

INPUT bewirkt die Eingabe eines Wertes von der Tastatur in eine Speichervariable. Die Anforderungszeichenkette (Prompt) wird vorher ausgegeben (Abschn. 13.1). Der Unterschied zu ACCEPT besteht darin, daß mit INPUT auch numerische und logische Werte eingegeben werden können. Zeichenketten müssen in Begrenzer (', " oder []) eingeschlossen werden. Wir empfehlen, die logischen Konstanten .T., .t., .F. oder .f. zu benutzen, da .Y., .y., und .J., .j. von den Implementationen von dBASE nicht einheitlich behandelt werden.
Siehe auch: @ ACCEPT, WAIT.

INSERT

Syntax: **INSERT [BLANK] [BEFORE]**

INSERT fügt einen neuen Datensatz nach oder vor (BEFORE) dem aktuellen Datensatz ein. Fehlt BLANK, so wird die Eingabemaske im

Direktmodus zur Erfassung der Daten dieses Satzes angeboten. Ist BLANK angegeben, so wird ein leerer Datensatz eingefügt. Es erfolgt kein Übergang in den Direktmodus. Der Dateizeiger zeigt auf diesen neuen Satz. Durch REPLACE können jetzt Daten in die Felder eingetragen werden. Diese Variante ist in Kommandodateien vorteilhaft einsetzbar.

INSERT sollte benutzt werden, wenn ein File sortiert vorliegt und diese Ordnung durch die Hinzunahme weiterer, meist weniger Sätze nicht gestört werden soll.

Siehe auch: @, APPEND, BROWSE, CHANGE, EDIT, REPLACE, SET FORMAT.

JOIN

Syntax: **JOIN WITH** <alias> **TO** <filename> **FOR** <bedingung>
[**FIELDS** <feldliste>]

JOIN vereinigt die angegebenen Felder von zwei Datenbankfiles, das aktive und das durch den Aliasnamen angegebene, zu einem neuen Datenbankfile <filename> (Abschn. 15.2).

LABEL

Syntax: **LABEL FORM** <filename> [**SAMPLE**] [<gb>]
[**FOR /WHILE** <bedingung>]
[**TO PRINT**] [**TO FILE** <filename>]

LABEL druckt Briefadressen entsprechend dem Format, wie es im angegebenen Formatfile (FORM <filename>) definiert ist (Abschn. 16). Formatfiles für Briefadressen (.lbl-Files) werden durch CREATE LABEL oder durch MODIFY LABEL erstellt.

Siehe auch: CREATE LABEL, MODIFY LABEL.

LIST

Syntax: **LIST** [**OFF**] [<gb>] [**FOR/WHILE** <bedingung>] [<ausdr_liste>]
LIST MEMORY [**TO PRINT**]
LIST STATUS [**TO PRINT**]
LIST STRUCTURE [**TO PRINT**]

LIST gibt die Werte der berechneten Ausdrücke für alle ausgewählten Sätze auf dem Bildschirm aus (Abschn. 3). OFF unterdrückt die Ausgabe der Satznummer. SET HEADING OFF (Standard ist ON) unterdrückt die Ausgabe einer Kopfzeile, die Überschriften für die Ausdrücke enthält. ALL ist die Standardannahme für den Gültigkeitsbereich.

LIST MEMORY gibt einen Überblick über die definierten Speichervariablen. Die in Files (.mem) ausgelagerten Speichervariablen sind dabei ohne Bedeutung.

1. Beispiel: Wir weisen einigen Speichervariablen Werte zu.

```
. m_zeit = TIME()
11:16:56
. m_datum = DATE()
26.01.88
. m_e = 2.7182818
2.7182818
* Welche Struktur hat der Memory-Bereich?
. LIST MEMORY
M_ZEIT      lokal (priv)  C  "11:16:56"
M_DATUM     lokal (priv)  D  26.01.88
M_E         lokal (priv)  N          2.7182818  (         2.71828180)
    3 Variable definiert,      28 BYTES benutzt
  253 Variable verfügbar,   5972 BYTES verfügbar
```

LIST STATUS gibt die Stellung ausgewählter Optionen an, die über SET oder 'config.db' eingestellt werden können.

2. Beispiel: Wir eröffnen zwei Datenbankfiles und verbinden diese logisch.

```
. USE lager
. SELECT 2
* Es wird ein Aliasname gewählt, der nicht mit dem Namen des
* Datenbankfiles übereinstimmt
   USE lageradr INDEX laadrh ALIAS adressen
. SELECT lager
* Die Datenbankfiles werden logisch verbunden
. SET RELATION TO hersteller INTO adressen
* Ein Prozedurfile wird eröffnet
. SET PROCEDURE TO proc
. LIST STATUS
Selektierte Datenbank
Selektierter Bereich : 1. Datenbank eröffnet - C:lager.dbf  ALIAS - LAGER
      Memo Datei     - C:lager.dbt
 verknüpft mit: ADRESSEN
    Relation: hersteller

Selektierter Bereich : 2. Datenbank eröffnet - C:lageradr.dbf  ALIAS - ADRESSEN
      Indexdatei     - C:laadrh.ndx  Schlüssel - hersteller


Protokolldatei       - C:mem.alt
Prozedurdatei        - C:proc.prg
Dateien-Suchpfad     -
Aktuelles Laufwerk   - C:
ALTERNATE   - ON   DEBUG       - OFF  ESCAPE      - ON   MENU        - ON
BELL        - ON   DELETED     - OFF  EXACT       - OFF  PRINT       - OFF
CARRY       - OFF  DELIMITERS  - OFF  HEADING     - ON   SAFETY      - ON
CONFIRM     - OFF  DEVICE      - SCRN HELP        - ON   STEP        - OFF
CONSOLE     - ON   ECHO        - OFF  INTENSITY   - ON   TALK        - ON
UNIQUE      - OFF
```

```
Rand =     0

Funktion 1  F1   - help;
Funktion 2  F2   - assist;
Funktion 3  F3   - list;
Funktion 4  F4   - dir;
Funktion 5  F5   - display structure;
Funktion 6  F6   - display status;
Funktion 7  F7   - display memory;
Funktion 8  F8   - display;
Funktion 9  F9   - append;
Funktion 10 F10  - edit;
```

LIST STRUCTURE liefert die Struktur des aktiven Datenbankfiles (Abschn. 3).
Siehe auch: DISPLAY, SET HEADING.

LOCATE

Syntax: **LOCATE** [<gb>] **FOR** <bedingung>

LOCATE beginnt am Anfang des Gültigkeitsbereichs (Standard ist ALL). Es positioniert den Dateizeiger auf den ersten Datensatz des aktuellen Datenbankfiles, der die Bedingung erfüllt (Abschn. 9.2).
Siehe auch: CONTINUE.

LOOP

Syntax: **LOOP**

LOOP ist nur im Kommandofile innerhalb eines DO WHILE-Kommandos sinnvoll. Es bewirkt dort die Übergabe der Programmsteuerung an den Anfang des innersten DO WHILE-Kommandos, der dieses LOOP enthält (Abschn. 11).
Siehe auch: DO WHILE, EXIT.

MODIFY COMMAND

Syntax: **MODIFY COMMAND** <filename>

MODIFY COMMAND ruft den angeschlossenen Textprozessor auf. Standard ist ein dBASE-interner Editor. Über TEDIT im Konfigurationsfile kann ein privater Editor angeschlossen werden (Abschn. 12). Das angegebene File (Standarderweiterung ist .prg) wird neu erstellt bzw. zur Korrektur angeboten.

MODIFY LABEL

Syntax: **MODIFY LABEL** <filename>

MODIFY LABEL dient zum Erstellen oder Editieren eines Formatfiles (.lbl) für Briefadressen (Abschn. 16). Die Ausgabe der Adressen erfolgt über das Kommando LABEL. MODIFY LABEL beinhaltet CREATE LABEL.

MODIFY REPORT

Syntax: **MODIFY REPORT** <filename>

MODIFY REPORT dient zum Erstellen oder Editieren eines Formatfiles (.frm) für Standardberichte (Abschn. 10). Die Ausgabe des Berichts erfolgt über das Kommando REPORT. MODIFY REPORT beinhaltet CREATE REPORT.

MODIFY STRUCTURE

Syntax: **MODIFY STRUCTURE**

MODIFY STRUCTURE gestattet die Korrektur der Struktur des aktiven Datenbankfiles im Direktmodus (Abschn. 7). Die im File gespeicherten Daten werden in das veränderte Datenbankfile übernommen.

NOTE/*

Syntax: **NOTE/*** <text>

NOTE oder * kennzeichnen eine Zeile als Kommentar.

PACK

Syntax: **PACK**

PACK bewirkt ein physisches Löschen aller Datensätze, die als gestrichen markiert sind. Es erfolgt eine Umspeicherung aller Sätze des Files. Deshalb sollte PACK nur selten benutzt werden. Als gestrichen markierte Sätze können von Operationen ausgenommen werden, indem man SET DELETED ON setzt.

PACK läßt ein zugehöriges Memo-File (.dbt) unverändert. Memo-Files können nur durch COPY gepackt (verkleinert) werden.

Siehe auch: BROWSE, COPY, DELETE, DELETED(), EDIT, SET DELETED.

PARAMETERS

Syntax: **PARAMETERS** <parliste>

PARAMETERS steht als erstes Kommando in einem Kommandofile oder in einer Prozedur und spezifiziert Speichervariable, in denen aktuelle Parameter übergeben werden (Abschn. 11.3).
Siehe auch: DO, PRIVATE, PROCEDURE, PUBLIC, SET PROCEDURE.

PRIVATE

Syntax: **PRIVATE** [**ALL** [**LIKE/EXCEPT** <maske>]] [<mem_var_liste>]

PRIVATE beschränkt die Sichtbarkeit von Speichervariablen auf die Prozedur, in der dieses PRIVATE steht. Als PRIVATE erklärte Speichervariablen sind weder von rufenden noch von gerufenen Prozeduren aus sichtbar, immer bezogen auf die Prozedur, in der PRIVATE steht.
Siehe auch: DO, PARAMETERS, PUBLIC.

PROCEDURE

Syntax: **PROCEDURE** <proc_name>

PROCEDURE ist nur im Kommandofile sinnvoll. Es kennzeichnet den Anfang einer Kommandofolge, die im DO-Kommando über den Prozedurnamen erreicht werden kann. Die so definierte und benannte Kommandofolge beginnt mit dem ersten Kommando nach diesem PROCEDURE und endet mit dem nächsten RETURN, PROCEDURE oder Fileende. Ein Kommandofile kann bis zu 32 Prozeduren enthalten. Man erreicht dadurch, daß bei Folgen von DO-Kommandos die Anzahl der gleichzeitig eröffneten Kommandofiles klein gehalten wird. Eine Prozedur wird durch DO <proc_name> aufgerufen. Zuvor muß das Kommandofile, in dem diese Prozedur steht, durch SET PROCEDURE TO eröffnet worden sein. Kommandofiles, die Prozeduren enthalten, heißen auch Prozedurfiles (Abschn. 11.2).
Siehe auch: DO, PARAMETERS, PRIVATE, SET PROCEDURE.

PUBLIC

Syntax: **PUBLIC** <mem_var_liste>

PUBLIC erweitert den Sichtbarkeitsbereich der aufgeführten Speichervariablen auf alle benutzten Kommandofiles (Abschn. 11).
Siehe auch: DO, PARAMETERS, PRIVATE, PROCEDURE.

QUIT

Syntax: **QUIT**

QUIT schließt alle eröffneten Files ab und verläßt dBASE. Es ist

der einzige sichere Weg, dBASE zu verlassen! Jeder andere, illegale Weg kann zum Datenverlust führen.
Siehe auch: RUN.

READ

Syntax: **READ**

READ aktiviert den GET-Teil aller seit dem letzten READ, CLEAR GETS bzw. dem Start von dBASE abgearbeiteten @-Kommandos und setzt den Kursor auf den Eingabebereich des ersten GET-Kommandos (Abschn. 13.2).
Siehe auch: @, CLEAR, SET FORMAT.

RECALL

Syntax: **RECALL** [<gb>] [**FOR/WHILE** <bedingung>]

RECALL streicht die Löschmarkierung der ausgewählten Datensätze.
Siehe auch: BROWSE, DELETE, DELETED(), EDIT, PACK, SET DELETED.

REINDEX

Syntax: **REINDEX**

REINDEX aktualisiert alle aktiven Indexfiles. REINDEX ist nach großen Korrekturen vorteilhaft einsetzbar. Vor der Korrektur werden alle Indexfiles abgeschlossen (CLOSE INDEX). Nach der Korrektur werden sie durch SET INDEX wieder in Benutzung genommen. Ein nachfolgendes REINDEX erstellt jetzt alle aktiven Indexfiles neu, ohne daß die Indexausdrücke und Filenamen wiederholt eingegeben werden müssen.
Siehe auch: CLOSE, INDEX, SET INDEX, SET UNIQUE, USE.

RELEASE

Syntax: **RELEASE** [<mem_var>] [**ALL** [**LIKE/EXCEPT** <maske>]]

RELEASE vernichtet die ausgewählten Speichervariablen. Dadurch wird Speicherplatz im Memory-Bereich frei. Muß man Variablen streichen, die man später noch benötigt, so kann man diese vor RELEASE in ein Memory-File (.mem) ausgeben (SAVE) und später über RESTORE zurückholen. Die Größe des Bereichs der Speichervariablen kann im Konfigurationsfile (Abschn. 17.2) festgelegt werden.
Siehe auch: CLEAR, LIST MEMORY, RESTORE, SAVE.

RENAME

Syntax: **RENAME** <alter_filename> **TO** <neuer_filename>

RENAME benennt Files auf einem Datenträger um.

REPLACE

Syntax: **REPLACE** [<gb>] <feld> **WITH** <ausdr>
[,<feld_2> **WITH** <ausdr_2>,...]
[**FOR/WHILE** <bedingung>]

REPLACE ersetzt in allen durch den Gültigkeitsbereich <gb> bestimmten Datensätzen den Inhalt der Felder <feld_i> durch den Wert des zugehörigen Ausdrucks <ausdr_i> (Abschn. 6.5).

REPORT

Syntax: **REPORT FORM** <filename> [<gb>] [**FOR** <bedingung>] [**PLAIN**]
[**HEADING** <zeichenkette>] [**NOEJECT**] [**TO PRINT**]
[**TO FILE** <filename>]

REPORT bewirkt die Ausgabe eines Berichts in einem Format, welches durch CREATE REPORT definiert wurde (Abschn. 10).

Siehe auch: CREATE REPORT, MODIFY REPORT.

RESTORE

Syntax: **RESTORE FROM** <filename> [**ADDITIVE**]

RESTORE lädt alle Speichervariablen, die über SAVE in das angegebene Memory-File (.mem) gerettet wurden. Fehlt ADDITIVE, so werden vor dem Laden des Files alle Speichervariablen vernichtet. Der Inhalt des Memory-Bereichs besteht hinterher aus genau den Variablen, einschließlich ihrer Typen und Inhalte, die im File gespeichert wurden (SAVE). Ist ADDITIVE angegeben, so bleiben die alten Speichervariablen erhalten. Sind im Memory-File Speichervariablen mit gleichem Namen vorhanden, so wird die alte Deklaration dieser Variablen überschrieben. Die Wirkung ist die gleiche wie bei einer Ergibtanweisung (STORE oder '=').
Siehe auch: =, CLEAR MEMORY, RELEASE, SAVE, STORE.

RETURN

Syntax: **RETURN** [**TO MASTER**]

RETURN bewirkt die Rückgabe der Programmsteuerung an das aufrufende Kommandofile (Prozedur) bzw. an das rufende Kommandofile des

höchsten Niveaus (TO MASTER). Voraussetzung für eine sinnvolle Abarbeitung des RETURN-Kommandos ist ein vorher ausgeführtes DO <name>. Mit Abarbeitung von RETURN wird die Programmsteuerung an das Kommando übergeben, das dem rufenden DO <name> folgt.

Wird eine Prozedur oder ein Kommandofile nicht mit RETURN beendet, so führt dBASE selbständig ein RETURN aus, wenn die Programmsteuerung zu dieser Stelle kommt. TO MASTER kann für eine Fehlerbehandlung sinnvoll eingesetzt werden.

RUN

Syntax: **RUN** <kommando>

<kommando> ist ein MS-DOS-Kommando, allgemein ein Kommando des umgebenden Betriebssystems einschließlich seiner Parameter. Dieses Kommando wird abgearbeitet, wobei alle Einstellungen von dBASE erhalten bleiben. RUN erfordert zusätzlichen Speicherplatz über die Minimalanforderung von 256 KByte hinaus (Abschn. 17.3). Ist nicht genügend Speicherplatz vorhanden, so erscheint eine Fehlernachricht.

SAVE

Syntax: **SAVE TO** <filename> [**ALL LIKE/EXCEPT** <maske>]

SAVE speichert alle ausgewählten Speichervariablen in ein Memory-File (.mem). Sie können danach gelöscht (CLEAR MEMORY, RELEASE) oder überschrieben werden. Werden Sie später wieder benötigt, so können sie über RESTORE wieder in den Memory-Bereich geladen werden.
Siehe auch: CLEAR MEMORY, RELEASE, RESTORE, SET SAFETY.

SEEK

Syntax: **SEEK** <ausdr>

Ein Datenbankfile sei mit einem Indexfile in Benutzung. Dann positioniert SEEK den Dateizeiger auf den ersten Datensatz, dessen Indexschlüssel mit dem angegebenen Ausdruck übereinstimmt. Der Ausdruck kann vom Typ numerisch oder String sein. FIND akzeptiert dagegen nur Stringausdrücke, hat ansonsten die gleiche Wirkung wie SEEK (Abschn. 9.2).
Siehe auch: FIND, INDEX, LOCATE, SET EXACT, SET DELETED, SET INDEX, SET UNIQUE, USE.

SELECT

Syntax: **SELECT** <arbeitsbereich>/<alias>

SELECT wählt einen Arbeitsbereich als aktiven Arbeitsbereich aus. Die zehn Arbeitsbereiche werden mit den Zahlen 1 bis 10 oder den

Buchstaben A bis J bezeichnet. Soll ein Arbeitsbereich ausgewählt werden, in dem bereits ein Datenbankfile eröffnet ist, so erreicht man ihn über seinen Aliasnamen. Der Aliasname wird bei USE festgelegt, oder er ist gleich dem Namen des Datenbankfiles (siehe Beispiel zu LIST STATUS in diesem Abschnitt).
Siehe auch: SET RELATION, USE.

SKIP

Syntax: **SKIP** [<ausdr>]

Der Ausdruck wird berechnet und liefert einen Wert n. Der Dateizeiger wird um n Datensätze vorwärts (n>0) oder rückwärts (n<0) entsprechend der logischen Ordnung bewegt. Ist kein Ausdruck angegeben, so wird SKIP 1 ausgeführt. Ist das Datenbankfile ohne Indexfile in Benutzung, so ist die logische Ordnung gleich der physischen Ordnung. Sonst bestimmt das erste Indexfile, der Hauptindex, die logische Ordnung.

Wird durch SKIP der Anfang oder das Ende des Datenbankfiles überschritten, so liefern BOF() bzw. EOF() wahr (.T.). RECNO() liefert bei EOF()=.T. die größte Satznummer + 1. Ist BOF()=.T., so liefert RECNO() den Wert 1 (siehe Beispiel zu BOF() im Abschn. 4.7).

Siehe auch: BOF(), EOF(), RECNO().

SORT

Syntax: **SORT** [<gb>] **TO** <neues_file> **ON** <feld> [**/A**] [**/D**]
[,<feld2] [**/A**] [**/D**]...] [<gb>] [**FOR/WHILE** <bedingung>]

SORT sortiert das aktive Datenbankfile nach den angegebenen Feldern und der angegebenen Reihenfolge. Dabei wird ein neues Datenbankfile erzeugt (Abschn. 8.1).

STORE/=

Syntax: **STORE** <ausdr> **TO** <mem_var_liste>
<mem_var> = <ausdr>

STORE speichert den Wert des Ausdrucks in alle angegebenen Speichervariablen. Die zweite Form weist den Wert des Ausdrucks der Speichervariablen zu. Beide Formen bezeichnen wir als Ergibtanweisung. Jede Ergibtanweisung definiert eine neue Variable. Typ und Format werden durch den Wert bestimmt. Existiert bereits eine Variable mit dem gleichen Namen, so wird diese vernichtet.

Haben ein Feld und eine Variable den gleichen Namen, so wird auf das Feld zugegriffen. Der Zugriff auf die Speichervariable kann durch M -> <mem_var> erzwungen werden (siehe Beispiel im Abschn. 4.2).

Es ist nicht möglich, Feldern mit einer Ergibtanweisung Werte zu-

zuweisen. Dafür ist REPLACE geeignet.
Siehe auch: REPLACE.

SUM

Syntax: **SUM** [<gb>] [<ausdr_liste>] [**TO** <mem_var_liste>]
[**FOR/WHILE** <bedingung>]

SUM berechnet die Summen der in der Ausdrucksliste angegebenen Werte für alle ausgewählten Datensätze und weist sie, wenn angegeben, den Speichervariablen zu. SUM ist seiner Wirkung nach eine Funktion, kann aber nicht in Ausdrücken verwendet werden. Die Standardannahme für den Gültigkeitsbereich ist ALL. Fehlt die Ausdrucksliste, so wird die Summe über alle numerischen Felder des aktiven Datenbankfiles gebildet.
Siehe auch: AVERAGE.

TEXT

Syntax: **TEXT**
<text>
ENDTEXT

Dieses Kommando ist nur in einem Kommandofile sinnvoll. Es gibt einen Text, der aus mehreren Zeilen bestehen kann, auf dem Bildschirm aus. Dieser Text wird von dBASE nicht interpretiert, d.h., es erfolgt auch keine Makroersetzung (Abschn. 11). Hierin besteht der Unterschied zum ?-Kommando.
Siehe auch: @, ?, ??.

TOTAL

Syntax: **TOTAL** [<gb>] **ON** <schlüssel> **TO** <filename>
[**FIELDS** <feldliste>] [**FOR/WHILE** <bedingung>]

TOTAL erzeugt aus einem sortierten oder indizierten Datenbankfile ein neues Datenbankfile. Das erzeugte Datenbankfile enthält die Summen der ausgewählten numerischen Felder (Abschn. 15.3).

TYPE

Syntax: **TYPE** <filename> [**TO PRINT**]

TYPE gibt ein Textfile auf dem Bildschirm oder auf dem Drucker (TO PRINT) aus. TYPE ist nicht zu verwechseln mit der Funktion TYPE().

UPDATE

Syntax: **UPDATE [RANDOM] ON** <schlüssel> **FROM** <alias>
REPLACE <feld_1> **WITH** <ausdr_1>
{,<feld_2> **WITH** <ausdr_2>...}

UPDATE korrigiert ein Datenbankfile, wobei die Korrekturdaten von einem anderen Datenbankfile eingelesen werden (Abschn. 15.1).

USE

Syntax: **USE** [<filename>] [**INDEX** <fileliste>] [**ALIAS** <alias>]

USE nimmt das Datenbankfile <filename> mit den in der <feldliste> aufgeführten Indexdateien in Benutzung. Indexfiles können auch extra in Benutzung genommen werden (SET INDEX). Der Aliasname kann frei gewählt werden und stellt ein Synonym für das Datenbankfile und den Arbeitsbereich, in dem es in Benutzung ist, dar.

Der Aliasname wird für zwei Dinge benötigt: erstens, um einen Arbeitsbereich als aktiven Arbeitsbereich auszuwählen, z.B. SELECT lager; zweitens, um auf ein Feld eines Datenbankfiles in einem anderen Arbeitsbereich Bezug zu nehmen, z.B. lageradr->ort. Fehlt der Aliasname, so wird der Name des Datenbankfiles zum Aliasnamen.

Nach USE zeigt der Dateizeiger auf den ersten Satz in logischer Ordnung. Sind mehrere Indexfiles angegeben, so bestimmt das erste die logische Ordnung (Hauptindex).

Siehe auch: CLEAR ALL, CLOSE, INDEX, SELECT, SET INDEX.

WAIT

Syntax: **WAIT** [<prompt>] [TO <mem_var>]

Wird ein WAIT abgearbeitet, so wird das Prompt ausgegeben. Die Abarbeitung des Programms wird unterbrochen, bis eine Taste gedrückt wird. Deren Zeichenkode wird, falls angegeben, der Speichervariablen zugewiesen. Wird eine Taste gedrückt, die kein druckbares Zeichen darstellt, z.B. **Enter**, so wird ein Byte 00h geliefert.

Das Prompt ist ein Ausdruck, der eine Zeichenkette liefert. Fehlt es, so wird ein Standardtext ausgegeben.

Siehe auch: @, ACCEPT, INPUT.

ZAP

Syntax: **ZAP**

ZAP löscht alle Datensätze des aktiven Datenbankfiles. Alle aktiven Indexfiles werden korrigiert. Als Resultat entsteht also ein Datenbankfile mit null Sätzen und ggf. leeren Indexfiles.

19. dBASE intern

In diesem Abschnitt wollen wir uns mit dem internen Aufbau von Datenbankfiles und Memo-Files in dBASE III beschäftigen. Diese Informationen können immer dann vorteilhaft ausgenutzt werden, wenn aus anderen Programmiersprachen heraus auf Datenbankfiles zugegriffen werden soll. Ein sinnvoller Anwendungsfall liegt z.B. vor, wenn auf Daten des Datenbankfiles in anderen Projekten, die ansonsten überhaupt nichts mit der Datenbank zu tun haben, zugegriffen werden muß. Man kann so eine Neuerfassung oder Dopplung der Daten verhindern. Sinnvoll ist der Zugriff auch dann, wenn in einer Programmiersprache Algorithmen bereitgestellt werden, die wesentliche Vorteile gegenüber den in dBASE implementierten Algorithmen bringen.

Schließlich ist es auch denkbar, daß aus einem anderen Datenbanksystem heraus Datenbankfiles von dBASE gelesen werden (vgl. auch Abschn. 17.5). Das gilt vor allem dann, wenn das Zielsystem als Quelle in einer höheren Programmiersprache vorliegt. Durch die rasante Verbreitung UNIX-ähnlicher Betriebssysteme und der Programmiersprache C sind in C geschriebene Datenbanksysteme stark im Kommen. Übrigens: Es gibt auch dBASE-Versionen, die in C, und andere, die in Modula geschrieben wurden!

Datenbankfiles von dBASE II und dBASE III unterscheiden sich wesentlich in ihrem Aufbau. Deshalb sind Konvertierungsprogramme erforderlich (Abschn. 17.4). Das trifft auch auf die Organisation der Indexfiles zu.

Der Aufbau der Datenbankfiles von dBASE III und dBASE III plus unterscheidet sich nur insofern, daß bestimmte Eintragungen in dBASE III nicht benutzt werden.

19.1. Aufbau eines Datenbankfiles

Wir geben hier den Aufbau eines Datenbankfiles in dBASE III plus an und kennzeichnen die Teile, auf die in dBASE III nicht zugegriffen wird.

Ein Datenbankfile besteht aus einem Kopf variabler Länge und aus den Daten der einzelnen Sätze. Sein Aufbau ist in Tafel 19.1 wiedergegeben.

Der Kopf enthält u.a. eine Beschreibung jedes Feldes, den sog. Felddeskriptor. Den Aufbau eines Felddeskriptors zeigt Tafel 19.2.

Die Länge eines Datensatzes ergibt sich aus der Summe der Längen der einzelnen Felder plus eins. Jedem Satz ist ein Byte vorangestellt, welches die Löschmarkierung aufnehmen kann (DELETE oder ^U im Direktmodus). Ein Stern ('*' entspricht 2Ah) in diesem Byte markiert den Satz als gestrichen. Sonst ist hier ein Leerzeichen eingetragen. Dann folgen die Felder entsprechend ihrer Reihenfolge in der Definition (CREATE) ohne ein Trennzeichen zwischen den Feldern und ohne ein Trennzeichen zwischen den Sätzen.

Tafel 19.1. Struktur des Kopfes eines Datenbankfiles (.dbf)

Byte	Länge oder Datenformat	Bedeutung
0	1 Byte	Versionsnummer: x3 für dBASE III und x=0, d.h. 03h, falls ohne dbt-File benutzt x=8, d.h. 83h, falls mit dbt-File
1 - 3	3 hex-Byte	Datum der letzten Änderung in der Form jjmmtt, wobei jeder Buchstabe eine Hexadezimalziffer angibt. 580116 für den 22.01.88
4 - 7	32-Bit-Zahl	Anzahl der Sätze im Datenbankfile
8 - 9	16-Bit-Zahl	Anzahl der Bytes im Kopf des Datenbankfiles
10 - 11	16-Bit-Zahl	Anzahl der Bytes, die ein Datensatz im Datenbankfile einnimmt. Das ist die Summe der Längen der Felder plus ein Byte für die Löschmarkierung.
12 - 14	3 Byte	Reserviert
15 - 27	13 Byte	Reserviert für Mehrnutzerbetrieb (III plus)
28 - 31	4 Byte	Reserviert
32 - max	32*i Byte	Deskriptoren der Felder. 32 Byte je Feld
max+1 - max+2	1 Byte	Markierung des Kopfendes: 0Dh 00h

Die Werte aller Datentypen werden als ASCII-Zeichen im Datenbankfile gespeichert. Die zulässigen Zeichen bzw. Werte sind für die einzelnen Typen:

String (character) - alle druckbaren ASCII-Zeichen, d.h. Sonderzeichen, insbesondere Umlaute, sind möglich. Es lassen sich aber nicht alle ASCII-Steuerzeichen eintragen.

numerisch - die Zeichen '-', '.' und '0' bis '9'. Es erfolgt also keine Darstellung als Gleitkommazahl, obwohl z.B. bei einigen Standardfunktionen im Gleitkommaformat der I8087 gerechnet wird.

logisch - die Zeichen '?', 'Y', 'y', 'J', 'j', 'T', 't', 'N', 'n', 'F' und 'f', abhängig von der Implementation.

Datum - 8 BCD-Ziffern (binary coded decimals) in der Form jjjjmmtt, wobei 'j' für Jahr, 'm' für Monat und 't' für Tag steht. 18750817 steht für das Datum 17.08.1875 (vgl. Abschn. 4.6).

Memo - 10 BCD-Ziffern, die eine Blocknummer im zugehörigen .dbt-File darstellen.

Das Ende eines Datenbankfiles wird durch ein Fileendekennzeichen (1Ah) markiert. Aus den Angaben des Kopfes des Datenbankfiles ist das Fileende ebenfalls berechenbar.

Tafel 19.2. Aufbau eines Felddeskriptors

Byte	Länge oder Datenformat	Bedeutung
0 - 10	11 Byte	Feldname als ASCII-Zeichenfolge. Ist die Bezeichnung kürzer, so wird mit Nullbytes (00h) aufgefüllt.
11	1 Byte	Typ des Feldes als ASCII-Zeichen (C, N, L, D oder M)
12 - 15	32-Bit-Zahl	Adresse der ersten Daten dieses Feldes. Gesetzt, sobald das File zur Bearbeitung in den Speicher geladen wird. Wird ein File von der Platte gelesen, so Inhalt undefiniert.
16	1 Byte	Feldlänge, binär
17	1 Byte	Anzahl der Dezimalstellen
18 - 19	2 Byte	Reserviert für Mehrnutzerbetrieb (III plus)
20	1 Byte	Kennzeichnung des Arbeitsbereichs (III plus)
21 - 22	2 Byte	Reserviert für Mehrnutzerbetrieb (III plus)
23	1 Byte	SET FIELDS-flag (III plus)
24 - 31	8 Byte	Reserviert

Wir wollen an einem Programm in der Sprache Pascal, übersetzt mit Turbo-Pascal Vers. 3.0, zeigen, wie man auf ein Datenbankfile aus einer anderen Programmiersprache zugreift. Pascal ist hier beliebig gewählt. Genauso kann man die Sprachen C, Modula oder Prolog benutzen. Wie man z.B. aus der Sprache Prolog auf Datenbankfiles zugreift, wird in der Arbeit von Konzack [Kon87] beschrieben. dBASE III plus-Werkzeuge für Turbo-Pascal sind bei Cooper [Coo87.3] zu finden und für die Programmiersprache C in [Coo87.2].

Der realistische Hintergrund für diese Aufgabe ergibt sich aus dem Bedürfnis, die einmal erstellten Datenbanken auch in anderen Programmen zu benutzen. Dadurch wird eine doppelte Datenspeicherung und eine doppelte Erfassung vermieden. Denkbar ist auch ein Umzug in ein anderes Betriebssystem, z.B. UNIX, oder die Nutzung von dBASE-Datenbanken für Expertensysteme.

Das Programm setzt voraus, daß die Angaben bekannt sind, die über LIST STRUCTURE gewonnen werden können. Stehen auch diese nicht zur Verfügung, so lassen sie sich prinzipiell genauso dem Kopf des Datenbankfiles entnehmen.

Um auch den Zugriff zu logischen Größen zu zeigen, erweitern wir unser Datenbankfile 'lager.dbf' am Ende um ein Feld vom Typ logisch.

Wir weisen noch darauf hin, daß es nicht möglich ist, ein Programm zu erstellen, das alle denkbaren Datenbankfiles beliebig verarbeitet. Solange man auf der Ebene der Zeichenverarbeitung bleibt, ist das möglich. Sobald aber Typen berücksichtigt werden müssen, muß der Quelltext korrigiert werden. Es sind Konvertierungsfunktionen anzuwenden und Operationen, die für Objekte eines Typs spezifisch sind. So ist die Addition in Pascal nur auf Größen von Typ INTEGER und REAL anwendbar. Ist eine Typumwandlung überhaupt erforderlich? Die Beantwortung dieser und ähnlicher Fragen beeinflußt das Pascal-Programm. Man sollte allerdings bemüht sein, die von Fall zu Fall vorzunehmen-

den Änderungen im Kopf des Programms zu konzentrieren.

```
program liesdbf;
(* Ganschow, Th.; Grafik, W.; Schulz, A.: Lesen eines
   dBASE III-Datenbankfiles in Turbo-Pascal
*)

CONST
  fdl = 32;          (* Länge eines Felddeskriptors *)
  (* Die folgenden Konstanten werden z.B. aus LIST STRUCTURE entnommen. *)

  Satzlaenge = 61;   (* Länge eines Datensatzes einschließlich
                        Löschmarkierung *)
  Satzanzahl = 9;    (* Anzahl der Datensätze im dbf-File *)
  Feldanzahl = 8;    (* Anzahl der Felder *)
  dbfName = 'LAGER1.DBF'; (* Name des Datenbankfiles *)
  lf1 = 15;          (* Länge des ersten Feldes 'artikel' *)
  lf2 = 4;           (* Länge des zweiten Feldes 'anzahl' *)
  lf3 = 8;           (* 'preis' *)
  lf4 = 2;           (* 'minimum' *)
  lf5 = 8;           (* 'lieferung' *)
  lf6 = 12;          (* 'hersteller' *)
  lf7 = 10;          (* 'bem' *)
  lf8 = 1;           (* 'bestellt' *)

TYPE Satztyp = ARRAY[1..Satzlaenge] OF CHAR;

VAR
  f1, f2, f3, f4, f5, f6, f7, f8: INTEGER;
  dbFile: FILE OF BYTE;
  Satz: Satztyp;     (* Bereich zum Einlesen eines Satzes *)
  recno: INTEGER;    (* Satzzeiger im Datenbankfile *)
  Datenanfang: INTEGER; (* Position des Datenanfangs im
                           dbf-File *)
  i: INTEGER;

PROCEDURE LiesSatz(SatzNr: INTEGER; VAR s: Satztyp);
(* Positioniert im Direktzugriff auf den Datensatz mit der Nummer
   SatzNr und liefert den Inhalt des Satzes in s.
   Vor.: dbf-File eröffnet
         globale Größen: dbFile, Datenanfang, Satzlaenge *)
VAR i:INTEGER; b: BYTE;
BEGIN
  seek(dbFile,Datenanfang + (SatzNr - 1) * Satzlaenge);
  read(dbFile,b); (* Löschmarkierung überlesen *)
  FOR i := 1 TO Satzlaenge - 1 DO
  BEGIN
    read(dbFile,b);
    s[i] := chr(b);
  END;
END; (* LiesSatz *)
```

```
PROCEDURE PrintString(s:Satztyp; anfPos, Laenge: INTEGER);
(* Druckt eine Teilzeichenkette aus dem String s ab der
   Anfangsposition in gegebener Länge *)
VAR i:INTEGER;
BEGIN
  FOR i := anfPos TO anfPos + Laenge -1 DO write(s[i]);
END; (* PrintString *)

PROCEDURE PrintDate(s:Satztyp; anfPos: INTEGER);
(* Druckt ein Datum aus dem String s ab der
   Anfangsposition in gegebener Länge *)
CONST
  monat = 4;    (* Verschiebung der Monatsangabe zum Feldanfang *)
  tag = 6;      (* Verschiebung der Tagesangabe zum Feldanfang *)
BEGIN
  write(s[anfPos + tag], s[anfPos + tag + 1], '.');
  write(s[anfPos + monat], s[anfPos + monat + 1], '.');
  write(s[anfPos], s[anfPos + 1], s[anfPos + 2], s[anfPos + 3]);
END; (* PrintDate *)

PROCEDURE PrintLog(s:Satztyp; anfPos: INTEGER);
(* Druckt einen logischen Wert aus dem String s ab der
   Anfangsposition in gegebener Länge *)
BEGIN
  IF (s[anfPos]='T') or (s[anfPos]='Y') THEN write('T')
  ELSE write('F');
END; (* PrintLog *)

BEGIN
  (* Die folgenden Wertzuweisungen zu den fi liefern die
     Verschiebung des i-ten Feldes relativ zum Satzanfang. Sie sind
     ihrer Bedeutung nach Konstanten. *)
  f1 := 1;            (* 0. Zeichen enthält die Löschmarkierung *)
  f2 := f1 + lf1;     (* 'anzahl' *)
  f3 := f2 + lf2;     (* 'preis' *)
  f4 := f3 + lf3;     (* 'minimum' *)
  f5 := f4 + lf4;     (* 'lieferung' *)
  f6 := f5 + lf5;     (* 'hersteller' *)
  f7 := f6 + lf6;     (* 'bem' *)
  f8 := f7 + lf7;     (* 'bestellt' *)

  (* Eröffnen des Datenbankfiles zum Lesen *)
  assign(dbFile,dbfName);
  reset(dbFile);
  (* Berechnen des Datenanfangs im .dbf-File *)
  Datenanfang := (Feldanzahl+1) * fdl + 2;
  (* Wir zeigen den richtigen Zugriff durch die Nachbildung
     des LIST-Kommandos. Im Normalfall folgen hier die vom Nutzer
     gewünschten Operationen. *)
  writeln;
```

```
    FOR i:= 1 TO SatzAnzahl DO
    BEGIN
      LiesSatz(i,Satz);
      PrintString(Satz,f1,lf1); write(' ');
      PrintString(Satz,f2,lf2); write(' ');
      PrintString(Satz,f3,lf3); write(' ');
      PrintString(Satz,f4,lf4); write(' ');
      PrintDate(Satz,f5); write(' ');
      PrintString(Satz,f6,lf6); write(' MEMO ');
      PrintLog(Satz,f8);
      writeln;
    END;
    close(dbFile);
  END.
```

Wir beenden diesen Abschnitt mit dem Bild 19.1, welches den internen Aufbau unseres Datenbankfiles 'lager.dbf' zeigt.

```
00000000   83 58 01 17 09 00 00 00 02 01 3C 00 00 00 00 00     X        <
00000010   00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000020   41 52 54 49 4B 45 4C 00 00 00 00 43 07 00 4E 6F    ARTIKEL    C  No
00000030   0F 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000040   41 4E 5A 41 48 4C 00 00 00 00 00 4E 16 00 4E 6F    ANZAHL     N  No
00000050   04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000060   50 52 45 49 53 00 00 00 00 00 00 4E 1A 00 4E 6F    PREIS      N  No
00000070   08 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00000080   4D 49 4E 49 4D 55 4D 00 00 00 00 4E 22 00 4E 6F    MINIMUM    N" No
00000090   02 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000A0   4C 49 45 46 45 52 55 4E 47 00 00 44 24 00 4E 6F    LIEFERUNG  D$ No
000000B0   08 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000C0   48 45 52 53 54 45 4C 4C 45 52 00 43 2C 00 4E 6F    HERSTELLER C, No
000000D0   0C 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
000000E0   42 45 4D 00 00 00 00 00 00 00 00 4D 38 00 4E 6F    BEM        M8 No
000000F0   0A 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00      .
00000100   0D 00 20 46 6C 61 63 68 73 63 68 69 65 62 65 72      Flachschieber
00000110   20 20 20 31 31 32 20 20 20 20 32 2E 34 30 31 35      112    2.4015
00000120   31 39 38 38 30 32 32 39 43 26 43 20 20 20 20 20    19880229C&C
00000130   20 20 20 20 20 20 20 20 20 20 20 20 20 33 20 4B                 3 K
00000140   75 72 62 65 6C 77 65 6C 6C 65 20 49 20 20 20 20    urbelwelle I
...
000002E0   20 20 20 46 75 E1 64 69 63 68 74 75 6E 67 20 20      Fu dichtung
000002F0   20 20 20 31 32 32 20 20 20 20 30 2E 31 35 33 30      122    0.1530
00000300   31 39 38 38 30 36 30 37 44 69 63 68 74 26 46 65    19880607Dicht&Fe
00000310   73 74 20 20 20 20 20 20 20 20 20 20 20 20 1A       st
```

Bild 19.1. Interne Darstellung des Datenbankfiles 'lager.dbf'

19.2. Aufbau eines Memo-Files

Memo-Files wurden in dBASE III eingeführt, um den Nachteil auszugleichen, den Datensätze konstanter Länge mit sich bringen. Sie haben den gleichen Filenamen wie das zugehörige Datenbankfile, tragen aber die Namenserweiterung '.dbt'. Die Datenbankfiles von dBASE haben konstante Länge. Ein Memo-File nimmt Textinformationen variabler Länge auf. Ein Memo-Feld im .dbt-File kann aus maximal 4096 Zeichen bestehen, ohne das Datenbankfile aufzublähen. Das täuscht dem Nutzer eine (effektive) Behandlung von Texten variabler Länge vor. Wie wird das Memo-File tatsächlich verwaltet?

Das gesamte Memo-File besteht aus Einheiten von 512 Byte, den sog. Blöcken, die, mit null beginnend, durchnumeriert werden. Der Block 0 ist der Kopfblock (header block). Vom Kopfblock werden nur die ersten 4 Byte benutzt. Sie enthalten die Nummer des nächsten freien Blocks im File. Der Rest des Kopfblocks ist leer. Die Blocknummer ist als binäre Zahl gespeichert, wobei das erste Byte das niederwertigste Byte ist. So steht z.B. 0B 00 00 00 für den Block 11 als nächsten freien Block.

Wird ein Memo-Feld betreten (^Home in APPEND, BROWSE, EDIT usw.), so wird ein neuer 512-Byte-Block bereitgestellt. Seine Nummer wird in 10 Byte des Memo-Feldes im Datenbankfile eingetragen. Der Text, so kurz er auch sei, belegt 512 Byte! Ist der Text länger als 510 Byte, so wird der nächste Block angefordert usw. bis zu 8 Blöcken je Memo-Feld. Zwei Byte 1Ah 1Ah kennzeichnen das Ende eines Memo-Textes.

Memo-Felder führen nur dann zu einer verträglichen Speicherauslastung, wenn viele Felder keine Texte enthalten. Dann wird auch kein Block im .dbt-File belegt. Die anderen Memo-Felder müssen dann sehr lange Texte enthalten. Am besten etwas weniger als ganze Vielfache von 512 Byte.

Werden Inhalte von Memo-Feldern korrigiert, so werden alle zu diesem Feld gehörenden Blöcke des .dbt-Files in neue Blöcke kopiert. Die alten Blöcke bleiben im File erhalten, jedoch existiert auf sie kein Verweis mehr. Sie werden beim Zusammenpacken gestrichener Sätze (PACK) nicht physisch vernichtet. Das .dbt-File wird durch PACK nicht bearbeitet. Die einzige Möglichkeit, überflüssige Memo-Felder zu streichen, besteht durch das Kommando COPY. Beim Kopieren der Datenbank werden alle Sätze des Memo-Files, auf die nicht aus dem .dbf-File verwiesen wird, gestrichen.

Diese Organisation der Memo-Felder stellt keine effektive Lösung für die Speicherung von Informationen variabler Länge dar. Eine Alternative sehen wir in einem selbstorganisierten zweiten Datenbankfile. Dieses besteht aus einem kleinen Kennzeichenfeld und einem Textfeld von Zeilenlänge (50, 64 oder 80 Zeichen, beliebig wählbar). Im ersten Datenbankfile haben wir für Sätze, die zusätzliche Informationen enthalten sollen, ein Kennzeichen eingetragen, welches im zweiten Datenbankfile wieder vorkommt. Das Verfahren ist das gleiche, wie wir es im Abschnitt 14 mit den Files 'lager.dbf' und 'lageradr.dbf' beschrieben haben.

In den Textteil des zweiten Datenbankfiles werden die Informationen jetzt auf so viele Felder hintereinander verteilt, wie notwendig

sind. Die Kennzeichnungsfelder dieser Fortsetzungssätze bleiben leer.

Drei Vorzüge sind sofort sichtbar: Erstens wird bei weitem nicht soviel Speicherplatz verschenkt wie bei Memo-Files. Durch die Wahl einer geeigneten Zeilenlänge kann die Effektivität beeinflußt werden. Zweitens läßt sich eine Nutzeroberfläche schaffen, die vor dem Endnutzer die konkrete Realisierung solcher privaten "Memo-Files" verbirgt. Drittens kann man mit diesen "privaten Memo-Feldern" mehr Operationen ausführen als mit den Standard-Memo-Feldern. So ist auch ein Suchen nach Teilzeichenketten möglich. Für Sammlungen von Stichwörtern, Literaturverzeichnissen usw. kann das sehr nützlich sein.

Wir unterstreichen diese Ausführungen durch einige Zahlen. Unser Datenbankfile 'lager.dbf' benötigt für neun Sätze 795 Byte. Das zugehörige Memo-File - der Inhalt ist aus den vorigen Abschnitten bekannt - belegt, nach einigen Korrekturen, 5632 Byte! Nach Ausführung des Kommandos COPY in dBASE werden noch 2560 Byte belegt.

20. Randprobleme

Betrachten wir nun einige Probleme, die bei der Arbeit mit dBASE oder mit Datenbanken allgemein sofort auftreten. Sie sind nur mittelbar Fragen von dBASE, können also bei unbefriedigender Beantwortung nicht ausschließlich dem Datenbanksystem angelastet werden.

20.1. Interpreter oder Compiler

Software ist, wie jede andere Technik, in den Entwicklungsprozeß einbezogen. Ein Programm, welches heute noch von hoher Qualität ist, kann den Anforderungen von morgen schon nicht mehr genügen. Jeder Anwender forciert diesen Prozeß, indem durch die Nutzung der Programme und der Technik die Möglichkeiten in allen Richtungen ausgelotet werden. Man versucht, die Grenzen zu weiten, neue Soft- und Hardware einzubeziehen. Auch dBASE ist in diesen Entwicklungsprozeß integriert. Deshalb ist es nicht verwunderlich, wenn nach einer gewissen Nutzungdauer von dBASE Wünsche entstehen, die nicht ohne weiteres befriedigt werden können. Die Geschwindigkeit von dBASE ist eins dieser Probleme.

dBASE ist ein Interpreter. Eine Textzeile wird mit dem Ziel analysiert, ein Kommando zu erkennen. Wurde ein Kommando erkannt, so wird es sofort ausgeführt. Nach der Ausführung wird die nächste Zeile des Textes betrachtet, analysiert usw. Dabei ist es unerheblich, ob diese Textzeilen im Kommandomodus von der Tastatur eingegeben oder aus einem Kommandofile gelesen werden. Werden die gleichen Kommandozeilen sehr oft interpretiert, wie es z.B. in Zyklen der Fall ist, so erfolgt die Analyse jedesmal. Das ist eigentlich überflüssig, denn das Kommando wurde ja schon beim erstenmal als richtig erkannt. Es entsteht also die Frage nach einem Compiler, d.h. nach einem Programm, welches Textzeilen analysiert und in eine maschinennahe Form überführt, die bei wiederholter Kommandoausführung nur noch abgearbeitet werden muß.

Es existieren Compiler, die dBASE-ähnliche Sprachen übersetzen. Der Übergang von dBASE zu einer dieser Sprachen ist dabei möglichst einfach gehalten. Viele dBASE-Programme können ohne irgendwelche Veränderungen sofort übersetzt werden. Mirecki [Mir87] untersucht drei dBASE III-Compiler: 'FoxBase', 'Clipper' [Cli85] und 'Quicksilver'. Er faßt sein Ergebnis wie folgt zusammen: Die Geschwindigkeit eines Anwenderprogramms für dBASE III kann, im Durchschnitt betrachtet, durch Compilation nicht über den Faktor 2 gesteigert werden. Das gilt "unabhängig davon, wie schnell der compilierte Kode ist". Da die dazu angeführten Meßwerte sich mit der Veränderung der Technik ebenfalls ändern, ist diese Aussage sicher zu relativieren. So ist es nicht verwunderlich, daß es genauso Nutzer gibt, die auf ihrer Technik mit compilierten dBASE-Kommandofiles gute Erfahrungen gesammelt

haben.

Dennoch gibt es bestimmte Gründe dafür, daß sich eine Übersetzung oft nicht lohnt:

1. dBASE arbeitet viel mit Files. Der Zugriff zu einem Datensatz, die Zugriffszeiten zur Platte und die Größe und Anzahl von Puffer- und Arbeitsbereichen sind deshalb bedeutsamer als die Übersetzung solcher Kommandos.

2. Die Kommandosprache von dBASE besitzt eine inhärente Effektivität. Sie kommt dadurch zustande, daß dBASE-Kommandos sehr komplexe Operationen beschreiben. Dadurch wird die Zeit zur Kommandoanalyse verschwindend gering im Vergleich zur Zeit für die Kommandoausführung. Auch in diesen Fällen ist kein großer Vorteil beim Übersetzen zu erwarten.

3. Es kommt entscheidend auf die Effektivität der implementierten Algorithmen an. So benutzt dBASE II z.B. einen relativ langsamen Sortieralgorithmus. Die Folge davon war, daß viele Nutzer mit anderen Programmen, z.B. mit DBplus, sortiert haben. dBASE III hat einen wesentlich schnelleren Sortieralgorithmus implementiert.

Darüber hinaus bietet ein Interpreter bessere Möglichkeiten für die Programmerstellung einschließlich des Programmtests. dBASE hat dafür die Kommandos SET DEBUG, SET ECHO, SET STEP und SET TALK.

Ein Compiler wird Vorteile bringen, wenn es um Kommandos geht, die oft abgearbeitet werden und viel im Speicher arbeiten. Zyklische Aufrufe von Kommandofiles und häufige Anwendung mathematischer Funktionen sind Beispiele dafür.
Ein vom Compiler erzeugter schneller Kode ist in seiner Wirkung vergleichbar mit einem schnelleren Prozessor.

Die einzige vernünftige Schlußfolgerung daraus ist, die Vorteile von Interpreter und Compiler zu nutzen. Entwickeln und testen Sie Ihre Programme mit dBASE III, und übersetzen Sie große und ausgetestete Programme dann mit einem Compiler.
Es gibt aber auch noch eine Zwischenlösung. Im Abschnitt 17.3 haben wir Möglichkeiten aufgezeigt, aus dBASE heraus andere Programme aufzurufen. Diese können dann Aufgaben realisieren, die besonders effektiv gelöst werden sollen. Auch hier verbietet sich ein ständiger Wechsel zwischen dBASE und einem andern, compilierten Programm.

20.2. Datenschutz und Sicherheit

dBASE III ist ein Datenbanksystem, zu dem auf einem Rechner immer nur ein Nutzer zugreifen kann. Der gleichzeitige Zugriff mehrerer Nutzer ist nicht möglich. Somit ist man sicher, daß während der

Arbeit mit dem System kein anderer Nutzer auf diese Daten zugreifen kann.

Schwieriger ist es schon bei der Benutzung von Festplatten. Sie verbleiben im Rechner. Dadurch kann der nächste Nutzer auf die Datenbank zugreifen, Informationen lesen, manipulieren oder die Datenbank beliebig zerstören. Für einen ernst zu nehmenden Nutzer ist das möglich, und es gibt keinen Schutz dagegen. Es sei denn, man schützt die Platte so, wie man auch einen Stapel Papier gegen unbefugte Benutzung schützt, indem man ihn unter Verschluß aufbewahrt.

Ist man gezwungen, seine Datenbank oder Teile davon anderen zugänglich zu machen, so läßt sich eine Nutzeroberfläche schaffen, die einen gewissen Schutz bietet. Schlüsselwörter, die in Kommandofiles abgefragt werden, und die Anzeige nur ausgewählter Felder bieten einen Schutz bei gutwilliger Nutzung. Es ist dagegen nicht kontrollierbar, ob ein Nutzer die Anwenderoberfläche durchbricht und unbefugt auf Informationen zugreift.

Für die Datenerfassung läßt sich eine Lösung anbieten. Man erfaßt die Daten unabhängig von der existierenden Datenbank und läßt sie später durch befugte Personen an die Datenbank anfügen. APPEND, UPDATE, TOTAL oder ein privates Kommandofile bieten sich hier an.

Diskutiert man das Problem der Datensicherheit in dBASE-Systemen, so muß man einfach berücksichtigen, daß die benutzte Hardware für einen personengebundenen Gebrauch bestimmt ist (personal computer). Verletzt man diese Voraussetzung, so darf man von der Software, hier von dBASE, keine Wunder bezüglich der Datensicherheit erwarten.

Interessanter ist das Problem der Datenkonsistenz, d.h., sind die Daten in sich richtig und verträglich? Formate, wie sie im @-Kommando durch PICTURE angegeben werden können, sind ein erster Schritt. Die Einhaltung bestimmter Grenzen für die eingegebenen Werte läßt sich ebenfalls im @-Kommando festlegen.

Zeichenfolgen, die durch Kontrollesen nur schwer zu verifizieren sind, z.B. Kontonummern, Kundennummern, Kostenstellen usw., können dadurch sicherer eingegeben werden, daß man diese Werte zweimal eingeben läßt. Erst bei der Übereinstimmung beider Angaben wird der Wert akzeptiert. Das ist durchaus nicht unzumutbar für den Datenerfasser, da konventionell in vielen Bereichen so gearbeitet wird.

Kontonummern, Personenkennzahlen usw. enthalten selbst schon Kontrollziffern. Am sichersten, dafür aber auch relativ langsam ist die erneute Berechnung der Kontrollziffer und ihre Überprüfung mit der eingegebenen. Ein Kommandofile kann hierfür eingesetzt werden.

Andere Angaben können vom Datenerfasser zwar verifiziert werden, sie müssen aber nicht mit den bereits erfaßten Daten verträglich sein. 'Kronenstraße', Kronenstr.' und 'Kronenstrasse' bzw. 'UdSSR', 'UDSSR', 'USA' und 'U.S.A.' sind Beispiele dafür. Durch eine günstige Programmierung können einige 'Fehler' ausgeglichen werden. Generell ist ein Konsistenztest dieser Art jedoch sehr schwierig.

Ein weiteres Problem der Datensicherheit betrifft die Erstellung von Sicherheitskopien der Datenbank. dBASE erstellt Sicherheitskopien nur für die Files, die mit MODIFY bearbeitet werden (MODIFY COMMAND,

MODIFY STRUCTURE). Das ist für Flüchtigkeitsfehler zwar ganz gut, für Datenbanken aber unzureichend. Der Nutzer ist selbst für die Erstellung von Sicherheitskopien verantwortlich. Generell sollte man darauf achten, daß nur vollständige Versionen eines Datenbanksystems kopiert werden. Wir verstehen darunter alle Files einschließlich Indexfiles, Formatfiles, Berichtfiles, Kommandofiles usw. Aus den vorangegangenen Abschnitten wissen wir, daß dBASE die Zusammengehörigkeit der ihm angebotenen Files nicht überprüfen kann. Datenverlust kann die Folge sein! Man sollte sich deshalb Kommandofiles für die Sicherung und die Wiedererstellung der Datenbank von einem Sicherheitsabzug aus erstellen.

Bei Fehlern in der Hardware oder Stromausfall hilft bei diesen Rechnern schließlich auch nur eine Sicherheitskopie. Zur regelmäßigen Wartung einer Datenbank gehört deshalb unbedingt eine Technologie für ihre Sicherung und Wiedererstellung [Bye87].

dBASE III plus besitzt durch seine Netzfähigkeit Möglichkeiten, Datensätze und Files gegen den gleichzeitigen Zugriff durch mehrere Nutzer zu schützen. Für die oben behandelten Probleme bietet es auch nicht mehr Schutz als dBASE III.

20.3. dBASE, verwandte Systeme, Entwicklungstendenzen

Der Einsatz von dBASE II und die dabei voranschreitende Entwicklung von Software und Hardware führten dazu, daß dBASE durch zusätzliche Programme ergänzt wurde. Der Nutzer von dBASE III möchte diese oder ähnliche Werkzeuge vorfinden. Betrachten wir deshalb einige Werkzeuge, die zwar nicht Bestandteil von dBASE sind, aber unbedingt in seine Umgebung gehören. Wir können diese Werkzeuge hier nur aufzählen und kurz charakterisieren.

dBASE II kennt Generatoren für Bildschirmmasken und Kommandofiles. ZIP ist ein Beispiel dafür. Zu dBASE III existiert ebenfalls ein verbesserter Generator für Kommandofiles: dFORMAT. Er erzeugt aus Bildern, die im Direktmodus auf dem Bildschirm erstellt werden, erst Textfiles, die er dann in Kommandofiles transformieren kann. Möchte man reine Bildschirmmasken wie bei ZIP haben, so ist das generierte Kommandofile zu besichtigen und ggf. zu trennen, wobei die @-Kommandos in ein .frm-File gelegt werden. In Abschnitt 13 haben wir die Vor- und Nachteile solcher Formatfiles betrachtet.

DataStar diente der verbesserten Datenerfassung für dBASE II-Datenbanken. Die Formatkontrollen in dBASE II und die mögliche Geschwindigkeit der Dateneingabe wurden oft als zu langsam eingeschätzt. dBASE III besitzt verbesserte Formatkontrollen, erfüllt aber noch nicht alle Anforderungen diesbezüglich. Zwei Lösungen können hier angeboten werden: erstens die Benutzung compilierter Kommandofiles, die die erforderlichen Formattests realisieren; zweitens die Datenerfassung über eine bewährte Lösung, die auf dBASE II-Files führt, und ihre anschließende Konvertierung nach dBASE III mit dCONVERT. Diese zweite Variante ist allerdings nur ein Notbehelf für

eine Phase des Umstiegs auf dBASE III. Mit hoher Sicherheit werden in Zukunft auch für die Datenerfassung unter dBASE III schnelle Varianten als Softwarepaket verfügbar sein. Der dBASE III-Compiler 'Clipper' bietet bereits erweiterte und schnelle Varianten der Bildschirmkommunikation im Direktmodus.

ReportStar ergänzte unter dBASE II die Erstellung von Berichten. dBASE III enthält ein wesentlich verbessertes REPORT-Kommando (Abschn. 10). Hier ist das gleiche anzuführen, was oben über die Datenerfassung gesagt wurde.

dGRAPH gestattet die graphische Auswertung von Daten, die in dBASE II-Datenbanken gespeichert sind. Es ist kein Problem von dBASE III, daß solche graphischen Darstellungsmöglichkeiten nicht implementiert sind. Aus der 16-Bit-Technik stehen genügend Programme zur Verfügung, die eine graphische Datendarstellung realisieren. Es ist also nur die Frage zu stellen, inwiefern dBASE III Daten so ausgeben kann, daß sie von einem solchen Programm aufgenommen werden können. dGRAPH existiert auch für dBASE III [Coo87.1].

dBCODE, dBLINKER und dBRUN sind Erweiterungsprogramme zu dBASE III, die eine Kodierung von dBASE-Programmen unterstützen. Das ist nicht zu verwechseln mit einer Übersetzung, wie sie durch einen dBASE-Compiler durchgeführt wird. Diese Programme bewirken eine Verschlüsselung und Komprimierung und erzeugen so eine effektivere Version des dBASE-Programms. dBCODE bewirkt die Kodierung. dBLINKER verbindet mehrere durch dBCODE kodierte Files zu einem File. Dieses kann dann mit dBRUN geladen und interpretativ ausgeführt werden.

Wir wollen zum Abschluß dieser Aufzählung betonen, daß die Existenz von Programmen, die sich um ein großes Programmpaket gruppieren und Teile von diesem weiter ausbauen oder effektiver gestalten, keineswegs gegen dBASE III spricht. Wollte man alle Teile in dBASE integrieren, so würde dBASE ins uferlose wuchern. Die Folge wären erhöhte Anforderungen an die Hardware, und damit könnte auch die universelle Einsetzbarkeit wieder zurückgehen. Betrachten wir diese Tochterprogramme von dBASE dagegen unter einem positiven Vorzeichen, so zeigt sich, daß dBASE III durchaus modular erweiterbar ist. Standardlösungen enthält dBASE selbst; für einige Probleme existieren eben diese Tochterprogramme, und für private Lösungen wurden in diesem Buch viele Möglichkeiten aufgezeigt, über Daten und Programme mit dBASE zu kooperieren.

Welche weitere Entwicklung von dBASE ist aus heutiger Sicht zu erwarten? 16-Bit-Rechner mit mehreren angeschlossenen Arbeitsplätzen (Terminals) sind heute Realität. Das verlangt, daß mehrere Nutzer gleichzeitig auf eine Datenbank zugreifen können. dBASE III plus besitzt Kommandos, um diesen gleichzeitigen und ausschließlichen Zugriff zu regeln. Man sagt: dBASE III plus ist netzfähig [Bat87] [Ash85.1] [Ash85.2] [Ash85.3]. Übrigens: Auch von allen oben aufgeführten Compilern existieren netzfähige Versionen.

Genauso steht die Frage des Zugriffs auf Datenbanken auf verbundenen Rechnern. Auch hierfür werden weiterentwickelte Varianten von

dBASE angeboten.

Wie sieht es mit verteilten Datenbanken auf mehreren Rechnern aus? Wie läßt sich die Kommunikation und Kooperation mit anderen Datenbanksystemen gestalten? Welche Möglichkeiten des Konsistenztests für Datenbanken gibt es? Diese komplexen Probleme bewegen die Theoretiker und Praktiker großer Datenbanken seit geraumer Zeit. Es ist abzusehen, daß dBASE sich diesen Anforderungen stellt. Mit der Ergänzung der Kommandosprache um einige Kommandos, wie z.B. beim Übergang von dBASE III zu dBASE III plus, dürfte es dabei nicht getan sein. Gesichert ist auf alle Fälle die Erkenntnis, daß dBASE III keine Sackgasse ist, sondern ein logischer Schritt innerhalb der Entwicklung. Stehen uns neue und vernetzte Rechner zur Verfügung, so werden wir ganz sicher unsere Daten von dBASE III auch dorthin übernehmen können.

Wie sieht es mit dem Übergang in andere Betriebssysteme aus? UNIX und UNIX-ähnliche Systeme stehen hier zur Diskussion. dBASE III, dBASE III plus und dBASE-Compiler existieren auch in UNIX-Systemen bzw. sind angekündigt. Möchte man mit seinen Daten in ein anderes Datenbanksystem, so führt der einfachste Weg über Textfiles. Im Abschnitt 17.5 wurde dieses Problem behandelt.

Zusammenfassend ist auch bezüglich des Übergangs in ein UNIX-ähnliches-Betriebssystem festzustellen, daß bereits Lösungen existieren und sicher weiter ausgebaut werden, so daß die Daten auch hier nicht neu erfaßt werden müssen.

Anhang 1:

Funktionen von dBASE III funktionell geordnet

Abschnitt 4.7 enthält eine ausführliche Beschreibung der Funktionen von dBASE III alphabetisch geordnet. In diesem Anhang teilen wir die Funktionen in fünf Gruppen ein. Tritt unter den Argumenten einer Funktion ein Ausdruck auf, so wird dieser berechnet, und die Funktion wird auf den so berechneten Wert angewendet.

Mathematische Funktionen

EXP(<ausdr>)	berechnet e<ausdr>. e=2.7182818.
INT(<ausdr>)	ganzzahliger Anteil des Wertes von <ausdr>
LOG(<ausdr>)	natürlicher Logarithmus
ROUND(<ausdr>,<n>)	rundet den Wert von <ausdr> auf n Dezimalstellen.
SQRT(<ausdr>)	Quadratwurzel

Stringmanipulation

&<mem_var>	Makroersetzung. Für &<mem_var> wird der Inhalt vom mem_var im Text ersetzt.
AT(<string_ausdr 1>,<string_ausdr 2>)	sucht die Teilzeichenkette 1 in der Zeichenkette 2 und liefert deren Position.
LOWER(<string_ausdr>)	formt alle großen Buchstaben der Zeichenkette in kleine Buchstaben um.
SPACE(<ausdr>)	liefert eine Zeichenkette, die aus so vielen Leerzeichen besteht, wie durch die Berechnung von <ausdr> ermittelt wird.
SUBSTR(<string_ausdr>,<anfangs_pos>[,<anzahl>])	liefert die Teilzeichenkette des ermittelten Strings, die auf der <anfangs_pos> beginnt und aus <anzahl> Zeichen besteht.
TRIM(<string_ausdr>)	Abschneiden aller Leerzeichen am Ende der Zeichenkette
UPPER(<string_ausdr>)	formt alle kleinen Buchstaben der Zeichenkette in große Buchstaben um.

Datums- und Zeitfunktionen

CDOW(<dat_ausdr>) liefert den Wochentag als Zeichenkette.
CTOD(<string_ausdr>) konvertiert eine Zeichenkette in ein Datum.
CMONTH(<dat_ausdr>) liefert den Monat als Zeichenkette.
DATE() liefert das Datum, so wie es im Betriebssystem eingestellt ist.
DAY(<dat_ausdr>) liefert den Tag des Monats als numerischen Wert.
DOW(<dat_ausdr>) liefert den Wochentag als numerischen Wert.
DTOC(<dat_ausdr>) liefert das Datum als Zeichenkette.
MONTH(<dat_ausdr>) liefert die Nummer des Monats als numerischen Wert.
TIME() liefert die Zeit, so wie sie im Betriebssystem eingestellt ist.
YEAR(<dat_ausdr>) liefert die Jahreszahl als vierstelligen numerischen Wert.

Konvertierungsfunktionen

ASC(<zeichen>) liefert den numerischen Wert des ASCII-Kodes des Zeichens.
CHR(<cardinal_byte>) liefert das Zeichen, dessen ASCII-Wert der Kardinalzahl entspricht.
STR(<ausdr>[,<länge>[,<dez>]])
konvertiert den numerischen Wert in eine Zeichenkette nach dem angegebenen Format.
VAL(<string_ausdr>) konvertiert eine Zeichenkette in einen numerischen Wert.

Sonstige Funktionen

BOF() zeigt an, ob der Dateizeiger auf den Fileanfang zeigt.
COL() Spaltennummer des Kursors auf dem Bildschirm
DELETED() zeigt an, ob der aktuelle Datensatz gestrichen ist.
FILE(<string_ausdr>) zeigt an, ob ein File mit dem angegebenen Namen existiert.
LEN(<string_ausdr>) Länge der Zeichenkette
PCOL() Spaltennummer des Kursors auf dem Drucker
PROW() Zeilennummer des Kursors auf dem Drucker
RECNO() Nummer das aktuellen Datensatzes
ROW() Zeilennummer des Kursors auf dem Bildschirm
TYPE("<ausdr>") liefert den Typ des Wertes des Ausdrucks als ein Zeichen.

Funktionen und ihre Typen

Funktion	Typ des Arguments	Typ des Ergebnisses
&	Speichervariable	String
ASC	String	numerisch
AT	String, String	numerisch
BOF	—	logisch
CDOW	Datum	String
CHR	numerisch	String
COL	—	numerisch
CTOD	String	Datum
CMONTH	Datum	String
DATE	—	Datum
DAY	Datum	numerisch
DELETED	—	logisch
DOW	Datum	numerisch
DTOC	Datum	String
EXP	numerisch	numerisch
FILE	String	logisch
INT	numerisch	numerisch
LEN	String	numerisch
LOG	numerisch	numerisch
LOWER	String	String
MONTH	Datum	numerisch
PCOL	—	numerisch
PROW	—	numerisch
RECNO	—	numerisch
ROUND	numerisch, ganze Zahl	numerisch
ROW	—	numerisch
SPACE	String	String
SQRT	numerisch	numerisch
STR	numerisch, ganze Zahl	String
SUBSTR	String	String
TIME	—	String
TRIM	String	String
TYPE	String	String
UPPER	String	String
VAL	String	numerisch
YEAR	Datum	numerisch

Anhang 2:

Kursorsteuerung im Direktmodus

In dBASE III können die folgenden Kommandos im Direktmodus arbeiten: APPEND, BROWSE, CHANGE, CREATE, CREATE LABEL, CREATE REPORT, EDIT, SET, MODIFY COMMAND, MODIFY LABEL, MODIFY REPORT, MODIFY STRUCTURE. Wir geben hier eine Zusammenfassung der Kursorsteuerung in diesen Kommandos. Bei der Besprechung der meisten Kommandos wurden Bilder angegeben, die die Hilfsmenüs enthalten und damit die wichtigsten Möglichkeiten der Kursorsteuerung wiedergeben. Voraussetzung ist eine Standardtastatur mit den Funktionstasten der 4 Pfeile und den Tasten **Home**, **End**, **PgUp** und **PgDn** sowie den Tasten **Enter** oder **Return**, **Del**, **Esc** und der Löschtaste <--. Sind diese Tasten nicht vorhanden, so können auch die aus dBASE II bekannten Kombinationen mit **Ctrl** (im folgenden dargestellt durch '^') und einem anderen Zeichen benutzt werden.

Taste	Ersatz	Wirkung
↑	^E	Eine Zeile oder ein Feld nach oben oder in BROWSE zum vorherigen Satz
↓	^X	Eine Zeile oder ein Feld nach unten oder in BROWSE zum vorherigen Satz
←	^S	Ein Zeichen nach links. Ist durch ^Home ein zusätzliches Menü eingeschaltet, so Auswahl
→	^D	Ein Zeichen nach rechts. Ist durch ^Home ein zusätzliches Menü eingeschaltet, so Auswahl
^←	^Z	In REPORT: verschiebt den Bildschirmausschnitt nach links. In MODIFY COMMAND: Kursor zum linken Rand
^→	^B	In REPORT: verschiebt den Bildschirmausschnitt nach rechts. In MODIFY COMMAND: Kursor zum Zeilenende
<--		Löscht das Zeichen links vom Kursor
Del	^G	Löscht das Zeichen unter dem Kursor
End	^F	Ein Wort nach rechts. In BROWSE: ein Feld nach rechts
^**End**	^W	Abspeichern aller Änderungen und verlassen des Direktmodus. Bem.: Verlassen wir in EDIT den Gültigkeitsbereich, so ist die Wirkung wie bei ^**End**
Esc	^Q	Verlassen des Direktmodus. Die Änderungen werden verworfen und der Zustand vor der Korrektur wird wiederhergestellt. In APPEND und BROWSE: Nur die Änderungen im letzten Satz werden verworfen. Die anderen sind gültig
Home	^A	Ein Wort nach links. In BROWSE: Ein Feld nach links
^**Home**		In BROWSE, LABEL, REPORT: Schaltet ein zusätzliches Menü ein und aus In EDIT, CHANGE, APPEND, wenn der Kursor in einem Memo-Feld steht: Aufruf des Editors für die Bearbeitung von Memo-Texte. Rückkehr mit ^**End**
Ins	^V	Schaltet den Einfügemodus ein und aus

PgUp	^R	In BROWSE: Blättern rückwärts
PgDn	^C	In BROWSE: Blättern vorwärts
Enter		Zum nächsten Feld. In APPEND als erstes Zeichen im ersten Feld: Beenden von APPEND. In EDIT im letzten Feld des letzten Satzes: wie ^**End**. In MODIFY COMMAND, wenn der Einfügemodus eingeschaltet ist: Zeile einfügen. In Menüs: Auswahl einer Option
	^N	In MODIFY STRUCTURE: Einfügen eines Feldes. In MODIFY COMMAND: Einfügen einer Zeile
	^T	Löscht das Wort rechts vom Kursor
	^U	Schaltet die Löschmarkierung für einen Satz ein und aus. In MODIFY REPORT, MODIFY STRUCTURE: Löscht die Felddefinition
	^Y	Löscht den Inhalt des Feldes. In MODIFY COMMAND: Löscht eine Zeile
	^KR	In MODIFY COMMAND: Fügt ein File an der Kursorposition ein
	^KW	In MODIFY COMMAND: Schreibt den gesamten Text in ein File

Außerdem wirken einige Steuerzeichen auch auf der Kommandoebene von dBASE:

←	^H	Löscht das Zeichen links vom Kursor
Enter	^M	Beendet eine Kommandozeile
	^P	Schaltet den Drucker logisch ein und aus (hard copy)
	^S	Stoppt die Bildschirmausgabe. Diese kann durch Drücken einer beliebigen Taste fortgesetzt werden
	^X	Löscht die Kommandozeile

Anhang 3:

Erweiterungen in dBASE III +

Ein * kennzeichnet Sprachelemente für die Arbeit im Rechnernetz.

Kommandos

@ <zeile_1>,<spalte_1> [CLEAR] TO <zeile_2>,<spalte_2> [DOUBLE]
Erstellen eines Rechtecks (Box) auf dem Bildschirm.

&&
Leitet einen Kommentar in Kommandozeilen ein.

APPEND FROM <filename> [FOR¦WHILE <bedingung>]
[[TYPE] SDF¦DIF¦SYLK¦WKS] [DELIMITED [WITH BLANK¦<zeichen>]]
SDF (Textfile), DIF (VisiCalc-File), SYLK (Multiplan-File) und WKS (Lotus 1-2-3-File).

BROWSE [FIELDS <feldliste>] [LOCK <n_ausdr>] [WIDTH <n_ausdr>]
[FREEZE <feld>] [NOAPPEND] [NOMENUE]
FREEZE läßt nur das angegebene Feld zur Korrektur zu. WIDTH bestimmt die Editierlänge der Felder. NOAPPEND verhindert das Anfügen von Sätzen, NOMENUE unterdrückt das Hilfsmenü und NOFOLLOW hält den Dateizeiger auf dem aktuellen Datensatz fest.

CALL <binärfile> [WITH <c_ausdr>¦<speicher_var>]
Aufruf eines Programms, welches durch LOAD geladen wurde.

CLEAR [ALL¦FIELDS¦GETS¦MEMORY¦TYPEAHEAD]
FIELDS löscht die Einstellung zur Anzeige von Feldern, die durch SET FIELDS TO ausgewählt wurden.
TYPEAHEAD löscht den Inhalt des Eingabepuffers der Tastatur.

COPY TO <filename> [<gb>] [FIELDS <feldliste>]
[FOR¦WHILE <bedingung>]
[SDF¦DIF¦SYLK¦WKS¦DELIMITED [WITH BLANK¦<zeichen>]]

CREATE¦MODIFY LABEL <filename>¦?

CREATE¦MODIFY QUERY <filename>¦?

CREATE¦MODIFY REPORT <filename>¦?

CREATE¦MODIFY SCREEN <filename>¦?
Erzeugen oder Korrektur eines Formatfiles (.scr) für Eingabekommandos im Direktmodus (SET FORMAT TO).

CREATE¦MODIFY VIEW <filename>¦?
Erzeugen, besichtigen oder Korrektur eines Files (.vue), das die Verbingungen zwischen Datenbankfiles beschreibt (SET RELATION).

CREATE VIEW <filename> FROM ENVIRONMENT
Erzeugen eines VIEW-Files (.vue), das den aktuellen Zustand der Datenbank beschreibt.

DISPLAY HISTORY [LAST <n_ausdr>] [TO PRINT]
Ausgabe der zuletzt eingegebenen Kommandos.

DISPLAY USERS
* Anzeige der parallel arbeitenden Nutzer im Rechnernertz.

EDIT [<gb>] [FIELDS <feldliste>] [FOR¦WHILE <bedingung>]
Editieren eines Datensatzes im Direktmodus.

ERASE <filename>¦?

EXPORT TO <filename> TYPE PFS
Konvertierung eines Datenbankfiles in ein File im PFS-Format,

welches von den Programmen VisiCalc, Multiplan und Lotus 1-2-3 verarbeitet werden kann.

FIND <c_ausdr>¦<zahl>

IMPORT FROM <filename> TYPE PFS

Erzeugen eines Datenbank-, Format- und View-Files aus einem File im PFS-Format (vgl. EXPORT).

INDEX ON <ausdr> TO <filename> [UNIQUE]

UNIQUE bewirkt, daß jeder Schlüssel nur einmal für den ersten entsprechenden Datensatz aufgenommen wird.

LABEL FORM <filename> [SAMPLE] [<gb>] [FOR ¦WHILE <bedingung>] [TO PRINT] [TO FILE <filename>]¦?

Ausgabe von Briefadressen.

LIST HISTORY [LAST <n_ausdr>][TO PRINT]

Ausgabe der zuletzt eingegebenen Kommandos.

LOAD <binärfile>

Laden eines ausführbaren Files (.bin) in den Speicher.

LOGOUT

* Abmeldung eines Nutzers bei der Arbeit im Netz.

MODIFY QUERY <filename>

MODIFY VIEW <filename>

ON ERROR¦ESCAPE¦KEY <kommando>

Anschluß einer nutzereigenen Prozedur zur Behandlung von dBASE-Fehlern bzw. zur Reaktion auf eine gedrückte Taste.

RELEASE MODULE <binärfile>

Löschen eines mit LOAD geladenen Files im Speicher.

REPORT FORM <filename> [<gb>] [FOR¦WHILE <bedingung>] [PLAIN] [HEADING <c_ausdr>] [NOEJECT] [TO PRINT] [TO FILE <filename>]¦? [SUMMARY]

Ausgabe eines Berichts.

RETRY

* Rückgabe der Programmsteuerung an die aufrufende Befehlszeile.

RESUME

Fortsetzen eines mit SUSPEND unterbrochenen Programms.

SELECT <arbeitsbereich>¦<alias>¦?

Auswahl eines Arbeitsbereichs.

SET CATALOG ON¦OFF

Aktivieren der Ausgabe in den Katalog.

SET CATALOG TO <filename>¦?

Definition eines Katalogfiles (.cat).

SET CENTURY ON¦OFF

Azeige des Jahrhunderts bei Datumsangaben.

SET COLOR ON¦OFF

Auswahl zwischen Color- (ON) und Monochrom-Bildschim (OFF).

SET COLOR TO <std_display>[.<erw_display>] [<rand>] [<hintergrund>]

Auswahl der Anzeigearten und Farben auf dem Bildschirm.

SET DATE AMERICAN¦ANSI¦BRITISH¦ITALIAN¦FRENCH¦GERMAN

Definition der Datumsform.

SET DOHISTORY ON¦OFF

Internes Aufzeichnen aller Kommandos eines Kommandofiles.

```
SET ENCRYPTION ON¦OFF
*    Datenverschlüsselung
SET EXCLUSIVE ON¦OFF
*    Festlegen des ausschließlichen Zugriffs auf Files.
SET FIELDS ON¦OFF
     Die in SET FIELDS TO definierte Feldliste  wird wirksam (ON).
SET FIELDS TO [<feldliste>]¦[ALL]
     Definition der anzuzeigenden Felder.
SET FILTER TO [<bedingung>]¦[FILE <.qry_filename>]
SET FORMAT TO [<.fmt_filename>]¦[<.scr_filename>¦?]
SET HISTORY ON¦OFF
     Schaltet die interne Aufzeichnung einer History ein (ON).
SET HISTORY TO <n_ausdr>
     Definition der Zeilenzahl der intern geführten History.
SET INDEX TO [<fileliste>]¦?
SET MEMOWIDTH TO <n_ausdr>
     Darstellungsbreite von Memo-Feldern (Standard: 50).
SET MESSAGE TO [<c_ausdr>]
     Anzeige des Texts in der untersten Bildschirmzeile.
SET ORDER TO <n_ausdr>
     Auswahl eines bei SET INDEX TO angegebenen Indexfiles.
SET PRINTER TO
*    Drucker-Spooler leeren und auf Standardgerät zurücksetzen.
SET PRINTER TO <DOS_gerät>
     Verbinden der Druckerausgabe mit einem log. Geräte (LPT1).
SET PRINTER TO \\<rechner>\<drucker>=<ziel>
*    Verbinden der Druckerausgabe mit einem Drucker im Netz.
SET PRINTER TO \\SPOOLER
*    Druckerausgabe an einen Drucker im Netz senden.
SET RELATION TO [<schlüssel_ausdr>¦RECNO()¦<ausdruck> INTO <alias>]
     Verbinden von Datenbankfiles.
SET SCOREBOARD ON¦OFF
     Ausgabe einer Statusmeldung auf dem Bildschirm in Zeile 0.
SET STATUS ON¦OFF
     Ausgabe einer Statusmeldung auf dem Bildschirm in Zeile 22.
SET TITEL ON¦OFF
     Anforderung eines Katalogtitels beim Erstellen eines Katalogs.
SET TYPEAHEAD TO <n_ausdr>
     Größe des Eingabepuffers für Eingaben von der Tastatur (20).
SET VIEW TO <filename>¦?
     Eröffnen eines View-Files (.vue).
SORT TO <neues_file> ON <feld_1> [/A] [/C] [/D] [,<feld_2>
     [/A] [/C] [/D] ...] [<gb>] [FOR¦WHILE <bedingung>]
     /C - keine Unterscheidung zwischen Groß- und Kleinbuchstaben.
SUSPEND
*    Aussetzen der Abarbeitung eines Programms.
UNLOCK [ALL]
*    Zugriffsschutz für Files und Sätze aufheben.
USE [<filename>] [INDEX <fileliste>] [ALIAS <alias>]¦? [EXCLUSIVE]
*    EXCLUSIVE sichert den ausschließlichen Zugriff im Netz.
```

Funktionen und ihre Typen

	Funktion	Ergebnistyp	Bemerkung
	ABS(<n_ausdr>)	numerisch	absoluter Wert
*	ACCESS()	numerisch	Zugriffsebene ermitteln
	DBF()	String	Name des aktuellen .dbf-Files
	DISKSPACE()	numerisch	freier Platz auf akt. Laufwerk
	ERROR()	numerisch	liefert Fehlerkode
	FIELD(<n_ausdr>)	String	liefert den Feldnamen
	FKLABEL(<n_ausdr>)	String	Funktionstastenbezeichnung
	FKMAX()	numerisch	Anzahl der Funktionstasten
*	FLOCK()	logisch	sperren eines Files und Erfolgsmeldung
	FOUND()	logisch	Ergebnis einer Suchoperation
	GETENV(<c_ausdr>)	String	Environment des Betriebssystems
	IIF(<bedingung>,<ausdr_1>,<ausdr_2>)	Datum¦numerisch¦String	bedingter Ausdruck
	INKEY()	numerisch	Kode einer Taste
	ISALPHA(<c_ausdr>)	logisch	.T., wenn erstes Zeichen Buchstabe
	ISCOLOR()	logisch	.T. für CGA; .F. für Monochrom
	ISLOWER(<c_ausdr>)	logisch	.T., wenn erstes Zeichen Kleinbuchstabe
	ISUPPER(<c_ausdr>)	logisch	.T., wenn erstes Zeichen Großbuchstabe
	LEFT(<c_ausdr>,<n_ausdr>)	String	linksbündiger Substring
*	LOCK()	logisch	sperren des aktuellen Satzes
	LTRIM(<c_ausdr>	String	entfernen führender Leerzeichen
	LUPDATE()	Datum	Datum der letzten Korrektur
	MAX(<n_ausdr_1>,<n_ausdr_2>)	numerisch	Maximum von zwei Werten
	MESSAGE()	String	Fehlermeldung
	MIN(<n_ausdr_1>,<n_ausdr_2>)	numerisch	Minimum von zwei Werten
	MOD(<n_ausdr_1>,<n_ausdr_2>)	numerisch	Rest bei ganzzahliger Division
	OS()	String	Name des Betriebssystems
	READKEY()	numerisch	Kode der Taste, mit der der Direktmodus verlassen wurde
	RECCOUNT()	numerisch	Sätze des aktiven .dbf-Files
	RECSIZE()	numerisch	Länge eines Datensatzes
	REPLICATE(<c_ausdr>,<n_ausdr>)	String	vervielfachen eines Strings
	RIGHT(<c_ausdr>,<n_ausdr>)	String	rechtsbündiger Substring
*	RLOCK()	logisch	sperren des aktuellen Satzes
	RTRIM(<c_ausdr>)	String	entfernen nachfolgender Leerzeichen
	STUFF(>c_ausdr_1>,<startpos>,<anzahl>,<c_ausdr_2>)	String	einfügen/ersetzen eines Strings
	TRANSFORM(<c_ausdr>¦<n_ausdr>,<maske>)	String	Ausgabe im PICTURE-Format
	VERSION()	String	dBASE III+-Version

Literaturverzeichnis

[Ash85.1]
dBASE III Plus: Quick Reference Guide. Torrance, California: Ashton-Tate 1985

[Ash85.2]
dBASE III Plus: Programming with dBASE III Plus. Torrance, California: Ashton-Tate 1985

[Ash85.3]
dBASE III Plus: Learning and Using dBASE III Plus. Torrance, California: Ashton-Tate 1985

[Bat87]
Bates, W.; Fortino, A. G.: dBASE III Plus im lokalen Netzwerk. München: Markt & Technik 1987

[Bye87]
Byers, R. A.; Long, J.; Ratliff, J. W.: Tools für dBASE III Plus: Utilities. München: Markt & Technik 1987

[Cas85]
Castro, L.; Hanson, J.; Rettig, T.: Advanced Programmer's Guide. Featuring dBASE III and dBASE II. Culver City, California: Ashton-Tate 1985

[Cas87]
Castro, L.; Hanson, J.; Rettig, T.: Das dBASE-Kompendium. München: Markt & Technik 1987

[Cli85]
Clipper Compiler Anwender-Handbuch. Culver City: Nantucket Corporation 1985

[Cod70]
Cod, E. F.: A Relational Model for Large Shared Data Banks. Comm. ACM, Vol.13, 6/1970

[Coo87.1]
Cooper, J. T.; & Company: Tools für dBASE III Plus: Grafik. München: Markt & Technik 1987

[Coo87.2]
Cooper, J. T.; & Company: Tools für dBASE III Plus: C-Routinen. München: Markt & Technik 1987

[Coo87.3]
Cooper, J. T.; & Company: Tools für dBASE III Plus: Turbo-Pascal. München: Markt & Technik 1987

[Egg86.1]
Eggerichs, W.: dBASE II. Band I Einführung. Berlin: Verlag Technik 1986

[Egg86.2]
Eggerichs, W.: dBASE II. Band II Programmierung. Berlin: Verlag Technik 1986

[Här87]
Härder, T.: Überblick zum Heft it 3/87 - Datenbanken. München: Oldenbourg-Verlag, Zeitschrift Informationstechnik (it), Heft 3/87

[Kon87]
Konzack, U.: Werkzeuge einer Programmierumgebung für die Modellierung und Simulation in Prolog. Dissertation. Berlin: Humboldt-Universität 1987

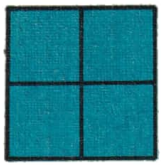
[Mir87]
Mirecki, T.: Dialects of dBASE. In PC TECH Journal April 1987

[Smo87]
Smode, D.: Das große MS-DOS-Profi-Handbuch. München: Franzis-Verlag 1987

[Wed83]
Wedekind, H.: Datenbanksysteme I. Mannheim, Wien, Zürich: Wissenschaftsverlag 1983

[Zeh83]
Zehnder, C. A.: Database Techniques for Professional Workstations. Zürich: Eidgenössische Technische Hochschule, Institut für Informatik 1983

Sachwörterverzeichnis



Was leistet dBASE III als Datenbanksystem auf 16-Bit-Rechnern? Wo sind seine Grenzen? Welche Unterschiede bestehen zu dBASE II? Wie komme ich von dBASE II zu dBASE III? Ist ein Datenaustausch zwischen dBASE III und anderen Programmen möglich? Kann dBASE III mit anderen Programmen zusammenarbeiten? Wie kommt man von dBASE III in andere Datenbanksysteme?
Diese und weitere Fragen werden in diesem Buch systematisch und anhand ausgetesteter Beispiele beantwortet. Zur besseren Übersicht sind die Kommandos und Funktionen von dBASE III alphabetisch zusammengestellt.

ISBN 3-341-00703-2