87-3

MIKRO + KLEIN COMPUTER

DAS SCHWEIZER FACHMAGAZIN FÜR KLEINE UND MITTLERE COMPUTERSYSTEME Fr. 8.–

RS232-Schnittstelle für Turbo-Pascal

Prozess-orientiertes Programmieren

ES STECKT NOCH MEHR IN IHREM MULTISYNC™!

Sehr viele EGA-Karten kommen bis hierher

Einige kommen darüber hinaus

Jetzt holen Sie noch mehr aus Ihrem Multisync™: die SuperEGA™ von GENOA bringt Sie bis hierhin

Und so nutzen Sie ihn ganz aus: mit der SuperEGA HiRes™ von GENOA

15.75 KHz	18.75 KHz	21.8 KHz	30.0 KHz	31.5 KHz	35.0 KHz
CGA 320 x 200	**MDA (Hercules) 720 x 348**	**EGA 640 x 350 1056 x 352 132 col. x 44**	**PGA resolution 640 x 480** Drivers for: Lotus 1-2-3, AutoCAD, GEM, Windows, and Desktop Publishing	**CGA DoubleScan: 320 x 400 640 x 400**	**CAD/CAM 800 x 600** Drivers for: AutoCAD, Windows, GEM, and Desktop Publishing

Schöpfen Sie Ihren MultiSync™ oder kompatiblen Monitor voll aus, bis zu 800x600 Punkten, mit einer SuperEGA-Karte von GENOA.

Die SuperEGA™ von GENOA bietet Ihnen CGA, MDA (inkl. Hercules) und EGA-Modus, alle Modi voll kompatibel, ohne Software-Emulation. Zusätzlich dem PGA entsprechende 640x480 und DoubleScan. Damit lösen Sie normale CGA-Software mit bis zu 640x400 Punkten auf. 132 Zeichen in bis zu 44 Zeilen können Sie selbstverständlich auch auf Ihrem Bildschirm darstellen. Und fürs Desktop Publishing stehen Ihnen 80 Zeichen mal 66 Zeilen zur Verfügung.

Wenn Sie noch höher auflösen wollen, dann bietet Ihnen die SuperEGA HiRes™ darüber hinaus 800x600 Punkte bei 16 Farben für CAD/CAM und Desktop Publishing.

Beide SuperEGA-Karten synchronisieren automatisch. Mit allen MultiSync-Frequenzen zwischen 15,75 und 35 kHz.

Für alle Grafik-Anwendungen können Sie auf Ihre SuperEGA zählen. Wir holen mehr aus Ihrem MultiSync™ oder kompatiblen Monitor.

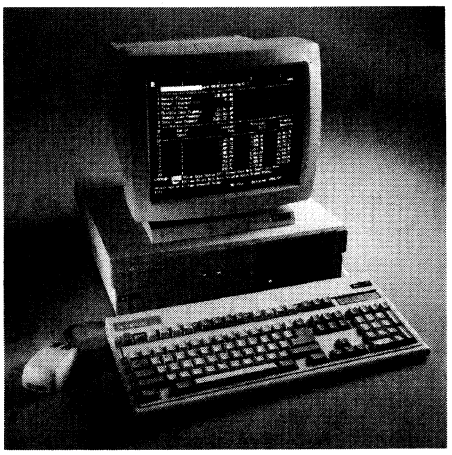
Und selbstverständlich können Sie auch einen Monochrom-, Color- oder EGA-Monitor anschließen.

Electronic Marketing AG

Your Swiss Distributor for high technology.

Electronic Marketing AG · Bahnhofstr. 60 · 4132 Muttenz-Basel · Tel. 061-61 53 53 · Tlx 965669 · Fax 061-42 45 49

DAS LESEN SIE IN DIESER NUMMER



Der BULL MICRAL 40 besitzt als zentrale Verarbeitungseinheit den Intel 80286 Mikroprozessor mit 6 oder 8 MHz Taktfrequenz, verfügt über eine Hauptspeicherkapazität von 640 KB, einen Datenbus sowie drei freie Erweiterungspositionen (zwei 16-Bit, eine 8-Bit) für Adapterkarten. Die beiden Basisversionen, in denen der neue Mikrocomputer angeboten wird, entsprechen dem jeweiligen Einsatzbereich: Als Einzelplatz-Rechner mit 5.25-Zoll-Diskettenlaufwerk (1,2 MB) und einem Laufwerk für 5.25-Zoll-Festplatte (20 MB); als Netzwerk-Station, ohne Speicherperipherie, jedoch durch einfache Integration spezieller Kits für Disketteneinheiten erweiterbar. Die Festplatten- und Diskettenlaufwerke sind im platzsparenden slim-line-Design konstruiert und beanspruchen keinen Einschub auf dem vertikal angeordneten Bus. Auf der Vorderseite des Gerätes befindet sich die Vorrichtung für den Einbau eines Lesegerätes für BULL CP8-Chipkarten. Durch diese neuartige und nützliche Option haben Benutzer die Möglichkeit, Spezial-Anwendungen mit dieser «intelligenten» Karte zu übernehmen bzw. Zugriffs- und Datenschutzvorkehrungen zu treffen. Erhältlich ist der neue BULL-Mikrocomputer als Einzelplatz-PC bereits für weniger als Fr. 7'300.-- (Konfiguration mit Diskettenlaufwerk, 20 MB Platte, Monochrom-Grafik-Bildschirm und Tastatur) und als Netzwerk-Arbeitsstation ab Fr. 4'618.--. Info: BULL (Schweiz) AG, Wengistrasse 28, 8021 Zürich, Tel. 01/242'12'33. □

Ausgabe Juni 1987
Erscheint 6mal pro Jahr
9. Jahrgang

COMPUTER aktuell

EPSON LQ-2500 und SQ-2500 7

Floppy Disks unter der Lupe 14

Verbesserte LCD-Technologie beim portablen Zenith PC Z-181 18

Computerszene 21

LAN mit Molecular 16/300 31

SuperDesk, die komplette Bürosoftware (3) 33

Softstrip Technologie 37

Kompatibilität für unterwegs: Bondwell 08 41

LEHRGÄNGE

Vom Umgang mit dBase III PLUS (2) 43

Künstliche Intelligenz (5) 53

GEWUSST WIE

Serielle Superschnittstelle für Turbo-Pascal 59

Spline-Interpolation einmal anders 67

Prozess-orientiertes Programmieren in Logo 73

Universelles EPROM-Programmiergerät für den Commodore C-64 85

COMPUTER-BÖRSE

Fundgrube für günstige Occasionen 92

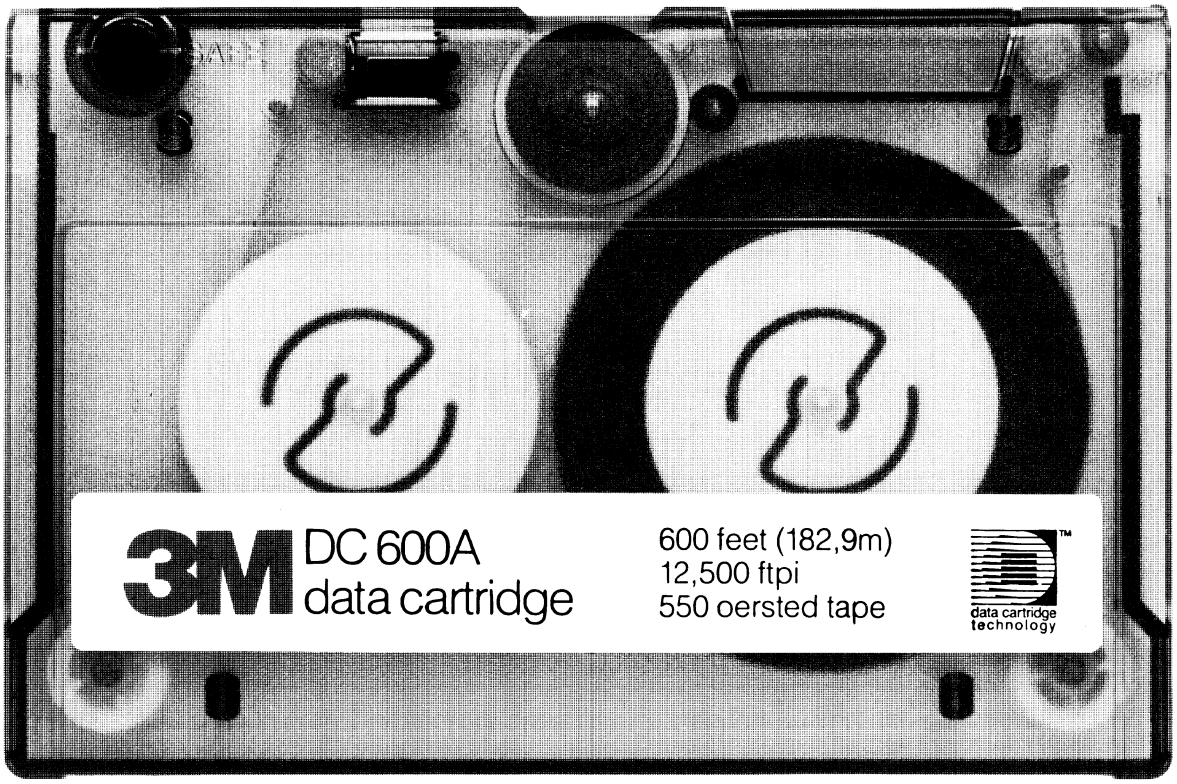
RUND UM DEN PC

Aktuelles vom Industriestandard 94

VORSCHAU 98

Das Original von 3M, an das keiner herankommt.

Data Cartridges garantieren eine absolut zuverlässige Datensicherung. Dies besonders kostengünstig und 100%-ig fehlerfrei. Wer dabei auf Nummer



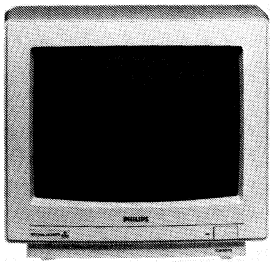
Eines der Cartridge-Originale von 3M in Original-Grösse. Daneben gibt es auch Mini-Versionen. Je nach Typ lassen sich damit 10 bis 70 Megabyte formatierte Daten speichern. Der Back-up im Streamerverfahren z.B. einer 10 MB-Disc dauert so nur wenige Minuten. Was nicht nur äusserst schnell, sondern auch absolut sicher ist. Dafür garantiert nämlich auch bei hohen Zugriffsgeschwindigkeiten der patentierte Präzisionsantrieb aller 3M Data Cartridges. Kommt dazu, dass Sie damit die Sicherung eines Megabyte fast konkurrenzlos günstig zu stehen kommt.

Sicher gehen will, vertraut am besten dem Know-how des Data Cartridges-Erfinders und setzt auf die Originale von 3M. Denn die sind heute weltweit zum Standard geworden.

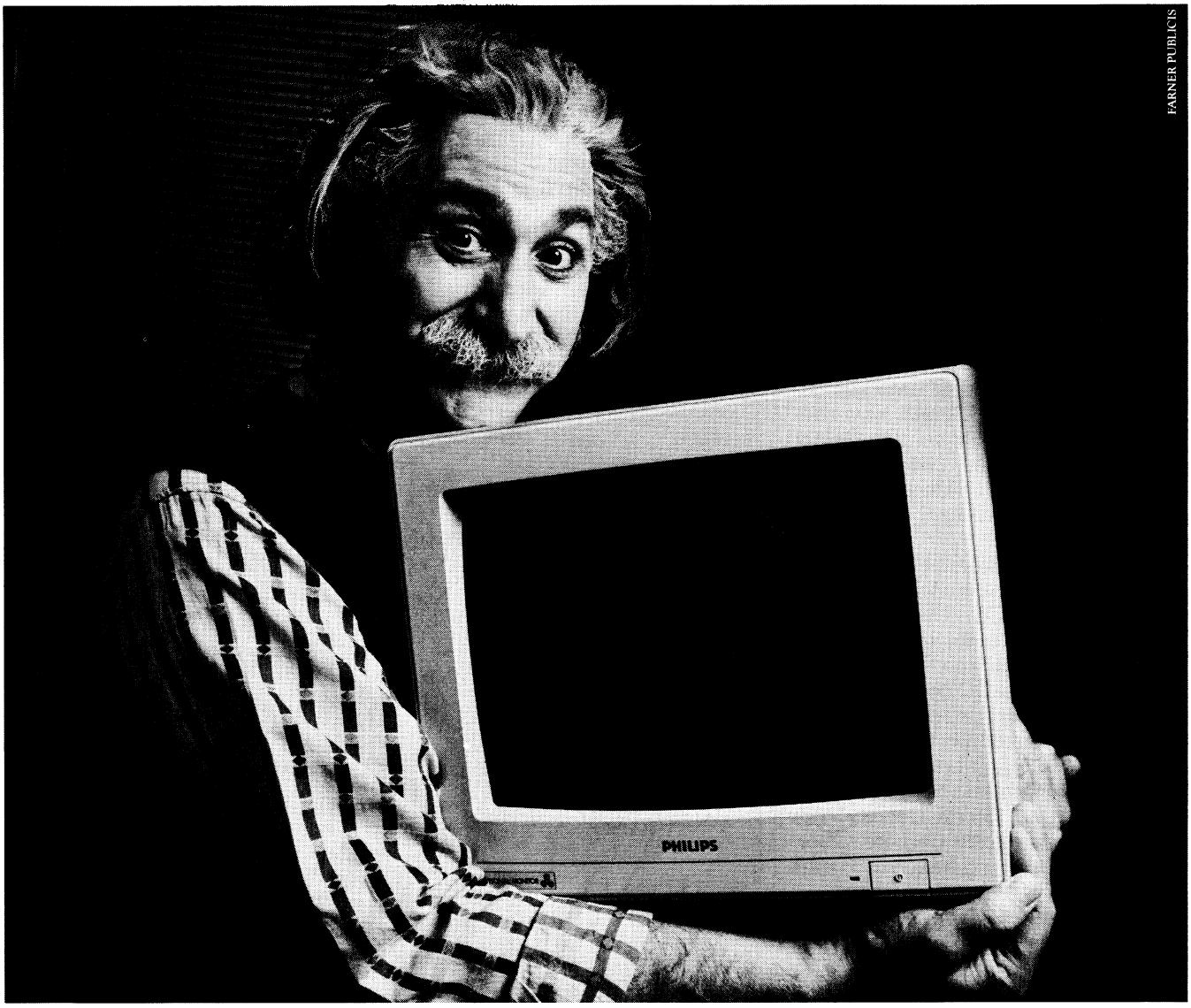
3M Cartridges sind im EDV- und Bürofachhandel erhältlich.

3M

3M (Schweiz) AG
EDV-Datenträger
Eggstrasse 93
8803 Rüschlikon
Telefon 01 724 93 32



PHILIPS HAT ÜBER DEN COMPUTER-Besitzer nachgedacht, der gern klarsehen will, wenn's um Gestaltung geht. Das Ergebnis ist ein hochauflösender Computer-Monitor von so bestechender Brillanz, dass einem das Hören vergeht. Und weil noch nicht jeder ein Philips Computer besitzt, verhilft der neue Monitor auch allen anderen IBM-kompatiblen Geräten zum besseren Durchblick. Philips Computer-Monitor. EIN BRILLANTER EINSTEIN.



PHILIPS

PHILIPS

Floppy Disks unter der Lupe

Wie kommt es, dass es bei einer 320 KByte-Diskette schon bei 200 KByte abgespeicherten Daten zu einem «Disk-full-error» kommen kann? Weshalb haben gleichgrosse Disketten oft sehr unterschiedlich grosse Speicherkapazitäten? Wieso sind Disketten gleicher Grösse nicht unbedingt kompatibel, d.h. austauschbar zwischen verschiedenen Diskettenlaufwerken?

Oliver Rosenbaum

So wie ein Laufwerk für eine bestimmte Diskettengrösse (8, 5,25, 3,5 oder 3 Zoll) gebaut wird, so ist auch ein System auf eine bestimmte Disketenkapazität ausgelegt. In der Regel ist es also nicht möglich, ohne weiteres die Speichermöglichkeit einer Diskette zu vergrössern. Diese Tatsache hat einen einfachen Grund: Das DOS (Disk Operating System) welches für die Verwaltung der Daten auf der Diskette verantwortlich ist, speichert diese nach vorgegebenen Formaten. Jeder PC-Benutzer weiss, dass neue Disketten vor ihrem ersten Gebrauch formatiert werden müssen. Das Betriebssystem liefert hierzu in der Regel ein Formatierungsprogramm. Dieses Dienstprogramm unterteilt die noch unbeschriebene Diskette in Sektoren nach einem vorgegebenen Muster, welches je nach Laufwerkshersteller sehr unterschiedlich sein kann in Anzahl und Lage der Sektoren. Auch die Zahl der Spuren (siehe Grafik) ist unterschiedlich und vor allen Dingen unveränderbar im Gegensatz zu den Sektoren. Diese können bei manchen Systemen den Anforderungen (an Grösse der Datensätze) angepasst werden.

Von alledem bekommt der Anwender wenig mit - schliesslich ist dies die Aufgabe des DOS. Die Diskettenkapazität ist also nicht veränderbar - auch nicht durch die Verwendung von DD-Disketten. DD bedeutet «Double Density», also doppelte Schreibdichte. Das funktioniert aber nur in dafür vorgesehenen Laufwerken. Die Diskette selbst unterscheidet sich von der SD-Diskette (Singe Density) nur durch eine qualitativ bessere Oberflächenbeschichtung oder durch eine Prüfung der Speicherfähigkeit.

Neben den Kategorien DD und SD gibt es auch noch die Bezeichnung SS und DS. Ersteres bedeutet «Single Sided«, also nur einseitig verwendbar. Folglich sind DS-diskette (Double Sided) von beiden Seiten zu nutzen, das bedeutet 100 % mehr Kapazität gegenüber einer SS-Diskette. Aber die Diskette muss zum Gebrauch der Rückseite herausgenommen werden, gewendet und wieder in das Laufwerk eingelegt werden, wenn das Laufwerk nicht mit zwei Schreib-Leseköpfen ausgestattet ist.

Der Aufbau der Diskette

Doch nun zum eigentlichen Aufbau der Diskette: Als Beispiel dient hier eine 5,25-Zoll-Diskette wie sie für 70 % aller gängigen Laufwerke verwendet werden kann. Formatiert besitzt diese 5,25-Zoll-Diskette ein Speicherformat von 39 Spuren aufgeteilt in jeweils 16 Sektoren mit 256 Bytes Kapazität/Sektor. Auf der Diskette haben also 39x16x256 = 159'744 Bytes (rund 160 KB) platz.

Die Sektorgrenzen sind statisch wie Bild 1 zeigt und die Anzahl der Sektoren ist konstant von der äusseren bis zu inneren Spur. Physikalisch nimmt die Spurlänge und damit die Sektorlänge von aussen nach innen zwar ab, aber das Zeitintervall, in der ein Sektor vom Schreib-Lesekopf überfolgen wird, ist immer gleich gross, obwohl sie physikalisch unterschiedlich gross sind.

Das bisher Gesagte erklärt schon die Tatsache der unterschiedlichen Speicherkapazitäten bei gleichgrossen Disketten: Je nach System werden die Disketten einfach enger beschrieben, oder die Sektoren und Spuren enger aneinandergelegt. Voraussetzung hierfür ist aber eine höhere Positioniergenauigkeit des Schreib/Lesekopfes. Laufwerke, die mehr Bytes auf eine Diskette packen können, sind in der Regel auch wesentlich aufwendiger in ihren mechanischen Bauteilen und damit teurer in der Anschaffung. In Bild 2 ist der schematische Aufbau eines Sektors dargestellt (Beispiel).

Allerdings ist der Sektor-Aufbau von System zu System verschieden. Das Beispiel zeigt einen Sektor des Commodore-DOS, wie er bei verschiedenen Laufwerkstypen angewendet wird.

Bei diesem System werden die Spuren mit einer unterschiedlichen Anzahl von Sektoren belegt:

Spur 1 - 17	21 Sektoren
Spur 18 - 24	19 Sektoren
Spur 25 - 30	18 Sektoren
Spur 31 - 35	17 Sektoren

Auf Spur 18 befindet sich hier das Directory der Diskette, welches alle

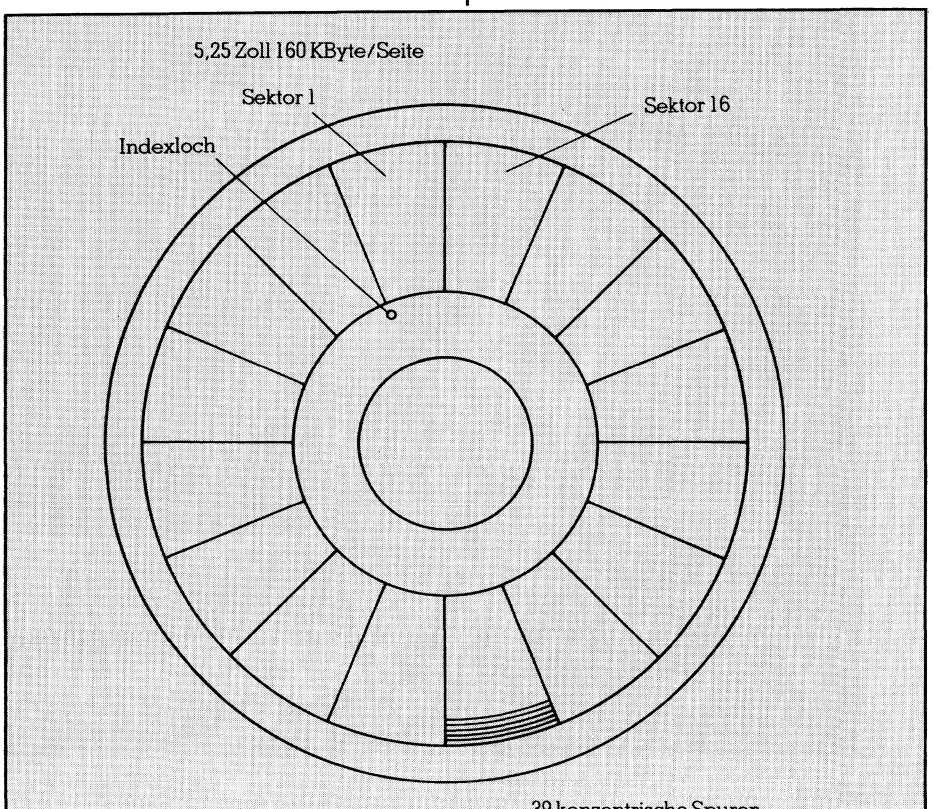


Bild 1

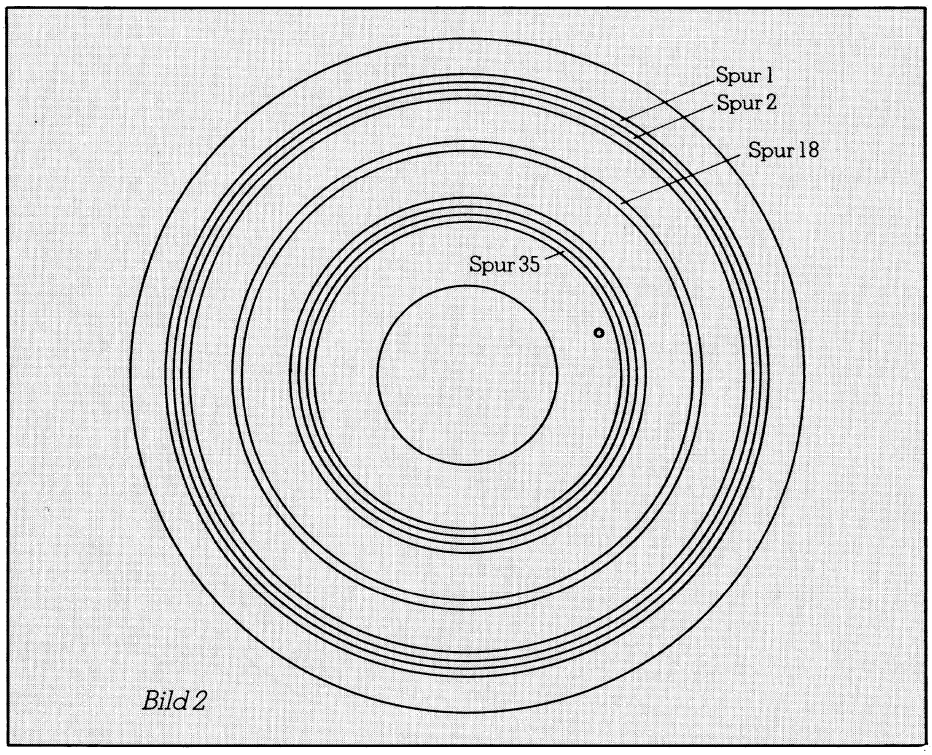


Bild 2

notwendigen Informationen über die Lage der einzelnen Files beinhaltet. Zwar sind die Spuren mit einer unterschiedlichen Anzahl von Sektoren belegt, diese sind aber ebenfalls statisch wie im ersten Beispiel (Bild 1).

Eine weitere Möglichkeit besteht in der variablen Blocklänge. Diese Methode hat den Vorteil, die Sektoren den tatsächlichen Anforderungen an die jeweilige Speicherkapazität anpassen zu können; die Gesamtkapazität der Diskette bleibt dabei unverändert.

Ein Sektor kann immer nur ganze Datensätze aufnehmen und so kann es passieren, dass bei ungünstigem Verhältnis von Blocklänge zu Satzlänge viel Speicherplatz ungenutzt bleibt und dann kann es durchaus sein, dass eine Diskette schon bei 200 KB abgespeicherten Daten einen «Disk-full-error» meldet, obwohl die Diskette für 320 KB ausgelegt ist.

Ein weiteres Beispiel: Disketten unter DOS 2.0 sind folgendermassen aufgebaut: Auf Spur 0 steht immer das Direktory (VTOC = Volume Table of Contens). Es belegt die gesamte Spur und dient der Verwaltung der einzelnen Dateien. Hier wird auch vermerkt, ob eine Spur belegt ist oder nicht.

Die Spuren, die von einer Datei belegt werden, müssen nicht der Nummer nach beieinander liegen, das gilt für fast alle Systeme, einschliesslich der Fest- und Wechselplatten.

Wenn das System für eine Datei eine freie Spur benötigt, nimmt es von den freien Spuren die mit der niedrigsten Nummer. Eine 8-Zoll-Diskette unter DOS 3.0 hat 77 Spuren. Die Spuren sind von aussen nach innen durchnumeriert. Falls die Diskette ein Betriebssystem enthält, steht dieses immer auf den Spuren 76 abwärts. Eine Spur hat hier brutto (Information und Verwaltung) 4'932 Byte. Bei der Diskette wird die Aufteilung einer Spur in Sektoren vom System vorgenommen; es benötigt dabei pro Sektor 60 Bytes zur Kennzeichnung und Trennung der Sektoren.

Beim Initialisieren (Formatieren) einer Diskette werden alle Spuren in Sektoren für Blöcke mit der Länge 488 (= Standard-Blocklänge) eingeteilt. Beim Anlegen einer Datei wird die Blocklänge der angegebenen Satzlänge angepasst, d.h. die Blocklänge liegt wie die Satzlänge im Bereich zwischen 32 und 2'047. Wird eine Satzlänge von 488 angegeben, dann wird die beim Initialisieren vorgenommene Einteilung beibehalten und nur die benötigten Spuren reserviert.

Bei anderen Satzlängen werden die benötigten Spuren in Sektoren für Blöcke mit der Länge: Blocklänge = Satzlänge umformatiert.

Beim Löschen einer Datei werden die freigegebenen Spuren in Sektoren für Blöcke mit einer Länge 488 umformatiert, so dass freie Spuren immer in Sektoren für Blöcke mit der Standardblocklänge eingeteilt sind. Dieser Aufwand wird bei anderen Systemen nicht getrieben. Die Anzahl der Sektoren pro Spur ist abhängig von der Blocklänge bzw. Satzlänge.

Hierzu gibt es eine Formel, welche die Anzahl der Sektoren pro Spur berechnet:

$n = 4932 / (60+L)$

n ist die Anzahl der Sektoren (ganzzahlig und abgerundet) und L ist die Block- bzw. Satzlänge. In der folgenden Tabelle sind häufig verwendete Satzlängen aufgelistet:

Satzlänge	Anzahl Sektoren/ Spur	Bytes/ Spur (netto)
32	53	1696
80	35	2800
128	26	3328
256	15	3840
488	9 (ohne Rest)	4392
512	8	4096
1024	4	4096
1584	3 (ohne Rest)	4752
2047	2	4094

Somit hat man bei der Satzlänge 1584 die beste Ausnutzung einer Spur mit 4'752 Bytes.

Neben der bisher beschriebenen «Softsektorierung» gibt es auch noch «hardsektorierte» Disketten. Softsektoriert bedeutet wie schon besprochen, dass die Sektoren mit einem Dienstprogramm auf die Diskette geschrieben werden, also softwaremässig. Diese Methode wird heute fast ausschliesslich angewandt.

Es gibt aber auch Disketten, bei denen die Sektoren hardwareseitig festgelegt sind (siehe Bild 4).

Die Festlegung erfolgt durch Sektorlöcher, welche die Sektorgrenzen festlegen. Sie werden wie das Indexloch der anderen Disketten im Laufwerk mit einer Lichtschranke abgetastet und signalisieren den Beginn

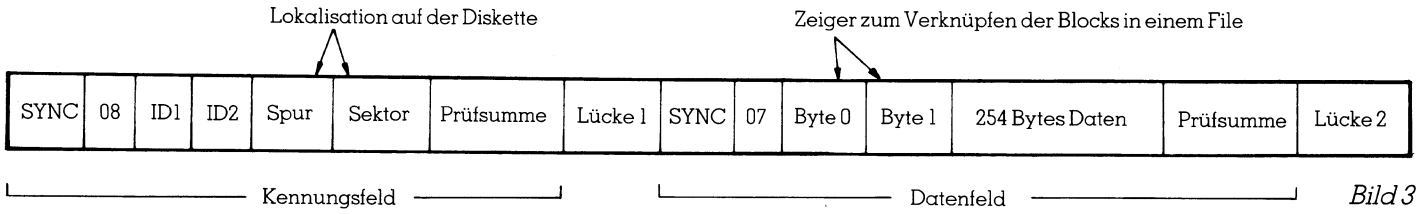


Bild 3

eines Sektors. Das Indexloch selbst dient lediglich der Synchronisation der Diskettenumdrehungen.

Tips zur Diskettenhandhabung

Zum Schluss noch zwei Tips zur Diskettenhandhabung: Disketten, welche für die Spurdichte von 48 Spuren pro Inch (tpi = tracks per inch) geprüft sind, also SD-Disketten, können in der Regel nicht fehlerfrei auf 96 tpi-Laufwerken eingesetzt werden. 96-tpi-Disketten funktionieren allerdings einwandrei auf 48-tpi-Laufwerken.

Zweiseitige Disketten sollen nicht als einseitige in Einkopf-Laufwerke verwendet werden, wenn sie anschliessend wieder in Doppelkopf-Laufwerken Einsatz finden. Der Andruckfilz, der anstelle des zweiten Kopfes die Magnetscheibe an den Magnetkopf drückt, könnte verschmutzt oder verhärtet sein und so die Beschichtung zerkratzen. Gleiches gilt für «Wendedisketten«. Hinzu kommt hierbei, dass einseitige (SS) Disketten auch nur einseitig geprüft werden, d.h. die Rückseite ist zwar theoretisch nutzbar, aber es können Fehler in der Beschichtung vorhanden sein.

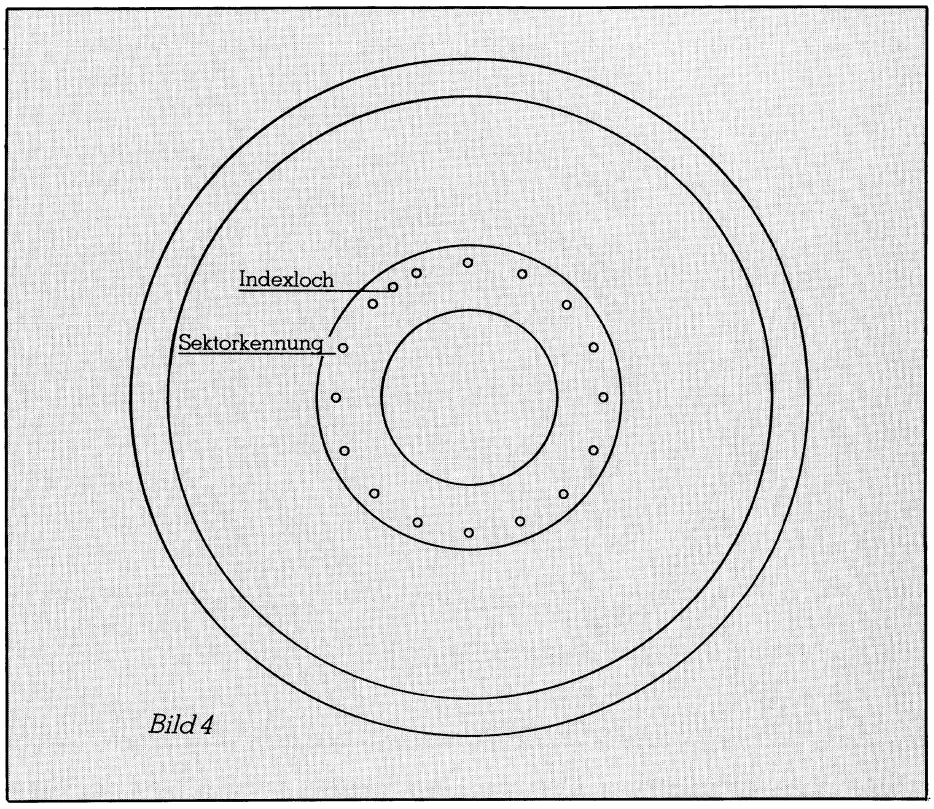


Bild 4

Bei guter Behandlung ist die Lebensdauer von Disketten fast unbegrenzt, zumindest leben sie meist länger als ein Gerät im Einsatz ist, bevor es von einer neuen Computergeneration abgelöst wird. □

Vom Umgang mit dBase III PLUS (2)

Nachdem die ersten Einstiegsschwierigkeiten ins dBase III PLUS überwunden sind, setzen wir mit einer weiteren Prozedur unsere Serie über diese interessante Programmiersprache fort, wobei auch diesmal eine ganze Reihe neuer Befehle besprochen wird. Gleichzeitig sollen aber auch zwei wichtige Hilfsprogramme Erwähnung finden, die für das dBase III PLUS wie das Salz in der Suppe sind und einen gewissen Komfort in das System bringen.

Schon in der letzten Folge (M+K 87-2) wurde eines dieser Hilfsprogramme, der «Clipper«, erwähnt. Dieses Dienstprogramm ist in erster Linie ein komfortabler Compiler. Zusammen mit einigen Optionen ist der «Clipper» aber ein völlig eigenständiges Programmiersystem, das bei nahezu gleichem Befehlssatz dBase III PLUS überflüssig macht. Vom dBase III PLUS wissen wir, das es sich um einen Interpreter handelt, der wie ein BASIC-Interpreter jeden Befehl einzeln liest, ihn in einen Maschinencode

Heinz Kastien

umwandelt und erst dann ausführt. Dies bringt den unbestrittenen Vorteil mit sich, dass auch Teilprogramme, ja sogar einzelne Befehle im Direktmodus lauffähig sind. Eine fehlerhafte Prozedur wird jedoch nur bis zur fehlerhaften Zeile korrekt ausgeführt, um dann mit einer Fehlermeldung abzubrechen. Durch die ständige Umwandlung der Befehle in den Maschinencode sind derartige Programme in ihrer Ausführung relativ langsam.

Durch die Compilierung wird das Sourcefile in ein Objektfile umgewandelt und dann mit einem RUNTIME-Modul gelinkt. Das compilierte File kann nun direkt vom Betriebssystem aus gestartet und ausgeführt werden, das dBase III PLUS ist also nicht mehr erforderlich. Bei der Compilierung werden fehlerhafte Befehle erkannt und auf dem Bildschirm gemeldet, ohne die Compilierung abzubrechen. Die Umwandlung des Sourceprogramms muss jedoch nach dessen Korrektur wiederholt werden. Erst wenn ein fehlerfrei umgewandeltes Objektfile vorliegt, kann mit dem RUNTIME-Modul gelinkt werden. Die compilierten Programme weisen gegenüber den Sourceprogrammen enorme Vorteile auf: Sie haben wesentlich geringere Laufzeiten, sind fehlerfrei und sie sind direkt vom Betriebssystem aus lauffähig.

Allerding darf auch ein erheblicher Nachteil nicht vergessen werden, die Programme lassen sich nach der Umwandlung in das Objektfile nicht mehr mutieren, oder besser gesagt, bei einer Aenderung muss das Quellprogramm mutiert und dieses erneut compiliert werden.

Die Compilierung sollte daher erst dann durchgeführt werden, wenn feststeht, dass die Sourcefile absolut fehlerfrei laufen. Wird mit dem «Clipper» gearbeitet, so ist der dBase III PLUS-Interpreter nicht unbedingt erforderlich, da die Programme mit nahezu jedem Textverarbeitungsprogramm geschrieben und anschliessend direkt mit dem «Clipper» in ein lauffähige Programm umgewandelt werden können. Der Clipper arbeitet aber nicht nur mit dBase III PLUS sondern auch mit dessen Vorgänger dBase III.

In den hier vorgestellten Beispielen wurden die Sourceprogramme mit SuperDesk und WordStar geschrieben. Zur Eröffnung der Datenbanken, die im dBase mit dem Befehl «CREATE» erfolgt, exisitiert beim «Clipper» ein spezielles Hilfsprogramm. Da die von uns abgedruckten Programme bereits im Sourcefile getestet worden sind, kann nach jeder der hier beschriebenen Methoden gearbeitet werden. Auf die eigentliche Compilierung mit dem «Clipper» werden wir später noch sehr genau eingehen.

Vereinsadressmenü

In der letzten Folge wurde das Programm «Vereinsadressmenü» abgedruckt. Der Aufbau und die Befehle dieser Prozedur sollen dieses erste Teilprogramm abschliessen. Verschiedentlich wurde schon die Art und Weise erwähnt, wie derartige Programme geschrieben werden, z.B. in dem

- mit MODIFY COMMAND MENU.PRG der Texteditor des dBase aufgerufen und das Programm eingegeben wird. Jede Zeile muss mit RETURN abgeschlossen werden. Nach dem Speichern der Prozedur kann diese direkt mit DO MENU gestartet werden. Der Texteditor des dBase verarbeitet jedoch nur 62 Zeilen mit maximal 66 Zeichen pro Zeile, oder Dateien mit maximal 4'096 Byte. Diese Sourcefile sind im dBase direkt lauffähig.

- mit einem Textverarbeitungsprogramm, z.B. WordStar oder SuperDesk wird die Prozedur geschrieben und beim Abspeichern mit dem Zusatz .PRG versehen. Auch hier kann das Programm vom dBase aus direkt gestartet werden. Bei anderen Textverarbeitungen muss allenfalls der Zusatz .PRG durch den RENAME- oder COPY-Befehl angehängt werden.

- das mit einer Textverarbeitungssoftware oder dem dBase-Texteditor geschriebene Sourcefile wird mit dem Clipper compiliert und dann direkt vom Betriebssystem aus gestartet.

Bildschirmsteuerung

Unsere spezielle Aufmerksamkeit soll aber heute den Befehlen des dBase III PLUS gelten, sofern sie in dieser Prozedur vorkommen. Der sicherlich wichtigste Befehl zur Positionierung von Ein- oder Ausgaben auf dem Bildschirm ist

@ Zeile,Spalte

Er entspricht dem LOCATE des Microsoft-BASIC und positioniert den Cursor an die gewünschte Stelle auf dem Bildschirm, an dem eine Eingabe erfolgen soll. Diesem Befehl folgt meist SAY oder GET. Die Angaben für Zeilen und Spalten können sowohl absolute Zahlen als auch numerische Variablen sein. dBase zählt die Bildschirmzeilen von 0-24 und die Spalten von 0-79. Wird dieser Bereich überschritten, erfolgt die Fehlermeldung «GET/SAY-Position liegt ausserhalb des Bildschirms».

SAY «TEXT»

SAY hat im BASIC das Pendant PRINT. Mit dem Befehl SAY wird der in Anführungszeichen gesetzte Text oder der Inhalt einer Konstanten auf dem Bildschirm ausgegeben.

@ 3,27 SAY «M+K Vereinsadressverwaltung»

schreibt die Textzeile «M+K Vereinsadressverwaltung» in die dritte Zeile des Bildschirms, beginnend ab Spalte 27.

```
TEXT = «Betrag : »
ZAHL = 25
@2,30 SAY TEXT
@2,40 SAY ZAHL
```

schreibt Betrag : 25.00 in die zweite Zeile ab Spalte 30. Mit diesen beiden Befehlen ist die gesamte Bildschirmmaske der Menüverwaltung in den Zeilen 7-15 aufgebaut.

GET (KONSTANTE)

Während der SAY-Befehl einen Text auf dem Bildschirm hinterlässt, holt GET den Inhalt einer Konstanten von der Tastatur. Dem GET-Befehl muss immer eine Konstante folgen.

```
STORE «J» TO ANTWORT
@3,20 «Ende des Programms ? » GET ANTWORT
READ
```

In der dritten Zeile in Spalte 41 wartet der Rechner auf die Eingabe eines Charakters, dessen weitere Auswertung individuell gestaltet werden kann.

READ

aktiviert alle GET-Befehle seit dem letzten CLEAR ALL oder CLEAR GETS. Hinter einem oder mehreren GET-Befehle ist ein READ zwingend.

SET-Befehle

Das Adressverwaltungsmenü beginnt mit zwei SET-Befehlen. Diese gehören zu einer ganzen Gruppe von SET-Befehlen, die wir im Verlauf dieses Lehrganges kennen lernen werden. Ihnen allen ist die erste Silbe, also das SET gemeinsam. Ebenso führen die Befehle dieser Gruppe ähnlich Funktionen aus, nämlich die Anzeige und Aenderung von Parametern. Es würde zu weit führen, alle SET-Befehle zu beschreiben, vielmehr sollen hier nur diejenigen erwähnt werden, die in den gelisteten Programmen vorkommen.

SET DEFAULT TO LF:

Es wird ein Laufwerk definiert.

SET DEFAULT TO A:

Dieser Befehl bestimmt für alle folgenden Operationen das Laufwerk A.

SET ESCAPE ON/OFF:

dBase-Prozesse können mit der ESCAPE-Taste abgebrochen werden. Um aber Programmunterbrüche durch irrtümliche Betätigung der ESCAPE-Taste zu vermeiden, kann mit

SET ESCAPE OFF

die Funktion dieser Taste abgeschaltet werden, mit

SET ESCAPE ON

wird die Funktion wieder rückgängig gemacht.

SET HEADING ON/OFF

unterdrückt die Anzeige der Feldüberschriften z.B. die des LIST-Befehls.

SET TALK ON/OFF

dBase bietet normalerweise mit jedem Befehl Systemmeldungen an, die im Anschluss an die Befehlsdurchführung auf dem Bildschirm ausgegeben werden. Die Anzeige dieser Meldungen wird mit

SET TALK OFF

unterdrückt.

SET STATUS ON/OFF

dBase III PLUS gibt auf der untersten Zeile eine Statusmeldung aus. Die Statusanzeige umfasst den gerade ausführenden Befehl, den Drive, das Feld usw. Diesen Befehl kennt dBase II und dBase III nicht. Die Ausgabe dieser Statusmeldung wird mit diesem SET-Befehl unterdrückt.

SET ECHO ON/OFF

dBase zeigt beim Programmablauf die auszuführenden Programmschritte an. Mit

SET ECHO OFF

wird die Anzeige unterdrückt.

SET COLOR TO

Auf dem Bildschirm lassen sich mit dieser Funktion Farben und Effekte der Bildschirmausgaben bestimmen. Bei monochromen Bildschirmen ist eine inverse, blinkende und unterstrichene Darstellung sowie erhöhte Helligkeit möglich. Farbmonitore erlauben überdies die Farbgebung in 15 verschiedenen Nuancen. Im BASIC entspricht dies dem Befehl COLOR. Im gleichen Befehl kann sowohl der Bildschirmvordergrund als auch dessen Hintergrund und Rand definiert werden.

SET COLOR TO +W/

stellt die Schriftzeichen mit erhöhter Helligkeit dar. Weiter Möglichkeiten des SET COLOR TO sind:

* (blinken der Charakter)
U (unterstreichen)
/ (Inversion der Darstellung)

Bei Farbbildschirmen werden mit

SET COLOR TO GR/B,W/R,GR

die Zeichen mit gelben Buchstaben auf blauem Hintergrund, und beim Hervorheben mit weissen Buchstaben auf roten Hintergrund dargestellt, der Rahmen ist gelb.

Viele Programme beginnen mit einem CLEAR Befehl.

CLEAR

löscht den Bildschirm, dieser Befehl ist identisch mit dem CLS des BASIC.

CLEAR ALL

schliesst alle .DBF .NDX und .FMT Dateien und löscht alle Temporärfelder.

CLEAR MEMORY

löscht dagegen nur die Temporärfelder, ohne die oben genannten Dateien zu schliessen. BASIC-Programme bewirken das gleiche mit dem Befehl CLEAR.

CLOSE

schliesst offene Dateien, es muss jedoch der Dateityp spezifiziert werden, z.B.

CLOSE FORMAT
CLOSE INDEX
CLOSE PROCEDURE

Der Befehl ist mit dem CLOSE des BASIC direkt vergleichbar, der ebenfalls offene Dateien schliesst.

Schleifen

Um das Menü-Programm vollständig zu verstehen, müssen zwei verschiedene Schleifenbefehle erklärt werden. Es sind dies die beiden Befehle

DO WHILE ... ENDDO
DO CASE .... ENDCASE

DO WHILE ... ENDDO ist ein echter Schleifenbefehl. Er beginnt immer mit DO WHILE (Bedingung) und endet immer mit ENDDO. Alle Befehle, die sich zwischen DO WHILE und ENDDO befinden, werden ausgeführt, solange die Bedingung der Schleife erfüllt ist. Es kann ausserdem eine Bedingung definiert werden, die ein Verlassen der Schleife bewirkt.

```
CLEAR
BEDINGUNG ="E"
VORGANG = 1
DO WHILE BEDINGUNG = "E"
§ 1,5 SAY " Wollen Sie den Vorgang beenden ?" GET BEDINGUNG
READ
VORGANG = VORGANG + 1
§ 2,5 SAY VORGANG
IF BEDINGUNG = "N"
STORE "E" TO BEDINGUNG
ENDDO
ENDIF
ENDDO
```

Programm «Schleifen»

DO CASE ... ENDCASE ist im BASIC mit dem IF THEN oder dem ON ... GOTO vergleichbar. Im Menü-Programm bewirkt die DO CASE ... ENDCASE Abfrage die Verzweigung zu den verschiedenen Detailprogrammen. Wir wollen hier der besseren Demonstration halber dem dBase zwei BASIC-Programm gegenüberstellen, die zwar die gleichen Funktionen ausführen, aber nicht direkt vergleichbar sind.

Gleichzeitig ist hier der Befehl DO ... erstmals erwähnt. DO führt eine Prozedur aus. Sofern sich diese nicht bereits im Speicher befindet, wird sie von der Disk geladen und gestartet. Dieser Befehl ist vergleichbar mit dem RUN des BASIC. Im Gegensatz zum RUN werden aber mit DO keine Variablen gelöscht.

```
DO CASE
CASE MENUWAHL = «1»
DO EINGABE
CASE MENUWAHL = «2»
DO ABRUF
.
.
.
ENDCASE
```

```
1. BASIC-Programm
  10 IF MENUWAHL = «1»
     THEN RUN «EINGABE»
  20 IF MENUWAHL = «2»
     THEN RUN «ABRUF»
2. BASIC-Programm
  10 ON A GOTO 100,200
  100 RUN «EINGABE«
  200 RUN «ABRUF»
```

Schliesslich wird mit RETURN ein Programm verlassen und zum aufrufenden Programm zurückgekehrt. RETURN TO MASTER kehrt zum aufrufenden Programm zurück.

Alle Befehle des Menü-Programmes sind besprochen worden. Es dürfte nun auch für den Anfänger des dBase kein Problem mehr sein, das Programm nach eigenen Wünschen abzuändern. Bevor wir aber nun das Listing des Dateneingabe-Programms besprechen, zuerst noch einige Worte über Maskengeneratoren und über Konstanten.

Maskengeneratoren

Die Programmierung ansprechender Bildschirmdarstellungen ist relativ zeitaufwendig. Es werden daher verschiedene Arten von Maskengeneratoren im Handel angeboten, die diese Arbeit wesentlich erleichtern. Bei allen bekannten Typen ist die Art der Bedienung immer gleich. Der Cursor definiert Bildschirmpositionen, an denen Fixtexte eingegeben werden. Diese erklärenden Texte lassen sich durch einfache Befehle farbig gestalten, hervorheben oder unterstreichen.

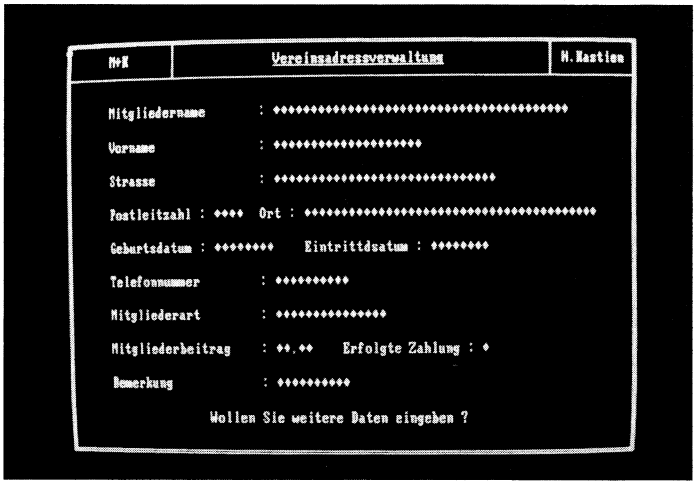


Bild 1: Entwurf einer Bildschirmmaske mit Maskengenerator

Der Maskengenerator eröffnet prinzipiell die gleichen Optionen, die auch mit dem SET COLOR TO-Befehl gegeben sind. Sobald die eigentliche Bildschirmmaske entworfen ist, wird die .dbf-Datei, die bearbeitet werden soll, vom Maskengenerator aufgerufen und die darin definierten Felder in die Maske übernommen.

Es entfällt also eine zweite Definition der Felder in der Maske. Schliesslich erstellt der Generator ein Maskenprogramm, das sich aus SAY, GET und SET COLOR TO zusammensetzt und nun individuell zum fertigen Programm ergänzt werden kann. Auch im dBase III PLUS ist ein derartiger Maskengenerator integriert der gleich arbeitet, die

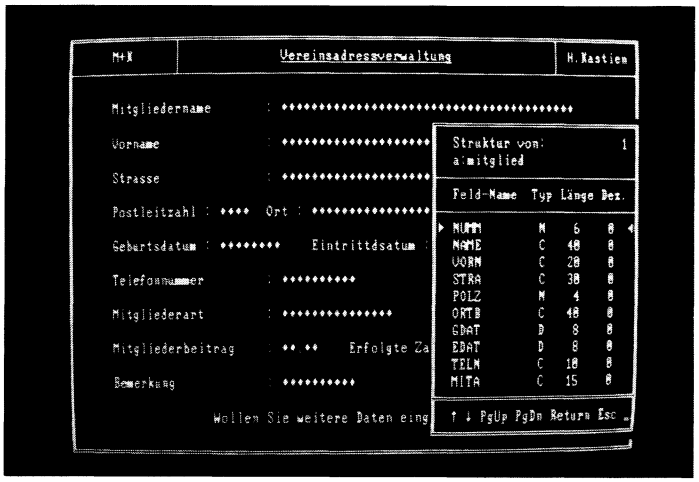


Bild 2: Uebernahme der Felder in die Maske

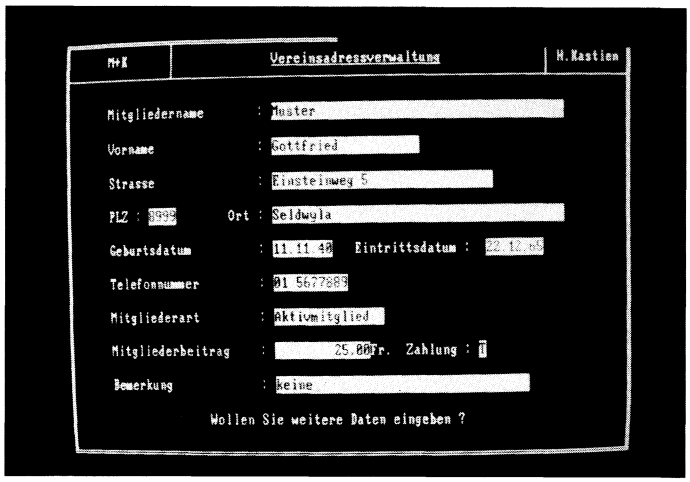


Bild 3: Fertig generiertes Eingabeprogramm

Bildschirmmaske jedoch in einer FORMAT-Datei abspeichert. Es darf jedoch nicht verschwiegen werden, das Masken, die mit dem dBase III PLUS-Generator entworfen worden sind, nicht unbedingt abwärts kompatibel sind, da dBase III die Befehle für Umrandungen und Linien nicht kennt. Die Bildschirmmasken können natürlich auch mit den SAY- und GET-Befehlen konventionell entworfen und in einer FORMAT-Datei abgespeichert werden, jedoch steht der Arbeitsaufwand in keinem Verhältnis zu einem Maskengenerator.

```
$  0, 0 say "┌──────────────────────────────────────────────────────────────────────────────┐"
$  1, 0 say "│       M+K                                                                    │"
SET COLOR TO B/
$  1,30 say "Vereinsadressverwaltung"
SET COLOR TO W/
$  1,67 say "│ H.Kastien │"
$  2, 0 say "├──────────┴──────────────────────────────────────────────────────────────────┤"
$  3, 0 say "│                                                                             │"
$  4, 0 say "│                                                                             │"
SET COLOR TO +W/
$  4, 7 say "Mitgliedername "
SET COLOR TO W/
$  4,28 say ":                                                                            │"
$  5, 0 say "│                                                                             │"
$  6, 0 say "│       Vorname                :                                              │"
$  7, 0 say "│                                                                             │"
$  8, 0 say "│       Strasse                :                                              │"
$  9, 0 say "│                                                                             │"
$ 10, 0 say "│       PLZ :              Ort :                                              │"
$ 11, 0 say "│                                                                             │"
$ 12, 0 say "│       Geburtsdatum           :          Eintrittsdatum :                    │"
$ 13, 0 say "│                                                                             │"
$ 14, 0 say "│       Telefonnummer          :                                              │"
$ 15, 0 say "│                                                                             │"
$ 16, 0 say "│       Mitgliederart          :                                              │"
$ 17, 0 say "│                                                                             │"
$ 18, 0 say "│       Mitgliederbeitrag      :            Fr.  Zahlung :                    │"
$ 19, 0 say "│                                                                             │"
$ 20, 0 say "│       Bemerkung              :                                              │"
$ 21, 0 say "│                                                                             │"
$ 22, 0 say "│                    Wollen Sie weitere Daten eingeben ?                      │"
$ 23, 0 say "│                                                                             │"
$ 24, 0 say "└─────────────────────────────────────────────────────────────────────────────┘"
SET COLOR TO
SET COLOR TO W/,/W
$  4,30 GET SNAME PICTURE "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
$  6,30 GET SVORN PICTURE "XXXXXXXXXXXXXXXXXXX"
$  8,30 GET SSTRA PICTURE "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
$ 10,13 GET SPOLZ PICTURE "####"
$ 10,30 GET SORTB PICTURE "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
$ 12,30 GET SGDAT
$ 12,59 GET SEDAT
$ 14,30 GET STELN PICTURE "XXXXXXXXXX"
$ 16,30 GET SMITA PICTURE "XXXXXXXXXXXXXXX"
$ 18,30 GET SBEIT
$ 18,58 GET SZAHL
$ 20,30 GET SBEME PICTURE "XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX"
$ 22,57 GET ANTWORT
READ
```

Listing der generierten Maske, die in der nächsten Folge weiter ausgebaut wird

Dateneingabeprogramm

Die zweite zu besprechende Struktur ist das Dateneingabe-Programm. Im einfachsten Fall können die Daten bereits im Anschluss an die Eröffnung der Datenbank erfasst werden, denn nach deren STRUCTURE-Sicherung in der «CREATE»-Routine fragt dBase, ob bereits Daten eingegeben werden sollen (auf diese Art der Datenerfassung wurde in der ersten Folge hingewiesen). Diese Befehlserweiterung kann aber nur eine Hilfslösung sein, denn es wird weder mit einer Bildschirmmaske noch mit irgendwelchem Komfort gearbeitet. Um die Daten in einer übersichtlichen Form zu erfassen, bestimmte Textvorgaben zu definieren und möglicherweise die einzelnen Felder auf logisch richtige Eigaben zu prüfen, ist eine gut aufgebaute Bildschirmmaske zwingend erforderlich. Der Aufbau der Bildschirmmaske erfolgt mit dem Befehl

MODIFY COMMAND MASKE.FMT

Es ist also der gleiche Befehl, der auch bereits bei der Abfassung des Menü-Programms verwendet worden ist. Da es sich aber diesmal nicht um ein Programm, sondern um eine Formatdatei handelt, muss die Extension .FMT an den Namen der Maske angefügt werden, da der Texteditor sonst die Maske als Prozedur interpretiert. Soll die Maske auf einem anderen Laufwerk als dem DEFAULT-Drive abgespeichert werden, so muss dieses vor dem Namen angegeben werden, also

MODIFY COMMAND A:MASKE.FMT

In einer Formatmaske dürfen nur die Befehle SAY, GET und READ verwendet werden. Da Befehle wie SET COLOR TO nicht unterstützt werden, sind die gestalterischen Möglichkeiten der FORMAT-Dateien sehr beschränkt. Wir wollen daher diese Möglichkeit nur am Beispiel der einfachen Vereinsadresseingabe beschreiben.

```
** Eingabemaske M+K Vereinsadressverwaltung by H. Kastien **
** 28.01.1987 **
§  1,  4  SAY "M+K                          Vereinsadressverwaltung"
§  4,  4  SAY "Mitgliedername          :"
§  4, 27  GET  MITGLIED->NAME
§  6,  4  SAY "Vorname                 :"
§  6, 27  GET  MITGLIED->VORN
§  8,  4  SAY "Strasse          :"
§  8, 27  GET  MITGLIED->STRA
§ 10,  4  SAY "PLZ :"
§ 10, 10  GET  MITGLIED->POLZ
§ 10, 21  SAY "Ort :"
§ 10, 27  GET  MITGLIED->ORTB
§ 12,  4  SAY "Geburtsdatum            :"
§ 12, 27  GET  MITGLIED->GDAT
§ 20,  8  SAY "Eintrittsdatum :"
§ 12, 55  GET  MITGLIED->EDAT
§ 14,  4  SAY "Telefonnummer           :"
§ 14, 27  GET  MITGLIED->TELN
§ 16,  4  SAY "Mitgliederart           :"
§ 16, 27  GET  MITGLIED->MITA
§ 18,  4  SAY "Mitgliederbeitrag       :"
§ 18, 27  GET  MITGLIED->BEIT
§ 18, 33  SAY "Fr.          Zahlung :"
§ 18, 54  GET  MITGLIED->ZAHL
§ 20,  4  SAY "Bemerkung               :"
§ 20, 27  GET  MITGLIED->REME
§ 23, 20  SAY "Wollen Sie weitere Eingaben ?"
§ 23, 51  GET ANTWORT
READ


** Eingabeprogramm M+K Vereinsadressverwaltung by H. Kastien **
** 28.01.1987 **
CLEAR
CLEAR ALL
SET TALK OFF
SET HEADING OFF
USE A:MITGLIED
SET FORMAT TO A:MITGLIED
STORE "J" TO ANTWORT
DO WHILE ANTWORT = "J"
APPEND
READ
IF UPPER(ANTWORT) <> "J"
EXIT
ENDIF
ENDDO
```

Maske und Prozedur «einfache Adressverwaltung»

Ein wichtiger Bestandteil aller Eingabeprogramme ist der Befehl

APPEND und APPEND BLANK

APPEND fügt am Ende der aktivierten Datei Datensätze an. APPEND stellt dabei die Standard-Datenmaske zur Verfügung. Wird eine indiviuelle Maske benutzt so muss der Befehl APPEND BLANK verwendet werden, der immer einen leeren Datensatz anhängt. Der Bildschirmmodus wird nicht eingeschaltet und der Datenzeiger auf den neuen, angehängten leeren Datensatz gestellt. Der APPEND Befehl kann mit ESC oder ^End verlassen werden, sofern keine besonderen Abfragen vorgesehen sind. Alle unter APPEND aktivierten Indexdateien werden aktualisiert. APPEND ist der grundlegende Befehl der bildschirmorientierten Dateneingabe.

In der aufgelisteten einfachen Adresseingabe sind alle bisherigen Aussagen und Befehle berücksichtigt worden.

Den Abschluss der Eingaberoutinen stellt ein komfortabels Eingabeprogramm dar. Durch den speziellen Aufbau kann dieser Programmtyp durch späteres Einfügen nur weniger Befehle auch als Netzwerkversion benutzt werden. Dieses Programm gliedert sich in folgende Teile.

1. Setzen der Parameter
2. Speichern der Konstanten
3. Bildschirmmaske
4. Dateneingabe mit Prüfung
5. Abschluss

1. Dieser erste Teil kann entfallen, wenn das Eingabeprogramm immer vom Menü aufgerufen wird, da im Menü diese Parameter bereits gesetzt wurden und Variable und Parameter durch den DO-Befehl nicht gelöscht oder verändert werden. Wird das Programm jedoch allein benutzt, so müssen diese Zeilen vorhanden sein, um die ordnungsgemässe Funktion dieses Programms zu gewährleisten.

2. Dem Problem der Konstantenspeicherung kann auf zwei verschiedene Arten begegnet werden. Einerseits können mit dem Befehl STORE die einzelnen Konstanten gespeichert werden, andererseits ist es auch möglich, die Konstante gleich dem Wert zu setzen. In der letzten Folge wurden die verschiedenen Feldtypen besprochen. Eine Konstante ist in seiner Art mit einem Feld prinzipell gleichzusetzen, eine Konstante aber nicht in einer Datei verankert sondern nur temporär wirksam. Sie kann also im Gegensatz zu einem Feld während des Programmablaufs gespeichert, geänder oder gelöscht werden.

STORE 15 TO AV oder AV = 15

speichert in der Konstanten AV den Wert 15.

STORE «Meier» TO AV15 oder «Meier» = AV15

speichert in der Konstanten AV15 den Text «Meier»

STORE DATE() TO DATUM oder DATUM = DATE()

speichert in der Konstanten DATUM das momentane Datum

STORE SPACE (20) TO LEER oder LEER = SPACE (20)

speichert 20 Leerzeichen in der Konstanten LEER.

MEMO-Felder können nicht in einer Konstante gespeichert werden, da aber unsere Adressverwaltungseingabeprogramm mit Konstanten arbeiten wird, empfiehlt es sich, das Feld BEME, das als MEMO-FELD deklariert worden ist, vor der eigentlichen Dateneingabe in ein alphanumerisches Feld mit 40 Zeichen zu ändern. Sie erreichen dies mit dem Befehl MODIFY STRUCTURE, es gehen hierbei keine Daten verloren, ausser den Bemerkungen in den MEMO-Feldern.

Werden die Konstanten im Programm selbst definiert, so muss bei einer allfälligen Aenderung der Werte zu einem späteren Zeitpunkt ein compiliertes Programm umgeschrieben werden. dBase III PLUS kennt aber die Möglichkeit, die Konstanten in einer MEMORY-Datei abzulegen und diese auf Disk zu speichern. In diesem Fall werden die Zeilen 1-10 im Direktmodus eingegeben. Nach der Definition aller Konstanten speichert man mit

SAVE TO MITGLIED

die Konstantendatei auf Disk. Die Memory-Datei trägt dann die Zusatzbezeichnung .MEM. Diese Daten werden mit dem Befehl

RESTORE FROM MITGLIED

wieder in den Konstantenspeicher eingelesen und mit

DISPLAY MEMORY

auf dem Bildschirm ausgegeben werden.

LIST MEMORY

hat die gleiche Funktion wie DISPLAY MEMORY. LIST MEMORY TO PRINT listet die Konstantendatei auf dem Printer. Eine Konstantendatei kann jederzeit mutiert werden, ohne das Programm verändern zu müssen. Gerade umfangreiche Programme werden hierdurch stark vereinfacht. Der Befehl RESTORE FORM ... löscht alle vor dem Aufruf der MEMORY-Datei im Speicher befindlichen Konstanten, ausser der RESTORE-Befehl trägt die Zusatzbezeichnung

ADDITIVE

Sollen Konstanten neu definiert werden, so lassen sie sich mit STORE einfach überschreiben. Zum Löschen der Konstanten aus dem Speicher wird der RELEASE-Befehl verwendet.

RELEASE AV

löscht die Konstante AV.

RELEASE ALL

löscht alle Konstanten.

RELEASE ALL EXCEPTE AV**

löscht alle Konstanten des Speichers ausser den Konstanten, die mit AV beginnen und zwei weitere Charakter haben. Das nachfolgende Beispiel zeigt eine kleine Memory-Datei, wie sie im nachfolgenden Dateneingabeprogramm verwendet wird.

```
GDAT        lokal (priv)  D  24.03.67
SNAME       lokal (priv)  C  "                    "

SORTB       lokal (priv)  C  "                    "

SVORN       lokal (priv)  C  "                    "
SMITA       lokal (priv)  C  "                    "
SSTRA       lokal (priv)  C  "                    "

SBEIT       lokal (priv)  N          25.00  (        25.00000000)
SPOLZ       lokal (priv)  N          1000  (      1000.00000000)
SEDAT       lokal (priv)  D  24.03.87
SGDAT       lokal (priv)  D  24.03.87
MINPOLZ     lokal (priv)  N          1000  (      1000.00000000)
MAXPOLZ     lokal (priv)  N          9999  (      9999.00000000)
MINEDAT     lokal (priv)  D  24.03.67
   13 Variable definiert,      232 BYTES benutzt
```

Memory-Datei

In der nächsten Folge wird ein komfortables Dateneingabeprogramm behandelt, das die Eingaben, sofern möglich, auf ihre Richtigkeit prüft. Das oben beschriebene einfache Programm kann aber bereits zu einer ersten Datenerfassung verwendet werden, da die eigentliche Datenbank für beide Programmtypen die gleiche ist. Neben diesem Listing und der Besprechung der neuen Befehle wird dann auch mit der Ausgabe der Daten begonnen. □

Künstliche Intelligenz (5. Teil)

Wenn Sie die bisher erschienenen Teile dieser Artikelserie gelesen haben, so werden Sie sich zu Recht fragen, ob man die dort beschriebenen Anwendungen auch zu Hause auf dem PC programmieren kann. Die Antwort ist prinzipiell ja, wenn auch mit einigen Einschränkungen.

Die erste dieser Einschränkungen betrifft die Schnelligkeit des Prozessors. Da die Programme in der Regel sehr viel Rechenzeit erfordern, werden sie zumindest auf dem IBM PC äusserst langsam ablaufen. Wird hingegen ein IBM AT mit 10 MHz eingesetzt, so sind die Ergebnisse schon ganz erträglich. Das zweite Hindernis bietet die Wahl der richtigen Programmiersprache. Natürlich kann man ein ELIZA-ähnliches Programm in BASIC oder Pascal schreiben. Allerdings wird man bei ehrgeizigeren Projekten hier schnell in Schwierig-

Michael Schlingmann

keiten geraten, da die Struktur dieser Sprachen den Zielen der künstlichen Intelligenz nicht entgegenkommt. Als Beispiel sei hier nur die Möglichkeit zum Aufruf von rekursiven Funktionen genannt, die in BASIC oder Pascal meist nicht implementiert sind.

Zum Glück gibt es aber noch andere Sprachen, die diesen Erfordernissen besser angepasst sind und die auch auf Personal Computern lauffähig sind. Als Beispiele wurden hier LISP und PROLOG ausgewählt. KI-Programme sind fast ausschliesslich in diesen Sprachen geschrieben. Sowohl LISP als auch PROLOG kann man für relativ wenig Geld anschaffen, so dass sich diese Ausgabe bei Interesse sicherlich lohnt. Im Folgenden sollen die Eigenschaften der beiden Sprachen kurz beschrieben werden.

LISP

LISP kann man quasi als Urvater aller KI-Sprachen ansehen. Es wurde zwischen 1956 und 1958 von John McCarthy aus der Sprache IPL entwickelt und auf einem Grossrechner IBM 704 implementiert. Der Name ist ein Akronym für **lis**t **p**rocessing, auf Deutsch etwa «Listenverarbeitung».

McCarthy wollte einem Computer Intelligenz beibringen und merkte bald, dass er nicht daran vorbeikam, mit symbolischen Ausdrücken zu arbeiten, die man manipulieren und miteinander verknüpfen kann. Das aus diesen Ansprüchen entstandene LISP hat unter anderm folgende Eigenschaften:

- es steht ein Interpreter zur Verfügung, das heisst eine neue Idee kann sofort ohne aufwendige Compilation ausprobiert werden.
- es existieren Möglichkeiten zum rekursiven Funktionsaufruf.
- polnische Notation
- jedes eingegebene Wort wird als Programm interpretiert.
- LISP-Programme bestehen aus ineinander verschachtelten Funktionen.
- extrem einfache Datenstrukturen
- jede mit einem RETURN abgeschlossene Programmzeile wird sofort ausgewertet.
- viele verschiedene Dialekte

Da es für LISP bisher keinen Standard gibt, existieren unzählige Dialekte, deren Kern aber dieselbe Struktur aufweist. Die Datenstrukturen, um die sich alles dreht, werden von den LISPlern (das sind die Leute, die mit LISP Programme schreiben) auch «Symbolic expressions» (SE) genannt. Eine SE ist entweder ein Atom oder eine Liste. Unter einem Atom versteht man wie in der Chemie etwas Unteilbares. Ein LISP-Atom kann man demnach nicht in einen noch einfacheren Ausdruck verwandeln. Atome sind entweder Zahlen oder Namen. Hierbei wird eine Zeichenkette, die mit einer Ziffer beginnt, als Zahl behandelt, alles andere ist ein Name. Das Gegenteil eines Atoms ist eine Liste, das heisst eine Ansammlung von Atomen. Eine Liste besteht aus einer runden Klammer am Anfang und einer Aufzählung von Atomen oder weiteren Listen, die wieder geklammert werden müssen. Dadurch erhält ein LISP-Programm sein charakteristisches Aussehen (siehe die Beispiele weiter unten).

Zur Anwendung von Atomen und Listen einige kurze Beispiele:

(plus 2 3) ergibt als Antwort des LISP-Interpreters die Zahl 5. Wie man hier sieht, bedient sich LISP der polnischen Notation; der Operator erscheint vor den Argumenten. Besitzern von HP-Taschenrechnern wird diese Notation bekannt vorkommen, da dort (wie auch in der Programmiersprache FORTH) die umgekehrte polnische Notation verwendet wird, das heisst, der Operator erscheint erst nach seinen Argumenten.

Die Vorteile der polnischen Notation liegen auf der Hand: Die Anzahl der Argumente in einer Funktion sind nicht von vornherein festgelegt und der Interpreter erhält dadurch einen sehr einfachen Aufbau, was wieder der Rechengeschwindigkeit zugute kommt. Die vier Grundrechenarten werden in LISP alphanumerisch beschrieben:

\+ = plus
\- = difference
» = quotient
– = times

Mit diesen Hilfsmitteln können wir schon eine wesentlich komplexere Anwendung ausführen:

(difference 20 (plus 3 (times 2 4)))

ergibt zum Beispiel den Wert 11.

An dieser Stelle werden Sie sich sicher nach der Motivation dieser so komplizierten Rechenmethode fragen. Die Antwort ist denkbar einfach: alles, was in einer Klammer steht, ist eine Funktion. Diese Funktion wird ausgewertet und damit eine übergeordnete Funktion berechnet. Bei der Auswertung wird von links nach rechts vorgegangen, wobei bei ineinander verschachtelten Klammern die tiefste Klammerebene zuerst bearbeitet wird.

Der grosse Vorteil dieser Vorgehensweise ist nun der, dass man komplexe Programme mit einem Wort (dem Funktionsnamen) aufrufen kann. In dieser Hinsicht ähneln die LISP-Funktionen den Pascal-Unterprogrammen. Die Definition des Wortes geschieht in jedem Dialekt anders. In Mu-LISP, einem verbeiteten Dialekt für PCs, schreibt man

(DEFUN Wortname (Argumentliste))

Wortname ist nun ein neuer Befehl in LISP, der ab sofort wie jede andere Anweisung ausgeführt werden kann. Somit kann man den LISP-Interpreter auf seine eigenen Bedürfnisse zuschneiden. Die Programmiersprache FORTH arbeitet übrigens nach demselben Prinzip.

Alles was keine Zahl ist, wird von LISP als Variable (oder gleichbedeutend als Name) behandelt. Einer Variablen kann natürlich ein Wert zugewiesen werden. Dies geschieht mit dem Befehl setq:

(setq hugo (plus 2 3)) weist der Variablen hugo den Wert 5 zu. Denselben Effekt erzielen übrigens (setq hugo (plus 1 1 1 1 1)) oder (setq hugo 5).

Das so definierte Atom hugo kann aber nicht nur Zahlen als Wert enthalten. Ihm lässt sich auch jedes andere beliebige LISP-Objekt zuweisen, z.B. eine Liste. So kann man hugo auch gleich der Liste (Hans Karl Georg) setzen oder gleich (plus 2 3). Da das letztere Beispiel eine ausführbare Funktion darstellt, muss dies unterbunden werden, was mit einem Häckchen (Apostroph) geschieht: (setq hugo'(plus 2 3)).

Auf diese Weise kann man hugo später manipulieren und erst dann auswerten, wenn es notwendig ist. Vielleicht ahnen Sie schon eine der Stärken (oder Schwächen) von LISP: gleich, was Sie eingeben, bekommen Sie vom Interpreter immer die Auswertung geliefert, vorausgesetzt, die Syntax ist richtig. Ein LISP-Programm kann also nicht abstürzen. Stattdessen muss man sich genau überlegen, was man mit einer Anweisung anrichten kann, da eine, in anderen Sprachen unkorrekte, Auswertung natürlich das ganze Programm unbrauchbar machen kann. Andererseits erhält man dadurch eine fast unglaubliche Flexibilität, und kann mit wenig Code umfangreiche Probleme bearbeiten.

In der mathematischen Theorie der Vektorräume kann man sich eine eigene Arithmetik aufbauen, die mit den üblichen Grundrechenarten nichts mehr zu tun haben muss. Damit die entstandene Theorie in sich schlüssig ist, muss der Vektorraum aber unter anderm eine Bedingung erfüllen: es muss ein neutrales Element existieren, dessen Anwendung auf ein anderes Element wieder dieses andere Element ergibt. In LISP ist hierfür das Wort NIL vorgesehen. NIL entspricht einer leeren Liste. Der Wert von NIL ist nichts (siehe die lateinische Uebersetzung). NIL braucht man, um einer Variablen überhaupt einen Wert zuweisen zu können, damit sie später manipulierbar ist. In BASIC entspricht NIL etwa der Zahl Null, mit dem Unterschied, dass NIL an keine Datenstrukturen gebunden ist.

Es gibt noch ein weiteres Atom, dessen Wert fest und gleich ihm selbst ist: t ist der Wahrheitsoperator und entspricht dem «true» in Pascal. Wer sich näher mit Listen, Atomen und ihren Auswirkungen beschäftigen will, dem sei die Artikelserie von Douglas R. Hofstadter empfohlen (siehe Literaturverzeichnis), der den Umgang damit in sehr amüsanter Weise lehrt.

LISP unterstützt die Erstellung von binären Bäumen, die unter anderem bei Datenbanken zum Einsatz kommen. Für diesen Zweck gibt es sogenannte «Paare«, die aus den Atomen «a» und «b» das geordnete Paar (a.b) bilden. Anstelle von a und b können natürlich wieder Paare stehen. Auf diese Weise fügen sich Paare zu binären Bäumen zusammen. Diese Bäume haben den Vorteil, dass bestimmte Daten recht schnell gefunden werden können.

LISP hat unter anderem den Nachteil, dass ausser GOTOs auch keine Schleifen vorgesehen sind. Allerdings kann man sich bei Bedarf diese Schleifen selbst definieren, wenn man sich die Mühe macht, einen Schleifenzähler zu setzen und diesen zu inkrementieren. Wie schon angedeutet, besteht in LISP die Möglichkeit des rekursiven Funktionsaufrufs. Mit dieser Methode werden üblicherweise Schleifen simuliert. Bei einer Rekursion ruft sich eine Funktion immer wieder selbst auf, bis das dabei entstandene Argument NIL ist oder eine Abbruchbedingung zutrifft. Ist die Bedingung erfüllt, so wird die Auswertung auf der nächst höheren Ebene fortgesetzt, was ja gerade einer FOR ...NEXT-Schleife in einer der anderen Programmiersprachen entspricht. Als Beispiel für eine Rekursion ein Programm, das die Summe der ungeraden Elemente eines Feldes berechnet:

```
(DEFUN unsum
 (LAMBDA (feld)
  (COND
   ((NIL feld) 0)
   ((ODD (CAR feld))
          (PLUS (CAR feld)
          (unsum CDR(feld))))
   (T (unsum (CDR feld))))))
```

Gibt man ein: unsum '(34 23 7 9) so erhält man den Wert 39. Die Funktion DEFUN kennen wir ja schon. LAMBDA hat die Aufgabe, den ihm folgenden Klammerausdruck als Parameterliste zu kennzeichnen. In diesem Fall besteht die Parameterliste nur aus «feld», das wiederum die Daten des interaktiv eingegebenen Zahlenfelds enthalten wird. CAR bezeichnet das erste Element einer Liste, CDR den Rest derselben. COND stellt eine Bedingung dar, wobei zuerst CAR ausgewertet wird. Wenn die Liste auch noch aus einem Rest besteht, ist die Bedingung wahr und CDR wird berechnet; ansonsten wird zur nächsten Liste übergegangen. Ist «feld» eine leere Liste, so ist NIL wahr und «unsum» erhält den Wert Null. Ist der CAR der Liste ungerade, so wird der CDR von «feld» berechnet und zu «unsum» addiert. Trifft keine der obigen Bedingungen zu, so wird einfach «unsum» mit «feld» als Parameter aufgerufen. Bis zum letzten Aufruf von «unsum» bleiben alle Berechnungen offen. Die Unterscheidung zwischen «gerade» und «ungerade» besorgt übrigens der Befehl ODD.

Wird eine Klammer zuwenig gesetzt, so fragt der Interpreter nach. Das Programm stürzt dabei aber nicht ab.

Mit diesem Beispiel beenden wir den kurzen Exkurs in die Programmiersprache LISP. Es soll nur noch darauf hingewiesen werden, dass LISP vor allem dann eingesetzt wird, wenn Wissen durch in verschiedenster zueinander in Beziehung stehender Symbole und Strukturen repräsentiert werden soll. Geht man das zu lösende Problem mehr von der logischen Seite her an, und versucht Wissen aufbauend auf logischen Theorien darzustellen, weicht man in der Regel auf PROLOG aus.

PROLOG

Wer zum ersten Mal ein PROLOG-Programm sieht, wird sich verwundert fragen, was er da denn vor sich hat. Es unterscheidet sich so sehr von anderen Sprachen, dass für effiziente Programme wohl eine längere Eingewöhnungszeit erforderlich ist. Nichtsdestotrotz hat sich PROLOG, vor allem bei den japanischen KI-Wissenschaftlern, durchgesetzt, da es für Expertensysteme geradezu prädestiniert ist.

PROLOG ist die Abkürzung für «**pro**gramming in **log**ic» und ist eine relativ junge Sprache. Sie wurde um 1974 zum ersten Mal beschrieben. Da verschiedene unabhängige Forschergruppen an der Entwicklung von PROLOG beteiligt waren, hat sich bis heute noch kein Standard herausgebildet. Viele Dialekte halten sich aber an die sogenannte «Edinburgh-Syntax», die als «DECsystem-10 PROLOG» implementiert wurde. Ist dies nicht der Fall, so werden zum Teil Konvertierungsprogramme bereitgestellt.

Die normalerweise benutzten Programmiersprachen wie BASIC, Pascal und FORTRAN werden als «prozedurale» Sprachen bezeichnet. Dabei wird ein Problem dadurch gelöst, indem verschiedene Anweisungen formuliert werden, die immer näher zum gewünschten Ergebnis hinführen. Etwas salopper gesprochen kann

man auch sagen, dass bei einer prozeduralen Sprache die Erstellung eines Flussdiagramms möglich ist.

Dagegen nennt man PROLOG eine «deklarative» Sprache. Das Programm besteht in der Regel nicht aus Anweisungen, sondern aus einer Sammlung von Fakten, die eine Wissensbasis aufbauen. Anstatt dass ein Algorithmus zur Problemlösung formuliert wird, versucht man in PROLOG, die logischen Sachverhalte und Zusammenhänge darzustellen. Um sich darunter etwas Konkretes vorstellen zu können, sei an den zweiten Teil dieser Beitragsreihe (M+K 86-5) erinnert: eine Wissenbasis kann man aufbauen unter Verwendung der Prädikatenlogik. Und gerade diese Logik verwendet PROLOG. Dazu nochmal ein Beispiel: Der Computer soll bei der Entwirrung eines komplizierten Verwandtschaftsverhältnisses helfen. Folgende Tatsachen seien bekannt:

Agnes ist die Mutter von Stefan
Helga ist die Mutter von Melanie
Peter ist der Vater von Helga
Helene ist die Mutter von Agnes

Die Frage besteht darin, wer der Enkel von Helene ist. In BASIC oder einer anderen prozeduralen Sprache wäre nun ein grösseres Programm erforderlich, um die gewünschte Antwort zu erhalten. In PROLOG dagegen muss man nur die Fakten (also die Verwandtschaftsverhältnisse) eingeben und dem Interpreter irgendwie erklären, was man unter einem Enkel versteht. Die Wissensbasis sieht dabei folgendermassen aus:

```
mutter (agnes, stefan)
mutter (helga, melanie)
vater  (peter, helga)
mutter (helene, agnes)
```

Literaturverzeichnis

Douglas R. Hofstadter
in: Spektrum der Wissenschaft, Heft 4/83 und folgende
In der Kolumne «Metamagikum» wird eine vergnügliche Einführung in LISP gegeben.

Patrick H. Winston, Bertold Klaus Horn
LISP. Addison Wesley,
ISBN 0-201-08329-9

W.F. Clocksin, C.S. Mellish
Programming in PROLOG
Springer Verlag

In der Sprache der Prädikatenlogik hat die Eingabe Eigenschaft

(Ausdruck1, Ausdruck2)

die Bedeutung «Ausdruck1 hat eine Eigenschaft, über die sie mit Ausdruck2 in Beziehung steht». Die Umkehrung gilt dabei aber nicht. Nun müssen wir PROLOG noch mitteilen, was Väter, Söhne, Mütter und vor allem Enkel sind:

```
enkel (Enkel, Grossmutter) :- sohn (Enkel, Sohn)
sohn (Sohn, Vater) :- vater(Vater, Sohn)
```

Dabei sollen Worte mit kleinen Anfangsbuchstaben dieselbe Bedeutung haben wie Konstante in einer der üblichen Programmiersprachen. Worte mit grossen Anfangsbuchstaben hingegen sollen Variable darstellen. Die Reihenfolge der Feststellungen ist im Prinzip gleichgültig, sieht man einmal davon ab, dass der Rechner bei der Problemlösung immer zuerst am Anfang der Wissenbasis sucht. Wenn man also schon eine Ahnung hat, in welcher Weise sich eine Aufgabe lösen lässt, so sollte man dies beim Aufbau der Wissensbasis beachten. Die erste Anweisung bedeutet in Klarschrift etwa «Jemand ist der Enkel einer Grossmutter, wenn diese Grossmutter die Grossmutter des gesuchten Enkels ist».

Gibt man nun ein:
?- enkel (werwohl?, helene)
(werwohl? ist eine Variable, in der die Antwort steht), so erfolgt als Reaktion des Computers:
werwohl? = stefan.

Natürlich kann man mit PROLOG nicht nur solche subtilen Probleme lösen, sondern auch ernsthafte Anwendung betreiben. Zu diesem Zweck wurde die Syntax der Prädikatenlogik dahingehend ausgebaut, dass als Argumente beliebig viele (oder auch keine) Ausdrücke stehen können. Die obige Bezeichnung «Eigenschaft» nennt man in der Prädikatenlogik einfach «Prädikat». Ein Prädikat ist wahr, wenn einer der in ihm zugeordneten Ausdrücke wahr ist. Mit dieser Regel kann man auch ganz leicht eine ODER-Verknüpfung festlegen, indem man definiert:

```
wetter (i) :- schlecht (i)
wetter (i) :- gut (i)
```

Das Wetter, das von der Bedingung i abhängen kann, ist entweder schlecht oder es ist gut. Eine andere Alternative würde nur dann bestehen, wenn man weitere Daten über das Wetter in die Wissenbasis eingibt.

Der Ablauf eines PROLOG-Programms ist, wie nicht anders zu erwarten, mit einem herkömmlichen nicht vergleichbar. PROLOG kämmt die Wissenbasis anhand der ihm zur Verfügung stehenden Regeln durch und versucht, neue Erkenntnisse daraus zu gewinnen. Ist eine solche Erkenntnis gewonnen und wird ein neues Prädikat ausgewertet, das diese Erkenntnis nicht erfüllt, so wird sie wieder verworfen. Dabei geht der Interpreter bei der Auswertung der Wissensbasis wieder an den Punkt zurück, an dem die (jetzt falsche) Erkenntnis gewonnen wurde. Aufgrund der neuen Daten wird eine bessere Lösung gesucht.

Da an jeden PROLOG-Interpreter eine Datenbank angekoppelt ist, kann man schon eingegebene Fakten leicht korrigieren oder auch löschen. PROLOG stellt sich dann automatisch auf die neuen Verhältnisse ein. Auch ist es in der Regel möglich, Unterprogramme zu definieren, wobei man allerdings berücksichtigen muss, dass GOTOs und Schleifen prinzipiell nicht vorgesehen sind, um dorthin zu verzweigen. Der Begriff «Unterprogramm» ist hier also nur mit Vorsicht zu geniessen.

Noch ein Wort zur Anwendung von PROLOG auf PCs: Aufgrund der

COMPUTER SPLITTER

Hacker, wollt ihr ewig probieren?

(505/fp) Ueber 10'000 Personen sollen vergeblich versucht haben, einen Text, welcher mit dem neuen Chiffrierprogramm mPROTECT verschlüsselt wurde, zu knacken. Nun, das Unterfangen scheint ebenso einfach wie hoffnungslos: Beim Codieren des Textes wird ein Zufallszahlengenerator an eine vom Benutzer eingegebene Zeichenfolge von bis zu 24 Zeichen gekoppelt. Dazu gibt es 2,5466E56 Möglichkeiten, eine für menschliches Denken keinesfalls fassbare Zahl. Erfreulich: Die den Wettbewerb veranstaltende Organa AG in München und Luzern verschenkt den ausgeschriebenen Preis als Sachspende an eine Aktion für krebskranke Kinder. □

Ueberprüfung von Prädikaten wird der Speicher des Rechners recht extensiv genutzt. So kann es vorkommen, dass das Programm abstürzt, da der Speicher für die Aufstellung einer neuen Hypothese nicht mehr ausreicht. Bei einer vernünftigen Anwendung wächst zudem die Grösse der Wissensbasis sehr schnell, so dass die Geduld des Benutzers beim Warten auf eine Antwort des Computers auf eine schwere Probe gestellt wird.

Es empfiehlt sich also auf jeden Fall die Anschaffung eines Compilers, der zwar finanzielle Belastung mit sich bringen wird, in der Regel aber erhebliche Geschwindigkeitsvorteile hat. Als Alternative wird von der Firma Borland das Programmpaket TURBO PROLOG angeboten. Dieses ist recht schnell und läuft auch auf speichermässig weniger üppig ausgestatteten Computern. Allerdings muss man dazu sagen, dass nicht alle Eigenschaften eines «richtigen» PROLOG zur Verfügung stehen. Trotzdem ist dieses Programm sehr interessant, da hiermit völlig neue Wege bei der Problemlösung beschritten werden können.

Wer LISP und PROLOG einmal probieren möchte, ohne gleich Unsummen investieren zu müssen, der sei auf die Anbieter von Public Domain-Programmen verwiesen. Die dort zur Verfügung stehenden Implementationen entsprechen zwar nicht dem neuesten Stand der Technik, man kann aber trotzdem interessante Versuche damit anstellen. Ausserdem befinden sich auf den Disketten normalerweise auch viele Beispielprogramme, die den Umgang mit der neuen Sprache erleichtern. □

COMPUTER SPLITTER

Kopierschutz ad absurdum

(734/ro) Die Münchner Firma Heimsoeth verkauft Software ohne jeden Kopierschutz. Zum Aerger der Konkurrenz sind die Heimsoeth-Programme zudem oft preiswerter als vergleichbare Software. Dabei sind die zum Teil in Lizenz verkauften Programme keineswegs minderer Qualität - ganz im Gegenteil: Zum Angebot gehören z.B. Turbo Pascal oder Lotus 1-2-3. Fritz Heimsoeth sieht hierin keinen Widerspruch und der Erfolg seines Unternehmens scheint seine Strategie zu bestätigen. Preiswerte und gute Software braucht keinen Kopierschutz. Ohnehin ist die meiste Software zumindest zum Teil ohne ein dazugehöriges Handbuch nur sehr eingeschränkt nutzbar. Ein Kopierschutz kostet ja irgendwo auch Geld und andere Firmen nehmen sich dieses ja letztlich vom Endverbraucher. Egal wieviele Raubkopien auch in Umlauf sein mögen: Bisher konnten z.B. 70'000 legale Kopien des bekannten Turbo Pascal alleine in Deutschland verkauft werden. So gelang der Firma Heimsoeth 1986 die Verdoppelung ihres Umsatzes auf 24 Mio. DM. Andere Softwarehäuser blickten anfangs auf Heimsoeth herab und setzten weiterhin auf Kopierschutz - 100 %igen gibt es allerdings immer noch nicht. Mittlerweile stellt man aber auch hier Ueberlegungen an, ob dies der richtige Weg ist und es ist anzunehmen, dass sich die Branche auf diesen zur Zeit noch unkonventionellen Weg einrichten muss. □

Serielle Superschnittstelle für Turbo-Pascal

In letzter Zeit werden wir auf der Redaktion immer wieder nach einer komfortablen Möglichkeit gefragt, Daten über die RS232-Schnittstelle an ein Turbo-Pascal-Programm zu übergeben. Wir haben uns deshalb entschlossen, ein Programm zu schreiben, das diesen Wunsch und gleich noch ein paar weitere Wünsche auf einen Schlag erfüllt.

Von diesem Artikel können auch Leser profitieren, die sich nicht für serielle Schnittstellen interessieren. Sie erfahren nämlich, wie sich Interrupt-Routinen weitgehend in einer Hochsprache (in diesem Falle Turbo-Pascal) schreiben lassen. Das am konkreten Beispiel Gezeigte kann dabei auch auf andere Anwendungsfälle

Eric Hubacher

übertragen werden. Im weiteren sehen Sie, wie die logischen Treiber von Turbo-Pascal durch eigene, einem speziellen Problem angepasste Treiber ersetzt werden können.

Der Wunschkatalog

Bevor man ein Programm schreibt, muss man wissen, was man überhaupt will. Also setzten wir uns hin und erstellten als erstes einen Forderungskatalog.

1. Das Programmpaket soll alle Möglichkeiten der RS-Schnittstelle ausnützen. Dies umfasst:

   a) Aktivieren der Schnittstelle
   b) Einstellen aller möglichen Schnittstellen-Parameter
   c) Lesen der Schnittstelle
   d) Datenausgabe über die Schnittstelle
   e) Desaktivieren der Schnittstelle

2. Das Programm soll im Hintergrund ablaufen, so dass auch Daten empfangen werden können, während das Hauptprogramm andere Arbeiten ausführt.

3. Um auch einen grösseren Datenanfall bewältigen zu können, muss das Programm über einen genügend gross dimensionierten Zwischenspeicher verfügen.

4. Die Prozedursammlung muss sich mit $I einfach in ein Turbo-Pascal Programm einbinden lassen.

5. Damit auch Leser, die der Assemblersprache nicht mächtig sind, die Routinen ihren speziellen Bedürfnissen anpassen können, soll ein möglichst grosser Teil der Prozeduren in Pascal geschrieben sein.

6. Alle Routinen müssen ausführlich dokumentiert sein, nach dem Motto: Ein leserlich geschriebenes Programm ist seine beste Dokumentation.

7. Klare Programmiervorschriften sollen die Lesbarkeit des Programmes weiter verbessern. (Siehe dazu Kästchen «Die wichtigsten Programmiervorschriften».)

8. Auch Programmieranfänger sollten das Programm verstehen können. Dazu hilft der beigefügte Text. Profis brauchen nur den Programmausdruck zu studieren.

Wie funktioniert das Paket?

Eine der Hauptforderungen an unsere Prozedursammlung ist die, dass der Computer zu jedem Zeitpunkt in der Lage sein soll, Daten über die RS232-Schnittstelle zu empfangen. Dies lässt sich mit den üblichen linearen Programmiertechniken, wie zum Beispiel dem regelmässigen Abfragen (polling) der entsprechenden Schnittstelle, nicht gewährleisten. Hier bietet uns die Interrupt-Fähigkeit unseres Computers eine komfortable Lösung.

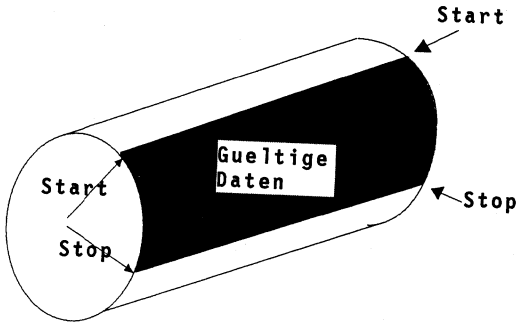


Abb. 1: Das Prinzip des Trommelspeichers

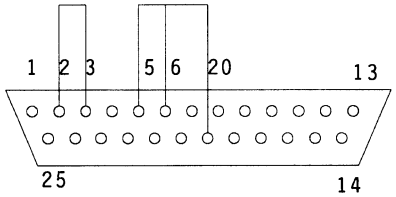


Abb. 3: Die Verdrahtung des Teststeckers

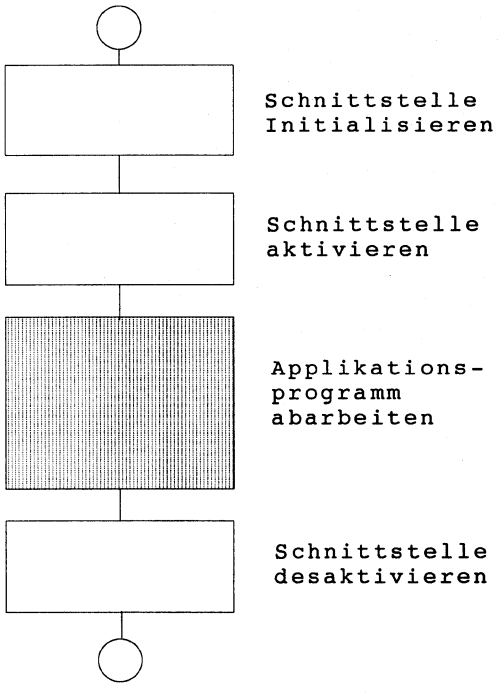


Abb. 2: Das Arbeiten mit der RS-Schnittstelle

Ein kleiner Ausflug in die Theorie

Eine Interrupt-Anforderung bewirkt, dass der Prozessor sein gerade in Bearbeitung befindliches Programm unterbricht, um eine bestimmte Aktion durchzuführen. Nach der Abarbeitung der meist kurzen Interrupt-Prozedur nimmt der Prozessor die Bearbeitung des Hauptprogrammes an der Stelle, an der er es verlassen hatte, wieder auf. Im folgenden Text werden wir den englischen Ausdruck «Interrupt» verwenden, einmal weil es sonst bei gewissen Ausdrükken unverständlich lange Wortwürmer gibt (wie z.B. Unterbrechungsanforderungsleitung für Interrupt-Leitung), und zum andern weil dieser Begriff sich auch bei den Fachleuten deutscher Zunge durchgesetzt hat.

Ein einfaches Beispiel aus dem täglichen Leben soll Ihnen das Wesen des Interrupts noch verdeutlichen. Angenommen Sie sitzen in Ihrem Büro bei einer wichtigen Arbeit, und plötzlich läutet das Telefon. Sie haben nun drei Möglichkeiten:

1. Sie lassen es einfach läuten

2. Sie bearbeiten kleine Teilaufgaben Ihrer Arbeit vollständig. Immer, wenn eine Teilaufgabe abgeschlossen ist, kontrollieren Sie, ob das Telefon läutet. Falls ja, übernehmen Sie den Anruf. Nach dem Bearbeiten des Telefonanrufes nehmen Sie die nächste Teilaufgabe in Angriff.

3. Sie unterbrechen Ihre Arbeit sofort und übernehmen den Anruf. Nach dem Bearbeiten des Telefonanrufes fahren Sie mit Ihrer Arbeit an der Stelle, an der Sie unterbrochen wurden, weiter.

Die selben drei Fälle lassen sich auch beim Betrieb eines Computers unterscheiden. Der erste Fall ist der einfachste: Nichts machen kann beinahe jeder.

Im zweiten Falle wird immer dann, wenn eine Teilaufgabe erledigt ist, geprüft, ob an der Schnittstelle Daten anstehen; ist dies der Fall, so werden die Daten abgeholt und nach Vorschrift verarbeitet. Die Problematik dieser Lösung liegt darin, dass immer in sehr kurzen Teilabschnitten ein Abfragen (Pollen) der Schnittstelle erfolgen muss. Bei einer eher langsamen Uebertragungsrate von 1'200 Baud muss dies mindestens alle acht Millisekunden (8 Tausendstel Sekunden) vorgenommen werden. Bei vielen Programmen ist dies mindestens

```
(*************************************************************************
                UNTERSTUETZUNG FUER RS232-SCHNITTSTELLEN
*************************************************************************

 Routinen zur Unterstüzung der seriellen Schnittstellen für Computer
 mit MS-DOS Betriebssystemen. Die Schnittstellenbehandlung läuft über
 Interrupts gesteuert, im Hintergrund ab. Die Zeichen werden in einem
 Datenbuffer von 500 Byte gespeichert. Die Grösse dieses Zeichenbuffers
 (BufferSize) kann angepasst werden. Das ganze Programmpaket kann mit
 $Irs.pas in ein Turbo-Pascal Programm eingebunden werden.

-------------------------------------------------------------------------

File-Name:  RSserver.PAS              Version 0.1

Betriebssystem: MS-DOS 2.0 und neuer.  PC-DOS 2.0 und neuer.

Programmiersprache: Turbo-Pascal Vers.3.00

Geschrieben von: E. Hubacher           Geschrieben am:      22.2.87
                                       Letzte Aenderung am: 25.3.87
-------------------------------------------------------------------------
Angebotene Routinen:

FUNCTION RsCharReady : BOOLEAN;
{ist wahr, wenn Daten zur Verarbeitung anstehen. Analog KeyPressed }

FUNCTION TxReady : BOOLEAN;
{ist wahr, wenn Daten ausgesendet werden können}

FUNCTION RsRead  : CHAR;
{liest ein Zeichen aus dem Zwischenspeicher.Stehen keine Zeichen an, wird
 der Empfang eines Zeichens abgewartet. Diese Routine lässt sich am besten
 über die logische Schnittstelle AUX ansteuern, z.B. Read(AUX, ch). }

PROCEDURE RsWrite (ch);
{Sendet Daten an die serielle Schnittstelle. Diese Routine lässt sich am
besten über die logische Schnittstelle AUX ansteuern, z.B. Write(AUX, ch). }

PROCEDURE RsInit (baudrate, parity, databits, stopbits);
{Bereitet die RS-Schnittstelle für die gewünschten Uebertragungsparameter vor}

PROCEDURE RsClear(buffer);
{Löscht den Zwischenspeicher und allenfalls bereits im Schnittstellenbaustein
 anstehende Daten}

 PROCEDURE RsStart (buffer, dsave );
 {Aktiviert den Interrupt-Mechanismus zum Empfang der Zeichen, und ersetzt
  die Turbo-Pascal-Treiber durch die selbstgestrickten}

 PROCEDURE RsStop (save);
 {Erstellt den ursprüngliche Zustand des Interrupt-Mechanismus wie er vor dem
  Aufruf des Programmes bestand, desaktiviert die RS-Schnittstelle und
  aktiviert die originalen Turbo-Pascal-Treiber}

*************************************************************************)

CONST  {Adressen für die Schnittstelle COM1:}
       ReceiverReg   = $3F8;
       TransmitReg   = $3F8;
       BaudRateGen   = $3F8;
       IntEnablReg   = $3F9;
       IntIdentReg   = $3FA;
       LineContrReg  = $3FB;
       ModemContrReg = $3FC;
       LineStatusReg = $3FD;
```

```
      IntMaskReg    = $21;      {Interrupt-Masken Register des 8259}
      BufferSize    = 500;

TYPE  BufferRec     = RECORD
                        content    : ARRAY [0..BufferSize] OF BYTE;
                        start, stop: INTEGER;
                      END;
      SaveRec       = RECORD
                        auxoutptr, auxinptr,
                        ivt_first_word, ivt_second_word  : INTEGER;
                        first_word, second_word          : INTEGER;
                        imr                              : BYTE;
                      END;

VAR dsave                 : INTEGER ABSOLUTE Cseg:$0006;
    buffer                : BufferRec;
    save                  : SaveRec;

{************************************************************************}
PROCEDURE RsReceiver;
{************************************************************************}
{Dies ist die Interrupt-Servic-Routine. Sobald ein Zeichen empfangen wird,
 wird diese Routine aufgerufen. Das Zeichen wird in einem Zwischen-
 speicher von der Grösse "BufferSize" abgespeichert. Eine Interpretation
 oder Veränderung der Zeichen wird nicht vorgenommen. Mehr Information
 über die Interrupt-Behandlung finden Sie im BYTE-Sonderheft  Herbst 1985,
 Seite 225}

BEGIN {RsReceiver}
  Inline($FB/ $1E/ $50/ $53/ $51/ $52/ $57/ $56/ $06);
  Inline($8C/ $C8/ $8E/ $D8/ $A1/ dsave/ $8E/ $D8);
  {Zeichen lesen und speichern}
  buffer.content[buffer.stop] := Port[ReceiverReg];
  IF ((buffer.stop + 1) MOD BufferSize) <> buffer.start {verhindert dass der}
  THEN buffer.stop := (buffer.stop +1) MOD BufferSize; {buffer überlaufen kann}
  Port[$0020] := $20;
  Inline($07/ $5E/ $5F/ $5A/ $59/ $5B/ $58/ $1F/ $CF);
END;  {RsReceiver}


{************************************************************************}
PROCEDURE RsWrite (ch : CHAR);
{************************************************************************}
{Uebergibt ein Zeichen an die serielle Schnittstelle. Es wird keine
 Interpretation von Zeichen vorgenommen. }

BEGIN {RsWrite}
  Port[TransmitReg] := ORD(ch);
END;  {RsWrite}

{************************************************************************}
FUNCTION RsCharReady(buffer: BufferRec) : BOOLEAN;
{************************************************************************}
{ist TRUE, wenn sich Zeichen im Zwischenspeicher befinden}

BEGIN {RsCharReady}
  IF buffer.stop = buffer.start
  THEN RsCharReady := FALSE                    {no character received}
  ELSE RsCharReady := TRUE;
END;  {RsCharReady}

{************************************************************************}
FUNCTION RsRead:  CHAR;
{************************************************************************}
{Liest ein Zeichen aus dem Zwischenspeicher. Befindet sich kein Zeichen
 im Zwischenspeicher so wird ein Carriage-Return (CR) zurückgesendet.}
```

organisatorisch ein Problem oder sogar eine unlösbare Forderung, wenn zum Beispiel gleichzeitig eine Dateneingabe über die Tastatur erfolgt.

Die beste Lösung stellt der dritte Fall, die Interrupt-Verarbeitung, dar. Sobald der Schnittstellenbaustein einen vollständigen Datensatz empfangen hat, meldet er dies über eine spezielle Leitung, die Interrupt-Leitung, dem Prozessor. Dieser schliesst den gerade in Bearbeitung befindlichen Maschinenbefehl ab - im schlechtesten Fall benötigt er dazu etwa 40 Mikrosekunden, normalerweise aber nicht mehr als 2 Mikrosekunden - und verzweigt in die Interrupt-Behandlungsroutine. Dort werden zuerst alle Daten, die für die spätere Wiederaufnahme der Arbeit im Hauptprogramm - an der richtigen Stelle natürlich - erforderlich sind, gesichert. Dann erfolgt die Ausführung der eigentlichen Interruptroutine. Zum Abschluss wird der Prozessorzustand, wie er vor der Unterbrechung bestand, wiederhergestellt und die Kontrolle an das eigentliche Hauptprogramm zurückgegeben.

Eine Vielzahl der Peripheriegeräte bei einem MS-DOS-Computer verkehren mit dem Hauptprozessor nur über Interrupts. Dazu gehören zum Beispiel die Festplattenstation, die Diskettenstationen, die eingebaute Uhr usw. All diese Komponenten benötigen während ihrer Arbeit die Rechenleistung des Hauptprozessors nur während einer kurzen Zeit.

Das Peripherigerät übermittelt dem Hauptprozessor seine Interrupt-Anforderung über eine von zwei speziellen Leitungen; beim 8088 sind dies die Anschlüsse 17 und 18. Der Anschluss 17 ist der Eingang für den nichtmaskierbaren Interrupt, der Anschluss 18 der Eingang für den maskierbaren Interrupt. Beide Interrupts werden vom System auf die gleiche Weise bearbeitet, die Annahme des Interrupts über den Anschluss 18 kann jedoch von einer Vorbedingung abhängig gemacht (maskiert) werden, während ein über den Anschluss 17 ausgelöster Interrupt immer sofort bearbeitet werden muss. Den Reset-Eingang könnte man übrigens ebenfalls als einen Interrupt-Eingang betrachten, mit der einzigen Spezialaufgabe, ein Maschinenprogramm aufzurufen, welches den Prozessor in den Grundzustand versetzt.

Ueber die Theorie und das Wesen der Interrupts gibt Ihnen jedes gute Mikroprozessor-Fachbuch gründlich Auskunft, weniger Hilfe finden Sie bei der praktischen Anwendung. Einen sehr guten Artikel, der Ihnen beim

Aufbau eigener Interrupt-Behandlungsroutinen Gold wert sein wird, finden Sie im Sonderheft Herbst 1985 der amerikanischen Zeitschrift Byte auf Seite 225 ff.

In diesem Beitrag finden Sie zwei Programmausdrucke: erstens die eigentliche Prozedursammlung, die sich mit «$Irsserv» einfach in jedes Turbo-Pascal-Programm einbinden lässt, vorzugsweise am Anfang des Programmes, sowie ein möglichst einfach gehaltenes Test- und Demonstrationsprogramm, welches Ihnen die Anwendung der Routinen vorführen soll. Das Testprogramm sendet ein über die Tastatur eingegebenes Zeichen sofort an die serielle Schnittstelle. Dieses ausgeschickte Zeichen wird über eine Drahtbrücke sogleich wieder in den Computer eingespiesen, über die Interrupt-Routine empfangen und auf dem Bildschirm zur Anzeige gebracht. Mit diesem Testprogramm können Sie nun nach Belieben spielen. Sie könnten beispielsweise die Interruptroutine so verändern, dass ein empfangener Buchstabe automatisch in einen Grossbuchstaben umgewandelt wird oder dass Sonderzeichen ignoriert werden. Der Möglichkeiten sind viele. Wie die erforderliche elektrische Ueberbrükkung hergestellt wird, zeigen wir Ihnen in der entsprechenden Abbildung.

Wichtiges zu den einzelnen Routinen

Der Zwischenspeicher

Bevor wir auf die eigentlichen Routinen zu sprechen kommen möchten wir Ihnen noch zeigen wie der Zwischenspeicher organisiert ist. Hier sind sehr viele Organisationsmöglichkeiten denkbar. Wichtig ist, dass der Speicher eine definierte, vorgegebene Grösse aufweist. So kann er auch bei einem schlecht geschriebenen Anwenderprogramm, welches ihn zu selten abfragt, nicht unbegrenzt wachsen. Deshalb und auch weil wir der Meinung sind, dass Pascal-Anfänger, die von BASIC herkommen, mit Zeigervariablen eher auf Kriegsfuss stehen, wurde auf eine flexible Speicherorganisation über Zeigervariablen verzichtet.

Unseren Speicher ist analog zu einem Fahrtenschreiber im Bus oder im Taxi organisiert. Die ältesten Daten werden immer durch die neuesten überschrieben, der Speicher hat kein Ende, sondern ist wie eine Trommel oder ein Kreis geschlossen (siehe

```
VAR  done   : BOOLEAN;      {TRUE sobald ein Zeichen gelesen werden konnte}

BEGIN {RsRead}
  done := FALSE;
  REPEAT
    IF RsCharReady(buffer) THEN
    BEGIN
      RsRead := CHR(buffer.content[buffer.start]);
      buffer.start := (buffer.start +1) MOD BufferSize;
      done := TRUE;
    END
    ELSE RsRead := CHR(13);  {Carriage return}
  UNTIL done
END;  {RsRead}

{************************************************************************}
PROCEDURE RsInit (baudrate: INTEGER; parity :CHAR; databits, stopbits: BYTE);
{************************************************************************}
{Erlaubt die Initialisierung der seriellen Schnittstelle
 Zulässig sind folgende Parameter:

 Baudrate: 50,75,110,135,150,300,600,1200,1800,2000,2400,3600,4800,7200,9600
 Parity  : N(o), O(dd), E(ven)
 Databits: 5..8
 Stopbits: 1,2

 Die Anwendung der korrekten Parameter ist Sache des Programmierers,
 die Routine liefert keine Fehlermeldungen. Unüblichere Baudraten sind
 ebenfalls realisierbar. Die Baudrate wird berechnet aus der Taktfrequenz
 des RS-Adapters (1,8432 MHz) dividiert durch 16 (erster Teiler) und die
 gewünschte Baudrate. Die übrigen Parameter werden durch Schreiben des
 entsprechenden Bytes in das Line-Control-Register festgelegt. Die
 Bedeutung der einzelnen Bit ist wie folgt:
           BIT   Bedeutung
           0     Wortlänge Low       (5 =0,0 / 6 =0,1)
           1     Wortlänge High      (7 =1,0 / 8 =1,1)
           2     Zahl der Stopbits, 1 oder 2 (0/1)
           3     Parität Ja/Nein (0/1)
           4     Gerade (Even =1) oder Ungerade (Odd =0) Parität
}

VAR  lineContrByte  : BYTE;

BEGIN {RsInit}
  Port[LineContrReg] := 128;     {ermöglicht das Setzen des Baudratenteilers}
  PortW[BaudRateGen] := ROUND(1843200.0 / 16 / BaudRate);
  lineContrByte := (databits -5) + 4 *(stopBits -1);

  IF parity ='E' THEN lineContrByte := lineContrByte + 24
                 ELSE IF parity ='O' THEN lineContrByte := lineContrByte +8;
  Port[LineContrReg] := lineContrByte;
END;  {RsInit}

{************************************************************************}
PROCEDURE RsClear (VAR buffer: BufferRec);
{************************************************************************}
{Ein Aufruf dieser Prozedur leert den Zwischenspeicher und den Eingangs-
 speicher des RS-Bausteins.}

VAR  dummy  : BYTE;

BEGIN {RsClear}
  dummy := Port[ReceiverReg];              {den RS-Baustein leeren}
  buffer.stop  :=0;                        {den Zwischenspeicher leeren}
  buffer.start :=0;
END;  {RsClear}
```

```
{************************************************************************}
FUNCTION TxReady : BOOLEAN;
{************************************************************************}
{Returns TRUE if the Transmitter ist ready to accept a character }

VAR status : BYTE;

BEGIN {TxReady}
  status := Port[LineStatusReg];
  IF (status AND 48) = 48 THEN TxReady := TRUE
                          ELSE TxReady := FALSE;
END;  {TxReady}

{************************************************************************}
PROCEDURE RsStart (VAR buffer: BufferRec; VAR dsave   : INTEGER);
{************************************************************************}
{Aktiviert den Interrupt-Mechanismus zum Empfang der Zeichen. }

{.....................................................................}
PROCEDURE Set_IVT(intLine : INTEGER);
{.....................................................................}
{Schreibt in die Interrupt-Vektor-Tabelle die Anfangsadresse unserer
 Interrupt-Behandlungsroutine. Die Variable IntLine bestimmt über
 welche Interrupt-Leitung die Unterbrechungsanforderung geschickt wird.


 IntLine 0 entspricht der Interrupt-Leitung 0, usw.}

VAR offset, segment, first_word, second_word  : INTEGER;

BEGIN {Set_IVT}
  first_word             := (intLine +8) *4;
  second_word            := first_word + 2;
  save.first_word        := first_word;
  save.second_word       := second_word;
  save.ivt_first_word    := MemW[$0000:first_word];
  save.ivt_second_word   := MemW[$0000:second_word];
  MemW[$0000:first_word] := Ofs(RsReceiver) +7;   {Adr. der Int.-Routine}
  MemW[$0000:second_word]:= Cseg;
END; {Set_IVT}

{.....................................................................}
  PROCEDURE Enable_IRQx(IRQ: BYTE);
{.....................................................................}

{Ermöglicht die Interrupt-Behandlung. IRQ enthält die Nummer der frei-
 zugebenden Interrupt-Leitung}

VAR imr,mask   : BYTE;

BEGIN {Enable_IRQx}
  mask      := NOT (1 shl IRQ);
  imr       := Port [IntMaskReg];          {get imr from 8259}
  save.imr  := imr;
  imr       := imr AND mask;               {clear mask bit}
  Port[$21] := imr;                        {AND return to controller}
  Port[$21] := 0;
END;  {Enable_IRQx}

BEGIN {RsStart}
  save.auxoutptr := auxoutptr; {Adresse der originalen Ausgaberoutine sichern}
  save.auxinptr  := auxinptr;  {Adresse der originalen Eingaberoutine sichern}
  dsave := Dseg;               {abspeichern der Adresse des Datensegmentes}
                               {Diese Adressen werden gespeichert um beim
                                Verlassen des Programmes den ursprünglichen
                                Zustand wieder herstellen zu können}
```

dazu Abb. 1 die dieses Funktionsprinzip noch verdeutlicht).

Um zu wissen, wo die gültigen Daten anfangen und wo sie aufhören, benötigen wir zwei Hilfsvariablen, Buffer.Start und Buffer.Stop. Der Zeiger Buffer.Stop, der das Ende des gültigen Datenbereiches markiert, wird immer, wenn ein Zeichen von der Schnittstelle gelesen worden ist, um eine Stelle weitergerückt. Holt die Auslese-Routine ein Zeichen aus dem Zwischenspeicher ab, so rückt sie den Zeiger Buffer.Start ebenfalls um eine Stelle vor.

Weisen die beiden Hilfszeiger Start und Stop auf zwei verschiedene Speicherbereiche, so ist eindeutig klar, dass sich noch Zeichen im Zwischenspeicher befinden. Der Test durch die Funktion RScharReady ist somit sehr einfach durchzuführen. Löschen lässt sich der Speicher ebenfalls sehr einfach, indem die beiden Hilfszeiger auf die gleiche Adresse gesetzt werden. Ein kleines Problem musste in der Empfangsroutine noch berücksichtigt werden. Bei einem grossen Datenanfall wäre es nämlich denkbar, dass der Zwischenspeicher zu klein dimensioniert ist. Das würde dazu führen, das der Hilfszeiger Stop plötzlich den Zeiger Start überholt - so wie bei einer Uhr der Minutenzeiger den Stundenzeiger überrunden kann -, was einen Verlust aller dazwischenliegenden Daten bedeutet. Dass sich die Zeiger immer schön auf diesem Trommelspeicher bewegen, dafür sorgt die mathematische Operation der Modula-Division, die in der Speicherverwaltung vorgenommen wird. Kommt Ihnen das Spanisch vor, dann lesen Sie in Ihrem bevorzugten Programmierhandbuch (auch ein BASIC-Buch ist genügend) das Kapitel über die Modula-Divison und betrachten Sie unsere erwähnte Zeichnung nochmals genau.

RSINIT

Dies ist eine Prozedur, die die vollständige Programmierung der in einem IBM-PC-kompatiblen Computer verwendeten seriellen Schnittstelle erlaubt. Sie leistet in etwa das gleiche wie das zum Betriebssystem mitgelieferte Programm MODE, erlaubt jedoch auch die Eingabe eher exotischer Parameter wie z.B. 5 oder 6 Datenbits und Baudraten unter 110 und über 9'600 Baud. Auf ein Abfangen möglicher Eingabefehler wurde bewusst verzichtet, um den Programmausdruck kurz zu halten. Für jeden, der mit der Programmiersprache Pascal umzugehen weiss, sollte

ein Einfügen der erforderlichen Kontrollen zudem ein Leichtes sein. Die für die Programmierung des asynchronen Kommunikationsadapters erforderlichen Informationen entnehmen Sie den technischen Unterlagen zu Ihrem Computer.

RSSTART

Diese Prozedur aktiviert den Interrupt-Mechanismus und weist unsere Routinen dem logischen Treiber AUX zu. Sie enthält zwei weitere Prozeduren, nämlich Set_IVT (macht einen Eintrag in die Interrupt-Vektor-Tabelle) und Enable_IRQx, die eine Interrupt-Anforderung zur Bearbeitung freigibt. Beide Prozeduren entstammen weitgehend dem bereits genannten BYTE-Artikel.

Steht ein Interrupt zur Bearbeitung an, so wird in einer Tabelle, eben der Interrupt-Vektor-Tabelle, die Adresse der zugehörigen Bearbeitungsroutine nachgeschlagen. Welche Zeile der Tabelle die richtige Adresse enthält, lässt sich aus der Nummer des Interrupts berechnen. Unsere Prozedur Set_IVT macht somit nichts anderes, als die Adresse unserer Bearbeitungsroutine (RSreceiver) in diese Tabelle einzutragen. Wie es gemacht wird, sollten Sie aus dem Programm und seiner Beschreibung entnehmen können. Bevor der Eintrag in der Tabelle vorgenommen wird, wird der dort vorhandene alte Wert noch gesichert - in den Record Save - so dass beim Beenden des gesamten Programmes der ursprüngliche Zustand wieder hergestellt werden kann.

RSSTOP

Nach Beenden eines Programmes sollte der Computer wieder in den Zustand gebracht werden, den er auch vor dem Start des Programmes hatte. Dazu dient die Prozedur RSSTOP. Sie wird kurz vor dem Verlassen des Programmes aufgerufen. Sie macht unsere Einträge in die Interrupt-Vektor-Tabelle rückgängig, schreibt die ursprüngliche Interrupt-Maske in das entsprechende Register und weist dem logischen Treiber AUX wieder seine Standardroutinen zu. Diese Daten, die es uns ermöglichen, den ursprünglichen Zustand wieder herzustellen, entnimmt die Prozedur dem Record Save.

RSCLEAR

Möchten Sie den Zwischenspeicher und den Speicher des Schnittstellen-

```
  auxoutptr := Ofs(RsWrite);    {Der Zeiger auf die Ausgaberoutine in
                                 Turbo-Pascal wird so umgestellt, dass er
                                 auf unsere Ausgaberoutine zeigt}
  auxinptr := Ofs(RsRead);      {Der Zeiger auf die Eingaberoutine wird so
                                 umgestellt, dass er auf unsere Eingaberoutine
                                 zeigt}
  Port[ModemContrReg] := $0A; {Bit0: setzt das DTR-Signal,
                               Bit1: setzt RTS,
                               Bit3: ermöglicht Int.-Behandlung. Dieses Bit
                               muss gesetzt sein, ansonsten wird kein
                               Interrupt weitergeleitet. Weitere Informationen
                               dazu im Handbuch Tech. Ref. IBM-PC, Seite 1-188}
  Port[IntEnablReg]   := $01; {Freigabe der Interrupt-Behandlung. 8259}
  RsClear(buffer);
  Set_Ivt(4);
  Enable_IRQx(4);
END;  {RsStart}

{*********************************************************************}
PROCEDURE RsStop (save : SaveRec);
{*********************************************************************}
BEGIN {RsStop;}
  Port[IntMaskReg]             := save.imr;
  auxoutptr                    := save.auxoutptr;
  auxinptr                     := save.auxinptr;
  MemW[$0000:save.first_word]  := save.ivt_first_word;
  MemW[$0000:save.second_word]:= save.ivt_second_word;
END;  {RsStop;}
```

Die wichtigsten Programmiervorschriften

① Sehen Sie einen Programmkopf vor, der alle wesentlichen Informationen enthält.

② Einrückungen entsprechend der Verschachtelungstiefe, machen ein Programm lesbarer. Rücken Sie pro Verschachtelungsstufe eine Zeile um zwei Leerstellen ein.

③ Pro Zeile sollte nur eine Anweisung aufgeführt sein. Ausnahmen die die Lesbarkeit verbessern, sind denkbar.

④ Wählen Sie aussagekräftige Bezeichnernamen. Setzen Sie die Gross/Kleinschreibung systematisch ein.

⑤ Regeln für die Gross-/Kleinschreibung:

- Alle Pascal-Befehle werden GROSS geschrieben
- Variablen-Namen werden alle klein geschrieben
- Konstanten-Namen beginnen mit einem Grossbuchstaben
- Typ-Bezeichner beginnen mit einem Grossbuchstaben
- Bein Namen von Prozeduren und Funktionen wird der erste Buchstabe gross geschrieben. Besteht der Name aus mehreren Wörtern, so beginnt jedes Wort mit einem Grossbuchstaben, z. B.: RsCharReady

Berücksichtigen Sie diese Gross/Kleinschreibe-Vorschriften, so kann jederzeit die Bedeutung eines Bezeichners bestimmt werden, ohne vorerst in der Deklarationsliste nachschlagen zu müssen.

⑥ Ergänzen Sie die End-Klauseln von Funktionen und Prozeduren mit den Namen der entsprechenden Funktion oder Prozedur.

```
Program RStest;
{$Irsserv2.pas}   {Bindet die Prozedursammlung in unser Testprogramm ein }
{$U-}

VAR  c    : CHAR;

BEGIN
  WRITELN('RS-server Testprogramm ');
  WRITELN('Test abbrechen mit CTRL-X ');
  RSinit(1200,'N',8,1);
  RSstart(Buffer,dsave);
  c := ' ';
  REPEAT
    IF Keypressed THEN
      BEGIN
        READ(kbd,c);
        WRITE(Aux,c);
      END;

      IF RSCharREADY(Buffer) THEN
        BEGIN
          READ(aux,c);
          WRITE(c);
        END;
  UNTIL c IN [^x,^X];
  RSstop(save);
END.
```

Das Hilfsprogramm um unsere Routinensammlung zu testen

bausteins löschen, so rufen Sie diese Prozedur auf.

RSCHARREADY

Diese Funktion erlaubt Ihnen zu prüfen, ob sich Daten im Zwischenspeicher befinden.

RSREAD

Diese Prozedur übergibt ein Zeichen aus dem Zwischenspeicher an das aufrufende Programm. Das ausgelesene Zeichen wird im Zwischenspeicher als gelöscht markiert. Befindet sich kein Zeichen im Zwischenspeicher, so müsste in der Routine so lange gewartet werden, bis eines vorhanden ist. Dies könnte zu einem scheinbaren «Hängen» des Programmes führen. In unserer Routine wird dieses Problem umgangen, indem bei leerem Zwischenspeicher ein «End of Line»Zeichen, das Zeichen CR, zurückgemeldet wird. Alle Abfragen des Speichers, die über den logischen Treiber AUX vorgenommen werden, erkennen dann, dass das Ende einer Eingabezeile vorliegt oder dass keine neuen Daten anstehen. Vergleichen Sie auch dazu das Turbo-Pascal-Handbuch, Seite 108.

TXREADY

Mit dieser Funktion kann geprüft werden, ob der Schnittstellenbaustein zu einer Datenübertragung bereit ist.

RSWRITE

Mittels dieser Prozedur lassen sich Daten über die RS-Schnittstelle aussenden. Um das ganze Programm nicht zu stark aufzublähen, wurde diese Prozedur sehr einfach gehalten. Es ist jedoch eine Erweiterung in dem Sinne denkbar, dass die dieser Prozedur übergebenen Zeichen nicht direkt ausgesendet werden, sondern ebenfalls in einem Zwischenspeicher abgelagert werden. Der Schnittstellenbaustein könnte dann die Daten Interrupt-gesteuert, asynchron zum übrigen Programmablauf, auslesen.

RSRECEIVER

Dies ist die eigentlich zentrale Prozedur unseres ganzen Programmes. Sie wird über die Interrupt-Anforderung unabhängig vom übrigen Programmablauf aufgerufen. Zu Beginn werden alle in den Registern des Prozessors enthaltenen Daten gesichert. Denn nur so ist es möglich, dass der Prozessor nach Abschluss der Interrupt-Routine die Arbeit an der Stelle wieder aufnehmen kann, an der er unterbrochen wurde. Dazu werden über Assemblerbefehle alle Daten auf dem Stack (dem Stapelspeicher) gespeichert. Es ist erforderlich, alle Register zu speichern, weil wir unsere Interrupt-Routine ja in einer Hochsprache schreiben und somit im allgemeinen nicht wissen können, welche Register durch die Routine verändert werden.

Die zweite Inline-Zeile hilft uns, ein DOS-spezifisches Problem zu umschiffen. Das Datensegment des BIOS ist nicht das gleiche wie das vom Anwendungsprogramm benüzte. Erfolgt der Interrupt innerhalb des Programmes, so stört das in keiner Weise, erfolgt er jedoch, während das Programm auf eine Betriebssystem-Funktion zugreift, so ist das Chaos perfekt. Unser Interrupt-Programm wird nämlich irgendwann auf eine Pascal-Variable zugreifen, sich dabei jedoch gründlich verrennen da es zur Bestimmung der Adresse das Datensegment des BIOS hinzuzieht. Um das zu verhindern, laden wir in dieser zweiten Assemblerzeile das DS-Register mit dem Wert des Datensegmentes für das Pascal-Programm.

Jetzt endlich ist es uns möglich, die eigentliche Aufgabe unserer Interrupt-Routine auszuführen, also den Schnittstellenbaustein zu lesen, das Zeichen im Zwischenspeicher abzulagern und die Speicherorganisation auf den neuesten Stand zu bringen.

Sobald unsere zentrale Interrupt-Arbeit beendet ist, geben wir dies dem Interrupt-Controller bekannt, indem wir ihm ein Signal End of Interrupt (EOI) übermitteln. Dann erstellen wir wieder die ursprünglichen Registerinhalte und verlassen die Interrupt-Routine.

Die gesamte Interrupt-Routine, wie sie hier vorliegt, benötigt weniger als eintausend Maschinenzyklen, das heisst ihre Abarbeitung erfolgt auf einem normalen «lahmen» IBM-PC in etwa 0.2 Millisekunden (ms). Dies ist genügend kurz, um auch bei Baudraten von 9'600 Baud - wo etwa alle 1 ms ein Interrupt ausgelöst wird - zuverlässig zu arbeiten. □

Spline-Interpolation einmal anders

Eine gegebene Punktefolge lässt sich mit einem Kurvenlineal zu einem stetig verlaufenden Kurvenzug verbinden. Dasselbe lässt sich auch mit einem Rechner über Spline- und Interpolations-Algorithmen lösen. Ueblicherweise weichen die maschinell erstellten Kurven von den «idealen» Kurvenlinealzügen stark ab. Der in diesem Beitrag vorgestellte Spline-Algorithmus bringt eine bessere Annäherung.

R. Kress

Sind von einer unbekannten Funktion nur wenige Stützpunkte bekannt, so lassen sich mit unterschiedlichen Interpolationsverfahren Funktionen finden, die für den jeweiligen Anwendungszweck befriedigende Lösungen bieten. Werden die Kurvenabschnitte von Stützpunkt zu Stützpunkt über stetig differenzierbare Polynome gewonnen, so spricht man von Polynomspline oder kurz Spline. Wobei sich der Kurvenzug gleich einer dünnen Latte (= spline) durch die Stützpunkte zieht. Sein Verlauf hängt zwangsläufig von der gewählten Methode ab. Der Name «Spline» soll hier für ein Verfahren beibehalten werden, das von oben angeführtem abweicht, dafür aber echten Kurvenlinealzügen näher kommt.

Zum Vergleich drei Beispiele für die gleiche Stützpunktfolge. Abb. 1 ist mit der Interpolation aus M+K 85-4 erstellt. *(Da dieses Heft vergriffen ist, stellt der Verlag gegen Einsendung eines adressierten und frankierten C5-Umschlages den entsprechenden Artikel als Fotokopie zur Verfügung. Red.)* Das Ergebnis eines kubischen Spline-Algorithmus zeigt Abb. 2, und Abb. 3 schliesslich das Resultat des hier beschriebenen Lösungswegs. Das vorliegende Programm zeichnet zum Zwecke allgemeiner Anwendbarkeit auf den Bildschirm. Die verwendete Sprache ist BASICA, das erweiterte BASIC auf dem IBM-PC. Wer die Erweiterung nicht besitzt, braucht nur die Befehle VIEW (Bildausschnitt) und WINDOW (Skalierung) in den Zeilen 170 und 175 zu ersetzen. Aus dem Listing geht ab Zeile 675 hervor, was ergänzt und ersetzt werden muss. Jede Stützpunktkoordinate wird dann in eine entsprechende Funktion gekleidet. Die Aenderung ist auch bei Kompilierung nötig, wenn der BASIC-Uebersetzer VIEW und WINDOW nicht versteht.

Neben den vier Grundrechnungsarten ist die Quadratwurzelfunktion erforderlich. Obwohl mit trignonometrischen Formeln gerechnet wird, werden Winkelfunktionen nicht gebraucht, da die Darstellungsebene nicht wechselt. Und jetzt zum Verfahren selbst.

Gehen wir davon aus, dass es zwei grundsätzliche Wege für den Einkurvungsalgorithmus gibt. Zum einen das Finden einer mehr oder weniger aufwendigen Funktion, die mehrere oder alle gegebenen Stützpunkte einschliesst. Zum anderen nur die periodische Abwicklung von Stützpunkt zu Stützpunkt. Der erste Weg - dazu gehören z.B. kubische Splines - stellt von vornherein sicher, dass der Linienzug stetig durch die Stützpunkte verläuft. Von Nachteil ist offenbar, dass bei einmal festgelegtem Ansatz auch die Funktion determiniert ist und bei verschiedenen Ansätzen wegen der Forderung stetiger Differenzierbarkeit, keine grossen Ergebnisunterschiede im Hinblick auf die Zielsetzung zu erwarten sind.

Beim zweiten Weg liegt das Problem bei den Verbindungen der Kurvensegmente, d.h. jeder Kurvenabschnitt muss fliessend in den nächsten übergehen. Doch es besteht die Chance, die Segmente freier zu gestalten. Dieser Weg soll jetzt beschritten werden.

Die beiden Randpunkte ausgenommen, sei nach Abb. 4 durch jeden Stützpunkt P_i eine Gerade g_i derart gelegt, dass die Verbindungsstrecken zu den benachbarten Punkten gleiche Winkel mit g_i bilden, d.h. $\varphi_{1b}=\varphi_{2a}, \varphi_{2b}=\varphi_{3a}$ oder allgemein $\varphi_{ib}=\varphi_{(i+1)a}$ (i = Kurvensegmentzähler).

Der Winkel φ_i ist der Uebergangswinkel zweier Kurvensegmente und berechnet sich in Altgrad aus $90 - \delta_i/2$. Es gilt

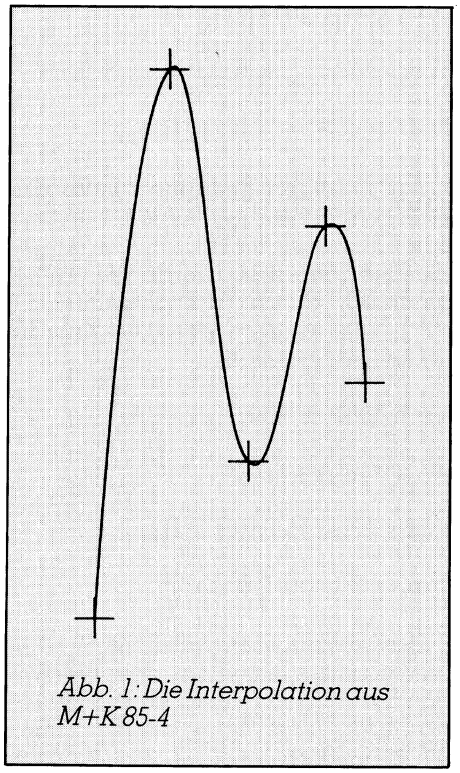


Abb. 1: Die Interpolation aus M+K 85-4

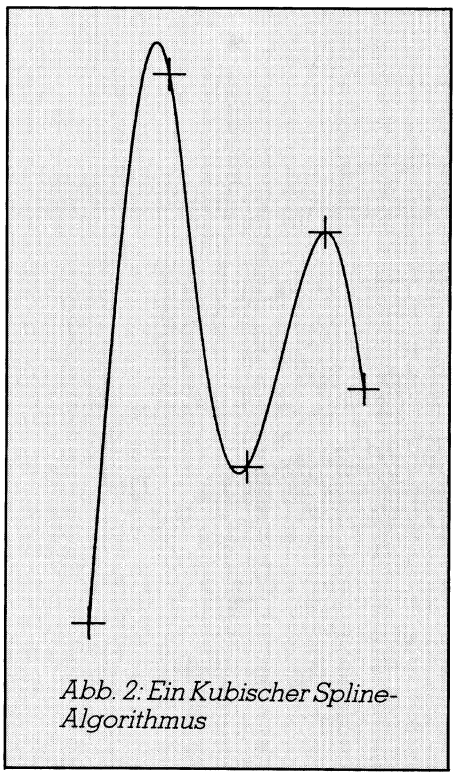


Abb. 2: Ein Kubischer Spline-Algorithmus

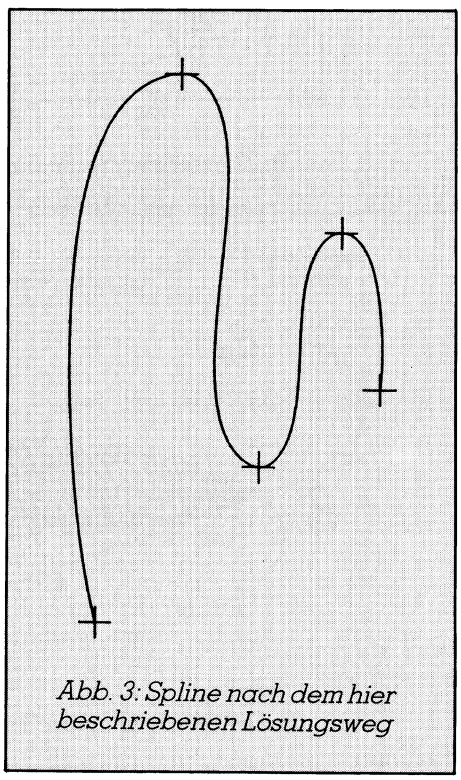


Abb. 3: Spline nach dem hier beschriebenen Lösungsweg

$$\cos\delta_i =$$

$$(P_{i-1}(x)-P_i(x))(P_{i+1}(x)-P_i(x))+(P_{i-1}(y)-P_i(y))(P_{i+1}(y)-P_i(y))$$

$$\sqrt{(P_{i-1}(x)-P_i(x))^2+(P_{i-1}(y)-P_i(y))^2} * \sqrt{(P_{i+1}(x)-P_i(x))^2+(P_{i+1}(y)-P_i(y))^2}$$

Für ein Kurvensegment sind zwei mögliche Fälle zu unterscheiden, z.B. die Strecken P_3P_4 und P_2P_3. Der erste sei nach dem Streckenzug $P_2P_3P_4P_5$ mit U-Form, der zweite nach dem Streckenzug $P_1P_2P_3P_4$ mit Z-Form bezeichnet.

Zuerst zur U-Form in Abb. 5a. Wir greifen aus Abb. 4 die Stützpunkte P_3 und P_4 heraus und drehen sie vereinfachend auf 0 Grad, so dass P_3P_4 zur waagrechten Strecke $P'_3P'_4$ wird. Nun werden durch P'_3 und P'_4, den Winkeln φ_{3a} und φ_{3b} entsprechend, zwei Ellipsen E_1 und E_2 gelegt. Für die Ellipse E_1 ist φ_{3a} der Winkel, den zwei symmetrische Tangenten durch P'_3 und P'_4 bilden. Anders ausgedrückt: Die Strecke $P'_3P'_4$ liegt auf der Polaren, deren Pol im Schnittpunkt der symmetrischen Ellipsentangenten liegt, die für E_1 den Winkel φ_{3a} und für E_2 den Winkel φ_{3b} einschliessen. Um den Rechenweg zu vereinfachen, wird der Betrag der Strecke $P'_3P'_4$ zu 1 normiert. Zur Bestimmung einer Ellipse mit den Halbachsen a und b wird $b/a=K$ gesetzt. Das gesuchte Kurvenstück liegt innerhalb der schraffierten Differenz der beiden Ellipsensegmente.

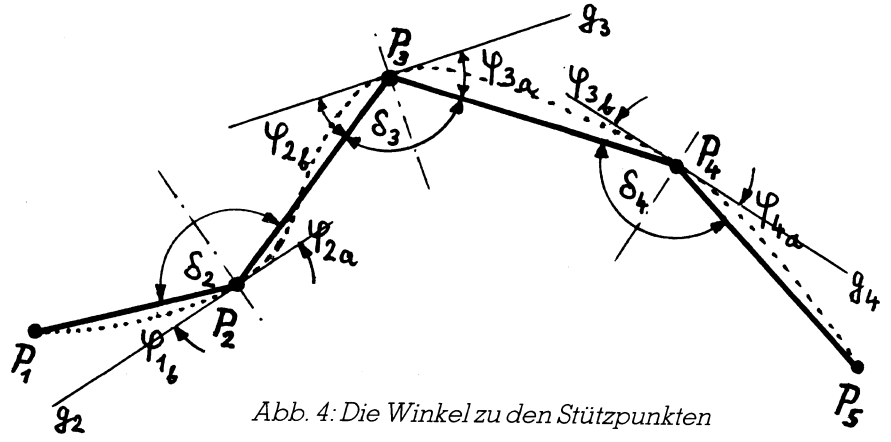


Abb. 4: Die Winkel zu den Stützpunkten

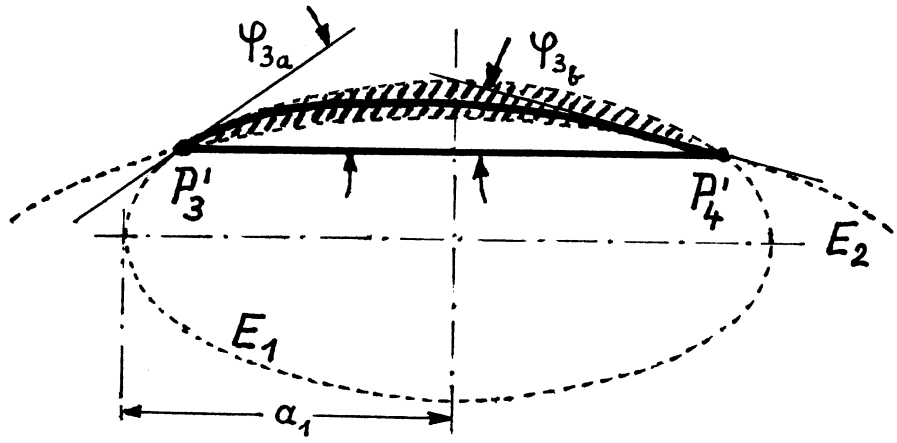


Abb. 5a: Die Ellipsensegmente bei der U-Form

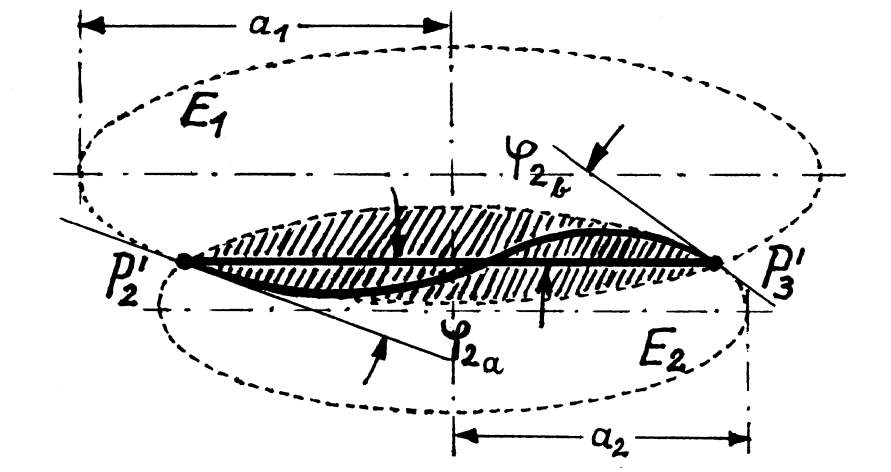


Abb. 5b: Die Ellipsensegmente bei der Z-Form

Für die Höhe D des normierten Ellipsensegments an der Stelle $x_{,,}$ mit dem Anfangs- und Endwinkel φ gilt

$$D = K\sqrt{a^2-(x_{,,}-0{,}5)^2}-K^2/(2*\tan\varphi)$$

für $0<=x_{,,}<=1$

$$a^2 = 0{,}25+K^2/(4*\tan^2\varphi)$$

Setzt man D_1 für die Segmenthöhe mit φ_{3a} und D_2 für die Höhe mit φ_{3b}, so ermittelt sich die Höhe des gesuchten Kurvensegments an der Stelle $x_{,,}$ aus

$$y_{,} = D_2 + x_{,,}^2(D_1-D_2)$$

für $0<=x_{,,}<=1$

Für die Z-Form ist die Winkellage φ_{2a} verändert (siehe Abb.5b). Die Ellipsensegmente berechnen sich analog der U-Form. Das gesuchte Kurvensegment ergibt sich aus

$$y_{,} = D_2 - Y_h(|D_1|+|D_2|)$$

$$Y_h = x_{,,}(\sin\varphi_{2a}/\sin\varphi_{2b}(1-x_{,,})+x_{,,})$$

für $0<=x_{,,}<=1$ und $\varphi_{2a}<\varphi_{2b}$

Zur richtigen Einfügung muss die Streckennormierung aufgehoben und das Kurvensegment zur Ausgangslage zurückgedreht werden. Das eine geschieht durch Multiplikation mit P_iP_{i+1}, das andere über die Koordinaten-Transformation

$$x = x'\cos\psi - y'\sin\psi$$

$$y = x'\sin\psi + y'\cos\psi$$

$$\sin\psi = (P_{i+1}(y)-P_i(y))/P_iP_{i+1}$$

$$\cos\psi = (P_{i+1}(x)-P_i(x))/P_iP_{i+1}$$

Das Halbachsenverhältnis K wird im Programm festgelegt. Es verändert sich dynamisch in Relation der Strecken P_iP_{i+1} und $P_{i+1}P_{i+2}$ zueinander, um den Kurvenverlauf von grossen

Stützpunktabständen, dem der danebenliegenden kleineren Abständen anzupassen. So ist

$$K = K_f \sqrt{S_v(2-S_v)}$$

für $S > S_s$ gilt $S_v = S_s/S$, sonst $S_v = 1$

$$S = P_iP_{i+1}$$

$$S_s = P_{i+1}P_{i+2}$$

K_f ist festgelegt und beträgt für die U-Form 0,5 und für die Z-Form 0,35.

Die Beginn- und Endwinkel φ_{1a} bzw. φ_{nb} werden aus ihren gegenüberliegenden Winkeln φ_{1b} bzw. φ_{na} ermittelt. Für sie wurde bestimmt

$$\varphi = \sin \varphi, (1\text{-}0{,}6 * \sin^2 \varphi, (3\text{-}2 * \sin \varphi,)),$$

φ, ist der gegenüberliegende Winkel.

Zurück zum Programm. Es gliedert sich in vier Teile:

1. Datenteil
2. Dimensionierung der Zeichenfläche (Skalierung)
3. Treiberteil (Einlesen der Stützpunkte und Prüfung)
4. Spline-Algorithmus

Im Datenteil sind die Stützpunkte in DATA-Zeilen abgelegt. Mit RESTORE in Zeile 50 lässt sich gezielt aufsetzen. Die Steuerdaten *,0 bzw. *,1 und *,* stehen für Neubeginn eines offenen, eines geschlossenen und beenden eines Kurvenzugs. Für Kommentare in DATA-Zeilen wurde «:REM» und nicht das Hochkomma gebraucht, da es in der benutzten BASIC-Version A2.00 als Dateneintrag gelesen wird. Die Stützpunktwerte sind in X/Y-Gruppen einzutragen.

Die Zeichenflächen-Dimensionierung legt eine quadratische Bild-

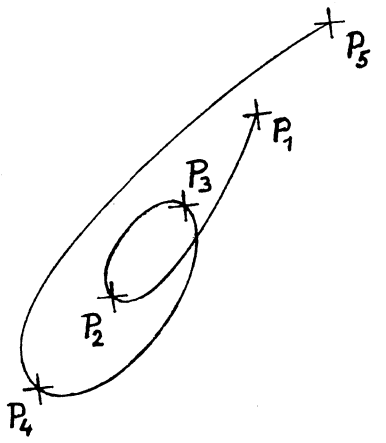


Abb. 6: Kurvenzug mit den Stützpunkten aus Zeile 75

```
10   ' ------------------------------------------------------------------------
15   ' Programm-Name: SPLINE       IBM-PC, BASICA A2.00       (C) R. Kress 1986
20   ' ------------------------------------------------------------------------
25   ' Eingabe der Stuetzpunkte in DATA-Zeilen (X/Y-Gruppen)
30   ' Steuerdaten: *,1    = Neubeginn und Kurve schliessen
35   '              *,0    = Neubeginn und Kurve nicht schliessen (default)
40   '              *,*    = Ende einer Serie, am Datenende nicht erforderlich
45   ' ------------------------------------------------------------------------
50  RESTORE 75  ' aufsetzen auf DATA-Zeile (DATA-Start)
55    DATA 3,2, 4,9, 5,4, 6,7, 6.5,5, *,*
60    DATA 1,2, 2,2.5, 3,4, *,0, 3,4, 3,2, 5,2.5, *,1, 3.3,4.9, 3.4,5.2, *,* :REM "i"
65    DATA *,0, 1,3, 3,1, 4,2, 5,3, 4,4, *,*
70    DATA *,1, 2,5, 8,5, *,* :REM   Elipse
75    DATA *,0, 4,4, 2,2, 3,3, 1,1, 5,5, *,*
80    DATA *,1, 8,5, 7.6,6.5, 6.5,7.6, 5,8, 3.5,7.6, 2.4,6.5, 2,5, 2.4,3.5, 3.5,2.4,
5,2, 6.5,2.4, 7.6,3.5, *,*  :REM   Kreis
85  '
90  ' ---------  Dimensionierung der Zeichenflaeche ---------
95   AUFLOES=2  ' 1=mittlere, 2=hohe Aufloesung
100  IF AUFLOES=1 THEN SCREEN 1:XPUNKTE=222 ELSE SCREEN 2:XPUNKTE=444
105  ' Schirmteilung (Zeichenflaeche in Pixel, Nullpunkt links oben):
110  XPMIN=0       ' kleinerer X-Pixelwert (min. 0)
115  XPMAX=XPUNKTE ' groesserer X-Pixelwert (max. 319 bzw. 638 je nach Auflsg.)
120  YPMIN=0       ' kleinerer Y-Pixelwert (min. 0)
125  YPMAX=199     ' groesserer Y-Pixelwert (max. 199)
130  '
135  ' Skalierung (Dimensionierung der Zeichenflaeche in beliebigen Einheiten,
140  '            Maxwerte <> Minwerte, Nullpunkt links unten):
145  SCALXMIN=0     ' X-Skalierung, Untergrenze <==========
150  SCALXMAX=10    '               Obergrenze  <==========
155  SCALYMIN=0     ' Y-Skalierung, Untergrenze <==========
160  SCALYMAX=10    '               Obergrenze  <==========
165  '
170  VIEW (XPMIN,YPMIN)-(XPMAX,YPMAX)
175  WINDOW (SCALXMIN,SCALYMIN)-(SCALXMAX,SCALYMAX)
180  XSCHRITTFAKTOR=(XPMAX-XPMIN)/XPUNKTE/(SCALXMAX-SCALXMIN)
185  YSCHRITTFAKTOR=(YPMAX-YPMIN)/199/(SCALYMAX-SCALYMIN)
190  LINE (SCALXMIN,SCALYMIN)-(SCALXMAX,SCALYMAX),,B ' Begrenzungsrahmenanzeige
195  '
200  ' ************************** Hauptprogramm ****************************
205  OPTION BASE 0
210  DIM X(100),Y(100) ' hier fuer max. 100-2= 98 Stuetzpunkte ausgelegt
215  SCHRITT=.3  ' Schrittlaenge als %-Wert der gleichseitigen Bildschirm-View
220  '           (XPUNKTE bzw. 199). Sie teilt die Punktverbindungsstrecke S
225  '           im Spline-UPGM  (tatsaechliche Zuglaenge >= Schritt, je nach
230  '           Steigung, wird auf ganzzahligen Teiler gerundet)
235  IF SCHRITT<=0 THEN SCHRITT=1 ' default
240  ENDMARKE=0:SCHLIESSEN=0:SCHRITTWEITE=SCHRITT/100
245  ANZ=0:ON ERROR GOTO 350
250  READ X$,Y$:X=VAL(X$):Y=VAL(Y$)
255  IF X$="*" AND Y$="*" THEN ENDMARKE=1:GOTO 280
260  IF Y$="*" OR X$="*" AND Y<>0 AND Y<>1 THEN SCREEN 0:PRINT "X,Y-Einlesedaten
unpaarig bzw. falsch!":BEEP:END
265  IF X$="*" THEN IF ANZ=0 THEN SCHLIESSEN=Y:GOTO 250 ELSE SCHLIESSENNEU=Y:GOTO
280 ' SCHLIESSEN:  0 = Kurvenzug offen, 1 = schliessen
270  ANZ=ANZ+1:X(ANZ)=X:Y(ANZ)=Y:GOTO 250
275  '
280  ON ERROR GOTO 0 ' ON ERROR inaktivieren
285  FOR I=1 TO ANZ-1 ' aufschieben bei folgegleichen Punkten
290    IF X(I+1)-X(I)<>0 OR Y(I+1)-Y(I)<>0 THEN 315
295    FOR J=I+1 TO ANZ-1
300      X(J)=X(J+1):Y(J)=Y(J+1)
305    NEXT J
310    ANZ=ANZ-1:GOTO 285
315  NEXT I
320  IF ANZ<2 THEN PRINT"Mindestens 2 Stuetzpunkte in einer Serie erforderlich!
ABBRUCH!":BEEP:END
325  GOSUB 425  ' Spline-Aufruf
330  GOSUB 630  ' Stuetzpunkte zum Test markieren
335  IF ENDMARKE=0 THEN SCHLIESSEN=SCHLIESSENNEU:GOTO 245
340  END
345  ' ----- Fehlerroutine ------
350  IF ERR=4 THEN ENDMARKE=1:RESUME 280 ELSE SCREEN 0:PRINT "Syntax- oder
Datenfehler in Zeile";ERL:BEEP:END
355  '
360  ' ---------------------- Spline-Unterprogramm -------------------------
365  ' Uebergabeparameter:  X(*), Y(*)  =  Stuetzpunktfelder (Beginn Index 1,
370  '                                     Index 0 intern verwendet)
375  '                              ANZ  =  Anzahl der Stuetzpunkte
380  '                        SCHLIESSEN =  Steuerung Kurvenschliessen
385  '                                     0 = offen, 1 = schliessen
390  '                      SCHRITTWEITE  =  Kurvenschrittweite
395  '                                     bei <=0 wird 0.01 angenommen
400  '                    XSCHRITTFAKTOR  =  Skalierungsabhaengiger Faktor (X),
405  '                                     bei <=0 wird 1 angenommen
410  '                    YSCHRITTFAKTOR  =  Skalierungsabhaengiger Faktor (Y),
415  '                                     bei <=0 wird 1 angenommen
420  ' ------------------------------------------------------------------------
425  DX=1E-37 ' DX = kleinster darstellbarer Betrag
```

```
430  GW=3162  ' GW^2 muss mit allen Stellen darstellbar sein
435  IF XSCHRITTFAKTOR<=0 THEN XSCHRITTFAKTOR=1 ' default
440  IF YSCHRITTFAKTOR<=0 THEN YSCHRITTFAKTOR=1 ' default
445  IF SCHRITTWEITE<=0 THEN SCHRITTWEITE=.01   ' default
450  AN=ANZ:IF SCHLIESSEN<>0 AND X(1)=X(AN) AND Y(1)=Y(AN) THEN AN=AN-1
455  IF SCHLIESSEN=0 THEN IVON=1:IBIS=AN-1:SA=1:SB=1 ELSE IVON=0:IBIS=AN:X(0)=X(AN)
:Y(0)=Y(AN):X(AN+1)=X(1):Y(AN+1)=Y(1):X(AN+2)=X(2):Y(AN+2)=Y(2)
460  PSET (X(1),Y(1)) ' zum 1. Zeichenpunkt bewegen
465  FOR I=IVON TO IBIS
470    SX=X(I+1)-X(I):SY=Y(I+1)-Y(I):SXY=SX*SX+SY*SY:S=SQR(SXY+DX)
475    IF SCHLIESSEN=0 AND I=IBIS THEN SB=SA*(1-.6*SA*SA*(3-2*SA)):CB=SQR(1-SB*SB)
:K2=K1:GOTO 510
480      SX2=X(I+2)-X(I+1):SY2=Y(I+2)-Y(I+1):SXY2=SX2*SX2+SY2*SY2
485      SS=SQR(SXY2+DX):CS=(-SX*SX2-SY*SY2)/S/SS:S2=SY2/SS:C2=SX2/SS
490      K2=SGN(SGN((SY*C2-SX*S2)/S)+.5)
495      SB=SQR(ABS(1+CS)/2+DX):CB=SQR(ABS(1-CS)/2+DX)
500      IF I=0 THEN 605 ' wenn SCHLIESSEN<>0
505      IF I=IVON THEN SA=SB*(1-.6*SB*SB*(3-2*SB)):CA=SQR(1-SA*SA):K1=K2
510    IF SA<SB THEN KV=K1:TA=SA/SB ELSE KV=K2:TA=SB/SA
515    IF S>SS THEN SV=SS/S+DX ELSE SV=1
520    IF K1=K2 THEN KF=.5 ELSE KF=.35  ' K1=K2 (U-Form), K1<>K2 (Z-Form)
525    K=KF*SQR(SV*(2-SV)):KK=K*K:KH=KK/2
530    YA=KH*CA/(SA+DX):IF YA>GW THEN YA=GW
535    YB=KH*CB/(SB+DX):IF YB>GW THEN YB=GW
540    A2=YA/KK*YA+.25:B2=YB/KK*YB+.25
545    SD=SY/S:CD=SX/S ' SD,CD Drehwinkel (SIN, COS)
550    XLAENGE=SX*XSCHRITTFAKTOR:YLAENGE=SY*YSCHRITTFAKTOR
555    SP=1/-INT(-SQR(XLAENGE*XLAENGE+YLAENGE*YLAENGE+DX)/SCHRITTWEITE)
560    FOR XH=0 TO 1+SP/2 STEP SP
565      IF I>1 AND XH=0 THEN XH=SP
570      XQ=XH-.5:XQ=XQ*XQ
575      D1=K*SQR(ABS(A2-XQ)+DX)-YA:D2=K*SQR(ABS(B2-XQ)+DX)-YB
580      IF SA<SB THEN D=D1:XX=XH ELSE D=D2:XX=1-XH
585      IF K1=K2 THEN YS=S*K1*(D+XX*XX*ABS(D1-D2)) ELSE YH=XX*(TA*(1-XX)+XX):YS=
S*KV*(D-YH*(ABS(D1)+ABS(D2)))
590      XS=XH*S:X=XS*CD-YS*SD+X(I):Y=XS*SD+YS*CD+Y(I)
595      LINE -(X,Y) ' Linie zu X,Y
600    NEXT XH
605    SA=SB:CA=CB:K1=K2:SS=S
610  NEXT I
615  RETURN
620  '
625  ' -------- Test-Unterprogramm Stuetzpunkte markieren --------
630  HALBSTRICH=4/2 ' HALBSTRICH als %-Wert der Bildschirm-View (XPUNKTE,199)
635  MX=HALBSTRICH/XSCHRITTFAKTOR/100:MY=HALBSTRICH/YSCHRITTFAKTOR/100
640  FOR I=1 TO ANZ
645    LINE (X(I)-MX,Y(I))-(X(I)+MX,Y(I))
650    LINE (X(I),Y(I)-MY)-(X(I),Y(I)+MY)
655  NEXT I
660  RETURN
665  '
670  ' ------------------------------------------------------------------
675  '  Wenn das erweiterte BASIC nicht verfuegbar ist, dann ergaenzen:
680  ' 102  DEF FNX(XORIG)=XPMIN+SCALXV*(XORIG-SCALXMIN)
685  ' 103  DEF FNY(YORIG)=199-SCALYV*(YORIG-SCALYMIN)
690  '
695  '  und zu aendern ist in:
700  ' 170  SCALXV=(XPMAX-XPMIN)/(SCALXMAX-SCALXMIN)
705  ' 175  SCALYV=(YPMAX-YPMIN)/(SCALYMAX-SCALYMIN)
710  ' 190  LINE (FNX(SCALXMIN),FNY(SCALYMIN))-(FNX(SCALXMAX),FNY(SCALYMAX)),,B
' Begrenzungsrahmenanzeige
715  ' 460  PSET FNX(X(1)),FNY(Y(1))  ' zum 1. Zeichenpunkt bewegen
720  ' 595  LINE -(FNX(X),FNY(Y)) ' Linie zu X,Y
725  '
730  '  ferner im Unterprogramm "Stuetzpunkte markieren"  alle X(I) bzw. Y(I)
735  '  in FNX(X(I)) bzw. FNY(Y(I)) umsetzen.
740  ' ------------------------------------------------------------------
```

schirm-Zeichenfläche fest um Verzerrungen leichter zu vermeiden. In die Zeilen 145 bis 160 sind die Bemassungsgrenzen einzutragen. Das ist sehr vorteilhaft. Der Benutzer wird der lästigen Pixelberechnung enthoben und verwendet gleich seine gewünschten Masseinheiten. Werden X- und Y-Einheiten nicht gleich gross gewählt, so kommt es zu Darstellungsverzerrungen.

VIEW legt den Bildschirmausschnitt fest und WINDOW ist der Skalierungsbefehl. Die parametrisierten Attribute sind oben bei «Schirmteilung» und vor allem bei «Skalierung» ab Zeile 145 einzutragen. X- und YSCHRITTFAKTOR sind Uebergabevariablen zur Schrittgrössenberechnung.

Das Hauptprogramm beginnt mit dem Treiberteil. Die Felddimensionierung in Zeile 210 ist mindestens um 2 grösser zu wählen als die maximale Stützpunktanzahl. Die Schrittlänge in Zeile 215 wird als %-Wert der Kantenlänge des quadratischen Zeichenausschnitts angegeben. Sie bleibt bei Skalierungsveränderungen konstant. Bei einer Kantenlänge am Bildschirm von z.B. 175 mm und einer Schrittlänge von 0,3 % beträgt die nominelle Schrittweite 175/100*0,3 = 0,525 mm. Bekanntlich setzt die Grafikauflösung Grenzen.

Es folgen Variableninitialisierung und Einleseroutine nebst Fehlerprüfung. Ab Zeile 285 werden folgegleiche Stützpunkte (X,Y identisch mit Vorgänger) eliminiert. Das ist nicht unbedingt erforderlich, schlimmstenfalls bricht das Programm ab.

Nach dem Spline-Aufruf in Zeile 325 können zum Test die Stützpunkte markiert werden. Auch hier bleiben die Markierungskreuze bei Massstabsänderungen konstant. In Zeile 335 wird auf Ende der Eingabedaten geprüft.

Im Spline-Unterprogramm liegt in den Zeilen 425 und 430 die Einstellung auf die Rechnergenauigkeit. Es folgt die Variableninitialisierung und die Anlage zusätzlicher Stützpunkte für geschlossene Kurvenzüge. In Zeile 460 wird zum ersten Zeichenpunkt bewegt.

Im I-Loop ab Zeile 465 wird $\cos\delta_i$ berechnet (Variable CS). K2 in Zeile 490 ist die Vorzeichenbestimmung nach Lage der Kurvensegmente. K2 kann -1 oder +1 annehmen und wird am Schleifenende in K1 gestellt. K1 und K2 dienen auch zur Identifizierung von U- und Z-Form (K1=K2 für U-Form, sonst Z-Form) und treten zur Lagebestimmung als Faktoren auf. Die Variable SB beinhaltet den $\sin\varphi_{ib}$ und CB den $\cos\varphi_{ib}$. A2 in Zeile 540 entspricht dem Quadrat der Ellipsenhalbachse von E_1, desgleichen B2 für E_2. Die Berechnung erfolgt aus YA

COMPUTER SPLITTER

Weniger Architektur bei IBM

(500/fp) Die Kenner werden in Zukunft auf einen klitzekleinen Unterschied in der Nomenklatur der Ankündigungen von IBM schauen - ist's jetzt eine Architektur, die da angekündigt wird oder ist's ein Datastream? Beides ist nämlich dasselbe, logisch nicht? Was soll das: Die EG hat der IBM verordnet, dass sie alle Architekturen zu publizieren habe. Und IBM reagiert wie IBM: Benenne das Ganze um und schon greifen die Vorschriften nicht mehr. □

bzw. YB, um die Korrektur auf GW in den Zeilen 530 und 535 mitzunutzen.

Die dynamische Stepweite SP wird in Zeile 555 errechnet. Die innere Schleife ab Zeile 560 überstreicht die normierte Strecke 1. Der Endwert 1+SP/2 kompensiert kleine Ungenauigkeiten der Aufsummierung, d.h. die Endsumme ist maximal 1+Summierungstoleranz. Zeile 565 verhindert doppelte Ausgaben bei Kurvensegmentwechsel. Es folgt die Berechnung der Höhen der Segmentabschnitte, Rückdrehung und Ausgabe in Zeile 595. Der Absolutbetrag unter der Wurzel in Zeile 575 verhindert negative Werte. Nach Uebergabe der Arbeitsparameter in Zeile 605 kommt das nächste Kurvensegment an die Reihe.

Im Datenteil sind Musterdaten angeführt. Die Figuren aus den Zeilen 70 und 75 (siehe Abb. 6) lassen erkennen, wie der Algorithmus auf entgegengesetzten Richtungswechsel reagiert, nämlich mit Uebergangswinkel von jeweils 90 Grad. Je kleiner bei der Berechnung von D der tan φ wird, desto mehr nähert sich D einem Kreisbogen. Das geht auch aus der Tatsache hervor, dass sich bei konstanter Ellipse mit dem Winkel das Ellipsensegment verkleinert und einem Kreissegment annähert.

Wird die Konstante K_f (Variable KF in Zeile 520) vergrössert, so erhöht sich das Ellipsensegment um so mehr, je grösser der Uebergangswinkel φ_i wird. Die Voreinstellung $K_f=1$ ergibt bei $S_v=1$ das Segment eines Kreises als Sonderform der Ellipse. □

COMPUTER SPLITTER

Fast untergegangen: SAA

(501/fp) Im Rummel rund um die Ankündigung der «neuen Generation von Personal Computern» ist eine kurze Zeit zuvor erfolgte Ankündigung von IBM fast untergegangen, die SAA, die Systems Application Architecture. Mit dem Konzept der SAA will IBM alle Hersteller von Applikationen dazu zwingen, eine einheitliche Linie bezüglich der Bedienung, der Interfaces, der Datenkommunkation usw. zu wahren. Fernziel ist eine Programm- und Daten-Portabilität aller Applikationen über die von vom SAA-Konzept unterstützten Systeme hinweg. Der aufmerksame Leser der Ankündigungen von IBM entnimmt die wesentlichsten Informationen zwischen den Zeilen, er achtet zum Beispiel darauf, was nicht erwähnt wird. So werden folgende höheren Programmiersprachen im Rahmen von SAA erwähnt: COBOL, FORTRAN 77, C, daneben natürlich die Datenbank-Tools und GDDM für die integrierte Darstellung von Grafik, Daten und Text. □

Big Bang in Zürich

(502/fp) Wir erinnern uns: Als Big Bang würdigte man zum voraus den Tag, als die Londoner Börse auf den computerisierten Handel übergehen sollte. Big Bang würde ein grosser Crash Down. Wenn sich M+K ausnahmsweise einmal als Anlageberater betätigen darf, dann gilt folgender Rat: Schweizer Anleger, sichert Eure Papiere! Im kommenden Herbst geht die Zürcher Börse zum «Computerunterstützten Handelssystem» (CHS) über. □

Prozess-orientiertes Programmieren in Logo

Logo-Prozeduren werden in ihrem Ablauf als Prozesse betrachtet. Die sich daraus ergebenden Folgerungen unterstützen zielbewusstes Programmieren. Die grafische Darstellung der Prozess-Dynamik wird mittels einer Klasse von rekursiven Logo-Oszillatoren näher untersucht.

Dr. Branco Milicevic

Das hier diskutierte Thema in Logo gehört eigentlich zu den allgemeinen Grundlagen des Programmierens. Beiträge zu diesem Themenkreis sind in der Literatur bestens bekannt. Dazu nur zwei Beispiele: N. Wirth [1,2], der dabei Pascal (neuestens auch Modula-2) benützt und Abelson, Sussman/Sussman [3], die sich des Lisp-Dialektes «Scheme» bedienen. Diese hervorragende Werke werden hier als ideenmässiger Leitfaden benützt, wobei selbstverständlich die Besonderheiten von Logo im Vordergrund stehen. Eine wichtige dieser Besonderheiten sei kurz erwähnt. Logo verfügt über eine interaktive TRACE- und WATCH-Einrichtung, die es tatsächlich ermöglicht, den Prozessverlauf in Einzelheiten zu verfolgen.

Obwohl die Leser zum Verständnis dieses Textes keine besonderen Vorkenntnisse brauchen, stellt dies keine systematische Einleitung zum Programmieren in Logo dar (vgl. dazu z.B. I,Lit. [1-4]). Auf diese Weise wird hier, zur Vermeidung unnötiger Wiederholungen, auf den Artikel «3D-Grafik in Logo» (M+K 87-2) verwiesen.

Rekursion

In Programmiersprachen versteht man unter Rekursion den expliziten Aufruf einer Prozedur ihrer selbst. Die Details dieser Definition werden in den nächsten Abschnitten eingehender erörtert. Hier wollen wir nun erwähnen, dass man den tieferen Sinn der Rekursion in der Programmierpraxis oft missversteht und die Frage stellt «warum rekursives Programmieren?».

Als eine erste Anwort auf diese Frage, hier der Standpunkt von D.R. Hofstadter [4]. Er meint, dass entsprechend komplizierte rekursive Systeme vielleicht als definierende Eigenschaften der Intelligenz selbst betrachtet werden könnten. Wenn dies auch extrem anmuten kann, wollen wir es nicht vergessen.

Nun zu unserem Ausgangspunkt. Logo wurde speziell im Hinblick auf Ausbildung entwickelt. Dazu benötigt man insbesondere saubere Definitionen. Wir beginnen nun schrittweise in dieses Gebiet vorzudringen.

Prozeduren und Prozesse

Betrachten wir vorerst zwei viel zitierte Prozeduren. NUM1 und NUM2 (Listing 1), die natürliche Zahlen erzeugen. Wenn wir die anfangs gegebene Definition über den Prozedur-Selbstaufruf benützen, so sind beide Prozeduren rekursiv.

Betrachtet man beide Prozeduren jedoch in ihrem Ablauf, so sind sie verschieden. NUM1 gibt gleich die Zahlenfolge N,...,3,2,1 zeilenweise auf den Bildschirm aus und funktioniert praktisch für jeden beliebigen Wert von N (es können also etliche Bildschirmseiten mit Zahlen beschrieben werden). Die Prozedur NUM2 gibt die Zahlenfolge 1,2,3,...,N aus, aber man muss vorher eine Zeit lang warten. Bei N gleich 200 funktioniert die Prozedur noch, bei N gleich 300 kommt schon die Meldung «Out of LOGO stack space». Mit anderen Worten benötigt NUM1 praktisch keinen Speicherplatz im Gegensatz zu NUM2. Prozeduren der Art wie NUM1 nennt man wegen ihres Ablaufs lineare *iterative* Prozesse, und Prozeduren der Art wie NUM2 heissen lineare *rekursive* Prozesse. Beide Prozeduren sind jedoch syntaktisch betrachtet rekursive Prozeduren. Wir unterscheiden also streng zwischen den Begriffen «Prozedur» und «Prozess».

Funktionen

Eingebaute Logo-Funktionen sind wahrscheinlich jedermann bestens bekannt (z.B. SQRT 3, also Quadratwurzel aus drei, ergibt 1.732051 da Logo über eine sechsstellige Mantisse verfügt). Man kann aber auch beliebige eigene Funktionen definieren, indem man eine geeignete Prozedur schreibt.

Wir können eine solche Prozedur, anstatt in der üblichen Art, auch direkt im Interpreter in der Logo-Listenform (vgl. I,Lit. [7]) eingeben, z.B.:

```
DEFINE «FUNC [[x] [OP :x*:x]]
```

Wenn nachher mit OP «FUNC abgefragt wird, erscheint die Prozedur in ihrer üblichen Art (TO FUNC ... END). Dabei bedeutet OP (OUTPUT) das Ergebnis, welches FUNC liefert. Für FUNC 2 erhält man demnach als Ergebnis 4.

Wenn man unter dem Namen FUNC schnell verschiedene Funktionen ausprobieren möchte, muss man nicht unbedingt den Umweg über den Editor benützen. Zur Aenderung einfach im Interpreter z.B.:

```
.REPLACE 2 TEXT «FUNC
[OP :x*:x*:x + 2*:x - 2]
```

eingeben und das Ergebnis von FUNC 2 ist jetzt natürlich 10. Die Instruktion .REPLACE (Ersetzen) stellt ein Primitivum des Logo-Systems dar.

Betrachten wir nun die Prozeduren NUM3 und NUM4 (Listing1), die ebenfalls Folgen der natürlichen Zahlen erzeugen, jedoch erscheint die Ausgabe jetzt in Listenform. Auf den ersten Blick scheinen diese Prozeduren fast ähnlich, nur ist dies sehr trügerisch. Die Prozedur NUM4 ist eine Funktionsprozedur («OP» vor

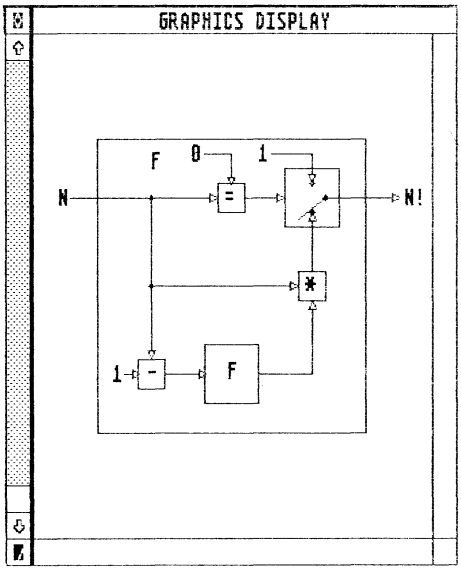


Abb. 1: Der Algorithmus für die Berechnung der Fakultät sowie die entsprechende rekursive Funktionsprozedur in Logo, dargestellt als eine abstrakte Maschine. Da der Logo-Interpreter auch als eine abstrakte Maschine gedeutet werden kann, lässt sich sagen: Der Logo-Evaluator emuliert die Prozedur. Dies nennt man metalinguistische Abstraktion [3].

dem END beachten), und daraus resultiert ein linearer rekursiver Prozess. Dagegen ist NUM3 eine sogenannte «last line» (11) -Rekursion, die einen linearen iterativen Prozess als Folge hat. Anmerkung: In den meisten Lisp-Dialekten verwendet man den Ausdruck «tail recursion» anstatt der erwähnten Namen für die endrekursive Prozedur.

Nun wollen wir uns das eigentliche Werkzeug zum prozess-orientierten Programmieren näher anschauen, das bei Logo besonders benutzerfreundlich ist. Wir geben einfach TRACE und WATCH ein, lassen die Prozeduren laufen und neben dem Bildschirm schalten wir auch den Drucker ein (COPYON).

Zum Vergleich (Trace 1,2) einige Kommentare. Da beide Prozeduren syntaktisch rekursiv sind, evaluiert (interpretiert) Logo beide Prozeduren bei jedem Durchlauf. Bei NUM3 wird in jedem Schritt auch das Teilergebnis ermittelt. Bei NUM4 dagegen werden die Zwischenresultate im Stack gespeichert und Logo sinkt im Stack um eine Stufe tiefer (Zeilennumerierung beachten). Bei NUM3 wird das letzte Ergebnis einfach als Endresultat ausgegeben. Bei NUM4 muss Logo zuerst alle gespeicherten Ergebnisse aus dem Stack holen und kann erst dann zum Endresultat gelangen. Wir können nun vorläufig feststellen, dass sich die beiden untersuchten Prozeduren als Prozesse grundsätzlich verschieden verhalten, obwohl sie «fast gleich» aussehen und selbstverständlich auch das gleiche Resultat liefern.

Literatur

[1] *N. Wirth,* Systematisches Programmieren, Teubner, Stuttgart, 1983

[2] *N. Wirth,* Algorithmen und Datenstrukturen, Teubner, Stuttgart, 1975

[3] *H. Abelson, G.J. Sussman, J. Sussman,* Structure and Interpretation of Computer Programs, MIT-Press, Cambridge, 1985

[4] *D.R. Hofstadter,* Gödel, Escher, Bach, Penguin Books, Harmondsworth, 1980

[5] *B. Allen,* Introducing LOGO, Granada Publ., London, 1984

[6] *E. Stiefel,* Einführung in die numerische Mathematik, Teubner, Stuttgart, 1961

[7] *M.J. Feigenbaum,* Universal Behavior in Nonlinear Systems, Los Alamos Science, Summer, 1980, 4-27

[8] *P. Collet, J-P. Eckmann,* Iterated Maps on the Interval as Dynamical Systems, Birkhäuser, Boston, 1980

[9] *H.-O. Peitgen, P.H. Richter,* The Beauty of Fractals, Springer, Berlin, 1986

Flexible Strukturen

Bevor wir die begonnene Diskussion aus der Logo-Sicht fortsetzen, wollen wir noch einen Abstecher zu Lisp unternehmen. Moderne Lisp-Compiler oder -Interpreter untersuchen aus Effizienz-Gründen vorerst Prozeduren auf ihre Rekursivität als Prozesse und, falls vorhanden, eliminieren sie diese.

So ist z.B. der Interperter von Scheme ein «tail recursion interpreter» [3], d.h. wenn wir z.B. NUM4 (in Lisp geschrieben) eingegeben hätten, würde in Scheme der Analogon von NUM3 automatisch generiert und ausgeführt.

Wie ist dies möglich? Nun, wir kennen ja schon die Eigenschaft des Logo-Interpreters, der aus einer üblichen Logo-Prozedur automatisch auch ihre Listenform generiert. Mit anderen Worten, die intraprozedurale Struktur in Listenform ist bekannt. In Scheme ist man noch weiter gegangen und hat erkannt, dass sich Prozeduraufrufe als «message-passing» allgemein definieren lassen. Daraus hat man weiter gefolgert, dass «tail recursion» nichts anderes als den natürlichen Weg zur Compilation (Uebersetzung) eines Prozeduraufrufs darstellt.

Dies alles gilt natürlich nur bei listenverarbeitenden Programmiersprachen. Die Operationen auf Listen ermöglichen einen sehr flexiblen Strukturwechsel.

Wie solche Programm-Manipulationen technisch realisierbar sind, ist natürlich schwer genau zu beschreiben. Es gibt ja sicherlich sehr viele Details, die man berücksichtigen muss. Wir können uns davon jedoch, mittels des Logo-Systemprimitivums .REPTAIL, eine Vorstellung machen. Einfach im Interpreter eingeben

```
.REPTAIL 1 NEXT «NUM4 [IF = :N 0
[OP :L]][NUM4 :N - 1 SE :N :1]]
```

und die Prozedur NUM4 ist nun endrekursiv wie die Prozedur NUM3. *Dazu eine Warnung für interessierte Leser:* bei solchen Eingaben kann eine falsch gesetzte Klammer zu unvorhersehbaren Systemantworten führen, und dies ohne irgendwelche Fehlermeldungen.

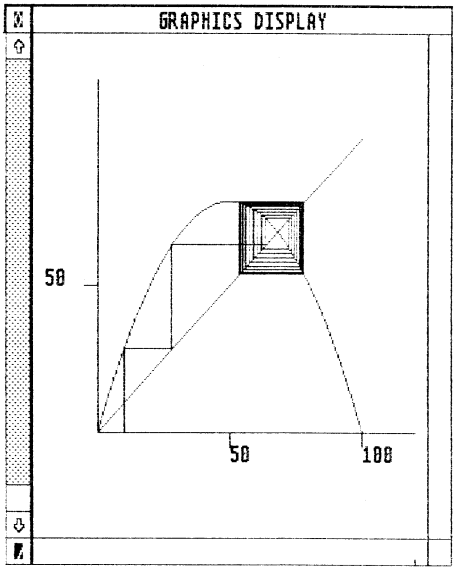


Abb. 2: Für P = 7.0 wird der stabile Fixpunkt x(n) = 64.37... der Parabel iterativ immer besser angenähert.

Primitiva und Vokabeln

Neben den Logo-Systemprimitiva kennen wir auch eine grössere Anzahl von Logo-Primitiva. Im Hinblick auf unser weiteres Vorhaben müssen wir auch hier Klarheit gewinnen.

Betrachten wir das Logo-Primitivum REPEAT und versuchen wir es mit der Prozedur RPT (Listing 2) nachzuahmen. Mit :COUNT ist die Anzahl von Wiederholungen gemeint, diese verringert sich natürlich bei jedem Durchgang um eins (:COUNT - 1). Mit :SEQUENCE ist die beliebige Instruktionsliste bezeichnet, die wir eben wiederholt laufen lassen wollen (mit RUN). Mit der Abbruchbedingung (STOP) wird auch der Fehler mit Minus verhindert, d.h. bei -n reagiert Logo gar nicht und bei - n erscheint die Meldung «doesn't like - n as input». Beim REPEAT gibt es in beiden Fällen eine Fehlermeldung. Wir können uns mit der Nachahmung jedoch zufrieden geben, da ja der Fehler verhindert wird. Wenn wir jetzt z.B.

```
RPT 40/8.5 [FD 50 RT 90]
```

eingeben, wird erstaunlicherweise gleich wie beim REPEAT, doch ein Quadrat gezeichnet.

Die Prozedur RPT kann man auch als Logo-Vokabel bezeichnen, damit ist gemeint, dass man zur Konstruktion einer Logo-Anweisung auch Vokabeln anstatt Primitiva verwenden kann. Wie sich dies auf Prozesse aus-

wirkt, werden wir etwas später untersuchen.

Es ist nun interessant zu bemerken, dass sich Logo mit der kleinsten Anzahl von Primitiva wesentlich von anderen Lisp-Dialekten absondert. Es wurde bisher auch kaum versucht, dies mit Logo-Vokabeln zu kompensieren. Dagegen hat man schon sehr früh begonnen, Pascal-Anweisungen mittels Logo-Vokabeln zu formulieren.

Um ein Beispiel zu geben, machen wir die folgende Vereinbarung. Wir schreiben wie bis jetzt Logo-Primitiva oder -Vokabeln mit Grossbuchstaben. Aus Pascal stammende Ausdrücke werden mit unterstrichenen Kleinbuchstaben geschrieben. Logo-Vokabeln als Pascal-Nachahmungen werden jedoch besonders gekennzeichnet, zum Beispiel: REPEAT unterscheidet sich von repeat, aber REPEAT1 ist eine Nachahmung von repeat ... until.

Betrachten wir jetzt die Prozeduren WHILE1 (I,Lit. [1]) und WHILE2 [5], die die Pascal-Anweisung while ... do nachahmen. Sie sind analog zur RPT geschrieben worden. In WHILE1 ist der rekursive Aufruf durch LABEL «BEGIN ... GO «BEGIN ersetzt worden. Die selbsterklärende Art von Pascal mit z.B. begin, do usw. ist in Logo nicht notwendig (WHILE1). Wenn man dies behalten will, kann es leicht mit «dummy» (scheinbaren) Funktionsprozeduren (DO2) realisiert werden (Listing 2).

Im nächsten Abschnitt wird die Uebernahme von fremden Sprachelementen ins Logo hinsichtlich des prozess-orientierten Programmierens

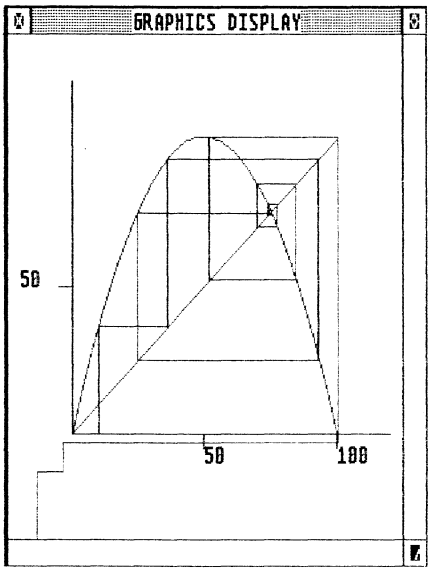


Abb. 3: Für P = 1.01 ist der Fixpunkt der Parabel instabil und kann nicht iterativ erreicht werden.

getestet. Unabhängig von diesem Test kann jedoch zum Ausdruck gebracht werden, dass verschiedenste Nachahmungen in Logo ohne weiteres durchführbar sind. Damit erhält Logo als ausbildende Programmiersprache einen hohen Stellenwert. Logo ist eine offene Programmiersprache, der Uebergang zu anderen hohen Programmiersprachen kann fast fliessend gestaltet werden, es existieren keine hindernden «Regeln» für Programmierer. Ob Lisp (zum Teil direkt übersetzbar in Logo) oder Pascal oder eine andere Sprache, man kann aus Logo mit minimalem Aufwand überall hin, vorausgesetzt man hat dafür einen Grund oder will es einfach.

Algorithmen

Algorithmen bzw. rekursive Funktionen sind Begriffe aus der jüngeren mathematischen Logik, die wir hier nicht behandeln können. Wir wollen nur an einem konkreten und gut bekannten Beispiel zeigen, worum es sich handelt.

Die Fakultät einer natürlichen Zahl dient uns als Beispiel für eine rekursive Funktion. Der Algorithmus lautet: fakt(0) = 1, wenn n > 0, dann fakt(n) = n * fakt(n-1).

Dieser Algorithmus lässt sich z.B. durch eine abstrakte Maschine darstellen (Abb. 1). Man kann den gleichen Algorithmus auch mittels verschiedener Logo-Prozeduren darstellen (Listing 3).

Dabei ist es nur bei der Funktionsprozedur FAKT1 offensichtlich, dass der Algorithmus, die abstrakte Maschine, sowie die Prozedur dasselbe Objekt, nur in anderer Symbolik, darstellen.

Damit sind wir zum Punkt angelangt, der uns veranschaulicht, dass das Programmieren mit rekursiven Funktionen dem menschlichen Verstand am nächsten liegt. Wir können uns hier auch an die Meinung von Hofstadter [4] erinnern, etwas stimmt doch daran. Die Wandlung der Prozeduren, um aus rekursiven iterative Prozesse zu erhalten, ist eigentlich nur durch unsere Computer bedingt, die bekanntlich keine rekursiven Maschinen sind. Deswegen benötigen wir auch prozess-orientiertes Programmieren. Darum stellen auch die neuen Lisp-Interpreter bzw. -Compiler einen wahren Fortschritt dar, da sie dem Benutzer erlauben so zu programmieren, wie es ihm Nahe liegt, und das rein Technische erledigt dann ein Programm. Damit soll aber nicht gesagt werden, dass wir auf das

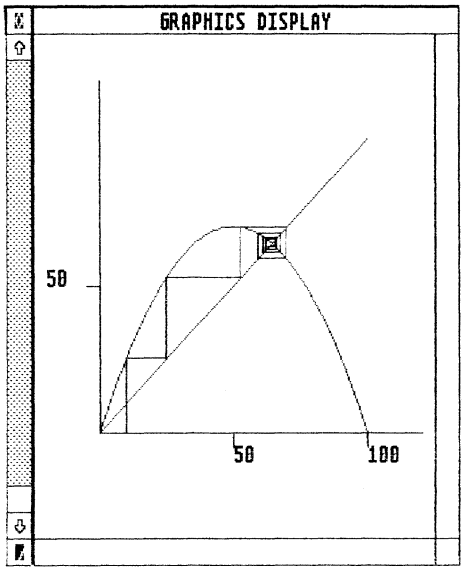


Abb. 4: Für P = 0.785 ist der Fixpunkt der Parabel zwar instabil, jedoch exisitiert ringsherum ein Grenzzyklus.

prozess-orientierte Programmieren verzichten können, denn jemand muss schliesslich die Compiler bauen und die anderen wollen es auch lernen.

Es gibt selbstverständlich auch Probleme, für die wir keine rekursiven Lösungen haben, aber da stehen wir auch vor keiner solchen Alternative.

Vergleichen wir nun verschiedene Prozeduren für die Falkultät (Listing 3) in ihrem Ablauf:

FAKT1. Den Ablauf der Funktionsprozedur kennen wir schon als linearen rekursiven Prozess (NUM4, Trace 2), wobei Stack für die Zwischenergebnisse benötigt wird.

FAKT2. Durch Anwendung von REPEAT erzeugen wir einen linearen iterativen Prozess. Logo evaluiert nur einmal zu Beginn, da ja die Prozedur selbst syntaktisch nicht rekursiv ist. Stack wird praktisch überhaupt nicht belastet, da Logo nur in der ersten Stufe rechnet (Trace 3). Aus der Prozedur erkennen wir aber nicht gleich, um welche Art von Algorithmus es sich handelt; dazu bedarf es schon gewisser Ueberlegung. Nur das Endresultat stimmt, also doch ein Trick, allerdings ein sehr gut gelungener.

FAKT3. Die in einigen Lisp-Dialekten implementierte Funktion PROG direkt ins Logo übersetzt. Dies ist zwar ein linearer, iterativer Prozess und Logo evaluiert nur einmal zu Beginn (findet dabei auch LABEL «BEGIN), aber muss dann bei jedem Durchlauf feststellen GO «BEGIN. Der Prozess wird nur unnötigerweise verlängert (Trace

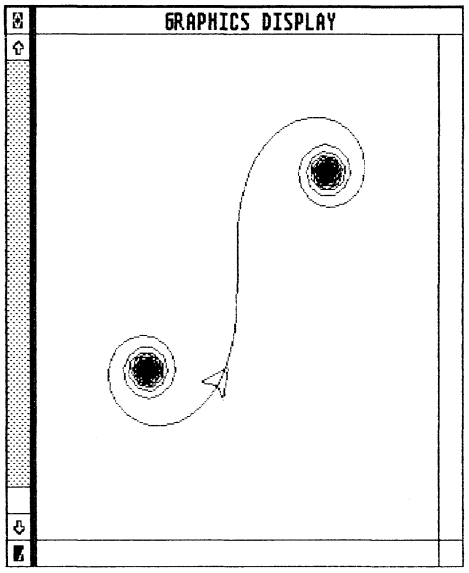


Abb. 5: Der rekursive Logo-Oszillator (Prozedur INSPI).

4). Damit soll festgehalten werden, dass mit GO keine sinnvolle Wandlung rekursiver in iterative Prozesse erzielbar ist, obwohl dies in einigen Logo-Schriften als einzige Möglichkeit erwähnt wird. Es wird aber damit nicht nur seines schlechten Rufs wegen abgelehnt (der Fall in mehreren Lisp-Schriften). Man kann GO auch sinnvoll verwenden, jedoch nicht als Ersatz für Rekursion.

FAKT2A. Ersatz des Logo-Primitivums REPEAT aus FACT2 durch die Logo-Vokabel RPT. Obwohl FAKT2A nur zu Beginn evaluiert wird, muss wie erwartet, immer wieder RPT evaluiert werden. Es ist wohl nicht notwendig zu betonen, dass der Ersatz von Primitiva durch Vokabeln keinen Sinn hat. Das analoge gilt für FAKT4 (WHILE2 und DO2) und FACT5 (WHILE1) betreffend die Evaluation. Hier stimmen auch die Zwischenergebnisse und man muss sich bewusst werden, welches Ziel man erreichen will: einen möglichst kurzen iterativen Prozess um jeden Preis oder einen etwas längeren iterativen Prozess mit richtigen Zwischenergebnissen. (Aus Platzmangel wurde hier kein TRACE gegeben, da sowieso daraus nichts Neues entnommen werden kann.)

Wir schliessen hier unseren Test ab, obwohl aus der ergiebigen Lisp-Literatur weitere Beispiele für die Berechnung der Fakultät leicht zu übersetzen wären. Wir haben aber genügend Material untersucht, um uns mit dem prozess-orientierten Programmieren vertraut zu machen.

Effektivität und Effizienz

Die vorherigen Ausführungen sind den Anpassungen der Prozeduren an die vorhandenen Ressourcen (Hard- und Software) gewidmet gewesen. Mit anderen Worten, wir haben über die Steigerung der Effektivität *(= «Wirkungskraft»)* des Codes gesprochen. Daneben gibt es auch den Begriff der Effizienz *(= «Wirkungsgrad»)*, der auch den zeitlichen Faktor berücksichtigt.

Um unsere Definitionen in zusammenfassender Art zu betrachten, wollen wir einige Vereinbarungen treffen. Bezeichnen wir mit R die Ressourcen, so kann man bezogen auf R eine «Grössenordnung des Wachstums des Prozesses» G definieren. Für einen linearen rekursiven Prozess gilt dann Gs(n) und Gt(n), d.h. der Prozess beansprucht Speicherplatz Gs linear proportional zu der Masseinheit n (in unserem Fall z.B. die Grösse der zu berechnenden Zahl). Das gleiche gilt für die Zeit Gt, sie erhöht sich auch linear. Für einen linearen iterativen Prozess gilt jedoch Gs(1) und Gt(n), der Bedarf für Speicherplatz ist also konstant und die benötigte Zeit wächst linear mit der Grösse der zu berechnenden Zahl.

Betrachten wir nun die ganz einfache Prozedur FI (Listing 4) zur Berechnung der Zahlenfolge nach Fibonacci. Wie erstaunlich es auch sein mag, gilt hier Gs(n) und $Gt(1.618^n)$, d.h. der Zeitbedarf wächst hier exponentiell mit n. Solche Prozesse nennt man *baumrekursiv* und sie sind wegen der redundanten Berechnungsart äusserst ineffizient, haben jedoch bei nichtnumerischen Applikationen eine Bedeutung (z.B. bedienen sich

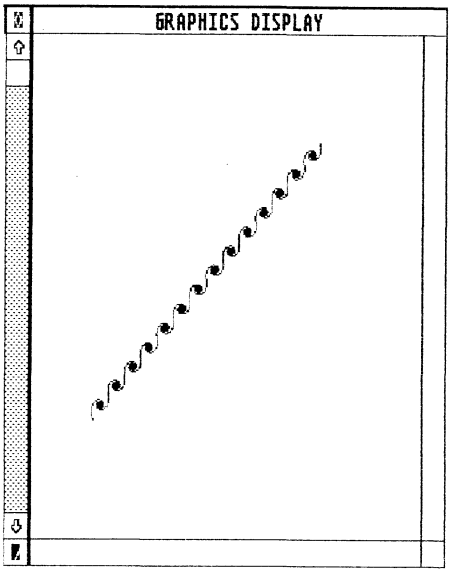


Abb. 6: Für L = 1 oder 5, eine Trajektorie bestehend aus Wirbeln.

```
TO NUM1 :N
IF = :N 0 [STOP]
TYPE :N
NUM1 :N - 1
END

TO NUM2 :N
IF = :N 0 [STOP]
NUM2 :N - 1
TYPE :N
END

TO NUM3 :N :L
IF = :N 0 [OP :L]
NUM3 :N - 1 SE :N :L
END

TO NUM4 :N :L
IF = :N 0 [OP :L]
OP (SE NUM4 :N - 1 :L :N)
END
```

Listing 1

```
TO RPT :COUNT :SEQUENCE
   IF < :COUNT 1 [STOP]
      RUN :SEQUENCE
   RPT :COUNT - 1 :SEQUENCE
END

TO WHILE1 :CONDITION :ACTION
LABEL "BEGIN
IF RUN :CONDITION [RUN :ACTION GO "BEGIN]
END

TO WHILE2 :CONDITION :ACTION
IF NOT (RUN :CONDITION) [STOP]
RUN :ACTION
WHILE2 :CONDITION :ACTION
END

TO DO2 :VAR
OP :VAR
END
```

Listing 2

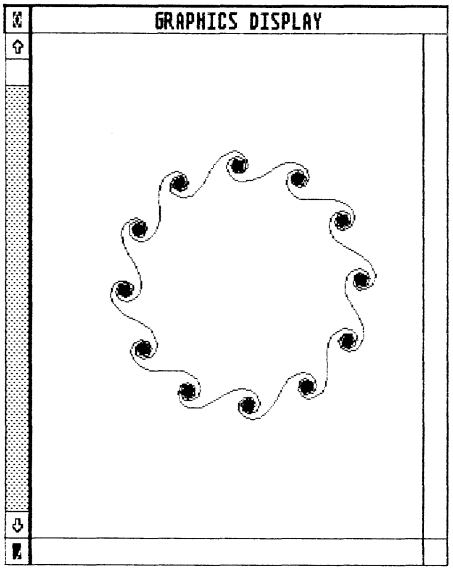


Abb. 7: Für L = 6, ein Grenzzyklus bestehend aus 12 Wirbeln. Das analoge gilt für L = 2 (4), L = 3 (3), L = 4 (8). Anzahl von Wirbeln in Klammern.

Interpreter teilweise gerade solcher Prozesse bei der Evaluierung). Andernfalls leistet hier eine Standardlösung aus Lisp ins Logo übersetzt gute Dienste (FIB und ITER). Dabei gilt nun Gs(1) und Gt(n), d.h. die Wandlung führt zu einem linearen iterativen Prozess. Zur Veranschaulichung dieses Unterschiedes ein Beispiel: bei der Berechnung von FI 10 (das ist nur die Zahl 89) beträgt der Ausdruck von Trace ganze 12 Seiten, im iterativen Prozess knapp eine Seite. Hier hat man beides erreicht, Steigerung sowohl der Effektivität als auch der Effizienz.

Bei der Entwicklung von professioneller Software spielen solche Unterschiede eine aus wirtschaftlichen Gründen wichtige Rolle; so ist man schon froh, wenn, durch Optimierung von Algorithmen, eine Reduktion der Grössenordnung des Wachstums von n auf SQRT n oder von n^2 auf n*log n erreicht wird.

Bei Logo spielt dagegen Wirtschaftlichkeit keine Rolle, jedoch muss man sich der Logo-Ressourcen bewusst bleiben. Logo kann mit seinem Interpreter bei aufwendigen Aufgaben mit keinem Compiler konkurrieren. Wenn wir Logo bei Aufgaben anwenden, für die man mit compilierten Code auf dem gleichen Rechner Stunden benötigt, so können wir höchstens durch die Ineffizienz des Interpreters eine Ineffektivität des Logo-Codes vortäuschen.

Zuletzt wollen wir auch die sogenannte rekursive Turtle Grafik (z.B. Hilbert- oder Sierpinski-Kurven, binäre Bäume usw.) kurz erwähnen. Bekanntlich enthalten entsprechende

```
TO FAKT1 :n
   IF = :n 0 [OP 1]
   OP PRODUCT :n FAKT1 - :n 1
END

TO FAKT2 :n
LOCAL "c MAKE "c 1
   REPEAT - :n 1 [MAKE "n :n * :c
                  MAKE "c :c + 1]
   OP :n
END

TO FAKT3 :n
(LOCAL "x "c)
   MAKE "x 1 MAKE "c 0
LABEL "BEGIN
   IF = :c :n [OP :x]
   MAKE "c :c + 1
   MAKE "x :x * :c
GO "BEGIN
END

TO FAKT4 :n
(LOCAL "x "c)
   MAKE "x 1 MAKE "c 0
     WHILE2 [:c < > :n] DO2
  [MAKE "c :c + 1
   MAKE "x :x * :c]
IF = :c :n [OP :x]
END

TO FAKT5 :N
(LOCAL "X "C)
   MAKE "X 1 MAKE "C 0
      WHILE1 [:C < > :N]
  [MAKE "C :C + 1
   MAKE "X :X * :C]
IF = :C :N [OP :X]
END
```

Listing 3

Logo-Prozeduren mehrfache rekursive Selbstaufrufe. Die generierten Prozesse sind auch linear rekursiv. Mit anderen Worten, entsprechend der eingegebenen Tiefe wird Stack nach unten und nach oben abgelaufen, aber es gibt nichts zum Speichern. Nach jeder Evaluation wird die entsprechende grafische Instruktion ausgeführt und Turtle erwartet in der neuen Position die nächste Instruktion. Zum Unterschied von den anderen kann man solche Prozesse als linear *semirekursiv* bezeichnen.

Iteration und Chaos

In der numerischen Mathematik nennt man bestimmte Näherungsverfahren Iteration [6]. (Die etwas verwirrende Nomenklatur soll zu keinem Missverständnis führen.) Dabei handelt es sich um die Rekursionsformel

$$x(n+1) = f(x(n))$$

Man beginnt mit einem Anfangswert x(0), berechnet das Resultat, setzt dieses als neuen Wert in die For-

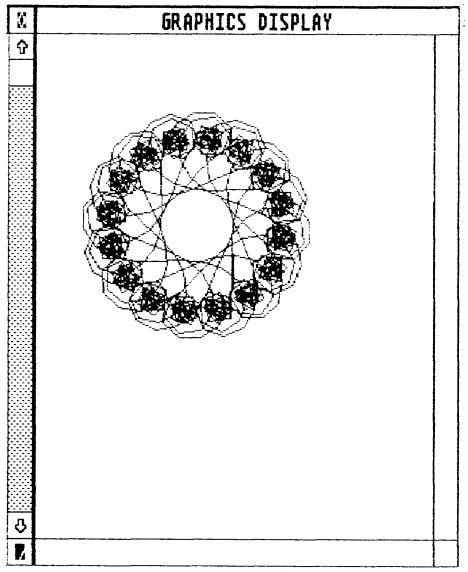


Abb. 8: Für L = 8, ein Ring von Wirbeln, die jetzt mit gegenüberliegenden Wirbeln verbunden sind. Die Trajektorie stellt einen besonderen Grenzzyklus dar. Der Kreis in der Mitte des Ringes ist offensichtlich ein Reppelor-Gebiet, da ihn kein Teil der Trajektorie durchquert.

```
TO DRAW :X :Y
IF > :X 100 [STOP]
MAKE "Y :X
SETPOS SE :X :Y
DRAW :X + 1 :Y
END

TO CURVE :X :Y :P
IF > :X 100 [STOP]
MAKE "Y :P * 4 * :X * (100 - :X) / 100
SETPOS SE :X :Y
CURVE :X + 1 :Y :P
END

TO FEIGEN :P
SETZOOM 2
SETPAN [50 50]
FD 120 BK 120 RT 90 FD 120 BK 20 LT 90 BK 5 FD 5 LT 90 FD 50 RT 90 BK 5 FD 5 PU
 HOME PD FD 50 LT 90 FD 5 PU HOME PD
PU SETPOS [50 -10] PD TT [50]
PU SETPOS [100 -10] PD TT [100]
PU SETPOS [-20 50] PD TT [50] PU HOME PD
DRAW 0 0 PU HOME PD
CURVE 0 0 :P
PU HOME PD
END

TO ITER :I :P
PU SETPOS SE :I 0 PD
ITEREX :I :P
END

TO ITEREX :I :P
MAKE "IN :I
MAKE "I :P * 4 * :IN * (100 - :IN) / 100
SETPOS SE :IN :I SETPOS SE :I :I
ITEREX :I :P
END
```

Listing 5

```
TO FI :N
IF OR :N = 0 :N = 1 [OP 1]
OP (FI :N - 1) + (FI :N - 2)
END

TO FIB :n
   IF :n = 0 [OP 1]
    OP ITER 1 0 :n
END

TO ITER :x :y :n
   IF :n = 0 [OP :x]
   ITER (:x + :y) :x (:n - 1)
END
```

Listing 4

mel ein und wiederholt dieses Verfahren so oft bis die gewünschten Näherung erreicht ist.

Wir wollen uns dies an einem Beispiel ansehen, wobei wir das Ganze mit Grafik veranschaulichen können. In der Prozedur CURVE (Listing 5) ist die Rekursionsformel mit der Parabel

$$x(n+1) = 4*P*x(n)(1-x(n))$$

gegeben, wobei P einen vorläufig willkürlichen, numerischen Parameter darstellt. Um die Iteration grafisch zu simulieren, benützen wir die Gerade

$$x(n+1) = x(n)$$

(DRAW), zeichnen auch das etwas geänderte Koordinatensystem (FEIGEN :P) mit der Parabel und können dann mit der Prozedur ITER :Anfangswert :Parameter unsere Iteration beginnen. Für den Anfangswert x(0) = 10 und für P = 0.7 sehen wir (Abb. 2), wie sich die Näherung stufenweise zum Fixpunkt x(n) = 64.37... bewegt und dort auch verbleibt (da wir keine Abbruchbedingung in der Prozedur haben, müssen wir diesen Prozess stoppen). Gerade solch einen Prozess wünscht man sich in der Numerik und sagt dabei «die Rekursion konvergiert».

Aendert man nun den Parameterwert z.B. auf P = 1.01 (Abb. 3), versucht die Rekursion den Fixpunkt zu erreichen; dieser ist jetzt aber abstossend (instabil), und die «Näherung» entfernt sich wiederum von ihm. Numeriker sagen: «Das können wir nicht gebrauchen, da die Rekursion divergiert». In Logo erscheint auch die Meldung «Turtle out of bounds» obwohl wir uns im WINDOW-Modus befinden.

Verlassen wir nun die Numerik und betrachten wir die Iteration mit den Augen des Physikers Mitchell J. Feigenbaum [7], der 1978 - ausgerüstet nur mit einem programmierbaren Taschenrechner - das Tor zum (mathematischen) Chaos [8] entdeckt hat.

Um weiterzukommen ist es sinnvoll, unsere Nomenklatur etwas zu ändern. Zu diesem Zweck bedienen wir uns der mathematischen Theorie der dynamischen Systeme. (In englischen Fachkreisen nennt man dies «dynamical systems«, was Sprachspezialisten zur Entrüstung bringt). So sagen wir: In Abb. 2 ist der Fixpunkt stabil, oder es ist ein Attraktor und in Abb. 3 ist der Fixpunkt instabil, oder es ist ein Reppelor.

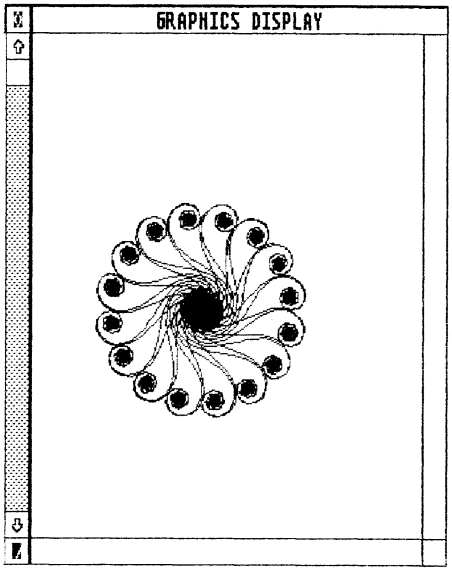


Abb. 9: Für L = 3.2 (ohne Nullen dahinter, da sie Logo nicht akzeptiert), wiederum ein Ring von Wirbeln, die aber jetzt mit Wirbeln in der Mitte des Ringes verbunden sind. Dieses Gebiet erscheint wegen Ueberschreibung schliesslich als ein einziges Attraktor-Bassin (schwarzes Loch). Die Trajektorie stellt wiederum einen besonderen Grenzzyklus dar.

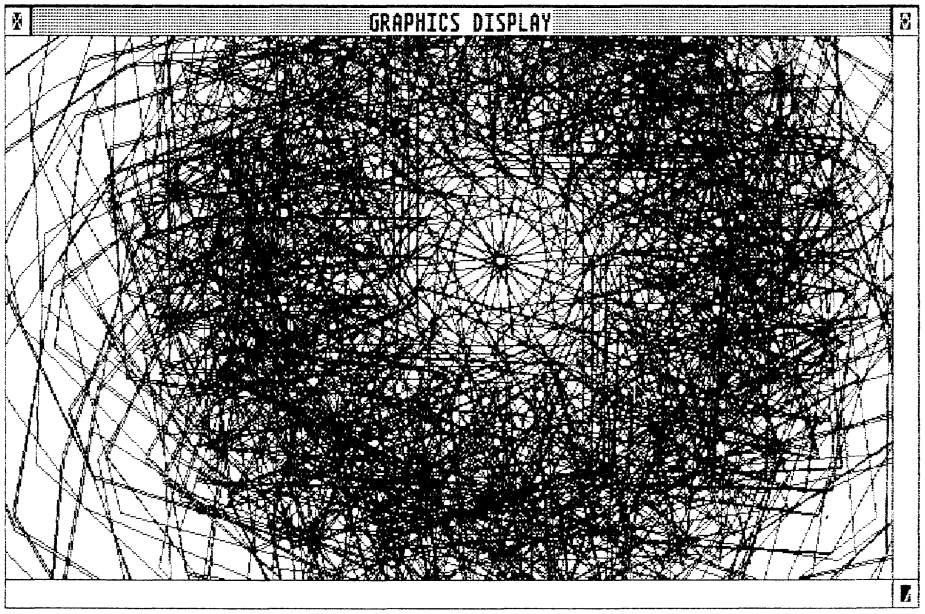


Abb. 10: Das «schware Loch» aus Abb. 9, dargestellt mit Zoom 50 (Eingabe nur via Menü Settings möglich). Eine gewisse Symmetrie bleibt erhalten, jedoch verschwindet die Selbstähnlichkeit zum Logo-Oszillator.

Beides kennen wir schon. Wiederholen wir nun unser Experiment mit P = 0.785 (Abb. 4). Was wir jetzt sehen, nennt man Grenzzyklus (cicle limite, H. Poincaré, 1881). Der Fixpunkt ist für diesen Wert des Parameters ein Reppelor, also instabil, und die äusserste Bahn (Trajektorie in der Phasenebene) ist nun ein Attraktor, d.h. von wo auch sich die Trajektorie nähert, sie verbleibt im Grenzzyklus. Solche vom Parameterwert abhängige Aenderungen der Stabilität nennt man Bifurkation.

Der in Abb. 4 dargestellte Grenzzyklus ist offensichtlich ein stabiler Oszillator. Wenn man statt der Phasenebene die zeitliche Darstellung (die mathematisch komplizierter ist) benützen würde, ergäbe dies zwei Zickzackkurven, die an sin und cos erinnern.

Wie schon erwähnt, kennt man Bifurkationen aus der Theorie der dynamischen Systeme, aber Feigenbaum ist weiter gegangen. Bei detaillierten Berechnungen hat er entdeckt, dass bei selbstähnlichen Wiederholungen (z.B: bei Trajektorien), die charakteristisch für chaotische Muster sind, immer wieder eine Zahl anzutreffen ist. Diese Zahl d = 4.6692016... nennt man heute Feigenbaum'sche Konstante.

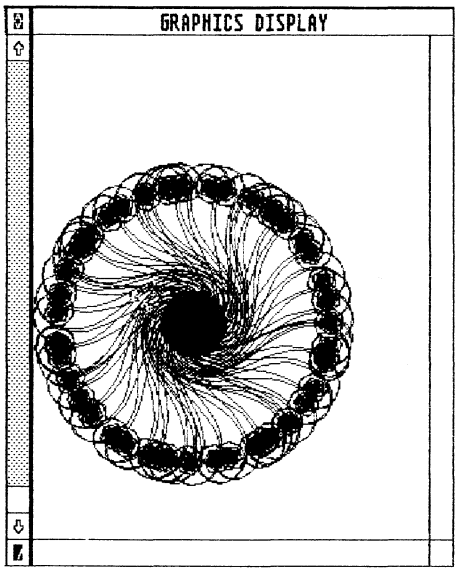


Abb. 11: Für L = 3.200001, Beginn der Instabilität vom Grenzzyklus aus Abb. 9. Das Zeichnen wird mit der Logo-Abbruchbedingung beendet.

Rekursiver Logo-Oszillator

Aus der Turtle-Geometrie (I,lit. [12]) kennt man eine äusserst einfache Prozedur namens INSPI, die, auf dem Bildschirm bei ihrer Ausführung beobachtet, etwas ganz überraschendes leistet (Abb. 5). Was soll diese Prozedur tun? Nichts anderes als z.B. 10 Schritte vorwärts und dann rechts um ein Grad mehr als beim vorherigen Durchlauf; man will also eine Spirale zeichnen.

Mit Trace beobachtet läuft dieser lineare iterative Prozess genau wie es in der Prozedur verlangt wird, aber Turtle benimmt sich anscheinend ganz anders. Beginnen wir mit dem Winkel 0. Turtle folgt wie angeordnet, biegt dann in eine Kreisbahn und wickelt sich in eine Spirale, versucht schliesslich Polygone zu zeichnen und wechselt dann plötzlich, nach dem was wir zu sehen glauben, ihre Drehrichtung. Genau der gleichen Bahn folgend kommt Turtle zurück, erreicht den Ausgangspunkt und wiederholt auf der gegenüberliegenden Seite das Gleiche. Da die Prozedur endrekursiv ist, können wir also über einen rekursiven Logo-Oszillator sprechen.

Es ist hier interessant zu bemerken, dass es in der ganzen Logo-Literatur kaum ein Buch gibt, in dem diese Prozedur nicht in irgendeiner Form erwähnt wird; ausser zwei Ausnahmen, kommentiert jedoch niemand dieses Turtle-Verhalten. Bei den Ausnahmen greift man zur Differentialgeometrie (I,Lit. [1] oder zur komplexen Ebene (I,Lit. [4]), um das Turtle-Verhalten zu erklären. Wir verzichten hier vorläufig auf jegliche Mathematik, da es sich um eine sehr einfache Täuschung des Beobachters handelt.

Wir vereinfachen das Problem zuerst, indem wir den Winkelinkrement auf 90 Grad erhöhen. Jetzt können wir das Ganze auch auf ein Stück Papier zeichnen, also FD 50 RT 0, und Turtle zeichnet eine Strecke wie befohlen. Nächster Schritt: FD 50 RT 90, auch alles normal. Weiter FD 50 RT 180, auch alles normal, ausser dass wir nicht sicher sind, ob Turtle sich nach rechts oder nach links gedreht hat. Entscheidender Schritt FD 50 RT 270, Turtle bewegt sich zwar in umgekehrter Richtung, aber diese Drehung? Kein Mensch (übrigens auch kein Tier) würde anstatt links abzubiegen eine rechte Pirouette drehen, um dasselbe zu erreichen. Da turtle gar nicht gedreht wird, sondern nur gelöscht und frisch gezeichnet (so schnell, dass man es fast nicht sieht), wird der

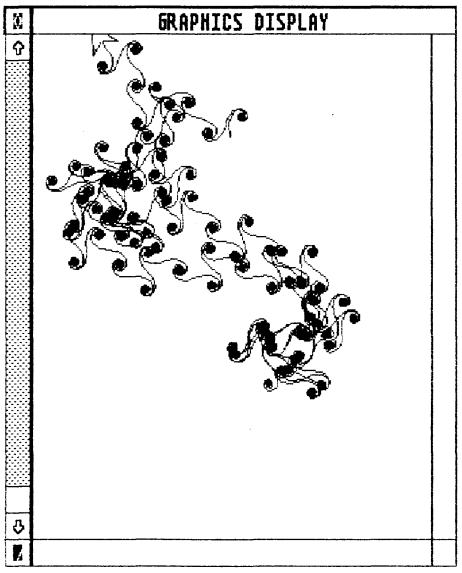


Abb. 12: Für L = 3.205, kein Grenzzyklus mehr, das Zeichnen wird von Logo unterbrochen.

```
?NUM3 6 []
[1] Evaluating NUM3
[1] L is []
[1] N is 6
[1] In NUM3, IF = :N 0 [OP :L]
[1] In NUM3, NUM3 :N - 1 SE :N :L
[2] Evaluating NUM3
[2] L is [6]
[2] N is 5
[1] In NUM3, IF = :N 0 [OP :L]
[1] In NUM3, NUM3 :N - 1 SE :N :L
[2] Evaluating NUM3
[2] L is [5 6]
[2] N is 4
[1] In NUM3, IF = :N 0 [OP :L]
[1] In NUM3, NUM3 :N - 1 SE :N :L
[2] Evaluating NUM3
[2] L is [4 5 6]
[2] N is 3
[1] In NUM3, IF = :N 0 [OP :L]
[1] In NUM3, NUM3 :N - 1 SE :N :L
[2] Evaluating NUM3
[2] L is [3 4 5 6]
[2] N is 2
[1] In NUM3, IF = :N 0 [OP :L]
[1] In NUM3, NUM3 :N - 1 SE :N :L
[2] Evaluating NUM3
[2] L is [2 3 4 5 6]
[2] N is 1
[1] In NUM3, IF = :N 0 [OP :L]
[1] In NUM3, NUM3 :N - 1 SE :N :L
[2] Evaluating NUM3
[2] L is [1 2 3 4 5 6]
[2] N is 0
[1] In NUM3, IF = :N 0 [OP :L]
[1] NUM3 returns [1 2 3 4 5 6]
[1 2 3 4 5 6]
?
```

Trace1

```
NUM4 6 []
[1] Evaluating NUM4
[1] L is []
[1] N is 6
[1] In NUM4, IF = :N 0 [OP :L]
[1] In NUM4, OP ( SE NUM4 :N - 1 :L :N )
[2] Evaluating NUM4
[2] L is []
[2] N is 5
[2] In NUM4, IF = :N 0 [OP :L]
[2] In NUM4, OP ( SE NUM4 :N - 1 :L :N )
[3] Evaluating NUM4
[3] L is []
[3] N is 4
[3] In NUM4, IF = :N 0 [OP :L]
[3] In NUM4, OP ( SE NUM4 :N - 1 :L :N )
[4] Evaluating NUM4
[4] L is []
[4] N is 3
[4] In NUM4, IF = :N 0 [OP :L]
[4] In NUM4, OP ( SE NUM4 :N - 1 :L :N )
[5] Evaluating NUM4
[5] L is []
[5] N is 2
[5] In NUM4, IF = :N 0 [OP :L]
[5] In NUM4, OP ( SE NUM4 :N - 1 :L :N )
[6] Evaluating NUM4
[6] L is []
[6] N is 1
[6] In NUM4, IF = :N 0 [OP :L]
[6] In NUM4, OP ( SE NUM4 :N - 1 :L :N )
[7] Evaluating NUM4
[7] L is []
[7] N is 0
[7] In NUM4, IF = :N 0 [OP :L]
[7] NUM4 returns []
[6] NUM4 returns [1]
[5] NUM4 returns [1 2]
[4] NUM4 returns [1 2 3]
[3] NUM4 returns [1 2 3 4]
[2] NUM4 returns [1 2 3 4 5]
[1] NUM4 returns [1 2 3 4 5 6]
[1 2 3 4 5 6]
?
```

Trace2

```
FAKT2 6
[1] Evaluating FAKT2
[1] n is 6
[1] In FAKT2, LOCAL "c MAKE "c 1 REPEAT - :n 1 [MAKE "n :n * :c MAKE "c :c + 1]
 OP :n
[1] In FAKT2, MAKE "c 1 REPEAT - :n 1 [MAKE "n :n * :c MAKE "c :c + 1] OP :n
[1] Making "c 1
[1] In FAKT2, REPEAT - :n 1 [MAKE "n :n * :c MAKE "c :c + 1] OP :n
[1] Making "n 6
[1] Making "c 2
[1] Making "n 12
[1] Making "c 3
[1] Making "n 36
[1] Making "c 4
[1] Making "n 144
[1] Making "c 5
[1] Making "n 720
[1] Making "c 6
[1] In FAKT2, OP :n
[1] FAKT2 returns 720
720
?
```

Trace3

```
FAKT3 6
[1] Evaluating FAKT3
[1] n is 6
[1] In FAKT3, ( LOCAL "x "c ) MAKE "x 1 MAKE "c 0
[1] In FAKT3, MAKE "x 1 MAKE "c 0
[1] Making "x 1
[1] In FAKT3, MAKE "c 0
[1] Making "c 0
[1] In FAKT3, LABEL "BEGIN IF = :c :n [OP :x] MAKE "c :c + 1 MAKE "x :x * :c
[1] In FAKT3, IF = :c :n [OP :x] MAKE "c :c + 1 MAKE "x :x * :c
[1] In FAKT3, MAKE "c :c + 1 MAKE "x :x * :c
[1] Making "c 1
[1] In FAKT3, MAKE "x :x * :c
[1] Making "x 1
[1] In FAKT3, GO "BEGIN
[1] In FAKT3, IF = :c :n [OP :x] MAKE "c :c + 1 MAKE "x :x * :c
[1] In FAKT3, MAKE "c :c + 1 MAKE "x :x * :c
[1] Making "c 2
[1] In FAKT3, MAKE "x :x * :c
[1] Making "x 2
[1] In FAKT3, GO "BEGIN
[1] In FAKT3, IF = :c :n [OP :x] MAKE "c :c + 1 MAKE "x :x * :c
[1] In FAKT3, MAKE "c :c + 1 MAKE "x :x * :c
[1] Making "c 3
[1] In FAKT3, MAKE "x :x * :c
[1] Making "x 6
[1] In FAKT3, GO "BEGIN
[1] In FAKT3, IF = :c :n [OP :x] MAKE "c :c + 1 MAKE "x :x * :c
[1] In FAKT3, MAKE "c :c + 1 MAKE "x :x * :c
[1] Making "c 4
[1] In FAKT3, MAKE "x :x * :c
[1] Making "x 24
[1] In FAKT3, GO "BEGIN
[1] In FAKT3, IF = :c :n [OP :x] MAKE "c :c + 1 MAKE "x :x * :c
[1] In FAKT3, MAKE "c :c + 1 MAKE "x :x * :c
[1] Making "c 5
[1] In FAKT3, MAKE "x :x * :c
[1] Making "x 120
[1] In FAKT3, GO "BEGIN
[1] In FAKT3, IF = :c :n [OP :x] MAKE "c :c + 1 MAKE "x :x * :c
[1] In FAKT3, MAKE "c :c + 1 MAKE "x :x * :c
[1] Making "c 6
[1] In FAKT3, MAKE "x :x * :c
[1] Making "x 720
[1] In FAKT3, GO "BEGIN
[1] In FAKT3, IF = :c :n [OP :x] MAKE "c :c + 1 MAKE "x :x * :c
[1] FAKT3 returns 720
720
?
```

Trace4

Beobachter nur getäuscht und er urteilt nach seinem eigenen Verhalten bzw. er sieht lieber RT 90 als RT 270 und umgekehrt. Turtle aber folgt nur der Prozedur.

Der letzte S-Oszillator kann fast als das Logo-Pendant zum mathematischen Pendel betrachtet werden. Wir wollen aber den Logo-Oszillator aus Abb. 5 etwas näher betrachten. Der Form nach ist er der Spirale ähnlich, die Klotoide heisst, nur ist diese ganz anders definiert. Die drei Wendepunkte, die er besitzt, sind je 180 Grad voneinander entfernt. Die zwei äusseren verhalten sich abwechselnd als Attraktor bzw. als Reppelor. Da uns ein gegebener Name fehlt, nennen wir sie einfach Wirbel. Schliesslich, für Leser, die Quarzuhren schätzen, seine Frequenz beträgt ca. 19 Millihertz (d.h. für eine vollständige Oszillation oder 2 mal 360 Grad benötigt er einige Sekunden mehr als 50 sec). Als Winkelsumme folgt die Zahl 360 mal 361 oder 129960.

Prozess-Dynamik

Wir wollen nun Bifurkationen des Logo-Oszillators betrachten. Diese können wir wiederum erzeugen, indem wir die Dynamik stören (z.B. durch kleine Aenderungen der Winkelgrösse N oder der Streckenlänge L). Unsere Wahl: anstatt RT :N schreiben wir

RT (:L * :N - 2.5)

also eine lineare Funktion als Rekursionsformel. Als Anfangswert für N nehmen wir Null und für den Inkrement nehmen wir Eins. Unser Bifurkationsparameter ist demnach L.

Zur Definition des Grenzzykluses bedienen wir uns einiger Theoreme aus der Turtle-Geometrie (I,Lit. [12]) und sagen: Wenn auf einer Trajektorie der Turtle-Zustand seine Anfangswerte erreicht, dann stellt diese Trajektorie einen Grenzzyklus dar. Der Turtle-Zustand ist eindeutig durch die Werte POS (Koordinaten) und HEADING (Richtung) definiert.

Bei unserem Experiment müssen wir eine einschränkende Bedingung von Logo berücksichtigen: 32767 Grad ist der maximale Wert eine Drehung, d.h. wir haben eine aus Logo stammende Abbruchbedingung.

Eine Auswahl erstellter Grafiken ist in Abb. 6-12 kommentiert wiedergegeben. Es ist offensichtlich, dass sich dieses Gebiet durch weitere einfache Variationen nicht leicht erschöpfen lässt, also ein grossartiges Spiel. Oder steckt mehr dahinter?

Was haben wir eigentlich getan? Wir haben mit zwei grafischen Instruktionen eine endrekursive Prozedur geschrieben. Durch Evaluation dieser Prozedur durch den Logo-Interpreter ist ein linearer iterativer Prozess gestartet worden, der eine (im Prinzip unendliche) Zahlenfolge generiert. Die grafische Darstellung dieser Prozess-Dynamik offenbart eine Vielzahl von ganz ungewöhnlichen Oszillatoren mit unerwarteten Bifurkationen. Man kann jetzt sagen: Wenn wir aber komplexe Zahlen berücksichtigen, dann...? Nein, hier hören wir auf! Eins muss man jedoch feststellen: Es ist erstaunlich, mit wie wenigen Mitteln man in Logo in noch unerforschte Gebiete vordringen kann.

Hier bleiben viele Fragen offen, da es noch keine Antworten gibt. Man denke nur an die wegen wunderschöner Bilder so populär gewordenen Julia-Mengen bzw. Mandelbrot-Menge [9]. Sind unsere Wirbel den dort bekannten «Seepferdchen» etwa nicht ähnlich? Die Antwort kann nicht aus Experimenten kommen. Wie steht's aber mit den Begriffen «chaotisch» oder «fraktal»? Diese Begriffe sind heute so viel im Gebrauch, dass ihre Definitionen dehnbar geworden sind. So bezeichnen manche Autoren schon einen binären Baum als Fraktal, andere sagen, es ist eine Graphtal und Mathematiker meinen... na, lassen wir es lieber. Wir verzichten deswegen auf weitere terminologische Diskussionen. Bei der Betrachtung der ungewöhnlichen Grafik können auch Fragen entstehen, die weit ausserhalb der hier behandelten Thematik liegen. Wie definiert man dabei z.B. Ordnung/Unordnung bzw. topologische Entropie oder Symmetrie/Asymmetrie bzw. Chiralität usw.? Natürlich keine Antwort.

Schlussbemerkung. Die hier behandelte Dynamik kann richtig nur am Bildschirm beurteilt werden. Sie wird dann vielen Beobachtern noch fremder erscheinen. Dennoch sind Menschen seit eh und je gut vertraut mit solchen Bewegungsvorschriften. Denken Sie nur an die vielen schönen Tänze. □

Nochmals «Schnellere Statistik»

Zu diesem Artikel (M+K 87-2) habe ich einige Bemerkungen anzubringen: Eher eine Selbstverständlichkeit ist die Bildung der Summen von X und X^2 in einer einzigen Schlaufe. Die Berechnung der Varianz lässt sich durch Kürzen noch wesentlich vereinfachen.

Wenn schon von Reduktion der Programmablaufzeit die Rede ist, dann sollte in Zeile 380 im Listing 2 (420 im Listing 1) nicht mit X(I)^2 sondern mit X(I)*X(I) gearbeitet werden. Die benötigte Zeit wird dann zusätzlich um ca. 70 % reduziert, d.h. die Zeit wird dadurch mehr als um den Faktor 3 kürzer!

```
350 SX=0 : SXQ=0
360 FOR I=1 TO N
370         SX=SX+X(I)
380         SXQ=SXQ+X(I)*X(I)
390 NEXT I
400 ! ----------
410 M=SX/N
420 VZ=(SXQ-SX*SX/N)/(N-1)
    oder
420 VZ=(SXQ-M*M*N)/(N-1)
```

Karl Baer, 6313 Menzingen

Universelles EPROM-Programmiergerät für den Commodore C-64

Mit dem hier vorgestellten Programmiergerät ist es möglich, fast alle gängigen EPROMs vom legendären 2704 bis zum 27256 zu programmieren, auszulesen oder den Inhalt mit einem Datenfeld im C-64-Speicher zu vergleichen (Verify). Der Benutzer kann je nach Anforderungen und Geldbeutel verschiedene Ausbaustufen wählen.

Dr. Robert Dorner

Beim Lesen und Programmieren sorgen verschiedene Ueberprüfungsroutinen automatisch für eine maximale Datensicherung. Die Auswahl der EPROM-Type erfolgt über einen Drehschalter. Dies schützt im Vergleich zu anderen Lösungen optimal vor Verwechslungen und so vor Zerstörung eines EPROMs.

Die Hardware

Der Hardwareaufbau des Programmiergerätes ist in Abb. 1 vollständig wiedergegeben. Man erkennt drei verschiedene EPROM-Fassungen, je eine 24-polige für 2704/2708 und 2716/2532/2732 und eine 28-polige für 2764 bis 27256.

Dieses Konzept hat mehrere Vorteile. So kann man durch Weglassen einer Fassung samt dem zugehörigen Schaltungsteil (gestrichelt umrandet) das Gerät individuell zuschneiden, wobei insgesamt der Aufwand der für die Umschaltung nötigen Bauteile minimal ist. Ausserdem ist die Zerstörung eines EPROMs durch eine falsche Einstellung des Wahlschalters weitgehend ausgeschlossen.

Mit dem Schalter S1 wird die EPROM-Type gewählt, die Zuordnung sieht folgendermassen aus:

S1	Type	S1	Type
0	2704	4	2732
1	2708	5	2764
2	2716	6	27128
3	2532	7	27256

Sie gilt für alle gängigen Hersteller, bei unbekannteren vergewissere man sich anhand von Datenblättern und Tab. 1, welche die Funktion der einzelnen Anschlüsse aufzeigt. Die Programmierung von 27C256-EPROMs ist nicht möglich, da diese andere Programmierspannungen benötigen. In allen anderen Fällen können auch die entsprechenden CMOS-Versionen angeschlossen werden.

Welche Type gewählt ist, wird dem Programm im C-64 über den Encoder IC7/IC11, den Analogschalter IC6 und den Portbaustein IC8 mitgeteilt.

Die Baugruppe T3 bis T6 dient der Erzeugung der Programmierspannung von 25V bzw. 21V, die am Emitter von T6 anliegt. Die Umschaltung dieser relativ hohen Spannung erfolgt der Einfachheit halber über ein Relais. Hierfür sollte man eine Ausführung mit vergoldeten Kontakten wählen. Der nötige Vorwiderstand R1 berechnet sich zu:

$$R1 = \frac{40\text{-}U_R}{U_R} R_W$$

U_R = Relaisspannung
R_W = Relaiswiderstand

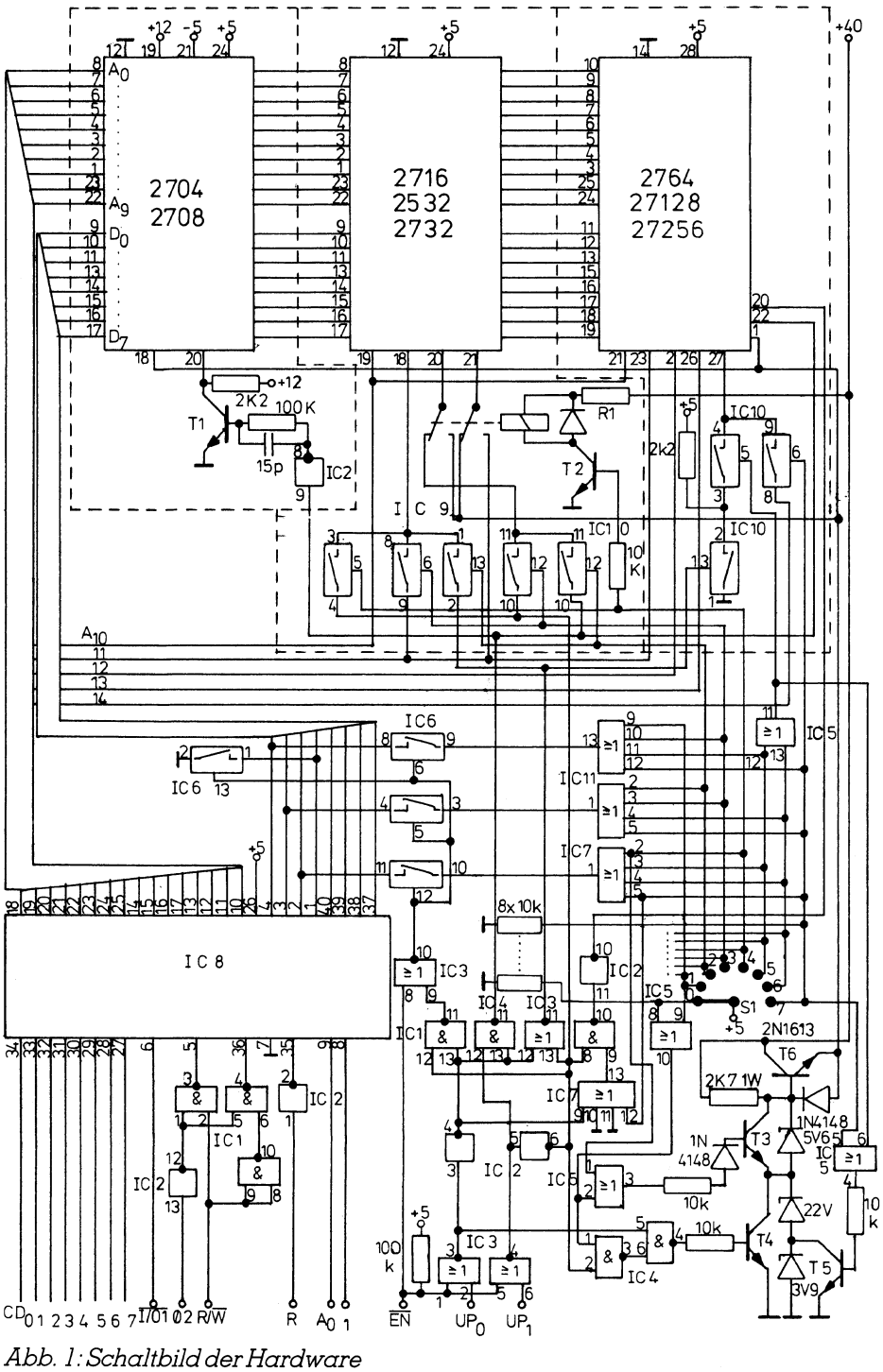


Abb. 1: Schaltbild der Hardware

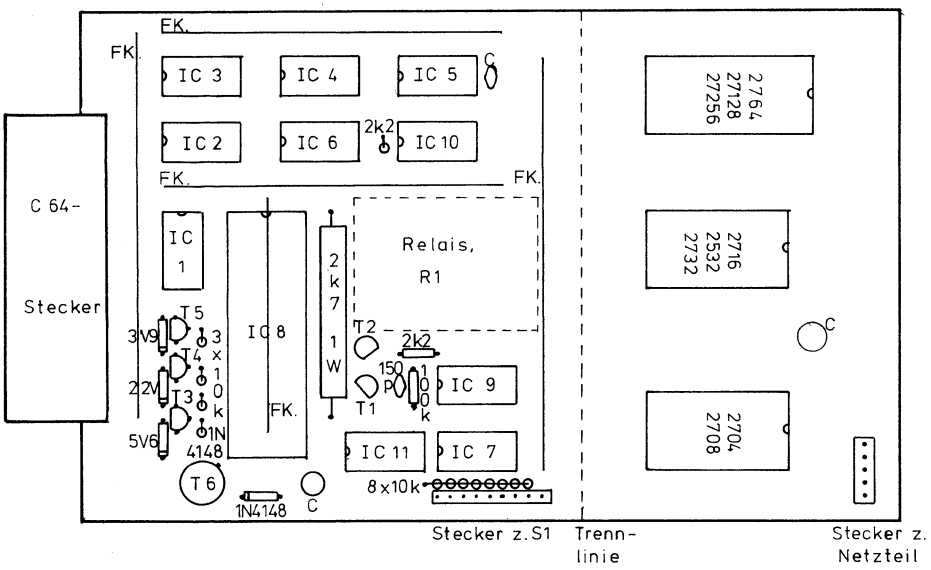


Abb. 2: Anordnung der Bauteile und Fädelkämme (FK.)

T1 - T5	BC 237 B	C	Abblockkond. 1µF Ta
IC1, 4	4011	IC6, 9, 10	4066
IC2	4069	IC7, 11	4072
IC3	4001	IC8	8255
IC5	4071		

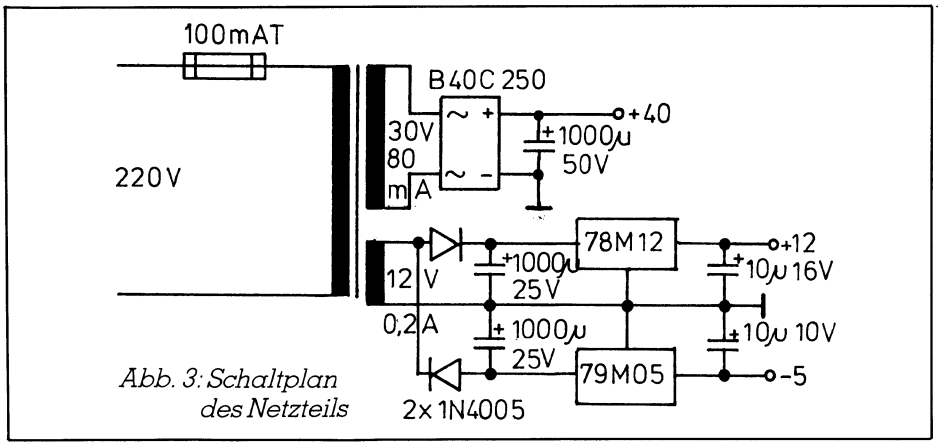


Abb. 3: Schaltplan des Netzteils

Bei der Auswahl des Relais ist unbedingt zu beachten, dass beim Umschalten nie der Arbeits- und Ruhekontakt gleichzeitig schliessen, da sonst die Programmierspannung auf CMOS-Eingänge gelangt und einige Gatter zerstören kann.

Der Port-Baustein IC8 übernimmt die Kommunikation mit dem Prozessorbus. Dieses sehr preiswerte IC muss allerdings mit drei NAND-Gattern aus IC1 und zwei Invertern aus IC2 an die 65XX-Familie angepasst werden.

Die restlichen Gatter und Analogschalter in Abb. 1 dienen zum Erzeugen und Umschalten der höherwertigen Adressen, der Programmierpulse usw. Von aussen gesehen kann die Hardware die folgenden vier Zustände annehmen:

UP0	UP1	Funktion
1	1	Stand-by, Lesen der Schalterposition
1	0	Lesen eines EPROMs
0	1	Anlegen der Programmierspannung
0	0	Programmierimpuls

Die Auswahl des Zustandes erfolgt über die User-Portleitungen UP0 und UP1. Nach dem Einschalten des Rechners wirken diese automatisch als Eingänge, das Programmiergerät ist dann über Pull-up-Widerstände im User-Port im Stand-by-Modus.

Die Enable-Leitung EN setzt das Programmiergerät ebenfalls in den Stand-by-Modus und verhindert Konflikte auf den Portleitungen. Mit diesem Eingang ist es möglich, vollkommen gefahrlos mehrere Geräte am Rechnerbus zu betreiben.

Der mechanische Aufbau

Es wurde hier bewusst auf die Angabe eines Layouts verzichtet, da dies auf jeden Fall doppelseitig wäre und ausserdem häufig Leitungen zwischen zwei IC-Beinen durchgeführt werden müssten. Die Erstellung einer solchen Leiterplatte einschliesslich Fehlerbeseitigung ist im Hobbybereich erfahrungsgemäss sehr arbeitsaufwendig, so dass hier auf die Fädeltechnik verwiesen werden soll.

Eine erprobte Anordnung der ICs und der Fädelkämme ist in Abb. 2 (Ansicht von unten) gezeigt. Der Schaltplan enthält alle Nummern der IC-Kontakte; so kann man durch Nachziehen der bereits gefädelten Leitungen mit einem Filzstift schnell die gesamte Hardware fehlerfrei nachvollziehen.

Empfohlen wird auch eine flexible Flachbandleitungen (ca. 10 bis 20 cm, 25-polig) zwischen der eigentlichen Schaltung und den EPROM-Sockeln (in Abb. 2 gestrichelt gezeichnet). So kann das Wechseln der EPROMs ohne mechanische Rückwirkung und Belastung der Steckanschlüsse am Rechner erfolgen.

Das Netzteil

Will man 2704- und 2708-EPROMs programmieren, benötigt man das vollständige Netzteil nach Abb. 3. Es erzeugt neben der Spannung von +40V für den Programmierimpuls noch +24V und -5V aus einer 12V-Wicklung. Die Stromentnahme ist insgesamt so gering, dass ein 5VA-Trafo ausreicht. Dabei ist ein Relaisstrom von maximal 20 mA zugrunde gelegt. Bei niederohmigeren Relais ist der Trafo, 30V-Gleichrichter und 50V-Elko entsprechend grösser zu dimensionieren.

Die immer nötigen +5V werden dem C-64-Netz entnommen.

Die Software

Das Programm für das Programmiergerät wird absolut nach 8000H geladen und mit «SYS 32780» gestartet. Es meldet sich (Curserposition) mit

.-

Danach kann man die folgenden fünf Befehle geben:

X
Rückkehr zu BASIC

G aaaa
GOTO aaaa

R aaaa,bbbb,cccc
von EPROM-Adresse aaaa bis bbbb Inhalt in C-64-Speicher ab cccc laden

V aaaa,bbbb,cccc
Inhalt ab EPROM-Adresse aaaa bis bbbb mit dem Inhalt im C-64-Speicher ab cccc vergleichen

P aaaa,bbbb,cccc
Inhalt im C-64-Speicher von Adresse aaaa bis bbbb in das EPROM ab Adresse cccc einbrennen.

Wird ein EPROM gelesen oder beschrieben, so wird zuerst die Schalterstellung von S1 abgefragt. Dem Programm sind dann Typ und Speichergrösse des EPROMs bekannt. Vor Beginn einer Aktion wird verglichen, ob der gewünschte Adressraum überhaupt im EPROM zur Verfügung steht. Wenn nicht, erfolgt die Fehlermeldung

.:?

Beim Lesen und Verifizieren wird jedes Byte zweimal gelesen und nur bei Uebereinstimmung der Werte akzeptiert. Bei einem Lesefehler (z.B. bei einem offenen oder defekten Anschlusskontakt) wird die Meldung

FAILURE AT xxxx

mit der Adresse xxxx, an der der Fehler auftrat, ausgegeben. Die weiteren Aktionen am EPROM werden gestoppt. Andernfalls wird beim Lesen bzw. Verifizieren ständig

.: xxxx
bzw.
.: VERIFYING xxxx

ausgegeben, wobei xxxx die laufende EPROM-Adresse ist. So kann der Benutzer immer beobachten, was gerade passiert. Wird der gesamte Bereich fehlerfrei gelesen bzw. verifiziert, erfolgt anstelle der Adresse xxxx die Meldung

OK

Beim Programmieren sind weitere Prüfroutinen eingebaut. Zunächst wird wieder untersucht, ob die EPROM-Grösse ausreicht. Danach erscheint

.: TYPE = eeee Y/N?

Dies ermöglicht eine nochmalige Prüfung der Schalterstellung, da bei Anlegen der Programmierspannung an einen falschen IC-Anschluss das EPROM mit Sicherheit zerstört werden würde.

Pin	2704	2708	2716	2532	2732		Pin	2764	27128	27256
21	-5V	-5V	+25V	+25V	A11	S	23	A11	A11	A11
	-5V	-5V	+25V	+25V	A11	R		A11	A11	A11
	-5V	-5V	+5V	+5V	A11	P		A11	A11	A11
	-5V	-5V	+5V	+5V	A11	PP		A11	A11	A11
20	+12V	+12V	+5V	0V	+25V	S	22	+5V	+5V	+5V
	+12V	+12V	+5V	+5V	+25V	R		+5V	+5V	+5V
	0V	0V	0V	0V	0V	P		0V	0V	0V
	+12V	+12V	+5V	+5V	0V	PP		+5V	+5V	+5V
19	+12V	+12V	A10	A10	A10	S	21	A10	A10	A10
	+12V	+12V	A10	A10	A10	R		A10	A10	A10
	+12V	+12V	A10	A10	A10	P		A10	A10	A10
	+12V	+12V	A10	A10	A10	PP		A10	A10	A10
18	+25V	+25V	+5V	A11	0V	S	20	0V	0V	0V
	0V	0V	0V	A11	+5V	R		0V	0V	+5V
	0V	0V	0V	A11	0V	P		0V	0V	0V
	0V	0V	0V	A11	+5V	PP		+5V	+5V	+5V
						S	27	0V	0V	A14
						R		+5V	+5V	A14
						P		+5V	+5V	A14
						PP		+5V	+5V	A14
						S	1	+21V	+21V	+21V
						R		+21V	+21V	+21V
						P		+5V	+5V	+5V
S = Stand-by						PP		+5V	+5V	+5V
R = Read										
P = Program						X	2	A12	A12	A12
PP = Programmierpuls										
X = unabhängig vom Zustand						X	26	A13	A13	A13

Tab. 1: Belegung der Anschlüsse an den EPROM-Sockeln

N	Quersumme	N	Quersumme	N	Quersumme
0	2358	22	53482	44	105432
1	5290	23	55838	45	107958
2	7346	24	58031	46	110201
3	9417	25	60519	47	111903
4	11075	26	63949	48	114692
5	13042	27	66418	49	117280
6	15205	28	68525	50	119515
7	17634	29	70942	51	121468
8	20131	30	73893	52	124493
9	22457	31	75877	53	126842
10	25177	32	78391	54	129379
11	27560	33	80584	55	131832
12	29840	34	82681	56	133733
13	32278	35	85200	57	135611
14	34181	36	87525	58	137811
15	36712	37	89634	59	139711
16	38827	38	91868	60	141993
17	40948	39	93979	61	143380
18	43846	40	96265	62	145418
19	46506	41	98767	63	147210
20	48788	42	100784	64	150285
21	50943	43	103009	65	153500

Tab. 3: Gesamtsumme der Daten nach insgesamt N Zeilen

Der Benutzer muss nun mit N (no) oder Y (yes) antworten. Im ersten Fall wird zur Eingabe zurückgesprungen, im zweiten erfolgt die Ausgabe von

.ILLEGAL BIT CHECK xxxx

Bei 2704- und 2708-Typen wird hier der gesamte Inhalt auf FF geprüft, bei allen anderen wird verglichen, ob das einzuschreibende Byte keine Einsen enthält, wo bereits Nullen stehen (bekanntlich kann in einem EPROM nur eine Eins zur Null verändert werden). Auf diese Art hat man die Möglichkeit, freigelassene (mit FF belegte Bereiche) nachträglich zu programmieren oder mit etwas Glück kleine Programmänderungen vornehmen zu können, ohne vorher das gesamte EPROM löschen zu müssen.

Nachdem eine bereichsweise Nachprogrammierung bei 2704/2708-Typen verboten ist, wurde hier diese Möglichkeit ausgeschlossen. Diese EPROMs erfordern mehrere Durchgänge (Programmierzyklen) durch den gesamten Speicherbereich mit sehr kurzen Impulsen. Daher wird hier unabhängig von der Eingabe der Anfangs- und Endadresse diese automatisch auf 0000 und 0200 (0400) gesetzt!

Nach erfolgreichem Bit check erscheint

PROGRAMMING xxxx yyyy

Das EPROM wird programmiert und jedes eingeschriebene Byte sofort mit dem aus dem C-64-Speicher verglichen. Im Fehlerfall erscheint wieder

FAILURE AT xxxx

Der Vorgang wird abgebrochen. Bei den Typen 2704/2708 wird zusätzlich bei yyyy die Nummer des Programmierzyklus ausgegeben.

Der Befehl G aaaa dient zum Springen in ein Monitorprogramm, welches eine sinnvolle Ergänzung zu dem hier vorgestellten Programmiergerät ist. Am Sprungziel muss ein vollkommen autonomes Programm stehen. Nach dem G-Befehl kann nicht, wie bei Monitoren üblich, mit BRK oder RTS zurückgekehrt werden.

Da die Monitorprogramme häufig bei C000 angesiedelt sind, wurde das hier beschriebene Programm nach 8000 gelegt. Wer es direkt in einen vorhandenen Monitor einbauen will, muss dort die Eingabeschleife aufbrechen und nach 81D1 springen. Das soeben im Monitor von der Tastatur gelesene Zeichen wird im ACCU übergeben. Ausserdem muss der ab 803C stehende unbedingte Sprung 4C 07 80 so verändert werden, dass er

```
0 DATA186,142,6,2,162,250,154,169,46,162,13,32,195,131,32,207,255,201,13
1 DATA240,242,201,46,240,245,201,32,240,241,201,88,208,9,174,6,2,154,162
2 DATA128,108,0,3,201,71,208,6,32,131,131,108,251,0,76,209,129,169,63,32
3 DATA210,255,76,7,128,234,84,89,80,69,32,61,160,89,47,78,32,191,73,76
4 DATA76,69,71,65,76,32,66,73,84,32,67,72,69,67,203,80,82,79,71,82,65,77
5 DATA77,73,78,71,160,70,65,73,76,85,82,69,160,65,84,160,79,203,86,69,82
6 DATA73,70,89,73,78,71,160,185,64,128,8,41,127,32,210,255,200,40,16,243
7 DATA96,234,234,32,225,131,32,153,131,32,225,131,32,153,131,165,95,197
8 DATA254,144,26,208,6,165,94,197,253,144,18,173,1,2,197,252,144,11,208
9 DATA7,173,0,2,197,251,144,2,24,96,56,96,165,87,133,253,165,88,133,254
10 DATA165,92,133,251,165,93,133,252,96,165,251,141,1,222,165,252,141,2
11 DATA222,96,169,0,141,1,221,120,165,57,240,12,169,232,141,4,221,169,3
12 DATA141,5,221,208,10,169,80,141,4,221,169,195,141,5,221,169,25,141,14
13 DATA221,169,1,44,13,221,240,251,88,169,3,141,1,221,96,32,236,131,160
14 DATA41,32,128,128,160,49,32,128,128,32,169,131,169,0,141,3,221,76,7,128
15 DATA32,169,131,56,165,211,233,4,133,211,96,169,243,141,3,221,169,3,141
16 DATA1,221,32,131,131,32,250,131,32,131,131,32,153,131,32,131,131,32,250
17 DATA131,160,58,32,236,131,169,144,141,3,222,173,0,222,41,15,73,0,170
18 DATA189,192,132,133,253,189,208,132,133,254,165,95,197,252,144,22,208
19 DATA6,165,94,197,251,144,14,165,254,197,95,144,8,208,9,165,253,197,94
20 DATA176,3,76,55,128,165,57,240,8,234,234,234,160,54,32,128,128,169,1
21 DATA141,1,221,32,207,128,32,37,129,160,0,165,57,208,5,173,0,222,145,92
22 DATA177,92,205,0,222,240,3,76,13,129,165,95,197,252,208,9,165,94,197
23 DATA251,208,3,76,51,131,32,225,131,32,250,131,32,225,131,32,250,131,24
24 DATA144,199,162,0,134,57,134,58,201,82,208,3,76,48,129,201,80,208,3,76
25 DATA0,130,201,86,240,3,76,55,128,169,255,133,57,76,48,129,234,234,234
26 DATA234,234,234,234,234,234,234,234,234,234,169,243,141,3,221,169,3,141
27 DATA1,221,169,144,141,3,222,32,131,131,165,252,133,88,165,251,133,87
28 DATA32,131,131,32,153,131,32,131,131,32,250,131,160,58,32,236,131,173
29 DATA0,222,41,15,73,0,170,201,2,176,4,169,255,133,57,189,192,132,133,253
30 DATA189,208,132,133,254,189,224,132,133,251,189,240,132,133,252,160,0
31 DATA32,128,128,32,8,132,32,217,131,160,7,32,128,128,165,95,197,88,144
32 DATA50,208,6,165,94,197,87,144,42,165,254,197,93,144,36,208,6,165,253
33 DATA197,92,144,28,165,57,240,27,165,92,208,20,165,93,208,16,24,165,87
34 DATA101,253,133,94,165,88,101,254,133,95,24,144,3,76,55,128,160,58,32
35 DATA236,131,32,87,241,201,89,240,7,201,32,240,245,76,29,129,169,1,133
36 DATA211,160,12,32,128,128,169,1,141,1,221,165,253,141,0,2,165,254,141
37 DATA1,2,32,190,128,32,207,128,32,37,129,173,0,222,160,0,81,253,49,253
38 DATA240,3,76,13,129,32,144,128,144,231,169,3,141,1,221,32,236,131,160
39 DATA29,32,128,128,169,112,133,58,32,190,128,169,2,141,1,221,169,128,141
40 DATA3,222,32,207,128,32,37,129,160,0,177,253,141,0,222,32,218,128,165
41 DATA57,208,23,169,144,141,3,222,169,1,141,1,221,32,207,128,177,253,205
42 DATA0,222,240,3,76,13,129,32,144,128,144,198,165,57,208,14,160,52,32
43 DATA128,128,32,214,131,32,214,131,76,29,129,165,211,72,169,20,133,211
44 DATA169,0,133,252,165,58,133,251,32,37,129,104,133,211,198,58,208,152
45 DATA160,58,32,236,131,169,0,133,251,133,252,173,0,2,133,253,133,94,173
46 DATA1,2,133,254,133,95,165,87,133,92,165,88,133,93,169,144,141,3,222
47 DATA76,139,129,32,22,132,32,37,132,144,13,72,32,44,132,144,6,133,251
48 DATA104,133,252,96,104,96,162,2,181,250,72,181,252,149,250,104,149,252
49 DATA202,208,243,96,162,1,181,250,72,181,251,32,180,131,104,72,74,74,74
50 DATA74,32,204,131,170,104,41,15,32,204,131,72,138,32,210,255,104,76,210
51 DATA255,24,105,246,144,2,105,6,105,58,96,32,217,131,169,32,44,169,13
52 DATA76,210,255,230,251,208,6,230,252,208,2,230,164,96,152,72,32,220,131
53 DATA104,162,46,32,195,131,76,214,131,162,3,180,251,181,92,148,92,149
54 DATA251,202,16,245,96,32,81,132,234,189,0,1,32,210,255,202,16,247,96
55 DATA165,208,240,7,32,207,255,201,13,208,41,169,32,24,96,32,157,132,234
56 DATA234,176,7,32,157,132,234,234,144,25,32,181,132,10,10,10,10,141,0
57 DATA1,32,157,132,234,234,144,223,32,181,132,13,0,1,56,96,32,22,132,24
58 DATA96,169,0,162,2,157,10,2,202,16,250,162,15,6,251,38,252,248,160,2
59 DATA185,10,2,121,10,2,153,10,2,136,16,244,202,16,234,216,162,5,32,142
60 DATA132,9,48,157,0,1,202,16,245,162,4,189,0,1,201,48,208,3,202,208,246
61 DATA96,169,16,14,12,2,46,11,2,46,10,2,42,144,244,96,32,22,132,201,48
62 DATA144,13,201,71,176,9,201,58,144,7,201,65,176,3,96,24,96,56,96,201
63 DATA58,8,41,15,40,144,2,105,8,96,0,0,0,0,0,0,0,0,255,255,255,255,255
64 DATA255,255,255,2,4,8,16,16,32,64,128,255,255,255,255,255,255,255,255
65 DATA144,148,156,228,172,204,248,120,255,255,255,0,255,255,255,255,10
66 DATA10,10,9,10,10,105,106,255,255,255,255,255,255,255,255
100 S=0
110 FORI=32768TO34047:READA:POKEI,A:S=S+A:NEXT
120 IFS=155800THENSYS32768
130 PRINT"DATENFEHLER!"
```

Tab. 2: Programmlisting

auf den Anfang der Eingabeschleife des Monitors zeigt.

Bei dieser Aenderung ist zu beachten, dass der Monitor nicht bereits vorher auf die Buchstaben R, V oder P reagiert. In diesem Fall müssen die Abfragen darauf mit anderen Buchstaben belegt werden. Die Buchstaben X und G werden bei der hier beschriebenen Aenderung im Programmierer nicht mehr abgefragt. Erfahrungsgemäss lohnt sich dieser Aufwand, da man so seinen Rechner in einen kompletten Entwicklungsplatz für Maschinenprogramme ausbauen kann.

In Tab. 2 ist das gesamte Programm gelistet, Tab. 3 zeigt die Quersummen bis einschliesslich Zeile N, womit Eingabefehler lokalisiert werden können. Dazu gibt man zuerst die Zeilen 100 bis 130 ein. Danach kann das Programm nach jeder neu eingegebenen Datazeile laufen gelassen werden, es meldet sich mit «OUT OF DATA ERROR IN 110«. Mit «?S» wird der Wert der bisherigen Zeilensumme abgefragt. □

COMPUTER MARKT

DIE AKTUELLE COMPUTERINFORMATION 3/87

Einen Drucker an mehrere Computer anschliessen Seite 10

ATI EGA Wonder Seite 13

Digital Research bringt verbesserte Grafiksoftware Seite 18

PC-Software kurz vorgestellt (10) Seite 21

3COM kündigt Unterstützung von Token-Ring an Seite 33

Elektronische Drucker-Umschalter mit Speicher Seite 48

Neue Logitech-Softwarepakete für Modula-2 Seite 54

Microsoft mit neuer Software für die nächste PC-Generation Seite 57

IBM bleibt kompatibel zu IBM

Mit der Vorstellung einer völlig neuen Personal Computer-Familie, dem IBM Personal System/2, hat die IBM wiederum für Wirbel im PC-Markt gesorgt. Das IBM Personal System/2 ist von Grund auf neu konzipiert, mit einer zukunftsweisenden Architektur, mit modernsten Technologien und umfassenden Integrationsmöglichkeiten. Als Betriebssysteme stehen PC-DOS 3.3 sowie das neue IBM Operating System/2 zur Verfügung. Und, das Personal System/2 ist mit den bisherigen IBM Personal Computern voll kompatibel.

Die Modelle 50, 60 und 80 des IBM Personal Systems/2 zeichnen sich durch eine völlig neuentwickelte «Micro Channel» Architektur aus. Diese beschleunigt nicht nur den internen Datentransfer, sie bildet auch - zusammen mit dem Betriebsprogramm IBM Operating System/2 - die Grund

lage für zukünftige
wie «Multitasking
gramming». D
kleinsten M
Bus Ar
en

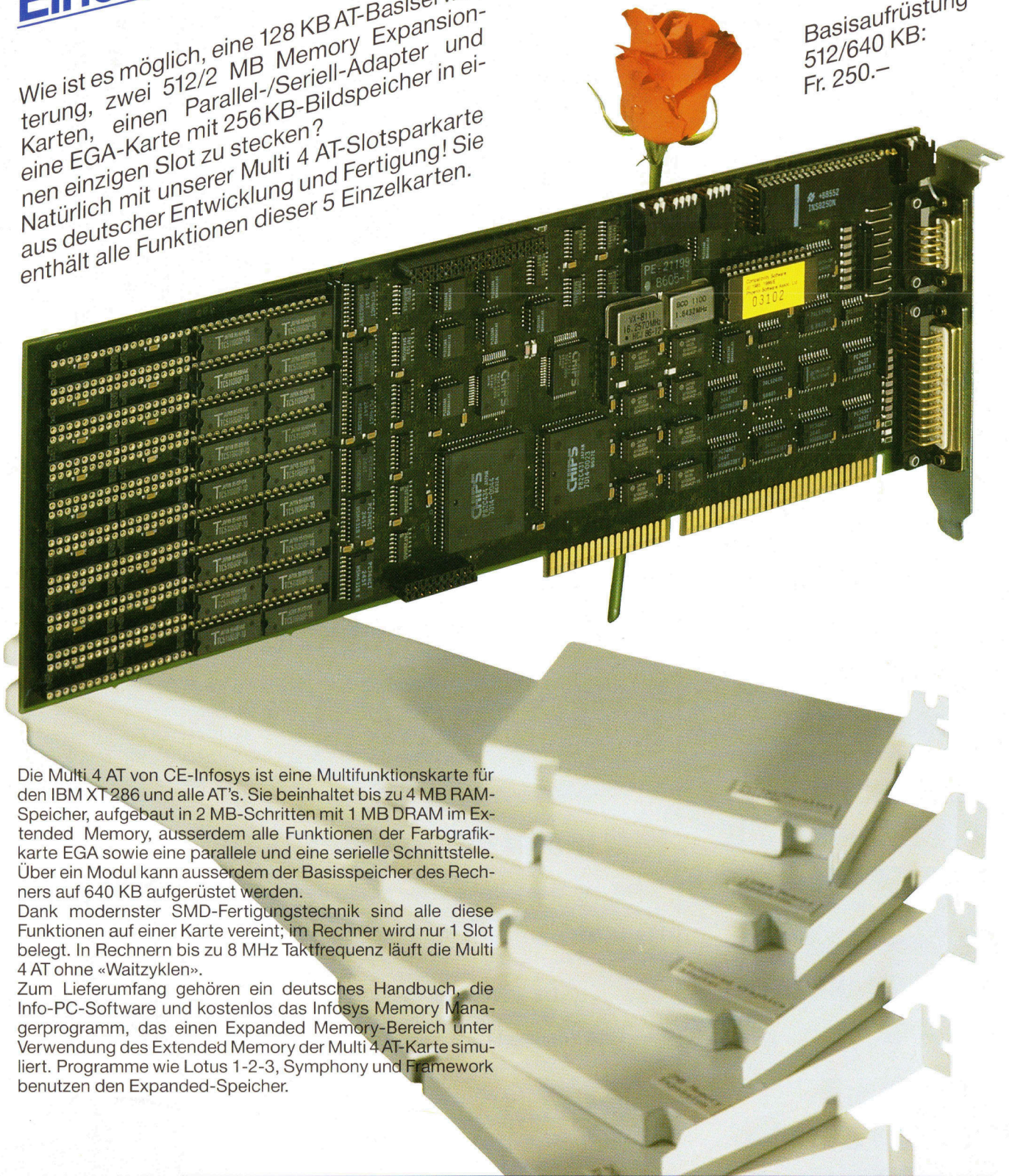
Infosys

Eine für alle-Multi 4 AT, die Slotsparkarte.

Sensationeller Preis von nur Fr. **2'675.–**, inkl. 2 MB Memory.

Basisaufrüstung 512/640 KB: Fr. 250.–

Wie ist es möglich, eine 128 KB AT-Basiserweiterung, zwei 512/2 MB Memory Expansion-Karten, einen Parallel-/Seriell-Adapter und eine EGA-Karte mit 256 KB-Bildspeicher in einen einzigen Slot zu stecken?
Natürlich mit unserer Multi 4 AT-Slotsparkarte aus deutscher Entwicklung und Fertigung! Sie enthält alle Funktionen dieser 5 Einzelkarten.



Die Multi 4 AT von CE-Infosys ist eine Multifunktionskarte für den IBM XT 286 und alle AT's. Sie beinhaltet bis zu 4 MB RAM-Speicher, aufgebaut in 2 MB-Schritten mit 1 MB DRAM im Extended Memory, ausserdem alle Funktionen der Farbgrafikkarte EGA sowie eine parallele und eine serielle Schnittstelle. Über ein Modul kann ausserdem der Basisspeicher des Rechners auf 640 KB aufgerüstet werden.
Dank modernster SMD-Fertigungstechnik sind alle diese Funktionen auf einer Karte vereint; im Rechner wird nur 1 Slot belegt. In Rechnern bis zu 8 MHz Taktfrequenz läuft die Multi 4 AT ohne «Waitzyklen».
Zum Lieferumfang gehören ein deutsches Handbuch, die Info-PC-Software und kostenlos das Infosys Memory Managerprogramm, das einen Expanded Memory-Bereich unter Verwendung des Extended Memory der Multi 4 AT-Karte simuliert. Programme wie Lotus 1-2-3, Symphony und Framework benutzen den Expanded-Speicher.

Computer Elektronik Infosys GmbH, Am Kümmerling 2, 6501 Bodenheim, Telefon 061 35/30 81
Computer Elektronik Infosys AG, Im Rötel 10a, CH-6301 Zug, Telefon 042/21 78 57
Lager und Vertrieb CH : Oberdorfstrasse 11, CH-8953 Dietikon, Telefon: 01/741 30 41